9. tekmovanje ACM v znanju računalništva

Institut Jožef Stefan, Ljubljana, 29. marca 2014

Bilten

Bilten 9. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2014

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Boris Gašperin, Matija Grabnar, Tomaž Hočevar, Boris Horvat, Nace Hudobivnik, Jurij Kodre, Mitja Lasič, Matjaž Leonardis, Matija Lokar, Mark Martinec, Jure Slak, Mitja Trampuš, Janez Brank.

Tisk: Grafika 3000, d. o. o.
Naklada: 200 izvodov

Ta bilten je dostopen tudi v elektronski obliki na domači strani tekmovanja:
`http://rtk.ijs.si/`

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

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

37.091.27:004(497.4)

TEKMOVANJE ACM v znanju računalništva (9 ; 2014 ; Ljubljana)
Bilten / 9. tekmovanje ACM v znanju računalništva, Ljubljana, 29. marca 2014 ; [avtorji nalog Nino Bašić ... [et al.] ; uredil Janez Brank]. — Ljubljana : Institut Jožef Stefan, 2014

ISBN 978-961-264-080-4
1. Bašić, Nino 2. Brank, Janez, 1979–
276417280

KAZALO

Struktura tekmovanja	5
Nasveti za 1. in 2. skupino	7
Naloge za 1. skupino	11
Naloge za 2. skupino	17
Navodila za 3. skupino	23
Naloge za 3. skupino	27
Naloge šolskega tekmovanja	33
Neuporabljene naloge iz leta 2012	37
Rešitve za 1. skupino	49
Rešitve za 2. skupino	55
Rešitve za 3. skupino	63
Rešitve šolskega tekmovanja	77
Rešitve neuporabljenih nalog 2012	85
Nasveti za ocenjevanje in izvedbo šolskega tekmovanja	131
Rezultati	135
Nagrade	141
Šole in mentorji	142
Off-line naloga: Zlaganje likov	145
Univerzitetni programerski maraton	147
Anketa	150
Rezultati ankete	155
Cvetke	163
Sodelujoče inštitucije	169
Pokrovitelji	173

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 so se zanjo odločili skoraj vsi tekmovalci.

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 24.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C# in java.

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. 23–25 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 145–146.

Podobno kot v zadnjih treh letih smo izvedli tudi šolsko tekmovanje, ki je poteklo 24. januarja 2014. 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. 131–134). 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 277 tekmovalcev s 27 šol.

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).

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. Dnevnik

V računalniškem sistemu imamo zabeležene dogodke in bi jih radi v strnjeni obliki shranjevali na datoteko. Vsak dogodek (kot na primer prijava ali odjava uporabnika, razne napake) je predstavljen z nizom znakov (kratko besedilo / vrstica), ne daljšim od 100 znakov. Dogodki so zapisani na vhodni datoteki, po en dogodek v vsaki vrstici.

Na voljo imamo podprogram (funkcijo) `PreberiDogodek()`, ki prebere naslednji dogodek z vhodne datoteke in ga vrne kot niz znakov. Če smo že na koncu vhodne datoteke in novih dogodkov ni več, ta podprogram vrne prazen niz.

Ker se nekateri dogodki včasih zgodijo večkrat zapored (ne da bi se vmes zgodil kakšen drug dogodek), jih lahko zapišemo na izhodno datoteko v skrajšani obliki: ponovitve zadnjega izpisanega dogodka le štejemo in jih ne izpisujemo. Ko se kasneje pojavi nek drugačen dogodek, izpišemo le, da se je zadnji izpisani dogodek ponovil še n-krat. V primeru, da gre le za eno dodatno ponovitev ($n = 1$, torej skupaj s prvo izpisano pojavitvijo dve pojavitvi, glej zgled), namesto sporočila o ponovitvi izpišemo kar sam ponovljeni dogodek, saj ne bi s sporočilom o ponovitvi nič prihranili.

Tako bi denimo naslednje zaporedje dogodkov:

```
aaa
bbbbb
ccc
ccc
ccc
dd
dd
aaa
aaa
aaa
aaa
```

izpisali v strnjeni obliki kot:

```
aaa
bbbbb
ccc
ponovljeno se 2-krat
dd
dd
aaa
ponovljeno se 3-krat
```

Napiši program, ki bo bral dogodke z vhodne datoteke in jih, takoj ko bo to mogoče, izpisoval v tukaj opisani strnjeni obliki. (Dogodke lahko bereš po svoje s pomočjo standardnih funkcij za delo z datotekami ali pa uporabiš zgoraj omenjeno funkcijo PreberiDogodek.)

2. Proizvodnja čopičev

V Ajdovščini tovarna Wlahna d. o. o. proizvaja krasne veganske bio čopiče, *v celoti* narejene iz lesa. Leseni ročaji so tako ali tako nekaj običajnega, v tej tovarni pa celó konico čopiča izdelajo iz lesa iste vrste, ki ga zmeljejo in predelajo v celulozna vlakna.

V skladišču podjetja imajo n lesenih paličk enake debeline, a različnih dolžin, iz katerih želijo izdelati same enake čopiče. Za posamezen ročaj potrebujejo r centimetrov lesa v enem kosu. Za konico čopiča pa potrebujejo toliko zmletega lesa, kot ga nastane iz k centimetrov ene ali več paličk.

Opiši postopek (ali napiši program, če ti je lažje), s katerim bi ugotovil, kolikšno je največje število čopičev, ki jih podjetje s trenutno zalogo lesa lahko proizvede. Števila n, r in k so podana in so naravna števila. Prav tako so podane dolžine paličk; lahko si recimo predstavljaš, da nekje obstaja tabela (*array*) L, v kateri i-ti element opisuje dolžino i-te paličke v centimetrih (tudi dolžine paličk so naravna števila).

3. Pacifistični generali

Ker imajo vse svetovne vojaške velesile jedrsko orožje, ga moramo nujno imeti tudi pri nas v Sloveniji. Pa je vlada naročila na IJS izdelavo jedrskih konic, od Rusov pa so kupili bojno plovilo VNL-11 Triglav, kjer so smrtonosne rakete zdaj shranjene. Dostopa do tako uničujočega orožja ne sme imeti kdorkoli, ampak ga ima samo peščica najpomembnejših generalov. Ker bi se lahko med generali našel norec, ki bi za zabavo poslal raketo ali dve na katero od sosednjih republik, so se na vojaškem ministrstvu odločili za stroge varnostne ukrepe. Naročili so izdelavo k različic ključev za aktivacijo jedrskih konic. Ključi so oštevilčeni s števili od 1 do k. Vsaka različica ključa je bila izdelana v več izvodih. Te ključe so nato razdelili med n generalov, tako da je vsak prejel neko podmnožico (samih različnih) ključev. Označimo z G_i podmnožico ključev, ki jih ima i-ti general. Skupina generalov lahko sproži atomsko bombo, če imajo vsi skupaj vsaj po eno kopijo vsakega od ključev.

Zgled: recimo, da imamo $k = 3$ ključe in $n = 3$ generale. Naj bo

$$G_1 = \{1, 2\}, G_2 = \{2, 3\}, G_3 = \{1, 3\}.$$

Prvi general ima torej ključ št. 1 in ključ št. 2. Drugi general ima ključ št. 2 in ključ št. 3. Tretji general ima ključ št. 1 in ključ št. 3. Vsak posamezni general ne more aktivirati jedrske konice, če pa se združita npr. prvi in drugi general, imata skupaj $G_1 \cup G_2 = \{1, 2, 3\}$ vse ključe.

Na ministrstvu so pred kratkim opazili fenomen, na katerega pri snovanju sistema sploh niso pomislili. Med generali je vedno več pacifistov, ki niso pod nobenimi pogoji pripravljeni uporabiti jedrskega orožja. Vlada se zdaj boji, da bi lahko manjša skupinica pacifistov ostalim generalom preprečila uporabo orožja. Rekli bomo, da je sistem *r-odporen*, če nobena skupina r (ali manj) generalov ne more ostalim preprečiti aktivacije jerskih konic.

Sistem iz zgleda je 1-odporen, saj nobeden od generalov ne more sam „blokirati“ ostalih dveh.

Opiši postopek (ali napiši program, če ti je to lažje), ki bo za dani r in dani sistem preveril, če je ta sistem r-odporen. Opiši tudi, kako bi tvoja rešitev hranila in organizirala podatke (množice $G_1, \ldots, G_n$ in podobno).

4. Uniforme

Podjetje Wlahna d. o. o. se je odločilo svoje delavce obleči v praktične, trpežne uniforme iz debelega platna, skozi katerega jih ne bodo mogle bosti lesene trske, ki jih je v proizvodni hali podjetja vse polno. Uniforma sestoji iz treh kosov: hlač, jopiča in rokavic. Vsak kos uniforme je dobavljiv v velikostih od 1 do 100.

V podjetje je pravkar prispela nova pošiljka kosov uniform. Ko so jih razkladali s tovornjaka, so sproti popisali vsak kos uniforme (recimo: „hlače velikosti 73“). Zdaj jih zanima, koliko popolnih uniform lahko sestavijo. Popolna uniforma sestoji iz hlač, jopiča in rokavic v enaki velikosti.

Napiši program, ki prebere podatke o razpoložljivih kosih uniforme in izpiše največje število popolnih uniform, ki se jih da sestaviti. Podatke lahko bereš s standardnega vhoda ali pa iz datoteke, kar ti je lažje. Podatki imajo naslednjo obliko: v prvi vrstici je zapisano število dostavljenih kosov n. Vsaka od naslednjih n vrstic vsebuje dve števili, ločeni s presledkom, in opisuje posamezen kos uniforme. Prvo število (1, 2, ali 3) opisuje tip kosa (hlače, jopič, ali rokavice), drugo število (med 1 in 100) pa velikost.

Primer vhoda:

```
15
3 98
1 45
1 74
1 45
2 98
1 45
2 45
1 74
2 74
2 98
2 74
3 74
3 74
1 98
2 74
```

Pripadajoči izpis:

```
3
```

Komentar: pri teh vhodnih podatkih lahko sestavimo tri popolne uniforme (in sicer dve uniformi velikosti 74 in eno uniformo velikosti 98).

5. Davek na ograjo

Slovenski državni urad za odkrivanje novih davkov je ugotovil, da ima vsak Slovenec svoje posestvo omejeno z ograjo in da za to še ni predpisanega nobenega davka. Zato so hitro naročili svojim matematikom, naj pripravijo informativni izračun.

Matematiki so seveda takoj brez škode za splošnost predpostavili, da je Slovenija pravokotna mreža $w \times h$ kvadratkov, kjer vsak kvadratek predstavlja en kvadraten kilometer, in da je vsa zemlja razdeljena med n lastnikov, pri čemer ima vsak kvadratek samo enega lastnika. Nato so oštevilčili vse lastnike zemlje s števili od 0 do $n-1$, vsak kvadratni kilometer zemlje pa so označili s številko lastnika. Torej, kot na skici: polja, označena z 0, pripadajo lastniku 0; polja, označena z 1, pripadajo lastniku 1 in tako naprej.

0	1	1	2	2	2	3
1	1	2	2	4	4	4
5	5	5	0	4	4	4
4	6	6	6	6	4	4
4	4	6	3	6	6	6

Vsa posestva so popolnoma ograjena: ograje stojijo na vseh državnih mejah in povsod tam, kjer kvadratek enega lastnika meji na kvadratek drugega lastnika. Na zgornji sliki so ograje narisane z debelimi črtami.

Davek se zaračuna sorazmerno z dolžino ograje, ki obdaja posestva posameznega lastnika. Ker vsi lastniki trdijo, da ograja spada k njihovemu zemljišču in ne sosednjemu, se jo vsem lastnikom tudi zaračuna — večina ograj se tako šteje dvakrat. **Napiši program**, ki za vsakega lastnika izračuna in izpiše skupno dolžino ograj, od katerih bo moral plačati davek. Obliko izpisa si izberi sam. Predpostavi, da že obstajajo naslednje funkcije, ki vračajo podatke o mreži:

- Visina() vrne h, torej višino mreže (celo število, vsaj 1 in največ 250);
- Sirina() vrne w, torej širino mreže (celo število, vsaj 1 in največ 250);
- StLastnikov() vrne n, torej število različnih lastnikov (največ $w \cdot h$);
- Lastnik(x, y) vrne številko lastnika za celico (x, y); to je celo število od 0 do $n-1$. Pri tem je x številka stolpca (od 0 do $w-1$), y pa številka vrstice (od 0 do $h-1$).

Za primer z gornje slike bi moral tvoj program ugotoviti, da je za lastnika 0 skupna dolžina ograj enaka 8, za lastnika 1 je skupna dolžina 10, za lastnika 2 je skupna dolžina 12, za lastnika 3 je skupna dolžina 8, za lastnika 4 je skupna dolžina 20, za lastnika 5 je skupna dolžina 8, za lastnika 6 pa je skupna dolžina ograj enaka 18.

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).

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. Vnos šifre

Dana je številčna tipkovnica, na kateri so tipke kvadratne oblike in razporejene v pravokotniku podoben lik, kot kaže naslednja slika:

1	2	3
4	5	6
7	8	9
	0	

Radi bi si izbrali neko n-mestno zaporedje števk, ki ga bomo uporabljali kot šifro oz. geslo. Da ga bo čim lažje tipkati, si želimo, da bi se vsak par zaporednih števk v šifri tipkal z isto tipko ali pa s tipkama, ki sta si na gornji tipkovnici sosedi. Pri tem sosednost pomeni, da imata tipki skupno eno od stranic; na primer, v šifri se lahko takoj za števko `4` pojavi števka `1`, `5` ali `7`, ne pa `2` ali `8`. Nekaj primerov veljavnih 6-mestnih šifer: `414558`, `696969`, `089632`.

Napiši podprogram `NastejSifre(n)`, ki kot parameter dobi naravno število n in izpiše vse n-mestne šifre, ki ustrezajo opisani omejitvi. Pri tem je vseeno, v

kakšnem vrstnem redu jih program izpiše, mora pa izpisati vsako primerno šifro natanko enkrat.[1]

Lažja različica naloge: če ti je dosedanja naloga pretežka, lahko rešiš lažjo različico, pri kateri ima n vedno vrednost 6, torej nas zanimajo le 6-mestne šifre. Za rešitev te različice lahko dobiš pri tej nalogi največ 15 točk (od 20 možnih).

2. Prenova ceste

V bližini mesteca Cocklebiddy v Zahodni Avstraliji se nahaja najdaljša popolnoma ravna cesta na svetu. Zaradi nedavnih poplav morajo prenoviti 100 km ceste. Za popravilo ceste kandidirata dve podjetji, Kangaroads Ltd. in Wallabyway Inc. Vzdolž cestnega odseka, ki ga je treba popraviti, živi 1 000 000 prebivalcev[2] in vsak od njih ima svoje mnenje o tem, katero podjetje bi moralo popravljati cesto.

Na koncu so se prebivalci odločili za kompromis: cesto lahko popravljata obe podjetji, pri čemer bo vsak še tako majhen košček ceste popravilo tisto podjetje, za katerega glasuje večina od k najbližje stanujočih prebivalcev; k je liho število. (Za namen te naloge je cesta daljica, stanovanja prebivalcev pa so točke na njej.)

Opiši postopek, ki izračuna, koliko kilometrov ceste bo popravilo podjetje Kangaroads Ltd. in koliko Wallabyway Inc. Predpostavi, da že obstajajo naslednje spremenljivke: `k`, kot je opisan zgoraj; tabela `x`, kjer je `x[i]` realno število med 0 in 100 in opisuje položaj i-tega prebivalca (oddaljenost od zahodnega konca popravljanega odseka ceste, v kilometrih), ter tabela `p`, kjer je `p[i]` enak 0, če i-ti prebivalec glasuje za podjetje Kangaroads Ltd., in 1 sicer. Vse koordinate prebivalcev so različne in so podane v naraščajočem vrstnem redu.

3. Skrivno sporočilo

Si vohun SOVE, ki dela pod krinko in ravnokar si „opravil“ z agentom zlobne zločinske organizacije OREL, ki je poskušal poslati podatke o času in kraju dostave pomembnega paketa svojim nadrejenim. Pri nakazovanju svoje superiornosti si bil malce pregrob, tako da agenta sedaj ne moreš izpraševati, temveč nemočno držiš v rokah njegovo komunikacijsko napravo, kjer se na zaslonu bleščita sporočilo, ki ga je agent želel poslati, in njegova napol zašifrirana kopija. Tvoja želja je, da v njegovem imenu pošlješ svoje sporočilo, ki bo seveda vsebovalo napačen kraj in čas dostave, da boste lahko ne le varno prejeli paket, temveč tudi ujeli še kakšnega agenta.

Agenti ORLA svoja sporočila šifrirajo zelo primitivno, črke abecede le malo zamešajo med seboj in pišejo namesto `a` na primer `r` in podobno. Tvoja naloga je, da preveriš, ali je delno zašifrirano sporočilo veljavno, in čim bolj ugotoviš šifro ter na enak način, kolikor je le mogoče, zašifriraš svoje sporočilo. Zašifirano sporočilo je veljavno, če se enaki črki vedno zašifirirata v enaki črki, poleg tega pa se morata različni črki šifrirati v različni črki, da je sporočilo mogoče odkodirati. (Bolj matematično: kodirna funkcija mora biti bijektivna). Primera neveljavnih kodiranj sta `cat` → `bgb` in `zoo` → `srt`, primer veljavnega kodiranja pa je npr. `please` → `rlagha`.

[1] Zanimivo, vendar težjo različico naloge dobimo, če želimo le izračunati, koliko je vseh n-mestnih šifer, ne da bi jih pri tem tudi vse naštevali.

[2] Poplave in kakšnih 999 950 prebivalcev je izmišljenih; tisto o najdaljši ravni cesti je pa res, dolga je 90 milj.

Napiši podprogram (funkcijo) `Desifriraj(p1, c1, p2)`, ki kot parametre dobi tri nize:

- `p1` je originalno (nešifrirano) sporočilo.

- `c1` je delno šifrirano sporočilo, ki smo ga zasegli sovražnemu agentu. Ta niz je enako dolg kot originalno sporočilo. V njem nastopajo že zašifrirane črke; na mestih, ki jih agent še ni utegnil zašifrirati, pa je namesto črke zvezdica (znak `*`).

- `p2` je tvoje sporočilo, ki ga želiš zašifrirati po enakem postopku, kot ga je uporabil sovražni agent.

Vsa sporočila vsebujejo samo male črke angleške abecede in so krajša od 10000 znakov. Tvoj podprogram naj izpiše zašifrirano različico tvojega sporočila `p2` (če za nekatere znake ni mogoče zanesljivo ugotoviti, v kaj bi se morali zašifrirati, namesto njih izpiši zvezdico `*`); če pa dano šifrirano sporočilo ni veljavno, izpiši „`neveljavna šifra`“.

Primer: recimo, da dobimo nize

p1 = `nexttuesdayfourfiftypminthemainsquare`
c1 = `r**aa**k***s****e***q*e**o********wl*`
p2 = `thistuesdayfourfiftypmanddefinetlynotnexttuesdayfourfiftypminthemainsquare`

Pravilni izpis je potem takšen:

```
aoeka**k*w*s**lsesa*q*wr***ser*a**r*ar**aa**k*w*s**lsesa*q*erao**werk**wl*
```

4. Potenciranje

Predstavljajmo si zelo preprost, zbirniku podoben programski jezik. Program je sestavljen iz zaporedja ukazov; pred vsakim ukazom je lahko še oznaka (labela), na katero se lahko sklicujemo pri pogojnih skokih. Dovoljeni ukazi so naslednji:

- ADD x, y — izračuna vsoto $x + y$ in jo shrani v x;
- SUB x, y — izračuna razliko $x - y$ in jo shrani v x;
- MUL x, y — izračuna zmnožek $x \cdot y$ in ga shrani v x;
- DIV x, y — izračuna celi del količnika x/y in ga shrani v x;
- MOD x, y — izračuna ostanek po deljenju x z y in ga shrani v x;
  Opomba: pri gornjih ukazih mora biti x spremenljivka, y pa je lahko spremenljivka ali celoštevilska konstanta.

- JL x, y, z — pogojni skok: če je $x < y$, skoči na ukaz z oznako (labelo) z. Pri tem sta x in y lahko spremenljivki in/ali celoštevilski konstanti. Če pogoj $x < y$ ni izpolnjen, se izvajanje nadaljuje z naslednjim ukazom.

Program lahko uporablja poljubno mnogo spremenljivk. Vse spremenljivke so celoštevilske (kot tip **int** oz. integer, le da lahko za razliko od teh tipov hranijo poljubno velika cela števila) in jih pred uporabo ni treba posebej deklarirati. Ukazi se izvajajo po vrsti, razen če pride do pogojnega skoka (ukaz JL).

V tem programskem jeziku **napiši program**, ki izračuna vrednost a^b in jo shrani v spremenljivko c. Pri tem je b naravno število. (Predpostavi, da imata spremenljivki a in b želeni začetni vrednosti že pred začetkom izvajanja tvojega programa, torej se ti ni treba ukvarjati s tem, kako bi ju prebral ali incializiral.) Primer: če je pred začetkom izvajanja tvojega zaporedja ukazov v spremenljivki a vrednost 2, v spremenljivki b pa vrednost 3, mora biti na koncu izvajanja tvojega programa v spremenljivki c vrednost 8. Tvoj program naj pri tem izvede čim manj ukazov za velika števila a in b.

Svojo rešitev tudi dobro utemelji in komentiraj, kako in zakaj deluje.

Spomnimo se, da je b-ta potenca števila a definirana takole:

$$a^b = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ členov}}$$

in da za potence med drugim velja

$$a^{b+c} = a^b \cdot a^c.$$

Namig: najprej razmisli, kako bi učinkovito računal a^2, a^4, a^8, a^{16} itd., nato pa še o tem, kako bi s tem prišel do rešitve za poljuben b.

Za ilustracijo je tule primer programa, ki rešuje malo drugačen problem: v spremenljivki vsota izračuna vsoto števil od 1 do n (ob predpostavki, da je n večji od 0):

		Razlaga (ni del programa)
	SUB vsota, vsota	postavi *vsota* na 0
lab1:	ADD vsota, n	prišteje vsoti trenutno vrednost n
	SUB n, 1	zmanjša n za 1
	JL 0, n, lab1	ponavlja, dokler je $n > 0$

Pa še en primer: spodnji program računa isto vsoto, vendar namesto zanke uporabi formulo $1 + 2 + \ldots + n = n \cdot (n + 1)/2$.

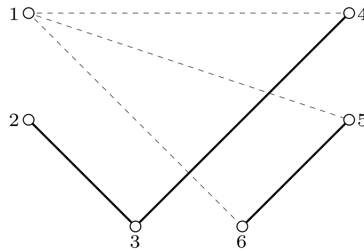
	Razlaga (ni del programa)
SUB vsota, vsota	postavi *vsota* na 0
ADD vsota, n	*vsota* je zdaj enaka n
ADD vsota, 1	*vsota* je zdaj enaka $n + 1$
MUL vsota, n	*vsota* je zdaj enaka $n(n + 1)$
DIV vsota, 2	*vsota* je zdaj enaka $n(n + 1)/2$

5. Tiskana vezja

Ker se Štefko zanima za elektroniko, je za rojstni dan dobil začetniški komplet za izdelavo tiskanih vezij. Paket vsebuje več plošč, ki že imajo narejene luknjice za priključke. Poleg tega je dobil orodje, s katerim lahko med poljubnima dvema luknjicama (priključkoma) nariše *ravno* (bakreno) povezavo. Luknjice so postavljene na ploščo na tak zvit način, da nobena ravna črta med dvema luknjicama ne prečka katere druge luknjice. Štefko je že pripravil načrte za nekaj genialnih vezij, ko pa jih je hotel narisati, je opazil — ojoj! — da bi se nekatere daljice med seboj sekale,

če bi jih zares narisal na ploščo. Ko je vezje načrtoval, je mislil samo na to, kateri pari priključkov morajo biti med seboj povezani.

Ampak ni še vse izgubljeno. Štefko je kmalu opazil, da je plošča dvostranska! Razmišljal je takole: kaj pa, če nekaj povezav narišem na eno, nekaj pa na drugo stran plošče? Potem morda lahko dosežem, da se nobeni dve povezavi ne bosta sekali? Zdaj ga zanima, katera vezja je možno narisati, če lahko uporabi obe strani plošče. Ker je problem za Štefka preveč zapleten, potrebuje tvojo pomoč. **Opiši postopek**, ki ugotovi, ali je mogoče dano vezje narisati na opisani način, ne da bi se kakšni dve povezavi sekali. Pri tem tvoj postopek kot vhodne podatke dobi koordinate vseh luknjic (luknjice oštevilčimo od 1 do n in recimo, da ima i-ta luknjica koordinate (x_i, y_i)) in seznam parov luknjic, ki jih je treba povezati. Opiši tudi, kako bi v svoji rešitvi predstavil in organiziral te podatke v pomnilniku. Predpostavi, da že obstaja funkcija SeSekata(ax, ay, bx, by, cx, cy, dx, dy), ki vrne **true**, če se daljica od točke (a_x, a_y) do točke (b_x, b_y) seka z daljico od točke (c_x, c_y) do točke (d_x, d_y), sicer pa vrne **false**.



Primer: gornja slika kaže vezje, ki bi se ga dalo narisati na opisani način; pri tem debele polne črte pomenijo povezave na eni strani plošče, tanke črtkane črte pa povezave na drugi strani plošče. Če pa bi hoteli temu vezju dodati še povezavo med luknjicama 2 in 5, se ga ne bi več dalo narisati brez križanja povezav.

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 imenik `U:_Osebno`, v katerem lahko kreiraš svoje datoteke. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C# ali java, mi pa jih bomo preverili s 64-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz OpenJDK 1.7 in s prevajalnikom Mono 4 za C#. Za delo lahko uporabiš `FP` oz. `ppc386` (Free Pascal), `gcc/g++` (GNU C/C++ — command line compiler), `javac` (za javo 1.7), Visual Studio in druga orodja.

Na spletni strani `http://rtk/` oz. `http://rtk.std.fmf.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`

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 desetih testnih primerih. 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 desetih testnih primerih; pri vsakem od njih dobi 10 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://rtk/`):

- 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();
   }
}
```

NALOGE ZA TRETJO SKUPINO

1. Ljudožerci na premici (`ljudozerci.in`, `ljudozerci.out`)

Vzdolž premice stoji n ljudožercev, ki so oštevilčeni od 1 do n, vendar ne nujno v kakšnem posebnem vrstnem redu (npr. od leve proti desni ali kaj podobnega). Ljudožerec s številko i se nahaja na koordinati x_i (to je celo število, večje ali enako 0). Te koordinate niso nujno različne; lahko se zgodi, da dva ali več ljudožercev stoji na isti točki.

En za drugim se na premico s padalom spusti še m ljudi. Pri tem j-ti od njih naredi naslednje:

- Pristane na neki koordinati a_j.
- Pomaha najbližjemu ljudožercu (če je takšnih več, izbere tistega z manjšo koordinato; če je tudi takšnih več, pa tistega z najmanjšo zaporedno številko).
- Ljudožerec mu pomaha nazaj, pride k njemu (padalec ga počaka na točki a_j) in ga požre.

Napiši program, ki prebere začetne koordinate ljudožercev ter zaporedje prihodov padalcev, nato pa izpiše končne koordinate ljudožercev.

Vhodna datoteka: v prvi vrstici sta števili n in m, ločeni s presledkom. Sledi n vrstic; i-ta od njih vsebuje koordinato x_i, na kateri na začetku stoji posamezen ljudožerec. Sledi m vrstic; j-ta od njih vsebuje koordinato a_j, na katero se spusti j-ti padalec. Veljalo bo $1 \leq n \leq 10^5$, $0 \leq m \leq 10^5$ in $0 \leq x_i \leq 10^9$, $0 \leq a_i \leq 10^9$.

Izhodna datoteka: izpiši končne koordinate vseh n ljudožercev v naraščajočem vrstnem redu, vsako v svojo vrstico.[3]

Primer vhodne datoteke:

```
6 4
30
10
50
40
30
20
22
35
37
6
```

Pripadajoča izhodna datoteka:

```
6
22
30
37
40
50
```

[3] Zanimivo, vendar malo težjo različico naloge dobimo, če zahtevamo, da morajo biti ljudožerci v izpisu urejeni po zaporednih številkah, ne po koordinatah: program naj torej najprej izpiše koordinato ljudožerca številka 1, nato koordinato ljudožerca številka 2 in tako naprej.

2. Po Indiji z avtobusom (`avtobus.in`, `avtobus.out`)

Čez celotno Indijo poteka v smeri zahod–vzhod dolga cesta. Poljuben kraj ob cesti lahko opišemo s številom kilometrov, ki jih je treba prepotovati od zahodnega konca ceste, da pridemo do tega kraja. Vzdolž ceste vozi n avtobusov, i-ti od njih med krajema a_i in b_i v obe smeri. Avtobusi peljejo zelo počasi in imajo ves čas odprta vrata, da se vsaj malo zračijo; tako je možno na poljubni točki izstopiti iz avtobusa ali vstopiti nanj. Praveen želi potovati od mesta x do mesta y. Pomagaj mu najti potovalni načrt, s katerim bo uporabil čim manj avtobusov.

Vhodna datoteka: v prvi vrstici so po vrsti zapisana cela števila n, x in y, ločena s po enim presledkom. Vsaka od naslednjih n vrstic vsebuje števili a_i in b_i za posamezen avtobus; to sta celi števili, ločeni z enim presledkom. Veljalo bo $1 \leq n \leq 200\,000$, $0 \leq x < y \leq 10^9$ in (za vsak i) $0 \leq a_i < b_i \leq 10^9$.

Vsi primeri so taki, da rešitev obstaja (torej da je res mogoče priti od kraja x do kraja y).

V 50% testnih primerov bo veljalo tudi $n \leq 1000$.

Izhodna datoteka: izpiši najmanjše število avtobusov, s katerimi je možno prepotovati pot med x in y.

Primer vhodne datoteke:

```
6 3 21
10 20
4 11
2 6
11 21
1 4
6 12
```

Pripadajoča izhodna datoteka:

```
3
```

Komentar: lahko se na primer z avtobusom 2–6 peljemo od 3 do 6, nato z avtobusom 6–12 do 11 in nato z avtobusom 11–21 do 21. Možni pa so še tudi drugi poteki potovanja, ki ravno tako pridejo do cilja s tremi avtobusi.

3. Luči (`luci.in`, `luci.out`)

Znašel si se v sobi z L lučmi in S stikali. Po daljšem preklapljanju stikal si ugotovil, da je vsako stikalo povezano z nekaj lučmi (lahko tudi z nobeno ali vsemi), pritisk nanj pa spremeni stanje luči, s katerimi je povezano — če je posamezna luč ugasnjena, se prižge in obratno. Ugotovil si tudi, da lahko na vsako luč vplivaš z največ dvema različnima stikaloma. Na vsak način bi rad ugasnil vse luči, zato si se naloge lotil sistematično in zabeležil, katera stikala so povezana s katerimi lučmi. Na koliko načinov lahko ugasneš vse luči? Ker je vsako stikalo smiselno uporabiti največ enkrat, izračunaj število podmnožic stikal, s pritiskom na katera ugasnemo vse luči.

Vhodna datoteka: v vhodni datoteki je več testnih primerov. Število primerov T je podano v prvi vrstici, sledijo pa s praznimi vrsticami ločeni opisi posameznih testnih primerov. Vsak testni primer se začne z vrstico, ki vsebuje število luči L in število stikal S, ločeni s presledkom. V naslednji vrstici se nahaja L števil 0 ali 1 (ločena so s po enim presledkom). Če je luč k trenutno ugasnjena, bo k-to število enako 0, sicer pa 1. Sledi še S vrstic, ki opisujejo povezave stikal z lučmi. Prvo število v i-ti vrstici predstavlja število luči n_i, s katerimi je povezano stikalo i. V nadaljevanju vrstice je podanih še n_i zaporednih številk luči, na katere vpliva to stikalo.

Veljalo bo $1 \leq T \leq 10$, $1 \leq L \leq 300$ in $1 \leq S \leq 300$. V 30 % testnih primerov bo veljalo tudi $L \leq 20$ in $S \leq 20$.

Izhodna datoteka: za vsak testni primer izpiši po eno vrstico, vanjo pa izpiši celo število, ki pove, na koliko načinov lahko pri tem testnem primeru s stikali, ki so nam na voljo, ugasnemo vse luči. Ker je to število lahko zelo veliko, izpiši le njegov ostanek pri deljenju z 1 000 000 007 ($10^9 + 7$).

Primer vhodne datoteke:

```
2

7 5
0 1 1 1 1 1 0
2 1 3
2 1 2
2 2 3
0
3 4 5 6

4 2
0 1 0 0
4 1 2 3 4
2 2 3
```

Pripadajoča izhodna datoteka:

```
4
0
```

Komentar: možni nabori stikal pri prvem testnem primeru so $\{1, 2, 5\}$, $\{1, 2, 4, 5\}$, $\{3, 5\}$ in $\{3, 4, 5\}$.

4. Bloki (`bloki.in`, `bloki.out`)

Mnogi urejevalniki besedil imajo ukaz, ki nas z znaka „`{`“ premakne na pripadajoči „`}`“ ali obratno; to pride prav pri urejanju izvorne kode v tistih programskih jezikih, ki z zavitimi oklepaji označujejo začetek in konec bloka oz. skupine stavkov (npr. zanke, funkcije ipd.). Kaj pa, če delamo v jeziku, ki zavitih oklepajev ne uporablja, pač pa pripadnost blokom izraža z zamikanjem vrstic (na primer python)?

Definirajmo *zamik vrstice* kot število presledkov na začetku vrstice. Predpostavi, da se drugih presledkom podobnih znakov (npr. tabulatorjev) ne uporablja. Vrstico, ki vsebuje same presledke, štejemo za prazno.

Blok je skupina več zaporednih vrstic, določena z naslednjimi pravili:

- V neki vrstici se začne blok, če je ta vrstica neprazna in
  - je to prva neprazna vrstica sploh ali pa
  - ima večji zamik kot prejšnja neprazna vrstica (pri tem ni pomembno, koliko praznih vrstic je med njima).
- Ta blok se potem nadaljuje do prve take neprazne vrstice, ki ima manjši zamik kot vrstica, s katero se je blok začel; ta že ne pripada več bloku (tista pred njo pa še, četudi je prazna). Če take vrstice ni, se blok nadaljuje do konca vhodne datoteke.

Ta definicija omogoča, da se bloki gnezdijo eden v drugem in da posamezna vrstica pripada več blokom. Nekaj primerov (bloki so označeni z oglatimi oklepaji na levi strani; presledki so prikazani z znakom ␣, da se jih bolje vidi):

```
In␣the␣second
␣␣century␣of␣the
␣␣Christian␣Aera,
␣␣␣␣the␣empire␣of
␣␣Rome␣comprehended
␣␣␣␣the␣fairest
part␣of␣the
␣␣earth,␣and␣the
␣␣most␣civilized
```

```
␣␣portion␣of
␣␣␣␣␣␣mankind.␣The
␣␣␣␣␣␣frontiers␣of
␣␣␣␣that␣extensive
monarchy␣were
␣␣guarded␣by
␣␣ancient␣renown
```

```
␣␣
and␣disciplined

␣␣valor.␣The
␣␣gentle␣but
␣␣␣␣
powerful
␣␣influence␣of
␣
␣␣␣␣laws␣and

manners␣had
␣␣␣
```

Kot vidimo iz drugega in tretjega primera, se lahko zgodi tudi, da kakšna vrstica ne pripada nobenemu bloku.

Napiši program, ki prebere besedilo, poišče v njem vse bloke in za vsak blok izpiše, v kateri vrstici se začne in v kateri se konča.

Vhodna datoteka: v prvi vrstici je celo število n, ki pove, koliko vrstic je dolgo vhodno besedilo. Veljalo bo $1 \leq n \leq 10^5$. Sledi n vrstic z besedilom, ki ga moraš obdelati. Vsaka vrstica je dolga največ 1000 znakov.

Izhodna datoteka: za vsak blok izpiši po eno vrstico, vanjo pa dve celi števili, ločeni s presledkom. Prvo od teh števil naj bo številka prve vrstice tega bloka, drugo pa številka zadnje vrstice tega bloka. Vrstice so oštevilčene od 1 do n. Bloke izpiši v naraščajočem vrstnem redu glede na številko začetne vrstice.

Primer vhodne datoteke:

```
10
gradually cemented the union
  of the provinces. Their peaceful
  inhabitants enjoyed and abused the
    advantages of wealth and luxury.
    The image of a free constitution
  was preserved with decent reverence:
the Roman senate appeared to possess
    the sovereign authority, and
    devolved on the emperors all the
      executive powers of government.
```

Pripadajoča izhodna datoteka:

```
1 10
2 6
4 5
8 10
10 10
```

5. Poravnavanje desnega roba (`poravnavanje.in`, `poravnavanje.out`)

Besedilo, ki se razteza čez več vrstic, je videti lepše, če je njegov desni rob poravnan — z drugimi besedami, v krajših vrsticah presledke malo razširimo, da so na koncu videti vse vrstice enako dolge (razen zadnje vrstice besedila; ta je lahko krajša od ostalih). Po drugi strani pa si ne želimo, da bi v kakšni vrstici nastale prevelike vrzeli med besedami. Zato se pri razbijanju besedila na vrstice včasih splača dati v kakšno vrstico manj besed, kot bi jih sicer lahko šlo vanjo, če nam to omogoči lepše razbiti preostanek besedila.

Recimo, da je naše besedilo dolgo n besed in da so znane tudi širine vseh teh besed, $w_1, w_2, \ldots, w_n$ (pri čemer je w_i širina i-te besede). Podana je tudi minimalna zahtevana širina presledka med besedami; recimo ji s. Presledki se pojavijo le med besedami, ne pa na začetku ali koncu vrstice. Predpisana je tudi zahtevana širina vrstice po poravnavanju desnega roba; recimo ji d.

Definirajmo zdaj *oceno vrstice* takole: recimo, da vrstica vsebuje besede od vključno i-te do vključno j-te. To je torej skupaj $j - i + 1$ besed, med katerimi mora biti zato $j - i$ presledkov; vsak od teh presledkov mora biti širok vsaj s. Skupna dolžina besed in teh minimalnih presledkov je torej $w_i + w_{i+1} + \ldots + w_j + (j-i) \cdot s$; če ta vsota presega d, potem take vrstice sploh ne smemo uporabiti, ker bi bila preširoka. Drugače pa vidimo, da bo treba to vrstico razširiti za $d - (w_i + w_{i+1} + \ldots + w_j + (j-i) \cdot s)$, da bo dosegla predpisano širino d. Za oceno vrstice vzemimo kvadrat tega števila:

$$\mathit{ocena}(i, j) = (d - (w_i + w_{i+1} + \ldots + w_j + (j - i) \cdot s))^2 .$$

Izjema nastopi na koncu besedila: ker pri zadnji vrstici ne poravnavamo desnega roba, vanjo tudi ne vrivamo dodatnih presledkov, zato vzamemo $\mathit{ocena}(i, n) = 0$.

Dano zaporedje besed želimo razbiti na vrstice tako, da nobena vrstica ne presega širine d in da je vsota njihovih ocen najmanjša možna. **Napiši program**, ki izračuna najmanjšo možno vsoto ocen vrstic, ki jo je mogoče doseči na ta način.[4]

Vhodna datoteka: v prvi vrstici so cela števila n, s in d, ločena s po enim presledkom. Sledi n vrstic, ki po vrsti podajajo cela števila $w_1, w_2, \ldots, w_n$. Velja $1 \leq n \leq 10^6$, $1 \leq s \leq d \leq 1000$ in (za vsak i) $1 \leq w_i \leq d$.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjšo možno vsoto ocen vrstic, ki jo je mogoče doseči v skladu z zahtevami naloge.

Primer vhodne datoteke:

```
6 2 30
9
9
3
10
10
18
```

Pripadajoča izhodna datoteka:

```
89
```

Komentar primera: najboljše razbitje besed na vrstice je, da vzamemo prve tri besede v prvo vrstico, naslednji dve v drugo in zadnjo besedo v tretjo vrstico. Širina treh besed v prvi vrstici, skupaj z minimalnim razmikom ($s = 2$) med vsakima zaporednima besedama, je $9 + 2 + 9 + 2 + 3 = 25$, torej nam do $d = 30$ manjka še 5, zato je ocena te vrstice enaka $5^2 = 25$. Podobno je drugo vrstica z minimalnim razmikom široka $10 + 2 + 10 = 22$, torej ji do d manjka še 8, zato je ocena te vrstice enaka $8^2 = 64$. Ocena zadnje vrstice je po definiciji 0. Končni rezultat je zato $25 + 64 + 0 = 89$.

[4] Zanimivo vprašanje v zvezi s to nalogo je tudi naslednje: ali je tu opisani postopek za deljenje besedila na vrstice že dovolj dober, da bi bil uporaben v praksi? Kaj bi bilo treba v njem še spremeniti?

NALOGE ZA ŠOLSKO TEKMOVANJE

24. januarja 2014

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. Ocenjevanje profesorjev

Na neki šoli so izvedli anketo, v kateri so dijaki ocenjevali profesorje. Na anketo dijaki anonimno napišejo njim najboljše in najslabše tri profesorje. Ocene profesorjev se nato izračuna na sledeč način: najboljši profesor prejme 3 točke, drugi najboljši dve točki in tretji najboljši eno točko. Podobno dobi najslabši profesor -3 točke, drugi najslabši -2 in tretji najslabši -1 točko. **Napiši program**, ki za vsakega profesorja izračuna vsoto točk, ki jih je na ta način dobil, in izpiše, koliko profesorjev je dobilo negativno število točk in kateri profesor je dobil najnižje število točk.

Namesto z imeni in priimki so profesorji v teh anketah predstavljeni z zaporednimi številkami od 1 do n, dijaki pa z zaporednimi številkami od 1 do m (imamo torej n profesorjev in m dijakov). Predpostavi, da že obstajajo naslednje funkcije, ki naj jih tvoj program kliče, da pride do podatkov iz anket.

- `StProfesorjev()` — vrne n, torej število profesorjev (vsaj 1, največ 10 000);
- `StDijakov()` — vrne m, torej število dijakov (vsaj 1, največ 100 000);
- `Ocena(d, p)` — vrne število od -3 do 3, ki pove, kakšno oceno je dal dijak d profesorju p; če ta profesor ni eden od tistih šestih, ki jih je dijak d omenil v svoji anketi, vrne funkcija vrednost 0;
- `Komu(d, t)` — vrne številko profesorja, ki mu je dijak d namenil t točk; pri tem mora biti t eno od števil $-3, -2, -1, 1, 2, 3$.

Tvoj program naj izpiše, koliko profesorjev ima v skupnem seštevku negativno število točk, nato pa še številko profesorja, ki ima v skupnem seštevku najnižje število točk. Ker je profesorjev in dijakov veliko, je zaželeno, da je tvoja rešitev čim bolj učinkovita (učinkovitejše rešitve dobijo več točk).

2. Spraševanje

Imamo n učencev in za vsakega vemo, kolikokrat je bil že vprašan (to so cela števila $v_1, \ldots, v_n$, pri čemer nam v_i pove, kolikokrat je bil vprašan učenec i). Obstaja tudi pravilo, da učenca i ne smemo vprašati še enkrat, če obstaja kak drug učenec j, za katerega je $v_i \geq v_j + d$. Konstanta d je podana vnaprej in je majhna v primerjavi z n.

Opiši, kako bi v programu predstavil te podatke, da bi lahko na njih čim učinkoviteje izvajal naslednji dve operaciji:

- Vprasan(i) — mora povečati v_i za 1;
- Koliko() — mora vrniti število učencev, ki jih v tem trenutku ne smemo vprašati (če nočemo prekršiti zgoraj omenjenega pravila).

Opiši tudi, kako bi implementiral ti dve operaciji. Svoje podatkovne strukture inicializiraj tako, kot da so na začetku vsi v_i enaki 0. Predpostavi, da se operacija Vprasan(i) vedno kliče le s takim i, ki ga je mogoče vprašati, ne da bi bilo prekršeno zgoraj omenjeno pravilo.

Zaželeno je, da je tvoja rešitev čim bolj učinkovita (učinkovitejše rešitve dobijo več točk), torej da deluje hitro tudi za velike vrednosti n.

3. Žica

Peter se pri pouku dostikrat dolgočasi. Danes mu je med poukom fizike pod roko prišla dolga, ravna in enako debela žica. Odločil se je, da bo iz te žice naredil umetnino: žico bo na poljubnih mestih prepognil za pravi kot v levo ali desno, tako da bo vsa žica zvita le v dvodimenzionalni ravnini. **Napiši program**, ki prebere podatke o prepogibih žice in izračuna, ali je tako prepognjena žica sklenjena (torej ali se konča v isti točki, kjer se je začela). Tvoj program lahko bere podatke s standardnega vhoda ali pa iz datoteke `zica.txt` (kar ti je lažje).

Vhodni podatki: v prvi vrstici je število prepogibov žice (n), naslednjih n vrstic pa vsebuje po dva podatka, s in d (ločena z enim presledkom), pri čemer je s eden od znakov „`L`“ ali „`D`“, ki pove smer prepogiba (levo ali desno), naravno število d pa je razdalja od prejšnjega prepogiba. (Smer prepogiba v zadnji vrstici je sicer nepomembna, saj se žica tam, kjer bi moral biti ta prepogib, konča, tako da prepogiba sploh ni. Smer je v tej vrstici navedena le zato, da lahko tvoj program to vrstico obdela na enak način kot prejšnje vrstice.)

Izhodni podatki: kot rezultat izpiši „`Da`“, če je žica sklenjena, drugače pa izpiši „`Ne`“.[5]

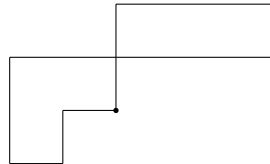
Primer vhodne datoteke:

```
8
D 2
D 3
D 1
L 5
L 2
L 1
D 1
D 1
```

Pripadajoči izpis:

```
Da
```

Za ilustracijo si oglejmo še sliko žice s tega primera (pika označuje točko, kjer se žica začne in konča):



[5] Zanimiva, vendar malo težja različica naloge je tudi naslednja: recimo, da na žico postavimo koordinatni sistem tako, da je točka, v kateri se žica začne, na koordinatah $(0, 0)$ in da žica tam kaže v smeri pozitivne y-osi; kot vhodne podatke dobimo po vrsti koordinate vseh prepogibov žice in na koncu še koordinate točke, v kateri se žica konča. Iz tega podatkov bi radi rekonstruirali takšen opis žice, kot smo ga v prvotni različici naloge dobili na vhodu, torej zaporedje parov $\langle$smer, razdalja$\rangle$.

4. Račja

Mama raca plava po ribniku, za njo pa v ravni vrsti n malih račk; „`GA ga ga`“, reče mama raca. Račka številka 1, ki plava tik za njo, ponovi za njo, a ker ni poslušala čisto natančno, se zmoti v prvem in tretjem zlogu: „`ga ga GA`“, reče. Račka številka 2 mame race ne sliši več, ker je predaleč, sliši pa račko tik pred seboj, zato ponovi za njo; zmoti se v prvem zlogu: „`GA ga GA`“. Tako se nadaljuje vse do n-te račke: vsaka ponovi oglašanje račke pred seboj in se pri tem morda zmoti v kakšnem od zlogov. Edina zloga, ki ju race uporabljajo, sta „`ga`“ in „`GA`“. Nobena raca se ne zmoti v številu zlogov; to ostane enako vse od mame race do n-te račke. (Ne smeš pa predpostaviti, da je število zlogov nujno enako tri kot v zgornjem primeru; predpostavi le, da je število zlogov vsaj 1 in največ 100.)

Napiši program, ki prebere oglašanje vseh račk po vrsti in izpiše zaporedno številko najbolj neposlušne račke, torej tiste, ki se po oglašanju od svoje predhodnice razlikuje v največjem številu zlogov. Če je po tem kriteriju več račk izenačenih, izpiši tisto z najnižjo zaporedno stevilko. Račke številčimo s števili od 1 naprej.

Tvoj program lahko bere podatke s standardnega vhoda, lahko pa iz datoteke `racke.txt` (kar ti je lažje). V prvi vrstici je število n, v drugi vrstici je oglašanje mame race, nato pa sledi še n vrstic, ki po vrsti navajajo oglašanje račk. Zlogi v posamezni vrstici so vedno ločeni s po enim presledkom; pred prvim in za zadnjim zlogom v posamezni vrstici ni nobenega presledka ali kakšnih drugih znakov.

Primer vhodne datoteke:

```
4
GA ga ga
ga ga GA
GA ga GA
GA ga GA
ga GA ga
```

Pripadajoči izpis:

```
4
```

Komentar: prva račka se je zmotila v dveh zlogih (prvem in tretjem), druga v enem (namreč v prvem), tretja v nobenem, četrta pa v vseh treh. Pravilni odgovor je torej 4, ker se je največkrat zmotila četrta račka.

Lažja različica naloge: če ti je naloga pretežka, lahko predpostaviš, da že obstaja funkcija `Primerjaj(s, t)`, ki ji kot parametra podaš niza s in t, ki opisujeta oglašanje dveh rac, funkcija pa vrne število zlogov, v katerih se niza razlikujeta. S to dodatno predpostavko lahko dobiš pri tej nalogi največ 10 točk.

5. Kraljice

Imamo šahovnico nestandardne velikosti: namesto 8×8 polj jo sestavlja $n \times n$ polj. Na njej stoji k šahovskih kraljic; za vsako poznamo njen položaj, torej par koordinat (x_i, y_i), ki nam pove, da i-ta kraljica stoji na preseku vrstice y_i in stolpca x_i (vrstice in stolpci so oštevilčeni s celimi števili od 1 do n). Za dve kraljici rečemo, da se napadata, če ležita v isti vrstici, stolpcu ali diagonali in med njima (v tej vrstici, stolpcu ali diagonali) ne leži še kakšna tretja kraljica.

Opiši postopek, ki iz teh podatkov (torej n, k in koordinat vseh kraljic) izračuna, koliko parov kraljic se napada med sabo. Kraljic je največ 10 000, dolžina

stranice n pa je največ 1 000 000. Predpostavi, da se ne more zgoditi, da bi več kraljic stalo na istem polju. Tvoja rešitev naj bo čim bolj učinkovita; bolj učinkovite rešitve dobijo več točk. Ni ti treba izpisati, kateri pari kraljic se napadajo med sabo, pač pa le to, koliko je takih parov.

Primer: recimo, da imamo $n = 7$, $k = 5$ in da stojijo kraljice na poljih $(3, 2)$, $(1, 4)$, $(5, 4)$, $(7, 4)$ in $(6, 5)$. Potem je parov kraljic, ki se med seboj napadajo, 6:

- napadata se kraljici na poljih $(1, 4)$ in $(5, 4)$;
- napadata se kraljici na poljih $(5, 4)$ in $(7, 4)$;
- napadata se kraljici na poljih $(1, 4)$ in $(3, 2)$;
- napadata se kraljici na poljih $(3, 2)$ in $(5, 4)$;
- napadata se kraljici na poljih $(5, 4)$ in $(6, 5)$;
- napadata se kraljici na poljih $(6, 5)$ in $(7, 4)$.

Po drugi strani pa se na primer kraljici na poljih $(1, 4)$ in $(7, 4)$ ne napadata: ležita sicer v isti vrstici, vendar je med njima na njej še kraljica $(5, 4)$. Podobno se tudi kraljici $(3, 2)$ in $(6, 5)$ ne napadata: ležita sicer na isti diagonali, vendar je med njima na njej še kraljica $(5, 4)$.

NEUPORABLJENE NALOGE IZ LETA 2012

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 7. tekmovanjem ACM v znanju računalništva (leta 2012), 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. Poudariti pa velja, da niti besedilo teh nalog niti njihove rešitve (ki so na str. 85–129) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Prestopna leta

Napiši program, ki ugotovi, na kateri dan v tednu se je pričelo največ prestopnih let v obdobju od vključno 1900 do 2012. Pogoji, ki morajo biti izpolnjeni, da je leto prestopno, so:

- letnica mora biti deljiva s 4; in
- letnica ne sme biti deljiva s 100, razen če je deljiva tudi s 400.

Leto 1900 se je začelo na ponedeljek.

2. Kazalca

Imamo uro z dvema kazalcema; eden kaže ure, drugi minute, vendar ne vemo, kateri je kateri.

(*a*) **Napiši podprogram** `Kazalca(k1, k2)`, ki dobi kot parametra položaj obeh kazalcev in ugotovi, koliko je ura (lahko se tudi zgodi, da vhodni podatki niso veljavni ali pa da imajo več rešitev). Kazalca se premikata zvezno in njun položaj je predstavljen kot realno število v stopinjah ($0°$ je navpično navzgor, $90°$ je desno, $180°$ je navpično navzdol ipd.).

Poišči tudi kakšen primer dvoumnih vhodnih podatkov, torej takih, pri katerih je možnih več veljavnih rešitev.

(*b*) Reši nalogo (*a*) še za primer, ko se kazalca premikata diskretno, le na začetku vsake minute (urni kazalec se premakne za pol kotne stopinje, minutni pa za 6 kotnih stopinj). Ali tudi pri tej nalogi obstajajo primeri dvoumnih vhodnih podatkov?

(*c*) Reši nalogo (*a*) še za primer, ko se kazalca premikata diskretno in to vedno v korakih po 6 kotnih stopinj; urni kazalec se premakne enkrat na 12 minut, minutni pa vsako minuto. Tudi tu razmisli o tem, ali so lahko vhodni podatki dvoumni.

3. Motocikel

Motocikli so zanimivo vozilo, ki ponuja voznikom veliko užitka in zabave v primerjavi z avtomobili. A tega užitka je kaj hitro konec, če zadnje kolo prehiti sprednje zaradi presilnega pospeševanja v ovinkih. Rešitve takrat praktično ni in padec je neizbežen. Zato se moderni motocikli vedno bolj pogosto ponašajo z nadzorom zdrsa koles.

V poenostavljenem primeru predpostavimo, da se prednje kolo in zadnje kolo motocikla tudi v ovinkih vrtita enako hitro (kar je dosti blizu resnici). Napisati

moramo podprogram nadzora zdrsa koles in preprečevanja le-tega. Ob pospeševanju moramo poskrbeti, da se zadnje kolo nikoli ne zavrti hitreje od prednjega. Hitrost vrtenja koles nam povedo tipala na obeh kolesih, naš podprogram pa mora posegati v nadzor plina in ga primerno odvzemati, da voznika ne bi odneslo iz ovinka.

Na obeh kolesih sta nameščeni luknjasti plošči s 48 luknjami (v enakomernih presledkih) in foto tipaloma, ki zaznata, ali je pod njima luknjica ali pregrada.

Na razpolago imamo sledeče podprograme:

- **bool** TipaloSpredaj() in **bool** TipaloZadaj() — povesta trenutno stanje tipal na kolesih (vrneta **true**, če je pod tipalom luknjica, sicer pa **false**);
- **double** Cas() — vrne trenutni čas kot realno število v sekundah, merjen od nekega začetnega trenutka; ta je postavljen dovolj daleč v preteklost, da funkcija vedno vrača pozitivna števila;
- **double** PlinVoznika() — vrne realno številko med 0 in 1, ki pove, v kakšno stanje je voznik trenutno zasukal ročko plina (0 — ročka ni zavrtena, 1 — na polno odprt plin).
- **void** PlinMotorja(**double** vrednost) — ta podprogram pokličemo mi in z njim povemo, koliko ukazanega dejanskega plina spustimo k motorju (število vrednost mora biti na območju od 0 do 1).

Napiši program, ki bo nadzoroval dodajanje plina voznika in pri tem poskrbel, da se zadnje kolo ne bo vrtelo hitreje od prednjega. To naj naredi tako, da če se zadnje kolo vrti x-krat hitreje od prednjega in če voznik zahteva plin p, naj k motorju spusti le p/x plina; če pa se zadnje kolo ne vrti hitreje od prednjega, naj voznikove ukaze posreduje motorju nespremenjene. Predpostaviš lahko, da računalnik deluje izredno hitro in lahko nekajdesetkrat preveri stanje enega senzorja, preden se ta tudi pri največji hitrosti premakne za eno luknjo naprej. Predpostavi tudi, da se kolesa nikoli ne vrtijo nazaj.

4. Tročrkovne kode

Imamo seznam imen jezikov in vsakemu imenu bi radi dodelili neko tročrkovno kodo. Idealno bi bilo, če bi bila koda sestavljena kar iz prvih treh črk imena, vendar to ni vedno mogoče, ker potem kode ne bi bile enolične (npr. *slo*venščina in *slo*vaščina). V takem primeru poskusimo s prvo, drugo in četrto črko; če tudi ta koda ni enolična, poskusimo s prvo, drugo in peto; itd.

Napiši program, ki po vrsti obdela imena jezikov iz danega vhodnega seznama in za vsak jezik sproti izbere primerno kodo iz čim zgodnejših črk v imenu (torej táko, ki je nismo uporabili še za noben jezik doslej; ali pa naj izpiše, da zanj primerne kode sploh ni mogoče izbrati, ker so vse že zasedene). Predpostaviš lahko, da se v imenih jezikov pojavljajo le male črke angleške abecede.

Bolj formalno lahko zapišemo: če ime jezika predstavlja niz n znakov $s = s_1 s_2 \ldots s_n$, mu moramo pripisati takšno kodo $s_i s_j s_k$, za katero velja $1 \leq i < j < k \leq n$, ki je nismo uporabili še za noben jezik doslej in ki ima v okviru teh omejitev leksikografsko najmanjšo trojico (i, j, k).

5. Google Mobil

Firma BAAS (Besni automatski automobili Slovenije) se je odločila narediti avtomobil, ki bi sam vozil po cestah in bil še boljši od Google mobila. Ena od uporabniških zahtev je, da bi uporabniki čim manj časa čakali pred semaforji: menijo, da je bolje, da se automobil do semaforja pripelje počasi, kot pa da pride do semaforja hitro, potem pa čaka zeleno luč.

Napiši podprogram, ki dobi naslednje podatke:

- **double** Razdalja — razdalja do semaforja kot realno število v metrih;
- **char** Stanje — stanje semaforja: `'z'`, če je trenutno zelen, ali `'r'`, če je rdeč;
- **double** Trajanje — koliko sekund bo trenutno stanje še trajalo;
- **double** zTrajanje — koliko sekund traja na tem semaforju zeleni del cikla;
- **double** rTrajanje — koliko sekund traja na tem semaforju rdeči del cikla;
- **double** Omejitev — največja dovoljena hitrost v kilometrih na uro.

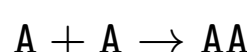
Podprogram naj vrne realno število, ki je največja primerna hitrost v kilometrih na uro, s katero mora avto voziti, da bo do semaforja pripeljal, ko bo ta zelen.

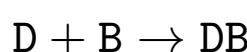
6. Celične himere

V laboratoriju imamo primarne celice tipa `A`, `B`, `C` in `D`. Celice so v nadzorovanem okolju podvržene fusogenu, kar omogoča njihovo združevanje. Med seboj se lahko združita dve celici istega tipa ali celici dveh različnih tipov. Ko se dve celici združita, ostane v novo nastali združeni celici zapis tipa obeh celic, iz katerih je nova celica nastala. Dobimo novo generacijo celic — nov tip celice. Celice različnih generacij se ne morejo združevati med sabo.

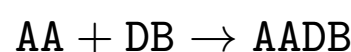
Ko dobimo novo generacijo združenih celic, jo izoliramo in z njo ponovimo poskus. Možen potek poskusa je na primer:

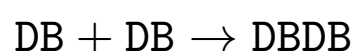
- 1. generacija:

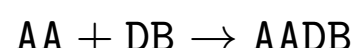

A + A → AA


D + B → DB

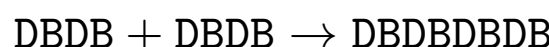
- 2. generacija:

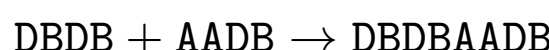

AA + DB → AADB


DB + DB → DBDB


AA + DB → AADB

- 3. generacija:


DBDB + DBDB → DBDBDBDB


DBDB + AADB → DBDBAADB

⋮

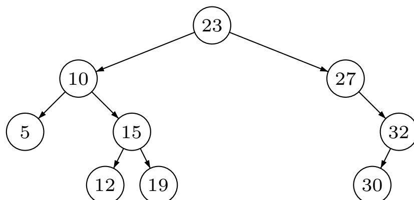
Po n generacijah pogledamo zapise v vseh nastalih združenih celicah zadnje generacije. **Napiši program** ali podprogram, ki z analizo zapisov n-te generacije za vsako od predhodnih generacij ugotovi:

- koliko različnih tipov je nastalo v tej generaciji;

- kateri tip te generacije je bil najbolj agresiven pri združevanju (in je torej zajet v največ združenih celicah naslednje generacije).

7. Rekonstrukcija drevesa

Binarno iskalno drevo je drevesasta podatkovna struktura, ki ima v vsakem vozlišču neko vrednost, poleg tega pa ima vsako vozlišče dve poddrevesi, levo in desno; vsako od poddreves je tudi sámo zase binarno iskalno drevo. (Eno ali drugo poddrevo ali pa celo obe sta lahko tudi prazni.) Pri tem za vsako vozlišče velja, da so vse vrednosti v njegovem levem poddrevesu manjše od tiste v vozlišču, ta pa je manjša od vseh vrednosti v njegovem desnem poddrevesu. Primer binarnega iskalnega drevesa kaže spodnja slika:



Znanih je več načinov, kako sistematično našteti vse vrednosti v drevesu. Ena možnost je na primer *premi obhod* (*pre-order traversal*), pri katerem drevo obdelamo tako, da najprej izpišemo vrednost iz korena drevesa, nato pa z rekurzivnim klicem obdelamo njegovo levo poddrevo (če ni prazno) in nato še desno poddrevo (če ni prazno). Še ena možnost je *obratni obhod* (*post-order traversal*), pri katerem najprej obdelamo levo poddrevo, nato desno poddrevo in nazadnje izpišemo vrednost iz korena.

Za drevo z gornje slike bi na primer pri premem obhodu dobili seznam 23, 10, 5, 15, 12, 19, 27, 32, 30, pri obratnem obhodu pa seznam 5, 12, 19, 15, 10, 30, 32, 27, 23.

Opiši postopek, ki kot vhodne podatke dobi seznam vrednosti, ki je nastal z obratnim obhodom nekega binarnega iskalnega drevesa, in izpiše seznam vrednosti, ki bi nastal s premim obhodom istega binarnega iskalnega drevesa.[6]

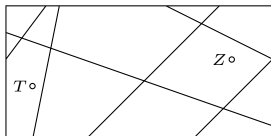
8. Avtomobil na daljinsko vodenje

Po karirasti mreži se premika avtomobil na daljinsko vodenje. Na daljincu so štiri tipke: premik 1 celico naprej, premik 1 celico nazaj (vzvratno), obrat na mestu za 90 stopinj levo in obrat na mestu za 90 stopinj desno. Težava je, da so oznake na tipkah zbledele in zdaj ne vemo, kaj počne katera tipka. Pred vsakim pritiskom na tipko izveš položaj avtomobila (kot par koordinat (x, y)), nikoli pa njegove orientacije. Mreža je neomejeno velika in na njej ni nikakršnih ovir. **Opiši postopek**, ki pripelje avtomobil do nekega zahtevanega končnega položaja (x_c, y_c) (te koordinate dobiš kot vhodni podatek).

[6] Na temo rekonstrukcije dreves se da sestaviti še več podobnih nalog; gl. npr. nalogo 1998.3.4, str. 348 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004*, kjer dobimo premi obhod in obhod po nivojih, nimamo pa zagotovil o urejenosti elementov v drevesu.

9. Tat in laserji

Laserski žarki, vzporedni tlom, razdelijo pravokotno sobo z zakladom na „celice“ (glej sliko). V vsaki takšni celici je tudi detektor teže. Imamo koordinate tatu in zaklada. Tat lahko stopi čez laserje (največ enega naenkrat, t.j. ne more stopiti čez točko, v kateri se seka več žarkov), detektorjem teže pa se ne zna izogniti. **Opiši postopek**, ki ugotovi, najmanj koliko detektorjev mora izključiti, da lahko pride do zaklada. Žarki so podani kot enačbe premic v ravnini.



Primer sobe, ki jo 6 žarkov deli v 12 celic. Točka T označuje začetni položaj tatu, Z pa položaj zaklada.

10. Malica

Dan je graf z vozlišči, oštevilčenimi od 1 do n. V vozlišču 1 se nahaja Janezkov dom, v n pa šola. Za vsako povezavo e našega grafa je podana utež p_e (z območja $0 \leq p_e \leq 1$), ki pomeni verjetnost, da mu na tem delu poti huligani vzamejo malico. **Opiši postopek**, ki ugotovi, kakšna je verjetnost, da bo Janezek v šolo prišel z malico, če izbere najbolj varno pot.

11. 3-d šah

Predstavljajmo si trodimenzionalno šahovnico, v kateri posamezna polja niso kvadratki, ampak kockice, pa tudi celotna šahovnica je kocka, ki jo sestavlja $n \times n \times n$ polj. V to šahovnico postavimo tri kraljice. **Opiši postopek**, ki kot vhodne podatke dobi koordinate vseh treh kraljic in izpiše število polj, ki jih istočasno napadajo vse tri kraljice. Koordinate so cela števila od 1 do n. Tvoja rešitev naj deluje učinkovito tudi za primere, ko je n zelo velik.[7]

Pravila za to, katera polja napada posamezna kraljica, so podobna kot pri običajnem šahu v dveh dimenzijah. Vsaka kraljica napada:

- 3 linije (v smeri vsake osi)
- 3×2 ravninski diagonali (v vsaki od ravnin, na katerih leži polje s kraljico)
- 4 prostorske diagonale.

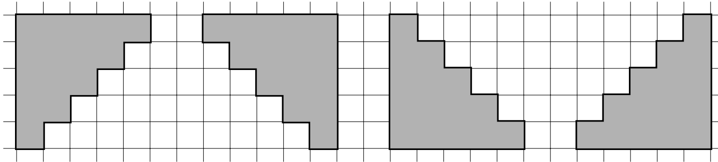
To, ali neka kraljica napada neko polje ali ne, je neodvisno od tega, ali med to kraljico in tistim poljem stoji še kakšna druga kraljica ali ne (kraljice torej ne blokirajo napadov druga druge). Kraljica napada tudi polje, na katerem stoji.

Ali znaš svojo rešitev posplošiti tudi primere z več kraljicami in s šahovnico, ima več kot tri dimenzije (recimo m kraljic na d-dimenzionalni šahovnici)? Kakšna je v tem primeru časovna zahtevnost tvoje rešitve? Posebej razmisli tudi o primeru, ko je kraljica ena sama ($m = 1$, število dimenzij d pa je veliko).

[7] To je posplošitev 2. naloge s šolskega tekmovanja 2013; tam je bila velikost šahovnice fiksirana na $8 \times 8 \times 8$ polj (gl. str. 30 v biltenu 2013).

12. Tangram

Pri igri tangram imamo sedem likov (kvadrat, paralelogram in 5 trikotnikov), ki jih poskušamo zlagati skupaj v razne nove like. V tej nalogi si bomo ogledali poenostavljeno različico igre, ki se dogaja na karirasti mreži in brez kvadrata in paralelograma. Vsi liki, ki jih imamo na voljo, so enakokraki pravokotni trikotniki (oz. približki takih trikotnikov, kolikor je to na karirasti mreži sploh mogoče — glej spodnjo sliko), podane pa so dolžine njihovih katet (število in velikost trikotnikov ni nujno takšna kot pri originalni različici tangrama). Tak trikotnik smemo pri sestavljanju novega lika poljubno zavrteti, a le v korakih po 90 stopinj. Tako na primer pri trikotniku s kateto 5 dobimo naslednje štiri možnosti:



Tudi ciljni lik, ki bi ga radi sestavili, je podan na karirasti mreži (torej kot dvodimenzionalna tabela, v kateri vsak element pove, ali je posamezna celica mreže črna ali bela). **Opiši postopek**, ki iz teh vhodnih podatkov (opis ciljnega lika in seznam dolžin katet vseh razpoložljivih trikotnikov) ugotovi, ali je mogoče iz danih trikotnikov sestaviti zahtevani ciljni lik. Pri tem se trikotniki ne smejo prekrivati in vsak trikotnik smemo uporabiti največ enkrat. Trikotnikov je največ 7 in dolžina katete posameznega trikotnika je največ 10. (Zanimiva je tudi težja različica naloge, pri kateri so trikotniki lahko večji, recimo s katetami do 1000.)

13. Taksist

Ekrem je dobil delovno vizo za n let. V New Yorku bo vozil taksi. Na začetku vsakega leta lahko kupi nov (ne-rabljen) avto. Pri tem mu v račun vzamejo stari avto po programu „staro za novo“. Zakonodaja tudi zahteva, da taksi ni starejši od petih let. Na koncu zadnjega leta bo prodal avto, preden se bo vrnil domov.

Ekrem ima seznam m avtomobilov, ki bodo prišli na trg v naslednjih n letih. Za vsakega ima naslednje podatke:

$$l \quad c \quad s_1 \quad s_2 \quad \ldots \quad s_5$$

Ti podatki pomenijo naslednje:

- l je leto izida
- c je cena, po kateri lahko kupi ta avtomobil v letu l;
- s_i je znesek, ki ga dobi za avtomobil, če ga proda po i letih („staro za novo“).

Za avtomobile bi rad zapravil kar najmanj denarja. **Opiši postopek**, ki ugotovi, koliko denarja zapravi, če ravna optimalno.

14. VIP

Odbor društva za pravice neskončnih zank se vsak mesec dobiva na sestankih, kjer obravnava razne zlorabe in mučenja neskončnih zank s strani programerjev. Sedejo za okroglo mizo z n stoli, kjer je za vsak stol določeno, kdo od članov odbora lahko na njem sedi ter kdo ga nadomešča v primeru njegove odsotnosti. Na žalost pa društvu tako zelo primanjkuje članov, da so nekatere osebe na tem seznamu napisane večkrat, zgodi se pa tudi, da se kdo od članov društva opraviči. Pomagaj tajnici društva organizirati sestanek za april, če imaš podano razporeditev po stolih ter kdo se je ta mesec opravičil (torej ga ne bo na sestanek). **Opiši postopek**, ki določi, kdo naj sedi na katerem stolu, ali pa ugotovi, da primernega razporeda sploh ni. Če je možnih več rešitev, je vseeno, katero poišče tvoj postopek.

(a) Lažja različica naloge: „kdo ga nadomešča v primeru njegove odsotnosti“ vzamemo zelo dobesedno — če je za stol i najprej naveden član a_i, nadomešča pa da ga b_i, to pomeni, da sme b_i zasesti ta stol le, če člana a_i sploh ni na sestanek.

(b) Težja različica: pravilo o nadomeščanju malo omilimo; če je za stol i naveden član a_i, nadomešča pa da ga b_i, potem sme b_i zasesti ta stol tudi v primeru, ko je a_i prisoten na sestanku (tako da smemo na primer člana a_i poslati na kakšen drug stol, če to pride prav).

Primer: recimo, da imamo n stolov in naslednje vhodne podatke:

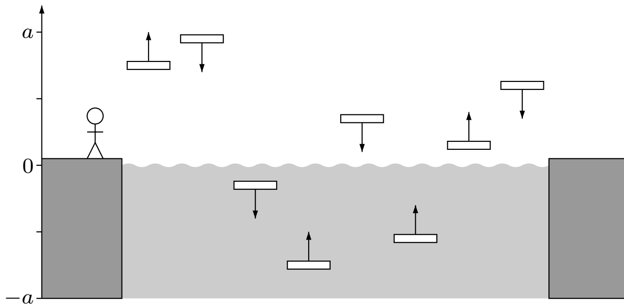
Št. stola	1	2	3	4
Član	A	B	B	C
Nadomešča ga	B	E	D	B

Recimo še, da člana B ne bo na sestanek. Primeren razpored članov na sestanku je tedaj (A, E, D, B) in to za obe različici naloge.

Če pa bi od sestanka izostal član E, bi bila različica (a) zdaj nerešljiva (član B ne more sedeti na stolih 2 in 3 hkrati, nadomeščati pa ga tudi ne sme nihče, saj je prisoten na sestanku), pri različici (b) pa je primeren razpored (A, B, D, C).

15. Prehod

V dvorazsežni arkadni igri Najboljši Marko mora naš mali dvorazsežni junak prečkati dolino, skozi katero teče hudournik. Pomaga si lahko z lebdečimi ploščami, ki se premikajo navzgor in navzdol s konstantno hitrostjo. Ko plošča pri premikanju navzgor doseže najvišjo lego a, se obrne in se začne premikati navzdol, dokler ne doseže $-a$, kjer se spet obrne, dokler ne doseže spet višine a in tako naprej. Marko lahko skoči na sosednjo ploščico levo ali desno, če se ta nahaja pod njim. Če pa se Marko nahaja na ploščici pod višino 0, ga bo odnesla voda in bo igre konec. Gibanje plošč in Marka poteka v diskretnih časovnih korakih; v vsakem koraku se višina vsake plošče spremeni za ± 1 (odvisno od tega, ali se premika gor ali dol). Vse plošče se gibljejo enako hitro in njihovo stanje (višina in smer gibanja) na začetku igre (ob času $t = 0$) je podano.



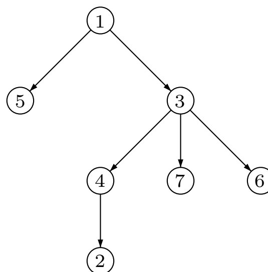
Opiši postopek, ki bo Marku povedal, v kolikšnem najmanjšsem času od začetnega stanja lahko Marko varno prečka reko. Če reke ne more prečkati, pa naj postopek ugotovi vsaj to, katera je najbolj desna plošča, do katere lahko pride, preden ga odnese hudournik.

16. Vodovod

Po deževju, ki sledi daljši suši, se vzdrževalec vodovodnega omrežja sooča s problemom. Čeprav ima dovolj vode, s katero lahko oskrbuje vse naročnike, so nekatere vodovodne cevi preprosto preozke in ne omogočajo dovolj velikega pretoka vode, da bi poskrbel za vse naročnike.

Omrežje je organizirano kot drevo; vozlišča so oštevilčena od 1 do n, pri čemer je vozlišče 1 koren drevesa (tam je vodno zajetje in od tam se potem vodo razpošilja naprej po sistemu). Za vsako od ostalih vozlišč (od 2 do n) je znano, iz katerega vozlišča priteka voda vanj in kolikšna je kapaciteta te povezave; poleg tega je tudi znano, koliko vode potrebujejo odjemalci pri tem vozlišču.

Opiši postopek, ki ugotovi, koliko cevi, ki povezujejo posamezna vozlišča, bo potrebno zamenjati, da bodo lahko naročnikom dovajali zadostne količine vode. (Cilj je torej ta, da bo za vsako vozlišče u veljalo, da kapaciteta cevi, ki priteka v u, zadošča za porabo odjemalcev v vozlišču u in še v vseh drugih vozliščih, ki se posredno ali neposredno oskrbujejo prek vozlišča u.)



Primer: recimo, da imamo drevo z gornje slike (v krogih so številke vozlišč) z naslednjimi podatki:

Št. vozlišča	1	2	3	4	5	6	7
Poraba pri njem	–	5	3	8	3	9	2
Oskrbuje se iz	–	4	1	3	1	3	3
Kapaciteta te povezave	–	5	23	15	6	5	4

Pravilni odgovor v tem primeru je, da moramo razširiti dve povezavi, namreč $3 \to 6$ (kjer je zdaj kapaciteta le 5, potrebujemo pa 9 enot pretoka za potrebe vozlišča 6) in $1 \to 3$ (kjer je zdaj kapaciteta 23, potrebujemo pa $3 + 8 + 5 + 2 + 9 = 27$ enot pretoka za potrebe vozlišča 3 in vseh, ki se posredno ali neposredno napajajo skozi to vozlišče).

17. Podobna števila

Podano imamo množico n nenegativnih celih števil $x_1, \ldots, x_n$, nad katerimi bi radi zdaj opravili veliko število poizvedb oblike (q_i, k_i). Pri vsaki poizvedbi nas zanima, katero število je k_i-to najbolj podobno številu q_i, pri čemer je definiramo podobnost takole: števili x in y sta si tem bolj podobni, čim manjša je vrednost x xor y. (Spomnimo se, da operator xor deluje takole: če si mislimo števila predstavljena v dvojiškem zapisu, so v številu x xor y prižgani biti na natanko tistih mestih, kjer je prižgan bit v natanko enem od števil x in y, v drugem pa ne. Tako je na primer 10 xor $6 = 1010_2$ xor $0110_2 = 1100_2 = 12$.)

Opiši, kako bi vhodne podatke $x_1, \ldots, x_n$ organiziral v neko primerno podatkovno strukturo, ki bi ti potem omogočala čim učinkoviteje odgovarjati na opisane poizvedbe. Opiši tudi postopek, s katerim bi (pri tej podatkovni strukturi) poiskal odgovor na posamezno poizvedbo.

18. Žetoni

Na karirasti mreži $w \times h$ polj imamo postavljene črne in bele žetone. Žetone smemo premikati s trenutnega polja na eno od sosednjih osmih polj (tistih, ki imajo s trenutnim poljem skupno vsaj eno oglišče), vendar le pod pogojem, da na tistem sosednjem polju še ne stoji nek žeton nasprotne barve; dovoljeno je torej, da se na istem polju znajde več žetonov, vendar le, če so vsi iste barve. S takšnim premikanjem bi radi razporedili žetone tako, da bo na koncu čim manj polj zasedenih (polje je zasedeno, če na njem stoji eden ali več žetonov).

V tej nalogi se pravzaprav skriva več precej različnih nalog, ki se razlikujejo po tem, kakšne premike dovolimo:

(a) Premikati smemo le bele žetone, črnih pa ne; pri tem pa smemo posamezni beli žeton premakniti tudi po večkrat. **Opiši postopek**, ki ugotovi, kakšno je najmanjše število zasedenih polj, ki ga je mogoče s takšnimi premiki doseči.

(b) Reši nalogo (a) tudi za primer, ko smemo premikati tako bele kot črne žetone. Premikamo vedno le po en žeton naenkrat, vendar lahko posamezni žeton pride na vrsto po večkrat. **Opiši postopek** premikanja žetonov, s katerim dosežemo najmanjše možno število zasedenih polj.

(c) Vsak žeton smemo premakniti največ enkrat in vse te premike izvedemo hkrati; torej lahko žeton premaknemo le na tako polje, ki je bilo v začetnem stanju mreže prazno ali pa je vsebovalo žeton iste barve. **Opiši postopek**, ki ugotovi

najmanjše število zasedenih polj, ki ga je mogoče na ta način doseči. Predpostavi, da mreža obsega eno samo vrstico, torej $h = 1$.

(d) Reši nalogo (c) še za poljubno mrežo, torej sta lahko tako w kot h večja od 1; predpostavi pa, da je vsaj ena od dimenzij mreže majhna: $\min\{w, h\} \leq 10$.

19. Dvojiško Conwayevo zaporedje

Conwayevo zaporedje je zaporedje nizov $s_0, s_1, s_2, \ldots$, v katerem vsak niz opisuje prejšnjega. Začnimo z nizom $s_0 = 1$. Opišimo ga: to je ena (1) enica (1); tako dobimo $s_1 = 11$. Opišimo zdaj tega: to sta dve (2) enici (1); tako dobimo $s_2 = 21$. Opišimo njega: najprej ena (1) dvojka (2), nato ena (1) enica (1); tako dobimo $s_3 = 1211$. Opišimo še njega: ena (1) enica (1), ena (1) dvojka (2), dve (2) enici (1); tako dobimo $s_4 = 111221$. Če tako nadaljujemo, dobimo $s_5 = 312211$, $s_6 = 13112221$ in tako naprej.

Pri tej nalogi bomo namesto s tem zaporedjem delali z njegovo dvojiško različico. Edina razlika je torej v tem, da števila zapisujemo v dvojiškem zapisu. Začnemo spet z nizom $t_0 = 1$. Opišimo ga: to je ena (1) enica (1); tako dobimo $t_1 = 11$. Opišimo zdaj tega: to sta dve (10_2) enici (1): tako dobimo $t_2 = 101$. Opišimo njega: ena (1) enica (1), ena (1) ničla (0) in ena (1) enica (1); tako dobimo $t_3 = 111011$. Opišimo še njega: tri (11_2) enice (1), ena (1) ničla (0) in še dve (10_2) enici (1); tako dobimo $t_4 = 11110101$. Opišimo še tega: štiri (100_2) enice (1), ena (1) ničla (0), ena (1) enica (1), ena (1) ničla (0) in še ena (1) enica (1); tako dobimo $t_5 = 100110111011$. Če tako nadaljujemo, dobimo $t_6 = 111001011011110101$ in tako naprej.

Napiši podprogram, ki za dani števili n in m izračuna m-to števko v nizu t_n. Predpostaviš lahko, da velja omejitev $1 \leq n \leq 100$.

20. Neprireditveni stavki

Zamislimo si preprost programski jezik, v katerem imamo na voljo n spremenljivk z imeni $x_1, \ldots, x_n$, program pa je sestavljen iz zaporedja stavkov. Možni obliki posameznega stavka sta dve:

- Prireditveni stavek: $x_i == x_j$ zapiše vrednost spremenljivke x_i v spremenljivko x_j.
- Neprireditveni stavek: $x_i =! x_j$ ne zapiše vrednosti spremenljivke x_i v spremenljivko x_j (z drugimi besedami, ta stavek ne naredi ničesar).[8]

Drugih operacij (aritmetičnih, logičnih itd.) naš programski jezik ne podpira, prav tako nima pogojnih skokov, zank, podprogramov in podobnih reči.

Predpostavimo, da nimata na začetku izvajanja programa nobeni dve spremenljivki enake vrednosti. Med izvajanjem programa pa se zaradi prireditvenih stavkov seveda lahko zgodi, da dobi več spremenljivk enako vrednost. Definirajmo *oceno* programa kot število spremenljivk, ki imajo na koncu izvajanja tisto vrednost, ki je bila pred začetkom izvajanja v spremenljivki x_1.

Radi bi dosegli, da bi imel program čim višjo oceno, in v ta namen ga smemo tudi malo spremeniti: v nekaterih stavkih (sami se odločimo, v katerih; lahko tudi

[8]To nalogo je navdihnila ena od cvetk s prejšnjega tekmovanja (gl. str. 133 v biltenu 2011).

v nobenem) lahko spremenimo operatorje (iz prireditvenih v neprireditvene in obratno). Ne smemo pa spreminjati vrstnega reda stavkov ali pa indeksov spremenljivk, ki nastopata v posameznem stavku. **Opiši postopek**, ki ugotovi, kakšna je najvišja ocena, ki jo lahko na ta način dosežemo, in kolikšno je najmanjše število potrebnih sprememb operatorjev, s katerim lahko dosežemo to oceno. Število spremenljivk, n, je največ 20, stavkov pa je lahko več sto.

REŠITVE NALOG ZA PRVO SKUPINO

1. Dnevnik

Vhodno datoteko berimo po vrsticah in si poleg trenutne vrstice zapomnimo še prejšnjo vrstico in dosedanje število pojavitev te prejšnje vrstice (spremenljivki prejsnja in n). Na začetku izvajanja prejšnje vrstice še ni, zato postavimo n na 0.

Ko preberemo novo vrstico, jo primerjamo s prejšnjo; če sta enaki, ne izpišemo ničesar, pač pa le povečamo števec pojavitev. Če pa sta različni, to pomeni, da se dosedanji blok enakih vrstic (tistih, ki so enake nizu prejsnja) končuje in moramo najprej dokončati pripadajoči izpis zanj (saj smo doslej od tega bloka izpisali le prvo vrstico). Zdaj preverimo števec pojavitev n; če je enak 2, izpišemo vrstico prejsnja še enkrat, če pa je večji od 2, izpišemo niz oblike „ponovljeno še $(n-1)$-krat“. Nato izpišemo novo vrstico, si jo zapomnimo v spremenljivki prejsnja in postavimo števec n na 1.

Poseben primer nastopi na koncu vhodne datoteke, ko namesto nove vrstice dobimo prazen niz. Tega ne izpišemo, pač pa zanko takrat prekinemo. (Pred tem pa ga obravnavamo enako kot običajno vrstico, kar bo zagotovilo, da bomo pravilno zaključili izpis zadnjega bloka nepraznih vrstic.)

Zapišimo dobljeni postopek v pythonu:

```
prejsnja = ""; n = 0
while True:
  nova = PreberiDogodek()
  if n > 0 and prejsnja == nova: n += 1; continue
  if n == 2: print "%s" % prejsnja
  elif n > 2: print "ponovljeno se %d-krat" % (n − 1)
  prejsnja = nova; n = 1
  if nova: print nova
  else: break
```

Rešitev v C-ju je malenkost bolj zapletena, ker je C za delo z nizi malo bolj neroden. Predpostavimo, da PreberiDogodek ob vsakem klicu alocira nov niz na kopici in nam vrne kazalec nanj (tipa **char** *), dealokacija tega pomnilnika pa je naša skrb; na koncu vhodne datoteke pa naj PreberiDogodek vrne 0. (Čisto dobro bi se dalo narediti tudi drugače, na primer tako, da bi podprogram PreberiDogodek shranil niz v neko tabelo, ki bi mu jo podali kot parameter.) Razlika v primerjavi z gornjo pythonovsko rešitvijo je torej predvsem ta, da niz nova po obdelavi posamezne vrstice dealociramo (kličemo standardno funkcijo free), razen če smo si ta niz zapomnili tudi kot prejsnja; v tem primeru pa moramo dealocirati stari niz prejsnja, preden mu priredimo nova. (Na primer, ko je prejsnja v prvi iteraciji zanke še 0, nam ni treba posebej paziti, saj je klic free(0) dovoljen in po definiciji ne naredi ničesar.)

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

extern char *PreberiDogodek();

int main()
{
  char *prejsnja = 0, *trenutna = 0; int n = 0;
  do
```

```
  {
    nova = PreberiDogodek();
    if (n > 0 && nova && strcmp(prejsnja, nova) == 0) { n++; free(nova); continue; }
    if (n == 2) printf("%s\n", prejsnja);
    else if (n > 2) printf("ponovljeno se %d-krat\n", n − 1);
    n = 1; free(prejsnja); prejsnja = nova;
    if (nova) printf("%s\n", nova);
  }
  while (nova != 0);
  return 0;
}
```

2. Proizvodnja čopičev

Dolžino i-te palice označimo z L_i. Če to dolžino delimo z r, nam celi del količnika pove, koliko ročajev bi se dalo narediti iz te palice. Če seštejemo to po vseh palicah, dobimo maksimalno število ročajev, ki bi se jih dalo narediti iz razpoložljivih palic. To je tudi zgornja meja za število čopičev, ki bi se jih dalo izdelati, saj za vsak čopič potrebujemo po en ročaj.

Ni pa nujno, da toliko čopičev res lahko naredimo, saj nam mora po izdelavi ročajev ostati še dovolj lesa za konice. Če bi radi na primer naredili c čopičev, potrebujemo poleg ročajev še $c \cdot k$ lesa za izdelavo konic. Če od skupne dolžine palic $L_1 + \ldots + L_n$ odštejemo skupno dolžino ročajev, $c \cdot r$, nam razlika pove, koliko lesa ostane na razpolago za izdelavo konic. Ta razlika mora biti torej vsaj $c \cdot k$, sicer toliko čopičev ne bomo mogli izdelati. Tako imamo neenačbo $L_1 + \ldots + L_n - c \cdot r \geq c \cdot k$, iz česar dobimo $c \leq (L_1 + \ldots + L_n)/(k + r)$.

Naše število čopičev mora ustrezati obema omejitvama (da bo dovolj ročajev in tudi dovolj lesa za konice), zato moramo od obeh doslej dobljenih zgornjih mej vzeti nižjo.

Zapišimo našo rešitev še v C-ju:

```
int KolikoCopicev(int n, int k, int r, int L[ ])
{
  int skupDolzina = 0, maxRocajev = 0, maxCopicev, i;
  for (i = 0; i < n; i++) {
    skupDolzina += L[i];
    maxRocajev += L[i] / r; }
  maxCopicev = skupDolzina / (k + r);
  if (maxRocajev < maxCopicev) maxCopicev = maxRocajev;
  return maxCopicev;
}
```

3. Generali

Iz definicije v opisu naloge sledi, da lahko skupina generalov prepreči sprožitev bombe, če niti vsi ostali generali skupaj nimajo vseh ključev. Z drugimi besedami, to se zgodi takrat, ko za vsaj en ključ velja, da so vsi izvodi tega ključa v rokah generalov iz naše pacifistične skupine. Sistem je torej r-odporen, če ima vsak ključ vsaj $r + 1$ generalov (saj takrat gotovo velja, da kakorkoli izberemo r generalov, ne bomo mogli obvladovati vseh izvodov posameznega ključa; in po drugi strani, če bi obstajal kak ključ v r ali manj izvodih, potem bi se dalo izbrati r generalov tako, da bi obvladovali vse izvode tega ključa in bi lahko ostalim preprečili sprožitev bombe).

Naš postopek mora torej za vsak ključ prešteti, koliko generalov ga ima. V ta namen si lahko pomagamo s tabelo k elementov, ki za vsak ključ povedo, pri koliko generalih smo ga doslej opazili. V zanki se sprehodimo po vseh generalih in pri vsakem od njih po vseh njegovih ključih ter povečujemo števce v tabeli. Na koncu le še preverimo, če so vsi elementi tabele večji od r. Zapišimo ta postopek še s psevdokodo:

algoritem JeOdporen(r, G_1, ..., G_n):
  za vsak ključ j od 1 do k postavi *štPojavitev*[j] na 0;
  za vsakega generala i do 1 do n:
    za vsak ključ j iz množice G_i:
      povečaj *štPojavitev*[j] za 1;
  za vsak ključ j od 1 do k:
    if *štPojavitev*[j] $\leq r$ **then return false**;
  return true;

Naloga sprašuje tudi, kako bi v računalniku predstavili množice G_i. Ena možnost je, da predstavimo vsako tako množico s tabelo k logičnih vrednosti, za vsak ključ po eno, ki nam pove, ali general i ta ključ ima ali ne. (Še bolj varčna različica te predstavitve je tabela k bitov, torej približno $k/8$ bytov.) Slabost takšne predstavitve je, da ko hočemo pregledati ključe, ki jih ima general, moramo iti v zanki po vseh možnih ključih od 1 do k in za vsakega od njih v tabeli preverjati, ali ga ta general ima ali ne. To je še posebej neugodno, če so posamezne množice G_i majhne v primerjavi s številom vseh možnih ključev (torej k).

Druga možnost (ki je za naš namen primernejša) pa je, da predstavimo množico G_i s seznamom, v katerem so navedeni le tisti ključi, ki jih general i dejansko ima. Ta seznam je lahko predstavljen s tabelo ali pa kot veriga členov, povezanih s kazalci (*linked list*).

4. Uniforme

Pomagamo si lahko s tabelo, ki za vsako kombinacijo velikosti (od 1 do 100) in kosa (od 1 do 3) hrani število prejetih kosov te velikosti. V spodnjem programu imamo v ta namen tabelo zaloga; na začetku postavimo vse njene elemente na 0, nato pa beremo vhodne podatke in po vsaki prebrani vrstici povečamo ustrezni element tabele za 1. (Paziti moramo še na to, da se indeksi v tabelo štejejo od 0 naprej, v vhodnih podatkih pa so števila od 1 naprej.)

Nato lahko za vsako velikost določimo število popolnih uniform, ki jih lahko sestavimo pri tej velikosti. Pogledati moramo, katerega od teh kosov imamo (v tej velikosti) na razpolago v najmanj izvodih; toliko popolnih uniform lahko sestavimo (saj imamo tudi ostala dva kosa v vsaj toliko izvodih), več pa ne (saj bi nam tega kosa za nekatere uniforme zmanjkalo). Ko za neko velikost poznamo število popolnih uniform te velikosti, lahko to število prištejemo spremenljivki rezultat, v kateri se bo tako na koncu nabralo skupno število popolnih uniform, po katerem sprašuje naloga.

```
#include <stdio.h>
#define MaxVelikost 100
#define StKosov 3

int main()
```

```
{
   int zaloga[MaxVelikost][StKosov], n, vel, kos, naj, rezultat;

   /* Inicializirajmo tabelo zaloga. */
   for (vel = 0; vel < MaxVelikost; vel++) for (kos = 0; kos < StKosov; kos++)
      zaloga[vel][kos] = 0;

   /* Preberimo vhodne podatke. */
   scanf("%d", &n);
   while (n−− > 0) {
      scanf("%d %d", &kos, &vel);
      zaloga[vel − 1][kos − 1]++; }

   /* Izračunajmo rezultat. */
   for (rezultat = 0, vel = 0; vel < MaxVelikost; vel++) {
      /* Koliko popolnih uniform velikosti vel lahko sestavimo? */
      naj = zaloga[vel][0];
      for (kos = 1; kos < StKosov; kos++)
         if (zaloga[vel][kos] < naj)
            naj = zaloga[vel][kos];
      rezultat += naj; }

   printf("%d\n", rezultat); return 0;
}
```

Gornja rešitev poišče minimum (pri posamezni velikosti) z zanko po kosih; ker pa je vnaprej znano, da so kosi le trije, bi lahko minimum poiskali tudi z dvema pogojnima stavkoma:

```
      naj = zaloga[vel][0];
      if (zaloga[vel][1] < naj) naj = zaloga[vel][1];
      if (zaloga[vel][2] < naj) naj = zaloga[vel][2];
```

5. Davek na ograjo

Z dvema gnezdenima zankama (po x in po y) se sprehodimo po vseh kvadratkih naše mreže. Pri vsakem si zapomnimo njegovega lastnika (v spremenljivki lastnik), nato pa (s še eno vgnezdeno zanko) preglejmo njegove štiri sosede. Sosedje kvadratka (x, y) imajo koordinate $(x \pm 1, y)$ in $(x, y \pm 1)$; spodnji program jih računa s pomočjo tabel dx in dy. Če ima sosednji kvadratek drugega lastnika kot naš opazovani (x, y), povečajmo dolžino ograj našega lastnika za 1 (v tabeli dolzina, v kateri smo na začetku vse elemente inicializirali na 0). Paziti moramo še na možnost, da sosednji kvadratek leži zunaj mreže; spodnji program v tem primeru postavi sosed $= -1$, kar bo gotovo različno od lastnika trenutnega kvadratka (x, y), tako da bo program pravilno preštel tudi ograje na zunanjem robu mreže.

Na koncu tega prehoda čez celo mrežo imamo v tabeli dolzina za vsakega lastnika skupno dolžino ograj okoli njegovih kvadratkov in te rezultate moramo le še izpisati.

```
#include <stdio.h>
#define MaxW 250
#define MaxH 250
#define MaxLastnikov (MaxW * MaxH)

int main()
{
   const int dx[ ] = { −1, 1, 0, 0 }, dy[ ] = { 0, 0, −1, 1 };
```

```
  int dolzina[MaxLastnikov], x, y, xx, yy, smer, lastnik, sosed;
  int h = Visina(), w = Sirina(), n = StLastnikov();

  /* Inicializirajmo tabelo dolžin. */
  for (lastnik = 0; lastnik < n; lastnik++) dolzina[lastnik] = 0;

  /* Preglejmo celo mrežo. */
  for (x = 0; x < w; x++) for (y = 0; y < h; y++) {
    lastnik = Lastnik(x, y);

    /* Preglejmo sosede polja (x, y). */
    for (smer = 0; smer < 4; smer++) {
      xx = x + dx[smer]; yy = y + dy[smer]; /* sosednje polje */

      /* Kdo je lastnik polja (xx, yy)? */
      if (xx < 0 || xx >= w || yy < 0 || yy >= h) sosed = −1;
      else sosed = Lastnik(xx, yy);

      /* Povečajmo števec ograj, če sta lastnika različna. */
      if (lastnik != sosed) dolzina[lastnik]++; }}

  /* Izpišimo rezultate. */
  for (lastnik = 0; lastnik < n; lastnik++)
    printf("Dolžina ograj lastnika %d je %d.\n", lastnik, dolzina[lastnik]);
  return 0;
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Vnos šifre

Naloga je zelo primerna za reševanje z rekurzijo. Če šifro gradimo postopoma in dodajamo števke na konec šifre, imamo na vsakem koraku več možnosti, kako nadaljevati: za naslednjo števko lahko vzamemo katerokoli sosedo prejšnje števke (vključno s prešnjo števko samo). Za vsako od teh možnih nadaljevanj izvedemo rekurziven klic, ki na podoben način zgenerira še preostanek šifre. Robni primer rekurzije nastopi takrat, ko je šifra že dolga n znakov; takrat jo moramo le še izpisati. V spodnji rešitvi za te stvari skrbi podprogram Rekurzija, ki dobi tri parametre: tabelo sifra, v kateri pripravljamo šifro (in jo na koncu izpišemo); zahtevano končno dolžino šifre n; in pa trenutno globino rekurzije i — ta parameter nam pove, da so v nizu sifra že vpisani znaki na indeksih od 0 do i − 1, ne pa še tisti na indeksih od i do n − 1.

Vprašanje je še, kako vemo, katere so možne sosede prejšnje števke. Spodnja rešitev ima za to kar tabelo (sosede), v kateri je za vsako števko od 0 do 9 naveden niz z vsemi njenimi sosednjimi števkami (vključno s to števko samo).

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

const char *sosede[] = { "08", "124", "1235", "236", "1457",
                         "24568", "3569", "478", "57890", "689" };

void Rekurzija(char *sifra, int i, int n)
{
  const char *nasl;
  /* Če je šifra zdaj že dolga n znakov, jo le še izpišemo. */
  if (i >= n) printf("%s\n", sifra);
  /* Sicer pa na vse možne načine izberimo naslednji znak in nadaljujmo z rekurzijo. */
  else for (nasl = sosede[sifra[i - 1] - '0']; *nasl; nasl++) {
    sifra[i] = *nasl;
    Rekurzija(sifra, i + 1, n); }
}
```

Glavni podprogram NastejSifre zdaj nima veliko dela: pripraviti mora tabelo sifra in za vsako možno začetno števko (od 0 do 9) sprožiti rekurzivni klic, ki bo izpisal vse šifre, ki se začnejo na to števko.

```
void NastejSifre(int n)
{
  int d;
  char *sifra = (char *) malloc(n + 1);
  sifra[n] = 0; /* Postavimo znak za konec niza. */

  for (d = 0; d < 10; d++) { /* Prva števka je lahko katera koli. */
    sifra[0] = '0' + d;
    Rekurzija(sifra, 1, n); }

  free(sifra);
}
```

Razmislimo še o težji različici naloge, pri kateri nas zanima le, koliko je vseh n-mestnih šifer. Gornjo rekurzivno rešitev bi lahko seveda spremenili tako, da bi šifre štela, namesto da jih izpisuje (pri tem bi si na primer pomagala z neko globalno

spremenljivko), vendar je ta rešitev neučinkovita, saj se izkaže, da število šifer hitro narašča z n.

Naj bo $a_n(d)$ število n-mestnih šifer, ki se začnejo na števko d. Pri $n = 1$ je seveda $a_1(d) = 1$ za vse d od 0 do 9, saj je edina primerna enomestna šifra takrat kar števka d sama. Pri večjih n pa lahko razmišljamo takole: če se mora šifra začeti na d, je preostanek te šifre dolg $n - 1$ števk, začeti pa se sme na poljubno tako števko d', ki leži na isti tipki kot d ali pa na kakšni od njenih sosed. Množico takih števk označimo s $S(d)$; torej imamo rekurzivno zvezo $a_n(d) = \sum_{d' \in S(d)} a_{n-1}(d')$. Z njo ni težko računati vrednosti a_n za vse večje n; na koncu pa izračunamo skupno število vseh n-mestnih šifer (ne glede na prvo števko) po formuli $b_n = \sum_{d=0}^{9} a_n(d)$.

Zapišimo to rešitev še v C-ju. Opazimo lahko, da ko računamo vrednosti a_n, potrebujemo le vrednosti a_{n-1}, ne pa več a_{n-2}, a_{n-3} in podobno. Torej je dovolj, če hranimo te vrednosti le za dva zaporedna n-ja; spodnji podprogram ima v ta namen tabelo a z dvema vrsticama in vrednost $a_n(d)$ hrani v a[n % 2][d].

```
long long PrestejSifre(int n)
{
  int d, i; const char *nasl;
  long long a[2][10], skupaj;
  /* Vsi a_1(d) so enaki 1; kar vpišimo jih v tabelo a. */
  for (d = 0; d < 10; d++) a[1][d] = 1;
  for (i = 2; i <= n; i++)
    /* Vrednosti a_{i-1}(d) že poznamo; izračunajmo zdaj vse a_i(d). */
    for (d = 0; d < 10; d++) {
      a[i % 2][d] = 0;
      for (nasl = sosede[d]; *nasl; nasl++)
        a[i % 2][d] += a[(i - 1) % 2][*nasl - '0']; }
  /* Seštejmo a_n(d) za vse d, da dobimo iskani rezultat. */
  for (d = 0, skupaj = 0; d < 10; d++)
    skupaj += a[n % 2][d];
  return skupaj;
}
```

Lepo pri tem postopku je, da za izračun b_n (števila vseh n-mestnih šifer) porabi le $O(n)$ časa. Tip **long long**, ki smo ga uporabili v gornji rešitvi, je načeloma 64-biten, kar bi, kot se izkaže, zadostovalo do $n = 31$. Naslednja tabela kaže skupno število n-mestnih šifer za nekaj majhnih n-jev:

n	b_n	n	b_n	n	b_n
1	10	6	7 990	15	1 583 522 062
2	36	7	30 984	20	1 388 236 003 974
3	138	8	120 130	25	1 217 034 591 721 250
4	532	9	465 832	30	1 066 946 359 189 051 678
5	2 062	10	1 806 282	35	935 367 455 921 546 859 044

V splošnem je $b_n \approx 2{,}35 \cdot 3{,}8776^n$.[9]

[9] Za boljše razumevanje tega pojava si je koristno pomagati z linearno algebro. Mislimo si 10-dimenzionalni vektor (stolpec) s komponentami $a_n(d)$; recimo mu $\mathbf{a}_n$. Rekurzivno zvezo $a_n(d) = \sum_{d' \in S(d)} a_{n-1}(d')$ lahko zdaj zapišemo v matrični obliki: $\mathbf{a}_n = S\mathbf{a}_{n-1}$ za primerno izbrano matriko S velikosti 10×10 (njeni elementi so enaki 0 ali 1, pri čemer enice povedo, kdaj sta si

2. Prenova ceste

Načeloma nas za vsako točko (recimo t) zanima, katerih k prebivalcev ji leži najbližje (in za katero podjetje bi ti prebivalci glasovali). Ta soseščina ima naslednjo lepo lastnost: če sta i in j (za $i < j$) med najbližjimi k sosedi naše točke t, potem so tudi vsi vmesni prebivalci ($i + 1, i + 2, \ldots, j - 1$) med najbližjimi k sosedi točke t.[10] To pomeni, da skupina k najbližjih sosedov točke t gotovo tvori neko strnjeno podzaporedje (oblike $z, z + 1, \ldots, z + k - 1$). Na primer, če leži t levo od vseh prebivalcev, so njegovi najbližji sosedje kar najbolj levi prebivalci in imamo $z = 1$; podobno, če leži t desno od vseh prebivalcev, so najbližji sosedje kar najbolj desni prebivalci in imamo $z = n - k + 1$ (če je n število vseh prebivalcev).

Tako torej vidimo, da čeprav je možnih točk t neskončno mnogo (saj je premica zvezna), je možnih različnih soseščin le $n - k + 1$ (ker so odvisne le od tega, pri katerem prebivalcu z se začnejo).

Kako se spreminja z (torej najbolj levi med najbližjimi k sosedi), če se s točko t počasi premikamo od leve proti desni? Na začetku so naši najbližji sosedje prebivalci od 1 do k. Do prve spremembe pride, ko prebivalec 1 izpade iz te soseščine in vanjo pride prebivalec $k + 1$ (najbolj levi prebivalec v soseščini ima zdaj številko 2, torej imamo $z = 2$). To se očitno zgodi takrat, ko nam postane $k + 1$ bližje kot 1; torej takrat, ko pridemo ravno na pol poti med tema dvema prebivalcema, pri $t = (x_1 + x_{k+1})/2$. Naslednja sprememba bo nastopila pri $t = (x_2 + x_{k+2})/2$, ko bo iz soseščine izpadel prebivalec 2, vanjo pa bo prišel prebivalec $k + 2$. Tako lahko nadaljujemo vse do konca seznama. Sproti lahko vzdržujemo tudi števec, ki pove, koliko izmed najbližjih k sosedov podpira prvo podjetje; tega števca ni težko popraviti, ko prebivalci prihajajo v soseščino in izpadajo iz nje.

Dobljeni postopek je dovolj preprost, da ga lahko zapišemo kar kot podprogram v C-ju. Poleg tabel x in p pričakuje kot parametre še skupno število prebivalcev n, velikost soseščine k in skupno dolžino opazovanega odseka ceste D. (V besedilu naloge je n fiksiran na 10^6, dolžina D pa na 100 km.)

```
#include <stdio.h>

void PrenovaCeste(int n, int k, const double x[], const int p[], double D)
{
  double xOd, xDo, dolzina[2] = { 0, 0 };
  int k1 = 0, z;
  for (z = 0; z < k; z++) if (p[z]) k1++;
```

dve števki sosednji ali enaki). Pri $n = 1$ imamo seveda $\mathbf{a}_1 = (1, 1, \ldots, 1)^t$. Matriki S poiščimo lastne vrednosti λ_i in sestavimo bazo iz pripadajočih lastnih vektorjev $\mathbf{e}_i$; izrazimo $\mathbf{a}_1$ v tej bazi kot $\mathbf{a}_1 = \sum_i \mu_i \mathbf{e}_i$. Potem je $S\mathbf{a}_1 = \sum_i \mu_i S\mathbf{e}_i = \sum_i \mu_i \lambda_i \mathbf{e}_i$ in v splošnem $\mathbf{a}_n = S^{n-1}\mathbf{a}_1 = \sum_i \mu_i \lambda_i^{n-1} \mathbf{e}_i$. Vidimo, da v tej vsoti sčasoma močno prevladuje tisti člen, ki ima največjo $|\lambda_i|$. Brez izgube za splošnost recimo, da to nastopi pri $i = 1$; torej je $\mathbf{a}_n \approx \mu_1 \lambda_1^{n-1} \mathbf{e}_1$ in $b_n = \mathbf{a}_n^t \mathbf{a}_1 \approx C \cdot \lambda_1^n$ za konstanto $C = \mu_1(\mathbf{e}_1^t \mathbf{a}_1)/\lambda_1$. Pri naši konkretni matriki S se izkaže, da je največja lastna vrednost enaka približno 3,8776; tako pridemo do zveze $b_n \approx C \cdot 3{,}8776$, ki smo jo videli zgoraj.

[10] O tem se lahko prepričamo takole. Če i in j ležita levo od t, potem so vsi vmesni prebivalci točki t še bližje kot prebivalec i; torej, če je i med najbližjimi k sosedi, so ti vmesni prebivalci tudi. Podobno razmišljamo, če i in j ležita desno od t. Ostane še možnost, da i leži levo od t, prebivalec j pa desno od t; v tem primeru so med sosedi $i + 1, \ldots, j - 1$ tisti, ki ležijo levo od t, gotovo bližje t-ju kot sosed i, in tisti, ki ležijo desno od t, so mu gotovo bližje kot sosed j; ker sta i in j oba med najbližjimi k sosedi, so torej ti vmesni prebivalci tudi.

```
  for (z = 0; z + k <= n; z++) {
    /* Na katerem intervalu x-koordinat so najbližji sosedje ravno
       prebivalci z, z + 1, . . . , z + k − 1? */
    xOd = (z > 0) ? (x[z − 1] + x[z − 1 + k]) / 2 : 0;
    xDo = (z + k < n) ? (x[z] + x[z + k]) / 2 : D;
    /* Med temi k prebivalci jih k1 podpira podjetje 1, ostali pa podjetje 0. */
    dolzina[k1 >= k − k1 ? 1 : 0] += xDo − xOd;
    /* Popravimo števec k1, da bo pripravljen za naslednjo iteracijo. */
    if (z + k < n) k1 = k1 − p[z] + p[z + k]; }
  printf("Kangaroads popravi %g km, Wallabyway pa %g km.\n",
    dolzina[0], dolzina[1]);
}
```

3. Skrivno sporočilo

V zanki pregledujemo istoležne znake prvotnega niza p1 in njegove šifrirane različice c1. Pri tem si v tabelo sifra zapisujemo šifre posameznih znakov; pri tistih znakih, za katere šifre še ne poznamo, pa imejmo sifra[c] = '*'. Podobno imamo tudi tabelo original, v kateri za vsak znak piše, kateri znak originalnega besedila se zašifrira vanj; če za nek znak še ne vemo, kaj se zašifrira vanj, pa imamo original[c] = '*'.

Na vsakem koraku preverimo naslednje: če za trenutni znak niza p1 šifre še ne poznamo, jo lahko zdaj odčitamo iz trenutnega znaka niza c1 (če le-ta ni zvezdica); če pa za trenutni znak niza p1 šifro že poznamo, lahko preverimo, če je v c1 zdaj ista šifra. Če tu opazimo neujemanje, izpišemo obvestilo o napaki. Podobno, če za trenutni znak niza c1 še ne poznamo originala, ga lahko zdaj odčitamo iz trenutnega znaka niza p1; če pa nek original za trenutni znak niza c1 že poznamo, moramo preveriti, če se ujema s trenutnim znakom niza p1; če opazimo neujemanje, izpišemo obvestilo o napaki.

Poseben primer nastopi, če lahko na ta način ugotovimo šifre vseh črk abecede razen ene (za lažje preverjanje tega pogoja imamo spremenljivko stNeznanih, v kateri štejemo, za koliko znakov še ne poznamo šifre). V tem primeru lahko sklepamo tudi na šifro tiste ene preostale črke: njena šifra je edina črka, ki še ni šifra nobene druge črke (to je tista, pri kateri je pripadajoči element tabele original še vedno enak '*'). (Ta sklep je upravičen zato, ker besedilo naloge zagotavlja, da je kodirna funkcija bijektivna.)

Nazadnje se le še zapeljemo po nizu p2, kodiramo znake s pomočjo tabele sifra in jih izpisujemo.

```
#include <stdio.h>

void Desifriraj(const char *p1, const char *c1, const char *p2)
{
  char sifra[26], original[26]; int c, neznana, stNeznanih = 26;
  for (c = 0; c < 26; c++) sifra[c] = '*', original[c] = '*';
  for ( ; *p1; ++p1, ++c1) {
    /* Znaki, ki v c1 niso šifrirani, nam nič ne pomagajo. */
    if (*c1 == '*') continue;
    /* Mogoče smo zdaj šele prvič izvedeli, v kaj se šifrira trenutni znak niza p1
       in kaj se šifrira v trenutni znak niza c1. */
    if (sifra[*p1 − 'a'] == '*') sifra[*p1 − 'a'] = *c1, stNeznanih−−;
```

```
    if (original[*c1 − 'a'] == '*') original[*c1 − 'a'] = *p1;
    /* Mogoče pa smo za ta znak nekoč prej videli že neko drugo šifro ali za to
       šifro nek drug originalni znak; to pomeni, da je v vhodnih podatkih napaka. */
    if (sifra[*p1 − 'a'] != *c1 || original[*c1 − 'a'] != *p1) {
      printf("neveljavno sporočilo\n"); return; }}

  /* Če smo ugotovili šifre vseh črk abecede razen ene, potem tudi za tisto eno
     lahko sklepamo, v kaj se zašifrira. */
  if (stNeznanih == 1) {
    /* Poglejmo, za katero črko še ne poznamo šifre. */
    for (c = 0; c < 26; c++)
      if (sifra[c] == '*') { neznana = c; break; }
    /* V tabeli „original“ je zdaj ena sama črka, za katero še ne poznamo šifre,
       in ta mora biti šifra črke „neznana“. */
    for (c = 0; c < 26; c++)
      if (original[c] == '*') { sifra[neznana] = c + 'a'; break; }}

  /* Izpišimo šifrirano obliko niza p2. */
  while (*p2) putchar(sifra[*p2++ − 'a']);
  putchar('\n');
}
```

4. Potenciranje

Pomagajmo si z namigom iz besedila naloge. Števil a^2, a^4, a^8 in tako naprej ni težko računati z zaporednim kvadriranjem:

$$a^2 = a \cdot a, \quad a^4 = (a^2) \cdot (a^2), \quad a^8 = (a^4) \cdot (a^4)$$

in tako naprej. Kako pa s temi števili pridemo do poljubne potence a^b? Oglejmo si konkreten primer; recimo, da nas zanima $b = 87$. Število b lahko izrazimo kot vsoto nekaj potenc števila 2; v našem primeru je $87 = 64 + 16 + 4 + 2 + 1 = 2^6 + 2^4 + 2^2 + 2^1 + 2^0$. (To je pravzaprav isti razmislek, ki ga opravimo pri pretvorbi v dvojiški zapis: na primer, dvojiški zapis števila 87 je 1010111 — enice so ravno pri tistih potencah števila 2, ki jih moramo sešteti, da dobimo 87.) Spomnimo se, kaj velja za produkt vsote: $a^{b+c} = a^b \cdot a^c$. V našem primeru to pomeni, da je

$$a^{87} = a^{64+16+4+2+1} = a^{64} \cdot a^{16} \cdot a^4 \cdot a^2 \cdot a^1.$$

Tako torej vidimo, da lahko a^{87} izračunamo tako, da najprej z zaporednim kvadriranjem izračunamo $a^2, a^4, a^8, a^{16}, a^{32}, a^{64}$ in potem nekatere od teh potenc zmnožimo med sabo.

Zapišimo zdaj ta postopek za splošen b:

$c := 1;$
for $i := 0, 1, 2, \ldots$:
  if je bit i v dvojiškem zapisu števila b prižgan **then**
    $c := c \cdot a^{2^i}$;

Kako lahko z operacijami, ki jih imamo na voljo pri naši nalogi, čim preprosteje preverjamo, ali je bit i v številu b prižgan ali ne? Na primer, če izračunamo $b \bmod 2$, torej ostanek po deljenju b z 2, nam rezultat pove, ali je v b-ju prižgan bit 0 (najnižji

bit). Celi del količnika po deljenju b z 2 pa je pravzaprav število, ki ga dobimo, če b-ju najnižji bit odrežemo. V naši zanki lahko torej po vsaki iteraciji delimo b z 2, tako da nam bo po i iteracijah v bitu 0 pristal tisti bit, ki je bil na začetku na i-tem mestu. Zanka se ustavi, ko b pade na 0, saj takrat vemo, da so vsi višje ležeči biti b-ja ugasnjeni. Podobno tudi potence a^{2^i} računamo sproti preprosto tako, po vsaki iteraciji število a kvadriramo (pomnožimo s samim sabo). Tako smo prišli do naslednjega postopka:

```
c := 1;
while b > 0:
  if b mod 2 = 1 then
    c := c · a;
  a := a · a;
  b := ⌊b/2⌋; (* celi del količnika po deljenju z 2 *)
```

Tega postopka ni težko zapisati v jeziku, ki ga zahteva besedilo naloge. Namesto stavkov **while** in **if**, ki ju ta jezik nima, si bomo morali pomagati s pogojnimi skoki (ukaz JL):

		Razlaga (ni del programa)
	SUB c, c	postavi c na 0
	ADD c, 1	c je zdaj 1
zanka:	JL b, 1, konec	če je $b = 0$, končaj
	SUB t, t	postavi t na 0
	ADD t, b	postavi t na b
	MOD t, 2	trenutni bit števila b
	JL t, 1, preskok	če je ugasnjen, preskoči naslednji ukaz
	MUL c, a	pomnoži c s trenutno potenco števila a
preskok:	MUL a, a	pripravi naslednjo potenco števila a
	DIV b, 2	zamakni b za en bit navzdol
	JL 0, 1, zanka	skoči nazaj na začetek zanke
konec:		

Kot zanimivost si oglejmo še en podoben postopek za izračun potence a^b. Doslej smo bite b-ja gledali od nižjih proti višjih, lahko pa jih namesto tega gledamo od višjih proti nižjim. Zapišimo b po bitih kot $b = \sum_{i=0}^{k} b_i 2^i$; potem lahko a^b računamo takole:

```
c := 1;
for i := k, k − 1, . . . , 2, 1, 0:
  c := c · c;
  if b_i = 1 then c := c · a;
```

Prepričajmo se, da ta postopek na koncu v c res izračuna vrednost a^b. Označimo z B_i število, ki ga dobimo, če v b-ju porežemo vse bite od 0 do $i - 1$; torej je $B_i = \lfloor b/2^i \rfloor = \sum_{j=i}^{k} b_j 2^{j-i}$. Za ta števila velja $B_i = 2B_{i+1} + b_i$. Potem trdimo, da na začetku vsake iteracije naše zanke velja, da je $c = a^{B_{i+1}}$, na koncu te iteracije pa je $c = a^{B_i}$. Na začetku zanke, pri $i = k$, to drži, saj je $B_{k+1} = 0$ in zato $a^{B_{k+1}} = 1$, v c pa imamo takrat res vrednost 1. Nato pa v vsaki iteraciji zanke s tem, ko c

kvadriramo in morebiti (če je $b_i = 1$) še pomnožimo z a, iz vrednosti $a^{B_{i+1}}$ dobimo $a^{2B_{i+1}+b_i}$, to pa je ravno a^{B_i}. Na koncu zadnje iteracije (pri $i = 0$) nam torej v c nastane vrednost a^{B_0}, to pa je ravno a^b.

Ta rešitev izvede sicer prav toliko množenj kot prejšnja, vendar je zdaj pri množenjih oblike $c := c \cdot a$ eden od faktorjev razmeroma majhen (namreč a), medtem ko je prvotna rešitev vedno izvajala množenja, pri katerih sta oba faktorja velika (ker je množila z vrednostjo oblike a^{2^i}). Če imamo opravka z dovolj velikimi števili, bi se moralo prej ali slej izkazati, da je množenje z velikim številom dražje kot množenje z manjšim, zato je v takem primeru naša nova rešitev hitrejša. Je pa res, da je malo manj prikladna za implementacijo v tako omejenem zbirnem jeziku, kakršnega predpisuje naša naloga.

5. Vezja

Primeri, ko se dve povezavi sekata, nam določajo omejitve: takšni povezavi ne smeta biti obe na isti strani plošče; če je ena od njiju na sprednji strani plošče, mora biti druga na zadnji strani in obratno. Lahko torej začnemo pri poljubni povezavi in si izberemo, na kateri strani plošče bi bila; iz tega potem enolično sledi, kje morajo biti tiste povezave, ki se sekajo z našo: biti morajo pač na nasprotni strani plošče; in zato morajo biti tiste, ki se sekajo z *njimi*, spet na prvi strani plošče; in tako naprej. Pri tem si lahko pomagamo z vrsto, v katero dodajamo povezave, ki smo jim že določili stran, nismo pa še pregledali, kaj vse se seka z njimi (in mora biti zaradi tega na nasprotni strani).

Ko tako sledimo omejitvam, bomo prej ali slej bodisi narisali vse povezave bodisi ugotovili, da nas omejitve pripeljejo v protislovje (ker na primer neko povezavo narišemo na eno stran plošče, nato pa ugotovimo, da smo neko drugo povezavo, ki se seka z njo, že prej narisali na isto stran plošče).

Nalogo si lahko predstavljamo tudi kot problem na grafih. Sestavimo graf, ki ima po eno točko za vsako povezavo našega vezja; povezava med dvema točkama pa naj v našem grafu obstaja takrat, ko se pripadajoči povezavi vezja med seboj sekata (in torej ne smeta biti obe na isti strani plošče). Naša naloga ni zdaj nič drugega kot problem barvanja tega grafa z dvema barvama (vsaka barva predstavlja eno od strani plošče) in z običajno omejitvijo, da krajišči povezave ne smeta biti iste barve.

naj bo m število povezav in (z_i, k_i) naj bosta luknjici, ki ju povezuje i-ta povezava;
for $i := 1$ **to** m **do** $barva[i] := 0$;
for $i := 1$ **to** m **do**
  if $barva[i] > 0$ **then continue**; (* *Točka i je pobarvana že od prej.* *)
  $barva[i] := 0$;
  (* *Zdaj smo točki i določili barvo; iz tega pa enolično sledijo tudi barve njenih sosed, pa njihovih sosed in tako naprej.* *)
  naj bo Q prazna vrsta; dodaj i v Q;
  while Q ni prazna:
    vzemi poljubno u iz vrste Q;
    for $v := 1$ **to** m:
      if not SeSekata$(x[z_u], y[z_u], x[k_u], y[k_u], x[z_v], y[z_v], x[k_v], y[k_v])$ **then continue**; (* *barva u-ja ne vpliva neposredno na barvo v-ja* *)

```
        if barva[v] = barva[u] then return false; (* grafa ni mogoče pobarvati *)
        if barva[v] > 0 then continue; (* v je že primerno pobarvan *)
        barva[v] := 3 − barva[u]; dodaj v v Q;
return true;
```

Če nam postopek na koncu vrne **true**, lahko s pomočjo tabele *barva* vezje tudi narišemo; tiste povezave, ki imajo $barva[i] = 1$, narišemo na eno stran plošče, ostale pa na drugo stran plošče.

Za namene našega tekmovanja je gornji postopek čisto dovolj dober, dalo pa bi se ga še malo izboljšati: v notranji zanki gremo zdaj z v po vseh povezavah in za vsako preverjamo, ali se seka s povezavo u; ker moramo to storiti za vsako povezavo u, je časovna zahtevnost tega postopka $O(m^2)$. Obstajajo algoritmi, ki vse pare sekajočih se daljic poiščejo hitreje (če takih parov ni veliko), na primer Bentley–Ottmannov algoritem, ki porabi $O((m + s) \log m)$ časa, če je s število parov sekajočih se daljic.

REŠITVE NALOG ZA TRETJO SKUPINO

1. Ljudožerci na premici

Koordinate ljudožercev bomo hranili v naraščajočem vrstnem redu. (Ker v vhodni datoteki niso urejeni, jih bomo na začetku pred nadaljnjo obdelavo uredili naraščajoče.) Ko pride nov padalec na koordinato a_j, lahko v našem urejenem seznamu ljudožercev z bisekcijo poiščemo najbližjega ljudožerca. Naloga pravi, da če na isti točki stoji več ljudožercev, bo šel k našemu padalcu tisti, ki ima najnižjo zaporedno številko, vendar je ta omejitev čisto nebistvena, saj bomo morali na koncu ljudožerce izpisati urejene po koordinatah, ne po zaporedni številki. Zato lahko v primeru, ko recimo a_j pade med $x[i]$ in $x[i+1]$, brez slabe vesti k padalcu pošljemo enega od teh dveh ljudožercev (tistega, ki je bližje; če sta oba enako daleč od a_j, pošljemo levega) in se ne ukvarjamo s tem, ali ima mogoče še več prejšnjih ali naslednjih ljudožercev isto koordinato. Tako moramo po vsakem padalcu popraviti le en element našega seznama koordinat ljudožercev in ta seznam po takem popravku tudi ostane urejen.

Ker bisekcija na tabeli n elementov porabi $O(\log n)$ časa, je časovna zahtevnost celotnega postopka $O((n+m)\log n)$.

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

#define MaxN 100000
int n, x[MaxN];

int PoisciNajblizjega(int a)
{
   int l, d, m;
   if (a <= x[0]) return 0; /* Poseben primer, če je padalec levo od vseh ljudožercev. */
   l = 0; d = n;
   while (d − l > 1)
   {
      /* Na tem mestu velja x[l] < a ≤ x[d]. (Pri d = n si mislimo x[d] = ∞.) */
      m = (l + d) / 2;
      if (x[m] < a) l = m; else d = m;
   }
   /* Tu velja x[d − 1] < a ≤ x[d].
      Poglejmo, kateri od ljudožercev d − 1 in d je bližji padalcu a. */
   return d < n && x[d] − a < a − x[d − 1] ? d : d − 1;
}

/* Primerjalna funkcija za urejanje ljudožercev po koordinati. */
int Primerjaj(const void *a, const void *b) {
   return *(const int *) a − *(const int *) b; }

int main()
{
   int i, j, m, a;
   FILE *f = fopen("ljudozerci.in", "rt");
   fscanf(f, "%d %d", &n, &m);

   /* Preberimo koordinate ljudožercev in jih uredimo naraščajoče. */
   for (i = 0; i < n; i++) fscanf(f, "%d", &x[i]);
   qsort(x, n, sizeof(x[0]), &Primerjaj);

   /* Obdelajmo padalce. */
```

```
  for (j = 0; j < m; j++)
  {
    fscanf(f, "%d", &a); /* Preberimo naslednjega padalca. */
    i = PoisciNajblizjega(a); /* Poiščimo najbližjega ljudožerca... */
    x[i] = a; /* ...in ga postavimo na mesto, kjer je pristal padalec. */
  }
  fclose(f);

  /* Izpišimo končni položaj ljudožercev. */
  f = fopen("ljudozerci.out", "wt");
  for (i = 0; i < n; i++) fprintf(f, "%d\n", x[i]);
  fclose(f);
  return 0;
}
```

Razmislimo še o različici naloge, pri kateri moramo ljudožerce v izpisu urediti po zaporednih številkah, ne po koordinatah. Pri tej različici zdaj v primeru, ko na isti koordinati stoji več ljudožercev, ni več vseeno, katerega izmed njih premaknemo k našemu padalcu; zdaj se moramo dosledno držati omejitve iz besedila naloge, da mora iti k padalcu tisti ljudožerec, ki ima najnižjo zaporedno številko.

Ko pride na premico nov padalec, lahko pravega ljudožerca načeloma še vedno poiščemo z bisekcijo, paziti moramo le na to, da seznam ljudožercev zdaj ni le urejen naraščajoče po koordinati, pač pa da so ljudožerci z enako koordinato urejeni naraščajoče po zaporedni številki. Tako med ljudožerci z enako koordinato ne bo težko najti tistega z najnižjo zaporedno številko. Zaplete pa se pri popravljanju tega seznama: recimo, da imamo na točki x cel kup ljudožercev z zaporednimi številkami $z_1, \ldots, z_t$ (v naraščajočem vrstnem redu); in recimo, da naslednji padalec pade na a_j, ki leži nekje desno od točke x in to tako, da so mu ljudožerci na točki x najbližji. V tem primeru se bo ljudožerec z_1 premaknil s koordinate x na a_j, torej bo po novem desno od ljudožercev $z_2, \ldots, z_t$. V seznamu se torej podzaporedje $z_1, z_2, \ldots, z_t$ spremeni v $z_2, \ldots, z_t, z_1$. Če je seznam predstavljen s tabelo (*array*), nam bo ta popravek vzel $O(t)$ časa, kar je v najslabšem primeru $O(n)$; ker se to lahko zgodi pri vsakem padalcu, bo časovna zahtevnost naše rešitve zdaj $O(nm)$ namesto dosedanje $O(m \log n)$. Če bi bil seznam predstavljen z verigo (*linked list*), bi nam premik ljudožerca vzel le $O(1)$ časa, vendar na takšnem seznamu ne bi mogli učinkovito izvajati bisekcije.

Boljša rešitev je, da ljudožerce predstavimo z binarnim iskalnim drevesom (po možnosti uravnoteženim, npr. rdeče-črnim ali pa AVL-drevesom), v katerem bodo urejeni po koordinati, tisti z enako koordinato pa še po zaporedni številki. Iskanje primernega ljudožerca (ko pride nov padalec) nam vzame v takem drevesu $O(\log n)$ časa, premik ljudožerca na novo koordinato pa lahko izvedemo tako, da ga pobrišemo iz drevesa in nato ponovno dodamo (z novo koordinato), kar tudi vzame $O(\log n)$ časa. Začetno gradnjo drevesa lahko izpeljemo tako, da začnemo s praznim drevesom in vanj dodamo vse ljudožerce enega za drugim (z njihovimi začetnimi koordinatami); vsako tako dodajanje tudi vzame $O(\log n)$ časa. Časovna zahtevnost celotne rešitve je tako $O((n+m) \log n)$.

Za končni izpis ljudožercev (po zaporednih številkah) si lahko pomagamo s tabelo T, v katero bomo vpisali končni položaj vseh ljudožercev. Sprehodimo se po vseh ljudožercih našega drevesa in ko smo pri ljudožercu z zaporedno številko z in položajem x, si jo zapišimo kot $T[z] := x$. Na koncu tega sprehoda imamo v T

položaje vseh ljudožercev in jih lahko preprosto izpišemo z zanko po z.

Z nekaj pazljivosti pa lahko seznam ljudožercev vseeno predstavimo tudi s tabelo namesto z drevesom. Recimo, da imamo spet tabelo, v kateri so ljudožerci urejeni naraščajoče po koordinati; ni pa nujno, da so tisti z isto koordinato urejeni po zaporedni številki (saj bi bilo takšen vrstni red, kot smo videli zgoraj, pretežko vzdrževati). Z bisekcijo ni težko v času $O(\log n)$ poiskati v naši tabeli najbolj levega in najbolj desnega ljudožerca z neko koordinato x, ki nas zanima. Tako torej lahko ugotovimo, kateri del naše tabele pokrivajo ljudožerci s koordinato x: recimo, da stojijo na indeksih od i do j (za neka $1 \leq i \leq j \leq n$).

Ko pride nov padalec, ki mu je skupina ljudožercev na koordinati x najbližja, se mora iz te skupine ljudožerec z najmanjšo zaporedno številko premakniti k padalcu. Recimo, da je bil ta ljudožerec na indeksu k (za nek k, pri čemer je $i \leq k \leq j$). Če je padalec pristal levo od x, se mora torej ljudožerec z indeksa k premakniti na indeks i, tistega z indeksa i pa lahko premaknemo na k; naša skupina ljudožercev na koordinati x tako zdaj pokriva le še indekse od $i + 1$ do j. Podobno pa, če je padalec pristal desno od x, se mora ljudožerec z indeksa k premakniti na j, tistega z j pa lahko premaknemo na k; naša skupina zdaj pokriva indekse od i do $j - 1$.

Vprašanje je le, kako priti do k — torej kako ugotoviti, kateri ljudožerec v skupini ima najmanjšo zaporedno številko. Dogovorimo se, da bomo ta podatek hranili pri ljudožercu na začetku skupine, torej na indeksu i. Če se levi rob skupine premakne z i na $i + 1$ (ker je padalec pristal levo od x), bomo pač tudi ta podatek premaknili v celico $i + 1$. Poleg tega, ko ljudožerec k odide k padalcu in s tem zapusti našo skupino na koordinati x, bomo morali imeti pri roki podatek o tem, kateri ljudožerec v skupini ima zdaj po novem najmanjšo zaporedno številko; torej je koristno, če imamo pri vsakem ljudožercu zapisano še to, kateri je njegov neposredni naslednik (po zaporedni številki) med ljudožerci z isto koordinato kot on.

Tako torej vidimo, da bomo pravzaprav potrebovali več tabel. Opozorimo na to, da moramo v naslednjem razmisleku pazljivo razlikovati med *zaporedno številko* ljudožerca (ki izhaja iz vhodnih podatkov in se ne more spremeniti) ter njegovim *indeksom* v našem seznamu (ta pa se bo spreminjal). $X[i]$ naj pove koordinato i-tega ljudožerca v seznamu (kot rečeno, je seznam urejen po koordinatah, tako da velja $X[1] \leq X[2] \leq \ldots \leq X[n]$), $Z[i]$ pa njegovo zaporedno številko; $I[z]$ naj pove indeks ljudožerca z zaporedno številko z (torej, če je $Z[i] = z$, je $I[z] = i$ in obratno); poleg tega pa bomo imeli še dve tabeli, ki nam ljudožerce z isto koordinato povežeta v seznam, urejen naraščajoče po zaporedni številki: $P[i]$ naj bo najmanjša zaporedna številka med ljudožerci s koordinato $X[i]$, $N[z]$ pa naslednja najmanjša (za z) zaporedna številka ljudožerca s koordinato $X[I[z]]$.[11] (Bolj formalno bi to zapisali takole: $P[i] = \min\{z : X[I[z]] = X[i]\}$ in $N[z] = \min\{z' : X[I[z']] = X[I[z']] \wedge z' > z\}$.) Pri tem pa bo tabela P vsebovala veljavne vrednosti le za tiste i, ki predstavljajo prvega ljudožerca z neko koordinato, torej le če je $X[i] > X[i-1]$ (ali pa $i = 1$).

Začetnega stanja teh tabel ni težko pripraviti; ljudožerce uredimo po koordinati, tiste z enako koordinato pa naraščajoče po zaporednih številkah. V tem vrstnem redu jih zložimo v X in Z; tabele I ni težko pripraviti iz Z; tabeli P in N pa

[11] Tabeli P in N hranita zaporedne številke ljudožercev, ne njihovih indeksov, ker se zaporedne številke nič ne spreminjajo. Zato pa potem potrebujemo tabelo I, da vemo, kje v seznamu se nahaja posamezni ljudožerec (z znano zaporedno številko).

si pomagata z dejstvom, da so ljudožerci z isto koordinato urejeni po zaporedni številki:

for $i := 1$ **to** n **do** $X[i] := x_i$; $Z[i] := i$;
uredi pare $(X[i], Z[i])$ v naraščajočem vrstnem redu;
for $i := 1$ **to** n:
  $I[Z[i]] := i$; $P[i] := i$;
  if $i < n$ **and** $X[i] < X[i+1]$
    then $N[Z[i]] := Z[i+1]$
    else $N[Z[i]] :=$ NIL;

Ko pride nov padalec (recimo, da pristane na koordinati a), lahko z bisekcijo pogledamo, na katerih indeksih stoji najbližja skupina ljudožercev; recimo, da pokriva ta skupina indekse od i do j. Če je $X[i] = a$, se noben ljudožerec ne premakne in lahko takoj nadaljujemo z naslednjim padalcem;[12] če je $i = j$, je najbližji ljudožerec en sam in mu moramo le popraviti $X[i]$ na a. Če pa je $i < j$ (torej ima skupina več kot enega ljudožerca), razmišljajmo takole:

$z_p := P[i]$; $k := I[z_p]$; $X[k] := a$;
$z_n := N[z_p]$; $N[z_p] :=$ NIL;
if $a < X[i]$ **then** SWAP(i, k); $i := i + 1$
        else SWAP(k, j);
$P[i] := z_n$;

Najprej torej določimo, kateri ljudožerec v naši skupini ima najmanjšo zaporedno številko: to je z_p na indeksu k; nato poiščemo naslednjega (po zaporedni številki), recimo mu z_n. Ljudožerec z_p se mora nato z indeksa k premakniti na i (če je padalec levo od skupine) oz. na j (če je padalec desno s skupine), torej ga zamenjamo s tistim, ki je bil prej na tem indeksu, in mu postavimo x-koordinato na a. Po novem se torej skupina začne na $i+1$ (če je bil padalec levo od skupine) oz. še vedno na i (če je bil padalec desno od skupine), zato pri tem indeksu popravimo P, da kaže na novega prvega člana skupine (z_n).

Zdaj potrebujemo le še podprogram, ki zamenja dva ljudožerca v tabeli:

podprogram SWAP(i_1, i_2):
  $t := X[i_1]$; $X[i_1] := X[i_2]$; $X[i_2] := t$;
  $z_1 := Z[i_1]$; $z_2 := Z[i_2]$;
  $Z[i_1] := z_2$; $Z[i_2] := z_1$;
  $I[z_1] := i_2$; $I[z_2] := i_1$;

Časovna zahtevnost te rešitve je $O((n+m)\log n)$, zaradi urejanja ljudožercev na začetku in zaradi bisekcije pri vsakem padalcu; lepo pri njej pa je, da namesto drevesa potrebuje le nekaj tabel.

[12] Koristna posledica tega je, da se skupine ljudožercev z enako x-koordinato lahko le zmanjšujejo, nikoli ne povečujejo: če se nek ljudožerec premakne z x na x', se skupina ljudožercev na x zmanjša, nobena skupina pa se ne poveča, saj na x' od prej gotovo ni bilo nobenega ljudožerca — če bi bil, bi kar on pojedel našega padalca, zato se ljudožerec z x sploh ne bi premikal. Z drugimi besedami, vsak ljudožerec, ko se enkrat premakne, je na novem položaju sam in odtlej tudi ostane sam.

2. Po Indiji z avtobusom

Ko razmišljamo, na kateri avtobus stopiti, se splača izbrati (med vsemi, ki peljejo mimo naše trenutne točke) tistega, ki pelje najdlje (torej čigar končna postaja leži najbolj desno). O tem se lahko prepričamo takole: recimo, da smo v točki s in da ima avtobus, izbran na opisani način, končno postajo t; in recimo, da boljšo rešitev dobimo, če ne stopimo na ta avtobus, pač pa na nekega drugega, ki ima končno postajo $u < t$. V tisti optimalni rešitvi prej ali slej sestopimo s tega novega avtobusa — očitno nekje na območju $(s, u]$, saj dlje ta avtobus ne pelje. Toda to območje v celoti prevozi tudi avtobus, ki pelje do t, torej bi se lahko peljali tudi z njim in še vedno nadaljevali pot enako, kot bi jo pri naši optimalni rešitvi. Tako torej vidimo, da optimalne rešitve gotovo ne bomo spregledali, če bomo pri vsakem koraku stopili na tisti avtobus, ki nas pripelje najdlje. S podobnim razmislekom se lahko prepričamo, da ni pametno sestopiti z avtobusa prej kot na njegovi končni postaji (razen seveda če nas ta avtobus že prej pripelje do našega cilja y).

Tako smo dobili naslednji požrešni algoritem:

```
naj bo x naš začetni položaj, y pa naš cilj;
štVoženj := 0;
while x < y:
   med vsemi avtobusi, ki peljejo mimo točke x, izberi tistega z
      najbolj desno končno postajo; recimo mu i;
   x := b_i;
   štVoženj := štVoženj + 1;
```

Razmislimo, kako lahko na vsakem koraku učinkovito poiščemo pravi avtobus. Zanimajo nas le tisti, ki se peljejo mimo točke x, torej za katere je $a_i \leq x \leq b_i$. Drugi del tega pogoja, torej $x \leq b_i$, je pravzaprav odveč; med vsemi avtobusi z $a_i \leq x$ bomo tako ali tako izbrali tistega z največjim b_i; če niti ta ne leži desno od x, potem sploh noben avtobus ne pelje dlje kot do x, torej je problem nerešljiv (naloga pa zagotavlja, da se to ne bo zgodilo). Tako nam ostane pravzaprav pogoj, da med vsemi avtobusi, ki imajo $a_i \leq x$, izberemo tistega z največjim b_i. V vsaki iteraciji glavne zanke se x malo poveča, zato pogoju $a_i \leq x$ ustreza vse več avtobusov (in je tudi b_i lahko vse večji). Vidimo torej, da je koristno avtobuse urediti naraščajoče po a_i; tako bomo lahko po vsakem premiku nadaljevali s pregledovanjem tam, kjer smo prej končali (in bomo na novo pregledali le tiste avtobuse, na katere zdaj lahko stopimo, pred zadnjim premikom pa še nismo mogli).

```
uredi avtobuse naraščajoče po a_i;
štVoženj := 0; i := 1;
while x < y:
   b := x;
   while i ≤ n and a_i ≤ x:
      if b_i > b then b := b_i;
      i := i + 1;
   x := b;
   štVoženj := štVoženj + 1;
```

Ta postopek je prijetno učinkovit, saj ima notranja zanka vsega skupaj (po vseh iteracijah zunanje zanke) le $O(n)$ iteracij, za vsak avtobus po eno. Največ časa,

$O(n \log n)$, tako porabimo za urejanje avtobusov na začetku postopka. Zapišimo našo rešitev še v C-ju:

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

#define MaxN 200000

typedef struct { int z, k; } Proga;
Proga proge[MaxN];

/* Primerjalna funkcija za urejanje avtobusov po začetni postaji. */
int Primerjaj(const void *a, const void *b) {
  return ((const Proga *) a)->z - ((const Proga *) b)->z; }

int main()
{
  int i, n, x, y, stVozenj, doKod;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("avtobus.in", "rt");
  fscanf(f, "%d %d %d", &n, &x, &y);
  for (i = 0; i < n; i++) fscanf(f, "%d %d", &proge[i].z, &proge[i].k);
  fclose(f);

  /* Uredimo avtobuse po začetni postaji. */
  qsort(proge, n, sizeof(proge[0]), &Primerjaj);

  stVozenj = 0; i = 0;
  while (x < y)
  {
    /* Trenutno se nahajamo na postaji x. Med avtobusi, ki vozijo mimo nje,
       poiščimo tistega z najbolj desno končno postajo. */
    doKod = -1;
    for (doKod = -1; i < n && proge[i].z <= x; i++)
      if (proge[i].k > doKod) doKod = proge[i].k;
    stVozenj++; x = doKod;
  }

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

3. Luči

Za vsako stikalo imamo načeloma dve možnosti: lahko ga pritisnemo enkrat ali pa nobenkrat. Vendar pa stikala med seboj niso neodvisna; če si izberemo stanje nekega stikala u (torej ali je pritisnjeno ali ne), potegne to za sabo posledice za luči, ki so povezane na u, in s tem tudi za druga stikala, ki so povezana z istimi lučmi kot u.

Na primer, če je neka luč l prižgana in je povezana samo s stikalom u (in nobenim drugim), potem u *moramo* pritisniti; če pa je taka luč ugasnjena, potem u *ne smemo* pritisniti. Podobno, če je luč l prižgana in povezana s stikaloma u in v, potem moramo pritisniti natanko eno od stikal u in v, drugega pa ne; če pa je taka luč ugasnjena, potem moramo pritisniti bodisi obe stikali ali pa nobeno od njiju.

Začnemo lahko pri poljubnem stikalu u in se na primer vprašamo, kaj se zgodi, če to stikalo pritisnemo. Oglejmo si luči, ki so povezane z u; s pomočjo omejitev iz prejšnjega odstavka lahko zdaj določimo, v kakšnem stanju morajo biti druga stikala, ki so povezana s temi lučmi; in ker smo zdaj določili stanje teh stikal, lahko v nadaljevanju pregledamo še njihove ostale luči in tako naprej. Pri tem pregledovanju se lahko zgodi, da pridemo v protislovje (npr. za neko stikalo nam ena luč zahteva, da mora biti pritisnjeno, neka druga luč pa, da ne sme biti biti pritisnjeno); tedaj vemo, da naša začetna odločitev, da pritisnemo stikalo u, ni sprejemljiva. Če pa se ta postopek ustavi (ko pregleda vse luči in stikala, dosegljiva iz u), ne da bi prišel v protislovje, to pomeni, da so kombinacije, pri katerih je u pritisnjeno, načeloma možne; imajo pa potem vsa druga stikala, ki smo jih dosegli iz u, že enolično določeno stanje in si pri njih ne moremo več izbirati, kaj bi naredili z njimi.

V nadaljevanju lahko podoben razmislek ponovimo še za možnost, da u-ja ne pritisnemo. Tako ugotovimo, koliko stanj u-ja je možnih (lahko sta obe, eno ali pa celo nobeno od njiju).

Videli smo, da je stanje vseh stikal, ki so dosegljiva iz u, enolično določeno, čim si izberemo stanje u-ja. Pač pa je mogoče, da obstajajo še kakšna druga stikala, ki iz u-ja niso dosegljiva; tista pa so neodvisna od u in lahko zdaj isti razmislek ponovimo še pri njih. Na koncu moramo rezultate za takšna neodvisna stikala pomnožiti med sabo (če imamo dve možnosti za stanje stikala u, pa dve možnosti za u' in dve možnosti za neko še tretje neodvisno stikalo u'', nam dá to vsega skupaj $2 \times 2 \times 2$ možnih kombinacij stanj stikal).

Nalogo si lahko predstavljamo kot problem na grafih; stikala in luči tvorijo točke našega grafa. Gornji razmislek ne pomeni nič drugega kot to, da za vsako povezano komponento tega grafa posebej ugotovimo, ali je možnih stanj te komponente 0, 1 ali 2, in te rezultate potem pomnožimo med sabo.

Poseben primer so še luči, ki niso povezane z nobenim stikalom. Če je taka luč na začetku ugasnjena, nas ne moti; če pa je na začetku prižgana, je ne bomo mogli ugasniti z nobeno kombinacijo stikal, zato bo moral naš postopek vrniti 0.

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

#define MaxL 300
#define MaxS 300
#define M 1000000007

int L, S;
/* Podatki o stikalih: številke luči, s katerimi je povezano stikalo s,
   so v tabeli neighS na indeksih od firstS[s] do firstS[s] + degS[s] − 1]. */
int firstS[MaxS], degS[MaxS], neighS[MaxL * 2];
/* Podatki o lučeh; številke stikal, s katerimi je povezana luč l,
   so v tabeli stikala[l] na indeksih od 0 do degL[l] − 1. */
int prizgana[MaxL], degL[MaxL], stikala[MaxL][2];
/* Naslednje tabele uporabljamo med pregledovanjem grafa.
   stanje[s] pove, ali bi stikalo s pritisnili ali ne (0 ali 1);
   če pa mu stanja še nismo določili, imamo stanje[s] = −1. */
int stanje[MaxS], vrsta[MaxS], glava, rep;
bool obdelano[MaxS];

/* Naslednji podprogram preveri, če smemo stikalo u0 postaviti v stanje s. */
```

```
bool Preizkusi(int u0, int s)
{
  int u, v, i, luc;

  /* Postavimo u0 v stanje s in ga dodajmo v vrsto. */
  stanje[u0] = s; obdelano[u0] = true;
  glava = 0; rep = 1; vrsta[glava] = u0;

  /* Preglejmo vse, kar je dosegljivo iz tega stikala. */
  while (glava < rep)
  {
    u = vrsta[glava++]; /* Vzemimo naslednje stikalo iz vrste. */
    /* Stikalu u smo že določili stanje; kaj to pomeni za luči, s katerimi je povezano? */
    for (i = 0; i < degS[u]; i++)
    {
      luc = neighS[firstS[u] + i];

      if (degL[luc] == 1) {
        /* Ta luč je priklopljena samo na u. Preverimo, če bo na koncu ugasnjena. */
        if (stanje[u] ^ prizgana[luc]) return false; }

      else
      {
        v = stikala[luc][stikala[luc][0] == u ? 1 : 0];
        /* Ta luč je priklopljena na u in še na neko drugo stikalo v.
            Če v-ju še nismo določili stanja, mu ga lahko določimo zdaj. */
        if (stanje[v] < 0)
          stanje[v] = prizgana[luc] ^ stanje[u],
          vrsta[rep++] = v, obdelano[v] = true;

        /* Če pa ima v stanje že od prej, lahko preverimo, če bo luč na koncu res */
        else if (stanje[v] ^ stanje[u] ^ prizgana[luc]) return false;    /* ugasnjena. */
      }
    }
  }
  return true;
}

int main()
{
  FILE *fi = fopen("luci.in", "rt");
  FILE *fo = fopen("luci.out", "wt");
  int T, luc, i, stikalo, nMoznosti, nPovezav, rezultat;

  fscanf(fi, "%d", &T);
  while (T-- > 0)
  {
    /* Preberimo naslednji testni primer. */
    fscanf(fi, "%d %d", &L, &S);
    nPovezav = 0;
    for (luc = 0; luc < L; luc++) {
      fscanf(fi, "%d", &prizgana[luc]); degL[luc] = 0; }
    for (stikalo = 0; stikalo < S; stikalo++) {
      fscanf(fi, "%d", &degS[stikalo]);
      stanje[stikalo] = -1; obdelano[stikalo] = false;
      for (i = 0, firstS[stikalo] = nPovezav; i < degS[stikalo]; i++) {
        fscanf(fi, "%d", &luc); luc--;
        neighS[nPovezav++] = luc; stikala[luc][degL[luc]++] = stikalo; }}

    rezultat = 1;
    /* Preverimo, če je kakšna luč prižgana in ni povezana z nobenim stikalom. */
```

```
    for (luc = 0; luc < L; luc++)
      if (prizgana[luc] && degL[luc] == 0) { rezultat = 0; break; }
    /* Preglejmo zdaj vsa stikala. */
    for (stikalo = 0; stikalo < S && rezultat > 0; stikalo++)
    {
      if (obdelano[stikalo]) continue;
      /* Poskusimo, ali je dopustno, da tega stikala ne pritisnemo. */
      nMoznosti = 0;
      if (Preizkusi(stikalo, 0)) nMoznosti++;
      /* Pobrišimo stanje stikal, ki smo jih dosegli pri tem poskusu. */
      for (glava = 0; glava < rep; glava++) stanje[vrsta[glava]] = -1;
      /* Poskusimo še, ali je dopustno, da to stikalo pritisnemo. */
      if (Preizkusi(stikalo, 1)) nMoznosti++;
      rezultat = (rezultat * nMoznosti) % M;
    }

    fprintf(fo, "%d\n", rezultat); /* Izpišimo rezultat. */
  }
  fclose(fi);
  fclose(fo);
  return 0;
}
```

4. Bloki

Definicije blokov so odvisne le od zamika posameznih vrstic (števila presledkov na začetku vrstice), ne pa od preostanka vsebine v vrstici. Zato lahko že ob branju vhodnih podatkov izračunamo zamik vsake vrstice in ga shranimo v neki tabeli; v nadaljevanju bomo delali le s to tabelo, vhodno besedilo pa sproti pozabljali. V spodnjem programu imamo v ta namen tabelo zamik; prazne vrstice (in vrstice, ki vsebujejo same presledke) so predstavljene z zamikom -1.

Tabelo zamikov pregledujmo po vrsti, od prve vrstice proti zadnji; spremenljivka i nam pove indeks trenutne vrstice. Koristno je imeti pri roki tudi zamik prejšnje neprazne vrstice (spremenljivka prejZamik), saj ga bomo potrebovali za ugotavljanje, kje se začne blok. Ko pridemo do neprazne vrstice, moramo preveriti naslednje:

- Če je zamik te vrstice večji od zamika prejšnje neprazne vrstice, se tu začne nov blok. Preostanek tega bloka bomo pregledali z rekurzivnim klicem, ob vrnitvi iz njega pa nam bo indeks trenutne vrstice povedal, kje se ta blok konča. Ta podatek si zapomnimo v tabeli blokDo, da bomo lahko na koncu izpisali bloke v izhodno datoteko.

- Če pa je zamik trenutne vrstice manjši od zamika trenutnega bloka, se ta blok tukaj konča in se moramo vrniti iz trenutnega rekurzivnega klica.

Kljub rekurziji je ta postopek učinkovit, saj se indeks i ves čas le povečuje in po vrnitvi iz rekurzivnega klica nadaljujemo s pregledovanjem tam, kjer je ta rekurzivni klic končal. Časovna zahtevnost tega postopka je le $O(n)$.

Šlo bi tudi brez rekurzije, vendar bi potem morali zamike trenutno odprtih blokov hraniti v nekakšnem seznamu, ki bi imel enako vlogo kot sklad, na katerem se pri rekurziji hranijo parametri posameznih vgnezdenih rekurzivnih klicev.

```
#include <stdio.h>

#define MaxN 100000
#define MaxDolz 1000

int i, n, zamik[MaxN], blokDo[MaxN], prejZamik;

void PoisciBloke(int zamikBloka)
{
   int razlika, blokOd;
   while (i < n)
   {
      /* Prazne vrstice preskočimo. */
      if (zamik[i] < 0) { i++; continue; }

      /* Zapomnimo si razliko glede na prejšnji zamik; nato pa
         zamik trenutne vrstice shranimo v prejZamik, kjer bo prišel
         prav v nadaljevanju. */
      razlika = zamik[i] - prejZamik;
      prejZamik = zamik[i];

      /* Če je zamik manjši kot zamik trenutnega bloka, se blok konča. */
      if (zamik[i] < zamikBloka) break;

      /* Če je zamik vsaj tolikšen kot v prejšnji vrstici, se blok nadaljuje. */
      else if (razlika <= 0) i++;

      /* Če je zamik večji kot v prejšnji vrstici, se začne nov vgnezden blok. */
      else /* if (razlika > 0) */
      {
         blokOd = i; i++;
         PoisciBloke(prejZamik); /* Poglejmo, do kod se razteza ta vgnezdeni blok. */
         blokDo[blokOd] = i - 1; /* Zapomnimo si ta blok v tabeli blokDo. */
      }
   }
}

int main()
{
   int z; char s[MaxDolz + 2];

   /* Preberimo vhodno datoteko. */
   FILE *f = fopen("bloki.in", "rt");
   fgets(s, sizeof(s), f); sscanf(s, "%d", &n);
   for (i = 0; i < n; i++)
   {
      fgets(s, sizeof(s), f);

      /* Določimo zamik trenutne vrstice. */
      z = 0; while (s[z] == ' ') z++;
      if (! s[z] || s[z] == '\r' || s[z] == '\n') z = -1;
      zamik[i] = z; blokDo[i] = -1;
   }
   fclose(f);

   prejZamik = -1; i = 0; PoisciBloke(-1);

   /* Izpišimo rezultate. */
   f = fopen("bloki.out", "wt");
   for (i = 0; i < n; i++)
      if (blokDo[i] >= 0)
         fprintf(f, "%d %d\n", i + 1, blokDo[i] + 1);
```

```
  fclose(f);
  return 0;
}
```

5. Poravnavanje desnega roba

Vprašanje, kako najbolje razbiti besedilo na vrstice, lahko razdelimo na dva podproblema: (1) izbrati si moramo, koliko besed bi vzeli v prvo vrstico (recimo prvih k); (2) potem pa moramo najti še najboljše razbitje preostalega besedila. Vidimo lahko, da je podproblem (2) pravzaprav enak prvotnemu, le da ima malo krajše besedilo: namesto besed $w_1, \ldots, w_n$ gledamo le besede $w_{k+1}, \ldots, w_n$. Ko bi reševali ta podproblem, bi znotraj njega našli podobne podprobleme s še krajšim besedilom.

Označimo torej s $f(i)$ oceno najboljšega razbitja besed $w_i, w_{i+1}, \ldots, w_n$. (Rezultat, po katerem sprašuje naloga, je potem $f(1)$.) Razmislek iz prejšnjega odstavka nam je torej pokazal, da velja

$$f(i) = \min\{ \mathit{ocena}(i, j) + f(j+1) : i \leq j \leq n \}.$$

Z drugimi besedami, če vzamemo v prvo vrstico besede od w_i do w_j, nam ostane podproblem z besedami $w_{j+1}, \ldots, w_n$; ocena prve vrstice je potem $\mathit{ocena}(i, j)$, ocena preostanka (če ta preostanek razbijemo na vrstice na najboljši možni način) pa $f(j+1)$. V gornji enačbi smo napisali $i \leq j \leq n$, vendar smemo iti v resnici z j le tako daleč, dokler še lahko spravimo vse besede od w_i do w_j v eno samo vrstico.

Funkcijo f bi lahko računali z rekurzivnim podprogramom, ki bi klical samega sebe, da bi pri izračunu vrednosti $f(i)$ izračunal vrednosti $f(j+1)$ za razne vrednosti j. Pri tem bi večkrat prišlo do rekurzivnih klicev z enako vrednostjo parametra, zato je koristno, če si že izračunane rezultate zapomnimo v neki tabeli, da jih ne bo treba računati po večkrat. Če računamo $f(i)$ po padajoči vrednosti i-ja, bomo vedno imeli že izračunane vse rešitve manjših podproblemov, ki jih bomo potrebovali za izračun $f(i)$. Zato rekurzivne funkcije niti ne potrebujemo več in lahko funkcijo računamo sistematično z zanko po i. Tako smo dobili naslednji postopek:

$f[n+1] := 0$;
for $i := n$ **downto** 1:
  $j := i$; $f[i] := \infty$;
  while $j \leq n$ **and** besede od i do j gredo lahko v eno vrstico:
    $f[i] := \min\{f[i], \mathit{ocena}(i, j) + f[j+1]\}$;
    $j := j + 1$;

Razmislimo še o tem, kako bi učinkovito preverjali, ali gredo besede od i do j še lahko v eno vrstico. Širina take vrstice bi bila $w_i + w_{i+1} + \ldots + w_j + (j-i) \cdot s$; preveriti moramo torej, če je ta vsota $\leq d$. Ista vsota bo prišla prav tudi pri izračunu ocene te vrstice. V vsaki iteraciji, ko se j poveča za 1, pridobi vsota $w_i + w_{i+1} + \ldots + w_j$ na koncu en nov člen, torej nove vsote ni težko računati iz prejšnje; v spodnji rešitvi hrani dosedanjo širino vrstice spremenljivka sirina, ki ji v vsaki iteraciji prištejemo dolžino naslednje besede (in presledka s).

```
#include <stdio.h>
#define MaxN 1000000
```

```
long long wi[MaxN + 1], f[MaxN + 1];

int main()
{
  int i, j, n; long long s, d, sirina, kand;

  /* Preberimo vhodne podatke. */
  FILE *g = fopen("poravnavanje.in", "rt");
  fscanf(g, "%lld %lld %lld", &n, &s, &d);
  for (i = 0; i < n; i++) fscanf(g, "%lld", &wi[i]);
  fclose(g);

  f[n] = 0;
  for (i = n - 1; i >= 0; i--)
  {
    sirina = 0;
    for (j = i; j < n; j++)
    {
      sirina += wi[j]; if (j > i) sirina += s;
      /* Ali gredo lahko besede od i do j še vse v eno vrstico? */
      if (sirina > d) break;
      /* Izračunajmo oceno najboljšega razbitja, ki se začne z vrstico i..j. */
      if (j == n - 1) kand = 0;
      else kand = (d - sirina) * (d - sirina) + f[j + 1];
      /* Če je to najboljše razbitje doslej, si ga zapomnimo. */
      if (j == i || kand < f[i]) f[i] = kand;
    }
  }
  /* Izpišimo rezultat. */
  g = fopen("poravnavanje.out", "wt");
  fprintf(g, "%lld\n", f[0]);
  fclose(g);
  return 0;
}
```

Za izračun ocen smo uporabili 64-bitne spremenljivke, ker je besed veliko in bi se lahko zgodilo, da bi bila skupna ocena celotnega razbitja večja od 2^{31}.

Časovna zahtevnost tega postopka je načeloma $O(n \cdot m)$, če je m največje število besed, ki gredo v eno vrstico. V našem primeru gre lahko n do 10^6, število besed v eni vrstici pa je lahko največ okrog 500 (ker imamo $d \leq 1000$, vsaka beseda je široka najmanj 1 in med besedami je še presledek širine najmanj 1). Za namene naše naloge je ta rešitev čisto dovolj dobra, omenimo pa lahko, da obstajajo za reševanje tega problema tudi učinkovitejši postopki, ki dosežejo časovno zahtevnost $O(n \log n)$ ali celo le $O(n)$.[13]

Razmislimo še o tem, kaj bi morali v doslej opisanem pristopu za delitev besedila na vrstice (s poravnavanjem desnega roba) spremeniti, da bi postal tudi praktično uporaben. Kriterij za delitev besedila in ocenjevanje vrstic, kot smo ga doslej uporabljali v tej nalogi, ni daleč od tega, kar se v resnici uporablja v nekaterih sistemih

[13]Glej npr. `http://xxyxyz.org/line-breaking/` in tam navedeno literaturo, še posebej: D. S. Hirschberg, L. L. Larmore: *The least weight subsequence problem*, SIAM J. on Computing, 16(4):628–38, April 1987; A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, R. Wilber: *Geometric applications of a matrix-searching algorithm*, Algorithmica 2(1–4):195–208, November 1987; R. Wilber: *The concave least-weight subsequence problem revisited*, J. of Algorithms 9(3):418–25, September 1988.

za stavljenje besedila, na primer v TEXu. Glavna stvar, ki bi jo morali v našem postopku spremeniti, da bi postal bolj realističen, je naslednje: recimo, da imamo vrstico, ki obsega besede od w_i do w_j; „naravna“ širina presledkov v njej bi bila torej $(j-i)\cdot s$, pri poravnavi desnega roba pa bo skupna širina presledkov v njej narasla na $d-(w_i+w_{i+1}+\ldots+w_j)$. Pri naši nalogi smo za oceno vrstice vzeli kvadrat razlike med tema širinama, v resnici pa bi morali upoštevati, da se bo dodatni vrinjeni prostor enakomerno razporedil med vseh $j-i$ presledkov med besedami v tej vrstici in za bralca je moteče predvsem, če so posamezni presledki zaradi tega preširoki; več ko je presledkov v tej vrstici, manj se razširi vsak od njih in manj je takšna razširitev moteča za bralca. Zato je za oceno vrstice koristno vzeti *razmerje* med širino presledkov po poravnavanju desnega roba in pred njo:

$$\mathit{ocena}'(i,j) = \left(\frac{d-(w_i+w_{i+1}+\ldots+w_j)}{(j-i)\cdot s}\right)^2.$$

Možne razširitve so še: oceni lahko prištejemo nek člen, ki je odvisen le od j in s katerim lahko določimo, da so nekatera mesta primernejša za prelom vrstice kot druga; oceni lahko prištejemo neko konstanto (večja ko je, bolj bo algoritem motiviran razlomiti odstavek v čim manjše število vrstic); namesto kvadrata lahko vzamemo v formuli kub in s tem še bolj silimo algoritem k izogibanju preširokim presledkom; poleg raztezanja presledkov bi lahko dovolili tudi krčenje, če je na primer d manjši od $w_i+\ldots+w_j+(j-i)\cdot s$, vendar večji od $w_i+\ldots+w_j$ same.

Še ena pomembna razširitev, ki bi jo morali podpreti, da bi dobili praktično uporaben postopek, pa je deljenje besed. Slednje bi v naš algoritem lahko vključili tako, da za $w_1,\ldots,w_n$ namesto dolžin besed vzamemo dolžine posameznih zlogov, pri tem pa si tudi zapomnimo, kateri zlogi so na koncu besede (in zato za njimi pride presledek); pri računanju ocene za vrstico od w_i do w_j to zdaj tudi pomeni, da število presledkov ni nujno $j-i$, ampak je praviloma manjše.[14]

Potrebo po prehudem širjenju presledkov med besedami lahko zmanjšamo tudi tako, da dovolimo širjenje besed samih, delno z dodajanjem drobnih razmikov med črke, deloma pa z uporabo širših različic nekaterih posameznih črk. Pri tem moramo seveda paziti, da te razširitve niso tolikšne, da bi postale za bralca opazne in s tem moteče.[15]

---

[14] Lep in podroben opis tega, kako deluje primer dobrega algoritma za razbijanje besedila na vrstice, najdemo npr. v knjigi D. E. Knutha *The TEXbook*, 1984, 14. poglavje, še posebej str. 97–8.

[15] Primer takšnega algoritma je *hz*, ki ga je razvil Hermann Zapf; gl. npr. v Wikipediji članke Hz-program, Microtypography, pdfTeX in tam navedeno literaturo, npr. H. Zapf, *About micro-typography and the hz-program*, Electronic Publishing, 6(3):283–8, September 1993; Hàn Thế Thành, *Micro-typographic extensions to the TEX typesetting system*, Ph. D. thesis, Masaryk University Brno, 2000.

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Ocenjevanje profesorjev

Pripravimo tabelo skupajTock, ki bo imela za vsakega profesorja po en element, v katerem bomo postopoma računali skupno število točk tega profesorja. Na začetku postavimo vse elemente tabele na 0, nato pa se z dvema gnezdenima zankama sprehodimo po vseh anketah: ena zanka gre po dijakih (z d od 1 do m), druga pa po številu točk, ki jih posamezni dijak lahko nameni posameznemu profesorju (torej gremo s t od -3 do 3, pri čemer moramo paziti, da preskočimo $t = 0$); pri vsakem dijaku in številu točk pokličemo funkcijo Komu(d, t), da vidimo, kateri profesor je od tega dijaka dobil toliko točk, in nato ustreznemu elementu tabele skupajTock prištejemo t.

Nato se moramo še enkrat sprehoditi po tabeli skupajTock in pri vsakem profesorju preveriti, če ima negativno število točk (takšne preštejemo v spremenljivki stNegativnih) in če ima manj točk kot kdorkoli doslej (tistega z najmanj točkami si zapomnimo v spremenljivki kdoNajmanj. Tako dobljena rezultata na koncu izpišemo.

```
#include <stdio.h>

int skupajTock[1000000];

int main()
{
   int n = StProfesorjev(), m = StDijakov(), d, t, p, stNegativnih, kdoNajmanj;
   /* Inicializirajmo skupno število točk vsakega profesorja na 0. */
   for (p = 0; p < n; p++) skupajTock[p] = 0;
   /* Seštejmo ocene iz vseh anket. */
   for (d = 1; d <= m; d++) for (t = -3; t <= 3; t++)
      if (t != 0) skupajTock[Komu(d, t) - 1] += t;
   /* Poiščimo najnižje ocenjenega in preštejmo negativne. */
   stNegativnih = 0; kdoNajmanj = 0;
   for (p = 0; p < n; p++) {
      if (skupajTock[p] < 0) stNegativnih++;
      if (skupajTock[p] < skupajTock[kdoNajmanj]) kdoNajmanj = p; }
   /* Izpišimo rezultate. */
   printf("%d %d\n", stNegativnih, kdoNajmanj + 1); return 0;
}
```

2. Spraševanje

Zelo preprosta rešitev je, da imamo tabelo n elementov in hranimo vrednosti v_i v njej. Na začetku postavimo vse elemente na 0; operacija Vprasan(i) mora le povečati enega od elementov za 1. Pri operaciji Koliko bi lahko šli z eno zanko po i, pri vsakem učencu i pa bi šli še z eno vgnezdeno zanko po vseh ostalih učencih j in preverjali, ali je $v_i \geq v_j + d$.

```
int Koliko()
{
   int i, j, koliko = 0;
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) if (v[i] >= v[j] + d) break;
```

```
      if (j < n) koliko++; }
   return koliko;
}
```

Slabost te rešitve je, da operacija Koliko v najslabšem primeru porabi $O(n^2)$ časa. Opazimo lahko, da je pogoj $v_i \geq v_j + d$ tem lažje izpolnjen, čim manjša je desna stran. Koristno je torej na začetku poiskati najmanjši v_j in ga potem uporabiti pri vsakem i. S tem se izognemo vgnezdeni zanki po j:

```
int Koliko()
{
   int i, najmanjsi, koliko = 0;
   najmanjsi = v[0]; for (i = 1; i < n; i++) if (v[i] < najmanjsi) najmanjsi = v[i];
   for (i = 0; i < n; i++)
      if (v[i] >= najmanjsi + d) koliko++;
   return koliko;
}
```

Tako porabimo za operacijo Koliko le še $O(k)$ časa.

Naloga pravi, da so na začetku vsi v_i enaki 0 in da poteka spraševanje tako, da učenec i gotovo ne bo vprašan, če za nekega j velja, da je $v_i \geq v_j + d$. Iz tega sledi, da se največja in najmanjša vrednost v tabeli v ne moreta razlikovati za več kot za d. (Recimo, da to ni res in da zdajle na primer velja $v_i > v_j + d$. Preden je bil i nazadnje vprašan, je bil v_i za 1 manjši kot zdaj, torej je veljalo $v_i \geq v_j + d$; toda tak i sploh ne bi mogel biti vprašan, tako da smo prišli v protislovje.) Označimo najmanjšo izmed vrednosti $v_1, \ldots, v_n$ z m. Možne vrednosti v tej tabeli so torej le $m, m+1, \ldots, m+d$. Učenci, ki jih trenutno ne smemo vprašati, so natanko tisti, ki imajo $v_i = m + d$. Koristno bi torej bilo, če bi za vsako možno vrednost od m do $m + d$ hranili podatek o tem, pri koliko učencih ima v_i to vrednost.

Označimo s k_x število učencev, ki so bili vprašani že točno x-krat (torej število takih i, za katere je $v_i = x$). Teh števcev ni težko vzdrževati; ko operacija Vprasan(i) poveča v_i z neke vrednosti x na $x + 1$, mora zmanjšati k_x za 1 in povečati k_{x+1} za 1. Pazimo še na to, da ko k_m pade na 0, to pomeni, da ni več nobenega učenca, ki bi imel $v_i = m$, zato najmanjša vrednost v tabeli $v_1, \ldots, v_n$ ni več m, ampak $m+1$.

Vrednosti k_x bi lahko hranili v tabeli $k[0..d]$, pri čemer bi bil k_x shranjen v celici $k[x-m]$ (spomnimo se, da nas zanimajo vrednosti k_x le za x od m do $m+d$). Vprašanje je, kaj narediti, ko se m poveča za 1 (ker smo vprašali še zadnjega učenca, ki je imel doslej $v_i = m$). Načeloma bi lahko vse elemente tabele k zamaknili za en indeks navzdol; neugodno pri tem je, da nam to vzame $O(d)$ časa in če imamo smolo, se nam lahko to zgodi pri vsakem drugem spraševanju. Bolje je, če si tabelo k predstavljamo ciklično: vrednost k_x hranimo v celici $k[x \bmod (d+1)]$. Tako si sicer k_m in k_{m+d+1} delita isto celico; vendar bomo vrednost k_{m+d+1} potrebovali šele, ko se bomo z m premaknili na $m+1$, takrat pa vrednosti k_m ne bomo potrebovali več (ker bo že padla na 0). Tako smo dobili naslednjo elegantno rešitev, v kateri obe operaciji, Vprasan in Koliko, vzameta le $O(1)$ časa:

```
int v[n], k[d + 1], m;

void Inicializacija()
{
   int i, x;
```

```
   /* Postavimo vse v_i na 0. */
   for (i = 0; i < n; i++) v[i] = 0;
   /* Ker imajo vsi učenci v_i = 0, je m = 0 in k_0 = n, za x > 0 pa je k_x = 0. */
   m = 0; k[0] = n; for (x = 1; x <= d; x++) k[x] = 0;
}

void Vprasan(int i)
{
   int x;
   i -= 1;      /* Ker so številke učencev 1..n, indeksi v tabelo pa 0..n − 1. */
   x = v[i];    /* Zapomnimo si staro vrednost v_i. */
   v[i] += 1;   /* Popravimo v_i. */

   /* Zmanjšajmo k_x za 1, ker učenec i nima več v_i = x. */
   k[x % (d + 1)] -= 1;

   /* Če je k_m padel na 0, povečajmo m za 1. Ista celica tabele k, ki je doslej
      hranila k_m (za stari m), bo po novem hranila k_{m+d} (za novo vrednost m). */
   if (x == m && k[x % (d + 1)] == 0) m++;

   /* Povečajmo k_{x+1}, ker ima učenec i po novem v_i = x + 1. */
   k[(x + 1) % (d + 1)] += 1;
}

int Koliko() {
   /* Vrniti moramo k_{m+d}, ki se hrani v k[(m + d) % (d + 1)]. */
   return k[(m + d) % (d + 1)]; }
```

Možne so tudi še druge rešitve; na primer, vrednosti $v_1, \ldots, v_n$ bi lahko hranili v primerno uravnoteženem drevesu (npr. rdeče-črnem ali AVL-drevesu), kar bi nam omogočilo obe operaciji izvajati v času $O(\log n)$.

3. Žica

Postavimo našo žico v koordinatni sistem in to tako, da bo začetek žice v točki $(0, 0)$ in da bo žica tam kazala v desno, torej v smeri pozitivne x-osi. Nato prebirajmo opis žice vrstico za vrstico; pri vsaki vrstici najprej izvedimo premik (v trenutni smeri) za takšno dolžino, kot je navedena v tej vrstici, nato pa popravimo smer za 90 stopinj v levo ali desno, odvisno od tega, ali je v vrstici znak `L` ali `D`.

Smer hranimo v spremenljivki `smer`, ki ima lahko vrednost 0 (desno), 1 (gor), 2 (levo) ali 3 (dol). Vidimo lahko, da če smo imeli pred obratom smer s, imamo po obratu v levo smer $(s + 1) \bmod 4$, po obratu v desno pa smer $(s + 3) \bmod 4$. Hkrati lahko tako definirano smer uporabljamo tudi kot indeks v tabeli `DX` in `DY`, ki nam povesta, kako se spreminjata naši x- in y-koordinati pri premiku dolžine 1 v to smer.

Ker vemo, da smo začeli v točki $(0, 0)$, moramo na koncu le še preveriti, če se zdaj spet nahajamo v točki $(0, 0)$; če se, je žica sklenjena, sicer pa ni.

```
#include <stdio.h>

int main()
{
   const int DX[4] = { 1, 0, −1, 0 }, DY[4] = { 0, 1, 0, −1 };
   int n, premik, x = 0, y = 0, smer = 0; char obrat[2];

   /* Preberimo opis žice. */
   scanf("%d", &n);
   while (n-- > 0)
```

```
  {
    /* Preberimo naslednji segment žice. */
    scanf("%s %d", obrat, &premik);

    /* Premaknimo se v trenutni smeri. */
    x += DX[smer] * premik; y += DY[smer] * premik;

    /* Izračunajmo novo smer. */
    if (obrat[0] == 'L') smer = (smer + 1) % 4;
    else if (obrat[0] == 'D') smer = (smer + 3) % 4;
  }
  /* Izpišimo rezultat. */
  printf("%s\n", (x == 0 && y == 0) ? "Da" : "Ne"); return 0;
}
```

Razmislimo še o različici naloge, pri kateri moramo iz zaporedja koordinat vseh pregibov rekonstruirati zaporedje parov ⟨smer, razdalja⟩. Če imamo recimo $n+1$ točk $T_i(x_i, y_i)$ za $i = 0, \ldots, n$ (pri čemer je T_0 začetek žice, T_n konec žice, vmesne točke pa so prepogibi), bo naša žica sestavljena iz n segmentov (daljic), torej bo naše izhodno zaporedje seznam parov (s_i, r_i) za $i = 1, \ldots, d$. Tak par nam pove, da gre i-ti prepogib v smer s_i in da do njega pride r_i enot za prejšnjim, $(i-1)$-vim prepogibom. Torej je r_i preprosto razdalja med točkama T_i in T_{i-1}. Razdaljo bi lahko računali s Pitagorovim izrekom, $r_i = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$, še lažje pa je, če se spomnimo, da je vsak segment naše žice vodoraven ali navpičen, torej se dve zaporedni točki gotovo ujemata v eni od koordinat (v x-koordinati, če je segment navpičen, oz. v y-koordinati, če je vodoraven). Torej lahko rečemo kar $r_i = |x_i - x_{i-1}| + |y_i - y_{i-1}|$.

Smer prepogiba s_i pa nam mora povedati, ali je se pri prehodu iz segmenta $T_{i-1}T_i$ v naslednji segment T_iT_{i+1} obrnemo v levo ali v desno. Preprosta rešitev je, da to naredimo z nekaj pogojnimi stavki. Najprej za vsak segment določimo smer: naj bo recimo $\triangle x = x_i - x_{i-1}$ in $\triangle y = y_i - y_{i-1}$. Če je $\triangle y = 0$ in $\triangle x > 0$, kaže segment $T_{i-1}T_i$ v levo; če je $\triangle y = 0$ in $\triangle x < 0$, kaže v desno; in tako naprej. Podobno lahko naredimo še za segment T_iT_{i+1}, pri vsakem od njiju predstavimo smer s številom od 0 do 3 (tako kot prej pri rešitvi prvotne različice naloge) in nato preverimo, ali je $smer_{i+1}$ enaka $(smer_i + 1) \bmod 4$ (prepogib v levo) ali $(smer_i + 3) \bmod 4$ (prepogib v desno).

Elegantna možnost pa je, da si pomagamo z vektorskim produktom. Mislimo si, da imajo vse naše točke še z-koordinato, ki je pri vseh enaka 0; naša dva segmenta sta zdaj opisana z vektorjema $\mathbf{u} := (x_i - x_{i-1}, y_i - y_{i-1}, 0)$ in $\mathbf{v} := (x_{i+1} - x_i, y_{i+1} - y_i, 0)$. Njun vektorski produkt je $\mathbf{u} \times \mathbf{v} = (0, 0, p)$ za $p = (x_i - x_{i-1})(y_{i+1} - y_i) - (x_{i+1} - x_i)(y_i - y_{i-1})$. Iz pravila desne roke, s katerim je definiran vektorski produkt, sledi, da mora biti $p > 0$, če je $\mathbf{v}$ usmerjen levo glede na $\mathbf{u}$, sicer pa $p < 0$. Smer prepogiba torej dobimo tako, da izračunamo p in pogledamo njegov predznak. Zapišimo to rešitev še v C-ju:

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

void Zica(int n, const int x[], const int y[])
{
  int i, r, p, dx1, dy1, dx2, dy2;
  for (i = 1; i <= n; i++)
```

```
  {
    dx1 = x[i] − x[i − 1]; dy1 = y[i] − y[i − 1];
    r = abs(dx1) + abs(dy1);
    if (i == n) p = 1; /* Na koncu žice prepogiba ni in je vseeno, kaj izpišemo. */
    else { /* Določimo smer prepogiba z vektorskim produktom. */
      dx2 = x[i + 1] − x[i]; dy2 = y[i + 1] − y[i];
      p = dx1 * dy2 − dx2 * dy1; }
    printf("%c %d\n", (p > 0) ? 'L' : 'D', r);
  }
}
```

4. Račja

Glavni del programa najprej prebere število rač n, nato pa v zanki prebere naslednjih $n+1$ vrstic z opisi oglašanja rac. Ker moramo oglašanje vsake race primerjati le z oglašanjem njene neposredne predhodnice, je dovolj, če hranimo le dve vrstici — trenutno in prejšnjo. Spodnji program za to skrbi tako, da izmenično bere vrstice v tabeli vrstica[0] in vrstica[1]. Nato obe vrstici podamo funkciji Primerjaj, ki nam pove, v koliko zlogih se razlikujeta. Največjo doslej znano razliko med dvema zaporednima vrsticama si zapomnimo v spremenljivki najRazlika, skupaj z njo pa tudi številko račke, pri kateri je do te razlike prišlo (spremenljivka kjeNaj). Slednjo na koncu izpišemo.

```
#include <stdio.h>

int main()
{
  char vrstica[2][300];
  int n, i, razlika, najRazlika = −1, kjeNaj = 0;
  /* Preberimo število rač. */
  scanf("%d\n", &n);
  for (i = 0; i <= n; i++)
  {
    /* Preberimo oglašanje naslednje race. */
    gets(vrstica[i % 2]);
    /* Če je to mama raca, si jo le zapomnimo in pojdimo na naslednjo. */
    if (i == 0) continue;
    /* V koliko zlogih se ta raca razlikuje od prejšnje? */
    razlika = Primerjaj(vrstica[0], vrstica[1]);
    /* Če je to največja razlika doslej, si jo zapomnimo. */
    if (razlika > najRazlika) najRazlika = razlika, kjeNaj = i;
  }
  /* Izpišimo rezultat. */
  printf("%d\n", kjeNaj); return 0;
}
```

Napišimo zdaj še podprogram Primerjaj, ki mora ugotoviti, v koliko zlogih se razlikujeta obe vrstici. To je precej preprosto, saj naloga pravi, da vsebujejo vse vrstice enako število zlogov, vsi zlogi so dolgi po dva znaka, ločeni so s po enim presledkom, pred in za njimi pa ni v vrstici nobenih drugih znakov. Iz tega sledi, da na primer prvi znak vsakega niza vsebuje prvo črko prvega zloga; drugi znak vsebuje

drugo črko prvega zloga; tretji znak vsebuje presledek; četrti znak vsebuje prvo črko drugega zloga in tako naprej. Naš podprogram lahko preprosto primerja istoležne znake v obeh vrsticah in šteje neujemanja; vsak neujemajoč se zlog (torej tak, ki je pri eni raci „ga“, pri eni pa „GA“) nam prispeva dva neujemajoča se znaka, zato moramo na koncu število neujemajočih se znakov še deliti z 2, pa dobimo število neujemajočih se zlogov.

```
int Primerjaj(const char *s, const char *t)
{
   int razlika = 0;
   while (*s) if (*s++ != *t++) razlika++;
   return razlika / 2;
}
```

5. Kraljice

Nalogo lahko rešimo na veliko načinov, ki se razlikujejo po tem, koliko časa in pomnilnika porabijo.

(1) Preprosta, a neučinkovita rešitev je, da si pripravimo tabelo $n \times n$ logičnih vrednosti, ki za vsako polje šahovnice povedo, ali na njem stoji kraljica ali ne. Nato se za vsako kraljico in za vsako od osmih možnih smeri napada premikamo od te kraljice v tisto smer, dokler ne naletimo na rob šahovnice ali pa na kakšno drugo kraljico. Če smo naleteli na kraljico prej kot na rob, potem vemo, da se tidve kraljici napadata in lahko povečamo nek števec napadajočih se parov za 1. Na koncu ta števec delimo z 2 (ker smo vsak par kraljic šteli dvakrat) in ga izpišemo. Slabost te rešitve je, da porabi $O(n^2)$ pomnilnika (in časa), kar je za našo nalogo pravzaprav že preveč.

(2) Lahko se za vsako vrstico šahovnice sprehodimo po vseh kraljicah in štejemo, koliko kraljic leži v tej vrstici. Če ugotovimo, da je v tej vrstici na primer a kraljic in je $a > 1$, potem vemo, da prispevajo $a - 1$ napadajočih se parov (prva in druga se napadata; druga in tretja se napadata; tretja in četrta se napadata; in tako naprej). Enako naredimo še s stolpci in diagonalami. Ta rešitev porabi $O(1)$ pomnilnika in $O(n \cdot k)$ časa (imamo $O(n)$ vrstic, stolpcev in diagonal, pri vsaki od njih pa moramo iti po vseh k kraljicah), kar je še vedno veliko, vendar že precej bolje od prejšnje rešitve.

(3) Imejmo tabelo a z n elementi in na začetku postavimo vse na 0; nato pojdimo v zanki po vseh kraljicah in pri i-ti kraljici povečajmo $a[y_i]$ za 1. Na koncu te zanke vemo za vsako vrstico, koliko kraljic je v njej: v vrstici y je $a[y]$ kraljic. Število napadajočih se parov lahko zdaj računamo enako kot pri rešitvi (2). Enako naredimo še s stolpci in diagonalami. Ta rešitev porabi zdaj sicer $O(n)$ pomnilnika, vendar le $O(n + k)$ časa.

(4) Namesto običajne tabele lahko v rešitvi (3) uporabimo razpršeno tabelo (*hash table*), v kateri so prisotni elementi le za tiste vrstice (ali stolpce ali diagonale), v katerih leži vsaj ena kraljica. Poraba pomnilnika se zato zmanjša na $O(k)$, poraba časa pa tudi na $O(k)$.

(5) Lahko gremo v dveh gnezdenih zankah po vseh parih kraljic in pri vsakem paru preverimo, ali se napadata ali ne. V ta namen moramo najprej preveriti, ali kraljici ležita v isti vrstici, stolpcu ali diagonali, nato pa še, ali med njima leži kakšna

tretja kraljica. Zato uporabimo še tretjo gnezdeno zanko po vseh kraljicah. Tako dobimo rešitev s časovno zahtevnostjo $O(k^3)$ in prostorsko zahtevnostjo $O(1)$.

(6) Izberimo si neko konkretno kraljico i; za vsako od osmih možnih smeri napada se zapeljimo v zanki po vseh ostalih kraljicah in preverimo, ali leži kakšna kraljica v tisti smeri glede na kraljico i. Če najdemo kakšno tako kraljico, potem vemo, da naša kraljica i napada natanko eno kraljico v tisti smeri — namreč tisto, ki ji je najbližja, ampak za naše potrebe tako ali tako ni pomembno, da vemo, katera točno je ta kraljica. Ta postopek zdaj ponovimo za vse možne i in tako dobimo skupno število napadajočih se parov. Ta rešitev porabi $O(k^2)$ časa in le $O(1)$ pomnilnika.

Naloge so sestavili: pacifistični generali, tiskana vezja, spraševanje — Nino Bašić; luči — Tomaž Hočevar; žica — Nace Hudobivnik; poravnavanje desnega roba — Matjaž Leonardis; potenciranje — Matjaž Leonardis in Janez Brank; dnevnik — Mark Martinec; davek na ograjo, skrivno sporočilo, ocenjevanje profesorjev — Jure Slak; proizvodnja čopičev, uniforme, prenova ceste, račja, kraljice, ljudožerci na premici, po Indiji z avtobusom — Mitja Trampuš; vnos šifre, bloki — Janez Brank.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2012

1. Prestopna leta

Dan v tednu lahko opišemo s celim številom od 0 do 6; recimo, da 0 pomeni ponedeljek, 1 torek in tako naprej. Dnevu s številko d načeloma sledi dan s številko $d + 1$, razen pri $d = 6$, ko naslednji dan ni $d = 7$, pač pa $d = 0$. To lahko elegantno opišemo s pomočjo matematične operacije mod (ostanek po deljenju): dnevu d sledi dan $(d + 1)$ mod 7. Enak razmislek velja tudi za daljša obdobja; k dni po dnevu d pride dan $(d+k)$ mod 7. Če torej za neko leto vemo, da se je začelo na dan d, lahko sklepamo, da se naslednje leto začne na dan $(d+365)$ mod 7 ali pa $(d+366)$ mod 7, odvisno od tega, ali je bilo naše leto prestopno ali ne.

Naš program gre lahko v zanki po letih od 1900 do 2012 in sproti popravlja spremenljivko d, ki pove, na kateri dan v tednu se začne trenutno leto. Pri vsakem letu tudi preverimo, če je prestopno, in če je, povečajmo števec prestopnih let, ki so se začela na dan d. Te števce hranimo v tabeli koliko; na koncu se program sprehodi po tej tabeli in pogleda, pri katerem indeksu (spremenljivka naj) nastopi v tej tabeli največji element. To je rezultat, ki ga iščemo; nato ga moramo le še izpisati.

```
#include <stdio.h>
int main()
{
  const char *imena[ ] = { "ponedeljek", "torek", "sreda", "četrtek", "petek",
                          "sobota", "nedelja" };
  int koliko[7], d, naj, leto;
  /* Inicializirajmo tabelo števcev. */
  for (d = 0; d < 7; d++) koliko[d] = 0;
  /* Preglejmo vsa leta v opazovanem obdobju. */
  for (d = 0, leto = 1900; leto <= 2012; leto++)
  {
    /* Trenutno leto se začne na dan d. Ali je prestopno? */
    if (leto % 4 == 0 && (leto % 100 != 0 || leto % 400 == 0))
    {
      koliko[d]++; /* Povečajmo števec let, ki se začnejo na dan d. */
      d++; /* Upoštevajmo, da je to leto en dan daljše od običajnih let. */
    }
    d = (d + 365) % 7; /* Izračunajmo, na kateri dan se začne naslednje leto. */
  }
  /* Poglejmo, na kateri dan se začne največ let. */
  for (naj = 0, d = 1; d < 7; d++)
    if (koliko[d] > koliko[naj]) naj = d;
  /* Izpišimo rezultat. */
  printf("Največ prestopnih let se začne na %s.\n", imena[naj]);
  return 0;
}
```

Opazovano obdobje sicer pri tej nalogi ni najbolj posrečeno izbrano: leto 1900 ni prestopno, nato pa sledi ravno 28 prestopnih let (od 1904 do 2012), ki si sledijo v enakomernih presledkih po 4 leta. Takšno 4-letno obdobje obsega $3 \cdot 365 + 366 = 1461$ dni, kar je 208 tednov in 5 dni. Tu torej velja, da če se eno prestopno leto

začne na dan d, se naslednje začne na dan $(d + 5) \bmod 7$. Ker sta si 5 in 7 tuji števili, se ta vzorec začne ponavljati na vsakih 7 prestopnih let, tako da v našem obdobju 28 zaporednih prestopnih let nastopi vsak možen začetni dan v tednu ravno štirikrat. Tudi če si začetno in končno leto izberemo drugače, se pogosto zgodi, da najpogostejši začetni dan ni enoličen.

Rešitev bi lahko še izboljšali, da bi delovala hitro tudi za zelo dolga časovna obdobja. Pri tem bi si pomagali z opažanjem, da se na vsakih 400 let ciklično ponavlja ne le vzorec tega, katera leta so prestopna, ampak tudi to, na kateri dan v tednu se začnejo (kajti 400 let, od tega 97 prestopnih, obsega celo število tednov — namreč 20 871 tednov). Če torej obdobje, ki nas zanima, obsega recimo n zaporednih let, ga lahko razdelimo na $\lfloor n/400 \rfloor$ ciklov po 400 let in še na zadnjih $n \bmod 400$ let. Potem je dovolj, če obdelamo prvi 400-letni cikel in rezultate zanj pomnožimo z $\lfloor n/400 \rfloor$, nato pa prištejemo še rezultate za zadnjih $n \bmod 400$ let.[16]

Pri tem se izkaže, da prevladujejo prestopna leta, ki se začnejo na petek ali nedeljo: če pogledamo katerih koli 400 zaporednih let, je med 97 prestopnimi leti v tem obdobju po 15 takih, ki se začnejo na petek ali nedeljo; po 14 takih, ki se začnejo na torek ali sredo; in po 13 takih, ki se začnejo na ponedeljek, četrtek ali soboto. Tudi v splošnem velja, da če je opazovano obdobje dolgo vsaj 384 let, je najpogostejši začetni dan prestopnega leta gotovo petek ali nedelja; to, kateri od teh dveh je pogostejši od drugega (če nista oba enako pogosta), pa je odvisno od tega, kdaj (s katerim letom) se naše opazovano obdobje začne in kdaj konča.

2. Kazalca

(a) Recimo, da k_1 kaže ure in k_2 minute. Torej naredi k_1 obrat za 360 stopinj v 12 urah, tako da je trenutni čas v urah enak $t = 12 \cdot (k_1/360)$. Razdelimo t na celi del $u = \lfloor t \rfloor$ in neceli del $m = t - u$ (to je tisto, kar je za decimalno vejico). Število m nam pove, koliko časa (v urah) je preteklo od začetka trenutne ure. Drugi kazalec bi v eni uri naredil 360 stopinj; v m urah torej $m \cdot 360$ stopinj. Torej mora biti $k_2 = m \cdot 360$; če ni, je to znak, da je bila naša predpostavka (da k_1 kaže ure, k_2 pa minute) napačna.

Podobno preizkusimo še drugo možnost, torej da k_2 kaže ure, k_1 pa minute. Lahko se zgodi, da pri nobeni od obeh možnosti ne dobimo rešitve; tedaj vemo, da so vhodni podatki neveljavni. Lahko pa dobimo rešitev pri obeh; če sta tako dobljena časa različna, potem vemo, da so vhodni podatki dvoumni.

Zapišimo to rešitev še v C-ju. Pri preverjanju, ali je $k_2 = m \cdot 360$, moramo biti previdni: ker imamo v računalniku realna števila predstavljena le z omejeno natančnostjo, se lahko zaradi zaokrožitvenih napak zgodi, da sta števili različni, čeprav ne bi smeli biti. Zato namesto enakosti raje preverimo, če se razlikujeta za dovolj malo, recimo za manj kot ε, pri čemer je ε neko majhno pozitivno število.

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

void KazalcaA(double k1, double k2)
{
  double t1 = 12 * (k1 / 360), t2 = 12 * (k2 / 360);
```

[16] Na ta način smo reševali že nalogo 2010.X.3; gl. str. 78–9 v biltenu 2012.

```
  int u1 = (int) floor(t1), u2 = (int) floor(t2);
  double m1 = t1 − u1, m2 = t2 − u2;
  const double eps = 1e-6;
  bool ok1 = fabs(m1 * 360 − k2) < eps, ok2 = fabs(m2 * 360 − k1) < eps;
  if (ok1 && ok2)
    if (u1 == u2 && m1 == m2) printf("Ura je %d:%g, vendar ne vemo, "
                                    "kateri kazalec je kateri.\n", u1, m1 * 60);
    else printf("Podatki so dvoumni, ura je lahko %d:%g ali %d:%g.\n",
              u1, m1 * 60, u2, m2 * 60);
  else if (ok1) printf("Ura je %d:%g.\n", u1, m1 * 60);
  else if (ok2) printf("Ura je %d:%g.\n", u2, m2 * 60);
  else printf("Podatki so neveljavni.\n");
}
```

Razmislimo o tem, kdaj so lahko vhodni podatki dvoumni. Recimo, da je trenutni čas $u : m$, pri čemer je u celo število ur (od 0 do 11), minute m pa tudi izrazimo v urah, tako da je m neko realno število z intervala $[0, 1)$. Položaj urnega kazalca je torej $(u + m) \cdot 30$ stopinj (spomnimo se, da urni kazalec naredi 360 stopinj v 12 urah, torej 30 stopinj na uro), minutnega pa $m \cdot 360$ (ker naredi cel krog v eni uri). Do dvoumnosti pride, če bi se dalo ob nekem drugem času $u' : m'$ dobiti isti položaj kazalcev, le da v zamenjanih vlogah. Tedaj bi torej imeli $(u + m) \cdot 30 = m' \cdot 360$ in $m \cdot 360 = (u' + m') \cdot 30$. Iz tega dobimo $m' = (u + m)/12$ in $m = (u' + m')/12$. Če prvo enačbo vstavimo v drugo, dobimo $m = u'/12 + (u+m)/144$, iz česar lahko izrazimo $m = (u + 12u')/143$. Podobno bi dobili še $m' = (u' + 12u)/143$.

Načeloma si lahko torej izberemo poljubna u in u' (samo da sta celi števili od 0 do 11), pa bomo po teh formulah lahko našli primerna m in m'; vendar pa niso vse tako dobljene kombinacije dvoumne — če vzamemo $u = u'$, bomo dobili tudi $m = m'$ in takšen vhodni primer v resnici ne bo dvoumen (program bi lahko ugotovil, koliko je ura, le tega ne bi vedel, kateri kazalec je urni in kateri minutni). Pri $u \neq u'$ pa res nastopijo dvoumnosti. Oglejmo si konkreten primer: če vzamemo $u = 0$, $u' = 1$, nam gornji formuli povesta, da moramo vzeti še $m = 12/143$ in $m' = 1/143$. Pri $u : m$ (kar je približno 5 minut in 2 sekundi po polnoči ali poldnevu) bi bil urni kazalec pod kotom približno 2,5°, minutni pa približno 30,2°; pri $u' : m'$ (kar je približno 1 uro in 25 sekund po polnoči ali poldnevu) pa bi bila kota enaka, le vlogi kazalcev bi bili zamenjani.

(b) Razmislek je podoben kot pri (a), ker pa se kazalca zdaj premikata diskretno, bomo tako ure kot minute šteli s celimi števili. Recimo, da je trenutni čas $u : m$, pri čemer je u celo število od 0 do 11, m pa celo število od 0 do 59. Spomnimo se, da se urni kazalec vsako minuto premakne za pol stopinje, minutni pa za šest stopinj; ob času $u : m$ bo torej urni kazalec pod kotom $30u + m/2$ stopinj, minutni pa $6m$ stopinj. Če je torej na primer k_2 minutni kazalec, mora biti k_2 večkratnik števila 6, minute m lahko potem dobimo preprosto kot $m = k_2/6$, urni kazalec pa nam da enačbo $k_1 = 30u + m/2$, tako da lahko izračunamo u kot $u = (k_1 - m/2)/30$. Če tako dobljen u ni celo število, lahko zaključimo, da k_2 v resnici ni bil minutni kazalec.

Podobno kot pri (a) moramo nato preizkusiti še drugo možnost, torej da je k_2 urni kazalec, k_1 pa minutni. Ali lahko tudi zdaj pride do dvoumnosti? Videli smo, da pri času $u : m$ dobimo urni kazalec pod kotom $30u+m/2$ in minutnega pod kotom

$6m$. Če lahko do enakih kotov, le da v zamenjanih vlogah, pride tudi pri nekem drugem času $u' : m'$, to pomeni, da je $30u + m/2 = 6m'$ in $6m = 30u' + m'/2$. Iz prve enačbe dobimo $m = 12m' - 60u$, iz druge $m' = 12m - 60u'$; če nesemo zdaj drugo v prvo, dobimo $m = 144m - 720u' - 60u$, torej $143m = 720u' + 60u$. Spomnimo se, da nas zanimajo le celoštevilske rešitve; pri celih u, u' je desna stran te enačbe vedno večkratnik 60; leva stran pa za noben m od 1 do 59 ni večkratnik 60 (saj m sam po sebi ni večkratnik 60, faktor 143 pa nam pri tem nič ne pomaga, saj je $143 = 11 \cdot 13$). Ostane torej le primer z $m = 0$ in $u = u' = 0$, vendar je takrat tudi $m' = 0$ in dvoumnosti glede časa takrat sploh ni (je le dvoumnost glede tega, kateri kazalec je urni in kateri minutni).

Zapišimo našo rešitev še v C-ju. Če vhodna kota k_1 in k_2 pomnožimo z 2, bi morali nastati celi števili, saj smo v besedilu naloge videli, da se premika urni kazalec v korakih po pol stopinje (minutni pa celo po 6 stopinj). Naš podprogram torej najprej naredi to (dobimo spremenljivki K1 in K2) in v nadaljevanju računa vse s celoštevilsko aritmetiko namesto s plavajočo vejico.

```
void KazalcaB(double k1, double k2)
{
   int K1 = (int) (2 * k1), K2 = (int) (2 * k2);
   if (K1 != 2 * k1 || K2 != 2 * k2) { printf("Podatki so neveljavni.\n"); return; }
   int m1 = K2 / 12, m2 = K1 / 12;
   int u1 = (K1 − m1) / 60, u2 = (K2 − m2) / 60;
   bool ok1 = (K1 == 60 * u1 + m1 && K2 == 12 * m1);
   bool ok2 = (K2 == 60 * u2 + m2 && K1 == 12 * m2);

   if (ok1) printf("Ura je %d:%02d.\n", u1, m1);
   else if (ok2) printf("Ura je %d:%02d.\n", u2, m2);
   else printf("Podatki so neveljavni.\n");
}
```

Pri preverjanju, ali je na primer K1 (zaokrožen v celo število) res enak 2 * k1 (pri čemer je k1 še v plavajoči vejici), smo uporabili kar primerjanje po enakosti, kajti pri predstavitvi realnih števil s plavajočo vejico je mogoče števila oblike $a/2$ za celoštevilske a predstaviti brez zaokrožitvenih napak.

Nalogo bi lahko rešili tudi preprosteje, le malo manj učinkovito: z dvema gnezdenima zankama bi šli po vseh možnih u in m in pri vsakem preverili, ali bi ob času $u : m$ bila kazalca v takem položaju, kot ga navajajo naši vhodni podatki. Ker ne vemo, kateri kazalec je urni in kateri minutni, moramo preizkusiti obe možnosti:

```
void KazalcaB2(double k1, double k2)
{
   int u, m; double urni, minutni;
   for (u = 0; u < 12; u++) for (m = 0; m < 60; m++) {
      urni = u * 30 + m / 2.0; minutni = m * 6;
      if ((k1 == urni && k2 == minutni) || (k2 == urni && k1 == minutni)) {
         printf("Ura je %d:%02d.\n", u, m); return; }}
   printf("Podatki so neveljavni.\n");
}
```

(c) Nalogo rešujemo tako kot (b), le formula za položaj urnega kazalca je malo drugačna. Ob času $u : m$ (pri čemer je u spet celo število od 0 do 11, m pa celo število od 0 do 59) bo urni kazalec pod kotom $30u + 6\lfloor m/12 \rfloor$ stopinj, minutni pa pod kotom $6m$ stopinj. Če je recimo k_1 urni kazalec, k_2 pa minutni, dobimo $m = k_2/6$ in

potem $u = (k_1 - 6\lfloor m/12 \rfloor)/30$, seveda le, če sta tako dobljena u in m res celi števili. Podobno kot pri prejšnjih različicah naloge moramo tudi tu preveriti še možnost, da je k_2 minutni kazalec, k_1 pa urni.

Zapišimo rešitev še v C-ju. Položaj kazalcev lahko zdaj predstavimo kar s celimi števili (tip **int**), saj naloga pravi, da se kazalca vedno premikata v korakih po 6 stopinj.

```
void KazalcaC(int k1, int k2)
{
  int m1 = k2 / 6, m2 = k1 / 6;
  int u1 = (k1 − 6 * (m1 / 12)) / 30, u2 = (k2 − 6 * (m2 / 12)) / 30;
  bool ok1 = (k1 == 30 * u1 + 6 * (m1 / 12) && k2 == 6 * m1);
  bool ok2 = (k2 == 30 * u2 + 6 * (m2 / 12) && k1 == 6 * m2);
  if (ok1 && ok2)
    if (u1 == u2 && m1 == m2) printf("Ura je %d:%02d, vendar ne vemo, "
                                     "kateri kazalec je kateri.\n", u1, m1);
    else printf("Podatki so dvoumni, ura je lahko %d:%02d ali %d:%02d.\n",
             u1, m1, u2, m2);
  else if (ok1) printf("Ura je %d:%02d.\n", u1, m1);
  else if (ok2) printf("Ura je %d:%02d.\n", u2, m2);
  else printf("Podatki so neveljavni.\n");
}
```

Razmislimo še o tem, kdaj pride pri tej različici naloge do dvoumnosti. Namesto s parom $u : m$ lahko čas predstavimo tudi s številom minut od polnoči, $t = 60u + m$; to je celo število od 0 do 719. Pri takem času imamo urni kazalec $k_1 = 6\lfloor t/12 \rfloor$ in minutnega $k_2 = 6(t \bmod 60)$. Recimo, da lahko do istega položaja kazalcev, le da v obrnjenih vlogah, pride še ob nekem drugem trenutku $u' : m'$. Takrat imamo torej $k_1 = 6m'$ in $k_2 = 30u' + 6\lfloor m'/12 \rfloor$. Iz prve enačbe dobimo $m' = k_1/6 = \lfloor t/12 \rfloor$, nato pa iz druge $u' = (k_2 - 6\lfloor m'/12 \rfloor)/30$, kar nas sčasoma pripelje do $u' = ((t \bmod 60) - \lfloor t/144 \rfloor)/5$. Z dvoumnostjo imamo opravka le, če je tako dobljeni u' celo število, saj bi drugače vedeli, da položaj z obrnjenima vlogama kazalcev (k_1 kot minutni in k_2 kot urni kazalec) ni mogoč. Z drugimi besedami, trenutek t je dvoumen, če je razlika $(t \bmod 60) - \lfloor t/144 \rfloor$ večkratnik števila 5. Ta pogoj lahko še poenostavimo: ker je tudi razlika med t in $t \bmod 60$ večkratnik števila 5, lahko rečemo, da so dvoumni natanko tisti t, pri katerih je razlika $t - \lfloor t/144 \rfloor$ večkratnik števila 5; oz. še drugače, pri katerih je $t \equiv \lfloor t/144 \rfloor \pmod 5$.

To si lahko predstavljamo takole: $\lfloor t/144 \rfloor$ nam pove, v kateri petini dneva je naš trenutek t; v vsaki petini se potem dvoumni trenutki pojavljajo na vsakih 5 minut: v prvi petini takrat, ko se minute končajo na števko 0 ali 5, v drugi petini takrat, ko se minute končajo na števko 1 ali 6 in tako naprej. Vsega skupaj je tako dvoumnih t-jev 144, je pa med njimi 12 takih, pri katerih kazalca kažeta v isto smer in lahko zanesljivo določimo čas, le tega ne vemo, kateri kazalec je urni in kateri je minutni; pri ostalih 132 t-jih pa res ne moremo določiti, koliko je ura.

3. Motocikel

Naš program bo v neskončni zanki preverjal stanje tipal; poleg trenutnega stanja si zapomnimo še prejšnje stanje (spremenljivka prejStanje), tako da lahko opazimo, kdaj pride do spremembe v stanju, na primer iz **false** iz **true**, torej takrat, ko tipalo

zazna začetek luknje na plošči. Zapomnimo si tudi čas prejšnjega takega prehoda (spremenljivka prehod); iz razlike v času vemo, koliko časa je kolo potrebovalo za zasuk od ene luknje do naslednje. Iz tega lahko ocenimo hitrost vrtenja obeh koles (v luknjah na sekundo — koliko je to v stopinjah ali metrih, sicer ne vemo, ampak za primerjavo hitrosti koles je to že dovolj).[17] Zdaj vemo dovolj, da lahko naredimo, kar zahteva besedilo naloge: voznikov ukaz (iz funkcije PlinVoznika) posredujemo motorju (s funkcijo PlinMotorja), pri tem pa ga še ustrezno popravimo, če smo opazili, da je sprednje kolo hitrejše od zadnjega.

```
int main()
{
   bool stanje[2] = { TipaloSpredaj(), TipaloZadaj() }, prejStanje[2];
   double prehod[2] = { −1, −1 }, hitrost[2] = { −1, −1 }, cas, ukaz; int i;
   while (true)
   {
      cas = Cas();
      for (i = 0; i < 2; i++) {
         prejStanje[i] = stanje[i];
         stanje[i] = (i == 0) ? TipaloSpredaj() : TipaloZadaj();
         /* Ob spremembi stanja na tipalu odčitamo čas in izračunamo hitrost vrtenja. */
         if (stanje[i] && ! prejStanje[i]) {
            if (prehod[i] >= 0) hitrost[i] = 1.0 / (cas − prehod[i]);
            prehod[i] = cas; }}
      ukaz = PlinVoznika();
      /* Če poznamo hitrost koles in je sprednje kolo hitrejše od zadnjega,
          ustrezno prilagodimo ukaz. */
      if (hitrost[0] > hitrost[1] && hitrost[1] > 0)
         ukaz *= hitrost[1] / hitrost[0];
      PlinMotorja(ukaz);
   }
   return 0;
}
```

4. Tročrkovne kode

Imena jezikov bomo obdelovali v zanki po vrsti, kot se poajvljajo v vhodnem seznamu. Pri vsakem imenu pojdimo s tremi gnezdenimi zankami po njegovih črkah in pregledujmo možne tročrkovne kode tega imena. Za vsako kodo moramo preveriti, če smo jo že uporabili kot kodo kakšnega od prej obdelanih imen; čim najdemo tako, ki je še nismo uporabili, jo dodelimo trenutnemu imenu in se premaknemo na naslednje ime v seznamu. Podatke o že uporabljenih kodah lahko hranimo na primer v razpršeni tabeli (*hash table*), v drevesu (*trie*) ali pa celo v navadni tabeli $26 \times 26 \times 26$ logičnih vrednosti, saj besedilo naloge pravi, da se v imenih pojavljajo le male črke angleške abecede, torej je za vsako črko le 26 možnosti.

Zapišimo našo rešitev še v pythonu. Že uporabljene kode bomo hranili v množici (pythonov razred set, za katero se v praksi skriva nekakšna razpršena tabela). Ko najdemo primerno kodo, je smiselno takoj prekiniti gnezdene zanke po črkah; v ta namen si pomagamo z logično spremenljivko ok. Če pregledamo vse možne trojice

[17] Mimogrede, s podobnim problemom ocenjevanja hitrosti s pomočjo tipal smo se pred leti že srečali: glej nalogo 1988.2.3 na str. 24 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004*.

črk trenutnega imena, ne da bi našli primerno kodo, bo ok na koncu še vedno False, kar je koristen podatek, saj bomo tako vedeli, da moramo izpisati, da primerne kode ni bilo mogoče izbrati.

```
def PoisciKode(imena):
   uporabljene = set()
   for ime in imena:
      n = len(ime); ok = False
      for i in range(n − 2):
         for j in range(i + 1, n − 1):
            for k in range(j + 1, n):
               koda = ime[i] + ime[j] + ime[k]
               if koda not in uporabljene: ok = True; break
            if ok: break
         if ok: break
      if not ok: print("Za %s ni mogoce izbrati kode." % ime)
      else: uporabljene.add(koda); print("%s -> %s" % (ime, koda))
```

5. Google Mobil

Naj bo t čas, v katerem bomo (pri izbrani hitrosti) prišli do semaforja. V poštev pridejo le časi, ki so $\geq$ Razdalja/Omejitev, saj bi drugače morali prekoračiti omejitev hitrosti. Tej spodnji meji recimo t_0. V okviru omejitve $t \geq t_0$ nas zdaj zanima najkrajši čas, pri katerem je semafor zelen.

Naj bo z (= zTrajanje) trajanje zelene luči, r (= rTrajanje) pa trajanje rdeče; stanje semaforja se torej ponavlja na vsakih $z + r$ sekund. Stanje semaforja znotraj tega cikla lahko opišemo s številom s_0 z intervala $[0, z + r)$; dogovoriti se moramo, kje si mislimo začetek cikla — recimo, da takrat, ko se prižge rdeča luč. To, kje znotraj tega cikla je semafor na začetku naše vožnje, lahko ugotovimo iz spremenljivk Stanje in Trajanje. Če je začetno stanje semaforja rdeče, je $s_0 = r -$ Trajanje, če pa je začetno stanje semaforja zeleno, imamo $s_0 = r + z -$ Trajanje.

Po t_0 sekundah je semafor v stanju $s := (s_0 + t_0) \bmod (z + r)$. Če je to stanje na območju $[0, r)$, je semafor takrat rdeč, če pa je s na območju $[r, z + r)$, je semafor takrat zelen. Če je zelen, je to rešitev, ki smo jo iskali; če pa je rdeč, moramo voziti malo počasneje, da bomo prišli do semaforja ravno v trenutku, ko se naslednjič prižge zelena luč. Čas vožnje moramo torej podaljšati za $r - s$ sekund.

Po tem razmisleku poznamo najmanjši primerni čas vožnje t, iz njega pa lahko izračunamo tudi hitrost: to je Razdalja$/t$.

Zapišimo to rešitev še v C-ju. Pri izračunu $(s_0 + t_0) \bmod (z + r)$ se stvari rahlo zapletejo; za operacijo mod, torej ostanek po deljenju, imamo v C-ju načeloma operator %, ki pa deluje le za cela števila. Pri naši nalogi imamo opravka z realnimi števili, zato si pomagajmo z dejstvom, da je $x \bmod y = x - \lfloor x/y \rfloor \cdot y$. Pri tem $\lfloor \cdot \rfloor$ pomeni zaokrožanje navzdol (do najbližjega celega števila), za kar lahko uporabimo funkcijo floor iz standardne knjižnice.

```
#include <math.h>

double NajvecjaHitrost(double Razdalja, char Stanje, double Trajanje,
                       double zTrajanje, double rTrajanje, double Omejitev)
{
   /* Določimo začetno stanje semaforja. */
   double s0 = rTrajanje + (Stanje == 'r' ? 0 : zTrajanje) − Trajanje;
```

```
  /* Izračunajmo najmanjši čas, v katerem lahko dosežemo semafor. */
  double t = Razdalja / Omejitev;

  /* V kakšnem stanju bo semafor ob tem času? */
  double cikel = zTrajanje + rTrajanje;
  double s = (s0 + t) − floor((s0 + t) / cikel) * cikel;

  /* Če to stanje ne bo zeleno, počakajmo na zeleno luč. */
  if (s < rTrajanje) t += rTrajanje − s;

  /* Vrnimo hitrost, s katero dosežemo semafor ob tem času. */
  return Razdalja / t;
}
```

6. Celične himere

Sprehodimo se v zanki po vseh zapisih n-te generacije. Vsak zapis razdelimo na dva enako dolga dela; to sta tipa njegovih staršev. Ker nas zanima, koliko različnih tipov se pojavlja v starševski generaciji, je koristno tipe staršev zlagati v razpršeno tabelo (*hash table*). Ob vsakem tipu bomo v tej tabeli hranili še dva števca: koliko celic tega tipa je v $(n-1)$-vi generaciji in pri koliko celicah n-te generacije so se celice tega tipa pojavile kot eden od staršev. Tako na primer, če se v n-ti generaciji pojavi x celic tipa `AADB`, to pomeni, da je moralo biti v $(n-1)$-vi generaciji tudi x celic tipa `AA`, ki so prispevale k tem x celicam tipa `AADB` v n-ti generaciji; zato moramo v tabeli staršev pri tipu `AA` povečati oba števca za x. Enako naredimo tudi za `DB`. Paziti moramo še na primere, ko ima tip dva enaka starša. Na primer, če najdemo v n-ti generaciji x celic tipa `DBDB`, to pomeni, da je moralo biti v $(n-1)$-vi generaciji $2x$ celic tipa `DB`, ki so prispevale k x celicam tipa `DBDB` v n-ti generaciji; zato moramo v tabeli staršev pri tipu `DB` povečati prvi števec za $2x$, drugega pa le za x.

Ko pregledamo vse zapise n-te generacije, se sprehodimo po vseh tipih staršev in poglejmo, kateri je prispeval k največ celicam n-te generacije (torej kateri ima največjo vrednost drugega števca); tega moramo izpisati, skupaj s številom različnih tipov.

Nato se lahko premaknemo za eno generacijo nazaj in postopek ponovimo; to, kar je bila doslej tabela s tipi staršev, postane zdaj tabela s tipi otrok. Postopek se konča, ko pridemo do generacije, pri kateri so opisi staršev dolgi le eno črko.

Oglejmo si še implementacijo te rešitve v pythonu. Predpostavili smo, da naš podprogram dobi opise zadnje generacije kot seznam (list v pythonu) nizov, zato ga najprej predelamo v razpršeno tabelo (dict v pythonu), s kakršnimi bomo delali v nadaljevanju postopka.

```
def Himere(opisi):
  # Seznam opisov predelajmo v razpršeno tabelo, s kakršnimi bomo delali
  # v preostanku postopka. V njej se bo vsak tip pojavil enkrat, ob njem
  # pa bo število celic tega tipa v seznamu „opisi“.
  otroci = {}
  for otrok in opisi:
    if otrok in otroci: otroci[otrok][0] += 1
    else: otroci[otrok] = [1, 0]

  # Obdelajmo generacije od kasnejših proti starejšim.
  while True:
    starsi = {}
```

```
  for (otrok, [stOtrok, _]) in otroci.items():
    stars1 = otrok[:len(otrok) // 2]; stars2 = otrok[len(otrok) // 2:]
    # Popravimo števce pri obeh starševskih tipih.
    if stars1 not in starsi: starsi[stars1] = [stOtrok, stOtrok]
    else: starsi[stars1][0] += stOtrok; starsi[stars1][1] += stOtrok
    if stars2 not in starsi: starsi[stars2] = [stOtrok, stOtrok]
    else:
      starsi[stars2][0] += stOtrok
      if stars2 != stars1: starsi[stars2][1] += stOtrok
  # Poglejmo, kateri tip v prejšnji generaciji je imel največ potomcev.
  naj = 0; kdoNaj = ""
  for (stars, [_, pogostost]) in starsi.items():
    if pogostost > naj: naj = pogostost; kdoNaj = stars
  print("%d različnih, najpogostejši je %s (pri %d otrocih)" %
                                                    (len(starsi), kdoNaj, naj))
  if len(kdoNaj) <= 1: break
  otroci = starsi
```

7. Rekonstrukcija drevesa

Vhodno zaporedje, ki ga dobi naš postopek (in ki je bilo dobljeno z obratnim obhodom po drevesu) označimo z $a_1, \ldots, a_n$.

Pri obratnem obhodu vedno najprej obdelamo obe poddrevesi nekega vozlišča in šele nato izpišemo tisto vozlišče samo; iz tega sledi, da koren drevesa izpišemo nazadnje, čisto na koncu obhoda. Tako torej vemo, da je a_n koren drevesa, vse pred njim pa je opis njegovih poddreves, in sicer je (za nek k) $a_1, \ldots, a_{k-1}$ obratni obhod levega poddrevesa, $a_k, \ldots, a_{n-1}$ pa je obratni obhod desnega poddrevesa. Težava je le v tem, da ne poznamo števila k, torej ne vemo, kje v zaporedju se konča opis levega poddrevesa in začne opis desnega poddrevesa.

Spomnimo se, da nam naloga zagotavlja, da imamo opravka z binarnim iskalnim drevesom, kar pomeni, da so vse vrednosti v levem poddrevesu manjše od korena, vse v desnem poddrevesu pa večje od korena. Lahko se torej sprehodimo po zaporedju $a_1, \ldots, a_{n-1}$ od leve proti desni in preverjamo, ali so vrednosti še vedno manjše od korena (torej od a_n); čim naletimo na vrednost, ki je večja od korena, vemo, da se je zdaj končal opis levega poddrevesa in začel opis desnega poddrevesa.

Zdaj točno vemo, kateri del zaporedja opisuje levo poddrevo in kateri desno poddrevo; izpišimo koren v izhodno zaporedje (ki bo predstavljalo premi obhod drevesa), nato pa z rekurzivnim klicem obdelajmo najprej levo poddrevo in nato desno poddrevo, da sestavimo še preostanek premega obhoda. Zapišimo naš postopek še s psevdokodo:

algoritem ObdelajPoddrevo(i, j):
(* *Ta postopek obdela poddrevo, ki ga predstavlja zaporedje* $a_i, \ldots, a_j$. *)
1 $k := i$; **while** $k < j$ **and** $a_k < a_j$ **do** $k := k + 1$;
2 izpiši a_j;
3 **if** $k > i$ **then** ObdelajPoddrevo(i, $k - 1$);
4 **if** $k < j$ **then** ObdelajPoddrevo(k, $j - 1$);

Glavni klic, s katerim bi obdelali celotno drevo, je potem ObdelajPoddrevo($1, n$).

Potencialna slabost dosedanje rešitve je, da lahko zanka po k porabi pri vsakem vozlišču $O(n)$ časa, tako da je časovna zahtevnost celotnega postopka skupaj $O(n^2)$.

(To se zgodi, če je levo drevo vedno zelo veliko v primerjavi z desnim, npr. če je drevo izrojeno v seznam.) Učinkovitejši način za določanje k-ja je z bisekcijo: razdelimo zaporedje $a_i, \ldots, a_{j-1}$ na dve polovici in poglejmo srednji element; če je ta manjši od korena (torej od a_j), pomeni, da leva polovica v celoti leži znotraj levega poddrevesa, zato meja med poddrevesoma leži v desni polovici; podobno pa, če je srednji element večji od korena, to pomeni, da meja med poddrevesoma leži v levi polovici. Zdaj torej vemo, v kateri polovici leži meja in enak razmislek ponovimo na njej; tako nadaljujemo, dokler ne določimo točnega položaja meje. Za določitev meje zdaj pri vsakem poddrevesu porabimo le $O(\log n)$ časa, zato je skupna časovna zahtevnost le $O(n \log n)$. Zapišimo našo bisekcijo še s psevdokodo (z njo moramo nadomestiti vrstico 1 prejšnjega postopka):

$l := i - 1$; $k := j$;
while $k - l > 1$:
  (* *Na tem mestu velja, da so elementi* $a_i, \ldots, a_l$ *vsi manjši od* a_j,
   elementi $a_k, \ldots, a_j$ *pa so vsi večji ali enaki* a_j.
  $m := \lceil (l + k)/2 \rceil$; (* *Zaokrožimo navzgor, da bo* m *gotovo* $\geq i$. *)
  if $a_m < a_j$ **then** $l := m$ **else** $k := m$;

Ta postopek pravzaprav v zaporedju $a_i, \ldots, a_j$ išče prvi tak indeks k, za katerega je $a_k \geq a_j$. (Če to zgodi šele pri $k = j$, to pomeni, da je desno poddrevo prazno.) Na začetku vsake iteracije naše zanke vemo, da so elementi $a_i, \ldots, a_l$ vsi manjši od a_j, elementi $a_k, \ldots, a_j$ pa so vsi večji ali enaki a_j. Iskani indeks bo torej nekje na območju $a_{l+1}, \ldots, a_k$; v novi iteraciji zanke pogledamo element na sredini tega zaporedja (indeks m) in enega od indeksov l in k postavimo nanj, tako da se območje, na katerem bi še utegnil biti iskani indeks, razpolovi. Ob koncu izvajanja te zanke imamo $l = k - 1$ in skupaj z invarianto nam to pove, da so vsi elementi pred a_k manjši od a_j, element a_k pa je večji ali enak a_j, torej je k res prav tisti indeks, ki smo ga iskali.

Do še boljše rešitve lahko pridemo, če smo pripravljeni porabiti nekaj dodatnega pomnilnika. Oštevilčimo vozlišča našega drevesa s števili od 1 do n v takem vrstnem redu, v kakršnem se pojavljajo njihove vrednosti v obratnem obhodu. Oglejmo si zdaj vozlišče u; v obratnem obhodu se tik pred njim pojavlja vozlišče $u - 1$; kaj lahko povemo o odnosu med u in $u - 1$?

(1) Iz definicije obratnega obhoda vemo, da če u sploh ima desnega otroka, se ta otrok pojavlja v obratnem obhodu tik pred njim, torej kot $u - 1$. To je posledica dejstva, da je moral biti tak desni otrok izpisan na koncu obratnega obhoda po desnem poddrevesu vozlišča u, ta obhod pa je bil izveden tik pred tem, preden smo izpisali u.

(2) Če u nima desnega otroka, ima pa levega, nam podoben razmislek pove, da mora biti ta levi otrok ravno na indeksu $u - 1$.

(3) Kaj pa, če u sploh nima otrok, torej če je to eden od listov našega drevesa? Označimo u-jevega starša s p. Če ima u levega brata v (torej če ima p dva otroka, levega v in desnega u), je bilo pri obratnem obhodu po p-jevem poddrevesu najprej obdelano v-jevo poddrevo (na koncu tega smo izpisali v), nato pa u-jevo (ki ga sestavlja le vozlišče u samo, saj smo rekli, da je u list); tako torej vidimo, da je tik pred u-jem v obratnem obhodu prišel ravno u-jev levi brat v. Če pa u nima levega brata (bodisi zato, ker je u levi otrok p-ja ali pa je u sicer desni otrok p-ja, vendar

ta nima levega otroka), se lahko s podobnim razmislekom prepričamo, da mora biti v obratnem obhodu tik pred u-jem levi brat p-ja; če pa tudi ta nima levega brata, mora biti tik pred u-jem levi brat p-jevega starša in tako naprej.

Kako vemo, s katero od teh treh možnosti imamo opraviti? Pri (1), ko je $u-1$ desni otrok vozlišča u, mora veljati, da je $a_{u-1} > a_u$ (saj je po definiciji binarnega iskalnega drevesa vrednost v vozlišču manjša od tiste v desnem otroku). Pri (2) je $u-1$ levi otrok vozlišča u, torej je gotovo $a_{u-1} < a_u$. Pri (3) velja podobno: če je na primer $u-1$ levi brat vozlišča u, mora veljati $a_{u-1} < a_p < a_u$, torej spet $a_{u-1} < a_u$; podobno se lahko prepričamo tudi v primeru, ko je $u-1$ levi brat p-jevega starša ali kakšnega še bolj oddaljenega prednika.

Možnosti (1) torej ni težko ločiti od (2) in (3); kako pa lahko ločimo slednji dve med sabo? Bistvena razlika med njima je, da pri (2) vozlišče $u-1$ leži v u-jevem poddrevesu, pri (3) pa ne in leži nekje levo od njega (ker je $u-1$ levi brat nekega u-jevega prednika ali pa celo u-ja samega), torej je gotovo manjše od vsega, kar leži v u-jevem poddrevesu. Koristno bi bilo torej, če bi poznali neko spodnjo mejo za vsebino u-jevega poddrevesa, torej če bi vedeli, da so vse vrednosti v u-jevem poddrevesu večje od tiste meje in da je vse, kar je v drevesu levo od u-jevega poddrevesa, manjše ali enako tej meji. Take meje pa ni težko dobiti: če je u desni otrok svojega starša p, je primerna meja kar a_p; če pa je u levi otrok svojega starša, je primerna meja kar tista, ki je bila primerna tudi za p in jo lahko prenašamo navzdol ob rekurzivnem klicu.

Zdaj znamo torej ugotoviti, ali je $u-1$ otrok vozlišča u (in kateri otrok je). Če je to u-jev desni otrok, lahko zdaj njegovo poddrevo obdelamo z rekurzivnim klicem; ko se ta klic vrne, vemo, da je vozlišče, pri katerem smo se ustavili (v vhodnem seznamu, dobljenem z obratnim obhodom), ravno u-jev levi otrok (če ga ima). Zdaj torej smo pri levem otroku (če ga u ima) in lahko tudi njegovo poddrevo obdelamo z rekurzivnim klicem. S tem je u-jevo poddrevo v celoti obdelano in se lahko vrnemo iz rekurzivnega klica za u.

Tako smo prišli do naslednjega postopka, ki za vsako vozlišče u določi, na katerem indeksu $L[u]$ v zaporedju leži njegov levi otrok (če ga sploh ima):

algoritem NajdiLeveOtroke(indeks i, meja m):
  (* *Opomba: i predstavlja naš trenutni položaj v vhodnem zaporedju;*
    prenaša naj se po referenci oz. ga imejmo v neki globalni spremenljivki,
    tako da se bodo spremembe, ki jih naredimo v vgnezdenih
    rekurzivnih klicih, poznale tudi navzven.
  $u := i$; (* *Trenutno vozlišče.* *)
  $i := i - 1$; (* *Premaknimo se nazaj po zaporedju.* *)
  if $i > 0$ **and** $a_i > a_u$:
    (* *Vozlišče i je desni otrok vozlišča u.* *)
    NajdiLeveOtroke(i, a_u);
  if $i > 0$ **and** $a_i > m$:
    (* *Vozlišče i je levi otrok vozlišča u.* *)
    $L[u] := i$; NajdiLeveOtroke(i, m);
  else $L[u] := -1$; (* *u sploh nima levega otroka.* *)

Postopek poženemo z $i = n$ in mejo $m = -\infty$ (oz. s poljubno tako mejo, ki je manjša od vseh elementov drevesa). Ob koncu izvajanja bo $i = 0$ in v tabeli L bomo za

vsa vozlišča dobili indeks njihovih levih otrok (ali -1, če levega otroka ni). Lepo pri tem postopku je, da porabi le $O(n)$ časa, saj se ob vsakem rekurzivnem klicu pomaknemo za eno mesto nazaj po zaporedju (i se zmanjša za 1).

S pomočjo te tabele ni težko obhoditi drevesa v premem obhodu. Ko smo v vozlišču u in bi se radi premaknili v enega od otrok, zdaj točno vemo, kje ju najdemo: levega otroka najdemo na $L[u]$ (če ta indeks ni -1), desnega pa na $u-1$ (če je $a_{u-1} > a_u$).

algoritem PremiObhod(vozlišče u):
  izpiši a_u;
  if $L[u] > 0$ **then**
    PremiObhod($L[u]$); (* *Obdelaj levo poddrevo.* *)
  if $u > 1$ **and** $a_{u-1} > a_u$ **then**
    PremiObhod($u-1$); (* *Obdelaj desno poddrevo.* *)

Tudi ta izpis je porabil le $O(n)$ časa, tako da je skupna časovna zahtevnost naše nove rešitve le $O(n)$ namesto $O(n \log n)$ ali celo $O(n^2)$.

8. Avtomobil na daljinsko vodenje

Za vsakega od štirih gumbov izvedimo naslednji poskus: zapomnimo si trenutni položaj; pritisnimo gumb; odčitajmo novi položaj. Pri dveh od teh štirih gumbov bomo opazili, da se je položaj zaradi pritiska na gumb spremenil; enega od teh gumbov razglasimo za naprej, drugega za nazaj (vseeno je, kateri je kateri). Pri preostalih dveh gumbih se položaj ne spremeni, saj predstavljata obrat na mestu; moramo pa še ugotoviti, kateri gumb je za obrat v levo, kateri pa za v desno.

Naj bo zdaj $r_0 = (x_0, y_0)$ trenutni položaj; pritisnimo naprej; naj bo $r_1 = (x_1, y_1)$ novi položaj; pritisnimo še enega od preostalih dveh gumbov (recimo mu G) in nato še enkrat naprej; naj bo $r_2 = (x_2, y_2)$ novi položaj. Če je premik iz smeri $(r_1 - r_0)$ v smer $(r_2 - r_1)$ bil premik v levo, potem vemo, da je bil gumb $G =$ levo, sicer pa desno. Za preostali gumb potem tudi vemo, da pomeni zasuk v ravno nasprotno smer kot G.

Smer zasuka lahko ugotovimo z vektorskim produktom: če je $(x_1 - x_0)(y_2 - y_1) > (y_1 - y_0)(x_2 - x_1)$, je bil to zasuk v levo, sicer pa v desno (ob predpostavki, da y-os kaže navzgor). Lahko pa seveda tudi za vsakega od vektorjev $(x_1 - x_0, y_1 - y_0)$ in $(x_2 - x_1, y_2 - y_1)$ z nekaj pogojnimi stavki preverimo, v katero od štirih možnih smeri kaže — torej $(1, 0)$, $(0, 1)$, $(-1, 0)$ in $(0, -1)$.

Zdaj torej tudi vemo, da je avto na koordinatah (x_2, y_2) in obrnjen v smer $(x_2 - x_1, y_2 - y_1)$. Ker poznamo delovanje vseh gumbov, avta ni težko odpeljati proti cilju.[18] Avto obrnemo v smer $(1, 0)$ in ga premikamo naprej (če je $x_c > x_2$) oz. nazaj (če je $x_c < x_2$), dokler x-koordinata avtomobila ne doseže x_c. Nato naredimo podobno še za y-koordinate — obrnemo avto v smer $(0, 1)$ in ga premikamo naprej oz. nazaj, dokler še njegova y-koordinata ne doseže y_c.

[18] Mimogrede, v resnici ne moremo zanesljivo ugotoviti, kako je avto obrnjen. Lahko se namreč zgodi, da gumb naprej v resnici premakne avto nazaj, gumb nazaj pa naprej; v tem primeru tudi velja, da ga levo obrne desno, desno pa levo. Tega primera ne moremo ločiti od tistega, ko naprej res premakne avto naprej itd.; vendar pa lahko avtomobil še vseeno usmerjamo in točno vemo, kako se bo premikal.

9. Tat in laserji

Recimo, da imamo n premic, ki opisujejo žarke: $p_1, \ldots, p_n$. Vsako celico c lahko zdaj opišemo z n-terico bitov $(c_1, \ldots, c_n)$, pri čemer c_i pove, na kateri strani premice p_i leži celica c (to, katero stran premice označimo z 0 in katero z 1, ni pomembno).

Mislimo si dve sosednji celici c in c', ki ju razmejuje premica p_k. Če stopimo čez to premico iz c v c' (ali obratno), se je spremenilo to, na kateri strani premice p_k se nahajamo; glede ostalih premic pa se ni spremenilo nič (saj nismo prečkali nobene od njih). Torej se c in c' razlikujeta le v bitu k: $c'_k = 1 - c_k$, za vse $i \neq k$ pa je $c_i = c'_i$.

Naj bo z celica, v kateri je zaklad; za poljubno celico c definirajmo $d(c)$ kot število istoležnih bitov, v katerih se c razlikuje od z (torej pri koliko indeksih i velja $c_i \neq z_i$). Med drugim lahko opazimo, da če sta c in c' sosednji celici, potem zanju velja $d(c') = d(c) \pm 1$ (recimo, da je k tisti edini bit, v katerem se c in c' razlikujeta; potem, če se v tem bitu razlikujeta tudi c in z, je $d(c') = d(c) - 1$; če pa se c in z v tem bitu ujemata, je zdaj $d(c') = d(c) + 1$).

Naj bo t celica, v kateri stoji tat na začetku; ta se torej od z razlikuje v $d(t)$ bitih. Kot smo videli v prejšnjem odstavku, lahko tat v vsakem koraku spremeni le en bit, torej bo potreboval vsaj $d(t)$ korakov, da pride do celice z. Ali se lahko zgodi, da bi potreboval strogo več kot $d(t)$ korakov? Izkaže se, da ne; prepričali se bomo, da je mogoče v vsakem trenutku narediti tak korak iz trenutne celice c v eno od njenih sosed c', pri katerem se d zmanjša za 1, torej da je $d(c') = d(c) - 1$.

Pa recimo, da to ne drži; torej da se trenutno nahajamo v neki celici c in da za vsako njeno sosedo c' velja $d(c') = d(c) + 1$. Vsaka celica ima obliko konveksnega mnogokotnika. Oglejmo si stranice celice c; naj bo P množica premic, na katerih ležijo te stranice. Izberimo si poljubno premico $p_k \in P$; naj bo c' tista soseda celice c, v katero pridemo iz c, če prečkamo premico p_k. Torej se c in c' razlikujeta v bitu k (in v nobenem drugem). Po predpostavki velja $d(c') = d(c) + 1$, torej mora biti k eden od tistih bitov, v katerem se je c ujemal z z (c' pa se v tem bitu razlikuje od z). Ta razmislek velja za vsako $p_k \in P$, torej lahko zaključimo, da se c ujema z z v vseh bitih, ki pripadajo premicam iz P.

Vsaka premica $p_k \in P$ razdeli ravnino na dve polravnini; označimo ju z A_{k0} in A_{k1} (točk, ki ležijo prav na premici p_k, ne štejmo k nobeni od teh dveh polravnin). Bit c_k nam pove, v kateri od teh dveh polravnin leži celica c, torej je $c \subseteq A_{k,c_k}$. To velja za vsako premico iz P, zato je c tudi podmnožica preseka $\cap_{p_k \in P} A_{k,c_k}$. Ta presek označimo z B. Podobno velja tudi za z, saj smo v prejšnjem odstavku videli, da se c in z ujemata v vseh bitih, ki pripadajo premicam iz P: torej imamo $z \subseteq \cap_{p_k \in P} A_{k,z_k} = \cap_{p_k \in P} A_{k,c_k} = B$.

Vsaka polravnina je konveksna množica, presek več konveksnih množic pa je tudi sam konveksna množica; torej je B konveksna množica; torej za poljubni dve točki U in V iz B velja, da tudi daljica UV v celoti leži znotraj B. Izberimo si za U poljubno točko iz notranjosti celice c, za V pa poljubno točko iz notranjosti celice z. Ker sta c in z podmnožici množice B, to pomeni, da točki U in V ležita v B, torej mora (ker je B konveksna) tudi daljica UV ležati znotraj B. Če se po daljici UV počasi premikamo od U proti V, se na začetku te poti nahajamo v celici c, prej ali slej pa jo bomo zapustili, saj se na koncu te poti znajdemo v celici z. Ko prvič zapustimo celico c, se to zgodi tako, da prečkamo neko premico $p_k \in P$. Zdaj

torej nismo več na isti strani premice p_k kot prej; prej smo bili v A_{k,c_k}, zdaj pa smo očitno na nasprotni strani, v $A_{k,1-c_k}$. Toda obenem smo še vedno na daljici UV, ki v celoti leži znotraj B, ta pa je naprej $\subseteq A_{k,c_k}$. Torej smo zdaj hkrati v $A_{k,1-c_k}$ in A_{k,c_k}, kar pa je protislovje, saj neka točka ne more hkrati ležati na obeh straneh premice p_k.

Naša predpostavka (da za vse c-jeve sosede c' velja $d(c') = d(c) + 1$) nas je pripeljala v protislovje, torej je napačna; tako torej vidimo, da za vsako c obstaja vsaj ena soseda, ki nas pripelje bližje k ciljni celici z. Najmanjše število korakov, ki jih tat potrebuje, da doseže zaklad, je torej $d(t)$.

Naloga pravzaprav sprašuje, koliko detektorjev mora tat izključiti, zato je verjetno treba prišteti še detektor v celici t, kjer tat stoji na začetku; pravilni odgovor je torej $d(t) + 1$. Kakorkoli že, $d(t)$ ni težko izračunati; kot smo videli že zgoraj, moramo le pogledati, pri koliko premicah stojita tat in zaklad na nasprotnih straneh te premice. Če je na primer premica podana z enačbo $ax + by + c = 0$, bi za točke (x, y) na eni strani premice dobili $ax + by + c > 0$, za točke na drugi strani premice pa $ax + by + c < 0$. Vse, kar moramo pri posamezni premici narediti, je torej to, da vstavimo koordinate tatu in zaklada v izraz $ax + by + c$ tiste premice in preverimo, ali imata dobljeni vrednosti različen predznak.

10. Malica

Začnimo s konkretnim primerom. Recimo, da gremo po dveh povezavah, e_1 in e_2; verjetnost, da nas oropajo na prvi povezavi, je 0,1 (torej 10 %), na drugi povezavi pa 0,2 (torej 20 %). Kakšna je verjetnost, da nas ne oropajo? Predpostavimo, da so verjetnosti ropa na različnih povezavah med seboj neodvisne. V 10 % primerov nas oropajo že na prvi povezavi; v ostalih 90 % primerih pa se nam potem v 20 % (od teh 90 %, kar je 18 % od celote) zgodi, da nas oropajo na drugi povezavi. Skupna verjetnost tega, da nas oropajo vsaj enkrat, je torej 10 % + 18 % = 28 %; ostane pa 100 % − 28 % = 72 % možnosti, da nas ne oropajo na nobeni od obeh povezav.

Lažja pot do rešitve pa je, če ta razmislek malo obrnemo: verjetnost, da nas na prvi povezavi pustijo pri miru, je $1 - 0{,}1 = 0{,}9$ (torej 90 %); na drugi povezavi je ta verjetnost $1 - 0{,}2 = 0{,}8$ (torej 80 %); verjetnost, da nas pustijo pri miru na obeh povezavah, je zato $0{,}9 \cdot 0{,}8 = 0{,}72 = 72\,\%$.

Ta razmislek lahko tudi posplošimo: recimo, da je p_1 verjetnost ropa na prvi povezavi, p_2 pa na drugi. Potem je verjetnost, da nas na prvi povezavi ne oropajo, enaka $1 - p_1$, na drugi povezavi pa $1 - p_2$. Verjetnost, da nas ne oropajo na nobeni, je potem $(1 - p_1) \cdot (1 - p_2)$.[19] Tako lahko nadaljujemo tudi za daljše poti: če imamo k povezav z verjetnostmi ropa $p_1, \ldots, p_k$, potem je verjetnost, da nas na nobeni od njih ne oropajo, enaka $\prod_{i=1}^{k}(1 - p_i) = (1 - p_1)(1 - p_2) \cdots (1 - p_k)$.[20]

[19] Če bi namesto tega uporabili razmislek iz prvega odstavka, bi morali od 1 odšteti verjetnost ropa na prvi povezavi, torej p_1, in verjetnost, da nas oropajo šele na drugi povezavi (na prvi pa ne), to pa je $(1 - p_1) \cdot p_2$. Rezultat bi bil torej $1 - p_1 - (1 - p_1) \cdot p_2$; hitro lahko vidimo, da je to enako $(1 - p_1) \cdot (1 - p_2)$.

[20] Razmislek iz prvega odstavka bi nam v tem primeru pokazal, da je verjetnost, da nas nobenkrat ne oropajo, enaka $1 - \sum_{i=1}^{k} p_i \prod_{j=1}^{i-1}(1 - p_j)$. Tudi zdaj se lahko prepričamo, da se oba rezultata ujemata. Če prvi člen vsote zapišemo posebej, dobimo $1 - p_1 - \sum_{i=2}^{k} p_i \prod_{j=1}^{i-1}(1 - p_j)$; zdaj lahko izpostavimo faktor $1 - p_1$ in dobimo $(1 - p_1)\left[1 - \sum_{i=2}^{k} p_i \prod_{j=2}^{i-1}(1 - p_j)\right]$. Izraz v oglatih

Radi bi torej našli pot (od vozlišča 1 do vozlišča n) z največjim možnim zmnožkom $\prod_i (1 - p_i)$. Spomnimo se, da je logaritem strogo naraščajoča funkcija; zato lahko, namesto da maksimiziramo neko količino, maksimiziramo njen logaritem, pa bo rezultat enak. V našem primeru torej lahko maksimiziramo $\log \prod_i (1-p_i)$, kar je enako $\sum_i \log(1 - p_i)$. Lahko pa posamezne člene te vsote pomnožimo z -1 in nato vsoto minimiziramo namesto maksimiziramo: iščemo torej pot z najmanjšo vsoto $\sum_i (-\log(1-p_i))$. To pa ni nič drugega kot najkrajša pot (od vozlišča 1 do vozlišča n), če si vrednost $-\log(1 - p_i)$ predstavljamo kot dolžino i-te povezave.

Tako torej vidimo, da lahko našo nalogo rešimo tako, da vsaki povezavi z verjetnostjo ropa p_i pripišemo dolžino $-\log(1-p)$ in v tako dopolnjenem grafu poiščemo najkrajšo pot od 1 do n. Za iskanje najkrajših poti lahko uporabimo kakšnega od dobro znanih algoritmov, na primer Dijkstrovega.

11. 3-d šah

Ogledali si bomo več rešitev te naloge, od preprostejših (in počasnejših) do malo bolj zapletenih (vendar učinkovitejših).

(1) Rešitev s pregledom celotne šahovnice. Preprosta, vendar neučinkovita rešitev je, da se sprehodimo po vseh poljih šahovnice in za vsako polje preverimo, ali ga napada vsaj ena od kraljic.[21]

```
N := 0;
for x := 1 to n do
  for y := 1 to n do
    for z := 1 to n do
      napadeno := false;
      for k := 1 to m do
        if kraljica k napada polje (x, y, z) then napadeno := true;
      if napadeno then N := N + 1;
```

Na koncu tega postopka imamo v N skupno število vseh napadenih polj.

Razmisliti moramo še o tem, kako preveriti, ali neka kraljica napada opazovano polje. Smer napada lahko opišemo s trojico $(\triangle_x, \triangle_y, \triangle_z)$, pri čemer je vsaka od komponent lahko enaka -1, $+1$ ali 0 in nam torej pove, ali se tista koordinata v tej smeri zmanjšuje, povečuje ali ostaja nespremenjena. Če stoji kraljica na polju (x_k, y_k, z_k) in napada v smeri $(\triangle_x, \triangle_y, \triangle_z)$, bodo imela napadena polja koordinate oblike $(x_k + t\triangle_x, y_k + t\triangle_y, z_k + t\triangle_z)$ za $t = 1, 2, 3, \ldots$. Polje (x, y, z) je torej napadeno, če ga je mogoče izraziti v tej obliki za nek $t \geq 1$. V tem primeru je razlika $x - x_k$ enaka $t\triangle_x$, kar je naprej enako t (če je $\triangle_x = +1$) ali $-t$ (če je $\triangle_x = -1$) ali 0 (če je $\triangle_x = 0$). Podobno velja tudi pri y- in z-koordinati. Napadeno polje torej prepoznamo po tem, da so si vse neničelne vrednosti v vektorju $(x-x_k, y-y_k, z-z_k)$ po absolutni vrednosti enake.

Opisana rešitev deluje za tri dimenzije, prav lahko pa bi jo posplošili tudi na d dimenzij; takrat bi imeli d vgnezdenih zank od 1 do n, lahko pa bi namesto tega uporabili rekurzijo.

---

oklepajih je zdaj enake oblike kot prvotni izraz na začetku, le da se indeksi začnejo pri 2 namesto pri 1. Na ta način bi lahko nadaljevali in vse skupaj sčasoma predelali v $(1-p_1)(1-p_2)\cdots(1-p_k)$.

[21] Takšno rešitev smo opisali že pri nalogi 2013.S.2 na str. 76–78 v biltenu 2013.

Slabost te rešitve je, da ima šahovnica n^d polj, za vsako polje moramo iti po vseh m kraljicah in pri vsaki kraljici imamo $O(d)$ dela, da preverimo, ali napada naše polje ali ne. Časovna zahtevnost te rešitve je torej kar $O(n^d\, md)$. V prvotni različici naloge, kjer imamo tri kraljice in trodimenzionalno šahovnico (torej $m = d = 3$), je ta časovna zahtevnost $O(n^3)$. Ta pristop bi bil dober za majhne šahovnice, naša naloga pa posebej poudarja, da nas zanima rešitev, ki bo učinkovita tudi pri velikih n.

(2) Rešitev s pregledom napadenih polj. Malo boljša rešitev je, da se za začetek omejimo na eno samo kraljico in naštejemo vsa polja, ki jih ta kraljica napada. V ta namen ni treba pregledati vseh polj šahovnice. Zgoraj smo že videli, da kraljica na polju (x_k, y_k, z_k) v smeri $(\triangle_x, \triangle_y, \triangle_z)$ napada polja oblike $(x_k + t\triangle_x, y_k + t\triangle_y, z_k + t\triangle_z)$ za $t \geq 1$. Ta polja lahko preštejemo tako, da v zanki pregledujemo vse večje t, dokler ne pademo čez rob šahovnice (torej se ustavimo, čim kakšna od koordinat polja pade pod 1 ali naraste nad n).

V treh dimenzijah je možnih smeri napada 26 ali 27, ker so za vsako od $\triangle_x$, $\triangle_y$ in $\triangle_z$ tri možnosti: 0, +1 in −1; tako je skupaj $3 \cdot 3 \cdot 3 = 27$ možnosti, vendar je smer $(0, 0, 0)$ poseben primer, saj z njo nikamor ne pridemo. Smer $(0, 0, 0)$ bomo v spodnjem postopku vseeno tudi upoštevali, saj bomo z njo šteli polja, na katerih stojijo kraljice same (tudi ta polja namreč veljajo za napadena); moramo pa pri njej paziti, da ne pregledamo več kot enega t-ja, saj tam za vsak t dobimo eno in isto polje (namreč tisto, na katerem stoji kraljica).

Ta postopek lahko ponovimo še za ostale kraljice, paziti pa moramo na to, da ne bomo polj, ki jih napada več kot ena kraljica, šteli po večkrat. Zato moramo pri vsakem polju še preveriti, če ga napada kakšna od prej obdelanih kraljic, in takih polj ne smemo šteti še enkrat. Tako dobimo naslednji postopek:

$N := 0$;
for $k := 1$ **to** m:
  za vsako možno smer napada $(\triangle_x, \triangle_y, \triangle_z) \in \{-1, 0, 1\}^3$:
    $t := 1$;
    while polje $(x_k + t\triangle_x, y_k + t\triangle_y, z_k + t\triangle_z)$ leži na šahovnici:
      novo := **true**;
      for $k' := 1$ **to** $k - 1$:
        if kraljica k' napada polje $(x_k + t\triangle_x, y_k + t\triangle_y, z_k + t\triangle_z)$:
          novo := **false**; **break**;
      if *novo* **then** $N := N + 1$;
      if $\triangle_x = \triangle_y = \triangle_z = 0$ **then break else** $t := t + 1$;

Tudi pri tem postopku lahko vidimo, da ga ne bi bilo težko posplošiti na več dimenzij; število možnih smeri napada bi bilo tedaj 3^d. Da ocenimo časovno zahtevnost tega postopka, si oglejmo, koliko ponovitev izvedejo naše vgnezdene zanke: imamo zanko po m kraljicah, v njej zanko po vseh 3^d smereh, v vsaki smeri pregledamo (z zanko **while**) največ $O(n)$ polj, pri vsakem polju gremo po $O(m)$ prejšnjih kraljicah in za vsako od njih porabimo $O(d)$ časa, da preverimo, ali že ona napada trenutno polje. Če vse to zmnožimo, dobimo časovno zahtevnost $O(3^d\, m^2\, nd)$.[22] V prvotni različici

[22] Omenimo lahko še eno različico te rešitve: koordinate vsakega napadenega polja dodajmo v razpršeno tabelo (*hash table*). Zato nam ni treba iti z zanko po vseh prejšnjih kraljicah (s

naloge, kjer je število dimenzij in število kraljic fiksno ($d = m = 3$), je ta časovna zahtevnost le $O(n)$. To je vsekakor veliko bolje od prejšnje rešitve.

(3) Rešitev z načelom vključitev in izključitev. Dosedanjo rešitev lahko še izboljšamo. Omejimo se spet na eno samo kraljico; ko si izberemo neko konkretno smer napada in nas zanima, koliko polj napada v tej smeri, nam teh polj pravzaprav ni treba naštevati, ampak lahko njihovo število preprosto izračunamo. Kraljica (x_k, y_k, z_k) v smeri $(\triangle_x, \triangle_y, \triangle_z)$ napada polja oblike $(x_k + t\triangle_x, y_k + t\triangle_y, z_k + t\triangle_z)$, pri čemer so dopustni tisti t, pri katerih vse tri koordinate ležijo znotraj območja $\{1, \ldots, n\}$. Pri x-koordinati to na primer pomeni $t \leq n - x_k$ (če je $\triangle_x = +1$) oz. $t \leq 1 - x_k$ (če je $\triangle_x = -1$). Podobno naredimo še z y- in z-koordinatami in med vsemi tako dobljenimi zgornjimi mejami za t vzamemo najnižjo; recimo ji t^{max}; v tej smeri gre torej lahko t od 1 do t^{max}, torej je v njej napadenih t^{max} polj (sem ni všteto polje, na katerem stoji kraljica). Če ta razmislek ponovimo za vseh 26 možnih smeri in rezultate seštejemo, na koncu pa dodamo še polje, na katerem kraljica stoji, dobimo skupno število polj, ki jih ta kraljica napada.

Podobno lahko preštejemo tudi polja, ki jih napada druga kraljica, in polja, ki jih napada tretja kraljica. Število napadenih polj po vseh treh kraljicah moramo zdaj načeloma sešteti, težava pri tem pa je, da smo zdaj nekatera polja šteli po dvakrat ali trikrat (če jih napadata dve ali tri kraljice). Ta moramo zdaj torej spet odšteti. (Pri prejšnji rešitvi te težave ni bilo, ker smo tam vsa napadena polja pregledali in sproti preverjali, če neko polje napada že kakšna od prej obdelanih kraljic. Tukaj pa polj ne pregledujemo, ampak le izračunamo, koliko jih je.)

Oglejmo si prvi dve kraljici in poskusimo prešteti, koliko polj napadata obe hkrati. Za vsako od kraljic si izberimo eno od 26 možnih smeri napada; recimo, da kraljica k (za $k = 1, 2$) stoji na polju $\mathbf{r}_k = (x_k, y_k, z_k)$ in napada v smeri $\mathbf{s}_k = (\triangle_x^{(k)}, \triangle_y^{(k)}, \triangle_z^{(k)})$. Polja, ki jih kraljica k v izbrani smeri napada, tvorijo daljico oblike $\mathbf{r}_k + t_k\mathbf{s}_k$ (za $1 \leq t_k \leq t_k^{max}$; to, kako določiti t_k^{max}, smo videli že zgoraj). Polja, ki jih napadata obe kraljici, so tista, ki ležijo na preseku obeh daljic; tako polje prepoznamo po tem, da ga je mogoče izraziti v obliki $\mathbf{r}_k + t_k\mathbf{s}_k$ za oba k-ja hkrati, torej $\mathbf{r}_1 + t_1\mathbf{s}_1 = \mathbf{r}_2 + t_2\mathbf{s}_2$. Če to zapišemo po komponentah, dobimo $x_1 + t_1\triangle_x^{(1)} = x_2 + t_2\triangle_x^{(2)}$ in podobno še za y in z. Zdaj imamo torej sistem treh linearnih enačb z dvema neznankama (t_1 in t_2); poleg tega pa imamo še neenačbi $1 \leq t_k \leq t_k^{max}$ za $k = 1, 2$. Tega načeloma ni težko rešiti: eno od enačb uporabimo, da izrazimo t_2 s t_1; to vstavimo v ostali dve enačbi in v neenačbe; iz ostalih dveh enačb se potem lahko izkaže, da ju reši natanko en t_1 ali pa noben ali pa da je dober zanju vsak t_1;[23] tako dobljeni interval možnih t_1 nato še oklestimo s pomočjo omejitev, ki jih določajo neenačbe. Na koncu nam ostane nek interval možnih vrednosti t_1, za katerega ni težko izračunati, koliko vrednosti leži na njem.

To torej naredimo za vsak par smeri in rezultate seštejemo; če kraljici tudi na-

števcem k'), ampak moramo le preveriti, če imamo trenutno polje že v razpršeni tabeli ali ne. Takšno preverjanje in dodajanje v razpršeno tabelo nam pri vsakem polju vzame $O(d)$ časa, tako da je časovna zahtevnost celotnega postopka zdaj $O(3^d\, m\, nd)$, torej za faktor $O(m)$ manjša kot prej. Slabost pa je, da za razpršeno tabelo porabimo tudi $O(3^d\, m\, nd)$ pomnilnika, medtem ko so vse druge tu omenjene rešitve veliko varčnejše s prostorom (porabijo le $O(d)$ pomnilnika).

[23] To je pravzaprav poseben primer Gaussove eliminacijske metode, ki je bolj sistematičen postopek za reševanje poljubnega sistema linearnih enačb; za več o tem gl. npr. Wikipedijo *s. v.* Gaussian elimination.

padata druga drugo, prištejemo še obe polji, na katerih stojita; tako dobimo skupno število polj, ki jih napadata obe kraljici hkrati. Zdaj smo obdelali en par kraljic, podobno naredimo še z ostalima dvema (prvo in tretjo kraljico ter drugo in tretjo kraljico).

Ostane pa še ena težava: morebitna polja, ki jih napadajo vse tri kraljice, smo najprej šteli trikrat (pri vsaki kraljici po enkrat), nato pa smo jih tudi odšteli trikrat (pri vsakem od treh parov kraljic po enkrat). Zdaj moramo torej ta polja prišteti še enkrat. Njihovo število lahko ugotovimo na podoben način kot prej za dve kraljici, le da imamo zdaj več enačb kot prej: ko si izberemo smeri napada za vse tri kraljice ($\mathbf{s}_1$, $\mathbf{s}_2$, $\mathbf{s}_3$), morajo biti koordinate polja, ki ga napadajo vse tri kraljice, zdaj oblike $\mathbf{r}_k + t_k\mathbf{s}_i$ za $k = 1, 2, 3$; torej imamo $\mathbf{r}_1 + t_1\mathbf{s}_1 = \mathbf{r}_2 + t_2\mathbf{s}_2 = \mathbf{r}_3 + t_3\mathbf{s}_3$, v čemer se pravzaprav skrivata dve enačbi. Ko to zapišemo po komponentah, dobimo sistem šestih linearnih enačb s tremi neznankami (t_1, t_2 in t_3) in s podobnimi neenačbami kot prej ($1 \leq t_k \leq t_k^{max}$). Sistem rešimo podobno kot prej za dve neznanki; tudi zdaj je možna ena, nobena ali pa več rešitev. Ta razmislek moramo ponoviti za vsako možno kombinacijo smeri napadov vseh treh kraljic (to je kar $26 \cdot 26 \cdot 26 = 17\,576$ možnosti); na koncu pa še za vsako od polj, na katerem stoji kakšna kraljica, preverimo, če ga napadata tudi preostali dve.

Mimogrede, v jeziku teorije množic bi lahko našo rešitev opisali takole: naj bo A_k (za $k = 1, 2, 3$) množica polj, ki jih napada kraljica k. Naloga pravzaprav sprašuje, koliko elementov ima unija $A_1 \cup A_2 \cup A_3$. V naši rešitvi smo izkoristili dejstvo, da za poljubne množice A_i velja $|A_1 \cup A_2 \cup A_3| = |A_1| + |A_2| + |A_3| - |A_1 \cap A_2| - |A_1 \cap A_3| - |A_2 \cap A_3| + |A_1 \cap A_2 \cap A_3|$. To je poseben primer tehnike, ki je v matematiki znana kot *načelo vključitev in izključitev.*[24] Kot smo videli, lahko v našem primeru z nekaj truda izračunamo velikosti množic A_k in njihovih presekov zelo učinkovito, v času, ki sploh ni odvisen od velikosti šahovnice, torej n. Časovna zahtevnost te rešitve je torej $O(1)$ (seveda ob predpostavki, da imajo tudi posamezne aritmetične operacije na celih številih do n časovno zahtevnost le $O(1)$).

Doslej smo razmišljali o treh kraljicah na trorazsežni šahovnici; žal se ta rešitev izkaže za precej manj vabljivo, ko jo posplošimo na m kraljic in d dimenzij. Načelo vključitev in izključitev nam pri m kraljicah dá formulo

$$|\cup_{k=1}^{m} A_k| = \sum_{r=1}^{m} (-1)^{r-1} \sum_{K \subseteq \{1,\ldots,m\}:|K|=r} |\cap_{k\in K} A_k| \,.$$

Z drugimi besedami, najprej seštejemo velikosti vseh množic A_k, potem odštejemo velikosti vseh presekov po dveh takih množic, nato prištejemo velikosti vseh presekov po treh takih množic, nato odštejemo velikosti vseh presekov po štirih takih množic in tako naprej. (Bralec se lahko z indukcijo po m za vajo prepriča, da bi to res dalo pravi rezultat.) Naš postopek lahko zapišemo takole:

$N := 0$;
for $r := 1$ **to** m:
  za vsak nabor r kraljic $K = \{k_1, \ldots, k_r\}$,
  pri čemer je $1 \leq k_1 < k_2 < \ldots < k_r \leq m$:
    za vsak nabor r smeri $\mathbf{s}_1, \ldots, \mathbf{s}_r \in \{-1, 0, 1\}^d$:
      rešimo sistem enačb (in neenačb), s katerim ugotovimo, koliko polj

[24] Gl. npr. Wikipedijo *s. v.* Inclusion–exclusion principle.

je hkrati napadenih od kraljice k_i v smeri $\mathbf{s}_i$ za vsak $i = 1, \ldots, r$;
številu teh polj recimo q;
$N := N + (-1)^{r-1} \cdot q$;

Ko se ukvarjamo s preseki r množic, si lahko r kraljic izberemo na $\binom{m}{r}$ načinov; nato si lahko za vsako od teh r kraljic izberemo njeno smer napada na $O(3^d)$ načinov; nato dobimo sistem $O(r)$ linearnih enačb z r neznankami, ki ga lahko z Gaussovo eliminacijo rešimo v $O(r^3)$ časa. Tako smo prišli do časovne zahtevnosti $O(\sum_{r=1}^m \binom{m}{r} 3^{dr} r^3)$, kar je naprej približno enako $O(3^{dm} m^3)$.

Implementacijo te rešitve bi se dalo še malo izboljšati. Če si izberemo konkretnih r kraljic in za vsako od njih neko konkretno smer napada, se najbrž največkrat izkaže, da sploh ni nobenega takega polja, ki bi ga napadale vse te kraljice hkrati v izbranih smereh; še toliko bolj nesmiselno je, da se bomo kasneje (pri večjih k) ukvarjali z nabori, ki poleg vseh teh kraljic vsebujejo še kakšno dodatno. Še en vir neučinkovitosti je tudi ta, da se dva (ali več) različna nabora kraljic (in njihovih smeri) lahko razlikujeta le v nekaj kraljicah (in smereh), nekaj pa imata skupnih in je zato potratno, da vsak nabor obravnavamo popolnoma ločeno od drugih.

Bolj formalno lahko razmišljamo takole: množica A_k vseh polj, ki jih napada kraljica k, je oblike $\cup_{\mathbf{s}} A_{k\mathbf{s}}$, pri čemer gre unija po vseh smereh $\mathbf{s} \in \{-1, 0, +1\}^d$, oznaka $A_{k\mathbf{s}}$ pa pomeni množico vseh polj, ki jih kraljica k napada v smeri $\mathbf{s}$. (Polje, na katerem stoji kraljica k sama, štejmo v $A_{k\mathbf{s}}$ za $\mathbf{s} = (0, 0, \ldots, 0)$ in za nobeno drugo smer $\mathbf{s}$.) Opazimo lahko, da so množice $A_{k\mathbf{s}}$ (pri fiksnem k) med seboj disjunktne.

Naš gornji postopek poskuša pri vsakem naboru kraljic $K = \{k_1, \ldots, k_r\}$ računati velikost preseka $\cap_{i=1}^r A_{k_i} = \cap_{i=1}^r \cup_{\mathbf{s}_i} A_{k_i,\mathbf{s}_i} = \cup_{\mathbf{s}_1,\ldots,\mathbf{s}_r} \cap_{i=1}^r A_{k_i,\mathbf{s}_i}$. To velikost smemo računati kot $|\cap_{i=1}^r A_{k_i}| = \cup_{\mathbf{s}_1,\ldots,\mathbf{s}_r} |\cap_{i=1}^r A_{k_i,\mathbf{s}_i}|$ (in prav to naš postopek tudi počne), ker so množice $\cap_{i=1}^r A_{k_i,\mathbf{s}_i}$ za različne nabore smeri $\mathbf{s}_1, \ldots, \mathbf{s}_r$ med seboj disjuntne.[25]

Če bi zdaj namesto nabora kraljic K gledali malo večji nabor K', ki bi poleg vseh kraljic iz K vseboval še eno novo, bi pri njem prišli v poštev zelo podobni preseki $\cap_i A(k_i, \mathbf{s}_i)$ kot pri K, le da bi vsak tak presek obsegal še en člen več. Geometrijsko si lahko posamezno množico $A_{k_i,\mathbf{s}_i}$ predstavljamo kot daljico (ki lahko obsega tudi samo eno polje); pa tudi presek več takšnih množic je zato lahko le daljica (ali pa je presek prazen). Če v presek (ki je že zdaj daljica) dodamo še en člen oblike $A_{k_i,\mathbf{s}_i}$ (ki je tudi daljica), ni težko izračunati novega preseka (presek dveh daljic je točka, daljica ali pa je prazen). Če pri tem opazimo, da presek postane prazen, nam v nabor ni treba dodajati še dodatnih kraljic, saj bo presek tudi po tem ostal prazen. Tako dobimo naslednji rekurzivni postopek:

podprogram REKURZIJA:
vhod: dosedanji nabor kraljic $K = \{k_1, \ldots, k_r\}$ in naslednja kraljica k
(velja torej $1 \leq k_1 < \ldots < k_r < k$);
dosedanji nabor smeri $\mathbf{s}_1, \ldots, \mathbf{s}_r$, vsaka od njih je $\in \{-1, 0, +1\}^d$;
dosedanji presek $D = \cap_{i=1}^r A_{k_i,\mathbf{s}_i}$, opisan kot daljica;

[25] O tem se lahko prepričamo takole: mislimo si dva različna nabora smeri, recimo $(\mathbf{s}_1, \ldots, \mathbf{s}_r)$ in $(\mathbf{s}'_1, \ldots, \mathbf{s}'_r)$; ker sta različna, obstaja torej vsaj en tak indeks j, pri katerem je $\mathbf{s}_j \neq \mathbf{s}'_j$; tedaj sta $A_{k_j,\mathbf{s}_j}$ in $A_{k_j,\mathbf{s}'_j}$ disjuntkni; zato sta tudi preseka $\cap_{i=1}^r A_{k_i,\mathbf{s}_i}$ (ki je $\subseteq A_{k_j,\mathbf{s}_j}$) in $\cap_{i=1}^r A_{k_i,\mathbf{s}'_i}$ (ki je $\subseteq A_{k_j,\mathbf{s}'_j}$) disjunktna.

N — globalna spremenljivka s številom napadenih polj;

if $k > m$ **then** (* *Konec rekurzije; prištejmo ali odštejmo velikost preseka* D *)
$N := N + (-1)^{r-1} \cdot |D|$; **return**; (* *od globalne spremenljivke* N. *)
(* *Ena možnost je, da kraljice* k *ne dodamo v nabor* K. *)
Rekurzija(K, $k+1$, $\mathbf{s}_1, \ldots, \mathbf{s}_r$, D);
(* *Druga možnost je, da jo dodamo v* K*; izbrati ji moramo smer napada.* *)
za vsako možno smer napada $\mathbf{s}_{r+1} \in \{-1, 0, +1\}^d$:
naj bo D' presek daljice D in daljice $A_{k,\mathbf{s}_{r+1}}$;
if D' ni prazen **then** Rekurzija($K \cup \{k\}$, $k+1$, $\mathbf{s}_1, \ldots, \mathbf{s}_{r+1}$, D');

Glavni klic te rekurzivne rešitve bi načeloma bil s praznim naborom kraljic, torej $K = \{\}$ in $k = 1$; tega moramo obravnavati kot rahlo poseben primer, saj dosedanji presek D v tem primeru ni daljica, ampak kar cela šahovnica; izračun D' je v tem primeru trivialen, saj je presek D (= cele šahovnice) in daljice $A_{k,\mathbf{s}_{r+1}}$ kar daljica $A_{k,\mathbf{s}_{r+1}}$ sama.

Vidimo torej, da se rekurzija na vsakem koraku razveji v največ $b := 3^d + 1$ vgnezdenih klicev; tako imamo 1 klic s $k = 1$, pa b klicev s $k = 2$, b^2 klicev s $k = 3$ in tako naprej do b^m klicev s $k = m+1$, ko se rekurzija ustavi. To je skupaj $\sum_{k=0}^{m} b^k$ klicev, kar je naprej enako $(b^{m+1} - 1)/(b-1) \approx b^m \approx 3^{md}$. Pri vsakem klicu imamo $O(d)$ dela; sem lahko štejemo tako izračun $|D|$ (število polj v daljici D) v robnem primeru rekurzije (ko je $k = m+1$) kot delo, ki ga ima pred klicem njegov nadrejeni klic, da izračuna D' (in da v zanki „za vsako $\mathbf{s}_{r+1}$“ popravi vektor $\mathbf{s}_{r+1}$ na naslednjo možno smer). Tako smo prišli do časovne zahtevnosti $O(3^{md} \cdot d)$. Spomnimo se, da je prvotna različica te rešitve imela zahtevnost $O(3^{dm} m^3)$, tako da smo nekaj s to izboljšavo načeloma pridobili (razen če je $d \gg m^3$, torej veliko dimenzij in malo kraljic); upamo lahko tudi, da bi se ocena zahtevnosti naše rekurzivne rešitve izkazala za pesimistično, ker bi se pri mnogih vejah rekurzije izkazalo, da je D' prazna in zato z rekurzijo tam sploh ne bi nadaljevali.

(4) Rešitev z štetjem napadenih polj. To si lahko predstavljamo kot izboljšano različico druge rešitve. Tam smo šli za vsako kraljico po vseh poljih, ki jih napada, in pri vsakem preverjali še, ali ga napada že kakšna od prej obdelanih kraljic. Pri tretji rešitvi pa smo videli, da lahko za vsako smer kar izračunamo, koliko polj napada naša kraljica v tisti smeri. To bi se dalo uporabiti tudi pri drugi rešitvi, vprašanje je le, kako zdaj od teh polj s čim manj truda odšteti tista, ki so jih že napadle prejšnje kraljice. Kakorkoli že, ogrodje rešitve bo zdaj takšno:

$N := 0$;
for $k := 1$ **to** m:
za vsako možno smer napada $\mathbf{s} \in \{-1, 0, 1\}^d$:
t^{max} := število polj, ki jih kraljica k (ki stoji na polju $\mathbf{r}_k$) napada v smeri $\mathbf{s}$
(vse do roba šahovnice) — to je torej dolžina daljice $A_{k\mathbf{s}}$;
$E := \{\}$;
for $k' := 1$ **to** $k - 1$:
if $\mathbf{s} = (0, 0, \ldots, 0)$:
if kraljica k' napada kraljico k **then**
$t^{max} := 0$; **break**;
else if je $\mathbf{r}_k - \mathbf{r}_{k'}$ vzporeden smeri $\mathbf{s}$ **then**

$t^{max} := 0$; **break**;
else dodaj v E vse tiste t, za katere kraljica k' napada polje $\mathbf{r}_k + t\mathbf{s}$; (†)
if $t^{max} > 0$ **then**
$N := N + t^{max} - |E|$;

Najprej torej ugotovimo, koliko polj napada kraljica k (ki stoji na polju $\mathbf{r}_k$) v smeri $\mathbf{s}$ — to je t^{max}, ki smo se ga naučili računati že zgoraj pri tretji rešitvi. Polja, ki jih ta kraljica napada v tej smeri, tvorijo daljico, ki smo ji pri tretji rešitvi rekli $A_{k\mathbf{s}}$; koordinate teh polj so oblike $\mathbf{r}_k + t\mathbf{s}$ za $t = 1, 2, \ldots, t^{max}$. Vprašanje je le, pri katerih t-jih dobimo polja, ki jih napada že kakšna od prej obdelanih kraljic (in teh polj zdaj ne smemo šteti še enkrat). Naš postopek gre v zanki po vseh prejšnjih kraljicah k' in za vsako pogleda, katera polja naše daljice napada že ta kraljica k'; za ta polja dodamo njihove t-je v množico E, tako da bomo na koncu vedeli upoštevati, da smo $|E|$ polj naše daljice šteli že prej in jih zdaj ne smemo (zato N, skupno število napadenih polj, povečamo le za $t^{max} - |E|$). Posebej obravnavamo še primer, ko kraljica k leži v smeri $\mathbf{s}$ ali $-\mathbf{s}$ glede na kraljico k' (torej če je vektor $\mathbf{r}_k - \mathbf{r}_{k'}$ vzporeden vektorju $\mathbf{s}$); tedaj k' napada kar celotno daljico $A_{k\mathbf{s}}$. V tem primeru bi bilo lahko neučinkovito, če bi skušali v E dodati vse t-je (od 1 do t^{max}), zato je enostavneje kar postaviti t^{max} na 0. Še en primer, ki ga obravnavamo posebej, pa je $\mathbf{s} = (0, 0, \ldots, 0)$; tedaj daljica $A_{k\mathbf{s}}$ obsega le polje, na katerem stoji kraljica k, in zanj ni težko preveriti, če ga že napada kakšna prejšnja kraljica k'.

Zdaj moramo torej razmisliti predvsem o tem, kako bi učinkovito izvedli vrstico (†). Recimo, da kraljica k' stoji na polju $\mathbf{r}_{k'} = (x'_1, \ldots, x'_d)$ in da nas zanima, ali ta kraljica napada polje $\mathbf{u} = (u_1, \ldots, u_d)$. Že pri prvi rešitvi te naloge smo videli, da kraljica napada to polje natanko tedaj, ko so si vse razlike $|u_i - x'_i|$ enake po absolutni vrednosti, razen tistih, ki so enake 0. Še drugače lahko to zapišemo takole: kraljica k napada polje $\mathbf{u}$ natanko tedaj, ko je množica razlik $\{u_i - x'_i : i = 1, \ldots, d\}$ podmnožica množice $\{-a, 0, a\}$ za neko celo število $a > 0$. Opazimo lahko tudi, da te lastnosti množica razlik niti ne pridobi niti ne izgubi, če v njej nekatere elemente pomnožimo z -1 (torej če pri nekaterih i vzamemo $x'_i - u_i$ namesto $u_i - x'_i$).

V našem primeru nas zanima, ali kraljica k' napada kakšno od polj $\mathbf{r}_k + t\mathbf{s}$ (oz. pri katerih t se to zgodi). Recimo, da je $\mathbf{r}_k = (x_1, \ldots, x_d)$ in $\mathbf{s} = (\triangle_1, \ldots, \triangle_d)$; naša množica razlik je zdaj oblike $\{x_i - x'_i + t\triangle_i : i = 1, \ldots, d\}$. Na koncu prejšnjega odstavka smo videli, da smemo nekatere razlike pomnožiti z -1; pa naredimo to s tistimi razlikami, pri katerih je $\triangle_i = -1$. Pri teh i torej zdaj dobimo razliko $x'_i - x_i + t$; ostale razlike pa ostanejo take, kot so bile, torej oblike $x_i - x'_i + t$ (če je $\triangle_i = +1$) ali pa $x_i - x'_i$ (če je $\triangle_i = 0$). Naša množica razlik je torej unija dveh podmnožic: $U_t = \{y_i + t : \triangle_i = \pm 1\}$ in $V = \{y_i : \triangle_i = 0\}$, pri čemer je posamezni y_i enak bodisi $x'_i - x_i$ (če je $\triangle_i = -1$) bodisi $x_i - x'_i$ (če je $\triangle_i = 0$ ali 1). Vpeljimo še množico $U = \{y_i : \triangle_i = \pm 1\}$, tako da je $U_t = \{u + t : u \in U\}$.

Zdaj moramo torej preveriti, pri katerih t je množica $U_t \cup V$ oblike $\{-a, 0, a\}$ za nek $a > 0$. (1) Oglejmo si za začetek množico U; če ima ta več kot tri različne elemente, jih bo imela tudi U_t in zato tudi unija $U_t \cup V$ in to ne glede na t; takrat torej kraljica k' ne napada nobenega polja naše daljice (pri nobenem t).

(2) Naslednja možnost je, da ima U natanko tri različne elemente; imenujmo jih (v naraščajočem vrstnem redu) p, q in r. Če naj iz njih (ob nekem primernem t) nastanejo v $U_t \cup V$ vrednosti $-a$, 0 in a, bo moralo očitno veljati $-a = p + t$,

$0 = r+t$ in $a = r+t$; to je mogoče le, če je $r-q = q-p$, za t pa bomo takrat morali vzeti $t = -q$. Tedaj je $U_t = \{-a, 0, a\}$ primerne oblike, nato pa moramo preveriti le še, ali V ne vsebuje kakšnega takega elementa, ki ga v U_t ni.

(3) Kaj pa, če ima U natanko dva različna elementa? Recimo manjšemu p, večjemu pa q. (3.1) Ena možnost je, da v U_t nastane iz p-ja 0, iz q-ja pa a. To se zgodi pri $t = -p$ (a pa je potem enak $q-p$); preveriti pa moramo še, če potem tudi V ne pripelje v unijo nobenega drugega elementa kot 0, a in $-a$. (3.2) Druga možnost je, da v U_t nastane iz p-ja $-a$, iz q-ja pa 0. To se zgodi pri $t = -q$ (a pa je potem enak $q-p$); potem moramo preveriti še V, podobno kot v prejšnjem primeru. (3.3) Tretja možnost pa je, da v U_t nastane iz p-ja $-a$, iz q-ja pa a. To se zgodi pri $t = -(p+q)/2$ (a pa je potem enak $(q-p)/2$). Ta možnost pride v poštev seveda le, če je razlika $q-p$ soda, tako da sta t in a še vedno celi števili. Pri tem t moramo potem preveriti še V, enako kot pri prejšnjih primerih.

(4) Ostane še možnost, da vsebuje U natanko en element — recimo mu p.[26] V tem primeru si pobliže oglejmo množico V. (4.1) Če V vsebuje več kot tri različne elemente, bo to veljalo tudi za unijo $U_t \cup V$ in to za vsak t; torej kraljica k' ne napada nobenega polja naše daljice. (4.2) Če vsebuje V natanko tri različne elemente, moramo preveriti, če je srednji med njimi enak 0, druga dva pa sta si enaka po absolutni vrednosti. V tem primeru recimo največjemu od teh treh elementov a, zdaj pa moramo preveriti le še, pri katerih t tudi U_t ne vsebuje nobene druge vrednosti kot a, 0 ali $-a$. Ker smo videli, da je $U = \{p\}$ in zato $U_t = \{p+t\}$, lahko zaključimo, da so primerni t-ji naslednji trije: $a-p$, $-p$ in $-a-p$. (4.3) Če vsebuje V natanko dva različna elementa, recimo manjšemu od njiju q, večjemu pa r. Tedaj bo imela množica $U_t \cup V$ primerno obliko le v naslednjih primerih: (4.3.1) Lahko je $q = 0$ in $r = a$. Primerni t-ji so spet le tisti, pri katerih iz elementa p (v množici U) nastane (v množici U_t) ena od vrednosti a, $-a$ ali 0; možni t-ji so torej $r-p$, $-p$ in $-r-p$. (4.3.2) Lahko je $q = -a$ in $r = 0$. Podoben razmislek kot v prejšnjem primeru nam pokaže, da so primerni t-ji potem $-q-p$, $-p$ in $q-p$. (4.3.3) Lahko pa je $q = -a$ in $r = a$. To je torej mogoče le, če sta si q in r enaka po absolutni vrednosti. Podoben razmislek kot prej nam pokaže, da so primerni t-ji zdaj $r-p$, $-p$ in $q-p$. (4.4) Če vsebuje V en sam element, recimo mu q, ločimo naslednje možnosti: (4.4.1) Lahko je $q = 0$. Tedaj je primeren poljuben t, saj bo množica $U_t \cup V = \{0, p+t\}$ v vsakem primeru prave oblike. (4.4.2) Lahko je $q > 0$. Če naj bo $U_t \cup V$ zdaj $\subseteq \{-a, 0, a\}$, moramo torej za a vzeti ravno tale q, primerni t-ji pa so zdaj tisti, pri katerih je tudi $p+t$ ena od vrednosti a, 0 ali $-a$; torej so primerni t-ji naslednji: $q-p$, $-p$ in $-q-p$. (4.4.3) Če pa je $q < 0$, bomo morali za a vzeti $-q$, primerni t-ji pa bodo potem isti trije kot v prejšnjem primeru. (4.5) Ostane še možnost, da je V prazna množica. Tedaj je primeren poljuben t, saj bo $U_t \cup V$ vsebovala en sam element (namreč $p+t$).

Zdaj torej znamo v vsakem primeru preveriti, pri katerih t-jih (če sploh katerih) napada kraljica k' kakšna polja z naše opazovane daljice $A_{k\mathbf{s}}$. Pri nekaterih primerih našega razmisleka (točki 4.4.1 in 4.5) se je izkazalo, da k' napada kar célo daljico, vendar se lahko hitro prepričamo, da do teh primerov pride le, če je $\mathbf{r}_k - \mathbf{r}_{k'}$ vzporeden smeri $\mathbf{s}$; in ker naš postopek to možnost preveri prej, še preden sploh pride do vrstice

[26] Množica U gotovo ni prazna, saj bi to pomenilo, da so vsi $\triangle_i = 0$, torej je $\mathbf{s} = (0, 0, \ldots, 0)$, pri takem $\mathbf{s}$ pa se naš postopek sploh ne ukvarja z vrstico (†).

(†), se nam v (†) s točkama 4.4.1 in 4.5 ne bo treba ukvarjati. Predpostavimo torej lahko, da bo (†) dodala največ tri različne t-je v množico E. (Če bi seveda kakšen od teh t-jev ležal zunaj območja $1, \ldots, t^{max}$, ga v E ne bomo dodali, saj tak t predstavlja neko polje, ki v resnici ne leži na naši daljici $A_{k\mathbf{s}}$.)

Razmislimo za konec še o časovni zahtevnosti tako dobljenega postopka. Imamo zanko po kraljicah k (ki naredi m iteracij), znotraj nje zanko po smereh $\mathbf{s}$ (tu je vsakič 3^d iteracij) in znotraj nje po kraljicah k' (tu je $O(m)$ iteracij). Pri vsaki k' imamo $O(d)$ dela: v tem času lahko preverimo, če je $\mathbf{r}_k - \mathbf{r}_{k'}$ vzporeden smeri $\mathbf{s}$; in v tem času lahko tudi sestavimo množici U in V. Tidve množici lahko predstavimo kar s tabelama ali seznamoma, lahko tudi neurejenima; čim pri tvorbi teh množic kakšna od njiju dobi več kot tri različne elemente, lahko nad vsem skupaj obupamo in se posvetimo naslednji kraljici k'; zato lahko predpostavimo, da imata množici ves čas največ štiri elemente in je zato časovna zahtevnost operacij na teh množicah le $O(1)$; tudi postopek ugotavljanja primernih t-jev lahko zato izvedemo v $O(1)$ časa. Videli smo tudi, da pri vsaki k' dodamo v E največ tri različne t-je, torej ima E največ $3m$ elementov; množico E lahko predstavimo kar s tabelo, kjer nove elemente ves čas dodajamo na konec, nazadnje (ko je zanka po k' končana) pa pomečemo iz nje vse duplikate, da ugotovimo, koliko različnih elementov v resnici vsebuje. To lahko naredimo tako, da tabelo uredimo, kar bo torej vzelo $O(m \log m)$ časa. Pri vsaki k in $\mathbf{s}$ imamo torej najprej $O(md)$ dela z zanko po k', nato pa $O(m \log m)$ dela z urejanjem tabele E. Časovna zahtevnost celotnega postopka je tako $O(3^d m^2(d + \log m))$, kar je daleč najboljša rešitev doslej (pri drugi rešitvi smo imeli še faktor n, pri tretji rešitvi pa smo namesto 3^d imeli 3^{dm}).

(5) Rešitev za primer z eno samo kraljico. Recimo, da naša kraljica stoji na polju $\mathbf{x} = (x_1, \ldots, x_d) \in \{1, \ldots, n\}^d$; izberimo si še smer napada $\mathbf{s} = (\triangle_1, \ldots, \triangle_d) \in \{-1, 0, +1\}^d$. Že pri tretji rešitvi smo videli, da so napadena polja oblike $\mathbf{x} + t\mathbf{s}$, pri čemer so dopustni tisti $t \geq 1$, ki ustrezajo neenačbam $1 \leq x_i + t\triangle_i \leq n$ za vsak i (od 1 do n). Naj bo torej t_i največji t, ki ustreza i-ti neenačbi; to je

$$t_i = \begin{cases} n - 1 - x_i, & \text{če je } \triangle_i = +1 \\ x_i - 1, & \text{če je } \triangle_i = -1 \\ \infty, & \text{če je } \triangle_i = 0. \end{cases}$$

Največji t, ki ustreza vsem neenačbam, je potem $t^\star := \min\{t_i : 1 \leq i \leq d\}$; to je število napadenih polj v smeri $\mathbf{s}$.

Minimum, s katerim je definiran $t^\star$, gotovo ni dosežen pri takem i, ki bi imel $t_i = \infty$; to bi bilo možno le, če bi bili vsi $t_i = \infty$, kar se zgodi le pri $\mathbf{s} = (0, 0, \ldots, 0)$, ampak to smer napada lahko obravnavamo posebej. Drugače pa je torej $t^\star$ enak eni od vrednosti $n - 1 - x_i$ ali $x_i - 1$ za nek i. Za $t^\star$ je torej največ $2d$ različnih možnih vrednosti; uredimo jih naraščajoče in jih označimo z $u_1, \ldots, u_p$ (pri čemer je $p \leq 2d$). Možnih smeri je sicer $3^d - 1$, vendar pri vsaki od njih za $t^\star$ dobimo eno od teh p možnih vrednosti. Število različnih smeri, pri katerih je $t^\star = u_j$, označimo z N_j. Skupno število napadenih polj je zdaj preprosto $(\sum_{j=1}^p N_j u_j) + 1$ (enico na koncu smo prišteli zato, da upoštevamo tudi polje, na katerem stoji kraljica sama). Vprašanje je le, kako lahko (za vsak j) učinkovito izračunamo N_j, torej število smeri, pri katerih je napadenih natanko u_j polj.

Za začetek si postavimo malo drugačno vprašanje: koliko je takih smeri, pri katerih je napadenih *vsaj* u polj? V smeri $\mathbf{s}$ je napadenih vsaj u polj natanko tedaj, ko je pri tej smeri vsak t_i večji ali enak u (potem bo tudi $t^\star = \min_i t_i$ večji ali enak u). Naj bo c_i število, ki pove, na koliko načinov si lahko izberemo $\triangle_i$, da bo $t_i \geq u_j$; torej $c_i = |\{\triangle_i : -1 \leq \triangle_i \leq +1,\ t_i \geq u_j\}|$. (Pri tem se spomnimo, da je t_i odvisen od $\triangle_i$ po formuli, ki smo jo videli zgoraj.) Gotovo pride v poštev $\triangle_i = 0$, saj je tam $t_i = \infty$, mogoče pa pride v poštev še eden od $\triangle_i = \pm 1$ ali pa celo oba. Vrednost c_i je torej lahko 1, 2 ali 3. Število smeri $\mathbf{s}$, pri katerih je napadenih vsaj u polj, je torej $(\prod_{i=1}^d c_i) - 1$; na koncu smo odšteli 1 zato, da ne štejemo smeri $\mathbf{s} = (0, 0, \ldots, 0)$.

To znamo izračunati za poljuben u, torej tudi za u_j; naj bo N'_j število smeri, pri katerih je napadenih vsaj u_j polj. Ko računamo število smeri, pri katerih je napadenih *natanko* u_j polj, moramo od N'_j odšteti število smeri, pri katerih je napadenih več kot u_j polj. Naslednje možno število napadenih polj, večje od u_j, pa je u_{j+1}; torej so smeri, pri katerih je napadenih več kot u_j polj, natanko tiste, pri katerih je napadenih vsaj u_{j+1} polj. Tako torej vidimo, da je $N_j = N'_j - N'_{j+1}$. (Pri $j = p$ si mislimo $N'_{p+1} = 0$ in $N_p = N'_p$, saj v nobeni smeri ni napadenih več kot u_p polj.)

Zgoraj smo videli, da število napadenih polj pri posameznem u dobimo v obliki $(\prod_i c_i) - 1$; razmislimo še o tem, kako čim ceneje računati ta produkt. Ker je vsak od faktorjev c_i eno od števil 1, 2 ali 3, je produkt na koncu oblike $2^\alpha 3^\beta$. Stopnji α in β sta seveda pri različnih u-jih različni; tisti stopnji, ki ju dobimo pri $u = u_j$, označimo z α_j in β_j. Ko se premaknemo z enega u-ja na naslednjega — na primer z u_j na u_{j+1} — se nekateri c_i zmanjšajo, nekateri pa ostanejo enaki (nikoli pa se ne povečujejo). V takem primeru moramo stopnjo, ki ustreza stari vrednosti c_i zmanjšati za 1; stopnjo, ki ustreza novi vrednosti c_i, pa moramo povečati za 1. Na primer, če se c_i zmanjša s 3 na 2, moramo zmanjšati β za 1 in povečati α za 1; če pa se c_i zmanjša z 2 na 1, moramo zmanjšati α za 1.

Vprašanje je le še, kako ugotoviti, kateri c_i se pri posameznem koraku povečajo. Ko pripravljamo urejeno zaporedje $u_1, \ldots, u_p$, si lahko pri vsakem u_j pripravimo še seznam indeksov i, pri katerih more t_i imeti vrednost u_j. Ko nas zanima, kako se spremenijo vrednosti c_i, če se premaknemo z u_j na u_{j+1}, moramo le pogledati, pri katerih i je bila u_j ena od možnih vrednosti t_i, in pri koliko možnih vrednostih $\triangle_i$ bi t_i dobil vrednost u_j; za toliko se potem zmanjša c_i. Tako imamo naslednji postopek:

L := seznam, ki za vsak i (od 1 do d) vsebuje para $(n - 1 - x_i, i)$ in $(x_i - 1, i)$;
zdaj je L seznam parov (u, i); uredi jih naraščajoče po u;
če ima več zaporednih členov enak u, jih združi v enega,
  ob vsakem u pa pripravi množico pripadajočih i-jev;
zdaj je L zaporedje parov (u_j, I_j) za $j = 1, \ldots, p$, pri čemer je I_j množica
  indeksov i, pri katerih je nastopila vrednost u_j;

(* *Pri* u_1 *ima vsaka smer* $t^\star \geq u_1$, *zato so vsi* $c_i = 3$. *)
for $i := 1$ **to** d **do** $c_i := 3$;
$\alpha := 0$; $\beta := d$; $N'_1 := 2^\alpha 3^\beta - 1$;

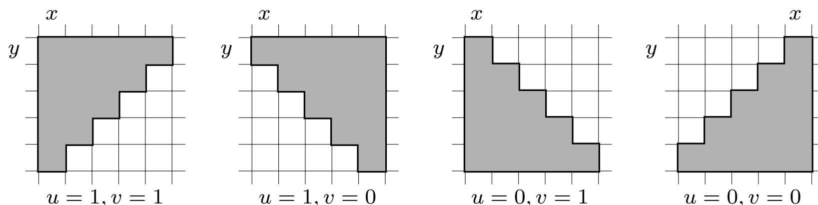
for $j := 1$ **to** $p - 1$:
  za vsak indeks $i \in I_j$:

if $c_i = 2$ **then** $\alpha := \alpha - 1$ **else if** $c_i = 3$ **then** $\beta := \beta - 1$;
$c_i := |\{\triangle_i : -1 \le \triangle_i \le +1,\ t_i(\triangle_i) \ge u_{j+1}\}|$;
if $c_i = 2$ **then** $\alpha := \alpha + 1$ **else if** $c_i = 3$ **then** $\beta := \beta + 1$;
$N'_{j+1} := 2^\alpha 3^\beta - 1$;

Potence števil 2 in 3 (do stopnje d) si lahko potabeliramo vnaprej, tako da zahteva izračun izraza $2^\alpha 3^\beta$ le eno množenje. Časovna zahtevnost našega postopka je zdaj $O(d \log d)$ zaradi urejanja seznama L; vse ostale zanke vzamejo le $O(d)$ časa.[27] To je velika izboljšava v primerjavi s prejšnjimi rešitvami (ki pa so delovale tudi za več kraljic), saj se nam ni treba ukvarjati z vsako od 3^d smeri posebej, tako da smo se znebili eksponentne časovne zahtevnosti v odvisnosti od d.

12. Tangram

Ciljni lik lahko poskusimo sestaviti sistematično s pomočjo rekurzije. V karirasti mreži, ki predstavlja naš ciljni lik, poiščimo najvišje črno polje; če je takih več na isti višini, pa vzemimo najbolj levo med njimi; dobljenemu polju recimo (x, y). Če je naš ciljni lik res mogoče sestaviti iz danih trikotnikov, mora tudi polje (x, y) pripadati enemu od njih; še več, na tem polju mora biti eno od oglišč nekega trikotnika. Za vse možne trikotnike in za vse možne orientacije posameznega trikotnika lahko zdaj poskusimo ta trikotnik postaviti na mrežo tako, da se najbolj levo med njegovimi najvišjimi oglišči znajde ravno na polju (x, y). Možne orientacije so štiri in vsako lahko opišemo s parom (u, v), pri čemer u pove, ali je vodoravna stranica trikotnika zgoraj ali spodaj, v pa pove, ali je navpična stranica trikotnika levo ali desno.



Če kakšen del trikotnika zdaj štrli ven iz ciljnega lika (torej če pokrije kakšno od polj, ki bi v resnici morala ostati bela), potem vemo, da tega trikotnika v tej orientaciji tu ne moremo uporabiti (in lahko nadaljujemo s preizkušanjem drugih trikotnikov in orientacij). Po drugi strani, če je novi trikotnik v celoti pokril ciljni lik, smo nalogo rešili in lahko končamo. Tretja možnost pa je, da ostane kak del ciljnega lika še nepokrit; tedaj lahko nadaljujemo rekurzivno in skušamo s preostalimi trikotniki (tistimi, ki jih še nismo uporabili) po enakem postopku pokriti še preostanek ciljnega lika (to pomeni, da zdaj poiščemo najvišje še nepokrito polje, če pa je takih več, vzamemo najbolj levo med njimi, in tako naprej).

Dobljeni postopek lahko s psevdokodo opišemo takole:

[27] Tu smo predpostavili, da posamezna aritmetična operacija vzame $O(1)$ časa. Če je d tako velik, da ta predpostavka ni več smiselna in se moramo zateči k aritmetiki z velikimi celimi števili, bi morali biti pri izračunu N'_{j+1} malo bolj pazljivi. Najbolje bi bilo vzdrževati vrednost $2^\alpha 3^\beta$ v neki pomožni spremenljivki ν; ko se α (oz. β) poveča za 1, moramo ν pomnožiti z 2 (oz. 3); ko pa se α (oz. β) zmanjša za 1, moramo ν deliti z 2 (oz. 3). Vsaka taka operacija traja $O(d)$ časa, saj ima ν največ $O(d)$ števk. Časovna zahtevnost celotnega postopka je zato $O(d^2)$.

algoritem Tangram(T):
vhod: T — množica še neuporabljenih trikotnikov;
postopek:
1 **if** je ciljni lik že v celoti pokrit **then return true;**
2 naj bo (x, y) najvišje nepokrito polje (če jih je več, vzemimo najbolj levo med njimi);
3 za vsak trikotnik $t \in T$ in za vsako možno orientacijo tega trikotnika:
4 poskusimo postaviti ta trikotnik v tej orientaciji na mrežo tako, da njegovo zgornje levo oglišče pride na (x, y);
5 **if** noben del tega trikotnika ne štrli ven iz ciljnega lika:
6   **if** Tangram($T - \{t\}$) **then return true**;
7 odstranimo trikotnik t z mreže;
8 **return false**;

Naš postopek torej vrne logično vrednost, ki pove, ali je uspel ciljni lik v celoti pokriti ali ne.

Razmislimo še o tem, kako bi ta postopek izvedli v praksi. Predpostavimo, da dobimo opis ciljnega lika kot dvodimenzionalno tabelo (*array*). Prav takšno tabelo lahko uporabimo tudi za označevanje tega, katera polja so že pokrita (z doslej uporabljenimi trikotniki), katera pa ne (v koraku 4 označimo nekaj polj za pokrita, v koraku 7 pa spet za nepokrita). Tudi iskanje najvišjega nepokritega polja v točki 2 je potem enostavno, saj moramo iti le z zanko po vrsticah in v vsaki vrstici še z zanko po poljih te vrstice.

Oglejmo si konkreten primer takšne rešitve v C-ju. Funkcija Tangram dobi kot parameter število trikotnikov, ki so še na voljo, in tabelo z dolžinami njihovih katet; vrne pa logično vrednost (**true** ali **false**), ki pove, ali je uspela z njimi v celoti pokriti vsa še nepokrita polja v mreži (slednjo hranimo v globalni spremenljivki mreza).

```
#include <stdbool.h>
#define w ... /* širina mreže */
#define h ... /* višina mreže */
typedef enum { Belo, Crno, Pokrito } Polje;
Polje mreza[h][w];

bool Tangram(int trikotniki[ ], int nTrikotnikov)
{
  int x, y, i, a, u, v, t, dx, dy, xx, yy; bool ok;

  /* Poiščimo najvišje nepokrito polje. */
  for (y = 0; y < h; y++) {
    for (x = 0; x < w; x++) if (mreza[y][x] == Crno) break;
    if (x < w) break; }
  if (y >= h) return true; /* Če smo pokrili vsa črna polja, končajmo. */

  /* Poskusimo na to polje postaviti oglišče enega od trikotnikov. */
  for (i = 0; i < nTrikotnikov; i++)
  {
    /* Zapomnimo si stranico tega trikotnika in ga pobrišimo iz seznama. */
    a = trikotniki[i]; trikotniki[i] = trikotniki[nTrikotnikov − 1];

    /* Preizkusimo vse možne orientacije tega trikotnika. */
    for (u = 0; u < 2; u++) for (v = 0; v < 2; v++)
    {
      ok = true;
```

```
      for (dy = 0; dy < a && ok; dy++)
        for (dx = 0; dx < (u ? a − dy : dy + 1); dx++) {
          yy = y + dy; xx = x + (v ? dx : (u ? a − 1 : 0) − dx);
          if (xx < 0 || yy < 0 || xx >= w || yy >= h) { ok = false; break; }
          if (mreza[yy][xx] != Crno) { ok = false; break; }}
      if (! ok) continue;
      /* Če smo ta trikotnik uspešno postavili, označimo njegova polja kot pokrita. */
      for (dy = 0; dy < a; dy++) for (dx = 0; dx < (u ? a − dy : dy + 1); dx++) {
        yy = y + dy; xx = x + (v ? dx : (u ? a − 1 : 0) − dx);
        mreza[yy][xx] = Pokrito; }
      /* Z rekurzijo poskusimo pokriti preostanek lika. */
      if (Tangram(trikotniki, nTrikotnikov − 1)) return true;
      /* Označimo polja trikotnika spet kot nepokrita. */
      for (dy = 0; dy < a && ok; dy++)
        for (dx = 0; dx < (u ? a − dy : dy + 1); dx++) {
          yy = y + dy; xx = x + (v ? dx : (u ? a − 1 : 0) − dx);
          mreza[yy][xx] = Crno; }
    }
    /* Vrnimo trenutni trikotnik nazaj v seznam. */
    trikotniki[i] = a;
  }
  return false; /* Če pridemo do sem, vemo, da lika ni mogoče pokriti. */
}
```

Besedilo naloge zagotavlja, da so trikotniki majhni (in da jih je malo), in za take primere je ta rešitev dobra. Razmislimo še o učinkovitejši rešitvi, ki bi delovala hitro tudi za velike trikotnike. Če so trikotniki (in mreža s ciljnim likom) veliki, bo naša dosedanja rešitev porabila precej časa za pregledovanje velikih delov tabele pri (1) iskanju najvišjega nepokritega polja; (2) preverjanju, ali na novo položeni trikotnik pokrije kakšno polje, ki ne pripada ciljnemu liku; (3) označevanju polj za pokrita ali nepokrita (ko položimo nov trikotnik na mrežo ali pa ga odstranimo).

Opis ciljnega lika v vhodnih podatkih si lahko predstavljamo kot dvodimenzionalno tabelo ali matriko, v kateri ima element m_{xy} vrednost 1, če je polje (x, y) črno (torej če pripada ciljnemu liku), in 0, če je belo (torej ne pripada ciljnemu liku). Predpostavimo še, da gredo x-koordinate od 1 do w, y-koordinate pa od 1 do h. Iz polja (x, y) usmerimo navzgor tri poltrake: levo diagonalo $(x - t, y - t)_{t \geq 0}$, navpični poltrak $(x, y - t)_{t \geq 0}$ in desno diagonalo $(x + t, y - t)_{t \geq 0}$. Naj bo L_{xy} množica polj, ki ležijo med levo diagonalo in navpičnim poltrakom, D_{xy} pa množica polj med navpičnim poltrakom in desno diagonalo. (Tadva lika imata obliko enakokrakega pravokotnega trikotnika ali pa pravokotnega trapeza.) Poleg tega naj bo še K_{xy} pravokotnik z zgornjim levim kotom v zgornjem levem kotu mreže, torej $(1, 1)$, in s spodnjim levim kotom v polju (x, y).

Naj bo zdaj k_{xy} število črnih polj v liku K_{xy}, podobno pa l_{xy} v liku L_{xy} in d_{xy} v liku D_{xy}. Ta števila bi lahko zapisali kot vsote:

$$k_{xy} = \sum_{\substack{1 \leq i \leq x \\ 1 \leq j \leq y}} m_{ij}, \quad l_{xy} = \sum_{\substack{1 \leq j \leq y, \\ x+y-j \leq i \leq x, \\ 1 \leq i}} m_{ij}, \qquad d_{xy} = \sum_{\substack{1 \leq j \leq y, \\ x \leq i \leq x-y+j, \\ i \leq w}} m_{ij}.$$

Vsako tako vsoto bi lahko računali z dvema gnezdenima zankama po i in j, vendar bi bilo to precej neučinkovito. Do učinkovitejšega postopka pridemo, če upoštevamo,

kako se ti liki med seboj prekrivajo. Na primer, če pravokotniku K_{xy} pobrišemo spodnjo vrstico, dobimo ravno $K_{x,y-1}$; če mu pobrišemo desni stolpec, dobimo $K_{x-1,y}$. Unija $K_{x,y-1} \cup K_{x-1,y}$ torej obsega vsa polja pravokotnika K_{xy} razen polja (x, y) samega. Presek $K_{x,y-1} \cap K_{x-1,y}$ pa je ravno pravokotnik $K_{x-1,y-1}$. Če hočemo prešteti črna polja v K_{xy}, jih lahko torej za začetek preštejemo v $K_{x-1,y}$ in $K_{x,y-1}$; tista v njunem preseku, to je $K_{x-1,y-1}$, smo zdaj šteli dvakrat, zato jih moramo enkrat odšteti; na koncu pa prištejemo še 1, če je tudi polje (x, y) črno. Tako smo dobili formulo

$$k_{xy} = k_{x-1,y} + k_{x,y-1} - k_{x-1,y-1} + m_{xy}.$$

Podoben razmislek nam pokaže tudi, da je

$$l_{xy} = l_{x-1,y-1} + l_{x,y-1} - l_{x-1,y-2} + m_{xy}$$

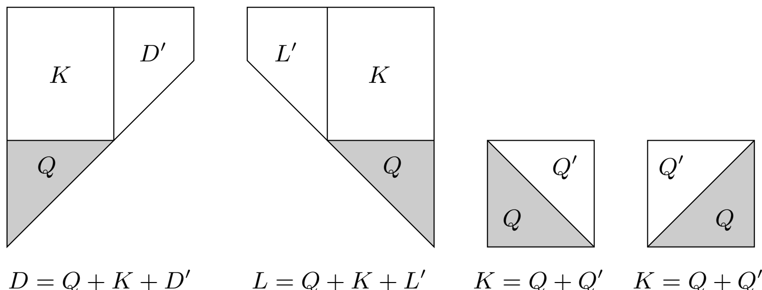
in

$$d_{xy} = d_{x,y-1} + d_{x+1,y-1} - d_{x+1,y-2} + m_{xy}.$$

Če gremo sistematično po naraščajočih x in y, lahko vsako novo vrednost k_{xy}, l_{xy} in d_{xy} izračunamo v $O(1)$ časa, tako da za izračun vseh k_{xy}, l_{xy} in d_{xy} skupaj porabimo le $O(wh)$ časa.

S pomočjo vsot k_{xy} lahko za poljuben pravokotnik (ne le za takega, ki ima zgornji levi kot v $(1, 1)$) hitro ugotovimo, koliko črnih polj vsebuje. Recimo na primer, da nas zanima pravokotnik Q z zgornjim levim kotom v $(x' + 1, y' + 1)$ in spodnjim desnim kotom (x, y). Začnimo s K_{xy}; od njega odštejmo vse, kar leži levo od Q: to je $K_{x',y}$; in vse, kar leži nad Q: to je $K_{x,y'}$; tisto, kar leži hkrati levo od Q in nad njim, smo zdaj odšteli dvakrat, torej prištejmo to območje enkrat nazaj: to je $K_{x'y'}$. Tako torej vidimo, da je število črnih polj v pravokotniku Q enako $k_{xy} - k_{x',y} - k_{x,y'} + k_{x'y'}$.

Če bi namesto pravokotnika za Q vzeli enakokrak pravokotni trikotnik v poljubni izmed štirih orientacij, ki smo jih videli na začetku naše rešitve, bi lahko s podobnim razmislekom tudi zanj prešteli, koliko črnih polj vsebuje. Če ima trikotnik Q vodoravno stranico na vrhu, ga lahko obdelamo tako, da od enega od L- ali D-likov odštejemo nek manjši lik istega tipa (na spodnji sliki je označen z L' ali D') in nek pravokotnik K; če pa ima Q vodoravno stranico na dnu, ga lahko obdelamo tako, da od nekega pravokotnika K odštejemo trikotnik Q', ki ima vodoravno stranico na vrhu.



Zdaj torej znamo za enakokrak pravokotni trikotnik v poljubni od štirih možnih orientacij izračunati (in to v samo $O(1)$ časa), koliko črnih polj vsebuje. Ker tudi znamo izračunati njegovo ploščino (če ima kateto a, je njegova ploščina $a(a + 1)/2$

polj), torej ni težko preveriti, če so vsa njegova polja res črna. Če niso, to pomeni, da takega trikotnika na mrežo ne smemo postaviti, ker bi nek njegov del štrlel zunaj ciljnega lika.

Dosedanji razmislek je temeljil na ideji, da se mreža ne bo spreminjala — zato je dovolj, če tabele k_{xy}, l_{xy} in d_{xy} izračunamo le na začetku, preden začnemo razporejati trikotnike z rekurzijo. Torej tabele z mrežo ne smemo uporabljati za označevanje pokritih delov mreže, kot smo jo v naši prvotni rešitvi. Označevanje pokritih polj v mreži je imelo tam dva namena: za preverjanje, ali bi se novi trikotnik prekrival s kakšnim od dosedanjih; in za iskanje najvišjega še nepokritega polja.

Preverjanje, ali bi se novi trikotnik prekrival s kakšnim od dosedanjih trikotnikov, lahko izpeljemo tako, da se v zanki sprehodimo po vseh dosedanjih trikotnikih in za vsakega od njih preverimo, ali se kaj prekriva z našim novim trikotnikom. Takšno preverjanje, ali se dva trikotnika prekrivata, lahko izvedemo v $O(1)$ časa z nekaj geometrijskega razmišljanja: preveriti moramo, ali en trikotnik leži v drugem (na primer tako, da to preverimo za njegova oglišča) in ali se seka (ali dotika) kakšen par njunih stranic.

Ostane še vprašanje, kako poiskati najvišje še nepokrito polje v mreži. Vprašajmo se za začetek malo drugače: recimo, da si izberemo neko pravokotno območje Q in nas zanima, koliko je na njem nepokritih polj. Število nepokritih polj je razlika med številom črnih polj in številom pokritih polj; število črnih polj na pravokotniku Q smo se naučili računati že zgoraj. Število pokritih polj pa lahko računamo spet z geometrijskim razmislekom: za vsakega od doslej postavljenih pravokotnikov moramo pravzaprav izračunati ploščino njegovega preseka s pravokotnikom Q, kar lahko naredimo v $O(1)$ časa za vsak trikotnik. Najvišje nepokrito polje lahko zdaj poiščemo z bisekcijo:

Q := pravokotnik, ki pokriva celo mrežo;
if Q ne vsebuje nobenega nepokritega polja **then**
  return (* *ciljni lik je v celoti pokrit in lahko končamo* *)
while je Q visok več kot eno vrstico:
  Q' := zgornja polovica pravokotnika Q;
  if vsebuje Q' kakšno nepokrito polje **then** $Q := Q'$
  else Q := spodnja polovica pravokotnika Q;
while je Q širok več kot en stolpec:
  Q' := leva polovica pravokotnika Q;
  if vsebuje Q' kakšno nepokrito polje **then** $Q := Q'$
  else Q := desna polovica pravokotnika Q;

Na koncu tega postopka je pravokotnik Q velik eno samo polje in to je ravno najvišje nepokrito polje na mreži (če je tam v isti vrstici več enako visokih, je Q najbolj levo med njimi). Izvesti moramo torej $O(\log wh)$ preverjanj, ali nek pravokotnik vsebuje kakšno nepokrito polje, za vsako takšno preverjanje pa porabimo $O(r)$ časa, če je n število trikotnikov, ki smo jih doslej postavili v mrežo.

Kakšna je v najslabšem primeru časovna zahtevnost tako dobljene rešitve? Recimo, da imamo n trikotnikov in da sestavljamo ciljni lik na mreži velikosti $w \times h$. Koliko je takih rekurzivnih klicev, pri katerih je že razporejenih r od teh n trikotnikov? Te trikotnike si lahko izberemo na $n(n-1)\cdots(n-r+1)$ načinov, pri vsakem pa imamo 4 možne orientacije, tako da je tu $O(n!/(n-r)! \cdot 4^n)$ klicev. Pri vsakem

klicu imamo $O(r \log wh)$ dela, da poiščemo najvišje nepokrito polje, pa še $O(r)$ dela pri nadrejenem klicu, s katerim smo preverili, da se zadnji postavljeni trikotnik ne prekriva z dosedanjimi. Če vse to seštejemo in dodamo še $O(wh)$ za izračun tabel k_{xy}, l_{xy} in d_{xy} na začetku, vidimo, da je časovna zahtevnost v najslabšem primeru $O(wh + 4^n\,(n+1)!\,\log wh)$. V praksi bo verjetno še občutno manjša, saj se mnogi razporedi trikotnikov že zgodaj izkažejo za neobetavne (ker trikotnik štrli ven iz ciljnega lika in podobno).

13. Taksist

Oštevilčimo avtomobile od 1 do m; za avtomobil a poznamo torej leto izida l_a, takrat ga lahko kupimo za ceno c_a, po i letih (za $i = 1, \ldots, 5$) pa ga lahko prodamo za znesek s_{ia}.

Nalogo lahko rešimo z dinamičnim programiranjem. Zastavimo si podproblem: recimo, da smo na začetku leta t in potrebujemo nov avtomobil; naj bo $f(t)$ najmanjša možna skupna poraba denarja od tega trenutka do konca n-tega leta. Če bi radi izračunali $f(t)$, se moramo med drugim odločiti, kateri avtomobil (izmed tistih, ki izidejo v letu t) bi kupili in po koliko letih bi ga prodali; če se na primer odločimo za avtomobil a in ga prodamo po i letih, se naša poraba denarja poveča za $c_a - s_{ia}$ (avtomobil kupimo za c_a in ga kasneje prodamo za s_{ia}), nato pa se znajdemo na začetku leta $t + i$ spet brez avtomobila. Od tu naprej imamo torej problem enake oblike kot na začetku, le da smo v letu $t+i$ namesto t; najmanjša možna poraba od tu naprej je torej $f(t+i)$.

Med vsemi možnimi a in i moramo seveda izbrati tista, ki pripeljeta do najmanjše skupne porabe. Tako smo dobili rekurzivno zvezo

$$f(t) = \min_{a,i}\{c_a - s_{ia} + f(t+i) \ : \ l_a = t,\ 1 \le i \le 5,\ t+i \le n+1\}.$$

Robni primer te rekurzije pa je $f(n+1) = 0$: ko preteče n let in smo na začetku leta $n+1$, gremo domov in avtomobilov ne bomo več niti kupovali niti prodajali. Odgovor, po katerem sprašuje naloga, pa je $f(1)$, torej najmanjša možna poraba od začetka prvega leta do konca opazovanega obdobja.

Za učinkovito računanje funkcije f je koristno, če jo računamo po padajočih t in si že izračunane rezultate shranjujemo v tabelo. Koristno je tudi, če si za vsako leto t pripravimo seznam avtomobilov a, ki izidejo tisto leto (torej ki imajo $l_a = t$). To lahko naredimo z enim prehodom čez vse avtomobile, tako da nam ne bo treba iti pri vsakem t po vseh avtomobilih. Zapišimo dobljeni postopek še s psevdokodo:

```
for t := 1 to n do A[t] := prazen seznam;
for a := 1 to m do dodaj a v seznam A[l_a];

f[n + 1] := 0;
for t := n downto 1:
   f[t] := ∞;
   za vsak a iz seznama A[t]:
      for i := 1 to min{5, n + 1 − t}:
         x := c_a − s_ia + f[t + i];
         if x < f[t] then f[t] := x;
return f[1];
```

14. VIP

(a) Pri lažji različici naloge nimamo nobene prave izbire: za vsak i moramo na stol i posaditi člana a_i, razen če ga sploh ni na sestanek, v tem primeru pa mora na ta stol sesti b_i. Če se v tako dobljenem razporedu zgodi, da bi moral kak človek sedeti na dveh ali več stolih hkrati, potem primernega razporeda, ki bi ustrezal zahtevam naloge, sploh ni. Zapišimo ta postopek s psevdokodo:

```
    algoritem NajdiRazpored:
    vhod: par (a_i, b_i) za i = 1, ..., n; in množica Odsotni
1   ŽeUporabljen := {};
2   for i := 1 to n:
3     if a_i ∈ Odsotni then r_i := a_i else r_i := b_i;
4     if r_i ∈ ŽeUporabljen then
5       (* primernega razporeda sploh ni *) return;
6   izpiši razpored (r_1, ..., r_n);
```

V praksi bi lahko množici *Odsotni* in *ŽeUporabljen* predstavili s tabelama logičnih vrednosti (tip **bool** ali **boolean**; to je primerno, če so osebe predstavljene s celimi števili od 1 naprej) ali pa z razpršenima tabelama (*hash table*; to je koristno, če so osebe predstavljene z nizi, npr. z imenom in priimkom).

(b) Pri tej različici naloge smemo na stol i posaditi člana b_i tudi v primeru, ko je a_i prisoten na sestanku. Zato vrstni red članov (a_i kot prvi in b_i kot rezerva) ni več pomemben; za vsak stol i si lahko predstavljamo množico kandidatov, ki smejo sedeti na njem. Problem lahko zdaj predstavimo z dvodelnim grafom: imejmo množico n točk, ki predstavljajo stole, in množico točk, ki predstavljajo ljudi; in med njimi imejmo povezave oblike (i, a_i) in (i, b_i), ki torej povedo, kateri ljudje smejo sedeti na katerih stolih. Točke, ki predstavljajo ljudi, ki se sestanka ne bodo udeležili, pobrišimo (skupaj s povezavami, incidenčnimi nanje).

Naloga od nas zahteva, da za vsak stol izberemo po enega človeka (ki sme sedeti na tem stolu) in to tako, da noben človek ni izbran pri več kot enem stolu. Takemu razporedu ustreza v našem grafu neka podmnožica povezav, ki ima zanimivo lastnost: nobena točka ni krajišče dveh ali več izbranih povezav (kajti za noben stol nismo izbrali po več kot enega človeka in nobenega človeka nismo izbrali pri več kot enem stolu). Taki množici povezav se v teoriji grafov reče *ujemanje* (*matching*). Opazimo lahko, da velja tudi obratno: vsako ujemanje v našem grafu predstavlja nek veljaven razpored ljudi na stole (v katerem pa mogoče nekateri stoli ostanejo nezasedeni). Nalogo torej lahko rešimo tako, da v našem grafu poiščemo največje možno ujemanje; če ima to ujemanje n povezav, so zasedeni vsi stoli in smo našli razpored, kakršnega zahteva naloga; če pa ima tudi največje ujemanje manj kot n povezav, potem vemo, da primernega razporeda sploh ni.

Za iskanje največjega ujemanja v dvodelnih grafih (*bipartite matching*) obstajajo razni algoritmi s polinomsko časovno zahtevnostjo, na primer algoritem dopolnilnih poti (*augmenting paths algorithm*) in Hopcroft-Karpov algoritem.[28]

[28] Za več o tem gl. npr. Wikipedijo *s. v.* Bipartite matching. Problem maksimalnega ujemanja lahko prevedemo tudi na problem maksimalnega pretoka v grafu, s katerim smo se srečali na primer pri nalogi 2011.X.12 (gl. str. 110 v biltenu 2013).

15. Prehod

Ker igra poteka v diskretnih časovnih korakih, si lahko reševanje predstavljamo kot pregledovanje prostora stanj. Opazimo lahko, da se stanje vseh plošč (njihova višina in smer gibanja) ponavlja na vsakih $4a$ časovnih korakov (ker se v toliko časa plošča premakne od najnižje do najvišje točke in nazaj). Stanje plošč lahko torej opišemo kar s celim številom u od 0 do $4a-1$, ki nam pove, koliko časa po začetku igre so se plošče prvič znašle v tem stanju (v enakem stanju se bodo potem znašle tudi ob času $u+4a$, $u+8a$ in tako naprej). Markov položaj pa lahko opišemo s celim številom x od 0 do $n+1$ (pri čemer 0 pomeni levi breg, $n+1$ desni breg, števila od 1 do n pa plošče).

Prostor stanj lahko pregledujemo v širino in si pri tem za vsako stanje (x,u) v neki tabeli tudi zapišemo, kateri je najzgodnejši čas, ob katerem je bilo dosegljivo; recimo temu času $T[x,u]$. Med delom hranimo v vrsti Q stanja, ki smo jih že odkrili, ne pa tudi pregledali, kaj je dosegljivo iz njih. Ko neko stanje prvič zagledamo (to prepoznamo po tem, da je v T zanj še čas ∞), vpišemo v T čas, ob katerem smo ga dosegli, in ga dodamo v vrsto Q.

for $x := 0$ **n+1 do for** $u := 0$ **to** $4a-1$ **do** $T[x,u] := \infty$;
$Q :=$ prazna vrsta;
$T[0,0] := 0$; dodaj $(0,0)$ v vrsto Q;
while Q ni prazna:
  vzemi stanje (x,u) z začetka vrste Q;
  za vsako stanje (x',u'), ki je dosegljivo v enem koraku iz (x,u):
    if $T[x',u'] = \infty$ **then**
      $T[x',u'] := T[x,u]+1$; dodaj (x',u') na konec vrste Q;

Postopek se ustavi, ko pregleda vsa dosegljiva stanja; ker pregleduje stanja po naraščajočem času, bi ga lahko pravzaprav ustavili že prej, čim doseže kakšno stanje z $x=n+1$. Če pa nobeno od stanj z $x=n+1$ ni dosegljivo (torej imajo v tabeli T še vrednost ∞), potem vemo, da Marko sploh ne more doseči desnega brega; takrat lahko s pregledom tabele T ugotovimo, vsaj to, kateri je največji x, pri katerem je kakšno stanje vendarle dosegljivo.

Razmislimo še o tem, katera stanja (x',u') so dosegljiva v enem koraku iz u. Za u' je možna le vrednost $(u+1) \bmod 4a$ — stanje plošč se v enem koraku načeloma premakne iz u v $u+1$, upoštevati moramo le to, da po $4a$ korakih pademo nazaj v začetno stanje. Za x' pa pridejo v poštev vrednosti x (ostanemo na plošči, kjer smo bili), $x-1$ (skočimo s trenutne plošče na levo) in $x+1$ (skočimo s trenutne plošče na desno); pri tem moramo upoštevati omejitve, kot so, da z $x=0$ ne moremo v levo, z $x=n+1$ ne moremo desno; skok na sosednjo ploščo je mogoč le, če ta leži nižje od naše dosedanje; poleg tega je stanje nedosegljivo, če plošča x' ob času u' leži pod gladino vode.

Višine plošče v poljubnem času ni težko računati: če je ob času 0 bila na višini v in se je premikala navzgor, bi bila ob času u načeloma na višini $v+u$; ob času $u=a-v$ doseže višino a in se začne premikati navzdol, tako da je odtlej ob času u na višini $2a-u-v$; ob času $u=3a-v$ doseže višino $-a$ in se začne spet premikati navzdol, tako da je odtlej ob času u na višini $u+v-4a$. Pri ploščah, ki so se ob času 0 premikale navzdol, je razmislek podoben.

16. Vodovod

Vpeljimo nekaj oznak za vhodne podatke, ki jih dobimo pri tej nalogi: za vsak u naj bo p_u vozlišče, iz katerega priteka voda v vozlišče u; c_u naj bo kapaciteta povezave $p_u \to u$; in r_u naj bo količina vode, ki jo porabi neposredno vozlišče u.

Povezava $p_u \to u$ mora zadoščati ne le za potrebe vozlišča u, ampak tudi za potrebe vseh vozlišč v u-jevem poddrevesu, torej tistih vozlišč, v katera se iz korena pride skozi u. Vsoto potreb vseh teh vozlišč označimo z R_u. Razširiti bo treba torej vse tiste povezave, pri katerih je $R_u > c_u$; vprašanje je le še to, kako izračunati R_u za vsak u.

Recimo, da gredo iz u neposredne povezave v vozlišča $v_1, \ldots, v_k$. Torej je u-jevo poddrevo pravzaprav unija vozlišča u poddreves vozlišč $v_1, \ldots, v_k$; zato je $R_u = r_u + R_{v_1} + \ldots + R_{v_k}$. Drevo lahko torej pregledujemo rekurzivno: ko se ukvarjamo z vozliščem u in računamo R_u, bomo z rekurzivnimi klici obdelali vse njegove otroke $v_1, \ldots, v_k$ in izračunali njihove R_{v_i}. Tako smo dobili naslednji postopek:

algoritem ObdelajPoddrevo(vozlišče u):
  $R_u := r_u$; *ŠtRazširitev* $:= 0$;
  za vsako povezavo $u \to v$:
    ŠtRazširitev := *ŠtRazširitev* + ObdelajPoddrevo(v);
    $R_u := R_u + R_v$;
  if $R_u > c_u$ **then**
    ŠtRazširitev := *ŠtRazširitev* $+ 1$;
  return *ŠtRazširitev*;

Naš postopek torej izračuna R_u in vrne število povezav, ki jih je treba razširiti v u-jevem poddrevesu (vključno s povezavo $p_u \to u$, če je potrebna razširitve). Pri tem izvede za vsakega u-jevega otroka v rekurzivni klic, ki izračuna R_v in vrne število potrebnih razširitev v v-jevem poddrevesu. Glavni del programa bi moral klicati ObdelajPoddrevo(1), da se obdela celotno drevo (vozlišče 1 je koren drevesa).

Razmisliti moramo še o tem, kako za dano vozlišče u našteti vse povezave, ki izhajajo iz njega. V vhodnih podatkih imamo za vsako vozlišče u navedenega njegovega starša v drevesu, p_u; ta podatek nam pove, da iz p_u izhaja povezava v u. Lahko si torej z enim prehodom po vhodnih podatkih pripravimo za vsako vozlišče seznam njegovih otrok (torej vozlišč, v katera kažejo povezave iz njega):

  for $u := 1$ **to** n **do**
    Otroci[u] := prazen seznam;
  for $u := 2$ **to** n **do**
    dodaj u v seznam *Otroci*[p_u];

Druga zanka se začne pri $u = 2$, ker je $u = 1$ koren drevesa in torej nima predhodnika p_u. V praksi lahko takšne sezname poceni in učinkovito predstavimo z dvema tabelama: $F[u]$ naj bo prvi u-jev otrok, $S[u]$ pa naj bo naslednji otrok u-jevega starša p_u (torej naslednji za u v seznamu p_u-jevih otrok). Tako dobimo:

  for $u := 1$ **to** n **do**
    $F[u] := -1$, $S[u] := -1$;
  for $u := 2$ **to** n **do**
    $S[u] := F[p_u]$, $F[p_u] := u$;

Ko se mora podprogram ObdelajPoddrevo sprehoditi po vseh u-jevih otrocih, lahko to naredi takole:

$v := F[u]$; **while** $v \neq -1$:
  obdelaj v z rekurzivnim klicem itd.;
  $v := S[v]$;

Časovna zahtevnost tega postopka je le $O(n)$: toliko časa porabimo za pripravo seznamov otrok, potem pa imamo za vsako vozlišče po en rekurzivni klic, ki (če odštejemo čas v vgnezdenih rekurzivnih klicih) porabi $O(1)$ časa, kar je skupaj spet $O(n)$.

17. Podobna števila

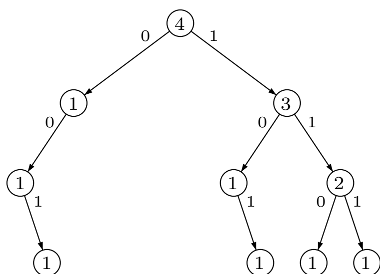
Oglejmo si pobliže, kaj se zgodi, ko z operacijo xor merimo podobnost med dvema številoma. Recimo, da so naša števila največ t-bitna; bit i (za $i = 0, \ldots, t-1$) števila a označimo z a_i. Tako je torej $a = \sum_{i=0}^{t-1} a_i 2^i$. Podobno naredimo še za b in razmislimo, kaj se zgodi pri izračunu $c = a$ xor b. V c so prižgani biti natanko na tistih mestih, kjer je $a_i \neq b_i$, drugod pa so ugasnjeni. Torej je $c = \sum_i 2^i$, pri čemer gre vsota le po tistih i, pri katerih je $a_i \neq b_i$.

Za potence števila 2 med drugim velja zanimiva lastnost $2^0 + 2^1 + \ldots + 2^{i-1} = 2^i - 1$. Z drugimi besedami, 2^i je večji (za 1 večji) od vseh nižjih potenc števila 2 skupaj. To pa pomeni, da če se a in b razlikujeta v bitu i, to prispeva k naši meri različnosti med številoma več kot vse morebitne razlike na nižjih bitih. Če primerjamo a z več števili $b, b', \ldots$, ki se med seboj ujemajo v bitih od $i+1$ naprej, potem so med temi števili tista, ki se z a ujemajo v bitu i, vsekakor bolj podobna a-ju kot tista, ki se od njega razlikujejo v bitu i — ne glede na to, kaj se dogaja na nižjih bitih.

Uporabimo zdaj ta razmislek pri naši nalogi. Recimo, da smo dobili za poizvedbo število q in iščemo k-to njemu najpodobnejše število v naši množici $x_1, \ldots, x_n$. Za začetek torej vemo, da so števila, ki se s q-jem ujemajo v najvišjem bitu (bitu $t-1$) vsekakor bližje q-ju kot tista, ki se od njega v tem bitu razlikujejo. Recimo, da se m števil s q-jem ujema v bitu $t-1$, ostalih $n-m$ pa se od njega razlikuje. Če je torej $k \leq m$, lahko zaključimo, da je k-to najbližje število q-ju v prvi skupini (med tistimi, ki se s q-jem ujemajo v bitu $t-1$) in lahko iskanje nadaljujemo tam; sicer pa vemo, da moramo pravzaprav iskati $(k-m)$-to najbližje število q-ju v drugi skupini (torej med tistimi, ki se od q-ja razlikujejo v bitu $t-1$).

Zdaj smo se torej omejili na neko manjšo podmnožico naše prvotne množice $\{x_1, \ldots, x_n\}$; ta podmnožica obsega vsa tista števila, ki imajo neko določeno vrednost bita $t-1$. V nadaljevanju bi zdaj naš razmislek ponovili na tej podmnožici in gledali bit $t-2$. Tako sčasoma (v največ $O(t)$ korakih) pridemo do enega čisto konkretnega števila, ki je odgovor na našo poizvedbo.

Za učinkovito izvajanje tega postopka je koristno zložiti števila $x_1, \ldots, x_n$ v neke vrste črkovno drevo (*trie*; glej sliko na str. 119); iz posameznega vozlišča drevesa gresta po dve povezavi z oznakama 0 in 1; vozlišče, ki predstavlja število x_i, dosežemo tako, da začnemo v korenu in po vrsti sledimo povezavam, ki ustrezajo bitom števila x_i (od višjih proti nižjim). V vsakem vozlišču vzdržujemo tudi števec, ki nam



Primer drevesa za množico štirih 3-bitnih števil: $\{1, 5, 6, 7\}$ oz. dvojiško $\{001, 101, 110, 111\}$. V vsakem vozlišču je napisana vrednost njegovega števca.

---

pove, koliko elementov naše množice $\{x_1, \ldots, x_n\}$ leži v poddrevesu, ki se začne pri tem vozlišču.

V spodnji psevdokodi si predstavljamo, da je vsako vozlišče struktura s polji *števec* (na začetku 0) ter *otrok*[0] in *otrok*[1] (kazalca na poddrevesi, na začetku NIL). Postopek za gradnjo drevesa je torej takšen:

algoritem DODAJ(število x):
  v := koren drevesa; povečaj v.*števec* za 1;
  for $i := t - 1$ **downto** 0:
    b := i-ti bit števila x;
    če v še nima otroka z oznako b, ustvarimo zanj novo vozlišče;
    v := v.*otrok*[b]; (* *premaknimo se v to poddrevo* *)
    povečaj v.*števec* za 1;

Drevo zgradimo tako, da na začetku ustvarimo novo prazno vozlišče za koren drevesa in nato po vrsti pokličemo DODAJ(x_i) za vsa števila iz naše množice.

Zapišimo s pomočjo te strukture še prej opisani postopek za odgovarjanje na poizvedbe:

algoritem POIZVEDBA(q, k):
  v := koren drevesa; $x := 0$;
  for $i := t - 1$ **downto** 0:
    b := i-ti bit števila q;
    if v.*otrok*[b] = NIL **then** $m := 0$ **else** $m := v$.*otrok*[b].*števec*;
    if $k > m$ **then** $b := 1 - b$, $k := k - m$;
    $x := x + b \cdot 2^i$;
    v := v.*otrok*[b];
  return x;

Na vsakem koraku torej pogledamo, koliko elementov naše množice je v tistem poddrevesu, ki se s q-jem ujema v trenutnem (i-tem) bitu; recimo, da jih je m; če je to manj kot k, potem vemo, da moramo iskati $(k - m)$-ti element v drugem poddrevesu. Ko se spuščamo dol po drevesu, v spremenljivki x sproti postavljamo bite na vrednosti, ki ustrezajo oznakam na povezavah, po katerih smo se spustili; tako bo x na koncu, ko pridemo do lista, vseboval ravno tisto število, ki ga predstavlja ta list (in ki pomeni pravilni odgovor na našo poizvedbo).

18. Žetoni

(a) Belih žetonov ne moremo premikati na polja, kjer stojijo črni žetoni; črnih žetonov pa sploh ne moremo premikati. To pomeni, da so za naše namene tista polja, na katerih stojijo črni žetoni, povsem neprehodna. Ostala polja pa si lahko predstavljamo kot točke grafa, pri čemer med dvema točkama obstaja povezava, če imata tisti dve točki skupno vsaj eno oglišče. V tako dobljenem grafu poiščimo povezane komponente, torej maksimalne skupine točk, za katere velja, da je vsaka točka dosegljiva iz vsake druge. Z vidika naše mreže vsaka taka povezana komponenta pomeni skupino polj, po katerih se lahko z belim žetonom poljubno premikamo; tako lahko med drugim vse bele žetone s takšne skupine polj spravimo na en velik kup na enem od teh polj. Najmanjše možno število zasedenih polj torej dobimo tako, da preštejemo polja s črnimi žetoni (na te ne moremo vplivati) in jim dodamo število tistih povezanih komponent, ki vsebujejo vsaj po en bel žeton (na vsaki taki komponenti mora biti na koncu zasedeno eno polje, na katerem je kup z vsemi belimi žetoni, ki so prvotno stali na poljih te komponente).

Povezane komponente lahko poiščemo v času $O(w \cdot h)$ preprosto tako, da začnemo v poljubnem polju in preiskujemo graf v širino, pri tem si označujemo, katera polja smo že zagledali (tabela v), in postopek ponavljamo, dokler ni pregledana cela mreža:

```
z := 0; (* Število zasedenih polj. *)
for y := 1 to h do for x := 1 to w do v[x, y] := false;
for y := 1 to h do for x := 1 to w do
  if je na (x, y) črn žeton then z := z + 1; continue;
  if v[x, y] then continue;
  (* Z iskanjem v širino preglejmo naslednjo povezano komponento. *)
  Q := prazna vrsta; dodaj (x, y) v vrsto Q;
  b := false; (* Ali je na trenutni komponenti kak bel žeton? *)
  while Q ni prazna:
    vzemi poljubno polje (x, y) iz vrste Q;
    if je na (x, y) bel žeton then b := true;
    za vsakega soseda (x', y') polja (x, y):
      if v[x', y'] ali pa je na (x', y') črn žeton then continue;
      v[x', y'] := true; dodaj (x', y') v vrsto Q;
  if b then z := z + 1;
```

Na koncu tega postopka imamo v z najmanjše možno število zasedenih polj, ki ga lahko dosežemo.

(b) Začnimo pri prvi vrstici mreže; poiščimo najbolj levi žeton in ge premaknimo v levo do $x = 1$. Nato vzemimo drugi najbolj levi žeton in ga premaknimo v levo, kolikor daleč je mogoče: če je enake barve kot prejšnji žeton, se mu lahko pridruži na istem polju, drugače pa naj se ustavi eno polje desno od njega. Tako nadaljujemo z vsemi žetoni trenutne vrstice. Enak postopek nato ponovimo še z vsemi ostalimi vrsticami.

Zdaj imamo v vsaki vrstici vse žetone poravnane na levem robu vrstice, poleg tega pa tvorijo njihove barve izmeničen vzorec: za belim žetonom pride črni in obratno. Če ima mreža eno samo vrstico ($h = 1$), kaj več od tega že ne moremo doseči, saj nobeden od belih žetonov ne more preskočiti svojih črnih sosedov, da bi

se združil s kakšnim drugim belim žetonom in obratno. Podoben razmislek lahko uporabimo tudi, če ima mreža en sam stolpec (torej $w = 1$), le da moramo tam pač premikati žetone navzgor namesto na levo.

V nadaljevanju torej prepostavimo, da ima mreža vsaj dve vrstici in vsaj dva stolpca. Opazujmo za začetek prvi dve vrstici. Če je ena od teh dveh vrstic prazna, lahko vse bele žetone premaknemo iz neprazne vrstice v prazno; zdaj imamo eno vrstico samih belih žetonov in eno samih črnih, torej lahko s premiki v levo naredimo dva kupa žetonov (oz. največ dva, saj ni nujno, da so prisotni žetoni obeh barv), vsakega v svoji vrstici na $x = 1$. Tistega iz gornje vrstice nato premaknemo na $x = 2$ v spodnji vrstici; to bo prišlo prav za kasneje.

Oglejmo si zdaj primer, ko sta prvi dve vrstici obe neprazni. Če ena od teh dveh vrstic vsebuje vsaj dva žetona, lahko razmišljamo takole: oglejmo si najbolj leva dva žetona te vrstice; gotovo sta različnih barv, torej je eden od njiju črn, drugi pa bel. Obe tidve polji imata skupno vsaj po eno oglišče s prvim (najbolj levim) poljem druge vrstice; torej bomo enega od teh dveh žetonov gotovo lahko premaknili na prvo polje druge vrstice, ne glede na to, kakšne barve žeton je tam. S tem smo število zasedenih polj zmanjšali za ena; tako nadaljujemo, dokler še ima vsaj ena od opazovanih dveh vrstic več kot en žeton. Tako torej sčasoma pridemo v stanje, ko imata obe po največ eno zasedeno polje; če so na obeh žetoni iste barve, ju lahko združimo v en kup, recimo v spodnji vrstici; drugače pa kup iz zgornje vrstice premaknemo v spodnjo.

V vsakem primeru smo torej prvi dve vrstici predelali tako, da je prva zdaj popolnoma prazna, v drugi pa sta zasedeni največ dve polji (in to s kupoma različnih barv). Enak razmislek lahko zdaj ponovimo za drugo in tretjo vrstico, nato za tretjo in četrto in tako naprej. Sčasoma torej spravimo vse bele žetone na en sam velik kup (v zadnji vrstici), prav tako tudi vse črne. Končno število zasedenih polj je tako kar enako številu različnih barv žetonov, ki so bili prisotni v začetnem stanju mreže (torej je lahko 0, 1 ali 2).

(c) Iz besedila naloge sledi, da je vseeno, v kakšnem vrstnem redu gledamo žetone, saj se bo v bistvu vse premike izvedlo hkrati (oz. z drugimi besedami: to, ali je nek premik dovoljen, je odvisno le od začetnega stanja mreže). Zato ni nič narobe, če žetone obravnavamo sistematično, od leve proti desni. Označimo polja naše mreže od leve proti desni s števili od 1 do w.

Pojdimo zdaj po naši mreži od leve proti desni, dokler ne naletimo na prvi žeton; recimo, da je to bel žeton na polju k. Nobene koristi ni od tega, da bi ta žeton premikali levo, saj so levo od njega le prazna polja, tako da s tem števila zasedenih polj ne bomo nič zmanjšali. Premik v desno pa je smiseln, če bomo lahko s tem naredili kup dveh ali celo treh belih žetonov na polju $k + 1$; v tem primeru ni nobene koristi od tega, da žetona k ne bi premaknili.[29] Če bi bil žeton na polju k črn, bi seveda razmišljali podobno. S postopkom potem nadaljujemo na polju $k + 1$, če žetona k nismo premikali; če pa smo ga, nadaljujemo pri prvem naslednjem polju, ki ga nismo uporabili pri tvorbi kupa na polju $k + 1$.

Zapišimo dobljeni postopek še s psevdokodo. Predpostavimo, da nam vrednost

[29] O tem se lahko prepričamo takole: če tak kup naredimo, bo po novem od polj k in $k + 1$ zasedeno le eno, namreč polje $k + 1$ (na katerem smo naredili kup). Če pa takega kupa ne naredimo, bo po novem od polj k in $k + 1$ zasedeno *vsaj* eno (namreč polje k, kjer smo beli žeton pustili pri miru); zato ta rešitev gotovo ni nič boljša, lahko pa je celo slabša.

z_k pove začetno stanje polja k (1 = bel žeton, −1 = črn žeton, 0 = prazno polje).

```
k := 1;
while k ≤ n:
  b := z_k;
  if b = 0 or z_{k+1} = −b or (z_{k+1} = 0 and z_{k+2} = −b) then
    k := k + 1 (* Polje k je prazno ali pa vsebuje žeton, ki ne more
                  tvoriti kupa na polju k + 1. *)
  else (* Naredimo kup barve b na polju k + 1. *)
    z_k := 0; z_{k+1} := b;
    if z_{k+2} = b then z_{k+2} := 0; k := k + 3
    else k := k + 2;
```

Predpostavili smo še, da imamo $z_{n+1} = z_{n+2} = 0$ (če pogledamo čez rob mreže, si mislimo, da so tam prazna polja). Na koncu bomo tako v tabeli z dobili končno stanje mreže (katera polja so zasedena in kakšne barve žetoni so tam), iz česar ni težko prešteti, koliko polj je zasedenih; lahko pa bi zasedena polja šteli že kar sproti, med samim izvajanjem glavne zanke (in tabele z sploh ne bi spreminjali, če nas podrobnosti o končnem stanju mreže ne zanimajo).

(d) Da bomo lažje razmišljali, se za začetek še vedno omejimo na mreže z eno samo vrstico, torej $h = 1$. Naloga sprašuje le po tem, koliko polj je v (najboljšem možnem) končnem stanju zasedenih; pri tem je torej vseeno, koliko žetonov je na nekem zasedenem polju. V končnem stanju mreže so torej za vsako polje pravzaprav možna le tri različna stanja: ali je tam kup belih žetonov, kup črnih žetonov ali pa je polje prazno. Končno stanje mreže si lahko torej predstavljamo kot urejeno w-terico $\mathbf{s} = (s_1, s_2, \ldots, s_w) \in \{\mathsf{B}, \mathsf{Č}, \mathsf{P}\}^w$. Rekli bomo, da je končno stanje $\mathbf{s}$ *veljavno*, če za vsak žeton v začetnem stanju mreže velja, da v končnem stanju mreže nastane na njegovem polju ali pa na vsaj enem od sosednjih polj kup žetonov enake barve, kot je naš opazovani žeton.

Podobne oznake lahko uporabimo tudi za začetno stanje mreže, recimo mu $\mathbf{z} = (z_1, z_2, \ldots, z_w)$, pri čemer nam posamezna komponenta pove, ali je na tistem polju v začetnem stanju bel žeton, črn žeton ali pa je prazno.

Kot pri mnogih nalogah je tudi tu koristno, če znamo v našem problemu opaziti malo manjše podprobleme istega tipa. Recimo, da bi se omejili na prvih $w - 1$ polj naše mreže; torej si mislimo, da polje w ne obstaja, na njem ni nobenega žetona in nanj tudi ne moremo premakniti morebitnega žetona s polja $w - 1$. Od stanja $\mathbf{s}$ tako ostane le $\mathbf{s}' = (s_1, \ldots, s_{w-1})$. Kakšna je zveza med veljavnostjo stanj $\mathbf{s}$ in $\mathbf{s}'$?

Opazimo lahko, da če je stanje $\mathbf{s}$ veljavno, to še ne pomeni, da je tudi $\mathbf{s}'$ veljavno; na primer, recimo, da je v začetnem stanju mreže na polju $w-2$ črn žeton, na polju $w-1$ bel žeton, polje w pa je prazno. V tem primeru si lahko predstavljamo veljavno stanje $\mathbf{s}$, ki ima $s_{w-1} = \mathsf{P}$, ker je to mogoče doseči tako, da beli žeton s polja $w - 1$ premaknemo na polje w; v stanju $\mathbf{s}'$ pa to ni veljavno, ker tam polje w ne obstaja in belega žetona s polja $w - 1$ nimamo kam premakniti.

Zato je koristno, če v opis stanja posameznega polja poleg dosedanjih treh možnosti (B, $\mathsf{Č}$ in P) dodamo še četrto, ki ji bomo rekli $\mathsf{Ž}$: to pomeni, da na tistem polju ne nastaja kup žetonov, nanj ne smemo nobenega žetona premakniti, vendar pa polje tudi ni prazno, temveč je na njem še vedno prav isti žeton kot v začetnem stanju mreže. (Tu nam ni treba ločiti med črnim in belim $\mathsf{Ž}$, saj je za vsako polje

mogoč le eden od teh dveh, namreč tisti, ki se ujema z barvo žetona, ki je bil na tem polju v začetnem stanju; če pa je bilo to polje v začetnem stanju prazno, je stanje Ž tam sploh nemogoče.)

Naj bo zdaj $f(k, s)$ najmanjše število zasedenih polj, ki ga lahko dosežemo, če se omejimo na prvih k polj mreže in če zahtevamo, da je končno stanje polja k enako s. Zdaj lahko razmišljamo takole:

- Za $f(k, \mathsf{P})$: če je bilo polje k v začetnem stanju prazno, je $f(k, \mathsf{P}) = \min_s f(k-1, s)$ (polje k ostane prazno in ne zanima nas, kaj se dogaja levo od njega); če je bil na polju k v začetnem stanju bel žeton, je $f(k, \mathsf{P}) = f(k-1, \mathsf{B})$ (polje k lahko spraznimo le tako, da beli žeton z njega prestavimo na beli kup, ki mora nastajati na $k-1$); in podobno, če je bil na k v začetnem stanju črn žeton, je $f(k, \mathsf{P}) = f(k-1, \check{\mathsf{C}})$.

- Za $f(k, \check{\mathsf{Z}})$: če je bilo polje k v začetnem stanju prazno, je $f(k, \check{\mathsf{Z}}) = \infty$ (ker je Ž dovoljen le na polju, kjer je bil na začetku žeton); sicer pa je $f(k, \check{\mathsf{Z}}) = \min_s f(k-1, s) + 1$ (žeton na polju k ostane, kjer je bil, polje je zato zasedeno, ni pa nam pomembno, kaj se dogaja levo od njega).

- Za $f(k, \mathsf{B})$: če je bil na polju k v začetnem stanju črn žeton, je $f(k, \mathsf{B}) = \infty$ (saj na takem polju ne moremo narediti belega kupa); sicer pa je $f(k, \mathsf{B}) = \min_s f(k-1, s)[+1]$, pri čemer $+1$ ne uporabimo, če je $z_{k-1} = \mathsf{B}$ in $s = \check{\mathsf{Z}}$ (kajti takrat lahko beli žeton s $k-1$ premaknemo na nastajajoči beli kup na k, tako da se število zasedenih polj ne spremeni).

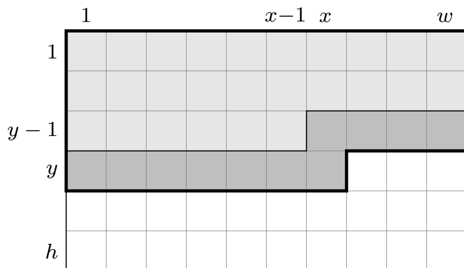
- Za $f(k, \check{\mathsf{C}})$ razmišljamo analogno kot pri $f(k, \mathsf{B})$.

S pomočjo tega rekurzivnega razmisleka lahko računamo rešitve $f(k, s)$ sistematično po naraščajočih k. Rešitev, po kateri sprašuje naloga, je na koncu preprosto $\min_s f(w, s)$.

Doslej smo razmišljali o mrežah z eno samo vrstico. Da se nam je rekurzivni razmislek lepo izšel, smo morali v opis podproblema poleg k (števila polj, na katera smo se pri tem podproblemu omejili) vključiti tudi s, torej stanje zadnjega polja (v tistem delu mreže, na katerega smo se omejili). Podoben, le malo bolj zapleten razmislek bo deloval tudi pri večjih mrežah (z več vrsticami, torej $h > 1$). Vrstice si mislimo oštevilčene od 1 do h, stolpce pa od 1 do w. Namesto enega samega indeksa k imejmo zdaj par koordinat (x, y), ki nam povesta, da smo se omejili na del mreže, ki ga tvorijo vrstice od 1 do $y-1$ in še levih x polj v vrstici y. Podobno kot prej tudi zdaj ta „omejili“ pomeni, da si mislimo, da preostanek mreže ne obstaja in da nanj ne moremo premikati žetonov.

Podobno kot smo prej (v rešitvi za eno samo vrstico) morali, ko smo razmišljali o polju k, vedeti nekaj o stanju njegovega soseda $k-1$, moramo tudi zdaj, ko razmišljamo o polju (x, y), poznati stanje njegovih sosedov, to pa so $(x-1, y-1)$, $(x, y-1)$, $(x+1, y-1)$ in $(x-1, y)$. (Ostali štirje sosedi ležijo zunaj dela tabele, na katerega smo se omejili.) In podobno, kot smo prej reševali podprobleme po naraščajočem k, si lahko predstavljamo, da jih bomo zdaj reševali po naraščajočem y, pri posameznem y pa po naraščajočem x. Ko bomo obdelali (x, y), se bomo torej posvetili polju $(x+1, y)$, takrat pa nas bo zanimalo tudi stanje njegovega soseda $(x+2, y-1)$; kasneje se bomo ukvarjali s poljem $(x+2, y)$ in takrat potrebovali

stanje njegovega soseda $(x+3, y-1)$. Če tako nadaljujemo, vidimo, da moramo v opis podproblema (x, y) vključiti stanje ne le vseh njegovih sosedov v že obdelanem delu mreže, ampak tudi vseh tistih polj, ki so v obdelanem delu mreže in ki so sosedje kakšnega polja v neobdelanem delu mreže. To so torej polja $(k, y-1)$ za $x-1 \le k \le w$ in (k, y) za $1 \le k \le x$.



Osenčeno območje kaže, na kateri del mreže se omejimo pri podproblemu (x, y). Temneje osenčena so tista polja, katerih stanje vključimo v opis podproblema (ker so sosedje kakšnega od polj zunaj osenčenega območja).

Prišli smo torej do takšnih podproblemov: naj bo $f(x, y, s_1, \dots, s_x, t_{x-1}, \dots, t_w)$, najmanjše število zasedenih polj, ki jih lahko dosežemo, če se omejimo na del mreže do polja (x, y) (zgornjih $y-1$ vrstic in še levih x polj v vrstici y) in če zahtevamo, da je končno stanje polja (k, y) enako s_k (za $1 \le k \le x$), polja $(k, y-1)$ pa t_k (za $x-1 \le k \le w$). Da bo manj pisanja, bomo spodaj uporabljali zapis $\mathbf{s}_i^j$ za zaporedje stanj $s_i, s_{i+1}, \dots, s_{j-1}, s_j$ (in podobno pri t-jih).

- Za $f(x, y, \mathbf{s}_1^{x-1}, \mathsf{P}, \mathbf{t}_{x-1}^w)$ moramo vzeti $\min_u f(x-1, y, \mathbf{s}_1^{x-1}, u, \mathbf{t}_{x-1}^w)$, če je bilo polje (x, y) prazno ali pa je bil na njem v začetnem stanju žeton in je na enem od sosednjih polj $s_{x-1}, t_{x-1}, t_x, t_{x+1}$ kup iste barve; sicer pa vzamemo ∞. Minimum po u vzamemo zato, ker nam je vseeno, v kakšnem stanju je bilo polje $(x-2, y-1)$.

- Za $f(x, y, \mathbf{s}_1^{x-1}, \check{\mathsf{Z}}, \mathbf{t}_{x-1}^w)$ moramo vzeti ∞, če je bilo polje (x, y) v začetnem stanju prazno, sicer pa vzamemo $\min_u f(x_1, y, \mathbf{s}_1^{x-1}, u, \mathbf{t}_{x-1}^w) + 1$: žeton ostane, kjer je bil, polje (x, y) je zato zasedeno (zato $+1$); za polje $(x-2, y-1)$ pa nam je vseeno, v katerem stanju u je bilo.

- Za $f(x, y, \mathbf{s}_1^{x-1}, \mathsf{B}, \mathbf{t}_{x-1}^w)$ moramo vzeti ∞, če je bil na polju (x, y) v začetnem stanju črn žeton; sicer pa vzamemo $\min f(x, y, \mathbf{s}_1^{x-2}, s'_{x-1}, u, t'_{x-1}, t'_x, t'_{x+1}, \mathbf{t}_{x+2}^w) + 1 - \triangle$. Ta minimum gre po vseh u in po vseh takih kombinacijah stanj $s'_{x-1}, t'_{x-1}, t'_x, t'_{x+1}$, iz katerih je mogoče dobiti stanja $s_{x-1}, t_{x-1}, t_x, t_{x+1}$ s premikom belih žetonov s teh sosednjih polj na polje (x, y). Pri s'_{x-1} nam ta pogoj na primer zahteva, da mora biti $s'_{x-1} = s_{x-1}$ ali pa mora veljati $s'_{x-1} = \check{\mathsf{Z}}$, $s_{x-1} = \mathsf{B}$ in žeton, ki je bil v začetnem stanju na $(x-1, y)$ mora biti bele barve. Analogne pogoje sestavimo tudi za t'_{x-1}, t'_x in t'_{x+1}. Število $\triangle$ je odvisno od $s'_{x-1}, t'_{x-1}, t'_x, t'_{x+1}$ in nam pove, s koliko od teh sosednjih polj smo premaknili žeton na (x, y). V mislih začnemo s $\triangle = 0$; če je $s'_{x-1} = \check{\mathsf{Z}}$ in $s_{x-1} = \mathsf{B}$, povečamo $\triangle$ za 1; podobno naredimo še pri t'_{x-1}, t'_x, t'_{x+1}.

- Za $f(x, y, \mathbf{s}_1^{x-1}, \check{\mathsf{C}}, \mathbf{t}_{x-1}^w)$ je razmislek analogen tistemu iz prejšnjega odstavka.

V dosedanjem razmisleku smo predpostavili, da polje (x, y) dejansko ima vse štiri sosede $(x-1, y)$, $(x-1, y-1)$, $(x, y-1)$ in $(x+1, y-1)$. Če je $x = 1$ ali $x = w$ ali pa $y = 1$, nekateri od teh sosedov odpadejo, zato bi bilo treba za te primere gornji razmislek še malo prilagoditi.

Število različnih podproblemov, ki jih pri tej rešitvi dobimo, je kar veliko: za x je w možnosti, za y je h možnosti, poleg tega pa imamo v opisu podproblema še $w + 1$ stanj, za vsakega od njih pa so štiri možnosti (P, Č, B in Ž); to je skupaj $wh \cdot 4^{w+1}$ podproblemov. Zato je tudi časovna zahtevnost naše rešitve $O(wh \cdot 4^w)$, prostorska pa je za faktor h manjša, kajti ko rešujemo podprobleme za nek y, lahko že pozabimo rešitve podproblemov za $y-2$, $y-3$ in tako naprej.

Če je $w > h$, je koristno mrežo na začetku zavrteti za 90 stopinj, tako da pride v eksponent krajša od obeh stranic. Naloga pravi, da bo krajša od obeh stranic dolga največ 10 enot, 4^{10} pa je približno en milijon, tako da je ta rešitev za našo nalogo še dovolj hitra.

19. Dvojiško Conwayevo zaporedje

Opazimo lahko, da se vsak niz t_k začne in konča na enico. O tem se lahko prepričamo z indukcijo: $t_0 = 1$ se res začne in konča na enico, ki je tudi edini znak niza. Za večje k pa sklepamo takole: predpostavimo, da za $k-1$ že vemo, da se t_{k-1} začne in konča na enico. Recimo natančneje, da se začne s skupino x enic, konča pa s skupino y enic. Njegov opis (na katerem temelji niz t_k) je torej oblike „x enic, nekaj ničel, ..., y enic“. Ker se ta opis konča na „enic“, se bo t_k končal s števko 1; in ker se opis začne z „x enic“, se bo t_k začel z dvojiškim zapisom števila x (ki mu bo sledila števka 1). Ker je x gotovo ≥ 1 (saj se je t_{k-1} začel na vsaj eno enico), se njegov dvojiški zapis začne z enico, zato se tudi t_k začne z enico.

Tako torej vidimo, da je vsak niz t_k takšne oblike: nekaj enic, nekaj ničel, nekaj enic, nekaj ničel, ..., nekaj ničel, nekaj enic. Takšni maksimalni strnjeni skupini enic, ki ji sledi maksimalna strnjena skupina ničel, recimo *blok*. (Zadnji blok v nizu je malo poseben primer, saj ima le enice in nobenih ničel.) Meja med blokoma nastopi natanko tam, kjer v nizu za ničlo pride enica.

Primer: niz $t_4 = 11110101$ je iz treh blokov, 11110 10 1.

Recimo, da imamo v t_k nek blok x enic in y ničel; kaj bo iz njega nastalo v naslednjem nizu (torej t_{k+1})? Opis takšnega bloka se začne z dvojiškim zapisom števila x; nato pride števka 1; sledi dvojiški zapis števila y; nato še števka 0. Podobno kot zgoraj tudi tu velja, da se dvojiški zapis števila x gotovo začne na 1; zato se torej naš opis začne z enico in konča z ničlo.

Opis naslednjega bloka niza t_k se bo spet začel z enico in končal z ničlo in tako naprej. Ker se torej opis enega bloka konča z ničlo, opis naslednjega bloka pa začne z enico, iz tega sledi, da bo meja med tema opisoma hkrati tudi meja med blokoma v nizu t_{k+1}. Tako torej vidimo, da iz enega bloka v nizu t_k nastane en ali več blokov v nizu t_{k+1}.

Blok iz x enic in y ničel označimo z B_{xy}. Pri nizu $t_0 = 1$ imamo le blok 1, to je B_{10}. Iz njega v naslednjem koraku dobimo 11, to je B_{20}. Iz tega dobimo nato 101, to je sklop dveh blokov: naprej 10 (torej B_{11}) in nato 1, kar je že znani B_{10}. Tako razmišljajmo naprej in si zapisujmo, kakšni bloki nastanejo in v kaj se nato

razvijejo:

$$\begin{array}{rclcrcl} 1 & \to & 11 & & B_{10} & \to & B_{20} \\ 11 & \to & 101 & & B_{20} & \to & B_{11}B_{10} \\ 10 & \to & 1110 & & B_{11} & \to & B_{31} \\ 1110 & \to & 11110 & & B_{31} & \to & B_{41} \\ 11110 & \to & 100110 & & B_{41} & \to & B_{12}B_{21} \\ 100 & \to & 11100 & & B_{12} & \to & B_{32} \\ 11100 & \to & 111100 & & B_{32} & \to & B_{42} \\ 111100 & \to & 1001100 & & B_{42} & \to & B_{12}B_{22} \\ 110 & \to & 10110 & & B_{21} & \to & B_{11}B_{21} \\ 1100 & \to & 101100 & & B_{22} & \to & B_{11}B_{22} \end{array}$$

Lepo pri tem je, da nam dolžine blokov ne naraščajo v nedogled; na desni strani vseh teh pravil se pojavljajo le taki bloki, ki jih imamo tudi na levi strani nekega drugega pravila. Vsak niz t_n je sestavljen le iz gornjih desetih blokov, ne glede na to, kako velik n opazujemo.

Označimo z $B_{xy}(k)$ niz, ki nastane iz bloka B_{xy} po k korakih. Zgoraj smo videli, da iz B_{10} v prvem koraku nastane B_{20}, zato iz B_{10} v k korakih nastane prav tisto, kar iz B_{20} nastane v $k-1$ korakih; torej imamo $B_{10}(k) = B_{20}(k-1)$. Podobno vidimo, da je $B_{20}(k) = B_{11}(k-1) \cdot B_{10}(k)$, pri čemer $\cdot$ pomeni stik nizov; in tako naprej.

S pomočjo teh rekurzivnih zvez ni težko določiti posameznega znaka poljubnega izmed teh nizov. Recimo, da nas zanima m-ti znak niza $B_{xy}(k)$. Če smo za blok B_{xy} zgoraj videli pravilo $B_{xy} \to B_{uv}$, to pomeni, da je niz $B_{xy}(k)$ enak nizu $B_{uv}(k-1)$, tako da smo naš problem prevedli na podoben problem, le z $k-1$ namesto k.

Če pa smo za blok B_{xy} zgoraj videli pravilo oblike $B_{xy} \to B_{uv}B_{wz}$, to pomeni, da je $B_{xy}(k) = B_{uv}(k-1)B_{wz}(k-1)$. Niz, ki ga opazujemo, je torej sestavljen iz dveh delov: naprej $B_{uv}(k-1)$ (recimo, da je dolg d znakov) in nato $B_{wz}(k-1)$. Če je $m > d$, je iskani znak v bistvu $(m-d)$-ti znak v nizu $B_{wz}(k-1)$, sicer pa moramo iskati m-ti znak v nizu $B_{uv}(k-1)$. Tudi zdaj smo torej naš problem prevedli na enak problem z $k-1$ namesto k. S takšnim razmišljanjem bi zdaj lahko nadaljevail, dokler ne pridemo do $k = 0$, tam pa je problem trivialen: niz $B_{xy}(0)$ je kar prvotni B_{xy}, sestavljen iz x enic in y ničel; torej, če je $m > x$, je iskani znak ničla, sicer pa enica.

V prejšnjem odstavku smo videli, da je koristno poznati tudi dolžine teh nizov, da lahko vemo, ali se iskani znak nahaja v levem ali desnem kosu niza. Tudi te dolžine lahko računamo s podobnimi rekurzivnimi zvezami kot same nize. Označimo z $d_{xy}(k)$ dolžino niza $B_{xy}(k)$. Če je ta niz oblike $B_{xy}(k) = B_{uv}(k-1)$, imamo tudi $d_{xy}(k) = d_{uv}(k-1)$; če pa je naš niz oblike $B_{xy}(k) = B_{uv}(k-1) \cdot B_{wz}(k-1)$, je njegova dolžina enaka $d_{xy}(k) = d_{uv}(k-1) + d_{wz}(k-1)$. Teh dolžin ni težko računati po naraščajočem k.

Naloga nas sprašuje po m-tem znaku niza t_n; ta niz pa ni nič drugega kot $B_{10}(n)$, saj je nastal v n korakih iz niza t_0, ki je enak bloku B_{10}. Ker smo zgoraj videli, kako izračunati m-ti znak poljubnega niza $B_{xy}(n)$, znamo zdaj odgovoriti tudi na vprašanje naše naloge. Zapišimo rešitev še v C-ju:

```
#define MaxN 100
enum { B10, B11, B12, B20, B21, B22, B31, B32, B41, B42, nBlokov };
```

```
const int stEnic[ ] = { 1, 1, 1, 2, 2, 2, 3, 3, 4, 4 };
const int stNicel[ ] = { 0, 1, 2, 0, 1, 2, 1, 2, 1, 2 };
const int pravila[nBlokov][2] = {
   {B20, −1}, {B31, −1}, {B32, −1}, {B11, B10},
   {B11, B21}, {B11, B22}, {B41, −1}, {B42, −1}, {B12, B21}, {B12, B22} };

int Stevka(int n, long long m)
{
   int b, b1, b2, x, y, k;
   long long dolzine[nBlokov][MaxN];
   /* Izračunajmo dolžine nizov. */
   for (b = 0; b < nBlokov; b++) dolzine[b][0] = stEnic[b] + stNicel[b];
   for (k = 1; k < n; k++) for (b = 0; b < nBlokov; b++) {
      b1 = pravila[b][0]; b2 = pravila[b][1];
      dolzine[b][k] = dolzine[b1][k − 1] + (b2 >= 0 ? dolzine[b2][k − 1] : 0); }
   /* Določimo iskani znak. */
   for (b = B10; n > 0; n−−)
   {
      /* Iščemo m-ti znak v nizu, ki nastane po n korakih iz bloka b. */
      b1 = pravila[b][0]; b2 = pravila[b][1];
      if (m <= dolzine[b1][n − 1]) b = b1;
      else b = b2, m −= dolzine[b1][n − 1];
   }
   return m <= stEnic[b] ? 1 : 0;
}
```

Naloga pravi, da gre lahko n do 100; izkaže se, da je niz t_{100} dolg približno $7{,}46 \cdot 10^{16}$ znakov, zato smo za računanje dolžin (in za indeks m) uporabili 64-bitni celoštevilski tip **long long**. (V splošnem je dolžina niza t_n približno $1{,}87 \cdot 1{,}46^n$ znakov.)

Mimogrede, podoben pristop bi se dalo uporabiti tudi pri desetiški različici Conwayevega zaporedja, vendar bi bili tam osnovni bloki daljši in bi jih bilo precej več kot deset.[30]

20. Neprireditveni stavki

Pri vsakem stavku imamo načeloma dve možnosti, kateri operator uporabimo v njem (prireditveni ali neprireditveni); če je vseh stavkov m, je to skupaj 2^m možnosti. Za vsako od teh možnosti bi lahko šli potem v zanki po vrsti čez vse stavke in simulirali izvajanje programa, na koncu pa pogledali, kje vse se je znašla vrednost, ki je bila ob začetku izvajanja v x_1. Ta rešitev bi dajala pravilne odgovore za majhne m (torej za kratke programe), za naš namen pa je prepočasna, saj besedilo naloge pravi, da je lahko število stavkov m tudi več sto. Oglejmo si zato še učinkovitejšo rešitev, ki temelji na dinamičnem programiranju.

Vrednost, ki je bila pred začetkom izvajanja v spremenljivki x_1, označimo z Z. Opazimo lahko, da je za nas pri delovanju programa pravzaprav pomembno le to, katere spremenljivke trenutno vsebujejo vrednost Z. Temu recimo *stanje* sistema; lahko ga predstavimo na primer z m-bitnim celim številom (torej številom od 0 do $2^m - 1$), pri čemer prižgani biti povedo, katere spremenljivke imajo vrednost Z. Za

[30] Za več o teh zaporedjih glej npr. Wikipedijo *s. v.* Look-and-say sequence ter zaporedji A001387 in A005150 v *The Online Encyclopedia of Integer Sequences*.

nadaljnje delovanje programa je pomembno le trenutno stanje sistema, ne pa tudi to, kako (torej s kakšnim zaporedjem stavkov) je sistem v to stanje prišel.

Naloga pravi, da imajo pred začetkom izvajanja vse spremenljivke različno vrednost; torej ima vrednost Z takrat samo spremenljivka x_1, zato je sistem v stanju $s = 1$ (prižgan je le najnižji bit, ostali so ugasnjeni). Od tu naprej pa, če poznamo stanje s pred izvedbo nekega stavka, ni težko izračunati stanja po tem stavku. Pri neprireditvenem stavku je stvar sploh trivialna, saj ta stavek ne naredi ničesar, zato se tudi stanje sistema ne spremeni. Pri prireditvenem stavku $x_i == x_j$ pa razmišljamo takole: po tej prireditvi ima x_j vrednost Z natanko v primeru, ko je imel pred prireditvijo x_i vrednost Z; torej moramo v stanju s bit j postaviti na prav tisto vrednost, ki jo ima bit i.

Rekli bomo, da je stanje s *dosegljivo* po k korakih, če lahko s primernim izborom operatorjev v prvih k stavkih programa dosežemo, da bo sistem po koncu izvajanja teh k stavkov v stanju s. Naloga zdaj pravzaprav sprašuje po tem, katero izmed stanj, ki so dosegljiva po m korakih (torej na koncu programa), vsebuje največ spremenljivk (= ima največ prižganih bitov). Neko stanje s je dosegljivo po m korakih takrat, ko je bodisi dosegljivo že po $m - 1$ korakih (tedaj je dovolj že, če v m-tem koraku uporabimo neprireditveni operator, tako da se stanje ne spremeni), bodisi je dosegljivo po $m - 1$ korakih neko takšno stanje s', iz katerega nastane stanje s, če v m-tem koraku uporabimo prireditveni operator.

Vidimo torej, da če hočemo ugotoviti, katera stanja so dosegljiva po m korakih, je koristno pred tem ugotoviti, katera stanja so dosegljiva po $m - 1$ korakih; pred tem pa seveda, katera so dosegljiva po $m-2$ korakih in tako nazaj. Zato je koristno dosegljivost računati lepo sistematično od začetka programa proti koncu. Naloga sprašuje tudi po tem, kako do najboljšega stanja priti s čim manj spremembami operatorjev, zato je koristno poleg tega, ali je neko stanje dosegljivo ali ne, hraniti tudi podatek o tem, kolikšno je najmanjše število sprememb operatorjev, s katerimi je dosegljivo; temu recimo $f_k(s)$. Če stanje s po k sploh ni dosegljivo, si mislimo $f_k(s) = \infty$. Tako smo dobili naslednji postopek:

(* *Pred začetkom izvajanja je sistem v stanju* $s = 1$ (*le spremenljivka* x_1 *vsebuje vrednost* Z) *in to je edino takrat dosegljivo stanje.* *)
for $s := 0$ **to** $2^n - 1$ **do** $f_k[s] := \infty$;
$f_k[1] := 0$;
(* *Poglejmo, kaj je dosegljivo po več korakih.* *)
for $k := 1$ **to** m:
  Recimo, da je k-ti stavek programa oblike x_i *op* x_j.
  if je *op* prireditveni operator **then** $c := 0$ **else** $c := 1$;
  (* *Za ceno* c *sprememb lahko v* k*-ti vrstici dobimo neprireditveni operator; z njim postanejo dosegljiva ista stanja kot po* $k - 1$ *korakih.* *)
  for $s := 0$ **to** $2^n - 1$ **do** $f_k[s] := f_{k-1}[s] + c$;
  (* *Za ceno* $1 - c$ *sprememb pa lahko v* k*-ti vrstici dobimo prireditveni operator; poglejmo, kaj lahko dosežemo z njim.* *)
  for $s := 0$ **to** $2^n - 1$:
    $s' :=$ stanje, ki ga dobimo, če v s postavimo bit, ki predstavlja spremenljivko x_j, na vrednost, ki jo ima bit, ki predstavlja spremenljivko x_i;

$f_k[s'] := \min\{f_k[s'], f_{k-1}[s] + (1 - c)\}$;

Na koncu moramo iti še enkrat po tabeli f_n in med tistimi stanji s, ki imajo $f_n[s] < \infty$, poiskati tisto z največ prižganimi biti; če pa je takih več, med njimi vzamemo tisto z najmanjšo vrednostjo $f_n[s]$. Časovna zahtevnost tega postopka je le $O(m \cdot 2^n)$, kar je precej bolj obvladljivo kot $O(2^m)$, saj naloga pravi, da je n (število spremenljivk) največ 20. Opazimo lahko še, da smemo tabele f_k sproti pozabljati: ko računamo f_k, potrebujemo le f_{k-1}, ne pa več tabel f_{k-2}, f_{k-3} itd. Tako bomo porabili le $O(2^n)$ pomnilnika namesto $O(m \cdot 2^n)$.

Naloge so sestavili: malica, taksist — Nino Bašić; motocikel — Boris Gašperin; Google Mobil — Matija Grabnar; podobna števila, žetoni — Tomaž Hočevar; kazalca, tangram — Boris Horvat; vodovod — Nace Hudobivnik; VIP, prehod — Jurij Kodre; prestopna leta, celične himere, 3-d šah — Mitja Lasič; rekonstrukcija drevesa — Matija Lokar; avtomobil na daljinsko vodenje, tat in laserji — Mitja Trampuš; dvojiško Conwayevo zaporedje — Mitja Trampuš in Janez Brank; training kode, neprireditveni stavki — 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. Ocenjevanje profesorjev

- Mišljeno je, da učinkovita rešitev pri tej nalogi porabi $O(n + m)$ časa; rešitve, ki porabijo na primer $O(n \cdot m)$ časa (ker imajo gnezdeni zanki po vseh dijakih in po vseh profesorjih) naj dobijo (če so sicer pravilne) največ 12 točk.

- Format izpisa pri tej nalogi ni pomemben — dovolj je, da program izračuna obe vrednosti, po katerih sprašuje naloga, in ju v nekakšni obliki izpiše.

- Če si rešitev pripravi tabelo, v kateri je po en element za vsakega profesorja, je vseeno, ali je ta tabela vedno maksimalno velika (100 000 elementov, ker je tudi profesorjev največ toliko) ali pa jo program alocira dinamično glede na dejansko število profesorjev.

- Pri tej nalogi gredo številke profesorjev od 1 naprej, indeksi v tabelah pa so v mnogih programskih jezikih od 0 naprej. Program lahko, če hoče naslavljati tabelo s številkami profesorjev, preračunava med obojim tako, da številki profesorja odšteje 1, preden jo uporabi kot indeks v tabelo, lahko pa tudi alocira za en element večjo tabelo in potem v njej uporablja elemente z indeksi od 1 naprej, element 0 pa ostane neuporabljen. Oboje je enako dobro. Če ima v zvezi s tem program kakšne napake (na primer alocira tabelo n elementov in jo naslavlja s števili od 1 do n namesto od 0 do $n - 1$; ali pa če v rezultatih izpiše številko profesorja od 0 do $n - 1$ namesto od 1 do n), naj se mu zaradi tega odbije največ dve točki.

- Če program sicer pravilno izračuna skupno število točk vsakega profesorja, vendar se zmoti pri štetju profesorjev z negativnim številom točk, naj se mu zaradi tega odbije največ tri točke. Podobno naj se mu tudi, če se zmoti pri ugotavljanju profesorja z najnižjim številom točk, zaradi tega odbije največ tri točke.

2. Spraševanje

- Glavni izziv pri tej nalogi je, kako učinkovito izvesti operaciji Vprasan(i) in Koliko. Kot smo videli v našem primeru rešitve, je mogoče obe operaciji izvesti v $O(1)$ časa. Točkovanje manj učinkovitih rešitev naj bo takšno: rešitev, pri kateri kakšna od teh operacij porabi $O(n^2)$ časa, naj dobi največ 8 točk (če je drugače pravilna); če operaciji porabita $O(n)$ časa, naj dobi rešitev največ 14 točk (če je drugače pravilna); če operaciji porabita $O(d)$ ali $O(\log n)$ časa, naj dobi rešitev največ 18 točk (če je drugače pravilna).

- Ker je naloga tipa „opiši“, je popolnoma sprejemljivo tudi, če je rešitev opisana s psevdokodo ali v naravnem jeziku, pomembno je le, da je dovolj jasna.

- Če bi rešitev dajala napačne rezultate zaradi drobnih napak pri naslavljanju tabel (npr. ker bi tabelo imela tabelo z indeksi od 0 do $n - 1$, pomotoma pa bi jo naslavljala s številkami učencev od 1 do n), naj se ji zaradi tega odšteje največ dve točki.

3. Žica

- Pri ocenjevanju rešitve naj bo poudarek predvsem na tem, ali vrača pravilne rezultate, ne pa toliko na tem, ali je elegantna. Na primer, zasuk v levo bi se dalo izvesti s takšnim pogojnim stavkom:

```
enum { Gor, Dol, Levo, Desno } smer;
⋮
if (smer == Gor) smer = Levo;
else if (smer == Levo) smer = Dol;
else if (smer == Dol) smer = Desno;
else smer = Gor;
```

  Podobno bi se dalo s pogojnimi stavki narediti tudi zasuk v desno in premike. Takšne rešitve naj se šteje za enako dobre kot tiste elegantnejše.

- Če program namesto nizov „`Da`“ in „`Ne`“ izpiše kaj drugega oz. je iz njega kako drugače razvidno, da je pravilno izračunal, ali je žica sklenjena ali ne, naj se mu zaradi tega odšteje največ tri točke.

- Pričakujemo, da bodo imeli pri tej nalogi nekateri tekmovalci težave z branjem vhodnih podatkov, ker se v njih mešajo črke in števila. Ni pa mišljeno, da bi bil to pomemben del izziva pri tej nalogi; rešitvam naj se za manjše napake pri branju vhodnih podatkov odšteje največ tri točke.

4. Račja

- Nekateri programski jeziki imajo v svoji standardni knjižnici razne koristne funkcije za delo z nizi, na primer za razbijanje niza na podnize pri presledkih. Čisto v redu je, če si rešitev pomaga s takšnimi funkcijami, četudi to pomeni, da je zaradi tega malo manj učinkovita.

- Če rešitev pomotoma prebere n vrstic z opisi oglašanja rac namesto $n + 1$ vrstic, naj se ji zaradi tega odšteje največ dve točki.

- Če rešitev predpostavi, da so v vsaki vrstici po trije zlogi (tako kot v primeru v besedilu naloge, čeprav naloga posebej poudarja, da je v splošnem število zlogov lahko drugačno), naj se ji zaradi tega odšteje pet točk. Če pa rešitev sicer načeloma deluje za poljubno število zlogov, vendar ne do 100 zlogov (kot pravi besedilo naloge), pač pa si pomotoma postavi neko nižjo zgornjo mejo za število zlogov, naj se ji zaradi tega odšteje dve točki.

5. Kraljice

- Poudarek pri tej nalogi je, da je šahovnica velika, število kraljic pa ni ekstremno veliko. Temu naj bo prilagojeno tudi ocenjevanje rešitev. Za vse točke pričakujemo rešitev, ki porabi največ $O(n + k)$ ali $O(k^2)$ časa in največ $O(n + k)$ prostora. Rešitev, ki porabi $O(k^3)$ časa, naj dobi največ 18 točk (če

je drugače pravilna); rešitev, ki porabi $O(n \cdot k)$ časa, naj dobi največ 14 točk (če je drugače pravilna); rešitev, ki porabi (n^2) časa in/ali pomnilnika, naj dobi največ osem točk (če je drugače pravilna).

- Če bi rešitev pomotoma predpostavila, da so koordinate kraljic v vhodnih podatkih podane v razponu od 0 do $n - 1$ namesto od 1 do n, naj se ji zaradi tega ne odbija točk.
- Če rešitev pri preverjanju, ali se dve kraljici napadata, pozabi preveriti, ali ni med njima nobene tretje (ki bi njuno medsebojno napadanje blokirala), in zaradi tega vrne napačen rezultat, naj se ji zaradi tega odšteje pet točk.

Težavnost nalog

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

Naloga	Kam bi sodila po težavnosti na državnem tekmovanju ACM
1. Ocenjevanje profesorjev	lažja do srednja naloga v prvi skupini
2. Spraševanje	težka v prvi ali lažja naloga v drugi skupini
3. Žica	težja v prvi ali lažja naloga v drugi skupini
4. Račja	težja v prvi ali lahka naloga v drugi skupini
5. Kraljice	srednje težka naloga v drugi skupini

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

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

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, so pa možna tudi odstopanja od tega, če je to glede na rezultate bolj primerno; do nekaj takih primerov je prišlo tudi letos. 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 277) in smo jih letos podelili deset. Srebrna priznanja pa se podeljujejo po podobnih kriterijih kot v prejšnjih letih 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 107 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	∑
1Z	1	Jaka Kordež	3	ŠC Kranj, SŠER	20	14	19	20	20	93
2Z	2	Miha Rot	4	Gimnazija Kranj	20	10	18	20	19	87
2Z	3	Nejc Kadivnik	3	ŠC Kranj, Str. gim.	15	16	15	20	20	86
2Z	4	Miloš Ljubotina	4	SŠJJ Ivančna Gorica	15	12	19	19	20	85
3S	5	Žiga Kokelj	4	Gimnazija Škofja Loka	13	15	20	20	16	84
3S	6	Robi Novak	3	SERŠ Maribor	15	19	18	17	14	83
3S	7	Tadej Plos	3	III. gimnazija Maribor	20	9	17	17	18	81
S	8	Simon Oberžan	4	ŠC Celje, Gimn. Lava	12	15	15	16	20	78
S	9	Jaka Mohorko	3	II. gimnazija Maribor	12	12	15	18	20	77
S	10	Žiga Patačko Koderman	1	Gimnazija Vič	19	15	10	16	15	75
S		David Pintarič	2	SPTŠ Murska Sobota	12	15	15	19	14	75
S	12	Jan Tomšič Pivk	3	Vegova Ljubljana	12	15	19	9	19	74
S	13	Boštjan Kloboves	1	ZRI	15	15	7	19	17	73
S		Andraž Jelenc	3	Gimnazija Škofja Loka	12	14	7	20	20	73
S		Martin Prelog	9	OŠ S. Jenka Kranj	20	15	9	15	14	73
S	16	Aleksander Rajnhard	4	Gimnazija Škofja Loka	12	15	17	10	18	72
S	17	Matija Lazić	3	Vegova Ljubljana	8	15	12	19	17	71
S		Klemen Gumzej	3	ŠC Celje, SŠ za KER	7	15	15	17	17	71
S	19	Klemen Pevec	2	Vegova Ljubljana	15	5	10	19	20	69
S	20	Janez Radeščček	4	ŠC Novo mesto, SEŠTG	20	12	0	19	17	68
S		Leon Gorjup	3	SERŠ Maribor	20	1	18	16	13	68
S	22	Peter Matičič	2	Vegova Ljubljana	15	10	5	18	18	66
S	23	David Popović	1	Gimnazija Bežigrad	5	20	15	10	15	65
S	24	Anže Bertoncelj	4	ŠC Kranj, Str. gim.	10	9	7	18	20	64
S		Klemen Kogovšek	3	Vegova Ljubljana	5	8	12	19	20	64
S	26	Luka Pogačnik	3	Gimnazija Vič	20	12	5	15	11	63
S		Primož Hrovat	3	ŠC Novo mesto, SEŠTG	12	15	0	18	18	63
S	28	Boštjan Pintar	3	ŠC Kranj, SŠER	12	11	15	13	11	62
S	29	Jakob Bambič	3	ŠC Novo mesto, SEŠTG	15	4	7	20	15	61
S		Miha Mitič	9	OŠ S. Jenka Kranj	10	15	7	12	17	61
S		Matej Tomc	3	Škof. klas. gimn. Lj.	20	8	9	6	18	61
S		Jernej Klarić	4	ŠC Celje, Gimn. Lava	15	8	12	6	20	61
S	33	Gregor Ažbe	3	ŠC Kranj, SŠER	15	15	0	12	18	60
S	34	Gregor Rihtaršič	1	ZRI	19	12	15	12	0	58
S	35	Rok Šeško	3	II. gimnazija Maribor	15	15	12	5	10	57
S	36	Amon Stopinšek	3	STPŠ Trbovlje	12	10	5	15	14	56
S		Jakob Gaberc Artenjak	3	ŠC Ptuj, ERŠ	12	15	5	7	17	56
S	38	Marko Laharnar	3	STPŠ Trbovlje	20	7	11	6	11	55
S		Maj Škerjanc	4	Gimnazija Kranj	10	16	12	5	12	55
S	40	Aljaž Koželj	4	Gimnazija Kranj	8	12	0	18	16	54
S	41	Domen Lanišnik	4	STPŠ Trbovlje	12	2	5	13	20	52
S		Zen Lednik	2	ŠC Celje, SŠ za KER	20	8	0	8	16	52
S		Klemen Jesenovec	3	Vegova Ljubljana	10	10	2	18	12	52

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
S	44	Benjamin Benčina	2	ZRI	7	15	3	10	16	51
S		Martin Čebular	3	ŠC Novo mesto	15	8	5	6	17	51
S		Aljaž Mislovič	3	ŠC Ptuj, ERŠ	8	12	0	14	17	51
S		Andraž Juvan	2	Vegova Ljubljana	20	3	3	12	13	51
	48	Luka Zorko	3	ŠC Novo mesto	12	10	15	8	3	48
		Benjamin Kraner	2	II. gimnazija Maribor	7	10	5	10	16	48
		Alen Nemec	4	STPŠ Trbovlje	20	5	5	14	4	48
	51	Matej Bevec	1	Gimnazija Bežigrad	14	13	5	15	0	47
		Vid Drobnič	1	Gimnazija Vič	0	12	3	14	18	47
		Rok Ljubešek	2	Gimnazija Vič	15	20	7	5	0	47
	54	Uroš Štok	2	SERŠ Maribor	0	5	5	20	15	45
	55	Matej Hacin	4	STPŠ Trbovlje	15	12	4	8	5	44
	56	Matic Jan	3	ŠC Kranj, SŠER	15	10	2	11	5	43
	57	Aleš Ravnikar	3	STPŠ Trbovlje	8	0	3	10	19	40
		Žiga Kotnik Klovar	4	ŠC Celje, Gimn. Lava	10	1	1	10	18	40
	59	Rok Kovač	1	Gimnazija Vič	2	15	12	1	9	39
	60	Matej Logar	4	Gimnazija Vič	0	12	10	6	10	38
	61	Gašper Romih	4	Gimnazija Vič	2	7	3	9	16	37
		Blaž Ocepek	4	STPŠ Trbovlje	10	0	1	9	17	37
	63	Anže Košir	2	Vegova Ljubljana	0	4	5	12	13	34
	64	Anže Kovač	4	STPŠ Trbovlje	10	0	5	8	10	33
	65	Uroš Burjek	3	III. gimnazija Maribor	8	8	6	10	0	32
	66	Ivan Kolundžija	4	Gimnazija Vič	8	15	7	0	0	30
	67	Tilen Merše	2	SŠ Domžale	17	7	0	5	0	29
	68	Erika Stanković	2	SŠ V. Pilon Ajdovščina	5	8	5	4	5	27
	69	Tine Bajec	2	SŠ V. Pilon Ajdovščina	2	3	16	0	5	26
	70	Goran Tubić	3	Vegova Ljubljana	15	2	0	5	3	25
		Domen Gašperlin	3	Gimnazija Kranj	12	5	3	0	5	25
		Noel Gregori	1	Gimnazija Jesenice	5	0	12	5	3	25
	73	Lara Prijon	1	Gimnazija Vič	0	8	0	7	9	24
		Nejc Hirci	1	Gimnazija Vič	7	10	7	0	0	24
		Aleksander Krašovec	4	STPŠ Trbovlje	10	3	5	6	0	24
		Darjo Uršič	2	SŠ V. Pilon Ajdovščina	6	15	3	0	0	24
	77	Jaka Jenko	2	ŠC Velenje, ERŠ	12	0	1	0	10	23
		Katja Gosar	2	Gimnazija Vič	10	10	0	0	3	23
	79	Jaka Strmčnik	1	Gimnazija Bežigrad	10	2	3	3	1	19
		Franci Šacer	3	ŠC Celje, SŠ za KER	8	1	1	7	2	19
	81	Gal Giacomelli	4	Gimnazija Šentvid	4	0	12	0	2	18
		Žiga Sitar	2	ŠC Velenje, ERŠ	7	8	1	2	0	18
	83	Matic Dokl	1	II. gimnazija Maribor	0	1	1	0	13	15
	84	Uroš Šinigoj	1	Gimnazija Vič	5	5	0	4	0	14
		Matic Šincek	2	ŠC Velenje, ERŠ	2	7	0	5	0	14
	86	Anže Kuret	2	SŠ Domžale	10	1	0	2	0	13
	87	Jaša Žnidar	9	OŠ S. Jenka Kranj	0	0	1	5	5	11
	88	Jaka Grbac	2	Gimnazija Vič	8	1	0	0	0	9
		Žiga Udovič	2	SŠ Domžale	5	0	0	4	0	9
	90	Tadej Kostanjevec	3	ŠC Ptuj, ERŠ	2	0	0	5	0	7
	91	Rok Kovačič	2	SŠ Domžale	2	0	1	0	0	3
	92	Leon Abraham	2	SPTŠ Murska Sobota	0	0	1	1	0	2
		Miha Bogataj	2	ŠC Kranj, Str. gim.	0	1	0	0	1	2
	94	Vid Prezelj	1	Gimnazija Bežigrad	0	0	1	0	0	1

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1Z	1	Luka Kolar	3	Gimnazija Vič	18	12	14	13	20	77
1Z	2	Žiga Željko	1	ZRI	20	10	18	20	8	76
2Z	3	Samo Remec	3	Vegova Ljubljana	18	13	12	19	8	70
2Z	4	Sandi Režonja	3	Gim. Murska Sobota	19	20	17	2	10	68
2S	5	Matevž Poljanc	3	Škof. klas. gimn. Lj.	20	7	13	17	10	67
3S	6	Nejc Smrkolj Koželj	4	Vegova Ljubljana	19	7	16	14	8	64
S	7	Jakob Erzar	3	Gimnazija Kranj	18	7	17	13	8	63
S	8	Žiga Šmelcer	4	Šk. kl. gim. Lj. + ZRI	20	1	14	16	0	51
S	9	Tadej Medved	4	ŠC Nova Gorica	14	4	12	13	5	48
S	10	Rok Kos	2	Gimnazija Vič in ZRI	17	1	14	7	3	42
S	11	Jan Likar	4	SŠ V. Pilon Ajdovščina	2	2	13	16	8	41
S	12	Nejc Prikeržnik	3	ŠC Ravne na Koroškem	11	2	7	10	8	38
	13	Nejc Savodnik	3	Gimnazija Šentvid	0	0	17	11	8	36
	14	Bor Brecelj	2	Gimnazija Vič in ZRI	0	7	15	6	0	28
	15	Žan Knafelc	3	ZRI	10	0	9	6	2	27
	16	Mark Štokelj	4	ŠC Nova Gorica	5	0	16	3	2	26
	17	Domen Jerič	4	Gimnazija Kranj	1	1	12	5	1	20
	18	Mark Lisičič	2	ZRI	4	0	0	13	0	17
	19	Tim Hafner	3	ZRI in Gimn. Kranj	3	2	0	3	7	15
	20	Žiga Leskovšek	2	Vegova Ljublj. in ZRI	0	0	0	5	5	10
		Vid Leskovar	4	II. gimnazija Maribor	1	0	7	0	2	10
		David Štaleker	3	ŠC Ravne na Koroškem	0	0	0	10	0	10
	23	Drago Miklauc	3	ŠC Ravne na Koroškem	0	2	0	0	1	3
	24	Aljaž Šešo	3	ŠC Ptuj, ERŠ	0	0	0	0	0	0
		Matic Korošec	4	ŠC Velenje, ERŠ	0	0	0	0	0	0

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj)					
					1	2	3	4	5	∑
1Z	1	Patrik Zajec	4	ŠC Srečka Kosovela Sežana in ZRI	100	100	94	100	97	491
1Z	2	Vid Kocijan	4	Gimnazija Vič in ZRI	100	100	24	94	97	415
2Z	3	Aleksej Jurca	1	Gimnazija Bežigrad	91	100		97		288
2S	4	Filip Koprivec	4	Gimnazija Vič	87	97			90	274
2S	5	Tobias Mihelčič	4	Vegova Ljubljana	72	90		88	14	264
3S	6	Žiga Gradišar	4	Šk. kl. gim. Lj. + ZRI	87	82				169
S	7	Aljaž Jeromel	4	II. gimnazija Maribor	11	44		90	10	155
	8	Metod Medja	3	ŠC Kranj, SŠER	24	97		0	11	132
	9	Aljaž Eržen	2	ZRI	8	94	0	0	0	102
	10	Jan Aleksandrov	4	ZRI	21	50				71
	11	Jakob Murko	4	Gimnazija Ptuj	17	50				67
	12	Simon Weiss	4	ZRI	14	47				61
	13	Peter Filip Lebar	4	Gimnazija Vič	0	50				50
		Matej Mulej	2	ŠC Kranj, SŠER	0	50				50
	15	Žiga Simončič	4	Vegova Ljubljana	5	7				12

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Jaka Kordež	tablični računalnik GoClever Orion 97
1	2	Miha Rot	tablični računalnik GoClever Orion 70
1	2	Nejc Kadivnik	3 TB zunanji disk
1	2	Miloš Ljubotina	2 TB zunanji disk
1	3	Žiga Kokelj	2 TB flash disk
1	3	Robi Novak	miška Razer Naga Hex
1	3	Tadej Plos	miška Razer Naga Hex
2	1	Luka Kolar	tablični računalnik GoClever Orion 97
			Raspberry Pi model B
			Dasgupta *et al.: Algorithms*
2	1	Žiga Željko	tablični računalnik GoClever Orion 97
			Dasgupta *et al.: Algorithms*
2	2	Samo Remec	2 TB zunanji disk
			Dasgupta *et al.: Algorithms*
2	2	Sandi Režonja	2 TB zunanji disk
2	2	Matevž Poljanc	miška Razer Naga Hex
2	3	Nejc Smrkolj Koželj	miška Razer Naga Hex
3	1	Patrik Zajec	tablični računalnik GoClever Orion 97
			Raspberry Pi model B
			Cormen *et al.: Introduction to algorithms*
			Eijkhout: *TEX by Topic*
3	1	Vid Kocijan	tablični računalnik GoClever Orion 70
			Raspberry Pi model B
			Cormen *et al.: Introduction to algorithms*
			Eijkhout: *TEX by Topic*
3	2	Aleksej Jurca	2 TB zunanji disk
			Knuth: *The Art of Computer Programming*, Vol. 4A
			Eijkhout: *TEX by Topic*
3	2	Filip Koprivec	2 TB zunanji disk
3	2	Tobias Mihelčič	64 GB USB ključ
3	3	Žiga Gradišar	64 GB USB ključ
Off-line naloga — Zlaganje likov			
	1	Patrik Zajec	Raspberry Pi model B

Poleg tega je vsak od nagrajencev prejel tudi izvod knjige *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004* (v dveh zvezkih, IJS, 2006).

SODELUJOČE ŠOLE IN MENTORJI

Šola	Mentorji
Druga gimnazija Maribor	Mirko Pešec
Gimnazija Bežigrad	Andrej Šuštaršič, Jurij Železnik
Gimnazija Jesenice	Marko Kikelj
Gimnazija Kranj	Zdenka Vrbinc, Mateja Žepič
Gimnazija Murska Sobota	Romana Vogrinčič
Gimnazija Ptuj	
Gimnazija Šentvid	Nastja Lasič
Gimnazija Škofja Loka	Anže Nunar
Gimnazija Vič	Klemen Bajec, Andrej Brodnik, Marjan Greselj, Nataša Kristan, Marina Trost
Osnovna šola Simona Jenka Kranj	Luka Flajnik
Srednja elektro-računalniška šola Maribor (SERŠ)	Milena Milanovič, Vida Motaln, Manja Sovič Potisk
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Igor Kutoš, Boris Ribaš
Srednja šola Domžale	Marko Bešlič
Srednja šola Josipa Jurčiča Ivančna Gorica	Darko Pandur
Srednja šola Veno Pilon Ajdovščina	Marko Pregeljc
Srednja tehniška in poklicna šola Trbovlje (STPŠ)	Uroš Ocepek
Škofijska klasična gimnazija Šentvid	Helena Medvešek, Jure Slak
Šolski center Celje, Gimnazija Lava	Karmen Kotnik
Šolski center Celje, Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Dušan Fugina
Šolski center Kranj, Srednja šola za elektrotehniko in računalništvo (SŠER)	Aleš Hvasti
Šolski center Kranj, Strokovna gimnazija	Gašper Strniša
Šolski center Nova Gorica	Boštjan Vouk

Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija (SEŠTG) — Albert Zorko, Simon Vovko

Šolski center Ptuj, Elektro in računalniška šola (ERŠ) — Zoltan Sep, Franc Vrbančič

Šolski center Ravne na Koroškem, Srednja šola Ravne — Zdravko Pavleković

Šolski center Srečka Kosovela Sežana

Šolski center Velenje, Elektro in računalniška šola (ERŠ) — Miran Zevnik

Tretja gimnazija Maribor — Maja Čelan

Vegova Ljubljana — Aleksandar Lazarević, Nataša Makarovič, Darjan Toth

Zavod za računalniško izobraževanje (ZRI), Ljubljana

OFF-LINE NALOGA — ZLAGANJE LIKOV

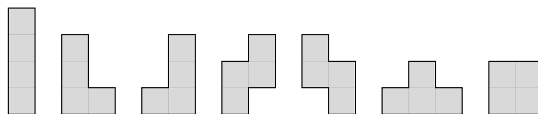
Na računalniških tekmovanjih, kot je naše, je čas reševanja nalog precej omejen in tekmovalci imajo za eno nalogo v povprečju le slabo uro časa. To med drugim pomeni, da je marsikak zanimiv problem s področja računalništva težko zastaviti v obliki, ki bi bila primerna za nalogo na tekmovanju; pa tudi 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. Takšne naloge smo razpisovali že v letih 2007 in 2008, letos pa smo s tem poskusili znova. Opis naloge in testne primere smo objavili novembra 2013 skupaj z razpisom za tekmovanje v znanju; tekmovalci so imeli čas do 28. marca 2013 (dan pred tekmovanjem), da pošljejo svoje rešitve.

Opis naloge

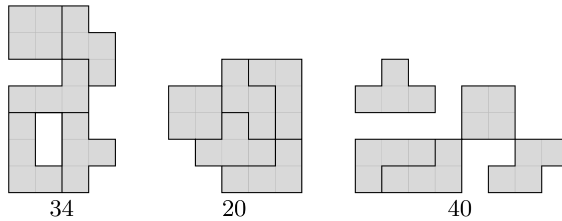
Dano je veliko število likov iz igre Tetris. Naloga je zložiti like v večji lik s čim manjšim obsegom, pri čemer se liki med seboj ne smejo prekrivati. Pri tem je novi „lik“ lahko tudi sestavljen iz več nepovezanih delov, lahko vsebuje luknje in podobno. Obseg je definiran kot skupna dolžina vseh robov, pri katerih liki mejijo na belo podlago naše kariraste mreže (namesto na druge like).

Možne oblike likov so naslednje:



Primer: recimo, da imamo naslednje like:

Teh pet likov lahko zložimo na veliko različnih načinov in dosežemo različno velike obsege. Naslednja slika prikazuje tri izmed njih in pod vsakim še njegov obseg:



Med temi tremi razporedi je torej najboljši tisti v sredini, ki ima obseg samo 20 enot.

Testni primeri

Pripravili smo 300 testnih primerov, pri vsakem od njih pa velja omejitev, da je število likov posamezne oblike kvečjemu 300. Skupno število likov pri vsakem testnem primeru je torej lahko največ 2100; ni pa nujno, da so v vsakem testnem primeru prisotni liki vseh sedmih oblik. Število testnih primerov je veliko zato, ker smo hoteli odvrniti ljudi od oddajanja ročno sestavljenih razporedov (po naših izkušnjah je z nekaj truda pogosto mogoče ročno dobiti zelo dobre razporede likov).

Rezultati

Sistem točkovanja je bil tak kot pri off-line nalogi v letih 2007 in 2008. 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 9 in tretji 8. Na koncu smo za vsakega tekmovalca sešteli njegove točke po vseh tristo testnih primerih.

Enako off-line nalogo smo izvedli že na lanskem tekmovanju (leta 2013); takrat so v njej sodelovali samo trije tekmovalci, zato smo se odločili nalogo ponoviti tudi v letu 2014 (z novimi testnimi primeri). Žal smo letos dobili rešitve le od enega tekmovalca. Končna razvrstitev je tako naslednja:

Patrik Zajec (ZRI) 3000 točk

V upanju, da bo odziv prihodnje leto boljši, bomo enako nalogo (spet z novimi testnimi primeri) izvedli tudi v letu 2015. Zato bomo tudi razmislek o reševanju te naloge objavili v biltenu 2015.

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 14.–16. novembra 2014 v Krakowu), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bo 16.–21. maja 2015 v Marakešu v Maroku).

Na letošnjem UPM je sodelovalo 53 ekip s skupno 150 tekmovalci, ki so prišli z vseh 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, razen ene ekipe, ki je bila diskvalificirana, ker je poskušala na finalnem krogu oddajati zlonamerno kodo.

	Ekipa	Št. rešenih nalog*	Čas
1	Patrik Zajec (ŠC Srečka Kosovela Sežana / FRI + FMF), Matej Aleksandrov (FMF), Vid Kocijan (Gim. Vič / FRI + FMF)	18	25:01:53
2	Jure Slak, Maks Kolman, Žiga Gosar (FMF)	15	15:33:34
3	Sven Cerk (FRI), Veno Mramor (FMF), Martin Šušterič (FRI)	14	14:07:27
4	Ernest Beličič, Jure Kolenko, Milutin Spasić (FRI)	14	25:49:56
5	Alexei Drake, Andraž Dobnikar (FRI + FMF), Tibor Djurica Potpara (FMF)	13	18:24:02
6	Aleksandar Todorović, Marko Tavčar (FAMNIT)	12	18:35:57
7	Rok Poje, Žan Kustrle, Jan Živkovič (FRI)	12	21:57:35
8	Blaž Sobočan, Erik Grabljevec, Sara Pohl (FMF)	12	25:10:41
9	Matej Petković, Luka Černe, Tomaž Stepišnik Perdih (FMF)	11	15:10:23
10	Filip Koprivec, Filip Peter Lebar, Tina Lekše (Gim. Vič)	11	21:55:51
11	Tadej Jagodnik, Primož Kariž, Tomaž Kariž (FRI)	10	14:38:15
12	Primož Godec, Manca Žerovnik, Luka Krsnik (FRI)	10	16:57:33
13	Rok Fortuna, Matej Pelko, Urban Marovt (FRI)	10	17:43:20
14	Petra Hvala, Miha Eleršič, Fedja Beader (FRI)	9	15:40:15
15	Jasna Urbančič, Jure Brence, Uroš Hekić (FMF)	8	15:19:39
16	Luka Toni, Aleš Omerzel, Domen Urh (FRI)	7	9:11:16
17	Žiga Emeršič (FRI), Jure Senegačnik (F. za strojništvo, Lj.), Rok Bajec (FRI)	7	13:10:00
18	Dragana Božović, Martin Duh, Gregor Pirš (FNM Maribor)	7	15:12:27
19	Matjaž Kavčič, Manja Kocet, Andrej Dolenc (FERI)	6	11:12:53
20	Vladan Jovičić, Marko Palangetić, Roman Solodukhin (FAMNIT)	5	8:31:14
21	Sandi Mikuš (FRI), Jan Vatovec	5	9:11:09
22	Anja Petković, Vesna Iršič, Žiga Lukšič (FMF)	5	9:18:06
23	Tadej Novak, Mitja Rozman (FMF), Gregor Mubi (FRI+ FMF)	5	9:43:12
24	Matevž Poljanc, Matej Tomc, Rok Lekše (Škof. klas. gim. Lj.)	5	10:23:38
25	Ožbolt Menegatti (FE), Jani Bevk, Tjaž Brelih (FRI)	5	11:26:31
26	Marko Novak (FRI), Aljaž Jelen (FE), Lenart Bezek (FRI)	4	5:15:58
27	Blaž Šnuderl, Gregor Panič (FERI)	4	5:32:54
28	Rok Bezlaj (FMF), Urban Leben (FRI), Samo Mikuš (FMF)	4	5:55:29
29	Žiga Šmelcer, Žiga Gradišar, Lojze Žust (Škof. klas. gim. Lj.)	4	7:33:37
30	Bor Brecelj, Luka Kolar, Rok Kos (Gim. Vič)	4	7:49:36

* 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
31	Jure Taslak, Matija Skala, Aleš Zavec (FRI)	4	10:38:31
32	Andraž Krašovec, Luka Krhlikar, Rok Založnik (FRI)	4	15:37:06
33	Nejc Lovrenčič, Jan Pomer, Dejan Skledar (FERI)	4	24:15:27
34	Benjamin Novak, Maks Vrščaj, Rok Lampret (FRI)	3	7:00:18
35	Sebastijan Kužner, Patrik Kokol, Klemen Forstnerič (FERI)	3	7:06:06
36	Matej Brlec, Tomaž Sabadin (FAMNIT)	3	7:35:24
37	Luka Hrvatin, Jan Koštric, Patrik Širok (FAMNIT)	3	7:48:50
38	Rok Novosel, Žan Pevec, Simon Prešern (FRI)	3	8:39:08
39	Izak Pucko, Robert Koprivnik (FERI)	3	9:41:33
40	Rok Mohar, Sašo Marić, Rok Ljalić (FRI)	3	19:10:19
41	Igor Lalić, Janez Majdič (FRI)	2	3:27:39
42	Marko Prelevikj, Naum Gjorgjeski, Andrejaana Andova (FRI)	2	4:22:19
43	Urban Lavbič (FERI)	2	6:46:51
44	Kristjan Voje, Tine Šubic, Aleksander Tomič (FRI)	2	7:39:58
45	Nejc Galof, Gregor Menih, Aljaž Prislan (FERI)	2	9:06:46
46	Matej Kramberger, Jure Zgorelec, Boštjan Budna (FERI)	1	0:01:04
47	Tadej Ciglarič (FRI), Erik Langerholc (FMF), Jaka Konda (FRI)	1	1:25:36
48	Nejc Smrkolj Koželj, Žiga Simončič (Vegova Ljubljana)	1	2:28:43
49	Borut Budna, Urša Nered, Marcel Salmič (FMF)	1	2:34:21
50	Alen Rajšp, Patrik Rek, Aleš Pečovnik (FERI)	1	4:15:10
51	Klemen Kogovšek, Samo Remec, Tobias Mihelčič (Vegova Lj.)	0	0:00:00
	Alen Ajanović, Pija Balaban, Matic Podpadec (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 sta nastopili ekipi 1 in 2 kot predstavnici Univerze v Ljubljani, malo spremenjena ekipa 18 kot predstavnica Univerze v Mariboru in kombinacija ekip 6 + 20 kot predstavnica Univerze na Primorskem. V konkurenci 79 ekip s 33 univerz iz 7 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
22	Matej Aleksandrov, Vid Kocijan, Patrik Zajec	5	10:19
29	Žiga Gosar, Maks Kolman, Jure Slak	4	4:37
54	Vladan Jovičić, Marko Palangetić, Aleksandar Todorović	3	6:33
65	Jernej Borlinić, Dragana Božović, Martin Duh	2	7:23

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. 155–162.

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, 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, 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# in javi. 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 84 tekmovalcev prve skupine, 13 tekmovalcev druge skupine in 13 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. 156. 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 malo lažje kot prejšnja leta. 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: 3,40 v prvi skupini (v prejšnjih letih 3,28, 3,39, 3,56, 3,34, 3,56), 3,44 v drugi skupini (prejšnja leta 3,35, 3,50, 3,39, 3,38, 3,46) in 3,19 v tretji skupini (prejšnja leta 3,40, 3,21, 3,57, 3,92).

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 včasih šibka negativna korelacija, letos pa je praktično ni bilo ($R^2 = 0{,}14$; v prejšnjih letih 0,52, 0,21, 0,11, pred tem okoli 0,4).

Največ pripomb o tem, kako da je naloga težka, je bilo pri nalogah 1.3 (pacifistični generali), 1.5 (davek na ograjo) in 3.3 (luči). Pri slednji je težavnost mogoče posledica tega, da je z njo precej dela in da tekmovalci niso navajeni razmišljati o grafih. Pri 1.3 je težavnost mogoče povezana s tem, da je nalogo težko opisati na res preprost in razumljiv način (pri tej nalogi je bilo tudi veliko pripomb, češ da je besedilo težko razumljivo).

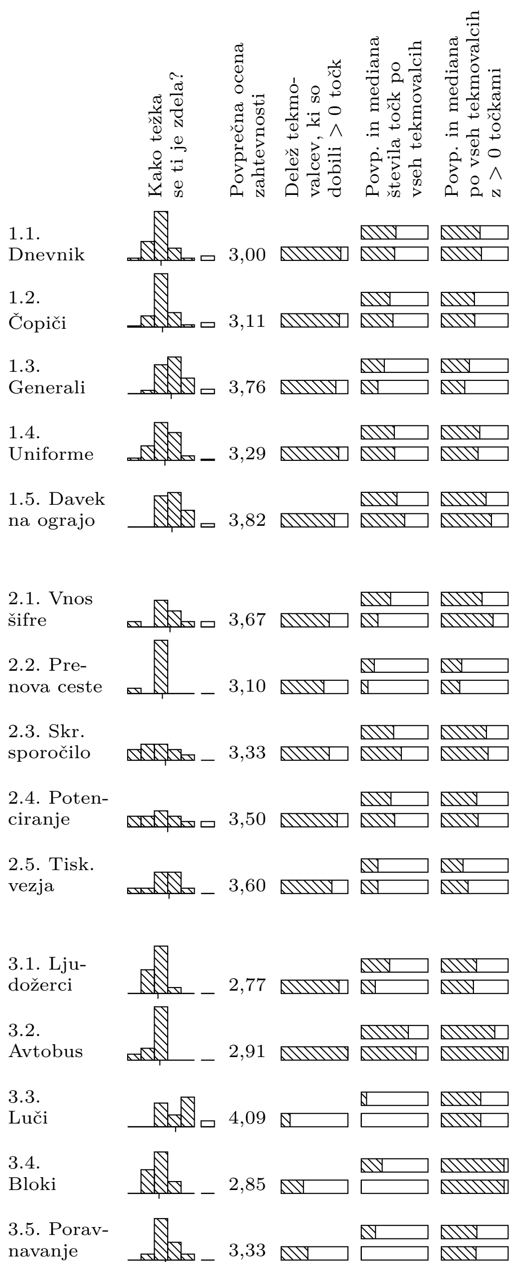
Kot najlažjo so tekmovalci ocenili nalogo 1.1 (dnevnik), pri kateri je bilo celo tudi nekaj pripomb, da je prelahka. Pri rezultatih sicer ni videti, da bi jo reševali občutno bolje kot druge naloge. V tretji skupini so kot najlažjo ocenili nalogo 3.1 (ljudožerci).

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 157. Nad razumljivostjo besedil ni veliko pripomb (podobno kot prejšnja leta); kot težko razumljivi izstopata predvsem nalogi 1.3 (generali) in 2.4 (potenciranje). Pri generalih je problem, ki se skriva v nalogi, nekoliko abstrakten in ga je težko enostavno razložiti; pri potenciranju pa je težava najbrž v tem, da naloga sili reševalca delati v nekem izmišljenem zbirnem jeziku, ki ga ni navajen.

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, približno enako kot v prejšnjih letih. Še največ pripomb na dolžino je pri nalogah 1.3 (generali) in 3.5 (poravnavanje desnega roba), ki sta se nekaterim zdeli predolgi.

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta, le v tretji skupini malo nižje. Kot bolj dolgočasna izstopa naloga 1.3 (generali), 2.4 (potenciranje) in 3.3 (luči), kot bolj zanimiva pa naloga 2.3 (skrivno sporočilo).

Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk



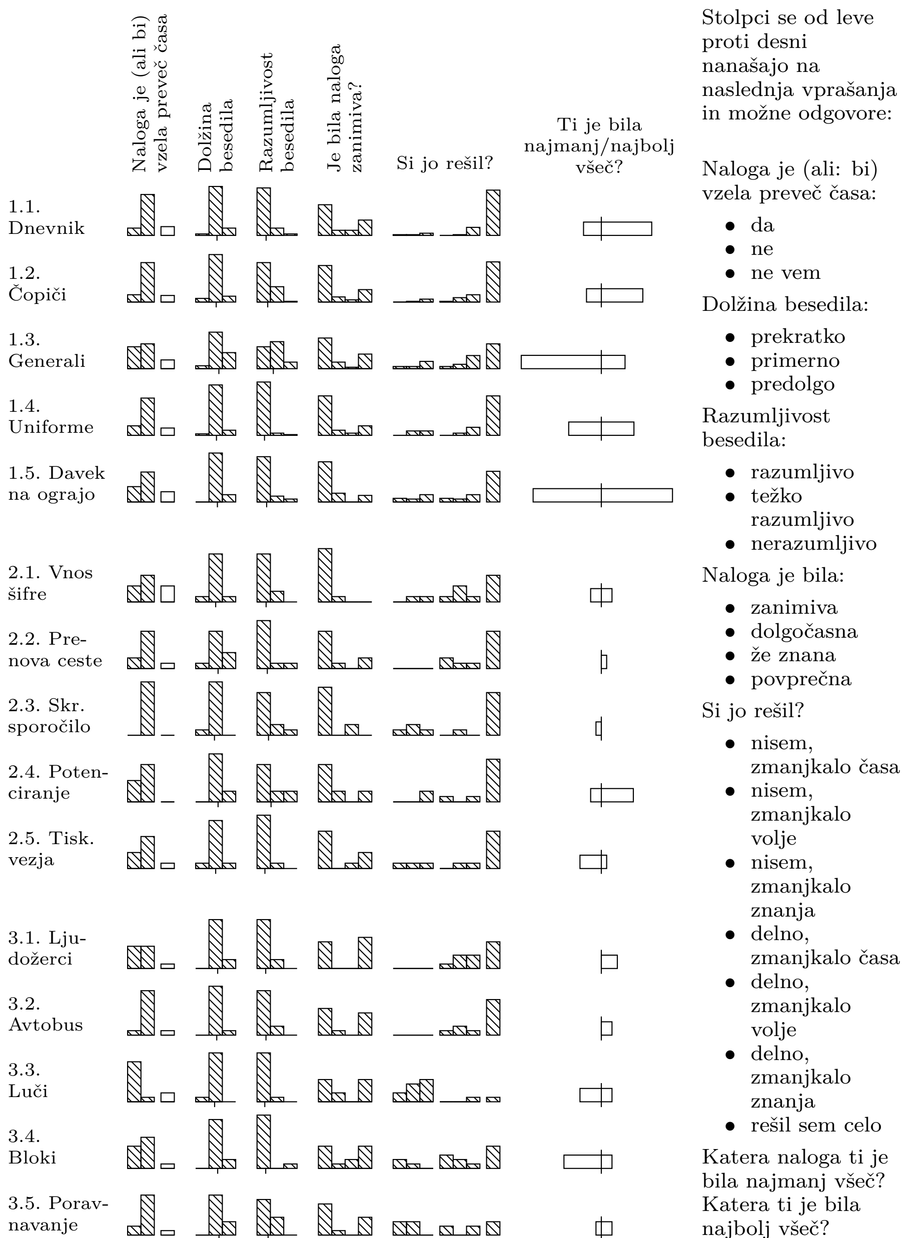
Pomen stolpcev v vsaki vrstici:

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.



Pripomb, da bi naloga vzela preveč časa, je bilo podobno kot lani in manj kot v prejšnjih letih. Največ takih pripomb je bilo pri nalogah 1.3 (generali; s to sicer ni veliko dela, ko jo enkrat razumemo — jo je pa verjetno težje razumeti kot ostale naloge v prvi skupini) in 3.3 (luči; ta je bila res bolj zamudna za reševanje).

Pri glasovih o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, je kot bolj popularna izstopala naloga 2.4 (potenciranje), kot nepopularne pa naloge 1.3 (generali; ker se jim je zdela pretežka in nerazumljiva), 2.5 (tiskana vezja) in 3.4 (bloki). Zanimiv primer je še naloga 1.5 (davek na ograjo), ki je dobila veliko glasov v v obeh kategorijah (nekaterim je bila všeč, ker jim je bila lep izziv; nekaterim pa se je zdela pretežka in jim ni bila všeč).

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.

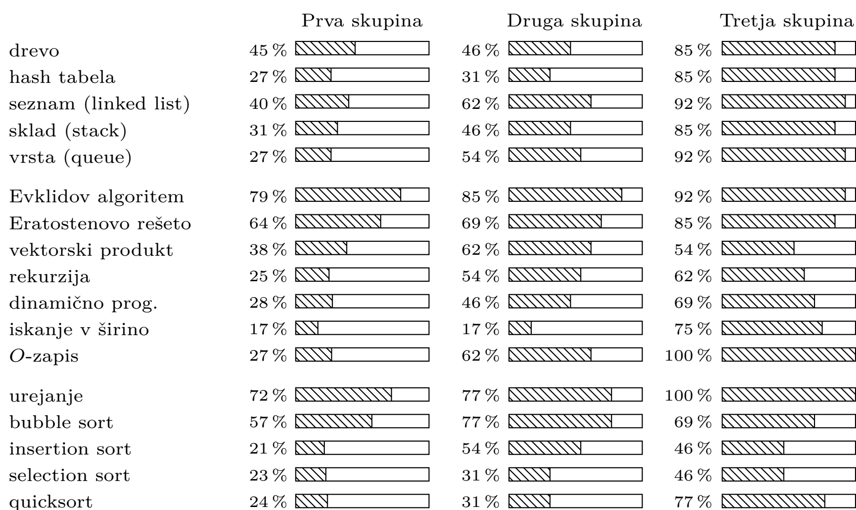
	Prva skupina	Druga skupina	Tretja skupina
priority_queue v C++ ipd.	8 %	31 %	54 %
map v C++ ipd.	13 %	15 %	62 %
unordered_map v C++ ipd.	15 %	38 %	54 %
zamikanje s shl, shr	17 %	46 %	62 %
operatorji na bitih	42 %	46 %	77 %
strukture	40 %	54 %	85 %
naštevni tipi	28 %	46 %	69 %
gnezdenje zank	85 %	92 %	92 %
zanka while	96 %	92 %	92 %
zanka for	95 %	100 %	92 %
kazalci	25 %	31 %	77 %
rekurzija	32 %	58 %	69 %
podprogrami	81 %	85 %	85 %
več-d tabele (array)	46 %	62 %	85 %
2-d tabele (array)	63 %	77 %	92 %
1-d tabele (array)	76 %	92 %	92 %
delo z datotekami	63 %	77 %	92 %
std. vhod/izhod	75 %	92 %	92 %

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. Stvari, ki jih tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja leta: rekurzija, kazalci, naštevni tipi in operatorji na bitih, v prvi skupini tudi strukture.

Uporaba programskih jezikov

Največ tekmovalcev tudi letos uporablja C/C++ (podobno kot zadnja leta je čisti C razmeroma redek, je pa letos presenetljivo pogost v drugi skupini), je pa njegova prednost pred drugimi jeziki manjša. V prvi skupini je letos najpogostejši jezik java,



	Prva skupina	Druga skupina	Tretja skupina
drevo	45 %	46 %	85 %
hash tabela	27 %	31 %	85 %
seznam (linked list)	40 %	62 %	92 %
sklad (stack)	31 %	46 %	85 %
vrsta (queue)	27 %	54 %	92 %
Evklidov algoritem	79 %	85 %	92 %
Eratostenovo rešeto	64 %	69 %	85 %
vektorski produkt	38 %	62 %	54 %
rekurzija	25 %	54 %	62 %
dinamično prog.	28 %	46 %	69 %
iskanje v širino	17 %	17 %	75 %
O-zapis	27 %	62 %	100 %
urejanje	72 %	77 %	100 %
bubble sort	57 %	77 %	69 %
insertion sort	21 %	54 %	46 %
selection sort	23 %	31 %	46 %
quicksort	24 %	31 %	77 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

---

z nekaj zaostanka ji sledi C++, malo manj ljudi je uporabljalo python in še malo manj C#. V drugi skupini sta najpogostejša C in python, java in C# pa sta redka. V tretji skupini je C++ daleč najpogotejši. Pascal je zelo redek, tako kot že zadnjih nekaj let.

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`“). Letos smo za poskus pri nalogah tipa „opiši postopek“ pripisali „ali napiši podprogram (kar ti je lažje)“, vendar ni očitno, da bi to kaj dosti pomagalo.

Podobno kot v prejšnjih letih je v anketi precej tekmovalcev napisalo, da dobro poznajo tudi PHP, vendar sta ga na tekmovanju uporabljala le dva. Na temo rešitev v eksotičnih jezikih lahko omenimo še, da je en tekmovalec reševal v javascriptu, eden je oddal sploh ne slabe rešitve v jeziku paketnih datotek za Windowse (vključno z `@echo off` na začetku `:)`), nepričakovan preporod pa so letos doživeli diagrami poteka, ki so jih risali kar trije tekmovalci (vsi v prvi skupini).

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže tabela na str. 160. Glede štetja C in C++ v tej tabeli je treba pripomniti, da je razlika med 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

	Leto in skupina																	
	2014			2013			2012			2011			2010			2009		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal	$2\frac{1}{2}$	2	1	1		1	6	1	4	3	4	3	$4\frac{1}{2}$	5	2	4	2	1
C	$3\frac{1}{2}$	6		2	7		7	2	1	7	2		6	6	1	$9\frac{1}{2}$	$3\frac{1}{2}$	$\frac{1}{2}$
C++	19	$4\frac{1}{2}$	$10\frac{1}{2}$	17	$12\frac{1}{2}$	7	26	16	9	$23\frac{1}{2}$	19	8	33	$17\frac{1}{2}$	13	$26\frac{1}{2}$	2	$12\frac{1}{2}$
java	23	2	$1\frac{1}{2}$	12	8	1	17	$6\frac{1}{2}$	1	6	5	3	5	9	4	8	8	11
PHP	2		—	1	$\frac{1}{2}$	—		1	—	$\frac{1}{2}$		—	1	1	—	2	1	—
basic		1	—	1		—			—			—			—			—
C#	12	$1\frac{1}{2}$	2	18	$\frac{1}{2}$		17	1	3	4	2	3		$\frac{1}{2}$	1			
python	16	6	—	16	8	—	25	5	—	20	6	—	12	2	—	4	$\frac{1}{2}$	—
NewtonScript			—		$\frac{1}{2}$	—		$\frac{1}{2}$	—			—			—			—
javascript	1		—			—			—			—			—			—
batch	1		—			—			—			—			—			—
psevdokoda	10		—	6		—	3		—	6		—	4		—	8		—
nič	4	2		2			2			1	1		1	5		1		

Število tekmovalcev, ki so uporabljali posamezni programski jezik. Nekateri uporabljajo po dva različna jezika (pri različnih nalogah) in se štejejo polovično k vsakemu jeziku. „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“; pred letom 2009 takih nismo šteli posebej in če je kdo uporabljal le psevdokodo, je štet pod „nič“.

uporablja `string` namesto **char** * in tip `vector` namesto tradicionalnih tabel (*arrays*).

Pri pythonu letos prvič več ljudi uporablja python 3 kot python 2; je pa res, da je pri tako preprostih programih, s kakršnimi se srečujemo na našem tekmovanju (še posebej v prvi skupini, kjer je največ uporabnikov pythona), 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. Ta nabor jezikov očitno kar dobro pokrije znanje naših tekmovalcev, saj je delež tekmovalcev, ki pravijo, da deklaracije razumejo, visok (72/78 v prvi skupini in 11/13 v drugi). 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 pascal; edina originalna predloga sta bila lua in javascript. Tako ali tako 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 zadovoljna (42/80 v prvi skupini, 8/13 v drugi, 13/13 v tretji; to je približno enak delež kot lani, le v drugi skupini je nižji). Med jeziki, ki bi jih radi videli namesto (ali poleg) C-ja, jih največ omenja javo in python, malo manj je glasov za C++, v prvi skupini 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 prvi skupini je povprečja starost malo nižja kot lani, v tretji pa malo višja. V prvi skupini so nastopili tudi trije osnovnošolci; teh pri izračunu povprečnega letnika v spodnji tabeli nismo upoštevali.

Skupina	Št. tekmovalcev po letnikih OŠ	1	2	3	4	Povprečni letnik
prva	3	14	24	32	21	2,7
druga		1	4	12	8	3,1
tretja		1	2	1	11	3,5

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.

Pri vprašanju, kje so se naučili programirati, je podobno kot prejšnja leta najpogostejši odgovor, da so se naučili programirati sami; sledijo tisti, ki so se naučili programirati v šoli, še malo manj (vendar več kot prejšnja leta) 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: nekateri želijo več nalog, drugi manj, podobno tudi pri času. V tretji skupini letos izstopajo želje, da bi imeli več časa (pri enakem številu nalog); s tem načeloma ne bi bilo nič narobe, bi se pa potem reševanje nalog v tretji skupini že neugodno zajedlo v popoldanski del programa.

	Kje si izvedel za tekmovanje				Kje si se naučil programirati					Čas reševanja			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	76	0	5	3	55	34	21	9	5	7	10	64	6	10	64	27	27	23	33	33	18	36	48
II	11	2	2	4	8	5	5	3	3	3	0	8	2	1	7	3	5	5	2	6	2	4	5
III	11	0	1	1	9	2	3	4	2	5	2	4	1	1	9	2	2	4	2	5	6	5	5

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. Letos je bila najpogostejša težava pri oddaji odgovorov na računalniku ta, da je sistem zavrgel tabulatorje na začetkih vrstic in s tem porušil zamike v oddani izvorni kodi (to je otežilo delo tudi ocenjevalcem, še posebej neugodno pa je pri pythonu, kjer je zamik vrstice pravzaprav del sintakse jezika).

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 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) Neobičajno dolg prazen niz:

```
while b != '  ': # Zanka teče, dokler PreberiDogodek() ne izpiše praznega niza.
```

(1.1) Nekdo je zjutraj vzel preveč tavtoloških tablet...

```
// this should kinda work... or not...
```

(1.1) Eden od razlogov, zakaj smo pri tej nalogi podali funkcijo PreberiDogodek, je bil tudi ta, da bi odvrnili tekmovalce od tega, da si celo vhodno datoteko naenkrat preberejo v pomnilnik. Mnogi so to mirno zaobšli, tale je celo predpostavil, da je vrstic največ 100:

```
char n[100][101], s; // nizi
for (k = 0; (s = PreberiDogodek()) != ""; k++) // branje v tabelo
  strcpy(n[i], s);
```

(1.1) Za ljubitelje messy kodiranja:

> pridobi dogodek na katerega kaže spremenljivka st_vrstic s funkcijo PridobiDogodek in ga shrani v spermenljivko dogodek

(1.1) Vsako leto imamo kakšen primer, ko ljudje opisujejo izvorno kodo programa, namesto da bi opisali postopek. Letošnjo nagrado za tok zavesti dobi:

> Če sta A in B enaki, vstopimo v **if** stavek. V **if** stavku priredimo spremenljivko Števec_ponovitev na 2. V **if** stavku uporabimo funkcijo PreberiDogodek() in sprejeto vrednost primerjamo z A ali B, če sta enaki, prištejemo spremenljivki Števec_ponovitev +1 in vstopimo v **while** zanko, katere logični pogoj je preverjanje nove prebrane vrednosti z A ali B && C.

in tako naprej.

(1.2) Navdušen komentar iz ene od rešitev:

```
if (tmpRoc * r + tmpRoc * k < skupnaDolzina)
begin
  // Krasno smo pa res lahko srečni
  izpis "Št. vegi čopičev: " + tmpRoc;
end
```

(1.2) Rešitev z upoštevanjem fizikalne realnosti:

> poskrbimo, da so vse dolžine cela števila (po potrebi vsa pomnožimo s potenco števila 10 (1000 ali 10 000 — več (če so to dejanski čopiči) ne potrebujemo, saj je natančnost večja od 1/10 mm neuporabna; po potrebi lahko množimo tudi z večjimi števili)
> [iz navodil nisem uspel jasno razbrati, če so vse vrednosti cela števila — previdnost ni nikoli odveč]

Očitno mu fraza „tudi dolžine paličk so naravna števila“ v besedilu naloge ni bila dovolj jasna...

(1.2) Ena od najbolj posrečenih tipkarskih napak letos:

> n paličk enake debeline,
> ⋮
> r cent lesa 1 kos
> k cent ene ali več palačnik zmletih

Ni sicer očitno, zakaj bi človek mlel palačinke :)

(1.2) En tekmovalec števcu dosledno pravi števnik:

> — iz te paličice sestavi kolikor je možno čopičev in to zapiše v nek števnik (i)
> ⋮
> — izpiše števnik i (št. čopičev)

(1.2) Rešitev z odporom do stavka **break**:

```
bool notsurehowbreakworks = true; // cause I don't really use break
while (notsurehowbreakworks)
{
   if ((totallength − ((maxpossible * min) + (maxpossible * k)) > 0)
   {
      notsurehowbreakworks = false;
   }
   maxpossible−−;
}
```

(1.2) Tale tekmovalec precej dosledno piše „konjice“ namesto „konice“. Pomislil sem, da je mogoče doma iz Slovenskih Konjic, ampak najbrž ni, saj hodi na srednjo šolo v Domžale.

> Dobimo število konjic, ki jih lahko izdelamo. Število konjic, ki jih lahko izdelamo odštejemo od števila ročajev.

(1.2) Impresivno zapleten način za izražavo števila 1:

> Primerjaj vsako paličico (od n do $n - (n - 1)$) z r;

(1.3) Zaskrbljujoče navdušen komentar:

```
// če imamo na listi vse ključe
if (keys.size() == k) // imamo vse ključe !BUM!
```

(1.3) Iz komentarja na začetku rešitve:

```
// Zelo enostavna rešitev (Pacifističnim generalom pobereš ključe in jih razdeliš
// med ostale :)
```

Problem je, da vnaprej ne veš, kateri so pacifistični. Poleg tega, če ključe prerazporediš med manjše število generalov, se poveča tveganje za napako nasprotnega tipa: da majhna skupina zarotnikov zbere dovolj ključev in sproži bombo, četudi za to niso dobili ukaza.

(1.3) Rešitev z dobrimi nameni:

> Najprej bi prebral podatke o številu generalov, številu ključev in o ključih, ki jih ima posamezen general v lasti. Podatke bi nato lahko hranil v datoteki na primer tak zapis: [. . .]
>
> Nato bi s pomočjo teh podatkov še napisal metodo za izračun r-odpornosti.

Tu pa se njegova rešitev konča.

(1.3) Odlična tipkarska napaka:

> Program izbere enega generala iz skubine.

Bralec si ob tem lahko predstavlja generale kot nasršene ptiče, ki v neki luknji čakajo na skubljenje :)

(1.3) Komentar na začetku ene od rešitev:

```
// jedrsko orožje je nevarno zato ga ne smemo uporabljati!!!
```

(1.3) Rešitev z velikim zaupanjem v psihološke zmožnosti programa:

> Najprej mora program preveriti, koliko od n generalov ni pripravljenih uporabiti jedrskega orožja oz. celo preprečiti uporabo drugim. To lahko ugotoviš tako, da najprej preveri, ali je skupina r generalov v večini proti ali za uporabo jedrskega orožja. Če je v večini proti, preverimo kakšne so možnosti, da lahko pacifisti s svojimi ključi blokirajo ostale.

(To je celoten odgovor tega tekmovalca pri tej nalogi.)

(1.3) Prijetno starinski izrazi v nekaterih komentarjih:

> OPIS HRANITVE PODATKOV:

Isti tekmovalec pri peti nalogi:

> Slično pregledamo vsa polja.

(1.4) Eden od letošnjih prispevkov na temo nestandardnih razširitev operatorjev je tale zloraba vejice kot logičnega in:

```
if (k[1] = 1, k[2] = 1, k[3] = 1) i = i + 1;
```

(1.4) Zanimivo nekonsistenten komentar:

```
d = dict(d)          (datoteko spremeni v list)
```

(1.5) Tale rešitev poskuša določiti, koliko ograj bo treba okoli trenutnega polja, tako, da analizira vse možne kombinacije tega, katera od štirih sosednjih polj pripadajo istemu lastniku kot trenutno polje. Pri tem je sintakso operatorjev ==, || in && razširil do neslutenih razsežnosti:

```
if (Lastnik(x, y) == Lastnik(x − 1, y) == Lastnik(x, y − 1) == Lastnik(x + 1, y) ==
    Lastnik(x, y + 1)) ograja[Lastnik(x, y)] += 0;
if (Lastnik(x, y) == Lastnik(x − 1, y) || Lastnik(x, y − 1) || Lastnik(x + 1, y) ||
    Lastnik(x, y + 1)) ograja[Lastnik(x, y)] += 3;
if (Lastnik(x, y) == Lastnik(x − 1, y) == Lastnik(x, y − 1) || Lastnik(x + 1, y) ||
    Lastnik(x, y + 1)) ograja[Lastnik(x, y)] += 2;
if (Lastnik(x, y) == Lastnik(x − 1, y) == Lastnik(x, y − 1) == Lastnik(x + 1, y) ||
    Lastnik(x, y + 1)) ograja[Lastnik(x, y)] += 1;
if (Lastnik(x, y) != Lastnik(x − 1, y) && Lastnik(x, y − 1) && Lastnik(x + 1, y) &&
    Lastnik(x, y + 1)) ograja[Lastnik(x, y)] += 4;
```

V prvem stavku **if** imamo večmestni ==, precej podobno, kot bi se to počelo v matematiki in kot se lahko počne npr. v pythonu. V drugem stavku si je večmestni || očitno treba razlagati takole: pogoj „a == b || c || d || e“ je izpolnjen natanko tedaj, ko je a enak natanko enemu od b, c, d in e. V tretjem stavku si stvari ne moremo smiselno razlagati tako, da bi lahko pogoj res pokril vse tiste primere, pri katerih imata natanko dve sosednji polji istega lastnika kot naše; podobno je tudi v četrtem stavku. Peti stavek je spet lepši: pogoj „a != b && c && d && e“ je izpolnjen natanko tedaj, ko je a različen od vseh b, c, d in e.

(1.5) Rešitev z dramatičnim pogledom na nepremičninski davek:

```
int obubozanci[StLastnikov()];
for (int i = 0; i < StLastnikov(); i++)
   obubozanci[i] = 0; // Zaenkrat HAHAHA
```

(1.5) Tale rešitev ima tabelo, v kateri za vsakega lastnika hrani število ograj; nato gre enkrat čez celo mrežo in počasi povečuje števce v tej tabeli. To je vse lepo in prav, ampak to je tabela *nizov*, ki jih potem ob vsaki priliki pretvarja v cela števila in nazaj:

```
ArrayList<String> stOgraj = new ArrayList<String>();
for (int a = stLastnikov − 1; a >= 0; a−−) stOgraj.set(a, "0");
```

Pri vsaki celici mreže naredi nekaj takega:

```
ograje = Integer.parseInt(stOgraj.get(lastnik));
⋮
if (Lastnik(x_a, y) != lastnik) ograje++;
⋮
stOgraj.set(lastnik, ograje + "");
```

Na koncu bi človek pomislil, da je imel tabelo nizov (namesto celih števil) zato, ker mu bodo prišli prav vsaj pri izpisu. Toda ne:

```
for (int a = stLastnikov − 1; a >= 0; a−−) {
   System.out.println("Lastnik " + a + " ima " +
                     Integer.parseInt(stOgraj.get(a)) + " ograj.");
```

(1.5) Predrzen komentar na koncu ene od rešitev:

> Outputa ni, ker v navodilih ne piše, da mora biti.

Zakaj neki smo potem v besedilu naloge napisali „Napiši program, ki za vsakega lastnika izračuna in izpiše skupno število ograj“...

(1.5) Rešitev z nenavadno velikimi lopami:

```
FOR %%L IN (0,1,%stLastnikov) DO ( :: loop čez vseh lasnikov vsi možni
   FOR %%i IN (0,1,%x) DO (
      FOR %%j IN (0,1,%h) DO ( :: dve lopi čez cel sistem zemlje
```

(1.5) Čudovita kombinacija ograj in orgij:

```
cout << "lastnik";
cout << x;
cout << "ima";
cout << Lastniki[x];
cout << "orgraj";
```

(2.1) Pri tej nalogi je bila oddana tudi najdaljša letošnja rešitev, dolga kar 196 vrstic. Žal je bila ta rešitev močno zgrešena in ni zajemala kaj dosti drugega kot to, da izpiše eno od desetih konkretnih 6-mestnih šifer (to, katero, je odvisno od tega, kaj je vnesel uporabnik).

(2.2) Komentar na začetku ene od rešitev:

```
/* ta cesta bo totalna mineštra :) */
```

(2.2) Rešitev za ljubitelje represivnih organov:

> ULTIMATIVNA OPTIMIZACIJA: Namesto da država izgublja denar z anketiranjem, bi ga lahko bolj pametno porabila in zagotovila močan policijski nadzor, da se ljudje ne bi upirali. Podjetji pa bi si razdelili cesto na 50:50.

(2.2) Rešitev z oštevanjem prebivalcev:

> Ker bi to naneslo 10.000 prebivalcev bi to porazdelil malce drugače, naprimer da bi vsakemo 2 kilometro oštel 1 prebivalca in vsakemu sosednemu kilometru enega prištel.

(2.4) Naslednji tekmovalec je svojo rešitev zaključil z neskončno zanko:

```
end: JL 0, 1, end
```

(2.4) Tale tekmovalec je iznašel nekakšen pogojni MOD, resda pa so skoki vedno kazali na neposredno naslednjo vrstico, na primer takole:

```
      MOD b, trenutno, lab3
lab3: ADD c, 1
```

Podobno ima še dvakrat. Na koncu pa je še tale komentar, ki povzdigne MOD takorekoč v univerzalno operacijo:

```
// Prvi mod prišteje le a, da lahko drugi mod množi, dokler c ne doseže potence a,
// tretji mod preveri, ali je b nič, in priredi c vrednost 1.
```

(2.4) Zanimiv način za inicializacijo spremenljivke na 1:

```
DIV vs, vs // da dobimo v vs = 1
```

Očitno ga ni skrbelo, da bi bila njena začetna vrednost lahko 0.

(2.4) Rešitev s prevelikimi pričakovanji do spremenljivke a:

```
MUL a, a      // a je zdaj a^2
⋮
DIV a, a      // a^2 nastavim nazaj na a
```

(2.5) Naslednji tekmovalec je okoli funkcije SeSekata naredil svoj wrapper, ki najprej preveri, če imata daljici skupno kakšno krajišče, in v tem primeru ne obremenjuje funkcije SeSekata po nepotrebnem:

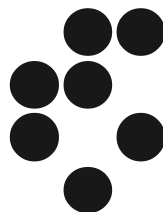
```
funkcija sekanje(i, j)
   če (from[i] == from[j] || from[i] == to[j] || to[i] == from[j] || to[i] == to[j])
      // če je ena od točk ista pol se zihr ne sekata
      return false
   sicer
      return SeSekata(x[from[i]], y[from[i]], x[to[i]], y[to[i]],
                      x[from[j]], y[from[j]], x[to[j]], y[to[j]])
```

(3.0) Posrečena rešitev poskusne naloge v C# z enim samim stavkom:

```
File.WriteAllText("poskus.out",
   File.ReadLines("poskus.in")
      .First()
      .Split(' ')
      .Select(x => int.Parse(x))
      .Sum().ToString());
```

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.

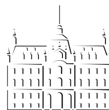
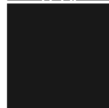
Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi 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.

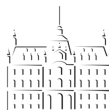
*

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. K temu bo fakulteta v prihodnjih letih (2009–2013) z razvojem kakovostnega oblikovanja in izvajanja naravoslovnih študijskih programov tudi stremela. 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

Association for Computing Machinery

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.

ZLATI POKROVITELJ

Družina rešitev PANTHEON
ki vam pomaga voditi posel!

Družino PANTHEON sestavlja 8 rešitev,
najdete pa nas v 12 evropskih državah.

PANTHEON™
Enterprise

PANTHEON™
Retail

PANTHEON™
Manufacturing

PANTHEON™
Specific Solutions

PANTHEON™
Farming

PANTHEON™
Public Service

PANTHEON™
Accounting

www.datalab.eu | (01) 252 89 50

datalab
Spremenite podatke v dobiček!

BRONASTA POKROVITELJA

Rtk