13. tekmovanje ACM v znanju računalništva

Institut Jožef Stefan, Ljubljana, 24. marca 2018

Bilten

Bilten 13. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2019

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Matija Grabnar, Tomaž Hočevar, Vid Kocijan, Jurij Kodre, Filip Koprivec, Mitja Lasič, Matjaž Leonardis, Borut Lesjak, Matija Lokar, Mark Martinec, Polona Novak, Jure Slak, Jasna Urbančič, Patrik Zajec, Janez Brank.

Tisk: Collegium Graphicum, d. o. o.
Naklada: 170 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 — Kat176ožni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

37.091.27:004(497.4)
004.42(497.4)(079.1)(082)

TEKMOVANJE ACM v znanju računalništva (13 ; 2018 ; Ljubljana)
Bilten / 13. tekmovanje ACM v znanju računalništva, Ljubljana, 24. marca 2018 ; [avtorji nalog Nino Bašić . . . [et al.] ; uredil Janez Brank]. — Ljubljana : Institut Jožef Stefan, 2019

ISBN 978-961-264-145-0
1. Bašić, Nino 2. Brank, Janez, 1979–
298541568

KAZALO

Struktura tekmovanja 5

Nasveti za 1. in 2. skupino 7
Naloge za 1. skupino 11
Naloge za 2. skupino 16
Navodila za 3. skupino 21
Naloge za 3. skupino 24
Naloge šolskega tekmovanja 29
Neuporabljene naloge iz leta 2016 33

Rešitve za 1. skupino 47
Rešitve za 2. skupino 55
Rešitve za 3. skupino 67
Rešitve šolskega tekmovanja 83
Rešitve neuporabljenih nalog 2016 90

Nasveti za ocenjevanje
in izvedbo šolskega tekmovanja 144

Rezultati 148
Nagrade 155
Šole in mentorji 156
Off-line naloga: Risanje s pravokotniki 158

Univerzitetni programerski maraton 161

Anketa 164
Rezultati ankete 169
Cvetke 177

Sodelujoče inštitucije 185
Pokrovitelji 189

STRUKTURA TEKMOVANJA

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

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

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

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

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, pa tudi z dokumentacijo raznih programskih jezikov, ki je 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. 21–23 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.

Od lani objavljamo v biltenu rešitve v C++17, za prvo skupino pa tudi v pythonu, ker precejšen delež tekmovalcev v tej skupini še ne pozna nobenega drugega jezika.

Poleg tekmovanja v znanju računalništva smo organizirali tudi tekmovanje v off-line nalogi, ki je podrobneje predstavljeno na straneh 158–160.

Podobno kot v zadnjih nekaj letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 19. januarja 2018. 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. 144–147). 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 364 tekmovalcev s 33 šol (ena od njih je bila osnovna, 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 želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku (npr. kopiraj in prilepi v Notepad in shrani v datoteko). **Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave in želiš, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici.**

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši idejo, na kateri temelji tvoja rešitev. Če ni v nalogi drugače napisano, lahko tvoje rešitve predpostavljajo, da so vhodni podatki brez napak (da ustrezajo formatu in omejitvam, kot jih podaja naloga). Zaželeno je, da so tvoje rešitve poleg tega, da so pravilne, tudi učinkovite; bolj učinkovite rešitve dobijo več točk (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

1. Collatz++

Collatzevo zaporedje je zaporedje naravnih števil, v katerem je vsak člen izračunan iz prejšnjega po naslednjem pravilu: če je prejšnji člen — recimo mu n — deljiv z 2, je naslednji člen enak $n/2$, sicer pa je naslednji člen enak $3n + 1$. Ustavimo se, ko pridemo do števila 1.[1]

Konkretno zaporedje, ki ga na ta način dobimo, je odvisno od tega, s katerim številom začnemo. Na primer, Collatzevo zaporedje z začetnim členom 15 je:

$$15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1.$$

Gojmir je napisal funkcijo, ki poišče takšno število k z območja od vključno a do vključno b, da Collatzevo zaporedje z začetkom k nekje doseže maksimalno vrednost. Npr. na območju od 30 do 50 je takšno število $k = 31$, saj je pripadajoče zaporedje:

$$31, 94, 47, 142, 71, 214, \ldots, 6154, 3077, \mathbf{9232}, 4616, 2308, 1154, \ldots, 5, 16, 8, 4, 2, 1.$$

[1] Vprašanje, ali na ta način res vedno pridemo do 1, je eden od vélikih nerešenih matematičnih problemov (Collatzeva domneva); vsaj za „majhne“ n (do 10^{20}) pa je bilo že eksperimentalno preverjeno, da ta domneva drži.

To zaporedje doseže maksimum 9232. Če izberemo kot začetni člen katerokoli drugo celo število z območja od 30 do 50, tega ne bomo presegli.

Kmalu pa je opazil, da ima njegova funkcija pomanjkljivost, saj $k = 41$ prav tako doseže enak maksimum. Ker je izčrpan in ne more več programirati, potrebuje tvojo pomoč. **Napiši podprogram** oz. funkcijo PoisciVse(a, b), ki poišče *vsa* takšna števila. (Tvoja funkcija lahko rezultat vrne v obliki seznama, tabele, vektorja ali česa podobnega, lahko pa ga izpiše na standardni izhod ali v kakšno datoteko, karkoli ti je lažje.)

Primer: PoisciVse(30, 50) mora najti števila 31, 41 in 47 — če se namreč Collatzevo zaporedje začne z enim od teh treh števil, bo sčasoma doseglo vrednost 9232; in te vrednosti ne bo nikoli doseglo (ali preseglo), če se začne s katerimkoli drugim številom od 30 do 50.

2. Alfa Bravo

Palček Godrnjavček je postal vodja tajne službe, ki skrbi za varnost Sneguljčice, njegova naloga pa je, da sprejema podatke agentov na terenu in jim daje navodila, kako naj ukrepajo. Da bi se palčki zaščitili pred napakami pri sporazumevanju, so uvedli fonetično abecedo, ki posamezni črki priredi besedo.

Črka	Beseda	Črka	Beseda	Črka	Beseda
A	ALFA	J	JULIET	S	SIERRA
B	BRAVO	K	KILO	T	TANGO
C	CHARLIE	L	LIMA	U	UNIFORM
D	DELTA	M	MIKE	V	VICTOR
E	ECHO	N	NOVEMBER	W	WHISKY
F	FOXTROT	O	OSCAR	X	X-RAY
G	GOLF	P	PAPA	Y	YANKEE
H	HOTEL	Q	QUEBEC	Z	ZULU
I	INDIA	R	ROMEO		

Godrnjavček te prosi, da **napišeš program**, ki bo znal vnešeno besedilo odkodirati. Kot vhod torej dobi zaporedje kodnih besed iz gornje tabele (ločene bodo s presledki; če ti je lažje, pa lahko predpostaviš, da je vsaka beseda v svoji vrstici), izpiše pa naj pripadajoče zaporedje črk. Pri tem naj bo odporen tudi na manjše napake pri kodiranju: namesto prave kodne besede se lahko v vhodnih podatkih pojavi taka beseda, ki se od prave razlikuje v največ enem znaku (je pa zagotovo še vedno enako dolga kot prava kodna beseda). Tako se lahko na primer zgodi, da namesto besede `LIMA` dobimo `RIMA` ali `LINA` ali `LQMA` in tako naprej, vse te besede pa ravno tako predstavljajo črko L.

Tvoj program lahko podatke bere s standardnega vhoda in piše na standardni izhod, lahko pa namesto tega bere iz datoteke `vhod.txt` in piše v datoteko `izhod.txt` (karkoli ti je lažje).

Primer: če na vhodu dobimo

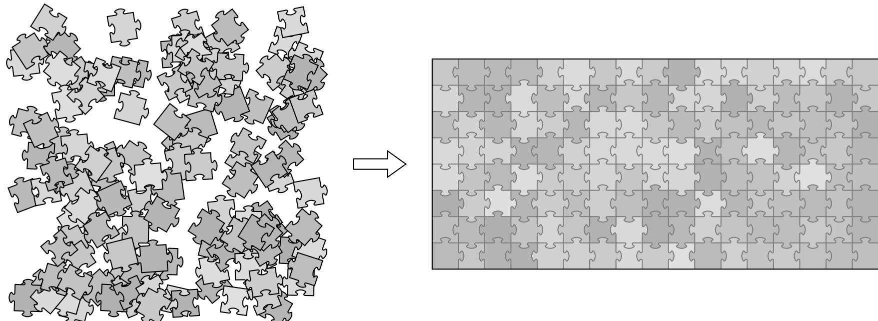
```
SIERRA RODEO ERHO CHARLIF MOVEMBER OSQAR
```

moramo izpisati:

```
SRECNO
```

3. Sestavljanka

Gojmir je na polici opazil škatlo s sestavljanko, ki ima 136 koščkov (glej sliko).



Gojmir ve, da sestavljanko dobimo tako, da sliko razrežemo s (skoraj) vodoravnimi in (skoraj) navpičnimi črtami, tako da dobimo (skoraj) kvadratne koščke. Opazil pa je, da slika, ki jo tvori sestavljanka, v resnici ni kvadratna. Takole čez palec je ocenil, da je razmerje med višino in širino slike približno 3 : 7.

Koliko koščkov v resnici pride po višini in koliko po širini? Pravokotnik iz 136 koščkov bi lahko bil oblike 1×136 ali 2×68 ali 4×34 ali 8×17 in tako naprej. Vprašanje je torej, pri katerem od teh pravokotnikov je razmerje med višino in širino najbližje tistemu, ki ga je ocenil Gojmir.

Napiši podprogram oz. funkcijo Sestavljanka(steviloKosckov, razmerje), ki bo to izračunal v splošnem primeru, za poljubno število koščkov in poljubno razmerje. Gojmir določi razmerje višine in širine dokaj natančno, vendar ne povsem natančno.

Primer: če pokličemo Sestavljanka(136, 0.428) (to je primer od zgoraj — 0,428 je približno 3/7), mora funkcija vrniti ali izpisati rezultat 8×17 (8 vrstic, 17 stolpcev). Pri tej sestavljanki je razmerje med višino in širino enako $8/17 \approx 0{,}471$, kar je od vseh možnih pravokotnih sestavljank s 136 koščki še najbližje iskanemu razmerju 0,428.

4. Pisalni stroj

Imamo pisalni stroj, ki deluje tako, da se ob vsakem izpisu znaka nujno pomakne za eno mesto v desno. (Eden od teh znakov je tudi presledek, ki sicer ne izpiše ničesar, nas pa ravno tako premakne za eno mesto v desno.) Poleg tega obstaja tudi tipka za pomik na začetek trenutne vrstice (*carriage return*). Drugih možnosti za premik v levo nimamo. Radi bi napisali neko vrstico tako, da bo ponekod na istem mestu več črk ena čez drugo (npr. da kaj podčrtamo z znakom `_`, prekrižamo z znakom `x`, dodamo kakšno naglasno znamenje nad črko in podobno). Recimo, da bo vrstica na koncu dolga n znakov, pri čemer hočemo na mestu i natipkati a_i znakov. **Opiši postopek**, ki iz teh podatkov (torej števil $n, a_1, a_2, \ldots, a_n$) izračuna, kolikšno je najmanjše število pritiskov na tipke (med pritiske šteje tudi pomik na začetek vrstice), ki jih potrebujemo, da natipkamo to vrstico. Pri tem ni pomembno, kje v

vrstici se nahajamo na koncu tipkanja. Tvoja rešitev naj bo čim bolj učinkovita, da bo delovala hitro tudi v primerih, ko so n in števila a_i zelo velika.

Primer: recimo, da hočemo natipkati takšno vrstico dolžine $n = 6$:

`äbç̂ d̲e`

Tabela a bi bila torej v tem primeru takšna:

i	1	2	3	4	5	6
a_i	2	1	3	0	2	1

Minimalno potrebno število pritiskov na tipke je v tem primeru 16. Primerno zaporedje (ni pa edino) tipk je: `a`, `b`, `c`, presledek, `d`, `e`, pomik na začetek, ¨, presledek, vejica, presledek, `_`, pomik na začetek, presledek, presledek in `^`.

5. Brzinomer

V avtomobilu za prikaz trenutne hitrosti vozila skrbi instrument (brzinomer), ki je lahko digitalen (prikazuje številke) ali pa analogen (fizični kazalec instrumenta se pomika po skali in s svojo lego kaže izmerjeno hitrost). A tudi prikazovalniki s kazalcem in skalo so doživeli svojo prenovo in niso več preprosti analogni merilniki neke fizikalne veličine, ampak gre za prikazovalnike, kjer je kazalec pritrjen na os miniaturnega koračnega elektromotorčka, pomike tega pa krmili avtomobilski računalnik glede na hitrost, ki jo izmerijo tipala hitrosti.

Naš prikazovalnik ima skalo v obsegu od 0 km/h do 250 km/h. Koračni motorček lahko pomika kazalec v vsakem koraku le za 1 km/h navzgor ali navzdol, ali pa kazalca ne premakne. Če je kazalec že v najnižji legi, ukaz za pomik navzdol ne stori ničesar (kazalec ostane v najnižji legi, t.j. kaže na 0 km/h na skali). Podobno tudi pri najvišji legi: ukaz za pomik navzgor ne stori ničesar, kazalec ostane pri najvišji legi, t.j. kaže na 250 km/h na skali.

Premik koračnega motorčka za en korak (oz. kazalca, ki je pritrjen nanj, za 1 km/h) je sicer precej hiter, vendar vseeno zahteva določen čas. Da ne bi avtomobilski računalnik skušal premikati kazalca hitreje, kot to koračni motorček zmore, skrbi ura, ki se proži nekajkrat v sekundi in jamči, da je takrat koračni motorček pripravljen sprejeti nov ukaz za pomik za en korak.

Napiši podprogram oz. funkcijo Premik, ki jo bo ura periodično klicala (približno stokrat v sekundi, točna vrednost intervala ni pomembna). Kot argument bo funkcija ob vsakem klicu prejela celo število med 0 in 300, ki predstavlja trenutno hitrost avtomobila v km/h. Glede na svoje védenje o trenutni legi kazalca brzinomera naj funkcija vrne vrednost -1, $+1$ ali 0, kar bo povzročilo pomik kazalca za en korak navzdol ali navzgor ali pa pomika v tem časovnem intervalu ne bo. Funkcija naj skrbi, da bo lega kazalca brzinomera čim bolj verno sledila dejanski hitrosti avtomobila:

```
function Premik(Hitrost: integer): integer;  { v pascalu }
int Premik(int hitrost);                     /* v C/C++ in podobnih jezikih */
def Premik(hitrost): ...                     # v pythonu; „hitrost“ bo tipa int
```

Upoštevaj, da se lahko hitrost avtomobila med dvema zaporednima klicema tvoje funkcije spremeni tudi za več kot 1 km/h. V tem primeru bo sicer kazalec merilnika lahko za krajši čas zaostajal s pravim prikazom, a mora pravo lego po najboljših močeh čim prej ujeti. Pri hitrostih nad prikazovalnim območjem (t.j. nad 250 km/h) naj kazalec tiči v svoji najvišji legi.

Po potrebi lahko deklariraš poljubne globalne spremenljivke in jim tudi določiš začetne vrednosti.

Upoštevaj tudi, da ob zagonu avtomobilskega računalnika ne vemo točno, kje je obtičal kazalec brzinomera — lahko bi se npr. zgodilo, da je bil motor avtomobila (in računalnik) ugasnjen, še preden se je avtomobil dokončno ustavil na domačem dvorišču. Da zagotovimo znano lego kazalca, si lahko pomagamo z informacijo iz drugega odstavka, ki zagotovi, da z 250 koraki navzdol zagotovo dosežemo spodnjo lego (torej prikaz 0 km/h) ne glede na začetni položaj kazalca. Če ti to dela preglavice, lahko zapišeš, da tvoj program predpostavlja ob zagonu ničelno lego kazalca, a za to boš prikrajšan za polovico točk.

NALOGE ZA DRUGO SKUPINO

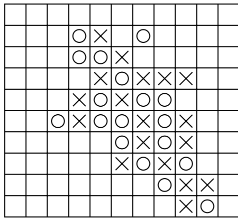
Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del prek računalnika. Vse te možnosti so enakovredne. Odgovore, oddane prek računalnika, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

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 želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku (npr. kopiraj in prilepi v Notepad in shrani v datoteko). **Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave in želiš, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici.**

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši idejo, na kateri temelji tvoja rešitev. Če ni v nalogi drugače napisano, lahko tvoje rešitve predpostavljajo, da so vhodni podatki brez napak (da ustrezajo formatu in omejitvam, kot jih podaja naloga). Zaželeno je, da so tvoje rešitve poleg tega, da so pravilne, tudi učinkovite; bolj učinkovite rešitve dobijo več točk (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

1. Križci in krožci

Imamo igralno ploščo z mrežo m vrstic in n stolpcev. V mrežo na prazna polja rišeta izmenično dva igralca: en igralec postavlja križce, drugi pa krožce. Zmaga tisti, ki prvi postavi 5 svojih znakov skupaj, torej da je 5 enakih znakov zaporedno v isti vrstici, stolpcu ali diagonali.



Primer mreže, v kateri je zmagal igralec, ki postavlja krožce.

Napiši podprogram (funkcijo) ali del programa, ki za dano tabelo $m \times n$ znakov ugotovi, kdo je zmagovalec oziroma, da ni zmagovalca, če ni nihče postavil 5 svojih znakov skupaj v vrsto, stolpec ali diagonalo. Vsak element tabele je eden od znakov `'x'` (križec), `'o'` (krožec) in `' '` (presledek, ki predstavlja prazno polje). Zagotovljeno je, da je zmagovalec lahko največ en igralec (lahko pa tudi ni zmagovalca).

2. Popravljanje testov

Profesor Golob pazi razred n učencev, ki pišejo test iz matematike. Na robu katedra ima označeno mesto, kamor morajo učenci odložiti test, ko zaključijo z delom. Test vedno odložijo na vrh kupa, če ta obstaja. Po tem, ko je na kupu vsaj en test, začne prof. Golob popravljati teste. Nov test vedno vzame z vrha in po popravljanju ga odloži drugam. Primer zaporedja oddajanja in popravljanja bi lahko predstavili z nizom `ABCXDXXRXXHXQX`, kjer `X` pomeni, da je profesor vrhnji test vzel s kupa, druge črke pa, da je neka oseba test odložila na kup. V zgornjem primeru bi prof. Golob po vrsti popravljal teste `CDBRAHQ`.

Učencem je obljubil, da jim bo teste vrnil v enakem vrstnem redu, kot so jih oddajali. Zgolj iz zaporedja `CDBRAHQ` vrstnega reda oddajanja ne more rekonstruirati, verjame pa, da bi lahko pravilno rekonstruiral vrstni red, če bi vedel še, koliko testov je bilo na kupu vsakič, ko je vzel test s kupa. Če bi torej imel zapis oblike `???C?DB?RA?H?Q`, ki za vsak `X` v zgornjem zapisu pove, kateri test je vzel s kupa, znak `?` pa predstavlja, da je nek učenec oddal test, bi lahko rekonstruiral niz `ABCXDXXRXXHXQX`.

Napiši program ali funkcijo, ki sprejme niz oblike `???C?DB?RA?H?Q` in vrne ali izpiše niz oblike `ABCXDXXRXXHXQX`. Če vhodni niz ni mogel nastati iz nobenega zaporedja veljavnih oddaj testov, potem to tudi sporoči (vhodni niz štejemo za neveljavnega tudi, če zaporedja oddaj iz njega ni mogoče rekonstruirati). Zaželeno je, da je tvoja rešitev čim bolj učinkovita, torej da bi delovala hitro tudi za velike n (npr. $n = 10^8$).

Nekaj primerov:

Vhod: `?A`
Izhod: `AX`

Vhod: `???`
Izhod: *neveljaven vhod*

Vhod: `???C?DB?RA?H?Q`
Izhod: `ABCXDXXRXXHXQX`

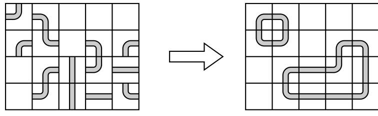
Vhod: `??QWE?`
Izhod: *neveljaven vhod*

Vhod: `?????DC??GBAA??WQBA`
Izhod: `ABAACDXXBGXXXXQWXXXX`

3. Cevi

Na pravokotni mreži dimenzij $v \times s$ (pri čemer sta v in s manjša ali enaka 100) je vsaka celica prazna ali pa vsebuje enega od dveh tipov cevi. Prvi tip je zavita cev (tip L), ki povezuje dve sosednji stranici celice. Drugi tip je ravna cev (tip I), ki povezuje nasprotni stranici celice. Z eno potezo lahko zavrtimo celico (oz. cev v njej) za 90 stopinj v poljubno smer. S takimi potezami želimo povezati vse cevi tako, da ne bo nikjer kakšnega odprtega konca cevi. **Opiši** in dobro utemelji **postopek**, ki izračuna najmanjše število potez oz. ugotovi, da to ni mogoče.

Primer: mrežo na spodnji sliki lahko primerno uredimo v 11 potezah.



4. Palačinke

Mihec je navdušen nad palačinkami, zato si jih navadno speče ogromno. Poleg palačink pa ga skoraj še bolj navdušuje prelaganje le-teh. Prelaga jih na poseben način. Pred vsakim prelaganjem si izmisli poljubno naravno število k (med 1 in številom palačink), nato prime kup, sestavljen iz zgornjih k palačink, ga obrne na glavo in nanj (v enakem vrstnem redu, kot so bile do sedaj) postavi ostanek kupa. Tokrat je malo pretiraval v količni palačink, pa še vsaka je malo drugačna, zato te je prosil za pomoč pri simuliranju igre.

Opiši postopek ali napiši program (karkoli ti je lažje), ki simulira potek obračanja palačink, pri čemer palačinke zaradi preprostosti označimo z naravnimi števili od 1 do n. V prvi vrstici standardnega vhoda se nahaja naravno število $n > 1$, ki predstavlja število palačink. Na začetku so palačinke zložene na kupu tako, da je na dnu palačinka številka 1, potem pa si po vrsti sledijo palačinke do vrhnje, ki je označena s številko n. S standardnega vhoda nato do konca beri cela števila (od vključno 0 do vključno n), ki so zapisana vsaka v svoji vrstici. Če je to število večje od 0, potem ustrezno obrni Mihčev kup palačink, če pa je to število enako 0, potem Mihca zanima celotno zaporedje palačink na kupu od spodaj navzdol in ga ustrezno izpiši.

Zaželeno je, da je tvoj postopek čim bolj učinkovit. Pri tem lahko predpostaviš, da so števila k v povprečju majhna v primerjavi z n in da bo zahtev po izpisu celotnega zaporedja malo v primerjavi s številom obračanj.

Primer vhoda:

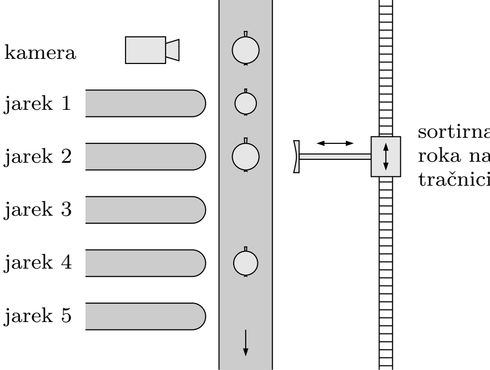
```
5
2
0
3
0
2
4
1
0
```

Pripadajoči izhod:

```
5 4 1 2 3
3 2 1 5 4
4 1 2 3 5
```

5. Jabolka

Imamo tekoči trak, po katerem prihajajo jabolka. Na začetku traku imamo kamero, ki prepoznava kvaliteto jabolka in ga oceni s številko od 1 do 5. Za kamero je 5 jarkov in sortirna roka, ki lahko jabolko odrine v jarek, pred katerim se nahaja. Vsak jarek pelje v drug zabojček, v katerem se nabirajo sortirana jabolka.



Tekoči trak se premika v diskretnih korakih, vedno po en jarek naprej. V enem takem koraku se jabolka na traku premaknejo za en jarek naprej; tisto jabolko, ki je bilo v prejšnjem koraku pred kamero, se premakne do prvega jarka; pred kamero pa lahko pride naslednje jabolko (ni pa nujno, ker so lahko med jabolki na traku tudi presledki — glej sliko).

Napiši program, ki teče v neskončni zanki in skrbi za to, da sortirna roka odrine vsako jabolko v pravi jarek (torej v tisti jarek, čigar številka je enaka oceni jabolka). Predpostavi, da so že na voljo naslednji podprogrami oz. funkcije, ki jih lahko tvoj program kliče za upravljanje s trakom:

- **int** PremakniTrak() premakne trak za en jarek naprej in vrne celo število z oceno novega jabolka (od 1 do 5), ki je zdaj (po tem premiku) pred kamero. Če pred kamero ni jabolka, funkcija vrne 0. Med dvema zaporednima klicema funkcije PremakniTrak je trak pri miru.

- **void** PremakniRoko(**int** k) premakne sortirno roko pred jarek k (parameter k mora biti od 1 do 5).

- **void** SproziRoko() sproži iztegovanje roke. Če je pred roko jabolko, se bo zvalilo v jarek. Če pred roko ni jabolka, se ne bo zgodilo nič hudega. Podprogram se vrne šele, ko je roka spet v skrčenem položaju in pripravljena na morebitni naslednji premik.

Še deklaracije v drugih programskih jezikih:

```
{ V pascalu: }
function PremakniTrak: integer;
procedure PremakniRoko(k: integer);
procedure SproziRoko;

# V pythonu:
def PremakniTrak(): ...      # vrne int
def PremakniRoko(k): ...     # parameter „k“ mora biti int
def SproziRoko(): ...
```

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše. Programi naj berejo vhodne podatke s standardnega vhoda in izpisujejo svoje rezultate na standardni izhod. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih podatkov. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih podatkov, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih podatkov.

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.

Po prijavi se bo pred teboj pojavilo namizje XFCE. Nekaj morda uporabnih bližnjic je že na namizju, če pa se ti zdi, da bi ti prav prišlo še kakšno orodje, lahko klikneš na ikono miši v zgornjem levem kotu zaslona. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali python, mi pa jih bomo preverili s prevajalniki FreePascal, GNUjevima `gcc` in `g++` 4.7.2 (pozor: ta verzija skoraj v celoti podpira C++11, novejših različic standarda C++ pa ne), prevajalnikom za javo iz JDK 7, s prevajalnikom Mono 2.10 za C# in z interpreterjema za python 2.7 in 3.2. Za delo lahko uporabiš Lazarus (IDE za pascal), `gcc/g++` (GNU C/C++ — command line compiler), `javac` (za javo), Eclipse in druga orodja.

Na spletni strani `http://www.putka.si/tasks/slo_tekme/rtk2018/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi rešitve svojih nalog. Uporabniška imena in gesla za Putko so enaka kot za računalnike.

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 preveč pomnilnika (več kot 250 MB), ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

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

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

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na več testnih primerih; pri vsakem od njih dobi 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk. Izjemi sta druga naloga, kjer je testnih primerov

20 in za pravilen odgovor pri posameznem testnem primeru dobiš 5 točk, in peta naloga, kjer je možno dobiti tudi delne točke za delno pravilne odgovore.

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 M točk.

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.

Primer naloge (ne šteje k tekmovanju)

Napiši program, ki s standardnega vhoda prebere dve celi števili (obe sta v prvi vrstici, ločeni z enim presledkom) in izpiše desetkratnik njune vsote na standardni izhod.

Primer vhoda:

```
123 456
```

Ustrezen izhod:

```
5790
```

Primeri rešitev:

- V pascalu:

```
program PoskusnaNaloga;
var i, j: integer;
begin
  ReadLn(i, j);
  WriteLn(10 * (i + j));
end. {PoskusnaNaloga}
```

- V C-ju:

```
#include <stdio.h>
int main()
{
  int i, j; scanf("%d %d", &i, &j);
  printf("%d\n", 10 * (i + j));
  return 0;
}
```

- V C++:

```
#include <iostream>
using namespace std;
int main()
{
  int i, j; cin >> i >> j;
  cout << 10 * (i + j) << '\n';
}
```

- V pythonu:

```
import sys
L = sys.stdin.readline().split()
i = int(L[0]); j = int(L[1])
print("%d" % (10 * (i + j)))
```

(*Opomba:* namesto `'\n'` lahko uporabimo `endl`, vendar je slednje ponavadi počasneje.)

(*Primeri rešitev se nadaljujejo na naslednji strani.*)

- V javi:

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

public class Poskus
{
   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(10 * (i + j));
   }
}
```

- V C#:

```
using System;

class Program
{
   static void Main(string[] args)
   {
      string[] t = Console.In.ReadLine().Split(' ');
      int i = int.Parse(t[0]), j = int.Parse(t[1]);
      Console.Out.WriteLine("{0}", 10 * (i + j));
   }
}
```

NALOGE ZA TRETJO SKUPINO

1. Buteljke

Skupina prijateljev je sklenila, da si bodo poenostavili življenje in si za rojstni dan ne bodo več podarjali domiselnih daril, saj takšno načrtovanje terja čas in energijo. Od zdaj naprej bo vsak obdarovanec dobil za darilo buteljko vina.

Ker so Gorenjci, so kmalu ugotovili, da lahko buteljko, ki so jo dobili v dar, prihranijo in jo poklonijo ob naslednjem rojstnem dnevu enemu od prijateljev (lahko tudi tistemu, ki jim jo je poklonil). Posamezni človek podari posameznemu prejemniku natanko eno buteljko (če ga sploh obdaruje).

Napiši program, ki izračuna, koliko je minimalno število buteljk, ki jih morajo vsi skupaj kupiti, da bo sistem deloval na dolgi rok, če za vsakega med njimi vemo, kdaj ima rojstni dan in koga vse obdaruje. (Predpostavimo, da z načrtom začnejo ob optimalnem trenutku.)

Da bo prišla rešitev prav tudi skopuhom na drugih planetih, boš pri tej nalogi dobil kot vhodni podatek tudi število dni v letu, recimo mu d. Dnevi so predstavljeni z zaporednimi številkami od 1 do d.

Vhodni podatki: na vhodu dobiš enega ali več testnih primerov. V prvi vrstici je celo število T (zanj velja $1 \leq T \leq 3$). Sledi T testnih primerov, pred vsakim pa je še prazna vrstica.

Posamezni testni primer je opisan takole: v prvi vrstici so tri cela števila, ločena s po enim presledkom: število dni v letu d, skupno število prijateljev n in število vseh obdarovanj k (veljalo bo $1 \leq n \leq d \leq 10^5$ in $1 \leq k \leq 10^6$). Sledi n vrstic s podatki o rojstnih dnevih: i-ta od teh vrstic vsebuje število r_i, to je številka dneva, ko ima oseba i rojstni dan. (Veljalo bo $1 \leq r_i \leq d$. Nikoli se ne bo zgodilo, da bi imela dva človeka rojstni dan na isti dan.) Sledi še k vrstic, od katerih i-ta vsebuje dve različni števili a_i in b_i (zanju velja $1 \leq a_i \leq n$ in $1 \leq b_i \leq n$), ločeni s presledkom; to pomeni, da oseba a_i za rojstni dan obdaruje osebo b_i.

Izhodni podatki: za vsak testni primer iz vhodnih podatkov izpiši po eno vrstico. V tej vrstici izpiši minimalno število buteljk, ki jih morajo (pri tistem testnem primeru) kupiti vsi skupaj; ali pa niz „`Pijanci so med nami!`“ (brez narekovajev), če sistem v tej družbi ne bo deloval.

(*Nadaljevanje na naslednji strani.*)

Primer vhoda:

```
2

100 3 2
53
22
90
1 2
2 3

100 3 3
53
22
90
1 2
2 3
3 1
```

Pripadajoči izhod:

```
Pijanci so med nami!
2
```

Komentar: v vhodnih podatkih sta dva testna primera; v obeh nastopajo trije prijatelji, ki imajo rojstne dneve na 53., 22. in 90. dan v letu. V prvem primeru oseba 1 obdaruje osebo 2, ta pa osebo 3; tu sistem ne deluje. Drugi primer je tak kot prvi, le da poleg tega še oseba 3 obdaruje osebo 1. Zdaj je dovolj, če kupijo dve buteljki in si ju podajajo na veke vekov.

2. Pravokotnik

Podane imamo točke v ravnini. Nekatere so pobarvane modro, nekatere pa rdeče. Zanimajo nas pravokotniki, ki ustrezajo vsem naslednjim pogojem:

- Stranice pravokotnika morajo biti vodoravne ali navpične.
- Nobena modra točka ne sme ležati zunaj pravokotnika (lahko pa ležijo na njegovem robu ali znotraj pravokotnika).
- Nobena rdeča točka ne sme ležati znotraj pravokotnika (lahko pa ležijo na njegovem robu ali zunaj pravokotnika).

Napiši program, ki za dani nabor točk izračuna ploščino največjega takega pravokotnika (ko rečemo „največjega“ pravokotnika, mislimo s tem na tistega z največjo ploščino). Pri tej nalogi bodo testni primeri sestavljeni tako, da vsaj en tak največji pravokotnik gotovo obstaja.

Vhodni podatki: v prvi vrstici sta dve celi števili, m (število modrih točk) in r (število rdečih točk), ločeni s presledkom. Veljalo bo $1 \leq m \leq 10^4$ in $1 \leq r \leq 10^4$. Sledi m vrstic, ki vsebujejo koordinate modrih točk; i-ta od teh vrstic vsebuje dve celi števili x_i in y_i, ločeni s presledkom, to sta koordinati i-te modre točke. Nato sledi še r vrstic, ki na enak način podajajo koordinate rdečih točk. Za koordinate vseh točk velja $|x_i| \leq 10^9$ in $|y_i| \leq 10^9$.

Izhodni podatki: izpiši eno samo celo število, in sicer ploščino največjega pravokotnika, ki ustreza pogojem iz besedila naloge. *Nasvet:* ker je lahko ta ploščina večja od 2^{32}, je koristno pri tej nalogi uporabljati 64-bitne celoštevilske tipe (npr. **long long** v C/C++, **long** v C# in javi, int64 v pascalu).

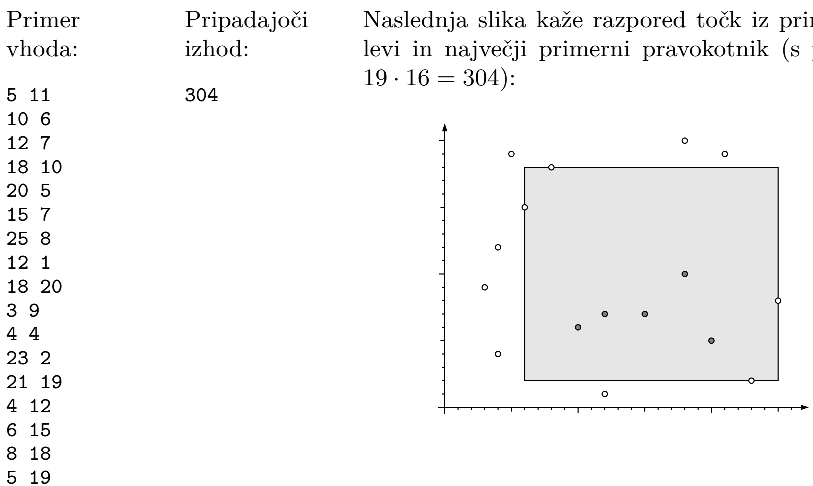
Primer vhoda:

```
5 11
10 6
12 7
18 10
20 5
15 7
25 8
12 1
18 20
3 9
4 4
23 2
21 19
4 12
6 15
8 18
5 19
```

Pripadajoči izhod:

```
304
```

Naslednja slika kaže razpored točk iz primera na levi in največji primerni pravokotnik (s ploščino $19 \cdot 16 = 304$):



3. Tekoči trak

Načrtujemo dolgo tovarniško linijo z n vzporednimi tekočimi trakovi, dolgimi po L metrov. Tekoči trakovi se gibljejo z različnimi hitrostmi, prvi z 1 m/s, sosednji z 2 m/s, naslednji 3 m/s in tako naprej do zadnjega, ki se giblje s hitrostjo n m/s. Izdelek začne svojo pot na začetku prvega traku, po vsakem prevoženem metru (razen po zadnjem) pa lahko izdelek premaknemo na enega izmed sosednjih trakov, tako da se mu hitrost poveča ali zmanjša za 1 m/s. Dodatno imamo na nekaterih dolžinskih odsekih poti hitrostne omejitve, saj morajo izdelek zaznati različni senzorji. **Napiši program**, ki ugotovi, kako naj se izdelek premika po trakovih, da bo prepotoval celotno tovarniško linijo v čim krajšem času, pri čemer mora potovanje začeti in zaključiti na najpočasnejšem tekočem traku (to pomeni, da moramo prvi meter, torej od $x = 0$ do $x = 1$, in zadnji meter, torej od $x = L - 1$ do $x = L$, prevoziti po prvem traku).

Vhodni podatki. V prvi vrstici so podana tri naravna števila, ločena s po enim presledkom: število tekočih trakov n, dolžina linije L in število omejitev m (veljalo bo $1 \leq n \leq 10^9$, $1 \leq L \leq 10^9$ in $1 \leq m \leq 10^3$). Sledi m vrstic, od katerih vsaka vsebuje tri cela števila s_i, e_i in v_i (ločena s po enim presledkom), ki predstavljajo omejitev, da izdelek na območju od točke s_i do e_i (merjeno v metrih od začetka linije) ne sme iti hitreje kot v_i (lahko pa gre točno s hitrostjo v_i). Za vsako od teh omejitev velja $1 \leq v_i \leq n$ in $0 \leq s_i < e_i \leq L$.

V 60 % testnih primerov bo veljalo tudi $n \leq 100$, $L \leq 200$ in $m \leq 200$.

Izhodni podatki: pot izdelka si lahko predstavljamo kot zaporedje trojic oblike l_i d_i t_i, ki pomenijo, da je izdelek prevozil interval od x-koordinate l_i do x-koordinate d_i v celoti po traku t_i. (Seveda mora za vsako tako trojico veljati $0 \leq l_i < d_i \leq L$ in $1 \leq t_i \leq n$.) Tiste izmed teh trojic, pri katerih je $d_i - l_i > 1$, izpiši na izhod, vsako v svojo vrstico, pri čemer naj bodo števila l_i, d_i in t_i ločena s po enim presledkom. Vrstice naj bodo urejene naraščajoče po l_i. Tistih trojic, pri katerih je $d_i = l_i + 1$, pa sploh ne izpisuj.

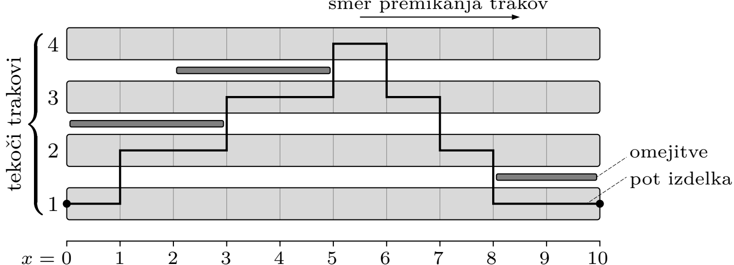
Primer vhoda:

```
4 10 3
0 3 2
8 10 1
2 5 3
```

Pripadajoči izhod:

```
1 3 2
3 5 3
8 10 1
```

Skica gornjega primera:



Komentar: izdelek potuje s hitrostjo 1 m/s na prvem metru, nato potuje dva metra s hitrostjo 2 m/s, nato potuje dva metra s hitrostjo 3 m/s, nato en meter s hitrostjo 4 m/s, nato en meter s hitrostjo 3 m/s, en meter s hitrostjo 2 m/s in dva metra s hitrostjo 1 m/s. Kot zahteva opis naloge, smo na izhod izpisali le tiste od teh korakov, kjer je izdelek potoval vsaj dva metra skupaj po istem traku.

4. Knjige

Na polici imaš od leve proti desni razporejenih več različno visokih knjig. Nekatere boš s police pospravil drugam, da se na njih ne bo nabiral prah. Za preostale knjige na polici pa želiš, da (brez preurejanja) najprej naraščajo po višini, nato pa padajo. Lahko tudi samo naraščajo ali samo padajo. **Napiši program**, ki bo izračunal, kakšno je največje število knjig, ki jih lahko pustiš na polici.

Vhodni podatki: prva vrstica vsebuje število knjig na polici; to je celo število n, za katero velja $1 \leq n \leq 10^6$. V drugi vrstici so naštete s presledki ločene višine knjig, kot si sledijo od leve proti desni. Višine knjig so cela števila med vključno 1 in n. Vse knjige so različnih višin.

V 60 % testnih primerov bo veljalo $n \leq 10^4$, v 40 % primerov bo veljalo $n \leq 100$, v 20 % primerov pa celo $n \leq 15$.

Izhodni podatki: izpiši eno samo celo število, in sicer število knjig, po katerem sprašuje naloga.

Primer vhoda:

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

Pripadajoči izhod:

```
7
```

Komentar: v optimalni rešitvi ostanejo na polici od leve proti desni knjige višine 4, 8, 11, 7, 6, 5 in 2.

5. Posredne volitve

V neki državi imajo zelo nenavaden volilni sistem. Imajo n državljanov, ki so oštevilčeni z naravnimi števili od 1 do n. Vsak državljan, recimo u, si izbere nekega državljana, recimo $g(u)$, za svojega *zastopnika*. Lahko izbere tudi samega sebe, torej da je $g(u) = u$.

Na začetku dobi vsak državljan eno kroglico, nato pa si v več *fazah* podajajo kroglice med seboj po naslednjem preprostem pravilu: v posamezni fazi izroči vsak državljan vse kroglice, ki jih je imel ob začetku te faze, svojemu zastopniku.

Število kroglic, ki jih ima državljan u po k fazah, označimo s $f_k(u)$. Na začetku, pred prvo fazo, ima vsak eno kroglico, torej je $f_0(u) = 1$ za vsak u.

Pri opisanem prenašanju kroglic se lahko zgodi, da nek državljan ostane brez kroglic, torej da ima $f_k(u) = 0$. Ni se težko prepričati, da v tem primeru tudi v kasnejših fazah ne bo dobil nobene kroglice več. Zato bomo rekli, da je tak državljan *izpadel* iz volitev. Prej ali slej pa se nujno zgodi, da državljani nehajo izpadati. Naj bo K najmanjše tako število, za katero velja, da v fazah od vključno $K + 1$ naprej ne izpade noben državljan več. (Lahko se zgodi celo, da je $K = 0$.)

Volilna komisija namerava po K fazah prenašanja kroglic ta postopek končati; tiste državljane, ki bodo imeli takrat še kakšno kroglico, bodo razglasili za poslance v parlamentu, vpliv posameznega poslanca pri kasnejših glasovanjih v parlamentu pa bo sorazmeren s tem, koliko kroglic ima na koncu K-te faze. Pomagaj volilni komisiji in ji **napiši program**, ki določi K in izračuna število kroglic po koncu K-te faze, torej $f_K(u)$ za vse državljane u.

Vhodni podatki: v prvi vrstici je število državljanov n; to je celo število z območja $1 \leq n \leq 10^5$. Sledi n vrstic, pri čemer u-ta od njih vsebuje le $g(u)$; to je celo število z območja $1 \leq g(u) \leq n$.

V 40 % testnih primerov bo veljalo tudi $n \leq 1000$.

Izhodni podatki: v prvo vrstico izpiši vrednost K, po kateri sprašuje naloga, v drugo vrstico pa po vrsti izpiši vrednosti $f_K(1), f_K(2), \ldots, f_K(n)$, ločene s po enim presledkom.

Če je v tvoji rešitvi kakšna od vrednosti $f_K(u)$ napačna, vrednost K pa je pravilna, dobiš za tisti testni primer polovico vseh možnih točk.

Primer vhoda:

```
8
5
3
8
1
8
8
3
1
```

Pripadajoči izhod:

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

Komentar: v prvi fazi izpadejo državljani 2, 4, 6 in 7, v drugi fazi pa še državljan 3. Po tistem ne izpade nihče več, tako da je $K = 2$.

NALOGE ZA ŠOLSKO TEKMOVANJE

19. januarja 2018

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. Avtobus

Si na avtobusu in dobiš SMS „Kje si?“ od mame, ki čaka doma s kosilom. Želiš odgovoriti z imenom postaje, kjer se nahajaš, hkrati pa hočeš, da bo odgovor točen, torej, da ko SMS pošlješ, boš res na postaji, ki je napisana v SMSu. Dana imaš imena n postaj, $p_1, \ldots, p_n$, ki jih boš še obiskal do doma, in čase $d_1, \ldots, d_n$ v sekundah, ki jih bo avtobus potreboval za potovanje do naslednje postaje (d_i je čas vožnje od postaje p_{i-1} do p_i in je gotovo večji od 0). Da napišeš eno črko postaje, potrebuješ eno sekundo, prav tako pa tudi za to, da odpošlješ SMS (odpošlješ ga torej lahko eno sekundo po tistem, ko si napisal zadnjo črko sporočila). **Napiši program** ali del programa, ki izpiše, katero ime postaje boš sporočil. Če tako ime ne obstaja, naj izpiše „`doma`“. Če obstaja več primernih postaj, izpiši prvo med njimi (torej tisto z najnižjim indeksom i). Tvoj (pod)program lahko predpostavi, da so nizi p_i in števila d_i že podani v globalnih spremenljivkah (tabele ali seznami), lahko pa jih prebere s standardnega vhoda ali iz kakšne datoteke (karkoli ti je lažje).

2. Jedilnik

Srednje šole po Sloveniji si prizadevajo, da bi njihovi dijaki jedli čim bolj zdravo malico. Ravnatelj ene od srednjih šol bi rad, da mu **napišeš program**, s katerim bo lahko analiziral jedilnike, saj si želi, da bi bili njegovi dijaki zdravi in srečni.

```
primer_jedilnika = [
  "svinjski zrezek v omaki", "sirov kanelon", "ocvrt oslič",
  "svinjski zrezek v omaki", "ocvrt oslič", "sirov burek",
  "sirov kanelon", "ocvrt oslič", "sirov kanelon", "sirov kanelon"]
```

Jedilnik opišemo s tabelo oz. seznamom nizov, kot vidite na primeru. Vsak zaporedni element seznama ustreza enemu od zaporednih dni. Jedilnik se periodično ponavlja. Če je dolžina jedilnika n, bodo dijaki $(n+1)$-vi dan spet jedli tisto, kar je na prvem mestu na jedilniku.

Ravnatelj je sicer ugotovil, da se jedi ponavljajo. Če pa je od dneva, ko je bila neka jed nazadnje na jedilniku, minilo dovolj časa, ni s tem nič narobe. Napišite funkcijo `KdajSpet(L)`, ki dobi nek jedilnik (seznam oz. tabelo `L`) in vrne enako dolg seznam, kjer je za vsak dan ena številka, ki pove, koliko dni po tistem dnevu bo ista jed naslednjič spet na jedilniku. (Če ne znaš vrniti seznama iz funkcije, lahko rezultate namesto tega tvoja funkcija tudi preprosto izpiše.) Upoštevajte, da je jedilnik

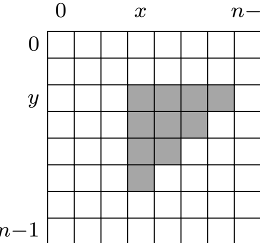
periodičen, torej se na primer jedilnik ["burek", "jogurt", "burek"] v naslednjih 10 dneh nadaljuje kot ["burek", "jogurt", "burek", "burek", "jogurt", "burek", "burek", "jogurt", "burek", "burek"].

Primer:

- KdajSpet(["burek", "jogurt", "burek"]) mora vrniti [2, 3, 1].
- KdajSpet(primer_jedilnika) (pri čemer je seznam primer_jedilnika tak, kot je definiran zgoraj v opisu naloge) mora vrniti [3, 5, 2, 7, 3, 10, 2, 5, 1, 2].

3. Trikotniki

Podana je tabela velikosti $n \times n$ (elementi tabele so cela števila), v kateri bomo zelo pogosto izvajali poizvedbe o največji vrednosti v trikotnem območju. Vsaka poizvedba bo opisana s koordinatama (x, y) levega zgornjega kota trikotnika ter velikostjo trikotnika v. Odgovor na poizvedbo je največja vrednost po vseh tistih celicah (x', y'), za katere velja $x \leq x'$, $y \leq y'$, $(y' - y) + (x' - x) < v$. Primer kaže naslednja slika:



Primer tabele za $n = 8$. Pri poizvedbi (x, y) in $v = 4$ nas zanima maksimum vrednosti po vseh tistih celicah tabele, ki so na sliki pobarvane sivo. Trikotnik, ki nas pri posamezni poizvedbi zanima, ima vedno takšno obliko in orientacijo, kot jo vidimo na tej sliki, spremeni se lahko le njegova velikost in položaj v tabeli.

Opiši postopek (in oceni njegovo časovno zahtevnost), ki bo čim hitreje odgovarjal na opisane poizvedbe. (Bolj učinkoviti postopki dobijo več točk kot manj učinkoviti.) Vrednosti v tabeli se ne bodo spreminjale, zato si smeš pomagati z vnaprejšnjo obdelavo vrednosti v tabeli (torej si še pred prvo poizvedbo izračunaš kakšne pomožne podatke, ki ti bodo pomagali, da boš kasneje hitreje odgovarjal na poizvedbe), vendar naj bo časovna zahtevnost tega dela $o(n^3)$ — tvoj postopek naj porabi strogo manj od reda n^3 operacij.

4. Točke in koši

Napiši podprogram (funkcijo) Kosi(m), ki izračuna, na koliko načinov lahko ekipa na košarkarski tekmi doseže m točk. Posamezni koš je lahko vreden 1, 2 ali 3 točke. Z drugimi besedami nas torej zanima, na koliko načinov se da izraziti m kot vsoto števil 1, 2 ali 3 (pri tem je pomemben tudi vrstni red seštevancev). Zaželeno je, da je tvoja rešitev čim bolj učinkovita.

Primer: $m = 3$ točke se da doseči na 4 načine: $1 + 1 + 1$, $1 + 2$, $2 + 1$ in 3.

5. Povprečni položaj znaka

V nekem besedilu želimo izračunati povprečni položaj določenega znaka v posamezni vrstici in skupno v celotnem besedilu. (Pri tem štejemo male in velike črke kot različne znake.)

Položaj znaka je številka mesta, kjer se znak nahaja, šteto od začetka vrstice. Prvi znak v vrstici se nahaja na mestu 1.

Izračunani povprečni položaj zaokrožimo navzdol na celo število. Če se znak v neki vrstici ali besedilu sploh ne pojavlja, bomo kot povprečni položaj vzeli vrednost 0.

Primer besedila:

```
V Notranjem stoji vas, Vrh po imenu. V tej vasici je živel
v starih časih Krpan, močan in silen človek. Bil je neki
tolik, da ga ni kmalu takega. Dela mu ni bilo mar; ampak nosil
je od morja na svoji kobilici angleško sol, kar je bilo
pa že tistikrat ostro prepovedano.
```

Če nas zanima znak „`k`“, je njegov povprečni položaj:

- v prvi vrstici: 0,0 $\Longrightarrow$ 0
- v drugi vrstici: 49,0 $\Longrightarrow$ 49
- v tretji vrstici: 25,75 $\Longrightarrow$ 25
- v četrti vrstici: 34,66... $\Longrightarrow$ 34
- v peti vrstici: 12,0 $\Longrightarrow$ 12
- v celotnem besedilu: 31,7 $\Longrightarrow$ 31.

Napiši podprogram (ali opiši postopek), ki bo za dani znak izračunal in izpisal

- število pojavitev tega znaka v vsaki vrstici in v celotnem besedilu,
- povprečni položaj tega znaka v vsaki vrstici in v celotnem besedilu.

Besedilo lahko tvoj podprogram prebere s standardnega vhoda ali pa iz kakšne datoteke (karkoli ti je lažje).

NEUPORABLJENE NALOGE IZ LETA 2016

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 11. tekmovanjem ACM v znanju računalništva (leta 2016), 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. 90–143) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Žaba

Žaba skače po ravnini. Na začetku in po vsakem skoku zabeležimo projekcijo točke (t.j. položaja žabe) na abscisno in ordinatno os, s čemer dobimo seznam parov koordinat (x_i, y_i).

Žaba po nekaj skokih skoči v pravokotno blatno mlako. **Napiši podprogram** vMlaki(tocke, mlaka), ki vrne (ali izpiše, če ti je lažje) prvi strnjeni podseznam skokov, ki v celoti potekajo po mlaki. Pri tem so skoki podani s seznamom vmesnih točk tocke, pravokotna mlaka pa s četverico mlaka, ki je oblike [x1, y1, x2, y2], kjer sta (x_1, y_1) in (x_2, y_2) koordinati nasprotnih oglišč. Če takega podseznama ni, naj funkcija vrne prazen seznam.

2. Tipkanje

Znašel si se v vlogi zapisnikarja, ki mora na računalniku vnesti seznam n besed. To počneš tako, da s pritiski na tipkovnico vnašaš črko po črko v vnosno polje, ki je na začetku prazno. Ko je v polju izpisana zahtevana beseda, zaključiš vnos s pritiskom na tipko Enter. Beseda pri tem ostane v vnosnem polju. Nato s pritiski na tipko Backspace pobrišeš nekaj (morda vse, ali pa nobene) zadnjih črk in nadaljuješ z vnosom naslednje besede.

Besede lahko vneseš v poljubnem vrstnem redu, ne nujno v takem, v kakršnem se pojavljajo v seznamu.

Opiši postopek, ki kot parameter dobi seznam besed in izračuna, najmanj koliko pritiskov tipk boš potreboval, da vneseš podane besede.

(To je težja različica naloge, ki smo jo na tekmovanju 2016 uporabili kot prvo v prvi skupini. Tam je bilo treba besede vnesti v takem vrstnem redu, v kakršnem so bile podane v vhodnem seznamu.)

3. Srečna števila

Mislimo si naslednji postopek, ki je podoben Eratostenovemu rešetu, le malo drugačen:

1. Začnimo z zaporedjem vseh lihih števil:

$$1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, \ldots$$

2. Naj bo $k = 2$.

3. Poglejmo, katero je k-to število v našem zaporedju. Recimo, da je to število x.
4. Pobrišimo vsako x-to število v našem zaporedju.
5. Povečajmo k za 1 in pojdimo nazaj na korak 3.

Številom, ki jih nikoli ne pobrišemo, rečemo, da so *srečna števila.* **Napiši podprogram**, ki za dano število n preveri, če je srečno.

Primer: na začetku imamo zaporedje

$$1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, \ldots$$

Zdaj je $k = 2$ in zato $x = 3$. Pobrišimo vsako tretje število:

$$1, 3, 7, 9, 13, 15, 19, 21, 25, \ldots$$

Zdaj je $k = 3$ in zato $x = 7$. Pobrišimo vsako sedmo število:

$$1, 3, 7, 9, 13, 15, 21, 25, \ldots$$

V naslednjem koraku bi pobrisali vsako deveto število ipd. Števila, ki jih nikoli ne pobrišemo in so torej srečna, so:

$$1, 3, 7, 9, 13, 15, 21, 25, 31, 33, 37, 43, 49, 51, 63, 67, 69, 73, 75, 79, 87, 93, 99, \ldots$$

4. Vsota zmnožkov

Dan je izraz oblike $a_1 \circ a_2 \circ \ldots \circ a_n$, pri čemer so $a_1, \ldots, a_n$ znana realna števila. **Opiši postopek**, ki za vsak $\circ$ odloči, ali naj bo $+$ ali $\cdot$, tako da bo na koncu vrednost celega izraza največja možna.

Težja različica naloge: kaj pa, če smemo v izraz dodajati tudi oklepaje?

5. Izštevanka

(To je malo težja različica naloge, ki smo jo na tekmovanju 2016 uporabili kot četrto v prvi skupini.) V krogu stoji n otrok, oštevilčenih od 1 do n. Na začetku igre ima vsak od njih z življenj. Začnemo pri otroku številka 1 in od njega naredimo k korakov po krogu (pri tem torej njega ne štejemo, ampak štejemo šele otroke od 2 naprej) — k je pozitivno celo število, lahko je tudi večje od n. Otrok na tako določenem mestu izgubi eno življenje, število k pa povečamo za 1. Zdaj od njega naprej naštejemo k korakov in otrok, pri katerem se zdaj ustavimo, tudi izgubi eno življenje. Nato spet povečamo k za 1 in nadaljujemo igro. Če nek otrok izgubi vsa življenja, izpade iz igre in krog postane malo manjši. Igra se nadaljuje tako dolgo, dokler ne ostane v igri en sam otrok. **Napiši podprogram**, ki za dane n, z in k simulira dogajanje in izračuna številko zadnjega otroka, ki ostane v igri, ko vsi ostali izpadejo.

6. Temperature

Dano je zaporedje meritev temperatur $t_1, t_2, \ldots, t_n$ za n zaporednih dni. Odgovarjati bomo morali na poizvedbe oblike „ali obstaja v danem zaporedju kakšna taka strnjena skupina vsaj k dni, pri katerih je bila v vsakem od teh dni temperatura vsaj m?“ Preden začnemo odgovarjati nanje, si smemo iz zaporedja meritev zgraditi morebitne pomožne podatkovne strukture, ki nam bodo kasneje pomagale čim hitreje odgovarjati na poizvedbe. **Opiši postopke**, s katerimi se tega lotil: kakšne podatkovne strukture bi uporabil, kako bi jih pripravil in kako bi potem odgovarjal na poizvedbe.

7. Zakon prve števke

Za nekatere množice številčnih podatkov iz realnega življenja velja zanimiva opažena lastnost (Benfordov zakon): če vsa števila zapišemo kot desetiške številke (privzemimo, da gre za pozitivna cela števila), se na prvem mestu števka „1“ pojavi pogosteje kot „2“, dvojka pogosteje kot trojka, ... in devetica na prvem mestu je najmanj pogosta. V idealnih primerih se „1“ pojavi v približno 30 % primerov, „2“ v 18 %, ... in „9“ v manj kot 5 % primerov.[2]

Kot zanimivost: tako lastnost imajo zlasti podatki, ki po vrednosti obsegajo več velikostnih razredov. Tako na primer: površine vsake reke na Zemlji, število prebivalcev vseh naselij in mest, molekularne mase, borzni in računovodski podatki. Če za neko množico podatkov ta lastnost bistveno odstopa od idealnih razmerij, lahko posumimo, da je morda šlo za ponarejanje ali prirejanje podatkov.

Napiši program, ki bo prebral veliko množico pozitivnih celih števil ter izračunal in izpisal za števke od 1 do 9, v koliko odstotkih podanih števil se ta števka pojavi na prvem mestu. Podatki *niso* na datoteki, pač pa je podana podprogramska funkcija `Preberi()`, ki ob vsakem klicu vrne naslednji podatek (pozitivno celo število). Če podatkov ni več, ta podprogram vrne vrednost -1.

8. Bankomat

Na vogalu za lokalno trgovino stoji bankomat. Ta bankomat vsebuje bankovce za 10, 20 in 50 evrov in lahko izplačuje neomejeno visoke vsote denarja, zaokrožene na 10 evrov. Ker štedi z bankovci, vsoto vedno izplača tako, da porabi minimalno bankovcev. Vsoto 50 evrov izplača kot bankovec za 50 evrov, vsoto 60 evrov pa kot bankovec za 50 evrov in bankovec za 10 evrov.

V denarnici pa seveda ne maramo prevelikih bankovcev, saj plačevanje z njimi pomeni veliko komplikacij z vračanjem denarja. Bolj nam ustreza, če imamo v denarnici 2 bankovca po 20 evrov in enega po 10 evrov, kot pa enega za 50 evrov. Pred dvigom denarja smo si zamislili, kakšne bankovce želimo, sedaj pa nas zanima, kolikokrat bomo morali na bankomatu dvigniti denar in kakšne vsote moramo dvigniti, da bomo dobili želene bankovce.

[2] Natančneje povedano, do tega pojava pride, če so logaritmi števil, ki jih opazujemo, porazdeljeni enakomerno na intervalu oblike $[a, b)$ za neka celoštevilska a in b. Za več o teh stvareh glej npr. Wikipedijo *s. v.* Benford's law.

Iz bankomata želimo dvigniti e bankovcev po 10 evrov, d bankovcev po 20 evrov in p bankovcev po 50 evrov. **Opiši postopek**, kako naj dvigamo vsote iz bankomata, da bomo želene bankovce dobili v minimalno dvigih. Bankomat lahko izplača poljubno visoke vsote, bankovcev mu ne zmanjka.

9. Kontrolna vsota

Za prebrano pozitivno celo število n želimo določiti njegovo „kontrolno vsoto“, ki je število med 0 in 9. Izračunamo jo tako, da seštejemo števke v desetiškem zapisu tega števila. Če je dobljena vsota večja od 9, postopek ponavljamo nad dobljeno vsoto, dokler ne dobimo števila pod 10.

Primer: $731102 \to 7+3+1+1+0+2 = 14 \to 1+4 = 5$.

Napiši program, ki bo prebral število n in izračunal ter izpisal njegovo „kontrolno vsoto“, kot jo določa opisani postopek.

10. Stave

Dandanes se dá staviti na vse mogoče. Janezek ima najrajši stave, pri katerih je treba napovedati skupno število golov, doseženih na rokometni tekmi, in razliko v golih po rednem delu tekme. Na listek si je zapisal n svojih stav. **Napiši program**, ki bo najprej prebral število stav in nato posamezne stave ter pri vsaki izpisal, kakšen mora biti rezultat, da bo Janezek stavo dobil. Če to ni možno, pa naj izpiše „`nemogoče`“.

Primer vhoda:

```
5
40 20
20 40
27 3
24 0
26 1
```

Pripadajoči izhod:

```
30 10
nemogoče
15 12
12 12
nemogoče
```

11. Sudoku

Cilj igre sudoku je zapolniti kvadratno mrežo velikosti 9×9 s števili od 1 do 9. Vsako število se lahko pojavi točno enkrat v vsakem stolpcu, vsaki vrstici in vsakem manjšem kvadratu velikosti 3×3 (kvadrat 9×9 razdelimo na 9 manjših kvadratov velikosti 3×3).

Dano je delno izpolnjeno kvadratno polje 9×9, predstavljeno s tabelo `T[9][9]`. V tabeli so številke (števke) od 1 do 9, prazno polje pa predstavlja število 0.

Naloga: na mesto (x, y) postavimo števko a (med 1 in 9). **Napiši podprogram** (oz. del programa), ki preveri, ali je postavitev a na to polje dopustna glede na dosedanje izpolnjene podatke, torej ali bi po tej postavitvi še vedno veljalo, da so v vsakem stolpcu, vsaki vrstici in vsakem manjšem kvadratu sama različna števila. Pri tem lahko predpostaviš, da so doslej ti pogoji veljali. Ni pa ti treba preveriti, ali postavitev a na polje (x, y) lahko pripelje k dokončni rešitvi (torej taki, pri kateri je pravilno izpolnjena cela mreža) ali ne.

12. Kakuro

Kakuro je neke vrste številska križanka, zelo podobna bolj znanim skandinavskim križankam, le da namesto črk v njej nastopajo števke. Dana je pravokotna mreža, v kateri so nekatera polja bela, nekatera pa črna. (V najbolj levem stolpcu in najbolj zgornji vrstici so vsa polja črna.) Strnjena skupina belih polj, ki ležijo vsa v isti vrstici oz. stolpcu, na obeh koncih pa jo zamejuje črno polje in/ali rob mreže, tvori „besedo“. V polja, ki tvorijo besedo, moramo vpisati števke (od 1 do 9, vse morajo biti različne), namesto opisa besede (kot pri običajni križanki) pa je tu predpisana vsota števk v besedi.

\	\	\	4\	10\	\	\	9\	7\
\	3\	6\7	1	6	11\	\4	3	1
\15	2	1	3	4	5	10\9	5	4
\4	1	3	23\	25\11	3	5	1	2
\	\16	2	6	3	1	4	15\	\
\	24\	23\23	8	9	2	1	3	15\
\29	7	8	9	5	16\	17\16	7	9
\17	8	9	\35	8	7	9	5	6
\15	9	6	\	\17	9	8	\	\

Napiši podprogram, ki kot parameter dobi opis popolnoma izpolnjene križanke (za vsako polje je znano, ali je belo ali črno; za črno polje je znana predpisana vsota za besedo pod njim in za besedo desno od njega; za belo polje je znano, katera števka je vpisana v njem) in preveri, če je izpolnjena pravilno (v skladu z vsemi omejitvami iz prejšnjega odstavka). Podatkovno strukturo za opis križanke si zamisli sam in jo tudi opiši.

13. Iskanje števila

Dana je tabela različnih celih števil $[a_0, a_1, \ldots, a_{n-1}]$; urejena so naraščajoče. Pri tem je n največ 1000. **Napiši podprogram** `Ujemanje(a)`, ki z največ 10 vpogledi v tabelo ugotovi, ali v tabeli obstaja tak i, da velja $a_i = i$.

Primer: v tabeli $[-4, 0, 1, 3, 7, 9]$ tak i obstaja (namreč $i = 3$), v tabeli $[7, 8, 9, 10]$ pa ne.

14. Torta

Dan je acikličen graf na n točkah, ki predstavlja zbirko kuharskih receptov. Vsaka točka predstavlja neko jed ali sestavino zanjo. Povezava $u \to v$ obstaja v tistih primerih, ko recept za jed v zahteva jed u kot sestavino. Na vsaki taki povezavi je tudi število d_{uv}, ki pove, da za pripravo ene enote jedi v potrebujemo d_{uv} enot jedi u. Mogoče je tudi, da v nekatere v ne kaže nobena povezava (to so sestavine, ki jih lahko kupimo v trgovini).

Opiši postopek, ki zna učinkovito odgovarjati na poizvedbe oblike „koliko enot sestavine s potrebujemo, če hočemo pripraviti eno enoto sestavine t?“. Pri tem moraš

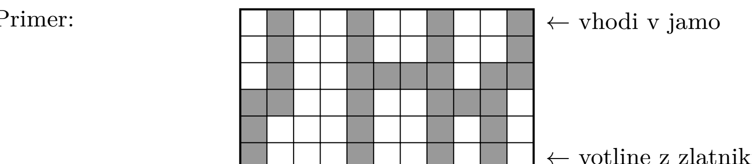
upoštevati tudi posredne odvisnosti (na primer: za pripravo t-ja potrebujemo nekaj jedi x, za pripravo le-te nekaj jedi y, za pripravo slednje pa nekaj jedi s).

15. Iskanje zlata

Podan je zemljevid jame zakladov kot karirasta mreža $w \times h$. V prvi vrstici se nahajajo luknje na površju, skozi katere lahko v jamo vstopimo, v h-ti vrstici pa se nahajajo votline z neko določeno količino zlatnikov (i-ta votlina naj vsebuje z_i zlatnikov). Po rovih v jami se lahko bodisi spuščamo po plezalni vrvi ali po njih potujemo v vodoravni smeri, nikakor pa ni mogoče plezati po nekem rovu navzgor (razen seveda, če smo skozenj prišli in je zato v njem že napeljana plezalna vrv).

Opiši postopek, ki za vsako vhodno luknjo izračuna število zlatnikov, ki jih lahko naberemo, če v njej pričnemo naš sprehod po jami.

Primer:



16. Davki

Davkarija poskuša z nagradno igro privabiti prebivalce k skeniranju QR-kod na računih. Z vidika davkarije tvorijo poskenirani računi seznam trojic $\langle$*ID* računa, izdajatelj, pošiljatelj$\rangle$ (pošiljatelj je tisti človek, ki je račun poskeniral in ga poslal davkariji). Seznam je urejen v takem vrstnem redu, v kakršnem je davkarija poskenirane račune dobila.

Pravila, po katerih bo davkarija izžrebala nagrajenca, so naslednja. Šteje le prva pojavitev vsakega računa; trojico torej ignorirajo, če se je pred njo v zaporedju že pojavila kakšna druga trojica z istim IDjem računa. Za vsakega pošiljatelja gredo nato po seznamu, dokler se jim ne naberejo računi 10 različnih izdajateljev; iz njih sestavijo *paket* (od vsakega izdajatelja vzamejo prvi račun) in uporabljene račune pobrišejo iz seznama. To ponavljajo, dokler je še mogoče. Ko tako obdelajo vse pošiljatelje, izmed vseh tako zbranih paketov naključno izberejo enega (pri tem ima vsak paket enako verjetnost, da bo izbran); nagrado dobi tisti pošiljatelj, ki je poslal račune v tem paketu.

(a) **Napiši program**, ki obdela podatke o prejetih računih in za vsakega pošiljatelja izračuna verjetnost, da bo izžreban prav on. Da bo lažje, lahko predpostaviš, da so tako računi kot izdajatelji in pošiljatelji predstavljeni s preprostimi številskimi identifikatorji tipa `int`.

(b) Recimo, da ima nek pošiljatelj zbrane račune od n različnih izdajateljev, in sicer a_i računov od izdajatelja i (za $i = 1, \ldots, n$). Recimo še, da davkarija iz njih tvori pakete velikosti p po zgoraj opisanih pravilih (tam je bilo $p = 10$). **Opiši postopek**, ki ugotovi, v kakšnem vrstnem redu mora ta pošiljatelj poslati svoje račune, da bo iz njih nastalo največje možno število paketov.

17. Tetris

Z liki iz igre tetris bi radi sestavili pravokotnik velikosti $w \times h$. Pri tem se liki ne smejo prekrivati ali štrleti ven iz pravokotnika, morajo pa ga popolnoma pokriti. Uporabimo lahko poljubno število izvodov posameznega lika. **Opiši postopek**, ki izračuna, na koliko načinov je mogoče sestaviti tak pravokotnik in kolikšno je skupno število pojavitev vsakega lika po vseh teh možnih razporedih.

18. RGB-šahovnica

Dana je karirasta mreža, ki ima obliko kvadrata velikosti $2^n \times 2^n$. Razdelimo jo lahko na 4 enake dele dimenzij $2^{n-1} \times 2^{n-1}$, vsakega od njih spet na štiri enake dele in tako naprej, dokler ne pridemo do posameznih polj mreže, torej enotskih kvadratkov $2^0 \times 2^0$. Delom velikosti $2^k \times 2^k$, ki jih dobimo pri takšnem deljenju, bomo rekli *kvadrati reda k*. Celotna mreža je torej kvadrat reda n, posamezna polja pa so kvadrati reda 0.

Mrežo bi radi pobarvali s tremi barvami (rdečo, zeleno in modro) tako, da bo vsako polje v celoti pobarvano z eno od teh barv, za večje kvadrate pa bodo veljale določene omejitve glede tega, katera barva v njih prevladuje. **Opiši postopek**, ki pobarva mrežo v skladu z omejitvami. V nadaljevanju sledi več različic naloge, ki se razlikujejo po tem, kakšne vrste omejitev dopuščajo:

(a) Omejitve so oblike „obstajati mora nek kvadrat reda k_i, v katerem je več kot polovica polj barve c_i“ (za $i = 1, \ldots, n$).

(b) Omejitve so oblike „v tistem kvadratu reda k_i, ki ima zgornji levi kot v (x_i, y_i), mora biti več kot polovica polj barve c_i“ (za $i = 1, \ldots, n$).

(c) Omejitve so oblike „v tistem kvadratu reda k_i, ki ima zgornji levi kot v (x_i, y_i), mora barva c_i pokrivati več polj kot vsaka od ostalih dveh barv (ne pa nujno več kot obe ostali barvi skupaj)“ (za $i = 1, \ldots, n$).

(d) Sestavi primer takega nabora omejitev, ki je v različici (c) rešljiv, v različici (b) pa ne.

19. Poskočno besedilo

Na listu papirja je napisano besedilo, kjer črke ne stojijo nujno v ravni črti, na primer takole: „$\LaTeX$“. Z optičnim čitalnikom smo prečitali list papirja, zdaj pa ga želimo s programom za prepoznavanje besedila pretvoriti v čisto besedilo. Žal pa črke, pomešane po vsem papirju, predstavljajo težavo za program, ki jih ne zna urediti v lep niz. Zato nam izpiše seznam vseh črk in koordinate zgornjega levega kota vsake črke, zdaj pa bi jih mi radi s programom uredili v smiselno besedilo.

Napiši program, ki na standardnem vhodu dobi število črk n in višino vrstice (na poskeniranem listu papirja) v, nato pa v n vrsticah (standardnega vhoda) črke in koordinate njihovega zgornjega levega kota. Črka je v k-ti vrstici (poskeniranega lista), če za njeno koordinato y velja $(k-1) \cdot v \leq y < k \cdot v$. Črke so podane v nekem neznanem vrstnem redu, ki nima nujno kakšne posebne zveze z njihovim položajem na listu papirja. Program naj izpiše lepo urejeno besedilo, vsako vrstico v svoji vrsti. Vse črke bodo male in velike črke angleške abecede.

20. Brisanje nezaklenjenih celic

Imamo tabelo oz. razpredelnico (*spreadsheet*) velikosti $w \times h$. V vseh celicah tabele so vpisani podatki. Nekatere celice so zaklenjene za spreminjanje. Vsaka od zaklenjenih celic je podana s koordinatama (x, y). Iz nezaklenjenih celic želimo izbrisati vse podatke. Podatke iz nezaklenjenih celic brišemo tako, da navedemo pravokotno območje celic, katerih vsebino želimo brisati. Pravokotno območje za brisanje, ki ga navedemo, ne sme vsebovati nobene zaklenjene celice, lahko pa vsebuje kakšno od celic, ki smo jo že pobrisali z enim od predhodnih brisanj.

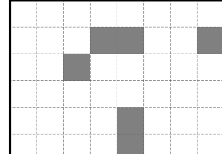
Nezaklenjene celice nameravamo pobrisati po naslednjem postopku:

1. Po tabeli se sprehajamo vodoravno od prve (najbolj leve) do zadnje (najbolj desne) celice prve (najvišje) vrstice, se nato premaknemo na začete naslednje spodnje vrstice in tako nadaljujemo do zadnje celice zadnje (najnižje) vrstice.

2. Ustavimo se na prvi nezaklenjeni in še nepobrisani celici.

3. Označimo vodoravno vse celice do prve zaklenjene celice ali konca vrstice, nato pa pravokotnik raztegujemo navzdol, dokler so na tem intervalu v spodnjih vrsticah same nezaklenjene celice. Ustavimo se pred vrstico, ki ima na tem intervalu vsaj eno zaklenjeno celico, oziroma na dnu stolpca. S tem smo pobrisali prvi pravokotnik nezaklenjenih celic.

4. Gremo na naslednjo nezaklenjeno celico (gledano od zadnje celice zgornje vrstice pravkar pobrisanega pravokotnika) in določimo naslednji pravokotnik (skočimo na korak 3). Postopek ponavljamo, dokler ne pobrišemo vseh nezaklenjenih celic.



Primer: leva slika kaže začetno stanje mreže (temno osenčena polja so zaklenjena), srednja kaže stanje po petih pobrisanih pravokotnikih (pri tem se peti pravokotnik malo prekriva s tretjim), desna pa po osmih pravokotnikih (takrat so pobrisane vse nezaklenjene celice; šesti pravokotnik se je delno prekrival s tretjim in petim).

Napiši podprogram, ki kot vhod dobi opis tabele (podatke o tem, katere celice so zaklenjene) in izpiše seznam pobrisanih pravokotnikov, kot jih dobimo po zgoraj opisanih pravilih.

Težja različica: **opiši postopek**, ki reši nalogo tudi za primer, ko je tabela zelo velika, zaklenjene celice pa so opisane tako, da je podanih n pravokotnikov, v katerih so vse celice zaklenjene, zunaj teh pravotkonikov pa so vse celice odklenjene. Časovna in prostorska zahtevnost tvoje rešitve naj ne bosta odvisni od w in h, pač pa le od n.

21. Evklidov algoritem

Dan je Evklidov algoritem za iskanje največjega skupnega delitelja dveh števil:

```
  vhod: naravni števili a, b;
1 while b > 0:
2    t := a mod b; a := b; b := t;
3 return a;
```

Opiši postopek, ki za dano zgornjo mejo M poišče nek tak par (a, b), da je $1 \leq b \leq a \leq M$ in da zgornji algoritem izvede čim več iteracij zanke. (Različica: opiši postopek, ki za dani k poišče naravni števili a in b, pri katerih je $1 \leq b \leq a$ in zgornji algoritem izvede natanko k iteracij zanke. Pri tem naj bo a najmanjši možni in če je pri tem a mogoče k iteracij doseči z več različnimi b-ji, naj bo tudi b najmanjši možni.)

22. Prepisovanje

V učilnici eden za drugim sedi n učencev in piše esej, ki si ga za potrebe te naloge predstavljajmo kot niz dolžine natanko d znakov. Vsak učenec i si zamisli svoj niz s_i, nato pa se za vsak indeks j (od 1 do d) posebej odloči, ali bi kot j-to črko svojega eseja uporabil j-to črko niza s_i ali pa bi prepisal j-to črko iz eseja predhodnega (torej $(i-1)$-vega) učenca. Nizu, ki na ta način nastane in ki ga i-ti učenec napiše, recimo t_i. Prvi učenec ($i = 1$) nima od koga prepisovati in pri njem je $t_i = s_i$. Pri kasnejših učencih pa je t_i lahko nekakšna mešanica istoležnih črk iz nizov $s_1, \ldots, s_i$, odvisno od tega, katere črke so se kateri učenci odločili prepisati. Dogajanje bi lahko opisali s takšno psevdokodo:

```
for i := 1 to n:
   for j := 1 to d:
      if i = 1 then t_i[j] := s_i[j]
      else t_i[j] := eden od t_{i−1}[j] in s_i[j];
```

Opiši postopek, ki kot vhodne podatke dobi nize $s_1, \ldots, s_n$ (vsi so dolgi po d znakov) in še nek niz p ter ugotovi, ali bi se lahko p pojavil kot (strnjen) podniz v nizu t_n.

23. Žaga

Imamo tanko pravokotno desko dimenzij $w \times h$ in računalniško vódeno žično žago, ki je na začetku postavljena v levi zgornji kot. Žaga dobiva ukaze za premik $\pm n$ cm vzdolž x- ali y-osi, npr. takole: $(x, +4), (y, +2), (x, -4)$. **Opiši postopek**, ki bo prejemal tovrstne ukaze in sporočil, ko bo deska razpadla na dva dela.

Z drugimi besedami: lahko si predstavljamo, da rišemo v pravokotniku neko lomljeno črto, sestavljeno iz vodoravnih in navpičnih daljic, zanima pa nas, kdaj začne nek del te črte tvoriti sklenjen lik, ki razdeli naš pravokotnik na dva ali več delov.

(a) *Lažja različica:* premiki so izmenično vodoravni in navpični.

(b) *Težja različica:* lahko je več zaporednih premikov vzdolž iste osi; to pomeni, da se žaga lahko vrne nazaj po zadnjih nekaj že prežaganih daljicah in nato od nekod nadaljuje z žaganjem v novo smer.

24. Sladkosnedi osel

Kmet bi rad zoral svojo kvadratno njivo dimenzij $n \times n$ s pomočjo svojega osla Tanga. Njivo orje postopoma tako, da v enem obhodu obdela najbolj zgornji, levi, desni ali spodnji pas v celoti (torej iz mreže $n \times n$ odstrani neko robno vrstico ali stolpec). Na nekaterih izmed polj njive pa se nahaja zavojček čokolade. Ker je Tango zares sladkosneda žival, opravilo pa zanj precej utrudljivo, se nikoli ne sme zgoditi, da bi zaporedno naredil k obhodov, ne da bi se vmes z zavojčkom posladkal. **Opiši postopek**, ki ugotovi, ali obstaja táko zaporedje obhodov, da bo na koncu preorana celotna njiva.

25. Wikipedija (I)

Dano je besedilo oblike:

```
main
Ni lepšega [[grm|grma]], kot je [[rožmarin]].
#####
drevo
Drevo je lahko podatkovna struktura ali pa
entiteta iz kraljestva rastlin.
#####
smreka
Smreka je [[drevo]].
#####
grm
Grm ni [[drevo]].
```

Besedilo predstavlja članke iz Wikipedije, ki so med seboj ločeni z vrsticami, ki vsebujejo le niz `#####`. (Zagotovljeno je, da taka vrstica vedno pomeni mejo med dvema zaporednima člankoma in se torej nikoli ne pojavlja kot del vsebine posameznega članka.) Vsak članek ima v prvi vrstici naslov, vse naslednje vrstice pa tvorijo besedilo članka. V članku se pojavljajo povezave v običajni wiki-sintaksi, torej je naslov članka, na katerega kaže povezava, zapisan med `[[` in `]]`. Zagotovljeno je, da se `[[` in `]]` ne uporabljajo v nobenem drugem kontekstu. Če je beseda sklanjana, se med oklepaji pojavi znak `|`. Pred `|` stoji naslov članka, na katerega kaže povezava, za znakom `|` pa je beseda, ki se dejansko prikaže v članku, kjer se ta povezava pojavlja.

Napiši program, ki prebere besedilo v zgoraj opisani obliki in vrne slovar, ki ima za ključe imena obstoječih člankov, za vrednosti pa sezname povezav, ki se pojavijo v besedilu članka. Iz zgornjega bi tako dobili:

```
{
    'main': ['grm', 'rožmarin'],
    'drevo': [],
    'smreka': ['drevo'],
    'grm': ['drevo']
}
```

26. Wikipedija (II)

Vhod pri tej nalogi je take oblike kot izhod pri prejšnji nalogi, torej slovar povezav v Wikipediji, v katerem ključi predstavljajo naslove člankov, pripadajoča vrednost pri

ključu x pa je seznam naslovov tistih člankov, na katere kažejo povezave v članku z naslovom x. Zagotovljeno je, da obstaja članek z imenom `main`, ki predstavlja glavno stran Wikipedije. **Opiši postopke**, ki analizirajo povezave med članki na naslednje načine:

(a) *Sirota* je stran, ki ni dosegljiva, če začnemo na članku `main` in sledimo povezavam. Poišči vse sirote.

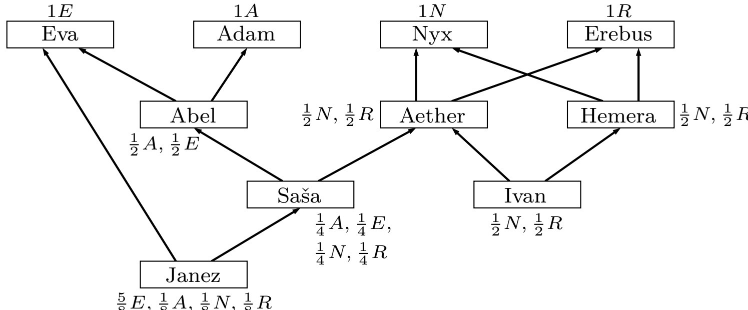
(b) *Rdeča povezava* je tista povezava, ki nima svojega članka. Poišči vse rdeče povezave.

(c) Poišči *Slakove komponente*, t. j. krepko povezane komponente, ki jih sestavljajo same *sirote*. (Krepko povezana komponenta je taka skupina strani, za katere velja, da je vsaka od njih dosegljiva po povezavah iz vsake druge v enem ali več korakih in da v skupino ne moremo dodati nobene dodatne strani, ne da bi ta pogoj prenehal veljati.)

(d) Poišči minimalno število povezav, ki jih je treba še vnesti v obstoječe članke, da se znebimo vseh sirot.

27. Praljudje

Praljudje so ljudje, ki nimajo staršev, saj jih je ustvaril vsemogočni Bog. Vsi nadaljnji ljudje imajo natanko dva starša. Zgled:



Vsi praljudje imajo popolnoma različen genski material. Vsak otrok od vsakega starša podeduje natanko polovico genetskega materiala. Tako nastanejo v kasnejših generacijah razne mešanice genetskega materiala praljudi (kot kažejo ulomki ob pravokotnikih z imeni na gornji sliki).

Pravimo, da je oseba A rodu B, če v njenem genetskem materialu prevladujejo geni pračloveka B. Sicer pravimo, da je mešane krvi. Tako je pri primeru z gornje slike Janez pripadnik Evinega rodu, Ivan pa je mešane krvi.

Opiši postopek, ki kot vhod dobi podatke o tem, kdo so praljudje in kdo sta starša vsakega nepračloveka, in ki za vsakega nepračloveka ugotovi, čigavega rodu je oz. če je mešane krvi.

28. Brisanje duplikatov

Mislimo si preprost skriptni jezik, podoben zbirnemu jeziku (assemblerju). Skripto sestavlja zaporedje ukazov; vsak mora biti v svoji vrstici. Skripta lahko uporablja

poljubno število spremenljivk, ki jih ni treba posebej deklarirati; vse so celoštevilske (tipa **int**). Poleg teh spremenljivk lahko skripta dela tudi s tabelo T, ki jo sestavlja n celic $T[0], T[1], \ldots, T[n-1]$. Vrednost n se na začetku izvajanja nahaja v spremenljivki n; vse ostale spremenljivke, ki jih program uporablja, imajo na začetku izvajanja vrednost 2016.

V vrstici je lahko tudi oznaka vrstice (labela), na katero se lahko sklicujemo pri skokih (več o tem glej spodaj). Za oznako vrstice mora biti dvopičje. Če je v isti vrstici tudi nek ukaz, mora ta priti šele za dvopičjem.

Imena spremenljivk in oznake vrstic lahko vsebujejo črke angleške abecede, števke in podčrtaje, pri čemer prvi znak ne sme biti števka. Skripta ne loči med velikimi in malimi črkami.

Dovoljeni so tudi komentarji (vse od znaka # do konca vrstice se šteje za komentar in se ignorira).

Možni ukazi so naslednji:

- ADD x, y — izračuna $x + y$ in rezultat shrani v x.
- SUB x, y — izračuna $x - y$ in rezultat shrani v x.
- SET x, y — priredi spremenljivki x vrednost y.

V gornjih ukazih mora biti x ime spremenljivke, y pa je lahko ime spremenljivke ali pa celoštevilska konstanta.

- EQ x, y — primerja vrednosti x in y in si zapomni, ali je bil izpolnjen pogoj $x = y$ izpolnjen ali ne.
- NE x, y — kot EQ, le za pogoj $x \neq y$.
- LT x, y — kot EQ, le za pogoj $x < y$.
- LE x, y — kot EQ, le za pogoj $x \leq y$.
- GT x, y — kot EQ, le za pogoj $x > y$.
- GE x, y — kot EQ, le za pogoj $x \geq y$.
- AEQ x, y — kot EQ, le za pogoj $T[x] = T[y]$.
- ANE x, y — kot EQ, le za pogoj $T[x] \neq T[y]$.
- COPY x, y — skopira vrednost $T[y]$ v celico $T[x]$.
- STOP x — konča izvajanje programa in kot rezultat vrne x.

V gornjih ukazih sta lahko tako x kot y imeni spremenljivk ali pa celoštevilski konstanti.

- JMP lab — skoči v vrstico z oznako lab.
- CJMP lab — če je bil pogoj v nazadnje izvedeni primerjavi (katerikoli od ukazov EQ, NE, LT, LE, GT, GE, AEQ, ANE) izpolnjen, skoči v vrstico z oznako lab (sicer pa nadaljuje izvajanje v naslednji vrstici).

Primer: naslednja skripta prešteje, koliko elementov tabele T je enakih elementu $T[0]$:

```
        SET i, 0        # i bo števec v zanki po elementih tabele.
        SET k, 0        # k bo štel, koliko elementov je enakih T[0].
zac:
        GE i, n         # Ali smo že na koncu tabele?
        CJMP kon
        ANE i, 0        # Ali je T[i] ≠ T[0]?
```

```
          CJMP preskok
          ADD k, 1         # Tu vemo, da je T[i] = T[0], zato števec k povečajmo.
preskok:
          ADD i, 1         # Povečajmo i za 1.
          JMP zac          # Nadaljujmo z naslednjo iteracijo zanke.
kon:
          STOP k           # Vrnimo število elementov, ki so enaki T[0].
```

Pri naši nalogi pa bomo reševali malo drugačen problem. Lahko se zgodi, da ima več zaporednih elementov tabele T enako vrednost; take duplikate bi radi pobrisali, tako da bi od vsake take skupine več zaporednih enakih elementov ostal en sam element. Iz tabele

b	b	c	a	a	a	b	b

mora na primer na koncu nastati

b	c	a	b	?	?	?	?

pri čemer nam je za elemente, označene z vprašajem, vseeno, kakšno vrednost imajo na koncu.

Napiši program, ki prebere n in izpiše skripto v zgoraj opisanem jeziku; ta skripta mora znati pravilno pobrisati duplikate iz poljubne tabele n elementov in vrniti (z ukazom STOP) število elementov, ki so ostali po brisanju (v zgornjem primeru bi na primer morala vrniti 4).

Veljalo bo $n \leq 100$. Skripta, ki jo izpiše tvoj program, sme biti dolga največ 100 000 vrstic.

Opozorimo na to, da ni nujno, da skripta, ki jo tvoj program izpiše pri nekem konkretnem n, deluje tudi za tabele drugih velikosti. Tvoj program sme torej izpisati skripto, ki je posebej prilagojena le enemu samemu konkretnemu n-ju.

Pri vsakem testnem primeru (torej pri vsakem n-ju) bomo skripto, ki jo bo izpisal tvoj program, preizkusili na več vhodnih tabelah z n elementi. Če bo skripta kdaj vrnila napačen rezultat, dostopala do neveljavnega indeksa, se izvajala več kot 100 000 korakov, itd., bo tvoja rešitev pri tem testnem primeru dobila 0 točk. Sicer pa je število točk odvisno od največjega števila korakov (recimo mu K), ki jih bo izvedla tvoja skripta pri kakšni od vhodnih tabel za ta n.

Če bo ...	... dobiš toliko točk
$K \leq 3n$	10
$3n < K \leq 4n$	9
$4n < K \leq 11n/2$	8
$11n/2 < K \leq 7n$	7
$7n < K$	5

29. Požrešni taksist

Dan je neusmerjen graf cestnega omrežja v nekem mestu. Cestno omrežje torej sestavljajo točke in povezave med njimi, za vsako povezavo pa je znana tudi dolžina. Po povezavi se je mogoče peljati v obe smeri. Taksist mora peljati potnika od točke s do točke t, pri tem pa, ker je pohlepen, bi rad izbral čim daljšo pot; ne sme pa

nobene točke obiskati več kot enkrat. **Opiši postopek**, ki iz teh podatkov poišče najdaljšo možno taksistovo pot.

30. Konjunkcije

Podan je seznam $n \leq 10^6$ naravnih števil $x_1, x_2, \ldots, x_n$. Radi bi izračunali bitno operacijo IN (AND) nad vsemi njegovimi podseznami; tako na primer pri podseznamu od i-tega do j-tega elementa dobimo $a_{ij} = x_i \,\&\, x_{i+1} \,\&\, \ldots \,\&\, x_j$ (za binarni IN smo uporabili simbol &, podobno kot v C-ju in sorodnih jezikih). Da je tvoj algoritem sposoben tega izračuna, dokaži tako, da izpišeš vsoto kvadratov vrednosti a_{ij} za vse pare števil i in j ($1 \leq i \leq j \leq n$).

Primer: če kot vhod dobimo seznam $\langle 3, 5, 2 \rangle$, je pravilni rezultat 11.

Lažja različica naloge: namesto vsote kvadratov vseh a_{ij} izpiši vsoto vrednosti a_{ij} samih.

31. Domine

Na voljo imamo n različnih besednih domin, iz katerih skušamo sestaviti kar najdaljšo kačo. Določi najdaljšo kačo, ki jo lahko sestaviš iz danih domin. V kači se sme posamezno domino uporabiti največ enkrat in dve zaporedni domini se morata prekrivati z vsaj eno črko (konec prejšnje domine se mora ujemati z začetkom naslednje).

Primer: če imamo domine `ENTOLOGIJA`, `NEVERJETNO`, `DOMINE`, `MINEŠTRA`, `ETNOLOGIJA`, `JABOLKO`, lahko sestavimo kačo dolžine 25:

```
DOMINE
   NEVERJETNO
        ETNOLOGIJA
              JABOLKO
```

32. Robotek

Mislimo si neskončno binarno drevo, v katerem ima vsako vozlišče natanko dva otroka, levega in desnega. Pot od korena do vozlišča u lahko opišemo z zaporedjem ničel in enic, pri čemer ničle pomenijo premik v levega otroka, enice pa v desnega. Če tako do vozlišča u vodi pot $u_1 u_2 \ldots u_n$ (pri čemer so $u_1, \ldots, u_n \in \{0, 1\}$), bomo rekli, da vozlišče u predstavlja število $f(u) := \sum_{k=1}^{n} u_k / 2^k$.

Po tem drevesu se sprehaja robot in išče poljubno tako vozlišče, čigar število leži na danem intervalu $[a, b)$. Števili a in b sta racionalni in zanju velja $0 \leq a < b \leq 1$.

(a) Robot išče tako, da začne v korenu in se nato v vsakem koraku premakne v enega od otrok po naslednjem pravilu: če v levem poddrevesu trenutnega vozlišča leži kakšno tako vozlišče, čigar število leži na $[a, b)$, se robot premakne iz trenutnega vozlišča v njegovega levega otroka, sicer pa v desnega. Ko pa doseže kakšno vozlišče, čigar število leži na $[a, b)$, se ustavi in konča. Ugotovi, ali ta postopek deluje oz. kdaj ne in kaj je narobe z njim.

(b) Robot začne v korenu in gre proti najbližjemu vozlišču, čigar število $f(u)$ leži na $[a, b)$. **Opiši postopek**, ki izračuna, kolikokrat se mora premakniti, da ga doseže.

REŠITVE NALOG ZA PRVO SKUPINO

1. Collatz

Pomagamo si lahko z dvema gnezdenima zankama; zunanja gre po vseh možnih vrednostih začetnega člena k od a do b, notranja pa pregleda celotno Collatzevo zaporedje z začetnim členom k. Pri tem računa nove člene po formuli iz besedila naloge, ustavi pa se, ko pride do 1. V spremenljivki kNaj si zapomnimo največji člen tega zaporedja. Ko pridemo do konca, ga primerjamo z največjo vrednostjo drugih doslej pregledanih zaporedij (z začetnimi členi od a do $k-1$), ki jo hranimo v spremenljivki naj; če je nova rešitev enako dobra kot najboljše doslej, dodamo trenutni k v seznam rezultatov (spremenljivka v). Če je nova rešitev celo boljša od najboljše doslej, pa dosedanje rezultate zavržemo (izpraznimo vektor v, v katerem smo jih hranili) in si novo rešitev zapomnimo v naj.

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

vector<int> PoisciVse(int a, int b)
{
   int naj = 0;      /* Največja doslej najdena vrednost. */
   vector<int> v;  /* Začetni členi, pri katerih je bila dosežena. */
   /* Preglejmo vse možne začetne člene od a do b. */
   for (int k = a; k <= b; k++)
   {
      int kNaj = k; /* Največja doslej najdena vrednost v k-jevem zaporedju. */
      /* Preglejmo Collatzevo zaporedje z začetkom pri k. */
      for (int n = k; n > 1; n = (n % 2) ? 3 * n + 1 : n / 2)
         if (n > kNaj) kNaj = n;  /* Če je člen n največji doslej, si ga zapomnimo. */
      /* Če je to boljša rešitev od dosedanjih, dosedanje pobrišimo. */
      if (kNaj > naj) { v.clear(); naj = kNaj; }
      /* Dodajmo k med rešitve, če je vsaj tako dober kot najboljše doslej. */
      if (kNaj == naj) v.push_back(k);
   }
   return v;
}
```

Zapišimo to rešitev še v pythonu:

```
def PoisciVse(a, b):
   naj = 0; v = []
   for k in range(a, b + 1):
      n = kNaj = k
      while n != 1:
         n = n // 2 if n % 2 == 0 else 3 * n + 1
         if n > kNaj: kNaj = n
      if kNaj > naj: v.clear(); naj = kNaj
      if kNaj == naj: v.append(k)
   return v
```

Morebitna drobna izboljšava je še, da to, ali je nova vrednost n največja doslej, preverjamo le po koraku $n \to 3n+1$, ne pa tudi po $n \to n/2$, saj se pri slednjem vrednost n-ja gotovo zmanjša, ne pa poveča.

Če nočemo hraniti seznama rešitev (kot je npr. vektor v v gornjem podprogramu), bi morali narediti s k-jem dva prehoda od a do b; v prvem prehodu bi določili naj, v drugem pa izpisovali tiste k-je, pri katerih je kNaj == naj.

2. Alfa bravo

Vhodno besedilo berimo po besedah; za vsako prebrano besedo pojdimo z zanko po vseh črkah abecede in primerjajmo našo besedo s kodo tiste črke. Če sta niza enako dolga in se razlikujeta v največ enem istoležnem znaku, je to prava koda in lahko izpišemo pripadajočo črko. Spodnja rešitev hrani kode v tabeli (globalna spremenljivka kode), za primerjanje dveh nizov pa ima podprogram SeUjema. Ta najprej preveri, če sta niza enako dolga, nato pa primerja istoležne znake in šteje neujemanja. Čim opazi drugo neujemanje, odneha in sporoči, da sta niza preveč različna; če pa pride do konca z največ enim neujemanjem, sporoči, da se ujemata.

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

const string kode[] = { "ALFA", "BRAVO", ..., "ZULU" };

bool SeUjema(const string &s, const string &t)
{
   /* Če sta niza različno dolga, se gotovo ne ujemata. */
   int n = s.length(); if (t.length() != n) return false;
   /* Sicer primerjajmo istoležne znake. */
   for (int i = 0, stNeujemanj = 0; i < n; i++)
      /* Če je to že drugo neujemanje, lahko obupamo. */
      if (s[i] != t[i]) if (++stNeujemanj > 1) return false;
   /* Če smo prišli do konca, se dovolj dobro ujemata. */
   return true;
}

int main()
{
   while (true)
   {
      /* Preberimo naslednjo besedo. */
      string s; cin >> s; if (! cin.good()) break;
      /* Primerjajmo jo z vsemi kodami in izpišimo ustrezni znak. */
      for (int c = 0; c < 26; c++)
         if (SeUjema(s, kode[c])) { cout.put('A' + c); break; }
   }
   return 0;
}
```

Oglejmo si to rešitev še v pythonu:

```
kode = ["ALFA", "BRAVO", ..., "ZULU"]

def SeUjema(s, t):
   n = len(s); stNeujemanj = 0
   if len(t) != n: return False
   for i in range(n):
```

```
    if s[i] == t[i]: continue;
    stNeujemanj += 1
    if stNeujemanj > 1: return False
  return True

import sys
for vrstica in sys.stdin:
  for beseda in vrstica.split():
    for c in range(len(kode)):
      if SeUjema(beseda, kode[c]):
        sys.stdout.write(chr(ord('A') + c))
        break
```

Če bi bilo kod veliko, bi jih bilo koristno namesto v tabeli hraniti v drevesu po črkah (*trie*), po katerem bi se potem spuščali glede na črke pravkar prebrane besede (niz s), pri čemer bi imeli v mislih tudi to, da sme enkrat med tem spuščanjem priti do neujemanja. Lahko bi imeli celo po eno tako drevo za vsako dolžino kode in potem gledali le v tisto drevo, v katerem so kode enako dolge kot niz s, saj naloga zagotavlja, da do napak pri dolžini kod ne prihaja.

Dosedanja rešitev se ni opirala na nobene posebne predpostavke o tem, kakšne so posamezne kodne besede. Naloga postane še lažja, če upoštevamo, da se pri fonetični abecedi, kot je podana v besedilu naloge, vsaka beseda začne ravno na tisto črko, katero predstavlja; na primer: `ALFA` se začne na `A` in tudi res predstavlja črko A. Naloga pravi, da se lahko posamezna beseda na vhodu razlikuje od prave kodne besede na največ enem mestu. Ločimo dve možnosti: ali do neujemanja pride pri prvi črki kodne besede ali pa nekje kasneje. Če pride do neujemanja pri prvi črki, mora biti preostanek vhodne besede popolnoma enak preostanku prave kodne besede; če pa pride do neujemanja nekje kasneje, je prva črka vhodne besede enaka kot pri kodni besedi in je torej to že tudi tista črka, ki jo prava kodna beseda predstavlja.

Ta razmislek nas pripelje do naslednje rešitve, ki s pomočjo slovarja preveri, če je preostanek vhodne besede (brez prve črke) enak preostanku (torej brez prve črke) kakšne od kodnih besed, in če ni, izpiše kar prvo črko vhodne besede, saj ve, da je to prava črka.

```
kode = ["ALFA", "BRAVO", ..., "ZULU"]
preostanki = {koda[1:]: koda[0] for koda in kode}

import sys
for vrstica in sys.stdin:
  for beseda in vrstica.split():
    sys.stdout.write(preostanki.get(beseda[1:], beseda[0]))
```

3. Sestavljanka

Označimo število kosov naše sestavljanke z n, želeno razmerje med višino in širino pa z x. Iščemo torej tako izražavo $n = w \cdot h$ (za naravni števili w in h), pri kateri bo h/w čim bližje x. Lahko gremo v zanki po h in pri vsakem preverimo, ali je res delitelj n-ja; če je, potem tudi izračunamo h/w, ga primerjamo z najboljšim doslej in si ga, če je boljši, zapomnimo:

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

```
#include <stdio.h>

void Sestavljanka(int n, double x)
{
  int hNaj; double dNaj;
  for (int h = 1; h <= n; h++)
  {
    /* Ali je h sploh delitelj n-ja? */
    if (n % h != 0) continue;

    /* Izračunajmo w in razliko |h/w − x|. */
    int w = n / h;
    double d = fabs(h / double(w) − x);

    /* Če je najboljša doslej, si jo zapomnimo. */
    if (h == 1 || d < dNaj) hNaj = h, dNaj = d;
  }
  printf("%d * %d\n", hNaj, n / hNaj);
}
```

Še rešitev v pythonu:

```
def Sestavljanka(n, x):
  for h in range(1, n + 1):
    if n % h != 0: continue
    w = n // h
    d = abs(h / w − x)
    if h == 1 or d < dNaj: hNaj = h; dNaj = d
  return (hNaj, n // hNaj)
```

Časovna zahtevnost te rešitve je $O(n)$, ker v zanki pregleda n različnih vrednosti h-ja in ima z vsako od njih $O(1)$ dela. To bi se dalo na razne načine še izboljšati.

Na primer, naša zanka gre po vseh h-jih in pri vsakem najprej preveri, ali je delitelj n-ja. Če h je delitelj n-ja, se dá n izraziti kot $n = h \cdot w$ za nek celoštevilski w. Iz te enakosti takoj sledi, da ne moreta biti h in w oba večja od $\sqrt{n}$ (oz. oba manjša od $\sqrt{n}$), ker bi bil potem njun produkt večji od n (oz. manjši od n), ne pa enak n. Delitelji n-ja torej vedno nastopajo v parih, pri čemer je eden $\leq \sqrt{n}$, drugi pa $\geq \sqrt{n}$. Torej bi bilo dovolj, če bi naša zanka po h šla le do $\sqrt{n}$, ne pa do n. Pri vsakem h-ju, za katerega bi opazili, da je delitelj n-ja, pa bi takrat preizkusili še n/h (ki je v tem primeru tudi delitelj n-ja). Tako še vedno pregledamo vse delitelje n-ja, časovna zahtevnost našega postopka pa se z $O(n)$ zmanjša na $O(\sqrt{n})$.

Še bolje bi bilo, če bi n najprej razbili na prafaktorje, potem pa ne bi bilo težko kar našteti vseh njegovih deliteljev: če je $n = \prod_i p_i^{r_i}$, so njegovi delitelji vsa števila oblike $\prod_i p_i^{s_i}$ za $0 \leq s_i \leq r_i$. Vsakega od tako dobljenih deliteljev bi vzeli za h in pogledali, pri katerem je razmerje h/w najbližje x.

Lahko pa se sistematičnega pregledovanja h-jev lotimo malo drugače. Ko je h delitelj n-ja in torej velja $n = h \cdot w$ (za nek celoštevilski w), lahko razmerje h/w zapišemo kot $h/(n/h) = h^2/n$. Pogoj, da mora biti h/w čim bližje x (to zahteva naloga), je torej enak pogoju, naj bo h^2/n čim bližje x, to pa je enakovredno pogoju, naj bo h^2 čim bližje $x \cdot n$. Namesto da pregledujemo h-je v naraščajočem vrstnem redu od 1 naprej, bi bilo torej bolje, če bi jih pregledovali naraščajoče po vrednosti $E(h) := |h^2 - x \cdot n|$. Tako bi se lahko ustavili že kar pri prvem takem h-ju, za katerega bi se izkazalo, da je delitelj n-ja — tisto je potem najboljša rešitev, ki jo

iščemo. Funkcija E seveda doseže svoj minimum, $E = 0$, pri $h_0 := \sqrt{x \cdot n}$, kar pa ni nujno celo število, zato bomo začeli s $h = \lfloor h_0 \rfloor$ in $h = \lceil h_0 \rceil$ in se od tam premikali malo navzdol in malo navzgor, kakor pač nanesejo vrednosti funkcije E.

```
void Sestavljanka2(int n, double x)
{
   /* Funkcija napake, ki jo minimiziramo. */
   auto E = [x, n] (const int h) { return fabs(h * h − x * n); };
   double h0 = sqrt(n * x);  /* Tu doseže E svoj minimum. */
   int h1 = int(floor(h0)); if (h1 < 1) h1 = 1;
   int h2 = h1 + 1; if (h2 > n) h2 = n;
   double e1 = E(h1), e2 = E(h2); int h;
   /* Pregledujmo h-je po naraščajoči vrednosti E(h),
      dokler ne najdemo kakšnega takega h, ki deli n. */
   while (true)
      /* Kateri izmed e1 = E(h1) in e2 = E(h2) je manjši? */
      if (e1 <= e2)
      {
         if (n % h1 == 0) { h = h1; break; }
         h1−−; e1 = E(h1);
      }
      else
      {
         if (n % h2 == 0) { h = h2; break; }
         h2++; e2 = E(h2);
      }
   printf("%d * %d\n", h, n / h);
}
```

Ali v pythonu:

```
import math

def Sestavljanka2(n, x):
   def E(h): return abs(h * h − x * n)
   h0 = math.sqrt(n * x)
   h1 = max(1, math.floor(h0))
   h2 = min(h1 + 1, n)
   while True:
      if E(h1) <= E(h2):
         if n % h1 == 0: return (h1, n // h1)
         h1 −= 1
      else:
         if n % h2 == 0: return (h2, n // h2)
         h2 += 1
```

Pokazati je mogoče, da ima ta rešitev pri $x \leq 1$ v najslabšem primeru časovno zahtevnost $O(\sqrt{n})$; če pa je $x > 1$, je časovna zahtevnost v najslabšem primeru lahko kar $O(n)$. Na misel nam lahko pride, da bi takrat namesto za x raje rešili problem za $1/x$ in nato zamenjali širino in višino tako dobljene rešitve, vendar se hitro vidi, da lahko to pripelje do napačnih rezultatov.[3] To, kar bi morali narediti,

[3] Na primer: recimo, da je $n = 11$ in $x = 2$. Možni razbitji na sta le dve, 1×11 (razmerje $1/11$, kar je $\approx 0{,}09$) in 11×1 (razmerje 11); iskanemu $x = 2$ je očitno bližja prva od teh dveh možnosti. Toda če rešimo problem za razmerje $1/x$ (torej za $1/2$), bomo tudi dobili rešitev 1×11, in če jo potem obrnemo, dobimo 11×1, kar pa *ni* pravilna rešitev prvotnega problema (za $x = 2$).

je, da bi v takem primeru (torej za $x > 1$) šli z zanko po w-jih namesto po h-jih, pregledovali pa bi jih spet po naraščajočem odstopanju razmerja h/w od iskane vrednosti x, torej $\hat{E}(w) = |h/w - x| = |(n/w)/w - x| = |n/w^2 - x|$. Minimum doseže ta funkcija pri $w_0 := \sqrt{n/x}$, mi pa bi se potem v zanki premikali po celoštevilskih w gor in dol od te začetne vrednosti.

4. Pisalni stroj

Poiščimo največji i, pri katerem je $a_i \geq 1$ — z drugimi besedami, to je najbolj desni stolpec, v katerem je treba sploh kaj natipkati. Vsaj enkrat se bo moral torej naš stroj premakniti vse do stolpca i, da bomo lahko tam kaj natipkali; nobenega razloga pa ni, da bi se premaknili kaj dlje v desno, ker od tam naprej ni treba natipkati ničesar več. Ko se premikamo proti levi od začetka vrstice do stolpca i, lahko spotoma natipkamo še po en znak v vseh ostalih stolpcih $j < i$ (če imajo $a_j \geq 1$; kjer pa je $a_j = 0$, se le premaknemo naprej s tipko za presledek).

Če so bili vsi $a_j \leq 1$, smo s tem že natipkali vse, kar potrebujemo, in lahko končamo. Sicer pa se moramo premakniti v levo, da bomo lahko v kakšnem stolpcu natipkali še kaj; in premakniti v levo se ne moremo drugače kot s skokom na začetek vrstice.

Zdaj lahko poiščemo največji i, pri katerem je $a_i \geq 2$ — to je zdaj zadnji tak stolpec, v katerem je treba še kaj natipkati (ker smo doslej v njem natipkali šele en znak in to očitno ni dovolj). Ko se premikamo od začetka vrstice do tega i, lahko natipkamo še po en znak tudi pri vseh drugih stolpcih $j < i$, ki imajo $a_j \geq 2$.

Če so bili vsi $a_j \geq 2$, smo končali, drugače pa poiščemo največji i, pri katerem je $a_i \geq 3$ in tako naprej.

Označimo z b_k najbolj desni stolpec, v katerem je treba natipkati vsaj k znakov (torej: $b_k := \max\{i : a_i \geq k\}$) in naj bo $m := \max_i a_i$ največje število znakov, ki jih je treba natipkati v kakšnem stolpcu. Dosedanji razmislek nam je povedal, da je skupno število pritiskov na tipke enako

$$s := b_1 + 1 + b_2 + 1 + \ldots + b_{m-1} + 1 + b_m.$$

Pri tem členi $+1$ v tej vsoti predstavljajo pritiske na tipko za vrnitev v začetek vrstice. Po zadnjem (m-tem) prehodu v desno nam je ni treba pritisniti, saj naloga nič ne pravi o tem, da bi morali končati ravno na začetku vrstice.

Vprašanje je zdaj, kako čim ceneje izračunati to vsoto, sploh ker naloga pravi, da so lahko n in števila a_i (zato pa tudi število m) velika. Neugodno bi bilo, če bi naša rešitev porabila $O(m)$ ali celo $O(n \cdot m)$ časa, npr. ker bi m-krat pregledovali celo tabelo, da določimo $b_1, \ldots, b_m$.

Bolje je, če pregledujemo tabelo od desne proti levi. Ko pri tem prvič naletimo na stolpec z $a_i \geq 1$, je številka tega stolpca (torej i) ravno naš b_1. Ko prvič naletimo na stolpec z $a_i \geq 2$, je njegov i ravno naš b_2 in tako naprej. Vsakič, ko na ta način za nek k prvič izvemo za vrednost b_k, bi jo bilo treba prišteti k neki spremenljivki, v kateri se bo tako sčasoma nabrala vsota s.

Da bo postopek učinkovit, moramo biti pozorni še na to, da se lahko več vrednosti b_k pojavi pri istem stolpcu; na primer, če ima najbolj desni stolpec vrednost $a_n = 4$, bomo imeli $b_1 = b_2 = b_3 = b_4 = n$. Namesto da bi vsakega od teh štirih b_k-jev

posebej prištevali k s, je bolje, če upoštevamo, da so pač štirje, in prištejemo k s-ju kar $4 \cdot n$. Tako dobimo naslednji postopek:

```
m := 0;    (* največja doslej videna vrednost a_i *)
s := 0;    (* potrebno število pritiskov na tipke *)
for i := n downto 1:
  if a_i > m:
    (* Zdaj smo izvedeli, da je b_{m+1} = b_{m+2} = ... = b_{a_i} = i.
       Prištejmo te člene k vsoti s. *)
    s := s + i · (a_i − m);
    m := a_i; (* To je zdaj novi največji doslej videni a_i. *)
if m > 1:
  s := s + m − 1; (* Pritiski na tipko za vrnitev na začetek vrstice. *)
return s;
```

Časovna zahtevnost te rešitve je le $O(n)$, česa boljšega od tega pa že ne moremo pričakovati, saj gre $O(n)$ časa že samo za branje vhodnih podatkov.

5. Brzinomer

Koristno je imeti globalno spremenljivko s trenutnim položajem kazalca (v spodnji rešitvi je to kazalec). Ko sistem pokliče našo funkcijo Premik in nam pove novo hitrost, jo primerjajmo s trenutnim položajem kazalca in na podlagi tega določimo premik (+1, −1 ali 0). Preden se vrnemo iz funkcije, še prištejmo zamik k našemu položaju kazalca, da bomo pripravljeni na naslednji klic.

Naloga pravi, da kazalec ne more iti čez 250 in da bo sistem nadaljnje premike navzgor v tem primeru ignoriral. Zato ni posebne koristi od tega, da v takem primeru sploh vrnemo premik +1. Temu se lahko izognemo na primer tako, da če ob klicu dobimo hitrost nad 250, jo pred nadaljnjo obdelavo postavimo na 250.

Paziti moramo še na to, da na začetku ne poznamo pravega položaja kazalca. Naloga zato priporoča, naj prvih 250 korakov zahtevamo premike navzdol, tako da bo števec zagotovo prišel na 0. Spodnja rešitev uporablja zato še eno globalno spremenljivko (zacetek), ki pove, koliko od teh začetnih 250 korakov nam je še ostalo. Vsakič jo zmanjšamo za 1, ko pa pade na 0, začnemo z običajnim delovanjem kazalca. (Šlo pa bi tudi brez te dodatne spremenljivke — lahko bi na primer spremenljivko kazalec inicializirali na −250 in bi jo funkcija Premik počasi povečevala za 1, vračala pa premike navzdol; na običajno delovanje pa bi preklopila, ko bi bil kazalec večji ali enak 0.)

```
enum { Max = 250 };    /* najvišji možni položaj kazalca */
int kazalec = 0;       /* trenutni položaj kazalca */
int zacetek = Max;     /* koliko je še ostalo od začetnega premikanja navzdol */

int Premik(int hitrost)
{
  /* Na začetku se premikamo navzdol, da bo kazalec zagotovo prišel na 0. */
  if (zacetek > 0) { zacetek--; return -1; }

  /* Prevelike hitrosti oklestimo na Max. */
  if (hitrost > Max) hitrost = Max;

  /* Določimo premik, s katerim se bo kazalec približal pravi hitrosti. */
```

```
    int premik = (hitrost > kazalec) ? 1 : (hitrost < kazalec) ? −1 : 0;
    kazalec += premik; /* Nova vrednost kazalca po tem premiku. */
    return premik;
}
```

Zapišimo to rešitev še v pythonu:

```
Max = 250
kazalec = 0
zacetek = Max

def Premik(hitrost):
    global kazalec, zacetek
    if zacetek > 0: zacetek −= 1; return −1
    if hitrost > Max: hitrost = Max
    premik = 1 if hitrost > kazalec else −1 if hitrost < kazalec else 0
    kazalec += premik
    return premik
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Križci in krožci

Preprosta rešitev je, da gremo z dvema gnezdenima zankama po vseh poljih mreže in za vsako polje preverimo, ali se tam začne zmagovalna peterica enakih znakov. Če torej trenutno polje ni prazno, ampak je na njem križec ali krožec, moramo preveriti, ali se v kakšni smeri naprej od njega pojavijo še štirje enaki znaki. Za to preverjanje torej uporabimo še dve vgnezdeni zanki: eno po smereh in eno, s katero se premikamo naprej v trenutni smeri. Pri tem premikanju se ustavimo, če pademo čez rob mreže, naletimo na napačen znak ali pa naberemo pet enakih znakov.

Smer premikanja lahko opišemo s parom števil (DX[i], DY[i]), ki povesta, kako se v tisti smeri spreminjata x- in y-koordinata. Potrebujemo štiri smeri: vodoravno, navpično in dve diagonalni.

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

char KdoJeZmagal(const vector<string>& a)
{
   int h = a.size(); if (h == 0) return ' ';
   int w = a[0].size();
   const int DX[ ] = { 1, 0, 1, 1 }, DY[ ] = { 0, 1, 1, −1 };
   /* Preverimo vsak možni začetni položaj v tabeli. */
   for (int x = 0; x < w; x++) for (int y = 0; y < h; y++)
   {
      int c = a[y][x]; if (c != 'x' && c != 'o') continue;
      /* Na tem mestu je znak c. Ali so v kakšni smeri še štirje taki znaki? */
      for (int smer = 0; smer < 4; smer++)
      {
         /* Ali je iz (x, y) v smeri „dir“ prostora še za štiri znake? */
         int xx = x + 4 * DX[dir], yy = y + 4 * DY[dir];
         if (xx < 0 || yy < 0 || xx >= w || yy >= h) continue;
         /* Z zanko gremo naprej v smeri „smer“, dokler še vidimo znake „c“.
            Števec „d“ pove, koliko smo jih že videli. */
         int d = 1; while (d < 5 && a[y + d * DY[dir]][x + d * DY[dir]] == c) d++;
         if (d == 5) return c; /* Če smo našli pet znakov „c“, je ta igralec zmagal. */
      }
   }
   return ' '; /* Če pridemo do sem, ni zmagal nihče. */
}
```

Morebitna drobna izboljšava bi bila, da bi, preden se začnemo iz nekega (x, y) premikati v smeri smer, preverili še, če ni slučajno na sosednjem polju v *nasprotni* smeri že tudi znak c — to bi pomenilo, da kolikor dolgo skupino c-jev bi že našli iz (x, y) v smeri smer, se bo dalo najti še daljšo skupino, ko bomo začeli iz tistega sosednjega polja (in tisto polje bo prej ali slej prišlo na vrsto, saj zunanji dve zanki po x in y sčasoma preizkusita vsa možna začetna polja).

2. Popravljanje testov

Potek dogodkov je najlažje rekonstruirati, če gremo po znakih vhodnega niza od

konca proti začetku. Recimo, da poznamo stanje kupa po i-tem vhodnem znaku. Če je i-ti vhodni znak vprašaj (?), to pomeni, da je nek učenec takrat oddal svoj test. Oddal ga je seveda na vrh kupa, torej je to ravno tisti učenec, ki je po tem znaku na vrhu kupa; in pred tem znakom je bil kup tak kot po njem, le brez tega zadnjega testa na vrhu.

Po drugi strani, če je i-ti vhodni znak kaj drugega kot vprašaj, to pomeni, da je profesor vzel ta test z vrha kupa. Pred tem korakom je bil torej kup tak kot po njem, le s tem dodatnim testom na vrhu.

Tako se lahko počasi premikamo od konca niza proti začetku, sproti popravljamo stanje kupa (naša spodnja rešitev ga hrani kar v nizu kup) in rekonstruiramo oddaje.

Na začetku tega postopka bi načeloma hoteli vedeti, kakšen je bil kup na koncu vseh korakov iz vhodnega niza. Tega v resnici ne vemo, vemo pa, da so bili takrat na njem le še taki testi, ki jih profesor ni popravil. Zanje nimamo upanja, da bi ugotovili, kateri učenci so jih oddajali. Zato lahko vzamemo, kot da je bil kup na koncu vhodnega niza prazen; nato pa, če bomo med premikanjem nazaj po nizu kdaj naleteli na oddajo (znak ?), spremenljivka kup pa bo kazala, da je bil kup po tej oddaji prazen, bomo vedeli, da gre tu za oddajo takega testa, ki ga profesor ni popravil, zato te oddaje ne moremo rekonstruirati in je vhodni niz neveljaven.

Na koncu našega postopka, torej ko smo prišli na začetek vhodnega niza, moramo preveriti še to, ali je kup takrat prazen. Če ni prazen, to pomeni, da je profesor pobiral s kupa nekakšne teste, ki jih ni nihče oddal, torej je vhodni niz neveljaven.

Spodnji podprogram vhodni niz s sprejme po referenci in ga spremeni v izhodnega, vrne pa logično vrednost, ki pove, ali je bil vhodni niz veljaven. (Če ni bil, ostane s po vrnitvi iz naše funkcije v nekem delno spremenjenem stanju.)

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

bool Testi(string &s)
{
   string kup;
   for (int i = s.length() − 1; i >= 0; i−−)
   {
      /* Na tem mestu velja: po tistem, ko se je zgodilo prvih i + 1 dogodkov (oddaj
         ali popravljanj), je bilo stanje kupa takšno, kot je trenutno v nizu „kup“. */
      if (s[i] == '?')
      {
         /* V tem dogodku je nek učenec oddal svoj test. To je lahko le tisti,
            čigar test je po tem dogodku na vrhu kupa. Če je bil kup po tem
            dogodku prazen, je to nemogoče, torej je vhodni niz neveljaven. */
         if (kup.empty()) return false;

         /* Znak ? v vhodnem nizu zamenjajmo z oznako tega učenca. */
         s[i] = kup.back();

         /* Pred to oddajo je bil kup tak kot po njej, le brez vrhnjega testa. */
         kup.pop_back();
      }
      else
      {
         /* V tem dogodku je profesor pobral test z vrha kupa. Pred njim
            je bil torej kup tak kot po njem, le s tem dodatnim testom na vrhu. */
         kup.push_back(s[i]);
```

```
      /* V izhodnem nizu hočemo imeti pobiranja predstavljena z znakom X. */
      s[i] = 'X';
    }
  }
  /* Če kup zdaj ni prazen, to pomeni, da je bil vhodni niz neveljaven
     (profesor je pobiral teste, ki jih ni nihče oddal). */
  return kup.empty();
}
```

3. Cevi

Recimo, da se postavimo v neko celico naše mreže in razmišljamo o tem, kako bi obrnili cev v njej. Če že poznamo stanje zgornje sosede naše celice, nam to predstavlja določeno omejitev za našo celico: naša celica na tisto sosedo meji s svojim zgornjim robom, tako da, če se v zgornji sosedi en konec cevi konča na tem robu (ki je njen spodnji rob), se mora tudi v naši trenutni celici en konec cevi končati na tem robu (ki je njen zgornji rob); in po drugi strani, če se v zgornji sosedi na tem robu ne konča noben konec cevi, se tudi v naši trenutni celici ne sme. Drugače namreč cevi ne bi bile pravilno sklenjene, pač pa bi nekje nek konec cevi obvisel v zraku. (Ta razmislek deluje tudi, če zgornje sosede sploh ni, ker naša trenutna celica leži v prvi vrstici; takrat pač dobimo omejitev, da se v naši trenutni celici cev ne sme končati na zgornjem robu).

Podoben razmislek lahko naredimo tudi z levo sosedo, ki nam postavi omejitev glede levega roba naše trenutne celice. Hitro lahko opazimo, da obe omejitvi skupaj že enolično določata, kako mora biti zasukana cev v trenutni celici:

Tip cevi v trenutni celici	Naj se cev navezuje na zgornji oz. levi rob? na nobenega	le zgornjega	le levega	na oba
prazna		×	×	×
tip I	×			×
tip L				

Pri tem križci × označujejo primere, ko je problem nerešljiv — kakorkoli zasukamo trenutno celico, stanje na zgornjem in levem robu ne bo táko, kot ga zahtevata zgornja in leva soseda.

Tako lahko torej tudi za trenutno celico enolično določimo, kako mora biti obrnjena (ali pa vidimo, da je problem nerešljiv). Potem tudi ni težko določiti, koliko korakov potrebujemo, da jo obrnemo v pravo smer; izvedemo 0 ali več vrtenj za 90° v levo, razen če vidimo, da bi to zahtevalo tri korake, tedaj pa raje izvedemo eno vrtenje za 90° v desno.

Ker se je dosedanji razmislek opiral na to, da za zgornjo in levo sosedo že vemo, kako sta obrnjeni, je koristno mrežo pregledovati sistematično po vrsticah od zgoraj navzdol, v vsaki vrstici pa gremo po celicah od leve proti desni. Tako bomo vedno, preden pridemo do neke celice, že imeli obdelani njeno zgornjo in levo sosedo.

Zapišimo dobljeni postopek še s psevdokodo:

```
n := 0;
for y := 1 to v:
  for x := 1 to s:
```

$Z := (y > 1)$ **and** *mreža*$[x, y-1] \in \{$║, ╔, ╗$\}$;
$L := (x > 1)$ **and** *mreža*$[x-1, y] \in \{$═, ╝, ╔$\}$;
(* *Z in L povesta, ali na zgornjem oz. levem robu trenutne celice cev mora biti ali ne sme biti.* *)
$C :=$ vrednost v zgoraj omenjeni tabeli glede na Z, L in
tip cevi v celici (x, y);
if $C = \times$:
izpiši, da je problem nerešljiv; **return**;
povečaj n za toliko, kolikor vrtenj je treba,
da trenutna celica pride v stanje C;
mreža$[x, y] := C$;
return n;

4. Palačinke

Stanje kupa palačink lahko predstavimo s tabelo celih števil, recimo kup, v kateri so palačinke navedene od najnižje (kup[0]) do najvišje (kup[n − 1]). Na začetku inicializiramo elemente tabele preprosto na števila od 1 do n. Ko uporabnik zahteva izpis kupa, moramo iti le z zanko po vrsti čez tabelo in izpisovati vrednosti v njej.

Pri obračanju zgornjih k palačink lahko novo stanje kupa izračunamo v pomožni novi tabeli (recimo nova). Zadnjih k elementov stare tabele skopiramo na začetek nove, vendar v obrnjenem vrstnem redu. Nato prvih $n - k$ elementov stare tabele skopiramo na konec nove (v nespremenjenem vrstnem redu). Zdaj lahko vsebino nove tabele vpišemo nazaj v staro ali pa staro zavržemo in odslej namesto nje uporabljamo novo. Oglejmo si primer implementacije te rešitve v C++:

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

int n;
vector<int> kup;

void Obrni(int k)
{
   vector<int> nova(n);
   /* Zgornjih k palačink gre v obrnjenem vrstnem redu na dno novega kupa. */
   for (int i = 0; i < k; i++) nova[i] = kup[n - 1 - i];
   /* Nad njih pride spodnjih n - k palačink prvotnega kupa. */
   for (int i = k; i < n; i++) nova[i] = kup[i - k];
   kup = move(nova);
}

void Izpisi()
{
   for (int i = 0; i < n; i++)
      printf("%d%c", kup[i], i == n - 1 ? '\n' : ' ');
}

int main()
{
```

```
    /* Inicializirajmo kup. */
    scanf("%d", &n); kup.resize(n);
    for (int i = 0; i < n; i++) kup[i] = i + 1;

    /* Odgovarjajmo na poizvedbe. */
    for (int k; scanf("%d", &k) == 1; )
        if (k == 0) Izpisi(); else Obrni(k);

    return 0;
}
```

Slabost te rešitve je, da za obračanje k palačink vedno porabimo $O(n)$ časa (in tudi $O(n)$ prostora za pomožno tabelo nova — temu se sicer z nekaj dodatnega truda lahko izognemo). To je potratno, saj naloga pravi, da je k v povprečju majhen v primerjavi z n.

Boljšo rešitev dobimo, če si tabelo kup predstavljamo kot krožno (*ring buffer*) — najnižja palačinka ni nujno kup[0], ampak kup[z] za nek začetni indeks z, od tam naprej pa si sledijo kup[z + 1], ..., kup[n − 1], kup[0], ..., kup[z − 1]. V splošnem bomo palačinko, ki bi bila po starem na indeksu i, po novem hranili na $(z+i) \bmod n$. (Na to moramo zdaj paziti pri izpisovanju tabele.)

Ta nova interpretacija tabele kup na primer pomeni, da če povečamo z za 1, ne da bi v tabeli kup karkoli spremenili, nam bo tabela po novem predstavljala takšen kup, kot če bi v prejšnjem kupu vzeli najnižjo palačinko in jo premaknili na vrh kupa.

Če torej povečamo z za $n - k$, bo učinek tak, kot da bi vzeli spodnjih $n - k$ palačink in jih premaknili na vrh kupa; preostalih k palačink pa (ki so bile prej na vrhu kupa) pride zdaj na dno. To je že skoraj tisto, kar potrebujemo za obračanje k palačink, kot ga zahteva naša naloga; vse, kar moramo še narediti, je, da obrnemo vrstni red teh k palačink. To nam bo vzelo le $O(k)$ časa, pa tudi pomožne tabele ne potrebujemo:

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

int n, z = 0;
vector<int> kup;

void Obrni(int k)
{
    /* Ciklično zamaknimo tabelo za n − k mest.
       S tem pride zgornjih k palačink na dno kupa. */
    z = (z + n − k) % n;

    /* Obrnimo vrstni red spodnjih k palačink. */
    for (int i = 0, j = k − 1; i < j; i++, j−−)
        swap(kup[(z + i) % n], kup[(z + j) % n]);
}

void Izpisi()
{
    /* Pri izpisu moramo zdaj upoštevati, da je na dnu kupa tista palačinka,
       ki je na indeksu z, od tam pa gremo ciklično po preostanku kupa. */
    for (int i = 0; i < n; i++)
```

```
    printf("%d%c", kup[(z + i) % n], i == n - 1 ? '\n' : ' ');
}
```

Funkcija main lahko ostane enaka kot pri prvotni rešitvi, zato je nismo pisali še enkrat.

Še ena rešitev, ki bi tudi porabila le $O(k)$ časa za obračanje palačink, je, da namesto tabele (ali vektorja) uporabimo seznam, predstavljen z dvojno verigo členov, povezanih s kazalci (*doubly linked list*). Tu se lahko v zanki sprehajamo od konca seznama nazaj, pobiramo palačinke z njega in jih odlagamo v nek pomožni seznam; ko jih tako naberemo n, pa novi seznam vrinemo na začetek glavnega seznama (kar ustreza temu, da tistih k palačink vrinemo na dno kupa); to vzame le $O(1)$ časa, ker je treba le popraviti nekaj kazalcev med elementi seznamov. Oglejmo si primer takšne rešitve v C++:

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

list<int> kup;

void Obrni(int k)
{
  list<int> novi;
  /* Vzemimo zgornjih k palačink s kupa in jih obrnimo. */
  for (int i = 0; i < k; i++) {
    novi.push_back(kup.back());
    kup.pop_back(); }
  /* Vrinimo jih na dno kupa (v O(1) časa). */
  kup.splice(kup.begin(), novi);
}

void Izpisi()
{
  for (auto i = kup.begin(); i != kup.end(); ++i)
    printf("%s%d", i == kup.begin() ? "" : " ", *i);
  printf("\n");
}

int main()
{
  /* Inicializirajmo kup. */
  int n; scanf("%d", &n);
  for (int i = 0; i < n; i++) kup.push_back(i + 1);
  /* Odgovarjajmo na poizvedbe. */
  for (int k; scanf("%d", &k) == 1; )
    if (k == 0) Izpisi(); else Obrni(k);
  return 0;
}
```

Slabost te rešitve v primerjavi s prejšnjo je, da porabi $O(n)$ dodatnega pomnilnika za hrambo kazalcev med elementi seznama.

Še ena možnost je neke vrste lena obdelava vhodnih podatkov (*lazy evaluation*). Namesto da bi po vsakem obračanju gornjih k palačink izračunali novo stanje kupa,

si za začetek k-je le shranjujmo v nek seznam in jih nato obdelajmo vse skupaj. Če imamo r obračanj, pri čemer i-to od njih obrne k_i palačink, in če je vsota $k_1 + \ldots + k_r \leq n$, lahko v $O(n)$ časa preprosto izračunamo stanje kupa po vseh teh obračanjih, ker se bloki palačink, ki jih prizadenejo posamezna obračanja, med seboj nič ne prekrivajo. V spodnji rešitvi hranimo odložena obračanja (torej taka, ki jih še nismo upoštevali v tabeli kup) v seznamu odlozena, vsoto njihovih k_i pa v skupajOdlozenih. Ko pride nov ukaz za obračanje, ga dodamo v odlozena, razen če bi s tem vsota skupajOdlozenih presegla n — tedaj pa najprej izvedemo dosedanja odložena obračanja in jih pobrišemo iz seznama odlozena. Slednje naredimo tudi pred vsakim izpisom.

```
#include <stdio.h>
#include <vector>
using namespace std;

int n, skupajOdlozenih = 0;
vector<int> kup, odlozena;

void ObrniOdlozena()
{
  vector<int> nova(n);
  int z = n, na = skupajOdlozenih;
  for (int k : odlozena) {
    z -= k; na -= k;
    /* Naslednje obračanje prestavi palačinke z indeksov z, ..., z + k − 1
       na indekse na, ..., na + k − 1 v obrnjenem vrstnem redu. */
    for (int i = 0; i < k; i++)
      nova[na + k - 1 - i] = kup[z + i]; }
  /* Spodnjih n − skupajOdlozenih palačink pa ni bilo predmet nobenega
     od teh odloženih obračanj in zdaj pridejo na vrh kupa. */
  for (int i = 0; i < z; i++)
    nova[skupajOdlozenih + i] = kup[i];
  kup = move(nova);
  odlozena.clear(); skupajOdlozenih = 0;
}

void Obrni(int k)
{
  if (k + skupajOdlozenih > n) ObrniOdlozena();
  odlozena.push_back(k); skupajOdlozenih += k;
}

void Izpisi()
{
  if (! odlozena.empty()) ObrniOdlozena();
  for (int i = 0; i < n; i++)
    printf("%d%c", kup[i], i == n - 1 ? '\n' : ' ');
}
```

Funkcija main je lahko enaka kot pri prvi rešitvi in je nismo pisali še enkrat. Kaj lahko povemo o časovni zahtevnosti te rešitve? Vsak klic funkcije ObrniOdlozena porabi $O(n)$ časa. (1) Če jo pokličemo iz Izpisi, se to nič ne pozna, saj bi ta že sama tako ali tako porabila $O(n)$ časa za izpis celotne tabele kup. (2) Če pa jo pokličemo iz Obrni, lahko razmišljamo takole: naj bo s dosedanja vsota skupajOdlozenih in naj

bo k število palačink pri novem obračanju. Ker smo poklicali ObrniOdlozena, to pomeni, da je $s + k > n$, torej je vsaj eden od s in k gotovo večji od $n/2$. Torej lahko ceno $O(n)$ časa za klic ObrniOdlozena bodisi razdelimo med doslej odložena obračanja, če se njihovi k-ji skupaj seštejejo v $s \geq n/2$, bodisi jo lahko pripišemo k pravkar prihajajočemu obračanju, če je njegov k sam večji od $n/2$. V obeh primerih pride do tega, da mora vsako obračanje dolžine k nositi le $O(k)$ časa za izvajanje funkcije ObrniOdlozena. Tako je torej amortizirana časovna zahtevnost posameznega obračanja $O(k)$, kar je ravno to, kar smo si želeli.

Naloga pravi, da bo k v povprečju majhen, zato so za potrebe našega tekmovanja zadnje tri opisane rešitve, torej take, ki porabijo za obračanje k palačink $O(k)$ časa, dovolj dobre. Vseeno pa je zanimivo razmisliti še o rešitvah, ki bi se dobro obnesle tudi pri večjih k.

Spomnimo se naše prve rešitve, ki je celoten kup predstavila s tabelo. Na začetku nastopajo palačinke v tabeli kar po vrsti od 1 do n. Tabelo si lahko predstavljamo kot blok:

$$\langle 1..n \rangle.$$

Po preložitvi k palačink nastaneta dva bloka:

$$\langle n..(n-k+1) \rangle \; \langle 1..(n-k) \rangle.$$

Po naslednji preložitvi, recimo za l, načeloma nastanejo trije bloki. Če je $l \leq n - k$, zadeva ta preložitev le drugega izmed gornjih dveh blokov in dobimo

$$\langle (n-k)..(n-k-l+1) \rangle \; \langle n..(n-k+1) \rangle \; \langle 1..(n-k-l) \rangle.$$

Če pa je $l > n - k$, prizadene ta preložitev oba gornja bloka in dobimo

$$\langle (n-k+1)..(n-k+c) \rangle \; \langle (n-k)..1 \rangle \; \langle n..(n-k+c+1) \rangle \qquad \text{za } c = l - (n-k).$$

Tako lahko nadaljujemo in po vsaki preložitvi se lahko število blokov poveča za 1. Po r preložitvah imamo $r + 1$ blokov in zato naslednja preložitev stane $O(r)$ časa. Ko pridemo do $r = \sqrt{n}$, smo vsega skupaj porabili za teh r preložitev že $O(n)$ časa. Zato si lahko zdaj privoščimo porabiti še $O(n)$ časa, da izračunamo novo stanje tabele in si ga zapišemo. Odtlej imamo spet le en blok (par števil pri vsakem bloku zdaj pomeni indeksa v nazadnje izračunano stanje tabele).

Tako nam torej vsaka skupina $\sqrt{n}$ preložitev vzame skupaj $O(n)$ časa, povprečno torej $O(\sqrt{n})$ za vsako preložitev. Ali je to bolje ali slabše od prvotne rešitve, ki je porabila $O(k)$ časa za preložitev dolžine k? To je seveda odvisno od tega, kako velike k smo tam dobivali.

Rešitev z bloki lahko poskusimo še izboljšati. Namesto da imamo bloke v seznamu, jih zložimo v drevo — predstavljajmo si ga kot B-drevo, da bo lažje razmišljati o uravnoteževanju.[4] Bloki se bodo sčasoma drobili, pa recimo, da jih kar mi že na začetku razdrobimo do dolžine 1. Vsak list tako vsebuje eno palačinko p. Vsako vozlišče (tako list kot notranje) vsebuje tudi dolžino d območja, ki ga predstavlja, in zastavico f, ki pove, ali je to območje obrnjeno.

Naj bo $B(v)$ zaporedje palačink, ki ga predstavlja vozlišče v. Definiramo ga takole:

[4] Razlika v primerjavi z B-drevesom je sicer ta, da imajo pri B-drevesu elementi nekakšne ključe, po katerih so urejeni. Pri nas takih ključev ne bo, vrstni red elementov pa bo pač tak, kakršen je vrstni red palačink na kupu.

- če je v list: $B(v) := [v.p]$;
- če je v notranje vozlišče z otroki $c_1, \ldots, c_t$:
  - če $v.f = \textbf{false}$: $B(v) := B(c_1) + \ldots + B(c_t)$;
  - če $v.f = \textbf{true}$: $B(v) := \textsf{reversed}(B(c_1) + \ldots + B(c_t))$.

Kot običajno pri B-drevesu si izberimo neko stopnjo t in vzdržujmo lastnost, da ima vsako notranje vozlišče od t do $2t - 1$ otrok (edina izjema je koren, ki jih sme imeti manj). Drevo ima torej globino $O(\log_t n)$.

Razmislimo zdaj o postopku za preložitev k palačink. Iz korena začasno pobrišimo desnih nekaj poddreves, kolikor je še mogoče, ne da bi skupno število palačink v njih preseglo k. Nato v naslednjem najbolj desnem poddrevesu začnemo brisati njegova poddrevesa, spet kolikor je mogoče, ne da bi skupno število palačink v pobrisanih poddrevesih (vključno s tistimi od prej) preseglo k. Tako nadaljujemo navzdol po drevesu, dokler skupno število palačink v pobrisanih poddrevesih ne doseže točno k. V korenih pobrisanih poddreves obrnimo f (iz **false** v **true** in obratno) in nato ta drevesa v obrnjenem vrstnem redu vrinimo na začetek drevesa.

Kakšna je cena te operacije? Ko smo iz nekega vozlišča pobrisali eno ali več poddreves, je to vozlišče zdaj mogoče podhranjeno in bomo mogoče morali premakniti vanj nekaj poddreves iz kakšnega brata ali pa ju celo združiti. To vzame $O(t)$ časa in ker se lahko zgodi po enkrat na vsakem nivoju, bo to skupaj $O(t \log_t n)$. Podobno je pri dodajanju; zaradi dodajanja novih otrok je lahko vozlišče prepolno, zato ga je treba razcepiti (ali pa preseliti nekaj poddreves v brata). Tudi to bo skupaj $O(t \log_t n)$. Ker je t konstanta, torej vidimo, da nam obračanje k palačink (za poljuben k) vzame le $O(\log n)$ časa. Izpis kupa pa gre še vedno v $O(n)$ časa, saj moramo le po vrsti obiskati vsa vozlišča drevesa in pri tem izpisovati elemente v listih.

5. Jabolka

Koristno je vzdrževati tabelo z ocenami jabolk na tekočem traku. Če imamo n jarkov (pri naši nalogi je $n = 5$), imejmo tabelo $n + 1$ elementov, tako da prvi element predstavlja jabolko pred kamero, ostali pa jabolka pred posameznimi jarki. Če na nekem mestu jabolka sploh ni, naj bo tisti element tabele enak 0.

Ko premaknemo tekoči trak za en korak naprej, moramo ustrezno premakniti za eno mesto naprej tudi elemente tabele. To je lažje početi od konca tabele proti začetku, tako da si ob premikanju elementov naprej ne povozimo tistih, ki jih še nismo premaknili. Ob vrnitvi iz funkcije PremakniTrak dobimo oceno jabolka, ki je zdaj na novo prišlo pred kamero; to vpišemo v prvo celico naše tabele.

Tako tabela spet vsebuje prave podatke o stanju vseh jabolk na traku; zdaj se lahko v zanki zapeljemo po vseh jarkih in gledamo, ali je pred jarkom ravno takšno jabolko, ki sodi vanj. Če je, premaknimo roko do tistega jarka in jo sprožimo.

```
int main()
{
  enum { n = 5 };    /* Število jarkov. */
  int jabolka[n + 1];  /* Ocene jabolk pred kamero in jarki. */
  for (int k = 0; k <= n; k++) jabolka[k] = 0; /* Na začetku je trak prazen. */
  while (true)
```

```
  {
    /* Premaknimo jabolka naprej po tabeli. */
    for (int k = n; k >= 1; k--) jabolka[k] = jabolka[k - 1];

    /* Premaknimo trak in vpišimo v tabelo tisto jabolko, ki je zdaj pred kamero. */
    jabolka[0] = PremakniTrak();

    /* Poglejmo, katera jabolka je treba pahniti v jarke. */
    for (int k = 1; k <= n; k++)
      if (jabolka[k] == k) /* Je pred pravim jarkom? */
      {
        PremakniRoko(k); SproziRoko();
        jabolka[k] = 0; /* To mesto na traku je zdaj prazno. */
      }
  }
  return 0;
}
```

Slabost te rešitve je, da pri n trakovih porabi po vsakem časovnem koraku $O(n)$ časa, da premakne podatke o jabolkih na traku (v tabeli jabolka) in pregleda, katera so zdaj pred pravim jarkom. Učinek je tak, kot če bi se z vsakim jabolkom ukvarjali $O(n)$-krat; lepše bi bilo, če bi se z njim ukvarjali le $O(1)$-krat — ko pride na trak in ko pride do pravega jarka.

Če pride ob času t pred kamero jabolko z oceno o, to pomeni, da ga bo treba ob času $t + o$ odriniti v jarek o. Dotlej se nam z njim načeloma ni treba ukvarjati. Pripravimo si torej za vsak časovni trenutek v prihodnosti seznam jabolk (z njihovimi ocenami), ki jih bo treba takrat odriniti v ustrezne jarke. Takšnih seznamov potrebujemo n, torej za n časovnih korakov v prihodnost (saj gredo ocene jabolk le do n), in vseh jabolk skupaj je na teh seznamih največ n (saj je na traku prostora le za toliko jabolk).

```
int main()
{
  enum { n = 5 };
  struct Jabolko { int ocena, nasl; };
  Jabolko jabolka[n];

  /* Na začetku obsega seznam prostih zapisov vse elemente tabele „jabolka“,
     vsi seznami pa so prazni. */
  int seznami[n], t = 0, prosto = 0;
  for (int i = 0; i < n; i++) jabolka[i].nasl = i + 1, seznami[i] = -1;
  jabolka[n - 1].nasl = -1;

  while (true)
  {
    /* Premaknimo se na naslednji časovni korak in preberimo novo jabolko,
       ki pride zdaj pred kamero. */
    int novo = PremakniTrak(); t = (t + 1) % n;

    /* seznami[t] kaže na seznam jabolk, ki so zdaj (ob novem času t) pri pravem jarku
       in jih je treba odriniti vanj. Elemente tega seznama prestavimo v seznam
       prostih zapisov. */
    for (int i = seznami[t]; i >= 0; )
    {
      auto &J = jabolka[i]; PremakniRoko(J.ocena); SproziRoko();
      int nasl = J.nasl; J.nasl = prosto; prosto = i; i = nasl;
    }
```

```
      seznami[t] = −1;
      /* Če je res prišlo novo jabolko, ga dodajmo v seznam jabolk, ki jih bo
         treba odriniti v jarek ob času „t + novo“. */
      if (novo > 0)
      {
         int i = prosto; auto &J = jabolka[i]; auto &S = seznami[(t + novo) % n];
         J.ocena = novo; prosto = J.nasl; J.nasl = S; S = i;
      }
   }
   return 0;
}
```

V gornji rešitvi so elementi vseh seznamov skupaj v tabeli jabolka, polje nasl pa pri vsakem jabolku pove, katero je naslednje v istem seznamu (in ga bo torej treba ob istem času odriniti v pravi jarek; če je jabolko zadnje v svojem seznamu, ima nasl $= -1$). Na začetke seznamov kažejo indeksi v tabeli seznami, pri čemer seznami[t % n] kaže na seznam tistih jabolk, ki jih bo treba odriniti v jarke ob času t. Morebitne proste (neuporabljene) zapise v tabeli jabolka imamo tudi povezane v seznam, na začetek tega seznama pa kaže spremenljivka prosto. Ko se premaknemo na nov časovni korak (s klicem PremakniTrak), gremo po jabolkih s seznama za trenutni čas t in jih odrinemo v jarke, njihove zapise pa preselimo v seznam prostih zapisov. Nato novo jabolko (če je v tem časovnem koraku prišlo pred kamero kakšno novo jabolko) dodamo v ustrezni seznam (v ta namen uporabimo prvi prosti zapis iz seznama prostih zapisov).

Zdaj imamo z vsakim jabolkom le $O(1)$ dela — ko pride na trak (in ga dodamo v enega od naših seznamov) in ko ga odrinemo v ustrezni jarek (in ga pobrišemo iz našega seznama). Ker pride v vsakem časovnem koraku na trak največ en jarek, imamo tako v povprečju pri vsakem koraku le $O(1)$ dela, ne več $O(n)$ kot pri prejšnji rešitvi.

REŠITVE NALOG ZA TRETJO SKUPINO

1. Buteljke

Ob branju vhodnih podatkov štejmo za vsakega človeka, od koliko ljudi prejme buteljko za svoj rojstni dan in koliko ljudem podari buteljko za njihove rojstne dneve. Če se na koncu pri kakšnem človeku število prejetih in podarjenih buteljk ne ujemata, lahko takoj zaključimo, da sistem podajanja buteljk ne bo deloval — če nekdo podari v enem letu drugim več buteljk, kot jih sam prejme na svoj rojstni dan, to pomeni, da jih bo moral vsako leto dokupovati; in podobno, če nekdo na svoj rojstni dan prejme več buteljk, kot jih sam v enem letu podari drugim, to pomeni, da se bo pri njem iz leta v leto nabiralo več buteljk (ki jih bo moral nekdo drug dokupovati).

Če pa vsakdo prejme enako število buteljk, kolikor jih tudi podari, lahko sistem podajanja buteljk deluje v nedogled: ko ima nekdo prvič po uvedbi sistema rojstni dan, dobi natanko toliko buteljk, kolikor jih bo moral podariti drugim v času med tem rojstnim dnevom in naslednjim. Enako se potem spet zgodi na naslednji rojstni dan in tako naprej. Vprašanje je torej le, koliko buteljk bo moral ta človek podariti drugim še pred svojim prvim rojstnim dnevom — te bo moral namreč kupiti, ker drugače pred svojim prvim rojstnim dnevom še nima nobene buteljke.

Če sistem uvedemo na začetku leta, lahko skupno število kupljenih buteljk določimo zelo preprosto: za vsak par (a, b), kjer oseba a obdaruje osebo b, moramo pogledati, če ima b rojstni dan prej kot a (torej če $r_b < r_a$); če da, potem vemo, da bo moral a to buteljko prvo leto kupiti.

Naloga pravi, da prijatelji z načrtom začnejo „v optimalnem trenutku“, torej moramo razmisliti še o možnosti, da sistema ne uvedemo ravno na začetku leta, ampak nekoč kasneje. Naj bo u tisti človek, čigar rojstni dan nastopi najbolj zgodaj v letu (torej tisti z najmanjšo vrednostjo r_u). Recimo zdaj, da našega sistema ne uvedemo v začetku leta, ampak šele takoj takoj za u-jevim rojstnim dnevom. Kaj se zaradi tega spremeni pri kupovanju buteljk?

- Človek u je prej (ko smo sistem uvedli na začetku leta) imel rojstni dan kot prvi po uvedbi sistema, torej je takrat dobil toliko buteljk, kot jih preostanek leta razdeli drugim, zato mu ni bilo treba kupiti nobene. Po novem pa, ker se sistem uvede tik za njegovim rojstnim dnevom, to pomeni, da bo prve buteljke od drugih dobil šele čisto zadnji, na koncu prvega leta po uvedbi sistema. Vse buteljke, ki jih mora sam podariti drugim, bo moral torej prvo leto kupovati. Število kupljenih buteljk se torej poveča za toliko, kolikor buteljk podari človek u.
- Ko smo sistem uvedli na začetku leta, so morali vsi, ki podarijo u-ju buteljko, le-to prvo leto kupiti, ker je imel on rojstni dan kot prvi po uvedbi sistema in zato do takrat še nihče drug ni prejel nobene buteljke. Po novem pa, ker ima u rojstni dan šele čisto na koncu prvega leta po uvedbi sistema, to pomeni, da imajo vsi ostali svoje rojstne dneve prej in takrat dobi vsak toliko buteljk, kolikor jih bo moral sam podariti drugim do svojega naslednjega rojstnega dneva. Zato po novem nihče več ne kupuje buteljke za u-ja, ampak mu podari eno od tistih, ki jih je pred tem sam dobil za rojstni dan. Število kupljenih buteljk se torej zmanjša za toliko, kolikor buteljk prejme človek u.

Če obe omenjeni spremembi seštejemo in upoštevamo, da u podari enako število buteljk, kolikor jih tudi sam dobi (saj drugače sistem sploh ne bi mogel delovati in bi nad njim že na začetku obupali), lahko zaključimo, da se skupno število kupljenih buteljk sploh nič ne spremeni. Če datum uvedbe sistema počasi pomikamo še naprej po letu, nam enak razmislek pove, da se število kupljenih buteljk tudi kasneje ne bo spreminjalo. V resnici se torej prijateljem ni treba truditi z iskanjem optimalnega trenutka za uvedbo sistema, saj bo število kupljenih buteljk enako ne glede na datum uvedbe sistema.

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

int main()
{
   int nPrimerov; scanf("%d", &nPrimerov);
   while (nPrimerov-- > 0)
   {
      int n, d, k; scanf("%d %d %d", &d, &n, &k);
      // Preberimo podatke o rojstnih dnevih.
      struct Oseba { int r, dobi = 0, da = 0; };
      vector<Oseba> osebe {n};
      for (int i = 0; i < n; i++) scanf("%d", &osebe[i].r);
      // Preberimo podatke o obdarovanjih in štejmo, koliko buteljk bo treba kupiti.
      int stKupljenih = 0;
      for (int i = 0; i < k; i++) {
         int a, b; scanf("%d %d", &a, &b);
         auto &A = osebe[a - 1], &B = osebe[b - 1]; A.da++; B.dobi++;
         if (B.r < A.r) stKupljenih++; }
      // Preverimo, če vsakdo prejme in podari enako število buteljk.
      for (const auto &O : osebe)
         if (O.da != O.dobi) { stKupljenih = -1; break; }
      // Izpišimo rezultat.
      if (stKupljenih < 0) printf("Pijanci so med nami!\n");
      else printf("%d\n", stKupljenih);
   }
   return 0;
}
```

2. Pravokotnik

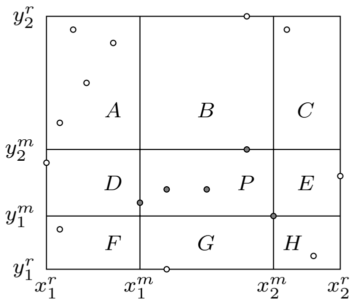
Za začetek poiščimo najmanjši tak pravokotnik, ki vsebuje vse modre točke (ko rečemo „vsebuje“, mislimo s tem, da ležijo na robu ali v notranjosti); moramo se le sprehoditi po njih in si zapomniti najmanjšo in največjo x-koordinato ter najmanjšo in največjo y-koordinato. Recimo temu pravokotniku $P = [x_1^m, x_2^m] \times [y_1^m, y_2^m]$.

Ker mora tudi tisti pravokotnik, po katerem sprašuje naša naloga, vsebovati vse modre točke, si ga lahko predstavljamo kot nekakšno razširitev pravokotnika P, ki jo dobimo tako, da P-jevo levo stranico premaknemo še malo bolj v levo, zgornjo stranico še malo bolj gor in tako naprej. Vprašanje je zdaj, katere stranice premakniti in kako daleč, da bomo dobili največji primerni pravokotnik.

Na primer, zgornjo stranico lahko dvigujemo le tako daleč, dokler se ne dotakne neke rdeče točke. Višino, na kateri se to zgodi, lahko dobimo tako, da med vsemi

rdečimi točkami (x, y), za katere je $y \geq y_2^m$ in $x_1^m < x < x_2^m$, poiščemo najnižjo (tisto z najmanjšim y). Podoben razmislek opravimo tudi pri ostalih treh stranicah in tako dobimo v vsaki smeri skrajni možni položaj stranice našega razširjenega pravokotnika. Recimo, da gre na levi do x_1^r, na desni do x_2^r, spodaj do y_1^r in zgoraj do y_2^r. Vse rdeče točke, ki ležijo zunaj območja $Q = [x_1^r, x_2^r] \times [y_1^r, y_2^r]$, lahko popolnoma ignoriramo, saj pravokotnika zagotovo ne bomo mogli razširiti tako daleč, da bi nas tiste točke kaj ovirale.

Območje Q lahko zdaj v mislih razrežemo pri $x = x_{1,2}^m$ in $y = y_{1,2}^m$; dobimo devet kosov, kot kaže spodnja slika (za primer iz besedila naloge):



Za B že vemo, da ne vsebuje nobene rdeče točke, razen prav na zunanjem (zgornjem) robu. Podoben razmislek velja tudi za D, E in G. Rdeče točke, ki bi nas utegnile pri raztegovanju pravokotnika P začeti ovirati že prej, preden dosežemo zunanji rob Q-ja, lahko torej ležijo le na vogalnih kosih A, C, F in H.

Spomnimo se, da iščemo največji pravokotnik, ki ne vsebuje nobene rdeče točke. To pa pomeni, da na vsaki od njegovih stranic leži vsaj ena rdeča točka; kajti če na neki stranici ne bi ležala nobena, bi lahko v tisto smer pravokotnik še malo razširili, ne da bi pri tem začel kršiti pogoje, ki jih postavlja naloga. Torej na primer za levo stranico našega iskanega pravokotnika pridejo v poštev le x-koordinate rdečih točk iz kosov A, D in F (tem bomo rekli *leve točke*), za desno stranico pa le x-koordinate rdečih točk iz kosov C, E in H (tem bomo rekli *desne točke*).

Vse možne položaje leve in desne stranice lahko torej preiščemo z dvema gnezdenima zankama, eno po x-koordinatah levih točk in eno po x-koordinatah desnih točk. Toda ko pravokotnik širimo (s premikanjem leve in desne stranice), pride vse več rdečih točk v tak položaj (nad ali pod našim pravokotnikom), da nas bodo omejevale pri premikanju zgornje in spodnje vrstice. Vprašanje je torej, kako pri izbranem položaju leve in desne stranice učinkovito izračunati, kako daleč smemo premakniti zgornjo in spodnjo stranico.

Recimo, da levo stranico fiksiramo in desno stranico počasi premikamo desno. Ko se desna stranica premakne desno mimo neke desne točke $T(x, y)$, to za zgornjo in spodnjo stranico pomeni naslednje: če je $y \geq y_2^m$ (torej če $T \in C$), v bodoče (ko bo desna stranica ležala desno od x) zgornja stranica ne bo smela biti nad y (ker bi sicer točka T prišla v notranjost pravokotnika); in podobno, če je $y \leq y_1^m$ (torej če $T \in H$), v bodoče spodnja stranica ne bo smela biti pod y. Če pa je $y_1^m < y < y_2^m$ (torej če $T \in E$), se desna stranica sploh ne sme premakniti še bolj v desno, saj bi pri tem točka T takoj prišla v notranjost pravokotnika.

S tem razmislekom lahko torej počasi premikamo desno stranico v desno in pri tem sproti spuščamo zgornjo in dvigujemo spodnjo stranico. Pred vsakim premikom moramo še izračunati ploščino pravokotnika in si izmed tako dobljenih ploščin zapomniti največjo.

Ko tako pregledamo vse možne položaje desne stranice pri trenutnem položaju leve stranice, lahko premaknemo levo stranico malo v levo (do naslednje primerne x-koordinate) in s podobnim razmislekom kot prej pri desni stranici tudi zdaj primerno popravimo višino zgornje in spodnje stranice; nato pa z notranjo zanko spet preizkusimo vse možne položaje desne stranice in tako naprej. Ker imamo dve gnezdeni zanki po rdečih točkah, je časovna zahtevnost naše rešitve $O(m + r^2)$.

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

typedef long long llint;
struct Tocka { llint x, y; };

int main()
{
  // Preberimo vhodne podatke.
  int m, r; scanf("%d %d", &m, &r);
  vector<Tocka> mt {m}, rt {r};
  for (int i = 0; i < m; i++) scanf("%lld %lld", &mt[i].x, &mt[i].y);
  for (int i = 0; i < r; i++) scanf("%lld %lld", &rt[i].x, &rt[i].y);

  // Poiščimo najmanjši pravokotnik P, ki pokrije vse modre točke.
  llint xm1 = mt[0].x, ym1 = mt[0].y, xm2 = xm1, ym2 = ym1;
  for (const auto &T : mt) {
    if (T.x < xm1) xm1 = T.x; else if (T.x > xm2) xm2 = T.x;
    if (T.y < ym1) ym1 = T.y; else if (T.y > ym2) ym2 = T.y; }

  // Poglejmo, kako daleč ga lahko raztegnemo gor in dol.
  llint yr1 = ym1 + 1, yr2 = ym2 - 1;
  for (const auto &T : rt)
    if (xm1 < T.x && T.x < xm2) {
      if (T.y <= ym1) if (yr1 > ym1 || T.y > yr1) yr1 = T.y;
      if (T.y >= ym2) if (yr2 < ym2 || T.y < yr2) yr2 = T.y; }

  // Pripravimo si urejen seznam rdečih točk levo in desno od P.
  sort(rt.begin(), rt.end(), [] (const auto &a, const auto &b) {
    return a.x < b.x || (a.x == b.x && a.y < b.y); });
  vector<Tocka> leve, desne;
  for (const auto &T : rt)
    if (T.x <= xm1) leve.push_back(T);
    else if (T.x >= xm2) desne.push_back(T);

  // Preizkusimo vse možne položaje leve stranice.
  llint maxPloscina = (xm2 - xm1) * (ym2 - ym1);
  llint yl1 = yr1, yl2 = yr2;
  for (int il = leve.size() - 1; il >= 0; il--)
  {
    const auto &L = leve[il];

    // Če postavimo levo stranico na L.x, nas leve točke po y-koordinati
    // omejijo na območje [yl1, yl2].
    llint y1 = yl1, y2 = yl2;
```

```
      // Preizkusimo vse možne položaje desne stranice.
      for (const auto &D : desne)
      {
         // Če premaknemo desno stranico na D.x, lahko po y-koordinati
         // pokrijemo območje [y1, y2].
         maxPloscina = max(maxPloscina, (D.x − L.x) * (y2 − y1));

         // Če premaknemo desno stranico desno od D.x, nas začne ovirati tudi točka D.
         if (D.y <= ym1) y1 = max(y1, D.y);
         else if (D.y >= ym2) y2 = min(y2, D.y);
         else break;
      }
      // Če premaknemo levo stranico levo od L.x, nas začne ovirati tudi točka L.
      if (L.y <= ym1) yl1 = max(yl1, L.y);
      else if (L.y >= ym2) yl2 = min(yl2, L.y);
      else break;
   }
   // Izpišimo rezultat.
   printf("%lld\n", maxPloscina); return 0;
}
```

3. Tekoči trak

Pot izdelka po trakovih lahko predstavimo kot lomljeno črto na ravnini. Če gre črta skozi točko (x, y) za $x \in \{1, \ldots, L\}$ in $y \in \{1, \ldots, n\}$, nam bo to pomenilo, da je izdelek prepotoval x-ti meter na y-tem tekočem traku. Ker se lahko premikamo le po vsakem prevoženem metru in še takrat le za en trak stran od trenutnega, to pomeni, da mora točki (x, y) slediti ena od točk $(x+1, y)$, $(x+1, y-1)$ in $(x+1, y+1)$. Naša lomljena črta je torej sestavljena iz odsekov, ki so lahko vodoravni ali pa gredo diagonalno gor ali diagonalno dol.

Ker mora izdelek svojo pot začeti in končati na prvem traku, se mora naša lomljena črta začeti v točki $(1, 1)$ in končati v $(L, 1)$. Ker se od $(1, 1)$ lahko dviguje največ z naklonom 1, se ne bo nikoli povzpela nad premico $y = x$. Podobno, ker se lahko proti $(L, 1)$ spušča največ z naklonom -1, se ne sme nikoli povzpeti nad premico $y = -x + L + 1$. In ker imamo le n trakov, se naša lomljena črta ne sme nikoli povzpeti nad premico $y = n$.

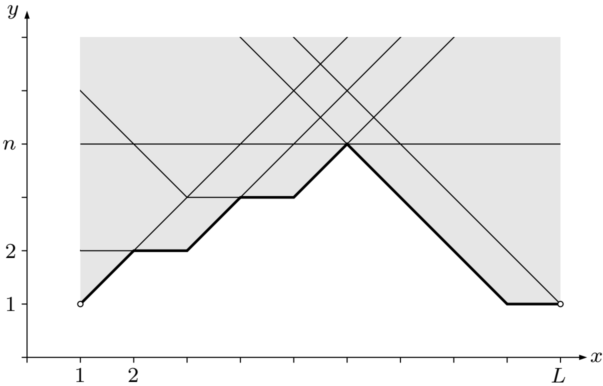
Kako pa na potek naše lomljene črte vplivajo omejitve hitrosti? Recimo, da imamo omejitev (s, e, v).

- Ta omejitev pomeni, da smemo po $(s+1)$-vem, $(s+2)$-gem, …, e-tem metru peljati s hitrostjo največ v; naša lomljena črta se torej na intervalu $[s+1, e]$ ne sme povzpeti nad premico $y = v$.
- Ista omejitev pomeni tudi, da smemo iti po s-tem metru s hitrostjo največ $v+1$, po $(s-1)$-vem metru s hitrostjo največ $v+2$ in tako nazaj (če bi namreč šli hitreje kot toliko, potem ne bi mogli pravočasno zmanjšati hitrosti, preden bi prišli do začetka omejitve). Naša lomljena črta torej se na intervalu $[1, s+1]$ ne sme povzpeti nad premico $y = -x + s + 1 + v$.
- Omenjena omejitev pomeni tudi, da gremo lahko po $(e+1)$-vem metru s hitrostjo največ $v+1$, po $(e+2)$-gem metru s hitrostjo največ $v+2$ in tako naprej (ker hitreje kot toliko ne moremo pospeševati po koncu naše omejitve).

Naša lomljena črta se torej na intervalu $[e, L]$ ne sme povzpeti nad premico $y = x + v - e$.

Tako se nam je nabral cel kup daljic (z naklonom -1, 0 ali $+1$), nad katere naša lomljena črta ne sme iti. Ker hočemo, da bi se izdelek premikal čim hitreje, pa se mora naša lomljena črta gibati čim višje; najbolje bo torej, če pri vsakem x teče prav po zgornjem robu dovoljenega območja.

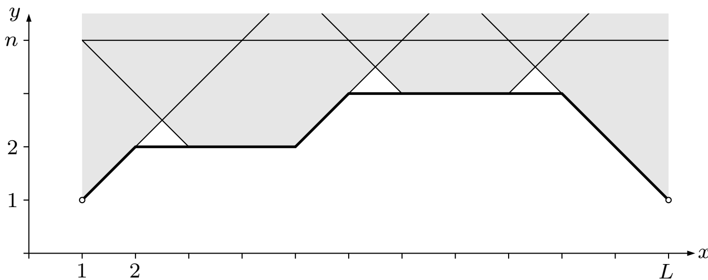
Naslednja slika kaže, kaj nastane po tem razmisleku za primer iz besedila naloge. Začetek in konec naše lomljene črte sta označena s krožcema, prepovedani deli ravnine (tisti, ki ležijo nad kakšno daljico) pa so osenčeni sivo. Če se ves čas gibljemo po zgornjem robu dovoljenega območja, nastane pot, ki je na sliki narisana z debelo črto.



Naloga zahteva, naj izpišemo tiste dele poti, kjer je izdelek potoval po enem traku vsaj dva metra skupaj. Na naši lomljeni črti se tak del poti pokaže kot vodoraven odsek. Tako na primer odsek od (x_1, y) do (x_2, y) pomeni, da je naš izdelek prepotoval na y-tem traku vse metre od vključno x_1-vega do vključno x_2-gega (skupaj je to $x_2 - x_1 + 1$ metrov, kar je ≥ 2, če je $x_2 > x_1$).

Poteku naše lomljene črte ni težko slediti. Začnemo v točki $(1, 1)$, nato pa na vsakem koraku pogledamo, katera izmed daljic, ki so prisotne na trenutnem x, ima pri tem x najnižjo y-koordinato (če je takih več, moramo izbrati tisto z najnižjim naklonom); po tisti se nato premaknemo naprej. Premaknemo se do prvega naslednjega presečišča te daljice s kakšno drugo, če pa takega presečišča ni, gremo pač vse do desnega krajišča daljice. Postopek se ustavi, ko dosežemo točko $(L, 1)$.

Pri računanju presečišča dveh daljic moramo paziti še na naslednjo podrobnost. Če imamo eno naraščajočo in eno padajočo daljico, recimo $y = x + b$ in $y = -x + b'$, je njuno presečišče pri $x = (b' - b)/2$, kar ni nujno celo število. Tudi y-koordinata pri tem x potem ne bi bila celo število. V takem primeru naša lomljena črta prav v to presečišče ne more priti; največ, kar lahko naredimo, je, da gremo po naraščajoči daljici do $\lfloor x \rfloor$, od tam naredimo korak vodoravno do $\lceil x \rceil$ in potem nadaljujemo po padajoči daljici. Naslednja slika kaže več takšnih primerov:



Ta slika ustreza vhodnemu primeru z $n = 4$ trakovi dolžine $L = 12$ in dvema omejitvama hitrosti: $(2, 5, 2)$ in $(6, 9, 3)$.

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

typedef long long llint;

struct Daljica // Daljica, ki jo določata pogoja y = ax + b in xOd ≤ x ≤ xDo.
{
   llint a, b, xOd, xDo;
   Daljica(llint x1, llint y1, llint x2, llint y2) // Inicializira daljico z danima dvema krajiščema.
   {
      if (x2 < x1) swap(x1, x2), swap(y1, y2);
      xOd = x1; xDo = x2;
      if (y1 == y2) a = 0, b = y1;
      else if (y2 − y1 == x2 − x1) a = 1, b = y1 − x1;
      else /* if (y2 − y1 == x1 − x2) */ a = −1, b = y1 + x1;
   }
};

struct Izhod // Pomožni razred za izpis poti izdelka.
{
   llint x = 1, y = 1;                    // Trenutni položaj.
   llint xv1 = 0, xv2 = 1, yv = 1;  // Zadnji vodoravni odsek (še ne izpisan).
   // Izpiše zadnji vodoravni odsek (če je dolg vsaj 2 metra).
   void Izpisi() { if (xv2 − xv1 > 1) printf("%lld %lld %lld\n", xv1, xv2, yv); }
   void Dodaj(llint xx, llint yy)       // Doda premik od (x, y) do (xx, yy).
   {
      if (x == xx) return;
      if (y != yy) { x = xx; y = yy; return; } // Poševen odsek.
      // Sicer imamo vodoravni odsek.
      if (x == xv2) { // Podaljšajmo trenutni vodoravni odsek.
         xv2 = xx; x = xx; return; }
      else {
         Izpisi(); // Izpišimo prejšnji vodoravni odsek, če je treba.
         // Začnimo nov vodoravni odsek.
         xv1 = x − 1; xv2 = xx; yv = y; x = xx; }
   }
};

int main()
```

```
{
  // Preberimo vhodne podatke.
  int n, L, m;
  scanf("%d %d %d", &n, &L, &m);
  vector<Daljica> ds; // Seznam daljic, ki predstavljajo omejitve.
  ds.emplace_back(1, 1, L, L); // ker moramo prvi meter prevoziti po prvem traku
  ds.emplace_back(L, 1, 1, L); // ker moramo zadnji meter prevoziti po prvem traku
  ds.emplace_back(1, n, L, n); // ker imamo le n trakov

  for (int i = 0; i < m; i++)
  {
    llint s, e, v; scanf("%lld %lld %lld", &s, &e, &v);
    // Za vsako omejitev hitrosti dodajmo ustrezne tri daljice.
    s++; if (s < e) ds.emplace_back(s, v, e, v);
    if (llint d = s − 1; d > 0) ds.emplace_back(s, v, s − d, v + d);
    if (llint d = L − e; d > 0) ds.emplace_back(e, v, e + d, v + d);
  }

  // Sledimo poti po spodnjem robu omejitev.
  Izhod izhod; llint &xOd = izhod.x, &yOd = izhod.y;
  while (xOd < L)
  {
    // Naš trenutni položaj je (xOd, xDo), pri čemer je yOd najnižji y, pri katerem
    // je za x = xOd prisotna kakšna od naših daljic. Izmed teh daljic bomo v
    // naslednjem koraku sledili tisti, ki ima najnižji naklon.
    int dNaj = −1; llint aNaj;
    for (int i = 0; i < ds.size(); i++)
    {
      const auto &D = ds[i];
      if (xOd < D.xOd || xOd >= D.xDo) continue; // Ne pokriva trenutnega xOd.
      if (D.a * xOd + D.b != yOd) continue;        // Ne gre skozi (xOd, xDo).
      if (dNaj < 0 || D.a < aNaj) dNaj = i, aNaj = D.a;
    }

    // Do kod se lahko gotovo premaknemo? Šli bomo do prvega presečišča s kakšno
    // drugo daljico, če pa takega ni, pa do konca trenutne daljice DD.
    const auto &DD = ds[dNaj];
    llint xDo2 = DD.xDo * 2;
    for (int i = 0; i < ds.size(); i++)
    {
      // Daljice, ki so vzporedne trenutni, preskočimo.
      const auto &D = ds[i];
      if (D.a == DD.a) continue;

      // Poglejmo, kje se D seka z našo DD. Namesto x bomo izračunali
      // njegov dvakratnik, ki je gotovo celo število.
      llint xPres2 = ((D.b − DD.b) * 2) / (DD.a − D.a);

      // To je pravzaprav presečišče njunih nosilk;
      // preverimo še, če res leži tudi na obeh daljicah.
      if (xPres2 > 2 * xOd && xPres2 > 2 * D.xOd && xPres2 <= 2 * D.xDo)
        xDo2 = min(xDo2, xPres2);
    }

    // Izračunajmo novi y in se premaknimo tja.
    llint xDo = xDo2 / 2, yDo = DD.a * xDo + DD.b;
    izhod.Dodaj(xDo, yDo);

    // Če je preseščišče na ne-celih koordinatah, ne moremo čisto do njega,
    // zato naredimo namesto tega vodoraven korak dolžine 1.
    if ((xDo2 % 2) == 1) izhod.Dodaj(++xDo, yDo);
```

```
  }
  // Izpišimo še zadnji vodoravni korak.
  izhod.Izpisi(); return 0;
}
```

Ko naša lomljena črta zapusti neko daljico, se nanjo ne more več vrniti. Daljic pa je $O(m)$, zato ima tudi naša lomljena črta le $O(m)$ odsekov. Toliko je zato tudi iteracij zunanje zanke, v vsaki iteraciji pa imamo $O(m)$ dela, da pogledamo, katere izmed ostalih daljic se sekajo s trenutno. Tako vidimo, da je časovna zahtevnost tega postopka $O(m^2)$, kar je za naše testne primere dovolj hitro.

Do še hitrejše rešitve pa pridemo, če za vsak možni naklon (-1, $+1$ in 0) vzdržujemo seznam trenutno aktivnih daljic s tem naklonom (torej takih, ki pokrivajo našo trenutno x-koordinato); v vsakem od teh seznamov naj bodo daljice urejene po konstantnem členu b. Potem je pri računanju presečišč dovolj, če gledamo le presečišča trenutne daljice z najnižjo daljico pri vsakem naklonu (torej tisto z najmanjšim b). Ko se z x premaknemo mimo začetnega krajišča kakšne daljice, jo moramo dodati na ustrezni seznam, ko se premaknemo mimo končnega krajišča, pa jo moramo s tistega seznama pobrisati. (V ta namen si je koristno na začetku pripraviti urejen seznam začetnih in končnih krajišč vseh daljic; to nam vzame $O(m \log m)$ časa.) Da bomo lahko učinkovito dodajali daljice v sezname in jih brisali, je bolje vsakega od teh seznamov v resnici predstaviti s primerno uravnoteženim binarnim drevesom (npr. rdeče-črnim), tako da bo dodajanje, brisanje in iskanje najnižje daljice vzelo le po $O(\log m)$ časa. Časovna zahtevnost celotnega postopka je tako le $O(m \log m)$ namesto $O(m^2)$.

Za konec si oglejmo še veliko krajšo in preprostejšo rešitev, ki sledi poti izdelka meter za metrom in na vsakem koraku preveri, kateri je najhitrejši trak, po katerem lahko nadaljuje svojo pot. Če smo trenutno na traku y, lahko poskusimo iti na $y+1$, nato pa to število po potrebi zmanjšamo, če tako zahtevajo omejitve: upoštevati moramo, da je trakov le n; če na naslednjem metru velja kakšna omejitev hitrosti, je ne smemo prekoračiti; pred omejitvami moramo pravočasno začeti zavirati, enako pa tudi proti koncu linije, da bomo zadnji meter prevozili na traku 1. Da vse to upoštevamo, moramo iti v gnezdeni zanki po vseh omejitvah, zato je časovna zahtevnost te rešitve $O(L \cdot m)$. V splošnem je to prepočasi, deluje pa dovolj hitro pri majhnih L. Besedilo naloge pravi, da pri 60 % testnih primerov velja $L \leq 100$, torej bi ta rešitev dobila 60 % točk.

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

int main()
{
  // Preberimo vhodne podatke.
  int n, L, m; scanf("%d %d %d", &n, &L, &m);
  struct Omejitev { int s, e, v; };
  vector<Omejitev> om {m};
  for (auto &O : om) scanf("%d %d %d", &O.s, &O.e, &O.v);

  // Odsimulirajmo pot izdelka.
```

```
  for (int xOd = 0, x = 1, y = 1; x <= L; x++)
  {
    // Trenutno smo na traku y, po katerem smo prevozili interval od
    // xOd do x (kjer smo zdaj). Po katerem traku naj nadaljujemo?
    int yy = y + 1;          // Lahko bi poskusili pospešiti.
    yy = min(yy, n);         // Toda ne smemo prekoračiti števila trakov n.
    yy = min(yy, L − x);     // Imeti moramo dovolj prostora za zaviranje na koncu.
    // Preglejmo še omejitve hitrosti.
    for (const auto &O : om)
      if (x < O.s)                        // Imeti moramo dovolj prostora za zaviranje,
        yy = min(yy, O.s − x + O.v);      // preden dosežemo omejitev.
      else if (x < O.e)
        yy = min(yy, O.v);                // Smo na območju omejitve.
    // Ali se bomo premaknili na nov trak?
    if (yy != y || x == L) {
      // Prej še izpišimo, kaj smo prevozili na starem traku.
      if (x − xOd > 1) printf("%d %d %d\n", xOd, x, y);
      y = yy; xOd = x; }
  }
  return 0;
}
```

4. Knjige

Označimo naše vhodno zaporedje velikosti knjig z $a_1, a_2, \ldots, a_n$. Naloga sprašuje po najdaljšem takem podzaporedju, ki najprej narašča in nato pada. Razmislimo najprej o malo lažjem problemu: recimo, da nas zanimajo podzaporedja, ki ves čas le naraščajo.

Izberimo si nek konkretni člen našega zaporedja, recimo a_i, in se vprašajmo, kako dolgo je najdaljše naraščajoče podzaporedje, ki se konča s tem členom. Tej dolžini recimo d_i. Ena možnost za d_i je vedno $d_i = 1$, saj je člen a_i sam zase tudi naraščajoče podzaporedje (dolžine 1). Glede daljših podzaporedij pa lahko razmišljamo takole: pred a_i mora biti v takem podzaporedju nek zgodnejši člen a_j, za katerega mora veljati $a_j < a_i$ (da bo podzaporedje res naraščajoče) in seveda $j < i$ (da bo to res podzaporedje prvotnega zaporedja, saj naloga pravi, da vrstnega reda knjig ne smemo spreminjati). Najdaljše naraščajoče podzaporedje s koncem pri a_i torej dobimo tako, da vzamemo najdaljše naraščajoče podzaporedje s koncem pri a_j (to pa je dolgo d_j) in mu dodamo člen a_i; dolžina tako podaljšanega podzaporedja bo torej $d_i = d_j + 1$. Da bomo dobili največji možni d_i, moramo med vsemi možnimi j-ji (tistimi, ki ustrezajo pogojema $j < i$ in $a_j < a_i$) izbrati tistega z največjim d_j. Tako si lahko predstavljamo naslednji postopek:

for $i := 1$ **to** n:
  $d_i := 1$;
  for $j := 1$ **to** $i - 1$:
    if $a_j < a_i$ **then** $d_i := \max\{d_i, d_j + 1\}$;

Slabost te rešitve je, da ima časovno zahtevnost $O(n^2)$, kar je za največje testne primere pri naši nalogi že prepočasi.

Do boljše rešitve nas pripelje naslednji razmislek. Ko dobimo nov člen a_i, se lahko vprašamo: ali je mogoče s tem novim členom podaljšati kakšno naraščajoče

podzaporedje dolžine k? (Če je to mogoče, potem vemo, da obstaja naraščajoče podzaporedje dolžine $k+1$ s koncem pri členu a_i.) To bomo najlažje naredili, če izmed vseh naraščajočih podzaporedij dolžine k vzamemo tisto, pri katerem je zadnji člen najmanjši. Če je celo pri njem zadnji člen $\geq a_i$, to pomeni, da niti tega podzaporedja niti nobenega drugega naraščajočega podzaporedja dolžine k ne bomo mogli podaljšati s členom a_i. Koristno bi bilo torej hraniti vrednost najmanjšega takega člena (v doslej pregledanem delu zaporedja a), pri katerem se konča kakšno naraščajoče podzaporedje dolžine k. Recimo tej vrednosti z_k. Pri $k = 0$ si lahko mislimo $z_k = -\infty$.

Ko dobimo nov člen a_i, moramo torej najti največji tak k, pri katerem je $z_k < a_i$, potem pa vemo, da lahko neko naraščajoče zaporedje dolžine k (z zadnjim členom z_k) podaljšamo s členom a_i, tako da bo $d_i = k + 1$. Ni se težko prepričati, da je zaporedje z_k strogo naraščajoče ($z_0 < z_1 < z_2 < \ldots$), zato lahko največji primerni k poiščemo kar z bisekcijo. Pri tako dobljenem k-ju imamo $z_k < a_i \leq z_{k+1}$. Ker je a_i manjši od (dosedanjega) z_{k+1} in ker smo pravkar ugotovili, da se pri členu a_i konča neko naraščajoče podzaporedje dolžine $k+1$, lahko torej po novem popravimo z_{k+1} na a_i, tako da bo z_{k+1} tudi po novem res vseboval najmanjši doslej znani člen, pri katerem se konča kakšno naraščajoče podzaporedje dolžine $k + 1$. Tako imamo naslednji postopek:

$z_0 := -\infty$; $K := 0$;
for $i := 1$ **to** n:
  z bisekcijo poišči tak k (od 0 do K), da je $z_k < a_i \leq z_{k+1}$
    (če je treba, si mislimo $z_{K+1} = \infty$);
  $d_i := k + 1$; $z_{k+1} := a_i$;
  if $k = K$ **then** $K := k + 1$;

V spremenljivki K hranimo dolžino najdaljšega doslej znanega naraščajočega podzaporedja, tako da vemo, do kod ima smisel iskati k-je. Ker nam bisekcija vzame vsakič le $O(\log n)$ časa, je časovna zahtevnost tega postopka le še $O(n \log n)$.[5]

Vrnimo se zdaj k prvotni nalogi. Za vsak člen a_i znamo torej izračunati d_i, dolžino najdaljšega naraščajočega podzaporedja, ki se konča s členom a_i. Na enak način lahko izračunamo tudi dolžino najdaljšega padajočega podzaporedja, ki se začne s členom a_i — recimo tej dolžini d'_i. (Uporabimo enak postopek kot prej, le da zaporedje a pregledujemo od konca proti začetku.)

Podzaporedje, po kakršnem sprašuje naloga, mora najprej nekaj časa naraščati in nato nekaj časa padati; vmes je torej nek največji element, recimo a_i, pred tem pa imamo naraščajoče podzaporedje (ki se konča pri a_i), za njim pa padajoče podzaporedje (ki se začne pri a_i). Za prvi del torej vemo, da je lahko dolg največ d_i, za drugi del pa, da je dolg največ d'_i. Podzaporedje, po kakršnem sprašuje naloga, z vrhom pri a_i je torej lahko dolgo največ $d_i + d'_i - 1$ (-1 potrebujemo zato, ker bi sicer člen a_i šteli dvojno). Vse, kar moramo torej še narediti, je, da gremo v zanki po vseh i in pogledamo, katera od tako dobljenih dolžin $d_i + d'_i - 1$ je največja.

[5] S problemom najdaljšega naraščajočega podzaporedja smo se na naših tekmovanjih že srečali; gl. rešitev naloge 1992.3.3 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj, 1988–2004* (str. 148–52) in tam navedeno literaturo. Ker so pri naši nalogi velikosti knjig cela števila od 1 do n, bi se dalo rešitev še izboljšati, če bi namesto bisekcije po tabeli z uporabili van Emde Boasovo drevo; časovna zahtevnost bi bila potem le še $O(n \log \log n)$.

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

void NajdaljseNarascajoceZaporedje(const vector<int> &a, vector<int> &d)
{
   vector<int> z; z.push_back(0);
   int n = a.size(), k = 0; d.resize(n);
   for (int i = 0; i < n; i++)
   {
      int ai = a[i];
      // Poiščimo najdaljše táko naraščajoče podzaporedje,
      // ki se konča z vrednostjo, manjšo od a[i].
      int L = 0, R = k + 1;
      while (R - L > 1) {
         // Na tem mestu velja a[L] < a[i] ≤ a[R]. (Za R = k + 1 si mislimo a[R] = ∞.)
         int M = (L + R) / 2;
         if (z[M] > ai) R = M; else L = M; }
      // Najdaljše táko naraščajoče podzaporedje, čigar zadnji element
      // je manjši od a[i], je dolgo L členov; torej ga lahko s členom a[i]
      // podaljšamo v naraščajoče podzaporedje dolžine L + 1 (to pa je enako R).
      if (R > k) z.push_back(ai), k++; else z[R] = ai;
      d[i] = R;
   }
}

int main()
{
   // Preberimo vhodne podatke.
   int n; scanf("%d", &n);
   vector<int> a(n), nnz, npz;
   for (int i = 0; i < n; i++) scanf("%d", &a[i]);
   // Naj bo nnz[i] dolžina tistega najdaljšega naraščajočega podzaporedja,
   // ki se konča s členom a[i].
   NajdaljseNarascajoceZaporedje(a, nnz);
   // Obrnimo a in poženimo isti postopek še enkrat. Tako bo (gledano z
   // vidika prvotnega zaporedja a) npz[n - 1 - i] dolžina tistega najdaljšega
   // padajočega podzaporedja, ki se začne s členom a[i].
   reverse(a.begin(), a.end());
   NajdaljseNarascajoceZaporedje(a, npz);
   // Poiščimo dolžino najdaljšega zaporedja, po katerem sprašuje naloga.
   int naj = 0;
   for (int i = 0; i < n; i++)
      // Seštejmo dolžino naraščajočega podzaporedja s koncem pri a[i] in
      // padajočega podzaporedja z začetkom pri a[i].
      naj = max(naj, nnz[i] + npz[n - 1 - i]);
   // Izpišimo rezultat.
   printf("%d\n", naj - 1); return 0;
}
```

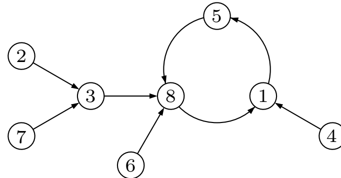
5. Posredne volitve

Vhodne podatke si lahko predstavljamo kot graf, v katerem imamo za vsakega državljana u po eno točko (ki ji tudi recimo u) in še usmerjeno povezavo od u do točke

$g(u)$, ki predstavlja njegovega zastopnika. Tako imamo torej n točk in prav toliko povezav; vsaka točka ima natanko eno izhodno povezavo.

Recimo, da začnemo v neki točki u in se premikamo naprej po izhodnih povezavah. Ker ima vsaka točka natanko eno izhodno povezavo, nimamo pri tem nobene izbire, kako nadaljevati pot; možna je ena sama pot z začetkom pri u. To je $u \to g(u) \to g(g(u)) \to g(g(g(u))) \to \ldots$ in tako naprej. To zaporedje se nadaljuje v neskončnost, torej se morajo točke v njem prej ali slej začeti ponavljati, saj ima naš graf le n različnih točk. Čim se neka točka v zaporedju pojavi drugič, to pomeni, da iz tiste točke po nekaj korakih (lahko tudi v enem samem koraku) pridemo nazaj v isto točko, torej je tu na grafu cikel in se bo naša pot odtlej le še v nedogled vrtela naprej po tem ciklu.

Če bi namesto pri u našo pot začeli pri kakšni drugi točki, na primer v, nam zdaj enak razmislek pove, da bi prej ali slej tudi tista pot prišla do cikla — lahko do istega kot pot iz u, lahko pa do kakšnega drugega. Tako torej vidimo, da je naš graf sestavljen iz ene ali več ločenih komponent, pri čemer vsako komponento tvori en usmerjen cikel in mogoče še eno ali več dreves, ki so pripeta na točke cikla in v katerih so vse povezave usmerjene k ciklu. Primer take komponente kaže naslednja slika (to je graf iz primera v besedilu naloge):



Kaj ta struktura grafa pomeni za postopek podajanja kroglic, ki ga opisuje besedilo naloge? Pot $u \to g(u) \to g(g(u)) \to \ldots$ pravzaprav opisuje, kako se premika med volilci tista kroglica, ki jo je na začetku (pred prvo fazo) imel volilec u. Ker smo videli, da ta pot sčasoma pride v cikel, to pomeni, da bo tudi u-jeva kroglica sčasoma pristala na tem ciklu, kjer si jo bodo potem v nedogled podajali volilci, ki tvorijo ta cikel. Ker se to zgodi pri vsakem u, to pomeni, da bodo tisti volilci, ki ne ležijo na ciklih, sčasoma vsi ostali brez kroglic in bodo izpadli iz volitev.

Tisti volilci pa, ki ležijo na kakšnem ciklu, ne morejo nikoli izpasti: na začetku je imel vsak od jih po eno kroglico in te kroglice se v vsakem koraku ciklično zamaknejo za eno mesto naprej po ciklu, tako da ima tudi v naslednjem koraku še vedno vsak od njih po eno od teh kroglic. Tako nikoli nihče od njih ne ostane brez kroglic (se jim pa lahko seveda število kroglic še poveča, ker bodo sčasoma na cikel prišle kroglice iz dreves, ki so pripeta na ta cikel).

Naj bo d_u razdalja med točko u in njej najbližjo točko na kakšnem ciklu; če u že sama leži na ciklu, je $d_u = 0$. To torej pomeni, da bo u-jeva kroglica po natanko d_u fazah prišla na cikel (in se odtlej gibala po ciklu). Maksimum tega po vseh u je torej število faz, ki je potrebno za to, da vse kroglice pridejo na cikle; do takrat volilci še izpadajo, po tistem pa ne več. Število K, po katerem sprašuje naloga, je torej ravno $K = \max_u d_u$.[6]

[6] O tem se lahko bolj formalno prepričamo takole. Vzemimo nek tak u, za katerega je $d_u = K$. Označimo pot od njega do cikla z $u = u_0 \to u_1 \to \ldots \to u_K$. Zadnja od teh točk, u_K, torej leži

Ostane nam še izračun funkcije f_K, po kateri tudi sprašuje naloga. Namesto da se za vsak u vprašamo, koliko kroglic ima po K fazah (to je vrednost $f_K(u)$), se je lažje za vsak u vprašati, kateri volilec ima po K fazah tisto kroglico, ki jo je imel na začetku (pred prvo fazo) volilec u. Vemo, da u-jeva kroglica po d_u fazah doseže cikel, odtlej pa se le še premika naprej po tem ciklu, torej naredi še $K - d_u$ korakov naprej po ciklu. Če je cikel dolg c točk (in povezav), je to isto, kot če bi naredila le $(K - d_u) \bmod c$ korakov naprej po ciklu. Tako torej vemo, pri kom je po K fazah u-jeva kroglica. Če ta razmislek ponovimo za vsak u, bomo točno vedeli, koliko kroglic ima kdo.

Da bomo ta postopek lahko izvedli učinkovito, si je koristno za vsak u poleg razdalje do cikla d_u zapomniti tudi številko točke, ki jo iz u po teh d_u korakih dosežemo. Poleg tega je koristno tudi za vsak cikel imeti seznam točk v njem, za vsako točko na ciklu pa podatek o tem, na katerem ciklu leži in katera po vrsti je ta točka v seznamu točk tega cikla. S temi podatki bomo lahko zelo učinkovito prišli najprej od u do najbližje točke cikla, nato pa še skočili za $(K - d_u) \bmod c$ korakov naprej po ciklu. Pazimo tudi na to, da ko za vsako točko u sledimo izhodnim povezavam, da ugotovimo, do katerega cikla se pride iz nje, nam ni treba nujno iti čisto do cikla; ustavimo se lahko, čim naletimo na neko tako točko, za katero smo že nekoč prej ugotovili, do katerega cikla se pride iz nje (in kako daleč je ta cikel). To nam zagotavlja, da se bomo z vsako povezavo grafa ukvarjali le $O(1)$-krat in časovna zahtevnost naše rešitve je le $O(n)$.

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

struct Tocka
{
   int g;             // zastopnik
   int f = 0;         // število kroglic po K fazah
   int uc;            // najbližja točka na ciklu
   int d = −1;        // razdalja do uc
   int stCikla = −1, stNaCiklu = −1;  // uc == cikli[stCikla][stNaCiklu]
};

int main()
{
   // Preberimo število točk.
   int n; scanf("%d", &n);

   // Preberimo podatke o zastopnikih.
   vector<Tocka> t {n};
   for (const auto &T : t) { scanf("%d", &T.g); T.g−−; }

   // Za vsako točko sledimo poti do cikla.
   vector<vector<int>> cikli;
```

na ciklu, njena predhodnica u_{K-1} pa še ne. Za $t = 0, \ldots, K-1$ vidimo, da ima po končanih t fazah volilec u_{K-1} med drugim tisto kroglico, ki jo je imel na začetku (pred prvo fazo) volilec u_{K-1-t}. Po teh fazah torej volilec u_{K-1} še ne izpade. Kaj pa po K-ti fazi? Če bi imel takrat u_{K-1} še kakšno kroglico, to pomeni, da obstaja neka točka v, od katere je pot do u_{K-1} dolga K korakov, toda iz take v bi bila potemtakem pot do najbližjega cikla dolga $K+1$ korakov, to pa je protislovje, saj smo za K vzeli maksimum dolžine poti do cikla po vseh možnih točkah. Tako torej vidimo, da je res šele K-ta faza (in ne že kakšna zgodnejša) zadnja, v kateri izpade kak volilec.

```
  vector<int> pot; int K = 0;
  for (int u = 0; u < n; u++)
  {
    if (t[u].d >= 0) continue; // že poznamo
    // Sledimo povezavam u → g(u), dokler ne dosežemo cikla ali pa
    // vsaj neke točke, za katero že poznamo oddaljenost od cikla.
    // Točke na tej poti začasno dodajamo v vektor „pot“ in jim
    // razdaljo d začasno spremenimo iz −1 na −2.
    pot.clear();
    int v = u;
    while (t[v].d == −1) {
      pot.push_back(v); t[v].d = −2;
      v = t[v].g; }
    if (t[v].d == −2)
    {
      // v je prva točka na tej poti, ki smo jo videli dvakrat,
      // torej smo našli nov cikel.
      int i = 0; while (pot[i] != v) i++;
      int stCikla = (int) cikli.size();
      cikli.emplace_back(pot.begin() + i, pot.end());
      pot.resize(i);
      auto &cikel = cikli.back();
      for (i = 0; i < cikel.size(); i++) {
        v = cikel[i]; auto &V = t[v];
        V.uc = v; V.stCikla = stCikla; V.stNaCiklu = i; V.d = 0; }
    }
    // Pot od u do v ni del cikla, pač pa je za v znano, do katerega cikla
    // se pride iz nje; pojdimo nazaj po tej poti in si za vsako
    // točko zapomnimo razdaljo do najbližjega cikla (za v je že znana).
    for (int i = pot.size() − 1; i >= 0; −−i)
    {
      v = pot[i]; auto &V = t[v]; const auto &W = t[V.g];
      V.uc = W.uc; V.stCikla = W.stCikla; V.stNaCiklu = W.stNaCiklu;
      V.d = W.d + 1;
    }
    if (t[u].d > K) K = t[u].d;
  }
  // Izračunajmo razporeditev kroglic po K fazah.
  for (const auto &T : t) {
    const auto &cikel = cikli[T.stCikla];
    t[cikel[(T.stNaCiklu + K − T.d) % cikel.size()]].f++; }
  // Izpišimo rezultate.
  printf("%d\n", K);
  for (int u = 0; u < n; u++) printf("%d%c", t[u].f, u == n − 1 ? '\n' : ' ');
  return 0;
}
```

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Avtobus

Dolžino niza p_i označimo s $|p_i|$. Da napišemo in pošljemo ime postaje i, potrebujemo $|p_i|+1$ sekund; torej, če vožnja do nje traja manj kot $|p_i|+1$ sekund, njenega imena ne bomo mogli izpisati, sicer pa lahko. Postaje lahko pregledujemo po naraščajočih i, dokler ne najdemo prve take, pri kateri velja $|p_i| + 1 \leq d_1 + \ldots + d_i$. Vsote $d_1 + \ldots + d_i$ nam seveda ni treba računati pri vsakem i znova, ampak jo hranimo v neki spremenljivki in ji vsakič le prištejemo d_i trenutne postaje. Oglejmo si primer take rešitve v C++:

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

vector<string> imena;    // p_i
vector<int> casi;        // d_i

int main()
{
   int casVoznje = 0;
   for (int i = 0; i < imena.size(); i++)
   {
      casVoznje += casi[i];
      if (imena[i].length() + 1 <= casVoznje) {
         cout << imena[i] << endl; return 0; }
   }
   cout << "doma" << endl;
   return 0;
}
```

Še primer rešitve v pythonu:

```
imena = [...]    # p_i
casi = [...]     # d_i

def KajIzpisati():
   casVoznje = 0
   for i in range(len(imena)):
      casVoznje += casi[i]
      if len(imena[i]) + 1 <= casVoznje: return imena[i]
   return "doma"
print(KajIzpisati())
```

2. Jedilnik

Preprosta rešitev je, da uporabimo dve gnezdeni zanki. V zunanji zanki (po i) gremo po vseh elementih jedilnika; v notranji zanki (po d) pa gremo od trenutnega elementa naprej, dokler ne najdemo naslednje pojavitve iste jedi. Pri tem pazimo na to, da se jedilnik ciklično ponavlja na vsakih n dni (kjer je n dolžina jedilnika). Če smo začeli pri dnevu i in se premaknili za d dni naprej, nas zanima jed na dan $i + d$, v jedilniku pa jo najdemo na indeksu $(i + d) \bmod n$. Notranja zanka se ustavi najkasneje pri $d = n$, saj se po n dneh ponovi cel jedilnik.

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

vector<int> KdajSpet(const vector<string>& jedilnik)
{
   int n = jedilnik.size();
   vector<int> rezultati;
   for (int i = 0; i < n; i++)
   {
      int d = 1;
      while (jedilnik[(i + d) % n] != jedilnik[i]) d++;
      rezultati.push_back(d);
   }
   return rezultati;
}
```

Zapišimo to rešitev še v pythonu:

```
def KdajSpet(jedilnik):
   n = len(jedilnik); rezultati = [ ]
   for i in range(n):
      d = 1
      while jedilnik[(i + d) % n] != jedilnik[i]: d += 1
      rezultati.append(d)
   return rezultati
```

Slabost te rešitve je, da v najslabšem primeru (če so v jedilniku vse jedi različne in se torej vsaka ponovi šele po n dnevih) notranja zanka naredi vsakič po $n - 1$ iteracij in je zato časovna zahtevnost tega postopka kar $O(n^2)$.

Boljša rešitev je, da si pri branju jedilnika za vsako jed zapomnimo, kdaj smo jo nazadnje videli. Primerna podatkovna struktura za to je slovar (ki je v praksi ponavadi implementiran kot razpršena tabela). Ko recimo na dan i naletimo na neko jed, pogledamo v slovar; če tam vidimo, da smo jo nazadnje videli na dan p, si lahko zdaj (v izhodnem seznamu) zapišemo, da se jed z dneva p ponovi po $i - p$ dneh. Z zanko po i je koristno iti do $2n$, kar nam zagotovi, da bomo vsako jed zanesljivo videli vsaj dvakrat. Oglejmo si primer takšne rešitve v C++, kjer lahko kot slovar uporabimo razred unordered_map iz standardne knjižnice:

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

vector<int> KdajSpet2(const vector<string>& jedilnik)
{
   int n = jedilnik.size();
   unordered_map<string, int> kdajPrej;
   vector<int> rezultati; rezultati.resize(n);
   for (int i = 0; i < 2 * n; i++)
   {
      const string &jed = jedilnik[i % n];
      if (auto it = kdajPrej.find(jed); it != kdajPrej.end())
         if (int prej = it->second; prej < n)
            rezultati[prej] = i - prej;
```

```
    kdajPrej[jed] = i;
  }
  return rezultati;
}
```

Podobno lahko naredimo tudi v pythonu, kjer je slovar še bolj pri roki:

```
def KdajSpet2(jedilnik):
  n = len(jedilnik); rezultati = [−1] * n
  kdajPrej = {}
  for i in range(2 * n):
    jed = jedilnik[i % n]
    prej = kdajPrej.get(jed, n)
    if prej < n: rezultati[prej] = i − prej
    kdajPrej[jed] = i
  return rezultati
```

3. Trikotniki

(1) Najpreprostejša rešitev je, da pri poizvedbi pregledamo vse celice trikotnika, na katerega se poizvedba nanaša. V neki spremenljivki hranimo največjo vrednost doslej in jo sproti popravljamo, ko zagledamo kakšno še večjo vrednost. Za pregled trikotnika uporabimo dve gnezdeni zanki, eno po vrsticah in drugo po celicah znotraj vrstice. Spotoma pazimo še na to, da nekateri deli trikotnika mogoče štrlijo ven iz mreže in do tistih celic tabele ne smemo dostopati (ker sploh ne obstajajo):

```
int Poizvedba(int x, int y, int v)
{
  int naj = T[x][y];
  for (int dy = 0; dy < v && y + dy < n; dy++)
    for (int dx = 0; dx < v − dy && x + dx < n; dx++)
      if (T[x + dx][y + dy] > naj) naj = T[x + dx][y + dy];
  return naj;
}
```

Časovna zahtevnost te rešitve je $O(v^2)$, saj mora pregledati vse celice trikotnika, teh pa je $v+(v-1)+(v-2)+\ldots+2+1 = v(v+1)/2$. Do boljših rešitev lahko pridemo, če si (kot svetuje že besedilo naloge) vnaprej izračunamo maksimume nekaterih območij tabele in jih nato pri poizvedbi na primeren način skombiniramo v maksimum ravno tistega trikotnega območja, po katerem sprašuje trenutna poizvedba.

(2) Vsako vrstico lahko na primer razdelimo na „bloke“, dolge po približno $\sqrt{n}$ celic (tako je tudi blokov v vsaki vrstici približno $\sqrt{n}$). Za vsak blok si zapomnimo maksimum vrednosti po vseh celicah v bloku. Ta predpriprava nam vzame $O(n^2)$ časa in zasede $O(n\sqrt{n})$ prostora.

Pri poizvedbi lahko razmišljamo podobno kot prej, le da ko smo pri prvi celici kakšnega bloka, preverimo, če ta blok v celoti leži v našem trikotniku; če da, vzamemo maksimum bloka (ki smo si ga izračunali vnaprej) in nato preostanek bloka preskočimo (nadaljujemo pri prvi celici naslednjega bloka). Tako moramo posamično pregledati le tiste celice, ki ležijo v trikotniku, ne leži pa cel njihov blok v trikotniku; takih je največ $\sqrt{n}$ na začetku vrstice in $\sqrt{n}$ na koncu vrstice. Vmes pa je v vrstici še največ $\sqrt{n}$ celih blokov. Tako imamo z vsako vrstico $O(\sqrt{n})$ dela, s celim trikotnikom pa $O(v\sqrt{n})$.

(3) Podobna, a še boljša rešitev je naslednja: naredimo podobno kot zgoraj, le da z bloki dolžine 2, 4, 8, 16 in tako naprej. To si lahko predstavljamo tako, kot da smo nad vsako vrstico zgradili binarno drevo, ki vsebuje maksimume vse daljših blokov (katerih dolžine so potence števila 2). Priprava vseh teh maksimumov nam vzame $O(n^2)$ časa in zasede $O(n^2)$ prostora.

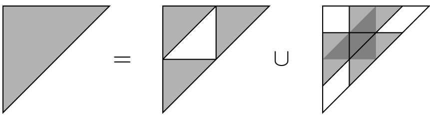
Pri poizvedbi opazimo, da lahko zdaj vsako vrstico našega trikotnika razdelimo na bloke tako, da uporabimo največ dva bloka posamezne dolžine (enega na začetku vrstice in enega na koncu). Dolžine blokov so 2^k za $k = 0, 1, \ldots, \lfloor \log_2 n \rfloor$, torej moramo pregledati približno $O(\log n)$ blokov, da dobimo maksimum ene vrstice trikotnika. Časovna zahtevnost ene poizvedbe je torej $O(v \log n)$.

(4) Rešitev lahko še izboljšamo, če smo pripravljeni žrtvovati še malo več pomnilnika. Za vsako celico (x, y) naše tabele in za vsak k od 0 do $\lfloor \log_2 n \rfloor$ izračunajmo maksimum vrednosti v bloku dolžine 2^k, ki se začne pri celici (x, y). To je torej podobno prejšnji rešitvi, le da se bloki zdaj prekrivajo. Pri vsakem k imamo torej $O(n^2)$ blokov in predpriprava podatkov nam zdaj vzame $O(n^2 \log n)$ časa ter prav toliko prostora.

Pri poizvedbi spet razdelimo trikotnik na vrstice, nato pa v vsaki vrstici vzemimo največji tak k, za katerega je $2^k \leq$ od dolžine vrstice. Če torej skombiniramo dva bloka dolžine 2^k — enega, ki se začne na začetku vrstice, in enega, ki se konča na koncu vrstice — bosta skupaj pokrila celo vrstico (malo se bosta tudi prekrivala med sabo, kar pa nas za izračun maksimuma nič ne moti). Tako moramo torej v vsaki vrstici trikotnika pregledati le dva bloka in časovna zahtevnost poizvedbe je le še $O(v)$.

(5) Še boljša rešitev pa je naslednja: vnaprej izračunajmo maksimume trikotnih območij prav take oblike, na kakršne se nanašajo poizvedbe pri tej nalogi, za vse možne položaje zgornjega levega kota trikotnika in za velikosti oblike $v = 2^k$. Pri vsakem k imamo torej $O(n^2)$ trikotnikov, zato nam ta predpriprava podatkov vzame $O(n^2 \log n)$ časa in prostora.

Pri poizvedbi s trikotnikom velikosti v nato vzemimo največji tak k, za katerega je $2^k \leq v$. Naš trikotnik velikosti v lahko popolnoma pokrijemo s šestimi trikotniki velikosti 2^k, kot vidimo na spodnjem primeru:



Slika kaže, kako lahko trikotnik velikosti v pokrijemo s šestimi trikotniki velikosti $v/2$. Prvi trije manjši trikotniki pokrijejo tri četrtine večjega, zadnjo četrtino (ki je na desni sliki osenčena temno sivo) pa pokrijemo z drugimi tremi manjšimi trikotniki. Pri naših poizvedbah manjši trikotniki nimajo nujno stranice $v/2$, pač pa stranico 2^k, ki je $\geq v/2$ (saj smo za 2^k vzeli največjo tako potenco števila 2, ki je $\leq v$). S takimi bo večji trikotnik še toliko lažje pokriti (naši trikotniki velikosti 2^k se bodo še malo bolj prekrivali med seboj kot na gornji sliki, kar pa nas pri računanju maksimuma nič ne ovira).

Ker smo za vsakega od šestih manjših trikotnikov že vnaprej izračunali maksimum vrednosti v njem, moramo zdaj le še poiskati največjega izmed teh šestih

maksimumov, pa imamo rezultat, po katerem sprašuje poizvedba. Tako nam torej posamezna poizvedba vzame le še $O(1)$ časa.

4. Točke in koši

Naj bo $f(k)$ število načinov, na katere se dá izraziti k kot vsoto števil 1, 2 in 3. Če je prvi seštevanec enak a, morajo ostali seštevanci dati vsoto $k-a$, to pa je mogoče narediti na $f(k-a)$ načinov. Za prvi seštevanec so, tako kot za vsakega drugega, le tri možnosti: a je lahko 1, 2 ali 3. Vsoto k lahko torej dobimo na $f(k-1)$ takih načinov, pri katerih ima prvi seštevanec vrednost 1, na $f(k-2)$ takih načinov, pri katerih ima prvi seštevanec vrednost 2, in še na $f(k-3)$ takih načinov, pri katerih ima prvi seštevanec vrednost 3. Vsega skupaj imamo torej $f(k-1)+f(k-2)+f(k-3)$ načinov, kako dobiti vsoto k. Tako smo dobili zvezo

$$f(k) = f(k-1) + f(k-2) + f(k-3).$$

Treba je še nekaj pazljivosti pri robnih primerih. Vsoto 0 je mogoče dobiti na en način, namreč tako, da sploh ne damo nobenega koša; za negativne k pa vsote sploh ni mogoče dobiti. Vzeti moramo torej $f(0) = 1$ in $f(k) = 0$ za $k < 0$. To funkcijo lahko zelo preprosto računamo z rekurzijo:

```
int Kosi(int m)
{
   if (m < 0) return 0;
   else if (m == 0) return 1;
   else return Kosi(m − 1) + Kosi(m − 2) + Kosi(m − 3);
}
```

Ali v pythonu:

```
def Kosi(m):
   if m < 0: return 0
   elif m == 0: return 1
   else: return Kosi(m − 1) + Kosi(m − 2) + Kosi(m − 3)
```

Slabost te rešitve je, da računa iste stvari po večkrat. Na primer, klic Kosi(m) pokliče Kosi(m − 1), ta pa Kosi((m − 1) − 1), torej Kosi(m − 2). Toda Kosi(m − 2) se pokliče kasneje še enkrat neposredno iz Kosi(m). Sčasoma je takega ponavljanja vse več in naša rešitev je zaradi tega pri večjih m neuporabno počasna. Bolje je, če si že izračunane vrednosti zapomnimo in jih kasneje ne računamo znova. Iz formul zgoraj vidimo, da bomo pri izračunu $f(k)$ potrebovali vrednosti $f(k-1)$, $f(k-2)$ in $f(k-3)$, torej bo najlažje, če bomo vrednosti funkcije $f(k)$ računali kar po vrsti po naraščajočih k. Pri tem si moramo vedno zapomniti le zadnje tri že izračunane vrednosti, starejše (za $k-4$, $k-5$ in tako naprej) pa lahko sproti pozabljamo, saj jih ne bomo več potrebovali. Tako dobimo naslednjo rešitev:

```
int Kosi2(int m)
{
   if (m < 0) return 0;
   int r[3] = { 1, 0, 0 };
   for (int k = 1; k <= m; k++)
      /* Na tem mestu za i = 0, 1, 2 velja: r[(k − i + 3) % 3] = f(k − i). */
      r[k % 3] = r[0] + r[1] + r[2];
   return r[m % 3];
}
```

In podobno v pythonu:

```
def Kosi2(m):
   if m < 0: return 0
   r = [1, 0, 0]
   for k in range(1, m + 1):
      r[k % 3] = r[0] + r[1] + r[2]
   return r[m % 3]
```

Zadnje tri vrednosti funkcije f torej hranimo v tabeli r in to tako, da je $f(k)$ shranjena v r[k % 3]. Ko izračunamo $f(k)$, lahko v tabeli r povozimo vrednost $f(k-3)$ z vrednostjo $f(k)$, saj je obema namenjena ista celica tabele in vrednosti $f(k-3)$ ne bomo več potrebovali.

Opazimo lahko, da je formula za $f(n)$ zelo podobna tisti za Fibonaccijeva števila, le da tam nastopa vsota prejšnjih dveh členov, tu pa vsota prejšnjih treh. Vrednostim $f(n)$, s kakršnimi se ukvarjamo pri tej nalogi, zato včasih pravijo „Tribonaccijeva števila“. Pokazati je mogoče, da zanje velja približno $f(n) \approx 0{,}618 \cdot 1{,}839^n$.

5. Povprečni položaj znaka

Naloge se lahko lotimo na več načinov. Ena možnost je, da beremo vhodne podatke po znakih; pri tem v nekaj spremenljivkah vzdržujmo trenutni položaj v vrstici (v spodnji rešitvi je to x), število pojavitev iskanega znaka v trenutni vrstici (nVrst) in vsoto njihovih položajev (sVrst). Poleg tega vzdržujmo tudi število pojavitev iskanega znaka v vsem doslej prebranem besedilu (nBes) in vsoto njihovih položajev (sBes). Vsak prebrani znak primerjamo z iskanim; če se ujema, povečamo števili pojavitev (nVrst in nBes) za 1 in prištejemo trenutni položaj k vsotama sVrst in sBes. Ko pridemo do konca trenutne vrstice, izpišemo število pojavitev in povprečni položaj (slednjega izračunamo tako, da vsoto položajev delimo s številom pojavitev, pri tem pa pazimo, da ne bomo delili z 0), nato pa trenutni položaj x postavimo nazaj na 0, da bomo pripravljeni na začetek naslednje vrstice. Ko pridemo do konca celotnega vhodnega besedila, pa na podoben način izpišemo še število pojavitev in povprečni položaj v celotnem besedilu. Oglejmo si primer take rešitve v C-ju:

```
#include <stdio.h>

int Povprecni(char c)
{
   int nBes = 0, sBes = 0, nVrst = 0, sVrst = 0, stVrstice = 0, x = 0;
   while (true)
   {
      /* Preberimo naslednji znak. */
      int ch = fgetc(stdin); x++;
      /* Smo na koncu vrstice? */
      if (ch == '\n' || (ch == EOF && x > 0))
      {
         x = 0; stVrstice++;
         printf("V %d. vrstici: %d pojavitev, povprečni položaj %d.\n",
               stVrstice, nVrst, sVrst / (nVrst <= 0 ? 1 : nVrst));
         nVrst = 0; sVrst = 0;
      }
      if (ch == EOF) break; else if (ch == '\n') continue;
```

```
    /* Ali je trenutni znak tisti, ki ga iščemo? */
    if (ch == c) nBes++, nVrst++, sBes += x, sVrst += x;
  }
  printf("Skupaj: %d pojavitev, povprečni položaj: %d.\n",
       nBes, sBes / (nBes <= 0 ? 1 : nBes));
}
```

Lahko pa namesto po znakih beremo vhodno besedilo po vrsticah. Pri vsaki vrstici se v notranji zanki sprehodimo po znakih, iščemo pojavitve iskanega znaka ter ustrezno povečujemo spremenljivki nVrst in sVrst. Na koncu vrstice izpišemo število pojavitev in povprečni položaj, nato pa ustrezno povečamo še število pojavitev in vsoto položajev v celotnem besedilu (nBes in sBes). Na koncu izpišemo še statistike za celotno besedilo, enako kot pri prejšnji rešitvi. Oglejmo si implementacijo te rešitve v pythonu:

```
import sys

def Povprecni(c):
  stVrstice = 0; nBes = 0; sBes = 0
  for vrstica in sys.stdin:
    stVrstice += 1; nVrst = 0; sVrst = 0
    for x in range(len(vrstica)):
      if vrstica[x] == c:
        nVrst += 1; sVrst += (x + 1)
    print("V %d. vrstici: %d pojavitev, povprečni položaj %d." % (
        stVrstice, nVrst, sVrst / max(nVrst, 1)))
    nBes += nVrst; sBes += sVrst
  print("Skupaj: %d pojavitev, povprečni položaj %d." % (
      nBes, sBes / max(nBes, 1)))
```

Naloge so sestavili: Collatz++, sestavljanka, buteljke — Nino Bašić; jabolka — Matija Grabnar; cevi, knjige, trikotniki — Tomaž Hočevar; pravokotnik — Vid Kocijan; palačinke — Filip Koprivec; povprečni položaj znaka — Mitja Lasič; jedilnik, točke in koši — Matija Lokar; brzinomer — Mark Martinec; križci in krožci — Polona Novak; popravljanje testov, tekoči trak, avtobus — Jure Slak; alfa bravo — Jasna Urbančič; pisalni stroj, posredne volitve — Janez Brank.

1. Žaba

V zanki bomo po vrsti pregledovali točke v vhodnem seznamu. Skok v celoti leži v mlaki, če ležita v njej tako začetna kot končna točka tega skoka, torej iščemo v vhodnem seznamu strnjeno podzaporedje vsaj dveh točk v mlaki. Ko najdemo prvi tak primer, recimo točki tocke[i] in tocke[i + 1], moramo še ugotoviti, kako daleč naprej se še nadaljujejo skoki znotraj mlake. Pojdimo torej naprej po zaporedju, dokler ne naletimo na kakšno točko zunaj mlake (ali pa pridemo do konca zaporedja); tam potem vemo, da se naše podzaporedje konča.

Oglejmo si primer implementacije take rešitve v C++. Točke in mlako predstavimo s strukturami, za seznam točk pa uporabimo razred **vector** iz standardne knjižnice.

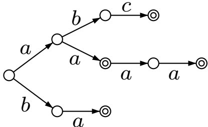
```
#include <vector>
using namespace std;

struct Tocka { int x, y; };
struct Mlaka { int x1, y1, x2, y2; };

// Pove, ali točka t leži v mlaki m.
bool JeZnotraj(const Tocka& t, const Mlaka &m)
{
  return ((m.x1 <= t.x && t.x <= m.x2) || (m.x2 <= t.x && t.x <= m.x1)) &&
         ((m.y1 <= t.y && t.y <= m.y2) || (m.y2 <= t.y && t.y <= m.y1));
}

vector<Tocka> vMlaki(const vector<Tocka>& tocke, const Mlaka& mlaka)
{
  // Poiščimo prvi skok (torej dve zaporedni točki) znotraj mlake.
  int i = 0, n = tocke.size();
  while (i + 1 < n && ! (JeZnotraj(tocke[i], mlaka) && JeZnotraj(tocke[i + 1], mlaka)))
    i++;
  // Če takega skoka ni, vrnimo prazen seznam.
  if (i >= n − 1) return {};
  // Nadaljujmo do prve točke, ki ni v mlaki.
  int j = i + 2; while (j < n && JeZnotraj(tocke[j], mlaka)) j++;
  // Iskani podseznam je tocke[i, i + 1, . . . , j − 1].
  return {tocke.begin() + i, tocke.begin() + j};
}
```

Pri preverjanju, ali točka leži v mlaki ali ne (funkcija JeZnotraj), se stvari malo zapletejo zato, ker ne vemo, ali v opisu mlake pomeni x1 levi rob in x2 desnega ali obratno, torej ne vemo, ali moramo preveriti pogoj $x_1 \leq x \leq x_2$ ali $x_2 \leq x \leq x_1$. Zato preverimo oba in smo zadovoljni, če je izpolnjen katerikoli od njiju. Podobno je tudi pri y-koordinatah. Druga možnost je, da bi podprogram vMlaki na začetku preveril, če je mlaka.x1 <= mlaka.x2, in če ni, bi vrednosti teh dveh atributov zamenjal (in podobno za y-koordinate).

2. Tipkanje

Naše vhodne nize označimo z $x_1, \ldots, x_n$. Zložimo jih v drevo (*trie*), ki ima na povezavah črke. Vsako vozlišče drevesa tako predstavlja neko možno stanje vnosnega polja med tipkanjem (koren predstavlja stanje, ko je vnosno polje prazno, npr. na začetku tipkanja). Ko pritisnemo tipko a, se iz trenutnega vozlišča prestavimo v enega od otrok po tisti povezavi, ki je označena s črko a; ko pritisnemo Backspace, pa se iz trenutnega vozlišča prestavimo v njegovega starša.

Naslednja slika kaže primer za nize *aa*, *aaaa*, *abc* in *ba*; vozlišča, pri katerih se konča kakšen od vhodnih nizov (torej tista, pri katerih pritisnemo Enter), so označena z dvojnim robom:



Da vnesemo vse vhodne nize, se moramo iz korena drevesa sčasoma sprehoditi do vseh listov (ti namreč predstavljajo naše vhodne nize x_i) in pri njih pritisniti Enter. Ker na število pritiskov te tipke ne moremo vplivati (v vsakem primeru jih potrebujemo natanko n), o njih v nadaljevanju ne bomo več razmišljali.

Pri tem sprehodu moramo načeloma vsako povezavo prehoditi dvakrat — prvič, da se spustimo po njej v poddrevo, v katero vodi ta povezava; in drugič, da se vzpnemo iz tega poddrevesa (po tistem, ko smo obskali že vse liste v njem in vnesli njim pripadajoče nize). Več kot toliko korakov ne potrebujemo, saj bi to pomenilo, da obiskujemo kakšne dele drevesa po večkrat ali pa celo tipkamo stvari, ki nas ne bodo pripeljale do nobenega vhodnega niza in jih bomo morali prej ali slej spet pobrisati.

Je pa mogoč še en prihranek: če obiščemo vsako povezavo dvakrat, bomo končali v korenu drevesa, kjer smo tudi začeli. Toda po tistem, ko obiščemo zadnji še neobiskani list, se nam ni treba vzpenjati v koren drevesa, saj smo takrat vnesli že vse nize in lahko takoj končamo. (V koren bi se morali vrniti, če bi naloga zahtevala, da moramo po koncu vnašanja vnosno polje pobrisati z ustreznim številom pritiskov na Backspace; toda naloga tega ne zahteva.) Da bomo imeli od tega prihranka čim več koristi, si je torej koristno za konec prihraniti najdaljšega izmed vhodnih nizov.

Kako bi zdaj potrebno število pritiskov na tipke zares izračunali? Ena možnost je, da res zgradimo drevo (*trie*), karkšno smo opisovali doslej, in preštejemo povezave v njem (vsako štejmo dvojno, na koncu odštejemo dolžino najdaljšega vhodnega niza ter prištejemo n pritiskov na Enter). Druga možnost pa je, da si pomagamo z rešitvijo naloge 2016.1.1, za katero smo že rekli, da je čisto podobna naši, le da je moral biti tam vrstni red vnašanja nizov enak kot v vhodnem seznamu.[7] Pri našem drevesu je, če odmislimo prihranek iz prejšnjega odstavka, vseeno, v kakšnem vrstnem redu obiščemo poddrevesa nekega vozlišča, zato se lahko domenimo, da jih bomo obiskovali v abecednem vrstnem redu. To ustreza temu, da tudi vhodne nize vnašamo v abecednem vrstnem redu. Zato lahko vhodni seznam uredimo po abecedi in na njem poženemo rešitev naloge 2016.1.1; ta nam pove, s koliko pritiski

[7] Gl. str. 41 v biltenu 2016.

na tipke bi se dalo vnesti vhodne nize v tem vrstnem redu. Prištejmo njenemu rezultatu dolžino zadnjega niza v seznamu (tega namreč ona na koncu pusti v vnosnem polju), pa imamo število pritiskov na tipke, s katerim lahko obiščemo celotno drevo, izpišemo vse nize in končamo v korenu. Nato moramo le še odšteti dolžino najdaljšega vhodnega niza, da upoštevamo prihranek iz prejšnjega odstavka.

3. Srečna števila

Če je n sodo število, že iz opisa naloge sledi, da ne more biti srečno, zato lahko v tem primeru takoj vrnemo **false**. Sicer pa si pripravimo seznam lihih števil do vključno n in na njem poženimo postopek iz besedila naloge. Če pri tem število n kdaj pobrišemo, lahko takoj zaključimo, da ni srečno; če pa pridemo do konca, ne da bi n kdaj pobrisali, vemo, da je srečno.

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

bool JeSrecno(int n)
{
  // Soda ali nepozitivna števila niso srečna.
  if (n < 1 || n % 2 == 0) return false;
  // Pripravimo seznam lihih števil do n.
  int dolzina = (n + 1) / 2;
  vector<int> s; s.reserve(dolzina);
  for (int i = 1; i <= n; i += 2) s.push_back(i);
  // Poženimo postopek iz besedila naloge.
  for (int k = 1; k < dolzina; k++)
  {
    int korak = s[k];
    // Pobrisali bomo vsak korak-ti element.
    // Ali je med njimi tudi zadnji element, torej n?
    if (dolzina % korak == 0) return false;
    // Ali je korak tako velik, da sploh ne bomo ničesar več pobrisali?
    if (korak > dolzina) break;
    // Pobrišimo zdaj te elemente.
    int dolzNova = 0;
    for (int i = 0; i < dolzina; i++)
      if ((i + 1) % korak != 0)
        s[dolzNova++] = s[i];
    dolzina = dolzNova;
  }
  // Če smo prišli do sem, ne da bi n pobrisali, je n srečno število.
  return true;
}
```

V zgornji rešitvi smo seznam hranili v vektorju (ki je pravzaprav nekakšna tabela). Da prihranimo čas pri brisanju elementov, ne bomo brisali vsakega posebej, pač pa gremo (pri vsakem koraku) le enkrat čez celoten seznam in kopiramo elemente na nove indekse v istem vektorju (pri tem sproti računamo novo dolžino uporabljenega dela vektorja — spremenljivka dolzNova).

Za več o srečnih številih glej npr. *The On-line Encyclopedia of Integer Sequences*, zaporedje A000959.

4. Vsota zmnožkov

Nalogo lahko rešujemo z dinamičnim programiranjem. Naj bo $f(k)$ največji rezultat, ki ga lahko dosežemo s postavljanjem operatorjev v izraz $a_1 \circ a_2 \circ \ldots \circ a_k$. (Naloga torej sprašuje po $f(n)$.) Ta rezultat bo vsota enega ali več zmnožkov; recimo, da se zadnji zmnožek začne s členom a_{i+1}. Izraz bo torej oblike $a_1 \circ \ldots \circ a_i + a_{i+1} \cdot a_{i+2} \cdot \ldots \cdot a_k$. Da bo njegova vrednost čim večja, moramo operatorje v začetni del izraza, $a_1 \circ \ldots \circ a_i$, postaviti tako, da bo tudi vrednost tega začetnega dela čim večja. Za ta podizraz pa po definiciji vemo, da je njegova največja možna vrednost $f(i)$. Če se torej odločimo zadnji + postaviti med a_i in a_{i+1} (od tam naprej do a_k pa same operatorje $\cdot$), bo največja možna vrednost tako dobljenega izraza enaka $f(i) + a_{i+1} \cdot a_{i+2} \cdot \ldots \cdot a_k$. Med vsemi možnimi i (od 1 do $k-1$) si moramo izbrati tistega, pri katerem bo ta vrednost največja.

Iz tega razmisleka vidimo, da pri izračunu $f(k)$ potrebujemo vrednosti $f(1), \ldots, f(k-1)$, zato je koristno funkcijo f računati po naraščajočih vrednostih k-ja in shranjevati že izračunane rezultate v tabelo, da jih kasneje ne bo treba računati ponovno, ko jih bomo spet potrebovali. Oglejmo si primer implementacije te rešitve v C++:

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

double Resi(const vector<double>& a)
{
  int n = (int) a.size();
  vector<double> f; f.resize(n + 1);
  f[0] = 0;
  for (int k = 1; k <= n; k++)
  {
    double prod = 1;
    // Izračunati moramo najboljši rezultat za a[0], ..., a[k − 1].
    // Preglejmo vse možne položaje zadnjega + pred številom a[k − 1].
    for (int i = k − 1; i >= 0; i−−)
    {
      // Recimo, da je zadnji + med a[i − 1] in a[i],
      // od tam do a[k − 1] pa so sami *.
      prod *= a[i]; // prod je zdaj produkt števil a[i], ..., a[k − 1].
      double kand = f[i] + prod;
      // Če je to najboljši kandidat doslej (pri tem k), si ga zapomnimo.
      if (i == k − 1 || kand > f[k]) f[k] = kand;
    }
  }
  return f[n];
}
```

Zanimiv poseben primer naše naloge nastopi, če so vsi a_i večji ali enaki 2. Tedaj je naloga trivialna, saj največji rezultat dobimo, če povsod uporabimo operator $\cdot$ (o tem se lahko prepričamo z indukcijo po n).

Razmislimo zdaj še o težji različici naše naloge, pri kateri smemo dodajati tudi oklepaje. Zdaj ni več nujno, da bo izraz kot celota na koncu vsota zmnožkov števil, ampak bo lahko tudi zmnožek vsot (seštevanci v teh vsotah pa so lahko spet zmnožki vsot in tako naprej). Če v izrazu kje nastopa vsota več kot dveh seštevancev ali pa

zmnožek več kot dveh faktorjev, jo lahko z dodatnimi oklepaji vedno predelamo tako, da bo imela le dva seštevanca ali faktorja; na primer: iz $a + b + c$ naredimo $((a + b) + c)$.

V nadaljevanju torej lahko predpostavimo, da bo tudi izraz kot celota na koncu bodisi vsota dveh podzirazov ali pa zmnožek dveh podizrazov. Prvi od teh podizrazov mora torej pokriti prvih nekaj števil našega zaporedja, recimo od a_1 do a_k, drugi pa preostale, torej od a_{k+1} do a_n. Za operator med tema podizrazoma lahko postavimo + ali · in ker v splošnem ne vemo, kateri bo dal večji rezultat, moramo preizkusiti oba.

Pri seštevanju je tako, da če hočemo čim večjo vsoto, morata biti tudi seštevanca čim večja. Ko torej razmišljamo o možnosti, da med oba glavna podizraza postavimo +, se znajdemo pred podproblemoma: kako postaviti operatorje in oklepaje v levi del podzaporedja in kako v desnega, da bo vsak od njiju imel največjo možno vrednost? To sta problema enake oblike kot na začetku, le s krajšim zaporedjem števil.

Pri množenju pa se pojavi še ena možnost: če je eden od faktorjev negativen, bo zmnožek tem večji, čim *manjši* bo drugi faktor. Kandidata za največji možni zmnožek sta tako dva: ena možnost je, da vsak od faktorjev zasede največjo možno vrednost, druga pa, da vsak od njiju zasede najmanjšo možno vrednost.

V vsakem primeru smo torej prišli do podproblemov, ki se nanašajo na neko strnjeno podzaporedje prvotnega zaporedja $a_1 \circ \ldots \circ a_n$. Videli pa smo tudi, da ni dovolj, če za vsako tako podzaporedje poiščemo največjo možno vrednost, ampak potrebujemo tudi najmanjšo, ker lahko pride prav kasneje pri iskanju čim večjega produkta pri večjih podizrazih.

Naj bo torej $f(i, j)$ največja, $g(i, j)$ pa najmanjša možna vrednost, ki jo lahko sestavimo iz podzaporedja $a_i \circ \ldots \circ a_j$, če vanj primerno postavimo oklepaje in operatorje. Dosedanji razmislek lahko povzamemo z naslednjimi rekurzivnimi zvezami:

$$f(i,j) = \max_{i \le k < j}\{f(i,k) + f(k+1,j), f(i,k) \cdot f(k+1,j), g(i,k) \cdot g(k+1,j)\}$$
$$g(i,j) = \min_{i \le k < j}\{g(i,k) + g(k+1,j), f(i,k) \cdot f(k+1,j), g(i,k) \cdot g(k+1,j)\}.$$

Podobno kot pri lažji različici naloge je tudi tu koristno že izračunane vrednosti f in g hraniti v tabelah, računamo pa jih lahko po naraščajoči dolžini podizraza (torej $j - i$). Tako dobimo rešitev s časovno zahtevnostjo $O(n^3)$ (rešiti moramo $O(n^2)$ podproblemov in pri vsakem imamo $O(n)$ dela, da pregledamo vse možne položaje k).

5. Izštevanka

Krog otrok lahko predstavimo s tabelo, pri čemer vsak element hrani številko tega otroka (od 1 do n) in število življenj, ki so mu še ostala. Na vsakem koraku simulacije premaknemo trenutni položaj p za k mest naprej, pri čemer upoštevamo, da tabela predstavlja ciklično zaporedje: od vsote $p + k$ torej obdržimo le ostanek po deljenju z n. Ko otroku pade število življenj na 0, ga pobrišemo iz tabele; ko ostane v njej en sam otrok, pa vrnemo njegovo številko.

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

int Izstevanka(int n, int k, int z)
```

```
{
  struct Otrok { int st, z; };

  // Inicializirajmo tabelo otrok.
  vector<Otrok> a; a.resize(n);
  for (int i = 0; i < n; i++) a[i] = {i + 1, z};
  int p = 0; // trenutni položaj v tabeli a

  while (a.size() > 1)
  {
    // Premaknimo se k korakov naprej.
    p = (p + k) % a.size();

    // Zmanjšajmo število življenj tega otroka.
    if (--a[p].z <= 0)
    {
      // Trenutni otrok bo izpadel iz igre.
      a.erase(a.begin() + p);

      // Postavimo se za eno mesto nazaj po krogu, tako da,
      // ko bomo v naslednji iteraciji glavne zanke začeli šteti naprej,
      // bo učinek enak, kot da bi bil ta otrok takrat še tam.
      p = (p + a.size() - 1) % a.size();
    }
    k++; // Povečajmo dolžino izštevanke za 1.
  }
  return a[0].st; // Vrnimo številko zadnjega preostalega otroka.
}
```

Ker začnemo z n otroki, ki imajo vsak po z življenj, mora simulacija teči $O(nz)$ korakov, preden izpadejo vsi otroci razen enega. Brisanje posameznega elementa iz vektorja (s katerim smo v gornji rešitvi predstavili seznam otrok) traja v najslabšem primeru $O(n)$ časa, torej porabimo še $O(n^2)$ časa, da pobrišemo vse otroke razen enega. Časovna zahtevnost te rešitve je torej $O(nz + n^2)$.

Namesto vektorja bi lahko uporabili kakšno drevesasto podatkovno strukturo, na primer rdeče-črno drevo, v katerem bi bili otroci urejeni po številkah (in s tem tudi po položaju vzdolž kroga). Pri vsakem vozlišču vzdržujmo tudi število otrok v poddrevesu, ki se začenja s tem vozliščem. S pomočjo tega dodatnega podatka se lahko v $O(\log n)$ časa premaknemo za k otrok naprej po krogu, pa tudi brisanje otroka vzame $O(\log n)$ časa. Časovna zahtevnost rešitve je v tem primeru $O(nz \log n)$. To je lahko precej bolje od rešitve z vektorjem, če je z majhen v primerjavi z n.

6. Temperature

Recimo, da na začetku vrednosti $t_1, \ldots, t_n$ uredimo naraščajoče in pobrišemo morebitne duplikate; ostane $u_1 < u_2 < \ldots < u_r$. Če m neke poizvedbe leži med u_{i-1} in u_i, je pogoj „temperatura mora biti vsaj m“ enakovreden pogoju „temperatura mora biti vsaj u_i“, saj med m in u_i ni v tabeli t nobene možne vrednosti. Podobno tudi, če je $m < u_1$, je pogoj „vsaj m“ enakovreden pogoju „vsaj u_1“. Tako je torej možnih le $O(n)$ zares različnih m-jev.

Če zdaj pri fiksnem m gledamo vse daljše k-je, je seveda pogoj, ki ga postavlja naša poizvedba, tem strožji (tem zahtevnejši), čim večji je k. Pri dovolj majhnem k (če ne drugega, pri $k = 0$) je odgovor na poizvedbo gotovo „da“, pri dovolj velikem (če ne prej, pa pri $k = n + 1$) pa gotovo „ne“. Vprašanje je torej le, pri katerem

k pride do tega preklopa; oz. z drugimi besedami, vprašanje je, katero (oz. kako dolgo) je najdaljše strnjeno zaporedje dni, ki imajo temperaturo vsaj m. Za $m = u_i$ označimo dolžino najdaljšega takega zaporedja s K_i.

To lahko računamo od nižjih m proti višjim. Na začetku si mislimo $m = u_1$ in najdaljše zaporedje s temperaturo vsaj m je dolgo kar n dni; to je $1..n$, celotno vhodno zaporedje. Tu imamo torej $K_1 = n$.

Nato m dvignimo na u_2. V zaporedju je vsaj en dan s temperaturo u_1, lahko pa jih je tudi več; pri teh dnevih nam zdaj $1..n$ razpade na krajše kose: na primer, če je temperatura u_1 dosežena na dneva a in b (za $a < b$), nam $1..n$ razpade na tri dele: $1..(a-1)$, $(a+1)..(b-1)$ in $(b+1)..n$. To so zdaj maksimalna strnjena zaporedja dni, ko je temperatura vsaj u_2; za K_2 si moramo zapomniti, kako dolg je najdaljši od teh treh delov.

Nato se m dvigne na u_3; naši deli razpadejo na še krajše dele in spet si moramo zapomniti, kako dolg je zdaj najdaljši od njih. Tako nadaljujemo, dokler se vsi deli popolnoma ne izpraznijo (kar se zgodi najkasneje tedaj, ko m naraste čez u_r). Tako imamo približno takšen postopek:

$Z := \{(1, n)\}$; $K_1 := n$;
for $i := 2$ **to** r:
  za vsak dan j, ki ima $t_j = u_i$:
    poišči v Z tisti par (ℓ, d), za katerega je $\ell \leq j \leq d$, in ga pobriši iz Z;
    if $\ell < i$ **then** dodaj v Z par $(l, i-1)$;
    if $i < d$ **then** dodaj v Z par $(i+1, d)$;
  $K_i := \max\{d - \ell + 1 \ : \ (l, r) \in Z\}$;

Vprašanje je, kako to početi čim bolj učinkovito. V zgornji prevdokodi je Z množica parov, ki predstavljajo maksimalne strnjene intervale dni s temperaturo vsaj u_j. V praksi je koristno predstaviti Z z kakšno uravnoteženo drevesasto strukturo, na primer rdeče-črnim drevesom, v katerem so pari urejeni naraščajoče. Ker imamo največ $O(n)$ parov, bo vsaka operacija na drevesu (iskanje para, dodajanje, brisanje) zahtevala $O(\log n)$ časa. Za računanje maksimuma v zadnji vrstici pa je koristno, če v vsakem vozlišču drevesa sproti vzdržujemo še maksimalno dolžino intervalov v celotnem poddrevesu, ki se začenja pri tem vozlišču. Tako bomo iskani maksimum lahko preprosto odčitali iz korena drevesa. Celoten zgornji postopek ima tako časovno zahtevnost $O(n \log n)$, prav toliko časa pa porabimo tudi za začetno urejanje temperatur, da iz $t_1, \ldots, t_n$ dobimo $u_1, \ldots, u_r$.

Razmislimo zdaj še o odgovarjanju na poizvedbe. Ko pride poizvedba (k, m), najprej z bisekcijo poglejmo, na katerem intervalu $(u_{i-1}, u_i]$ leži m (za potrebe tega razmisleka si predstavljajmo še $u_0 = -\infty$). Takrat potem vemo, da je najdaljše strnjeno zaporedje dni s temperaturo vsaj m dolgo K_i dni; odgovor na poizvedbo je tako enak odgovoru na vprašanje, ali je $k \leq K_i$. (Robni primer: če se izkaže, da je $m > u_r$, to pomeni, da v meritvah ni sploh nobenega dneva s temperaturo vsaj m, torej je odgovor na poizvedbo enak odgovoru na vprašanje, ali je $k \leq 0$.) Tako nam torej odgovor na posamezno poizvedbo vzame le $O(\log n)$ časa (zaradi bisekcije).

7. Zakon prve števke

Naša rešitev bo vhodna števila prebirala v zanki in sproti v tabeli `koliko` štela, koliko števil se začne na posamezno števko od 1 do 9. Na koncu moramo vsakega od

števcev v tabeli koliko deliti z njihovo vsoto ter pomnožiti s 100, da dobimo delež v odstotkih, ki ga moramo izpisati. Spodnja rešitev vsoto računa kar sproti (v spremenljivki skupaj), lahko pa bi jo tudi na koncu (po tistem, ko preberemo vsa vhodna števila, bi v zanki seštelі elemente tabele koliko).

Vprašanje je še, kako za trenutno vhodno število n izračunati njegovo prvo (najbolj levo) števko. Če je n že manjši od 10, to pomeni, da ima eno samo števko in njena vrednost je kar n. Sicer pa lahko n delimo z 10; celi del količnika je število, ki nastane, če n-ju odrežemo najbolj desno števko (enice). Na primer: če imamo $n = 567$, bo celi del po deljenju z 10 enak 56. To ponavljamo, dokler n ne postane manjši od 10, takrat pa vemo, da je ostala le še najbolj leva števka prvotnega števila.

```
#include <stdio.h>

int main()
{
  int koliko[10], skupaj = 0, n;

  // Postavimo števce na 0.
  for (n = 1; n <= 9; n++) koliko[n] = 0;

  // V zanki obdelajmo vsa števila.
  while ((n = Preberi()) > 0)
  {
    // Odrežimo n-ju vse števke razen najbolj leve.
    while (n >= 10) n /= 10;

    // Povečajmo števec za to števko in za vsa števila skupaj.
    koliko[n]++; skupaj++;
  }
  if (skupaj == 0) skupaj = 1; // Da se izognemo deljenju z 0.

  // Izpišimo rezultate.
  for (n = 1; n <= 9; n++)
    printf("Števka %d se pojavlja na prvem mestu pri %.2f %% števil.\n",
      n, 100.0 * koliko[n] / double(skupaj));

  return 0;
}
```

8. Bankomat

V enem dvigu lahko dobimo največ en bankovec za 10 evrov — če bi nam bankomat dal dva taka bankovca naenkrat, bi nam lahko isti znesek z manj bankovci izplačal tako, da bi nam namesto teh dveh desetakov dal en dvajsetak, to pa je v protislovju z zagotovilom naloge, da bankomat vedno izplača minimalno število bankovcev.

Podobno tudi vidimo, da lahko v enem dvigu dobimo največ dva bankovca za 20 evrov; če bi nam namreč bankomat kdaj izplačal tri take bankovce naenkrat, bi nam lahko enak znesek z manj bankovci izplačal tako, da bi nam namesto teh treh dvajsetakov izplačal en petdesetak in en desetak, kar je spet v protislovju z zagotovilom, da bankomat izplača minimalno število bankovcev.

Potrebujemo torej vsaj e dvigov, da bomo dobili dovolj desetakov. Pri vsakem takem dvigu lahko dobimo še en dvajsetak, dveh pa ne, kajti v tem primeru bi nam bankomat znesek $10 + 20 + 20 = 50$ izplačal z enim samim petdesetakom. Torej, če je $d \leq e$, lahko dobimo vse dvajsetake in desetake tako, da dvignemo d-krat po 30 evrov in $(e - d)$-krat po 10 evrov. Če pa je $d > e$, lahko s tistimi e dvigi, ki jih

že tako ali tako potrebujemo zaradi desetakov, dobimo še največ e dvajsetakov (če pri vseh teh dvigih zahtevamo po 30 evrov); nato pa za preostalih $d - e$ dvajsetakov potrebujemo še $\lfloor (d-e)/2 \rfloor$ dvigov po 40 evrov in na koncu, če je bil $d - e$ lih, še en dvig za 20 evrov.

Bankovce za 50 evrov lahko dobimo vse v enem dvigu; dovolj je že, če znesek $50 \cdot p$ dodamo h kateremu koli izmed prej omenjenih dvigov (tistih, s katerimi dvigujemo desetake in dvajsetake); če takih ni (ker je $d = e = 0$), pa pač za petsedetake izvedemo poseben dvig.

```
int SteviloDvigov(int e, int d, int p)
{
  // Dvigi za desetake (in mogoče nekaj dvajsetakov).
  int n = e;

  // Dodatni dvigi za preostale dvajsetake (če jih potrebujemo).
  if (d > e) n += (d - e + 1) / 2;

  // Dodaten dvig za petdesetake (če ga potrebujemo).
  if (n == 0 && p > 0) n += 1;

  return n;
}
```

9. Kontrolna vsota

Če število n delimo z 10, nam ostanek po deljenju dá ravno njegovo najbolj desno števko (enice), celi del količnika pa je število, ki ostane, če n-ju pobrišemo to najbolj desno števko. S to operacijo lahko v zanki režemo n-jeve števke od desne proti levi in jih sproti seštevamo. Če je tako dobljena vsota manjša od 10, je to že naša iskana kontrolna vsota, sicer pa na njej ponovimo isti postopek in tako dalje, dokler ne pridemo pod 10.

Vhodno število bi lahko na začetku prebrali v celoštevilsko spremenljivko (npr. tipa **int**) in jo potem obdelali po postopku iz prejšnjega odstavka; še bolje pa je, če ga beremo znak po znak in te znake sproti pretvarjamo v števke in jih seštevamo. Tako bo lahko naš program obdelal tudi števila, ki so prevelika za v spremenljivko tipa **int**.

```
#include <stdio.h>

int main()
{
  int n = 0;
  while (true)
  {
    // Preberimo naslednji znak.
    int c = fgetc(stdin);

    // Če ni števka, je vhodnega števila konec.
    if (c < '0' || c > '9') break;

    // Prištejmo vrednost te števke k n.
    n += (c - '0');
  }
  // Postopek s seštevanjem števk moramo ponavljati, dokler
  // vsota ne pade pod 10 — takrat imamo želeno kontrolno vsoto.
  while (n >= 10)
```

```
  {
    int m = 0; // V m bomo izračunali vsoto števk n-ja.
    // V zanki režimo n-jeve števke od desne proti levi in jih seštevajmo.
    while (n > 0) { m += n % 10; n /= 10; }
    n = m; // Starega n-ja ne potrebujemo več, namesto njega si zapomnimo vsoto števk.
  }
  // Izpišimo rezultat.
  printf("%d\n", n); return 0;
}
```

10. Stave

Besedilo naloge govori o skupnem številu golov in o razliki po rednem delu tekme. Predstavljamo si torej lahko, da poleg rednega dela obstaja tudi izredni del tekme, na katerem so tudi lahko doseženi kakšni goli. Tako pridemo do dveh različic naloge glede na to, ali v „skupno število golov“ štejemo tudi tiste iz izrednega dela tekme ali samo tiste iz rednega dela.

Recimo, da je ena ekipa dosegla v rednem delu a golov, druga b, poleg tega pa sta v izrednem delu dosegli obe skupaj c golov. Pri tem naj a pomeni tisto ekipo, ki je dosegla v rednem delu več golov kot druga; torej bo vedno $a \geq b$.

Če je pri neki stavi Janezek napovedal, da bo skupno število golov s, razlika po rednem delu pa r, bo ta napoved pravilna, če bosta veljali naslednji enačbi:

$$a + b + c = s \quad \text{in} \quad a - b = r.$$

Iz druge enačbe dobimo $a = b + r$; če to nesemo v prvo, tam dobimo $2b + c = s - r$. Ker število golov ne more biti negativno, sta b in c oba ≥ 0, zato je tudi leva stran enačbe ≥ 0; če je torej desna stran negativna, enačba ne more biti izpolnjena in stave ni mogoče dobiti. Drugače pa ni težko najti primernega para (b, c). Iz enačbe vidimo, da bo veljalo $b = (s - r - c)/2$, in ker mora biti b celo število, moramo izbrati c tako, da bo enake parnosti kot $s - r$ (potem bo namreč razlika $s - r - c$ soda). Primerna možnost je na primer $c = 0$, če je razlika $s - r$ soda, in $c = 1$, če je liha. Na koncu določimo še a po prej omenjeni formuli $a = b + r$.

Doslej smo razmišljali o tisti interpretaciji naloge, pri kateri štejejo v skupno število golov pri stavi tudi tisti iz izrednega dela igre. Če pa si nalogo razlagamo tako, da goli iz izrednega dela ne štejejo, je to enako, kot če dodamo pogoj $c = 0$. Kot smo videli v prejšnjem odstavku, zaradi te spremembe stave ne bo več mogoče dobiti, če je razlika $s - r$ liha.

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

int main()
{
  int n; cin >> n;
  while (n-- > 0)
  {
    int s, r; cin >> s >> r;
    if (r < 0 || s - r < 0 || (s - r) % 2 == 1)
```

```
      cout << "nemogoče" << endl;
    else
    {
      int b = (s − r) / 2;
      int a = b + r;
      cout << a << " " << b << endl;
    }
  }
}
```

Zgornja implementacija rešitve v C++ vključuje pogoj $c = 0$; če ga nočemo, moramo v stavku **if** zakomentirati zadnji del pogoja, „|| (s − r) % 2 == 1“.

11. Sudoku

Pri tej nalogi ni treba drugega, kot da gremo z nekaj zankami po y-ti vrstici, po x-tem stolpcu in po ustreznem kvadratu 3×3 in preverjamo, da se v njih še nikjer ne pojavlja število a. Glede kvadrata 3×3 lahko razmišljamo takole: recimo, da štejemo koordinate (vrstic oz. stolpcev) od 0 do 8; potem polje (x, y) pripada tistemu kvadratu, ki ima v svojem zgornjem levem kotu koordinati $(3\lfloor x/3 \rfloor, 3\lfloor y/3 \rfloor)$. Celoten kvadrat lahko pregledamo z dvema gnezdenima zankama od 0 do 2.

Poleg zgoraj naštetih stvari je koristno preveriti še, če je polje (x, y) prazno, saj naloga tega ne zagotavlja posebej. Oglejmo si primer rešitve v C++:

```
bool SmemoVpisati(const int T[9][9], int x, int y, int a)
{
  if (T[y][x] != 0) return false;
  for (int xx = 0; xx < 9; xx++) if (T[y][xx] == a) return false;
  for (int yy = 0; yy < 9; yy++) if (T[yy][x] == a) return false;
  for (int yy = 0; yy < 3; yy++) for (int xx = 0; xx < 3; xx++)
    if (T[3 * (y / 3) + yy][3 * (x / 3) + xx] == a) return false;
  return true;
}
```

12. Kakuro

Posamezno polje mreže lahko na primer predstavimo s preprosto strukturo z nekaj atributi, ki povedo barvo polja, števko na njem (če je belo) in predpisano vsoto za besedo desno od njega in besedo pod njim (če je črno). V primerih, ko črno polje na desni in/ali spodaj nima belega soseda, si mislimo, da je predpisana vsota tam 0. Celotno mrežo lahko zdaj predstavimo kot 2-d tabelo takšnih struktur; v spodnji rešitvi bomo na primer uporabili vektorje iz C++ove standardne knjižnice.

Naš podprogram se zdaj lahko z dvema gnezdenima zankama sprehodi po vseh poljih mreže in preverja pogoje iz besedila naloge. Glavnino pogojev lahko preverimo pri črnih poljih; pri belem polju preverimo le, da ne leži v prvi vrstici ali stolpcu. Pri črnem polju pa se v gnezdeni zanki sprehodimo po besedi desno od njega (torej gremo v desno, dokler ne naletimo na naslednje črno polje ali pridemo do roba mreže) in podobno še po besedi pod njim. Pri belih poljih, ki jih takrat obiščemo, lahko sproti preverjamo, da vsebujejo veljavne števke (od 1 do 9) in da so vse te števke različne (v spremenljivki videno si prižigamo bite za tiste števke, ki smo jih že videli, tako da lahko zelo enostavno preverimo, ali se kakšna števka pojavi že

drugič), sproti pa jih lahko tudi seštevamo in na koncu besede preverimo, če je dobljena vsota enaka predpisani (iz črnega polja pred začetkom besede).

Če ugotovimo, da je kakšna od omejitev iz besedila naloge prekršena, se lahko takoj ustavimo in sporočimo klicatelju, da je mreža neveljavna (vrnemo **false**); če pa pridemo uspešno do konca, to pomeni, da je vse v redu (in vrnemo **true**).

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

struct Polje
{
   bool crno;
   int stevka;                        // če je belo
   int vsotaSpodaj, vsotaDesno;  // če je črno
};

bool Preveri(const vector<vector<Polje>>& t)
{
   const int DX[ ] = { 1, 0 }, DY[ ] = { 0, 1 };  // Vodoravna in navpična smer.
   int w = t[0].size(), h = t.size();                  // Širina in višina mreže.
   for (int y = 0; y < h; y++) for (int x = 0; x < w; x++)
   {
      const auto &P = t[y][x];
      // Če je polje belo, preverimo le, da ne leži v prvi vrstici ali stolpcu.
      // Vsebino belih polj bomo podrobneje pregledali drugje
      // (ko se bomo iz črnega polja sprehodili po celotni besedi).
      if (! P.crno) {
         if (x == 0 || y == 0) return false;
         else continue; }
      // Preglejmo besedi desno od tega polja (smer = 0) in pod njim (smer = 1).
      for (int smer = 0; smer < 2; smer++)
      {
         int videno = 0, vsota = 0;
         for (int d = 1; ; d++)
         {
            // Izračunajmo koordinati polja d korakov od (x, y) v trenutni smeri.
            int xx = x + DX[smer] * d, yy = y + DY[smer] * d;
            // Na robu mreže se beseda konča.
            if (xx >= w || yy >= h) break;
            // Konča se tudi pri črnem polju.
            const auto &PP = t[yy][xx];
            if (PP.crno) break;
            // Preverimo, če je tu veljavna števka.
            if (PP.stevka < 1 || PP.stevka > 9) return false;
            // Preverimo, če so vse števke v besedi različne.
            if (videno & (1 << PP.stevka)) return false;
            // Zapomnimo si, da smo to števko že videli, in jo prištejmo k vsoti.
            videno |= (1 << PP.stevka); vsota += PP.stevka;
         }
         // Na koncu besede še preverimo, če je vsota enaka zahtevani.
         if (vsota != (smer == 0 ? P.vsotaDesno : P.vsotaSpodaj)) return false;
      }
   }
```

```
    return true;
}
```

13. Iskanje števila

Pogoj $a_i = i$ lahko zapišemo kot $a_i - i = 0$; vrednost na levi strani imenujmo b_i. Kaj lahko povemo o zaporedju $b_0, \ldots, b_{n-1}$? Ker je prvotno zaporedje, $a_0, \ldots, a_{n-1}$, urejeno naraščajoče in so vsa števila v njem različna in cela, je vsako število vsaj za 1 večje od prejšnjega:

$$a_i \geq a_{i-1} + 1.$$

Če v tem izrazu upoštevamo, da je $a_i = b_i + i$ in $a_{i-1} = b_{i-1} + (i-1)$, dobimo

$$b_i + i \geq b_{i-1} + (i-1) + 1,$$

torej $b_i \geq b_{i-1}$. Zaporedje b-jev je torej tudi urejeno, le da ni nujno strogo naraščajoče (kot je bilo zaporedje a-jev), pač pa je lahko več zaporednih elementov enakih (zaporedje b-jev je torej nepadajoče). Ker je urejeno, lahko v njem z bisekcijo poiščemo element z vrednostjo $b_i = 0$ (če obstaja), s tem pa bomo tudi rešili prvotni problem, saj smo videli, da je pogoj $b_i = 0$ enakovreden pogoju $a_i = i$.

Do enake rešitve lahko pridemo tudi z naslednjim razmislekom. Recimo, da pri nekem indeksu i velja $a_i < i$. Če zdaj počasi zmanjšujemo i za 1, se bo desna stran te neenakosti na vsakem koraku zmanjšala za 1, leva stran pa *vsaj* za 1; torej bo leva stran še naprej ostala manjša od desne. Če torej velja $a_i < i$, moramo z našim iskanjem (da bomo našli $a_i = i$) nadaljevati pri večjih i, ne pri manjših. Analogen razmislek nam tudi pove, da če velja $a_i > i$, moramo z iskanjem nadaljevati pri manjših i, ne pri večjih.

Naša rešitev lahko torej vzdržuje par indeksov, ℓ in d, ki povesta, da so v tabeli elementi na indeksih levo od ℓ gotovo premajhni, tisti od d naprej pa gotovo preveliki. Območje, ki ga moramo še preiskati, so torej indeksi $\ell, \ldots, d-1$. Na začetku postavimo ℓ na 0 in d na n, tako da to območje pokriva celotno tabelo a.

Na vsakem koraku nato poglejmo element na sredini opazovanega območja, torej na indeksu $m = \lfloor(\ell + d)/2\rfloor$. (1) Če pri njem velja $a_m < m$, je torej ta element premajhen in vsi levo od njega tudi, zato lahko postavimo ℓ kar na $m + 1$. (2) Če velja $a_m > m$, je ta element prevelik in vsi desno od njega tudi, zato lahko postavimo d na m. (3) Če pa je $a_m = m$, smo našli ravno to, kar smo iskali, in lahko iskanje takoj končamo. — Te tri možnosti lahko ločimo z enim samim vpogledom v tabelo, če si vrednost a_m skopiramo v neko pomožno spremenljivko.

Tako se iskalno območje v vsaki iteraciji približno razpolovi in prej ali slej se skrči na en sam element ali pa celo nobenega (torej $d - \ell \leq 1$). Če je ostal še en element (na indeksu ℓ), moramo zanj še z enim vpogledom v tabelo preveriti, če pri njem velja $a_\ell = \ell$.

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

int Ujemanje(const vector<int>& a)
{
    int L = 0, D = a.size();
    if (D <= 0) return −1; // prazna tabela
```

```
  while (D − L > 1)
  {
    // Na tem mestu velja: a[i] < i za i < L in a[i] > i za i ≥ D.
    // Oglejmo si element na polovici med L in D.
    int M = (L + D) / 2;
    int aM = a[M];
    if (aM < M) L = M + 1;   // Če je a[M] < M, moramo iskati desno od njega.
    else if (aM > M) D = M;  // Če je a[M] > M, moramo iskati levo od njega.
    else return M;           // Če je a[M] = M, smo našli, kar smo iskali.
  }
  // Tu je D = L + 1 ali D = L; vsi elementi levo od L so premajhni,
  // vsi desno od L pa preveliki. Preveriti moramo le še L samega.
  return (D > L && a[L] == L) ? L : −1;
}
```

Prepričajmo se, da ta rešitev res porabi največ 10 vpogledov za tabele dolžine največ $n = 1000$. V vsaki iteraciji glavne zanke izvedemo en vpogled v tabelo; kako pa se spremeni dolžina iskalnega območja? Recimo, da je bilo iskalno območje prej dolgo s elementov (za $s = d - \ell$). Če je s lih, recimo $s = 2k + 1$, nam območje razpade na k elementov levo od m, element m sam in nato še k elementov desno od m; v najslabšem primeru bo torej iskalno območje v naslednji iteraciji dolgo $k = \lfloor s/2 \rfloor$ elementov. Če pa je s sod, recimo $s = 2k$, nam območje razpade na k elementov levo od m, element m sam in nato še $k-1$ elementov desno od m; tudi zdaj bo torej iskalno območje v naslednji iteraciji dolgo kvečjemu $k = \lfloor s/2 \rfloor$ elementov.

Ker je na začetku iskalno območje dolgo n elementov, je po t iteracijah (in s tem po t vpogledih v tabelo) dolgo kvečjemu $\lfloor n/2^t \rfloor$ elementov. Zanka se konča, ko to pade pod 2; to je pri $n/2^t < 2$, torej $n < 2^{t+1}$, torej $\log_2 n < t + 1$, torej $t > -1 + \log_2 n$; ker mora biti t cel, je ta pogoj enakovreden $t > -1 + \lfloor \log_2 n \rfloor$; ker je desna stran zdaj cela, je najmanjši primerni t tisti, ki je le za 1 večji od nje, to pa je $t = \lfloor \log_2 n \rfloor$. Ker imamo lahko še en vpogled v tabelo (za element a_ℓ) po koncu glavne zanke, je skupno število vpogledov enako $1 + \lfloor \log_2 n \rfloor$. Ker gre n do 1000, to pa leži med 2^9 in 2^{10}, je skupno število vpogledov v tabelo največ $1 + 9 = 10$, kar ravno še ustreza zahtevam naloge.

Bisekcijo se pogosto izvede tako, da iskalno območje v vsaki iteraciji razdelimo na dva dela (ali je a_m prevelik ali ne) namesto na tri (ali je a_m premajhen, prevelik ali ravno pravšnji):

```
int Ujemanje2(const vector<int>& a)
{
  int L = 0, D = a.size();
  if (D <= 0) return −1; // prazna tabela
  while (D − L > 1)
  {
    // Na tem mestu velja: a[i] < i za i < L in a[i] > i za i ≥ D.
    // Oglejmo si element na polovici med L in D.
    int M = (L + D) / 2;
    if (a[M] > M) D = M;  // a[M] je prevelik
    else          L = M;  // a[M] ni prevelik
  }
  // Tu je D = L + 1; vsi elementi levo od L so premajhni,
  // vsi desno od L pa preveliki. Preveriti moramo le še L samega.
```

```
  return (a[L] == L) ? L : −1;
}
```

Podoben razmislek kot prej bi zdaj pokazal, da se iskalno območje v vsaki iteraciji skrajša s s na $\lceil s/2 \rceil$ elementov, zato po t iteracijah s prvotne dolžine n pade na $\lceil n/2^t \rceil$ elementov, tako da v najslabšem primeru potrebujemo $t = \lceil \log_2 n \rceil$ iteracij. Skupaj s še enim vpogledom po koncu glavne zanke imamo največ $1 + \lceil \log_2 n \rceil$ vpogledov v tabelo, kar pri $n = 1000$ lahko v naslabšem primeru pomeni 11 vpogledov, ne pa samo 10, kot zahteva besedilo naloge.

14. Torta

Koristno je, če graf najprej topološko uredimo; to pomeni, da poiščemo tak vrstni red točk, v katerem začetno krajišče vsake povezave nastopi pred končnim krajiščem iste povezave (torej: če obstaja povezava $u \to v$, mora biti u v topološkem vrstnem redu prej kot v). To lahko naredimo tako, da v vrstni red najprej postavimo tiste točke, ki nimajo nobene vhodne povezave; nato te točke v mislih pobrišemo iz grafa; v vrstni red dodamo tiste točke, ki *zdaj* nimajo nobene vhodne povezave; nato tudi njih v mislih pobrišemo iz grafa; in tako naprej. V resnici točk ni treba brisati iz grafa; dovolj le že, če za vsako točko štejemo, koliko vhodnih povezav bi ji še ostalo, če bi iz grafa res pobrisali točke, ki smo jih že postavili v naš vrstni red.

Ko imamo enkrat vse točke v topološkem vrstnem redu, jih lahko pregledujemo od konca proti začetku in popravljajmo podatke o tem, koliko katere jedi potrebujemo. To lahko hranimo v tabeli, v kateri element k_v pove, koliko jedi v potrebujemo. Na začetku postavimo vse k_v na 0, le k_t (za jed t iz naše poizvedbe) naj bo 1. Nato pa, ko pri pregledovanju topološkega vrstnega reda pridemo do jedi v, pojdimo, če je $k_v > 0$, po vseh vhodnih povezavah točke v in pri vsaki povezavi $u \to v$ povečajmo k_u za $d_{uv} \cdot k_v$ — toliko bomo namreč potrebovali jedi u za pripravo tistih k_v enot jedi v, kolikor jih potrebujemo. Ko pri tem postopku pridemo do točke s (za jed s iz naše poizvedbe), lahko končamo, saj jedi, ki so pred njo v vrstnem redu, gotovo ne potrebujejo jedi s, zato se v nadaljevanju postopka potrebe po jedi s ne bi več povečevale.

Zapišimo našo rešitev še s psevdokodo. Najprej topološko urejanje:

za vsako točko u: $deg[u] := 0$;
za vsako povezavo $u \to v$: povečaj $deg[v]$ za 1;
$L :=$ prazen seznam; $Q :=$ prazna vrsta;
za vsako točko u: **if** $deg[u] = 0$ **then** dodaj u v Q;
while Q ni prazna:
  naj bo u poljubna točka iz Q; pobriši jo iz Q in dodaj na konec L;
  za vsako povezavo $u \to v$:
    zmanjšaj $deg[v]$ za 1;
    if $deg[v] = 0$ **then** dodaj v v Q;

Ta postopek moramo izvesti le enkrat, potem pa lahko dobljeni topološki vrstni red L uporabimo za poljubno število poizvedb. V praksi lahko L in Q hranimo oba v isti tabeli, le zapomniti si moramo, pri katerem indeksu v njej se neha L in začne Q. Oglejmo si zdaj še postopek za odgovarjanje na poizvedbo (s, t):

za vsako točko u: $k[u] := 0$;
$k[t] := 1$;
for $i := n$ **downto** 1:
  $v := L[i]$; **if** $v = s$ **then break**;
  if $k[v] = 0$ **then continue**;
  za vsako povezavo $u \to v$: povečaj $k[u]$ za $k[v] \cdot d_{uv}$;
return $k[s]$;

Glavne zanke v tem postopku pravzaprav ni treba začeti na koncu seznama L, ampak bi lahko začeli tam, kjer se v njem pojavlja točka t, saj tistih, ki so za njo v topološkem vrstnem redu, za pripravo jedi t ne potrebujemo niti posredno niti neposredno. Časovna zahtevnost naše rešitve je $O(n + m)$, če ima naš graf n točk in m povezav.

15. Iskanje zlata

Z enim pregledom mreže lahko ugotovimo (npr. z iskanjem v širino), katere votline so zazidane, torej niso dosegljive iz nobenega vhoda. Če te votline v mislih pobrišemo (lahko kar zazidamo vsa bela polja v njih, spet z iskanjem v širino), ostanejo le še votline, ki so dosegljive iz vsaj enega vhoda. Oštevilčimo jih od leve proti desni. Zdaj velja tole: če sta iz nekega polja dosegljivi votlini a in b, so iz njega dosegljive tudi vse votline med njima.

Prepričajmo se, da je to res. Pa recimo, da to v splošnem ne drži; vzemimo najnižje polje $p = (x_p, y)$, pri katerem to ne drži, torej sta iz njega dosegljivi dve votlini a in b, ne pa tudi neka vmesna votlina c. Pot od p do a mora prej ali slej narediti korak navzdol iz vrstice y v $y + 1$; recimo, s tem korakom stopi v polje $q = (x_q, y + 1)$. Podobno recimo, da pot od p do b s prvim korakom navzdol stopi v polje $r = (x_r, y + 1)$.

Iz q je torej dosegljiva a, iz r pa b; gotovo pa ni iz q dosegljiva b, iz r pa ne a, saj bi tedaj bila iz q oz. r dosegljiva tudi c (ker sta q in r nižje od p, ta pa je najnižji, pri katerem se zgodi, da sta iz njega dosegljivi a in b, ne pa tudi c), s tem pa tudi iz p, kar bi bilo protislovje.

Ker votlina c ni dosegljiva iz p, q ali r, vendarle pa mora biti dosegljiva iz nekega vhoda (saj bi drugače votlino c že zazidali), mora obstajati neka pot od nekega vhoda v do c. Ta pot se mora prej ali slej tudi spustiti iz vrstice y v $y + 1$; recimo, da to stori pri $x = x_s$. Gotovo x_s ni na območju $x_q, \ldots, x_r$, saj bi se dalo potem nadaljevanje te poti izkoristiti tudi iz p in bi bila c dosegljiva iz p. Ena možnost je zdaj, da je $x_s < x_q$. Takrat je torej pot $v \rightsquigarrow c$ levo od poti $q \rightsquigarrow a$; toda na koncu mora priti desno od nje, saj c leži desno od a. Poti se morata torej nekje prekrižati, to pa pomeni, da obe obiščeta isto belo polje t; to polje je zdaj dosegljivo iz q (in s tem iz p), iz njega pa je dosegljiva votlina c, torej je c dosegljiva iz p, kar je protislovje. Podoben razmislek nas pripelje v protislovje tudi, če je x_s desno od x_r.

Tako torej vidimo, da nas predpostavka, da c ni dosegljiva iz p (a in b pa sta), v vsakem primeru pripelje v protislovje. □

Intervale votlin, dosegljivih iz posameznega belega polja, lahko računamo za vsa bela polja mreže od spodaj navzgor. Ko vidimo v neki vrstici strnjeno skupino belih polj, lahko pogledamo intervale njihovih belih sosedov eno vrstico nižje; te intervale

zlijemo (moramo si le zapomniti najbolj levo in najbolj desno votlino, dosegljivo iz kakšnega od teh spodnjih belih sosedov) in tako dobimo interval, ki je dosegljiv iz belih polj naše opazovane skupine.

Ko tako za vsak vhod vemo, kateri interval votlin je dosegljiv iz njega, moramo le še izračunati skupno število zlatnikov v njih. To lahko zelo poceni naredimo tako, da si najprej izračunamo kumulativne vsote: $v_0 = 0$ in $v_k = v_{k-1} + z_k$. Skupno število zlatnikov v votlinah od vključno a do vključno b je potem preprosto $v_b - v_{a-1}$.

16. Davki

(a) Pri tej podnalogi ni treba dosti drugega kot pazljivo slediti pravilom iz besedila naloge. Račune bomo obdelovali v zanki v takem vrstnem redu, v kakršnem so podani v vhodnem seznamu. Pri vsakem računu moramo najprej preveriti, če smo račun z isto številko videli že kdaj prej; če da, ga moramo zdaj ignorirati. Zato je koristno številke že videnih računov shranjevati v neki razpršeni tabeli (spodnja rešitev ima v ta namen spremenljivko zeVideno).

Za vsakega pošiljatelja bomo vzdrževali podatke o računih, ki jih je poslal in smo jih že videli, nismo pa jih še združili v pakete. Ker moramo pakete le šteti (ni nam treba npr. izpisovati identifikacijskih številk konkretnih računov, ki tvorijo nek paket), pravzaprav ni pomembno, kateri računi so to, ampak le, od katerih izdajateljev so. Zato bomo (za vsakega pošiljatelja) imeli majhno razpršeno tabelo (v spodnji rešitvi je to izdajatelji), v kateri kot ključ nastopa številka izdajatelja, kot pripadajoča vrednost pa število še neuporabljenih računov tega izdajatelja (pri trenutnem pošiljatelju). Poleg tega bomo za vsakega pošiljatelja hranili še število že sestavljenih paketov. Oboje skupaj hrani spodnja rešitev v strukturah tipa Posiljatelj, te pa v razpršeni tabeli pos (ključ v njej je številka pošiljatelja).

Ko opazimo nov račun, moramo torej najprej preveriti, če smo njegovega pošiljatelja že kdaj videli, in če ga nismo, pripraviti zanj novo strukturo Posiljatelj in jo dodati v razpršeno tabelo pošiljateljev. Nato lahko v njegovi strukturi izdajatelji povečamo števec paketov ustreznega izdajatelja (če tega izdajatelja še ni v njej, ga moramo vanjo dodati). Če je zdaj v strukturi 10 različnih izdajateljev, lahko iz njihovih računov sestavimo paket; povečajmo torej števec paketov pri tem pošiljatelju, število neuporabljenih računov pri vsakem od desetih izdajateljev pa zmanjšajmo za 1. Pri nekaterih izdajateljih lahko zdaj števec neuporabljenih računov pade na 0; njih pobrišimo iz strukture izdajatelji.

Ko pridemo do konca vhodnega seznama računov, se moramo le še enkrat sprehoditi po vseh pošiljateljih in pri vsakem izpisati verjetnost, da bo izžreban (število njegovih paketov delimo s številom vseh paketov in pomnožimo s 100, pa dobimo verjetnost v odstotkih).

```
#include <memory>
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <stdio.h>
using namespace std;

struct Racun { int racun, izdajatelj, posiljatelj; };

struct Posiljatelj
```

```
{
  int stPaketov = 0;
  // V naslednji razpršeni tabeli je ključ številka izdajatelja, pripadajoča vrednost
  // pa je število še neuporabljenih računov tega izdajatelja (pri tem pošiljatelju).
  unordered_map<int, int> izdajatelji;
};

void Obdelava(const vector<Racun>& v, int velikostPaketa = 10)
{
  unordered_set<int> zeVideno;
  unordered_map<int, unique_ptr<Posiljatelj>> pos;
  int skupajPaketov = 0;
  for (const auto &R : v)
  {
    // Če smo račun s to številko že videli, ga ignorirajmo.
    if (zeVideno.find(R.racun) != zeVideno.end()) continue;
    zeVideno.insert(R.racun);
    // Dodajmo tega pošiljatelja v „pos“, če ga še nismo videli.
    auto pr = pos.emplace(R.posiljatelj, nullptr);
    if (pr.second) pr.first->second.reset(new Posiljatelj);
    auto &P = *(pr.first->second);
    auto &I = P.izdajatelji;
    // Povečajmo števec računov tega izdajatelja (pri trenutnem pošiljatelju).
    if (auto pr = I.emplace(R.izdajatelj, 1); ! pr.second)
      { pr.first->second++; continue; }
    // Ali je dovolj različnih pošiljateljev za nov paket?
    if (I.size() < velikostPaketa) continue;
    // Zmanjšajmo število računov vsakega pošiljatelja za 1;
    // tiste, ki padejo na 0, pobrišimo.
    for (auto it = I.begin(); it != I.end(); )
      if (--it->second == 0) it = I.erase(it); else ++it;
    // Povečajmo števca paketov.
    P.stPaketov++; skupajPaketov++;
  }
  // Izpišimo rezultate.
  if (skupajPaketov == 0) skupajPaketov = 1; // da ne bo deljenja z 0
  for (auto it = pos.begin(); it != pos.end(); ++it)
    printf("Pošiljatelj %d bo izžreban z verjetnostjo %.2f %%.\n", it->first,
      it->second->stPaketov * 100.0 / double(skupajPaketov));
}
```

(*b*) Oglejmo si konkreten primer: recimo, da imamo štiri izdajatelje (a, b, c in d) in od vsakega po tri račune; in recimo, da davkarija tvori pakete velikosti $p = 3$. Če pošljemo račune v vrstnem redu *aaabbbcccddd*, bo davkarija sestavila tri pakete *abc*, na koncu pa ji bodo ostali le še trije računi izdajatelja d, s katerimi ni mogoče sestaviti novega paketa (ker nimamo p različnih izdajateljev). Če pa pošljemo račune v vrstnem redu *abcdabcdabcd*, bo davkarija sestavila štiri pakete (*abc*, *dab*, *cda* in *bcd*).

Ob tem primeru nam lahko pride na misel, da je neugodno, če nam pri pošiljanju zmanjka računov nekaterih izdajateljev prej kot drugih; bolje je, če imamo čim dlje v rokah še nekaj računov od čim več različnih izdajateljev. Tako dobimo naslednji požrešni postopek: v vsakem koraku poglejmo, katerih p izdajateljev ima

največ računov, in pošljimo po en račun vsakega od njih (iz tako poslanih p računov bo davkarija zanesljivo lahko sestavila en paket). Ko ostanejo računi manj kot p različnih izdajateljev, je vseeno, ali jih sploh še pošiljamo ali ne, saj davkarija iz njih tako ali tako ne bo mogla sestaviti nobenega paketa več.

Prepričajmo se, da ta postopek res pripelje do največjega možnega števila paketov. Brez izgube za splošnost se lahko omejimo na take vrstne rede, pri katerih se računi vedno pošiljajo v skupinah po p, pri čemer so vsi računi v skupini od različnih izdajateljev. Če jih pošiljamo kako drugače, se bodo pač nekaj časa nabirali pri davkariji (npr. v strukturi izdajatelji iz naše rešitve prvega dela naloge), dokler se ne bo nabralo dovolj primernih kasnejših računov za tvorbo novega paketa.

Kot je običajno pri požrešnih algoritmih, si lahko pomagamo s protislovjem. Recimo, da obstaja nek drug optimalni vrstni red, ki doseže več paketov od našega požrešnega vrstnega reda. Oglejmo si prvi trenutek, v katerem se optimalni vrstni red razlikuje od našega požrešnega vrstnega reda. (Če je možnih več optimalnih vrstnih redov, vzemimo med njimi tistega, pri katerem prva taka razlika nastopi najkasneje.) Ker se doslej vrstna reda nista razlikovala, je število še neposlanih računov posameznega izdajatelja pri obeh za zdaj še enako. Naj bo b_i število še neposlanih računov izdajatelja i, pri čemer izdajatelje oštevilčimo tako, da bo veljalo $b_1 \geq b_2 \geq \ldots \geq b_n$.

Požrešni algoritem bi torej zdaj poslal paket računov prvih p izdajateljev: $G = \{1, 2, \ldots, p\}$, optimalna rešitev pa nek drugačen paket p računov, recimo mu R. Ker sta množici G in R različni, a enako veliki, mora v R gotovo manjkati eno od števil iz G (recimo, da je to i), obenem pa mora R gotovo vsebovati neko število, ki ni iz G (recimo, da je to j). Ker vsebuje G števila od 1 do p, imamo torej $i \leq p < j$; in ker je torej $i < j$, je $b_i \geq b_j$.

Preglejmo v optimalni rešitvi vse pakete od vključno R naprej in preštejmo, v koliko paketih je prisoten izdajatelj i, ne pa tudi j (recimo, da je takih n_i); v koliko paketih je prisoten j, ne pa i (recimo n_j); in v koliko paketih sta prisotna oba (recimo n_{ij}). Gotovo velja $n_i + n_{ij} \leq b_i$ in $n_j + n_{ij} \leq b_j$. Ker R vsebuje le izdajatelja j, ne pa tudi i, je $n_j \geq 1$, zato $n_{ij} < b_j \leq b_i$. Ločimo zdaj dve možnosti. (1) Če je $n_i = 0$, nam to skupaj s $n_{ij} < b_i$ pove, da bo do konca optimalnega vrstnega reda uporabljenih le n_{ij} od b_i računov izdajatelja i, nekaj (več kot nič) pa bo ostalo neuporabljenih. Torej lahko R spremenimo tako, da v njem namesto j uporabimo i, pa bo vrstni red ostal veljaven (zdaj bo uporabljenih $1 + n_{ij} \leq b_i$ računov izdajatelja i) in prinesel enako število paketov kot prej, torej bo še vedno optimalen. (2) Če pa je $n_i > 0$, to pomeni, da nekje v nadaljevanju optimalnega vrstnega reda obstaja nek paket R', ki vsebuje izdajatelja i, ne pa tudi j. Optimalni vrstni red lahko zdaj spremenimo tako, da v R uporabimo i namesto j, v R' pa j namesto i; vrstni red s tem ostane veljaven in prinese enako število paketov kot prej, torej je še vedno optimalen.

Razmislek iz prejšnjih dveh odstavkov lahko ponavljamo, dokler se R še kaj razlikuje od G. Vidimo torej, da obstaja nek optimalni vrstni red, ki se z našim požrešnim ujema tudi še po paketu G, to pa je v protislovju z našo predpostavko, da smo prej vzeli optimalni vrstni red, ki se z našim požrešnim ujema najdlje, pa se je vendarle od njega razlikoval že pri G. Torej v resnici ne more obstajati noben vrstni red, boljši od požrešnega. $\square$

Razmislimo še o tem, kako bi doslej opisani požrešni pristop čim učinkoviteje implementirali. Naj bo N skupno število računov našega pošiljatelja, torej $N = a_1 + \ldots + a_n$. Ker so tudi posamezni a_i zato števila z območja od 0 do N, jih lahko uredimo s štetjem (*counting sort*) v $O(N)$ časa. Pripravimo si seznam S (*linked list*) parov $\langle k, s_k \rangle$, pri čemer je s_k seznam tistih izdajateljev i, pri katerih je $a_i = k$. Seznam S naj bo urejen padajoče po k.

Prvi paket lahko pripravimo takole: pregledujmo elemente S-ja od začetka naprej in štejmo izdajatelje v seznamih s_k, dokler se nam jih ne nabere p. Naj bo torej k tisto število, pri katerem velja, da je v seznamih $s_{k'}$ za $k' > k$ skupno manj kot p izdajateljev (recimo, da jih je le p'), v seznamih $s_{k'}$ za $k' \geq k$ pa jih je skupno vsaj p. V paket moramo torej dodati po en račun vsakega od izdajateljev iz seznamov $s_{k'}$ za $k' > k$ in še po en račun za vsakega od prvih $p - p'$ izdajateljev iz seznama s_k. Te slednje izdajatelje moramo iz seznama s_k preseliti v s_{k-1} (če ta še ne obstaja, ga zdaj ustvarimo in vrinemo za s_k v glavni seznam S). Pri vseh prejšnjih seznamih, $s_{k'}$ za $k' > k$, pa moramo le zmanjšati k' za 1, drugače pa seznami ostanejo taki, kot so bili. Edina izjema je seznam s_{k+1}, če je obstajal; ta bi se zdaj spremenil v s_k, toda ker s_k že imamo, moramo njegove izdajatelje premakniti v s_k, njega pa pobrisati (da ne bomo imeli dveh ločenih seznamov za isti k).

Tako smo imeli $O(p)$ dela, da smo sestavili en paket; ta postopek potem ponavljamo v zanki in tako sčasoma sestavimo celoten vrstni red v $O(N)$ časa.

Manjša podrobnost, ki smo jo zgoraj zanemarili, je, da je koristno v seznamih s_k poleg vsake številke izdajatelja hraniti tudi seznam vseh še neuporabljenih računov tega izdajatelja — to bo prišlo prav pri sestavljanju konkretnega vrstnega reda računov, ki jih bo treba poslati na davkarijo.

17. Tetris

Naloga je zelo primerna za reševanje z rekurzijo. Dogovorimo se, da bomo like na mrežo postavljali sistematično od zgoraj navzdol. Začnimo s prazno mrežo, nato pa na vsakem koraku poiščimo najvišje ležečo vrstico, ki še ni popolnoma pokrita (z doslej položenimi liki), in v njej poiščimo najbolj levo še nepokrito polje — recimo temu polju P. Zdaj na vse možne načine poskusimo postaviti v mrežo nov lik tako, da bo pokril polje P (ne da bi se pri tem prekrival z že položenimi liki ali pa štrlel iz mreže).

Trenutno stanje mreže hranimo v dvodimenzionalni tabeli; ko položimo nov lik, v tabeli označimo, da so tista polja zdaj pokrita. Nato nadaljujemo z rekurzivnim klicem, ko pa se iz njega vrnemo, lik spet pobrišemo in v tabeli njegova polja označimo kot prosta. Poleg tabele s stanjem mreže moramo med rekurzijo vzdrževati tudi število trenutno položenih likov vsakega tipa, saj naloga zahteva, da na koncu izpišemo tudi podatke o pogostosti posameznih likov.

Robna primera, pri katerih se naša rekurzija konča, sta dva. Ena možnost je, da pri iskanju naslednjega nepokritega polja pridemo do konca mreže in vidimo, da je ta zdaj popolnoma pokrita; tedaj moramo le povečati števec razporedov in števce pojavitev posameznega lika (vse to imejmo npr. v globalnih spremenljivkah).

Druga možnost pa je, da polja, ki ga poskušamo pokriti, ni mogoče pokriti z nobenim likom (ker bi se vsak lik tam prekrival z že postavljenimi liki ali pa štrlel čez rob mreže); to pomeni, da smo se pri postavljanju prejšnjih likov „zaplezali“ v

brezizhoden položaj in da mreže v tem stanju ne bo mogoče pokriti do konca; zato se lahko iz trenutnega rekurzivnega klica v tem primeru takoj vrnemo.

Mimogrede še omenimo, da če $w \times h$ ni večkratnik števila 4, se mreže gotovo ne bo dalo pokriti na noben način (saj vsak lik pokrije natanko štiri polja) in se nam z rekurzijo sploh ni treba ukvarjati.

Oglejmo si implementacijo takšne rekurzivne rešitve v jeziku C++. Obliko lika bomo predstavili s strukturo Lik, ki pove njegovo velikost (w vrstic, h stolpcev), tip (s tem mislimo na tisto, kar se ne spremeni, če lik zavrtimo za 90°; tipov likov je 7 in jih lahko označimo s črkami, na katere spominja njihova oblika: I, O, T, L, J, S in Z).

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

enum Tip { TipI, TipO, TipT, TipL, TipJ, TipS, TipZ, nTipov };
struct Lik { int w, h; Tip tip; bool a[4][4]; };
vector<Lik> liki;
```

Podrobnosti tega, kako inicializirati vektor liki, da bodo v njem opisi vseh možnih likov v vseh možnih orientacijah (skupaj jih bo 19), prepustimo bralcu za vajo, da naša rešitev tukaj ne bo predolga.

V globalnih spremenljivkah bomo med rekurzijo hranili trenutno stanje mreže, število likov vsakega tipa na njej ter število doslej najdenih razporedov (in število vseh uporabljenih likov posameznega tipa na njih):

```
int w, h, stRazporedov, likovVMrezi[nTipov], likovSkupaj[nTipov];
vector<vector<bool>> mreza;
```

Zdaj lahko zapišemo glavni del naše rešitve — rekurzivni podprogram, ki dobi parametra x in y ter predpostavi, da je mreža nad vrstico y že popolnoma pokrita, v vrstici y levo od stolpca x pa tudi. Njegova naloga je, da zapolni mrežo do konca in ustrezno poveča števce v globalnih spremenljivkah:

```
void Rekurzija(int x, int y)
{
   // Poiščimo naslednje prosto polje.
   while (y < h && mreza[y][x])
      if (++x >= w) { x = 0; y++; }
   // Mogoče smo že zapolnili celo mrežo.
   if (y >= h) {
      for (int i = 0; i < nTipov; i++) likovSkupaj[i] += likovVMrezi[i];
      stRazporedov++; return; }
   // Poskusimo vse možnosti glede postavitve naslednjega lika.
   for (const auto &L : liki)
   {
      if (y + L.h > h) continue; // Ali bi spodaj štrlel iz mreže?
      // Poiščimo prvo pokrito polje v prvi vrstici lika.
      int d = 0; while (! L.a[0][d]) d++;
      if (x - d < 0 || x - d + L.w > w) continue; // Ali bi štrlel iz mreže na levi/desni?
      // Ali se prekriva z že postavljenimi liki?
      bool ok = true;
```

```
        for (int yy = 0; yy < L.h && ok; yy++) for (int xx = 0; xx < L.w; xx++)
            if (L.a[yy][xx] && mreza[y + yy][x − d + xx]) { ok = false; break; }
        if (! ok) continue;

        // Postavimo ga v mrežo.
        likovVMrezi[L.tip]++;
        for (int yy = 0; yy < L.h; yy++) for (int xx = 0; xx < L.w; xx++)
            if (L.a[yy][xx]) mreza[y + yy][x − d + xx] = true;

        // Nadaljujmo z rekurzivnim klicem pri naslednjem polju.
        Rekurzija(x == w − 1 ? 0 : x + 1, y + (x == w − 1 ? 1 : 0));

        // Odstranimo lik spet iz mreže.
        likovVMrezi[L.tip]−−;
        for (int yy = 0; yy < L.h; yy++) for (int xx = 0; xx < L.w; xx++)
            if (L.a[yy][xx]) mreza[y + yy][x − d + xx] = false;
    }
}
```

Na koncu imejmo še glavni podprogram, ki inicializira globalne spremenljivke, požene rekurzijo in izpiše rezultate:

```
void PrestejRazporede(int W, int H)
{
    w = W; h = H; stRazporedov = 0;
    for (int i = 0; i < nTipov; i++) likovVMrezi[i] = 0, likovSkupaj[i] = 0;
    mreza.clear(); mreza.resize(h, vector<bool>(w, false));
    if ((w * h) % 4 == 0) Rekurzija(0, 0);

    auto _ = [] (auto x) { return likovSkupaj[x]; };
    printf("w = %d, h = %d: %d razporedov; pogostost likov: "
           "%d I, %d O, %d T, %d L, %d J, %d S, %d Z.\n", w, h, stRazporedov,
           _(TipI), _(TipO), _(TipT), _(TipL), _(TipJ), _(TipS), _(TipZ));
}
```

Časovna zahtevnost te rešitve je sorazmerna s številom vseh možnih razporedov (saj smo videli, da globalni števec razporedov povečujemo po 1), to pa, kot se izkaže, narašča eksponentno s površino mreže (torej z $w \cdot h$). Oglejmo si zdaj še malo učinkovitejšo rešitev.

Mislimo si katerikoli razpored likov, ki pravilno pokrije celo mrežo (torej pri katerem vsako polje pokriva natanko en lik in noben lik ne štrli ven iz mreže). Rekli bomo, da se lik *začne* v vrstici y, če pokriva kakšno polje v tej vrstici in nobenega v višje ležečih vrsticah. Naša prej omenjena rekurzivna rešitev je postavljala like od zgoraj navzgol, torej po naraščajočem vrstnem redu vrstic, v katerih se začnejo. Preden je postavila kak lik z začetkom v y, je zagotovo že pokrila vsa polja v vrstici $y - 1$ in tistih nad njo. Pri takšnem vrstnem redu polaganja likov lahko opazimo, da so v kateremkoli trenutku delno pokrite največ štiri vrstice, tiste nad njimi so popolnoma pokrite, tiste pod njimi pa popolnoma prazne.[8]

Recimo, da smo že položili nekaj likov in s tem popolnoma pokrili vrstice nad y, vrstice od y do $y + 3$ so delno pokrite, vrstice pod $y + 3$ pa popolnoma prazne.

[8] O tem se prepričamo takole: en lik lahko leži v največ štirih vrsticah hkrati; označimo z y najvišjo vrstico, ki še ni popolnoma pokrita; če bi zdaj bilo pokrito kakšno polje v vrstici $y + 4$, bi to pomenilo, da ga je moral pokriti nek lik, ki se začne v $y + 1$ ali še nižje; toda takega lika gotovo še nismo postavili, ker vrstica y še ni v celoti pokrita, torej zdaj še postavljamo like z začetkom v y in tisti z začetkom v $y + 1$ še niso prišli na vrsto.

Za nadaljevanje pokrivanja ni pomembno, *kako* (s kakšno razporeditvijo likov) smo pokrili tiste dele mreže, ki so zdaj pokriti — pomembno je le, da smo jih pokrili. Le od tega, kateri deli mreže so zdaj pokriti, je odvisno število razporeditev likov, s katerimi je mogoče pokriti preostanek mreže. Naša dosedanja rekurzivna rešitev je zapravila ogromno časa, ker je do istega stanja pokritosti mreže prišla po večkrat (z različnimi razporeditvami likov) in vsakič znova nadaljevala z rekurzijo na vse možne načine vse do pokritja cele mreže.

Razmišljati lahko torej začnemo o rešitvi z dinamičnim programiranjem, pri kateri si zastavimo podprobleme takšne oblike: „če je znano stanje delno zapolnjenih vrstic od y do $y+3$ in če je znano, da so vrstice nad y popolnoma polne, pod $y+3$ pa popolnoma prazne, na koliko načinov je mogoče razporediti like na prazna polja tako, da pokrijemo celo mrežo in kolikokrat se pri tem uporabi posamezni lik?“ Pri tem pa naslednji lik še vedno lahko polagamo tako kot pri rekurzivni rešitvi: začeti se mora v vrstici y in pokriti najbolj levo nepokrito polje v njej.

Če pogledamo za posamezni stolpec, kakšno je pri njem stanje v naših štirih delno zapolnjenih vrsticah, imamo tu štiri polja in zato načeloma $2^4 = 16$ možnosti, vendar vse v resnici niso mogoče: ker trenutno polagamo šele like z začetkom v y, je polje v vrstici $y+3$ lahko pokrito le, če smo v ta stolpec postavili navpičen lik I z začetkom pri y, tedaj pa so pokrita v tem stolpcu tudi polja v vrsticah y, $y+1$ in $y+2$. Od 16 možnosti jih torej ostane le 9, tako da moramo rešiti vsega skupaj $O(h \cdot 9^w)$ različnih podproblemov.

Zapišimo zdaj psevdokodo funkcije, ki rešuje posamezni podproblem. Le-tega opišemo s parametri y (prva vrstica, ki ni povsem pokrita) in $a = (a_1, \ldots, a_w)$ (pri tem a_x pove stanje mreže v stolpcu x in vrsticah od y do $y+3$; lahko si ga predstavljamo kot 4-bitno celo število, pri čemer so možne vrednosti, kot smo videli v prejšnjem odstavku, le 0, …, 7 in 15). Funkcija mora vrniti par (n, ℓ), pri čemer je n število razporeditev likov, s katerimi je mogoče zapolniti doslej prazne dele mreže, ℓ pa je tabela s številom uporabljenih likov posameznega tipa pri vseh teh razporeditvah skupaj (recimo $\ell = (\ell_1, \ldots, \ell_7)$, ker imamo 7 tipov likov).

funkcija $f(y, a)$:
    $n := 0$; $\ell := (0, 0, 0, 0, 0, 0, 0)$;
1   **if** imajo vsi a_x prižgan bit 0 **then**
        (* *Vrstica y je polna, premaknimo se navzdol.* *)
        return $f(y+1, \lfloor a_1/2 \rfloor, \ldots, \lfloor a_w/2 \rfloor)$;
2   (* *Preverimo, če so vhodni podatki veljavni.* *)
    for $x := 1$ **to** w **do if** $a_x > 0$:
        $b :=$ najvišji prižgani bit v a_x; **if** $y + b > h$ **then return** (n, ℓ);
3   (* *Preverimo, če je mreža že čisto polna.* *)
    if $y = h$ **then return** $(1, \ell)$;
4   (* *Poiščimo prvo prosto polje v vrstici y.* *)
    $x := 1$; **while** a_x ima prižgan bit 0 **do** $x := x + 1$;
5   za vsak tip lika T, za vsako možno orientacijo tega lika:
6   če je mogoče ta lik v tej orientaciji postaviti tako, da se začne v vrstici y
        in da je najbolj levo polje, ki ga v tej vrstici pokrije, tisto v stolpcu x,
        in če ta lik ne štrli ven iz mreže in ne pokrije kakšnega že pokritega polja:
7   naj bosta x' in x'' najbolj levi in najbolj desni stolpec,

```
        v katerih je prisoten ta lik;
      for z := x' to x'':
        a'_z := a_z; (* Shranimo staro stanje. *)
        prižgi v a_z bite, ki predstavljajo polja, ki jih je pokril novi lik;
8     (n', ℓ') := f(y, a);  (* Zapolnimo preostanek mreže. *)
9     n := n + n';          (* Prištejmo razporede, s katerimi se je dalo *)
      ℓ := ℓ + ℓ';          (* zapolniti preostanek mreže, k našemu rezultatu. *)
      ℓ_T := ℓ_T + n';      (* Vsak od teh razporedov ima tudi en izvod lika T,
                               ki smo ga pravkar postavili. *)
10    for z := x' to x'' do a_z := a'_z; (* Nazadnje dodani lik spet pobrišimo. *)
11  return (n, ℓ);
```

Kot je običajno pri dinamičnem programiranju, je koristno vpeljati pomnjenje rezultatov: ko se f prvič pokliče za nek par (y, a), naj si rezultat (n, ℓ) zapomni v neki tabeli, ob kasnejših klicih za isti (y, a) pa naj rezultat prebere od tam, da ji ga ne bo treba računati po večkrat. Lahko pa se rekurzivni funkciji čisto odpovemo in podprobleme rešujemo sistematično, po padajočem y in pri vsakem y po padajočem skupnem številu prižganih bitov v tabeli a. Tako bomo lahko prepričani, da bomo vedno že imeli izračunane rešitve tistih podproblemov, ki jih potrebujemo, saj se rešitev nekega podproblema vedno opira le na podprobleme s še bolj pokrito mrežo (in še manj prostimi polji). Ko rešujemo podprobleme za y, potrebujemo le rešitve podproblemov za y in $y + 1$, tiste od $y + 2$ naprej pa lahko že pozabimo (saj jih ne bomo več potrebovali) in tako prihranimo nekaj pomnilnika.

Če mrežo zasukamo za 90 stopinj, se število možnih razporeditev likov nanjo nič ne spremeni; z drugimi besedami, rezultat za mrežo $w \times h$ je enak kot za $h \times w$. Ker smo videli, da časovna zahtevnost naše narašča eksponentno z w in linearno s h, je koristno (če mreža ni kvadratna) za w vzeti krajšo od obeh stranic, za h pa daljšo. Spodnja tabela kaže število možnih razporedov likov za mreže velikosti do 8×8.[9]

Krajša stranica	Daljša stranica 1	2	3	4	5	6	7	8
1	0	0	0	1	0	0	0	1
2		1	0	4	0	9	0	25
3			0	23	0	0	0	997
4				117	454	2 003	9 157	40 899
5					0	0	0	800 290
6						178 939	0	22 483 347
7							0	657 253 434
8								19 077 209 438

Glede pogostosti pojavljanja posameznih likov lahko najprej opazimo, da bo skupno število L-jev in J-jev vedno enako, saj lahko vsako razporeditev prezrcalimo čez navpično os, pa dobimo razporeditev z enako širino in višino, le vsi L-ji so se spremenili v J-je in obratno. Iz enakega razloga je tudi skupno število S-jev in Z-jev vedno enako. Tipični rezultati pri naših poskusih z majhnimi mrežami (do 8×8) so bili naslednji: najpogostejši so bili L-ji in J-ji (po približno 20–25 % vsakih), nato

[9] Za več rezultatov na temo pokrivanja mreže z liki iz igre Tetris gl. zaporedje A230031 v *The Online Encyclopedia of Integer Sequences* in tam navedeno literaturo. Iz rezultatov na OEIS je videti, da je število različnih razporedov za mrežo $w \times h$ približno $\phi^{(w-1)(h-1)}$, pri čemer je ϕ razmerje iz zlatega reza ($\phi \approx 1{,}618$).

I-ji (približno 20 %), nato O-ji in T-ji (po približno 10–15 % vsakih), najredkejši pa so S-ji in Z-ji (po približno 5–7 % vsakih). Pri zelo razpotegnjenih mrežah je seveda lahko tudi drugače; na primer, mrežo oblike $1 \times h$ lahko zapolnimo le z I-ji.

18. RGB-šahovnica

(a) Če obstaja več omejitev za isto barvo in velikost kvadrata, jih lahko obravnavamo kot eno samo omejitev, saj lahko vsem ustrežemo z enim in istim kvadratom. Poglejmo zdaj pri vsaki barvi največjo omejitev (tisto z največjim k_i). Če se pri več kot eni barvi pojavlja omejitev s $k_i = n$, je problem nerešljiv, saj bi to pomenilo, da mora biti več kot polovica celotne mreže ene barve in hkrati več kot polovica mreže neke druge barve.

Če se pri nobeni barvi ne pojavlja omejitev s $k_i = n$, ampak se vse omejitve nanašajo na manjše kvadrate, lahko vse omejitve izpolnimo tako, da enega od štirih kvadratov reda $k - 1$ pobarvamo rdeče, enega zeleno in enega modro; za četrtega je potem vseeno, kako ga pobarvamo.

Ostane še možnost, da se omejitev s $k_i = n$ pojavlja pri natanko eni barvi; recimo brez izgube za splošnost, da pri rdeči. Pobarvajmo zdaj dva kvadrata reda $k - 1$ v celoti rdeče; pri tretjem kvadratu reda $k - 1$ pobarvajmo eno četrtino (kvadrat reda $k - 2$) v celoti rdeče, preostanek v celoti zeleno; pri zadnjem kvadratu reda $k - 1$ pa pobarvajmo eno četrtino rdeče, preostanek pa modro. Tako imamo torej en kvadrat reda $k - 1$, v katerem prevladuje zelena, kar ustreže morebitni zeleni omejitvi s $k_i = n - 1$ (če obstaja); v njem so tudi kvadrati reda $k - 2$, ki so v celoti zeleni, kar ustreže tudi vsem zelenim omejitvam s $k_i < n - 1$. Podobno je tudi z modrimi omejitvami. Glede rdečih omejitev pa mreža kot celota ustreže rdeči omejitvi s $k_i = n$ (saj je 5/8 mreže rdeče), katerikoli od popolnoma rdečih kvadratov reda $k - 1$ pa ustreže tudi vsem ostalim rdečim omejitvam (tistim s $k_i < n$).

(b) Na začetku naj bo cela mreža nepobarvana, torej bela. Omejitve obdelujmo od manjših proti večjim, torej po naraščajočem vrstnem redu k_i. Pri vsaki omejitvi pobarvajmo v njenem kvadratu najmanjše število polj, kolikor jih je treba, da bo omejitvi zadoščeno (pri tem seveda upoštevajmo, da so nekatera polja v njem mogoče že pobarvana zaradi prej obdelanih omejitev; lahko se tudi izkaže, da zaradi teh že pobarvanih polj trenutni omejitvi ni mogoče ustreči ni je problem nerešljiv). Tako si bomo najmanj omejili možnosti v nadaljevanju postopka, ko se bomo mogoče ukvarjali s kakšno večjo omejitvijo, ki tudi pokriva trenutni kvadrat.

Pri tem ni pomembno, *katera* izmed nepobarvanih polj v trenutnem kvadratu pobarvamo, pač pa le, *koliko* jih pobarvamo; kajti če se bodo kasnejše omejitve kaj sklicevale na območje našega trenutnega kvadrata, je to mogoče le tako, da se bodo nanašale na nek večji kvadrat, ki našega v celoti vsebuje, takrat pa je za potrebe vprašanja, katera barva prevladuje v tistem večjem kvadratu, čisto vseeno, kje točno znotraj našega trenutnega manjšega kvadrata so polja posamezne barve (saj so vsa ta polja v vsakem primeru tudi znotraj tistega večjega kvadrata).

(c) Recimo, da iz kvadrata, o katerem govori i-ta omejitev, v mislih izrežemo vse tiste manjše kvadrate ostalih omejitev, ki v celoti ležijo znotraj njega; dobljenemu liku (kvadratu z luknjami) recimo Q_i. V tem, kar ostane, označimo število kvadratkov posamezne barve z r_i, g_i, b_i. Število kvadratkov določene barve, recimo rdečih,

znotraj kvadrata i lahko zdaj izrazimo takole:

$$r'_i = r_i + \sum_j [\![\text{kvadrat } j \text{ v celoti leži znotraj kvadrata } i]\!]\, r_j.$$

(Pri tem izraz $[\![\ldots]\!]$ pomeni vrednost 1, če je pogoj v oklepajih izpolnjen, sicer pa 0.) Podobno je tudi za zelene in modre. Recimo zdaj, da omejitev i zahteva, da mora v kvadratu i prevladovati rdeča barva. Tako dobimo neenačbi $r'_i > g'_i$ in $r'_i > b'_i$. Analogno bi razmišljali v primerih, ko omejitev zahteva, da prevladuje kakšna druga barva. Poleg tega imamo v vsakem primeru še enačbo $r_i + g_i + b_i = \mathit{ploščina}(Q_i)$ — te ploščine seveda lahko izračunamo vnaprej.

Tako smo dobili nek nabor linearnih enačb in neenačb (z neznankami r_i, g_i, b_i za $i = 1, \ldots, n$) in če poiščemo poljubno rešitev tega sistema, lahko potem ustrezno pobarvamo naše kvadrate. Tu imamo opravka s primerom celoštevilskega linearnega programiranja, ki je sicer v splošnem NP-težak problem, vendar se dá pogosto sprejemljivo hitro rešiti dovolj velike primere problema, da je zanimiv tudi v praksi.

(d) Vzemimo dovolj velik n (kako velik, bomo videli na koncu) in vpeljimo pet omejitev take oblike kot pri podnalogi (b): za celotno mrežo (kvadrat reda n) zahtevajmo, naj bo v njej več kot polovica polj rdečih; za njene štiri podkvadrate reda $n - 1$ pa pri dveh zahtevajmo, naj bo v njiju več kot polovica polj zelenih, pri dveh pa, da naj bo v njiju več kot polovica polj modrih. Pri tem naboru omejitev je problem nerešljiv, ker je v vsaki četrtini mreže manj kot polovica polj rdečih, zato je tudi v mreži kot celoti manj kot polovica polj rdečih in je nemogoče ustreči prvi omejitvi.

Če pa omejitve spremenimo v tip (c), postane naloga rešljiva. V vsaki četrtini mreže pobarvajmo eno manj kot polovico vseh polj rdeče, preostala (torej eno več kot polovico vseh polj) pa bodisi zeleno bodisi modro, odvisno od tega, katera barva mora biti v tisti četrtini najpogostejša. S tem smo ustregli omejitvam za posamezne četrtine mreže, za celotno mrežo pa vidimo, da v njej rdeča pokriva skoraj polovico polj (štiri manj), zelena in modra pa le dobro četrtino, torej je tudi prvi omejitvi (da mora biti najpogostejša barva v celotni mreži rdeča) ustreženo.

Natančneje povedano, rdečih polj imamo v vsaki četrtini $\frac{1}{2}(2^{n-1} \cdot 2^{n-1}) - 1$, kar je skupaj $2 \cdot 2^{2n-2} - 4$. Zelenih polj pa je v dveh četrtinah po $\frac{1}{2}(2^{n-1} \cdot 2^{n-1}) + 1$, v ostalih dveh pa nič, kar znese skupaj $2^{2n-2} + 2$. Da bo rdečih res več kot zelenih, mora veljati $2 \cdot 2^{2n-2} - 4 > 2^{2n-2} + 2$, torej $2^{2n-2} > 6$, torej $n \geq 3$.

19. Poskočno besedilo

Podatke o črkah preberimo v tabelo ali vektor ter pri tem pretvorimo njihove y-koordinate v številke vrstic — dovolj je že, če jih delimo z v (tako dobimo številke vrstic od 0 naprej namesto od 1 naprej, kar pa je za naš namen tudi čisto dobro). Potem črke uredimo naraščajoče po številki vrstice, tiste v isti vrstici pa naraščajoče po x-koordinati. V tako dobljenem vrstnem redu jih lahko zdaj izpišemo.

Naloga ne pove, ali naj izpisujemo tudi morebitne prazne vrstice ali ne. Če hočemo izpisati tudi prazne vrstice, gremo lahko v zanki po vrsticah in se pri vsaki vrstici premikamo z gnezdeno zanko naprej po urejenem seznamu črk in jih izpisujemo, dokler ne pridemo do prve take črke, ki ni v tej vrstici. Tako dela naša spodnja rešitev:

```
#include <vector>
#include <algorithm>
#include <stdio.h>
using namespace std;

int main()
{
   int n, v; scanf("%d %d\n", &n, &v);
   // Preberimo podatke o črkah. Y-koordinate bomo sproti
   // pretvarjali v številke vrstic.
   struct Crka { char c; int x, y; };
   vector<Crka> crke {n};
   for (int i = 0; i < n; i++) {
      auto &C = crke[i]; scanf("%c %d %d\n", &C.c, &C.x, &C.y); C.y /= v; }
   // Uredimo jih po vrsticah in v vsaki vrstici od leve proti desni.
   sort(crke.begin(), crke.end(), [] (const Crka& a, const Crka& b) {
      return a.y < b.y || a.y == b.y && a.x < b.x; });
   // Izpišimo jih.
   for (int vrstica = 0, i = 0; i < n; vrstica++) {
      while (i < n && crke[i].y == vrstica) fputc(crke[i++].c, stdout);
      fputc('\n', stdout); }
   return 0;
}
```

Če praznih vrstic nočemo, gremo lahko preprosto v zanki po črkah in jih izpisujemo, po izpisu črke pa preverimo, ali moramo izpisati tudi znak za konec vrstice. Tega izpišemo, če je trenutna črka zadnja ali pa če je naslednja črka v drugi vrstici kot trenutna:

```
   for (int i = 0; i < n; ) {
      fputc(crke[i++].c, stdout);
      if (i == n || crke[i].y > crke[i − 1].y) fputc('\n', stdout); }
```

Naša rešitev ima še to slabost, da ne izpisuje presledkov med besedami. Iz vhodnih podatkov, ki jih dobimo, je težko reči, ali je med dvema zaporednima črkama v isti vrstici presledek ali ne, ker imamo le x-koordinato levega roba vsake črke, ne vemo pa, kako široka je posamezna črka. Če bi imeli pri vsaki črki tudi x-koordinato desnega roba, bi lahko po izpisu vsake črke takole preverili, ali je med njo in naslednjo nekaj podobnega presledku:

```
    { fputc(crke[i++].c, stdout);
      if (i < n && crke[i].y == crke[i − 1].y
               && crke[i].xLevo − crke[i − 1].xDesno >= v / 5) fputc(' ', stdout); }
```

20. Brisanje nezaklenjenih celic

Za lažjo različico naloge ni treba dosti drugega kot pazljivo slediti navodilom. Označimo zgornji levi kot pravokotnika z (x_1, y_1). Z dvema zankama se z x_1 in y_1 premikajmo naprej po tabeli; če je trenutna celica še nepobrisana, moramo v njej začeti nov pravokotnik. S še eno zanko pojdimo naprej po vrstici y_1 do naslednje zaklenjene celice (ali pa do konca vrstice); tako določimo desni rob pravokotnika, recimo mu x_2. Nato lahko spodnji rob pravokotnika, recimo mu y_2, počasi premikamo navzdol. Pred vsakim premikom preverimo, če bi se v naslednji vrstici zadeli ob kakšno

zaklenjeno celico; če da, se ustavimo in pravokotnik izpišemo, če ne, pa označimo te celice kot pobrisane in s premikanjem spodnjega roba nadaljujmo.

Oglejmo si implementacijo takšne rešitve v jeziku C++. Stanje celice predstavimo z naštevnim tipom StanjeCelice, ki ima tri možne vrednosti: zaklenjena; nezaklenjena in že pobrisana; ter nezaklenjena in še ne pobrisana.

```
#include <vector>
#include <stdio.h>
using namespace std;

enum StanjeCelice { Zaklenjena, Pobrisana, Polna };

void Pobrisi(vector<vector<StanjeCelice>>& t)
{
  int h = t.size(), w = t[0].size(); // Velikost mreže.
  for (int y1 = 0; y1 < h; y1++) for (int x1 = 0; x1 < w; )
  {
    // Poiščimo naslednjo polno celico.
    if (t[y1][x1] != Polna) { x1++; continue; }
    // Pojdimo v desno do naslednje zaklenjene.
    int x2 = x1 + 1; while (x2 < w && t[y1][x2] != Zaklenjena) x2++;
    // Pravokotnik bo torej pokrival stolpce od x1 do vključno x2 − 1.
    // Poglejmo, kako daleč dol ga lahko razširimo.
    int y2 = y1 + 1;
    for ( ; y2 < h; y2++)
    {
      // Ali so v vrstici y2 vse celice od x1 do x2 − 1 odklenjene?
      bool ok = true; for (int x = x1; x < x2; x++)
        if (t[y2][x] == Zaklenjena) { ok = false; break; }
      if (! ok) break;
      // Če da, jih lahko zdaj pobrišemo.
      for (int x = x1; x < x2; x++) t[y2][x] = Pobrisana;
    }
    // Izpišimo pravkar dobljeni pravokotnik.
    printf("%d..%d x %d..%d\n", x1, x2 − 1, y1, y2 − 1);
    x1 = x2;
  }
}
```

Kakšna je časovna zahtevnost tega postopka? Posamezno celico (x, y) lahko sicer pobrišemo po večkrat, vendar imajo pravokotniki, pri katerih jo pobrišemo, vsi različne y_1. Če ima namreč več pravokotnikov isto y_1 (torej če se začnejo na isti višini), so njihovi intervali $x_1, \ldots, x_2 - 1$ disjunktni in lahko le eden od njih pokriva stolpec x, v katerem leži naša opazovana celica. Našo celico torej pobriše največ en pravokotnik za vsako možno y_1, torej največ h pravokotnikov. Vseh celic pa je $w \cdot h$, tako da je časovna zahtevnost našega postopka $O(w \cdot h^2)$. Konkreten primer, pri katerem se naš postopek res izvaja toliko časa, je kvadratna tabela, v kateri so zaklenjene celice na diagonali od spodnjega levega do zgornjega desnega kota.

Našo rešitev lahko izboljšamo, če za vsako vrstico zgradimo binarno drevo, v katerem listi predstavljajo posamezne celice, notranja vozlišča pa skupine po 2, 4, 8, 16 itd. zaporednih celic. Višina tega drevesa je zato približno $\log_2 w$ nivojev. Vsako vozlišče naj hrani 1 bit, ki, če je prižgan, pomeni, da so vse celice na območju, ki

ga pokriva to vozlišče, pobrisane. Ko je treba označiti vse celice od x_1 do $x_2 - 1$ za pobrisane, tega ne označimo nujno v listih drevesa, ampak uporabimo čim višje ležeča vozlišča, ki predstavljajo daljše skupine celic znotraj intervala $x_1, \ldots, x_2 - 1$. Tako moramo na vsakem nivoju drevesa označiti največ dve vozlišči, torej za brisanje poljubne strnjene skupine celic v eni vrstici porabimo le $O(\log w)$ časa. Podobno lahko v $O(\log w)$ časa tudi za poljubno celico ali strnjeno skupino celic v isti vrstici preverimo, ali je v celoti pobrisana ali ne.

Kakšna je časovna zahtevnost tega postopka? Za vsako celico največ enkrat razmišljamo, ali bi jo uporabili kot zgornji levi kot (x_1, y_1) naslednjega pravokotnika; takrat moramo preveriti, če še ni pobrisana; to vzame $O(\log w)$ časa, skupaj torej $O(wh \log w)$. Pri vsakem pravokotniku višine v porabimo potem še $O(v \log w)$ časa za premikanje spodnjega roba navzdol. To si lahko predstavljamo tako, kot da vsaka celica na levem robu pravokotnika nosi ceno $O(\log w)$. Hitro se lahko prepričamo, da leži posamezna celica (x, y) na levem robu samo enega pravokotnika; kajti recimo, da bi ležala na levem robu dveh; označimo njuna zgornja leva vogala z (x, y_1) in (x, y'_1) za $y_1 < y'_1 \leq y$. Ker se prvi od njiju začne višje, smo očitno najprej pobrisali njega; ker smo iz njega (po predpostavki) dosegli (x, y), smo morali torej doseči tudi (x, y'_1); ker je bila ta s tem že pobrisana, pa je v nadaljevanju gotovo nismo uporabili kot zgornji levi kot kakšnega kasnejšega pravokotnika; tako smo prišli v protislovje, torej se res ne more zgoditi, da bi ležala neka celica na levem robu več pravokotnikov. Ker leži torej vsaka celica na levem robu največ enega pravokotnika, je skupna cena za premikanje spodnjega roba vseh pravokotnikov (in za označevanje njihovih celic kot pobrisanih) $O(wh \log w)$. Tako torej naša izboljšana rešitev namesto $O(wh^2)$ porabi le $O(wh \log h)$ časa. Poraba pomnilnika je še vedno $O(wh)$.

Razmislimo zdaj še o težji različici naloge, kjer je tabela velika, zaklenjene celice pa so opisane s seznamom n pravokotnikov. Pazimo na to, da imamo tu zdaj opravka z dvema vrstama pravokotnikov; eno so zaklenjeni pravokotniki, ki jih dobimo kot vhodne podatke, drugo pa so pravokotniki za brisanje, ki jih mora izpisati naš postopek.

Za začetek pokažimo, da pridejo v poštev za robove pravokotnikov za brisanje le tiste x- in y-koordinate, na katerih leži rob kakšnega zaklenjenega pravokotnika ali pa zunanji rob cele razpredelnice.

Glede desnih in spodnjih robov se to vidi takoj: ko določamo desni oz. spodnji rob pravokotnika za brisanje, se ustavimo šele na zunanjem robu ali pa ko dosežemo neko zaklenjeno celico, torej na levem oz. zgornjem robu nekega zaklenjenega pravokotnika.

Glede levih robov pravokotnikov za brisanje razmišljajmo takole. Recimo, da zgornji levi vogal našega pravokotnika tvori celica (x, y). Če je $x = 0$, leži levi rob našega pravokotnika na zunanjem robu mreže; če ima (x, y) na levi zaklenjeno sosedo, leži naš levi rob na desnem robu nekega zaklenjenega pravokotnika. Ostane še možnost, da ima (x, y) na levi odklenjeno sosedo. V trenutku, ko začenjamo nov pravokotnik z zgornjim levim kotom (x, y), je bila ta leva soseda gotovo že pobrisana. Tisti pravokotnik, ki jo je pobrisal, se je torej raztezal do vključno stolpca $y - 1$, ne pa do stolpca y, saj bi sicer pobrisal tudi našo trenutno celico (x, y); na levem robu naše celice (in s tem našega novega pobrisanega pravokotnika) torej leži desni rob nekega prejšnjega pobrisanega pravokotnika, za te pa smo že v prejšnjem odstavku videli,

da ležijo le na tistih x-koordinatah, kjer so tudi robovi zaklenjenih pravokotnikov (ali pa zunanji robovi tabele).

Ostanejo še zgornji robovi pravokotnikov za brisanje, pri katerih lahko razmišljamo analogno kot v prejšnjem odstavku za leve robove. □

Našo tabelo torej lahko ne glede na njene prave dimenzije (w in h) razrežemo pri vseh x- in y-koordinatah, kjer leži kakšen rob kakšnega zaklenjenega pravokotnika. Ker je teh pravokotnikov n, nam tabela razpade na največ $(2n+1)\times(2n+1)$ pravokotnih območij, ki jih lahko obravnavamo kot celice neke nove tabele in na njej poženemo algoritem iz prvega dela naloge; le pri izpisu rezultatov moramo še paziti, da koordinate iz nove tabele preslikamo v tiste iz prvotne. Tako smo rešili nalogo v $O(n^2 \log n)$ časa ne glede na w in h.

21. Evklidov algoritem

Nalogo je lažje reševati v drugi različici (kjer dobimo zahtevano število iteracij k in iščemo čim manjša a in b, pri katerih Evklidov algoritem izvede natanko k iteracij); ko bomo rešili to, bomo zlahka prišli tudi do rešitve prve različice (kjer dobimo zgornjo mejo za a in b in skušamo v okviru te meje doseči čim več iteracij zanke).

Označimo z $g(a,b)$ število iteracij, ki jih izvede Evklidov algoritem iz besedila naše naloge, če mu kot vhodna parametra podamo števili a in b. Iz ustavitvenega pogoja v prvi vrstici algoritma vidimo, da je $g(a,0)=0$; za $b>0$ pa nam druga vrstica algoritma (telo zanke) pove, da je $g(a,b)=g(b,a \bmod b)+1$.

Ker nas zanimajo le primeri, ko je $a \geq b$, lahko a zapišemo kot $a = q\cdot b + r$, pri čemer je q celi del količnika pri deljenju a z b, število r pa je ostanek po tem deljenju. Če to vstavimo v zvezo $g(a,b)=g(b,a \bmod b)$, dobimo $g(q\cdot b+r,b)=g(b,r)+1$.

Če torej hočemo doseči, da bo $g(a,b)=k$ za nek predpisani k, moramo poiskati najprej nek r, za katerega bo $g(b,r)=k-1$, potem pa lahko za a vzamemo poljubno število oblike $a=q\cdot b+r$. Ker hočemo čim manjši a, bomo vzeli čim manjša b in r (ob omejitvi, da mora biti $g(b,r)=k-1$) in nato $q=0$, tako da bo $a=b+r$ (pri kateremkoli večjem q bi bil a večji od tega).

Pri $k=1$ sta primerna kar $a=b=1$; z njima dosežemo eno iteracijo, manjših a in b pa ne smemo uporabiti, saj naloga zahteva $1 \leq b \leq a$. Od tam naprej nadaljujmo z razmislekom iz prejšnjega odstavka; pri $k=2$ dobimo $a=2$, $b=1$; pri $k=3$ dobimo $a=3$, $b=2$; pri $k=4$ dobimo $a=5$, $b=3$; pri $k=5$ dobimo $a=8$, $b=5$ in tako naprej. Vidimo torej, da rešitev vedno tvorita dve zaporedni Fibonaccijevi števili, $a=f_{k+1}$ in $b=f_k$. (Spomnimo se: Fibonaccijeva števila so definirana po formulah $f_0=0$, $f_1=1$ in od tam naprej $f_k=f_{k-1}+f_{k-2}$.)

Z indukcijo po k se prepričajmo, da je $g(f_{k+1},f_k)=k$ in da, če je $g(a,b)=k$ (in $1\leq b\leq a$), potem gotovo velja $a\geq f_{k+1}$ in $b \geq f_k$.

Pri $k=1$ pogoj $g(f_{k+1},f_k)=k$ postane $g(1,1)=1$, kar je res; in če je $g(a,b)=1$ in $1\leq b\leq a$, sta pogoja $a\geq f_{k+1}=1$ in $b\geq f_k=1$ trivialno izpolnjena.

Recimo zdaj, da trditev drži pri $k-1$, in poglejmo, kaj se zgodi pri k. Za $a=f_{k+1}$ in $b=f_k$ imamo $g(f_{k+1},f_k)=1+g(f_k,f_{k+1} \bmod f_k)=1+g(f_k,f_{k+1}-f_k)=1+g(f_k,f_{k-1})$, kar je po induktivni predpostavki enako $1+(k-1)=k$, kar smo tudi želeli dokazati. Vzemimo zdaj poljuben par (a,b), pri katerem je $1\leq b\leq a$ in $g(a,b)=k$. Pišimo spet $a=q\cdot b+r$; po prvi iteraciji algoritma nam iz para (a,b) nastane par (b,r) in odtlej se mora izvesti še $k-1$ iteracij, da bo skupno

število iteracij ravno k. Ker je torej $g(b,r) = k-1$, lahko za par (b,r) po naši induktivni predpostavki zaključimo, da velja $b \geq f_k$ in $r \geq f_{k-1}$. Za a torej velja $a = q \cdot b + r \geq b + r \geq f_k + f_{k-1} = f_{k+1}$, prav to pa smo tudi želeli dokazati. □

Tako torej vidimo, da je najmanjši par (a,b), pri katerem Evklidov algoritem izvede k iteracij, res ravno (f_{k+1}, f_k). Razmislimo zdaj še o prvi različici naloge, pri kateri je podana zgornja meja M, ki je a ne sme preseči. Če hočemo v okviru te omejitve doseči čim več iteracij, moramo torej poiskati največje Fibonaccijevo število, ki je še $\leq M$; to število vzemimo za a, prejšnje Fibonaccijevo število pa za b. To je največje možno število iteracij pri pogoju $a \leq M$, kajti eno iteracijo več bi dosegli šele, če bi bil a večji ali enak naslednjemu Fibonaccijevemu številu, to pa je že večje od M.

22. Prepisovanje

Na posameznem indeksu j se v nizu t_n lahko znajde istoležna črka iz kateregakoli od nizov $s_1, \ldots, s_n$. (Natančneje povedano, $t_n[j] = s_i[j]$ v primeru, če se pri indeksu j učenec i odločil, da ne bo prepisoval od predhodnika, učenci $i+1, \ldots, n$ pa, da bodo). Marsikaterega od znanih algoritmov za iskanje podniza v nizu lahko prilagodimo za naš problem, če pogoje oblike „$t_n[j] = c$“ zamenjamo z „vsaj eden od $s_1[j], \ldots, s_n[j]$ je enak c“. Ta pogoj lahko za prvo silo implementiramo z zanko, ki gre po vseh vhodnih nizih od s_1 do s_n; lahko pa si tudi na začetku pripravimo množico parov $(j, s_i[j])$ za $i = 1, \ldots, n$ in $j = 1, \ldots, d$; če ima pri nekem j več nizov s_i isti znak, shranimo od teh parov le enega; pare imejmo v razpršeni tabeli, pa bomo lahko poceni preverjali, ali se more na indeksu j niza t_n znajti nek znak c.

Najpreprostejša možnost je verjetno algoritem, ki gre v zunanji zanki po vseh možnih položajih p-ja znotraj t_n, pri vsakem od njih pa v notranji zanki primerja znake p-ja in t_n-ja, da vidi, če se vsi ujemajo:

```
for j := 1 to d − |p| + 1:
  (* Preverimo, ali se lahko pojavitev p-ja začne v t_n na indeksu j. *)
  k := 1;
  while k ≤ |p|:
    (* Preverimo, ali se lahko p[k] pojavi v t_n[j + k − 1]. *)
    i := 1;
    while i ≤ n do if p[k] = t_i[j + k − 1] then break else i := i + 1;
    (* Če se ne more, imamo neujemanje in se torej p ne more pojaviti
       v t_n z začetkom pri j. *)
    if i > n then break
    (* Sicer pa nadaljujmo s preverjanjem ujemanja pri naslednjem
       znaku p-ja. *)
    else k := k + 1;
  (* Ali smo prišli do konca p-ja, ne da bi opazili kakšno neujemanje? *)
  if k ≤ |p| then p se lahko pojavi v t_n na indeksih od j do j + |p| − 1;
```

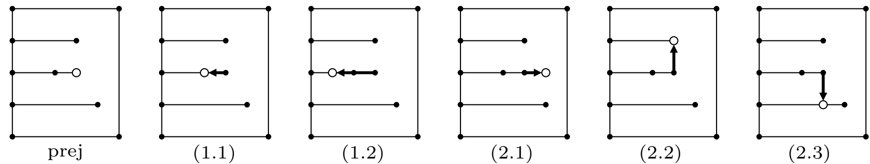
Podobno bi lahko za naš problem prilagodili tudi Knuth-Morris-Prattov in Boyer-Moorov algoritem, ki sta učinkovitejša od zgornje naivne rešitve. Primer algoritma, ki je neugoden za naš problem, pa je Rabin-Karpov, kjer zdaj za podnize dolžine $|p|$ v nizu t_n ni le po ena možnost, ampak eksponentno veliko (ker si vsak znak niza

t_n lahko izberemo na n načinov) in bi morali torej vzdrževati eksponentno veliko razprševalnih kod (ter preverjati, če je kakšna od njih enaka razprševalni kodi našega iskanega niza p).

23. Žaga

Razmislimo o nalogi najprej za splošnejši primer, torej v različici (b). Reze, ki jih žaga naredi v deski, si lahko predstavljamo kot graf; povezave v grafu predstavljajo daljice, po katerih se je žaga premikala, točke v grafu pa predstavljajo krajišča teh daljic. Na začetku imejmo graf s štirimi točkami, ki predstavljajo oglišča deske, in štirimi povezavami, ki predstavljajo robove deske; kasneje pa bomo med premikanjem vanj postopoma dodajali točke in povezave.

Glede na to, kako se pri posameznem premiku spremeni graf, lahko ločimo nekaj vrst premikov (glej sliko).



Leva slika kaže stanje grafa pred premikom, ostale pa po premiku za različne tipe premikov. Pike in krožci predstavljajo točke, pri čemer je krožec tista točka, v kateri se trenutno nahajamo. Debela puščica označuje zadnji premik.

(1) Ena možnost je, da premik v celoti poteka po že prerezanih daljicah; (1.1) če je tudi konec premika v krajišču ene od že obstoječih daljic, se moramo le premakniti v tisto točko grafa; (1.2) če pa je konec trenutnega premika nekje med dvema obstoječima točkama, moramo tam dodati v graf novo točko in povezavo med njima razdeliti na dve povezavi.

(2) Druga možnost pa je, da premik prej ali slej — mogoče že na samem začetku — zareže v še neprerezan del deske. (2.1) Če do konca poteka po nerazrezanem območju, moramo le dodati novo točko na koncu premika in povezavo do nje. (2.2) Če naš premik nekje — lahko šele na koncu, lahko pa že prej — doseže neko točko, ki je bila v grafu prisotna že od prej, moramo le dodati povezavo med dvema že obstoječima točkama. Če se tu premik še ne konča, bi morali graf načeloma dopolnjevati še naprej, vendar bomo zelo kmalu videli, da tega v resnici ni treba početi. (2.3) Ostane še možnost, da naš premik nekje doseže eno od že obstoječih daljic, vendar na mestu, kjer doslej v grafu še ni bilo točke. Tam moramo zdaj dodati točko, razbiti tisto daljico na dve, novo točko pa povezati s tisto, iz katere je naš trenutni premik zarezal v dotlej nedotaknjen del deske. Podobno kot pri 2.2 ni nujno, da se premik tu že konča, vendar se nam tudi tu z morebitnim preostankom premika ne bo treba ukvarjati.

Za ravninske grafe, kot je naš, velja znana Eulerjeva formula: $v - e + f = 2$, pri čemer je v število točk, e število povezav, f pa število območij, v katera naš graf razdeli ravnino. Iz te formule lahko izrazimo število območij kot $f = e - v + 2$. Na začetku smo imeli $v = 4$ točke (vogale deske), $e = 4$ povezave (robove deske)

in $f = 2$ območji (desko in zunanjost). Naloga od nas pravzaprav zahteva, da ugotovimo, kdaj se f poveča na 3 (takrat deska razpade na dva dela).

Pri premikih oblike 1.1 se v in e ne spremenita (ne dodajamo niti točk niti povezav), zato se tudi f ne spremeni; pri premikih oblike 1.2 in 2.1 se v in e povečata za 1 (dodamo eno točko in eno povezavo), zato se f tudi takrat ne spremeni. Pri premiku oblike 2.2 smo dodali novo povezavo, točke pa ne, torej se e poveča za 1, v pa ostane nespremenjen; iz $f = e - v + 2$ vidimo, da se takrat f poveča za 1, torej deska razpade. Če se to zgodi še pred koncem tega premika, nas preostanek premika niti ne zanima in se nam z nadaljnjim popravljanjem grafa ni treba ukvarjati (če pa bi to storili, bi se lahko izkazalo, da se bo f pri tem premiku še povečeval in nam bo trenutni rez razrezal desko ne na dva, ampak na še več kosov). Podobno je pri premiku 2.3, kjer smo dodali točko, razdelili neko obstoječo povezavo na dve in dodali še eno novo povezavo; torej se v poveča za 1, e pa za 2 in zato se f poveča za 1, torej deska razpade.

(a) Prvih nekaj rezov mogoče poteka po robovih deske; prvi rez, s katerim rob deske zapustimo, je gotovo oblike 2, ker poteka po dotlej še nedotaknjeni notranjosti deske. Ali lahko sčasoma pride do premika oblike 1? Mislimo si prvi tak premik; recimo brez izgube za splošnost, da je vodoraven. Ker je vsak premik pravokoten na prejšnjega, je bil prejšnji premik (ki je bil še oblike 2) navpičen. Naš premik oblike 1 mora torej potekati po neki še zgodnejši daljici, na kateri se je končal naš prejšnji navpični premik oblike 2. Toda premik oblike 2, ki se konča na neki že obstoječi daljici, je s tem oblike 2.2 ali 2.3, za tidve pa smo videli, da z njima deska razpade in se naš postopek konča. Do premika oblike 1 torej tu v resnici ne more priti — in prav to je tisto, zaradi česar je ta različica naloge lažja od (b).

Potek dogajanja pri različici (a) je torej tak: najprej nekaj (0 ali več) premikov po robovih deske (s katerimi se nam ni treba ukvarjati); nato nekaj (0 ali več) premikov oblike 2.1; in končno nek premik oblike 2.2 ali 2.3, s katerim deska razpade. Tak premik se seka z eno od prejšnjih daljic iz premikov oblike 2.1 (ali pa se je vsaj dotakne) ali pa se konča na zunanjem robu deske. Slednjo možnost, torej da se premik konča na zunanjem robu deske, lahko obravnavamo enako kot prvo, če v mislih med premike oblike 2.1 dodamo še štiri daljice, ki predstavljajo zunanje robove deske. Tako smo dobili naslednji postopek:

$L :=$ seznam daljic, ki na začetku vsebuje vse štiri robove deske;
$P := (0, 0)$; (* *trenutni položaj* *)
za vsak premik $(\triangle x_i, \triangle y_i)$ po vrsti:
  $Q := P + (\triangle x_i, \triangle y_i)$; (* *novi položaj* *)
  if daljica PQ leži na robu deske **then**
    $P := Q$; **continue**;
  else if ima daljica PQ kakšno skupno točko (ki ni P) s kakšno
      izmed daljic iz L **then**
    deska je razpadla; **break**;
  dodaj PQ v L; $P := Q$;

Preverjanje, ali imata dve daljici kakšno skupno točko, je zelo preprosto: ker so naše daljice le vodoravne ali navpične, si jih lahko predstavljamo kot (neskončno ozke) pravokotnike. Zato lahko uporabimo kar pravilo za izračun preseka dveh pravokotnikov:

vhod: daljici AB in CD;
(* *Predstavljajmo si vsako daljico kot pravokotnik in določimo njegov levi, desni, spodnji in zgornji rob.* *)
$l_1 := \min\{A_x, B_x\}$; $d_1 := \max\{A_x, B_x\}$; $s_1 := \min\{A_y, B_y\}$; $z_1 := \max\{A_y, B_y\}$;
$l_2 := \min\{C_x, D_x\}$; $d_2 := \max\{C_x, D_x\}$; $s_2 := \min\{C_y, D_y\}$; $z_2 := \max\{C_y, D_y\}$;
(* *Določimo robove preseka obeh pravokkotnikov.* *)
$l_3 := \max\{l_1, l_2\}$; $d_3 := \min\{d_1, d_2\}$; $s_3 := \max\{s_1, s_2\}$; $z_3 := \min\{z_1, z_2\}$;
if $l_3 > d_3$ **or** $s_3 > z_3$ **then**
daljici nimata skupnih točk;
else if $l_3 = d_3$ **and** $s_3 = z_3$ **then**
daljici imata eno samo skupno točko, (l_3, s_3);
else
daljici imata neskončno skupnih točk (torej se vsaj delno prekrivata);

(b) Tudi pri težji različici naloge ni nujno, da graf gradimo eksplicitno. Pri vsakem premiku najprej preverimo, če gre za premik tipa 1. Le-te lahko prepoznamo po tem, da daljico, ki predstavlja naš trenutni premik, v celoti pokrijejo daljice, ki predstavljajo njemu vzporedne prejšnje premike. Koristno je torej, če za vsako x-koordinato, pri kateri je doslej prišlo do kakšnega navpičnega premika, vzdržujemo urejen seznam intervalov y-koordinat, ki so jih ti premiki doslej pokrili; in podobno za vodoravne premike. Nad vsakim takim seznamom imejmo še drevesasto indeksno strukturo (npr. rdeče-črno drevo), ki nam bo omogočila v $O(\log n)$ časa (če je n število vseh premikov doslej) dodajati daljice in preverjati, ali je neka daljica povsem pokrita ali ne.

Če trenutni premik ni tipa 1, lahko nato preverimo, ali je mogoče tipa 2.1. Najprej uporabimo podatkovne strukture iz prejšnjega odstavka, da preverimo, če se kakšen del trenutne daljice prekriva s kakšno od njej vzporednih daljic iz prejšnjih premikov; če se, vemo, da ne gre za premik tipa 2.1 in da bo torej deska razpadla. Sicer pa moramo preveriti še, če ima trenutna daljica kakšno skupno točko s tistimi, ki so pravokotne nanjo (če jo ima, spet vemo, da ne gre za premik tipa 2.1 in da bo deska razpadla). To lahko naredimo enako kot pri različici (a). Da ne bo treba pregledati čisto vseh obstoječih daljic, lahko na primer vzdržujemo seznam, v katerem so navpične daljice urejene po x-koordinati, in če je trenutna daljica vodoravna od x_1 do x_2, jo primerjamo le s tistimi navpičnimi daljicami, katerih x-koordinata leži na $[x_1, x_2]$ (podobno imamo še en seznam, v katerem so vodoravne daljice urejene po y-koordinati in ki ga uporabljamo, če je trenutna daljica navpična). Še ena možnost pa je, da daljice organiziramo v kakšno prostorsko podatkovno strukturo, na primer R-drevo.

24. Sladkosnedi osel

Označimo stolpce od leve proti desni z $1, \ldots, n$ in vrstice od zgoraj navzdol prav tako z $1, \ldots, n$. Ko odstranimo nekaj robnih vrstic in stolpcev, ostane pravokotnik oblike $\{x_L, \ldots, x_D\} \times \{y_Z, \ldots, y_S\}$. Naloga ne pove natančno, kakšne so omejitve pri gibanju osla, ampak smiselno je reči, da se po obdelavi neke robne vrstice ali stolpca nahaja v enem od vogalov pravokotnika in da lahko v naslednjem koraku obdela tisto robno vrstico ali stolpec, ki se stikata prav v tem vogalu. Na primer: če je v zadnjem obhodu obdelal zgornjo vrstico, se zdaj nahaja nekje pri zgornjem levem

ali zgornjem desnem kotu pravokotnika; če se nahaja pri zgornjem levem, lahko v naslednjem obhodu obdela (naslednjo) zgornjo vrstico ali pa levi stolpec (ne pa spodnje vrstice ali desnega stolpca); ipd. V opis stanja je torej koristno poleg koordinat pravokotnika vključiti še podatek o tem, v katerem vogalu se trenutno nahaja osel. Vse skupaj lahko predstavimo s četverico (x_1, y_1, x_2, y_2), pri čemer je (x_1, y_1) tisti vogal pravokotnika, v katerem stoji osel, (x_2, y_2) pa diagonalno nasprotni vogal.

Pri takem stanju je zanimivo vprašanje ne le to, ali ga je mogoče doseči, ampak tudi, kako lačen je osel v tem stanju — z drugimi besedami, koliko obhodov je minilo od zadnjega takega obhoda, v katerem je pojedel čokolado. To namreč vpliva na to, kako se bo lahko gibal v nadaljevanju (ali se mu mudi priti do naslednje čokolade ali ne). Manj ko je osel v trenutnem stanju lačen, več svobode ima glede nadaljnjega gibanja (na primer: če je nazadnje jedel pred tremi obhodi, lahko v nadaljevanju naredi vse, kar bi lahko tudi v primeru, če je nazadnje jedel pred štirimi obhodi, obratno pa ni nujno res). Označimo torej s $f(x_1, y_1, x_2, y_2)$ najmanjše tako število t, za katero velja, da lahko osel doseže stanje (x_1, y_1, x_2, y_2) in je pri tem nazadnje jedel pred t obhodi; $t = 0$ naj pomeni, da je osel nazadnje jedel prav v tistem obhodu, s katerim je dosegel svoj trenutno vogal (x_1, y_1); $t = k$ pa naj označuje primere, ko je stanje (x_1, y_1, x_2, y_2) sploh nedosegljivo.

V stanje $\mathbf{s} := (x_1, y_1, x_2, y_2)$ je načeloma mogoče priti iz dveh predhodnih stanj, odvisno od tega, ali smo prišli vanj po vrstici ali po stolpcu. Na primer, recimo, da je $x_1 \leq x_2$ in $y_1 \leq y_2$, torej da je osel zdaj v zgornjem levem kotu pravokotnika. Prejšnji obhod je torej moral biti po stolpcu $x_1 - 1$ ali pa po vrstici $y_1 - 1$.

(1) Če po stolpcu, je moral biti to premik navzgor iz spodnjega levega kota prejšnjega (malo večjega) pravokotnika v zgornji levi kot sedanjega, torej je bilo prejšnje stanje $\mathbf{p} := (x_1 - 1, y_2, x_2, y_1)$. Označimo z $g(\mathbf{p}, \mathbf{s})$ stopnjo lakote, s katero lahko osel doseže $\mathbf{s}$ s premikom iz $\mathbf{p}$. (a) Če je $f(\mathbf{p}) = k$, je to $\mathbf{p}$ nedosegljivo in tudi o nadaljevanju poti iz njega v $\mathbf{s}$ ne moremo razmišljati, torej bo $g(\mathbf{p}, \mathbf{s}) = k$. ($b$) Sicer poglejmo, če je v stolpcu, po katerem smo se premaknili (torej v poljih $(x_1 - 1, y)$ za $y_1 \leq y \leq y_2$) kakšno polje s čokolado; če je, to pomeni, da je mogoče stanje $\mathbf{s}$ doseči z lakoto 0 in je torej $g(\mathbf{p}, \mathbf{s}) = 0$; ($c$) če pa na tistem stolpcu čokolade ni, bo osel po premiku iz $\mathbf{p}$ v $\mathbf{s}$ za eno stopnjo bolj lačen kot prej, torej $g(\mathbf{p}, \mathbf{s}) = f(\mathbf{p}) + 1$.

(2) Druga možnost pa je, da je osel na trenutni položaj prišel po vrstici, torej je bilo prejšnje stanje $\mathbf{p}' := (x_2, y_1 - 1, x_1, y_2)$. Zdaj lahko določimo $g(\mathbf{p}', \mathbf{s})$ z enakim razmislekom kot pri (1); pri (b) moramo pregledati vrstico, po kateri se je osel nazadnje premaknil, torej polja $(x, y_1 - 1)$ za $x_1 \leq x \leq x_2$.

Ko opravimo razmislek v prejšnjih dveh odstavkih, vzemimo za $f(\mathbf{s})$ seveda ugodnejšo od obeh možnosti: $f(\mathbf{s}) := \min\{g(\mathbf{p}, \mathbf{s}), g(\mathbf{p}', \mathbf{s})\}$. Doslej smo gledali primer, ko je osel v zgornjem levem kotu pravokotnika, razmislek za ostale možnosti pa je analogen.

Funkcijo f lahko torej računamo z rekurzivnim podprogramom:

funkcija $f(\mathbf{s})$:
 vhod: stanje $\mathbf{s} = (x_1, y_1, x_2, y_2)$;
1 določi prejšnje stanje $\mathbf{p}$ za premik po stolpcu in izračunaj $g(\mathbf{p}, \mathbf{s})$;
2 določi prejšnje stanje $\mathbf{p}'$ za premik po vrstici in izračunaj $g(\mathbf{p}', \mathbf{s})$;
3 vrni $\min\{g(\mathbf{p}, \mathbf{s}), g(\mathbf{p}', \mathbf{s})\}$;

Da bo rešitev učinkovitejša, moramo dodati le še pomnjenje (memoizacijo) rezul-

tatov, tako da ne bomo računali vrednosti $f(\mathbf{s})$ po večkrat za isto stanje $\mathbf{s}$. Lahko pa vrednosti funkcije f računamo tudi čisto sistematično od manjših pravokotnikov proti večjim. Tako smo dobili rešitev z dinamičnim programiranjem, ki ima časovno zahtevnost $O(n^4)$, saj imamo za x_1, y_1, x_2 in y_2 po n možnih vrednosti, z vsakim stanjem pa se moramo ukvarjati $O(1)$ časa. Rezultat, ki nas na koncu zanima, je minimum vrednosti $f(\mathbf{s})$ za tista stanja, ki predstavljajo celotno njivo, osel pa je v enem od štirih vogalov njive, torej $(1, 1, n, n)$, $(n, 1, 1, n)$, $(1, n, n, 1)$ in $(n, n, 1, 1)$.

Pri implementaciji funkcije f moramo paziti še na to, da bomo znali dovolj hitro preverjati, ali je v vrstici/stolpcu, po katerem se je osel nazadnje premaknil, kaj čokolade ali ne. Da ne bomo preverjali tega z zanko po vseh poljih v vrstici oz. stolpcu, si je koristno na samem začetku pripraviti dve tabeli: $L[x, y]$ naj pove, koliko je polj s čokolado levo od (x, y) v y-ti vrstici, $N[x, y]$ pa, koliko je polj s čokolado nad (x, y) v x-tem stolpcu. Vse to lahko naredimo v $O(n^2)$ časa. Če moramo kasneje na primer preveriti, ali je v vrstici pravokotnika, ki jo tvorijo polja (x, y) za $x_1 \leq x \leq x_2$, kaj čokolade, lahko to naredimo preprosto tako, da pogledamo, če je $L[x_2 + 1, y] > L[x_1, y]$. Podobno je tudi pri stolpcih.

25. Wikipedija (I)

Nalogo se da rešiti na več načinov. Preprosta možnost je, da beremo vhodno besedilo po vrsticah, pri tem pa v neki spremenljivki hranimo naslov trenutnega članka. Ko pridemo do konca članka (vrstica oblike `#####`), postavimo ta naslov na neko neveljavno vrednost (npr. prazen niz), kar nam bo pri naslednji vrstici povedalo, da je to zdaj naslov naslednjega članka in ne del vsebine članka.

Za vsako vrstico, ki ni niti `#####` niti naslov članka, pa se moramo v zanki premikati po njej in iskati podnize `[[`. Ko tak podniz najdemo, poiščemo še prvi naslednji podniz `]]`; zdaj vemo, da je del niza med njima opis povezave. Pogledati moramo še, če je na tem območju tudi kak znak `|`, in če je, ta znak in vse desno od njega odrezati; tako nam bo ostal le naslov članka, na katerega kaže povezava. Te naslove potem dodajajmo v slovar, ki ga na koncu vrnemo.

Oglejmo si implementacijo takšne rešitve v pythonu:

```
def IzlusciPovezave(f):
    povezave = {}
    naslov = None # naslov trenutnega članka
    for s in f:
        s = s.strip()

        # Če smo na koncu članka, pobrišimo trenutni naslov.
        # Pri naslednji vrstici bomo opazili, da je naslov prazen,
        # in ga zato prebrali iz tiste vrstice.
        if s == "#####": naslov = None; continue
        if naslov is None: naslov = s; povezave[s] = []; continue

        # Sicer pa poiščimo povezave v trenutni vrstici.
        i = 0
        while True:

            # Poiščimo začetek naslednje povezave.
            i = s.find("[[", i)

            # Če ni v nizu nobene povezave več, končajmo.
            if i < 0: break
```

```
        # Poiščimo konec te povezave.
        i += 2; j = s.find("]]", i)
        # Izrežimo del niza med "[[" in "]]".
        povezava = s[i:j]; i = j + 2
        # Če je v povezavi znak "|", obdržimo le del pred njim.
        j = povezava.find("|")
        if j >= 0: povezava = povezava[:j]
        # Shranimo povezavo v slovar „povezave“.
        povezave[naslov].append(povezava)
    return povezave
```

Zgornji podprogram kot parameter `f` pričakuje nek objekt, ki vrača vrstice besedila — lahko mu podamo npr. datoteko, ki smo jo odprli z `open`.

Še lažje je, če si pomagamo z regularnim izrazom. Razmislimo, kako bi sestavili primeren izraz:

- V naslovu se lahko pojavi katerikoli znak razen `|`. To opišemo z izrazom `[^|]`.
- Takih znakov je lahko poljubno mnogo: `[^|]*?`. Ponavljanje smo opisali z `*?`, ker hočemo, da se naš regularni izraz ujame s čim krajšim delom vrstice, ne čim daljšim. V nasprotnem primeru bi namreč pri vrstici oblike `[[a]]b[[c]]` pomotoma mislili, da gre za eno povezavo na članek z naslovom `a]]b[[c`.
- Del izraza, ki se ujame z naslovom članka, ovijmo v oklepaje, da bomo lahko do tega naslova kasneje prišli z metodo `group`: `([^|]*?)`.
- Za naslovom lahko pride znak `|` in nato še poljubno zaporedje znakov: `\|.*?`. Pred `|` smo morali dati poševnico `\`, ker je drugače `|` v regularnih izrazih metaznak (loči dva podizraza, za katera je dovolj, če se vhodni niz ujame s katerimkoli od njiju).
- Ker pa ta del povezave ni obvezen, ga ovijmo v oklepaje in dodajmo še en vprašaj: `(\|.*?)?`.
- Oboje (naslov in tisto, kar pride za njim) staknimo skupaj in dodajmo še dvojne oglate oklepaje in zaklepaje. Pred znake `[` in `]` postavimo poševnico `\`, s čimer povemo, da oglatih oklepajev tu ne uporabljamo kot metaznakov v regularnem izrazu, ampak kot nekaj, kar se mora pojaviti v vhodnem nizu. Tako dobimo `\[\[([^\|]*?)(\|.*?)?\]\]`.

Ko enkrat imamo regularni izraz, se lahko v zanki zapeljemo po vseh pojavitvah tega izraza v trenutni vrstici in za vsako od njih dodamo naslov ciljne povezave v ustrezni seznam:

```
import re

def IzlusciPovezave2(f):
    povezave = {}; naslov = None
    rex = re.compile(r"\[\[([^\|]*?)(\|.*?)?\]\]")
    for s in f:
        s = s.strip()
        if s == "#####": naslov = None; continue
        if naslov is None: naslov = s; povezave[s] = []; continue
        for m in rex.finditer(s):
            povezave[naslov].append(m.group(1))
    return povezave
```

26. Wikipedija (II)

(*a*) Z iskanjem v širino ali v globino lahko poiščemo vse strani, ki so dosegljive iz začetne strani. Vzdrževali bomo množico strani, ki smo jih na ta način že odkrili (v spodnji rešitvi je to spremenljivka dosegljive) in seznam strani, ki smo jih že odkrili, ne pa še pregledali, kam se da priti iz njih (v spodnji rešitvi je to seznam toDo). Na začetku dodamo vanju le glavno stran, nato pa na vsakem koraku vzamemo neko stran u iz seznama in pregledamo njene naslednice (strani, na katere kažejo povezave iz u — pri tem pazimo tudi na možnost, da u v slovarju ni nujno prisotna; lahko smo do nje prišli prek rdeče povezave); za vsako tako naslednico v preverimo, če smo jo kdaj prej že dosegli, in če je nismo, jo dodamo tako v množico dosegljive kot v seznam toDo. Prej ali slej na ta način dosežemo in pregledamo vse strani, ki so dosegljive iz glavne strani. Nato se moramo le še enkrat zapeljati po vseh straneh v slovarju in za vsako pogledati, ali je med dosegljivimi; če ni, to pomeni, da je sirota. Ker smo v pythonu reševali že prejšnjo nalogo, nadaljujmo v njem še pri tej:

```
def PoisciSirote(slovar):
    toDo = ["main"]; dosegljive = set(toDo)
    while toDo:
        u = toDo.pop()
        for v in slovar.get(u, []):
            if v in dosegljive: continue
            toDo.append(v); dosegljive.add(v)
    return [u for u in slovar if u not in dosegljive]
```

(*b*) Moramo se le zapeljati po vseh povezavah, na primer z eno zanko po vseh straneh u v našem slovarju in nato z gnezdeno zanko po vseh naslednicah v posamezne strani u; pri vsaki povezavi (u, v) nato preverimo, če v obstaja kot ključ v slovarju — če ne obstaja, gre za rdečo povezavo in seznam takšnih bomo vrnili kot rezultat našega podprograma.

```
def PoisciRdecePovezave(slovar):
    return [(u, v) for u in slovar if v not in slovar]
```

(*c*) Za iskanje krepko povezanih komponent je znanih več algoritmov. Pomagamo si lahko z iskanjem v globino, torej podobnim postopkom, kot smo ga že uporabili v točki (*a*) pri iskanju sirot. Če poženemo iskanje v globino ne iz glavne strani kot pri (*a*), ampak iz poljubne sirote, bodo vse strani, ki jih bomo na ta način dosegli, tudi sirote; ni pa nujno, da vse pripadajo eni sami krepko povezani komponenti. Za nadaljnje razmišljanje je lažje, če postopek iskanja v globino zapišemo kot rekurziven podprogram:

podprogram OBIŠČI(u):
  $s[u] := t$; $t := t + 1$;
  za vsako povezavo $u \to v$, če je v sirota:
    if $s[v] < 0$ **then** OBIŠČI(v);
  $e[u] := t$; $t := t + 1$;

(* *v glavnem delu programa* *)
$t := 0$; za vsako siroto u: postavi $s[u]$ in $e[u]$ na -1;
za vsako siroto u:
  if $s[u] < 0$ **then** OBIŠČI(u);

V tabelah s in e torej shranimo čas vstopa v posamezno stran in izstopa iz nje, poleg tega pa ju uporabljamo tudi, da preverimo, če smo na stran kdaj prej že naleteli.

Če zberemo vse povezave $u \to v$, pri katerih smo izvedli rekurzivni klic OBIŠČI(v), nastane vpeto drevo; recimo mu T. Pravzaprav lahko nastane več ločenih dreves (za vsak klic OBIŠČI(u) iz glavne zanke nastane po eno drevo), vendar je za razmislek v naslednjem odstavku dovolj, če gledamo drevesa eno po eno.

Mislimo si neko krepko povezano komponento A; prvo stran iz A, ki jo obiščemo pri našem iskanju v globino, označimo z x. Preden se vrnemo iz x, bomo obiskali tudi vse strani, ki so dosegljive iz x in ki prej še niso bile obiskane; torej bomo takrat obiskali tudi vse preostale strani iz A. Vse te strani bodo torej v x-ovem poddrevesu (znotraj drevesa T), ne bodo pa nujno vse strani iz tega poddrevesa v x-ovi krepko povezani komponenti, pač pa le tiste, iz katerih je dosegljiva x. Poiščemo jih lahko torej tako, da začnemo v x in sledimo povezavam v nasprotni smeri (pri tem je vseeno, ali uporabimo iskanje v globino ali v širino), pri čemer pa se omejimo le na strani iz x-ovega poddrevesa. Strani, ki jih na ta način dosežemo, tvorijo ravno x-ovo krepko povezano komponento, torej A. To, ali neka stran y leži v x-ovem poddrevesu, je enostavno preveriti s pomočjo tabel s in e: y je v x-ovem poddrevesu natanko tedaj, ko je $s[u] < s[v]$ in $e[v] < e[u]$.

Pri razmisleku v prejšnjem odstavku je bilo pomembno, da smo za x vzeli tisto stran iz A, ki smo jo v iskanju v globino obiskali kot prvo. Koristno je torej, če strani pregledujemo po naraščajočem času vstopa $s[u]$ (lahko si med iskanjem v globino pripravimo seznam, v katerem so strani urejene po $s[u]$); pri vsaki strani pogledamo, ali smo jo že uvrstili v kakšno krepko povezano komponento, in če je nismo, to pomeni, da je ona prva obiskana stran iz neke nove krepko povezane komponente (preostanek te komponente pa lahko zdaj raziščemo po postopku iz prejšnjega odstavka).

Ker bomo morali slediti povezavam v nasprotni smeri, si je koristno pripraviti „obrnjeni“ slovar, v katerem so ključi spet strani, pripadajoče vrednosti pa so seznami neposrednih predhodnic (namesto naslednic kot v vhodnem slovarju) posamezne strani. Tak obrnjeni slovar si lahko pripravimo spotoma med iskanjem v globino v prvi fazi algoritma.

Tako smo dobili naslednjo rešitev:

```
def PoisciKrepkoPovezaneKomponente(slovar):
    sirote = set(PoisciSirote(slovar))

    # Prva faza: iskanje v globino, pri katerem si zapomnimo čas vstopa s[u]
    # in izstopa e[u] za vsako siroto. Spotoma pripravimo še obrnjeni slovar.
    s = {}; e = {}; vrstniRed = []; obrnjeniSlovar = {}; t = 0
    def Obisci(u):
        nonlocal t; s[u] = t; t += 1;
        vrstniRed.append(u)
        for v in slovar[u]:
            if v not in sirote: continue
            if v not in obrnjeniSlovar: obrnjeniSlovar[v] = [u]
            else: obrnjeniSlovar[v].append(u)
            if v not in s: Obisci(v)
        e[u] = t; t += 1

    for u in sirote:
        if u not in s: Obisci(u)
```

```
    # Druga faza: gremo po naraščajoči vrednosti s[u] in pregledujemo krepko
    # povezane komponente. Množica c vsebuje strani, ki smo jih že uvrstili
    # v eno od dosedanjih komponent.
    komponente = []; c = set()
    def PreglejKomponento(u, koren):
        # Parameter „koren“ je stran, pri kateri smo začeli pregledovati trenutno
        # komponento; ostati moramo znotraj njenega poddrevesa.
        c.add(u); komponente[−1].append(u)
        for v in obrnjeniSlovar.get(u, []):
            if v in s and v not in c and s[koren] < s[v] and e[v] < e[koren]:
                PreglejKomponento(v, koren)
    for u in vrstniRed:
        if u not in c: komponente.append([]); PreglejKomponento(u, u)
    return komponente
```

(*d*) Pomagamo si lahko s krepko povezanimi komponentami, ki smo jih našli pri podnalogi (*c*). Če dodamo povezavo na poljubno stran v neki krepko povezani komponenti, se bo dalo iz nje po povezavah znotraj komponente priti tudi do vseh drugih strani v tej komponenti. Dovolj je torej, če za vsako komponento dodamo največ eno novo povezavo, ki kaže vanjo (iz poljubne strani, ki ni sirota; lahko na primer kar iz glavne strani). Upoštevati pa moramo še možnost, da lahko že v vhodnem grafu obstajajo povezave iz ene krepko povezane komponente v drugo. V tem primeru je dovolj že, če dodamo povezavo iz kakšne strani, ki ni sirota, v eno od strani prve komponente, od tam pa se bo potem dalo priti ne le v vse ostale strani prve komponente, ampak tudi v drugo komponento. Dodati moramo torej natanko toliko novih povezav, kolikor je krepko povezanih komponent, v katere ne kaže nobena povezava od zunaj.

```
def KolikoNovihPovezav(slovar):
    # Predpostavimo, da smo funkcijo PoisciKrepkoPovezaneKomponente
    # spremenili tako, da poleg spremenljivke „komponente“ vrne tudi „obrnjeniSlovar“.
    komponente, obrnjeniSlovar = PoisciKrepkoPovezaneKomponente(slovar)
    return sum(1 for k in komponente
        if not any(v for u in k for v in obrnjeniSlovar.get(u, []) if v not in k))
```

27. Praljudje

Če pregledujemo ljudi od prednikov k potomcem, lahko za vsakega sproti izračunamo porazdelitev genetskega materiala, torej od katerih praljudi ga ima in v kakšnih razmerjih. Začnemo s praljudmi, pri katerih je stvar trivialna (celoten genetski material je od tega pračloveka samega), nato pa lahko na vsakem koraku obdelamo poljubnega takega človeka, za katerega smo pred tem že obdelali oba starša. V nalogi se torej skriva pregled grafa v topološkem vrstnem redu.

Porazdelitev genetskega materiala pri posameznem človeku lahko predstavimo kot seznam parov (*pračlovek*, *delež*), ki povedo, da ima ta človek tolikšen delež svojega genetskega materiala od tistega pračloveka. Najbolje je, če so ti seznami urejeni po praljudeh (kakšen točno je ta vrstni red, ni pomembno; važno je le, da je dosleden), saj lahko potem seznama od obeh staršev skombiniramo preprosto z zlivanjem in tako sestavimo seznam za potomca. Medtem ko pripravljamo novi seznam, lahko sproti še preverjamo, če je od kakšnega pračloveka več kot polovica

krvi; tako bomo, ko bo seznam končan, tudi vedeli, čigavega rodu je ta človek oz. ali je mešane krvi.

Zapišimo ta postopek s psevdokodo:

```
za vsakega človeka u: otroci[u] := prazen seznam;
Q := prazna množica;
za vsakega človeka u:
  if je u pračlovek: then dodaj u v Q; continue;
  d[u] := 2; za vsakega u-jevega starša p: dodaj u na seznam otroci[p];
while Q ni prazna:
  naj bo u poljuben element Q; pobriši ga iz Q;
  if je u pračlovek then
    G[u] := seznam z enim samim elementom (u, 1);
  else:
    naj bosta o in m njegova starša;
    izračunaj G[u] z zlivanjem G[o] in G[m];
  za vsakega v iz otroci[u]:
    zmanjšaj d[v] za 1; if d[v] = 0 then dodaj v v Q;
```

Porazdelitev genetskega materiala za človeka u torej hranimo v $G[u]$. V množici Q hranimo tiste ljudi, za katere še nismo izračunali $G[u]$, smo ga pa že izračunali za njihove starše; v praksi lahko Q implementiramo na primer kot vrsto ali sklad. Vrednost $d[u]$ pove, za koliko u-jevih staršev še ne poznamo $G[u]$. Ko za nekega človeka izračunamo $G[u]$, gremo torej po njegovih otrocih in jim zmanjšamo d za 1, ko pa pade d na 0, dodamo tistega otroka v Q. Prej ali slej na ta način obdelamo vse ljudi.

Oglejmo si podrobneje še postopek za izračun porazdelitve genetskega materiala pri otroku iz porazdelitev pri njegovih starših. To je klasičen primer zlivanja (*merging*) dveh urejenih seznamov. Z indeksoma i_1 in i_2 se sprehajamo hkrati po seznamih obeh staršev; na vsakem koraku pogledamo, pri katerem od staršev se trenutni par nanaša na „zgodnejšega“ pračloveka (zgodnejšega glede na vrstni red, po katerem so urejeni praljudje v naših seznamih), in prenesemo tega pračloveka v izhodni seznam s polovičnim deležem ter se premaknemo za eno mesto naprej po vhodnem seznamu tistega starša, pri katerem smo ga dobili; če pa pri obeh starših zagledamo istega pračloveka, prenesemo v izhodni seznam povprečje njegovega deleža pri obeh starših in se premaknemo za eno mesto naprej po obeh vhodnih seznamih.

```
vhod: seznama staršev, recimo G1 dolžine n1 in G2 dolžine n2;
      njuni elementi so pari oblike (p, d), ki povedo pračloveka in delež;
izhod: podoben seznam G za njunega otroka.

G := prazen seznam; i1 := 1; i2 := 1;
while i1 ≤ n1 or i2 ≤ n2:
  if i1 ≤ n1 then (p1, d1) := G1[i1] else p1 := ∞;
  if i2 ≤ n2 then (p2, d2) := G2[i2] else p2 := ∞;
  if p1 = p2 then
    dodaj v G par (p1, (d1 + d2)/2); i1 := i1 + 1; i2 := i2 + 1;
  else if p1 < p2 then
```

```
      dodaj v G par (p_1, d_1/2); i_1 := i_1 + 1;
   else
      dodaj v G par (p_2, d_2/2); i_2 := i_2 + 1;
return G;
```

V gornjo psevdokodo bi morali dodati še preverjanje, ali je pri kakšnem paru (p, d) v G delež d večji od 1/2; če da, lahko izpišemo, da je otrok, za katerega računamo G, p-jevega rodu, sicer pa, da je mešane krvi.

Časovna zahtevnost opisane rešitve je v najslabšem primeru $O(n^2)$, če v vhodnih podatkih nastopa n ljudi, saj se lahko z nekaj smole zgodi, da pri $O(n)$ ljudeh nastane seznam $G[u]$ dolžine $O(n)$.

28. Brisanje duplikatov

Skripta, ki jo bomo zgenerirali, bo tako ali drugače izvajala naslednji postopek:

```
zadnji = 0;
for (nasl = 1; nasl < n; nasl++)
   if (T[nasl] != T[zadnji])
      T[++zadnji] = T[nasl];
```

Na začetku vsake iteracije te zanke torej velja, da smo iz prvih nasl elementov tabele že pobrisali duplikate in da je tako skrajšano zaporedje shranjeno na indeksih od 0 do vključno zadnji. V naslednji iteraciji moramo preveriti, če je naslednji element (na indeksu nasl) enak zadnjemu dosedanjemu (na indeksu zadnji); če ni, moramo naslednjega skopirati za zadnjega in on potem postane novi zadnji (zato se indeks zadnji poveča za 1).

Tega postopka ni težko zapisati kot kratko skripto v našem zbirnem jeziku, na primer takole:

```
      GE n, 1              # n = 0 in n = 1 obravnavajmo posebej.
      CJMP netrivialno
      STOP n               # Pri n ≤ 1 vrnimo kar n.
netrivialno:
      SET zadnji, 0        # Inicializirajmo spremenljivki.
      SET nasl, 1
zanka:
      GE nasl, n           # Če smo pri n, končajmo.
      CJMP konec
      AEQ zadnji, nasl     # Naslednji element preskočimo,
      CJMP naprej          # če je enak zadnjemu.
      ADD zadnji, 1        # Sicer ga skopirajmo za doslej zadnjega
      COPY zadnji, nasl    # in bo postal novi zadnji.
naprej:
      ADD nasl, 1          # Premaknimo se naprej po vhodnem zaporedju
      JMP zanka            # in nadaljujmo z zanko.
konec:
      ADD zadnji, 1        # Vrnimo vrednost zadnji + 1.
      STOP zadnji
```

Trivialna primera $n = 0$ in $n = 1$, ko ni treba v tabeli ničesar spreminjati in lahko takoj vrnemo kar n, smo torej obravnavali posebej, za večje n pa teče naša zanka $n - 1$ iteracij (ker gre nasl od 1 do vključno $n - 1$) in v vsaki iteraciji izvedemo

v najslabšem primeru 8 korakov (če zaznamo duplikat, pa dva manj, ker ni treba povečati števca zadnji in skopirati naslednjega elementa). Poleg tega imamo še štiri korake pred začetkom zanke in štiri po zadnji iteraciji (dva za zadnje preverjanje ustavitvenega pogoja in dva za vračanje zadnji + 1). Tako imamo v najslabšem primeru ravno $8n$ korakov. Tabela v besedilu naloge nam pove, da dobimo za táko rešitev le 50 % točk; poskusimo jo torej še izboljšati.

Gornja rešitev preverja ustavitveni pogoj (ali je nasl $\geq$ n) na začetku zanke. Ker smo primera $n \leq 1$ obravnavali posebej, ustavitveni pogoj na začetku prve iteracije gotovo ni izpolnjen (saj je nasl $= 1$ in $n \geq 2$); lahko torej preverjanje ustavitvenega pogoja premaknemo na konec zanke. Namesto da preverjamo, ali se je treba ustaviti, bomo raje preverjali, če je treba nadaljevati; če da, skočimo nazaj na začetek zanke, če ne, pa se izvajanje samo po sebi nadaljuje pri naslednji inštrukciji za koncem zanke:

```
    GE n, 1               # n = 0 in n = 1 obravnavajmo posebej.
    CJMP netrivialno
    STOP n                # Pri n ≤ 1 vrnimo kar n.
netrivialno:
    SET zadnji, 0         # Inicializirajmo spremenljivki.
    SET nasl, 1
zanka:
    AEQ zadnji, nasl      # Naslednji element preskočimo,
    CJMP naprej           # če je enak zadnjemu.
    ADD zadnji, 1         # Sicer ga skopirajmo za doslej zadnjega
    COPY zadnji, nasl     # in bo postal novi zadnji.
naprej:
    ADD nasl, 1           # Premaknimo se naprej po vhodnem zaporedju
    LT nasl, n            # Če še nismo pri n,
    CJMP zanka            # izvedimo še eno iteracijo zanke.
konec:
    ADD zadnji, 1         # Vrnimo vrednost zadnji + 1.
    STOP zadnji
```

Namesto treh korakov za skok na začetek zanke in preverjanje ustavitvenega pogoja imamo zdaj le dva. To je 7 korakov za vsako iteracijo zanke, pa še štirje pred zanko in dva po njej, skupaj $7n - 1$. Ker smo prišli pod mejo $7n$ korakov, dobimo za to rešitev 60 % točk.

Če pogledamo, kaj vse moramo storiti v vsaki iteraciji zanke, vidimo, da se je večini teh stvari težko izogniti: preverjanje, če je naslednji element enak zadnjemu; pogojni skok po njem (šteje za en korak, tudi če se ne izvede); kopiranje in povečevanje indeksa zadnji (če ni bilo duplikata — na kar pa ne moremo vplivati); povečevanje indeksa nasl (da se sploh premaknemo naprej po tabeli). To je že pet korakov na iteracijo. Da spravimo rešitev pod $5{,}5n$ korakov (kar naloga zahteva za 80 % točk), nam ostane povprečno le še pol koraka na iteracijo za preverjanje ustavitvenega pogoja in skok na začetek zanke. To lahko dosežemo le tako, da teh reči ne počnemo v vsaki iteraciji zanke.

Izberimo si na primer nek $t > 1$ in združimo t iteracij naše dosedanje zanke v en sam blok brez vmesnega preverjanja ustavitvenega pogoja. Tak blok t iteracij smemo izvesti, če na začetku bloka velja nasl $\leq n - t$ namesto dosedanjega $\leq n - 1$. Ko ta pogoj ne velja več — torej ko je nasl že precej blizu n-ja in smo že skoraj na koncu tabele — pa izvedimo še preostale iteracije naše prvotne zanke po enakem

postopku kot zgoraj. Ker znotraj bloka ni treba preverjati ustavitvenega pogoja, imamo tam le 5 korakov za vsak element tabele, nato pa še 2 koraka za ustavitveni pogoj; skupaj torej $5t+2$ korakov za obdelavo t elementov. Na koncu nam ostane v najslabšem primeru še $t-1$ elementov, ki jih moramo obdelati po starem s 7 koraki na iteracijo. Tako smo pri približno $(5t+2)(n/t)+7(t-1)+O(1)$ korakih, kar je približno $(5+2/t)n+O(1)$. Če vzamemo $t=5$, bomo pri dovolj velikih n že prišli pod mejo $5{,}5n$.

Doslej si naša rešitev ni še nič pomagala z dejstvom, da poznamo n vnaprej in smemo zgenerirati skripto, ki je posebej prilagojena le temu n. S tem dejstvom lahko prejšnjo rešitev še izboljšamo: vzemimo kar blok dolžine $t=n-1$ iteracij. Ustavitvenega pogoja nam sploh ni treba preverjati, saj skripta kot celota vsebuje natanko toliko iteracij, kolikor jih potrebujemo; prav tako tudi odpadejo skoki s konca zanke nazaj na začetek, saj je zanka v celoti razvita v eno samo dolgo zaporedje ukazov. Tudi števca nasl ni treba povečevati, saj lahko njegovo vrednost vgradimo kot primerno konstanto v vsako iteracijo naše razvite zanke posebej.

Program, s katerim lahko zgeneriramo takšno skripto, je tudi zelo preprost. Oglejmo si primer rešitve v pythonu:

```
n = int(input())
if n <= 1: print("STOP n"); return    # Primere n ≤ 1 obravnavajmo posebej.
print("SET zadnji, 0")
for nasl in range(1, n):
    print("AEQ zadnji, %d" % nasl)    # Primerjamo zadnji element z naslednjim.
    print("CJMP konec_%d" % nasl)     # Če sta enaka, kopiranje preskočimo.
    print("ADD zadnji, 1")            # Skopiramo naslednji element za doslej zadnjega
    print("COPY zadnji, %d" % nasl)   # in on s tem postane novi zadnji.
    print("konec_%d:" % nasl)
print("ADD zadnji, 1")                # Vrnemo zadnji + 1.
print("STOP zadnji")
```

V vsaki iteraciji naše razvite zanke so le štirje koraki; poleg tega imamo še en korak pred zanko in dva po njej, skupaj torej $4(n-1)+3=4n-1$ korakov, kar je dovolj za 90 % točk. Skripta, ki jo izpišemo, je sicer zdaj dolga $O(n)$ korakov namesto $O(1)$ in deluje le za en sam n, vendar to pri tej nalogi ni ovira.

S tem torej, ko smo kodo v naši skripti razmnožili po enkrat za vsako možno vrednost nasl, smo se te spremenljivke v skripti znebili in tako prihranili nekaj časa. Naravna naslednja ideja je, da se na podoben način znebimo še spremenljivke zadnji. Zdaj torej potrebujemo nek fragment kode za vsako možno kombinacijo vrednosti nasl in zadnji. Spremenljivk nasl in zadnji naša skripta nima več; to, kako se med izvajanjem spreminjata njuni vrednosti, se zdaj odraža v tem, kje v skripti se izvajanje trenutno nahaja. Naš fragment bo torej nekako takšen:

```
fragment_⟨zadnji⟩_⟨nasl⟩:
    AEQ ⟨zadnji⟩, ⟨nasl⟩
    CJMP fragment_⟨zadnji⟩_⟨nasl + 1⟩
    COPY ⟨zadnji + 1⟩, ⟨nasl⟩
    JMP fragment_⟨zadnji + 1⟩_⟨nasl + 1⟩
```

Poseben primer nastopi, ko pridemo do konca zanke; takrat moramo le še vrniti rezultat:

```
fragment_⟨zadnji⟩_⟨n⟩: STOP ⟨zadnji + 1⟩
```

Z znaki $\langle \ldots \rangle$ smo ponazorili dejstvo, da morajo biti na tistih mestih v skripti celoštevilske konstante, ki jih naš program za generiranje skripte izračuna iz vrednosti *nasl* in *zadnji* za trenutni fragment. Prvi skok (CJMP) pokrije primer, ko opazimo duplikat in moramo povečati le *nasl*, ne pa tudi *zadnji*; drugi skok (JMP) pa primer, ko moramo povečati oba, ker ni bilo duplikata. Temu drugemu skoku se lahko izognemo, če fragmente primerno zložimo skupaj, tako da fragmentu $(zadnji, nasl)$ vedno sledi fragment $(zadnji + 1, nasl + 1)$. Tako ima vsak fragment le še 3 korake (od katerih se prva dva gotovo izvedeta, tretji pa le, če ni bilo duplikata); in ker se po vsakem fragmentu vrednost *nasl* poveča za 1, se izvede teh fragmentov le $n - 1$; če prištejemo še stavek STOP na koncu, se tako dobljena skripta izvaja le $3n - 2$ korakov. Končno imamo rešitev, ki pri tej nalogi dobi vse točke. Dobljena skripta je sicer dolga $O(n^2)$ ukazov, vendar nas to nič ne moti, saj pri tej nalogi šteje le to, koliko korakov se skripta izvaja.

Drobna optimizacija, ki nikoli ne škoduje, v najslabšem primeru pa sicer tudi nič ne koristi, je, da ukaza COPY ne izpišemo, če velja $zadnji + 1 = nasl$ (torej v primerih, ko doslej nismo odkrili še nobenega duplikata), saj takrat takšno kopiranje tabele sploh ne spremeni.

Zapišimo še program v pythonu, ki zgenerira primerno skripto:

```
n = int(input())
if n <= 1: print("STOP n"); return

for razlika in range(1, n + 1):
    for nasl in range(razlika, n):
        zadnji = nasl - razlika

        # Labela na začetku trenutnega fragmenta.
        print("fragment_%d_%d:" % (zadnji, nasl))

        # Primerjamo zadnji element z naslednjim.
        print("AEQ %d, %d" % (zadnji, nasl))

        # Če sta enaka, kopiranje preskočimo; „nasl“ se poveča za 1.
        print("CJMP fragment_%d_%d" % (zadnji, nasl + 1))

        # Sicer skopiramo novi element za doslej zadnjega.
        if razlika > 1: print("COPY %d, %d" % (zadnji + 1, nasl))
        # Zdaj se tako „nasl“ kot „zadnji“ povečata za 1. Tisti fragment skripte,
        # ki ustreza njunima novima vrednostma, bo prišel takoj za trenutnim,
        # zato skoka nanj ne potrebujemo.

    # Ko pridemo do konca tabele, se ustavimo.
    print("fragment_%d_%d:" % (n - razlika, n))
    print("STOP %d" % (n - razlika + 1))
```

29. Požrešni taksist

Iskanje najdaljše poti, kot jo zahteva naša naloga, je NP-težak problem,[10] zato zanj ne poznamo res učinkovitega algoritma. Lahko bi preizkusili vse možne poti z rekurzivnim spuščanjem; na vsakem koraku pregledamo vse sosede zadnje točke na

[10] O tem se lahko prepričamo tako, da nanj prevedemo problem Hamiltonove poti, ki je znan kot NP-poln problem. Če postavimo dolžine vseh povezav na 1 in se vprašamo, ali ima najdaljša pot od s od t dolžino $n - 1$ (pri čemer je n število točk v grafu), je odgovor pritrdilen natanko tedaj, ko obstaja v grafu Hamiltonova pot. Če bi torej znali v polinomskem času iskati najdaljšo pot, bi znali iskati tudi Hamiltonovo pot.

trenutni poti (razen tistih sosed, ki jih je trenutna pot že obiskala) in za vsako od njih izvedemo nov rekurzivni klic. Ko pridemo do t, pa pogledamo, ali je dosedanja pot daljša od najdaljše znane (to si zapomnimo npr. v neki globalni spremenljivki). Dolžino povezave od u do v bomo označili z d_{uv}.

globalni spremenljivki: najdaljša pot doslej $\pi^\star$ in njena dolžina $d^\star$;

podprogram NadaljujPot(dosedanja pot π, njena dolžina d):
  naj bo u zadnja točka na poti π;
  za vsako sosedo v točke u:
    if v že leži na π **then continue**;
    dodaj v na konec poti π;
    if $v \neq t$ **then** NadaljujPot$(d + d_{uv})$
    else if $d > d^\star$ **then** $\pi^\star := \pi$; $d^\star := d$;
    pobriši v s konca poti π;

(* *Poženimo rekurzijo s potjo, ki na začetku vsebuje le s.* *)
$d^\star := -\infty$; NadaljujPot$(\langle s \rangle, 0)$;

Na koncu tega postopka je v $\pi^\star$ najdaljša pot od s do t, torej tista, po kateri sprašuje naša naloga. Slabost tega postopka je, da je lahko zelo počasen. Opazimo lahko, da se izvede po en rekurzivni klic za vsako možno pot, ki se začne pri s in ne vsebuje t. Če imamo n točk (vključno s s in t) in povezave med vsemi možnimi pari točk, je število poti, dolgih k korakov, ki se začnejo pri s in ne vsebujejo t, enako $(n-2)(n-3)\cdots(n-1-k) = (n-2)!/(n-2-k)!$. Če seštejemo to po vseh možnih k (od 0 do $n-2$), imamo $P := (n-2)! \sum_{k=0}^{n-2} 1/(n-2-k)! = (n-2)! \sum_{k=0}^{n-2} 1/k!$. Vsota konvergira proti e, ko gre $n \to \infty$ (o tem se lahko prepričamo npr. s Taylorjevo formulo), tako da je P (število vseh poti, ki se začnejo pri s in ne vsebujejo t), nekje med $(n-2)!$ in $e \cdot (n-2)!$. Vsako od teh poti lahko z enim korakom podaljšamo v pot od s do t, zato je tudi število poti od s do t enako P. Naš postopek torej izvede P rekurzivnih klicev; če sproti vzdržujemo množico še neobiskanih točk, gremo lahko v zanki le po tistih u-jevih sosedah, ki še ne ležijo na π, zato lahko ceno vsake iteracije te zanke štejemo k tisti poti, ki nastane, ko se π podaljša s korakom $u \to v$. Tako imajo torej te zanke pri vseh rekurzivnih klicih skupaj ravno $2P$ iteracij in časovna zahtevnost našega postopka je v najslabšem primeru $O((n-2)!)$.

Rešitev lahko malo izboljšamo z neke vrste dinamičnim programiranjem. Opazimo lahko, da je za nadaljevanje poti pomembno le, v kateri točki se trenutno nahajamo (v gornji psevdokodi je bila to u) in katere točke so še neobiskane; ni pa pomembno, kakšen točno je dosedanji potek poti med s in u (pomembno je le, katere točke so tam, ne pa to, v kakšnem vrstnem redu so). Označimo z $f(u, A)$ dolžino najdaljše take poti od u do t, ki vmes uporablja le točke iz A. Pravkar opravljeni razmislek nam pove, da lahko to funkcijo računamo takole:

$$f(u, A) = \max\{d_{ut}, \max\{f(v, A - \{v\}) + d_{uv} \;:\; v \in A\}\}.$$

Ena možnost je torej korak naravnost iz u v t, druga možnost pa je, da gremo iz u v neko $v \in A$ in pot nadaljujemo od tam. V resnici pridejo izmed $v \in T$ v poštev le tiste, ki so u-jeve sosede; pri tistih, ki niso, si lahko v gornji formuli predstavljamo $d_{uv} = -\infty$.

Te rekurzivne formule ne bi bilo težko predelati v rekurziven podprogram, da pa bo ta rešitev učinkovita, si moramo že izračunane vrednosti $f(u, A)$ nekje shranjevati, da jih ne bomo računali po večkrat, ampak bomo kasneje vzeli že izračunano vrednost, ko jo bomo spet potrebovali. Lahko pa jih računamo tudi sistematično od manjših množic A proti večjim; tako bomo vedno imeli pri roki rešitve manjših podproblemov, ko jih bomo potrebovali za izračun večjega podproblema.

for $k := 0$ **to** $n - 2$:
  za vsako množico $A \subseteq V - \{s, t\}$ velikosti k:
    za vsako $u \in V - \{t\} - A$:
      if $k < n - 2$ **and** $u = s$ **then continue**;
      $f[u, A] := d_{ut}$;
      za vsako $v \in A$, če je v soseda točke u:
        $f[u, A] := \max\{f[u, A], f[v, A - \{v\}] + d_{uv}\}$;
return $f[s, V - \{s, t\}]$;

Primeri z $u = s$ nas zanimajo le pri $k = n - 2$, torej ko nismo obiskali še nobenega vozlišča razen začetnega s; kasneje, ko smo že obiskali nekaj vozlišč, se gotovo ne moremo več nahajati v s (saj smo tam svojo pot začeli in smo torej zdaj gotovo nekje drugje).

Za množico A je (po vseh k skupaj) 2^{n-2} možnosti (ker lahko vsebuje katerokoli točko razen s in t), pri vsaki si lahko na $O(n)$ načinov izberemo u in pri vsakem paru (u, A) imamo potem $O(n)$ dela, da pregledamo vse $v \in A$ in izračunamo $f(u, A)$. Časovna zahtevnost tega postopka je torej $O(2^n \cdot n^2)$. To je sicer še vedno eksponentno v odvisnosti od n, je pa vseeno občutno hitreje kot $O(n!)$, kar smo imeli pri prejšnji rešitvi. Slabost tega postopka pa je, da porabi tudi $O(2^n \cdot n)$ pomnilnika za shranjevanje rezultatov, torej že izračunanih vrednosti $f(u, A)$.

30. Konjunkcije

Opazimo lahko, da je $a_{ij} = a_{i,j-1} \,\&\, x_j$, torej lahko pri fiksnem i preprosto računamo vrednosti a_{ij} po naraščajočih j. Tako pridemo do naslednje preproste, vendar prepočasne rešitve:

$s := 0$;
for $i := 1$ **to** n:
  $a := x_i$; $s := s + a$;
  for $j := i + 1$ **to** n:
    $a := a \,\&\, x_j$; $s := s + a^2$;
izpiši s;

Njena časovna zahtevnost je $O(n^2)$, kar je za naše namene že prepočasi, saj naloga pravi, da gre lahko n do 10^6. Do boljše rešitve pridemo, če upoštevamo, kako deluje binarni operator IN. Ko v gornjem postopku popravimo a z izrazom $a := a \,\&\, x_j$, se lahko v a-ju kakšen prižgan bit ugasne, nikoli pa se ne more kak ugasnjen bit prižgati. Če je imel a na začetku (pri $a = x_i$) prižganih k bitov, se v notranji zanki (po j) lahko spremeni največ k-krat, ko se tisti biti eden po eden ugašajo (lahko je sprememb še manj, npr. če se kdaj ugasne več bitov hkrati). Potratno je torej, da notranja zanka računa $O(n)$ vrednosti a_{ij}, ko pa jih je lahko le $O(k)$ različnih. Bolje

bi bilo, če bi izračunali vrednosti a_{ij} le tam, kjer pride do sprememb, pri tem še ugotovili, koliko zaporednih vrednosti je bilo enakih, in kvadrat dosedanje vrednosti pomnožili z njihovim številom, preden bi ga prišteli k s. Tako dobimo naslednji postopek:

```
s := 0;
for i := 1 to n:
   a := x_i; j := i;
   while a > 0:
      (* Na tem mestu spremenljivka a vsebuje vrednost a_ij. *)
      j' := najmanjši tak indeks, ki je ≥ j in pri katerem je v x_j'
         ugasnjen kakšen tak bit, ki je v a prižgan (da bo zagotovo obstajal,
         si mislimo, da imamo na koncu zaporedja še člen x_{n+1} = 0);
      (* Zdaj vemo, da so a_ij, a_{i,j+1}, ..., a_{i,j'-1} enake,
         a_{i,j'} pa je drugačna od njih. *)
      s := s + (j' − j) · a^2;
      j := j';  a := a & a_j;
izpiši s;
```

Vprašanje je še, kako učinkovito izračunati j', torej naslednji indeks, pri katerem se vrednost a spremeni. Načeloma moramo za vsak bit, ki je v začetni vrednosti a-ja, to je $a = x_i$, prižgan, pogledati, kateri je prvi naslednji indeks $j > i$, pri katerem je v x_j ta bit ugasnjen. Tako dobljene indekse j uredimo naraščajoče in tako dobimo zaporedje indeksov, pri katerih pride do sprememb pri a (ker se biti, ki so bili sprva — pri $a = x_i$ — prižgani, postopoma ugašajo).

Označimo z m_{ib} prvi tak indeks $j > i$, pri katerem je bit b v x_j ugasnjen. Hitro lahko opazimo, da je te stvari lažje računati od konca tabele proti začetku, torej po padajočih i: če je v x_{i+1} bit b ugasnjen, je $m_{ib} = i + 1$, sicer pa je $m_{ib} = m_{i+1,b}$. To lahko računamo spotoma v glavni zanki našega postopka, če jo obrnemo tako, da gre po padajočih i namesto po naraščajočih. Naša rešitev je torej zdaj takšna:

```
B := ⌊log_2 max{x_1, ..., x_n}⌋;
s := 0; for b := 0 to B do m_b := n + 1;
for i := n downto 1:
   C := prazna množica;
   for b := 0 to B do
      if je bit b v x_{i+1} prižgan then m_b := i + 1;
      if je bit b v x_i prižgan then dodaj b v množico C;
   a := x_i; j := i;
   za vsak b ∈ C v naraščajočem vrstnem redu vrednosti m_b:
      s := s + (m_b − j) · a;  j := m_b;  a := a & x_j;
izpiši s;
```

Z B smo torej označili indeks najbolj levega bita, s katerim imamo v seznamu x kdaj še opravka; vsi biti nad B so pri vseh številih iz seznama ugasnjeni. Pri vsakem i imamo zdaj $O(B)$ dela, da popravimo vrednosti m_b, in potem še $O(B)$ dela, da pregledamo vse bite iz C, torej tiste, ki so v x_i prižgani. Pregledovati jih moramo naraščajoče po m_b; če zapišemo C v seznam in ga uredimo s kakšnim od običajnih

postopkov za urejanje, bo to vzelo $O(B \log B)$ časa; in ker moramo to storiti pri vsakem i, bo rešitev kot celota vzela $O(nB \log B)$ časa.

Bolje pa je, če razmislimo, kako bi se tak urejen seznam spreminjal, ko se premaknemo od $i+1$ k i in popravljamo vrednosti m_b. Do sprememb pride le tako, da se nek m_b postavi na $i+1$, to pa je manjše od dosedanjih vrednosti drugih m_b, tako da tak b pride na začetek našega urejenega seznama. Najceneje torej bo, če takrat b iz urejenega seznama pobrišemo in ga vrinemo na začetek. V ta namen je koristno imeti dvojno povezan seznam (*doubly linked list*), obenem pa še tabelo, ki za vsak b vsebuje kazalec na tisti element seznama, ki vsebuje vrednost b. Pri vsaki spremembi kakšne od vrednosti m_b lahko torej zdaj v $O(1)$ časa popravimo seznam b-jev, da bo še vedno urejen naraščajoče po m_b. V seznamu zdaj ne bodo le tisti b-ji, ki so v x_i prižgani, ampak kar vsi b-ji od 0 do B, mi pa bomo nato seveda za popravljanje s-ja uporabili le tiste b-je, pri katerih je bit b v a_i prižgan. Zdaj porabi naša rešitev le $O(nB)$ časa (in $O(B)$ prostora za vrednosti m_b in urejeni seznam b-jev). Oglejmo si še primer implementacije v jeziku C++:

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

int VsotaKvadratov(const vector<int>& x)
{
   // Poiščimo največjo vrednost v seznamu in določimo B.
   int n = x.size(), xMax = 0;
   for (int xi : x) if (xi > xMax) xMax = xi;
   int B = 0; for (int X = xMax; X > 0; X >>= 1) B++;

   // Pripravimo seznam, v katerem so b-ji urejeni naraščajoče po m_b.
   struct Clen { int prej, nasl, mb; };
   vector<Clen> C(B + 1);
   for (int b = 0; b <= B; b++)
      C[b] = { b − 1, b + 1, n };
   C[B].nasl = −1;
   int prvi = 0, s = 0;

   // Glavna zanka pregleduje seznam x po padajočih indeksih.
   for (int i = n − 1; i >= 0; i−−)
   {
      // Popravimo vrednosti m_b pri tistih b, kjer je bit b v x[i + 1] ugasnjen.
      if (i < n − 1) for (int b = 0; b <= B; b++)
      {
         if ((x[i + 1] >> b) & 1) continue;
         C[b].mb = i + 1; if (b == prvi) continue;

         // Če b še ni bil na začetku seznama, ga moramo premakniti tja.
         int prej = C[b].prej, nasl = C[b].nasl;
         C[b].prej = −1; C[b].nasl = prvi;
         C[prvi].prej = b; prvi = b;
         C[prej].nasl = nasl; if (nasl >= 0) C[nasl].prej = prej;
      }
      // Prištejmo k s vsote kvadratov števil a_ij za trenutni i in vse možne j.
      for (int a = x[i], j = i, b = prvi; b >= 0; b = C[b].nasl) if ((x[i] >> b) & 1)
      {
         // Vrednosti a_ij od trenutnega j do C[b].mb − 1 so vse enake a.
         // Prištejmo njihove kvadrate k s.
         s += a * a * (C[b].mb − j);
```

```
      // Prestavimo se z j na C[b].mb in izračunajmo novo vrednost a_ij pri njem.
      j = C[b].mb;
      a &= (j < n) ? x[j] : 0;
    }
  }
  return s;
}
```

Razmislimo zdaj še o različici naloge, pri kateri namesto vsote $\sum_{i,j} a_{ij}^2$ zahtevamo vsoto $\sum_{i,j} a_{ij}$. Seveda bi jo lahko računali z enakim postopkom kot doslej, le da bi pri prištevanju k s namesto a^2 uporabili a. Vendar pa lahko zdaj do rešitve s časovno zahtevnostjo $O(nB)$ pridemo tudi na preprostejši način.

Kot vsako nenegativno celo število si tudi vrednost a_{ij} lahko predstavljamo kot vsoto nekaj potenc števila 2 — namreč vrednosti 2^b za tiste b, pri katerih je bit b v dvojiškem zapisu števila a_{ij} prižgan. Vsota, po kateri sprašuje naloga, je torej

$$\begin{aligned} s &= \textstyle\sum_i \sum_j a_{ij} \\ &= \textstyle\sum_i \sum_j \sum_b 2^b \, [\![\text{bit } b \text{ je v } a_{ij} \text{ prižgan}]\!] \\ &= \textstyle\sum_i \sum_b 2^b \sum_j [\![\text{bit } b \text{ je v } a_{ij} \text{ prižgan}]\!]. \end{aligned}$$

(Pri tem $[\![\ldots]\!]$ pomeni vrednost 1, če je pogoj v oklepajih izpolnjen, sicer pa 0.) Če pri fiksnem i gledamo vrednosti a_{ij} po naraščajočih j, jih ima prvih nekaj bit b mogoče še prižgan, ko pa se ta bit enkrat ugasne, bo ugasnjen tudi ostal. Vsota po j v zadnji vrstici gornje formule nam torej ne pove nič drugega kot to, kako daleč naprej od i moramo iti po vhodnem seznamu, preden se bit b ugasne (torej: preden prvič ne naletimo na takšno število x_j, ki ima bit b ugasnjen). Te reči pa smo računali že v rešitvi prvotne različice naloge, kjer smo jih označili z m_b. Vemo torej, da če je bil bit b prižgan v x_i (in s tem v a_{ii}), bo prižgan tudi v a_{ij} do $j = m_b - 1$, od $j = m_b$ naprej pa bo ugasnjen. Vrednost 2^b moramo torej k vsoti šteti $(m_b - i)$-krat. Tako smo dobili naslednjo rešitev:

> $B := \lfloor \log_2 \max\{x_1, \ldots, x_n\} \rfloor$;
> $s := 0$; **for** $b := 0$ **to** B **do** $m_b := n + 1$;
> **for** $i := n$ **downto** 1:
>   **for** $b := 0$ **to** B **do**
>   **if** je bit b v x_{i+1} prižgan **then** $m_b := i + 1$;
>   **if** je bit b v x_i prižgan **then** $s := s + (m_b - i) \cdot 2^b$;
> izpiši s;

Razlika med prvotno in tole lažjo različico naloge je torej v tem, da če nas zanima le vsota števil a_{ij}, je dogajanje na različnih bitih neodvisno, zato je vseeno, v kakšnem vrstnem redu se ugašajo (če pri fiksnem i povečujemo j); če pa moramo števila a_{ij} pred seštevanjem še kvadrirati, začnejo različni biti vplivati drug na drugega in ni vseeno, kateri se ugasnejo prej in kateri kasneje.

Še primer implementacije v C++:

```
int Vsota(const vector<int>& x)
{
  // Poiščimo največjo vrednost v seznamu in določimo B.
  int n = x.size(), xMax = 0, s = 0;
```

```
  for (int xi : x) if (xi > xMax) xMax = xi;
  int B = 0; for (int X = xMax; X > 0; X >>= 1) B++;

  // Pripravimo začetno stanje tabele m_b.
  vector<int> mb(B + 1, n);

  // Glavna zanka pregleduje seznam x po padajočih indeksih in nato po bitih.
  for (int i = n − 1; i >= 0; i−−) for (int b = 0; b <= B; b++)
  {
    // Popravimo vrednosti m_b pri tistih b, kjer je bit b v x[i + 1] ugasnjen.
    if (i < n − 1 && ((x[i + 1] >> b) & 1) == 0) mb[b] = i + 1;

    // Če je b v x[i] prižgan, je 2^b prisotna v a_ij za i ≤ j < mb[b],
    // (kasneje pa ne več), zato ga tolikokrat prištejmo k vsoti.
    if ((x[i] >> b) & 1) s += (mb[b] − i) << b;
  }

  return s;
}
```

31. Domine

Mislimo si graf, ki ima za vsako domino po eno točko; povezava $u \to v$ pa naj obstaja takrat, ko je nek neprazen začetek (prefiks) v-ja enak nekemu koncu (sufiksu) u-ja. Če je takih ujemanj med koncem u-ja in začetkom v-ja več, vzemimo najkrajše med njimi, nato pa za dolžino povezave d_{uv} vzemimo razliko med dolžino v-ja in dolžino tega najkrajšega ujemanja. To nam pove, da če bi v kačo iz domin za u postavili v, se lahko kača podaljša za d_{uv} znakov, več pa ne. Dodajmo še posebno točko s, ki predstavlja začetek kače; iz nje speljimo povezavo do vsake domine u, dolžina d_{su} pa naj bo enaka dolžini niza na domini u.

Vsaki kači iz domin zdaj ustreza neka taka pot v grafu, ki se začne v s in nobene točke ne obišče več kot enkrat; dolžina te poti (torej vsota dolžin povezav, ki jo tvorijo) pa je ravno enaka dolžini kače. Problem najdaljše kače iz domin smo tako prevedli na problem najdaljše poti v grafu. Ta problem smo že srečali pri nalogi s požrešnim taksistom (str. 45, rešitev na str. 134), le da smo imeli tam predpisano tudi to, kje se mora pot končati, tu pa je za nas vseeno, s katero domino se kača konča. Tako kot tam lahko tudi tu uporabimo rešitev z rekurzijo ali pa z dinamičnim programiranjem; podrobnosti ne bomo opisovali še enkrat, saj smo si jih že ogledali pri rešitvi tiste naloge.

Razmislimo še o tem, kako učinkovito pripraviti graf. Za vsak par besed u in v nas bo zanimalo, kateri je najkrajši konec u-ja, ki je enak nekemu začetku v-ja. Preprosta in učinkovita možnost je, da si pomagamo z Rabinovimi razprševalnimi kodami (*rolling hash*). Navajeni smo že, da so v računalniku znaki v nizih v resnici predstavljeni kot majhna cela števila (po standardih, kot so ASCII, Unicode in podobni). Niz s si torej lahko predstavljamo kot zaporedje števil $s_1 s_2 \ldots s_n$; za njegovo razprševalno kodo pa vzemimo $h(s) := \sum_{i=1}^{n} s_i b^{i-1} \bmod m$, pri čemer sta b in m neki konstanti. Lepo pri teh kodah je, da jih je zelo lahko dopolniti, če nizu na začetku ali na koncu pritaknemo po en znak: $h(as) = (a + b \cdot h(s)) \bmod m$ in $h(sa) = (h(s) + a \cdot b^n) \bmod m$. Tako lahko v času, ki je sorazmeren z vsoto dolžin vseh nizov na naših dominah, izračunamo razprševalne kode vseh njihovih prefiksov in sufiksov. Kode prefiksov dodajmo v razpršeno tabelo (*hash table*); pravzaprav imejmo več takih tabel, po eno za vsako dolžino prefiksov; nato pa za vsako kodo

kakšnega sufiksa poglejmo, če se taka koda pojavlja v razpršeni tabeli s kodami prefiksov enake dolžine. Če se, to pomeni, da se ta sufiks mogoče ujema s tistim prefiksom; to preverimo s primerjavo znak po znak in če se res ujemata, dodajmo ustrezno povezavo v graf (če je med tema dvema dominama še ni).

32. Robotek

Mislimo si poljubno vozlišče u na globini n v drevesu (koren je na globini 0, njegova otroka na globini 1 in tako naprej) in nekega njegovega potomca v na globini m. Pot od korena do u in pot od korena do v se v prvih n korakih seveda ujemata (saj pot od korena do v vodi skozi v-jeve prednike, med drugim tudi u). Zato je $f(v) = \sum_{k=1}^{m} v_k/2^k = \sum_{k=1}^{n} u_k/2^k + \sum_{k=n+1}^{m} v_k/2^k = f(u) + (1/2^n) \sum_{k=1}^{m-n} v_{n+k}/2^k$. Zadnja vsota je najmanjša takrat, ko so $v_{n+1}, \ldots, v_m$ vsi enaki 0, torej ko je v najbolj levi potomec u-ja na globini m; takrat je vsota 0 in zato $f(v) = f(u)$; največja pa je ta vsota takrat, ko so $v_{n+1}, \ldots, v_m$ vsi enaki 1, torej ko je v najbolj desni potomec u-ja na globini m; takrat je vsota enaka $1 - 1/2^{m-n}$ in zato $f(v) = f(u) + 1/2^n - 1/2^m$. Ko gre $m \to \infty$ (spomnimo se, da je drevo neskončno globoko), se lahko vrednosti $f(v)$ poljubno približajo $f(u) + 1/2^n$, čisto dosežejo pa je nikoli. Zaključimo lahko torej, da se v u-jevem poddrevesu pojavljajo vozlišča s števili z intervala $[f(u), f(u) + 1/2^n)$. Pri tem vozlišča v u-jevem levem poddrevesu pokrijejo levo polovico tega intervala, tista v desnem pa desno polovico.

(a) Pri opisanem postopku se bo robot ustavil le, če je a oblike $c/2^m$ za nek celoštevilski c (in m); tedaj bo tudi uspešno našel neko primerno vozlišče. Pri kakršnem koli drugačnem a pa se bo spuščal vse globlje v drevo, ne da bi našel kakšno primerno vozlišče (četudi taka vozlišča obstajajo).

Oglejmo si najprej primer, ko a ni oblike $c/2^m$. Z indukcijo se bomo prepričali, da v vsakem trenutku velja: če je robot v vozlišču u na globini n, potem leži a strogo v notranjosti intervala $[f(u), f(u) + 1/2^n)$, ki ga pokriva vozlišče u. (1) Na začetku, v korenu drevesa, to očitno drži, saj je takrat $f(u) = 0$ in $n = 1$, število a pa res leži na $(0, 1)$ — gotovo namreč a ni enak 0, saj smo rekli, da ni oblike $c/2^m$. (2) Recimo zdaj, da trditev drži v nekem vozlišču u, in poglejmo, kaj se zgodi pri premiku dol po drevesu. Naloga pravi, da se robot premakne v u-jevega levega otroka, če v u-jevem levem poddrevesu leži kakšno primerno vozlišče (tako, čigar število je na $[a, b)$). Pišimo $U = f(u)$. Vozlišča v levem poddrevesu pokrivajo interval $[U, U + 1/2^{n+1})$, tista v desnem pa interval $[U + 1/2^{n+1}, U + 1/2^n)$. Če je torej $a \geq U + 1/2^{n+1}$, mora iti robot v desnega otroka, ker v levem poddrevesu gotovo ni primernih vozlišč, sicer pa bo šel v levega. (2.1) Če je šel robot v levega otroka: velja torej $a < U + 1/2^{n+1}$, iz induktivne predpostavke pa sledi tudi $a > U$, torej a zdaj leži strogo v notranjosti intervala, ki ga pokriva levi otrok, torej vozlišče, v katerega se je robot pravkar premaknil. (2.2) Če pa je šel robot v desnega otroka: velja torej $a \geq U + 1/2^{n+1}$, toda tu gotovo ne velja enakost, saj vemo, da a ni oblike $c/2^m$. Torej je $a > U + 1/2^{n+1}$; in ker iz induktivne predpostavke sledi tudi $a < U + 1/2^n$, lahko zaključimo, da a leži strogo v notranjosti intervala, ki ga pokriva desni otrok, torej vozlišče, v katerega se je robot pravkar premaknil.

Gornji induktivni razmislek nam torej pove, da robot po vsakem premiku stoji na vozlišču u, za katerega je $f(u) < a$, torej to vozlišče ne ustreza zahtevi naloge — za to bi moralo veljati $a \leq f(u) < b$. Robot se torej ne sme ustaviti, ampak bo

moral narediti še en korak. Takrat bo v novem vozlišču imel enako težavo in tako naprej v nedogled.

Oglejmo si zdaj še primer, ko je a oblike $c/2^m$. Med več možnimi takšnimi izražavami vzemimo tisto z najmanjšim m; takrat je torej c lih. Zapišimo c v dvojiškem zapisu kot $c_1 c_2 \ldots c_m$, tako da je $a = \sum_{k=1}^{m} c_k/2^k$. Z indukcijo se lahko prepričamo, da biti $c_1, \ldots, c_m$ zdaj opisujejo ravno tisto pot, po kateri se bo premikal robot; z drugimi besedami: po n korakih je robot v vozlišču z vrednostjo $\sum_{k=1}^{n} c_k/2^k$.

Pri $k = 0$ je robot v korenu, ki ima vrednost 0, torej trditev drži. Recimo zdaj, da trditev drži pri n (za nek $n < m$) in poglejmo, kaj se zgodi pri $n + 1$. Robot je zaenkrat v vozlišču z vrednostjo $U := \sum_{k=1}^{n} c_k/2^k$. Ker je $k < m$, manjka vsoti v U še $m - n$ členov, da bi postala enaka a; in vsaj zadnji od teh členov je gotovo neničeln (ker je c lih in zato $c_m = 1$). Torej je $U < a$ in robot se še ne bo ustavil. Da se odloči, v katerega otroka se naj premakne, bo pogledal vrednost $U + 1/2^{n+1}$. (1) Če je $c_{n+1} = 0$, je $a = U + \sum_{k=n+2}^{m} c_k/2^k < U + 1/2^{n+1}$, torej se bo robot odločil za premik v levega otroka. (2) Če je $c_{n+1} = 1$, je $a \geq U + 1/2^{n+1}$, torej se bo robot odločil za premik v desnega otroka. — V obeh primerih se torej robot odloči za natančno tak premik, kot ga naša trditev napoveduje iz vrednosti c_{n+1}, torej naša trditev drži tudi pri $n + 1$.

Ta induktivni razmislek nam pove, da robot po m korakih doseže vozlišče z vrednostjo a, ta pa seveda leži na $[a, b)$, zato se robot takrat ustavi. V primerih torej, ko je a oblike $c/2^n$, robot deluje pravilno.

(*b*) V n korakih lahko dosežemo poljubno vozlišče na globini n, ta pa imajo vrednosti oblike $c/2^n$ za $c = 0, \ldots, 2^n - 1$. Vprašanje je torej, kateri je najmanjši n, za katerega leži kakšno tako število na intervalu $[a, b)$. Označimo dolžino tega intervala z $d := b - a$. Če vzamemo dovolj velik n, bo $1/2^n$ gotovo $\leq d$: najmanjši primerni n je $n := \lceil -\log_2 d \rceil$. Zanj je torej $1/2^n \leq d < 1/2^{n-1}$. Ker je interval $[a, b)$ dolg vsaj $1/2^n$, leži na njem gotovo vsaj eno število oblike $c/2^n$; mogoče celo dve, gotovo pa ne tri, kajti če bi na njem poleg $c/2^n$ in $(c + 1)/2^n$ ležalo tudi $(c + 2)/2^n$, bi bila dolžina tega intervala že vsaj $2/2^n = 1/2^{n-1}$, mi pa smo videli, da je $d < 1/2^{n-1}$.

Da bo $c/2^n$ ležal na intervalu $[a, b)$, mora veljati $a \cdot 2^n \leq c < b \cdot 2^n$; najmanjši primerni c je torej $c_1 := \lceil a \cdot 2^n \rceil$, največji pa $c_2 := \lceil b \cdot 2^n \rceil - 1$. Tako smo našli eno ali dve primerni vozlišči na globini n (recimo jima u_1 in u_2), vprašanje pa je, ali obstaja tudi kako primerno vozlišče više v drevesu, bliže korenu.

Za začetek opazimo, da če že obstaja kakšno primerno vozlišče na globini $n - 1$, je gotovo eno samo, kajti če bi bili dve, bi se njuni vrednosti razlikovali za vsaj 2^{n-1}, kar je že več od dolžine našega intervala.

Če je c_1 sod, ima oče vozlišča u_1 enako vrednost kot vozlišče u_1 sámo in torej tudi leži na $[a, b)$. Če pa je c_1 lih, ima oče vozlišča u_1 vrednost $(c_1 - 1)/2^n$, kar je (zaradi načina, kako smo izbrali c_1) že manj kot a in torej ne pride v poštev. Tedaj moramo preveriti še naslednje vozlišče na globini $n - 1$; to ima vrednost $(c_1 + 1)/2^n$, ki mogoče leži na $[a, b)$ in je torej primerno za naš namen (to se zgodi v primeru, če je $c_2 = c_1 + 1$ in opazovano vozlišče na globini $n - 1$ je tedaj oče vozlišča u_2), mogoče pa je njegova vrednost že $\geq b$ in ni primerno za naš namen (to se zgodi v primeru, če je $c_2 = c_1$).

Zdaj smo torej ugotovili, ali obstaja na globini $n - 1$ kakšno primerno vozlišče in

katero je to (vemo tudi, da je edino takšno). Recimo, da je to vozlišče u_3 z vrednostjo $c_3/2^{n-1}$. Če je c_3 lih, ima oče vozlišča u_3 vrednost $(c_3 - 1)/2^{n-1}$, naslednje vozlišče na globini $n-1$ pa vrednost $(c_3+1)/2^{n-1}$. Oboje je oddaljeno od $c_3/2^{n-1}$ za $1/2^{n-1}$, kar je $\geq d$, torej tidve vozlišči gotovo ležita zunaj intervala $[a, b)$; tedaj torej lahko zaključimo, da na globini $n - 2$ ali nad njo ni nobenega primernega vozlišča in je najboljša rešitev vozlišče u_3, ki ga lahko robot iz korena doseže z $n - 1$ premiki. Če pa je c_3 sod, ima oče vozlišča u_3 enako vrednost kot u_3 sámo in lahko z enakim razmislekom nadaljujemo pri njem.

Zapišimo dobljeni postopek še s psevdokodo:

$n := -\lfloor \log_2(b - a) \rfloor$;
$c := \lceil a \cdot 2^n \rceil$;
if c je lih **and** $(c + 1)/2^n < b$ **then** $c := c + 1$;
while c je sod **do** $c := c/2$; $n := n - 1$;
return n; (* *globina, na kateri leži najvišje primerno vozlišče* *)

Naloge so sestavili: Wikipedija (I in II), praljudje — Nino Bašić; tipkanje, konjunkcije — Tomaž Hočevar; bankomat, poskočno besedilo, robotek — Vid Kocijan; žaga, domine — Jurij Kodre; kakuro, brisanje nezaklenjenih celic — Mitja Lasič; Evklidov algoritem, požrešni taksist — Matjaž Leonardis; tetris — Borut Lesjak; žaba, stave, iskanje števila — Matija Lokar; srečna števila, izštevanka, temperature, zakon prve števke, kontrolna vsota — Mark Martinec; sudoku — Polona Novak; vsota zmnožkov — Polona Novak in Matija Lokar; torta, davki, prepisovanje — Jure Slak; iskanje zlata, RGB-šahovnica, sladkosnedi osel — Patrik Zajec; brisanje duplikatov — 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. Avtobus

- Rešitvam, ki po nepotrebnem računajo vsoto $d_1 + \ldots + d_i$ vsakič od začetka, namesto da bi le prištele trenutni d_i k prejšnji vsoti, naj se zaradi tega odšteje tri točke.

- Naloga pravi, da lahko SMS pošljemo 1 sekundo po tistem, ko natipkamo zadnji znak sporočila. Rešitvam, ki pomotoma predpostavijo, da ga lahko pošljemo takoj, ko natipkamo zadnji znak (in ne šele 1 sekundo kasneje), naj se zaradi tega odšteje dve točki.

- Za morebitne manjše napake pri branju vhodnih podatkov naj se rešitvi odšteje največ dve točki.

- Če rešitev zahtevanega niza ne izpiše, pač pa ga le vrne kot rezultat neke funkcije, naj se ji zaradi tega ne odšteva točk.

- Rešitev sme predpostaviti, da je dolžina posameznega niza p_i krajša od neke razumne zgornje meje (npr. nekaj deset znakov).

2. Jedilnik

- Rešitve, ki imajo časovno zahtevnost $O(n^2)$, lahko dobijo največ 15 točk. Rešitve, ki imajo časovno zahtevnost $O(n \log n)$ (npr. ker uporabljajo slovar, implementiran z drevesom, ali pa ker urejajo seznam parov $\langle L[i], i \rangle$), lahko dobijo največ 18 točk.

- Pri tej nalogi so jedi predstavljene z nizi, vendar ni mišljeno, da bi bil velik poudarek na delu z nizi. Rešitev sme predpostaviti, da so ti nizi krajši od neke razumne zgornje meje (npr. nekaj deset znakov). Ne sme pa predpostaviti, da je nabor vseh možnih jedi znan vnaprej, npr. da lahko v vhodnih podatkih nastopajo le tisti nizi, kakršne vidimo v primerih v besedilu naloge.

- Naloga večkrat poudari, da se jedilnik ciklično ponavlja. Rešitvi, ki tega ne upošteva in npr. pravilno obdela le tiste jedi, ki se naslednjič pojavijo že v istem ciklu (ne pa šele v naslednjem), naj se zaradi tega odšteje 8 točk.

3. Trikotniki

- Naloga pravi, naj bo postopek čim bolj učinkovit. Rešitve, pri katerih je časovna zahtevnost posamezne poizvedbe $O(n^2)$, lahko dobijo največ 10 točk. Rešitve, ki imajo časovno zahtevnost poizvedbe strogo manj kot $O(n^2)$, lahko dobijo vse točke.

- Rešitev sme brez škode predpostaviti, da velja $0 \leq x < n$, $0 \leq y < n$ in $0 < v$, torej da so vhodni podatki smiselni.

- Lahko se zgodi, da kakšen del poizvedovalnega območja leži zunaj mreže. Če poskuša rešitev v takih primerih dostopati do neobstoječih celic tabele, naj se ji zaradi tega odšteje 3 točke.
- Rešitev, ki bi npr. predpostavila, da je n vedno 8 ali pa v vedno 4, ker je slučajno tako v primeru v besedilu naloge, naj dobi največ 5 točk.
- Naloga pravi, da mora rešitev za morebitno predpripravo podatkov pred prvo poizvedbo porabiti strogo manj kot $O(n^3)$ časa. Rešitve, ki porabijo $O(n^3)$ ali še več časa, lahko dobijo največ 10 točk.

4. Točke in koši

- Naloga pravi, naj bo rešitev čim bolj učinkovita. Rešitve z eksponentno zahtevnostjo (npr. preprosta rekurzivna rešitev brez pomnjenja že izračunanih rezultatov) lahko dobijo največ 12 točk. Rešitve s polinomsko časovno zahtevnostjo lahko dobijo vse točke.
- Pri rešitvah s polinomsko časovno zahtevnostjo je vseeno, ali rešitev porabi le konstantno mnogo pomnilnika (kot npr. funkcija `Kosi2` iz našega primera rešitve, ki hrani med izračunom le prejšnje tri vrednosti funkcije f) ali pa kar $O(n)$ pomnilnika (npr. ker bi si shranjevala vse doslej izračunane vrednosti funkcije f).
- Za morebitno čudno obnašanje pri $m < 0$ naj se rešitvam ne odšteva točk. Za morebitne napake pri majhnih nenegativnih m, npr. $m = 0, 1, 2$, naj se rešitvi odšteje tri točke.
- Vrednost $f(m)$, ki jo mora funkcija vrniti, narašča eksponentno hitro v odvisnosti od m, torej kmalu postane prevelika za običajne celoštevilske tipe (npr. 32-bitni `int`, kot ga ponavadi najdemo v C-ju in podobnih jezikih). Pri tej nalogi ni mišljeno, da bi se morala tekmovalčeva rešitev kaj ukvarjati s to težavo (npr. uporabiti arbitrary-precision aritmetiko) ali se je sploh kakorkoli zavedati.

5. Povprečni položaj znaka

- Objavili smo dve rešitvi: eno, ki bere vhodno besedilo po znakih, in eno, ki ga bere po vrsticah. Oboje je enako dobro. Prav tako je enako dobro tudi, če rešitev prebere celo vhodno besedilo naenkrat v pomnilnik. Rešitev sme tudi predpostaviti neko zgornjo mejo za dolžino posamezne vrstice (npr. da je vrstica dolga največ 100 znakov ali kaj podobnega).
- Pri tej nalogi ni mišljeno, da bi bil poudarek na branju vhodnih podatkov. Za drobne napake pri branju vhodnega besedila naj se rešitvi odšteje največ 2 točki. (Primer take drobne napake bi na primer bil, če bi v naši prvi rešitvi, torej tisti, ki bere po znakih, pozabili na pogoj `|| (ch == EOF && x > 0)`, ki skrbi za to, da pravilno obdelamo zadnjo vrstico tudi tedaj, če se ne konča na znak za konec vrstice.)

- Če rešitev vhodnega besedila sploh ne bere, ampak predpostavi, da ga kar dobi v nekem seznamu ali tabeli, naj se ji zaradi tega odšteje največ tri točke.

- Format izpisa pri tej nalogi ni posebej predpisan, pomembno je le, da so v njem med drugim vsi podatki, ki jih naloga zahteva, torej število pojavitev znaka in povprečni položaj (zaokrožen navzdol na celo število). Nič ni narobe, če rešitev poleg tega izpiše še kaj drugega (npr. povprečni položaj kot realno število). Zaradi morebitnih napak pri zaokrožanju povprečnega položaja na celo število naj se rešitvi odšteje največ dve točki.

- Če rešitev zaradi kakšnega deljenja z 0 (npr. če se iskani znak v neki vrstici sploh ne pojavlja) daje napačne rezultate ali pa se celo sesuje, naj se ji zaradi tega odšteje štiri točke.

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. Avtobus	lažja naloga v prvi skupini
2. Jedilnik	srednje težka naloga v prvi skupini
3. Trikotniki	srednja do težja naloga v drugi skupini[11]
4. Točke in koši	težka naloga v prvi ali lažja do srednja v drugi skupini
5. Povp. položaj	težja naloga v prvi ali lažja 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.

Zadnja leta na državnem tekmovanju opažamo, da je v prvi skupini izrazito veliko tekmovalcev v primerjavi z drugo in tretjo, med njimi pa je tudi veliko takih z zelo dobrimi rezultati, ki bi prav lahko tekmovali tudi v kakšni težji skupini. Mentorjem zato priporočamo, naj tekmovalce, če se jim zdi to primerno, spodbudijo k udeležbi v zahtevnejših skupinah.

[11] Opomba: težavnost te naloge je zelo odvisna od tega, kako točkujemo rešitve odvisno od njihove učinkovitosti. Na primer, če bi dali vse točke že za rešitev, kjer traja vsaka poizvedba $O(v^2)$ časa, bi bila lahko to lažja naloga za prvo skupino. Če bi za vse točke zahtevali rešitev s poizvedbami v času $O(1)$, bi bila to težka naloga tudi za tretjo skupino. V navodilih za letošnje šolsko tekmovanje smo ubrali srednjo pot: za poizvedbe v času $O(v^2)$ se dobi polovico točk, za karkoli hitrejšega od tega pa vse točke.

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 eno prvo in tri tretje nagrade, v tretji skupini pa eno drugo in 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 364) in smo jih letos podelili dvanajst. Srebrna priznanja pa se podeljujejo po podobnih kriterijih kot pred leti pohvale; prejmejo jih tekmovalci, ki ustrezajo naslednjim trem pogojem: (1) tekmovalec ni dobil zlatega priznanja; (2) je boljši od vsaj polovice tekmovalcev v svoji skupini; in (3) je tekmoval v prvi ali drugi skupini in dobil vsaj 20 točk ali pa je tekmoval v tretji skupini in dobil vsaj 80 točk. Namen srebrnih priznanj je, da izkažemo priznanje in spodbudo vsem, ki se po rezultatu prebijejo v zgornjo polovico svoje skupine. Podobno prakso poznajo tudi na nekaterih mednarodnih tekmovanjih; na primer, na mednarodni računalniški olimpijadi (IOI) prejme medalje kar polovica vseh udeležencev. Poleg zlatih in srebrnih priznanj obstajajo tudi bronasta, ta pa so dobili najboljši tekmovalci v okviru šolskih tekmovanj (letos smo podelili 131 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	Job Petrovčič	2	Gimnazija Vič	20	18	20	14	20	92
1Z	2	Miha Zupan	3	Gimnazija Bežigrad	20	15	19	19	17	90
2Z	3	Janez Koprivec	2	Gimnazija Vič	20	20	16	16	16	88
2Z	4	Miha Frangež	3	SERŠ Maribor	19	20	17	12	19	87
3S	5	Leon Samotorčan	4	Gimnazija Bežigrad	19	12	18	20	17	86
3S	6	Matej Pevec	3	Vegova Ljubljana	19	20	16	10	20	85
3S		Gregor Bučar	3	ŠC N. mesto, SEŠTG	18	19	20	8	20	85
S	8	Rok Štular	1	Gimnazija Bežigrad	20	20	16	12	16	84
S	9	Žan Žnidar	2	Gimnazija Kranj	20	20	11	12	20	83
S		Jan Zorko	3	ŠC Celje, SŠ za KER	20	20	13	10	20	83
S		Jonas Lasan	3	ŠC Kranj, Str. gimn.	20	20	14	11	18	83
S	12	Aljaž Medič	2	ŠC Kranj, Str. gimn.	19	20	19	12	12	82
S		Luka Železnik	3	II. gimnazija Maribor	18	20	13	19	12	82
S	14	Peter Kosem	3	Gimnazija Vič	20	15	18	10	18	81
S	15	Martin Prelog	4	ŠC Kranj, STŠ Kranj	20	17	16	10	17	80
S	16	Rok Šerak	4	STPŠ Trbovlje	20	20	18	8	12	78
S		Luka Peršolja	1	Gimnazija Vič	19	18	18	11	12	78
S	18	Mateja Žvegler	3	ŠC Celje, SŠ za KER	18	20	16	20	3	77
S	19	Gašper Pistotnik	3	ŠC Celje, Gimn. Lava	20	18	14	8	16	76
S	20	Luka Skeledžija	2	Gimnazija Vič	17	20	11	9	18	75
S	21	Gašper Irman	3	ŠC Velenje, ERŠ	15	15	16	14	14	74
S	22	Jakob Ocepek	2	Gimnazija Vič	18	19	12	12	12	73
S	23	Luka Laharnar	4	STPŠ Trbovlje	19	18	16	7	12	72
S	24	Luka Gole	3	ŠC N. mesto, SEŠTG	20	5	18	10	18	71
S		Andrej Sušnik	4	Vegova Ljubljana	19	19	14	7	12	71
S		Aleš Erjavec	3	ŠC N. mesto, SEŠTG	20	15	10	6	20	71
S	27	Jan Hribar	3	Vegova Ljubljana	19	20	15	5	11	70
S	28	Miha Meglič	2	ŠC Kranj, Str. gimn.	20	20	0	9	20	69
S		Staš Horvat	2	II. gimnazija Maribor	18	18	16	12	5	69
S		Jernej Domajnko	4	II. gimnazija Maribor	20	20	17	2	10	69
S		Filip Štamcar	8	ZRI	19	15	10	10	15	69
S		Aleksander Piciga	2	Gimnazija Vič	20	18	9	2	20	69
S		Luka Maček	3	Vegova Ljubljana	18	20	8	15	8	69
S	34	Nik Tomažič	2	SERŠ Maribor	19	20	12	6	11	68
S		Vid Kreča	3	ŠC Celje, SŠ za KER	19	20	13	11	5	68
S	36	Jaka Vrhovec	5	SŠ teh. strok Šiška	19	18	15	5	10	67
S		Blaž Košir	3	Gimnazija Vič	17	16	14	12	8	67
S		Lucijan Semprimožnik	3	I. gimnazija v Celju	18	15	18	6	10	67
S	39	Urban Žiberna	3	Vegova Ljubljana	18	5	15	7	19	64
S		Tadej Kraševec	4	ŠC N. mesto, SEŠTG	17	18	0	9	20	64
S		Klemen Napast	4	SERŠ Maribor	13	20	18	2	11	64
S	42	Tadej Strah	2	Gimnazija Vič	18	8	8	11	18	63

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
S	43	Jaka Šivavec	3	Gim. Bežigrad, medn. šola	18	0	20	12	12	62
S		Miha Marinko	3	Gimnazija Vič	18	3	16	20	5	62
S	45	Jan Hrastnik	2	Gimnazija Vič	17	19	12	6	7	61
S		Urh Robič	2	Škof. klas. gimn. Lj.	18	17	0	11	15	61
S		Matic Koračin	3	ŠC Novo mesto, SEŠTG	18	10	10	15	8	61
S		Frenk Dragar	4	I. gimnazija v Celju	17	20	10	6	8	61
	49	Matevž Rom	4	ŠC Novo mesto, SEŠTG	17	15	16	6	5	59
		Andraž Pevcin	3	ŠC Celje, Gimn. Lava	20	15	1	8	15	59
	51	Iztok Bajcar	2	Gimnazija Vič	5	20	11	11	10	57
	52	Žan Koren Kern	3	STPŠ Trbovlje	10	20	7	9	9	55
	53	Matic Hrastelj	4	STPŠ Trbovlje	16	17	6	8	7	54
	54	Daniel Blažič	1	Gimnazija Vič	18	12	1	10	12	53
		Jakob Salmič	4	ŠC Kranj, STŠ Kranj	5	20	13	10	5	53
		Juš Nirtič	3	Gimnazija Bežigrad	5	0	19	19	10	53
	57	Sandi Pečečnik	3	ŠC Velenje, ERŠ	18	0	8	6	18	50
	58	Jure Pustoslemšek	4	ŠC Celje, SŠ za KER	15	12	15	0	5	47
		Mark Mihelič	1	Gimnazija Bežigrad	17	5	12	8	5	47
	60	Anže Vidmar	2	STPŠ Trbovlje	20	8	2	15	1	46
	61	Jan Alojz Gačnik	4	SERŠ Maribor	19	8	8	8	2	45
	62	Marko Kovačič	4	SŠ Domžale, PSŠ	10	18	10	0	5	43
		Tadej Zorman	5	SŠ tehniških strok Šiška	5	15	3	8	12	43
	64	Žan Ostrožnik	4	STPŠ Trbovlje	15	5	14	3	5	42
		Klemen Šuštar	2	STPŠ Trbovlje	18	12	8	3	1	42
	66	Benjamin Steiner	5	SŠ tehniških strok Šiška	16	16	0	6	1	39
		Leon Dolničar	1	ZRI	5	3	7	18	6	39
	68	Blaž Matija Samotorčan	1	Gimnazija Vič	5	5	14	10	3	37
	69	Matic Frol	3	STPŠ Trbovlje	5	5	11	11	3	35
	70	Pia Golob	2	II. gimnazija Maribor	15	8	5	6	0	34
		Sara Veber	4	Gimnazija Poljane	10	5	9	10	0	34
	72	Klemen Hercog	4	SERŠ Maribor	6	5	11	8	2	32
		Igor Kepe	4	SERŠ Maribor	14	0	6	2	10	32
		Jani Suban	4	STŠ Koper	0	8	16	0	8	32
		Žiga Pirc	3	STPŠ Trbovlje	2	19	2	9	0	32
	76	Leon Macur	5	SŠ tehniških strok Šiška	8	0	10	3	10	31
		Luka Kopše	2	II. gimnazija Maribor	15	0	10	3	3	31
	78	Nik Jan Špruk	2	Gimnazija Šiška	10	20	0	0	0	30
		Blaž Dominc	3	ŠC Ptuj, ERŠ	10	8	0	7	5	30
	80	Urban Plaskan	4	Prva gimnazija v Celju	1	15	1	5	6	28
		Žan Rogan	4	SPTŠ Murska Sobota	14	10	0	2	2	28
		Matija Pilko	1	Prva gimnazija v Celju	16	0	0	12	0	28
		Žan Jelen	5	SŠ tehniških strok Šiška	11	0	5	0	12	28

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
84	Luka Laharnar	2	ŠC Velenje, ERŠ	5	10	3	2	3	23
85	Luka Pavčnik	2	ŠC Velenje, ERŠ	10	0	5	2	5	22
	Matej Volkar	4	SŠ Domžale, PSŠ	10	0	12	0	0	22
87	Dominik Petrovčič	3	ŠC Nova Gorica, GZŠ	11	0	10	0	0	21
	Nejc Kolmanko	1	Gimnazija Murska Sobota	3	18	0	0	0	21
	Alen Plavec	2	SPTŠ Murska Sobota	18	0	3	0	0	21
90	Janez Sedeljšak	2	ŠC Velenje, ERŠ	3	3	6	4	4	20
91	Jernej Žuraj	3	ŠC Celje, SŠ za KER	0	18	0	0	0	18
92	Marko Rozman	3	ŠC Kranj, STŠ Kranj	3	0	0	11	2	16
93	Gašper Žnider	5	SŠ tehniških strok Šiška	5	5	0	0	5	15
94	Vito Verdnik	1	II. gimnazija Maribor	13	0	0	0	0	13
95	Erik Toplak	4	SPTŠ Murska Sobota	1	0	1	0	8	10
	Anja Kotnik	2	Gimnazija Šiška	5	0	4	0	1	10
97	Luka Gašparič	3	II. gimnazija Maribor	2	0	1	4	0	7
98	Vid Kranjec	1	Gimnazija Murska Sobota	3	0	0	0	1	4
99	Bojan Medič Marolt	4	SŠ Domžale, PSŠ	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	Jakob Pogačnik Souvent	2	Gimnazija Vič	20	20	17	20	19	96
2Z	2	Matevž Miščič	3	Gimnazija Vič	20	17	13	20	19	89
2Z		Rok Strah	4	Vegova Ljubljana	20	20	17	12	20	89
2Z	4	Patrik Žnidaršič	1	ZRI + Gimnazija Vič	20	20	15	12	20	87
3S	5	Lenart Bučar	2	Gimnazija Bežigrad	19	20	17	11	18	85
3S	6	Urban Mišmaš	3	Vegova Ljubljana	18	18	17	9	18	80
S	7	Janez Ignacij Jereb	1	ZRI	20	20	10	12	16	78
S	8	Miha Pompe	3	ZRI + Gimnazija Vič	18	15	15	11	18	77
S	9	Jakob Zmrzlikar	4	Gimnazija Vič	0	19	17	18	20	74
S	10	Jernej Jezeršek	3	Vegova Ljubljana	20	12	11	12	18	73
S		Urh Primožič	3	Škof. klas. gimn. Lj.	18	18	12	17	8	73
S	12	Blaž Čerenak	9	OŠ Griže	20	20	5	12	15	72
S	13	Miha Krajnc	3	STPŠ Trbovlje	16	14	11	11	18	70
S	14	Tim Retelj	3	Vegova Ljubljana	20	16	8	9	16	69
S	15	Tadej Petrič	4	Vegova Ljubljana	16	14	10	11	16	67
S	16	Luka Pepelnjak	3	Gimnazija Vič	17	12	12	9	16	66
S	17	Matic Conradi	3	I. gimnazija v Celju	12	15	10	11	16	64
S		David Kraševec	4	Vegova Ljubljana	20	8	10	6	20	64
S		Luka Dragar	4	Vegova Ljubljana	16	7	13	8	20	64
S	20	Tadej Logar	3	ŠC Ravne, SŠ Ravne	20	7	3	11	20	61
S		Lan Sevčnikar	1	II. gimnazija Maribor	17	12	12	4	16	61
S		Jon Mikoš	3	Gimnazija Vič	20	12	0	10	19	61
	23	Marko Čmrlec	3	G. Bežigrad, medn. šola	12	14	12	6	16	60
	24	Jaša Žnidar	4	ŠC Kranj, Str. gimn.	14	16	16	9	4	59
	25	Adrijan Rogan	4	Gimn. Murska Sobota	20	5	5	12	16	58
		Gregor Kržmanc	2	Gimnazija Vič	8	13	9	12	16	58
	27	Gregor Kovač	2	ZRI + Vegova Lj.	20	10	3	12	12	57
		Veno Lan Banovšek	4	Vegova Ljubljana	20	12	4	9	12	57
		Kevin Šarlah	4	ŠC Celje, SŠ za KER	13	18	7	11	8	57
	30	David Grabnar	4	Vegova Ljubljana	16	12	10	9	8	55
		Žiga Deutschbauer	4	ŠC Velenje, ERŠ	19	10	4	10	12	55
	32	Samo Debeljak	3	Gimnazija Vič	15	19	0	0	16	50
	33	Žan Bajuk	4	Gimnazija Vič	0	15	5	11	18	49
	34	Domen Vilar	4	ŠC Kranj, Str. gimn.	10	0	12	8	17	47
	35	Vladimir Smrkolj	2	Gimnazija Bežigrad	18	2	5	9	12	46
		Luka Toplak	3	Vegova Ljubljana	10	20	4	12	0	46
	37	Aleksander Georgiev	3	Vegova Ljubljana	17	12	4	10	0	43
	38	Gašper Zgonec	3	Vegova Ljubljana	10	8	4	11	4	37
	39	Domen Kastelic	1	ZRI	16	10	0	6	0	32
		Miha Breznik	4	ŠC Ravne, SŠ Ravne	8	2	3	11	8	32
	41	Blaž Novak	4	ŠC Ravne, SŠ Ravne	17	10	0	0	0	27
	42	Maj Andrejč	2	Gimn. Murska Sobota	7	0	7	12	0	26
	43	Nejc Benkovič	4	Gimn. Murska Sobota	0	0	6	8	0	14
	44	Antonio Žibert	4	ŠC Ravne, SŠ Ravne	0	6	0	1	0	7

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1Z	1	Urban Duh	4	II. gimnazija Maribor	100	25	60	80	80	345
1Z	2	Gregor Kikelj	4	ZRI + ŠC NM, SEŠTG	100	35	60	50	90	335
2Z	3	Tim Poštuvan	4	ZRI	100		60	100	20	280
2Z	4	Jakob Schrader	1	ZRI + Gimn. Vič	70	0	40	60	80	250
3S	5	Blaž Zupančič	4	Škof. klas. gimn. Lj.	0			100	80	180
3S	6	Aljaž Kolar	4	ŠC Kranj, Str. gimn.	60	10	0	20	80	170
S	7	Tevž Lotrič	2	ZRI	60	35	10	0	40	145
S	8	Luka Jevšenak	4	ŠC Velenje, Gimnazija			60		80	140
	9	Bor Grošelj Simić	3	ZRI + Gimn. Vič	100		0	20		120
	10	Luka Govedič	4	II. gimnazija Maribor			20		80	100
		Marko Hostnik	4	Gimnazija Bežigrad	0			20	80	100
	12	Adrian Mladenić Grobelnik	1	ZRI	0		0	10	80	90
	13	Matija Kocbek	2	I. gimnazija v Celju		0		60	0	60
	14	Domen Ramšak	4	ŠC Velenje, ERŠ				20		20
		Tobija Ličen	3	ŠC Nova Gorica, GZŠ	10			10		20
	16	Gregor Brantuša	4	II. gimnazija Maribor	0			0		0
		Yon Ploj	2	ZRI						0

VRSTNI RED ŠOL

Da bi spodbudili šole k čim večji udeležbi in čim boljšim rezultatom v vseh treh skupinah, smo letos prvič objavili tudi vrstni red šol v neke vrste skupnem seštevku. Posamezni šoli prinesejo točke najboljši štirje tekmovalci iz te šole v prvi skupini, najboljši trije v drugi in najboljša dva v tretji skupini. Točke šole so enake vsoti točk njenih tekmovalcev. Točke, ki jih prispeva tekmovalec k vsoti, se izračuna tako, da se delež točk (od vseh možnih točk), ki jih je ta tekmovalec dosegel na tekmovanju, pomnoži z utežjo za skupino, v kateri je tekmoval. Utež za prvo skupino je 100, za drugo skupino 200 in za tretjo skupino 300.

Mesto	Šola	Točke
1	Gimnazija Vič	1105
2	Vegova Ljubljana	779
3	II. gimnazija Maribor	643
4	Gimnazija Bežigrad, Gimnazija	635
5	ŠC Kranj, Strokovna gimnazija	548
6	ŠC Novo mesto, SEŠTG	492
7	STPŠ Trbovlje	399
8	ŠC Celje, SŠ za KER	389
9	I. gimnazija v Celju	348
10	Škofijska klasična gimnazija Ljubljana	315
11	ŠC Velenje, ERŠ	291
12	SERŠ Maribor	264
13	ŠC Ravne na Koroškem, Srednja šola	240
14	Gimnazija Murska Sobota	221
15	Gimnazija Bežigrad, Mednarodna šola	182
16	SŠ tehniških strok Šiška	180
17	ŠC Kranj, STŠ Kranj	149
18	OŠ Griže	144
19	ŠC Celje, Gimnazija Lava	135
20	ŠC Velenje, Gimnazija	84
21	Gimnazija Kranj	83
22	SŠ Domžale, Poklicna in strokovna šola	65
23	SPTŠ Murska Sobota	59
24	Gimnazija Šiška	40
25	Gimnazija Poljane	34
26	ŠC Nova Gorica, Gimnazija in zdr. šola	33
27	STŠ Koper	32
28	ŠC Ptuj, ERŠ	30

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Job Petrovčič	telefon Samsung Galaxy S8 in ovitek
1	1	Miha Zupan	telefon Samsung Galaxy A5 in ovitek
1	2	Janez Koprivec	telefon Samsung Galaxy A5 in ovitek
1	2	Miha Frangež	telefon Samsung Galaxy J7 in ovitek
1	3	Leon Samotorčan	telefon Samsung Galaxy J7 in ovitek
1	3	Matej Pevec	miška Razer Death Adder Elite
1	3	Gregor Bučar	miška Razer Death Adder Elite
2	1	Jakob Pogačnik Souvent	telefon Samsung Galaxy S8 in ovitek
			Dasgupta *et al.: Algorithms*
2	2	Matevž Miščič	telefon Samsung Galaxy A5 in ovitek
			Dasgupta *et al.: Algorithms*
2	2	Rok Strah	telefon Samsung Galaxy A5 in ovitek
2	2	Patrik Žnidaršič	telefon Samsung Galaxy A5 in ovitek
2	3	Lenart Bučar	telefon Samsung Galaxy J7 in ovitek
2	3	Urban Mišmaš	miška Razer Death Adder Elite
3	1	Urban Duh	telefon Samsung Galaxy S8 in ovitek
			Raspberry Pi 3 model B
			Cormen *et al.: Introduction to Algorithms*
3	1	Gregor Kikelj	telefon Samsung Galaxy A5 in ovitek
			Raspberry Pi 3 model B
			Cormen *et al.: Introduction to Algorithms*
3	2	Tim Poštuvan	telefon Samsung Galaxy A5 in ovitek
			Cormen *et al.: Introduction to Algorithms*
3	2	Jakob Schrader	telefon Samsung Galaxy A5 in ovitek
3	3	Blaž Zupančič	telefon Samsung Galaxy J7 in ovitek
3	3	Aljaž Kolar	telefon Samsung Galaxy J7 in ovitek
Off-line naloga — Risanje s pravokotniki			
	1	Aleksej Jurca	Raspberry Pi 3 model B
	2	Franci Obid	Raspberry Pi 3 model B

SODELUJOČE ŠOLE IN MENTORJI

II. gimnazija Maribor	Mitja Osojnik, Mirko Pešec
Gimnazija Bežigrad	Andrej Šuštaršič
Gimnazija Bežigrad, mednarodna šola	Gregor Anželj
Gimnazija Kranj	Zdenka Vrbinc
Gimnazija Murska Sobota	Romana Vogrinčič
Gimnazija Poljane	Boštjan Žnidaršič
Gimnazija Šiška	Edi Kuklec
Gimnazija Vič	Klemen Bajec, Marina Trost
Osnovna šola Griže	Anita Mandelj
I. gimnazija v Celju	Luka Zlatečan
Srednja elektro-računalniška šola Maribor (SERŠ)	Vida Motaln, Slavko Nekrep, Branko Potisk
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Igor Kutoš, Dominik Letnar
Srednja šola Domžale, Poklicna in strokovna šola	Tadej Trinko
Srednja šola tehniških strok Šiška	Maruša Perič Vučko, Boris Ribaš, Peter Krebelj
Srednja tehniška in poklicna šola Trbovlje (STPŠ)	Uroš Ocepek
Srednja tehniška šola Koper	Andrej Florjančič
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	Miha Baloh
Šolski center Kranj, Strokovna gimnazija	Gašper Strniša
Šolski center Nova Gorica, Elektrotehniška in računalniška šola (ERŠ)	Tomaž Mavri, Edi Medveščček
Šolski center Nova Gorica, Gimnazija in zdravstvena šola (GZŠ)	Barbara Pušnar

Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija (SEŠTG)	Albert Zorko, Danijela Erenda
Šolski center Ptuj, Elektro in računalniška šola (ERŠ)	Franc Vrbančič
Šolski center Ravne na Koroškem, Srednja šola Ravne	Zdravko Pavleković
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Miran Zevnik
Šolski center Velenje, Gimnazija	Miran Zevnik
Škofijska klasična gimnazija Šentvid	Helena Starc Grlj, Matej Tomc
Vegova Ljubljana	Marko Kastelic, Nataša Makarovič, Darjan Toth
Zavod za računalniško izobraževanje (ZRI), Ljubljana	

OFF-LINE NALOGA — RISANJE S PRAVOKOTNIKI

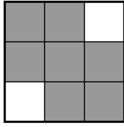
Na računalniških tekmovanjih, kot je naše, je čas reševanja nalog precej omejen in tekmovalci imajo za eno nalogo v povprečju le slabo uro časa. To med drugim pomeni, da je marsikak zanimiv problem s področja računalništva težko zastaviti v obliki, ki bi bila primerna za nalogo na tekmovanju; pa tudi 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 2017 skupaj z razpisom za tekmovanje v znanju; tekmovalci so imeli čas do 23. marca 2018 (dan pred tekmovanjem), da pošljejo svoje rešitve.

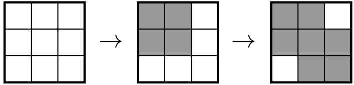
Opis naloge

Dana je karirasta mreža, na kateri so na začetku vse celice bele. Na tej mreži lahko rišemo črne pravokotnike, pri čemer je največja velikost posameznega pravokotnika omejena. Predpisano je končno stanje mreže, ki ga hočemo na ta način doseči (torej nekakšna črno-bela slika, ki jo hočemo narisati). Naloga je poiskati čim manjši nabor pravokotnikov, ki doseže zahtevano končno stanje. Pravokotniki se smejo med seboj tudi prekrivati. Vrstni red, v katerem pravokotnike rišemo, ni pomemben.

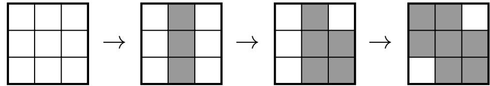
Primer: recimo, da imamo mrežo 3×3 celic, v kateri bi radi na koncu dosegli takšno stanje:



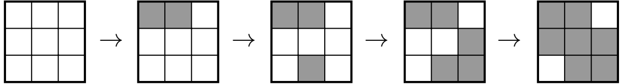
Če so največji dovoljeni pravokotniki velikosti 2×2, lahko to naredimo že z dvema pravokotnikoma:



Če so največji dovoljeni pravokotniki velikosti 3×1 (ali 1×3), potrebujemo vsaj tri pravokotnike:



Če so največji dovoljeni pravokotniki velikosti 2×1 (ali 1×2), potrebujemo vsaj štiri pravokotnike:



Rezultati

Sistem točkovanja je bil tak kot pri off-line nalogah v prejšnjih letih. Pripravili smo 30 testnih primerov, pri čemer je bila mreža velika od 10×10 do 300×300 polj, maksimalna velikost pravokotnika pa je bila od 2×1 do 8×8. Pri vsakem testnem primeru smo razvrstili tekmovalce po številu uporabljenih pravokotnikov, nato pa je prvi tekmovalec (tisti z najmanj pravokotniki) 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 30 testnih primerih.

Letos je svoje rešitve pri off-line nalogi poslalo šest tekmovalcev, od tega trije srednješolci in trije študentje. Končna razvrstitev je naslednja:

Mesto	Ime	Letnik	Šola	Točke
1	Aleksej Jurca	1	FNT, Univ. v Novi Gorici	300
2	Franci Obid	2	ŠC Nova Gorica, SPLŠ	282
3	Uroš Koritnik	1	FRI	206
4	Gregor Kikelj	4	ŠC Novo mesto, SEŠTG	195
5	Loris Štrosar Grmek	4	ŠC Nova Gorica	132
6	Dean Cerin	3	FAMNIT	131

Rešitev

Naloge se lahko lotimo na različne načine odvisno od velikosti mreže in pravokotnikov. Pri nekaj testnih primerih je bila največja dovoljena velikost pravokotnikov 2×1, kar pomeni, da lahko posamezni pravokotnik pokriva le eno ali dve polji. V tem primeru ni nobene koristi od tega, da bi se dva pravokotnika prekrivala: prekrivata se lahko le tako, da je eden čisto vsebovan v drugem (in ga lahko brez škode zavržemo) ali pa imata eno skupno polje, poleg tega pa vsak od njiju pokrije še eno drugo polje; v slednjem primeru lahko enega od njiju zmanjšamo z velikosti 2×1 na 1×1, tako da pokriva le tisto polje, ki ga drugi pravokotnik ne; s tem je izhodna slika enaka kot prej, prekrivanje pa smo odpravili.

Omejimo se lahko torej na take razporeditve pravokotnikov, pri katerih se pravokotniki ne prekrivajo. Ker je skupno število črnih polj fiksno (izhaja iz izhodne slike), bomo najmanjše število pravokotnikov zdaj dosegli tako, da bomo uporabili čim več pravokotnikov velikosti 2×1 (tisto, česar ne bomo pokrili z njimi, pa bomo pač pokrili s pravokotniki 1×1). Vprašanje je torej, kakšno je največje število pravokotnikov 2×1, ki jih lahko postavimo na črna polja naše mreže, ne da bi se med seboj prekrivali.

Definirajmo graf, v katerem je za vsako črno polje mreže po ena točka, dve točki pa sta med seboj neposredno povezani, če imata njuni črni polji skupno stranico (in bi se ju torej dalo pokriti obe hkrati z enim pravokotnikom velikosti 2×1). Ta graf je celo dvodelen, o čemer se najlažje prepričamo tako, da si mrežo predstavljamo kot šahovnico: če pogledamo na njej dve polji, ki imata skupno stranico, vidimo, da

je eno od njiju zagotovo črno, eno pa belo. Točke našega grafa lahko torej razdelimo na dve skupini (črne in bele oz. leve in desne) tako, da gre vsaka povezava med eno točko iz ene skupine in eno iz druge skupine.

Naš problem, kako postaviti čim več pravokotnikov 2×1, ne da bi se prekrivali, je zdaj enakovreden problemu, kako v našem grafu izbrati čim več povezav tako, da nobeni dve izbrani povezavi nimata kakšnega skupnega krajišča. To je problem ujemanja (*matching*) v dvodelnem grafu, ki ga lahko rešimo optimalno v polinomskem času z raznimi znanimi algoritmi, npr. Hopcroft-Karpovim.

Pri nekaterih testnih primerih so bili sicer dovoljeni tudi malo večji pravokotniki, do 4×4, vendar je imela mreža to posebnost, da je bila ena od njenih stranic zelo kratka, dolga največ 10 enot. Takrat lahko nalogo rešujemo z neke vrste dinamičnim programiranjem. Recimo brez izgube za splošnost, da je širina krajša od višine; imamo torej $w \leq 10$, višina h pa je lahko tudi večja (tja do 300). Predstavljamo si lahko, da polagamo pravokotnike v mrežo le od zgoraj navzdol; preden torej položimo na mrežo kakšen tak pravokotnik, ki se začne v vrstici y, poskrbimo, da s prejšnjimi pravokotniki pokrijemo vsa črna polja v vrsticah od 0 do $y - 1$. Ker so pravokotniki lahko veliki največ 4×4, si lahko mrežo med polaganjem v mislih predstavljamo razdeljeno na tri dele: zgornji del, vrstice od 0 do $y - 1$, kjer so vsa črna polja že pokrita točno tako, kot morajo biti v končnem stanju slike; vmesni del, vrstice od y do $y + 3$, kjer so mogoče nekatera črna polja že pokrita, nekatera pa še ne; in spodnji del, od $y + 4$ do $h - 1$, kjer je mreža še popolnoma bela (nepokrita). Stanje slike lahko opišemo z $(y, \mathbf{a})$, pri čemer je $\mathbf{a} = (a_0, a_1, \ldots, a_{w-1})$ in a_x pove, da so v stolpcu x pokrita že vsa črna polja v vrsticah od 0 do $y + a_x$, tista v vrsticah od $y + a_x + 1$ do $h - 1$ pa še ne.

Takšna stanja si lahko predstavljamo kot točke nekega velikega grafa (prostora stanj), pri čemer iz stanja $(y, \mathbf{a})$ vodi v stanje $(y', \mathbf{a}')$ usmerjena povezava (dolžine 1), če je mogoče $(y', \mathbf{a}')$ dobiti iz $(y, \mathbf{a})$ tako, da vanj dodamo en pravokotnik z začetkom v vrstici y. Naloga zdaj pravzaprav sprašuje po najkrajši poti od $(0, \mathbf{0})$, ki predstavlja prazno mrežo, do $(h, \mathbf{0})$, ki predstavlja končno stanje mreže. Poiščemo jo lahko z iskanjem v širino. To je za silo še obvladljivo: ker so pravokotniki omejeni na velikost 4×4, gre lahko a_x od 0 do 4, torej je možnih stanj $O(h \cdot 5^w)$. Poleg tega lahko stanja pregledujemo po naraščajočem y in jih sproti pozabljamo, ko jih ne potrebujemo več, zato je poraba pomnilnika le $O(5^w)$.

V splošnem primeru, torej ko sta obe stranici mreže veliki, postane število stanj neobvladljivo in tega pristopa ne moremo uporabiti. Namesto z optimalno rešitvijo se bomo morali zadovoljiti s kakšnimi hevristikami, ki bodo poskušale najti čim boljšo rešitev (razpored s čim manj pravokotniki), ki pa ne bo nujno najboljša možna. Primer tega je požrešni algoritem: pravokotnike polagajmo enega po enega; na vsakem koraku poiščimo najvišje še nepokrito črno polje; če je takih več v isti vrstici, vzemimo najbolj levo med njimi; v mrežo zdaj položimo črni pravokotnik z zgornjim levim kotom v tem polju, njegovo velikost pa izberimo tako, da bo pokril čim več še nepokritih črnih polj (in seveda nobenega belega, pa tudi čez rob mreže ne sme štrleti).

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, `tekmovanja.acm.si/upm`) 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 od 30. novembra do 2. decembra 2018 v Pragi), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bo od 31. marca do 5. aprila 2019 v Portu na Portugalskem).

Na letošnjem UPM je sodelovalo 52 ekip s skupno 154 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	Žiga Željko (FRI + FMF), Žan Knafelc (FDV), Tim Poštuvan Gim. Vič)	23	24:39:08
2	Filip Koprivec (FMF), Filip Peter Lebar (FE), Patrik Zajec (FRI + FMF)	22	35:40:37
3	Aljaž Eržen, Marko Rus, Žiga Vene (FRI + FMF)	19	27:51:52
4	Gregor Kikelj, Gregor Bučar, Matevž Rom (ŠC NM, SEŠTG)	17	17:52:23
5	Vid Drobnič, Matej Marinko, Žiga Patačko Koderman (FMF)	17	24:17:20
6	Mitja Žalik, Vid Keršič, Niko Uremović (FERI)	17	26:49:58
7	Luka Avbreht, Samo Kralj, Gašper Romih (FMF)	15	20:54:54
8	Ljupche Milosheski, Boshko Koloski, Ilija Tavchioski (FRI)	14	19:03:54
9	Gal Meznarič, Aljaž Jeromel, Jan Mikolič (FERI)	14	22:37:40
10	Žiga Šmelcer (FE), Žiga Gradišar (FMF), Lojze Žust (FRI)	13	17:34:30
11	Izak Glasenčnik, Robi Novak (FERI), Peter Bernad (FNM)	12	9:48:09
12	Andrej Kolar-Požun, Jan Jezeršek, Miha Rot (FMF)	12	14:46:44
13	Matija Kocbek, Luka Horjak, Lovro Drofenik (I. gim. v Celju)	11	20:30:37
14	Arsen Matej Golubovikj, Predrag Zvezdakoski, Igor Saveski (FAMNIT)	10	20:36:34
15	Eda Kaja, Lisi Qarkaxhija, Arbër Avdullahu (FAMNIT)	9	8:50:26
16	Tilen Jesenko, Goran Tubić, Gašper Moderc (FAMNIT)	9	10:29:47
17	Zala Erič, Miha Benčina (FRI), Bor Brecelj (FRI + FMF)	7	12:53:13
18	Luka Lodrant, Lenart Treven (FMF), Matej Tomc (FE)	6	7:40:23
19	Bor Grošelj Simić (G. Vič), Andraž Juvan, Peter Matičič (FRI)	4	7:41:42
20	Vid Megušar, Janez Radešček, Severin Mejak (FMF)	4	10:53:43
21	Ines Meršak, Matic Oskar Hajšen, Jan Rozman (FMF)	4	13:23:57
22	Jernej Rudi Finžgar, Metod Jazbec, Luka Medic (FMF)	3	2:25:27
23	Domen Vaupotič (FKKT), Tim Bauer, Timotej Lemut (FMF)	3	4:36:30
24	Eva Erzin, Nina Slivnik, Eva Zmazek (FMF)	3	4:57:44
25	Miha Rajter, Domen Vreš, Timen Stepišnik Perdih (FRI)	3	5:25:19
26	Maj Bajuk, David Nabergoj, Gregor Štefanič (FRI)	3	6:46:07
27	Jakob Valič, Vid Štrancar, Matija Bolko (FMF)	3	7:03:57
28	Rok Strah, Gašper Golob, Tadej Petrič (Vegova Lj.)	3	7:24:42
29	Žan Hafner Petrovski, Klementina Pirc, Tine Makovecki (FMF)	3	7:37:49
30	Kristjan Kostelec, Benjamin Benčina (FMF), Rok Kos (FRI + FMF)	3	7:47:00
31	Jakob Zmrzlikar, Jakob Pogačnik Souvent, Jakob Schrader (Gim. Vič)	3	8:00:59
32	Marko Kužner, Luka Kobale, Jani Kaukler (FERI)	3	11:02:10
33	Peter Fajdiga, Juš Debelak, Jure Jesenšek (FRI)	3	11:43:48
34	Branka Kojić, Sabina Marancina, Maja Umek (FRI)	2	1:13:08
35	Mirza Krbezlija, Đorđe Mitrović, Isidora Rapajić (FAMNIT)	2	1:37:29
36	Alen Verk, Tristan Višnar (FERI)	2	2:51:31
37	Miha Bastl, Jan Geršak (FRI + FMF), Urban Kocmut (FRI)	2	2:52:07
38	Žan Magerl, Domen Grzin (FRI), Urban Cör (FRI + FMF)	2	3:42:23
39	David Mikek, Urban Knupleš, Tone Krnc (FERI)	2	4:54:14

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

(*nadaljevanje na naslednji strani*)

	Ekipa	Št. rešenih nalog*	Čas
40	Sandra Herševan, Marina Kovač, Barbara Robba (FRI)	2	5:00:11
41	Žiga Flajs, Jure Pirman, Simon Perovnik (FMF)	2	5:17:33
42	Deni Cerovac (FRI), Dan Toškan (FMF)	2	6:07:31
43	Aleš Kert (FRI + FMF), Miha Petek, Luka Kozina (FMF)	1	0:17:28
44	Tobias Mihelčič, Aljoša Rakita, Kevin Cvetežar (FRI)	1	0:40:21
45	Tilen Podsedenšek, Martin Ogrin, Gal Zakrajšek (FMF)	1	1:16:18
46	Kristjan Bleiweis, Aleksander Merhar, Matevž Kušar (FRI)	1	1:26:01
47	Emir Hasanbegović, Andrej Martinovič (FRI), Dominik Žnidaršič (FMF)	1	2:23:18
48	Davor Ornik, Timotej Korda Mlakar, Marcel Mumel (FERI)	1	2:52:43
49	Sebastian Mežnar, Anže Alič, Alen Bizjak (FRI + FMF)	1	3:47:39
50	Miha Markež, Din Mušić, Nejc Jamnik (FRI)	1	4:01:39
51	Urban Vidovič, Andraž Vrečko, Tadej Podrekar (FERI)	0	0:00:00
	Klemen Krsnik, Katja Logar, Timotej Knez (FRI)	0	0:00:00

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

Na srednjeevropskem tekmovanju so nastopile ekipe 1, 2 in 3 kot predstavnice Univerze v Ljubljani, ekipi 7 in 9 kot predstavnici Univerze v Mariboru in ekipa 14 kot predstavnica Univerze na Primorskem. V konkurenci 74 ekip z 31 univerz iz 7 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
38	Niko Uremović, Vid Keršič, Mitja Žalik	2	4:26
44	Filip Koprivec, Samo Kralj, Luka Avbreht	2	7:18
45	Žiga Željko, Žan Knafelc, Tim Poštuvan	2	7:20
53	Marko Rus, Žiga Vene, Aljaž Eržen	1	3:08
60	Gal Meznarič, Jan Mikolič, Aljaž Jeromel	1	6:36
61	Arsen Matej Golubovikj, Predrag Zvezdakoski, Igor Saveski	0	0:00

Na srednjeevropskem tekmovanju je bilo 12 nalog; tokrat je zmagovalna ekipa rešila vse.

ANKETA

Tekmovalcem vseh treh skupin smo na tekmovanju skupaj z nalogami razdelili tudi naslednjo anketo. Rezultati ankete so predstavljeni na str. 169–176.

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++.
— 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 pythonu. 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 80 tekmovalcev prve skupine, 37 tekmovalcev druge skupine in 9 tekmovalcev tretje skupine. Vprašanja so bila pri letošnji anketi enaka kot lani.

Mnenje tekmovalcev o nalogah

Tekmovalce smo spraševali: kako zahtevna se jim zdi posamezna naloga; ali se jim zdi, da jim vzame preveč časa; ali je besedilo primerno dolgo in razumljivo; ali se jim zdi naloga zanimiva; ali so jo rešili (oz. zakaj ne); in katera naloga jim je bila najbolj/najmanj všeč.

Rezultate vprašanj o zahtevnosti nalog kažejo grafi na str. 170. 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: 3,32 v prvi skupini (v prejšnjih letih 3,11, 3,31, 3,41, 3,40, 3,28), 3,19 v drugi skupini (prejšnja leta 3,51, 3,65, 3,33, 3,44, 3,35) in 3,59 v tretji skupini (prejšnja leta 3,73, 3,43, 3,61, 3,19, 3,40).

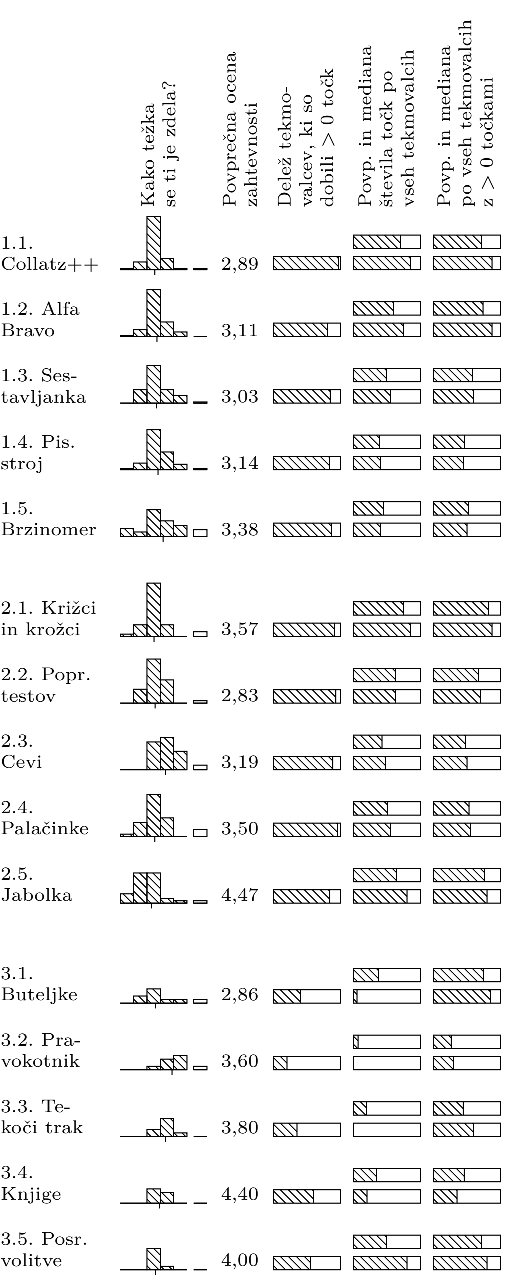
Med tem, kako težka se je naloga zdela tekmovalcem, in tem, kako dobro so jo zares reševali (npr. merjeno s povprečnim številom točk pri tej nalogi), je ponavadi (šibka) negativna korelacija; letos je bila precej močna, podobno kot lani ($R^2 = 0{,}67$; v prejšnjih letih 0,70, 0,39, 0,56, 0,14, 0,52, 0,21, 0,11, pred tem okoli 0,4).

Daleč največ pripomb o tem, kako da je naloga (pre)težka, je bilo pri nalogi 1.5 (brzinomer); pri njej je bilo tudi veliko pripomb, da je besedilo predolgo in težko razumljivo. To, da se tekmovalcem takšne realnočasovne naloge zdijo težke, ni nič neobičajnega, je pa res, da je bilo besedilo tu daljše kot običajno. Rešitev naloge je sicer krajša in preprostejša, kot bi človek na prvi pogled mogoče pričakoval. V drugi skupini se je zdela tekmovalcem težka predvsem naloga 2.3 (cevi); ta je težka, če se ne domislimo, da se splača cevi obravnavati sistematično od zgoraj navzdol in od leve proti desni. V tretji skupini se je zdela tekmovalcem težka predvsem naloga 3.2 (pravokotnik), kar mogoče ni presenetljivo, saj gre za geometrijsko nalogo.

Kot najlažjo so tekmovalci v prvi skupini ocenili nalogo 1.1 (Collatz++), v drugi nalogo 2.5 (jabolka — kar je še posebej lepo, ker gre za realnočasovno nalogo in take se ljudem pogosto zdijo težke), v tretji skupini pa 3.1 (buteljke) in 3.5 (posredne volitve).

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 171. Nad razumljivostjo besedil ni veliko pripomb, v drugi skupini še malo manj kot prejšnja leta. Kot težje razumljiva izstopa še posebej naloga 1.5 (brzinomer), ki smo jo že omenjali zgoraj; poleg nje so kot težje razumljive ocenili še naloge 2.2 (popravljanje testov), 2.4 (palačinke) in 3.3 (tekoči trak).

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, približno enako kot v prejšnjih letih. Pri tem še najbolj izstopa naloga 1.5 (brzinomer), ki se je zdela precej tekmovalcem predolga, nekaj podobnih pripomb pa je bilo tudi pri 3.1 (buteljke). Mnenj, da je kakšno besedilo prekratko, je bilo letos zelo malo.



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

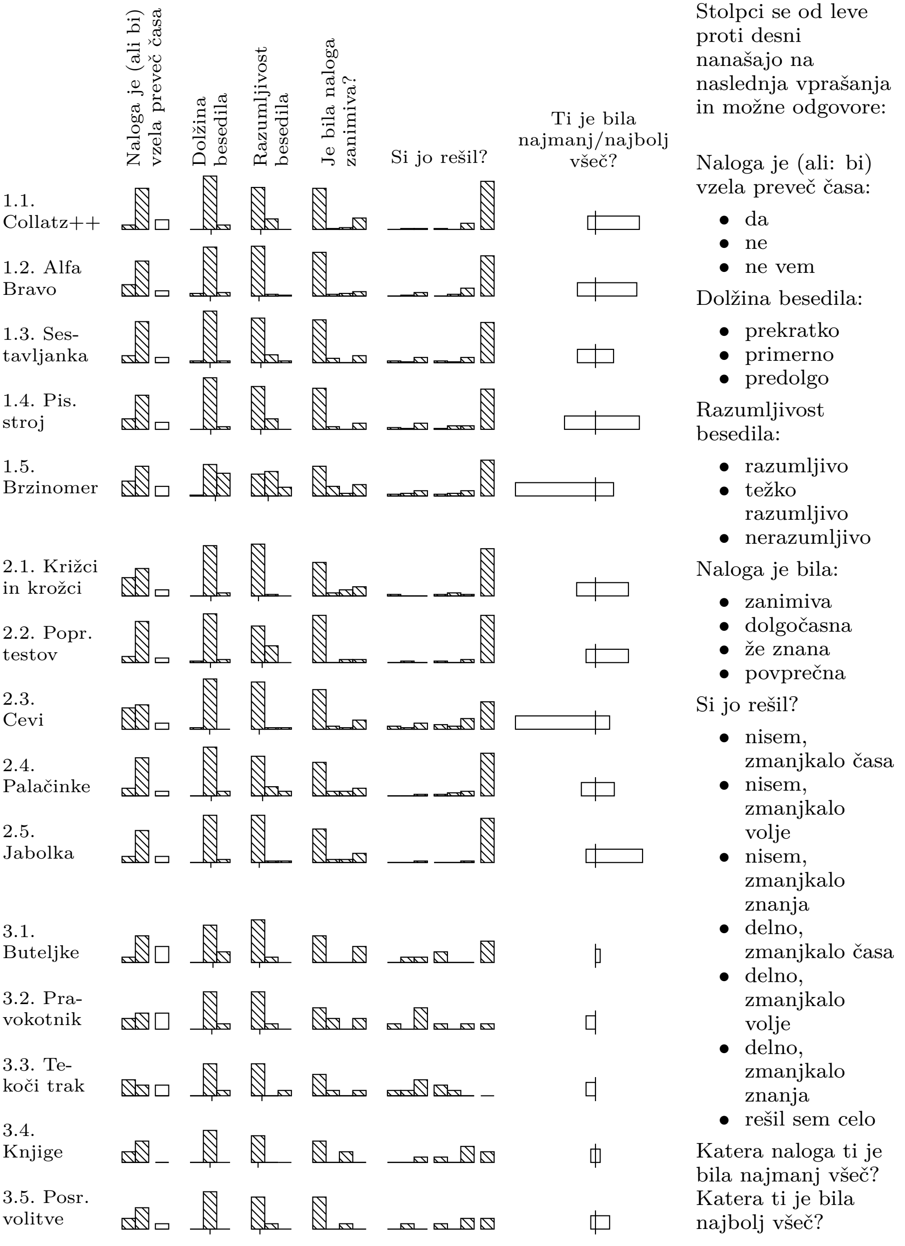
Pomen stolpcev v vsaki vrstici:

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, v drugi in tretji skupini še malo višje. Pripomb, da je neka naloga že znana, je bilo največ pri nalogi 2.1 (križci in krožci), ki res temelji na zelo dobro znani igri. Kot bolj zanimive izstopajo naloge 1.2 (Alfa Bravo), 2.2 (popravljanje testov) in 3.5 (posredne volitve). Ob tem izboru se je težko upreti zaključku, da se tekmovalcem naloge zdijo zanimive predvsem, če so ovite v zanimivo zgodbico.

Pripomb, da bi naloga vzela preveč časa, je bilo malo, podobno kot prejšnja leta. Največ takih pripomb je bilo pri nalogah 2.1 (križci in krožci), 2.3 (cevi) in 3.3 (tekoči trak). Pri 3.3 je implementacija rešitve res lahko zoprna in tudi zamudna. Pri 2.1 so si mnogi tekmovalci otežili delo s tem, da so velik del rešitve razmnožili na štirih izvode, za vsako smer posebej.

Pri glasovih o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, je bila v prvi skupini daleč najbolj nepriljubljena naloga 1.5 (brzinomer), glasovi za najbolj všeč pa so precej razpršeni med 1.1 (Collatz++), 1.2 (Alfa Bravo) in 1.4 (pisalni stroj). V drugi skupini je bila najbolj priljubljena 2.5 (jabolka), kot izrazito nepriljubljena pa izstopa 2.3 (cevi). V tretji skupini jim je bila najbolj všeč naloga 3.5 (posredne volitve), najmanj pa 3.3 (tekoči trak).

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.	15 %	28 %	89 %
map v C++ ipd.	14 %	31 %	78 %
unordered_map v C++ ipd.	26 %	39 %	78 %
zamikanje s **shl**, **shr**	16 %	43 %	78 %
operatorji na bitih	55 %	75 %	67 %
strukture	46 %	83 %	100 %
naštevni tipi	34 %	44 %	89 %
gnezdenje zank	89 %	89 %	78 %
zanka **while**	93 %	100 %	100 %
zanka **for**	94 %	100 %	100 %
kazalci	21 %	51 %	56 %
rekurzija	57 %	81 %	100 %
podprogrami	87 %	100 %	100 %
več-d tabele (**array**)	64 %	85 %	100 %
2-d tabele (**array**)	74 %	94 %	100 %
1-d tabele (**array**)	89 %	100 %	100 %
delo z datotekami	65 %	86 %	100 %
std. vhod/izhod	92 %	94 %	100 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

Rezultati pri vprašanjih o programerskem znanju so podobni tistim iz prejšnjih let. Zlasti v prvi in drugi skupini (pravijo, da) znajo malo več kot lani. Stvari, ki jih

	Prva skupina	Druga skupina	Tretja skupina
drevo	58 %	73 %	78 %
hash tabela	41 %	54 %	89 %
seznam (linked list)	39 %	68 %	89 %
sklad (stack)	47 %	65 %	100 %
vrsta (queue)	43 %	57 %	100 %
Evklidov algoritem	80 %	84 %	100 %
Eratostenovo rešeto	71 %	62 %	89 %
vektorski produkt	47 %	62 %	67 %
rekurzija	57 %	68 %	100 %
dinamično prog.	30 %	44 %	78 %
iskanje v širino	24 %	24 %	88 %
O-zapis	48 %	67 %	89 %
urejanje	68 %	92 %	100 %
bubble sort	41 %	84 %	89 %
insertion sort	26 %	57 %	78 %
selection sort	24 %	46 %	56 %
quicksort	25 %	57 %	89 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

---

tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja leta: kazalci, naštevni tipi in operatorji na bitih, v prvi in drugi skupini tudi strukture. Znanje rekurzije naj bi bilo letos malo boljše kot ponavadi.

Uporaba programskih jezikov

Na splošno so razmerja med različnimi jeziki podobna kot v prejšnjih letih. V prvi skupini je letos z občutno prednostjo najpogostejši jezik python, sledita mu java in C++, nato pa C#. V drugi skupini sta najpogostejša C++ in python (približno izenačena), sledita pa jima C# in java. V tretji skupini je C++ daleč najpogostejši, vendar je bilo letos še kar nekaj tudi uporabnikov jave. Edini jezik, ki se je še pojavil poleg doslej omenjenih, je javascript, ki ga je uporabljal en tekmovalec v drugi skupini. Pascala letos ni uporabljal nihče (to je prvič, odkar zbiramo statistike o uporabi programskih jezikov); tudi basica ne, podobno kot lani in predlani.

Podobno kot prejšnja leta se je tudi letos pojavilo nekaj tekmovalcev, 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 prejšnja leta smo tudi letos pri nalogah tipa „opiši postopek“ pripisali „ali napiši podprogram (kar ti je lažje)“ (kjer je bilo to primerno).

Podobno kot v prejšnjih letih je v anketi še kar nekaj tekmovalcev napisalo, da dobro poznajo tudi PHP in/ali javascript, vendar PHPja na tekmovanju letos ni uporabljal nihče, javascript pa le en tekmovalec.

	Leto in skupina																	
	2018			2017			2016			2015			2014			2013		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal				4			$\frac{1}{3}$	3		5	2		$2\frac{1}{2}$	2	1	1		1
C	5	4	$\frac{1}{2}$	4	3	$2\frac{1}{2}$	$4\frac{1}{3}$	1	2	3		1	$3\frac{1}{2}$	6		2	7	
C++	$18\frac{1}{2}$	13	11	23	10	$15\frac{1}{2}$	28	8	9	27	9	$9\frac{1}{2}$	19	$4\frac{1}{2}$	$10\frac{1}{2}$	17	$12\frac{1}{2}$	7
java	$21\frac{1}{2}$	$8\frac{1}{2}$	4	28	3	2	24	6	5	22	6	$3\frac{1}{2}$	23	2	$1\frac{1}{2}$	12	8	1
PHP			–			–			–	3		–	2		–	1	$\frac{1}{2}$	–
basic			–									1		1	–	1		–
C#	11	6		7	6		12	5	1	16	5		12	$1\frac{1}{2}$	2	18	$\frac{1}{2}$	
python	38	11	$\frac{1}{2}$	42	11	–	$29\frac{1}{3}$	12	–	26	1	–	16	6	–	16	8	–
NewtonScript			–			–			–			–			–		$\frac{1}{2}$	–
javascript		$\frac{1}{2}$	–			–	1		–	1		–	1		–			–
julia			–	1		–			–			–			–			–
batch			–			–			–			–	1		–			–
psevdokoda	3	1	–	5		–	5		–	6	1	–	10		–	6		–
nič	2		1				2	3		4	1		4	2		2		

Število tekmovalcev, ki so uporabljali posamezni programski jezik. Nekateri uporabljajo po več različnih jezikov (pri različnih nalogah) in se štejejo delno k vsakemu jeziku. (V letu 2018 sta dva tekmovalca uporabljala python in C++, eden python in javo, eden C in C++, eden pa javo in javascript.) „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“.

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; zdaj že veliko tekmovalcev na primer uporablja `string` namesto **`char`** `*` in tip `vector` namesto tradicionalnih tabel (*arrays*). Novosti, po katerih se zadnje različice C++ (od vključno C++11 naprej) razlikujejo od C++98, pa letos ni uporabljal nihče.

Pri pythonu zdaj že vsi uporabljajo python 3 in ne python 2; je pa res, da je pri tako preprostih programih, s kakršnimi se srečujemo na našem tekmovanju, razlika večinoma le v tem, ali `print` uporabljajo kot stavek ali kot funkcijo.

V besedilu nalog za 1. in 2. skupino objavljamo deklaracije tipov, spremenljivk, podprogramov ipd. v pascalu, C/C++, C#, pythonu in javi. Delež tekmovalcev, ki pravijo, da deklaracije razumejo, je letos v prvi skupini podobno visok kot lani (67/74), v drugi malo nižji (34/36). 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 ali C#; originalna predloga sta bila javascript in VB.NET. 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 objavljamo od 2017 izvorno kodo v C++, pri prvi skupini pa tudi v pythonu. Tekmovalce smo v anketi vprašali, če razumejo C++ dovolj, da si lahko kaj pomagajo s izvorno kodo v rešitvah, in če bi radi videli izvorno kodo

rešitev še v kakšnem drugem jeziku. Večina je s C++ sicer zadovoljna (37/67 v prvi skupini, 27/36 v drugi, 9/9 v tretji); ti deleži so še malo višji kot lani. Kljub temu je v prvi skupini skoraj polovica takih, ki pravijo, da izvorne kode v rešitvah ne razumejo; mogoče bi bil ta delež nižji, če ne bi mi v anketi pozabili omeniti, da je izvorna koda rešitev za prvo skupino zdaj objavljena tudi v pythonu. 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++, jih največ omenja python in (malo manj) javo. Do nadaljnjega bomo zato objavljali rešitve nalog za prvo skupino tudi v pythonu in ne le v C++.

Letnik

Običajno so tekmovalci zahtevnejših skupin večinoma v višjih letnikih kot tisti iz lažjih skupin, vendar so letos te razlike manjše kot ponavadi. Razmerja so podobna kot prejšnja leta, v povprečju pa so se tekmovalci letos rahlo postarali. Letos sta nastopila tudi dva osnovnošolca, eden v prvi in eden v drugi skupini.

	Št. tekmovalcev po letnikih							
Skupina	8	9	1	2	3	4	5	Povprečni letnik
prva	1		10	24	33	25	6	3,1
druga		1	4	6	16	17		3,0
tretja			2	3	2	10		3,2

Druga vprašanja

Podobno kot prejšnja leta je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!). V smislu širitve zanimanja za tekmovanje in večanja števila tekmovalcev se zelo dobro obnese šolsko tekmovanje, ki ga izvajamo zadnjih nekaj let, saj se odtlej v tekmovanje vključuje tudi nekaj šol, ki prej na našem državnem tekmovanju niso sodelovale.

Pri vprašanju, kje so se naučili programirati, je podobno kot prejšnja leta najpogostejši odgovor, da so se naučili programirati sami; sledijo tisti, ki so se tega naučili v šoli pri pouku, 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, najpogostejši kombinaciji pa sta „več časa, enako nalog“ in „enako časa, manj nalog“.

Iz odgovorov na vprašanje, kakšne potekmovalne dejavnosti bi jih zanimale, je težko zaključiti kaj posebej konkretnega.

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb.

Od 2009 imajo tekmovalci v prvi in drugi skupini možnost pisati svoje odgovore na računalniku namesto na papir (kar so si prej v anketah že večkrat želeli). Velika večina jih je res oddajala odgovore na računalniku, nekaj pa jih je vseeno reševalo na papir. Pri oddajanju odgovorov na računalniku se stanje izboljšuje; letos je bilo sicer še nekaj težav s shranjevanjem.

Podobno kot prejšnja leta si je veliko tekmovalcev želelo tudi, da bi imeli v prvi in drugi skupini na računalnikih prevajalnike in podobna razvojna orodja. Razlog,

	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	77	0	3	0	56	42	20	1	4	5	8	50	6	3	53	23	19	19	27	27	17	21	38
II	31	2	2	4	27	19	9	4	3	8	2	24	4	4	26	17	13	19	15	19	15	19	17
III	8	0	0	1	7	0	4	1	2	1	0	5	0	3	3	1	3	0	1	4	4	3	2

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) Nekdo očitno ni ljubitelj Damjana Murka:

```
a = int(input("Vnesi prvi parameter a: ")
b = int(input("Vnesi drugi parameter b, hvala, da nisi pozabil name: ")
if a < b: print("Žal si sorodnik Damjana Murka, ne bo šlo tako, "
             "a mora biti večji od b.")
```

(1.1) Včasih se zgodi, da tekmovalci pišejo rešitve, ki delujejo samo za primer iz besedila naloge namesto v splošnem. Letos je pri prvi nalogi v prvi skupini kakšnih 17 tekmovalcev napisalo rešitev, ki poišče tiste k, pri katerih ima Collatzevo zaporedje maksimum 9232, ker je bil slučajno tak maksimum pri primeru v besedilu naloge...

(1.1) Nekdo ima tako rad dolga imena spremenljivk, da je celo vhodna podatka a in b skopiral v spremenljivki z daljšima imenoma in odtlej uporabljal tidve:

```
int najmanjse_stevilo_intervala = a;
int najvecje_stevilo_intervala = b;
int stevila_s_intervala_ki_imajo_najvecjo[najvecje_stevilo_intervala −
                                    najmanjse_stevilo_intervala + 1];
```

(1.1) Tale tekmovalec ni vedel, kaj je standardni vhod, pa je vseeno uspešno bral z njega:

```
prvostevilo = input("Vpisite prosim prvo stevilo: ") # ker ne vem, kaj so standardni
   # vhodi in izhodi, se bo treba zadovoljiti s takim vnosom
```

(1.1) Rešitev z zaupanjem v nadnaravne sile:

```
i = g # ne me vprašat, zakaj sem to naredil
⋮
while (i > g && i < h): # magična zanka, ki se mi je ne da razlagati, ampak samo
   # vedite, da je meni imela na neki točki smisel. . . . ok probal bom razložit
```

Ker postavi i na g namesto na g + 1, se zanka ne bo izvedla niti enkrat.

(1.1) Impresivno zapleten način za preverjanje sodosti:

```
if (Integer[x / 2] * 2 == (x / 2) * 2) {    // pregledam, če je x deljiv z 2
```

(1.2) To se pa zgodi, če programiramo s copy + paste:

```
if (besedna_crka == "ALFA")
   beseda += "A";
if (besedna_crka == "ALFA")
   beseda += "B";
if (besedna_crka == "ALFA")
   beseda += "C";
```

In tako naprej do Z. Tega, ali se vhodna beseda razlikuje od prave v eni črki, pa ni preverjal.

(1.2) Rešitev s pomanjkanjem časa:

Če bi imela čas, bi prvo vse vzorčne besede zapisala v array stringov.

(1.2) Verjetno najdaljši identifier letos:

```
def primerjavaEkstraTajnihBesedZEkstraTajnimSlovarjem(beseda, pozicijaSlovar):
```

(1.2) Domiselna rešitev: ker vemo, da lahko pride do napake le pri eni črki kodne besede, to pomeni, da jo lahko razdelimo na levo in desno polovico in se mora vsaj ena od teh dveh polovic ujemati z našo vhodno besedo:

```
for i in list_besed:
    if "AL" or "FA" in i:
        koncni_niz += "A"
    elif "BRA" or "VO" in i:
        koncni_niz += "B"
```

in tako naprej. Gornja implementacija ima sicer to slabost, da ne preverja, če se nek niz pojavlja ravno na začetku ali na koncu niza i, zato bi nekatere besede dekodirala narobe (npr. tako GOLF kot TANGO vsebujeta GO, zato bi se oba dekodirala v G).

(1.2) Komentar po deklaraciji tabele z vsemi 26 kodnimi besedami:

```
// Tole je bilo mukotrpno
```

Še en podoben primer (kodne besede so bile seveda vse z velikimi črkami):

```
// ZGORI SM VNESU VSE PODATKE
// sry caps ostal od prepisvanja
```

(1.2) V čast nam je, da lahko naslednji rešitvi podelimo posebno priznanje — veliki zlati tabulator za najglobljo indentacijo vseh časov (oz. vsaj odkar pomni uredništvo Cvetk):

```
int main() {
    ⋮
    for (i = 0; i < stBesed; i++) { // Za vsako besedo
        if (PreveriBesedo(besede[i], "ALFA") == 1) { // Preveri besedo ALFA (A)
            printf("A");
        } else {
            if (PreveriBesedo(besede[i], "BRAVO") == 1) { // Preveri besedo BRAVO (B)
                printf("B");
            } else {
                    ⋱
                        if (PreveriBesedo(besede[i], "ZULU") == 1) {
                            printf("Z");
```

Razlika med prvo in zadnjo vrstico je kar 28 nivojev (prejšnji rekord je bil 23 nivojev; glej bilten 2013, str. 138). Resnici na ljubo je sicer treba priznati, da dolga zaporedja stavkov **if** ... **else if** na naših tekmovanjih niso nobena redkost, le da ponavadi niso takole zamaknjena. Brez takšnega veriženja **if**-ov pa je letošnji rekord 11 nivojev (pri 1. nalogi v 3. skupini), kar je vsekakor tudi impresivno.

(1.2) Rešitev, ki se res potrudi, da se izvajanje ne bi nadaljevalo iz enega **case**a v drugega:

```
public static int vrniSt(char c) {
   case 'A': return 0; break;
   case 'B': return 1; break;
```

(1.3) Prispevek za rubriko „tekmovalci čestitajo in pozdravljajo“:

```
# pozdravčke tistemu ki to popravlja :) (meu sm še minutko)
```

(1.3) Zanimivo ime za logično spremenljivko:

```
boolean tru = true;
while (tru) {
   ⋮
   if (tmpold < tmp) { // če se odstopanje spet začne povečevati, ustavi program
      tru = false;
```

(1.3) Rešitev z velikim nezaupanjem do deklaracije tipa spremenljivke:

```
int bb = p / aa;
if (aa * (int) bb == p) // Vem da bi bb moralo biti celo število, ampak ne želim reskirati
```

(1.3) Rešitev s podlimi insinuacijami na račun protagonista naše naloge:

```
print = ("DOBER DAN GOSPOD GOJMIR VELIKI!!!") # Lepo pozdravimo saj nočemo biti
# nevljudni in v nalogi ne piše kako agresiven je lahko ta Gojmir. Mislim jezus kristus
# kaj če je neki nasilnež? Nočem bit tepen.
```

(1.3) Da človek ne pozna (ali noče uporabiti) funkcije abs, lahko razumemo; težje pa, zakaj potem ne uporabi unarnega minusa:

```
if (razlikarazmerja < 0) // Če je spremenljivka razlikarazmerja negativna,
                         // spremeni v pozitivno
   razlikarazmerja = razlikarazmerja − razlikarazmerja − razlikarazmerja;
```

Podobno je pri isti nalogi naredil še en tekmovalec.

(1.3) Prijetno ekscentričen način za preverjanje, ali je x praštevilo:

```
delitelji_x = []
for n in range(1, x + 1): # iščem delitelje števila x v range od 1 do x + 1
   if x % n == 0: # če obstaja delitelj, ga dodam na seznam
      delitelji_x.append(n)
if sum(delitelji_x) == 1 + x: # seštejem delitelje, če sta to le 1 in x, je x praštevilo
   mozni_prafaktorji.append(x)
```

(1.4) Rešitev za ljubitelje abstruzne terminologije:

> Število pomikov izračunamo z enačbo (največje št. vnosov na mesto) krat (število znakov oz. dolžina vrstice) plus (največji i mesta z največjim številom vnosov na mesto)
> to postulira številu g
> Natisnemo g

Ni sicer prav očitno, zakaj bi bilo primerno uporabljati besedo „postulira“ v pomenu „priredi“.

(1.5) Komentar na začetku ene od rešitev:

```
# Funkcija vrača števila in ne nize, saj se da s tem prihraniti prostor v pomnilniku,
# kar je pri hitrih spremembah lahko pomembno. :)
```

Kako mu je sploh prišlo na misel, da bi kdo lahko pričakoval, da bo vračala nize?

(1.5) Pri tej nalogi je imelo precej tekmovalcev težave s tem, kako na začetku spraviti kazalec na 0. Naslednja rešitev je dobro zamišljena, a pozabi, da se izvajanje s stavkom **return** vrne iz funkcije:

```
if c <= 250: # ob prvih 250 klicih funkcije se vrača vrednost za pomikanje kazalca
             # navzdol, tako da zagotovimo njegovo začetno lego na nič
  return (−1)
  c −= 1
```

Še bolj ekstremen primer:

```
def začetni položaj: # nastavi začetni položaj na 0
  lega = 0
  for i in range(250):
    return("-1")
```

(1.5) Naloga pravi, da začetnega položaja kazalca ne poznamo; tale tekmovalec se s tem dejstvom spoprime tako, da pač reče uporabniku, naj ta položaj vnese:

```
def Začetnalega(skrivna)
  y = 0
  skrivna = int(input("Pač te vrednosti ne vemo, "
                      "ampak v naravi mora obstajati: "))
```

(1.5) Zakaj bi imeli navaden **int**, če imamo lahko tabelo z enim elementom:

```
int hitrostt[1];
hitrostt[0] = 0; // shranjena prejšnja hitrost
```

(1.5) Zanimiv fizikalni razmislek na začetku ene od rešitev:

> Ta program ne potrebuje sistema, ki bi prestavljal kazalec za več kot 1 km/h gor/dol naenkrat, saj, če avto v eni stotinki sekunde (ura kliče podprogram približno 100× v sekundi) pospeši za 1 km/h, je ta pospešek velik 27,78 m/s^2, kar avti ne dosegajo, hitrejši je celo od gravitacijskega pospeška, zato sem to izpustil, ker se mi ne zdi smiselno. Še formula 1 ima le okrog 13 m/s^2 pospeška.

(1.5) Čudovito dekadentna rešitev: namesto globalne spremenljivke uporablja kar datoteko!

> Metoda premik prebere zadnjo hitrost iz datoteke `zadnja.txt` in jo zapiše v spremenljivko `temp`. Zatem v isto datoteko zapiše novo hitrost.

To je lepo tudi implementiral (v javi).

(1.5) Rešitev za ljubitelje konstruktivistične poezije:

```
inline f(100 times per second);
n ∈ ℕ
0 ≤ n ≤ 300
n =/= ++301
if inline 1 < inline 2; // vsako stotinko sekunde en inline
int Premik "int hitrost +1 " endl;
```

(2.1) So šle reforme učnega načrta vendarle predaleč?

```
# to je implementacija greedy algoritma (se mi zdi, naučil sem se ga včeraj
# pri uri športne vzgoje)
```

(2.1) Komentar na začetku rešitve:

```
# Sklepam, da ima tabela m stolpcev in n vrstic, to mi ni najbolj jasno iz navodil.
```

Ko bi vsaj v prvem stavku navodil pisalo „imamo igralno ploščo z mrežo m vrstic in n stolpcev“ ali kaj podobnega...

(2.3) Eden od tekmovalcev stavkom **if** dosledno pravi „zanke“:

> Tudi ta problem bi lahko rešil s pomočjo nekaj **if** zank, a več kot v primeru ravnih cevi. Pri za vsakem premiku bi pazil, da se število premikov bilo pravilno prešteto z dodatnimi **if** zankami. [...] Vse skupaj bi lahko rešil z **if** zankami.

(2.3) Komentar na koncu neučinkovite rešitve:

> Iteriramo celotno matriko, kjer za vsak element poskusimo vse možne rotacije z vsako možno rotacijo. Imamo še števec, ki prišteva vrednosti za vsako izvedeno rotacijo. Tako poskusimo za vse možnosti, hkrati pa najdemo najbolj optimalno rešitev.
> O(zelo počasi :( )

(2.3) Rešitev za ljubitelje toka zavesti:

> [...] in če bi program prišel okoli in bi prišel do ene ki ima X1 oziroma X2 (X označuje za koliko je zarotiran) potem bi sklepal da je naredil prav če to ni res bi vse ponastavil v tem krogu in bi postavil korake na 0 in bi v naslednjem poskusu priključil na druga 2 soseda in tako bi potem bi tako lahko izvedel v koliko potezah bi to naredil verjetno sem kakšno stvar spustil kako bi naredil ker je malo težko opisati vendar ključne zadeve bi morale biti seveda optimizacija ni najboljša vendar nikjer v nalogi ne piše da moramo paziti na optimizacijo

(2.4) Komentar na začetku rešitve:

```
// to je opis postopka, ki je mogoče malo podoben kodi v javi
import java.util.Scanner;
```

Mogoče pa res!

(2.4) Nekdo je v imenu spremenljivke okrajšal besedo „palačinke“ na . . . zanimiv način:

```
pinke = []
for i in range(n)
    pinke.append(i) # creates stack of pancakes
```

(2.4) Komentar na začetku rešitve:

```
// jest se z Mihcem strinjam, čeprav jih raje jem kot obračam
```

(2.5) Nekomu se je očitno zdelo ime naše funkcije SproziRoko premalo slikovito:

```
if trak[i] == i:
    premakni_roko(i)
    rukn_ga_u_jarek()
```

(2.5) Na koliko načinov se da napisati besedo „jabolko“?

```
List<Jabuk> japkaNaTraku = new LinkedList<Jabuk>(),
            dirty = new ArrayList<Jabuk>();
while (true) { // Mmmmm, neskončno jabolk
```

(2.5) Komentar na začetku rešitve:

```
// Predpostavim, da bo program zagnan pred začetkom sortiranja jabolk ali tik ob začetku.
// Kmet, ki pričakuje, da bo ugasnjen računalnik urejal jabolka, naj jih ureja sam.
```

(3.1) Rešitev z močnimi stališči o indeksiranju:

```
// iz primera v navodilih razvidno, da je one-index
// (tri osebe, od 1 do 3). hocem, da so zero-index ker sem civilizirana oseba
```

(3.1) Rešitev za ljubitelje ameriške politike:

```
for (int georgeBushIsTheRealMVP = 0; georgeBushIsTheRealMVP < jjj;
                                                  georgeBushIsTheRealMVP++)
```

(3.2) Pri tej nalogi je nekdo oddal rešitev, ki vedno izpiše najmanjši pravokotnik, ki vsebuje vse modre točke; nekdo drug pa rešitev, ki izpiše najmanjši pravokotnik, ki vsebuje vse točke ne glede na barvo. Na srečo sta oba dobila po 0 točk.

(3.3) Zakaj bi rekli „**if** (prej < 9)“, če lahko zakompliciramo:

```
if (10 − prej > 1) {
```

(3.3) Na tekmovanjih si ljudje pogosto poskušajo z raznimi **typedef**i prihraniti tipkanje, kar pa včasih pripelje do precej kriptičnih imen tipov. Lep primer z našega letošnjega tekmovanja:

```
typedef pair<int, int> ii;
typedef list<ii> lii;
  ⋮
typedef list<pair<ii, int> > liii;
```

Kasneje se je na iteratorje v slednjem seznamu skliceval kot liii::iterator in tako zamudil priložnost, da bi deklariral še tip liiii :)

(3.4) Nek tekmovalec je pri tej nalogi oddal kar 25 rešitev, ki so večinoma izpisovale kakšno konstanto v upanju, da bodo slučajno zadele pravilni rezultat. Ko prvih nekaj ni imelo uspeha, jih je poskusil takole izboljšati z dodatkom naključnosti:

```
srand(time(NULL));
if (rand() % 2 ==1 ) { cout << 6; }
else { cout << 9; }
```

Podobne bleferske rešitve je enako neuspešno oddajal tudi pri prvi nalogi.

(3.5) Nekdo o volilcih očitno nima dobrega mnenja:

```
struct drzav { int u, k[2]; };
⋮
vector<drzav> voli;
```

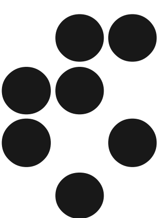
(3.5) Komentar iz ene od rešitev (ki sicer sploh ni bila slaba):

```
/* Ojoj. To stvar z DFS in SCC pa sem res malo preveč zakompliciral.
```

Če je računal krepko povezane komponente, je res malo preveč kompliciral. V resnici je naš graf tak, da vsak cikel tvori eno krepko povezano komponento, vsaka od preostalih točk pa je krepko povezana komponenta sama zase.

SODELUJOČE INŠTITUCIJE

Institut Jožef Stefan



Institut je največji javni raziskovalni zavod v Sloveniji s skoraj 800 zaposlenimi, od katerih ima približno polovica doktorat znanosti. Več kot 150 naših doktorjev je habilitiranih na slovenskih univerzah in sodeluje v visokošolskem izobraževalnem procesu. V zadnjih desetih letih je na Institutu opravilo svoja magistrska in doktorska dela več kot 550 raziskovalcev. Institut sodeluje tudi s srednjimi šolami, za katere organizira delovno prakso in jih vključuje v aktivno raziskovalno delo. Glavna raziskovalna področja Instituta so fizika, kemija, molekularna biologija in biotehnologija, informacijske tehnologije, reaktorstvo in energetika ter okolje.

Poslanstvo Instituta je v ustvarjanju, širjenju in prenosu znanja na področju naravoslovnih in tehniških znanosti za blagostanje slovenske družbe in človeštva nasploh. Institut zagotavlja vrhunsko izobrazbo kadrom ter raziskave in razvoj tehnologij na najvišji mednarodni ravni.

Institut namenja veliko pozornost mednarodnemu sodelovanju. Sodeluje z mnogimi uglednimi institucijami po svetu, organizira mednarodne konference, sodeluje na mednarodnih razstavah. Poleg tega pa po najboljših močeh skrbi za mednarodno izmenjavo strokovnjakov. Mnogi raziskovalni dosežki so bili deležni mednarodnih priznanj, veliko sodelavcev IJS pa je mednarodno priznanih znanstvenikov.

Tekmovanje sta podprla naslednja odseka IJS:

CT3 — Center za prenos znanja na področju informacijskih tehnologij

Center za prenos znanja na področju informacijskih tehnologij izvaja izobraževalne, promocijske in infrastrukturne dejavnosti, ki povezujejo raziskovalce in uporabnike njihovih rezultatov. Z uspešnim vključevanjem v evropske raziskovalne projekte se Center širi tudi na raziskovalne in razvojne aktivnosti, predvsem s področja upravljanja z znanjem v tradicionalnih, mrežnih ter virtualnih organizacijah. Center je partner v več EU projektih.

Center razvija in pripravlja skrbno načrtovane izobraževalne dogodke kot so seminarji, delavnice, konference in poletne šole za strokovnjake s področij inteligentne analize podatkov, rudarjenja s podatki, upravljanja z znanjem, mrežnih organizacij, ekologije, medicine, avtomatizacije proizvodnje, poslovnega odločanja in še kaj. Vsi dogodki so namenjeni prenosu osnovnih, dodatnih in vrhunskih specialističnih znanj v podjetja ter raziskovalne in izobraževalne organizacije. V ta namen smo postavili vrsto izobraževalnih portalov, ki ponujajo že za več kot 500 ur posnetih izobraževalnih seminarjev z različnih področij.

Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi virtualno učečo se skupnost in pripomoči k učinkovitejšemu povezovanju znanosti in industrije ter večji prepoznavnosti domačega znanja v slovenskem, evropskem in širšem okolju.

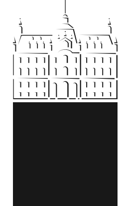
E3 — Laboratorij za umetno inteligenco

Področje dela Laboratorija za umetno inteligenco so informacijske tehnologije s poudarkom na tehnologijah umetne inteligence. Najpomembnejša področja raziskav in razvoja so: (*a*) analiza podatkov s poudarkom na tekstovnih, spletnih, večpredstavnih in dinamičnih podatkih, (*b*) tehnike za analizo velikih količin podatkov v realnem času, (*c*) vizualizacija kompleksnih podatkov, (*d*) semantične tehnologije, (*e*) jezikovne tehnologije.

Laboratorij za umetno inteligenco posveča posebno pozornost promociji znanosti, posebej med mladimi, kjer v sodelovanju s Centrom za prenos znanja na področju informacijskih tehnologij (CT3) razvija izobraževalni portal VideoLectures.NET in vrsto let organizira tekmovanja ACM v znanju računalništva.

Laboratorij tesno sodeluje s Stanford University, University College London, Mednarodno podiplomsko šolo Jožefa Stefana ter podjetji Quintelligence, Cycorp Europe, LifeNetLive, Modro Oko in Envigence.

*



Fakulteta za matematiko in fiziko

Fakulteta za matematiko in fiziko je članica Univerze v Ljubljani. Sestavljata jo Oddelek za matematiko in Oddelek za fiziko. Izvaja dodiplomske univerzitetne študijske programe matematike, računalništva in informatike ter fizike na različnih smereh od pedagoških do raziskovalnih.

Prav tako izvaja tudi podiplomski specialistični, magistrski in doktorski študij matematike, fizike, mehanike, meteorologije in jedrske tehnike.

Poleg rednega pedagoškega in raziskovalnega dela na fakulteti poteka še vrsta obštudijskih dejavnosti v sodelovanju z različnimi institucijami od Društva matematikov, fizikov in astronomov do Inštituta za matematiko, fiziko in mehaniko ter Instituta Jožef Stefan. Med njimi so tudi tekmovanja iz programiranja, kot sta Programerski izziv in Univerzitetni programerski maraton.

Fakulteta za računalništvo in informatiko

Glavna dejavnost Fakultete za računalništvo in informatiko Univerze v Ljubljani je vzgoja računalniških strokovnjakov različnih profilov. Oblike izobraževanja se razlikujejo med seboj po obsegu, zahtevnosti, načinu izvajanja in številu udeležencev. Poleg rednega izobraževanja skrbi fakulteta še za dopolnilno izobraževanje računalniških strokovnjakov, kot tudi strokovnjakov drugih strok, ki potrebujejo znanje informatike. Prav posebna in zelo osebna pa je vzgoja mladih raziskovalcev, ki se med podiplomskim študijem pod mentorstvom univerzitetnih profesorjev uvajajo v raziskovalno in znanstveno delo.

Fakulteta za elektrotehniko, računalništvo in informatiko

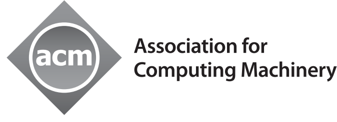
Fakulteta za elektrotehniko, računalništvo in informatiko (FERI) je znanstveno-izobraževalna institucija z izraženim regionalnim, nacionalnim in mednarodnim pomenom. Regionalnost se odraža v tesni povezanosti z industrijo v mestu Maribor in okolici, kjer se zaposluje pretežni del diplomantov dodiplomskih in podiplomskih študijskih programov. Nacionalnega pomena so predvsem inštituti kot sestavni deli FERI ter centri znanja, ki opravljajo prenos temeljnih in aplikativnih znanj v celoten prostor Republike Slovenije. Mednarodni pomen izkazuje fakulteta z vpetostjo v mednarodne raziskovalne tokove s številnimi mednarodnimi projekti, izmenjavo študentov in profesorjev, objavami v uglednih znanstvenih revijah, nastopih na mednarodnih konferencah in organizacijo le-teh.

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Fakulteta za matematiko, naravoslovje in informacijske tehnologije Univerze na Primorskem (UP FAMNIT) je prvo generacijo študentov vpisala v študijskem letu 2007/08, pod okriljem UP PEF pa so se že v študijskem letu 2006/07 izvajali podiplomski študijski programi Matematične znanosti in Računalništvo in informatika (magistrska in doktorska programa).

Z ustanovitvijo UP FAMNIT je v letu 2006 je Univerza na Primorskem pridobila svoje naravoslovno uravnoteženje. Sodobne tehnologije v naravoslovju predstavljajo na začetku tretjega tisočletja poseben izziv, saj morajo izpolniti interese hitrega razvoja družbe, kakor tudi skrb za kakovostno ohranjanje naravnega in družbenega ravnovesja. V tem matematična znanja, področje informacijske tehnologije in druga naravoslovna znanja predstavljajo ključ do odgovora pri vprašanjih modeliranja družbeno ekonomskih procesov, njihove logike in zakonitosti racionalnega razmišljanja.

ACM Slovenija



ACM je največje računalniško združenje na svetu s preko 80 000 člani. ACM organizira vplivna srečanja in konference, objavlja izvirne publikacije in vizije razvoja računalništva in informatike.

ACM Slovenija smo ustanovili leta 2001 kot slovensko podružnico ACM. Naš namen je vzdigniti slovensko računalništvo in informatiko korak naprej v bodočnost.

Društvo se ukvarja z:

- Sodelovanjem pri izdaji mednarodno priznane revije Informatica — za doktorande je še posebej zanimiva možnost objaviti 2 strani poročila iz doktorata.
- Urejanjem slovensko-angleškega slovarčka — slovarček je narejen po vzoru Wikipedije, torej lahko vsi vanj vpisujemo svoje predloge za nove termine, glavni uredniki pa pregledujejo korektnost vpisov.
- ACM predavanja sodelujejo s Solomonovimi seminarji.
- Sodelovanjem pri organizaciji študentskih in dijaških tekmovanj iz računalništva.

ACM Slovenija vsako leto oktobra izvede konferenco Informacijska družba in na njej skupščino ACM Slovenija, kjer volimo predstavnike.

IEEE Slovenija

Inštitut inženirjev elektrotehnike in elektronike, znan tudi pod angleško kratico IEEE (Institute of Electrical and Electronics Engineers) je svetovno združenje inženirjev omenjenih strok, ki promovira inženirstvo, ustvarjanje, razvoj, integracijo in pridobivanje znanja na področju elektronskih in informacijskih tehnologij ter znanosti.

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.

SREBRNA POKROVITELJA

Quintelligence
Obstoječi informacijski sistemi podpirajo predvsem procesni in organizacijski nivo pretoka podatkov in informacij. Biti lastnik informacij in podatkov pa ne pomeni imeti in obvladati znanja in s tem zagotavljati konkurenčne prednosti. Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja. IKT (informacijsko-komunikacijska tehnologija) je postavila temelje za nemoten pretok in hranjenje podatkov in informacij. S primernimi metodami je potrebno na osnovi teh informacij izpeljati ustrezne analize in odločitve. Nivo upravljanja in delovanja se tako seli iz informacijske logistike na mnogo bolj kompleksen in predvsem nedeterminističen nivo razvoja in uporabe metodologij. Tako postajata razvoj in uporaba metod za podporo obvladovanja znanja (knowledge management, KM) vedno pomembnejši segment razvoja.

Podjetje Quintelligence je in bo usmerjeno predvsem v razvoj in izvedbo metod in sistemov za pridobivanje, analizo, hranjenje in prenos znanja. S kombiniranjem delnih — problemsko usmerjenih rešitev, gradimo kompleksen in fleksibilen sistem za podporo KM, ki bo predstavljal osnovo globalnega informacijskega centra znanja.

Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja.

Zemanta
Na Zemanti gradimo najnaprednejšo platformo za nativno oglaševanje na svetu. Oglaševalske agencije obožujejo naše orodje — uporabljajo ga za upravljanje oglaševalskih kampanj v obsegu več sto tisoč dolarjev na mesec. V ta namen naš sistem sprocesira več kot 300 tisoč avkcijskih zahtevkov na sekundo v številnih podatkovnih centrih po svetu. Napredno strojno učenje na pridobljenih podatkih omogoča hipno odločanje za vsak zahtevek posebej.

Zemanta se je pred kratkim pridružila podjetju Outbrain, največjemu nativnemu oglaševalskemu podjetju na svetu, katerega poslanstvo je, da vsem omogoči dostop do dobrih vsebin.

Veliko vlagamo v razvoj naše ekipe: pomembna sta nam rast in uspeh vsakega zaposlenega. Vsakega člana ekipe spodbujamo, da sodeluje pri razvoju na vseh stopnjah produkta. We work smart & get things done!

BRONASTI POKROVITELJ

Rtk