10. tekmovanje ACM v znanju računalništva

Institut Jožef Stefan, Ljubljana, 21. marca 2015

Bilten

Bilten 10. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2015

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Primož Gabrijelčič, Boris Gašperin, Tomaž Hočevar, Jurij Kodre, Mitja Lasič, Matjaž Leonardis, Mark Martinec, Jure Slak, Mitja Trampuš, Patrik Zajec, Janez Brank.

Tisk: Tiskarna knjigoveznica Radovljica, 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 — Katoložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

37.091.27:004(497.4)

TEKMOVANJE ACM v znanju računalništva (10 ; 2015 ; Ljubljana)
Bilten / 10. tekmovanje ACM v znanju računalništva, Ljubljana, 21. marca 2015 ; [avtorji nalog Nino Bašić … [et al.] ; uredil Janez Brank]. — Ljubljana : Institut Jožef Stefan, 2015

ISBN 978-961-264-061-3
1. Bašić, Nino 2. Brank, Janez, 1979–
282329344

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	22
Naloge za 3. skupino	26
Naloge šolskega tekmovanja	31
Neuporabljene naloge iz leta 2013	34
Rešitve za 1. skupino	45
Rešitve za 2. skupino	53
Rešitve za 3. skupino	65
Rešitve šolskega tekmovanja	77
Rešitve neuporabljenih nalog 2013	87
Nasveti za ocenjevanje in izvedbo šolskega tekmovanja	143
Rezultati	147
Nagrade	153
Šole in mentorji	154
Off-line naloga: Zlaganje likov	157
Univerzitetni programerski maraton	163
Anketa	166
Rezultati ankete	171
Cvetke	179
Sodelujoče inštitucije	187
Pokrovitelji	191

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. Da bi bilo tekmovanje pošteno tudi do morebitnih reševalcev na papir, pa so bili na računalnikih za prvo in drugo skupino le urejevalniki besedil, ne pa tudi razvojna orodja, prevajalniki in dokumentacija o programskih jezikih in knjižnicah.

V tretji skupini rešujejo vsi tekmovalci naloge na računalnikih, za kar imajo pet ur časa. Pri vsaki nalogi je treba napisati program, ki prebere podatke iz vhodne datoteke, izračuna nek rezultat in ga izpiše v izhodno datoteko. Programe se potem ocenjuje tako, da se jih na ocenjevalnem računalniku izvede na več testnih primerih, število točk pa je sorazmerno s tem, pri koliko testnih primerih je program izpisal pravilni rezultat. (Podrobnosti točkovanja v 3. skupini so opisane na strani 23.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C#, java in VB.NET.

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

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

Na začetku smo tekmovalcem razdelili tudi list z nekaj nasveti in navodili (str. 7–9 za 1. in 2. skupino, str. 22–25 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 157–162.

Podobno kot v zadnjih nekaj letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 23. januarja 2015. 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. 143–146). 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 306 tekmovalcev s 30 šol (dve od njih sta bili osnovni, ostale pa srednje).

NASVETI ZA 1. IN 2. SKUPINO

Nekatere naloge so tipa **napiši program** (ali **napiši podprogram**), nekatere pa tipa **opiši postopek**. Pri slednjih ti ni treba pisati programa ali podprograma v kakšnem konkretnem programskem jeziku, ampak lahko postopek opišeš tudi kako drugače: z besedami (v naravnem jeziku), psevdokodo (glej spodaj), diagramom poteka itd. Glavno je, da je tvoj opis dovolj natančen, jasen in razumljiv, tako da je iz njega razvidno, da si dejansko našel in razumel pot do rešitve naloge.

Psevdokodi pravijo včasih tudi strukturirani naravni jezik. Postopek opišemo v naravnem jeziku, vendar opis strukturiramo na podoben način kot pri programskih jezikih, tako da se jasno vidi strukturo vejitev, zank in drugih programskih elementov.

Primer opisa postopka v psevdokodi: recimo, da imamo zaporedje besed in bi ga radi razbili na več vrstic tako, da ne bo nobena vrstica preširoka.

```
naj bo trenutna vrstica prazen niz;
pregleduj besede po vrsti od prve do zadnje:
    če bi trenutna vrstica z dodano trenutno besedo (in presledkom
    pred njo) postala predolga,
        izpiši trenutno vrstico in jo potem postavi na prazen niz;
    dodaj trenutno besedo na konec trenutne vrstice;
če trenutna vrstica ni prazen niz, jo izpiši;
```

(Opomba: samo zato, ker je tu primer psevdokode, to še ne pomeni, da moraš tudi ti pisati svoje odgovore v psevdokodi.)

Če pa v okviru neke rešitve pišeš izvorno kodo programa ali podprograma, obvezno poleg te izvorne kode v nekaj stavkih opiši, kako deluje (oz. naj bi delovala) tvoja rešitev in na kakšni ideji temelji.

Pri ocenjevanju so vse naloge vredne enako število točk. Svoje odgovore dobro utemelji. Prizadevaj si predvsem, da bi bile tvoje rešitve pravilne, ob tem pa je zaželeno, da so tudi čim bolj učinkovite; take dobijo več točk kot manj učinkovite (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). Za manjše sintaktične napake se ne odbije veliko točk. Priporočljivo in zaželeno je, da so tvoje rešitve napisane pregledno in čitljivo. Če je na listih, ki jih oddajaš, več različic rešitve za kakšno nalogo, jasno označi, katera je tista, ki naj jo ocenjevalci upoštevajo.

Če naloga zahteva branje ali obdelavo vhodnih podatkov, lahko tvoja rešitev (če v nalogi ni drugače napisano) predpostavi, da v vhodnih podatkih ni napak (torej da je njihova vsebina in oblika skladna s tem, kar piše v nalogi).

Nekatere naloge zahtevajo branje podatkov s standardnega vhoda in pisanje na standardni izhod. Za pomoč je tu nekaj primerov programov, ki delajo s standardnim vhodom in izhodom:

- Program, ki prebere s standardnega vhoda dve števili in izpiše na standardni izhod njuno vsoto:

```
program BranjeStevil;
var i, j: integer;
begin
  ReadLn(i, j);
  WriteLn(i, ' + ', j, ' = ', i + j);
end. {BranjeStevil}
```

```
#include <stdio.h>
int main() {
  int i, j; scanf("%d %d", &i, &j);
  printf("%d + %d = %d\n", i, j, i + j);
  return 0;
}
```

- Program, ki bere s standardnega vhoda po vrsticah, jih šteje in prepisuje na standardni izhod, na koncu pa izpiše še skupno dolžino:

```
program BranjeVrstic;
var s: string; i, d: integer;
begin
  i := 0; d := 0;
  while not Eof do begin
    ReadLn(s);
    i := i + 1; d := d + Length(s);
    WriteLn(i, '. vrstica: "', s, '"');
  end; {while}
  WriteLn(i, ' vrstic, ', d, ' znakov.');
end. {BranjeVrstic}
```

```
#include <stdio.h>
#include <string.h>
int main() {
  char s[201]; int i = 0, d = 0;
  while (gets(s)) {
    i++; d += strlen(s);
    printf("%d. vrstica: \"%s\"\n", i, s);
  }
  printf("%d vrstic, %d znakov.\n", i, d);
  return 0;
}
```

Opomba: C-jevska različica gornjega programa predpostavlja, da ni nobena vrstica vhodnega besedila daljša od dvesto znakov. Funkciji gets se je v praksi bolje izogibati, ker pri njej nimamo zaščite pred primeri, ko je vrstica daljša od naše tabele s. Namesto gets bi bilo bolje uporabiti fgets; vendar pa za rešitev naših tekmovalnih nalog v prvi in drugi skupini zadošča tudi gets.

- Program, ki bere s standardnega vhoda po znakih, jih prepisuje na standardni izhod, na koncu pa izpiše še število prebranih znakov (ne vštevši znakov za konec vrstice):

```
program BranjeZnakov;
var i: integer; c: char;
begin
  i := 0;
  while not Eof do begin
    while not Eoln do
      begin Read(c); Write(c); i := i + 1 end;
    if not Eof then begin ReadLn; WriteLn end;
  end; {while}
  WriteLn('Skupaj ', i, ' znakov.');
end. {BranjeZnakov}
```

```
#include <stdio.h>

int main() {
  int i = 0, c;
  while ((c = getchar()) != EOF) {
    putchar(c); if (i != '\n') i++;
  }
  printf("Skupaj %d znakov.\n", i);
  return 0;
}
```

Še isti trije primeri v pythonu:

```
# Branje dveh števil in izpis vsote:
import sys

a, b = sys.stdin.readline().split()
a = int(a); b = int(b)
print "%d + %d = %d" % (a, b, a + b)

# Branje standardnega vhoda po vrsticah:
import sys

i = d = 0
```

```
for s in sys.stdin:
    s = s.rstrip('\n') # odrežemo znak za konec vrstice
    i += 1; d += len(s)
    print "%d. vrstica: \"%s\"" % (i, s)
print "%d vrstic, %d znakov." % (i, d)

# Branje standardnega vhoda znak po znak:
import sys

i = 0
while True:
    c = sys.stdin.read(1)
    if c == "": break # EOF
    sys.stdout.write(c)
    if c != '\n': i += 1
print "Skupaj %d znakov." % i
```

Še isti trije primeri v javi:

```
// Branje dveh števil in izpis vsote:
import java.io.*;
import java.util.Scanner;

public class Primer1
{
    public static void main(String[] args) throws IOException
    {
        Scanner fi = new Scanner(System.in);
        int i = fi.nextInt(); int j = fi.nextInt();
        System.out.println(i + " + " + j + " = " + (i + j));
    }
}

// Branje standardnega vhoda po vrsticah:
import java.io.*;

public class Primer2
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader fi = new BufferedReader(new InputStreamReader(System.in));
        int i = 0, d = 0; String s;
        while ((s = fi.readLine()) != null) {
            i++; d += s.length();
            System.out.println(i + ". vrstica: \"" + s + "\""); }
        System.out.println(i + " vrstic, " + d + " znakov.");
    }
}

// Branje standardnega vhoda znak po znak:
import java.io.*;

public class Primer3
{
    public static void main(String[] args) throws IOException
    {
        InputStreamReader fi = new InputStreamReader(System.in);
        int i = 0, c;
        while ((c = fi.read()) >= 0) {
            System.out.print((char) c); if (c != '\n' && c != '\r') i++; }
        System.out.println("Skupaj " + i + " znakov.");
    }
}
```

NALOGE ZA PRVO SKUPINO

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

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

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. Delni izid

Na košarkarskih tekmah ekipe običajno zelo nihajo v nivoju igre. Tako pride do zanimivih delnih izidov v prid eni ali drugi ekipi, ki jih novinarji vestno analizirajo in sporočajo. *Delni izid* je razlika med točkami, ki sta jih v nekem časovnem intervalu dosegli ekipi.

Opiši postopek (ali napiši program, če ti je lažje), ki izračuna največji delni izid (ne glede na to, katera od obeh ekip je pri tem delnem izidu dosegla več točk, katera pa manj). Tvoj postopek kot vhodne podatke dobi zaporedje celih števil, ki predstavljajo posamezne koše, dosežene v tekmi. Urejeni so po času (od začetka tekme proti koncu); koši, ki jih je dosegla ena ekipa, so predstavljeni s pozitivnimi števili, tisti, ki jih je dosegla druga ekipa, pa z negativnimi. Z branjem vhodnih podatkov se ti ni treba ukvarjati, pač pa predpostavi, da jih tvoj postopek oz. program dobi v neki primerni spremenljivki ali podatkovni strukturi.

Primer: če dobimo vhodno zaporedje

$$-2, -2, +2, +2, +3, +2, -2, +2, -2, -3, +2, -2, -2, -2, +2,$$

je največji možni delni izid 9. Dosežen je celo večkrat: pri podzaporedju $\langle +2, +2, +3, +2 \rangle$ je razlika 9 v prid prve ekipe, pri podzaporedju $\langle -2, -3, +2, -2, -2, -2 \rangle$ je razlika 9 v prid druge ekipe, obstaja pa še nekaj drugih podzaporedij, ki tudi dajo delni izid 9.

2. Kompresija

Merilna postaja meri višino Blejskega jezera in podatke po mobilni povezavi sporoča v meteorološki center. Meritve so pozitivna cela števila in se le počasi spreminjajo. Tipičen primer izmerjenih podatkov:

$$310, 310, 310, 310, 310, 311, 311, 311, 313, 313, 311.$$

Ker bi radi varčevali pri količini zakupljenega podatkovnega prometa, smo se odločili, da bomo vsak drugi ponovljeni podatek izpustili. Namesto zgornje serije meritev bi tako radi v center sporočili le:

$$310, 310, 310, 311, 311, 313, 311.$$

Kadar je zaporednih podatkov z isto vrednostjo liho mnogo, število sporočenih podatkov zaokrožimo navzgor. V zgornjem primeru se to zgodi pri meritvah 310 in 311.

Napiši program, ki se bo vrtel v neskončni zanki, bral podatke z merilne postaje in jih sporočal v center. Na voljo ima dva podprograma oz. funkciji:

- **int** Preberi();
  Funkcija počaka, da strojna oprema opravi meritev, ter vrne izmerjeno vrednost. Ta je vedno večja od 0.

- **void** Sporoci(**int** meritev);
  Podprogram sporoči meritev v center.

Program napiši tako, da bo izmerjene vrednosti kar najhitreje poslal dalje. (To pomeni, da ne boš dobil(a) vseh točk, če bo program čakal, da se bo vrednost meritve spremenila, ter šele nato sporočil polovico zapomnjenih vrednosti.)

Še deklaracije v drugih jezikih:

```
{ v pascalu }
function Preberi: integer;
procedure Sporoci(meritev: integer);

# v pythonu
def Preberi(): ...      # vrne int
def Sporoci(meritev): ...
```

3. znajdi.se

Razvili smo nov iskalnik *znajdi.se*, ki nam izpiše, kako pridemo iz točke A v točko B. Problem je, da nam iskalnik korake izpiše v pomešanem vrstnem redu. Tako se opis poti iz kraja G v kraj S glasi:

```
G S
- mimo O nadaljujemo skozi tri križišča, dokler ne pridemo do A
- pri B zavijemo desno, dokler ne dospemo do S
- iz A zavijemo v krožno križišče, od tam nadaljujemo naravnost do D
- na H zavijemo desno proti O
- od D sledimo modrim oznakam, dokler ne zagledamo B
- nato pri J peljemo naprej 1,2 km proti H
- iz G zavijemo levo proti J
```

Na srečo veljajo naslednje omejitve, ki nam bodo prišle prav pri urejanju navodil v pravi vrstni red:

- vsak korak poti (od enega kraja do naslednjega) je v svoji vrstici (ki je dolga največ 100 znakov) in v njej se ime kraja, pri katerem se ta korak začne, pojavi prej kot ime kraja, pri katerem se ta korak konča; to pa sta tudi edina dva kraja, ki sta v tej vrstici omenjena;
- ime vsakega kraja je ena sama velika črka angleške abecede (od `A` do `Z` — kot vidimo tudi v gornjem primeru);
- drugače se v navodilih velike črke ne pojavljajo (vsi drugi znaki so male črke, števke, ločila in presledki);
- imena vseh krajev so med seboj različna in v nobenega ne gremo več kot enkrat;
- pot se gotovo ne konča v istem kraju, v katerem se začne;
- vsaka vrstica (razen prve, ki vsebuje začetni in končni kraj) predstavlja en korak poti (v vhodnih podatkih torej ni kakšnih odvečnih vrstic, ki ne bi bile del iskane poti).

Napiši program, ki bo prebral začetni in končni kraj ter zmešana navodila. Tvoja naloga je, da izpišeš kraje na poti v pravilnem vrstnem redu. Če torej tvoj program prebere zgornja navodila in podatek, da želiš priti iz G v S, mora izpisati:

```
GJHOADBS
```

Tvoj program lahko podatke bere s standardnega vhoda ali pa iz datoteke `pot.txt`, karkoli ti je lažje.

4. Dva od petih

Leta 1958 je podjetje IBM dalo na trg računalnik IBM 7070, ki je nadomestil predhodne modele računalnikov z elektronkami, saj je bil izdelan s tranzistorji in zato zmogljivejši in energijsko manj potraten.

Ker je bil model IBM 7070 namenjen poslovnim obdelavam, so bila števila v njem shranjena kot deset desetiških števk, vsaka števka pa zapisana s petimi biti, skupaj torej 50 bitov (in dodatni predznak, s katerim pa se v tej nalogi ne bomo ukvarjali).

Čeprav bi za zapis števke med 0 in 9 zadoščali štirje biti, so se izdelovalci zavedali možnosti napak pri hranjenju in obdelavi podatkov, zato so se odločili za pet bitov in izmed vseh možnih 32 kombinacij izbrali deset takih, pri katerih velja, da ima vsaka veljavna desetiška števka natanko dva bita od petih postavljena na 1, ostali trije pa morajo biti 0. Če se je med obdelavo kje pojavila nedovoljena kombinacija bitov, je bila javljena napaka.

Tako kodiranje omogoča, da zaznamo vsako posamično napako (sprememba enega bita iz 1 v 0 ali obratno), lahko pa celo več napak, če so vse iste vrste (vse iz 0 v 1 ali pa vse iz 1 v 0).

Tole je tabela, po kateri se pri IBM 7070 števila med 0 in 9 zakodirajo v petbitno kodo:

1	11000
2	10100
3	10010
4	01010
5	00110
6	10001
7	01001
8	00101
9	00011
0	01100

Napiši program, ki bo prebral eno vrstico z vhodne datoteke ali standardnega vhoda (kar ti je lažje), v kateri se nahaja zapis enega desetmestnega kodiranega števila. Vrstica vsebuje 50 takih znakov, ki so enice ali ničle (poleg njih so lahko v vrstici še drugi znaki, na primer presledki, vendar vse take druge znake zanemarimo; vsega skupaj pa je vrstica dolga največ 100 znakov), in predstavlja deset desetiških števk. Program naj izpiše prebrano število s števkami med „0“ in „9“, morebitne neveljavne števke v številu pa naj izpiše kot zvezdice.

Primer vhodnih podatkov:

```
01a100 0110000110, 10100  10110  x 010 10 00110;;;;10001 00000 00011
```

Pri tem primeru je rezultat:

```
0052*456*9
```

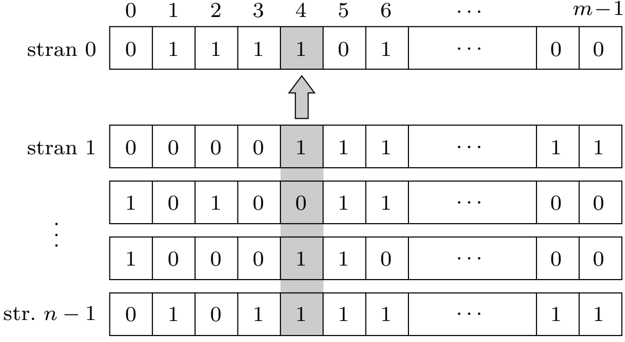
5. Kontrolne vsote

V računalniku imamo pomnilnik razdeljen na n strani, vsako pa sestavlja m pomnilniških celic; posamezna celica hrani en bit podatkov (torej vrednost 0 ali 1). Na voljo sta dve funkciji, s katerima dostopamo do podatkov v tem pomnilniku:

- **int** Beri(**int** stran, **int** naslov) — prebere podatek iz celice z danim naslovom (naslov je število od 0 do $m - 1$) iz dane strani (številke strani so od 0 do $n - 1$); vrne vrednost 0 ali 1, če pa podatka ni bilo mogoče prebrati, vrne -1.

- **void** Pisi(**int** stran, **int** naslov, **int** novaVrednost) — zapiše novo vrednost (ki mora biti 0 ali 1) v dano stran na dani naslov. Če pri pisanju pride do napake, se funkcija vseeno vrne, kot da se ni zgodilo nič neobičajnega.

Ker se posamezne pomnilniške celice včasih okvarijo, smo se odločili eno od strani nameniti za *kontrolne vsote*. Recimo, da bo to stran številka 0. Kontrolne vsote so definirane takole: v posamezni celici strani 0 mora biti vrednost 1, če je v istoležnih celicah na ostalih straneh liho mnogo enic, sicer pa vrednost 0. (Istoležne celice so tiste, ki imajo znotraj strani enak naslov.)



Primer za $n = 5$. S sivo barvo je označen stolpec celic na naslovu 4; puščica ponazarja, da je kontrolna vsota (na strani 0) določena z vsebino ostalih celic na tem naslovu. V tem konkretnem primeru so na teh ostalih celicah tri enice, kar je liho število, zato je kontrolna vsota enaka 1.

Napiši funkciji Beri2 in Pisi2, ki ju bo lahko uporabnik našega pomnilnika poklical za branje in pisanje podatkov, pri čemer bosta skrbeli za kontrolne vsote (za dejansko branje in pisanje podatkov v pomnilnik naj kličeta funkciji Beri in Pisi):

- **int** Beri2(**int** stran, **int** naslov) — številka strani je zdaj le od 1 do $n - 1$, predpostavimo torej lahko, da uporabnik ve, da iz strani 0 ne sme brati, ker je ta stran rezervirana za kontrolne vsote. Funkcija Beri2 naj prebere vrednost iz celice naslov strani stran, če pa to branje spodleti, naj pravo vrednost te celice rekonstruira (če je to mogoče) iz istoležnih celic na ostalih $n - 1$ straneh. Prebrano (oz. rekonstruirano) vrednost naj funkcija Beri2 vrne kot rezultat, če pa vrednosti ni mogla niti prebrati niti rekonstruirati, naj vrne -1.

- **void** Pisi2(**int** stran, **int** naslov, **int** novaVrednost) — zapiše naj novo vrednost (ki je 0 ali 1) v dano stran (od 1 do $n - 1$) na dani naslov (od 0 do $m - 1$) in ustrezno popravi kontrolno vsoto v istoležni celici na strani 0.

Opiši, kako se tvoja rešitev obnaša ob primeru raznih napak pri branju ali pisanju, za katere v tej nalogi obnašanje ni natančno predpisano. Predpostavi, da so na začetku delovanja našega programa v vseh celicah na vseh straneh shranjene ničle (kar med drugim pomeni, da so tudi vse kontrolne enote na strani 0 pravilne).

Še deklaracije v drugih jezikih:

```
{ v pascalu }
function Beri(stran, naslov: integer): integer; { in podobno za Beri2 }
procedure Pisi(stran, naslov, novaVrednost: integer); { in podobno za Pisi2 }
```

```
# v pythonu
def Beri(stran, naslov): ... # in podobno za Beri2
def Pisi(stran, naslov, novaVrednost): ... # in podobno za Pisi2
```

NALOGE ZA DRUGO SKUPINO

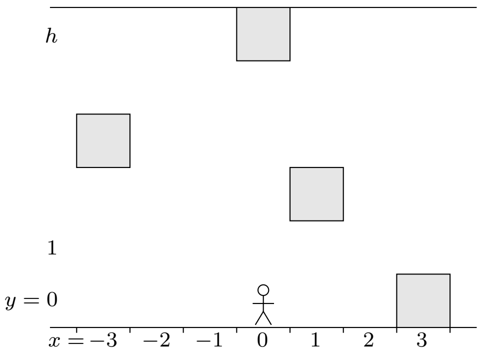
Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del prek računalnika. Vse te možnosti so enakovredne. Odgovore, oddane prek računalnika, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

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. It’s raining cubes

Si figurica v arkadni igrici, kjer s stropa padajo kocke. Vsako sekundo se nekje na stropu pojavi kocka, ki s stalno hitrostjo en kvadratek na sekundo pada proti tlom. Ti se lahko vsako sekundo premakneš en kvadratek levo ali desno. Tvoj cilj je, da se izogneš vsem kockam in preživiš; če te kakšna kocka uspe speštati pod sabo v bitne črepinje, pa si igro izgubil. Ko kocka pade na tla, tam ostane in ti blokira pot, tako da ne moreš mimo.



Skupno število kock označimo z n (lahko jih je veliko, $n \leq 10^7$); za vsak čas t od 0 do $n-1$ vemo, da kocka, ki začne padati ob času t, pada na x-koordinati x_t. Vse kocke začnejo padati z višine h (kocka, ki se pojavi ob času t na višini h, torej pristane na tleh ob času $t+h$; ob tem času ti torej ne moreš več varno stati na polju x_t ali se nanj premakniti). Tvoj začetni položaj (ob času $t=0$) je $x=0$.

Števila $n, h, x_0, x_1, \ldots, x_{n-1}$ so podana. **Opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki na podlagi teh podatkov ugotovi, ali lahko preživiš ali ne. Če lahko preživiš, opiši tudi potrebne premike. **Utemelji**, zakaj tvoj postopek deluje.

2. Strahopetni Hektor

Pojdimo v čas trojanske vojne, ko Ahajci napadajo Trojance, ki čepijo za svojim obzidjem. Zid je zgrajen iz n dolgih kamnitih blokov, visokih 1 enoto, ki so položeni vodoravno drug na drugega, tako da se ne porušijo. Na vsaki višini imamo le en blok; za blok na višini i (za $i = 1, 2, \ldots, n$) je znano, da se začne na x-koordinati z_i in je dolg d_i enot.

Ti si na strani Ahajcev in želiš z ogromnim katapultom porušiti trojansko obzidje; na voljo imaš več krogel, s katerimi lahko ustreliš v zid in razbiješ najbolj levo enoto izbranega bloka. Če uspeš popolnoma uničiti en blok, se zid podre in Hektor, poveljnik trojanske strani, se nemudoma vda (to se zgodi celo, če je uničen vrhnji blok). Vendar je dovolj razbiti kakšen blok tudi le toliko, da se del zidu nad njim prevrne. To se zgodi, ko njegovo težišče ni več podprto s spodnjim blokom. Tudi če se v tem primeru zid le nagne, Hektor in njegova vojska izgubijo vse upanje na zmago in se predajo.

Opiši postopek (ali napiši program, če ti je lažje), ki bo pomagal Ahajcem simulirati potek rušenja takega obzidja, da se bodo lažje pripravili na boj. Postopek na začetku dobi opis zidu (vrednosti n, $z_1, \ldots, z_n$ in $d_1, \ldots, d_n$), nato pa naj v zanki bere podatke o izstreljenih kroglah (za vsako kroglo je podano, v kateri blok je bila izstreljena — to je število od 1 do n, pri čemer 1 pomeni najnižji blok, n pa najvišjega) in po vsaki prebrani krogli takoj *sproti* (še preden prebere naslednjo kroglo) izpiše, ali zid zdaj še stoji ali je že podrt.[1]

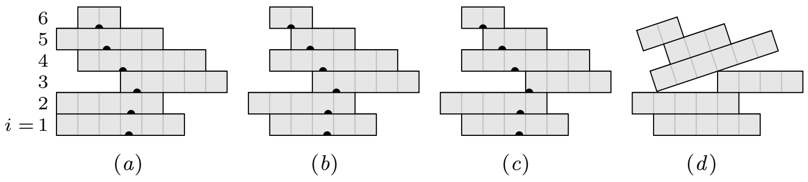
Fizikalni poduk: x-koordinata težišča je definirana kot povprečje x-koordinat posameznih enot zidu. Vse enote zidu so enako težke. Če se težišče nahaja točno nad robom podpore, se telo še ne prevrne.

Primer: recimo, da imamo zid višine $n = 6$ z naslednjimi začetnimi podatki:

i	1	2	3	4	5	6
z_i	0	0	3	1	0	1
d_i	6	5	5	6	5	2

Ahajci bi lahko zid (ne nujno optimalno) napadali takole:

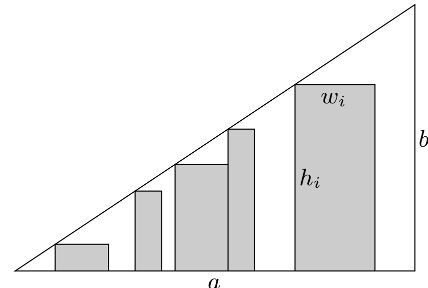
[1] Zelo zanimiva, vendar težja, je tudi naslednja različica naloge: za dani opis zidu ugotovi, koliko najmanj strelov potrebujemo (in v katere vrstice), da ga porušimo.



(a) Začetno stanje zidu. Črna pika na spodnjem robu posameznega bloka označuje položaj težišča tistega dela zidu, ki ga sestavljajo ta blok in vsi nad njim. — (b) Stanje zidu po treh strelih, enem na višini 1 in dveh na višini 5. Zid je še vedno stabilen. — (c) Če nato izvedemo še en strel na višini 3, zid ni več stabilen: blok 3 ne podpira več težišča blokov 4, 5, 6. Ti bi se zato nagnili in prevrnili, kot kaže slika (d).

3. Polaganje plošč

Imamo pravokotni trikotnik, širok a enot in visok b enot. Dobili smo tudi več pravokotnih plošč in poznamo njihove velikosti: i-ta plošča je široka w_i enot in visoka h_i enot. Plošče bi radi zložili v trikotnik tako, da bo spodnja stranica plošče ležala na spodnjem robu trikotnika (ki ima dolžino a; glej sliko spodaj), zgornje levo oglišče plošče pa bo ležalo na hipotenuzi trikotnika. Plošč ne smemo vrteti tako, da bi stranica w_i prišla po višini, h_i pa po širini. Poleg tega se plošče ne smejo prekrivati ali štrleti ven iz trikotnika, lahko pa se dotikajo. Naslednja slika kaže primer takega trikotnika, v katerega smo položili pet plošč:



Opiši postopek (ali napiši program ali podprogram, če ti je lažje), ki v okviru teh omejitev poišče največje možno število plošč, ki jih je mogoče položiti v trikotnik. Kot vhodne podatke tvoj postopek dobi velikost trikotnika (a in b), število plošč n in njihove dimenzije $w_1, h_1, w_2, h_2, \ldots, w_n, h_n$. **Utemelji**, zakaj je število plošč, ki ga najde tvoja rešitev, res največje možno.

4. Kodiranje

Recimo, da bi radi števke od 0 do 9 predstavili s 5-bitnimi kodami, torej z zaporedji petih ničel in enic. Takšnih peteric je kar $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 32$, mi pa moramo med temi 32 petericami izbrati deset peteric, ki jih bomo uporabili kot kode. Izbranim desetim petericam bomo rekli, da so *veljavne*, za ostalih 22 peteric pa recimo, da so *neveljavne*.

Če deset veljavnih peteric izberemo dovolj pazljivo, ima lahko tako dobljeni nabor kod nekatere lepe lastnosti, zaradi katerih je bolj odporen na napake pri branju, pisanju ali prenašanju podatkov. Primeri takšnih lepih lastnosti so:

(a) Če poljubni veljavni peterici spremenimo en bit (iz ničle v enico ali obratno), ali je tako spremenjena peterica zagotovo neveljavna? (Če to drži, potem vemo, da bomo zagotovo lahko zaznali napako pri prenosu podatkov, če se spremeni samo en bit.)

(b) Če poljubni veljavni peterici spremenimo eno ali več ničel v enice, ali je tako spremenjena peterica zagotovo neveljavna?

(c) Če v poljubni veljavni peterici enkrat med seboj zamenjamo dva sosednja različna bita (torej en par 01 spremenimo v 10 ali obratno), ali je tako spremenjena peterica zagotovo neveljavna?

Nekaj konkretnih primerov naborov 10 peteric:

Nabor	Katere lastnosti ima
00011, 00101, 00110, 01001, 01010, 01100, 10001, 10010, 10100, 11000	(a) in (b)
00000, 00011, 00110, 01001, 01100, 01111, 10010, 10101, 11000, 11011	(a) in (c)
00000, 00001, 00010, 00011, 00100, 00101, 00110, 00111, 01000, 01001	nobene

Prvi od teh treh naborov na primer nima lastnosti (c): 00011 je veljavna peterica in če v njej zamenjamo tretji in četrti bit, dobimo 00101, ki je tudi veljavna peterica.

Napiši program ali podprogram, ki za dani nabor 10 različnih peteric preveri, ali ima lastnost (c) ali ne. Z branjem vhodnih podatkov se ti ni treba ukvarjati; predpostaviš lahko, da so peterice že shranjene v neki globalni spremenljivki. Peterice lahko obravnavaš kot nize 5 znakov (`'0'` in `'1'`) ali pa kot cela števila (od 0 do 31, pri čemer posamezni biti predstavljajo posamezne števke v peterici), kar ti je lažje.

5. Golovec

Na predoru Golovec so ob vhodu in izhodu iz predora namestili merilnike, ki ob vstopu ali izstopu iz predora zabeležijo registrsko številko avtomobila in čas vstopa v sekundah od začetka dneva. S pomočjo teh podatkov lahko, ko avtomobil zapelje iz predora, izračunamo njegovo povprečno hitrost pri vožnji skozi predor in odkrijemo tiste, ki so vozili prehitro.

Napiši podprogram (oz. funkcijo), ki ga bo sistem poklical vsakič, ko bo kak avtomobil zapeljal v predor ali iz njega. Tvoj podprogram naj ob izstopu avtomobila iz predora izračuna njegovo povprečno hitrost in če ta presega 22 m/s, naj izpiše registrsko številko tega avtomobila. Za shranjevanje podatkov si lahko deklariraš tudi globalne spremenljivke in opišeš, kako jih je treba inicializirati. Tvoj podprogram naj bo take oblike:

```
procedure Golovec(regStevilka: string; cas: integer);          { v pascalu }
void Golovec(char* regStevilka, int cas);                       /* v C/C++ */
void Golovec(string regStevilka, int cas);                      // v C++
public static void Golovec(String regStevilka, int cas);        // v javi
public static void Golovec(string regStevilka, int cas);        // v C#
def Golovec(regStevilka, cas): . . .                            # v pythonu
```

Registrska številka je dolga največ 7 znakov. V predoru je največ 300 vozil hkrati in vsa imajo različne registrske številke. Vsako vozilo, ki kdaj zapelje v predor, prej ali slej zapelje tudi iz njega; za vsako tako vozilo bo sistem poklical naš podprogram natanko dvakrat: prvič, ko vozilo zapelje v predor, in drugič, ko zapelje iz njega. Vozila iz predora ne izstopajo nujno v enakem vrstnem redu, v kakršnem so vstopila vanj. Predor je dolg 622 m, ponoči pa v njem vedno potekajo nujna vzdrževalna dela, tako da naj te ne skrbi, da bi kakšen avtomobil vozil skozi predor ob polnoči.

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

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

Delovno okolje

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

Na vsakem računalniku imaš na voljo disk `D:\`, v katerem lahko kreiraš svoje datoteke in imenike. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali VB.NET, mi pa jih bomo preverili s 64-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz OpenJDK 1.7 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, Eclipse in druga orodja.

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

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

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

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

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

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

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

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk. (Izjemi sta prva naloga, kjer je testnih primerov 25 in za pravilen odgovor pri posameznem testnem primeru dobiš 4 točke, in peta naloga, kjer je testnih primerov 20 in za pravilen odgovor pri posameznem testnem primeru dobiš 5 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://tekmovanje.fri1.uni-lj.si/`):

- V pascalu:

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

- V C-ju:

```
#include <stdio.h>

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

- V C++:

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

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

- V javi:

```
import java.io.*;
import java.util.Scanner;

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

- V C#:

```
using System.IO;

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

- V Visual Basic.NETu:

```
Imports System.IO

Module Poskus

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

End Module
```

NALOGE ZA TRETJO SKUPINO

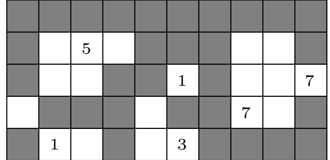
1. Nurikabe (`nurikabe.in`, `nurikabe.out`)

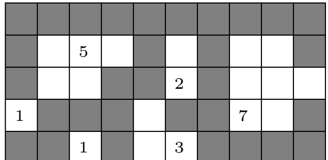
Nurikabe je igra, ki poteka na karirasti mreži. Posamezno polje mreže je lahko črno ali belo; na nekaterih belih poljih lahko stojijo tudi števila (od 1 do 9). Skupino črnih polj imenujemo *morje*, skupino belih polj pa *otok*. Polji, ki imata skupno stranico in sta iste barve, pripadata istemu morju (če sta črni) oz. istemu otoku (če sta beli).

Za otok rečemo, da je *pravilno označen*, če vsebuje natanko eno polje s številko in če je ta številka ravno enaka številu polj, ki tvorijo ta otok.

Med drugim lahko vidimo, da iz teh pravil sledi, da noben otok z več kot 9 polji ne more biti pravilno označen; in da imata dva različna otoka ali dve različni morji lahko skupno kakšno oglišče, ne pa tudi kakšne stranice.

Cilj igre nurikabe je izpolniti mrežo tako, da je morje eno samo, da so vsi otoki pravilno označeni in da v morju ni nobenega kvadrata 2×2 črnih polj. **Napiši program**, ki bo prebral opis mreže in pomagal preverjati te pogoje.





Primer: na levi mreži sta dve morji (črno polje v spodnjem levem kotu tvori majceno morje samo zase, vsa ostala črna polja pa tvorijo še drugo veliko morje) in 6 otokov, pri čemer trije niso pravilno označeni (otok velikosti 1 v najbolj levem stolpcu je brez številke; otok velikosti 2 v spodnji vrstici je pomotoma označen s številko 1; otok velikosti 7 na desni strani mreže pa je pomotoma označen dvakrat); in v mreži se pojavljata dva črna kvadrata velikosti 2×2 (ki se tudi malo prekrivata — ležita v prvih dveh vrsticah, od 5. do 7. stolpca). Desna mreža pa je izpolnjena po vseh pravilih.

Vhodna datoteka: v prvi vrstici sta dve celi števili, w in h, ločeni s presledkom. Pri tem je w širina mreže, h pa njena višina; veljalo bo $1 \leq w \leq 1000$ in $1 \leq h \leq 1000$. (V 60 % testnih primerov bo veljalo tudi $w \leq 20$ in $h \leq 20$.) Sledi h vrstic, vsaka od njih pa vsebuje w znakov, ki podajajo opis mreže. Pri tem so črna polja predstavljena z znaki `#`, bela polja brez številk s pikami `.`, bela polja s številko pa so predstavljena z znaki od `1` do `9`.

Izhodna datoteka: v prvo vrstico izpiši število morij; v drugo vrstico izpiši število otokov; v tretjo vrstico izpiši število pravilno označenih otokov; v četrto vrstico izpiši število črnih kvadratov velikosti 2×2 polj (štejejo tudi kvadrati, ki se prekrivajo oz. tvorijo še večja črna območja).

Primer vhodne datoteke:

```
10 5
##########
#.5.###..#
#..##1#..7
.###.##7.#
#1.#.3####
```

Pripadajoča izhodna datoteka:

```
2
6
3
2
```

(Opomba: to je leva mreža z gornje slike.)

2. Analiza signala (`signal.in`, `signal.out`)

Več oddajnikov oddaja signale — zaporedja ničel in enic, pri čemer se enice pojavljajo z neko konstantno periodo (ki pa je lahko pri različnih oddajnikih različna). Na primer, nek oddajnik oddaja s periodo 2:

$$1, 0, 1, 0, 1, 0, 1, 0, 1, 0, \ldots$$

Nek drug oddajnik pa s periodo 3:

$$1, 0, 0, 1, 0, 0, 1, 0, 0, 1, \ldots$$

Oddajniki so med seboj sinhronizirani tako, da na začetku opazovanja vsi oddajo enico (kot vidimo tudi v zgornjih dveh primerih).

Mi imamo detektor, ki zazna te signale. Pravzaprav ne more zaznati signalov posameznih oddajnikov, pač pa le njihovo vsoto. Pri zgornjih dveh oddajnikih bi na primer naš detektor zaznal takšno zaporedje:

$$2, 0, 1, 1, 1, 0, 2, 0, 1, 1, \ldots$$

Napiši program, ki analizira takšno zaporedje in ugotovi, koliko je vseh oddajnikov in kakšne so njihove periode.

Vhodna datoteka: v prvi vrstici je eno samo celo število n, ki pove dolžino našega zaznanega zaporedja; veljalo bo $1 \leq n \leq 100\,000$. (V 60 % testnih primerov bo veljalo tudi $n \leq 10\,000$.) V drugi vrstici je n nenegativnih celih števil, ki tvorijo prvih n členov zaporedja, ki ga je zaznal naš detektor; vsako od teh števil je manjše ali enako 10^9.

Zagotovljeno je, da v vhodnih podatkih ne bo napak; vhodno zaporedje je torej vedno takšno, do kakršnega bi res lahko prišlo z nekim primernim naborom oddajnikov in njihovih period.

Izhodna datoteka: v prvo vrstico izpiši dve celi števili, ločeni z enim presledkom; prvo od njiju naj bo število oddajnikov, drugo pa število oddajnikov, za katere z analizo prebranega zaporedja ni mogoče določiti periode. Sledi naj 0 ali več vrstic, po ena za vsako periodo, ki jo ima vsaj kakšen oddajnik. Vsaka od teh vrstic naj vsebuje dve pozitivni celi števili, ločeni z enim presledkom; prvo od njiju naj bo perioda, drugo pa število oddajnikov s to periodo. Urejene naj bodo naraščajoče po periodi.

Primer vhodne datoteke:

```
6
4 0 1 2 1 0
```

Pripadajoča izhodna datoteka:

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

3. Mafijski semenj (`semenj.in`, `semenj.out`)

Leto za letom v mesecu marcu na javnosti neznanem kraju poteka tradicionalni mafijski semenj, na katerem mafijci trgujejo z orožjem, ponarejenimi listinami, belimi praški in ostalim tovrstnim blagom. Letos se je zbralo n mafijcev, ki pripadajo m različnim mafijskim združbam.

Ravno letos pa je prišlo do neljubega incidenta. Iz daljave se sliši zavijanje policijskih siren in brnenje helikopterja. Kot vse kaže, jih je nekdo izdal. V paniki je vsak mafijec potegnil pištoli (kot se za resne mafijce spodobi, ima vsak pri sebi po dve pištoli) in ju usmeril v neka dva udeleženca sejma. Ker je bila panika nepopisna, se je lahko zgodilo tudi to, da je kateri od mafijcev obe pištoli usmeril v isto osebo ali celo samemu sebi v glavo.

Po začetnem preplahu so se mafijci hitro zbrali. Vsak je s pogledom premeril vse ostale in ugotovil, kdo vse je uperil pištolo v njega. Tisti mafijci, v katere ni usmerjena nobena pištola oz. tisti, v katere so usmerjene samo pištole mafijcev, ki pripadajo isti združbi, lahko pobegnejo. Tako so začeli drug za drugim bežati. Na koncu bo ostalo nekaj takih, ki ne bodo mogli pobegniti. Tem grozi, da bodo mnogo let preživeli na hladnem.

Napiši program, ki bo prebral podatke o tem, katerim mafijskim združbam pripadajo posamezni udeleženci in v koga merijo s pištolami, nato pa bo določil tiste mafijce, ki bodo šli na hladno.

Vhodna datoteka: v prvi vrstici sta števili n in m, ločeni z enim presledkom. Nato sledi n vrstic; i-ta med njimi vsebuje števila z_i, l_i in d_i, ločena s po enim presledkom. Pri tem je z_i številka združbe, ki ji pripada i-ti mafijec, l_i in d_i pa sta oznaki mafijcev, v katera i-ti meri s svojima pištolama. V vhodnih podatkih so mafijci označeni s števili $1, 2, \ldots, n$. Mafijske združbe so označene s števili $1, 2, \ldots, m$. Veljalo bo $1 \leq m \leq n \leq 10^6$, $1 \leq l_i \leq n$, $1 \leq d_i \leq n$ in $1 \leq z_i \leq m$. (Pri 50 % testnih primerov bo veljalo tudi $n \leq 1000$.)

Izhodna datoteka: izpiši oznake vseh mafijcev, ki bodo šli na hladno. Vsako oznako izpiši v svojo vrstico. Urejene naj bodo v naraščajočem vrstnem redu.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
2
4
5
6
```

4. Trgovanje z zrni (`zrna.in`, `zrna.out`)

V neki deželi je trgovanje z žitnimi zrni glavna dejavnost. Trgovanje poteka v glavnem pristanišču, kjer se vsako jutro zberejo večji in manjši lokalni kmetje ter trgovci z žitom. Cilj vsakega kmeta je, da trgovcu proda celoten pridelek, ki ga je pripeljal v pristanišče, in od tega nikakor ne odstopa. Vsak trgovec pride s svojo ladjo, ki lahko nosi določeno število zrn in ki bi se ob večji obremenitvi (četudi za eno samo zrno) nemudoma potopila.

Cilj vsakega trgovca je, da svojo ladjo čim bolj napolni; ne pozabimo pa, da hoče vsak kmet svoj pridelek prodati v celoti. **Napiši program**, ki za vsakega izmed trgovcev izračuna količino žitnih zrn, ki je najbližja nosilnosti njegove ladje (nikakor pa je ne presega) in bi jo lahko kupil ob predpostavki, da še nihče izmed kmetov ni prodal svojega pridelka.

Vhodna datoteka: v prvi vrstici sta dve celi števili, n in m, ločeni s presledkom; pri tem je n število kmetov (zanj velja $1 \leq n \leq 40$), m pa število trgovcev (zanj velja $1 \leq m \leq 400$). V drugi vrstici je n pozitivnih celih števil, ločenih s po enim presledkom; ta števila za vsakega od kmetov povedo, koliko zrn ima naprodaj. Sledi m vrstic, za vsakega trgovca po ena; vsaka od teh vrstic vsebuje po eno pozitivno celo število, ki pove nosilnost ladje tega trgovca (v zrnih). Noben kmet nima naprodaj več kot $3 \cdot 10^{12}$ zrn in noben trgovec ne želi kupiti več kot $2 \cdot 10^{13}$ zrn. (Nasvet: za računanje z zrni je koristno uporabiti kakšnega od 64-bitnih celoštevilskih tipov, na primer **long long** v C/C++, int64 v pascalu, **long** v C# in javi, Long v Visual Basic.NETu.)

Izhodna datoteka: vanjo izpiši m vrstic, za vsakega trgovca po eno (v enakem vrstnem redu, v kakršnem se ti trgovci pojavljajo v vhodni datoteki); v vsako od teh vrstic izpiši po eno samo celo število, namreč največje število zrn, ki bi jih ta trgovec lahko kupil ob upoštevanju omejitev naloge.

Primer vhodne datoteke:

```
5 7
2 3 5 11 44
1
6
42
49
8
12
22
```

Pripadajoča izhodna datoteka:

```
0
5
21
49
8
11
21
```

5. Razcep niza (`razcep.in`, `razcep.out`)

Pri tej nalogi se ukvarjamo z nizi, ki jih sestavljajo same ničle in enice. Število ničel v nizu s označimo z $N(s)$, število enic v nizu s pa označimo z $E(s)$. Zdaj lahko definiramo *oceno niza* kot $f(s) = \min\{N(s), E(s)\}$. To je torej število tistih števk (ničel oz. enic), ki jih je v tem nizu manj. Nekaj primerov: $f(\texttt{00110}) = 2$ (ker sta v tem nizu 2 enici in 3 ničle), $f(\texttt{111}) = 0$, $f(\texttt{1010}) = 2$.

Napiši program, ki dani niz razbije na natanko k nepraznih podnizov tako, da bo vsota ocen teh podnizov najmanjša možna.

Vhodna datoteka: v prvi vrstici sta dve celi števili, n in k, ločeni z enim presledkom; pri tem je n dolžina niza, ki bi ga radi razbili, k pa je število podnizov, na katere bi ga radi razbili. Veljalo bo $1 \leq k \leq n \leq 1000$ (v 40 % testnih primerov bo veljalo tudi $n \leq 20$). V drugi vrstici je niz, ki bi ga radi razbili; to je zaporedje n znakov, vsak od njih pa je bodisi „`0`“ bodisi „`1`“.

Izhodna datoteka: vanjo izpiši eno samo celo število, namreč najmanjšo možno vsoto ocen podnizov, ki jo lahko dosežemo, če niz iz vhodne datoteke primerno razbijemo na k nepraznih podnizov.

Primer vhodne datoteke:

```
9 3
110100100
```

Pripadajoča izhodna datoteka:

```
2
```

Komentar: niz `110100100` lahko razbijemo na tri podnize kot `1101`+`001`+`00`. Vsota ocen teh podnizov je $f(\texttt{1101}) + f(\texttt{001}) + f(\texttt{00}) = 1 + 1 + 0 = 2$. Pokazati je mogoče, da manjše skupne ocene pri razbijanju tega niza na tri podnize ni mogoče doseči.

NALOGE ZA ŠOLSKO TEKMOVANJE

23. januarja 2015

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. Lov na sataniste

CIA lovi sataniste s pomočjo vohunjenja po sporočilih elektronske pošte. Včasih so bili satanisti ponosni ljudje in so v vsako sporočilo napisali „`666`“. Zdaj so bolj previdni in sicer še vedno pogosto uporabijo v sporočilu tri šestice skupaj, vendar včasih namesto „`6`“ napišejo „`six`“, poleg tega pa se šestice ne pojavijo nujno čisto ena zraven druge. **Napiši podprogram** (funkcijo) oblike `Preveri(s, p)`, ki za dani niz s ugotovi, če se v kateremkoli razponu p zaporednih znakov pojavljajo trije znaki „`6`“ ali nizi „`six`“; lahko mešano. Na primer, pri nizu

$s =$ `"I say 6 bla 6six hail satan!"`

bi morala funkcija vrniti **false** za $p < 10$ in **true** za $p \geq 10$ (ker se v s pojavlja podniz `"6 bla 6six"`, dolg 10 znakov). Poskusi funkcijo napisati tako, da bo učinkovita tudi za velike p (učinkovitejše rešitve dobijo več točk).

2. Hišna številka

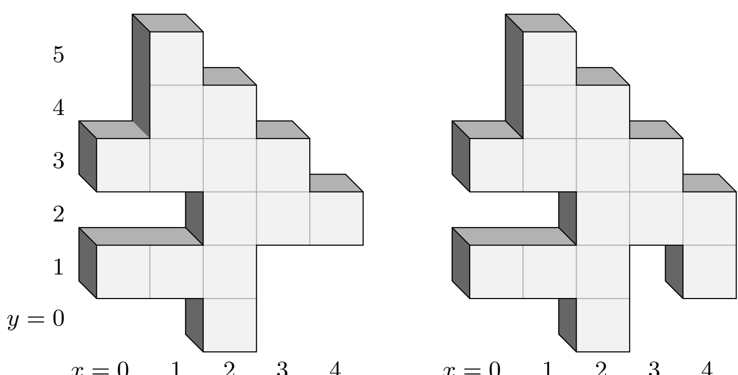
Tablica s hišno številko je pritrjena na steno z dvema žebljičkoma. Zgornji odpade in tablica se okrog spodnjega zavrti za 180 stopinj. Števke, ki so imele v prvotnem številu vrednost 6, so v tem novem položaju videti kot 9; bivše 9 so zdaj videti kot 6; števke 0, 1 in 8 so še zdaj videti kot 0, 1 ali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

prej *potem*

Polde se pripravlja na gradnjo hiše in ker ne bi rad imel dela z obračanjem tablice s hišno številko nazaj, hoče takšno hišno številko, ki se pri obratu za 180 stopinj ne spremeni (kot na primer 96 v prejšnjem odstavku). Pri tem si želi tudi, da bi bila številka čim manjša; biti pa mora večja od n, ker so številke od 1 do n že zasedene. **Napiši podprogram** (funkcijo) Naslednja(n), ki za dani n poišče najmanjšo številko, ki ustreza tem pogojem. Predpostavi, da za n velja $1 \leq n \leq 1\,000\,000$.

3. Stolpnica

V nekem mestu bi radi zgradili stolpnico zelo moderne oblike, sestavljeno iz več ogromnih, enako velikih kock. Stolpnica je sicer trodimenzionalna, vendar bodo imele v eni od dimenzij vse kocke enako koordinato, zato lahko to dimenzijo zanemarimo. Načrt stolpnice lahko zato opišemo s karirasto mrežo, ki ima h vrstic in w stolpcev — pravzaprav je to dvodimenzionalna tabela T, v kateri ima element T[y][x] vrednost **true**, če je kocka (x, y) v stolpnici prisotna, sicer pa ima vrednost **false**. Pri tem nam y pomeni številko vrstice (najnižja vrstica ima $y = 0$, najvišja pa $y = h - 1$), x pa številko stolpca (najbolj levi stolpec ima $x = 0$, najbolj desni pa $x = w - 1$). Primer dveh stolpnic (s širino $w = 5$ in višino $h = 6$) kaže naslednja slika:



Težava je, da bodo stolpnico gradili od spodaj navzgor, torej od manjših y proti večjim, in ne bi bilo dobro, če bi nekatere kocke med gradnjo „visele v zraku“, ker jih s preostankom stolpnice povezujejo le višje ležeče plasti, ki jih sploh še niso zgradili.

Definirajmo: kocka (x, y) je *podprta*, če leži v najnižji vrstici (torej če je $y = 0$) ali pa če je podprta njena spodnja soseda $(x, y - 1)$, njena leva soseda $(x - 1, y)$ ali njena desna soseda $(x + 1, y)$.

Opiši postopek (ali napiši program/podprogram, kar ti je lažje), ki za dano tabelo T preveri, ali so vse kocke v njej podprte.

Primer: pri levi stolpnici na gornji sliki so podprte vse kocke, pri desni pa ne (kajti kocka $(4, 1)$ pri tej stolpnici ni podprta).

4. Kontrolne naloge

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

Opiši postopek (ali napiši podprogram, če ti je lažje), ki ugotovi, ali lahko vse naloge rešijo v t (ali manj) dnevih, če si delo primerno razdelijo med seboj. Pri tem predpostavi, da so vse količine, omenjene v tej nalogi (p, d, s, t in $k_1, \ldots, k_p$), znane.

5. Tehtnica

Imamo staromodno tehtnico z dvema posodama; dane so tudi uteži z masami $1, 2, 4, 8, 16, 32, 64, 128, \ldots$ gramov (po ena utež za vsako maso, ki je potenca števila 2; masa uteži ni navzgor omejena, torej je uteži neskončno mnogo). Dan je tudi nek predmet z maso n gramov (n je pozitivno celo število in je podan), ki ga položimo na levo posodo tehtnice. **Opiši postopek** (ali napiši podprogram, če ti je lažje), ki določi najmanjše število uteži, ki jih moramo uporabiti, da spravimo tehtnico v ravnovesje. Pri tem lahko uteži polagamo v levo in/ali desno posodo.

Primer: $n = 7$ lahko spravimo v ravnovesje tako, da v desno posodo položimo uteži 1, 2 in 4. Boljša rešitev pa je, da v desno posodo položimo utež 8, v levo pa utež 1 (tako smo uporabili le dve uteži namesto treh).

Namig: ločeno obravnavaj sode in lihe n.

NEUPORABLJENE NALOGE IZ LETA 2013

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 8. tekmovanjem ACM v znanju računalništva (leta 2013), 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. 87–141) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Statistika na besedilu

Narediti želimo preprosto statistično analizo besedila. **Napiši program**, ki bere besedilo s standardnega vhoda, dokler ne prebere prazne vrstice, ki označuje konec besedila za analizo. Za prazno vrstico se nahajajo vrstice s po eno črko. Za vsako tako črko mora program izpisati, kolikokrat se je ta črka pojavila v vhodnem besedilu in to ne glede na to, ali je črka velika ali mala (primer: „`A`“ ali „`a`“ štejemo kot isto črko). Če je pa za to črko že bilo spraševano, mora podprogram izpisati tudi, kolikokrat je bilo spraševano za to črko.

2. Problematične formule

Venceslava je imela od vedno rada matematiko. Rada je štela svoje igrače, računala, koliko dni še manjka do njenega rojstnega dne ali koliko bonbonov so ji dolžni njeni sošolci. Vedno se je veselila novih izzivov, na primer seštevanja s prehodom prek desetic, pa poštevanke do 10, seštevanja in množenja večmestnih števil, kasneje še potenciranja in tetriranja. Najraje od vsega je imela oklepaje. V drugem razredu je bila cel mesec popolnoma srečna, ko so vzeli računanje z oklepaji. Ko so jo v srednji šoli naučili, kaj so množice in absolutna vrednost, se je njen nabor najljubših ločil le še razširil. Na fakulteti pa so se ji odprle povsem nove razsežnosti matematike, saj je spoznala, da obstajajo tudi oglati in kotni oklepaji, s katerimi se računa na zelo zanimive načine.

Vendar je s časom spoznala, da veliko oklepajev ne pomeni le zabave, ampak tudi nepreglednost in zmedo. Ko se je na primer ukvarjala s formulo

$$L(a,b) = \max_{h \in \mathbb{R}} \left\{ \frac{\vartheta h}{b-a} \left[\left(\frac{f(a)(1-\vartheta) + f(b)\vartheta}{h} \right) \frac{\langle a,b \rangle}{\langle a,a \rangle} a \right] ; \forall \vartheta \in [0,1] \right\}$$

in jo poskusila vtipkati v računalnik, je dobila znano sporočilo `Syntax Error!` in bila je prepričana, da je pozabila nek oklepaj. Toda katerega! Po nekajminutnem naporu je našla svojo napako in se odločila, da se to ne sme več ponoviti.

Napiši program, ki preveri, ali so vsi oklepaji v dani formuli zaprti in pravilno gnezdeni.

Vhodni podatki: v vsaki vrstici standardnega vhoda bo podana ena formula. Formule bodo vsebovale štiri tipe oklepajev:

- okrogle `( )`

- oglate `[ ]`
- zavite `{ }`
- kotne `< >`

Posamezna formula bo krajša od 50 000 znakov.

Izhodni podatki: izpiši `Pravilno!`, če je ujemanje in gnezdenje oklepajev v formuli brez napak, sicer izpiši `Narobe!`.

Težja različica naloge: kaj pa, če v formulah poleg oklepajev dovolimo še navpične črte `| |`, kjer je težava v tem, da sta „oklepaj“ in „zaklepaj“ enaka? Dopolni svojo rešitev tako, da bo pravilno obravnavala tudi navpične črte. Utemelji pravilnost svoje rešitve. (Nekaj primerov: formule `(||)`, `(|()|)` in `|(||)|` so veljavne, formuli `(|)|` in `|(|)|` pa ne.)

Primer vhoda:

```
[(<>)]{}
2+(3-5x)*34y
(3u+4-3x*[12x+8(y+z])
f(2x (3x)^5 + 4y (<a,t> - [1-t]/[3+h])))
L(A,B)=max_(h in R){(t h)/(b-a)*((f(a)(1-t)+f(b)t)/h)(<a,b>/<a,a>)*a for t in [0,1]}
```

Pripadajoč izhod:

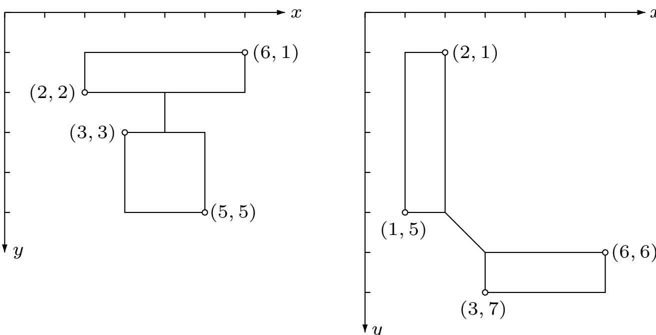
```
Pravilno!
Pravilno!
Narobe!
Narobe!
Pravilno!
```

3. Miselni vzorec

Napisati želimo program za risanje miselnih vzorcev. Dana sta dva pravokotnika (njune stranice so vodoravne in navpične, torej vzporedne koordinatnima osema — vedno, ko pri tej nalogi rečemo „pravokotnik“, bomo s tem mislili takega, ki ima vodoravne in navpične stranice). Iščemo najkrajšo daljico, ki ima eno krajišče na robu prvega pravokotnika in drugo krajišče na robu drugega pravokotnika. **Napiši podprogram**, ki bo kot vhodne parametre prejel celostevilske koordinate oglišč na zaslonu in bo izpisal koordinate krajišč daljice.

```
void Povezi(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4);
```

Za vsak pravokotnik dobimo koordinate dveh diagonalno nasprotnih oglišč: pri prvem pravokotniku sta to (x_1, y_1) in (x_2, y_2), pri drugem pa (x_3, y_3) in (x_4, y_4). Koordinate krajišč daljice naj tvoj podprogram izpiše tako, da najprej izpiše tisto krajišče, ki leži na robu prvega pravokotnika, nato pa tisto, ki leži na robu drugega pravokotnika. Če je možnih več enako dobrih najkrajših daljic, je vseeno, katero od njih izpišeš.



Če torej pokličemo (kar ustreza primeroma z gornje slike):

```
Povezi(6, 1, 2, 2, 5, 5, 3, 3);
Povezi(3, 7, 6, 6, 1, 5, 2, 1);
```

je eden od možnih primernih izpisov naslednji:

```
(4, 2)-(4, 3)
(3, 6)-(2, 5)
```

4. Pranje denarja

V neki deželi je živel slikar, ki se je poleg slikanja ukvarjal tudi s pranjem denarja. Iz tujine je za svoje delo prejel na primer 25 000 goldinarjev, nato pa je prijatelju A poslal 10 000 goldinarjev, prijatelju B pa 15 000 goldinarjev. Tako prijateljema A in B niso mogli dokazati, da sta denar prejela zaradi prodaje medvedjih kožuhov, ki je bila v tej deželi prepovedana. Slikar pa si je, da ga ne bi nihče opeharil, vse svoje transakcije skrbno zapisoval kot seznam celih števil: prejemke s predznakom + ter izdatke s predznakom −.

Nekega dne pa je slikarja obiskala lokalna komisija za preprečevanje pranja denarja in od njega zahtevala seznam prejemkov in izdatkov v kronološkem vrstnem redu. Slikar je vedel, da komisija posebno skrbno pregleda t. i. *sumljive posle*, to so skupine zaporednih prejemkov in izdatkov z vsoto nič, tam kjer slikar ni nič pridobil in nič izgubil. Tako se je slikar odločil, da seznam pred oddajo takih poslov očisti. Izbrisal bo skupine zaporednih prejemkov in izdatkov z vsoto nič, vendar tako, da bo s tem izbrisal kar največ transakcij. **Opiši postopek**, ki bo za dani seznam našel tak izbris.

5. Simbolične povezave

Datotečni sistem si lahko malo poenostavljeno predstavljamo kot grafu podobno strukturo, ki jo sestavljajo vozlišča treh vrst: imeniki (direktoriji), datoteke in simbolične povezave. Vsako vozlišče ima enolično identifikacijsko številko (nenegativno celo število).

Imenik je vozlišče, ki lahko vsebuje 0 ali več drugih vozlišč — lahko si ga predstavljamo kot seznam parov ⟨*ime*, *ID vozlišča*⟩, ki navajajo vsebino imenika. Vsako

vozlišče razen enega pripada natanko enemu imeniku; izjema je *korensko vozlišče*, ki ne pripada nobenemu imeniku (je pa sámo zagotovo imenik).

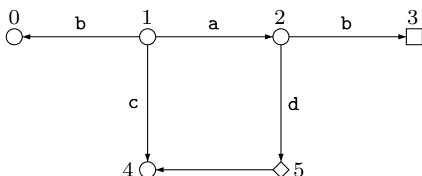
Simbolična povezava je poseben tip vozlišča, ki nima druge vsebine kot ID nekega drugega vozlišča — temu rečemo, da povezava *kaže* na tisto drugo vozlišče (to drugo vozlišče je lahko imenik ali datoteka). Na posamezno vozlišče lahko kaže 0 ali več simboličnih povezav.

Pot je niz oblike `/`s_1`/`s_2`/`$\ldots$`/`s_n, čigar pomen je definiran z naslednjimi pravili:

- če je $n = 0$, imamo *prazno pot* (namesto s praznim nizom jo lahko zapišemo tudi kot „`/`“), ki predstavlja korensko vozlišče;
- pot oblike P`/..` predstavlja imenik, ki vsebuje tisto vozlišče, ki ga predstavlja pot P (če P predstavlja korensko vozlišče, potem tudi P`/..` predstavlja korensko vozlišče);
- pot oblike P`/`s_n za $s_n \neq$ `..` je veljavna le, če P predstavlja nek imenik in če ta imenik vsebuje par oblike $\langle s_n, u\rangle$; potem pot P`/`s_n predstavlja vozlišče u, razen če je u simbolična povezava, tedaj pa pot P`/`s_n predstavlja vozlišče, na katero kaže u.

Na misel nam lahko pride, da bi pot s segmenti oblike `..` poenostavili — iz `/a/b/../c` bi na primer naredili `/a/c`. V splošnem bomo definirali, da pot *poenostavimo* tako, da gremo po segmentih poti od leve proti desni in vsakič, ko naletimo na segment oblike `..`, pobrišemo ta segment in še tistega pred njim (natančneje: zadnjega takega pred njim, ki še ni bil pobrisan).

Zaradi simboličnih povezav se lahko zgodi, da tako poenostavljena pot ne predstavlja istega vozlišča kot prvotna pot. Za primer si oglejmo datotečni sistem na naslednji sliki; krogi ○ predstavljajo imenike, kvadrati □ datoteke, karo ◇ pa je simbolična povezava. Ob vsakem vozlišču je njegova identifikacijska številka (vozlišče 1 je korensko vozlišče).



S povezavami je prikazana vsebina imenikov in simboličnih povezav. Na gornjem primeru na primer vidimo, da je vozlišče 2 imenik, ki vsebuje dva para: $\langle$`b`$, 3\rangle$ in $\langle$`d`$, 5\rangle$; vozlišče 5 je simbolična povezava, ki kaže na vozlišče 4; ipd.

Po naših prej opisanih definicijah lahko rečemo, da pri tem datotečnem sistmeu pot `/a/d` predstavlja vozlišče 4, enako kot tudi pot `/c`; in pot `/a/d/../b` predstavlja vozlišče 0, enako kot tudi pot `/b`. Po drugi strani, če pot `/a/d/../b` poenostavimo, dobimo `/a/b`, ta pot pa predstavlja vozlišče 3. V tem primeru torej poenostavljena pot ne predstavlja istega vozlišča kot prvotna pot pred poenostavljanjem.

Napiši program, ki za dano pot ugotovi, ali predstavlja isto vozlišče kot pot, ki nastane iz nje pri poenostavljanju. Predpostavi, da so za delo z datotečnim sistemom na voljo naslednje funkcije:

- **int** GetRootNode() — vrne številko korenskega vozlišča;
- **bool** IsDir(**int** node) — vrne **true**, če je vozlišče s številko node imenik, sicer pa vrne **false**;
- **int** GetChildNode(**int** node, **const char*** s) — če je vozlišče s številko node imenik in vsebuje par $\langle \mathsf{s}, x \rangle$, funkcija vrne x, sicer pa vrne -1;
- **int** GetParentNode(**int** node) — vrne številko imenika, ki vsebuje vozlišče node; če je node korensko vozlišče, funkcija vrne -1;
- **int** IsLink(**int** node) — če je vozlišče node simbolična povezava, vrne številko vozlišča, na katero kaže ta simbolična povezava, sicer pa vrne -1.

6. Letala

Nadzorniki letalskega prometa imajo precej stresnega dela, saj morajo preprečevati, da bi se zgodile nesreče na nebu. V pomoč pri tem so jim seveda radarji, ki opazujejo nebo, in odzivniki v letalih, ki sporočajo radarjem, katero letalo so ravnokar „pogledali“.

Da bi jim pri tem pomagali in jih razbremenili, so se domislili, da bi jim pri tem delu lahko pomagali računalniki. Vendar je zanje treba še napisati program, ki bo opravljal to delo. Ker so programerji leni, so napisali nekaj funkcij (podprogramov), potem so pa šli na demonstracije. Ker se mudi, moraš **napisati** glavni **program**, ki bo bral podatke z radarja in pazil, da dve letali nisti na poti trčenja. Pot trčenja pomeni, da letali potujeta skozi isto točko in sta od nje približno enako oddaljeni. Približno enako pomeni, da je razlika oddaljenosti manj, kot je definirana varnostna konstanta razmika (spremenljivka Razmik tipa **double**). Da je zadeva enostavnejša, predpostavi, da so vsa letala na isti višini in letijo enako hitro. Predpostaviš lahko tudi, da je radar v točki $(0, 0)$ in da radar vedno vidi vsa letala, ki so v njegovem dosegu.

Radar se vrti zelo hitro in lahko predpostaviš, da vsa letala, ki so trenutno v nadzorovanem prostoru, zazna tako hitro, da se v tem času še ne utegnejo premakniti tako daleč, da bi nas to motilo. Če radar nekega letala v enem prehodu ne zazna, je bodisi pristalo bodisi odšlo iz nadzorovanega prostora.

Na voljo imaš nekaj že prej omenjenih podprogramov in funkcij, ki so jih že napisali programerji pred teboj in lahko predpostaviš, da delajo povsem pravilno:

- **void** Radar(**double*** x1, **double*** y1, **int*** oznaka), ki vrne koordinate točke, kjer se nahaja letalo in oznaka odzivnika (transponderja). Podprogram blokira, če radar ne „vidi“ nobenega letala;
- **void** Presek(**double** x1, **double** y1, **double** x2, **double** y2, **double** x3, **double** y3, **double** x4, **double** y4, **double*** x0, **double*** y0), ki vrne presečišče dveh premic, in sicer prve, ki gre skozi točki (x1, y1) in (x2, y2), in druge, ki gre skozi točki (x3, y3) in (x4, y4);
- **int** Stevec(), ki vrne zaporedno številko trenutnega obrata radarja;
- **int** abs(**double** i), ki vrne absolutno vrednost podanega števila;

- **double** Razdalja(**double** x1, **double** y1, **double** x2, **double** y2), ki vrne razdaljo med dvema točkama.

7. Oklepajski izrazi

Dan je niz, ki ga sestavljajo znaki `( ) [ ] { } < >`, ki pa niso nujno pravilno gnezdeni. **Opiši postopek**, ki ugotovi najmanjše potrebno število znakov, ki jih je treba v dani niz vriniti, da dobiš pravilno gnezden oklepajski izraz.

8. Generator naključnih števil

Dano je celo število $k > 1$ in funkcija Nakljucno1(), ki vrne naključno število iz $\{0, 1, \ldots, k-1\}$, pri čemer so vsa števila enako verjetna. Dano je tudi celo število n. S pomočjo funkcije Nakljucno1 sestavi podobno funkcijo Nakljucno2(), ki mora vrniti naključno število iz $\{0, 1, \ldots, n-1\}$, pri čemer so vsa števila enako verjetna. Razmisli o tem, kolikokrat kliče tvoja funkcija Nakljucno2 funkcijo Nakljucno1 v povprečju in kolikokrat v najslabšem primeru.

9. Naključni vzorec

Smo lastnik popularne spletne strani, na katero lahko uporabniki nalagajo (uploadajo) ogromne datoteke. Nimamo prostora, da bi vse te datoteke shranjevali, vendar vseeno želimo narediti analizo njihove vsebine. Odločimo se, da bomo analizirali 100 naključno izbranih datotek, ki bodo naložene v naslednjem dnevu. Vemo, da bo v enem dnevu naloženih veliko več kot 100 datotek, ne vemo pa, koliko.

(a) **Opiši postopek**, ki bo naredil ta naključni izbor in pri tem nikoli hranil hkrati več kot 1000 datotek. Tvoj postopek naj bo sestavljen iz dveh funkcij: OnUpload(d), ki se bo klicala vsakič, ko od uporabnikov dobimo nov dokument d, in GetSample(), ki se bo klicala čisto na koncu in mora vrniti ali izpisati 100 naključnih naloženih dokumentov. Pomembno pri tem je, da mora imeti vsak od možnih izborov 100 dokumentov enako verjetnost, da bo prav to tisti izbor, ki ga bo vrnila funkcija GetSample. Predpostaviš lahko, da ti je na voljo funkcija Random(), ki vrne naključno realno število z območja $[0, 1)$ (števila, ki jih vrača ta funkcija, so enakomerno porazdeljena po tem območju).

(b) V prejšnjem odstavku smo imeli pogoj, da morajo biti vsi možni izbori 100 dokumentov enako verjetni. Kaj pa, če ga nadomestimo s pogojem, da mora imeti vsak od prebranih dokumentov (kolikor koli jih že pač je) enako verjetnost, da bo prišel v končni izbor? Preveri, če je ta pogoj enakovreden prvotnemu, in če ni, opiši primer postopka, ki reši nalogo s tem novim pogojem, s prvotnim pa ne.

10. Štetje nizov

Recimo, da imamo abecedo z d znaki; ko pri tej nalogi govorimo o nizih, s tem mislimo samo nize, ki jih sestavljajo le znaki te abecede.

Dan je nek niz s, dolg k znakov. **Opiši postopek**, ki ugotovi, koliko je takih nizov x dolžine n, ki vsebujejo s kot podniz.

V tej nalogi se pravzaprav skriva več različic, odvisno od tega, ali (a) zahtevamo, da je s strnjen podniz niza x, ali dovolimo tudi, da je s nestrnjen podniz niza x;

(b) dodamo predpostavko, da so v s sami različni znaki, ali dovolimo tudi možnost, da se isti znak v s-ju pojavi po večkrat; in (c) se omejimo na take nize x, ki so sestavljeni iz samih različnih znakov, ali dovolimo, da se v x isti znak pojavi po večkrat.

Pri vsaki od teh treh podrobnosti imamo dve možnosti, tako da skupaj nastane $2 \cdot 2 \cdot 2 = 8$ različic naloge. Poskusi rešiti vseh osem.

11. Botanika

Janko in Metka sta se pri pouku diskretne matematike učila o drevesih. Drevo sestavljajo točke in (neusmerjene) povezave med njimi, pri čemer mora za povezave veljati, da nikjer ne tvorijo cikla in da je mogoče po njih priti od vsake točke do vsake druge točke drevesa (temu rečemo, da je graf *povezan*). Vsaka povezava povezuje dve točki, ki jima pravimo *krajišči* te povezave; *stopnja* točke pa je število povezav, ki imajo to točko za eno od krajišč.

Med poukom je Janko na mizo s svinčnikom narisal več dreves, poleg tega pa je svoje risbe obogatil še s kulijem napisanimi stopnjami točk. Naslednjo uro je Metka z radirko uspela uničiti Jankove umetnine, stopnje točk pa so preživele Metkin vandalizem. Janko si želi rekonstruirati svoja drevesa, zato te prosi za pomoč. **Napiši program ali podprogram**, ki bo iz seznama stopenj n točk zgradil eno možno drevo oz. sporočil, da to ni mogoče.

Omejitve: $1 \leq n \leq 10^6$.

12. Volitve

Država je hierarhično razdeljena na več enot. Na prvem nivoju jo lahko razdelimo na pokrajine. Nekatere pokrajine (ne nujno vse) se delijo na občine. Večje občine se lahko delijo naprej na upravne enote itd. Enoti, ki se ne deli, bomo rekli *osnovna enota*, ne glede na to, na katerem nivoju hierarhije je. Zaradi vedno večjega števila protestov politiki že razmišljajo o volitvah. Volitve se bodo izvedle v vseh n osnovnih enotah, rezultati višjih enot pa so enaki vsoti rezultatov osnovnih enot, ki jih vključujejo. Osnovne enote so oštevilčene od 1 do n; i-ta osnovna enota ima p_i prebivalcev in vsi glasujejo za kandidata k_i. Ne-osnovne enote so oštevilčene od $n+1$ do m, pri čemer zadnja med njimi (enota m) predstavlja celotno državo. Enoti i (za $1 \leq i < m$) je v hierarhiji neposredno nadrejena enota t_i. Politike pa zanima, koliko glasov bodo dobili v okviru različnih enot, ki niso nujno osnovne.

(a) **Opiši postopek**, ki dobi več poizvedb oblike (k_j, e_j) in za vsako od njih ugotovi, koliko glasov dobi politik k_j v enoti e_j. (Na poizvedbe ni treba odgovarjati sproti; tvoj postopek dobi vse poizvedbe na začetku, preden začne odgovarjati nanje.)

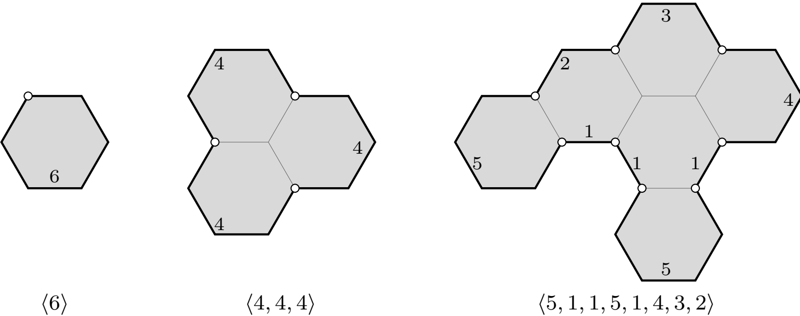
(b) **Opiši postopek**, ki za vsako enoto (na vseh nivojih hierarhije) ugotovi, kdo je zmagovalec volitev (kdo je dobil največ glasov v tisti enoti).

Omejitve: $1 \leq m \leq 10^6$, $1 \leq k_i \leq 10^6$, $1 \leq p_i \leq 10^6$.

13. Šestkotna mreža

V šestkotni mreži izberemo nekaj šestkotnikov, ki tvorijo povezan lik brez lukenj. Spodnja slika kaže primere treh takih likov: levi lik je en sam šestkotnik, srednji lik

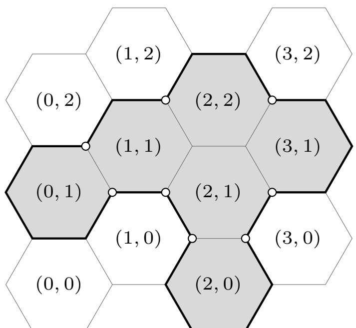
je iz treh šestkotnikov, desni lik pa je iz šestih šestkotnikov. Tak lik lahko opišemo takole: na zunanjem robu lika si oglejmo tista oglišča, kjer se stikata dva šestkotnika našega lika (na spodnji sliki so ta oglišča označena s krožci); preštejmo robove od enega takega oglišča do naslednjega (na sliki kažejo to številke na notranji strani zunanjega roba lika) in kot opis lika vzemimo zaporedje tako dobljenih števil (na sliki je ta opis prikazan v oklepajih pod vsakim likom).



$\langle 6\rangle$ $\langle 4,4,4\rangle$ $\langle 5,1,1,5,1,4,3,2\rangle$

Lik ima lahko tudi več enakovrednih opisov, odvisno pač od tega, kje začnemo in v kateri smeri se sprehajamo po njem. Desni lik z gornje slike bi lahko med drugim opisali tudi kot $\langle 1,4,3,2,5,1,1,5\rangle$, pa kot $\langle 5,1,1,5,2,3,4,1\rangle$ itd. Naš opis si torej lahko namesto kot zaporedje predstavljamo kot cikel, zato bomo tej predstavitvi rekli *ciklični opis* lika.

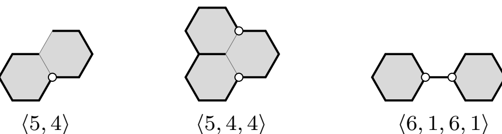
Drugi način za opis naših likov pa je, da v šestkotno mrežo vpeljemo koordinatni sistem. Vsakemu šestkotniku pripišimo par koordinat (x, y), pri čemer je x številka stolpca, y pa številka vrstice. Ker imamo opravka z mrežo šestkotnikov namesto kvadratkov, je vsak drugi stolpec zamaknjen za pol vrstice navzgor:



Lik na tej sliki je enak kot desni lik s prve slike; vidimo, da ga sestavljajo šestkotniki $(0,1)$, $(1,1)$, $(2,1)$, $(3,1)$, $(3,0)$ in $(3,2)$.

(a) Recimo, da imamo lik opisan kot 2-d tabelo logičnih vrednosti (tip **bool** ali boolean), v kateri element na indeksih [x][y] pove, ali je šestkotnik s koordinatami (x, y) prisoten v našem liku ali ne. **Opiši postopek**, ki za dani lik izračuna njegov ciklični opis. Če vhodna tabela sploh ne opisuje veljavnega lika (npr. ker lik vsebuje luknje ali pa je sestavljen iz več nepovezanih kosov), naj postopek to zazna in javi napako.

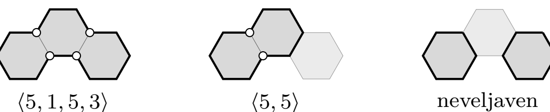
(b) **Opiši postopek**, ki za dano zaporedje števil preveri, ali je to veljaven ciklični opis kakšnega lika. To pomeni, da mora biti rob lika, kot ga dobimo iz danega zaporedja, lepo sklenjen in da se nikjer ne sme sekati sam s sabo ali pa po večkrat obiskati iste stranice. Če se ciklični opis izkaže za veljavnega, naj tvoj postopek izračuna ploščino lika (iz koliko osnovnih šestkotnikov je sestavljen). Naslednja slika kaže tri primere neveljavnih opisov:



(c) **Opiši postopek**, ki dobi dva ciklična opisa likov (predpostaviš lahko, da sta veljavna) in preveri, ali sta oba lika enaka (z drugimi besedami, ali lahko dobimo en opis iz drugega tako, da ga ciklično zamaknemo in mogoče še preberemo z desne proti levi). Na primer: $\langle 4, 4, 1, 5, 2\rangle$, $\langle 1, 5, 2, 4, 4\rangle$ in $\langle 4, 2, 5, 1, 4\rangle$ vsi opisujejo enak lik.

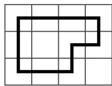
(d) Dano je zaporedje števil $\langle a_1, \ldots, a_n\rangle$, ki je veljaven ciklični opis nekega lika. Izberimo si nek indeks k (z območja $1, \ldots, n$) in poglejmo, kateremu šestkotniku pripada člen a_k v našem opisu. Ta šestkotnik pobrišimo iz lika. **Opiši postopek**, ki ugotovi, ali je tako spremenjen lik še veljaven (lahko se namreč zgodi, da po brisanju lik razpade na več nepovezanih delov); če je veljaven, naj tvoj postopek izračuna ciklični opis novega lika.

Primer takega brisanja kaže naslednja slika. Če dobimo opis $\langle 5, 1, 5, 3\rangle$ in $i = 3$, pobrišemo desni šestkotnik in dobimo veljaven lik z opisom $\langle 5, 5\rangle$. Če pa bi imeli $i = 2$ ali $i = 4$, bi pobrisali srednji šestkotnik in dobljeni lik ne bi bil veljaven, saj bi ga sestavljala dva nepovezana šestkotnika.



14. Kocke

Imamo več kvadratnih ploščic s stranico dolžine 1. Na vsaki ploščici je narisana črta, ki povezuje središče kvadrata z razpoloviščema dveh stranic. Ploščice zato lahko razdelimo na dve vrsti: ploščice tipa I (pri katerih je črta ravna in povezuje razpolovišči dveh nasprotnih stranic) in ploščice tipa L (pri katerih je črta prelomljena in povezuje razpolovišči dveh sosednjih stranic, pri tem pa gre skozi središče kvadrata). Te ploščice bi radi zložili v celice kariraste mreže tako, da bi črte na ploščicah tvorile nepretrgano sklenjeno pot (pri tem lahko ploščice vrtimo v korakih po 90°). Primer takšne poti kaže naslednja slika (na njej smo uporabili 3 ploščice tipa I in 6 ploščic tipa L).



Takšno sklenjeno pot lahko opišemo tako, da začnemo v poljubni taki točki poti, ki leži na meji med dvema ploščicama, in od tam naprej po vrsti navedemo za vsako ploščico na poti, ali predstavlja korak naravnost naprej ali ovinek v levo ali ovinek v desno. Korak naprej se seveda zgodi le pri ploščicah tipa I, ovinek pa le pri ploščicah tipa L. Pot lahko torej predstavimo z zaporedjem znakov `N`, `L` in `D`; pot z zgornje slike bi lahko na primer opisali kot `DNNDDLDNDN` (možni pa so še tudi drugi opisi, odvisno od tega, kje na poti začnemo in v katero smer gremo po njej).

(a) **Opiši postopek**, ki za dani opis poti preveri, ali je to sploh veljaven opis poti (torej ali se ploščice na njej ne prekrivajo in ali je pot na koncu sklenjena).

(b) Recimo, da dobimo s ploščic tipa I in t ploščic tipa L. **Opiši postopek**, ki ugotovi, ali je iz teh ploščic sploh mogoče sestaviti primerno sklenjeno pot (pri tem moramo uporabiti vse ploščice, lahko pa si sami izberemo, v kakšnem vrstnem redu jih bomo uporabili); če jo je mogoče, naj tudi sestavi opis neke takšne poti.

(c) Reši podnalogo (b) še z dodatno omejitvijo, da je predpisano, v kakšnem vrstnem redu moramo uporabiti ploščice — lahko si predstavljamo, da kot vhodni podatek dobimo niz `I`-jev in `L`-jev, ki nam po vrsti povedo, katero ploščico moramo uporabiti naslednjo. Edino, na kar smemo mi pri sestavljanju poti vplivati, je to, ali posamezno ploščico tipa L uporabimo kot ovinek v levo ali kot ovinek v desno. **Opiši postopek**, ki ugotovi, kako moramo obrniti ovinke, da pri teh omejitvah sestavimo primerno sklenjeno pot (lahko se tudi zgodi, da je to sploh nemogoče; če pa je možnih več rešitev, je vseeno, katero od njih poišče tvoj postopek).

Nalogo lahko posplošimo tudi v tri dimenzije. Namesto kvadratnih ploščic imamo zdaj enotske kockice, črta skozi posamezno kockico pa povezuje središče kockice s središčema dveh njenih ploskev. Če imata tidve ploskvi skupno stranico, dobimo kockico tipa L, sicer pa kockico tipa I. Zdaj lahko podobne tri podnaloge kot zgoraj za dve dimenziji rešujemo tudi v treh dimenzijah:

(d) Razmisli o tem, kako bi zdaj opisali potek poti (podobno kot smo v dveh dimenzijah opisali pot z zaporedjem simbolov `N`, `L` in `D`). Za tako dobljeni pristop k opisovanju poti opiši postopek, ki bo za dani opis poti preveril, ali je ta pot veljavna.

(e) Če dobimo s kockic tipa I in t kockic tipa L, ali lahko iz njih sestavimo sklenjeno pot (in kako)?

(f) Reši prejšnjo podnalogo še z dodatno omejitvijo, da je predpisano, v kakšnem vrstnem redu moramo uporabiti kockice (torej kot vhodni podatek dobimo niz `I`-jev in `L`-jev, ki nam po vrsti povedo, katero kockico moramo uporabiti naslednjo). Edino, na kar lahko mi pri sestavljanju poti vplivamo, je to, v katero od štirih možnih smeri obrnemo posamezno od ploščic tipa L.

REŠITVE NALOG ZA PRVO SKUPINO

1. Delni izid

Označimo vhodno zaporedje z $a_1, a_2, \ldots, a_n$ — torej je bilo na tekmi doseženih n košev in število a_i pove, koliko točk (in za katero ekipo) je bilo doseženih pri i-tem košu. Delni izid zdaj ni nič drugega kot vsota nekega podzaporedja tega zaporedja, torej vsota oblike $a_i + a_{i+1} + a_{i+2} + \ldots + a_{j-1} + a_j$ za neka i in j. Če je delni izid v prid prve ekipe, je ta vsota pozitivna, če je v prid druge ekipe, pa je negativna. Nas bo pravzaprav zanimala le absolutna vrednost te vsote, saj naloga sprašuje po največjem možnem delnem izidu ne glede na to, kateri ekipi je v prid.

Zelo preprosta rešitev je torej ta, da gremo v zankah po vseh možnih začetnih indeksih i, vseh možnih končnih indeksih j in pri vsakem paru (i, j) izračunamo delni izid; med tako dobljenimi delnimi izidi pa si zapomnimo največjega:

```
naj := 0;
for i := 1 to n:
   for j := i to n:
      vsota := 0;
      for k := i to j:
         vsota := vsota + a_k;
      if |vsota| > naj then naj := |vsota|;
return naj;
```

Časovna zahtevnost tega postopka je kar $O(n^3)$, saj imamo tri gnezdene zanke, ki imajo po $O(n)$ iteracij. Spomnimo se lahko, da če pri istem i povečamo j za 1, je vsota zelo podobna kot prej, le na koncu pridobi en dodatni člen. Torej vsote ni treba računati vsakič od začetka, ampak jo lahko postopoma dopolnjujemo, ko povečujemo j:

```
naj := 0;
for i := 1 to n:
   vsota := 0;
   for j := i to n:
      vsota := vsota + a_j;
      (* Zdaj je spremenljivka „vsota“ enaka a_i + a_{i+1} + ... + a_j. *)
      if |vsota| > naj then naj := |vsota|;
return naj;
```

Časovna zahtevnost tega postopka je le še $O(n^2)$, saj imamo pri vsakem paru (i, j) le še konstantno mnogo dela, da popravimo vsoto in si jo (če je treba) zapomnimo v *naj*.

Še boljšo rešitev pa dobimo takole: delne vsote, ki smo jih dobili pri $i = 1$, si je koristno zapomniti; naj bo torej $s_j = a_1 + a_2 + \ldots + a_{j-1} + a_j$. Pri $j = 0$ si mislimo še $s_j = 0$. Če bi imeli vse te vsote shranjene v neki tabeli, bi lahko zelo poceni izračunali poljubno vsoto oblike $a_i + a_{i+1} + \ldots + a_{j-1} + a_j$ — ta vsota je preprosto enaka $s_j - s_{i-1}$. Naloga torej pravzaprav sprašuje, kakšna je največja možna $|s_j - s_{i-1}|$. Če zapišemo vse delne vsote kot zaporedje $s_0, s_1, s_2, \ldots, s_{n-1}, s_n$, je jasno, da bomo največjo razliko (po absolutni vrednosti) dosegli takrat, če za s_{i-1}

in s_j vzamemo največji in najmanjši člen tega zaporedja. To, ali je s_{i-1} najmanjši in s_j največji ali obratno, je pravzaprav nepomembno, saj bomo na koncu tako ali tako vzeli absolutno vrednost razlike med njima. Naš postopek mora torej le poiskati največjo in najmanjšo vrednost v zaporedju delnih vsot in vrniti razliko med njima:

```
min := 0; max := 0; vsota := 0;
for i := 1 to n:
  vsota := vsota + a_i;
  (* Zdaj je spremenljivka „vsota“ enaka s_i = a_1 + a_2 + ... + a_i. *)
  if vsota > max then max := vsota;
  if vsota < min then min := vsota;
return |max − min|;
```

Tu imamo torej le še eno zanko z n iteracijami, pri vsaki od njih pa konstantno mnogo dela, tako da je časovna zahtevnost tega postopka le še $O(n)$ in bi lahko učinkovito obdelal tudi zelo dolga vhodna zaporedja.

2. Kompresija

Naloga pravi, da moramo meritve sporočati naprej čim bolj sproti. Ko se odločamo o tem, ali bi trenutno meritev sporočili ali ne, lahko ločimo naslednje tri možnosti:

- če je trenutna meritev različna od prejšnje, jo vsekakor moramo sporočiti;
- če je trenutna meritev enaka prejšnji in smo prejšnjo že sporočili, potem trenutne rešitve ne smemo sporočiti;
- če pa je trenutna meritev enaka prejšnji in prejšnje nismo sporočili, potem trenutno rešitev moramo sporočiti.

Zadnji dve točki poskrbita, da od vsake skupine več zaporednih enakih meritev sporočimo prvo, tretjo, peto, sedmo in tako dalje; iz tega tudi sledi, da če je taka meritev lihe dolžine, bomo število sporočenih meritev pravilno zaokrožili navzgor, tako kot zahteva naloga (od petih meritev izpišemo tri, od sedmih meritev izpišemo štiri ipd.).

Koristno je torej, če naša rešitev poleg trenutne meritve hrani še prejšnjo meritev in še podatek o tem, ali smo prejšnjo meritev sporočili ali ne. Za to bi lahko uporabili dve spremenljivki, gre pa tudi z eno samo; v spodnjem programu je to spremenljivka `prejsnja`. Če prejšnje rešitve nismo sporočili, postavimo `prejsnja` na 0; pri naslednji meritvi (ki je zagotovo večja od 0 — to zagotavlja besedilo naloge) se nam bo torej zazdelo, da je drugačna od prejšnje, zato jo bomo zagotovo sporočili naprej. Zdaj sicer od prej omenjenih treh možnosti ne moremo razlikovati med možnostma 1 in 3, vendar to za nas niti ni pomembno, saj moramo v obeh primerih narediti isto stvar: sporočiti trenutno meritev naprej.

```
#include <stdbool.h>
int main()
{
  int meritev, prejsnja = 0;
  while (true)
  {
```

```
      meritev = Preberi();
      if (meritev == prejsnja)
         /* Trenutna meritev je enaka prejšnji in tisto smo že sporočili,
             zato trenutne ne smemo. To pa pomeni, da bomo naslednjo zagotovo
             morali sporočiti, zato postavimo spremenljivko prejsnja na 0. */
         prejsnja = 0;
      else {
         /* Trenutna meritev je različna od prejšnje (ali pa prejšnje nismo
             sporočili naprej), zato jo moramo sporočiti. */
         Sporoci(meritev);
         /* Zapomnimo si trenutno meritev v spremenljivki prejsnja; če ji bo
             naslednja meritev enaka, te naslednje ne bomo sporočili naprej. */
         prejsnja = meritev; }
   }
}
```

3. znajdi.se

Pri tej nalogi so imena krajev dolga le 1 črko in ta črka je velika črka angleške abecede, torej je možnih le 26 različnih krajev. Zato si lahko privoščimo tabelo, v kateri za vsak možen kraj od A do Z piše, kateri je njegov neposredni naslednik na naši poti. Spodnji program ima v ta namen tabelo nasl.

Na začetku preberimo začetni in končni kraj poti in si ju zapomnimo v spremenljivkah zac in kon; nato lahko postopoma beremo preostanek navodil in si v tabelo nasl shranjujemo podatke o poteku poti. Vse znake, ki niso velike črke, lahko kar preskočimo, saj za nas niso zanimivi. Velike črke pa nastopajo v parih, pri čemer prva pove začetni kraj na enem od korakov poti, druga pa končni kraj na tem koraku. Ko preberemo še drugo, si lahko v nasl zapomnimo, da je ona neposredna naslednica tiste prve. Prvo si med tem zapomnimo v spremenljivki od; ko pa preberemo še drugo veliko črko, postavimo od na -1, kar bo znak, da v nadaljevanju spet čakamo na prvo veliko črko v naslednjem paru (v naslednji vrstici navodil).

Na koncu se moramo le še z zanko sprehoditi po poti od začetnega kraja do končnega in jih sproti izpisovati; pri tem si pomagamo s tabelo nasl, da vemo, kako nadaljevati. Končni kraj prepoznamo po tem, da ima v tabeli nasl vrednost -1 (ker pač nima naslednika, saj je tam konec poti).

```
#include <stdio.h>

int main()
{
   int nasl[26], od = −1, c;
   char zac, kon;

   /* Preberimo ime začetnega in končnega kraja. */
   fscanf(stdin, "%c %c\n", &zac, &kon);
   zac −= 'A'; kon −= 'A';
   nasl[kon] = −1;

   /* Preberimo preostanek navodil znak po znak. */
   while ((c = fgetc(stdin)) != EOF)
   {
      /* Znake, ki niso velike črke, preskočimo. */
      if (c < 'A' || c > 'Z') continue;
```

```
      /* Če je od == −1, to pomeni, da je naslednja velika črka,
         ki jo bomo prebrali, ime začetnega kraja v trenutni vrstici. */
      if (od == −1) od = c − 'A';
      else {
         /* Sicer pa je naslednja prebrana velika črka ime končnega kraja v trenutni
            vrstici. Zdaj si lahko ta korak zapomnimo v tabeli nasl. */
         nasl[od] = c − 'A';
         /* Naslednja velika črka bo spet začetni kraj, zato postavimo „od“ spet na −1. */
         od = −1; }
   }

   /* Izpišimo potek poti. */
   while (zac != −1)
   {
      printf("%c", zac + 'A'); /* Izpišimo trenutni kraj. . . */
      zac = nasl[zac]; /* . . . in se premaknimo na naslednjega. */
   }
   return 0;
}
```

4. Dva od petih

Vhodne podatke lahko beremo znak po znak; prebrane ničle in enice si sproti zapomnimo v neki spremenljivki (vse znake, ki niso 0 ali 1, lahko sproti preskočimo). Ko se nabere pet takih znakov, moramo ugotoviti, katero števko predstavlja ta peterica, in jo izpisati; če pa se izkaže, da peterica ne predstavlja nobene od desetih števk, izpišemo *.

Podrobnosti tega postopka je mogoče izvesti na več načinov. Peterice lahko predstavimo z nizi oz. tabelami znakov; vseh deset veljavnih peteric lahko hranimo v tabeli in gremo po njej z zanko, da ugotovimo, kateri od njih (če sploh kateri) je enaka naša pravkar prebrana peterica; malo manj elegantna možnost je tudi, da imamo deset stavkov **if** (ali pa stavek **switch**) in z njimi za vsako od veljavnih peteric preverimo, če je naša enaka ravno njej. Še ena možnost pa je, da na peterice gledamo kot na cela števila, zapisana v dvojiškem sestavu. Iz vsake peterice tako nastane celo število od 0 (= dvojiško 00000) do 31 (= dvojiško 11111). Zdaj si lahko privoščimo tabelo, ki za vseh 32 možnih peteric pove, kateri znak moramo pri tisti peterici izpisati; pri desetih so to števke od 0 do 9, pri 22 neveljavnih petericah pa izpišemo zvezdico *.

Tako dobimo naslednjo rešitev: trenutno peterico hranimo v spremenljivki x, število prebranih bitov pa v b; ko le-ta doseže 5, vemo, da smo prebrali že celo peterico in moramo izpisati pripadajoči znak, ki ga dobimo iz tabele izpis. Slednjo smo pripravili s pomočjo tabele v besedilu naloge; na primer, tam piše, da ima števka 6 kodo 10001, kar je dvojiški zapis števila 17, zato mora biti izpis[17] enak '6' ipd.

```
#include <stdio.h>
int main()
{
                 /* 01234567890123456789012345678901 */
   char izpis[ ] = "***9*85**74*0****63*2***1*******";
   int c, x = 0, b = 0;
```

```
  /* Berimo vhodne podatke znak po znak. */
  while ((c = fgetc(stdin)) != EOF && c != '\n')
  {
    /* Znake, ki niso 0 ali 1, preskočimo. */
    if (c != '0' && c != '1') continue;
    /* Dodajmo pravkar prebrani bit v x. */
    x <<= 1; if (c == '1') x |= 1;
    /* Po vsakem petem prebranem bitu dekodirajmo prebrano peterico
       in izpišimo ustrezni znak. */
    if (++b == 5) {
      fputc(izpis[x], stdout);
      /* Pripravimo se na branje naslednje peterice. */
      x = 0; b = 0; }
  }

  return 0;
}
```

5. Kontrolne vsote

Razmislimo najprej o funkciji Beri2, ki je lažji del naloge. Za začetek lahko poskusimo zahtevani bit prebrati kar s funkcijo Beri; če se to branje posreči, vrnemo prebrano vrednost in smo končali. Če pa to branje spodleti, moramo v zanki prebrati istoležne celice na vseh ostalih straneh in iz njih rekonstruirati vrednost, ki nas zanima. Če spodleti tudi kakšno od teh branj, iskane vrednosti ne moremo rekonstruirati in nam ne preostane kaj dosti drugega, kot da vrnemo -1.

Kako naj iz vrednosti na ostalih straneh rekonstruiramo tisto, ki nam je ni uspelo prebrati? Naj bo x_i (za $i = 0, \ldots, n-1$) vrednost celice na strani i in na naslovu, ki nas zanima (naša funkcija Beri2 ga je dobila v parametru naslov). Naloga pravi, da so kontrolne vsote definirane takole: če je $x_1 + x_2 + \ldots + x_{n-1}$ sodo število, je $x_0 = 0$, sicer pa je $x_0 = 1$. Isto pravilo pa lahko zelo elegantno zapišemo tudi takole: x_0 ima táko vrednost (izmed 0 in 1), da je $x_0 + x_1 + x_2 + \ldots + x_{n-1}$ v vsakem primeru sodo število.

Recimo zdaj, da iščemo x_s za neko stran s (pri čemer je $1 \leq s < n$). Lahko torej izračunamo vsoto $x_0 + \ldots + x_{s-1} + x_{s+1} + \ldots + x_{n-1}$ (recimo ji y), v kateri manjka le naša iskana vrednost x_s. Vemo, da mora biti $y + x_s$ sodo število; torej, če je y lih, mora biti $x_s = 1$, sicer pa mora biti $x_s = 0$. Vidimo torej, da iskani x_s ni nič drugega kot ostanek po deljenju y z 2; ali pa še drugače: x_s je enak najnižjemu bitu števila y.

Zelo elegantna možnost pa je, da namesto seštevanja uporabimo operacijo xor (v C/C++ in sorodnih jezikih jo dobimo z operatorjem ^). Spomnimo se, kako deluje xor na dveh bitih: če nek bit xor-amo z 0, se ne spremeni, če pa ga xor-amo z 1, se obrne (iz 0 v 1 ali obratno). Če torej začnemo z bitom 0 in ga po vrsti xor-amo s števili $x_0, x_1, \ldots, x_{s-1}, x_{s+1}, \ldots, x_{n-1}$, bo na koncu prižgan, če je bilo med temi števili liho mnogo enic, sicer pa bo na koncu ugasnjen. Tako torej vidimo, da bo rezultat tega xor-anja na koncu ravno enak vrednosti x_s, ki jo iščemo. Zapišimo dobljeno rešitev v C-ju:

```
int Beri2(int stran, int naslov)
{
  int i, x, y;
```

```
  /* Poskusimo prebrati zahtevano celico. */
  x = Beri(stran, naslov);

  /* Če se je branje posrečilo, vrnimo njeno vrednost. */
  if (x >= 0) return x;

  /* Sicer preberimo istoležne celice na vseh ostalih straneh;
     xor njihovih vrednosti je ravno vrednost celice, ki nas zanima. */
  y = 0;
  for (i = 0; i < n; i++) if (i != stran)
  {
    x = Beri(i, naslov);

    /* Če spodleti branje kakšne od istoležnih celic, ne bomo mogli
       rekonstruirati vrednosti iskane celice. */
    if (x < 0) return −1;
    y ^= x;
  }
  return y;
}
```

Funkcija Pisi2 mora zapisati novo vrednost v zahtevano celico (kar lahko stori preprosto tako, da kliče funkcijo Pisi) in popraviti kontrolno vsoto na istoležni celici strani 0. Po vsem, kar smo doslej videli o kontrolnih vsotah, lahko razmišljamo takole: če se je trenutna celica s tem pisanjem spremenila (iz 0 v 1 ali obratno), se je s tem spremenila tudi parnost vsote $x_1 + x_2 + \ldots + x_{n-1}$, zato se mora spremeniti tudi kontrolna vsota x_0: če je bila le-ta prej 0, mora zdaj postati 1 in obratno. Če pa se trenutna celica pri pisanju ni spremenila (ker je bila novaVrednost enaka dosedanji vrednosti te celice), se tudi kontrolna vsota ne sme spremeniti.

Zametek funkcije Pisi2 je torej nekaj takšnega:

```
void Pisi2(int stran, int naslov, int novaVrednost)
{
  int staraVrednost, staraVsota;

  /* Preberimo staro vrednost celice. */
  staraVrednost = Beri(stran, naslov);

  /* Zapišimo novo vrednost. */
  Pisi(stran, naslov, novaVrednost);

  /* Če se vrednost ni spremenila, tudi kontrolne vsote ni treba spreminjati. */
  if (staraVrednost == novaVrednost) return;

  /* Sicer preberimo staro kontrolno vsoto. */
  staraVsota = Beri(0, naslov);

  /* In zapišimo novo kontrolno vsoto. */
  Pisi(0, naslov, 1 − staraVsota);
}
```

Toda v tej rešitvi dvakrat kličemo Beri; kaj se zgodi, če pri kakšnem od teh branj pride do napake? Če pride do napake pri branju Beri(stran, naslov), še ni treba takoj obupati, saj lahko poskusimo staro vrednost naše celice rekonstruirati iz kontrolne vsote in istoležnih celic na drugih straneh. To je ista stvar, ki jo že počne naša funkcija Beri2, zato lahko za branje uporabimo kar njo. Če spodleti tudi njej, potem vemo, da stare vrednosti naše celice ne moremo rekonstruirati (ker je okvarjena poleg nje še vsaj ena druga istoležna celica), zato se nam tudi s popravljanjem

kontrolne vsote ni treba ukvarjati (če imamo dve okvari na istem naslovu, nam tudi kontrolna vsota ne more več pomagati, saj z njo pri kasnejših branjih ne bomo mogli rekonstruirati vrednosti okvarjenih celic).

Če pa pride do napake pri branju Beri(0, naslov), to pomeni, da je okvarjena celica s kontrolno vsoto. Načeloma bi lahko tudi tu klicali Beri2, ki bi v takem primeru uspešno rekonstruiral staro kontrolno vsoto iz vrednosti istoležnih celic na straneh od 1 do $n - 1$. Toda če je celica, v kateri bi morali hraniti kontrolno vsoto, okvarjena, si lahko mislimo, da ni posebne koristi od tega, da poskušamo računati novo kontrolno vsoto in jo vpisovati tja. Zato spodnja rešitev staro kontrolno vsoto vseeno bere kar z Beri namesto Beri2 in če to branje spodleti, kontrolne vsote ne poskuša popravljati.

```
void Pisi2(int stran, int naslov, int novaVrednost)
{
  int staraVrednost, staraVsota;
  /* Preberimo staro vrednost celice. */
  staraVrednost = Beri2(stran, naslov);
  /* Zapišimo novo vrednost. */
  Pisi(stran, naslov, novaVrednost);
  /* Če stare vrednosti nismo mogli rekonstruirati, tudi kontrolne vsote
     ne moremo popraviti; če pa je stara vrednost enaka novi, potem
     kontrolne vsote ni treba spreminjati. V obeh primerih se lahko takoj vrnemo. */
  if (staraVrednost < 0 || staraVrednost == novaVrednost) return;
  /* Preberimo staro kontrolno vsoto. */
  staraVsota = Beri(0, naslov);
  /* Če je branje uspelo, zapišimo novo kontrolno vsoto. */
  if (staraVsota >= 0) Pisi(0, naslov, 1 − staraVsota);
}
```

Mimogrede lahko še omenimo, da bi si lahko tudi pri računanju nove kontrolne vsote pomagali z operacijo xor. Recimo, da istoležne celice na vseh straneh spet označimo z $x_0, \ldots, x_{n-1}$, pri čemer je x_0 kontrolna vsota, torej je $x_0 = x_1 \operatorname{xor} \ldots \operatorname{xor} x_{n-1}$. Če se v izrazu na desni člen x_s spremeni v $\hat{x}_s$, se leva stran spremeni v $x_0 \operatorname{xor} x_s \operatorname{xor} \hat{x}_s$. O tem se lahko prepričamo, če upoštevamo lastnosti operacije xor: je komutativna in za poljuben a velja $a \operatorname{xor} a = 0$ in $a \operatorname{xor} 0 = a$.

REŠITVE NALOG ZA DRUGO SKUPINO

1. It's raining cubes

Nobene koristi ni od tega, da bi se premikali malo levo in malo desno; recimo namreč, da se najprej premikamo malo desno, nato pa v nekem trenutku naredimo korak v levo, z x na $x-1$. Vprašanje je, zakaj nismo že kar prej, ko smo že bili na $x-1$, tam tudi počakali, namesto da smo šli naprej na x (in zdaj z x nazaj na $x-1$). Edini možen razlog je, da je na $x-1$ medtem padla kocka in smo se ji s premikom na x izognili, da nas ni ubila; toda če je tako, je ta kocka še zdaj tam na $x-1$ in se tja zdaj sploh ne moremo premakniti. Ta razlog torej ne pride v poštev. Podobno bi se lahko prepričali, da če smo se najprej premikali v levo, nima smisla potem narediti koraka v desno.

Podobno tudi vidimo, da v resnici nima smisla čakati na nekem polju x in kasneje nadaljevati poti. Zakaj bi čakali tam (in stali pri miru, namesto da nadaljujemo z gibanjem v isto smer kot doslej)? Mar zato, ker vidimo, da bo vsak hip padla kocka na $x+1$, pa ne bi radi, da nas ubije? Če je tako, bo ta kocka potem tako ali tako ostala na $x+1$ in nam preprečila, da bi se še kdaj premaknili tja; torej je vseeno, če kar končamo naš sprehod in trajno ostanemo na x.

Tako torej vidimo, da se lahko omejimo na takšne vzorce gibanja, pri katerih se najprej nekaj časa premikamo (ves čas v isto smer, brez postankov), nato pa na nekem polju obstanemo in se odtlej ne premikamo več.

Katera polja pa lahko na ta način dosežemo? Naloga pravi, da ob času i začne na x-koordinati x_i in na višini h padati kocka, ki pristane na tleh ob času $i+h$. Ker se mi začnemo premikati ob času 0 na x-koordinati 0 in se na vsakem koraku premaknemo le za 1 mesto levo ali desno, do koordinate x_i ne moremo priti prej kot ob času $|x_i|$. Torej, če je $|x_i| \geq i+h$, nam bo ta kocka preprečila, da bi prišli do polja x_i in sploh do vsakega polja, ki je v tisti smeri oddaljeno od našega začetnega položaja ($x=0$) za vsaj $|x_i|$. Glede vseh ostalih polj pa velja, da nas ta kocka pri dostopu do njih ne bo nič ovirala.

Če zdaj za vsako kocko opravimo takšen razmislek, nam ostane nek interval možnih x-koordinat, ki so nam načeloma dosegljive; recimo $L < x < D$ za neka L in D. Če za vsaj eno koordinato na tem intervalu velja, da nanj nikoli ne pade nobena kocka, se lahko na začetku premaknemo nanj in tam počakamo do konca igre; če pa takega mesta ni, potem vemo, da ne bomo mogli preživeti.

Zapišimo dobljeni postopek še s psevdokodo:

$L := -\infty$; $D := \infty$;
for $i := 1$ **to** n:
  if $|x_i| < i+h$ **then continue**;
  if $x_i \geq 0$ **and** $x_i < D$ **then** $D := x_i$;
  if $x_i \leq 0$ **and** $x_i > L$ **then** $L := x_i$;
if obstaja tak x, ki je različen od vseh x_i in leži na $L < x < D$ **then** ($\star$)
  lahko preživimo: v prvih $|x|$ korakih se premaknemo na x in tam obstanemo;
else
  ne moremo preživeti;

Razmislimo še o tem, kako preveriti pogoj v vrstici ($\star$). Ena možnost je, da je interval neomejen; na primer, če je $L = -\infty$, lahko pridemo poljubno daleč proti

levi, torej gremo lahko na primer na $x = \min_i x_i - 1$, kjer bomo levo od vseh kock, in tam počakamo konec igre. Podobno, če je $D = \infty$, lahko gremo na $x = \max_i x_i + 1$.

Še en poseben primer je, če je interval prazen (kar prepoznamo po tem, da je $D \leq L$); takrat lahko takoj zaključimo, da primernega x ni in da ne bomo preživeli.

Drugače pa moramo najti nekakšno vrzel med dvema kockama (ali pa med kocko in enim od krajišč L in D), torej območje, na katerega ne pade nobena kocka. Lahko si na primer koordinate vseh kock, ki padejo na območju $L < x_i < D$, zapišemo v neko tabelo; dodajmo vanjo še števili L in D; tabelo uredimo; v tako urejeni tabeli zdaj iščemo dve zaporedni števili, ki se razlikujeta za več kot 1. Če najdemo kakšen tak primer, to pomeni, da je tam med dvema kockama (ali pa med neko kocko in L ali D) vrzel, kamor se lahko premaknemo in tam počakamo konec igre.

Časovna zahtevnost tega postopka je $O(n \log n)$, zaradi urejanja; gre pa tudi brez urejanja. Vse kocke x_i, ki ležijo na $L < x_i < D$, zložimo v razpršeno tabelo; v tej razpršeni tabeli lahko zdaj v času $O(1)$ za poljubno x-koordinato preverimo, ali na njej leži kakšna kocka ali ne. Preverimo to za vsak $x = x_i + 1$ (za tiste x_i, ki ležijo na $L < x_i < D - 1$) in še za $x = L + 1$. Če je vsaj ena od teh koordinat prosta (na njej ne leži nobena kocka), se lahko premaknemo tja in preživimo, sicer pa je naš položaj brezupen. Tako imamo $O(n)$ poizvedb v razpršeno tabelo, tako da je časovna zahtevnost tega postopka le še $O(n)$.

2. Strahopetni Hektor

Za začetek se dogovorimo, kako bomo uporabljali koordinatni sistem. Če neka enota leži na x-koordinati a (za neko celo število a), kaj točno to pomeni? Ena možna interpretacija je, da je to x-koordinata središča enote; ta enota torej pokriva x-koordinate od $a - 1/2$ do $a + 1/2$; težišče bloka z začetkom z_i in dolžino d_i je tedaj $z_i + (d_i - 1)/2$. Druga interpretacija je, da a pomeni x-koordinato levega roba enote; ta enota torej pokriva x-koordinate od a do $a + 1$; težišče bloka z začetkom z_i in dolžino d_i je tedaj $z_i + d_i/2$. Načeloma je vseeno, katero interpretacijo si izberemo, važno je le, da se je potem dosledno držimo pri vseh izpeljavah in izračunih. V naši rešitvi bomo uporabljali prvo interpretacijo.

Označimo s t_i položaj (x-koordinato) težišča tistega dela zidu, ki ga tvorijo bloki na višinah od vključno i do vključno n. Težišče je povprečje x-koordinat vseh enot v teh blokih, torej ga lahko zapišemo kot $t_i = s_i/b_i$, pri čemer je s_i vsota x-koordinat vseh enot v teh blokih, b_i pa je število teh enot. Vse te stvari lahko računamo preprosto in učinkovito, če gremo od vrha zidu navzdol. Pri $i = n + 1$, kar je že nad našim zidom, si mislimo $s_{n+1} = b_{n+1} = 0$. Recimo zdaj, da za nek i že poznamo s_{i+1} in b_{i+1}; kako bi izračunali s_i in b_i? Vse enote, ki so prišle v poštev za s_{i+1} in b_{i+1}, moramo šteti tudi v s_i in b_i, dodati pa jim moramo še vse enote bloka i. Teh je d_i, torej je $b_i = b_{i+1} + d_i$; in njihove koordinate so $z_i, z_i + 1, \ldots, z_i + d_i - 1$, torej je $s_i = s_{i+1} + z_i + (z_i + 1) + \ldots + (z_i + d_i - 1) = s_{i+1} + d_i \cdot (z_i + (d_i - 1)/2)$. Naš postopek gre lahko med inicializacijo v zanki po blokih od zgoraj navzdol in izračuna vse s_i, b_i in tudi težišča t_i.

Naslednje vprašanje je, kako te podatke vzdrževati, ko krogle podirajo zid in se njegova oblika spreminja. Recimo, da zadene krogla blok na višini v. V njem torej poruši najbolj levo enoto, zato se z_v poveča za 1; in blok je zdaj za eno enoto krajši kot prej, zato se d_v zmanjša za 1. Ta blok je prispeval k vsotam s_i in b_i za vse i od

1 do v, zato jih moramo zdaj ustrezno zmanjšati: vsak tak s_i se zmanjša za (staro vrednost) z_v (to je bila namreč koordinata pravkar porušene enote), vsak b_i pa za 1. Ko tako popravimo vse s_i in b_i, lahko na novo izračunamo tudi težišča t_i.

Ko pa poznamo vsa težišča, ni težko preveriti, če zid še stoji. Blok i sestavljajo enote s koordinatami od vključno z_i do vključno $z_i + d_i - 1$, torej ta blok pokriva interval x-koordinat $[z_i - 1/2, z_i + d_i - 1/2]$. Na tem intervalu mora ležati težišče višje ležečega dela zidu (torej dela, ki ga tvorijo bloki od $i + 1$ do n), sicer se bo ta del nagnil in prevrnil. Pogoj za stabilnost je torej, da pri vsakem i (od 1 do $n - 1$) velja $z_i - 1/2 \leq t_{i+1} \leq z_i + d_i - 1/2$ (težišča t_1 nam ni treba preverjati, saj blok 1 leži na tleh in je torej težišče v vsakem primeru podprto), poleg tega pa mora seveda še pri vsakem i (od 1 do n) veljati $d_i > 0$ (da ni kakšen blok popolnoma uničen).

Časovna zahtevnost našega postopka je: $O(n)$ za inicializacijo in nato še $O(n)$ po vsaki izstreljeni krogli (toliko časa potrebujemo tako za izračun novih s_i, b_i, t_i kot za preverjanje, če zid še stoji). Ni pa nujno, da hranimo vrednosti s_i, b_i in jih popravljamo po vsaki krogli. Lahko bi le popravili z_v in d_v ter nato čisto na novo izračunali vse s_i, b_i in t_i po enakem postopku kot med inicializacijo. Časovna zahtevnost takšne rešitve bi bila še vedno le $O(n)$ za vsak strel.

Razmislimo še o težji različici naloge, ki jo omenja opomba pod črto: za dano začetno stanje zidu (števila n, $z_1, \ldots, z_n$, $d_1, \ldots, d_n$) nas zanima, kako ga s čim manj streli porušiti. Ločimo lahko tri razloge, zakaj se zid podre: (a) ker neko vrstico v celoti porušimo; (b) ker težišče nekega dela zidu (zgornjih nekaj vrstic) ni več podprto v naslednji vrstici in se ta del zidu nagne v levo; (c) ker težišče zgornjega dela zidu ni več podprto v naslednji vrstici in se ta del zidu nagne v desno. Razmislili bomo o vsaki od teh treh možnosti posebej in na koncu uporabili tisto, ki poruši zid z najmanj streli.

(a) Če hočemo neko vrstico porušiti v celoti, je stvar preprosta: vrstica i je dolga d_i enot in da jo porušimo, potrebujemo pač d_i strelov. Med temi možnostmi izberimo najmanjšo; tako je torej $\min_i d_i$ kandidat za najmanjše možno število strelov, s katerim je mogoče porušiti zid.

(b) Recimo zdaj, da bi radi dosegli, da se zgornjih nekaj vrstic zidu nagne, ker njihovo težišče ni več podprto v naslednji vrstici. Spomnimo se oznak, ki smo jih vpeljali zgoraj: b_v je skupno število enot v vseh blokih od v do n; vsota njihovih x-koordinat je s_v, težišče te skupine blokov pa je $t_v = s_v/b_v$. Če hočemo zid destabilizirati, moramo (s čim manj streli) doseči, da nek blok v ne bo več podpiral težišča blokov nad njim; doseči moramo torej, da bo $t_{v+1} < z_v - 1/2$ (takrat se zgornji del zidu — tisti, ki ga tvorijo bloki od $v + 1$ do n — nagne na levo; primer tega vidimo na sliki pod besedilom naloge) ali pa $t_{v+1} > z_v + d_v - 1/2$ (takrat se zgornji del zidu nagne desno).

Za začetek razmislimo o tem, kako bi s čim manj streli dosegli, da se nek del zidu nagne v levo. Recimo, da smo to naredili na optimalen način in naj bo v tista vrstica, ki zdaj ne podpira več težišča vrstic od $v + 1$ do n; pri tem v torej velja $t_{v+1} < z_v - 1/2$. Kaj lahko povemo o strelih, ki so pripeljali do tega stanja zidu? Gotovo ni bil nobeden od teh strelov usmerjen v vrstice od 1 do $v - 1$, saj taki streli ne vplivajo niti na z_v niti na t_{v+1}, torej ne bi k našemu rušenju zidu nič pripomogli in bi brez njih dobili še boljšo rešitev (z manj streli), ki bi ravno tako dosegla, da se vrstice od $v + 1$ naprej nagnejo v levo.

Gotovo so torej vsi streli bili v vrstice od v do n. Pri njih lahko razmišljamo takole: doseči skušamo pogoj $t_{v+1} < z_v - 1/2$, kar lahko zapišemo tudi kot $e_v < 0$ za $e_v = t_{v+1} - z_v + 1/2$. Število e_v torej pove, koliko še manjka težišču t_{v+1} do levega roba vrstice v. Kako vplivajo streli na e_v? Če ustrelimo v vrstico v, se e_v zmanjša za 1 (kajti pri takem strelu se z_v poveča za 1, težišče t_{v+1} pa se ne spremeni). Če pa ustrelimo v eno od vrstic od $v + 1$ do n, se z_v ne spremeni, lahko pa se spremeni težišče t_{v+1}. Ali je mogoče, da bi kak tak strel zmanjšal t_{v+1} za več kot 1?

Recimo, da smo ustrelili v vrstico u. Potem so se stvari spremenile takole (nove vrednosti po strelu označimo s črtico $'$): $b'_{v+1} = b_{v+1} - 1$, $s'_{v+1} = s_{v+1} - z_u$ (pri tem z_u pomeni stari začetek vrstice u, pred strelom) in $t'_{v+1} = s'_{v+1}/b'_{v+1}$ Če naj bo t'_{v+1} za več kot 1 manjši od t_{v+1}, imamo pogoj $t'_{v+1} < t_{v+1} - 1$, kar lahko z nekaj telovadbe predelamo v $t_{v+1} < z_u - b_{v+1} + 1$.

Naj bo $L_{v+1} = \max\{z_i : v < i \leq n\}$ najbolj leva x-koordinata po vseh enotah v opazovanem delu zidu, $D_{v+1} = \min\{z_i + d_i - 1 : v < i \leq n\}$ pa najbolj desna. Za vsako enoto v vrsticah od $v + 1$ do n torej velja $L_{v+1} \leq x \leq D_{v+1}$, zato velja to tudi za njihovo povprečje, to pa je ravno težišče t_{v+1}; torej je $L_{v+1} \leq t_{v+1} \leq D_{v+1}$. Podobno tudi za z_u, ki je koordinata neke konkretne enote v tem delu zidu (namreč najbolj leve v u-ti vrstici), velja $z_u \leq D_{v+1}$.

Če zdaj v prej dobljenem pogoju $t_{v+1} < z_u - b_{v+1} + 1$ upoštevamo, da je $L_{v+1} \leq t_{v+1}$ in $z_u \leq D_{v+1}$, dobimo iz njega pogoj $L_{v+1} < D_{v+1} - b_{v+1} + 1$ oz. $b_{v+1} < D_{v+1} - L_{v+1} + 1$.

Po drugi strani vemo, da naše vrstice od $v + 1$ do n pokrivajo vse x-koordinate od vključno L_{v+1} do vključno D_{v+1}; na vsaki od teh x-koordinat mora biti najmanj ena enota; skupaj mora biti torej enot vsaj $D_{v+1} - L_{v+1} + 1$. Vseh enot pa je b_{v+1}, torej je gotovo $b_{v+1} \geq D_{v+1} - L_{v+1} + 1$; to pa je v protislovju z zaključkom s konca prejšnjega odstavka.

Tako torej vidimo, da nas je predpostavka, da bi nek strel lahko zmanjšal t_{v+1} za več kot 1, pripeljala v protislovje. Posamezni strel v vrstice od $v + 1$ do n lahko torej e_{v+1} zmanjša kvečjemu za 1, lahko pa tudi za manj (lahko ga celo poveča, ker se lahko težišče t_{v+1} ob takem strelu premakne tudi v desno). Prej pa smo videli, da posamezni strel v vrstico v vedno zmanjša e_{v+1} točno za 1. Torej ni nobene koristi od tega, da bi kdaj streljali v vrstice od $v + 1$ do n, saj s takimi streli ne bomo mogli zidu spodkopati (torej doseči pogoja $e_v < 0$) nič hitreje kot z enakim številom strelov v vrstico v.

Pri razmišljanju o tem, kako bi zid pripravili do tega, da se nagne v levo, se smemo torej omejiti na scenarije, pri katerih so vsi streli usmerjeni v vrstico v. Zdaj pa tudi ni težko izračunati, koliko strelov potrebujemo: izračunajmo začetno vrednost $e_v = t_{v+1} - z_v + 1/2$; če se pri vsakem strelu zmanjša za 1, bo padla pod 0 po $\lfloor e_v \rfloor + 1$ strelih. Ta razmislek opravimo pri vsakem v in si zapomnimo tisto rešitev, ki zahteva najmanj strelov.

(c) Za konec nam ostane še možnost, da z nekaj streli dosežemo, da se nek del zidu nagne v desno. Najbolj desna enota vrstice v se konča pri x-koordinati $z_v + d_v - 1/2$ in višje ležeče vrstice se bodo nagnile na desno, če njihovo težišče t_{v+1} leži bolj desno od tega: $t_{v+1} > z_v + d_v - 1/2$. Podobno kot pri ($b$) vidimo, da streli pod vrstico v na ta pogoj ne vplivajo, zato jih lahko odmislimo; vendar pa zdaj tudi streli v vrstico v ne vplivajo na naš pogoj, saj se težišče t_{v+1} zaradi njih ne

spremeni, desna stran $z_v + d_v - 1/2$ pa se tudi ne spremeni (ker v vrstico streljamo z leve, se njen desni rob ne premakne; pri vsakem strelu vanjo se sicer z_v poveča za 1, vendar pa se tudi d_v zmanjša za 1).

Vprašanje je torej, kako s čim manj streli v vrstice od $v+1$ do n premakniti težišče t_{v+1} dovolj daleč proti desni. Recimo, da smo pripravljeni izstreliti k strelov; kako to vpliva na težišče t_{v+1}? Spomnimo se, da je bilo staro težišče $t_{v+1} = s_{v+1}/b_{v+1}$, pri čemer je imenovalec b_{v+1} skupno število enot v vrsticah od $v + 1$ do n, števec s_{v+1} pa je vsota njihovih x-koordinat. Z našimi k streli smo k enot uničili, zato se števec zmanjša za vsoto x-koordinat uničenih enot; recimo ji $\triangle_{v+1,k}$. Imenovalec pa se preprosto zmanjša za k, ker imamo zdaj pač k enot manj. Novo težišče je torej $t'_{v+1} = (s_{v+1} - \triangle_{v+1,k})/(b_{v+1} - k)$. Edina stvar, na katero lahko v tej formuli vplivamo (če je k fiksen), je $\triangle_{v+1,k}$; da bo novo težišče čim dlje proti desni, mora biti t'_{v+1} čim večji, torej mora biti $\triangle_{v+1,k}$ čim manjši; torej moramo naših k strelov usmeriti v take vrstice, da bomo uničili najbolj levih k enot (torej enot z najmanjšo x-koordinato).

Zdaj moramo razmisliti o tem, kako za dani k (in v) izračunati $\triangle_{v+1,k}$ (iz tega bomo potem lahko izračunali težišče in preverili, ali bi se zid pri tem k že podrl ali ne) in kako potem poskati najmanjši k, pri katerem bi se zid podrl.

Uredimo začetne in končne koordinate vseh vrstic (torej vse z_i in $z_i + d_i$) naraščajoče in jih v tem vrstnem redu oštevilčimo: $x_0, x_1, \ldots, x_t$. Razpon x-koordinat, ki jih pokriva naš zid, smo tako razdelili na t intervalov oblike $I_j = [x_j, x_{j+1}]$; naj bo $w_j = x_{j+1} - x_j$ širina intervala I_j.

Razpon koordinat, ki jih pokriva posamezna vrstica, torej $[z_i, z_i + d_i]$, je unija enega ali več zaporednih intervalov I_j. Naj bo h_j število vrstic (izmed vrstic od $v + 1$ do n), ki pokrivajo interval I_j. Zdaj vemo, da je na intervalu I_j vsega skupaj $\beta_j := h_j \cdot w_j$ enot, od tega ima najbolj levih h_j enot koordinato x_j, naslednjih h_j enot ima koordinato $x_j + 1$ in tako naprej, vse do zadnjih (najbolj desnih) h_j enot, ki imajo koordinato $x_{j+1} - 1$. Vsota x-koordinat vseh enot na tem intervalu je torej $\sigma_j := h_j \cdot w_j(x_j + x_{j+1} - 1)/2$. Podobno lahko izpeljemo tudi formulo za vsoto x-koordinat najbolj levih nekaj enot s tega intervala (namesto vseh $h_j w_j$ enot); podrobnosti prepustimo bralcu za vajo.

Zdaj si lahko predstavljamo takšen postopek za iskanje najmanjšega primernega k-ja:

1 postavi vse h_j na 0;
2 **for** $v := n - 1$ **downto** 1:
3 za vsak j, pri katerem vrstica $v + 1$ pokriva interval I_j,
  povečaj h_j za 1;
4 z zanko po naraščajočih j poišči najmanjši tak j (od 0 do $t - 1$),
  pri katerem se zid poruši, če uničimo v vrsticah $v + 1, \ldots, n$ vse
  enote iz intervalov $I_0, \ldots, I_j$;
5 **if** takega j ni **then continue**;
6 izračunaj najmanjše število strelov k (v vrstice $v + 1, \ldots, n$), s katerimi
  lahko zid porušimo;
7 če je to najmanjši k doslej, si ga zapomnimo;

Nekaj korakov tega postopka si je primerno ogledati še malo podrobneje. V koraku 4 lahko razmišljamo takole: v intervalih $I_0, \ldots, I_j$ je skupno $\beta_0 + \ldots + \beta_j$ enot

(recimo tej vsoti β), vsota njihovih x-koordinat pa je $\sigma_0 + \ldots + \sigma_j$ (recimo temu σ). (Posamezne β_i in σ_i lahko računamo sproti iz pripadajočih h_i, lahko pa jih tudi hranimo v neki tabeli in jih na novo izračunamo le takrat, ko se v koraku 3 kakšna od h_i spremeni.) Takrat torej vemo, da če bi z β streli uničili najbolj levih β enot v vrsticah $v+1, \ldots, n$, bi se težišče teh vrstic zato premaknilo v $(s_{v+1} - \sigma)/(b_{v+1} - \beta)$; nato moramo le še preveriti, če je to $> z_v + d_v - 1/2$ (takrat se zid nagne na desno).

V koraku 6 vemo, da če uničimo (v vrsticah $v + 1, \ldots, n$) vse enote iz intervalov $I_0, \ldots, I_{j-1}$, se zid še ne bo nagnil; pač pa se bo nagnil, če nato uničimo še vse enote iz I_j. Število enot na intervalih $I_0, \ldots, I_{j-1}$ označimo z β $(= \beta_0 + \ldots + \beta_{j-1})$, vsoto njihovih x-koordinat pa s σ $(= \sigma_0 + \ldots + \sigma_{j-1})$. Najmanjši k (pri trenutnem v), ki poruši naš zid, torej leži nekje na območju $\beta < j \leq \beta + \beta_j$. Spomnimo se, da je na tem območju, na intervalu I_j, najprej (če gledamo od leve proti desni) h_j enot z x-koordinato x_j, nato h_j enot s koordinato $x_j + 1$ in tako dalje, vse do zadnjih h_j enot z x-koordinato $x_{j+1} - 1$. Naš k lahko zapišemo v obliki $k = \beta + c \cdot h_j + r$ za neka $0 \leq c < w_j$ in $0 \leq r < h_j$ (ali pa $c = w_j$ in $r = 0$, kar predstavlja primer, ko porušimo čisto vse enote z intervala I_j); ta izražava nam pove, da bomo z intervala I_j v celoti uničili vse enote v levih c stolpcih (na x-koordinatah od x_j do $x_j + c - 1$), nato pa še r (od h_j) enot iz naslednjega stolpca (na x-koordinati $x_j + c$).

Omejimo se za začetek na primere, ko je $r = 0$; tedaj lahko zapišemo težišče kot funkcijo c-ja: $t(c) = (s_{v+1} - \sigma - h_j c(x_j + (c-1)/2))/(b_{v+1} - \beta - h_j c)$. Zanima nas najmanjši c, pri katerem je $t(c) > z_v + d_v - 1/2$. Z nekaj premetavanja lahko to neenačbo zapišemo v obliki $Uc^2 + Vc + W > 0$, pri čemer je koeficient pri kvadratnem členu negativen: $U = -h_j/2$. Taka kvadratna funkcija je torej večja od 0 med svojima dvema ničlama, tako da bomo najmanjši c dobili tako, da vzamemo levo ničlo in poiščemo prvo celo število, ki je večje od nje.

Za ta c torej vemo, da če porušimo vse enote iz intervalov $I_0, \ldots, I_{j-1}$ in še levih c stolpcev intervala I_j, se bo zid nagnil v desno; in da, če bi iz iz I_j porušili le levih $c - 1$ stolpcev, bi zid še stal. Zdaj moramo le še preveriti, če je mogoče zid porušiti že tako, da c-tega stolpca ne porušimo v celoti, ampak od njegovih h_j enot porušimo le r enot (za nek r z območja $1 \leq r < h_j$). Zdaj lahko zapišemo težišče kot funkcijo r-ja: $t(r) = (s_{v+1} - \sigma - h_j(c-1)(x_j + (c-2)/2) - r(x_j + c - 1))/(b_{v+1} - \beta - (c-1)h_j - r)$. Pogoj $t(r) > z_v + d_v - 1/2$ nas zdaj pripelje do linearne neenačbe oblike $Ur + V > 0$, pri čemer je vodilni koeficient $U = (z_v + d_v - 1/2) - (x_j + c - 1)$ zanesljivo > 0 (zakaj, lahko bralec razmisli za vajo); najmanjši primerni celoštevilski r je torej $1 + \lfloor -V/U \rfloor$.

Kakšna je časovna zahtevnost tako dobljenega postopka? Vseh intervalov skupaj je $O(n)$; vsako izvajanje koraka 3 in 4 zato vzame po $O(n)$ časa; vsako izvajanje koraka 6 vzame le $O(1)$ časa, saj smo pravkar videli, da moramo le rešiti eno kvadratno in eno linearno neenačbo. Vse te korake moramo izvesti $O(n)$-krat (zaradi zanke po v v koraku 2), kar nam dá skupaj časovno zahtevnost $O(n^2)$.

Hitrejši postopek lahko dobimo, če naše intervale organiziramo v drevo (takemu drevesu pravimo *drevo intervalov* ali *drevo segmentov*). Listi drevesa predstavljajo posamezne intervale; njihovi starši predstavljajo pare po dveh zaporednih intervalov; stari starši predstavljajo skupine štirih zaporednih intervalov in tako naprej. Nivoje drevesa bomo označili z indeksom v oklepajih; nivo 0 so listi, nivo 1 so njihovi starši in tako naprej; najvišji nivo je $\lceil \log_2 t \rceil$, na katerem je eno samo vozlišče, ki

predstavlja vse intervale skupaj. Označimo z $I_j^{(r)}$ unijo intervalov $I_{j'}$ za $j \cdot 2^r \leq j' < (j+1) \cdot 2^r$; ta unija se razteza od $x_{j \cdot 2^r}$ do vključno $x_{(j+1) \cdot 2^r} - 1$, širina tega intervala je torej $w_j^{(r)} := x_{(j+1) \cdot 2^r} - x_{j \cdot 2^r}$. Na nivoju r zdaj j-to vozlišče predstavlja interval $I_j^{(r)}$; njegova otroka sta intervala $I_{2j}^{(r-1)}$ in $I_{2j+1}^{(r-1)}$, njegov starš pa je interval $I_{\lfloor j/2 \rfloor}^{(r+1)}$.

Za vsako vozlišče $I_j^{(r)}$ bomo hranili naslednje vrednosti: $h_j^{(r)}$, ki naj predstavlja število tistih vrstic (izmed doslej obdelanih vrstic, torej od $v+1$ do n), ki v celoti pokrivajo interval $I_j^{(r)}$, ne pa tudi njegovega starša; $\beta_j^{(r)}$ naj bo skupno število enot na intervalu $I_j^{(r)}$ v vseh tistih vrsticah, ki ne pokrivajo v celoti starša intervala $I_j^{(r)}$; vsota x-koordinat vseh teh enot pa naj bo $\sigma_j^{(r)}$. Opazimo lahko, da je $\beta_j^{(r)} = \beta_{2j}^{(r-1)} + \beta_{2j+1}^{(r-1)} + h_j^{(r)} \cdot w_j^{(r)}$ in $\sigma_j^{(r)} = \sigma_{2j}^{(r-1)} + \sigma^{(r-1)} 2j + 1 + h_j^{(r)} \cdot w_j^{(r)} (x_{j \cdot 2^r} + (w_j^{(r)} - 1)/2)$.

S pomočjo takega drevesa lahko koraka 3 in 4 našega postopka izvedemo bolj učinkovito. Oglejmo si najprej korak 3, v katerem moramo v našo podatkovno strukturo dodati vrstico $v+1$, ki se razteza od z_{v+1} do $z_{v+1} + d_{v+1}$. Ko na začetku postopka pripravimo urejeni seznam $x_0, x_1, \ldots, x_t$, si moramo ob vsakem od teh x-ov tudi zapisati, katere vrstice se začnejo ali končajo pri tem x; s pomočjo teh podatkov si lahko pri vsaki vrstici zabeležimo, kje v seznamu $x_0, \ldots, x_t$ se nahajata njen začetek in njen konec. Korak 3 lahko s tem v $O(1)$ časa ugotovi, da se trenutna vrstica na primer razteza od x_l do x_d (torej da je $x_l = z_{v+1}$ in $x_d = z_{v+1} + d_{v+1}$. Naša vrstica je torej unija intervalov od I_l do I_{d-1}.

Zdaj za vsak nivo drevesa, recimo r, poglejmo, katere intervale s tega nivoja pokriva naša vrstica v celoti. Interval $I_j^{(r)}$ je unija intervalov od $I_{j \cdot 2^r}$ do $I_{(j+1) \cdot 2^r - 1}$, torej ga naša vrstica pokriva v celoti, če je $l \leq j \cdot 2^r$ in $(j+1) \cdot 2^r \leq d$. Iz prve neenačbe dobimo $l' \leq j$ (za $l' = \lceil l/2^r \rceil$), iz druge pa $j \leq d'$ (za $d' = \lfloor d/2^r \rfloor - 1$). Če je $l' > d'$, naša vrstica na trenutnem nivoju sploh ne pokriva nobenega intervala v celoti, zato se s tem nivojem ni več treba ukvarjati. Če je $l' = d'$, pokriva naša vrstica na trenutnem nivoju v celoti le en interval, torej povečamo njegovo $h_{l'}^{(r)}$ za 1 in smo s tem nivojem končali. Sicer pa razmišljamo takole: če je l' lih, moramo povečati $h_{l'}^{(r)}$ za 1, in če je d' sod, moramo povečati $h_{d'}^{(r)}$ za 1; za vse ostale intervale pa gotovo velja, da je poleg njih v celoti pokrit tudi njihov brat, torej je v celoti pokrit tudi njun starš in bomo to pokritost upoštevali na višjih nivojih drevesa, ne pa na trenutnem. Tako moramo torej pri vsakem r povečati največ dve števili $h_j^{(r)}$.

Kaj pa $\beta_j^{(r)}$ in $\sigma_j^{(r)}$? Te je treba ponovno izračunati tam, kjer nova vrstica leži vsaj delno na intervalu $I_j^{(r)}$, ne pokrije pa v celoti njegovega starša. Na vsakem nivoju velja, da naša vrstica največ dva intervala pokrije delno, tiste vmes v celoti, ostalih pa sploh nič. Če našega $I_j^{(r)}$ pokrije vsaj delno, mora tudi njegovega starša pokriti vsaj delno, torej starš ne more biti eden od tistih, ki niso čisto nič pokriti; in ker po drugi strani hočemo le take intervale, katerih starš ni v celoti pokrit, mora biti torej starš eden od tistih (največ dveh) intervalov na nivoju $r+1$, ki sta pokrita delno, ne pa v celoti. Vsak od teh največ dveh staršev ima največ dva otroka, torej bo treba popraviti $\beta_j^{(r)}$ in $\sigma_j^{(r)}$ pri največ 4 intervalih na vsakem nivoju.

Tako torej vidimo, da ima naš korak 3 na vsakem nivoju le $O(1)$ dela s spreminjanjem vrednosti $h_j^{(r)}$, $\beta_j^{(r)}$ in $\sigma_j^{(r)}$. Časovna zahtevnost koraka 3 je tako sorazmerna z višino drevesa, to pa je $O(\log n)$.

Pri koraku 4 pa si lahko z našim drevesom pomagamo, da hitro poiščemo naj-

manjši j: začnemo v korenu drevesa in nato na vsakem koraku preverimo, ali leži iskani j v levem ali desnem otroku trenutnega vozlišča; vanj se nato premaknemo in po tem postopku nadaljujemo, dokler ne pridemo do lista. Zapišimo ta postopek s psevdokodo:

$r := \lceil \log_2 t \rceil$; $j := 0$; (* *začnimo v korenu drevesa* *)
$\beta_L := 0$; $\sigma_L := 0$; $h := 0$;
while $r > 0$:
  (* *Trenutno smo v vozlišču* $I_j^{(r)}$. *V vrsticah* $v+1, \ldots, n$ *leži levo od njega*
   (*torej v intervalih* $I_0, \ldots, I_{j \cdot 2^r - 1}$) β_L *enot, vsota njihovih*
   koordinat pa je σ_L. *Če porušimo vse te enote, zid ostane stabilen;*
   če pa poleg njih uničimo še vse enote iz $I_j^{(r)}$, *se zid poruši.* *)
  $h := h + h_j^{(r)}$; (* h *je število tistih vrstic* (*izmed* $v+1, \ldots, n$),
         ki v celoti pokrivajo interval $I_j^{(r)}$. *)
  $\beta_M := \beta_L + \beta_{2j}^{(r-1)} + h \cdot w_{2j}^{(r-1)}$;
  $\sigma_M := \sigma_L + \sigma_{2j}^{(r-1)} + h \cdot w_{2j}^{(r-1)}(x_{j \cdot 2^r} + (w_{2j}^{(r-1)} - 1)/2)$;
  (* *Interval* $I_{2j+1}^{(r-1)}$ *je desni otrok trenutnega intervala* $I_j^{(r)}$. *Levo od njega leži*
   (*v vrsticah* $v+1, \ldots, n$) β_M *enot, vsota njihovih* x*-koordinat pa je* σ_M. *)
  $t := (s_{v+1} - \sigma_M)/(b_{v+1} - \beta_M)$;
  if $t > z_v + d_v - 1/2$
  then $r := r - 1$; $j := 2j$;
  else $r := r - 1$; $j := 2j + 1$; $\beta_L := \beta_M$; $\sigma_L := \sigma_M$;

Med spuščanjem po drevesu torej vedno stojimo v takem intervalu $I_j^{(r)}$, za katerega velja, da če uničimo vse enote levo od njega, zid ostane stabilen, če pa poleg njih uničimo še vse enote v trenutnem intervalu, se zid prevrne. Nato preverimo, kaj se zgodi, če uničimo vse enote v levem podintervalu trenutnega intervala; če je zid takrat še stabilen, se premaknemo v desnega otroka, sicer pa v levega. Pri izračunu β_M in σ_M smo si pomagali z naslednjim razmislekom: enote, ki ležijo levo od našega desnega otroka $I_{2j+1}^{(r-1)}$, lahko razdelimo na tri skupine; (1) tiste, ki ležijo levo od našega intervala $I_j^{(r)}$ sploh: takih je β_L; (2) tiste, ki ležijo na intervalu našega levega otroka, $I_{2j}^{(r-1)}$; te pa razdelimo na (2.1) tiste, ki ležijo v vrsticah, ki našega intervala $I_j^{(r)}$ ne pokrivajo v celoti: takih enot je $\beta_{2j}^{(r-1)}$; in (2.2) tiste, ki ležijo v vrsticah, ki naš interval $I_j^{(r)}$ pokrivajo v celoti; takih vrstic je h; in ker one potemtakem tudi našega levega otroka $I_{2j}^{(r-1)}$ pokrivajo v celoti, je teh enot skupno $w_{2j}^{(r-1)} \cdot h$. Izračun σ_M gre po podobnem razmisleku.

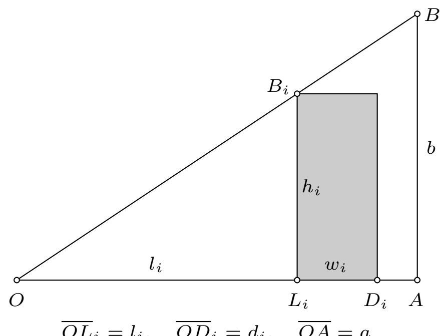
Vsaka iteracija te zanke ima torej le $O(1)$ dela, zato je časovna zahtevnost koraka 4 zdaj sorazmerna z višino drevesa, to je $O(\log n)$. Ker moramo koraka 3 in 4 izvesti po enkrat pri vsakem v (teh pa je $O(n)$), nam skupaj porabita $O(n \log n)$ časa. Toliko časa gre tudi za urejanje seznama $x_0, \ldots, x_t$ na začetku postopka. Časovna zahtevnost naše rešitve je tako zdaj le $O(n \log n)$ namesto prejšnje $O(n^2)$.

V praksi lahko obe različici rešitve (tisto z drevesom in tisto brez njega) izboljšamo še z naslednjim opažanjem: z intervali, ki ležijo desno od $\max\{z_i + d_i : v < i \leq n\}$, se ni treba ukvarjati, kajti če pri uničevanju enot pridemo tako daleč proti desni, bo vsaj ena od vrstic $v+1, \ldots, n$ v celoti pobrisana, torej tako dobljena rešitev

ne bo nič boljša od tega, kar smo že dobili pri (a), tako da se nam z njo ni treba ukvarjati.

3. Polaganje plošč

Naloga pravi, da mora zgornji levi kot plošče ležati na hipotenuzi našega trikotnika; s tem je položaj posamezne plošče čisto enolično določen — glede tega, kam jo položiti, nimamo nobene izbire, izbiramo lahko le to, ali jo sploh položimo ali ne. Kot vidimo iz spodnje slike, moramo ploščo i položiti tako, da njen levi rob leži na x-koordinati $l_i := a \cdot h_i/b$; njen desni rob pa torej leži na x-koordinati $d_i := x_i + w_i$. (Če se pri kakšni plošči izkaže, da je $d_i > a$, jo lahko takoj zavržemo, saj bi taka plošča nujno štrlela iz našega trikotnika, česar pa naloga ne dovoli.)



Veliki trikotnik $\triangle OAB$, v katerega smo položili sivo ploščo, je podoben manjšemu trikotniku $\triangle OL_iB_i$, ki ga tvori levo oglišče O skupaj z levim robom naše plošče. Ker sta si podobna, so razmerja v dolžinah stranic enaka:

$$\frac{l_i}{a} = \frac{h_i}{b},$$

iz česar dobimo $l_i = a \cdot h_i/b$.

Recimo, da polagamo plošče od leve proti desni. Če se odločimo najprej uporabiti ploščo i, to pomeni, da ne bomo mogli uporabiti nobene take plošče j, ki bi imela $x_j < d_i$, saj bi se taka plošča prekrivala s ploščo i (ali pa celo ležala levo od nje, kar bi kršilo našo odločitev, da bomo plošče polagali od leve proti desni). Torej je smiselno za začetek uporabiti tisto ploščo, ki ima najmanjšo vrednost d_i, saj nas bo ta najmanj omejevala pri izbiri nadaljnjih plošč. Če bi namesto s to ploščo začeli z neko drugo, recimo k, ki ima $d_k > d_i$, in potem nadaljevali z neko ploščo j (ki ima torej $x_j > d_k$), bi lahko namesto plošče k uporabili ploščo i, pa bi razpored še vedno ostal veljaven (če je $x_j > d_k$ in $d_k > d_i$, je tudi $x_j > d_i$, tako da sme plošča i stati pred ploščo j) in enako dober (število plošč se nič ne spremeni). Torej res ni nič narobe, če začnemo naš razpored s ploščo, ki ima najmanjšo vrednost d_i — tako gotovo ne bomo spregledali najboljšega možnega razporeda.

Ko smo si na ta način izbrali prvo ploščo (recimo i), lahko v mislih zavržemo vse tiste plošče j, ki imajo $x_j < d_i$, nato pa nadaljujemo s podobnim razmislekom kot prej: med preostalimi ploščami izberemo tisto z najmanjšim d_j; nato zavržemo vse plošče, ki imajo $x_k < d_j$; tako nadaljujemo, dokler nam ne zmanjka plošč.

Zapišimo dobljeni postopek še s psevdokodo:

za vsako ploščo i:
  $l_i := a \cdot h_i/b$; $d_i := l_i + w_i$;
$d := 0$; (* *d predstavlja desni rob zadnje doslej položene plošče* *)
uredi plošče naraščajoče po d_i;
za vsako ploščo i v tem vrstnem redu:

```
    if l_i < d then
      continue; (* plošča i se prekriva z doslej položenimi *)
    položi ploščo i;
    d := d_i;
```

Časovna zahtevnost tega postopka je $O(n \log n)$, zaradi urejanja; vsi drugi koraki nam vzamejo le $O(n)$ časa.

4. Kodiranje

Recimo, da imamo naših deset peteric predstavljenih kar s tabelo 10×5 znakov '0' in '1' (v spodnji rešitvi je to globalna spremenljivka kode). Z dvema gnezdenima zankama se lahko sprehodimo po vseh možnih parih peteric in za vsak par preverimo, če bi se eno od njiju dalo predelati v drugo z eno zamenjavo dveh sosednjih bitov. Če najdemo kakšen tak par, lahko zaključimo, da naš nabor peteric iskane lastnosti (c) nima, sicer pa jo ima.

Razmisliti moramo še o tem, kako za dani dve peterici preveriti, ali je mogoče eno predelati v drugo z eno samo zamenjavo dveh sosednjih bitov. V zanki lahko primerjamo istoležne bite obeh peteric; ko opazimo neujemanje, preverimo, ali bi se ga dalo odpraviti z zamenjavo trenutnega in naslednjega bita. Če to ne gre, lahko takoj zaključimo, da sta si peterici preveč različni. Drugače pa si zapomnimo, da smo zamenjavo izvedli, in nadaljujmo s primerjanjem istoležnih bitov; če odtlej opazimo še kakršno koli drugo neujemanje, lahko tudi zaključimo, da sta si peterici preveč različni. Če pa pridemo do konca, ne da bi opazili še kakšno neujemanje, potem vemo, da bi se dalo eno peterico predelati v drugo z zamenjavo dveh sosednjih bitov (in torej nabor kot celota nima iskane lastnosti).

```
#include <stdbool.h>

enum { n = 10, d = 5 };
char kode[n][d];

bool ImaLastnostC()
{
  int x, y, i; bool dovoljRazlicni, zamenjava;

  /* Preglejmo vse pare različnih peteric in za vsak par preverimo,
     če sta si peterici dovolj različni. */
  for (x = 0; x < n − 1; x++) for (y = x + 1; y < n; y++)
  {
    dovoljRazlicni = false; zamenjava = false;

    /* Primerjajmo istoležne bite in iščimo neujemanja. */
    for (i = 0; i < d; i++)
    {
      if (kode[x][i] == kode[y][i]) continue;

      /* Če smo zamenjavo že izvedli in opazimo še kakšno neujemanje,
         potem sta si peterici dovolj različni. */
      if (zamenjava) { dovoljRazlicni = true; break; }

      /* Če zamenjave še nismo izvedli, preverimo, ali bi lahko trenutno
         neujemanje odpravili z zamenjavo bitov i in i + 1. */
      if (i + 1 < d && kode[x][i] == kode[y][i + 1] &&
          kode[x][i + 1] == kode[y][i]) { i++; zamenjava = true; }
```

```
      /* Če se neujemanja ne da odpraviti z zamenjavo, sta si peterici dovolj različni. */
      else { dovoljRazlicni = true; break; }
    }
    if (! dovoljRazlicni) return false;
  }
  return true;
}
```

5. Golovec

Za vsako vozilo v predoru v neki podatkovni strukturi zapomnimo njegovo registrsko številko in čas prihoda v predor. Ko sistem pokliče naš podprogram, moramo preveriti, ali je vozilo z dano številko trenutno v predoru ali ne; če ga še ni v predoru, si ga le zapomnimo v naši podatkovni strukturi; če pa je že v predoru, lahko zdaj izračunamo njegovo povprečno hitrost (čas prihoda v predor imamo v naši podatkovni strukturi, čas izhoda iz predora pa smo pravkar dobili kot parameter funkcije Golovec) in če je previsoka, izpišemo njegovo registrsko številko. V vsakem primeru pa moramo nato podatke o tem vozilu pobrisati iz naše podatkovne strukture, saj ga zdaj ni več v predoru. (Če bo kasneje prišel spet kak klic za to vozilo, bo to pomenilo, da je isto vozilo ponovno zapeljalo v predor.)

Vprašanje je, kakšno podatkovno strukturo bi uporabili. Ker je vozil malo (naloga pravi, da jih je naenkrat v predoru največ 300), smo v naši spodnji rešitvi uporabili kar navadno tabelo (pravzaprav dve tabeli, eno za registrske številke in eno za čase prihoda). Tabela ima 300 elementov, dejansko število vozil v predoru (in s tem v tabeli) pa hranimo v globalni spremenljivki stVozil. Ta vozila so v naši tabeli shranjena na indeksih od 0 do stVozil − 1, vendar brez kakšnega posebnega vrstnega reda, tako da moramo pri preverjanju, ali je neko vozilo že v tabeli, iti kar v zanki po vseh elementih tabele in za vsakega primerjati njegovo registrsko številko s številko trenutnega vozila. Nova vozila dodajamo na konec tabele (na indeks stVozil), pri brisanju pa podatke za zadnje vozilo v tabeli (tisto na indeksu stVozil − 1) preprosto skopiramo v celico, iz katere smo vozilo pobrisali, in zmanjsamo stVozil za 1.

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

enum { dolzinaPredora = 622, maxHitrost = 22, maxVozil = 300, dolzinaStevilke = 7 };
int stVozil = 0, casPrihoda[maxVozil];
char regStevilke[maxVozil][dolzinaStevilke + 1];

void Golovec(char *stevilka, int cas)
{
  int i, casVoznje;

  /* Poglejmo, če že imamo podatek o vozilu s to številko. */
  i = 0; while (i < stVozil && 0 != strcmp(regStevilke[i], stevilka)) i++;

  /* Če vozila s to številko še nimamo, je očitno pravkar zapeljalo
     v predor, zato si ga le zapomnimo. */
  if (i >= stVozil) {
    strcpy(regStevilke[stVozil], stevilka);
    casPrihoda[stVozil++] = cas; return; }

  /* Sicer pa je pravkar zapeljalo iz predora; preverimo,
```

```
      če je njegova povprečna hitrost previsoka. */
  casVoznje = cas − casPrihoda[i];
  if (maxHitrost * casVoznje < dolzinaPredora)
    /* S povprečno hitrostjo maxHitrost se v tem času ne bi dalo prevoziti
       celega predora, torej je ta avtomobil vozil prehitro. */
    printf("%s\n", stevilka);
  /* Zdaj lahko to vozilo pobrišemo iz naše tabele. */
  strcpy(regStevilke[i], regStevilke[stVozil − 1]);
  casPrihoda[i] = casPrihoda[stVozil − 1];
  −−stVozil;
}
```

Pri takšni tabeli je dodajanje in brisanje vozila poceni, saj vzame le $O(1)$ časa, drago pa je iskanje, ki traja kar $O(n)$ časa, če je n število vozil v predoru. Možnih je še več drugih podatkovnih struktur: namesto tabele lahko uporabimo verigo, povezano s kazalci (*linked list*); vozila lahko hranimo urejena po registrski številki, kar bi nam omogočilo iskanje (z bisekcijo) v času $O(\log n)$, vendar bi za dodajanje in brisanje porabili po $O(n)$ časa, tako da s tem ne bi nič pridobili (saj vsako vozilo enkrat dodamo, enkrat pobrišemo in dvakrat iščemo). Boljša možnost je, da bi vozila hranili v drevesu (na primer AVL-drevesu ali rdeče-črnem drevesu), kjer bi tako iskanje kot dodajanje in brisanje trajala le po $O(\log n)$ časa. Še lepše pa bi bilo hraniti vozila v razpršeni tabeli (*hash table*), kjer vzamejo te operacije le po $O(1)$ časa.

REŠITVE NALOG ZA TRETJO SKUPINO

1. Nurikabe

Podatke o mreži si preberimo kar v dvodimenzionalno tabelo (v spodnjem programu je to mreza), v kateri bo vsako polje predstavljal en znak (tipa **char**).

Števila črnih kvadratov velikosti 2×2 ni težko določiti; pojdimo v zanki po vseh poljih mreže, razen tistih v najbolj spodnji vrstici in najbolj desnem stolpcu, in za vsako preverimo, če tisto polje skupaj s svojim spodnjim, desnim in spodnjim desnim sosedom tvori tak črn kvadrat (če so vsa štiri polja črna, povečamo števec kvadratov za 1).

Malo več dela bo s štetjem otokov in morij. Postavimo se v poljubno polje mreže, recimo kar tisto v zgornjem levem kotu; iz pripadajočega znaka v tabeli mreza lahko ugotovimo, ali je tam morje ali otok; zdaj pa bi radi odkrili še vsa ostala polja, ki pripadajo temu morju ali otoku. To lahko naredimo z iskanjem v širino: začetno polje dodajmo v vrsto, nato pa na vsakem koraku vzemimo po eno polje iz vrste in dodajmo v vrsto vse tiste njegove sosede, ki pripadajo istemu morju ali otoku. Tako bomo prej ali slej obiskali vsa polja trenutnega morja ali otoka. Pri tem moramo paziti, da ne dodamo istega polja v vrsto po večkrat; zato bomo, ko polje prvič dodamo v vrsto, postavili njemu pripadajoč element v tabeli mreza na `'x'`, da bomo kasneje vedeli, da ga ne smemo več dodajati v vrsto.

Vrsto je mogoče implementirati na različne načine, najenostavnejši pa je kar s tabelo (v spodnjem programu je to spremenljivka vrsta) in dvema števcema, glava in rep. Elemente vrste tako hranimo na indeksih od glava do rep − 1; elemente pobiramo iz vrste pri glavi, nove elemente pa dodajamo pri repu.

Na koncu tega postopka smo torej pregledali celotno morje ali otok in označili vsa njegova polja z `'x'`. Če gre za otok (in ne morje), lahko spotoma tudi gledamo, če je na kakšnem njegovem polju številka; to številko si zapomnimo v spremenljivki oznaka. Če številke še nismo našli, naj ima oznaka vrednost 0; če pa smo našli več kot eno številko, postavimo oznaka na -1. Na koncu vemo, da je otok pravilno označen le, če je oznaka enaka številu polj na otoku; to število pa imamo v spremenljivki rep, saj se ta ob vsakem dodajanju polja v vrsto poveča za 1, torej na koncu pove ravno skupno število vseh polj na otoku.

Zdaj vemo, ali imamo morje ali otok in ali je otok pravilno označen, tako da lahko primerno povečamo ustrezne števce (nMorij, nOtokov in nOznacenih).

Ko smo tako obdelali prvi otok ali morje, lahko poiščemo naslednje še neobdelano polje (torej tako, ki v tabeli še nima znaka `'x'`) in na enak način obdelamo tudi njegov otok ali morje; tako nadaljujemo, dokler ni obdelana cela mreža. Na koncu moramo le še izpisati rezultate.

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

#define MaxW 1000
#define MaxH 1000

const int DX[] = { −1, 1, 0, 0 }, DY[] = { 0, 0, −1, 1 };
char mreza[MaxH][MaxW + 2];
int vrsta[MaxW * MaxH];

int main()
```

```
{
   int w, h, x, y, x1, y1, x2, y2, u, d, oznaka, glava, rep; bool otok; char c;
   int nMorij = 0, nOtokov = 0, nOznacenih = 0, nKvadratov = 0;

   /* Preberimo vhodne podatke. */
   FILE *f = fopen("nurikabe.in", "rt");
   fscanf(f, "%d %d\n", &w, &h);
   for (y = 0; y < h; y++) fgets(mreza[y], w + 2, f);
   fclose(f);

   /* Preštejmo črne kvadrate 2 * 2. */
   for (y = 0; y < h − 1; y++) for (x = 0; x < w − 1; x++)
      if (mreza[y][x] == '#' && mreza[y][x + 1] == '#' &&
      mreza[y + 1][x] == '#' && mreza[y + 1][x + 1] == '#') nKvadratov++;
   /* Poiščimo otoke in morja. */
   for (y = 0; y < h; y++) for (x = 0; x < w; x++)
   {
      /* Če je na trenutnem polju mreže znak 'x', to pomeni, da to polje
          pripada nekemu morju ali otoku, ki smo ga že obdelali, zato ga
          lahko zdaj preskočimo. */
      c = mreza[y][x]; if (c == 'x') continue;

      /* Sicer pa bomo njegov otok ali morje obdelali zdaj. Dodajmo ga v vrsto. */
      glava = 0; rep = 0; vrsta[rep++] = y * w + x; mreza[y][x] = 'x';

      /* Poglejmo, ali gre za otok in če da, ali ima že tudi oznako. */
      otok = (c != '#');
      oznaka = (c >= '1' && c <= '9') ? (c − '0') : 0;

      /* Preglejmo preostanek tega morja ali otoka. */
      while (glava < rep)
      {
         /* Vzemimo naslednje polje iz vrste in preglejmo njegove sosede. */
         u = vrsta[glava++]; x1 = u % w; y1 = u / w;
         for (d = 0; d < 4; d++) {
            x2 = x1 + DX[d]; y2 = y1 + DY[d];
            if (x2 < 0 || y2 < 0 || x2 >= w || y2 >= h) continue;

            /* Ali ta sosed sploh pripada istemu morju oz. otoku? */
            c = mreza[y2][x2];
            if (c == 'x' || (otok ? c == '#' : c != '#')) continue;

            /* Če je tu oznaka otoka, si jo zapomnimo. */
            if (c >= '1' && c <= '9') oznaka = (oznaka == 0) ? (c − '0') : −1;

            /* Dodajmo tega soseda v vrsto. */
            vrsta[rep++] = y2 * w + x2; mreza[y2][x2] = 'x'; }
      }
      /* Primerno povečajmo števce otokov in morij. */
      if (otok) { nOtokov++; if (oznaka == rep) nOznacenih++; }
      else nMorij++;
   }

   /* Izpišimo rezultate. */
   f = fopen("nurikabe.out", "wt");
   fprintf(f, "%d\n%d\n%d\n%d\n", nMorij, nOtokov, nOznacenih, nKvadratov);
   fclose(f); return 0;
}
```

2. Analiza signala

Označimo naš vhodni signal z $a_0, a_1, \ldots, a_{n-1}$. Ker so oddajniki sinhronizirani tako, da na začetku vsi oddajo enico, mi pa zaznamo vsoto vseh oddanih signalov, to pomeni, da je a_0 ravno število vseh oddajnikov.

Zdaj pa lahko razmišljamo takole: poiščimo najmanjši tak $t > 0$, pri katerem je $a_t > 0$. Ob času t oddajo signal le tisti oddajniki, katerih perioda je delitelj števila t. Če bi nek tak oddajnik imel periodo $p < t$, bi oddal enico že tudi ob času p, torej bi bil $a_p > 0$; mi pa smo t izbrali tako, da so med a_0 in a_t v zaporedju same ničle, torej takega oddajnika ni. Torej so enice, ki so se seštele v a_t, prispevali le oddajniki s periodo točno t. Zdaj torej vemo, da imamo točno a_t oddajnikov s periodo t; to si zapomnimo v neki tabeli (spodnji program ima v ta namen tabelo `np`). Ti oddajniki seveda ne oddajo enice le ob času t, ampak tudi ob času 0, $2t$, $3t$ in tako naprej. Ob vseh teh večkratnikih t-ja zmanjšajmo vrednost našega signala za število oddajnikov s periodo t. Tako nam ostane v naši tabeli a le vsota tistih signalov, ki so jih oddali oddajniki s periodo, večjo od t.

S tem postopkom lahko zdaj nadaljujemo in odkrivamo še nove oddajnike z vse večjimi periodami, dokler vse vrednosti $a_1, \ldots, a_{n-1}$ ne padejo na 0. Na tej točki se lahko zgodi, da je a_0 še vedno večji od 0. Če pride do tega, vemo, da obstajajo nekateri oddajniki s periodo, večjo ali enako n; od njih smo zaznali le enico ob času 0, kasnejših enic pa ne, ker jih nismo poslušali dovolj dolgo, da bi zaznali naslednjo enico. Zanje torej ne moremo ugotoviti, kakšne točno so njihove periode, nam pa sedanja vrednost a_0 pove vsaj to, koliko je takih oddajnikov. To pa je tudi ena od stvari, po katerih sprašuje naloga.

```
#include <stdio.h>
#define MaxN 100000
int a[MaxN], np[MaxN];

int main()
{
  int nOddajnikov, n, i, j;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("signal.in", "rt");
  fscanf(f, "%d", &n);
  for (i = 0; i < n; i++) fscanf(f, "%d", &a[i]);
  fclose(f);

  /* Ker vsi oddajniki na začetku oddajo enico, je prvi element
     prebranega signala ravno skupno število oddajnikov. */
  nOddajnikov = a[0];

  /* Preglejmo vse možne periode od 1 do n − 1. */
  for (i = 1; i < n; i++) {
    /* Na tej točki vsebuje tabela a samo še vsoto signalov vseh tistih oddajnikov,
       ki imajo periodo vsaj i. Poleg tega so a[1], ..., a[i − 1] že vsi enaki 0.
       Če je torej v a[i] neka neničelna vrednost, jo morajo povzročati oddajniki s periodo i
       (in ne kakšno krajšo). Zapomnimo si, koliko jih je. */
    np[i] = a[i];

    /* Odštejmo iz skupnega signala a tisto, kar prispevajo ti oddajniki.
       Torej moramo odšteti število teh oddajnikov pri vseh večkratnikih i-ja. */
```

```
      if (np[i] > 0) for (j = 0; j < n; j += i) a[j] −= np[i]; }

  /* Izpišimo rezultate. Za vsak i imamo np[i] oddajnikov s periodo i. Kar na koncu še
     ostane v a[0], so oddajniki s periodo, večjo ali enako n (ki v našem signalu
     prispevajo le enico v a[0] in nikjer drugje, ker je naša meritev prekratka). */
  f = fopen("signal.out", "wt");
  fprintf(f, "%d %d\n", nOddajnikov, a[0]);
  for (i = 1; i < n; i++)
    if (np[i] > 0) fprintf(f, "%d %d\n", i, np[i]);
  fclose(f); return 0;
}
```

3. Mafijski semenj

V nalogi se skriva problem topološkega urejanja grafa (v katerem je za vsakega mafijca po ena točka, usmerjena povezava pa obstaja tam, kjer en mafijec meri s pištolo na drugega, ki ne pripada isti združbi), vendar si lahko rešitev predstavljamo tudi kot preprosto simulacijo dogajanja, ki ga opisuje besedilo naloge.

Vhodne podatke preberimo v tri tabele, eno za z_i, eno za l_i in eno za d_i. Nato za vsakega mafijca izračunamo, koliko članov drugih združb meri vanj; to shranimo v tabeli stopnja (v grafu je *vhodna stopnja* točke definirana kot število povezav, ki kažejo v to točko).

Zdaj vemo, da lahko mafijci s stopnjo 0 takoj odidejo s prizorišča; zaradi njihovega odhoda se potem lahko zmanjša stopnja nekaterih drugih mafijcev; če kakšnemu od njih pade stopnja na 0, odide tudi on in tako naprej. Koristno je torej vzdrževati nekakšen seznam mafijcev, za katere že vemo, da bodo odšli, nismo pa še pregledali, na koga so merili in kako se zaradi njihovega odhoda spremenijo stopnje. V spodnjem programu imamo v ta namen tabelo toDo, v kateri hranimo mafijce, ki jih bo treba še obdelati (na indeksih od 0 do nToDo − 1).

Na začetku dodamo v seznam tiste, ki imajo že na začetku stopnjo 0. Nato na vsakem koraku vzamemo enega mafijca (recimo mu u) iz seznama; za vsakega v, v katerega je u uperil kakšno od svojih pištol, pogledamo, če pripada drugi združbi kot u, in če je tako, zmanjšamo v-jevo stopnjo za 1 (ker vemo, da bo u sčasoma odšel, zato bo takrat v v-ja uperjena ena pištola manj). Če kakšnemu v-ju zaradi tega pade stopnja na 0, dodamo v seznam toDo še njega.

Ta postopek se ustavi, ko se seznam toDo izprazni. Mafijci, ki imajo še zdaj stopnjo, večjo od 0, pa so tisti, ki bodo obtičali na sejmišču in jih moramo zdaj izpisati.

```
#include <stdio.h>
#define MaxN 1000000
int n, m, zi[MaxN], li[MaxN], di[MaxN];
int stopnja[MaxN], toDo[MaxN], nToDo;

int main()
{
  int u, v;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("semenj.in", "rt");
  fscanf(f, "%d %d", &n, &m);
```

```
  for (u = 0; u < n; u++) {
    fscanf(f, "%d %d %d", &zi[u], &li[u], &di[u]);
    li[u]--; di[u]--; stopnja[u] = 0; }
  fclose(f);

  /* Določimo vhodne stopnje vseh točk. */
  for (u = 0; u < n; u++) {
    if (zi[u] != zi[li[u]]) stopnja[li[u]]++;
    if (zi[u] != zi[di[u]]) stopnja[di[u]]++; }
  /* Točke z vhodno stopnjo 0 dodajmo v seznam toDo. */
  for (u = 0, nToDo = 0; u < n; u++)
    if (stopnja[u] == 0) toDo[nToDo++] = u;
  /* Topološko uredimo graf. */
  while (nToDo > 0) {
    u = toDo[--nToDo];
    v = li[u]; if (zi[u] != zi[v]) if (--stopnja[v] == 0) toDo[nToDo++] = v;
    v = di[u]; if (zi[u] != zi[v]) if (--stopnja[v] == 0) toDo[nToDo++] = v; }

  /* Izpišimo rezultate: točke, ki niso nikoli prišle v seznam toDo. */
  f = fopen("semenj.out", "wt");
  for (u = 0; u < n; u++) if (stopnja[u] > 0) fprintf(f, "%d\n", u + 1);
  fclose(f); return 0;
}
```

4. Trgovanje z zrni

Označimo z $a_1, \ldots, a_n$ število zrn, ki jih imajo naprodaj posamezni kmetje. Recimo, da bi nek trgovec rad kupil t zrn. Radi bi torej nekaj (nič ali več) števil izmed $a_1, \ldots, a_n$ sešteli tako, da bi bila vsota čim bližje t (vendar ne večja od t). Pri vsakem a_i imamo dve možnosti: lahko ga vzamemo v vsoto ali pa ne; tako je skupaj $2 \cdot 2 \cdot \ldots \cdot 2 = 2^n$ možnih vsot. Ker gre lahko n do 40, si ne moremo privoščiti, da bi izračunali vse te vsote in pogledali, katera med njimi je najbližja t, saj bi nam to vzelo preveč časa.

Koristna ideja je, da razdelimo kmete na dve skupini. V prvi bodo kmetje $a_1, \ldots, a_k$ (za nek primerno izbran k), v drugi pa preostali kmetje, torej $a_{k+1}, \ldots, a_n$. Za vsako skupino izračunajmo vse možne vsote in jih uredimo naraščajoče; tako imamo seznam 2^k vsot za prvo skupino in seznam 2^{n-k} vsot za drugo skupino. Če je vseh kmetov na primer $n = 40$, si lahko izberemo $k = 20$ in imamo torej dva seznama s po 2^{20} vsotami. To je malo več kot milijon v vsakem seznamu, kar je čisto obvladljivo; pri manjših n pa so seznami še krajši.

Označimo vsote na prvem seznamu z $v_1, \ldots, v_s$ in podobno tiste na drugem seznamu z $\hat{v}_1, \ldots, \hat{v}_{\hat{s}}$. Načeloma sta dolžini seznamov $s = 2^k$ in $\hat{s} = 2^{n-k}$, vendar sta lahko tudi krajša, če iz seznamov pobrišemo duplikate (če lahko enako vsoto dobimo na več načinov, si jo zapomnimo le enkrat). Zdaj bi torej radi z vsoto oblike $v_i + \hat{v}_j$ (za neka indeksa i in j) prišli čim bližje t (ne smemo pa ga preseči).

Tega se lahko lotimo na različne načine, pri čemer si pomagamo z dejstvom, da sta seznama urejena. Ena možnost je, da za vsak v_i s prvega seznama izvedemo bisekcijo po drugem seznamu, v katerem na ta način poiščemo največji tak $\hat{v}_j$, ki je še $\leq t - v_i$. Taka bisekcija nam pri vsakem v_i vzame $O(\log \hat{s}) = O(n - k)$ časa, tako da je časovna zahtevnost skupaj $O(2^k(n - k))$.

Druga možnost pa je neke vrste zlivanje seznamov. Načeloma nas pri vsakem elementu prvega seznama (recimo i) zanima, kateri je največji tak element drugega seznama (recimo mu $j(i)$), pri katerem vsota $v_i + \hat{v}_{j(i)}$ še ne preseže t. Za $j(i)$ torej velja, da če vzamemo poljuben kasnejši element, recimo $u > j(i)$, je vsota $v_i + \hat{v}_u$ gotovo prevelika (večja od t). Ker je tudi prvo zaporedje urejeno naraščajoče, je potem za $u > j(i)$ tudi vsota $v_{i+1} + \hat{v}_u$ večja od t (saj je v_{i+1} večji od v_i). Iz tega vidimo, da bo $j(i+1) \leq j(i)$ — vsi členi drugega zaporedja, ki dajo skupaj z v_i preveliko vsoto, dajo skupaj z v_{i+1} še bolj preveliko vsoto. Ko torej povečamo i za 1, se $j(i)$ lahko le zmanjša ali ostane enak, ne more pa se povečati. Po vsakem povečanju i-ja moramo v zanki zmanjševati j, dokler vsota $v_i + \hat{v}_j$ ne posane $\leq t$.

Časovna zahtevnost tega zlivanja je $O(s + \hat{s}) = O(2^k + 2^{n-k})$, saj se po prvem seznamu premikamo ves čas le navzgor (i se povečuje), po drugem pa ves čas le navzdol (j se zmanjšuje). Če sta seznama približno enako dolga (na primer pri $k \approx n/2$), je to hitreje od bisekcije (ima pa tudi to prednost, da pri zlivanju prevladujejo zaporedni dostopi do pomnilnika, pri bisekciji pa naključni dostopi, kar slabše izkoristi procesorjev predpomnilnik). Bisekcija bi bila lahko hitrejša, če vzamemo dovolj majhen k, vendar v tem primeru potrebujemo toliko več pomnilnika za drugi seznam. Pri omejitvah, ki veljajo na našem tekmovanju v tretji skupini, je rešitev z zlivanjem boljša.

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

#define MaxN 40
#define MaxM 400
#define MaxNakup 20000000000000LL

typedef long long znesek_t;

/* Ta funkcija vpiše v tabelo „vsote“ vse možne vsote členov iz tabele „cleni“
   (teh členov je nClenov) in vrne število teh vsot. */
int PripraviVsote(znesek_t *cleni, int nClenov, znesek_t *vsote)
{
  int nVsot = 0, i, j, vsota;
  vsote[nVsot++] = 0;
  for (i = 0; i < nClenov; i++)
    /* Na tem mestu imamo v vsote[0..nVsot − 1] že vse možne vsote prvih i členov.
       Dodajmo v tabelo še eno kopijo teh vsot, povečanih za cleni[i]; tako bomo
       dobili vse možne vsote prvih i + 1 členov. */
    for (j = nVsot − 1; j >= 0; j−−) {
      vsota = vsote[j] + cleni[i];
      if (vsota > MaxNakup) continue; /* prevelike vsote sproti zavržemo */
      vsote[nVsot++] = vsota; }
  return nVsot;
}

/* Primerjalna funkcija za urejanje s qsort() iz standardne knjižnice. */
int Primerjaj(const void *a, const void *b) {
  znesek_t A = *(const znesek_t *) a, B = *(const znesek_t *) b;
  return A > B ? 1 : A < B ? −1 : 0; }

int main(int argc, char** argv)
{
  int i, j1, j2, m, n, k, nVsot1, nVsot2;
  znesek_t kmetje[MaxN], *vsote1, *vsote2, nakup, naj;
```

```
   /* Preberimo seznam kmetov. */
   FILE *f = fopen("zrna.in", "rt"), g = fopen("zrna.out", "wt");
   fscanf(f, "%d %d", &n, &m);
   for (i = 0; i < n; i++) fscanf(f, "%lld", &kmetje[i]);

   /* Razdelimo jih na dve skupini (prvih k in ostalih n − k)
      in za vsako pripravimo urejen seznam vseh možnih vsot. */
   k = n / 2;
   vsote1 = (znesek_t *) malloc(sizeof(znesek_t) << k);
   vsote2 = (znesek_t *) malloc(sizeof(znesek_t) << (n − k));
   nVsot1 = PripraviVsote(kmetje, k, vsote1);
   nVsot2 = PripraviVsote(kmetje + k, n − k, vsote2);
   qsort(vsote1, nVsot1, sizeof(vsote1[0]), &Primerjaj);
   qsort(vsote2, nVsot2, sizeof(vsote2[0]), &Primerjaj);

   /* Obdelajmo vse trgovce. */
   for (i = 0; i < m; i++) {
      fscanf(f, "%lld", &nakup);
      naj = 0; /* Najboljši doslej najdeni rezultat. */
      j2 = nVsot2 − 1; /* Položaj v zaporedju „vsote2“. */
      /* Pojdimo po vseh možnih vsotah prvih k kmetov. */
      for (j1 = 0; j1 < nVsot1 && j2 >= 0; j1++)
      {
         /* Na tem mestu vemo, da so vse vsote oblike
            vsote1[j1] + vsote2[j] za j > j2 že prevelike (večje od iskanega nakupa).
            Mogoče je celo vsota za j = j2 prevelika — če je treba, zmanjšajmo j2. */
         while (j2 >= 0 && vsote1[j1] + vsote2[j2] > nakup) j2−−;
         /* Zdaj je v j2 največji tak indeks, pri katerem je vsota vsot1[j1] + vsote2[j2]
            še ≤ nakup. Če je to največja doslej dosežena vsota, si jo zapomnimo v „naj“. */
         if (j2 >= 0 && vsote1[j1] + vsote2[j2] > naj) naj = vsote1[j1] + vsote2[j2];
      }
      /* Izpišimo rezultat. */
      fprintf(g, "%lld\n", naj);
   }

   /* Pospravimo za sabo. */
   fclose(f); fclose(g); free(vsote1); free(vsote2); return 0;
}
```

Omenimo še dve drobni izboljšavi te rešitve. Ena je, da kmete na začetku uredimo padajoče, tako da v prvo skupino pridejo tisti z največjimi a_i. Tako bodo vsote na prvem seznamu čim večje in lahko upamo, da se bo pri mnogih t že kmalu med zlivanjem zgodilo, da bo v_i že sam po sebi presegel t in se bo lahko zlivanje takoj končalo (saj če je že v_i večji od t, bo tudi vsaka vsota oblike $v_i + \hat{v}_j$ večja od t).

Druga izboljšava pa se nanaša na pripravo urejenega seznama vsot. V gornji rešitvi smo najprej pripravili neurejen seznam (funkcija PripraviVsote) in ga nato uredili (s funkcijo qsort iz standardne knjižnice). Če gledamo na primer skupino k kmetov, bomo najprej porabili $O(2^k)$ časa za pripravo seznam vsot in nato $O(k \cdot 2^k)$ za urejanje. Z nekaj pazljivosti lahko do urejenega sezanam pridemo že v času $O(2^k)$. Recimo, da že imamo urejen seznam vseh možnih 2^{k-1} vsot za prvih $k-1$ kmetov. V mislih si pripravimo še eno kopijo tega seznama, v kateri vsako vsoto povečamo za a_k.[2] Zdaj imamo dva urejena seznama, ki oba skupaj vsebujeta ravno vse možne

[2] „V mislih“ pravimo zato, ker si ni treba zares delati kopije seznama, saj lahko sproti računamo

vsote k kmetov; vse, kar moramo še narediti, je, da ju zlijemo v en sam seznam. Zlivanje poteka tako, da začnemo na začetku obeh seznamov in na vsakem koraku pogledamo, kateri od njiju ima na trenutnem mestu manjši element; ta element premaknemo v izhodni seznam in se premaknemo za eno mesto naprej po tistem vhodnem seznamu, iz katerega smo ta element dobili. Časovna zahtevnost takega zlivanja je le $O(2^k)$; da pa smo sploh prišli do seznama vseh vsot za prvih $k-1$ kmetov, smo morali pred tem zlivati dva seznama za prvih $k-2$ kmetov in tako naprej. Skupna cena vseh teh zlivanj je $O(2^k + 2^{k-1} + 2^{k-2} + \ldots + 1) = O(2^{k+1})$, kar je še vedno precej hitreje od $O(k \cdot 2^k)$.

Namesto funkcije PripraviVsote in klica qsort za njo bi morali torej poklicati takšno funkcijo:

```
int PripraviUrejeneVsote(znesek_t *cleni, int nClenov, znesek_t *vsote)
{
  int nVsot = 0, nVsot2, i, i1, i2; znesek_t vsota, clen;
  /* Pripravimo si pomožno tabelo, ki bo dovolj velika za polovico vseh možnih vsot. */
  znesek_t *vsote2 = (znesek_t *) malloc(
                         sizeof(znesek_t) << (nClenov > 0 ? nClenov - 1 : 0));
  /* Začnemo z eno samo vsoto (prvih 0 členov). */
  vsote[nVsot++] = 0;
  /* Pripravimo večje vsote. */
  for (i = 0; i < nClenov; i++) {
    /* Na tem mestu imamo v vsote[0..nVsot − 1] urejen seznam vseh možnih vsot
       prvih i členov. Skopirajmo jih v vsote2 in nVsot2. */
    for (i1 = 0, nVsot2 = nVsot; i1 < nVsot2; i1++) vsote2[i1] = vsote[i1];
    /* V mislih si predstavljajmo še eno kopijo tega seznama vsot, pri čemer vsaki
       prištejemo še naslednji člen, cleni[i]. Oba seznama bomo zdaj zlili (v tabelo
       vsote). Števca i1 in i2 povesta naš trentni položaj v obeh seznamih. */
    i1 = 0; i2 = 0; nVsot = 0; clen = cleni[i];
    while (i1 < nVsot2 || i2 < nVsot2) {
      /* Na indeksu i1 v prvem seznamu je vrednost vsote2[i1], na indeksu i2 v
         drugem seznamu pa je vrednost vsoste2[i2] + clen. Manjšo od teh vrednosti
         bomo prenesli v izhodni seznam (vsote) in se premaknili naprej po
         tistem od obeh vhodnih seznamov, iz katerega smo jo dobili. */
      if (i1 == nVsot2 || (i2 < nVsot2 && vsote2[i2] + clen < vsote2[i1]))
        vsota = vsote2[i2++] + clen;
      else vsota = vsote2[i1++];
      /* Spotoma še zavrzimo duplikate in morebitne prevelike vsote. */
      if (vsota <= MaxNakup && (nVsot == 0 || vsota > vsote[nVsot − 1]))
        vsote[nVsot++] = vsota; } }
  free(vsote2); /* Pobrišimo pomožno tabelo vsote2. */
  return nVsot;
}
```

5. Razcep niza

Označimo naš vhodni niz z $a = a_1 a_2 \ldots a_n$. Nalogo lahko rešujemo z rekurzivnim razmislekom: če hočemo razbiti a na k nepraznih podnizov, se moramo nekako odločiti, kako dolg naj bo zadnji od njih — recimo, da se zadnji podniz začne pri

njegove elemente iz elementov prvotnega seznama.

indeksu $i+1$ (zadnji podniz je torej $a_{i+1}a_{i+2}\dots a_n$). Potem nam preostane le še to, da ostanek niza, torej $a_1a_2\dots a_{i-1}a_i$, čim bolje razbijemo na $k-1$ nepraznih podnizov. Tu imamo torej enak problem kot na začetku, le niz je malo krajši (manjka mu zadnjih nekaj znakov) in zahtevan je en podniz manj kot prej.

V splošnem imamo torej podprobleme takšne oblike: naj bo $g(n', k')$ najmanjša možna vsota ocen podnizov pri razbitju niza $a_1a_2\dots a_{n'}$ na k' nepraznih podnizov. Naloga na koncu sprašuje po $g(n, k)$. Kot je pokazal razmislek v prejšnjem odstavku, lahko rešujemo te podprobleme s formulo $g(n', k') = \min\{g(i, k'-1) + f(a_{i+1}a_{i+2}\dots a_{n'}) : k'-1 \le i < n'\}$. (Omejitev $k'-1 \le i < n'$ izhaja iz dejstva, da če hočemo niz $a_1\dots a_i$ razbiti na $k'-1$ nepraznih podnizov, mora biti ta niz dolg vsaj $k'-1$ znakov; in podobno, da bo tudi zadnji podniz $a_{i+1}\dots a_{n'}$ neprazen, mora biti $i < n'$.)

Da ne bomo računali istih vrednosti $g(n', k')$ po večkrat, si jih je koristno zapomniti v neki tabeli. Opazimo lahko, da ko računamo $g(n', k')$, potrebujemo le rešitve oblike $g(i, k'-1)$, torej za podprobleme s $k'-1$ podnizi, ne pa tudi za podprobleme s $k'-2$ ali manj podnizi — tiste lahko sproti pozabljamo. Zato ima spodnji program tabelo g z le dvema vrsticama, eno za k' in eno za $k'-1$ (rešitev podproblema $g(n', k')$ hranimo v g[k′ % 2][n′]). Podprobleme je pametno reševati sistematično od manjših k' proti večjim — tako bomo imeli vedno pri roki rešitve manjših podproblemov, ko jih bomo potrebovali.

Razmislimo še o tem, kako bi učinkovito računali funkcijo f za ocenjevanje posameznega podniza. Načeloma moramo prešteti, koliko je v podnizu ničel in koliko enic, ocena f pa je potem manjše od teh dveh števil. Koristno si je vnaprej pripraviti dve tabeli, s_0 in s_1: pri tem naj $s_c[i]$ pove, kolikokrat se pojavlja števka c v nizu $a_{i+1}a_{i+2}\dots a_n$. To je najlažje računati po padajočih i; na začetku imamo $s_c[n] = 0$, nato pa $s_c[i] = s_c[i+1] + 1$, če je $a_{i+1} = c$, oz. $s_c[i] = s_c[i+1]$, če $a_{i+1} \ne c$.

Zdaj za poljuben podniz oblike $a_{i+1}a_{i+2}\dots a_j$ vemo, da vsebuje $s_c[i] - s_c[j]$ pojavitev števke c. Če izračunamo to za $c = 0$ in $c = 1$ in vzamemo manjšo od obeh vrednosti, dobimo ravno oceno $f(a_{i+1}\dots a_j)$.

```
#include <stdio.h>
#define MaxN 1000

int main()
{
  char a[MaxN + 2];
  int g[2][MaxN + 1], s0[MaxN + 1], s1[MaxN + 1];
  int n, k, nn, kk, i, kand, naj;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("razcep.in", "rt");
  fscanf(f, "%d %d\n", &n, &k);
  fgets(a, MaxN + 2, f); fclose(f);

  /* s0[i] in s1[i] naj bosta število ničel in enic v a[i..n − 1]. */
  for (i = n − 1, s0[n] = 0, s1[n] = 0; i >= 0; i−−) {
    s0[i] = s0[i + 1] + (a[i] == '0' ? 1 : 0);
    s1[i] = s1[i + 1] + (a[i] == '1' ? 1 : 0); }

  /* g(kk, nn) nam pomeni oceno najboljšega možnega razcepa
     niza a[0..nn − 1] na kk nepraznih podnizov. Hranili ga bomo v
```

```
      g[kk % 2][nn]. Za začetek jih izračunajmo za kk = 1. */
  for (nn = 0; nn <= n; nn++)
    g[1][nn] = (s0[0] − s0[nn] < s1[0] − s1[nn]) ? s0[0] − s0[nn] : s1[0] − s1[nn];

  /* Rešimo še podprobleme za večje kk. */
  for (kk = 2; kk <= k; kk++) for (nn = kk; nn <= n; nn++)
  {
    naj = n + 1; /* To je večje od ocene vsakega razbitja. */
    /* Niz a[0..nn-1] hočemo razbiti na kk nepraznih podnizov. Zadnji med
       njimi bo torej oblike a[i..nn − 1] za nek i, pred tem pa imamo tedaj
       problem, kako razbiti a[0..i − 1] na kk − 1 nepraznih podnizov. */
    for (i = kk − 1; i < nn; i++) {
      /* Izračunajmo oceno zadnjega kosa, a[i..nn−1]. */
      kand = s0[i] − s0[nn];
      if (kand > s1[i] − s1[nn]) kand = s1[i] − s1[nn];
      /* Prištejmo še oceno najboljšega razbitja niza a[0..i − 1]
         na kk − 1 nepraznih podnizov. */
      kand += g[(kk − 1) % 2][i];
      /* Če je to najboljša rešitev doslej, si jo zapomnimo. */
      if (kand < naj) naj = kand; }
    /* Shranimo rezultat v tabelo g. */
    g[kk % 2][nn] = naj;
  }

  /* Izpišimo rezultat, torej g(k, n), ki ga hranimo v g[k % 2][n]. */
  f = fopen("razcep.out", "wt");
  fprintf(f, "%d", g[k % 2][n]); fclose(f); return 0;
}
```

Pri vsakem paru (n', k') imamo $O(n')$ dela, da pregledamo vse možne i in poiščemo najboljši razcep. Časovna zahtevnost tega postopka je torej $O(n^2 k)$. Za kratke nize, kot so bili tisti pri našem tekmovanju, je to dovolj hitro; mogoče pa je to rešitev še izboljšati.

Vrnimo se k prej omenjeni rekurzivni formuli za $g(n', k')$. V njej med drugim nastopa ocena podniza $a_{i+1} \dots a_{n'}$; spomnimo se, da lahko to oceno izrazimo kot $f(a_{i+1} \dots a_{n'}) = \min\{s_c[i] - s_c[n'] : 0 \le c \le 1\}$. Če nesemo to v formulo za $g(n', k')$, dobimo:

$$g(n', k') = \min\{g(i, k'-1) + s_c[i] - s_c[n'] : k'-1 \le i < n', 0 \le c \le 1\}.$$

Ker je c neodvisen od i, lahko pri vsakem c posebej izračunamo minimum po vseh i in nato vzamemo minimum po obeh možnih c:

$$g(n', k') = \min\{\min\{g(i, k'-1) + s_c[i] - s_c[n'] : k'-1 \le i < n'\} : 0 \le c \le 1\}.$$

Opazimo lahko, da je $s_c[n']$ neodvisen od i, zato ga lahko nesemo ven iz notranjega min:

$$g(n', k') = \min\{\min\{g(i, k'-1) + s_c[i] : k'-1 \le i < n'\} - s_c[n'] : 0 \le c \le 1\}.$$

Kaj se zgodi, če namesto n' vzamemo $n' + 1$, torej če razbijamo še za en znak daljši podniz? V notranjem min so posamezni členi enaki kot prej, le da se jim

pridruži še en nov člen (za $i = n'$, ki zdaj ustreza omejitvi $i < n' + 1$, prej pa ni ustrezal omejitvi $i < n'$). Koristno je torej, če si za vsak c posebej hranimo vrednost notranjega min; ko se premaknemo z n' na $n' + 1$, lahko obe vrednosti poceni popravimo (saj moramo le pogledati, če je novi člen $g(n', k' - 1) + s_c[n']$ kaj manjši od dosedanjega minimuma), nato pa izračunamo $\min\{\ldots\} - s_c[n']$ za oba c-ja in pogledamo, kateri je manjši. Tako imamo pri vsakem (n', k') le $O(1)$ dodatnega dela in časovna zahtevnost celotnega postopka je le še $O(nk)$.

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Lov na sataniste

V zanki se sprehodimo po nizu s od leve proti desni in iščimo pojavitve znaka „6“ ali podniza „six“. Ko najdemo kakšno tako pojavitev, nas zdaj načeloma zanima, ali se ta pojavitev skupaj s prejšnjima dvema (če smo doslej sploh že videli dve pojavitvi) pojavlja znotraj območja p ali manj znakov. Koristno bi si bilo torej zapomniti, na katerih indeksih v nizu sta se začeli prejšnji dve pojavitvi; spodnja rešitev v ta namen uporablja spremenljivki a in b (b je zadnja pojavitev pred trenutno, a pa predzadnja). Tako lahko zelo poceni preverimo, ali je od začetka predprejšnje pojavitve (na indeksu a) do konca trenutne (ki se začne na indeksu c in konča na indeksu k; pri pojavitvi znaka „6“ je $k = c$, pri pojavitvi podniza „six“ pa je $k = c + 2$) kvečjemu p znakov ali ne. Nato lahko predprejšnjo pojavitev pozabimo in si za nadaljnje pregledovanje niza zapomnimo prejšnjo in trenutno.

```
bool Preveri(const char *s, int p)
{
  int a = −p − 1, b = −p − 1, c, k;
  for (c = 0; s[c]; c++) {
    /* Preverimo, ali se na c začne pojavitev šestice;
       če se, si v k zapomnimo, kje se konča. */
    if (s[c] == 's' && s[c + 1] == 'i' && s[c + 2] == 'x') k = c + 2;
    else if (s[c] == '6') k = c;
    else continue; /* Na c se ne začne pojavitev šestice. */
    /* Našli smo pojavitev šestice, ki pokriva indekse od c do k.
       Prejšnji dve pojavitvi se začneta na indeksih a in b.
       Ali je od a do k največ p znakov? */
    if (k − a + 1 <= p) return true;
    /* Popravimo indeksa zadnjih dveh pojavitev. */
    a = b; b = c; }
  /* Če pridemo do sem, vemo, da nismo našli primerne skupine treh pojavitev. */
  return false;
}
```

2. Hišna številka

Ker so pri tej nalogi hišne številke majhne, gre lahko naša funkcija kar z zanko od $n + 1$ naprej in po vrsti za vsako številko preveri, ali je primerne oblike ali ne; čim najdemo kakšno primerno, zanko prekinemo in vrnemo pravkar najdeno številko. Naloga pravi, da bo vhodni n največ 1 000 000, torej bomo primeren rezultat našli najkasneje pri številki 1 000 001.

```
int Naslednja(int n)
{
  do { n++; } while (! JePrimerna(n));
  return n;
}
```

Zdaj moramo napisati še funkcijo JePrimerna, ki naj bi preverila, ali je dana številka n primerne oblike. Za začetek jo je koristno razbiti na posamezne števke in si jih shraniti v neki tabeli; spodnja funkcija ima v ta namen tabelo stevke, v spremenljivki d pa si zapomnimo, koliko števk imamo. Če označimo indekse števk od 0 do $d - 1$,

vidimo, da pri obračanju tablice s hišno številko pride števka, ki je bila prej na indeksu i, zdaj na indeks $d - 1 - i$. Za vsak tak i moramo torej preveriti, če je števka na indeksu $d - 1 - i$ ravno obrnjena različica števke z indeksa i. Pri tem si bomo pomagali s tabelo obrnjenaStevka, ki nam za vsako števko od 0 do 9 pove, katera je njena obrnjena različica (pri tistih, ki ob obračanju postanejo neveljavne, naj bo tudi v obrnjenaStevka neka neveljavna vrednost, na primer -1, ki se zagotovo razlikuje od katerekoli števke n-ja).

```
const int obrnjenaStevka[10] = { 0, 1, −1, −1, −1, −1, 9, −1, 8, 6 };

bool JePrimerna(int n)
{
   int stevke[7], d = 0, i;
   /* Razbijmo n na posamezne števke; v d si zapomnimo, koliko jih je. */
   while (n > 0) { stevke[d++] = n % 10; n /= 10; }
   /* Preverimo, če se število obrne v samo sebe. */
   for (i = 0; i <= d − 1 − i; i++)
      if (stevke[i] != obrnjenaStevka[stevke[d − 1 − i]]) return false;
   return true;
}
```

Kot zanimivost omenimo, da je primernih hišnih številk do 10^6 le 198. Možna rešitev bi bila zato na primer tudi ta, da bi si vse te številke izračunali naprej in jih shranili v tabeli, nato pa bi funkcija Naslednja(n) morala le poiskati v tej tabeli najmanjšo številko, večjo od n (to bi se dalo narediti še posebej učinkovito, če bi imeli številke v tabeli urejene in bi jo lahko preiskovali z bisekcijo).

3. Stolpnica

Podprtost je koristno preverjati po vrsticah od spodaj navzgor. Imejmo tabelo podprta, ki za vsako kocko trenutne vrstice pove, ali je podprta ali ne. V spodnji vrstici so vse kocke podprte že po definiciji. Ko se premaknemo iz vrstice $y - 1$ navzgor v vrstico y, se podprtost prenese navzgor povsod, kjer kocka v vrstici y leži nad istoležno kocko v vrstici $y - 1$; nato gremo lahko v zanki po trenutni vrstici od leve proti desni in prenašamo podprtost od vsake kocke na njeno desno sosedo, podobno pa gremo nato še od desne proti levi in prenašamo podprtost od vsake kocke na njeno levo sosedo. Na koncu preverimo, če so vse kocke v trenutni vrstici podprte; če niso, lahko pregledovanje takoj končamo, sicer pa se premaknemo eno vrstico navzgor in s postopkom nadaljujemo.

```
#define w . . .
#define h . . .
bool T[h][w];

bool Preveri()
{
   int x, y; bool podprta[w];
   /* V spodnji vrstici so vse kocke podprte, zato je ni treba preverjati.
      Preglejmo ostale vrstice. */
   for (y = 1; y < h; y++)
   {
      /* Na tem mestu vemo, da so vse kocke v vrstici y − 1 podprte.
```

```
        V vrstici y so zaradi njih podprte tiste kocke, ki stojijo na
        kakšni kocki iz vrstice y − 1. */
      for (x = 0; x < w; x++) podprta[x] = T[y − 1][x] && T[y][x];

      /* Vsaka kocka v vrstici y podpira tudi svoje sosede na levi in desni. */
      for (x = 1; x < w; x++)
        if (podprta[x − 1] && T[y][x]) podprta[x] = true;
      for (x = w − 2; x >= 0; x−−)
        if (podprta[x + 1] && T[y][x]) podprta[x] = true;

      /* Preverimo, če so vse kocke v tej vrstici podprte. */
      for (x = 0; x < w; x++)
        if (T[y][x] && ! podprta[x]) return false;
    }
  return true;
}
```

Nalogo lahko rešimo tudi brez pomožne tabele podprta. Čim naš postopek v neki vrstici opazi kakšno nepodprto kocko, takoj odneha in vrne **false**; zato, ko se ukvarjamo z vrstico y, lahko predpostavimo, da so vse kocke v vrstici $y - 1$ podprte. Recimo zdaj, da v vrstici y opazimo strnjeno skupino kock od $x = x_1$ do $x = x_2$. Ta skupina kock je podprta natanko tedaj, če na vsaj eni od teh x-koordinat (za $x_1 \leq x \leq x_2$) stoji tudi kocka v vrstici $y - 1$. To, ali kakšna taka kocka obstaja, lahko preverjamo že sproti, ko ugotavljamo, kako daleč se sploh razteza naša strnjena skupina v vrstici y. Tako dobimo naslednjo rešitev:

```
bool Preveri2()
{
  int x1, x2, y; bool ok;
  for (y = 1; y < h; y++)
    for (x1 = 0; x1 < w; x1++) if (T[y][x1]) {
      /* Poglejmo, do kod se razteza strnjena skupina kock v vrstici y,
         ki se začne pri x = x1. Spotoma si v spremenljivki ok še zapomnimo,
         ali je vsaj kakšna od teh kock podprta od spodaj. */
      for (ok = false, x2 = x1; x2 < w && T[y][x2]; x2++)
        if (T[y − 1][x2]) ok = true;

      /* Če je vsaj ena od kock v skupini podprta od spodaj, je cela skupina
         podprta, sicer pa ni nobena kocka v skupini podprta. */
      if (! ok) return false;

      /* Naša strnjena skupina kock se pravzaprav razteza od x1 do vključno x2 − 1;
         polje (x2, y) je že prazno. Zdaj lahko postavimo x1 na x2 in v nadaljevanju
         naše zanke po x1 bomo to prazno polje takoj preskočili. */
      x1 = x2; }

  return true;
}
```

4. Kontrolne naloge

Pri predmetu i imamo k_i nalog; poljubno jih razdelimo na nekaj skupin s po n nalogami, le zadnja skupina je lahko manjša, ker pač ni nujno, da se deljenje k_i/n izide. To naredimo za vsak predmet in tako dobljene skupine nalog uredimo padajoče po številu nalog.

Naloga pravi, da lahko vsak dober dijak reši po eno tako skupino nalog na dan in da imamo d dobrih dijakov ter t dni. V tem času lahko torej dobri dijaki rešijo do

$d \cdot t$ skupin nalog. Če nam je nastalo $d \cdot t$ ali manj skupin nalog, bodo dobri dijaki lahko kar sami rešili vse in je problem rešen.

Drugače lahko razdelimo naloge med dijake takole. Slabi dijaki lahko rešijo poljubne naloge, vendar le po eno na dan; ker jih je s, lahko v t dneh rešijo največ $s \cdot t$ nalog. Vprašanje je torej, če lahko dobrim dijakom razdelimo naloge tako, da jih bo za slabe dijake ostalo kvečjemu $s \cdot t$. Ker lahko dobrim dijakom razdelimo poljubnih $d \cdot t$ skupin, jim je smiselno razdeliti *največjih* $d \cdot t$ skupin — tako bomo zagotovili, da bodo dobri dijaki rešili največje možno število nalog in da jih bo za slabe dijake ostalo kar najmanj. Če slabim dijakom kljub temu ostane več kot $s \cdot t$ nalog, pa bomo vedeli, da je problem nerešljiv (vseh nalog ni mogoče rešiti v t dnevih).

Razmislimo malo še o tem, kako bi ta postopek učinkovito izvedli v praksi. Pri vsakem predmetu nam nastane največ ena skupina z manj kot n nalogami; vse ostale skupine imajo po n nalog. Za slednje je dovolj že, če si izračunamo, koliko jih je; manjše skupine, ki jih je največ p (pri vsakem predmetu največ ena), pa shranimo v neko tabelo in jo uredimo padajoče. Tako dobimo rešitev s časovno zahtevnostjo $O(p \log p)$ in prostorsko zahtevnostjo $O(p)$. Zapišimo dobljeni postopek še s psevdokodo:

```
(* V K bomo hranili število še nerazdeljenih nalog. *)
K := 0; for i := 1 to p do K := K + k_i;
L := prazen seznam; (* skupine z manj kot n nalogami *)
D := d · t; (* koliko skupin še lahko razdelimo dobrim dijakom *)

(* Razdelimo dobrim dijakom skupine velikosti n,
   manjše skupine pa shranimo v seznam L. *)
for i := 1 to p:
  △ := min{⌊k_i/n⌋, D}; (* toliko skupin velikosti n bomo razdelili dobrim dijakom *)
  D := D − △; K := K − n · △;
  dodaj (k_i mod n) v seznam L;

(* Razdelimo dobrim dijakom preostale (manjše) skupine. *)
uredi seznam L padajoče;
while D > 0 and L ni prazen:
  pobriši prvi element iz L-ja in si ga zapomni v △;
  D := D − 1; K := K − △;

(* Ali lahko slabi dijaki rešijo vse preostale naloge? *)
return K ≤ s · t;
```

Še ena možnost je, da si v neki tabeli za vsako možno velikost skupine (od 1 do n nalog) zapomnimo, koliko skupin te velikosti imamo. Tako imamo postopek s časovno zahtevnostjo $O(n+p)$ in prostorsko zahtevnostjo $O(n)$; to je bolje od prejšnje rešitve, če je število predmetov p veliko v primerjavi z n.

5. Tehtnica

Označimo s $f(n)$ najmanjše število uteži, ki jih moramo uporabiti, da uravnovesimo tehtnico, če je v levi posodi predmet z maso n. Če hočemo uravnovesiti tehtnico,

moramo uteži položiti tako, da bo skupna masa uteži v desni posodi za n večja od skupne mase uteži v levi posodi.

Opazimo lahko, da imajo vse uteži sodo maso, razen tiste z maso 1. Če uteži z maso 1 ne uporabimo, bo skupna masa uteži v vsaki od posod soda, torej bo tudi razlika soda; podobno pa, če utež z maso 1 uporabimo, bo skupna masa uteži v eni posodi liha (namreč v tisti, kjer je utež z maso 1), v drugi pa soda, torej bo razlika liha. Mi pa bi radi dosegli razliko n; iz tega vidimo, da če je n sod, uteži z maso 1 ne smemo uporabiti, če pa je n lih, to utež nujno moramo uporabiti.

Oglejmo si za začetek primer, ko je n sod, torej $n = 2k$. Pravkar smo videli, da so v poljubni rešitvi za tak n vse uporabljene uteži sode. Za vsako od njih torej obstaja tudi utež s pol manjšo maso. Torej, če bi vsako utež v naši rešitvi zamenjali s pol lažjo, bi dobili rešitev, pri kateri je razlika med desno in levo posodo za pol manjša kot prej, torej samo $n/2 = k$ namesto n. Po drugi strani pa, če bi vzeli poljubno rešitev za k in v njej vsako utež zamenjali z dvakrat težjo, bi dobili rešitev z razliko $2k = n$ namesto k. Vidimo torej, da za vsako rešitev za k obstaja tudi rešitev z enako utežmi za n in obratno. Najboljša rešitev (taka z najmanj utežmi) za n torej porabi prav toliko uteži kot najboljša rešitev za k. Za sode n torej velja $f(2k) = f(k)$.

Kaj pa, če je n lih, recimo $n = 2k + 1$? Prej smo videli, da v vsaki rešitvi za n nujno moramo uporabiti utež z maso 1. Tu zdaj lahko ločimo dve možnosti: če damo to utež v levo posodo, smo zdaj na istem, kot če bi imeli predmet z maso $n+1$ namesto n; če pa damo to utež v desno posodo, smo na istem, kot če bi imeli na začetku predmet z maso $n-1$ namesto n. Od teh dveh možnosti moramo vzeti tisto, ki zahteva manj uteži; tako smo dobili $f(2k + 1) = 1 + \min\{f(2k), f(2k + 2)\}$, kar pa je (kot smo videli že v prejšnjem odstavku) naprej enako $1+\min\{f(k), f(k+1)\}$.

Te ugotovitve že lahko uporabimo, da nalogo rešimo z rekurzivnim podprogramom. Rekurzija se ustavi pri $n = 1$, ko je problem trivialen (tehtnico uravnotežimo z eno utežjo v desni posodi). Tako dobimo:

```
int Tehtnica(int n)
{
  int a, b;
  if (n == 1) return 1;
  else if (n % 2 == 0) return Tehtnica(n / 2);
  else {
    a = Tehtnica(n / 2); b = Tehtnica(n / 2 + 1);
    return 1 + (a < b ? a : b); }
}
```

Slabost te rešitve je, da se lahko rekurzivni klici neugodno namnožijo in da pri njih rešujemo ene in iste podprobleme po večkrat. Razmislimo o tem, kako jo lahko še izboljšamo. Do dveh rekurzivnih klicev pride pri lihih $n = 2k + 1$, kjer moramo rešiti manjša problema za dve zaporedni masi, k in $k + 1$. Če je k sod, recimo $k = 2m$, imamo naprej $f(k) = f(m)$ in $f(k + 1) = 1 + \min\{f(m), f(m + 1)\}$; če pa je k lih, recimo $k = 2m + 1$, imamo naprej $f(k) = 1 + \min\{f(m), f(m + 1)\}$ in $f(k + 1) = f(m + 1)$. Tu imamo torej zdaj spet dva še manjša problema za dve zaporedni masi, m in $m + 1$. Tako lahko nadaljujemo proti vse manjšim masam, dokler ne pridemo do trivialno majhnih problemov: $f(1) = f(2) = 1$, saj lahko takrat tehtnico uravnotežimo že z eno samo utežjo v desni posodi.

Koristno je torej, če vedno rešujemo problem za dve zaporedni masi skupaj, torej če ob $f(n)$ računamo še $f(n+1)$.

```
/* Naslednja funkcija v *fn vrne f(n), v *fn1 pa f(n + 1). */
void Tehtnica2r(int n, int *fn, int *fn1)
{
  int fk, fk1;
  if (n == 1) { *fn = 1; *fn1 = 1; return; }
  Tehtnica2r(n / 2, &fk, &fk1);
  if (n % 2 == 0)
  {
    /* f(n) = f(2k) = f(k);
       f(n + 1) = f(2k + 1) = 1 + min{f(k), f(k + 1)}. */
    *fn = fk;
    *fn1 = 1 + (fk < fk1 ? fk : fk1);
  }
  else
  {
    /* f(n) = f(2k + 1) = 1 + min{f(k), f(k + 1)};
       f(n + 1) = f(2k + 2) = f(k + 1). */
    *fn = 1 + (fk < fk1 ? fk : fk1);
    *fn1 = fk1;
  }
}

int Tehtnica2(int n)
{
  int fn, fn1;
  Tehtnica2r(n, &fn, &fn1);
  return fn;
}
```

Zdaj imamo pri vsakem klicu funkcije Tehtnica2r le en rekurzivni klic s pol manjšo maso, tako da je časovna zahtevnost te rešitve le še $O(\log n)$.

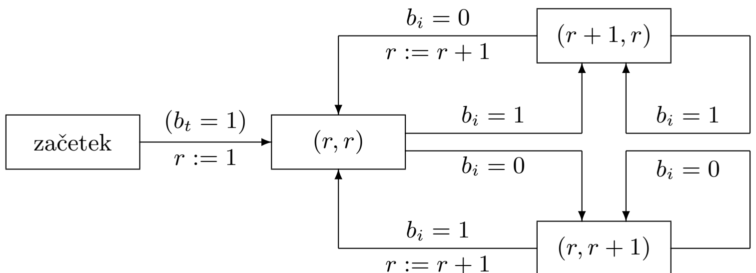
Naša funkcija Tehtnica2r najprej izvede rekurzivni klic za $k = \lfloor n/2 \rfloor$; to je ravno število, ki nastane, če n-ju odrežemo najnižji bit. Ob vrnitvi iz tega klica dobi par rezultatov $\langle f(k), f(k+1) \rangle$; nato pa izračuna $z := 1+\min\{f(k), f(k+1)\}$ in vrne (kot $\langle f(n), f(n+1) \rangle$) bodisi par $\langle f(k), z \rangle$ bodisi $\langle z, f(k+1) \rangle$, odvisno pač od tega, ali je spodnji bit n-ja (tisti, ki smo ga sicer odrezali, ko smo iz n-ja delali k), ugasnjen ali prižgan. Par rezultatov za n torej dobimo tako, da izračunamo par rezultatov za k in nato v njem eno od komponent zamenjamo $1 + \min$ obeh komponent. Pri rekurzivnem klicu za k bi se dogajalo podobno; funkcija bi tudi k-ju odrezala najnižji bit (rezultat je tak, kot če bi n-ju odrezali dva najnižja bita) in nato ta bit (ki je enak kot drugi najnižji bit n-ja) uporabila za odločitev o tem, katero komponento v paru popraviti. Podobno se dogaja tudi pri ostalih še globlje vgnezdenih rekurzivnih klicih.

Zapišimo n po bitih: bit i v številu n označimo z b_i, tako da lahko zapišemo $n = (b_t b_{t-1} \ldots b_2 b_1 b_0)_2 = \sum_{i=0}^{t} b_i 2^i$; pri tem izberimo t tako, da bo najvišji bit prižgan ($b_t = 1$). Označimo z $n_i = (b_t b_{t-1} \ldots b_{i+1} b_i)_2 = \lfloor n/2^i \rfloor = \sum_{j=i}^{t} b_j 2^{j-i}$ število, ki nastane, če n-ju odrežemo spodnjih i bitov; in naj bo $\hat{n}_i = n_i + 1$. Pri $i = 0$ je seveda $n_0 = n$; pri $i = t$ je $n_t = 1$. Zdaj lahko razmislek iz prejšnjega odstavka zapišemo še bolj elegantno: Tehtnica2r izračuna par $\langle f(n_i), f(\hat{n}_i) \rangle$ tako, da

najprej izračuna par $\langle f(n_{i+1}), f(\hat{n}_{i+1})\rangle$ in potem popravi eno od komponent v njem, odvisno od vrednosti bita b_i. Rekurzija se ustavi pri $i = t$, kjer moramo vrniti par $\langle f(1), f(2)\rangle = \langle 1, 1\rangle$. Vidimo torej, da bi lahko naš postopek opisali kar z zanko:

```
x := 1; y := 1;
(* Zdaj je x = f(n_t), y = f(n̂_t). *)
for i := t − 1 downto 0 do begin
   (* Zdaj je x = f(n_{i+1}), y = f(n̂_{i+1}). *)
   z := 1 + min{x, y};
   if b_i = 0 then y := z
             else x := z;
   (* Zdaj je x = f(n_i), y = f(n̂_i). *)
end; (* for *)
(* Zdaj je x = f(n) in y = f(n + 1). *)
return x;
```

Tako zapisan postopek nam pomaga bolje razumeti funkcijo f. V vsaki iteraciji se par (x, y) spremeni v (x, z) ali (z, y), odvisno od bita b_i; pri tem je $z = 1+\min\{x, y\}$. Opazimo lahko, da se x in y vedno razlikujeta za največ 1.[3] Par (x, y) je torej vedno oblike (r, r), $(r + 1, r)$ ali $(r, r + 1)$ za nek r. Obnašanje našega postopka lahko opišemo s takšnim diagramom stanj:



Opazimo lahko, da je vrednost r-ja enaka številu vstopov v stanje (r, r). Prvi vstop se zgodi, ko preberemo b_t (najvišji bit n-ja; ta je gotovo prižgan). Od tam naprej lahko vstopimo v (r, r) največ enkrat na vsaka dva prebrana bita (ko naletimo na par bitov 01 ali 10). Najkrajše zaporedje bitov, s katerim lahko dosežemo neko vrednost r-ja, je torej enica, ki ji sledi $r - 1$ parov 01 ali 10; to zaporedje je dolgo $2r - 1$ bitov.

Zgornja meja za $f(n)$. Zanimivo vprašanje je, kateri je najmanjši n, pri katerem doseže funkcija $f(n)$ neko vrednost, recimo q. Spomnimo se, da naš postopek za izračun $f(n)$ na koncu zanke vrne x, torej prvo komponento stanja, v katerem se nahaja po tistem, ko je prebral vse bite n-ja. Ena možnost, da vrne q, je torej ta, da

[3] O tem se prepričamo z indukcijo. Na začetku postopka trditev drži, saj sta x in y enaka ($x = y = 1$). Če sta x in y enaka na začetku iteracije, bo z za eno večji od njiju in ko bomo enega od x in y povozili s tem z-jem, se bosta po novem x in y razlikovala za 1. Če pa sta se x in y že na začetku iteracije razlikovala za 1, bo $min\{x, y\}$ manjši od njiju, $1 + \min\{x, y\}$ pa večji od njiju; z je torej enak večjemu od njiju in ko enega od x in y povozimo s tem z-jem, bodisi postaneta x in y enaka (če smo povozili manjšega) bodisi se nič ne spremeni (če smo povozili večjega) in se torej še vedno razlikujeta za 1.

je bil na koncu v stanju (r, r) ali $(r, r+1)$ za $r = q$; druga možnost pa je, da je bil v stanju $(r+1, r)$ za $r = q-1$. Kot smo videli malo prej, bi za to, da dosežemo $r = q$, potrebovali vsaj $2q-1$ bitov; pri drugi možnosti pa potrebujemo $2(q-1)-1 = 2q-3$ bitov, da dosežemo stanje (r, r) za $r = q-1$, in nato še en bit (prižgan), ki nas iz njega premakne v $(r+1, r)$ (pri čemer se r ne spremeni). Tako vidimo, da je druga možnost boljša, saj nam bo omogočila doseči $f(n) = q$ že z zaporedjem $2q-2$ bitov. Primeren n je torej takšen: najprej enica, nato $q-2$ parov 01 ali 10, nato pa še ena enica. Če hočemo čim manjši n, moramo vedno vzeti 01 namesto 10, saj bodo tako prižgani biti na nižjih mestih. Naš n je torej oblike $(1\,01\,01\ldots01\,1)_2$, pri čemer se 01 pojavi $(q-2)$-krat. Ta n je naprej enak $1 + \sum_{i=0}^{q-2} 2\cdot 4^i = (2^{2q-1}+1)/3$.

Zdaj poznamo najmanjši n, pri katerem je $f(n) = q$. Z drugimi besedami, za poljuben n velja, da iz $f(n) = q$ sledi $n \geq (2^{2q-1}+1)/3$. To neenačbo lahko predelamo v $q \leq (1/2)\lg n + c$ za $c = (1+\lg 3)/2 \approx 2{,}085$. (Pri tej nalogi bomo oznako lg uporabljali za dvojiški logaritem: $\lg\alpha = \beta \Leftrightarrow 2^\beta = \alpha$.) Skupaj z $f(n) = q$ imamo torej $f(n) \leq (1/2)\lg n + c$ — zgornja meja za število uteži, ki jih potrebujemo, da uravnotežimo tehtnico pri predmetu z maso m.

Ta meja je približno pol nižja, kot bi bila, če bi smeli dajati uteži le v desno posodo, v levo pa ne. Takrat bi vedno potrebovali točno toliko uteži, kolikor ima n prižganih bitov; zato bi q uteži potrebovali že pri $n = (11\ldots1)_2 = 2^q - 1$. Podoben razmislek kot prej bi nam zdaj dal mejo $f(n) \leq \lg(n+1)$.

Časovna zahtevnost prve rekurzivne rešitve. Razmislimo še o tem, kakšna je pravzaprav časovna zahtevnost naše prvotne, naivne rekurzivne rešitve (funkcije Tehtnica). Skupno število vseh klicev funkcije, ki se izvedejo pri izračunu $f(n)$, označimo z $g(n)$.

Pri klicu s parametrom n_i se najprej izvede rekurzivni klic za n_{i+1} in, če je bil n_i lih, še za $\hat{n}_{i+1}$. Podobno se lahko hitro prepričamo, da se pri klicu za $\hat{n}_i$ v vsakem primeru izvede rekurzivni klic za $\hat{n}_{i+1}$; če pa je bil $\hat{n}_i$ lih, se izvede tudi klic za n_{i+1}. Vidimo torej, da ob izračunu vrednosti $f(n)$ pride le do rekurzivnih klicev za števila oblike n_i in $\hat{n}_i$ (za i-je od 0 do t).

Število klicev na nivoju i lahko predstavimo s parom (u_i, v_i), pri čemer je u_i število klicev s parametrom n_i, v_i pa število klicev s parametrom $\hat{n}_i$. Na začetku, pri $i = 0$, imamo seveda par $(1, 0)$ — en klic za $n_0 = n$ in nobenega za $\hat{n}_0 = n+1$. Kako se število klicev razvija na nižjih nivojih? Če je $b_i = 0$, je n_i sod, zato vsak klic za n_i povzroči po en klic za n_{i+1}; $\hat{n}_i$ pa je tedaj lih, zato vsak klic za $\hat{n}_i$ povzroči po en klic za n_{i+1} in enega za $\hat{n}_{i+1}$. Tako imamo torej $u_{i+1} := u_i + v_i$ klicev za n_{i+1} in $v_{i+1} := v_i$ klicev za $\hat{n}_{i+1}$. Če je $b_i = 1$, razmišljamo podobno: n_i je lih, zato vsak klic za n_i povzroči po en klic za n_{i+1} in enega za $\hat{n}_{i+1}$; $\hat{n}_i$ pa je tedaj sod in vsak njegov klic povzroči po en klic za $\hat{n}_{i+1}$. Tako imamo skupaj $u_{i+1} := u_i$ klicev za n_{i+1} in $v_{i+1} := u_i + v_i$ klicev za $\hat{n}_{i+1}$.

Vidimo torej, da iz para (u_i, v_i) en nivo nižje nastane bodisi par $(u_i + v_i, v_i)$ bodisi $(u_i, u_i + v_i)$, odvisno pač od tega, ali je b_i ugasnjen ali prižgan. V vsakem primeru se eno od števil v paru zamenja z vsoto $u_i + v_i$. Ker nas zanima časovna zahtevnost našega postopka v najslabšem možnem primeru, bi radi sestavili tak n, pri katerem bo prišlo do čim večjega števila rekurzivnih klicev. Da bodo števila klicev v naših parih čim hitreje naraščala, je smiselno na vsakem koraku povoziti manjšo od obeh vrednosti v paru. Tako lahko iz $(1, 0)$ naredimo $(1, 1)$, nato $(2, 1)$,

nato $(2,3)$, nato $(5,3)$, nato $(5,8)$ in tako naprej. V tem zaporedju vidimo, da izmenično popravljamo desno in levo komponento para, kar pomeni, da se morajo v n-ju izmenično pojavljati prižgani in ugasnjeni biti (če jih gledamo od manjših proti višjim):

$$\begin{array}{cccccccccccc} i=0 & & 1 & & 2 & & 3 & & 4 & & 5 & \\ (1,0) & \xrightarrow{b_0=1} & (1,1) & \xrightarrow{b_1=0} & (2,1) & \xrightarrow{b_2=1} & (2,3) & \xrightarrow{b_3=0} & (5,3) & \xrightarrow{b_4=1} & (5,8) & \xrightarrow{b_5=0} \cdots \end{array}$$

Naš n bo torej oblike $(1010\ldots101)_2$; biti na sodih mestih so prižgani, na lihih pa ugasnjeni. Najvišji prižgani bit označimo s t, tako kot prej; pri našem sedanjem n-ju je t gotovo sodo število. Vidimo lahko, da bi $3n$ v dvojiškem zapisu sestavljalo neprekinjeno zaporedje $t+2$ enic, tako da imamo $3n = 2^{t+2} - 1$, torej $n = (2^{t+2} - 1)/3$.

Opazimo lahko tudi, da so naši pari sestavljeni iz Fibonaccijevih števil (definiranih s pravilom $f_0 = 0$, $f_1 = 1$, $f_{r+2} = f_r + f_{r+1}$). Za sode i imamo pare oblike (f_{i+1}, f_i), za lihe i pa pare oblike (f_i, f_{i+1}). Pri $i = t$ (ki je sod) to med drugim pomeni, da bomo imeli f_{t+1} klicev s parametrom $n_t = 1$ in še f_t klicev s parametrom $\hat{n}_t = 2$, vsak od slednjih pa bo povzročil še po en klic s parametrom 1. Skupno število vseh klicev je pri našem n-ju torej enako $g(n) = S + f_t$ za $S = \sum_{i=0}^{t}(f_i + f_{i+1})$. Da bomo vsoto S lažje poenostavili, označimo $s = \sum_{i=0}^{t} f_i$. Zdaj lahko S po eni strani razbijemo na dve vsoti: $S = \sum_{i=0}^{t} f_i + \sum_{i=0}^{t} f_{i+1}$; prva je enaka s, druga pa je s brez člena f_0 (ki pa je tako ali tako enak 0), a hkrati z novim členom f_{t+1}; torej imamo $S = 2s + f_{t+1}$. Po drugi strani lahko v definiciji S-ja zamenjamo $f_i + f_{i+1}$ s f_{i+2}, tako da imamo $S = \sum_{i=0}^{t} f_{i+2} = s - f_0 - f_1 + f_{t+1} + f_{t+2}$. Če obe ugotovitvi združimo, dobimo $s = f_{t+2} - 1$, iz tega pa $g(n) = S + f_t = 2s + f_{t+1} + f_t = 2s + f_{t+2} = 3f_{t+2} - 2$.

Spomnimo se, da za Fibonaccijeva števila velja $f_r = (\phi^r - \theta^r)/\sqrt{5}$ za $\phi = (1+\sqrt{5})/2 \approx 1{,}62$ in $\theta = (1-\sqrt{5})/2 \approx -0{,}62$. Ker je θ po absolutni vrednosti manjša od 1, pada zaporedje θ^r hitro proti 0, zato velja približno $f_r \approx \phi^r/\sqrt{5}$ (in ta približek postaja tem bolj natančen, čim večje r gledamo). V našem primeru lahko torej za $g(n)$ rečemo $g(n) \approx 3\phi^{t+2}/\sqrt{5} - 2$. Po drugi strani smo malo prej videli, da je $n = (2^{t+2} - 1)/3$, torej $t = \lg(3n+1) - 2$. Če vstavimo to v formulo za $g(n)$, dobimo $g(n) \approx (3/\sqrt{5})\phi^{\lg(3n+1)} - 2 \approx (3/\sqrt{5})\phi^{\lg(3n)} = (3/\sqrt{5})(3n)^{\lg\phi} = C\, n^{\lg\phi}$ za $C = (3/\sqrt{5})3^{\lg\phi} \approx 2{,}88$. Eksponent pri n-ju, $\lg\phi$, je približno 0,69.

Doslej smo se ukvarjali le z n-ji oblike $n = (1010\ldots101)_2$, ki smo jih izbrali zato, ker je bilo videti, da ravno tu nastopi največ rekurzivnih klicev. Prepričajmo se, da je to res; pri dosedanjih n-jih smo videli, da je $g(n) \approx C\, n^{\lg\phi}$, zdaj pa bomo pokazali, da za vsak n (tudi če ni oblike $(1010\ldots101)_2$) velja $g(n) \le C\, n^{\lg\phi}$.

O tem se lahko prepričamo z indukcijo. Iz funkcije `Tehtnica` vidimo, da bi se dalo število klicev $g(n)$ izraziti rekurzivno: za sode $n = 2k$ imamo $g(n) = 1 + g(k)$, za lihe $n = 2k+1$ pa imamo $g(n) = 1 + g(k) + g(k+1)$. Za majhne n (do $n = 12$ bo dovolj) lahko izračunamo g kar po teh rekurzivnih formulah in z računalnikom preverimo, da trditev $g(n) \le C\, n^{\lg\phi}$ tu res drži.

Recimo zdaj, da smo našo trditev dokazali že za vse $n < m$. Kaj se zgodi pri m? Ena možnost je, da je m sod, torej je $g(m) = 1 + g(m/2)$, kar je po induktivni predpostavki naprej $\le 1 + C\,(m/2)^{\lg\phi}$. Kdaj je to $\le C\, m^{\lg\phi}$ (kar pravi naša trditev)? Iz neenačbe $1 + C\,(m/2)^{\lg\phi} \le C\, m^{\lg\phi}$ dobimo $1 \le C'\, m^{\lg\phi}$ za $C' = C(1 - (1/2)^{\lg\phi})$. Slednji je večji od 1 ($C' \approx 7{,}53$); drugi faktor, $m^{\lg\phi}$, pa je tudi ≥ 1, saj je $m \ge 1$

in eksponent je pozitiven; torej je zmnožek $C' \, m^{\lg \phi}$ tudi ≥ 1, prav to pa smo hoteli dokazati.

Druga možnost je, da je m lih; tedaj je $g(m) = 1 + g((m-1)/2) + g((m+1)/2)$; ker sta $(m \pm 1)/2$ oba soda, je $g(m)$ naprej enako $3 + g((m-1)/4) + g((m+1)/4)$, to pa je po induktivni predpostavki $\leq 3 + C[(m-1)/4]^{\lg \phi} + C[(m+1)/4]^{\lg \phi}$; ker je drugi člen manjši od tretjega, je to naprej $\leq 3 + 2C[(m+1)/4]^{\lg \phi} = 3 + C(m+1)^{\lg \phi} C''$ za $C'' = 2(1/4)^{\lg \phi} \approx 0{,}76$. To je naprej enako $3 + Cm^{\lg \phi} C''(1 + 1/m)^{\lg \phi}$. Spomnimo se, da je $m \geq 13$, saj smo na začetku rekli, da primere do 12 preverimo posebej; zato je $C''(1 + 1/m)^{\lg \phi} \leq C''(14/13)^{\lg \phi} < 0{,}81$. Zdaj torej vidimo, da je $g(m) \leq 3 + 0{,}81 Cm^{\lg \phi}$, dokazati pa želimo, da je to naprej $\leq Cm^{\lg \phi}$. To bo držalo, če je $3 \leq 0{,}19 Cm^{\lg \phi}$, torej $(3/(0{,}19C))^{1/\lg \phi} \leq m$; leva stran je približno 11,6, mi pa imamo $m \geq 13$, tako da je pogoj zagotovo izpolnjen.

Tako torej vidimo, da naša trditev res drži za vsak n: torej je $g(n) \leq C \, n^{\lg \phi}$. Časovna zahtevnost naše funkcije Tehtnica je torej $O(n^{\lg \phi})$, kar je približno $O(n^{0{,}694})$. Ta rešitev je torej sublinearna v odvisnosti od n-ja, kar pa še vseeno pomeni, da je eksponentno slabša od rešitev z zahtevnostjo $O(\log n)$.

Naloge so sestavili: mafijski semenj, hišna številka — Nino Bašić; kompresija — Primož Gabrijelčič; kontrolne vsote, stolpnica — Boris Gašperin; delni izid, kontrolne naloge — Tomaž Hočevar; znajdi.se — Jurij Kodre; polaganje plošč, nurikabe — Mitja Lasič; analiza signala — Matjaž Leonardis; dva od petih — Mark Martinec; kodiranje — Mark Martinec in Janez Brank; it's raining cubes, strahopetni Hektor, Golovec — Jure Slak; lov na sataniste, tehtnica — Mitja Trampuš; trgovanje z zrni — Patrik Zajec; razcep niza — Janez Brank.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2013

1. Statistika na besedilu

Vhodno besedilo lahko beremo znak po znak in v neki tabeli (v spodnjem programu je to stCrk) štejemo pojavitve vsake črke. Pri tem pazimo na to, da štejemo tako velike kot male črke, ostale znake pa preskočimo. Da iz spremenljivke, ki predstavlja črko, dobimo indeks v tabelo stCrk (ki mora biti celo število od 0 do 25), si lahko pomagamo z dejstvom, da lahko v C-ju posamezni znak uporabljamo tudi kot celo število (ki predstavlja številsko kodo znaka) in da so velikim črkam angleške abecede dodeljene zaporedne številke kode (po abecedi, tako da ima A najmanjšo). To pomeni, da če spremenljivka c po branju znaka s standardnega vhoda vsebuje številsko kodo neke velike črke angleške abecede, potem je c − 'A' neko število od 0 do 25, ki pove položaj te črke v angleški abecedi. Na podoben način obravnavamo tudi male črke.

Ob branju vhodnega besedila spotoma še štejmo prebrane znake v trenutni vrstici (spodnji program uporablja za to spremenljivko v); ko pridemo do konca vrstice, preverimo, če je ta števec še vedno 0 — to pomeni, da je vrstica prazna in je vhodnega besedila konec; sicer pa postavimo števec nazaj na 0 in nadaljujemo z branjem naslednje vrstice.

Tudi vprašanja lahko beremo znak po znak; nečrkovne znake (to so načeloma le znaki za konec vrstice) pri tem preskočimo oz. ignoriramo. Pri vsakem črkovnem znaku podobno kot zgoraj izračunamo položaj te črke v abecedi, da dobimo število od 0 do 25, in ga uporabimo kot indeks v tabelo stCrk. Poleg te tabele pa potrebujemo še eno (recimo ji stVprasanj), ki šteje, kolikokrat smo že dobili vprašanje za to črko. Ta števec ob vsakem vprašanju povečamo za 1, če pa je bil že pred vprašanjem večji od 0 (to pomeni, da vprašanja za to črko zdaj ne dobivamo prvič), ga tudi izpišemo, saj tako zahteva besedilo naloge.

```
#include <stdio.h>

int main()
{
  int stCrk[26], stVprasanj[26], c, v;
  /* Inicializirajmo obe tabeli. */
  for (c = 0; c < 26; c++) stCrk[c] = 0, stVprasanj[c] = 0;
  /* Preštejmo črke v vhodnem besedilu. */
  v = 0;
  while ((c = fgetc(stdin)) != EOF)
  {
    if (c == '\n') /* Konec vrstice. */
      if (v == 0) break; /* Prazna vrstica. */
      else { v = 0; continue; }
    v++; /* Povečajmo števec znakov v trenutni vrstici. */
    if ('A' <= c && c <= 'Z') /* Velika črka. */
      stCrk[c - 'A']++;
    else if ('a' <= c && c <= 'z') /* Mala črka. */
      stCrk[c - 'a']++;
  }
```

```
  /* Odgovarjajmo na poizvedbe. */
  while ((c = fgetc(stdin)) != EOF)
  {
    if ('A' <= c && c <= 'Z') c -= 'A';
    else if ('a' <= c && c <= 'z') c -= 'a';
    else continue;
    printf("Črka %c se pojavi %d-krat.", 'A' + c, stCrk[c]);
    if (stVprasanj[c]++ > 0)
      printf(" To je že %d. vprašanje za to črko.", stVprasanj[c]);
    printf("\n");
  }
  return 0;
}
```

2. Problematične formule

Formulo lahko beremo znak po znak in pri tem sproti vzdržujemo podatke o tem, kateri oklepaji so trenutno odprti; to bo torej nekakšen seznam, v katerem bodo oklepaji navedeni od najbolj zunanjih do najbolj notranjih (najgloblje vgnezdenih). Če naletimo na oklepaj, ga dodamo na konec seznama trenutno odprtih oklepajev; če pa naletimo na zaklepaj, moramo preveriti, ali se ujema z najbolj notranjim trenutno še odprtim oklepajem — če se ne ujemata, lahko vhodni niz takoj zavrnemo, saj vemo, da oklepaji niso pravilno gnezdeni. To se zgodi na primer pri nizu `[(])`. Če pa se zaklepaj ujema z oklepajem na koncu našega seznama (torej sta npr. oba okrogla ali oba oglata ipd.), je ta oklepaj s tem primerno zaprt in ga lahko pobrišemo s konca našega seznama. Na koncu tega postopka mora biti seznam prazen — če ni, to pomeni, da nekateri oklepaji nimajo pripadajočega zaklepaja in torej formula ni pravilne oblike. Znake, ki niso niti oklepaji niti zaklepaji, lahko kar sproti ignoriramo, saj nimajo nobenega vpliva na to, ali so v formuli oklepaji pravilno gnezdeni (in zaprti) ali ne.

Pri seznamu odprtih oklepajev lahko opazimo, da ga spreminjamo (dodajamo in brišemo elemente) le na koncu (pri najgloblje vgnezdenem oklepaju), zato je to pravzaprav sklad. V spodnji rešitvi ga bomo predstavili kar z nizom oz. tabelo `s`; spremenljivka `sp` nam pove število trenutno odprtih oklepajev (število elementov na skladu), če pa v formuli odkrijemo napako, postavimo `sp` na -1 (takrat preostanek formule le preberemo in ignoriramo). Na koncu formule moramo tako le preveriti, če je `sp` enaka 0.

```
#include <stdio.h>
#define MaxDolz 50000

int main()
{
  char s[MaxDolz]; /* sklad odprtih oklepajev */
  int c, sp;
  /* Preberimo prvi znak prve formule. Brali bomo do konca standardnega vhoda. */
  c = fgetc(stdin);
  while (c != EOF)
  {
    sp = 0; /* Na začetku formule je sklad prazen. */
    /* V c je prvi znak trenutne formule. Brali bomo po znakih do konca vrstice. */
```

```
      for ( ; c != EOF && c != '\n'; c = fgetc(stdin))
      {
        /* Če smo v formuli že odkrili napako, preostanek formule preskočimo. */
        if (sp < 0) continue;
        /* Oklepaje dodajamo na sklad. */                                    (⋆)
        if (c == '(' || c == '[' || c == '{' || c == '<')
          { s[sp++] = c; continue; }
        /* Če smo prebrali zaklepaj, kakšen je pripadajoč oklepaj? */
        if (c == ')') c = '('; else if (c == ']') c = '[';
        else if (c == '}') c = '{'; else if (c == '>') c = '<';
        else continue; /* Druge znake ignoriramo. */
        /* Če je na vrhu sklada pravi oklepaj, ga pobrišemo. */
        if (sp > 0 && s[sp − 1] == c) −−sp;
        /* Drugače vemo, da je v formuli napaka. */
        else sp = −1;
      }
      /* Če je sklad prazen in še nismo zaznali nobene napake, je formula pravilna. */
      if (sp == 0) printf("Pravilno!\n"); else printf("Narobe\n");
      /* Preberimo prvi znak naslednje formule. */
      c = fgetc(stdin);
    }
    return 0;
}
```

Razmislimo zdaj še o težji različici naloge, pri kateri kot posebna vrsta „oklepajev“ nastopajo tudi navpične črte `|` `|`. Nerodno pri teh je, da ne moremo ločiti med oklepajem in zaklepajem. Ko naletimo v formuli na znak `|`, ali naj ga obravnavamo kot oklepaj (in ga torej dodamo na sklad) ali kot zaklepaj (in torej preverimo, če je na vrhu sklada: če je, ga pobrišemo, sicer pa formulo razglasimo za napačno)? Preden se odločimo za eno ali drugo možnost, je koristno pogledati, kaj je trenutno na vrhu sklada.

Če je na vrhu sklada že ena `|`, je koristno trenutno `|` obravnavati kot zaklepaj in torej pobrisati tisto `|` z vrha sklada (ne pa obravnavati trenutne `|` kot oklepaja in zato dodati na sklad še eno `|`) — če ne naredimo tako, bomo znake `|` ves čas le dodajali na sklad, brisali pa jih ne bomo nikoli.

Če pa na vrhu sklada trenutno ni znaka `|` (lahko da je sklad celo prazen), je bolje obravnavati trenutno `|` kot oklepaj in jo dodati na sklad — kajti če bi se namesto tega odločili obravnavati trenutno `|` kot zaklepaj, bi morali takoj razglasiti formulo za napačno, to pa ni dobro (saj bi tako razglasili za napačno vsako formulo, ki vsebuje kakšno `|`).

Tega razmisleka ni težko vključiti v našo rešitev. Vse, kar moramo narediti, je, da pred vrstico (⋆) dodamo takšen stavek:

```
        if (c == '|') {
          /* Če je na vrhu sklada že navpična črta, obravnavamo c kot zaklepaj. */
          if (sp > 0 && s[sp − 1] == c) −−sp;
          else s[sp++] = c; /* Sicer obravnavamo c kot oklepaj. */
          continue; }
```

Oglejmo si zdaj še malo bolj formalen dokaz, da je naša rešitev res pravilna. Za začetek predpostavimo, da naš vhodni niz (formula) sploh ne vsebuje drugih znakov

kot oklepajev, zaklepajev in navpičnih črt (saj druge znake naša rešitev tako ali tako preskoči in nič ne vplivajo na pravilnost formule). Nizu take oblike, v katerem so vsi oklepaji (in navpične črte) zaprti in pravilno gnezdeni, bomo rekli *oklepajski izraz*. Pogoja, da so oklepaji zaprti in pravilno gnezdeni, torej ne bomo kar naprej ponavljali, ampak ga imejmo implicitno v mislih vedno, ko nečemu rečemo „oklepajski izraz".

Recimo, da imamo k vrst oklepajev, ki jih bomo oštevilčili od 1 do k (navpičnih črt ne bomo šteli med oklepaje); oklepaj r-te vrste zapišimo kot $\texttt{(}_r$, pripadajoči zaklepaj pa kot $\texttt{)}_r$. Tako se nam ne bo treba ukvarjati z vsako vrsto oklepajev (oglati, zaviti itd.) posebej.

Oklepajske izraze lahko opišemo z naslednjimi preprostimi pravili: prazen niz (označili ga bomo z ε) je oklepajski izraz; če je w oklepajski izraz, je tudi $\texttt{|}w\texttt{|}$ oklepajski izraz, pa vsak tudi $\texttt{(}_r w\texttt{)}_r$ (za r od 1 do k) je oklepajski izraz; če sta w in x oklepajska izraza (in neprazna niza), je tudi wx (stik nizov w in x) oklepajski izraz; če se za nek niz po dosedanjih pravilih ne da pokazati, da je oklepajski izraz, potem ta niz ni oklepajski izraz.

Da si bomo bolje predstavljali delovanje naše rešitve, jo zapišimo namesto v C-ju še s psevdokodo; dobimo naslednji preprosti postopek, ki gre takole po vhodnem nizu $w = w_1 w_2 \ldots w_n$:

s := prazen sklad;
for $i := 1$ **to** n:
  if $w_i = \texttt{|}$ **then**
    if je na vrhu sklada s znak $\texttt{|}$
    then pobriši ta znak z vrha sklada **else** dodaj $\texttt{|}$ na vrh sklada
  else if $w_i = \texttt{)}_r$ za nek r **then**
    if je na vrhu sklada s znak $\texttt{(}_r$
    then pobriši ta znak z vrha sklada **else return false**
  else if $w_i = \texttt{(}_r$ za nek r **then**
    dodaj $\texttt{(}_r$ na vrh sklada s;
if je sklad s prazen **then return true else return false**;

Če ta postopek vrne **true**, bomo rekli, da je niz w *sprejel*, sicer pa, da ga je *zavrnil*. Da dokažemo pravilnost naše rešitve, moramo dokazati, da ta postopek sprejme natanko tiste nize, ki so oklepajski izrazi, vse ostale pa zavrne.

Pri nadaljnjem razmišljanju o delovanju našega postopka bo prišlo prav naslednje opažanje. Recimo, da naš postopek pri obdelavi znaka w_i pobriše vrhnji element s sklada (to se zgodi, če je w_i zaklepaj in je na vrhu sklada oklepaj iste vrste ali pa če je w_i navpična črta in je tudi na vrhu sklada navpična črta); tisti element smo morali torej nekoč prej prebrati iz niza w in ga dodati na sklad — recimo, da je bilo to pri znaku w_p (za nek $p < i$). Trdimo, da bi naš postopek sprejel niz $w_{p+1} \ldots w_{i-1}$.

O tem se lahko prepričamo takole. (i) V tisto, kar je bilo na skladu v trenutku, ko smo obdelali znak w_p, naš postopek med obdelavo znakov $w_{p+1} \ldots w_{i-1}$ gotovo ni posegal (kajti če bi, bi prvi tak poseg nujno moral biti to, da bi pobrisal s sklada element, ki smo ga nanj dodali pri obdelavi znaka w_p; mi pa smo vendar predpostavili, da se ta element pobriše šele pri obdelavi znaka w_i in nič prej). (ii) Poleg tega je bil na koncu tega, po obdelavi znaka w_{i-1}, na vrhu sklada očitno spet tisti element, ki smo ga dodali pri obdelavi znaka w_p (kajti če ne bi bil, potem ga mi zdajle pri

obdelavi znaka w_i ne bi mogli pobrisati); karkoli se je torej med obdelavo podniza $w_{p+1} \ldots w_{i-1}$ dodalo na sklad, se je med obdelavo tega podniza tudi pobrisalo.

To dvoje, (i) in (ii) skupaj, pa pomeni, da bi ta del postopka (obdelava podniza $s_{p+1} \ldots s_{i-1}$) enako dobro deloval tudi, če bi ga pognali na praznem skladu (nikoli ne bi naletel na težavo, ko bi želel brisati oklepaj z vrha sklada, ta pa bi bil prazen); in na koncu te obdelave bi bil sklad spet prazen. To torej pomeni, da bi naš postopek niz $w_{p+1} \ldots w_{i-1}$ res sprejel, kot smo hoteli dokazati. □

Mimogrede omenimo, da si lahko tudi vsebino sklada predstavljamo kot niz, pri čemer konec niza predstavlja vrh sklada (tisti del sklada, kjer dodajamo in brišemo elemente), začetek niza pa predstavlja dno sklada. Da bo v nadaljevanju manj pisanja, vpeljimo naslednji zapis: $s \xrightarrow{w} t$ naj pomeni, da če je v nekem trenutku vsebina sklada s in nato postopek po vrsti obdela vse znake niza w, bo uspešno prišel do konca (torej ne bo prišel v položaj, ko bi prebral zaklepaj in na vrhu sklada ne bi bilo pripadajočega oklepaja) in bo na koncu vsebina sklada enaka t.

S pomočjo tega zapisa lahko na primer rečemo, da naš postopek niz w sprejme natanko tedaj, ko velja $\varepsilon \xrightarrow{w} \varepsilon$ (pri tem znak ε pomeni prazen niz oz. prazen sklad).

Iz opisa našega postopka lahko vidimo, da za vsak tip oklepajev velja $\varepsilon \xrightarrow{\mathtt{(}_r} \mathtt{(}_r$ in $\mathtt{(}_r \xrightarrow{\mathtt{)}_r} \varepsilon$; in podobno velja $s\mathtt{|} \xrightarrow{\mathtt{|}} s$ za vsak s; če pa se s ne konča na $\mathtt{|}$, velja tudi $s \xrightarrow{\mathtt{|}} s\mathtt{|}$. Ni se težko tudi prepričati, da če $s \xrightarrow{w} t$, potem za poljuben u, ki se ne konča na znak $\mathtt{|}$, velja tudi $us \xrightarrow{w} ut$; in da iz $s \xrightarrow{w} t$ in $t \xrightarrow{x} u$ sledi $s \xrightarrow{wx} u$.

Prepričajmo se zdaj, da naš postopek pravilno reši nalogo — torej da sprejme natanko tiste nize, ki so oklepajski izrazi. Za nize lihe dolžine je stvar preprosta; iz pravil, s katerimi smo definirali oklepajske izraze, se takoj vidi, da noben niz lihe dolžine ni oklepajski izraz. Obenem se tudi vidi, da je pri našem postopku lahko sklad na koncu prazen le, če je bilo enako število dodajanj in brisanj, to pa je mogoče le, če je bil niz sode dolžine; niz lihe dolžine bo torej gotovo zavrnjen.

Razmislimo zdaj še o nizih sode dolžine. Našo trditev (da postopek sprejme natanko tiste nize, ki so oklepajski izrazi) bomo dokazali z indukcijo po dolžini niza. Za $w = \varepsilon$ (prazen niz) je očitno, da je to oklepajski izraz in da ga naš postopek sprejme. Recimo zdaj, da smo dokazali našo trditev za vse nize dolžine največ $n-2$ (za nek sod n) in da zdaj gledamo nek niz w dolžine n.

($\Leftarrow$) Recimo, da je w oklepajski izraz. (1) Ena možnost je, da je w oblike $w = xy$, pri čemer sta x in y dva krajša (neprazna) oklepajska izraza; zato po induktivni predpostavki postopek tadva izraza sprejme, torej velja $\varepsilon \xrightarrow{x} \varepsilon$ in $\varepsilon \xrightarrow{y} \varepsilon$. Iz tega dvojega sledi $\varepsilon \xrightarrow{xy} \varepsilon$, ker pa je xy ravno enako w, lahko zaključimo, da naš postopek sprejme w.

(2) Druga možnost je, da je w oblike $w = \mathtt{(}_r x \mathtt{)}_r$ za nek tip oklepajev r in za nek x, ki je tudi sam oklepajski izraz (in je dolg $n-2$ znakov). Po induktivni predpostavki torej naš postopek sprejme x, torej $\varepsilon \xrightarrow{x} \varepsilon$; torej $\mathtt{(}_r \xrightarrow{x} \mathtt{(}_r$. Delovanje postopka pri obdelavi niza $w = \mathtt{(}_r x \mathtt{)}_r$ je torej takšno: $\varepsilon \xrightarrow{\mathtt{(}_r} \mathtt{(}_r \xrightarrow{x} \mathtt{(}_r \xrightarrow{\mathtt{)}_r} \varepsilon$, torej bo postopek res sprejel niz w.

(3) Tretja možnost je, da je w oblike $w = \mathtt{|}x\mathtt{|}$ za nek x, ki je tudi sam oklepajski izraz. Po induktivni predpostavki torej naš postopek sprejme x, torej $\varepsilon \xrightarrow{x} \varepsilon$. Pri obdelavi niza w najprej naletimo na znak $\mathtt{|}$ in ga dodamo na sklad, ker je bil pred tem sklad še prazen. Nato obdelujemo niz x; če se pri tem nikoli ne zgodi, da

pobrišemo tisti $\mathtt{|}$ z dna sklada, to pomeni, da je ta obdelava potekala popolnoma enako, kot če bi jo pognali na praznem skladu, le da je bil na dnu sklada ves čas tisti dodatni $\mathtt{|}$; na koncu te obdelave torej na skladu ostane le ta $\mathtt{|}$; ko nato preberemo še zadnji znak niza w, ki je spet $\mathtt{|}$, bomo pobrisali $\mathtt{|}$ s sklada in ta bo ostal prazen, zato bomo niz w sprejeli.

Bolj zanimivo je, če med obdelavo niza x (znotraj $w = \mathtt{|}x\mathtt{|}$) nekoč vendarle pobrišemo znak $\mathtt{|}$, ki smo ga na sklad dodali na začetku, pri obdelavi znaka $w_1 = \mathtt{|}$. Niz x je seveda sestavljen iz znakov $x = w_2 w_3 \ldots w_{n-1}$; recimo, da tisti začetni $\mathtt{|}$ pobrišemo z dna sklada pri obdelavi znaka w_i (ta znak mora torej biti $\mathtt{|}$). Če bi obdelovali niz x sam po sebi, torej ne kot del w-ja, in bi torej začeli obdelavo x-a s praznim skladom, bi bil v tem trenutku, pri obdelavi znaka w_i (ki je $(i-1)$-vi znak niza x), sklad torej prazen, torej bi naš postopek takrat na sklad dodal nov element $\mathtt{|}$. Nekoč kasneje pa ta element gotovo pobrišemo s sklada, saj vendar vemo, da je niz x na koncu sprejet (torej se mora njegova obdelava končati s praznim skladom); recimo, da ga pobrišemo pri obdelavi znaka w_j (ki mora torej tudi biti enak $\mathtt{|}$); po obdelavi tega znaka (med obdelavo niza x samega po sebi) je torej sklad spet prazen.

Strnimo dosedanje ugotovitve: niz x je oblike $x = y\mathtt{|}z\mathtt{|}q$ za $y = w_2 \ldots w_{i-1}$, $z = w_{i+1} \ldots w_j$ in $q = w_{j+1} \ldots w_{n-2}$ in pri obdelavi x samega po sebi imamo sklad prazen po obdelavi y in spet po obdelavi $y\mathtt{|}z\mathtt{|}$; oz. z drugimi besedami, $\varepsilon \xrightarrow{y} \varepsilon \xrightarrow{\mathtt{|}z\mathtt{|}} \varepsilon \xrightarrow{q} \varepsilon$. Iz tega torej vidimo, da naš postopek sprejme niza y in q. Poleg tega smo v prejšnjem odstavku videli, da se ob obdelavi znaka $w_j = \mathtt{|}$ pobriše z vrha sklada ravno tisti $\mathtt{|}$, ki smo ga tja dodali ob obdelavi znaka $w_i = \mathtt{|}$; vmesni del niza, to pa je ravno z, je torej niz, ki bi ga naš postopek sprejel.

Ker naš postopek sprejme nize y, z in q in ker so vsi ti nizi krajši od n znakov, so po induktivni predpostavki ti nizi oklepajski izrazi; zato pa iz naše definicije oklepajskih izrazov sledi, da so oklepajski izrazi tudi nizi $\mathtt{|}y\mathtt{|}$, $\mathtt{|}q\mathtt{|}$ in $z\mathtt{|}q\mathtt{|}$. Naš niz w je torej stik dveh krajših oklepajskih izrazov, namreč $\mathtt{|}y\mathtt{|}$ in $z\mathtt{|}q\mathtt{|}$, za take w pa smo že v točki (1) pokazali, da jih naš postopek res sprejme.

($\Rightarrow$) Recimo, da niz w sprejmemo. Njegov prvi znak, w_1, je gotovo oklepaj ali navpična črta, ne pa zaklepaj, saj bi drugače postopek niz že pri tem znaku zavrnil (ker bi poskušal brisati oklepaj s sklada, sklad pa je takrat še prazen). Pri obdelavi w_1 torej dodamo ta znak na sklad; ker pa w kot celoto sprejmemo, mora biti sklad na koncu obdelave niza w prazen, torej moramo ta element, ki smo ga dodali pri obdelavi znaka w_1, prej ali slej tudi pobrisati; recimo, da ga pobrišemo pri znaku w_i. Kot smo videli že prej, to pomeni, da bi bil vmesni del niza, $w_2 \ldots w_{i-1}$, sam po sebi sprejet, če bi pognali naš postopek na njem; in ker je krajši od niza w, to po induktivni predpostavki pomeni, da je ta vmesni del sam zase tudi oklepajski izraz. Ker je bil pred obdelavo znaka w_1 sklad prazen in ker element, ki smo ga takrat dodali na sklad pri obdelavi tega znaka, kasneje pobrišemo pri obdelavi znaka w_i, to pomeni, da je po obdelavi znaka w_i sklad spet prazen. To pa pomeni, da se obdelava preostanka niza, to je $w_{i+1} \ldots w_n$, izvaja natanko tako, kot da bi obdelovali le ta preostanek niza sam po sebi; in ker se ta obdelava konča s sprejemom, to pomeni, da bi naš postopek sprejel tudi niz $w_{i+1} \ldots w_n$ sam po sebi; ker pa je ta krajši od niza w, to po induktivni predpostavki pomeni, da je ta podniz oklepajski izraz. Zdaj torej vidimo, da je w oblike $(_r x)_r y$ ali pa $\mathtt{|}x\mathtt{|}y$ za $x = w_2 \ldots w_{i-1}$ in $y = w_{i+1} \ldots w_n$

in da sta x in y oklepajska izraza; torej je (po naši definiciji oklepajskih izrazov) tudi $(_r x)_r$ oz. $|x|$ oklepajski izraz, zato pa tudi stik njega in niza y, torej niz w sam. □

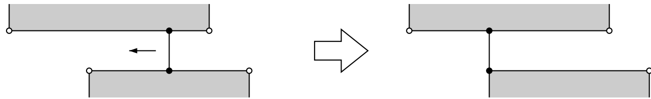
Iz tega razmisleka sledi tudi, da je naša prvotna rešitev (tista, ki se ne ukvarja z navpičnimi črtami) res pravilna rešitev prvotne različice naloge (tiste, ki ima le oklepaje in zaklepaje, navpičnih črt pa ne), saj se na nizih brez navpičnih črt prvotna rešitev obnaša čisto enako kot tale druga, za katero smo pravkar dokazali pravilnost.

3. Miselni vzorec

Daljica, ki jo iščemo, se mora začeti na robu enega pravokotnika in končati na robu drugega pravokotnika. Ker je rob pravokotnika sestavljen iz štirih stranic, lahko nalogo prevedemo na naslednji podproblem: imamo dve stranici (vodoravni in/ali navpični) in bi ju radi povezali s čim krajšo daljico (tako da eno krajišče daljice leži na prvi daljici, eno pa na drugi). Če rešimo ta problem za vseh 4×4 parov stranic (ena stranica prvega pravokotnika in ena stranica drugega) in vzamemo najkrajšo izmed tako dobljenih daljic, bo to ravno rešitev, po kateri sprašuje naša naloga.

Ko iščemo najkrajšo daljico, ki povezuje dve stranici, lahko ločimo dve možnosti: ena možnost je, da se stranici sekata; takrat je najkrajša daljica med njima izrojena v eno samo točko, presečišče obeh stranic, njena dolžina pa je 0. Ker naj bi naši pravokotniki predstavljali miselni vzorec, v njem pa se stvari ponavadi ne sekajo, bi lahko pravzaprav predpostavili, da do te možnosti sploh ne bo prišlo.

Druga možnost je, da se stranici ne sekata; v tem primeru se lahko hitro prepričamo, da je najkrajša daljica med tema dvema stranicama taka, da ima eno od krajišč v enem od oglišč ene od obeh stranic (drugo krajišče daljice pa ni nujno v enem od oglišč druge stranice). Če sta na primer obe stranici vodoravni, lahko razmišljamo takole: recimo, da najkrajša daljica med njima povezuje dve „notranji“ točki stranic (taki, ki nista oglišči); toda to daljico lahko zdaj premikamo v levo (ali pa v desno), dokler eno od njenih krajišč ne doseže oglišča kakšne od stranic; pri tem se dolžina daljice nič ne spremeni, torej imamo še vedno najkrajšo daljico, poleg tega pa se zdaj tudi dotika enega od oglišč. Primer kaže naslednja slika:



Podobno lahko razmišljamo tudi v primeru, če sta obe stranici navpični (tedaj moramo daljico premikati navzgor ali navzdol). Ostane še primer, ko je ena od stranic vodoravna, druga pa navpična. Recimo, da navpična stranica povezuje oglišči (x_n, y_n) in (x_n, y_n+d_n), vodoravna stranica pa oglišči (x_v, y_v) in (x_v+d_v, y_v). Vsaka točka na navpični stranici je torej oblike $(x_n, y_n + \lambda d_n)$ za neko $\lambda \in [0, 1]$, podobno pa je vsaka točka na vodoravni stranici oblike $(x_v + \mu d_v, y_v)$ za neko $\mu \in [0, 1]$. Kakšna je razdalja med tema dvema točkama v odvisnosti od λ in μ? Po Pitagorovem izreku je kvadrat te razdalje enak

$$(x_n - (x_v + \mu d_v))^2 + ((y_n + \lambda d_n) - y_v)^2 .$$

To razdaljo (oz. njen kvadrat) bi radi minimizirali. Vidimo lahko, da je levi člen te vsote odvisen le od μ, desni pa le od λ, torej lahko minimiziramo vsakega posebej. Levi člen lahko predelamo v $((x_n - x_v) - \mu d_v)^2$, kar je kvadratna funkcija spremenljivke μ, ki svoj minimum doseže kar v ničli, pri $\mu_0 = (x_n - x_v)/d_v$; bolj ko pa se μ oddaljuje od te ničle, večja je vrednost funkcije. Če torej μ_0 leži na intervalu $[0, 1]$, moramo uporabiti kar njega, sicer pa za μ vzamemo tisto krajišče intervala $[0, 1]$, ki je bližje vrednosti μ_0. Točka $(x_v, y_v + \mu d_v)$, ki jo na ta način dobimo za krajišče daljice, je pri $\mu = 0$ enaka (x_v, y_v), pri $\mu = 1$ pa dobimo $(x_v, y_v + d_v)$, torej pri teh dveh μ krajišče daljice pride v eno od oglišč naše navpične stranice; pri vmesnih μ (tistih, za katere je $0 < \mu < 1$) pa krajišče daljice pride v eno od notranjih točk te stranice. Prav tak razmislek bi lahko zdaj opravili tudi za drugi del naše vsote in videli, da moramo λ izbrati tako, da je čim bližje vrednosti $\lambda_0 = (y_v - y_n)/d_n$.

Ali se zdaj lahko zgodi, da smo obe krajišči daljice postavili v notranji točki stranic? To bi pomenilo, da imamo $\lambda = \lambda_0$ in $\mu = \mu_0$ (in da tako λ_0 kot μ_0 ležita na intervalu $[0, 1]$), tedaj pa je razdalja med krajiščema enaka 0. To pa je mogoče le, če se stranici sekata — ta primer pa smo obdelali že prej (oz. pravzaprav ugotovili, da do njega najbrž sploh ne more priti, če so vhodni podatki smiselni). Tako torej vidimo, da se v primerih, ko se stranici ne sekata, smemo omejiti na daljice, pri katerih vsaj eno krajišče leži v enem od oglišč ene od stranic.

Tako zdaj pravzaprav ni treba več razmišljati o problemu, kako za vsak par stranic poiskati najkrajšo daljico med njima, ampak smo nalogo prevedli na še lažji problem, kako za vsako oglišče enega pravokotnika in vsako stranico drugega pravokotnika poiskati najkrajšo daljico med njima. Na primer, recimo, da imamo oglišče (x, y) in navpično stranico od (x', y_1) do (x', y_2). Na tej stranici zdaj točki (x, y) leži najbližje točka (x', y') za tisti y', ki leži najbližje vrednosti y; če je y na območju $[y_1, y_2]$, lahko vzamemo kar $y' = y$, sicer pa pač $y' = y_1$ ali $y' = y_2$, odvisno od tega, katero od teh krajišč je bližje y. Zapišimo ta razmislek s podprogramom:

```
/* Poišče najkrajšo daljico od točke (x, y) do navpične stranice
   z ogliščema (xx, y1) in (xx, y2). */
void OglisceStranica(int x, int y, int xx, int y1, int y2,
                     int *dNaj, int *dx1, int *dy1, int *dx2, int *dy2)
{
  int t, yy, d;

  /* Če y1 ni manjši od y2, oglišči zamenjajmo. */
  if (y2 < y1) t = y1, y1 = y2, y2 = t;

  /* Naj bo yy tista koordinata na intervalu [y1, y2], ki je najbližja y. */
  if (y < y1) yy = y1; else if (y > y2) yy = y2; else yy = y;

  /* Točki (x, y) je na stranici od (xx, y1) do (xx, y2) najbližja točka (xx, yy).
     Izračunajmo razdaljo med njima. */
  d = (x - xx) * (x - xx) + (y - yy) * (y - yy);

  /* Če je to najkrajša daljica doslej, si jo zapomnimo. */
  if (*dNaj < 0 || d < *dNaj) *dNaj = d, *dx1 = x, *dy1 = y, *dx2 = xx, *dy2 = yy;
}
```

Kot vidimo, naš podprogram svoje rezultate vrača tako, da vpiše koordinate krajišč dobljene daljice v `*dx1`, `*dy1`, `*dx2` in `*dy2`, kvadrat njene dolžine pa v `*dNaj`, vendar le, če je ta daljica krajša od tiste, ki je bila doslej v `*dNaj`. Tako bomo lahko podprogram klicali po večkrat za razna oglišča in stranice ter na koncu dobili najkrajšo izmed vseh

najdenih daljic. Podprogram predpostavlja, da *dNaj < 0 pomeni, da nimamo sploh še nobene daljice, v tem primeru bo torej zagotovo shranil svojo pravkar dobljeno daljico. (To si lahko privoščimo, ker so kvadrati razdalj drugače vedno ≥ 0, nikoli manjši od 0.)

Če hočemo zdaj za neko oglišče (x, y) prvega pravokotnika poiskati najbližje povezave do vseh štirih stranic drugega pravokotnika, lahko podprogram OglisceStranica pokličemo štirikrat, po enkrat za vsako stranico. Ker ta podprogram pričakuje navpično stranico, naš pravokotnik pa ima tudi vodoravne stranice, si lahko pomagamo tako, da pri tistih klicih zamenjamo x- in y-koordinate; tako se vodoravna stranica spremeni v navpično, razdalje pa se pri takšni transformaciji nič ne spremenijo.

```
/* Poišče najkrajšo daljico od točke (x, y) do roba pravokotnika z diagonalo
   od (x1, y1) do (x2, y2). */
void OgliscePravokotnik(int x, int y, int x1, int y1, int x2, int y2,
                        int *dNaj, int *dx1, int *dy1, int *dx2, int *dy2)
{
  /* Poiščimo razdalje do navpičnih stranic pravokotnika. */
  OglisceStranica(x, y, x1, y1, y2, dNaj, dx1, dy1, dx2, dy2);
  OglisceStranica(x, y, x2, y1, y2, dNaj, dx1, dy1, dx2, dy2);

  /* Poiščimo razdalje do vodoravnih stranic pravokotnika. */
  OglisceStranica(y, x, y1, x1, x2, dNaj, dy1, dx1, dy2, dx2);
  OglisceStranica(y, x, y2, x1, x2, dNaj, dy1, dx1, dy2, dx2);
}
```

Vse, kar moramo zdaj narediti, je, da kličemo ta podprogram po enkrat za vsako oglišče prvega pravokotnika in mu naročimo, naj izračuna razdaljo do roba drugega pravokotnika, ter po enkrat za vsako oglišče drugega pravokotnika in mu naročimo, naj izračuna razdaljo do roba prvega pravokotnika.

```
void Povezi(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4)
{
  int dNaj = −1, dx1, dy1, dx2, dy2, i, j;
  for (i = 0; i < 2; i++) for (j = 0; j < 2; j++) {
    /* Preglejmo daljice, ki imajo eno krajišče v enem od oglišč prvega pravokotnika,
       drugo pa na robu drugega pravokotnika. */
    OgliscePravokotnik((i ? x1 : x2), (j ? y1 : y2), x3, y3, x4, y4,
                       &dNaj, &dx1, &dy1, &dx2, &dy2);

    /* Preglejmo daljice, ki imajo eno krajišče v enem od oglišč drugega pravokotnika,
       drugo pa na robu prvega pravokotnika. */
    OgliscePravokotnik((i ? x3 : x4), (j ? y3 : y4), x1, y1, x2, y2,
                       &dNaj, &dx2, &dy2, &dx1, &dy1); }

  /* Izpišimo rezultat. */
  printf("(%d, %d)-(%d, %d)\n", dx1, dy1, dx2, dy2);
}
```

Dosedanja rešitev temelji na predpostavki, da se nobeni dve stranici ne sekata tako, da bi bilo presečišče v notranjosti obeh stranic. (Pravilno pa bi rešila primere raznih dotikanj, ko (vsaj) eno od oglišč enega pravokotnika leži na eni od stranic drugega pravokotnika.) Kaj pa, če bi vendarle hoteli podpreti tudi takšna sekanja? Vrnimo se spet k razmišljanju o tem, kako s čim krajšo daljico povezati dve stranici (eno z roba prvega pravokotnika in eno z roba drugega).

Če sta obe stranici vodoravni, lahko z enakim razmislekom kot zgoraj vidimo, da se lahko omejimo na daljice, ki imajo eno od krajišč v enem od oglišč ene od stranic. Ta razmislek velja tudi v primerih, če se stranici deloma prekrivata ali pa celo ena povsem leži znotraj druge (le da je takrat pač daljica, ki ju povezuje, izrojena v točko in ima dolžino 0). Recimo brez izgube za splošnost, da gledamo vodoravno stranico AB s pravokotnika $ABCD$ in vodoravno daljico EF z drugega pravokotnika ter da ima najkrajša daljica, ki ju povezuje, eno od krajišč v oglišču A. V prvem pravokotniku to oglišče ne pripada le vodoravni stranici AB, ampak tudi navpični stranici DA, torej bomo isto daljico lahko našli tudi takrat, ko bomo iskali najdaljšo daljico med stranicama DA in EF. Ta razmislek velja na splošno, saj se v vsakem oglišču pravokotnika stikata po ena vodoravna in ena navpična stranica. Tako torej vidimo, da se nam s primeri, ko sta obe stranici vodoravni, sploh ni treba ukvarjati. Podoben razmislek lahko uporabimo tudi v primerih, ko sta obe stranici navpični.

Zdaj nam torej ostanejo le še taki pari stranic, kjer imamo eno vodoravno stranico z roba enega pravokotnika in eno navpično stranico z roba drugega pravokotnika. Tu lahko razmišljamo enako kot prej, le da se nam zdaj lahko zgodi, da tako λ_0 kot μ_0 ležita znotraj območja $(0, 1)$. Naša najkrajša daljica, ki povezuje obe stranici, ima na navpični stranici krajišče (x_n, y_N) za $y_N = x_n + \lambda d_n$, na vodoravni stranici pa krajišče (x_V, y_v) za $x_V = x_v + \mu d_v$. Brez izgube za splošnost recimo, da je $d_n > 0$ (če je $d_n < 0$, lahko oglišči navpične stranice preprosto zamenjamo — x_n popravimo na $x_n + d_n$, nato pa d_n spremenimo v $-d_n$) in podobno $d_v > 0$.

Zdaj lahko razmišljamo takole: (1) Lahko je $\mu_0 < 0$; to z drugimi besedami pomeni $(x_n - x_v)/d_v < 0$, kar je isto kot $x_n < x_v$. Videli smo že, da moramo pri $\mu_0 < 0$ vzeti $\mu = 0$, kar pomeni, da je $x_V = x_v$. — (2) Lahko je $\mu_0 > 0$, kar je isto kot $(x_n - x_v)/d_v > 1$, kar je isto kot $x_n > x_v + d_v$. Videli smo že, da moramo pri $\mu_0 > 1$ vzeti $\mu = 1$, kar pomeni, da je $x_V = x_v + d_v$. — (3) Če pa je $\mu_0 \in [0, 1]$, kar se zgodi pri $x_v \leq x_n \leq x_v + d_v$, moramo vzeti $\mu = \mu_0$, tedaj pa dobimo $x_V = x_v + \mu d_v = x_v + \mu_0 d_v = x_v + (x_n - x_v)/d_v \cdot d_v = x_n$.

Z besedami bi lahko razmislek iz prejšnjega odstavka opisali takole: če leži navpična stranica v celoti levo (oz. v celoti desno) od vodoravne, moramo krajišče naše daljice postaviti v levo (oz. v desno) oglišče vodoravne stranice; sicer pa postavimo krajišče naše daljice na tisto točko vodoravne stranice, ki leži na isti x-koordinati kot navpična stranica.

Na podoben način lahko razmišljamo tudi o λ_0, λ in iz nje dobljenem y_N, tako da določimo še drugo krajišče naše nakrajše daljice, ki povezuje obe stranici. Zapišimo ta razmislek kot podprogram v C-ju:

```
/* Poišče najkrajšo daljico, ki ima eno krajišče na navpični stranici od (xn, yn) do
   (xn, yn + dn), drugo krajišče pa na vodoravni stranici od (xv, yv) do (xv + dv, yv). */
void ParStranic(int xn, int yn, int dn, int xv, int yv, int dv,
                int *dNaj, int *dx1, int *dy1, int *dx2, int *dy2)
{
  int d, xV, yN;

  /* Če je dolžina kakšne stranice negativna, se premaknimo v nasprotno
     oglišče in spremenimo dolžino v pozitivno. */
  if (dn < 0) { yn += dn; dn = −dn; }
  if (dv < 0) { xv += dv; dv = −dv; }
```

```
  /* Določimo x-koordinato krajišča na vodoravni stranici. */
  if (xn < xv) xV = xv;
  else if (xn > xv + dv) xV = xv + dv;
  else xV = xn;

  /* Določimo y-koordinato krajišča na navpični stranici. */
  if (yv < yn) yN = yn;
  else if (yv > yn + dn) yN = yn + dn;
  else yN = yv;

  /* Če je to najkrajša daljica doslej, si jo zapomnimo. */
  d = (xn − xV) * (xn − xV) + (yN − yv) * (yN − yv);
  if (*dNaj < 0 || d < *dNaj) *dNaj = d, *dx1 = xn, *dy1 = yN, *dx2 = xV, *dy2 = yv;
}
```

Zdaj ni težko zapisati podprograma, ki pokliče ParStranic za vsak tak par stranic, kjer je navpična stranica z roba prvega pravokotnika, vodoravna pa z roba drugega pravokotnika. Pri prvem pravokotniku, ki ima diagonalo od (x_1, y_1) do (x_2, y_2), se navpični stranici začneta pri y_1 in sta dolgi $y_2 - y_1$, njuna x-koordinata pa je lahko x_1 ali pa x_2 — ti dve možnosti pregledamo v zanki. Pri vsaki od njiju lahko z vgnezdeno zanko na podoben način pregledamo še obe navpični stranici z roba drugega pravokotnika.

```
/* Poišče najkrajšo povezavo med navpičnimi stranicami prvega pravokotnika
   in vodoravnimi stranicami drugega. */
void PoveziNV(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4,
              int *dNaj, int *dx1, int *dy1, int *dx2, int *dy2)
{
  int i, j;
  for (i = 0; i < 2; i++) for (j = 0; j < 2; j++)
    ParStranic((i ? x1 : x2), y1, y2 − y1, x3, (j ? y3 : y4), x4 − x3,
               dNaj, dx1, dy1, dx2, dy2);
}
```

Isti podprogram lahko uporabimo tudi za primere, ko moramo vzeti navpično stranico z roba drugega pravokotnika in vodoravno z roba drugega pravokotnika — vse, kar moramo narediti, je, da v mislih oba pravokotnika zamenjamo. Naš glavni podprogram bo torej takšen:

```
void Povezi2(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4)
{
  int dNaj = −1, dx1, dy1, dx2, dy2;
  PoveziNV(x1, y1, x2, y2, x3, y3, x4, y4, &dNaj, &dx1, &dy1, &dx2, &dy2);
  PoveziNV(x3, y3, x4, y4, x1, y1, x2, y2, &dNaj, &dx2, &dy2, &dx1, &dy1);
  printf("(%d, %d)-(%d, %d)\n", dx1, dy1, dx2, dy2);
}
```

4. Pranje denarja

Nalogo lahko rešujemo z dinamičnim programiranjem. Označimo vhodno zaporedje transakcij z $a_1, \ldots, a_n$ in naj bo $f(k)$ največje število transakcij, ki jih lahko izbrišemo, če se omejimo na zaporedje $a_1, \ldots, a_k$. Naloga torej na koncu sprašuje po $f(n)$. Možne scenarije brisanja znotraj zaporedja $a_1, \ldots, a_k$ lahko ločimo na dve skupini: tiste, pri katerih pobrišemo a_k, in tiste, kjer a_k ne pobrišemo. Za vsako od

teh dveh možnosti izračunajmo najboljšo rešitev in nato od njiju obdržimo tisto, ki pobriše največ transakcij.

Če se odločimo, da a_k ne pobrišemo, nam ostane le še, da pobrišemo čim več transakcij iz zaporedja $a_1, \ldots, a_{k-1}$; v tem primeru torej vemo, da bo pobrisanih $f(k-1)$ transakcij.

Če pa se odločimo pobrisati a_k, moramo v resnici pobrisati blok transakcij oblike $a_{i+1}, a_{i+2}, \ldots, a_k$ za nek tak i, pri katerem je vsota pobrisanih števil enaka 0. Potem nam ostane še problem, kako pobrisati čim več transakcij izmed $a_1, \ldots, a_i$, tako da bo skupno število pobrisanih transakcij enako $f(i) + (k - i)$.

Kaj storiti, če je takih možnih indeksov i več? Recimo, da poleg i obstaja še nek zgodnejši indeks j (torej $j < i$), pri katerem tudi velja $a_{j+1} + a_{j+2} + \ldots + a_k = 0$ (torej bi lahko brisali vse od $j + 1$ do k in ne le od $i + 1$ do k). Toda to vsoto lahko razbijemo na $(a_{j+1} + \ldots + a_i) + (a_{i+1} + \ldots + a_k)$; drugi del je po predpostavki enak 0, torej mora biti prvi del tudi. Zato lahko enak učinek, kot če bi brisali vse od $j+1$ do k, dosežemo tudi tako, da zdajle brišemo le od $i+1$ do k, nato pa bomo pri reševanju problema $f(i)$ tako ali tako razmišljali o možnosti, da brišemo transakcije na indeksih od $j + 1$ do i.

Tako torej vidimo, da ko razmišljamo o brisanju transakcije a_k, je dovolj, če za začetni indeks brisanja vzamemo zadnji (najbolj desni) tak i, ki je $\leq k$ in pri katerem je vsota $a_{i+1} + \ldots + a_k$ enaka 0. Kako bi učinkovito poiskali tak i? Pomagamo si lahko z delnimi vsotami: naj bo $s_i = a_1 + \ldots + a_i$. Potem je $a_{i+1} + \ldots + a_k = s_k - s_i$, torej iz pogoja $a_{i+1} + \ldots + a_k = 0$ dobimo pogoj $s_k - s_i = 0$ oz. $s_k = s_i$. Poiskati moramo torej zadnji indeks (pred k), pri katerem je delna vsota enaka kot pri k. To lahko zelo učinkovito počnemo tako, da sproti dodajamo delne vsote v razpršeno tabelo, kjer kot ključ uporabimo vsoto samo, kot spremljevalni podatek k njemu pa zadnji dosedanji indeks, pri katerem je ta vsota nastala.

Zapišimo tako dobljeno rešitev s psevdokodo:

```
1   f[0] := 0; s := 0; H := prazna razpršena tabela;
    dodaj v H ključ 0 s pripadajočo vrednostjo 0;
2   for k := 1 to n:
       (* Ena možnost je, da a_k sploh ne pobrišemo. *)
3      f[k] := f[k − 1];
       (* Druga možnost je, da pobrišemo nek blok transakcij,
          ki se konča pri a_k. *)
4    s := s + a_k; (* Zdaj je s = a_1 + . . . + a_k. *)
5    if se s pojavlja kot ključ v H:
6       naj bo i pripadajoča vrednost pri tem ključu v H;
        (* i je torej zadnji indeks doslej, pri katerem je s_i = s_k.
           Zato smemo pobrisati blok od a_{i+1} do a_k — to je k − i transakcij;
           pred tem pa nam ostanejo transakcije a_1, . . . , a_i, za katere vemo,
           da lahko od tam pobrišemo največ f[i] transakcij. *)
7          c := f[i] + (k − i);
          (* Uporabimo to rešitev, če je boljša od prve
             (tiste, kjer a_k ne pobrišemo). *)
8       if c > f[k] then f[k] := c;
9    dodaj v H ključ s s pripadajočo vrednostjo k (oz. če ta ključ že obstaja,
```

spremeni pri njem pripadajočo vrednost na k);
return $f[n]$;

Ta postopek je zelo učinkovit; če predpostavimo, da za posamezne operacije na razpršeni tabeli porabimo po $O(1)$ časa, imamo pri vsakem k le konstantno mnogo dela in je časovna zahtevnost celotnega postopka $O(n)$.

Še ena možnost, ki bi bila tudi sprejemljivo hitra za naš namen, pa je, da vnaprej izračunamo vse delne vsote, sestavimo seznam parov (s_k, k), jih uredimo (po s_k, tiste z enako vsoto s_k pa uredimo naraščajoče po indeksu k) in se nato sprehodimo po parih v tem vrstnem redu. Če imata dva zaporedna para enako vsoto, na primer (s_i, i) in (s_k, k), pri čemer je $s_i = s_k$, potem vemo, da bomo morali pri izračunu $f[k]$ pregledati možnost, da pobrišemo blok transakcij od a_{i+1} do a_k. Ta podatek si zapomnimo v neki pomožni tabeli, recimo kot $p[k]$, od koder ga bomo lahko prebrali v glavnem delu našega postopka.

$s := 0$; $L :=$ prazen seznam;
for $k := 1$ **to** n:
  $p[k] := -1$;
  $s := s + a_k$; dodaj (s, k) v seznam L;
uredi seznam L;
$(s', i) :=$ prvi element seznama L;
for $t := 2$ **to** n:
  $(s, k) :=$ t-ti element seznama L;
  if $s = s'$ **then** $p[k] := i$;
  $s' := s$; $i := k$;

Glavnega dela našega postopka ni težko spremeniti, da uporablja tabelo p namesto razpršene tabele H. V koraku 5 moramo pogoj spremeniti v „**if** $p[k] \neq -1$“, v koraku 6 naredimo preprosto $i := p[k]$, korak 9 pa sploh pobrišemo (prav tako pobrišemo tudi inicializacijo razpršene tabele H v koraku 1).

Lepo pri tej rešitvi je, da ne potrebuje razpršene tabele, po drugi strani pa za urejanje seznama n elementov porabimo $O(n \log n)$ časa (za glavni del postopka pa še vedno $O(n)$, tako kot prej), tako da je ta rešitev počasnejša od tiste z razpršeno tabelo.

5. Simbolične povezave

Poenostavljeno pot lahko pripravimo tako, da vhodno pot razbijemo na posamezne segmente pri znakih / — tako iz poti /s_1/s_2/.../s_n nastane seznam $[s_1, s_2, \ldots, s_n]$. Po tem seznamu se sprehodimo v zanki in poenostavljajmo pot: v spremenljivki L hranimo seznam segmentov, ki smo jih že prebrali in jih (še) nismo pobrisali; segmente iz vhodnega seznama dodajamo na konec seznama L, razen ko v vhodnem seznamu naletimo na segment .., takrat pa pobrišemo zadnji segment iz seznama L. Na koncu tega postopka nam v L ostane zaporedje segmentov, ki tvorijo poenostavljeno različico prvotne poti. Zdaj se lahko v zanki sprehodimo po tem zaporedju in ugotovimo, katero vozlišče grafa predstavlja tako dobljena pot; za vsak segment zaporedja se s funkcijo GetChildNode premaknemo iz starševskega imenika v novo vozlišče, nato pa še z IsLink preverimo, če je to vozlišče simbolična povezava (in če je, se premaknemo v vozlišče, na katero ta simbolična povezava kaže).

Ker je pri tej nalogi precej dela z nizi, bomo rešitev namesto v C-ju raje zapisali v pythonu, kjer je obdelovanje nizov lažje in preprostejše. Naslednji podprogram pripravi poenostavljeno pot in vrne ID vozlišča, ki ga predstavlja ta pot:

```
def NajprejGor(pot):
    L = [] # V L bomo pripravljali poenostavljeno pot.
    for s in pot.split('/'):
        if s == "..":
            if L: L.pop() # Pri segmentu .. pobrišimo zadnji segment poenostavljene poti.
        elif s:
            L.append(s) # Druge segmente sproti dodajajmo v poenostavljeno pot.
    # Poglejmo, katero vozlišče predstavlja poenostavljena pot L.
    u = GetRootNode()
    for s in L:
        # Trenutni segment poti je s; premaknimo se iz trenutnega imenika
        # po povezavi z imenom s.
        u = GetChildNode(u, s)
        if u < 0: break # Če take povezave ni, je pot neveljavna.
        # Če smo v simbolični povezavi, ji sledimo.
        v = IsLink(u)
        if v >= 0: u = v
    return u
```

Morali bomo ugotoviti še, katero vozlišče v resnici predstavlja prvotna (nepoenostavljena) pot. To lahko naredimo z zanko po segmentih te poti, podobno tisti, ki smo jo uporabili na koncu zgornjega podprograma. Razlika je le v tem, da smo imeli tam poenostavljeno pot, ki ni več imela segmentov oblike `..` (ker smo jih med poenostavljanjem že vse pobrisali); pri prvotni poti pa moramo znati obdelovati tudi takšne segmente. Pri segmentu oblike `..` se moramo iz trenutnega vozlišča premakniti v njegovega starša (s funkcijo GetParentDir); če pa slučajno trenutno vozlišče starša sploh nima, to pomeni, da gre za korensko vozlišče in lahko ostanemo kar v njem.

```
def PoVrsti(pot):
    u = GetRootNode()
    # Pregledujmo vhodno pot po segmentih.
    for s in pot.split('/'):
        if not s: continue
        # Pri segmentu .. se premaknemo v starševski imenik trenutnega vozlišča.
        if s == "..":
            u = GetParentDir(u)
            if u < 0: u = GetRootNode()
            continue
        # Sicer se premaknimo naprej po povezavi z imenom s.
        u = GetChildNode(u, s)
        if u < 0: break # Če take povezave ni, je pot neveljavna.
        # Če smo v simbolični povezavi, ji sledimo.
        v = IsLink(u)
        if v >= 0: u = v
    return u
```

Ostane nam le še glavni podprogram, ki pokliče zgornja dva podprograma, da interpretira dano pot na oba načina (s poenostavljanjem in brez njega) ter preveri, ali pri obeh pridemo do istega vozlišča (sem štejemo tudi možnost, da se obe različici poti izkažeta za neveljavni, torej če obe gornji funkciji vrneta -1).

```
def Primerjaj(pot):
  u = NajprejGor(pot); v = PoVrsti(pot)
  return u == v
```

6. Letala

Radar nam pošilja podatke le za eno letalo naenkrat, tako da moramo sami vzdrževati neko primerno podatkovno strukturo, v kateri bomo hranili podatke o vseh letalih, ki so trenutno na radarju. Spodnji program uporablja v ta namen kar verigo zapisov, povezanih s kazalci (globalna spremenljivka seznam). Za vsako letalo imamo strukturo tipa Letalo, ki poleg oznake letala hrani zadnji dve točki, na katerih smo ga videli: zadnji znani položaj je (x2, y2) ob času t2, predzadnji pa (x1, y1) ob času t1. Čas v tem primeru merimo kar s štetjem obratov radarja (to nam sporoča funkcija Stevec); negativen čas (t1 $= -1$) uporabimo za predstavitev primerov, ko smo letalo videli šele enkrat, ne dvakrat (predpostavimo torej, da funkcija Stevec vedno vrača nenegativne vrednosti).

Ko dobimo od radarja novo meritev, se sprehodimo po seznamu in poiščemo zapis za to letalo. Če takega zapisa še ni, ga ustvarimo zdaj in ga dodajmo v seznam, sicer pa popravimo obstoječi zapis (tisto, kar je bilo doslej zadnja meritev tega letala, postane zdaj predzadnja). Spotoma lahko še pobrišemo iz seznama zapise za letala, ki jih v zadnjem končanem obratu radarja nismo zaznali, saj naloga pravi, da so taka letala bodisi pristala bodisi zapustila območje, ki ga nadzorujemo. Take primere prepoznamo takole: če je stevec trenutni obrat radarja, neko letalo pa smo nazadnje videli v obratu stevec − 2 ali še prej, potem lahko to letalo pobrišemo. V spodnji rešitvi to počne funkcija NajdiLetalo (spotoma, ko išče tudi zapis za pravkar opaženo letalo).

Če imamo za trenutno letalo zdaj že dve meritvi, lahko potem s še enim prehodom čez seznam pogledamo, ali je na poti trčenja s kakšnim drugim letalom. V spodnji rešitvi počne to funkcija PoglejPresecisca. Pri tem pridejo v poštev le tista letala, za katera tudi že imamo dve meritvi; za vsak primer bomo tudi preskočili tista, ki jih v trenutnem obratu radarja še nismo videli, saj zanje ne poznamo trenutnega položaja (če jih bomo videli kasneje v trenutnem obratu, bomo njihovo pot trčenja s trenutnim letalom tako ali tako preverili takrat). Preverjanje, ali sta dve letali na poti trčenja, je načeloma preprosto — s funkcijo Presek izračunamo presečišče premic, po katerih letita,[4] nato pa s funkcijo Razdalja pogledamo, če sta obe letali približno enako oddaljeni od tega presečišča. Paziti moramo še na primere, ko se kakšno od letal oddaljuje od presečišča — do trčenja lahko pride le, če se letali presečišču približujeta.

Glavni blok programa nima drugega dela, kot da v neskončni zanki prebira meritve radarja in jih sproti dodaja v seznam (s funkcijo NajdiLetalo) ter preverja, če so na poti trčenja s kakšnim drugim letalom (s funkcijo PoglejPresecisca).

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

typedef struct Letalo_
```

[4] Naloga ne pove nič o tem, kaj naredi funkcija Presek, če sta premici vzporedni, zato bomo predpostavili, da se letala pač ne gibljejo po vzporednih premicah.

```
{
   int oznaka, t1, t2;
   double x1, x2, y1, y2;
   struct Letalo_*nasl;
} Letalo;

Letalo *seznam = 0;

Letalo *NajdiLetalo(int oznaka, int stevec)
{
   Letalo *p = seznam, *prej = 0, *t, *r = 0;
   while (p)
   {
      if (p->oznaka == oznaka) { r = p; prej = p; p = p->nasl; }
      else if (p->t2 < stevec - 1) {
         /* Pobrišimo letala, ki jih v prejšnjem prehodu radarja nismo videli. */
         t = p->nasl; free(p); p = t;
         if (prej) prej->nasl = p; else seznam = p; }
      else { prej = p; p = p->nasl; }
   }
   /* Če iskanega letala ni v seznamu, ga dodajmo zdaj. */
   if (! r) {
      r = (Letalo *) malloc(sizeof(Letalo));
      r->t1 = -1; r->t2 = -1; r->oznaka = oznaka;
      r->nasl = seznam; seznam = r; }
   return r;
}

void PreglejPresecisca(Letalo *p)
{
   Letalo *q; double x, y, d1, dp, dq;
   for (q = seznam; q; q = q->nasl) if (q != p && q->t1 >= 0 && q->t2 == p->t2)
   {
      Presek(p->x1, p->y1, p->x2, p->y2, q->x1, q->y1, q->x2, q->y2, &x, &y);
      d1 = Razdalja(p->x1, p->y1, x, y); dp = Razdalja(p->x2, p->y2, x, y);
      if (dp > d1) continue; /* Se letalo p oddaljuje od presečišča? */
      d1 = Razdalja(q->x1, q->y1, x, y); dq = Razdalja(q->x2, q->y2, x, y);
      if (dq > d1) continue; /* Se letalo q oddaljuje od presečišča? */
      if (abs(dp - dq) < Razmik) /* Ali sta obe približno enako daleč od presečišča? */
         printf("Letali %d in %d sta na poti trčenja.\n", p->oznaka, q->oznaka);
   }
}

int main()
{
   double x, y; int oznaka, stevec; Letalo *p;
   while (true)
   {
      Radar(&x, &y, &oznaka);
      stevec = Stevec();
      p = NajdiLetalo(oznaka, stevec);
      p->t1 = p->t2; p->x1 = p->x2; p->y1 = p->y2;
      p->t2 = stevec; p->x2 = x; p->y2 = y;
      if (p->t1 >= 0) PreglejPresecisca(p);
   }
}
```

7. Oklepajski izrazi

Nalogo lahko rešimo z dinamičnim programiranjem. Naj bo $s = s_1 s_2 \ldots s_n$ naš vhodni niz in naj bo $f(i, j)$ najmanjše potrebno število znakov, ki bi jih bilo treba vriniti v $s_i s_{i+1} \ldots s_{j-1} s_j$, da bi iz njega nastal pravilno gnezden oklepajski izraz. (V nadaljevanju ne bomo posebej ponavljali besed „pravilno gnezden“; vedno, ko bomo govorili o oklepajskih izrazih, bomo s tem mislili le na pravilno gnezdene oklepajske izraze.)

Vsak oklepajski izraz nastane na enega od naslednjih dveh načinov: lahko vzamemo nek krajši oklepajski izraz in ga ovijemo v še en par oklepajev, lahko pa vzamemo dva krajša oklepajska izraza in ju staknemo skupaj. Z drugimi besedami je torej vsak oklepajski izraz oblike `(`x`)` (namesto `( )` je lahko tudi kakšen drug tip oklepajev, važno je le, da sta oklepaj in zaklepaj istega tipa), ali pa oblike xy, pri čemer sta x in y tudi pravilno gnezdena oklepajska izraza.

Če bi torej recimo hoteli predelati podniz $s_i \ldots s_j$ v pravilno gnezden oklepajski izraz oblike `(`x`)`, imamo naslednje možnosti:

- Če je s_i nek oklepaj, s_j pa zaklepaj iste vrste (s tem mislimo, da sta npr. oba okrogla ali oba oglata ipd.), ju lahko pustimo pri miru in moramo le še predelati vmesni podniz $s_{i+1} \ldots s_{j-1}$ v oklepajski izraz; za to pa po definiciji funkcije f vemo, da bomo morali vriniti $f(i+1, j-1)$ dodatnih znakov.

- Če je s_i nek oklepaj, lahko takoj za s_j vrinemo pripadajoči zaklepaj in se nato ukvarjamo s tem, kako predelati podniz $s_{i+1} \ldots s_j$ v oklepajski izraz — to pa bo zahtevalo vrivanje $f(i+1, j)$ dodatnih znakov.

- Podobno, če je s_j nek zaklepaj, lahko tik pred s_i vrinemo pripadajoči oklepaj in nato predelamo podniz $s_i \ldots s_{j-1}$ v oklepajski izraz — to pa bo zahtevalo vrivanje $f(i, j-1)$ dodatnih znakov.

Ostane še možnost, da je s_i zaklepaj in s_j oklepaj (ne nujno iste vrste), v tem primeru pa ni smiselno, da poskušamo $s_i \ldots s_j$ predelati v oklepajski izraz oblike `(`x`)`. To bi namreč pomenilo, da moramo pred s_i vriniti nek oklepaj, pred s_j pa nek zaklepaj, niz med njima pa mora biti tudi sam zase veljaven oklepajski izraz; toda to pomeni, da od tega vrivanja dveh znakov ni bilo nobene koristi, saj smo pristali pred enakim problemom kot prej (namreč kako predelati $s_i \ldots s_j$ v oklepajski izraz).

Doslej smo razmišljali o tem, kako predelati $s_i \ldots s_j$ v oklepajski izraz oblike `(`x`)`. Druga možnost pa je, da ga predelamo v oklepajski izraz oblike xy, torej stik dveh krajših oklepajskih izrazov. To pomeni, da si moramo izbrati nek vmesni indeks k ($i \leq k < j$) in predelati podniz $s_i \ldots s_k$ v en oklepajski izraz, podniz $s_{k+1} \ldots s_j$ pa v drug oklepajski izraz. Pri tem bomo morali vriniti skupaj $f(i, k) + f(k+1, j)$ dodatnih znakov.

Zdaj smo torej videli več možnosti, kako lahko iz $s_i \ldots s_j$ naredimo oklepajski izraz. Med vsemi temi možnostmi moramo za $f(i, j)$ vzeti tisto, ki zahteva najmanj dodatnih znakov. Poseben primer nastopi pri $i = j$, ko imamo podniz dolžine 1; takrat moramo v vsakem primeru vriniti natanko en dodaten znak, da nastane iz njega oklepajski izraz (če je s_i oklepaj, moramo za njim vriniti pripadajoč zaklepaj, če pa je s_i zaklepaj, moramo pred njim vriniti pripadajoč oklepaj). Pri $j = i - 1$ pa imamo pravzaprav opravka s praznim nizom, ki je že sam po sebi veljaven oklepajski

izraz, zato lahko takrat vrnemo $f(i, j) = 0$. Tako dobljeno rešitev lahko zapišemo kot rekurzivni podprogram:

funkcija $f(i, j)$:
1 (* *Najprej obdelajmo robne primere.* *)
  if $j < i$ **then return** 0;
  if $j = i$ **then return** 1;
2 (* *Spremenljivka* r *hrani najboljši rezultat doslej.* *)
  $r := \infty$;
3 (* *Poskusimo* $s_i \ldots s_j$ *predelati v oklepajski izraz oblike* xy. *)
  for $k := i$ **to** $j - 1$:
    $c := f(i, k) + f(k + 1, j)$; **if** $c < r$ **then** $r := c$;
4 (* *Poskusimo ga predelati še v oklepajski izraz oblike* (*x*). *)
  if s_i je oklepaj:
    if s_j je zaklepaj iste vrste kot s_i:
      $c := f(i + 1, j - 1)$; **if** $c < r$ **then** $r := c$;
    else:
      $c := 1 + f(i + 1, j)$; **if** $c < r$ **then** $r := c$;
  if s_j je zaklepaj:
    $c := 1 + f(i, j - 1)$; **if** $c < r$ **then** $r := c$;
5 (* *Vrnimo najboljši najdeni rezultat.* *)
  return r;

Rezultat, po katerem sprašuje naloga, je potem $f(1, n)$.

Gornja funkcija bi se v praksi izkazala za neučinkovito, ker bi po večkrat računala rezultate za iste pare (i, j). Tej težavi se izognemo, če vpeljemo tabelo, v kateri hranimo že izračunane vrednosti funkcije. Funkcijo bi morali pri tem dopolniti tako, da bi najprej pogledala, če ima rezultat za trenutni par (i, j) že v tabeli; če pa ga nima, naj ga izračuna in shrani v tabelo, preden ga vrne klicatelju. S tem lahko zagotovimo, da se vrednost $f(i, j)$ izračuna le enkrat za vsak možni par (i, j). (Takemu pomnjenju rezultatov pravimo včasih s tujko tudi *memoizacija*.)

Opazimo lahko tudi, da pri izračunu $f(i, j)$ potrebujemo rezultate funkcije f le za nekatere krajše podnize znotraj niza $s_i \ldots s_j$. Zato lahko funkcijo f računamo tudi čisto sistematično od krajših podnizov proti daljšim; tako bomo lahko, kadarkoli bomo potrebovali rešitev nekega podproblema, prepričani, da jo že imamo v tabeli. Tako dobimo naslednji postopek, v katerem je f dvodimenzionalna tabela:

for $i := 1$ **to** n:
  $f[i, i - 1] := 0$; $f[i, i] := 1$;
for $d := 2$ **to** n **do for** $i := 1$ **to** $n - d + 1$:
  $j := i + d - 1$;
  $r :=$ rezultat, kot ga izračunajo koraki 2–4 v funkciji $f(i, j)$ zgoraj,
    le da namesto rekurzivnih klicev $f(\cdot, \cdot)$ beremo rezultate podproblemov
    iz tabele $f[\cdot, \cdot]$;
  $f[i, j] := r$;

Rezultat, po katerem sprašuje naloga, je na koncu v $f[1, n]$. Časovna zahtevnost tega postopka je $O(n^3)$ — rešiti moramo $O(n^2)$ podproblemov (ker gresta lahko i in j od 1 od n), pri vsakem pa je $O(n)$ dela (zaradi zanke po k).

8. Generator naključnih števil

Naloga je lažja, če je $n \le k$. Lahko poskusimo z idejo, da s funkcijo Nakljucno1 dobimo naključno število od 0 do $k-1$, nato pa vrnemo njegov ostanek po deljenju z n (ki je torej eno od števil od 0 do $n-1$):

```
int Nakljucno2a() { return Nakljucno1() % n; }
```

Ta rešitev odpove v primerih, ko k ni večkratnik n-ja. V tem primeru so nekateri ostanki (od 0 do $(k \bmod n) - 1$) malo pogostejši od ostalih. Na primer: recimo, da je $n = 4$ in $k = 10$. Ostanek 0 dobimo v 3/10 primerov (če Nakljucno1 vrne 0, 4 ali 8), ostanek 1 tudi (pri 1, 5 ali 9), ostanek 2 pa le v 2/10 primerov (če Nakljucno1 vrne 2 ali 6), prav tako pa tudi ostanek 3 (pri 3 ali 7).

Tej težavi se lahko izognemo na primer tako, da poiščemo največji večkratnik n-ja, ki ne presega k; to je $k' := \lfloor k/n \rfloor \cdot n$. Če Nakljucno1 vrne vrednost od 0 do $k'-1$, jo uporabimo, sicer (če vrne vrednost od k' do $k-1$) pa jo zavržemo in poskusimo znova:

```
int Nakljucno2b()
{
   int kk = (k / n) * n, x;
   do { x = Nakljucno1(); } while (x >= kk);
   return x % n;
}
```

Kaj pa, če je n večji od k? Tedaj lahko poiščemo najmanjši tak c, pri katerem je $n \le k^c$. Če c-krat pokličemo funkcijo Nakljucno1, dobimo c števil $x_1, \ldots, x_c \in \{0, \ldots, n-1\}$, ki jih lahko zdaj uporabimo kot števke nekega velikega c-mestnega celega števila v k-iškem številskem sestavu: $x = (x_1 x_2 \ldots x_c)_k = \sum_{i=1}^{c} k^{c-i} x_i$. Tako dobljeno število x ima vrednost z območja $\{0, \ldots, k^c - 1\}$, pri čemer so vse te vrednosti enako verjetne. Zdaj lahko nalogo rešujemo enako kot zgoraj pri $n \le k$, le da namesto k uporabljamo k^c (v spodnjem podprogramu hrani k^c spremenljivka kc).

```
int Nakljucno2c()
{
   int i, x, c = 1, kc = k, kk;
   /* kc naj bo najmanjša potenca k-ja, ki je večja ali enaka n. */
   while (kc < n) { c++; kc *= k; }
   /* kk naj bo največji večkratnik n-ja, ki je manjši ali enak kc. */
   kk = (kc / n) * n;
   /* Sestavimo naključno število od 0 do kc − 1. */
   do { for (i = 0, x = 0; i < c; i++) x = x * k + Nakljucno1(); }
   /* Prevelike x zavržemo in to ponavljamo, dokler ni x < kk. */
   while (x >= kk);
   /* Zdaj je x naključno število od 0 do kk − 1. */
   return x % n;
}
```

Ta rešitev deluje tudi za $n \le k$, ne le za $n > k$ (pri $n \le k$ pač nastane $c = 1$ in se funkcija obnaša enako kot Nakljucno2b).

Kakšna je časovna zahtevnost naše rešitve (merjena s številom klicev funkcije Nakljucno1)? Oglejmo si najprej funkcijo Nakljucno2b. Tu se funkcija Nakljucno1 kliče

tolikokrat, kolikorkrat se izvede telo glavne zanke. Pri deljenju k z n označimo celi del količnika z a, ostanek pa z r; tako imamo $k = a \cdot n + r$ in $k' = a \cdot n$. Pogoj za nadaljevanje zanke, $x \geq k'$, je torej izpolnjen z verjetnostjo $p = r/k$. Verjetnost, da se zanka izvede natanko t-krat, je potem $p^{t-1}(1-p)$, kajti v prvih $t-1$ iteracijah mora biti izpolnjen pogoj za nadaljevanje (verjetnost tega je pri vsaki iteraciji p), v t-ti iteraciji pa ne (verjetnost slednjega je $1-p$). Pričakovano število izvajanj zanke je torej $\sum_{t=1}^{\infty} t \cdot p^{t-1}(1-p)$, kar se dá z nekaj premetavanja poenostaviti v $1/(1-p)$. Ta vrednost je seveda tem večja, čim večji je p (verjetnost, da je pogoj za nadaljevanje zanke izpolnjen). Kakšen je največji možni p? Spomnimo se, da je $p = r/k = r/(an + r)$. Ker smo rekli, da je $k \geq n$, je a gotovo ≥ 1; in ker je poleg tega $0 \leq r < n$, lahko zaključimo, da je $r < an$, zato je $an + r > 2r$ in $p = r/(an + r) < r/(2r) = 1/2$. Pričakovano število izvajanj naše glavne zanke, torej $1/(1-p)$, pa je zato < 2. Tako torej vidimo, da je pričakovano število izvajanj naše zanke le reda $O(1)$ ne glede na to, s kako velikimi k in n imamo opravka.[5]

Ta razmislek pride prav tudi pri funkciji Nakljucno2c, le da moramo tu namesto k uporabiti k^c; spet pa pridemo do zaključka, da je pričakovano število iteracij glavne zanke reda $O(1)$. Vendar pa zdaj pri vsaki iteraciji izvedemo $c = \lceil \log_k n \rceil$ klicev funkcije Nakljucno1, tako da je pričakovano število teh klicev zdaj reda $O(\log n)$.

Doslej smo razmišljali o povprečni (oz. pričakovani) časovni zahtevnosti; če pa nas zanima časovna zahtevnost v najslabšem možnem primeru, je naša rešitev (v obeh različicah — za $n \leq k$ in v splošni) videti precej manj privlačna: za to zahtevnost sploh ne moremo določiti nobene zgornje meje. Lahko si izberemo še tako velik t, pa se lahko zgodi, da glavna zanka naše funkcije izvede več kot t iteracij. Verjetnost tega je sicer majhna, nikoli pa ne pade na 0: kot smo videli že zgoraj, je verjetnost tega, da v nobeni od prvih t iteracij ni izpolnjen pogoj za ustavitev, enaka $(1-p)^t$.

Ali obstaja kakšna drugačna rešitev, ki bi zagotavljala, da število klicev funkcije Nakljucno1 niti v najslabšem primeru ne bo preseglo neke vnaprej znane zgornje meje? Pa recimo, da nek tak postopek res obstaja in da pri danih k in n vemo, da ta postopek ne bo izvedel več kot t klicev funkcije Nakljucno1. Mogoče je, da jih kdaj izvede tudi manj kot t; ampak ni si težko predstavljati, da lahko naš postopek dopolnimo tako, da bi sproti štel, koliko klicev funkcije Nakljucno1 je že izvedel, in nato, tik preden bi se postopek končal (npr. s stavkom **return** v C-ju in podobnih jezikih), bi lahko funkcijo Nakljucno1 poklical še nekajkrat, tako da bi število klicev doseglo točno t (rezultate teh dodatnih klicev na koncu pa bi ignoriral). Zato bomo v nadaljevanju našega razmisleka predpostavili, da naš postopek vedno izvede *natanko* t klicev funkcije Nakljucno1.

Smiselno je tudi predpostaviti, da je vse drugo v našem postopku deterministično — edini vir naključnosti so klici funkcije Nakljucno1. Ker je teh klicev t in ker ima vsak od njih k možnih izidov (vsi so enako verjetni), ima naš postopek kot celota k^t možnih potekov (in vsi so enako verjetni); vsak od teh potekov pa se konča s

[5] Kdaj se $p = r/(an+r)$ najbolj približa vrednosti $1/2$? Da bo p čim večji, mora biti imenovalec čim manjši, torej je za a smiselno vzeti 1 ($a = 0$ ne smemo vzeti, ker potem k ne bi bil večji od n). Zdaj imamo $p = r/(n + r) = 1/(n/r + 1)$; da bo imenovalec čim manjši, mora biti n/r čim manjši, torej mora biti r čim večji; največji možni ostanek r pa je $r = n - 1$; skupaj z $a = 1$ nam to dá $k = 2n - 1$. Zdaj imamo $p = r/k = (n-1)/(2n-1) = 1/2 - 1/(4n-2)$, kar je torej vedno (pri vsakem n) manjše od $1/2$, vendar se lahko vrednosti $1/2$ poljubno približa, če vzamemo dovolj velik n.

tem, da postopek vrne neko vrednost od 0 do $n-1$ (ki je pri tem poteku enolična). Če naj bodo torej vse te vrednosti enako verjetne (kot zahteva naša naloga), mora vsaka od njih nastati pri enakem številu potekov, to je pri k^t/n potekih. To je torej mogoče le, če je k^t večkratnik števila n.

V številu k^t so seveda prisotni isti prafaktorji kot v k (le da s t-krat tolikšno stopnjo). Če torej n vsebuje kakšen tak prafaktor, ki ga k ne, lahko takoj zaključimo, da k^t pri nobenem t ne bo večkratnik n-ja, tako da naše naloge ni mogoče rešiti s postopkom, ki bi vnaprej zagotavljal neko zgornjo mejo za število klicev funkcije Nakljucno1. Če pa so vsi prafaktorji n-ja prisotni tudi v k (četudi mogoče z nižjo stopnjo), bo pri nekem dovolj velikem t vsak od teh prafaktorjev dobil v k^t vsaj tolikšno stopnjo, kot jo ima v n, in od tistega t naprej bodo k^t večkratniki t-ja, tedaj pa je naš problem rešljiv — rešimo ga lahko po podobnem postopku kot zgoraj: s t klici Nakljucno1 si sestavimo naključno število $x \in \{0, \ldots, k^t-1\}$ in ker je k^t večkratnik n-ja, lahko kar vrnemo $x \bmod n$ in vemo, da ima ta rezultat zahtevane lastnosti (da so vsi možni rezultati od 0 do $n-1$ enako verjetni).

To, kako velik t potrebujemo, je v splošnem odvisno od k in n (in od tega, kakšne so stopnje njunih prafaktorjev), lahko pa ga navzgor omejimo takole: če so vsi n-jevi prafaktorji prisotni tudi v k, ima vsak od njih v k stopnjo vsaj 1; če je torej t vsaj tolikšen, kot je najvišja stopnja kakšnega prafaktorja v n, bodo takrat imeli vsi ti prafaktorji tudi v k^t vsaj tolikšno stopnjo kot v n. Najvišja možna stopnja kakšnega prafaktorja v n pa je $\log_2 n$, tako da lahko zaključimo, da če sploh obstaja kakšna rešitev z zagotovljeno zgornjo mejo na število klicev funkcije Nakljucno1, potem zagotovo obstaja rešitev, pri kateri je ta meja $\leq \log_2 n$.

Zapišimo našo rešitev še v C-ju. S pomočjo razmisleka iz gornjega odstavka tudi vemo, kdaj sme naša rešitev obupati (in zaključiti, da za konkretna k in n, s katerima ima opravka, naloge ni mogoče rešiti tako, da bi zagotavljali neko zgornjo mejo na število klicev Nakljucno1): namreč če 2^t preseže n, mi pa še vedno nismo našli takega k^t, ki bi bil večkratnik n-ja.

```
int Nakljucno2d()
{
  int t = 0, x = 0, kt = 1, dve_t = 1;
  while (dve_t <= n && kt % n != 0)
  {
    /* Tu je dve_t = 2^t, kt = k^t in x je naključno število od 0 do kt − 1. */
    t++; dve_t *= 2; kt *= k;
    x = x * k + Nakljucno1();
  }
  /* Če na tem mestu še vedno nismo našli takega k^t, ki bi bil večkratnik n-ja,
     to pomeni, da je zdaj 2^t že > n in noben prafaktor v n nima
     stopnje t ali več; tedaj je stvar brezupna. */
  if (kt % n == 0) return x % n;
  else return −1; /* Napaka: n ima nek prafaktor, ki ga k nima. */
}
```

9. Naključni vzorec

(a) Naj bo n velikost zahtevanega vzorca, torej število dokumentov, ki jih moramo na koncu vrniti. (Pri podatkih iz besedila naloge je $n = 100$, vendar lahko o rešitvi razmišljamo čisto dobro tudi za splošen n.)

Naš postopek naj sprva shranjuje vse prejete dokumente, dokler se jih ne nabere n. V nadaljevanju pa naj deluje takole: recimo, da smo že prebrali $k-1$ dokumentov (za nek $k > n$) in da imamo v pomnilniku shranjen nek naključen izbor n izmed teh dokumentov. Ko nato preberemo naslednji, k-ti dokument, se moramo odločiti, ali bi ga vzeli v naš izbor ali ne; in če ga vzamemo, se moramo odločiti, katerega od starih n dokumentov bi ob tem zavrgrli iz izbora.

Intuitivno lahko razmišljamo takole: če hočemo od k dokumentov izbrati n dokumentov in to tako, da bodo vsi izbori enako verjetni, si lahko predstavljamo, da je za posamezni dokument verjetnost, da bo pristal v končnem izboru, enaka n/k. Pri k-tem dokumentu se torej z verjetnostjo n/k odločimo, da ga vključimo v izbor, z verjetnostjo $1 - n/k$ pa, da ga završemo. Če se ga odločimo obdržati, moramo zavreči enega od n starih dokumentov iz dosedanjega izbora; tega, ki ga bomo zavrgli, si izberimo tako, da ima vsak od n doslej izbranih dokumentov enake možnosti, da bomo zavrgli prav njega (za vsakega od njih je torej verjetnost, da ga bomo zavrgli, enaka $1/n$).

Zapišimo dobljeni postopek s psevdokodo:

inicializacija: $k := 0$; $X :=$ tabela n dokumentov, na začetku prazna;

podprogram OnUpload(dokument d):
1 $k := k + 1$;
2 **if** $k \leq n$ **then** $i := k - 1$
3 **else** $i := \lfloor \mathsf{Random}() \cdot k \rfloor$;
4 **if** $i < n$ **then** $X[i] := d$;

podprogram GetSample():
vrni tabelo X;

Dosedanji izbor dokumentov torej hranimo v tabeli X, ki jo naslavljamo z indeksi od 0 do $n-1$. Spomnimo se, da Random vrne naključno realno število, enakomerno porazdeljeno na $[0, 1)$; če to pomnožimo s k, je torej zmnožek enakomerno porazdeljen na intervalu $[0, k)$; in ko ga zaokrožimo navzdol na najbližje celo število (operacija $\lfloor \cdot \rfloor$), dobimo naključno celo število i iz množice $\{0, 1, \ldots, k-1\}$. Z verjetnostjo n/k je $i < n$ in ga lahko uporabimo kar kot indeks, ki pove, kam v tabelo X bi vpisali novi dokument d (vsi indeksi od 0 do $n - 1$ so v tem primeru enako verjetni, kar je točno to, kar si želimo: enega od dosedanjih dokumentov v izboru bomo tako povozili z novim, pri čemer imajo vsi dosedanji dokumenti enako verjetnost, da se jim to zgodi). Z verjetnostjo $1-n/k$ pa je $i \geq n$ in v tem primeru novega dokumenta ne sprejmemo v izbor, kar je tudi točno to, kar smo hoteli.

Prepričajmo se, da je tako dobljeni postopek res pravilen (torej da ustreza zahtevam naloge). Za začetek opozorimo na to, da čeprav je naš postopek zgoraj napisan tako, da izbor X hrani v tabeli, nam vrstni red elementov v tabeli v resnici ni pomemben — za potrebe našega razmisleka si bomo X predstavljali kot množico. Ker se X med delovanjem postopka spreminja, bomo uporabili oznako X_k za tisti izbor, ki je zapisan v X ob koncu obdelave k-tega vhodnega dokumenta.

Pri nadaljnjem razmišljanju bo koristno vpeljati nekaj dodatnih oznak. Oznaka $P(\ldots)$ naj pomeni verjetnost dogodka „…“ v oklepajih. Dokumente oštevilčimo z $d_1, d_2, \ldots, d_m$ v takem vrstnem redu, v kakršnem jih funkcija OnUpload dobiva od uporabnikov (m je torej skupno število vseh dokumentov; kot pravi naloga, tega

števila vnaprej ne poznamo). Množico prvih k dokumentov označimo z $D_k := \{d_1, \ldots, d_k\}$. Izbor n dokumentov izmed prvih k dokumentov zdaj ni nič drugega kot taka podmnožica množice D_k, ki vsebuje n elementov; vse take izbore označimo s $C_{k,n} = \{Z \subseteq D_k \;:\; |Z| = n\}$. Opazimo lahko, da jih je $\binom{k}{n}$, to je $k!/(n!(k-n)!)$.

Če se hočemo prepričati, da je naš postopek pravilen, moramo pokazati, da ima na koncu izvajanja vsak od $\binom{m}{n}$ možnih izborov iz $C_{m,n}$ enako verjetnost, da bo izbran; pri vsakem je torej ta verjetnost enaka $1/\binom{m}{n}$. Z drugimi besedami: pokazati moramo, da za vsak $Z \in C_{m,n}$ velja $P(X_m = Z) = 1/\binom{m}{n}$.

O tem se bomo prepričali z indukcijo: pokazali bomo, da nekaj podobnega ne velja le pri m, ampak že prej; natančneje, za vsak $k \geq n$ in za vsak $Z \in C_{k,n}$ velja $P(X_k = Z) = 1/\binom{k}{n}$. (Z besedami: na koncu obdelave k-tega dokumenta se v tabeli X nahaja eden od izborov iz $C_{k,n}$ in to tako, da ima vsak od njih enako verjetnost, da je za X izbran prav on — ta verjetnost pa je $1/\binom{k}{n}$.)

Pri $k = n$ se o tem ni težko prepričati: tu moramo dokazati, da za vsak $Z \in C_{n,n}$ velja $P(X_n = Z) = 1/\binom{n}{n} = 1$. V resnici je tu možen en sam Z, namreč $Z = D_n$ (od dosedanjih n dokumentov moramo izbrati vseh n); in naš postopek tudi res sestavi prav ta izbor, saj do vključno $k = n$ le sproti dodaja vse prebrane dokumente v izbor X. Torej zagotovo velja $X_n = D_n$, tako da je verjetnost $P(X_n = Z)$ res enaka 1 (za $Z = D_n$, drugega Z pa v $C_{n,n}$ tako ali tako ni).

Recimo zdaj, da naša trditev velja pri $k-1$ (za nek $k > n$), in razmislimo, da velja tudi pri k. Vzemimo poljuben $Z \in C_{k,n}$ in poskusimo izračunati verjetnost $P(X_k = Z)$. Ločimo dve možnosti: ali Z vsebuje d_k ali ne.

(i) Recimo, da Z ne vsebuje d_k. Do $X_k = Z$ lahko tedaj pride le tako, da naš postopek v vrstici 4 ne vpiše dokumenta d_k v tabelo X, to pa se zgodi le, če je v vrstici 3 izbral tak i, ki je $\geq n$ (natančneje: ki je z območja od n do $k-1$ namesto od 0 do $n-1$); verjetnost tega pa je $(k-n)/k$. V tem primeru naš postopek pri obdelavi dokumenta d_k sploh ne bo spreminjal tabele X, torej bo takrat $X_k = X_{k-1}$. Tako torej imamo $P(X_k = Z) = (k-n)/k \cdot P(X_{k-1} = Z)$.

Ker Z ne vsebuje d_k, to pomeni, da vseh n dokumentov v Z prihaja iz D_{k-1}, ne le iz D_k, torej je $Z \in C_{k-1,n}$. Zato pa lahko uporabimo induktivno predpostavko in zaključimo, da je $P(X_{k-1} = Z) = 1/\binom{k-1}{n}$. Skupaj s formulo s konca prejšnjega odstavka imamo torej $P(X_k = Z) = (k-n)/k \cdot 1/\binom{k-1}{n} = \ldots = 1/\binom{k}{n}$.

(ii) Druga možnost pa je, da Z vsebuje d_k. Označimo $Z' = Z - \{d_k\}$; to je izbor $n-1$ elementov izmed D_{k-1}, torej je $Z' \in C_{k-1,n-1}$.

Do $X_k = Z$ lahko pride le tako, da je pred obdelavo k-tega dokumenta tabela X vsebovala izbor, ki se od Z razlikuje le v enem elementu — namesto d_k ima nek $y \in D_{k-1} - Z'$ — in da je med obdelavo k-tega dokumenta naš postopek v vrstici 4 ta element povozil z novim dokumentom d_k. Izbor pred to spremembo označimo z $Z_y = Z' \cup \{y\}$. Verjetnost, da vrstica 3 izbere točno tisti i, na katerem se v tabeli X takrat nahaja element y, da bo potem vrstica 4 ravno njega zamenjala z d_k, je seveda $1/k$. Tako imamo $P(X_k = Z) = \sum_y P(X_{k-1} = Z_y) \cdot 1/k = \sum_y 1/\binom{k-1}{n} \cdot 1/k$, pri čemer mora iti vsota po vseh $y \in D_{k-1} - Z'$; možnih y je torej $|D_{k-1} - Z'| = k-n$, vsi členi vsote pa so enaki, tako da dobimo $P(X_k = Z) = (k-n) \cdot 1/\binom{k-1}{n} \cdot 1/k = \ldots = 1/\binom{k}{n}$.

V obeh primerih, (i) in (ii), torej vidimo, da je $P(X_k = Z) = 1/\binom{k}{n}$, prav to pa

smo tudi želeli dokazati. □

Oglejmo si še eno, malo drugačno rešitev naše naloge. Predstavljajmo si, da vsakemu dokumentu d_k pripišemo neko *oceno* U_k, ki naj bo naključno število z intervala $[0, 1)$. Na koncu bi dokumente lahko uredili po njihovih ocenah in vrnili prvih n dokumentov v tem vrstnem redu (tiste z najvišjimi ocenami). Ker so ocene v resnici naključne, bo tudi tako dobljen nabor n dokumentov naključen, prav to pa zahteva naša naloga.

Težava je, da v pomnilniku ne moremo hraniti vseh dokumentov naenkrat, saj jih je preveč; torej si ne moremo privoščiti, da bi jih najprej le shranjevali, uredili (in vrnili prvih n) pa šele na koncu. Bolje je, da hranimo vedno le n dokumentov z najvišjimi ocenami izmed vseh doslej prebranih dokumentov. Ko dobimo nov dokument, mu pripišemo oceno in pogledamo, ali je nižja od n-te najnižje dosedanje ocene; če je, si novi dokument zapomnimo namesto tistega, ki je imel doslej n-to najnižjo oceno. Zapišimo ta postopek s psevdokodo:

inicializacija: $k := 0$; $X :=$ izbor največ n parov $\langle$*dokument*, *ocena*$\rangle$, sprva prazen;

podprogram OnUpload(dokument d):
1 $k := k + 1$; $U :=$ Random();
2 **if** $k \leq n$ **then** dodaj $\langle d, U\rangle$ v X
3 **else**:
4 naj bo $\langle d', U'\rangle$ par z najmanjšo oceno v X;
5 **if** $U > U'$ **then**
6 pobriši $\langle d', U'\rangle$ iz X in vanj dodaj $\langle d, U\rangle$;

podprogram GetSample():
vrni množico vseh dokumentov iz X;

Vprašanje je še, kako v praksi predstaviti X. Preprosta rešitev je kar tabela, v kateri so pari $\langle d_i, U_i\rangle$ zloženi brez kakšnega posebnega vrstnega reda; žal to pomeni, da imamo pri vsakem dokumentu v vrstici 4 kar $O(n)$ dela, da najdemo tistega z najnižjo oceno. Možna izboljšava je, da si zapomnimo, kje v tabeli je dokument z najnižjo oceno; ta podatek moramo popraviti le po vsaki spremembi v tabeli v vrstici 6 (torej nam zdaj ta vzame $O(n)$ dela, vrstica 4 pa le $O(1)$). Podoben učinek lahko dosežemo tudi, če dokumente v X hranimo urejene po oceni; dokument z najvišjo oceno je tako vedno pri roki na koncu seznama, tako da vrstica 4 vzame le $O(1)$ časa, dodajanje novega elementa v seznam (v vrstici 6) pa zdaj traja $O(n)$ časa; seznam X je lahko shranjen v tabeli ali pa kot veriga, povezana s kazalci (*linked list*). Še bolje pa je, če X hranimo v kopici (*heap*); to je drevesasta struktura, v kateri bo dokument z najnižjo oceno vedno v korenu drevesa, tako da vrstica 4 vzame $O(1)$ časa, popravljanje kopice v vrstici 6 pa le $O(\log n)$ časa, kar je precej bolje od dosedanjih rešitev.

Da ocenimo, koliko so te izboljšave koristne, bi bilo torej dobro vedeti, kako pogosto se izvede vrstica 6 oz. z drugimi besedami, kako pogosto pride do sprememb v izboru; ali še drugače, kako pogosto je izpolnjen pogoj v vrstici 5. Predstavljamo si lahko, da če smo doslej prebrali že k dokumentov, je verjetnost, da ocena pravkar prebranega dokumena pride med najvišjih n, enaka n/k.[6] Pričakovano število izva-

[6] Če bi hoteli to dokazati bolj formalno, bi lahko razmišljali takole. Označimo l-to največjo

janj vrstice 6 je torej $\sum_{k=n+1}^{m} \frac{n}{k} = n \sum_{k=n+1}^{m} \frac{1}{k} = n(H_m - H_n) \approx n \ln \frac{m}{n}$; pri tem so H_k harmonična števila, $H_k = \sum_{i=1}^{k} 1/k$, za katera velja $H_k \approx \ln k$.[7] Pri rešitvi s kopico je torej časovna zahtevnost našega postopka skupaj $O(m + \log n \cdot \log \frac{m}{n})$.

Prepričajmo se še, da je naša rešitev pravilna, torej da imajo res vsi možni izbori n dokumentov enako verjetnost, da bodo izbrani. Podobno kot pri prvi rešitvi označimo z X_k izbor n dokumentov, ki ga naš postopek hrani po obdelavi k-tega dokumenta. Dokazali bi torej radi, da (pri $k \geq n$) za vsak $Z \in C_{k,n}$ velja $P(X_k = Z) = 1/\binom{k}{n}$.

Recimo torej, da smo prebrali k dokumentov; naj bo Z poljuben izbor iz $C_{k,n}$. Naš postopek bo v X_k vrnil tistih n dokumentov, ki imajo najvišje ocene doslej; ta izbor bo torej enak Z natanko tedaj, če imajo vsi dokumenti iz Z višje ocene kot vsi drugi dokumenti (iz $D_k - Z$). Označimo torej z $V = \min\{U_i : d_i \in Z\}$ najnižjo oceno po vseh dokumentih iz Z, z $W = \max\{U_i : d_i \in D_k - Z\}$ pa najvišjo oceno po vseh ostalih dokumentih. Pravkar smo videli, da je $P(X_k = Z) = P(V > W)$.

Razmislimo o porazdelitvi W-ja: za vsak dokument d_i je verjetnost, da je njegova ocena manjša od w, enaka $P(U_i < w) = w$, saj so ocene enakomerno porazdeljene po intervalu $[0, 1)$. V množici $D_k - Z$ je $k - n$ dokumentov; ker so njihove ocene neodvisne med sabo, je verjetnost, da so vse manjše od w, enaka w^{k-n}. Torej je $P(W < w) = w^{k-n}$. Gostota porazdelitve W-ja je odvod tega, torej $p(w) = (k - n)w^{k-n-1}$.

Podobno je tudi pri V; za vsak dokument d_i imamo $P(U_i > w) = 1 - w$; izbor Z vsebuje n dokumentov in njihove ocene so neodvisne med sabo, zato je verjetnost, da so vse večje od w, enaka $(1 - w)^n$. Tako vidimo $P(V > w) = (1 - w)^n$.

Zdaj lahko izrazimo $P(V > W)$ kot $\int_0^1 P(V > w)p(w)dw$ in v to vstavimo pravkar dobljene formule za $P(V > w)$ in $p(w)$; z nekaj premetavanja se nam izraz poenostavi v $1/\binom{k}{n}$, prav to pa smo želeli dokazati.[8] □

(*b*) Pogoj iz podnaloge (*a*) lahko, kot smo videli že večkrat na zadnjih nekaj straneh, zapišemo kot $P(X_m = Z) = 1/\binom{m}{n}$ za vsak $Z \in C_{m,n}$. Novi pogoj, da mora imeti vsak dokument enako verjetnost, da pride v končni izbor, pa je torej $P(d \in X_m) = n/m$ za vsak $d \in D_m$. (Da mora biti ta verjetnost n/m, si ni težko predstavljati, saj moramo od m dokumentov na koncu izbrati n dokumentov.)

---

med vrednostmi $U_1, \ldots, U_k$ z $V_{k,l}$. Potem je $P(V_{k,l} < v) = P$(kvečjemu $l-1$ od števil $U_1, \ldots, U_k$ je večjih od v, ostala pa so manjša) $= \sum_{i=0}^{l-1} P$(natanko i od števil $U_1, \ldots, U_k$ je večjih od v, ostala pa so manjša) $= \sum_{i=0}^{l-1} \binom{k}{i}(1-v)^i v^{k-i}$. Pogoj v vrstici 5 je izpolnjen, če je $V_{k-1,n} < U_k$; spomnimo se, da je U_k, ocena k-tega dokumenta, porazdeljena enakomerno na intervalu $[0, 1)$; njena verjetnostna gostota je torej $p(u) = 1$ za $0 \leq u < 1$ in $p(u) = 0$ drugod. Tako imamo $P(V_{k-1,n} < U_k) = \int_0^1 P(V_{k-1,n} < u)p(u)du$; z nekaj telovadbe se izkaže, da je to ravno enako n/k. (Vstavimo malo prej dobljeni izraz za $P(V_{k,l} < v)$, zamenjajmo vrstni red seštevanja in integriranja, nato pa pri poenostavljanju izraza upoštevajmo dejstvo, da je $\int_0^1 u^a(1-u)^b du = a!b!/(a+b+1)!$; za več o slednjem gl. npr. Wikipedijo *s. v.* Beta function.)

[7] Mimogrede, ni se težko prepričati, da do prav enakega števila sprememb v izboru X pride tudi pri naši prvotni rešitvi. Tam do spremembe pride, če je v vrstici 4 izpolnjen pogoj $i < n$, verjetnost tega pa je n/k, ker smo v vrstici 3 izbrali i z območja od 0 do $k-1$. Tako je pričakovano skupno število sprememb enako $\sum_{k=n+1}^{m} n/k$, kar je enako kot tukaj pri naši drugi rešitvi.

[8] Tudi pri tej izpeljavi pride prav funkcija beta, ki smo jo videli v eni od zgornjih opomb.

Izkaže se, da pogoja nista enakovredna; vsaka rešitev, ki ustreza prvemu pogoju, ustreza tudi drugemu, obratno pa ne drži. Prepričajmo se za začetek o tem, da drugi pogoj res sledi iz prvega. Mislimo si poljubno rešitev, ki vrača vse izbore $Z \in C_{m,n}$ z enako verjetnostjo $1/\binom{m}{n}$. Vzemimo poljuben dokument $d \in D_m$ in se vprašajmo, kakšna je verjetnost, da ta dokument pride v izbor X_m. Koliko izborov $Z \in C_{m,n}$ sploh vsebuje dokument d? Da sestavimo takšen Z, moramo poleg d-ja izbrati še $n-1$ dokumentov izmed ostalih $m-1$ dokumentov (torej iz $D_m - \{d\}$), kar lahko naredimo na $\binom{m-1}{n-1}$ načinov. Iskana verjetnost je torej $P(d \in X_m) = \sum_{Z:d\in Z} P(X_m = Z) = \sum_{Z:d\in Z} 1/\binom{m}{n} = \binom{m-1}{n-1}/\binom{m}{n} = n/m$, kar smo tudi želeli dokazati.

Ni pa težko sestaviti postopka, ki vrača izbore, v katerih imajo vsi dokumenti enako verjetnost, da so izbrani, niso pa vsi izbori enako verjetni. V nadaljevanju si bomo ogledali en možen primer takšnega postopka. Naj bo Y_k izbor, ki vsebuje d_k in še prejšnjih $n-1$ dokumentov: $Y_k = \{d_{k-n+1}, \ldots, d_k\}$; če pa je $k < n$, vzemimo nekaj dokumentov še s konca zaporedja: $Y_k = \{d_1, \ldots, d_k, d_{m-(n-k)+1}, \ldots, d_m\}$. Mislimo si zdaj postopek, ki za vsak k od 1 do m vrne izbor Y_k z verjetnostjo $1/m$, drugačnih izborov (torej takih, ki ne vsebujejo n zaporednih dokumentov) pa sploh nikoli ne vrne. Očitno je, da ta postopek ne ustreza našemu prvotnemu pogoju iz naloge (a), hitro pa se vidi, da ustreza pogoju iz (b): od m možnih izborov, ki jih naš novi postopek utegne vrniti (torej $Y_1, \ldots, Y_m$), se dokument d_k pojavlja v n izborih (to so $Y_k, Y_{k+1}, \ldots, Y_{k+n-1}$; če kakšen od teh indeksov preseže m, pa moramo v mislih od njega odšteti m). Ker ima vsak od teh izborov verjetnost $1/m$, da bo izbran, je torej verjetnost, da pride d_k v končni izbor, enaka n/m, ravno to pa zahteva novi pogoj iz (b).

Pri implementaciji tega postopka lahko opazimo zanimivo povezavo z rešitvijo podnaloge (a): če tam vzamemo $n = 1$, bomo pravzaprav izbrali en dokument d_k, pri čemer imajo vsi dokumenti enako verjetnost, da so izbrani. Vse, kar moramo narediti, da dobimo implementacijo našega novega postopka za podnalogo (b), je, da poleg d_k vrnemo še prejšnjih $n-1$ elementov (ki skupaj z d_k tvorijo izbor Y_k). Za osnovo lahko vzamemo katerokoli od obeh rešitev, ki smo ju videli pri (a); če na primer začnemo z drugo, je ogrodje našega novega postopka takšno:

inicializacija: $k := 1$; $U^\star := -1$;

podprogram OnUpload(dokument d):
    $k := k + 1$; $U :=$ Random();
    if $U > U^\star$ **then**
        $k^\star := k$; $U^\star := U$;

podprogram GetSample():
    vrni razpored $Y_{k^\star}$;

Stvari se malo zapletejo zaradi dejstva, da v pomnilniku ne moremo hraniti vseh dokumentov hkrati. V resnici je dovolj, če hranimo vedno le trenutni dokument in še prejšnjih $n-1$; spodnji postopek ima v ta namen tabelo T (z indeksi od 0 do $n-1$), ki jo uporabljamo kot krožni medpomnilnik (*ring buffer*): v njej se dokument d_k vedno hrani na indeksu $k \bmod n$. To pomeni, da se d_k hrani v isti celici tabele kot d_{k-n}, tako da s tem, ko vpišemo d_k v tabelo, obenem tudi zavržemo iz nje dokument d_{k-n}, ki ga ne potrebujemo več. Po takem popravku torej tabela T

vsebuje pravzaprav izbor Y_k; in če se odločimo trenutni k uporabiti kot $k^\star$, si tudi trenutni Y_k zapomnimo v neki tabeli (v spodnji psevdokodi je to X), da jo bomo lahko na koncu vrnili.

Poseben primer nastopi pri $k^\star < n$; tedaj vsebuje $Y_{k^\star}$ nekaj dokumentov s konca zaporedja, ki jih na začetku zaporedja še ne poznamo. Zato jih mora GetSample (ki se kliče na koncu zaporedja) v tem primeru šele dodati v izbor (na srečo so takrat vsi pri roki v tabeli T). Podrobnejši opis naše rešitve je torej takšen:

inicializacija: $k := 1$; $U^\star := -1$; $T, X :=$ prazni tabeli za n dokumentov;

podprogram OnUpload(dokument d):
1 $k := k + 1$; $U :=$ Random(); $T[k \bmod n] := d$;
2 **if** $U > U^\star$ **then**
3 $k^\star := k$; $U^\star := U$; $X := T$;

podprogram GetSample():
if $k^\star < n$ **then**
for $i := k^\star + 1$ **to** n **do** $X[i \bmod n] := T[(i + m - n) \bmod n]$;
vrni tabelo X;

Kakšna je časovna zahtevnost tega postopka? Pri vsakem dokumentu imamo $O(1)$ dela v vrstici 1, če pa pride do spremembe v izbranem indeksu ($k^\star$ in oceni $U^\star$), imamo še $O(n)$ dela v vrstici 3 (za kopiranje tabele T v X). Kot smo videli že v rešitvi podnaloge (*a*), je pričakovano število takšnih sprememb približno $n \ln(m/n)$. Skupaj je časovna zahtevnost našega postopka torej $O(m + n^2 \log \frac{m}{n})$.

10. Štetje nizov

Pri reševanju naloge predpostavimo, da je $n \geq k$, saj je drugače takoj jasno, da ni nobenega primernega niza x. Vpeljimo tudi oznake s_i oz. x_i za i-ti znak niza s oz. x; torej je $s = s_1 s_2 \ldots s_k$ in $x = x_1 x_2 \ldots x_n$. Podniz oblike $x_i x_{i+1} \ldots x_{j-1} x_j$ pa zapišimo krajše kot $x_{i..j}$.

Oglejmo si zdaj najprej tiste različice naloge, pri katerih zahtevamo, da je s strnjen podniz niza x.

(1) Če ima s same različne znake in zahtevamo, da ima tudi x same različne znake, je naloga čisto kombinatorična. V nizu x dolžine n si lahko na $n-k+1$ načinov izberemo, kje se v njem pojavi niz s. Ta pojavitev pokrije k mest niza x, za ostalih $n-k$ mest niza x pa si lahko izberemo poljubne take znake abecede, ki ne nastopajo v nizu s; takih znakov je $d-k$, torej si jih lahko izberemo na $\binom{d-k}{n-k}$ načinov, nato pa si še na $(n-k)!$ načinov izberemo njihov vrstni red v nizu x. Tako torej vidimo, da je število primernih nizov x enako $(n-k+1)\binom{d-k}{n-k}(n-k)! = (n-k+1)(d-k)!/(d-n)!$.

(2) Če s nima samih različnih znakov in zahtevamo, da x ima same različne znake, potem takih x sploh ni in je naloga trivialna.

(3) Recimo zdaj, da ima s same različne znake, za x pa je dovoljeno tudi, da ima več enakih znakov. Število nizov x dolžine n, ki vsebujejo s kot podniz, označimo s $f(n)$. Take nize lahko v mislih razdelimo na dve ločeni skupini: (*i*) tiste, ki vsebujejo s kot podniz že znotraj prvih $n-1$ znakov, torej v $x_{1..n-1}$; (*ii*) in tiste, ki vsebujejo s kot podniz šele prav na koncu (torej imajo $x_{n-k+1..n} = s$), prej (znotraj $x_{1..n-1}$) pa nikjer.

Pri (i) vidimo, da lahko vse takšne x dobimo tako, da vsem možnim nizom dolžine $n-1$, ki vsebujejo s kot podniz (takih nizov je $f(n-1)$), na vse možne načine pritaknemo na koncu še en znak (to lahko naredimo na d načinov, ker imamo abecedo z d znaki). Skupini (i) torej pripada $f(n-1) \cdot d$ nizov.

Pri (ii) je zadnjih k znakov niza x enolično določenih ($x_{n-k+1..n} = s_{1..k}$), prvih $n-k$ znakov niza x pa je načeloma lahko poljubnih. Ker imamo v naši abecedi d možnih znakov, si torej lahko niz $x_{1..n-k}$ izberemo na d^{n-k} načinov; vendar pa moramo od tega zdaj odšteti nize, pri katerih nastopi s kot podniz že nekje znotraj niza $x_{1..n-1}$. Ker se $x_{1..n}$ konča na $s_{1..k}$, se niz $x_{1..n-1}$ konča na $s_{1..k-1}$, torej je $x_{1..n-1}$ oblike $x_{1..n-k}s_{1..k-1}$. Recimo, da se v njem nekje pojavi celoten s kot podniz. Ali se lahko to zgodi tako, da zadnji znak te pojavitve s-ja nastopi nekje znotraj območja $s_{1..k-1}$ na koncu našega niza $x_{1..n-1}$? Gotovo ne, kajti to bi pomenilo, da je eden od znakov $s_{1..k-1}$ enak znaku s_k, naša naloga pa zagotavlja, da ima s same različne znake. Tako torej vidimo, da če se s pojavlja kot podniz v $x_{1..n-1}$, se mora pojaviti kot podniz že znotraj $x_{1..n-k}$, takih nizov $x_{1..n-k}$ pa je $f(n-k)$. Skupini (ii) torej pripada $d^{n-k} - f(n-k)$ nizov x.

Če oboje združimo, dobimo rekurzivno zvezo $f(n) = d^{n-k} - f(n-k) + f(n-1) \cdot d$. Poseben primer nastopi pri $n < k$, ko moramo vzeti $f(n) = 0$, saj je takrat niz x prekratek, da bi lahko vseboval s kot podniz. Vrednosti funkcije f je koristno računati po naraščajočih n in jih sproti shranjevati v tabelo, odkoder jih lahko preberemo, ko jih ponovno potrebujemo pri kasnejših izračunih.

(4) Ostane še možnost, da s nima (nujno) samih različnih znakov, pa tudi od nizov x ne zahtevamo, da bi jih imeli. Razmišljamo lahko podobno kot pri (3), vendar se zdaj razmislek pri skupini (ii) zaplete: ko hočemo od nizov oblike $x_{1..n-k}s_{1..k-1}$ odšteti tiste, ki vsebujejo s kot podniz, se načeloma lahko zgodi, da taka pojavitev s-ja deloma leži tudi v zadnjih $k-1$ znakih, ne le v prvih $n-k$. Da jih bomo znali odšteti, moramo torej ugotoviti, koliko je nizov dolžine $n-1$, ki se končajo na $s_{1..k-1}$ in ki nekje vsebujejo s kot podniz.

Če tako razmišljamo naprej, sčasoma opazimo, da bomo pravzaprav morali reševati podprobleme takšne oblike: koliko je nizov x dolžine n, ki se končajo na $s_{1..t}$ in ki nekje vsebujejo celoten s kot podniz? Temu številu recimo $f(n, t)$; tisto, po čemer sprašuje naša naloga, je potem pravzaprav $f(n, 0)$, kajti pri $t = 0$ nimamo nobene omejitve več glede tega, na kaj se mora končati niz x.

Tudi nize iz $f(n, t)$ lahko razdelimo na dve skupini, enako kot zgoraj. (i) Oglejmo si najprej tiste x, pri katerih se s pojavlja kot podniz že znotraj $x_{1..n-1}$. Če je $t > 0$, je zadnji znak niza x enolično določen ($x_n = s_t$), za $x_{1..n-1}$ pa lahko vzamemo poljuben niz dolžine $n-1$, ki vsebuje s in se konča na $s_{1..k-1}$; takih je po definiciji funkcije f preprosto $f(n-1, t-1)$. Če pa je $t = 0$, smemo x_n izbrati poljubno (zanj je torej d možnosti), za $x_{1..n-1}$ pa lahko vzamemo poljuben niz dolžine $n-1$, ki vsebuje s, takih pa je $f(n-1, 0)$.

(ii) Ostanejo še tisti x, ki so dolgi n znakov, se končajo na $s_{1..t}$ in pri katerih se s pojavlja kot podniz šele prav na koncu, kot $x_{n-k+1..n}$, prej pa ne. Tak x se torej konča na $s_{1..k}$ in hkrati na $s_{1..t}$, kar je mogoče le, če je $s_{1..t} = s_{k-t+1..k}$; če to ne velja, takih x sploh ni. Drugače pa lahko razmišljamo takole: zadnji znak, x_n, je enolično določen ($x_n = s_k$), pred tem pa nam znaki $x_{1..n-1}$ tvorijo niz dolžine $n-1$, ki se konča na $s_{1..k-1}$ in sploh nikjer ne vsebuje s kot podniza. Nizov dolžine

$n-1$, ki se končajo na $s_{1..k-1}$, je d^{n-k} (prvih $n-k$ znakov si namreč lahko izberemo poljubno, zadnjih k pa je enolično določenih, saj se morajo ujemati s $s_{1..k-1}$); od njih pa moramo zdaj odšteti tiste, ki vsebujejo s kot podniz, takih pa je $f(n-1, k-1)$.

Če zdaj oboje združimo, dobimo:

$$f(n,t) = f_i(n,t) + f_{ii}(n,t)$$
$$f_i(n,t) = \begin{cases} f(n-1,t-1), & \text{če } t > 0 \\ d \cdot f(n-1,t) & \text{sicer} \end{cases}$$
$$f_{ii}(n,t) = \begin{cases} d^{n-k} - f(n-1,k-1), & \text{če } s_{1..t} = s_{k-t+1..k} \\ 0 & \text{sicer.} \end{cases}$$

Robni primer nastopi pri $n = k$, kjer je edini x, ki vsebuje s kot podniz, kar $x = s$ sam; ta pa je za naš namen primeren le, če se hkrati tudi konča na $s_{1..t}$; tako je torej $f(k,t) = 1$, če je $s_{1..t} = s_{k-t+1..k}$, sicer pa je $f(k,t) = 0$.

Kot vidimo iz teh formul, lahko f računamo sistematično po naraščajočih n, pri vsakem n pa po naraščajočih t. Podobno kot pri (3) si že izračunane vrednosti zapomnimo v tabeli, kjer nam bodo prišli prav pri kasnejših izračunih. Mimogrede, rešitev (4) bi lahko uporabili tudi v primerih, ko ima s same različne znake, vendar je takrat bolje uporabiti rešitev (3), ki je hitrejša in preprostejša.

Kaj pa, če naloga ne zahteva, da mora biti s strnjen podniz niza x, ampak dovoli tudi nestrnjene pojavitve s-ja kot podniza v x? (To pomeni, da smejo biti v nizu x med znaki, ki tvorijo s, vrinjeni tudi še kakšni drugi znaki; na primer, niz *aa* se pojavlja kot nestrnjen podniz v nizu *aba*.)

($1'$) Če ima s same različne znake in zahtevamo, da ima tudi x same različne znake, je razmislek zelo podoben kot pri (1), le da si to, kje v nizu x se pojavijo znaki niza s, zdaj lahko izberemo na $\binom{n}{k}$ načinov (izmed n znakov niza x si jih moramo izbrati k, ki bodo tvorili podniz s) in ne le na $n - k + 1$ načinov. Rešitev je zdaj torej $\binom{n}{k}(d-k)!/(d-n)!$.

($2'$) Če s nima samih različnih znakov in zahtevamo, da x ima same različne znake, potem takih x sploh ni, enako kot prej pri (2).

($3'$ in $4'$) Recimo zdaj, da ne zahtevamo, da ima x same različne znake. To, da x vsebuje $s_{1..k}$ kot (lahko nestrnjen) podniz, si lahko predstavljamo takole: nekje v x mora nastopiti znak s_k, recimo kot x_m; levo od njega pa mora niz $x_{1..m-1}$ vsebovati $s_{1..k-1}$ kot (lahko nestrnjen) podniz. Največ možnosti, da najdemo $s_{1..k-1}$ kot podniz v $x_{1..m-1}$, bomo imeli, če za $x_{1..m-1}$ vzamemo čim daljši kos niza x, torej če za t vzamemo indeks zadnje (najbolj desne) pojavitve znaka s_k v nizu x. Kakorkoli že, zdaj lahko naprej razmišljamo podobno; nekje v nizu $x_{1..m-1}$ mora nastopiti znak s_{k-1}, levo od njega pa se mora kot (lahko nestrnjen) podniz pojaviti niz $s_{1..k-2}$ in tako naprej.

Koristno se je torej vprašati: koliko je nizov x dolžine n, ki kot (lahko nestrnjen) podniz vsebujejo $s_{1..t}$, ne pa tudi $s_{1..t+1}$ (ali kakšnega še daljšega začetka niza s)? Temu številu recimo $f(n,t)$. Take nize lahko razdelimo na dve skupini: (i) Ena možnost je, da se $s_{1..t}$ pojavlja kot podniz že v $x_{1..n-1}$; tak $x_{1..n-1}$ si lahko izberemo na $f(n-1,t)$ načinov, za znak x_n pa si lahko izberemo karkoli razen s_{t+1} (kajti če bi vzeli $x_n = s_{t+1}$, bi niz x vseboval že podniz $s_{1..t+1}$ in ne le $s_{1..t}$). Tu torej dobimo

$f(n-1,t) \cdot (d-1)$ podnizov, razen pri $t = k$, ko znak s_{t+1} sploh ne obstaja in si smemo zato x_n izbrati čisto poljubno, tako da dobimo $f(n-1,t) \cdot d$ nizov.

(*ii*) Druga možnost pa je, da se $s_{1..t}$ ne pojavlja kot podniz v $x_{1..n-1}$. Ker se mora vendarle pojaviti vsaj v $x_{1..n}$, to pomeni, da mora biti $x_n = s_t$, v nizu $x_{1..n-1}$ pa se mora kot podniz pojaviti vsaj $s_{1..t-1}$. Tu si lahko torej $x_{1..n-1}$ izberemo na $f(n-1,t-1)$ načinov, znak x_n pa le na en način. Pri $t = 0$ ta možnost sploh odpade, saj je $s_{1..t}$ takrat prazen niz, ta pa se pojavlja kot podniz v vsakem nizu.

Če zdaj oboje združimo, dobimo:

$$f(n,t) = f_i(n,t) + f_{ii}(n,t)$$
$$f_i(n,t) = \begin{cases} f(n-1,t) \cdot d, & \text{če } t = k \\ f(n-1,t) \cdot (d-1) & \text{sicer} \end{cases}$$
$$f_{ii}(n,t) = \begin{cases} f(n-1,t-1), & \text{če t} > 0 \\ 0 & \text{sicer.} \end{cases}$$

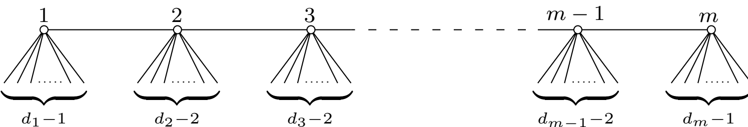
Robni primer je $f(n,t) = 0$ za $n < t$ (takrat je x prekratek, da bi vseboval $s_{1..t}$ kot podniz) in $f(n,t) = 1$ za $n = t$ (takrat je primeren le $x = s_{1..t}$). Tudi zdaj lahko računamo vrednosti funkcije f sistematično po naraščajočih n, pri vsakem n pa po naraščajočih t. Rezultat, po katerem sprašuje naloga, dobimo na koncu v $f(n,k)$.

11. Botanika

Naj bodo $d_1, \ldots, d_n$ stopnje naših vozlišč. Drevo lahko gradimo postopoma: mislimo si najprej prvo vozlišče stopnje d_1; iz njega torej izhaja d_1 povezav, ki pa zaenkrat še niso povezane z drugimi vozlišči (ker teh še nimamo); takim povezavam recimo *proste* (lahko bi tudi rekli, da „visijo v zraku“). Nato dodajmo drugo vozlišče s stopnjo d_2; iz njega izhaja d_2 povezav; eno od njih povežimo s prvim vozliščem, ostale pa naj zaenkrat ostanejo proste. Nato podobno dodajmo tretje vozlišče s stopnjo d_3; eno od njegovih d_3 povezav povežimo z enim od prvih dveh vozlišč (vseeno je, katerim), ostale pa naj še ostanejo proste. Tako nadaljujemo, dokler ne postavimo v drevo vseh n vozlišč. Če je na koncu kakšna povezava še prosta (torej ima le eno krajišče, drugi konec te povezave pa ni povezan z nobenim konkretnim vozliščem), dobljeno drevo ni veljavno, sicer pa smo našli drevo, po kakršnem sprašuje naloga.

Ko v graf dodamo k-to vozlišče s stopnjo d_k, smo rekli, da eno od povezav tega vozlišča povežemo z enim od prvih $k-1$ vozlišč (pri tem se število prostih povezav v dosedanjem delu drevesa zmanjša za 1), ostalih $d_k - 1$ povezav pa zaenkrat ostane prostih. Skupno število prostih povezav v grafu se pri tem torej poveča za $d_k - 2$ in torej zdaj znaša $d_1 + (d_2 - 2) + (d_3 - 2) + \ldots + (d_k - 2)$. Neugodno bi bilo, če bi to število kdaj padlo na 0, saj bi to pomenilo, da kasnejših vozlišč ne bi mogli povezati z dosedanjim delom grafa in naš graf na koncu ne bi bil povezan (naloga pa zahteva povezan graf). Opazimo lahko, da če v graf dodamo vozlišče s stopnjo $d_k > 1$, se število prostih povezav ohrani ali celo poveča, zmanjša pa se le pri $d_k = 1$. Zato je koristno, če najprej obdelamo vozlišča s stopnjo > 1, tista s stopnjo 1 pa prihranimo za na konec. Recimo brez izgube za splošnost, da imamo najprej m vozlišč s stopnjo > 1 (to so $d_1, \ldots, d_m$), ostalih $n-m$ vozlišč pa ima stopnjo natanko 1.[9] Ko obdelamo prvih m vozlišč, ima naše drevo torej takšno obliko:

[9] Poseben primer so še vozlišča s stopnjo 0. Če imamo $n = 1$ in ima tisto edino vozlišče stopnjo 0, bo že to vozlišče samo po sebi primerna rešitev naloge (če ima edino vozlišče stopnjo > 0, pa



Zdaj nam torej ostane še $(d_1+\ldots+d_m)-2(m-1)$ prostih povezav. Če je to število ravno enako številu točk s stopnjo 1 (to je $n-m$), bomo s temi točkami ravno zaprli vse proste povezave in dobili veljavno drevo; drugače pa nam bo bodisi zmanjkalo nekaj točk stopnje 1 (in preostalih prostih povezav ne bomo imeli kam povezati — če jih povežemo med sabo, bodo nastali cikli, tega pa naloga ne dovoli) bodisi nam jih bo nekaj ostalo (in jih ne bomo mogli povezati z dosedanjim delom grafa; če pa tiste preostale točke stopnje 1 povežemo med sabo, dobljeni graf ne bo povezan kot celota).

Vidimo torej, da naš dosedanji postopek uspe rešiti nalogo natanko tedaj, ko velja $(d_1+\ldots+d_m)-2(m-1)=n-m$; prepričajmo se še o tem, da v primerih, ko ta pogoj ne velja, takšnega drevesa, po kakršnem sprašuje naloga, sploh ni. Če na obeh straneh enačbe prištejemo $n+m-2$, dobimo $(d_1+\ldots+d_m)+(n-m)=2(n-1)$. Spomnimo se, da je $n-m$ ravno število točk s stopnjo 1, to so $d_{m+1}=d_{m+2}=\ldots=d_n=1$, tako da je $n-m=d_{m+1}+\ldots+d_n$. Naš pogoj je torej enakovreden pogoju $d_1+\ldots+d_n=2(n-1)$. Ali je mogoče, da drevo obstaja tudi v kakšnem takem primeru, ko ta pogoj ni izpolnjen? Ne, kajti drevo z n točkami ima vedno natanko $n-1$ povezav,[10] vsota stopenj vseh točk v njem pa je seveda dvakratnik števila povezav (ker ima vsaka povezava dve krajišči); če torej drevo obstaja, je pogoj $d_1+\ldots+d_n=2(n-1)$ gotovo izpolnjen.

Zapišimo našo rešitev še v C-ju. Predpostavili bomo, da dobimo seznam stopenj točk v tabeli `d` in da sme naš podprogram to tabelo spreminjati (številke točk naj bodo od 0 do $n-1$ namesto od 1 do n, tako da jih lahko uporabimo kar kot indekse v tabelo). Za začetek prerazporedimo elemente tabele `d` tako, da točke s stopnjo 1 pridejo na konec tabele; spotoma lahko še računamo vsoto stopenj vseh točk (spremenljivka `s`) in preštejemo točke s stopnjo 0 (spremenljivka `n0`). S pomočjo tega dvojega bomo lahko preverili, če je drevo za dane podatke sploh mogoče sestaviti ali ne. Če ga je, moramo zdaj izpisati nabor povezav, ki tvorijo drevo z zahtevanimi stopnjami točk. To lahko naredimo tako, da gremo po točkah in vsako od njih povežemo z eno od predhodnih točk — seveda s tako, ki še ima kakšno prosto povezavo. Spodnja rešitev uporabi vedno kar prvo tako točko, ki še ima kakšno prosto povezavo; za štetje prostih povezav uporabimo kar tabelo `d`, z indeksom `k` pa kažemo na prvo točko, ki še ima kakšno prosto povezavo, tako da nam tabele ni

zahtevanega drevesa ne moremo sestaviti). Če pa je $n>1$ in ima kakšno vozlišče stopnjo 0, se drevesa ne bo dalo sestaviti (ker vozlišče s stopnjo 0 ne bo moglo biti povezano s preostankom drevesa).

[10] O tem se lahko prepričamo z indukcijo po številu točk, n. Pri $n=1$ bi bila edina možna povezava zanka, to pa je v bistvu cikel, teh pa v drevesu ni; pri $n=1$ ima torej drevo 0 povezav, kar je res $n-1$. Recimo zdaj, da smo trditev že preverili za drevesa z manj kot n točkami. Vzemimo poljubno drevo z n točkami; v njem gotovo obstaja neka točka s stopnjo 1 (če take ni, se lahko hitro prepričamo, da mora v drevesu obstajati nek cikel, kar bi bilo protislovje); če to točko in njeno edino povezavo izrežemo iz grafa, le-ta ostane povezan (in brez ciklov), torej je še vedno drevo; ker pa ima $n-1$ točk, mora imeti (po induktivni predpostavki) $n-2$ povezav; pred brisanjem naše točke in njene povezave smo imeli torej drevo z n točkami in $n-1$ povezavami. Ker ta razmislek velja za poljubno drevo z n točkami, lahko zaključimo, da ima res vsako drevo z n točkami natanko $n-1$ povezav.

treba preiskovati vsakič od začetka. Časovna zahtevnost tega postopka je le $O(n)$.

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

bool SestaviDrevo(int n, int *d)
{
  int i, k, m, s, n0;
  /* Premaknimo točke s stopnjo 1 na konec tabele d. */
  for (i = 0, s = 0, n0 = 0, m = n; i < m; ) {
    /* Na tem mestu velja: obdelali smo že točke d[0..i − 1] in d[m..n − 1];
       vsota njihovih stopenj je s; med njimi je n0 točk s stopnjo 0; vse stopnje v
       d[m..n − 1] so enake 1, vse stopnje v d[0..i − 1] pa različne od 1. */
    /* Povečajmo vsoto stopenj in števec točk s stopnjo 0. */
    s += d[i]; if (d[i] == 0) n0++;
    /* Če ima točka i stopnjo 1, jo zamenjajmo s točko m − 1;
       tako pridejo točke s stopnjo 1 na konec tabele. */
    if (d[i] == 1 && i < m) { d[i] = d[−−m]; d[m] = 1; }
    else i++; }
  /* Ali je tako drevo sploh mogoče sestaviti? */
  if (s != 2 * (n − 1) || n0 > 1 || (n0 > 0 && n > 1)) return false;
  /* Izpišimo primeren nabor povezav. */
  for (i = 1, k = 0; i < n; i++)
  {
    /* Tabela d zdaj hrani za vsako točko le število prostih povezav,
       ne število vseh povezav. Na tem mestu tudi velja, da je k < i in
       da točke od 0 do k − 1 nimajo nobenih prostih povezav. */
    /* Poiščimo prvo naslednjo točko, ki še ima kakšno prosto povezavo. */
    while (d[k] == 0) k++;
    /* Povežimo to točko (torej k) z novo točko i. */
    printf("%d %d\n", i, k);
    d[k]−−; d[i]−−; /* Popravimo števca prostih povezav. */
  }
  return true;
}
```

12. Volitve

Razmislimo najprej o podnalogi (a); kot bomo videli kasneje, bo njena rešitev z nekaj majhnimi spremembami prišla prav tudi pri (b). Načeloma bi bilo dobro za vsako enoto naše hierarhije in za vsakega kandidata določiti, koliko glasov je ta kandidat dobil v tej enoti; težava je v tem, da je enot in kandidatov veliko, zato se moramo tega lotiti pazljivo, da ne bo nastal postopek s časovno ali prostorsko zahtevnostjo $O(mk)$ (pri tem pomeni m število enot, k pa število kandidatov; naloga pravi, da je lahko obojih po milijon).

Na misel nam lahko pride, da bi za vsako enoto i izračunali izide glasovanja G_i; to bi bila množica parov oblike (k, g), pri čemer je k številka kandidata, g pa število glasov, ki jih je dobil (na območju enote i). Takšne množice lahko precej učinkovito računamo od nižjih nivojev hierarhije proti višjim. Če je i neka osnovna enota, gredo vsi njeni glasovi enemu samemu kandidatu, tako da imamo preprosto $G_i = \{(k_i, p_i)\}$; če pa leži i višje v hierarhiji in so ji neposredno podrejene enote $c_{i1}, \ldots, c_{i,r_i}$, lahko množico G_i izračunamo tako, da združimo elemente množic $G_{c_{i1}}, \ldots, G_{c_{i,r_i}}$; če se v

več množicah pojavi isti kandidat k, njegove glasove pri tem seštejemo in ga dodamo v množico G_i le enkrat.

Ta postopek med drugim pričakuje, da za poljubno i poznamo seznam neposredno podrejenih enot; v vhodnih podatkih pa je ravno obratno, tam za vsako enoto i izvemo, katera (namreč t_i) ji je neposredno nadrejena. Pred začetkom izvajanja našega postopka moramo torej te podatke predelati v sezname neposredno podrejenih enot.

S pomočjo takšnih množic G_i ni težko odgovarjati na poizvedbe. Poizvedbe (k, i) lahko na začetku postopka uredimo tako, da pridejo vse poizvedbe za isto enoto i skupaj; nato pa, čim izračunamo množico G_i, gremo po vseh poizvedbah za to enoto in odgovorimo nanje. Na poizvedbo (k, i) odgovorimo tako, da pogledamo, če je v množici G_i kakšen element oblike (k, g); če je, je g odgovor na našo poizvedbo, sicer pa je odgovor 0 (ker kandidat k ni dobil v enoti i in njej podrejenih enotah nobenega glasu).

Ko razmišljamo o tem, kakšna je časovna (in prostorska) zahtevnost tega postopka, je glavno vprašanje to, koliko elementov bo v množicah G_i. Lahko si predstavljamo neugodno sestavljene vhodne podatke, pri katerih bi za $O(m)$ enot veljalo, da imajo njihove G_i po vsaj $O(m)$ elementov. Če bo to pomenilo, da porabimo za pripravo teh množic $O(m^2)$ časa (ali pa pomnilnika), bo naš postopek neučinkovit.

Zgoraj smo že videli, da čim izračunamo množico G_i, lahko takoj odgovorimo na vse poizvedbe za enoto i. To pa tudi pomeni, da odtlej ne potrebujemo več množic za enote, ki so podrejene enoti i, saj smo na vse njihove poizvedbe očitno odgovorili že prej (čim smo izračunali tiste množice); take podrejene množice lahko torej sproti pozabljamo. To bo prišlo prav v nadaljevanju razmisleka.

Naj bo $l(i)$ število osnovnih enot, ki so posredno ali neposredno podrejene enoti i. Če je i tudi sama osnovna enota, je $l(i) = 1$, če pa je i neka višja enota z neposredno podrejenimi enotami $c_{i1}, \ldots, c_{i,r_i}$, je $l(i) = l(c_{i1}) + \ldots + l(c_{i,r_i})$. Opazimo lahko, da za vsako enoto i velja $|G_i| \leq l(i)$ (kajti vsaka osnovna enota pod i prispeva glasove za enega kandidata v G_i, torej je v G_i lahko kvečjemu toliko kandidatov, kolikor je osnovnih enot; lahko pa je kandidatov tudi manj, če glasuje več osnovnih enot za istega kandidata).

Ko razmišljamo o množici G_i, ločimo nekaj možnosti: (1) če je i osnovna enota, torej če sploh nima podrejenih enot, njena množica G_i že po definiciji vsebuje le 1 element. (2) Če je i višja enota in ima natanko eno neposredno podrejeno enoto (recimo ji c), potem je množica G_i popolnoma enaka množici G_c. Ker smo malo prej videli, da množice G_c v prihodnje tako ali tako ne bomo več potrebovali, lahko prav to isto množico zdaj brez kakršnih koli nadaljnjih sprememb uporabimo kot množico G_i. S pripravo G_i torej pravzaprav nimamo nobenega dodatnega dela. (3) Ostane še možnost, da je i višja enota in ima več neposredno podrejenih enot; tem recimo $c_{i1}, \ldots, c_{i,r_i}$. Brez izgube za splošnost naj bo c_{i1} tista med njimi, ki ima največjo vrednost $l(c_{i1})$. Ker množic $G_{c_{i1}}, \ldots, G_{c_{i,r_i}}$ v prihodnje ne bomo več potrebovali za nič drugega razen za izgradnjo množice G_i, lahko G_i sestavimo tako, da gremo po vseh elementih množic $G_{c_{i2}}, \ldots, G_{c_{i,r_i}}$ in jih dodamo v množico $G_{c_{i1}}$ (pri tem pa sproti še preverjamo, če je nek kandidat v $G_{c_{i1}}$ že prisoten, in če je, le povečamo njegovo število glasov).

Z vsako enoto iz skupin (1) in (2) imamo torej le $O(1)$ dela, tako da je skupna

cena priprave množic G_i za vse te enote le $O(m)$. Za enote iz skupine (3) pa razmišljajmo takole: pri vsakem i iz te skupine imamo toliko dela, kolikor je skupaj elementov v množicah $G_{c_{i2}}, \ldots, G_{c_{i,r_i}}$, teh pa je največ toliko, kolikor je osnovnih enot na območju enot $c_{i2}, \ldots, c_{i,r_i}$. Zgornjo mejo za skupno ceno izračuna vseh množic G_i za enote iz skupine (3) lahko torej dobimo tako, da za vsako osnovno enoto (recimo e, za vse e od 1 do n) preštejemo, pri koliko njej nadrejenih enotah i iz skupine (3) se zgodi, da e ne leži znotraj c_{i1}, pač pa znotraj kakšne od i-jevih drugih neposredno podrejenih enot.

Mislimo si torej neko osnovno enoto e in se premikajmo od nje proti neposredno nadrejenim enotam, dokler ne pridemo do enote m, ki predstavlja celotno državo. Tako dobimo neko zaporedje $e_0, e_1, \ldots, e_d$, pri čemer je $e_0 = e$, $e_d = m$ in $e_i = t_{e_{i-1}}$ (za $i = 1, \ldots, d$). Kot smo videli v prejšnjem odstavku, bi radi ocenili, kolikokrat se v tem zaporedju zgodi, da ima e_j vsaj dve neposredno podrejeni enoti in da je obenem $e_{j-1} \neq c_{e_j,1}$. Vpeljimo okrajšavo $l_j := l(e_j)$. Kaj lahko povemo o zaporedju $l_0, \ldots, l_d$? Velja seveda $l_0 = 1$ (ker je e_0 osnovna enota) in $l_d = n$ (ker je e_d enota, ki pokriva celo državo, ta pa obsega n osnovnih enot). Hkrati tudi vemo, da je to zaporedje nepadajoče (če je e_j nadrejena enoti e_{j-1} in slednja vsebuje l_{j-1} osnovnih enot, vsebuje vse te osnovne enote tudi enota e_j, zato je $l_{j-1} \leq l_j$). Če pa pri nekem j velja, da ima e_j vsaj dve neposredno podrejeni enoti in da je $e_{j-1} \neq c_{e_j,1}$, lahko sklepamo še takole: e_j vsebuje vse osnovne enote iz e_{j-1} in še vse enote iz $c_{e_j,1}$, pri čemer je slednjih vsaj toliko kot v e_{j-1} (ker smo $c_{e_j,1}$ izbrali tako, da ima med vsemi enotami, ki so neposredno podrejene enoti e_j, največ osnovnih enot); torej je $l_j \geq l_{j-1} + l(c_{e_j,1}) \geq l_{j-1} + l_{j-1} = 2l_{j-1}$. Vsakič torej, ko e_0 prispeva k ceni izračuna množice G_{e_j}, se vrednost l_j vsaj podvoji; ker pa je zaporedje l_j-jev nepadajoče in se začne z 1 in na koncu doseže n, se ne more podvojiti več kot $(\log_2 n)$-krat. Posamezna osnovna enota e_0 (takih enot pa je n) torej prispeva k ceni izračuna množic G_i pri največ $\log_2 n$ enotah i iz skupine (3), zato je skupna cena izračuna vseh množic iz te skupine $O(n \log n)$.

Zapišimo dobljeni postopek še s psevdokodo:

```
    (* Pripravi množice C[i], ki za vsak i povedo,
       katere enote so neposredno podrejene enoti i. *)
 1  for i := 1 to m do C[i] := {};
 2  for i := 1 to m do dodaj i v C[t_i];

    (* Pripravi L, vrstni red enot od osnovnih proti višjim. *)
 3  L := prazen seznam; A := {m};
 4  while A ni prazna:
 5    naj bo i poljubna enota iz A;
 6    pobriši i iz A in dodaj i na konec seznama L;
 7    dodaj v A vse enote iz C[i];
 8  obrni vrstni red elementov v seznamu L;

    (* Združi poizvedbe po enotah. *)
 9  for i := 1 to m do Q[i] := {};
10  za vsako poizvedbo (k_j, e_j) iz vhodnih podatkov:
11    dodaj j v množico Q[e_j];

12  za vsako enoto i iz seznama L (v takem vrstnem redu, kot so v L):
```

```
         (* Izračunaj G_i. *)
13       if |C[i] = 0| then G_i := {(k_i, p_i)}; l_i := 0;
14       else if |C[i]| = 1 then
15         naj bo c edini element množice C[i];
16         G_i := G_c; l_i := l_c;
17       else: (* i ima vsaj dve neposredno podrejeni enoti *)
18         l* := 0; l_i := 0;
           (* Poiščimo c_1, to bo i-ju neposredno podrejena enota z največjo l_c. *)
19         za vsako c iz C[i]:
20           l_i := l_i + l_c;
21           if l_c > l* then c_1 := c; l* := l_c;
22         G_i := G_{c_1};
           (* Dodajmo v G_i še vsebino množic G_c za ostale
              i-ju neposredno podrejene enote c (za c ≠ c_1). *)
23         za vsako c iz C[i], če c ≠ c_1:
24           za vsako (k, p) iz G_c:
25             if v G_i obstaja element oblike (k, p')
26             then pri tistem elementu v G_i povečaj p' za p
27             else dodaj (k, p) v G_i;
28           pobriši množico G_c;

         (* Odgovori na poizvedbe za enoto i. *)
29       za vsako j iz Q[i]:
30         if v G_i obstaja element oblike (k_j, p)
31         then r := p else r := 0;
32         zdaj je r odgovor na j-to poizvedbo, torej na (k_j, e_j);
```

Množice G_i je koristno v praksi predstaviti z razpršenimi tabelami; tako lahko operacije, kot so dodajanje, brisanje in spreminjanje elementa, izvajamo v $O(1)$ časa. Za množice $C[i]$ in $Q[i]$ pa so dovolj dobri čisto navadni seznami, saj pri teh množicah potrebujemo le dodajanje elementa in sprehod po vseh elementih.

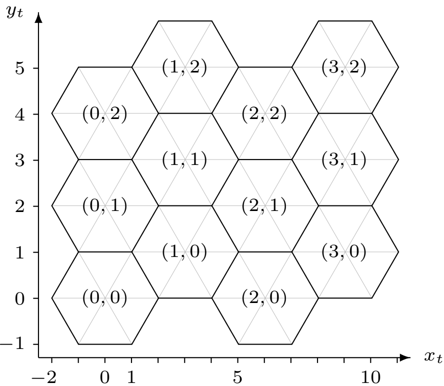
Zdaj vidimo, da naš postopek porabi $O(m)$ časa v korakih 1–8; $O(q)$ časa v korakih 9–11, če je q skupno število vseh poizvedb; $O(n)$ časa v koraku 13 (po vseh i skupaj); $O(m)$ časa v korakih 14–16 (po vseh i skupaj); kot smo videli zgoraj, $O(n \log n)$ časa v korakih 17–28 (po vseh i skupaj); in še $O(q)$ časa v korakih 29–32 (po vseh i skupaj). Časovna zahtevnost celotnega postopka je torej $O(q + m + n \log n)$. (Pri tem opozorimo, da so prireditve množic v korakih 16 in 22 mišljene tako, da se ne dela kopija množice na desni strani prireditve, pač pa G_i le pokaže na isti objekt kot množica na desni strani. Vsaka taka prireditev zato vzame le $O(1)$ časa.)

Če namesto podnaloge (*a*) rešujemo (*b*), lahko uporabimo zelo podoben postopek. Zmagovalca volitev v enoti i označimo z z_i. Korakov 9–11 in 29–32 zdaj ne potrebujemo, zmagovalce volitev v posamezni enoti pa lahko določimo takole: če je i osnovna enota (korak 13), je zmagovalec v njej $z_i := k_i$; če ima i natanko eno neposredno podrejeno enoto c (koraki 14–16), je zmagovalec v njej isti kot v enoti c, torej $z_i := z_c$; če pa ima i vsaj dve neposredno podrejeni enoti, lahko zmagovalca določimo sproti med izračunom množice G_i. Za začetek v koraku 22 postavimo še $z_i := z_{c_1}$, nato pa v vsaki iteraciji zanke 24–27 poglejmo še, če ima po tej iteraciji

kandidat k v množici G_i več glasov kot kandidat z_i; če jih ima, postavimo $z_i := k$. Na koncu tega postopka imamo tako izračunane zmagovalce v vseh enotah, časovna zahtevnost postopka pa je $O(m + n \log n)$, podobno kot prej.

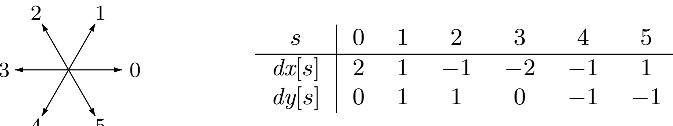
13. Šestkotna mreža

Naloga se dogaja na mreži enakostraničnih šestkotnikov, vendar pa jo bo lažje reševati, če vsakega od njih razdelimo na šest enakostraničnih trikotnikov. V to trikotno mrežo vpeljimo tudi nov koordinatni sistem (x_t, y_t), pri čemer je enota v smeri x dolga polovico stranice naših trikotnikov (in šestkotnikov), enota v smeri y pa je enaka višini trikotnika. Vsaka točka naše trikotne mreže je bodisi središče enega od šestkotnikov bodisi oglišče, v katerem se stikajo trije šestkotniki. Izhodišče našega novega koordinatnega sistema postavimo v središče tistega šestkotnika, ki je imel v prvotnem šestkotnem sistemu iz besedila naloge koordinate $(0, 0)$:



Gornja slika v vsakem šestkotniku v oklepajih kaže tudi njegove koordinate v prvotnem šestkotnem koordinatnem sistemu. Med obema sistemoma ni težko prehajati. Oglejmo si šestkotnik, ki je imel v prvotnem šestkotnem koordinatnem sistemu koordinati (x_s, y_s); kakšne so koordinate njegovega središča (x_t, y_t) v novem trikotnem koordinatnem sistemu? Opazimo lahko, da je mogoče iz šestkotnih koordinat priti v trikotne po formulah $x_t = 3x_s$ in $y_t = 2y_s + (x_s \bmod 2)$, iz teh pa ni težko dobiti tudi formul za prehod iz trikotnih koordinat v šestkotne: $x_s = x_t/3$ in $y_s = (y_t - (x_s \bmod 2))/2$.

Po naši mreži se lahko premikamo v šestih možnih smereh, ki jih bomo predstavili kar s števili od 0 do 5, kot kaže spodnja slika. V trikotnem koordinatnem sistemu je mogoče te premike opisati zelo preprosto: če se iz točke (x_t, y_t) premaknemo za eno enoto (stranico trikotnika) v smeri d, pridemo v točko $(x_t + dx[d], y_t + dy[d])$, pri čemer sta tabeli dx in dy prikazani spodaj:



s	0	1	2	3	4	5
$dx[s]$	2	1	−1	−2	−1	1
$dy[s]$	0	1	1	0	−1	−1

Tudi obračanja na mestu ni težko opisati. Če gledamo v smer d in se zasukamo za 60° v levo, je naša nova smer $(d+1) \bmod 6$; če pa se zasukamo za 60° v desno, je naša nova smer $(d-1) \bmod 6$. Slednje je sicer varneje računati kot $(d+5) \bmod 6$, da ne bo prišlo do napak pri $d = 0$. Podobno nas zasuk za 120° v levo postavi v smer $(d+2) \bmod 6$, zasuk za 120° v desno v smer $(d-2) \bmod 6$ (oz. $(d+4) \bmod 6$), zasuk za 180° pa v smer $(d+3) \bmod 6$. Da bo manj pisanja, v nadaljevanju rešitve tega mod 6 ne bomo pisali, si pa moramo pri računanju s smermi predstavljati, da rezultat pač vedno postavimo v razpon $0, \ldots, 5$.

S pomočjo številk smeri lahko označimo tudi oglišča, stranice in sosede posameznega šestkotnika. Oglišče d naj bo tisto, v katerega pridemo, če se iz središča šestkotnika premaknemo v smeri d; stranica d naj bo tista, ki povezuje oglišči d in $d+1$; na drugi strani te stranice pa leži šestkotnik, ki mu bomo rekli sosed d našega šestkotnika.

(a) Pri tej podnalogi dobimo mrežo šestkotnih polj, za vsakega od teh šestkotnikov pa imamo (v tabeli) podatek o tem, ali pripada našemu liku ali ne. Da si bomo to lažje predstavljali, bomo tistim poljem, ki pripadajo našemu liku, rekli, da so črna, ostalim pa, da so bela.

Opis lika bomo sestavili tako, da se bomo sprehajali po zunanjem robu lika in to tako, da bo lik ves čas na naši levi (na naši desni pa bodo bela polja). Tako bomo sčasoma obhodili celoten rob lika v pozitivni smeri (torej v smeri, ki je nasprotna smeri urinega kazalca). (Enakovreden opis bi seveda dobili tudi, če bi šli v smeri urinega kazalca in pazili na to, da bo lik ves čas na naši desni, prazna polja pa na naši levi.) Zapišimo postopek najprej s psevdokodo, nato pa bomo nekatere stvari v njem komentirali še malo podrobneje:

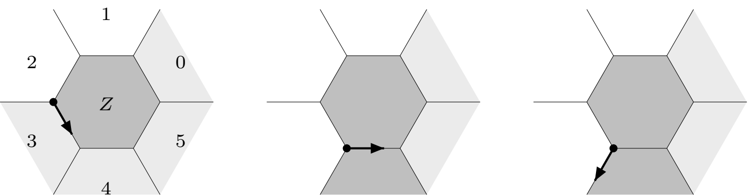
1 Poiščimo na mreži najvišje črno polje (torej tako z največjim y_s, če pa je takih več, vzemimo med njimi najbolj levega, torej tistega z najmanjšim x_s); recimo mu *začetno polje* in ga označimo z Z.
2 Z-jeva soseda 1 in 2 sta gotovo bela.
Postavimo se v Z-jevo oglišče 3 in se obrnimo v smer $d = 5$.
3 Na naši levi je Z, na naši desni je njegov sosed $d - 2$.
Če je ta sosed črn, pojdi na korak 5.
4 Sicer se premakni za eno enoto naprej in se obrni za 60° v levo ($d := d + 1$).
Če s tem pridemo nazaj v začetno točko ($d = 5$), je lik sestavljen le iz šestkotnika Z; postopek takoj končajmo in vrnimo opis $\langle 6 \rangle$.
Če pa še nismo nazaj v začetni točki, pojdi na korak 3.
5 Na naši levi je Z, na naši desni je njegov sosed $d - 2$ (ki je tudi črn), za nami pa je sosed $d - 3$ (ki je bel). Obrni se za 120° v desno; pravkar omenjeni črni sosed je zdaj na naši levi, beli sosed na naši desni, polje Z pa za nami.
Zapomni si trenutni položaj (x_t, y_t) v (x_t^0, y_t^0).
Inicializiraj *opis* kot prazen seznam.
6 Na naši levi je črno polje, na naši desni pa belo. Postavi $n := 0$.
7 Premakni se za eno enoto naprej, se obrni za 60° v levo in povečaj n za 1.
Če je na naši desni zdaj črno polje, pojdi na korak 8,
sicer pa pojdi nazaj na korak 7.
8 Dodaj n na konec seznama *opis*.

Če je trenutni položaj enak (x_t^0, y_t^0), končaj in vrni *opis*,
sicer pa pojdi nazaj na korak 6.

Kako preverimo, kakšne barve je polje na naši levi ali desni? Recimo, da stojimo v oglišču (x_t, y_t) in gledamo v smer d. V tem oglišču se stikajo trije šestkotniki; eden je na naši levi, eden na naši desni in eden za našim hrbtom. Slednji nas ne bo zanimal, o prvih dveh pa lahko ugotovimo, da bi v središče levega šestkotnika prišli, če bi se iz našega trenutnega položaja premaknili za eno enoto v smeri $d+1$, v središče desnega pa, če bi se premaknili v smeri $d-1$. Tako torej ni težko izračunati koordinat središč obeh šestkotnikov, iz teh pa tudi njunih (x_s, y_s), ki ju lahko uporabimo kot indeksa v vhodno tabelo, da bomo lahko preverili, ali sta šestkotnika prisotna v našem liku ali ne.

Kako v koraku 2 vemo, da sta Z-jeva soseda 1 in 2 bela? Označimo Z-jevi koordinati v šestkotni mreži z (x_s, y_s). Sosed 1 ima potem koordinate $(x_s, y_s + 1)$; sosed 2 pa bodisi $(x_s - 1, y_s)$ (če je x_s sod) bodisi $(x_s - 1, y_s + 1)$ (če je x_s lih). Oba soseda torej ležita bodisi v višji vrstici kot Z ali pa levo od njega v isti vrstici; vsa taka polja pa so bela, saj smo za Z vzeli najvišje črno polje (in če je v tisti vrstici več črnih, smo vzeli za Z najbolj levo med njimi).

Namen korakov 3 in 4 je, da poiščeta na robu Z-ja kakšno stičišče dveh črnih polj (Z-ja in enega od njegovih sosedov); v tem stičišču lahko potem res začnemo naš obhod po robu lika. Tadva koraka ilustrira naslednja slika:



Leva slika kaže naš začetni položaj; za Z vemo, da je črn, njegova soseda 1 in 2 sta bela, za ostale sosede pa v splošnem ne vemo, zato so na sliki osenčeni s svetlejšim odtenkom sive. Pri obhodu hočemo imeti lik ves čas na svoji levi. Ker za Z-jeva soseda 1 in 2 vemo, da sta bela, se lahko postavimo v Z-jevo oglišče 3 in gledamo v smer 5 (torej proti oglišču 4); tedaj imamo na svoji levi Z, na svoji desni pa soseda 3. Zdaj se v zanki sprehajamo vzdolž roba šestkotnika Z, dokler se na naši desni ne pojavi črno polje. (Na primer: če je sosed 3 bel, sosed 4 pa črn, pridemo po eni iteraciji koraka 4 v stanje, kot ga kaže srednja slika zgoraj.) Takrat vemo, da smo v stičišču dveh črnih polj, namreč Z-ja in nekega njegovega črnega soseda; in če se zdaj obrnemo za 120° v desno (korak 5), bomo imeli tega črnega soseda na svoji levi, prejšnjega soseda (ki je bel) pa na desni in bomo tako pripravljeni zares začeti z obhodom lika. (To stanje kaže desna slika zgoraj.)

Koraka 6 in 7 se premakneta naprej vzdolž roba trenutnega črnega polja, dokler ne prideta do naslednjega stiščišča s kakšnim črnim sosedom. Pri tem štejemo, koliko korakov smo naredili, in to število dodamo v opis. Ko pridemo do naslednjega stičišča, je situacija podobna kot v koraku 5 in se moramo spet obrniti za 120° v desno, da bomo lahko nadaljevali z obhodom. Ko pridemo nazaj v stičišče, kjer smo obhod začeli (njegove koordinate smo si zapomnili v koraku 5), pa vemo, da je obhod končan in lahko vrnemo doslej dobljeni opis.

Podnaloga (a) zahteva tudi, da preverimo, ali je opis lika v mreži sploh veljaven, torej povezan in brez lukenj. Tega naš dosedanji postopek ne počne; luknje bi sploh ignoriral, če pa je lik iz več nepovezanih delov, bi vrnil opis le enega od njih (tistega, ki mu pripada polje Z). Zato moramo dodati še preverjanje veljavnosti lika. To lahko zelo preprosto preverimo takole: če je lik veljaven, bo vsota števil v našem seznamu *opis* ravno enaka obsegu lika; če pa ni veljaven, bo pravi obseg večji (ker k njemu prispevajo tudi drugi nepovezani deli lika in notranji robovi okrog lukenj v njem). Pravi obseg pa lahko preprosto izračunamo takole: pojdimo po vseh poljih mreže; za vsako polje preverimo, če je črno; če je, preštejmo, koliko njegovih sosedov je belih; tako dobljena števila seštejmo in dobimo ravno obseg lika.

(b) Dani opis lika označimo z $\langle a_1, a_2, \ldots, a_n \rangle$. Predstavljajmo si, kako bi tak lik narisali. Moramo ga nekako vložiti v našo šestkotno mrežo, pri čemer pravzaprav ni pomembno, kam točno ga postavimo (če lik premikamo sem in tja po mreži ali ga vrtimo, se njegova oblika pri tem nič ne spreminja in tudi na veljavnost opisa to nič ne vpliva). Recimo na primer, da začnemo v $x_t = 1$, $y_t = 0$ (desno oglišče šestkotnika z $x_s = y_s = 0$) in gledamo v smeri 1. Dogovorimo se še, da bomo lik risali tako, da bo notranjost lika na naši levi, zunanjost pa na naši desni. Število a_1 v našem opisu nam zdaj pove, da zunanji rob lika vsebuje naslednjih a_1 stranic šestkotnika, ki je trenutno na naši levi. Za vsako od teh stranic se moramo torej najprej premakniti za eno enoto naprej in se nato obrniti za 60° v levo, da bomo pripravljeni na risanje naslednje stranice. Na koncu teh a_1 korakov se nahajamo v oglišču, kjer se dosedanji črni šestkotnik stika z naslednjim in trenutno smo obrnjeni tako, da gledamo naprej po daljici, s katero se stikata tadva šestkotnika. Ker bomo morali v nadaljevanju slediti zunanjemu robu drugega šestkotnika, se moramo zdaj obrniti za 120° v desno, preden lahko nadaljujemo naš obhod.

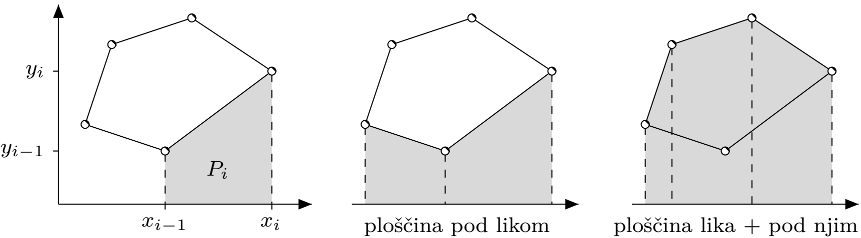
Ta postopek ponavljamo, dokler ne obdelamo vseh členov našega opisa. Opis je veljaven, če končamo v isti točki, kjer smo začeli, in če nobene točke ne obiščemo več kot enkrat. Zato je koristno, če si po vsakem opravljenem premiku zapomnimo novi položaj (x_t, y_t) v neki množici (recimo ji S; v praksi bi jo implementirali npr. z razpršeno tabelo), tako da bomo kasneje lahko preverjali, ali smo to točko že kdaj obiskali ali ne.

Zapišimo dobljeni postopek še s psevdokodo:

1 $x_t^0 := 1$; $y_t^0 := 0$; $d := 1$; (* *začetni položaj in smer* *)
$x_t := x_t^0$; $y_t := y_t^0$; $S := \{\}$;
2 **for** $i := 1$ **to** n:
3 **for** $j := 1$ **to** a_i:
4 $x_t := x_t + dx[d]$; $y_t := y_t + dx[d]$; (* *premik naprej* *)
$d := (d + 1) \bmod 6$; (* *obrni se 60° v levo* *)
5 **if** $(x_t, y_t) \in S$
then return false; (* *opis je neveljaven* *)
else dodaj (x_t, y_t) v množico S;
6 $d := (d + 4) \bmod 6$; (* *obrni se 120° v desno* *)
7 **return** $x_t = x_t^0$ **and** $y_t = y_t^0$; (* *opis je veljaven, če smo prišli nazaj v začetno točko* *)

Naloga zahteva, da moramo izračunati tudi ploščino lika (če je njegov opis veljaven).

Pri tem si lahko pomagamo z znanim postopkom iz računske geometrije, s katerim lahko izračunamo ploščino poljubnega mnogokotnika, le malo ga bomo prilagodili za naše potrebe. Recimo, da imamo mnogokotnik z oglišči $T_0, \ldots, T_n$, pri čemer ima T_i koordinati (x_i, y_i) in velja $x_0 = x_n$, $y_0 = y_n$; mislimo si še, da so vse $y_i > 0$, torej da lik v celoti leži nad x-osjo. Oglejmo si stranico $T_{i-1}T_i$. Potegnimo navpično črto od vsakega krajišča do x-osi; tako dobimo trapez z oglišči $T_{i-1} = (x_{i-1}, y_{i-1})$, $T_i = (x_i, y_i)$ in še $(0, y_{i-1})$ ter $(0, y_i)$. Širina tega trapeza je $x_i - x_{i-1}$, višina pa gre od y_{i-1} do y_i; povprečna višina je torej $(y_{i-1} + y_i)/2$, ploščina pa zato $P_i := (x_i - x_{i-1})(y_{i-1} + y_i)/2$. Primer tega kaže leva slika spodaj:



Recimo še, da so oglišča oštevilčena v pozitivni smeri (nasproti smeri urinega kazalca), tako da imamo, če se sprehajamo po ogliščih v tem vrstnem redu, mnogokotnik ves čas na svoji levi. Stranice lahko razdelimo na tiste, ki imajo $x_i > x_{i-1}$, in tiste, ki imajo $x_i < x_{i-1}$. Pri prvih je mnogokotnik nad stranico, pri drugih pa pod njo; pri prvih je P_i pozitivna, pri drugih pa negativna. Spomnimo se, da je P_i ploščina trapeza med stranico in x-osjo; če seštejemo te trapeze po vseh takih stranicah, ki imajo mnogokotnik nad sabo, dobimo ploščino območja pod mnogokotnikom (med njim in x-osjo; glej srednjo sliko zgoraj):

$$P_{pod} = \textstyle\sum_{i: x_i > x_{i-1}} P_i;$$

če pa seštejemo trapeze tistih stranic, ki imajo mnogokotnik pod sabo, dobimo ploščino območja, ki ga sestavljata mnogokotnik in prostor pod njim (vse do x-osi; glej desno sliko zgoraj).

$$P_{lika} + P_{pod} = \textstyle\sum_{i: x_i < x_{i-1}} (-P_i).$$

V tej drugi formuli smo morali uporabiti $-P_i$, ker je P_i pri teh stranicah negativna. Ploščina našega lika je torej razlika med obojim, kar je ravno $-\sum_i P_i$.

V našem primeru lahko za koordinate oglišč mnogokotnika uporabimo kar pare (x_t, y_t), ki jih dobivamo po vsakem premiku v koraku 4 našega postopka. Spomnimo se, da ena enota na x_t-osi naše trikotne mreže predstavlja polovico stranice osnovnih trikotnikov (in šestkotnikov), ena enota na y_t-osi pa višino osnovnega trikotnika. Produkt teh dveh enot je torej ravno ploščina enega osnovnega trikotnika. Ploščina, kot jo dobimo po formuli $-\sum_i P_i$, je torej izražena v ploščinah osnovnih trikotnikov; naloga pa zahteva ploščino kot število osnovnih šestkotnikov, zato jo moramo še deliti s 6 (vsak šestkotnik je sestavljen iz šestih trikotnikov). Paziti moramo še na to, da čeprav je ploščina celotnega lika $-\sum_i P_i$ gotovo celo število trikotnikov, pa posamezni P_i niso nujno cela števila, saj so oblike *nekaj*/2 in števec tega ulomka

je lahko tudi lih. Da nam ne bo treba računati z ne-celimi števili, je bolje, če med izračunom delamo z dvakratniki ploščin in jih razpolovimo šele na koncu.

Vse, kar moramo torej narediti, je, da korak 4 našega postopka popravimo takole:

$x'_t := x_t$; $y'_t := y_t$; (* *zapomnimo si prejšnjo točko* *)
$x_t := x_t + dx[d]$; $y_t := y_t + dy[d]$; $d := (d + 1) \bmod 6$; (* *premik in zasuk* *)
$P := P - (x_i - x_{i-1})(y_{i-1} + y_i)$;

Na začetku postopka moramo inicializirati P na 0, na koncu pa (če se je opis izkazal za veljavnega) vrnemo $P/12$.

(*c*) Recimo, da vhodna opisa dobimo v tabelah $\langle a_0, \ldots, a_{n-1}\rangle$ in $\langle b_0, \ldots, b_{n-1}\rangle$; predpostavili smo, da sta enako dolgi, saj če nista, lahko takoj zaključimo, da se opisa nanašata na različne like.

Če se opisa nanašata na isti lik, je vendarle mogoče, da je eden od opisov ciklično zamaknjen glede na drugega; torej da za nek zamik z velja $a_i = b_{(z+i) \bmod n}$ za $i = 0, 1, \ldots, n-1$. To lahko preverimo tako, da gremo z zanko po vseh možnih z in pri vsakem še v gnezdeni zanki po vseh i ter preverjamo te pogoje. Če se pri nobenem zamiku zaporedji ne ujemata, lahko prvi opis še obrnemo in postopek ponovimo; tako bomo zaznali tudi primere, ko eden od opisov sledi liku v nasprotni smeri kot drugi. Če na ta način ne pridemo do ujemanja, vemo, da se opisa nanašata na različna lika. Zapišimo ta postopek s psevdokodo:

1 **for** $z := 0$ **to** $n - 1$:
2   $i := 0$; **while** $i < n$:
3   **if** $a_i \neq b_{(z+i) \bmod n}$ **then break**
      else $i := i + 1$;
4   **if** $i = n$ **then return true**;
5 $i := 0$; $j := n - 1$; **while** $i < j$ **do** zamenjaj a_i in a_j; $i := i + 1$; $j := j + 1$;
6 ponovi korake 1–4;
7 **return false**;

Slabost te rešitve je, da v najslabšem primeru porabi $O(n^2)$ časa. Učinkovitejši postopek dobimo z naslednjo idejo: med vsemi zaporedji, ki jih lahko dobimo s cikličnim zamikanjem zaporedja a, vzemimo tisto, ki je leksikografsko največje. Primer: iz zaporedja $\langle 5, 1, 5, 3\rangle$ lahko z zamikanjem dobimo še $\langle 1, 5, 3, 5\rangle$, $\langle 5, 3, 5, 1\rangle$ in $\langle 3, 5, 1, 5\rangle$. Med vsemi štirimi vzamemo torej $\langle 5, 3, 5, 1\rangle$, ki je leksikografsko največje. Podobno naredimo nato še z zaporedjem b. Če je leksikografsko največje zaporedje v obeh primerih enako, se vhodna cikla ujemata, sicer pa sta različna. (Nato moramo podobno kot zgoraj enega od njiju še obrniti in postopek ponoviti.) Lepo pri tem je, da lahko zamik, ki nam dá leksikografsko največje zaporedje, z nekaj zvitosti poiščemo že v $O(n \log n)$ časa; glej rešitev naloge 2012.3.5 na str. 67 v biltenu 2012 (tista naloga se je ukvarjala s točno tem problemom).

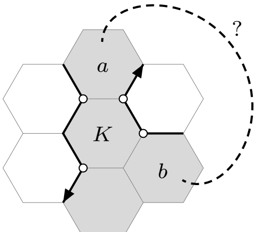
(*d*) Spomnimo se, da smo v rešitvi podnaloge (*b*) sledili opisu našega lika tako, da smo imeli lik ves čas na svoji levi. V vsaki točki, ki leži na stičišču dveh črnih šestkotnikov — torej v trenutku, ko začnemo z zanko v koraku 3 tamkajšnjega postopka obdelovati naslednji člen a_i našega vhodnega zaporedja — lahko izračunamo

koordinati (x_s, y_s) šestkotnika, ki trenutno leži na naši levi (in po čigar robu se bomo sprehajali, ko bomo obravnavali člen a_i). Formule za to smo že videli v rešitvi podnaloge (a). Ko na ta način pridemo do člena a_k (to je tisti, pri katerem hočemo pripadajoči črni šestkotnik pobrisati iz lika), si zapomnimo koordinati njegovega šestkotnika; recimo temu šestkotniku K.

Radi bi preverili, ali lik po brisanju K-ja ostane veljaven; spomnimo se, da je lik veljaven, če je povezan in nima lukenj. Pri tem smemo predpostaviti, je pred brisanjem zagotovo bil veljaven. Brisanje K-ja gotovo ne more ustvariti luknje v liku; K je namreč zagotovo imel vsaj enega belega soseda (če bi bili vsi njegovi sosedje črni, bi K ležal v notranjosti lika in se nanj sploh ne bi nanašal noben člen v opisu lika, saj ti členi odražajo sprehod po zunanjem robu lika) in če bi K po brisanju pripadal neki luknji v liku, bi to pomenilo, da je tej isti luknji pripadal tisti beli sosed že pred brisanjem — to pa je nemogoče, saj smo rekli, da je bil lik pred brisanjem K-ja veljaven, torej ni mogel imeti lukenj.

Razmislimo še o povezanosti lika po brisanju. Spomnimo se, kaj člen a_k pravzaprav pove o našem sprehodu po zunanjem robu lika: pove nam, da smo na rob polja K vstopili iz nekega sosednjega šestkotnika (to je nek K-jev črni sosed), nato smo naredili a_k korakov po robu K-ja (in pri tem imeli na svoji desni bela polja; tu ima je torej skupina a_k sosedov K-ja, ki so bele barve) in nato se od K-ja spet odlepili in nadaljevali pot po robu nekega njegovega črnega soseda (mogoče celo istega soseda, po čigar robu smo prej prišli do K-ja). Člen a_k nam torej pove, da je vsaj nek del K-jeve soseščine videti takole: črno polje, nato skupina a_k belih polj in nato še eno črno polje (lahko tudi isto kot prvo).

Mogoče je, da vsebuje naš opis še kak drug člen, ki se nanaša na isti šestkotnik K. Vsak tak člen pomeni, da ima K neko strnjeno skupino belih sosedov, pred in za njo pa po enega črnega soseda. Če je takih členov več (lahko sta dva ali celo trije), ima torej K več strnjenih skupin belih sosedov, med njimi pa so strnjene skupine črnih sosedov, ki so med seboj ločene s skupinami belih sosedov. Primer kaže naslednja slika:



Primer kaže polje K, na katero se nanašata dva člena v opisu našega lika (en člen ima vrednost 2, en pa vrednost 1); zato K obdajata dve skupini črnih polj in dve skupini belih polj. Če obstaja od a do b še kakšna pot (po samih črnih poljih), ki ne gre skozi K, mora zaobiti eno od skupin K-jevih belih sosedov (kot kaže črtkana črta), torej ta skupina tvori luknjo v prvotnem liku.

Ali je mogoče, da tak lik po brisanju K-ja ostane povezan? To pomeni, da se mora dati od ene skupine K-jevih črnih sosedov še vedno dati priti tudi v drugo skupino K-jevih črnih sosedov, četudi je K zdaj bel; torej mora obstajati neka druga pot (po samih črnih poljih), ki zaobide tudi K-jeve bele sosede; v prvotnem liku (pred brisanjem K-ja) pa bi lahko to pot dopolnili še s črnim poljem K in tako dobili cikel, ki popolnoma zaobjame eno od skupin K-jevih belih sosedov. Ta skupina je bila torej luknja v prvotnem liku, to pa je v protislovju s predpostavko, da je bil vhodni opis veljaven. Torej ni mogoče, da bi bil tak lik po brisanju povezan (in s

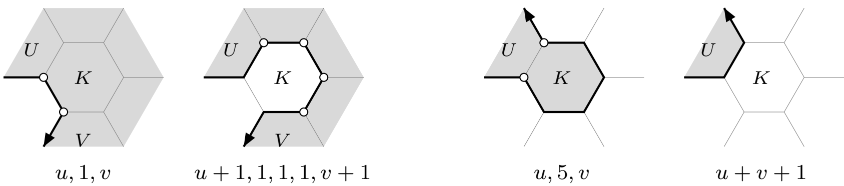
tem veljaven).

Po drugi strani, če naš opis ne vsebuje nobenega drugega člena, ki bi se nanašal na šestkotnik K, to pomeni, da ima K le eno strnjeno skupino črnih sosedov in eno strnjeno skupino belih sosedov. V tem primeru pa lik po brisanju K-ja gotovo ostane povezan: katerakoli pot (od enega črnega polja do drugega), ki je prej vodila skozi K, je morala v K vstopiti iz enega od njegovih črnih sosedov in nato izstopiti v nekega drugega črnega soseda; in ker vsi K-jevi črni sosedje tvorijo eno samo strnjeno skupino, lahko to pot zdaj popravimo tako, da gre od enega soseda do drugega, ne da bi šla vmes v K, torej brisanje K-ja ne bo nič poslabšalo povezanosti ostalih polj v liku.

Tako torej vidimo, da lahko veljavnost lika po brisanju preverimo preprosto tako, da gremo še enkrat po celem opisu in pogledamo, ali se na polje K nanaša še kak drug člen poleg člena a_k.

Zdaj moramo razmisliti le še o tem, kako popraviti opis, da bo odražal stanje lika po brisanju polja K. To je odvisno od vrednosti člena a_k. Recimo, da je imel prejšnji člen (to je a_{k-1}, razen pri $k = 1$, ko je prejšnji člen a_n) vrednost u, naslednji člen (to je a_{k+1}, razen pri $k = n$, ko je naslednji člen a_1) pa vrednost v. Šestkotnik, na katerega se je nanašal prejšnji člen, označimo z U, šestkotnik naslednjega člena pa z V.

Pri $a_k = 1$ je zdaj stanje takšno (glej levo sliko spodaj): ker ima ta člen vrednost 1, to pomeni, da ima K enega belega soseda in strnjeno skupino petih črnih sosedov; slednja se začne z U (iz katerega smo prišli v K) in konča z V (v katerega smo šli iz K), vmes pa so še trije drugi črni sosedje. Če K pobrišemo (glej drugo sliko spodaj), poteka pot po robu U-ja eno stranico dlje kot prej (zato se mora prejšnji člen povečati z u na $u + 1$), podobno tudi pot po robu V-ja (zato se mora naslednji člen povečati z v na $v + 1$); vmes pa imamo namesto člena a_k zdaj tri člene z vrednostjo 1, ki odražajo sprehod po vmesnih treh K-jevih črnih sosedih. V našem cikličnem opisu se je torej podzaporedje $u, 1, v$ spremenilo v $u + 1, 1, 1, 1, v + 1$.



$u, 1, v$ $\quad$ $u + 1, 1, 1, 1, v + 1$ $\quad$ $u, 5, v$ $\quad$ $u + v + 1$

Pri $a_k = 2, 3, 4$ je razmislek podoben in nas pripelje do naslednjih sprememb:

$$\begin{aligned} u, 2, v &\quad\to\quad u + 1, 1, 1, v + 1 \\ u, 3, v &\quad\to\quad u + 1, 1, v + 1 \\ u, 4, v &\quad\to\quad u + 1, v + 1. \end{aligned}$$

Pri $a_k = 5$ je situacija malenkost drugačna (glej tretjo sliko zgoraj). Opazimo lahko, da se prejšnji in naslednji člen v tem primeru nujno nanašata na isto šestkotno polje ($U = V$). Po brisanju K-ja se oba člena zlijeta v enega: najprej imamo u korakov po tem črnem sosedu (enako kot pred brisanjem), nato še en korak (po stranici, ki razmejuje tega soseda od K) in nato še v korakov (enako kot po brisanju). Namesto podzaporedja $u, 5, v$ imamo torej zdaj le en člen $u + v + 1$.

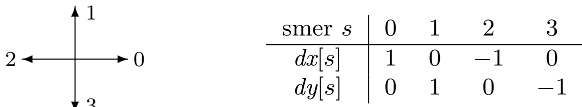
Pri tem moramo paziti še na naslednji posebni primer: mogoče je, da se prejšnji in naslednji člen ne le nanašata na isti šestkotnik, ampak da sta to dejansko en in isti člen v opisu. Do tega pride, če ima opis le dva člena; edini tak opis je $\langle 5, 5\rangle$, iz katerega po brisanju enega od črnih polj nastane $\langle 6\rangle$.

Ostane nam še primer $a_k = 6$; tedaj vemo, da je lik obsegal le polje K in nič drugega, tako da je po brisanju čisto prazen in je tudi kot njegov opis še najbolj smiselno vzeti kar prazno zaporedje.

14. Kocke

(*a*) Vhodni opis lahko obdelujemo znak po znak in pri vsakem znaku izračunamo, v kateri celici mreže se zdaj nahajamo in v katero smer smo obrnjeni (začetni položaj in smer si lahko izberemo poljubno, saj za veljavnost poti ni pomembno, kam v mrežo postavimo našo pot in kako jo zasukamo). Koordinate vseh že obiskanih celic si zapomnimo v neki primerni podatkovni strukturi (na primer v razpršeni tabeli), s pomočjo katere bomo lahko sproti preverjali, če naš opis kakšno celico doseže več kot enkrat (v tem primeru je namreč neveljaven). Na koncu pa moramo preveriti še, če se naša pot konča v isti točki, v kateri se je začela.

Da nam pri računanju koordinat ne bo treba delati z necelimi števili, bomo predpostavili, da so naše ploščice kvadrati s stranico 2 namesto 1. Smer, v katero smo pri sprehodu po črti trenutno obrnjeni, pa predstavimo s številom od 0 do 3, kot kaže spodnja slika. Definirajmo še dve tabeli, *dx* in *dy*, ki povesta, kako se spremenijo naše koordinate, če se premaknemo za eno enoto v določeno smer.



smer s	0	1	2	3
$dx[s]$	1	0	-1	0
$dy[s]$	0	1	0	-1

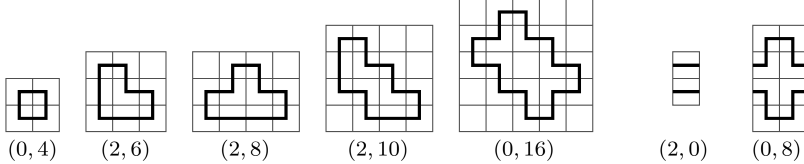
Zdaj lahko naš postopek takole zapišemo s psevdokodo:

vhod: niz *opis*, ki ga tvori n znakov `I`, `L`, `D`;

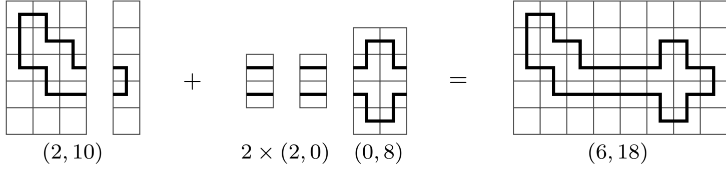
1 $x_0 := 0$; $y_0 := 0$; $s := 0$; (* *začetni položaj in smer* *)
  $x := x_0$; $y := y_0$; (* *postavimo se na začetni položaj* *)
  $S := \{\}$; (* *množica že pokritih celic mreže* *)
2 **for** $i := 1$ **to** n:
3   (* *Trenutno smo na robu neke celice; naredimo korak v smeri s, da* *)
    $x := x + dx[s]$; $y := y + dy[s]$; (* *pridemo v središče te celice.* *)
4   **if** $(x, y) \in S$ **then return false**; (* *Smo to celico že kdaj prej obiskali?* *)
    else dodaj (x, y) v množico S; (* *Sicer si jo zapomnimo zdaj.* *)
5   (* *Če je trenutni korak ovinek, popravimo smer.* *)
    if $opis[i]$ = `L` **then** $s := (s + 1) \bmod 4$
    else if $opis[i]$ = `D` **then** $s := (s + 3) \bmod 4$;
6   (* *Naredimo korak naprej v novi smeri, da pridemo na rob celice.* *)
    $x := x + dx[s]$; $y := y + dy[s]$;
7 (* *Na koncu še preverimo, če je črta sklenjena.* *)
  return $x = x_0$ **and** $y = y_0$;

(*b*) Za nekaj majhnih parov (s, t) lahko ročno sestavimo primeren razpored ploščic,

pri katerem nastane veljavna sklenjena pot. Vidimo jih na levem delu spodnje slike, pod vsakim pa je napisan pripadajoči (s, t):



Na desni strani slike pa vidimo dva razporeda ploščic (eden ima dve ploščici tipa I, drugi pa osem ploščic tipa L), s katerima lahko razširimo like na levi strani slike. Vsak od tistih likov ima namreč na desni strani par ploščic ⊟, pri katerem lahko lik v mislih prerežemo in na tistem mestu vstavimo enega ali več takih dopolnilnih razporedov ploščic. Primer kaže naslednja slika, pri kateri smo v lik $(2, 10)$ vrinili dva razporeda $(2, 0)$ in enega $(0, 8)$ ter tako dobili lik $(6, 18)$, torej iz 6 ploščic tipa I in 18 ploščic tipa L:



Iz $(0, 4)$ lahko s takšnim dopolnjevanjem dobimo $(2, 4)$; skupaj s prej prikazanimi osnovnimi liki imamo zdaj pri $s = 2$ pokrite primere $t = 4, 6, 8, 10$; iz teh pa lahko z dodajanjem razporeda $(0, 8)$ dobimo (pri istem s) tudi $t = 12, 14, 16, 18$, nato še $t = 20, 22, 24, 28$ in tako naprej; skratka, sestaviti znamo razpored za $s = 2$ in poljuben sod $t \geq 4$. Od tam lahko z dodajanjem razporeda $(2, 0)$ dosežemo (pri vseh teh t-jih) tudi $s = 4, 6, 8$ in tako naprej. Pri $s = 0$ pa smo zgoraj z osnovnimi liki pokrili primera, ko je $t = 4$ in $t = 16$, od tam pa lahko z dodajanjem razporeda $(0, 8)$ dosežemo tudi $t = 12, 20, 24, 30$ in tako naprej. Zdaj torej znamo sestaviti primerne poti v tistih primerih, ko veljajo vsi naslednji pogoji:

- s in t morata biti soda;
- t mora biti ≥ 4;
- če je $s = 0$, mora biti t večkratnik števila 4 in različen od 8.

Prepričajmo se, da je v drugih primerih (če s in t ne ustrezata tem pogojem) problem nerešljiv. Predstavljajmo si poljubno veljavno sklenjeno pot v naši karirasti mreži; recimo spet, da so naše ploščice (in celice mreže) kvadratki velikosti 1×1. Sprehodimo se po naši poti (vseeno, v kateri smeri); vsakič ko naša pot vstopi v novo celico, si zapomnimo „stanje“ — trojico $(x \bmod 2, y \bmod 2, s)$, pri čemer sta (x, y) koordinati nove celice, $s \in \{S, J, V, Z\}$ pa je smer, v katero smo bili obrnjeni, ko smo v to celico vstopili (da si bomo smeri lažje predstavljali, smo jih tukaj označili po straneh neba namesto s številkami).

Če poznamo stanje ob vstopu v neko celico in če vemo, ali naša pot v tisti celici naredi ovinek levo, ovinek desno ali korak naprej, lahko iz tega enolično določimo

stanje ob vstopu v naslednjo celico. Na primer, recimo, da smo vstopili v celico (x, y) obrnjeni proti vzhodu in naredili korak v levo; s tem torej pridemo v celico $(x, y+1)$ in smo obrnjeni proti severu. Tako smo torej iz stanja $(x \bmod 2, y \bmod 2, V)$ prišli v stanje $(x \bmod 2, (y+1) \bmod 2, S)$. Pri starem stanju sicer ne poznamo vrednosti x in y, pač pa le $x \bmod 2$ in $y \bmod 2$, vendar je to že dovolj, da izračunamo novo stanje: v slednjem potrebujemo $(y+1) \bmod 2$, kar je ravno enako $1 - (y \bmod 2)$.

Da bo manj pisanja, bomo stanja odslej pisali brez oklepajev in vejic. V prejšnjem odstavku smo torej videli, da bi na primer iz $01V$ z ovinkom levo prišli v $00S$; podobno bi se lahko tudi prepričali, da bi iz $01V$ z ovinkom desno prišli v $00J$, s korakom naravnost naprej pa v $11V$. Če zdaj tak razmislek opravimo za vsa možna stanja, lahko prehode med njimi kratko in jedrnato opišemo tako, da stanja zapišemo v naslednjo tabelo:

$00S$	$00J$	$01S$	$01J$
$10Z$	$10V$	$00Z$	$00V$
$11J$	$11S$	$10J$	$10S$
$01V$	$01Z$	$11V$	$11Z$

Možni prehodi med stanji so zdaj naslednji:

- Ovinek nas premakne za eno vrstico navzdol (iz zadnje vrstice pa nazaj v prvo), če smo bili v prvih dveh stolpcih, oz. za eno navzgor (iz prve vrstice pa nas premakne v zadnjo), če smo bili v drugih dveh stolpcih.

- Če smo bili v levi polovici tabele, nas ovinek v levo pusti v istem stolpcu, ovinek v desno pa nas premakne iz prvega stolpca v drugega ali obratno. Če smo bili v desni polovici tabele, nas ovinek v desno pusti v istem stolpcu, ovinek v levo pa nas premakne iz tretjega stolpca v četrtega ali obratno.

- Pri koraku naravnost naprej ostanemo v isti vrstici, le da se premaknemo iz prvega stolpca v tretjega (ali obratno) oz. iz drugega v četrtega (ali obratno).

Če imamo opravka z neko veljavno sklenjeno potjo, bomo na koncu seveda pristali v istem stanju, v katerem smo začeli. Brez izgube za splošnost lahko predpostavimo, da smo začeli v $00S$ (to lahko vedno dosežemo, le lik moramo primerno postaviti v ravnino, ga primerno zasukati in si primerno izbrati začetek poti), torej moramo v tem stanju tudi končati. Zdaj na primer vidimo, da nas vsak korak naravnost naprej (do teh pride pri ploščicah tipa I) premakne iz leve polovice tabele v desno ali obratno; ker smo začeli v levi polovici tabele in moramo tam tudi končati, iz tega sledi, da mora biti ploščic tipa I sodo mnogo. Tako torej vidimo, da če je s lih, je problem res nerešljiv.

Podobno vidimo, da nas vsak ovinek premakne iz trenutne vrstice v sosednjo vrstico, torej iz lihe v sodo ali obratno; ploščice tipa I pa nas premikajo le znotraj trenutne vrstice. Ker smo začeli v prvi vrstici, ki je liha, potrebujemo sodo mnogo ovinkov, da bomo spet v lihi vrstici; tako torej vidimo, da če je t lih, je problem res nerešljiv.

Če je $t = 0$, je problem nerešljiv, ker imamo v tem primeru le ploščice tipa I in poti ni mogoče skleniti. Če je $t = 2$, to pomeni, da moramo enkrat narediti ovinek v levo in enkrat ovinek v desno, da bomo na koncu obrnjeni v enako smer

kot na začetku; recimo, da je torej naša pot take oblike: najprej a korakov naprej, nato ovinek v levo, nato b korakov naprej, nato ovinek v desno in nato še c korakov naprej. Ni se težko prepričati, da je razlika med začetno in končno točko naše poti v tem primeru $(b + 1, a + c + 1)$, torej pot gotovo ni sklenjena (saj morajo biti a, b in c nenegativni). Tako torej vidimo, da je pri $t < 4$ problem res nerešljiv.

Če je $s = 0$, nimamo ploščic tipa I, torej nimamo korakov naravnost naprej, kar pomeni, da se ves čas gibljemo le po stanjih iz leve polovice tabele. Zdaj nas vsak ovinek premakne za eno vrstico navzdol (iz četrte pa nazaj v prvo), kar pomeni, da nazaj v prvo vrstico pridemo šele enkrat na vsake štiri korake. Tako torej vidimo, da če je $s = 0$, potem t res mora biti večkratnik 4, sicer je problem nerešljiv.

Ostane nam le še primer, ko je $s = 0$ in $t = 8$. Da je ta nerešljiv, se lahko prepričamo tako, da si napišemo program, ki z rekurzijo preizkusi vse možne oblike poti iz osmih ploščic tipa L in se prepriča, da res nobena od njih ni veljavna (sklenjena in brez prekrivanja). Pri vsakem ovinku imamo dve možnosti (ali je levi ali desni), tako da je skupaj $2^t = 256$ možnih poti. Več o tem, kako to narediti, bomo videli pri rešitvi podnaloge (c). □

Zdaj torej vemo, za katere (s, t) je problem rešljiv in za katere ni. Naloga pravi še, da mora naš postopek sestaviti primeren opis poti. Spomnimo se, da smo pot v vsakem primeru sestavili takole: vzeli smo enega od osnovnih likov in ga prerezali na levi in desni del, pri čemer levi del obsega vse razen zadnjega stolpca, desni del pa obsega le zadnji stolpec (s ploščicama ⊐); ta rez je prerezal pot po osnovnem liku pri dveh točkah, zgornji in spodnji; nato pa smo med oba dela vrinili nič ali več razporedov tipa $(0, 8)$ in $(2, 0)$. V vsakem od teh vrinjenih razporedov pa je črta sestavljena iz dveh ločenih kosov, spodnjega in zgornjega. Opis celotne poti je torej takšen: najprej pot po levem delu osnovnega lika (recimo, da od zgornje točke do spodnje); nato spodnji kos poti po vseh vrinjenih razporedih; nato pot po desnem delu osnovnega lika (od spodnje točke do zgornje; ta del poti bo vedno `DD`); in nato še zgornji kos poti po vseh vrinjenih razporedih (pri čemer jih moramo zdaj našteti v nasprotnem vrstnem redu kot prej). Vse te kose poti lahko predstavimo kot nize (v programu jih shranimo kot konstante) in jih nato le primerno staknemo skupaj:

if (s je lih) **or** (t je lih) **or** ($t < 4$) **or** ($s = 0$ **and** $t = 8$) **then return false**;
(* *Izberimo si osnovni lik* (s_0, t_0). *)
if $s = 0$ **then**
  $s_0 := 0$; **if** (t mod 8) $= 0$ **then** $t_0 := 16$ **else** $t_0 := 4$;
else:
  $t_0 := 4 + (t - 4) \bmod 8$;
  if $t_0 = 4$ **then** $s_0 := 0$ **else** $s_0 := 2$;
(* *Koliko vmesnih razporedov potrebujemo?* *)
$n_{08} := (t - t_0)/8$; $n_{20} := (s - s_0)/2$;
(* *Sestavimo opis.* *)
return $\mathit{levo}(s_0, t_0) + n_{08} \cdot \mathit{spodaj}(0, 8) + n_{20} \cdot \mathit{spodaj}(2, 0)$
  $+$ `DD` $+ n_{20} \cdot \mathit{zgoraj}(2, 0) + n_{08} \cdot \mathit{zgoraj}(0, 8)$;

Pri sestavljanju opisa si moramo predstavljati, da $a + b$ pomeni stik nizov a in b, zapis $a \cdot b$ pa stakne skupaj a izvodov niza b. Tabele nizov *levo*, *spodaj* in *zgoraj* pa lahko odčitamo s pogledom na sliko z začetka te rešitve:

s	t	$levo(s,t)$
0	4	LL
2	6	DLLNLN
2	8	DLLDLLNN
2	10	DLDLLNLDLN
0	16	DLDLLDLLDLDLLD

s	t	$spodaj(s,t)$
2	0	N
0	8	DLLD

Tabele *zgoraj* nismo posebej pisali, saj se izkaže, da je popolnoma enaka tabeli *spodaj*.

(c) Če bi se za vsako ploščico tipa L v našem zaporedju odločili, ali jo bomo uporabili kot ovinek levo ali ovinek desno, bi dobili tak opis poti, kot smo ga imeli pri podnalogi (a), in bi lahko njegovo veljavnost preverili po enakem postopku kot tam. Ker imamo v našem zaporedju t ploščic tipa L in ker sta pri vsaki dve možnosti, kako jo obrnemo (kot ovinek levo ali ovinek desno), je mogoče iz tega zaporedja konkretno pot sestaviti na 2^t načinov. Vse te možnosti lahko preizkusimo z rekurzijo; ob tem se premikamo po zaporedju, sproti računamo trenutni položaj in preverjamo, če bi se trenutna ploščica prekrivala s kakšno od predhodnih (v primeru prekrivanja nam s trenutno vejo rekurzije ni treba nadaljevati, saj že vemo, da bo ta pot neveljavna); pri vsakem ovinku izvedemo dva rekurzivna klica, ki preizkusita obe možnosti (ali je ovinek levi ali desni); če pa z rekurzijo pridemo do konca zaporedja, ne da bi opazili kakšno prekrivanje, moramo le še preveriti, če je pot zdaj sklenjena (torej če smo prišli nazaj v začetno točko). Tak postopek bo sčasoma našel vse možne veljavne poti.

Opazimo lahko še, da se naloga nič ne spremeni, če vhodno zaporedje ploščic ciklično zamaknemo (na primer iz `ILLILL` v `LLILLI`), kajti če je mogoče (s primernim izborom ovinkov) sestaviti sklenjeno pot pred takim zamikom, je prav taka sklenjena pot veljavna tudi po zamiku, le da bi jo morali začeti gledati pri neki drugi ploščici kot prej. To pa pomeni, da lahko vhodno zaporedje vedno zamaknemo tako, da se začne s ploščico tipa L; naj bo zdaj t število ploščic tipa L in za vsak i naj bo s_i število ploščic tipa I med i-to in $(i+1)$-vo ploščico tipa L.

Zapišimo zdaj našo rešitev s psevdokodo. Na vsakem nivoju rekurzije počnemo zelo podobne stvari kot v rešitvi podnaloge (a); podobno kot tam smo predpostavili, da so ploščice kvadrati s stranico 2 namesto 1. Obe možnosti glede usmeritve trenutnega ovinka (levo in desno) preizkusimo v zanki (korak 3). Upoštevati moramo še skupino s_i ploščic tipa I, ki sledijo trenutnemu ovinku (ploščici tipa L). Če ne zaznamo nobenega prekrivanja, lahko z rekurzijo nadaljujemo (korak 7); če pa smo že na koncu vhodnega zaporedja ($i = t$), moramo le še preveriti, če je pot zdaj sklenjena.

podprogram PREGLEJPOTI(x, y, s, i, S, *ovinki*):
vhodni podatki: trenutni položaj (x, y) in smer s; $i \in \{1, \ldots, n\}$ pove, pri kateri ploščici tipa L smo trenutno; S je množica doslej pokritih celic mreže; *ovinki* je tabela, v kateri element *ovinki*[k] pove, v katero smer gre k-ti ovinek.

1 (* *Stopimo v središče trenutne ploščice in preverimo prekrivanje.* *)
$x := x + dx[s]$; $y := y + dy[s]$;
if $(x, y) \in S$ **then return false else** dodaj (x, y) v množico S;

```
2   (* Preizkusimo obe možni smeri ovinka. *)
    for ovinki[i] ∈ {L, D}:
3     (* Izračunajmo novo smer. *)
      if ovinki[i] = L then s' := (s + 1) mod 4 else s' := (s + 3) mod 4;
4     (* Preverimo, ali pride do prekrivanja pri ploščicah tipa I. *)
      j := 1; while j ≤ s_i:
        x' := x + (2j + 1)dx[s']; y' := y + (2j + 1)dy[s'];
        if (x', y') ∈ S then break else j := j + 1;
      if j ≤ s_i then continue;
5     (* Označimo na novo pokrite ploščice. *)
      for j := 1 to s_i:
        dodaj (x + (2j + 1)dx[s'], y + (2j + 1)dy[s']) v množico S;
6     (* Izračunajmo položaj ob izstopu iz zadnje ploščice. *)
      x' := x + (2s_i + 1)dx[s']; y' := y + (2s_i + 1)dy[s'];
7     (* Nadaljujmo z rekurzijo oz. izpišimo rešitve. *)
      if i < t then PreglejPoti(x', y', s', i + 1, S', ovinki)
      else if x = x_0 and y = y_0 then izpiši tabelo ovinki;
8     (* Pobrišimo, kar smo dodali v koraku 5. *)
      for j := 1 to s_i:
        pobriši (x + (2j + 1)dx[s'], y + (2j + 1)dy[s']) iz množice S;
```

Začetni položaj (x_0, y_0) in smer s si lahko izberemo poljubno; tabela *ovinki* mora imeti dovolj prostora za vseh t ovinkov, drugače pa nam je ni treba posebej inicializirati, saj jo bo naš rekurzivni postopek sproti zapolnil s podatki o trenutnem stanju ovinkov. Glavni del programa bi moral potem vse skupaj pognati s klicem PreglejPoti$(x_0, y_0, s, 1, \{\}, ovinki)$. Izpis v koraku 7 bi lahko še izboljšali; trenutno izpiše le tabelo *ovinki*, ki pove, kateri ovinki so levi in kateri desni; lahko bi po vsakem od teh ovinkov izpisali še zaporedje s_i korakov naravnost naprej in tako dobili res pravi opis najdene poti.

(*d*) Ko smo se v dveh dimenzijah sprehajali vzdolž naše črte, sta bili pri vsaki ploščici tipa L dve možnosti: ali predstavlja ovinek v levo ali pa ovinek v desno. V treh dimenzijah pa so pri vsaki kockici tipa L štiri možnosti: poleg ovinka v levo in desno si lahko predstavljamo še ovinek navzgor in navzdol. Še en način, da si to predstavljamo, je naslednji: kockica tipa L ima šest ploskev; pri eni od njih je naša črta ravnokar vstopila vanjo; če bi kockico zapustila skozi nasprotno ploskev, bi bila to kockica tipa I, ne tipa L; tako ostanejo še štiri ploskve, skozi katere lahko naša črta zapusti kockico.

V treh dimenzijah to, kaj je za nas levo in desno (in kaj je gor in dol) ni odvisno le od tega, v katero smer gledamo; če se na primer zavrtimo okoli smeri, v katero gledamo, bomo še vedno gledali v isto smer kot prej, pač pa se bodo spremenile smeri levo, desno, gor in dol. Primer vidimo na naslednji sliki:

Zato je koristno med sprehajanjem po prostoru poleg našega položaja in smeri, v katero gledamo (na zgornji sliki je označena z „naprej“), vzdrževati še smer, ki si jo trenutno predstavljamo kot „gor“. S tem podatkom lahko potem enolično določimo, kaj za nas pomeni ovinek v levo, v desno, navzdol ali navzgor, in lahko tudi določimo novo smer naprej in novo smer gor. Teh stvari sicer ne moremo računati tako preprosto kot v dveh dimenzijah, kjer smo lahko predstavili smeri s števili od 0 do 3 in smo jih morali pri vsakem ovinku le povečati ali zmanjšati za 1 (po modulu 4).

Funkcijo, ki računa novo smer naprej in novo smer gor iz stare smeri naprej, stare smeri gor in tipa ovinka, bi lahko kar potabelirali; da pa nam ne bo treba ročno razmišljati o vseh teh možnih smereh in ovinkih, je lažje, če smeri računamo s pomočjo vektorjev. Smer naprej predstavimo z vektorjem $\mathbf{s}$, smer gor pa z vektorjem $\mathbf{g}$; oba naj bosta dolžine 1. V nadaljevanju si bomo pomagali z vektorskim produktom; spomnimo se, da je vektorski produkt dveh enotskih vektorjev $\mathbf{a} \times \mathbf{b}$ tudi sam enotski vektor, ki bo pravokoten na $\mathbf{a}$ in na $\mathbf{b}$, njegova smer pa je določena s pravilom desne roke: če s palcem desne roke pokažemo v smer $\mathbf{a}$, s kazalcem pa v $\mathbf{b}$ in iztegnemo sredinec, bo le-ta kazal v smer $\mathbf{a} \times \mathbf{b}$. S pomočjo gornje slike lahko vidimo, da če $\mathbf{s}$ kaže naprej in $\mathbf{g}$ kaže gor, potem lahko smer desno izrazimo kot $\mathbf{s} \times \mathbf{g}$, smer levo kot $\mathbf{g} \times \mathbf{s}$ (ali kot $-\mathbf{s} \times \mathbf{g}$), smer dol pa kot $-\mathbf{g}$.

Če sta $\mathbf{s}$ in $\mathbf{g}$ smeri naprej in gor na začetku ovinka, označimo s $\hat{\mathbf{s}}$ in $\hat{\mathbf{g}}$ novo smer naprej in gor na koncu ovinka. Zdaj lahko razmišljamo takole: če imamo ovinek levo, je smer naprej po novem tista, ki je bila prej levo; smer gor pa se ne spremeni: torej imamo $\hat{\mathbf{g}} = \mathbf{g}$ in $\hat{\mathbf{s}} = \mathbf{g} \times \mathbf{s}$. Podobno razmislimo še za ostale tipe ovinkov; pri ovinku desno dobimo $\hat{\mathbf{g}} = \mathbf{g}$ in $\hat{\mathbf{s}} = \mathbf{s} \times \mathbf{g}$; pri ovinku gor dobimo $\hat{\mathbf{g}} = -\mathbf{s}$ in $\hat{\mathbf{s}} = \mathbf{g}$; pri ovinku dol pa dobimo $\hat{\mathbf{g}} = \mathbf{s}$ in $\hat{\mathbf{s}} = -\mathbf{g}$.

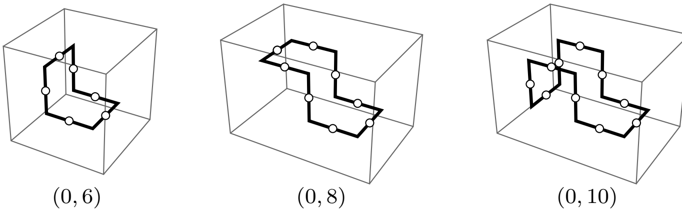
Potek naše poti lahko zdaj opišemo kot zaporedje znakov `N` (naprej), `L` (levo), `D` (desno), `G` (gor) in `S` (dol). Postopek, ki preverja, ali je tako opisana pot veljavna (sklenjena in brez prekrivanja), je zelo podoben tistemu iz rešitve podnaloge (*a*):

vhod: niz *opis*, ki ga tvori n znakov `I`, `L`, `D`, `G`, `S`;

1 $\mathbf{r}_0 := (0,0,0)$; $\mathbf{s} := (1,0,0)$; $\mathbf{g} := (0,1,0)$; (* *začetni položaj in smer* *)
  $\mathbf{r} := \mathbf{r}_0$; (* *postavimo se na začetni položaj* *)
  $S := \{\}$; (* *množica že pokritih celic mreže* *)
2 **for** $i := 1$ **to** n:
3   (* *Trenutno smo na robu neke celice; naredimo korak v smeri s, da* *)
    $\mathbf{r} := \mathbf{r} + \mathbf{s}$;   (* *pridemo v središče te celice.* *)
4   **if** $\mathbf{r} \in S$ **then return false**; (* *Smo to celico že kdaj prej obiskali?* *)
    else dodaj $\mathbf{r}$ v množico S; (* *Sicer si jo zapomnimo zdaj.* *)
5   (* *Če je trenutni korak ovinek, popravimo smer.* *)
    if *opis*[i] = `L` **then** $\mathbf{s} := \mathbf{g} \times \mathbf{s}$
    else if *opis*[i] = `D` **then** $\mathbf{s} := \mathbf{s} \times \mathbf{g}$
    else if *opis*[i] = `G` **then** $\mathbf{t} := \mathbf{s}$; $\mathbf{s} := \mathbf{g}$; $\mathbf{g} := -\mathbf{t}$
    else if *opis*[i] = `S` **then** $\mathbf{t} := \mathbf{s}$; $\mathbf{s} := -\mathbf{g}$; $\mathbf{g} := \mathbf{t}$;
6   (* *Naredimo korak naprej v novi smeri, da pridemo na rob celice.* *)
    $\mathbf{r} := \mathbf{r} + \mathbf{s}$;
7 (* *Na koncu še preverimo, če je črta sklenjena.* *)
  return $\mathbf{r} = \mathbf{r}_0$;

(e) Če s in t ustrezata pogojem, ki smo jih videli v rešitvi podnaloge (b) (torej da sta s in t oba soda; da je $t \geq 4$; in če je $s = 0$, potem je t večkratnik 4 in različen od 8), lahko nalogo rešimo na enak način kot tam. Vse kockice zložimo v eno plast naše 3-d kockaste mreže in jih obrnemo tako, da vse črte skozi te kocke ležijo v isti ravnini; od tu naprej lahko tretjo dimenzijo odmislimo, kockice obravnavamo kot kvadratne ploščice in rešujemo nalogo enako kot pri (b).

Izkaže pa se, da lahko v treh dimenzijah sestavimo sklenjeno pot tudi za nekatere take pare (s, t), za katere je v dveh dimenzijah ne moremo. Naslednja slika prikazuje rešitve za $s = 0$, $t = 6, 8, 10$ (ni jih težko najti ročno, lahko pa si tudi napišemo program, ki jih bo poiskal z rekurzijo):

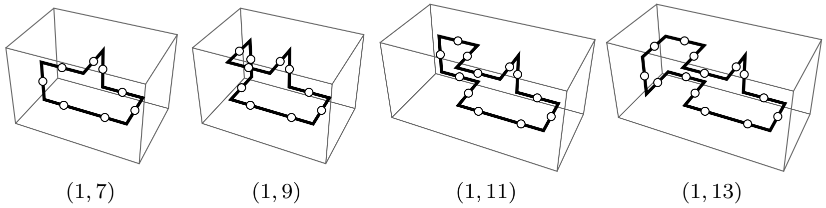


$(0, 6)$ $(0, 8)$ $(0, 10)$

(Zaradi preglednosti nismo narisali robov vseh posameznih kockic, pač pa le črto skoznje, s tanjšimi črtami pa še zunanje robove kvadra $2 \times 2 \times 2$ oz. $3 \times 2 \times 2$, ki ga tvorijo vse kockice skupaj. Krožci označujejo točke, kjer gre črta skozi mejo med dvema sosednjima kockicama.)

Vidimo lahko, da imajo vse te rešitve na vsaj enem koncu tudi par kockic ▯ v primernem položaju, da lahko pot tam prerežemo in vmes vrivamo dodatne razporede oblike $(2, 0)$ in $(0, 8)$, čisto tako kot pri dvodimenzionalni različici naloge. Tako lahko gornje rešitve za $s = 0$ in $t = 6, 8, 10$ razširimo še na $t = 14, 16, 18, 20, 22, 24$ in tako naprej. Skupaj s tem, kar smo rešili že pri (b), imamo zdaj pri $s = 0$ rešitve že za vse sode $t \geq 4$.

Spomnimo se, da so bili v dveh dimenzijah primeri z lihim s ali lihim t nerešljivi; v treh dimenzijah pa lahko nekatere vendarle rešimo. Naslednja slika kaže rešitve za $s = 1$ in $t = 7, 9, 11, 13$ (vidimo lahko, da jih je mogoče sestaviti precej sistematično: ko dobimo rešitev za $t = 7$, vanjo le dodajamo dodatne zavoje, vsak tak zavoj pa poveča število kockic tipa L za dve).



$(1, 7)$ $(1, 9)$ $(1, 11)$ $(1, 13)$

Tudi te poti imajo primeren par kockic ▯, pri katerem jih lahko razširimo z razporedi oblike $(2, 0)$ in $(0, 8)$; tako dobimo rešitve za vse (s, t), pri katerih sta s in t oba liha in je $t \geq 7$.

Zdaj ostajajo nerešeni le tisti (s, t), pri katerih velja kaj od naslednjega:

- s in t sta različne parnosti (eden je sod, drugi pa lih);

- s je sod in $t \in \{0, 2\}$;
- s je lih in $t \in \{1, 3, 5\}$.

Prepričajmo se, da so ti primeri res nerešljivi. Razmišljamo lahko podobno kot prej v dveh dimenzijah. Vzemimo poljubno veljavno sklenjeno pot, sestavljeno iz naših kockic, in se sprehodimo po njej; vsakič ko vstopimo v novo kockico (x, y, z), si kot stanje zapomnimo četverico $(x \bmod 2, y \bmod 2, z \bmod 2, d)$, pri čemer je d ena od šestih možnih smeri (za vsako koordinatno os po dve). Če poznamo stanje in vemo, za kakšen premik gre v naslednji kockici (korak naravnost naprej ali pa ovinek, pri čemer so ovinki zdaj štirje, v dveh dimenzijah pa sta bila le dva), lahko izračunamo naslednje stanje (ob vstopu v naslednjo kockico vzdolž naše poti). Vse to je torej zelo podobno kot prej v dveh dimenzijah, le da je stanj zdaj $2 \cdot 2 \cdot 2 \cdot 6 = 48$ (prej pa jih je bilo le 16) in iz vsakega gre po 5 premikov (prej pa le trije). Zato zdaj ne bomo poskušali eksplicitno zapisati vseh stanj in opisati vseh premikov med njimi, saj bi bilo tega že preveč; ni pa težko tega grafa stanj in prehodov med njimi zgenerirati (in kasneje preiskovati) z računalniškim programom. Zapišimo psevdokodo za postopek, ki izračuna vse možne prehode med stanji:

za vsak $x \in \{0, 1\}$, $y \in \{0, 1\}$, $z \in \{0, 1\}$, $d \in \{0, 1, 2, 3, 4, 5\}$:
  (* *d predstavlja smer, v katero smo bili obrnjeni ob vstopu v kockico* (x, y, z). *)
  za vsako možno izstopno smer $d' \in \{0, 1, 2, 3, 4, 5\}$:
    if je d' ravno nasprotna smer kot d **then**
      continue; (* *Obrat za* $180°$ *na mestu ni mogoč.* *)
    (* *Primer, ko je $d' = d$, predstavlja korak naravnost naprej, ostale možnosti za d' pa ovinke. Izračunajmo koordinate* (*po modulu 2*) *naslednje kockice na naši poti.* *)
    $x' := (x + dx[d']) \bmod 2$ in podobno za y' in z';
    dodaj v graf povezavo od (x, y, z, d) do (x', y', z', d')

Označimo množico vseh stanj z V, množico vseh prehodov med njimi (oz. povezav v našem grafu) pa z E.

Brez izgube za splošnost lahko rečemo, da se naša pot začne s stanjem $u_0 := (0, 0, 0, 0)$. Vse možne poti iz tega stanja lahko v mislih razdelimo na štiri skupine glede na parnost s-ja in t-ja (pri čemer je s število kockic tipa I na tej poti, t pa število kockic tipa L; oz. še drugače, s je število korakov naravnost naprej, t pa število ovinkov). Naj bo $D_{s',t'}$ množica tistih stanj, ki so dosegljiva iz u_0 s takšnimi potmi, pri katerih je $s' = s \bmod 2$ in $t' = t \bmod 2$. Te množice lahko poiščemo takole:

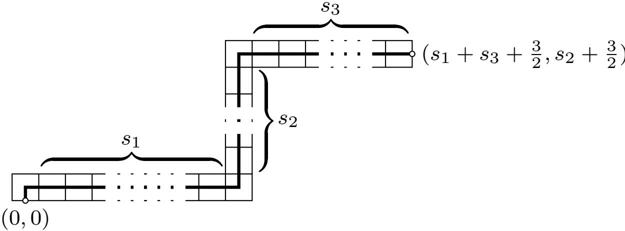
  $D_{00} := \{u_0\}$; $D_{01} := \{\}$; $D_{10} := \{\}$; $D_{11} := \{\}$;
  konec := **false**;
  while not konec:
    konec := **true**;
    for $s' := 0$ **to** 1 **do for** $t' := 0$ **to** 1:
      za vsako $u \in D_{s',t'}$, za vsako povezavo $u \to v$ iz E:
        if je ta povezava ovinek **then** $s'' := s'$; $t'' := (t' + 1) \bmod 2$
        else $s'' := (s' + 1) \bmod 2$; $t'' := t'$;
        if $v \in D_{s'',t''}$ **then continue**; (* *nič novega* *)
        dodaj v v $D_{s'',t''}$; *konec* := **false**;

Postopek torej v zanki pregleduje množice dosegljivih stanj in jih dopolnjuje: če je stanje u dosegljivo in obstaja povezava $u \to v$, potem je dosegljivo tudi stanje v, paziti moramo le na to, v katero od štirih množic $D_{s',t'}$ ga dodamo — če je bil prehod $u \to v$ ovinek, se z dodatkom tega koraka v našo pot poveča t (število ovinkov) za 1, torej se spremeni njegova parnost, medtem ko s ostane nespremenjen; če pa je bil prehod $u \to v$ korak naravnost naprej, se spremeni parnost s-ja (ker se s poveča za 1), t pa ostane nespremenjen. Ta postopek ponavljamo, dokler se množice dosegljivih stanj povečujejo; ko pa enkrat izvedemo prehod čeznje, ne da bi vanje še kaj dodali, lahko končamo, saj vemo, da se odtlej ne bodo več spreminjale.

Če poženemo ta postopek in na koncu pogledamo, katere množice $D_{s',t'}$ vsebujejo začetno stanje u_0, ga bomo našli v D_{00} in D_{11}, ne pa tudi v D_{01} in D_{10}. Tako torej vidimo, da primeri, ko je eden od s in t sod, drugi pa lih, res niso rešljivi — pri takih (s, t) poti na koncu ne bomo mogli skleniti (pripeljati nazaj v stanje, v katerem smo začeli).

Zdaj moramo razmisliti še, kaj se dogaja pri majhnih t (pri $t \in \{0, 2\}$ za sode s in pri $t \in \{1, 3, 5\}$ za lihe s). Vzemimo nek konkreten t in neko poljubno pot s tem številom kockic tipa L. Če je pot res sklenjena — in samo take nas konec koncev zanimajo — se lahko začnemo po njej sprehajati pri eni od teh kockic tipa L. (Brez izgube za splošnost si lahko tudi predstavljamo, da se ta sprehod začne v točki s koordinatami $(0, 0, 0)$ in da smo tedaj obrnjeni v smer $d = 0$.) Pot si lahko v tem primeru predstavljamo takole: najprej kockica tipa L, nato s_1 kockic tipa I, nato spet kockica tipa L, nato še s_2 kockic tipa I, ..., sčasoma še zadnja (t-ta) kockica tipa L in za njo še s_t kockic tipa I. Skupno število kockic tipa I je potem $s = s_1 + s_2 + \ldots + s_t$. Posamezni s_i so lahko tudi enaki 0 (ne morejo pa seveda biti manjši od 0).

Pri vsakem od t ovinkov imamo štiri možnosti glede tega, kako se pri tem ovinku spremeni naša smer (v dveh dimenzijah smo imeli samo dve možnosti, leve in desne ovinke; tu v treh dimenzijah pa imamo štiri). Skupaj imamo torej za pot kot celoto 4^t možnih oblik poti. Če si izberemo eno od teh konkretnih 4^t oblik poti, so zdaj koordinate konca poti odvisne le od števil $s_1, \ldots, s_t$. Primer (za $t = 3$) kaže naslednja slika, sicer v dveh dimenzijah namesto v treh; imamo pot s tremi ovinki (desno, levo in desno); če se pot začne v točki $(0, 0)$, se konča v točki $(s_1 + s_3 + \frac{3}{2}, s_2 + \frac{3}{2})$.



Če hočemo, da bo pot sklenjena, se mora končati v isti točki, kjer se je začela. V primeru na sliki bi to pomenilo, da mora biti $s_1 + s_3 + \frac{3}{2} = 0$ in $s_2 + \frac{3}{2} = 0$. Ker so števila s_1, s_2 in s_3 cela (in nenegativna), je jasno, da ti dve enačbi v nobenem primeru ne bosta izpolnjeni, torej s takšnim zaporedjem treh ovinkov ne bomo mogli sestaviti sklenjene poti (ne glede na to, koliko kockic tipa I uporabimo in kako jih razdelimo med ovinke).

Zdaj si lahko napišemo program, ki (za izbrani t) z rekurzijo preizkusi vseh možnih 4^t oblik poti in pri vsaki izračuna koordinate konca poti v odvisnosti od $s_1, \ldots, s_t$. Tako bo na primer x-koordinata konca poti vedno oblike $x_0 + x_1 s_1 + \ldots + x_t s_t$, pri čemer so $x_0, x_1, \ldots, x_t$ neke konstante, ki jih bo izračunal naš postopek (v primeru na gornji sliki smo imeli $x_0 = 3/2$, $x_1 = 1$, $x_2 = 0$ in $x_3 = 1$). Podobno je tudi pri y- in z-koordinatah.

podprogram Rekurzija($t' \in \{1, \ldots, t\}$, dosedanja smer $d \in \{0, \ldots, 5\}$):
  za vsako novo smer $d' \in \{0, \ldots, 5\}$:
    if sta d in d' enaki ali pa nasprotni smeri **then**
      continue; (* *to sploh ne bi bil ovinek* *)
    (* *Upoštevajmo premik, ki ga naredi črta po trenutni kockici tipa L.* *)
    $x_0 := x_0 + \frac{1}{2} dx[d] + \frac{1}{2} dx[d']$ in podobno za y in z;
    (* *Upoštevajmo premik po skupini $s_{t'}$ kockic tipa I, ki zdaj sledijo.* *)
    $x_{t'} := dx[d']$ in podobno za y in z;
    if $t' < t$ **then** Rekurzija($t' + 1$, d')
    else Preveri($x_0, \ldots, x_t, y_0, \ldots, y_t, z_0, \ldots, z_t$);

Glavni blok programa bi moral na začetku postaviti x_0, y_0 in z_0 na 0 in poklicati Rekurzija$(1, 0)$.

Podprogram Preveri mora zdaj preveriti, če je mogoče $s_1, \ldots, s_t$ izbrati tako, da bodo koordinate konca poti enake začetnim, torej da bo $x_0 + x_1 s_1 + \ldots + x_t s_t = 0$ in podobno za y in z. Če je na primer x_0 ne-celo število (kot npr. 3/2 v našem primeru zgoraj), je sistem enačb gotovo nerešljiv ($x_1, \ldots, x_t$ so namreč zagotovo celi. Podobno, če je $x_0 > 0$ in če so vsi $x_1, \ldots, x_t$ nenegativni, je sistem nerešljiv (saj so tudi $s_1, \ldots, s_t$ nenegativni, torej bo vsota $x_0 + x_1 s_1 + \ldots + x_t s_t$ strogo večja od 0, mi pa bi radi, da bi bila enaka 0). Podobno je sistem nerešljiv tudi, če je $x_0 < 0$ in so vsi $x_1, \ldots, x_t$ manjši ali enaki 0. Enak razmislek lahko ponovimo tudi za y in z.

Če na ta način preverimo vse možne poti za $t = 0, 1, 2, 3, 5$, bomo pri vseh že na podlagi dosedanjih razmislekov videli, da je dobljeni sistem enačb nerešljiv, torej sklenjene poti s tistim zaporedjem ovinkov ni. Tako torej vidimo, da so pri teh majhnih t res nerešljivi vsi (s, t), ne glede na to, kakšen s vzamemo.

(*f*) Ta podnaloga se pravzaprav v ničemer bistveno ne razlikuje od podnaloge (*c*). Vse, kar moramo narediti, je, da rešitev podnaloge (*c*) prilagodimo tako, da bo delovala v treh dimenzijah. Pri tem se lahko zgledujemo po naši rešitvi podnaloge (*d*). Naš trenutni položaj bo zdaj trojica $\mathbf{r} = (x, y, z)$, tudi trenutno smer naprej predstavimo zdaj z vektorjem $\mathbf{s}$, poleg nje pa bomo ob rekurzivnih klicih prenašali še smer gor (vektor $\mathbf{g}$). V vsakem rekurzivnem klicu moramo zdaj z zanko preizkusiti vse štiri možnosti glede smeri trenutnega ovinka (poleg levo in desno še gor in dol). Tako dobimo naslednji postopek:

podprogram PreglejPoti($\mathbf{r}$, $\mathbf{s}$, $\mathbf{g}$, i, S, *ovinki*):
1 $\mathbf{r} := \mathbf{r} + \mathbf{s}$; **if** $\mathbf{r} \in S$ **then return false else** dodaj $\mathbf{r}$ v množico S;
2 **for** *ovinki*[i] $\in \{\mathtt{L}, \mathtt{D}, \mathtt{G}, \mathtt{S}\}$:
3   izračunaj novi smeri $\hat{\mathbf{s}}$ in $\hat{\mathbf{g}}$ na enak način kot v rešitvi podnaloge (*d*);
4    $j := 1$; **while** $j \le s_i$:
      if $\mathbf{r} + (2j + 1)\hat{\mathbf{s}} \in S$ **then break else** $j := j + 1$;

```
        if j ≤ s_i then continue;
5       for j := 1 to s_i do dodaj r + (2j + 1)ŝ v S;
6       r̂ := r + (2s_i + 1)ŝ;
7       if i < t then PreglejPoti(r̂, ŝ, ĝ, i + 1, S, ovinki)
        else if r = r_0 then izpiši tabelo ovinki;
8       for j := 1 to s_i do pobriši r + (2j + 1)ŝ iz S;
```

Začetni položaj in smer si lahko izberemo poljubno, paziti moramo le na to, da bo $\mathbf{g}$ pravokotna na $\mathbf{s}$. Vzamemo lahko na primer enako kot v naši rešitvi podnaloge (*d*): $\mathbf{r}_0 := (0, 0, 0)$, $\mathbf{s} := (1, 0, 0)$ in $\mathbf{g} := (0, 1, 0)$; glavni del programa lahko zdaj rekurzijo začne s klicem PreglejPoti$(\mathbf{r}_0, \mathbf{s}, \mathbf{g}, 1, \{\}, \mathit{ovinki})$.

Naloge so sestavili: šestkotna mreža, kocke — Nino Bašić; statistika na besedilu, letala — Boris Gašperin; botanika, volitve — Tomaž Hočevar; miselni vzorec, pranje denarja — Jurij Kodre; štetje nizov — Matjaž Leonardis; problematične formule — Jure Slak; naključni vzorec — Mitja Trampuš; simbolične povezave, oklepajski izrazi, generator naključnih števil — 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. Lov na sataniste

- Mišljeno je, da učinkovita rešitev pri tej nalogi porabi $O(n)$ časa; rešitve, ki porabijo na primer $O(n^2)$ ali $O(n \cdot p)$ časa (ker gredo pri vsaki pojavitvi kakšne šestice še enkrat po celem nizu in iščejo naslednji dve ali prejšnji dve pojavitvi ipd.) naj dobijo (če so sicer pravilne) največ 12 točk.
- Vseeno je, kako je v rešitvi predstavljen niz s, saj naloga tega ne predpisuje natančno (v C/C++ je lahko na primer **char***, **const char***, string ipd.).
- Ravno tako tudi ni samo po sebi nič narobe, če si rešitev pri iskanju pojavitev znaka 6 in niza six pomaga s funkcijami iz standardne knjižnice svojega programskega jezika (namesto da bi sama pregledovala posamezne znake niza s, kot to na primer počne naša rešitev).
- Naloga ne pričakuje, da bi rešitev poleg niza six odreagirala še na kakšne različice tega niza, v katerih se namesto malih črk pojavljajo velike (na primer SIX, Six ipd.). Če kakšna rešitev vendarle odreagira tudi na takšne različice niza six, naj se ji zaradi tega ne odšteva točk.
- Če rešitev pri preverjanju, ali se tri šestice pojavijo znotraj podniza dolžine p, šteje le do začetka zadnje šestice namesto do konca (kot če bi npr. v funkciji iz naše rešitve uporabili pogoj c − a + 1 namesto k − a + 1), naj se ji zaradi tega odbije tri točke.

2. Hišna številka

- Poudarek pri tej nalogi je, da je rešitev pravilna, ne pa toliko na tem, da je učinkovita. (Poleg postopka, opisanega v naši rešitvi, obstaja sicer za to nalogo še veliko učinkovitejša rešitev, ki porabi le $O(\log n)$ časa, vendar ni mišljeno, da bi se tekmovalci domislili take rešitve.)
- Mogoče bo kakšnemu od tekmovalcev prišlo na misel, da je primernih hišnih številk pravzaprav malo in da bi si jih lahko izračunal vnaprej, jih hranil v neki tabeli (ali kakšni drugi primerni podatkovni strukturi) in potem v funkciji Naslednja le pregledal to tabelo. Tudi to je čisto dobra rešitev in lahko dobi vse točke, če sicer vrača pravilne rezultate (in če vsebuje izvorno kodo, ki takšno tabelo dejansko pripravi). Ni pa zahtevano, da bi rešitev počela kaj takega; popolnoma dovolj je že, če pri vsakem klicu funkcije Naslednja po vrsti pregleduje številke od $n + 1$ naprej, dokler ne najde naslednje primerne.
- Čisto sprejemljivo je tudi, če si rešitev pri pretvarjanju števila v števke pomaga z nizi in z morebitnimi funkcijami za pretvorbo števil v nize iz standardne knjižnice.
- Naloga pravi, da gre lahko n do največ 10^6. Če bi rešitev zaradi kakšnih nespametnih omejitev pri velikosti kakšne tabele ali česa podobnega delovala le do $n = 999\,999$, ne pa tudi za $n = 1\,000\,000$, ali pa celo, če bi delovala le do $n = 999\,665$ (kar je največji n, pri katerem je rezultat še 6-mestno število namesto 7-mestno), naj se ji zaradi tega odbije največ tri točke.

3. Stolpnica

- 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.
- Mišljeno je, naj ima rešitev časovno zahtevnost $O(w \cdot h)$. Rešitve, ki so bolj neučinkovite od tega, na primer $O(w^2 \cdot h)$, lahko dobijo pri tej nalogi največ 15 točk (če so sicer pravilne). Rešitve z eksponentno časovno zahtevnostjo (npr. zaradi kakšne nespametne rekurzije) lahko dobijo pri tej nalogi največ 10 točk (če so sicer pravilne).
- Glede prostorske zahtevnosti naj velja katera koli zahtevnost do vključno $O(w \cdot h)$ za enako dobro. V naših rešitvah imamo dve različici rešitve, eno s pomožno tabelo `podprta` dolžine $O(w)$ in eno brez take tabele (torej s prostorsko zahtevnostjo $O(1)$); oboje je enako dobro, prav tako dobra bi bila tudi tabela velikosti $O(w \cdot h)$.
- Za razne drobne napake pri delu s tabelo `T` (npr. če rešitev zamenja vrstni red indeksov x in y ali pa če šteje indekse od 1 naprej namesto od 0 naprej) naj se odbije največ dve točki.
- Če si rešitev razlaga tabelo `T` tako, da v njej namesto logičnih vrednosti (**true** in **false** pričakuje kaj drugega (npr. števila 0 in 1, znake `'#'` in `'.'` ipd.), naj se ji zaradi tega ne odšteva točk.
- Rešitve, ki prenašajo podprtost le navzgor, ne pa tudi levo in desno, lahko dobijo največ 8 točk.

4. Kontrolne naloge

- Pri tej nalogi je poudarek bolj na pravilnosti rešitve kot na njeni učinkovitosti. Rešitev naj bi se zavedala predvsem naslednjega: (1) različnih nalog istega predmeta ne ločimo med sabo, ravno tako ne ločimo dobrih dijakov med sabo, niti ne slabih dijakov med sabo; (2) naloge je treba razdeliti tako, da jih dobri dijaki rešijo čim več in to lahko dosežemo tako, da delimo skupine nalog od večjih proti manjšim, dokler ne zmanjka bodisi nalog bodisi dobrih dijakov.
- Naloga zahteva le, da tekmovalčeva rešitev preveri, ali primeren razpored nalog med dijake obstaja, ni pa treba, da tak konkreten razpored tudi najde ali izpiše (tudi postopek v naši rešitvi tega ne stori).
- V naši rešitvi smo podrobno razložili tudi to, kako učinkovito implementirati razdeljevanje skupin nalog med dobre dijake (opisali smo celo dve različici, eno s časovno zahtevnostjo $O(p \log p)$ in eno z zahtevnostjo $O(n)$). Če tekmovalčeva rešitev nič ne pove o tem, kako bi v praksi izračunala rezultat (pač pa npr. le na splošno reče, da dobrim dijakom razdeli naloge tako, da jih rešijo čim več), naj dobi največ 12 točk (če je drugače pravilna). Rešitev, ki za razdeljevanje nalog med dijake porabi eksponentno mnogo časa, naj dobi največ 15 točk; rešitev, ki za razdeljevanje nalog med dobre dijake porabi $O(d \cdot t \cdot p)$

časa (za vsakega dijaka in vsak dan naredi zanko po vseh predmetih), naj dobi največ 18 točk.

5. Tehtnica

- Rešitev, ki ne upošteva tega, da lahko uteži dajemo tudi v levo posodo in ne le v desno, naj dobi največ 8 točk (če je v okviru te sicer neupravičene omejitve pravilna, torej sestavi maso n z najmanjšim možnim številom uteži v desni posodi).

- Od dobrih rešitev pričakujemo predvsem neko obliko rekurzivnega razmisleka, ki nam omogoča prevesti problem za n na približno pol manjše probleme. Rešitev, ki to rekurzijo potem izvede na neučinkovit način, ki rezultate nekaterih podproblemov računa po večkrat (kot na primer funkcija Tehtnica v naši rešitvi), naj dobi največ 18 točk (če je drugače pravilna).

- Rešitev, ki dejansko generira in preizkuša različne možne razporede uteži, npr. z rekurzijo in metodo „razveji in omeji“ (*branch and bound*), naj dobi največ 15 točk (če je drugače pravilna).

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. Lov na sataniste	lažja do srednja naloga v prvi skupini
2. Hišna številka	srednja v prvi ali lahka naloga v drugi skupini
3. Stolpnica	težka v prvi ali lažja naloga v drugi skupini
4. Kontrolne naloge	srednja do težja naloga v drugi skupini
5. Tehtnica	težja 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, letos pa so se rezultati izšli tako, da smo v drugi skupini izjemoma podelili tri tretje nagrade. Poleg nagrad na državnem tekmovanju v skladu s pravilnikom podeljujemo tudi zlata in srebrna priznanja. Število zlatih priznanj je omejeno na eno priznanje na vsakih 25 udeležencev šolskega tekmovanja (teh je bilo letos 306) in smo jih letos podelili dvanajst. 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 112 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	Blaž Zupančič	1	Škof. klas. gimn. Lj.	20	20	20	19	19	98
1Z	2	Martin Peterlin	2	Vegova Ljubljana	20	20	18	19	20	97
2Z	3	Aleš Ravnikar	4	STPŠ Trbovlje	20	20	18	19	17	94
2Z		Rok Krumpak	3	ŠC Celje, Gimn. Lava	20	20	18	19	17	94
3S	5	Amon Stopinšek	4	STPŠ Trbovlje	20	20	18	16	19	93
3S		Aljaž Mislovič	4	ŠC Ptuj, ERŠ	20	20	18	18	17	93
S	7	Boštjan Pintar	4	ŠC Kranj, STŠ	20	20	18	19	15	92
S	8	Matej Marinko	2	Gimnazija Vič	18	20	18	19	16	91
S	9	Vid Klopčič	2	Gimnazija Vič	20	20	18	20	12	90
S		Timotej Knez	3	Škof. klas. gimn. Lj.	20	20	13	18	19	90
S		David Pintarič	3	SPTŠ Murska Sobota	16	19	20	18	17	90
S	12	Andraž Jelenc	4	Gimnazija Škofja Loka	20	15	19	17	18	89
S	13	Mitja Žalik	2	II. gimnazija Maribor	18	20	18	19	13	88
S	14	Arthur-Louis Heath	3	Gimnazija Bežigrad	20	20	17	18	12	87
S	15	Urban Duh	1	II. gimnazija Maribor	15	20	13	19	16	83
S	16	Vid Drobnič	2	Gimnazija Vič	15	20	12	19	16	82
S		Žiga Patačko Koderman	2	Gimnazija Vič	17	20	10	16	19	82
S	18	Zen Lednik	3	ŠC Celje, SŠ za KER	20	20	8	19	14	81
S		Jakob Bambič	4	ŠC Novo mesto, SEŠTG	19	20	8	18	16	81
S	20	Žiga Vene	3	ŠC Krško-Sevnica	3	20	18	18	20	79
S	21	Tomaž Jerman	3	STPŠ Trbovlje	15	20	15	15	13	78
S	22	Luka Pogačnik	4	Gimnazija Vič	10	10	16	20	20	76
S		Matic Gačar	3	SERŠ Maribor	18	20	17	17	4	76
S	24	Karin Frlic	4	Škof. klas. gimn. Lj.	5	20	18	20	12	75
S	25	Jure Hudoklin	3	Gimnazija Bežigrad	16	20	12	16	10	74
S	26	Tomaž Martinčič	3	STPŠ Trbovlje	5	20	16	20	11	72
S		Matic Rašl	2	ŠC Ptuj, ERŠ	19	18	17	18	0	72
S		Luka Govedič	1	II. gimnazija Maribor	0	20	17	20	15	72
S	29	Jakob Vokač	4	II. gimnazija Maribor	20	20	16	15	0	71
S	30	David Popović	2	Gimnazija Bežigrad	20	20	7	10	12	69
S		Simon Tušar	4	Vegova Ljubljana	5	20	16	19	9	69
S		Gregor Štefanič	3	SERŠ Maribor	14	5	16	18	16	69
S		Primož Hrovat	4	ŠC Novo mesto, SEŠTG	20	20	0	19	10	69
S	34	Goran Tubić	4	Vegova Ljubljana	13	20	0	15	18	66
S	35	Jure Bevc	3	ŠC Krško-Sevnica	20	20	10	12	2	64
S	36	Klemen Gumzej	4	ŠC Celje, SŠ za KER	3	20	12	18	10	63
S		Martin Dagarin	1	Vegova Ljubljana	14	5	8	17	19	63
S		Leon Pahole	4	SERŠ Maribor	5	20	16	18	4	63
S	39	Žiga Klemenčič	2	Vegova Ljubljana	20	1	12	14	14	61
S		Jaka Jenko	3	ERŠ Velenje	13	12	7	16	13	61

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
S	41	Marko Laharnar	4	STPŠ Trbovlje	1	20	14	19	6	60
S		Jaka Pelaič	3	Gimnazija Škofja Loka	16	8	18	18	0	60
S		Tadej Šinko	2	SPTŠ Murska Sobota	4	8	12	20	16	60
S	44	Gregor Štefanič	3	ŠC Novo mesto, SEŠTG	20	0	13	18	8	59
S	45	Rok Hudobivnik	4	ŠC Kranj, STŠ	17	5	17	18	0	57
S		Marko Korasa	3	ŠC Novo mesto, SEŠTG	17	20	0	19	1	57
S		Tim Simičak Hafner	4	Gimnazija Kranj	14	15	5	19	4	57
S		Benjamin Kraner	3	II. gimnazija Maribor	5	3	16	17	16	57
S	49	Marko Hostnik	1	Gimnazija Bežigrad	2	20	9	20	5	56
S		Marko Drvarič	4	Gimn. Murska Sobota	19	15	5	7	10	56
S	51	Jernej Leskovšek	2	STPŠ Trbovlje	5	20	12	18	0	55
S	52	Tomaž Hribernik	2	Gimnazija Kranj	18	7	8	14	5	52
S		Jure Vreček	3	ŠC Kranj, STŠ	3	10	10	18	11	52
S		Nino Serec	4	Gim. Murska Sobota	20	3	12	16	1	52
S	55	Franci Šacer	4	ŠC Celje, SŠ za KER	5	12	2	19	13	51
S		Amadej Kristjan Kocbek	4	II. gimnazija Maribor	20	5	10	16	0	51
	57	Aljaž Žel	1	II. gimnazija Maribor	20	14	1	12	3	50
		Matic Šincek	3	ERŠ Velenje	0	20	12	18	0	50
	59	Urban Kocmut	4	ŠC Celje, Gimn. Lava	7	20	5	0	17	49
	60	Jakob Jesenko	1	Škof. klas. gimn. Lj.	5	20	3	19	0	47
	61	Dani Cerovac	3	STŠ Koper	15	20	10	0	0	45
		Marko Kužner	2	SERŠ Maribor	5	3	10	18	9	45
		Rok Friš	4	II. gimnazija Maribor	5	10	12	18	0	45
		Matej Pevec	9	OŠ Veliki Gaber	15	20	10	0	0	45
	65	Miha Černe	3	Gimnazija Vič	5	5	12	19	3	44
		Domen Dolanc	3	STPŠ Trbovlje	5	20	0	19	0	44
		Peter Bohanec	3	Gim. Bežigrad + ZRI	10	19	6	9	0	44
		Luka Jevšenak	1	Gimnazija Velenje	0	19	14	5	6	44
		Andreja Merše	3	Škof. klas. gimn. Lj.	20	8	8	7	1	44
	70	Mitja Celec	4	SPTŠ Murska Sobota	16	0	7	18	0	41
		Luka Zorko	4	ŠC Novo mesto	5	1	10	18	7	41
		Daniel Vitas	2	Gimnazija Vič	13	20	2	4	2	41
		Gregor Ažbe	4	ŠC Kranj, STŠ	5	20	16	0	0	41
	74	Sebastian Mežnar	3	STŠ Koper	5	20	2	12	1	40
	75	Domen Leš	1	II. gimnazija Maribor	5	13	5	16	0	39
	76	Tilen Merše	3	SŠ Domžale	13	7	5	10	3	38
		Luka Dragar	1	Vegova Ljubljana	5	8	5	18	2	38
	78	Vid Jerovšek	3	ŠC Novo mesto, SEŠTG	16	5	2	14	0	37
		Rok Kovač	2	Gimnazija Vič	15	4	18	0	0	37
	80	Jaka Basej	1	Vegova Ljubljana + ZRI	5	18	3	10	0	36

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
	81	Nejc Hirci	2	Gimnazija Vič	5	20	8	0	0	33
		Luka Miklavčič	2	Gimnazija Vič	15	17	0	1	0	33
	83	Tim Jeric	9	OŠ Trnovo	0	17	0	8	7	32
		Katja Logar	3	Škof. klas. gimn. Lj.	0	1	12	16	3	32
	85	Neža Jesenko	4	Škof. klas. gimn. Lj.	3	7	17	3	1	31
	86	Katja Pivec	4	ŠC Ptuj, ERŠ	5	5	3	15	1	29
		Klemen Nemec	4	SPTŠ Murska Sobota	0	12	5	12	0	29
	88	Jan Zaletel	3	SŠJJ Ivančna Gorica	5	5	8	0	8	26
	89	Žiga Udovič	3	SŠ Domžale	5	10	0	10	0	25
	90	Aljaž Zakošek	2	I. gimnazija Celje	7	17	0	0	0	24
		Peter Gladek	3	Gimnazija Vič	2	20	2	0	0	24
	92	Miha Gjura	1	Gimnazija Vič	3	7	8	2	2	22
		Matic Dokl	2	II. gimnazija Maribor	12	5	3	1	1	22
	94	Alen Pungertnik	1	Gimnazija Vič	15	1	3	2	0	21
	95	Jure Cvetko	4	SPTŠ Murska Sobota	14	0	5	0	0	19
	96	Jan Biteżnik	1	Škof. klas. gimn. Lj.	3	2	8	5	0	18
		David Rozman	1	ZRI	3	3	12	0	0	18
	98	Aleš Zaplatil	2	Gimnazija Vič	5	0	5	5	0	15
	99	Jože Gašperlin	1	Gimnazija Kranj	3	1	8	2	0	14
		Jon Galonja	2	Gimnazija Vič	5	0	0	9	0	14
	101	Denis Perčič	2	ŠC Krško-Sevnica	0	2	6	3	0	11
	102	Blaž Novak	1	ŠC Ravne na Koroškem	10	0	0	0	0	10
	103	Eva Oblak	1	Gimnazija Vič	0	0	8	0	0	8
		Žan Magerl	2	Gimnazija Bežigrad	3	0	5	0	0	8
	105	Julijana Djordjević	2	SŠ Domžale	2	3	2	0	0	7
		Peter Knez	2	Gimnazija Vič	4	3	0	0	0	7
	107	Rok Kovačič	3	SŠ Domžale	4	1	0	0	0	5
	108	Tomaž Klopčič	3	SŠ Domžale	1	0	3	0	0	4
	109	Urban Matko	2	SŠ Domžale	2	0	1	0	0	3
	110	Žiga Avbreht	2	Gimnazija Vič	0	1	0	0	0	1
		Andraž Rojc	2	Gim. in sr. šola R. Maistra Kamnik	0	0	0	1	0	1
		Miha Breznik	1	ŠC Ravne na Koroškem	1	0	0	0	0	1
	113	Tilen Teodorovič	2	ŠC Krško-Sevnica	0	0	0	0	0	0

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1Z	1	Nejc Kadivnik	4	ŠC Kranj, Str. gim.	18	18	13	13	18	80
1Z	2	Jaka Mohorko	4	II. gimnazija Maribor	18	14	12	17	17	78
2Z	3	Leon Gorjup	4	SERŠ Maribor	8	13	15	17	17	70
2Z	4	Nejc Prikeržnik	4	ŠC Ravne na Koroškem	15	9	8	20	16	68
3S	5	Metod Medja	4	ŠC Kranj, STŠ	17	17	8	3	20	65
3S		Matej Tomc	4	Škof. klas. gimn. Lj.	18	11	5	12	19	65
3S	7	Jakob Erzar	4	Gimnazija Kranj	6	15	7	17	19	64
S	8	Klemen Kogovšek	4	Vegova Ljubljana	0	8	17	17	18	60
S	9	Tevž Murkovič	3	Vegova Ljubljana + ZRI	6	15	5	14	18	58
S	10	Andraž Juvan	3	Vegova Ljubljana	14	12	12	0	19	57
	11	Jaka Kordež	4	ŠC Kranj, STŠ	13	15	8	2	18	56
		Tadej Plos	4	III. gimnazija Maribor	3	17	10	10	16	56
		Klemen Pevec	3	Vegova Ljubljana	15	5	3	15	18	56
	14	Rok Šeško	4	II. gimnazija Maribor	0	16	11	14	13	54
	15	Boštjan Kloboves	2	ZRI	16	16	3	0	17	52
	16	Benjamin Benčina	3	Gim. Bežigrad + ZRI	5	12	12	13	7	49
	17	Tim Poštuvan	1	ZRI	16	13	1	0	18	48
	18	Gregor Rihtaršič	2	ZRI	3	19	5	8	4	39
	19	Žan Vidrih	3	SERŠ Maribor	4	6	1	10	15	36
	20	Vid Trtnik	4	Gimnazija Poljane	0	1	3	17	13	34
	21	Enej Ravbar	4	ŠC Nova Gorica, GZŠ	0	18	6	0	6	30
	22	Miha Jamšek	4	ŠC Nova Gorica, GZS	3	11	3	6	6	29
	23	Jan Pelicon	4	ŠC Nova Gorica, GZŠ	3	0	10	0	1	14
	24	Robi Novak	4	SERŠ Maribor	0	3	5	0	0	8
	25	Jure Pavlin	3	SŠ Domžale	2	1	2	0	0	5

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Žiga Željko	2	Gimnazija Bežigrad	100	100	94	100	100	494
1Z	2	Aleksej Jurca	2	Gimnazija Bežigrad		100	70		64	234
2Z	3	Aljaž Eržen	3	ZRI	88	97	20	0		205
2Z	4	Samo Remec	4	Vegova Ljubljana	34	31	90	0	40	195
3S	5	Miha Štravs	4	Gimnazija Poljane		97	47	15	9	168
3S	6	Matevž Poljanc	4	Škof. klas. gimn. Lj.	94	70		0		164
S	7	Rok Kos	3	Gimnazija Vič	97	0	17	0		114
	8	Sandi Režonja	4	Gim. Murska Sobota	16	67	20			103
	9	Dan Toškan	3	STŠ Koper		100	0	0		100
	10	Bor Brecelj	3	ZRI		51	44			95
	11	Matej Mulej	3	ŠC Kranj, STŠ		74		0		74
	12	Tadej Gašparovič	8	OŠ J. Krajca Rakek	0		0		25	25
	13	Jakob Gaberc Artenjak	4	ŠC Ptuj, ERŠ		0	7		0	7
	14	Žan Knafelc	4	ZRI		0			2	2
	15	Peter Urbanč	3	ŠC Krško-Sevnica	0				1	1

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Blaž Zupančič	tablični računalnik Asus MeMO Pad HD 7 8 GB
1	1	Martin Peterlin	tablični računalnik Asus MeMO Pad ME70CX
1	2	Aleš Ravnikar	3 TB zunanji disk
1	2	Rok Krumpak	3 TB zunanji disk
1	3	Aljaž Mislovič	2 TB flash disk
1	3	Amon Stopinšek	miška Razer Naga 2014
2	1	Nejc Kadivnik	tablični računalnik Asus MeMO Pad HD 7 8 GB
			Raspberry Pi model B
			Dasgupta *et al.: Algorithms*
2	1	Jaka Mohorko	tablični računalnik Asus MeMO Pad ME70CX
			Dasgupta *et al.: Algorithms*
2	2	Leon Gorjup	2 TB zunanji disk
			Dasgupta *et al.: Algorithms*
2	2	Nejc Prikeržnik	2 TB zunanji disk
2	3	Metod Medja	miška Razer Naga 2014
2	3	Matej Tomc	miška Razer Naga 2014
2	3	Jakob Erzar	miška Razer Naga 2014
3	1	Žiga Željko	tablični računalnik Asus MeMO Pad HD 7 8 GB
			Raspberry Pi model B
			Cormen *et al.: Introduction to algorithms*
3	1	Aleksej Jurca	2 TB zunanji disk
			Cormen *et al.: Introduction to algorithms*
3	2	Aljaž Eržen	tablični računalnik Asus MeMO Pad ME70CX
			Raspberry Pi model B
			Cormen *et al.: Introduction to algorithms*
3	2	Samo Remec	2 TB zunanji disk
			64 GB USB ključ
3	3	Miha Štravs	2 TB zunanji disk
3	3	Matevž Poljanc	64 GB USB ključ
Off-line naloga — Zlaganje likov			
	1	Tomaž Hočevar	Raspberry Pi model B
	2	Rok Kralj	Raspberry Pi model B

SODELUJOČE ŠOLE IN MENTORJI

Šola	Mentorji
II. gimnazija Maribor	Mirko Pešec
Gimnazija Bežigrad	Andrej Šuštaršič, Jurij Železnik
Gimnazija in srednja šola Rudolfa Maistra Kamnik	Janez Klemenčič
Gimnazija Kranj	Zdenka Vrbinc
Gimnazija Murska Sobota	Valerija Režonja, Romana Vogrinčič
Gimnazija Poljane	Janez Malovrh, Boštjan Žnidaršič
Gimnazija Škofja Loka	Anže Nunar
Gimnazija Vič	Klemen Bajec, Marjan Greselj, Nataša Mori, Marina Trost
Osnovna šola Jožeta Krajca Rakek	Lidija Anzeljc
Osnovna šola Trnovo	
Osnovna šola Veliki Gaber	Matjaž Lavrih
I. gimnazija v Celju	Simona Magdalenc Lindner
Srednja elektro-računalniška šola Maribor (SERŠ)	Vida Motaln, Slavko Nekrep, Manja Sovič Potisk
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Igor Kutoš, Boris Ribaš
Srednja šola Domžale	Tadej Trinko
Srednja šola Josipa Jurčiča Ivančna Gorica	Darko Pandur
Srednja tehniška in poklicna šola Trbovlje (STPŠ)	Uroš Ocepek
Srednja tehniška šola (STŠ) Koper	Andrej Florjančič
Škofijska klasična gimnazija Šentvid	Helena Medvešek, Matevž Poljanc, Jure Slak, Matej Tomc
Šolski center Celje, Gimnazija Lava	Karmen Kotnik
Šolski center Celje, Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Dušan Fugina

Šolski center Kranj, Srednja tehniška šola	Aleš Hvasti
Šolski center Kranj, Strokovna gimnazija	Gašper Strniša
Šolski center Krško-Sevnica	Andrej Peklar
Šolski center Nova Gorica, Gimnazija in zdravstvena šola (GZŠ)	Barbara Pušnar, 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)	David Drofenik, Zoltan Sep, Franc Vrbančič
Šolski center Ravne na Koroškem, Srednja šola Ravne	Gorazd Geč, Zdravko Pavleković
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Miran Zevnik
Šolski center Velenje, Gimnazija Velenje	Miran Zevnik
III. gimnazija Maribor	Maja Čelan
Vegova Ljubljana	Marko Kastelic, 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.

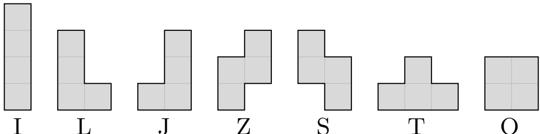
Off-line naloga je poskus, da se tovrstnim omejitvam malo izognemo: besedilo naloge in testni primeri zanjo so objavljeni več mesecev vnaprej, tekmovalci pa ne oddajajo programa, ki rešuje nalogo, pač pa oddajajo rešitve tistih vnaprej objavljenih testnih primerov. Pri tem imajo torej veliko časa in priložnosti, da dobro razmislijo o nalogi, preizkusijo več možnih pristopov k reševanju, počasi izboljšujejo svojo rešitev in podobno. Opis naloge in testne primere smo objavili oktobra 2014 skupaj z razpisom za tekmovanje v znanju; tekmovalci so imeli čas do 20. marca 2015 (dan pred tekmovanjem), da pošljejo svoje rešitve.

Ker se je lani in predlani tekmovanja v off-line nalogi udeležilo zelo malo tekmovalcev, smo isto nalogo uporabili tudi letos (sicer z novimi testnimi primeri). Tokrat je bila udeležba precej boljša, tako da bomo prihodnje leto za off-line nalogo razpisali kaj novega.

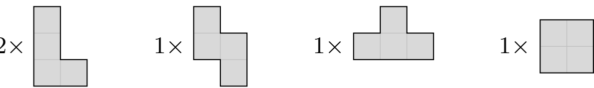
Opis naloge

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).

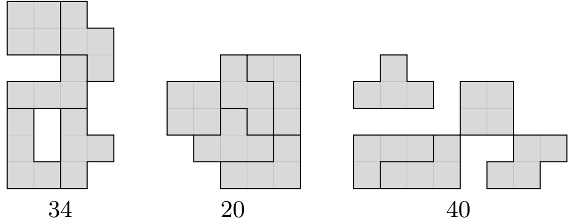
Možne oblike likov so naslednje (označimo jih lahko tudi z velikimi črkami abecede, ki so prikazane pod njimi, ker so tem črkam po obliki približno podobni):



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

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

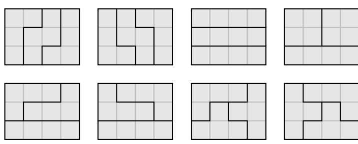
Tomaž Hočevar (FRI)	2893 točk
Rok Kralj (FMF)	2385
David Fabijan (FMF)	2100
Miloš Ljubotina (FE)	2056
Rok Lampret (FRI)	1691
Patrik Zajec (FRI + FMF)	1368
Dean Cerin (FAMNIT)	536
Tomaž Tomažič (FRI)	329
Tadej Vodopivec (FRI)	56

Na spletni strani tekmovanja so objavljene tudi vizualizacije vseh prejetih rešitev vseh tekmovalcev (`http://rtk.ijs.si/2015/zlaganje/rtk2015-zlaganje-vse.pdf`).

Rešitev

Osnovnim likom iz igre tetris pravimo tudi *tetromine* (ker so sestavljeni iz štirih enotskih kvadratov, podobno kot so domine sestavljene iz dveh enotskih kvadratov). Naloga od nas zahteva, da dani nabor tetromin zložimo v lik s čim manjšim obsegom. Najbolj primerni za to se zdita t.i. I-tetromina (1×4) in O-tetromina (2×2). Hitro opazimo, da bi bila naloga precej bolj obvladljiva, če bi imeli opravka samo z liki v obliki pravokotnikov; če bi bili vsi liki enakih dimenzij, pa še toliko bolje. Tetromine bomo torej poskusili zložiti v pravokotnike fiksnih dimenzij, nato pa bomo te pravokotnike zložili v končno obliko.

Pravokotnik dimenzije 4×3 lahko na primer sestavimo na več načinov — s tremi I-tetrominami ali pa z dvema T-tetrominama in eno L-tetromino ali pa še drugače, kot kaže spodnja slika.



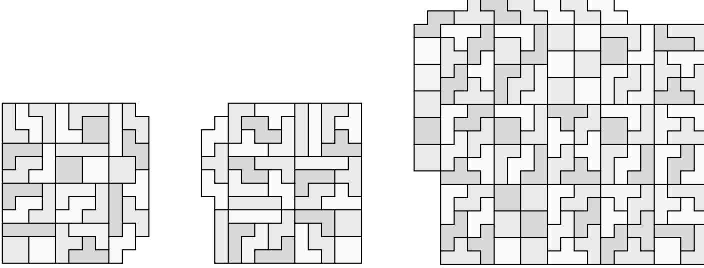
Pri večjih pravokotnikih je takih možnosti še več. Vnaprej si bomo pripravili vse možne nabore tetromin, s katerimi lahko sestavimo pravokotnik izbrane dimenzije $w \times h$. Za vsakega od veljavnih naborov si shranimo še eno od razporeditev tetromin v pravokotnik. V končni rešitvi bomo preizkusili različne ter vedno večje dimenzije w in h, zato potrebujemo učinkovit pristop za generiranje vseh možnih naborov tetromin. Izkaže se, da je smiselno polniti pravokotnik s tetrominami po diagonalah (namesto po vrsticah). Najprej poskrbimo, da so zasedene vse celice na prvi diagonali, nato na drugi itd. S tem v našem rekurzivnem preiskovanju dosežemo bolj zgodnje omejevanje stanj, ki ne vodijo do polnega pravokotnika. Pomagamo si lahko tudi s simetrijami, da ne generiramo simetričnih zlaganj tetromin v pravokotnik.

Sedaj se moramo odločiti, katere od razpoložljivih tetromin uporabiti za formiranje pravokotnika. Recimo, da imamo na razpolago $n_1, n_2, \ldots, n_7$ tetromin posameznega tipa. Pri izbiri načina formiranja pravokotnika želimo poskrbeti, da nam katerih tetromin ne bi prezgodaj zmanjkalo, ker bi si s tem omejili nadaljnje možnosti sestavljanja pravokotnikov. Za sestavljanje novega pravokotnika izberemo veljaven nabor tetromin $x_1, \ldots, x_n$, ki maksimizira vrednost $\min\{n_1 - x_1, \ldots, n_7 - x_7\}$. Tako zaporedoma sestavljamo vedno nove pravokotnike, dokler je to še mogoče. Nekaj tetromin nam običajno ostane, te pa bomo povsem na koncu zložili h končni rešitvi (na primer s požrešno strategijo, ki je opisana spodaj).

Iz nabora tetromin smo prišli do množice enako velikih pravokotnikov, ki jih želimo zložiti v lik s čim manjšim obsegom. Zlagali jih bomo kar enega poleg drugega v pravokotno mrežo. Pri dani (fiksni) površini je lik z najmanjšim obsegom kvadrat, zato lahko izračunamo, koliko vrstic in koliko stolpcev pravokotnikov potrebujemo, da bo končna oblika čim bolj kvadratna. Preizkusimo še kakšno vrstico več ali manj.

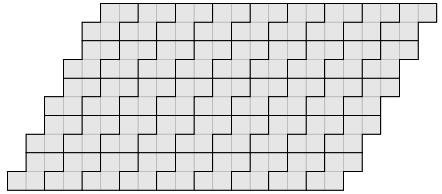
Naslednja slika prikazuje nekaj primerov razporedov (iz rešitev, ki jih je na tekmovanju oddal Tomaž Hočevar), dobljenih s to strategijo. Pri razporedu na levi so osnovni pravokotniki veliki 4×3, pri srednjem imamo pravokotnike 5×6, pri desnem razporedu pa pravokotnike 4×6. Vidimo lahko tudi, da je pri vsakem od

razporedov na koncu ostalo še nekaj tetromin, iz katerih se ni dalo sestaviti še enega pravokotnika, zato so bile s požrešnim pristopom razporejene po robovih lika:



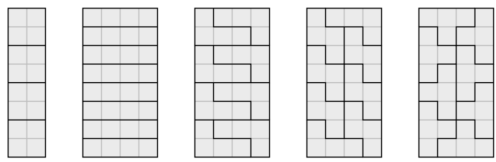
(Opomba: različni odtenki sive na tej in naslednjih slikah nimajo nobenega posebnega pomena, uporabili smo jih le zato, da se lažje razloči sosednje tetromine.)

V večini naključno generiranih primerov se doslej opisana strategija obnese izjemno dobro. Težava pa nastane v primerih, ki vsebujejo pretežno Z- in S-tetromine. Z njimi namreč ni mogoče zgraditi pravokotnika. Te primere obravnavamo z ločeno strategijo zlaganja v končno obliko paralelograma.



Pri testnih primerih, ki vsebujejo manjše število tetromin, se dobro obnese tudi požrešna strategija. V naključnem vrstnem redu dodajamo tetromine na poljubno mesto, ki maksimizira dotikajočo površino med že zgrajenim likom in novo tetromino. Z dovolj velikim številom ponovitev lahko pridemo do dobre rešitve. Poleg popolnoma naključnih vrstnih redov dodajanja tetromin lahko preizkusimo tudi take, pri katerih najprej porabimo vse tetromine ene oblike, nato vse tetromine druge oblike in tako naprej. Pri večjih testnih primerih se požrešna strategija ponavadi obnese slabše, ker je verjetnost, da bi s preizkušanjem nekaj naključnih vrstnih redov dobili res dober razpored, zelo majhna. Pogosto nastanejo razporedi, ki tvorijo lik zelo nepravilne oblike z veliko majhnimi štrlečimi deli, to pa močno poveča obseg lika.

Sistematično lahko tetromine zlagamo tudi tako, da iz tetromin posamezne oblike tvorimo „stolpe“ — pravokotnike, široke 2 ali 4 enote — nato pa te stolpe staknemo med sabo. Višino stolpov si moramo primerno izbrati, da bo na koncu nastal lik čim bolj kvadratne oblike. Nekaj primerov kaže naslednja slika: od leve proti desni imamo stolp iz O-jev, stolp iz I-jev, stolp iz J-jev, stolp iz S-jev (z dvema J-jema na koncih, da nastane pravokotnik) in stolp iz T-jev. Ni si tudi težko predstavljati, da lahko z dodajanjem tetromin iste vrste višino teh stolpov še poljubno povečamo:



Na stolpih temelji naslednji pristop, ki ga je na tekmovanju uporabil Rok Kralj. Najprej poiščimo vse možne načine, kako iz tetromin sestaviti pravokotnike velikosti $4 \times h$ za h od 1 do 14. Za posamezni testni primer potem s celoštevilskim linearnim programiranjem poiščimo tak nabor teh pravokotnikov, ki porabi čim več razpoložljivih tetromin. Kriterijska funkcija, ki jo pri tem minimiziramo, je $c \cdot r - p$, pri čemer je r število neuporabljenih tetromin, p število uporabljenih pravokotnikov, c pa je neka primerno velika konstanta, npr. $c = 100$; namen te kriterijske funkcije je, da skuša uporabiti čim več tetromin (zato je c velik), obenem pa spodbuja manjše pravokotnike, ker jih bo potem lažje zlagati naprej. Nato si izberemo višino stolpov, recimo v, in naše pravokotnike s požrešnim algoritmom (ki najprej uporabi daljše pravokotnike) zlagamo v stolpe, ki se po višini čim bolj približajo v, ne smejo pa ga preseči; tako dobljene stolpe na koncu le še staknemo skupaj. Preizkusimo več različnih višin in na koncu vrnemo najboljši dobljeni razpored. (Če imamo skupaj n tetromin, je njihova površina $4n$ in najmanjši obseg bi dobili, če bi lahko iz naših tetromin sestavili kvadrat $v \times v$ za $v = 2\sqrt{n}$. Ker to ni nujno mogoče, pa preizkusimo na primer vse višine z območja $v \pm 20$.)

Še en pristop, ki se je dobro obnesel pri majhnih testnih primerih, temelji na iskanju v snopu (*beam search*). Če hočemo zložiti n tetromin, lahko razmišljamo takole:

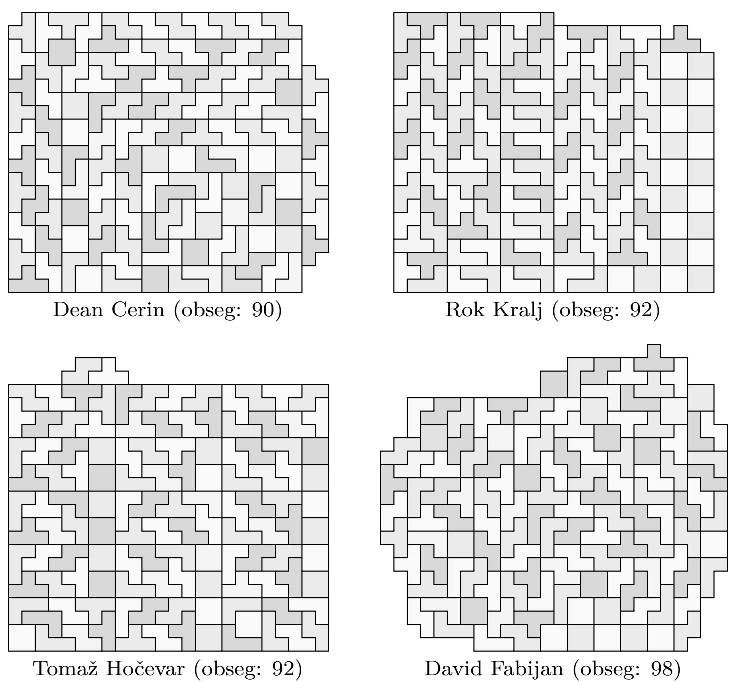
A := { razporedi, ki vsebujejo le eno tetromino (take oblike, za kakršno
  imamo pri trenutnem testnem primeru na voljo vsaj eno tetromino) };
for i := 2 **to** n:
  B := {};
  za vsak razpored r iz A:
    za vsak tak tip tetromine, ki ga imamo na voljo v več izvodih,
    kot pa jih trenutno vsebuje r:
      za vsak možen način, kako pritakniti eno táko tetromino k liku r
      (razpored, ki pri tem nastane, imenujmo r'):
        dodaj r' v B;
  A := najboljših b razporedov iz množice B; ($\star$)
vrni tisti razpored iz A, ki ima najmanjši obseg;

Parameter b (širina snopa), ki smo ga uporabili v vrstici ($\star$), je konstanta, ki si jo lahko izberemo poljubno (Dean Cerin, ki je uporabil ta postopek na tekmovanju, je vzel $b = 5000$). V zadnji vrstici se moramo nekako odločiti, kateri razporedi so najboljši. Dobro se obnese na primer možnost, da vsakemu razporedu očrtamo pravokotnik (s stranicami, vzporednimi koordinatnim osem); naj bo $w \times h$ velikost tega pravokotnika; razporede lahko zdaj uredimo po $|w - h|$, tiste z enako $|w - h|$ pa po obsegu. Tako na vsakem koraku spodbujamo razporede, ki bodo pripeljali do čim bolj kvadratnega lika. Ker je množica B, ki nastane pri gornjem postopku, lahko precej velika, je v praksi bolje, če namesto cele množice B v vsakem trenutku

hranimo le najboljših b razporedov iz nje; za to je koristno uporabiti kopico (*heap*), pri kateri imamo z vsakim novim razporedom največ $O(\log b)$ dela, da ga dodamo v kopico (če se izkaže za boljšega od b-tega najboljšega razporeda doslej).

Še ena možnost pa je, da tetromine zlagamo ročno; lahko bi si celo napisali program z grafičnim uporabniškim vmesnikom, ki bi nam ročno zlaganje tetromin olajšal. Naloga ročnega zlaganja tetromin nikakor ne prepoveduje in po naših izkušnjah se dá z zmerno veliko truda ročno sestaviti zelo dobre razporede. To je tudi glavni razlog, zakaj smo pri tej nalogi pripravili tako veliko testnih primerov (kar 300) — da bi odvrnili tekmovalce od ročnega zlaganja tetromin.

Za konec si oglejmo nekaj konkretnih primerov razporedov, ki so jih oddajali tekmovalci pri enem od srednje velikih testnih primerov ($3 \times$ L, $20 \times$ J, $26 \times$ Z, $26 \times$ S, $27 \times$ T in $20 \times$ O) in ki lepo ilustrirajo različne zgoraj opisane pristope k zlaganju likov.



Dean Cerin (obseg: 90) — Rok Kralj (obseg: 92)

Tomaž Hočevar (obseg: 92) — David Fabijan (obseg: 98)

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 13.–15. novembra 2015 v Zagrebu), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bo 15.–20. maja 2016 v Phuketu na Tajskem).

Na letošnjem UPM je sodelovalo 70 ekip s skupno 203 tekmovalci, ki so prišli s treh slovenskih univerz, nekaj pa je bilo celo srednješolcev. Tabela na naslednjih dveh straneh prikazuje vse ekipe, ki so se pojavile na vsaj enem krogu tekmovanja.

	Ekipa	Št. rešenih nalog*	Čas
1	Patrik Zajec, Vid Kocijan (FRI + FMF), Jasna Urbančič (FMF)	22	29:55:36
2	Jure Slak, Maks Kolman, Marko Ljubotina (FMF)	19	21:12:12
3	Sven Cerk, Martin Šušterič (FRI + FMF), Veno Mramor (FMF)	17	25:13:16
4	Žiga Željko, Žan Knafelc (Gim. Bežigrad), Aljaž Eržen (Vegova Lj.)	14	15:34:44
5	Vladan Jovičić, Daniel Siladji, Marko Palangetić (FAMNIT)	14	19:15:37
6	Jure Kolenko, Milutin Spasić, Ernest Beličič (FRI)	13	12:29:59
7	Alexei Drake, Andraž Dobnikar (FRI + FMF), Tibor Djurica Potpara (FMF)	12	15:31:15
8	Marko Novak (FRI), Aljaž Jelen (FE), Boštjan Zupančič (FMF)	12	15:48:24
9	Jure Bevc, Tomaž Stepišnik Perdih, Matej Petković (FMF)	12	16:38:22
10	Tadej Novak, Mitja Rozman (FMF), Dejan Krejić (Pedag. F.)	12	17:48:15
11	Sandi Mikuš, Jan Vatovec (FRI), Rok Kralj (FRI + FMF)	12	27:07:29
12	Anja Petković, Vesna Iršič, Žiga Lukšič (FMF)	11	16:17:42
13	Filip Koprivec (FMF), Filip Lebar (FE), Luka Kolar (Gim. Vič)	10	17:17:53
14	Jan Živković, Rok Poje, Žan Kusterle (FRI)	10	20:34:28
15	Aleksandar Todorović, Marko Tavčar (FAMNIT)	9	12:00:19
16	Andrej Dolenc, Peter Lazar (FRI + FMF), Klemen Bratec (FRI)	9	22:35:32
17	Domen Vidovič, Žan Skamljič, Dominik Korošec (FERI)	7	20:43:27
18	Gal Meznarič, Jan Mikolič, Aljaž Jeromel (FERI)	5	12:33:15
19	Urban Malc, Tomaž Vesel, Luka Loboda (FRI)	4	4:32:12
20	Mateja Hrast (FMF), Miloš Ljubotina (FE)	4	5:17:11
21	Matjaž Leonardis	4	6:16:55
22	Rok Kos, Bor Brecelj, Matej Marinko (Gim. Vič)	4	6:18:33
23	Rok Mohar, Benjamin Novak, Vitjan Zavrtanik (FRI)	4	6:48:20
24	Žiga Šmelcer (FE), Žiga Gradišar (FMF), Lojze Žust (FRI)	4	8:32:59
25	Matej Drobnič, Robert Koprivnik, Jan Maleš (FERI)	4	10:23:02
26	Tadej Jagodnik, Primož Kariž, Tomaž Kariž (FRI)	3	3:58:36
27	Erik Langerholc (FMF), Tadej Ciglarič, Domen Lušina (FRI)	3	4:08:28
28	Andraž Bajt, Blaž Repas (FRI + FMF), Luka Zakrajšek (FRI)	3	4:20:45
29	Sašo Stanovnik, Žiga Vučko, Kristijan Mirčeta (FRI)	3	6:27:45
30	Tobias Mihelčič (FRI + FMF), Sebastian Škoić (Gim. Ledina), Sebastjan Hinko Filip (FE)	3	6:33:12
31	Andraž Krašovec, Luka Krhlikar, Rok Založnik (FRI)	3	8:10:18
32	Tomaž Tomažič, Žiga Zupanec, Gašper Urh (FRI)	3	8:41:03
33	Gašper Romih, Miha Rot, Matej Logar (FMF)	3	9:31:04
34	Žiga Černigoj, Andrej Jugovic, Aleksander Tomič (FRI)	3	10:03:33
35	Jakob Šalej, Matevž Krajnik (FRI)	3	10:21:21
36	Ladislav Škufca, Aljaž Turk, Jan Blatnik (FRI)	2	1:53:54
37	Dejan Skledar, Jan Porner, Klemen Forstnerič (FERI)	2	2:55:35
38	Lidija Magdevska, Klarisa Trojer, Jernej Katanec (FRI + FMF)	2	3:42:27
39	Martin Turk, Klemen Rekajne, Jan Geršak (FRI + FMF)	2	3:52:12
40	Luka Toni (FRI + FMF), Domen Urh, Juš Debeljak (FRI)	2	4:28:34

* 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
41	Simon Weiss, Žiga Krajnik (FMF), Jan Aleksandrov (FRI)	2	5:04:45
42	Manca Žerovnik, Luka Krsnik, Ožbolt Menegatti (FRI)	2	5:23:58
43	Simon Prešern, Andrej Muhič, Marko Murgelj (FRI + FMF)	2	5:47:03
44	Gregor Pirš, Dragana Božović, Martin Duh (FNM)	1	0:09:54
45	Dan Toškan, Deni Cerovac, Sebastian Mežnar (STŠ Koper)	1	0:19:12
46	Rok Ljalić, Luka Prijatelj, Matjaž Mav (FRI)	1	0:55:24
47	Kevin Sedevčič, Tadej Magajna, Peter Us (FRI)	1	1:00:04
48	Andrej Rus, Domen Balantič, Špela Čopi (FRI + FMF)	1	1:04:16
49	Leo Gombač, Jan Grbac (FAMNIT), Peter Kozlovič	1	1:13:07
50	Urban Rajter, Tine Tetičkovič, Nikola Klipa (FNM)	1	1:25:47
51	Nejc Lovrenčič, Miha Hozjan, Aljaž Razpotnik (FERI)	1	1:41:52
52	David Šket, Srečko Šmerc, Klemen Uršič (FERI)	1	1:53:18
53	Aljaž Heričko, Tadej Stošič, Sašo Marković (FERI)	1	2:00:06
54	David Možina, Boštjan Lasnik, Nejc Škerjanc (FRI)	1	2:03:14
55	Denis Rajković, Matic Bizjak (FRI + FMF), Tine Šubic (FRI)	1	2:04:16
56	Matej Tomc, Marko Rus, Matevž Poljanc (Šk. klas. gimn.)	1	2:09:24
57	Miha Štravs (FRI + FMF), Blaž Horjak (FRI), Jaka Šauer (FE)	1	2:51:10
58	Tadej Žerak, Alen Vegi Kalamar, Marko Jovčeski (FNM)	1	3:16:00
59	Miha Mencin, Eva Križman, Nejc Nadižar (FRI + FMF)	1	3:22:14
60	Klemen Turšič, Grega Mežič (FRI)	1	3:33:44
61	Žiga Kern, Rok Hudobivnik, Simon Cof (FRI)	1	3:35:29
62	Ivan Kolundžija, Blaž Suhadolnik, Jan Likar (FRI + FMF)	1	3:43:42
63	Blaž Kranjc, Jan Malec, Katja Klobas (FMF)	1	3:46:53
64	Tadej Ternar, Janez Krnc, Valentin Kerman (FERI)	0	0:00:00
	Luka Avbreht, Samo Kralj, Marcel Čampa (FMF)	0	0:00:00
	Jani Pezdevšek, Mitja Gologranc, Lenart Bobek (FERI)	0	0:00:00
	Izak Pučko, Nino Petrovič (FERI)	0	0:00:00
	Miha Kebe, Florijan Klezin, Mario Kenda (FERI)	0	0:00:00
	Lea Vohar, Nina Vehovec, Tina Avbelj (FRI + FMF)	0	0:00:00
	Nejc Kadivnik, Kristjan Dragovan Širnik, Miha Bogataj (ŠC Kranj, Strok. gimn.)	0	0:00:00

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

Na srednjeevropskem tekmovanju so nastopile ekipe 1, 2 in 3 kot predstavnice Univerze v Ljubljani, 17 kot predstavnica Univerze v Mariboru in 5 kot predstavnica Univerze na Primorskem. V konkurenci 62 ekip s 27 univerz iz 7 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
34	Sven Cerk, Martin Šušterič, Veno Mramor	4	13:01
37	Patrik Zajec, Vid Kocijan, Jasna Urbančič	3	3:26
38	Jure Slak, Maks Kolman, Marko Ljubotina	3	4:38
45	Vladan Jovičić, Daniel Siladji, Marko Palangetić	2	6:51
50	Domen Vidovič, Žan Skamljič, Dominik Korošec	1	1:14

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. 171–178.

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

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

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

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

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

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

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

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

Ali bi znal(a) v programu uporabiti naslednje podatkovne strukture:

Drevo	☐ da	☐ ne
Hash tabela (razpršena / asociativna tabela)	☐ da	☐ ne
S kazalci povezan seznam (linked list)	☐ da	☐ ne
Sklad (stack)	☐ da	☐ ne
Vrsta (queue)	☐ da	☐ ne

Ali bi znal(a) v programu uporabiti naslednje algoritme:

Evklidov algoritem (za največji skupni delitelj)	☐ da	☐ ne
Eratostenovo rešeto (za iskanje praštevil)	☐ da	☐ ne
Poznaš formulo za vektorski produkt	☐ da	☐ ne
Rekurzivni sestop	☐ da	☐ ne
Iskanje v širino (po grafu)	☐ da	☐ ne
Dinamično programiranje	☐ da	☐ ne

[če misliš, da to pomeni uporabo new, GetMem, malloc ipd., potem obkroži „ne“]

Katerega od algoritmov za urejanje ☐ da ☐ ne

Katere(ga)? ☐ bubble sort (urejanje z mehurčki)
☐ insertion sort (urejanje z vstavljanjem)
☐ selection sort (urejanje z izbiranjem)
☐ quicksort
☐ kakšnega drugega: ____________________

Ali poznaš zapis z velikim O za časovno zahtevnost algoritmov?
[npr. $O(n^2)$, $O(n \log n)$ ipd.] ☐ da ☐ ne

[Le pri 1. in 2. skupini.] V besedilu nalog trenutno objavljamo deklaracije tipov in podprogramov v pascalu, C/C++, C#, pythonu in javi.
— Ali razumeš kakšnega od teh jezikov dovolj dobro, da razumeš te deklaracije v besedilu naših nalog? ☐ da ☐ ne
— So ti prišle deklaracije v pythonu kaj prav? ☐ da ☐ ne
— Ali bi raje videl(a), da bi objavljali deklaracije (tudi) v kakšnem drugem programskem jeziku? Če da, v katerem? ____________________

V rešitvah nalog trenutno objavljamo izvorno kodo v C-ju.
— Ali razumeš C dovolj dobro, da si lahko kaj pomagaš z izvorno kodo v naših rešitvah? ☐ da ☐ ne
— Ali bi raje videl(a), da bi izvorno kodo rešitev pisali v kakšnem drugem jeziku? Če da, v katerem? ____________________

[Le pri 1. in 2. skupini.] Kakšno je tvoje mnenje o sistemu za oddajanje odgovorov prek računalnika? ____________________

____

[Le pri 3. skupini.] Letos v tretji skupini podpiramo reševanje nalog v pascalu, C, C++, C#, javi in VB.NET. Bi rad uporabljal kakšen drug programski jezik? Če da, katerega? __

____

Katere od naslednjih jezikovnih konstruktov in programerskih prijemov znaš uporabljati?	ne poznam	da, slabo	da, dobro
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz standardnega vhoda ali pa ju zapisati na standardni izhod?	☐	☐	☐
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz datoteke ali pa ju zapisati v datoteko?	☐	☐	☐
Tabele (**array**):			
— enodimenzionalne	☐	☐	☐
— dvodimenzionalne	☐	☐	☐
— večdimenzionalne	☐	☐	☐
Znaš napisati svoj podprogram (**procedure**, **function**)	☐	☐	☐

Poznaš rekurzijo ☐ ☐ ☐
Kazalce, dinamično alokacijo pomnilnika (New/Dispose, GetMem/FreeMem, malloc/free, **new**/**delete**, ...) ☐ ☐ ☐
Zanka **for** ☐ ☐ ☐
Zanka **while** ☐ ☐ ☐
Gnezdenje zank (ena zanka znotraj druge) ☐ ☐ ☐
Naštevni tipi (*enumerated types* — **type** ImeTipa = (Ena, Dve, Tri) v pascalu, **typedef enum** v C/C++) ☐ ☐ ☐
Strukture (**record** v pascalu, **struct**/**class** v C/C++) ☐ ☐ ☐
and, **or**, **xor**, **not** kot aritmetični operatorji (nad biti celoštevilskih operandov namesto nad logičnimi vrednostmi tipa boolean) (v C/C++/C#/javi: &, |, ^, ~) ☐ ☐ ☐
Operatorja **shl** in **shr** (v C/C++/C#/javi: <<, >>) ☐ ☐ ☐
Znaš uporabiti kakšnega od naslednjih razredov iz standardnih knjižnic:
hash_map, hash_set, unordered_map, unordered_set (v C++), Hashtable, HashSet (v javi/C#), Dictionary (v C#), dict, set (v pythonu) ☐ ☐ ☐
map, set (v C++), TreeMap, TreeSet (v javi), SortedDictionary (v C#) ☐ ☐ ☐
priority_queue (v C++), PriorityQueue (v javi), heapq (v pythonu) ☐ ☐ ☐

[Naslednja skupina vprašanj se je ponovila za vsako nalogo po enkrat.]
Zahtevnost naloge: ☐ prelahka ☐ lahka ☐ primerna ☐ težka ☐ pretežka ☐ ne vem
Naloga je (ali: bi) vzela preveč časa: ☐ da ☐ ne ☐ ne vem
Mnenje o besedilu naloge:
— dolžina besedila: ☐ prekratko ☐ primerno ☐ predolgo
— razumljivost besedila: ☐ razumljivo ☐ težko razumljivo ☐ nerazumljivo
Naloga je bila: ☐ zanimiva ☐ dolgočasna ☐ že znana ☐ povprečna
Si jo rešil(a)?
☐ nisem rešil(a), ker mi je zmanjkalo časa za reševanje
☐ nisem rešil(a), ker mi je zmanjkalo volje za reševanje
☐ nisem rešil(a), ker mi je zmanjkalo znanja za reševanje
☐ rešil(a) sem jo le delno, ker mi je zmanjkalo časa za reševanje
☐ rešil(a) sem jo le delno, ker mi je zmanjkalo volje za reševanje
☐ rešil(a) sem jo le delno, ker mi je zmanjkalo znanja za reševanje
☐ rešil(a) sem celo
Ostali komentarji o tej nalogi: ______________________________

____

Katera naloga ti je bila najbolj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? ______________________________

____

Katera naloga ti je bila najmanj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? ______________________________

____

Na letošnjem tekmovanju ste imeli tri ure / pet ur časa za pet nalog.
Bi imel(a) raje: ☐ več časa ☐ manj časa ☐ časa je bilo ravno prav
Bi imel(a) raje: ☐ več nalog ☐ manj nalog ☐ nalog je bilo ravno prav

Kakršne koli druge pripombe in predlogi. Kaj bi spremenil(a), popravil(a), odpravil(a), ipd., da bi postalo tekmovanje zanimivejše in bolj privlačno? ____________________

____

Kaj ti je bilo pri tekmovanju všeč? ____________________

____

Kaj te je najbolj motilo? ____________________

____

Če imaš kaj vrstnikov, ki se tudi zanimajo za programiranje, pa se tega tekmovanja niso udeležili, kaj bi bilo po tvojem mnenju treba spremeniti, da bi jih prepričali k udeležbi?

____

____

Poleg tekmovanja bi radi tudi v preostalem delu leta organizirali razne aktivnosti, ki bi vas zanimale, spodbujale in usmerjale pri odkrivanju računalništva. Prosimo, da nam pomagate izbrati aktivnosti, ki vas zanimajo in bi se jih zelo verjetno udeležili.

Udeležil bi se oz. z veseljem bi spremljal:

- ☐ izlet v kak raziskovalni laboratorij v Evropi (po možnosti za dva dni)
- ☐ poletna šola računalništva (1 teden na IJS, spanje v dijaškem domu)
- ☐ poletna praksa na IJS
- ☐ predstavitve novih tehnologij (.NET, mobilni portali, programiranje „vgrajenih računalnikov“, strojno učenje, itd.) (1× mesečno)
- ☐ predavanja o algoritmih in drugih temah, ki pridejo prav na tekmovanju (1× mesečno)
- ☐ reševanje tekmovalnih nalog (naloge se rešuje doma in bi bile delno povezane s temo, predstavljeno na predavanju; rešitve se preveri na strežniku) (1× mesečno)
- ☐ tvoji predlogi: ____________________

  ____

Vesel(a) bi bil pomoči pri:

- ☐ iskanju štipendije
- ☐ iskanju podjetij, ki dijakom ponujajo njim prilagojene poletne prakse in druge projekte, kjer se ob mentorstvu lahko veliko naučijo.

Ali si pri izpolnjevanju ankete prišel/la do sem? ☐ da ☐ ne

Hvala za sodelovanje in lep pozdrav! Tekmovalna komisija

REZULTATI ANKETE

Anketo je izpolnilo 78 tekmovalcev prve skupine, 21 tekmovalcev druge skupine in 8 tekmovalcev tretje skupine. (Opozorimo na to, da je zaradi majhnega števila tekmovalcev v tretji skupini iz tamkajšnjih anket še posebej težko vleči kakšne pametne posplošitve in zaključke.) 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. 172. Tam so tudi podatki o povprečnem številu točk, doseženem pri posamezni nalogi, tako da lahko primerjamo mnenje tekmovalcev o zahtevnosti naloge in to, kako dobro so jo zares reševali.

V povprečju so se zdele tekmovalcem v vseh skupinah naloge še kar težke, vendar so številke podobne kot v prejšnjih letih. Če pri vsaki nalogi pogledamo povprečje mnenj o zahtevnosti te naloge (1 = prelahka, 3 = primerna, 5 = pretežka) in vzamemo povprečje tega po vseh petih nalogah, dobimo: 3,41 v prvi skupini (v prejšnjih letih 3,40, 3,28, 3,39, 3,56, 3,34, 3,56), 3,33 v drugi skupini (prejšnja leta 3,44, 3,35, 3,50, 3,39, 3,38, 3,46) in 3,61 v tretji skupini (prejšnja leta 3,19, 3,40, 3,21, 3,57, 3,92).

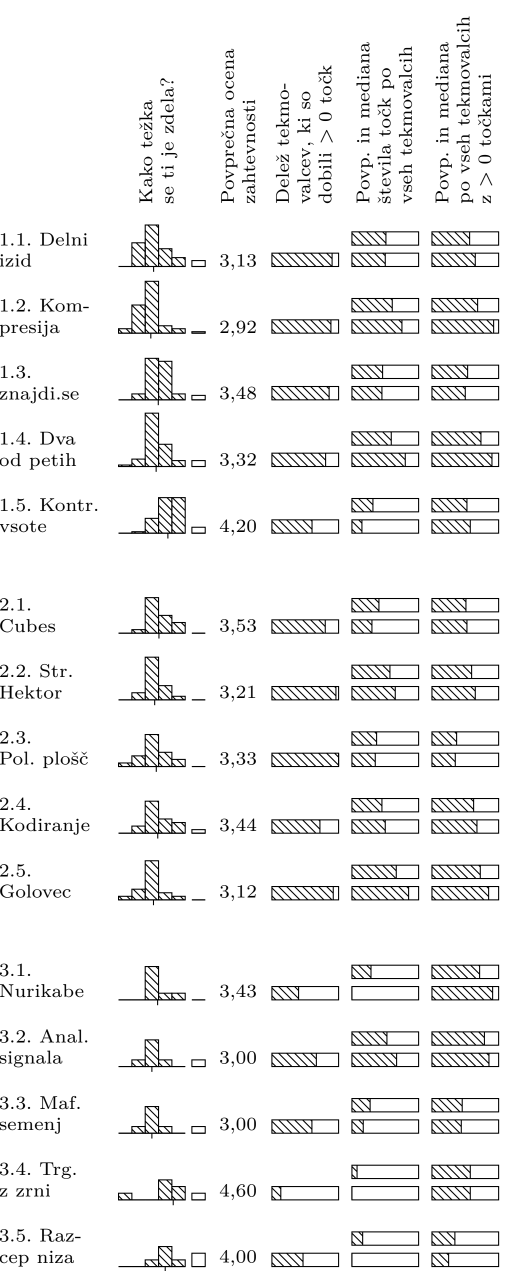
Med tem, kako težka se je naloga zdela tekmovalcem, in tem, kako dobro so jo zares reševali (npr. merjeno s povprečnim številom točk pri tej nalogi), je ponavadi (šibka) negativna korelacija; letos je bila še malo močnejša kot običajno ($R^2 = 0{,}56$; v prejšnjih letih 0,14, 0,52, 0,21, 0,11, pred tem okoli 0,4).

Daleč največ pripomb o tem, kako da je naloga težka, je bilo pri nalogi 1.5 (kontrolne vsote); naloge, pri kateri moraš implementirati nek predpisan vmesnik in pri tem uporabljati že podane funkcije, se tekmovalcem pogosto zdijo težke, poleg tega se je pri tej nalogi zdelo mnogim tudi besedilo dolgo in težko razumljivo. Kot težje so ocenili tudi naloge nekaj tudi pri 1.3 (znajdi.se), 2.1 (it's raining cubes) in 3.4 (trgovanje z zrni).

Kot najlažjo so tekmovalci ocenili nalogo 1.2 (kompresija), v drugi skupini pa 2.5 (Golovec). Pri slednji je bilo celo nekaj pripomb, da je v mnogih sodobnih programskih jezikih praktično trivialna (npr. če je razpršena tabela del jezika in jo je enostavno uporabljati).

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 173. Nad razumljivostjo besedil ni veliko pripomb (sicer malo več kot prejšnja leta); kot težko razumljiva izstopa predvsem naloga 1.5 (kontrolne vsote), deloma tudi 2.1 (it's raining cubes), 2.2 (strahopetni Hektor) in 3.1 (Nurikabe). Pri večini teh nalog (razen mogoče 3.1) je težava najbrž v tem, da jih je težko razložiti tako, da bo opis naloge preprost in razumljiv, vendar hkrati tudi dovolj natančen in nedvoumen.

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, približno enako kot v prejšnjih letih oz. celo še malo bolj. Edina naloga, pri kateri je bilo veliko pripomb, da je predolga, je bila 1.5 (kontrolne vsote). Pri večini ostalih nalog se je besedilo več ljudem zdelo prekratko kot predolgo.



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

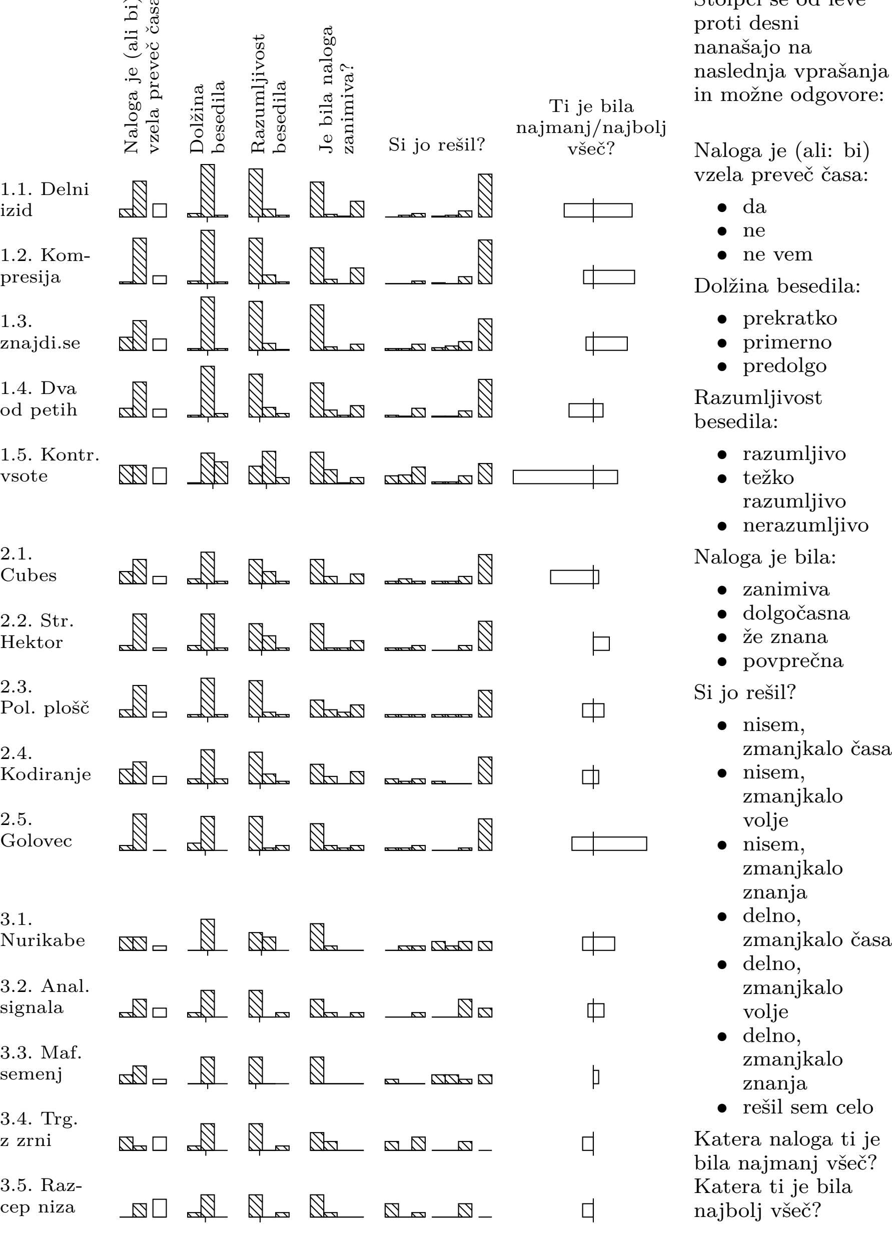
Pomen stolpcev v vsaki vrstici:

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.



Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta. Razlike v oceni zanimivosti med nalogami so večinoma majhne, kot bolj zanimive izstopajo 1.3 (znajdi.se), 2.2 (strahopetni Hektor) in 3.3 (mafijski semenj). Ob tem izboru se je težko upreti zaključku, da se tekmovalcem naloge zdijo zanimive predvsem, če so ovite v zanimivo zgodbico. Kot bolj dolgočasna izstopa naloga 1.5 (kontrolne vsote). Pripomb, da je naloga že znana, je bilo malo (pol manj kot lani).

Pripomb, da bi naloga vzela preveč časa, je bilo še malo manj kot lani. Največ takih pripomb je bilo pri nalogah 1.5 (kontrolne vsote), 2.4 (kodiranje), 3.2 (analiza signala) in 3.4 (trgovina z zrni). Najbrž to mnenje izvira iz dejstva, da so te naloge malo težje, saj drugače same po sebi niso zamudne za reševanje (še posebej 2.4 in 3.2).

Pri glasovih o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, je kot izrazito nepopularna izstopala naloga 1.5 (kontrolne vsote), malo manj pa tudi 1.4 (dva od petih). V prvi skupini jim je bila najbolj všeč naloga 1.2 (kompresija), v drugi je kot popularna izstopala 2.5 (Golovec). Naloga 1.1 (delni izid) pa je dobila veliko glasov pri obeh vprašanjih (nekaterim je bila najbolj všeč, nekaterim najmanj).

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.	5 %	22 %	67 %
map v C++ ipd.	5 %	28 %	50 %
unordered_map v C++ ipd.	12 %	33 %	67 %
zamikanje s **shl**, **shr**	10 %	33 %	86 %
operatorji na bitih	41 %	56 %	100 %
strukture	26 %	50 %	86 %
naštevni tipi	23 %	32 %	67 %
gnezdenje zank	76 %	89 %	86 %
zanka **while**	92 %	100 %	100 %
zanka **for**	91 %	100 %	100 %
kazalci	24 %	33 %	71 %
rekurzija	31 %	63 %	100 %
podprogrami	62 %	74 %	100 %
več-d tabele (**array**)	39 %	56 %	100 %
2-d tabele (**array**)	59 %	79 %	100 %
1-d tabele (**array**)	80 %	100 %	100 %
delo z datotekami	52 %	74 %	100 %
std. vhod/izhod	76 %	89 %	100 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

	Prva skupina	Druga skupina	Tretja skupina
drevo	36 %	48 %	100 %
hash tabela	25 %	38 %	75 %
seznam (linked list)	22 %	57 %	100 %
sklad (stack)	33 %	57 %	100 %
vrsta (queue)	42 %	43 %	100 %
Evklidov algoritem	79 %	67 %	86 %
Eratostenovo rešeto	68 %	57 %	57 %
vektorski produkt	28 %	57 %	100 %
rekurzija	21 %	43 %	86 %
dinamično prog.	17 %	42 %	86 %
iskanje v širino	7 %	28 %	71 %
O-zapis	31 %	55 %	86 %
urejanje	54 %	86 %	100 %
bubble sort	37 %	57 %	63 %
insertion sort	26 %	24 %	75 %
selection sort	23 %	48 %	75 %
quicksort	19 %	52 %	63 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

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 njegova prednost pred drugimi jeziki manjša. V prvi skupini sta letos najpogostejša jezika C++ in python, ki sta približno izenačena, malo manj ljudi je uporabljalo javo, še malo manj pa C#. V drugi skupini je najpogostejši C++, malo za njim sta java in C#; python pa je bil letos v drugi skupini redek. Razmerja med temi štirimi jeziki iz leta v leto po malem nihajo brez kakšnega očitnega trenda. V tretji skupini je C++ daleč najpogostejši, sledi pa mu java; C# ni letos uporabljal nihče; novost v tretji skupini pa je Visual Basic.NET, ki smo ga letos podpirali prvič in ga je uporabljal en tekmovalec.

Podobno kot prejšnja leta se je tudi letos pojavilo nekaj tekmovalcev (in tekmovalk), ki oddajajo le rešitve v psevdokodi ali pa celo naravnem jeziku, tudi tam, kjer naloga sicer zahteva izvorno kodo v kakšnem konkretnem programskem jeziku. Iz tega bi človek mogoče sklepal, da bi bilo dobro dati več nalog tipa „opiši postopek“ (namesto „napiši podprogram“), vendar se v praksi običajno izkaže, da so takšne naloge med tekmovalci precej manj priljubljene in da si večinoma ne predstavljajo preveč dobro, kako bi opisali postopek (pogosto v resnici oddajo dolgovezne opise izvorne kode v stilu „nato bi s stavkom **if** preveril, ali je spremenljivka x večja od spremenljivke y“). Podobno kot lani smo tudi letos pri nalogah tipa „opiši postopek“ pripisali „ali napiši podprogram (kar ti je lažje)“.

Podobno kot v prejšnjih letih je v anketi precej tekmovalcev napisalo, da dobro poznajo tudi PHP, vendar so ga na tekmovanju uporabljali le trije. En tekmovalec

	Leto in skupina																	
	2015			2014			2013			2012			2011			2010		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal	5	2		$2\frac{1}{2}$	2	1	1		1	6	1	4	3	4	3	$4\frac{1}{2}$	5	2
C	3		1	$3\frac{1}{2}$	6		2	7		7	2	1	7	2		6	6	1
C++	27	9	$9\frac{1}{2}$	19	$4\frac{1}{2}$	$10\frac{1}{2}$	17	$12\frac{1}{2}$	7	26	16	9	$23\frac{1}{2}$	19	8	33	$17\frac{1}{2}$	13
java	22	6	$3\frac{1}{2}$	23	2	$1\frac{1}{2}$	12	8	1	17	$6\frac{1}{2}$	1	6	5	3	5	9	4
PHP	3		–	2		–	1	$\frac{1}{2}$	–		1	–	$\frac{1}{2}$		–	1	1	–
basic			1		1	–	1		–			–			–			–
C#	16	5		12	$1\frac{1}{2}$	2	18	$\frac{1}{2}$		17	1	3	4	2	3		$\frac{1}{2}$	1
python	26	1	–	16	6	–	16	8	–	25	5	–	20	6	–	12	2	–
NewtonScript			–			–		$\frac{1}{2}$	–		$\frac{1}{2}$	–			–			–
javascript	1		–	1		–			–			–			–			–
batch			–	1		–			–			–			–			–
psevdokoda	6	1	–	10		–	6		–	3		–	6		–	4		–
nič	4	1		4	2		2			2			1	1		1	5	

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“.

---

je reševal v javascriptu, kakšnih bolj eksotičnih jezikov pa letos na tekmovanju niso uporabljali.

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže gornja tabela. 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 uporablja `string` namesto **char** * in tip `vector` namesto tradicionalnih tabel (*arrays*). Letos smo prvič opazili pri enem tekmovalcu tudi rešitve v C++11/14 (med drugim je uporabljal **auto** v njegovem novem pomenu).

V besedilu nalog za 1. in 2. skupino objavljamo deklaracije tipov, spremenljivk, podprogramov ipd. v pascalu, C/C++, C#, pythonu in javi. Delež tekmovalcev, ki pravijo, da deklaracije razumejo, je letos v prvi skupini nižji kot običajno (54/72), v drugi je še kar visok (17/20). 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. Tako se človek vpraša, koliko se smemo na odgovore pri teh vprašanjih sploh zanašati. V vsakem primeru pa se poskušamo zadnja leta v besedilih nalog izogibati deklaracijam v konkretnih programskih jezikih in jih zapisati bolj na splošno, na primer „napiši funkcijo foo(x, y)“ namesto „napiši funkcijo **bool** foo(**int** x, **int** y)“.

V rešitvah nalog zadnja leta objavljamo izvorno kodo le v C-ju; tekmovalce smo v anketi vprašali, če razumejo C dovolj, da si lahko kaj pomagajo s to izvorno kodo, in če bi radi videli izvorno kodo rešitev še v kakšnem drugem jeziku. Večina je s C-jem sicer zadovoljna (31/68 v prvi skupini, 13/20 v drugi, 6/7 v tretji), vendar je letos v prvi skupini prvič več takih, ki pravijo, da izvorne kode v rešitvah ne razumejo, kot takih, ki jo razumejo. Zanimivo vprašanje je, ali bi s kakšnim drugim

jezikom dosegli večji delež tekmovalcev (koliko tekmovalcev ne bi razumelo rešitev v javi? ali v pythonu?). Med jeziki, ki bi jih radi videli namesto (ali poleg) C-ja, jih največ omenja 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 drugi skupini je povprečja starost malo višja kot lani, v tretji pa malo nižja. Nastopili so tudi trije osnovnošolci (dva v prvi skupini in eden v tretji); teh pri izračunu povprečnega letnika v spodnji tabeli nismo upoštevali.

Skupina	Št. tekmovalcev po letnikih: OŠ	1	2	3	4	Povprečni letnik
prva	2	19	29	33	30	2,7
druga		1	2	6	16	3,5
tretja	1		2	6	6	3,3

Druga vprašanja

Podobno kot prejšnja leta je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!). V smislu širitve zanimanja za tekmovanje in večanja števila tekmovalcev se zelo dobro obnese šolsko tekmovanje, ki ga izvajamo zadnjih nekaj let, saj se odtlej v tekmovanje vključuje tudi nekaj šol, ki prej na našem državnem tekmovanju niso sodelovale. Nekaj ljudi je za naše tekmovanje slišalo na računalniškem tekmovanju Bober, ki ga tudi organizira društvo ACM Slovenija.

Pri vprašanju, kje so se naučili programirati, je podobno kot prejšnja leta najpogostejši odgovor, da so se naučili programirati sami; sledijo tisti, ki so se naučili programirati v šoli (teh je malo več kot lani), še malo manj pa je takih, ki so se naučili programirati na krožkih ali tečajih.

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni. Med tistimi, ki niso, so mnenja precej razdeljena, vendar je podobno kot lani najpogostejša kombinacija „več časa, enako nalog“.

	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	64	2	7	5	45	34	22	5	5	15	10	40	5	12	48	26	26	19	30	28	21	27	35
II	16	0	4	1	14	6	7	1	2	5	1	11	0	2	15	7	3	4	3	8	1	4	5
III	6	1	0	2	8	2	2	1	2	2	0	5	1	0	6	1	1	2	1	3	3	2	1

Iz odgovorov na vprašanje, kakšne potekmovalne dejavnosti bi jih zanimale, je težko zaključiti kaj posebej konkretnega.

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb. Od 2009 imajo tekmovalci v prvi in drugi skupini možnost pisati svoje odgovore na računalniku namesto na papir (kar so si prej v anketah že večkrat želeli). Velika večina jih je res oddajala odgovore na računalniku, nekaj pa jih je vseeno reševalo na papir. Pri oddajanju odgovorov na računalniku je bilo žal še vedno nekaj težav, vendar manj kot lani.

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) Presenetljivo veliko tekmovalcev ne uporablja unarnega minusa, ampak namesto tega raje množijo z −1:

```
if ((tr * (−1)) > naj) { naj = tr * (−1); }
```

Podobnih primerov je bilo še precej. Eden se je na ta način izogibal celo odštevanju:

```
if kos < 0: # točke ločimo glede na predznak
  a += kos * −1;
```

(1.1) Vsako leto imamo kakšnega tekmovalca, ki si navodilo „opiši postopek“ razlaga kot „opiši, kako človek napiše program“. Letos smo dobili med drugim tale sijajni primer:

1. Vključimo knjižnico
2. Zapišemo našo sistemsko funkcijo `int main(void){`.

⋮

15. Končamo program z `return(0);`

Še dobro, da ni začel s „prižgemo računalnik“.

(1.1) Megalomanski komentarji na začetku ene od rešitev:

```
/* Made by —— ———, lord of the time, space and rings
   Če špila Slovenija je čisto preprosto, toliko košev ko je dala
   naša reprezentanca, nasprotniki jasno niso dali nobenega xD */
```

(1.1) Letos imamo celo dva tekmovalca, ki spremenljivke imenujeta „pomnilniki“ (za prejšnji tak primer glej bilten 2013, str. 182).

> To zabeležimo v nek pomnilnik (razliko/odstopanje). [...] Rezultat posamezne ekipe shranjujemo v več pomnilnikih.
>
> V programu najprej ločim posamezne točke v korakih ter jih zapišem v nove pomnilnike z besedilom (kor). Ko imam n teh pomnilnikov in v vsakem 2 veliki črki, ki predstavljata točke, grem na iskanje 1. koraka.

(1.1) Rešitev s seštevanjem vsevprek:

> program bi moral prvo števila seštevati z leve proti desni ter shraniti največji delni izid
> potem z desne proti levi nato pa z sredine levo ter potem desno

(1.1) Rešitev z veliko bazami podatkov:

> Program podatke jemlje iz enega vhoda in jih shranjuje v tri baze. [...] Podatke, večje od nič, shranjuje v bazo 1, manjše od nič pa v bazo 2. Najprej prebere en podatek. Njegovo razliko od 0 zapiše v bazo 3. [...] Nato sešteje vrednosti v vseh bazah.

(1.1) Posrečena tipkarska napaka:

> Na koncu sešteje vse izpisane ekipe in izpiše, katera ekipa je imela več delnih izvidov.

(1.1) Rešitev, ki iz programiranja naredi doživljajski spis:

> Želel sem ustvariti program, ki najprej prebere meritve s funkcijo Preberi in nato te meritve shrani v seznam, imenovan višina. V naslednjem koraku sem ustvaril spremenljivki prejsnji in naslednji.

itd. itd. Njegove rešitve so sploh polne dobrih namenov; pri tretji nalogi začne komentar s „Hotel sem ustvariti program, ki bi najprej sprejel prvi input. . . “ in pri četrti s „Program bi najprej sprejel input.“

(1.2) Iz ene od bolj nekoherentnih rešitev:

```
def Preberi()
   visina = int(input())
   prejsnji = visina[0]
   prejsnji += naslednji
```

(1.2) Zanimiva sintaktična inovacija — ekstra generični seznam, ki ima lomljene oklepaje < > ne le okoli int, ampak tudi okoli list:

```
<list><int> list = new List <int><list>; // novi List
```

(1.2) Rešitev z nezaupanjem do operatorja %:

```
if n % 2 == 0 or n == 0:
```

(1.2) Izviren pristop k neskončni zanki:

```
ponavljaj, dokler veljajo vsi zakoni fizike:
   kliči funkcijo preberi
   kliči funkcijo meritev
```

(1.2) Komentar pri eni od neskončnih zank:

```
while (true) // neskooooooooooooooončna zanka
```

Verjetno je tako, da če hočemo še bolj neskončno zanko, moramo le dodati še nekaj o-jev :)

(1.2) Rešitev z veliko rjovenja:

```
// the description is bellow...
```

(1.2) Nagrado za najboljšo pravopisno napako letos dobi:

```
pubic static void main(String[] args) {
```

(1.2) Rešitev z zelo neobičajnim pogledom na podprograme:

```
function p: integer;
procedure s: integer;
⋮
Read(p); { preberem p (število, ki ga pridobim s funkcijo p) }
⋮
s := p;
```

(1.2) Zelo posrečena pravopisna napaka:

```
// infinite loop within witch to do the magic
```

(1.3) Rešitev za ljubitelje grafičnega uporabniškega vmesnika:

```
OpenDialog text = new OpenDialog();
⋮
MessageBox.Show(rezultat.ToString());
```

(1.3) Zanimivo neroden način za preverjanje, če je spremenljivka `a` (tipa **char**) velika črka:

```
for (int i = (int) A; i < (int) Z; i++)
   if (a == (char) i) tocke += (char) i;
```

Očitno se je ta tekmovalec zavedal, da lahko znake pretvori v njihove celoštevilske kode, ni pa pomislil, da lahko zato zanko poenostavi v **if** ('A' <= a && a <= 'Z'). Naslednji tekmovalec pa je nekaj takega sicer naredil, vendar je močno precenil število velikih črk:

```
if (ss.charAt(i) > 67 && ss.charAt(i) < 122) // 67 in 122 naj bi bili številki oz. mesto
                                             // velikih črk v ASCII tabeli
```

Saj ne pričakujemo, da bodo ljudje na pamet vedeli kode črk v tabeli ASCII, lahko pa bi pomislil, da je velikih črk angleške abecede le 26, razlika $122 - 67$ pa je občutno večja, torej mora biti tam še precej drugih znakov...

(1.3) Rešitev z veliko spoštovanja do velikih črk:

> To drugo črko bi spet poiskal še med drugimi vrsticami, vrstico s to črko pregledal, shranil drugo Veliko črko v tej vrstici in tako nadaljeval

(1.3) Lep prispevek na temo razširitve pomena operatorjev: $-=$ za brisanje znakov iz niza:

```
for i in range(0, len(b),1):
   if b[i] != "A" or b[i] != "B" or  b[i] != "C" ... or b[i] != "Z":
      b −= b[i] # v vsaki vrstici naj bi ostali po dve veliki črki angleške abecede
```

Še en podoben primer:

```
for x in range(0, len(a) + 1, 1):
   if a[x] != 0 and a[x] != 1:
      a = a − a[x]
      x −= x
```

Pri nekem drugem tekmovalcu pa najdemo tole razširitev operatorja ||:

```
int[] array = list[i].indexOf("A" || "B" || ...|| "Z" || "W" || "Q" || "Y");
```

Še en podoben primer:

```
if (vrednost == 0 || 1)
```

(1.3) Še en nov izraz za spremenljivke:

> Po vrsticah preberemo podani file `pot.txt` in ga shranimo v neznanko x
> naj bo ta vrstica prazna neznanka y, tipa list

(1.3) Iz ene od rešitev:

```
// Ti dve spremenljivki sta nujni za iskanje poti po kroničnem zaporedju, saj je
// lastChar zadnja črka v prejšnjem navodilu, newChar pa je nova prva črka v
// trenutnem navodilu.
```

„Kronično zaporedje“ se sliši kot nov medicinski fenomen — zanimivo vprašanje je, kaj je hujše, kronično zaporedje ali akutno…

(1.3) Pogumen način za preverjanje, če je nek znak velika črka:

```
if (a == 'ABCDEFGHIJKLMNOPQRSTUVWXYZ') // če je črka velika, jo na doda v navodila.
  strcat(navodila, a);
```

(1.3) Rešitev, ki se ji ne da ukvarjati s podrobnostmi branja vhodne datoteke:

```
while (/* scaner.readshit */) {
```

Še en podoben primer:

```
podatki = int() # sprejme stavke z navodili. Vsak stavek je samostojna enota. stavek = s
```

(1.3) Pri tej nalogi smo prejeli tudi najdaljšo rešitev letos — dolga je kar 294 vrstic. Večina stvari v programu je razmnoženih po 25-krat, za vsako veliko črko angleške abecede po enkrat (razen črke X, ki jo vztrajno ignorira).

(1.4) Zakaj bi napisali `if (c == '0' || c == '1')`, če lahko namesto tega uporabimo zanko in napišemo devet vrstic kode:

```
char dovoljeni_znaki[] = { '0', '1' };
// poišče, če je c številka
for (int i = 0; i < dovoljeni_znaki.length(); i++)
{
  if (c.equals(dovoljeni_znaki[i]))
  {
    return true;
  }
}
```

Tekmovalec je na dobri poti, da postane zelo produktiven industrijski programer :)

(1.5) Tale zanka verjetno poskuša prebrati istoležne celice na vseh straneh, vendar ker pozabi povečevati številko strani, je njen učinek v resnici ta, da bere eno in isto celico, vse dokler se ta ne okvari:

```
while (Beri(i, c) != −1)
   vsota += Beri(i,c);
```

(1.5) Nekaj dobrih tipkarskih napak v tej rešitvi:

> Algoritem Beri2 bo prebral vrednost na ženjenem naslovu. [. . .] Če vrednosti v pomnilniku ni (NULL), bo podprogram poskusil rekunstruirati. Po rekunstruaciji pa se vrednost vrne glede na pravila, (1 (ena) ali 0 (nič)).

Rekonstrukcijo dosledno piše z u namesto o, čeprav je lahko pravilni zapis večkrat prebral v besedilu naše naloge. . .

(1.5) Rešitev z zanašanjem na nadnaravne sile:

```
import godFunction
call.godFunction
```

(1.5) Cela rešitev nekega tekmovalca (vključno s trojnimi pikami):

```
def Beri2(a, b):
. . .
def Pisi2(c, d, e):
. . .
Če pride do sodega števila napak v istoležnih stranicah,
program ne bo zaznal napake
```

Tale program bo tako ali tako zaznal bolj malo napak :)

(1.5) Rešitev z globokim humanističnim čutom:

```
int Beri2(int stran, int naslov){ // ker smo predpostavili, da uporabnik ve, kaj mora
   // vpisati za naslov, lahko vseeno pride do zmote, saj je ČLOVEK in bi lahko
   // spremenil celotno kontrolno stran
```

(1.5) Rešitev z nezaupanjem v dvodimenzionalne tabele:

> Do napak bo prišlo zaradi neurejenega sistema, boljše bi bilo, če bi bile strani in naslovi skupaj pod enim arrayem in ne narazen.

Še bolje bi bilo, če bi on za branje preprosto klical našo funkcijo Beri in se ne bi ukvarjal s tem, ali za njo tiči array ali kaj drugega. . .

(1.5) Eden od tekmovalcev ni mogel natipkati znakov [in {, zato je namesto njih uporabljal pripadajoče zaklepaje:

```
public static void main(String ]]args) }
⋮
int tabela ]] ]] = new int ]j] ]o];
```

(1.*) Eden od tekmovalcev je na konec vsake od rešitev dodal po en odstavek naključno generiranih obscenosti, pridobljenih s pythonovim modulom madlibs. V biltenu jih raje ne bomo objavljali, radovednim bralcem pa priporočamo, naj si modul prenesejo z githuba.[11]

[11] https://github.com/mozai/python-madlibs/blob/master/filthy.json

(2.1) Tale očitno ni bil zadovoljen s funkcijo abs iz standardne knjižnice in si je napisal svojo:

```
inline int abs(int a) // Absolutna funkcija
{
  if (a < 0)
    return −a;
  return a;
}
```

(2.1) Rešitev s sprejemljivimi tveganji:

> Postopek v veliki verjetnosti deluje, saj preverjam tudi, kam pade naslednja kocka, in s tem lažje preživim.

Njegova rešitev sicer čaka, da se tik nad njim pojavi kocka, in šele takrat odskoči; pri tem pogleda še, kam bo padla naslednja kocka, da se lažje odloči, ali bi odskočil levo ali desno.

(2.2) Stroga podatkovna struktura:

```
struct zid {
  float tezisce;
  int zacetek, konec;
}
strict zid obzidje*;
```

(2.2) Rešitev za ljubitelje sovražnega govora:

```
if (Y == n − 1) Console.Write("Smrdljivi tronjanci so se predali!");
```

(2.4) Rešitev s popolnoma zgrešeno predpostavko:

```
/* Program temelji na predpostavki, da nabor ima lastnost (c) v primeru,
   da vseh 5 bitov zavzamejo vse vrednosti (0 in 1). */
```

S tem je hotel reči, da za vsak bit obstaja neka peterica, v kateri je ta bit prižgan, in neka peterica, v kateri je ugasnjen. Ni težko najti protiprimera — če imamo v našem naboru peterice 10000, 01000 in 11111 (ostalih sedem peteric je lahko poljubnih), je ta pogoj izpolnjen, lastnosti (c) pa nima.

Najti se dá tudi protiprimer v obratno smer: nabor 00000, 00001, 00011, 00100, 00110, 00111, 01001, 01100, 01101, 01111 ima lastnost (c), čeprav je najbolj levi bit v vseh petericah ugasnjen.

(2.4) Impresivno zapleten način, kako izračunati število 1:

```
if (i + 2**s) − (i + 2**s − 1) in peterice:
```

(V resnici je verjetno hotel izračunati, kaj se zgodi, če v peterici i (5-bitno celo število) zamenjamo dva sosednja bita.)

(2.4) Presenetljivo veliko tekmovalcev je imelo težave s tem, kako zamenjati dva elementa tabele:

```
c[g] = c[g + 1]; c[g + 1] = c[g];
```

In še eden z zanimivim pogledom na operator ++ (če si v spodnji kodi namesto i++ mislimo i + 1, postane pravilna):

```
strcpy(kopija_niza, niz);
kopija_niza[i] = niz[i++];
kopija_niza[i++] = niz[i];
```

(2.5) Rešitev z izogibanjem stavku **break**:

```
int n = 500;
:
for (i = 0; i <= 299 && n == 500; i++)
   if (avtomobili[i].p == false)
      n = i;
```

(2.5) Zelo nenavaden pristop k računanju hitrosti:

```
hitr := (cas − casvoz[i]) mod 622;
```

(2.5) Prijetno širok pogled na nize:

> Zunaj procedure deklariramo dva niza, tipa string in tipa **int**.

Pogled na izvorno kodo njegove rešitve sicer pokaže, da je besedo „niz“ tukaj uporabljal v pomenu “array” in je imel en array stringov in en array **int**ov.

(3.1) Tale predrzna rešitev se ni trudila šteti kvadratov in pravilno označenih otokov, ampak je kar predpostavila, da je polovica otokov pravilno označenih, število kvadratov pa je za ena manjše:

```
bw.write(Integer.toString(stMorij));
bw.newLine();
bw.write(Integer.toString(stOtokov));
bw.newLine();
bw.write(Integer.toString(stOtokov / 2));
bw.newLine();
bw.write(Integer.toString(stOtokov / 2 − 1));
```

Najbrž ne bo nikogar presenetilo, da je ta program dobil 0 točk.

(3.1) Besedilo naloge zagotavlja, da bo v 60 % testnih primerov veljalo $w \leq 20$ in $h \leq 20$. Tale tekmovalec si je to zagotovilo razlagal zelo narobe: po tistem, ko prebere w in h iz vhodne datoteke, ju z verjetnostjo 60 % „popravi“ tako, da sta manjša ali enaka 20:

```
int random_number = random.nextInt(10);
if (random_number < 6) { // 60 %
   if (width > 20) width = 20;
   if (height > 20) height = 20;
}
```

(3.3) Še ena rešitev z neobičajnim odporom do stavka **break**:

```
while (head > tail) {
   i = *tail++;
   if (i <= 0) goto done;
   ⋮
}
done:;
```

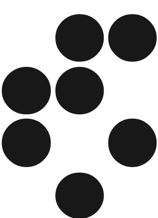
(3.4) Nekdo ni bil zadovoljen s sortom iz standardne knjižnice in si je napisal svoj bubble sort:

```
private static void sort(long[] arr) { // the fuck, Arrays.sort bi mogu delat
```

Težava njegove rešitve je sicer bolj v tem, da uporablja požrešni algoritem; pri vsakem trgovcu pregleduje ponudbe kmetov v naraščajočem vrstnem redu in posamezno ponudbo sprejme, če s tem še ne bi presegel nosilnosti ladje trenutnega trgovca. Ta algoritem pa pri tej nalogi ne najde vedno optimalne rešitve.

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.

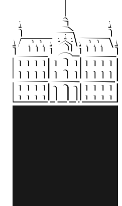
Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi 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

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.

Ministrstvo za izobraževanje, znanost in šport

Ministrstvo za izobraževanje, znanost in šport opravlja upravne in strokovne naloge na področjih predšolske vzgoje, osnovnošolskega izobraževanja, osnovnega glasbenega izobraževanja, nižjega in srednjega poklicnega ter srednjega strokovnega izobraževanja, srednjega splošnega izobraževanja, višjega strokovnega izobraževanja, izobraževanja otrok in mladostnikov s posebnimi potrebami, izobraževanja odraslih, visokošolskega izobraževanja, znanosti, ter športa.

SREBRNI POKROVITELJ

Quintelligence
Obstoječi informacijski sistemi podpirajo predvsem procesni in organizacijski nivo pretoka podatkov in informacij. Biti lastnik informacij in podatkov pa ne pomeni imeti in obvladati znanja in s tem zagotavljati konkurencne prednosti. Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja. IKT (informacijsko-komunikacijska tehnologija) je postavila temelje za nemoten pretok in hranjenje podatkov in informacij. S primernimi metodami je potrebno na osnovi teh informacij izpeljati ustrezne analize in odločitve. Nivo upravljanja in delovanja se tako seli iz informacijske logistike na mnogo bolj kompleksen in predvsem nedeterminističen nivo razvoja in uporabe metodologij. Tako postajata razvoj in uporaba metod za podporo obvladovanja znanja (knowledge management, KM) vedno pomembnejši segment razvoja.

Podjetje Quintelligence je in bo usmerjeno predvsem v razvoj in izvedbo metod in sistemov za pridobivanje, analizo, hranjenje in prenos znanja. S kombiniranjem delnih — problemsko usmerjenih rešitev, gradimo kompleksen in fleksibilen sistem za podporo KM, ki bo predstavljal osnovo globalnega informacijskega centra znanja.

Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja.

BRONASTI POKROVITELJ

Rtk