12. tekmovanje ACM v znanju računalništva

Institut Jožef Stefan, Ljubljana, 25. marca 2017

Bilten

Bilten 12. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2018

Uredila Janez Brank in Tomaž Hočevar

Avtorji nalog: Nino Bašić, Primož Gabrijelčič, Tomaž Hočevar, Branko Kavšek, Vid Kocijan, Jurij Kodre, Mitja Lasič, Matjaž Leonardis, Mark Martinec, Polona Novak, Jure Slak, Mitja Trampuš, Miha Vuk, Patrik Zajec, Janez Brank.
Rešitve nalog: Janez Brank, Tomaž Hočevar.

Tisk: Collegium Graphicum, d. o. o.
Naklada: 170 izvodov

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

Kataložni zapis o publikaciji (CIP) pripravili v
Narodni in univerzitetni knjižnici v Ljubljani

COBISS.SI-ID=293561088
ISBN 978-961-264-123-8 (PDF)

KAZALO

STRUKTURA TEKMOVANJA

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

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

V lažjih dveh skupinah traja tekmovanje tri ure; tekmovalci lahko svoje rešitve napišejo na papir ali pa jih natipkajo na računalniku, nato pa njihove odgovore oceni temovalna komisija. Naloge v teh dveh skupinah večinoma zahtevajo, da tekmovalec opiše postopek ali pa napiše program ali podprogram, ki reši določen problem. Pri pisanju izvorne kode programov ali podprogramov načeloma ni posebnih omejitev glede tega, katere programske jezike smejo tekmovalci uporabljati. Podobno kot v zadnjih nekaj letih smo tudi letos ponudili možnost, da tekmovalci v prvi in drugi skupini svoje odgovore natipkajo na računalniku; tudi tokrat se je za to odločila večina tekmovalcev, je pa bilo letos nekaj več kot prejšnja leta tudi primerov pisanja odgovorov na papir. Da bi bilo tekmovanje pošteno tudi do morebitnih reševalcev na papir, so bili na računalnikih za prvo in drugo skupino le urejevalniki besedil, ne pa tudi razvojna orodja 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 22.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C#, java in VB.NET.

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

Tekmovalci vseh treh skupin si lahko pri reševanju pomagajo z zapiski in literaturo, 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. 21–24 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 leta 2006 smo v biltenih objavljali izvorno kodo rešitev v C89 (do vključno 2008 tudi v pascalu). Z letošnjim letom prehajamo na C++17 kot glavni jezik izvorne kode v rešitvah, pri tem pa se bomo trudili čim bolj uporabljati stvari, ki so prisotne tudi v C-ju in v starejših različicah C++, da bi bile rešitve razumljive čim več ljudem. V prvi skupini smo letos objavili rešitve 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 187–209.

Podobno kot v zadnjih nekaj letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 20. januarja 2017. 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. 171–174). 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 351 tekmovalcev s 30 šol (vse so bile srednje).

Letos je v Sloveniji potekala tudi srednjeevropska računalniška olimpijada (CEOI 2017), zato v letošnjem biltenu objavljamo tudi rezultate te olimpijade (str. 184) ter slovenske prevode nalog (str. 35–51) in rešitev z nje (str. 103–128).

NASVETI ZA 1. IN 2. SKUPINO

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
#include <stdio.h>

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

Še isti trije primeri v pythonu:

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

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

# Branje standardnega vhoda po vrsticah:
import sys

i = d = 0
```

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

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

Še isti trije primeri v javi:

```
// Branje dveh števil in izpis vsote:
import java.io.*;
import java.util.Scanner;
public class Primer1
{
    public static void main(String[] args) throws IOException
    {
        Scanner fi = new Scanner(System.in);
        int i = fi.nextInt(); int j = fi.nextInt();
        System.out.println(i + " + " + j + " = " + (i + j));
    }
}

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

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

NALOGE ZA PRVO SKUPINO

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

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

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

1. Zaokrožanje temperature

Leta 1966 so pri ameriškem zveznem koordinatorju za meteorologijo ugotovili, da meteorologi po državi v svojih poročilih ne zaokrožajo odčitanih temperatur na cele stopinje vsi na enak način, zato so izdelali priporočilo, ki določa, kako je treba odčitek (realno število) zaokrožiti na najbližje celo število. Pri tem naj se vrednost, ki je točno na sredini med dvema celima številoma, zaokroži vedno navzgor (t.j. proti višji temperaturi, to velja tudi za negativne odčitke). Tako se npr. vrednost 1,5 zaokroži na 2, vrednost $-1{,}5$ pa na -1.

V deželah, kjer merimo temperaturo v stopinjah Celzija (kjer ničla ustreza ledišču vode in so zato meritve blizu ničle pomembne), je v navadi še dodatno pravilo, ki določa, da temperature med $-0{,}5$ (vključno) in 0 (izključno) stopinjami sicer zaokrožimo na 0, a zapišemo kot „-0“, torej z negativnim predznakom.

Napiši program, ki bo s standardnega vhoda prebiral realna števila, jih zaokrožal na cela števila upoštevaje obe gornji pravili in jih izpisoval.

Lahko predpostaviš, da ti je na voljo funkcija Odrezi(x), ki realnemu številu x v argumentu odreže decimalke in vrne celo število. Če želiš, lahko privzameš točen format vhoda, npr. predznak, celi del in dve decimalki (v tem primeru tudi dokumentiraj, kaj si predpostavil o formatu vhoda), ali pa uporabiš standardni način za branje realnih števil v svojem programskem jeziku.

Opomba: če te morda mika, da bi za zaokrožanje uporabil kakšno drugo funkcijo iz svojega izbranega programskega jezika, je treba biti pri tem zelo previden, skrbno prebrati dokumentacijo in utemeljiti odgovor. Večinoma namreč tovrstne funkcije ne izpolnjujejo pogojev te naloge brez dodatne prilagoditve.

2. Najlepši esej

Ministrstvo za lepobesedje je razpisalo natečaj za najlepše eseje. Pri ocenjevanju esejev je določilo naslednje pravilo: esej je lep, če so v njem vse besede dolge od 3 do 8 znakov.

Pomagaj ministrstvu in **napiši program** (ali del programa), ki prebere en esej in izpiše, ali je esej lep (vse besede so znotraj predpisane dolžine). Če esej ni lep, naj izpiše, koliko besed ima krajših od 3 znake in koliko daljših od 8 znakov (za vsak slučaj, če ne bo nobenega eseja, ki bi zadoščal določenemu pogoju, kajti potem se bo posebna komisija odločala med tistimi eseji, ki imajo najmanj besed krajših od 3 znake in daljših od 8 znakov).

Beseda je sestavljena iz malih in velikih črk angleške besede, to so znaki od „`a`“ do „`z`“ in od „`A`“ do „`Z`“. Med besedami so presledki, vejice, pike in drugi znaki, ki niso črke po angleški abecedi. Beseda je vedno v eni vrstici (besede niso nikoli deljene). Tvoja rešitev lahko bere esej s standardnega vhoda ali pa iz datoteke `esej.txt` (karkoli ti je lažje). Posamezne vrstice vhodnega besedila so dolge po največ 100 znakov.

3. Pomanjkanje sendvičev

Vodstvo Caféja Maφja se je odločilo, da lačnim študentom ponudi 6 tipov sendvičev na študentske bone. Prosijo te, da jim pomagaš in **napišeš program** za robota, ki bo prodajal sendviče. Program naj na začetku prebere podatke o trenutni zalogi sendvičev. Nato naj kupce sprašuje po želenih sendvičih in vsakemu sproti postreže, če je sendvič še na zalogi, sicer pa naj izpiše, da ga ni. Program naj se konča, ko kupec zahteva sendvič št. 0. Na koncu naj program izpiše še podatke o najpopularnejšem sendviču (tistem, ki je bil največkrat zahtevan) in izpiše številke sendvičev, ki jih je bilo premalo. Predpostaviš lahko, da v vhodnih podatkih ni napak (npr. zahtevana številka sendviča je vedno od 0 do 6).

Primer:

Program izpiše:	Uporabnik vnese:
`Zaloga sendvicev st. 1:`	`1`
`Zaloga sendvicev st. 2:`	`4`
`Zaloga sendvicev st. 3:`	`0`
`Zaloga sendvicev st. 4:`	`5`
`Zaloga sendvicev st. 5:`	`1`
`Zaloga sendvicev st. 6:`	`365`
`Pozdravljeni, kateri sendvic zelite?`	`1`
`Izvolite sendvic st. 1. Dober tek!`	
`Pozdravljeni, kateri sendvic zelite?`	`3`
`Sendvicev tipa 3 nam je zal zmanjkalo. Vec srece prihodnjic!`	
`Pozdravljeni, kateri sendvic zelite?`	`4`
`Izvolite sendvic st. 4. Dober tek!`	
`Pozdravljeni, kateri sendvic zelite?`	`4`
`Izvolite sendvic st. 4. Dober tek!`	
`Pozdravljeni, kateri sendvic zelite?`	`5`
`Izvolite sendvic st. 5. Dober tek!`	
`Pozdravljeni, kateri sendvic zelite?`	`4`
`Izvolite sendvic st. 4. Dober tek!`	
`Pozdravljeni, kateri sendvic zelite?`	`1`
`Sendvicev tipa 1 nam je zal zmanjkalo. Vec srece prihodnjic!`	
`Pozdravljeni, kateri sendvic zelite?`	`0`
`Premalo je bilo sendvicev st. 1 3.`	
`Najbolj popularni so sendvici st. 4.`	

(*Opomba:* zgornji primer je le za ilustracijo, tvoj program lahko izpis oblikuje tudi malo drugače.)

4. Prehod za pešce

Na prehodu za pešce čez prometno cesto se pogosto dogajajo nesreče, ker pešci prečkajo cesto pri rdeči luči. Preden se poda v urejanje prometa, si mestna uprava želi izvedeti, koliko je takšnih nevzgojenih meščanov. Zato so na prehod namestili števec, ki zazna vsako osebo, ki stopi na cestišče. Na tebi pa je, da napišeš program, ki bo opravljal meritve.

Na voljo je funkcija Dogodek, ki vrne celo število, ko se zgodi eden od dogodkov, ki jih zna sistem meriti. Takšni dogodki so trije. Ko se prižge zelena luč, funkcija vrne vrednost 1. Ko se prižge rdeča luč, vrne vrednost 2. Ko stopi pešec na cestišče, vrne vrednost 3. Funkcija čaka (ne vrne ničesar), dokler se ne zgodi ena od teh stvari. Rdeča in zelena luč se prižigata izmenično.

```
function Dogodek: integer;      { v pascalu }
int Dogodek();                  /* v C/C++ in podobnih jezikih */
def Dogodek(): ...              # v pythonu; vrne int
```

Napiši program, ki se bo vrtel v neskončni zanki in bo vsakič, ko se bo prižgala zelena luč, izpisal, koliko pešcev je pri prejšnji rdeči luči stopilo na cesto. Izpisuješ lahko na zaslon ali v datoteko `pesci.txt`, kar ti je lažje.

5. Pike za tisočice

Pri zapisovanju velikih števil pogosto zaradi preglednosti ločimo skupine treh števk s piko: na primer, število 12345 zapišemo kot „12.345“; število 5379473457 zapišemo kot „5.379.473.457“; število 9876,01234 pa zapišemo kot „9.876,01234“ (brez narekovajev). Kot vidimo iz zadnjega primera, vrivamo pike le levo od decimalne vejice, desno od nje pa ne.[1]

Napiši podprogram (funkcijo) `IzpisStevila(s)`, ki kot parameter dobi niz `s`, ki predstavlja neko število v desetiškem zapisu. V tem nizu nastopajo le znaki od `0` do `9` in mogoče še decimalna vejica. Tvoj podprogram naj dobljeno število izpiše s pikami, kot je bilo to opisano v prejšnjem odstavku.

Tvoj podprogram naj bo takšne oblike:

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

[1] Še dve zanimivi različici te naloge: (*a*) pike vrivamo ne le levo od decimalne vejice, ampak tudi desno od nje: 1234,56789 na primer zapišemo kot „1.234,567.89“; (*b*) vhodno število je celo (nima decimalne vejice), pri tisočicah pa izmenično pišemo pike in vejice: 12345678987654 na primer zapišemo kot „12,345.678,987.654“.

NALOGE ZA DRUGO SKUPINO

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

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

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

1. Zvončki

Imamo podano skladbo — zaporedje tonov, ki ga lahko predstavimo kot zaporedje števil med 0 in n. Podanih imamo m skupin po 10 različnih zvončkov, ki so postavljene ena ob drugi. Vsak zvonček igra nek vnaprej določen ton; v različnih skupinah so lahko zvončki, ki igrajo isti ton. Če stojimo pred skupino i, lahko zaigramo ton poljubnega zvončka iz skupin i, $i-1$ in $i+1$ (z nekaj izjemami na robovih: pri $i = 1$ skupina $i-1$ ne obstaja, pri $i = m$ pa ne obstaja skupina $i+1$). Sicer se moramo premakniti do neke skupine, da tak zvonček dosežemo. (Premaknemo se lahko do poljubne skupine, ne nujno le do sosednje.) Premikom bi se radi izognili. **Opiši postopek**, ki za dano zaporedje tonov ugotovi, s koliko minimalno premiki ga lahko zaigramo. Začetni položaj si lahko izberemo poljubno.

Primer: da bo manj pisanja, si oglejmo primer, v katerem imamo skupine po 3 zvončke namesto po 10 zvončkov, drugače pa je vse enako kot v zgornjem opisu naloge. Recimo, da imamo naslednjih $m = 9$ skupin:

i	1	2	3	4	5	6	7	8	9
zvončki	1	5	7	2	9	8	8	7	6
v sku-	2	1	3	4	4	6	9	0	5
pini i	3	4	8	5	2	1	3	5	3

In recimo, da bi radi zaigrali naslednje zaporedje devetih tonov:

$$7,\ 2,\ 3,\ 9,\ 0,\ 6,\ 5,\ 3,\ 2.$$

Potem je najmanjše potrebno število premikov enako 2. Dobimo ga na primer tako, da začnemo pri $i = 2$, kjer zaigramo tone 7, 2 in 3; nato se premaknemo na $i = 8$, kjer zaigramo tone 9, 0, 6 in 5; in nato se premaknemo na $i = 1$, kjer zaigramo še tona 3 in 2.

2. Rastlinjak

V rastlinjaku je postavljenih nekaj merilnikov temperature zraka, ki svojo vsakokratno meritev prek radijskega oddajnika pošiljajo osrednji nadzorni postaji. Meritve izvajajo in pošiljajo periodično večkrat na uro. Merilniki so samostojni in med seboj niso časovno usklajeni, tudi čas med zaporednimi meritvami ni posebno natančen, tako da se vrstni red prejetih meritev lahko tudi spreminja. Ker se zaradi časovne neusklajenosti ali radijskih motenj lahko zgodi, da bi se kakšna meritev izgubila, je vsak merilnik nastavljen tako, da svojo vsakokratno meritev pošlje trikrat zapored v kratkem časovnem razmaku (od tega se lahko kakšna ponovitev tudi izgubi na poti do sprejemnika, drugih napak pri prenosu pa ni). Vsako oddano sporočilo vsebuje oznako (številko) merilnika in odčitano temperaturo.

Vsak merilnik je opremljen z enolično številko merilnika med 1 in 9 (vseh merilnikov je največ devet). Sprejemna postaja vsako prejeto sporočilo opremi še s časom, ko je bilo sprejeto, in ga zapiše na izhod kot vrstico, ki vsebuje tri polja, ločena s presledkom: čas, številka merilnika in temperatura. Vse ponovitve sporočil so opremljene z enakim časom. Izhod sprejemne postaje je povezan z vhodom v računalnik, tako da program, ki se tam izvaja, lahko sproti bere vrstice prek svojega standardnega vhoda, kot prihajajo iz sprejemnika.

Napiši program, ki bo z vhoda bral zaporedne meritve in sproti izpisoval vsak nov odčitek temperature, skupaj s številko merilnika — takoj, ko se nov odčitek pojavi. Ponovljenih kopij iste meritve naj program ne izpisuje.

Primer vhodnih podatkov:

```
2017-02-23T09:58:38 1 11.0
2017-02-23T09:58:38 1 11.0
2017-02-23T09:58:38 1 11.0
2017-02-23T09:59:00 3 15.1
2017-02-23T09:59:00 2 14.7
2017-02-23T09:59:00 3 15.1
2017-02-23T09:59:00 3 15.1
2017-02-23T09:59:00 2 14.7
2017-02-23T09:59:00 2 14.7
2017-02-23T10:02:21 1 11.0
2017-02-23T10:02:47 3 14.8
2017-02-23T10:02:47 3 14.8
2017-02-23T10:02:47 3 14.8
2017-02-23T10:02:55 2 14.6
```

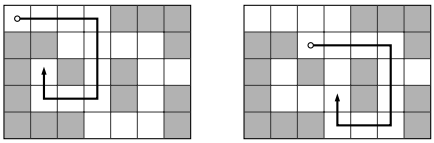
Pripadajoči izpis:

```
1 11.0
3 15.1
2 14.7
1 11.0
3 14.8
2 14.6
```

Tvoja rešitev lahko predpostavi, da so podatki v približno taki obliki, kot jo kaže gornji primer: največ 20 znakov za čas in datum (ta ne more vsebovati presledkov), nato en presledek, en znak s številko merilnika, še en presledek in nato temperatura, zapisana na eno decimalko.

3. Labirint

V nekem trenutku Miha odpre oči in ugotovi, da se nahaja sredi labirinta. Ne ve točno, kje v labirintu se nahaja, najde pa v žepu svoj mobilni telefon. Ker ima Miha predplačniški paket, ugotovi, da ima dobroimetja le še za 100 sporočil SMS. Odloči se, da bo za vsak svoj premik za eno polje v labirintu prijateljem poslal sporočilo SMS. Miha v vsakem sporočilu SMS (ki vsebuje le en znak) prijateljem napiše, v katero smer se je trenutno premaknil (`S` = premaknil sem se v smeri severa, `J` = v smeri juga, `V` = vzhoda, `Z` = zahoda). Čez čas se Miha utrudi hoditi po labirintu in prijateljem pošlje SMS z znakom `@`. Miha se takrat ustavi in čaka na trenutni poziciji. Mihovi prijatelji pa so si med tem priskrbeli zemljevid labirinta in hočejo Mihu pomagati ugotoviti, kje točno v labirintu se nahaja. Poskušajte jim pri tem pomagati in **napišite program**, ki prebere opis labirinta in Mihovih premikov ter izpiše vse možne koordinate Mihovega trenutnega položaja.



Na obeh slikah je enak labirint in pot, ki ustreza enakemu zaporedju premikov (tri korake na vzhod, tri na jug, dva na zahod in enega na sever). Kot vidimo iz slik, je táko zaporedje premikov možno pri dveh različnih začetnih položajih, zato tudi Mihovega sedanjega (končnega) položaja ne moremo enolično določiti.

Vhodni podatki: prva vrstica vhoda vsebuje dve naravni števili h in w (med 3 in 100), ki označujeta velikost labirinta. Labirint ima obliko pravokotne kariraste mreže, sestavljene iz h vrstic in w stolpcev. V sledečih h vrsticah vhoda je predstavljen labirint v obliki tabele ničel in enic. Ničle označujejo prosta polja v labirintu, enice pa neprehodna polja (zidove). Od $(h + 2)$-ge vrstice naprej pa se po vrsti nahaja vsebina vseh sporočil SMS, ki jih je Miha poslal prijateljem (v vsaki vrstici po eno sporočilo).

Izhodni podatki: program naj izpiše koordinate možnih položajev (v poljubnem vrstnem redu), vsakega v svoji vrstici v obliki x y. Pri tem x pomeni številko stolpca (1 je najbolj levi, w je najbolj desni stolpec), y pa številko vrstice (1 je najbolj zgornja vrstica, h je najbolj spodnja vrstica).

Tvoj program lahko za branje in pisanje uporablja standardni vhod in izhod ali pa datoteke (karkoli ti je lažje).

Primer vhodne datoteke:

```
5 7
0 0 0 0 1 1 1
1 1 0 0 0 0 1
1 0 1 0 1 0 0
1 0 0 0 1 0 1
1 1 1 0 0 0 1
V
V
V
J
J
J
Z
Z
S
@
```

Pripadajoča izhodna datoteka:

```
2 3
4 4
```

4. Šifriranje

Tajni agent Janez šifrira svoja sporočila s ključem dolžine 5 znakov po naslednjih pravilih:

- Ključ sestavljajo same male črke angleške abecede (od `a` do `z`).
- S prvo črko ključa šifrira 1., 6., 11., 16. itd. znak sporočila; z drugo črko ključa šifrira 2., 7., 12., 17. itd. znak sporočila; itd.
- Šifriranje posameznega znaka sporočila (recimo z) z neko črko ključa (recimo c) poteka takole: če z ni črka angleške abecede, ostane pri šifriranju nespremenjen; sicer pogledamo, na katerem mestu v abecedi je tista črka c (tako nastane neko število n od 1 do 26), in nato znak z ciklično zamaknemo za n mest naprej po abecedi. Na primer, če je $c =$ `d`, nastane iz njega $n = 4$ in pri cikličnem zamiku za 4 mesta se znak `a` zašifrira v `e`, znak `b` v `f`, itd., znak `v` v `z`, znak `w` v `a`, znak `x` v `b`, znak `y` v `c` in znak `z` v `d`.

Dobil si zašifrirano sporočilo, ključa pa ne poznaš. Toda veš, da se Janez, ki pošilja sporočila, vedno na koncu podpiše „`Lp, Janez`“, tako da bodo to gotovo zadnji znaki v sporočilu. **Napiši program**, ki odšifrira sporočilo in ga izpiše. Tvoj program lahko prebere sporočilo s standardnega vhoda ali pa iz datoteke `sporocilo.txt` (kar ti je lažje). Predpostaviš lahko, da je celotno sporočilo v eni sami vrstici, dolgi največ 10 000 znakov.

Primer sporočila pred in po šifriranju, če za ključ uporabimo besedo `labod`:

Nešifrirano sporočilo:	`Resi se, kdor se more! Lp, Janez`
Znaki ključa:	`labodlabodlabodlabodlabodla`
Šifrirano sporočilo:	`Dfux ef, oppt wq odvq! At, Lprqa`

5. Neskončna pokrajina

Z Daljnega vzhoda smo uvozili poceni 3D skener — napravo, ki zna meriti višino točk na neki pravokotni površini. Žal pa so izdelovalci izdelavo programskega vmesnika prepustili najcenejšemu podizvajalcu (pa še tega so očitno poiskali nekje med učenci lokalne osnovne šole), zato nam je na voljo le ena sama funkcija za merjenje — Visina(x, y). Tej pošljemo celoštevilski koordinati točke, ki ji želimo izmeriti višino, vrne pa nam — seveda — izmerjeno višino. Funkcija je takšne oblike:

```
function Visina(x, y: integer): integer;   { v pascalu }
int Visina(int x, int y);                  /* v C/C++ in podobnih jezikih */
def Visina(x, y): ...                      # v pythonu; vrne int
```

Drugih informacij nismo dobili, zato ne vemo niti, kaj pomenita števili, ki ju pošljemo v funkcijo (v kakšnih enotah sta podani). S poskušanjem smo ugotovili le, da vrednosti ne smeta biti negativni (potem se program sesuje) in da je izmerjena vrednost vedno večja ali enaka 0. S postavljanjem različnih elementov na začetno koordinato smo tudi hitro ugotovili, kako program meri višino. Ne uspe pa nam umeriti koordinatnega sistema, ker je genialni programer funkcijo Visina napisal tako, da ob neveljavnih pozitivnih koordinatah vrne vrednost 0. Zato bomo problem predali tebi.

Napiši podprogram Poisci(ciljnaVisina), ki bo poiskal in izpisal koordinate (x, y) polja, ki je visoko toliko kot podana vrednost (ciljnaVisina). Edino zagotovilo, ki ga imaš, je, da zagotovo obstajata nenegativni koordinati x in y, pri katerih bo funkcija Visina(x, y) vrnila vrednost ciljnaVisina. Če ti je lažje, lahko koordinate zapišeš v datoteko, namesto da bi jih izpisal na zaslon.

Upoštevaj, da je v mreži lahko veliko polj višine 0, torej se ne moreš zanašati na to, da rezultat 0 pomeni, da si funkcijo Visina poklical z neveljavnimi koordinatami. Upoštevaj tudi, da je delovanje skenerja relativno počasno in da zato ne moreš poklicati funkcije Visina kar za vse možne celoštevilske koordinate (torej za vse take koordinate, ki jih je mogoče predstaviti z vrednostjo tipa integer oz. **int**).

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

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

Delovno okolje

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

Na vsakem računalniku imaš na voljo disk `D:\`, v katerem lahko kreiraš svoje datoteke in imenike. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali VB.NET, mi pa jih bomo preverili s 64-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz OpenJDK 9 in s prevajalnikom Mono 4.2 za C# in VB.NET. Za delo lahko uporabiš Lazarus (IDE za pascal), `gcc`/`g++` (GNU C/C++ — command line compiler), `javac` (za javo 1.8), Visual Studio, Eclipse in druga orodja.

Na spletni strani `http://rtk2017.fri1.uni-lj.si/` boš dobil nekaj testnih primerov.

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

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

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

Sistem na spletni strani bo tvojo izvorno kodo prevedel in pognal na več testnih primerih (praviloma desetih). Za vsak testni primer se bo izpisalo, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund ali pa porabil preveč pomnilnika, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru. (Omejitev pomnilnika je 250 MB v pascalu, C in C++ ter 500 MB v javi, C# in VB.NET.)

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.

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

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

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

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

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

Primer vhodne datoteke:

```
123 456
```

Ustrezna izhodna datoteka:

```
5790
```

Primeri rešitev (dobiš jih tudi kot datoteke na `http://rtk2017.fri1.uni-lj.si/`):

- V pascalu:

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

- V C-ju:

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

- V C++:

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

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

- V javi:

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

- V C#:

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

- V Visual Basic.NETu:

```
Imports System.IO

Module Poskus

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

End Module
```

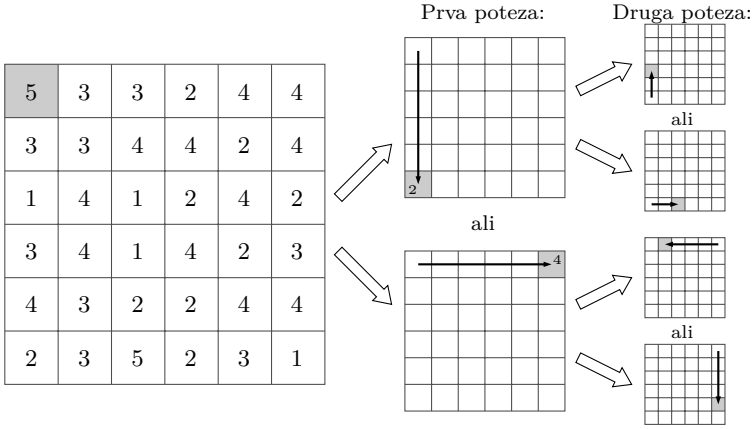
NALOGE ZA TRETJO SKUPINO

1. Back from the Klondike (`klondike.in`, `klondike.out`)

Na igralni plošči $w \times h$ polj je v vsakem polju zapisano neko celo število, večje ali enako 1. Igrati začnemo v zgornjem levem kotu in želimo priti v spodnji desni kot. Skozi vso igro velja, da se iz polja, na katerem se trenutno nahajamo, lahko vedno premaknemo samo v eno od štirih smeri (levo, desno, gor, dol) po naslednjih pravilih:

- Premakniti se moramo vedno za natanko toliko polj, kolikor znaša zapisano število v polju, v katerem ravnokar stojimo.
- V izbrano smer se lahko premaknemo le, če je v tej smeri še najmanj toliko polj do roba plošče, kolikor znaša zapisano število v polju, v katerem stojimo.

Primer (sivo polje označuje naš trenutni položaj):



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

Napiši program, ki prebere opis mreže in izračuna najmanjše število potez, s katerim je mogoče priti iz začetnega v končno polje.

Vhodna datoteka: v prvi vrstici sta dve celi števili, w (širina mreže) in h (višina mreže), ločeni s presledkom. Veljalo bo $1 \leq w \leq 1000$ in $1 \leq h \leq 1000$. Sledi h vrstic, ki opisujejo mrežo; v vsaki od njih je w pozitivnih celih števil, ločenih s po enim presledkom.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše število potez, s katerim je mogoče (po pravilih iz besedila naloge) priti iz zgornjega levega kota mreže v spodnji desni kot mreže. Če takšna pot sploh ne obstaja, izpiši `-1`.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
13
```

2. Trojane (`trojane.in`, `trojane.out`)

Tekmovalcu Rajku s Štajerske se neznansko mudi na računalniško tekmovanje. Zjutraj je zaspal, zdaj pa poskuša ujeti začetek tekmovanja za vsako ceno. No, za skoraj vsako ceno. Rajko ve, da je na Trojanah polno relacijskih radarjev, ki mu bodo pri prehitri vožnji prinesli točke zvestobe in izpraznili bančni račun. Na radiu je slišal podatke o vseh radarjih, zdaj pa ga zanima, v kakšnem najkrajšem času lahko prevozi Trojane, ne da bi dobil kakšno kazen. Rajka sicer ne skrbi za varnost ali cestnoprometne predpise.

Trojane so cesta, dolga m kilometrov. Rajko se na začetku nahaja na začetku (torej na točki 0) in bi rad čim hitreje prišel do konca (torej do točke m). Njegov avto pri najvišji hitrosti en kilometer prevozi v u sekundah. Na Trojanah se nahaja n radarjev. Vsak radar ima začetno točko s_i, končno točko t_i (zanju velja $0 \leq s_i < t_i \leq m$) in v_i, to je čas, ki ga mora avto porabiti na odseku, da je njegova povprečna hitrost dovolj nizka, da ne dobi kazni. Če torej Rajko odsek od s_i do t_i prevozi v manj kot v_i sekundah, bo dobil mastno kazen. **Napiši program**, ki izračuna, najmanj koliko časa potrebuje Rajko, da pride čez Trojane, če noče dobiti nobene kazni. Zaviranje in pospeševanje zanemarimo in predpostavimo, da se zgodi v trenutku. Prav tako zanemarimo morebitne fizikalne zakone in geografske značilnosti, ki bi nasprotovali opisu naloge ali vhodnim podatkom.

Vhodna datoteka: v prvi vrstici so podana cela števila m, n in u, ločena s po enim presledkom. Nato sledi n vrstic, v i-ti od njih (za vsak i od 1 do n) so tri cela števila, s_i, t_i in v_i, ločena s po enim presledkom.

Veljalo bo $n \leq 10^6$, $m \leq 10^9$, $u \leq 10^9$ in $v_i \leq 10^9$. Pri 20 % testnih primerov bo veljalo $n \leq 20$ in $m \leq 20$. Pri nadaljnjih 40 % testnih primerov bo veljalo $n \leq 10^3$ in $m \leq 10^3$.

Izhodna datoteka: vanjo izpiši eno samo število, in sicer najmanjši možni čas, ki ga Rajko potrebuje za vožnjo v skladu z zahtevami naloge.

Primer vhodne datoteke:

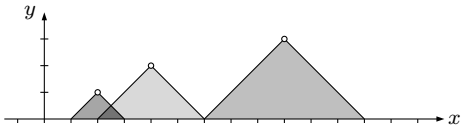
```
8 3 1
1 4 4
3 6 3
5 7 5
```

Pripadajoča izhodna datoteka:

```
12
```

3. V soju žarometov (`zarometi.in`, `zarometi.out`)

Društvo računalnikarjev organizira vsakoletni festival „STOPIE“ za pomoč ljudem, ki še vedno uporabljajo Internet Explorer. Da bi zbrali čim več denarja, so povabili q popularnih glasbenikov, ki so napisali himno festivala in jo bodo ob zaključku skupaj izvedli na odru. Kot odrski tehnik si postavil n reflektorjev na različne položaje in vsakemu nastavil neko intenziteto žarenja. Vsi reflektorji so obrnjeni naravnost proti odru in osvetljujejo stožec s kotom 90°, kot je prikazano na spodnji sliki. Nastopajoči v zaključni točki imajo že predpisane položaje na odru, zanima pa jih, v kako močnem soju žarometov se bo kopal vsak izmed njih. Moč soja žarometov na neki točki na odru je vsota intenzitet vseh žarometov, ki osvetljujejo to točko. (Če se v neki točki stikata dva stožca, jo osvetljujeta oba.) **Napiši program**, ki sprejme postavitev žarometov in njihove intenzitete ter odgovarja na vprašanja nastopajočih.



Vhodna datoteka: v prvi vrstici je dano število žarometov n (zanj velja $1 \le n \le 10^6$). Nato sledi n vrstic; i-ta od njih vsebuje tri cela števila x_i, y_i in c_i, ločena s po enim presledkom, ki predstavljajo x- in y-koordinato i-tega žarometa ter njegovo intenziteto. (Veljalo bo $-10^9 \le x_i \le 10^9$, $0 < y_i \le 10^9$ in $0 < c_i \le 10^9$.) Nato sledi število q, ki predstavlja število poizvedb nastopajočih (veljalo bo $1 \le q \le 10^6$). Sledi q vrstic; j-ta od njih vsebuje celo število p_j, ki pomeni naslednje: pri $j = 1$ je p_j položaj prvega nastopajočega na odru; pri $j > 1$ pa je p_j razlika med položajem j-tega nastopajočega in med rezultatom prejšnje (torej $(j-1)$-ve) poizvedbe. Da dobiš položaj j-tega nastopajočega, moraš torej rezultatu prejšnje poizvedbe prišteti p_j. Za vsak j velja $|p_j| \le 10^{15}$, poleg tega pa za položaj vsakega nastopajočega velja, da je po absolutni vrednosti manjši od $2 \cdot 10^9$.

V 30 % testnih primerov bo veljalo tudi $n \cdot q \le 10^6$.

Izhodna datoteka: za vsako poizvedbo izpiši po eno vrstico, vanjo pa eno samo celo število, namreč vsoto intenzitet vseh žarometov, ki sijejo na položaj nastopajočega, po katerem sprašuje tista poizvedba.

Primer vhodne datoteke:

```
3
4 2 3
9 3 5
2 1 8
6
6
0
8
4
-2
-6
```

Pripadajoča izhodna datoteka:

```
8
5
0
3
8
11
```

4. Najkrajša pot (`pot.in`, `pot.out`)

V Digitalnem kraljestvu se nahaja n mest (označena so s številkami od 1 do n), ki so med sabo povezana z več cestami. (Vsaka cesta neposredno povezuje dve mesti in je dvosmerna.)

Kraljeva palača se nahaja v mestu s številko 1, poletna rezidenca pa v mestu s številko n. Vsako poletje se kralj želi odpraviti iz palače v poletno rezidenco in priti tja po poti, ki uporabi kar se da majhno število različnih cest.

Obenem pa želi vsako leto v poletno rezidenco iti po drugi poti.

V kraljestvu bodo letos zgradili novo cesto. **Napiši program**, ki za dano omrežje cest izračuna, koliko je največje možno število različnih *najkrajših* poti med mestoma 1 in n, potem ko omrežju dodamo novo cesto. (Najkrajša pot je tista, ki uporabi najmanj cest.)

Dve poti sta različni, če se razlikujeta v katerikoli cesti.

Pozor: posamezna mesta so lahko med sabo povezana z več različnimi cestami. Tudi novo cesto lahko dodaš med takim parom mest, ki sta že od prej povezani z

eno ali več cestami.[2] Ni nujno, da se dá po obstoječih cestah sploh priti od palače do poletne rezidence.

Vhodna datoteka: v prvi vrstici sta podani celi števili n in m, ločeni s presledkom. Pri tem je n število mest, m pa je število parov mest, ki so povezana z eno ali več cestami. Nato sledi m vrstic, v vsaki od njih pa so podana tri cela števila a_i, b_i in c_i, ločena s po enim presledkom; to pomeni, da sta mesti a_i in b_i povezani s c_i cestami.

Veljalo bo $2 \leq n \leq 10^6$, $1 \leq m \leq 10^6$ in $1 \leq c_i \leq 10^9$. Pri 20 % testnih primerov bo veljalo $n \leq 10$ in $m \leq 10$, rezultat pa ne bo presegal 10^6. Pri nadaljnjih 20 % testnih primerov bo veljalo $n \leq 10^3$ in $m \leq 10^3$. Pri nadaljnjih 20 % testnih primerov bo veljalo, da je $m = n - 1$ in da med vsakim parom mest obstaja ena pot.

Izhodna datoteka: tvoj program naj izpiše zgolj največje število najkrajših poti med mestoma 1 in n, ki ga lahko dobimo, če dodamo eno povezavo. Vsi testni primeri pri tej nalogi bodo sestavljeni tako, da to število ne bo presegalo 10^{18}.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
18
```

[2] Zanimivo, vendar težjo različico naloge dobimo, če dodamo pravilo, da je lahko med posameznim parom cest le ena neposredna cestna povezava. V tem primeru je torej zagotovljeno, da to velja v začetnem stanju omrežja, ki ga dobimo kot vhod; pri razmišljanju o tem, kje bi se dalo dodati novo cesto (in kaj bi se potem zgodilo s številom najkrajših poti med 1 in n), pa moramo paziti na to, da se lahko novo cesto doda le med takim parom mest, ki doslej še nista bili neposredno povezani s cesto.

5. Listi (`listi.in`, `listi.out`)

Na mizi imamo razmetanih n pravokotnih listov papirja različnih dimenzij. Ker na mizi zasedajo precej prostora, bi radi naredili red in jih zložili na enega ali več kupov. Za vsak list poznamo njegovo širino in višino. Listov na smemo obračati, da ostane besedilo na njih pravilno orientirano. Liste bomo zložili na kup tako, da bodo imeli poravnan levi spodnji kot. Tako je površina, ki jo zaseda posamezen kup listov, enaka produktu največje širine in največje višine listov s tega kupa. Skupna zasedena površina pa je enaka vsoti površin, ki jih zasedajo posamezni kupi. **Napiši program**, ki izračuna, kakšna je minimalna zasedena površina, če liste optimalno zložimo v kupe, in kakšno je minimalno število kupov, s katerim lahko to dosežemo.

Vhodna datoteka: v prvi vrstici datoteke se nahaja število listov $n \leq 10\,000$. Sledi n vrstic, ki opisujejo posamezne liste. Vsak list je opisan s širino w in višino h ($1 \leq w \leq 10^9$ in $1 \leq h \leq 10^9$), ki sta ločeni s presledkom. V 50 % testnih primerov bo veljalo $n \leq 20$.

Izhodna datoteka: vanjo izpiši dve celi števili, ločeni s presledkom: najprej minimalno zasedeno površino in nato minimalno število kupov.

Primer vhodne datoteke:

```
4
20 40
2 50
100 20
10 30
```

Pripadajoča izhodna datoteka:

```
2900 3
```

Komentar: v optimalni rešitvi gornjega primera bosta prvi in četrti list tvorila en kup, drugi in tretji list pa bosta vsak sam na svojem kupu. Tako imamo tri kupe, zasedena površina pa je $20 \cdot 40 + 2 \cdot 50 + 100 \cdot 20 = 2900$.

NALOGE ZA ŠOLSKO TEKMOVANJE

20. januarja 2017

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. Umor

V Nori vasi se je zgodil umor. Detektivi so pri preiskavi ugotovili, da bi jim prišlo prav, če bi vedeli, kdo je umorjenca nazadnje videl živega. Zaslišali so vse vaščane in ugotovili, kdo se je zadnje čase s kom pogovarjal; ti podatki so že urejeni po času pogovora (od najzgodnejših do najkasnejših). Zaradi varovanja zasebnosti je vsak od vaščanov predstavljen le z začetnico svojega imena (predpostavimo, da se pri nobenih dveh vaščanih ne začneta njuni imeni na isto črko). **Napiši program**, ki prebere podatke o pogovorih med vaščani in izpiše, kdo (če sploh kdo) je umorjenca zadnji videl živega. Podatke lahko tvoj program bere s standardnega vhoda ali pa iz datoteke `umor.txt` (karkoli ti je lažje). V prvi vrstici je ime umorjenca, v drugi vrstici število pogovorov med vaščani, nato pa je v vsaki vrstici opisan po en pogovor (začetnici dveh vaščanov, ločeni s presledkom).

Primer: recimo, da imamo naslednje vhodne podatke.

```
B
5
A B
B D
D E
E A
A S
```

Potem lahko zaključimo, da je umorjenca (vaščana `B`) zadnji živega videl `D`.

2. Naraščajoče besede

Napiši program, ki prebere vhodno besedilo in izpiše dolžino najdaljše take besede (v prebranem besedilu), v kateri so črke urejene naraščajoče po abecedi. Besedilo lahko bereš s standardnega vhoda ali pa iz datoteke `besedilo.txt` (karkoli ti je lažje). Predpostaviš lahko, da nobena vrstica vhodnega besedila ni daljša od 100 znakov in da se v besedilu pojavljajo le male črke angleške abecede (od „`a`“ do „`z`“), presledki in ločila.

Primer: recimo, da imamo vhodno besedilo

```
gospod benjamin je bil z njive zagledal tujega gospoda iti proti
gradu. sel je tedaj k potu in gostu naproti. ko sta se sesla,
spoznal ga je brz. vesel ga je pozdravil in mu pomolil roko.
```

Dolžina najdaljše naraščajoče besede v njem je 5 črk (to je beseda `gostu`; druge naraščajoče besede v tem besedilu so še `bil`, `in`, `mu`, `brz` in nekaj enočrkovnih).

3. Popularni dnevi

Za nek strežnik za n zaporednih dni poznamo število obiskovalcev na posamezen dan. Rekli bomo, da je dan d *popularen*, če ima večji obisk kot povprečje njemu najbližjih k dni iz opazovanega n-dnevnega obdobja. (Pri tem d-ja samega ne štejemo med njemu najbližjih k dni.) **Opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki ugotovi, koliko je bilo popularnih dni. Predpostaviš lahko, da imaš vrednosti n in k ter tabelo obiskovalcev po dnevih že podane v nekih spremenljivkah (tako da se ti ni treba ukvarjati z branjem podatkov iz datoteke ali česa podobnega), da je k sodo število in da je k manjši od n. Zaželeno je, da je tvoj postopek čim bolj učinkovit, tako da bo deloval hitro tudi v primerih, ko je k velik.

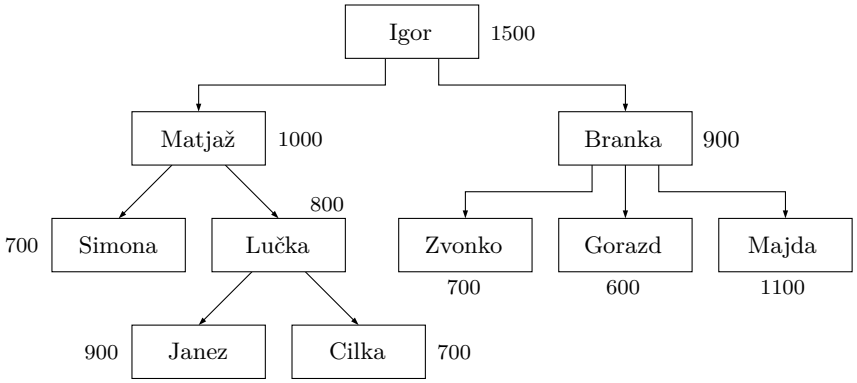
Primer: recimo, da imamo zaporedje $n = 7$ dni z naslednjim številom obiska:

$$10,\ 14,\ 17,\ 20,\ 9,\ 15,\ 14.$$

Pri $k = 2$ so popularni dnevi štirje od teh sedmih (namreč drugi, četrti, šesti in sedmi).

4. Sindikat

V nekem uspešnem slovenskem podjetju so zaposleni urejeni hierarhično. Vsak razen direktorja ima natanko enega nadrejenega. Vsak uslužbenec ima lahko pod seboj več podrejenih. Primer takšne hierarhije (številke ob imenih so njihove plače):



V tem podjetju imajo zelo močan sindikat šefov. Sindikalisti so ugotovili, da višine plač niso pravične. Nedopustno je, da imajo nekateri podrejeni višje plače od svojih nadrejenih! Zato sindikat zahteva, da mora imeti vsak zaposleni vsaj za 100 € višjo plačo od kateregakoli svojega podrejenega.

Lastnik bi rad analiziral podatke, preden se spusti v pogajanja s sindikalisti. Lastnika zanima, koliko dodatnega denarja bi potreboval vsak mesec, če bi ugodil zahtevam sindikata (in to seveda tako, da se plača nikomur ne bi znižala). Pomagaj

mu in **opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki izračuna najmanjšo možno skupno vsoto denarja, ki bi ga potreboval za odpravo krivic. Poleg tega naj tudi poišče (oz. izpiše) imena zaposlenih, ki bi pri tem prejeli povišico.

Predpostaviš lahko, da so zaposleni oštevilčeni z zaporednimi številkami od 1 do n in da so podatki o njih že podani v nekaj tabelah: za zaposlenega s številko k imamo v ime[k] njegovo ime, v placa[k] njegovo plačo in v sef[k] zaporedno številko njegovega nadrejenega (če nadrejenega nima, je tu vrednost 0).

5. 3-d labirint

Dana je pravokotna karirasta mreža $w \times h$ celic. V vsaki celici stoji steber; stebri so različnih višin in te višine so podane v tabeli: $v[x][y]$ je višina stebra v celici (x, y). Po vrhovih teh stebrov se želimo sprehoditi. S stebra lahko prestopimo na sosednji steber le, če se stebra po višini razlikujeta največ za eno enoto in če se celici, v katerih tadva stebra stojita, dotikata z eno stranico. Če se celici stikata le z vogalom, prehod med njunima stebroma ni mogoč (vsaj ne neposredno). **Opiši postopek**, ki ugotovi, ali je možno priti z najnižjega stebra do najvišjega.

NALOGE S CEOI 2017

Društvo ACM Slovenija je letos sodelovalo tudi pri organizaciji srednjeevropske olimpijade v računalništvu (Central European Olympiad in Informatics — CEOI), ki je potekala na Fakulteti za računalništvo in informatiko v Ljubljani od 11. do 14. julija 2017. Uradna besedila nalog in rešitev (v angleščini) so objavljena na spletni strani olimpijade, `http://ceoi2017.acm.si/`, v pričujočem biltenu pa objavljamo besedila nalog in rešitev v slovenščini.

Preden si ogledamo naloge, še nekaj opomb o načinu tekmovanja in ocenjevanja na CEOI. Tekmovanje poteka podobno kot v tretji skupini na našem državnem tekmovanju: tekmovalci rešujejo naloge na računalnikih, svoje rešitve pa oddajajo na ocenjevalni strežnik, ki jih sproti testira in ocenjuje. Tekmovanje obsega dva tekmovalna dneva (letos sta bila to 12. in 14. julij), na vsakem od njiju pa so tekmovalci reševali po tri naloge in imeli za to po pet ur časa. Vsaka naloga je razdeljena na podnaloge, ki se lahko med seboj razlikujejo po omejitvah, ki veljajo za vhodne podatke. Posamezna podnaloga je vredna določeno število točk (ki je navedeno v besedilu naloge); tekmovalčeva rešitev dobi te točke, če pravilno reši vse testne primere pri tej podnalogi, sicer pa ne dobi pri tej podnalogi nobenih točk. Odštevanja točk zaradi večkratnih oddaj pa na CEOI ni.

Poleg nalog z obeh tekmovalnih dni si bomo ogledali tudi rezervne (neuporabljene) naloge in naloge s poskusnega tekmovanja (11. julija), ki je bilo namenjeno predvsem temu, da se preizkusi, če je s tekmovalnim okoljem vse v redu, in da se tekmovalci privadijo na to okolje.

POSKUSNO TEKMOVANJE

1. Jenga

(*Omejitev časa: 4 s.* *Omejitev pomnilnika: 512 MB.*)

Cilj igre Jenga je odstranjevanje posameznih kvadrov iz stolpa, ne da bi ga pri tem podrli. Tudi če poznate pravila igre, natančno preberite opis, saj v tej nalogi obravnavamo igro z malenkost spremenjenimi pravili.

Na začetku igre je stolp zgrajen iz več nadstropij, ki jih sestavljajo trije kvadri, postavljeni en poleg drugega. V vsakem nadstropju so kvadri orientirani pravokotno glede na tiste v spodnjem in zgornjem nadstropju. Dva igralca se izmenjujeta pri odstranjevanju posameznih kvadrov iz stolpa. Igralec, ki je na potezi, mora odstraniti en kvader od koderkoli, razen iz vrhnjega nadstropja, in ga dodati v vrhnje nadstropje. Če je vrhnje nadstropje polno (ima 3 kvadre), začne igralec graditi novo vrhnje nadstropje s tem, da postavi kvader na tiste iz dosedanjega vrhnjega nadstropja. Igralec, s svojo potezo ki poruši stolp, izgubi. Igralca se smeta dotikati samo kvadra, ki ga odstranita in postavita na vrh stolpa. Kakršnokoli drugo dotikanje ali premikanje kvadrov ni dovoljeno.

Kot trenutni prvak v igri Jenga si tako spreten, da lahko odstraniš poljuben kvader, če je to seveda fizično mogoče. Enako spretni pa so tudi tvoji nasprotniki. Nov izzivalec ti želi odvzeti ta prestižni naslov. Je enako spreten kot ti in ne bo porušil stolpa po nesreči. Ga lahko premagaš?

Na začetku igre ima stolp N nadstropij, ki jih oštevilčimo od 1 do N. Vsako nadstropje ima 3 kvadre, označene z 1, 2 in 3 (pri čemer je kvader številka 2 sredinski). Če posamezno nadstropje nima nobenega kvadra ali pa ima samo kvader 1 ali kvader 3, se stolp zruši. Sicer predpostavimo, da se stolp ne podre. V tej nalogi bomo ignorirali težišče in druge fizikalne omejitve.

Kot trenutni prvak lahko izbiraš, ali boš začel ti ali tvoj nasprotnik. Ko si na potezi, moraš odstraniti en kvader in ga postaviti na vrhnje nadstropje. Igra se konča, ko igralec poruši stolp.

Interakcija

Naloga je interaktivna. Tvoj program naj bere s standardnega vhoda in piše na standardni izhod. Program mora zmagati igro proti ocenjevalnemu programu.

Prva vrstica vhoda vsebuje število nadstropij N. Tvoja prva vrstica izhoda naj bo „`first`“, če želiš začeti prvi, sicer naj bo „`second`“.

Ko je na potezi tvoj program, naj izpiše dve s presledkom ločeni števili, L in B. To pomeni, da boš odstranil kvader B iz nadstropja L in ga položil na vrh. Ocenjevalni program (nasprotnik) bo svoje poteze oblikoval na enak način. Tvoj program naj se konča, ko se stolp poruši.

Igra se mora končati s potezo, ki poruši stolp. Če se tvoj program po tem ne konča, je lahko ocenjen s prekoračitvijo časovne omejitve. Ne pozabi izprazniti medpomnilnika standardnega izhoda (*flush*) po vsakem izpisu. Tvoj program prejme točke za vsako dobljeno igro.

Omejitve

Za vse podnaloge veljajo naslednje omejitve:

- $2 \leq N \leq 1000$

Podnaloga 1 (20 točk)

- $N = 2$

Podnaloga 2 (20 točk)

- $N \leq 10$

Podnaloga 3 (20 točk)

- $N \leq 40$

Podnaloga 4 (40 točk)

- Brez dodatnih omejitev.

Primer interakcije

Ocenjevalec	*Program*
`2`	
	`first`
	`1 1`
`1 3`	
	`2 2`
`2 1`	

Komentar. Primer prikazuje veljaven izpis programa. Bodite pozorni na to, da sta vhod in izhod prepletena. Levi stolpec prikazuje vhod tvojemu programu, desni pa izpis tvojega programa.

Igra se začne z dvonadstropnim stolpom. Tvoj program se odloči, da bo začel. Odstraniš kvader 1 iz prvega nadstropja. Nasprotnik (ocenjevalec) odstrani kvader 3 iz prvega nadstropja. Odstraniš kvader 2 iz drugega nadstropja. Nasprotnik ne sme odstraniti kvadra z vrhnjega nadstropja, če odstrani katerikoli drug kvader, pa se bo stolp porušil. Odloči se za kvader 1 v drugem nadstropju in s to potezo izgubi igro.

2. Množenje

(*Omejitev časa: 2 s.* *Omejitev pomnilnika: 128 MB.*)

Napiši program, ki izračuna produkt dveh nenegativnih celih števil A in B. Števili sta zapisani v desetiškem sistemu. Število A je dolgo N števk, število B pa M.

Omejitve

Za vse podnaloge veljajo naslednje omejitve:

- $1 \leq N, M \leq 50\,000$

Podnaloga 1 (20 točk)

- $N, M \leq 4$

Podnaloga 2 (20 točk)

- $N, M \leq 9$

Podnaloga 3 (30 točk)

- $N, M \leq 5\,000$

Podnaloga 4 (30 točk)

- Brez dodatnih omejitev.

Vhod. V prvi vrstici sta podani dolžini N in M, ločeni s presledkom. V drugi vrstici je podano število A, v tretji pa B. Števili nimata vodilnih ničel.

Izhod. Izpiši produkt števil A in B brez vodilnih ničel.

Primer vhodne datoteke:

```
3 4
123
4567
```

Pripadajoča izhodna datoteka:

```
561741
```

Še en primer vhodne datoteke:

```
3 1
100
0
```

Pripadajoča izhodna datoteka:

```
0
```

3. Muzej

(*Omejitev časa: 3 s.* *Omejitev pomnilnika: 1 MB.*)

Turist je vstopil v muzej, ki hrani zbirko čiste pitne vode z različnih delov sveta. Na srečo gre zgolj za začasno razstavo z namenom opozarjanja na problem pitne vode, morda pa bo nekoč v prihodnosti taka razstava trajna.

Muzej sestavlja n sob (označene od 1 do n), ki so med seboj povezane z vrati in hodniki. Vsak hodnik neposredno povezuje dve sobi. Organizacija muzeja je taka, da obstaja med vsakim parom sob natanko ena pot, ki gre pri tem lahko tudi skozi eno ali več vmesnih sob. Turist se trenutno nahaja v sobi z. Pri sebi ima načrt muzeja in tako ve, da i-ti hodnik povezuje sobi a_i in b_i ter da hoja vzdolž hodnika zahteva c_i časa.

Rad bi obiskal d različnih sob (vključno z izhodiščno sobo z). V vsaki sobi se bo zadržal zanemarljivo malo časa. Vseeno mu je, v kateri sobi zaključi svoj ogled. Kakšen je najkrajši čas, v katerem lahko to doseže?

Omejitve

Za vse podnaloge veljajo naslednje omejitve:

- $1 \leq n \leq 10\,000$
- $1 \leq d, z \leq n$
- $1 \leq a_i, b_i \leq n$
- $0 \leq c_i \leq 10\,000$

Podnaloga 1 (20 točk)

- $n \leq 20$

Podnaloga 2 (25 točk)

- $d \leq 100$
- vsaka soba ima največ 3 sosednje sobe

Podnaloga 3 (35 točk)

- $d \leq 100$

Podnaloga 4 (20 točk)

- Brez dodatnih omejitev.

Vhod. V prvi vrstici se nahajajo cela števila n, d in z. Naslednjih $n-1$ vrstic opisuje hodnike med sobami s števili a_i, b_i in c_i, ki predstavljajo hodnik med sobama a_i in b_i, za prehod katerega porabi turist c_i časa.

Izhod. Izpiši najkrajši čas, v katerem lahko turist obišče d sob.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
29
```

Še en primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
0
```

PRVI TEKMOVALNI DAN

1. Enosmerne ceste

(*Omejitev časa: 3 s.* *Omejitev pomnilnika: 256 MB.*)

Nekoč je bila dežela z n mesti, povezanimi z m dvosmernimi cestami. Zaradi tehnološkega razvoja so bila vozila vse hitrejša in večja, to pa je povzročilo težavo — ceste so postajale preozke, da bi se lahko na njih srečali dve vozili, ki vozita v nasprotnih smereh. To težavo so se odločili rešiti s predelavo vseh cest v enosmerne in enopasovne.

Slaba stran tega, da ceste postanejo enosmerne, je, da se zaradi tega lahko zgodi, da nekatera mesta niso več dosegljiva iz nekaterih drugih mest, čeprav so prej bila. Vlada je pripravila seznam pomembnih parov mest, za katere mora biti mogoče iz prvega mesta v paru priti do drugega mesta v paru. Tvoja naloga je določiti, v katero smer naj po novem poteka promet na vsaki cesti. Zagotovljeno je, da rešitev obstaja.

Pri nekaterih cestah ni nobenih alternativ glede tega, v katero smer moramo speljati promet po njih, če hočemo dobiti dopustno rešitev; promet mora nujno teči od prvega mesta proti drugemu (temu bomo rekli „desno“ in označili s črko `R`) ali pa mora nujno teči od drugega mesta proti prvemu (temu bomo rekli „levo“ in označili s črko `L`). Pri nekaterih cestah pa obstajajo tako rešitve, pri katerih je promet na tej cesti usmerjen v levo, kot tudi rešitve, pri katerih je promet na tej cesti usmerjen desno. Take ceste moraš predstaviti s črko `B`, ki torej pove, da je promet mogoče speljati v eno ali drugo smer.

Izpiši niz m znakov; i-ti znak tega niza naj bo:

- `R`, če je i-ta cesta v vsaki dopustni rešitvi usmerjena desno;
- `L`, če je i-ta cesta v vsaki dopustni rešitvi usmerjena levo;
- `B`, če obstajajo tako dopustne rešitve, pri katerih je i-ta cesta usmerjena levo, kot tudi dopustne rešitve, pri katerih je i-ta cesta usmerjena desno.

Omejitve
Za vse podnaloge veljajo naslednje omejitve:

- $1 \leq n, m, p \leq 100\,000$
- $1 \leq a_i, b_i, x_i, y_i \leq n$

Podnaloga 1 (30 točk)

- $n, m \leq 1000$
- $p \leq 100$

Podnaloga 2 (30 točk)

- $p \leq 100$

Podnaloga 3 (40 točk)

- Brez dodatnih omejitev.

Vhod. V prvi vrstici je število mest n in število cest m. Sledi m vrstic, ki opisujejo ceste; vsaka od njih vsebuje par števil a_i in b_i, ki povesta, da obstaja neposredna cesta od mesta a_i do mesta b_i. Lahko se zgodi, da je isti par mest povezan z več cestami, ali pa celo to, da se cesta začne in konča v istem mestu.

V naslednji vrstici je p, število parov mest, ki morajo biti dosegljiva. Sledi še p vrstic, ki vsebujejo pare mest x_i in y_i; vsaka taka vrstica predstavlja zahtevo, da se mora dati iz mesta x_i priti v mesto y_i.

Izhod. Izpiši niz m znakov, kot je opisano v opisu naloge.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
BBRBBL
```

Komentar. Pokažimo, da je mogoče peto cesto, $(1, 3)$, usmeriti v poljubno smer. Dve dopustni rešitvi (usmeritvi vseh cest) z različno usmeritvijo pete ceste sta `LLRLRL` in `RLRRLL`.

2. Zanesljiva stava

(*Omejitev časa: 2 s.* *Omejitev pomnilnika: 128 MB.*)

Pri stavah je sreča ena od bistvenih stvari. Nekateri ljudje izboljšajo svoje možnosti in zaslužek z dobrim poznavanjem tega, na kar stavijo. Mi pa bomo ubrali drugačno pot.

Različni pobiralci stav ponujajo za isti izid različna *razmerja*. (*Razmerje* x pomeni, da če staviš 1 evro in pravilno napoveš izid, dobiš x evrov. Če izid napoveš

narobe, ne dobiš ničesar, vložek pa seveda ne glede na izid v vsakem primeru plačaš.) Kaj če lahko zvito skleneš več stav tako, da boš zagotovo prišel do dobička? Seveda si želiš, da bi bil ta zanesljivi dobiček čim višji.

Dogodek, na katerega hočemo staviti, ima dva možna izida. Imamo n pobiralcev stav, ki ponujajo različna razmerja. Označimo z a_i razmerje, ki ga ponuja i-ti pobiralec na prvi izid, z b_i pa razmerje, ki ga ta pobiralec ponuja na drugi izid. Skleneš lahko stave na poljubno kombinacijo tako ponujenih razmerij, lahko celo pri istem pobiralcu staviš na oba izida. Vendar pa mora biti vsaka od tvojih stav za natanko 1 evro in pri posameznem pobiralcu ne moreš po večkrat staviti na isti izid.

Če pride do prvega izida, boš od vsakega pobiralca i, pri katerem si stavil na ta izid, dobil a_i evrov. Podobno, če pride do drugega izida, boš dobil b_i evrov od vsakega pobiralca, pri katerem si stavil na drugi izid. Seveda si v obeh primerih že plačal po 1 evro za vsako stavo, ki si jo sklenil.

Kolikšen je največji *zagotovljeni* dobiček (torej neodvisen od izida), če svoje stave skleneš optimalno?

Omejitve

Za vse podnaloge veljajo naslednje omejitve:

- $1 \leq a_i, b_i \leq 1000$
- $1 \leq n \leq 100\,000$

Podnaloga 1 (20 točk)

- $n \leq 10$

Podnaloga 2 (40 točk)

- $n \leq 1000$

Podnaloga 3 (40 točk)

- Brez dodatnih omejitev.

Vhod. V prvi vrstici je število pobiralcev stav, n. Sledi n vrstic, od katerih i-ta vsebuje realni števili a_i in b_i, ločeni s presledkom — to sta razmerji, ki ju i-ti pobiralec ponuja za stave na prvi oz. drugi izid. Razmerja bodo podana na največ 4 decimalke.

Izhod. Izpiši največji zagotovljeni dobiček na natanko 4 decimalke.
Decimalna števila (*floating point*) lahko izpišeš s sledečimi ukazi:

- C in C++: `printf("%.4lf", (double) x);`
- Java: `System.out.printf("%.4lf", x);`
- Pascal: `WriteLn(x:0:4);`
- Python 3: `print("%.4lf"% x)`
- C#: `Console.WriteLine(String.Format("{0:0.0000}", x));`

Primer vhodne datoteke:

```
4
1.4 3.7
1.2 2
1.6 1.4
1.9 1.5
```

Pripadajoča izhodna datoteka:

```
0.5000
```

Komentar. Najboljša strategija pri tem primeru je, da pri prvem pobiralcu stavimo na drugi izid, pri tretjem in četrtem pobiralcu pa na prvi izid. Če se zgodi prvi izid, bomo zaslužili $1{,}6 + 1{,}9 - 3 = 0{,}5$, pri drugem izidu pa $3{,}7 - 3 = 0{,}7$. Tako imamo torej zagotovljen dobiček 0,5 evrov ne glede na izid.

3. Mišolovka

(*Omejitev časa: 5 s.* *Omejitev pomnilnika: 512 MB.*)

Slonček Dumbo ima velik labirint z n sobami, oštevilčenimi od 1 do n, in z $n-1$ prehodi med njimi, ki so speljani tako, da je možno iz katerekoli sobe priti v katerokoli drugo. Na njegovo nesrečo se je v njegov labirint prikradla miš. Dumbo se miši zelo boji, zato je v sobo t postavil mišolovko. Miš se sobe z mišolovko seveda izogiba, zato si mora Dumbo izmisliti strategijo, kako jo zvabiti vanjo. Miš neprestano teka po labirintu in se nikoli ne ustavi, razen v primeru, če se ne more premakniti nikamor več. Dumbo je opazil, da miš za seboj pušča umazano sled iztrebkov v vsakem prehodu, ki ga uporabi, tako umazanih prehodov pa ne uporablja več. Dumbo prehode lahko očisti ali jih zazida. Z zazidavo oziroma čiščenjem prehodov želi prisiliti miš, da bo pritekla v sobo z mišolovko. To bi rad opravil s čim manjšim številom potez, ker se v prisotnosti miši počuti zelo neudobno.

To lahko opišemo kot igro dveh igralcev. Miš poskuša število Dumbovih potez čim bolj povečati, Dumbo pa želi zmagati s čim manjšim številom potez. Prvi je na potezi Dumbo. Ko je na potezi, lahko ali očisti en prehod ali en prehod zazida ali pa ne naredi ničesar. Zazida lahko tako čist kot tudi umazan prehod, zazidanega prehoda pa ne more več sprostiti. Koraki, v katerih Dumbo ne stori ničesar, se ne upoštevajo v seštevku opravljenih potez. Ko je na potezi miš, se preko čistega in nezazidanega prehoda premakne v eno od sosednjih sob. Če tak prehod ne obstaja, se miš ne premakne.

Na začetku so vsi prehodi čisti, miš je v sobi m, mišolovka je v sobi t, na potezi pa je Dumbo. Katero je najmanjše število potez (čiščenj ali zazidav prehoda), če oba igralca igrata optimalno (Dumbov cilj je čim manjše število potez, cilj miši pa ravno obratno)?

Omejitve

Za vse podnaloge veljajo naslednje omejitve:

- $1 \leq n, t, m \leq 10^6$

Podnaloga 1 (20 točk)

- $n \leq 10$

Podnaloga 2 (25 točk)

- Prehod med sobama m in t zagotovo obstaja.

Podnaloga 3 (20 točk)

- $n \leq 1000$

Podnaloga 4 (35 točk)

- Brez dodatnih omejitev.

Vhod. V prvi vrstici so cela števila n, t in m, ločena s presledki. Sledi $n-1$ vrstic. V vsaki vrstici sta podana a_i in b_i (ločena s presledkom), ki označujeta prehod med sobama a_i in b_i.

Upoštevaj, da je število podatkov na vhodu lahko zelo veliko.

Izhod. Tvoj program naj izpiše najmanjše možno število Dumbovih potez, če oba igralca igrata optimalno.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
4
```

Komentar. En možen scenarij:

- Dumbo zazida prehod med sobama 4 in 7.
- Miš se premakne v sobo 6. Prehod med sobama 4 in 6 je zdaj umazan.
- Dumbo zazida prehod med sobama 6 in 8.
- Miš se ne more premakniti.
- Dumbo počisti prehod med sobama 4 in 6.
- Miš se premakne v sobo 4. Prehod med sobama 4 in 6 je umazan.
- Dumbo zazida prehod med sobama 2 in 3.
- Miš se premakne v sobo 2. Prehod med sobama 2 in 4 je umazan.
- Dumbo ne naredi ničesar.
- Miš se lako premakne le v sobo 1, kjer se ujame v mišolovko.

Dumbo je naredil 4 poteze.

DRUGI TEKMOVALNI DAN

1. Gradnja mostov

(*Omejitev časa: 3 s.* *Omejitev pomnilnika: 128 MB.*)

Na široki reki stoji n stebrov potencialno različnih višin. Razporejeni so v ravni vrsti od enega brega do drugega. Zgraditi želimo most, stebre pa uporabiti za podporo. Izbrali bomo le neko podmnožico vseh stebrov in povezali njihove vrhove z odseki mostu. Prvi in zadnji steber morata biti nujno vključena v to podmnožico.

Cena gradnje mostu med stebroma i in j je $(h_i - h_j)^2$, kjer je h_i višina i-tega stebra, saj bi se radi ognili neravnim odsekom. Stebre, ki jih med gradnjo ne bomo uporabili, moramo porušiti, saj motijo rečni promet. Cena odstranitve i-tega stebra je w_i. Ta cena je lahko negativna — nekatere stranke so nam pripravljene plačati, da se znebimo nekaterih stebrov. Vse višine h_i in cene w_i so cela števila.

Kakšna je najnižja cena izgradnje mostu, ki povezuje prvi in zadnji steber?

Omejitve
Za vse podnaloge veljajo naslednje omejitve:

- $2 \leq n \leq 10^5$
- $0 \leq h_i \leq 10^6$
- $0 \leq |w_i| \leq 10^6$

Podnaloga 1 (30 točk)

- $n \leq 1000$

Podnaloga 2 (30 točk)

- optimalna rešitev poleg prvega in zadnjega vsebuje največ 2 vmesna stebra
- $|w_i| \leq 20$

Podnaloga 3 (40 točk)

- Brez dodatnih omejitev.

Vhod. V prvi vrstici je podano število stebrov, n. V drugi vrstici so podane višine stebrov h_i, po vrsti, ločene s presledki. V tretji vrstici so podane cene odstranitve stebrov w_i, v istem vrstnem redu.

Izhod. Izpiši minimalno ceno izgradnje mostu. Upoštevaj, da je lahko cena negativna.

Primer vhodne datoteke:

```
6
3 8 7 1 6 6
0 -1 9 1 2 0
```

Pripadajoča izhodna datoteka:

```
17
```

2. Palindromske razdelitve
(*Omejitev časa: 10 s.* *Omejitev pomnilnika: 128 MB.*)

Razdelitev niza s je zaporedje enega ali več nepraznih podnizov s, ki se ne prekrivajo (recimo jim $a_1, a_2, \ldots, a_d$), tako da jih lahko zlepimo v s: $s = a_1 + a_2 + \ldots + a_d$. Recimo tem podnizom *koščki* in definirajmo *dolžino* take razdelitve kot število koščkov, d.

Razdelitev lahko predstavimo kot niz tako, da vsak košček zapišemo v oklepaje. Niz „`decode`“ lahko na primer razdelimo kot `(d)(ec)(ode)`, `(d)(e)(c)(od)(e)`, `(decod)(e)`, `(decode)`, `(de)(code)` ali še na mnogo drugih načinov.

Razdelitev je *palindromska*, če njeni koščki sestavljajo palindrom, ko jih obravnavamo kot nedeljive enote. Edini palindromski razdelitvi niza „`decode`“ sta `(de)(co)(de)` in `(decode)`. Iz primera je tudi razvidno, da ima vsaka beseda vsaj eno trivialno palindromsko razdelitev dolžine ena.

Tvoja naloga je izračunati največje možno število koščkov v palindromski razdelitvi.

Omejitve
Naj bo n dolžina vhodnega niza s. Za vse podnaloge veljajo naslednje omejitve:

- $1 \leq t \leq 10$
- $1 \leq n \leq 10^6$

Podnaloga 1 (15 točk)

- $n \leq 30$

Podnaloga 2 (20 točk)

- $n \leq 300$

Podnaloga 3 (25 točk)

- $n \leq 10\,000$

Podnaloga 4 (40 točk)

- Brez dodatnih omejitev.

Vhod. V prvi vrstici je podano število testnih primerov t. Naslednjih t vrstic opisuje posamezne testne primere. Vsak testni primer je ena beseda s, sestavljena iz malih črk angleške abecede. Na vhodu ne bo nobenih presledkov.

Izhod. Za vsakega od t testnih primerov izpiši eno število: dolžino najdaljše palindromske razdelitve vhodne besede s.

Primer vhodne datoteke:

```
4
bonobo
deleted
racecar
racecars
```

Pripadajoča izhodna datoteka:

```
3
5
7
1
```

3. Lov

(*Omejitev časa: 4 s.* *Omejitev pomnilnika: 512 MB.*)

Maček Tom zopet lovi Jerryja! Jerry skuša pridobiti nekaj prednosti pred Tomom s tem, da teče skozi skupine golobov, skozi katere mu Tom težje sledi. Jerry je prispel v ljubljanski park Tivoli. V parku je n, kipov oštevilčenih od 1 do n, in $n-1$ nesekajočih se potk, ki povezujejo pare kipov tako, da se je po njih možno sprehoditi od katerega koli kipa do katerega koli drugega. Okrog vsakega kipa je zbrana gosta skupina golobov — okrog i-tega kipa je p_i golobov. Jerry ima v žepu d krušnih drobtinic.

Če pri nekem kipu drobtinico vrže na tla, golobi s sosednjih kipov (tistih, ki so s tem kipom neposredno povezani s potko) takoj priletijo k temu kipu, da bi drobtinico pojedli. Posledično se število golobov p pri tem in vseh sosednjih kipih spremeni. Vse se zgodi v sledečem vrstnem redu: najprej Jerry prispe h kipu i in naleti na p_i golobov. Potem na tla vrže drobtinico in se odpravi naprej. Golobi odletijo od sosednjih kipov do kipa i, predno Jerry prispe do naslednjega kipa, tako da jih Jerry pri naslednjem kipu ne sreča.

Jerry lahko v park vstopi pri kateremkoli kipu, teče po kateremkoli zaporedju potk, vendar po vsaki največ enkrat, in nato zapusti park pri kateremkoli kipu. Ko Jerry zapusti park, vanj vstopi Tom in ga prečka po isti poti. Z metanjem drobtinic

želi Jerry maksimizirati razliko med številom golobov, ki jih bo srečal Tom, in tistimi, ki jih je srečal sam. V skupno vsoto števila golobov, ki jih je srečal Jerry, štejemo samo golobe, ki so pri kipu, tik preden do njega prispe Jerry. Za dodatno razlago glej komentar pri spodnjem primeru. Jerry lahko uporabi največ v drobtinic.

Omejitve
Za vse podnaloge veljajo naslednje omejitve:

- $1 \leq n \leq 10^5$
- $0 \leq d \leq 100$
- $0 \leq p_i \leq 10^9$

Podnaloga 1 (20 točk)

- $1 \leq n \leq 10$

Podnaloga 2 (20 točk)

- $1 \leq n \leq 1000$

Podnaloga 3 (30 točk)

- Optimalna pot se prične pri kipu 1.

Podnaloga 4 (30 točk)

- Brez dodatnih omejitev.

Vhod. V prvi vrstici sta števili kipov n in razpoložljivih drobtinic d. V naslednji vrstici je n celih števil $p_1, \ldots, p_n$, ločenih s presledki. Naslednjih $n-1$ vrstic vsebuje pare števil a_i in b_i, ki označujejo potke med kipi a_i in b_i.

Izhod. Izpiši eno število, in sicer največjo razliko med številom golobov, ki jih sreča Tom, in številom golobov, ki jih sreča Jerry.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
36
```

Komentar. Ena izmed možnih rešitev je naslednja: Jerry vstopi v park pri kipu številka 6, kjer naleti na 5 golobov. Nato spusti drobtinico. p_6 se zdaj poveča na 27, p_5, p_7, p_8 in p_9 pa postanejo 0. Nato priteče do kipa 7, kjer ni nič golobov. Spusti drugo drobtinico. p_7 se poveča na 41, p_2, p_4, p_6 in p_{10} pa postanejo 0. Nato Jerry

zapusti park. Na svoji poti je srečal skupno $5 + 0 = 5$ golobov. Tom mu sledi po isti poti, vendar sreča $p_6 + p_7 = 0 + 41 = 41$ golobov. Razlika je $41 - 5 = 36$.

REZERVNE NALOGE

1. Laser Tag

(*Omejitev časa: 5 s.* *Omejitev pomnilnika: 512 MB.*)

S prijatelji v zapuščenem skladišču igrate igro Laser Tag (streljanje z laserskimi pištolami). Razdelili ste se v dve skupini. Tvoja ekipa se je razporedila na južno stran skladišča, nasprotna pa na severno. Po prostoru med vami so razporejene ovire. Tekom igre se pojavljajo še nove ovire. Vsaka ovira je plošča, ki stoji pokonci tako, da je s ploskvijo obrnjena proti igralcem obeh ekip. Če bi načrt stavbe narisali iz ptičje perspektive, bi bilo skladišče pravokotnik, poravnan s koordinatnima osema, ovire pa bi bile daljice, vzporedne abscisni osi. Pri tem se i-ta ovira nahaja na razdalji d_i od južnega zidu. Njeno levo krajišče je oddaljeno a_i enot od zahodnega zidu skladišča, desno krajišče pa je oddaljeno b_i enot od zahodnega zidu. Ovira se pojavi ob času t_i. Nobeni dve oviri se ne sekata, prekrivata ali dotikata.

Pridobiti želiš nekaj taktične prednosti, zato te zanima, kako daleč v severni smeri bodo potovali streli, izstreljeni ob času u_j na razdalji y_j od južnega in x_j od zahodnega zidu. Strel, izstreljen proti krajišču i-te ovire (torej $x_j = a_i$ ali $x_j = b_i$) bo zadel i-to oviro. Če strel leti skozi oviro, ki se pojavi ob istem času, kot je bil izstreljen strel ($u_j = t_i$), strel zadane to oviro. Če je strel izstreljen iz pozicije, kjer se nahaja ovira ($y_j = d_i$ in $a_i \leq x_j \leq b_i$), jo strel zadane — predstavljaj si, da stojiš ravno za oviro. Spiši program, ki bo učinkovito odgovarjal na poizvedbe takega tipa.

Omejitve

Za vse podnaloge veljajo naslednje omejitve:

- $1 \leq n \leq 100\,000$
- $1 \leq q \leq 300\,000$
- $0 \leq t_i, d_i < 10^9$
- $0 \leq a_i \leq b_i < 10^9$
- $0 \leq u_j, y_j, x_j < 10^9$

Vsaka podnaloga je sestavljena iz dveh delov. V prvem delu so vse ovire prisotne vse od začetka (torej $t_i = 0$ za vse ovire). Pravilna rešitev enega izmed delov prinese polovico točk podnaloge.

Podnaloga 1 (20 točk)

- $1 \leq n, q \leq 1000$

Podnaloga 2 (30 točk)

- $x_j \leq x_{j+1}$

Podnaloga 3 (50 točk)

- Brez dodatnih omejitev.

Vhod. V prvi vrstici sta podani števili n in q, ločeni s presledkom. Naslednjih n vrstic opisuje ovire s števili t_i, d_i, a_i in b_i.

Nato sledi q vrstic, ki opisujejo poizvedbe. Da zagotovimo, da so izračunane v danem vrstnem redu, so podane v zakodirani obliki s števili u'_j, y'_j in x'_j ($0 \leq u'_j, y'_j, x'_j < 10^9$). Naj bo r_j rezultat j-te poizvedbe (za $j \leq 0$ privzamemo $r_j = 0$). Dejanske vrednosti poizvedb izračunamo po sledečih formulah: $u_j = (u'_j + r_{j-1}) \bmod 10^9$,$y_j = (y'_j + r_{j-2}) \bmod 10^9$ in $x_j = (x'_j + r_{j-3}) \bmod 10^9$.

Bodi pozoren, kako tvoj programski jezik obravnava negativna števila, ko računa ostanek pri deljenju. Dekodirane vrednosti u_j, y_j, x_j morajo biti nenegativne.

Izhod. Za vsako poizvedbo izpiši eno vrstico, razdaljo, ki jo prepotuje laserski žarek. Če se žarek ne zaleti v nobeno oviro, razen morebiti severnega zidu skladišča, izpiši `-1`.

Primer vhodne datoteke:

```
4 3
7 3 4 7
0 4 7 8
2 7 5 6
0 5 0 2
2 1 5
999999999 2 3
5 999999999 1
```

Pripadajoča izhodna datoteka:

```
6
-1
0
```

Komentar. Dekodiran seznam poizvedb je:

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

2. Podzaporedja

(*Omejitev časa: 4 s.* *Omejitev pomnilnika: 1024 MB.*)

Analiza nizov je eno od pomembnejših področij bioinformatike. Analizirani nizi so tipično zaporedja DNK, ki so sestavljena iz 4 oznak (`A`, `C`, `G` in `T`), ali zaporedja aminokislin, ki sestavljajo beljakovine in so označene z 20 simboli. Ne glede na vrsto zaporedja je zanimivo opazovati, kolikokrat se nek vzorec pojavi kot podzaporedje v določenih odsekih zaporedja oz. podnizih.

Pozor, podzaporedje dobimo z brisanjem nič ali več znakov iz niza. Podzaporedje torej ni nujno strnjen del izhodiščnega niza, katerim bomo rekli podnizi. Podniz dobimo z brisanjem predpone in pripone izhodiščnega niza, ki sta lahko tudi prazni.

Podana bosta niz s dolžine n in niz p dolžine m, ki bosta vsebovala samo male črke angleške abecede. Napišite program, ki bo učinkovito odgovoril na q poizvedb, definiranih z indeksoma i in j: kolikokrat se p pojavi kot podzaporedje v podnizu $s_i s_{i+1} \ldots s_j$?

Omejitve

Za vse podnaloge veljajo naslednje omejitve:

- $1 \leq n, q \leq 100\,000$
- $1 \leq m \leq 40$

- $1 \leq i \leq j \leq n$

Podnaloga 1 (20 točk)

- $n \leq 20$
- $q \leq 100$

Podnaloga 2 (30 točk)

- $m \leq 2$

Podnaloga 3 (30 točk)

- $m \leq 8$

Podnaloga 4 (20 točk)

- Brez dodatnih omejitev.

Vhod. Prva vrstica vsebuje niz s, druga pa vzorec p. Tretja vrstica vsebuje število poizvedb q, ki so podane v naslednjih vrsticah. Vsaka poizvedba je opisana z začetnim in končnim indeksom (i in j) podniza, ki nas zanima.

Izhod. Za vsako poizvedbo izpišite odgovor v svojih vrstici. Ker so odgovori lahko precej veliki, izpišite zgolj ostanke pri deljenju odgovora z 1 000 000 007.

Primer vhodne datoteke:

```
ababbacaba
abba
3
1 10
3 4
3 6
```

Pripadajoča izhodna datoteka:

```
17
0
1
```

Primer vhodne datoteke:

```
ababbacaba
ab
3
1 10
3 4
4 8
```

Pripadajoča izhodna datoteka:

```
9
1
0
```

3. Popravilo ceste

(*Omejitev časa: 1 s.* *Omejitev pomnilnika: 512 MB.*)

Programer Joe je med pripravami na svoje naslednje računalniško tekmovanje naletel na naslednji problem. To še ni problem, ki ga boš moral ti reševati pri tej nalogi, je pa vseeno povezan s tvojo nalogo, zato si preberimo opis problema, preden se posvetimo tvoji nalogi.

Joejev problem. Dan je cestni odsek, ki je po obliki daljica s celoštevilsko dolžino d. Tako lahko poljubno točko na cesti predstavimo z realnim številom x (z območja $0 \leq x \leq d$), ki pove razdaljo med to točko in levim krajiščem naše daljice. Vzdolž te ceste živi n prebivalcev; pri tem živi i-ti prebivalec (za $i = 1, \ldots, n$) na

točki x_i. Ti položaji so cela števila z območja $0 \le x_i \le d$ in vsa so različna (torej iz $i \ne j$ sledi $x_i \ne x_j$).

Ta cestni odsek bo treba v celoti popraviti. Za ta posel se potegujeta dve podjetji, ki ju bomo označili preprosto z 0 in 1. Vlada se je odločila, da ne bo najela le enega od teh podjetij za popravilo celotnega odseka, pač pa bo popravilo razdelila med obe podjetji v skladu z željami prebivalcev. Vsakega prebivalca so vprašali, katero od obeh podjetij podpira; naj bo $p_i \in \{0, 1\}$ številka podjetja, ki ga podpira i-ti prebivalec. Izbrali so tudi neko liho število k z območja $1 \le k \le n$. Za vsako točko x na obravnavanem cestnem odseku so definirali *soseščino* točke x kot množico tistih k prebivalcev, ki živijo najbližje točki x. (Če k-ti in $(k+1)$-vi najbližji sosed živita enako daleč od točke x, se za k-tega soseda in s tem za del soseščine šteje tistega od njiju, ki živi na nižji koordinati.) Točko x bo popravilo tisto podjetje, ki ga podpira večina prebivalcev v soseščini točke x. (Ker je k lih, ni nobenega tveganja, da bi bila podpora obeh podjetij izenačena — eno od njiju bo gotovo imelo podporo večine prebivalcev v soseščini.)

Če to definicijo uporabimo pri vsakem realnem številu x z območja $0 \le x \le d$, vidimo, da bo v splošnem nekatere dele ceste popravilo podjetje 0, nekatere pa podjetje 1. (Lahko pa se, če so take želje prebivalcev, zgodi celo to, da bo opazovani odsek ceste v celoti popravilo samo eno od obeh podjetij.) Izračunaj skupno dolžino tistih delov ceste, ki jih bo popravilo podjetje 1 (to dolžino bomo označili z d_1).

Joejeva rešitev. Sčasoma je Joe obupal nad tem, da bi problem rešil pravilno. Ker se ni domislil, kako izračunati pravo vrednost d_1, je poskusil najti vsaj približek te vrednosti, in sicer z vzorčenjem. Na cesti si je izbral m točk, $t_1, \ldots, t_m$; to so cela števila z območja $0 \le t_i \le d$ in vsa so različna (če torej velja $i \ne j$, potem velja tudi $t_i \ne t_j$). Za vsako od teh točk je pregledal njeno soseščino (kot je leta definirana v zgornjem opisu problema) in si zapomnil, ali večina prebivalcev te soseščine podpira podjetje 0 ali podjetje 1. Pri m_1 od teh m primerov se je izkazalo, da večina prebivalcev podpira podjetje 1 (pri ostalih $m - m_1$ primerih pa je večina sosedov podpirala podjetje 0). Joe je zdaj zaključil, da če je celoten odsek dolg d in če je pri m_1/m vzorcih imelo večino podjetje 1, potem mora biti skupna dolžina tistih delov ceste, ki jih bo popravilo podjetje 1, približno $(m_1/m) \cdot d$, tako da je to vrednost izpisal kot svojo približno rešitev (to rešitev označimo z d_J).

Tvoja naloga. Najel te je zlobni ocenjevalec Bob, ki se mu zdi Joejeva rešitev precej slaba in bi rad, da mu pomagaš sestaviti testni primer, na katerem bo Joejeva rešitev dajala čim slabše rezultate. Vrednosti n, d, k, $x_1, \ldots, x_n$, m, $t_1, \ldots, t_m$ so že znane in jih ne moremo spremeniti, lahko pa še izbiramo želje prebivalcev, torej vrednosti $p_1, \ldots, p_n \in \{0, 1\}$. Tvoja naloga je izbrati te vrednosti tako, da bo razlika med Joejevo rešitvijo in pravilno rešitvijo njegovega problema (torej $|d_J - d_1|$) čim večja. Točkovanje pri tej nalogi je relativno glede na neko zgornjo mejo vrednosti $|d_J - d_1|$, tako da ni nujno, da najdeš res največjo možno vrednost $|d_J - d_1|$, pač pa bo tvoja rešitev dobila tem več točk, čim bližje bo prišla omenjeni zgornji meji.

Omejitve

Za vse podnaloge veljajo naslednje omejitve:

- n, m, d so cela števila; $10 \le n \le 150$, $10 \le m \le 150$, $1 \le d \le 10^6$
- k je liho celo število; $1 \le k \le n$
- $x_1, \ldots, x_n$ so različna cela števila; $0 \le x_i \le d$ za vse $i = 1, \ldots, n$

- $t_1, \ldots, t_m$ so različna cela števila; $0 \leq t_i \leq d$ za vse $i = 1, \ldots, m$.

Pri pripravi testnih primerov so bili n, m in k izbrani naključno z enakomerno porazdelitvijo na območjih, ki jih navajajo omejitve za posamezno podnalogo.

Podnaloga 1 (25 točk)

- $n \geq 60$
- $m \geq 40$
- $k \leq 15$

Podnaloga 2 (25 točk)

- $n \geq 60$
- $m \geq 40$
- $k \geq n/2$

Podnaloga 3 (25 točk)

- $n \geq 100$
- $m \geq 40$
- $35 \leq k \leq 2n/5$

Podnaloga 4 (25 točk)

- Brez dodatnih omejitev.

Vhod. V prvi vrstici so d, n, m in k. V drugi vrstici so $x_1, \ldots, x_n$. V tretji vrstici so $t_1, \ldots, t_m$.

Izhod. Izpiši vrednosti $p_1, \ldots, p_n$, vse v eni vrstici, ločene s presledki.

Točkovanje. Za vsak testni primer bomo tvoj rezultat točkovali po naslednjem postopku. Naj bo x vrednost $|d_J - d_1|$, ki jo je dosegla tvoja rešitev, in naj bo y neka dobro definirana zgornja meja za največjo možno vrednost $|d_J - d_1|$, ki jo je mogoče doseči pri tem testnem primeru. Potem dobi tvoja rešitev pri tem testnem primeru $100 \cdot (x/y)^2$ odstotkov skupnega števila točk, ki so na razpolago za ta testni primer. Znotraj posamezne podnaloge so vsi testni primeri vredni enako število točk.

Podroben opis zgornje meje, ki jo uporabljamo pri točkovanju, bi ti razkril preveč o tem, kako rešiti nalogo; za občutek o tem, da je uporabljena meja precej tesna, pa povejmo, da doseže najboljša organizatorjem znana rešitev pri tej nalogi skupaj 93 % točk.

Primer vhodne datoteke:

```
100 6 2 3
50 20 80 10 70 90
5 60
```

Ena od možnih pripadajočih izhodnih datotek:

```
1 0 1 1 0 0
```

Komentar. Pri tem konkretnem vhodu doseže zgoraj prikazani izhod vrednosti $d_1 = 60$ in $d_J = 100$. Dobljena razlika $|d_1 - d_J| = 40$ je tudi največja možna za te vhodne podatke; izkaže se, da je pri tem vhodnem primeru tudi enaka zgornji meji, ki uporabljamo pri točkovanju.

NEUPORABLJENE NALOGE IZ LETA 2015

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 10. tekmovanjem ACM v znanju računalništva (leta 2015), pa jih potem na tistem tekmovanju nismo uporabili (ker se nam je nabralo več predlogov nalog, kot smo jih potrebovali za tekmovanje). Ker tudi te neuporabljene naloge niso nujno slabe, jih zdaj objavljamo v letošnjem biltenu, če bodo komu mogoče prišle prav za vajo. Dodali smo tudi težje različice nekaj nalog, ki smo jih v lažji obliki uporabili na tekmovanju 2015. Poudariti pa velja, da niti besedilo teh nalog niti njihove rešitve (ki so na str. 129–169) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Pobeg

Nek usodni dan je bilo v zaporu veliko stražmojstrov na bolniški, zato so stražili izhodna vrata samo občasno. Natančneje, stražili so samo v časih od a_1 do b_1, od a_2 do b_2, ... in od a_m do b_m; časi so podani kot minute od polnoči. Tisti isti dan se je odločilo pobegniti n razbojnikov; i-ti od njih se je poskusil izmuzniti skozi izhodna vrata ob času c_i. **Opiši postopek**, ki iz teh podatkov izračuna, koliko razbojnikov je uspešno pobegnilo.

2. Igra 2048

Dana je karirasta mreža s h vrsticami in w stolpci. Nekatere celice mreže so prazne, nekatere pa vsebujejo ploščico, na kateri je napisano naravno število. Izberimo si smer premikanja, na primer navzdol (v originalni različici igre 2048 so možni tudi premiki gor, levo in desno). Ena „poteza“ v igri se sestoji iz tega, da se ploščice premikajo navzdol po mreži. Ploščica se neha premikati, če pride na dno mreže ali pa če se na spodnji strani dotakne neke druge ploščice. Če se dotakneta dve ploščici z enako številko, se združita v eno samo: zgornja ploščica izgine, na spodnji pa se številka podvoji. Če je v istem trenutku možnih več združitev, se izvrši najnižja med njimi. Pri tem pa velja omejitev, da ploščica, ki je nastala s takšno združitvijo, kasneje znotraj iste poteze ne more sodelovati v nobeni nadaljnji združitvi več.

Napiši podprogram, ki izvaja poteze v skladu z opisanimi pravili, dokler se stanje mreže še kaj spreminja. Predpostavi, da je stanje mreže shranjeno v neki tabeli celoštevilskih vrednosti (globalna spremenljivka), pri čemer vrednost 0 pomeni prazno celico, pozitivna vrednost pa pomeni, da celica vsebuje ploščico s tisto številsko vrednostjo.

3. znajdi.se

(To je težja različica naloge, ki smo jo na tekmovanju 2015 uporabili kot 3. nalogo v prvi skupini.) Razvili smo nov iskalnik *znajdi.se*, ki nam izpiše, kako pridemo iz točke A v točko B. Problem je, da nam iskalnik korake izpiše v pomešanem vrstnem redu. Tako se opis poti iz kraja G v kraj S glasi:

```
- mimo O nadaljujemo skozi tri križišča, dokler ne pridemo do A
- pri B zavijemo desno, dokler ne dospemo do S
- iz A zavijemo v krožno križišče, od tam nadaljujemo naravnost do D
```

```
- na H zavijemo desno proti O
- od D sledimo modrim oznakam, dokler ne zagledamo B
- nato pri J peljemo naprej 1,2 km proti H
- iz G zavijemo levo proti J
```

Pri tem veljajo naslednje omejitve:

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

Napiši program, ki prebere opis poti in izpiše kraje na poti v pravilnem vrstnem redu. Če torej tvoj program prebere zgornji opis poti, mora izpisati „`GJHOADBS`“.

4. Delni izid

(To je težja različica naloge, ki smo jo na tekmovanju 2015 uporabili kot 1. nalogo v prvi skupini.) Na košarkaški tekmi je *delni izid* razlika med točkami, ki sta jih v nekem časovnem intervalu dosegli ekipi.

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

Primer: če dobimo vhodno zaporedje

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

je največji delni izid v prid prve ekipe 7 točk, ki ga ta ekipa doseže pri podzaporedju $\langle +3, +2, -2, +2, +2\rangle$; največji delni izid v prid druge ekipe pa je 11 točk, dosežen pri podzaporedju $\langle -3, +2, -2, -3, +3, -2, -3, -3\rangle$.

5. Razporejanje študentov

Imamo k računalniških učilnic, vsaka ima znano kapaciteto: v učilnici i je prostora za c_i študentov. Imamo tudi n študentov, ki jih je načeloma treba razporediti vsakega v eno od učilnic, pri čemer je vsak oznanil svoje preference: v nekatere učilnice mogoče sploh noče, v druge pa bi šel, vendar je navedel nek vrstni red, katere so mu bolj in katere manj všeč. Študente razporejamo tako, da gremo po vrsti po seznamu študentov, za vsakega študenta pa pogledamo njegov seznam zaželenih učilnic in ga damo v prvo tako, ki še ni polno zasedena; če pa take ni, bo ta študent pač ostal nerazporejen. **Napiši program**, ki razporedi študente po opisanem postopku in izpiše za vsako učilnico seznam študentov, na koncu pa še število nerazporejenih študentov.

Vhodni podatki: v prvi vrstici sta n in k; v drugi vrstici so kapacitete učilnic $c_1, c_2, \ldots, c_k$, ločene s presledki; sledi še n vrstic, ki opisujejo preference študentov. V vsaki od teh vrstic je seznam učilnic, v katere je pripravljen iti tisti študent (od najbolj do najmanj zaželene); seznam se začne najprej s številom elementov, nato pa so po vrsti navedene številke učilnic. Učilnice so oštevilčene s celimi števili od 1 do k, študentje pa od 1 do n.

Primer vhodnih podatkov:

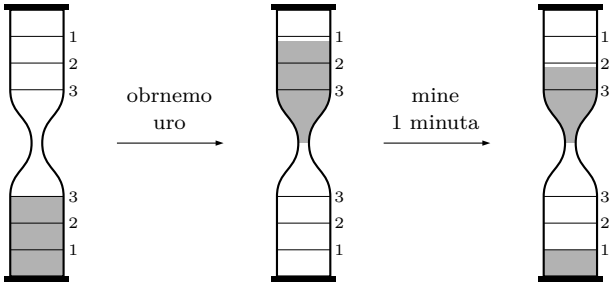
```
5 2
2 1
1 2
2 2 1
1 2
2 1 2
2 2 1
```

Pripadajoči izpis:

```
Učilnica 1: 2 4
Učilnica 2: 1
Nerazporejeni: 3 5
```

6. Peščena ura

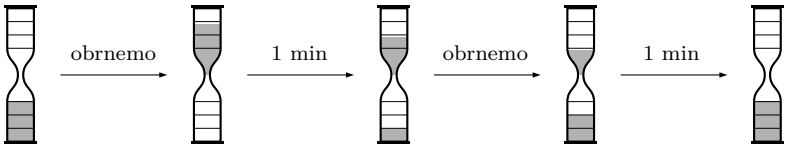
Imamo peščeno uro, v kateri je pesek, ki potrebuje 3 minute, da se pretoči iz zgornje v spodnjo polovico ure. Na obeh polovicah ima ura oznake za 1, 2 in 3 minute. Uro lahko obrnemo samo takrat, ko je nivo peska v spodnji polovici na oznaki 1, 2 ali 3 minute.



Z uro želimo izmeriti čas T minut za neko celo število T od 1 do 12 tako, da bo na koncu vsakega merjenja ves pesek v spodnji polovici ure. (Tudi na začetku je ves pesek v spodnji polovici ure.)

(*a*) **Opiši postopek**, ki bo glede na želeni čas merjenja s čim manj obrati obračal uro tako, da bo ob izteku želenega časa ves pesek v spodnji polovici ure.

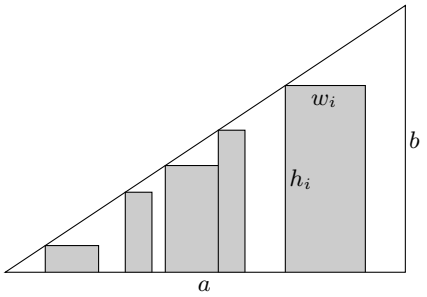
Primer. Recimo, da je želeni čas $T = 2$ minuti. Odmerimo ga lahko tako, da uro obrnemo dvakrat: najprej sprožimo (obrnemo) uro, počakamo, da izteče 1 minuta, nato uro spet obrnemo in počakamo še 1 minuto, da ves pesek izteče v spodnji del.



(*b*) Reši splošnejšo različico te naloge, pri kateri je v uri dovolj peska za c minut, pa tudi oznake na vsaki polovici ure gredo od 1 do c. Uro lahko obrnemo le, ko je nivo peska na eni od oznak. Odmerili bi radi čas T minut, pri čemer je T lahko poljubno naravno število. Kakšno je najmanjše potrebno število obračanj ure, s katerim lahko odmerimo T minut? Kdaj in kako moramo obračati uro, da to storimo?

7. Polaganje plošč

(To je težja različica naloge, ki smo jo na tekmovanju 2015 uporabili kot 3. nalogo v drugi skupini.) Imamo pravokotni trikotnik, širok a enot in visok b enot. Dobili smo tudi več pravokotnih plošč in poznamo njihove velikosti: i-ta plošča je široka w_i enot in visoka h_i enot. Plošče bi radi zložili v trikotnik tako, da bo spodnja stranica plošče ležala na spodnjem robu trikotnika (ki ima dolžino a; glej sliko spodaj), zgornje levo oglišče plošče pa bo ležalo na hipotenuzi trikotnika. Plošč ne smemo vrteti tako, da bi stranica w_i prišla po višini, h_i pa po širini. Poleg tega se plošče ne smejo prekrivati ali štrleti ven iz trikotnika, lahko pa se dotikajo. Naslednja slika kaže primer takega trikotnika, v katerega smo položili pet plošč:



(*a*) **Opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki v okviru teh omejitev poišče tak nabor plošč, ki jih je mogoče vse hkrati položiti v trikotnik in ki imajo največjo skupno ploščino. Kot vhodne podatke tvoj postopek dobi velikost trikotnika (a in b), število plošč n in njihove dimenzije $w_1, h_1, w_2, h_2, \ldots, w_n, h_n$.

(*b*) Kaj pa, če vendarle dovolimo tudi obračati plošče za 90 stopinj? Reši nalogo še za ta primer, vendar z dodatno omejtivijo, da velja $a \leq b$.

8. Kodiranje

Oglejmo si še dve različici naloge s kodiranjem, ki smo jo na tekmovanju 2015 uporabili kot 4. nalogo v drugi skupini.

(*a*) Števke od 0 do 9 bi radi predstavili s 5-bitnimi kodami, torej z zaporedji petih ničel in enic. Takšnih peteric je kar $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 32$, mi pa moramo med temi 32 petericami izbrati deset peteric, ki jih bomo uporabili kot kode.

Recimo, da bomo naše peterice prenašali po nekem komunikacijskem kanalu, na katerem lahko prihaja do napak: peterica, ki jo prebere prejemnik sporočila, ni nujno enaka tisti, ki jo je pošiljatelj odposlal. Možne napake, do katerih lahko na ta način pride, so znane: če pošljemo peterico x, bo do pošiljatelja prišla ena od peteric iz množice $N(x)$, ne vemo pa točno, katera. Množice $N(x)$ za vse možne peterice x so podane vnaprej. **Opiši postopek**, ki poišče tak nabor desetih peteric (če sploh obstaja), pri katerem bo prejemnik zagotovo vedno lahko na podlagi prejete peterice ugotovil, ali je pri prenosu prišlo do napake ali ne.

(*b*) Reši prejšnjo podnalogo, vendar s strožjo zahtevo: iščemo tak nabor desetih peteric, da bo lahko prejemnik na podlagi prejete peterice vedno pravilno ugotovil, katero od izbranih desetih peteric je pošiljatelj poslal. Temu lahko rečemo, da znamo napake pri pošiljanju ne le zaznavati, ampak tudi popravljati.

(*c*) Definirajmo naslednje vrste napak pri prenosu peterice od prejemnika do pošiljatelja: T_k = v peterici se spremeni[3] natanko k bitov; T_{Ak} = spremeni se k zaporednih bitov; C_1 = en prižgan bit se ugasne; S_1 = en ugasnjen bit se prižge; $C_\star$ = ugasne se eden ali več prižganih bitov; $S_\star$ = prižge se eden ali več ugasnjenih bitov; Z = zamenjata se dva sosednja bita (iz 01 v 10 ali obratno).

S pomočjo postopkov iz podnalog (*a*) in (*b*) ugotovi, za katere kombinacije vrst napak (izmed $T_1, \ldots, T_5$, T_{A2}, T_{A3}, T_{A4}, C_1, S_1, $C_\star$, $S_\star$, Z) je mogoče najti tak nabor desetih peteric, pri katerem se bo dalo zaznati oz. odpraviti vse napake tistih vrst.

(*d*) Nalogo lahko malo posplošimo: recimo, da imamo n različnih števil $b_1, \ldots, b_n$ (to so cela števila, večja ali enaka 0) in jih kodiramo z nizi k bitov, torej b_i predstavimo z nizom $a_{i1}a_{i2}\ldots a_{ik}$, pri čemer so $a_{ij} \in \{0, 1\}$. Pri tej podnalogi predpostavimo, da do napak pri prenašanju sporočil ne prihaja.

Prejemniku bi radi olajšali delo z dekodiranjem prejetih nizov: lepo bi bilo, če bi se dalo število b_i kar izračunati iz niza $a_{i1}a_{i2}\ldots a_{ik}$, in sicer po formuli

$$b_i = (a_{i1}w_1 + a_{i2}w_2 + \ldots + a_{ik}w_k) \bmod m$$

za neka cela števila $w_1, \ldots, w_k$ in neko naravno število m. **Opiši postopek**, ki poišče takšne $w_1, \ldots, w_k, m$, pri katerih omenjena formula daje pravilne rezultate za vse i od 1 do n, ali pa ugotovi, da ne obstajajo.

9. Drugi tir A

V neki deželi imajo železnico, ki povezuje glavno mesto z največjim pristaniščem. Med obema je še $n - 2$ vmesnih postaj, pri čemer je med i-to in $(i + 1)$-vo postajo zgrajen tir, po katerem lahko peljejo vlaki v obe smeri. Na vsaki postaji lahko stoji poljubno število vlakov, ne da bi motili promet, le po tiru med sosednjima postajama sme peljati naenkrat le en vlak. Vsaka postaja ima na levi in desni strani tudi po en semafor, prek katerega dovolimo vlaku iz postaje, ki že najdlje čaka, da

[3] S tem, da se bit spremeni, imamo v mislih to, da se prižge, če je bil prej ugasnjen, oz. ugasne, če je bil prej prižgan.

postajo zapusti v smeri naprej ali nazaj. Ko zapelje vlak mimo, se na semaforju avtomatsko nazaj prižge rdeča luč. Seveda vlaki ne želijo spreminjati svoje smeri in če so namenjeni proti pristanišču, ne bodo peljali nazaj proti glavnemu mestu.

Glede na to, da v deželni malhi primanjkuje denarja, so se v deželi odločili, da, namesto da bi postaje povezali z dodatnim tirom, najamejo programerja, ki bo usmerjal semaforje. Pomagaj jim in **napiši podprogram** ObPrihodu, ki ga bo sistem klical vsakokrat, ko bo vlak prispel na kakšno postajo, da bo prižigal in ugašal semaforje. Tvoj podprogram naj bo takšne oblike.

```
procedure ObPrihodu(p: integer; Levo: boolean);      { v pascalu }
void ObPrihodu(int p, bool levo);                    /* v C/C++ in podobnih jezikih */
def ObPrihodu(p, levo): ...                          # v pythonu
```

Pri tem parameter p pove številko postaje, na katero je vlak prišel, parameter levo pa pove smer vlaka (če je levo == **true**, se vlak premika levo, sicer pa desno). Postaje so oštevilčene od 1 (na levem koncu proge) do n (na desnem koncu proge).

Predpostavi, da je na voljo naslednji podprogram, s katerim lahko prižigaš in ugašaš semaforje:

```
procedure PrizgiSemafor(p: integer; Levo, Stanje: boolean);      { v pascalu }
void PrizgiSemafor(int p, bool levo, bool zelena);               /* v C/C++ in podobnih jezikih */
def PrizgiSemafor(p, levo, smer): ...                            # v pythonu
```

Podprogram postavi na postaji p semafor za vožnjo v smer, ki jo določa parameter levo (torej levo, če je levo == **true**, sicer pa desno) v stanje, ki ga določa parameter zelena (torej prižge zeleno luč, če je zelena == **true**, sicer pa rdečo).

V svoji rešitvi lahko uporabiš tudi poljubne globalne spremenljivke, za katere lahko tudi določiš, kako jih je treba na začetku delovanja sistema inicializirati.

Železnica je na začetku dogajanja prazna in na njej ni nobenih križišč, kjer bi lahko vlaki prihajali nanjo ali jo zapuščali. Vlaki se vedno peljejo od postaje 1 do n ali obratno, torej nikoli le po delu železniške proge. Nov vlak se vedno pojavi na postaji 1 ali n (takrat se zanj tudi prvič pokliče podprogram ObPrihodu), nato pa od tam pelje do nasprotnega konca proge.

10. Drugi tir B

Dana je enaka železniška proga kot pri prejšnji nalogi: imamo n postaj, ki so po vrsti oštevilčene od 1 (na levem koncu proge) do n (na desnem koncu proge). Po dve zaporedni postaji sta povezani z enotirno progo. Zaradi varnostnih ukrepov se lahko po taki povezavi med dvema zaporednima postajama pelje le en vlak naenkrat (torej nanjo ne smemo spustiti dveh ali več vlakov niti v primeru, če se vsi peljejo v isto smer). Na vsaki postaji je dovolj prostora, da lahko na njej poljubno dolgo časa čaka poljubno število vlakov, ne da bi kaj ovirali pot morebitnih drugih vlakov, ki bi želeli peljati mimo postaje; velja pa omejitev, da morajo vlaki, ki vozijo v isto smer, zapustiti posamezno postajo v enakem vrstnem redu, v kakršnem so prišli nanjo. Vsi vlaki porabijo za vožnjo od ene postaje do naslednje enako mnogo časa, zato si lahko predstavljamo, da se promet odvija v diskretnih časovnih korakih. Vlak torej potrebuje $n-1$ časovnih korakov, da pripelje od enega konca proge do nasprotnega. Dan je dovolj dolg za v takih časovnih korakov.

Na začetku dneva čaka na levem koncu proge (postaja številka 1) ℓ vlakov, ki bi se radi peljali do desnega konca proge (postaja številka n). Ti vlaki prevažajo tovor in sicer i-ti od teh vlakov prevaža a_i enot tovora. Podobno tudi na desnem koncu proge (postaja številka n) stoji d vlakov, ki bi se radi peljali do levega konca proge (postaja številka 1), in sicer i-ti od teh vlakov prevaža b_i enot tovora.

Opiši, kako je treba v okviru danih omejitev premikati vlake, tako da bo do konca dneva (torej po v časovnih korakih) skupna količina tovora na vseh tistih vlakih, ki pridejo na cilj, največja možna.

11. Knjižnica

V neko knjižnico je vpisanih m uporabnikov, ki so predstavljeni z zaporednimi številkami od 0 do $m-1$. V tej knjižnici je na voljo za izposojo n plošč, ki so predstavljene z zaporednimi številkami od 0 do $n-1$. **Opiši** podatkovno strukturo, s katero bo knjižnica lahko čim bolj učinkovito vzdrževala podatke o izposoji plošč.

(a) Tvoja podatkovna struktura naj podpira naslednje operacije (za vsako od njih tudi opiši postopek, s katerim bi se jo izvedlo):

- Izposodi(stPlosce, stOsebe) — kliče se jo, ko si poskuša uporabnik izposoditi ploščo; če je ta plošča že izposojena, naj javi napako;
- Vrni(stPlosce) — kliče se jo, ko uporabnik vrne ploščo, ki jo je imel izposojeno;
- PriKomJe(stPlosce) — naj vrne številko uporabnika, ki ima izposojeno to ploščo, ali -1, če trenutno ni izposojena;
- KajIma(stOsebe) — naj izpiše seznam plošč, ki jih ima ta uporabnik izposojene;
- KolikoIma(stOsebe) — naj vrne število plošč, ki jih ima ta uporabnik izposojene.

(b) Kaj pa, če hočemo poleg zgoraj naštetih operacij podpreti še naslednji dve?

- KajVseJeZeImel(stOsebe) — naj izpiše seznam vseh plošč, ki jih je imel ta uporabnik vsaj enkrat že izposojene;
- KdoVseJoJeZeImel(stPlosce) – naj izpiše seznam vseh uporabnikov, ki so si že kdaj izposodili to ploščo.

12. Cenik

V trgovini se cena izdelka skozi čas pogosto spreminja, kar nam prikazuje spodnji primer cenika.

Od	Do	Cena
1	5	12,6
6	9	13,5
10	10	13,1
11	19	13,8
19	31	12,9

Cenik določa ceno za vsak dan v obdobju, ki nas zanima (recimo, da je dolgo n dni; v zgornjem primeru je $n = 31$). Cene se ne prekrivajo.

Cenik se lahko spreminja tako, da cene dodajamo ali brišemo, pri čemer ustrezno popravimo končni/začetni datum prejšnje/naslednje cene, da ne pride do prekrivanj ali lukenj.

Če pri dodajanju nova cena v celoti prekrije staro, staro odstranimo.

Pri brisanju popravimo le veljavnost prejšnje cene.

Opiši podatkovno strukturo, s katero bi predstavil cenik (npr. kot globalno spremenljivko), in opiši postopka ali napiši podprograma Vstavi in Briši, ki ta cenik ustrezno popravita ob vstavljanju oz. brisanju cene. Podprograma naj bosta takšne oblike:

```
void Vstavi(int Od, int Do, double Cena);
void Brisi(int Od, int Do);
```

13. Komplet

V trgovini poleg posameznih izdelkov prodajajo tudi komplete, pri čemer je cena kompleta vsota cen izdelkov (elementov), ki ga sestavljajo. Vsak izdelek v kompletu nastopa največ enkrat.

Podan imaš cenik cen izdelkov za tekoči mesec. Cena velja od navedenega dne do dne naslednje kasnejše cene istega izdelka. Cenik določa cene vseh izdelkov za vse dni v mesecu (31 dni).

Opiši postopek ali napiši podprogram CenikKompleta, ki kot vhod dobi cenik izdelkov in seznam elementov, ki tvorijo komplet, on pa na podlagi tega izračuna cenik kompleta.

```
struct CenaIzdelka { string izdelek; int od; double znesek; };
struct CenaKompleta { int od; double znesek; };

void CenikKompleta(vector<CenaIzdelka> &cenikIzdelkov,
                   vector<string> &elementi,
                   vector<CenaKompleta> &cenikKompleta);
```

Primer: naj bo komplet sestavljen iz izdelkov A + B + D. Recimo, da je dan naslednji cenik izdelkov:

Izdelek	Od	Znesek
A	1	12
A	18	13
B	1	2
B	5	3
C	1	7
D	1	4

Cenik kompleta (torej rezultat, ki ga mora izračunati tvoj podprogram) je potem takšen:

Od	Znesek
1	18
5	19
18	20

14. Dvigalo

Imamo stolpnico z n nadstropji, ki so vsa enako visoka in oštevilčena od 0 (pritličje) do $n-1$ (najvišje nadstropje). V 0. nadstropju je dvigalo, ki lahko pelje največ eno osebo naenkrat. Podan je še spisek oseb: i-ta bi se rada peljala iz nadstropja a_i v b_i, pri čemer vedno velja $b_i > a_i$ (vsi se peljejo navzgor). Potnike bi radi prepeljali tako, da bo na koncu vsak v nadstropju, kamor želi priti, in da bo skupna prevožena pot dvigala najmanjša možna. **Opiši postopek**, kako naj prevažamo potnike, in izračunaj prevoženo pot dvigala pri tem (merjeno v številu nadstropij). Pri tem ni nujno, da posameznega človeka prepeljemo od a_i do b_i v enem kosu; lahko ga vmes še malo odložimo v kakšnem vmesnem nadstropju in se nekaj časa ukvarjamo z drugimi potniki.

15. Globalno segrevanje

Dana je neka dežela, ki ima obliko pravokotne kariraste mreže, sestavljene iz $w \times h$ celic (enotskih kvadratov). Zaradi globalnega segrevanja se bo gladina morij in oceanov dvignila in nekatere celice mreže se bodo znašle pod vodo. Zanima nas, koliko celic bo ostalo nad vodo ob različnih scenarijih globalnega segrevanja.[4]

Tako imamo za vsako celico mreže (x, y) podano višino v metrih $h(x, y)$ kot celo število med -100 in 3000. Če se zaradi globalnega segrevanja dvignejo morja za q metrov, se potopijo vse tiste celice, ki imajo $h(x, y) \leq q$. (Torej se lahko celica potopi tudi, če vse njene sosede ostanejo nad vodo — predstavljajmo si, da so tla porozna in jih lahko voda zalije od spodaj.)

(a) Recimo, da kot poizvedbe dobimo veliko število celoštevilskih q-jev in bi radi za vsakega od njih izračunali, koliko celic ostane nad vodo, če se morja dvignejo za q metrov. **Opiši postopek**, ki čim hitreje odgovori na vse poizvedbe.

(b) Kaj pa, če so višine $h(x, y)$ in poizvedbe q lahko poljubna realna števila?

(c) Kaj pa, če tla niso porozna in voda lahko vdre v mrežo le z zunanjih robov? Pri tej različici naloge torej definiramo potopljene celice takole: če se morja dvignejo za q metrov, bo celica (x, y) potopljena natanko tedaj, če (1) leži na robu mreže in je $h(x, y) \leq q$ ali pa (2) ima skupno stranico z neko potopljeno celico (x', y'), za katero velja $h(x, y) \leq h(x', y')$.

16. Drevo

Dano je drevo (neusmerjen povezan acikličen graf), v katerem ima vsako vozlišče neko vrednost (vrednost vozlišča u je realno število c_u). **Opiši postopek**, ki izbere nekaj (poljubno mnogo) vozlišč tako, da nobeni dve nista neposredno povezani, skupna vsota njihovih vrednosti pa naj bo največja možna.

Lažja različica: reši enak problem, le da imajo vsa vozlišča enako vrednost ($c_u = 1$ za vse u).

17. Skupni geni

Podana imamo genska zapisa dveh organizmov; vsak od njiju je niz, sestavljen iz črk `A`, `C`, `T` in `G`. Kot mero sorodnosti med organizmoma opazujemo njuno najdaljše

[4] Malo preprostejšo nalogo na temo globalnega segrevanja in dvigovanja morij smo imeli že na šolskem tekmovanju leta 2011 (2. naloga; glej bilten 2011, str. 33 in rešitev na str. 69).

skupno podzaporedje, ki ni nujno strnjeno. **Opiši postopek**, ki izračuna njegovo dolžino in izpiše eno izmed takih najdaljših skupnih podzaporedij. Vhodna niza sta lahko precej dolga, recimo po 10^6 znakov.

18. Dvolični stavki

Podan imamo slovar besed v jeziku, kjer sestavljamo stavke z enostavnim zlaganjem besed eno za drugo brez kakršnihkoli presledkov ali ločil. Poišči dolžino najkrajšega stavka, ki ga je mogoče iz besed sestaviti na vsaj dva različna načina. Pri tem smemo posamezno besedo uporabiti tudi po večkrat.

Primer: če imamo slovar slovenskih besed (v različnih oblikah), lahko niz *domačetudi* sestavimo na dva načina, kot *domače* + *tudi* ali kot *doma* + *četudi*.

19. Pristajalne ploščadi

Piše se leto 2542. Že zadnjih 30 let ima vsako gospodinjstvo namesto avtomobila vsaj en leteči krožnik. Da bi oblasti zajezile kaos v zraku, ki ga je povzročala jata sem in tja begajočih plovil, so se lotile projekta OPVP (omrežje pristajalno-vzletnih ploščadi). Po celotni državi nameravajo zgraditi n ploščadi, med katerimi bo mogoče potovati po točno določenih koridorjih. Poleg tega bo odslej mogoče pristajanje in vzletanje le s teh ploščadi in ne več kar z lastnega dvorišča, kot je bila to navada do sedaj. Ploščadi so sicer že postavljene, vendar lahko občasno pride do raznih okvar zapletene elektronike, zato takrat vzletanje in pristajanje na ploščadi ni mogoče — pravimo, da ploščad ne obratuje. Takoj, ko je popravilo na ploščadi končano, je ponovno pripravljena in v obratovanju. Ploščadi so oštevilčene z zaporednimi številkami od 1 do n.

Omrežje pristajalno-vzletnih ploščadi je predstavljeno s seznamom direktnih povezav. Direktna povezava je predstavljena s parom zaporednih števil dveh ploščadi (a, b), ki označuje, da obstaja neposredna pot med ploščadjo a in ploščadjo b, po kateri lahko z letečim krožnikom potujemo v obeh smereh. Pravimo, da je med ploščadjo a ter ploščadjo b *direkten prehod*, če med njima obstaja direktna povezava in sta trenutno obe ploščadi v obratovanju.

Ministrstvo za leteče objekte bo danes pričelo z objavo načrta OPVP, ki bo objavljen postopoma, saj bodo sproti opravljali razne zapletene analize trenutnega prometa v omrežju. Ker pa bi radi prebivalcem olajšali prehod na nov način potovanja, te prosijo za pomoč. **Opiši postopek**, ki bo postopoma prejemal informacije iz ministrstva o trenutnem stanju, prebivalcem pa odgovarjal na poizvedbe.

Spremembe stanja:

- Poveži(a, b): med ploščadjo a in b se pojavi direktna povezava (predpostaviš lahko, da sta a in b različni in da direktne povezave med njima doslej še ni bilo);
- Vklop(a): ploščad številka a postane aktivna in pripravljena za pristanke in vzlete letečih krožnikov;
- Izklop(a): na ploščadi s številko a je prišlo do tehničnih težav in do nadaljnjega ni v obratovanju.

Hkrati pa mora program tudi znati odgovoriti na poizvedbo prebivalcev:

- PreštejSosede(a): vrne naj število ploščadi, ki tvorijo s ploščadjo a direkten prehod.

Omejitve: število ploščadi je $n \leq 100\,000$, število sprememb in poizvedb skupaj pa je $q \leq 250\,000$.

REŠITVE NALOG ZA PRVO SKUPINO

1. Zaokrožanje temperature

Temperature bomo brali v zanki in jih sproti zaokrožali ter izpisovali. Trenutni temperaturi t najprej s funkcijo Odrezi odrežimo decimalke; dobljeno celo število imenujmo u.

Če je $t > 0$, se pri rezanju decimalk zmanjša ali pa ostane nespremenjen (če je bil že t sam po sebi celo število); v tem primeru je torej u vrednost, ki jo dobimo, če t zaokrožimo navzdol. Razlika $t - u$ je tedaj torej ≥ 0. Če je ta razlika $\geq 1/2$, bi morali po pravilih naloge zaokrožiti t navzgor, zato moramo v tem primeru u povečati za 1.

Če pa je bil $t < 0$, se pri rezanju decimalk poveča ali ostane nespremenjen (na primer: iz $-2{,}7$ nastane -2, to pa je večje od $-2{,}7$), torej je u takrat vrednost, ki jo dobimo, če t zaokrožimo navzgor. Razlika $t - u$ je tedaj torej ≤ 0. Če je ta razlika $< -1/2$, bi morali po pravilih naloge zaokrožiti t navzdol, zato v tem primeru u zmanjšamo za 1.

Zdaj lahko izpišemo u, paziti moramo le še na primere, ko je t negativen in se je zaokrožil na $u = 0$; takrat moramo najprej izpisati še minus, da bo nastal izpis -0 namesto le 0.

```
#include <stdio.h>

int main()
{
  double t;
  while (1 == scanf("%lf", &t)) /* Preberimo temperaturo. */
  {
    int u = Odrezi(t); /* Odrežimo ji decimalke. */

    /* Po potrebi popravimo zaokrožanje v pravo smer. */
    if (t - u >= 0.5) u++;
    else if (t - u < -0.5) u--;

    /* Izpišimo zaokroženo število, pazimo na -0. */
    if (t < 0 && u == 0) printf("-");
    printf("%d\n", u);
  }
}
```

Zapišimo to rešitev še v pythonu:

```
import sys
for vrstica in sys.stdin:
  t = float(vrstica)    # Preberimo temperaturo.
  u = Odrezi(t)         # Odrežimo ji decimalke.

  # Po potrebi popravimo zaokrožanje v pravo smer.
  if t - u >= 0.5: u += 1
  elif t - u < -0.5: u -= 1

  # Izpišimo zaokroženo število, pazimo na -0.
  print("%s%d" % ("-" if t < 0 and u == 0 else "", u))
```

2. Najlepši esej

Vhodno besedilo lahko beremo po znakih; dokler beremo črke, v spremenljivki dolzina štejmo dolžino (doslej prebranega dela) trenutne besede. Ko pridemo do znaka, ki ni črka, vemo, da je trenutne besede konec. (Paziti moramo na to, da pravilno zaznamo tudi konec zadnje besede; spodnji program se pri tem opira na dejstvo, da funkcija fgetc iz standardne knjižnice takrat vrne EOF, kar tudi ni črka.) Zdaj torej poznamo njeno pravo dolžino in lahko preverimo, če je prekratka (krajša od 3 znake) ali predolga (daljša od 8 znakov); v tem primeru tudi povečamo spremenljivki prekratke in predolge, ki štejeta prekratke in predolge besede. Nato pa postavimo spremenljivko dolzina nazaj na 0, da bomo pripravljeni na branje naslednje besede. Na koncu vemo, da je esej lep le, če sta spremenljivki prekratke in predolge obe enaki 0.

```
#include <stdio.h>

int main()
{
  int prekratke = 0, predolge = 0, dolzina = 0, c;
  do
  {
    c = fgetc(stdin); /* Preberimo naslednji znak. */
    /* Če smo prebrali črko, povečajmo števec, ki meri dolžino trenutne besede. */
    if ('A' <= c && c <= 'Z' || 'a' <= c && c <= 'z')
      dolzina++;
    /* Sicer smo na koncu besede. */
    else if (dolzina > 0)
    {
      /* Po potrebi povečajmo števca prekratkih in predolgih besed. */
      if (dolzina > 8) predolge++;
      else if (dolzina < 3) prekratke++;
      dolzina = 0; /* Pripravimo se na naslednjo besedo. */
    }
  }
  while (c != EOF);
  /* Izpišimo rezultate. */
  if (prekratke == 0 && predolge == 0) printf("Esej je lep.\n");
  else printf("Esej ni lep: %d prekratkih, %d predolgih besed.\n",
              prekratke, predolge);
  return 0;
}
```

Zapišimo to rešitev še v pythonu.

```
import sys
prekratke = 0; predolge = 0; dolzina = 0
while True:
  c = sys.stdin.read(1)
  if c.isalpha():                                                    (⋆)
    dolzina += 1
  elif dolzina > 0:
    if dolzina < 3: prekratke += 1
    elif dolzina > 8: predolge += 1
    dolzina = 0
  if not c: break
if prekratke == 0 and predolge == 0: print("Esej je lep.")
else: print("Esej ni lep: %d prekratkih, %d predolgih besed." % (prekratke, predolge))
```

V gornji rešitvi smo za preverjanje, ali je nek znak črka abecede, uporabili kar metodo isalpha; vendar pa ta vrne True ne le pri črkah angleške abecede (kot zahteva naloga), ampak tudi pri črkah drugih pisav. Če bi se hoteli res natančno držati navodila naloge (torej da lahko kot del besede štejemo le črke angleške abecede), bi lahko v vrstici (⋆) naredili nekaj takega:

```
  if 'A' <= c <= 'Z' or 'a' <= c <= 'z':
```

Elegantna možnost je tudi ta, da beremo besedilo po vrsticah in besede iščemo z regularnim izrazom:

```
import sys, re
prekratke = 0; predolge = 0; dolzina = 0
for vrstica in sys.stdin:
  for beseda in re.findall("[A-Za-z]+", vrstica):
    if len(beseda) < 3: prekratke += 1
    elif len(beseda) > 8: predolge += 1

if prekratke == 0 and predolge == 0: print("Esej je lep.")
else: print("Esej ni lep: %d prekratkih, %d predolgih besed." % (prekratke, predolge))
```

3. Pomanjkanje sendvičev

Koristno je imeti dve tabeli: v eni hranimo podatke o zalogi sendvičev posameznega tipa, v drugi pa število zahtev po sendvičih posameznega tipa. To drugo tabelo bomo potrebovali na koncu, da bomo lahko izpisali, kateri tip sendviča je bil največkrat zahtevan. Glede prve tabele pa se lahko vprašamo, ali naj v njej hranimo trenutno stanje zaloge (po vseh dosedanjih zahtevah) ali začetno stanje zaloge (tisto, ki smo ga na začetku izvajanja prebrali od uporabnika). Druga možnost pride bolj prav, ker drugače na koncu ne bomo vedeli, katerih sendvičev je bilo premalo (če bi videli le to, da je stanje zaloge tistega tipa sendvičev na koncu 0, iz tega še ne bi vedeli, ali je bil ta sendvič zahtevan kdaj po tistem, ko mu je zaloga že padla na 0 — šele to pa naredi razliko med tem, ali je bilo sendvičev tega tipa premalo ali pa ravno prav).

Na začetku torej preberimo zalogo in jo shranimo v tabelo zaloga, v tabeli s števci zahtev (stZahtev) pa inicializiramo vse elemente na 0. Nato po vrsti beremo zahteve in povečujemo števce zahtev, pri vsaki pa tudi preverimo, če je število zahtev tega tipa zdaj že preseglo začetno zalogo, tako da vemo, kaj odgovoriti uporabniku. Na koncu lahko s primerjavo obeh tabel ugotovimo, katerih sendvičev je bilo premalo, najpopularnejši tip sendviča pa ugotovimo tako, da poiščemo največjo vrednost v tabeli stZahtev in nato izpišemo tiste indekse, kjer se v tabeli pojavlja ta vrednost (tako bomo pravilno odkrili tudi primere, ko obstaja več enako popularnih najpopularnejših tipov).

```
#include <stdio.h>

int main()
{
  enum { N = 6 }; /* Število tipov sendvičev. */

  /* Preberimo zalogo in inicializirajmo števce zahtev na 0. */
  int zaloga[N], stZahtev[N]; bool premalo = false;
  for (int t = 0; t < N; t++)
```

```
  {
    stZahtev[t] = 0;
    printf("Zaloga sendvicev st. %d: ", t + 1);
    scanf("%d", &zaloga[t]);
  }
  /* Prebirajmo zahteve in odgovarjajmo nanje. */
  while (true)
  {
    printf("Pozdravljeni, kateri sendvic zelite? ");
    int t; scanf("%d", &t); if (t == 0) break;
    /* Popravimo števec zahtev tega tipa in preverimo,
       če so sendviči tega tipa še na zalogi. */
    if (++stZahtev[t - 1] > zaloga[t - 1])
      printf("Sendvicev tipa %d nam je zal zmanjkalo. "
             "Vec srece prihodnjic!\n", t), premalo = true;
    else
      printf("Izvolite sendvic st. %d. Dober tek!\n", t);
  }
  /* Izpišimo, katerih sendvičev je bilo premalo. */
  if (premalo) {
    printf("Premalo je bilo sendvicev st.");
    for (int t = 0; t < N; t++)
      if (stZahtev[t] > zaloga[t]) printf(" %d", t + 1);
    printf(".\n"); }
  /* Izpišimo, po katerih sendvičih je bilo največ zahtev. */
  int naj = 0; for (int t = 0; t < N; t++) if (stZahtev[t] > naj) naj = stZahtev[t];
  printf("Najbolj popularni so sendvici st.");
  for (int t = 0; t < N; t++) if (stZahtev[t] == naj) printf(" %d", t + 1);
  printf(".\n"); return 0;
}
```

Pri delu s tabelami je treba nekaj previdnosti pri indeksih: indeksi v tabeli gredo od 0 do 5, uporabnik pa vnaša števila od 1 do 6. Ena možnost je, da primerno prištevamo ali odštevamo 1, da preračunavamo med obema načinoma številčenja tipov sendvičev (to počne na primer gornja rešitev), druga možnost pa bi bila, da bi deklarirali tabeli s po 7 elementi, ki bi imeli torej indekse od 0 do 6, pri čemer elementa z indeksom 0 potem pač ne bi uporabljali.

Naloga ne pove natančno, ali naj vrstico „Premalo je bilo sendvičev št. . . . “ izpišemo tudi v primeru, če ni bilo nobenih sendvičev premalo. Gornji program jo izpiše le, če je kakšnih sendvičev res bilo premalo, to pa si že sproti (med odgovarjanjem na zahteve) označi v spremenljivki premalo.

Zapišimo našo rešitev še v pythonu:

```
N = 6
zaloga = [int(input("Zaloga sendvicev st. %d: " % t)) for t in range(N)]
stZahtev = [0] * N; premalo = False
while True:
  t = int(input("Pozdravljeni, kateri sendvic zelite? "))
  if not 1 <= t <= N: break
  stZahtev[t - 1] += 1
  if stZahtev[t - 1] > zaloga[t - 1]:
    print("Sendvicev tipa %d nam je zal zmanjkalo. "
          "Vec srece prihodnjic!" % t)
```

```
      premalo = True
   else: print("Izvolite sendvic st. %d. Dober tek!" % t)

if premalo: print("Premalo je bilo sendvicev tipa %s." %
   " ".join(str(t + 1) for t in range(N) if stZahtev[t] > zaloga[t]))
najvec = max(stZahtev)
print("Najbolj popularni so sendvici st. %s." %
   " ".join(str(t + 1) for t in range(N) if stZahtev[t] == najvec))
```

4. Prehod za pešce

Potrebovali bomo spremenljivko (v spodnji rešitvi je to stevec), v kateri štejemo pešce, ki so prečkali cesto. Ko se prižge rdeča luč, postavimo ta števec na 0; ko prečka cesto kak pešec, povečamo števec za 1; ko se prižge zelena, pa števec izpišemo. (Tako se bo števec sicer povečeval tudi pri tistih pešcih, ki prečkajo cesto pri zeleni luči, vendar nas to nič ne moti, saj ga bomo tako ali tako postavili nazaj na 0, ko se bo naslednjič prižgala rdeča luč.)

```
#include <stdio.h>

int main()
{
   int stevec = 0;
   while (true)
   {
      int dogodek = Dogodek();
      if (dogodek == 1) printf("%d\n", stevec);
      else if (dogodek == 2) stevec = 0;
      else stevec++;
   }
}
```

Zapišimo to rešitev še v pythonu:

```
stevec = 0
while True:
   dogodek = Dogodek()
   if dogodek == 1: print(stevec)
   elif dogodek == 2: stevec = 0
   else: stevec += 1
```

5. Pike za tisočice

Za začetek se sprehodimo v zanki po nizu in poiščimo indeks n, na katerem se v njem pojavlja decimalna vejica; če pa te sploh ni, bomo namesto nje za n vzeli dolžino niza. Zdaj lahko razmišljamo takole: piko za tisočice moramo vriniti pred znake $n-3$, $n-6$, $n-9$ in tako nazaj. Z drugimi besedami, pred znak i moramo vriniti piko natanko tedaj, če je razlika $n-i$ večkratnik števila 3. Izjema je le prvi znak niza ($i=0$), pred katerega pike ne vrivamo. Zdaj se lahko sprehodimo še enkrat po nizu od začetka do konca; pri vsakem indeksu preverimo, če moramo pred trenutni znak vriniti piko; če da, jo izpišemo, nato pa v vsakem primeru izpišemo še trenutni znak.

```
#include <stdio.h>

void IzpisStevila(const char *s)
{
  /* Poiščimo decimalno vejico ali konec niza. */
  int n = 0;
  while (s[n] && s[n] != ',') n++;

  /* Izpišimo ustrezno popravljen niz. */
  for (int i = 0; s[i]; i++)
  {
    /* Če smo levo od decimalne vejice (vendar ne na začetku niza)
       in je število znakov med trenutnim položajem in decimalno vejico
       večkratnik 3, vrinemo piko za tisočice. */
    if (i > 0 && i < n && (n - i) % 3 == 0) fputc('.', stdout);          (⋆)

    /* Izpišimo trenutni znak. */
    fputc(s[i], stdout);
  }
}
```

Slabost gornje rešitve je, da izpisuje vsak znak posebej; praviloma bo precej hitreje, če si najprej pripravimo celoten izhodni niz v pomnilniku in ga nato izpišemo v enem zamahu. Oglejmo si še primer takšne rešitve:

```
void IzpisStevila2(const char *s)
{
  /* Določimo položaj decimalne vejice (n) in dolžino niza (d). */
  int n = 0; while (s[n] && s[n] != ',') n++;
  int d = n; while (s[d]) d++;

  /* Pripravimo izhodni niz t. */
  char *t = new char[d + (n + 2) / 3 + 1]; int j = 0;
  for (int i = 0; i < d; )
  {
    if (i > 0 && i < n && (n - i) % 3 == 0)
      t[j++] = '.';    /* Vrinimo piko, kjer je to potrebno. */
    t[j++] = s[i++];   /* Skopirajmo trenutni znak. */
  }
  t[j] = 0; /* Znak za konec izhodnega niza. */

  /* Izpišimo izhodni niz in pospravimo za sabo. */
  fwrite(t, sizeof(*t), j, stdout);
  delete[] t;
}
```

Zapišimo obe rešitvi še v pythonu:

```
def IzpisStevila(s):
  n = s.find(',')
  if n < 0: n = len(s)
  for i in range(len(s)):
    if 0 < i < n and (n - i) % 3 == 0: sys.stdout.write('.')
    sys.stdout.write(s[i])

def IzpisStevila2(s):
  n = s.find(',')
  if n < 0: n = len(s)
  L = []
```

```
for i in range(len(s)):
    if 0 < i < n and (n − i) % 3 == 0: L.append('.')
    L.append(s[i])
print("".join(L))
```

Razmislimo še o različicah naloge, ki ju omenja opomba pod črto na str. 14. Pri (a) moramo izpisovati pike tudi desno od decimalne vejice, ne le levo od nje. Pred znak i moramo po novem vriniti piko tudi v naslednjem primeru: če leži desno od decimalne vejice (torej $i > n$) in je število znakov med njim in decimalno vejico (torej $i - n - 1$) večkratnik števila 3. Ker pa pred znak $i = n + 1$ ne smemo vriniti pike (saj bi ta pika prišla takoj za vejico, ki stoji na indeksu n), bomo pogoj $i > n$ spremenili v $i > n + 1$. Pogoj, da mora biti $i - n - 1$ večkratnik števila 3, lahko zapišemo tudi tako, da mora $i - n$ po deljenju s 3 dati ostanek 1. Tako torej vidimo, da moramo v naši zgornji rešitvi le dopolniti pogoj v vrstici $(\star)$:

```
if ((i > 0 && i < n && (n − i) % 3 == 0) ||
    (i > n + 1 && (i − n) % 3 == 1)) fputc('.', stdout);
```

Pri (b) naše vhodno število nima decimalne vejice, pri tisočicah pa moramo izpisovati izmenično pike in vejice. Ker decimalne vejice ni, nam n zdaj pomeni dolžino niza. Pogoj za izpisovanje pik ali vejic je tak kot v prvotni različici naloge, le člen $i < n$ lahko zavržemo, saj bo vedno izpolnjen. Preden pa piko ali vejico res izpišemo, preverimo še, ali je $n - i$ tudi večkratnik 6, ne le večkratnik 3; če je, moramo izpisati vejico, sicer pa piko. Vrstico $(\star)$ prvotne rešitve moramo torej spremeniti takole:

```
if (i > 0 && (n − i) % 3 == 0) fputc((n − i) % 6 == 0 ? ',' : '.', stdout);
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Zvončki

Če s trenutnega položaja lahko zaigramo naslednji ton naše melodije, potem ni razloga, da bi se pred igranjem tega tona kam premaknili; kamorkoli se že imamo namen premakniti, se lahko tja premaknemo tudi po tem tonu, njega pa zaigramo še s trenutnega položaja. Ta razmislek nam pove, da moramo o premikih razmišljati šele takrat, ko s trenutnega položaja ne dosežemo zvončka, ki ga potrebujemo za naslednji ton (torej ko ni primernega zvončka niti v skupini, pred katero stojimo, niti v njeni levi ali desni sosedi).

Ko pa se vendarle moramo premakniti, si moramo nekako izbrati novi položaj. Takrat si je pametno novi položaj izbrati tako, da se nam potem čim dlje ne bo treba spet premikati. O tem se lahko prepričamo takole. Recimo, da primerjamo dva položaja, a in b; naj bo n_a število naslednjih tonov melodije, ki jih bomo lahko odigrali s položaja a, in podobno n_b za položaj b; in recimo, da je $n_a > n_b$. Ali je mogoče, da bi se bilo vendarle bolje zdaj premakniti na b namesto na a? Če se premaknemo na b, bomo tam zaigrali naslednjih n_b tonov, potem pa se bomo morali spet premakniti, recimo k neki skupini c. Toda če se namesto tega zdaj premaknemo na a, bomo tudi tam lahko zaigrali naslednjih n_b tonov (pravzaprav celo n_a tonov, kar je več kot n_b), potem pa se lahko še vedno premaknemo na c, če se že hočemo. Torej gotovo nismo nič na slabšem, če se premaknemo na a namesto na b.

Z enakim razmislekom si izberemo tudi začetni položaj; postavimo se k tisti skupini, pri kateri bomo lahko zaigrali največ tonov z začetka melodije, preden se bomo morali prvič premakniti.

Za čim učinkovitejše preverjanje tega, ali lahko nek ton zaigramo z določenega položaja, bi si lahko pred začetkom našega postopka pripravili razpršeno tabelo (*hash table*), v kateri bi kot ključe shranili pare $\langle$*številka skupine*, *zvonček*$\rangle$ za vse zvončke vseh skupin; tako bi lahko v $O(1)$ časa preverili, ali je iskani zvonček v trenutni skupini (ali pa v eni od sosednjih dveh skupin). Če n (število različnih zvončkov) ni prevelik, pa bi lahko namesto razpršene tabele pripravili kar bitno karto — tabelo $n \times m$ bitov, ki bi nam za vsako kombinacijo tona in skupine povedali, ali je v tisti skupini zvonček, ki igra tisti ton.

2. Rastlinjak

Naloga zahteva, da vsako meritev izpišemo le prvič, ko se pojavi, njene ponovitve pa ignoriramo. Ker se lahko v vhodnih podatkih prepletajo meritve z različnih merilnikov, si moramo za vsak merilnik zapomniti čas zadnje meritve, ki smo jo dobili od njega (spodnji program ima v ta namen tabelo zadnjiCas). Ko pride nova meritev, jo primerjamo s časom zadnje meritve istega merilnika in če sta enaka, vemo, da gre za ponovitev že videne meritve. Če pa te meritve še nismo videli, jo lahko zdaj izpišemo, njen čas pa si zapomnimo v zadnjiCas.

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

int main()
{
```

```
  enum { N = 9 };
  string zadnjiCas[N];
  while (true)
  {
    string cas, temp; int n;
    /* Preberimo naslednjo meritev. */
    cin >> cas >> n >> temp;
    if (! cin.good()) break; /* Najbrž smo na koncu vhoda. */
    /* Ali smo to meritev že videli? */
    if (zadnjiCas[n - 1] == cas) continue;
    /* Če ne, jo zapišimo in si jo zapomnimo. */
    cout << n << " " << temp << endl;
    zadnjiCas[n - 1] = cas;
  }
  return 0;
}
```

3. Labirint

Med branjem Mihovih premikov je koristno vzdrževati seznam njegovih možnih trenutnih položajev (v spodnjem programu bo to tabela kandidati, število kandidatov v njej pa hrani spremenljivka stKand). Na začetku, preden se prvič premakne, ne vemo o njegovem položaju ničesar drugega kot to, da je trenutno na nekem prostem polju, torej na seznam kandidatov dodamo vsa prosta polja. To lahko počnemo spotoma, medtem ko beremo opis labirinta; ta opis si tudi zapomnimo v neki tabeli, ker ga bomo kasneje še potrebovali.

Nato beremo premike enega po enega; pri vsakem premiku si pripravimo par števil $(\triangle x, \triangle y)$, ki pove, kako se pri tem premiku spremenita Mihovi koordinati. Preglejmo vse kandidate v seznamu in pri vsakem izračunamo novi položaj po tem premiku. Če bi bil novi položaj na neprehodnem polju ali pa zunaj labirinta, kandidata pobrišemo iz seznama (ker očitno Miha ni začel svoje poti na tistem začetnem položaju, iz katerega je ta kandidat nastal).

Ko pridemo do konca zaporedja premikov, moramo seznam kandidatov za Mihov trenutni položaj le še izpisati. Pri izpisu pazimo na to, da naloga šteje koordinate od 1 naprej, za indekse v tabele pa je koristno delati s koordinatami od 0 naprej.

Spodnji program hrani kandidata (x, y) kot celo število $x \cdot w + y$, iz česar ni težko nazaj izračunati x in y; lahko pa bi namesto tega hranili tudi majhne strukture z dvema ločenima atributoma za x in y.

```
#include <stdio.h>

int main()
{
  enum { MaxW = 100, MaxH = 100 };
  int w, h, lab[MaxH][MaxW], kandidati[MaxW * MaxH], stKand = 0;
  /* Preberimo opis labirinta in pripravimo seznam kandidatov za
     Mihov trenutni položaj. Na začetku so to kar vsa prosta polja. */
  scanf("%d %d", &h, &w);
  for (int y = 0; y < h; y++) for (int x = 0; x < w; x++) {
    scanf("%d", &lab[y][x]);
    if (! lab[y][x]) kandidati[stKand++] = y * w + x; }
```

```
  /* Preberimo Mihove premike in sproti popravljajmo seznam kandidatov. */
  while (true)
  {
    /* Poglejmo, za kakšen premik gre. */
    char premik[5]; scanf("%s", premik);
    int dx, dy;
    if (premik[0] == 'S') dx = 0, dy = −1;
    else if (premik[0] == 'J') dx = 0, dy = 1;
    else if (premik[0] == 'V') dx = 1, dy = 0;
    else if (premik[0] == 'Z') dx = −1, dy = 0;
    else break;
    /* Preglejmo vse kandidate in jih ustrezno popravimo. */
    for (int i = 0; i < stKand; )
    {
      int x = kandidati[i] % w, y = kandidati[i] / w;
      x += dx; y += dy;
      /* Zdaj sta x in y koordinati, kot bi ju Miha imel po tem premiku.
         Če ta položaj ni veljaven, kandidata pobrišimo, sicer pa
         na njegovo mesto v tabeli vpišimo novi položaj. */
      if (x >= 0 && x < w && y >= 0 && y < h && ! lab[y][x])
        kandidati[i++] = y * w + x;
      else
        kandidati[i] = kandidati[−−stKand];
    }
  }
  /* Izpišimo rezultate. */
  for (int i = 0; i < stKand; i++)
    printf("%d %d\n", kandidati[i] % w + 1, kandidati[i] / w + 1);
  return 0;
}
```

To rešitev bi se dalo še izboljšati. Na primer, lahko najprej preberemo celotno zaporedje premikov in sproti računamo, kakšen bi bil po posameznem premiku Mihov odmik od začetnega položaja (če ga labirint ne bi nič oviral); recimo, da po i-tem premiku Miha stoji $\triangle x_i$ polj desno in $\triangle y_i$ polj dol od začetnega položaja. Zapomnimo si največji in najmanjši odmik v smeri x in podobno v smeri y. Zdaj lahko razmišljamo takole: če Miha začne v (x_0, y_0), se mu med potjo x-koordinata giblje od $x_0 - \min_i \triangle x_i$ do $x_0 + \max_i \triangle x_i$; če nočemo, da pade iz labirinta, mora veljati $1 \le x_0 - \min_i \triangle x_i$ in $x_0 + \max_i \triangle x_i \le w$ oz. z drugimi besedami $1 + \min_i \triangle x_i \le x_0 \le w - \max_i \triangle x_i$. S prav takim razmislekom lahko dobimo podoben pogoj tudi za y_0. Tako torej vemo, s katerega območja smemo vzeti začetni položaj; s tistimi (x_0, y_0), ki ne ustrezajo tema pogojema, se nam sploh ni treba ukvarjati. Tako je začetni nabor kandidatov nekoliko manjši, pa tudi med premikanjem nam ni treba več preverjati, ali Miha pade iz labirinta, pač pa le še to, ali se zaleti v neprehodno polje.

Kljub tej izboljšavi se v najslabšem primeru lahko zgodi, da ima naš algoritem časovno zahtevnost $O(nwh)$. Če je Mihova pot zelo dolga, se prej ali slej mora začeti dogajati, da obišče isto polje po večkrat (npr. ko naredi $n \ge w \cdot h$ korakov). V našem zaporedju odmikov $(\triangle x_i, \triangle y_i)$ se to pokaže tako, da nastopi enak par odmikov pri dveh ali več različnih i. Za preverjanje, ali je pot (pri nekem začetnem položaju) možna, je dovolj že, če tak odmik pregledamo le enkrat. Koristno je torej

iz zaporedja $(\triangle x_i, \triangle y_i)$ pobrisati duplikate (namesto kot zaporedje si ga lahko zdaj predstavljamo kot množico), nato pa, namesto da gremo za vsak možni začetni položaj (x, y) po celotnem zaporedju Mihovih korakov, gremo le po naši množici odmikov $(\triangle x_i, \triangle y_i)$ in za vsakega od njih preverimo, če je polje $(x + \triangle x_i, y + \triangle y_i)$ v labirintu prehodno. Časovna zahtevnost bo zdaj $O(wh \min\{n, wh\})$, kar je v najslabšem primeru $O(w^2h^2)$.

Še ena izboljšava pa je naslednja. Labirint si lahko predstavljamo kot funkcijo: $L(x, y) = 0$, če je polje (x, y) prosto, in 1, če je zazidano. Podobno si lahko tudi množico odmikov predstavljamo kot funkcijo: $O(x, y)$ naj bo 1, če pri kakšnem i velja $x = -\triangle x_i$ in $y = -\triangle y_i$, sicer pa 0. Zdaj definirajmo $f(x, y) := \sum_u \sum_v L(u, v)O(x - u, y - u)$. Produkt v tej formuli nam pove, ali bi imel Miha, če bi začel svojo pot na polju (x, y), kdaj težave na polju (u, v); težave ima, če je tisto polje neprehodno (torej $L(u, v) = 1$) in če se kdaj mora premakniti nanj (torej če ima kdaj odmik $(u - x, v - y)$, to pa je tedaj, ko je $O(x - u, y - u) = 1$). Vrednost $f(x, y)$ nam torej pove, kolikokrat bi se Miha znašel na neprehodnem polju, če bi svojo pot začel na (x, y). Naloga torej pravzaprav sprašuje po tem, pri katerih (x, y) je $f(x, y) = 0$. Funkciji f, definirani na zgoraj opisani način, pravimo *konvolucija* funkcij L in O; lepo pri tem pogledu na naš problem je, da obstajajo postopki (npr. hitra Fourierjeva transformacija), s katerimi lahko takšno konvolucijo za vse (x, y) izračunamo v $O(wh \log wh)$ časa namesto v $O(w^2h^2)$ časa.

4. Šifriranje

Označimo prvotni niz (pred šifriranjem) s p, šifriranega pa s s; recimo, da sta dolga po n znakov, indekse znakov pa štejmo od 0 do $n - 1$, tako kot je to v navadi v C-ju in podobnih jezikih.

Pri šifriranju je posamezni znak $s[i]$ nastal tako, da smo $s[i]$ ciklično zamaknili za $k[i \bmod 5]$ mest naprej; pri tem si $k[0..4]$ predstavljajmo kot številsko predstavitev našega ključa (kjer ima ključ črko `a`, naj ima tabela k vrednost 1, kjer ima ključ črko `b`, naj ima tabela k vrednost 2 itd.). Če si znake nizov s in p namesto kot črke predstavljamo kot števila od 0 do 25, lahko šifriranje opišemo s preprosto formulo:

$$s[i] = (p[i] + k[i \bmod 5]) \bmod 26.$$

Zadnjih pet (pravzaprav celo zadnjih osem) znakov niza p poznamo, ker vemo, da se sporočilo konča na „`Lp, Janez`“; niz s pa poznamo v celoti. Za $i = n - 5, \ldots, n - 1$ torej poznamo tako $s[i]$ kot $p[i]$, zato lahko iz gornje formule izrazimo znake ključa:

$$k[i \bmod 5] = (s[i] - p[i]) \bmod 26.$$

Ker bomo to naredili za pet zaporednih vrednosti i, bodo ostanki $i \bmod 5$ gotovo pokrili vse vrednosti od 0 do 4, tako da bomo dobili cel ključ.

Podobno lahko iz prve formule izrazimo $p[i]$ in tako dobimo formulo za dešifriranje, ki jo bomo lahko uporabljali, ko bomo poznali ključ:

$$p[i] = (s[i] - k[i \bmod 5]) \bmod 26.$$

V praksi je treba biti pri uporabi zadnjih dveh formul malo previden. Obe sta oblike $(x - y) \bmod 26$; če je razlika $x - y$ negativna, operator za računanje ostankov

po deljenju — na primer % v C-ju in podobnih jezikih — praviloma ne bo dajal rezultatov, kot jih tu pričakujemo.[5] Zato je bolje namesto $x - y$ vzeti $x - y + 26$, kar ima enak ostanek po deljenju s 26, je pa zagotovo večje ali enako 0.

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

void Desifriraj(const char *s)
{
  const char konec[] = "Janez"; /* Zadnjih 5 znakov dešifriranega niza. */
  char kljuc[5];
  int n = strlen(s);
  /* Rekonstruirajmo številsko predstavitev ključa. */
  for (int i = n - 5; i < n; i++)
    kljuc[i % 5] = (s[i] - konec[i - (n - 5)] + 26) % 26;
  /* Dešifrirajmo znake niza in jih izpisujmo. */
  for (int i = 0; i < n; i++)
  {
    char c = s[i]; int k = 26 - kljuc[i % 5];
    /* Če je c črka, jo ciklično zamaknimo za k mest,
       sicer naj c ostane nespremenjen. */
    if (c >= 'A' && c <= 'Z') c = (c - 'A' + k) % 26 + 'A';
    else if (c >= 'a' && c <= 'z') c = (c - 'a' + k) % 26 + 'a';
    fputc(c, stdout);
  }
  fputc('\n', stdout);
}

int main()
{
  char s[10002]; fgets(s, sizeof(s), stdin);
  /* Pobrišimo znak za konec vrstice, ki ga je fgets pustil v nizu. */
  char *p = strchr(s, '\n'); if (p) *p = 0;
  Desifriraj(s); return 0;
}
```

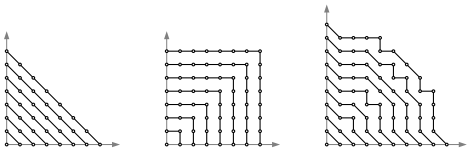
Kot zanimivost omenimo, da se postopek šifriranja, ki ga uporablja ta naloga, imenuje *Vigènerjeva šifra*.

5. Neskončna pokrajina

Če bi poznali velikost naše pravokotne površine, bi lahko vse celoštevilske pare (x, y) na njej pregledali tako, da bi šli sistematično npr. po vrsticah od zgoraj navzdol, v vsaki vrstici pa od leve proti desni; tako bi sčasoma obiskali vse točke v pravokotniku. Ker pa njegove velikosti ne poznamo, bi ta postopek odpovedal že v prvi vrstici, saj ne moremo vedeti, kdaj lahko z njo končamo in se premaknemo v naslednjo vrstico.

Namesto tega moramo najti nek postopek, ki obiskuje točke (x, y) za vse nenegativne celoštevilske x in y v takem vrstnem redu, da pride vsaka točka na vrsto po končno mnogo korakih. Nekaj možnosti kaže naslednja slika:

[5] Za več o tem glej npr. rešitev naloge 1997.2.1, str. 309 v zbirki *Rešenih nalog s srednješolskih računalniških tekmovanj 1988–2004*.



Verjetno najpreprostejša je prva rešitev, ki pregleduje točke po diagonalah. Za vsak $d \geq 0$ imamo diagonalo od točke $(0, d)$ do $(d, 0)$. Točke na njej imajo koordinate oblike $(x, d - x)$. Vse, kar potrebujemo, je zanka po d in v njej še vgnezdena zanka po x:

```
bool Preizkusi(int x, int y, int ciljnaVisina)
{
  if (Visina(x, y) != ciljnaVisina) return false;
  printf("%d %d\n", x, y); return true;
}

void PoisciPoDiagonalah(int ciljnaVisina)
{
  for (int d = 0; ; d++)
    for (int x = 0; x <= d; x++)
      if (Preizkusi(x, d - x, ciljnaVisina)) return;
}
```

Klic funkcije Visina in izpis rezultata (če smo našli pravo točko) smo preselili v ločen podprogram Preizkusi zato, ker nam bo to prihranilo nekaj ponavljanja približno iste kode tudi v naslednjih rešitvah.

Druga možnost na gornji sliki je, da si predstavljamo ravnino kot zaporedje vse večjih kvadratov s spodnjim levim kotom v $(0, 0)$ in zgornjim desnim v (a, a) za vse večje a. Kvadrat s stranico a se od svojega predhodnika, kvadrata s stranico $a - 1$, razlikuje po tem, da pokrije tudi točke (a, t) in (t, a) za $t = 0, 1, \ldots, a$. Lahko gremo torej v zanki po a in pri vsakem a obiščemo vse tiste točke, ki jih ima kvadrat s stranico a, njegov predhodnik s stranico $a - 1$ pa ne:

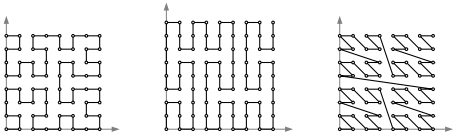
```
void PoisciPoGnomonih(int ciljnaVisina)
{
  for (int a = 0; ; a++)
  {
    for (int y = 0; y <= a; y++)
      if (Preizkusi(a, y, ciljnaVisina)) return;
    for (int x = a - 1; x >= 0; x--)
      if (Preizkusi(x, a, ciljnaVisina)) return;
  }
}
```

Če namesto vse večjih kvadratov pri takšnem razmisleku uporabimo vse večje kroge (s središčem v koordinatnem izhodišču), dobimo tretjo sliko zgoraj. Namesto stranice kvadrata a zdaj gledamo polmer kroga r; pri polmeru r moramo obiskati vse točke, ki ležijo na kolobarju med krogoma s polmerom $r-1$ in r (lahko tudi na krogu s polmerom r, ne pa na krogu s polmerom $r - 1$, kajti tiste smo obiskali že prej). Tak pregled lahko začnemo v točki $(r, 0)$, nato pa razmišljamo takole: iz trenutne točke (x, y) se premaknemo v $(x-1, y)$, če ta točka leži v našem kolobarju; če pa ne,

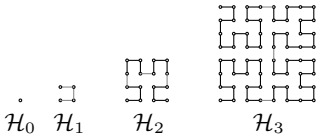
se premaknemo v $(x, y+1)$; če tudi ta ne leži v kolobarju (ker je od koordinatnega izhodišča oddaljena za več kot r), pa se premaknemo v $(x-1, y-1)$.

```
void PoisciPoKolobarjih(int ciljnaVisina)
{
  for (int r = 0; ; r++)
  {
    int x = r, y = 0;
    while (x >= 0)
    {
      if (Preizkusi(x, y, ciljnaVisina)) return;
      if ((x - 1) * (x - 1) + y * y > (r - 1) * (r - 1)) { x--; continue; }
      y++; if (x * x + y * y > r * r) x--;
    }
  }
}
```

Pri pregledovanju vseh točk ravnine si lahko pomagamo tudi s fraktalnimi krivuljami. Naslednja slika kaže nekaj primerov: Hilbertovo, Peanovo in Z-krivuljo.

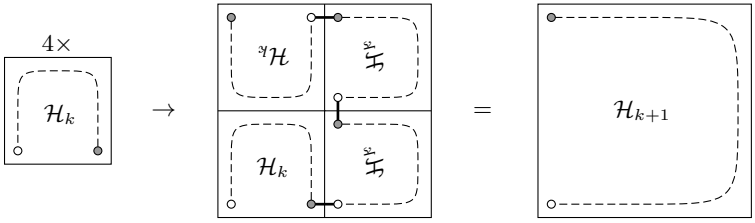


Primer na prvi od teh slik je začetni del Hilbertove krivulje, ene od najbolj znanih fraktalnih krivulj. Dobimo jo tako, da začnemo z eno samo točko, nato pa na vsakem koraku vzamemo štiri kopije prejšnje krivulje in jih povežemo (te povezave so na spodnji sliki prikazane z tanjšimi črtami):



Na sliki vidimo krivulje od $\mathcal{H}_0$ do $\mathcal{H}_3$, očitno pa bi lahko na ta način nadaljevali še poljubno dolgo.

Vidimo lahko, da krivulja $\mathcal{H}_k$ obišče vse tiste točke, ki imajo obe koordinati z območja od 0 do $2^k - 1$. Krivulja $\mathcal{H}_k$ se začne v točki $(0, 0)$, konča pa v $(2^k - 1, 0)$, če je k sod, oz. v $(0, 2^k - 1)$, če je k lih. Pri sodem k lahko štiri kopije krivulje $\mathcal{H}_k$ takole zložimo v $\mathcal{H}_{k+1}$:



Enak razmislek je uporaben tudi pri lihem k, če like pred in po zlaganju prezrcalimo prek simetrale lihih kvadrantov (z drugimi besedami, če zamenjamo x- in y-koordinate).

Tako lahko sestavimo naslednji postopek, ki nam za poljuben $n \geq 0$ izračuna koordinati n-te točke na Hilbertovi krivulji:

```
void TockaNaHilbertoviKrivulji(int n, int &x, int &y)
{
  int k = 0, u = 1; x = 0; y = 0;
  while (n > 0)
  {
    int xx, yy, t;
    if (k & 1) t = x, x = y, y = t;
    switch (n & 3)
    {
    case 0: xx = x; yy = y; break;
    case 1: xx = u + y; yy = x; break;
    case 2: xx = u + y; yy = u + x; break;
    case 3: xx = u - 1 - x; yy = 2 * u - 1 - y; break;
    }
    x = xx; y = yy;
    if (k & 1) t = x, x = y, y = t;
    n >>= 2; u <<= 1; k++;
  }
}
```

Bite n-ja pregledujemo od nižjih proti višjim; na vsakem koraku odčitamo naslednja dva bita n-ja, ki nam povesta, v kateri od štirih kopij krivulje $\mathcal{H}_k$ se nahaja naša točka. Glavni podprogram mora le računati točke v zanki in jih preverjati, dokler ne najde prave:[6]

```
void PoisciPoHilbertoviKrivulji(int ciljnaVisina)
{
  for (int n = 0; ; n++)
  {
    int x, y; TockaNaHilbertoviKrivulji(n, x, y);
    if (Preizkusi(x, y, ciljnaVisina)) return;
  }
}
```

Peanovo krivuljo (druga med tremi fraktalnimi krivuljami zgoraj) dobimo na podoben način, le da na vsakem koraku namesto štirih kopij prejšnje krivulje vzamemo devet kopij in jih zložimo v kvadrat 3×3, v katerem si sledijo v obliki črke N. Pri tem moramo tiste v srednjem stolpcu ($\mathsf{nx} = 1$ v spodnjem podprogramu) prezrcaliti prek vodoravne osi, tiste v srednji vrstici ($\mathsf{ny} = 1$) pa prek navpične osi, da bo krivulja lepo zvezna. (Za namen naše naloge to sicer ni nujno potrebno; naloga zahteva le, da obiščemo vse točke ravnine.) Tako dobimo naslednjo rešitev:

```
void TockaNaPeanoviKrivulji(int n, int &x, int &y)
{
  int u = 1; x = 0; y = 0;
```

[6] S Hilbertovo krivuljo smo se na naših tekmovanjih že srečali; gl. nalogo 2002.3.7, str. 497 v zbirki *Rešenih nalog s srednješolskih računalniških tekmovanj 1988–2004*.

```
  while (n > 0)
  {
    int ny = n % 3; n /= 3; int nx = n % 3; n /= 3;
    if (nx == 1) y = u - 1 - y, ny = 2 - ny;
    if (ny == 1) x = u - 1 - x;
    x += u * nx; y += u * ny; u *= 3;
  }
}

void PoisciPoPeanoviKrivulji(int ciljnaVisina)
{
  for (int n = 0; ; n++)
  {
    int x, y; TockaNaPeanoviKrivulji(n, x, y);
    if (Preizkusi(x, y, ciljnaVisina)) return;
  }
}
```

Ostane nam še tretja izmed zgoraj omenjenih fraktalnih krivulj, to je Z-krivulja. Do nje lahko pridemo zelo preprosto: če nas zanima n-ta točka (za $n \geq 0$) na krivulji, zapišimo n v dvojiškem zapisu, torej kot zaporedje bitov: $n = (n_{k-1}n_{k-2}\ldots n_1n_0)_2$. Če je treba, vrinimo na levi še eno vodilno ničlo, tako da je k (število bitov v tem zaporedju) sod. Zdaj to zaporedje preprosto razpletimo na dve: iz bitov na sodih mestih sestavimo x-koordinato $x = (n_{k-2}n_{k-4}\ldots n_2n_0)_2$, iz bitov na lihih mestih pa y-koordinato $y = (n_{k-1}n_{k-3}\ldots n_3n_1)_2$. Tako dobimo naslednjo rešitev:

```
void TockaNaZKrivulji(int n, int &x, int &y)
{
  x = 0; y = 0;
  for (int b = 1; n > 0; b <<= 1)
  {
    if (n & 1) x |= b; n >>= 1;
    if (n & 1) y |= b; n >>= 1;
  }
}

void PoisciPoZKrivulji(int ciljnaVisina)
{
  for (int n = 0; ; n++)
  {
    int x, y; TockaNaZKrivulji(n, x, y);
    if (Preizkusi(x, y, ciljnaVisina)) return;
  }
}
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Back from the Klondike

V nalogi se skriva problem najkrajše poti v grafu, pri čemer dolžino poti merimo le s številom korakov na njej (številom povezav, vse povezave pa štejejo za enako dolge). Zato jo lahko rešujemo z iskanjem v širino.

V tabeli `d` bomo za vsako doseženo polje hranili dolžino najkrajše poti od začetnega položaja (zgornjega levega polja) do njega; poleg tega pa imamo še vrsto (v spodnjem programu je to tabela `q`), v kateri hranimo polja, do katerih že poznamo najkrajšo pot, nismo pa še pregledali, kam lahko iz tega polja pridemo v naslednjem koraku.

V zanki jemljemo po eno polje iz vrste in zanj pregledamo, kam lahko pridemo iz njega v enem koraku. Možni premiki so največ štirje (za vsako smer po eden, če z njim ne bi padli iz mreže). Pri vsakem premiku pogledamo, če nas je pripeljal v neko tako polje, do katerega prej še nismo prišli; če je tako, si zapomnimo dolžino te poti v tabeli `d` in dodamo novo polje na konec vrste `q`. Tako bomo sčasoma pregledali vsa tista polja, do katerih je iz začetnega polja sploh mogoče priti.

Na koncu tega postopka imamo v tabeli `d` dolžine najkrajših poti od začetnega polja do vseh ostalih, do katerih je iz njega sploh mogoče priti; pri nedosegljivih poljih pa je v `d` še vedno vrednost -1, s katero smo to tabelo na začetku inicializirali.

```
#include <stdio.h>

const int DX[] = { -1, 1, 0, 0 }, DY[] = { 0, 0, -1, 1 };
enum { MaxW = 1000, MaxH = 1000 };
int a[MaxH][MaxW], d[MaxW][MaxH], q[MaxW * MaxH];

int main()
{
  /* Preberimo vhodne podatke in inicializirajmo tabelo dolžin poti. */
  FILE *f = fopen("klondike.in", "rt");
  int w, h; fscanf(f, "%d %d", &w, &h);
  for (int y = 0; y < h; y++) for (int x = 0; x < w; x++) {
    fscanf(f, "%d", &a[y][x]); d[y][x] = -1; }
  fclose(f);

  /* Na začetku poznamo le pot dolžine 0 do polja v zgornjem levem kotu. */
  d[0][0] = 0; int glava = 0, rep = 0; q[rep++] = 0;

  /* Od tam nadaljujmo z iskanjem v širino. */
  while (glava < rep)
  {
    int x = q[glava] % w, y = q[glava] / w; glava++;

    /* Poskusimo se iz (x, y) premakniti v vse štiri smeri za a[y][x]. */
    for (int smer = 0; smer < 4; smer++)
    {
      /* Izračunajmo novi položaj (xx, yy) po tem premiku. */
      int xx = x + DX[smer] * a[y][x], yy = y + DY[smer] * a[y][x];

      /* Ali je novi položaj sploh znotraj mreže? */
      if (xx < 0 || yy < 0 || xx >= w || yy >= h) continue;

      /* Ali smo to polje dosegli že kdaj prej? */
      if (d[yy][xx] >= 0) continue;
```

```
        /* Zapomnimo si dolžino poti do (xx, yy) in ga dodajmo v vrsto. */
        d[yy][xx] = d[y][x] + 1; q[rep++] = yy * w + xx;
      }
    }
    /* Izpišimo rezultat. */
    f = fopen("klondike.out", "wt");
    fprintf(f, "%d\n", d[h - 1][w - 1]);
    fclose(f); return 0;
}
```

2. Trojane

Recimo, da bi se ves čas peljali z maksimalno hitrostjo (torej 1 km na u sekund). Kakšne težave bi imeli zaradi tega? Edini trenutek, ko imamo lahko zaradi take vožnje težave, je tedaj, ko dosežemo končno točko t_i kakšnega radarja. Če se takrat izkaže, da je od trenutka, ko smo prečkali točko s_i (torej začetno točko istega radarja), minilo manj kot v_i časa, moramo tik pred točko t_i malo počakati, dokler ta čas ne mine (lahko si predstavljamo, da avtomobil ustavimo ali pa ga vsaj upočasnimo na neko zelo nizko hitrost).

Da bomo simulirali tak potek vožnje, je koristno zložiti začetne in končne točke vseh omejitev v en seznam (v spodnjem programu je to vektor `tocke`) in ga urediti po koordinati. Če na isti x-koordinati ležita tako začetna točka ene omejitve kot končna točka neke druge omejitve, postavimo v našem urejenem seznamu končno točko pred začetno, kajti končne točke so tiste, ki lahko vplivajo na čas, ob katerem bomo prečkali to x-koordinato (ker moramo zaradi nekaterih končnih točk upočasniti oz. čakati, preden jih dosežemo).

Za vsako točko našega seznama bomo poleg x-koordinate hranili še indeks omejitve, ki ji pripada, in podatek o tem, ali je to začetek ali konec tiste omejitve. V ločeni tabeli (spodnji program ima v ta namen vektor `omejitve`) pa za vsako omejitev hranimo najzgodnejši čas, ob katerem smemo prečkati končno točko te omejitve. Na začetku bodo v tej tabeli le časi v_i, ko pa prečkamo začetek neke omejitve, bomo ustreznemu elementu te tabele prišteli trenutni čas (dobljena vsota je ravno najzgodnejši čas, ob katerem smemo prečkati končno točko te omejitve).

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

int main()
{
  struct Tocka { int x, omejitev, zacetek; };
  vector<Tocka> tocke; vector<long long> omejitve;

  /* Preberimo vhodne podatke. */
  int n, m, u;
  FILE *f = fopen("trojane.in", "rt");
  fscanf(f, "%d %d %d", &m, &n, &u);
  tocke.reserve(2 * n); omejitve.resize(n);

  for (int i = 0; i < n; i++)
  {
    int si, ti, vi; fscanf(f, "%d %d %d", &si, &ti, &vi);
```

```
    /* Za vsako omejitev dodajmo začetno in končno točko v vektor „tocke“. */
    tocke.push_back({si, i, 1}); tocke.push_back({ti, i, 0});
    /* omejitve[i] sprva vsebuje v_i, kasneje pa mu bomo prišteli še čas,
       ob katerem bomo dosegli s_i, tako da bo dobljena vsota pomenila najzgodnejši
       čas, ob katerem smemo prevoziti točko t_i. */
    omejitve[i] = vi;
  }
  fclose(f);
  /* Uredimo točke po koordinati; če jih je več na isti koordinati,
     postavimo končne pred začetne. */
  sort(tocke.begin(), tocke.end(), [] (const Tocka &a, const Tocka &b) {
    if (a.x != b.x) return a.x < b.x; else return a.zacetek < b.zacetek; });
  /* Odsimulirajmo vožnjo. */
  long long cas = 0;
  for (int i = 0; i <= 2 * n; i++)
  {
    int razdalja = (i == 2 * n ? m : tocke[i].x) - (i == 0 ? 0 : tocke[i - 1].x);
    cas += (long long) u * razdalja; /* Čas vožnje od i - 1 do i. */
    if (i < 2 * n) /* Upoštevajmo morebitno čakanje tik pred i. */
    {
      const Tocka &t = tocke[i];
      if (t.zacetek) omejitve[t.omejitev] += cas;
      else cas = max(cas, omejitve[t.omejitev]);
    }
  }
  /* Izpišimo rezultat. */
  f = fopen("trojane.out", "wt");
  fprintf(f, "%lld\n", cas);
  fclose(f); return 0;
}
```

Oglejmo si še malo temeljitejši dokaz tega, da je naša požrešna rešitev res optimalna. Recimo, da obstaja še nek drug potek, ki doseže konec odseka (točko m) prej kot naš in je tudi skladen z vsemi omejitvami. Našemu poteku (kot ga sestavi zgoraj opisana rešitev) recimo P, temu drugemu pa P'.

Začetne in končne točke omejitev nam razbijejo naš odsek $[0, m]$ na krajše intervale (kot smo videli tudi v gornji rešitvi): $0 = x_0 < x_1 < \ldots < x_k = m$. Opazimo lahko, da na skladnost poteka z omejitvami nič ne vpliva to, kako ta potek vozi znotraj takega intervala, ampak le to, ob katerem času prečka krajišča intervalov.

Gotovo pri nekem i velja, da P' doseže t_i prej kot P, kajti če to ne bi veljalo, potem bi oba poteka dosegla konec zadnje omejitve istočasno (ali pa P' celo kasneje kot P), od tam pa P vozi ves čas z maksimalno hitrostjo, torej ni mogoče, da bi P' dosegel cilj pred njim. Točka t_i pa je seveda ena od x_j za nek j. Vzemimo zdaj najmanjši tak j, pri katerem P' doseže x_j prej kot P. Gotovo je $j > 0$, saj sta začela (pri $x_0 = 0$) oba ob istem času 0. Pri $j - 1$ torej še velja, da oba poteka prečkata x_{j-1} ob istem času. Zakaj je torej P za pot od x_{j-1} do x_j porabil več časa kot P'?

P ves čas vozi z največjo možno hitrostjo, razen če mora čakati tik pred neko t_i; torej je edina možnost, zakaj je P' prevozil interval $[x_{j-1}, x_j]$ hitreje kot P, ta, da je moral P pred točko x_j čakati zato, ker je bila ta točka konec ene ali več omejitev, torej oblike $x_j = t_i$. Toda začetek vsake take omejitve, s_i, leži še levo od x_j, torej je to še ena od tistih točk, ki sta jih oba poteka prevozila istočasno, recimo ob času

u_i. Potek P je čakal le tako dolgo, da je točko x_j prečkal ob času $\max_i\{u_i + v_i\}$, pri čemer gre i po vseh tistih omejitvah, ki se končajo v x_j. Ker je P' prečkal začetke vseh teh omejitev ob istih časih u_i kot potek P, to pomeni, da tudi točke x_j ne sme prečkati prej kot ob času $\max_i\{u_i + v_i\}$. Tako smo prišli v protislovje: najprej smo rekli, da je P' skladen z omejitvami; nato smo ugotovili, da doseže x_j prej kot P; zdaj pa smo videli, da je to slednje mogoče le, če prekrši neko omejitev. Torej ni mogoče, da bi obstajal tak P', ki bi bil skladen z omejitvami in bi dosegel konec odseka hitreje kot P.

3. V soju žarometov

Žaromet v točki (x_i, y_i) osvetljuje na tleh interval x-koordinat od vključno $x_i - y_i$ do vključno $x_i + y_i$; ta žaromet prispeva po c_i k osvetljenosti vsake točke tega intervala. Če se torej v mislih premikamo po x-osi od leve proti desni, lahko do sprememb v osvetljenosti tal pride le pri tistih x-koordinatah, kjer se začne ali konča kakšen izmed teh intervalov. Tako lahko x-os razdelimo na območja s konstantno osvetljenostjo; pri primeru iz besedila naloge (in na tamkajšnji sliki) bi na primer dobili interval $(-\infty, 1)$ z osvetljenostjo 1, nato interval $[-1, 2)$ z osvetljenostjo 8, nato $[2, 3]$ z osvetljetnostjo 11, nato $(3, 6)$ z osvetljenostjo 3, nato točko $x = 6$ z osvetljenostjo 8, nato interval $(6, 12]$ z osvetljenostjo 5 in končno interval $(12, \infty)$ z osvetljenostjo 0.

Za učinkovito odgovarjanje na poizvedbe je koristno, če si pripravimo tak seznam intervalov, urejen po x-koordinati, potem pa lahko pri vsaki poizvedbi z bisekcijo hitro poiščemo interval, na katerem leži točka, po kateri sprašuje ta poizvedba. V ta namen najprej sestavimo seznam parov ⟨x-koordinata, sprememba intenzitete⟩ (vsak žaromet prispeva dva taka para, enega za začetek in enega za konec svojega intervala) in jih uredimo po x-koordinati. Nato gremo po tem seznamu (po naraščajočih x-koordinatah), sproti seštevamo spremembe v intenziteti in vsakič, ko se x-koordinata res spremeni, začnemo nov interval.

V točkah, kjer se stikata dva ali več stožcev, je potrebno nekaj pazljivosti. Lahko si na primer pri vsaki taki točki poleg intenzitete intervala, ki se začne v tej točki, zapomnimo tudi skupno intenziteto stožcev, ki se končajo v tej točki. Če potem pride poizvedba, ki pade prav na to točko (in ne na interval med njo in naslednjo točko), potem vemo, da jo osvetljujejo tudi tisti stožci, ki se v tej točki končajo, in moramo rezultatu prišteti tudi njihovo intenziteto.

V spodnji rešitvi so intervali opisani kot zaporedje struktur tipa `Tocka`; vsaka taka struktura `t` pove, da se na x-koordinati `t.x` začenja interval z osvetljenostjo `t.c` (ki se razteza do naslednje točke v zaporedju), točko `t.x` sámo pa dodatno osvetljujejo še stožci, ki se končajo v njej in ki imajo skupno intenziteto `t.k`.

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

int main()
{
  /* Preberimo vhodne podatke. */
  FILE *f = fopen("zarometi.in", "rt"), *g = fopen("zarometi.out", "wt");
```

```
int n; fscanf(f, "%d", &n);
vector<pair<int, int>> v; v.reserve(2 * n);
for (int i = 0; i < n; i++)
{
  int x, y, c; fscanf(f, "%d %d %d", &x, &y, &c);
  /* Za vsak žaromet dodajmo v seznam obe krajišči;
     pri začetnem pomnožimo intenziteto z −1. */
  v.push_back({x − y, −c}); v.push_back({x + y, c});
}
/* Uredimo krajišča po x. */
sort(v.begin(), v.end());
/* Združimo krajišča z istim x. Pri vsakem x si tudi zapomnimo
   novo intenziteto (ki velja desno od tega x) in še to, kakšna
   je skupna intenziteta žarometov, ki so se pri tistem x končali. */
struct Tocka { int x; long long c, k; };
vector<Tocka> tocke;
long long intenziteta = 0;
for (int i = 0; i < v.size(); )
{
  Tocka t; t.x = v[i].first; t.k = 0;
  for ( ; i < v.size() && v[i].first == t.x; i++)
  {
    int c = v[i].second; intenziteta −= c;
    if (c > 0) t.k += c;
  }
  t.c = intenziteta; tocke.push_back(t);
}
/* Odgovarjajmo na poizvedbe. */
int q; fscanf(f, "%d", &q);
long long rezultat = 0;
while (q−−)
{
  long long p; fscanf(f, "%lld", &p); p += rezultat;
  if (p < tocke[0].x)
    rezultat = 0; /* p je levo od vseh stožcev. */
  else
  {
    /* Z bisekcijo poiščimo interval, ki vsebuje točko p. */
    int L = 0, D = (int) tocke.size();
    while (D − L > 1)
    {
      /* Na tem mestu velja: tocke[L].x ≤ p < tocke[D].x
         (če je D = tocke.size(), si mislimo tam x = ∞). */
      int M = (L + D) / 2;
      if (tocke[M].x <= p) L = M; else D = M;
    }
    /* p torej leži na tocke[L].x ≤ p < tocke[L + 1].x. */
    rezultat = tocke[L].c;
    /* Če leži prav na točki L, prištejmo še stožce, ki se v tej točki končajo. */
    if (tocke[L].x == p) rezultat += tocke[L].k;
  }
  fprintf(g, "%lld\n", rezultat); /* Izpišimo rezultat. */
}
fclose(f); fclose(g);
return 0;
```

}

Še en zelo eleganten način, kako se izogniti zapletom v točkah, kjer se stika več stožcev, pa je ta, da vsak stožec v mislih malo razširimo. Naloga pravi, da žaromet i prispeva svojo osvetljenost c_i na intervalu $[x_i - y_i, x_i + y_i]$; namesto tega pa zdaj razširimo ta interval na $[x_i - y_i - \varepsilon, x_i + y_i + \varepsilon]$ za nek $\varepsilon > 0$, ki naj bo dovolj majhen, da je razdalja med krajiščema dveh intervalov (pred razširitvijo), pa tudi med krajiščem intervala in poizvedbo, gotovo v vsakem primeru večja od ε, razen kadar je enaka 0. Ta pogoj nam zagotavlja, da zaradi razširitve intervalov ne bo prišlo do napak v odgovorih na poizvedbe, obenem pa zagotavlja, da nobena poizvedba ne bo več padla ravno na krajišče kakšnega intervala (in s tem na stičišče več stožcev).

V našem primeru so vse koordinate celoštevilske, zato je načeloma dober vsak ε, manjši od 1. Lahko bi vzeli na primer $\varepsilon = 1/2$; da pa nam ne bo treba delati z necelimi števili, je nato koristno pomnožiti vse koordinate z 2. Žaromet (x_i, y_i) torej osvetli interval $[2(x_i - y_i) - 1, 2(x_i + y_i) + 1]$, položaj j-tega nastopajočega pa, če je bil v prvotni različici rešitve enak q_j, bo zdaj $2q_j$. Tako so zdaj vsa krajišča intervalov liha, poizvedbe pa sode, tako da se ne more več zgoditi, da bi poizvedba padla ravno na krajišče intervala (in s tem na točko, kjer se stikajo stožci več žarometov).

4. Najkrajša pot

V nalogi se skriva neusmerjen graf, v katerem mesta predstavljajo točke, ceste pa povezave. Pravzaprav je to multigraf, ker je lahko med dvema točkama tudi več neposrednih povezav. Začetno točko označimo s s, končno pa s t; pri naši nalogi je torej $s = 1$ in $t = n$. Označimo z $d(u, v)$ dolžino najkrajše poti od u do v, s $\#(u, v)$ pa število poti te dolžine od u do v. Ker so ceste v naši nalogi dvosmerne, velja $d(u, v) = d(v, u)$ in $\#(u, v) = \#(v, u)$.

Če začnemo v točki s, lahko z iskanjem v širino izračunamo $d(s, u)$ in $\#(s, u)$ za vse točke u. Postopek iskanja v širino smo videli že v rešitvi prve naloge in bo tukaj zelo podoben, dopolniti ga moramo le še z računanjem števila poti, $\#(s, u)$. Ko iz vrste vzamemo točko u in pregledujemo povezave (ceste), po katerih je mogoče u zapustiti, lahko razmišljamo takole: če vidimo, da obstaja c povezav od u do v, to pomeni, da bi se dalo vsako od $\#(s, u)$ poti dolžine $d(s, u)$ od s do u podaljšati na c načinov v pot dolžine $d(s, u) + 1$ od s do v. Če do v že poznamo kakšno krajšo pot, si s tem ne moremo pomagati, sicer pa lahko zdaj $\#(s, v)$ povečamo za število teh novih poti, torej za $c \cdot \#(s, u)$.

(Če je kakšna točka u nedosegljiva iz s, si lahko predstavljamo $d(s, u) = \infty$ in $\#(s, u) = 0$. V spodnjem programu bomo namesto vrednosti ∞ uporabljali -1.)

Z enakim postopkom lahko izračunamo tudi $d(t, u)$ in $\#(t, u)$ za vse u. (V spodnji rešitvi imamo v ta namen podprogram NajkrajsePoti.)

Naloga zahteva, da dodamo v graf eno povezavo tako, da bo potem število najkrajših poti od s do t čim večje. Recimo, da nova povezava neposredno poveže točki x in y; za namen tega razmisleka se pretvarjajmo, da je povezava usmerjena $x \to y$, o obrnjeni povezavi $y \to x$ pa bomo razmišljali posebej. Najkrajša pot, ki jo je mogoče speljati po novi povezavi $x \to y$, gre torej najprej od s do x, nato po $x \to y$ in nato od y do t, tako da je skupaj dolga $d(s, x) + 1 + d(y, t)$. Pravzaprav taka pot ni ena sama, saj si lahko del poti od s do x izberemo na $\#(s, x)$ načinov, od y do t pa na $\#(y, t)$ načinov, tako da imamo zdaj kar $\#(s, x) \cdot \#(y, t)$ novih poti.

Glede na dolžino teh novih poti ločimo tri primere:

(1) Če je $d(s,x)+1+d(y,t) > d(s,t)$, te nove poti ne bodo najkrajše in se torej število najkrajših poti od s do t ne bo nič spremenilo; taka povezava $x \to y$ za nas ni zanimiva.

(2) Druga možnost je, da je nova pot enako dolga kot najkrajša pot doslej, torej $d(s,x)+1+d(y,t) = d(s,t)$. V tem primeru bo po novem obstajalo $\#(s,t)+\#(s,x)\cdot\#(y,t)$ najkrajših poti od s do t.

(3) Ostane še možnost, da je nova pot krajša od najkrajše doslej, torej $d(s,x)+1+d(y,t) < d(s,t)$. V tem primeru bo po novem pač obstajalo $\#(s,x)\cdot\#(y,t)$ najkrajših poti od s do t.

Načeloma bi lahko pregledali vse možne pare (x,y) in uporabili tistega, pri katerem dobimo po novem največje število najkrajših poti. Težava je, da je točk veliko (do milijon) in si ne moremo privoščiti, da bi se posebej ukvarjali z vsemi možnimi pari točk.

Do učinkovitejše rešitve pridemo z naslednjim razmislekom. Izberimo si le x, ne pa tudi točke y. (Pri tem se omejimo na take x, ki so dosegljivi iz s. Če x sploh ni dosegljiv iz s, potem se tudi poti od s do t pač ne bo dalo speljati prek x.) Tisti y, pri katerih bi zgoraj padli v primer (1), nas tako ali tako ne zanimajo. V primer (2) pademo pri tistih y, za katere velja $d(y,t) = d(s,t) - 1 - d(s,x)$. Med njimi bo največ poti nastalo pri tistem, ki ima med vsemi takimi y največjo vrednost $\#(y,t)$. Koristno bi torej bilo, če bi si za vsako možno oddaljenost δ od t zapomnili maksimum vrednosti $\#(y,t)$ po vseh tistih y, ki imajo $d(y,t) = \delta$. (Spodnji program izračuna in hrani te vrednosti v najSt[δ].) Tako bomo lahko izračunali maksimalno število novih najkrajših poti po vseh takih y, pri katerih pademo v primer (2), ne da bi se nam bilo treba posebej ukvarjati z vsakim od teh y.

Podobno je tudi v primeru (3); da obdelamo tega, si je koristno za vsako δ zapomniti maksimum vrednosti $\#(y,t)$ po vseh tistih y, ki imajo $d(y,t) \le \delta$. (Spodnji program izračuna in hrani te vrednosti v najStDo[δ].)

Paziti moramo še na možnost, da v prvotnem grafu točka t sploh ni dosegljiva iz s. To pomeni, da do primera (2) sploh ne more priti, do primera (3) pa pride pri vsakem y, iz katerega je dosegljiv t. Ker ima graf n točk, je t dosegljiv iz v v največ $n-1$ korakih (ali pa je nedosegljiv), torej lahko v tem primeru vzamemo $\delta = n-1$.

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

int n;
vector<int> ps, ns; /* prvi sosed, število sosedov */
struct Sosed { int u, c; };
vector<Sosed> sosedi;

/* Za vsako u vrne v d[u] dolžino najkrajše poti od točke „od“ do u,
   v st[u] pa število takih najkrajših poti. Če u ni dosegljiva iz „od“,
   dobimo d[u] = −1 in st[u] = 0. */
void NajkrajsePoti(int od, vector<int> &d, vector<long long> &st)
{
  d.resize(n); st.resize(n);
  for (int u = 0; u < n; u++) d[u] = −1, st[u] = 0;
  vector<int> vrsta; vrsta.push_back(od); d[od] = 0; st[od] = 1;
```

```
  int glava = 0;
  while (glava < vrsta.size())
  {
    int u = vrsta[glava++];
    for (int i = ps[u]; i < ps[u] + ns[u]; i++)
    {
      int v = sosedi[i].u, c = sosedi[i].c;
      if (d[v] < 0) { d[v] = d[u] + 1; st[v] = st[u] * c; vrsta.push_back(v); }
      else if (d[v] == d[u] + 1) st[v] += st[u] * c;
    }
  }
}

int main()
{
  FILE *f = fopen("pot.in", "rt");
  int m; fscanf(f, "%d %d", &n, &m);
  ps.resize(n); ns.resize(n);
  for (int u = 0; u < n; u++) ns[u] = 0;

  /* Preberimo seznam cest. */
  struct Cesta { int u, v, c; };
  vector<Cesta> ceste; ceste.resize(m);
  for (int i = 0; i < m; i++) {
    Cesta &c = ceste[i]; fscanf(f, "%d %d %d", &c.u, &c.v, &c.c);
    ns[--c.u]++; ns[--c.v]++; }
  fclose(f);

  /* Predelajmo ga v sezname sosedov. */
  for (int u = 0, p = 0; u < n; u++) { ps[u] = p; p += ns[u]; ns[u] = 0; }
  sosedi.resize(2 * m);
  for (Cesta c : ceste) {
    sosedi[ps[c.u] + ns[c.u]++] = { c.v, c.c };
    sosedi[ps[c.v] + ns[c.v]++] = { c.u, c.c }; }

  /* Izračunajmo najkrajše poti od s in t do vseh ostalih mest. */
  const int s = 0, t = n - 1;
  vector<int> ds, dt; vector<long long> sts, stt;
  NajkrajsePoti(s, ds, sts); NajkrajsePoti(t, dt, stt);
  int D = ds[t];

  /* Za vsako oddaljenost d poglejmo, katera izmed točk, ki so na
     oddaljenosti d od t, ima največ poti te dolžine do t.
     To shranimo v najSt[d]. */
  vector<long long> najSt, najStDo;
  najSt.resize(n); for (int d = 0; d < n; d++) najSt[d] = 0;
  for (int u = 0; u < n; u++) {
    int d = dt[u]; if (d < 0) continue;
    if (stt[u] > najSt[d]) najSt[d] = stt[u]; }

  /* V najStDo[d] pa naj bo maksimum vrednosti najSt[0..d]. */
  najStDo.resize(n);
  for (int d = 0; d < n; d++)
    najStDo[d] = max(najSt[d], d > 0 ? najStDo[d - 1] : 0);

  /* Poiščimo najboljšo rešitev. */
  long long naj = 0;
  for (int u = 0; u < n; u++)
  {
    /* Recimo, da bi želeli dodati neko povezavo (u, v) tako,
```

```
        da bi šla nova najkrajša pot od s do t po njej (in najprej skozi u).
        Če u sploh ni dosegljiva iz s, take poti sploh ne bo. */
    int du = ds[u]; if (du < 0) continue;
    /* Poglejmo, kako daleč sme biti v od t-ja, da ne bo takšna
       pot daljša od najkrajše dosedanje poti od s do t. */
    int dv = (D < 0) ? n - 1 : D - 1 - du; if (dv < 0) continue;
    /* Ena možnost je, da vzamemo tako v, da bo nova pot enako dolga
       kot najkrajša doslej. */
    naj = max(naj, sts[u] * najSt[dv] + sts[t]);
    /* Druga možnost je, da bo nova pot krajša. Namesto najStDo[dv - 1]
       lahko vzamemo kar najStDo[dv], kajti če se tidve vrednosti razlikujeta,
       je to zato, ker je najStDo[dv] = najSt[dv], tedaj pa spodnja rešitev
       tako ali tako ne bo boljša od tiste iz prejšnje vrstice, kjer smo
       prišteli tudi sts[t]. */
    naj = max(naj, sts[u] * najStDo[dv]);
  }
  /* Izpišimo rezultat. */
  f = fopen("pot.out", "wt");
  fprintf(f, "%lld\n", naj);
  fclose(f); return 0;
}
```

Razmislimo zdaj še o težji različici naloge, ki jo omenja opomba pod črto na str. 28. Pri tej različici lahko med posameznim parom mest obstaja največ ena neposredna povezava, pa tudi novo cesto $x \to y$ smemo dodati le med takim parom mest, ki doslej še nista bili neposredno povezani. Tega novega pogoja ne bi bilo težko preverjati, če bi naša dosedanja rešitev delovala tako, da bi šla v zanki po vseh možnih parih (x, y); ker pa tega ne počne (saj bi bilo prepočasi), moramo biti malo previdnejši.

Recimo, da smo si (tako kot pri rešitvi prvotne naloge) izbrali nek x in nas zdaj zanima, kako bi se dalo z dodajanjem ceste $x \to y$ (za nek primerno izbran y) čim bolj povečati število najkrajših poti od s do t. Enako kot pri rešitvi prvotne naloge vidimo, da nova cesta $x \to y$ prinese od s do t nove poti dolžine $d(s, x) + 1 + d(y, t)$ in da je takih novih poti $\#(s, x) \cdot \#(y, t)$; in enako kot pri prvotni rešitvi so takšne nove poti zanimive le, če je njihova dolžina $\le d(s, t)$.

Oglejmo si najprej možnost, da je nova pot enako dolga kot najkrajša doslej, torej da je $d(s, x) + 1 + d(y, t) = d(s, t)$. To pomeni, da mora biti $d(y, t) = \delta$ za $\delta = d(s, t) - d(s, x) - 1$. Razlika v primerjavi s prvotno nalogo je, da zdaj za nas ni več dober kar vsak tak y, saj imamo še dodatno omejitev: v poštev pridejo le tisti y, za katere cesta $x \to y$ doslej (v vhodnem grafu) še ni obstajala. Med tistimi y, ki ustrezajo tudi tej dodatni omejitvi, pa nas bo potem, enako kot pri rešitvi prvotne naloge, zanimal tisti z največjo vrednostjo $\#(y, t)$.

Vidimo torej, da ni dovolj le, če si za vsako δ zapomnimo maksimum $\#(y, t)$ po vseh tistih y, ki imajo $d(y, t) = \delta$, saj bomo včasih (odvisno od x) morali nekatere od teh y-ov ignorirati. Koristno bi bilo torej pri vsaki δ imeti seznam vseh y, ki imajo $d(y, t) = \delta$; in ta seznam naj bo urejen padajoče po $\#(y, t)$. Temu seznamu recimo $L[\delta]$. Ko potem pri nekem konkretnem x razmišljamo, v kateri y bi speljali novo cesto $x \to y$, moramo iti po tem seznamu in za mesta y z njega po vrsti preverjati, ali povezava $x \to y$ že obstaja (v vhodnem grafu) ali ne; čim najdemo tak y, pri

katerem povezave $x \to y$ še ni, se lahko ustavimo. (Lahko se seveda tudi izkaže, da takega y sploh ni, kar pač pomeni, da za naš trenutni x ni mogoče dodati nove povezave $x \to y$ tako, da bi lahko čeznjo speljali od s do t pot, ki bi bila enako dolga kot dosedanja $d(s,t)$.)

Na podoben način lahko obravnavamo tudi možnost, da je nova pot krajša od najkrajše doslej, torej da je $d(s,x) + 1 + d(y,t) < d(s,t)$. To pomeni, da mora biti $d(y,t) < \delta$ za $\delta = d(s,t) - d(s,x) - 1$. Načeloma bi bilo zdaj koristno imeti seznam $L'[\delta]$ vseh takih y, za katere je $d(y,t) < \delta$, ta seznam bi moral biti urejen padajoče po $\#(y,t)$ in mi bi se sprehajali po njem, dokler ne bi našli prvega takega y, za katerega povezave $x \to y$ v prvotnem grafu ni. Težava te rešitve je, da se lahko isti y pojavlja na več seznamih — če je pojavi na $L'[\delta]$, se bo gotovo tudi na $L'[\delta+1]$, $L'[\delta+2]$ in tako naprej — zato so lahko vsi seznami skupaj dolgi $O(n^2)$, kar je že preveč. Domisliti se moramo česa boljšega.

Doslej se nismo kaj dosti ukvarjali s tem, v kakšnem vrstnem redu pregledujemo različne x, vemo le to, da moramo prej ali slej pregledati vse (razen tistih, ki sploh niso dosegljivi iz s, torej ki imajo $d(s,x) = \infty$ in $\#(s,x) = 0$). Recimo pa, da jih pregledujemo po padajoči vrednosti $d(s,x)$. Ko se z zunanjo zanko premikamo od enega x do naslednjega, se $d(s,x)$ počasi zmanjšuje, zato pa se δ počasi povečuje. Pogoju $d(y,t) < \delta$ zato počasi ustreza vse več mest y. Namesto da hkrati hranimo sezname $L'[\delta]$ za vse možne δ, bo zdaj dovolj že en sam seznam L', na katerega bomo vsakič, ko se δ poveča, dodali še tiste y, ki po novem ustrezajo pogoju $d(y,t) < \delta$, prej (pred zadnjim povečanjem vrednosti δ) pa mu še niso ustrezali. Še vedno pa mora biti ta seznam urejen padajoče po $\#(y,t)$. Da bo delo s takim L' dovolj poceni, bomo namesto seznama v resnici raje uporabili kakšno primerno uravnoteženo drevesasto strukturo, na primer B-drevo.

Zapišimo tako dobljeno rešitev s psevdokodo:

```
1   za vse u izračunaj d(s,u), d(u,t), #(s,u) in #(u,t),
      tako kot pri rešitvi prvotne naloge;
2   za vsako δ od 0 do n − 1: naj bo L[δ] prazen seznam;
3   za vsako mesto y:
      δ := d(y,t); if δ < ∞ then dodaj y v seznam L[δ];
4   za vsako δ od 0 do n − 1:
      uredi mesta y v seznamu L[δ] padajoče po #(y,t);
5   naj bo L' prazno drevo; naj := −∞; δ' := 0;
6   za vsako mesto x, ki ima d(s,x) < ∞, v padajočem vrstnem redu po d(s,x):
      δ := d(s,t) − d(s,x) − 1;
7     za vsak y s seznama L[δ]:
        if že obstaja povezava x → y then continue;
        naj := max{naj, #(s,t) + #(s,x) · #(x,y)}; break;
8     while δ' < δ:
        za vsak y s seznama L[δ']: dodaj y v L';
        δ' := δ' + 1;
9     za y iz L', v naraščajočem vrstnem redu po #(y,t):
        if že obstaja povezava x → y then continue;
        naj := max{naj, #(s,x) · #(x,y)}; break;
    return naj;
```

Kakšna je časovna zahtevnost tega postopka? Recimo, da ima vhodni graf n točk in m povezav. Za korak 1 potrebujemo iskanje v širino, enako kot pri rešitvi prvotne naloge, kar nam vzame $O(n+m)$ časa. Priprava seznamov v korakih 2 in 3 nam vzame $O(n)$ časa, urejanje v koraku 4 pa $O(n \log n)$ časa (ker imajo vsi seznami skupaj največ n elementov). Preden začnemo glavno zanko v koraku 6, moramo mesta x urediti po $d(s,x)$, kar lahko naredimo celo v $O(n)$ časa, ker so vrednosti d vse na območju od 0 do $n-1$ (urejanje s štetjem).

Skupno število iteracij zanke 7 (skupno po vseh iteracijah zunanje zanke 6) lahko opišemo takole: za vsako povezavo $x \to y$, ki v vhodnem grafu že obstaja, se lahko izvede največ ena taka iteracija zanke 7, ki se konča s stavkom **continue**; in za vsako mesto x se izvede največ ena taka iteracija, ki se konča s stavkom **break**. Skupno število iteracij te zanke je torej $O(n+m)$, vsaka iteracija pa vzame le $O(1)$ časa, če predpostavimo, da imamo povezave prvotnega grafa v razpršeni tabeli, tako da lahko v $O(1)$ časa preverimo, ali povezava $x \to y$ že obstaja.

Naloga zanke 8 je, da v drevo L' po potrebi dodaja točke tako, da bo to drevo vsebovalo natanko tiste y, za katere je $d(y,t) < \delta$. Pri tem se mora (po vseh iteracijah zunanje zanke 6) δ' počasi povečati od 0 do največ $n-1$ in vsak y bo treba največ enkrat dodati v L'; vsako dodajanje v drevo pa vzame $O(\log n)$ časa. Tako porabimo za zanko 8 $O(n \log n)$ časa.

Pri zanki 9 nam enak razmislek kot pri zanki 7 pokaže, da je skupno število iteracij te zanke $O(n+m)$. Pri ceni posamezne iteracije je vprašanje, koliko časa porabimo, da se v drevesu L' premaknemo od trenutnega y do naslednjega; koristno je na primer, če za L' uporabimo B-drevo in imamo liste drevesa povezane v verigo (*doubly-linked list*): tedaj se na naslednji y zlahka premaknemo v $O(1)$ časa.

Časovna zahtevnost celotne rešitve je tako $O(n \log n + m)$; to, kateri od obeh členov v tej vsoti prevladuje, pa je odvisno od tega, kako gost graf smo dobili (v najslabšem primeru gre lahko m do $O(n^2)$).

5. Listi

Uredimo liste padajoče po širini; če je kje več enako širokih, jih uredimo padajoče po višini. Sprehodimo se po tem seznamu; pri vsakem listu poglejmo, če je višji od prejšnjega v seznamu; če ni, ga lahko pobrišemo, kajti prejšnji list je gotovo vsaj tako širok kot trenutni in če je hkrati tudi vsaj tako visok kot trenutni, to pomeni, da lahko trenutni list vedno damo na isti kup kot prejšnjega in se ta kup zaradi njega ne bo nič povečal.

Po tem pregledu nam ostane seznam, v katerem so listi urejeni padajoče po širini, hkrati pa naraščajoče po višini. V tem vrstnem redu jih oštevilčimo od 1 do m (pri tem je m število listov, ki so nam ostali od prvotnih n po morebitnem brisanju nekaterih listov v prejšnjem odstavku). Recimo zdaj, da bi lista i in j dali na isti kup; naj bo $j < i$. Če je med njima na seznamu še nek list k, to pomeni, da velja $j < k < i$; ker so listi urejeni padajoče po širini, je list j vsaj tako širok kot k, in ker so urejeni naraščajoče po višini, je i vsaj tako visok kot k. Če torej lista i in j damo na isti kup, bo ta kup dovolj velik tudi za list k, ne da bi se moral pri tem še kaj povečati. Ta razmislek nam pove, da se lahko omejimo na rešitve, ki tvorijo kupe le iz strnjenih skupin listov (v našem prej opisanem vrstnem redu).

Zdaj lahko nalogo rešujemo z dinamičnim programiranjem. Definirajmo naslednje podprobleme: recimo, da bi radi na kupe čim bolje razdelili le prvih i listov (namesto vseh listov). Skupno površino teh kupov označimo s $f(i)$. Videli smo, da se lahko omejimo na razporede, kjer vsak kup obsega neko strnjeno skupino listov; zadnji kup pri našem podproblemu bo recimo obsegal liste od j do i za nek $j \leq i$. Zadnji kup bo torej velik $w_j \cdot h_i$, pred tem pa imamo neko razbitje prvih $j-1$ listov na kupe, najboljše tako razbitje pa ima skupno površino $f(j-1)$. Tako torej vidimo:

$$f(i) = \min\{w_j \cdot h_i + f(j-1) \ : \ 1 \leq j \leq i\}.$$

Robni primer je $f(0) = 0$ (ko ni nobenega lista več, tudi ni treba tvoriti kupov).

Vidimo lahko, da pri izračunu $f(i)$ potrebujemo vrednosti $f(0), \ldots, f(i-1)$, zato je koristno reševati te podprobleme v zanki po naraščajočih i in si rešitve sproti shranjevati v tabelo.

Naloga sprašuje še po najmanjšem številu kupov, s katerimi je mogoče doseči minimalno skupno površino. Označimo to število kupov s $k(i)$. Ko vidimo, pri katerem j je dosežen minimum v formuli za $f(i)$, lahko zaključimo, da je $k(i) = k(j-1) + 1$; če je mogoče enako dober minimalni $f(i)$ doseči pri več različnih j, moramo vzeti med njimi tistega z najmanjšim $k(j-1)$, da bo potem tudi $k(i)$ najmanjši možni.

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

int main()
{
  /* Preberimo vhodne podatke. */
  FILE *f = fopen("listi.in", "rt");
  int n; fscanf(f, "%d", &n);
  struct List { int w, h; };
  vector<List> listi; listi.resize(n);
  for (int i = 0; i < n; i++) fscanf(f, "%d %d", listi[i].w, listi[i].h);
  fclose(f);

  /* Uredimo seznam in pobrišimo liste, ki so po obeh dimenzijah
     manjši ali enaki kakšnemu drugemu. */
  sort(listi.begin(), listi.end(), [] (const List& a, const List &b) {
    if (a.w != b.w) return a.w > b.w; else return a.h > b.h; });
  int m = 0;
  for (int i = 0; i < n; i++)
  {
    List L = listi[i];
    if (m > 0 && listi[m - 1].h >= L.h) continue;
    listi[m++] = L;
  }
  listi.resize(m); n = m;

  /* Rešimo podprobleme od manjših proti večjim. */
  vector<long long> povrsina; vector<int> stKupov;
  povrsina.resize(m + 1); stKupov.resize(m + 1);
  povrsina[0] = 0; stKupov[0] = 0; /* Rešitev za 0 listov. */
  for (int i = 0; i < n; i++)
  {
```

```
    /* Poiščimo najboljšo rešitev za prvih i + 1 listov (listi 0..i). */
    for (int j = 0; j <= i; j++)
    {
      /* Kako bi bilo, če bi sestavili en kup z listi j..i? */
      long long kand = povrsina[j] + (long long) listi[j].w * listi[i].h;
      if (j > 0 && (kand > povrsina[i + 1] ||
                    kand == povrsina[i + 1] && stKupov[j] + 1 >= stKupov[i + 1]))
        continue;
      povrsina[i + 1] = kand; stKupov[i + 1] = stKupov[j] + 1;
    }
  }
  /* Izpišimo rezultat. */
  f = fopen("listi.out", "wt");
  fprintf(f, "%lld %d\n", povrsina[n], stKupov[n]);
  fclose(f); return 0;
}
```

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Umor

Najprej preberimo ime umorjenca in si ga zapomnimo v neki spremenljivki (v spodnji rešitvi je to na primer zrtev). Nato po vrsti berimo vrstice s podatki o pogovorih (v spremenljivki n hranimo število vrstic, ki jih še moramo prebrati — ko ta števec pade na 0, se ustavimo). Pri vsakem pogovoru pogledamo, če je eden od obeh sogovornikov kasnejši umorjenec; če je, je drugi sogovornik zadnji človek doslej, ki je umorjenca še videl živega, zato si tega sogovornika zapomnimo (v spremenljivki zadnji). Na koncu spremenljivko zadnji izpišemo. Paziti moramo še na možnost, da z umorjencem sploh ni nihče govoril; to lahko odkrijemo tako, da spremenljivko zadnji za začetku inicializiramo na nek znak, ki ni črka, in preverimo, če ima na koncu še vedno to vrednost.

```
#include <stdio.h>

int main()
{
  char zrtev, zadnji = '.', a, b;
  int n;

  scanf("%c\n", &zrtev);
  scanf("%d\n", &n);

  while (n-- > 0)
  {
    scanf("%c %c\n", &a, &b);
    if (a == zrtev) zadnji = b;
    else if (b == zrtev) zadnji = a;
  }

  if (zadnji == '.') printf("Z umorjencem ni govoril nihče.\n");
  else printf("Z umorjencem je zadnji govoril %c.\n", zadnji);
  return 0;
}
```

2. Naraščajoče besede

Vhodno besedilo lahko beremo znak po znak; poleg trenutnega znaka (v spodnji rešitvi je to spremenljivka c) si zapomnimo še prejšnjega (spremenljivka cp), tako da bomo lahko po vsakem znaku preverili, ali je trenutna beseda še urejena naraščajoče. O trenutni besedi si moramo poleg tega, ali je naraščajoča, zapomniti še njeno dolžino. Če je trenutni znak črka, povečamo dolžino in preverimo, če beseda narašča; če pa trenutni znak ni črka, pomeni, da smo na koncu besede in moramo pogledati, če je to najdaljša naraščajoča beseda doslej; če je, si njeno dolžino zapomnimo v spremenljivki naj. Zapišimo to rešitev še v C-ju:

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

int main()
{
  int naj = 0, dolzina = 0, cp = -1;
  bool narasca = true;
```

```
  do
  {
    int c = fgetc(stdin);
    if ('a' <= c && c <= 'z')
    {
      /* Smo znotraj neke besede. Povečajmo dolžino in preverimo,
         če je še vedno urejena naraščajoče. */
      dolzina++;
      if (c < cp) narasca = false;
    }
    else
    {
      /* Smo med dvema besedama. Če je bila prejšnja beseda naraščajoča
         in najdaljša taka doslej, si jo zapomnimo. */
      if (narasca && dolzina > naj) naj = dolzina;
      /* Pripravimo se na branje naslednje besede. */
      dolzina = 0; narasca = true;
    }
    cp = c; /* Zapomnimo si trenutno črko kot prejšnjo v spremenljivki cp. */
  }
  while (cp != EOF);
  printf("%d\n", naj); return 0;
}
```

3. Popularni dnevi

Recimo, da imamo naše dneve oštevilčene od 0 do $n-1$. Ko bi za nek dan d radi preverili, ali je popularen ali ne, moramo najprej ugotoviti, katerih je k njemu najbližjih dni (brez d-ja samega). Lahko si predstavljamo, da izmenično jemljemo dneve z leve in desne, dokler se nam jih ne nabere k: tako dobimo $d+1$, $d-1$, $d+2$, $d-2$, ... Načeloma bomo tako pobrali z vsake strani d-ja po $k/2$ dni, mogoče pa je tudi, da nam z ene strani zmanjka dni (na primer: če je $d < k/2$, potem pred njim v zaporedju ni $k/2$ dni), takrat pa jih bomo morali pač ustrezno več pobrati z druge strani. Spotoma lahko obisk v teh dneh tudi seštevamo in na koncu primerjamo povprečje po teh k dneh z obiskom na dan d. Če je obisk na dan d nad povprečjem, povečamo števec popularnih dni. Zapišimo ta postopek v C-ju:

```
int Popularni(int n, int k, const int obisk[])
{
  int stPopularnih = 0;
  for (int d = 0; d < n; d++)
  {
    /* Seštejmo obisk v najbližjih k dnevih. Spremenljivka s pove, koliko
       smo jih že sešteli, i pa pove, kako daleč od d-ja smo že. */
    int vsota = 0;
    for (int s = 0, i = 0; s < k; i++)
    {
      if (d - i >= 0) vsota += obisk[d - i], s++;
      if (d + i < n) vsota += obisk[d + i], s++;
    }
    if (vsota < obisk[d] * k) stPopularnih++;
  }
  return stPopularnih;
}
```

Časovna zahtevnost te rešitve je $O(n \cdot k)$, saj moramo pri vsakem od n dni iti z zanko po najbližjih k dnevih in jih seštevati. Razmislimo o tem, kako lahko rešitev še izboljšamo. Ko računamo najbližjih k dni dnevu d, načeloma nastane interval od $d - k/2$ do $d + k/2$, razen če je d blizu začetka zaporedja (pri $d < k/2$) — takrat nastane interval od 0 do k — ali pa blizu konca zaporedja (pri $d > n - 1 - k/2$ — takrat nastane interval od $n - k - 1$ do $n - 1$. (Pravzaprav ti intervali, ki smo jih pravkar zapisali, obsegajo tudi dan d sam, na kar bomo morali kasneje paziti pri izračunu povprečja.) Če se z d-jem premaknemo za en dan naprej, se tudi vsako od krajišč intervala premakne za en dan naprej ali pa ostane nespremenjeno. Iz vsote obiska po vseh dneh v intervalu zato na levi strani lahko izpade en člen (če se levo krajišče premakne naprej), na desni strani pa lahko vsota en člen pridobi (če se premakne naprej desno krajišče). Zato po vsakem premiku d-ja ni težko popraviti dosedanje vsote, namesto da bi jo računali vsakič znova od začetka. Tako dobimo naslednjo rešitev:

```
int Popularni(int n, int k, const int obisk[])
{
  int vsota = 0, vsotaOd = 0, vsotaDo = 0, stPopularnih = 0;
  for (int d = 0; d < n; d++)
  {
    /* Naj bo novaOd..novaDo − 1 interval, ki pokriva dan d in še
       njemu najbližjih k drugih dni. */
    int novaOd = d − k / 2, novaDo = d + k / 2 + 1;
    if (novaOd < 0) novaOd = 0, novaDo = novaOd + k + 1;
    else if (novaDo > n) novaDo = n, novaOd = novaDo − k − 1;

    /* Trenutno je v spremenljivki „vsota“ vsota obiskov za dneve
       vsotaOd..vsotaDo − 1. Popravimo jo na vsoto dni novaOd..novaDo − 1. */
    while (vsotaOd < novaOd) vsota −= obisk[vsotaOd++];
    while (vsotaDo < novaDo) vsota += obisk[vsotaDo++];

    /* Preverimo, če ima trenutni dan nadpovprečni obisk. */
    if (vsota − obisk[d] < obisk[d] * k) stPopularnih++;
  }
  return stPopularnih;
}
```

Časovna zahtevnost te rešitve je le še $O(n)$, saj vsak člen zaporedja enkrat vstopi v vsoto in enkrat pade iz nje, tako da imamo s popravljanjem vsote vsega skupaj le $O(n)$ dela.

4. Sindikat

Plače je koristno popravljati po drevesu od spodaj navzgor. Na primer, če nek uslužbenec nima podrejenih, njegove plače ni treba spreminjati: znižati mu je ne smemo (tako pravi besedilo naloge), zvišati pa mu je ni treba, saj nam to ne bi nič pomagalo pri doseganju pogoja, da mora imeti vsak nadrejeni višjo plačo od svojih podrejenih.

Višje gor v drevesu pa lahko razmišljamo takole. Recimo, da opazujemo nekega uslužbenca u in smo že v vseh njegovih poddrevesih popravili plače tako, da ustrezajo zahtevam naloge. Zdaj lahko pogledamo plače u-jevih neposredno podrejenih uslužbencev in po potrebi ustrezno zvišamo u-jevo plačo. Na veljavnost pogojev v u-jevih poddrevesih to nič ne vpliva, tako da zdaj pogoj velja za u in vse njegove

posredno ali neposredno podrejene. Ta postopek ponavljamo navzgor po drevesu, dokler ne obdelamo vseh uslužbencev.

Zapišimo postopek s psevdokodo:

```
podprogram ObdelajPoddrevo(u):
  p := plača[u];   d := 0;
  za vsakega u-jevega neposredno podrejenega uslužbenca v:
    d := d + ObdelajPoddrevo(v);
    p := max{p, plača[v] + 100};
  if p > plača[u]:
    izpiši ime[u];
    d := d + p − plača[u];   plača[u] := p;
  return d;
```

Postopek poženemo tako, da poiščemo direktorja (to je tisti u, ki nima nadrejenih) in zanj pokličemo ObdelajPoddrevo(u). Ko naš podprogram pride v neko vozlišče u, najprej z rekurzivnimi klici obdela njegova poddrevesa, nato izračuna novo plačo u-ja in vrne skupni znesek, za kolikor so se povečale plače u-ja in vseh njemu posredno ali neposredno podrejenih uslužbencev. Imena uslužbencev, ki se jim plača zviša, sproti tudi izpisujemo.

Koristno je razmisliti še o tem, kako našteti vse u-jeve neposredno podrejene uslužbence v. Preprosta rešitev je, da gremo z v po vseh uslužbencih od 1 do n in pri vsakem pogledamo, če velja $u = šef[v]$. Tako bomo imeli pri vsakem uslužbencu $O(n)$ dela in časovna zahtevnost celotnega postopka ob $O(n^2)$. Boljša rešitev je, da si na začetku, preden prvič pokličemo ObdelajPoddrevo, pripravimo za vsakega uslužbenca seznam podrejenih:

```
  for u := 1 to n:
    podrejeni[u] := prazen seznam;
  for u := 1 to n:
    dodaj u v seznam podrejeni[šef[u]];
```

Ti dve zanki porabita le $O(n)$ časa, s pomočjo seznamov *podrejeni* pa bomo lahko kasneje pri vsakem uslužbencu pregledali njegove podrejene v času, ki je sorazmeren s številom podrejenih. Časovna zahtevnost celotnega postopka bo le še $O(n)$, majhna slabost te rešitve pa je, da porabi $O(n)$ dodatnega pomnilnika (za sezname podrejenih).

5. 3-d labirint

V nalogi se skriva problem preiskovanja grafa. Recimo, da začnemo preiskovati mrežo v celici z najvišjim stebrom. Vpeljimo tabelo $w \times h$ logičnih vrednosti, v kateri element $d[x][y]$ pove, ali smo steber na celici (x, y) že uspeli doseči ali ne. Poleg tega bomo imeli še množico Q, v kateri hranimo koordinate stebrov, ki smo jih že dosegli, nismo pa še pregledali, kam je mogoče iz njih pot nadaljevati. Postopek teče v zanki: vsakič vzamemo nek steber iz Q in pogledamo, v katere njegove sosede je mogoče priti iz njega; če kakšen od teh sosedov doslej še ni bil označen kot dosegljiv, ga zdaj tako označimo in ga dodamo v Q, tako da bomo sčasoma pregledali tudi *njegove* sosede in tako naprej. Ko se Q izprazni, pa vemo, da smo pregledali vse stebre, ki

jih je mogoče doseči z začetnega (to je z najvišjega); takrat moramo le še pregledati, če je tudi najnižji steber zdaj označen kot dosegljiv. Zapišimo našo rešitev še s psevdokodo:

```
(* Poiščimo najvišji in najnižji steber. Vse stebre označimo kot nedosegljive. *)
x_n := 1; y_n := 1; x_v := 1; y_v := 1;
for x := 1 to w do for y := 1 to h do
  d[x][y] := false;
  if v[x][y] > v[x_v][y_v] then x_v := x; y_v := y;
  if v[x][y] < v[x_n][y_n] then x_n := x; y_n := y;
(* Začnimo pri najvišjem stebru. *)
Q := {(x_v, y_v)}; d[x_v][y_v] := true;
(* Preglejmo, kaj vse je dosegljivo iz njega. *)
while Q ni prazna:
  naj bo (x, y) poljuben element Q; pobriši ta element iz Q;
  (* Steber (x, y) je dosegljiv. Preverimo njegove sosede. *)
  za vsako sosedo (x', y') celice (x, y):
    if |v[x][y] − v[x'][y']| > 1 or d[x'][y'] then continue;
    (* Steber (x', y') je mogoče doseči iz (x, y). *)
    d[x'][y'] := true; dodaj (x', y') v Q;
return d[x_n][y_n];
```

V zanki, ki mora pregledati sosede celice (x, y), so to načeloma štiri celice: $(x \pm 1, y)$ in $(x, y \pm 1)$, vendar pa moramo pri vsaki od njih še preveriti, če sploh leži na mreži (torej če je $1 \leq x' \leq w$ in $1 \leq y' \leq h$).

Načeloma je vseeno, v kakšnem vrstnem redu jemljemo stebre iz Q, saj bomo v vsakem primeru prej ali slej obiskali vse stebre, ki so dosegljivi z najvišjega. Pogosta rešitev je, da Q implementiramo z vrsto (*queue*), torej vedno vzamemo iz Q tisti steber, ki je že najdlje v Q; takšnemu vrstnemu redu pregledovanja prostora pravimo *iskanje v širino*.

Naloge so sestavili: sindikat — Nino Bašić; prehod za pešce — Primož Gabrijelčič; listi — Tomaž Hočevar; labirint — Branko Kavšek; zvončki, Trojane — Vid Kocijan; neskončna pokrajina — Vid Kocijan in Primož Gabrijelčič; umor — Jurij Kodre; back from the Klondike, 3-d labirint — Mitja Lasič; najkrajša pot — Matjaž Leonardis in Vid Kocijan; zaokrožanje temperature, rastlinjak, naraščajoče besede — Mark Martinec; najlepši esej — Polona Novak; pomanjkanje sendvičev, šifriranje, v soju žarometov — Jure Slak; popularni dnevi — Mitja Trampuš; pike za tisočice — Janez Brank. Primer v besedilu naloge Naraščajoče besede je iz 25. poglavja *Desetega brata*.

REŠITVE NALOG S CEOI 2017

POSKUSNO TEKMOVANJE

1. Jenga

Opraviti imamo z nepristransko igro za dva igralca. V nepristranski igri imata oba igralca na voljo iste poteze, če sta na potezi v enakem stanju igre. Za vsako stanje igre lahko torej določimo, ali je stanje zmagovalno za igralca na potezi ob predpostavki, da oba igrata optimalno. Stanje igre oz. stolpa je določeno s stanjem posameznih nadstropij, vsako nadstropje pa je lahko v enem izmed 8 stanj (vse podmnožice manjkajočih kvadrov), pri čemer 3 stanja porušijo stolp in zaključijo igro, tako da jih lahko ignoriramo. Poleg tega je nepomemben tudi vrstni red nadstropij v stolpu (z izjemo vrhnjega nadstropja). Če jih poljubno premešamo, bodo zmagovalna stanja še vedno zmagovalna in obratno. Stanje igre lahko torej opišemo s številom ne-vrhnjih nadstropij vsakega od petih tipov. Število kvadrov v vrhnjem nadstropju sledi iz števila ne-vrhnjih nadstropij posameznega tipa.

Minimax. Ugotavljanja, ali je stanje zmagovalno, se lotimo s t.i. pristopom minimax, kjer na svoji potezi izberemo najboljšo, na nasprotnikovi pa za nas najbolj neugodno potezo. Stanje igre ni zmagovalno, če ne moremo odvzeti nobenega kvadra, ne da bi pri tem porušili stolp. Če obstaja poteza, ki nas pripelje v stanje, ki ni zmagovalno, je trenutno stanje zmagovalno. To drži zato, ker s to potezo postavimo nasprotnika v položaj, kjer mora nadaljevati igro v stanju, za katerega že vemo, da iz njega niti ob optimalni igri ne more zmagati. Ker prehajanje med stanji ni ciklično, lahko z omenjeno rekurzivno formulacijo za vsako stanje določimo, ali je zmagovalno.

Če je začetno stanje zmagovalno, bomo začeli prvi, sicer prvo potezo prepustimo nasprotniku. Ko smo na vrsti, bomo vedno lahko izbrali tako potezo, ki bo postavila nasprotnika v izgubljen položaj, vse dokler ni v neki točki prisiljen porušiti stolpa. Za izbiro poteze lahko enostavno preverimo vse možnosti glede tipa nadstropja in kvadra, ki ga bomo odstranili.

Dinamično programiranje. Iz zgornjega opisa hitro pridemo do ideje, da bi bilo dobro shraniti izračunano zmagovalnost posameznih stanj, namesto da to rekurzivno računamo vsakič znova. Ker mora biti v vsakem nadstropju vsaj en kvader, lahko predpostavimo, da stolp ne bo imel več kot $3n$ nadstropij. Vsak izmed 5 tipov nadstropja se lahko pojavi največ $3n$-krat, zato lahko ocenimo število stanj igre z $O(n^5)$. Če hočemo biti bolj natančni, pa izračunamo, da lahko $3n$ nadstropij razdelimo med 5 tipov na $\binom{3n+5-1}{5-1}$ načinov. Število stanj je dovolj majhno za rešitev podnaloge, kjer je $n \leq 40$.

Število stanj pa lahko še zmanjšamo, in sicer lahko namesto 5 tipov nadstropij obravnavamo samo 3: „polno nadstropje“, „nadstropje z manjkajočim stranskim kvadrom“ in „ostalo“. Iz nadstropij prve kategorije lahko odstranimo katerikoli kvader, iz nadstropij druge kategorije pa samo stranski kvader. Iz nadstropij zadnje kategorije (ostalo) ne moremo odstraniti nobenega kvadra, ne glede na to, ali gre za nadstropje s samo sredinskim kvadrom ali pa za nadstropje z obema stranskima

in brez sredinskega kvadra. Stanje igre lahko opišemo s številom (ne-vrhnjih) nadstropij prvega in drugega tipa ter s številom kvadrov v vrhnjem nadstropju. Število nadstropij tretjega tipa namreč potem sledi iz skupnega števila kvadrov. Tako dobimo rešitev s časovno in prostorsko zahtevnostjo $O(n^2)$.

2. Množenje

To je najlažja ogrevalna naloga. Treba je zmnožiti dve celi števili, ki pa sta lahko precej dolgi. Z uporabo 32-bitnih števil lahko rešimo prvo podnalogo, s 64-bitnimi števili pa tudi drugo podnalogo.

Števila v tretji podnalogi so prevelika za celoštevilske tipe v večini programskih jezikov. Implementirati je treba postopek pisnega množenja števil, kot bi ga izvedli s svinčnikom na listu papirja, kar zahteva $O(NM)$ operacij.

V zadnji podnalogi so števila malenkost predolga za prej omenjeni postopek. Že manjše pohitritve za konstanten faktor so dovolj za rešitev naloge. Namesto izvajanja pisnega množenja števko po števko jih lahko združimo v skupine po 8 števk. Dve števili, ki ju tvori po 8 števk, pa lahko zmnožimo s 64-bitnimi celoštevilskimi tipi. S tem pravzaprav predstavimo števila v sistemu z osnovo 10^8 namesto v desetiškem. Taka rešitev je približno 64-krat hitrejša.

Seveda obstajajo tudi drugi algoritmi za množenje števil s časovno zahtevnostjo, ki je nižja od kvadratne (npr. Karatsubov algoritem ali pa hitra Fourierjeva transformacija), vendar presegajo srednješolsko znanje. Najlažji način za rešitev naloge pa je bil z uporabo pythona ali knjižnice `BigInteger` v javi, ki uporabljata omenjene metode hitrega množenja.

3. Muzej

Muzej ima obliko drevesa (torej acikličnega neusmerjenega povezanega grafa), kjer sobe ustrezajo vozliščem, hodniki pa povezavam. Izberimo za koren drevesa sobo, v kateri se na začetku nahaja turist. Če bi se turist moral vrniti v začetno sobo, bi obiskal nek povezan podgraf velikosti d in se pri tem po vsaki povezavi v tem podgrafu premaknil natanko dvakrat (prvič, ko se premika stran od korena, in drugič, ko se vrača proti korenu). Ker pa lahko svoj sprehod zaključi kjerkoli, bodo povezave na poti od korena do končnega vozlišča sprehoda prispevale svojo dolžino samo enkrat in ne dvakrat, kot velja za ostale povezave, ki so del obiskanega podgrafa.

Izčrpno preiskovanje. Omejitve v prvi podnalogi so dovolj majhne, da lahko preverimo vse možne podgrafe in končna vozlišča sprehoda. Število podgrafov je manjše od 2^n, število končnih vozlišč pa ne preseže n. Pri omejitvi $n \leq 20$ v prvi podnalogi je taka rešitev dovolj hitra.

Dinamično programiranje. Pri drugi podnalogi imamo zagotovilo, da stopnja vozlišč ni večja od 3. To pomeni, da ima vsako vozlišče x največ dva otroka (levega in desnega: $L(x), R(x)$) in zato tudi le dve poddrevesi; izjema je koren, ki ima lahko tudi tri. Naj $f(x, k, s)$ predstavlja dolžino najkrajšega takega sprehoda, ki se začne v x, ostane ves čas znotraj poddrevesa s korenom pri x in obišče k vozlišč. Zastavica s pa določa, ali naj se sprehod obvezno zaključi v izhodišču x ($s = T$) ali ne ($s = F$). Odgovor, po katerem nas sprašuje naloga, je torej $f(z, d, F)$.

S $C_{x,y}$ označimo dolžino povezave med vozliščema x in y. Da bi lahko rešili omenjeni podproblem $f(x, k, s)$, se moramo odločiti, koliko vozlišč bomo obiskali

v levem (recimo l) in koliko v desnem poddrevesu ($k-1-l$) vozlišča x. Če se nam ni treba vrniti v izhodišče, lahko eno od povezav proti levemu ali desnemu poddrevesu obravnavamo z utežjo 1 (če v tisto poddrevo sploh ne gremo, pa celo z utežjo 0). Koren celotnega drevesa obravnavamo ločeno in preverimo vse razdelitve števila obiskanih vozlišč (torej $k-1$) med tri poddrevesa korena.

Naslednje enačbe formalizirajo zgornji razmislek. Ooglejmo si najprej lažji primer, ko je $s = T$ in se moramo vrniti na izhodišče. Če želimo v levem poddrevesu obiskati l vozlišč, se moramo premakniti vanj, obiskati $l-1$ vozlišč z vrnitvijo v njegovo izhodišče in se nato premakniti nazaj. Podobno velja za desno poddrevo, kjer obiščemo preostalih $k-1-l$ vozlišč.

$$f(x,k,T) = \min_{0\le l<k} \begin{array}{l} \{\ 2C_{x,L(x)}\,[\![l>0]\!] \quad + f(L(x),l,T) + \\ \quad\ 2C_{x,R(x)}\,[\![l<k-1]\!] + f(R(x),k-1-l,T)\ \}. \end{array}$$

Pri tem ima izraz $[\![\ldots]\!]$ (Iversonovi oklepaji) vrednost 1, če je pogoj v oklepajih izpolnjen, sicer pa 0.

Bolj kompleksen je primer, ko je $s = F$ in se nam po obisku k vozlišč v poddrevesu s korenom pri x ni treba vrniti nazaj na začetek, v vozlišče x. Recimo, da se nam bolj splača zaključiti sprehod nekje v levem poddrevesu (nasprotni primer bomo obravnavali na enak način in na koncu izbrali manjšega od obeh rezultatov). Če želimo v levem poddrevesu obiskati l vozlišč, se bomo najprej premaknili v desno poddrevo, tam obiskali $k-1-l$ vozlišč in se premaknili po povezavi nazaj v koren x. Nato se premaknemo v levo poddrevo in tam obiščemo l vozlišč z zaključkom v poljubnem vozlišču.

$$f(x,k,F) = \min_{0\le l<k} \min \begin{cases} 2C_{x,R(x)}\,[\![l<k-1]\!] + f(R(x),k-1-l,T) + \\ \quad C_{x,L(x)}\,[\![l>0]\!] \quad + f(L(x),l,F), \\ 2C_{x,L(x)}\,[\![l<k-1]\!] + f(L(x),l,T) + \\ \quad C_{x,R(x)}\,[\![l>0]\!] \quad + f(R(x),k-1-l,F). \end{cases}$$

Robni primer pri teh formulah nastopi, če kakšen od x-ovih otrok (ali pa celo oba) ne obstaja. Na primer, če otroka y ni, si moramo zanj misliti $f(y,0,s) = 0$ in $f(y,k,s) = \infty$ za $k > 0$; vrednost $C_{x,y}$ pa si lahko mislimo v tem primeru kot nedefinirano, saj se bo v gornjih formulah tako ali tako pomnožila z izrazom $[\![\ldots]\!]$, ki bo takrat vedno enak 0.

V nadaljnji analizi bomo zaradi preglednosti ignorirali parameter s, obravnavamo pa ga lahko na enak način kot doslej.

Višje stopnje vozlišč. Kako lahko tako rešitev posplošimo na primere, kjer vozlišča največ dveh poddreves? Ne moremo namreč preveriti vseh možnih razdelitev števila $k-1$ (obiskanih vozlišč) med vsa poddrevesa. Gre za pogost problem, s katerim se srečamo pri uporabi dinamičnega programiranja na drevesnih strukturah. Uporabili bomo še en nivo dinamičnega programiranja, da združimo rezultate vseh poddreves vozlišča x. Recimo, da ima x otroke $y_1,\ldots,y_m$ in si zastavimo podproblem: v poddrevesih s korenih pri $y_1,\ldots,y_\mu$ bi radi obiskali κ vozlišč (ne vštevši vozlišča x); najcenejši poti, ki to doseže, recimo $g(\mu,\kappa)$. Rešitev, ki nas res zanima, torej $f(x,k)$, je zdaj ravno enaka $g(m,k-1)$. (Za vsak primer poudarimo, da čeprav v našem zapisu to ni eksplicitno označeno, sta g in m seveda oba odvisna od x.)

Funkcijo g lahko računamo z naslednjim razmislekom: obiskati hočemo κ vozlišč v prvih μ poddrevesih; recimo, da jih od tega obiščemo l v μ-tem poddrevesu; potem nam ostane še vprašanje, kako čim ceneje obiskati $\kappa - l$ vozlišč v prvih $\mu - 1$ poddrevesih. Tako dobimo:

$$g(\mu, \kappa) = \min_{0 \le l \le \kappa} \{g(\mu - 1, \kappa - l) + 2C_{x,y_\mu}[\![l > 0]\!] + f(y_\mu, l)\}.$$

Robni primer pa je $g(1, \kappa) = f(y_1, \kappa)$.

V gornji formuli je prikazano, kot da se C_{x,y_μ} šteje dvojno (če je $l > 0$, torej če sploh gremo v μ-to poddrevo); to ima smisel pri $s = T$, pri $s = F$ pa bi ga morali včasih šteti le enojno.

Za izračun $g(\mu, \cdot)$ uporabljamo samo vrednosti $g(\mu-1, \cdot)$, zato potrebujemo samo $O(k)$ dodatnega pomnilnika. Za vsak $g(\mu, \kappa)$ porabimo $O(\kappa)$ časa, da pregledamo vse primerne l. Ker nas bo na koncu zanimalo imeti $f(x, k)$ za vse možne k (do $k = d$), moramo izračunati $g(\mu, \kappa)$ za vse možne κ od 0 do $d - 1$ in vse μ od 1 do m, kar bo skupaj vzelo $O(d^2 m)$ časa. Vsota tega po vseh x je $O(d^2 n)$, ker je vsota stopenj vseh vozlišč x (torej vseh m-jev) enaka $n - 1$. To je dovolj hitro za rešitev tretje podnaloge.

Izboljšave. Naj bo $S(x)$ število vozlišč v poddrevesu s korenom x (vključno z x samim). Podproblem $f(x, k)$ je torej za $k > S(x)$ nerešljiv (lahko si mislimo tam $f = \infty$) in se je z njim povsem nesmiselno ukvarjati. Z drugimi besedami, na koncu nas bo sicer zanimalo $f(z, d)$, toda zato še ni treba iti pri vsakem x s k-jem vse do d, ampak je dovolj iti le do $\min\{d, S(x)\}$.

Podobno lahko izboljšamo pomožni algoritem dinamičnega programiranja za izračun vrednosti $g(\mu, \kappa)$. Rekli smo, da bi obiskali l vozlišč v μ-tem poddrevesu in $\kappa - l$ vozlišč v prvih $\mu - 1$ poddrevesih; očitno je to smiselno le, če je $l \le S(y_\mu)$ in $\kappa - l \le S(y_1) + \ldots + S(y_{\mu-1})$. Skupaj z dosedanjim pogojem $0 \le l \le \kappa$ tako dobimo

$$\max\{0, \kappa - S(y_1) - \ldots - S(y_{\mu-1})\} \le l \le \min\{\kappa, S(y_\mu)\}.$$

Ta optimizacija je dovolj, da zniža časovno zahtevnost z $O(nd^2)$ na $O(n^2)$, kar je precejšnja izboljšava za velike vrednosti d. Ta izboljšava je presenetljiva, zato si na kratko poglejmo, zakaj pride do tega. Za izračun vrednosti $g(\mu, \cdot)$ obravnavamo vse relevantne pare vrednosti $\alpha := \kappa - l$ in l (na katere se da razbiti κ), kar si lahko vizualiziramo na sledeč način. Ker je $\alpha \le S(y_1) + \cdots + S(y_{m-1})$ in $l \le S(y_m)$, lahko l-ju priredimo eno izmed vozlišč v poddrevesu y_m, vrednosti α pa eno vozlišče iz poddreves $y_1, \ldots, y_{m-1}$. Ker to počnemo za vsak koren x, bomo vsak par vozlišč „obravnavali“ natanko enkrat (ko bo x enak njunemu najnižjemu skupnemu predniku). Vseh parov vozlišč pa je $O(n^2)$ ne glede na vrednost k.

Bolj formalno lahko ta razmislek zapišemo takole. Ko pri nekem konkretnem x in μ računamo vrednosti $g(\mu, \kappa)$ za vse možne κ, je treba pri vsaki κ izračunati minimum po več možnih l. Koliko je parov (κ, l), ki nastanejo na ta način? Pišimo $a_\mu = S(y_1) + \ldots + S(y_{\mu-1})$ in $b_\mu = S(y_\mu)$. Spomnimo se, da smo se omejili le na take l, za katere velja $0 \le l \le b_\mu$ in $0 \le \kappa - l \le a_\mu$. Iz drugega od teh pogojev sledi, da pride pri vsakem l v poštev največ $a_\mu + 1$ različnih vrednosti κ; torej je vseh parov (κ, l) le $(a_\mu + 1)b_\mu$. Skupno časovno zahtevnost celotnega postopka dobimo, če to seštejemo po vseh x in μ:

$$\textstyle\sum_x \sum_\mu (a_\mu + 1) b_\mu = \sum_x \sum_\mu a_\mu b_\mu + \sum_x \sum_\mu b_\mu.$$

Pri drugi od teh dveh vsot je stvar preprosta: $\sum_\mu b_\mu = S(x)-1 < n$ in vsota tega po vseh x je $\le n^2$. Pri prvi vsoti, $\sum_x \sum_\mu a_\mu b_\mu$, pa opazimo, da si lahko zmnožek $a_\mu \cdot b_\mu$ predstavljamo takole: to je ravno število takih parov vozlišč, kjer je eno vozlišče iz poddrevesa s korenom y_μ (takih vozlišč je b_μ), drugo pa iz enega od predhodnih poddreves (tistih s koreni $y_1, \ldots, y_{\mu-1}$; takih vozlišč je a_μ). Ko se to sešteje po vseh μ pri našem trenutnem x, pokrijemo ravno vse take pare vozlišč, pri katerih sta vozlišči iz dveh različnih x-ovih poddreves — torej take pare, pri katerih je x najgloblji skupni prednik obeh vozlišč. Ko to nazadnje seštejemo po vseh x, smo pokrili ravno vse pare vozlišč (razen tistih, kjer je eden od elementov koren; vsak par pokrijemo pri tistem x, ki je najgloblji skupni prednik vozlišč v paru); vseh parov vozlišč pa je $\le n^2$, torej je $\sum_x \sum_\mu a_\mu b_\mu \le n^2$. Tako torej vidimo, da je časovna zahtevnost našega postopka zdaj res le $O(n^2)$, ne glede na vrednost d.

PRVI TEKMOVALNI DAN

1. Enosmerne ceste

Naloga zahteva, da za vsako povezavo ugotovimo, ali v usmerjenem grafu nujno mora imeti neko določeno usmeritev ali pa obstajajo rešitve (primerno usmerjeni grafi) za obe možni usmeritvi te povezave. Pri usmerjanju pa moramo zagotoviti dosegljivost med podanimi pari točk.

Naš vhodni graf (oz. natančneje multigraf, ker lahko med dvema točkama obstaja tudi po več neposrednih povezav) ni nujno povezan; vendar pa točki iz dveh različnih povezanih komponent gotovo nista dosegljivi ena iz druge in to tudi ne bosta postali ne glede na to, kako usmerimo povezave. Zato se lahko v nadaljevanju našega razmisleka ukvarjamo z vsako povezano komponento posebej in ko gledamo le eno komponento, si lahko mislimo, da imamo pred seboj povezan graf.

Cikli. Recimo, da v neusmerjenem grafu obstaja cikel. Povezave, ki sestavljajo cikel, lahko usmerimo v krogu v eno ali drugo smer, pri tem pa so vse točke na ciklu še vedno dosegljive med seboj. Z vidika preostanka grafa sta obe tidve možnosti enakovredni: v obeh primerih velja, da če je iz neke u dosegljiva ena točka cikla, so dosegljive vse, in če je neka u dosegljiva iz ene točke cikla, je tudi iz vseh ostalih. Za te povezave lahko torej takoj zaključimo, da bodo v izhodnem nizu, po katerem sprašuje naloga, dobile odgovor `B`. Poleg tega pa, ker smo pravkar videli, da so si vse točke na ciklu popolnoma enakovredne glede tega, od kod so dosegljive in kaj je dosegljivo iz njih, v nadaljevanju našega razmisleka ni več prave potrebe po tem, da bi te točke med seboj sploh še razlikovali. V mislih lahko zato celoten cikel združimo v eno samo točko (na primer takole: izberimo si neko točko c na ciklu; povezave, ki imajo obe krajišči na ciklu, pobrišimo; tistim povezavam pa, ki imajo na ciklu le eno krajišče, vendar ne c, premaknimo tisto krajišče v c;[7] s tem ostanejo ostale točke cikla brez povezav in se lahko odslej delamo, kot da teh točk ni).

[7] Pri premikanju krajišč v c se lahko zgodi, da iz več različnih povezav nastanejo enake, vzporedne povezave; na primer, če sta c' in c'' na C, se povezavi (u, c') in (u, c'') za neko $u \notin C$ obe spremenita v (u, c). Pomembno pri tem je, da ju še vedno obravnavamo kot dve ločeni povezavi in ju ne zamenjamo z eno samo povezavo — če bi naredili to slednje, bi se lahko zgodilo, da bi se kak cikel izrodil v eno samo povezavo in bi pri nadaljevanju našega postopka spregledali, da je tista povezava nekoč bila na ciklu.

Ta postopek v mislih ponavljajmo, dokler je v grafu še kaj ciklov; prej ali slej postane graf acikličen, torej drevo. Ker je v drevesu pot med vsakim parom vozlišč enolično določena, morajo imeti povezave fiksno usmeritev. Za vsak par mest (x_i, y_i) iz vhodnih podatkov se sprehodimo po poti od x_i do y_i in povezave na njej usmerimo od izhodiščnega proti ciljnemu vozlišču. Povezave, ki niso del nobene izmed teh poti, lahko usmerimo poljubno. Omejitve nam zagotavljajo, da pri usmerjanju ne bo prišlo do konfliktnih situacij. Ta rešitev je pravilna, vendar ne dovolj učinkovita.

Mostovi. Povezave v drevesu, ki nam ostane po skrčitvi vseh ciklov v grafu, predstavljajo mostove v izhodiščnem neusmerjenem grafu. Mostovi so tiste povezave, z odstranitvijo katerih bi graf razpadel na dve povezani komponenti. Prepričajmo se, da je to res, torej da naš postopek krčenja ciklov res pobriše vse povezave, ki niso mostovi, in ne pobriše nobenega mostu.

Pri tem si bomo pomagali z naslednjim opažanjem: ko naš postopek skrči cikel $C = (c_1, c_2, \ldots, c_k)$ v eno samo točko, recimo $c = c_1$, velja za vsako povezavo pred skrčitvijo naslednje: (1) če je bila v tej skrčitvi pobrisana, je pred skrčitvijo ležala na nekem ciklu; (2) če pa ni bila pobrisana, potem leži po skrčitvi na nekem ciklu natanko tedaj, če je že tudi pred njo ležala na kakšnem ciklu.[8]

Iz tega opažanja sledi, da če je neka povezava v prvotnem grafu ležala na kakšnem ciklu, bo tudi po vsaki skrčitvi še vedno ležala na nekem ciklu, dokler je ne bo naš postopek pobrisal; če pa v prvotnem grafu ni ležala na nobenem ciklu, je postopek tudi pri nobeni skrčitvi ne bo pobrisal in bo ostala v drevesu, s katerim se naš postopek konča. Naš postopek je torej pobrisal natanko tiste povezave, ki so v prvotnem grafu ležale na kakšnem ciklu, obdržal pa natanko tiste, ki niso ležale na nobenem ciklu. To pa so ravno mostovi: ($\Rightarrow$) recimo, da povezava (u, v) ne leži na nobenem ciklu; pa recimo, da ni most; torej, če jo pobrišemo iz grafa, mora še vedno

[8] Prepričajmo se, da to drži. (1) Ob skrčitvi cikla C se pobrišejo le take povezave, ki imajo obe krajišči na C, recimo c_i in c_j; če tako povezavo dopolnimo še z ostalimi točkami cikla med c_i in c_j, nastane cikel (ki je mogoče malo krajši od C), torej je (c_i, c_j) tudi ležala na ciklu.

(2, $\Rightarrow$) Recimo, da povezava (u, v) pri skrčitvi ni bila pobrisana in da je pred njo ležala na nekem ciklu. Ker ni bila pobrisana, vsaj eno od njenih krajišč ne leži na C; recimo brez izgube za splošnost, da je to u. Ker u ne leži na C, tudi tisti cikel, na katerem je ležala (u, v), ni bil C, ampak nek drug cikel, recimo $C' = (u, v, x_1, \ldots, x_t)$. Če tudi nobena druga točka s C' ni na C, potem skrčitev sploh ni vplivala na C' in (u, v) je del istega cikla tudi po njej. Če pa je kakšna $x \in C'$ (lahko tudi $x = v$) bila v C, lahko zdaj C' popravimo tako, da vse takšne x zamenjamo s c, torej tisto točko, v katero smo skrčili celoten cikel C. Če se zdaj kje pojavi točka c po večkrat zaporedoma, obdržimo od teh pojavitev le eno (cikel se pri tem gotovo ne izrodi, saj v njem še vedno ostaneta vsaj dve točki — namreč u in c). Zdaj smo dobili nek obhod v novem grafu (po skrčitvi C-ja); gotovo tudi vsebuje povezavo (u, v) (oz. povezavo (u, c), ki je nastala iz (u, v), če je bila $v \in C$). Ta obhod ni nujno cikel, ker lahko vsebuje c po večkrat; ampak v tem primeru lahko od njega obdržimo tisti cikel, ki vsebuje povezavo (u, v). Tako torej vidimo, da (u, v) tudi po skrčitvi cikla C leži na nekem ciklu.

(2, $\Leftarrow$) Recimo, da povezava (u, v) pri skrčitvi ni bila pobrisana in da po njej leži na nekem ciklu $C' = (u, v, x_1, \ldots, x_t)$. Spomnimo se, da se je pri skrčitvi cikel C skrčil v točko c. Če c ne leži na C', to pomeni, da je bil ta cikel v nespremenjeni obliki prisoten že pred skrčitvijo, torej je (u, v) že takrat ležala na ciklu. Ostane še primer, ko c leži na C'. Spomnimo se, da so v ciklu vse točke različne (sicer je to le nek obhod, ne pa cikel), torej c nastopi v C' le enkrat. Označimo njegovo predhodnico v C' z y, njegovo naslednico pa z z (lahko je tudi $y = z$, če ima C' dolžino 2). Ker c nastopi v C' le enkrat, točki y in z gotovo nista iz C, torej sta bili povezavi (y, c) in (c, z) prisotni v grafu tudi pred skrčitvijo C-ja, je pa mogoče, da sta imeli namesto c-ja za krajišče kakšno drugo točko cikla C, torej sta bili to povezavi oblike (y, c') in (c'', z) za neki $c', c'' \in C$. Popravimo zdaj C' tako, da namesto točke c v njem uporabimo $c', \ldots, c''$, pri čemer tri pike predstavljajo tisti del cikla C, ki leži med c' in c''. Tako nastane cikel, ki je bil prisoten v grafu že pred skrčitvijo C-ja in ki vsebuje tudi povezavo (u, v) (oz. tisto povezavo, iz katere je ta ob skrčitvi nastala). Torej je ta povezava res že pred skrčitvijo ležala na nekem ciklu. □

obstajati neka pot od u do v; toda ta pot je v prvotnem grafu skupaj s povezavo (u, v) tvorila cikel, kar je protislovje; torej je bila (u, v) most. ($\Leftarrow$) Recimo, da je povezava (u, v) most; pa recimo, da leži na nekem ciklu; če jo pobrišemo iz grafa, se dá od u do v še vedno priti po preostanku tistega cikla, torej (u, v) ni bila most, kar je protislovje; torej (u, v) ni ležala na ciklu.

Lepo pri tem, da zdaj poznamo tesno povezavo med našim postopkom in mostovi, je to, da lahko mostove poiščemo zelo učinkovito s Tarjanovim algoritmom v linearnem času, kar je dovolj za rešitev druge podnaloge. Različica Tarjanovega algoritma deluje z uporabo vpetega drevesa, ki ga dobimo s preiskovanjem v globino (DFS oz. depth-first search). Za vsako vozlišče x v vpetem drevesu izračunajmo, katero je najvišje táko vozlišče y, ki je sosed kakšnega x-ovega potomca z. Če je to vozlišče y bližje korenu kot x, obstaja alternativa povezavi, ki povezuje x s svojim staršem. V nasprotnem primeru bi odstranitev povezave med x in njegovim staršem odrezala poddrevo s korenom v vozlišču x od preostalega grafa in je zato most.

O tem se lahko prepričamo takole. Spomnimo se, da lahko pri iskanju v širino vsaki točki u pripišemo čas b_u, ko je algoritem vstopil vanjo, in čas e_u, ko je izstopil iz nje, in da za vsako točko velja, da je njen čas vstopa manjši kot pri vseh njenih potomcih, čas izstopa pa večji kot pri njih. Z vidika x lahko ločimo naslednje možnosti glede tega, kje v vpetem drevesu je točka y: (1) lahko je $b_y < b_x$ in $e_x < e_y$: torej je y na veji od korena drevesa do x; (2) lahko je $e_y < b_x$: torej je y levo od omenjene veje (če si predstavljamo drevo tako, da pod vsako točko uredimo njene otroke od leve proti desni po času vstopa); (3) lahko je $e_x < b_y$: torej je y desno od omenjene veje; (4) lahko pa je $b_x \leq b_y$ in $e_y \leq e_x$: torej je y nekje v x-ovem poddrevesu.

Toda primer (2) je nemogoč: to bi pomenilo, da je DFS vstopil v y prej kot v x; takrat je bil x skupaj s celotnim svojim poddrevesom (v katerem je tudi z) še neobiskan; torej bi se DFS iz y med drugim spustil v njegovega soseda z; torej bi bil z otrok točke y in ne bi bil v x-ovem poddrevesu. — Tudi primer (3) je nemogoč: to bi pomenilo, da je bil y še neobiskan, ko smo že zapustili x, torej tudi takrat, ko smo zapustili x-ovega potomca z; torej bi se DFS takrat, ko je obravnaval z, moral od tam spustiti tudi v njegovega soseda y, ki je bil takrat še neobiskan, in y bi bil v x-ovem poddrevesu.

Ostaneta možnosti (1) in (4), ravno med njima pa razločuje naš kriterij, ki se vpraša, ali je y bližje korenu kot x — če je, je na veji (1), če pa ni, je v poddrevesu (4). Če torej ni nobenega takega y, ki bi ležal na veji (1), to pomeni, da ima vsak x-ov potomec z lahko za sosede le točke iz x-ovega poddrevesa, torej iz tega poddrevesa ni mogoče izstopiti drugače kot skozi povezavo med x in njegovim očetom; ta povezava je torej most. Če pa obstaja nek y na veji (1), bi se dalo po povezavi (z, y) izstopiti iz x-ovega poddrevesa (ali priti vanj), tudi če povezavo med x in njegovim očetom pobrišemo, torej ta povezava ni most.

Najnižji skupni prednik. Preostali vir neučinkovitosti algoritma izhaja iz usmerjanja povezav v drevesu. Usmerjanje vsake povezave na vsaki poti $x_i \rightsquigarrow y_i$ je prepočasno, ker bi v določenih primerih usmerjali isto povezavo v isto smer večkrat. Kako se lahko temu izognemo? En pristop vključuje obravnavo poti v koristnem vrstnem redu (glede na njihovega najnižjega skupnega prednika). Izberimo si poljubno točko za koren drevesa, tako da se v drevesu vzpostavi pojem staršev in otrok

(starš točke u je njen neposredni predhodnik na poti od korena do u). Vsako pot v drevesu lahko razdelimo na dva dela — enega, ki se vzpenja proti korenu, in drugega, ki se spušča po drevesu. Poti obravnavamo po vrsti glede na njihovega najnižjega skupnega prednika, pri čemer začnemo s tistimi, katerih prednik je bližje korenu drevesa. Pri usmerjanju povezav na poti od nekega vozlišča proti korenu bomo v nekem trenutku dosegli že usmerjeno povezavo. Vrstni red obravnave povezav nam zagotavlja, da bodo od tu naprej vse povezave že usmerjene. Ker naloga zagotavlja, da ni konfliktov, lahko na tem mestu usmerjanje zaključimo. Tako bomo v drevesu vsako povezavo usmerili največ enkrat.

Oglejmo si še, kako lahko učinkovito, v $O(\log n)$ časa, poiščemo najnižjega skupnega prednika dveh točk. Zapomnimo si globino vsake točke (koren naj ima globino 0) in naj bo $p(u,k)$ tisti prednik točke u, ki je v drevesu 2^k nivojev nad njo (če je u na globini manj kot 2^k, bomo za $p(u,k)$ vzeli kar koren drevesa). Tega ni težko izračunati: $p(u,1)$ je kar oče točke u (razen pri korenu, kjer je $p(u,1)=u$); in ko imamo enkrat $p(u,k)$ za vse u, lahko računamo $p(u,k+1)$ po formuli $p(u,k+1)=p(p(u,k),k)$. Tako porabimo $O(n\log n)$ časa in pomnilnika, da si pripravimo te podatke o prednikih.

Recimo zdaj, da nas zanima najnižji skupni prednik točk u in v. Če nista na isti globini, recimo brez izgube za splošnost, da je u globlje kot v; s pomočjo podatkov o prednikih lahko v $O(\log n)$ časa poiščemo u-jevega prednika na istem nivoju kot v (vse, kar moramo narediti, je, da razliko v globini u-ja in v-ja izrazimo kot vsoto potenc števila 2, in s pomočjo tabele $p(\cdot,\cdot)$ izvedemo ustrezne skoke od u-ja gor po drevesu); recimo mu w, globino teh dveh vozlišč pa označimo z g. Če je $w=v$, je torej kar v sam njun najnižji skupni prednik. Sicer pa glejmo zdaj prednike $p(w,k)$ in $p(v,k)$ za vse višje k in poiščimo najvišji tak k, pri katerem sta $w':=p(w,k)$ in $v':=p(v,k)$ še različna. Če sta enaka že pri $k=0$, smo najnižjega prednika našli, to je kar oče točk w in v. Sicer pa ob koncu tega postopka vemo, da je skupni prednik w in v manj kot 2^k nivojev nad w' in v'. Nadaljujemo lahko z bisekcijo:

while $k>0$:
  (* w' *in* v' *sta različna in imata skupnega prednika* $\le 2^k$ *nivojev nad sabo.*
    Ali ga imata že $\le 2^{k-1}$ *nivojev nad sabo?* *)
  $k:=k-1$; $w'':=p(w',k)$; $v'':=p(v',k)$;
  if $w''\neq v''$ **then** $w':=w''$; $v':=v''$;

Na koncu te zanke je $k=0$ in torej vemo, da imata w' in v' skupnega prednika le en nivo nad sabo, torej svojega očeta.

2. Zanesljiva stava

Naloga zahteva od nas, da maksimiziramo vrednost $v=\min\{\sum_i a_i - n_a - n_b, \sum_j b_j - n_a - n_b\}$, kjer a_i predstavlja ponujeno razmerje s strani stavnice i za prvi izid, n_a je število stav na prvi izid, b_j je razmerje pri stavnici j za drugi izid, n_b pa je število stav na drugi izid. Pri tem gre $\sum_i$ le po tistih stavnicah, kjer smo stavili na prvi izid, $\sum_j$ pa le po tistih, kjer smo stavili na drugi izid.

Izčrpno preiskovanje. Najenostavnejši pristop vključuje preiskovanje vseh možnih podmnožic vplačanih stav. Teh je 2^{2n}, kar je dovolj malo za $n\le 10$ in zato reši prvo podnalogo.

Požrešen pristop. Izraz, ki ga želimo maksimizirati, lahko najprej malenkost poenostavimo. Če vsakemu ponujenemu razmerju odštejemo 1, se izraz poenostavi v $v = \min\{\sum_i a_i - n_b, \sum_j b_j - n_a\}$. Opazimo lahko, da sta, če fiksiramo n_a in n_b, izida med seboj neodvisna: množica stav, ki jih vplačamo za prvi izid, ne vpliva na optimalen izbor množice stav za drugi izid in obratno. Smiselno je torej po velikosti urediti vrednosti a_i in b_i ter nato izbrati n_a največjih vrednosti izmed a_i in n_b največjih vrednosti izmed b_i. To naredimo za vsak možen par (n_a, n_b) in na koncu uporabimo tistega, ki je dal največjo vrednost v. Da bomo to lahko počeli dovolj učinkovito, je koristno, če si vnaprej za vsak $k = 1, \ldots, n$ izračunamo vsoto največjih k števil a_i oz. največjih k števil b_i; lahko pa pare (n_a, n_b) pregledujemo v naraščajočem vrstnem redu po n_a in pri vsakem n_a naraščajoče po n_b, tako da lahko vsote enostavno računamo oz. popravljamo sproti:

uredimo stave padajoče, tako da bo $a_1 \geq a_2 \geq \ldots \geq a_n$ in $b_1 \geq b_2 \geq \ldots \geq b_n$;
$v^\star := 0$; (* *največji zagotovljeni dobiček doslej* *)
for $n_a := 0$ **to** n:
  if $n_a = 0$ **then** $s := 0$ **else** $s := s + a_{n_a}$;
  (* *Zdaj je s vsota n_a največjih števil a_i.* *)
  for $n_b := 0$ **to** n:
    if $n_b = 0$ **then** $t := 0$ **else** $t := t + b_{n_b}$;
    (* *Zdaj je t vsota n_b največjih števil b_j.* *)
    $v := \min\{s_a - n_b, s_b - n_a\}$;
    if $v > v^\star$ **then** $v^\star := v$;

Taka rešitev ima časovno zahtevnost $O(n^2)$ in uspešno reši drugo podnalogo.

Dvojiško in trojiško iskanje. Poglejmo, kaj se dogaja z vrednostjo v, ki jo želimo maksimizirati, pri fiksni vrednosti n_a in različnih vrednostih n_b. Spomnimo se, da je v definiran kot manjša od vrednosti $\sum_i a_i - n_b$ in $\sum_j b_j - n_a$. Če začnemo pri $n_b = 0$, sta tidve vrednosti enaki $\sum_i a_i$ in $-n_a$, zato je minimum dosežen pri drugi od njiju. Če nato pri fiksnem n_a povečujemo n_b, se prva vrednost $(\sum_i a_i - n_b)$ počasi zmanjšuje, druga $(\sum_j b_j - n_a)$ pa počasi povečuje. Zato se minimum obeh (torej v) sprva nekaj časa povečuje, dokler druga vrednost ne preseže prve; odtlej pa je minimum dosežen pri prvi vrednosti in v se začne zmanjševati, ker se tudi prva vrednost še naprej zmanjšuje.

Maksimum funkcije lahko torej poiščemo z bisekcijo po seznamu razlik v vrednosti v med zaporednimi vrednostmi n_b. Te bodo sprva pozitivne, nato negativne, nas pa zanima, kje je meja. Pri tem je koristno, če si vnaprej za vse možne k pripravimo vsote največjih k števil a_i oz. največjih k števil b_i, tako da bomo lahko v za poljubno kombinacijo n_a in n_b izračunali v konstantnem času.

Iskanja maksimuma pa se lahko lotimo tudi s trojiškim iskanjem (*ternary search*). To je postopek za iskanje maksimuma take funkcije $f(x)$, ki najprej ves čas le narašča, od maksimuma naprej pa le pada (v našem primeru je v taka funkcija v odvisnosti od n_b, če vzamemo, da je n_a fiksiran). Recimo, da se maksimum nahaja nekje na intervalu med $x = l$ in $x = d$. Razdelimo interval $[l, d]$ na tretjine pri točkah m_1 in m_2, tako da je $l < m_1 < m_2 < d$. Zdaj lahko razmišljamo takole: če je $f(m_1) \leq f(m_2)$, potem maksimum gotovo ni dosežen na $[l, m_1)$, ker mora od maksimuma naprej funkcija strogo padati, primerjava $f(m_1)$ in $f(m_2)$ pa

je pokazala, da na $[m_1, m_2]$ funkcija očitno ne pada strogo. Če pa se izkaže, da je $f(m_1) \geq f(m_2)$, lahko s podobnim razmislekom zaključimo, da maksimum gotovo ni dosežen na $(m_2, d]$. V vsakem primeru lahko torej interval, na katerem še iščemo maksimum, zmanjšamo za eno tretjino.

Za vsako izmed n vrednosti n_a torej potrebujemo $O(\log n)$ časa za iskanje optimalne vrednosti n_b. Časovna zahtevnost te rešitve je torej $O(n \log n)$.

Linearna rešitev. Ob predpostavki, da so vrednosti a_i in b_i že urejene, obstaja celo linearna rešitev. Za vsako vrednost a_i, na katero se odločimo staviti, se prvi člen v minimumu (s katerim je definiran v) poveča za a_i, medtem ko se drugi zmanjša za 1. Pri tem pa želimo, da je manjši od obeh členov čim večji. Recimo, da je n_b optimalna izbira za n_a. Če povečamo n_a na $n_a + 1$, bi bilo smiselno povečati tudi b_i ali pa ga ohraniti na isti vrednosti, nikakor pa ga ne želimo zmanjšati.

Prepričajmo se, da je to res. Naj bo $A = s(n_a) - n_b$ in $B = t(n_b) - n_a$, tako da je $v(n_a, n_b) = \min\{A, B\}$, pri čemer je n_b optimalna izbira pri tem n_a. Ko povečamo n_a za 1, dobimo $v(n_a + 1, n_b) = \min\{A', B'\}$ za $A' = A + a_{n_a+1}$ in $B' = B - 1$. Če bi potem n_b zmanjšali, bi prišli na $v(n_a + 1, n_b - 1) = \min\{A' + 1, B' - b_{n_b}\}$. Ločimo zdaj dva primera:

1. Če je $A' \geq B'$: torej je $v(n_a + 1, n_b) = B'$; če bi zmanjšali n_b, bi se člen A' povečal v $A' + 1$, člen B' pa bi se zmanjšal (ali ostal enak) v $B' - b_{n_b}$, torej je minimum še vedno dosežen pri drugem členu: zato je $v(n_a + 1, n_b - 1) = B' - b_{n_b} \leq B' = v(n_a + 1, n_b)$, torej zmanjšanje n_b-ja nima smisla.

2. Če je $A' < B'$: predpostavimo, da bi zmanjšanje n_b izboljšalo rešitev. Nova vrednost v-ja, torej $v(n_a+1, n_b-1)$, bi bila v tem primeru $\min\{A'+1, B'-b_{n_b}\}$ in bi morala biti večja od prejšnje, to je od $v(n_a + 1, n_b) = \min\{A', B'\} = A'$. Torej bi veljalo $\min\{A' + 1, B' - b_{n_b}\} > A'$; iz tega sledi $B' - b_{n_b} > A'$, torej (ker je $B' = B - 1$) tudi $B - b_{n_b} > A' + 1 \geq A + 1$; torej je $v(n_a, n_b - 1) = \min\{A + 1, B - b_{n_b}\} = A + 1 > A \geq \min\{A, B\} = v(n_a, n_b)$. Iz tega pa lahko sklepamo, da n_b ni optimalna izbira za n_a, kot smo predpostavili na začetku, ker bi lahko z znižanjem n_b prišli do boljše rešitve pri n_a; prišli smo do protislovja.

To pomeni, da nam ni treba za vsako vrednost n_a iskati optimalne vrednosti n_b povsem od začetka. Če obravnavamo vrednosti n_a v naraščajočem vrstnem redu, lahko s povečevanjem n_b (dokler vrednost v ne začne padati) najdemo optimalno vrednost. Za $n_a + 1$ pa iskanje optimalnega n_b nadaljujemo tam, kjer smo ga ravnokar končali, in tako naredimo zgolj en prehod čez vse možne vrednosti n_b.

3. Mišolovka

Labirint si lahko predstavljamo kot graf, v katerem so sobe predstavljene s točkami, prehodi pa s povezavami. Iz besedila naloge sledi, da je graf povezan, ima n točk in $n - 1$ povezav, torej mora biti drevo.

Izčrpno preiskovanje. Pomagati si moramo z ugotovitvijo, da je smiselno zazidavati prehode pred čiščenjem kateregakoli prehoda. Čiščenje prehoda namreč miši poveča možnosti gibanja, zato se Dumbu splača prehode zazidavati, preden počisti prvega.

Slon bo torej v nekem zaporedju blokiral povezave, nato pa po možnosti počakal nekaj potez, preden začne s čiščenjem povezav, da se miš ustavi v nekem vozlišču, recimo v. S čakanjem pred čiščenjem ne bo nič na slabšem, le miši ne bo odpiral novih možnosti za premik. Ko se miš ustavi, lahko predpostavimo, da so vse stranske povezave ob poti P od vozlišča v do pasti v tem trenutku zazidane ali pa umazane. Če bi bila kakšna povezava še prosta, se jo slonu bolj splača zazidati kot pa naknadno čistiti. Slona torej loči do zmage samo še nekaj umazanih povezav na koncu poti P (pri vozlišču v), ki zadržujejo miš in jih mora počistiti. Če to stori vzdolž poti P od pasti proti miši, bo za to porabil minimalno število potez.

Edina neznanka v opisani rešitvi je nabor in vrstni red zazidav, ki je stvar izčrpnega preiskovanja. Podproblem oz. stanje igre lahko opišemo z množico zazidanih povezav in lokacijo miši; za vsakega izmed $O(n \cdot 2^n)$ podproblemov izračunamo optimalno število potez, ki jih slon od tam potrebuje za zmago. V vsakem stanju lahko slon zazida kakšno novo povezavo ali pa v skladu z zgoraj opisano strategijo počaka, da se miš ustavi, in nato počisti povezave do pasti. Množica zazidanih povezav s slonovimi potezami narašča, zato med temi podproblemi nimamo cikličnih odvisnosti in jih lahko rešujemo sistematično po naraščajočem številu zazidanih povezav.

Miš začne poleg pasti. Izberimo koren drevesa tako, da sovpada s pastjo. V prejšnjem odstavku smo videli, kaj bi se zgodilo, če Dumbo ne bi posredoval, dokler se miš ne ustavi. Miš se bo ustavila v listu, Dumbo bo zazidal vse stranske prehode ob poti proti korenu in počistil vse umazane povezave na tej poti. Izračunamo lahko točno število potez, ki jih potrebuje Dumbo, če se miš ustavi v določenem listu. Recimo temu *teža* lista l in označimo z w_l. Če Dumbo miši ne ovira, bo očitno stekla v list največjo težo. Dumbov cilj pa je, da z zazidavo povezav miši prepreči dostop do listov z visokimi utežmi.

Ker se miš nahaja na globini 1, lahko predpostavimo, da se bo premikala samo navzdol proti listom drevesa. Če bi se v prvi potezi premaknila navzgor, bi končala v pasti. Kasneje pa se tudi ne more premakniti navzgor, ker je umazala povezavo, ki vodi tja. Recimo, da se miš nahaja v vozlišču v, ki ima i otrok $u_1 \dots u_i$. Predpostavimo, da je u_1 najboljša, u_2 pa druga najboljša izbira za miš. Dumbo bi s svojo potezo najraje zazidal prehod do u_1. Ta sprememba ne vpliva na uteži listov v poddrevesih $u_2 \dots u_i$. Povezava (v, u_1) je namreč stranska povezava na vseh poteh do korena iz poddreves $u_2 \dots u_i$ in je že vključena v njihovo težo, ker mora biti v vsakem primeru zazidana. Zazidava povezave neposredno ob miši torej ne spremeni uteži listov v poddrevesih. Dumbo torej lahko zazida povezavo proti u_1, miš pa se premakne po drugi najboljši povezavi v u_2.

Uteži lahko poleg listov posplošimo na poljubno vozlišče. Utež vozlišča v je število Dumbovih potez, ki so potrebne za zmago, če se miš trenutno nahaja v vozlišču v (in se ne more premakniti gor iz v, ker je tista povezava umazana), na potezi pa je Dumbo. Dumbo bo zazidal povezavo proti najbolj neugodnemu otroku, miš pa se bo premaknila po drugi najbolj neugodni povezavi. Utež vozlišča v je torej enaka uteži otroka z drugo največjo utežjo. Uteži lahko izračunamo v linearnem času od listov proti korenu. Iskani rezultat je enak uteži vozlišča, v katerem začne miš.

Splošen primer. Do kakšne spremembe pride v splošnem primeru, ko miš ne začne na globini 1? V tem primeru se bo morda premaknila nekaj korakov navzgor proti korenu, preden se bo začela spuščati proti listom. Dumbo ne more zazidati

povezav na poti proti korenu, ker se potem miš sploh ne bi mogla ujeti v past. Čim se miš premakne navzdol, pa igra poteka tako, kot smo opisali v predhodnih odstavkih. Ko Dumbo izbira, kateri prehod naj zazida, se torej ne sme omejiti samo na povezave ob vozlišču, v katerem se trenutno nahaja miš. Upoštevati mora tudi povezave ob poti P, ki vodi od začetne pozicije miši proti pasti v korenu. Dumbo ne sme zazidati povezav na poti P. Nanjo so pripeta poddrevesa oz. njihovi koreni, ki imajo že omenjeno definicijo uteži.

Recimo, da miš s poti P zavije v poddrevo S_i, ki se odcepi od poti P proti korenu na razdalji i korakov od začetnega položaja miši. Če je Dumbo pred tem zazidal B_i stranskih povezav ob poti P, ki se nahajajo pod vozliščem, od katerega vodi povezava v S_i, bo potreboval $B_i + w_i$ potez, da ujame miš.[9] Dumbo želi torej minimizirati vrednost $B_i + w_i$, česar se lahko lotimo z bisekcijo — predpostavimo, da bo Dumbo potreboval vsaj X potez, in preverimo, ali mu to lahko uspe ali ne.

Ali X potez zadošča, preverimo s simulacijo premikanja miši po poti P in sprotnim zazidavanjem vseh stranskih povezav, za katere velja $w_i + B_i > X$. S „simulacijo“ želimo ugotoviti, ali lahko miši preprečimo vstop s poti P v tako stransko vejo, da bi bilo število potez večje od X. Ko je miš v začetnem vozlišču, moramo nujno zazidati sosednje povezave, ki vodijo do vozlišč z utežjo $w > X$. Če je takih povezav več, ne bo šlo. Če je taka točno ena, jo slon zazida (poveča B za 1); miš lahko sedaj dovolj hitro izigra slona samo, če se premakne navzgor. Če takih povezav ni, lahko to izkoristimo za zazidavo neke stranske povezave višje ob poti P. Težava pa nastane, ker ni očitno, katera bi bila najboljša izbira. Ta problem lahko rešimo s prenašanjem takih neizkoriščenih potez v naslednje korake. V naslednjem vozlišču mora slon zopet blokirati vse povezave, za katere velja $B + w > X$. Če je takih povezav več, kot ima na voljo potez, ne bo šlo. Sicer jih vse zazida, morebitne neizkoriščene poteze pa zopet prenese naprej.

Postopek simulacije se konča bodisi tako, da pridemo do korena, ali pa tako, da na neki točki ugotovimo, da Dumbo z X potezami ne more zmagati. Vsak korak bisekcije (simulacija za nek konkreten X) zahteva $O(n)$ časa, torej je časovna zahtevnost opisane rešitve $O(n \log n)$.

DRUGI TEKMOVALNI DAN

1. Gradnja mostov

Nalogo si najprej poenostavimo tako, da se bomo ukvarjali samo s stebri, ki so del mostu, ne pa tudi s tistimi, ki jih bo treba podreti. Skupna cena izgradnje mostu je sestavljena iz cene zaradi razlik v višinah stebrov in cene rušenja odvečnih stebrov. Mislimo pa si lahko, da bomo porušili vse stebre in postavili nazaj tiste, ki jih dejansko potrebujemo. Ceno rušenja odvečnih stebrov torej dobimo tako, da od vsote vseh cen za rušenje odštejemo cene tistih stebrov, ki jih bomo obdržali.

Dinamično programiranje. Zastavimo si naslednji podproblem: kakšna je najmanjša cena $f(i)$ za izgradnjo mostu, ki se začne s prvim stebrom in konča z

[9] Poudarimo, da se B_i nanaša le na število zazidanih povezav pod trenutnim položajem miši. Morebitne povezave, ki bi jih slon zazidal nad tem položajem, pa so že upoštevane v w_i, kajti ko miš enkrat zavije navzdol, mora slon za zmago zazidati vse stranske povezave ob poti proti korenu, da mu miš kasneje spet ne odtava v kakšno stransko poddrevo.

i-tim? Zadnji odsek v optimalnem mostu bo povezoval nek steber $j < i$ s stebrom i. Tako dobimo rekurzivno formulacijo:

$$\begin{aligned} f(1) &= -w_1 \\ f(i) &= \min_{j<i} \left\{ f(j) + (h_j - h_i)^2 - w_i \right\}. \end{aligned}$$

Taka rešitev z dinamičnim programiranjem mora rešiti $O(n)$ podproblemov, za vsakega pa porabi $O(n)$ časa. Rešitve v naslednjih odstavkih poskušajo izboljšati časovno zahtevnost $O(n^2)$.

Dva stebra. Lahko prejšnjo rešitev izboljšamo, če vemo, da sta v optimalni rešitvi vključena samo dva stebra (poleg prvega in zadnjega)? Če je v optimalni rešitvi samo en steber, lahko enostavno preizkusimo vse možnosti. V primeru dveh stebrov pa bomo preizkusili vse druge stebre i in za vsakega izmed njih na učinkovit način izbrali optimalen prvi steber j.

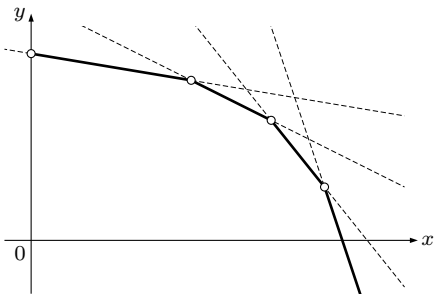
Prispevek prvega stebra k skupni ceni je $(h_1 - h_j)^2 + (h_j - h_i)^2 - w_j$. Za začetek ignorirajmo člen w_j (predpostavimo $w_j = 0$). Najbolj optimalno je za prvi steber izbrati tistega, ki je po svoji višini h_j čim bližje povprečju h_1 in h_i. V to se lahko prepričamo tako, da prispevek prvega stebra zapišemo kot funkcijo njegove višine in jo odvajamo: $c(x) = (h_1 - x)^2 + (x - h_i)^2$; odvod je $c'(x) = 4x - 2h_1 - 2h_i$, kar je enako 0 pri $x = (h_1 + h_i)/2$; tam funkcija g doseže svoj minimum. Zanimata nas torej samo dva kandidata za prvi steber: največji med tistimi, ki so manjši ali enaki $(h_1 + h_i)/2$, in najmanjši med večjimi. V procesu preverjanja drugih stebrov i od prvega proti n-temu lahko v drevesni strukturi hranimo vse stebre j (za vse $j < i$), urejene po višini. Tako lahko v času $O(\log n)$ najdemo tista dva, ki sta najbližje povprečju. Kot drevo lahko uporabimo npr. podatkovno strukturo `set` v programskem jeziku C++.

Preostane nam še obravnava člena w_j. V tem primeru je koristna njihova omejena zaloga vrednosti. Za vsako od 41 vrednosti med -20 in 20 lahko vzdržujemo ločeno drevesno strukturo. Pri iskanju optimalnega stebra pa poiščemo optimalen steber za vseh 41 vrednosti. Časovna zahtevnost rešitve je $O(na \log n)$, pri čemer je $a := \max_i |w_i|$.

Konveksna optimizacija. Zgornjo rekurzivno definicijo iz razdelka o dinamičnem programiranju lahko zapišemo nekoliko drugače.

$$\begin{aligned} f(i) &= \min_{j<i} \{f(j) + (h_j - h_i)^2 - w_i\} \\ &= \min_{j<i} \{f(j) + h_j^2 - 2h_j h_i + h_i^2 - w_i\} \\ &= h_i^2 - w_i + \min_{j<i} \{f(j) + h_j^2 - 2h_j h_i\} \\ &= C_i + \min_{j<i} \{A_j h_i + B_j\} \\ \text{za } A_j &= -2h_j, \quad B_j = h_j^2 + f(j), \quad C_i = h_i^2 - w_i. \end{aligned}$$

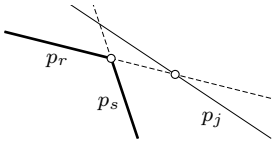
Člen C_i ni odvisen od izbire predhodnega stebra j, ampak samo od zadnjega stebra i, tako da ga lahko pri iskanju minimuma po j obravnavamo kot konstanto in ga nesemo ven iz $\min\{\ldots\}$. Izraz, ki ostane znotraj $\min\{\ldots\}$, je zdaj neka linearna funkcija v odvisnosti od h_i; za vsak j si torej lahko predstavljamo premico z naklonom A_j in začetno vrednostjo B_j. Problem iskanja optimalnega stebra j se tako prevede na iskanje tiste izmed teh premic, ki ima najnižjo vrednost pri $x = h_i$. Najnižje premice lahko iščemo učinkovito z vzdrževanjem spodnje ovojnice (angl. *lower*



Primer ovojnice (debela črta) s štirimi odseki; njihove nosilke so narisane s črtkanimi črtami. Zadnji odsek se razteza v neskončnost. Ovojnico smo začeli pri $x = 0$, saj nas negativni x tu ne bodo zanimali.

envelope) množice premic. To je funkcija $p(x) = \min_j p_j(x)$ za $p_j(x) = A_j x + B_j$. Ker so p_j linearne funkcije, je p odsekoma linearna funkcija, v kateri vsak odsek pripada eni od premic p_j in odseki so urejeni po naklonu (ko x narašča, imajo odseki vse nižji naklon oz. z drugimi besedami, funkcija pada vse hitreje). Te odseke premic lahko hranimo v (primerno uravnoteženi) drevesni strukturi, ki mora biti dinamična — omogočati mora tudi vstavljanje nove premice s poljubnim naklonom v času $O(\log n)$. Taka struktura nam omogoča iskanje optimalne premice (in s tem predzadnjega stebra) in vstavljanje novih. Tudi v tem primeru si lahko pomagamo s podatkovno strukturo `set` v programskem jeziku C++. Ta optimizacija je v povezavi z dinamičnim programiranjem znana pod imenom "convex hull trick" in pospeši rešitev z $O(n^2)$ na $O(n \log n)$.

Za ključe, po katerih so urejeni odseki v drevesu, je koristno vzeti naklone, saj bodo tako odseki urejeni tudi po x-koordinati, poleg tega pa so nakloni cela števila in jih je zato lažje primerjati med sabo. Ko dodajamo novo premico p_j v drevo, najprej poglejmo, kam bi po naklonu sodila; recimo, da med premici p_r in p_s. Nato preverimo, če je p_j sploh treba dodati v drevo; mogoče leži v celoti nad ovojnico. To se zgodi v primeru, če leži presečišče p_r in p_j desno od presečišča p_r in p_s.



Primer dodajanja nove premice p_j. Po naklonu sodi med p_r in p_s, ki sta že na ovojnici; vendar pa je presečišče p_j in p_r desno od presečišča p_r in p_s, torej leži p_j v celoti nad ovojnico in je ni treba dodati vanjo.

Če p_j prestane ta preizkus, jo dodamo v drevo, nato pa moramo še preveriti, če je treba prejšnjih in/ali naslednjih nekaj odsekov pobrisati, ker ležijo v celoti nad p_j in zato niso več na ovojnici; pri tem preverjanju uporabimo analogen kriterij kot prej za p_j. Tako je lahko sicer pri dodajanju ene premice tudi več sprememb v drevesu in vsaka sprememba nam vzame $O(\log n)$ časa, vendar pa je skupno število vseh sprememb v okviru celotnega postopka le $O(n)$, saj lahko vsako premico dodamo v drevo največ enkrat in jo zato tudi pobrišemo največ enkrat; zato je skupna cena vseh operacij na drevesu le $O(n \log n)$.

2. Palindromske razdelitve

V palindromski razdelitvi niza morata biti prvi in zadnji podniz enaka; če ju nato odmislimo, sta v preostanku razdelitve tudi zdaj prvi in zadnji podniz enaka; in

tako naprej. Palindromsko razdelitev niza si lahko torej predstavljamo tako, kot da v vsakem koraku pobrišemo nek podniz z začetka in (enak podniz) tudi s konca vhodnega niza s. Ker iščemo optimalno (najdaljšo) razdelitev, si seveda želimo, da bi bilo takih korakov čim več.

Izčrpno preiskovanje. Kako veliki naj bodo kosi predpon in pripon, ki jih bomo odstranili na vsakem koraku? Preizkusimo lahko vse možne dolžine k in rekurzivno ponovimo postopek za iskanje optimalne razdelitve na preostanku niza:

$$razdeli(s) = \max_{1 \le k \le |n|/2} \begin{cases} 2 + razdeli(s[k : -k]), & \text{če } s[: k] = s[-k :] \\ 1, & \text{sicer.} \end{cases}$$

Zgornji izraz uporablja notacijo po zgledu programskega jezika Python: $s[: k]$ je predpona niza s dolžine k, $s[-k :]$ je taka pripona, $s[k : -k]$ pa vse vmes.

Dinamično programiranje. Zgornja rešitev ima eksponentno časovno zahtevnost v odvisnosti od dolžine vhodnega niza in reši le najmanjše testne primere. Opazimo lahko, da je vhod funkcije *razdeli* popolnoma določen že z dolžino podanega niza. Iz tega sledi, da je možnih samo $O(n)$ različnih klicev funkcije *razdeli* pri iskanju optimalne razdelitve nekega niza s. Namesto s-jevega podniza lahko funkciji *razdeli* podamo kar dolžino tega podniza. Z uporabo memoizacije (pomnjenja že izračunanih vrednosti funkcije *razdeli*, da jih kasneje ni treba računati ponovno) dobimo rešitev s časovno zahtevnostjo $O(n^3)$.

Požrešen pristop. Namesto preverjanja vseh možnih dolžin k, ki jih lahko odščipnemo z začetka in konca niza, se lahko s požrešnim pristopom osredotočimo samo na najmanjši k. Intuitiven argument je, da bomo z odstranjevanjem manjših kosov dobili boljšo razdelitev (torej tako z več kosi). Dokažimo, da je temu res tako.

Brez škode za splošnost lahko predpostavimo, da se požrešni in optimalni algoritem razlikujeta v prvem kosu, ki ga odstranita vhodnemu nizu s: požrešni odstrani kos c, medtem ko optimalni izbere strogo daljši kos $c^\star$; velja torej $|c^\star| > |c|$. Ker se s začenja tako na c kot na $c^\star$ in ker je $c^\star$ daljši od c, mora biti $c^\star$ oblike $c^\star = ca$ za nek niz a. Podobno, ker se s končuje tako na c kot na $c^\star$ in ker je slednji daljši od c, mora biti $c^\star$ oblike $c^\star = bc$ za nek niz b. S protislovjem bomo dokazali, da to v resnici ni mogoče.

Ločimo dva primera:

1. Če je $|c| \le |c^\star|/2$: to pomeni, da sta a in b, ki sta dolga $|c^\star| - |c|$, vsaj tako dolga kot c.

s =	$c^\star$		preostanek niza s	$c^\star$	
=	c	a	preostanek niza s	b	c

   Zgoraj smo že videli, da je $c^\star = ca = bc$; ker se torej $c^\star$ tako začne kot konča na c in ker je c dolg kvečjemu za polovico niza $c^\star$, lahko slednjega zapišemo kot $c^\star = cxc$ za nek niz x (ki je lahko tudi prazen, torej dolžine 0). Če bi iz niza s namesto predpone/pripone $c^\star$ zaporedoma izbrisali c, x in c, bi dobili daljšo razdelitev, kot bi jo vrnil „optimalni“ algoritem, kar je očitno protislovje.

2. Če pa je $|c| > |c^\star|/2$: podoben razmislek kot zgoraj nam zdaj pove, da sta a in b krajša od c.

s =	$c^\star$		preostanek niza s	$c^\star$	
=	c	a	preostanek niza s	b	c

Spet si pomagajmo z dejstvom, da je $c^\star = ca = bc$; ker se $c^\star$ začne tako na c kot na b in ker je b krajši od c, se mora tudi c začeti na b. Pišimo torej $c = bd$ za neko pripono d niza c. Vstavimo to v enačbo $ca = bc$, pa dobimo $bda = bbd$, torej je $da = bd$. Tu za desno stran že vemo, da je enaka c; leva stran nam torej zdaj pove, da d ni le pripona c-ja, ampak tudi njegova predpona. Torej se c začne in konča na d; in ker obenem tudi vemo, da se s začne in konča na c, lahko zaključimo, da se s tudi začne in konča na d. Niz d pa je krajši od c, kar je v protislovju z dejstvom, da je bil c dobljen z našim požrešnim algoritmom in je zato c najkrajši tak niz, na katerega se s začne in konča.

V obeh primerih smo prišli do protislovja, zato lahko zaključimo, da je požrešni pristop optimalen. Implementacija požrešnega algoritma še vedno zahteva $O(n^2)$ časa. Vzrok za to je testiranje, ali je predpona dolžine k enaka priponi enake dolžine in jo torej smemo odščipniti; tako preverjanje zahteva $O(n)$ časa.

Prstni odtisi. Ugotavljanje enakosti nizov lahko pospešimo z uporabo zgoščevalnih funkcij za izračun prstnih odtisov podnizov. Pri tem pa moramo izbrati tako zgoščevalno funkcijo, ki nam bo omogočala hiter izračun novega prstnega odtisa, če nizu dodamo kakšen znak na začetek ali na konec. Pristop je podoben Rabin-Karpovemu algoritmu. Z $A(c)$ označimo kodo ASCII znaka c in izberimo praštevilo $p > A(\mathbf{z})$, npr. $p = 131$. Definirajmo *prstni odtis* niza $c_1c_2\cdots c_m$ kot $h(c_1c_2\cdots c_m) := \sum_{i=1}^{m} A(c_i)p^{i-1} \bmod M$. Da se izognemo računanju z zelo velikimi števili, izberimo za M neko čim večje praštevilo, vendar dovolj majhno, da z njim še lahko računamo z vgrajenimi celoštevilskimi tipi. Ko podaljšujemo predpono in pripono niza s v iskanju prvega kosa, ki ga lahko odščipnemo, lahko namesto predpone in pripone najprej primerjamo njuna prstna odtisa; dejansko primerjavo po znakih, ki je glavni vzrok počasnosti algoritma, pa izvedemo samo takrat, ko se prstna odtisa ujemata. Do tega primerjanja po znakih bo prišlo v veliki večini primerov samo takrat, ko sta si predpona in pripona enaki, takrat pa bomo za to primerjavo porabili recimo $O(k)$ časa (če je k dolžina naše predpone oz. pripone) in nato niz skrajšali za $2k$ znakov (ker bomo to predpono in pripono pobrisali), tako da lahko zaključimo, da v povprečju porabimo le $O(1)$ časa na vsak znak vhodnega niza s. Poleg tega lahko pri povečanju predpone in pripone za en znak popravimo prstna odtisa v $O(1)$ časa. Tako je skupna časovna zahtevnost našega algoritma le $O(n)$.

Deterministična časovna zahtevnost. Rešitev z uporabo zgoščevalnih funkcij je v praksi učinkovita, v teoriji pa bi lahko ob nesrečni izbiri konstant p in M ter ob ravno prav zlobnem nizu s še vedno imeli kvadratno časovno zahtevnost. Nalogo lahko rešimo tudi brez zanašanja na srečo.

Požrešna rešitev z naivnim primerjanjem nizov niti ni tako slaba, če se izkaže, da so predpone/pripone, ki jih režemo stran, dovolj kratke. Z uporabo Z-algoritma ali Knuth-Morris-Prattovega algoritma bi lahko obdelali celoten niz in našli najkrajšo pripono/predpono v linearnem času. Vendar bi bila to izguba časa, če se izkaže, da je iskani košček zelo majhen. Naredimo lahko kompromis. Izberimo neko mejo t in z naivnim primerjanjem nizov preverimo vse koščke dolžine $k < t$. Če ne

najdemo rešitve, obdelamo celoten niz v linearnem času. Časovna zahtevnost takega algoritma je $O((t^2+n)\cdot n/t)$.[10] Če si izberemo $k=\sqrt{n}$, dobimo rešitev s časovno zahtevnostjo $O(n^{1,5})$.

Obstaja tudi deterministična rešitev z linearno časovno zahtevnostjo, čeprav ni najbolj praktična za uporabo na tekmovanju. V linearnem času lahko zgradimo priponsko polje (angl. *suffix array*) a — to je tabela, v kateri so števila i (od 1 do n) urejena glede na leksikografski vrstni red pripon $s[i:]$. Za vsako pripono $s[i:]$ si zapomnimo njeno mesto $p[i]$ v urejenem priponskem polju. Recimo, da smo že odstranili i znakov z začetka in konca niza s, sedaj pa ugotavljamo, ali je k primerna velikost za naslednjo pripono/predpono — torej ali je $s[i:i+k]=s[n-i-k:n-i]$. Ta pogoj je enakovreden pogoju, da se mora pripona $s[n-i-k:]$ začeti na niz $s[i:i+k]$. Spomnimo se, da so v priponskem polju vse pripone urejene leksikografsko; to pa pomeni, da če se več pripon začne na isti niz u, pridejo v leksikografskem vrstnem redu skupaj ena za drugo. Torej tudi za $u=s[i:i+k]$ velja, da pripone, ki se začnejo na u, ležijo v tabeli pripon od nekega začetnega indeksa l do nekega končnega indeksa r. Naš pogoj, da se mora pripona $s[n-i-k:]$ začeti na u, lahko torej preverimo tako, da pogledamo, če je $l\le p[n-i-k]\le r$. Časovno potraten del te rešitve je v prilagajanju mej l in r intervala v priponskem polju a pri povečanju dolžine k. Ko bomo povečali k za 1, se bo naš niz u spremenil iz $s[i:i+k]$ v $s[i:i+(k+1)]$, torej se podaljša za znak $c:=s[i+k]$. Za pripone v $a[l..r]$ že vemo, da se začnejo na u, vprašanje je torej zdaj to, katere od njih se začnejo na uc; to pa so ravno tiste, ki imajo na $(k+1)$-vem mestu znak c. Ker so urejene leksikografsko, lahko prvo in zadnjo med takimi priponami poiščemo z bisekcijo in tako dobimo nova l in r. Bisekcija nam vzame vsakič $O(\log n)$ časa, zato je časovna zahtevnost take rešitve vsega skupaj $O(n\log n)$.

Namesto uporabe bisekcije v priponskem polju lahko (v linearnem času) zgradimo priponsko drevo (angl. *suffix tree*), kar vodi do rešitve z linearno časovno zahtevnostjo. Namesto intervala $l..r$ v priponskem polju zdaj gledamo neko vozlišče v priponskem drevesu (namreč tisto, do katerega se pride iz korena po nizu u) oz. poddrevo, ki se začne pri tem vozlišču. Ko se u podaljša v uc, se le premaknemo iz tega vozlišča v njegovega otroka po povezavi s črko c (če sploh obstaja). Namesto preverjanja $l\le p[n-i-k]\le r$ pa moramo zdaj preveriti, če je med potomci omenjenega vozlišča tudi tisti list drevesa, ki predstavlja pripono $s[n-i-k:]$; tudi to je mogoče v priponskem drevesu narediti v $O(1)$ časa, tako da ima celotna rešitev časovno zahtevnost $O(n)$.

3. Lov

Iz opisa naloge je razvidno, da lahko park modeliramo z drevesom (neusmerjenim acikličnim povezanim grafom), kjer vsako vozlišče predstavlja kip, povezava pa pot v parku.

[10] O tem se lahko prepričamo takole. Naj bo k dolžina najkrajše take predpone, ki je hkrati pripona. Če je $k<t$, bomo to ugotovili v $O(k^2)$ časa z naivnim primerjanjem, niz s pa si bomo s tem skrajšali za k znakov na začetku in na koncu. Če pa je $k\ge t$, bomo to ugotovili v $O(t^2+n)$ časa (najprej $O(t^2)$ za naivno primerjanje do t, nato pa $O(n)$ za enega od linearnih algoritmov na celem nizu), niz s pa si bomo tudi skrajšali za k znakov na začetku in na koncu. Razmerje med porabo časa in tem, za koliko smo si skrajšali k, je pri kratkem kosu torej $k^2:k=k$, pri dolgem pa $(t^2+n):k$; najvišje razmerje dobimo pri $t=k$. Najslabši primer torej nastopi, če je s sestavljen iz $O(n/t)$ kosov dolžine t, poraba časa pa je takrat zato $O((t^2+n))\cdot n/t)$.

Izčrpno preiskovanje. Vsak par vozlišč v drevesu definira pot, ki jih je torej $O(n^2)$. Na vsaki poti lahko odvržemo drobtinice na $O(2^n)$ načinov.[11] Z izčrpnim preverjanjem lahko simuliramo vse možne strategije v času $O(n^2 2^n)$.

Požrešen pristop. Recimo, da se Jerry nahaja v vozlišču v, v katerega je prišel iz vozlišča t, pot pa bo nadaljeval v vozlišču w. Z $N(v) = \sum_{(v,x)\in E} p_x$ označimo število golobov v sosedih vozlišča v. Če Jerry odvrže drobtinico v vozlišču v, se bo razlika v številu golobov, ki jih srečata junaka, povečala za $g(v) = N(v) - p_t$ ne glede na to, kje so bile že odvržene druge drobtinice. Prepričajmo se, da je to res. Če Jerry odvrže drobtinico v vozlišču v, bo Tom srečal še vse golobe s sosednjih vozlišč x, ki niso t ali w. Golobe z vozlišča t je Jerry že srečal, Tom pa jih bo po njihovem premiku srečal v vozlišču v, zato ne prispevajo k razliki. Golobov z vozlišča w Jerry ni srečal in jih tudi ne bo, Tom pa jih bo srečal v vozlišču w. Razlika v številu golobov ob uporabi drobtinice v vozlišču v je torej odvisna samo od predhodnega vozlišča t.

Če je začetno vozlišče Jerryjeve poti znano vnaprej, lahko izračunamo vse razlike $g(v)$, ker so v drevesu predhodna vozlišča točno določena z začetnim vozliščem. Izbrati moramo tako končno vozlišče, da bo vsota d največjih razlik (če imamo na voljo d drobtinic) na poti od začetnega do končnega vozlišča čim večja. Ob preiskovanju drevesa v globino lahko hranimo vse razlike od korena do trenutnega vozlišča v prioritetni vrsti in vsakič poiščemo d največjih. Če poznamo začetno vozlišče, zahteva tak algoritem $O(nd \log d)$ časa, sicer pa $O(n^2 d \log d)$, če moramo preveriti vsako potencialno začetno vozlišče. Še ena možnost je, da razlike namesto v prioritetni vrsti hranimo v uravnoteženem drevesu (npr. rdeče-črnem ali pa v AVL-drevesu); v njem naj bodo urejene naraščajoče po vrednosti $g(v)$, vsako vozlišče pa naj hrani tudi vsoto vseh vrednosti v svojem poddrevesu. Tako lahko vsoto največjih d dobimo v $O(\log n)$ časa, časovna zahtevnost celotnega postopka pa je potem $O(n^2 \log n)$.

Dinamično programiranje. Poljubnemu vozlišču lahko določimo vlogo korena drevesa. Optimalna pot bo sestavljena iz dveh delov: en del se bo dvigoval proti korenu, drugi pa se bo spuščal stran od korena. Za vsako vozlišče lahko izračunamo optimalen odsek poti, ki se spušča iz tega vozlišča, ter optimalen odsek, ki se dviguje proti temu vozlišču. S $c(i,j)$ označimo največjo razliko, ki jo lahko z dosežemo na poti, ki se začne v vozlišču i in nadaljuje v enem izmed poddreves, pri čemer lahko na poti v poddrevesu odvržemo največ j drobtinic. Podobno definiramo še vrednost $b(i,j)$, ki naj predstavlja največjo razliko, ki jo lahko dosežemo na poti po drevesu navzgor — če začnemo nekje v poddrevesu vozlišča i, pot končamo v vozlišču i, pri tem pa odvržemo največ j drobtinic v poddrevesu ali vozlišču i.

Obe funkciji lahko dokaj enostavno izračunamo od listov proti korenu. Pri tem obravnavamo dve možnosti: v vozlišču odvržemo drobtinico in zmanjšamo število

[11] Pravzaprav je ta ocena malo pesimistična; če moramo izmed (največ) n vozlišč na poti izbrati d vozlišč, kjer odvržemo drobinice, lahko to naredimo na „le“ $\binom{n}{d} = \frac{n!}{d!\,(n-d)!}$ načinov. Če je na primer d dovolj majhen, je to bolj podobno $O(n^d)$ kot $O(2^n)$. V vsakem primeru pa je izčrpno preiskovanje tu uporabno le za zelo majhne testne primere.

razpoložljivih drobtinic ali pa ne.

$$\begin{aligned} c(i,0) &= 0 \\ c(i,j) &= \max_k\{c(k,j), c(k,j-1) + N(k) - p_i\} \\ b(i,0) &= 0 \\ b(i,j) &= \max_k\{b(k,j), b(k,j-1) + N(i) - p_k\}. \end{aligned}$$

Pri tem gre k po vseh otrocih vozlišča i.

Če predpostavimo, da optimalna pot seže najvišje v drevesu do vozlišča i, potem je za tako pot največja razlika enaka vsoti najboljše poti, ki se povzpne do vozlišča i, in najboljše poti, ki se spusti iz vozlišča i (npr. v poddrevo, ki se začne pri i-jevem otroku k). Ker ne vemo, katero je optimalno poddrevo k in koliko drobtinic porabiti na vsakem izmed obeh delov poti, izberemo maksimum vseh možnosti: $\max_k \max_{1\le j\le d} b(i,j) + c(k,d-j)$. Ta rešitev pa ima manjšo težavo. Oba odseka poti namreč ne smeta biti del istega poddrevesa. Vozlišče k ne sme biti del optimalne poti $b(i,j)$, sicer bi se odseka poti prekrivala med seboj. To težavo rešimo tako, da hranimo najboljši dve vrednosti za podproblema $b(i,j)$ in $c(i,j)$. Poleg vrednosti pa shranimo tudi, v katerem poddrevesu taka pot poteka. Tako lahko z obravnavo štirih možnosti najdemo par najboljših vrednosti, ki ne bosta konfliktni. Časovna zahtevnost algoritma je $O(nd)$.

REZERVNE NALOGE

1. Laser tag

Naivno iskanje. Prvo podnalogo lahko rešimo tako, da za vsako poizvedbo preverimo vse ovire in poiščemo najbližjo, ki jo laser zadane. Časovna zahtevnost je $O(qn)$. Izziv naloge pa je, kako organizirati podatke oz. ovire, da bomo lahko odgovarjali na poizvedbe bolj učinkovito.

Stiskanje koordinat. Majhne diskretne koordinate nam običajno precej poenostavijo reševanje naloge. Podane koordinate lahko preštevilčimo na manjši interval, ne da bi pri tem spremenili rešitev. V navpični smeri lahko preprosto uredimo vse vrednosti in nato višine ovir po vrsti preštevilčimo s celimi števili od 0 naprej od najnižje proti najvišji. Moramo pa si shraniti podatek o tem, kateri dejanski vrednosti ustreza katera od stisnjenih koordinat, da lahko na koncu pravilno izračunamo razdalje med ovirami. V vodoravni smeri so relevantne koordinate a_i in b_i. Če bi razrezali prostor z navpičnimi rezi pri koordinatah $a_i - 1/2$ in $b_i + 1/2$ (torej tik pred in za posamezno oviro), bi dobili sama homogena območja. Tako lahko stisnemo koordinate na interval $[0, n-1]$ v navpični in $[0, 2n-1]$ v vodoravni smeri.

Prelet ravnine s premico. Množica daljic, ki jih preseka navpična premica pri stisnjeni koordinati x, se ne razlikuje prav dosti od tiste pri $x+1$, kar lahko izkoristimo. Naredimo prelet ravnine s premico, pri čemer vzdržujemo množico aktivnih oz. presekanih daljic v primerni podatkovni strukturi. Med tem postopkom lahko sproti odgovarjamo še na poizvedbe, če so podane v naraščajočem vrstnem redu x-koordinat, kot nam to zagotavlja druga podnaloga. Podatkovna struktura aktivnih daljic mora poleg dodajanja in odstranjevanja daljic iz strukture omogočati tudi učinkovite poizvedbe o najbližji oviri. Zaradi stisnjenih y-koordinat lahko zgradimo

tabelo, ki na indeksu d_i (ki predstavlja višino i-te ovire) hrani vrednost t_i (torej čas, ko se ta ovira pojavi). Nad to tabelo zgradimo statično drevesno strukturo, ki bo v vsakem listu hranila pripadajočo vrednost v tabeli. V notranjih vozliščih pa nam pridejo prav minimalne vrednosti v listih poddrevesa — najzgodnejši časi, ko se v pripadajočem navpičnem območju pojavi neka ovira.

Za poizvedbo oz. strel z višine y ob času u moramo učinkovito najti najbližji tak indeks v tabeli, ki je večji ali enak y (ovira je nad strelom) in na katerem je v tabeli vrednost, ki je manjša ali enaka u (ovira se pojavi pred časom strela). Območje $[y, n]$, ki ga določa poizvedba, lahko razdelimo na $O(\log n)$ disjunktnih območij, ki ustrezajo vozliščem v našem drevesu. Rešitev se bo nahajala v prvem (najnižjem) takem območju (recimo temu vozlišču drevesa v), ki vsebuje vrednost, manjšo ali enako u. Točno celico določimo tako, da se iz vozlišča v počasi spustimo proti listom. Na vsakem koraku se moramo odločiti, ali bomo nadaljevali pot v levem ali desnem poddrevesu. Če levo poddrevo vsebuje dovolj visoko vrednost, se premaknemo vanj, ker je bližje izhodišču. Sicer se premaknemo v desno poddrevo, za katerega smo lahko tedaj prepričani, da bo vseboval primerno vrednost. Z opisanim postopkom lahko na vsako poizvedbo odgovorimo v $O(\log n)$ časa. Prav tako lahko vsako spremembo drevesa zaradi dodajanja in odstranitve daljic naredimo v $O(\log n)$ časa. Skupna časovna zahtevnost je torej $O((n + q) \log n)$.

Obstojno drevo. Rešitev s preletom ravnine zahteva, da sproti odgovarjamo na poizvedbe, ko smo s preletom na ustreznem mestu. V nalogi pa so poizvedbe podane v zakodirani obliki, ki nas prisili, da nanje odgovarjamo po vrstnem redu, ki je lahko povsem naključen. Pri premagovanju te ovire si lahko pomagamo z obstojnimi oz. vztrajnimi podatkovnimi strukturami (angl. *persistent data structures*). Z vsako spremembo podatkovne strukture (npr. vstavljanjem, brisanjem) dobimo novo verzijo. Obstojne podatkovne strukture hranijo poleg trenutne strukture tudi vse svoje pretekle verzije. Pri vsaki spremembi bi seveda lahko naredili kopijo celotne strukture; bistvo obstojnih podatkovnih struktur pa je, da smo lahko pri tem bolj učinkoviti. Pri spremembi vrednosti, ki jo hranimo v listu drevesne strukture, lahko pride do spremembe tudi pri vseh vozliščih na poti do korena. Preostanek drevesa pa ostane nespremenjen, kar lahko izkoristimo.

Eden od primernih pristopov za gradnjo obstojnih dreves je metoda kopiranja poti (angl. *path copying*). Ko naredimo spremembo v nekem listu drevesa, naredimo kopijo vseh vozlišč na poti do korena in v njih izračunamo nove vrednosti. Vsa ta vozlišča imajo enega otroka nespremenjenega in lahko s pripadajočim kazalcem kažejo na isto poddrevo kot prejšnja verzija drevesa. Časovna zahtevnost se zaradi tega ne poveča, ker moramo v vsakem primeru izračunati nove vrednosti v vozliščih na poti proti korenu. Poveča pa se prostorska zahtevnost, in sicer z $O(n)$ na $O(n \log n)$, ker vsaka izmed $O(n)$ sprememb drevesa ustvari $O(\log n)$ novih vozlišč. Ko opravimo celoten prelet ravnine nastane, skupaj tudi $O(n)$ novih korenov drevesa, ki ustrezajo različnim verzijam drevesne strukture. Za odgovarjanje na posamezno poizvedbo moramo najprej najti koren drevesa, ki ustreza x-koordinati poizvedbe. To naredimo z bisekcijo v $O(\log n)$ časa. Poizvedba na drevesu pa je povsem enaka kot v prej opisani rešitvi. Skupna časovna zahtevnost algoritma je torej $O((n+q) \log n)$.

Drevo dreves. Če obstojnih podatkovnih struktur ne poznamo, lahko nalogo dovolj dobro rešimo tudi takole. Videli smo že, da gredo x-koordinate ovir po

stiskanju le od 0 do $2n$, njihove y-koordinate pa od 0 do n. Recimo za začetek, da x-koordinate odmislimo, kot da bi se vse ovire raztezale po celotni širini našega skladišča. Pripravimo tabelo n celic, v kateri bo za vsak y zapisan najzgodnejši čas, ob katerem se na tem y pojavi ovira; nad njo zgradimo še pol manjšo tabelo, ki hrani minimume po dveh zaporednih celic prve tabele; nad to drugo tabelo zgradimo na enak način še eno pol manjšo in tako naprej. Nastane drevo, zelo podobno tistemu, ki smo ga videli pri preletu ravnine. Vse te tabele pravzaprav lahko zložimo v eno samo, ki ima zdaj približno $2n$ celic; tako lahko ob strelu zelo učinkovito v $O(\log n)$ časa ugotovimo, katera je najnižja ovira, ki jo ta strel zadane.

Ker pa imajo ovire tudi x-koordinate, moramo rešitev še malo dopolniti. Na misel nam lahko pride, da bi za vsak možni x (od 0 do $2n$) zgradili eno drevo $T(x)$, ki bi vsebovalo vse tiste ovire, ki (med drugim) pokrivajo tudi to x-koordinato. Pri poizvedbi bi morali le pogledati v tisto drevo, ki ustreza x-koordinati strela. Težava je, da se v najslabšem primeru lahko zgodi, da bo treba vse ovire dodati v vsa drevesa, zaradi tega pa bi gradnja dreves trajala $O(n^2)$ časa. Zato poleg dreves $T(x)$, ki pokrivajo le eno x-koordinato, vpeljimo tudi drevesa za širše intervale x-koordinat. Drevo $T_k(x)$ naj pokriva razpon x-koordinat od vključno $2^k x$ do vključno $2^k(x+1)-1$. Naša prvotna drevesa $T(x)$ so tako zdaj pravzaprav $T_0(x)$. Tako si lahko predstavljamo ta drevesa zložena v drevo dreves, pri čemer ima drevo $T_k(x)$ otroka $T_{k+1}(2k)$ in $T_{k+1}(2k+1)$, kajti njuna intervala skupaj dasta ravno interval drevesa $T_k(x)$.

Pri dodajanju ovir v drevesa se bomo držali pravila, da dodamo oviro v $T_k(x)$ le, če ovira v celoti pokriva interval tega drevesa, ne pokriva pa v celoti intervala njegovega starša v drevesu. Tako pride ovira na vsakem nivoju (pri vsakem k) v največ dve drevesi, skupaj torej v največ $O(\log n)$ dreves, zato gradnja vseh dreves vzame $O(n \log n)$ časa. Pri strelu pa moramo zdaj pogledati v eno drevo na vsakem nivoju (torej pri vsakem k), namreč na tisto drevo, čigar interval vsebuje x-koordinato tega strela. Med vsemi ovirami, ki jih na ta način odkrijemo, pa moramo vrniti najnižjo. Ker moramo pogledati v $O(\log n)$ dreves in imamo pri poizvedbi v posamezno drevo $O(\log n)$ dela, je cena ene poizvedbe zdaj $O((\log n)^2)$. Skupni čas celotne rešitve je torej $O(n \log n + q(\log n)^2)$. To je za faktor $O(\log n)$ slabše od rešitve z obstojnim drevesom, vendar je z dovolj učinkovito implementacijo lahko tudi ta rešitev dovolj hitra, da reši naše testne primere v okviru časovne omejitve. Pri tem nam pride prav predvsem to, da lahko drevesa implementiramo s tabelami in so dostopi do pomnilnika zato bolj lokalni kot pri obstojnem drevesu.

2. Podzaporedja

Naivna rešitev. Pri prvi podnalogi je niz s dovolj kratek, da lahko preverimo vsako podzaporedje črk velikosti m. V najslabšem primeru bomo obravnavali $\binom{20}{10} < 200\,000$ podzaporedij, kar je dovolj malo za rešitev te podnaloge.

Vzorci dolžine 2. Naj $f(p,i,j)$ predstavlja, kolikokrat se niz p pojavi kot podzaporedje v podnizu $s_i s_{i+1} \ldots s_j$. Uporabljali bomo še skrajšano notacijo $f(p,i) = f(p,1,i)$, ki naj pove, kolikokrat se vzorec pojavi v predponi (dolžine i) niza s. Če je naš vzorec dolg le 2 znaka, recimo $p = p_1p_2$, velja naslednja zveza:

$$f(p_1p_2,i,j) = f(p_1p_2,j) - f(p_1p_2,i-1) - f(p_1,i-1) \cdot f(p_2,i,j)$$

V zgornji enačbi začnemo z vsemi pojavitvami vzorca do mesta j, ki jim odštejemo vse pojavitve pred mestom i in vse pojavitve, ki se začnejo pred mestom i in končajo nekje od i do j. Vrednosti $f(a, i)$, kjer je a posamezna črka, lahko izračunamo vnaprej z enim samim prehodom čez niz od leve proti desni, pri katerem hranimo pogostosti posameznih črk. Podoben pristop deluje tudi v primeru dveh črk, torej za $f(ab, i)$. Vsakič, ko srečamo črko b med prehodom iz leve proti desni, dodamo k vrednosti $f(ab, i)$ število pojavitev črke a do sedaj (torej levo od trenutne črke b).

Drevo. V nalogi imamo opravka z množico poizvedb na različnih območjih niza. V takih primerih je koristno vnaprej izračunati določene vrednosti na manjših območjih niza, ki jih lahko nato združimo in s tem dobimo rezultat poizvedbe na večjem območju. Zgradimo lahko binarno drevesno strukturo (angl. *segment/range tree*), v kateri vsako vozlišče predstavlja neko območje (podniz) v nizu s. Listi drevesa predstavljajo posamezne znake; en nivo više imamo notranja vozlišča, ki predstavljajo po dva zaporedna znaka, nato pride nivo z vozlišči, ki predstavljajo po štiri zaporedne znake in tako naprej. V splošnem torej r-to vozlišče na nivoju k predstavlja podniz $s_i \dots s_j$ za $i = 2^k(r-1)+1$ in $j = \min\{2^k r, n\}$, njegova otroka pa sta $(2r-1)$-vo in $2r$-to vozlišče na nivoju $k-1$ (njuna podniza skupaj tvorita ravno podniz njunega starša).

V vsakem vozlišču hranimo podatke o tem, kolikokrat se vsak izmed $O(m^2)$ podnizov vzorca pojavi kot podzaporedje v tistem območju oz. podnizu niza s, ki ustreza temu vozlišču. Recimo, da neko vozlišče ustreza podnizu $s' = s_i s_{i+1} \dots s_j$. V tem vozlišču bomo hranili matriko $m \times m$ vrednosti $g(u, v)$ — kolikokrat se v nizu s' kot podzaporedje pojavi $p_u p_{u+1} \dots p_{v-1}$. Za izgradnjo drevesa potrebujemo način za združevanje rezultatov iz dveh sosednjih vozlišč. Vrednost $g(u, v)$ v starševskem vozlišču izračunamo tako, da obravnavamo vse primere, kjer se prvih k črk pojavi v levem otroku, preostale črke vzorca pa v desnem (matriki obeh otrok označimo z g_1 in g_2):

$$g(u, v) = \sum_k g_1(u, u+k) \cdot g_2(u+k, v)$$

Drevo vsebuje $O(n)$ vozlišč, vsako vozlišče pa hrani tabelo velikosti $O(m^2)$. Za izračun vsake vrednosti potrebujemo $O(m)$ časa, torej je skupna časovna zahtevnost izgradnje podatkovne strukture enaka $O(nm^3)$. Glavna težava te rešitve pa leži v učinkovitosti odgovarjanja na poizvedbe. Za vsako poizvedbo moramo združiti rezultate oz. matrike iz $O(\log n)$ vozlišč, ki s svojimi območji natančno pokrijejo celotno območje poizvedbe (pri tem združevanju rezultatov uporabljamo enak razmislek kot zgoraj pri izračunu g iz g_1 in g_2). Časovna zahtevnost odgovora na posamezno poizvedbo je tore $O(m^3 \log n)$, ker je dovolj dobro za tretjo podnalogo, ne pa tudi za zadnjo.

Vzorci poljubne dolžine. Rešitev z izgradnjo drevesa nam je omogočala relativno učinkovito odgovarjanje na katerokoli izmed $O(n^2)$ možnih poizvedb. Nas pa zanima q točno določenih poizvedb. To lahko izkoristimo s posplošitvijo rešitve za odgovarjanje na poizvedbe dolžine 2. Velja namreč naslednje:

$$f(p_1 \dots p_m, i, j) = f(p_1 \dots p_m, j) - \sum_{k=1}^{m} f(p_1 \dots p_k, i-1) \cdot f(p_{k+1} \dots p_m, i, j)$$

Zgornja enačba je posplošitev primera $m = 2$. Vsem rešitvam do vključno mesta j odštejemo tiste, pri katerih se prvih $k \in [1, m]$ črk pojavi pred mestom i, preostanek pa v območju od i do j. Vrednosti $f(p_u \dots p_v, i)$ lahko izračunamo vnaprej v $O(nm^2)$ na sledeč način.

$$f(p_u \dots p_v, i) = \begin{cases} f(p_u \dots p_v, i-1) & \text{če } p_v \neq s_i \\ f(p_u \dots p_v, i-1) + f(p_u \dots p_{v-1}, i-1) & \text{če } p_v = s_i. \end{cases}$$

Pomagali smo si torej z opažanjem, da se lahko $p_u \dots p_v$ pojavi v $s_1 \dots s_i$ bodisi tako, da ta pojavitev uporabi tudi znak s_i, ali pa tako, da ga ne uporabi; pri tem lahko do prve možnosti pride le, če se zadnji znak niza (s_i) ujema z zadnjim znakom vzorca (p_v).

Osredotočimo se zdaj na člen $f(p_{k+1} \dots p_m, i, j)$ v prej omenjeni rekurzivni zvezi za $f(p_1 \dots p_m, i, j)$. Območje, definirano s spremenljivkama i in j, se ne spreminja. Prvi argument funkcije pa vedno sestoji iz pripone vzorca p. Pri vsaki poizvedbi moramo torej rešiti samo $O(m)$ podproblemov in ne $O(m^2n^2)$, kot bi morda napačno sklepali ob hitrem ogledu argumentov funkcije. Algoritem, ki odgovori na vse poizvedbe, ima torej časovno zahtevnost $O((n+q)m^2)$.

3. Popravilo ceste

Za začetek uredimo prebivalce naraščajoče po x-koordinati in jih v tem vrstnem redu oštevilčimo, tako da bo $x_1 < x_2 < \ldots < x_n$. (Ob tem pa si moramo zapomniti tudi njihove prvotne indekse, da bomo lahko na koncu izpisali njihove želje p_i v pravem vrstnem redu.)

Razmislimo zdaj o tem, kaj se dogaja s soseščino (kot je definirana v besedilu naloge), ko se premikamo po cesti. Na levem krajišču, torej pri $x = 0$, tvori soseščino najbolj levih k prebivalcev, torej $\{1, 2, \dots, k\}$. Premikajmo se počasi proti desni; ko prečkamo mejo $x = (x_1 + x_{k+1})/2$, izpade prebivalec 1 iz soseščine, vanjo pa namesto njega pride prebivalec $k+1$, ker smo odslej bližje njemu kot prebivalcu 1. Zato je zdaj soseščina enaka $\{2, 3, \dots, k+1\}$. Do naslednje spremembe pride, ko prečkamo točko $x = (x_2 + x_{k+2})/2$, ko iz soseščine izpade prebivalec 2 in vanjo pride prebivalec $k+2$; in tako naprej.

Tako si lahko predstavljamo, da je cesta razdeljena na $s = n - k + 1$ *segmentov*, pri čemer segment j tvorijo vse tiste točke x, pri katerih je soseščina enaka $\{j, j+1, \dots, j+k-1\}$. V prejšnjem odstavku smo že videli krajišča teh segmentov, iz njih pa lahko izračunamo tudi njihove dolžine; naj bo ℓ_j dolžina segmenta j.

Za segment j naj bo $g_j := p_j + p_{j+1} + \ldots + p_{j+k-1}$ število prebivalcev (v pripadajoči soseščini), ki podpirajo podjetje 1. Naj bo $r_j := 1$, če je $g_j > k/2$, sicer pa $r_j := 0$. Celoten segment j bo torej popravilo podjetje r_j.

Po teh definicijah lahko zaključimo, da skupna dolžina tistih delov ceste, ki jih popravi podjetje 1, znaša $d_1 = \sum_{j=1}^{s} r_j \ell_j$.

Razmislimo zdaj o Joejevi rešitvi. Naj bo μ_j število tistih njegovih vzorcev (izmed točk $t_1, t_2, \dots, t_m$), ki ležijo na segmentu j. Vrednosti μ_j lahko učinkovito izračunamo tako, da vzorce uredimo naraščajoče in to zaporedje zlijemo z zaporedjem krajišč segmentov.

Za vsak vzorec, ki leži na segmentu j, Joe opazi, da ga popravi podjetje r_j. Skupno število vzorcev, ki jih popravi podjetje 1, je torej $m_1 := \sum_{j=1}^{s} r_j\mu_j$, zato bo Joe izračunal rešitev $d_J = (m_1/m)d = \sum_{j=1}^{s} r_j\mu_j d/m$.

Vidimo torej, da je razlika med pravilno in Joejevo rešitev enaka $d_1 - d_J = \sum_{j=1}^{s} r_j w_j$ za $w_j = \ell_j - \mu_j d/m$. V tem izrazu so r_j edine stvari, na katere imamo kaj vpliva; vrednosti ℓ_j in μ_j (in s tem w_j) sledijo iz vhodnih podatkov in jih ne moremo spreminjati.

Naloga zahteva, naj maksimiziramo absolutno vrednost $|d_1 - d_J|$. Toda mislimo si poljubno rešitev $p_1, \ldots, p_n$ in definirajmo novo rešitev tako, da vse želje obrnemo: $\hat{p}_i = 1 - p_i$. Zato zdaj vsak segment popravlja drugo podjetje kot prej: $\hat{r}_j = 1 - r_j$. Razlika med pravilno in Joejevo rešitvijo je zdaj $\hat{d}_1 - \hat{d}_J = \sum_{j=1}^{s} \hat{r}_j w_j = \sum_{j=1}^{s}(1 - r_j)w_j = \sum_{j=1}^{s} w_j - \sum_{j=1}^{s} r_j w_j = \sum_{j=1}^{s}(\ell_j - \mu_j \cdot d/m) - (d_1 - d_J) = (d - m \cdot d/m) - (d_1 - d_J) = -(d_1 - d_J)$. Tako torej za vsak nabor želja $p_1, \ldots, p_n$ obstaja nek komplementarni nabor, pri katerem je razlika med Joejevo in optimalno rešitvijo enaka po absolutni vrednosti, vendar nasprotno predznačena. Zato je dovolj že, če namesto $|d_1 - d_J|$ maksimiziramo kar $d_1 - d_J$, saj bo rezultat enak.

Dinamično programiranje. Do optimalne rešitve lahko pridemo z dinamičnim programiranjem. Naj bo $f(j, p_j, \ldots, p_{j+k-1})$ največja možna razlika med pravilno in Joejevo rešitvijo, če se omejimo na prvih j segmentov in če so vrednosti $p_j \ldots, p_{j+k-1}$ fiksirane tako, kot pravijo argumenti funkcije f (vrednosti $p_1, \ldots, p_{j-1}$ pa si smemo izbrati poljubno). To nas pripelje do naslednje rekurzivne rešitve:

$$f(j, p_j, \ldots, p_{j+k-1}) = \max\{\, f(j-1, p_{j-1}, \ldots, p_{j+k-2}) + r_j w_j \;: \\ p_{j-1}, p_{j+k-1} \in \{0,1\}\,\}.$$

(Ne pozabimo, da je v tem izrazu r_j v resnici funkcija vrednosti $p_j, \ldots, p_{j+k-1}$.) Robni primer nastopi pri $j = 1$, ko imamo $f(1, p_1, \ldots, p_k) = r_1 w_1$. Rešitev, po kateri sprašuje naloga in ki jo moramo na koncu izpisati, pa je $\max\{f(s, p_{n-k+1}, \ldots, p_n) : p_{n-k+1}, \ldots, p_n \in \{0,1\}\,\}$. Tako lahko z dinamičnim programiranjem rešimo nalogo v $O(s \cdot 2^k)$ časa in $O(2^k)$ prostora. Ta rešitev je primerna za prvo podnalogo, pri kateri je k majhen.

Rešitev za velike k, namreč $k \geq n/2$. Videli smo že, da skušamo maksimizirati $d_1 - d_J = \sum_{j=1}^{s} r_j w_j$. Ker mora biti vsak r_j bodisi 0 bodisi 1, lahko to vsoto maksimiziramo tako, da vzamemo $r_j = 1$, če je $w_j > 0$, in $r_j = 0$, če je $w_j < 0$; če pa je $w_j = 0$, je vseeno, kaj vzamemo za r_j. V splošnem ni vsako zaporedje r-jev $(r_1, \ldots, r_s)$ veljavno, ker ni nujno, da obstaja neko tako zaporedje p-jev $(p_1, \ldots, p_n)$, iz katerega bi nastalo tisto zaporedje r-jev. Če pa je $k \geq n/2$, se izkaže, da je vsako zaporedje r-jev veljavno, tako da bomo zlahka sestavili takšno zaporedje p-jev, ki pripelje do prej omenjenega optimalnega zaporedja r-jev (torej $r_j = 1$ pri $w_j > 0$ in $r_j = 0$ sicer). Tako bomo lahko optimalno rešili drugo podnalogo.

Naše p_i bomo izbirali tako, da bo pri vsakem j veljalo $g_j = (k-1)/2$, če je $r_j = 0$, in $g_j = (k+1)/2$, če je $r_j = 1$. (To dvoje lahko združimo: pri vsakem j bo veljalo $g_j = (k-1)/2 + r_j$.) Z drugimi besedami, želje naših prebivalcev bodo vedno tik pod ali tik nad pragom $k/2$, odvisno od tega, kakšen r_j hočemo doseči.

Naše zaporedje p-jev, $(p_1, \ldots, p_n)$, lahko v mislih razdelimo na tri dele: *levi* del sestavlja prvih $s-1$ členov, *desni* del zadnjih $s-1$ členov, *srednji* del pa srednjih $V = n - 2(s-1) = 2k - n$ členov. (Spomnimo se, da se ukvarjamo tu le s primeri, ko

je $k \geq n/2$. Če pri tem velja stroga enakost, bo $V = 0$, torej bo srednji del prazen, kar nas tudi ne bo nič motilo.) Opazimo lahko, da sta levi in srednji del skupaj dolga natanko k členov, prav tako pa tudi srednji in desni del skupaj.

Oglejmo si *prehod* iz r_j v r_{j+1}. Prehode razdelimo na tri skupine: *rastoče* ($r_j = 0$ in $r_{j+1} = 1$), *padajoče* ($r_j = 1$ in $r_{j+1} = 0$) in *stalne* ($r_j = r_{j+1}$). Pri rastočem prehodu moramo imeti g_j pod $k/2$ in g_{j+1} nad njim, torej bo $g_{j+1} > g_j$. Spomnimo se, da je $g_{j+1} = g_j - p_j + p_{j+k}$; torej je lahko g_{j+1} večji od g_j le, če je $p_j = 0$ in $p_{j+k} = 1$. Podoben razmislek nam pove, da če je prehod $r_j \to r_{j+1}$ padajoč, mora biti $p_j = 1$ in $p_{j+k} = 0$; in če je ta prehod stalen, mora biti $p_j = p_{j+k}$.

(Opozorimo še na to, da kadarkoli smo v prejšnjem odstavku omenili p_j, je ta pripadal levemu delu našega zaporedja; in kadarkoli smo omenili p_{j+k}, je pripadal desnemu delu našega zaporedja.)

V zaporedju r-jev, $(r_1, \ldots, r_s)$, je $s - 1$ prehodov oblike $r_j \to r_{j+1}$; označimo število padajočih prehodov s P, rastočih z R in stalnih s S. Tako je $P+R+S = s-1$. Po rastočem prehodu je zaporedje r-jev pri vrednosti 1 in preden lahko naredi še kak rastoč prehod, mora najprej pasti na 0. Torej mora biti med dvema rastočima prehodoma po en padajoč in obratno. To pomeni, da se lahko število rastočih in padajočih prehodov razlikuje največ za 1: $|R - P| \leq 1$.

Omejitve, ki smo jih videli doslej, nam povedo, da mora biti v levem delu zaporedja p-jev R ničel in P enic, na ustreznih mestih v desnem delu pa mora biti R enic in P ničel; za ostalih S členov levega dela pa velja, da morajo biti enaki ustreznim členom desnega dela. Med temi S členi je lahko nekaj ničel in nekaj enic; naj bo S_1 število enic. Dosedanje omejitve ne povedo ničesar o srednjem delu; recimo, da je med njegovimi V členi V_1 enic, ostali pa so ničle. Pišimo $U := S + V$ in $U_1 := S_1 + V_1$.

Videli smo že, da je $R + P + S = s - 1$; če na obeh straneh prištejemo V, vidimo tudi, da je $R + P + U = k$.

Oglejmo si $g_1 = p_1 + \ldots + p_k$. Ta vsota pokrije ravno celoten levi in srednji del zaporedja p-jev; tu je skupno $P + U_1$ enic. Torej je $g_1 = P + U_1$. Mi pa bi radi, da bi bil g_1 enak $(k-1)/2 + r_1$. Videli smo že, da je $|R - P| \leq 1$, zato lahko ločimo tri možnosti:

(1) Če je $R = P$: v tem primeru imamo $k = R + P + U = 2P + U$; ker je k sod, $2P$ pa lih, mora biti torej U lih in zato večji od 0. Razdelimo U na dve „polovici“ po $u := (U - 1)/2$ členov in še en preostali člen. Zapolnimo eno „polovico“ z ničlami, drugo z enicami, preostali člen pa postavimo na r_1. Tako dobimo $U_1 = u + r_1$ in ni težko pokazati, da je $g_1 = P + U_1 = (k - U)/2 + (u + r_1) = (k - 1)/2 + r_1$, prav to pa smo tudi želeli.

(2) Če je $R = P+1$: torej je rastočih prehodov več kot padajočih; torej mora biti prvi prehod rastoč, zato se mora zaporedje r-jev začeti z $r_1 = 0$. Iz $k = R + P + U$ zdaj dobimo $k = 2P + 1 + U$, kar pomeni, da mora biti U sod. Razdelimo ga na dve polovici s po $u := U/2$ členi in zapolnimo eno polovico z ničlami, drugo pa z enicami. Zdaj imamo $g_1 = P+U_1 = (k-1-U)/2+U/2 = (k-1)/2 = (k-1)/2+r_1$ (kajti $r_1 = 0$), prav to pa smo tudi želeli.

(3) Če je $P = R+1$, razmišljamo podobno kot v primeru (2); tudi tokrat moramo uporabiti $U/2$ ničel in $U/2$ enic.

Tako smo torej videli, da lahko vrednosti p-jev v levem in srednjem delu vedno

postavimo tako, da bo g_1 dobil želeno vrednost $(k-1)/2 + r_1$. Zaradi omejitev, ki povezujejo levi in desni del zaporedja p-jev, so s tem enolično določene tudi vrednosti členov v desnem delu. Poleg tega nam te omejitve zagotavljajo, da se prehodi v zaporedju g-jev natanko ujemajo s tistimi v zaporedju r-jev: kadarkoli imamo rastoč prehod iz $r_j = 0$ v $r_{j+1} = 1$, imamo tudi rastoč prehod iz $g_j = (k-1)/2$ v $g_{j+1} = (k+1)/2$ in podobno za padajoče in stalne prehode. Vidimo torej, da bo pri vseh j veljalo $g_j = (k-1)/2 + r_j$, tako da je naše zaporedje p-jev res pripeljalo do prav takega zaporedja r-jev, kakršnega smo želeli.

V splošnem (torej pri podnalogah 3 in 4) lahko poskušamo najti čim boljše (čeprav ne nujno optimalno) zaporedje p-jev z raznimi hevrističnimi optimizacijskimi postopki; pri naših poskusih se je dobro obneslo simulirano ohlajanje (prišlo je tudi zelo blizu optimalnih rešitev pri podnalogah 1 in 2).

Zgornja meja za točkovanje. Kot pravi besedilo naloge, je točkovanje relativno glede na neko zgornjo mejo za največjo možno vrednost $|d_1 - d_J|$. Zgornja meja, ki smo jo uporabljali, je $\sum_{j=1}^{s} r_j w_j$, pri čemer smo vzeli $r_j = 1$ pri $w_j > 0$ in $r_j = 0$ drugod. (Kot smo videli zgoraj, je ta meja pri tistih testnih primerih, ki imajo $k \geq n/2$, celo tesna.)

Naloge so sestavili: množenje, muzej, enosmerne ceste, zanesljiva stava, gradnja mostov, laser tag, podzaporedja — Tomaž Hočevar; jenga, mišolovka, lov — Vid Kocijan; palindromske razdelitve — Mitja Trampuš; popravilo ceste — Janez Brank.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2015

1. Pobeg

Verjetno najpreprostejša rešitev je, da gremo v zanki po razbojnikih in pri vsakem od njih še v vgnezdeni zanki po vseh časovnih intervalih, v katerih je izhod zastražen. Pri vsakem intervalu preverimo, ali naš trenutni razbojnik poskuša pobegniti ravno v tem času; če da, potem vemo, da ni pobegnil (in nam ostalih intervalov pri tem razbojniku niti ni treba pregledovati). Če takega intervala ne najdemo, pa lahko zaključimo, da je pobegnil, in povečamo števec pobeglih zapornikov; tega na koncu vrnemo kot rezultat, po katerem sprašuje naloga. Zapišimo to rešitev v C++ (predpostavili smo, da vrednosti a_i, b_i, c_i dobimo v tabelah, kjer gredo indeksi od 0 naprej namesto od 1 naprej):

```
int Pobeg(int n, int m, int a[], int b[], int c[])
{
  int stPobeglih = 0; /* Število pobeglih razbojnikov. */
  /* Preglejmo vse razbojnike. */
  for (int i = 0; i < n; i++)
  {
    bool pobegnil = true;
    /* Pojdimo po vseh intervalih in glejmo, ali je razbojnik i
       poskušal pobegniti v času, ko je bil izhod zastražen. */
    for (int j = 0; j < m; j++)
      if (a[j] <= c[i] && c[i] < b[j]) { pobegnil = false; break; }
    /* Če je pobegnil, povečajmo število pobeglih razbojnikov. */
    if (pobegnil) stPobeglih++;
  }
  return stPobeglih;
}
```

Naloga ne pove natančno, ali to, da je izhod zastražen od časa a_i do časa b_i, pomeni, da zapornik, ki poskuša pobegniti točno ob času a_i ali b_i, lahko pobegne ali ne. V gornji rešitvi (in tudi še v ostalih rešitvah, ki si jih bomo ogledali v nadaljevanju) smo predpostavili, da ob času a_i razbojnik ne more pobegniti, ob času b_i pa lahko.

Časovna zahtevnost gornje rešitve je $O(n \cdot m)$, kar je lahko pri velikih n in m že neugodno počasi. Oglejmo si še nekaj učinkovitejših rešitev. Lahko se opremo na dejstvo, da so časi podani v minutah znotraj enega samega dneva. Dan ima le $24 \cdot 60 = 1440$ minut, zato si lahko privoščimo tabelo logičnih spremenljivk (tip **bool** oz. boolean), v kateri bo za vsako minuto pisalo, ali je tisto minuto izhod zastražen ali ne. Na začetku postavimo vse vrednosti v tej tabeli na **false**, nato pa za vsak časovni interval $[a_i, b_i)$, v katerem je izhod zastražen, postavimo ustrezne vrednosti v tabeli na **true**. Nato se moramo le še sprehoditi po vseh razbojnikih in pri vsakem s pomočjo tabele preveriti, ali je ob času c_i lahko pobegnil ali ne.

```
int Pobeg2(int n, int m, int a[], int b[], int c[])
{
  /* Pripravimo tabelo, ki za vsako minuto pove, ali je izhod takrat zastražen.
     Na začetku postavimo vse elemente na false. */
  bool straza[1440]; for (int t = 0; t < 1440; t++) straza[t] = false;
  for (int i = 0; i < m; i++)
```

```
    /* Za interval (a_i, b_i) označimo, da je takrat stražar prisoten. */
    for (int t = a[i]; t < b[i]; t++) straza[t] = true;
  /* Za vsakega razbojnika poglejmo, ali uspe pobegniti ali ne. */
  int stPobeglih = 0; for (int i = 0; i < n; i++) if (! straza[c[i]]) stPobeglih++;
  return stPobeglih;
}
```

V splošnem ima ta rešitev časovno zahtevnost $O(n+m+T)$, če gledamo obdobje T minut in če se intervali (a_i, b_i) ne prekrivajo. Če pa se smejo ti intervali prekrivati, se v najslabšem primeru lahko zgodi, da mora notranja zanka po t pri vsakem i pregledati vse minute, zato je časovna zahtevnost takrat $O(n + m \cdot T)$. V vsakem primeru pa porabimo tudi $O(T)$ prostora, medtem ko je porabila prejšnja rešitev le $O(1)$ prostora. Naša nova rešitev je torej privlačna predvsem zato, ker je T v našem primeru majhen, le 1440.

Še ena možnost pa je, da tako razbojnike kot intervale pregledujemo naraščajoče po času. Ker naloga nič ne govori o tem, kako so vhodni podatki urejeni, jih bomo za začetek uredili sami: razbojnike uredimo naraščajoče po c_i, intervale pa po a_i. Za vsak j definirajmo $B_j := \max\{b_1, \ldots, b_j\}$. Zdaj se bomo pri pregledovanju razbojnikov in intervalov držali naslednjega načela: preden se bomo začeli ukvarjati z j-tim intervalom, torej z (a_j, b_j), bomo pregledali vse razbojnike, za katere čas pobega c_i leži pred B_{j-1}; pri j-tem intervalu pa nato pregledamo vse razbojnike, za katere je c_i na intervalu $[B_{j-1}, B_j)$ (ta interval je lahko tudi prazen, namreč če je $b_j \leq B_{j-1}$ in zato $B_j = B_{j-1}$).

Pri vsakem od teh razbojnikov lahko razmišljamo takole. (1) Če je $c_i < a_j$, to pomeni, da je c_i za vsemi intervali stražarjev od (a_1, b_1) do (a_{j-1}, b_{j-1}) (ker je $c_i \geq B_{j-1}$) in pred vsemi intervali od (a_j, b_j) do (a_m, c_m) (ker je $c_i < a_j \leq a_{j+1} \leq \ldots \leq a_m$ — spomnimo se, da smo intervale uredili po času začetka). Tak razbojnik je torej uspel pobegniti. (2) Druga možnost pa je, da je $c_i \geq a_j$. Ker se ukvarjamo le z razbojniki na $[B_{j-1}, B_j)$ in imamo na tem intervalu očitno vsaj razbojnika i, to pomeni, da je ta interval neprazen; torej je $B_j > B_{j-1}$; in ker je $B_j = \max\{B_{j-1}, b_j\}$, je to mogoče le tako, da je $B_j = b_j$. Naš razbojnik torej leži na $[B_{j-1}, b_j)$, torej je $c_j < b_j$; skupaj s $c_i \geq a_j$ to pomeni, da leži na j-tem intervalu, torej mu stražar takrat prepreči pobeg.

Ko gremo tako v zanki po vseh intervalih, bomo s tem razmislekom odkrili vse razbojnike, ki jim ne uspe pobegniti. Oglejmo si še implementacijo te rešitve v C++. Doslej smo predpostavljali, da naš podprogram dobi števila a_i in b_i v dveh ločenih tabelah, kar pa je za urejanje parov (a_i, b_i) po a_i malo bolj nerodno, zato jih bomo za začetek skopirali v vektor parov in uredili tega.[12]

```
int Pobeg3(int n, int m, int a[], int b[], int c[])
{
  /* Uredimo razbojnike in pare (a[i], b[i]). */
  vector<pair<int, int>> p;
```

[12] Potencialna slabost tega je, da porabimo $O(m)$ dodatnega pomnilnika. Z nekaj truda se lahko temu sicer izognemo: bodisi napišemo svojo funkcijo za urejanje, ki primerno premika istoležne elemente tabel `a` in `b` hkrati; bodisi napišemo svoj iterator, ki ga bomo lahko podtaknili funkciji `std::sort` iz standardne knjižnice in ki bo interno hranil par iteratorjev, enega v tabelo `a` in enega v tabelo `b`; lahko pa prevalimo problem na klicatelja in spremenimo deklaracijo naše funkcije tako, da bo kot parameter zahtevala seznam parov (a_i, b_i), ne pa dveh ločenih seznamov, enega za a_i-je in enega za b_i-je.

```
  for (int i = 0; i < m; i++) p.emplace_back(a[i], b[i]);
  sort(c, c + n); sort(p.begin(), p.end());
  /* Preglejmo intervale vseh stražarjev. */
  int stUjetih = 0;
  for (int i = 0, j = 0; j < m; j++)
    /* Pišimo a_i = p[i].first in b_i = p[j].second. Naj bo B_j = max{b_0, ..., b_{j-1}}.
       Na tem mestu velja: za razbojnike k = 0, ..., i - 1 velja c[k] < B_j
       in med njimi je „stUjetih“ takih, ki jim ni uspelo pobegniti.
       Za razbojnike k = i, ..., n - 1 pa velja c[k] ≥ B_j.
       Preglejmo zdaj razbojnike, za katere je B_j ≤ c[i] < b_j;
       tiste med njimi, ki imajo c[i] ≥ a_j, bo ujel stražar j. */
    for ( ; i < n && c[i] < p[j].second; i++)
      if (c[i] >= p[j].first) stUjetih++;
  return n - stUjetih;
}
```

Ta rešitev porabi $O(n \log n + m \log m)$ časa za urejanje obeh zaporedij (razbojnikov in časovnih intervalov), nato pa le še $O(n + m)$ časa za pregled obeh zaporedij in štetje pobeglih razbojnikov. To je veliko bolje od prve rešitve (podprogram Potek), od druge (Potek2) pa je sicer slabše, vendar ima to prednost, da ne dela nobenih predpostavk glede časov (npr. da so to cela števila do 1440; uporabljeni postopek bi deloval tudi, če časi sploh ne bi bili cela števila).

Za konec pa si oglejmo še eno preprosto in zelo elegantno rešitev. Časovne trenutke a_i, b_i in c_i si lahko predstavljamo kot točke na časovni premici; složimo vse te trenutke v en sam dolg seznam in jih uredimo naraščajoče po času. Pri vsakem od njih pa si tudi zapomnimo, kaj predstavlja: poskus pobega (torej je to eden od c_i), začetek straže (torej eden od a_i) ali konec straže (torej eden od b_i). Nato se sprehodimo po tem seznamu in vzdržujmo števec stražarjev, ki so trenutno na straži (v spodnji rešitvi je to spremenljivka stStrazarjev; če se intervali ne prekrivajo, bo njena vrednost vedno 0 ali 1, sicer pa bo mogoče tudi večja od 1). Ko v seznamu pridemo mimo kakšnega a_i, moramo števec stražarjev povečati, pri b_i ga moramo zmanjšati, pri poskusu pobega c_i pa le pogledamo, ali je števec stražarjev trenutno enak 0 (tedaj je pobeg uspel in lahko povečamo števec pobeglih razbojnikov) ali večji od 1 (tedaj pobeg ni uspel).

Paziti moramo še na naslednjo podrobnost: če se ob istem času zgodi več dogodkov, je koristno obravnavati začetke in konce intervalov prej kot poskuse pobega, tako da bomo, ko se bomo ukvarjali s poskusi pobega, že imeli pripravljeno pravo število stražarjev ob tem času. Spodnja rešitev za to poskrbi tako, da dogodke predstavi s pari $\langle$*čas*, *tip dogodka*$\rangle$, pri čemer tip dogodka 0 pomeni začetek intervala, 1 konec intervala, 2 pa poskus pobega, tako da, če ob istem času nastopi več dogodkov, obdelamo med njimi začetke in konce intervalov prej kot poskuse pobegov.

```
int Pobeg4(int n, int m, int a[], int b[], int c[])
{
  /* Pripravimo seznam vseh dogodkov (začetki/konci intervalov, poskusi pobegov). */
  vector<pair<int, int>> pari;
  for (int i = 0; i < m; i++) { pari.emplace_back(a[i], 0); pari.emplace_back(b[i], 1); }
  for (int i = 0; i < n; i++) pari.emplace_back(c[i], 2);
  sort(pari.begin(), pari.end());
  int stPobeglih = 0, stStrazarjev = 0;
  /* Preglejmo dogodke po naraščajočem času. */
  for (const auto &p : pari)
```

```
    if (p.second == 0) stStrazarjev++;          // Začetek straže.
    else if (p.second == 1) stStrazarjev--;     // Konec straže.
    else if (stStrazarjev == 0) stPobeglih++;   // Poskus pobega.
  return stPobeglih;
}
```

Ta rešitev ima časovno zahtevnost $O((n+m)\log(n+m))$ in prostorsko $O(n+m)$. Uporabljeni postopek je primer tehnike, ki se sicer veliko uporablja zlasti v računski geometriji in je tam znana kot „prelet“ (*sweep*) premice, ravnine ipd.

2. Igra 2048

Ker potekajo premiki in združitve le v smeri navzdol, je dogajanje v posameznem stolpcu neodvisno od dogajanja v drugih stolpcih. Zato lahko (v zanki po x) obravnavamo vsak stolpec posebej. Znotraj posameznega stolpca pojdimo v še eni zanki (po y) po vrsticah od spodaj navzgor, pri tem pa vzdržujmo podatek o tem, do katere višine lahko pade ploščica iz trenutne vrstice (v spodnji rešitvi je to spremenljivka yy). Poleg tega imejmo še logično spremenljivko lahkoZdruzi, ki pove, ali se lahko ploščica po padcu iz vrstice y v yy združi s svojo spodnjo sosedo (torej tisto v vrstici yy + 1).

Če je v trenutni vrstici (y) prazna celica, se lahko takoj premaknemo naprej (navzgor) in pustimo yy nespremenjen; sicer pa vemo, da bo ploščica iz trenutne vrstice načeloma padla v yy, nato pa moramo še preveriti, če se lahko združi s spodnjo sosedo (torej če imata enako vrednost). Če je združitev možna, podvojimo vrednost ploščice v vrstici yy + 1, pustimo yy nespremenjen in v spremenljivki lahkoZdruzi označimo, da še ena združitev zdaj ni več možna. Sicer (če ni prišlo do združitve) pa premaknemo trenutno ploščico v yy, zmanjšamo yy za 1 in v lahkoZdruzi označimo, da ta ploščica lahko sodeluje v kakšni bodoči združitvi.

Koristno je vzdrževati še podatek o tem, ali je pri trenutni potezi sploh prišlo do kakšne spremembe. Do spremembe pride pri vsaki združitvi in pri vsakem takem premiku iz y v yy, če sta y in yy različna. To si zapomnimo v spremenljivki spremembe.

```
enum { w = ..., h = ... };
int a[h][w] = ...; /* Globalna spremenljivka z vsebino mreže. */

bool Poteza()
{
  bool spremembe = false;
  for (int x = 0; x < w; x++)
  {
    int yy = h - 1;              /* Vrstica, v katero se premakne ploščica iz vrstice y. */
    bool lahkoZdruzi = false;    /* Ali se lahko ploščica po premiku v yy združi
                                    s tisto v vrstici yy + 1? */
    for (int y = h - 1; y >= 0; y--) if (a[y][x] > 0)
    {
      int n = a[y][x]; a[y][x] = 0;
      if (lahkoZdruzi && a[yy + 1][x] == n)
        a[yy + 1][x] += n, lahkoZdruzi = false, spremembe = true;
      else {
        if (y < yy) spremembe = true;
        a[yy--][x] = n; lahkoZdruzi = true; }
    }
```

```
  }
  return spremembe;
}
```

Zdaj torej znamo izvesti eno potezo. Naloga pravi, da moramo izvajati poteze, dokler se na mreži še kaj spreminja, torej potrebujemo še eno zanko, ki kliče funkcijo Poteza, dokler ta vrača **true**:

```
void Simulacija()
{
  bool spremembe;
  do { spremembe = Poteza(); }
  while (spremembe);
}
```

3. znajdi.se

Razlika v primerjavi z nalogo 2015.1.3 je, da zdaj na vhodu nimamo eksplicitno podanega začetnega in končnega kraja, zato bomo morali sami ugotoviti, v katerem kraju se pot začne. To ni težko: začetek poti je v tistem kraju, v katerem se nek korak začne, noben korak pa se v njem ne konča. Zato pa je koristno, če za vsak kraj pripravimo ne le podatek o tem, kam se pot iz njega nadaljuje, pač pa tudi to, ali ima ta kraj na poti kakšnega predhodnika ali ne.

Vhodne podatke lahko beremo znak po znak; vse znake, ki niso velike črke, lahko preskočimo. Prva velika črka, ki jo vidimo v trenutni vrstici, je ime kraja, v katerem se trenutni korak začne; zapomnimo si jo v spremenljivki od. Ko pridemo v isti vrstici do še ene velike črke, recimo c, vemo, da je v opisu poti prisoden korak od kraja od do kraja c. Vzdrževali bomo dve tabeli, nasl in pred, v katerih si lahko zdaj zapišemo, da je c naslednik kraja od na naši poti, kraj od pa je predhodnik kraja pred.

Ko pridemo do konca vhodnih podatkov, poiščemo s pomočjo teh dveh tabel začetni kraj celotne poti: to je tisti, ki ima na poti naslednika, nima pa predhodnika. Od tam naprej lahko s tabelo nasl sledimo še preostanku poti in jo izpišemo.

```
#include <stdio.h>

int main()
{
  int pred[26], nasl[26], od = −1, c;
  for (c = 0; c < 26; c++) pred[c] = −1, nasl[c] = −1;
  /* Berimo navodila po znakih. */
  while ((c = fgetc(stdin)) != EOF)
  {
    /* Znake, ki niso velike črke, preskočimo. */
    if (c < 'A' || c > 'Z') continue;
    c −= 'A';
    /* Če je „od“ še enak −1, to pomeni, da je naslednja velika črka,
       ki jo bomo prebrali, ime začetnega kraja v trenutni vrstici. */
    if (od == −1) od = c;
    else {
      /* Sicer pa je naslednja prebrana velika črka ime končnega kraja v trenutni
         vrstici. Zdaj si lahko ta korak zapomnimo v tabelah pred in nasl. */
```

```
        pred[c] = od; nasl[od] = c;
        /* Naslednja velika črka bo spet začetni kraj, zato postavimo „od“ nazaj na −1. */
        od = −1; }
    }
    /* Poiščimo začetek poti, torej kraj, ki ima naslednika, nima pa predhodnika. */
    for (c = 0; c < 26 && ! (pred[c] < 0 && nasl[c] >= 0); c++) ;
    /* Izpišimo potek poti. */
    while (c != −1)
    {
      printf("%c", c + 'A'); /* Izpišimo trenutni kraj... */
      c = nasl[c];           /* ... in se premaknimo na naslednjega. */
    }
    return 0;
}
```

4. Delni izid

Označimo vhodno zaporedje košev z $a_1, \ldots, a_n$. Delni izid zdaj ni nič drugega kot vsota oblike $a_i + a_{i+1} + \ldots + a_{j-1} + a_j$ za neka i in j z območja $1 \le i \le j \le n$. Če je ta vsota pozitivna, je to delni izid v prid prve ekipe, če je negativna, pa imamo delni izid v prid druge ekipe. Preprosta rešitev je torej ta, da z dvema gnezdenima zankama preizkusimo vse možne kombinacije i in j, nato pa pri vsaki od njih s še eno zanko izračunamo pripadajočo delno vsoto in preverimo, če je najboljša doslej (za eno ali drugo ekipo):

```
naj1 := 0; naj2 := 0;
for i := 1 to n:
  for j := i to n:
    vsota := 0;
    for k := i to j:
      vsota := vsota + ak;
    if vsota > naj1 then naj1 := vsota
    else if −vsota > naj2 then naj2 := −vsota;
return naj1, naj2;
```

Časovna zahtevnost tega postopka je kar $O(n^3)$, saj imamo tri gnezdene zanke, ki izvedejo v najslabšem primeru po $O(n)$ iteracij. Rešitev lahko izboljšamo z naslednjim opažanjem: če se pri nespremenjenem i premaknemo z j na $j + 1$, vsebuje nova delna vsota iste seštevance kot prejšnja, le člen a_{j+1} se v njej pojavi na novo. Torej nove delne vsote ni treba računati čisto od začetka (z zanko po k), ampak lahko le prištejemo novi člen k stari delni vsoti:

```
naj1 := 0; naj2 := 0;
for i := 1 to n:
  vsota := 0;
  for j := i to n:
    vsota := vsota + aj;
    if vsota > naj1 then naj1 := vsota
    else if −vsota > naj2 then naj2 := −vsota;
return naj1, naj2;
```

Tako imamo le še dve gnezdeni zanki in časovno zahtevnost postopka smo zmanjšali na $O(n^2)$. Še boljša rešitev pa je naslednja: predstavljajmo si delne izide od začetka tekme, torej vsote oblike $s_j := a_1 + a_2 + \ldots + a_{j-1} + a_j$. (Pri $j = 0$ si mislimo $s_0 = 0$.) Delni izid od i do j (torej vsota $a_i + \ldots + a_j$) je potem ravno enak razliki $s_j - s_{i-1}$. Če gledamo delne izide, ki se končajo pri j, je torej člen s_j v tej razliki pri vseh enak, z izbiro i-ja lahko vplivamo le na člen s_{i-1}. Če torej hočemo, da bo delni izid čimbolj ugoden za prvo ekipo, moramo vzeti tak i (med $1 \le i \le j$), pri katerem je s_{i-1} najmanjša; tedaj bo razlika $s_j - s_{i-1}$ najvišja. Podobno pa, če hočemo najboljši delni izid za drugo ekipo, moramo izbrati i tako, da bo s_{i-1} največja; tedaj bo razlika $s_j - s_{i-1}$ najnižja. Koristno si je torej, medtem ko se premikamo naprej v zanki po j, zapomniti najnižjo in najvišjo izmed vseh dosedanjih delnih vsot $s_1, \ldots, s_{j-1}$:

$naj_1 := 0$; $naj_2 := 0$;
$vsota := 0$; $minVsota := 0$; $maxVsota := 0$;
for $j := 1$ **to** n:
  $vsota := vsota + a_j$;
  if $vsota - minVsota > naj_1$ **then** $naj_1 := vsota - minVsota$;
  if $maxVsota - vsota > naj_2$ **then** $naj_2 := maxVsota - vsota$;
  if $vsota > maxVsota$ **then** $maxVsota := vsota$
  else if $vsota < minVsota$ **then** $minVsota := vsota$;
return naj_1, naj_2;

Tu imamo le eno zanko z n iteracijami, v vsaki iteraciji pa konstantno mnogo dela, tako da je časovna zahtevnost te rešitve le še $O(n)$.

Za konec omenimo še eno podrobnost: dosedanje rešitve so na začetku inicializirale naj_1 in naj_2 na 0 in ju odtlej le še povečevale. To pomeni, da če na primer dobimo tekmo, v kateri ena od ekip ni dala nobenega koša, bodo naše rešitve kot najugodnejši delni izid za to ekipo vrnile rezultat 0, kar ustreza poljubnemu praznemu podzaporedju oz. z drugimi besedami: delni vsoti z 0 seštevanci. Besedilo naloge pravzaprav ne pove natančno, ali so takšni delni izidi dovoljeni ali pa nas zanimajo le tisti delni izidi, ki obsegajo vsaj en koš. Če se odločimo za to drugo interpretacijo, je bolje inicializirati naj_1 in naj_2 na -4; tako se bo vrednost naj_x za tisto ekipo, ki ne doseže nobenega koša, sčasoma ustalila pri -3 (če je nasprotna ekipa dosegla same trojke) ali -2.

5. Razporejanje študentov

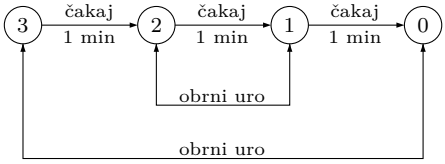
Pri tej nalogi je algoritem natančno predpisan že v besedilu naloge, mi ga moramo le še implementirati. Ogledali si bomo primer implementacije v C++. Ko pregledujemo študente in jih razporejamo v učilnice, je koristno za vsako učilnico vzdrževati seznam študentov, ki smo jih že razporedili vanjo. Iz dolžine tega seznama bomo tudi videli, ali je učilnica že polna ali lahko vanjo pošljemo še kakšnega študenta. Poleg tega bomo imeli še en podoben seznam za nerazporejene študente (tiste, ki se jih ni dalo razporediti v nobeno učilnico). Na koncu moramo vse te sezname le še izpisati. Za predstavitev seznamov lahko uporabimo na primer razred `vector` iz standardne knjižnice.

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

int main()
{
  // Preberimo n in k.
  int n, k; cin >> n >> k;
  vector<vector<int>> razpored { k };
  vector<int> nerazporejeni, kapaciteta { k };
  // Preberimo kapacitete.
  for (int u = 0; u < k; u++) cin >> kapaciteta[u];
  // Obdelajmo vse študente.
  for (int student = 1; student <= n; student++)
  {
    int d; cin >> d;
    bool razporejen = false;
    for (int i = 0; i < d; i++)
    {
      // Preberimo naslednjo preferenco tega študenta.
      int u; cin >> u; u--;
      if (razporejen) continue; // Je bil že razporejen?
      // Ali je v tej učilnici še kaj prostora?
      if (razpored[u].size() >= kapaciteta[u]) continue;
      // Razporedimo ga v to učilnico.
      razpored[u].push_back(student); razporejen = true;
    }
    // Nerazporejene študente dodajmo na ustrezni seznam.
    if (! razporejen) nerazporejeni.push_back(student);
  }
  // Izpišimo rezultate.
  for (int u = 0; u <= k; u++)
  {
    if (u < k) cout << "Učilnica " << (u + 1) << ":";
    else cout << "Nerazporejeni:";
    for (int student : (u < k) ? razpored[u] : nerazporejeni) cout << " " << student;
    cout << endl;
  }
  return 0;
}
```

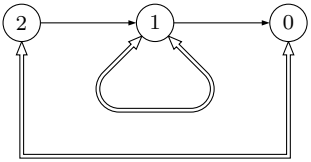
6. Peščena ura

(a) Stanje ure lahko opišemo s številom od 0 do 3, ki pove, koliko peska je v zgornji polovici ure. Na začetku je torej ura v stanju 0, pa tudi na koncu mora biti v stanju 0. Zanimajo nas le celoštevilska stanja, saj lahko le takrat obračamo uro. Možni so torej naslednji premiki med stanji:



Iščemo torej tak obhod po tem grafu stanj, ki se začne in konča v točki 0, trikrat uporabi povezave z oznako „čakaj 1 min“ in čim manjkrat uporabi povezave z oznako „obrni uro“.

Nobene koristi ni od tega, da bi uro obrnili dvakrat zaporedoma, ne da bi vmes počakali vsaj eno minuto. Zato lahko gornji graf predelamo tako, da namesto prehodov „obrni uro“ uporabimo prehode tipa „obrni uro in počakaj 1 minuto“. S takim prehodom na primer pridemo iz stanja 0 naravnost v 2 (ko uro obrnemo, iz 0 nastane 3, nato pa počakamo še eno minuto in ura pride v stanje 2).



Stanja 3 zdaj ne potrebujemo več, saj vanj ne moremo priti (ker vsak prehod vključuje tudi eno minuto čakanja po morebitnem obratu, po tej minuti pa ura gotovo ni več v stanju 3). V tem novem grafu traja vsak prehod eno minuto, pred tem pa imajo nekateri prehodi še obrat (ti so narisani z debelimi puščicami), ostali pa ne. Zdaj torej lahko rečemo, da nas zanima v tem grafu obhod, ki se začne in konča v točki 0, naredi t korakov in od tega čim manj takih po debelih puščicah.

Take obhode lahko sicer iščemo ročno, saj je graf majhen in nas zanima le do $t = 10$; lepše pa je, če naredimo to z algoritmom, saj smo vendarle na računalniškem temovanju, pa še pri splošnejši različici naloge (za poljubne c in t) bo prišel prav. V naslednjem razmisleku bomo povezave, ki vključujejo obrat, označevali z dvojno puščico $\Rightarrow$, ostale pa z navadno $\to$. Pri stanju pa bomo spodaj označili še število obratov, ki smo jih porabili, da smo prišli do njega; tako naj na primer 1_3 pomeni, da smo v stanju 1 in da smo do njega prišli s tremi obrati ure.

Na začetku smo torej v stanju 0_0 in za prvi korak nimamo druge možnosti kot $0_0 \Rightarrow 2_1$. Po enem koraku smo torej v stanju 2 (in smo izvedli en obrat). Tu imamo dve možnosti za nadaljevanje: lahko gremo po $2_1 \to 1_1$ ali pa po $2_1 \Rightarrow 0_2$; tako torej po dveh korakih lahko pristanemo v stanju 1 (in smo izvedli en obrat) ali pa v stanju 0 (in smo izvedli dva obrata). Nato, v tretjem koraku, imamo še več možnosti nadaljevanja: $1_1 \Rightarrow 1_2$, pa $1_1 \to 0_1$ in še $0_2 \Rightarrow 2_3$.

Oglejmo si še četrti korak: iz 1_2 lahko nadaljujemo po $1_2 \Rightarrow 1_3$ ali po $1_2 \to 0_2$; iz 0_1 gremo lahko le po $0_1 \Rightarrow 2_2$; iz 2_3 pa gremo lahko po $2_3 \Rightarrow 0_4$ ali po $2_3 \to 1_3$. Po četrtem koraku smo torej lahko v stanju 0 (z dvema ali pa s štirimi obrati) ali v stanju 1 (s tremi obrati) ali v stanju 2 (z dvema obratoma). Ker nas vedno zanimajo le obhodi s čim manj obračanji ure, si bomo pri stanju 0 zapomnili le možnost z dvema obratoma, tisto s štirimi obrati pa lahko odmislimo. Za naš algoritem je torej dovolj, če si pri vsakem stanju a zabeleži le to, s koliko najmanj obrati ga je mogoče v določenem številu korakov t doseči (če ga je sploh mogoče doseči). Temu številu recimo $f[t, a]$. Če stanja a v natanko t korakih ni mogoče doseči, si mislimo $f[t, a] = \infty$.

Zdaj že vemo dovolj, da lahko zapišemo naš postopek; namesto za 3-minutno uro ga zapišimo kar v splošni obliki, za c-minutno uro, tako da bo prišel prav tudi pri reševanju podnaloge (b). Učinek obračanja ure je zdaj ta, da se iz stanja a

premakne v $c-a$, ker pa smo rekli, da v vsak prehod vključimo tudi enominutno čakanje, imamo tako prehode oblike $a \Rightarrow (c-1-a)$.

$f[0,0] := 0$; **for** $a := 1$ **to** $c-1$ **do** $f[0,a] := \infty$;
for $t := 0$ **to** $T-1$:
  (* *Zdaj vemo, kaj je možno doseči po t korakih; izračunajmo možna stanja po $t+1$ korakih. Najprej inicializirajmo naslednjo vrstico tabele.* *)
  for $a := 0$ **to** $c-1$ **do** $f[t+1,a] := \infty$;
  (* *Upoštevajmo povezave brez obratov, torej $a \to (a-1)$.* *)
  for $a := 1$ **to** $c-1$ **do**
    $f[t+1,a-1] := \min\{f[t+1,a-1], f[t,a]\}$;
  (* *Upoštevajmo povezave z obrati, torej $a \Rightarrow (c-1-a)$.* *)
  for $a := 0$ **to** $c-1$ **do**
    $f[t+1,c-1-a] := \min\{f[t+1,c-1-a], f[t,a]+1\}$;

Še bolj elegantno lahko isti postopek zapišemo tako, da upoštevamo, da je mogoče v stanje a priti bodisi iz $a+1$ (s povezavo $\to$) ali iz $c-1-a$ (s povezavo $\Rightarrow$), tako da moramo le pogledati, katera od teh dveh možnosti zahteva najmanj obratov. Izjema pri tem je le $a = c-1$, do katerega ne moremo priti iz stanja $a+1$, ker tega v našem grafu ni.

$f[0,0] := 0$; **for** $a := 1$ **to** $c-1$ **do** $f[0,a] := \infty$;
for $t := 1$ **to** T:
  $f[t,c-1] := f[t-1,0]+1$;
  for $a := 0$ **to** $c-2$:
    $f[t,a] := \min\{f[t-1,a+1], f[t-1,c-1-a]+1\}$;

Ob koncu tega postopka lahko iz $f[0,t]$ odčitamo (za vsak čas $t = 0,\ldots,T$), koliko obračanj ure potrebujemo, da z našo uro odmerimo t minut. Če bi si poleg tega ob računanju $\min\{\ldots\}$ še zapomnili, katera od obeh možnosti je res dala minimum, bi lahko iz teh podatkov tudi rekonstruirali potek obhoda, torej to, kdaj smo uporabili povezave $\Rightarrow$ z obračanjem ure, kdaj pa navadne povezave $\to$.

Za preprosto trominutno uro, o kateri govori naša podnaloga (a), so rezultati takšni:

t	1	2	3	4	5	6	7	8	9	10	11	12	13
$f[t,0]$	∞	2	1	2	3	2	3	4	3	4	5	4	5

Ene same minute ($t = 1$) z našo uro ne moremo odmeriti; $t = 2$ minuti lahko odmerimo preprosto tako, da uro obrnemo, počakamo 1 minuto, jo obrnemo še enkrat in počakamo še eno minuto, da se izteče. Za $t = 3$ moramo uro le obrniti in počakati, da se izteče; za $t = 4$ pa jo obrnemo, počakamo dve minuti, obrnemo še enkrat in počakamo, da se izteče. Daljše čase pa odmerimo tako, da najprej odmerimo nekajkrat po 3 minute, dokler nam ne ostanejo le še 4 minute ali manj, to pa odmerimo po malo prej opisanih načinih.

(b) Postopek, ki smo ga zapisali zgoraj pri podnalogi (a), že deluje za poljubno velikost ure c in čas t, ki bi ga radi odmerili. Označimo z $g(t,c)$ najmanjše potrebno število obračanj ure, s katerimi lahko odmerimo t minut. Če malo poganjamo zgoraj opisani postopek za različne t in c ter si ogledamo, kakšne vrednosti $g(t,c)$ dobimo,

lahko rezultate povzamemo v naslednji tabeli. Pri tem bomo pisali $t = q \cdot c + r$ za $0 \le r < c$ (torej je r ostanek po deljenju t s c, celi del količnika pa je q).

	če je...	...potem je $g(t,c)$ (za $t = q \cdot c + r$)
(1)	$r = 0$	q
(2)	c lih, r lih, $q = 0$	ni rešitve
(3)	c lih, r lih, $q > 0$	$q + 1$
(4)	c lih, r sod in > 0	$q + 2$
(5)	c sod, r lih	ni rešitve
(6)	c sod, r sod in > 0, $q = 0$	2
(7)	c sod, r sod in > 0, $q > 0$	$q + 1$

Prepričajmo se, da so rešitve v tej tabeli res optimalne, torej da z manj kot toliko obračanji c-minutne peščene ure ni mogoče odmeriti t minut. Spomnimo se, da pri c-minutni peščeni uri naš graf stanj tvorijo točke (oz. stanja) od 0 do $c-1$ in da za vsak u obstaja „draga“ povezava $u \Rightarrow (c-1-u)$, za vsak $u > 0$ pa tudi „poceni“ povezava $u \to (u-1)$. Cena dragih povezav je 1, poceni povezav pa 0; naš problem odmerjanja t minut časa je enakovreden problemu, kako sestaviti v grafu čim cenejši obhod dolžine t z začetkom in koncem v točki 0.

Opazimo lahko, da naš obhod ne more imeti nikoli več kot $c-1$ zaporednih poceni povezav, saj se z vsako poceni povezavo premaknemo v stanje, ki je za 1 nižje od prejšnjega, začeti pa ne moremo višje kot pri $c-1$, tako da po največ $c-1$ korakih pridemo v stanje 0, iz katerega je mogoče nadaljevati le po dragi povezavi.

Ker se naš obhod začne v stanju 0, to tudi pomeni, da je prvi korak obhoda nujno po dragi povezavi. Če ima obhod ceno k, to pomeni, da vsebuje k dragih korakov, za vsakim od teh pa lahko pride še največ $c-1$ poceni korakov, tako da ima obhod s ceno k lahko dolžino največ $k \cdot c$. Ker nas zanima obhod dolžine $t = q \cdot c + r$, to pomeni, da mora biti njegova cena vsaj q, če je $r = 0$, in vsaj $q+1$, če je $r > 0$.

(1) Če je $r = 0$ in zato $t = q \cdot r$, smo ravnokar videli, da ima obhod gotovo ceno vsaj q. Obhod s to ceno pa tudi res obstaja: ni treba drugega, kot da q-krat izvedemo cikel $0 \Rightarrow (c-1) \to (c-2) \to \ldots \to 1 \to 0$. Torej je v tem primeru $g(t,c)$ res enako q.

(2) Če je c lih, r lih in $q = 0$: tu je torej $t = r$ za nek $0 < r < c$. Če poženemo postopek iz rešitve podnaloge (a) in si ogledamo, kaj se pri njem dogaja v tabeli f, lahko opazimo, da so iz stanja 0 v t korakih dosegljiva natanko stanja iz množice $A_t \cup B_t$ (in nobena druga), pri čemer sta A_t in B_t definirana takole:

$$A_t = \{u \,:\, u < t \text{ in } u \text{ je enake parnosti kot } t\}$$
$$B_t = \{u \,:\, u \ge c - t \text{ in } u \text{ je nasprotne parnosti kot } t\}.$$

(O tem, da to res drži za vsak t od 0 do $c-1$, se lahko prepričamo z indukcijo po t; podrobnosti prepustimo bralcu za vajo.) Nas seveda zanima obhod, ki se začne in konča v 0, torej nas zanima, ali je $0 \in A_t \cup B_t$ ali ne. Ker je $t < c$, pogoj $0 \ge c - t$ ni izpolnjen, torej 0 gotovo ni v B_t; če naj bo torej v $A_t \cup B_t$, je to mogoče le tako, de v A_t. In iz definicije A_t vidimo, da je 0 v tej množici natanko tedaj, ko je enake parnosti kot t, torej ko je t sod. Mi pa imamo $t = r$ za lih r, torej obhoda dolžine t res ne bo mogoče sestaviti.

(3) Če je c lih, r lih in $q > 0$: že zgoraj smo videli, da obhod gotovo ne more biti cenejši od $q+1$. Pokažimo, da obhod s ceno točno $q+1$ res obstaja. (3.1) Recimo najprej, da je $q = 1$, torej naš $t = c + r$ leži na območju $c < t < 2c$. Ker sta r in c liha, je $t = c + r$ sod. Mislimo si obhod, ki začne z dragim korakom $0 \Rightarrow (c-1)$ in nato naredi še $c-1$ poceni korakov, kar ga pripelje nazaj v 0. Ta obhod je torej dolg c korakov. Radi bi ga podaljšali še za r korakov. Ker je c lih, pišimo $c = 2s+1$. Dragi koraki so oblike $u \Leftrightarrow (c-1-u) = (2s-u)$. To na primer pomeni, da imamo drage korake $s \Leftrightarrow s$, pa $(s-1) \Leftrightarrow (s+1)$, pa $(s-2) \Leftrightarrow (s+2)$ in tako naprej, vse do $0 \Leftrightarrow 2s = (c-1)$. Recimo, da ko naš prvotni obhod pride v točko $s-a$ (za nek a izmed $0, \ldots, s$), vrinemo tja zdaj drag korak v $s+a$, nato pa nadaljujemo s poceni koraki vse do točke 0. Naš obhod se je tako podaljšal za $2a+1$ korakov (enega dragega in še $2a$ poceni korakov), skupaj pa ima zdaj dva draga koraka. Njegova cena je torej 2, dolžina pa je skupaj $c + 2a + 1$. Ker je r lih, si lahko a izberemo tako, da je $2a + 1 = r$, dolžina našega obhoda je tedaj $c + r = t$, cena pa $2 = q + 1$, ravno tak obhod pa smo iskali.

(3.2) Če pa je $q > 1$, lahko najprej naredimo $q-1$ obhodov $0 \Rightarrow (c-1) \to \ldots \to 1 \to 0$, ki imajo ceno 1 in dolžino c, nato pa uporabimo rešitev za $q = 1$ (in nespremenjen r) iz prejšnjega odstavka. Tako pridemo na obhod dolžine r s ceno $(q-1)+2 = q+1$, ravno tak obhod pa smo iskali.

(4) Če je c lih, r sod (in > 0): zgoraj smo že videli, da obhod s ceno manj kot $q+1$ tu ne more obstajati. Kaj pa obhod s ceno natanko $q+1$? Ker je c lih, nam drag korak $u \Rightarrow (c-1-u)$ ohrani parnost našega stanja (število $c-1-u$ ima enako parnost kot u); po drugi strani pa poceni korak $u \to (u-1)$ seveda nujno spremeni parnost našega stanja. Spomnimo se zdaj, da je $t = qc + r$ in da je c lih, r pa sod; ker je r sod, ima qc enako parnost kot t; in ker je c lih, ima q enako parnost kot qc in s tem tudi kot t. Če je skupna cena obhoda k, to pomeni, da smo imeli k dragih povezav; in ker je skupna dolžina obhoda t, smo morali imeti $t-k$ poceni povezav. Poceni povezava nam spremeni parnost stanja, draga pa ne; ker je na koncu obhoda stanje enako kot na začetku, smo morali imeti poceni povezav sodo mnogo; torej je $t-k$ sod, torej je k enake parnosti kot t in zato tudi kot q. Torej je nemogoče, da bi bil $k = q+1$.

Pač pa obstaja obhod s ceno $q+2$. (4.1) Oglejmo si najprej primer, ko je $q = 0$, torej je $t = r$ eno od števil $2, 4, \ldots, c-1$. Ker je t sod, pišimo $t = 2u$. Začnimo naš obhod z dragim korakom $0 \Rightarrow (c-1)$ (saj drugače niti ne moremo); nato naredimo $u-1$ poceni korakov, kar nas pripelje do stanja $c-u$. Tu naredimo spet drag korak, s čimer pridemo v stanje $(c-1)-(c-u) = u-1$. Nato naredimo še $u-1$ poceni korakov, s čimer pridemo spet v stanje 0. Tako imamo obhod s ceno $2 = q+2$ in dolžino $1 + (u-1) + 1 + (u-1) = t$, ravno takega pa smo iskali.

(4.2) Pri $q > 0$ naredimo najprej q ciklov $0 \Rightarrow (c-1) \to \ldots \to 1 \to 0$ (cena 1, dolžina c) nato pa dodajmo še rešitev za $q = 0$ (pri nespremenjenem r) iz prejšnjega odstavka. Tako dobimo obhod dolžine $q \cdot c + r = t$ s ceno $q \cdot 1 + 2 = q + 2$, prav takega pa smo iskali.

(5) Če je c sod in r lih: ker je c sod, nam drag korak $u \Rightarrow (c-1-u)$ spremeni parnost našega stanja (število $c-1-u$ ima nasprotno parnost kot u), ravno tako pa jo seveda spremeni tudi poceni korak $u \to (u-1)$. Torej vsak korak obrne parnost stanja. Ker se mora obhod končati v istem stanju, v katerem se je začel, to pomeni,

da mora biti sode dolžine. Toda nas zanima obhod dolžine t, to pa je liho število (ker je $t = q \cdot c + r$ in ker je c sod, r pa lih); torej takega obhoda sploh ni.

(6) Če c sod, r sod (in > 0) in $q = 0$: tu je torej $t = r$ eden od $2, 4, \ldots, c-2$. Obhod z 0 dragimi povezavami ima lahko le dolžino 0; obhod z 1 drago povezavo ima lahko le dolžino c; torej naš problem gotovo ni rešljiv z manj kot dvema dragima povezavama. Z natanko dvema pa je res rešljiv: $t = r$ je sod (ker je r sod), torej recimo $t = 2u$, pa naredimo tak obhod: najprej $0 \Rightarrow (c-1)$ po dragi povezavi; nato $u-1$ poceni povezav, kar nas pripelje v $c-u$; nato od tam v $u-1$ z drago povezavo, nato pa še $u-1$ poceni povezav, kar zaključi obhod s skupno ceno 2. (To je prav tak obhod kot v točki (4.1).)

(7) Če je c sod, r sod (in > 0) in $q > 0$: že na začetku smo videli, da obhoda s ceno manj kot $q+1$ ni. Prepričajmo se, da obhod s ceno natanko $q+1$ obstaja. (7.1) Najprej si oglejmo primer, ko je $q = 1$. Torej je $t = c + r$, pri čemer sta tako c kot r soda in $0 < r < c$. Začnimo z obhodom $0 \Rightarrow (c-1) \to \ldots \to 1 \to 0$ (dolžina c, cena 1). Zdaj bi ga radi podaljšali še za r korakov. Ker je c sod, pišimo $c = 2s$. Drage povezave so oblike $u \Leftrightarrow (c-1-u)$, torej na primer $s \Leftrightarrow (s-1)$, pa $(s+1) \Leftrightarrow (s-2)$, pa $(s+2) \Leftrightarrow (s-3)$ in tako naprej do $s+(s-1) \Leftrightarrow (u-1)-(u-1)$, kar je ravno $(c-1) \Leftrightarrow 0$. V splošnem nas torej draga povezava $(s-k) \Rightarrow (s+k+1)$ postavi za $2k+1$ korakov nazaj, zato nam podaljša obhod za $2k+2$ korakov (en drag in $2k+1$ poceni korakov). Ker je naš r sod, moramo torej vzeti $k = (r/2) - 1$ in obhod podaljšati na ta način, pa bo dolg ravno $c + r = t$ (in imel bo dve dragi povezavi, torej ima ceno $q+1$, kar smo tudi želeli).

(7.2) Pri $q > 1$ pa za začetek $(q-1)$-krat izvedimo cikel $0 \Rightarrow (c-1) \to \ldots 1 \to 0$ (dolžina c, cena 1), nato pa dodajmo še obhod dolžine $c+r$ iz prejšnjega odstavka. Skupaj imamo obhod dolžine $(q-1) \cdot c + (c+r) = qc + r = t$ s ceno $(q-1)+2 = q+1$, prav takega pa smo tudi iskali.

7. Polaganje plošč

Naloga pravi, da mora zgornji levi kot plošče ležati na hipotenuzi našega pravokotnega trikotnika; s tem je položaj posamezne plošče čisto enolično določen: plošča višine h_i se mora začeti (imeti svoj levi rob) pri x-koordinati $l_i := a \cdot h_i / b$, konča pa se (oz. ima svoj desni rob) potem pač pri x-koordinati $d_i := l_i + w_i$.

(a) Uredimo plošče po naraščajočem d_i. Zastavimo si podproblem: naj bo $f(i)$ največja možna skupna pokrita površina, če smemo uporabljati le plošče od 1 do i, ne pa tudi plošč od $i+1$ do n. Ena možnost, ko računamo $f(i)$, je ta, da plošče i sploh ne uporabimo; v tem primeru je največja možna skupna pokrita površina kar enaka $f(i-1)$. Druga možnost pa je, da ploščo i vendarle uporabimo (ta možnost sicer pride v poštev le, če je $d_i \leq b$, saj bi drugače plošča na desni strani štrlela iz našega pravokotnega trikotnika). V tem primeru lahko levo od nje uporabimo le take plošče j, katerih desni rob leži levo od levega roba plošče i, torej ki imajo $d_j \leq l_i$. Ker so plošče že urejene po desnem robu, moramo torej le poiskati zadnji (največji) tak j, pri katerem še velja $d_j \leq l_i$; za tisti j vemo, da smemo pred ploščo i uporabljati plošče od 1 do j, ne pa tudi plošč od $j+1$ do $i-1$ (kajti te bi se prekrivale s ploščo i). Največja površina, ki jo lahko s temi ploščami (od 1 do j) pokrijemo, pa je seveda $f(j)$; k njej moramo nato prišteti še površimo i-te plošče, torej $w_i \cdot h_i$. Tako smo dobili naslednji postopek:

uredi plošče po koordinati desnega roba in jih v tem vrstnem redu
  oštevilči od 1 do n;
$f[0] := 0$;
for $i := 1$ **to** n:
  $f[i] := f[i-1]$;
  if $d_i > b$ **then continue**;
  z bisekcijo poišči največji tak j z območja $0, ..., i-1$, pri katerem
    še velja $d_j \leq l_i$ (pri $j = 0$ si mislimo $d_j = 0$);
  $f[i] := \max\{f[i], f[j] + w_i \cdot h_i\}$;
return $f[n]$;

Časovna zahtevnost tega postopka je $O(n \log n)$, saj ta čas zadošča tako za urejanje ploščna začetku kot tudi za izvajanje bisekcije v vsaki iteraciji glavne zanke.

(b) Vsako ploščo lahko zdaj v mislih podvojimo in eno od kopij zavrtimo za 90 stopinj: kjer smo prej imeli ploščo širine w_i in višine h_i, dodajmo zdaj še eno širine h_i in višine w_i. Na tako podvojenem seznamu plošč poženimo zdaj postopek za rešitev podnaloge (a), torej tiste, pri kateri se plošč ne sme obračati. Očitno je, da vsaki dopustni rešitvi podnaloge (b) — torej vsakemu takemu izboru plošč, pri katerem so nekatere plošče mogoče zasukane za 90°, obenem pa se nobeni dve plošči ne prekrivata — ustreza tudi neka dopustna rešitev podnaloge (a). Vprašanje je le, ali velja tudi obratno; če je to res, potem bo najboljša rešitev, ki jo poišče naš postopek za podnalogo (a), dala tudi odgovor, po katerem sprašuje podnaloga (b).

Zahteve podnaloge (a) že poskrbijo za to, da se plošče ne prekrivajo; in vsaka plošča v našem podvojenem seznamu ustreza eni od plošč prvotnega seznama, mogoče zavrteni za 90°. Edini način, da bi bila lahko neka dopustna rešitev podnaloge (a) neustrezna z vidika podnaloge (b), bi bila ta, da bi uporabila oba izvoda iste plošče, torej $w_i \times h_i$ in hkrati še $h_i \times w_i$ za eno in isto ploščo i iz podnaloge (b).

Prepričajmo se, da do te težave ne more priti. Pišimo $c = a/b$ (to razmerje je gotovo ≤ 1, saj naloga pravi, da je $a \leq b$). Če neko ploščo postavimo tako, da ima višino h in širino w, se mora začeti pri $x = ch$, konča pa se torej pri $x = ch + w$; če pa jo zasukamo, tako da ima višino w in širino h, se mora začeti pri $x = cw$, konča pa se torej pri $x = cw + h$. V težave pridemo le, če se tadva intervala, $[ch, ch + w]$ in $[cw, cw + h]$, ne prekrivata — takrat bi se namreč lahko zgodilo, da bi algoritem, ki rešuje podnalogo (a), hkrati uporabil obe kopiji te plošče. To, da se omenjena intervala ne prekrivata, se lahko zgodi na dva načina: bodisi prvi leži levo od drugega (kar se zgodi pri $ch + w \leq cw$) ali pa drugi leži levo od prvega (to pa se zgodi pri $cw + h \leq ch$). Prvi pogoj lahko predelamo v $ch \leq (c-1)w$, drugega pa v $cw \leq (c-1)h$; v obeh primerih je leva stran pozitivna, desna pa gotovo ne (ker je $c \leq 1$), torej nobeden od teh dveh pogojev ni izpolnjen. Tako lahko zaključimo, da bo rešitev, ki jo najde na podvojenem seznamu postopek iz podnaloge (a), veljavna tudi s stališča podnaloge (b).

8. Kodiranje

(a) Omejitve, ki jih podaja besedilo naloge, lahko predstavimo z grafom. V njem naj bo za vsako možno peterico po ena točka; tako dobimo množico točk $V = \{0, 1\}^5$. Med točkama u in v naj obstaja (neusmerjena) povezava natanko tedaj, ko peteric u in v ne moremo uporabiti obeh hkrati — torej če eno od njiju uporabimo kot

kodo neke števke, potem druge ne smemo uporabiti kot kode neke druge števke, ker bi lahko pri dekodiranju prišlo do napake, ki je prejemnik ne bi mogel zaznati. To se zgodi, če je $u \in N(v)$ (takrat prejemnik, če prejme u, ne more vedeti, ali je bil poslan v in ni bilo napake ali pa je bil poslan u in se je pri prenosu zaradi napake spremenil v v) ali pa $v \in N(u)$ (takrat ima prejemnik podobno težavo, če prejme v).

Za nadaljevanje rešitve bo koristno imeti graf predstavljen tako, da za vsako točko u hranimo množico njenih sosed $S(u)$, torej tistih točk, ki so neposredno povezane z njo. Pripravimo jih lahko tako, da se sprehodimo po množicah $N(u)$ in upoštevamo, da če je $v \in N(u)$, to pomeni, da morata biti točki u in v sosedi druga druge:

$E := \{\}$; **for** $u \in V$ **do** $S(u) := \{\}$;
for $u \in V$ **do for** $v \in N(u)$ **do** dodaj v v $S(u)$ in dodaj u v $S(v)$;

Naloga zdaj pravzaprav zahteva, da v tem grafu poiščemo množico takih desetih točk, da nobeni dve nista neposredno povezani s povezavo. Taki množici se v teoriji grafov reče *neodvisna množica*; iščemo torej neodvisno množico velikosti 10. V splošnem je iskanje največje neodvisne množice NP-težak problem; ker pa je naš graf majhen, lahko neodvisne množice dovolj hitro iščemo že s preprostim rekurzivnim algoritmom. Začeli bomo s prazno neodvisno množico A in vanjo dodajali točke eno po eno; pri tem vzdržujmo množico kandidatk C, torej točk, ki bi se jih še dalo dodati v trenutno neodvisno množico (in bi ta pri tem ostala neodvisna). Ko neko točko u dodamo v neodvisno množico, pa moramo njo in vse njene sosede (množico u-jevih sosed označimo s $S(u)$) v grafu pobrisati iz množice kandidatk. Če imamo v nekem trenutku na voljo več kandidatk, moramo z rekurzivnimi klici preizkusiti vsako od njih.

Koristno je imeti v mislih še naslednjo podrobnost: do iste množice A lahko načeloma pri rekurziji pridemo po večkrat, namreč po enkrat za vsak možni vrstni red, v katerem je mogoče dodati elemente v množico A. Če bi se po večkrat ukvarjali z eno in isto množico A, bi le po nepotrebnem zapravljali čas. Preprost način, s katerim se lahko temu izognemo, je ta, da se dogovorimo, da bomo kode dodajali v množico le v naraščajočem vrstnem redu. (Ker so kode v bistvu peterice bitov, si jih lahko predstavljamo kot cela števila od 0 do 31.) Spodnja rešitev zato, ko računa novo množico kandidatk C' po dodajanju kode u v množico A, pobriše iz dosedanje C ne le u-jeve sosede, pač pa tudi morebitne kandidatke v, za katere je $v \leq u$.

podprogram Rekurzija(dosedanja množica A, kandidatke C):
  if $|A| \geq 10$ **then**
    našli smo primerno neodvisno množico A, izpišimo jo in končajmo;
  else for $u \in C$ **do**
    $C' := C - \{v \in V : v \leq u\} - S(u)$;
    Rekurzija($A \cup \{u\}$, C');

Postopek začnemo tako, da pokličemo Rekurzija($\{\}, V$). Ker je vseh možnih peteric le 32, je pri implementaciji tega postopka koristno, če množice predstavimo kar kot 32-bitna cela števila, pri čemer vsak bit za eno od možnih peteric pove, ali pripada tej množici ali ne. Tako lahko za računanje unije $A \cup \{u\}$ in razlike

$C - \{v \in V : v \leq u\} - S(u)$ uporabimo kar preproste binarne operacije na celoštevilskih spremenljivkah.

(b) To podnalogo lahko rešujemo enako kot (a), spremeni se le definicija sosednosti v grafu. Zdaj velja, da moramo dodati povezavo med u in v natanko tedaj, ko imata $N(u)$ in $N(v)$ neprazen presek — tedaj se namreč lahko tako u kot v pri prenosu spremenita v isto peterico (ki je sicer mogoče različna tako od u kot od v), zato prejemnik ne bi mogel zanesljivo ugotoviti, ali mu je pošiljatelj poslal u ali v. Množica povezav našega grafa je torej $E = \{(u, v) \ : \ N(u) \cap N(v) \neq \emptyset\}$.

Povezave oz. množice sosed lahko učinkovito pripravimo takole: najprej si za vsako v pripravimo „kazalo“, to je množico $K(v)$ vseh tistih peteric u, pri katerih se v pojavlja v $N(u)$. Nato moramo le še za vsako $K(v)$ pregledati vse pare točk u, u' iz nje in za vsak tak par dodati v graf povezavo med u in u'.

$E := \{\}$; **for** $v \in V$ **do** $K(v) := \{\}$; $S(v) := \{\}$;
for $u \in V$ **do for** $v \in N(u)$ **do** dodaj u v $K(v)$;
for $v \in V$ **do for** $u \in K(v)$ **do for** $u' \in K(v)$ **do**
  dodaj u v $S(u')$ in dodaj u' v $S(u)$;

(c) Vsaka izmed vrst napak, naštetih v opisu te podnaloge, nam določa množico $N(u)$ za vse $u \in V$. Če hočemo biti odporni na več vrst napak, moramo pri vsakem u vzeti unijo množic $N(u)$ po vseh teh vrstah napak. Vse možne kombinacije vrst napak lahko pregledujemo z rekurzijo in pri vsaki kombinaciji pokličemo postopek iz rešitve podnaloge (a) oz. (b) (odvisno od tega, ali bi radi napake le zaznavali ali tudi odpravljali). Pri tem si lahko nekaj časa prihranimo z naslednjim opažanjem: če pri neki kombinaciji vrst napak ni bilo mogoče najti primernega nabora desetih peteric, potem nima smisla v to kombinacijo dodajati še kakšne dodatne vrste napak, ker pri tako razširjeni kombinaciji gotovo tudi ne bo mogoče najti primernega nabora peteric.

Nekaj rezultatov pri zaznavanju napak:

- Zelo dober nabor desetih peteric je tisti, pri katerem uporabimo ravno vse peterice z natanko dvema prižganima bitoma (in tremi ugasnjenimi). S tem naborom lahko zaznavamo vse naslednje vrste napak: C_1, $C_\star$, S_1, $S_\star$, T_1, T_3, T_{A3} in T_5. Tega ni težko razložiti: ker imajo vse uporabljene peterice enako število prižganih bitov, je nabor odporen na vse take vrste napak, pri katerih se število prižganih bitov spremeni (spreminjanje lihega števila bitov ter prižiganje ali ugašanje poljubnega števila bitov).

- Nabor {00010, 00101, 01000, 01011, 01110, 10001, 10100, 10111, 11010, 11101} nam omogoči zaznavati napake tipov Z, C_1, S_1, T_1, T_4 in T_{4A}.

- Nabor {00000, 00011, 00110, 01001, 01100, 01111, 10010, 11000, 11011, 11110} nam omogoči zaznavati napake tipov Z, C_1, S_1, T_1, T_3, T_{3A} in T_5.

- Napak vrste T_2 ne moremo zaznavati z nobenim naborom desetih peteric; prav tako ni nabora, ki bi hkrati zaznaval tako napake vrste Z kot napake vrste $C_\star$ in/ali $S_\star$.

Če bi radi napake tudi odpravljali, ne le zaznavali, so rezultati manj spodbudni:

- Za odpravljanje napak vrste Z lahko uporabimo nabor {00000, 00001, 00011, 00111, 01000, 01010, 01110, 01111, 11000, 11001}.

- Za odpravljanje napak vrste T_5 lahko uporabimo na primer nabor {00000, 00001, 00010, 00011, 00100, 00101, 00110, 00111, 01000, 01001}. Ta problem je skoraj trivialen, saj napaka vrste T_5 pomeni, da se spremenijo vsi biti v peterici, torej moramo paziti le na to, da v naboru ne bomo imeli hkrati kakšne peterice in še njenega komplementa.

- Za nobeno drugo vrsto napak (izmed tistih, ki so naštete v besedilu naloge) ne obstaja nabor desetih peteric, s katerim bi se dalo odpravljati vse napake tiste vrste.

(d) Rešiti moramo sistem enačb $b_i = (\sum_{j=1}^{k} a_{ij} w_j) \bmod m$ za $i = 1, \ldots, n$. Na desni strani imamo ostanke po deljenju z m, ki so vsekakor manjši od m; sistem bo torej rešljiv le, če bo m večji od vseh b_i, torej $m > B$ za $B := \max_i b_i$. Če to omejitev vzamemo v zakup, lahko naše enačbe predelamo v kongruence: $\sum_{j=1}^{k} a_{ij} w_j \equiv b_i \pmod m$. (Simbol „$\equiv$“ pomeni, da morata imeti leva in desna stran enak ostanek po deljenju z m.) Lahko jih zapišemo tudi v matrični obliki: $A\mathbf{w} \equiv \mathbf{b} \pmod m$.

Ta sistem precej spominja na sistem linearnih enačb, le da imamo mi kongruence namesto enačb (in da ne poznamo m-ja). V nadaljevanju našega postopka se bomo zato zgledovali po znani Gaussovi eliminacijski metodi za reševanje sistemov linearnih enačb. Če spremenimo vrstni red kongruenc ali vrstni red neznank $w_1, \ldots, w_k$, se nabor rešitev zaradi tega nič ne spremeni. Podobno tudi, če eno kongruenco odštejemo od druge, se nabor rešitev zaradi tega nič ne spremeni. Kaj pa, če neko kongruenco pomnožimo z neko celoštevilsko konstanto c? Tedaj pri vsakem m velja naslednje: nabor $\mathbf{w}$-jev, ki skupaj s tem m-jem rešijo naš sistem, se zaradi opisane množitve ni nič zmanjšal (vsak $\mathbf{w}$, ki je bil prej rešitev, je zdaj še vedno), in če je c tuj m-ju, potem se tudi ni nič povečal (vsak $\mathbf{w}$, ki je zdaj rešitev, je bil prej tudi).

Z naštetimi operacijami — prerazporejanje kongruenc in neznank, množenje s konstanto, odštevanje ene kongruence od druge — lahko naš sistem postopoma predelamo v tako obliko, da je matrika A diagonalna in da so za nek $t \le \min\{k, m\}$ diagonalni elementi $a_{11}, \ldots, a_{tt}$ različni od 0, vsi ostali elementi matrike A pa so enaki 0. Naš sistem kongruenc je zdaj veliko preprostejši kot na začetku:

$$\begin{aligned} a_{ii} w_i &\equiv b_i \pmod m \quad \text{za } i = 1, \ldots, t \text{ (tu je } a_{ii} \neq 0) \\ 0 &\equiv b_i \pmod m \quad \text{za } i = t+1, \ldots, n. \end{aligned}$$

Pri nekaterih pogojih iz druge skupine, torej $b_i \equiv 0 \pmod m$, je mogoče b_i enak 0; tisti pogoji so trivialno izpolnjeni ne glede na to, kakšne $\mathbf{w}$ in m vzamemo, zato jih lahko v nadaljevanju odmislimo. Ločimo zdaj dve možnosti:

(1) Če je ostalo še kaj pogojev oblike $b_i \equiv 0 \pmod m$ (z neničelnimi b_i), so ti pogoji izpolnjeni natanko tedaj, ko je m delitelj vseh $b_{t+1}, \ldots, b_n$. Vsakega od teh b_i razcepimo na prafaktorje in izračunajmo množico vseh njegovih deliteljev; zdaj vemo, da pridejo v poštev pri reševanju našega sistema le taki m-ji, ki pripadajo preseku vseh teh množic (in ki so večji od B). Kandidatov za m je torej le končno mnogo in lahko za vsakega od njih posebej preverimo, ali obstaja pri tem m kakšen $\mathbf{w}$, ki skupaj z njim tvori rešitev našega prvotnega sistema kongruenc. Kajti če si

izberemo nek konkreten m, lahko naš prvotni sistem kongruenc zapišemo kot čisto običajen sistem linearnih diofantskih enačb, če uvedemo še po eno novo neznanko pri vsaki kongruenci: pogoj $\sum_j a_{ij}w_j \equiv b_i \pmod m$ moramo predelati v $\sum_j a_{ij}w_j + m \cdot x_i = b_i$. Tako dobimo sistem n linearnih diofantskih enačb z $n + k$ neznankami $(w_1, \ldots, w_k, x_1, \ldots, x_n)$, ki ga lahko rešujemo po običajnih postopkih za reševanje takih sistemov.[13]

(2) Druga možnost pa je, da pogojev $b_i \equiv 0 \pmod m$ z neničelnimi b_i sploh ni. Ostanejo še pogoji $a_{ii}w_i \equiv b_i \pmod m$; ti so neodvisni med sabo, ker vsaka neznanka w_i nastopa le v enem od njih. Za m lahko vzamemo poljubno praštevilo, ki je večje od B in ki je tuje vsem tistim c-jem, s katerimi smo kdaj pomnožili kakšno vrstico, ko smo naš sistem predelovali v diagonalno obliko (to bo zagotovilo, da ima predelani sistem enake rešitve kot prvotni). Tak m gotovo obstaja, saj je praštevil neskončno mnogo. Ko imamo enkrat v mislih nek konkreten m, lahko vsak pogoj $a_{ii}w_i \equiv b_i \pmod m$ zapišemo kot linearno diofantsko enačbo $a_{ii}w_i + m \cdot x_i = b_i$ (z dvema neznankama, w_i in x_i). Ker je m praštevilo, je ta enačba gotovo rešljiva, rešimo pa jo lahko na primer z razširjenim Evklidovim algoritmom.

9. Drugi tir A

Ko pride vlak na postajo p, moramo poskrbeti za dvoje stvari: (1) če vlak še ni končal svoje poti, mu moramo omogočiti nadaljevanje poti — torej prižgimo zeleno luč, če je povezava, po kateri bi moral nadaljevati pot, trenutno prosta; (2) če vlak ni šele začel svoje poti, torej če se je v p pripeljal po neki povezavi, to pomeni, da je ta povezava zdaj prosta, zato lahko zdaj nanjo spustimo kakšen drug vlak.

Tega, ali je vlak šele začel svojo pot, ni težko preveriti: to se zgodi, če je vlak na $p = 1$ in se pelje v desno ali pa je na $p = n$ in se pelje v levo. Podobno lahko tudi preverimo, ali je vlak prišel na konec svoje poti: to je takrat, če je na $p = n$ in se pelje v desno ali pa je na $p = 1$ in se pelje v levo.

Da bomo vedeli, kdaj smemo prižgati zeleno luč in na neko povezavo spustiti nov vlak, moramo v neki tabeli hraniti za vsako povezavo podatek o tem, ali je zasedena ali prosta (torej ali po njej trenutno pelje kak vlak ali ne). Hraniti moramo tudi podatke o tem, koliko vlakov čaka na posamezni postaji, in sicer ločeno za vlake, ki peljejo v levo, in tiste, ki peljejo v desno. Tako bomo zeleno luč prižgali le takrat, ko jo kakšen vlak res potrebuje. Imejmo torej naslednje globalne spremenljivke:

```
int stCakajocih[n][2];
bool zasedena[n − 1];
```

Ob inicializaciji moramo le postaviti vse števce na 0 in označiti vse povezave kot proste:

```
void Inicializacija()
{
  for (int i = 0; i < n; i++) stCakajocih[i][0] = 0, stCakajocih[i][1] = 0;
  for (int i = 0; i < n − 1; i++) zasedena[i] = false;
}
```

[13] Gl. npr. knjižico Jožeta Grassellija *Diofantske enačbe* (Ljubljana, DMFA 1984) od str. 29 naprej in Wikipedijo *s. v.* Diophantine equations.

Zdaj lahko zapišemo glavni del naše rešitve. Pazimo na to, da naloga šteje postaje od 1 do n, za naše namene pa je bolj prikladno, če imamo števila od 0 do $n-1$, da jih bomo lahko uporabljali kot indekse pri delu s tabelami. Zato parameter p za začetek zmanjšamo za 1.

```
void ObPrihodu(int p, bool levo)
{
  p−−;
  /* Če vlak še ni prišel na konec poti... */
  bool konec = (levo && p == 0) || (! levo && p == n − 1);
  if (! konec)
  {
    /* ...povečajmo števec čakajočih na postaji p. */
    stCakajocih[p][levo ? 0 : 1]++;
    /* Poskusimo ga poslati naprej proti cilju. */
    Poslji(p, levo);
  }
  /* Če se ni ravnokar pojavil na progi... */
  bool zacetek = (levo && p == n − 1) || (! levo && p == 0);
  if (! zacetek)
  {
    /* ...označimo povezavo, po kateri se je pripeljal, za prosto. */
    zasedena[levo ? p : p − 1] = false;
    /* Ker je ta povezava zdaj prosta, lahko poskusimo po njej poslati kak vlak
       s postaje p v nasprotni smeri od te, iz katere je trenutni vlak pravkar prispel. */
    Poslji(p, ! levo);
    /* Če takega ni, pa poskusimo poslati po njej še en vlak v isto smer,
       v katero pelje tudi trenutni vlak. */
    Poslji(levo ? p + 1 : p − 1, levo);
  }
}
```

Naša gornja rešitev, ko se neka povezava sprosti, poskusi po njej najprej poslati vlak v nasprotno smer od prejšnjega; šele če takega ni, pa poskusi po njej poslati vlak v isto smer. S tem poskušamo poskrbeti, da bodo vlaki v eno smer v povprečju vozili približno enako hitro kot v drugo.

Oglejmo si še podprogram Poslji, ki ga kliče gornja rešitev, da poskusi z določene postaje p spustiti en vlak naprej v določeno smer. Ta podprogram mora torej preveriti, ali kakšen tak vlak sploh obstaja (in čaka na postaji p) in ali je povezava, po kateri bi se ta vlak peljal, trenutno prosta. Če sta tadva pogoja izpolnjena, lahko prižgemo zeleno luč na ustreznem semaforju, zmanjšamo števec čakajočih vlakov in označimo povezavo, po kateri se vlak zdaj pelje, kot zasedeno:

```
void Poslji(int p, bool levo)
{
  if (stCakajocih[p][levo ? 0 : 1] <= 0) return;
  if (zasedena[levo ? p − 1 : p]) return;
  PrizgiSemafor(p + 1, levo, true);
  stCakajocih[p][levo ? 0 : 1]−−;
  zasedena[levo ? p − 1 : p] = true;
}
```

10. Drugi tir B

Če nekega vlaka ne uspemo (do konca dneva) pripeljati na cilj, ampak z njim prevozimo le del poti, ne bomo imeli od tega nobene koristi, saj naloga pravi, da šteje le tovor na tistih vlakih, ki pridejo na cilj. Torej se je smiselno ukvarjati le s toliko vlaki, kolikor jih bomo lahko prepeljali vse do cilja, ostale vlake pa je bolje kar pustiti na tistem koncu proge, kjer stojijo na začetku dneva.

Še eno koristno opažanje je, da na gibanje vlaka nič ne vpliva to, koliko tovora ta vlak prevaža. Če imamo scenarij, po katerem lahko (v v ali manj časovnih enotah) prepeljemo k vlakov z levega konca proge na desnega in m vlakov z desnega konca proge na levega, potem je ta scenarij enako dobro izvedljiv ne glede na to, katerih k vlakov izberemo izmed ℓ vlakov, ki na začetku dneva čakajo na levem koncu proge, in katerih m vlakov izberemo izmed d vlakov, ki na začetku dneva stojijo na desnem koncu proge. Razmisliti moramo torej predvsem o tem, za kakšne (k, m) je tak scenarij sploh mogoč (v okviru časovne omejitve, torej v največ v korakih), nato pa bomo lahko količino prepeljanega tovora maksimizirali preprosto tako, da bomo leve vlake uredili padajoče po a_i in vzeli prvih k, podobno pa tudi desne vlake uredili padajoče po b_i in vzeli prvih m.

V nadaljevanju razmisleka se dogovorimo, da bomo čas merili takole: $t = 0$ nam pomeni čas, ko se vožnja začne; $t = 1$ je časovni trenutek po prvem koraku, $t = 2$ po drugem koraku in tako naprej. Z drugimi besedami se torej u-ti korak začne ob času $t = u - 1$ in konča ob času $t = u$.

Razmislimo zdaj o tem, koliko časa potrebujemo, da prepeljemo k vlakov z leve na desno in m vlakov z desne na levo; čas, ki ga porabimo za to, bomo označili s $f(k, m)$. Če bi se vsi vlaki premikali v isto smer, bi bila stvar zelo preprosta. Na primer, če imamo en sam vlak, bo pač potreboval $n - 1$ korakov, da pride od enega konca proge do drugega; torej velja $f(1, 0) = f(0, 1) = n - 1$. Drugi vlak lahko sledi prvemu z zamikom enega koraka, tretji prav tako drugemu in tako naprej; tako dobimo $f(k, 0) = f(0, k) = n + k - 2$.

Zanimivejši pa je primer, ko sta tako k kot m večja od 0, torej imamo nekaj vlakov, ki vozijo v levo, in nekaj vlakov, ki vozijo v desno. Vprašanje je, kdo naj čaka, če hočeta dva vlaka istočasno vstopiti na neko povezavo, eden z leve strani in eden z desne strani. Za koristno se izkaže naslednja zamisel: razdelimo našo progo na levo in desno polovico; na levi polovici naj imajo prednost vlaki, ki peljejo v desno, na desni polovici pa vlaki, ki peljejo v levo.

Natančneje povedano: če je n lih, recimo $n = 2p+1$, je število povezav $n-1 = 2p$ sodo; leva polovica torej obsega levih p povezav, desna pa desnih p povezav. Če pa je n sod, recimo $n = 2p$, je število povezav $n - 1 = 2p - 1$ liho; tedaj za levo polovico vzemimo levih p povezav, za desno pa desnih $p - 1$ povezav (tu je torej desna polovica za eno povezavo krajša od leve). Oba primera lahko združimo v enega, če rečemo, da leva polovica obsega levih $p := \lfloor n/2 \rfloor$ povezav, desna pa desnih $q := n - 1 - p = \lceil n/2 \rceil - 1$ povezav. Postaji, kjer se leva in desna polovica srečata (torej postaji $p + 1$, ker so postaje oštevilčene od 1 naprej), bomo rekli *srednja* postaja. Opazimo lahko, da je $p \geq q$.

Rekli smo torej, naj imajo na levi polovici prednost vlaki, ki peljejo v desno, na desni polovici pa naj imajo prednost vlaki, ki peljejo v levo. Za ilustracijo si oglejmo, kako bi ta postopek deloval na konkretnem primeru z $n = 7$ postajami,

$k = 4$ vlaki, ki hočejo priti z levega konca na desnega, in $m = 2$ vlakoma, ki hočeta priti z desnega konca na levega.

Stanje ob času	Vlaki na postaji številka						
	1	2	3	4	5	6	7
$t = 0$	DCBA						ZY
1	DCB	A				Z	Y
2	DC	B	A		Z	Y	
3	D	C	B	AZ	Y		
4		D	C	BAZY			
5			D	CBZY	A		
6				DCZY	B	A	
7			Z	DY	C	B	A
8		Z	Y		D	C	BA
9	Z	Y				D	CBA
10	ZY						DCBA

Tu imamo torej $n = 7$ in zato $p = q = 3$; srednja postaja ima številko $p + 1 = 4$. Opazimo lahko, da se vlaki z obeh strani neovirano peljejo do srednje postaje, nato pa se začnejo na njej nabirati, ker jim vlaki, ki prihajajo iz nasprotne strani, preprečujejo nadaljevanje poti. Pot lahko nadaljujejo šele, ko vlakov iz nasprotne strani zmanjka.

V splošnem, če imamo k vlakov, ki se peljejo od leve proti desni, bo prvi od njih speljal z levega konca (postaja 1) ob času 0 in prišel na srednjo postajo ob času p; ostali mu sledijo v naslednjih časovnih korakih, tako da zadnji od njih pride na srednjo postajo ob času $p + k - 1$. Podobno je tudi z m vlaki, ki se peljejo od desne proti levi; prvi od njih gre na pot ob času 0 in pride na srednjo postajo ob času q, zadnji pa zato doseže srednjo postajo ob času $q + m - 1$.

Vlaki, ki peljejo proti desni, lahko zapustijo srednjo postajo šele takrat, ko so nanjo prišli že vsi vlaki, ki peljejo proti levi; kajti povezava, po kateri desno vozeči vlak zapusti srednjo postajo, pripada že desni polovici proge, zato imajo na njej prednost levo vozeči vlaki, dokler jih je še kaj. Prvi desno vozeči vlak torej ne more zapustiti srednje postaje prej kot ob času $q + m - 1$; obenem seveda tudi velja, da vlak ne more zapustiti postaje prej, preden je prišel nanjo; torej prvi desno vozeči vlak zapusti srednjo postajo ob času $\max\{p, q + m - 1\}$. Po q korakih pride ta vlak na cilj, torej na desni konec proge; ostali desno vozeči vlaki mu sledijo v naslednjih $m-1$ korakih, tako da zadnji od njih pride na cilj ob času $\max\{p, q+m-1\}+q+k-1$.

Analogen razmislek za levo vozeče vlake bi nam pokazal, da zadnji od njih pride na cilj (torej na levi konec proge) ob času $\max\{q, p + k - 1\} + p + m - 1$. Skupni čas, potreben za to, da vsi vlaki pridejo na cilj, je torej

$$f(k, m) = \max\{\max\{p, q + m - 1\} + q + k - 1, \max\{q, p + k - 1\} + p + m - 1\}.$$

Pišimo $k' = k - 1$ in $m' = m - 1$. Ker sta k in m oba večja od 0, sta k' in m' oba ≥ 0. Gornjo enačbo lahko predelamo takole:

$$\begin{aligned} f(k, m) &= \max\{\max\{p, q + m'\} + q + k', \max\{q, p + k'\} + p + m'\} \\ &= \max\{p + q + k', 2q + m' + k', p + q + m', 2p + k' + m'\} \\ &= \max\{p + q + \max\{k', m'\}, \max\{2p, 2q\} + k' + m'\} \\ &= \max\{p + q + \max\{k', m'\}, 2p + k' + m'\}. \end{aligned}$$

Pri tem smo v zadnjem koraku upoštevali, da je $p \geq q$, zato je $\max\{2p, 2q\} = 2p$. Iz enakega razloga lahko zdaj opazimo, da je $2p \geq p + q$; in ker sta tako k' kot m' nenegativna, je $k' + m' \geq \max\{k', m'\}$. Torej je v zunanjem max na koncu prejšnje izpeljave druga možnost zagotovo vedno večja ali enaka prvi. Tako torej lahko zaključimo: $f(k, m) = 2p + k' + m'$. To lahko naprej zapišemo kot $2p + k + m - 2$; pri lihih n je to enako $n + k + m - 3$, pri sodih n pa $n + k + m - 2$. Z drugimi besedami lahko rečemo, da je $f(k, m) = n + k + m - 2 - (n \bmod 2)$.

Naloga pravi, da imamo za prevažanje vlakov na voljo le v enot časa, torej moramo k in m izbrati tako, da bo $f(k, m) \leq v$. Iz pravkar dobljene formule za $f(k, m)$ tako dobimo $k + m \leq v - n + 2 + (n \bmod 2)$. Vidimo lahko, da nam razpoložljivi čas v pravzaprav omeji le skupno število prepeljanih vlakov (torej $k + m$), ne pa tudi tega, koliko od teh vlakov gre z leve na desno in koliko z desne na levo. Odločimo se torej, da bomo prepeljali $r := \min\{\ell + d, v - n + 2 + (n \bmod 2)\}$ vlakov. Zdaj lahko vse čakajoče vlake, tako leve kot desne, zložimo v en seznam in ga uredimo padajoče po količini tovora na vlaku; prvih r vlakov na tako urejenem seznamu prepeljimo po doslej opisanem scenariju (torej tako, da imajo na levi polovici proge vedno prednost desno vozeči vlaki, na desni polovici pa levo vozeči), ostale vlake pa pustimo stati, kjer so bili na začetku dneva.

Prepričajmo se, da je tako dobljena rešitev najboljša možna. Dovolj bo, če pokažemo, da je čas vožnje $f(k, m)$, ki smo ga dobili pri dosedanji rešitvi, najmanjši možni, torej ni mogoče v manj kot $f(k, m)$ časa prepeljati k vlakov z levega konca proge na desnega in m vlakov z desnega konca proge na levega.

Oglejmo si dogajanje na povezavi med srednjo postajo $p + 1$ in njeno levo sosedo p. Levo od te povezave je na progi še $p - 1$ povezav, desno od nje pa q povezav. Prvi desno vozeči vlak se torej po njej ne more peljati prej kot v p-tem časovnem koraku (ker mora prej prevoziti $p - 1$ povezav levo od nje), prvi levo vozeči vlak pa ne prej kot v $(q + 1)$-vem časovnem koraku (ker mora prej prevoziti q povezav desno od nje). Prvi vlak torej našo povezavo prepelje v $\min\{p, q + 1\}$-tem časovnem koraku ali kasneje. Ker mora po povezavi prej ali slej peljati vseh $k + m$ vlakov, jo torej zadnji prepelje v časovnem koraku $\min\{p, q + 1\} + k + m - 1$ ali kasneje. Če je ta zadnji vlak eden od tistih, ki peljejo v desno, potrebuje po tistem, ko prevozi našo povezavo, še q korakov, da pride do cilja (ker je toliko povezav desno od nje); če pa je eden od tistih, ki peljejo v levo, potrebuje do cilja še $p - 1$ korakov (ker je toliko povezav levo od naše). Zadnji vlak torej ne more doseči cilja prej kot ob času $\tau := \min\{p, q + 1\} + k + m - 1 + \min\{p - 1, q\}$. Spomnimo se, da velja bodisi $p = q$ (če je n lih) bodisi $p = q + 1$ (če je n sod); v obeh primerih je $\min\{p, q + 1\} = p$ in $\min\{p - 1, q\} = p - 1$. Tako dobimo $\tau = 2p + k + m - 2$; pri lihem n $(n = 2p + 1)$ je to naprej enako $n + k + m - 3$, pri sodem n $(n = 2p)$ pa je enako $n + k + m - 2$.

Tako vidimo, da je τ, torej spodnja meja za najkrajši možni čas prevoza k vlakov z leve na desno in m vlakov z desne na levo, ravno enaka $f(k, m)$, torej času, ki ga za prevoz teh vlakov porabi naša rešitev. Torej je res nemogoče, da bi obstajala kakšna boljša rešitev od naše.

11. Knjižnica

(*a*) Za vsakega uporabnika je koristno imeti dvojno povezan seznam (*doubly-linked list*) vseh plošč tega uporabnika. Tako bomo lahko katerokoli ploščo brez težav

dodali ali pobrisali, pa tudi našteli vse plošče, ki jih ima izposojene. Pri vsakem seznamu bomo vzdrževali tudi njegovo dolžino, da nam pri operaciji KolikoIma ne bo treba pregledovati celega seznama. Poleg tega bo prišel za vsako ploščo prav še podatek o tem, kdo (če sploh kdo) jo ima trenutno izposojeno. Spodnji primer kaže, kako lahko vse te podatke zložimo v dve tabeli, eno za plošče in eno za osebe:

```
struct Plosca
{
  int kdo = −1;                /* kdo jo ima izposojeno */
  int prej = −1, nasl = −1;    /* prejšnja in naslednja plošča istega uporabnika */
};
struct Oseba
{
  int prva = −1;               /* prva plošča tega uporabnika */
  int stPlosc = 0;             /* koliko plošč ima izposojenih */
};
Plosca plosce[n];
Oseba osebe[m];
```

Pri izposoji plošče lahko s poljem kdo takoj preverimo, če je že izposojena; če ni, jo moramo dodati na začetek seznama izposojenih plošč tistega uporabnika, ki si jo zdaj izposoja. Popraviti moramo tudi števec izposojenih plošč tega uporabnika (stPlosc).

```
bool Izposodi(int p, int o)
{
  if (plosce[p].kdo >= 0) return false;
  plosce[p].prej = −1; plosce[p].kdo = o;
  int nasl = osebe[o].prva; plosce[p].nasl = nasl;
  if (nasl >= 0) plosce[nasl].prej = p;
  osebe[o].prva = p; osebe[o].stPlosc++;
  return true;
}
```

Pri vračanju plošče je stvar podobna; pobrisati jo moramo iz seznama (če brišemo prvo ploščo, je treba ustrezno popraviti tudi polje prva pri tistem uporabniku) in zmanjšati števec izposojenih plošč. Spodnja funkcija ne preverja, če je plošča sploh bila izposojena, saj naloga tega ne zahteva.

```
void Vrni(int p)
{
  int o = plosce[p].kdo, prej = plosce[p].prej, nasl = plosce[p].nasl;
  if (prej >= 0) plosce[prej].nasl = nasl;
  else osebe[o].prva = nasl;
  if (nasl >= 0) plosce[nasl].prej = prej;
  osebe[o].stPlosc−−; plosce[p].kdo = −1;
}
```

Za izpis izposojenih plošč se moramo le sprehoditi po seznamu danega uporabnika: začnemo pri plošči prva in se s pomočjo vrednosti nasl premikamo naprej po seznamu.

```
void KajIma(int o)
{
  for (int p = osebe[o].prva; p >= 0; p = plosce[p].nasl)
    printf("%d\n", p);
}
```

Operaciji PriKomJe in KolikoIma pa sta še posebej preprosti, saj morata le vrniti podatke, ki jih imamo že pri roki:

```
int PriKomJe(int p) { return plosce[p].kdo; }
int KolikoIma(int o) { return osebe[o].stPlosc; }
```

Tako torej vidimo, da je časovna zahtevnost vseh operacij le $O(1)$, razen pri izpisu seznama izposojenih plošč, kjer je časovna zahtevnost linearna v odvisnosti od dolžine tega seznama.

(Mimogrede: v gornji rešitvi smo si dvojno povezan seznam napisali sami, v praksi pa je koristno preveriti, če je kaj podobnega že na voljo v standardni knjižnici našega programskega jezika. V C++ je to na primer razred std::list.)

(b) Preprosta rešitev je dvodimenzionalna tabela velikosti $m \times n$, v kateri za vsako kombinacijo plošče in uporabnika hranimo podatek o tem, ali je imel ta uporabnik to ploščo že kdaj izposojeno. Če je plošč in uporabnikov veliko, bo ta tabela neugodno velika, sploh ker bo najbrž večinoma prazna (posamezni uporabnik si je najbrž izposodil le majhen delež izmed vseh plošč v knjižnici). Neugodna je tudi poraba časa: za operacijo KajVseJeZelmel je treba iti po vseh ploščah in v omenjeni tabeli preverjati, ali je imel dani uporabnik tisto ploščo že kdaj izposojeno; to bo torej vzelo $O(n)$ časa, četudi je imel ta uporabnik doslej izposojenih le malo plošč. Podobno je tudi pri KdoVseJoJeZelmel.

Namesto tega je bolje uporabiti razpršeno tabelo (*hash table*) in v njej hraniti kot ključe le tiste pare (p, o), za katere je imel uporabnik p že kdaj izposojeno ploščo o. Pri vsakem takem ključu pa imejmo kot spremljevalno vrednost še podatek o tem, katera je naslednja plošča izmed tistih, ki jih je imel kdaj izposojene uporabnik o, in kateri je naslednji uporabnik izmed tistih, ki so si kdaj izposodili ploščo p. Tako so pari (p, o) povezani v sezname (enojno povezane — to je dovolj, saj bomo morali vanje elemente le dodajati, ne pa jih tudi brisati); imamo po en seznam za vsakega uporabnika in za vsako ploščo. Poleg tega imejmo še dve tabeli z začetki teh seznamov: ena torej za vsakega uporabnika pove prvo izposojeno ploščo, druga pa za vsako ploščo prvega uporabnika, ki si jo je izposodil. Tako se bomo brez težav lahko sprehodili po kateremkoli od teh seznamov.

```
#include <unordered_map>
#include <utility> // za pair

typedef std::pair<int, int> Par;
struct MyHash {
  size_t operator () (Par p) const noexcept {
    size_t a = p.first + p.second; return (a * (a + 1)) / 2 + p.first; } };

std::unordered_map<Par, Par, MyHash> zgodovina;
int prvaPlosca[m], prvaOseba[n];
```

Na začetku moramo vse elemente v tabelah prvaPlosca in prvaOseba inicializirati na -1. V funkciji Izposodi tik pred koncem (stavek return true) dodajmo še:

```
  auto pr = zgodovina.insert({{p, o}, {prvaPlosca[o], prvaOseba[p]}});
  if (pr.second) { prvaPlosca[o] = p; prvaOseba[p] = o; }
```

Prvi stavek doda ključ (p, o) v razpršeno tabelo (s primerno spremljevalno vrednostjo), če ga v njej še ni bilo; drugi stavek pa, če je do tega dodajanja res prišlo, še poskrbi, da se novi par znajde na začetku seznamov za ploščo p in za uporabnika u.

Sprehajanje po seznamih je s pomočjo naše razpršene tabele preprosto:

```
void KajVseJeZelmel(int o)
{
  for (int p = prvaPlosca[o]; p >= 0; p = zgodovina.at({p, o}).first)
    printf("%d\n", p);
}
void KdoVseJoJeZelmel(int p)
{
  for (int o = prvaOseba[p]; o >= 0; o = zgodovina.at({p, o}).second)
    printf("%d\n", p);
}
```

Če predpostavimo, da je cena posameznega dostopa do razpršene tabele $O(1)$, je zdaj čas operacije Izposoja še vedno $O(1)$, čas operacij KajVseJeZelmel in KdoVseJoJeZelmel pa je linearna v odvisnosti od dolžine seznamov, ki jih morata izpisati.

12. Cenik

Iz opisa v besedilu naloge vidimo, da je cenik nekakšno zaporedje (urejeno po času) zapisov oblike $\langle od, do, cena \rangle$. Ker pa med zapisi ni prekrivanj ali lukenj, za dva zaporedna zapisa vedno velja, da je *do* prejšnjega zapisa ravno za 1 manjši kot *od* naslednjega zapisa. Torej pravzaprav ni treba hraniti pri vsakem zapisu tudi časa *do*, saj ga bomo, če ga bomo kdaj potrebovali, lahko vedno izračunali iz *od* naslednjega zapisa. Recimo torej, da bo naš cenik urejeno zaporedje parov (d_i, c_i) (za $i = 1, 2, \ldots, n$) ki povedo, da se z dnem d_i začne cena c_i (ki traja do vključno dneva $d_{i+1} - 1$). Na koncu seznama si mislimo kot stražarja še zapis $(n + 1, 0)$, ki predstavlja konec zadnjega intervala z znano ceno (ki se konča z dnevom n).

V računalniku bi lahko tako zaporedje predstavili s tabelo (*array*), s seznamom (*linked list*), lahko pa tudi s kakšno (primerno uravnoteženo) drevesasto strukturo, npr. rdeče-črnim drevesom. Vsaka od teh struktur ima svoje prednosti in slabosti: pri tabeli imamo poceni naključni dostop do vsakega elementa, poleg tega lahko iščemo zapise (po času) z bisekcijo; po drugi strani pa je vrivanje in brisanje elementov drago, ker moramo pri tem premakniti vse nadaljnje elemente v tabeli. Pri seznamu je vrivanje in brisanje poceni, nimamo pa poceni naključnega dostopa ali iskanja po času (saj moramo v ta namen pregledovati elemente seznama po vrsti vse od začetka). Pri drevesu lahko vse omenjene operacije izvajamo precej hitro (v logaritemskem času), vendar pa je drevo bolj zapleteno za implementacijo (če ga hočemo implementirati sami; v praksi ga imamo zelo verjetno že v knjižnici našega programskega jezika).

Razmislimo zdaj o postopku za dodajanje novega zapisa v našo podatkovno strukturo. Recimo, da naj bi novi zapis pokril dneve od Z do K s ceno C. Pri dodajanju moramo poskrbeti za to, da bomo ustvarili zapis za začetek tega intervala (dan Z, cena C), zapis za konec tega intervala (dan $K + 1$ in taka cena, kot je že pred dodajanjem veljala na ta dan), in da bomo pobrisali morebitne zapise med njima (ker zdaj za celotni interval od Z do K velja ena sama cena, namreč C).

1. Če v zaporedju še ni zapisa z dnevom $K+1$, ga dodajmo. Poiščimo torej tak indeks j, za katerega je $d_{j-1} < K+1 \leq d_j$. Če pri tem j velja $K+1 < d_j$, vrinimo med zapisa $j-1$ in j nov zapis $(K+1, c_{j-1})$ (novi zapis torej zdaj stoji na indeksu j).

2. Če v zaporedju še ni zapisa z dnevom Z, ga dodajmo. Poiščimo torej tak indeks i, za katerega je $d_{i-1} < Z \leq d_i$. Če pri tem i velja $Z < d_i$, vrinimo med zapisa $i-1$ in i nov zapis (Z, C) (ki bo po novem stal na indeksu i), sicer pa le popravimo $c_i := C$ pri že obstoječem zapisu z dnevom Z.

3. Nato pobrišimo vse zapise, ki imajo datum na območju $Z < d \leq K$.

Pri drugi točki je možna še drobna izboljšava: ko enkrat imamo zapis za datum Z, lahko pogledamo, če ima slučajno enako ceno kot prejšnji zapis v seznamu; če da, lahko zapis za datum Z pobrišemo.

Pri brisanju je stvar podobna; poskrbeti moramo za to, da na intervalu od Z do K velja ves čas ista cena kot na dan $Z-1$. V ta namen moramo poskrbeti, da bomo v seznamu imeli zapis za konec tega intervala (dan $K+1$, cena pa enaka, kot je na ta dan veljala že pred brisanjem), nato pa lahko pobrišemo zapise, ki ležijo znotraj našega intervala. Vprašanje je še, kaj storiti, če je $Z = 1$; opis naloge pri brisanju omenja prejšnjo ceno, pri $Z = 1$ pa prejšnje cene ni. Spodnja rešitev v tem primeru namesto prejšnje cene uporabi naslednjo ceno (tisto z dneva $K+1$).

1. Če v zaporedju še ni zapisa z dnevom $K+1$, ga dodajmo. Poiščimo torej tak indeks j, za katerega je $d_{j-1} < K+1 \leq d_j$. Če pri tem j velja $K+1 < d_j$, vrinimo med zapisa $j-1$ in j nov zapis $(K+1, c_{j-1})$ (novi zapis torej zdaj stoji na indeksu j).

2. Nato pobrišimo vse zapise, ki imajo datum na območju $Z \leq d \leq K$.

3. Če je $Z = 1$, je na začetku seznama zdaj zapis $(K+1, c)$ za neko ceno c. Popravimo pri tem zapisu datum na 1.

13. Komplet

Za začetek je koristno vhodni cenik urediti po datumu (namesto po izdelkih ali kakorkoli je že pač bil urejen); spotoma lahko iz njega še zavržemo zapise, ki se nanašajo na izdelke, ki niso del našega kompleta. Ko je cenik urejen po datumu, se lahko sprehajamo po njem in sproti v neki tabeli (spodnji program ima zato vektor cene) vzdržujemo podatke o trenutnih cenah vseh izdelkov v kompletu. Pri vsakem zapisu se cena nekega izdelka spremeni, novo ceno vpišemo v tabelo in popravimo vsoto, ki predstavlja ceno celotnega kompleta (v spodnji rešitvi je to spremenljivka cenaKompleta). Ko obdelamo vse zapise za določen datum, poznamo pravo ceno kompleta od tistega dneva naprej (do naslednje spremembe) in jo lahko skupaj z datumom dodamo v izhodni seznam (spremenljivka cenikKompleta.

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

struct CenaIzdelka { string izdelek; int od; double znesek; };
```

```
struct CenaKompleta { int od; double znesek; };

void CenikKompleta(const vector<CenaIzdelka> &cenikIzdelkov,
                   const vector<string> &elementi,
                   vector<CenaKompleta> &cenikKompleta)
{
  // Iz cenika skopirajmo podatke za izdelke, ki so del kompleta.
  // Pri tem zamenjajmo njihova imena z indeksi (v vektor „elementi“).
  struct Cena { int izdelek, od; double znesek; };
  vector<Cena> cenik;
  for (const auto &c : cenikIzdelkov)
  {
    auto i = find(elementi.begin(), elementi.end(), c.izdelek);
    if (i == elementi.end()) continue;
    cenik.push_back({i - elementi.begin(), c.od, c.znesek});
  }
  // Uredimo cenik po datumu.
  sort(cenik.begin(), cenik.end(),
       [] (const Cena &x, const Cena &y) { return x.od < y.od; });
  // Naslednji vektor bo hranil trenutne cene posameznih izdelkov v kompletu.
  vector<double> cene(elementi.size(), 0);
  // Sprehodimo se po ceniku in računajmo ceno kompleta.
  double cenaKompleta = 0; cenikKompleta.clear();
  for (int i = 0; i < cenik.size(); )
  {
    int datum = cenik[i].od;
    // Na isti dan je lahko več zapisov; preglejmo jih vse,
    // da bomo dobili pravo ceno kompleta na ta dan.
    for ( ; i < cenik.size() && cenik[i].od == datum; i++)
    {
      int izdelek = cenik[i].izdelek;
      // Popravimo ceno kompleta zaradi spremembe cene tega izdelka.
      cenaKompleta -= cene[izdelek];
      cene[izdelek] = cenik[i].znesek;
      cenaKompleta += cene[izdelek];
    }
    // Dodajmo novo ceno v izhodni vektor.
    cenikKompleta.push_back({datum, cenaKompleta});
  }
}
```

14. Dvigalo

Označimo najnižje nadstropje, v katerem je na začetku kak potnik, z a; najvišje nadstropje, v katerega želi kak potnik priti, pa z b. Torej je $a := \min_i a_i$ in $b := \max_i b_i$.

Če je $a > 0$, mora dvigalo najprej narediti a korakov navzgor, preden lahko začne prevažati potnike — in nobene koristi ni od tega, da bi se v tej začetni fazi gibalo še kako drugače. Odtlej lahko predpostavimo, da se dvigalo ves čas giblje na območju od a do b, saj ne more biti nobene koristi od tega, da bi se še kdaj spustilo pod a ali dvignilo nad b.

Naj bo U_t (za $a \le t < b$) množica potnikov, ki morajo prečkati mejo med nadstropjema t in $t+1$ — torej takih, ki začnejo v nadstropju t ali še nižje, prišli pa

bi radi v nadstropje $t+1$ ali še višje. Z drugimi besedami, $U_t := \{i \,:\, a_i \le t < b_i\}$. Število teh potnikov pa označimo z $u_t := |U_t|$. Dvigalo se mora torej vsaj u_t-krat premakniti iz nadstropja t v $t+1$, da bo lahko prepeljalo vse te potnike čez mejo med njima. Ko enkrat prečka to mejo v smeri navzgor, jo mora nato prečkati v smeri navzdol, preden jo lahko še kdaj prečka v smeri navzgor; poleg u_t premikov iz nadstropja t v $t+1$ moramo torej imeti še vsaj $u_t - 1$ premikov iz nadstropja $t+1$ v t.

Skupno število premikov dvigala bo torej nujno vsaj $a + \sum_{t=a}^{b-1}(2u_t - 1)$. Ni težko sestaviti postopka, ki prepelje vse potnike v natanko toliko korakih: na vsakem koraku bomo pobrali najnižjega takega potnika, ki še ni prišel na cilj, in ga prepeljali eno nadstropje višje. To ponavljamo, dokler niso vsi potniki tam, kjer želijo biti. Zapišimo ta postopek s psevdokodo:

```
1 najprej naredimo a korakov navzgor, da pride dvigalo v nadstropje a;
2 for t := a to b − 1:
3    (* Zdaj je dvigalo v nadstropju t in za vsakega potnika i velja:
        (a) če je a_i > t, je ta potnik še vedno v nadstropju a_i;
        (b) če je b_i ≤ t, je ta potnik že v nadstopju b_i;
        (c) sicer je ta potnik v nadstropju t. (To so ravno potniki iz U_t.) *)
4    za vsakega potnika iz U_t:
5       prepeljimo tega potnika v nadstropje t + 1;
6       če to še ni bil zadnji potnik iz U_t, se vrnimo s praznim
           dvigalom nazaj v nadstropje t;
```

Dvigalo v vrstici 1 naredi a korakov, nato pa v vsaki iteraciji zunanje zanke (po t) naredi še $2u_t - 1$ korakov v vrsticah 4–6.

Prepričajmo se, da invarianta v točki 3 res drži. Glede točke (a) je tako: ker naš postopek prevaža potnike le navzgor in ker smo zdaj na začetku iteracije za t, to pomeni, da so doslej najvišji prevozi bili iz $t-1$ v t. Ker je $a_i > t$, to pomeni, da potnika i doslej še nismo nikamor prepeljali, torej je še vedno v a_i, prav kakor trdi naša invarianta.

Za (b) in (c) bomo morali dokazovati z indukcijo. Preden se je lotimo, pa poudarimo, da so možnosti (a), (b) in (c) disjunktne in da (c) res pokrije natanko U_t. Najprej pokažimo, da sta (a) in (b) disjunktni: res, ker če bi kak potnik padel pod oba primera, bi to pomenilo, da ima $a_i > t$ in zato $b_i > a_i > t$, obenem pa $b_i \le t$, kar je protislovje. Primer (c) je po definiciji disjunkten z (a) in (b), tako da pokažimo le še to, da res pokrije ravno potnike iz U_t. Ker v (c) pridejo le tisti potniki, ki niso prišli v (a) ali (b), velja zanje $\neg(a_i > t \vee b_i \le t)$, kar je isto $a_i \le t \wedge b_i > t$ oz. $a_i \le t < b_i$, prav to pa je pogoj, s katerim smo definirali U_t.

Lotimo se zdaj naše indukcije. Na začetku prve iteracije imamo $t = a$. Točka (b) je izpolnjena trivialno, ker vanjo sploh ne pade noben potnik: $b_i \le t$ bi namreč pomenilo $b_i \le a$, kar bi skupaj z $a_i < b_i$ dalo $a_i < a$, kar je protislovje. Točka (c) pokrije, kot smo videli, potnike iz U_t, torej tiste z $a_i \le t < b_i$. Ker imamo $t = a = \min_i a_i$, je pogoj $a_i \le t$ lahko izpolnjen le tako, da velja enakost. V U_t so torej sami taki potniki, ki imajo $a_i = t$. Ker nismo prepeljali še nobenega potnika, so vsi potniki še na svojem začetnem položaju; tisti iz U_t so torej vsi še v a_i, torej v t, prav to pa zatrjuje naša invarianta.

Recimo zdaj, da invarianta velja na začetku iteracije t. Radi bi pokazali, da na koncu te iteracije velja tudi za $t+1$. — Primer (b) pokriva takrat (na koncu iteracije t oz. na začetku iteracije $t+1$) vse tiste potnike, ki jih je že na začetku iteracije t, poleg njih pa zdaj tudi še tiste, za katere je $b_i = t+1$. Ker je $a_i < b_i$, to pomeni, da so ti potniki imeli $a_i \le t$ in $b_i > t$, torej so bili v U_t; torej smo jih med t-to iteracijo v vrsticah 4–6 prepeljali iz t v $t+1$; torej so ti potniki zdaj res v $t+1$, torej v svojem b_i, prav to pa zatrjuje zanje pogoj (b). — Pri primeru (c) pa invarianta zdaj trdi, da so vsi potniki iz U_{t+1} zdaj v nadstropju $t+1$. Spomnimo se, da je $U_{t+1} = \{i \ : \ a_i \le t+1 < b_i\}$. Ločimo dve možnosti: (1) če je $a_i = t+1$, to pomeni, da tega potnika doslej še nikoli nismo premikali, saj je bil doslej najvišji premik navzgor lahko le tisti iz t v $t+1$. Taki potniki so torej še na svojem začetnem položaju, torej v $t+1$, prav to pa zanje trdi naša invarianta. (2) Če pa je $a_i < t+1$, to pomeni $a_i < t$, kar skupaj z $b_i > t+1$ (in zato $b_i > t$) pomeni, da so ti potniki pripadali že množici U_t. Njih smo iteraciji t v vrsticah 4–6 prepeljali v nadstropje $t+1$, torej so zdaj res tam, prav kot zatrjuje naša invarianta.

Zdaj torej vidimo, da če velja invarianta na začetku iteracije t, bo veljala tudi na koncu te iteracije za $t+1$. Na koncu zadnje iteracije naše zanke torej velja invarianta tudi za b; takrat za vsakega potnika i velja $b_i \le b$, torej pade ta potnik pod točko (b), zato iz naše invariante zanj sledi, da je ta potnik že v nadstropju b_i. Tako smo se prepričali, da so na koncu našega postopka res vsi potniki v tistih nadstropjih, kamor so želeli priti.

15. Globalno segrevanje

(a) Naloga pravi, da so višine le cela števila od -100 do 3000, torej možnih višin ni prav veliko. Da bo naša rešitev bolj splošna, označimo najnižjo višino s h_{min}, najvišjo pa s h_{max}. Lahko si za začetek pripravimo tabelo, v kateri za vsako možno višino v preštejemo, koliko celic je na tej višini — recimo $s[v]$. Nato izračunamo kumulativne vsote teh števil, pa dobimo za vsako višino podatek o tem, koliko celic je na tej višini ali pod njo. S temi podatki pa potem zlahka odgovorimo na katerokoli poizvedbo (torej vprašanje, koliko mreže ostane nad vodo, če se gladina morja dvigne za q). Zapišimo ta postopek s psevdokodo:

for $v := h_{min}$ to h_{max} **do** $s[v] := 0$;
for $y := 0$ **to** $h-1$ **do for** $x := 0$ **to** $w-1$ **do** $s[h(x,y)] := s[h(x,y)] + 1$;
(* *Zdaj za vsako višino v vrednost $s[v]$ pove, koliko celic ima $h(x,y) = v$.* *)
for $v := h_{min}+1$ to h_{max} **do** $s[v] := s[v] + s[v-1]$;
(* *Zdaj za vsako višino v vrednost $s[v]$ pove, koliko celic ima $h(x,y) \le v$.* *)

za vsako poizvedbo q:
  if $q < h_{min}$ **then** $r := w \cdot h$
  else if $q \ge h_{max}$ **then** $r := 0$
  else $r := w \cdot h - s[q]$;
  odgovor na to poizvedbo je r;

Naš postopek torej porabi $O(w \cdot h)$ časa, da prebere vhodne podatke (višine vseh celic v mreži) in si pripravi tabelo s, nato pa le $O(1)$ časa za vsako poizvedbo.

(b) Če so višine lahko poljubna realna števila, jih ne moremo uporabljati kot indekse v tabelo s, kot smo to naredili pri rešitvi podnaloge (a). Namesto tega pa lahko

višine vseh celic mreže zložimo v tabelo, dolgo $w \cdot h$ elementov, in jih uredimo naraščajoče. Pri posamezni poizvedbi q lahko po tej tabeli z bisekcijo pogledamo, do katerega indeksa so v njej elementi z vrednostmi, manjšimi ali enakimi q. To nam pove število potopljenih celic pri dvigu gladine za q, s tem podatkom pa lahko odgovorimo na poizvedbo. Zapišimo še ta postopek s psevdokodo:

```
naj bo a tabela z w · h elementi;
for y := 0 to h − 1 do for x := 0 to w − 1 do a[y · w + x] := h(x, y);
uredi tabelo a naraščajoče;

za vsako poizvedbo q:
  if q < a[0] then
    odgovor na to poizvedbo je 0, pojdi na naslednjo;
  (* Sicer poženimo bisekcijo. *)
  ℓ := 0; r := w · h;
  while r − ℓ > 1:
    (* Tu velja a[ℓ] ≤ q < a[r]. Pri r = w · h si mislimo a[r] = ∞. *)
    m := ⌊(ℓ + r)/2⌋;
    if a[m] ≤ q then ℓ := m else r := m;
  (* Elementi a[0], . . . , a[r − 1] predstavljajo potopljene celice,
     elementi a[r], . . . , a[w · h − 1] pa celice, ki ostanejo nad vodo.
  odgovor na to poizvedbo je w · h − r;
```

Časovna zahtevnost tega postopka je $O(wh \log wh)$ za pripravo in urejanje tabele a, nato pa še $O(\log wh)$ za vsako poizvedbo (zaradi bisekcije). Za n poizvedb bi bil torej skupni čas $O((wh+n) \log wh)$. Če je poizvedb malo (in so vse znane vnaprej, tako da nam ni treba odgovarjati na vsako sproti), je boljša naslednja rešitev: uredimo vse poizvedbe naraščajoče in nato zlivajmo seznam poizvedb in tabelo a. Tako dobimo naslednji postopek:

```
pripravi tabelo a enako kot zgoraj;
r := 0;
za vsako poizvedbo q, v naraščajočem vrstnem redu:
  while r < w · h:
    if a[r] ≤ q then r := r + 1 else break;
  odgovor na to poizvedbo je w · h − r;
```

Tu smo se torej oprli na dejstvo, da če so poizvedbe urejene naraščajoče, se število potopljenih celic ves čas le povečuje. Skupna cena te rešitve je $O(wh \log wh)$ za pripravo tabele a (enako kot prej), $O(n \log n)$ za urejanje poizvedb in nato še $O(n + wh)$ za zlivanje (imamo zunanjo zanko po q, ki naredi n iteracij, notranja zanka po r pa naredi vsega skupaj največ wh iteracij, saj se r ves čas le povečuje).

(*c*) Za začetek si pripravimo urejen seznam vseh možnih višin, ki nastopajo na robovih naše kariraste mreže (torej v najbolj zgornji in najbolj spodnji vrstici ter v najbolj levem in najbolj desnem stolpcu); morebitne duplikate zavrzimo in recimo, da nam ostane seznam $a_1, \ldots, a_m$. (Pri vsaki višini si zapomnimo še, katere izmed celic na robu mreže so bile na tej višini.)

Te višine nam razdelijo realno os na intervale $(-\infty, a_1)$, $[a_1, a_2)$, $\ldots$, $[a_{m-1}, a_m)$, $[a_m, \infty)$, pri čemer za vsak interval velja, da je za vse q s tega intervala odgovor

na našo poizvedbo enak. Če torej za vse te intervale vnaprej izračunamo odgovor na poizvedbo, bomo potem lahko za poljuben q odgovorili na poizvedbo tako, da z bisekcijo (po seznamu a) pogledamo, na katerem intervalu leži q, in vrnemo odgovor za tisti interval.

Odgovore za vse možne intervale lahko učinkovito izračunamo z iskanjem v širino. Pripravimo si tabelo velikosti $w \times h$, v kateri bo za vsako celico označeno, ali je že poplavljena ali ne. Na začetku so vse celice suhe. Pojdimo zdaj po seznamu višin (v naraščajočem vrstnem redu); pri vsaki višini a_i označimo kot poplavljene vse tiste celice (x, y), ki ležijo na robu mreže in imajo višino a_i. Iz teh celic poplavimo preostanek mreže z iskanjem v širino, pri čemer se smemo seveda premakniti iz ene celice v sosednjo le, če ima ta sosednja enako ali nižjo višino in če je bila doslej še suha.

```
(* Označimo vse celice kot suhe. *)
for y := 0 to h − 1 do for x := 0 to w − 1 do s[x, y] := true;
r := 0; (* števec potopljenih celic *)
for i := 1 to m:
  Q := prazna vrsta;
  za vsako robno celico (x, y), ki ima višino a_i:
    s[x, y] := true; r := r + 1; dodaj (x, y) v Q;
  while Q ni prazna:
    naj bo (x, y) poljubna celica iz Q; pobriši jo iz Q;
    za vsako sosedo (x', y') celice (x, y):
      if h(x', y') > h(x, y) or not s[x', y'] then continue;
      s[x', y'] := true; r := r + 1; dodaj (x', y') v Q;
  r_i := r; (* to je število potopljenih celic, če se gladina dvigne
               za vsaj a_i in manj kot a_{i+1} *)
```

Zdaj lahko na poizvedbo za poljuben q odgovorimo takole: če je $q < a_1$, se ne potopi nič, sicer pa z bisekcijo poiščimo tisti i, za katerega je $a_i \leq q < a_{i+1}$ (na koncu si mislimo $a_{m+1} = \infty$), in odgovorimo, da se potopi r_i celic. Podobno kot pri (b) lahko tudi tu, če je poizvedb malo, namesto bisekcije pripravimo urejen seznam poizvedb in ga nato zlijemo s seznamom višin $a_1, \ldots, a_r$.

Časovna zahtevnost naše rešitve je $O((w + h) \log(w + h))$ za pripravo seznama višin robnih celic mreže, nato $O(wh)$ za iskanje v širino in izračun vseh možnih odgovorov; nato pa za odgovor na n poizvedb porabimo $O(n \log(w + h))$ časa, če uporabimo bisekcijo, ali $O(n \log n + w + h)$ časa, če poizvedbe uredimo in uporabimo zlivanje.

16. Drevo

Izberimo si poljubno točko r kot koren drevesa in vzpostavimo v drevesu relacijo med starši in otroki (starš točke u je tista točka p_u, ki je neposredna predhodnica točke u na poti od r do u).

Zastavimo si podproblem: recimo, da namesto celega drevesa gledamo le poddrevo s korenom pri u (to je torej poddrevo, ki ga tvorijo u in vsi njegovi potomci). Zdaj bi torej radi izbrali primeren nabor vozlišč le iz tega poddrevesa. Če izberemo u, to pomeni, da ne bomo smeli izbrati nobenega od njegovih otrok (ker so neposredno povezani z u); pri vsakem od poddreves, ki se začenjajo pri u-jevih otrocih,

bomo imeli torej podoben problem kot pri u, le z dodatno omejitvijo, da korena tistega poddrevesa ne smemo izbrati (ker je tisti koren u-jev otrok, u pa smo že izbrali).

Vidimo torej, da bo treba pri vsakem poddrevesu pravzaprav reševati dva podproblema, odvisno od tega, ali smemo izbrati tudi koren poddrevesa ali ne. Naj bo torej $f(u)$ največja možna vsota vrednosti vozlišč, ki jih je mogoče izbrati (ne da bi bili katerikoli dve izbrani vozlišči neposredno povezani) iz poddrevesa s korenom pri u, pri čemer smemo izbrati tudi vozlišče u samo; in naj bo $g(u)$ definiran na enak način, le z dodatno omejitvijo, da vozlišča u ne smemo izbrati.

Za liste, torej vozlišča brez otrok, je stvar preprosta: $f(u) = c_u$ in $g(u) = 0$. Pri notranjih vozliščih pa lahko rešujemo podprobleme takole: recimo, da gledamo notranje vozlišče u z otroki $u_1, \ldots, u_k$. Potem imamo

$$g(u) = \textstyle\sum_{i=1}^{k} f(u_i) \quad \text{in} \quad f(u) = \max\{g(u), c_u + \sum_{i=1}^{k} g(u_i)\}.$$

Z drugimi besedami, ko računamo $g(u)$, vemo, da u-ja ne smemo izbrati, zato njegove otroke lahko izberemo, torej pri vsakem otroku u_i vzamemo rešitev podproblema $f(u_i)$. Ko pa računamo $f(u)$, imamo dve možnosti: lahko u-ja ne izberemo in smo na istem kot pri $g(u)$, lahko pa u izberemo (s tem pridobi naš izbor vrednost c_u) in potem u-jevih otrok ne smemo izbrati, zato moramo pri vsakem od njih vzeti rešitev podproblema $g(u_i)$.

Po opisanih formulah lahko rešujemo podprobleme po drevesu od spodaj navzgor; na koncu je vrednost najboljšega izbora, po katerem sprašuje naloga, ravno $f(r)$ (pri čemer je r koren drevesa). Če bi hoteli tak izbor tudi izpisati, gremo lahko nazaj dol po drevesu in razmišljamo takole: izbor, ki je rešil podproblem $g(u)$, je lahko izpišemo tako, da izpišemo izbore, ki so rešili podrobleme $f(u_i)$ (za u-jeve otroke u_i); če pa hočemo izpisati izbor, ki je rešil podproblem $f(u)$, poglejmo najprej, če je $f(u) > g(u)$; če to ni res, mora biti $f(u) = g(u)$ in lahko izpišemo kar izbor, ki je rešil $g(u)$, sicer pa izpišimo vozlišče u in nato še izbore, ki so rešili podprobleme $g(u_i)$ (za vse u-jeve otroke u_i).

Razmislimo še o lažji različici naloge, pri kateri imajo vsa vozlišča enako vrednost. Zdaj torej lahko rečemo, da iščemo izbor s čim več vozlišči, pri čemer nobeni dve vozlišči ne smeta biti neposredno povezani.

Mislimo si nek list u našega drevesa; naj bo v njegov starš. Če v naš izbor sprejmemo v, potem ne smemo izbrati nobenega od v-jevih otrok, tudi u-ja ne. Če bi iz izbora zavrgli v in namesto njega dodali u, bi bil izbor še vedno veljaven (saj u, ker je list, ni neposredno povezan z nobenim drugim vozliščem, le z v, ta pa zdaj ni izbran); obenem pa bi se s tem odprla možnost, da bi v izbor mogoče dodali v-jevega starša ali pa kakšnega od v-jevih ostalih otrok (u-jevih bratov), ki jih prej (ko smo imeli v izboru še v) nismo mogli izbrati. S tem razmislekom lahko iz izbora zavržemo vsa taka vozlišča, ki so starši kakšnega lista; in ko je izbor brez njih, lahko vanj vsekakor sprejmemo vse liste.

Zdaj torej vemo, da listi našega drevesa gotovo pridejo v naš izbor, starši listov pa gotovo ne. Zdaj lahko vsa ta vozlišča v mislih pobrišemo iz drevesa in nadaljujemo z enakim razmislekom. Če neko vozlišče u, ki je *zdaj* list, ni izbrano, njegov starš (recimo mu v) pa je, lahko namesto v-ja izberemo u; težav z u-jevimi (pobrisanimi) otroki ne bo, kajti u-jevi otroci so bili lahko le starši listov (in zato neizbrani), ne

pa listi (kajti potem bi bil u starš lista in bi ga že pobrisali iz drevesa). Torej lahko v izbor sprejmemo vsa taka vozlišča, ki so zdaj listi, iz njega pa zavržemo vsa taka vozlišča, ki so zdaj starši listov. Nato liste in njihove starše spet pobrišemo in tako naprej, dokler ne pobrišemo celega drevesa.

17. Skupni geni

Recimo, da imamo vhodna niza s (dolžine n) in t (dolžine m); i-ti znak niza s označimo s s_i ali $s[i]$ (za $1 \le i \le n$), podniz od i-tega do (vključno) j-tega znaka pa označimo s $s[i..j]$.

Najdaljše skupno podzaporedje lahko računamo z dinamičnim programiranjem. Naj bo $f(i,j)$ dolžina najdaljšega skupnega podzaporedja nizov $s[1..i]$ in $t[1..j]$. Ločimo nekaj možnosti: (1) lahko da to skupno podzaporedje ne vsebuje znaka $s[i]$; tedaj je to tudi najdaljše skupno podzaporedje nizov $s[1..i-1]$ in $t[1..j]$, njegova dolžina pa je torej $f(i-1,j)$. (2) Mogoče skupno podzaporedje ne vsebuje znaka $t[j]$, torej je to tudi najdaljše skupno podzaporedje nizov $s[1..i]$ in $t[1..j-1]$, njegova dolžina pa je torej $f(i,j-1)$. (3) Če je $s[i] = t[j]$, je mogoče ta znak del skupnega podzaporedja, preostanek tega podzaporedja pa mora biti torej najdaljše skupno podzaporedje nizov $s[1..i-1]$ in $t[1..j-1]$, torej je dolgo $f(i-1,j-1)$. Tako smo dobili rekurzivno zvezo:

$$f(i,j) = \max\{f(i-1,j), f(i,j-1), [\![s_i = t_j]\!](1 + f(i-1,j-1))\}.$$

Robni primeri nastopijo, če je eden od nizov prazen: $f(i,0) = f(0,j) = 0$. Funkcijo f lahko računamo sistematično z dvema zankama po i in j, rezultate pa shranjujemo v tabelo:

```
for j := 0 to m do f[0,j] := 0;
for i := 1 to n:
    f[i,0] := 0;
    for j := 1 to m:
        f[i,j] := max{f[i-1,j], f[i,j+1]};
        if s_i = t_j then f[i,j] := max{f[i,j], 1 + f[i-1,j-1]};
```

Ko se ta postopek konča, imamo v $f(n,m)$ dolžino najdaljšega skupnega podzaporedja celotnih nizov s in t. S pomočjo vrednosti v tabeli f pa tudi ni težko rekonstruirati tega podzaporedja:

```
i := n; j := m; rezerviraj niz r dolžine f[n,m];
while f[i,j] > 0:
    if s_i = t_j then r[f[i,j]] := s_i; i := i - 1; j := j - 1;
    else if f(i-1,j) > f(i,j-1) then i := i - 1
    else j := j - 1;
```

Sprehajamo se torej po tabeli in kjer vidimo, da sta bila s_i in t_j enaka in je $f(i,j)$ nastal kot $1 + f(i-1,j-1)$, lahko vključimo ta znak (torej s_i oz. t_j) v naše podzaporedje. Na koncu tega postopka imamo v r podzaporedje, po katerem sprašuje naloga.

Opisani postopek porabi $O(nm)$ časa in tudi $O(nm)$ pomnilnika za tabelo f. Naloga pravi, da sta niza dolga po 10^6 znakov, torej je $O(nm)$ časa za silo še sprejemljivo, $O(nm)$ pomnilnika pa je že neugodno veliko (tabela $10^6 \times 10^6$ celih števil

bi bila dolga nekaj TB). Opazimo lahko, da vedno potrebujemo le dve vrstici tabele f naenkrat: ko računamo vrednosti $f[i,\cdot]$, potrebujemo pri tem le druge $f[i,\cdot]$ in še nekatere $f[i-1,\cdot]$, ne potrebujemo pa več vrednosti $f[i-2,\cdot]$ in tako naprej. Podobno je tudi pri drugem delu postopka (rekonstrukcija najdaljšega skupnega podniza), ki v vsakem trenutku gleda le dve vrstici (i-to in $(i-1)$-vo). Lahko bi pri računanju tabele f sproti odlagali vrstice naše tabele iz glavnega pomnilnika na disk in jih kasneje (v drugem delu postopka) brali z njega. Še vseeno pa je tako velika poraba prostora neugodna in je bolje, če razmislimo o boljši rešitvi.

Ena preprosta izboljšava je, da tabelo f v mislih razdelimo na „bloke“, sestavljene iz po približno $\sqrt{m}$ vrstic. Takih blokov bo torej približno $\sqrt{m}$. Pri računanju tabele f, ko izračunamo prvo vrstico nekega bloka, pozabimo celoten prejšnji blok, razen *njegove* prve vrstice. Tako torej poraba pomnilnika znaša le $O(n\sqrt{m})$, ker moramo v pomnilniku hraniti po eno vrstico vsakega od $\sqrt{m}$ blokov in celoten trenutni blok (ki ima $\sqrt{m}$ vrstic).

Lepo pri tem je, da lahko iz prve vrstice bloka kadarkoli ponovno izračunamo vse ostale vrstice tega bloka. To nam pride prav v drugem delu postopka, pri rekonstrukciji najdaljšega skupnega podniza. Ko se zaradi zmanjšanja i za 1 premaknemo iz trenutnega bloka v prejšnji blok, pozabimo trenutni blok (razen prve vrstice) in na novo izračunamo prejšnjega iz njegove prve vrstice. Časovna zahtevnost tega postopka je še vedno $O(nm)$, čeprav je konstantni faktor zdaj večji (vsak blok moramo izračunati po dvakrat), prostorska zahtevnost pa je le $O(n\sqrt{m})$.

To je za naš namen, ko sta n in m približno 10^6, verjetno že dovolj. Porabo pomnilnika lahko zmanjšamo še bolj, če si privoščimo manjše število večjih blokov; recimo $m^{1/3}$ blokov po $m^{2/3}$ vrstic. Pri prvem delu postopka, ki računa tabelo f, hranimo le trenutno in prejšnjo vrstico (i in $i-1$), trajno pa si zapomnimo prvo vrstico vsakega bloka. To je le $m^{1/3}$ vrstic. Pri drugem delu postopka, ko hočemo rekonstruirati najdaljši skupni podniz, si ne smemo privoščiti, da bi v pomnilniku hkrati hranili celoten trenutni blok (ker je prevelik, dolg kar $m^{2/3}$ vrstic), zato ga razdelimo na „podbloke“, recimo $m^{1/3}$ podblokov s po $m^{1/3}$ vrstcami. Ko blok računamo, hranimo le trenutni podblok, prejšnje pa sproti pozabljamo, razen prve vrstice vsakega podbloka. Ko pa moramo pozabiti nek blok, pozabimo tudi prve vrstice vseh njegovih podblokov, razen prve vrstice celega bloka. Prostorsko zahtevnost smo tako zmanjšali na $O(n\sqrt[3]{m})$, časovna zahtevnost pa je še vedno $O(nm)$, vendar s še večjim konstantnim faktorjem kot prej (vse računamo po trikrat).

Tako bi lahko komplicirali še naprej; veliko lepša in elegantnejša rešitev pa je Hirschbergov algoritem,[14] ki zmanjša porabo pomnilnika na samo $O(n+m)$, pri čemer poraba časa ostane $O(nm)$. Poleg funkcije f definirajmo še eno podobno: naj bo $g(i,j)$ dolžina najdaljšega skupnega podniza nizov $s[i..n]$ in $t[j..m]$. Podobno kot lahko f računamo po vrsticah za naraščajoče i, lahko tudi g računamo po vrsticah za padajoče i. Pri tem lahko stare vrstice sproti pozabljamo in tako porabimo le $O(m)$ prostora. Pa vzemimo $p = \lfloor m/2 \rfloor$ in izračunajmo na ta način vrstici $f(p,\cdot)$ in $g(p+1,\cdot)$. Za poljubno skupno podzaporedje w nizov s in t lahko razmišljamo takole: ker je w podzaporedje s-ja in ker je $s = s[1..p] + s[p+1..n]$, se mora dati tudi w razbiti na dva dela, w' in w'', pri čemer bo w' podzaporedje niza $s[1..p]$ in

[14] Za več o njem gl. npr. Wikipedijo *s. v.* Hirschberg's algorithm in Dan S. Hirschberg, *A linear space algorithm for computing maximal common subsequences*, CACM 18(6):341–343 (June 1975). Srečali ga bomo tudi na str. 191 pri rešitvi off-line naloge 2017.

w'' bo podzaporedje niza $s[p+1..n]$. In tudi obratno: ker je w podzaporedje t-ja in ker je $w = w'w''$, to pomeni, da se mora dati tudi t razbiti na dva podniza $t[1..q]$ in $t[q+1..m]$ tako, da bo w' podzaporedje niza $t[1..q]$ in w'' bo podzaporedje niza $t[q+1..m]$. Torej bo w' skupno podzaporedje nizov $s[1..p]$ in $t[1..q]$, podobno pa bo w'' skupno podzaporedje nizov $s[p+1..n]$ in $t[q+1..m]$. Ker hočemo čim daljši w, moramo vzeti tudi čim daljša w' in w'', to pa sta ravno $f(p,q) + g(p+1, q+1)$. Ker q-ja ne poznamo, bomo med vsemi možnimi q-ji vzeli tistega, ki dá največjo vsoto. Ker imamo vrstici $f(p,\cdot)$ in $g(p+1,\cdot)$ v celoti izračunani, to ne bo težko. Zdaj lahko naš algoritem poženemo rekurzivno, enkrat za $s[1..p]$ in $t[1..q]$ ter enkrat za $s[p+1..n]$ in $t[q+1..m]$, nato pa staknemo podzaporedji, ki ju dobimo pri obeh rekurzivnih klicih, in tako dobimo najdaljše skupno podzaporedje celotnih nizov s in t.

Robni primer te rekurzije nastopi, ko je eden od nizov dolg le 1 znak (takrat moramo le preveriti, ali se ta znak pojavi v drugem nizu) ali pa celo prazen (takrat je tudi najdaljši skupni podniz prazen).

Kakšna je časovna zahtevnost tega postopka? Recimo temu $T(n,m)$. Najprej porabimo $O(nm)$ časa, da izračunamo tabeli f in g do polovice ter poiščemo najboljši q, nato pa imamo še dva rekurzivna klica: skupaj torej $T(n,m) \le c{\cdot}nm + T(n/2, q) + T(n/2, m-q)$ za neko dovolj veliko konstanto c. Radi bi preverili, da za nek dovolj velik (vendar konstanten) d velja $T(n,m) \le d{\cdot}nm$. Počnimo to z indukcijo po n. Za robni primer, $n = 1$, je stvar trivialna, saj smo rekli, da moramo takrat le pregledati, če se $s[1]$ pojavi v nizu t, to pa traja $O(m)$ časa. Recimo zdaj, da naša indukcija velja do $n-1$; pri n imamo zdaj $T(n,m) \le c{\cdot}nm + d(n/2)q + d(n/2)(m-q) = (c + d/2)nm$, kar bo $\le d{\cdot}nm$, če vzamemo $d \ge 2c$. Torej je časovna zahtevnost tega postopka res le $O(nm)$.

18. Dvolični stavki

V mislih lahko dvolični stavek tvorimo tako, da imamo dva niza in da na vsakem koraku enemu od nizov pritaknemo neko besedo iz slovarja. (Cilj je seveda priti v stanje, ko sta oba niza enaka, pri tem pa nismo obeh dobili na enak način.) Stanje tega postopka lahko predstavimo s parom nizov (s,t). Daljšega izmed nizov s in t označimo z $D(s,t)$, njegovo dolžino pa z $d(s,t) := |D(s,t)| = \max\{|s|, |t|\}$. Ničesar ne izgubimo, če se dogovorimo še za naslednjo omejitev: podaljšali bomo vedno krajšega od obeh nizov.

Recimo, da je krajši od obeh nizov s, daljši pa t. To pomeni, da se mora t začeti na niz s, saj drugače na koncu gotovo ne bomo mogli dobiti dveh enakih nizov (ker niza le podaljšujemo s pritikanjem novih besed na desni strani, že postavljenih delov nizov pa ne moremo več spreminjati). Torej lahko rečemo, da je t oblike $t = su$. Ker je t daljši od s, se bo postopek nadaljeval tako, da bomo s-ju na koncu pritaknili neko besedo w. Če nočemo, da pride do neujemanja, se mora bodisi w začeti na u ali pa se mora u začeti na w. Vidimo torej, da na to, katere besede lahko zdaj pritaknemo s-ju, vpliva le u, ne pa tudi s sam. Zato je za nadaljevanje postopka pravzaprav dovolj, če stanje našega sistema opišemo le z u.

O tem nizu u, torej tistem delu t-ja, ki štrli čez s, lahko povemo še nekaj. Niz t je moral nastati tako, da smo neki zgodnejši različici t' tega niza pritaknili neko besedo w. Ta zgodnejši t' pa je gotovo krajši od s. O tem se prepričamo takole:

(1) Če smo v zadnjem koraku podaljšali t' v t, medtem ko je s ostal nespremenjen, to pomeni, da je moral biti t' krajši od s, saj vemo, da vedno podaljšamo krajšega od obeh nizov. (2) Če pa smo v zadnjem koraku podaljšali neko zgodnejšo različico s-ja (recimo s') v sedanji s, to pomeni, da je do podaljšanja iz t' v t prišlo enkrat prej; takratni različici s-ja recimo s''. Ker smo takrat podaljšali t' in ne s'', to pomeni, da je bil t' krajši od s''; slednji pa ni gotovo nič daljši od s', kasnejše različice tega niza; in s' je krajši od s. Tudi v tem primeru je torej moral biti t' krajši od s.

Vidimo torej, da v vsakem primeru velja $|t'| < |s|$. Za w pa to pomeni naslednje: ker je $t = t'w = su$, je $|w| = |t| - |t'| > |t| - |s| = |u|$. Niz u, za katerega t štrli čez s, je torej nujno krajši od w, to je od zadnje besede, ki smo jo pritaknili k nizu t. In ker se t obenem konča na w in na u, to pomeni, da se tudi w konča na u. Tako torej vidimo, da v poštev za u pridejo le sufiksi (končnice) besed iz našega slovarja.

Sestavimo zdaj graf stanj, v katerem je po ena točka za vsak možni u, torej za vsako končnico vsake besede našega slovarja. Med njimi naj bo tudi prazni niz, ki ga označimo z ε. Če uspemo doseči stanje, v katerem je $u = \varepsilon$, to pomeni, da je $t = su = s$, torej sta niza enaka in smo našli tak dvoličen stavek, kakršnega smo iskali.

Kaj se zgodi, če smo v stanju u in krajšega od naših dveh nizov podaljšamo z besedo w? Videli smo že, da je to dopustno le, če se w začne na u ali pa se u začne na w. (1) Če se w začne na u, pišimo $w = uw'$. Iz (s, t) smo prišli v $(sw, t) = (suw', su)$, torej niz w' zdaj pove, za koliko daljši niz štrli čez krajšega. Za dolžino pa velja $d(sw, t) = |suw'| = |t| + |w'| = d(s, t) + |w'|$; v primerjavi s prejšnjim stanjem, (s, t), se je torej d povečala za $|w'|$.

(2) Če pa se u začne na w, pišimo $u = wu'$. Iz (s, t) smo prišli v $(sw, t) = (sw, su) = (sw, swu')$, torej je zdaj niz u' tisti, ki pove, za koliko daljši niz štrli čez krajšega. Daljši niz je še vedno isti kot prej, torej t, zato se d nič ne spremeni: $d(sw, t) = |t| = d(s, t)$.

Dodajmo torej v naš graf naslednje povezave: za vsako stanje u in vsako besedo w našega slovarja, če je $w = uw'$, dodajmo v graf povezavo $u \to w'$ z oznako (labelo) w'; in če je $u = wu'$, dodajmo v graf povezavo $u \to u'$, njena oznaka pa naj bo ε (prazen niz). Za vsako povezavo definirajmo tudi dolžino, in sicer kot dolžino oznake te povezave.

Vsak sprehod po tem grafu zdaj predstavlja zaporedje korakov, pri katerem vsakič podaljšamo krajšega od obeh nizov z eno od besed iz slovarja in pri čemer nikoli ne nastane kakšno neujemanje med nizoma. Če staknemo oznake tako prehojenih povezav, dobimo niz, ki pove, kako se je pri tem zaporedju korakov podaljšal niz $D(s, t)$; če seštejemo dolžine teh povezav, pa dobimo podatek o tem, za koliko se je povečala dolžina $d(s, t)$.

Našo pot po grafu želimo končati v točki ε, saj ta predstavlja stanje, ko je $s = t$, torej ko smo sestavili en in isti niz na dva različna načina; in ta pot mora biti čim krajša (merjeno z vsoto dolžin povezav), da bo sestavljeni niz čim krajši. Kje pa naj našo pot začnemo? Ko začnemo tvoriti naš dvolični stavek, imamo dva prazna niza: $(\varepsilon, \varepsilon)$; enega od njiju podaljšamo z neko besedo w iz slovarja: dobimo (w, ε); v naslednjem koraku moramo podaljšati drugega (kajti ta je zdaj krajši od prvega) z neko besedo x iz slovarja: dobimo (w, x). To je smiselno le, če je $x \neq w$, saj sicer ta začetek ne bo nič pripomogel k nastanku dvoličnega stavka (ker se bosta obe različici

stavka začeli z isto besedo iz slovarja). Poleg tega mora se mora bodisi x začeti na w ali pa w na x, sicer bi nastopilo neujemanje, zaradi katerega na koncu ne bi mogli dobiti dveh enakih nizov. Naš graf lahko torej dopolnimo takole: dodajmo novo točko z, ki predstavlja začetek poti; nato pa za vsak par besed w in x iz slovarja, če je $w = xw'$, dodajmo povezavo od z do w' z oznako w. V tako dopolnjenem grafu moramo poiskati najkrajšo pot od z do ε, za kar lahko uporabimo na primer Dijkstrov algoritem.

Zapišimo naš postopek še s psevdokodo:

```
1  začnimo z grafom, ki vsebuje le točki z in ε;
2  za vsako besedo w iz slovarja:
3     za vsako razbitje w na dva kosa, w = ux:
4        dodaj v graf točko x (če je še ni v njem);
5        če je u v slovarju, dodaj v graf povezavo z → x z oznako w;
6  za vsako točko u našega grafa (razen z in ε):
7     za vsak začetek (prefiks) w niza u, ki je tudi beseda v slovarju:
8        naj bo u = wu'; dodaj v graf povezavo u → u' z oznako ε;
9     za vsako slovarsko besedo w, ki se začne na u:
10       naj bo w = uw'; dodaj v graf povezavo u → w' z oznako w';
11 dolžina vsake povezave naj bo enaka dolžini njene oznake;
12 v dobljenem grafu poišči najkrajšo pot od z do ε z Dijkstrovim algoritmom;
```

Slovar besed bi bilo koristno predstaviti z drevesom po črkah (*trie*); tako bomo lahko učinkovito preverjali, ali je nek niz v slovarju (v korakih 5 in 7), pa tudi poiskali vse slovarske besede, ki se začnejo na dani niz (v koraku 9).

Ko v tem grafu poiščemo najkrajšo pot od z do ε, je dolžina te poti (torej vsota dolžin povezav na njej) ravno odgovor, po katerem sprašuje naloga, torej dolžina najkrajšega dvoličnega stavka. Če hočemo tak stavek tudi izpisati, moramo le stakniti oznake povezav na naši poti. Če pa hočemo tudi rekonstruirati oba načina tvorbe našega dvoličnega stavka, se moramo sprehoditi po poti od začetka do konca in razmišljati takole:

- ko gremo po povezavi $z \to x$ z oznako w, to pomeni, da smo na začetku poti, ko sta oba niza prazna, in da je w oblike $w = ux$; enemu od nizov pritaknimo slovarsko besedo w, drugemu pa u;
- ko gremo po povezavi $u \to u'$ z oznako ε, to pomeni, da je u' sufiks u-ja — torej pišimo $u = wu'$ — in da moramo krajšemu od naših dveh nizov pritakniti slovarsko besedo w;
- sicer pa imamo povezavo $u \to w'$ z oznako w' in moramo krajšemu od naših dveh nizov pritakniti slovarsko besedo uw'.

Ko pridemo do konca poti, sta oba niza enaka in predstavljata naš dvolični stavek, med potjo pa smo torej tudi videli, kako moramo stikati slovarske besede, da pridemo do njega na oba načina.

19. Pristajalne ploščadi

Ploščadi si predstavljajmo kot vozlišča (točke) grafa — množico vseh ploščadi označimo z V. Izberimo si nek k in razdelimo množico ploščadi na dve disjunktni podmnožici glede na stopnjo vozlišča: tista s stopnjo $< k$ so *lahka* vozlišča V_L, ostala

pa so *težka* vozlišča V_T. Dve vozlišči sta soseda, če obstaja med njima direktna povezava. Za vsako vozlišče v vodimo ločen seznam lahkih in težkih sosedov ($N_L(v)$ in $N_T(v)$), poleg tega pa še število tistih lahkih sosedov vozlišča v, ki so trenutno v obratovanju (c_v).

Opazimo lahko, da je število težkih vozlišč omejeno. Recimo, da je v našem omrežju trenutno m direktnih povezav. Vsaka povezava ima dve krajišči, kar je skupno $2m$ krajišč. Vsako težko vozlišče pokrije vsaj k krajišč, torej je lahko težkih vozlišč največ $2m/k$. To je naprej $\leq 2q/k$, ker naloga pravi, da je skupno število sprememb in poizvedb le q, z vsako od sprememb pa se lahko doda največ eno direktno povezavo.

Ob spremembi stanja lahkega vozlišča (Vklop ali Izklop) primerno posodobimo števce c_v pri vseh sosedih tistega lahkega vozlišča; to ni drago, ker je vozlišče láhko in torej sosedov nima veliko (manj kot k).

Če se spremeni stanje težkega vozlišča, ni treba zaradi tega v naših podatkovnih strukturah spreminjati ničesar.

Ob poizvedbi (PreštejSosede) nad vozliščem v dobimo odgovor tako, da pregledamo težke sosede in preštejemo, koliko jih obratuje, nato pa prištejemo še c_v. To ni drago, ker težkih sosedov ni veliko (ker težkih vozlišč nasploh ni veliko — zgoraj smo videli, da jih je največ $2q/k$).

Ob dodajanju povezave (Poveži) se lahkó zgodi, da láhko vozlišče u postane težko; tedaj moramo iti po njegovih sosedih (ki jih imamo v seznamih $N_L(u)$ in $N_T(u)$) in pri vsakem u-jevem sosedu v premakniti u iz seznama v-jevih lahkih sosedov $N_L(v)$ v seznam v-jevih težkih sosedov $N_T(v)$, pa še zmanjšati c_v, če je bilo vozlišče u vklopljeno. To ni drago, kajti če je vozlišče u pravkar postalo težko, to pomeni, da ima zdaj natanko k sosedov.

Poleg tega moramo seveda pri dodajanju povezave med a in b dodati b-ja v enega od seznamov a-jevih sosedov ($N_L(a)$ ali $N_T(b)$, odvisno od tega, ali je b láhko ali težko vozlišče) in obratno; in če je b láhko ter v obratovanju, moramo povečati c_a (in obratno).

Vse opisane operacije lahko izvajamo zelo elegantno in učinkovito, če za predstavitev vseh $N_L(u)$ in $N_T(u)$ uporabimo dvojno povezane sezname (*doubly linked lists*) in imamo za vsako povezavo (u, v) le en zapis, ki je istočasno povezan v dva taka seznama (enega od seznamov u-jevih sosedov in enega od seznamov v-jevih sosedov).

Kako naj si izberemo k? Označimo število težkih vozlišč z n_t; videli smo, da je $n_t \leq 2q/k$. Videli smo tudi, da poizvedbe trajajo $O(n_t)$ časa, spremembe stanja in dodajanja povezav pa $O(k)$ časa. Želimo si torej, da bi bila k in n_t oba čim manjša, kar lahko dosežemo tako, da vzamemo $k \approx \sqrt{2q}$ (takrat bo tudi $n_t \approx k$).

Oglejmo si primer implementacije takšne rešitve v jeziku C++. Začnimo s predstavitvijo povezav:

```
struct Povezava;
struct Krajisce
{
  int t; Povezava *p, *n; // krajišče ter njegova prejšnja in naslednja povezava
  Krajisce(int T) : t(T), p(nullptr), n(nullptr) { }
};
struct Povezava
```

```
{
  Krajisce u, v;
  Povezava(int U, int V) : u(U), v(V) { }
  Krajisce &Kraj(int t) { return u.t == t ? u : v; }
  Krajisce &Drugo(int t) { return u.t == t ? v : u; }
};
```

Povezava je torej par krajišč, vsako krajišče pa hrani številko vozlišča (recimo t) in prejšnjo in naslednjo povezavo v seznamu t-jevih sosedov. Razred Povezava ima tudi dve metodi: Kraj vrne referenco na krajišče za točko t, Drugo pa na drugo krajišče (tisto, ki ni za točko t).

Vozlišče ni nič posebnega; vsebuje c_v, stopnjo, stanje (ali ploščad obratuje ali ne) in kazalca na začetek obeh seznamov sosedov ($N_L(v)$ in $N_T(v)$). Vozlišča bomo hranili v vektorju.

```
struct Vozlisce
{
  bool obratuje;
  int stopnja;          // število sosedov
  int c;                // število obratujočih lahkih sosedov
  Povezava *nl, *nt;    // seznama lahkih in težkih sosedov
  Vozlisce() : obratuje(false), stopnja(0), c(0), nl(nullptr), nt(nullptr) { }
};
#include <vector>
std::vector<Vozlisce> vozlisca;  int k;
```

Operaciji Vklop in Izklop sta si zelo podobni, zato bomo obe prevedli na en sam podprogram po imenu Sprememba. Ta popravi stanje vozlišča, nato pa, če je bilo láhko, gre po obeh njegovih seznamih sosedov in popravi c_v pri vseh sosedih:

```
void Sprememba(int a, bool obratuje)
{
  Vozlisce &A = vozlisca[a];
  if (A.obratuje == obratuje) return; // ni sprememb
  A.obratuje = obratuje;
  if (A.stopnja >= k) return; // a je težko vozlišče
  // Sicer je a lahek; pri njegovih sosedih popravimo števce obratujočih lahkih sosedov.
  for (int lahka = 0; lahka <= 1; lahka++)
  for (Povezava *p = (lahka) ? A.nl : A.nt; p; p = p->Kraj(a).n)
    vozlisca[p->Drugo(a).t].c += (obratuje ? 1 : -1);
}
void Vklop(int a) { Sprememba(a, true); }
void Izklop(int a) { Sprememba(a, false); }
```

PrestejSosede ima število vklopljenih lahkih sosedov že v polju c, nato pa se mora še sprehoditi po seznamu težkih sosedov in prešteti, koliko od njih je vklopljenih:

```
int PrestejSosede(int a)
{
  Vozlisce &A = vozlisca[a];
  int r = A.c; // Število obratujočih lahkih sosedov.
  // Preštejmo še obratujoče težke sosede.
  for (Povezava *p = A.nt; p; p = p->Kraj(a).n)
```

```
    if (vozlisca[p->Drugo(a).t].obratuje) r++;
  return r;
}
```

S podprogramom Povezi bo nekaj več dela. Preden si ga ogledamo, pripravimo pomožni podprogram PostaloTezko, ki ga bomo klicali, ko se vozlišče a spremeni iz lahkega v težko. Takrat se moramo sprehoditi po vseh njegovih sosedih (tako lahkih kot težkih) in pri vsakem sosedu b premakniti a iz $N_L(b)$ v $N_T(b)$.

```
void PostaloTezko(int a)
{
  Vozlisce &A = vozlisca[a];
  for (int lahka = 0; lahka <= 1; lahka++)
  for (Povezava *p = (lahka) ? A.nl : A.nt; p; p = p->Kraj(a).n)
  {
    Krajisce &b = p->Drugo(a);
    Vozlisce &B = vozlisca[b.t];
    if (A.obratuje) B.c--; // a ni več lahek obratujoč sosed b-ja (ker ni več lahek)
    // Pobrišimo to povezavo iz seznama b-jevih lahkih sosedov.
    if (B.nl == p) B.nl = b.n; else b.p->Kraj(b.t).n = b.n;
    if (b.n) b.n->Kraj(b.t).p = b.p;
    // Dodajmo jo v seznam b-jevih težkih sosedov.
    b.p = nullptr; b.n = B.nt; B.nt = p; if (b.n) b.n->Kraj(b.t).p = p;
  }
}
```

Podprogram Povezi najprej preveri, če bo kakšno od krajišč nove povezave zaradi nje postalo težko (ker se mu bo stopnja povečala s $k-1$ na k), in po potrebi pokliče PostaloTezko. Nato poveča stopnji krajišč in po potrebi poveča števca obratujočih lahkih sosedov (če je krajišče vklopljeno in je tudi po dodajanju nove povezave láhko, moramo povečati števec pri drugem krajišču in obratno). Nato pripravimo novo strukturo Povezava in jo pri vsakem vozlišču dodamo na začetek enega od seznamov njegovih sosedov (lahkih ali težkih, odvisno od tega, ali je drugo krajišče povezave láhko ali težko).

```
void Povezi(int a, int b)
{
  Vozlisce &A = vozlisca[a], &B = vozlisca[b];
  // Če bo kakšno vozlišče postalo težko, ga pri sosedih ustrezno premaknimo
  // iz seznamov lahkih v sezname težkih sosedov.
  if (A.stopnja == k - 1) PostaloTezko(a);
  if (B.stopnja == k - 1) PostaloTezko(b);
  // Popravimo stopnje in (če je treba) števce obratujočih lahkih sosedov.
  A.stopnja++; B.stopnja++;
  if (A.obratuje && A.stopnja < k) B.c++;
  if (B.obratuje && B.stopnja < k) A.c++;
  // Pripravimo novo povezavo.
  Povezava *P = new Povezava(a, b);
  // Dodajmo b v ustrezni seznam a-jevih sosedov.
  { Povezava *&glava = (B.stopnja < k) ? A.nl : A.nt;
  P->Kraj(a).n = glava; if (glava) glava->Kraj(a).p = P; glava = P; }
  // Dodajmo a v ustrezni seznam b-jevih sosedov.
```

```
  { Povezava *&glava = (A.stopnja < k) ? B.nl : B.nt;
  P->Kraj(b).n = glava; if (glava) glava->Kraj(b).p = P; glava = P; }
}
```

Naloge so sestavili: igra 2048, razporejanje študentov, knjižnica — Nino Bašić; delni izid, skupni geni, dvolični stavki — Tomaž Hočevar; drevo — Vid Kocijan; znajdi.se, drugi tir (A in B), globalno segrevanje — Jurij Kodre; peščena ura, polaganje plošč — Mitja Lasič; kodiranje — Mark Martinec in Janez Brank; pobeg, dvigalo — Mitja Trampuš; cenik, komplet — Miha Vuk; pristajalne ploščadi — Patrik Zajec.

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. Umor

- Če rešitev ne deluje pravilno v primeru, da se umorjeni ni pogovarjal z nikomer, naj se ji zaradi tega odšteje pet točk.
- Umorjenec lahko v pogovoru nastopa bodisi kot prvi bodisi kot drugi sogovornik. Rešitve, ki pravilno obravnavajo le eno od teh dveh možnosti, ne pa obeh, lahko dobijo največ 12 točk.
- Rešitvam, ki po nepotrebnem preberejo celoten vhodni seznam pogovorov v pomnilnik (namesto da bi jih brale in obdelovale sproti), naj se zaradi tega odšteje tri točke.
- Za morebitne manjše napake pri branju vhodnih podatkov naj se rešitvi odšteje največ dve točki.

2. Naraščajoče besede

- Rešitvam, ki po nepotrebnem preberejo v pomnilnik celotno vhodno besedilo (namesto da bi ga brale po znakih ali pa po vrsticah), naj se zaradi tega odšteje največ tri točke.
- Nekateri programski jeziki imajo v knjižnici že pripravljene funkcije za razbijanje nizov na besede in podobne operacije. Nič ni narobe, če rešitev takšne funkcije uporablja (če seveda na koncu deluje v skladu z zahtevami naloge).
- Naloga ne določa eksplicitno, ali to, da so črke v besedi urejene naraščajoče, zahteva strogo urejenost ali pa sme biti po več zaporednih črk tudi enakih. Obe tidve interpretaciji sta sprejemljivi in rešitev lahko dobi vse točke ne glede na to, za katero od njiju se odloči.
- Če bi rešitev naredila kakšne neupravičene predpostavke o obliki vhodnih podatkov, ki ji delo občutno olajšajo (na primer to, da je vsaka beseda v svoji vrstici), naj se ji zaradi tega odšteje šest točk.
- Če rešitev ne deluje pravilno v primeru, ko je najdaljša naraščajoča beseda čisto na koncu vhodnega besedila (npr. zaradi kakšnih nerodno zastavljenih ustavitvenih pogojev v zankah), naj se ji zaradi tega odšteje tri točke.

3. Popularni dnevi

- Naloga pravi, naj bo postopek čim bolj učinkovit. Rešitve, ki imajo časovno zahtevnost $O(n \cdot k)$ namesto le $O(n)$, naj se zaradi tega odšteje štiri točke.
- Naivni rešitvi, ki poskuša vedno gledati (največ) $k/2$ dni pred in za opazovanim dnevom d, torej se ne zaveda tega, da mora npr. takrat, ko je $d < k/2$, vzeti več dni za d-jem kot pred njim), naj se zaradi tega odšteje osem točk. Če se rešitev tega problema zaveda, vendar takšnih robnih primerov ($d < k/2$ in $d > n-1-k/2$) ne obravnava pravilno, naj se ji zaradi tega odšteje štiri točke.

- Naloga pravi, da je dan popularen, če je obisk v njem večji od povprečja sosednjih dni. To pomeni, da če je obisk povprečju le enak, dan še ni popularen. To je razvidno tudi iz primera na koncu besedila (tretji dan ni popularen). Rešitvi, ki bi neupravičeno razglasila za popularne tudi take dni, pri katerih je obisk le enak povprečju, ne pa večji od njega, naj se zaradi tega odšteje dve točki.

- Naš primer rešitve preverja, ali je obisk nekega dne večji od povprečja, s pogojem oblike „`if (vsota < obisk[d] * k)`“. Enako dobro je tudi, če rešitev izračuna povprečje eksplicitno in preverja „`if (vsota / k < obisk[d])`“ ipd.

4. Sindikat

- Od rešitve pričakujemo predvsem, da se zaveda, da je treba plače popravljati od spodaj navzgor, in da zna izračunati nove plače v skladu z zahtevami naloge.

- Če rešitev sicer pravilno računa popravljene plače, ne izračuna pa pravilno vsote vseh zvišanj plač (ta vsota je eden od rezultatov, ki jih zahteva besedilo naloge), naj se ji zaradi tega odšteje tri točke.

- Podobno tudi, če rešitev sicer pravilno računa popravljene plače, vendar ne izpiše imen tistih zaposlenih, ki se jim je plača zvišala (oz. jih nekako drugače označi ali identificira), naj se ji zaradi tega odšteje tri točke.

- Učinkovitost postopka pri tej nalogi ni tako zelo pomembna in rešitev, ki ima časovno zahtevnost $O(n^2)$ namesto $O(n)$, lahko kljub temu dobi do 18 točk (če je drugače pravilna). Ravno tako tudi ni mišljeno, da bi se rešitev veliko ukvarjala s podrobnostmi tega, kako predstaviti drevo v pomnilniku (niti ni kakšne posebne nuje po tem, da bi bilo drevo predstavljeno še s čim drugim kot z nekaj tabelami, kot nakazuje že besedilo naloge).

- Rešitev, ki dvigne plače tako, da ima po novem vsak zaposleni vsaj 100 € višjo plačo od svojih podrejenih, vendar ti dvigi niso najmanjši možni, naj dobi največ 10 točk.

- Rešitev, ki poskuša plačo komu tudi znižati, ali pa ki spremeni plače tako, da ne zagotavlja, da bo imel vsak zaposleni po novem vsaj 100 € višjo plačo od svojih podrejenih, naj dobi največ 5 točk.

5. 3-d labirint

- Pri tej nalogi ni veliko poudarka na učinkovitosti, vendarle pa, če bi kakšna rešitev zaradi kakšne hudo nespametno implementirane rekurzije ali česa podobnega imela eksponentno časovno zahtevnost, naj dobi največ 7 točk (če je drugače pravilna).

- Ni pomembno, v kakšnem vrstnem redu rešitev preiskuje mrežo (npr. v širino, v globino ali še kako drugače), niti ne pričakujemo od nje, da bo skušala od najvišjega do najnižjega stebra (ali obratno) priti po najkrajši poti oz. ne da bi pri tem preiskala še vse druge stebre, ki so dosegljivi z njiju.

- Rešitvi, ki pomotoma dovoli neposreden premik z enega stebra na drugega tudi v primerih, ko imata njuni celici skupno le eno oglišče, ne pa stranice, naj se zaradi tega odšteje tri točke.

- Rešitvi, ki dela napake na robovih mreže, npr. ker pozabi, da tam celica določenih sosed sploh nima, naj se zaradi tega odšteje dve točki.

- Rešitvi, ki ne preverja pogoja, da je korak z enega stebra na sosednjega mogoč le, če se njuni višini razlikujeta največ za 1, ali pa ga preverja narobe, naj se zaradi tega odšteje dve točki.

Težavnost nalog

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

Naloga	Kam bi sodila po težavnosti na državnem tekmovanju ACM
1. Umor	lažja do srednja naloga v prvi skupini
2. Naraščajoče besede	srednje težka naloga v prvi skupini
3. Popularni dnevi	težka v prvi ali lažja do srednja naloga v drugi skupini
4. Sindikat	srednja do težja naloga v drugi skupini
5. 3-d labirint	srednja do težja v drugi ali lahka v tretji skupini

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

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

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.

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 351) in smo jih letos podelili enajst. Srebrna priznanja pa se podeljujejo po podobnih kriterijih kot pred leti pohvale; prejmejo jih tekmovalci, ki ustrezajo naslednjim trem pogojem: (1) tekmovalec ni dobil zlatega priznanja; (2) je boljši od vsaj polovice tekmovalcev v svoji skupini; in (3) je tekmoval v prvi ali drugi skupini in dobil vsaj 20 točk ali pa je tekmoval v tretji skupini in dobil vsaj 80 točk. Namen srebrnih priznanj je, da izkažemo priznanje in spodbudo vsem, ki se po rezultatu prebijejo v zgornjo polovico svoje skupine. Podobno prakso poznajo tudi na nekaterih mednarodnih tekmovanjih; na primer, na mednarodni računalniški olimpijadi (IOI) prejme medalje kar polovica vseh udeležencev. Poleg zlatih in srebrnih priznanj obstajajo tudi bronasta, ta pa so dobili najboljši tekmovalci v okviru šolskih tekmovanj (letos smo podelili 134 bronastih priznanj).

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

PRVA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Jernej Leskovšek	4	STPŠ Trbovlje	20	20	17	20	18	95
1Z	2	Gregor Kržmanc	1	Gimnazija Vič	18	18	18	20	20	94
2Z	3	Jon Mikoš	2	Gimnazija Vič	20	15	16	20	15	86
2Z	4	Lenart Bučar	1	Gimnazija Bežigrad	16	17	17	20	15	85
3S	5	Kevin Šarlah	3	ŠC Celje, SŠ za KER	12	20	17	20	13	82
3S		Miha Krajnc	2	STPŠ Trbovlje	19	12	18	17	16	82
S	7	Žiga Deutschbauer	3	ERŠ Velenje	7	19	15	20	20	81
S	8	Jernej Grlj	3	Škof. klas. gimn. Lj.	15	18	20	20	7	80
S		Patrik Žnidaršič	9	ZRI	10	18	17	20	15	80
S	10	Gregor Gabrovšek	4	Škof. klas. gimn. Lj.	20	17	10	20	12	79
S		Uroš Šmajdek	4	ŠC Novo mesto, SEŠTG	20	17	9	18	15	79
S	12	Jakob Salmič	3	ŠC Kranj, STŠ Kranj	7	20	18	20	13	78
S		Job Petrovčič	1	Gimnazija Vič	20	2	17	20	19	78
S		Luka Pepelnjak	2	Gimnazija Vič	18	10	19	20	11	78
S		Matevž Miščič	2	Gimnazija Vič	20	12	14	20	12	78
S	16	Luka Laharnar	3	STPŠ Trbovlje	19	7	16	20	15	77
S	17	Luka Dragar	3	Vegova Ljubljana	13	15	18	17	13	76
S		Miha Zupan	2	Gimnazija Bežigrad	12	11	15	20	18	76
S	19	Jaka Smrekar	2	Gimnazija Vič	10	18	10	20	17	75
S		Domen Slemenšek	4	I. gimnazija v Celju	12	8	18	18	19	75
S		Iztok Bajcar	1	Gimnazija Vič	15	10	10	20	20	75
S		David Žele	4	ŠC Celje, Gimn. Lava	16	12	12	16	19	75
S	23	Andraž Pevcin	2	ŠC Celje, Gimn. Lava	20	15	10	18	11	74
S	24	Matija Kotrle	4	STŠ Koper	12	18	15	18	10	73
S	25	Veno Lan Banovšek	3	Vegova Ljubljana	12	17	12	19	12	72
S	26	Filip Štamcar	7	ZRI	11	18	13	19	9	70
S		Martin Prelog	3	ŠC Kranj, STŠ Kranj	12	12	15	20	11	70
S		Matic Conradi	2	I. gimnazija v Celju	7	18	20	15	10	70
S		Luka Železnik	2	II. gimnazija Maribor	16	8	18	20	8	70
S	30	Domen Kržmanc	3	Gimnazija Vič	15	2	20	20	11	68
S		Jure Pustoslemšek	3	ŠC Celje, SŠ za KER	20	8	5	20	15	68
S	32	Tim Retelj	2	Vegova Ljubljana	20	8	16	13	10	67
S	33	Gregor Gajič	1	Gimnazija Bežigrad	5	18	17	18	8	66
S		Boštjan Planko	3	ŠC Celje, SŠ za KER	10	14	11	20	11	66
S		Jan Hribar	2	Vegova Ljubljana	9	18	8	17	14	66
S		Matic Hrastelj	4	STPŠ Trbovlje	11	12	8	20	15	66
S	37	Leon Samotorčan	3	Gimnazija Bežigrad	18	12	6	20	9	65
S		Vladimir Smrkolj	1	Gimnazija Bežigrad	15	15	16	19	0	65
S		Gregor Brantuša	3	II. gimnazija Maribor	0	17	18	20	10	65
S		Franc Klavž	4	ERŠ Velenje	12	0	16	20	17	65

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
S	41	Rok Šerak	3	STPŠ Trbovlje	9	12	10	17	16	64
S		David Kraševec	3	Vegova Ljubljana	15	7	12	20	10	64
S		Domen Ramšak	3	ERŠ Velenje	16	16	8	16	8	64
S		Jaka Metelko	2	ŠC N. mesto, SEŠTG	7	10	17	16	14	64
S	45	Žan Rogan	3	SPTŠ Murska Sobota	17	10	10	14	12	63
S		Jani Kaukler	4	SERŠ Maribor	16	16	10	16	5	63
S	47	Gregor Rakef	2	Gimnazija Jesenice	8	8	16	20	10	62
S	48	Jaša Žnidar	3	ŠC Kranj, Str. gimn.	15	6	6	20	14	61
S	49	Domen Ogorevc	9	ZRI	8	3	10	19	20	60
S		Maj Zirkelbach	4	ŠC N. mesto, SEŠTG	20	12	9	14	5	60
S	51	Gregor Rant	4	ŠC Kranj, STŠ Kranj	13	14	12	10	10	59
S		Miha Frangež	2	SERŠ Maribor	10	9	8	20	12	59
S		Andraž Sivec	3	Škof. klas. gimn. Lj.	15	18	7	10	9	59
S		Domen Vilar	3	ŠC Kranj, Str. gimn.	12	10	17	12	8	59
S		Jon Selič	4	ŠC Celje, Gimn. Lava	16	5	15	17	6	59
S		Nejc Ačkun	4	STPŠ Trbovlje	14	7	10	20	8	59
	57	Aleš Kolar	3	SPTŠ Murska Sobota	2	8	17	17	14	58
		Žan Bajuk	3	Gimnazija Vič	20	0	10	20	8	58
	59	Lucian Semprimožnik	2	I. gimnazija v Celju	12	13	5	18	9	57
		Gašper Irman	2	ERŠ Velenje	15	10	10	18	4	57
	61	David Ošlaj	1	Škof. klas. gimn. Lj.	12	8	10	20	4	54
		Boris Pirečnik	3	ERŠ Velenje	20	0	9	17	8	54
	63	Andreja Kernc	4	ŠC N. mesto, SEŠTG	14	7	8	16	8	53
		Urban Matko	4	SŠ Domžale	7	14	9	18	5	53
		Boštjan Mokrin	3	ŠC Nova Gorica, ERŠ	2	14	15	12	10	53
		Aleksandar Georgiev	2	Vegova Ljubljana	14	10	11	8	10	53
		Jakob Kreft	2	Gimnazija Poljane	15	4	17	17	0	53
		Jan Hrastnik	1	Gimnazija Vič	12	6	7	20	8	53
		Matjaž Ciglič	4	Gimnazija Vič	0	8	20	18	7	53
	70	Blaž Košir	2	Gimnazija Vič	5	10	15	19,5	3	52,5
	71	Nejc Šuklje	2	ŠC N. mesto, SEŠTG	15	8	7	10	12	52
		Matej Zečiri	2	STPŠ Trbovlje	3	6	9	20	14	52
		Tilen Ravnak	4	ŠC Celje, Gimn. Lava	5	6	10	20	11	52
	74	Janez Koprivec	1	Gimnazija Vič	12	0	14	20	5	51
	75	Kristijan Petrič	1	ZRI	13	2	15	15	5	50
		Matevž Rom	3	ŠC N. mesto, SEŠTG	10	7	8	18	7	50
	77	David Grah	4	SPTŠ Murska Sobota	5	0	15	20	9	49
	78	Aleksander Piciga	1	Gimnazija Vič	0	3	11	20	14	48
		Samo Debeljak	2	Gimnazija Vič	5	2	14	20	7	48
		Žan Koren Kern	2	STPŠ Trbovlje	3	8	8	17	12	48

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
81	Aljaž Žel	3	II. gimnazija Maribor	15	8	9	5	10	47
	Matic Jeseničnik	3	ERŠ Velenje	7	6	16	18	0	47
83	Luka Tovornik	1	Gimnazija Vič	1	8	11	16	10	46
84	Tomaž Hrovat	3	ŠC Novo mesto, SEŠTG	16	8	12	0	9	45
	Juš Mirtič	2	Gimnazija Bežigrad	8	5	8	12	12	45
	Igor Kepe	3	SERŠ Maribor	20	6	19	0	0	45
	Gašper Žajdela	4	Škof. klas. gimn. Lj.	12	12	5	4	12	45
88	Luka Četina	4	ŠC Celje, Gimn. Lava	20	12	12	0	0	44
	Simon Sambolec	2	ŠC Ptuj, ERŠ	5	0	11	20	8	44
90	Blaž Bogar	2	SPTŠ Murska Sobota	20	0	3	20	0	43
	Andraž Žnidar	4	Gimnazija Kranj	8	0	5	20	10	43
	Aljoša Kalacanović	2	Gimnazija Vič	13	4	13	7	6	43
	Sebastjan Tkavc	4	I. gimnazija v Celju	12	8	11	11	1	43
94	Jani Suban	3	STŠ Koper	16	18	8	0	0	42
	Miha Komatar	4	SŠ Domžale	0	8	12	16	6	42
96	Jan Lupše	1	Gimnazija Jesenice	8	2	7	19	5	41
97	Luka Vrečar	3	Škof. klas. gimn. Lj.	5	15	8	2	8	38
98	Žan Hozjan	2	SPTŠ Murska Sobota	12	12	10	1	1	36
99	Timotej Petrovčič	2	Gimnazija Vič	8	2	5	10	10	35
100	Andraž Adamič	1	II. gimnazija Maribor	12	12	7	3	0	34
	Lucija Tomc	1	Škof. klas. gimn. Lj.	5	8	7	10	4	34
	Rebeka Backer	4	STŠ Koper	11	0	5	14	4	34
	Oskar Vavtar	2	Gimn. Poljane + ZRI	5	0	8	18	3	34
104	Frenk Dragar	3	I. gimnazija v Celju	7	0	10	14	0	31
105	Miha Gošte	3	STPŠ Trbovlje	16	3	3	3	5	30
106	Pia Golob	1	II. gimnazija Maribor	13	8	5	0	3	29
107	Matija Legat	3	Gimnazija Jesenice	3	0	9	10	6	28
	Anže Mihevc	4	Gimnazija Šentvid	12	6	8	2	0	28
109	Žan Mencigar	2	SPTŠ Murska Sobota	2	5	10	6	2	25
110	Luka Skeledžija	1	Gimnazija Vič	12	0	10	0	0	22
111	Jure Tič	4	Gimnazija Vič	2	0	11	5	3	21
112	Anamarija Hauptman	1	Škof. klas. gimn. Lj.	2	0	2	7	2	13
113	Klemen Ledinek Grm	3	ŠC Celje, SŠ za KER	0	0	5	5	0	10
114	Urška Eržen	2	Zavod sv. Frančiška Saleškega, Gimnazija Želimlje	5	0	2	1	0	8

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Žiga Patačko Koderman	4	Gimnazija Vič	5	20	20	18	19	82
2Z	2	Luka Jevšenak	3	Gimnazija Velenje	15	10	17	12	20	74
2Z	3	Marko Kužner	4	SERŠ Maribor	15	19	16	12	10	72
3Z	4	Vid Drobnič	4	Gimnazija Vič	10	18	20	20	0	68
3S	5	Aljaž Kolar	3	ŠC Kranj, Str. gimn.	5	12	18	20	10	65
3S	6	Martin Dagarin	3	Vegova Ljubljana	15	14	18	16	1	64
S	7	Jakob Pogačnik Souvent	1	Gimnazija Vič	15	13	18	16	0	62
S		Žiga Klemenčič	4	Vegova Ljubljana	10	18	20	9	5	62
S	9	David Murko	4	ŠC Ptuj, ERŠ	13	15	18	14	1	61
S	10	Rok Strah	3	Vegova Ljubljana	5	20	18	12	5	60
S		Tine Žnidaršič	4	Vegova Ljubljana	15	9	18	18	0	60
S	12	Marko Hostnik	3	Gimnazija Bežigrad	15	12	19	8	5	59
S	13	Gašper Golob	3	Vegova Ljubljana	15	5	19	10	8	57
S	14	Matija Kocbek	1	I. gimnazija v Celju	10	10	20	16	0	56
S	15	Urh Robič	1	Škof. klas. gimn. Lj.	0	10	14	12	18	54
	16	Žan Bizjak	4	Vegova Ljubljana	15	16	19	0	3	53
		Aljaž Zakošek	4	I. gimnazija v Celju	15	14	17	7	0	53
	18	David Grabnar	3	Vegova Ljubljana	12	5	18	11	5	51
	19	Jan Vasiljević	2	Gimnazija Tolmin	0	16	17	17	0	50
	20	Maj Fontana Korošec	2	Sred. ekon. šola Maribor	0	20	16	13	0	49
	21	Miha Pompe	2	ZRI in Gimnazija Vič	15	5	15	12	0	47
	22	Lan Vukušič	2	Gimnazija Tolmin	15	13	13	5	0	46
	23	Gašper Čopi	2	Gimnazija Tolmin	12	4	17	7	0	40
	24	Gregor Kovač	1	ZRI	15	10	6	5	0	36
	25	David Rozman	3	ZRI	15	10	10	0	0	35
	26	Jakob Zmrzlikar	3	Gimnazija Vič	0	5	17	8	2	32
	27	Blaž Novak	3	ŠC Ravne, SŠ Ravne	12	10	4	3	0	29
	28	Mark Valentin Vovk	4	Gimnazija Poljane	5	6	15	1	0	27
	29	Mihael Berčič	3	Gimnazija Bežigrad	0	10	5	10	0	25
	30	Miha Breznik	3	ŠC Ravne, SŠ Ravne	8	3	8	2	0	21
	31	Antonio Žibert	3	ŠC Ravne, SŠ Ravne	0	10	5	0	0	15
	32	Denis Krajnc	4	ŠC Ravne, SŠ Ravne	0	1	5	0	0	6
	33	Klemen Herman	3	ŠC Ravne, SŠ Ravne	0	0	0	0	0	0

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Žiga Željko	4	Gimn. Bežigrad + ZRI	100	87	90	57	57	391
1Z	2	Urban Duh	3	II. gimnazija Maribor	100	94	94		50	338
2Z	3	Tim Poštuvan	3	ZRI	100		50		97	247
3S	4	Mitja Žalik	4	II. gimnazija Maribor	97	11	57		30	195
3S	5	Gregor Kikelj	3	ŠC N. mesto, SEŠTG	100	2	21	0	71	194
3S	6	Mihail Denkovski	4	II. gimnazija Maribor	97		21	0	50	168
S	7	Matej Marinko	4	Gimnazija Vič	97	7	27		30	161
S	8	Blaž Zupančič	3	Škof. klas. gimn. Lj.	100	60			0	160
S	9	Luka Govedič	3	II. gimnazija Maribor	94	0	41	0		135
S	10	Jakob Schrader	9	ZRI	100		0			100
	11	Bor Grošelj Simić	2	ZRI	94					94
	12	Martin Peterlin	4	Vegova Lj. + ZRI	34	7	27			68
	13	Tevž Lotrič	1	ZRI	17	7	10		30	64
	14	Miha Mitič	3	Gimnazija Kranj	18					18
	15	Matic Rašl	4	ŠC Ptuj, ERŠ			10			10
	16	Adrian Mladenić Grobelnik	9	ZRI					1	1
	17	Urh Primožič	2	Škof. klas. gimn. Lj.	0				0	0
		Peter Berkopec	3	Škof. klas. gimn. Lj.						0
		Jaka Mele	1	ZRI						0
		Jon Kuhar	3	Gimnazija Kranj	0					0

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Jernej Leskovšek	telefon Samsung Galaxy S7 in ovitek
1	1	Gregor Kržmanc	telefon Samsung Galaxy A5 in ovitek
1	2	Jon Mikoš	telefon Samsung Galaxy A5 in ovitek
1	2	Lenart Bučar	telefon HTC Desire 825 in ovitek
1	3	Kevin Šarlah	telefon HTC Desire 825 in ovitek
1	3	Miha Krajnc	miška Logitech G502 Proteus Spectrum RGB
2	1	Žiga Patačko Koderman	telefon Samsung Galaxy S7 in ovitek Dasgupta *et al.: Algorithms*
2	2	Luka Jevšenak	telefon Samsung Galaxy A5 in ovitek Dasgupta *et al.: Algorithms*
2	2	Marko Kužner	telefon Samsung Galaxy A5 in ovitek Dasgupta *et al.: Algorithms*
2	3	Vid Drobnič	telefon Samsung Galaxy A5 in ovitek
2	3	Aljaž Kolar	telefon HTC Desire 825 in ovitek
2	3	Martin Dagarin	miška Logitech G502 Proteus Spectrum RGB
3	1	Žiga Željko	telefon Samsung Galaxy S7 in ovitek Raspberry Pi 3 model B Cormen *et al.: Introduction to Algorithms*
3	1	Urban Duh	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	3	Mitja Žalik	telefon Samsung Galaxy A5 in ovitek
3	3	Gregor Kikelj	telefon HTC Desire 825 in ovitek
3	3	Mihail Denkovski	telefon HTC Desire 825 in ovitek
Off-line naloga — Volilna območja			
	1	Uroš Koritnik	Raspberry Pi 3 model B
	4	Tomaž Martinčič	Raspberry Pi 3 model B

SODELUJOČE ŠOLE IN MENTORJI

II. gimnazija Maribor	Aleksander Kelenc, Mitja Osojnik, Mitja Žalik
Gimnazija Bežigrad	Andrej Šuštaršič, Jurij Železnik
Gimnazija Jesenice	Marko Kikelj
Gimnazija Kranj	Zdenka Vrbinc
Gimnazija Poljane	Janez Malovrh, Boštjan Žnidaršič
Gimnazija Šentvid	Nastja Lasič
Gimnazija Tolmin	Jernej Cvek, Lučka Uršič
Gimnazija Vič	Klemen Bajec, Alenka Krapež, Marina Trost
I. gimnazija v Celju	Branko Bezgovšek, Luka Zlatečan
Srednja elektro-računalniška šola Maribor (SERŠ)	Vida Motaln, Slavko Nekrep, Branko Potisk
Srednja ekonomska šola Maribor	Mitja Osojnik
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Igor Kutoš, Dominik Letnar, Karel Maček
Srednja šola Domžale	Tadej Trinko
Srednja tehniška in poklicna šola Trbovlje (STPŠ)	Uroš Ocepek
Srednja tehniška šola Koper (STŠ)	Andrej Florjančič
Škofijska klasična gimnazija Šentvid	Helena Medvešek, Matevž Poljanc, Matej Tomc
Šolski center Celje, Gimnazija Lava	Karmen Kotnik
Šolski center Celje, Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Dušan Fugina
Šolski center Kranj, Srednja tehniška šola	Miha Baloh, Aleš Hvasti
Šolski center Kranj, Strokovna gimnazija	Gašper Strniša
Šolski center Nova Gorica, Elektrotehniška in računalniška šola (ERŠ)	Tomaž Mavri, Boštjan Vouk

Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija (SEŠTG)	Albert Zorko, Simon Vovko
Šolski center Ptuj, Elektro in računalniška šola (ERŠ)	Marjan Čeh, David Drofenik, Zoltan Sep, Franc Vrbančič
Šolski center Ravne na Koroškem, Srednja šola Ravne	Gorazd Geč, Zdravko Pavleković
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Miran Zevnik
Šolski center Velenje, Gimnazija Velenje	Miran Zevnik
Vegova Ljubljana	Marko Kastelic, Nataša Makarovič, Darjan Toth
Zavod sv. Frančiška Saleškega, Gimnazija Želimlje	Benjamin Tomažič
Zavod za računalniško izobraževanje (ZRI), Ljubljana	

REZULTATI CEOI 2017

Ker smo letos organizirali srednjeevropsko računalniško olimpijado (CEOI 2017) pri nas v Ljubljani, objavljamo v našem biltenu še rezultate te olimpijade. Naloge so na str. 35–51, rešitve pa na str. 103–128.

Poleg srednjeevropskih držav je na CEOI 2017 sodelovalo tudi nekaj gostujočih delegacij od drugod. Sodelujoče države so bile tako naslednje: Avstrija (AUT), Azerbajdžan (AZE), Češka (CZE), Gruzija (GEO), Hrvaška (HRV), Madžarska (HUN), Italija (ITA), Izrael (ISR), Nemčija (DEU), Poljska (POL), Romunija (ROU), Slovaška (SVK), Slovenija (SVN) in Švica (CHE).

				Točke (po nalogah in skupaj)						
Medalja	Mesto	Ime	Država	Enosmerne ceste	Zanesljiva stava	Mišolovka	Gradnja mostov	Palindr. razdelitve	Lov	$\sum$
Z	1	Anadi Agrawal	POL	100	100	100	100	100	100	600
Z		Andrei Popa	ROU	100	100	100	100	100	100	600
Z		Giorgi Kldiashvili	GEO	100	100	100	100	100	100	600
Z	4	Mariusz Trela	POL	100	100	65	100	100	100	565
Z	5	Costin-Andrei Oncescu	ROU	100	100	45	100	100	100	545
Z		Tamio-Vesa Nakajima	ROU	100	100	45	100	100	100	545
S	7	Josip Klepec	HRV	100	100	65	100	100	60	525
S	8	Aleksandre Khokhiashvili	GEO	100	100	100	100	20	100	520
S	9	Vilim Lendvaj	HRV	100	100	45	60	100	100	505
S	10	Jan Olkowski	POL	100	100	100	30	70	100	500
S	11	Stanisław Strzelecki	POL	100	100	25	60	100	100	485
S	12	Andrei-Costin Constantinescu	ROU	100	100	25	100	70	60	455
S		Paulína Smolárová	SVK	100	100	25	30	100	100	455
S	14	Josip Kelava	HRV	100	100	45	30	100	60	435
S	15	Vano Gamezardashvilli	GEO	100	100	45	30	50	100	425
S		Marian Dietz	DEU	100	100	65	30	70	60	425
S	17	Lukas Michel	DEU	100	100	25	30	100	60	415
B	18	Leonard Inkret	HRV	100	100		30	100	60	390
B	19	Martin Melicher	SVK		100	45	60	70	100	375
B	20	Máté Busa	HUN	60	100	45	30	70	60	365
B	21	Nir Shalmon	ISR	100	100	0	60	0	100	360
B		Attila Gáspár	HUN	100	100	25	30	70	35	360
B	23	Tobias Schindler	DEU	60	100	45	30	20	100	355
B	24	Gregor Kikelj	SVN		100	45	30	70	100	345
B	25	Yuval Salant	ISR	100	100	45	30		60	335
B	26	Ondrej Baranovič	SVK	30	100		30	70	100	330
B		Alan Marko	SVK	60	100		30	40	100	330
B	28	Orsolya Lili Janzer	HUN		100	25	30	70	100	325
B		Nicolas Camenisch	CHE		100	25	30	70	100	325
B		Matteo Zappia	ITA		100	25	30	70	100	325

(*nadaljevanje na naslednji strani*)

REZULTATI CEOI 2017 (*nadaljevanje*)

Medalja	Mesto	Ime	Država	Točke (po nalogah in skupaj)						
				Enosmerne ceste	Zanesljiva stava	Mišolovka	Gradnja mostov	Palindr. razdelitve	Lov	$\sum$
	31	Fabian Haller	CHE	0	100		30	70	100	300
	32	Matilde Padovano	ITA	60	100		30	70	35	295
	33	Áron Noszály	HUN	30	100		30	70	60	290
	34	Otto Winter	AUT	30	100	25	30	0	100	285
	35	Ziya Mukhtarov	AZE		100	20	30	70	60	280
	36	Teimuraz Toloraia	GEO	0	100	45	30	0	100	275
		Idan Izmirli	ISR	60	100	25	30	0	60	275
	38	Mirko Giacchini	ITA	60	100	0	30	20	60	270
		Daniele Venier	ITA	0	100	0	30	40	100	270
	40	Tim Poštuvan	SVN	0	100	0	30	70	60	260
	41	Abutalib Namazov	AZE	0	100	0	30		100	230
		Matej Marinko	SVN	0	100	0	30	0	100	230
	43	Žiga Željko	SVN	30	100	0	30		60	220
		Florian Jüngermann	DEU		20	0	30	70	100	220
	45	Pavel Hudec	CZE	0	100	25	30	0	60	215
		Alakbar Askarov	AZE	0	100	25	30	0	60	215
	47	Mitja Žalik	SVN	0	100	0	30	20	60	210
	48	Roee Sinai	ISR	60	100		30			190
		Elias Boschung	CHE	0	100		30	0	60	190
	50	Blaž Zupančič	SVN		100	45	30		0	175
	51	Thomas Kaar	AUT	0	60		30	20	60	170
		Bibin Muttappillil	CHE		80	0	30	0	60	170
	53	Martin Kurečka	CZE		60	25	30	0		115
	54	Danil Koževnikov	CZE		20		30	0	60	110
	55	Josef Minařík	CZE	30	20	25	30	0		105
	56	Rafail Saddatimov	AZE		0	0	30			30
	57	Bor Grošelj Simić	SVN		20	0	0		0	20
	58	Tadej Gašparovič	SVN		0			0	0	0

OFF-LINE NALOGA — NAJKRAJŠI SKUPNI NADNIZ

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 2016 skupaj z razpisom za tekmovanje v znanju; tekmovalci so imeli čas do 24. marca 2017 (dan pred tekmovanjem), da pošljejo svoje rešitve.

Opis naloge

Danih je več nizov, ki jih sestavljajo male črke angleške abecede. Naloga je poiskati kakšen čim krajši niz, v katerem se kot podnizi pojavljajo vsi dani vhodni nizi. Z drugimi besedami torej iščemo čim krajši skupni nadniz vseh vhodnih nizov. Pri tem so dovoljene tudi take pojavitve podnizov, v katerih se črke podniza ne pojavljajo strnjeno skupaj. (Na primer: niz `baa` se pojavlja kot podniz v nizu `banana`, in to celo na tri načine: `banana`, `banana` in `banana`.) Skupna dolžina vhodnih nizov je največ milijon znakov.

Primer: recimo, da imamo vhodne nize `miza`, `zima` in `mazivo`. Nekaj primernih nizov, ki so skupni nadnizi vseh treh vhodnih nizov, je na primer:

- `abcdefghijklmizazivonopqrstuvwxyzima` (36 znakov)
- `mizazimamazivo` (14 znakov)
- `mizimazivo` (10 znakov)
- `miazimavo` (9 znakov)
- `mazizmavo` (9 znakov)

Izkaže se, da je 9 znakov pri tem konkretnem primeru najboljša možna rešitev — noben skupni nadniz vseh treh vhodnih nizov ni krajši od 9 znakov.

Rezultati

Sistem točkovanja je bil tak kot pri off-line nalogah v prejšnjih letih. Pripravili smo 60 testnih primerov, torej naborov nizov, za katere iščemo skupni nadniz (za več o testnih primerih gl. str. 206). Pri vsakem testnem primeru smo razvrstili tekmovalce po dolžini njihovega nadniza, nato pa je prvi tekmovalec (tisti z najkrajšim nadnzom) 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 60 testnih primerih.

Letos je svoje rešitve pri off-line nalogi poslalo kar dvanajst tekmovalcev, od tega devet srednješolcev. Končna razvrstitev je naslednja:

Mesto	Ime	Letnik	Šola	Točke
1	Uroš Koritnik	4	ŠC Nova Gorica	486
2	Bor Grošelj Simić	2	Gim. Vič	444
3	Franci Obid	1	ŠC Nova Gorica	414
4	Tomaž Martinčič	1	FRI	383
5	Gregor Kikelj	3	ŠC NM, SEŠTG	333
6	Andrej Golčer			286
7	Blaž Zupančič	3	Šk. klas. gimn.	277
8	Martin Prelog	3	ŠC Kranj, STŠ	148
9	Amon Stopinšek	2	FRI	78
10	Nejc Šuklje	3	ŠC NM, SEŠTG	60
11	Matevž Rom	3	ŠC NM, SEŠTG	10
12	Žan Pust	1	ŠC NM, SEŠTG	1

Rešitev

Naj bo k število vhodnih nizov, posamezne vhodne nize pa označimo s $t^1, \ldots, t^k$. Množico vseh znakov, ki se pojavijo v vsaj enem vhodnem nizu, imenujmo *abeceda* in jo označimo s Σ, njeno velikost pa z $a = |\Sigma|$. Iščemo čim krajši skupni nadniz vseh vhodnih nizov, torej tak niz s, ki vsebuje vse nize $t^1, \ldots, t^k$ kot podnize.

Uvod. Problem iskanja najkrajšega skupnega nadniza je NP-težak [11, 16] — še več, tak ostane celo, če ga na razne načine omejimo, npr. če se omejimo na abecedo z dvema znakoma, na vhodne nize dolžine največ 3 ali pa če prepovemo, da bi se v vhodnih nizih kdaj pojavljali po dve zaporedni enaki črki [18, 13, 10]. Za iskanje optimalne rešitve (najkrajši skupni nadniz sploh) poznamo zato le algoritme z eksponentno časovno zahtevnostjo (v odvisnosti od števila vhodnih nizov), ki so primerni za manjše testne primere, za večje pa so prepočasni. Obstaja pa tudi več algoritmov, ki poskušajo najti čim krajši (vendar ne nujno najkrajši) skupni nadniz in ki so dovolj hitri, da so uporabni tudi na večjih testnih primerih. V splošnem je težko reči, kateri od njih je boljši (saj dajejo na nekaterih testnih primerih boljše rezultate eni algoritmi, na drugih drugi; poleg tega so nekateri od teh algoritmov za največje testne primere vendarle prepočasni), zato je v praksi najbolje, če jih preizkusimo več in pri vsakem testnem primeru obdržimo najboljšega izmed tako dobljenih nadnizov.

Preden nadaljujemo, še opomba glede notacije. Za poljuben niz u naj pomeni $|u|$ njegovo dolžino, u_i ali $u[i]$ njegov i-ti znak (za $i = 1, \ldots, |u|$), $u_{i..j}$ ali $u[i..j]$ pa podniz $u_i u_{i+1} \ldots u_{j-1} u_j$. Dolžine vhodnih nizov $t^1, \ldots, t^k$ bomo označili z $n_1, \ldots, n_k$.

Optimalna rešitev za dva niza. Recimo, da imamo samo dva vhodna niza, s in t. Označimo njuni dolžini z $n = |s|$ in $m = |t|$. Razmišljamo lahko takole: najkrajši skupni nadniz nizov s in t se mora končati bodisi na zadnji znak s-ja bodisi na zadnji znak t-ja. V prvem primeru imamo torej nadniz oblike us_n, pri čemer mora biti u najkrajši skupni nadniz nizov $s[1..n-1]$ in t, v drugem primeru pa imamo nadniz oblike ut_m, pri čemer mora biti u najkrajši skupni nadniz nizov s in $t[1..m-1]$. Če pa sta zadnja znaka obeh nizov enaka, $s_n = t_m$, je dovolj že, če za u vzamemo najkrajši skupni nadniz nizov $s[1..n-1]$ in $t[1..m-1]$. V vsakem primeru smo se torej znašli pred problemom enake oblike kot na začetku, le da namesto nizov s in

t zdaj gledamo neka prefiksa (začetka) teh nizov. S takšnim razmišljanjem lahko nadaljujemo proti vse krajšim nizom, dokler ne pridemo do trivialno preprostih problemov, ki jih znamo rešiti; na primer, če je eden od nizov prazen, je najkrajši skupni nadniz kar enak drugemu nizu.

Označimo s $f(i,j)$ dolžino najkrajšega skupnega nadniza nizov $s[1..i]$ in $t[1..j]$. Potem lahko razmislek iz prejšnjega odstavka povzamemo takole:

$$f(i,j) = 1 + \begin{cases} f(i-1,j-1), & \text{če } s_i = t_j \\ \min\{f(i-1,j), f(i,j+1)\} & \text{sicer.} \end{cases}$$

Robni primeri (ko je eden od nizov prazen) pa so $f(0,j) = j$ in $f(i,0) = i$. Vrednosti funkcije f bi lahko računali z rekurzivnim podprogramom (funkcijo); da ne bomo računali enih in istih vrednosti po večkrat, si vsako vrednost $f(i,j)$, ko jo prvič izračunamo, zapomnimo v neki tabeli, odkoder jo bomo lahko kasneje prebrali, kadarkoli jo bomo spet potrebovali. Še bolj elegantno kot z rekurzivnim podprogramom pa je, če računamo f sistematično po naraščajočih i in pri vsakem i po naraščajočih j. Tako imamo vedno, ko računamo funkcijo za nek (i,j), že izračunane rešitve za vse tiste podprobleme, ki jih pri njem potrebujemo. Prišli smo do naslednje rešitve z dinamičnim programiranjem:

algoritem O_2
```
1   for j := 0 to m do f[0,j] := j;
2   for i := 1 to n:
3     f[i,0] := i;
4     for j := 1 to m:
5       if s_i = t_j then q := f[i-1,j-1];
6       else q := min{f[i-1,j], f[i,j-1]};
7       f[i,j] := q + 1;
```

Za f torej potrebujemo tabelo $(n+1)\times(m+1)$ elementov. Na koncu tega postopka imamo v $f[n,m]$ dolžino najkrajšega skupnega nadniza nizov s in t. Naloga zahteva, da izpišemo primeren nadniz, ne le izračunamo njegovo dolžino. Pri sestavljanju takega nadniza lahko razmišljamo takole: recimo spet, da iščemo najkrajši skupni nadniz nizov $s[1..i]$ in $t[1..j]$. Če je $s_i = t_j$, je zadnji znak nadniza pač ta znak, pred tem pa imamo najkrajši skupni nadniz nizov $s[1..i-1]$ in $[1..j-1]$; torej zmanjšamo i in j za 1 in nadaljujemo s podobnim postopkom tudi tam. Če pa sta znaka s_i in t_j različna, moramo pogledati, katera od vrednosti $f[i-1,j]$ in $f[i,j-1]$ je manjša: če je $f[i-1,j] < f[i,j-1]$, moramo na zadnje mesto nadniza postaviti s_i (in pri nadaljevanju postopka zmanjšati i za 1), sicer pa t_j (in pri nadaljevanju postopka zmanjšati j za 1). (Če je $f[i-1,j] = f[i,j-1]$, je načeloma vseeno, ali za zadnji znak nadniza uporabimo s_i ali t_j; v obeh primerih bi nastal enako dolg nadniz, vendar pa sta tako dobljena nadniza različna.) Zapišimo s psevdokodo še ta del postopka:

algoritem O_2 (*nadaljevanje*)
```
8   i := n; j := m; alocirajmo izhodni niz u dolžine f[n,m];
9   while i > 0 or j > 0:
10    q := f[i,j];
11    if i = 0 then u_q := t_j; j := j - 1;
12    else if j = 0 then u_q := s_i; i := i - 1;
```

```
13      else if s_i = t_j then u_q := s_i; i := i − 1; j := j − 1;
14      else if f[i − 1, j] < f[i, j − 1] then u_q := s_i; i := i − 1;
15      else u_q := t_j; j := j − 1;
16  return u;
```

Mimogrede, najkrajši skupni nadniz dveh nadnizov seveda ni nujno enolično določen. (Na primer, iz *miza* in *zima* lahko dobimo *mizima* ali pa *zimiza*.) V našem gornjem postopku se to pokaže v vrsticah 14 in 15: to, ali za u_q vzamemo s_i ali t_j, je odvisno od tega, katera od vrednosti $f[i-1,j]$ in $f[i,j-1]$ je manjša. Če sta obe enaki, pa se lahko odločimo za eno ali za drugo možnost; tako lahko dobimo različne, vendar enako dolge nadnize. (V zgornji psevdokodi se v takih primerih vedno odločimo za t_j. Še ena možnost bi bila, da bi to odločitev prepustili generatorju naključnih števil.) To, katerega od več možnih enako dolgih najkrajših skupnih nadnizov dobimo, ni nujno čisto vseeno: nekateri od njih utegnejo dajati boljše, nekateri pa slabše rezultate v nadaljevanju postopka (npr. če bomo kasneje iskali skupni nadniz med dosedanjim nadnizom in kakšnim tretjim vhodnim nizom).

Doslej opisani postopek vrne najkrajši skupni podniz dveh vhodnih nizov in pri tem porabi $O(nm)$ pomnilnika (za tabelo f) in časa. Če sta vhodna niza zelo dolga, si toliko pomnilnika ne moremo privoščiti: na primer, pri nekaterih naših testnih primerih smo imeli dva niza dolžine približno 500 000 znakov; takrat bi imela tabela f približno 250 milijard elementov in če je vsak dolg 4 byte, je to skupaj približno 1 TB, toliko pomnilnika pa običajni računalniki dandanes nimajo.

Našo dvodimenzionalno tabelo f si lahko predstavljamo kot razdeljeno na vrstice in stolpce, tako da je indeks i številka vrstice, indeks j pa številka stolpca. Porabo pomnilnika lahko zmanjšamo z naslednjim opažanjem: ko naš gornji postopek računa vrednosti v i-ti vrstici tabele f, uporablja pri tem že izračunane vrednosti iz iste vrstice in iz prejšnje, torej $(i-1)$-ve vrstice; ne uporablja pa več vrednosti iz vrstic $i-2$, $i-3$ in tako naprej. Zato lahko vsebino teh vrstic sproti pozabljamo in tako sprostimo pomnilnik, ki bi ga sicer potrebovali za hranjenje teh vrstic. Za izračun funkcije f je dovolj že, če hranimo v pomnilniku le dve vrstici naenkrat (vrstico i, ki jo trenutno računamo, in poleg nje še prejšnjo vrstico $i-1$). Težava pa je, da se bomo morali na koncu sprehoditi nazaj po tabeli, da bomo sestavili primeren nadniz u. Takrat bomo morali torej vrstice, ki smo jih prej zavrgli in pozabili, izračunati ponovno.

Tabelo f razdelimo v mislih na *bloke* po b vrstic. Že izračunane vrstice tabele f bomo sproti pozabljali, le prvo vrstico vsakega bloka (torej $i = 0, b, 2b, 3b, \ldots$) si bomo zapomnili. V drugem delu postopka, ko sestavljamo nadniz u in se pri tem premikamo po tabeli navzgor (zmanjšujemo i), lahko vsakič, ko pri premiku navzgor dosežemo nov blok, ta blok v celoti izračunamo iz njegove prve vrstice (ki jo imamo še shranjeno v pomnilniku); spodnji blok, iz katerega smo ravnokar prišli, pa lahko zdaj pozabimo, saj ga ne bomo več potrebovali. Poraba pomnilnika je torej zdaj takšna: imamo približno n/b blokov in od vsakega si moramo zapomniti po eno vrstico, poleg tega pa še celoten trenutni blok s približno b vrsticami. Tako torej porabimo $O((b+n/b)m)$ pomnilnika. Vidimo lahko, da bo ta poraba najnižja pri $b \approx \sqrt{n}$; takrat dobimo $O(m\sqrt{n})$. Če sta vhodna niza s in t različno dolga, je koristno za s vzeti daljšega od njiju. Časovna zahtevnost tega postopka je še vedno le $O(nm)$, vendar moramo zdaj vsak blok izračunati dvakrat (najprej pri premikanju dol po

tabeli, ko prvič računamo tabelo f, nato pa še enkrat pri premikanju gor po tabeli, ko sestavljamo nadniz u), zato si lahko predstavljamo, da se bo izvajal približno dvakrat dlje kot prvotni postopek. Za naše testne primere s po dvema nizoma dolžine 500 000 je to čisto primerna rešitev (porabi približno 2,6 GB pomnilnika). Zapišimo ta postopek še s psevdokodo:

algoritem O_2-S
  for $i := 0$ **to** n:
    izračunaj vrstico $f[i]$ s pomočjo vrednosti iz vrstice $f[i-1]$;
    pozabi vrstico $f[i-1]$, razen če je $(i-1)$ večkratnik b;
  $i := n$; $j := m$; alocirajmo izhodni niz u dolžine $f[n, m]$;
  while $i > 0$ **or** $j > 0$:
    if $i > 0$ in je i večkratnik b in vrstice $i-1$ še nimamo v pomnilniku:
      (* *Izračunajmo ponovno zgornji blok.* *)
      for $i' := i - b + 1$ **to** $i - 1$:
        izračunaj vrstico $f[i']$ s pomočjo vrednosti iz vrstice $f[i'-1]$;
      (* *Pozabimo spodnji blok, ki ga ne bomo več potrebovali.* *)
      for $i' := i + 1$ **to** $\min\{i+b, n\}$:
        pozabi vrstico $f[i']$;
    določi trenutni znak niza u enako kot v vrsticah 10–15 prejšnjega algoritma;
  return u;

Za naše namene je ta rešitev čisto dovolj dobra, kot zanimivost pa omenimo, da obstajajo še varčnejši algoritmi, na primer Hirschbergov, ki porabi $O(nm)$ časa (enako kot naš gornji algoritem), vendar le $O(m+n)$ prostora [6] (srečali smo ga tudi že na str. 162 v tem biltenu). Težje pa je zmanjšati porabo časa; pokazati je mogoče, da v najslabšem primeru za iskanje najkrajšega skupnega nadniza dveh nizov nujno potrebujemo $O(nm)$ časa [1], obstaja pa precej algoritmov, ki skušajo zmanjšati porabo časa pod to mejo vsaj za nekatere pare nizov (glej npr. pregled v [18] in tam navedeno literaturo), na primer take, kjer je pogoj $s_i = t_j$ izpolnjen le pri majhnem deležu parov (i, j).

Optimalna rešitev za več nizov. Dosedanje rešitve za dva niza ni težko posplošiti na več nizov. Recimo, da imamo k vhodnih nizov $t^1, \ldots, t^k$ z dolžinami $n_1, \ldots, n_k$. Podobno kot prej si zastavimo podproblem: naj bo $f(i_1, \ldots, i_k)$ dolžina najkrajšega skupnega nadniza nizov $t^1[1..i_1], \ldots, t^k[1..i_k]$. Tak nadniz se mora končati na enega od znakov $t^r[i_r]$ (za $r = 1, \ldots, k$); če se konča recimo na znak c, smo s tem pokrili $t^r[i_r]$ pri tistih r, kjer je $t^r[i_r] = c$, zato se moramo pri tistih nizih premakniti za eno mesto nazaj. Lažje kot s formulo je to opisati s postopkom:

funkcija $f(i_1, \ldots, i_k)$:
  $C := \{\}$; **for** $r := 1$ **to** k **do if** $i_r > 0$ **then** dodaj $t^r[i_r]$ v C;
  if C je prazna **then return** 0;
  $g := \infty$;
  za vsak $c \in C$:
    for $r := 1$ **to** k:
      if $i_r > 0$ **and** $t^r[i_r] = c$
      then $j_r := i_r - 1$ **else** $j_r := i_r$;

$g := \min\{g, f(j_1, \ldots, j_k)\}$;
return $g + 1$;

Vidimo lahko, da ko rešujemo podproblem $(i_1, \ldots, i_k)$, si pri tem pomagamo z rešitvami takšnih podproblemov $(j_1, \ldots, j_k)$, pri katerih je vsak j_r enak bodisi i_r bodisi $i_r - 1$. Podobno kot pri rešitvi za dva niza je torej koristno reševati te podprobleme po naraščajočih indeksih in si rezultate shranjevati v tabelo; ta bo zdaj k-dimenzionalna in velika $(n_1 + 1) \times (n_2 + 1) \times \ldots \times (n_k + 1)$ elementov.

algoritem O_k
alociraj tabelo f;
for $r := 1$ **to** k **do** $i_r := 0$;
while true:
  izračunaj $f(i_1, \ldots, i_k)$ in ga shrani v ustrezni element tabele;
  $r := 1$;
  while $r \leq k$:
    if $i_r < n_r$ **then** $i_r := i_r + 1$; **break**
    else $i_r := 0$; $r := r + 1$;
  if $r > k$ **then break**;

V vsaki iteraciji glavne zanke torej izračunamo vrednost f za trenutni nabor indeksov (pri tem bodo prišle prav že shranjene rešitve prejšnjih podproblemov, ki jih imamo v tabeli), nato pa se premaknemo na naslednji nabor. Ta premik je zelo podoben kot pri povečevanju števila za 1: najprej povečujemo i_1, ko pa ta doseže n_1, ga postavimo nazaj na 0 in povečamo i_2; ko sčasoma i_2 doseže n_2, postavimo tudi njega na 0 in povečamo i_3 in tako naprej.

Ko imamo enkrat potabelirane vse vrednosti funkcije f, lahko najkrajši podniz sestavimo podobno kot pri rešitvi za dva niza:

algoritem O_k (*nadaljevanje*)
for $r := 1$ **to** k **do** $i_r := n_r$;
$q := f[i_1, \ldots, i_k]$; alocirajmo izhodni niz u dolžine q;
while $q > 0$:
  ponovi izračun $f(i_1, \ldots, i_k)$, vendar si tudi zapomni,
    pri katerem c je vrednost g dosegla svoj minimum;
  $u_q := c$; $q := q - 1$;
  for $r := 1$ **to** k **do if** $i_r > 0$ **and** $t^r[i_r] = c$ **then** $i_r := i_r - 1$;
return u;

Ta postopek torej porabi $O(\prod_{r=1}^{k}(n_r + 1))$ prostora; ta produkt nastopa tudi v porabi časa, kjer pa mu moramo dodati še vsaj en faktor $O(k)$, ker se v vsakem izračunu funkcije f skrivajo zanke po vseh vhodnih nizih.

Doslej opisani postopek je primeren za nekatere naše testne primere z majhnim številom kratkih nizov; trije testni primeri pa so zanj že neugodno veliki. Pri enem imamo na primer šest nizov dolžine 17 in tri nize dolžine 18, torej potrebujemo tabelo z $18^6 \cdot 19^3$ elementi, kar je približno 233 milijard. Ker so nizi tako kratki, je tudi njihov nadniz kratek in za posamezni element tabele zadošča že 1 byte; vseeno pa je dobrih 217 GB velika tabela za glavni pomnilnik običajnega dandanašnjega

osebnega računalnika že prevelika. Razmislimo torej o tem, kako porabo pomnilnika še zmanjšati.

Podobno kot smo si pri rešitvi za $k = 2$ predstavljali dvodimenzionalno tabelo f kot sestavljeno iz $n + 1$ vrstic (= enodimenzionalnih tabel), si lahko zdaj našo k-dimenzionalno tabelo f predstavljamo kot sestavljeno iz $n_1 + 1$ manjših, $(k-1)$-dimenzionalnih podtabel: $f[0], f[1], \ldots, f[n_1]$. Podobno kot prej tudi zdaj vidimo, da ko računamo vrednosti v $f[i_1]$, potrebujemo pri tem nekaj že prej izračunanih vrednosti iz $f[i_1]$ in pa vrednosti iz $f[i_1 - 1]$, ne potrebujemo pa več vrednosti iz $f[i_1 - 2]$, $f[i_2 - 3]$ in tako naprej.

Na primer, če pri testnem primeru, o katerem smo govorili malo prej (šest nizov dolžine 17 in trije nizi dolžine 18) za prvi niz vzamemo enega od daljših ($n_1 = 18$), nam tabela f (dolga približno 217 GB) razpade na 19 podtabel, od katerih je vsaka dolga dobrih 11 GB. Recimo, da si lahko v pomnilniku privoščimo imeti dve taki podtabeli naenkrat, več kot toliko pa že težko.

Zdaj bi lahko te podtabele v mislih združevali v bloke, podobno kot smo naredili pri algoritmu O_2-S, vendar bi se bilo pri tem težko izogniti potrebi po tem, da hranimo v pomnilniku več kot dve podtabeli hkrati. Raje si pomagajmo z naslednjo, še preprostejšo rešitvijo: ko podtabele ne potrebujemo več, jo preprosto odložimo na disk.

algoritem O_k-S
  for $i_1 := 0$ **to** n_1:
    if $i_1 \geq 2$:
      shrani podtabelo $f[i_1 - 2]$ na disk in jo zavrzi iz glavnega pomnilnika;
    izračunaj vse vrednosti f v podtabeli $f[i_1]$;
  nadaljuj enako kot v vrsticah 10–16 algoritma O_k
    z naslednjo razliko: ko se i_1 zmanjša z x na $x - 1$,
    zavrzi podtabelo $f[x]$ iz glavnega pomnilnika
    in (če je $x \geq 2$) naloži $f[x - 2]$ z diska v glavni pomnilnik;

Časovna zahtevnost tega algoritma je še vedno taka kot prej, le za nek konstantni faktor večja (ker moramo večino tabele f enkrat zapisati na disk in enkrat prebrati z njega). Tudi prostorska zahtevnost je enaka kot prej, vendar se zdaj nanaša na porabo prostora na disku; v glavnem pomnilniku pa hranimo le $2\prod_{r=2}^{k}(n_r + 1)$ elementov tabele f naenkrat. Pri največjem izmed naših majhnih testnih primerov pri letošnji off-line nalogi bomo tako porabili približno 22 GB glavnega pomnilnika in 217 GB prostora na disku, kar je za današnje računalnike še obvladljivo.

Požrešni algoritem po znakih. Če je vhodnih nizov malo več in so malo daljši (recimo vsaj nekaj deset nizov, dolgih po vsaj nekaj deset ali sto znakov), si doslej omenjenih optimalnih rešitev ne moremo privoščiti, saj nimamo niti dovolj časa niti dovolj pomnilnika, zato moramo poseči po raznih hevristikah, s katerimi lahko poskusimo najti čim boljšo rešitev, nimamo pa zagotovil o tem, kako blizu najboljše možne bomo na ta način prišli.

Oglejmo si algoritem, ki se je pri naših poskusih dobro obnesel pri srednje velikih testnih primerih — takšnih, kjer obstaja nek skupni nadniz, dolg največ kakšnih 1000 znakov. Nadniz bomo gradili postopoma, znak za znakom; začeli bomo s praznim nizom in vanj v vsakem koraku vrinili po eno črko. Vprašanje je, katero črko vriniti

in kam. Kako obetaven je niz v, ki ga dobimo po nekem takem vrivanju? Ker smo začeli s praznim nizom in vanj postopoma dodajamo črke, niz v v resnici najbrž sploh še ni nadniz vseh vhodnih nizov. Toda kako blizu je temu, da bi vendarle bil nadniz vseh vhodnih nizov? To lahko ocenimo takole: predstavljajmo si najdaljši skupni podniz nizov t^r in v; označimo ga z $\textsc{nsp}(t^r, v)$. Razlika $|t^r| - |\textsc{nsp}(t^r, v)|$ nam pove, kolikšno je najmanjše število dodatnih znakov, ki bi jih bilo treba vriniti v v, da bi postal nadniz niza t^r; če je razlika 0, to pomeni, da je v že zdaj nadniz niza t^r. Torej je v tem bolj obetaven, čim daljši je $\textsc{nsp}(t^r, v)$. Dolžino slednjega seštejmo po vseh $r = 1, \ldots, k$, pa dobimo oceno obetavnosti v-ja:

$$J(v) := \sum_{r=1}^{k} |\textsc{nsp}(t^r, v)|.$$

Največja možna vrednost te ocene nastopi takrat, ko je v že nadniz vseh $t^1, \ldots, t^k$; takrat imamo oceno $J^\star = \sum_{r=1}^{k} |t^r|$. Osnovna ideja našega algoritma je torej takšna:

algoritem G:
1 $u :=$ prazen niz; $q := 0$;
2 **while** $J(u) < J^\star$:
3 **for** $i := 1$ **to** $|u| + 1$:
4 **za vsako** črko c, ki se pojavlja v kakšnem vhodnem nizu:
5 $v :=$ niz, ki ga dobimo, če v u vrinemo c na indeks i;
6 izračunaj oceno $J(v)$;
7 $u :=$ tisti med vsemi tako pregledanimi v, ki je imel največjo $J(v)$;

Postopek je torej požrešen v tem smislu, da vsakič vrine tak znak, ki najbolj poveča oceno $J(u)$. (Ni pa na primer toliko požrešen, da bi poskušal graditi niz u od leve proti desni in torej vedno dodajati črke le na konec u-ja. Pri naših poskusih je tak algoritem dajal precej slabše rezultate, je pa res, da deluje veliko hitreje.)

V vsaki iteraciji zunanje zanke tega postopka moramo oceniti kar precej nizov v; razmislimo o tem, kako lahko to naredimo dovolj učinkovito, da bo ta postopek uporaben tudi v praksi. Ko ocenjujemo nek v (ki smo ga dobili tako, da smo v u vrinili c na indeksu i), nas zanimajo najdaljši skupni podnizi med njim in vsemi t^r. Tak podniz lahko v nizu v zajame tudi pravkar vrinjeni c (ki je v nizu v na indeksu i) ali pa ga ne. Če ga ne, je ta podniz hkrati tudi podniz u-ja in je neodvisen od c in i, zato tega ne bo treba računati pri vsakem v posebej. Druga možnost pa je, da najdaljši skupni podniz nizov v in t^r zajame tudi tudi c na mestu $v[i]$; tedaj se mora ta c pojaviti tudi v pojavitvi tega podniza v t^r, recimo na nekem indeksu j (torej imamo $t^r[j] = c$). Vse tri nize — v, t^r in njun najdaljši skupni podniz — lahko zdaj razdelimo vsakega na tri dele: levi del (vse pred c-jem), c sam in desni del (vse za c-jem). Tako imamo:

$$\begin{aligned} v &= v[1..i-1] \quad c \quad v[i+1..q+1] \\ &= u[1..i-1] \quad c \quad u[i..q]\,, \\ t^r &= t^r[1..j-1] \quad c \quad t^r[j+1..n_r] \text{ in} \\ \textsc{nsp}(v, t^r) &= x \quad c \quad y \end{aligned}$$

za neka niza x in y. Naš $\textsc{nsp}(v, t^r)$ ne bi mogel biti res najdaljši skupni podniz nizov v in t^r, če ne bi bil hkrati tudi njegov levi (oz. desni) del najdaljši skupni podniz

levih (oz. desnih) delov nizov v in t^r. Torej mora biti $x = \text{NSP}(u[1..i-1], t^r[1..j-1])$ in $y = \text{NSP}(u[i..q], t^r[j+1..n_r])$. Ker se v splošnem lahko c pojavi na več mestih j v nizu t^r, moramo preizkusiti vse te j in med njimi uporabiti tistega, ki pripelje do najdaljšega podniza. Tako smo dobili:

$$|\text{NSP}(v, t^r)| = \max\{\, |\text{NSP}(u, t^r)|, \max\{\, |\text{NSP}(u[1..i-1], t^r[1..j-1])| + 1 + |\text{NSP}(u[i..q], t^r[j+1..n_r])| : 1 \le j \le n_r,\ t^r[j] = c \,\}\,\}.$$

Če torej izračunamo dolžino najdaljših skupnih podnizov med vsemi začetki (prefiksi) nizov u in t^r in podobno še med vsemi končnicami (sufiksi) teh nizov, bomo lahko s pomočjo teh dolžin precej hitro izračunali $|\text{NSP}(v, t^r)|$ za poljuben v (torej za poljubna i in c). Tega pa ni težko računati z dinamičnim programiranjem, podobno kot smo že prej videli za najkrajši skupni nadniz. Pišimo $f(i,j) = |\text{NSP}(u[1..i], t^r[1..j])|$ in $g(i,j) = |\text{NSP}(u[i..q], t^r[j..n_r])|$. Funkcijo f lahko računamo takole:

```
for j := 0 to n_r do f[0, j] := 0;
for i := 1 to q:
    f[i, 0] := 0;
    for j := 1 to n_r:
        if u_i = t^r[j] then c := f[i − 1, j − 1]
        else c := min{f[i − 1, j], f[i, j − 1]};
        f[i, j] := 1 + c;
```

Zelo podobno pa tudi g:

```
for j := n_r + 1 downto 1 do g[q + 1, j] := 0;
for i := q to 1:
    g[i, n_r + 1] := 0;
    for j := n_r downto 1:
        if u_i = t^r[j] then c := g[i − 1, j + 1]
        else c := min{g[i − 1, j], f[i, j + 1]};
        g[i, j] := 1 + c;
```

Med drugim se v teh dveh tabelah skriva tudi dolžina $|\text{NSP}(u, t^r)|$, in sicer v $f[q, n_r]$ in v $g[1, 1]$.

S pomočjo tabel f in g lahko zdaj dovolj poceni izračunamo $|\text{NSP}(v, t^r)|$ za poljuben v. Zdaj imamo vse, kar potrebujemo, da lahko učinkovito ocenimo vse možne v. Njihove ocene bomo počasi računali v tabeli, v kateri $J[i, c]$ predstavlja oceno $J(v)$ tistega niza v, ki ga dobimo z vrivanjem znaka c na indeks i v nizu u.

```
    (* Ta postopek naredi to, kar je bilo nakazano v vrsticah 3–6 algoritma G. *)
1   for i := 1 to q:
2       za vsak znak abecede c: J[i, c] := 0;
3   for r := 1 to k:
4       pripravi tabeli f in g po zgoraj opisanem postopku;
5       for i := 1 to q:
6           za vsak znak abecede c ∈ Σ:
                (* V d izračunajmo |NSP(v, t^r)|. Ena možnost za to je
                   |NSP(u, t^r)|, ki jo imamo na primer v g[1, 1]. *)
```

```
 7        d := g[1, 1];
          (* Druga možnost je, da podniz pokrije c na mestu v[i],
             torej mora biti c prisoten tudi v t^r na nekem indeksu j. *)
 8        for j := 1 to n_r do if t^r[j] = c then
 9          d := max{d, f[i − 1, j − 1] + 1 + g[i, j + 1] };
10      J[i, c] := J[i, c] + d;
```

Notranjo zanko po j lahko še malo izboljšamo, če si vnaprej pripravimo za vsako možno črko c in vsak vhodni niz t^r seznam indeksov, kjer se ta črka pojavlja v t^r. Tako bo morala iti naša notranja zanka le po tistih j, ki nas zanimajo. Pri vsakem konkretnem i se tako, pri vseh c-jih skupaj, notranja zanka po j izvede največ n_r-krat, ker se z vsakim j srečamo pri največ enem c (namreč pri $c = t^r[j]$).

Kakšna je časovna zahtevnost tega postopka? Pri vsakem r (v zanki v vrsticah 3–10) porabimo najprej $O(q \cdot n_r)$ časa za izračun tabel f in r (vrstica 4), nato pa imamo še q iteracij zanke po i (vrstice 5–10) in v vsaki od njih a iteracij zanke po c (npr. tolikokrat se izvede vrstica 7; pri tem je $a = |\Sigma|$ velikost naše abecede) ter vsega skupaj (po vseh c pri tem i) največ n_r iteracij zanke po j (vrstica 9); to je za celotno zanko po i skupaj $O(q(a + n_r))$ časa. Za vrstice 4–10 imamo tako skupaj $O(q(a + n_r))$ časa in za celoten postopek računanja vseh ocen $J(v)$ dobimo skupaj $O(q(ka + \sum_{r=1}^{k} n_r))$. Če označimo povprečno dolžino vhodnih nizov z $\bar{n}$, lahko ta izraz poenostavimo v $O(qk(a + \bar{n}))$.

Zdaj lahko razmislimo še o časovni zahtevnosti algoritma G kot celote. Pravkar opisani postopek pokrije vrstice 3–6 tistega algoritma; nato pa porabimo (v vrstici 7) še $O(qa)$ časa za izbor najbolj obetavnega q in $O(q)$ časa, da se premaknemo vanj. Če je nadniz, ki ga na koncu dobimo, dolg d znakov, to pomeni, da glavna zanka algoritma (vrstica 2) naredi d iteracij, pri katerih se q počasi povečuje od 0 do d, torej je časovna zahtevnost celotnega algoritma G reda $O(d^2k(a + \bar{n}))$. Zaradi te kvadratne zahtevnosti v odvisnosti od d je ta algoritem primeren le za tiste testne primere, pri katerih obstaja nek dovolj kratek skupni nadniz vseh vhodnih nizov (recimo do $d \approx 1000$).

Omenimo še naslednje: v vrstici 7 algoritma G se pogosto zgodi, da niz v z najvišjo $J(v)$ ni en sam, ampak je takih nizov več (vsi pa imajo enako oceno) in se bomo morali naključno odločiti za enega od njih. Ker pa niso nujno vsi ti nizi enako primerni za nadaljevanje postopka, se zato lahko zgodi, da bo na koncu dobljeni nadniz malo krajši ali pa malo daljši, odvisno od naših naključnih odločitev. Zato je v takem primeru koristno naš algoritem pognati po večkrat in si zapomniti najkrajšega od tako dobljenih nadnizov.

Sestavljanje nadniza od leve proti desni. Naš pravkar opisani požrešni algoritem je dodajal znake v nadniz enega po enega, vendar jih je bil pripravljen vrivati kjerkoli v nadnizu. Videli smo, da nas je to pripeljalo do neugodno velike časovne zahtevnosti, ker smo morali vsakič oceniti vse možne položaje, kamor bi se dalo vriniti naslednji znak v naš nastajajoči nadniz. Hitrejši (in preprostejši), vendar malo slabši algoritem dobimo, če se odločimo, da bomo znake vedno dodajali na konec nadniza. Tako nam vprašanje o tem, *kam* vriniti naslednji znak, odpade in razmišljati moramo le še o tem, *kateri* znak bi dodali.

algoritem W:
1 $u :=$ prazen niz; **for** $r := 1$ **to** k **do** $i_r := 1$;
2 ponavljaj, dokler pri kakšnem r še velja $i_r \leq n_r$:
3 izberi naslednjo črko $c \in \Sigma$ in jo dodaj na konec niza u;
4 **for** $r := 1$ **to** k **do if** $i_r \leq n_r$ **and** $t^r[i_r] = c$
then $i_r := i_r + 1$;

Postopek se torej z indeksi $i_1, \ldots, i_r$ premika naprej po vhodnih nizih; v vsaki iteraciji glavne zanke si izbere naslednji znak c in se premakne za eno mesto naprej po tistih vhodnih nizih, ki so imeli na trenutnem mestu c (torej $t^r[i_r] = c$). Ko pridemo do konca vseh vhodnih nizov ($i_r > n_r$ pri vseh r), se lahko ustavimo; takrat je u nek skupni nadniz vseh vhodnih nizov.[15]

Obstaja več različic tega postopka, ki se razlikujejo po tem, kako si v vrstici 3 izberejo naslednjo črko:

- Ena možnost je, da si izberemo naslednjo črko tako, da se bomo premaknili naprej po tistem vhodnem nizu, iz katerega smo doslej pobrali najmanj črk. Med vsemi r, ki imajo $i_r \leq n_r$, torej vzamemo tistega z najmanjšim i_r in za ta r potem vzamemo $c := t^r[i_r]$. Če obstaja več enako dobrih r-jev (z enakim i_r), si enega od njih izberemo naključno. Tej hevristiki v literaturi ponavadi pravijo *min-height* [9], vendar se pri naših poskusih na naših testnih primerih ni dobro obnesla.

- Naslednjo črko si lahko izberemo tako, da se bomo premaknili naprej po čim več vhodnih nizih. Vzamemo torej tisto c, za katero velja pogoj $t^r[i_r] = c$ pri največ r-jih (izmed tistih $r \in \{1, \ldots, k\}$, pri katerih je $i_r \leq n_r$, torej da še nismo na koncu tistega vhodnega niza). Če je po tem kriteriju več črk c enako dobrih, si eno od njih izberemo naključno. Tej hevristiki ponavadi pravijo *sum-height* ali *majority merge* (MM) [7].[16]

- Prejšnjo hevristiko (MM) lahko posplošimo tako, da vsakemu znaku vsakega vhodnega niza pripišemo neko utež: naj bo $w(r, i)$ utež i-tega znaka niza t^r. Ko se potem odločamo, kateri c bi dodali v naš izhodni niz, izračunajmo $\omega(c) := \sum_r w(r, i_r)$, pri čemer gre vsota po tistih r, pri katerih je $i_r \leq n_r$ in $t^r[i_r] = c$. V izhodni niz dodamo tisti c, ki ima največjo $\omega(c)$. Če postavimo vse $w(r, i)$ na 1, smo na istem kot pri MM. Vprašanje je seveda, kako si izbrati te uteži; nekateri avtorji so to počeli z znanimi (in pogosto precej zamudnimi) postopki za naključno preiskovanje prostora, na primer genetskimi algoritmi [4] ali kolonijami mravelj [15].

- Eleganten in poceni razmislek, s katerim lahko pridemo do dobrega nabora uteži, pa je naslednji [5]: lahko se zgodi, da nam je od nekaterih vhodnih nizov ostalo še veliko znakov, od drugih pa malo (ali pa smo celo prišli pri njih že do konca niza); in tedaj je koristno, če se osredotočamo predvsem na črke

[15] O pravilnosti tega postopka se lahko prepričamo s pomočjo naslednje invariante: na začetku vsake iteracije glavne zanke velja, je u nek skupni nadniz nizov $t^r[1..i_r - 1]$ za vse $r = 1, \ldots, k$.

[16] Ime *majority merge* (večinsko zlivanje) je sicer rahlo zavajajoče, saj ni nujno, da se c pojavlja na trenutnem položaju pri *večini* vhodnih nizov, pač pa le, da se nobena druga črka ne pojavlja na trenutnem položaju pri več nizih kot c.

iz tistih nizov, pri katerih nam je ostalo še veliko znakov (kajti če je od nekega niza ostalo le še malo znakov, si lahko mislimo, da jih bomo najbrž brez težav prej ali slej pobrali spotoma, ne da bi se posebej osredotočali nanje). Tako lahko za $w(r,i)$ vzamemo kar dolžino tistega dela niza t^r, ki ga še nismo pokrili z doslej sestavljenim delom nadniza: torej $w(r,i) := n_r - i + 1$. Tej hevristiki pogosto pravijo *weighted majority merge* (WMM).

- Pri naših poskusih so je pogosto še bolje obnesle uteži oblike $w(r,i) := (n_r - i + 1)^\alpha$ za neko konstanto $\alpha > 1$. To, katera α je dajala najboljše rezultate, je pri različnih testnih primerih različno. Učinek takšnega potenciranja je, da algoritem še bolj prisilimo k osredotočanju na tiste nize, od katerih nam je ostalo še veliko znakov. To različico bomo v nadaljevanju označevali s PMM(α).[17]

- Če se v PMM splača vzeti $\alpha > 1$ in s tem v vsoti $\sum_r (n_r - i + 1)^\alpha$ še bolj poudariti tiste r, pri katerih je dolžina ostanka $n_r - i + 1$ velika, nam lahko pride na misel, da bi šli s tem razmislekom do konca in gledali le tisti r, pri katerem je razlika $n_r - i_r$ največja; za tisti r bi potem vzeli $c = t^r[i_r]$. Če je takih r-jev več (enako dobrih), si izberimo tisti c, ki se pojavlja pri največ teh r-jih. Vendar pa se ta hevristika pri naših poskusih ni dobro obnesla (pa tudi pri poskusih s PMM se je pokazalo, da prevelika vrednost α začne škodovati).

Ko enkrat poznamo uteži $w(r,i)$ (oz. znamo poljubno utež izračunati v $O(1)$ časa), opisanega požrešnega algoritma ni težko implementirati tako, da ima časovno zahtevnost $O(d \cdot a + \sum_r n_r)$, če je d dolžina nadniza, ki ga vrne na koncu.

Videli smo, da se včasih lahko zgodi, da se mora algoritem odločiti med več možnimi c, ki so z vidika njegove hevristike videti enako dobri, zato si enega od njih izbere naključno. To pomeni, da če algoritem poženemo po večkrat, lahko nastanejo različni (in tudi različno dolgi) nadnizi, zato ga je koristno (če imamo čas) pognati večkrat in si zapomniti najkrajši tako dobljeni nadniz.[18]

Doslej opisani postopek je nekoliko kratkoviden: ko razmišlja o tem, kateri znak $c \in \Sigma$ bi zdaj dodal na konec izhodnega niza, gleda le na to, katere vhodne znake bi z njim pokrili (torej sešteje uteži $w(r,i_r)$ za trenutne indekse i_r pri tistih vhodnih nizih, ki imajo $t^r[i_r] = c$). Nič pa ne razmišlja o tem, kaj se bo zgodilo v nadaljevanju postopka; mogoče bi bilo bolje zdajle vzeti kakšen drug znak, ki je sicer na prvi pogled manj obetaven, vendar nam bo kasneje omogočil priti do boljše rešitve. Postopek lahko zato izboljšamo tako, da naj gleda več znakov naprej (*lookahead*), recimo ℓ znakov. Namesto vsakega znaka c moramo zdaj pregledati vse možne „podaljške“ — vse nize ℓ znakov iz abecede Σ, s katerimi bi se dalo zdaj podaljšati naš dosedanji izhodni niz u. Enako kot pri prvotnem postopku tudi tu vsak tak podaljšek ocenimo z vsoto uteži tistih znakov, ki jih ta podaljšek pokrije v vhodnih

[17] Ideja potenciranja uteži se pojavi v [15], kjer so uporabili $\alpha = 9$. Kasneje še mnogi avtorji omenjajo WMM, potenciranja pa ne, kar je škoda, ker je videti, da se res dobro obnese.

[18] Lahko pa gremo z naključnostjo še malo dlje: v vsaki iteraciji se najprej z verjetnostjo q odločimo, ali bi uporabili tisti c, ki maksimizira $\omega(c)$ (tako, kot smo to počeli doslej), ali pa bi (z verjetnostjo $1-q$) si c izbrali naključno (z verjetnostjo, ki je sorazmerna $\omega(c)$). Avtorja, ki opisujeta ta pristop [15], sta sicer uporabila $q = 0{,}9$, torej večinoma vendarle uporabita tisti c, ki maksimizira $\omega(c)$.

nizih. Na koncu potem podaljšamo izhodni niz s prvim znakom tistega podaljška, ki je imel najvišjo oceno:

algoritem W-L(ℓ)
1 $u :=$ prazen niz; **for** $r := 1$ **to** k **do** $i_r := 1$;
2 ponavljaj, dokler pri kakšnem r še velja $i_r \le n_r$:
3 $\omega^\star := 0$; (* *ocena najboljšega podaljška doslej* *)
4 za vsak niz v dolžine ℓ (nad abecedo Σ):
5 **for** $r := 1$ **to** k **do** $j_r := i_r$;
6 $\omega := 0$; (* ω *bo ocena trenutnega podaljška,* v *) *)
7 **for** $\lambda := 1$ **to** ℓ:
8 **for** $r := 1$ **to** k **do if** $j_r \le n_r$ **and** $t^r[j_r] = v[r]$
then $\omega := \omega + w(r, j_r)$; $j_r := j_r + 1$;
9 **if** $\omega \ge \omega^\star$ **then** $\omega^\star := \omega$; $c := v[1]$;
10 **for** $r := 1$ **to** k **do if** $i_r \le n_r$ **and** $t^r[i_r] = c$
then $i_r := i_r + 1$;
11 dodaj c na konec niza u;

Ta postopek je seveda precej počasnejši od prvotnega, saj mora v vsaki iteraciji glavne zanke pregledati kar a^ℓ različnih podaljškov. Večji ko je a (velikost abecede Σ), hitreje narašča ta časovna zahtevnost v odvisnosti od ℓ in krajše podaljške si bomo lahko privoščili. Pri naših testnih primerih smo imeli ponekod $a = 5$ (tu gremo lahko brez težav do $\ell = 7$ ali 8), ponekod pa $a = 22$ in 26 (tu gremo lahko vsaj do $\ell = 2$, z nekaj potrpežljivosti tudi do $\ell = 3$). Že $\ell = 2$ je pri naših poskusih dajal precej boljše rezultate kot $\ell = 1$ (torej prvotni postopek brez gledanja naprej), pri večjih ℓ pa so se sicer rezultati še izboljševali, vendar vse manj.

Tudi pri tem postopku se lahko zgodi, da ima več možnih nadaljevanj enako oceno in se moramo naključno odločiti za enega od njih; zato lahko tudi ta postopek poženemo po večkrat, dobimo različno dolge nadnize in na koncu obdržimo najkrajšega od njih. Vendar pa se je pri naših poskusih izkazalo, da je, če imamo dovolj časa, bolje le-tega porabiti tako, da postopek poženemo enkrat z malo večjim ℓ kot pa večkrat z manjšim ℓ.

Najkrajši skupni nadniz, ki se čim bolj prekriva z ostalimi vhodnimi nizi. Pri postopku O_2 smo že videli, da v splošnem lahko obstaja več različnih, vendar enako dolgih najkrajših skupnih nadnizov za dana dva niza s in t. V nadaljevanju bomo O_2 uporabljali kot osnovni gradnik postopkov za iskanje najkrajšega skupnega podniza več kot dveh nizov. Ko se mora O_2 odločiti, katerega od več enako dolgih najkrajših skupnih podnizov s in t naj vrne, se je dobro zavedati naslednjega: čeprav so vsi ti nadnizi enako dolgi, pa niso nujno vsi enako primerni za nadaljevanje postopka, npr. ko bomo poskušali poiskati nadniz tega nadniza in kakšnega tretjega vhodnega niza.

Zato nam lahko pride na misel, da bi pri sestavljanju najkrajšega skupnega nadniza dveh nizov s in t uporabili še nek dodaten kriterij, ki bi nam pomagal izbrati med najkrajšimi nadnizi takega, ki bo imel v nekem smislu čim več skupnega z ostalimi vhodnimi nizi, tako da se bo kasneje čim manj podaljšal, ko bomo iskali skupne nadnize med njim in temi ostalimi vhodnimi nizi. Označimo ostale vhodne nize z $u^1, \ldots, u^\kappa$. Zdaj bi lahko na primer rekli, da bi radi med najkrajšimi skupnimi nadnizi nizov s in t poiskali tak nadniz z, ki maksimizira vrednost

$J(z) := \sum_{r=1}^{\kappa} |\text{NSP}(z, u^r)|$ (torej vsoto dolžin najdaljših skupnih podnizov med z in vsemi dodatnimi nizi u^r). Vendar pa se računanje te hevristike izkaže za precej zamudno in celoten postopek iskanja skupnega nadniza vseh vhodnih nizov bi bil s to hevristiko počasnejši, kot bi si bilo želeti. Zato uporabimo raje naslednjo hevristiko, ki se jo bo dalo računati hitreje, pripelje pa do bolj ali manj podobno dobrih rezultatov: $\hat{J}(z) := \sum_{r=1}^{\kappa} |\text{NPP}(z, u^r)|$, pri čemer je $\text{NPP}(z, u^r)$ najdaljši tak začetek (prefiks) niza u^r, ki se pojavlja kot podniz (lahko tudi nestrnjen) v z. Med najkrajšimi skupnimi nadnizi nizov s in t torej iščemo takega z najmanjšo $\hat{J}(z)$.

S to zamislijo je nekaj težav. Naš algoritem O_2 se je opiral na dejstvo, da lahko najkrajši skupni nadniz nizov $s[1..i]$ in $t[1..j]$ dobimo tako, da z enim znakom podaljšamo nek najkrajši skupni nadniz nizov $s[1..i-1]$ in $t[1..j]$ ali pa nizov $s[1..i]$ in $t[1..j-1]$ (ali pa, če je $s_i = t_j$, nek najkrajši skupni nadniz nizov $s[1..i-1]$ in $t[1..j-1]$). Zaradi tega dejstva je bilo pri O_2 dovolj že, da smo za vsak par (i, j) izračunali dolžino $f(i, j)$ najkrajšega skupnega nadniza nizov $s[1..i]$ in $t[1..j]$; te dolžine so zadoščale tako za računanje podobnih dolžin pri večjih i in j kot tudi za to, da smo na koncu sestavili nek konkreten primer nadniza.

Ta prikladni razmislek pa ne deluje več, če bi se radi med vsemi najkrajšimi skupnimi nadnizi omejili le na tistega z največjo vrednostjo ocene $\hat{J}$. Na primer: recimo, da imamo niza $s =$ `b` in $t =$ `aa`, dodatna niza pa sta dva ($\kappa = 2$), in sicer $u^1 =$ `ab` in $u^2 =$ `ba`. Pri $i = j = 1$, torej ko gledamo najkrajše skupne nadnize nizov $s[1..1] =$ `b` in $t[1..1] =$ `a`, je dolžina takih nadnizov $f(1, 1) = 2$ in obstajata dva nadniza te dolžine: `ab` in `ba`. Oba imata $\hat{J} = 3$, torej si bo naš algoritem kot rešitev podproblema $i = j = 1$ naključno izbral enega od njiju. Če ima smolo, si bo izbral `ba`. Kmalu zatem se bo začel ukvarjati s podproblemom $i = 1, j = 2$, ko nas zanima najkrajši skupni nadniz nizov $s[1..1] =$ `b` in $t[1..2] =$ `aa`. Naš algoritem bo imel na izbiro, da bodisi rešitev podproblema $i = 0, j = 2$ (to je lahko le niz `aa`) podaljša z znakom s_1 — tako nastane `aab`, ki ima $\hat{J} = 3$ — ali pa da rešitev podproblema $i = 1, j = 1$ (za tega pa smo malo prej rekli, da imamo `ba`) podaljša z znakom t_2 — tako pa nastane `baa`, ki ima tudi $\hat{J} = 3$. V vsakem primeru torej dobimo nek skupni nadniz dolžine 3 z oceno $\hat{J} = 3$. Toda obstaja tudi skupni nadniz s-ja in t-ja z dolžino 3 in oceno $\hat{J} = 4$, namreč `aba`. Do njega bi prišli, če bi si bili pri $i = j = 1$ izbrali rešitev `ab` namesto `ba`, pa si je po nesrečnem naključju pač nismo.

Tako torej vidimo, da če si pri vsakem podproblemu (i, j) zapomnimo le eno rešitev, se bomo včasih prisiljeni naključno odločati med več enako dobrimi kandidati (z enako dolžino in enakim $\hat{J}$) in tedaj se lahko (če imamo smolo) zgodi, da si izberemo med njimi takega, zaradi katerega kasneje pri nekem večjem podproblemu ne bomo mogli priti do tistega najkrajšega skupnega nadniza, ki ima tam največji $\hat{J}$. Temu bi se lahko izognili, če bi si pri vsakem podproblemu zapomnili vse rešitve (vse najkrajše skupne nadnize nizov $s[1..i]$ in $t[1..j]$), toda to ne gre, saj jih je lahko eksponentno mnogo (v odvisnosti od i in j). Sprijazniti se moramo torej s tem, da se lahko sicer trudimo dobiti najkrajši skupni nadniz s čim večjim $\hat{J}$, ne bo pa to nujno tisti z največjim $\hat{J}$. V nadaljevanju bomo tisti konkretni nadniz, ki ga dobimo pri podproblemu (i, j), označili z $z(i, j)$ (njegova dolžina je seveda $f(i, j)$).

Druga težava pa je, da je računanje ocene $\hat{J}(z)$ zamudno početje. Dolžino $\text{NPP}(z, u^r)$ lahko izračunamo tako, da se v zanki sprehodimo po znakih z-ja in gledamo, koliko prvih znakov niza u^r smo že videli. To bo vzelo $O(|z| + |u^r|)$ časa in če

to naredimo za vse r, bomo $\hat{J}(z)$ izračunali v $O(\kappa|z| + \sum_{r=1}^{\kappa} |u^r|)$ časa. Na srečo gre tudi bolje. Spomnimo se, da je $z(i,j)$ vedno oblike xc, pri čemer je x eden od nizov $z(i-1,j)$, $z(i,j-1)$ in $z(i-1,j-1)$, znak c pa je bodisi s_i bodisi t_j. V vsakem primeru je naš $z(i,j) = xc$ podaljšek nekega malo krajšega niza x, za katerega smo nekoč prej že izračunali $\hat{J}(x)$.

Ko hočemo nato izračunati $\textsc{npp}(xc, u^r)$, lahko razmišljamo takole: če se nek prefiks u^r-ja pojavlja kot podniz v xc, se pojavlja (1) bodisi v celoti znotraj x (2) bodisi zajame tudi znak c na koncu. V primeru (1) je torej to tudi podniz niza x, najdaljši tak prefiks u^r-ja pa je kar $\textsc{npp}(x, u^r)$. V primeru (2) pa mora biti torej naš $\textsc{npp}(xc, u^r)$ oblike pc, pri čemer je p nek prefiks u^r-ja, ki se pojavlja kot podniz v x. Če je p kaj krajši od najdaljšega takega prefiksa (torej od $\textsc{npp}(x, u^r)$), bo pc kvečjemu tako dolg kot tisti najdaljši prefiks, torej takrat s primerom (2) ne bomo prišli do daljšega prefiksa kot z (1). Edini način, da nas (2) pripelje do še daljšega prefiksa, je ta, da je p kar enak $\textsc{npp}(x, u^r)$; da pa bo tedaj pc sploh res prefiks u^r-ja, mora biti seveda naslednji znak u^r-ja (za p) ravno c, torej $u^r[|\textsc{npp}(x, u^r)|] = c$. Torej lahko dolžino $\textsc{npp}(xc, u^r)$ računamo takole:

$$|\textsc{npp}(xc, u^r)| = |\textsc{npp}(x, u^r)| + \begin{cases} 1, & \text{če } u^r[|\textsc{npp}(x, u^r)|] = c \\ 0 & \text{sicer.} \end{cases}$$

Nato moramo to le še sešteti po vseh r, pa dobimo $\hat{J}(xc)$.

Iz tega razmisleka torej vidimo, da je koristno, če pri vsakem (i,j) hranimo ne le $\hat{J}(z(i,j))$, pač pa tudi dolžine posameznih prefiksov $|\textsc{npp}(z(i,j), u^r)|$ za vse $r = 1, \ldots, \kappa$. Za vsak (i,j) imamo zdaj takšno tabelo s κ elementi; da pa ne bomo po nepotrebnem zapravljali pomnilnika, lahko te tabele sproti pozabljamo, ko jih ne potrebujemo več: ko je glavna zanka našega postopka pri nekem konkretnem i, potrebujemo te tabele za trenutni i in še za $i-1$, ne pa več tistih za $i-2$, $i-3$ in tako nazaj.

Še en način, kako lahko prihranimo veliko časa, pa je tale: $\hat{J}(z(i,j))$ nas pravzaprav zanima le pri tistih (i,j), ki se jih bo dalo nekako podaljšati v nek najkrajši skupni nadniz celotnih nizov s in t. Niz $z(i,j)$ je najkrajši skupni nadniz nizov $s[1..i]$ in $[1..j]$; če hočemo iz njega narediti čim krajši skupni nadniz celotnih nizov s in t, ga moramo podaljšati še z najkrajšim skupnim nadnizom nizov $s[i+1..n]$ in $t[j+1..m]$. Označimo dolžino slednjega s $\hat{f}(i+1, j+1)$; vse te dolžine lahko izračunamo in potabeliramo na zelo podoben način, kot smo v prvotnem O_2 izračunali tabelo f. Zdaj torej vidimo, da je najkrajši tak skupni nadniz nizov s in t, ki ga je mogoče dobiti s podaljševanjem niza $z(i,j)$, dolg $f(i,j) + \hat{f}(i+1, j+1)$. Če je ta vsota enaka dolžini najkrajšega skupnega nadniza nizov s in t sploh (to dolžino najdemo v $f(n,m)$, pa tudi v $\hat{f}(1,1)$), potem je (i,j) dovolj obetaven, da moramo zanj izračunati $\hat{J}(z(i,j))$, sicer pa ga lahko preskočimo.

Zapišimo tako dobljeni algoritem še s psevdokodo. Dolžine $|\textsc{npp}(z(i,j), u^r)|$ hranimo v tabeli $P_{ij}[1..\kappa]$; nizov $z(i,j)$ ni treba hraniti eksplicitno, pač pa je dovolj že, če si pri vsakem (i,j) zapomnimo, iz katerega od sosednjih podproblemov smo ta niz dobili: iz $(i-1,j)$, iz $(i,j-1)$ ali iz $(i-1,j-1)$. V ta namen imamo tabelo ζ, ki za vsak (i,j) hrani eno od vrednosti $\{\leftarrow, \uparrow, \nwarrow\}$. Preden si ogledamo glavni del algoritma, zapišimo še tale pomožni podprogram, ki za dani znak c izračuna iz

tabele P z vrednostmi $|\textsc{npp}(x, u^r)|$ za nek niz x tabelo P' z vrednostmi $|\textsc{npp}(xc, u^r)|$ za niz xc in vrne njihovo vsoto, torej $\hat{J}(xc)$:

funkcija OCENI(P, c, P'):
  $\hat{J} := 0$;
  for $r := 1$ **to** κ:
    $p := P[r]$;
    if $p < |u^r|$ **and** $u^r[p+1] = c$ **then** $p := p + 1$;
    $P'[r] := p$; $\hat{J} := \hat{J} + p$;
  return $\hat{J}$;

Zdaj imamo vse, kar potrebujemo, da zapišemo glavni del našega algoritma:

algoritem O$_2$-P($s, t, \{u^1, \ldots, u^\kappa\}$)
  (* *Izračunaj f.* *)
  for $i := 0$ **to** n **do for** $j := 0$ **to** m:
    $c := i + j$;
    if $i > 0$ **then** $c := \min\{c, f[i-1, j] + 1\}$;
    if $j > 0$ **then** $c := \min\{c, f[i, j-1] + 1\}$;
    if $i > 0$ **and** $j > 0$ **and** $s_i = t_j$ **then** $c := \min\{c, f[i-1, j-1] + 1\}$;
    $f[i, j] := c$;
  (* *Izračunaj $\hat{f}$.* *)
  for $i := n + 1$ **downto** 1 **do for** $j := m + 1$ **downto** 1:
    $c := (n + 1 - i) + (m + 1 - j)$;
    if $i < n$ **then** $c := \min\{c, \hat{f}[i+1, j] + 1\}$;
    if $j < m$ **then** $c := \min\{c, \hat{f}[i, j+1] + 1\}$;
    if $i < n$ **and** $j < m$ **and** $s_i = t_j$ **then** $c := \min\{c, \hat{f}[i+1, j+1] + 1\}$;
    $\hat{f}[i, j] := c$;
  (* *Izračunaj P, $\hat{J}$ in ζ.* *)
  for $i := 0$ **to** n:
    if $i > 1$ **then** pozabi tabele $P_{i-2,j}$ za vse $j = 0, \ldots, m$;
    for $j := 0$ **to** m:
      if $f[i, j] + \hat{f}[i+1, j+1] > f[n, m]$ **then continue**;
      if $i = 0$ **and** $j = 0$:
        for $r := 1$ **to** κ **do** $P_{ij}[r] := 0$;
        continue;
      naj bodo $P_\leftarrow$, $P_\uparrow$, $P_\nwarrow$ tri pomožne tabele s po κ elementi;
      if $i > 0$ **and** $f[i-1, j] + 1 = f[i, j]$ **then** $\hat{J}_\leftarrow := \textsc{Oceni}(P_{i-1,j}, s_i, P_\leftarrow)$
                                                else $\hat{J}_\leftarrow := -\infty$;
      if $j > 0$ **and** $f[i, j-1] + 1 = f[i, j]$ **then** $\hat{J}_\uparrow := \textsc{Oceni}(P_{i,j-1}, t_j, P_\uparrow)$
                                                else $\hat{J}_\uparrow := -\infty$;
      if $i > 0$ **and** $j > 0$ **and** $s_i = t_j$ **and** $f[i-1, j-1] + 1 = f[i, j]$ **then**
                                                $\hat{J}_\nwarrow := \textsc{Oceni}(P_{i-1,j-1}, s_i, P_\nwarrow)$ **else** $\hat{J}_\nwarrow := -\infty$;
      $d :=$ tista izmed $\{\leftarrow, \uparrow, \nwarrow\}$, ki dá največjo $\hat{J}_d$;
      $P_{ij} := P_d$; $\zeta[i, j] := d$;

(* *Sestavi primeren izhodni niz.* *)
$i := n$; $j := m$; alociraj izhodni niz u dolžine $f[n, m]$;
while $i > 0$ **or** $j > 0$:
  $q := f[i, j]$; $d := \zeta[i, j]$;
  if $d = \leftarrow$ **then** $u_q := s_i$; $i := i - 1$;
  else if $d = \uparrow$ **then** $u_q := t_j$; $j := j - 1$;
  else (* *torej je* $d = \nwarrow$ *) $u_q := s_i$; $i := i - 1$; $j := j - 1$;
return u;

V naslednjem razdelku si bomo ogledali, kako lahko ta algoritem s pridom uporabimo pri iskanju čim krajšega skupnega nadniza več vhodnih nizov.

Dodajanje vhodnih nizov po vrsti. Zelo preprost način, da pridemo do skupnega nadniza vseh vhodnih nizov, je ta, da začnemo z enim od njih in nato v vsakem koraku poiščemo najkrajši skupni nadniz med dosedanjim nadnizom in naslednjim vhodnim nizom. Dolžina nadniza, ki nam na koncu nastane, je odvisna od tega, v kakšnem vrstnem redu jemljemo vhodne nize. Ta vrstni red lahko opišemo s permutacijo π nad indeksi $\{1, \ldots, k\}$. Tako dobimo naslednji postopek:[19]

algoritem R:
1 $s :=$ prazen niz;
2 **for** $r := 1$ **to** k:
3 $s :=$ najkrajši skupni nadniz nizov s in $t^{\pi(r)}$;
4 **return** s;

V vrstici 3 lahko najkrajši skupni nadniz dveh nizov poiščemo z algoritmom O_2, še boljše rezultate (krajše nadnize) pa bomo dobili, če uporabimo O_2-P; pokličemo ga kot $O_2\text{-P}(s, t^{\pi(r)}, \{t^{\pi(r+1)}, \ldots, t^{\pi(k)}\})$.

Dolžina nadniza, ki ga dobimo na koncu tega postopka, je odvisna od vrstnega reda π, v katerem smo dodajali nadnize, pa tudi od tega, kako smo si v posameznem izvajanju vrstice 3 izbrali najkrajši skupni nadniz nizov s in $t^{\pi(r)}$. Kot smo videli že zgoraj, se namreč lahko zgodi, da ta najkrajši skupni nadniz ni enoličen, ampak obstaja več različnih (enako dolgih) najkrajših skupnih nadnizov; čeprav so enako dolgi, pa niso nujno vsi enako primerni za nadaljevanje postopka (v naslednjih iteracijah glavne zanke).

Ker je vnaprej težko reči, kateri vrstni red π in kakšne odločitve glede izbiranja nadniza v vrstici 3 nam bodo dale na koncu postopka najkrajši niz, je še najpreprosteje, če poženemo postopek po večkrat za različne, naključno izbrane permutacije π (z drugimi besedami, pred vsakim poskusom naključno premešamo vrstni red vhodnih nizov), pa tudi nadniz v vrstici 3 vsakič izbiramo naključno; med vsemi tako dobljenimi rešitvami pa si zapomnimo najkrajšo.[20]

[19] V literaturi o problemu najkrajšega skupnega nadniza temu algoritmu pogosto pravijo preprosto Greedy, kar je sicer malo neugodno, ker deluje po požrešnem načelu tudi več drugih algoritmov za ta problem (npr. tisti, ki smo jih že videli zgoraj in ki sestavljajo nadniz znak po znak).

[20] Omeniti pa velja, da četudi bi lahko preizkusili vseh $k!$ možnih vrstnih redov (kar je sicer praktično mogoče le pri majhnih k) in če bi pri vsakem izvajanju vrstice 3 lahko preizkusili vse možne najkrajše skupne nadnize nizov s in $t^{\pi(r)}$, to še ne pomeni, da bi nekoč dobili najkrajši možni skupni nadniz sploh (torej takega, kot ga dobimo z algoritmom O_k). Na primer, če imamo tri vhodne nize {*abbaaa*, *aaabba*, *baaaab*}, je mogoče pokazati, da algoritem R vedno najde nek nadniz dolžine 10, najboljša možna rešitev pa je dolga le 9 znakov (to je niz *abaabaaba*).

Lahko si zapomnimo tudi vrstni red π, pri katerem smo ta najkrajši znani nadniz (ki seveda v splošnem ni nujno tudi najkrajši nadniz sploh) našli. Ta vrstni red lahko potem poskušamo še izboljšati z neke vrste *lokalno optimizacijo*: poskusimo naš vrstni red malo spremeniti, na primer tako, da zamenjamo položaj dveh nizov v njem. Če tako spremenjeni vrstni red pripelje do krajšega nadniza, spremembo obdržimo, sicer pa jo zavržemo in poskusimo kakšno drugo. Postopek ustavimo, ko se naveličamo čakati ali pa če pri nekem vrstnem redu preizkusimo vse možne spremembe, pa nobena od njih ne pripelje do krajšega nadniza. Tako dobimo približno takšen postopek:

algoritem R-L:
1 večkrat poženi algoritem R za različne naključne vrstne rede π
in si zapomni tisti π, pri katerem je nastal najkrajši nadniz s;
2 $T :=$ prazna množica;
3 **ponavljaj**, dokler ne mine dovolj časa:
4 izberi si naključna indeksa i in j tako, da bo $1 \le i < j \le k$
in $(i, j) \notin T$;
5 $\pi' := \pi$; $\pi'[i] := \pi[j]$; $\pi'[j] := \pi[i]$;
6 $s' :=$ nadniz, ki ga vrne algoritem R pri vrstnem redu π';
7 **if** $|s'| < |s|$ **then** $s := s'$; $\pi := \pi'$; $T :=$ prazna množica;
8 **else** dodaj (i, j) v T;
9 **return** s;

Neobetavne poskuse sprememb si torej zapisujemo v množico T (*tabu list*), da jih kasneje ne bomo preizkusili še enkrat; ko pa najdemo nek premik na bolje, spisek tabujev pobrišemo (vrstica 7). V vrstici 4 se lahko tudi zgodi, da vsebuje T že vse možne pare (i, j), kar pomeni, da smo našli lokalni minimum in moramo postopek ustaviti.

Ta postopek bi se dalo na razne načine še malo izboljšati. Na primer, prva sprememba v našem vrstnem redu nastopi na indeksu i, torej nadniza ne bi bilo treba računati vsakič od začetka, ampak bi lahko nadaljevali pri tistem nadnizu, ki je nastal iz prvih $i-1$ vhodnih nizov. Tako prihranimo nekaj časa, vendar povečamo porabo pomnilnika, saj si moramo poleg končnega nadniza zapomniti še vseh $k-1$ vmesnih. Še ena možna izboljšava je, da bi postopek včasih sprejel tudi spremembe na slabše, vendar s tem manjšo verjetnostjo, čim bolj poslabšajo rešitev. Temu prijemu pravimo *simulirano ohlajanje* (*simulated annealing*) in nam lahko pomaga, da se izognemo kakšnim neobetavnim lokalnim ekstremom.

Razbijanje vhodnih nizov na krajše kose. Zgoraj smo videli, da algoritem G pogosto daje dobre rezultate, vendar je uporabno hiter le, če so vhodni nizi (in njihov skupni nadniz) dovolj kratki. Če ima naš testni primer dolge vhodne nize, jih lahko poskusimo razbiti na več krajših. Izberimo si nek b in razbijmo vsak vhodni niz t^r na b približno enako dolgih kosov $t^{r,1}, \ldots, t^{r,b}$:

$$t^{r,i} = t^r[\lfloor n_r(i-1)/b \rfloor + 1 .. \lfloor n_r i/b \rfloor].$$

Istoležne kose vseh nizov združimo v nek (čim krajši) skupni nadniz in nato tako dobljene nadnize staknimo skupaj, pa imamo rešitev prvotnega problema:

algoritem C(b):
1 razbij vsak vhodni niz t^r na b kosov $t^{r,i}$, $i = 1, \ldots, b$;
2 **for** $i := 1$ **to** b:
3 $s^i :=$ čim krajši skupni nadniz nizov $t^{1,i}, t^{2,i}, \ldots, t^{k,i}$;
4 $s := s^1 s^2 \ldots s^b$; **return** s;

S tem, ko smo vhodne nize razcepili na b kosov in nato vsakič iskali nadniz le za istoležne kose, smo seveda vpeljali v postopek neko dodatno omejitev, ki je prvotna naloga ni imela, zato nas ne bo presenetilo, da so rešitve zaradi tega lahko malo slabše (nastane malo daljši nadniz). To poslabšanje je tem večje, čim večji je b (na čim več kosov razcepljamo vhodne nize), zato je pametno vzeti najmanjši b, ki si ga še lahko privoščimo. Glavna omejitev tu je, kdaj postanejo kosi vhodnih nizov dovolj kratki, da bomo v vrstici 3 lahko z nekim drugim algoritmom poiskali njihov (čim krajši) skupni nadniz. Na primer, če tam uporabimo algoritem G, si načeloma želimo, da obstaja nadniz s^i dolžine največ kakšnih 1000 znakov, sicer se začne izvajati neugodno dolgo.

Drugi algoritmi. Za konec le na kratko omenimo še nekaj drugih algoritmov, ki so jih v literaturi predlagali za iskanje čim krajših skupnih nadnizov.

Zapišimo po vrsti vse znake naše abecede Σ, vsakega po enkrat; dobimo nek niz dolžine a; staknimo skupaj d izvodov tega niza. Nastal je niz dolžine $d \cdot a$, ki ima za podnize prav vse nize (nad abecedo Σ) dolžine d ali manj. Če torej za d vzamemo $\max_r |t^r|$, imamo primeren skupni nadniz vseh naših vhodnih nizov. Ta algoritem se imenuje ALPHABET in je zanimiv predvsem s teoretičnega vidika, ker daje neko zagotovilo o tem, kako dolg je (v najslabšem primeru) lahko najkrajši skupni nadniz: največ a-krat toliko, kolikor je dolg najdaljši od vhodnih nizov.

Zgoraj smo videli algoritem R, ki dodaja vhodne nize v nadniz enega po enega. Še ena podobna ideja je, da najprej združujemo vhodne nize po dva skupaj v nadnize; tako iz prvotnega nabora k nizov dobimo $\lceil k/2 \rceil$ malo daljših nadnizov. Na tem novem naboru nizov isti postopek ponovimo in tako nadaljujemo, dokler ne ostane en sam niz. Temu postopku pravijo „turnir“ (TOURNAMENT), vendar je pri naših poskusih na naših testnih primerih dajal veliko slabše rezultate (daljše nadnize) kot algoritem R.

Reduce-expand [2] najprej oklesti vhodne nize tako, da od več zaporednih enakih znakov obdrži le po enega; nato poišče skupni nadniz tako okleščenih nizov; nato pa ta nadniz počasi napihuje nazaj tako, da podvaja znake v njem in sproti briše tiste, ki jih ne potrebuje. Pri naših poskusih se ni preveč obnesel.

Deposition-reduction [8, 14] poskuša nek že znan skupni nadniz s izboljšati tako, da ga na vse možne načine razbije na dva dela, levega s_L in desnega s_D; nato še vsak vhodni niz t^r razbije na levi in desni del glede na to, v katerem delu nadniza leži; nato poskuša za leve dele vhodnih nizov najti nov skupni nadniz in če je ta kaj krajši od s_L, ga uporabi v s namesto dosedanjega s_L. V obliki, kot jo opisujejo avtorji, je videti ta postopek precej počasen, je pa s prijemi tega tipa vsekakor mogoče včasih še malo skrajšati nadnize, ki smo jih dobili s kakšnim drugim algoritmom.

Precej avtorjev uporablja tudi pristope, ki sestavljajo nadniz s kakšno od znanih hevristik za preiskovanje prostora, kot sta na primer iskanje v snopu (*beam search*) [3, 12] in optimizacija po zgledu kemijskih reakcij (*chemical reaction optimization*, CRO) [17].

Testni primeri
Pripravili smo 60 testnih primerov, ki pokrivajo širok razpon kombinacij števila vhodnih nizov, dolžine vhodnih nizov, dolžine skupnega nadniza in velikosti abecede. Vsi pa se držijo omejitve iz besedila naloge, namreč da skupna dolžina vhodnih nizov ne presega milijon znakov. Glede na način, kako smo pripravili vhodne nize, lahko ločimo naslednje tipe testnih primerov:

- P(d, x): s postopkom x smo najprej pripravili niz dolžine d, nato pa smo za vhodne nize uporabljali njegove naključne podnize. Posamezni podniz dobimo tako, da naključno izberemo množico različnih indeksov od 1 do d, jih uredimo naraščajoče, odčitamo s teh indeksov črke našega niza in jih staknemo skupaj. Za x smo uporabili naslednje možnosti:
  - $x =$ S: posamezne črke niza so izbrane naključno in neodvisno od ostalih črk iz abecede 5 znakov (vsi znaki so enako verjetni).
  - $x =$ V: kot $x =$ S, le da imamo abecedo 22 znakov, ki niso vsi enako verjetni, ampak so verjetnosti porazdeljene približno po potenčni porazdelitvi (podobno kot v besedilih v naravnem jeziku).
  - $x =$ M: niz smo zgenerirali z markovskim modelom na črkah; to pomeni, da je verjetnostna porazdelitev naslednje črke odvisna od tega, kakšne so bile prejšnje 3 (že zgenerirane) črke. Te verjetnosti smo ocenili na slovarju približno 350 tisoč angleških besed.
  - $x =$ R: niz je „realističen“, dobili smo ga tako, da smo iz korpusa angleških besedil odstranili vse ne-črkovne znake.
- M(d): vsak vhodni niz posebej smo zgenerirali z enakim markovskim modelom, ki je bil že omenjen zgoraj; vhodne nize smo dodajali tako dolgo, dokler njihova skupna dolžina ni dosegla d.
- D(d): kot vhodne nize smo vzeli naključen izbor besed iz slovarja (že omenjeni seznam 350 tisoč angleških besed), pri čemer smo jih jemali tako dolgo, dokler njihova skupna dolžina ni dosegla d.

Podrobnejši opis posameznih testnih primerov je v tabeli z rezultati spodaj.

Najboljše nam znane rešitve
Tabela na naslednjih straneh podaja nekaj podrobnosti o testnih primerih in o najboljših nam znanih rešitvah za vsakega od njih. Stolpci od leve proti desni pomenijo: (1) # = zaporedna številka testnega primera; (2) k = število vhodnih nizov; (3) vsota dolžin vhodnih nizov; (4) povprečje dolžin vhodnih nizov; (5) postopek, s katerim je bil ta testni primer pripravljen; (6) dolžina najkrajšega nam znanega skupnega nadniza (zvezdica označuje tiste, pri katerih vemo, da je to res optimalna rešitev, torej najkrajši skupni nadniz sploh); (7) stopnja prekrivanja = razmerje med skupno dolžino vhodnih nizov in dolžino najkrajšega nam znanega skupnega nadniza; (8) postopek, s katerim je bil pridobljen najkrajši nam znani skupni nadniz. V zadnjem stolpcu zapis W(ℓ, α) pomeni, da smo uporabili postopek W-L(ℓ) z utežmi PMM z eksponentom α.

#	Št. nizov	Skupna dolž.	Povpr. dolž.	Kako pripravljen	Dolž. nadniza	Stop. prekr.	Katera rešitev
1	100	1 229	12,3	P(100, S)	50	24,6	G
2	300	3 669	12,2	P(100, S)	58	63,3	G
3	1 000	12 312	12,3	P(100, S)	65	189,4	G
4	100	1 501	15,0	P(1000, S)	58	25,9	G
5	300	4 505	15,0	P(1000, S)	65	69,3	G
6	30	224 722	7 491	P(10^5, S)	22 860	9,8	W(6, 6)
7	50	363 339	726	P(10^5, S)	23 904	15,2	W(7, 9)
8	100	774 567	7 746	P(10^5, S)	26 084	29,7	W(6, 11)
9	100	1 223	12,2	P(100, V)	83	14,7	G
10	300	3 648	12,2	P(100, V)	100	36,5	G
11	1 000	12 512	12,5	P(100, V)	101	123,9	G
12	100	1 523	15,2	P(1000, V)	131	11,6	G
13	300	4 490	15,0	P(1000, V)	153	29,3	G
14	1 000	14 876	14,9	P(1000, V)	169	88,0	G
15	100	9 520	95,2	P(1000, V)	816	11,7	G
16	300	31 893	106,3	P(1000, V)	959	33,3	G
17	1 000	15 115	15,1	P(10^4, V)	190	79,6	G
18	10 000	150 475	15,0	P(10^4, V)	227	662,9	G
19	50	51 551	1 031	P(10^4, V)	7 291	7,1	R-L + O_2-P
20	100	100 654	1 007	P(10^4, V)	8 113	12,4	G
21	30	234 545	7 818	P(10^5, V)	46 521	5,0	R-L + O_2-P
22	50	367 352	7 347	P(10^5, V)	51 310	7,2	R-L + O_2-P
23	100	751 859	7 519	P(10^5, V)	62 485	12,0	R-L + O_2-P
24	1 059	9 999	9,4	D(10^4)	150	66,7	G
25	10 603	100 000	9,4	D(10^5)	190	526,3	G
26	105 779	999 995	9,5	D(10^6)	226	4 424	G
27	1 111	9 999	9,0	M(10^4)	159	62,9	G
28	11 055	99 999	9,0	M(10^5)	196	510,2	G
29	109 650	999 994	9,1	M(10^6)	227	4 405	G
30	100	1 184	11,8	P(100, M)	106	11,2	G
31	300	3 867	12,9	P(100, M)	105	36,8	G
32	100	1 470	14,7	P(1000, M)	154	9,5	G
33	300	4 456	14,9	P(1000, M)	196	22,7	G
34	100	1 262	12,6	P(100, R)	102	12,4	G
35	300	3 813	12,7	P(100, R)	103	37,0	G
36	100	1 473	14,7	P(1000, R)	156	9,4	G
37	300	4 462	14,9	P(1000, R)	195	22,9	G
38	1 000	14 865	14,9	P(10^4, R)	239	62,2	G
39	10 000	150 145	15,0	P(10^4, R)	288	521,3	G
40	100	100 746	1 008	P(10^4, R)	10 518	9,6	C(6) + G
41	30	201 340	6 711	P(10^5, R)	48 937	4,1	R-L + O_2-P
42	50	370 670	7 413	P(10^5, R)	63 966	5,8	R-L + O_2-P
43	100	723 977	7 240	P(10^5, R)	76 145	9,5	R-L + O_2-P
44	2	995 531	497 766	P(10^6, V)	701 370*	1,4	O_2-S
45	4	196	49,0	P(100, V)	89*	2,2	O_k
46	7	144	20,6	P(80, V)	53*	2,7	O_k
47	9	92	10,2	P(60, V)	32*	2,9	O_k
48	9	108	12,0	P(70, V)	38*	2,8	O_k
49	9	156	17,3	P(150, V)	60*	2,6	O_k-S
50	4	199	49,8	P(100, V)	78*	2,6	O_k
51	7	139	19,9	P(80, V)	40*	3,5	O_k
52	9	87	9,7	P(60, V)	23*	3,8	O_k
53	9	108	12,0	P(70, V)	29*	3,7	O_k

#	Št. nizov	Skupna dolž.	Povpr. dolž.	Kako pripravljen	Dolž. nadniza	Stop. prekr.	Katera rešitev
54	9	154	17,1	P(150, V)	42⋆	3,7	O_k-S
55	4	201	50,3	P(10^4, M)	126⋆	1,6	O_k
56	7	138	19,7	P(10^4, M)	70⋆	2,0	O_k
57	9	86	9,6	P(10^4, M)	39⋆	2,2	O_k
58	9	110	12,2	P(10^4, M)	51⋆	2,2	O_k
59	9	156	17,3	P(10^4, M)	72⋆	2,2	O_k-S
60	2	996 063	498 031	P(10^6, M)	786 274⋆	1,3	O_2-S

Literatura

[1] A. V. Aho, D. S. Hirschberg, J. D. Ullman. Bounds on the complexity of the longest common subsequence problem. *J. of the ACM*, 23(1):1–12 (Jan. 1976).

[2] P. Barone, P. Bonizzoni, G. D. Vedova, G. Mauri. An approximation algorithm for the shortest common supersequence problem: an experimental analysis. *Proc. of the 2001 ACM Symposium on Applied Computing*, pp. 56–60.

[3] C. Blum, C. Cotta, A. J. Fernández, F. Gallardo. A probabilistic beam search approach to the shortest common supersequence problem. *Proc. of the 7th European Conf. on Evolutionary Computation in Combinatorial Optimization* (EvoCOP 2007), pp. 36–47.

[4] Jürgen Branke, M. Middendorf. Searching for shortest common supersequences by means of a heuristic-based genetic algorithm. *Proc. of the 2nd Nordic Workshop on Genetic Algorithms and Their Applications*, pp. 105–114 (1996).

[5] Jürgen Branke, M. Middendorf, F. Schneider. Improved heuristics and a genetic algorithm for finding short supersequences. *OR Spektrum*, 20:39–45 (1998).

[6] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343 (June 1975).

[7] T. Jiang, M. Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal of Computing*, 24(5):1122–1139 (October 1995).

[8] K. Ning, K. P. Choi, H. W. Leong, L. Zhang. A post-processing method for optimizing synthesis strategy for oligonucleotide microarrays. *Nucleic Acids Research*, 33(17): e144 (2005).

[9] S. Kasif, Z. Weng, A. Derti, R. Beigel, C. DeLisi. A computational framework for optimal masking in the synthesis of oligonucleotide microarrays. *Nucleic Acids Research*, 30(20):e103 (October 15, 2002).

[10] A. Lagoutte, S. Tavenas. The complexity of Shortest Common Supersequence for inputs with no identical consecutive letters. `arXiv.org/abs/1309.0422v2`, January 9, 2015.

[11] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336 (April 1978).

[12] S. R. Mousavi, F. Bahri, F. S. Tabataba. An enhanced beam search algorithm for the shortest common supersequence problem. *Engineering Applications of Artificial Intelligence*, 25(3):457–67 (April 2012).

[13] M. Middendorf. More on the complexity of common superstring and supersequence problems. *Theoretical Computer Science*, 125:205–28 (1994).

[14] K. Ning, H. W. Leong. Towards a better solution to the shortest common supersequence problem: the deposition and reduction algorithm. *BMC Bioinformatics*, 7 (Suppl 4): S12 (2006).

[15] M. René, M. Middendorf. An island model based ant system with lookahead for the shortest supersequence problem. *Parallel Problem Solving from Nature, 5th International Conference*, 1998. LNCS 1498, pp. 692–701.

[16] K.-J. Räihä, E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16(2):187–98 (1981).

[17] C. M. K. Saifullah, M. R. Islam. Solving shortest common supersequence problem using chemical reacton optimization. *Proc. 5th Int. Conf. on Informatics, Electronics and Vision* (ICIEV 2016), pp. 50–55.

[18] V. G. Timkovskii. Complexity of common subsequence and supersequence problems and related problems. Cybernetics 25(5):565–580, September 1989.

UNIVERZITETNI PROGRAMERSKI MARATON

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

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

UPM poteka v štirih krogih (dva spomladi in dva jeseni), pri čemer se za končno razvrstitev pri vsaki ekipi zavrže najslabši rezultat iz prvih treh krogov, četrti (finalni) krog pa se šteje dvojno. Najboljše ekipe se uvrstijo na srednjeevropsko regijsko tekmovanje (CERC, ki je bilo letos 17.–19. novembra 2017 v Zagrebu), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bo 15.–20. aprila 2018 v Pekingu).

Na letošnjem UPM je sodelovalo 57 ekip s skupno 161 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	Jure Slak, Maks Kolman, Marko Ljubotina (FMF)	22	27:22:58
2	Žiga Željko (Gim. Bežigrad), Martin Peterlin (Vegova Lj.), Žan Knafelc (FDV)	18	20:46:14
3	Filip Koprivec (FMF), Filip Peter Lebar (FE), Luka Kolar (FRI)	18	24:35:00
4	Luka Avbreht, Samo Kralj, Žiga Župančič (FMF)	18	25:31:10
5	Jasna Urbančič (FRI), Patrik Zajec (FRI + FMF)	17	24:32:06
6	Žiga Šmelcer (FE), Žiga Gradišar (FMF), Lojze Žust (FRI)	16	25:38:18
7	Izak Glasenčnik, Nejc Maček, Robi Novak (FERI)	16	31:05:55
8	Tim Poštuvan (Gim. Vič), Simon Weiss (FMF), Vid Kocijan (FRI + FMF)	15	18:08:12
9	Matej Tomc (FE), Lenart Treven, Luka Lodrant (FMF)	15	21:35:53
10	Mitja Žalik (II. gim. Maribor), Aljaž Jeromel, Grega Premoša (FERI)	15	24:14:01
11	Aljaž Eržen, Žiga Vene, Marko Rus (FRI + FMF)	13	16:50:00
12	Gregor Kikelj, Nejc Šuklje, Matevž Rom (ŠC NM, SEŠTG)	11	12:15:01
13	Nejc Kadivnik, Andrejaana Andova, Miha Zadravec (FRI)	10	13:40:43
14	Benjamin Benčina (FMF), Rok Kos, Bor Brecelj (FRI + FMF)	10	16:20:21
15	Matic Oskar Hajšen, Ines Meršak, Jan Rozman (FMF)	9	10:25:13
16	Mikita Akulich, Daniil Baldouski, Anton Uramer (FAMNIT)	9	15:05:31
17	Mateja Hrast (FMF)	9	17:34:02
18	Dejan Skledar, Matej Drobnič (FERI)	8	13:44:50
19	Tilen Jesenko, Goran Tubić, Gašper Moderc (FAMNIT)	7	9:10:02
20	Anja Petković, Vesna Iršič, Žiga Lukšič (FMF)	7	11:23:17
21	Miha Štravs, Miha Bastl, Jure Cezar (FRI + FMF)	7	14:11:23
22	Mirt Hlaj, Leo Gombač, Jan Grbac (FAMNIT)	6	5:11:59
23	Jan Mikolič, Gal Meznarič (FERI)	6	6:57:27
24	Eda Kaja, Lisi Qarkaxhija, Arbër Avdullahu (FAMNIT)	6	18:21:15
25	Petr Barta, Ilcho Koteski, Nemanja Soldo (FRI)	5	8:38:20
26	Marcel Čampa (FMF), Fedja Beader (FRI + FMF)	5	12:14:38
27	Jure Taslak, Samo Mikuš (FRI + FMF), Mihael Švigelj (FRI)	5	17:27:01
28	Isidora Rapajić, Mira Krbezlija (FAMNIT)	4	1:56:55
29	Vid Drobnič, Žiga Patačko Koderman, Matej Marinko (Gim. Vič)	4	5:02:16
30	Eva Erzin, Eva Zmazek, Nina Slivnik (FMF)	4	6:48:09
31	Jernej Rudi Finžgar, Luka Medic, Metod Jazbec (FMF)	4	10:25:29
32	Domen Dolanc, Tomaž Martinčič (FRI)	4	12:01:32
33	Jakob Valič, Mihael Pačnik, Matija Bolko (FMF)	4	17:00:38
34	Klemen Kozjek, Kristjan Sešek, Primož Črnigoj (FRI)	3	2:42:34
35	Erik Langerholc (FMF), Domen Lušina, Tadej Ciglarič (FRI)	3	2:42:56
36	Nina Mrzelj, Darja Peternel (FRI + FMF), Matevž Lenič (FRI)	3	3:15:31
37	Miha Rot, Gašper Domen Romih, Matej Logar (FMF)	3	3:15:55
38	Bor Grošelj Simić, Jakob Zmrzlikar, Jon Mikoš (Gim. Vič)	3	3:24:53
39	Tristan Višnar, Domen Kavran (FERI)	3	3:38:03

* 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	Klemen Krsnik, Katja Logar, Timotej Knez (FRI)	3	4:02:59
41	Tomaž Cvetko, Janez Turnšek (FMF), Rok Krumpak (FRI + FMF)	3	4:40:19
42	Sabina Marancina, Branka Kojić, Maja Umek (FRI + FMF)	3	4:54:49
43	Adam Veselič, Kristjan Žagar, Gregor Štefanič (FERI)	3	8:28:24
44	Žan Skamljič, Jan Pomer (FERI)	2	0:47:09
45	Domen Vidovič, Alisa Paulinič (FERI), Anja Goričan (FNM)	2	2:44:04
46	Anže Štular, Maj Škerjanc, Andrej Kolar Požun (FMF)	2	2:48:01
47	Sandi Režonja, Marko Drvarič, Luka Androjna (FRI)	2	3:53:14
48	Tomaž Krajcar, Niko Kolar, Tilen Pader (FERI)	2	7:54:30
49	Lara Jerman (FKKT), Klara Nosan (FRI), Lidija Magdevska (FRI + FMF)	1	0:06:03
50	Tobias Mihelčič, Mitja Žnideršič (FRI), Klemen Praznik (FMF)	1	0:17:08
51	Mihael Trajbarič, Klemen Kobau, Alja Kolenc (FRI + FMF)	1	0:40:40
52	Emilio Baresi, Radovan Lapár, Kalvis Kazoks (FRI)	1	1:30:58
53	Hristijan Jankulovski, Slavica Filiposka, Bozen Jovanovski (FRI)	1	2:11:40
54	Gregor Vavdi, Blaž Poljanec, Tina Zwittnig (FMF)	1	2:47:09
55	Deni Cerovac, Roni Likar (FRI), Dan Toškan (FMF)	1	2:50:28
56	Amela Špica, Amra Omanović, Magda Nowak-Trzos (FRI)	1	3:39:35
57	Aljaž Soderžnik, Kristjan Kotnik, Simon Slemenšek (FERI)	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 2, 3 in 4 kot predstavnice Univerze v Ljubljani, ekipa 7 kot predstavnica Univerze v Mariboru in ekipa 16 kot predstavnica Univerze na Primorskem. V konkurenci 69 ekip s 30 univerz iz 7 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
36	Žan Knafelc, Martin Peterlin, Žiga Željko	2	1:37
38	Luka Avbreht, Samo Kralj, Žiga Zupančič	2	3:19
39	Luka Kolar, Filip Koprivec, Filip Peter Lebar	2	3:22
43	Izak Glasenčnik, Nejc Maček, Robi Novak	2	4:03
57	Mikita Akulich, Daniil Baldouski, Anton Uramer	1	2:04

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

ANKETA

Tekmovalcem vseh treh skupin smo na tekmovanju skupaj z nalogami razdelili tudi naslednjo anketo. Rezultati ankete so predstavljeni na str. 219–226.

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 VB.NET. Bi rad uporabljal kakšen drug programski jezik? Če da, katerega? _
_____

Katere od naslednjih jezikovnih konstruktov in programerskih prijemov znaš uporabljati?	ne poznam	da, slabo	da, dobro
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz standardnega vhoda ali pa ju zapisati na standardni izhod?	☐	☐	☐
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz datoteke ali pa ju zapisati v datoteko?	☐	☐	☐
Tabele (**array**):			
— enodimenzionalne	☐	☐	☐
— dvodimenzionalne	☐	☐	☐
— večdimenzionalne	☐	☐	☐
Znaš napisati svoj podprogram (**procedure**, **function**)	☐	☐	☐

Poznaš rekurzijo	☐	☐	☐
Kazalce, dinamično alokacijo pomnilnika (New/Dispose, GetMem/FreeMem, malloc/free, **new**/**delete**, ...)	☐	☐	☐
Zanka **for**	☐	☐	☐
Zanka **while**	☐	☐	☐
Gnezdenje zank (ena zanka znotraj druge)	☐	☐	☐
Naštevni tipi (*enumerated types* — **type** ImeTipa = (Ena, Dve, Tri) v pascalu, **typedef enum** v C/C++)	☐	☐	☐
Strukture (**record** v pascalu, **struct**/**class** v C/C++)	☐	☐	☐
and, **or**, **xor**, **not** kot aritmetični operatorji (nad biti celoštevilskih operandov namesto nad logičnimi vrednostmi tipa boolean) (v C/C++/C#/javi: &, |, ^, ~)	☐	☐	☐
Operatorja **shl** in **shr** (v C/C++/C#/javi: <<, >>)	☐	☐	☐
Znaš uporabiti kakšnega od naslednjih razredov iz standardnih knjižnic:			
hash_map, hash_set, unordered_map, unordered_set (v C++), Hashtable, HashSet (v javi/C#), Dictionary (v C#), dict, set (v pythonu)	☐	☐	☐
map, set (v C++), TreeMap, TreeSet (v javi), SortedDictionary (v C#)	☐	☐	☐
priority_queue (v C++), PriorityQueue (v javi), heapq (v pythonu)	☐	☐	☐

[Naslednja skupina vprašanj se je ponovila za vsako nalogo po enkrat.]
Zahtevnost naloge: ☐ prelahka ☐ lahka ☐ primerna ☐ težka ☐ pretežka ☐ ne vem
Naloga je (ali: bi) vzela preveč časa: ☐ da ☐ ne ☐ ne vem
Mnenje o besedilu naloge:
— dolžina besedila: ☐ prekratko ☐ primerno ☐ predolgo
— razumljivost besedila: ☐ razumljivo ☐ težko razumljivo ☐ nerazumljivo
Naloga je bila: ☐ zanimiva ☐ dolgočasna ☐ že znana ☐ povprečna
Si jo rešil(a)?
- ☐ nisem rešil(a), ker mi je zmanjkalo časa za reševanje
- ☐ nisem rešil(a), ker mi je zmanjkalo volje za reševanje
- ☐ nisem rešil(a), ker mi je zmanjkalo znanja za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo časa za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo volje za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo znanja za reševanje
- ☐ rešil(a) sem celo

Ostali komentarji o tej nalogi: _____

Katera naloga ti je bila najbolj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? _____

Katera naloga ti je bila najmanj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? _____

Na letošnjem tekmovanju ste imeli tri ure / pet ur časa za pet nalog.
Bi imel(a) raje: ☐ več časa ☐ manj časa ☐ časa je bilo ravno prav
Bi imel(a) raje: ☐ več nalog ☐ manj nalog ☐ nalog je bilo ravno prav

Kakršne koli druge pripombe in predlogi. Kaj bi spremenil(a), popravil(a), odpravil(a), ipd., da bi postalo tekmovanje zanimivejše in bolj privlačno? _____

Kaj ti je bilo pri tekmovanju všeč? _____

Kaj te je najbolj motilo? _____

Če imaš kaj vrstnikov, ki se tudi zanimajo za programiranje, pa se tega tekmovanja niso udeležili, kaj bi bilo po tvojem mnenju treba spremeniti, da bi jih prepričali k udeležbi?

Poleg tekmovanja bi radi tudi v preostalem delu leta organizirali razne aktivnosti, ki bi vas zanimale, spodbujale in usmerjale pri odkrivanju računalništva. Prosimo, da nam pomagate izbrati aktivnosti, ki vas zanimajo in bi se jih zelo verjetno udeležili.

Udeležil bi se oz. z veseljem bi spremljal:

- ☐ izlet v kak raziskovalni laboratorij v Evropi (po možnosti za dva dni)
- ☐ poletna šola računalništva (1 teden na IJS, spanje v dijaškem domu)
- ☐ poletna praksa na IJS
- ☐ predstavitve novih tehnologij (.NET, mobilni portali, programiranje „vgrajenih računalnikov“, strojno učenje, itd.) (1× mesečno)
- ☐ predavanja o algoritmih in drugih temah, ki pridejo prav na tekmovanju (1× mesečno)
- ☐ reševanje tekmovalnih nalog (naloge se rešuje doma in bi bile delno povezane s temo, predstavljeno na predavanju; rešitve se preveri na strežniku) (1× mesečno)
- ☐ tvoji predlogi: _____

Vesel(a) bi bil pomoči pri:

- ☐ iskanju štipendije
- ☐ iskanju podjetij, ki dijakom ponujajo njim prilagojene poletne prakse in druge projekte, kjer se ob mentorstvu lahko veliko naučijo.

Ali si pri izpolnjevanju ankete prišel/la do sem? ☐ da ☐ ne

Hvala za sodelovanje in lep pozdrav! Tekmovalna komisija

REZULTATI ANKETE

Anketo je izpolnilo 84 tekmovalcev prve skupine, 25 tekmovalcev druge skupine in 8 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. 220. 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: 3,11 v prvi skupini (v prejšnjih letih 3,31, 3,41, 3,40, 3,28, 3,39), 3,51 v drugi skupini (prejšnja leta 3,65, 3,33, 3,44, 3,35, 3,50) in 3,73 v tretji skupini (prejšnja leta 3,43, 3,61, 3,19, 3,40, 3,21).

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 najmočnejša doslej ($R^2 = 0{,}70$; v prejšnjih letih 0,39, 0,56, 0,14, 0,52, 0,21, 0,11, pred tem okoli 0,4).

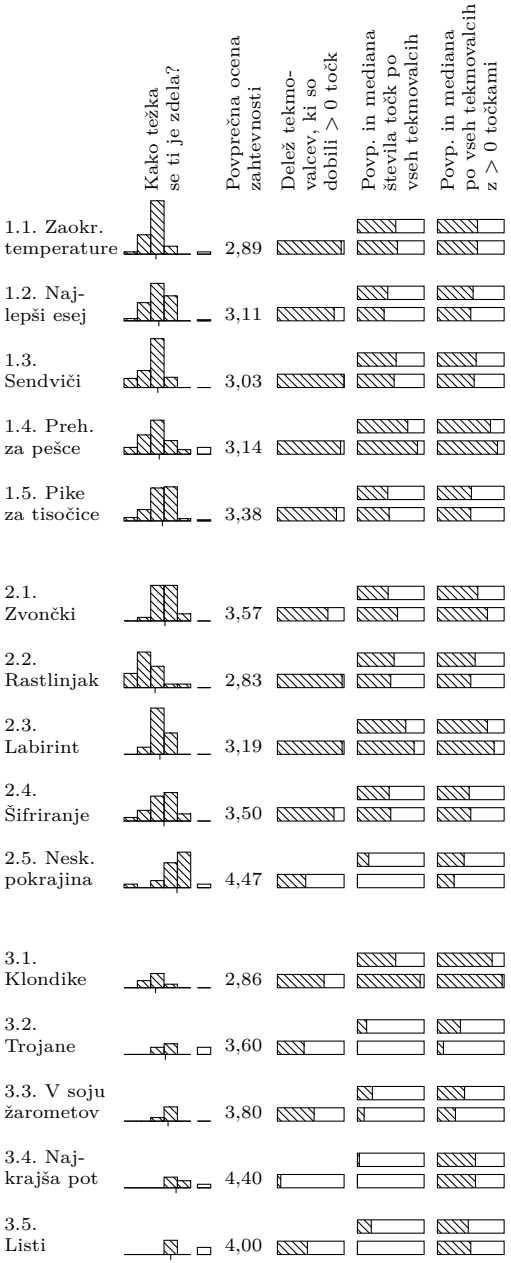
Daleč največ pripomb o tem, kako da je naloga (pre)težka, je bilo pri nalogi 2.5 (neskončna pokrajina); res je, da je ta naloga malo neobičajna, drugačna od ostalih in da je mnogi tekmovalci tudi niso dobro razumeli (pri njej je bilo tudi veliko pripomb, da je besedilo težko razumljivo). V prvi skupini se je zdela tekmovalcem težka predvsem naloga 1.5 (pike za tisočice), v tretji pa 3.4 (najkrajša pot).

Kot najlažjo so tekmovalci v prvi skupini ocenili nalogo 1.1 (zaokrožanje temperature), v drugi nalogo 2.2 (rastlinjak), v tretji skupini pa 3.1 (Klondike). Največ pripomb, da sta prelahki, pa je bilo pri nalogah 1.3 (pomanjkanje sendvičev) in 1.4 (prehod za pešce).

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 221. Nad razumljivostjo besedil ni veliko pripomb, v prvi skupini še malo manj kot prejšnja leta, v drugi in tretji pa malo več. Kot težje razumljiva izstopa še posebej naloga 2.5 (neskončna pokrajina), ki smo jo že omenjali zgoraj; poleg nje so kot težje razumljive ocenili še naloge 1.4 (prehod za pešce — to je naloga realnočasovnega tipa in take se tekmovalcem pogosto zdijo težje razumljive), 2.1 (zvončki) in 3.3 (v soju žarometov).

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, približno enako kot v prejšnjih letih. Edina naloga, pri kateri je bilo veliko pripomb, da je prekratka, je bila 1.4 (prehod za pešce). Predolgi pa sta se še največ ljudem zdeli nalogi 1.1 (zaokrožanje temperature) in 2.1 (zvončki). V tretji skupini so se nekaterim zdela besedila prekratka, nikomur pa predolga.

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta, v drugi in tretji skupini še malo višje. Je pa bilo letos več kot



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

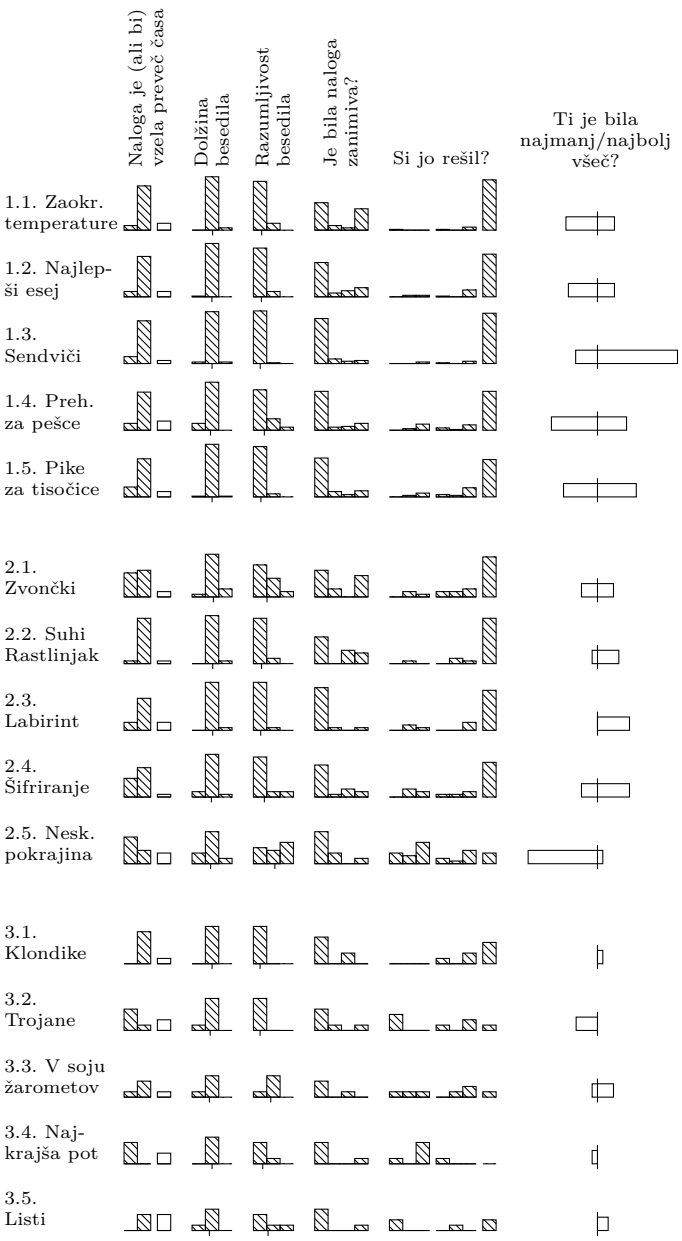
Pomen stolpcev v vsaki vrstici:

Na levi je skupina šestih stolpcev, ki kažejo, kako so tekmovalci v anketi odgovarjali na vprašanje o zahtevnosti naloge. Stolpci po vrsti pomenijo odgovore „prelahka“, „lahka“, „primerna“, „težka“, „pretežka“ in „ne vem“. Višina stolpca pove, koliko tekmovalcev je izrazilo takšno mnenje o zahtevnosti naloge. Desno od teh stolpcev je povprečna ocena zahtevnosti (1 = prelahka, 3 = primerna, 5 = pretežka). Povprečno oceno kaže tudi črtica pod to skupino stolpcev.

Sledi stolpec, ki pokaže, kolikšen delež tekmovalcev je pri tej nalogi dobil več kot 0 točk. Naslednji par stolpcev pokaže povprečje (zgornji stolpec) in mediano (spodnji stolpec) števila točk pri vsej nalogi. Zadnji par stolpcev pa kaže povprečje in mediano števila točk, gledano le pri tistih tekmovalcih, ki so dobili pri tisti nalogi več kot nič točk.

Mnenje tekmovalcev o nalogah

Višina stolpcev pove, koliko tekmovalcev je dalo določen odgovor na neko vprašanje.



Stolpci se od leve proti desni nanašajo na naslednja vprašanja in možne odgovore:

Naloga je (ali: bi) vzela preveč časa:

- da
- ne
- ne vem

Dolžina besedila:

- prekratko
- primerno
- predolgo

Razumljivost besedila:

- razumljivo
- težko razumljivo
- nerazumljivo

Naloga je bila:

- zanimiva
- dolgočasna
- že znana
- povprečna

Si jo rešil?

- nisem, zmanjkalo časa
- nisem, zmanjkalo volje
- nisem, zmanjkalo znanja
- delno, zmanjkalo časa
- delno, zmanjkalo volje
- delno, zmanjkalo znanja
- rešil sem celo

Katera naloga ti je bila najmanj všeč? Katera ti je bila najbolj všeč?

ponavadi pripomb, češ da je neka naloga že znana, največ pri nalogah 1.2 (najlepši esej), 1.4 (prehod za pešce) in 2.2 (rastlinjak). vendar je letos malo več pripomb, češ da je neka naloga že znana. Kot bolj zanimivi izstopata nalogi 1.3 (sendviči) in 2.3 (labirint).

Pripomb, da bi naloga vzela preveč časa, je bilo malo, podobno kot prejšnja leta, v prvi skupini še manj kot ponavadi. Največ takih pripomb je bilo pri nalogah 2.1 (zvončki), 2.5 (neskončna pokrajina), 3.2 (Trojane) in 3.4 (najkrajša pot). To je najbrž povezano s tem, da so se jim zdele te naloge tudi težke ali pa njihovo besedilo (pre)dolgo.

Pri glasovih o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, je bila v prvi skupini najbolj priljubljena naloga 1.3 (pomanjkanje sendvičev), glasovi za najmanj všeč pa so precej razpršeni med ostale štiri naloge; še največ jih je dobila naloga 1.4 (prehod za pešce). V drugi skupini sta bili najbolj priljubljeni 2.3 (labirint) in 2.4 (šifriranje), kot izrazito nepriljubljena pa izstopa 2.5 (neskončna pokrajina). V tretji skupini jim je bila najbolj všeč naloga 3.3 (v soju žarometov), najmanj pa 3.2 (Trojane).

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.

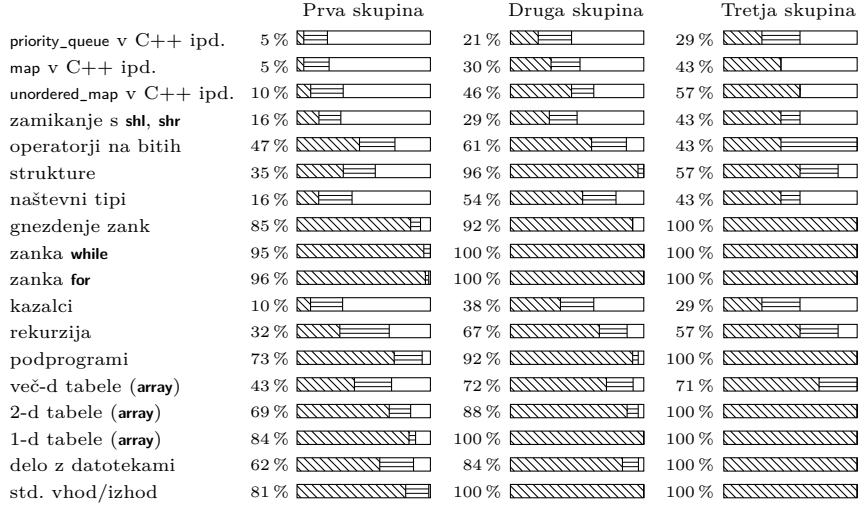


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

Rezultati pri vprašanjih o programerskem znanju so podobni tistim iz prejšnjih let. V drugi skupini (pravijo, da) znajo malo več kot lani, v tretji pa malo manj, vendar

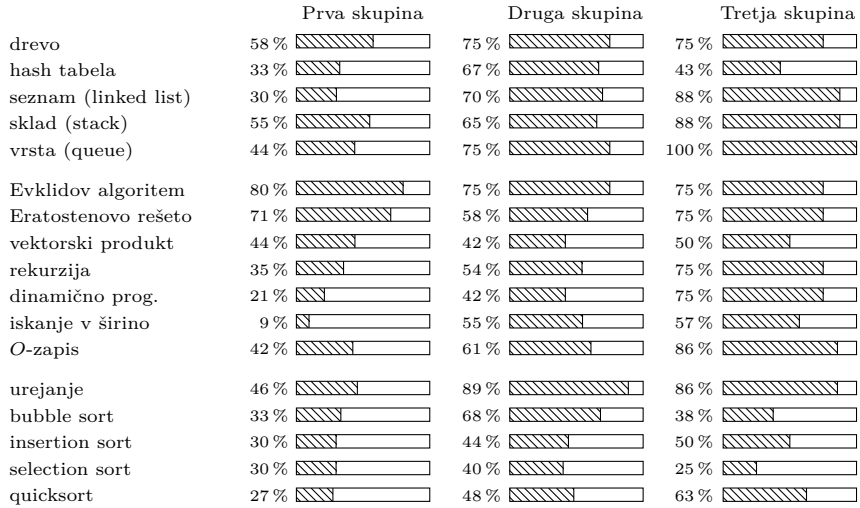


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

je zaradi majhnega števila anket v tretji skupini vprašljivo, koliko se lahko zanesemo na te podatke. Stvari, ki jih tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja leta: rekurzija, kazalci, naštevni tipi in operatorji na bitih, v prvi in drugi skupini tudi strukture.

Uporaba programskih jezikov

V prvi skupini je letos z občutno prednostjo najpogostejši jezik python, sledita mu java in C++; C# je letos redkejši kot prejšnja leta. V drugi skupini sta najpogostejša python in C++ (približno izenačena), ostali so redkejši. V tretji skupini je C++ daleč najpogostejši, na drugem mestu pa sta C in java. Na splošno je letos C-ja malo več kot zadnjih nekaj let, C#-a pa malo manj. Pascal je uporabljala le peščica tekmovalcev v prvi skupini, basica pa sploh nihče. Novost pa je letos jezik julia, ki ga je uporabljal eden od tekmovalcev v prvi skupini.

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 lani smo tudi letos pri nalogah tipa „opiši postopek“ pripisali „ali napiši podprogram (kar ti je lažje)“.

Podobno kot v prejšnjih letih je v anketi še kar nekaj tekmovalcev napisalo, da dobro poznajo tudi PHP in/ali javascript, vendar ju na tekmovanju letos ni uporabljal nihče.

	Leto in skupina																	
	2017			2016			2015			2014			2013			2012		
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	6	1	4
C	4	3	$2\frac{1}{2}$	$4\frac{1}{3}$	1	2	3		1	$3\frac{1}{2}$	6		2	7		7	2	1
C++	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	26	16	9
java	28	3	2	24	6	5	22	6	$3\frac{1}{2}$	23	2	$1\frac{1}{2}$	12	8	1	17	$6\frac{1}{2}$	1
PHP			–			–	3		–	2		–	1	$\frac{1}{2}$	–		1	–
basic									1		1	–	1		–			–
C#	7	6		12	5	1	16	5		12	$1\frac{1}{2}$	2	18	$\frac{1}{2}$		17	1	3
python	42	11	–	$29\frac{1}{3}$	12	–	26	1	–	16	6	–	16	8	–	25	5	–
NewtonScript			–			–			–			–		$\frac{1}{2}$	–		$\frac{1}{2}$	–
javascript			–	1		–	1		–	1		–			–			–
julia	1		–			–			–			–			–			–
batch			–			–			–	1		–			–			–
psevdokoda	5		–	5		–	6	1	–	10		–	6		–	3		–
nič				2	3		4	1		4	2		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 2017 sta dva tekmovalca uporabljala python in javo, eden pa C in C++.) „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; vse več tekmovalcev na primer uporablja `string` namesto **`char *`** in tip `vector` namesto tradicionalnih tabel (*arrays*). Tako kot lani smo tudi letos pri enem tekmovalcu opazili rešitve v C++11/14 (med drugim je uporabljal **`auto`** v njegovem novem pomenu).

Pri pythonu zdaj že skoraj 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 (72/78), v drugi pa še višji (25/25). Nenavadno je, da so pri vprašanju, ali bi želeli deklaracije še v kakšnem jeziku, nekateri tekmovalci navedli jezike, v katerih deklaracije že imamo, na primer javo, C++ in C#; od originalnih predlogov je bila najpogostejša lua. Tako se človek vpraša, koliko se smemo na odgovore pri teh vprašanjih sploh zanašati. V vsakem primeru pa se poskušamo zadnja leta v besedilih nalog izogibati deklaracijam v konkretnih programskih jezikih in jih zapisati bolj na splošno, na primer „napiši funkcijo `foo(x, y)`“ namesto „napiši funkcijo **`bool`** `foo(`**`int`** `x,` **`int`** `y)`“. Eden od tekmovalcev je podal zanimiv predlog, da bi v deklaracije v pythonu dodali anotacije tipov, kot jih ima python od verzije 3.5 naprej (PEP 484).

V rešitvah nalog smo zadnja leta objavljali izvorno kodo le v C-ju, letos pa

prehajamo na C++, pri čemer se bomo trudili večinoma uporabljati stvari, ki so razumljive v obeh jezikih. 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 (39/77 v prvi skupini, 20/25 v drugi, 5/60 v tretji), vendar so ti deleži letos malo nižji kot lani in v prvi skupini je že skoraj polovica takih, ki pravijo, da izvorne kode v rešitvah ne razumejo. Zanimivo vprašanje je, ali bi s kakšnim drugim jezikom dosegli večji delež tekmovalcev (koliko tekmovalcev ne bi razumelo rešitev v javi? ali v pythonu?). Med jeziki, ki bi jih radi videli namesto (ali poleg) C++, jih največ omenja python in javo (tadva jezika sta približno izenačena, python je v rahli prednosti). Zaradi teh rezultatov smo v letošnjem biltenu poskusno objavili rešitve nalog za prvo skupino tudi v pythonu in ne le v C++.

Letnik

Po pričakovanjih so tekmovalci zahtevnejših skupin večinoma v višjih letnikih kot tisti iz lažjih skupin. Razmerja so podobna kot prejšnja leta, v povprečju pa so se tekmovalci letos rahlo pomladili. Letos je nastopilo tudi nekaj osnovnošolcev, in sicer trije v prvi in dva v tretji skupini.

	Št. tekmovalcev po letnikih						Povprečni
Skupina	7	9	1	2	3	4	letnik
prva	1	2	18	32	35	26	2,5
druga			4	5	14	10	2,9
tretja		2	2	2	7	7	2,8

Druga vprašanja

Podobno kot prejšnja leta je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!). V smislu širitve zanimanja za tekmovanje in večanja števila tekmovalcev se zelo dobro obnese šolsko tekmovanje, ki ga izvajamo zadnjih nekaj let, saj se odtlej v tekmovanje vključuje tudi nekaj šol, ki prej na našem državnem tekmovanju niso sodelovale. Nekaj ljudi je za naše tekmovanje slišalo na računalniškem tekmovanju Bober, ki ga tudi organizira društvo ACM Slovenija.

Pri vprašanju, kje so se naučili programirati, sta podobno kot prejšnja leta najpogostejši odgovor, da so se naučili programirati sami ali pa v šoli (letos so samouki v rahli večini); 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. Nekatere tekmovalce moti, da je treba tako dolgo čakati na razglasitev rezultatov, vendar komisija hitreje žal ne more oceniti odgovorov vseh tekmovalcev.

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

	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	75	1	6	2	48	40	20	7	6	3	8	61	6	4	61	29	22	25	32	27	21	32	35
II	17	3	3	2	22	10	8	3	4	6	1	13	2	5	13	6	7	8	7	8	4	8	8
III	6	1	0	1	6	3	1	4	2	1	1	4	0	2	4	2	2	2	2	5	5	3	3

večina jih je res oddajala odgovore na računalniku, nekaj pa jih je vseeno reševalo na papir. Pri oddajanju odgovorov na računalniku se stanje izboljšuje, letos skoraj ni bilo težav s shranjevanjem, je pa še vedno nekaj pripomb glede okornosti sistema.

Podobno kot prejšnja leta si je veliko tekmovalcev tudi želelo, da bi imeli v prvi in drugi skupini na računalnikih prevajalnike in podobna razvojna orodja. Razlog, zakaj se v teh dveh skupinah izogibamo prevajalnikom, je predvsem ta, da hočemo s tem obdržati poudarek tekmovanja na snovanju algoritmov, ne pa toliko na lovljenju drobnih napak; in radi bi tekmovalce tudi spodbudili k temu, da se lotijo vseh 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) Lepo je videti, da so tekmovalci pozorni na razliko med dovršnimi in nedovršnimi glagoli:

```
while True: # V navodilih piše, da bo program štev
ila PREBIRAL in ne prebral,
            # kar pomeni, da bo to počel v neskončni zanki.
```

Komisija si sicer pri takšnih navodilih o branju vhoda načeloma predstavlja, naj program bere do konca vhodnih podatkov (EOF), vendar je za namen te naloge tudi neskončna zanka čisto dobra.

(1.1) Komentar na koncu ene od rešitev:

```
// to mora biti hudo narobe
// vredno je bilo poskusiti
```

(1.1) Komentar na koncu neke precej zgrešene rešitve (vse temperature, ki niso z intervala $[-1/2, 0)$, poskuša zaokrožiti kar s funkcijo round):

```
# če pa ne deluje pravilno:
  return("wow kok je python glup")
```

(1.1) Nagrado za najbolj ekstremno izogibanje smislu naloge dobi naslednja 207 vrstic dolga rešitev, ki temelji na predpostavki, da je razpon možnih temperatur omejen, zato lahko pokrije vse možnosti s kupom stavkov **if**:

```
# Predpostavimo, da temperatura ni bila višja od 60,5° C in nižja od −40,5° C
if temperatura >= −0,5 and temperatura < 0:
  zaokrozeno == −0
elif temperatura >= 0 and temperatura < 0,5:
  zaokrozeno == 0
elif temperatura >= 0,5 and temperatura < 1,5:
  zaokrozeno == 1
```

in tako naprej do 60 stopinj, potem pa še za negativne temperature do -40 stopinj.

(1.1) Pri tej nalogi je več ljudi računalo ne-celi del števila x z izrazom oblike x % 1. Pri tem pa so nekateri pozabili, da C in C++ ne dovolita uporabe operatorja % na ne-celoštevilskih operandih (za razliko od jave in pythona). Še en zanimiv način, kako z % priti do ne-celega dela števila x, pa je naslednja rešitev:

```
x_odrezan = Odrezi(x)
⋮
if x >= 0:
  if x % x_odrezan >= 0.5: resitev = x_odrezan + 1
```

Tudi naslednji tekmovalec je poskušal z %, vendar se je zmotil v delitelju:

```
if (x % 0.1 >= 0.5): # preveri, ali moramo zaokrožiti navzgor
```

(1.1) Zakaj bi rekli „**if** (y % 10 >=5) . . . “, če lahko uporabimo stavek **switch**:

```
switch (y % 10) {
  case 5:
  case 6:
  case 7:
  case 8:
  case 9:
    ⋮
    break;
}
```

(1.1) Posrečena besedna igra:

```
vhod = input() # v obliki „predznak celi del . dve decimalki“;
               # npr. „3.14“ (Notice me SenPi)
```

(1.1) Zanimiva inovacija v pascalu: sintakso z dvopičji, ki se jo pri Write in WriteLn uporablja za izpis na določeno število decimalk, je tale tekmovalec uporabil kar kot splošnonamenski operator za zaokrožanje.

```
(* odreže vse decimalke, razen zadnje *)
zaokrozeno := (kaj:1:0);
```

(1.1) Še ena zanimiva sintaktična inovacija:

```
if (cd[1].charAt(0).equals(5 || 4 || 3 || 2 || 1)) { // če se prva številka decimalke ujema
                                                     // ali z 5, 4, 3, 2, 1
```

(1.1) Rešitev z izgovarjanjem na omejitve programskega jezika:

```
# 0 ne more biti −0, ker v Pythonu tip int tega ne dovoljuje.
```

Očitno ni opazil, da naloga govori o tem, kaj moramo *izpisati*, ne o tem, kaj moramo hraniti v spremenljivki tipa **int**. . .

(1.1) Za ljubitelje nepotrebnega kompliciranja:

```
if (!(a > 0) && !(a < 0)) { // preverimo, če je število 0
```

In v neki drugi rešitvi:

```
temp = temp − 0.50 + 1;
```

(1.1) Sesutje programa kot način za izhod iz zanke:

```
while True:
  stevilo = sys.stdin.read(i)
  ⋮
# če bo na seznamu zmanjkalo števil, bo program crashov in loop se bo končal!!
```

(1.1) Besedilo naloge pravi, da je dodatno pravilo o zaokrožanju na −0 v navadi „v deželah, kjer merimo temperaturo v stopinjah Celzija“. Eden od tekmovalcev je zato napisal rešitev, ki uporabnika najprej vpraša, ali uporablja stopinje Celzija ali Fahrenheita, in se na podlagi tega odloči, ali je treba zaokrožati na −0.

(1.2) Verjetno najdaljši pogoj v stavku **if** letos:

```
// Grozljivo dolg if stavek pretvori pridobljeno vrstico v niz, ki vsebuje le črke in presledke
if (delEseja.charAt(i) == 'a' ||
    delEseja.charAt(i) == 'b' ||
    ⋮
    delEseja.charAt(i) == 'Z' ||
    delEseja.charAt(i) == ' ' ||) {
```

(1.2) Za ljubitelje dolgih imen spremenljivk:

```
zadnji_polozaj_znaka_ki_ga_ni_v_angleski_abecedi = −1
for odvecno_dolg_stevec in range(len(vrstica_dolga_do_100_znakov)):
```

(1.2) Na začetku rešitve, napisane v psevdokodi:

```
⟨!DOCTYPE = psevdokoda⟩
```

(1.2) Nepotrebno kompliciranje v pogoju in posrečen komentar o veznikih:

```
if (len(besede) − dalse − krajse) == len(besede): # Če bo dolzina seznama besede,
    # potem ko odštejemo število krajših in daljših besed, enaka dolžini seznama
    # besede, potem esej nima nobene besede krajše od ali daljše od predpisane dolžine
  print('Esej je lep! Čestitam. :)') # Lahko čestitamo človeku, ki ni
                                     # uporabljal veznikov.
```

(1.2) Komentar v eni od rešitev:

```
// by the way, super kriteriji za lepoto testa, take rabimo pisat pri Slovenščini
```

Mogoče bi morali dodati še kak kriterij o velikih začetnicah `:)`

(1.3) Pri tej nalogi je več tekmovalcev imelo razne dele programa razmnožene v šestih kopijah, po eno za vsak tip sendviča. Rekorderji so s tem prišli do programov, dolgih dobrih 150 vrstic. To bi človek še razumel, če tekmovalec ne pozna tabel, toda pri nekaterih izmed teh rešitev je očitno, da tabele znajo uporabljati. Najboljši primer je tale:

```
switch (x) {
case 1:
  if (tab[x − 1] > 0) {
    System.out.println("Izvolite sendvic st. " + x + ". Dober tek!");
  ⋮
case 2:
  if (tab[x − 1] > 0) {
    System.out.println("Izvolite sendvic st. " + x + ". Dober tek!");
```

in tako naprej za vseh šest tipov sendvičev. V vseh šestih primerih je koda enaka, česar se je zavedal tudi reševalec, saj je na koncu rešitve napisal razlago:

> Ob vsakem vnosu gremo v **switch**, kjer ob vsaki vneseni številki (1–6) izvedemo isti pogoj.

(1.3) Še ena zanimiva sintaktična inovacija: kako preveriti, če je ena spremenljivka večja od nekaj drugih:

```
if Q > W and E and R and T and Z:
  popularni = 1
```

(1.3) Dekadentno: naslednja rešitev namesto navadne tabele uporablja razpršeno tabelo, v kateri so ključi kar nizi, ki predstavljajo števila od 1 do 6.

```
priljubljenost = {"1":0, "2":0, "3":0, "4":0, "5":0, "6":0}
```

(1.3) Tale je imel čisto lepo tabelo (oz. list v pythonu), vendar jo je uporabljal na zelo neroden način:

```
if x == 1: najpop[0] += 1
if x == 2: najpop[1] += 1
⋮
if x == 6: najpop[5] += 1
```

(1.3) Verjetno najboljša tipkarska napaka letos:

```
pop = order(pop) # fukcija ki rzvrsti elemente po velikosti mi je ušla iz spomina
```

(1.3) Ko ti pythonov sort ni dovolj dober in uporabiš izmišljeno implementacijo bogosorta...

```
from sort import bogosort
⋮
bogosort(prl1, prl2, prl3, prl4, prl5, prl6) # še sanja se mi ne kaj mi sort vn vrže,
                                             # predpostavljam da list
```

(1.3) V navodilih smo poudarili, naj tekmovalec v nekaj stavkih opiše idejo, na kateri temelji njegova rešitev. Eden od tekmovalcev je napisal:

> Ideja sestavljalca je, da tekmovalec postane lačen. Moja ideja pa je že sproti natančno popisana. Torej zgolj sledenje navodilom in pisanje pogojev. Hvala Bogu, da obstaja „ctrl + c“ in „ctrl + v“.

Isti tekmovalec pri četrti nalogi:

> Osnovna ideja je, da nimam pojma, kaj hoče naloga od mene. Zato sem si jo interpretiral po svoje.

(1.3) Po tistem, ko prebere podatke o zalogi sendvičev, in preden začne brati naročila študentov:

```
Console.Clear(); // Študentom ne razkrijemo strogo zaupnih podatkov o številu sendvičev
```

Kasneje v isti rešitvi:

```
Console.WriteLine("Nom Nom Nom, dober tek. Uzivajte v sendviču tipa {0}",
                                                          zahtevanSendvic);
```

(1.3) Zakaj bi začeli zanko pri 1, če jo lahko pri 0 in nato prvo iteracijo preskočimo:

```
for (int i = 0; i < 6; i++) {
  if (i > 0) {
    ⋮
  }
}
```

(1.4) Rešitev, ki namesto v zanki obdeluje dogodke z rekurzijo:

```
public static void delovanje() {
  int stanje, st = 0;
  stanje = Dogodek();
  ⋮
  delovanje();
}
```

Podobno je naredil ta tekmovalec tudi pri tretji nalogi.

(1.4) Ta naloga je v marsikaterem tekmovalcu prebudila represivno žilico. Najdemo vse od bolj blagih primerov:

```
if stanje == "rdeča":
  prestopniki += 1
```

. . . prek malo hujših:

```
if rdeča:
  kriminalci += 1
```

in

```
if rdeca == True and pesci > 0: # ko pešec prečka cestišče pri rdeči luči
  ilegalci += pesci # prištejemo pešce k ilegalcem
  pesci = 0 # ponastavimo pešce na 0, saj smo jih že šteli k ilegalcem
```

. . . do ekstremnih:

```
if not gori_rdeca_luc: # luč ni rdeča, torej je zelena
  stevilo_vzornih_drzavljanov += 1 # za evidenco
else: # luč je rdeča
  koliko_jih_gre_v_gulag += 1 # tud za evidenco
  nedavni_prekrskarji += 1
```

Nekateri pa imajo raje bolj verski pogled:

```
if (jeRdeca == true) // Preveri, če gori rdeča luč
  stGrehov++; // Poveča stevec v primeru, da pogoj drži
```

Ali pa bolj vzgojiteljskega:

```
if (a == 3 && rdeca)
  porednipesci++;
```

(1.4) Hvalevredna skrb za jezik pri izpisu:

```
int Izpis() { // z obzirom na kulturo slovenskega jezika in kulturo dvojine.
  if (pescev == 0) {
    printf("Pri prejsnji rdeci luci je na prehod stopilo 0 ljudi.");
  } if (pescev == 1) {
    printf("Pri prejsnji rdeci luci je na prehod stopil 1 človek.");
  } if (pescev == 2) {
    printf("Pri prejsnji rdeci luci sta na prehod stopila 2 človeka.");
  } if (pescev == 3 || pescev == 4) {
    printf("Pri prejsnji rdeci luci so na prehod stopili %d ljudje.", pescev);
  } if (pescev > 4) {
    printf("Pri prejsnji rdeci luci je na prehod stopilo %d ljudi.", pescev);
  }
}
```

Pri veliki gneči sicer ta rešitev odpove — bolje bi bilo, če bi v pogojih namesto števila pešcev gledal njegov ostanek po deljenju s 100.

(1.4) Ta rešitev se res potrudi, da bi bilo zanke konec, ko je dogodek == 1:

```
while (dogodek != 1)
{
  ⋮
  else if (dogodek == 1)
    break;
}
```

(1.5) Navdušenje ob izpisu rezultata:

```
cout << "Stevilo s fancy zapisom je: " << beseda << endl; // slavnostni izpis števila
```

(1.5) Nenavadno zapleten način preverjanja, če je med prvimi $i+1$ znaki niza vejica:

```
if not ("," in str(s)[i::-1]):
```

Zakaj neki je niz obračal, namesto da bi uporabil preprosto str(s)[:i + 1]?

(1.5) Zanimiva inovacija pri vrivanju pik v niz k:

```
while l > 0:
  if l % 3 == 0:
    k[-l] = k[-l] + "."
  l = l - 1
```

(1.5) Mešani signali na koncu neke funkcije (ki vrača **void**):

```
return; // ni potrebno napisati „return“
```

(2.1) Modre misli v komentarjih:

```
# Čas je kot denar. Ni ga.
```

(2.3) Dekadentno: zakaj bi sami preverjali, ali bi Miha pri premiku v določeno smer padel iz mreže, ko pa lahko to delo prevalimo na prevajalnik in samo ujamemo izjemo, če bo prišlo do nje...

```
else if (premik == 'Z')
{
  y--;
  try { if (labirint[x][y] == 1) return false; }
  catch (ArrayIndexOutOfBoundsException e) { return false; }
}
```

(2.3) Komentar v eni od rešitev:

```
print(coord[0] + 1, coord[1] + 1) # Ko sem bil majhen, sem mislil,
                                  # da print() natisne besedilo na papir
```

(2.3) Rešitev, ki poskrbi, da bo datoteka res temeljito zaprta. Najprej pokliče s.close(), nato pa jo bo še enkrat poskusil zapreti context manager iz stavka **with**.

```
with open("vhodni_podatki.txt") as s: # preberemo vhodno datoteko in vsako vrstico
                                      # dodamo v podatke
   for line in s:
      podatki.append(line)
   s.close() # zapremo file
```

To sicer čisto lepo deluje, saj v pythonu ni napaka, če poskušamo že zaprto datoteko zapreti še enkrat.

(2.4) Tale tekmovalec očitno ne uporablja googla, pa je zato tudi glagol izpeljal iz imena drugega iskalnika:

```
# Ker ne vem na pamet, kako dobim seznam (list) vseh malih in velikih črk v obliki,
# kot si jo želim, bom vse pretvoril v male črke, dešifriral ter nato po potrebi v velike
# črke. To bi sicer poračkal (duckduckgo.com).
```

(2.4) Iz komentarja na začetku ene od rešitev:

> Prišel vas je rešit agent Janez, Poceni Janez.

(2.4) Rešitev, ki predvideva stvari v nasprotju z navodili naloge:

```
string standardenPodpis = "janez"; // predvideno, da ni velikih črk
```

(2.5) Nekateri so preiskovali točke (x, y) kar naključno. Naslednji tekmovalec je poskusil to idejo še izboljšati in je v tabeli označeval, katere točke je že pregledal. S tem se res prihrani nekaj klicev funkcije Visina, po drugi strani pa bi ta tabela zasedla približno 2^{64} bytov pomnilnika (če je tip **int** 32-biten):

```
int triedNumbs[INT_MAX][INT_MAX];
⋮
while (Visina(currX, currY) != ciljnaVisina)
{
   triedNumbs[currX][currY] = true;
   while (triedNumbs[currX][currX])
   {
      currX = rand() * INT_MAX; currY = rand() * INT_MAX;
   }
}
```

(2.5) Naslednja rešitev najprej sistematično pregleda celotno ravnino, nato pa, če iskane višine ne najde, za vsak primer še enkrat pregleda eno četrtino ravnine (in potem še četrtino tiste četrtine in tako naprej v zanki):

```
n = (2^32 − 1) / 2; m = (2^32 − 1) / 2
while True:
   for i in range(n):
      for j in range(m):
         if Visina(i, j): return (i, j)
   n = n // 2; m = m // 2
# S tem se sicer razišče samo polovica možnosti, ker se m in n samo zmanjšujeta
# in ne zvečujeta. Napisati bi moral še, da se n in m vsakič še povečata za polovico,
# vendar se trenutno ne spomnim, kako bi to napisal.
```

(2.5) Za ljubitelje Čukov:

```
# Vsem je nam le čas vladar
```

(3.1) Pri tej nalogi iščemo najkrajše poti v grafu, pri čemer so vse povezave enako dolge; zato je dovolj že iskanje v širino in pri njem navadna vrsta, ni treba prioritetne vrste (npr. kot za Dijkstrov algoritem). Tale tekmovalec pa jo je vseeno uporabil:

```
priority_queue<pair<long long, pair<long long, long long> > > q;
```

(3.2) Rešitev za ljubitelje stavka **else**:

```
// proti zapostavljanju stavka else.
if (pos == v[i].b) {
} else {
   t = time_at_pos[v[i].a] + u * (v[i].b − pos);
   pos = v[i].b;
}
```

(3.3) Čudovit primer nepotrebnega kompliciranja pri podatkovnih strukurah. Eden od tekmovalcev si je najprej pripravil seznam parov $(x, \triangle c)$, ki povedo, pri katerem x se spremeni intenziteta svetlobe in za koliko (vsak žaromet prispeva dva taka para). Nato je vse pare uredil po x. Do tod je torej rešitev zelo podobna naši. Zdaj bi bil primeren trenutek, da izračunamo v tem zaporedju kumulativne vsote sprememb $\triangle c$, tako da bomo pri vsakem x vedeli, kakšna je zdaj tam intenziteta svetlobe. Toda naš tekmovalec je namesto tega zgradil Fenwickovo drevo in nato pri vsakem nastopajočem izračunal potrebno kumulativno vsoto s pomočjo tega drevesa! Ta pristop bi bil zelo koristen, če bi se poizvedbe prepletale s kakšnimi spremembami v zaporedju parov $(x, \triangle c)$, npr. ker bi se lahko žarometi prižigali in ugašali, ampak pri naši nalogi se to ne dogaja.

(3.3) To se pa zgodi, če človek popravlja program s search and replace (na srečo je bilo v zakomentiranem delu programa):

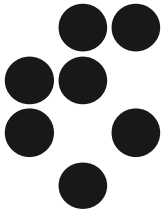
```
prlong longf("%d\n", p); */
```

Za konec pa še ena cvetka iz anket (in sicer iz prve skupine):

> (*Vprašanje v anketi.*) Katera naloga ti je bila najbolj všeč in zakaj?
> (*Odgovor.*) 3. naloga. Rad imam sendviče in neskončne zanke. Predvsem sendviče.

SODELUJOČE INŠTITUCIJE

Institut Jožef Stefan



Institut je največji javni raziskovalni zavod v Sloveniji s skoraj 800 zaposlenimi, od katerih ima približno polovica doktorat znanosti. Več kot 150 naših doktorjev je habilitiranih na slovenskih univerzah in sodeluje v visokošolskem izobraževalnem procesu. V zadnjih desetih letih je na Institutu opravilo svoja magistrska in doktorska dela več kot 550 raziskovalcev. Institut sodeluje tudi s srednjimi šolami, za katere organizira delovno prakso in jih vključuje v aktivno raziskovalno delo. Glavna raziskovalna področja Instituta so fizika, kemija, molekularna biologija in biotehnologija, informacijske tehnologije, reaktorstvo in energetika ter okolje.

Poslanstvo Instituta je v ustvarjanju, širjenju in prenosu znanja na področju naravoslovnih in tehniških znanosti za blagostanje slovenske družbe in človeštva nasploh. Institut zagotavlja vrhunsko izobrazbo kadrom ter raziskave in razvoj tehnologij na najvišji mednarodni ravni.

Institut namenja veliko pozornost mednarodnemu sodelovanju. Sodeluje z mnogimi uglednimi institucijami po svetu, organizira mednarodne konference, sodeluje na mednarodnih razstavah. Poleg tega pa po najboljših močeh skrbi za mednarodno izmenjavo strokovnjakov. Mnogi raziskovalni dosežki so bili deležni mednarodnih priznanj, veliko sodelavcev IJS pa je mednarodno priznanih znanstvenikov.

Tekmovanje sta podprla naslednja odseka IJS:

CT3 — Center za prenos znanja na področju informacijskih tehnologij

Center za prenos znanja na področju informacijskih tehnologij izvaja izobraževalne, promocijske in infrastrukturne dejavnosti, ki povezujejo raziskovalce in uporabnike njihovih rezultatov. Z uspešnim vključevanjem v evropske raziskovalne projekte se Center širi tudi na raziskovalne in razvojne aktivnosti, predvsem s področja upravljanja z znanjem v tradicionalnih, mrežnih ter virtualnih organizacijah. Center je partner v več EU projektih.

Center razvija in pripravlja skrbno načrtovane izobraževalne dogodke kot so seminarji, delavnice, konference in poletne šole za strokovnjake s področij inteligentne analize podatkov, rudarjenja s podatki, upravljanja z znanjem, mrežnih organizacij, ekologije, medicine, avtomatizacije proizvodnje, poslovnega odločanja in še kaj. Vsi dogodki so namenjeni prenosu osnovnih, dodatnih in vrhunskih specialističnih znanj v podjetja ter raziskovalne in izobraževalne organizacije. V ta namen smo postavili vrsto izobraževalnih portalov, ki ponujajo že za več kot 500 ur posnetih izobraževalnih seminarjev z različnih področij.

Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi virtualno učečo se skupnost in pripomoči k učinkovitejšemu povezovanju znanosti in industrije ter večji prepoznavnosti domačega znanja v slovenskem, evropskem in širšem okolju.

E3 — Laboratorij za umetno inteligenco
Področje dela Laboratorija za umetno inteligenco so informacijske tehnologije s poudarkom na tehnologijah umetne inteligence. Najpomembnejša področja raziskav in razvoja so: (*a*) analiza podatkov s poudarkom na tekstovnih, spletnih, večpredstavnih in dinamičnih podatkih, (*b*) tehnike za analizo velikih količin podatkov v realnem času, (*c*) vizualizacija kompleksnih podatkov, (*d*) semantične tehnologije, (*e*) jezikovne tehnologije.

Laboratorij za umetno inteligenco posveča posebno pozornost promociji znanosti, posebej med mladimi, kjer v sodelovanju s Centrom za prenos znanja na področju informacijskih tehnologij (CT3) razvija izobraževalni portal VideoLectures.NET in vrsto let organizira tekmovanja ACM v znanju računalništva.

Laboratorij tesno sodeluje s Stanford University, University College London, Mednarodno podiplomsko šolo Jožefa Stefana ter podjetji Quintelligence, Cycorp Europe, LifeNetLive, Modro Oko in Envigence.

*

Fakulteta za matematiko in fiziko
Fakulteta za matematiko in fiziko je članica Univerze v Ljubljani. Sestavljata jo Oddelek za matematiko in Oddelek za fiziko. Izvaja dodiplomske univerzitetne študijske programe matematike, računalništva in informatike ter fizike na različnih smereh od pedagoških do raziskovalnih.

Prav tako izvaja tudi podiplomski specialistični, magistrski in doktorski študij matematike, fizike, mehanike, meteorologije in jedrske tehnike.

Poleg rednega pedagoškega in raziskovalnega dela na fakulteti poteka še vrsta obštudijskih dejavnosti v sodelovanju z različnimi institucijami od Društva matematikov, fizikov in astronomov do Inštituta za matematiko, fiziko in mehaniko ter Instituta Jožef Stefan. Med njimi so tudi tekmovanja iz programiranja, kot sta Programerski izziv in Univerzitetni programerski maraton.

Fakulteta za računalništvo in informatiko
Glavna dejavnost Fakultete za računalništvo in informatiko Univerze v Ljubljani je vzgoja računalniških strokovnjakov različnih profilov. Oblike izobraževanja se razlikujejo med seboj po obsegu, zahtevnosti, načinu izvajanja in številu udeležencev. Poleg rednega izobraževanja skrbi fakulteta še za dopolnilno izobraževanje računalniških strokovnjakov, kot tudi strokovnjakov drugih strok, ki potrebujejo znanje informatike. Prav posebna in zelo osebna pa je vzgoja mladih raziskovalcev, ki se med podiplomskim študijem pod mentorstvom univerzitetnih profesorjev uvajajo v raziskovalno in znanstveno delo.

Fakulteta za elektrotehniko, računalništvo in informatiko

Fakulteta za elektrotehniko, računalništvo in informatiko (FERI) je znanstveno-izobraževalna institucija z izraženim regionalnim, nacionalnim in mednarodnim pomenom. Regionalnost se odraža v tesni povezanosti z industrijo v mestu Maribor in okolici, kjer se zaposluje pretežni del diplomantov dodiplomskih in podiplomskih študijskih programov. Nacionalnega pomena so predvsem inštituti kot sestavni deli FERI ter centri znanja, ki opravljajo prenos temeljnih in aplikativnih znanj v celoten prostor Republike Slovenije. Mednarodni pomen izkazuje fakulteta z vpetostjo v mednarodne raziskovalne tokove s številnimi mednarodnimi projekti, izmenjavo študentov in profesorjev, objavami v uglednih znanstvenih revijah, nastopih na mednarodnih konferencah in organizacijo le-teh.

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Fakulteta za matematiko, naravoslovje in informacijske tehnologije Univerze na Primorskem (UP FAMNIT) je prvo generacijo študentov vpisala v študijskem letu 2007/08, pod okriljem UP PEF pa so se že v študijskem letu 2006/07 izvajali podiplomski študijski programi Matematične znanosti in Računalništvo in informatika (magistrska in doktorska programa).

Z ustanovitvijo UP FAMNIT je v letu 2006 je Univerza na Primorskem pridobila svoje naravoslovno uravnoteženje. Sodobne tehnologije v naravoslovju predstavljajo na začetku tretjega tisočletja poseben izziv, saj morajo izpolniti interese hitrega razvoja družbe, kakor tudi skrb za kakovostno ohranjanje naravnega in družbenega ravnovesja. V tem matematična znanja, področje informacijske tehnologije in druga naravoslovna znanja predstavljajo ključ do odgovora pri vprašanjih modeliranja družbeno ekonomskih procesov, njihove logike in zakonitosti racionalnega razmišljanja.

ACM Slovenija

ACM je največje računalniško združenje na svetu s preko 80 000 člani. ACM organizira vplivna srečanja in konference, objavlja izvirne publikacije in vizije razvoja računalništva in informatike.

ACM Slovenija smo ustanovili leta 2001 kot slovensko podružnico ACM. Naš namen je vzdigniti slovensko računalništvo in informatiko korak naprej v bodočnost.

Društvo se ukvarja z:

- Sodelovanjem pri izdaji mednarodno priznane revije Informatica — za doktorande je še posebej zanimiva možnost objaviti 2 strani poročila iz doktorata.
- Urejanjem slovensko-angleškega slovarčka — slovarček je narejen po vzoru Wikipedije, torej lahko vsi vanj vpisujemo svoje predloge za nove termine, glavni uredniki pa pregledujejo korektnost vpisov.
- ACM predavanja sodelujejo s Solomonovimi seminarji.
- Sodelovanjem pri organizaciji študentskih in dijaških tekmovanj iz računalništva.

ACM Slovenija vsako leto oktobra izvede konferenco Informacijska družba in na njej skupščino ACM Slovenija, kjer volimo predstavnike.

IEEE Slovenija

Inštitut inženirjev elektrotehnike in elektronike, znan tudi pod angleško kratico IEEE (Institute of Electrical and Electronics Engineers) je svetovno združenje inženirjev omenjenih strok, ki promovira inženirstvo, ustvarjanje, razvoj, integracijo in pridobivanje znanja na področju elektronskih in informacijskih tehnologij ter znanosti.

REPUBLIKA SLOVENIJA
MINISTRSTVO ZA IZOBRAŽEVANJE, ZNANOST IN ŠPORT

Ministrstvo za izobraževanje, znanost in šport

Ministrstvo za izobraževanje, znanost in šport opravlja upravne in strokovne naloge na področjih predšolske vzgoje, osnovnošolskega izobraževanja, osnovnega glasbenega izobraževanja, nižjega in srednjega poklicnega ter srednjega strokovnega izobraževanja, srednjega splošnega izobraževanja, višjega strokovnega izobraževanja, izobraževanja otrok in mladostnikov s posebnimi potrebami, izobraževanja odraslih, visokošolskega izobraževanja, znanosti, ter športa.

SREBRNI POKROVITELJ

Quintelligence

Obstoječi informacijski sistemi podpirajo predvsem procesni in organizacijski nivo pretoka podatkov in informacij. Biti lastnik informacij in podatkov pa ne pomeni imeti in obvladati znanja in s tem zagotavljati konkurenčne prednosti. Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja. IKT (informacijsko-komunikacijska tehnologija) je postavila temelje za nemoten pretok in hranjenje podatkov in informacij. S primernimi metodami je potrebno na osnovi teh informacij izpeljati ustrezne analize in odločitve. Nivo upravljanja in delovanja se tako seli iz informacijske logistike na mnogo bolj kompleksen in predvsem nedeterminističen nivo razvoja in uporabe metodologij. Tako postajata razvoj in uporaba metod za podporo obvladovanja znanja (knowledge management, KM) vedno pomembnejši segment razvoja.

Podjetje Quintelligence je in bo usmerjeno predvsem v razvoj in izvedbo metod in sistemov za pridobivanje, analizo, hranjenje in prenos znanja. S kombiniranjem delnih — problemsko usmerjenih rešitev, gradimo kompleksen in fleksibilen sistem za podporo KM, ki bo predstavljal osnovo globalnega informacijskega centra znanja.

Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja.


RTK
40
LET
TEKMOVANJ V ZNANJU
RAČUNALNIŠTVA IN INFORMATIKE
V SLOVENIJI


acm