17. tekmovanje ACM v znanju računalništva

26. marca 2022

Bilten

Bilten 17. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, Ljubljana, 2024

Elektronska izdaja

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Urban Duh, Matija Grabnar, Bor Grošelj Simić, Gregor Kikelj, Tomaž Hočevar, Boris Horvat, Josip Klepec, Vid Kocijan, Maks Kolman, Filip Koprivec, Samo Kralj, Mitja Lasič, Mark Martinec, Polona Novak, Ella Potisek, Jakob Schrader, Jure Slak, Mitja Trampuš, Borut Žnidar, Peter Žnidar, Janez Brank.

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

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=198695171
ISBN 978-961-264-290-7 (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. (Kot običajno se jih je tudi letos velika večina odločila pisati odgovore na računalniku in ne na papir.) 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.

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 s standardnega vhoda, izračuna neki rezultat in ga izpiše na standardni izhod. Programe se potem ocenjuje tako, da se jih na ocenjevalnem računalniku izvede na več testnih primerih, število točk pa je sorazmerno s tem, pri koliko testnih primerih je program izpisal pravilni rezultat. (Podrobnosti točkovanja v 3. skupini so opisane na strani 21.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C#, java, python in rust.

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

Tekmovalci vseh treh skupin si lahko pri reševanju pomagajo z zapiski in literaturo, pa tudi z dokumentacijo raznih programskih jezikov, ki je nameščena na tekmovalnih računalnikih.

Na začetku smo tekmovalcem razdelili tudi list z nekaj nasveti in navodili (str. 7–9 za 1. in 2. skupino, str. 21–23 za 3. skupino).

Omenimo še, da so rešitve, objavljene v tem biltenu, večinoma obsežnejše od tega, kar na tekmovanju pričakujemo od tekmovalcev, saj je namen tukajšnjih rešitev pogosto tudi pokazati več poti do rešitve naloge in bralcu omogočiti, da bi se lahko iz razlag ob rešitvah še česa novega naučil.

Od leta 2017 objavljamo v biltenu rešitve v C++17, za prvo skupino pa tudi v pythonu, ker precej 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 226–229.

Podobno kot v zadnjih nekaj letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 28. januarja 2022. To je imelo eno samo težavnostno skupino, naloge (ki jih je bilo pet) pa so pokrivale precej širok razpon težavnosti. Tekmovalci so dobili enake strani 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. 210–213). 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 188 tekmovalcev z 28 šol (vse so bile srednje, razen osnovnošolske skupine za priprave na računalniška tekmovanja).

Prejšnji dve leti (2020–21) je tekmovanje zaradi epidemije potekalo v celoti prek interneta, letos pa spet v živo na Fakulteti za računalništvo in informatiko. Tradicionalno na računalnikih za prvo in drugo skupino ni bilo prevajalnikov in razvojnih orodij, pač pa le urejevalniki besedil, da bi bili tako pogoji za tekmovalce, ki pišejo odgovore na računalniku, čim bolj izenačeni s tistimi, ki pišejo na papir. Toda v prejšnjih dveh letih, ko so tekmovalci delali od doma, so imeli vsi dostop do prevajalnikov in razvojnih orodij na svojih računalnikih; ker ni bilo videti, da bi tekmovanju to kaj škodovalo, in ker so tekmovalci že prej v anketah pogosto izražali želje, da bi imeli prevajalnike na voljo tudi v prvi in drugi skupini, smo se odločili, da bomo od letos dali tekmovalcem prve in druge skupine na voljo enako okolje kot tistim v tretji skupini, tako da so imeli tekmovalci vseh treh skupin dostop do prevajalnikov in razvojnih orodij.

Letos je v Sloveniji potekalo tudi srednjeevropsko študentsko tekmovanje v računalništvu (CERC 2022), zato v letošnjem biltenu objavljamo tudi rezultate (str. 224) ter slovenske prevode nalog (str. 35–52) in opise rešitev (str. 123–180) s tega tekmovanja.

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 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(f"{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(f"{i}. vrstica: \"{s}\"")
print(f"{i} vrstic, {d} znakov.")

# 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(f"Skupaj {i} znakov.")
```

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. Če oddajaš kaj na papirju, napiši na vsak oddani list svoje ime. Pri delu si lahko pomagaš s prevajalniki in razvojnimi orodji, ki so na voljo na tvojem računalniku, vendar bomo tvoje odgovore v vsakem primeru pregledali in ocenili ročno (ne glede na to, ali si jih oddal prek računalnika ali na papirju), zato manjše napake v sintaksi ali pri klicih funkcij standardne knjižnice niso tako pomembne, kot bi bile na tekmovanjih z avtomatskim ocenjevanjem.

Tekmovanje bo potekalo na strežniku `https://rtk.fri.uni-lj.si/`, kjer dobiš naloge in oddajaš svoje odgovore. Uporabniška imena in gesla vas bodo čakala na mizi v učilnici. Pri oddaji preko računalnika odpreš dotično nalogo v spletni učilnici in rešitev natipkaš oz. prilepiš v polje za programsko kodo. 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“. Ker je vgrajeni urejevalnik dokaj preprost in ne omogoča označevanja kode z barvami, predlagamo, da rešitev pripraviš v kakšnem drugem urejevalniku na računalniku (Visual Studio Code, Geany, Lazarus) in jo nato prekopiraš v okno spletnega urejevalnika. Naj te ne moti, da se bodo barvne oznake kode pri kopiranju izgubile.

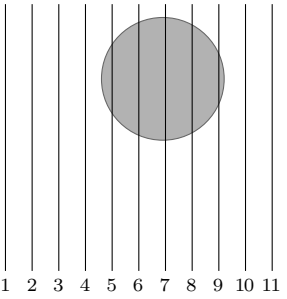
Ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo ali ko želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge, uporabi gumb „Shrani spremembe“ in nato klikni na „Nazaj na seznam nalog“, da se vrneš v glavni meni. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na svojem lokalnem računalniku.

Med reševanjem lahko vprašanja za tekmovalno komisijo postavljaš prek zasebnih sporočil na tekmovalnem strežniku (ikona oblačka zgoraj desno) ali pa vprašaš člane komisije, ki bodo prisotni v učilnicah. Prek zasebnih sporočil bomo pošiljali tudi morebitna pojasnila in popravke, če bi se izkazalo, da so v besedilu nalog kakšne nejasnosti ali napake. Zato med reševanjem redno preverjaj, če so se pojavila kakšna nova zasebna sporočila. Če imaš težave z računalnikom ali s povezavo s spletnim strežnikom za oddajo nalog in komunikacijo s tekmovalno komisijo, se nemudoma obrni na nadzornika v učilnici, ki bo zagotovil drug računalnik. **Če zaradi morebitnih težav pri oddajanju rešitev na strežnik želiš, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici, še preden odideš iz nje.**

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. Snežinke

Radi bi imeli napravo za merjenje velikosti snežink. V ta namen imamo rešetko vzporednih toplih merilnih žic, razdalja med njimi je en milimeter. Žice so oštevilčene od 1 do 100. Predpostavimo, da so snežinke okrogle, cela snežinka je vedno znotraj območja merilnih žic in vedno se dotakne vsaj ene žice, nato pa spolzi skozi (snežinke se ne nabirajo na žicah). Med dvema snežinkama je vedno dovolj časovnega razmaka, da se prejšnja stopi in spolzi mimo merilnih žic.



Slika kaže prvih enajst od 100 žic. Snežinko, ki jo predstavlja sivi krog, zaznavajo žice 5, 6, 7, 8 in 9.

Na voljo imaš funkcijo Senzor(n), ki vrne vrednost **true**, če n-ta žica zaznava snežinko, sicer vrne **false**. Predpostavimo, da se snežinka dotakne vseh žic hkrati, ko pa se čez čas stopi, je v istem trenutku ne zaznava nobena žica več. Delovanje programa je dovolj hitro, da lahko v času obstoja ali odsotnosti snežinke na merilnih žicah večkrat odčitamo stanje merilnih žic. Snežinka lahko pade na žice kadarkoli, tudi med dvema zaporednima klicema funkcije Senzor.

Napiši program, ki stalno pregleduje stanje merilnih žic in ob vsaki novi snežinki izpiše (natanko enkrat), koliko žic se je dotaknila (in tako izvemo njeno približno velikost). V primeru snežinke z gornje slike mora program izpisati 5.

2. Semafor

Na nekatere semaforje so v zadnjih letih namestili dodatne prikazovalnike, ki prikazujejo čas v sekundah do vklopa zelene luči. Ker želimo čas čakanja izkoristiti za kaj koristnega, smo v avto vgradili inteligentno kamero, ki prepozna številko na prikazovalniku.

Predpostavi, da je za dostop do kamere na voljo funkcija BeriStevec(), ki počaka, da se stanje prikazovalnika spremeni, in poskuša prepoznati številko, ki je po novem prikazana na prikazovalniku. Žal pa se je pri uporabi pokazalo, da kamera pri prepoznavanju ni vedno najbolj natančna (sonce, megla, kót snemanja), zato funkcija BeriStevec ne vrne nujno ene same številke, ampak eno ali več možnih številk, ki jih je kamera prepoznala. Vse te številke so z območja od 0 do 99; prava številka je zagotovo med njimi. Ko prikazovalnik kaže številko 0, vrne tudi BeriStevec le številko 0 in nobene druge. Funkcija BeriStevec je takšne oblike:

```
vector<int> BeriStevec();          // v C++
public static int[] BeriStevec();  // v javi ali C#
def BeriStevec() -> list[int]: ... # v pythonu
```

```
int stevilke[100];   /* v C/C++: funkcija BeriStevec shrani prepoznane številke v globalno */
int BeriStevec();    /* tabelo „stevilke“, kot vrednost funkcije pa vrne število teh števil. */

var stevilke: array [1..100] of integer;   { v pascalu; deluje enako }
function BeriStevec: integer;              { kot tista v C/C++ }
```

Napiši program, ki s čim manj zaporednimi klici funkcije BeriStevec ugotovi, katera številka je trenutno na prikazovalniku, in jo izpiše.

Primer: spodnja tabela kaže možen potek dogajanja pri več zaporednih klicih. V prvem stolpcu je prava številka s prikazovalnika, v drugem pa je seznam številk, ki bi ga utegnila vrniti funkcija BeriStevec.

Prikazana številka	Ob klicu BeriStevec vrne
53	$[59, 93, 53, 99]$
52	$[52, 92, 56, 96]$
51	$[51, 57, 91]$
50	$[90, 50, 58, 98]$
49	$[43, 79, 45, 48, 49]$

Po klicih v gornjem primeru bi se že dalo z gotovostjo zaključiti, da je številka na prikazovalniku res 49 in ne kakšna druga. Tvoj program naj zato tedaj izpiše 49.

3. Iskanje kvadrata

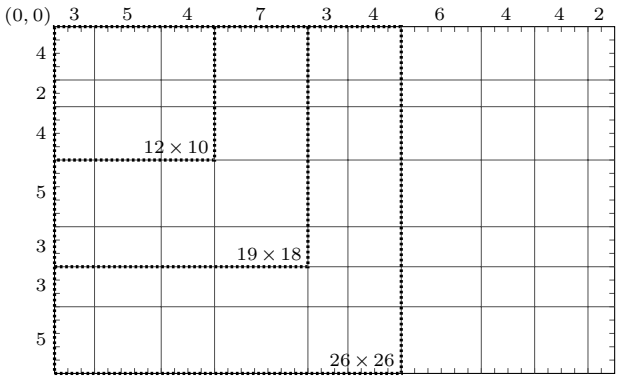
Imamo tabelo oz. razpredelnico (*spreadsheet*) z različnimi širinami stolpcev in višinami vrstic. Če vzamemo presek zgornjih nekaj vrstic in levih nekaj stolpcev, lahko dobimo pravokotnike različnih oblik in velikosti (odvisno od tega, koliko vrstic in koliko stolpcev smo uporabili), vsi pa se začnejo v zgornjem levem kotu. Med vsemi takimi pravokotniki želimo izbrati tistega, ki je po obliki čim bližje kvadratu (ali pa je celo res kvadrat). Z drugimi besedami: radi bi, da bi bilo razmerje med dolžino daljše in krajše stranice pravokotnika čim bližje 1.

Opiši postopek (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki kot vhodne podatke dobi zaporedje širin stolpcev in zaporedje višin vrstic ter izračuna širino in višino pravokotnika, o katerem govori prejšnji odstavek. Če je možnih več enako dobrih rešitev, je vseeno, katero izpiše. Širine stolpcev in višine vrstic so cela števila, večja od 0. Poleg tega tudi dobro **utemelji**, zakaj tvoj postopek vrača pravilne rešitve.

Primer: na sliki na str. 14 imamo stolpce s širinami $[3, 5, 4, 7, 3, 4, 6, 4, 4, 2]$ in vrstice z višinami $[4, 2, 4, 5, 3, 3, 5]$. Izmed vseh možnih pravokotnikov z začetkom v zgornjem levem kotu so označeni trije (z debelimi črtkanimi črtami), od katerih je najprimernejši tisti z velikostjo 26×26.

4. Neprevidni poeti

Znani poet Gimon Sregorčič se je odločil, da bo v umetnosti pesnjenja izuril nekaj vajencev. Ti seveda niso njegove generacije, temveč so bistveno mlajši in imajo na Gimonovo žalost bistveno bolj moderne poglede na svet in, kar je sploh hudo, poezijo. Najbolj opazna stvar je, da se sploh ne držijo ritma. Celo noč je obupan



---

prebiral njihove pesnitve in beležil, kdo je upošteval ritem in kdo ga ni. Zdaj je že tako utrujen, da tudi sam ne zaznava več poudarjenih in nepoudarjenih zlogov in potrebuje pomoč pri ugotavljanju, katere pesmi se držijo ritma.

V besedilu pesmi so nekateri zlogi naglašeni, drugi pa ne. Zaporedje naglašenih in nenaglašenih zlogov tvori ritem. V nalogi so naglašeni samoglasniki označeni z velikimi črkami, vse ostale črke pa so male. Sklop črk je zlog samo v primeru, da ima natanko en samoglasnik (*a*, *e*, *i*, *o* ali *u*) (v besedi „stržen“ je torej po našem štetju en zlog, zlogotvornega *r* ne upoštevamo). **Napiši program**, ki za dano pesem izpiše, ali je v vseh verzih pesmi isti ritem in če da, kakšen je. Tvoj program lahko prebere pesem s standardnega vhoda ali pa iz datoteke `vhod.txt` (kar ti je lažje). Predpostavi, da posamezne vrstice niso daljše od 100 znakov.

Primer vhoda:

```
od nEkdaj lepE so ljubljAnke slovEle
a lEpse od Urske bilO ni nobEne
nobEne ocEm bilo bOlj zazelEne
ne spOmnem se bEsedilA naprej sOri
```

Pripadajoči izhod:

```
DA
U-UU-UU-UU-U
```

Komentar: druga vrstica je ritem, `U` predstavlja nepoudarjen zlog, `-` pa poudarjen zlog.

Še en primer vhoda:

```
cuj vlAka zvIzg z vetrOvi gnAn
med mrAk oblAkov in poljAn
narascajOc, pojemajOc
takO moj klIc gre v nOc polnOc
```

Pripadajoči izhod:

```
NE
```

5. Stonoge

Malokdo ve, da je Alfred Hitchcock pred svojo uspešno grozljivko *Ptiči* posnel tudi *Stonoge*, ki pa iz različnih razlogov, med drugim zaradi izjemno nizkega filmskega proračuna, niso postale uspešnica. Ker stonoge niso pretirano ubogljiva bitja, so v filmu uporabljali izključno umetne stonoge, ki pa niso bile izdelane preveč prepričljivo. Za primer podajmo eno prepričljivo in eno očitno umetno stonogo:

```
........\||||||\/\///||/.............
.........###############>............
......../||||||/\/\/\||\.............
....................................
...................\|||/||/.........
...................<######..........
...................//||\||\.........
```

Trup stonoge torej tvori eden ali več zaporednih znakov „`#`“ (vsi v eni vrstici), glavo pa predstavlja bodisi znak „`<`“ tik levo ob trupu ali pa znak „`>`“ tik desno ob trupu. (V primeru zgoraj ima gornja stonoga glavo na desnem koncu, spodnja pa na levem.) Stonoge so prepričljive, če so vsi njihovi pari nog simetrični in če sta prvi in zadnji par nog usmerjena k trupu. V primeru zgoraj prva stonoga ni prepričljiva, ker peti par nog (gledano od spredaj) ni simetričen (leva noga kaže naprej, desna pa nazaj), druga stonoga pa je prepričljiva. V spodnjem primeru pa nobena stonoga ni prepričljiva, ker se prvi in/ali zadnji par nog pri nobeni ne drži trupa:

```
.............................////.............
..||\//.............|\\||/...##>.....//\\|..
...###>.............<####...\\\\......###>..
..||/\\..../|||/....|//||\...........\\//|..
...........<###..............................
...........\|||\..............................
........................................
```

Napiši program, ki iz položaja stonog v nekem trenutku filma ugotovi, koliko izmed njih je prepričljivih in koliko očitno umetnih. Stonoge se nahajajo na polju iz n vrstic in m stolpcev (n in m sta največ 100). Polje je sestavljeno zgolj iz znakov „`\`“, „`|`“, „`/`“, „`<`“, „`>`“, „`#`“ in „`.`“. Nobeni dve stonogi se ne prekrivata ali dotikata in nobena stonoga ni v kadru le delno. Predpostaviš lahko, da je z glavami stonog vse v redu in ti ni treba preverjati, ali ima res vsaka stonoga glavo na natanko enem koncu. Podrobnosti tega, v kakšni obliki tvoj program dobi ali prebere vhodne podatke, si izberi sam(a) in jih v svoji rešitvi tudi opiši.

NALOGE ZA DRUGO SKUPINO

[Pred nalogami so bila navodila, enaka tistim v prvi skupini (str. 11), zato jih tu ne bomo ponavljali.—*Op. ur.*]

1. Varnostno kopiranje

Podan imamo seznam poti do več map oz. imenikov (direktorijev) na računalniku, ki jih želimo kopirati na zunanji disk za varnostno kopijo. Toda na seznamu so tudi podvojene mape ter mape, ki so že podmape drugih map. Če imamo na seznamu mape `/home/user/`, `/home/user` in `/home/user/slike`, je to v resnici enako seznamu samo s `/home/user/`, saj se prvi dve poti nanašata na isto mapo, zadnja pa je podmapa in je ni treba kopirati posebej, saj bo skopirana, ko bomo kopirali njeno starševsko mapo.

Napiši program, ki prebere seznam poti in izpiše le tiste izmed njih, ki jih je treba skopirati, da se izognemo nepotrebnemu kopiranju. Pri tem ni pomemben vrstni red izpisanih poti, važno je le, da so podmnožica podanih poti in da jih je čim manj. Če obstaja več enako dobrih rešitev, je vseeno, katero izpišeš.

Predpostaviš lahko, da so poti na voljo v datoteki `poti.txt` in vsaka pot je v svoji vrstici. Zagotovljeno je, da se vse poti začnejo s poševnico „`/`“, imena map pa bodo vsebovala le male črke angleške abecede, števke ter podčrtaj „`_`“. Vse poti bodo veljavne (ni ti treba npr. skrbeti zaradi poti oblike `abc//def`). Dolžina vsake poti bo manjša od 4096 znakov.

Poti je lahko veliko, zato naj bo tvoj program čim bolj učinkovit.

Primer vhoda:

```
/home/admin/config
/home/user/slike
/home/user/slike_stare
/home/user/dokumenti/sola/
/home/user/dokumenti/sola/slo/spisi/
/home/admin/config/
/home/user/minecraft/savegames/svet1
/home/user/minecraft/savegames/svet2
/home/user/slike/2019/smucanje/
/var/www/web/
/home/admin/config/
```

Možen izhod (vrstni red ni pomemben):

```
/home/user/slike
/home/user/slike_stare
/home/user/dokumenti/sola/
/home/admin/config/
/home/user/minecraft/savegames/svet1
/home/user/minecraft/savegames/svet2
/var/www/web/
```

2. Luči

Dano je zaporedje n luči; znano je njihovo začetno stanje (katere so prižgane in katere ugasnjene) ter želeno končno stanje. Osnovna operacija, ki jo lahko nad lučmi izvajamo, je, da si izberemo števili i in j (z območja $1 \leq i \leq j \leq n$) in spremenimo stanje vseh luči od vključno i-te do vključno j-te (torej ugasnemo tiste med njimi, ki so bile prej prižgane, in prižgemo tiste, ki so bile prej ugasnjene).

Napiši podprogram (oz. funkcijo) `Luci(n, zacetno, koncno)`, ki izpiše zaporedje čim manj takšnih operacij, s katerimi lahko luči iz začetnega stanja spravimo v želeno končno stanje. Za vsako operacijo naj izpiše števili i in j, torej številko prve in zadnje spremenjene luči (luči so oštevilčene od 1 do n). Če obstaja več enako dobrih rešitev z najmanjšim možnim številom operacij, je vseeno, katero izpiše.

Kot parametra zacetno in koncno naj tvoj podprogram sprejme niza n znakov, ki opisujeta začetno in končno stanje luči (znak `'P'` predstavlja prižgano luč, znak `'U'` pa ugasnjeno).

Dobro tudi **utemelji**, zakaj tvoja rešitev res najde najmanjše možno število operacij.

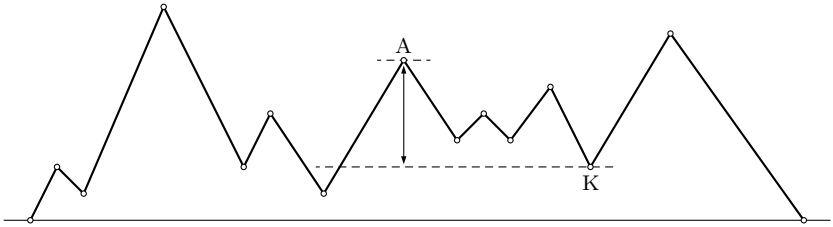
Primer: če imamo $n = 7$ luči in je začetno stanje `UUPPPUP`, končno pa `UPPUPPU`, potrebujemo vsaj tri operacije. Eno od možnih zaporedij treh operacij je: $i = 2$, $j = 5$ (dobimo `UPUUUUP`); $i = 4$, $j = 7$ (dobimo `UPUPPPU`); $i = 3$, $j = 4$ (dobimo `UPPUPPU`).

3. Planinarjenje

Za planince je pri izbiri ciljev ponavadi bolj zanimiv vrh, ki se izraziteje dviga višje od svoje okolice, manj pomembna pa je njegova absolutna nadmorska višina.

V ta namen je vpeljan pojem *topografske prominence* vrha (ali njegova *relativna višina*). Ta je določena z višinsko razdaljo (razliko višin) med vrhom in najnižjo tako plastnico (izohipso) terena, ki obkroža ta vrh in hkrati ne obkroža kakšnega višjega vrha. Z drugimi besedami: če bi se morska gladina dvignila do tiste plastnice, bi bil ta vrh najvišja točka otoka.

Problem si poenostavimo v dve dimenziji. Na spodnji sliki je prominenca vrha A razlika med nadmorskima višinama točk A in K (če bi se voda dvignila do višine točke K oz. tik nad njo, bi bila točka A najvišji vrh svojega otoka):



Da se ne trudimo z iskanjem vrhov in dolin, so podatki že pripravljeni tako, da si v seznamu (ali vhodni datoteki) sledijo izmenoma nadmorske višine dolin in vrhov. (Na gornji sliki so to točke, označene s krožci ∘, tako da bi za ta primer dobili seznam 15 višin.) Podatkov je liho število, seznam pa se začne in konča z dolino na nadmorski višini 0 (vse ostale višine so večje od 0).

Napiši program ali podprogram (funkcijo), ki bo za vsak vrh iz seznama ugotovil in izpisal njegovo prominenco. (Na primeru iz gornje slike bi moral program torej izpisati rezultate za 7 vrhov.) Tvoj (pod)program lahko seznam dobi kot parameter ali globalno spremenljivko (vektor, tabelo ali kaj podobnega), lahko pa ga prebere s standardnega vhoda ali iz vhodne datoteke (kar ti je lažje).

4. Sedežni red

Zloglasni 5. c je dobil novo učiteljico. Ta se je odločila, da bo v razredu naredila red, in sicer tako, da bo otroke posedla na točno določen način. Po novih pravilih noben otrok ne sme sedeti za otrokom, ki je strogo višji kot on sam, prav tako pa

drug poleg drugega v isti vrsti ne smeta sedeti dva dečka ali dve deklici, ker bi bilo v tem primeru zagotovo preveč klepetanja.

Primera dobrih sedežnih redov za 6 otrok, kjer črka predstavlja spol, številka pa višino:

```
143M   150Z
139Z   128M
129M   127Z
  (TABLA)

139Z   154M
135Z   148M
135Z   129M
  (TABLA)
```

Dva primera slabih postavitev:

```
130Z   154M
135Z   148M
130Z   129M
  (TABLA)
```

(V gornji postavitvi je učenka levo v zadnji vrsti nižja od učenke pred njo.)

```
150Z   154M
145Z   148Z
141M   129M
  (TABLA)
```

(V tej postavitvi v prvi in drugi vrsti drug poleg drugega sedita otroka istega spola.)

Opiši postopek (ali napiši (pod)program oz. funkcijo, če ti je lažje), ki prejme podatke o višinah in spolu vseh u otrok v razredu ter jih razporedi v n vrst in m stolpcev tako, kot si je to zaželela učiteljica (ali pa ugotovi, da takšen razpored sploh ne obstaja). Za število stolpcev m predpostavi, da je sodo. Če je možnih več različnih pravilnih postavitev, je vseeno, katero izmed njih najde tvoj postopek. S podrobnostmi branja vhodnih podatkov in izpisa rezultatov se ti ni treba ukvarjati.

5. Žabe

Nad močvirjem leta r rojev muh. Vsak roj je sestavljen iz zelo velikega števila muh. Gibanje posameznega roja opišemo z zaporedjem koordinat in časov, ko se roj na svoji poti za trenutek ustavi. Tako bi postanke i-tega roja muh opisali s seznamom trojic $(x_{i,j}, y_{i,j}, t_{i,j})$, ki predstavljajo j-ti postanek i-tega roja na koordinati $(x_{i,j}, y_{i,j})$ ob času $t_{i,j}$. Predpostavimo lahko, da so postanki na poti posameznega roja podani po naraščajočih časih, torej $t_{i,j} < t_{i,j+1}$.

Poleg muh se v močvirju nahaja tudi z žab, ki lovijo muhe. Žaba lahko ulovi muho iz roja samo v trenutku, ko se roj ustavi, če se ravno takrat nahaja na istem mestu kot roj. Poznamo tudi lokacije žab: i-ta žaba se trenutno (ob času 0) nahaja na koordinati (a_i, b_i). Vse žabe se lahko premikajo po močvirju s hitrostjo 1 enote

na sekundo (lahko pa tudi stojijo pri miru). Žaba se lahko med poljubnima dvema točkama premika v ravni črti, torej za merjenje razdalje uporabi evklidsko razdaljo (Pitagorov izrek).

Žabe načrtujejo lov na muhe. Ulovile bodo nekaj muh, nato pa se zbrale v koordinatnem izhodišču $(0, 0)$, kjer si bodo plen razdelile. Vsaka žaba bo ujela največ eno muho. Ker so muhe dobro rejene, je žabam dovolj, če vse skupaj ujamejo k muh (velja $k \leq z$), ki si jih nato razdelijo.

Opiši in utemelji **postopek**, ki bo določil najkrajši čas, v katerem se lahko vse žabe zberejo v izhodišču in pri tem skupaj ulovijo k muh. Koordinate in časi so realna števila.

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

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

Tvoji programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali python, mi pa jih bomo preverili s prevajalniki FreePascal, GNUjevima `gcc` in `g++` 9.4.0 (ta verzija podpira C++17, novejše različice standarda C++ pa le delno), prevajalnikom za javo iz JDK 17, s prevajalnikom Mono 6.8 za C# in z interpreterjem za python 3.8.

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2022-3/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi rešitve svojih nalog. Pred začetkom tekmovanja lahko poskusiš oddati katero od nalog iz arhiva `https://putka-rtk.acm.si/tasks/s/test-sistema/list/`. Uporabniško ime in geslo za Putko sta enaki kot za računalnike. Med tekmovanjem lahko vprašanja za tekmovalno komisijo postavljaš prek foruma na Putki (povezava „Diskusija“ na dnu besedila posamezne naloge).

Sistem na spletni strani bo tvojo izvorno kodo prevedel in pognal na več testnih primerih. Za vsak testni primer se bo izpisalo, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal predolgo ali pa porabil preveč pomnilnika (točne omejitve so navedene na ocenjevalnem sistemu pri besedilu vsake naloge), ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika (za podrobne nastavitve prevajalnikov na ocenjevalnem strežniku glej `https://putka-rtk.acm.si/info/`. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku.

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 ti lahko prinese od 0 do 100 točk. Vsak oddani program se preizkusi na več testnih primerih; pri vsakem od njih dobi vse točke, če je izpisal pravilen odgovor, sicer pa 0 točk (izjema je 1. naloga, kjer je možno tudi delno točkovanje). Pri drugi nalogi je testnih primerov 20 in vsak je vreden po 5 točk, pri ostalih nalogah pa je testnih primerov po 10 in vsak je vreden po 10 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.

Primer naloge (ne šteje k tekmovanju)

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

Primer vhoda:

```
123 456
```

Ustrezen izhod:

```
5790
```

Primeri rešitev:

- V pascalu:

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

- V C-ju:

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

- V C++:

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

- V pythonu:

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

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

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

- V javi:

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

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

- V C#:

```
using System;

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

NALOGE ZA TRETJO SKUPINO

1. Mastermind

Pri igri Mastermind računalnik izbere zaporedje n barvnih žetonov (barve se lahko ponavljajo) iz nabora m barv, ki jih označujemo s števili od 1 do m, in ga ne razkrije igralcu. Igralec nato poskuša uganiti izbrano zaporedje, računalnik pa mu ob vsakem ugibanju odgovori s parom števil, od katerih prvo pove, koliko žetonov je v danem ugibanju prave barve in na pravem mestu, drugo število pa pove, koliko žetonov je prave barve, vendar ne na pravem mestu.

Če je žetonov neke barve v ugibanju več kot v računalnikovem zaporedju, se pri vračanju informacij upošteva samo toliko žetonov, kot jih je v računalnikovem zaporedju, pri čemer imajo prednost žetoni prave barve na pravem mestu. Na primer: če računalnik izbere 1 1 2 2 2 in če igralec ugiba 1 3 1 1 1, bo računalnik vrnil $(1, 1)$. (Prvi žeton v igralčevem zaporedju je prave barve na pravem mestu, eden od zadnjih treh pa je prave barve na napačnem mestu.)

Recimo, da je v nekem ugibanju igralec predlagal zaporedje u, računalnik pa mu je odgovoril s parom števil o; rekli bomo, da je zaporedje z *skladno* z ugibanjem (u, o), če bi računalnik v primeru, da je njegovo izbrano zaporedje (ki ga mora igralec uganiti) ravno z in da je igralec pri ugibanju predlagal zaporedje u, odgovoril z odgovorom o.

Naš igralec je že podal k ugibanj in dobil odgovore nanje. V svojem naslednjem ugibanju želi predlagati takšno zaporedje p, po katerem bo tudi v najslabšem primeru (najslabšem po vseh možnih računalnikovih odgovorih) število še skladnih zaporedij (torej takih, ki so skladna z vsemi preteklimi ugibanji in tudi s tem novim, pravkar izvedenim ugibanjem) najmanjše možno. S tem kriterijem pa p ni nujno enolično določen; mogoče je, da obstaja več enako dobrih optimalnih predlogov p. **Napiši program**, ki iz zgodovine ugibanj in računalnikovih odgovorov nanje določi število vseh možnih takšnih zaporedij p, poleg tega pa še število vseh zaporedij, ki so skladna z dosedanjimi ugibanji in odgovori.

Vhodni podatki: v prvi vrstici so cela števila n (dolžina zaporedij), m (število barv) in k (število preteklih ugibanj). Nato sledi k vrstic s po $n + 2$ številoma; pri tem prvih n števil predstavlja igralčevo ugibanje, zadnji dve pa računalnikov odgovor.

Omejitve: $1 \leq n \leq 5$, $1 \leq m \leq 8$, $0 \leq k \leq 5$; število še skladnih zaporedij glede na dano zgodovino ugibanj je vsaj 1 in manj kot 100.

Izhodni podatki: izpiši dve celi števili, vsako v svoji vrstici. Prvo naj bo število zaporedij, ki so skladna z dosedanjimi ugibanji in odgovori nanje; drugo pa naj bo število takšnih zaporedij p, o katerih govori besedilo naloge.

Točkovanje: za vsako od obeh izhodnih števil dobiš polovico točk, če je pravilno (še vseeno pa moraš izpisati dve števili, sicer ne dobiš nobene točke).

Primer vhoda:

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

Pripadajoči izhod:

```
4
30
```

Razlaga: računalnikova zaporedja, ki so skladna z dano zgodovino, so štiri: 1 1 2, 1 5 2, 2 2 1 in 5 2 1. Če igralec ugiba na primer 1 5 3, je računalnikov odgovor pri vseh še skladnih zaporedjih drugačen; v najslabšem primeru (in pravzaprav v vseh primerih) bo po tem ugibanju torej obstajalo samo še eno tako zaporedje, ki bo skladno z vsemi odgovori. To pa ne velja le za ugibanje 1 5 3, pač pa še za devetindvajset drugih ugibanj, zato je pravilni odgovor pri tem testnem primeru 30.

2. Dolgovi

Srednja šola Butale se je odločila, da bo po prekinitvi ukrepov zaradi popularnega virusa organizirala ekskurzijo. Na njej ima vseh n dijakov polpenzion, kar pomeni, da si bodo morali kar nekaj obrokov priskrbeti sami.

Butalski dijaki pa so prijazna bitja in nočejo po nepotrebnem obremenjevati raznih natakarjev, dostavljalcev hrane in drugih prodajalcev. Zato so dogovorjeni, da jedo v skupini (njeno članstvo se skozi čas lahko spreminja), pri čemer vsak kdaj plača nakup in bodo na koncu ekskurzije poravnali račune med sabo. Pri tem ne upoštevajo morebitnih razlik v ješčosti dijakov in si bodo zneske razdelili enakomerno po vseh, ki so trenutno prisotni v skupini. Ker pa matematika naših Butalcev šepa, te prosijo, da jim **napišeš program**, ki iz zapisa njihovega zapravljanja izračuna, koliko je kdo „v plusu“ ali „v minusu“. Ta zapis je zaporedje *dogodkov* naslednjih treh tipov:

1. `+` *ime* ali `-` *ime* — pomeni, da se je oseba z imenom *ime* pridružila skupini (če je prvi znak `+`) ali jo zapustila (če je prvi znak `-`). Kdor skupino zapusti, se ji lahko kasneje tudi spet pridruži.

2. `+` *prefiks*`*` ali `-` *prefiks*`*` — pomeni, da so se skupini pridružile oz. jo zapustile vse tiste osebe, ki so bile do tega trenutka že omenjene v dogodkih prvega tipa in ki se jim ime začne na niz *prefiks*. Dogodkom tega tipa pravimo „prefiksni dogodki“. Vsaka oseba je torej prvič dodana s svojim celotnim imenom, kasneje pa je lahko dodana ali odstranjena tudi samo s prefiksom. Prefiks je lahko tudi prazen — v tem primeru se vrstica nanaša na vse dotlej poznane osebe.

3. `<-` *ime* z — pomeni, da je oseba z imenom *ime* darovala znesek z v skupno dobro. Pri tem se njen dolg do skupnosti zmanjša za ta znesek, nato pa se dolg vseh trenutnih članov skupine poveča za z/k, če je k trenutno število ljudi v skupini (predpostaviš lahko, da se dogodki tega tipa nikoli ne zgodijo takrat, ko je skupina prazna). Oseba, ki nekaj plača v skupno dobro, v tistem trenutku ni nujno tudi sama član skupine (je pa bila pred plačilom že vsaj enkrat dodana v skupino).

Predpostaviš lahko, da ne bo nobeno dodajanje (niti posamezno niti s prefiksom) pokrivalo kakšne take osebe, ki je trenutno že v skupini, in prav tako ne bo nobeno brisanje (niti posamezno niti s prefiksom) pokrivalo kakšne take osebe, ki je trenutno ni v skupini. Pri prefiksnih dogodkih velja to zagotovilo samo za tiste osebe, ki so bile pred tem vsaj enkrat že dodane v skupino. (Primer: zaporedje dogodkov `+ jaka`, `+ jure`, `- jaka`, `- j*` se v vhodnih podatkih ne more pojaviti; prav tako

ne zaporedje `+ jaka`, `+ jure`, `- j*`, `+ jaka`, `+ j*`. Lahko pa se pojavi zaporedje `+ jaka`, `- j*`, `+ j*`, `+ jure`.)[1]

Vhodni podatki: v prvi vrstici je celo število q, ki pove, koliko dogodkov sestavlja ta testni primer. Sledi q vrstic, od katerih vsaka opisuje po en dogodek v taki obliki, kot je opisano zgoraj.

Izhodni podatki: za vsako osebo, ki je bila kdaj dodana v skupino, izpiši po eno vrstico; v njej naj bo najprej ime osebe, nato presledek, nato pa znesek, ki ga ta oseba dolguje skupnosti (to število je lahko tudi negativno, če je v resnici skupnost dolžna tej osebi). Vrstni red oseb v izhodnih podatkih je lahko poljuben.

Rešitev se bo štela kot pravilna, če bodo dolgovi od uradne rešitve odstopali za manj kot 10^{-5} po relativni ali absolutni vrednosti.

Omejitve:

- Vedno bo veljalo $2 \leq q \leq 10^6$. Če z n označimo število različnih imen, omenjenih v ne-prefiksnih dogodkih, bo veljalo tudi $1 \leq n \leq 10^5$.
- Imena so sestavljena iz vsaj 1 in največ 10 znakov, vsi pa so male črke angleške abecede; pri prefiksnih dogodkih je prefiks dolg vsaj 0 in največ 9 malih črk, za njim pa pride še znak `*` (zvezdica). Imena so enolična (nobena dva človeka nimata enakega imena).
- Zneski pri dogodkih `<-` so cela števila z območja $0 \leq z < 10^5$.

Pri nekaterih testnih primerih veljajo dodatne omejitve:

- Pri prvih 20 % testnih primerov bo veljalo $n \leq 1000$ in $q \leq 5000$.
- Pri naslednjih 10 % testnih primerov ne bo prefiksnih dogodkov in veljalo bo $n \leq 1000$ ter $q_3 \leq 5000$ (kjer q_3 pomeni število dogodkov 3. tipa).
- Pri naslednjih 10 % testnih primerov ne bo prefiksnih dogodkov.
- Pri preostalih 60 % testnih primerov ni dodatnih omejitev.

Primer vhoda:

```
14
+ ljubinka
<- ljubinka 37
+ vilma
<- vilma 80
+ ticjana
<- ljubinka 48
+ onur
<- vilma 54
+ bernardica
- ticjana
+ ticjana
<- ticjana 22
<- vilma 35
<- onur 62
```

Eden od možnih pripadajočih izhodov:

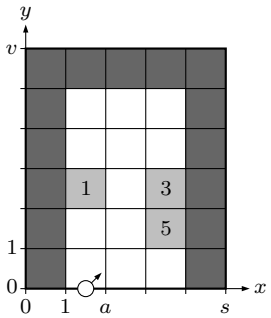
```
vilma -75.700000
onur -24.700000
ticjana 31.300000
ljubinka 45.300000
bernardica 23.800000
```

[1] Zanimivo, vendar težjo različico naloge dobimo, če to zagotovilo ukinemo. V tej težji različici se torej lahko zgodi, da se dodajanje (bodisi s prefiksom ali posamično) nanaša (tudi) na enega ali več takih ljudi, ki so trenutno že v skupini; tedaj ti ljudje pač ostanejo v skupini. Podobno se lahko tudi zgodi, da se brisanje nanaša na ljudi, ki jih trenutno ni v skupini, in taki pač pri tem tudi ostanejo zunaj skupine. Pri tej različici naloge je na primer možno zaporedje dogodkov `+ jaka`, `+ jure`, `+ jaka`, `- ja*`, `- j*` (pri tretjem od teh dogodkov se skupina ne spremeni, pri četrtem jo zapusti Jaka, pri petem pa še Jure; prefiks `j*` pri zadnjem dogodku sicer pokrije tudi Jako, vendar tega pač takrat ni v skupini in se zato zanj pri dogodku `- j*` nič ne spremeni).

3. Breakout

Morda poznate računalniško igro Breakout. Cilj igre je z odbijanjem žogice razbiti vse opeke na igralnem polju. Igralno polje je z leve, desne in zgornje strani obdano s stenami, znotraj polja pa se nahaja več opek. Žogica se odbija po polju in razbija opeke ob odboju od njih, pri tem pa moramo s premikanjem odbojne palice na spodnjem robu skrbeti, da žogica slučajno ne pade iz igralnega polja.

V tej nalogi ne bomo upoštevali odbojne palice, temveč nas zanima samo, kako bi se žogica odbijala po igralnem polju brez posredovanja igralca. Polje predstavimo s pravokotno mrežo širine s in višine v. Levo spodnje oglišče mreže se nahaja na koordinati $(0, 0)$, desno zgornje pa na (s, v). Nekatere celice polja vsebujejo opeke, ki se razbijejo po določenem številu odbojev žogice od njih. Najbolj levi in desni stolpec ter zgornja vrstica vsebujejo stene, ki jih bomo predstavili s trdnimi opekami, ki se razbijejo šele po 100 odbojih. Če se opeka zaradi odboja razbije, se to zgodi šele po odboju — žogica torej svojo pot nadaljuje enako, kot bi jo, če se opeka ob tem odboju ne bi razbila. Žogica začne svojo pot v spodnji vrstici na sredini a-tega stolpca, torej na koordinatah $(a - \frac{1}{2}, 0)$, in se od tam sprva (do prvega odboja) premika pod kotom 45° v smeri desno navzgor, kot kaže spodnja ilustracija:



Primer igralnega polja širine $s = 5$ in višine $v = 6$. Temno siva polja so trdne opeke, ki tvorijo stene in se razbijejo po 100 odbojih; svetlo siva polja pa so običajne opeke in števila na njih povedo, po koliko odbojih se razbijejo. Prikazan je tudi začetni položaj in smer gibanja žogice za $a = 2$.

Napiši program, ki izračuna, koliko opek razbije žogica, preden izstopi iz igralnega polja, in na kateri koordinati se to zgodi. Izstopi lahko v spodnji vrstici ali pa tudi kje drugje, če je zaradi številnih odbojev prebila steno.

Vhodni podatki: v prvi vrstici so podana cela števila s, v, n in a, ki po vrsti predstavljajo širino igralnega polja, njegovo višino, število opek znotraj polja in začetni stolpec žogice. Naslednjih n vrstic opisuje opeke znotraj polja (v to niso vključene opeke, ki predstavljajo stene). Vsaka opeka je opisana v svoji vrstici s števili x_i, y_i in k_i, pri čemer sta (x_i, y_i) koordinati zgornjega desnega kota celice, ki jo ta opeka zaseda, k_i pa je število odbojev, po katerih se opeka razbije (velja $2 \leq x_i \leq s-1$, $1 \leq y_i \leq v-1$ in $1 \leq k_i \leq 100$). Opeke se bodo nahajale v različnih celicah igralnega polja in nobena od njih nima koordinat $(a, 1)$, pri katerih bi se prekrivala z začetnim položajem žogice.

Omejitve: veljalo bo $3 \leq s \leq 10^5$, $3 \leq v \leq 10^5$ in $0 \leq n \leq 10^5$. V prvih 40 % testnih primerov bo veljalo $s, v, n \leq 1000$. V naslednjih 40 % testnih primerov bo veljalo $s, v, n \leq 10^5$. V zadnjih 20 % testnih primerov ni dodatnih omejitev.

Izhodni podatki: v prvi vrstici izpiši število razbitih opek (kar naj vključuje tudi razbite stenske opeke), v drugi vrstici pa s presledkom ločeni koordinati x in y, na

katerih žogica izstopi iz polja. Če je koordinata celo število, jo tako tudi izpiši, če pa je ne-cela, jo izpiši na eno decimalko.

Primer vhoda:	Pripadajoči izhod:

`5 6 3 2`
`2 3 1`
`4 2 5`
`4 3 3`
	`1`
	`3.5 0`

(*Opomba:* to je primer, ki ga kaže slika zgoraj.)

4. Pijansko urejanje

V skladišču je n podstavkov, oštevilčenih od 0 do $n-1$. Na vsakem podstavku stoji po en zaboj; tudi ti so oštevilčeni od 0 do $n-1$, vendar so premešani — številke zabojev se ne ujemajo s številkami podstavkov, na katerih posamezni zaboj stoji. Radi bi jih pravilno uredili, torej tako, da bo zaboj 0 na podstavku 0, zaboj 1 na podstavku 1 in tako naprej. (Različni zaboji imajo različne številke; različni podstavki prav tako.)

Žal se z viličarjem po skladišču preganja pijani skladiščnik in nas ne pusti blizu, da bi si ogledali razpored zabojev ali jih sami premikali. Edino, kar lahko počnemo, je, da mu od zunaj vpijemo številke podstavkov; ko sliši od nas številko k, naj bi skladiščnik zamenjal zaboja na podstavkih 0 in k. Žal pri tem včasih zgreši in namesto podstavka k uporabi enega od sosednjih dveh ($k-1$ in $k+1$), vendar ne vemo, kateri podstavek je zares uporabil. Vemo pa, da si podstavek med temi tremi (torej $k-1$, k in $k+1$) izbere naključno, pri čemer imajo vsi trije podstavki enako verjetnost, da bodo izbrani. (Možnost $k-1$ seveda odpade, če je $k=0$; in podobno možnost $k+1$ odpade, če je $k=n-1$. V teh primerih skladiščnik naključno izbere enega od ostalih dveh možnih podstavkov.) V vsakem primeru pa nam skladiščnik po zamenjavi pove številko zaboja, ki je pri tem prišel na podstavek 0 (ne glede na to, s katerega podstavka je ta zaboj prišel).

Napiši program, ki z zaporedjem takšnih operacij uredi zaboje (torej poskrbi, da se bo številka vsakega podstavka ujemala s številko zaboja, ki stoji na njem). To je interaktivna naloga; tvoj program se bo z ocenjevalnim strežnikom „pogovarjal“ tako, da bo bral s standardnega vhoda in pisal na standardni izhod. Ta pogovor naj poteka po naslednjih korakih:

1. Na začetku preberi s standardnega vhoda eno vrstico, v kateri bo celo število n (in nič drugega).

2. Nato lahko izvedeš 0 ali več zamenjav. Zamenjavo izvedeš tako, da na standardni izhod izpišeš eno vrstico, v kateri naj bo le celo število k (zanj mora veljati $0 \le k < n$), torej številka podstavka, za katerega želiš izvesti zamenjavo (še enkrat poudarimo, da bo računalnik zamenjavo mogoče izvedel s podstavkom $k-1$ ali $k+1$ namesto k). Nato s standardnega vhoda preberi eno vrstico; v njej bo le eno celo število, namreč številka zaboja, ki je pri tej zamenjavi prišel na podstavek številka 0.

3. Na koncu izpiši vrstico, v kateri naj bo le celo število -1, in prenehaj z izvajanjem.

Opozorilo: po vsaki izpisani vrstici splakni standardni izhod (*flush*), da bodo podatki res sproti prišli do ocenjevalnega sistema.

Tvoj program dobi pri posameznem testnem primeru vse točke, če izvede kvečjemu 10 000 zamenjav in če so na koncu zaboji pravilno urejeni; sicer pa ne dobi pri njem nobene točke (npr. če izvede preveč zamenjav, če zaboji na koncu niso pravilno urejeni ali če izpiše kaj, kar ni v skladu z zgoraj opisanim protokolom).

Omejitve: veljalo bo $2 \leq n \leq 100$. Pri 50 % testnih primerov bo $n \leq 40$.

Primer:

Tvoj program izpiše	Sistem izpiše	Komentar
	`2`	v skladišču sta dva zaboja
`1`	`1`	zaboj 1 je prišel na podstavek 0
`0`	`1`	zaboj 1 je še vedno na podstavku 0
`1`	`0`	zaboj 0 je prišel na podstavek 0
`-1`		končali smo z urejanjem

V gornjem primeru smo po tretji zamenjavi od sistema izvedeli, da je zaboj 0 prišel na podstavek 0; torej mora biti takrat zaboj 1 na podstavku 1 in lahko sklepamo, da sta oba zaboja urejena tako, kot naloga zahteva.

5. L-sistem

Teoretični biolog Polde preučuje obnašanje preprostih mnogoceličnih mikroorganizmov. Organizem je zaporedje celic; obstaja a različnih tipov celic in Polde je vsakemu tipu celice pripisal neko malo črko angleške abecede (črko, ki predstavlja i-ti tip celice, označimo s t_i), zato lahko organizem opiše z nizom takih črk. Organizem se skozi čas razvija, vendar na zelo predvidljiv način: v enem dnevu iz vsake celice tipa t_i nastane (vedno enako) zaporedje 0 ali več celic, ki ga torej spet lahko predstavimo z nizom znakov; temu nizu recimo $f(t_i)$.

Iz organizma, ki ga tvori niz n celic $s = c_1 c_2 \ldots c_n$, torej v enem dnevu nastane niz $f(c_1) \cdot f(c_2) \cdot \ldots \cdot f(c_n)$; temu novemu nizu recimo $f(s)$. Pri tem pike „$\cdot$“ pomenijo, da moramo nize $f(c_1), \ldots, f(c_n)$ po vrsti stakniti med seboj.

Po dveh dneh torej iz s nastane niz $f(f(s))$, kar zapišimo krajše kot $f^2(s)$; po treh dneh nastane $f(f(f(s)))$, kar zapišimo krajše kot $f^3(s)$; in tako naprej. Nizu, ki nastane po k dneh, recimo $f^k(s)$. Velja torej $f^0(s) = s$ in $f^k(s) = f(f^{k-1}(s))$.

Poldeta še posebej zanimajo organizmi, pri katerih se v njihovem nizu čim večkrat pojavlja podniz p ali kakšen od njegovih anagramov (to so nizi, ki jih lahko dobimo iz p, če premešamo vrstni red črk v njem). Pomagaj mu in **napiši program**, ki za dani začetni niz s in število dni k izračuna, koliko je v nizu $f^k(s)$ takih strnjenih podnizov, ki so anagrami niza p (tudi p je seveda sam svoj anagram). Ker utegne biti to število zelo veliko, bo dovolj, če v resnici izračunaš njegov ostanek po deljenju z nekim manjšim številom M.

Vhodni podatki: v prvi vrstici je niz $t_1 t_2 \ldots t_a$, ki ima torej toliko znakov, kolikor različnih tipov celic je pri tem testnem primeru, in i-ti znak tega niza pove, s katero črko so predstavljene celice i-tega tipa. Sledi a vrstic, od katerih i-ta vsebuje niz $f(t_i)$ — to je niz, ki se v enem dnevu razvije iz celice i-tega tipa. V naslednji vrstici

je niz s. V naslednji vrstici sta celi števili k in M, ločeni s presledkom. Nazadnje pride še vrstica z nizom p.

Omejitve: v spodnjih omejitvah zapis $|x|$ pomeni dolžino niza x.

- $1 \le a \le 26$; $0 \le k \le 300$; $2 \le M \le 10^9 + 20$;
- znaki t_i so vsi različni med sabo in vsi so male črke angleške abecede;
- $1 \le |s| \le 1000$; $1 \le |p| \le 300$; $0 \le |f(t_i)| \le 100$ za vse $i = 1, \ldots, a$;
- v vseh nizih $f(t_i)$, s in p nastopajo le črke iz množice $\{t_1, t_2, \ldots, t_a\}$, torej take, ki predstavljajo tipe celic, naštete v prvi vrstici vhodnih podatkov.

V 50 % testnih primerov bo veljalo tudi $|f^k(s)| \le 10^7$. V naslednjih 10 % primerov bo veljalo $|p| = 1$, v naslednjih 10 % primerov pa $|p| = 2$.

Izhodni podatki: izpiši rezultat, po katerem sprašuje besedilo naloge, torej število takih strnjenih podnizov niza $f^k(s)$, ki so anagrami p-ja; oz. natančneje povedano, izpiši ostanek po deljenju tega števila z M.

Primer vhoda:

```
vftb
b

ffvffv
fbv
tf
5 6
vbfb
```

Pripadajoči izhod:

```
3
```

Komentar: v tem testnem primeru nastopajo štirje tipi celic, označeni s črkami `v`, `f`, `t` in `b`. V enem dnevu iz celice tipa `v` nastane celica tipa `b`; iz celice tipa `f` nastane prazen niz (celica izgine brez sledu); iz celice tipa `t` nastane zaporedje šestih celic `ffvffv`; iz celice tipa `b` pa nastane zaporedje treh celic `fbv`. Niz $s =$ `tf` se v $k = 5$ dneh razvija takole:

$$\texttt{tf} \to \texttt{ffvffv} \to \texttt{bb} \to \texttt{fbvfbv} \to \texttt{fbvbfbvb} \to \texttt{fbvbfbvfbvbfbv}$$

Zadnji od teh nizov je $f^k(s)$ in ima kar 9 takih strnjenih podnizov, ki so anagrami niza $p =$ `vbfb` (edina njegova podniza dolžine $|p| = 4$, ki *nista* anagrama p-ja, sta `fbvf` in `vfbv`). Testni primer zahteva, da izpišemo ostanek po deljenju tega števila z $M = 6$; pravilni odgovor je torej 3 (ostanek po deljenju 9 s 6).

NALOGE ZA ŠOLSKO TEKMOVANJE

28. januarja 2022

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. Seštevanje ulomkov

V neki butalski družini imajo veliko otrok. Vsakič, ko ima kdo od otrok rojstni dan, slavljenec dobi torto, ostali otroci pa se postavijo v vrsto za njim. Slavljenec prereže torto na pol. Polovico bodisi pojé bodisi posadi v zemljo (da bo drugo leto zrasla nova torta), drugo polovico pa dá naslednjemu otroku v vrsti. Vsak naslednji otrok naredi enako: kos, ki ga prejme, prereže na pol, polovico bodisi pojé bodisi posadi, polovico poda naprej. Izjema je zadnji otrok v vrsti, ki vedno pojé cel kos, ki pride do njega.

Napiši program (ali podprogram oz. funkcijo), ki prebere podatke o več takšnih praznovanjih. V prvi vrstici dobi število praznovanj r (največ 1000) in število otrok n (največ 20), nato pa sledi r vrstic, ki opisujejo praznovanja; v vsaki od teh je niz n znakov, ki opisujejo, kaj naredi posamezni otrok pri tistem praznovanju (`'J'` = pojé, `'S'` = posadi). Podatke lahko bereš s standardnega vhoda ali pa iz datoteke `praznovanja.txt` (karkoli ti je lažje).

Vsaka torta tehta 1 kg. Tvoj program naj izpiše, koliko kilogramov torte, zaokroženo na najbližji celoštevilski kilogram, je na koncu zakopanih v zemlji. (Pol kilograma naj zaokroži navzgor; na primer, če je v zemlji točno 3,5 kg torte, naj izpiše 4.)

Vsi rezi so na vseh praznovanjih so bili popolnoma natančni; fizika pravi, da to ni mogoče, toda Butalci se tega niso nikoli naučili.

Primer vhodnih podatkov:

```
5 4
SJSJ
SSJJ
JSJJ
JJJJ
JJSJ
```

Pripadajoči izhod:

```
2
```

Komentar: na prvem praznovanju so zakopali $\frac{1}{2} + \frac{1}{8}$ kg, na drugem $\frac{1}{2} + \frac{1}{4}$ kg, na tretjem $\frac{1}{4}$ kg, na četrtem 0 kg in na petem $\frac{1}{8}$ kg. Skupaj je to $\frac{7}{4}$ kg, temu najbližje celo število pa je 2.

2. Slovar

Imamo podan seznam besed iz slovarja. V njem želimo poiskati besede, ki ustrezajo pogoju naslednje oblike: predpisana je dolžina besede in za nekatera mesta v besedi

je predpisano, katera črka mora stati na njih; za ostala mesta v besedi pa je vseeno, katere črke stojijo tam. Tak pogoj lahko opišemo z „vzorcem“ — to je niz, v katerem nastopajo črke in zvezdice, pri čemer zvezdice pomenijo, da je za tisto mesto v besedi vseeno, katera črka stoji tam.

Primer: vzorcu `*i*a` ustrezajo med drugim besede `miza`, `zima` in `riba`; ne ustrezajo pa mu na primer besede `mirta` (ker je predolga), `ica` (ker je prekratka) in `prva` (ker na drugem mestu nima `i`, ampak `r`).

(*a*) **Napiši program** (ali podprogram oz. funkcijo), ki kot vhodne podatke dobi seznam besed in vzorec ter izpiše tiste besede, ki ustrezajo temu vzorcu. Podrobnosti tega, v kakšni obliki dobi vhodne podatke, si izberi sam in jih v rešitvi tudi opiši.

(*b*) Recimo, da bi želeli izvesti takšna iskanja za veliko različnih vzorcev, pri čemer bi bil seznam besed ves čas enak. Zato je morda koristno ta seznam najprej nekako predelati ali preurediti, da bomo potem lahko čim hitreje iskali besede, ki ustrezajo različnim vzorcem. **Opiši**, kako bi to naredil in kakšen bi bil potem postopek iskanja besed, ki ustrezajo danemu vzorcu. (Pri tem delu naloge je dovolj opis postopka, ni treba pisati implementacije v kakšnem konkretnem programskem jeziku.)

Podnaloga (*a*) je vredna 13 točk, podnaloga (*b*) pa 7 točk. Pri obeh lahko predpostaviš, da v besedah in vzorcu nastopajo le male črke angleške abecede (od `a` do `z`), v vzorcu pa seveda poleg njih tudi zvezdice. Poleg tega lahko tudi predpostaviš, da gre res za besede iz slovarja kakšnega naravnega jezika (torej besede niso pretirano dolge, ne začnejo se vse na isto črko in podobno).

3. Genialno

Na pravokotni karirasti mreži, visoki h vrstic in široki w stolpcev, je na vsakem polju 0 ali več žetonov. Ko dodamo nov žeton na neko polje, dobimo za to določeno število točk, in sicer takole: iz polja, kamor smo postavili novi žeton, gremo v vse štiri možne smeri (gor, dol, levo, desno) tako daleč, dokler ne naletimo na prazno polje (táko z nič žetoni) ali na rob mreže. Dobimo toliko točk, kolikor je skupno število žetonov na takó obiskanih poljih (med slednja ne štejemo polja, na katero smo postavili novi žeton — glej primer spodaj).

Napiši program (ali podprogram oz. funkcijo), ki ugotovi, na katero polje moramo dati novi žeton, da bomo dobili največ točk. (Če obstaja več enako dobrih rešitev, je vseeno, katero izpiše.) Podrobnosti tega, kako dobiš podatke o mreži (velikost mreže in število žetonov na posameznem polju), si izberi sam in jih v rešitvi tudi opiši. Zaželeno je, da je tvoja rešitev učinkovita, tako da bo delovala hitro tudi za velike mreže (npr. s po nekaj tisoč stolpci in vrsticami).

Primer: recimo, da imamo mrežo, kot jo kaže naslednja slika.

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

Če postavimo novi žeton na drugo polje (z leve) v drugi vrstici (od zgoraj), dobimo $3 + 1 = 4$ točke. Če ga postavimo na drugo polje v tretji vrstici, dobimo $(2) + (2 + 1 + 2) = 7$ točk. Če ga postavimo na četrto polje v drugi vrstici, dobimo $(2) + (3 + 2) + (1 + 3) = 11$ točk, kar je pri tej mreži tudi najboljša možna poteza.

4. Parkirišče

Ob cesti je parkirišče, na katerem eden za drugim parkirajo tovornjaki. Časi prihodov in odhodov ter dolžine tovornjakov so znane vnaprej; vseh tovornjakov je n, pri čemer je i-ti izmed njih dolg d_i metrov, na parkirišče bo prišel ob času p_i in se na njem zadržal t_i časa, nato pa bo odpeljal. Tovornjaki niso nujno oštevilčeni v nobenem posebnem vrstnem redu.

Ko tovornjak zapusti parkirišče, se tisti, ki so bili parkirani za njim, v hipu pomaknejo naprej, tako da med parkiranimi tovornjaki ni lukenj (pa tudi ne med začetkom parkirišča in prvim tovornjakom). Za prihod ali odhod tovornjaka predpostavimo, da se zgodi v hipu, neskončno hitro. Mogoče je, da ob istem času pride in/ali odide več tovornjakov (v tem primeru moramo ravnati tako, kot da se najprej zgodijo odhodi, nato pa prihodi).

Opiši postopek (ali napiši podprogram oz. funkcijo, če ti je lažje), ki izračuna, kako dolgo parkirišče potrebujemo, da bo na njem vedno dovolj prostora za vse hkrati parkirane tovornjake. Za vhodne podatke bodo veljale naslednje omejitve: tovornjakov je največ milijon, dolžina posameznega tovornjaka je celo število od 1 do 1000, časi p_i in t_i pa so cela števila od 1 do 10^9 (recimo, da so podani v mikrosekundah od nekega izbranega začetnega trenutka naprej).

5. Barvanje stolpnic

Vzdolž ravne ulice so postavili n stolpnic samih različnih višin. Višino i-te stolpnice označimo s h_i. Rekli bomo, da sta stolpnici i in j *povezani*, če so vse stolpnice med njima nižje od njiju, torej če za vsak k z območja $i < k < j$ velja $h_k < h_i$ in tudi $h_k < h_j$. Arhitekti želijo poživiti sive stolpnice s svetlimi barvami fasad, pri tem pa nobeni dve povezani stolpnici ne smeta biti enake barve. **Opiši postopek** (ali napiši podprogram oz. funkcijo, če ti je lažje), ki določi barve stolpnic v skladu s tem pravilom in pri tem uporabi najmanjše možno število različnih barv. (Za rešitev, ki morda včasih uporabi več kot toliko barv, lahko še vedno dobiš delne točke.) V opisu postopka tudi dobro utemelji, zakaj tvoj postopek res vrača pravilne rezultate. Za predstavitev barv uporabi kar zaporedna naravna števila od 1 naprej. Kot vhodne podatke dobi tvoj postopek število stolpnic n in zaporedje višin $h_1, h_2, \ldots, h_n$. Tvoj postopek naj bo učinkovit, da bo deloval hitro tudi za velike n (npr. nekaj tisoč stolpnic).[2]

[2] Zanimiva naloga na temo takšnih stolpnic je tudi naslednja. Recimo, da imamo zaporedje n različno visokih stolpnic in ga barvamo od leve proti desni, pri čemer vsaki stolpnici dodelimo najmanjšo táko številko barve, ki je še nima nobena od z njo povezanih stolpnic. Koliko barv porabimo v najslabšem primeru (v odvisnosti od števila stolpnic n)?

NALOGE S CERC 2022

Društvo ACM Slovenija je letos sodelovalo tudi pri organizaciji srednjeevropskega študentskega tekmovanja v računalništvu (Central European Regional Contest — CERC), ki je potekalo 26. in 27. novembra 2022 na Fakulteti za računalništvo in informatiko v Ljubljani. Uradna besedila nalog in rešitev (v angleščini) so objavljena na spletni strani tekmovanja, `https://cerc.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 CERC. Tekmovanje poteka podobno kot na slovenskih študentskih tekmovanjih v programiranju (UPM), le da v samo enem kolu: tekmujejo ekipe s po tremi tekmovalci, vsaka ekipa ima en računalnik, svoje rešitve pa oddajajo na ocenjevalni strežnik, ki jih sproti testira in ocenjuje. Tekmovanje obsega en tekmovalni dan (letos je bil to 27. november), na katerem so tekmovalci reševali dvanajst nalog in imeli za to pet ur časa. (Dan prej je bilo tudi poskusno tekmovanje s tremi lažjimi nalogami; najdemo jih na koncu tega razdelka.) Podprti programski jeziki so bili C, C++, java, python in kotlin. Naloga velja za rešeno le, če program pravilno reši vse testne primere pri njej. Ekipe se razvrsti po številu rešenih nalog, tiste z enakim številom rešenih nalog pa po času; pri tem se za vsako uspešno rešeno nalogo sešteje čas (v minutah) od začetka tekmovanja do časa uspešne rešitve, prišteje pa se mu še po 20 minut za vsako pred tem oddano neuspešno rešitev te naloge.

Naloge na CERC so razvrščene po abecednem vrstnem redu naslovov (v angleščini). Približen vrstni red po težavnosti bi bil: L — igra; D — razgozdovanje; C — ozvezdja; G — požrešni predali; E — denormalizacija; K — spretno tabletno; B — kombinacijske ključavnice; I — pranje denarja; F — razlike; J — hipoteka; A — razbojniki; H — vrivanje.

A. Razbojniki

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

V nekem kraljestvu je n vasi in $n-1$ dvosmernih cest, po katerih lahko prebivalci pridejo od katerekoli vasi do katerekoli druge po poti, ki jo tvori ena ali več cest. Pri tem i-ta cesta povezuje vasi a_i in b_i, dolga pa je c_i enot.

Kralj dobiva vse več pritožb zaradi razbojnikov, ki napadajo trgovce, ki potujejo po cestah v kraljestvu. Svojemu svetovalcu je naročil, naj za rešitev te težave najame skupine zvestih robavsov, ki bodo delovali kot varnostne službe. Vsaka tako sklenjena pogodba o varovanju zagotavlja varnost vseh cest v „radiju“ oz. oddaljenosti r_j od vasi x_j, v kateri ima skupina svoje poveljstvo. Pogodba varuje posamezno cesto, če je le-ta del neke kvečjemu r_j enot dolge poti od x_j do neke druge vasi. Nekatere ceste lahko varuje tudi več pogodb in so zato še varnejše.

Napiši program, ki bo obdeloval podatke o novih pogodbah in odgovarjal na vprašanja o varnosti posameznih cest (torej koliko pogodb trenutno varuje tisto cesto).

Vhodni podatki. V prvi vrstici je število vasi n. Naslednjih $n-1$ vrstic opisuje ceste, ki povezujejo te vasi. Opis posamezne ceste je sestavljen iz celih števil a_i, b_i in c_i, ločenih s po enim presledkom; to predstavlja cesto dolžine c_i med vasema a_i in b_i. Vasi so oštevilčene od 1 do n.

V naslednji vrstici je število poizvedb q. Sledi q vrstic, ki opisujejo vsaka po eno poizvedbo. Poizvedba, ki predstavlja sklenitev nove pogodbe, se začne z znakom „+“, ki mu sledita vas s poveljstvom x_j ter radij r_j. Poizvedba, ki sprašuje o varnosti določene ceste, se začne z znakom „?“, ki mu sledi številka ceste y_j. Ceste so oštevilčene od 1 do $n-1$ v takem vrstnem redu, kot so navedene v vhodnih podatkih.

Omejitve vhodnih podatkov:

- $1 \leq n \leq 10^5$, $1 \leq q \leq 10^5$;
- $0 \leq c_i \leq 10^9$, $0 \leq r_j \leq 10^9$.

Izhodni podatki. Poizvedbe obdelaj po vrsti in za vsako poizvedbo tipa „?“ izpiši eno vrstico s številom pogodb, ki v tistem trenutku varujejo cesto y_j.

Primer vhoda:

```
7
1 2 4
4 2 7
5 1 3
3 6 4
1 6 9
2 7 1
7
+ 2 6
? 3
? 1
+ 6 14
? 1
? 2
? 3
```

Pripadajoči izhod:

```
0
1
2
0
1
```

B. Kombinacijske ključavnice
(*Omejitev časa: 2 s. Omejitev pomnilnika: 256 MB.*)

Alica in Bob se igrata s kombinacijskimi ključavnicami. Vsak od njiju ima kombinacijsko ključavnico, sestavljeno iz n vrtljivih obročkov, na vsakem od katerih so vgravirane številke od 0 do 9. Njun prijatelj Cene nima ključavnice in si je izmislil igro, s katero ju bo zaposlil. Preverjal bo, ali se istoležne številke njunih ključavnic ujemajo, in trenutno stanje opisoval z *vzorcem razlik*, nizom S. Pri tem je j-ti znak niza S bodisi „=“ (enačaj) bodisi „.“ (pika) in pove, ali se j-ta številka Aličine in Bobove ključavnice ujemata ali ne.

Cene bo vodil igro, Alica in Bob pa bosta izmenično delala poteze (najprej Alica). Pri vsaki potezi mora trenutni igralec spremeniti eno številko svoje kombinacijske ključavnice. Ker gleda Cene le vzorce razlik, se mora ta vzorec spremeniti, sicer poteza ni veljavna. Cene je tudi precej praznoveren in je pripravil seznam vzorcev, ki se med igro ne smejo pojaviti. Njegova glavna naloga je skrbeti za upoštevanje pravila, da se tekom igre noben vzorec razlike ne sme ponoviti. Igralec, ki ne more narediti veljavne poteze, izgubi.

Napiši program, ki ugotovi, kateri igralec bo zmagal, če oba igrata optimalno.

Vhodni podatki. V prvi vrstici je število testnih primerov T. Vsak testni primer se začne z vrstico, ki vsebuje celi števili n in c, ločeni s presledkom. Sledita dve

vrstici, ki opisujeta začetno kombinacijo Aličine in Bobove ključavnice (vsaka od teh kombinacij je niz n števk). Sledi c vrstic, ki opisujejo Cenetove praznoverne vzorce p_i. V tem praznovernem seznamu ni duplikatov, zagotovljeno pa je tudi, da na njem ni vzorca razlik med začetnima kombinacijama obeh ključavnic.

Omejitve vhodnih podatkov:

- $1 \leq T \leq 20$;
- $1 \leq n \leq 10$, $0 \leq c \leq 1000$.

Izhodni podatki. Za vsak testni primer izpiši eno vrstico z imenom zmagovalca („`Alice`“ ali „`Bob`“, brez narekovajev).

Primer vhoda:

```
3
2 2
12
89
=.
==
3 1
204
101
.==
3 2
000
000
...
==.
```

Pripadajoči izhod:

```
Alice
Bob
Bob
```

Komentar: pri prvem primeru Alica v prvi potezi nima druge možnosti, kot da spremeni drugo števko v eno od števil od 2 do 9. Katerakoli druga poteza je neveljavna, ker bodisi ne spremeni vzorca razlik bodisi bi pripeljala do enega od praznovernih vzorcev. Bob nato nima veljavne poteze, zato Alica zmaga.

C. Ozvezdja

(*Omejitev časa: 10 s.* *Omejitev pomnilnika: 512 MB.*)

Astrologi so se resno in znanstveno poglobili v napovedi svojih zodiakalnih horoskopov in se zavedeli, da ne daje njihova metodologija nič boljšega vpogleda v prihodnost kot golo naključje. Namesto da bi se zazrli vase, so za svoj neuspeh pri napovedovanju prihodnosti okrivili zvezde in zgodovinsko razporeditev zvezd v ozvezdja. Zdaj preizkušajo nov način tvorjenja ozvezdij, ki bo prerodil njihove zmožnosti videnja prihodnosti.

Potrebujejo pa tvojo pomoč pri implementaciji sistema za iterativno tvorjenje ozvezdij. Na začetku predstavlja vsaka zvezda svoje ozvezdje. V vsakem koraku moraš združiti dve ozvezdji v eno, in sicer tisti dve ozvezdji, ki sta si najbližji. Razdalja med ozvezdjema A in B je definirana kot povprečje kvadratov evklidskih razdalj med vsemi pari zvezd, v katerih je po ena zvezda iz vsakega ozvezdja:

$$d(A, B) = \frac{1}{|A|\,|B|} \sum_{a \in A} \sum_{b \in B} ||a - b||^2.$$

Če je več parov ozvezdij enako oddaljenih eno od drugega, združi najprej starejša ozvezdja. Ko torej primerjaš dva para ozvezdij za potrebe združitve, ju najprej primerjaj po razdalji med ozvezdjema v posameznem paru; če je ta razdalja pri obeh parih povsem enaka, ju primerjaj po starosti starejšega izmed obeh ozvezdij v paru; če se tudi pri tem ujemata, pa ju primerjaj po starosti mlajšega ozvezdja.

Starost ozvezdja se šteje od časa, ko je nastalo z združitvijo dveh manjših ozvezdij, če pa gre za ozvezdje z eno samo zvezdo, je njegova starost enaka starosti te zvezde. V vhodnih podatkih so zvezde naštete od najstarejše do najmlajše.

Vhodni podatki. V prvi vrstici je število zvezd n. Sledi n vrstic, od katerih i-ta vsebuje koordinati i-te zvezde, x_i in y_i, ločeni s presledkom.

Omejitve vhodnih podatkov:

- $2 \leq n \leq 2000$;
- $|x_i| \leq 1000$ in $|y_i| \leq 1000$ za vse $i = 1, \ldots, n$;
- vsi pari (x_i, y_i) so različni, saj fizično ni mogoče, da bi bili dve zvezdi hkrati na istem mestu.

Izhodni podatki. Po vsakem koraku opisanega postopka za tvorbo ozvezdij izpiši velikost pravkar ustvarjenega ozvezdja. Izpisati moraš $n - 1$ vrstic.

Primer vhoda:

```
3
0 0
-1 0
10 10
```

Pripadajoči izhod:

```
2
3
```

Še en primer:

```
4
0 0
0 -1
0 1
0 2
```

Pripadajoči izhod:

```
2
2
4
```

In še eden:

```
4
0 0
0 1
0 -1
0 2
```

Pripadajoči izhod:

```
2
3
4
```

D. Razgozdovanje
(*Omejitev časa: 2 s. Omejitev pomnilnika: 256 MB.*)

S svojega zemljišča želiš odstraniti veliko drevo, ki pa je pretežko, da bi ga lahko odnesel v enem kosu. Na koliko kosov ga moraš razžagati, če je največja masa, ki jo lahko neseš, w enot?

Drevo ima eno samo deblo, ki je vsajeno v tla in se potem lahko razveji na več vej. Te veje se lahko razvejijo naprej in tako dalje. Vsak člen drevesa je tako zvezna gmota lesa, ki se lahko (ni pa nujno) razveji na več vej.

Reze lahko narediš kjerkoli na drevesu, na začetku ali koncu kateregakoli člena ali kjerkoli vmes. Razvejitev lahko obravnavaš kot neskončno majhen del drevesa, torej lahko prežagaš, tik preden se člen razveji ali pa tik zatem, pa bo masa osnovnega člena v obeh primerih enaka; pač pa bo razlika v tem, ali bodo veje, ki izhajajo iz njega, odžagane vse kot en kos ali pa bo odžagana le ena veja sama zase.

Vhodni podatki. V prvi vrstici je tvoja nosilnost w. V naslednji vrstici je opis prvega člena drevesa, namreč njegovega debla.

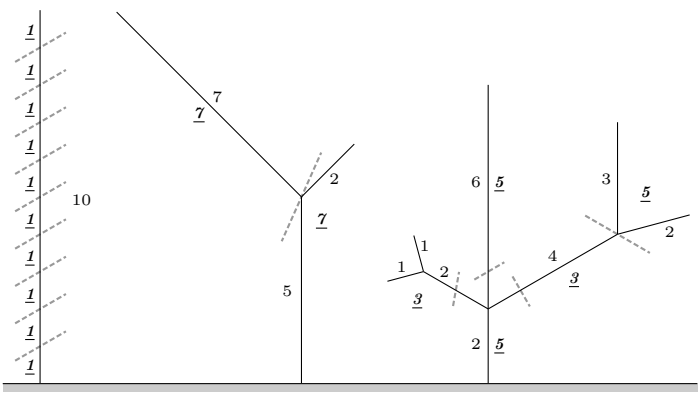
Opis posameznega člena drevesa je definiran rekurzivno. V prvi vrstici sta dve števili: m (masa tega člena) in k (število vej, na katere se ta člen na koncu razveji). Temu sledi n opisov členov, ki po vrsti opisujejo vsako od vej.

Omejitve vhodnih podatkov:

- $1 \leq w \leq 10^9$, $1 \leq m \leq 10^9$;
- $0 \leq k \leq 10^5$;
- skupna masa vseh členov bo največ 10^9;
- skupno število členov bo največ 10^5.

Izhodni podatki. Izpiši eno samo število, namreč najmanjše število kosov, na katere je treba razžagati drevo.

Trije primeri vhoda:	Pripadajoči izhodi:
1 10 0	10
7 5 2 7 0 2 0	2
5 2 3 2 2 1 0 1 0 6 0 4 2 3 0 2 0	5



Slika kaže možne razreze za tri primere na levi. Polne črte predstavljajo člene (debla in veje), števila v pokončnem tisku ob njih pa njihove mase. Črtkane črte predstavljajo reze, podčrtana števila v ležečem tisku pa mase kosov, na katere je bilo drevo s temi rezi razžagano.

E. Denormalizacija

(*Omejitev časa: 5 s. Omejitev pomnilnika: 256 MB.*)

Dr. Brodnik je pripravil seznam n celih števil, $A = (a_1, a_2, \ldots, a_n)$. Nihče ne ve natančno, kaj ta števila pomenijo, dobro pa je znano naslednje:

- $1 \leq a_i \leq 10\,000$ za vse $i = 1, 2, \ldots, n$; in
- največji skupni delitelj števil $a_1, \ldots, a_n$ je 1.

Dr. Hočevar se je namenil svojemu kolegu storiti uslugo in je seznam normaliziral, misleč, da predstavlja neki vektor v n-razsežnem realnem vektorskem prostoru. Izračunal je torej število

$$d = \sqrt{\sum_{i=1}^{n} a_i^2} = \sqrt{a_1^2 + a_2^2 + \ldots + a_n^2}$$

in potem zamenjal seznam dr. Brodnika z $(a_1/d, a_2/d, \ldots, a_n/d)$. Pri zapisu je ta števila tudi zaokrožil na 12 decimalnih mest. Tako normaliziran in zaokrožen

seznam označimo z $X = (x_1, x_2, \ldots, x_n)$. Čez čas se je zavedel, da je bilo to narobe, in bi zdaj rad prišel nazaj do prvotnega seznama A; seveda pa rezervne kopije tega seznama ni shranil. Ker je dr. Hočevar trenutno preveč zaposlen s pomembnejšimi opravili, bi mu prišla tvoja pomoč zelo prav.

Ker se je nekaj podatkov zaradi zaokrožanja izgubilo, bo zadovoljen s katerimkoli rekonstruiranim seznamom $R = (r_1, r_2, \ldots, r_n)$, pri katerem bi se po normalizaciji vsak element razlikoval od istoležnega elementa seznama X za kvečjemu 10^{-6}.

Vhodni podatki. V prvi vrstici je celo število n, ki pomeni dolžino seznama X. V i-ti od naslednjih n vrstic je decimalno število x_i, ki ima za decimalno piko natanko 12 števk. Zagotovljeno je, da so vhodni podatki veljavni, torej da so res nastali na opisani način iz seznama celih števil s prej omenjenimi lastnostmi.

Omejitve vhodnih podatkov:

- $2 \leq n \leq 10\,000$;
- $0 < x_i < 1$ za vse $i = 1, 2, \ldots, n$.

Izhodni podatki. Izpiši rekonstruirani seznam n celih števil $r_1, r_2, \ldots, r_n$, vsako v svoji vrstici. Izpišeš lahko katerokoli rešitev, ki je sprejemljiva glede na zgoraj opisane pogoje. Za vsak $i = 1, 2, \ldots, n$ mora veljati $1 \leq r_i \leq 10\,000$, poleg tega pa mora biti največji skupni delitelj števil $r_1, \ldots, r_n$ enak 1.

Primer vhoda:	Pripadajoči izhod:
`6`	`5`
`0.137516331034`	`6`
`0.165019597241`	`10`
`0.275032662068`	`15`
`0.412548993102`	`30`
`0.825097986204`	`6`
`0.165019597241`	

F. Razlike

(*Omejitev časa: 2 s. Omejitev pomnilnika: 256 MB.*)

Dan je seznam n nizov $s_1, \ldots, s_n$. Vsi nizi so dolgi po m znakov, ti znaki pa so le velike črke `A`, `B`, `C` in `D`. Definirajmo *razdaljo* med dvema nizoma x in y kot število indeksov j, kjer imata tadva niza različna znaka ($x[j] \neq y[j]$). Znano je, da je med nizi v seznamu natanko en tak poseben niz, ki je na razdalji točno k od vseh ostalih nizov v seznamu. (Mogoče je sicer, da je tudi še pri kakšnih drugih parih nizov oddaljen en niz od drugega za točno k.) Ker imamo pri iskanju tega posebnega niza težave, nam pomagaj in **napiši program**, ki ga poišče.

Vhodni podatki. V prvi vrstici so cela števila n, m in k, ločena s po enim presledkom. Sledi n vrstic, od katerih i-ta vsebuje niz s_i.

Omejitve vhodnih podatkov:

- $2 \leq n \leq 10^5$ in $2 \leq m \leq 10^5$;
- $1 \leq k \leq m$;
- $n \cdot m \leq 2 \cdot 10^7$.

Izhodni podatki. Izpiši indeks i posebnega niza. Nizi so oštevilčeni od 1 do n v takem vrstnem redu, v kakršnem so podani v vhodnih podatkih.

Primer vhoda:

```
5 10 2
DCDDDCCADA
ACADDCCADA
DBADDCCBDC
DBADDCCADA
ABADDCCADC
```

Pripadajoči izhod:

```
4
```

Še en primer:

```
4 6 5
AABAAA
BAABBB
ABAAAA
ABBAAB
```

Pripadajoči izhod:

```
2
```

G. Požrešni predali

(*Omejitev časa: 2 s. Omejitev pomnilnika: 256 MB.*)

Janko ima na mizi n pravokotnih zvezkov; i-ti zvezek ima stranice dolžine a_i in b_i. Poleg mize stoji predalnik z n predali, ki so tudi pravokotne oblike, vendar so lahko različnih velikosti: j-ti predal je širok x_j in globok y_j enot. Janko hoče shraniti vsak zvezek v svoj predal. Zvezke lahko sicer vrti, vendar jih bo dal v predal tako, da bodo stranice zvezka poravnane s stranicami predala. Zvezek gre v predal, če dolžina nobene njegove stranice ne presega dolžine tiste stranice predala, s katero je poravnana.

Janko se je odločil, da bo razporedil zvezke v predale po naslednjem postopku. Za vsak zvezek bo preštel, v koliko predalov gre; podobno bo tudi za vsak predal preštel zvezke, ki gredo vanj. Potem bo izbral tisti predmet (zvezek ali predal), pri katerem je teh možnosti najmanj; če je predmetov z minimalnim številom možnosti več, bo med njimi enega izbral naključno (pri čemer imajo vsi enako verjetnost, da bodo izbrani). Ta predmet bo potem dodelil eni od možnosti, ki jo bo tudi izbral naključno (in tako, da imajo vse možnosti enako verjetnost, da bodo izbrane). (Če nima izbrani predmet nobene možnosti, se postopek konča in Janko ni uspel najti rešitve.) Z drugimi besedami, če je bil izbrani predmet predal, bo vanj dal naključno izbran zvezek, ki gre v ta predal; če pa je bil izbrani predmet zvezek, ga bo dal v naključno izbran predal, v katerega gre ta zvezek. Uporabljeni zvezek in predal bo nato v mislih pobrisal in nadaljeval postopek s preostalimi zvezki in predali, vse dokler niso vsi zvezki razporejeni v predale.

Metka je slišala za Jankovo zamisel o razporejanju zvezkov v predale. Prepričana je, da njegov postopek ni dober in da mu lahko spodleti. Pomagaj ji in **napiši program**, ki prebere število zvezkov in predalov n ter izpiše seznam zvezkov in seznam predalov, na katerih Jankov požrešni postopek ne najde nujno veljavnega razporeda zvezkov v predale, četudi tak razpored obstaja.

Vhodni podatki. Prva in edina vrstica vsebuje le celo število n, to je število zvezkov in tudi število predalov. Velja $150 \leq n \leq 250$.

Izhodni podatki. Najprej izpiši n vrstic, od katerih i-ta vsebuje dolžini stranic i-tega zvezka, a_i in b_i, ločeni s presledkom. Nato izpiši prazno vrstico, potem pa še n vrstic, od katerih j-ta vsebuje širino in globino j-tega predala, x_j in y_j, ločeni s presledkom. Vse te dimenzije morajo biti cela števila od vključno 1 do vključno 1000.

Ocenjevanje. Da ocenimo izpis tvojega programa, bomo na tvojih podatkih (velikostih zvezkov in predalov) pognali Jankovo naključno požrešno metodo. Obstajati mora veljaven razpored zvezkov v predale, sicer bo tvoj odgovor štel za napačnega. Tvojo rešitev bomo preizkusili na dvajsetih testnih primerih in sprejeta bo le, če bo Jankovi metodi spodletelo v vseh primerih. Za vsak testni primer bomo Jankovo metodo pognali enkrat s fiksnim naključnim semenom.

Primer vhoda:	Možen pripadajoči izhod:

`1`	`4 3`
	
	`2 6`

Še en primer:	Možen pripadajoči izhod:

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

Komentar. Gornji primeri vhodov in izhodov so neveljavni; vhodi zato, ker ne držijo omejitve $150 \leq n$.

Pri prvem primeru izhoda edini zvezek ne gre v edini predal, zato veljaven razpored zvezkov v predale ne obstaja.

Pri drugem primeru izhoda Jankov postopek vedno uspešno razporedi vse zvezke v predale. Najprej bi izbral zadnji zvezek (6×1) ali prvi predal (2×7) in dal ta zvezek v ta predal, ker imata oba le po eno možnost. Potem gresta oba preostala zvezka v oba preostala predala, zato je dober vsak razpored.

H. Vrivanje

(*Omejitev časa: 1 s. Omejitev pomnilnika: 256 MB.*)

Dani so nizi s, t in p. Dolžino niza bomo označili z navpičnimi črtami: $|s|$ je torej dolžina niza s in tako naprej. Če *vrinemo* t v s na položaju k (pri čemer je $0 \leq k \leq |s|$), dobimo nov niz, ki ga tvori najprej prvih k znakov niza s, nato celoten niz t in končno preostalih $|s| - k$ znakov niza s. Radi bi izbrali k tako, da se bo v dobljenem novem nizu čim večkrat kot (strnjen) podniz pojavil niz p.

Če tako na primer vrinemo $t =$ `aba` v $s =$ `ab` na položaju $k = 0$, nastane niz `abaab`; pri $k = 1$ nastane `aabab`; pri $k = 2$ pa nastane `ababa`. Če nas zanimajo pojavitve niza $p =$ `aba`, je za vrivanje t v s najboljši položaj $k = 2$, pri katerem dobimo dve pojavitvi p-ja: `ababa` in `ababa` (kot vidimo iz tega primera, se lahko pojavitve p-ja tudi prekrivajo). Po drugi strani pa, če bi nas zanimale pojavitve

niza $p =$ `aa`, bi bili najboljši možnosti $k = 0$ in $k = 1$, pri katerih dobimo eno pojavitev p-ja, medtem ko pri $k = 2$ ne dobimo nobene.

Vhodni podatki. V prvi vrstici je niz s, v drugi niz t in v tretji niz p.

Omejitve vhodnih podatkov:

- $1 \leq |s| \leq 10^5$, $1 \leq |t| \leq 10^5$, $1 \leq |p| \leq 10^5$;
- vsi nizi so sestavljeni le iz malih črk angleške abecede.

Izhodni podatki. Izpiši eno samo vrstico, vanjo pa naslednja štiri cela števila, ločena s po enim presledkom:

1. največje število pojavitev p kot podniza, ki jih lahko dobimo po vrivanju t-ja v s na položaju k, če primerno izberemo k;
2. število različnih k-jev (izmed $k = 0, 1, \ldots, |s|$), pri katerih dosežemo to največje število pojavitev p-ja;
3. najmanjšo vrednost k, pri kateri je doseženo to največje število pojavitev p-ja;
4. največjo vrednost k, pri kateri je doseženo to največje število pojavitev p-ja.

Primer vhoda:

```
ab
aba
aba
```

Pripadajoči izhod:

```
2 1 2 2
```

Še en primer:

```
abaab
aba
ababa
```

Pripadajoči izhod:

```
1 3 1 5
```

In še eden:

```
eeoeo
eoe
eeo
```

Pripadajoči izhod:

```
2 3 1 4
```

Opomba: prvi od teh treh primerov je tisti, o katerem smo govorili že zgoraj v besedilu naloge.

I. Pranje denarja

(*Omejitev časa: 2 s. Omejitev pomnilnika: 256 MB.*)

Predstavljajmo si podjetje A, ki je letos ustvarilo 100 evrov dobička. Lastnika podjetja sta Ivan s 52,8-odstotnim lastniškim deležem in Robi s 47,2-odstotnim. Dobiček se seveda deli sorazmerno z lastniškim deležem, zato dobi Ivan 52,8 evra, Robi pa 47,2.

Na prejeti dobiček bosta morala plačati davek, čemur pa bi se rada izognila, če je le mogoče. Žal je lastniška struktura njunega podjetja preveč preprosta in preveč zlahka se dá ugotoviti, koliko dobička je prejel kdo od njiju.

Za naslednje leto pripravita načrt. Ustanovita slamnato podjetje B in spremenita lastniška deleža. Ivan je zdaj lastnik le 1 % podjetja A, Robi le 2 %, podjetje B je

lastnik 49 % podjetja A, poleg tega pa je podjetje A lastnik 48 % samega sebe. Podobno lastniško strukturo ima tudi podjetje B: 70 % tega podjetja pripada B-ju samemu, 25 % A-ju, 3 % Ivanu in 2 % Robiju.

Naivno gledano imata Ivan in Robi zelo majhne lastniške deleže. Toda v resnici nas zanimajo lastniški deleži *končnih upravičenih lastnikov*, torej ljudi, ki bodo na koncu zares pobrali dobiček, kar sta v našem primeru Ivan in Robi. Radi bi določili njuna končna lastniška deleža (ki sta, kot se izkaže, približno taka, kot sta bila pred ustanovitvijo B-ja).

Končne lastniške deleže lahko določimo na naslednji način. Recimo, da ima podjetje A 100 evrov dobička, podjetje B pa 0 evrov. Dobiček se izplača neposrednim lastnikom sorazmerno z njihovim lastniškim deležem. Ker sta A in B delna lastnika samih sebe, dobita tudi onadva delež dobička. Da določimo končni delež končnih upravičenih lastnikov, moramo ta postopek ponavljati — tisto, kar od dobička dobita A in B, se spet izplača njunim lastnikom, pri čemer dobita Ivan in Robi vsak svoj delež, prav tako pa tudi A in B. To se ponavlja v neskončnost, dokler ni (teoretično, po neskončno mnogo korakih) ves denar izplačan končnim upravičenim lastnikom; delež (glede na začetni dobiček podjetja A) skupnih zneskov, ki sta jih prejela Ivan in Robi, pa je po definiciji enak njunemu končnemu lastniškemu deležu v podjetju A.

Napiši program, ki za dano strukturo podjetij določi deleže končnih upravičenih lastnikov. Vendar pa podjetja nimajo čisto poljubne strukture lastništva, pač pa so razdeljena na gospodarske panoge. Podjetja znotraj iste panoge lahko tvorijo „hobotnice“, v katerih je lahko lastniška struktura poljubna, kar pa ne velja za podjetja iz različnih panog. Če pripadata podjetji P in Q različnima panogama, se ne more zgoditi, da

- bi bilo podjetje P lastnik (lahko posredni) nekega deleža v podjetju Q in
- bi bilo podjetje Q lastnik (lahko posredni) nekega deleža v podjetju P.

Ena ali nobena od teh dveh trditev lahko drži, obe hkrati pa ne.

Vhodni podatki. V prvi vrstici sta celi števili c (število podjetij) in p (število ljudi), ločeni s presledkom. Sledi c vrstic, od katerih i-ta opisuje i-to podjetje. V tej vrstici je najprej celo število k_i (število lastnikov podjetja i), nato pa k_i zapisov oblike $o_{ij}{:}p_{ij}$, kjer je o_{ij} oznaka j-tega lastnika (lahko je človek ali podjetje), p_{ij} pa je njegov lastniški delež (v odstotkih). Deleži bodo podani na natanko eno decimalko.

Omejitve vhodnih podatkov:

- $1 \le c \le 1000$ in $1 \le p \le 1000$;
- $1 \le \sum_{i=1}^{n} k_i \le 10^4$;
- o_{ij} ima dve možni obliki: `P`x oz. `C`y, ki povesta, da je lastnik x-ti človek oz. y-to podjetje. Pri tem gotovo velja $1 \le x \le p$ oz. $1 \le y \le c$.
- $0 < p_{ij} \le 100$ in $\sum_{j=1}^{k_i} p_{ij} = 100$;
- pri vsakem i so oznake $\{o_{i1}, \ldots, o_{i,k_i}\}$ vse različne, torej je vsak lastnik naveden le enkrat;
- število podjetij v posamezni gospodarski panogi je manj kot 10;
- vsako podjetje ima vsaj enega končnega upravičenega lastnika. Tako na primer ni dovoljena shema, v kateri bi bilo podjetje A lastnik 100 % podjetja B, slednje pa bi bilo lastnik 100 % podjetja A.

Izhodni podatki. Izpiši končne lastniške deleže vseh ljudi v vseh podjetjih. Pri tem mora i-ta vrstica vsebovati deleže vseh ljudi v i-tem podjetju, tudi ljudi z ničelnim deležem. Deleži naj bodo podani kot števila med 0 in 1. Deleži v vsaki vrstici naj bodo ločeni s po enim presledkom. Odgovor bo veljal za pravilnega, če bo njegova absolutna ali relativna napaka (odstopanje od uradne rešitve) manj kot 10^{-4}.

Štirje primeri vhoda:	Možni pripadajoči izhodi:

`2 2`
`2 P1:50.1 P2:49.9`
`2 P1:23.4 P2:76.6`
	`0.501000 0.499000`
	`0.234000 0.766000`

`2 2`
`2 P1:50.0 P2:50.0`
`3 P1:20.0 P2:30.0 C1:50.0`
	`0.500000 0.500000`
	`0.450000 0.550000`

`2 2`
`4 P1:1.0 P2:2.0 C2:49.0 C1:48.0`
`4 C2:70.0 C1:25.0 P1:3.0 P2:2.0`
	`0.528358 0.471642`
	`0.540299 0.459701`

`3 2`
`5 P1:1.0 P2:2.0 C2:49.0 C1:38.0 C3:10.0`
`4 C2:70.0 C1:25.0 P1:3.0 P2:2.0`
`2 P1:20.0 P2:80.0`
	`0.373228 0.626772`
	`0.411024 0.588976`
	`0.2 0.8`

J. Hipoteka

(*Omejitev časa: 3 s. Omejitev pomnilnika: 512 MB.*)

Andrej je tipičen sodoben študent, ki sanja o tem, da bi si nekega dne kupil hišo. Ker nakup nepremičnine ni mačji kašelj, načrtuje svoje življenje in poskuša ugotoviti, kako in kdaj si jo bo pravzaprav lahko privoščil. Za nakup hiše namerava vzeti posojilo, ki ga bo več mesecev odplačeval po obrokih. V i-tem od naslednjih n mesecev bo zaslužil a_i enot denarja, ki ga lahko porabi za odplačevanje posojila (drugi izdatki so bili v tem že upoštevani, tako da je lahko vrednost a_i celo negtivna). Zdaj si ogleduje seznam raznih nepremičnin in hipotekarnih posojil ter poskuša ugotoviti, katere od njih si lahko privošči.

Recimo, da vzame posojilo, pri katerem bo moral plačevati po x enot denarja na mesec skozi k mesecev, od meseca s do meseca $s+k-1$. V vsakem od teh mesecev mora biti zmožen plačati x enot denarja. Če mu v mesecu s ostane še kaj denarja, torej če je $a_s > x$, lahko ta preostanek prihrani in ga uporabi pri plačevanju obrokov v kasnejših mesecih; enako velja tudi za morebitne viške denarja v mesecih od $s+1$ do $s+k-1$. Ne sme pa računati na to, da bo privarčeval kaj denarja že pred mesecem s, ne glede na svoj zaslužek v tistih mesecih; takrat bo ves denar porabil za najemnino in opečen kruh z avokadom.

Dobil boš seznam Andrejevih prihodkov v naslednjih n mesecih in seznam m različnih časovnih intervalov. Pri tem je i-ti interval opisan s številoma s_i in k_i, ki povesta, da se odplačevanje posojila začne v mesecu s_i in traja k_i mesecev, torej se zadnji obrok odplača v mesecu $s_i + k_i - 1$. Za vsakega od teh časovnih intervalov izračunaj največji možni mesečni obrok, ki si ga Andrej še lahko privošči.

Vhodni podatki. V prvi vrstici sta dve celi števili, n (število mesecev) in m (število različnih časovnih intervalov), ločeni s presledkom. V drugi vrstici je n celih števil, $a_1, a_2, \ldots, a_n$ (ločenih s po enim presledkom), ki podajajo Andrejev zaslužek v

posameznih mesecih. Sledi še m vrstic, ki opisujejo časovne intervale; i-ta od teh vrstic vsebuje celi števili s_i in k_i, ločeni s presledkom.

Omejitve vhodnih podatkov:

- $1 \leq n \leq 2 \cdot 10^5$, $1 \leq m \leq 2 \cdot 10^5$;
- $-10^9 \leq a_i \leq 10^9$ za $i = 1, 2, \ldots, n$;
- za vsak $i = 1, \ldots, m$ velja $1 \leq s_i \leq n$, $1 \leq k_i$ in $s_i + k_i - 1 \leq n$.

Izhodni podatki. Izpiši m vrstic, po eno za vsak časovni interval. V i-to od teh vrstic izpiši največji celoštevilski znesek mesečnega obroka, ki si ga Andrej lahko privošči pri posojilu, čigar obdobje odplačevanja predstavlja i-ti časovni interval. Če je to število strogo manjše od 0, izpiši „`stay with parents`“ (brez narekovajev).

Primer vhoda:

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

Pripadajoči izhod:

```
4
3
8
stay with parents
0
```

Komentar. Pri prvem intervalu je največji mesečni obrok, ki si ga Andrej lahko privošči, 4 enote; pri mesečnem obroku 5 enot bi mu zmanjkalo denarja za zadnji obrok. Negativni zaslužek v šestem mesecu pomeni, da si Andrej pri četrtem intervalu ne more privoščiti nikakršnega posojila, ne glede na višino obroka.

K. Spretno tabletno

(*Omejitev časa: 1 s. Omejitev pomnilnika: 256 MB.*)

Neimenovani glavni junak te naloge je po elektronski pošti prejel več neverjetnih ponudb za čudodelne tablete, ki bodo izboljšale njegove umske in še vsakovrstne druge sposobnosti. Po pazljivi preučitvi vseh ponudb in njihovih stranskih učinkov se je odločil naročiti dve vrsti tablet; recimo jima A in B. Tableto vrste A mora vzeti vsakih a dni, tableto vrste B pa vsakih b dni. Tega se bo vestno držal naslednjih n dni.

Bolj formalno rečeno: v naslednjih n dneh ne sme biti nobenih a zaporednih dni, ko ne bi vzel nobene tablete vrste A, in tudi ne nobenih b zaporednih dni, ko ne bi vzel nobene tablete vrste B. Je pa pri tem še hakeljc: oboje tablete so zelo močne in se jih ne sme vzeti na isti dan, sicer bi nastopili strašni stranski učinki. Kolikšno je najmanjše število tablet, ki jih mora vzeti, da bo lahko upošteval vse te omejitve?

Vhodni podatki. Na vhodu dobiš eno vrstico, v njej pa tri cela števila, a, b in n, ločena s po enim presledkom.

Omejitve vhodnih podatkov:

- $2 \leq n \leq 10^6$;
- $2 \leq a \leq n$ in $2 \leq b \leq n$.

Izhodni podatki. Izpiši eno samo celo število — najmanjše število tablet, ki jih je treba vzeti. Ni težko dokazati, da pri danih omejitvah vhodnih podatkov rešitev vedno obstaja.

Trije primeri vhoda:	Pripadajoči izhodi:
`2 3 8`	`6`
`2 3 11`	`9`
`3 7 100`	`48`

Komentar. Pri prvem primeru lahko vzamemo tableto A na 2., 4., 5. in 7. dan, tableto B pa 3. in 6. dan, kar lahko predstavimo tudi z zaporedjem ␣`ABAABA`. Pri drugem primeru je najboljši pristop zaporedje ␣`ABAABAABA`␣, pri katerem je treba vzeti devet tablet.

L. Igra

(*Omejitev časa: 1 s. Omejitev pomnilnika: 256 MB.*)

Vladimir je najbolj osamljen otrok v svoji soseski. Nihče se noče igrati z njim. Njegovi starši so mu v želji, da ga razvedrijo, kupili igro s kartami, ki se imenuje preprosto *igra*. To je sicer igra za največ pet igralcev, ki pa se jo lahko igra tudi v enoigralskem načinu.

V kompletu je 98 *navadnih* igralnih kart, ki so označene s števili 2, 3, ..., 99, in štiri posebne *smerne* karte, od katerih sta dve označeni z „1 ↑“, dve pa s „100 ↓“.

V začetni fazi igre se navadne karte premeša in položi kot kup na mizo s številkami navzdol — to bo *kup za vlečenje*. Štiri smerne karte pa se položi na mizo v stolpec, pri čemer morata biti karti s številko 1 na vrhu. Desno od vsake smerne karte mora biti dovolj prostora, kajti tja se bo med igro polagalo navadne karte. Karta s številko 1 začenja *naraščajočo vrstico*, karta s številko 100 pa *padajočo vrstico*. V enoigralskem načinu pobere igralec osem kart z vrha kupa, eno po eno, in jih jemlje v svojo roko.

Po tej začetni fazi se igra začne zares. V vsaki potezi mora igralec položiti dve izmed kart, ki jih ima v roki, in sicer po naslednjih pravilih:

- Karto sme položiti na konec naraščajoče vrstice, če je višja od dosedanje zadnje (najbolj desne) karte v tej vrstici.
- Karto sme položiti na konec padajoče vrstice, če je nižja od dosedanje zadnje (najbolj desne) karte v tej vrstici.
- Karto z nižjo številko sme položiti na konec naraščajoče vrstice, ali pa karto z višjo številko na konec padajoče vrstice, če je absolutna vrednost razlike med njo in zadnjo karto v vrstici natanko 10. Tej potezi pravimo *vzvratni trik*. Pazimo na to, da zaradi tega dodatnega pravila vrednosti kart v „naraščajoči“ vrstici v resnici niso nujno naraščajoče, enako pa tudi tiste v „padajoči“ vrstici niso nujno padajoče.

Ko igralec tako položi dve karti, pobere s kupa dve novi, eno po eno. S tem se njegova poteza konča. Če na kupu ni več kart, nadaljuje z igro na enak način, le

brez pobiranja novih kart. Igra se konča, ko igralec bodisi nima v roki nobene karte več (v tem primeru je zmagal) bodisi ne more položiti nobene od kart, ki jih ima trenutno v roki (v tem primeru velja, da je igro izgubil).

Primer. Recimo, da ima igralec na začetku v roki (torej da je kot prvih osem kart pobral):

$$69, 17, 59, 32, 31, 77, 87, 89.$$

Lahko se odloči, da bo položil karto 89 v prvo padajočo vrstico in nato še karto 17 v drugo naraščajočo vrstico. Stanje vseh štirih vrstic po tej potezi je:

$$\begin{aligned} 1 &\uparrow \\ 1 &\uparrow 17 \\ 100 &\downarrow 89 \\ 100 &\downarrow \end{aligned}$$

Nato mora pobrati s kupa dve novi karti; recimo, da sta to karti 84 in 3. V roki ima zdaj karte:

$$69, 59, 32, 31, 77, 87, 84, 3.$$

V drugi potezi se morda odloči položiti karto 3 v prvo naraščajočo vrstico in karto 87 v prvo padajočo vrstico (za karto 89). Stanje vrstic po tej potezi je:

$$\begin{aligned} 1 &\uparrow 3 \\ 1 &\uparrow 17 \\ 100 &\downarrow 89, 87 \\ 100 &\downarrow \end{aligned}$$

Vladimir je odigral nekaj iger, vendar ni vsakič zmagal. Ker sovraži poraze, mu **napiši program**, ki preuči začetno stanje kupa za pobiranje kart in napove izid igre. To bo Vladimirju pomagalo pri odločitvi, ali naj igro odigra ali ne.

Upoštevaj tudi, da je Vladimir zelo logična in predvidljiva oseba. Igrá po naslednjih pravilih:

- Ko pobere karto, jo v roki potem postavi desno od vseh, ki jih je imel že pred tem v roki.
- Pri polaganju kart se drži naslednjih prioritet:
  1. Če mu ena ali več kart omogoča uporabiti vzvratni trik, bo položil najbolj levo od teh kart. Če je mogoče tisto karto uporabiti za vzvratni trik v več vrsticah, jo bo položil v najbolj zgornjo od teh vrstic.
  2. Sicer bo položil neko karto na običajen način. Karto in vrstico, v katero jo bo položil, si bo izbral tako, da bo minimiziral absolutno vrednost razlike med karto, ki jo polaga, in karto, ki je bila pred tem na koncu tiste vrstice. Če je mogoče ta minimum doseči pri več različnih kartah, bo uporabil najbolj levo od teh kart. Če je pri tisti karti potem mogoče minimum doseči pri več različnih vrsticah, jo bo položil v najbolj zgornjo od teh vrstic.

Tvoj program naj izpiše končno stanje igre.

Vhodni podatki. V prvi (in edini) vrstici je 98 celih števil, ločenih s presledki; to je neka permutacija množice $\{2, 3, \ldots, 99\}$ in predstavlja začetno stanje kupa, s katerega se pobira karte. Karte so navedene po vrsti od vrha kupa do dna.

Izhodni podatki. Izpiši šest vrstic. Prve štiri vrstice naj opisujejo štiri vrstice kart na mizi; v peti vrstici naštej karte, ki so igralcu ostale v roki (lahko tudi nobena); v šesti vrstici pa naštej karte, ki so na koncu še ostale na kupu (lahko tudi nobena). Glej tudi primer izhoda spodaj.

Primer vhoda (za potrebe prikaza je tukaj razbit na več vrstic, v resnici pa bo to v podatkih ena sama dolga vrstica):

```
96 69 40 94 35 7 53 88 10 89 47 37 16 61 24 46 90 6 33 25 63 73 26 81 2 45 77 75 48 57 66
      34 59 92 44 11 31 18 9 52 91 50 8 98 5 64 86 62 83 4 19 3 27 97 28 36 23 76 58 30
      38 12 39 78 41 56 80 67 70 99 13 42 17 49 84 22 32 29 54 71 51 74 79 95 72 15 87 21
      65 68 60 85 55 43 93 20 82 14
```

Pripadajoči izhod:

```
1 7 10 16 6 9 11 18 31 62 64 83 86 91 92 97 98 99
1 2 5 8 19 23 27 28 30 36 38 39 41 56 58 67 70 76 78 80 84 74 79 95
100 96 94 89 88 69 61 53 47 46 40 37 35 33 26 25 24 34 44 42 22 32 29 17 13 12 4 3
100 90 81 77 75 73 66 63 59 57 52 50 48 45 21 15
49 54 71 51 72 87 65 68
60 85 55 43 93 20 82 14
```

Še en primer vhoda:

```
87 31 58 56 82 93 9 68 65 41 26 64 3 11 5 84 24 46 16 30 14 85 52 12 91 75 96 17 47 37 76
      69 78 49 25 28 48 81 95 63 34 43 27 74 80 62 53 83 40 71 72 35 23 21 51 66 55 61 67
      32 38 29 60 39 4 18 20 77 7 94 59 42 79 10 92 97 57 2 86 33 89 90 88 19 22 99 45 44
      73 70 50 6 15 98 54 13 36 8
```

Pripadajoči izhod:

```
1 9 11 16 24 14 17 26 28 30 31 34 62 74 78 80 81 71 72 83 95 96 97 99
1 3 5 12 25 27 29 38 39 42 59 60 66 67 57 77 79 86 89 90 92 94 98 88
100 93 87 82 68 65 64 58 56 46 41 37 47 43 53 51 61 55 45 44 33 22 20 19 15 13 10 8 6
100 91 85 84 76 75 69 63 52 49 48 40 35 32 23 21 18 7 4 2
73 70 50 54 36
```

Pri tem drugem izhodu je šesta vrstica prazna.

POSKUSNO TEKMOVANJE
(26. novembra 2022)

X. Kronogram

(*Omejitev časa: 1 s.* *Omejitev pomnilnika: 256 MB.*)

Kronogram je napis, ki ga običajno najdemo na spomenikih in v katerem je zakodirano leto dogodka, ki ga spomenik obeležuje. Leto dobimo tako, da seštejemo

vrednosti tistih črk v napisu, ki lahko nastopajo v rimskih številkah. Vrednosti posameznih črk so:

$$\mathrm{I} = 1,\ \ \mathrm{V} = 5,\ \ \mathrm{X} = 10,\ \ \mathrm{L} = 50,\ \ \mathrm{C} = 100,\ \ \mathrm{D} = 500,\ \ \mathrm{M} = 1000.$$

Kronograme običajno preučujejo menihi. V okviru neke za srednjo Evropo zelo pomembne raziskave so zbrali dolg seznam kronogramov. Izogniti se želijo morebitnim napakam pri izračunu let za posamezne kronograme, zato so se odločili uporabiti računalniški program. Ker pa menihi niso vešči programiranja, nujno potrebujejo tvojo pomoč.

Napiši program, ki iz kronograma izračuna leto dogodka.

Vhodni podatki. V prvi vrstici je celo število T, ki pove, koliko kronogramov sledi. V vsaki od naslednjih T vrstic je po en kronogram, ki je dolg največ 1000 znakov. Besedilo je sestavljeno le iz velikih črk angleške abecede in presledkov.

Omejitve vhodnih podatkov: $1 \le T \le 10\,000$.

Izhodni podatki. Za vsak kronogram izpiši po eno vrstico, vanjo pa eno samo celo število, namreč leto, ki je zakodirano v tistem kronogramu.

Primer vhoda (ker sta kronograma predolga, sta tu za potrebe prikaza izpisana vsak čez dve vrstici, kot je vidno iz zamika):

```
2
VIDE ISTA ECCLESIA CATHEDRALIS SVB PRAESVLE FRANCISCO PAROCHOQVE
     STANSLAO LAETA EXSISTIT REVIVISCENS
ENTERTAINING REGIONAL CONTEST HELD IN EXTRAVAGANT LJUBLJANA WITH
     RIDICULOUSLY HARD TASKS
```

Pripadajoči izhod:

```
1999
2022
```

Y. Permutacije

(*Omejitev časa: 2 s. Omejitev pomnilnika: 256 MB.*)

Naj oznaka $[n]$ pomeni množico $\{1, 2, \ldots, n\}$. *Permutacija* reda n je bijektivna preslikava iz $[n]$ v $[n]$. Na primer: recimo, da je π: $[7] \to [7]$ neka konkretna permutacija reda 7. Običajno jo predstavimo s tabelo, takole:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 7 & 4 & 1 & 6 & 5 & 2 \end{pmatrix}.$$

To pomeni, da je $\pi(1) = 3$, $\pi(2) = 7$, $\pi(3) = 4$ in tako naprej. Ker je zgornja vrstica vedno 1 2 ... n, jo običajno opustimo in zapišemo permutacijo v *enovrstičnem zapisu*:

$$\pi = 3\ 7\ 4\ 1\ 6\ 5\ 2.$$

Zelo priljubljen je tudi *ciklični zapis*. Začnemo z elementom 1 in določimo njegovo sliko, $\pi(1) = 3$. Nato vzamemo element 3 in določimo njegovo sliko, $\pi(3) = 4$. Če tako nadaljujemo, pridemo prej ali slej spet do elementa 1. In res, v našem

primeru je že $\pi(4) = 1$. Permutacija π torej vsebuje cikel (1 3 4). Nato vzamemo najmanjše število, ki se ni pojavilo še v nobenem ciklu — v našem primeru je to 2 — in postopek ponovimo. Sčasoma dobimo:

$$\pi = (1\ 3\ 4)(2\ 7)(5\ 6).$$

Permutacijo reda n lahko predstavimo tudi s *tabelo inverzij* $b_1\ b_2\ b_3\ \ldots\ b_n$ (kjer je $0 \leq b_i \leq n - i$), pri čemer b_i pomeni število tistih elementov, ki so večji od i in v enovrstičnem zapisu stojijo levo od elementa i. Pri našem konkretnem primeru je tabela inverzij takšna:

$$3\ 5\ 0\ 1\ 2\ 1\ 0.$$

Znano je, da je mogoče vsako permutacijo enolično zapisati s tabelo inverzij in da vsaka tabela inverzij $b_1\ b_2\ b_3\ \ldots\ b_n$, pri kateri je $0 \leq b_i \leq n - i$ za vse $i \in [n]$, predstavlja neko veljavno permutacijo.

Napiši program, ki pretvori tabelo inverzij neke permutacije v njen ciklični zapis.

Vhodni podatki. V prvi vrstici je celo število n, torej red permutacije. V drugi vrstici je n celih števil, ločenih s po enim presledkom; ta števila opisujejo neko veljavno tabelo inverzij.

Omejitve vhodnih podatkov: $1 \leq n \leq 10^5$.

Izhodni podatki. Izpiši eno vrstico, ki predstavlja ciklični zapis permutacije, katere tabelo inverzij si dobil v vhodnih podatkih. Vsak cikel mora biti v oklepajih. Cikli naj bodo ločeni s po enim presledkom. Tudi števila v vsakem ciklu naj bodo ločena s po enim presledkom. Vsak cikel naj se začne s svojim najmanjšim elementom, cikli pa naj bodo urejeni naraščajoče glede na svoj najmanjši element. (Glej tudi primera spodaj.)

Primer vhoda:

```
7
3 5 0 1 2 1 0
```

Pripadajoči izhod:

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

Še en primer:

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

Pripadajoči izhod:

```
(1 5) (2 3) (4) (6)
```

Z. Sokoban

(*Omejitev časa: 10 s.* *Omejitev pomnilnika: 512 MB.*)

Če poznaš računalniško igro Sokoban, je opis te naloge zelo kratek. Najti moraš najmanjše število premikov škatel, s katerim je mogoče spraviti vse škatle na odlagališča. Število premikov igralca pri tem ni pomembno.

Igra Sokoban poteka na karirasti mreži, kjer se lahko igralec v vsakem koraku premakne s trenutnega polja na eno od štirih sosednjih polj v vodoravni ali navpični smeri. Nekatera polja so zazidana; na nekaterih nezazidanih poljih stojijo škatle; nekatera nezazidana polja so odlagališča. Igralčev cilj je spraviti vse škatle na odlagališča. Igralec lahko premakne škatlo tako, da se premakne na polje, na katerem

ta škatla stoji, ona pa se pri tem premakne v naslednje polje v smeri igralčevega premika. Če tisto polje ni prazno, se škatle ne dá premakniti. Igralec škatle ne more poriniti v drugo škatlo ali v zid, škatel pa tudi ne more vleči ali jih premikati kako drugače. Edini dovoljeni način za premikanje škatel je z rinjenjem. Škatlo se sme premakniti, četudi že stoji na odlagališču. Pomembno je le, da na koncu vse škatle pristanejo na odlagališčih.

Celo zelo majhni razporedi so lahko skrajno kompleksni, zato moramo za reševanje pri igri Sokoban uporabiti zelo učinkovit algoritem.

Vhodni podatki. Mreža velikosti $w \times h$ je opisana s h vrsticami, vsaka od teh pa vsebuje največ w znakov, ki opisujejo polja v tisti vrstici mreže. Pomen posameznih znakov je naslednji:

„`#`“ — zid;
„`@`“ — igralec (na polju, ki ni odlagališče);
„`+`“ — igralec, ki stoji na odlagališču;
„`$`“ — škatla (na polju, ki ni odlagališče);
„`*`“ — škatla, ki stoji na odlagališču;
„`.`“ — odlagališče, na katerem ni škatle ali igralca;
„ “ (presledek) — prazno polje (brez škatle ali igralca in ni odlagališče).

Nekatere vrstice imajo lahko manj kot w znakov, ker se lahko presledke na koncu vrstice opusti. Zagotovljeno je, da stoji igralec na območju, ki je z vseh strani obdano z zidovi. Število škatel b je enako številu odlagališč.

Omejitve vhodnih podatkov:

- $3 \leq w \leq 8$, $3 \leq h \leq 8$;
- $1 \leq b \leq 4$.

Izhodni podatki. Izpiši eno samo vrstico, vanjo pa eno samo celo število, namreč najmanjše število rinjenj škatel, s katerimi je mogoče premakniti vse škatle na odlagališča. Zagotovljeno je, da je problem rešljiv.

Trije primeri vhodov in pripadajoči izhodi:

Vhod 1:
```
###
#.#####
#..   #
# $$$@#
#    ##
######
```

Izhod 1:
```
9
```

Vhod 2:
```
####
#. ####
#.$   #
#@ $$ #
###.  #
  #####
```

Izhod 2:
```
13
```

Vhod 3:
```
 ####
 #  #
##  ###
#.  $ #
# +$* #
#     #
#######
```

Izhod 3:
```
6
```

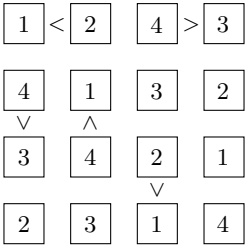
NEUPORABLJENE NALOGE IZ LETA 2020

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 15. tekmovanjem ACM v znanju računalništva (leta 2020), pa jih potem na tistem tekmovanju nismo uporabili (ker se nam je nabralo več predlogov nalog, kot smo jih potrebovali za tekmovanje). Ker tudi te neuporabljene naloge niso nujno slabe, jih zdaj objavljamo v letošnjem biltenu, če bodo komu mogoče prišle prav za vajo. Poudariti pa velja, da niti besedilo teh nalog niti njihove rešitve (ki so na str. 181–209) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Futoširi

Futoširi je še ena izmed popularnih japonskih logičnih ugank. Igra se na mreži $n \times n$ kvadratnih polj, kjer moramo v vsako polje vpisati eno od števil od 1 do n. V vsak stolpec in vrstico moramo napisati natanko vsa števila od 1 do n, torej se števila v posamezni vrstici ali stolpcu ne smejo ponavljati. Poleg tega lahko ponekod med dvema sosednjima poljema (torej takima, ki imata skupno eno od stranic) stoji tudi znak $<$ ali $>$, ki pove, da mora biti vrednost v enem polju manjša oz. večja kot v sosednjem polju.

Napiši program ali podprogram (funkcijo), ki kot vhodne podatke dobi že izpolnjeno mrežo (poleg števil v poljih dobi tudi podatke o znakih $<$ in $>$ med njimi) in preveri, ali je izpolnjena pravilno (v skladu s pogoji, opisanimi v prejšnjem odstavku). Podrobnosti tega, kako so podatki predstavljeni, določi sam in jih v svoji rešitvi tudi opiši. Spodnja slika kaže primer pravilno izpolnjene mreže:



2. Varnostnik

Varnostnik Miha je zadolžen za to, da pisarne zaradi strogih varnostnih standardov (smo vendar sredi epidemije) niso prepolne. Ko varnostnik zjutraj pride na delo, so vse pisarne prazne. Hkrati pa dobi tudi spisek največjih zapolnjenosti posameznih pisarn, ki jih te tekom dneva ne smejo preseči. Ker so razmere resne, je dobil tudi dodatna navodila, ki omejujejo število ljudi v vsaki skupini po 10 pisarn (1–10, 11–20 itd.). Varnostnik skozi celoten dan spremlja ljudi, ki vstopajo in izstopajo, in si zapisuje, v katero pisarno (ali iz katere) gredo. Če poskuša kdo vstopiti v pisarno, ki bi s tem postala prezasedena (ali pa bi s tem postala prezasedena njena skupina desetih pisarn), ga odslovi.

Napiši program, ki simulira obnašanje takega varnostnika: bere naj podatke o prihodih in odhodih ljudi ter za vsakega sproti izpiše, ali lahko vstopi ali ne.

Vhodni podatki: v prvi vrstici je število pisarn n; v drugi so omejitve posameznih pisarn (n celih števil, ki povedo maksimalno število ljudi v posamezni pisarni); v tretji so omejitve skupin po 10 pisarn ($\lceil n/10 \rceil$ števil); v četrti vrstici je število prihodov in odhodov m; nato pa sledi m vrstic, ki po vrsti opisujejo posamezne prihode in odhode. V vsaki od teh vrstic je celo število, čigar absolutna vrednost predstavlja številko pisarne (od 1 do n), predznak pa pove, ali je prišel nekdo, ki bi rad vstopil v to pisarno (če je število pozitivno), ali nekdo zdaj odhaja iz te pisarne (če je število negativno). Po vsakem prebranem prihodu naj tvoj program bodisi izpiše, da ta človek lahko vstopi („`OK`“), bodisi navede razlog, zakaj ne more vstopiti („`POLNA PISARNA`“, „`POLNA SKUPINA`“). Če je človek odslovljen, se odpravi domov (ne čaka pred vrati) in varnostnik nadaljuje z delom (branjem naprej).

3. Funkcije

Dan je niz znakov, ki predstavlja klic funkcije. Sestavljajo ga ime funkcije (sestavljeno iz samih malih črk), oklepaj, 0 ali več parametrov (ločenih z vejicami) in zaklepaj. Parametri so bodisi števila (sestavljena le iz števk od 0 do 9) bodisi so tudi sami klici funkcij. Ob vejicah in oklepajih je lahko tudi po eden ali več presledkov. Nekaj primerov:

```
f(1, g(g(2)), h())
ena(0, dve(10, ena(tri(4)), 5), stiri(tri(6), tri(7)))
abc(def(abc(def())))
```

Vidimo, da se lahko v posameznem izrazu pojavlja ista funkcija tudi po večkrat; v prvem izrazu zgoraj je to funkcija `g`, v drugem funkciji `ena` in `tri`, v tretjem pa funkciji `abc` in `def`. Rekli bomo, da je niz *dosleden*, če imajo različne pojavitve iste funkcije vse enako število parametrov. Od zgornjih primerov je tako prvi niz dosleden, ostala dva pa ne: pri drugem primeru ima funkcija `ena` enkrat tri parametre, enkrat pa samo enega; pri tretjem primeru pa ima `def` enkrat en parameter, enkrat pa nobenega. **Napiši podprogram** (funkcijo), ki kot parameter dobi niz in zanj preveri, če predstavlja sintaktično pravilen klic funkcije (v skladu z zgoraj opisanimi pravili) in če je dosleden.

4. Sestankovalna sova

Sestankovalna sova je naprava za podporo pri izvedbi videokonferenc. Na podstavku je glava v obliki sove, ki se lahko vrti levo in desno, v njej pa so kamera in mikrofoni. Glava je povezana s kabli, katerih dolžina je preračunana tako, da se lahko obrne za tri polne kroge levo ali desno od izhodiščne oz. „nevtralne“ smeri, preden se kabli zategnejo oz. pretrgajo. Ko nekdo v bližini sove govori, lahko naprava na podlagi podatkov iz mikrofonov določi, iz katere smeri prihaja zvok, in obrne glavo v tisto smer, tako da se bo govorca videlo na kameri. **Napiši program**, ki bo na ta način usmerjal glavo proti govorcem in pri tem pazil, da ne naredi toliko krogov v eno smer, da bi se kabli poškodovali.

Lažja različica: imaš funkcijo **int** SmerZvoka(), ki vrne smer v stopinjah, od koder prihaja zvok (smer je z območja $(-180, 180]$ in je podana relativno glede na nevtralno smer), in funkcijo **void** ObrniKamero(**int** kot), ki obrne kamero za kot stopinj v levo, če je kot negativen, oz. za $-$ kot stopinj v desno, če je kot negativen. Vse, za kar moraš

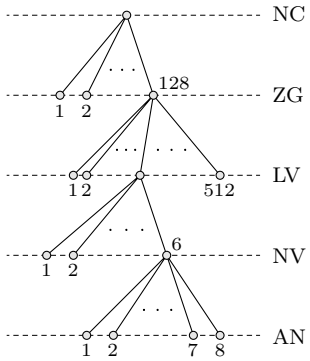
poskrbeti, je, da se glava ne zamota, ampak se včasih obrne skozi večji kot, zato da se ne bi zgodilo, da je kamera naredila tri polne kroge (glede na nevtralno smer) in s tem zapletla kable. Zagotovljeno je tudi, da ob vklopu naprave glava vedno obrnjena v nevtralno smer (0 stopinj).

Težja različica: nimaš funkcije SmerZvoka, pač pa imaš tri mikrofone (eden gleda tja kot kamera, druga dva sta vsak 120 stopinj stran). Za vsak mikrofon lahko dobiš podatek o jakosti zvoka; vrne jo funkcija **int** Jakost(**int** mikrofon), pri čemer kot številko mikrofona podaj 0 za mikrofon v smeri kamere ter ± 1 za stranska dva). Iz teh podatkov moraš pravilno usmeritev glave določiti sam (jakost, ki jo zaznava posamezni mikrofon, je tem večja, čim manjši je kót med smerjo, v katero je obrnjen mikrofon, in smerjo, iz katere prihaja zvok).

Še težja različica: enako kot prejšnja, vendar imaš samo *dva* mikrofona, postavljena pod kotom 90 stopinj glede na kamero (se pravi tako kot ušesa in oči pri človeku).

5. Ultranet

Imamo omrežje naprav, ki ima obliko drevesa z več nivoji: na vrhu je NC (nadzorni center), pod njim so ZG (zgoščevalniki), nato LV (linijski vmesniki), nato NV (naročniški vmesniki), nato AN (alarmne naprave). Vmesnim nivojem (ZG, LV in NV) pravimo krajše tudi *prenosne naprave*. Naslednja slika kaže del takega omrežja:



Naslovi naprav: vsaka naprava ima naslov, ki jo enolično definira v sistemu. Naslov je četverica celih števil naslednje oblike:

- ZG: $(\{1..128\}, 0, 0, 0)$
- LV: $(\{1..128\}, \{1..512\}, 0, 0)$
- NV: $(\{1..128\}, \{1..512\}, \{1..6\}, 0)$
- AN: $(\{1..128\}, \{1..512\}, \{1..6\}, \{1..8\})$

Oblika sporočil: naprave si med seboj pošiljajo sporočila; sporočilo je par ⟨naslov naprave; status⟩, pri čemer je status iz $\{0, 1, 2\}$.

Naprave delujejo po naslednjih pravilih:

(1) *Nadzorovanje prenosnih naprav in poti*

(1.1) *Normalno delovanje.* Vsaka naprava (alarmna in prenosna) vsaki dve sekundi pošlje svoji neposredno nadrejeni napravi sporočilo „še sem živ“:

$$\langle \text{naslov naprave (pošiljateljice); status} = 0 \rangle.$$

Nadrejena naprava takega sporočila ne pošlje naprej.

Dokler NC dobiva iz podrejenih ZG samo sporočila „še sem živ“, imajo v NC vse nadzorovane naprave status „normalno“.

(1.2) *Izpad naprave.* Če prenosna naprava več kot pet sekund ne dobi nobenega sporočila iz kakšne od svojih od neposredno podrejenih naprav, prične vsaki dve sekundi namesto sporočila „še sem živ“ pošiljati svoji neposredno nadrejeni napravi sporočilo o nedosegljivosti podrejene naprave:

$$\langle \text{naslov podrejene naprave, ki ne odgovarja; status} = 1 \rangle.$$

Vsaka naprava, ki od svoje podrejene naprave prejme tako sporočilo o nedosegljivosti, ga takoj posreduje svoji nadrejeni napravi in tako naprej, dokler sporočilo o nedosegljivosti ne pride v NC.

Ko pride v NC sporočilo o nedosegljivosti neke naprave, dobijo v NC vse nadzorovane AN, ki so vezane na napravo, ki je nedosegljiva, status „nedosegljiv“.

(1.3) *Ponovno delovanje prej izpadle naprave.* Ko dobi nadrejena naprava od prej nedosegljive podrejene naprave spet sporočilo „še sem živ“, preneha pošiljati sporočila o nedosegljivosti te naprave.

Ko v NC prenehajo prihajati sporočila o nedosegljivosti naprave, dobijo v NC vse nadzorovane naprave, ki so vezane na napravo, ki je bila prej nedosegljiva, spet status „normalno“.

(2) *Nadzorovanje alarmnih dogodkov*

(2.1) *Alarmno stanje.* Če katera od alarmnih naprav zazna alarmni dogodek, prične vsaki dve sekundi namesto sporočila „še sem živ“ pošiljati svoji neposredno nadrejeni napravi sporočilo o alarmnem stanju:

$$\langle \text{naslov alarmne naprave; status} = 2 \rangle.$$

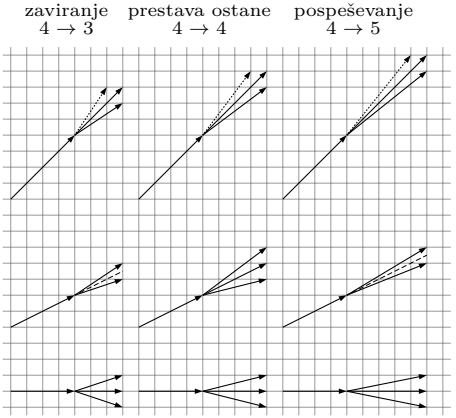
Vsaka naprava, ki od svoje podrejene naprave prejme tako sporočilo o alarmnem stanju, ga takoj posreduje svoji nadrejeni napravi in tako naprej, dokler sporočilo o alarmu ne pride v NC.

Ko pride v NC sporočilo o alarmnem stanju iz neke alarmne naprave, dobi v NC ta alarmna naprava status „alarm“.

(2.2) *Konec alarmnega stanja.* Ko alarmna naprava, ki je bila v alarmnem stanju, ne zaznava več alarmnega dogodka, preneha pošiljati sporočila o alarmnem stanju in prične spet pošiljati sporočilo „še sem živ“.

Ko v NC ne prihajajo več sporočila o alarmnem stanju, dobi v NC alarmna naprava, iz katere so prej prihajali alarmi, spet status „normalno“. (Če torej več kot 2 sekundi ne dobimo sporočila, da je tista naprava še vedno v alarmu, potem sklepamo, da je zdaj normalna.)

Napiši program, ki na osnovi podanega niza zaporednih sporočil, ki jih prejme NC, izpiše, katere AN so v alarmu oziroma katere niso dosegljive.



Nekaj primerov možnih sprememb smeri in hitrosti. V prvem koraku je bil pri vseh primerih avtomobil v prestavi 4. Levi stolpec kaže primere zaviranja iz četrte prestave v tretjo; srednji stolpec kaže možnosti zavijanja, pri čemer ostanemo v 4. prestavi; desni stolpec kaže primere pospeševanja iz četrte prestave v peto. V primerih iz zgornje vrstice je bila hitrost v prvem koraku $(4, 4)$, v srednji vrstici $(4, 2)$, v spodnji pa $(4, 0)$.

Črtkana črta pri prvem in zadnjem primeru v srednji vrstici kaže, kako bi se nadaljevala dosedanja smer (vendar tako ne moremo nadaljevati, ker ne bi prišli v eno od točk mreže).

Pikčasta puščica v primerih v zgornji vrstici kaže možnost zavijanja prek diagonale, ki je tudi dovoljeno, vendar se potem v drugem koraku prestava nanaša na drugo komponento hitrosti kot v prvem koraku (navpično namesto vodoravne).

6. Avtomobili

Na karirasti mreži se v diskretnih korakih premika namišljen avtomobil. Podano je zaporedje njegovih položajev $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, ki predstavljajo n korakov (i-ti korak je od (x_{i-1}, y_{i-1}) do (x_i, y_i); vse koordinate so celoštevilske). Hitrost vožnje avtomobila lahko torej predstavimo z vektorjem oblike $(\triangle_x, \triangle_y)$, v katerem števili $\triangle_x$ oz. $\triangle_y$ povesta, za koliko se v enem koraku spremeni x- oz. y-koordinata avtomobila.

Avtomobil se v vsakem trenutku nahaja v eni od petih prestav, ki so oštevilčene od 1 do 5; v prvem koraku mora biti prestava 1. Pri tem prestava p pomeni, da je ena od vrednosti $|\triangle_x|$ in $|\triangle_y|$ enaka p, druga pa je manjša ali enaka p. Prestava se lahko po vsakem koraku poveča ali zmanjša največ za 1 (lahko pa tudi ostane enaka); druga komponenta hitrosti pa se lahko spremeni le toliko, da smo največ za eno enoto oddaljeni od točke, ki bi jo dosegli, če bi nadaljevali v dosedanji smeri. Podrobnosti tega pravila kažejo primeri na gornji sliki.

Napiši program ali podprogram, ki za dani zapis vožnje (zaporedje položajev) preveri, če so bila med vožnjo upoštevana vsa pravila glede spreminjanja hitrosti.

7. BinoXXO

Polja na igralni deski z mrežo polj 6×6 moramo skladno s pravili igre zapolniti z znakoma „`X`“ in „`O`“. Pravila so naslednja:

- Vsako polje je treba zapolniti z `X` ali `O`.
- V vrstici/stolpcu ne smeta biti več kot dve polji z enakima znakoma skupaj.
- V vsaki vrstici/stolpcu mora biti enako število znakov `X` in `O`.
- Vsaka vrstica mora biti drugačna.
- Vsak stolpec mora biti drugačen.

Primer pravilno izpolnjene mreže:

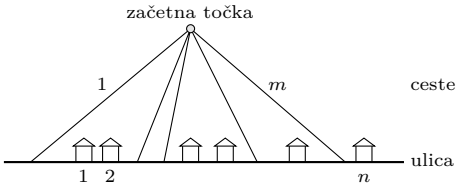
○	○	×	×	○	×
×	×	○	○	×	○
×	○	○	×	○	×
○	×	×	○	○	×
○	×	×	○	×	○
×	○	○	×	×	○

Napiši program ali podprogram, ki kot vhodni podatek dobi že izpolnjeno ploščo in preveri, ali so upoštevana vsa pravila.

8. Trgovski potnik

Na ravni ulici je n hiš; če trgovski potnik obišče hišo i, se s tem zamudi t_i časa in zasluži p_i denarja. Iz njegove začetne točke (ki ni na ulici) vodi do ulice m cest, pri čemer traja vožnja po j-ti od njih d_j časa. Za vsako od teh cest vemo, med katerima dvema hišama doseže ulico. Trgovski potnik bi šel rad od začetne točke po eni od cest in nato nazaj po neki drugi cesti, spotoma pa bi še obiskal vse hiše, ki ležijo na ulici med obema uporabljenima cestama. Pri tem bi rad maksimiziral razmerje med zaslužkom (vsota p_i po vseh hišah, ki ležijo na ulici med obema uporabljenima cestama) in časom (vsota t_i po teh hišah + še d_j obeh uporabljenih cest). **Opiši postopek**, ki ugotovi, kateri dve cesti naj potnik uporabi.

Naslednja slika kaže primer z $n = 6$ hišami in $m = 5$ cestami:



9. Sprehod po mreži I

Dana je karirasta mreža z $w \times h$ polji. Vsako polje je bodisi prehodno bodisi neprehodno. Po mreži se lahko premikamo, vendar le po prehodnih poljih. V enem koraku se lahko premaknemo iz polja v eno od osmih sosednjih polj, ki imajo z dosedanjim poljem skupno vsaj eno oglišče. Možnih je torej le osem smeri premika. V prvem koraku se lahko premaknemo v katerokoli smer, kasneje pa se moramo držati naslednje omejitve: da bo sprehod bolj razgiban, se smer premika v dveh zaporednih korakih ne se sme razlikovati za manj kot 90°. To na primer pomeni, da če smo se v enem koraku premaknili gor in desno $\nearrow$, se v naslednjem koraku ne smemo premakniti v eno od smeri $\nearrow$, $\uparrow$ in $\rightarrow$, lahko pa se v kakšno od ostalih petih smeri. **Opiši postopek**, ki za dano mrežo poišče pot, ki v najmanj korakih pride od danega začetnega polja do danega ciljnega polja.

10. Sprehod po mreži II

Nadobudni hribolazec se je v hribih izgubil in v gosti temi vandral naokoli. Ko se je zjutraj zbudil, je ugotovil, da je pristal v neznani dolini. Poti se zaradi teme ne

spomni, ve pa, da se je vedno zgolj spuščal. **Napiši program**, ki prebere karirasto mrežo, kjer so zapisane višine posameznih točk, in položaj doline, kjer se je naš hribolazec zbudil. Program naj najde najdaljšo pot, po kateri je lahko vandral hribolazec. Taka pot bo iz začetnega položaja vedno sestavljena iz strogo padajočih višin, končati pa se mora v točki, kjer se je hribolazec zbudil. Če je takih poti več, naj program izpiše najdaljšo, če pa je najdaljših več, lahko izpiše katerokoli izmed njih. Hribolazec se po mreži premika tako, da se v vsakem koraku iz trenutne točke mreže premakne za eno enoto na sever, jug, vzhod ali zahod.

Vhodni podatki: v prvi vrstici so štiri števila, w, h, z_x, z_y: širina in višina mreže ter koordinati točke, kjer se je hribolazec zbudil ($0 \leq z_x < w$ in $0 \leq z_y < h$; y-koordinate se štejejo od zgoraj navzdol). Sledi h vrstic s po w številkami (ločenimi s presledki), ki določajo višine posameznih polj mreže.

Izhodni podatki: izpiši zaporedje črk, ki opisujejo smeri premikov pri najdaljši poti, po kakršni sprašuje naloga (`S` = sever, `J` = jug, `V` = vzhod, `Z` = zahod).

Primer vhoda:

```
5 5 0 0
0 1 2 3 4
5 8 5 5 5
9 8 7 8 6
10 9 9 9 10
10 10 10 10 10
```

Eden od možnih pripadajočih izhodov:

```
SSVSSZZZZ
```

Komentar: ta izhod ustreza poti, ki se začne v točki $(4, 3)$, torej v predzadnji točki zadnje vrstice.

11. Ocenjevanje nalog

Sestavili smo kup nalog, a nismo prepričani, kako težke so. V reševanje jih damo učencem in gledamo, kateri učenci (in kdaj) so naloge uspešno rešili in kateri ne. Ko učenec nalogo poskusi rešiti, jo lahko reši ali pa tudi ne, v vsakem primeru pa se k njej kasneje ne vrača več. Sedaj želimo nalogam dodeliti težavnostne ocene (naravna števila), tako da velja naslednje:

1. Če je učenec rešil nalogo težavnosti x, potem bo od tistega trenutka naprej znal rešiti vse naloge težavnosti x ali manj (če bo torej kasneje poskušal rešiti kakšno nalogo enake ali nižje težavnosti, jo bo zagotovo rešil).

2. Če je učenec neuspešno poskušal rešiti nalogo težavnosti x, potem morajo imeti vse naloge, ki jih je uspešno rešil do tistega trenutka (torej pred tem neuspešnim poskusom), oceno $x - 1$ ali manj. (Torej dopuščamo, da se učenci skozi čas izboljšujejo, nikoli pa se ne poslabšajo).

Najti želimo ocene, ki ustrezajo tem pogojem, hkrati pa želimo, da bi bila najvišja ocena čim manjša. **Opiši postopek**, ki poišče tak nabor ocen ali pa ugotovi, da ne obstaja. Kot vhodne podatke dobi za vsakega učenca po en seznam parov (a_i, b_i), pri čemer je a_i številka naloge (naloge so oštevilčene od 1 do n), b_i pa pove, ali jo je uspel rešiti; pari so urejeni v takem vrstnem redu, v kakršnem jih je ta učenec poskušal reševati.

Lažja različica: za vsak k velja, da so tisti, ki so nalogo k sploh poskušali rešiti, to storili na dan k in ne na kak drug dan.

Težja različica: za nekatere naloge je ocena težavnosti predpisana vnaprej.

12. Knjižna kazala

Nekatere knjige imajo trak, ki ga lahko uporabljamo kot kazalo — pustimo ga med dvema listoma knjige in si tako označimo mesto, ki nas zanima. Pri tej nalogi pa imamo knjigo s k takšnimi kazali. Knjiga ima $n + 1$ listov, tako da je možnih položajev posameznega kazala n; te položaje oštevilčimo od 1 do n. Na začetku so vsa kazala na položaju 1. Kazala lahko premikamo, vendar nas to stane nekaj truda; cena tega, da kazalo premaknemo s položaja p na položaj q, je definirana kot $|p - q|$.

Radi bi čim ceneje izvedli zaporedje u ukazov oblike „premakni eno od kazal (sam si lahko izbereš, katero) na položaj x_i“. Pri tem je zaporedje števil $x_1, x_2, \ldots, x_u$ podano kot vhodni podatek. Skupna cena (vsota) takega zaporedja premikov je odvisna od tega, katero od k kazal premaknemo v posameznem koraku. **Opiši postopek**, ki določi najmanjšo možno skupno ceno, s katero je mogoče izvesti zahtevano zaporedje premikov.

13. Drevesasti izrazi

Pri tej nalogi se bomo ukvarjali z matematičnimi izrazi, v katerih nastopajo funkcije in naravna števila. Funkcije so lahko eno- ali večmestne in poljubno gnezdene.

Mislimo si preprost programski jezik za definiranje takšnih izrazov. V vsaki vrstici lahko definiramo en izraz tako, da navedemo ime funkcije in njene argumente; kot argumente pa lahko uporabljamo naravna števila in izraze iz naših definicij. Primer:

```
foo = b(3, 1)
bar = a(baz)
baz = c(5, 2)
quux = a(foo, bar, 7)
```

Če v definicijo nekega izraza nesemo definicije njegovih podizrazov in tako naprej, lahko izraz sčasoma razvijemo tako, da vsebuje le še funkcije in naravna števila, ne pa tudi naših imen podizrazov. V zgornjem primeru lahko na primer izraz `quux` razvijemo v `a(b(3, 1), a(c(5, 2)), 7)`.

Težava nastopi v primerih, ko odvisnosti med izrazi tvorijo cikle, na primer:

```
foo = b(1, bar)
bar = c(foo, 6)
```

Če bi zdaj poskušali razviti izraz `foo`, bi dobili najprej `b(1, bar)`, iz tega `b(1, c(foo, 6))`, iz tega `b(1, c(b(1, bar), 6))` in tako naprej — postopek bi se zaciklal, saj se izraza sploh ne da razviti do konca.

Temu se lahko poskušamo izogniti tako, da na mesta, kjer bi se izraz začel ponavljati, napišemo tri pike: `b(1, c(..., 6))`.

Natančneje lahko naša pravila za razvijanje izrazov opišemo takole:

(1) Če v trenutnem izrazu ne nastopa nobeno ime podizraza (iz naših definicij), je izraz razvit in končamo.

(2) Sicer poiščimo v njem najbolj levo omembo kakšnega imena podizraza.

(2.1) Če smo ime tega podizraza že kdaj prej kje drugje zamenjali (v koraku 2.2) z desno stranjo njegove definicije, zamenjajmo zdaj tokratno omembo njegovega imena s tremi pikami.

(2.2) Sicer zamenjajmo tokratno omembo njegovega imena z desno stranjo njegove definicije.

(3) Vrnimo se na korak 1.

Napiši podprogram, ki obdela definicije izrazov in vsakega od njih izpiše v razviti obliki v skladu z gornjimi pravili. Pri tem ni treba, da tvoj podprogram prebere definicije iz nizov oblike „`foo = b(1, bar)`“, karkšne smo videli zgoraj; predpostaviti smeš, da so definicije podane v kakšni prikladnejši podatkovni strukturi, ki pa jo seveda v svoji rešitvi tudi opiši.

14. Gospodarska rast

Dana je igra na neskončni mreži polj, kjer imamo na začetku d kovancev in osvojenih k polj. Vsa polja so na začetku prazna, kasneje pa bodo na nekaterih poljih stale tovarne ali vojaki. Igra poteka v rundah, pri čemer vsaka runda poteka po naslednjem vrstnem redu:

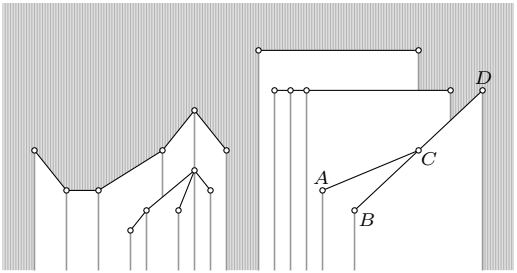
1. Vsakemu vojaku moramo plačati 1 kovanec. Če tega denarja nimamo, nam kraljestvo propade in igro izgubimo.
2. Vsaka tovarna nam dá 5 kovancev.
3. Vojak lahko osvoji novo polje (ki je sosednje kakšnemu od že osvojenih) tako, da se premakne na to polje (pri tem lahko prepotuje poljubno razdaljo; polje, ki ga zapusti, je potem prazno, novo osvojeno polje pa zasedeno; za polja, čez katera je potoval, je vseeno, ali so medtem zasedena ali ne). Posamezni vojak lahko v eni rundi osvoji največ eno novo polje. Vojak lahko osvoji tudi polje, ki je sosednje kakšnemu od polj, ki jih je malo prej v isti rundi osvojil neki drug vojak.
4. Na posamezno prazno osvojeno polje lahko postavimo tovarno ali vojaka; nova tovarna stane 12 kovancev, nov vojak pa 10 kovancev; polje je potem zasedeno. (V eni rundi lahko tako postavimo tudi več tovarn in/ali več vojakov.)

Opiši postopek, ki izračuna, kako lahko v najmanj rundah osvojimo n polj.

15. Dekodiranje besedila

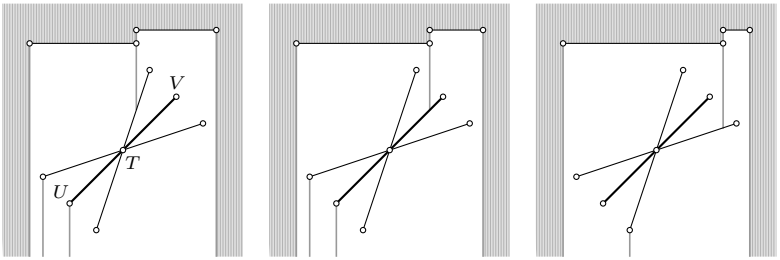
Neki niz znakov (samih malih črk angleške abecede) smo zakodirali tako, da smo si za vsako črko abecede izbrali neko zaporedje ničel in enic, ki mu pravimo *koda* tiste črke, nato pa smo vsako črko našega niza zamenjali z njeno kodo. Zagotovljeno je, da nobena koda ni predpona (prefiks) kakšne druge, tako da pri dekodiranju ne more priti do dvoumnosti. **Napiši program** ali podprogram, ki kot vhodne podatke prejme kode vseh črk in zakodirano obliko niza ter izpiše (ali vrne) niz v prvotni, dekodirani obliki.

Primer: če črko `a` zakodiramo v `001`, črko `b` pa v `10`, se niz `0011010001` dekodira v `abba`.



Primer za nalogo „Strešniki“. Črne daljice predstavljajo strešnike (krožci so njihova krajišča), sive navpične črte pa kažejo, kje pada voda.

Opomba glede točke C: mišljeno je, da je BD en sam strešnik, ki se ga v točki C dotika strešnik AC. Če pa bi namesto tega imeli dva ločena strešnika BC in CD, tako da bi bilo v točki C stičišče treh strešnikov (omenjenih dveh in še AC), bi nastal še en dodaten curek vode iz točke C navpično navzdol.



Pri teh treh primerih je mišljeno, da je UV (debela črta) en sam strešnik, ki se ga v točki T dotikajo štirje drugi strešniki (narisani s tankimi črtami). Zato voda v točki T ne more prodreti skozi strešnik UV; če doseže točko T tako, da se je gibala nad strešnikom UV, bo nad njim tudi ostala (leva in srednja slika), če pa je prišla do T pod strešnikom UV, bo tudi ostala pod njim (desna slika).

---

16. Strešniki

Imamo „streho“ iz n strešnikov. Vsak strešnik je daljica na mreži $[-100, -100] \times [100, 100]$ s celoštevilskimi krajišči. Strešniki niso navpični, pač pa poševni ali vodoravni. Dva strešnika imata lahko skupno največ eno točko in ta točka mora biti krajišče vsaj enega od teh dveh strežnikov (strežniki se torej ne sekajo, lahko pa se dotikajo s krajišči, pri čemer se lahko krajišče enega strešnika tudi dotika notranjosti drugega).

Dežuje od zgoraj in voda curlja po strešnikih navzdol, po naslednjih pravilih (gl. sliki zgoraj):

- Voda teče dol samo z ene strani strešnika (tam, kamor je nagnjen); če pa je strešnik vodoraven, se voda vedno razleze po celem strešniku in curlja z njega z obeh strani.
- Če padejo kapljice na krajišče strešnika, se razpršijo na obe strani krajišča.
- Če se več strešnikov dotika s krajišči, voda curlja tudi vmes (in to dovolj hitro, da se nikjer ne nabira), poleg tega pa se razširi še na sosednje strešnike (seveda le tam, kjer ji pri tem ni treba teči navkreber).

Opiši postopek, ki kot vhodne podatke dobi množico strešnikov (daljic) in izra-

čuna, kateri intervali x-koordinat pod streho so suhi.

17. Predlaganje naslednje besede

Večina pametnih telefonov ob tipkanju besedila predlaga najverjetnejšo naslednjo besedo. Na primer: uporabniku ob pisanju besedila „Danes sem za kosilo jedel kr“ predlaga besedo *krvavice*. **Opiši postopek**, ki predlaga naslednjo besedo glede na prejšnjo zapisano besedo in nekaj začetnih črk nove besede.

Tvoj postopek naj kot najverjetnejšo naslednjo besedo vrne tisto, ki se začne na podane začetne črke in ki je bila v preteklosti najpogostejša naslednja beseda za dano prejšnjo besedo. Prav tako naj upošteva, da se je uporabnik pri tipkanju prejšnje besede lahko zatipkal v največ eni črki in sicer tako, da je namesto nje natipkal neko drugo črko; na primer: namesto *muca* je lahko mislil *maca*, *mula*, gotovo pa ne *muc* ali *taca*. Pri vseh možnih prejšnjih besedah primerjaj najbolj verjetne naslednje besede in vrni tisto, ki je v zgodovini imela največ pojavitev. Tvoj postopek naj besede predlaga glede na zgodovino tipkanja uporabnika.

Vhodni podatki: najprej dobiš seznam z besed, dolgih po največ m znakov, ki predstavljajo zgodovino uporabnikovega tipkanja. V nalogi se bo pojavilo največ n različnih besed. Besede so sestavljene le iz malih črk angleške abecede.

Nato moraš izvesti q poizvedb; i-ta od njih je opisana s trojico (p_i, c_i, a_i), kjer je p_i prejšnja beseda, c_i je začetni (doslej natipkani) del trenutne besede, a_i pa je beseda, ki jo je uporabnik na koncu dejansko napisal. Tvoj postopek mora pri tej poizvedbi torej predlagati najverjetnejšo besedo glede na p_i in c_i, nato pa mora svoje podatkovne strukture popraviti oz. dopolniti tako, da bo upošteval, da je uporabnik zdaj napisal par besed (p_i, a_i), kajti to lahko vpliva na pogostost nekaterih besed in s tem na rezultat kasnejših poizvedb.

REŠITVE NALOG ZA PRVO SKUPINO

1. Snežinke

Naš program bo tekel v neskončni zanki. Pri tem se zanašamo na zagotovilo iz besedila naloge, da se nova snežinka ne pojavi, dokler prejšnja ne izgine, in da vmes mine dovolj časa, da lahko naš program pregleda vse žice.

V vsaki iteraciji glavne zanke preglejmo vse žice (z vgnezdeno zanko), da vidimo, če kakšna od njih zaznava snežinko. Ko prvič zaznamo snežinko, moramo izpisati njeno velikost, v ta namen pa moramo ugotoviti, pri kateri žici se začne in konča. Ni nujno, da je žica, pri kateri smo jo doslej našli, že tudi najbolj leva žica te snežinke, saj naloga pravi, da se snežinka dotakne več žic hkrati in da se lahko pojavi kadarkoli med dvema klicema funkcije Senzor. Iti moramo torej v zanki po žicah levo in desno od trenutne, dokler ne pridemo do take, ki snežinke ne zaznava več; tako bomo ugotovili najbolj levo in najbolj desno žico te snežinike, iz njiju pa lahko izračunamo njeno velikost in jo izpišemo.

Nato počakamo, da snežinka izgine; naloga pravi, da ko snežinka izgine, izgine z vseh senzorjev hkrati, torej je dovolj, če pregledujemo le eno od žic, na katerih smo dosedanjo snežinko našli. Ko snežinka izgine, nadaljujemo z glavno zanko, ki bo v naslednji iteraciji spet začela pregledovati vse žice, dokler ne bo zaznala naslednje snežinke.

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

int main()
{
  enum { N = 100 }; // Število žic.
  while (true)
  {
    // Poiščimo prvo žico, ki zaznava snežinko.
    int L = 1; while (L <= N && ! Senzor(L)) ++L;

    // Morda snežinke ne zaznava nobena žica.
    if (L > N) continue;

    // Sicer poglejmo, kako daleč na desno se razteza.
    int D = L; while (D < N && Senzor(D + 1)) ++D;

    // Poglejmo še, kako daleč na levo se razteza.
    while (L > 1 && Senzor(L − 1)) −−L;

    // Snežinka pokriva žice od L do D; izpišimo jo.
    cout << (D − L + 1) << endl;

    // Počakajmo, da izgine.
    while (Senzor(L)) ;
  }
}
```

Še enaka rešitev v pythonu:

```
N = 100 # Število žic.

while True:
  # Poiščimo prvo žico, ki zaznava snežinko.
```

```
  L = 1
  while L <= N and not Senzor(L): L += 1

  # Morda snežinke ne zaznava nobena žica.
  if L > N: continue

  # Sicer poglejmo, kako daleč na desno se razteza.
  D = L
  while D < N and Senzor(D + 1): D += 1

  # Poglejmo še, kako daleč na levo se razteza.
  while L > 1 and Senzor(L - 1): L -= 1

  # Snežinka pokriva žice od L do D; izpišimo jo.
  print(D - L + 1)

  # Počakajmo, da izgine.
  while Senzor(L): pass
```

Oglejmo si še malo drugačno rešitev. V zanki bomo šteli, koliko senzorjev trenutno zaznava snežinko. Dokler snežinke ni, je to število 0, nato pa poskoči na velikost snežinke, razen prvič, ko snežinko zaznamo: takrat je namreč mogoče, da se je pojavila po tistem, ko smo levih nekaj žic že pregledali, zato bomo snežinko zaznali na premalo žicah. Ko torej število žic, ki zaznavajo snežinko, naraste nad 0, ga ne izpišemo takoj, ampak šele v naslednji iteraciji. (Spomnimo se, da naloga zagotavlja, da je vsaka snežinka prisotna na žicah dovolj dolgo, da lahko v tem času večkrat pregledamo vse žice). Oglejmo si še implementacijo te rešitve:

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

int main()
{
  enum { N = 100 }; // Število žic.
  for (int prej1 = 0, prej2 = 0; ; )
  {
    // Preštejmo, koliko žic zaznava snežinko.
    int vsota = 0; for (int i = 1; i <= N; ++i) if (Senzor(i)) ++vsota;

    // Ko vsota naraste z 0, jo en korak kasneje izpišemo.
    if (prej1 == 0 && prej2 > 0 && vsota > 0) cout << vsota << endl;

    // Zadnji dve vsoti si zapomnimo.
    prej1 = prej2; prej2 = vsota;
  }
}
```

Še v pythonu:

```
N = 100 # Število žic.

prej1 = 0; prej2 = 0
while True:

  # Preštejmo, koliko žic zaznava snežinko.
  vsota = sum(1 for i in range(N) if Senzor(i + 1))

  # Ko vsota naraste z 0, jo en korak kasneje izpišemo.
  if prej1 == 0 and prej2 > 0 and vsota > 0: print(vsota)

  # Zadnji dve vsoti si zapomnimo.
  prej1 = prej2; prej2 = vsota
```

2. Semafor

Ko prvič pokličemo BeriStevec, nimamo nobene podlage za to, da bi se odločili, katera od številk, ki nam jih je funkcija vrnila, je prava. Ob naslednjem klicu BeriStevec vemo, da je številka na prikazovalniku zdaj za 1 manjša kot ob prvem klicu; če je na primer zdaj na prikazovalniku številka k, je morala biti ob prvem klicu tam številka $k+1$; in ker vemo, da nam BeriStevec vrne med drugim tudi pravo številko, to pomeni, da nam je ob prvem klicu gotovo vrnil $k+1$, ob drugem pa k. Tako so torej zdaj kandidati za pravo številko le tista števila k, ki jih je BeriStevec vrnila ob drugem klicu in za katera je ob prvem klicu vrnila $k+1$.

Podobno razmišljamo tudi v nadaljevanju. Po tretjem klicu so kandidati za pravo številko le tisti k, ki jih je BeriStevec vrnila ob tretjem klicu in za katere je bila vrednost $k+1$ eden od kandidatov po drugem klicu. Tako nadaljujemo in vse bolj klestimo množico kandidatov, dokler ne ostane v njej eno samo število; tisto mora biti potem prava trenutna številka na prikazovalniku in jo lahko izpišemo ter končamo z izvajanjem.

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

extern vector<int> BeriStevec();

int main()
{
  unordered_set<int> kandidati, noviKandidati;
  for (int k : BeriStevec()) kandidati.emplace(k);

  while (kandidati.size() > 1)
  {
    noviKandidati.clear();

    for (int k : BeriStevec())
      // Število k, ki ga kamera trenutno zaznava, je smiselno upoštevati kot kandidata
      // le, če smo prejšnjo sekundo imeli med kandidati število k + 1.
      if (kandidati.find(k + 1) != kandidati.end())
        noviKandidati.emplace(k);

    swap(kandidati, noviKandidati);
  }

  cout << *kandidati.begin() << endl; return 0;
}
```

Ker je možnih števil pri tej nalogi malo, bi lahko za predstavitev množic kandidatov uporabili tudi tabelo 100 logičnih vrednosti, kjer bi za vsako možno število od 0 do 99 pisalo, ali je kandidat ali ni.

Zapišimo našo rešitev še v pythonu:

```
kandidati = set(BeriStevec())

while len(kandidati) > 1:
  # Izmed števil k, ki jih vrne BeriStevec v naslednjem klicu,
  # obdržimo le tista, za katera je bil k + 1 kandidat po prejšnjem klicu.
  kandidati = set(k for k in BeriStevec() if k + 1 in kandidati)

# Izpišimo edinega preostalega kandidata.
print(list(kandidati)[0])
```

3. Iskanje kvadrata

Število vrstic označimo z n_v, število stolpcev pa z n_s. Naj bo s_i vsota širin prvih i stolpcev, v_j pa vsota višin prvih j vrstic. Tega dvojega ni težko izračunati iz zaporedij širin stolpcev in višin vrstic, ki ju dobimo kot vhodna podatka. Pravokotnik, ki ga iščemo, je potem v vsakem primeru velikosti $s_i \times v_j$ za neki dve števili i in j; vprašanje pa je, kako izbrati i in j, da bo ta pravokotnik čim bolj kvadraten.

Recimo, da se za hip omejimo na pravokotnike višine v_j, torej take, ki se raztezajo čez prvih j vrstic. Vzemimo najmanjši tak i, pri katerem je $s_i \geq v_j$. Potem so pravokotniki $s_k \times v_j$ za $k \geq i$ vsaj tako široki kot visoki in razmerje dolžine med daljšo in krajšo stranico je pri njih enako s_k/v_j; najbližje 1 je takrat, ko je s_k najmanjša, to pa je pri $k = i$. Med vsemi temi pravokotniki je torej kandidat za najboljšega le $s_i \times v_j$.

Podobno so pravokotniki $s_k \times v_j$ za $k < i$ vsaj tako visoki kot široki, zato je razmerje dolžine med daljšo in krajšo stranico pri njih enako v_j/s_k; najbližje 1 je takrat, ko je s_k največja, to pa je pri $k = i - 1$. Med vsemi temi pravokotniki je torej kandidat za najboljšega le $s_{i-1} \times v_j$.

Tako torej vidimo, da sta med vsemi pravokotniki oblike $s_k \times v_j$ (pri nekem konkretnem j) za nas zanimiva le dva, namreč tista za $k = i$ in $k = i - 1$. Ker vnaprej ne vemo, pri katerem j bomo dobili najboljši rezultat, bomo preizkusili vse možne j od 1 do števila vrstic. Recimo torej, da v zanki počasi povečujemo j za 1; kako se pri tem spreminja i? Spomnimo se, da smo i definirali kot najmanjše število stolpcev, pri katerem je $s_i \geq v_j$. Ko se j poveča za 1, se tudi v_j poveča (za višino naslednje vrstice), zato je pogoj $s_i \geq v_j$ še strožji kot prej; vsak tak s_i, ki je bil že prej premajhen, je zdaj še bolj premajhen; morda bo isti s_i kot doslej še vedno dovolj velik, drugače pa ga bo treba še povečati. Torej, ko se j poveča za 1, se lahko i poveča ali pa ostane enak, ne more pa se zmanjšati.

Zapišimo ta postopek s psevdokodo:

$i := 1$;
for $j := 1$ **to** n_v:
  while $i < n_s$ **and** $s_i < v_j$ **do** $i := i + 1$;
  if $i > 1$ **then** pravokotnik $s_{i-1} \times v_j$ je kandidat za rešitev;
  pravokotnik $s_i \times v_j$ je kandidat za rešitev;

Med vsemi kandidati za rešitev si zapomnimo tistega, ki je najbolj kvadraten. Časovna zahtevnost tega postopka je $O(n_v + n_s)$, saj naredimo n_v iteracij zunanje zanke, iteracij notranje zanke pa je vsega skupaj največ n_s.

Gornji postopek se premakne z j na $j + 1$ pri takem i, za katerega je $s_i \geq v_j$; z drugimi besedami, če je pravokotnik širši kot višji, ga spodaj podaljšamo za eno vrstico (kar je smiselno, kajti če bi ga namesto tega na desni razširili za en stolpec, bi postal še bolj preširok kot prej).[3] In po drugi strani, gornji postopek se premakne z i na $i + 1$ takrat, ko je $s_i < v_j$; z drugimi besedami, če je pravokotnik višji kot širši, ga na desni razširimo za en stolpec (kar je tudi smiselno, kajti če bi ga namesto tega spodaj podaljšali za eno vrstico, bi postal še bolj previsok kot prej).

[3] Po takem premiku z j na $j + 1$ nam tudi ni treba več razmišljati o pravokotnikih širine s_{i-1} (ali manjše): pri j smo imeli $s_{i-1} < v_j \leq s_i$, zdaj pa smo šli na $j + 1$, kjer je $v_j < v_{j+1}$. Torej sta pravokotnika $s_{i-1} \times v_j$ in $s_{i-1} \times v_{j+1}$ oba višja kot širša, zato je bolj kvadraten tisti izmed njiju, ki je nižji, to pa je $s_{i-1} \times v_j$. S pravokotnikom $s_{i-1} \times v_{j+1}$ se torej ni treba ukvarjati.

Naš postopek lahko torej opišemo še preprosteje kot doslej, če rečemo takole: začnemo pri $i = j = 1$; potem pa, če je trenutni pravokotnik preširok, mu dodamo eno vrstico (povečamo j za 1); sicer pa je preozek in mu dodamo en stolpec (povečamo i za 1). Med vsemi tako pregledanimi pravokotniki si zapomnimo tistega, ki je najbolj kvadraten. Oglejmo si implementacijo te rešitve v C++:

```
#include <vector>
#include <utility>
#include <algorithm>

pair<int, int> NajboljsiPravokotnik(const vector<int> &sirine, const vector<int> &visine)
{
  int ns = sirine.size(), nv = visine.size();
  int i = 1, j = 1, si = sirine[0], vj = visine[0];
  int najS = si, najV = vj;
  while (i <= ns || j <= nv)
  {
    // si = vsota prvih i širin; vj = vsota prvih j višin.
    // Če je pravokotnik višji kot širši, ga razširimo za en stolpec.
    if (si <= vj && i < ns) si += sirine[i++];

    // Če pa je širši kot višji, ga spodaj podaljšajmo za eno vrstico.
    else if (vj < si && j < nv) vj += visine[j++];

    // Če ga v izbrani smeri ne moremo povečati, končajmo,
    else break; // saj se lahko drugače rešitev le še poslabša.

    // Če je to najboljša rešitev doslej, si jo zapomnimo. Preveriti moramo
    // torej, ali je max(si, vj) / min(si, vj) < max(najS, najV) / min(najS, najV).
    // Da ne bo treba delati z ne-celimi števili, pomnožimo to
    // neenačbo z imenovalcema obeh ulomkov.
    if (max(si, vj) * min(najS, najV) < max(najS, najV) * min(si, vj))
      najS = si, najV = vj;
  }
  return {najS, najV}; // Vrnimo najboljšo rešitev.
}
```

In v pythonu:

```
def NajboljsiPravokotnik(sirine, visine):
  ns = len(sirine); nv = len(visine)
  i = 1; j = 1; si = sirine[0]; vj = visine[0]
  najS = si; najV = vj

  while i <= ns or j <= vj:
    # si = vsota prvih i širin; vj = vsota prvih j višin.
    # Če je pravokotnik višji kot širši, ga razširimo za en stolpec.
    if si <= vj and i < ns: si += sirine[i]; i += 1

    # Če pa je širši kot višji, ga spodaj podaljšajmo za eno vrstico.
    elif vj < si and j < nv: vj += visine[j]; j += 1

    # Če ga v izbrani smeri ne moremo povečati, končajmo,
    else: break # saj se lahko drugače rešitev le še poslabša.

    # Če je to najboljša rešitev doslej, si jo zapomnimo. Preveriti moramo
    # torej, ali je max(si, vj) / min(si, vj) < max(najS, najV) / min(najS, najV).
    # Da ne bo treba delati z ne-celimi števili, pomnožimo to
    # neenačbo z imenovalcema obeh ulomkov.
    if max(si, vj) * min(najS, najV) < max(najS, najV) * min(si, vj):
```

```
      najS = si; najV = vj
  return najS, najV # Vrnimo najboljšo rešitev.
```

4. Neprevidni poeti

Za posamezno vrstico lahko določimo njen ritem tako, da jo beremo znak po znak; če je trenutni znak samoglasnik in velika črka, dodamo v niz, ki predstavlja ritem, znak `'-'`; če je trenutni znak samoglasnik in mala črka, dodamo v ritem znak `'U'`; vse ostale znake besedila pa lahko ignoriramo.

Pri prvi vrstici moramo ritem le izračunati in si ga zapomniti (v spodnji rešitvi ga shranimo v spremenljivko ritem1). Pri vsaki naslednji vrstici pa njen ritem primerjamo s tistim iz prve vrstice; če pride do neujemanja, lahko takoj končamo, saj vemo, da bo rezultat na koncu `NE`. Če pa pridemo do konca vhodnih podatkov, ne da bi opazili kakšno neujemanje, izpišemo `DA` in ritem prebranih vrstic.

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

int main()
{
  string ritem, ritem1;      // Ritem trenutne in prve vrstice.
  bool vseEnake = true;      // Ali imajo vse doslej prebrane vrstice enak ritem?
  while (true)
  {
    int c = getchar();

    // Ko preberemo samoglasnik, dodamo U ali - v ritem trenutne vrstice.
    if (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U') ritem += '-';
    else if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') ritem += 'U';

    // Ali smo na koncu vrstice?
    else if (c == '\n' || c == EOF)
    {
      // Ritem prve vrstice si zapomnimo.
      if (ritem1.empty()) ritem1 = move(ritem);

      // Pri ostalih primerjamo ritem trenutne s prvo; prazne preskočimo.
      else if (! ritem.empty() && ritem != ritem1) { vseEnake = false; break; }

      if (c == EOF) break;
      ritem.clear(); // Pripravimo se na naslednjo vrstico.
    }
  }

  // Izpišimo rezultat.
  printf("%s\n", vseEnake ? "DA" : "NE");
  if (vseEnake) printf("%s\n", ritem1.c_str());
  return 0;
}
```

Pogoj pri naglašenih samoglasnikih bi lahko tudi poenostavili v „`if ('A' <= c && c <= 'Z')`“, saj besedilo naloge zagotavlja, da se velike črke v vhodnih podatkih drugače ne bodo pojavljale.

Oglejmo si še primer rešitve v pythonu. Vhodne podatke bomo brali kar po vrsticah, saj naloga zagotavlja, da niso pretirano dolge:

```
import sys

ritem1 = ""          # Ritem prve vrstice.
vseEnake = True      # Ali imajo vse doslej prebrane vrstice enak ritem?

for vrstica in sys.stdin:
    ritem = []
    for c in vrstica:
        # Ko preberemo samoglasnik, dodamo U ali - v ritem trenutne vrstice.
        if c in "AEIOU": ritem.append('-')
        elif c in "aeiou": ritem.append('U')

    # Ritem predelamo iz seznama v niz.
    ritem = "".join(ritem)

    # Ritem prve vrstice si zapomnimo.
    if not ritem1: ritem1 = ritem

    # Pri ostalih primerjamo ritem trenutne vrstice s prvo.
    elif ritem != ritem1: vseEnake = False; break

# Izpišimo rezultat.
print("DA" if vseEnake else "NE")
if vseEnake: print(ritem1)
```

Za ljubitelje pretiravanja z regularnimi izrazi: ritem bi lahko računali tudi tako, da bi iz vrstice pobrisali vse minuse, nato spremenili velike samoglasnike v minuse, nato male samoglasnike v U-je in končno pobrisali vse ostale znake.

```
import re
ritem = re.sub(r"[^U\-]", "", re.sub("[aeiou]", "U", re.sub("[AEIOU]", "-",
                                                    vrstica.replace("-", ""))))
```

5. Stonoge

Polje lahko predstavimo kot seznam (ali tabelo ali vektor) nizov, pri čemer vsak niz predstavlja eno vrstico polja. V zanki pregledujmo znake, dokler ne najdemo znaka `#`, ki kaže, da se tam začenja trup stonoge. Zapomnimo si njegov položaj in se v zanki premaknimo naprej mimo vseh znakov `#`, ki mu sledijo, dokler ne najdemo konca stonoge. Zdaj vemo, kje se trup začne in konča, in gremo lahko v še eni zanki po vseh parih nog (ne pozabimo tudi na tista tik pred in tik za trupom); za vsak par nog preverimo, če sta simetrični (torej je lahko ena `\` in druga `/` ali pa sta obe `|`), pri prvem in zadnjem paru pa še, če sta usmerjeni k trupu. Če je z nogami vse v redu, povečamo števec prepričljivih stonog, sicer pa števec umetnih.

V prvi in zadnji vrstici ter v prvem in zadnjem stolpcu nam trupov (znakov `#`) načeloma ni treba iskati, saj naloga zagotavlja, da nobena stonoga ni v kadru le delno. To pa potem tudi pomeni, da ko ob trupu pregledujemo noge, nam ni treba skrbeti, da bi pri tem poskušali dostopati do znakov z neveljavnimi indeksi (onkraj roba mreže).

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

void Preglej(const vector<string>& a, int &prepricljive, int &umetne)
{
    int h = a.size(), w = a[0].length(); prepricljive = 0; umetne = 0;
```

```
  for (int y = 1; y < h − 1; ++y) for (int x = 1; x < w − 1; ++x) if (a[y][x] == '#')
  {
    // Poglejmo, do kod gre trup te stonoge.
    int xx = x + 1; while (xx < w − 1 && a[y][xx] == '#') ++xx;
    // Trup stonoge obsega znake od a[y][x] do a[y][xx − 1].
    // Preverimo, ali so noge simetrične in usmerjene k trupu.
    bool ok = true;
    for (int u = x − 1; u <= xx; ++u) {
      char zgoraj = a[y − 1][u], spodaj = a[y + 1][u];
      if (! (zgoraj == '\\' && spodaj == '/' && u < xx ||
              zgoraj == '/' && spodaj == '\\' && u >= x ||
              zgoraj == '|' && spodaj == '|' && u >= x && u < xx))
        { ok = false; break; } }
    // Povečajmo ustreznega izmed števcev.
    if (ok) ++prepricljive; else ++umetne;
    x = xx; // Nadaljujmo desno od te stonoge.
  }
}
```

Še enaka rešitev v pythonu:

```
def Preglej(a):
  h = len(a); w = len(a[0]); prepricljive = 0; umetne = 0
  for y in range(1, h − 1):
    x = 1
    while x < w − 1:
      if a[y][x] != '#': x += 1; continue
      # Pri x se začenja trup stonoge. Kje se konča?
      xx = x
      while xx < w − 1 and a[y][xx] == '#': xx += 1
      # Trup stonoge obsega znake od a[y][x] do a[y][xx − 1].
      # Preverimo, ali so noge simetrične in usmerjene k trupu.
      ok = True
      for u in range(x − 1, xx + 1):
        zgoraj = a[y − 1][u]; spodaj = a[y + 1][u]
        if not (zgoraj == '\\' and spodaj == '/' and u < xx or
          zgoraj == '/' and spodaj == '\\' and u >= x or
          zgoraj == '|' and spodaj == '|' and x <= u < xx):
          ok = False; break
      # Povečajmo ustreznega izmed števcev.
      if ok: prepricljive += 1
      else: umetne += 1
      x = xx + 1 # Nadaljujmo desno od te stonoge.
  return (prepricljive, umetne)
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Varnostno kopiranje

Naloga med drugim pravi, da če se dve poti nanašata na isti direktorij, moramo eno od njiju zavreči. Taki dve poti nista nujno čisto enaki; lahko se razlikujeta po tem, da ima ena na koncu poševnico `/`, druga pa ne (na primer: `/ab/cd` in `/ab/cd/` predstavljata isti direktorij). Za lažje preverjanje tega pogoja je torej koristno, če tistim potem, ki se v vhodnih podatkih ne končajo na poševnico, le-to dodamo.

S tako dopolnjenimi potmi pa je potem laže preverjati tudi drugi pogoj, namreč ali ena pot predstavlja poddirektorij druge. To je namreč res natanko tedaj, ko je druga pot prefiks prve (ali, z drugimi besedami: ko se prva pot začne na drugo). Če na primer pogledamo poti `/ab/cd/` in `/ab/`, vidimo, da je druga prefiks prve (niz `"/ab/cd/"` se začne na niz `"/ab/"`), zato prva predstavlja poddirektorij druge in jo lahko zavržemo. To, da smo potem dodali znak `/` na koncu, kjer ga še ni bilo, je koristno zato, ker bi drugače na primer pri `/ab/cd` in `/ab/c` videli, da je drugi niz prefiks prvega, čeprav prvi niz ne predstavlja poddirektorija drugega.

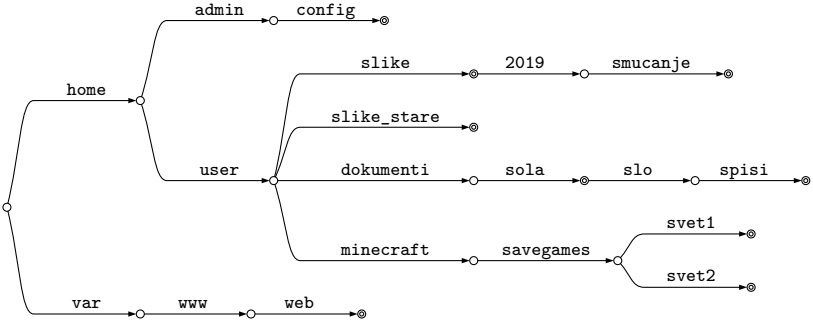
Oba pogoja lahko zdaj zelo preprosto preverjamo tako, da poti uredimo leksikografsko. Če je v vhodnem seznamu več enakih poti, bodo tako prišle skupaj in bomo odvečne zlahka zavrgli; poti za poddirektorije pa bodo prišle takoj za njihove naddirektorije in jih tudi ne bo težko opaziti. Pri izpisu pa moramo paziti na to, da naloga zahteva, da morajo biti izpisane poti podmnožica vhodnih; to pomeni, da če je bila neka pot v vhodu prisotna brez poševnice na koncu, jo moramo tudi mi izpisati brez poševnice (razen če je bila pot do istega direktorija nekje drugje v vhodnem seznamu zapisana tudi *s* poševnico in se odločimo izpisati to različico namesto tiste brez poševnice). V ta namen je koristno ob vsaki poti hraniti še podatek o tem, ali je poševnico na koncu imela že v vhodnem seznamu ali smo ji jo dodali šele mi.

```
#include <vector>
#include <string>
#include <cstring>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <utility>

int main()
{
  ifstream ifs("poti.txt");
  vector<pair<string, bool>> poti;
  while (true)
  {
    // Preberimo naslednjo pot.
    string pot; if (! getline(ifs, pot)) break;

    // Če nima poševnice na koncu, jo dodajmo.
    bool dodaj = (pot.back() != '/');
    if (dodaj) pot += '/';

    // Dodajmo pot na seznam.
    poti.emplace_back(pot, dodaj);
  }
```

```
  sort(poti.begin(), poti.end());
  string zadnjalzpisana;
  for (auto &[pot, dodaj] : poti)
  {
    // Če smo že izpisali kakšno pot in če je zadnja izpisana pot prefiks trenutne,
    // to pomeni, da se trenutna nanaša na isti direktorij ali pa na neki njegov poddirektorij.
    if (! zadnjalzpisana.empty() && strncmp(zadnjalzpisana.c_str(), pot.c_str(),
                                            zadnjalzpisana.length()) == 0) continue;
    // Sicer moramo trenutno pot izpisati.
    // Pred izpisom pa s konca pobrišimo poševnico, če smo jo prej dodali.
    zadnjalzpisana = pot;
    if (dodaj) pot.pop_back();
    cout << pot << endl;
  }
  return 0;
}
```

Za preverjanje, ali je ena pot prefiks druge, smo uporabili `strncmp` iz C-jeve standardne knjižnice; od C++20 naprej pa lahko uporabimo `pot.starts_with(zadnjalzpisana)`.

Zaradi urejanja nizov je časovna zahtevnost te rešitve $O(n \log n)$, če imamo na vhodu n nizov. Za namen naše naloge je to dovolj dobro, vseeno pa si oglejmo še rešitev s časovno zahtevnostjo $O(n)$. Poti lahko pri znakih / razrežemo na komponente in jih zložimo v drevo (*trie*), kot kaže naslednja slika:



Pri dodajanju novih poti v drevo pazimo na to, da če se več poti ujema v prvih nekaj komponentah, si delijo tudi ustrezna vozlišča v drevesu. V vsakem vozlišču si tudi označimo, ali se pri njem konča kakšna pot iz vhodnega seznama (na sliki so taka vozlišča označena z dvojnimi krožci) oz. na katerem mestu v vhodnem seznamu se nahaja. Na koncu se moramo le sprehoditi po drevesu in izpisati poti pri tistih vozliščih, za katera noben njihov prednik ni že sam prisoten v vhodnem seznamu.

Oglejmo si implementacijo takšne rešitve v pythonu:

```
# Preberimo vhodno datoteko.
with open("poti.txt", "rt") as f: poti = f.readlines()
# Pripravimo drevo, v katerem je zaenkrat le koren.
koren = 0; stars = [−1]; otrok = {}; odKod = [−1]
# Dodajmo v drevo vse vhodne poti.
for i, pot in enumerate(poti):
```

```
    u = koren
    for s in pot.rstrip().split('/'):
        # Preskočimo prazno komponento pred poševnico na začetku niza
        if not s: continue                    # in za tisto na koncu.
        # Premaknimo se iz trenutnega vozlišča u dol v otroka, do katerega
        # pelje povezava z oznako s.
        v = otrok.get((u, s), -1)
        if v < 0: # Če takega otroka še ni, ga dodajmo.
            v = len(stars); stars.append(u); odKod.append(-1)
            otrok[u, s] = v
        u = v
    odKod[u] = i # Zapomnimo si, katera vhodna pot se konča pri tem vozlišču.
# Izpišimo rezultate.
for u in range(len(stars)):
    if odKod[u] < 0: continue
    # Pri vozlišču u se konča ena od vhodnih poti.
    # Ali se konča tudi pri kakšnem njegovem predniku?
    v = stars[u]
    while v >= 0 and odKod[v] < 0: v = stars[v]
    # Če ne, lahko trenutno pot izpišemo.
    if v < 0: print(poti[odKod[u]].rstrip())
```

Vozlišča so oštevilčena z zaporednimi številkami od 0 naprej (0 je koren); `stars[u]` nam pove, kdo je starš vozlišča `u`; `otrok[u, s]` pa je tisti otrok vozlišča `u`, do katerega pelje povezava, označena z nizom `s`. Vrednost `odKod[u]` je -1, če se pri `u` ne konča nobena vhodna pot, sicer pa je `odKod[u]` indeks te poti (oz. zadnje od njih, če jih je več) v vhodnem seznamu — to pride prav pri izpisu.

Z vidika števila vhodnih poti n ima ta rešitev časovno zahtevnost $O(n)$; bolj pošteno pa bi bilo reči, da je njena časovna zahtevnost $O(d)$, če je d skupna dolžina vhodnih poti, kajti v najslabšem primeru bo imelo drevo $O(d)$ vozlišč. Boljše časovne zahtevnosti od te pa niti ne moremo pričakovati, saj mora vsaka rešitev porabiti $O(d)$ časa že samo zaradi branja vhodne datoteke.

2. Luči

V enem koraku lahko spremenimo stanje nekaj zaporednih luči. Opazimo lahko, da je vseeno, v kakšnem vrstnem redu izvajamo te operacije, kajti končno stanje posamezne luči je odvisno le od tega, koliko operacij je vplivalo nanjo: če jih je bilo sodo mnogo, bo končno stanje luči enako začetnemu, sicer pa nasprotno od začetnega.

Recimo zdaj, da na neko luč x vplivata dve različni operaciji, na primer ena, ki spremeni luči od a-te do b-te, in druga, ki spremeni luči od c-te do d-te. Ker obe tidve operaciji vplivata na luč x, gotovo velja $a \le x \le b$ in $c \le x \le d$. Vzemimo $A = \min\{a, c\}$, $B = \max\{a, c\}$, $C = \min\{b, d\}$ in $D = \max\{b, d\}$. Učinek naših dveh operacij je torej takšen: na luči od A do $B - 1$ vpliva le ena operacija; na tiste od $C + 1$ do D le druga; na tiste od B do C pa obe. Toda če na isto luč vplivata dve operaciji, bo druga postavila to luč nazaj v tisto stanje, v katerem je bila pred prvo operacijo, torej je učinek enak, kot če na tako luč ne bi vplivala nobena operacija. Lahko bi torej naši dve operaciji spremenili tako, da bi prva delovala na luči od A

do $B-1$, druga pa na luči od $C+1$ do D, pa bo učinek enak kot doslej, le da zdaj na nobeno luč ne bosta vplivali obe operaciji. (Če je $A=B$, lahko prvo od teh dveh operacij celo sploh pobrišemo; in podobno za drugo, če je $C=D$.)

S tem razmislekom lahko postopoma odpravimo vse primere, ko sta na kakšno luč vplivali (vsaj) dve različni operaciji, ne da bi se pri tem število operacij kaj povečalo. Torej optimalne rešitve ne bomo spregledali, če se bomo že od začetka omejili na takšna zaporedja operacij, pri katerih vsako luč spremeni kvečjemu ena operacija.

Če primerjamo, kje se istoležni znaki začetnega in končnega stanja luči razlikujejo, bomo videli, katerim lučem je treba stanje spremeniti. Vsaka operacija lahko poskrbi za največ eno strnjeno skupino takih luči. Najmanjše število operacij dobimo torej tako, da imamo za vsako táko strnjeno skupino po natanko eno operacijo.

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

void Luci(int n, const char *zacetno, const char *koncno)
{
  for (int j = 0; j < n; ++j)
  {
    int i = j; while (j < n && zacetno[j] != koncno[j]) ++j;
    // Luči od vključno i do vključno j − 1 se morajo spremeniti, luč j pa ne več
    // (zato lahko v naslednji iteraciji zunanje zanke nadaljujemo pri j + 1).
    // Pri izpisu uporabimo števila od 1 do n namesto od 0 do n − 1.
    if (i < j) cout << (i + 1) << " " << j << endl;
  }
}
```

Oglejmo si še malo drugačno, a enako dobro rešitev. Recimo, da imamo k strnjenih skupin luči, ki jim je treba spremeniti stanje, pri čemer i-ta skupina (gledano od leve proti desni) pokriva luči od a_i do b_i. Z eno operacijo lahko spremenimo stanje vseh luči od a_1 do b_n; s tem pride vseh prej omenjenih n skupin luči v pravo stanje, vendar pa so zdaj v napačnem stanju luči od b_1+1 do a_2-1, pa od b_2+1 do a_3-1 in tako naprej. Tako imamo torej zdaj $k-1$ strnjenih skupin luči, ki jim je treba spremeniti stanje; če nadaljujemo na enak način kot prej, bomo sčasoma dobili rešitev s k operacijami, torej enako dobro kot pri naši prvi rešitvi. Tako imamo torej rešitev, ki na vsakem koraku poišče najbolj levo in najbolj desno luč, ki jo je treba spremeniti, in z eno operacijo spremeni njiju in vse luči med njima:

```
void Luci2(int n, const char *zacetno, const char *koncno)
{
  bool razlicna = true; int i = 0, j = n − 1;
  while (i <= j)
  {
    // Pomaknimo i v desno in j v levo do prvega mesta, kjer se začetno in končno
    // stanje razlikuje oz. ujema (odvisno od spremenljivke „razlicno“).
    while (i <= j && ((zacetno[i] != koncno[i]) != razlicna)) ++i;
    while (i <= j && ((zacetno[j] != koncno[j]) != razlicna)) −−j;
    if (i > j) break;
    // Spremenimo stanje luči od vključno i do vključno j.
    cout << (i + 1) << " " << (j + 1) << endl;
    ++i; −−j; razlicna = !razlicna;
```

```
  }
}
```

3. Planinarjenje

Recimo, da nas, tako kot na sliki v besedilu naloge, zanima relativna višina vrha A. Pojdimo od A v desno, dokler ne naletimo na prvi višji vrh, recimo $d(A)$; in naj bo D_A najnižja dolina na tako prehojeni poti. Podobno pojdimo od A še v levo do prvega višjega vrha, recimo $\ell(A)$, in naj bo L_A najnižja dolina na tako prehojeni poti. Vzemimo za K_A višjo izmed dolin L_A in D_A. Če voda naraste do višine K_A, bo vrh A ločen tako od levega višjega vrha kot od desnega, torej bo najvišji na svojem otoku; po drugi strani, če voda naraste do kakšne nižje višine, bo A še vedno na istem otoku kot vsaj eden od omenjenih višjih vrhov (namreč tisti, pri katerem je najnižja dolina med A in njim na višini K_A). Torej je K_A ravno tista višina, o kateri govori naloga pri definiciji topografske prominence; topografska prominenca vrha A je razlika med njegovo višino in višino doline K_A.

Če morda desno od A-ja sploh ni nobenega višjega vrha, nas torej desna stran nič ne omejuje glede tega, kako visoko mora voda narasti, preden bo A najvišji vrh svojega otoka; to lahko zajamemo v gornji razmislek tako, da za D_A vzamemo točko višine 0 na koncu vhodnih podatkov. Podobno tudi za L_A, če levo od A-ja ni nobenega višjega vrha, vzamemo točko višine 0 na začetku vhodnih podatkov.

Vprašanje je torej zdaj, kako za vsak vrh A poiskati naslednji višji vrh v vsaki smeri (levo in desno) ter najnižjo dolino na poti do njega. Tega seveda ne bi bilo težko početi z zanko, toda če bomo takšno zanko izvedli za vsak vrh A, bo imel naš postopek na koncu časovno zahtevnost $O(n^2)$, pri čemer n pomeni dolžino vhodnega zaporedja:

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

void Planinarjenje1(const vector<int> &v)
{
  int n = v.size();
  for (int i = 1; i < n; i += 2)
  {
    int A = v[i];
    // Pojdimo v levo do naslednjega višjega vrha in si zapomnimo najnižjo dolino na poti.
    int L = A; for (int j = i; j >= 0 && v[j] <= A; --j) L = min(L, v[j]);
    // Nato naredimo enako še v desno.
    int D = A; for (int j = i; j < n && v[j] <= A; ++j) D = min(D, v[j]);
    // Izračunajmo topografsko prominenco vrha A in jo izpišimo.
    cout << (A - max(L, D)) << endl;
  }
}
```

Z malo pazljivosti pa lahko z enim samim prehodom čez podatke, v $O(n)$ časa, določimo *vsem* vrhovom A najbližji vrh na desni ter najnižjo dolino na poti do njega. Tako bomo za vsak A dobili njegov D_A, nato pa bomo podobno pregledali vhodno zaporedje še v nasprotni smeri in za vsak A dobili še njegov L_A.

Recimo, da smo trenutno pri vrhu A; naj bo $\mathcal{D}(A)$ zaporedje vrhov $d(A), d(d(A))$ in tako naprej — vsak je višji od prejšnjega in leži desno od njega. Prvi od njih je ravno $d(A)$, ki nas zanima. Recimo zdaj, da se od A premaknemo levo do naslednjega vrha, na primer B. Zdaj nas zanima $\mathcal{D}(B)$; ali ga lahko poceni dobimo iz $\mathcal{D}(A)$? Če je B nižji od A, bo $d(B) = A$, zato dobimo $\mathcal{D}(B)$ iz $\mathcal{D}(A)$ tako, da slednjemu na levem koncu dodamo še A. Če pa je B vsaj tako visok kot A, potem mora $d(B)$ ležati desno od A; in ker bo $d(B)$ višji od B, bo višji tudi od A; prvi primerni kandidat za $d(B)$ je torej kar $d(A)$. Če je tudi ta prenizek, bo naslednji primerni kandidat šele $d(d(A))$ in tako naprej. Zaključimo torej lahko, da dobimo $\mathcal{D}(B)$ iz $\mathcal{D}(A)$ v vsakem primeru tako, da slednjemu na levem koncu dodamo A in nato z levega konca pobrišemo vse vrhove, ki niso višji od B. Ker moramo ves čas brisati in dodajati na levem koncu zaporedja $\mathcal{D}$, je primerna podatkovna struktura za predstavitev tega zaporedja sklad, pri čemer vrh sklada predstavlja levi konec zaporedja. Tako bo prav na vrhu sklada vedno ravno tisti vrh, ki ga potrebujemo za $d(A)$. Ko se premikamo od desne proti levi po vhodnem zaporedju, dodamo vsak vrh enkrat na sklad in ga največ enkrat pobrišemo s sklada, zato je časovna zahtevnost vseh teh operacij skupaj le $O(n)$.

Res pa je, da nas pravzaprav ne zanima toliko $d(A)$ sam po sebi, pač pa najnižja dolina med njim in A — to je D_A, ki ga potrebujemo pri določanju topografske prominence A-ja. Na skladu je torej koristno ob vsakem vrhu hraniti še najnižjo dolino na poti od tega vrha do naslednjega višjega vrha (tistega, ki je na skladu eno mesto pod trenutnim vrhom). Ko pobiramo vrhove s sklada, lahko spotoma še računamo minimalno višino pripadajočih dolin, pa bomo na koncu dobili najnižjo dolino med A in $d(A)$.

Oglejmo si implementacijo te rešitve v C++. V prvem prehodu pregledamo vhodno zaporedje od desne proti levi, računamo D_A-je in jih shranjujemo v vektor D; v drugem prehodu pa pregledamo vhodno zaporedje od leve proti desni, računamo L_A-je in skupaj z D_A-ji (shranjenimi iz prvega prehoda) še topografske prominence, ki jih tudi sproti izpisujemo.

```
#include <stack>

vector<int> Planinarjenje2(const vector<int> &v)
{
  struct Par { int vrh, dno; };
  int n = v.size(); vector<int> D(n / 2);
  for (int prehod = 1; prehod <= 2; ++prehod) {
    // V prvem prehodu gremo od desne proti levi, v drugem pa od leve proti desni.
    stack<Par> S; S.push({−1, 0});
    for (int j = 1; j < n; j += 2) {
      int i = prehod == 2 ? j : n − 1 − j;
      int A = v[i], C = v[prehod == 2 ? i − 1 : i + 1];
      // Trenutni vrh je A, pred njim je dolina C. Pojdimo nazaj po zaporedju
      // do naslednjega višjega vrha, v C pa računajmo najnižjo dolino na tej poti.
      while (C > 0 && S.top().vrh <= A) {
        C = min(C, S.top().dno); S.pop(); }
      // Dodajmo na sklad trenutni vrh ter najnižjo dolino na poti do
      S.push({A, C});                    // naslednjega višjega vrha.
      // Pri prvem prehodu (od desne proti levi) si C le zapomnimo.
```

```
        if (prehod == 1) D[i / 2] = C;
        // Pri drugem prehodu (od leve proti desni) pa lahko že izračunamo topografsko
        else cout << A - max(D[i / 2], C) << endl; } }                    // prominenco.
}
```

4. Sedežni red

Naloga pravi, da je število stolpcev m sodo; recimo torej, da je $m = 2k$. V posamezno vrsto lahko torej postavimo največ k dečkov, saj bi drugače neizogibno morala sedeti dva skupaj v isti vrsti. Enako velja tudi za deklice. Če je torej enih in/ali drugih več kot $k \cdot n$, lahko takoj zaključimo, da razporeda, kakršnega zahteva naloga, ni mogoče sestaviti.

Recimo torej zdaj, da je dečkov kvečjemu $k \cdot n$, deklic pa tudi. Potem lahko sestavimo razpored, pri katerem sedijo v lihih stolpcih samo dečki, v sodih pa samo deklice. Dečke poljubno razdelimo v skupine po n, dokler jih pač ne zmanjka (nastane največ k skupin, pri čemer je v zadnji lahko tudi manj kot n otrok). Vsako skupino razporedimo v enega od lihih stolpcev (teh je k, torej vsaj toliko kot skupin), pri čemer seveda otroke v skupini uredimo po višini in jih postavimo tako, da so višji bolj zadaj. Nato enako naredimo še z deklicami.

```
#include <vector>
#include <algorithm>
using namespace std;
struct Otrok { int visina; char spol; };

bool Razporedi(int stVrstic, int stStolpcev, const vector<Otrok> &otroci)
{
  // Preštejmo dečke in deklice posebej.
  int stDeckov = 0, stDeklic = 0;
  for (auto &O : otroci) if (O.spol == 'M') ++stDeckov; else ++stDeklic;

  // Če je enih ali drugih preveč, primernega razporeda ni.
  if (max(stDeckov, stDeklic) > stVrstic * (stStolpcev / 2)) return false;

  // Razdelimo jih v stolpce.
  vector<vector<Otrok>> stolpci(stStolpcev); stDeckov = 0; stDeklic = 0;
  for (auto &O : otroci)
    if (O.spol == 'M') stolpci[2 * (stDeckov++ / stVrstic)].push_back(O);
    else stolpci[2 * (stDeklic++ / stVrstic) + 1].push_back(O);

  // Vsak stolpec uredimo padajoče po višini.
  for (auto &stolpec : stolpci) sort(stolpec.begin(), stolpec.end(),
    [] (const auto &x, const auto &y) { return x.visina > y.visina; });

  // Izpišimo rezultate (od zadnjih vrstic proti sprednjim).
  for (int y = 0; y < stVrstic; ++y)
    for (int x = 0; x < stStolpcev; ++x) {
      if (stolpci[x].size() <= y) printf("    ");
      else printf("%3d%c", stolpci[x][y].visina, stolpci[x][y].spol);
      putchar(x == stStolpcev - 1 ? '\n' : ' '); }

  return true;
}
```

Še en način, da pridemo do primernega razporeda, pa je naslednji: uredimo vse dečke po višini, nato pa v zadnjo vrsto posedemo najvišjih k dečkov (v lihe stolpce),

v predzadnjo vrsto naslednjih k po višini in tako naprej. Nato naredimo enako še z deklicami, le da jih pošiljamo v sode stolpce. Manjša slabost pri tej rešitvi je, da moramo urejati do $k \cdot n$ otrok naenkrat (vse dečke ali vse deklice), pri prejšnji pa smo jih urejali le po n naenkrat (vsak stolpec posebej).

5. Žabe

Razdaljo med točkama (x, y) in (x', y') označimo z $d(x, y, x', y') = ((x - x')^2 + (y - y')^2)^{1/2}$. Ker se žabe premikajo s hitrostjo ene enote na sekundo, je $d(x, y, x', y')$ tudi čas, v katerem žaba pride od točke (x, y) do (x', y').

Za vsako žabo izračunajmo najkrajši čas, v katerem lahko ujame muho in pride v koordinatno izhodišče — recimo temu T_s za s-to žabo. Da ga izračunamo, pojdimo v zanki po vseh položajih vsakega roja; s-ta žaba lahko ujame muho na j-tem postanku i-tega roja le v primeru, če lahko od svojega začetnega položaja (a_s, b_s) pride do točke $(x_{i,j}, y_{i,j})$ v $t_{i,j}$ ali manj časa (sicer ji bo roj ušel naprej po svoji poti, še preden bo prišla do tja). Če je ta pogoj izpolnjen, gre potem lahko žaba ob času $t_{i,j}$ naprej proti koordinatnemu izhodišču in ga doseže po $d(x_{i,j}, y_{i,j}, 0, 0)$ časa.

Mogoče je tudi, da žaba ne more ujeti nobene muhe, ker je predaleč od vseh položajev vsakega roja; takrat si mislimo zanjo $T_s = \infty$.

Naloga pravi, da je dovolj, če muhe lovi k žab, ostale pa gredo lahko naravnost od svojega začetnega položaja proti koordinatnemu izhodišču. Za takšno pot porabi s-ta žaba $U_s := d(a_s, b_s, 0, 0)$ časa. Množico žab, ki lovijo muhe, označimo z A; čas, ko se vse žabe zberejo v koordinatnem izhodišču, je potem $f(A) = \max\{\max_{s \in A} T_s, \max_{s \notin A} U_s\}$.

Za vsako žabo seveda velja $U_s \leq T_s$, kajti pri U_s potuje ta žaba od začetnega položaja naravnost v koordinatno izhodišče, pri T_s pa naredi vmes še ovinek do kraja, kjer bo ujela muho (in morda tam celo še čaka, da bo roj sploh prišel tja); zato ima pri U_s žaba krajšo pot (trikotniška neenakost) kot pri T_s. To pa pomeni, da ni nobene koristi od tega, da bi muhe lovilo več žab, kot je nujno potrebno, kajti v rešitvi, kjer lovi muhe več kot k žab, lahko kakšno od njih pošljemo naravnost v koordinatno izhodišče, pa se rešitev ne bo nič poslabšala: če žabo s vržemo iz množice A, bo odslej v $f(A)$ prispevala manjšo vrednost U_s namesto večje vrednosti T_s, torej maksimum tega po vseh žabah ne bo nič večji kot prej.

Lahko se torej omejimo na rešitve, kjer muhe lovi natanko k žab. Recimo, da žabe oštevilčimo naraščajoče po T_s, tako da je $T_1 \leq T_2 \leq \ldots \leq T_z$. Potem je najbolje, če na lov pošljemo kar prvih k žab, torej vzamemo $A = \{1, 2, \ldots, k\}$. Prepričajmo se, da je to res. Recimo, da bi obstajala neka še boljša rešitev B. Ker tudi tam lovi muhe le k žab in ker sta množici A in B različni, mora obstajati neka žaba $i \leq k$, ki v A lovi muhe, v B pa ne; in obstajati mora tudi neka žaba $j > k$, ki lovi muhe v B, ne pa v A. Recimo, da bi v B tema dvema žabama zamenjali vlogi, tako da bi i lovila muho, j pa ne; dobimo rešitev $C := B - \{j\} \cup \{i\}$. Zaradi te spremembe porabi žaba j zdaj manj ali enako časa kot prej (U_j namesto T_j), žaba i pa porabi zdaj manj časa (namreč T_i), kot ga je žaba j porabila prej (namreč T_j; spomnimo se, da je $i \leq k < j$, zato je $T_i \leq T_j$). Maksimum porabljenega časa po obeh žabah torej ni zdaj v C nič večji, kot je bil prej v B, ostalim žabam pa se čas sploh ni spremenil. Torej je C vsaj tako dobra rešitev kot B, poleg tega pa se ujema z našo rešitvijo A v eni žabi več kot B. Tako bi lahko nadaljevali in rešitev korak za

korakom spremenili v A, ne da bi se kdaj poslabšala; torej je tudi A bila optimalna rešitev.

Oglejmo si še implementacijo te rešitve v C++:

```
#include <vector>
#include <utility>
#include <algorithm>
#include <limits>
#include <cmath>
using namespace std;

struct Tocka { double x, y; };
struct Postanek { Tocka kje; double kdaj; };

// Izračuna razdaljo med točkama in s tem tudi čas, ki ga žaba porabi za pot med njima.
double D(const Tocka &a, const Tocka &b) {
  double dx = a.x − b.x, dy = a.y − b.y; return sqrt(dx * dx + dy * dy); }

double Zabe(const vector<Tocka>& zabe, const vector<vector<Postanek>>& roji, int k)
{
  const Tocka cilj { 0, 0 };
  vector<pair<double, double>> casi; // pari (T_s, U_s)

  for (const Tocka &zaba : zabe)
  {
    // Izračunajmo najkrajši čas, v katerem lahko ta žaba ujame muho in pride na cilj.
    double T = numeric_limits<double>::infinity();
    for (auto &roj: roji) for (auto &postanek : roj)

      // Ali lahko žaba pravočasno doseže kraj postanka?
      if (D(zaba, postanek.kje) <= postanek.kdaj)

        // Če po postanku nadaljuje pot, kdaj doseže cilj?
        T = min(T, postanek.kdaj + D(postanek.kje, cilj));

    // Izračunajmo še najkrajši čas, v katerem lahko pride žaba na cilj brez lova na muho.
    casi.emplace_back(T, D(zaba, cilj));
  }

  // Uredimo žabe po T_s.
  sort(casi.begin(), casi.end());

  // Prvih k žab bo lovilo muhe in porabilo T_s časa, ostale gredo
  // naravnost na cilj in porabijo U_s časa.
  double rezultat = 0;
  for (int s = 0; s < zabe.size(); ++s)
    rezultat = max(rezultat, s < k ? casi[s].first : casi[s].second);

  return rezultat;
}
```

Namesto da urejamo vse žabe po času T_s, bi lahko žabo s k-tim najmanjšim T_s poiskali s postopkom quickselect (v C++ovi standardni knjižnici ga imamo kot funkcijo `nth_element`). Tako bi se časovna zahtevnost zadnjega dela rešitve zmanjšala z $O(z \log z)$ (zaradi urejanja) na $O(z)$. Vendar pa od te izboljšave ni veliko koristi, saj si lahko predstavljamo, da bo rešitev ponavadi večino časa porabila za zanko na začetku, ki gre za vsako žabo po vseh postankih vseh rojev.

REŠITVE NALOG ZA TRETJO SKUPINO

1. Mastermind

Razmislimo najprej o tem, kako izračunati odgovor $o = (o_1, o_2)$, ki ga dobimo, če pri ugibanju predlagamo zaporedje $u = (u_1, \ldots, u_n)$, pravo zaporedje pa je $z = (z_1, \ldots, z_n)$. Žetonov prave barve na pravem mestu ni težko prešteti (to bo o_1), če gremo v zanki po indeksih od 1 do n in primerjamo istoležne žetone. Če pa pri nekem indeksu i pride do neujemanja (torej če velja $u_i \neq o_i$), je žeton u_i morda prave barve na napačnem mestu. Natančneje povedano: recimo, da v neujemanjih v zaporedju u nastopa $h_u[b]$ žetonov barve b, v zaporedju z pa $h_z[b]$ žetonov barve b. Potem lahko za žetone prave barve na pravem mestu razglasimo $\min\{h_u[b], h_z[b]\}$ žetonov — torej vse take žetone v u-ju, če jih ni več kot v z-ju, sicer pa le toliko, kolikor jih je v z-ju. Ob prvem prehodu čez zaporedji torej računajmo oba „histograma“ (torej tabeli h_u in h_z, ki za vsako barvo povesta število žetonov te barve na neujemajočih se mestih zaporedij u oz. z), nato pa seštejmo $\min\{h_u[b], h_z[b]\}$ po vseh barvah b in tako dobimo drugi del odgovora, število o_2.

Prvi del naloge zahteva, da preštejemo vsa skladna zaporedja; pravzaprav jih bomo tudi nekam shranili, ker bodo prišla prav pri drugem delu naloge. Pojdimo po vseh možnih zaporedjih z (teh je m^n, ker imamo za vsako od n mest po m možnosti, katere barve žeton je tam) in za vsako preverimo, ali bi dobili, če bi bilo to zaporedje pravo, pri dosedanjih ugibanjih ravno tiste odgovore, ki so zapisani v vhodnih podatkih. V ta namen uporabimo še vgnezdeno zanko po dosedanjih ugibanjih; čim pri kakšnem ugibanju vidimo, da bi bil odgovor pri z-ju drugačen od tistega v vhodnih podatkih, lahko nad trenutnim z obupamo in se lotimo naslednjega.[4] Na koncu nam tako nastane množica zaporedij, ki so skladna z vsemi dosedanjimi ugibanji; recimo ji S.

Malo več dela pa je z drugim delom naloge. Pojdimo po vseh m^n možnih zaporedjih p, ki bi jih igralec lahko predlagal v naslednjem ugibanju. Pri različnih kandidatih $z \in S$ bi ob takem ugibanju lahko nastali različni odgovori o; lahko si predstavljamo, da je ugibanje p razbilo množico S na podmnožice oblike $S_{p,o} := \{z \in S : \mathit{odgovor}(p, z) = o\}$. Z vidika igralca se torej, če na ugibanje p dobi odgovor o, množica kandidatov zmanjša s S na $S_{p,o}$. V za igralca najslabšem primeru je torej množica kandidatov po takem ugibanju ravno največja od vseh $S_{p,o}$, tako da mu ostane po ugibanju še $J(p) := \max_o |S_{p,o}|$ kandidatov. Vrednost $J(p)$ si lahko predstavljamo kot „oceno“ ugibanja p; naloga sprašuje, pri koliko različnih p nastopi najmanjša (in s tem najugodnejša) možna ocena, torej $\min_p J(p)$. Če torej za vsak p izračunamo oceno $J(p)$, ni težko določiti najmanjše ocene in tudi šteti, pri koliko p-jih je nastopila. Ocene posameznega p-ja pa tudi ni težko računati: lahko vzdržujemo tabelo z velikostmi množic $S_{p,o}$ za vse možne odgovore o; na začetku postavimo vse te velikosti na 0, nato pa gremo po vseh skladnih zaporedjih $z \in S$,

[4] Vse možne z lahko generiramo z rekurzijo in pri tem do neke mere že sproti preverjamo, ali je trenutno zaporedje sploh še mogoče podaljšati do nečesa, kar bo skladno z vsemi dosedanjimi ugibanji in odgovori nanje. Na primer, morda je bilo pri kakšnem ugibanju že z doslej zgeneriranim delom z-ja toliko ujemanj, da bo na koncu vrednost o_1 gotovo večja od tiste v vhodnih podatkih. Vendar pa pri naših testnih primerih ni nobene potrebe po tovrstnih optimizacijah, saj bo naš program levji delež časa tako ali tako porabil za drugi del naloge.

pri vsakem izračunamo odgovor $o = odgovor(p, z)$ in povečamo v tabeli velikost množice $S_{p,o}$ za 1.

Oglejmo si implementacijo takšne rešitve v C++. Za naštevanje vseh možnih zaporedij lahko uporabimo kar zanko, ki gre po številih od 0 do $m^n - 1$ in vsako od njih sproti pretvori v zaporedje n števil (enako kot pri pretvorbi števila v m-iški številski sestav). Podobno tudi odgovor $o = (o_1, o_2)$ predstavimo s številom $o_1(m + 1) + o_2$. Za štetje velikosti množic $S_{p,o}$ imamo tabelo stKand; ker bo ocena trenutnega p-ja na koncu enaka maksimumu po vseh vrednostih te tabele, lahko nad tem p-jem takoj obupamo, če kakšna vrednost v tabeli preseže najmanjšo dosedanjo oceno; ta drobna optimizacija skrajša čas izvajanja za več kot polovico (je pa program tudi brez nje več kot dovolj hiter).

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

enum { MaxN = 5, MaxM = 8 };
int n, m; // m barv, n žetonov v zaporedju

int Odgovor(int ugibanje, int prava)
{
  // hu[b], hp[b] = število takih žetonov barve b (v „ugibanje“ oz. „prava“),
  // ki niso prave barve na pravem mestu.
  int hu[MaxM] = {}, hp[MaxM] = {};
  int pp = 0, pn = 0; // št. pravih na pravem mestu, pravih na napačnem mestu
  for (int i = 0; i < n; ++i) {
    // Izluščimo naslednji žeton obeh zaporedij.
    int zu = ugibanje % m; ugibanje /= m;
    int zp = prava % m; prava /= m;
    // Preverimo, ali se ujemata; če ne, popravimo oba histograma.
    if (zu == zp) ++pp; else ++hu[zu], ++hp[zp]; }
  // Iz histogramov izračunajmo število pravih barv na napačnem mestu.
  for (int b = 0; b < m; ++b) pn += min(hu[b], hp[b]);
  // Oba odgovora, pp in pn, zapakirajmo v eno samo število.
  return pp * (n + 1) + pn;
}

int main()
{
  // Preberimo dosedanja ugibanja in odgovore.
  int k; cin >> n >> m >> k;
  vector<int> prejsnja(k), odgovori(k);
  for (int i = 0; i < k; ++i) {
    // Preberimo zaporedje in ga zapakirajmo v število u.
    int u = 0;
    for (int j = 0; j < n; ++j) { int b; cin >> b; u = u * m + (b - 1); }
    // Preberimo odgovor.
    int pp, pn; cin >> pp >> pn;
    prejsnja[i] = u; odgovori[i] = pp * (n + 1) + pn; }
  // Poglejmo, katera zaporedja so skladna z vsemi dosedanjimi odgovori.
  vector<int> skladna;
  int stVseh = 1; for (int i = 0; i < n; ++i) stVseh *= m;
```

```
  for (int z = 0; z < stVseh; ++z) {
    int i = 0; while (i < k && Odgovor(z, prejsnja[i]) == odgovori[i]) ++i;
    if (i == k) skladna.push_back(z); }
  // Ocenimo vse možnosti glede naslednjega ugibanja.
  int najOcena = stVseh + 1, stNaj = −1;
  for (int p = 0; p < stVseh; ++p)
  {
    // Če za naslednje ugibanje vzamemo p, nam množica „skladna“ dosedanjih
    // kandidatov z razpade na podmnožice glede na Odgovor(p, z).
    // Ocena p-ja je velikost največje izmed teh množic.
    int ocena = 0, stKand[(MaxN + 1) * (MaxN + 1)] = {};
    for (int z : skladna) {
      ocena = max(ocena, ++stKand[Odgovor(p, z)]);
      if (ocena > najOcena) break; } // Ta z gotovo ne bo dal najboljše ocene.
    // Zapomnimo si najnižjo oceno in to, pri koliko p-jih smo jo dobili.
    if (ocena < najOcena) najOcena = ocena, stNaj = 1;
    else if (ocena == najOcena) ++stNaj;
  }
  // Izpišimo rezultate.
  cout << skladna.size() << endl << stNaj << endl; return 0;
}
```

2. Dolgovi

Kot vidimo iz omejitev v besedilu naloge, lahko testne primere razdelimo na več skupin z vse večjimi in težjimi primeri. Približno tako lahko razmišljamo tudi o rešitvi; začnimo s preprosto rešitvijo za najmanjše primere in potem poglejmo, kaj bo treba v njej izboljšati.

Pri najmanjših testnih primerih je dovolj že imeti nekakšen seznam oz. tabelo, v kateri hranimo imena ljudi, njihove dolgove in podatek o tem, ali so v skupini ali ne. Poleg tega hranimo v neki spremenljivki tudi število ljudi v skupini. Pri vsakem dogodku se sprehodimo po tabeli in popravimo podatke o prisotnosti v skupini (če je trenutni dogodek prihod ali odhod) oz. o dolgovih (če je trenutni dogodek plačilo). Tako imamo rešitev s časovno zahtevnostjo $O(n \cdot q)$.

Potem imamo skupino testnih primerov, kjer je veliko prihodov in odhodov, vendar vedno le po enega človeka naenkrat; ljudi in plačil pa je malo. Zdaj si ne moremo privoščiti, da bi pri vsakem prihodu in odhodu pregledali celo tabelo. Namesto tabele bi lahko uporabili slovar oz. razpršeno tabelo, v kateri bi bili ključi imena, pripadajoče vrednosti pa dolgovi in podatki o pripadnosti skupini; tako bi lahko dodajali in brisali ljudi v $O(1)$ časa. Toda te rešitve ne bi mogli posplošiti na primere s prefiksnimi operacijami, saj so v razpršeni tabeli različna imena z istim prefiksom preveč razmetana naokrog. Namesto tega si raje pripravimo urejeno tabelo vseh n imen, ki se kdajkoli pojavijo v vhodnih podatkih; pri dodajanju ali brisanju posameznega človeka lahko z bisekcijo poiščemo, kje v tabeli se nahaja. Tako porabimo $O(\log n)$ časa za dodajanje ali brisanje posameznega človeka, pri plačilu pa moramo iti še vedno po celi tabeli in porabimo $O(n)$ časa.

Pri naslednji skupini testnih primerov so prihodi in odhodi še vedno le posamični (ne pa prefiksni), vendar je lahko zdaj tudi plačil veliko, zato si ne moremo več privoščiti, da bi šli pri vsakem plačilu po vseh ljudeh (niti po vseh ljudeh v skupini) in jim popravljali dolgove. Opazimo lahko, da je pri posameznem plačilu ta sprememba

dolgá enaka za vse ljudi, ki so takrat v skupini; recimo, da je pri t-tem plačilu ta sprememba enaka $\triangle_t$. Če se je neki posameznik pridružil skupini tik za r-tim plačilom in jo zapustil tik za t-tim, se mu je v tem času dolg spremenil za $\triangle_{r+1} + \triangle_{r+2} + \ldots + \triangle_t$. S tem, ko smo pri naši dosedanji rešitvi pri vsakem plačilu šli po vseh članih skupine in jim popravljali dolgove, smo pri njih pravzaprav postopoma računali takšne vsote po več zaporednih $\triangle_t$. Cenejši način za izračun takšne vsote pa je, da si pripravimo delne (kumulativne) vsote zaporedja $\triangle_t$: naj bo torej s_t vsota prvih t členov zaporedja $\triangle_t$, torej $s_0 = 0$ in $s_t = s_{t-1} + \triangle_t$. Potem lahko vsoto $\triangle_{r+1} + \triangle_{r+2} + \ldots + \triangle_t$ računamo kot $s_t - s_r$; namesto da dolg človeka popravljamo pri vsakem plačilu, ki nastopi, medtem ko je ta človek v skupini, bi torej lahko ob njegovem prihodu odšteli od njegovega dolga s_r in mu nato ob odhodu prišteli s_t. Tako imamo zdaj z vsakim plačilom le $O(1)$ dela, saj moramo le izračunati $\triangle_t$ (deliti znesek plačila s številom ljudi v skupini) in s_t. Posamezno dodajanje ali brisanje še vedno traja $O(\log n)$ časa in skupaj bo časovna zahtevnost takšne rešitve $O(q \log n)$.

V zadnji skupini testnih primerov se pojavijo tudi prefiksna dodajanja in brisanja. Ker imamo imena urejena po abecedi, tvorijo tista, ki se začnejo na neki dani prefiks, strnjeno podzaporedje našega seznama imen; začetek in konec tega podzaporedja lahko poiščemo z bisekcijo. Preveč časa bi nam vzelo, če bi hoteli zdaj vsakemu človeku v tem podzaporedju posebej povečati ali zmanjšati dolg za trenutno vsoto s_t; potrebujemo torej način, da bomo lahko hkrati spremenili dolg več zaporednim ljudem (vsem za enako količino).

Pomagamo si lahko z neke vrste drevesom segmentov; nad tabelo za dolgove posameznih ljudi zgradimo še tabelo za dolgove po dveh, štirih, osmih itd. zaporednih ljudi. Tako imamo skladovnico tabel T_h za $0 \le h \le \lfloor \log_2 n \rfloor$. Tabela T_h ima $\lfloor n/2^h \rfloor$ elementov, pri čemer element $T_h[i]$ vsebuje znesek dolga, ki velja za vse ljudi od $i \cdot 2^h$ do $(i+1) \cdot 2^h - 1$. (Pri tem si mislimo, da so indeksi v tabelah od 0 naprej, pa tudi ljudje so oštevilčeni od 0 do $n-1$ v abecednem vrstnem redu.) Da dobimo pravi dolg osebe i, moramo zdaj sešteti po vseh tabelah tiste elemente, ki to osebo pokrivajo, torej $\sum_h T_h[\lfloor i/2^h \rfloor]$.

Element $T_{h+1}[i]$ pokriva natančno isto skupino ljudi kot elementa $T_h[2i]$ in $T_h[2i+1]$ skupaj. Recimo, da želimo povečati dolgove ljudi od L do vključno $D-1$ za s. Naivna rešitev bi to naredila tako, da bi s prištela elementom $T_0[L], \ldots, T_0[D-1]$. Toda če sta L in D soda, lahko enak učinek dosežemo, če za s povečamo elemente $T_1[L/2], \ldots, T_1[D/2-1]$, torej polovico manj elementov v naslednji tabeli. (Če L ni bil sod, lahko povečamo $T_0[L]$ za s in nato povečamo L za 1, s čimer ta postane sod; in podobno, če D ni bil sod, lahko povečamo $T_0[D-1]$ za s in odtlej zmanjšamo D za 1, s čimer postane sod.) Podobno razmišljamo tudi pri tabeli T_1 in tako naprej; pridemo do naslednjega postopka:

postopek PrefiksniDogodek(L, D, s):
  $h := 0$;
  while $L < D$:
    če je L lih: povečaj $T_h[L]$ za s; $L := L + 1$;
    če je D lih: $D := D - 1$; povečaj $T_h[D]$ za s;
    $L := L/2$; $D := D/2$; $h := h + 1$;

Časovna zahtevnost tega postopka je $O(\log n)$, saj pri vsakem h spremeni največ dva elementa tabele T_h; tako lahko prefiksno dodajanje ali brisanje izvedemo v samo

$O(\log n)$ časa.

Prefiksni dogodki nam zapletejo rešitev še zaradi nečesa drugega: naloga pravi, da pri takem dogodku pridejo oz. odidejo le tisti ljudje, ki se jim ime začne na dani prefiks *in ki so nekoč prej že bili dodani v skupino.* To pomeni, da naš gornji postopek, ki je spremenil dolgove *vseh* ljudi od L do $D-1$, pravzaprav ni bil dober; moral bi spremeniti dolgove samo tistih, ki so bili kdaj prej že v skupini, toda če bi hoteli to preveriti za vsakega od ljudi na tem intervalu, bi postopek spet trajal $O(n)$ časa. Pomagamo si lahko takole: gornji postopek obdržimo brez sprememb, toda vzdržujmo tudi neko tabelo, ki nam za vsakega človeka pove, ali je bil že kdaj v skupini. Naloga zagotavlja, da prvi prihod človeka v skupino gotovo nastopi s posamičnim dodajanjem, ne s prefiksnim; pri posamičnem dodajanju torej preverimo, ali je ta človek že bil v skupini; če ni bil, izračunajmo njegov dosedanji dolg (ki se je nabral zaradi dosedanjih prefiksnih dohodkov in ki je v resnici popolnoma neupravičen, saj se na tega človeka dosedanji prefiksni dohodki niso nanašali) po prej omenjeni formuli $\sum_h T_h[\lfloor i/2^h \rfloor]$ (kjer je i številka človeka, ki zdaj prvič prihaja v skupino) in ga nato odštejmo od $T_0[i]$; tako bo njegov dolg prišel na 0, kar je zdaj tudi njegova prava vrednost.

Naslednji zaplet, povezan s prefiksnimi dogodki, se nanaša na plačila. Ko nekdo plača znesek z, se dolgovi članov skupine povečajo za z/k, kjer je k število ljudi v skupini. Vrednost k moramo torej znati vzdrževati tudi pri prefiksnih dodajanjih in brisanjih. Spomnimo se, da prefiksna operacija ne doda v skupino (ali pobriše iz nje) *vseh* ljudi od L do $D-1$, pač pa le tiste od njih, ki so bili kdaj prej že v skupini. Znati moramo torej hitro izračunati, koliko ljudi s takega intervala je bilo kdaj prej že v skupini. Tudi za to lahko uporabimo segmentno drevo; poleg tabel T_h imejmo še tabele Z_h, pri čemer $Z_h[i]$ pove, koliko ljudi od $i \cdot 2^h$ do $(i+1) \cdot 2^h - 1$ je bilo doslej že v skupini. Ko neki človek i prvič vstopi v skupino, povečamo pri vsakem h element $Z_h[\lfloor i/2^h \rfloor]$ za 1. Postopek PrefiksniDogodek pa je treba dopolniti tako, da kadarkoli poveča neko vrednost $T_h[i]$ za s, mora tudi povečati (če gre za prefiksno dodajanje) ali zmanjšati (če gre za brisanje) k (torej število ljudi v skupini) za $Z_h[i]$.

Še en zaplet pa nastopi na koncu, ko hočemo izpisati rezultate. Za ljudi, ki so takrat še v skupini, smo v segmentnem drevesu sicer zmanjšali njihov dolg za vsoto s_i, ki je veljala ob njihovem prihodu, nismo pa ga še povečali za vsoto s_i, ki velja zdaj na koncu. To moramo torej narediti zdaj, vendar moramo za to vedeti, kateri ljudje so v skupini. V ta namen lahko naše segmentno drevo še dopolnimo: poleg tabel T_h in Z_h imejmo še tabele P_h in O_h s časi prihodov oz. odhodov. Ko naš postopek PrefiksniDogodek poveča vrednost $T_h[i]$ za s, mora zdaj tudi vpisati trenutni „čas“ (zaporedno številko dogodka, pri katerem smo) v element $P_h[i]$ (če gre za prefiksni prihod) oz. $O_h[i]$ (če gre za odhod). Na koncu lahko za vsakega človeka i izračunamo čas zadnjega prihoda kot $\max_h P_h[\lfloor i/2^h \rfloor]$ in podobno čas zadnjega odhoda; če je zadnji prihod kasnejši kot zadnji odhod, je ta človek zdaj še v skupini, sicer pa ne.

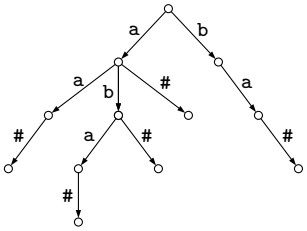
Tako smo dobili rešitev, ki podpira tudi prefiksne dogodke, pri tem pa še vedno porabi le $O(\log n)$ časa za vsak dogodek, skupaj $O(q \log n)$; k temu pa moramo prišteti še $O(n \log n)$ časa za urejanje imen na začetku.

```
#include <vector>
#include <string>
#include <iostream>
```

```
#include <iomanip>
#include <unordered_map>
#include <algorithm>
using namespace std;

int main()
{
  int q; cin >> q;

  // Preberimo vse dogodke in si pripravimo slovar imen, omenjenih v njih.
  struct Dogodek { char op; string ime; int znesek; };
  vector<Dogodek> dogodki(q);
  unordered_map<string, int> slovarImen;
  for (auto &Q : dogodki) {
    string op; cin >> op >> Q.ime; Q.op = op[0];
    if (Q.op == '<') cin >> Q.znesek;
    if (Q.op == '+' && Q.ime.back() != '*') slovarImen.emplace(Q.ime, -1); }

  // Uredimo imena po abecedi in shranimo njihove indekse v slovar.
  int n = slovarImen.size();
  vector<string> imena; imena.reserve(n);
  for (auto &pr : slovarImen) imena.emplace_back(pr.first);
  sort(imena.begin(), imena.end());
  for (int i = 0; i < n; ++i) slovarImen[imena[i]] = i;

  // Pripravimo prazno drevo segmentov.
  struct Vozlisce { int znanih = 0, prihod = -1, odhod = -1; double dolg = 0; };
  vector<vector<Vozlisce>> drevo; drevo.emplace_back(n);
  while (drevo.back().size() > 1) drevo.emplace_back(drevo.back().size() / 2);
  int H = drevo.size() - 1;
  double vsotaDolgov = 0; int stVSkupini = 0; // s_t in k

  // Obdelajmo vse dogodke.
  for (int qi = 0; qi < q; ++qi)
  {
    auto &Q = dogodki[qi]; string &ime = Q.ime;

    if (Q.op == '<') { // Ali je ta dogodek plačilo?
      // Določimo indeks osebe, ki je plačnik pri tem dogodku.
      int L = lower_bound(imena.begin(), imena.end(), ime) - imena.begin();

      // Zmanjšajmo plačnikov dolg in povečajmo kumulativno vsoto dolgov.
      drevo[0][L].dolg -= Q.znesek; vsotaDolgov += double(Q.znesek) / stVSkupini;
      continue; }

    // Sicer je ta dogodek prihod ali odhod.
    bool prefiks = (ime.back() == '*'); if (prefiks) ime.pop_back();
    bool prihod = (Q.op == '+');
    double sprememba = prihod ? -vsotaDolgov : vsotaDolgov;

    // Naslednja funkcija vpiše v vozlišče spremembo dolga in čas zadnjega
    // prihoda ali odhoda ter popravi števec ljudi v skupini.
    auto Popravi = [&] (Vozlisce &v) { v.dolg += sprememba;
      stVSkupini += (prihod ? 1 : -1) * v.znanih; (prihod ? v.prihod : v.odhod) = qi; };

    // Poglejmo, kateri razpon imen pokriva ta dogodek.
    int L = lower_bound(imena.begin(), imena.end(), ime) - imena.begin();
    if (! prefiks) {

      // Če prvič dodajamo to osebo, dosedanji prefiksni dogodki zanjo niso veljali
      // in moramo njen dosedanji dolg postaviti na 0.
      if (prihod && ! drevo[0][L].znanih)
```

```
        for (int h = H; h >= 0; --h) if (int i = L >> h; i < drevo[h].size()) {
          auto &v = drevo[h][i]; ++v.znanih; sprememba -= v.dolg; }
      // Dogodek vpliva le na eno osebo; popravimo dolg v njenem listu.
      Popravi(drevo[0][L]); }
    else {
      int D = lower_bound(imena.begin(), imena.end(),
                          ime + char('z' + 1)) - imena.begin();
      // Prefiksni dogodek vpliva na vse ljudi od vključno L do vključno D - 1.
      // Na vsakem nivoju drevesa moramo popraviti največ dve vozlišči.
      for (int h = 0; h <= H && L < D; ++h, L >>= 1, D >>= 1) {
        if (L & 1) Popravi(drevo[h][L++]);
        if (D & 1) Popravi(drevo[h][--D]); } }
  }
  // Porinimo vse podatke v liste drevesa.
  for (int h = H; h > 0; --h) for (int i = 0; i < 2 * (n >> h); ++i) {
    auto &v = drevo[h - 1][i], &p = drevo[h][i / 2];
    v.prihod = max(v.prihod, p.prihod);
    v.odhod = max(v.odhod, p.odhod); v.dolg += p.dolg; }
  // Izpišimo rezultate.
  for (int i = 0; i < n; ++i) { auto &v = drevo[0][i];
    cout << imena[i] << " " << setprecision(9)
         << v.dolg + (v.prihod > v.odhod ? vsotaDolgov : 0) << endl; }
  return 0;
}
```

Nalogo bi lahko rešili tudi tako, da bi imena vseh ljudi zložili v drevo po črkah (*trie*), pri čemer vsakemu imenu na koncu dodajmo še poseben zaključni znak `#`, ki se drugače v imenih ne pojavlja. Primer kaže naslednja slika, kjer so v drevesu imena `a`, `aa`, `ab`, `aba` in `ba`.



Za vsako vozlišče tega drevesa vzdržujmo tudi vrednosti T, Z, P in O (podobno kot pri prejšnji rešitvi za vsako vozlišče drevesa segmentov). Vsakemu imenu ustreza neki list drevesa (vozlišče, v katero kaže povezava z oznako `#`); ne-prefiksni prihodi in odhodi vplivajo le na tisti list. Vsakemu prefiksu pa ustreza neko notranje vozlišče (če sploh obstaja) in prefiksni prihodi in odhodi s tem prefiksom vplivajo le na tisto notranje vozlišče. Imen zato ni treba še posebej urejati po abecedi ali jih pregledovati vnaprej, pač pa lahko sproti pri ne-prefiksnih prihodih dodajamo morebitna manjkajoča vozlišča v drevo (ko prvič naletimo na neko ime). Časovna zahtevnost posameznega dogodka je torej $O(d)$, kjer je d dolžina imena oz. prefiksa pri tem dogodku; časovna zahtevnost celotne rešitve je $O(\sum d)$, kjer $\sum d$ predstavlja skupno dolžino vseh imen in prefiksov v vhodnih podatkih. Boljša od tega (v asimptotičnem

smislu) pa rešitev že ne bi mogla biti, saj potrebujemo toliko časa že tudi samo za branje vhodnih podatkov.

Razmislimo zdaj še o težji različici naloge, ki jo omenja opomba pod črto na str. 27. Za osnovo vzemimo pravkar omenjeno rešitev z drevesom po črkah, čeprav bi šlo tudi s segmentnim drevesom. Spomnimo se, da pri prefiksnem prihodu (oz. odhodu) načeloma želimo odšteti (oz. prišteti) trenutno vrednost s_t k dolgu vseh ljudi, ki takrat pridejo v skupino (oz. jo zapustijo). Ker bi bilo prepočasi, če bi se ukvarjali z vsakim od teh ljudi posebej, smo to reševali tako, da smo s_t prišteli oz. odšteli pri nekem višje ležečem vozlišču v drevesu (ki ustreza prefiksu trenutnega prefiksnega dogodka), pri čemer je veljalo, da te spremembe veljajo tudi za vse njegove potomce v drevesu (kar pomeni za vse ljudi, katerih ime se začne na prefiks trenutnega dogodka). Zdaj, pri težji različici naloge, pa ni več tako; prefiksni prihod velja le za tiste ljudi, ki se jim ime začne na pravi prefiks *in ki jih trenutno še ni v skupini*, analogno pa tudi za prefiksne odhode. Lahko se celo zgodi, da v vhodnih podatkih večkrat nastopi prefiksni prihod z istim prefiksom, pri čemer se vsakič nanaša na neko malo drugačno skupino ljudi (ker so vmes nekateri ljudje s tem prefiksom prihajali v skupino ali odhajali iz nje zaradi drugih dogodkov). Zato nam nič ne pomaga, če v vozlišču, ki ustreza temu prefiksu računamo le vsoto oz. razliko s_t-jev po vseh prefiksnih dogodkih za ta prefiks, saj bodo za različne potomce takega vozlišča na koncu relevantne različne podmnožice teh s_t-jev. Namesto tega bomo v vsakem vozlišču hranili seznam dogodkov (oz. dovolj bo že seznam njihovih časov t), ki se nanašajo nanj; šele po koncu vseh dogodkov pa bomo te podatke prenašali navzdol po drevesu.

Naslednja težava je, da moramo znati po vsakem prihodu ali odhodu hitro izračunati novo število ljudi v skupini, saj ga bomo potrebovali pri plačilih (da izračunamo, za koliko se poveča dolg vsakega trenutnega člana skupine). Pri prvotni različici naloge je bilo to lažje: pri prefiksnem dodajanju pridejo v skupino vsi ljudje, ki se jim ime začne na ta prefiks (in ki smo jih kdaj prej že dodali v skupino s posamičnim dodajanjem — toda ta slednji pogoj je pri uporabi drevesa po črkah trivialen, saj posameznega človeka tako ali tako dodamo v drevo šele takrat, ko prvič pride v skupino); zdaj pa je mogoče, da so nekateri od teh ljudi ob takem dogodku že v skupini in jih ne smemo šteti, ko računamo novo velikost skupine. Podobno je pri prefiksnem brisanju. Koristno bi torej bilo, če bi za vsako vozlišče drevesa vedeli, koliko ljudi iz njegovega poddrevesa je trenutno v skupini; toda tega podatka ne moremo zares vzdrževati za vsa vozlišča, kajti ko nastopi na primer prefiksni dogodek za neko notranje vozlišče, bi bilo treba potem popraviti število ljudi v skupini za vse potomce tega vozlišča, s tem pa bi bilo preveč dela. Pri posameznem dogodku si lahko privoščimo popraviti vse prednike vozlišča, na katero se dogodek nanaša, ne pa tudi vseh potomcev. Zato bomo v vsakem vozlišču hranili le podatek o tem, koliko ljudi iz njegovega poddrevesa bi bilo trenutno v skupini, če odmislimo prefiksne dogodke, ki se nanašajo na prednike tega vozlišča. Te slednje, torej prefiksne dogodke v prednikih, pa bomo primerno upoštevali ob spuščanju po drevesu.

Zdaj lahko opišemo našo rešitev malo natančneje. Za vozlišče u našega drevesa po črkah naj bo $niz(u)$ niz znakov, ki ga tvorijo povezave na poti od korena do u. Če je u list, predstavlja človeka z imenom $niz(u)$, le brez znaka `#` na koncu; zato bomo namesto o listu u včasih govorili o človeku u in podobno. Naj bo $\mathcal{T}(u)$ množica, ki

jo tvorijo u in vsi njegovi potomci — to je u-jevo *poddrevo*; in naj bo $\mathcal{L}(u)$ množica vseh listov iz $\mathcal{T}(u)$, kar bomo krajše imenovali „u-jevi listi“.

Človeku, ki smo ga že kdaj dodali v skupino s posamičnim dogajanjem, bomo rekli, da je *znan*. V drevesu so listi le za dotlej znane ljudi.

Dogodke prvih dveh tipov (prihode in odhode) oštevilčimo po vrsti, kot se pojavljajo v vhodnih podatkih (takšni zaporedni številki bomo včasih rekli tudi *čas* dogodka). Za dogodek t definirajmo $niz(t)$ takole: če je t prefiksni dogodek, naj bo $niz(t)$ njegov prefiks (brez zvezdice na koncu), sicer pa naj bo $niz(t)$ ime iz tega ne-prefiksnega dogodka, ki pa mu na koncu dodamo še znak `#`. Rekli bomo, da se dogodek t *nanaša* na vozlišče u, če je $niz(t) = niz(u)$.

Za vsako vozlišče u našega drevesa po črkah bomo hranili naslednje podatke:

- $E[u]$ naj bo seznam časov tistih dogodkov, ki se nanašajo na to vozlišče;
- $P[u]$ naj bo zadnji od teh dogodkov, ki je prihod, $O[u]$ pa zadnji, ki je odhod (če ni še nobenega prihoda ali odhoda, si mislimo -1);
- $Z[u]$ naj bo število u-jevih listov, torej $|\mathcal{L}(u)|$ — kar so z drugimi besedami doslej znani ljudje (zato Z) iz u-jevega poddrevesa;
- $F[u]$ naj bo zadnji dogodek, ki se nanaša na kakšno vozlišče iz $\mathcal{T}(u)$;
- $S[u]$ naj bo število u-jevih listov, ki so bili v skupini takoj po dogodku $F[u]$;
- *plačila*$[u]$ naj bo vsota vseh plačil, ki jih je doslej opravil človek u (to je smiselno seveda le, če je u list; pri notranjih vozliščih bo to 0).

Iz teh definicij sledi, da je v korenu drevesa vrednost $S[koren]$ ravno trenutno število ljudi v skupini.

(Za vajo razmislimo, kako lahko s temi podatki za poljuben u določimo, koliko u-jevih listov je res v skupini: naj bo $p := \max_v P[v]$, kjer gre v po vseh u-jevih prednikih, in podobno $o := \max_v O[v]$. Potem je število u-jevih listov, ki so trenutno v skupini, odvisno od tega, kateri izmed dogodkov p, o in $F[u]$ je najkasnejši. Če je to dogodek $F[u]$, je v skupini $S[u]$ izmed u-jevih listov; če dogodek p, jih je v skupini $Z[u]$; če dogodek o, pa ni v skupini nobenega u-jevega lista.)

Glavni del našega postopka bo zdaj takšen:

```
t := 0; vsotaDolgov := 0;
za vsak dogodek iz vhodnih podatkov:
  če je trenutni dogodek plačilo:
    u := vozlišče, do katerega pridemo iz korena po črkah imena plačnika;
    plačila[u] += (znesek plačila);
    vsotaDolgov += (znesek plačila)/S[koren];
  sicer:
    t := t + 1; (* t je zdaj čas trenutnega dogodka *)
    prihod[t] := logična vrednost, ki pove, ali je ta dogodek prihod ali odhod;
    s[t] := vsotaDolgov;
    PrihodOdhod(t);
IzpišiRezultate(koren, "", {});
```

Plačila torej obdelujemo sproti, pri čemer delimo znesek plačila s številom ljudi v skupini in računamo kumulativne vsote tako nastajajočih dolgov. Pri ostalih dogodkih pa si v $prihod[t]$ in $s[t]$ zapomnimo, ali so prihodi ali odhodi in kakšna je bila takrat kumulativna vsota dolgov; nato pokličemo podprogram PrihodOdhod, čigar

naloga je primerno popraviti drevo. Ko tako obdelamo vse dogodke, pokličemo podprogram IzpišiRezultate, ki se bo rekurzivno spustil po drevesu (od korena navzdol), računal dolgove in jih izpisoval. Oglejmo si najprej podprogram PrihodOdhod:

podprogram PrihodOdhod(dogodek t):
1 spusti se iz korena drevesa po črkah niza $niz(t)$
in naj bo u vozlišče, ki ga pri tem dosežemo;
če v drevesu še ni ustreznih vozlišč za takšno spuščanje:
if je t prefiksni dogodek ali odhod **then return**;
else manjkajoča vozlišča dodaj v drevo in vsem vozliščem v na poti od korena do u (vključno z njima) povečaj $Z[v]$ za 1;
ob spuščanju izračunaj tudi $p := \max_v P[v]$ in $o := \max_v O[v]$ po vseh u-jevih prednikih v;
2 naj bo s število u-jevih listov, ki so trenutno v skupini — malo prej smo videli, kako ga lahko izračunamo iz p, o, $F[u]$, $Z[u]$ in $S[u]$;
3 $F[u] := t$; dodaj t v $E[u]$;
if je t prihod **then** $S[u] := Z[u]$; $P[u] := t$
else $S[u] := 0$; $O[u] := t$;
4 $p := -1$; $o := -1$; $\triangle_s := S[u] - s$;
za vsakega u-jevega prednika v, od korena navzdol:
$s' :=$ število v-jevih listov, ki so bili pred trenutnim dogodkom v skupini (izračunamo ga iz p, o, $F[v]$, $Z[v]$ in $S[v]$);
$F[v] := t$; $S[v] := s' + \triangle_s$;

V koraku 1 se torej spustimo po črkah niza, ki smo ga dobili pri trenutnem dogodku; če primernih vozlišč ne najdemo, to pomeni, da doslej v skupino še nikoli ni prišel človek s tem imenom oz. z imenom, ki bi se začelo na ta prefiks. V tem primeru naš dogodek nima nobenega učinka, če je prefiksni ali če gre za odhod; če pa gre za neprefiksni prihod, je to zdaj prvi prihod tega človeka v skupino, zato moramo v drevo dodati ustrezna vozlišča in na celotni poti od korena do novega lista povečati števce listov $Z[v]$. Nato v koraku 2 izračunamo, koliko u-jevih listov je trenutno (pred trenutnim dogodkom) v skupini. V koraku 3 dodamo novi dogodek v seznam $E[u]$ in popravimo števila $P[u]$, $O[u]$ in $F[u]$; izračunamo tudi novo število u-jevih listov v skupini — če je trenutni dogodek prihod, bo zdaj v skupini vseh $Z[u]$ listov, sicer pa nobeden. To je nova vrednost $S[u]$; razliko med $S[u]$ in s, torej med novim in starim številom u-jevih listov v skupini, pa imenujmo $\triangle_s$. V koraku 4 gremo še enkrat po u-jevih prednikih v (mednje ne štejemo u-ja samega) in pri vsakem ustrezno popravimo $F[v]$ ter izračunamo novo število v-jevih listov v skupini (dobimo ga tako, da staremu prištejemo $\triangle_s$).

Razmislimo zdaj, kako bomo na koncu s pomočjo podatkov v drevesu za vsakega človeka izračunali njegov dolg. Človeka predstavlja neki list drevesa, recimo u, in nanj vplivajo ne le dogodki iz $E[u]$, pač pa tudi iz vseh $E[v]$ za u-jeve prednike v. Izračunati moramo torej unijo vseh teh množic dogodkov — recimo ji $\mathcal{E}(u)$. To lahko poceni počnemo med rekurzivnim pregledovanjem drevesa: ko smo pri notranjem vozlišču v, dodamo vse elemente $E[v]$ v našo unijo, nato izvedemo rekurzivne klice za vse v-jeve otroke, zatem pa elemente $E[v]$ spet pobrišemo iz unije.

Ko enkrat vemo, kateri dogodki vplivajo na človeka u, je načeloma ideja enaka kot pri prvotni nalogi: če je človek ob času t prišel v skupino, je treba od njegovega

dolga odšteti $s[t]$, če pa je ob času t odšel iz skupine, je treba njegovemu dolgu prišteti $s[t]$. (Spomnimo se, da je $s[t]$ vrednost spremenljivke *vsotaDolgov* v času t-tega prihoda ali odhoda.) Je pa tu treba paziti še na nekaj podrobnosti. (1) Dogodki pred prvim posamičnim (torej ne-prefiksnim) prihodom u-ja v skupino v resnici ne veljajo za u, zato jih moramo ignorirati. Na srečo ni težko ugotoviti, kateri je ta prvi posamični prihod: to je ravno prvi element seznama $E[u]$. (2) Če je po vseh dogodkih človek u še v skupini, moramo k njegovemu dolgu prišteti tudi končno vrednost *vsotaDolgov*. Ta primer prepoznamo po tem, da je v $\mathcal{E}(u)$ zadnji dogodek prihod, ne pa odhod. (3) Pri tej težji različici naloge se lahko pojavi na primer prefiksni prihod, pri čemer je nekaj ljudi s tem prefiksom že v skupini ipd. Če si predstavljamo $\mathcal{E}(u)$ kot urejen seznam (po času), je zato lahko v njem več zaporednih prihodov ali več zaporednih odhodov. Od takih mora vrednost $\pm s[t]$ v izračun u-jevega dolga zato prispevati le prvi dogodek, ostali pa ne.

Zato bo koristno, če si $\mathcal{E}$ predstavljamo kot nekakšen slovar, v katerem so ključi dogodki (oz. njihovi časi), pripadajoča vrednost pri dogodku t v tem slovarju pa je tisto, kar ta dogodek prispeva k dolgu človeka u: to je $s[t]$ za odhod, $-s[t]$ za prihod in 0 za dogodke, ki jih ignoriramo (ker imamo več zaporednih prihodov ali več zaporednih odhodov). Ko v $\mathcal{E}$ dodamo nov dogodek, moramo pogledati, kakšnega tipa sta prejšnji in naslednji dogodek iz $\mathcal{E}$, in po potrebi ustrezno popraviti pripadajoče vrednosti. Recimo, da dodajamo v $\mathcal{E}$ nov dogodek t; prejšnjega označimo s $p = \max\{t' \in \mathcal{E} : t' < t\}$, naslednjega pa s $q = \min\{t' \in \mathcal{E} : t' > t\}$. Recimo še, da je t prihod. Potem moramo razmišljati takole: če je tudi p prihod, moramo dodati t s pripadajočo vrednostjo 0 namesto $-s[t]$; če pa je p odhod ali pa sploh ne obstaja, moramo pogledati q; če je ta prihod, mu moramo popraviti pripadajočo vrednost z $-s[q]$ na 0, sicer pa z 0 na $s[q]$ — v vsakem primeru se mu torej pripadajoča vrednost poveča za $s[q]$. Doslej smo razmišljali o primeru, ko je t prihod; če je v resnici odhod, je razmislek analogen.

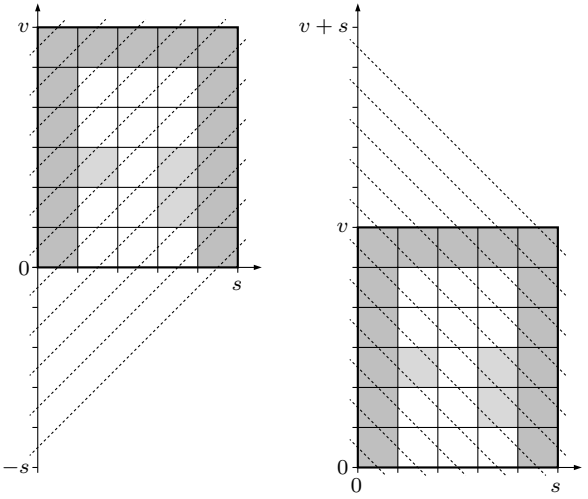
Zdaj torej vidimo, kakšne operacije pričakujemo od našega slovarja $\mathcal{E}$: dodajati in brisati elemente; poiskati največji element; za dani t poiskati prejšnji in naslednji element; obstoječemu elementu spremeniti pripadajočo vrednost; in izračunati vsoto pripadajočih vrednosti po vseh ključih, ki so večji ali enaki $\min E[u]$ — spomnimo se namreč, da je $\min E[u]$ čas prvega posamičnega prihoda osebe u in da dogodki pred tem časom na u v resnici ne vplivajo. Primerna podatkovna struktura za ta nabor operacij je na primer segmentno drevo, šlo pa bi seveda tudi z rdeče-črnim ali kakšnim drugim; tako lahko vsako od navedenih operacij izvedemo v $O(\log q_{12})$ časa, kjer je $q_{12} := q - q_3$ število vseh prihodov in odhodov (spomnimo se, da imamo v vhodnih podatkih q dogodkov, od tega q_3 plačil). Zdaj vemo dovolj, da lahko opišemo še podprogram IzpišiRezultate:

podprogram IzpišiRezultate(vozlišče u, niz *ime*, slovar $\mathcal{E}$):
1 $c :=$ znak na povezavi, ki kaže v u (če je u koren, naj bo c presledek);
  če u ni koren, dodaj c na konec niza *ime*;
2 za vsak dogodek t s seznama $E[u]$:
    $p := \max\{t' \in \mathcal{E} : t' < t\}$; $q := \min\{t' \in \mathcal{E} : t' > t\}$;
    če je t prihod:
      če p obstaja in je prihod: dodaj t v $\mathcal{E}$ z vrednostjo 0;
      sicer:

```
        dodaj t v E z vrednostjo −s[t];
        če q obstaja, povečaj njegovo vrednost v E za s[q];
    sicer (torej če je t odhod):
      če p ne obstaja ali je odhod: dodaj t v E z vrednostjo 0;
      sicer:
        dodaj t v E z vrednostjo s[t];
        če q obstaja, zmanjšaj njegovo vrednost v E za s[q];
3   če je u list (torej če je c = #):
    t := min E[u]; p := max{t' ∈ E : t' < t}; q := max E;
    če p obstaja in je prihod: zmanjšaj v E vrednost t-ja z 0 na −s[t];
    dolg := vsota vrednosti v E po vseh tistih ključih, ki so ≥ t;
    če je q odhod, povečaj dolg za vsotaDolgov;
    izpiši, da je oseba ime dolžna dolg − plačila[u] denarja;
4   za vsakega u-jevega otroka v: IzpišiRezultate(v, ime, E);
5   razveljavi v E vse spremembe, ki si jih izvedel v korakih 2 in 3;
    če u ni koren, pobriši znak c s konca niza ime;
```

V koraku 2 torej dodamo v $\mathcal{E}$ dogodke t, ki se nanašajo na trenutno vozlišče u, torej iz seznama $E[u]$. Pri vsakem t pogledamo prejšnji in naslednji dogodek v $\mathcal{E}$ (p in q), da lahko ustrezno določimo vrednost t-ja in po potrebi popravimo vrednost q-ja (glede na to, ali imamo dva zaporedna dogodka enakega tipa — dva prihoda ali dva odhoda — ali različnih tipov). V koraku 3 izračunamo vsoto vrednosti, ki jih prispevajo vsi dogodki v $\mathcal{E}$, da dobimo skupni dolg človeka u. Pri tem pa pazimo, da moramo ignorirati dogodke pred prvim posamičnim prihodom u-ja v skupino — to je prvi dogodek iz $E[u]$ in si ga začasno zapomnimo v t. Če je prejšnji dogodek (pred t) v $\mathcal{E}$ tudi prihod, to pomeni, da ima t zdaj v $\mathcal{E}$ vrednost 0; ker pa moramo dogodke pred t zdaj ignorirati, moramo t-ju v $\mathcal{E}$ postaviti vrednost na $-s[t]$, kot je običajno za prihode. Nato pogledamo še, če je zadnji dogodek v $\mathcal{E}$ prihod; v tem primeru je u na koncu vseh dogodkov še vedno v skupini in moramo dolgu prišteti še trenutno (torej končno) vrednost *vsotaDolgov* (iz glavnega dela postopka). Nazadnje zmanjšamo u-jev dolg še za zneske, ki jih je doslej vplačal in katerih vsota je shranjena v *plačila*$[u]$. V koraku 4 nadaljujemo rekurzivno po vseh u-jevih otrocih (če jih ima). V koraku 5 razveljavimo vse spremembe, ki smo jih v korakih 2 in 3 naredili v strukturi $\mathcal{E}$ in v nizu *ime*; tako bosta ob vrnitvi iz našega rekurzivnega klica oba v enakem stanju kot prej in bosta torej pripravljena na naslednji rekurzivni klic. To pomeni, da si moramo v korakih 2 in 3 zapisovati (v neki seznam), katere ključe dodajamo v $\mathcal{E}$ in s kakšno vrednostjo ter katerim ključem spreminjamo vrednost in za koliko, nato pa lahko v koraku 5 izvedemo ravno nasprotne spremembe v nasprotnem vrstnem redu — kjer smo prej prištevali, zdaj odštevamo, kjer smo prej dodajali, zdaj brišemo in tako naprej.

Kakšna je časovna zahtevnost te rešitve? Podprogram PrihodOdhod porabi $O(d)$ časa, kjer je d dolžina niza z imenom ali prefiksom pri tem dogodku; toliko ga porabi tudi glavni del postopka pri vsakem plačilu; za vse dogodke skupaj torej $O(\sum d)$ časa. Nato pride še IzpišiRezultate, kjer imamo za vsako od $O(\sum d)$ vozlišč po en klic; za vsakega od $O(n)$ listov imamo nekaj (konstantno mnogo) operacij na $\mathcal{E}$; ravno tako pa tudi za vsak element vsakega od seznamov $E[u]$. Ti seznami so vsega skupaj dolgi $O(q_{12})$, saj pri vsakem prihodu ali odhodu dodamo



Slika 1. Primer premic, ki nas zanimajo pri mreži širine $s = 5$ in višine $v = 6$. Leva slika kaže rastoče diagonale $y - x = C + \frac{1}{2}$ za $-s \le C < v$, desna slika pa padajoče diagonale $y + x = C + \frac{1}{2}$ za $0 \le C < v + s$.

po en element v en tak seznam.[5] Zato je tudi v $\mathcal{E}$ največ $O(q_{12})$ elementov, kar pomeni, da traja vsaka operacija na tej strukturi po $O(\log q_{12})$ časa. Vsega skupaj je torej časovna zahtevnost naše rešitve $O((\sum d) + q_{12} \log q_{12})$.

3. Breakout

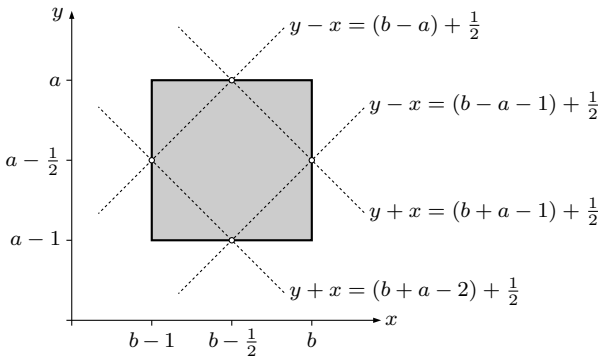
Žogica začne svojo pot na koordinatah $(a - \frac{1}{2}, 0)$, vedno se premika diagonalno in ko se odbije od stranice kakšne opeke, se to vedno zgodi na razpolovišču stranice. Zato za koordinati žogice tudi pri vsakem odboju velja, da je ena od njiju celo število, druga pa je na pol poti med dvema sosednjima celima številoma: $(x, y + \frac{1}{2})$ ali $(x + \frac{1}{2}, y)$.

Smer gibanja žogice lahko opišemo s parom števil $(\triangle_x, \triangle_y)$, pri čemer je vsako od njiju enako bodisi 1 bodisi -1. Če je $\triangle_x = \triangle_y$, se žogica premika po *rastoči* diagonali, torej taki premici, na kateri imajo vse točke enako razliko $y - x$; lahko jo opišemo z enačbo oblike $y - x = C + \frac{1}{2}$, pri čemer je C celo število. V našem primeru, ker nas zanimajo le koordinate z območja $0 \le x \le s$ in $0 \le y \le v$, gre lahko C pri takih premicah od $-s$ do $v - 1$ (glej sliko 1).

Če pa je $\triangle_x = -\triangle_y$, se žogica premika po *padajoči* diagonali, torej premici, na kateri imajo vse točke enako vsoto $y + x$; lahko jo opišemo z enačbo oblike $y + x = C + \frac{1}{2}$ za neko celo število C, ki gre zdaj lahko od 0 do $s + v - 1$.

V poštev pride torej $s + v$ rastočih in prav toliko padajočih premic. Za vsako od njih bi bilo dobro imeti urejen seznam opek, skozi katere gre ta premica; urejene naj bodo v takem vrstnem redu, v kakršnem jih premica doseže od leve proti desni. To pomeni, da jih moramo urediti naraščajoče po x-koordinati; paziti pa moramo na možnost, da ista premica pri istem x zadene dve opeki, ki sta ravno ena nad

[5] Tu tudi vidimo, zakaj je dobro, da smo uporabili drevo po črkah (*trie*) in ne segmentnega drevesa. Pri slednjem lahko vsak prefiksni dogodek vpliva na po največ dve vozlišči na vsakem nivoju drevesa, teh pa je $O(\log n)$; zato bi bila skupna dolžina vseh seznamov $E[u]$ zdaj $O(q_{12} \log n)$ in časovna zahtevnost celotne rešitve bi bila za faktor $O(\log n)$ počasnejša kot pri drevesu po črkah.



Slika 2. Primer opeke z zgornjim desnim kotom (a, b) in štirih diagonalnih premic, ki jo sekajo. Krožci kažejo razpolovišča stranic, kjer se lahko žogica odbije od te opeke.

drugo, torej (x, y) in $(x, y+1)$. V takem primeru mora priti najprej na vrsto opeka (x, y), če je premica rastoča, oz. najprej opeka $(x, y+1)$, če je premica padajoča. (Za urejanje opek po x-koordinati lahko uporabimo neke vrste urejanje s štetjem; ustvarimo s seznamov in nato vsako opeko (x, y) dodamo na x-ti seznam. Tako za urejanje porabimo le $O(N)$ časa, kjer je $N = 2(v-1)+s+k$ število vseh opek, tudi tistih v stenah.)

Takšne sezname lahko uporabimo za iskanje naslednje opeke, od katere se bo žogica odbila. Če poznamo trenutni položaj in smer gibanja žogice, vemo tudi, po kateri premici se bo gibala. Na njej bi lahko z bisekcijo iskali x-koordinato žogice in videli, med katerima dvema opekama leži; odbila se bo torej od leve oz. desne od teh dveh opek, odvisno od tega, ali je $\triangle_x$ enak -1 ali $+1$. Ta rešitev ima dve slabosti: ena slabost je, da bo občasno treba pobrisati kakšno opeko (ker opeka po določenem številu odbojev razpade) in ta operacija bo draga, ker bomo morali imeti sezname predstavljene s tabelami ali vektorji (da bomo imeli do opek naključen dostop, ki ga potrebujemo za bisekcijo). Druga slabost je, da bomo imeli pri vsakem odboju po $O(\log n)$ dela zaradi bisekcije, odbojev pa je lahko pri največjih testnih primerih toliko, da bo to že prepočasi.

Namesto tega naj bo vsak seznam raje veriga členov, povezanih s kazalci (*linked list*); brisanje iz takega seznama vzame le $O(1)$ časa. Vsako opeko sekajo štiri premice (glej sliko 2), torej pripada štirim takim seznamom in v neki tabeli si bomo zapomnili člene, ki predstavljajo to opeko v tistih štirih seznamih. Ko žogica zadene opeko, to pomeni, da smo v seznamu, ki predstavlja premico, po kateri se je žogica zdaj premikala, prišli do člena, ki predstavlja tisto opeko. Iz prejšnjega položaja žogice, položaja opeke ter smeri gibanja žogice lahko tudi določimo stranico, od katere se je žogica odbila; potem tudi poznamo koordinate točke odboja (to je ravno razpolovišče tiste stranice) in novo smer gibanja (pri odboju od vodoravne stranice se $\triangle_x$ pomnoži z -1, vrednost $\triangle_y$ pa ostane nespremenjena; pri odboju od navpične stranice je ravno obratno). Zdaj poznamo novi položaj in smer gibanja žogice, torej lahko tudi določimo, po kateri premici se bo premikala po odboju. Spomnimo se, da imamo za vsako opeko shranjene člene v seznamih, ki jim pripada; za trenutno opeko torej poznamo njen člen na premici, po kateri se bo gibala žogica po odboju; naslednja opeka, od katere se bo odbila, je torej bodisi prejšnji bodisi naslednji člen tega seznama (odvisno od tega, ali je novi $\triangle_x$ negativen ali pozitiven, saj vemo,

da so seznami urejeni po x). Tako se torej lahko v $O(1)$ časa premaknemo na tisti naslednji ali prejšnji člen in s tem izvemo, od katere opeke se bo žogica odbila v naslednjem odboju. Vsak odboj nam torej vzame le $O(1)$ časa, ne pa $O(\log n)$ kot pri bisekciji.

Če pri iskanju opeke za naslednji odboj vidimo, da naslednjega oz. prejšnjega člena v seznamu (ki predstavlja premico, po kateri se bo žogica gibala) sploh ni, to pomeni, da bo žogica odletela iz mreže in lahko simulacijo njenega gibanja končamo. Da izračunamo, v kateri točki bo ušla iz mreže, moramo pogledati, katero stranico mreže bo prej zadela, navpično (levo, če je $\triangle_x < 0$, oz. desno, če je $\triangle_x > 0$) ali vodoravno (spodnjo, če je $\triangle_y < 0$, oz. zgornjo, če je $\triangle_y > 0$).

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

struct Opeka;
typedef list<Opeka *> Premica;

struct Opeka
{
  int x, y, k;
  int naslNaX; // naslednja opeka s to x-koordinato
  Premica::iterator it[2][2]; // prvi indeks: 0 = padajoča, 1 = rastoča;
                              // drugi indeks: 0 = spodnja, 1 = zgornja

  void Dodaj(Premica &premica, int rastoca, int zgornja) {
    // Pazimo na to, kako morajo biti na tej premici urejene opeke z istim x.
    // Morda moramo trenutno opeko vriniti na predzadnje mesto, ne dodati na konec.
    auto kam = premica.end();
    if (! premica.empty()) { auto b = premica.back();
      if ((x == b->x) && ((rastoca == 1) == (y < b->y))) --kam; }
    it[rastoca][zgornja] = premica.insert(kam, this); }

  // Vrne parameter C ene od premic skozi to opeko.
  int C(int rastoca, int zgornja) const {
    return y + (rastoca ? -x : x - 1) - (zgornja ? 0 : 1); }
};

int main()
{
  int v, s, n, a; scanf("%d %d %d %d", &s, &v, &n, &a);

  // Preberimo opeke.
  int N = n + 2 * (v - 1) + s;
  vector<Opeka> opeke; opeke.reserve(N); opeke.resize(n);
  for (Opeka &O : opeke) scanf("%d %d %d", &O.x, &O.y, &O.k);

  // Dodajmo opeke, ki tvorijo stene.
  enum { K = 100 };
  for (int y = 1; y < v; ++y) { opeke.push_back({1, y, K}); opeke.push_back({s, y, K}); }
  for (int x = 1; x <= s; ++x) opeke.push_back({x, v, K});

  // Uredimo opeke po x-koordinati.
  vector<int> prvaNaX(s + 1, -1);
  for (int i = 0; i < N; ++i) { auto &O = opeke[i];
    O.naslNaX = prvaNaX[O.x]; prvaNaX[O.x] = i; }

  // rastoce[C] predstavlja premico y - x = C + 1/2; padajoce[C] pa y + x = C + 1/2.
```

```
vector<Premica> rastoce_(v + s), padajoce_(v + s);
auto rastoce = rastoce_.begin() + s, padajoce = padajoce_.begin();

// Dodajmo jih na premice; na vsaki premici bodo opeke urejene naraščajoče po x.
// Če imata dve opeki na isti premici isti x, premica zadene prej tisto z nižjim y,
// če je rastoča, oz. tisto z višjim y, če je padajoča. Na to pazimo pri dodajanju,
// ker imamo opeke urejene le po x, ne pa tudi po y.
for (int x = 0; x <= s; ++x) for (int i = prvaNaX[x]; i >= 0; ) { auto &O = opeke[i];
  for (int r = 0; r < 2; ++r) for (int z = 0; z < 2; ++z)
    O.Dodaj((r ? rastoce : padajoce)[O.C(r, z)], r, z);
  i = O.naslNaX; }

// Začnimo s simulacijo.
int x2 = 2 * a - 1, y2 = 0, dx = 1, dy = 1;

// Žogica je na (x2/2, y2/2). To pomeni, da leži na naraščajoči premici
// y - x = (y2 - x2) / 2 = (y2 - x2 - 1) / 2 + 1/2. Poiščimo prvo opeko, ki jo zadene.
int C = (y2 - x2 - 1) / 2;
Premica *premica = &rastoce[C];
auto opeka = premica->begin();
while (opeka != premica->end() && (*opeka)->x <= (x2 + 1) / 2) ++opeka;

int stRazbitih = 0;
while (true)
{
  Opeka &O = **opeka;

  // Od katere stranice te opeke se žogica odbije?
  enum { S, J, V, Z } str =
    (dx == 1) ? (dy == 1 ? (O.C(1, 1) == C ? Z : J) : (O.C(0, 1) == C ? S : Z)) :
                (dy == 1 ? (O.C(0, 1) == C ? V : J) : (O.C(1, 1) == C ? S : V));

  // Izračunajmo smer gibanja po odboju.
  if (str == Z || str == V) dx = -dx; else dy = -dy;

  // Izračunajmo koordinate točke, kjer se žogica odbije od opeke.
  x2 = 2 * O.x - (str == Z ? 2 : str == V ? 0 : 1);
  y2 = 2 * O.y - (str == S ? 0 : str == J ? 2 : 1);

  // Določimo premico, po kateri se bo gibala po odboju.
  int rastoca = (dx == dy) ? 1 : 0;
  C = (y2 + (rastoca ? -x2 : x2) - 1) / 2;
  premica = &(rastoca ? rastoce : padajoce)[C];

  // Določimo naslednjo opeko na njeni poti.
  opeka = O.it[rastoca][(O.C(rastoca, 1) == C) ? 1 : 0];
  bool konec = false; if (dx > 0) konec = (++opeka == premica->end());
  else { if (opeka == premica->begin()) konec = true; else --opeka; }

  // Pobrišimo trenutno opeko, če pri tem odboju razpade.
  if (--O.k == 0) { ++stRazbitih;
    for (int r = 0; r < 2; ++r) for (int z = 0; z < 2; ++z)
      (r ? rastoce : padajoce)[O.C(r, z)].erase(O.it[r][z]); }

  if (konec) break;
}
// Od trenutnega položaja naprej se žogica ne odbija več;
// izračunajmo, pri kateri koordinati zapusti igralno polje.
int premik = min((dx == 1 ? 2 * s - x2 : x2), (dy == 1 ? 2 * v - y2 : y2));
x2 += dx * premik; y2 += dy * premik;

printf("%d\n%d%s %d%s\n", stRazbitih, x2 / 2, (x2 % 2 ? ".5" : ""),
                                      y2 / 2, (y2 % 2 ? ".5" : ""));
```

```
  return 0;
}
```

Še nekaj podrobnosti o gornji implementaciji: za predstavitev premic smo uporabili razred list iz standardne knjižnice; na člene teh seznamov kažejo iteratorji. Za predstavitev trenutnega položaja žogice namesto njenih koordinat hranimo dvakratnika teh koordinat (spremenljivki x2 in y2), da se izognemo delu z ne-celimi števili (razen čisto na koncu, pri izpisu). Pri dodajanju opek v sezname pazimo na to, da sicer gledamo opeke po naraščajočih x, niso pa nujno pravilno urejene po y, zato moramo včasih vriniti opeko na predzadnje mesto seznama, včasih pa jo dodati prav na konec seznama (metoda Opeka::Dodaj).

4. Pijansko urejanje

Poskusimo za začetek ugotoviti, na katerem podstavku stoji kateri zaboj. Po prvi zamenjavi (ne glede na to, s katerim podstavkom) izvemo, kateri zaboj zdaj stoji na podstavku 0. Od tu naprej lahko nadaljujemo v zanki; recimo, da za podstavke od 0 do $k-1$ že vemo, kateri zaboji so na njih, in da bi radi to izvedeli še za podstavek k. Če naročimo zamenjavo s podstavkom $k-1$, bomo v resnici dobili na podstavek 0 enega od zabojev s podstavkov $k-2$, $k-1$ ali k (pri $k=1$ seveda možnost $k-2$ odpade). (1) Če je bil to eden od zabojev s podstavkov $k-2$ ali $k-1$, ga bomo zlahka prepoznali, saj smo pred to zamenjavo za podstavke do vključno $k-1$ že vedeli, kateri zaboji so na njih. V tem primeru nismo z zamenjavo izvedeli ničesar novega in moramo poskusiti še enkrat. (2) Če pa nam zamenjava pripelje na podstavek 0 neki zaboj, ki ga še ne poznamo, lahko zaključimo, da je ta zaboj moral priti s podstavka k in da je torej zdaj na podstavek k prišel tisti zaboj, ki je bil prej na podstavku 0. V tem primeru zdaj poznamo vsebino prvih $k+1$ podstavkov (in ne več le prvih k) in lahko v nadaljevanju povečamo k za 1.

Sčasoma pridemo s tem postopkom do konca zaporedja in potem za vsak podstavek vemo, kateri zaboj je na njem — ali obratno: za vsak zaboj vemo, na katerem podstavku se nahaja; pravzaprav je koristno imeti obe vrsti podatkov (spodnji program ima v ta namen globalni spremenljivki v in kje). Teh podatkov tudi ni težko vzdrževati pri vsaki izvedeni zamenjavi.

Za urejanje tabele lahko uporabimo znani postopek urejanja z izbiranjem (*selection sort*): najprej premaknemo zaboj $n-1$ na podstavek $n-1$, nato zaboj $n-2$ na podstavek $n-2$ in tako navzdol proti vse manjšim številkam zabojev in podstavkov. Recimo, da smo zaboje od $k+1$ do $n-1$ že postavili na pravo mesto; zdaj poskusimo to narediti z zabojem k. Tudi zanj, tako kot za vse zaboje, točno vemo, kje je; recimo, da je na podstavku p. Če naročimo zamenjavo s podstavkom $p-1$, bo na podstavek 0 prišel eden od zabojev s podstavkov $p-2$, $p-1$ in p, torej obstaja možnost, da bo to ravno zaboj k (če ni, moramo pač z enako zamenjavo poskusiti še večkrat). Ko imamo zaboj k na podstavku 0, pa naročimo zamenjavo s podstavkom $k-1$; tako pride zaboj k na enega od podstavkov $k-2$, $k-1$ in k. Če ni prišel ravno na podstavek k, ga moramo spraviti nazaj na podstavek 0 in nato poskusiti znova; prej ali slej bo prišel na podstavek k.

```
#include <vector>
#include <algorithm>
```

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

vector<int> v, kje;  // v[i] = številka zaboja na podstavku i;
                     // kje[i] = številka podstavka, na katerem je zaboj i.

void Zamenjaj(int k)
{
  printf("%d\n", k); fflush(stdout);
  int stari0 = v[0], novi0; scanf("%d", &novi0);

  // Če smo poznali stari položaj zaboja, ki je zdaj prišel na podstavek 0,
  // potem je to tudi novi položaj zaboja, ki je bil prej na podstavku 0.
  if (kje[novi0] >= 0) v[kje[novi0]] = stari0;
  if (stari0 >= 0) kje[stari0] = kje[novi0];

  // V vsakem primeru pa zdaj vemo, kje je zaboj, ki je prišel na podstavek 0.
  v[0] = novi0; kje[novi0] = 0;
}

int main()
{
  int n; scanf("%d", &n);
  v.resize(n, -1); kje.resize(n, -1);

  // Ugotovimo, kateri zaboj je na katerem podstavku.
  for (int k = 1; k < n; )
  {
    int x = v[0]; Zamenjaj(k - 1);                                    // (1)

    // Zaboj x je bil pred to zamenjavo na podstavku 0, zato je zdaj na enem od
    // podstavkov k - 2, k - 1 ali k. Toda za podstavka k - 2 in k - 1 smo vedeli, kaj je
    // bilo tam pred zamenjavo; če bi x prišel tja, bi zato tudi zanj zdaj vedeli, da je tam.
    // Če torej zdaj za x ne vemo, kje je, to pomeni, da je moral priti na podstavek k.
    if (x >= 0 && kje[x] < 0) { kje[x] = k; v[k++] = x; }
  }

  // Uredimo zaboje z izbiranjem.
  for (int k = n - 1; k >= 1; --k) while (v[k] != k) {

    // Spravimo zaboj k na podstavek 0. Pri tem pazimo, da ne bomo
    // spremenili zaboja desno od njega, kajti ta je morda že na pravem mestu.
    while (v[0] != k) Zamenjaj(max(0, kje[k] - 1));                   // (2)

    // Poskusimo spraviti zaboj k na podstavek k.
    Zamenjaj(k - 1); }                                                // (3)

  printf("-1\n"); return 0;
}
```

Razmislimo o tem, koliko zamenjav izvede ta rešitev. V vrstici (1) čakamo pri vsakem k na to, da pride na podstavek 0 zaboj s podstavka k in ne s podstavka $k-1$ ali $k-2$. To se torej v vsakem poskusu zgodi z verjetnostjo $\frac{1}{3}$, zato potrebujemo v povprečju 3 poskuse. V vrstici (3), ko je zaboj k enkrat na podstavku 0, imamo $\frac{1}{3}$ možnosti, da bo v naslednji zamenjavi prišel na podstavek k (in ne na $k-1$ ali $k-2$), torej potrebujemo v povprečju tri take poskuse; toda pred vsakim od njih moramo zaboj k najprej spraviti na podstavek 0, za kar potrebuje vrstica (2) povprečno tri zamenjave, da res pobere zaboj s podstavka $p = \texttt{kje[k]}$ in ne $p-1$ ali $p-2$. Skupaj imamo torej v vrsticah (2) in (3) povprečno po $3 \cdot (3+1) = 12$ zamenjav pri vsakem zaboju k, da ga končno spravimo na podstavek k. Ker mora tako pri (1) kot pri (2)

in (3) iti k od 1 do $n-1$, imamo vsega skupaj približno $15(n-1)$ zamenjav. Ker gre pri tej nalogi n do 100, nam torej ni treba skrbeti, da bi prekoračili mejo 10 000, kolikor znaša največje dovoljeno število zamenjav.

Natančnejša ocena števila zamenjav. Naš razmislek v prejšnjem odstavku je bil na več mestih nekoliko pesimističen; poskusimo še enkrat, vendar pazljiveje. Glede vrstice (1) lahko opazimo, da pri $k=1$ odpade možnost, da bi na podstavek 0 prišel zaboj s podstavka $k-2$, zato je takrat verjetnost, da na podstavek 0 pride zaboj s podstavka k (in ne $k-1$), kar $\frac{1}{2}$ (in ne samo $\frac{1}{3}$ kot pri $k>1$); zato potrebujemo povprečno le dva poskusa, ne treh. Toda spomnimo se, da pred prvo zamenjavo pri $k=1$ ne poznamo še čisto nobenega zaboja, niti tistega na podstavku 0; zato po prvi zamenjavi prav gotovo še ne bomo vedeli, kateri zaboj je zdaj na podstavku k, pač pa le to, kateri je zdaj na podstavku 0. Šele od takrat naprej potem velja, da potrebujemo še povprečno dva poskusa, preden bomo izvedeli, kateri zaboj je na podstavku 1. Skupaj torej tudi pri $k=1$ potrebujemo povprečno tri zamenjave, enako kot pri $k>1$ — v prejšnjem odstavku smo razmišljali površno, vendar smo za vrstico (1) po srečnem naključju vendarle dobili pravi rezultat.

V vrstici (2) skuša naš program spraviti zaboj k na podstavek 0; označimo z a_p povprečno število zamenjav, ki jih za to potrebujemo, če je p številka podstavka, na katerem se je pred začetkom izvajanja vrstice (2) nahajal zaboj k. Pri $p=0$ je ustavitveni pogoj takoj izpolnjen in ne izvede se nobena zamenjava, torej je $a_0=0$. Pri $p=1$ imamo pri vsaki zamenjavi $\frac{1}{2}$ možnosti, da poberemo zaboj s podstavka p in ne tistega s podstavka $p-1$, torej potrebujemo povprečno 2 zamenjavi in je $a_1=2$. Pri $p>1$ pa imamo pri vsaki zamenjavi $\frac{1}{3}$ možnosti, da poberemo zaboj s podstavka p in ne s podstavkov $p-1$ ali $p-2$, torej potrebujemo povprečno 3 zamenjave in je $a_p=3$.

Recimo zdaj, da se zaboj k nahaja na podstavku 0 in ga hočemo spraviti na podstavek k; za to potrebno število zamenjav označimo z b_k. Pri $k=0$ je zaboj že na pravem mestu in je $b_0=0$. — Pri $k=1$ izvedemo najprej eno zamenjavo; z verjetnostjo $\frac{1}{2}$ pride zaboj k na podstavek 1 in smo končali, z verjetnostjo $\frac{1}{2}$ pa je še vedno na podstavku 0 in smo na istem kot prej (in bomo potrebovali še b_1 zamenjav); velja torej $b_1=1+\frac{1}{2}b_1$, torej $b_1=2$. — Pri $k\geq 2$ izvedemo najprej eno zamenjavo; z verjetnostjo $\frac{1}{3}$ pride zaboj k na podstavek k in smo končali; z verjetnostjo $\frac{1}{3}$ pride na podstavek $k-1$ in potrebujemo a_{k-1} dodatnih zamenjav, da ga spravimo spet na podstavek 0 in lahko potem poskusimo znova (za kar porabimo še b_k zamenjav); z verjetnostjo $\frac{1}{3}$ pa se zgodi enako s podstavkom $k-2$. Velja torej $b_k=1+\frac{1}{3}(a_{k-1}+b_k)+\frac{1}{3}(a_{k-2}+b_k)$, torej $b_k=3+a_{k-1}+a_{k-2}$. Če upoštevamo še ugotovitve o $a_\bullet$ iz prejšnjega odstavka, lahko zaključimo, da je $b_2=5$, $b_3=8$, za $k\geq 4$ pa je $b_k=9$.

Recimo zdaj, da se zaboj k nahaja na podstavku p in ga hočemo spraviti na podstavek k; za to potrebno število zamenjav označimo s c_{pk}. Pri $p=k$ seveda ni treba nobene zamenjave in $c_{kk}=0$; drugače pa moramo zaboj k najprej spraviti na podstavek 0 in nato od tam na podstavek k, tako da je $c_{pk}=a_p+b_k$.

Če nas zanima obnašanje našega algoritma v povprečju po vseh možnih začetnih razporedih, pri čemer tudi štejemo vse te razporede za enako verjetne, lahko predpostavimo, da ko moramo spraviti zaboj k s podstavka p na podstavek k, so vse možne vrednosti p-ja (od 0 do k) enako verjetne. Zanima nas torej povprečje

$\bar{c}_k := C_k/(k+1)$ za $C_k := \sum_{p=0}^{k} c_{pk}$. Upoštevajmo, da je $c_{kk} = 0$ in (za $p < k$) $c_{pk} = a_p + b_k$, pa dobimo $C_k = \sum_{p=0}^{k-1}(a_p + b_k)$; in ker je $a_0 = 0$, $a_1 = 1$ in $a_p = 3$ za $p > 1$, dobimo naprej $C_k = k \cdot b_k + 1 + 3(k-2)$. Upoštevajmo, kar že vemo o b_k, pa dobimo: $C_2 = 12$, $C_3 = 29$ in $C_k = 12k - 4$ za $k \geq 4$. Posebej pa moramo obravnavati primer $k = 1$, kjer vsota ne le nima nobenega člena s $p > 1$, ampak nima niti člena s $p = 1$; takrat je $C_1 = c_{01} + c_{11} = (a_0 + b_1) + 0 = 2$.

Iz vsot C_k, ki smo jih zdaj izračunali, lahko potem dobimo povprečja $\bar{c}_k = C_k/(k+1)$: $\bar{c}_0 = 0$, $\bar{c}_1 = 1$, $\bar{c}_2 = 4$, $\bar{c}_3 = 29/4$ in $\bar{c}_k = 12 - \frac{16}{k+1}$ za $k \geq 4$. Da dobimo skupno število zamenjav v vseh izvajanjih vrstic (2) in (3), moramo to sešteti po vseh k od 1 do $n-1$; recimo tej vsoti $d_n := \sum_{k=1}^{n-1} \bar{c}_k$. Ker nas zanima obnašanje pri večjih n, lahko odmislimo robne primere, ko je $n < 4$; za $n \geq 4$ pa dobimo $d_n = \bar{c}_1 + \bar{c}_2 + \bar{c}_3 + \sum_{k=4}^{n-1}(12 - \frac{16}{k+1}) = 12n - \frac{143}{4} - 16 \cdot (H_n - H_4)$, pri čemer so $H_t = \sum_{u=1}^{t} \frac{1}{u}$ harmonična števila. Če upoštevamo še, da je $H_4 = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} = \frac{25}{12}$, dobimo $d_n = 12n - \frac{29}{12} - 16H_n$.

Da pa dobimo res skupno povprečno število *vseh* zamenjav, moramo prišteti še tistih $3(n-1)$ iz vrstice (1); skupaj imamo torej $T_n := d_n + 3(n-1) = 15n - \frac{65}{12} - 16H_n$ zamenjav. Znano je, da je $H_n = \ln n + \gamma + O(1/n)$, pri čemer je konstanta γ enaka približno 0,577. Če upoštevamo to v naši formuli, vidimo, da je $T_n \approx 15n - 16 \ln n - 14{,}66$. Pri $n = 100$, kar je največji možni testni primer pri naši nalogi, to pomeni (v povprečju) približno 1411,6 zamenjav.[6]

5. L-sistem

Recimo, da je niz s dolg n črk: $s = c_1 c_2 \ldots c_n$. Potem je niz $f^k(s)$, ki nas pri tej nalogi zanima, sestavljen iz n kosov oblike $f^k(c_i)$, torej $f^k(s) = f^k(c_1) f^k(c_2) \ldots f^k(c_n)$. Vsaka pojavitev p-ja (ali kakšnega njegovega anagrama — tega v nadaljevanju ne bomo kar naprej ponavljali) v $f^k(s)$ potem bodisi v celoti leži znotraj enega od kosov $f^k(c_i)$ bodisi prečka vsaj eno mejo med dvema zaporednima kosoma, na primer $f^k(c_i)$ in $f^k(c_{i+1})$.

Tiste, ki v celoti ležijo znotraj enega kosa, lahko torej preštejemo tako, da za vsako črko c naše abecede preštejemo pojavitve p-ja v $f^k(c)$, nato pa to seštejemo po vseh črkah niza s. Pojavitve p-ja v $f^k(c)$ pa lahko preštejemo tako, da upoštevamo, da je $f^k(c) = f^{k-1}(f(c))$; rešiti moramo torej enak problem kot pri $f^k(s)$, le s $k-1$ namesto k in z nizom $f(c)$ namesto s. Tako se nakazuje rešitev z rekurzijo (ali, še bolje, z dinamičnim programiranjem).

Ostane še vprašanje, kako prešteti tiste pojavitve p-ja v $f^k(s)$, ki ne ležijo v celoti znotraj enega kosa $f^k(c_i)$. Če na primer pojavitev prečka mejo med $f^k(c_i)$ in $f^k(c_{i+1})$, to pomeni, da mora prvih nekaj znakov niza p ležati na koncu kosa $f^k(c_i)$, preostanek niza p pa na začetku kosa $f^k(c_{i+1})$. Na posamezni strani meje med $f^k(c_i)$ in $f^k(c_{i+1})$ je lahko največ $|p| - 1$ znakov p-ja, kajti vsaj en znak mora še ostati na drugi strani meje (sicer bo pojavitev ležala v celoti znotraj enega kosa). Za štetje takšnih pojavitev p-ja, ki prečkajo meje med kosi, je torej pomembnih le prvih in zadnjih $|p| - 1$ znakov posameznega kosa. To je koristno zato, ker so kosi

[6] Za poskus smo pognali našo rešitev na 10^9 naključnih testnih primerih velikosti $n = 100$. Povprečno število zamenjav je bilo pri tem res približno 1411,6, standardna deviacija pa 108,1; pri nobenem od teh 10^9 testnih primerov ni bilo treba več kot 2185 zamenjav in nikoli ni šlo z manj kot 879 zamenjavami.

pri tej nalogi lahko zelo dolgi (eksponentno v odvisnosti od k), niz p pa ni dolg; celega kosa $f^k(c_i)$ ne moremo izračunati, njegovih prvih in zadnjih $|p|-1$ znakov pa lahko.

Pišimo $q = |p|-1$ in definirajmo funkcijo $\gamma(u)$ takole:

$$\gamma(u) = \begin{cases} u[:q] \,\#\, u[-q:], & \text{če je } |u| > 2q \\ u & \text{sicer.} \end{cases}$$

Od niza u torej funkcija $\gamma(u)$ obdrži le prvih in zadnjih q znakov, vse vmes pa zamenja s posebnim ločilnim znakom # (ki se drugače v naših nizih ne pojavlja). Če je u dolg kvečjemu $2q$ znakov, pa funkcija γ obdrži celega.

Tako je torej $\gamma(x)$ tisto pri nizu u, kar je pomembno za štetje pojavitev p-ja na mejah med kosi. Nas bo to seveda najbolj zanimalo za nize oblike $u = f^k(x)$; definirajmo torej $g_k(x) = \gamma(f^k(x))$. Niza $g_k(x)$ ne moremo računati tako, da bi najprej izračunali $f^k(x)$ in ga potem oklestili, saj je lahko $f^k(x)$ za kaj takega predolg. Recimo, da je x dolg d znakov, i-tega od njih pa označimo z x_i. Ker je $f^k(x) = f^k(x_1)\dots f^k(x_n)$, je prvih q znakov niza $f^k(x)$ ravno prvih q znakov niza $f^k(x_1)$; če pa je le-ta krajši od q, moramo vzeti še prvih nekaj znakov niza $f^k(x_2)$ in tako naprej. Podoben razmislek velja tudi za zadnjih q znakov. Velja torej

$$g_k(x) = \gamma(G_k(x)) \quad \text{za} \quad G_k(x) := g_k(x_1)g_k(x_2)\dots g_k(x_d).$$

Tako lahko nize oblike $g_k(x)$ (ki so prijetno kratki: vsi so krajši od $2|p|$ znakov) računamo, ne da bi morali imeti kdaj opravka z (eksponentno dolgimi) nizi oblike $f^k(x)$.

Ko imamo niz $G_k(x)$ pred sabo, lahko v njem preštejemo pojavitve p-ja;[7] in s tem smo prešteli tudi tiste pojavitve p-ja v $f^k(x)$, ki prečkajo kakšno od mejá med dvema sosednjima kosoma. Če je kak kos slučajno krajši od $2|p|-1$ znakov, je zanj $g_k(x_i) = f^k(x_i)$ in smo v $G_k(x)$ dobili tudi tiste pojavitve p-ja v $f^k(x)$, ki v celoti ležijo znotraj kosa $f^k(x_i)$. Paziti moramo torej, da takih pojavitev ne bomo šteli dvojno — (1) ko seštevamo po vseh kosih tiste pojavitve, ki v celoti ležijo znotraj enega kosa, in (2) ko štejemo pojavitve p-ja v $g_k(x)$. Kose, pri katerih $g_k(x_i)$ ne vsebuje znaka # na sredi, lahko torej pri (1) preskočimo, saj bomo pojavitve p-ja v njih prešteli pri (2).

Število pojavitev p-ja v $f^k(x)$ označimo s $h_k(x)$. Zdaj imamo vse, kar potrebujemo, da zapišemo psevdokodo našega postopka:

podprogram KORAK(x, k, g_k, h_k):
vhod: niz $x = x_1 \dots x_d$, število k, vrednosti $g_k(t_i)$ in $h_k(t_i)$ za vse črke abecede;
izhod: vrednosti $g_k(x)$ in $h_k(x)$;

$u :=$ prazen niz; $H := 0$;

[7] Kako prešteti pojavitve p-ja — in, ne pozabimo, njegovih anagramov — v nekem daljšem nizu? Preprosta možnost je, da se po tem daljšem nizu premikamo z oknom dolžine $|p|$ znakov in v neki tabeli vzdržujemo „histogram“ — za vsako črko imamo število pojavitev te črke v oknu. Teh števil ni težko vzdrževati: ko se okno premakne za en znak naprej, pride na desni ena črka vanj, na levi pa pade ena črka iz njega, torej moramo popraviti največ dva elementa v histogramu. Vnaprej preštejmo tudi pojavitve vsake črke v nizu p, pa bomo lahko sproti preverjali, koliko črk se v oknu pojavlja manjkrat kot v p, koliko pa večkrat kot v p. Če ni nobene take črke, lahko zaključimo, da se vsaka črka v oknu pojavlja natanko tolikokrat kot v p, torej je v oknu nek anagram p-ja.

for $i := 1$ **to** d:
  if $g^k(x_i)$ vsebuje **#** **then** $H := H + h_k(x_i)$;
  dodaj $g_k(x_i)$ na konec niza u;
(* *Spremenljivka u zdaj vsebuje niz* $G_k(x)$. *)
$H := H+$ število pojavitev p-ja v nizu u;
vrni $g_k(x) = \gamma(u)$ in $h_k(x) = H$;

Zdaj torej znamo izračunati g_k in h_k za daljše nize iz njunih vrednosti za posamezne črke abecede; slednje pa dobimo tako, da upoštevamo, da je $f^k(t_i) = f^{k-1}(f(t_i))$:

za vsako črko t_i naše abecede:
  $g_0(t_i) = t_i$;
  if $p = t_i$ **then** $h_0(t_i) := 1$ else $h_0(t_i) := 0$;
for $r := 0$ **to** $k - 1$:
  za vsako črko t_i naše abecede:
    za $x = f(t_i)$ pokliči KORAK(x, r, g_r, h_r);
    vrednosti $g_r(x)$ in $h_r(x)$, ki ju je KORAK vrnil,
      si zapomni kot $g_{r+1}(t_i)$ in $h_{r+1}(t_i)$;
pokliči KORAK(s, k, g_k, r_k) in vrni $h_k(s)$, ki jo je izračunal;

V praksi moramo pri izračunu paziti še na to, da moramo na koncu vrniti število pojavitev po modulu M in da je pametno zato že med izračunom ves čas delati le z ostanki po deljenju s M, saj bi bilo pravo število pojavitev lahko preveliko in bi prekoračili obseg celoštevilskih spremenljivk.

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

string p; int pd, a, M; // pd je dolžina niza p

// Struktura Opis opisuje vse, kar moramo vedeti o nizih oblike f^r(x):
// stPojavitev = h_r(x), število pojavitev p-ja v nizu;
// s = g_r(x) = γ(f^r(x)) = prvih in zadnjih |p| − 1 znakov niza f^r(x), vmes pa
// ločilni znak (število a). Če je niz dolg ≤ 2(|p| − 1) znakov, je v s prisoten v celoti.
struct Opis { int stPojavitev; string s; };

// Izračuna opis niza f^r(x), če so znani opisi f^r(t_i) za posamezne črke abecede.
void Korak(const vector<Opis>& opisi, const string &x, Opis &O)
{
  O.stPojavitev = 0; string &fx = O.s; fx.clear();
  // V „fx“ pripravimo stik nizov γ(f^r(x[i])) in seštejmo pojavitve p-ja znotraj nizov f^r(x[i])
  // za posamezne črke niza x. Če je neki f^r(x[i]) dovolj kratek, da dobimo celega v fx,
  // pojavitev p-ja v njem ne upoštevajmo, saj jih bomo dobili kasneje, ko pregledamo fx.
  for (char c : x) { const Opis &Oc = opisi[c]; fx += Oc.s;
    if (Oc.s.length() == 2 * pd − 1 && Oc.s[pd − 1] == a)
      O.stPojavitev = (O.stPojavitev + Oc.stPojavitev) % M; }
  // Upoštevajmo pojavitve p-ja, ki prečkajo kakšno od meja med f^r(x[i]) in f^r(x[i + 1]).
  // Po nizu fx se premikamo z oknom dolžine |p|; stCrk[c] je razlika med številom pojavitev
  // črke c (za c = 0, . . . , a; c = a predstavlja ločilni znak) v oknu in v vzorcu p.
  // „stPremalo“ pove, pri koliko c-jih je stCrk[c] < 0; „stPrevec“ pa, pri koliko c-jih
  int fsd = fx.length(), stPremalo = 0, stPrevec = 0;          // velja stCrk[c] > 0.
```

```
  static int stCrk[27]; for (int i = 0; i <= a; ++i) stCrk[i] = 0;
  for (char c : p) if (--stCrk[c] == -1) ++stPremalo;
  for (int i = 0; i < fsd; ++i) {
    // Na desni je v okno fx[i - pd + 1, ..., i] prišel znak fx[i].
    int x = ++stCrk[fx[i]]; if (x == 0) --stPremalo; else if (x == 1) ++stPrevec;
    // Na levi je iz okna izpadel znak fx[i - pd].
    if (i >= pd) { x = --stCrk[fx[i - pd]];
      if (x == 0) --stPrevec; else if (x == -1) ++stPremalo; }
    // Ali je v oknu ravno anagram p-ja?
    if (stPremalo == 0 && stPrevec == 0) ++O.stPojavitev; }
  O.stPojavitev %= M;
  // Če je f^r(x) daljši od 2(|p| - 1), obdržimo le prvih in zadnjih |p| - 1 črk,
  // vmes pa damo ločilni znak (število a). Tako dobimo γ(f^r(x)).
  if (fsd > 2 * (pd - 1))
    fx = fx.substr(0, pd - 1) + char(a) + fx.substr(fsd - pd + 1);
};

int main()
{
  // Preberimo abecedo.
  string abeceda; getline(cin, abeceda); a = abeceda.length();
  // Pripravimo funkcijo, s katero bomo znake abecede prevedli v števila od 0 do a - 1.
  string xlat(26, ' '); for (int i = 0; i < a; ++i) xlat[abeceda[i] - 'a'] = i;
  auto Prevedi = [&xlat] (string &s) { for (char &c : s) c = xlat[c - 'a']; };
  // Preberimo nize f(t_i) za vse črke abecede in jih prevedimo.
  vector<string> f(a); for (auto &fa : f) { getline(cin, fa); Prevedi(fa); }
  // Preberimo ostale podatke.
  string s; int k; cin >> s >> k >> M >> p;
  Prevedi(s); Prevedi(p); pd = p.length();
  // Pripravimo opise nizov f^0(t_i).
  vector<Opis> opisi(a), noviOpisi(a);
  for (int i = 0; i < a; ++i)
    opisi[i] = { (pd == 1 && p[0] == i ? 1 : 0), string(1, char(i)) };
  // Izračunajmo opise f^r(t_i) za r = 1, ..., k.
  for (int r = 0; r < k; ++r) {
    for (int i = 0; i < a; ++i) Korak(opisi, f[i], noviOpisi[i]);
    swap(opisi, noviOpisi); }
  // Izračunajmo opis f^k(s) in izpišimo rezultat.
  Opis O; Korak(opisi, s, O);
  cout << O.stPojavitev << endl; return 0;
}
```

Razmislimo še o časovni zahtevnosti te rešitve. Podprogram KORAK mora za dani niz x dolžine d izračunati niz $G_k(x)$, ki je dolg $O(d|p|)$ znakov, ker so posamezni kosi v njem, nizi $g_k(x_i)$, dolgi po manj kot $2|p|$ znakov. Za izračun niza $G_k(x)$ torej porabimo $O(d|p|)$ časa, nato pa še $O(a + d|p|)$ časa za štetje pojavitev p-ja v njem. Glavni del programa mora pri vsakem r od 0 do $k-1$ poklicati KORAK za nize $f(t_i)$, pri čemer so t_i vse črke abecede; na koncu pa pokličemo še KORAK za niz s (dolžine n). Če vse to seštejemo, imamo časovno zahtevnost $O(k\,a^2 + |p|(n + k\,\phi))$, kjer smo s ϕ označili skupno dolžino nizov $f(t_i)$ za vseh a črk abecede.

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Seštevanje ulomkov

Naloga pravi, da vsak otrok razen zadnjega prereže prejeti kos na pol in potem eno polovico mogoče posadi v zemljo; rezov je torej $n-1$, zato je najmanjši možni posajeni kos težak $1/2^{n-1}$ kilograma. To maso bomo pri naših izračunih vzeli za osnovno enoto; vse druge mase, s kateri bomo imeli pri tej nalogi opravka, so večkratniki te osnovne enote, zato bomo lahko ves čas računali s celimi števili in nam ne bo treba skrbeti, da bi nam rezultat pokvarile kakšne zaokrožitvene napake.

Za torto, težko 1 kilogram, bomo torej zdaj rekli, da je težka 2^{n-1} enot. Kos, ki ga posadi prvi otrok (če se odloči saditi in ne jesti), je potem težak 2^{n-2} enot; kos, ki ga posadi drugi otrok, je težak 2^{n-3} enot in tako naprej. V splošnem je torej kos, ki ga posadi i-ti otrok, težak 2^{n-1-i} enot.

Podatke o praznovanjih berimo v zanki, pri vsakem praznovanju pa pojdimo v vgnezdeni zanki po otrocih in seštevajmo mase posajenih kosov torte. Na koncu tega postopka imamo pred sabo skupno maso posajenih kosov; recimo, da je to v enot po $1/2^{n-1}$ kilograma; če delimo v z 2^{n-2} in rezultat zaokrožimo navzdol, dobimo skupno maso v enotah po pol kilograma. Če je ta rezultat lih, to pomeni, da je od zadnjega kilograma prisotna vsaj polovica, torej moramo pri zaokrožanju na kilogram zaokrožiti navzgor (kar lahko naredimo tako, da masi v polkilogramskih enotah prištejemo 1, preden jo nazadnje delimo z 2, da dobimo maso v kilogramskih enotah).

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

int main()
{
  // Preberimo število praznovanj in otrok.
  int r, n; cin >> r >> n;

  // Obdelajmo vsa praznovanja.
  int vsota = 0;
  while (r-- > 0)
  {
    // Preberimo niz z opisom praznovanja.
    string s; cin >> s;

    // Prištejmo posajene kose k vsoti.
    for (int i = 0; i < n; ++i)
      if (s[i] == 'S') vsota += 1 << (n - 2 - i);
  }
  // Zaokrožimo vsoto navzdol na polovico kilograma.
  vsota >>= n - 2;

  // Če je število polkilogramskih enot liho, moramo pri
  // pretvorbi v kilograme zaokrožiti navzgor.
  vsota = (vsota + (vsota & 1)) >> 1;

  // Izpišimo rezultat.
  cout << vsota << endl; return 0;
}
```

Lahko pa nalogo rešimo tudi z uporabo aritmetike s plavajočo vejico, saj imajo ulomki, s kakršnimi delamo pri tej nalogi, vsi za imenovalec neko potenco števila 2, take pa lahko v predstavitvi s plavajočo vejico predstavimo brez napak, dokler je v mantisi dovolj prostora. Pri naši nalogi imamo opravka z vrednostmi od 0 do r in potrebujemo natančnost do $1/2^{n-1}$, kar pri $r = 1000$ in $n = 20$ pomeni kakšnih 29 bitov; tip **double** ima mantiso dolgo 52 bitov, torej več kot dovolj, da do zaokrožitvenih napak ne bo prihajalo. Za izpis vsote v kilogramih na koncu lahko uporabimo funkcijo round iz standardne knjižnice, ki polovice zaokroža stran od 0, prav to pa naša naloga tudi zahteva.

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

int main()
{
  // Preberimo število praznovanj in otrok.
  int r, n; cin >> r >> n;

  // Obdelajmo vsa praznovanja.
  double vsota = 0;
  while (r-- > 0)
  {
    // Preberimo niz z opisom praznovanja.
    string s; cin >> s;

    // Prištejmo posajene kose k vsoti.
    double kos = 1;
    for (int i = 0; i < n; ++i) {
      kos /= 2;
      if (s[i] == 'S') vsota += kos; }
  }

  // Izpišimo vsoto, zaokroženo na celo število kilogramov.
  cout << round(vsota) << endl; return 0;
}
```

Zapišimo še rešitev v pythonu. Lahko bi računali s celimi števili kot pri prvi od gornjih dveh rešitev ali pa s plavajočo vejico (v pythonu sta primerna tipa float in Decimal) kot pri drugi, lahko pa namesto tega uporabimo tip Fraction, s katerim lahko brez zaokrožitvenih napak računamo z racionalnimi števili (ulomki):

```
import sys, fractions, math

# Preberimo število praznovanj in otrok.
r, n = (int(s) for s in sys.stdin.readline().split())

# Obdelajmo vsa praznovanja.
vsota = 0
for ri in range(r):

  # Preberimo niz z opisom praznovanja.
  s = sys.stdin.readline()

  # Prištejmo posajene kose k vsoti.
  for i in range(n):
    if s[i] == 'S': vsota = vsota + fractions.Fraction(1, 2**(i + 1))

# Vsota je zdaj racionalno število v kilogramih.
```

```
# Zaokrožimo jo navzdol na celo število v polovicah kilograma.
vsota = math.floor(2 * vsota)

# Če je število polkilogramskih enot liho, moramo pri
# pretvorbi v kilograme zaokrožiti navzgor.
print((vsota + (vsota % 2)) // 2)
```

Za zaokrožanje nismo uporabili funkcije round, kajti ta v pythonu zaokroža polovice na najbližje sodo število (npr. 3,5 $\mapsto$ 4 in tudi 4,5 $\mapsto$ 4) in ne vedno navzgor, kot zahteva naša naloga.

To rešitev lahko zapišemo tudi krajše:

```
import sys, fractions, math

# Preberimo število praznovanj in otrok.
r, n = (int(s) for s in sys.stdin.readline().split())

# Obdelajmo vsa praznovanja.
vsota = math.floor(sum(fractions.Fraction(1, 2**i)
                       for ri in range(r) for i, c in enumerate(sys.stdin.readline()) if c == 'S'))

# Vsota je zdaj zaokrožena navzdol na celo število polkilogramskih enot.
# Če je to število liho, moramo pri pretvorbi v kilograme zaokrožiti navzgor.
print((vsota + (vsota % 2)) // 2)
```

2. Slovar

(*a*) Prvi del naloge je enostaven: pojdimo v zanki po vseh besedah slovarja; pri vsaki besedi najprej preverimo, če je enako dolga kot vzorec; če ni, lahko takoj zaključimo, da se ne bo ujemala z danim vzorcem; sicer pa pojdimo v vgnezdeni zanki po črkah besede in vzorca hkrati ter preverjajmo, če se ujemajo. Slednje pomeni, da mora imeti bodisi vzorec na tistem mestu zvezdico bodisi morata imeti vzorec in beseda na tistem mestu enako črko. Čim opazimo kakšno neujemanje, lahko preverjanje trenutne besede prekinemo, saj že vemo, da ne bo ustrezala našemu vzorcu. Če pa pridemo do konca besede, ne da bi opazili kakšno neujemanje, jo izpišimo.

Primer implementacije take rešitve v C++ (recimo, da slovar dobimo kot vektor nizov):

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

void PoisciA(const vector<string>& slovar, const string& vzorec)
{
  int d = vzorec.length();
  for (const auto &beseda : slovar)
  {
    // Preverimo, če je beseda enako dolga kot vzorec.
    if (beseda.length() != d) continue;

    // Preverimo, če se znaki besede ujemajo z istoležnimi znaki vzorca.
    int i = 0; while (i < d && (vzorec[i] == '*' || vzorec[i] == beseda[i])) ++i;

    // Če smo prišli do konca, ne da bi opazili kakšno neujemanje,
    // je beseda ustrezna in jo izpišimo.
    if (i >= d) cout << beseda << endl;
  }
}
```

Zapišimo to rešitev še v pythonu; slovar tu pričakujemo kot seznam (pythonov tip list) oz. kot karkoli, po čemer je mogoče iterirati:

```
def PoisciA(slovar, vzorec):
  d = len(vzorec)
  for beseda in slovar:
    # Preverimo, če je beseda enako dolga kot vzorec.
    if len(beseda) != d: continue
    # Preverimo, če se znaki besede ujemajo z istoležnimi znaki vzorca.
    i = 0
    while i < d and (vzorec[i] == '*' or vzorec[i] == beseda[i]): i += 1
    # Če smo prišli do konca, ne da bi opazili kakšno neujemanje,
    # je beseda ustrezna in jo izpišimo.
    if i >= d: print(beseda)
```

Še ena možnost je, da uporabimo regularne izraze (ti so, mimogrede, na voljo tudi v C++ovi standardni knjižnici), pri čemer pa pazimo na to, da se v njih za ujemanje s poljubnim znakom uporablja pika in ne zvezdica:

```
import re
def PoisciA2(slovar, vzorec):
  r = re.compile(vzorec.replace("*", "."))
  for beseda in slovar:
    if r.match(beseda): print(beseda)
```

Še ena možnost v pythonu pa je modul fnmatch, pri katerem se za ujemanje s poljubnim znakom uporablja vprašaj namesto zvezdice:

```
import fnmatch
def PoisciA3(slovar, vzorec):
  vzorec = vzorec.replace("*", "?")
  for beseda in slovar:
    if fnmatch.fnmatch(beseda, vzorec): print(beseda)
```

(b) Razmislimo zdaj o drugem delu naloge, pri katerem bi radi slovar vnaprej predelali tako, da bomo lahko po njem čim hitreje iskali besede, ki se ujemajo z vzorci. Za začetek je koristno besede slovarja razdeliti na več seznamov po dolžini, saj se lahko vzorec dolžine d ujema samo z besedami dolžine d, daljših ali krajših besed pa sploh nima smisla pregledovati.

Recimo zdaj, da i-ti znak vzorca ni zvezdica, ampak neka črka, recimo c. Potem pridejo v poštev le tiste besede iz slovarja, ki imajo na i-tem mestu prav to črko in ne kakšne druge. Koristno bi bilo torej za vsako dolžino d, za vsak indeks i od 1 do d in za vsako črko abecede c pripraviti seznam tistih besed iz slovarja, ki so dolge natanko d znakov in imajo na i-tem mestu črko c; recimo temu seznamu $A(d, i, c)$.

Ko hočemo potem za neki vzorec (dolžine d) poiskati besede, ki se ujemajo z njim, poiščimo v njem poljubno mesto i, kjer ni zvezdice, ampak neka črka c; potem moramo le pregledati vse besede iz $A(d, i, c)$ in za vsako od njih preveriti, ali se ujema tudi s preostankom vzorca. Za to lahko uporabimo enak postopek kot pri podnalogi (a). Če je ima vzorec na več mestih črko (in ne zvezdice), imamo torej na voljo več primernih $A(d, i, c)$ in je smiselno pregledati najkrajšega med njimi, da bomo imeli čim manj dela.

Poseben primer nastopi, če v vzorcu ni nobene črke, ampak same zvezdice. Tedaj nam razmislek iz prejšnjega odstavka ne ponudi nobenega seznama možnih kandidatov, pač pa gremo lahko pri poljubnem i po vseh črkah abecede in za vsako črko c izpišemo vse besede iz $A(d, i, c)$; tako bomo sčasoma izpisali prav vse besede dolžine d, to pa je pri takem vzorcu tudi pravilni odgovor.

```
int maxDolzina;
vector<vector<string>> seznami;

void Pripravi(const vector<string>& slovar)
{
  // Poglejmo, kako dolga je najdaljša beseda.
  maxDolzina = 0;
  for (const auto &beseda : slovar) maxDolzina = max(maxDolzina, int(beseda.length()));
  // Pripravimo si dovolj seznamov.
  seznami.clear(); seznami.resize((maxDolzina + 1) * (maxDolzina + 1) * 26);
  // Pri vsaki besedi pojdimo po vseh njenih črkah in jo dodajmo
  // v ustrezni seznam.
  for (const auto &beseda : slovar)
    for (int d = beseda.size(), i = 0; i < d; ++i)
      seznami[(d * (maxDolzina + 1) + i) * 26 + (beseda[i] - 'a')].push_back(beseda);
}

void PoisciB(const string& vzorec) const
{
  // Morda je ta vzorec daljši od vseh besed v slovarju.
  int d = vzorec.length(); if (d > maxDolzina) return;
  // Poglejmo, pri katerem i dobimo najkrajši seznam kandidatov.
  int najIdx = -1, najDolz = 0;
  for (int i = 0; i < d; ++i) if (vzorec[i] != '*') {
    int idx = (d * (maxDolzina + 1) + i) * 26 + (vzorec[i] - 'a');
    int dolzina = seznami[idx].size();
    if (najIdx < 0 || dolzina < najDolz) najIdx = idx, najDolz = dolzina; }
  // Če smo tak seznam našli, ga preiščimo.
  if (najIdx >= 0) { PoisciA(seznami[najIdx], vzorec); return; }
  // Sicer ima vzorec same zvezdice in moramo izpisati vse nize dolžine d.
  for (int c = 0; c < 26; ++c)
    for (const auto &beseda : seznami[d * (maxDolzina + 1) * 26 + c])
      cout << beseda << endl;
}
```

Še podobna rešitev v pythonu:

```
def Pripravi(slovar):
  global maxDolzina, seznami
  # Poglejmo, kako dolga je najdaljša beseda.
  maxDolzina = max(len(beseda) for beseda in slovar)
  # Pripravimo si dovolj seznamov.
  seznami = [[[[] for c in range(26)] for i in range(d)] for d in range(maxDolzina + 1)]
  # Pri vsaki besedi pojdimo po vseh njenih črkah in jo dodajmo v ustrezni seznam.
  for beseda in slovar:
    d = len(beseda)
    for i in range(d):
```

```
        seznami[d][i][ord(beseda[i]) − ord('a')].append(beseda)

def PoisciB(vzorec):
  # Morda je ta vzorec daljši od vseh besed v slovarju.
  d = len(vzorec)
  if d > maxDolzina: return

  # Med seznami kandidatov za vsako črko vzorca preiščimo najkrajšega.
  kandidati = min((seznami[d][i][ord(c) − ord('a')] for i, c in enumerate(vzorec) if c != '*'),
                  key = len, default = None)
  if kandidati is not None: PoisciA(kandidati, vzorec); return

  # Če ni nobenega takega seznama, ker ima vzorec same zvezdice,
  # moramo izpisati vse nize dolžine d.
  for seznam in seznami[d][0]:
    for beseda in seznam: print(beseda)
```

3. Genialno

Preprosta, vendar manj učinkovita rešitev je, da gremo v zanki po vseh poljih mreže in pri vsakem od njih izračunamo število točk po pravilu iz besedila naloge. To pomeni, da potrebujemo še eno vgnezdeno zanko, ki gre po vseh štirih možnih smereh (gor, dol, levo in desno), za vsako smer pa imamo potem še eno zanko, ki se premika od trenutnega polja v tisti smeri in sešteva število žetonov na tako obiskanih poljih, ustavi pa se, ko bodisi pride do praznega polja (takega z nič žetoni) ali pa doseže rob mreže.

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

// a[y * w + x] = število žetonov na polju (x, y) za 0 ≤ x < w, 0 ≤ y < h.
pair<int, int> KamPostaviti(int w, int h, const vector<int> &a)
{
  int xNaj = −1, yNaj = −1, najTock = −1;
  const int DX[] = { −1, 1, 0, 0 }, DY[] = { 0, 0, 1, −1 };
  for (int y = 0; y < h; ++y) for (int x = 0; x < w; ++x)
  {
    int tocke = 0;
    // Pojdimo iz polja (x, y) v vse štiri smeri.
    for (int smer = 0; smer < 4; ++smer) for (int xx = x, yy = y; ; )
    {
      // Naredimo korak naprej v trenutno smer.
      xx += DX[smer]; yy += DY[smer];

      // Če smo padli čez rob mreže, končajmo.
      if (xx < 0 || xx >= w || yy < 0 || yy >= h) break;

      // Če smo na praznem polju, tudi končajmo.
      int zetoni = a[yy * w + xx]; if (zetoni == 0) break;

      // Sicer bomo dobili toliko točk, kolikor je tu žetonov.
      tocke += zetoni;
    }
    // Najboljši rezultat doslej si zapomnimo.
    if (tocke > najTock) najTock = tocke, xNaj = x, yNaj = y;
  }
```

```
  return {xNaj, yNaj};
}
```

Zapišimo podobno rešitev še v pythonu:

```
# a[y * w + x] = število žetonov na polju (x, y) za 0 ≤ x < w, 0 ≤ y < h.
def KamPostaviti(w, h, a):
  # Vrne točke, ki jih dobimo iz ene od smeri, če postavimo žeton na (x, y).
  def TockeS(x, y, smer):
    DX = [−1, 1, 0, 0][smer]; DY = [0, 0, 1, −1][smer]
    tocke = 0
    while True:
      # Naredimo korak naprej v trenutno smer.
      x += DX; y += DY

      # Če smo padli čez rob mreže, končajmo.
      if x < 0 or x >= w or y < 0 or y >= h: break

      # Če smo na praznem polju, tudi končajmo.
      zetoni = a[y * w + x]
      if zetoni == 0: break

      # Sicer bomo dobili toliko točk, kolikor je tu žetonov.
      tocke += zetoni
    return tocke
  # Vrne skupno število točk, ki ga dobimo, če postavimo žeton na (x, y).
  def Tocke(x, y): return sum(TockeS(x, y, smer) for smer in range(4))
  # Izračunajmo najboljši položaj novega žetona in ga vrnimo.
  return max((Tocke(x, y), (x, y)) for y in range(h) for x in range(w))[1]
```

Slabost dosedanje rešitve je, da je razmeroma počasna. V najslabšem primeru (če so vsa polja mreže neprazna) se bo najbolj notranja zanka vedno premikala vse do roba mreže, zato bomo pri vsakem (x, y) pregledali celotno vrstico y in celoten stolpec x, skupaj torej $w + h$ polj; časovna zahtevnost te rešitve je torej $O(wh(w + h))$.

Toda predstavljajmo si maksimalno strnjeno skupino nepraznih polj, ki so vsa v isti vrstici, recimo od (x_1, y) do (x_2, y). Ko pravimo, da naj bo maksimalna, hočemo s tem reči, naj na levi in desni mejí bodisi na rob mreže bodisi na prazno polje. Recimo, da razmišljamo o tem, da bi žeton postavili na neko polje (x, y) te skupine (torej za $x_1 \leq x \leq x_2$); s premikanjem levo in desno od njega bomo obiskali ravno celo skupino, preden se bomo ustavili. To pomeni, da je treba vsoto vseh žetonov te skupine šteti k točkam vsakega polja v skupini. Podobno je tudi, če razmišljamo o postavitvi žetona tik levo od skupine, na $(x_1 - 1, y)$; s premikanjem desno od tam bomo obiskali ravno celo skupino. Podobno je tudi pri postavljanju žetona tik desno od skupine, na $(x_2 + 1, y)$, ko obiščemo celo skupino med premikanjem levo.

Vidimo torej, da ko enkrat najdemo tako skupino, lahko izračunamo vsoto žetonov na njej in jo prištejemo k točkam vseh polj skupine in še sosednjih polj pred in za skupino. Če je skupina dolga d polj, smo imeli z njo $O(d)$ dela; in ker pripada vsako neprazno polje natanko eni taki skupini, so vse skupine skupaj dolge $O(wh)$ polj, zato imamo tudi z vsemi skupaj le $O(wh)$ dela. Tako sčasoma dobimo vse točke (za vse možne položaje novega žetona), ki nastanejo zaradi premikanja levo in desno od položaja novega žetona. V nadaljevanju moramo popolnoma enak razmislek ponoviti še po stolpcih namesto po vrsticah, da bomo upoštevali še točke, ki

nastanejo zaradi premikanja gor in dol, pa bomo na koncu dobili pravo število točk vsakega polja.

Tako imamo rešitev s časovno zahtevnostjo $O(wh)$, kar je veliko manj od prejšnje; njena slabost pa je v večji porabi pomnilnika, saj potrebujemo tabelo $w \cdot h$ elementov, v kateri računamo točke vsake celice.

```
pair<int, int> KamPostaviti2(int w, int h, const vector<int> &a)
{
  vector<int> tocke(w * h, 0);
  for (int obrni = 0; obrni < 2; ++obrni)
  {
    // Spodnji postopek je tak, kot da iščemo vodoravne skupine neničelnih polj; toda
    // pri obrni = 1 koordinatne osi obrnemo, tako da v resnici iščemo navpične skupine.
    int W = w, H = h; if (obrni) swap(W, H);
    auto A = [=, &a] (int X, int Y) { return a[obrni ? X * w + Y : Y * w + X]; };

    for (int Y = 0; Y < H; ++Y) for (int X = 0; X < W; ++X) if (A(X, Y) != 0)
    {
      // Tu se začne skupina neničelnih polj.
      // Kako daleč sega in kakšna je njihova vsota?
      int XX = X + 1, vsota = A(X, Y);
      while (XX < W && A(XX, Y) != 0) vsota += A(XX++, Y);

      // Prištejmo to vsoto k točkam polj v skupini in še tistih tik pred in za njo.
      for (int t = max(X - 1, 0); t < min(XX + 1, W); ++t)
        tocke[obrni ? t * w + Y : Y * w + t] += vsota - A(t, Y);

      X = XX; // Nadaljujmo za koncem te skupine.
    }
  }
  // Vrnimo najboljši položaj novega žetona.
  int naj = 0; for (int i = 1; i < w * h; ++i) if (tocke[i] > tocke[naj]) naj = i;
  return {naj % w, naj / w};
}
```

Zapišimo to rešitev še v pythonu:

```
def KamPostaviti2(w, h, a):
  tocke = [0] * (w * h)
  for obrni in range(2):
    # Spodnji postopek je tak, kot da iščemo vodoravne skupine neničelnih polj; toda
    # pri obrni = 1 koordinatne osi obrnemo, tako da v resnici iščemo navpične skupine.
    W, H = (h, w) if obrni else (w, h)
    def A(X, Y): return a[X * w + Y if obrni else Y * w + X]

    for Y in range(H):
      X = 0
      while X < W:
        vsota = A(X, Y)
        if vsota == 0: X += 1; continue

        # Tu se začne skupina neničelnih polj.
        # Kako daleč sega in kakšna je njihova vsota?
        XX = X + 1
        while XX < W and A(XX, Y) != 0: vsota += A(XX, Y); XX += 1

        # Prištejmo to vsoto k točkam polj v skupini in še tistih tik pred in za njo.
        for t in range(max(0, X - 1), min(XX + 1, W)):
          tocke[t * w + Y if obrni else Y * w + t] += vsota - A(t, Y)
```

```
        X = XX + 1 # nadaljujmo za to skupino
    # Izračunajmo najboljši položaj novega žetona in ga vrnimo.
    naj = max((tocke[i], i) for i in range(w * h))[1]
    return naj % w, naj // w
```

4. Parkirišče

Naloga pravzaprav sprašuje po tem, kakšna je največja skupna dolžina tovornjakov, ki so hkrati parkirani na našem parkirišču. Ta skupna dolžina se spremeni le ob prihodu ali odhodu kakšnega tovornjaka, torej (za vsak i od 1 do n) ob času p_i (ko se skupna dolžina poveča za d_i zaradi prihoda tovornjaka številka i) in ob času $p_i + t_i$ (ko se skupna dolžina zmanjša za d_i zaradi odhoda tovornjaka številka i). Dovolj bo torej, če za vsakega od teh časov izračunamo, kakšna bo skupna dolžina parkiranih tovornjakov po tej spremembi. To pa je najlaže računati po naraščajočih časih, saj lahko dobimo novo dolžino iz stare preprosto tako, da slednji prištejemo d_i ali ga od nje odštejemo.

Pripravimo torej seznam parov (p_i, d_i) in $(p_i + t_i, -d_i)$ za vse $i = 1, \ldots, n$. Prva komponenta pove čas, druga pa spremembo dolžine. Uredimo jih naraščajoče po času, tiste z enakim časom pa po drugi komponenti. V tako urejenem zaporedju moramo zdaj le seštevati druge komponente, pa bomo imeli pred seboj skupno dolžino po vsaki od sprememb.

Ker smo rekli, da pare z enakim časom uredimo po drugi komponenti, bomo v primerih, ko hkrati neki tovornjak pride in neki drug tovornjak odide, najprej obdelali odhode (kajti pri teh je druga komponenta negativna) in šele potem prihode (pri katerih je druga komponenta pozitivna), prav to pa naloga tudi zahteva.

Če nastopi več sprememb ob istem času, sicer skupna dolžina „med“ temi spremembami nima nobenega smisla (ker so spremembe pač istočasne in naloga tudi pravi, da se zgodijo v hipu), vendar tudi ne bo pokvarila naših rezultatov, saj nas zanima le največja možna skupna dolžina, ta pa bo nastopila bodisi po vseh teh spremembah (ko bodo upoštevani že vsi prihodi ob tem času) bodisi pred njimi (ko ne bo upoštevan še nobeden izmed odhodov ob tem času), ni pa mogoče, da bi bila kakšna od teh nesmiselnih vmesnih skupnih dolžin večja od obeh prej omenjenih.[8]

Oglejmo si še implementacijo te rešitve v C++:

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

struct Tovornjak { int dolzina, prihod, trajanje; };

int Parkirisce(const vector<Tovornjak> &tovornjaki)
{
  // Pripravimo podatke o spremembah skupne dolžine.
  vector<pair<int, int>> spremembe;
  for (const auto &t : tovornjaki) {
    spremembe.emplace_back(t.prihod, t.dolzina);
```

[8] Natančneje povedano: če smo upoštevali že nekaj prihodov ob tem času, ne pa še vseh, se bo skupna dolžina še povečala, ko bomo upoštevali še preostale prihode ob tem času. Če pa smo upoštevali že nekaj odhodov ob tem času, ne pa še vseh, potem je bila skupna dolžina večja, še preden smo upoštevali tistih dosedanjih nekaj odhodov.

```
    spremembe.emplace_back(t.prihod + t.trajanje, −t.dolzina); }
  // Uredimo spremembe po času.
  sort(spremembe.begin(), spremembe.end());
  // Računajmo skupno dolžino po vsaki spremembi.
  int maxDolzina = 0, dolzina = 0;
  for (auto [cas, sprememba] : spremembe) {
    dolzina += sprememba;
    // Najdaljšo skupno dolžino si zapomnimo.
    maxDolzina = max(maxDolzina, dolzina); }
  return maxDolzina;
}
```

Zapišimo to rešitev še v pythonu:

```
# Podprogram Parkirisce pričakuje kot parameter seznam takšnih objektov:
class Tovornjak: __slots__ = ["dolzina", "prihod", "trajanje"]

def Parkirisce(tovornjaki):
  # Pripravimo podatke o spremembah skupne dolžine.
  spremembe = [(t.prihod, t.dolzina) for t in tovornjaki]
  spremembe += [(t.prihod + t.trajanje, −t.dolzina) for t in tovornjaki]
  # Uredimo spremembe po času.
  spremembe.sort()
  # Računajmo skupno dolžino po vsaki spremembi.
  maxDolzina = 0; dolzina = 0
  for cas, sprememba in spremembe:
    dolzina += sprememba
    maxDolzina = max(maxDolzina, dolzina)    # Najdaljšo skupno dolžino si zapomnimo.
  return maxDolzina
```

5. Barvanje stolpnic

Povezanost med stolpnicami, kot je definirana v tej nalogi, ima naslednjo zanimivo lastnost: med stolpnicami, ki so povezane s stolpnico i, sta lahko največ dve taki, ki sta višji od i, in sicer ena levo od i ter ena desno od i; vse ostale stolpnice, s katerimi je i povezana, morajo biti nižje od nje.

O tem se lahko prepričamo takole: recimo, da se od i pomikamo desno; naj bo j prva višja stolpnica (višja od i-te), na katero naletimo. Ker so vmes vse stolpnice nižje od i-te, sta i in j povezani. Recimo zdaj, da bi bila nekje še bolj desno neka stolpnica k, ki bi bila tudi višja od i in povezana z njo; toda to bi pomenilo, da so vse stolpnice med i in k nižje od i (in tudi od k), kar pa ni res, saj je med temi tudi stolpnica j, ki je višja od i. Torej je lahko desno od i največ ena stolpnica, ki je višja od i in povezana z njo (in sicer je to kar prva višja stolpnica, ki stoji desno od i-te). Podoben razmislek lahko potem seveda opravimo tudi za stolpnice levo od i in se prepričamo, da je tudi tam i povezana z največ eno tako, ki je višja od nje (in sicer je to kar prva višja stolpnica levo od i-te). □

Zaradi te lastnosti je koristno barvati stolpnice od višjih proti nižjim. Ko pride na vrsto recimo stolpnica i, so bile doslej že pobarvane le tiste stolpnice, ki so višje od nje; med njimi pa sta z i povezani največ dve. Dodelimo i-ju najmanjšo tako številko barve (spomnimo se, da označujemo barve z naravnimi števili od 1 naprej),

ki je še nima nobena od tistih (največ) dveh z njo povezanih stolpnic, ki smo ju že pobarvali. Ena od barv 1, 2 in 3 bo torej gotovo primerna, saj nam tisti dve stolpnici lahko prepovesta največ dve od teh treh barv. Tako torej vidimo, da lahko stolpnice zagotovo pobarvamo s tremi barvami.

Za učinkovito izvedbo tega postopka je koristno razmisliti še o tem, kako naj ugotovimo, s katerima višjima stolpnicama je i povezana; z drugimi besedami, katera je prva višja stolpnica levo od i (recimo ji $L[i]$) in katera desno od i (recimo ji $D[i]$). Preprosta rešitev je, da gremo pri vsakem i v zanki po stolpnicah levo od i, dokler ne naletimo na prvo višjo od i; in potem podobno še v zanki po stolpnicah desno od i. Slabost te rešitve je, da v najslabšem primeru porabi $O(n^2)$ časa.

Boljša možnost je, da ko barvamo stolpnice padajoče po višini, vsako že pobarvano stolpnico dodamo v neko primerno uravnoteženo drevesasto podatkovno strukturo (na primer rdeče-črno drevo; v C++ lahko uporabimo razred `set`), v kateri bodo stolpnice urejene po svojem indeksu. Ko barvamo stolpnico i, so v tem drevesu že vse stolpnice, višje od i (in samo one); poiščimo torej v njem prvo stolpnico z indeksom $> i$, pa bo to ravno prva višja stolpnica desno od i, torej $D[i]$; podobno lahko v drevesu poiščemo zadnjo stolpnico z indeksom $< i$, pa bo to ravno prva višja stolpnica levo od i, torej $L[i]$. Tako imamo pri vsaki stolpnici $O(\log n)$ dela za tidve iskanji po drevesu (pa tudi za dodajanje stolpnice i v drevo, ko jo pobarvamo) in časovna zahtevnost celotnega postopka je $O(n \log n)$.

Še hitreje pa gre, če računamo vrednosti $L[i]$ od leve proti desni. Najbolj leva stolpnica seveda sploh nima nobene višje na svoji levi, zato postavimo $L[1] = 0$. Od tam naprej pri vsakem i razmišljajmo takole: en kandidat za $L[i]$ je i-jeva leva soseda, torej $i - 1$. Če je ta višja od i-te stolpnice, se ustavimo in jo razglasimo za $L[i]$; če pa je $i - 1$ prenizka, je potem naslednji primerni kandidat za $L[i]$ šele prva taka stolpnica, ki stoji levo od $i-1$ in je višja od stolpnice $i-1$ — to pa je stolpnica $L[i-1]$. Če je tudi ta prenizka (nižja od i-te stolpnice), je naslednji kandidat potem $L[L[i-1]]$ in tako naprej. Zapišimo ta postopek s psevdokodo:

```
for i := 1 to n:
  c := i − 1;
  while c > 0:
    if h_c > h_i then break
    else c := L[c];
  L[i] := c;
```

Na podoben način lahko računamo tudi $D[i]$, le ta gremo tam od desne proti levi. Kakšna je časovna zahtevnost tega postopka? Predstavljajmo si, da bi med izvajanjem našega postopka vzdrževali sklad, na katerem so stolpnice i, $L[i]$, $L[L[i]]$ in tako naprej, vse višje in vse bolj leve, dokler se pač to zaporedje ne konča pri neki stolpnici j, ki ima $L[j] = 0$. Zgornji postopek potem pri i pravzaprav naredi naslednje: z vrha sklada pobere stolpnice, nižje od h_i, in nato na vrh sklada doda stolpnico i. Ker torej vsako stolpnico enkrat dodamo na sklad, jo tudi največ enkrat poberemo s sklada, zato je časovna zahtevnost vseh teh operacij skupaj le $O(n)$.

S tem, ko smo vse $L[i]$ in $D[i]$ izračunali v $O(n)$ časa namesto $O(n \log n)$ kot pri rešitvi z drevesom, sicer nismo veliko pridobili, saj bomo stolpnice kasneje še vedno barvali padajoče po višini, torej jih moramo še vedno urediti in to nam bo še vedno vzelo $O(n \log n)$ časa. Ima pa zadnji postopek za izračun $L[i]$ in $D[i]$ vseeno to

prednost, da ne potrebuje drevesa (kar je koristno npr. v pythonu, kjer v standardni knjižnici takega drevesa nimamo).[9]

Zdaj torej znamo pobarvati stolpnice s kvečjemu tremi barvami, ostane pa še vprašanje, ali obstaja kakšno barvanje z manj barvami od tistega, ki ga najde naš dosedanji postopek. Pri $n = 0$ ali $n = 1$ je možno seveda barvanje z n barvami in naš postopek ga bo tudi našel; pri $n \geq 2$ pa gotovo potrebujemo vsaj dve barvi, saj sta najvišji dve stolpnici med seboj povezani. Pa recimo, da pri nekem razporedu stolpnic zadoščata že samo dve barvi, naš postopek pa jih pobarva s tremi. Naš postopek najvišji stolpnici vedno dá barvo 1; vzemimo neko barvanje z dvema barvama (seveda takšno, v katerem nobeni dve povezani stolpnici nista iste barve) in če najvišja stolpnica v njem nima barve 1, preprosto obrnimo barve vseh stolpnic v njem. Naj bo zdaj $d = (d_1, \ldots, d_n)$ zaporedje barv stolpnic v tem barvanju z dvema barvama, $t = (t_1, \ldots, t_n)$ pa v barvanju s tremi barvami, ki ga je našel naš postopek.

Pojdimo zdaj po stolpnicah od višjih proti nižjim in primerjajmo d_i in t_i; pri prvih nekaj stolpnicah se barve mogoče ujemajo, prej ali slej pa mora nastopiti razlika, recimo pri stolpnici i. Neujemanje med t_i in d_i lahko nastopi na dva načina: (1) lahko da je $t_i = 3$; toda če je naš postopek uporabil barvo 3, to pomeni, da je i povezana z dvema višjima stolpnicama j in k, ki sta bili barv 1 in 2; in ker se pri višjih stolpnicah t in d ujemata, imata j in k barvi 1 in 2 tudi v d; in ker je v d tudi stolpnica i ene od teh dveh barv, sta tam dve povezani stolpnici enake barve, kar je protislovje. (2) Neujemanje lahko nastopi torej le tako, da je $t_i = 3 - d_i$; toda ker i ni najvišja, je gotovo povezana z vsaj eno višjo stolpnico, recimo j; ta ima v obeh barvanjih enako barvo, $t_j = d_j$; ker sta i in j povezani, naš postopek ni pobarval i z enako barvo kot j, torej je $t_i = 3 - t_j = 3 - d_j$, in ko to združimo z $t_i = 3 - d_i$, vidimo, da mora biti $d_i = d_j$, torej sta v d dve povezani stolpnici enake barve, kar je spet protislovje.

Vidimo torej, da nas predpostavka, da je naš postopek porabil tri barve na takem razporedu, ki ga je mogoče pobarvati že z dvema, neizogibno pripelje v protislovje, torej naš postopek na takem razporedu res porabi samo dve barvi. □

Kot zanimivost razmislimo še, kdaj pravzaprav obstaja barvanje z dvema barvama. (1) Če je kakšna stolpnica nižja od obeh svojih sosed, torej če pri nekem i velja $h_{i-1} > h_i < h_{i+1}$, so vse tri povezane druga z drugo, zato jih je nemogoče pobarvati s samo dvema barvama. (2) Če pa ni nobena stolpnica nižja od obeh svojih sosed, to pomeni, da če pri nekem i velja $h_i > h_{i+1}$, morajo od tam naprej višine ves čas le še padati (kajti če bi začele spet naraščati, bi imeli tam stolpnico, ki je nižja od obeh svojih sosed). Primer (2) lahko torej nastopi le tako, da (če gremo od leve proti desni) višine stolpnic najprej nekaj časa (lahko tudi 0 korakov) samo naraščajo, nato pa (če zaporedja še ni konec) do konca samo padajo. Tedaj je vsaka stolpnica povezana le s svojima neposrednima sosedama in z nobeno drugo, zato jih lahko pobarvamo z dvema barvama preprosto tako, da gremo od leve proti desni in izmenično uporabljamo zdaj eno, zdaj drugo barvo. □

Čeprav naloga zahteva le opis postopka, si vseeno oglejmo še primer implementacije v C++; najprej rešitev z drevesom:

[9] Poleg tega, če morda višine stolpnic tvorijo permutacijo števil od 1 do n ali kaj podobnega (česar sicer naloga ne zagotavlja), jih lahko uredimo že v $O(n)$ časa in je zato naš zadnji postopek za izračun $L[i]$ in $D[i]$ še tem koristnejši.

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

void Pobarvaj(const vector<int> &h, vector<int> &barva)
{
  int n = h.size(); barva.resize(n);

  // Uredimo stolpnice padajoče po višini.
  vector<pair<int, int>> vrstniRed(n);
  for (int i = 0; i < n; ++i) vrstniRed[i] = {h[i], i};
  sort(vrstniRed.begin(), vrstniRed.end(), greater<pair<int, int>>());
  set<int> pobarvane;

  for (auto [hi, i] : vrstniRed)
  {
    int b = 0; // V b bomo s prižiganjem bitov označili, katerih barv ne smemo uporabiti.

    // Kakšne barve je naslednja višja stolpnica desno od i?
    auto it = pobarvane.upper_bound(i);
    if (it != pobarvane.end()) b |= 1 << barva[*it];

    // Kakšne barve je naslednja višja stolpnica levo od i?
    if (it != pobarvane.begin()) b |= 1 << barva[*--it];

    // Pripišimo stopnici i prvo prosto barvo.
    barva[i] = (b & 2) == 0 ? 1 : (b & 4) == 0 ? 2 : 3;
    pobarvane.emplace(i);
  }
}
```

In še rešitev, ki vse vrednosti $L[i]$ in $D[i]$ izračuna v $O(n)$ časa:

```
void Pobarvaj2(const vector<int> &h, vector<int> &barva)
{
  int n = h.size(); barva.resize(n);

  // Za vsako stolpnico poiščimo naslednjo višjo na levi in desni.
  vector<int> L(n), D(n);
  for (int i = 0; i < n; ++i) for (L[i] = i - 1; L[i] >= 0 && h[L[i]] < h[i]; L[i] = L[L[i]]) ;
  for (int i = n-1; i>=0; --i) for (D[i] = i+1; D[i]<n && h[D[i]]<h[i]; D[i]=D[D[i]]) ;

  // Uredimo stolpnice padajoče po višini.
  vector<pair<int, int>> vrstniRed(n);
  for (int i = 0; i < n; ++i) vrstniRed[i] = {h[i], i};
  sort(vrstniRed.begin(), vrstniRed.end(), greater<pair<int, int>>());

  for (auto [hi, i] : vrstniRed)
  {
    int b = 0; // V b bomo s prižiganjem bitov označili, katerih barv ne smemo uporabiti.

    // Kakšne barve je naslednja višja stolpnica levo od i?
    if (L[i] >= 0) b |= 1 << barva[L[i]];
    // Kakšne barve je naslednja višja stolpnica desno od i?
    if (D[i] < n) b |= 1 << barva[D[i]];

    // Pripišimo stopnici i prvo prosto barvo.
    barva[i] = (b & 2) == 0 ? 1 : (b & 4) == 0 ? 2 : 3;
  }
}
```

Zapišimo še primer implementacije te druge rešitve v pythonu.

```
def Pobarvaj2(visine):
  n = len(visine); barve = [−1] * n

  # Za vsako stolpnico poiščimo naslednjo višjo na levi in desni.
  L = [i − 1 for i in range(n)]; D = [i + 1 for i in range(n)]
  for i in range(n):
    while L[i] >= 0 and visine[L[i]] < visine[i]: L[i] = L[L[i]]
  for i in range(n − 1, −1, −1):
    while D[i] < n and visine[D[i]] < visine[i]: D[i] = D[D[i]]

  # Preglejmo stolpnice padajoče po višini.
  for (hi, i) in sorted(((visine[i], i) for i in range(n)), reverse = True):

    b = 0 # V b bomo s prižiganjem bitov označili, katerih barv ne smemo uporabiti.

    # Kakšne barve je naslednja višja stolpnica levo od i?
    if L[i] >= 0: b |= 1 << barve[L[i]]
    # Kakšne barve je naslednja višja stolpnica desno od i?
    if D[i] < n: b |= 1 << barve[D[i]]

    # Pripišimo stopnici i prvo prosto barvo.
    barve[i] = 1 if (b & 2) == 0 else 2 if (b & 4) == 0 else 3

  return barve
```

Oglejmo si zdaj še vprašanje, ki ga omenja opomba pod črto na koncu besedila naloge: če barvamo stolpnice od leve proti desni in vsaki dodelimo najmanjšo številko barve, ki je še nima nobena od z njo povezanih stolpnic, koliko barv porabimo v najslabšem primeru? Opazimo lahko, da konkretne višine stolpnic tu niso pomembne, pomembno je le, ali je neka stolpnica višja ali nižja od druge; če dani nabor stolpnic spremenimo tako, da najnižji pripišemo višino 1, drugi najnižji višino 2 in tako naprej, se pri barvanju ne bo nič spremenilo. Ker smo poleg tega rekli, da so stolpnice različno visoke, se lahko zato omejimo na primere, ko višine stolpnic tvorijo neko permutacijo množice $\{1, 2, \ldots, n\}$.

Ni težko napisati programa, ki pri danem n z rekurzijo pregleda vse permutacije množice $\{1, 2, \ldots, n\}$ in za vsako od njih izračuna število barv, če barvamo stolpnice s takšnimi višinami od leve proti desni. Če to naredimo, lahko opazimo, da je najmanjše število stolpnic, pri katerem se lahko zgodi, da potrebujemo b barv, enako $F(b+1)$, kjer so $F(\cdot)$ Fibonaccijeva števila: $F(0) = 0$, $F(1) = 1$, $F(t) = F(t-1) + F(t-2)$ za $t \geq 2$. Opazili bomo tudi, da pri $n = F(b+1)$ obstaja sicer več zaporedij n stolpnic, ki zahtevajo natanko b barv, vendar dajo vsi ti primeri enako zaporedje barv. Spodaj je nekaj primerov za majhne b:

$b = 1$, $n = 1$	$b = 2$, $n = 2$	$b = 3$, $n = 3$
višine: 1	višine: 1 2	višine: 2 1 3
barve: 1	barve: 1 2	barve: 1 2 3

$b = 4$, $n = 5$	$b = 5$, $n = 8$
višine: 4 3 1 2 5	višine: 7 6 4 5 2 1 3 8
barve: 1 2 1 3 4	barve: 1 2 1 3 1 2 4 5

$b = 6$, $n = 13$

višine:	12	11	9	10	7	6	8	4	3	1	2	5	13
barve:	1	2	1	3	1	2	4	1	2	1	3	5	6

Zaporedju barv, ki ga v teh primerih dobimo pri danem b, recimo C_b, zaporedju

višin pa H_b (pri čemer poudarimo, da je mogoče isto zaporedje barv dobiti tudi še pri nekaterih drugih zaporedjih višin). Opazimo lahko, da je mogoče ta zaporedja sestavljati na sistematičen način. Začnimo s $C_0 = H_0 = \langle\,\rangle$ (prazno zaporedje) in $C_1 = H_1 = \langle 1\rangle$, nato pa za $b \geq 2$ nadaljujmo po naslednjem pravilu: zaporedje C_b dobimo tako, da najprej vzamemo prvih $F(b) - 1$ členov zaporedja C_{b-1} (to so vsi členi razen zadnjega), nato prvih $F(b-1) - 1$ členov zaporedja C_{b-2} (tudi to so vsi razen zadnjega) in nato dodamo člena $b-1$ in b. Zaporedje H_b pa dobimo tako, da (1) najprej vzamemo prvih $F(b)-1$ členov zaporedja H_{b-1} (torej vse razen zadnjega) in jih povečamo za $F(b-1)$; (2) nato dodamo celotno zaporedje H_{b-2}; in (3) na koncu dodamo stolpnico višine $F(b+1)$ (kar je enako n).

Prepričajmo se zdaj, da to opažanje drži v splošnem, torej da za vsak b velja, da pri barvanju zaporedja stolpnic H_b od leve proti desni dobimo ravno zaporedje barv C_b. Dokazovali bomo z indukcijo po b. Za $b = 0$ in $b = 1$ takoj vidimo, da to drži; recimo zdaj, da trditev velja do vključno $b-1$, in razmislimo, kaj se zgodi pri b.

Ko barvamo začetni del zaporedja H_b, kjer so stolpnice enake kot v H_{b-1} (brez zadnjega člena), le dvignjene za $F(b-1)$, so torej tudi povezave med njimi enake kot pri H_{b-1}; zato dobijo tudi enake barve kot pri barvanju zaporedja H_{b-1}. Tako torej vidimo, da je prav, da se C_b v tem začetnem delu ujema z zaporedjem C_{b-1}.

Ko nato barvamo drugi del zaporedja H_b, kjer so stolpnice enake kot v H_{b-2}, je edina dodatna povezava, ki jo lahko nekatere od teh stolpnic zdaj imajo (in ki je ne bi imele, če bi barvali le H_{b-2} samo po sebi), ta, da so nekatere morda povezane z zadnjo stolpnico v začetnem delu — tista je namreč v njem najvišja, zato so stolpnice levo od nje prenizke, da bi bile lahko povezane s kakšno stolpnico desno od nje (v drugem delu). Tista zadnja stolpnica v prvem delu ima barvo $b-2$, torej nas morebitna povezava do nje lahko pri barvanju drugega dela začne ovirati šele takrat, ko bi sicer (če te stolpnice in povezav do nje ne bi bilo) neki stolpnici želeli dodeliti barvo $b-2$. Toda drugi del, ki ga zdaj barvamo, je enak zaporedju H_{b-2} in v tem zaporedju dobi barvo $b-2$ le zadnja stolpnica. Pri barvanju drugega dela torej šele v zadnji stolpnici drugega dela nastopi težava zaradi povezave z zadnjo stolpnico prvega dela (ta povezava gotovo obstaja, ker so vse stolpnice med njima nižje — zadnja stolpnica drugega dela je namreč najvišja v tem delu); in ta povezava nas prisili, da zadnji stolpnici drugega dela ne damo barve $b-2$, pač pa $b-1$. Tako torej vidimo, da je prav, da se C_b v tem drugem delu ujema z zaporedjem C_{b-2}, razen pri zadnji stolpnici drugega dela, ki je v C_b upravičeno dobila vrednost $b-1$.

Na koncu imamo še zadnjo, n-to stolpnico zaporedja H_b. Visoka je $n = F(b+1) = F(b) + F(b-1)$, torej je ravno za $F(b-1)$ višja od zadnje stolpnice zaporedja H_{b-1}. Zato vse povezave, ki bi jih v H_{b-1} imela zadnja stolpnica s tistimi levo od sebe, še zdaj obstajajo tudi v H_b med zadnjo stolpnico in med stolpnicami prvega dela (ki so ravno enake stolpnicam iz H_{b-1}, le dvignjene za $F(b-1)$). Te povezave nas pri barvanju zaporedja H_{b-1} prisilijo, da zadnji stolpnici damo barvo $b-1$ in ne kakšne nižje; zato nas tudi zdaj pri barvanju zaporedja H_b prisilijo, da zadnji stolpnici ne damo nobene od barv $1, \ldots, b-2$. Barve $b-1$ pa ji tudi ne moremo dati, saj takoj levo od nje stoji zadnja stolpnica drugega dela, ki je že sama barve $b-1$; zato zadnja stolpnica zaporedja H_b dobi barvo b, prav to pa je tudi zadnji člen zaporedja C_b. □

Zdaj torej vemo, da za vse n-je z območja $F(b+1) \leq n < F(b+2)$ vsekakor

obstaja kakšno tako zaporedje n stolpnic, pri katerem ob barvanju od leve proti desni porabimo b barv. Ali obstaja morda tudi kakšno krajše zaporedje (z manj kot $F(b+1)$ stolpnicami), kjer se tudi porabi b barv? Prepričajmo se, da se to ne more zgoditi; nakrajše zaporedje stolpnic, kjer naš postopek barvanja od leve proti desni porabi b barv, je dolgo $F(b+1)$.

Dokazovali bomo z indukcijo po b. Pri $b = 1$ trditev pravi, da je najkrajše zaporedje, pri katerem naš postopek porabi $b = 1$ barvo, dolgo $F(2) = 1$ stoplnico; pri takem zaporedju res porabi eno barvo, če pa bi imeli 0 stolpnic, bi porabil 0 barv, tako da trditev drži. Podobno lahko ročno preverimo $b = 2$, kjer trditev pravi, da je najkrajše zaporedje, pri katerem naš postopek porabi 2 barvi, dolgo $F(3) = 2$ stolpnici; in res pri kateremkoli primeru z dvema stolpnicama različne višine naš postopek porabi dve barvi, pri primerih s samo eno ali nobeno stolpnico pa manj kot dve barvi, tako da trditev drži tudi za $b = 2$.

Recimo zdaj, da trditev drži do vključno $b - 1$. Mislimo si najkrajše zaporedje stolpnic, pri katerem naš postopek porabi natanko b barv. Recimo, da je dolgo n stolpnic; oštevilčimo jih od leve proti desni s števili od 1 do n. Naš postopek barva stolpnice od leve proti desni in uporabi barvo b le, če je neka stolpnica na svoji levi povezana s stolpnicami vseh barv od 1 do $b - 1$. Tisti trenutek torej, ko neka stolpnica dobi barvo b, lahko zaključimo, da je bilo zdaj uporabljenih že vseh b barv; zato se to gotovo zgodi šele pri zadnji, n-ti stolpnici, saj bi lahko drugače stolpnice za prvo stolpnico barve b pobrisali in dobili krajši primer, ki bi še vedno zahteval b barv.

Zdaj torej vemo, da ima stolpnica n barvo b in da je na svoji levi povezana z vsaj eno stolpnico vsake barve od 1 do $b - 1$; vzemimo torej za vsako od teh barv po eno stolpnico, povezano s stolpnico n; teh $b-1$ stolpnic oštevilčimo od leve proti desni in naj bo x_i položaj i-te od njih v zaporedju n stolpnic, c_i pa njena barva. Ker smo jih oštevilčili od leve proti desni, velja seveda $x_1 < x_2 < \ldots < x_{b-1}$; barve $c_1, c_2, \ldots, c_{b-1}$ pa tvorijo neko permutacijo števil $1, 2, \ldots, b - 1$.

Ker je stolpnica x_1 barve c_1, to pomeni, da je naš postopek za stolpnice od 1 do x_1 uporabil vse barve od 1 do c_1; po induktivni predpostavki to pomeni, da je ta začetni del zaporedja dolg vsaj $F(c_1 + 1)$ stolpnic.

Vzemimo zdaj poljuben t z območja $1 < t < b$ in si oglejmo stolpnice od $x_{t-1}+1$ do x_t. Za vsako stolpnico u s tega območja velja, da je nižja od stolpnice x_{t-1} (ker je le-ta povezana s stolpnico n in zato višja od vseh vmesnih stolpnic, od $x_{t-1} + 1$ do $n - 1$); to pa pomeni, da je taka u sicer morda lahko povezana s stolpnico x_{t-1}, gotovo pa ne s kakšno od stolpnic levo od x_{t-1}, ker vmes stoji stolpnica x_{t-1} in prekinja povezavo (ker je višja od u).

To pomeni, da ko naš postopek barva stolpnice od $x_{t-1} + 1$ do x_t, ga lahko pri izbiri barv ovira izmed stolpnic levo od tega območja le stolpnica x_{t-1}, ki mu lahko prepreči uporabiti barvo c_{t-1}, drugače pa barvanje tega območja poteka tako, kot da bi se zaporedje s stolpnico $x_{t-1} + 1$ šele začelo.

Če je $c_{t-1} > c_t$, to pomeni, da naš postopek pri barvanju tega območja prej uporabi vse barve od 1 do c_t, preden ga lahko stolpnica x_{t-1} barve c_{t-1} sploh kaj ovira. Po induktivni predpostavki vemo, da mora biti to območje dolgo vsaj $F(c_t+1)$ stolpnic, saj sicer naš postopek na njem še ne bi mogel priti v položaj, ko bi moral uporabiti barvo c_t.

Če pa je $c_{t-1} < c_t$, se lahko zgodi, da naš postopek pri barvanju območja od $x_{t-1}+1$ do x_t prvič uporabi barvo c_t v takem trenutku, ko je dotlej na tem območju že uporabil barve $\{1, 2, \ldots, c_{t-1}-1, c_{t-1}+1, \ldots, c_t-1\}$, barve c_{t-1} pa zdaj ne more uporabiti zaradi povezave s stolpnico x_{t-1}. To pomeni, da če bi se zaporedje začelo šele pri $x_{t-1}-1$, bi na tem območju dobili barvanje s c_t-1 različnimi barvami; zato po induktivni predpostavki vemo, da mora biti to območje dolgo vsaj $F(c_t)$ stolpnic, saj sicer naš postopek na njem še ne bi mogel priti v položaj, ko bi čutil potrebo po uporabi barve c_t.

Tako torej vidimo, da je območje od $x_{t-1}+1$ do x_t gotovo dolgo vsaj $F(c_t)$, pa še to je mogoče le, če je $c_{t-1} < c_t$, sicer pa mora biti dolgo celó vsaj $F(c_t+1)$. Če to seštejemo po vseh t in prištejemo še $F(c_1+1)$ za stolpnice od 1 do x_1 ter na koncu še 1 za stolpnico n, vidimo, da mora biti naše zaporedje dolgo vsaj

$$\begin{aligned} n &\geq F(c_1+1) + 1 + \textstyle\sum_{t=2}^{b-1} F(c_t) \\ &= (F(c_1+1) - F(c_1)) + 1 + \textstyle\sum_{t=1}^{b-1} F(c_t) \\ &= F(c_1-1) + 1 + \textstyle\sum_{t=1}^{b-1} F(t), \end{aligned}$$

pri čemer smo pri zadnji enakosti izkoristili dejstvo, da tvorijo $c_1, \ldots, c_{b-1}$ neko permutacijo števil $1, 2, \ldots, b-1$. Meja, ki smo jo na koncu dobili, je torej tem nižja, čim manjši je c_1, najmanjša pa je zato pri $c_1 = 1$ (skupaj s pogojem $c_{t-1} < c_t$ pri vseh t to pomeni, da najnižjo mejo dobimo takrat, ko velja kar $c_t = t$ za vse $t = 1, \ldots, b-1$). Naša meja je potem naprej enaka $1 + \sum_{t=1}^{b-1} F(t)$, to pa je ravno $F(b+1)$.[10]

Dobili smo torej $n \geq F(b+1)$ — če je stolpnic manj kot $F(b+1)$, naš postopek prav gotovo ne bo porabil b (ali več) barv. □

Zdaj torej vemo, da če pri barvanju od leve proti desni porabimo b barv, je zaporedje dolgo vsaj $F(b+1)$ stolpnic: velja torej $n \geq F(b+1)$. Če upoštevamo še, da je $F(t) \approx \phi^t/\sqrt{5}$ za $\phi = (1+\sqrt{5})/2$, dobimo približno $n \geq \phi^{b+1}/\sqrt{5}$, torej $b \leq \log_\phi n + \log_\phi \sqrt{5} - 1$. Število barv, ki jih porabimo pri barvanju od leve proti desni, je torej v najslabšem primeru reda $O(\log n)$.

Naloge so sestavili: mastermind — Urban Duh; stonoge, sedežni red — Bor Grošelj Simić; žabe, breakout, barvanje stolpnic — Tomaž Hočevar; genialno — Boris Horvat; parkirišče — Filip Koprivec; planinarjenje — Mark Martinec; snežinke, slovar — Polona Novak; neprevidni poeti — Ella Potisek; dolgovi — Jakob Schrader; varnostno kopiranje — Jure Slak; seštevanje ulomkov — Mitja Trampuš; semafor, iskanje kvadrata — Borut in Peter Žnidar; luči, pijansko urejanje, L-sistem — Janez Brank. Primera pri nalogi Neprevidni poeti sta iz pesmi *Povodni mož* Franceta Prešerna in *V noč* Otona Župančiča (slednja je izšla v zbirki *Samogovori* leta 1908).

[10] Tudi o tem, da pri vsakem b velja $1 + \sum_{t=1}^{b-1} F(t) = F(b+1)$, se lahko prepričamo z indukcijo po b. Pri $b = 1$ dobimo $1 = F(1)$, kar je res; nato pa, če enakost velja za $b-1$, bomo pri b dobili $1 + \sum_{t=1}^{b-1} F(t) = (1 + \sum_{t=1}^{b-2} F(t)) + F(b-1) = F(b) + F(b-1) = F(b+1)$, kar smo tudi želeli dokazati.

A. Razbojniki

Ker je cestno omrežje pri tej nalogi povezano in ima eno povezavo manj, kot je točk, tvori drevo (neusmerjen povezan acikličen graf). Izberimo si v njem poljubno točko za koren, tako da bomo lahko govorili o starših in otrocih, prednikih in potomcih, kot je to običajno pri drevesih. Med poljubnima dvema točkama obstaja v drevesu natanko ena pot; dolžino poti med u in v označimo z d_{uv} (če sta u in v sosedi, je d_{uv} ravno dolžina povezave med njima). Poddrevo T_u tvorijo točka u in vsi njeni potomci.

Predpostavimo za začetek, da je to drevo lepo razvejeno (npr. kot binarno drevo) in zato ne pregloboko. V splošnem seveda ne bo nujno tako, vendar se bomo s tem ukvarjali kasneje.

Recimo, da nas zanima, koliko pogodb pokriva povezavo $(u : v)$, pri čemer predpostavimo, da je v otrok u-ja in ne obratno. Za vsako pogodbo (x, r) vemo, da izhaja iz neke točke x in velja v „radiju“ r od te točke; taka pogodba pokriva našo povezavo, če velja $\max\{d_{xu}, d_{xv}\} \leq r$, torej če dolžina poti od točke x do njej bolj oddaljene izmed točk u in v ne presega r. Pogodbe lahko razdelimo glede na to, katero od krajišč u in v je bližje točki x: če x leži v T_v, mu je točka v bližja kot u, sicer pa je ravno obratno.

(1) Razmislimo najprej o pogodbah, kjer x leži v T_v. Točk v T_v je preveč, da bi se lahko ukvarjali z vsako od njih (in ugotavljali, koliko pogodb iz nje doseže točko u ter tako pokrije našo povezavo); informacije o teh pogodbah je treba torej že ob dodajanju posamezne pogodbe prenesti gor po drevesu do v, da nam bodo pri roki, ko se bomo ukvarjali s povezavo $(u : v)$. Kaj točno pa potrebujemo pri v, da bomo lahko prešteli, koliko pogodb (x, r) z $x \in T_v$ pokriva povezavo $(u : v)$? Taka pogodba pokriva našo povezavo, če je $r \geq d_{xu} = d_{xv} + d_{uv}$, kar lahko zapišemo kot $r - d_{xv} \geq d_{uv}$. Tu je zdaj odvisna od pogodbe le leva stran neenačbe, desna pa ne. Koristno bi torej bilo, če bi imeli pri v v nekakšni podatkovni strukturi $\mathcal{S}_v$ zapisane vrednosti $r - d_{xv}$ za vse pogodbe, katerih x leži v T_v (to seveda pomeni, da ko se sklene nova pogodba s središčem v x, je treba iti po vseh x-ovih prednikih v in pri vsakem v strukturo $\mathcal{S}_v$ dodati vrednost $r - d_{xv}$). Matematično gledano si lahko $\mathcal{S}_v$ predstavljamo kot vrečo oz. multimnožico (ker je v njej lahko več enakih vrednosti):

$$\mathcal{S}_v := \{r_j - d_{x_j,v} : x_j \in T_v\},$$

kjer gre (x_j, r_j) po vseh doslej sklenjenih pogodbah (za $1 \leq j \leq m$, če imamo m pogodb). V praksi bi morale biti vrednosti v $\mathcal{S}_v$ organizirane tako, da bi se dalo hitro ugotoviti, koliko izmed njih je večjih ali enakih d_{uv}.[11] Primerna struktura za to je kakšno uravnoteženo drevo (npr. rdeče-črno), v katerem pa moramo pri vsakem vozlišču tudi vzdrževati podatek o tem, koliko elementov je v njegovem poddrevesu; tako se bo dalo v $O(\log m)$ časa prešteti, koliko vrednosti je $\geq d_{uv}$, prav tako pa tudi

[11] Pravzaprav za problem, ki je pred nami ta hip, sploh ne bi bilo treba hraniti cele $\mathcal{S}_v$; dovolj bi bilo le vzdrževati števec tega, koliko vrednosti v njej je $\geq d_{uv}$. Toda struktura $\mathcal{S}_v$, kakršno opisujemo tukaj, bo prišla zelo prav v nadaljevanju naše rešitve.

dodati novo vrednost v takšno drevo.[12] Definirajmo še $f_v(t)$ kot število elementov z vrednostjo $\geq t$ v $\mathcal{S}_v$; to je število pogodb (x, r), za katere x leži v T_v in je $r - d_{xv} \geq t$.

(2) Druga možnost so pogodbe, kjer x ne leži v T_v, zato mu je u bližja kot v. Taka pogodba pokriva našo povezavo natanko tedaj, če je $r \leq d_{xv}$. Naj bo p najglobljи skupni prednik točk x in u; pot od x do v gre potem najprej gor po drevesu od x do p in nato dol od p do v. Naš pogoj lahko torej zapišemo kot $r \leq d_{xp} + d_{pv}$ oz. $r - d_{xp} \leq d_{pv}$. To je neenačba prav take oblike kot v prejšnjem odstavku; število pogodb, ki ji ustrezajo, je torej $f_p(d_{pv})$, kar lahko računamo s pomočjo prav take drevesaste podatkovne strukture $\mathcal{S}_p$ kot prej za f_v.

Različne pogodbe z različnimi x imajo seveda lahko s točko u različne skupne prednike p (po enega za vsak nivo drevesa od u do korena). Vrednost $f_p(d_{pv})$ bi morali torej načeloma sešteti po vseh u-jevih prednikih p, vendar bomo pri tem nekatere pogodbe šteli po večkrat. Če je p prednik u-ja in ni koren drevesa, je p-jev starš tudi prednik u-ja; recimo temu staršu $\hat{p}$. V številu $f_{\hat{p}}(d_{\hat{p}v})$ so torej zajete vse pogodbe, ki izvirajo iz $T_{\hat{p}}$, tudi tiste, ki izvirajo iz T_p (saj vsaka točka, ki leži v T_p, leži tudi v $T_{\hat{p}}$). Koristno bi bilo torej vedeti, koliko elementov so v $\mathcal{S}_{\hat{p}}$ prispevale pogodbe, ki izvirajo iz T_p, da jih bomo znali odšteti; ti elementi so:

$$\mathcal{S}'_p := \{r_j - d_{x_j,\hat{p}} : x_j \in T_p\}, \text{pri čemer je } \hat{p} \text{ starš } p\text{-ja},$$

s $f'_p(t)$ pa označimo število tistih elementov iz $\mathcal{S}'_p$, ki so $\geq t$.[13]

Dosedanji razmislek lahko torej povzamemo takole: recimo, da je v otrok točke u in da pot od korena do v tvorijo p_0 (koren), $p_1, \ldots, p_{\lambda-2}$, $p_{\lambda-1} = u$ in $p_\lambda = v$; potem je število pogodb, ki pokrivajo povezavo $(u : v)$, enako

$$f_v(d_{uv}) + \sum_{i=0}^{\lambda-1} \big(f_{p_i}(d_{p_i,v}) - f'_{p_{i+1}}(d_{p_i,v})\big).$$

Videli smo, da če imamo naše vreče $\mathcal{S}_\bullet$ in $\mathcal{S}'_\bullet$ predstavljene s primernimi drevesastimi strukturami, nam izračun vsake vrednosti funkcij $f_\bullet$ in $f'_\bullet$ vzame $O(\log m)$ časa; in če je drevo globoko $O(\log n)$ nivojev, bomo za odgovor na posamezno poizvedbo porabili $O((\log m)(\log n))$ časa.

Težava je, da drevo ni nujno globoko le $O(\log n)$ nivojev; lahko je izrojeno v nekaj podobnega seznamu in ima do $O(n)$ nivojev. Naš dosedanji razmislek se je opiral na pojem poddreves: drevo T_w razdelimo na več delov tako, da prerežemo vse povezave okrog w-ja; T_w tako razpade na manjša drevesa T_z za vsakega w-jevega

[12] Neugodno pri tem je, da drevo iz C++ove standardne knjižnice (`std::set`) tega ne omogoča, pač pa to omogočajo nekatere nestandardne razširitve knjižnice, kot so npr. "policy-based data structures", ki so na voljo v `g++` (gl. npr. `https://codeforces.com/blog/entry/11080`). Če se hočemo nestandardnim stvarem izogniti in si napisati svojo implementacijo drevesa, bomo imeli s tem precej dela; v tem primeru si izberimo kakšno čim preprostejšo obliko drevesa, ki vendarle še daje zagotovila o uravnoteženosti, npr. *treap* (binarno iskalno drevo, v katerem vsako vozlišče dobi tudi naključno „prioriteto“ in po dodajanju premikamo novo vozlišče z rotacijami gor po drevesu tako dolgo, dokler je njegova prioriteta višja od prioritete njegovega starša; gl. npr. Wikipedijo *s. v.* Treap).

[13] Pozorni bralec bo tu pripomnil, da so v $\mathcal{S}'_p$ iste vrednosti kot v $\mathcal{S}_p$, le zmanjšane za $d_{p\hat{p}}$, in da je zato $f'_p(t) = f_p(t + d_{p\hat{p}})$. Zakaj smo torej definirali še $\mathcal{S}'_p$ kot posebno strukturo in se v funkciji f'_p sklicevali nanjo? Odgovor je, da se ta pripomba opira na dejstvo, da je $d_{x_j,\hat{p}} = d_{x_j,p} + d_{p\hat{p}}$, kar je tukaj res, kasneje v naši rešitvi pa ne bo tako.

otroka z, poleg tega pa imamo še točko w, ki ostane sama. Ključno pri tem je, da med različnimi T_z ni mogoče prehajati drugače kot skozi w.

Ta razmislek bi z minimalnimi popravki še vedno deloval tudi, če bi T_w delili na manjše dele tako, da ne bi prerezali povezav okrog w-ja, ampak okrog neke druge točke c, ležeče nižje dol v T_w. Za naše potrebe je koristno, če za c izberemo *centroid* drevesa T_w — to je tista točka, za katero velja, da ko prerežemo povezave okrog nje, ima vsak od nastalih delov kvečjemu pol toliko točk, kot jih je imelo prej celotno T_w. (Ali, z drugimi besedami: $|T_c|$ mora biti $\geq |T_w|/2$, za vsakega c-jevega otroka z pa mora veljati $|T_z| \leq |T_w|/2$.) Temu pogoju ustrezata ena ali dve točki v T_w; če sta dve, je vseeno, katero od njiju vzamemo za centroid.

Ko tako razdelimo T_w na več manjših delov, izvedimo isti postopek še rekurzivno na vsakem od njih. Tako smo prišli do znanega postopka *centroidne dekompozicije* drevesa; preden pa ga zapišemo, se dogovorimo, kaj točno nam mora ta postopek izračunati. Za vsako točko w naj nam $L[w]$ pove, na katerem nivoju dekompozicije[14] je w postal centroid; $P[w]$ naj pove, katera točka je bila centroid drevesa, ki mu je w pripadal na $(L[w]-1)$-vem nivoju dekompozicije; in $D[\ell, w]$ naj bo dolžina poti od w do centroida tistega drevesa, ki mu je w pripadal na ℓ-tem nivoju dekompozicije.

Te količine imajo precej vzporednic s tistimi, s katerimi smo se ukvarjali v začetnem delu naše rešitve: če bi drevo namesto pri centroidu vedno delili pri korenu, bi $L[w]$ povedal, na katerem nivoju drevesa (torej koliko korakov stran od korena) leži točka w; $P[w]$ bi bil starš točke w; in $D[\ell, w]$ bi bila razdalja od w do njenega prednika na nivoju ℓ. Toda zdaj pri centroidni dekompoziciji pozabimo na koren in na pojem staršev in otrok; vrnimo se nazaj k prvotnemu drevesu brez korena (*unrooted tree*), kakršno smo v resnici tudi dobili v vhodnih podatkih.

Na začetku inicializirajmo $L[w]$ na -1 za vse w; ko postane w na nekem nivoju dekompozicije centroid v svojem takratnem drevesu, vpišemo ta nivo v $L[w]$ in v mislih prerežemo vse povezave okrog w. Z drugimi besedami: povezava je prerezana natanko tedaj, ko vrednost $L[\cdot]$ pri vsaj enem od njenih krajišč ni več -1.

inicializacija: za vsako točko w naj bo $L[w] := -1$;

podprogram CentroidnaDekompozicija(točka s, nadrejeni centroid C):
(* *Vhod: točka s je soseda centroida C in pripada „njegovemu“ drevesu na $L[C]$-tem nivoju dekompozicije; naloga našega podprograma je rekurzivno obdelati tisti del drevesa, ki je dosegljiv iz s po povezavah, ki še niso prerezane — temu delu začasno recimo T_s.* *)
1 z iskanjem v globino iz s razišči celoten T_s; pri tem v mislih začasno štejmo s za koren drevesa T_s in za vsako točko w izračunajmo velikost $N[w]$ njenega poddrevesa, torej število točk, iz katerih pot do s vodi skozi w; pripravimo tudi seznam S obiskanih točk, v katerem so otroci vedno pred starši (*post-order traversal*);
2 $c :=$ prva točka v seznamu S, za katero je $N[c] \geq N[s]/2$; (* *centroid* *)
$P[c] := C$; **if** $C = -1$ **then** $L[c] := 0$ **else** $L[c] := L[C] + 1$;
$D[L[c], c] := 0$;

[14] Z drugimi besedami: pri kateri globini gnezdenja rekurzije; ker številka nivoja z globino gnezdenja narašča, to pomeni, to pomeni, da manjša vrednost $L[w]$ pomeni višji nivo dekompozicije (pri katerem imamo opravka z večjimi deli drevesa); nivo 0 je tisti, pri katerem gledamo celotno drevo.

3 z iskanjem v globino iz c obišči vse točke T_s, sproti računaj oddaljenosti točk od c in to oddaljenost pri vsaki obiskani točki w vpiši v $D[L[c], w]$;
4 za vsako c-jevo sosedo w, če je $L[w] = -1$:
CentroidnaDekompozicija(w, c);

glavni klic: CentroidnaDekompozicija(1, -1);

Pri glavnem klicu je načeloma vseeno, pri kateri točki s začnemo (zgoraj smo izbrali $s = 1$), saj ni še nobena povezava prerezana in bomo takrat v vsakem primeru raziskali celotno drevo. Zapišimo podrobneje še iskanje v globino v koraku 1:

podprogram IskanjeVGlobino1(točka w, seznam S):
$N[w] := 1$;
za vsako w-jevo sosedo z, če je $L[z] = -1$:
$N[w] := N[w] +$ IskanjeVGlobino1(z, S);
dodaj w na konec seznama S; vrni $N[w]$;

In v koraku 3:

podprogram IskanjeVGlobino3(točka w, nivo ℓ, oddaljenost δ od centroida):
$D[\ell, w] := \delta$;
za vsako w-jevo sosedo z, če je $L[z] = -1$:
IskanjeVGlobino3($z, \ell, \delta + d_{wz}$);

V podprogramu CentroidnaDekompozicija lahko torej koraka 1 in 3 zapišemo takole:

1 $S :=$ prazen seznam; IskanjeVGlobino1(s, S);

3 IskanjeVGlobino3(c, $L[c]$, 0);

Kakšna je časovna zahtevnost centroidne dekompozicije? Spomimo se, da smo centroid izbrali tako, da ko prerežemo povezave okrog njega, ima vsak od nastalih delov drevesa kvečjemu pol toliko točk kot drevo pred rezanjem. Ker smo začeli z drevesom z n točkami, gre torej dekompozicija največ $\log_2 n$ nivojev globoko. Če odmislimo vgnezdene rekurzivne klice, porabimo v podprogramu CentroidnaDekompozicija le $O(k)$ časa, če je k število točk, dosegljivih iz začne točke s; in vsota teh k po vseh klicih na posameznem nivoju dekompozicije je $O(n)$, saj se vsaka točka drevesa znajde le v enem od delov drevesa, na katere je le-to razpadlo na tem nivoju dekompozicije. Vsi klici na posameznem nivoju torej vzamejo $O(n)$ časa, nivojev pa je $O(\log n)$, torej je časovna zahtevnost celotnega postopka $O(n \log n)$.

Razmislimo zdaj o tem, kako si lahko s rezultati dekompozicije — prej omenjenimi tabelami D, P in L — pomagamo pri odgovarjanju na poizvedbe pri naši nalogi. Ker zdaj nimamo drevesa s korenom, bomo izraze, kot so „starši“, „otroci“ ipd. uporabljali malo drugače kot ponavadi: starš točke w bo točka $P[w]$, iz te definicije pa potem naravno sledijo tudi otroci, predniki in potomci. „Poddrevo“ T_w tvorijo vse tiste točke, ki jim je w prednik; to je ravno tisti del drevesa, pri katerem je w postal centroid.

Podobno kot pri našem prvotnem razmisleku (kjer smo v drevesu izbrali koren) se tudi zdaj odločimo v vsaki točki w vzdrževati podatkovni strukturi

$$\mathcal{S}_w := \{r_j - d_{x_j,w} : x_j \in T_w\} \text{ in } \mathcal{S}'_w := \{r_j - d_{x_j,P[w]} : x_j \in T_w\};$$

in podobno kot prej definirajmo funkciji, ki predstavljata poizvedbe po teh dveh strukturah: $f_w(t)$ oz. $f'_w(t)$ naj bo število takih elementov strukture $\mathcal{S}_w$ oz. $\mathcal{S}'_w$, ki so $\geq t$.[15]

Recimo zdaj spet, da nas zanima, koliko pogodb pokriva povezavo $(u : v)$. Med centroidno dekompozicijo je morala ena od točk u in v postati centroid prej kot druga; recimo brez izgube za splošnost, da je bila to u; potem takrat (preden je u postal centroid) povezava $(u : v)$ še ni bila prerezana, torej je bila v dosegljiva iz u, torej je $v \in T_u$; to pa tudi pomeni, da je v postal centroid v nekem vgnezdenem klicu znotraj klica, pri katerem je postal centroid u, zato je $L[v] > L[u]$. (Z drugimi besedami: če sta u in v sosedi v grafu, to pomeni, da je ena od njiju v poddrevesu druge in ima višjo vrednost $L[\cdot]$ kot druga.)

Zapišimo po vrsti v in njegove prednike: $p_\lambda, p_{\lambda-1}, \ldots, p_0$, kjer je $p_\lambda = v$ (za $\lambda = L[v]$) in $p_{\ell-1} = P[p_\ell]$ (za $0 \leq \ell < \lambda$); zaporedje se konča pri p_0, ki je centroid celotnega grafa; nekje v tem zaporedju je tudi $u = p_\mu$ za $\mu = L[u]$. Vsakemu predniku p_ℓ pripada poddrevo T_{p_ℓ}, kar bomo v nadaljevanju pisali krajše kar T_ℓ; ta drevesa tvorijo naraščajoče zaporedje $T_\lambda \subset T_{\lambda-1} \subset \ldots \subset T_0$, pri čemer je T_0 celoten graf. Pogodbe (x, r) lahko v mislih razdelimo na skupine glede na to, katero drevo T_ℓ je prvo v tem zaporedju (torej tako z največjim ℓ), ki vsebuje točko x.

(1) Če x leži v T_v: v tem drevesu leži tudi v (točka u pa ne), zato mora v njem obstajati tudi pot od x do v. To pot je mogoče potem podaljšati s povezavo med v in u. Če bi obstajala od x do u še kakšna druga pot, ki ne bi šla skozi v, bi to pomenilo, da obstaja v grafu cikel, kar pa je nemogoče. Od krajišč naše povezave $(u : v)$ je torej točki x bližje krajišče v, bolj oddaljeno pa u; taka pogodba torej pokriva našo povezavo, če je $r \geq d_{xu} = d_{xv} + d_{vu}$, torej $r - d_{xv} \geq d_{vu}$, takih pogodb pa je $f_v(d_{vu})$.

(2) Če x leži v T_ℓ, ne pa v $T_{\ell+1}$, in je $\ell > \mu$: v drevesu T_ℓ leži tudi točka v (točka u pa ne); ko smo med centroidno dekompozicijo razglasili točko p_ℓ za centroid drevesa T_ℓ in porezali povezave okrog nje, je T_ℓ razpadlo na več delov; eden od teh delov je $T_{\ell+1}$, ki vsebuje točko v; točka x pa je očitno v nekem drugem delu (ker ne leži v $T_{\ell+1}$). Pot od x do v gre torej skozi točko p_ℓ, nato pa jo je mogoče s povezavo $(u : v)$ podaljšati do u. Če bi obstajala od x do u še kakšna druga pot, ki ne bi šla skozi v, bi bil v grafu cikel, kar je spet protislovje. Točki x je torej krajišče v bližje kot u in taka pogodba (x, r) pokriva našo povezavo, če je $r \geq d_{xu} = d_{x,p_\ell} + d_{p_\ell,u}$, torej če $r - d_{x,p_\ell} \geq d_{p_\ell,u}$, takih pogodb pa je $f_{p_\ell}(d_{p_\ell,u}) - f'_{p_{\ell+1}}(d_{p_\ell,u})$. Pri tem lahko vrednost $d_{p_\ell,u}$ odčitamo iz $D[\mu, p_\ell]$, saj je točka $u = p_\mu$ prednik točke p_ℓ; lahko pa bi jo računali tudi kot $d_{p_\ell,u} = d_{p_\ell,v} + d_{vu} = D[\ell, v] + d_{vu}$.

(3) Če x leži v T_μ, ne pa v $T_{\mu+1}$: ko smo med centroidno dekompozicijo razglasili točko u za centroid drevesa T_μ in porezali povezave okrog nje, je T_μ razpadlo na več delov; eden od teh je $T_{\mu+1}$ (ki vsebuje tudi točko v), točka x pa očitno leži v nekem drugem delu. V prvotnem grafu je torej obstajala pot od x do u in od tam potem v enem koraku do v; in če bi obstajala še kakšna druga pot med x in v (ki

[15] Že v opombi 13 smo se pogovarjali o tem, zakaj bo koristno definirati $\mathcal{S}'$ ločeno od $\mathcal{S}$. Tukaj zdaj vidimo, zakaj je tako: pri centroidni kompoziciji točka w in njen starš $P[w]$ nista nujno soseda, zato o zvezi med $d_{x_j,w}$ in $d_{x_j,P[w]}$ v splošnem ne moremo reči ničesar koristnega, npr. da se razlikujeta za $d_{w,P[w]}$. To slednje namreč ne drži, če x_j leži na poti med w in $P[w]$ oz. še splošneje: če pot od x_j do $P[w]$ ne teče skozi w. Nekaterim $x_j \in T_w$ je lahko točka $P[w]$ celo bližja kot točka w.

ne bi šla medtem še skozi u), bi obstajal v grafu cikel, kar je nemogoče. Torej je točki x od krajišč naše povezave bližja u kot v. Zato taka pogodba pokriva našo povezavo, če velja $r \geq d_{xv} = d_{xu} + d_{uv}$, torej $r - d_{xu} \geq d_{uv}$. Takih pogodb je $f_u(d_{vu}) - f'_{p_{\mu+1}}(d_{vu})$.

(4) Če x leži v T_ℓ, ne pa v $T_{\ell+1}$, in je $\ell < \mu$: razmislek je podoben kot v prejšnjem odstavku. Ko je postala točka p_ℓ centroid drevesa T_ℓ, je le-to razpadlo na več delov, od katerih eden (namreč $T_{\ell+1}$) vsebuje tako točki u kot v, točka x pa je očitno v enem od preostalih delov. V prvotnem grafu je torej obstajala pot od x do p_ℓ in nato od tam do u in v; če bi obstajala od x do teh dveh točk še kakšna druga pot, ki ne bi šla skozi p_ℓ, bi obstajal v grafu cikel, kar je spet protislovje. Vemo torej, da je $d_{xu} = d_{x,p_\ell} + d_{p_\ell,u}$ in podobno za d_{xv}; tega, katera od d_{xu} in d_{xv} je večja, pa tu v splošnem ne moremo reči. Pogodba (x, r) potem pokriva povezavo $(u : v)$ natanko tedaj, če je $r \geq \max\{d_{xu}, d_{xv}\}$, torej $r - d_{x,p_\ell} \geq t_\ell$ za $t_\ell = \max\{d_{p_\ell,u}, d_{p_\ell,v}\} = \max\{D[\ell, u], D[\ell, v]\}$. Takih pogodb je $f_{p_\ell}(t_\ell) - f'_{p_{\ell+1}}(t_\ell)$. (Mimogrede, ta formula bi delovala tudi pri $\ell = \mu$, zato primera (3) niti ni treba obravnavati posebej. Pri $\ell = \mu$ namreč dobimo $t_\mu = \max\{d_{p_\mu,u}, d_{p_\mu,v}\} = \max\{d_{uu}, d_{uv}\} = d_{uv}$.)

Zdaj vemo dovolj, da lahko zapišemo naš postopek odgovarjanja na poizvedbe s psevdokodo:

funkcija POKRITOST(u, v, d_{uv}):
  if $L[u] > L[v]$ **then** zamenjaj u in v;
  $N := f_v(d_{uv})$; (* *Primer 1.* *)
  $w := v$;
  for $\ell := L[v] - 1$ **downto** 0:
    $z := P[w]$; (* *Zdaj velja* $z = p_\ell$ *in* $w = p_{\ell+1}$. *)
    if $\ell > \mu$ **then** $t := D[\mu, z]$ (* *Primer 2.* *)
    else $t := \max\{D[\ell, u], D[\ell, v]\}$; (* *Primera 3 in 4.* *)
    $N := N + f_z(t) - f'_w(t)$; $w := z$;
  return N;

Ne pozabimo, da se v vsakem klicu funkcij $f_\bullet$ in $f'_\bullet$ skriva poizvedba po eni od drevesastih struktur $\mathcal{S}_\bullet$ in $\mathcal{S}'_\bullet$; skupna časovna zahtevnost tega postopka je torej $O((\log n)(\log m))$, če imamo n točk in m pogodb.

Ko se pojavi v vhodnih podatkih nova pogodba (x, r), moramo primerno popraviti podatke v prednikih točke x. Zapišimo še ta postopek:

podprogram DODAJPOGODBO(x, r):
  dodaj element r v $\mathcal{S}_x$;
  $w := x$;
  for $\ell := L[x] - 1$ **downto** 0:
    $z := P[w]$; (* z *je prednik točke* x *na nivoju* ℓ *)
    dodaj element $r - D[\ell, x]$ v $\mathcal{S}_z$ in $\mathcal{S}'_w$;
    $w := z$;

Dodajanje v posamezno strukturo $\mathcal{S}_\bullet$ ali $\mathcal{S}'_\bullet$ vzame $O(\log m)$ časa, enako kot poizvedba, zato ima tudi dodajanje pogodbe časovno zahtevnost $O((\log n)(\log m))$. Pri naši nalogi je q dodajanj pogodb in izračunov pokritosti skupaj, torej tudi največ q pogodb; če upoštevamo še $O(n \log n)$ časa za centroidno dekompozicijo, je skupna časovna zahtevnost naše rešitve $O((n + q \log q) \log n)$.

V praksi je dobro pri implementaciji opisane rešitve paziti še na nekaj podrobnosti. Vrednosti, manjših od 0, v strukture $\mathcal{S}_\bullet$ in $\mathcal{S}'_\bullet$ ni treba dodajati, saj te pomenijo, da vpliv pogodbe ne seže niti do tistega prednika (in tudi pri poizvedbah, torej pri klicih funkcij $f_\bullet(t)$ in $f'_\bullet(t)$, bo parameter t vedno ≥ 0). (Pozor: to velja le za vrednosti, strogo manjše od 0, ne pa tudi za tiste, ki so enake 0, kajti dolžine povezav v vhodnih podatkih so lahko tudi 0, zato lahko celo pogodba z radijem $r = 0$ vpliva na nekaj točk.) Pri računanju dolžin poti, npr. v tabeli D, pazimo še na to, da so poti lahko daljše od 2^{32} (imamo do 10^5 povezav, vsaka pa je dolga do 10^9), torej bo lažje, če uporabimo kak 64-bitni celoštevilski tip.

B. Kombinacijske ključavnice

V besedilu naloge je vzorec razlik (med Aličino in Bobovo številko) predstavljen z zaporedjem pik in enačajev, v naši rešitvi pa bomo zaradi berljivosti raje uporabljali ničle in enice. Vzorec razlik je torej pravzaprav niz n bitov. Posamezna poteza se sestoji iz tega, da se eden od bitov v vzorcu obrne (iz 0 v 1 ali obratno). Taka poteza je načeloma vedno možna na kateremkoli od n bitov v vzorcu,[16] razen pač v tem smislu, da se moramo izogibati praznovernim vzorcem ter vzorcem, ki so se med igro pojavili že kdaj prej. Glede praznovernih vzorcev bomo za začetek predpostavili, da jih ni, in se bomo kasneje ukvarjali s tem, kako jih upoštevati pri reševanju naloge.

Vzorce si lahko predstavljamo kot točke grafa, možne premike med njimi pa kot (neusmerjene) povezave; opazimo lahko, da ni ta graf nič drugega kot n-razsežna hiperkocka. Ta graf je dvodelen: vzorce lahko v mislih razdelimo na „sode“ in „lihe“ glede na to, ali vsebujejo sodo ali liho mnogo enic; in ker se pri premiku spremeni število enic natanko za 1, se vzorec pri tem spremeni iz sodega v lihega ali obratno. Vsaka povezava grafa torej povezuje eno sodo in eno liho točko. Vendar pa naslednjih nekaj korakov našega razmisleka ne zahteva, da je graf ravno hiperkocka, pač pa veljajo za poljuben povezan neusmerjen graf.

Za vsakega od obeh igralcev velja, da se nobeni dve njegovi potezi ne končata v isti točki (saj bi s tem prekršil pravilo, da se med igro noben vzorec ne sme ponoviti); pa tudi začneta se ne v isti točki (saj bi to pomenilo, da je drugi igralec dvakrat naredil potezo v to točko in s tem prekršil omenjeno pravilo; ali pa bi ga prekršil s tem, da bi naredil potezo v začetni vzorec). Vse poteze enega igralca torej tvorijo *ujemanje* — množico povezav, od katerih nobeni dve nimata skupnega krajišča.

Ker se igra začne z Aličino potezo in ker vlečeta potem igralca poteze izmenično, naredita bodisi oba enako število potez (če se igra konča z Bobovo zmago, ker bi morala Alica narediti naslednjo potezo, pa je ne more) bodisi naredi Alica eno potezo več kot Bob (če se igra konča z njeno zmago, ker bi moral Bob narediti naslednjo potezo, pa je ne more). Z drugimi besedami: če pogledamo ob koncu igre množico Aličinih potez in množico Bobovih potez, bomo vedeli, da je zmagala Alica, če je njena množica večja, sicer pa je zmagal Bob.

[16] Če je bil bit tam prej 0, to pomeni, da sta se številki obeh ključavnic na tem mestu razlikovali, in igralec, ki je trenutno na potezi, lahko spremeni tisto številko svoje ključavnice tako, da bo potem enaka istoležni številki pri ključavnici drugega igralca; tako se bo v vzorcu razlik na tem mestu ničla spremenila v enico. — Po drugi strani, če bi radi v vzorcu razlik spremenili enico v ničlo, to pomeni, da se ključavnici na tistem mestu trenutno ujemata (imata enaki številki), in igralec, ki je trenutno na potezi, lahko tisto številko pri svoji ključavnici zamenja s katerokoli drugo (izmed številk od 0 do 9), pa se bo potem njegova ključavnica na tem mestu gotovo razlikovala od druge in se bo zato ustrezni bit v vzorcu razlik spremenil v ničlo.

Malo prej smo videli, da poteze posameznega igralca tvorijo ujemanje v grafu. Ker igrata oba igralca optimalno, si želita verjetno uporabiti čim večje ujemanje; za Alico je torej zanimivo največje táko ujemanje A, ki vsebuje tudi začetni vzorec (recimo mu z), za Boba pa največje táko ujemanje B, ki ne vsebuje začetnega vzorca. (S tem, da ujemanje vsebuje z, hočemo reči, da vsebuje neko povezavo, ki ji je z eno od krajišč.) Ti dve množici nam med drugim povesta, da Alica nikakor (ne glede na to, kako igra Bob) ne more narediti več kot $|A|$ potez, Bob pa (ne glede na to, kako igra Alica) ne več kot $|B|$ potez.

Hitro se lahko prepričamo, da je v A lahko bodisi enako število povezav kot v B ali pa ena več. (1) Ena možnost je, da je A prazna; to je mogoče le, če z ni krajišče nobene povezave; toda ker imamo opravka s povezanim grafom, to pomeni, da v njem sploh ni nobene druge točke in torej tudi nobene povezave, zato je tudi B prazna, torej imata A in B enako število elementov. (2) Če pa A ni prazna, mora vsebovati natanko eno povezavo s krajiščem z. Če jo pobrišemo, dobimo neko ujemanje, ki ne vsebuje z; največje tako ujemanje pa je B, torej ima B vsaj $|A|-1$ elementov. (3) Ali bi se lahko zgodilo, da bi imel B več elementov kot A? Tedaj B ni prazen; ker je graf povezan, mora biti torej tudi z krajišče kakšne povezave, recimo (z,u); (3.1) če v B ni nobene povezave s krajiščem v u, je $B \cup \{(z,u)\}$ ujemanje, ki vsebuje z, torej ne more imeti več elementov kot A, torej tudi B ne more imeti več elementov kot A. (3.2) Če pa je v B neka povezava s krajiščem v u, recimo (u,v), potem je $B - \{(u,v)\} \cup \{(z,u)\}$ ujemanje, ki vsebuje z, torej ne more imeti več elementov kot A; ima pa jih toliko kot B, torej tudi B ne more imeti več elementov kot A. □

Sumimo torej, da bo v primeru, ko je $|A| = |B| + 1$, zmagala Alica, če pa je $|A| = |B|$, bo zmagal Bob. Prepričajmo se, da to drži; ob tem bomo tudi videli, kakšna je pri enem in drugem zmagovalna strategija.

(1) Če je A prazna, je (kot smo videli že malo prej) graf sestavljen le iz točke z in Alica ne more narediti niti prve poteze; torej zmaga Bob, prav to pa smo hoteli dokazati.

(2) Če je $|A| = |B| = k$ za neki $k > 0$: po vsaki Aličini potezi, recimo v točko w, lahko Bob pogleda, ali je w krajišče kakšne povezave v B; če je, se premakne po njej. Če se to nadaljuje dovolj dolgo, bo po k Aličinih in k Bobovih potezah Alica gotovo v položaju, ko ne bo mogla narediti naslednje poteze, saj vemo, da ne more narediti več kot $|A| = k$ potez. Kaj pa, če Alica v nekem trenutku naredi potezo v neko tako točko w, ki ni krajišče nobene povezave v B? Recimo, da to prvič stori v svoji ℓ-ti potezi. Dotedanje poteze tvorijo sprehod oblike $z = z_1, u_1, z_2, u_2, z_3, \ldots\ u_{\ell-1}, z_\ell, u_\ell = w$; pri tem so povezave $(u_i, z_{i+1}) \in B$. Če teh $\ell - 1$ povezav pobrišemo iz B in namesto njih vanjo dodamo ℓ povezav (z_i, u_i), je dobljena množica tudi ujemanje, ima pa $k+1$ elementov in vsebuje tudi točko z. To pa je v protislovju s predpostavko, da je največje tako ujemanje, ki vsebuje z, množica A, ki ima le k elementov. Torej Alica ne more narediti poteze v kakšno točko, ki bi bila zunaj B; torej bo Bob zmagal.

(3) Ostane še možnost, da je $|B| = k > 0$ in $|A| = k + 1$. Začetna točka z je gotovo krajišče neke povezave iz A, recimo (z,v); Alica lahko v prvi potezi uporabi to povezavo in naredi korak v v. Po vsaki Bobovi potezi, recimo v točko u, lahko potem Alica pogleda, če je u krajišče kakšne povezave v A; če je, se Alica premakne po njej. Alica ima v A dovolj povezav, da lahko tako reagira na k Bobovih potez,

nato pa Bob ne bo mogel narediti naslednje poteze, saj ne more narediti več kot $|B| = k$ potez. Kaj pa, če naredi Bob v nekem trenutku potezo v tako točko w, ki ni krajišče nobene povezave v A? Recimo, da se to prvič zgodi v njegovi ℓ-ti potezi. Dotedanje poteze tvorijo sprehod oblike $z = z_0, v = u_0, z_1, u_1, \ldots, u_{\ell-1}, z_\ell = w$, pri čemer so povezave $(z_i, u_i) \in A$ za $i = 0, \ldots, \ell - 1$. Če teh ℓ povezav pobrišemo iz A in namesto njih vanjo dodamo ℓ povezav (u_i, z_{i+1}) za $i = 0, \ldots, \ell - 1$, je dobljena množica tudi ujemanje, ki ima enako število elementov kot A (torej $k + 1$), vendar pa ne vsebuje točke z. To pa je v protislovju s predpostavko, da je največje tako ujemanje, ki ne vsebuje točke z, množica B, ki ima le k elementov. Torej Bob ne more narediti poteze v kakšno tako točko, ki bi bila zunaj A; torej bo Alica zmagala. □

Naš dosedanji razmislek je, kot smo že rekli, veljal za poljuben povezan neusmerjen graf; v naslednjem koraku pa nam bo prišlo prav dejstvo, da je pri tej nalogi naš graf dvodelen (ker je hiperkocka). Poiskati si namreč želimo B, torej največje táko ujemanje, ki ne vsebuje točke z. Ker je naš graf dvodelen, lahko problem maksimalnega ujemanja prevedemo na problem maksimalnega pretoka: dodajmo izvor s in ponor t, za vsako levo točko u dodajmo povezavo $s \to u$, za vsako desno točko v dodajmo povezavo $v \to t$; povezave med levimi in desnimi točkami v mislih usmerimo, da bodo kazale z leve na desno; vsem povezavam, tako novim kot starim, določimo kapaciteto 1; vsakemu pretoku od s do t v tako dopolnjenem grafu ustreza neko ujemanje v prvotnem grafu in obratno. Maksimalni pretok lahko poiščemo na primer s Ford-Fulkersonovim algoritmom, ki je dobro znan in se vanj tu zato ne bomo poglabljali.[17] Pri tem pazimo le na to, da točke z ne bomo uporabljali — začasno se delajmo, kot da je v grafu ni.

Kar pa se tiče A, lahko po tistem, ko najdemo ujemanje B (in njemu pripadajoč pretok po grafu), točko z v mislih dodamo nazaj v graf in nadaljujemo z Ford-Fulkersonovim algoritmom. Če se pretok zdaj še kaj izboljša, se je moral na račun točke z in je torej moralo nastati ujemanje, ki vsebuje tudi to točko; to je potem naš A in zanj torej vemo, da je večji od B (in da bo zato zmagala Alica). Če pa se pretok pri tem ne izboljša, potem vemo, da ne obstaja nobeno tako ujemanje A, ki bi vsebovalo z in bilo večje od B (in zato bo zmagal Bob).[18]

Na začetku naše dosedanje rešitve smo predpostavili, da praznovernih vzorcev ni. Kaj pa za nas pomeni, če se praznoverni vzorci vendarle pojavijo? Ker se pri vlečenju potez ne smemo premakniti vanje, je učinek tak, kot če bi tiste točke pobrisali iz grafa; zato pa se lahko zgodi, da graf ni več povezan, pač pa razpade na več komponent. Načeloma bi torej lahko najprej z iskanjem v širino ugotovili, katere točke so sploh dosegljive iz z in torej pripadajo isti povezani komponenti kot z, potem pa bi uporabili našo dosedanjo rešitev samo na tej povezani komponenti, ne pa na celem grafu. Toda če tega ne naredimo in iščemo maksimalni pretok kar

[17] Še ena možnost je, da uporabimo Hopcroft-Karpov algoritem za iskanje maksimalnega ujemanja v dvodelnih grafih, ki je načeloma učinkovitejši, vendar so pri naši nalogi grafi majhni (graf ima 2^n točk in vsaka točka ima n povezav, pri čemer je n (dolžina kombinacij na ključavnici) največ 10; maksimalni pretok pa je 2^{n-1}) in je že Ford-Fulkersonov algoritem čisto dovolj dober.

[18] Če bi hoteli v tem primeru najti kakšen konkreten A (maksimalne velikosti), bi si lahko preprosto izbrali poljubno povezavo s krajiščem v z, recimo (z, u); pogledali, katera povezava s krajiščem u je prisotna v B — recimo, da je to (u, v); in nato vzeli $A = B - \{(u, v)\} \cup \{(z, u)\}$. Toda konkretnega A takrat v resnici ne potrebujemo, saj naloga zahteva le, da ugotovimo, kdo bo zmagal, za to pa že vemo, da bo Bob.

po celem grafu, bo učinek tak, da bomo na z-jevi komponenti našli B, na ostalih komponentah pa pač neka njihova maksimalna ujemanja; in ko bomo potem v graf dodali točko z in preverjali, ali se pretok še kaj poveča (da bi videli, ali je A večji od B), se zaradi tega ne bo v tistih ostalih komponentah nič spremenilo, le v z-jevi komponenti se lahko pretok kaj poveča. Ker naloga sprašuje le, kateri igralec zmaga, bomo torej še vseeno dobili pravilni odgovor, četudi poganjamo naš postopek na celotnem grafu in ne le ne na z-jevi komponenti. Zapišimo zdaj našo rešitev še s psevdokodo:[19]

```
G naj bo n-dimenzionalna hiperkocka;
tiste točke G-ja, ki se po parnosti ujemajo z z,
  razglasi za leve, ostale za desne;
pobriši iz G vse praznoverne vzorce in še začetni vzorec z;
usmeri povezave tako, da bodo kazale od levih točk v desne;
dodaj v G izvor s in povezave od s do vseh levih točk;
dodaj v G ponor t in povezave od vseh desnih točk do t;
vse povezave naj imajo kapaciteto 1;
poišči (npr. s Ford-Fulkersonovim algoritmom) maksimalni pretok
  od s do t, recimo f;
(* Potem vemo, da je |B| = f. *)
dodaj nazaj v G točko z in njene povezave (ki jih tudi usmeri,
  da bodo kazale iz z, in jim pripiši kapaciteto 1);
nadaljuj s Ford-Fulkersonovim algoritmom; naj bo f' novi maksimalni pretok;
(* Zdaj vemo, da je |A| = f'. *)
če je f' > f, zmaga Alica, sicer pa Bob;
```

C. Ozvezdja

Glavni izziv pri tej nalogi je, kako ne porabiti preveč časa za računanje razdalj med ozvezdji. V nalogi je razdalja med ozvezdjema definirana kot povprečje razdalj med vsemi pari zvezd v njiju; za naš namen pa je še bolj kot povprečje koristna vsota teh razdalj:

$$s(A, B) := \sum_{a \in A} \sum_{b \in B} ||a - b||^2.$$

Razdalja med ozvezdjema je potem $d(A, B) = s(A, B)/(|A| \cdot |B|)$. Lepo pri takšnih vsotah je, da če združimo dve ozvezdji, recimo A_1 in A_2, se potem tudi njune vsote do ostalih ozvezdij le seštejejo:

$$s(A_1 \cup A_2, B) = s(A_1, B) + s(A_2, B).$$

[19] Za konec še zgodovinska opomba: igra, s katero se ukvarjamo pri tej nalogi, je različica igre „zemljepis“, kjer igralca izmenično imenujeta zemljepisna imena, pri čemer se mora vsako ime začeti na tisto črko, na katero se prejšnje konča. To lahko posplošimo tako, da igralca izmenično premikata žeton po usmerjenem grafu, in v tej obliki je vprašanje, kateri igralec bo zmagal (če oba igrata optimalno), zelo težko (PSPACE-polno; T. J. Schafer, “On the complexity of some two-person perfect-information games”, *J. of Computer and System Sciences*, 16(2):185–225 (April 1978)); za neusmerjene grafe pa ga lahko, kot smo videli tudi pri naši rešitvi, rešimo v polinomskem času z maksimalnimi ujemanji (A. S. Fraenkel, E. R. Scheinerman, D. Ullman, “Undirected edge geography”, *Theoretical Computer Science*, 112(2):371–81 (10 May 1993), na str. 372–3). Če je graf dvodelen (kot tudi pri naši nalogi), je iskanje maksimalnih ujemanj preprostejše kot za poljubne neusmerjene grafe.

Iz $s(A_1 \cup A_2, B)$ pa lahko potem hitro izračunamo tudi $d(A_1 \cup A_2, B)$; tako bomo torej lahko dobili razdalje med združenim ozvezdjem in ostalimi, ne da bi nam bilo treba še enkrat računati dvojne vsote oblike $\sum_{a \in A_1 \cup A_2} \sum_{b \in B} ||a - b||^2$.

Da bomo lahko izbrali, kateri dve ozvezdji združiti v naslednjem koraku, je koristno razdalje med vsemi pari ozvezdij hraniti v kopici (poleg razdalje naj bo seveda še starost obeh ozvezdij, saj naloga pravi, da če ima več parov enako razdaljo, izberemo med njimi tistega z najstarejšim ozvezdjem).

Zapišimo zdaj našo rešitev s psevdokodo. Med delom bomo podatke o ozvezdjih vzdrževali v tabelah: ozvezdje i vsebuje $V[i]$ zvezd in je nastalo ob času $A[i]$, vsota razdalj med zvezdami ozvezdij i in j pa je $S[i, j]$.

(* *Na začetku je vsaka zvezda ozvezdje zase.* *)
for $i := 1$ **to** n **do** $V[i] := 1$, $A[i] := i$;
$H :=$ prazna kopica;
for $i := 1$ **to** n **do for** $j := 1$ **to** n:
  $S[i, j] := (x_i - x_j)^2 + (y_i - y_j)^2$;
  dodaj v H element $(S[i, j], A[i], A[j], i, j)$;

$a := n + 1$; (* *šteje čas nastanka novih ozvezdij* *)
while H ni prazna:
  naj bo (s, a_1, a_2, i, j) najmanjši element kopice H; pobriši ga iz H;

  (* *Morda je ta element kopice zastarel in smo eno od ozvezdij i in j po tistem, ko smo ta element dodali v kopico, že združili z nekim tretjim ozvezdjem.* *)
  if $a_1 \neq A[i]$ **or** $a_2 \neq A[j]$ **then continue;**

  (* *Ozvezdju i pridružimo ozvezdje j, slednje pa pri tem preneha obstajati.* *)
  $A[i] := a$; $A[j] := -1$; $a := a + 1$;
  $V[i] := V[i] + V[j]$; $V[j] := -1$; izpiši $V[i]$;

  (* *Dodajmo v kopico razdalje med novim ozvezdjem in vsemi ostalimi.* *)
  for $k := 1$ **to** n **do if** $V[k] > 0$ **then**
    $S[i, k] := S[i, k] + S[j, k]$;
    dodaj v H element $(S[i, k]/(V[i] \cdot V[k]), A[k], A[i], k, i)$;

Ko združimo dve ozvezdji i in j v eno, uporabimo številko enega od njiju (v gornji rešitvi je to i) za združeno ozvezdje, druga številka (pri nas j) pa odtlej ni več v rabi (kar prepoznamo po tem, da sta njeni $A[j]$ in $V[j]$ zdaj negativni). Potem gremo v zanki po vseh ostalih ozvezdjih, računamo razdalje od novega ozvezdja do ostalih in jih dodajamo v kopico. V kopici hranimo elemente (d, a_1, a_2, i, j), kjer sta i in j številki ozvezdij, d je razdalja med njima, a_1 oz. a_2 pa sta časa nastanka ozvezdja i oz. j, pri čemer je $a_1 < a_2$. Leksikografsko manjši elementi torej predstavljajo pare ozvezdij, ki jih je treba združiti prej. Paziti moramo še na to, da ko združimo ozvezdji i in j, so za vsakega od njiju morda v kopici še vedno tudi elementi, ki predstavljajo možnost združitve kakšnega od njiju s kakšnim tretjim ozvezdjem; ti elementi so zdaj neveljavni in ko jih bomo kasneje dobili iz kopice, jih moramo ignorirati. Prepoznamo jih po tem, da čas nastanka vsaj enega od ozvezdij, ki ju element omenja, ni več tak, kakršen je bil, ko smo element dodali v kopico.

Še opomba glede razdalj med ozvezdji: to so v splošnem ne-cela števila, ulomki oblike $d(A, B) = s(A, B)/(|A| \cdot |B|)$; koristno je torej ločeno hraniti števec in ime-

novalec, potem pa za primerjavo dveh ulomkov namesto pogoja „$s_1/t_1 > s_2/t_2$“ uporabiti „$s_1 t_2 > t_1 s_2$“. Tako se lahko izognemo računanju z ne-celimi števili in morebitnim zaokrožitvenim napakam pri njem. Pazimo pa na to, da uporabimo 64-bitne celoštevilske tipe, saj so vsote $s(A, B)$ lahko večje od 2^{32}.

V kopico smo na začetku dodali $O(n^2)$ elementov; potem smo izvedli $n - 1$ združitev ozvezdij in pri vsakem dodali v kopico še $O(n)$ elementov; vsak element sčasoma tudi vzamemo iz kopice; to je skupaj $O(n^2)$ operacij na kopici, torej je časovna zahtevnost naše rešitve $O(n^2 \log n)$, prostorska pa $O(n^2)$. Teoretično bi se dalo časovno zahtevnost še izboljšati, če bi namesto običajne dvojiške kopice uporabili npr. Fibonaccijevo, vendar za naše tekmovanje to ni potrebno.

D. Razgozdovanje

Drevo v naši nalogi je seveda tudi drevo v smislu teorije grafov, torej acikličen povezan graf. Točke grafa nam bodo predstavljale člene (segmente) drevesa (torej deblo in veje), povezava $u \to v$ pa naj bo prisotna v primerih, kjer se člen u na koncu razveji in se ena od tam nastalih vej začne z v. Koren drevesa je točka, ki predstavlja deblo. Kot imajo v drevesu iz naloge posamezni segmenti vsak svojo maso, ima zdaj tudi vsaka točka našega grafa prav takšno maso. Poddrevo T_u točke u tvorijo u in vsi njeni potomci (torej točke, za katere pot od korena do njih teče skozi u). Maso točke u označimo z m_u, maso celotnega poddrevesa T_u pa z $m(T_u)$.

(1) Če je masa celotnega poddrevesa T_u manjša ali enaka naši nosilnosti w, potem ni nobene koristi od tega, da bi kjerkoli v tem poddrevesu izvedli kakšen rez, ki bi odrezal kaj manj kot celotno T_u — kajti če tak rez prestavimo tako, da bo odrezal celotno T_u, bo še vedno veljaven (ker bo odrezani del, to je zdaj celotno T_u, še vedno težak največ w), za kasnejše reze više gor v drevesu pa bo to kvečjemu še bolj ugodno (ker bodo tisti kasneje odrezani deli drevesa zaradi tega še malo lažji).

(2) Kaj pa, če je celotno poddrevo T_u vendarle težje od w, ne drži pa to za poddrevo nobenega od u-jevih otrok? Nekje v T_u bo torej treba izvesti kak rez, toda prejšnji odstavek nam pove, da ni smiselno odrezati nobenega u-jevega otroka manj kot v celoti. Recimo, da ima u otroke $v_1, \ldots, v_k$. Naj bo $s = \sum_{i=1}^{k} m(T_{v_i})$ skupna masa vseh teh otrok in njihovih poddreves.

(2.1) Če je $s \le w$, lahko izvedemo rez na členu u tako, da odreže vse otroke in še $w - s$ mase od samega člena u; ostane nam še $m_u - (w - s)$ mase člena u in če je to še vedno težje od w, lahko režemo od konca kose, težke po w, dokler ne ostane manj kot w mase; število rezov je torej $1 + \lfloor (m_u - w + s)/w \rfloor = \lfloor (m_u + s)/w \rfloor$, od bivšega poddrevesa T_u pa je ostal le člen u s težo $(m_u + s) \bmod w$.

(2.2) Če pa je $s > w$, to pomeni, da ne moremo odrezati vseh otrok naenkrat; po drugi strani smo že prej videli, da se posameznih otrok (ker njihova poddrevesa niso težja od w) ne splača rezati drugače kot v celoti; odrezati moramo torej nekaj otrok v celoti — toliko, da potem vsota ostalih ne bo več večja od w. Smiselno je seveda rezati otroke po padajoči teži poddreves; tako bo treba najmanj rezov, preden bo skupna teža padla pod (ali na) w. Ko se zgodi to slednje, nadaljujemo po enakem razmisleku kot v odstavku (2.1).

Zapišimo našo rešitev s psevdokodo:

podprogram ObdelajDrevo(): (* *Prebere in obdela naslednje poddrevo in vrne število izvedenih rezov ter težo preostanka.*)

```
preberi maso m naslednjega člena in število njegovih poddreves k;
(* Preberimo in obdelajmo poddrevesa in jih porežimo toliko, da bo vsako od
   njih težko manj kot w. Mase teh preostankov shranjujmo v seznam L
   in njihovo vsoto seštevajmo v s, skupno število rezov pa v r. *)
L := prazen seznam; s := 0; r := 0;
for i := 1 to k:
  (r', m') := ObdelajPoddrevo();
  r := r + r'; s := s + m';
  if m' > 0 then dodaj m' v seznam L;
if s > w: (* primer 2.2 *)
  uredi seznam L padajoče;
  za vsak element m' seznama L, padajoče po masi:
    r := r + 1; s := s − m'; (* odrežimo to poddrevo *)
    if s ≤ w then break;
(* Ostala poddrevesa lahko odrežemo v celoti — primera 1 in 2.1. *)
m := m + s; return (r + ⌊m/w⌋, m mod w)
```

Glavni klic tega podprogram torej obdela celotno drevo in dobi število rezov r ter maso m tega, kar je ostalo od debla; če je $m > 0$, moramo torej kot rezultat izpisati $r + 1$ (iz drevesa je nastalo $r + 1$ kosov — najprej po en odrezan kos pri vsakem rezu, na koncu pa še tisto, kar je ostalo od debla), sicer pa le r.

Če ima celotno drevo n segmentov, je časovna zahtevnost te rešitve v najslabšem primeru $O(n \log n)$ zaradi urejanja seznama L. To bi se dalo načeloma izboljšati na $O(n)$: ko imamo pred seboj seznam L, poiščimo v njem mediano, kar lahko storimo v $O(n)$ časa z znanim postopkom hitrega izbiranja (*quickselect*). Poglejmo, ali bi s padel na $\leq w$, če bi odrezali iz L vsa poddrevesa, težja od mediane; če ne bi, jih odrežimo in imamo v naslednjem koraku pred seboj pol krajši L (obdržimo le elemente, lažje od mediane); če pa bi, potem pa gotovo ne bo treba rezati poddreves, lažjih od mediane, torej bomo imeli v naslednjem koraku pred seboj spet pol krajši L (obdržimo le elemente, težje od mediane). Tako smo torej v $O(n)$ časa skrajšali L za polovico, kar lahko potem ponavljamo, dokler ni dolg le še en element, in je časovna zahtevnost vsega skupaj še vedno le $O(n)$. Vendar pa so bili testni primeri na našem tekmovanju dovolj majhni, da je tudi preprostejša rešitev, ki ureja seznam L in za to porabi $O(n \log n)$ časa, čisto dovolj hitra.

E. Denormalizacija

Vsa števila $a_1, \ldots, a_n$ so bila pri normalizaciji deljena z istim d; če tako na primer pogledamo dva različna indeksa i in j, velja $x_i = a_i/d$ in $x_j = a_j/d$ (pri tem smo zanemarili dejstvo, da sta x_i in x_j zaradi zaokrožanja na dvanajst decimalk sicer lahko malo drugačna od a_i/d oz. a_j/d, in sicer za največ $1/(2 \cdot 10^{12})$). Iz prve od teh enakosti sledi $d = a_i/x_i$, iz druge pa potem $a_j = x_j \cdot d = a_i \cdot x_j/x_i$. Po tej zadnji formuli lahko torej, če poznamo vrednost a_i za neki konkretni i, izračunamo tudi vrednosti vseh ostalih števil a_j v začetnem zaporedju.

Vrednosti a_i sicer ne poznamo, ker pa naloga zagotavlja, da bo ta vrednost nekje med 1 in 10 000, si lahko privoščimo pri rekonstrukciji preizkusiti vse možne vrednosti; pri vsaki od njih, recimo r_i, izračunamo potem še vse ostale r_j in preverimo, če

so tako dobljeni r_j tudi cela števila z območja od 1 do 10 000 — če niso, to pomeni, da vrednost, ki smo jo uporabili za r_i, gotovo ni bila prava (in ne pripelje do veljavne rešitve).

Naloga pravi še, da mora biti največji skupni delitelj števil $r_1, \ldots, r_n$ enak 1; koristno je torej, če pregledujemo možne vrednosti r_i v naraščajočem vrstnem redu. Rešitev, ki jo najdemo pri najmanjšem primernem r_i, ima namreč gotovo največji skupni delitelj 1, kajti če bi imela tista števila še kakšnega večjega skupnega delitelja, bi jih lahko z njim delili in dobili rešitev s še manjšim r_i.

Glede izbora indeksa i lahko vidimo, da naš razmislek načeloma deluje ne glede na to, kateri i izberemo. Če se potrudimo izbrati tistega, pri katerem je število x_i največje, nam ne bo treba skrbeti, da bi nam formula $r_j = r_i \cdot x_j / x_i$ kdaj dala preveliko število r_j (večje od 10 000); po drugi strani, če izberemo tisti i, pri katerem je x_i najmanjši, nam ne bo treba skrbeti, da bi bilo kakšno r_j premajhno (manjše od 1), poleg tega pa bomo tako tudi najhitreje našli pravo vrednost r_i (ker jih preizkušamo od manjših proti večjim). Lahko bi tudi poiskali najmanjši element x_i in največjega, recimo x_k, in potem preizkusili za r_i vrednosti od 1 do $\lfloor 10\,000 \cdot x_i / x_k \rfloor$.

```
i := 1; k := 1;
for j := 2 to n do
  if x_j < x_i then i := j else if x_j > x_k then k := j;
for r_i := 1 to ⌊10 000 · x_i / x_k⌋:
  ok := true;
  for j := 1 to n do if j ≠ i:
    r_j := r_i · x_j / x_i;
    if r_j ni celo število: ok := false; break;
  if ok then break;
```

Tako bi vsaj bilo v idealnem svetu brez zaokrožitvenih napak pri računanju s plavajočo vejico. V resnici so napake že v naših vhodnih podatkih, saj jih dobimo zaokrožene na 12 decimalk; pa tudi pri množenju in deljenju v formuli $r_i \cdot x_j / x_i$ lahko pride do zaokrožitvenih napak (te so sicer majhne v primerjavi s tistimi v vhodnih podatkih, vsaj če uporabimo 64-bitni tip **double** in ne npr. 32-bitnega **float**). Zato, četudi smo izbrali pravo vrednost r_i, ne smemo pričakovati, da bo r_j, izračunan kot $r_i \cdot x_j / x_i$, res celo število. Preden pa si ogledamo nekaj načinov, kako se spoprijeti s to težavo, razmislimo najprej malo podrobneje o tem, kakšno vrednost sploh dobimo, ko (z zaokrožitvenimi napakami) izračunamo r_j po formuli $r_i \cdot x_j / x_i$. Predvsem bomo pokazali, da je ta vrednost, če izberemo pravi r_i, zelo blizu nekemu celemu številu, če pa izberemo napačen r_i, je gotovo precej bolj oddaljena od najbližjega celega števila; ko se prepričamo o tem, lahko nalogo rešimo tako, da uporabimo tisti r_i, pri katerem so vrednosti $r_i \cdot x_j / x_i$ najbližje celim številom (oz. celo poljuben r_i, pri katerem so te vrednosti dovolj blizu celim; seveda se bomo morali šele prepričati o tem, kaj tu šteje za dovolj blizu). Kogar ne zanimajo tehnične podrobnosti, lahko brez velike škode preskoči naslednji dve strani in nadaljuje z branjem pri neenakosti (3).

Zaradi zaokrožanja na 12 decimalk x_i ni nujno enak a_i/d, pač pa velja $|x_i - a_i/d| \leq \eta$ za $\eta = \frac{1}{2} \cdot 10^{-12}$; podobno tudi za x_j. Zaradi omejene natančnosti pri računanju s plavajočo vejico pa pri množenju $r_i \cdot x_j$ in nato pri deljenju tega zmnožka z x_i nastane relativna napaka največ $\varepsilon = 2^{-52}$ (če uporabimo 64-bitni tip **double**, ki

ima 52-bitno mantiso). Zato vrednost r_j ne bo nujno enaka $r_i \cdot x_j/x_i$, pač pa bo zanjo veljalo:

$$\begin{aligned} r_j &\le (1+\varepsilon)^2 r_i \cdot x_j/x_i \le r_i \cdot (a_j/d+\eta)/(a_i/d-\eta) \\ &= (1+\varepsilon)^2 (r_i a_j/a_i)(1+d\eta/a_j)/(1-d\eta/a_i); \end{aligned}$$

za $t = d\eta/a_i$ upoštevajmo, da je $1/(1-t) = 1+t/(1-t)$ in za $t>0$ naprej $< 1+t$:

$$< (1+\varepsilon)^2 (r_i a_j/a_i)(1+d\eta/a_j)(1+d\eta/a_i);$$

pišimo $\alpha = 2\varepsilon+\varepsilon^2$ in $\beta = d\eta$:

$$\begin{aligned} &= (r_i a_j/a_i)(1+\alpha)(1+\beta/a_j)(1+\beta/a_i) \\ &= (r_i a_j/a_i)\big(1+\beta(1/a_j+1/a_i)+\beta^2/(a_i a_j)+\alpha(1+\beta/a_j+\beta/a_i+\beta^2/(a_i a_j))\big) \\ &\le (r_i a_j/a_i)\big(1+\beta(1/a_j+1/a_i)+\beta^2+\alpha(1+\beta)^2\big); \end{aligned}$$

upoštevajmo, da so vsi $a_\bullet$ manjši ali enaki $m := 10^4$ ter da je $d \le m\sqrt{n} \le 10^6$, $\beta = d\eta \le \frac{1}{2}\cdot 10^{-6}$ in $\alpha = 2\varepsilon+\varepsilon^2 < 3\varepsilon$:

$$\le (r_i a_j/a_i)\big(1+\beta(1/a_j+1/a_i)+\tfrac{1}{4}\cdot 10^{-12}+\varepsilon(3+3\cdot 10^{-6}+\tfrac{3}{4}\cdot 10^{-12})\big);$$

upoštevajmo, da je zadnji člen $< 4\varepsilon = 1/2^{50} \le 1/10^{15}$:

$$\begin{aligned} &< (r_i a_j/a_i)\big(1+\beta(1/a_j+1/a_i)+\tfrac{1}{4}\cdot 10^{-12}+10^{-15}\big) \\ &\le (r_i a_j/a_i)\big(1+\beta(1/a_j+1/a_i)+10^{-12}\big). \end{aligned}$$

Podoben razmislek nas pripelje tudi do podobnega rezultata za spodnjo mejo vrednosti r_j in oboje lahko združimo v:

$$|r_j - r_i a_j/a_i| < (r_i a_j/a_i)\big(\beta(1/a_j+1/a_i)+10^{-12}\big). \tag{1}$$

Vrednost r_j, dobljena po formuli $r_i \cdot x_j/x_i$ in z računanjem s plavajočo vejico, je torej blizu vrednosti $r_i a_j/a_i$ (kar je dobro, saj smo razmerje x_j/x_i uporabili ravno zato, ker naj bi bilo blizu razmerja a_j/a_i, pri čemer smo vrednosti $x_\bullet$ dobili kot vhodne podatke, vrednosti $a_\bullet$ pa ne). Zlasti nas seveda zanima najti „pravo“ vrednost r_i, to je $r_i = a_i$, in pri njej dobiti r_j zelo blizu $a_i \cdot a_j/a_i = a_j$, tako da bo iz takega r_j po zaokrožanju na celo število nastala ravno vrednost a_j. Pri $r_i = a_i$, pa tudi pri manjših r_i, bodo vrednosti $r_i a_j/a_i \le a_j \le 10^4$; iskano rešitev bi se torej načeloma moralo dati dobiti, ne da bi imeli kdaj opravka s primerom, ko je $r_i a_j/a_i > 10^4$ — če se nam zgodi to slednje, je to znak, da smo vzeli prevelik r_i in se nam z njim ni treba ukvarjati. Pogoja „$r_i a_j/a_i > 10^4$“ seveda ne moremo neposredno preverjati, saj števil a_i in a_j ne poznamo; vemo pa, da je leva stran blizu r_j. Če uporabimo r_j na levi strani tega pogoja, kaj to pomeni za desno stran? Pri $r_i a_j/a_i \le 10^4$ velja po (1):

$$\begin{aligned} r_j &\le (r_i a_j/a_i)(1+\beta/a_j+\beta/a_i+10^{-12}) \\ &= (r_i a_j/a_i)(1+\beta/a_i+10^{-12})+\beta r_i/a_i; \end{aligned}$$

upoštevajmo $r_i a_j/a_i \le 10^4$, $r_i \le 10^4$ in $\beta \le \frac{1}{2}\cdot 10^{-6}$:

$$\begin{aligned} &\le 10^4(1+\tfrac{1}{2}\cdot 10^{-6}/a_i+10^{-12})+\tfrac{1}{2}\cdot 10^{-6}\cdot 10^4/a_i \\ &= 10^4+10^{-8}+\tfrac{1}{100}/a_i < 10^4+\tfrac{1}{99}, \end{aligned} \tag{2}$$

pri čemer smo v zadnjem koraku upoštevali $a_i \geq 1$. Če torej kadarkoli dobimo $r_j \geq 10^4 + \frac{1}{99}$, smemo iz tega sklepati, da je $r_i a_j/a_i > 10^4$, torej imamo prevelik r_i in se nam z njim ni treba ukvarjati.

V nadaljevanju se bomo zato omejili na primere, ko je $r_j < 10^4 + \frac{1}{99}$. Kaj lahko takrat povemo o $r_i a_j/a_i$? Ni nujno, da je $r_i a_j/a_i$ takrat manjši ali enak 10^4; lahko je tudi večji od 10^4, vendar ne more biti prav veliko večji. Recimo namreč, da je $r_i a_j/a_i \geq 10^4 + D$; po (1) tedaj velja:

$$\begin{aligned} r_j &\geq (r_i a_j/a_i)(1 - \beta(1/a_j + 1/a_i) - 10^{-12}) \\ &= (r_i a_j/a_i)(1 - \beta/a_i - 10^{-12}) - \beta r_i/a_i; \end{aligned}$$

upoštevajmo $\beta \leq \frac{1}{2} \cdot 10^{-6}$, $a_i \geq 1$ in $r_i \leq 10^4$:

$$\geq (r_i a_j/a_i)(1 - \tfrac{1}{2} \cdot 10^{-6} - 10^{-12}) - 10^4 \cdot \tfrac{1}{2} \cdot 10^{-6};$$

upoštevajmo $r_i a_j/a_i \geq 10^4 + D$:

$$\begin{aligned} &\geq (10^4 + D)(1 - \tfrac{1}{2} \cdot 10^{-6} - 10^{-12}) - \tfrac{1}{200} \\ &= 10^4 - \tfrac{1}{100} - 10^{-8} + D(1 - \frac{1}{2} \cdot 10^{-6} - 10^{-12}). \end{aligned}$$

Vprašajmo se, kdaj je to gotovo večje od prej omenjenega praga iz (2), torej od $10^4 + \frac{1}{99}$; to bo pri:

$$\begin{aligned} 10^4 - \tfrac{1}{100} - 10^{-8} + D(1 - \tfrac{1}{2} \cdot 10^{-6} - 10^{-12}) &\geq 10^4 + \tfrac{1}{99} \\ D &\geq (\tfrac{1}{99} + \tfrac{1}{100} + 10^{-8})/(1 - \tfrac{1}{2} \cdot 10^{-6} - 10^{-12}). \end{aligned}$$

Ni se težko prepričati, da je desna stran malo manjša od $\frac{1}{49}$. Lahko torej zaključimo, da če je $r_i a_j/a_i$ večji od $10^4 + \frac{1}{49}$, bo r_j gotovo večji od $10^4 + \frac{1}{99}$ in se bomo s trenutnim r_j nehali ukvarjati. Dokler pa r_j ostane pod $10^4 + \frac{1}{99}$, smo lahko prepričani, da je $r_i a_j/a_i \leq 10^4 + \frac{1}{49}$.

Zdaj ko imamo neko zgornjo mejo vrednosti $r_i a_j/a_i$, se lahko vrnemo k zgornji meji za $|r_j - r_i a_j/a_i|$ iz (1) in o njej povemo še malo več:

$$\begin{aligned} |r_j - r_i a_j/a_i| &< (r_i a_j/a_i)(\beta(1/a_j + 1/a_i) + 10^{-12}) \\ &= \beta r_i/a_i + (r_i a_j/a_i)(\beta/a_i + 10^{-12}) \end{aligned}$$

upoštevajmo $\beta \leq \frac{1}{2} \cdot 10^{-6}$, $r_i \leq 10^4$ in $r_i a_j/a_i \leq 10^4 + \frac{1}{49}$:

$$\begin{aligned} &\leq \tfrac{1}{2} \cdot 10^{-6} \cdot 10^4/a_i + (10^4 + \tfrac{1}{49})(\tfrac{1}{2} \cdot 10^{-6}/a_i + 10^{-12}) \\ &= (\tfrac{1}{100} + \tfrac{1}{98} \cdot 10^{-6})/a_i + 10^{-8} + \tfrac{1}{49} \cdot 10^{-12} \\ &\leq \tfrac{1}{99}/a_i + 2 \cdot 10^{-8}. \end{aligned} \tag{3}$$

Vpeljimo še nekaj oznak: $[t]$ naj bo celo število, ki je najbližje številu t (če je t na polovici med dvema zaporednima celima številoma, uporabimo manjše od njiju); $e(t) := |t - [t]|$ naj pove, za koliko se t premakne, če ga zaokrožimo na najbližje celo število; in $E(r_i) := \max\{e(r_i x_j/x_i) : 1 \leq j \leq n\}$ naj bo največji tak premik, če pri izbranem r_i izračunamo vse r_j po formuli $r_i x_j/x_i$ in vsakega zaokrožimo na najbližje celo število.

„Prava“ vrednost r_i bi bila $r_i = a_i$, dobri pa so tudi večkratniki a_i-ja; recimo torej, da je $r_i = k \cdot a_i$. Takrat nam neenakost (3) dá $|r_j - k \cdot a_j| < \frac{1}{99}/a_i + 2 \cdot 10^{-8}$;

desna stran je veliko manjša od 1/2, zato je r_j-ju najbližje celo število ravno $k \cdot a_j$; torej je na levi $|r_j - k \cdot a_j| = |r_j - [r_j]| = r(e_j)$. Ker velja to za vsak j, lahko zaključimo: $E(k \cdot a_i) < \frac{1}{99}/a_i + 2 \cdot 10^{-8}$.

Kaj pa, če uporabimo za r_i neko „napačno“ vrednost — táko, ki ni večkratnik a_i? Takrat nam (3) pove, da bo r_j blizu števila $r_i a_j/a_i$. Ker r_i ni večkratnik a_i, mora biti vsaj en prafaktor a_i-ja, recimo p, prisoten v r_i z nižjo stopnjo (morda 0) kot v a_i; in ker je največji skupni delitelj števil $a_1, \ldots, a_n$ enak 1, mora obstajati vsaj en tak a_j, ki ni večkratnik p-ja; zanj pa je potem p tudi v $r_i a_j$ prisoten z manjšo stopnjo kot v a_i, torej $r_i a_j/a_i$ ni celo število. Ker ni celo število, je pa ulomek z imenovalcem a_i, mora biti od najbližjega celega števila oddaljen vsaj za $1/a_i$. Iz (3) potem sledi, da mora biti r_j od najbližjega celega števila oddaljen vsaj za $1/a_i - \frac{1}{99}/a_i - 2 \cdot 10^{-8}$. Ker velja to pri vsaj enem j, velja tudi za maksimum po vseh j, torej imamo $E(r_i) \geq \frac{98}{99}/a_i - 2 \cdot 10^{-8}$, če r_i ni večkratnik a_i. (Ta spodnja meja za $E(r_i)$ je, ker je $a_i \leq 10^4$, večja ali enaka $\frac{98}{99} \cdot 10^{-4} - 2 \cdot 10^{-8} \geq \frac{97}{99} \cdot 10^{-4}$.)

Če primerjamo obe meji, ki smo ju dobili za E, je druga veliko večja od prve: njuno razmerje je

$$\frac{\frac{98}{99}/a_i - 2 \cdot 10^{-8}}{\frac{1}{99}/a_i + 2 \cdot 10^{-8}} = \frac{98 - 2 \cdot 10^{-8} \cdot 99 a_i}{1 + 2 \cdot 10^{-8} \cdot 99 a_i} \leq \frac{9780}{100},$$

pri čemer smo v zadnjem koraku upoštevali, da je $1 \leq a_i \leq 10^4$. Z drugimi besedami, $E(r_i)$ je pri vsakem napačnem r_i vsaj 97,8-krat tolikšna kot pri vsakem pravem r_i.

Nalogo lahko torej rešimo tako, da po vrsti preizkušamo $r_i = 1, 2, 3, \ldots$ in pri vsakem izračunamo $E(r_i)$. Tisti r_i, pri katerem je vrednost $E(r_i)$ najmanjša, je potem gotovo neki večkratnik a_i; zanj izračunamo vse $r_j = r_i x_j/x_i$, zaokrožimo vsakega na najbližje celo število in jih še delimo z njihovim največjim skupnim deliteljem (za primer, da naš r_i ni enak a_i, ampak je neki večji večkratnik števila a_i).

V resnici tudi ni nujno, da pri vsakem r_i res izračunamo $E(r_i)$; če pri trenutnem r_i za neki j opazimo, da je $e(r_i x_j/x_i)$ večja od najmanjše doslej znane vrednosti $E(\cdot)$, potem bo trenutni $E(r_i)$ gotovo večji od tiste najmanjše doslej znane, zato lahko nad trenutnim r_i takoj obupamo. Poleg tega, seveda, če opazimo, da je $r_j \geq 10^4 + \frac{1}{99}$, lahko zaključimo, da je trenutni r_i že prevelik, in postopek končamo. Še ena izboljšava pa je tale: zgoraj smo videli, da je pri „napačnih“ r_i vrednost $E(r_i)$ gotovo $\geq \frac{97}{99} \cdot 10^{-4}$; če torej pri nekem r_i dobimo manjšo vrednost $E(r_i)$ od te, je ta r_i gotovo eden od „pravih“ in lahko pri njem tudi končamo.

```
podprogram DENORM1(vhod: x_1, ..., x_n; izhod: r_1, ..., r_n):
  i := 1; for j := 2 to n do if x_j < x_i then i := j;
  E* := ∞;
  for r_i := 1 to 10 000:
    E := 0; ok := true;
    for j := 1 to n do if j ≠ i:
      r_j := r_i · x_j/x_i;
      if r_j > 10^4 + 1/99 then ok := false; break;
      e := |r_j − [r_j]|; if e ≤ E then continue;
      E := e; if E ≥ E* then break;
    if not ok then break; (* r_i je že prevelik *)
```

if $E \le E^\star$ **then** $E^\star := E$, $r_i^\star := r_i$;
if $E < \frac{97}{99} \cdot 10^{-4}$ **then break**;
for $j := 1$ **to** n **do** $r_j := [r_i^\star x_j / x_i]$;
deli vsa števila $r_1, \ldots, r_n$ z njihovim najmanjšim skupnim deliteljem;

Ta postopek bi sicer deloval ne glede na to, kateri i si izberemo; v zgornji psevdokodi smo izbrali i tako, da je x_i najmanjši, saj to pomeni, da je tudi a_i najmanjši in bo naša zanka tem prej dosegla kakšnega od „pravih“ r_i-jev (= večkratnikov a_i).

Nalogo lahko rešimo tudi tako, da pri vsakem r_i izračunamo vse r_j po formuli $r_i \cdot x_j / x_i$, jih zaokrožimo na cela števila, normaliziramo in pogledamo, če po normalizaciji odstopajo od vhodnih x_j za manj kot 10^{-6}; to mejo namreč besedilo naloge navaja kot pogoj za to, da bo naša rešitev sprejeta. Čim najdemo tako rešitev, se ustavimo:

podprogram DENORM2(vhod: $x_1, \ldots, x_n$; izhod: $r_1, \ldots, r_n$):
$i := 1$; **for** $j := 2$ **to** n **do if** $x_j < x_i$ **then** $i := j$;
for $r_i := 1$ **to** 10 000:
$d' := r_i \cdot r_i$;
for $j := 1$ **to** n **do if** $j \ne i$:
$r_j := [r_i \cdot x_j / x_i]$; $d' := d' + r_j \cdot r_j$;
$ok :=$ **true**; $d' := \sqrt{d'}$;
for $j := 1$ **to** n:
if $|r_j / d' - x_j| > 10^{-6}$ **then** $ok :=$ **false**; **break**;
if ok **then return**;

Če ne prej, se ta postopek ustavi pri $r_i = a_i$; takrat namreč, kot smo videli zgoraj, po zaokrožanju gotovo dobimo $r_j = a_j$ (za vse j), zato potem pri normalizaciji (torej pri deljenju z d', v katerem smo si pripravili vrednost $(\sum_{j=1}^n r_j^2)^{1/2}$) dobimo vrednosti, ki se od x_j razlikujejo le zato, ker so bile slednje pri izpisu zaokrožene na 12 decimalk; razlika $|r_j/d' - x_j|$ bo torej $\le \eta = \frac{1}{2} \cdot 10^{-12}$, kar je veliko manj od 10^{-6}. To pa tudi pomeni, da si lahko zaradi varnosti privoščimo prag 10^{-6} v našem ustavitvenem pogoju malo zmanjšati (npr. na 10^{-7}): drugače bi nas namreč lahko skrbelo, da bi naša rešitev po normalizaciji odstopala od x_j za malo več kot 10^{-6}, vendar bi naš program zaradi zaokrožitvenih napak pri računanju korena in deljenju z d' pomotoma dobil vtis, da je odstopanje malo manjše od 10^{-6} (ocenjevalni strežnik take napake ne more narediti, ker ne računa s plavajočo vejico, ampak z ulomki z neomejeno natančnostjo — razred Fraction v pythonu).

Slabost te rešitve je, da mora iti pri vsakem r_i po čisto vseh j, preden lahko izračuna d' in začne normalizirati dobljene r_j ter tako preverjati, ali so rezultati primerni. Lepo pri tej rešitvi pa je, da ni nujno, da gre glavna zanka vse do $r_i = a_i$; lahko se zgodi, da najde veljavno rešitev že prej. Na primer, če bi imeli $a = (4321, 10000)$, bi bila veljavna rešitev že $r = (35, 81)$:

$$\begin{array}{lcl} a = (4321, 10000) & \to & x = (0{,}396654047537,\ 0{,}917968172963) \\ r = (35, 81) & \to & \quad (0{,}396653092549,\ 0{,}917968585613) \end{array}$$

Doslej smo videli dve rešitvi, ki načeloma delujeta ne glede na to, kako izberemo indeks i, vendar pa smo v praksi izbrali tistega, pri katerem je x_i (in zato a_i)[20]

[20] O tem, da je vrstni red a-jev enak kot vrstni red x-ov, se ni težko prepričati. Recimo, da je

najmanjši. Zdaj pa recimo, da bi izbrali tisti i, pri katerem je x_i (in zato a_i) največji. To pomeni, da pri vsakem j velja $a_j \le a_i$, torej $a_j/a_i \le 1$; poleg tega je potem $d = (\sum_{j=1}^{n} a_j^2)^{1/2} \le a_i\sqrt{n} \le 100a_i$, zato $\beta/a_i = (d/a_i)\eta \le \frac{1}{2} \cdot 10^{-10}$. Uporabimo to v v meji (1):

$$\begin{aligned} |r_j - r_i a_j/a_i| &< (r_i a_j/a_i)\big(\beta(1/a_j + 1/a_i) + 10^{-12}\big) \\ &= r_i(\beta/a_i)(1 + a_j/a_i) + 10^{-12} r_i(a_j/a_i); \end{aligned}$$

upoštevajmo $a_j/a_i \le 1$, $\beta r_i/a_i \le \frac{1}{2} \cdot 10^{-10}$ in $r_i \le 10^4$:

$$\begin{aligned} &\le 10^4 \cdot \tfrac{1}{2} \cdot 10^{-10}(1+1) + 10^{-12} \cdot 10^4 \cdot 1 \\ &= 10^{-6} + 10^{-8} < 2 \cdot 10^{-6}. \end{aligned}$$

Vrednost r_j torej odstopa od $r_i a_j/a_i$ za manj kot $2 \cdot 10^{-6}$. Videli smo že, da je za „napačne“ r_i (take, ki niso večkratniki a_i) vrednost $r_i a_j/a_i$ gotovo pri vsaj enem j ne-cela in je oddaljena od najbližjega celega števila vsaj za $1/a_i$, kar je vsaj 10^{-4}; tedaj je torej r_j oddaljena od najbližjega celega števila za vsaj $10^{-4} - 2 \cdot 10^{-6}$. Po drugi strani pa je pri „pravih“ r_i (večkratnikih a_i) vrednost $r_i a_j/a_i = a_j$ celo število in r_j je oddaljena od njega za manj kot $2 \cdot 10^{-6}$. Ni torej treba drugega, kot da naša rešitev preizkuša različne r_i-je in preverja, ali je $E(r_i) < 2 \cdot 10^6$. Če gremo po naraščajočih r_i, bo prvi primerni ravno $r_i = a_i$, torej nam tudi ni treba skrbeti glede krajšanja r-jev z največjim skupnim deliteljem.

podprogram DENORM3(vhod: $x_1, \ldots, x_n$; izhod: $r_1, \ldots, r_n$):
  $i := 1$; **for** $j := 2$ **to** n **do if** $x_j > x_i$ **then** $i := j$;
  for $r_i := 1$ **to** 10 000:
    $ok :=$ **true**;
    for $j := 1$ **to** n **do if** $j \ne i$:
      $r_j := r_i \cdot x_j/x_i$;
      if $|r_j - [r_j]| \ge 2 \cdot 10^{-6}$ **then** $ok :=$ **false**; **break**;
      $r_j := [r_j]$;
    if ok **then return**;

Zunanja zanka se ustavi pri $r_i = a_i$, pri manjših r_i pa notranja zanka hitro odneha, čim opazi, da kakšen r_j preveč odstopa od najbližjega celega števila; zato je v praksi ta rešitev zelo hitra.

F. Razlike

Recimo, da bi za vsak položaj v nizih j (od 1 do m) in za vsako črko abecede c (pri naši nalogi so možne le štiri črke, od `A` do `D`) pripravili vektor z n komponentami, ki bi kazale, kateri vhodni nizi imajo na j-tem mestu znak c. To je torej vektor $\mathbf{z}_{jc} = (z_{jc1}, \ldots, z_{jcn})$, kjer je $z_{jci} = [\![s_i[j] = c]\!]$ (izraz $[\![\cdot]\!]$ ima vrednost 1, če je pogoj v oglatih oklepajih izpolnjen, sicer pa ima vrednost 0).

$a_i < a_j$. Ali bi se lahko zgodilo, da bi bil $x_i \ge x_j$? Iz tega bi sledilo $a_i/d + \eta \ge x_i \ge x_j \ge a_j/d - \eta$, torej $2d\eta \ge a_j - a_i$; leva stran je (ker je $\eta = \frac{1}{2} \cdot 10^{-12}$ in $d \le m\sqrt{n} \le 10^6$) manjša ali enaka 10^{-6}; desna stran pa je večja ali enaka 1, ker je $a_i < a_j$ in sta a_i in a_j celi števili. Tako je torej nemogoče, da bi bilo $2d\eta \ge a_j - a_i$; predpostavka $x_i \ge x_j$ nas je pripeljala v protislovje, torej mora veljati $x_i < x_j$.

Definirajmo še $\mathbf{e}_i$ kot n-dimenzionalni vektor, ki ima povsod ničle, le na i-tem mestu enico; in $\mathbf{1}$ kot n-dimenzionalni vektor samih enic, $\mathbf{1} = (1, \ldots, 1)$. Razlika $\mathbf{1} - \mathbf{z}_{jc}$ nam potem pove, kateri vhodni nizi na j-tem mestu nimajo znaka c.

Razdalja med nizi, definirana v besedilu naloge, se imenuje *Hammingova razdalja*. Recimo zdaj, da nas zanimajo Hammingove razdalje od nekega niza t do vseh s_i. Na j-tem mestu ima t znak $t[j]$, torej imamo v vektorju $\mathbf{1} - \mathbf{z}_{j,t[j]}$ enice pri tistih i, za katere je $s_i[j] \neq t[j]$, torej kjer se s_i razlikuje od t pri j-tem znaku. Če to seštejemo (po komponentah) po vseh j, bomo dobili pri i-ti komponenti ravno število mest j, na katerih se s_i razlikuje od t; to pa je ravno Hammingova razdalja med nizoma s_i in t:

$$\text{vektor } \mathbf{h}(t) := m\mathbf{1} - \sum_{j=1}^{m} \mathbf{z}_{j,t[j]} \text{ je torej enak } \big(H(t, s_1), \ldots, H(t, s_n)\big).$$

Naloga zahteva, da poiščemo med vhodnimi nizi takega, ki ima Hammingovo razdaljo do vseh ostalih enako k. Iščemo torej tak s_ℓ, za katerega je $\mathbf{h}(s_\ell) = k \cdot (\mathbf{1} - \mathbf{e}_\ell)$ (torej vektor, ki ima povsod vrednosti k, le na ℓ-tem mestu je ničla).

Težava dosedanjega razmisleka je, da je delo z vektorji preveč zamudno. Pri izračunu $\mathbf{h}(s_\ell)$ bi morali sešteti m vektorjev s po n komponentami, torej bi nam to vzelo $O(nm)$ časa, in ker bi morali to narediti za vsak ℓ od 1 do n, bi bila časovna zahtevnost te rešitve $O(n^2 m)$, kar je prepočasi.

Pomagamo si lahko tako, da s kakšno zgoščevalno funkcijo (*hash function*) preslikamo naše n-dimenzionalne vektorje v eno dimenzijo: vektorju $\mathbf{x}$ pripišemo zgoščevalno kodo $f(\mathbf{x})$, ki ji pravimo tudi „podpis“ ali „prstni odtis“ (angl. *fingerprint*) vektorja $\mathbf{x}$. Za preverjanje, ali sta dva vektorja enaka — na primer: ali je $\mathbf{h}(s_\ell) = k \cdot (\mathbf{1} - \mathbf{e}_\ell)$ — bomo potem najprej preverili, če imata enak podpis; če ga nimata, lahko takoj zaključimo, da sta različna. Če pa imata enak podpis, se sicer načeloma še vedno lahko zgodi, da sta vektorja različna, toda če primerno izberemo f, je verjetnost takšnega „trka“ dovolj majhna, da se zaradi nje ni treba vznemirjati. Lahko pa, ko pri nekem ℓ opazimo, da imata $\mathbf{h}(s_\ell)$ in $k(\mathbf{1} - \mathbf{e}_\ell)$ enak podpis, posebej izračunamo Hammingove razdalje med s_ℓ in vsemi ostalimi nizi, da se prepričamo, če so res vse enake k. Verjetnost, da bi morali takšen preizkus opraviti pri več kot enem ℓ, je neznatno majhna, če pa bi do nje vseeno prišlo, bomo tako vsaj zagotovo vrnili pravilen rezultat.

Toda da bo od takšnih podpisov res kakšna korist, je pomembno, da znamo podpis vektorja $\mathbf{h}(s_\ell)$, torej vrednost $f(\mathbf{h}(s_\ell))$, izračunati, ne da bi izračunali ta vektor sam — kajti če moramo najprej izračunati te vektorje v eksplicitni obliki, je potem že vseeno, če takrat še preverimo, ali imajo res povsod vrednost k, le na ℓ-tem mestu ničlo. Zelo koristno za naš namen je, če je funkcija f linearna, torej če je $f(\mathbf{x} \pm \mathbf{y}) = f(\mathbf{x}) \pm f(\mathbf{y})$; potem lahko podpis vsote več vektorjev izračunamo iz podpisov posameznih seštevancev, vsote same pa nam pri tem ni treba računati. Takrat bo na primer $f(\mathbf{h}(s_\ell)) = f(m\mathbf{1} - \sum_{j=1}^{m} \mathbf{z}_{j,t[j]}) = mf(\mathbf{1}) - \sum_{j=1}^{m} f(\mathbf{z}_{j,t[j]})$ in podobno $f(k \cdot (\mathbf{1} - \mathbf{e}_\ell)) = k \cdot (f(\mathbf{1}) - f(\mathbf{e}_\ell))$.

Primerna oblika funkcije f (torej linearna, da lahko uporabimo razmislek iz prejšnjega odstavka) je potem $f(\mathbf{x}) = (\sum_{i=1}^{n} \lambda_i x_i) \bmod M$, kjer za M vzamemo kakšno veliko praštevilo, npr. $10^9 + 7$. Z vrednostmi funkcije f računamo vedno znotraj $\mathbb{Z}_M$, torej s celimi števili po modulu M; tako lahko na primer $f(\mathbf{x} + \mathbf{y})$ izračunamo

kot $(f(\mathbf{x}) + f(\mathbf{y})) \bmod M$. Kaj vzamemo za konstante λ_i, ni tako pomembno, lahko so tudi kar naključna števila med 1 in $M - 1$; posebej elegantna možnost pa je, da si izberemo neko manjše praštevilo p in uporabimo $\lambda_i := p^i \bmod M$ — tako dobimo znano Rabinovo zgoščevalno funkcijo. Zapišimo našo rešitev še s psevdokodo; abecedo (množico vseh črk, ki se lahko pojavijo v naših nizih) označimo z A:

$\lambda[1] := p$; **for** $i := 2$ **to** n **do** $\lambda[i] := (p \cdot \lambda[i-1]) \bmod M$;
$\nu := 0$; **for** $i := 1$ **to** n **do** $\nu := (\nu + \lambda[i]) \bmod M$;
(* *Zdaj je* $\nu = f(\mathbf{1})$ *in* $\lambda[i] = f(\mathbf{e}_i)$. *)

for $j := 1$ **to** m **do for** $c \in A$ **do** $Z[j, c] := 0$;
for $i := 1$ **to** n **do for** $j := 1$ **to** m **do** $Z[j, s_i[j]] := (Z[j, s_i[j]] + \lambda[i]) \bmod M$;
(* *Zdaj je* $Z[j, c] = f(\mathbf{z}_{jc})$. *)

for $\ell := 1$ **to** n:
  $H := 0$; **for** $j := 1$ **to** m **do** $H := (H + \nu + M - Z[j, s_\ell[j]]) \bmod M$;
  $G := \big(k \cdot (\nu + M - \lambda[\ell])\big) \bmod M$;
  (* *Zdaj je* $H = f(\mathbf{h}(s_\ell))$ *in* $G = f(k \cdot (\mathbf{1} - \mathbf{e}_\ell))$. *)
  if $G = H$ **then** s_ℓ je (kandidat za) niz, po katerem sprašuje naloga;

Časovna zahtevnost te rešitve je $O(m|A| + nm)$, kar je naprej enako $O(nm)$, če je abeceda majhna v primerjavi s številom nizov n; v našem primeru to drži, pa tudi če ne bi držalo, nastopa $O(m|A|)$ v časovni zahtevnosti le zaradi inicializacije tabele Z, čemur bi se lahko izognili, če bi jo predstavili z razpršeno tabelo (ki bi hranila le neničelne elemente). Če dobimo več kot enega kandidata za iskani niz, lahko k temu dodamo še $O(nm)$ časa na vsakega kandidata, da izračunamo prave Hammingove razdalje med njim in vsemi ostalimi nizi; ali pa še enkrat ponovimo gornji postopek z drugačnima M in p in upamo, da bo tokrat ostal le en kandidat.

Oglejmo si še malo slabšo rešitev, ki pa je še vseeno dovolj hitra za testne primere in časovne omejitve na našem tekmovanju. Izberimo si neko podmnožico $I \subseteq \{1, \ldots, n\}$; za vsak položaj j od 1 do m in za vsako črko abecede $c \in A$ nato preštejmo, koliko nizov s_i za $i \in I$ ima na položaju j črko c; recimo temu ζ_{jc}. To lahko naredimo z enim prehodom čez vse te nize, torej v $O(m(|A| + |I|))$ časa.

Recimo zdaj, da nas za neki niz t zanima, ali ima Hammingovo razdaljo do vseh nizov s_i za $i \in I$ enako k. Pri vsakem položaju j nam razlika $|I| - \zeta_{j,t[j]}$ pove, koliko izmed tistih nizov se na tem položaju razlikuje od niza t; če to seštejemo po vseh j, dobimo ravno skupno število vseh neujemanj med t in nizi s_i za $i \in I$ — to pa je ravno vsota Hammingovih razdalj med t in vsemi temi nizi:

$$h(t) := m|I| - \sum_{j=1}^{m} \zeta_{j,t[j]} \text{ je torej enaka } \sum_{i \in I} H(t, s_i).$$

Če je t niz, ki ga iščemo, mora biti vsota na desni enaka $|I| \cdot k$ (ali pa $(|I| - 1) \cdot k$, če je tudi t eden od nizov s_i za $i \in I$). Če torej neki t temu pogoju ne ustreza, vemo, da to ni niz, po katerem sprašuje naloga. Vzdrževali bomo množico kandidatov L za iskani niz; na začetku ta vsebuje vse vhodne nize, nato pa iz nje zavržemo tiste, ki ne ustrezajo opisanemu pogoju. To ponavljamo z različnimi množicami I, dokler nam ne ostane le še en kandidat — zanj pa potem vemo, da mora biti iskani niz ravno on.

$L := \{1, 2, \ldots, n\}$;
while $|L| > 1$:
  $I :=$ množica, ki jo tvori naključno izbranih $n/2$ elementov iz $\{1, 2, \ldots, n\}$;
  for $\ell \in K$ **do** $H[\ell] := 0$; (* *V* $H[\ell]$ *bo nastajala vsota* $h(s_\ell)$. *)
  for $j := 1$ **to** m:
    for $c \in A$ **do** $Z[c] := 0$; (* *V* $Z[c]$ *bo nastajala vrednost* ζ_{jc}. *)
    for $i \in I$ **do** $Z[s_i[j]] := Z[s_i[j]] + 1$;
    for $\ell \in L$ **do** $H[\ell] := H[\ell] + |I| - Z[s_\ell[j]]$;
  (* *Obdržimo kandidate, ki imajo pravo vsoto Hammingovih razdalj* *)
  $L' := \{\}$; (* *do* s_i *za* $i \in U$. *)
  for $\ell \in L$:
    if $\ell \in I$ **then** $g := (|I| - 1) \cdot k$ **else** $g := |I| \cdot k$;
    if $H[\ell] = g$ **then** dodaj ℓ v L';
  $L := L'$;

Vsaka iteracija glavne zanke vzame $O(m(|A|+|I|+|L|))$ časa, kar je $O(nm)$. Koliko iteracij pa bo treba? Recimo, da za neki niz s_ℓ pogledamo vsoto $\sum_{i=1}^{n} H(s_\ell, s_i)$; neugoden izid za nas nastopi, če seštevanci (pri $i \neq \ell$) v tej vsoti niso vsi enaki k, vsota pa je vendarle enaka $(n-1)k$. To se zgodi na primer, če je od dveh seštevancev eden manjši od k, drugi večji, seštejeta pa se ravno v $2k$. Ko razdelimo seštevance na dve skupini — glede na to, ali je $i \in I$ ali ne — pride vsak od teh dveh seštevancev z verjetnostjo $1/2$ v skupino I (ker smo I izbrali tako, da je v njej $n/2$ od n vhodnih nizov); z verjetnostjo $1/2$ torej pride v I natanko eden od njiju, zato vsota $\sum_{i \in I} H(s_\ell, s_i)$ ne bo večkratnik k-ja in bomo torej niz s_ℓ zavrgli iz množice kandidatov. (Ta razmislek je seveda le neformalen, saj je takih parov seštevancev lahko več, lahko pa tudi nastopijo večje skupine npr. treh seštevancev, od katerih ni nobeden enak k, se pa vsi trije seštejejo v $3k$ in podobno; toda pri takih kombinacijah je verjetnost, da bi vsi seštevanci prišli v I ali vsi ostali zunaj njega, še manjša kot $1/2$.) Tako si torej lahko predstavljamo, da se množica kandidatov L po vsaki iteraciji glavne zanke zmanjša približno za polovico, torej bomo izvedli $O(\log n)$ iteracij glavne zanke in časovna zahtevnost rešitve je $O(nm \log n)$. Pri naših poskusih na testnih primerih s tekmovanja je bilo res treba največ kakšnih $\log_2 n$ iteracij, pri mnogih testnih primerih pa je že po prvi iteraciji ostal en sam kandidat in se je postopek končal.

Obe tu opisani rešitvi sta si v tesnem sorodstvu; če bi pri prvi rešitvi vzeli zgoščevalno funkcijo $f(\mathbf{x}) = \sum_{i \in I} x_i$, bi dobili ravno količine iz druge rešitve: $\zeta_{jc} = f(\mathbf{z}_{jc})$ in $h(t) = f(\mathbf{h}(t))$.

G. Požrešni predali

Požrešnemu algoritmu ne moremo povsem preprečiti, da bi našel veljaven razpored zvezkov v predale, če tak razpored obstaja.[21] Največ, kar lahko naredimo, je torej to, da sestavimo primer, v katerem bo verjetnost, da požrešni algoritem pride do

[21] O tem se lahko prepričamo takole. Če oštevilčimo zvezke in predale od 1 do n, si lahko razpored predstavljamo kot permutacijo π nad množico $\{1, \ldots, n\}$, pri čemer $\pi(z)$ pove, v kateri predal bi dali zvezek z. Naj bo torej π neki veljaven razpored, torej tak, v katerem noben zvezek z ni prevelik za v predal $\pi(z)$.

Potem lahko z indukcijo po t pokažemo naslednje: če požrešni algoritem v prvih $t-1$ korakih ni uporabil nobenega zvezka ali predala drugače, kot je tisti zvezek ali predal uporabljen v razporedu

veljavnega razporeda, čim manjša. Pri tem nam bo v pomoč dejstvo, da je število zvezkov in predalov n pri tej nalogi precej veliko, vsaj 150. Poiščimo najprej neki majhen primer z $m \ll n$ zvezki in predali, nato pa ga razmnožimo v $k := \lfloor n/m \rfloor$ izvodov. Če ima požrešni algoritem na vsakem izvodu verjetnost p, da najde veljaven razpored, potem bo verjetnost, da mu to uspe v vseh k izvodih, le še p^k.

Iskanje majhnih primerov z rekurzijo. Kako poiskati majhen primer (nabor m zvezkov in m predalov), ki bi ga lahko uporabili kot osnovo takšne konstrukcije? Ena možnost je, da si napišemo preprost rekurzivni podprogram, ki za dani primer izračuna verjetnost, da požrešni algoritem najde veljaven razpored. Pri tem ni treba drugega, kot da preizkusimo vse možnosti, ki jih ima požrešni algoritem na izbiro pri svojem delovanju:

podprogram VerjetnostUspeha(A):
  vhod: A je množica parov (z, p), ki povedo, katere zvezke smo že razporedili v katere predale;
  izhod: verjetnost, da požrešni algoritem dopolni A v veljaven razpored vseh n zvezkov;

  če je v A že n parov, vrni 1;

  (* *Za vsak zvezek z določi število kandidatov $k[z]$.* *)
  za vsak zvezek z:
    če nastopa z v kakšnem paru iz A, naj bo $k[z] := \infty$;
    sicer naj bo $k[z]$ število predalov p, ki še ne nastopajo v nobenem paru iz A in ki so dovolj veliki, da bi šel zvezek z lahko vanje;

  (* *Za vsak predal p določi število kandidatov $k'[p]$.* *)
  za vsak predal p:
    če nastopa p v kakšnem paru iz A, naj bo $k'[p] := \infty$;
    sicer naj bo $k'[p]$ število zvezkov z, ki še ne nastopajo v nobenem paru iz A in ki so dovolj majhni, da bi lahko šli v predal p;

  $\kappa :=$ najmanjša izmed vseh vrednosti $k[1], \ldots, k[n], k'[1], \ldots, k'[n]$;
  $\nu :=$ število zvezkov oz. predalov, ki imajo $k[z] = \kappa$ oz. $k'[p] = \kappa$;
  $s := 0$;
  za vsak zvezek z, če je $k[z] = \kappa$:
    za vsak predal p, ki še ne nastopa v nobenem paru iz A:
      če gre z v p, prištej $s := s +$ VerjetnostUspeha($A \cup \{(z, p)\}$);
  za vsak predal p, če je $k'[p] = \kappa$:
    za vsak zvezek z, ki še ne nastopa v nobenem paru iz A:

π, in če se potem na t-tem koraku odloči, da bo nekam razporedil zvezek z (oz. predal p), ima med drugim na voljo zanj tudi predal $\pi(z)$ (oz. zvezek $\pi^{-1}(p)$), in če ima srečo, bo izbral prav tega, tako da se bo njegov nastajajoči razpored tudi po t-tem koraku ujemal z razporedom π.

Pri $t = 1$ trditev drži, saj ni še noben zvezek ali predal zaseden. Recimo potem, da se je v prvih $t-1$ korakih naš požrešni razpored ujemal s π in da se je v t-tem koraku požrešni algoritem odločil razporejati zvezek z (če bi se odločil za predal, je razmislek podoben). Ali je mogoče, da predal $\pi(z)$ takrat ni več na voljo? To bi pomenilo, da smo v enem od prejšnjih korakov v ta predal dali neki drug zvezek z'; toda v π je v tem predalu zvezek z, ne pa z', torej smo v nasprotju s predpostavko, da se naš požrešni razpored v prvih $t-1$ korakih ni razlikoval od razporeda π. □

Po n takšnih korakih, če imamo vsakič srečo, lahko dobimo torej s požrešnim algoritmom ravno veljavni razpored π.

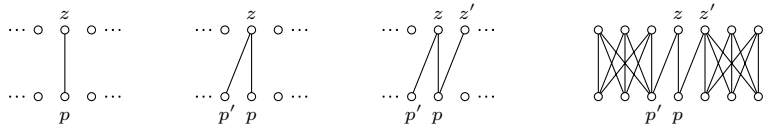
```
    če gre z v p, prištej s := s + VerjetnostUspeha(A ∪ {(z, p)});
  vrni s/(ν · κ);
```

glavni klic: VerjetnostUspeha({});

Zdaj lahko generiramo naključne majhne primere in jih preizkušamo s tem podprogramom. Pri $m \le 4$ se sicer izkaže, da se sploh ne more zgoditi, da požrešni algoritem ne bi našel veljavnega razporeda; pri $m = 5$ pa lahko hitro najdemo primere, kjer požrešnemu algoritmu uspe z verjetnostjo le $p = 5/6$. Če to razmnožimo na 30 izvodov, da dobimo $n = 150$ predalov in zvezkov (kar je najmanjši možni n, po katerem lahko naloga sprašuje), bo verjetnost uspeha le $(5/6)^{30}$, kar je približno 0,4 %. To je sicer še vedno neprijetno veliko; če se to zgodi ravno pri preizkušanju na ocenjevalnem strežniku (in dovolj je že, da se zgodi to enkrat v dvajsetih testnih primerih, kolikor jih je pri tej nalogi), bo naša rešitev zavrnjena.

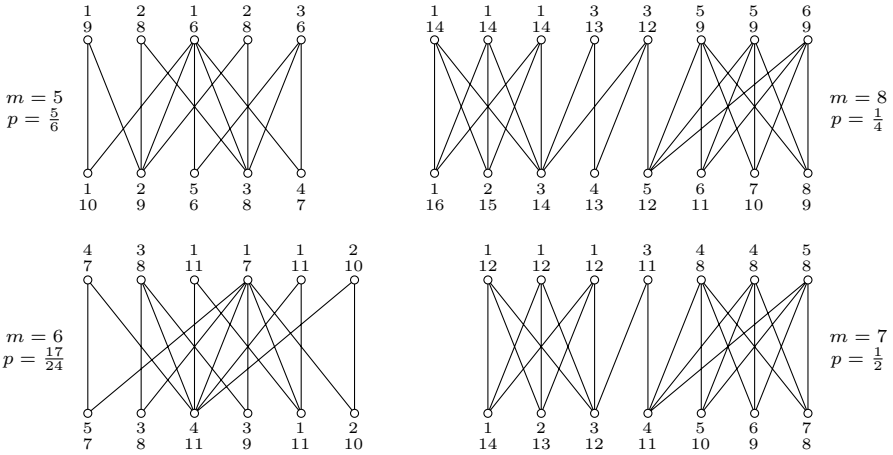
Pri $m = 6$ lahko z naključnim generiranjem hitro najdemo primere z verjetnostjo uspeha $p = 3/4$ in celo $17/24$, kar lahko razmnožimo na 25 izvodov (da dobimo $n = 150$) in bo verjetnost uspeha le še $(17/24)^{25}$, kar je pribl. 0,02 %, kar je že precej bolje. Do še precej boljših rešitev pa lahko pridemo pri $m = 7$ ali 8; rezultate povzemata tabela in slika na str. 147.

Sestavljanje majhnega primera z razmislekom. Lahko pa namesto generiranja in preizkušanja naključnih primerov sestavimo majhen testni primer tudi z razmislekom. To, kateri zvezek gre v kateri predal, lahko predstavimo z dvodelnim (bipartitnim) grafom, v katerem ena vrsta točk predstavlja zvezke, druga predale, povezave med njimi pa kažejo, kateri zvezek gre v kateri predal (gl. sliko spodaj). (1) Začnimo torej z zvezkom z, ki naj gre (pri veljavnem razporedu, za katerega upamo, da ga bo požrešni algoritem zgrešil) v predal p. (2) Toda če naj ima požrešni algoritem možnost zgrešiti veljavni razpored, mora imeti pri zvezku z na izbiro še neki drug predal, recimo p', tako da se bo imel možnost odločiti narobe, če bo imel smolo (in ta izbira ga mora potem sčasoma pripeljati v slepo ulico, ne pa do veljavnega razporeda vseh zvezkov v predale). (3) Ker imamo zdaj pri z na izbiro dva predala, morata biti tudi pri p na izbiro (vsaj) dva zvezka, saj drugače požrešni algoritem ne bo začel pri z, pač pa pri p (in ga pravilno sparil z z, ker druge možnosti sploh ne bo imel); recimo torej, da je p povezan ne le z z, pač pa še z nekim drugim zvezkom z'. (4) Koristno je, če imajo vsi drugi zvezki in predali vsaj tri povezave, tako da požrešni algoritem ne bo v skušnjavi, da bi začel kje drugje kot pri z ali p. Postavimo torej na vsako stran po eno gručo treh zvezkov in treh predalov, ki so povezani vsi z vsakim; eni taki gruči naj pripada prej omenjeni zvezek z', drugi pa predal p'.



(1) (2) (3) (4)

Razmislimo, kaj se lahko požrešnemu algoritmu zgodi na tem primeru. Na začetku si lahko izbere z ali p, ker imata po dva kandidata, ostali zvezki in predali pa vsaj tri. Recimo, da si izbere z; z verjetnostjo 50 % ga položi v predal p' namesto



Nekaj majhnih primerov, kjer se lahko zgodi, da požrešni algoritem ne najde prave rešitve. Našli smo jih z generiranjem in preizkušanjem naključnih primerov. Pri vsakem je napisana verjetnost p, da požrešni algoritem najde rešitev. Vsak primer je predstavljen z dvodelnim grafom, kjer zgornja vrstica točk predstavlja zvezke, spodnja predale, povezave pa kažejo, kateri zvezek gre lahko v kateri predal. Nad vsakim zvezkom in pod vsakim predalom je napisan par števil, ki pomenita širino in višino tistega zvezka oz. predala. Naslednja tabela kaže, kako lahko te majhne primere razmnožimo do 150 zvezkov in predalov in kakšna je potem verjetnost, da požrešni algoritem najde pravo rešitev:

št. zvezkov in predalov m	verjetnost uspeha p	št. izvodov $k = \lfloor 150/m \rfloor$	verjetnost uspeha po množitvi, p^k
5	5/6	30	$4{,}21 \cdot 10^{-3}$
6	17/24	25	$1{,}80 \cdot 10^{-4}$
7	1/2	21	$4{,}77 \cdot 10^{-7}$
8	1/4	18	$1{,}46 \cdot 10^{-11}$

v p. Zdaj prideta v poštev za leve tri zvezke le leva dva predala, torej se razporeda očitno ne bo dalo sestaviti do konca. Podobno bi bilo, če bi požrešni algoritem na začetku izbral p; z verjetnostjo 50 % položi vanj zvezek z' namesto z, potem pa prideta v poštev za desne tri predale le desna dva zvezka, torej se razporeda ne bo dalo sestaviti do konca. Vidimo torej, da ima pri tem primeru požrešni algoritem 50 % možnosti, da ne pride do veljavnega razporeda.

Doslej smo naš primer predstavljali z grafom, v resnici pa potrebujemo konkretne širine in višine zvezkov in predalov. Delo si bomo olajšali z opažanjem, da je pri primeru, ki smo ga sestavili, vsak zvezek povezan z neko strnjeno skupino več zaporednih predalov, vsak predal pa z neko strnjeno skupino več zaporednih zvezkov. Predal p (za $1 \le p \le m$) naj ima širino p in višino $2m + 1 - p$; tako torej širine predalov naraščajo, višine pa padajo, pri čemer so vse višine večje od vseh širin (tega slednjega se bomo držali tudi pri zvezkih: višina vsakega zvezka bo večja od širine vsakega predala; tako se bomo izognili komplikacijam z možnostjo, da se sme zvezek zavrteti za 90 stopinj, kar naloga drugače sicer dopušča). Pri zvezku potem širina a pomeni, da so zanj dovolj široki le predali od a do m; višina $2m + 1 - b$ pa pri

zvezku pomeni, da so zanj dovolj visoki le predali od 1 do b. Zvezek teh dimenzij (če vzamemo $1 \le a \le b \le m$) gre torej lahko le v predale od a do b, v ostale pa ne. Tako lahko za vsak zvezek enostavno izberemo, v kateri „interval“ predalov (skupino več zaporednih predalov) lahko gre. Kot smo videli na sliki (4), imamo torej pri prvih treh zvezkih $a = 1$, $b = 3$; pri četrtem $a = 3$, $b = 4$; pri petem $a = 4$, $b = 7$; in pri zadnjih dveh $a = 5$, $b = 7$.

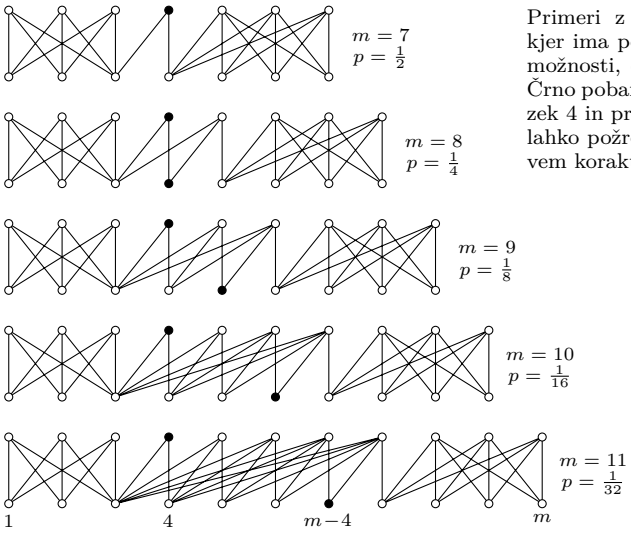
Priprava večjih primerov iz manjših. Zdaj smo torej na dva načina prišli do majhnih primerov, recimo z m zvezki in predali. Recimo še, da so njihove širine z območja od 1 do m, višine pa od $m + 1$ do $2m$ (kot pri primerih, ki smo jih zgoraj dobili z razmislekom; ampak tudi pri generiranju naključnih primerov ni težko paziti na to omejitev).

Naloga od nas zahteva primer z n zvezki in predali za neki večji n (vsaj 150). Pripravimo za začetek $k = \lfloor n/m \rfloor$ izvodov našega majhnega primera, pri čemer pa v i-tem izvodu povečajmo širine vseh zvezkov in predalov za $(i - 1) \cdot 2m$, njihove višine pa za $(k - i) \cdot 2m$. Tako bomo poskrbeli, da posamezni izvodi ne bodo vplivali drug na drugega. Če namreč poskusimo položiti zvezek iz i-tega izvoda v predal iz j-tega izvoda, bomo ugotovili, da je zvezek bodisi preširok (če je $j < i$) bodisi previsok (če je $j > i$). Če poskusimo tak zvezek prej še zasukati za 90 stopinj, pride sicer po dimenzijah blizu predalom iz j-tega izvoda za $j = k + 1 - i$, vendar še vedno ne bo šel vanje: pri naših predalih — pa tudi pri zvezkih, če jih ne zavrtimo za 90 stopinj — je namreč širina vedno iz spodnje polovice nekega intervala oblike $\{2mt + 1, \ldots, 2m(t + 1)\}$, višina pa iz zgornje polovice nekega takega intervala (ne nujno za isti t); pri zvezku, ki smo ga zavrteli za 90 stopinj, pa je ravno obratno: širina je iz zgornje polovice nekega takega intervala, zato bo tak zvezek preširok za predale iz j-tega izvoda.

Zdaj imamo torej k izvodov našega majhnega primera, ki ne vplivajo drug na drugega; to je skupaj $k \cdot m$ zvezkov in predalov. Do n, kolikor jih zahteva naloga, nam jih manjka še $r := n \bmod m$. Povečajmo širine in višine vseh dosedanjih zvezkov in predalov za eno enoto ter dodajmo še novih r zvezkov in r predalov širine 1 in višine 1000. Novi predali so preozki za vse prejšnje zvezke, novi zvezki pa previsoki za vse prejšnje predale, zato na delovanje požrešnega algoritma ne bodo vplivali (prisiljen bo razporediti nove zvezke v nove predale, njegove možnosti za uspešno razporeditev prvih $k \cdot m$ zvezkov v prvih $k \cdot m$ predalov pa se s tem ne bodo nič povečale).

Še boljša rešitev. Če si ogledamo še enkrat sliko na vrhu str. 147, lahko opazimo, da sta si primera za $m = 7$ (z verjetnostjo $p = 1/2$, da požrešni algoritem najde rešitev) in za $m = 8$ (s $p = 1/4$) precej podobna; in če bi iskanje majhnih naključnih primerov izvedli še za $m = 9$, bi lahko podoben primer našli tudi tam (s $p = 1/8$). To pa je že dovolj, da lahko opazimo vzorec, ki ga je moči posplošiti na poljuben $m \ge 7$, kot kaže slika na str. 149.

Tako kot prej naj bo p-ti predal širok p in visok $2m + 1 - p$, tako da lahko za vsak zvezek preprosto izberemo, v katere predale lahko gre: če hočemo, da gre zvezek z v predale od a_z do b_z (ne pa v ostale), mora biti širok a_z in visok $2m + 1 - b_z$. Vrednosti a_z in b_z si izberimo takole:



Primeri z m zvezki in m predali, kjer ima požrešni algoritem le 2^{6-m} možnosti, da najde pravilno rešitev. Črno pobarvani točki označujeta zvezek 4 in predal $m-4$, med katerima lahko požrešni algoritem izbira v prvem koraku.

---

z	1, 2, 3	$4, \ldots, m-3$	$m-2$, $m-1$, m
a_z	1	3	$m-3$
b_z	3	z	m

Kaj to pomeni za predale? Predal p lahko sprejme zvezke od α_p do β_p, drugih pa ne; te vrednosti kaže naslednja tabela:

p	1, 2	3	$4, \ldots, m-4$	$m-3$	$m-2$, $m-1$, m
α_p	1	1	p	$m-3$	$m-2$
β_p	3	$m-3$	$m-3$	m	m

Primeru, ki ga za m zvezkov in m predalov sestavimo po teh pravilih, recimo G_m. Razmislimo, kako deluje na njem požrešni algoritem iz naše naloge. Na začetku lahko izbira med zvezkom 4 in predalom $m-4$ (slednja možnost sicer pri $m = 7$ odpade), kajti tadva imata po dva kandidata, drugi zvezki in predali pa po tri ali več.

(1) Če se odloči za zvezek 4, ga lahko dá v predal 3 ali 4. (1.1) Če ga dá v predal 3, bosta potem za prve tri zvezke na voljo le prva dva predala, torej do rešitve gotovo ne bomo prišli. (1.2) Če pa ga dá v predal 4, ločimo dve možnosti. (1.2.1) Če je bil $m = 7$, nam zdaj ostaneta dve ločeni skupini: eno tvorijo prvi trije zvezki in prvi trije predali, drugo pa zadnji trije zvezki in zadnji trije predali. V vsaki skupini gre lahko vsak zvezek v vsak predal, zato se požrešni algoritem ne more zmotiti in bo gotovo našel veljavno rešitev. (1.2.2) Če pa je $m > 7$, lahko v mislih pobrišemo zvezek 4 in predal 4, tistim z višjimi številkami pa zmanjšamo številke za 1; to, kar imamo po tej spremembi pred seboj, pa je ravno primer G_{m-1} in je torej verjetnost, da požrešni algoritem zdaj pride do rešitve, točno taka, kot če začnemo reševati G_{m-1} od začetka.

(2) Če pa se požrešni algoritem v prvem koraku odloči za predal $m-4$ (poudarimo še enkrat, da je ta možnost na voljo le pri $m > 7$), lahko dá vanj zvezek $m-3$ ali

$m-4$. (2.1) Če dá vanj zvezek $m-3$, nam potem za zadnje štiri predale ostanejo le zadnji trije zvezki, torej do rešitve gotovo ne bomo prišli (ker mora na koncu vsak predal dobiti točno en zvezek). (2.2) Če pa dá vanj zvezek $m-4$, lahko potem v mislih pobrišemo zvezek $m-4$ in predal $m-4$ ter zmanjšamo številke zadnjih štirih zvezkov in predalov za 1; tako imamo pred seboj spet ravno primer G_{m-1}.

Povzemimo rezultate tega razmisleka; verjetnost, da požrešni algoritem na G_m najde pravo rešitev, označimo s p_m. Pri $m = 7$ smo videli, da imamo na prvem koraku na voljo le zvezek 4, pri čemer nas izbira predala 4 gotovo pripelje do rešitve, izbira predala 3 pa gotovo ne; tako imamo $p_7 = 1/2$. Pri $m > 7$ pa smo videli, da ne glede na to, ali v prvem koraku izberemo zvezek 4 ali predal $m-4$, imamo potem dve možnosti, od katerih nas ena zagotovo ne pripelje do rešitve, druga pa nas postavi pred problem G_{m-1}; ker se požrešni algoritem med tema možnostma odloči naključno (vsaka ima verjetnost $1/2$, da bo izbrana), imamo torej (za $m > 7$) $p_m = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot p_{m-1} = p_{m-1}/2$. Tako je torej $p_7 = 1/2$, $p_8 = 1/4$, $p_9 = 1/8$ in v splošnem $p_m = 2^{6-m}$. Ker bo naša naloga zahtevala primere z vsaj 150 zvezki in predali, je verjetnost, da bi požrešni algoritem pri posameznem testu našel veljaven razpored (in bi zato ocenjevalni strežnik zavrnil našo rešitev) kvečjemu $2^{-144} \approx 4{,}48 \cdot 10^{-44}$.

H. Vrivanje

Preden se lotimo rešitve, najprej nekaj opomb glede notacije. Dolžino niza x označimo z $|x|$, razen v zapisu z velikim $O(\cdot)$ ali pod korenom $\sqrt{\cdot}$, kjer bomo znaka $|\cdot|$ opuščali. Uporabljali bomo pythonovski zapis za posamezne znake in podnize. Niz x tvorijo znaki $x[0], \ldots, x[|x|-1]$; podniz $x[i{:}j]$ obsega znake $x[i], \ldots, x[j-1]$; če v tem zapisu i manjka, si mislimo $i = 0$; če j manjka, si mislimo $j = |x|$; če je i ali j negativen, mu v mislih prištejemo $|x|$. *Položaj* i v nizu x nam pomeni mejo med znakoma $x[i-1]$ in $x[i]$ (položaj 0 je levi rob prvega znaka, položaj $|x|$ pa desni rob zadnjega znaka). Zapis x^{R} nam bo pomenil niz, ki ga dobimo, če znake niza x preberemo od desne proti levi, torej $x^{\mathsf{R}}[i] = x[|x|-1-i]$; prefiksi in sufiksi nizov x in x^{R} so si seveda v tesnem sorodstvu: $x^{\mathsf{R}}[i{:}] = (x[{:}|x|-i])^{\mathsf{R}}$ in $x^{\mathsf{R}}[{:}i] = (x[|x|-i{:}])^{\mathsf{R}}$.

Pri reševanju te naloge nam bodo večkrat prišle prav tabele prefiksov, kot jih poznamo iz Knuth-Morris-Prattovega algoritma, zato si za začetek osvežimo spomin nanje. Za niz x definirajmo tabelo P_x, ki nam bo za vsak prefiks niza x povedala, na kako dolg krajši prefiks niza x se konča; z drugimi besedami, element $\mathsf{P}_x[i]$ vsebuje največje število $j < i$, za katero je $x[{:}j]$ sufiks niza $x[{:}i]$. To lahko poceni računamo po naraščajočih i: $x[{:}j]$ je neprazen sufiks niza $x[{:}i]$ natanko tedaj, ko je $x[{:}j-1]$ sufiks niza $x[{:}i-1]$ in je $x[j-1] = x[i-1]$. Pregledati moramo torej tiste sufikse niza $x[{:}i-1]$, ki so tudi njegovi prefiksi, in za vsakega preveriti, ali se ga dá podaljšati še za en znak, da bo prefiks in sufiks niza $x[{:}i]$. Ker nas zanima najdaljši tak, bomo kandidate pregledovali od daljših proti krajšim. Najdaljši kandidat je torej $x[{:}r]$ za $r = \mathsf{P}_x[i-1]$. Če se pri njem ne izide, kateri je potem drugi najdaljši kandidat? To bo torej neki $x[{:}u]$ (za $u < r$), na katerega se konča niz $x[{:}i-1]$; toda ker se slednji konča tudi na $x[{:}r]$, ki je daljši od $x[{:}u]$, to pomeni, da se tudi $x[{:}r]$ sam konča na $x[{:}u]$; in ker sta tadva niza oba prefiksa niza x in je drugi krajši od prvega, to pomeni, da se $x[{:}r]$ tudi začne na $x[{:}u]$. Slednji je torej hkrati prefiks in sufiks niza $x[{:}r]$; najdaljši tak pa je, kot vemo, dolg $\mathsf{P}_x[r]$ znakov, torej moramo vzeti $u = \mathsf{P}_x[r]$. Kandidati, ki jih moramo pregledati, so torej dolgi $\mathsf{P}_x[i-1]$, $\mathsf{P}_x[\mathsf{P}_x[i-1]]$ in tako

naprej.

```
P_x[0] := −1; P_x[1] := 0;
for i := 2 to |x|:
  r := P_x[i − 1];
  while true:
    if x[r] = x[i − 1] then P_x[i] := r + 1; break;
    else if r = 0 then P_x[i] := 0; break;
    else r := P_x[r];
```

V notranji zanki gledamo kandidata $x[:r]$, ki se pojavlja na koncu niza $x[:i-1]$; levi rob te pojavitve je na na položaju $i-1-r$ v nizu x in ta položaj se iz iteracije v iteracijo premika le v desno, nikoli v levo, tudi ko se i v zunanji zanki poveča za 1. Zato je časovna zahtevnost tega postopka le linearna, $O(x)$.

Recimo zdaj, da imamo poleg x še neki drug niz y; definirajmo potem še tabelo P_{yx}, ki nam bo za vsak prefiks niza y povedala, na kako dolg prefiks niza x se konča. Z drugimi besedami, $\mathsf{P}_{yx}[i]$ naj bo največje tako število $j < |x|$, za katero je $x[:j]$ sufiks niza $y[:i]$. Razmišljamo lahko zelo podobno kot zgoraj in dobimo zelo podoben postopek:

```
P_yx[0] := 0;
for i := 1 to |y|:
  r := P_yx[i − 1];
  while true:
    if r < |x| − 1 and x[r] = y[i − 1] then P_yx[i] := r + 1; break;
    else if r = 0 then P_yx[i] := 0; break;
    else r := P_yx[r];
```

Ta tabela je koristna na primer pri iskanju pojavitev niza x kot podniza v nizu y: te pojavitve se končajo na tistih položajih i (spomnimo se, da je položaj i meja med znakoma $y[i-1]$ in $y[i]$), kjer je $\mathsf{P}_{yx}[i-1] = |x|-1$ in $y[i-1] = x[-1]$. Tudi to bo koristno imeti v tabeli; naj bo torej $\mathsf{O}_{yx}[i] = 1$, če se x pojavlja v y z začetkom na položaju i (torej če je $y[i : i+|x|] = x$), sicer pa naj bo $\mathsf{O}_{yx}[i] = 0$. Definirajmo še tabelo delnih vsot: $\bar{\mathsf{O}}_{yx}[i] = \sum_{j=0}^{i} \mathsf{O}_{yx}[j]$, ki torej šteje vse pojavitve x kot podniza v y na položajih od 0 do i. Vse te tabele znamo torej izračunati v linearno mnogo časa.

Podobno kot doslej za prefikse lahko naredimo tudi za sufikse; vpeljimo tabeli S_x in S_{yx}, ki nam za vsak sufiks niza x oz. y povesta, kateri je najdaljši sufiks niza x, ki se pojavlja na začetku omenjenega sufiksa. Natančneje: naj bo $\mathsf{S}_x[i]$ najmanjša vrednost $j > i$, pri kateri je $x[j:]$ prefiks niza $x[i:]$; in naj bo $\mathsf{S}_{yx}[i]$ najmanjša vrednost $j > 0$, pri kateri je $x[j:]$ prefiks niza $y[i:]$. Obe tabeli lahko izračunamo po podobnem postopku kot P_x in P_{yx}, le da gremo po nizih od desne proti levi.[22]

Lotimo se zdaj naše naloge. Da jo rešimo, bomo za vsak $k = 0, \ldots, |s|$ prešteli pojavitve p-ja v nizu $s[:k]\,t\,s[k:]$, ki nastane, ko vrinemo t v s na položaju k. Te pojavitve lahko razdelimo na: (1) tiste, ki ležijo v celoti znotraj $s[:k]$ ali v celoti znotraj $s[k:]$; (2) tiste, ki ležijo v celoti znotraj t; (3) tiste, ki se začnejo v $s[:k]$ in

[22] Lahko pa tudi niza obrnemo in uporabimo dosedanja postopka za tabele prefiksov; velja namreč $\mathsf{S}_x[i] = |x| - \mathsf{P}_{x^\mathsf{R}}[|x|-i]$ in $\mathsf{S}_{yx}[i] = |x| - \mathsf{P}_{y^\mathsf{R}x^\mathsf{R}}[|y|-i]$.

končajo v t; (4) tiste, ki se začnejo v t in končajo v $s[k{:}]$; in (5) tiste, ki se začnejo v $s[{:}k]$ in končajo v $s[k{:}]$, vmes pa se raztezajo še čez celoten t. Vsako od teh petih skupin bomo prešteli posebej in rezultate na koncu sešteli. Skupina (2) pride seveda v poštev le, če je $|p| \le |t|$, skupina (5) pa, če je $|p| \ge |t| + 2$.

(1) Recimo, da se p pojavi na položaju i v s, torej da je $s[i : i + |p|] = p$. Ta pojavitev leži v celoti znotraj $s[{:}k]$, če je $i + |p| \le k$, torej $i \le k - |p|$; takih pojavitev je vsega skupaj $\bar{\mathsf{O}}_{sp}[k - |p|]$.

Če pa hočemo, da pojavitev leži v celoti znotraj $s[k{:}]$, to pomeni, da mora biti $i \ge k$; take pojavitve lahko preštejemo tako, da od števila *vseh* pojavitev p-ja v s odštejemo tiste, ki se začnejo nekje od 0 do $k-1$. Tako dobimo $\bar{\mathsf{O}}_{sp}[|s|] - \bar{\mathsf{O}}_{sp}[k-1]$.

(2) Drugo skupino tvorijo vse pojavitve p-ja v t; teh je torej $\bar{\mathsf{O}}_{tp}[|t|]$. Ta rezultat uporabimo pri vsakem k.

(3) Tretjo skupino tvorijo take pojavitve p-ja, ki se začnejo v $s[{:}k]$ in končajo v t. Recimo, da levih j znakov p-ja leži v $s[{:}k]$, preostanek pa v t; torej se $s[{:}k]$ konča na $p[{:}j]$, niz t pa se začne na $p[j{:}]$. Vemo že, da je najdaljši prefiks p-ja, ki se pojavlja na koncu $s[{:}k]$, tisti dolžine $i = \mathsf{P}_{sp}[k]$ znakov; za potrebe našega razmisleka lahko torej namesto $s[{:}k]$ uporabimo $p[{:}i]$.

Naj bo torej $D[i]$ število takih pojavitev p-ja v nizu $p[{:}i]\,t$, ki se začnejo že v prvem delu, torej v $p[{:}i]$. Ena možnost je, da stoji taka pojavitev že na začetku niza (v tem primeru se mora t začeti na $p[i{:}]$); druga možnost pa je, da se začne malo kasneje, tako da v $p[{:}i]$ leži le prvih $j < i$ znakov naše pojavitve. Torej se $p[{:}i]$ konča na $p[{:}j]$; in vemo že, da je prvi naslednji (največji) j, pri katerem je ta pogoj izpolnjen, $j = \mathsf{P}_p[i]$. Dobili smo torej naslednji postopek:

```
D[0] := 0;
for i := 1 to |p|:
  D[i] := D[P_p[i]]; (* pojavitve, ki imajo v p[:i] manj kot i znakov *)
  if se t začne na p[i:] then
    D[i] := D[i] + 1; (* pojavitev p na začetku niza p[:i] t *)
```

Tabelo D si pripravimo vnaprej (za kar porabimo linearno mnogo časa), nato pa pri vsakem k vemo, da se $D[\mathsf{P}_{sp}[k]]$ pojavitev p-ja začne v $s[{:}k]$ in konča v t.

Toda v gornjem postopku moramo znati hitro preverjati, ali se t začne na $p[i{:}]$ za dani i. Kako to naredimo? Pripravimo si vnaprej tabeli S_p in S_{tp}. Zdaj nam $j = \mathsf{S}_{tp}[0]$ pove, da je $p[j{:}]$ najdaljši sufiks p-ja, ki stoji na začetku t-ja; drugi najdaljši tak sufiks je potem $p[\mathsf{S}_p[j]{:}]$, tretji $p[\mathsf{S}_p[\mathsf{S}_p[j]]{:}]$ in tako naprej. Lahko si torej vnaprej pripravimo tabelo E, ki za vsak možni začetni indeks j pove, ali $p[j{:}]$ stoji na začetku t-ja:

```
for j := 1 to |p| do E[j] := false;
j := S_tp[0];
while j < |p|:
  E[j] := true; j := S_p[j];
```

Pogoj „ali se t začne na $p[i{:}]$“ v našem postopku za izračun tabele D lahko torej zdaj preverjamo tako, da pogledamo vrednost $E[i]$.

(4) Četrto skupino tvorijo pojavitve p-ja, ki se začnejo v t in končajo v $s[k{:}]$. To lahko rešujemo po prav takem postopku kot (3), le da prej v mislih obrnemo vse tri nize od desne proti levi.

(5) Ostanejo nam še tiste pojavitve p-ja, ki se začnejo v $s[:k]$ in končajo v $s[k:]$, vmes pa seveda vsebujejo še celoten t. To pomeni, da se mora $s[:k]$ končati na neki prefiks p-ja in vemo že, da je najdaljši tak prefiks dolg $i := \mathsf{P}_{sp}[k]$ znakov; in podobno se mora $s[k:]$ začeti na neki sufiks p-ja, najdaljši tak sufiks pa je dolg $|p| - j$ znakov za $j := \mathsf{S}_{sp}[k]$. Vidimo torej, da lahko v nadaljevanju našega razmisleka namesto nizov $s[:k]$ in $s[k:]$ gledamo le $p[:i]$ in $p[j:]$. (Ta i in j sta seveda pri različnih k-jih lahko različna in ju bomo kasneje, kjer bo to relevantno, pisali kot i_k in j_k.) Ogledali si bomo več načinov, kako lahko preštejemo pojavitve p v nizih oblike $p[:i]\,t\,p[j:]$, od preprostejših in manj učinkovitih do učinkovitejših, a bolj zapletenih. Vse, kar smo doslej počeli pri skupinah od (1) do (4), je imelo linearno časovno in prostorsko zahtevnost, $O(s+t+p)$, zato bo zahtevnost rešitve kot celote odvisna predvsem od tega, kako bomo zdaj rešili nalogo za skupino (5).

Kvadratna rešitev. Naj bo torej $f(i,j)$ število pojavitev p-ja v nizu $p[:i]\,t\,p[j:]$, seveda le takih, ki se začnejo že v $p[:i]$ in končajo šele v $p[j:]$. Ta funkcija je zanimiva za $0 \le i < |p|$ in $0 < j \le |p|$ (da bosta prvi in tretji del, $p[:i]$ in $p[j:]$, krajša od p). Robni primeri so $p(0,j) = p(i,|p|) = 0$, kajti takrat je prvi ali tretji del prazen in se torej pojavitev p-ja ne more prekrivati z njim. Razmislimo zdaj o splošnem primeru.

Ena pojavitev p v $p[:i]\,t\,p[j:]$ stoji morda že na začetku tega niza; ostale pa se morajo začeti kasneje, torej imajo znotraj prvega dela niza, $p[:i]$, le neki krajši prefiks, recimo $p[:i']$, ki je torej tudi sufiks niza $p[:i]$. Najdaljši tak prefiks pa je, kot vemo, dolg $i' := \mathsf{P}_p[i]$ znakov; vse te kasnejše pojavitve torej ležijo ne le znotraj $p[:i]\,t\,p[j:]$, pač pa tudi znotraj $p[:i']\,t\,p[:j]$, torej jih je $f(i',j)$. Tako smo dobili naslednji postopek za računanje vseh možnih vrednosti funkcije f z dinamičnim programiranjem:

```
1   for i := 1 to |p| − 1 do for j := 1 to |p| − 1:
2     f[i, j] := 0;
      (* Ali se p pojavlja na začetku niza p[:i] t p[j:]? *)
3     if se t pojavlja v p na mestu i (torej če je p[i : i + |t|] = t) then
4       if je i + |t| < |p| in se p[j:] začne na p[i + |t|:] then
5         f[i, j] := 1;
      (* Prištejmo še kasnejše pojavitve. *)
6     i' := P_p[i];
7     f[i, j] := f[i, j] + f[i', j];
```

Preden bo gornji postopek za izračun funkcije f res uporaben, moramo premisliti še o nekaj podrobnostih v njem. Pogoj v vrstici 3 preverimo tako, da pogledamo, če je $\mathsf{O}_{pt}[i] = 1$.

V vrstici 4 smo najprej s pogojem $i + |t| < |p|$ preverili, ali bi pojavitev p-ja, ki bi se začela na začetku niza $p[:i]\,t\,p[j:]$, sploh dosegla tisti $p[j:]$ na koncu, kajti zanimajo nas le take pojavitve — tiste, ki ležijo v celoti znotraj $p[:i]\,t$, smo namreč šteli že pri skupini (3). Nato pa moramo v vrstici 4 preveriti, ali se neki sufiks p-ja začne z nekim krajšim sufiksom p-ja. Spomnimo se, da je najdaljši $p[j':]$, ki stoji na začetku niza $p[j:]$, tisti za $j' = \mathsf{S}_p[j]$; naslednji je potem $p[j'':]$ za $j'' = \mathsf{S}_p[j']$ in tako naprej. V vrstici 4 se torej pravzaprav sprašujemo, ali to zaporedje vse krajših sufiksov sčasoma pride tudi do $p[i + |t|:]$. Pri tem si lahko pomagamo z drevesom,

kakršno bomo, kot bomo videli, tako ali tako potrebovali tudi v nadaljevanju našega razmisleka. Zgradimo torej drevo T_S, v katerem je po eno vozlišče za vsak u od 1 do $|p|$; koren drevesa je vozlišče $|p|$ (spomnimo se, da nam številka pri sufiksu predstavlja indeks, na katerem se ta sufiks začne; krajši sufiksi imajo torej večje indekse, največji med njimi pa je $|p|$, ki predstavlja prazen sufiks), za vsak $u < |p|$ pa naj bo vozlišče u otrok vozlišča $\mathsf{S}_p[u]$. Vrstica 4 zdaj pravzaprav sprašuje, ali je v drevesu T_S vozlišče $i + |t|$ prednik vozlišča j.

To lahko učinkovito preverjamo, če si pripravimo premi vrstni red (*preorder*) vseh točk v drevesu — to je tak vrstni red, v katerem najprej navedemo koren, nato pa rekurzivno dodamo premi vrstni red vsakega od njegovih poddreves. Za vsako točko u si zapomnimo njen položaj v tem vrstnem redu (recimo $\sigma_\mathsf{S}[u]$) in položaj zadnjega izmed njenih potomcev (temu recimo $\tau_\mathsf{S}[u]$). Potem so potomci u-ja natanko tiste točke, ki so v premem vrstnem redu na indeksih od $\sigma_\mathsf{S}[u]$ do $\tau_\mathsf{S}[u]$.

podprogram PreglejPoddrevo(drevo T, točka u, tabeli σ in τ, indeks i):
  $i := i + 1$; $\sigma[u] := i$;
  za vsakega otroka v točke u v drevesu T:
    $i :=$ PreglejPoddrevo(T, v, σ, τ, i);
  $\tau[u] := i$; **return** i;

podprogram PremiVrstniRed(vhod: drevo T; izhod: tabeli σ in τ):
  naj bosta σ in τ tabeli z indeksi od 1 do $|p|$;
  PreglejPoddrevo(T, koren T-ja, $\sigma, \tau, 0$);
  return σ, τ;

glavni klic: $\sigma_\mathsf{S}, \tau_\mathsf{S} :=$ PremiVrstniRed(T_S);

Da preverimo, ali je neki $p[\ell{:}]$ prefiks niza $p[j{:}]$, moramo torej le preveriti, ali je $\sigma_\mathsf{S}[\ell] \le \sigma_\mathsf{S}[j] \le \tau_\mathsf{S}[\ell]$.[23]

Naš dosedanji postopek je izračunal $f(i, j)$ tako, da je preveril, ali se p pojavlja na začetku niza $p[{:}i]\,t\,p[j{:}]$, kasnejše pojavitve pa je dobil iz $f(i', j)$ za $i' = \mathsf{P}_p[i]$. Lahko bi šli seveda tudi z druge strani: najprej bi preverili, ali se p pojavlja na koncu (namesto na začetku) niza $p[{:}i]\,t\,p[j{:}]$, nato pa bi prišteli še zgodnejše pojavitve, ki jih je $f(i, j')$ za $j' = \mathsf{S}_p[j]$. Da preverimo, ali se p pojavlja na koncu $p[{:}i]\,t\,p[j{:}]$, bi morali najprej preveriti, ali se t pojavlja v p na indeksih od $j - |t|$ do j (z enako tabelo kot že prej v vrstici 3), nato pa bi nas zanimalo, ali se $p[{:}i]$ konča na $p[{:}j - |t|]$. Tu bi bilo dobro zgraditi drevo T_P, ki bi imelo po eno vozlišče za vsak u od 0 do $|p| - 1$; vozlišče 0 bi bilo koren, za vsak $u > 0$ pa naj bo vozlišče u otrok vozlišča $\mathsf{P}_p[u]$. Za to drevo bi s pravkar opisanim postopkom PremiVrstniRed izračunali tabeli σ_P in τ_P. V tem drevesu bi morali potem preveriti, ali je $j - |t|$ prednik vozlišča i.

[23] Še en način, kako preveriti, ali je $p[\ell{:}]$ prefiks niza $p[j{:}]$, pa je z Z-algoritmom (z njim smo se že srečali lani; gl. *Bilten* 2021, str. 148 in tam navedeno literaturo). Za poljuben niz w lahko v $O(w)$ časa pripravimo tabelo Z_w, v kateri za vsak $i = 0, \ldots, |w|$ element $\mathsf{Z}_w[i]$ pove dolžino najdaljšega skupnega prefiksa nizov w in $w[i{:}]$. Vprašanje, ki nas zanima — torej ali je $p[\ell{:}]$ prefiks niza $p[j{:}]$ — je enakovredno vprašanju, ali je $p[\ell{:}] = p[j : j + |p| - \ell]$, to pa vprašanju, ali je $p^\mathsf{R}[{:}|p| - \ell] = p^\mathsf{R}[\ell - j : |p| - j]$, to pa vprašanju, ali se p^R in $p^\mathsf{R}[\ell - j{:}]$ ujemata v prvih $|p| - \ell$ znakih, torej ali je $\mathsf{Z}_{p^\mathsf{R}}[\ell - j] \ge |p| - \ell$. Za naš namen je sicer rešitev z drevesom koristnejša od rešitve z Z-algoritmom, ker bo prišlo drevo prav tudi kasneje pri boljših rešitvah od te s kvadratno časovno zahtevnostjo, ki jo opisujemo zdaj.

Tako torej vidimo, da lahko $f(i,j)$ hitro, v $O(1)$ časa, izračunamo bodisi iz $f(\mathsf{P}_p[i],j)$ bodisi iz $f(i,\mathsf{S}_p[j])$; v nadaljevanju nam bo prišlo prav oboje.

Naš dosedanji postopek za izračun funkcije f bi porabil $O(p)$ časa za pripravo dreves T_P in T_S ter tabel σ_P, σ_S, τ_P in τ_S, potem pa $O(p^2)$ časa za izračun vrednosti $f(i,j)$ za vse možne pare (i,j). Tako imamo rešitev s časovno zahtevnostjo $O(s+t+p^2)$, kar pa je za našo nalogo žal prepočasi, saj je lahko p dolg do 10^5 znakov.

Rešitev v času $O((s+p)\sqrt{p})$. Vrednosti $f(i,j)$, ki smo jih izračunali pri kvadratni rešitvi, v resnici ne potrebujemo za *vse* pare (i,j), pač pa le za po en tak par (i_k,j_k) pri vsakem k (torej vsakem možnem položaju, kamor lahko vrinemo p v s), torej le $O(s)$ vrednosti in ne vseh $O(p^2)$. Kako bi lahko izračunali tiste vrednosti funkcije f, ki jih res potrebujemo, ne pa ob tem še vseh ostalih?

Sprehodimo se po drevesu T_P od spodaj navzgor in računajmo velikosti poddreves; kadarkoli pri nekem vozlišču u opazimo, da ima njegovo poddrevo (recimo mu T_u; to je torej del drevesa, ki ga tvorijo u in vsi njegovi potomci) vsaj $\sqrt{p}$ vozlišč, označimo u in v mislih izrežimo poddrevo T_u iz drevesa. To, da ga izrežemo, pomeni, da vozlišč v njem ne bomo šteli, ko bomo kasneje razmišljali o velikosti poddreves u-jevih prednikov višje gor v drevesu. Ko nazadnje pridemo do korena, označimo tudi njega, ne glede na to, koliko vozlišč je takrat še ostalo v drevesu. Naslednji postopek izračuna za vsako vozlišče u njegovega najbližjega označenega prednika $M[u]$; če je u že sam označen, bo $M[u]=u$.

podprogram Označi(u):
  (* *V spremenljivki N računamo število vozlišč v u-jevem poddrevesu, razen tistih, do katerih se iz u pride skozi označene potomce.* *)
  $N := 1$;
  za vsakega u-jevega otroka v:
    $N := N +$ Označi(v);
  if $N \geq \sqrt{p}$ **or** $u = 0$
  then $M[u] := u$; **return** 0; (* *Označimo u.* *)
  else $M[u] := -1$; **return** N; (* *Sicer bo u neoznačen.* *)

glavni blok:
  Označi(0); (* *Začnemo v vozlišču 0, ki je koren drevesa.* *)
  (* *Podatke o najbližjem označenem predniku prenesimo dol po drevesu (s staršev na otroke) še na vsa neoznačena vozlišča.* *)
  for $u := 1$ **to** $|p|-1$ **do if** $M[u] < 0$ **then** $M[u] := M[\mathsf{P}_p[u]]$;

Zdaj imamo torej označenih kvečjemu $\sqrt{p}$ vozlišč (ker smo vsakič, ko smo označili eno vozlišče, tudi izrezali vsaj $\sqrt{p}$ vozlišč iz drevesa, vseh vozlišč skupaj pa je $|p|$); in vsako vozlišče je oddaljeno manj kot $\sqrt{p}$ korakov od svojega najbližjega označenega prednika (kajti če bi bilo oddaljeno vsaj $\sqrt{p}$ korakov, bi se že pri nekem nižje ležečem predniku tega vozlišča moralo izkazati, da ima njegovo poddrevo vsaj $\sqrt{p}$ vozlišč, in bi torej označili že tega nižje ležečega prednika in izrezali njegovo poddrevo).

Vemo, da je $f(i,|p|)=0$ za vsak i. Nato lahko za vsak označen i izračunamo iz $f(i,|p|)$ tudi vse vrednosti $f(i,j)$ za $1 \leq j < |p|$; ker je označenih vozlišč kvečjemu $\sqrt{p}$, vzame to $O(p\sqrt{p})$ časa. Zdaj za vsako od $O(s)$ poizvedb (i_k,j_k), ki nas v resnici zanimajo, razmišljajmo takole: najbližji označeni prednik vozlišča i_k je $M[i_k]$; ker je označen, zanj že poznamo $f(M[i_k],j)$ za vse j, torej tudi za $j=j_k$; in ker leži

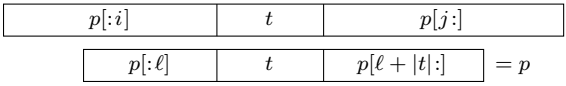
i_k v drevesu T_{P} največ $O(\sqrt{p})$ korakov pod vozliščem $M[i_k]$, lahko iz $f(M[i_k], j_k)$ v največ $O(\sqrt{p})$ korakih izračunamo $f(i_k, j_k)$. Ker moramo to narediti za vsako poizvedbo, porabimo skupno $O(s\sqrt{p})$ časa. Da prihranimo pri porabi prostora, bomo poizvedbe reševali v skupinah glede na $M[i_k]$; ko obdelamo vse poizvedbe z istim vozliščem $M[i_k]$, lahko rezultate $f(M[i_k], j)$ pozabimo.

Zapišimo dobljeno rešitev podproblema (5) še s psevdokodo:

```
(* Razvrstimo poizvedbe (i_k, j_k) glede na najbližjega označenega prednika. *)
for u := 0 to |p| − 1 do if M[u] = u then L[u] := prazen seznam;
for k := 1 to |s| − 1 do dodaj k v L[M[i_k]];

(* Pojdi po označenih točkah u. *)
for u := 0 to |p| − 1 do if M[u] = u:
  (* V F[j] izračunaj f(u, j) za vse j. *)
  F[|p|] := 0;
  for j := |p| − 1 downto 1 do v F[j] zapiši vrednost f(u, j), ki jo izračunaj
                               iz f(u, S_p[j]), ki je zapisana v F[S_p[j]];

  (* Odgovori na vse tiste poizvedbe (i_k, j_k), kjer je i_k-jev najbližji
     označeni prednik ravno u. *)
  za vsak k iz seznama L[u]:

    (* Pripravimo si pot v drevesu od i_k do njegovega prednika u. *)
    S := prazen sklad; i := i_k;
    while i ≠ u: dodaj i na vrh S in priredi i := P_p[i];

    (* Računajmo f(i, j_k) za vse točke i na poti od u do i_k. *)
    r := F[j_k]; (* to je f(u, j_k) *)
    while S ni prazen:
      (* i je starš točke, ki je trenutno na vrhu sklada S. *)
      i := točka na vrhu S; pobriši jo iz S;
      iz vrednosti f(P_p[i], j_k), ki je trenutno zapisana v r,
        izračunaj f(i, j_k) in to potem zapiši v r;

    (* r je zdaj enak f(i_k, j_k), torej odgovor na poizvedbo k; to je torej
       število takih pojavitev p-ja v s[:k] t s[k:], ki se začnejo v s[:k]
       in končajo v s[k:]. *)
```

Vse, kar smo počeli pri podproblemih od (1) do (4), je imelo linearno časovno zahtevnost v odvisnosti od dolžine vhodnih nizov; vsega skupaj je torej časovna zahtevnost naše rešitve $O(t + (p + s)\sqrt{p})$, prostorska pa $O(s + t + p)$. (Ta časovna zahtevnost je sicer v primerih, ko je $|s| \gg |p|\sqrt{p}$, lahko celo slabša kot pri kvadratni rešitvi.) Ta rešitev je za potrebe našega tekmovanja že dovolj hitra, vseeno pa si bomo ogledali še dve boljši.

Rešitev v času $O((s + p) \log p)$. Prešteti hočemo tiste pojavitve p-ja v nizu $p[:i]\,t\,p[j:]$, ki se začnejo že v $p[:i]$ in končajo šele v $p[j:]$. Taka pojavitev je torej oblike $p = p[:\ell]\,t\,p[\ell + |t|:]$, pri čemer mora biti $0 < \ell \le i$ in $j \le \ell + |t| < |p|$, veljati pa morajo naslednji trije pogoji: (a) niz $p[:\ell]$ mora biti sufiks niza $p[:i]$, (b) niz $p[\ell + |t|:]$ mora biti prefiks niza $p[j:]$, (c) niz t pa se mora pojavljati v p-ju kot podniz z začetkom na indeksu ℓ, torej $t = p[\ell : \ell + |t|]$.



Za preverjanje (c) si je koristno vnaprej pripraviti tabelo O_{pt} in preverjati, ali je $\mathsf{O}_{pt}[\ell] = 1$.

Razmislimo zdaj o pogoju (a), torej da mora biti $p[:\ell]$ sufiks niza $p[:i]$. Vemo, da je najdaljši prefiks p-ja, ki je tudi sufiks niza $p[:i]$, dolg $\mathsf{P}_p[i]$ znakov; drugi najdaljši je zato dolg $\mathsf{P}_p[\mathsf{P}_p[i]]$ znakov in tako naprej. Pogoj, da mora biti $p[:\ell]$ sufiks niza $p[:i]$, torej pravzaprav sprašuje o tem, ali se ℓ pojavi nekje v zaporedju $i, \mathsf{P}_p[i], \mathsf{P}_p[\mathsf{P}_p[i]], \ldots$. To pa je enakovredno vprašanju, ali je ℓ prednik i-ja v drevesu T_P, kar lahko, kot smo že videli, preverjamo s pogojem $\sigma_\mathsf{P}[\ell] \le \sigma_\mathsf{P}[i] \le \tau_\mathsf{P}[\ell]$.

Podobno lahko razmišljamo tudi pri pogoju (b), torej da mora biti $p[\ell + |t|:]$ prefiks niza $p[j:]$, le da zdaj namesto tabele P_p in drevesa T_P uporabimo tabelo S_p in drevo T_S. Pogoj (b) potem pravzaprav sprašuje, ali $\ell + |t|$ prednik j-ja v drevesu T_S, torej ali je $\sigma_\mathsf{S}[\ell + |t|] \le \sigma_\mathsf{S}[j] \le \tau_\mathsf{S}[\ell + |t|]$.

Tako smo dobili dve neenačbi, ki si ju lahko nazorno predstavljamo z geometrijsko interpretacijo: preverjamo pravzaprav, ali točka $Q_k := (\sigma_\mathsf{P}[i_k], \sigma_\mathsf{S}[j_k])$ leži v pravokotniku $R_\ell := [\sigma_\mathsf{P}[\ell], \tau_\mathsf{P}[\ell]] \times [\sigma_\mathsf{S}[\ell + |t|], \tau_\mathsf{S}[\ell + |t|]]$. Ali, bolje rečeno, zanima nas, v koliko takih pravokotnikih leži točka Q_k; pri tem pridejo v poštev pravokotniki R_ℓ za vse tiste ℓ, kjer se t pojavlja kot podniz v p (torej kjer velja $p[\ell:\ell+|t|] = t$ oz. $\mathsf{O}_{pt}[\ell] = 1$) — s tem, ko smo se omejili na takšne ℓ, smo poskrbeli tudi za pogoj (c). To nas bo zanimalo za vse k od 1 do $|s| - 1$; imamo torej $O(s)$ točk in za vsako od njih nas zanima, v koliko izmed $O(p)$ pravokotnikih leži.

Tak geometrijski problem pa lahko učinkovito rešimo z dobro znanim prijemom iz računske geometrije — s preletom ravnine (*plane sweep*). Pomikajmo se v mislih po ravnini z navpično premico od leve proti desni (x-koordinate — torej vrednosti iz tabele σ_P — gredo pri nas od 1 do $|p|$) in vzdržujmo neko podatkovno strukturo s podatki o tem, katere intervale y-koordinat pokrivajo tisti pravokotniki, ki so prisotni na trenutni x-koordinati. Ko dosežemo pri preletu ravnine levi rob nekega pravokotnika, ga moramo v to podatkovno strukturo dodati, ko dosežemo njegov desni rob, pa ga spet pobrišemo iz nje. Ker gredo tudi y-koordinate (torej vrednosti iz tabele σ_S) pri nas od 1 do $|p|$, je primerna podatkovna struktura na primer Fenwickovo drevo. Predstavljajmo si tabelo h, v kateri element $h[y]$ pove razliko med številom pravokotnikov (izmed tistih, ki so prisotni na trenutni x-koordinati), ki imajo spodnji rob pri y, in tistih, ki imajo zgornji rob pri $y - 1$. Potem je vsota $\sum_{z=0}^{y} h[y]$ ravno število vseh pravokotnikov, ki (pri trenutnem x) pokrivajo točko (x, y). Fenwickovo drevo nam omogoča računati takšne vsote v $O(\log p)$ časa, prav toliko pa stane tudi popravljanje drevesa, ko se kakšna od vrednosti $h[y]$ spremeni.[24] Zapišimo postopek preleta ravnine še s psevdokodo:

naj bo F Fenwickovo drevo, kjer so sprva vse vrednosti 0;
for $x := 1$ **to** $|p|$:
  za vsak pravokotnik $R_\ell = [x_1, x_2] \times [y_1, y_2]$, ki ima levi rob pri $x_1 = x$:
    v F povečaj $h[y_1]$ za 1 in zmanjšaj $h[y_2 + 1]$ za 1;

[24] V podrobnosti o Fenwickovem drevesu se tu ne bomo spuščali, saj je dobro znana podatkovna struktura. Na naših tekmovanjih smo jo že srečali na primer pri nalogi 2013.3.5 (gl. *Bilten* 2013, str. 72).

za vsako točko $Q_k = (x_k, y_k)$, ki ima $x_k = x$:
v F izračunaj vsoto $h[1] + \ldots + h[y_k]$; to je število takih pojavitev p v $s[:k]\,t\,s[k:]$, ki se začnejo v $s[:k]$ in končajo v $s[k:]$;
za vsak pravokotnik $R_\ell = [x_1, x_2] \times [y_1, y_2]$, ki ima desni rob pri $x_2 = x$:
v F zmanjšaj $h[y_1]$ za 1 in povečaj $h[y_2 + 1]$ za 1;

Pred tem si moramo seveda za vsak x pripraviti seznam pravokotnikov, ki imajo levi rob pri x, seznam pravokotnikov, ki imajo desni rob pri x, in seznam točk Q_k s to x-koordinato. Priprava teh seznamov nam vzame $O(p+s)$ časa; vsaka operacija na F nam vzame $O(\log p)$ časa, te operacije pa so: $O(p)$ dodajanj pravokotnikov, prav toliko brisanj in $O(s)$ računanj vsot. Če prištejemo še vse, kar smo videli v točkah od (1) do (4), je skupna časovna zahtevnost naše rešitve $O(t + (p+s)\log p)$. Če namesto Fenwickovega drevesa uporabimo korensko dekompozicijo na tabeli h (torej razdelimo tabelo h na približno $\sqrt{p}$ blokov s po $\sqrt{p}$ elementi in vzdržujemo vsoto vsakega bloka), se nam implementacija rešitve malo poenostavi, časovna zahtevnost pa naraste na $O(t + (p+s)\sqrt{p})$, kar je za naše tekmovanje še vedno dovolj hitro.

Linearna rešitev. Za konec omenimo še, da lahko nalogo rešimo v linearnem času, $O(s+p+t)$, če malo prilagodimo nedavno objavljeni algoritem M. Ganardija in P. Gawrychowskega.[25] Podrobnosti te rešitve bomo predvidoma objavili v enem od prihodnjih biltenov, najbrž na koncu razdelka z rešitvami neuporabljenih nalog.

I. Pranje denarja

Podatke o lastništvu podjetij lahko predstavimo z grafom, v katerem je po ena točka za vsakega človeka in za vsako podjetje, usmerjena povezava $u \to v$ pa pove, da je v eden od (neposrednih) lastnikov podjetja u. Podjetja prepoznamo po tem, da imajo njihove točke vsaj eno izhodno povezavo, ljudi pa po tem, da nimajo nobene izhodne povezave. Naloga pravi, da imamo c podjetij in p ljudi, obojih skupaj je torej $n := c + p$; oštevilčimo točke našega grafa tako, da dobijo podjetja številke od 1 do c, ljudje pa od $c+1$ do n.

Dobički se v našem grafu vedno prenašajo v smeri povezav, od podjetja k njegovim lastnikom. Če bi bil graf acikličen, bi bilo mogoče to prenašanje dobičkov zelo preprosto simulirati. Točke grafa bi morali pregledovati v topološkem vrstnem redu (torej takem, pri katerem pride začetno krajišče vsake povezave pred njenim končnim krajiščem), pa bi lahko pri vsaki prenesli njen dobiček na njene lastnike. Topološki vrstni red bi nam zagotovil, da ko pridemo do neke točke, je do nje prišel že ves dobiček, ki jo sploh lahko doseže (ker smo dotlej obdelali že vse točke, iz katerih je ona dosegljiva).

Iz besedila naloge vidimo, da sicer graf ni nujno acikličen, pač pa so lahko cikli le taki, da vse točke cikla pripadajo isti gospodarski panogi; to pa pomeni, da so cikli kratki, kajti v posamezni gospodarski panogi je lahko največ devet podjetij. Poiščimo torej v našem grafu krepko povezane komponente (zanje vemo, da obsegajo po največ devet točk), nato pa topološko uredimo te komponente, tako da bo za vsako

[25] M. Ganardi, P. Gawrychowski, "Pattern matching on grammar-compressed strings in linear time". *Proc. of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA22), str. 2833–46; algoritem, ki ga potrebujemo, je v izreku 3.2, le malenkost je treba prilagoditi, da bo pojavitve p-ja štel namesto le preverjal, ali kakšna pojavitev sploh obstaja.

povezavo, ki ima krajišči v različnih komponentah, veljalo, da stoji komponenta, ki jih pripada začetno krajišče, v topološkem vrstnem redu pred tisto, ki ji pripada končno krajišče.

V tem vrstnem redu moramo torej obravnavati komponente, da bomo lahko prenašali dobičke med podjetji iz različnih komponent; kaj pa se zgodi znotraj posamezne komponente? Recimo, da neko komponento tvorijo podjetja $u_1, \ldots, u_t$; naj bo $A = (a_{ij})$ matrika velikosti $t \times t$, v kateri je a_{ij} delež dobička, ki ga podjetje u_j pošilja svojemu lastniku u_i, če oba pripadata trenutni komponenti. Poleg tega naj bo matrika $B = (b_{ij})$ matrika velikosti $n \times t$, v kateri je b_{ij} delež dobička, ki ga podjetje u_j pošilja svojemu lastniku i, če le-ta ne pripada trenutni komponenti (sicer pa naj bo $b_{ij} = 0$).

Recimo zdaj, da imamo pred seboj vektor $\mathbf{x} = (x_1, \ldots, x_t)^\top$, v katerem nam x_j pove, koliko denarja je prišlo v podjetje u_j iz podjetij, ki jim je u_j posredni ali neposredni lastnik (in ki ležijo v komponentah, ki so v topološkem vrstnem redu pred trenutno). Prenašanje dobička znotraj komponente si lahko predstavljamo kot iterativni proces. V prvi iteraciji dobi vsako podjetje u_j na vhodu x_j denarja in ga pošlje $a_{ij}x_j$ vsakemu svojemu lastniku u_i v isti komponenti, poleg tega pa pošlje $b_{ij}x_j$ vsakemu svojemu lastniku i iz drugih komponent. Vektor $A\mathbf{x}$ torej pove, koliko denarja dobijo podjetja trenutne komponente na vhodu iz naslova delitve dobička v prvi iteraciji, vektor $B\mathbf{x}$ pa, koliko pri tem dobijo podjetja zunaj trenutne komponente.

V drugi iteraciji razmišljamo popolnoma enako, le da prejete zneske po podjetjih naše komponente zdaj opisuje vektor $A\mathbf{x}$ namesto vektorja $\mathbf{x}$. V drugi iteraciji torej podjetja komponente pošljejo drugim podjetjem svoje komponente vektor $A(A\mathbf{x}) = A^2\mathbf{x}$, svojim lastnikom v drugih komponentah pa vektor $B(A\mathbf{x}) = (BA)\mathbf{x}$.

Če tako razmišljamo naprej, dobijo lastniki zunaj komponente v tretji iteraciji $(BA^2)\mathbf{x}$, v četrti $(BA^3)\mathbf{x}$ in tako naprej. Skupna vsota prejetih zneskov je torej $BS\mathbf{x}$ za $S = I + A + A^2 + \ldots$. Pri vsoti S lahko opazimo, da če jo pomnožimo z A, dobimo $A + A^2 + A^3 + \ldots$, to pa je ravno ista vsota brez prvega člena: velja torej $SA = S - I$, torej $S(I - A) = I$, torej $S = (I - A)^{-1}$. Ker so te matrike majhne, reda $t \times t$, lahko S brez težav poiščemo npr. z Gaussovo eliminacijsko metodo (rešujemo sistem linearnih enačb $S(I - A) = I$).

(Še en način za izračun dovolj dobrega približka matrike S pa je naslednji. Definirajmo dvakrat večjo matriko

$$M = \begin{bmatrix} A & I \\ 0 & I \end{bmatrix}.$$

Zanjo se ni težko z indukcijo po k prepričati, da velja:

$$M^k = \begin{bmatrix} A^k & I + A + \ldots + A^{k-1} \\ 0 & I \end{bmatrix}.$$

Zgornja desna četrtina matrike M^k je torej približek S-ja, ki nastane, če od S-jeve neskončne vsote uporabimo le prvih k členov. Z zaporednim kvadriranjem lahko po vrsti računamo M^2, M^4, M^8 in tako naprej, ustavimo pa se, ko je zgornja desna četrtina matrike M^{2k} dovolj podobna tisti pri M^k; tisto mora biti potem dober približek S-ja.)

Pri izračunu $BS\mathbf{x}$ potem najprej izračunajmo zmnožek $\mathbf{y} := S\mathbf{x}$ (kar je spet vektor s t elementi), nato pa pri množenju z B upoštevajmo, da ima B le toliko neničelnih elementov, kolikor je povezav, ki kažejo iz točk trenutne komponente v točke zunaj nje; za vsako tako povezavo, recimo $u_j \to i$ (ki predstavlja lastniški delež b_{ij}), moramo potem točki i poslati $b_{ij}y_j$ denarja.

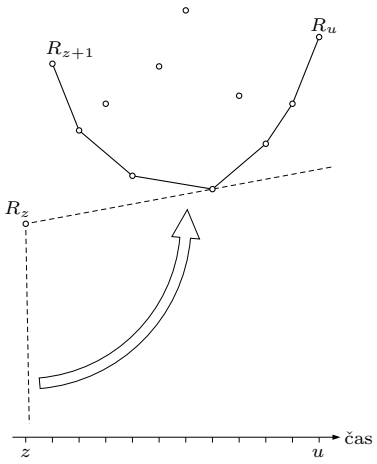
V praksi sicer vektor $\mathbf{x}$ ne bo en sam, pač pa nas bo hkrati zanimalo p različnih scenarijev: za vsako podjetje i od 1 do p nas zanima scenarij, v katerem to podjetje dobi 1 enoto dobička in se ta potem pretaka naprej po grafu v smeri povezav. Namesto stolpčnega vektorja $\mathbf{x}$ bomo imeli torej v resnici matriko X s p stolpci.

poišči v grafu krepko povezane komponente;
uredi jih v topološki vrstni red;
$X :=$ matrika reda $n \times p$, ki ima na diagonali enice, drugod ničle;
(* *Vrednost x_{ij} pove, koliko denarja ima trenutno točka i v scenariju, ki se je začel s tem, da je podjetje j ustvarilo 1 enoto dobička.* *)
za vsako komponento v topološkem vrstnem redu:
  recimo, da tvorijo to komponento točke $u_1, \ldots, u_t$;
  $A :=$ matrika reda $t \times t$, ki ima povsod ničle;
  za vsako točko u_j te komponente in za vsako njeno izhodno povezavo $u_j \to v$,
  ki predstavlja recimo lastniški delež d:
    če je tudi v v tej komponenti, recimo $v = u_i$:
      $a_{ij} := d$;
  z Gaussovo eliminacijsko metodo izračunaj $S := (I - A)^{-1}$;
  $Y :=$ matrika reda $t \times p$;
  for $i := 1$ **to** t **do for** $k := 1$ **to** p **do** $y_{ik} := \sum_{j=1}^{t} s_{ij}x_{jk}$;
  za vsako točko u_i te komponente in za vsako njeno izhodno povezavo $u_i \to v$,
  ki predstavlja recimo lastniški delež d:
    če v ni del trenutne komponente:
      for $k := 1$ **to** p **do** $x_{vk} := x_{vk} + d \cdot y_{ik}$;

Na koncu tega postopka imamo v spodnjih p vrsticah matrike X ravno rezultate, po katerih sprašuje naloga: $x_{c+i,k}$ pove, kolikšen delež podjetja k obvladuje (posredno ali neposredno) človek i. Pri izpisu sicer pazimo na to, da moramo izpisati po eno vrstico za vsako podjetje, v matriki X pa imamo po eno vrstico za vsakega človeka, torej jo moramo pri izpisu v mislih sproti še transponirati.

V podrobnosti tega, kako najti krepko povezane komponente, kako jih topološko urediti in kako računati matriko S z Gaussovo eliminacijo, se tu ne bomo spuščali, saj so vsi trije postopki dobro znani in jih tu uporabljamo brez kakšnih posebnih sprememb ali prilagoditev; kdor želi, si lahko prebere več o njih v mnogih učbenikih ali na internetu.

Razmislimo še o časovni zahtevnosti naše rešitve. Skupno število povezav v grafu označimo z m (z vidika vhodnih podatkov pri naši nalogi je to $m = \sum_{i=1}^{p} k_i$). Iskanje krepko povezanih komponent je mogoče izvesti v $O(n + m)$ časa, enako tudi topološko urejanje; pri povezani komponenti velikosti t porabimo $O(t^3)$ časa za Gaussovo eliminacijo in $O(t^2p)$ za izračun matrike Y; poleg tega za vsako povezavo grafa porabimo $O(p)$ časa, ko obdelamo komponento, ki ji pripada začetno krajišče



Slika 1. Primer, kako si lahko predstavljamo iskanje največjega obroka kredita c, ki ga Andrej še lahko odplačuje v mesecih od $z+1$ do u. Iz točke R_z (kjer je z zadnji mesec pred začetkom odplačevanja) speljemo poltrak (črtkana črta na sliki), ki sprva kaže navpično navzdol, nato pa ga počasi obračamo v smeri, nasprotni urinemu kazalcu. Ko se prvič dotakne ene od točk iz $M = \{R_{z+1}, \ldots, R_u\}$, je naklon poltraka ravno iskani obrok posojila. Točka, pri kateri do tega prvega dotika pride, je vedno ena od točk na spodnjem robu konveksne ovojnice množice M; ta rob je na sliki narisan s polno črto.

te povezave. Če je komponent n/t, bo to skupaj $O(n+m+(n/t)(t^3+t^2p)+mp) = O(nt(t+p)+mp)$; če upoštevamo še, da je $t \ll p$, je to naprej $O(p(nt+m))$.

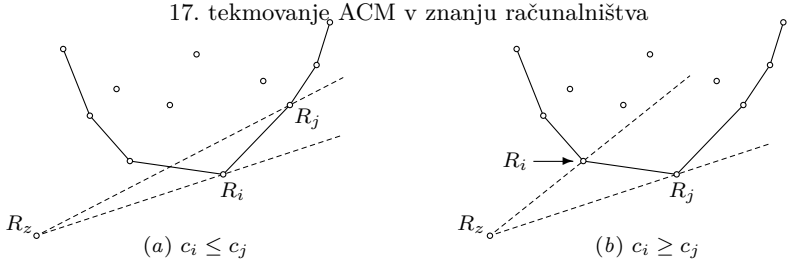
J. Hipoteka

Naj bo $A_t = a_1 + a_2 + \ldots + a_t$ skupna vsota denarja, ki ga Andrej zasluži v prvih t mesecih. Recimo zdaj, da razmišlja o posojilu, pri katerem bi od $(z+1)$-vega do u-tega meseca plačeval po c enot na mesec. Če je t z območja $z < t \leq u$, to pomeni, da do vključno t-tega meseca zasluži skupno $A_t - A_z$ denarja (saj naloga pravi, da ne sme računati na prihranke iz časa pred prvim mesecem, v katerem odplačuje posojilo), plača pa ga $(t-z)\cdot c$. Če razlika med prihodki in odhodki pri kakšnem t pade pod 0, si posojila ne more privoščiti, sicer pa si ga lahko. Imamo torej pogoj $A_t - A_z \geq (t-z)\cdot c$, kar lahko zapišemo tudi kot

$$\frac{A_t - A_z}{t-z} \geq c.$$

To si lahko predstavljamo geometrijsko: za vsak mesec t definirajmo v ravnini točko R_t s koordinatami (t, A_t). Na levi strani naše neenačbe imamo potem ravno naklon premice iz R_z skozi R_t; recimo mu $c_t := (A_t - A_z)/(t-z)$. Naloga torej sprašuje po največjem c, ki ni večji od nobenega c_t, to pa je ravno $c = \min\{c_{z+1}, \ldots, c_u\}$; oz. če smo natančni, naloga pravzaprav zahteva celoštevilski odgovor, torej $\lfloor c \rfloor$.

Lahko si predstavljamo, da bi začeli s premico, ki bi kazala iz R_z (skoraj) navpično navzdol ($c = -\infty$), in jo počasi obračali nasproti smeri urinega kazalca, dokler se ne bi od spodaj dotaknila ene od točk iz $M := \{R_{z+1}, \ldots, R_u\}$ (gl. sliko 1). Ta točka pa je lahko le ena od „najbolj zunanjih“ točk M-ja, s čimer mislimo tiste, ki tvorijo spodnji rob M-jeve konveksne ovojnice. O tem se lahko prepričamo takole: recimo, da se naša premica še ni dotaknila nobene točke z roba ovojnice; torej vse točke z roba ležijo še vedno nad premico; premica deli ravnino na dve polravnini in vsaka polravnina je konveksna množica; ker vse točke z roba ovojnice ležijo v zgornji polravnini in je ta konveksna, mora v tej polravnini ležati tudi celotna ovojnica, ta



(a) $c_i \le c_j$ (b) $c_i \ge c_j$

Slika 2.

Levo: če je $c_i \le c_j$, ležijo R_j in vse kasnejše točke (desno od R_j na robu ovojnice) nad premico skozi R_z in R_i (ali na njej).

Desno: če je $c_i \ge c_j$, ležijo R_i in vse prejšnje točke (levo od R_i na robu ovojnice) nad premico skozi R_z in R_j (ali na njej).

---

pa med drugim zajema tudi vse točke iz M; torej vse točke iz M ležijo še vedno nad premico in ni mogoče, da bi se bila naša premica kakšne od njih že dotaknila. Tako torej vidimo, da ko se premica prvič dotakne kakšne točke iz M, mora biti to ena od točk z roba njegove konveksne ovojnice.

Spodnji rob konveksne ovojnice si lahko predstavljamo kot zaporedje daljic, od katerih ima vsaka naslednja (če jih gledamo od leve proti desni) višji naklon kot prejšnja: če in ko padajo, padajo vse položneje, če in ko naraščajo, pa naraščajo vse strmeje. Recimo, da sta R_i in R_j dve zaporedni točki na tem robu; primerjajmo naklona c_i in c_j, pri katerih se ju dotakne naša premica iz R_z (gl. sliko 2). (1) Če je $c_i \le c_j$, to pomeni, da leži R_j nad premico R_zR_i ali na njej, poleg tega pa ima daljica R_iR_j višji ali enak naklon kot ta premica; zato imajo tudi vse kasnejše daljice na robu ovojnice (desno od R_j) višji ali enak naklon, zato pa so vsa krajišča teh daljic tudi nad premico R_zR_i ali na njej. Zato se naša premica ob obračanju nasproti smeri urinega kazalca ne dotakne nobene od točk $R_j, \ldots, R_u$ prej kot točke R_i. (2) Če pa je $c_i \ge c_j$, to pomeni, da leži R_i nad premico R_zR_j ali na njej, poleg tega pa ima daljica R_iR_j nižji ali enak naklon kot ta premica; zato imajo tudi vse predhodne daljice na robu ovojnice (levo od R_i) nižji ali enak naklon, zato pa so vsa krajišča teh daljic tudi nad premico R_zR_j ali na njej. Zato se naša premica ob obračanju nasproti smeri urinega kazalca ne dotakne nobene od točk $R_{z+1}, \ldots, R_i$ prej kot točke R_j.

Ta razmislek nam pove, da lahko tisto točko R_t, ki se je naša premica dotakne kot prve — torej tisto z najnižjim naklonom c_t — poiščemo s postopkom, podobnim bisekciji. Spodnji rob ovojnice tvori neka podmnožica množice M, recimo (od leve proti desni) točke $R_{t_1}, R_{t_2}, \ldots, R_{t_\tau}$ za $z+1 = t_1 < t_2 < \ldots < t_\tau = u$ (točki R_{z+1} in R_u kot najbolj leva in najbolj desna v M sta gotovo na robu ovojnice); toda da ne bomo ves čas pisali dvojnih indeksov, jih preimenujmo v $Q_1, \ldots, Q_\tau$.

podprogram NajmanjšiNaklon(R_z, H):
vhod: točka R_z tik pred začetkom odplačevanja kredita ter
zaporedje $H = \langle Q_1, \ldots, Q_\tau \rangle$ točk, ki tvorijo (od leve proti desni)
spodnji rob konveksne ovojnice množice M;
izhod: najmanjši naklon izmed premic $R_zQ_1, \ldots, R_zQ_\tau$;

$\ell := 1$; $d := \tau$;

while $\ell < d$:
  (* *Točke* $Q_1, \ldots, Q_{\ell-1}$ *ležijo nad premico* R_zQ_ℓ,
    točke $Q_{d+1}, \ldots, Q_\tau$ *pa nad premico* R_zQ_d. *)
  $m := \lfloor(\ell + d)/2\rfloor$;
  if ima premica R_zQ_{m+1} višji naklon kot premica R_zQ_m $(\star)$
  then $d := m$ (* *Točka* Q_{m+1} *leži nad premico* R_zQ_m, *zato ležijo nad to premico vse točke* $Q_{m+1}, \ldots, Q_\tau$. *)
  else $\ell := m + 1$; (* *Točka* Q_m *leži nad premico* R_zQ_{m+1}, *zato ležijo nad to premico vse točke* $Q_1, \ldots, Q_m$. *)

(* *Vse točke* $Q_1, \ldots, Q_\tau$ *ležijo nad premico* R_zQ_ℓ *ali na njej.* *)
return $(Q_\ell.y - R_z.y)/(Q_\ell.x - R_z.x)$;

Pri primerjavi naklonov v vrstici $(\star)$ je načeloma treba nekaj previdnosti. Naša naklona sta ulomka oblike α_i/β_i za $\alpha_i = Q_i.y - R_z.y$ in $\beta_i = Q_i.x - R_z.x$; pri tem naše x-koordinate štejejo mesece in gredo zato do največ $2 \cdot 10^5$, naše y-koordinate pa predstavljajo vsoto prihodkov po več mesecih in gredo zato do največ $2 \cdot 10^5 \cdot 10^9$, ker lahko Andrej v vsakem mesecu zasluži do 10^9 enot denarja. Če se želimo izogniti delu z ne-celimi števili in zato pogoj „$\alpha_{m+1}/\beta_{m+1} > \alpha_m/\beta_m$“ zapišemo kot „$\alpha_{m+1}\beta_m > \beta_{m+1}\alpha_m$“, bosta zmnožka na obeh straneh lahko velika do $4 \cdot 10^{19}$, kar je načeloma že preveč za 64-bitne celoštevilske tipe. Možne rešitve: lahko uporabimo kak nestandarden 128-bitni celoštevilski tip, če ga naš prevajalnik podpira (v C/C++ ponavadi **__int128**; mimogrede, v standardu za C bo od C23 naprej na voljo **_BitInt(128)**, ki bo sčasoma morda prišel tudi v standardni C++); lahko sami simuliramo 96-bitno množenje tako, da razdelimo α_m oz. α_{m+1} na zgornjih in spodnjih 32 bitov in vsako polovico posebej množimo z β_{m+1} oz. β_m; lahko izvedemo celoštevilsko deljenje in predstavimo α_i/β_i kot vsoto $\lfloor\alpha_i/\beta_i\rfloor + (\alpha_i \bmod \beta_i)/\beta_i$; lahko pa preprosto izračunamo α_m/β_m in α_{m+1}/β_{m+1} s plavajočo vejico in pričakujemo, da so zaokrožitvene napake dovolj majhne, da ne bodo vplivale na izid naše primerjave.[26]

Zdaj torej znamo izračunati največji obrok posojila, če imamo konveksno ovojnico; toda pri naši nalogi bomo morali odgovoriti na veliko poizvedb in si ne moremo privoščiti, da bi pri vsaki posebej računali konveksno ovojnico množice M. Pomagamo si lahko z neke vrste drevesom segmentov: vnaprej si pripravimo konveksne ovojnice za skupine 1, 2, 4, 8 itd. zaporednih mesecev, potem pa lahko minimalni

[26] Pri takšni primerjavi v resnici lahko pride do napak, vendar le v tem smislu, da se lahko pri predstavitvi s plavajočo vejico obe vrednosti α_m/β_m in α_{m+1}/β_{m+1} zaokrožita na isto število, zato naš program pomotoma misli, da sta enaki, čeprav je ena večja od druge. To lahko povzroči, da se bisekcija ustavi pri napačni od teh dveh točk (tisti, katere premica ima višji naklon kot druga). Toda naloga bo na koncu tako ali tako zahtevala celoštevilski rezultat — ne najnižjega naklona α_i/β_i, ampak najnižjo vrednost $\lfloor\alpha_i/\beta_i\rfloor$. Če pomotoma štejemo α_m/β_m in α_{m+1}/β_{m+1} za enaki, v resnici pa sta različni, bo to na koncu povzročilo napako le, če je med njima kakšno celo število in bosta zato različni tudi vrednosti $\lfloor\alpha_m/\beta_m\rfloor$ in $\lfloor\alpha_{m+1}/\beta_{m+1}\rfloor$. Toda imenovalci β_i teh naših naklonov so cela števila od 1 do $2 \cdot 10^5$ (število mesecev), zato se ulomek α_i/β_i od najbližjega celega števila ne more razlikovati za manj kot $1/(2 \cdot 10^5)$. Po drugi strani absolutna vrednost ulomka α_i/β_i ne more preseči 10^9, kolikor je Andrejev maksimalni mesečni prihodek. Dve števili, ki sta veliki do 10^9 in se razlikujeta za vsaj $1/(2 \cdot 10^5)$, bomo lahko pri predstavitvi s plavajočo vejico ločili, če imamo v mantisi prostora za vsaj $\log_2(10^9 \cdot 2 \cdot 10^5) \approx 47{,}5$ bitov; pri običajnem tipu **double** je mantisa dolga 52 bitov, torej bomo taki dve števili gotovo lahko ločili med seboj. Do primerov, ko se dva različna naklona zaokrožita v isto vrednost tipa **double**, bo torej prihajalo le takrat, ko dasta oba naklona po operaciji $\lfloor\cdot\rfloor$ isti rezultat, zato napaka pri primerjavi naklonov ne bo povzročila tudi napake v končnih rezultatih naše rešitve.

naklon (in s tem največji možni obrok posojila) za obdobje, ki nas zanima pri posamezni poizvedbi, določimo tako, da pregledamo pri vsaki velikosti po največ dve skupini. Naj bo torej $H_{\ell i}$ (za $0 \le \ell \le \lfloor \log_2 n \rfloor$ in $0 \le i < \lfloor n/2^\ell \rfloor$) zaporedje točk, ki tvorijo spodnji rob konveksne ovojnice množice $\{R_j : i \cdot 2^k < j \le (i+1) \cdot 2^k\}$. Potem lahko na posamezno poizvedbo odgovorimo takole:

podprogram NajvečjiObrok(s, k):
vhod: začetni mesec s, število mesecev k;
izhod: največji obrok, ki ga lahko Andrej plačuje vsak mesec tega obdobja;

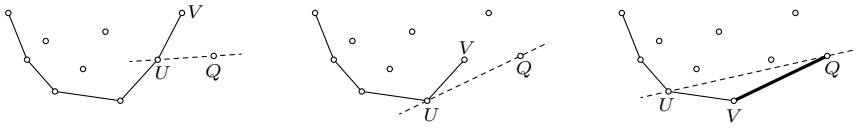
$z := s - 1$; $i := z$; $j := z + k$; $r := \infty$; $\ell := 0$;
while $i < j$:
  (* *V r je trenutno najmanjši naklon po vseh premicah* $R_z R_t$ *za* $s \le t \le i \cdot 2^\ell$ *in za* $j \cdot 2^\ell < t \le z + k$. *)
  if $i \bmod 2 = 1$ **then** $r := \min\{r, \text{NajmanjšiNaklon}(R_z, H_{\ell i})\}$; $i := i + 1$;
  if $j \bmod 2 = 1$ **then** $j := j - 1$; $r := \min\{r, \text{NajmanjšiNaklon}(R_z, H_{\ell j})\}$;
  $i := i/2$; $j := j/2$; $\ell := \ell + 1$;
return r;

Preden začnemo odgovarjati na poizvedbe, moramo pripraviti vse ovojnice $H_{\ell i}$, kar lahko naredimo z rekurzivnim razmislekom: pri $\ell = 0$ pokriva H_{0i} eno samo točko, R_{i+1}, ki zato seveda tudi leži na robu ovojnice, tako da je $H_{0i} = \langle R_{i+1} \rangle$; pri višjih ℓ pa pokriva $H_{\ell i}$ unijo množic točk, ki ju pokrivata $H_{\ell-1,2i}$ in $H_{\ell-1,2i+1}$. Tidve ovojnici lahko zato „zlijemo“ v $H_{\ell i}$ s postopkom, ki pravzaprav ni nič drugega kot znani Grahamov algoritem za računanje konveksne ovojnice:[27]

for $i := 0$ **to** $n - 1$ **do** $H_{0,i} :=$ seznam z edinim elementom R_{i+1};
for $\ell := 1$ **to** $\lfloor \log_2 n \rfloor$ **do for** $i := 0$ **to** $\lfloor n/2^\ell \rfloor - 1$:
  (* *Združimo ovojnici* $H_{\ell-1,2i}$ *in* $H_{\ell-1,2i+1}$ *v* $H_{\ell,i}$. *)
  $H_{\ell,i} :=$ kopija seznama $H_{\ell-1,2i}$;
  za vsako točko Q iz $H_{\ell-1,2i+1}$ (od leve proti desni):
    while ima $H_{\ell,i}$ vsaj dve točki:
      naj bo U predzadnja, V pa zadnja točka seznama $H_{\ell,i}$;
      if leži V nad premico UQ ali na njej (†)
      then pobriši V s konca seznama $H_{\ell,i}$
      else break;
    dodaj Q na konec seznama $H_{\ell,i}$;

Pogoj v vrstici (†) preveri, ali bi bila ovojnica še vedno konveksna, če bi ji na koncu dodali novo točko Q; če ne bi bila, moramo zadnjo točko V dosedanje ovojnice pobrisati (gl. sliko 3). Ta pogoj je enakovreden pogoju, da ima premica UV višji naklon kot premica UQ; gre torej za čisto tak pogoj kot ($\star$) v postopku NajmanjšiNaklon in ga lahko tudi preverjamo na enak način.

[27] Natančneje povedano gre za Andrewov algoritem, ki ločeno računa spodnji in zgornji rob konveksne ovojnice (za nas je zanimiv le spodnji rob), zato mu točk ni treba urediti po kotu glede na izbrano začetno točko (kot pri Grahamovem algoritmu), pač pa je dovolj že, če jih uredi le po x-koordinati. V našem primeru so točke že urejene po x-koordinati, zato tudi ni nujno, da manjše ovojnice zlivamo v večje na način, opisan v naši rešitvi; lahko bi preprosto izračunali vsako ovojnico na novo iz seznama točk, ki jih mora ta ovojnica pokrivati. Pri ovojnici $H_{\ell i}$ je to 2^ℓ točk in ker jih ni treba urejati, bi nam priprava ovojnice tudi takrat vzela le $O(2^\ell)$ časa, skupaj $O(n)$ za vse ovojnice na nivoju ℓ, zato pa $O(n \log n)$ za vse ovojnice sploh.



(1) (2) (3)

Slika 3. Primer dodajanja nove točke Q na konec konveksne ovojnice. (1) Dosedanja zadnja točka V leži nad premico med predzadnjo točko U in novo točko Q, zato jo pobrišemo, kajti če bi ovojnico podaljšali z daljico VQ, bi prenehala biti konveksna. — (2) Tudi po tem brisanju velja, da je zadnja točka (bivša predzadnja) nad premico med predzadnjo in novo točko, zato pobrišemo tudi njo. — (3) Zdaj zadnja točka leži *pod* premico med predzadnjo in novo, zato novo točko Q dodamo na konec ovojnice (ki se zato podaljša, kot kaže debela črta VQ).

Razmislimo še o časovni zahtevnosti naše rešitve. Ovojnice na nivoju ℓ so dolge po največ 2^ℓ točk vsaka, vse skupaj na tem nivoju pa največ n točk; zato porabimo na nivoju ℓ za vsako ovojnico po $O(2^\ell)$ časa, da jo izračunamo iz dveh pol manjših ovojnic z nivoja $\ell - 1$; za vse ovojnice na nivoju ℓ skupaj je to $O(n)$ časa. Ker gre lahko ℓ do $\lfloor \log_2 n \rfloor$, porabimo torej za pripravo vseh ovojnic $O(n \log n)$ časa (in pomnilnika). Pri odgovarjanju na poizvedbo pa izvede NajvečjiObrok po največ dva klica NajmanjšiNaklon pri vsakem ℓ; podprogram NajmanjšiNaklon pa takrat dela bisekcijo na seznamu dolžine kvečjemu 2^ℓ, zato porabi takrat $O(\ell)$ časa. Skupaj vzame torej poizvedba $O((\log n)^2)$ časa. Ker je poizvedb m, je časovna zahtevnost naše rešitve vsega skupaj $O((n + m \log n) \log n)$.

K. Spretno tabletno

Ogledali si bomo več rešitev, od preprostih in neučinkovitih do učinkovitejših, a bolj zapletenih.

Kubična rešitev. Pomagamo si lahko z dinamičnim programiranjem. Naloga pravi, da imamo n dni in moramo jemati tablete na vsakih a oz. b dni. Naj bo $f(\nu, \alpha, \beta)$ najboljša rešitev, če imamo podoben problem za ν dni z dodatno omejitvijo, da moramo eno tableto tipa A vzeti že v prvih α dneh, eno tableto tipa B pa v prvih β dneh. Ta funkcija je definirana za $0 \le \nu \le n$, $1 \le \alpha \le a$ in $1 \le \beta \le b$; rezultat, po katerem sprašuje naloga, je potem $f(n, a, b)$. Funkcije f ni težko računati z rekurzivno zvezo

$$f(\nu, \alpha, \beta) = \min\{f(\nu-1, \alpha-1, \beta-1), 1+f(\nu-1, a-1, \beta-1), 1+f(\nu-1, \alpha-1, b-1)\},$$

pri čemer se torej vsakič odločamo med tremi možnostmi: ali na prvi dan ne vzamemo nobene tablete ali vzamemo tableto A ali pa tableto B. Robna primera sta $f(0, \alpha, \beta) = 0$ (v 0 dneh ni treba vzeti nobene tablete) in $f(\nu, 0, \beta) = f(\nu, \alpha, 0) = \infty$ (za $\nu > 0$; v prvih 0 dneh bi morali vzeti eno tableto, kar pa je seveda nemogoče — do tega primera pridemo, če bi morali neko tableto vzeti že prejšnji dan, pa je nismo). Funkcijo f torej lahko računamo po naraščajočih ν, pri čemer lahko stare rezultate sproti pozabljamo: ko smo pri ν, potrebujemo rezultate za $\nu - 1$, tistih za $\nu - 2$ pa že ne več. Tako porabimo $O(nab)$ časa in $O(ab)$ pomnilnika.

Požrešni razmislek. Do boljših rešitev pridemo s požrešnim razmislekom. Če je $a = b = 2$, je lahko veljaven razpored samo tak, da izmenično jemljemo tablete obeh

tipov in jih skupaj vzamemo n (razen pri $n = 1$, kjer ni treba vzeti nobene tablete). V nadaljevanju se bomo zato omejili na primere, ko je vsaj eden od a in b večji od 2. Lahko si predstavljamo požrešen postopek za pripravo razporeda tablet po naraščajočih dnevih, pri katerem bi naslednjo tableto določenega tipa vzeli na zadnji dan, ko je to še mogoče: naslednjo tableto tipa A natanko a dni po prejšnji tableti tipa A, naslednjo tipa B pa natanko b dni po prejšnji tipa B. Toda če bi to pravilo zahtevalo, da na določen dan vzamemo obe tableti, bomo morali eno od njiju vzeti že dan prej, torej dan bolj zgodaj, kot bi bilo sicer nujno (če nas ne bi omejevala druga tableta); tako nastaneta dva zaporedna dneva, kjer prvi dan vzamemo eno tableto dan pred njenim zadnjim možnim dnem, naslednji dan pa vzamemo drugo tableto na dan, ki je zanjo zadnji možni. Nobenega dobrega razloga pa ni, da bi kakšno tableto vzeli dva ali več dni bolj zgodaj (pred njenim zadnjim možnim dnem), kakor tudi ne, da bi dva zaporedna dneva vzeli tableti, od katerih ne bi nobena padla na svoj zadnji možni dan.

O tem se je primerno prepričati tudi malo bolj formalno. Tableti tipa A (oz. B), ki jo (za neki $k > 0$) vzamemo $a - k$ (oz. $b - k$) dni po prejšnji tableti istega tipa (ali po začetku razporeda), bomo rekli, da je *zgodnja*, oz. še natančneje, da je *zgodnja za k dni*. Tableti, ki je zgodnja za več kot en dan (torej $k > 1$), bomo rekli *zelo zgodnja*. Zgodnji tableti, ki ji naslednji dan ne sledi neka ne-zgodnja tableta (pač pa še ena zgodnja ali pa dan, ko ne vzamemo nobene tablete, ali pa naslednjega dne sploh ni, ker se razpored že konča), pa bomo rekli *problematična*.

Dokazali bomo, da je med optimalnimi razporedi (torej takimi z najmanjšim možnim številom tablet) tudi tak, ki nima nobene zelo zgodnje in nobene problematične tablete.

Pa recimo, da to ni res; torej ima vsak optimalni razpored vsaj eno tableto, ki je zelo zgodnja in/ali problematična. Za vsak optimalni razpored naj bo z prvi dan, ko nastopi kakšna taka tableta; med optimalnimi razporedi izberimo tistega z največjim z (če jih je več, je vseeno, katerega vzamemo).

Tableta na dan z je vsekakor zgodnja; morda sledi v naslednjih nekaj dneh še nekaj zgodnjih tablet (ki niso nujno zelo zgodnje in/ali problematične); naj bo u zadnji od teh dni. Na vsak dan od z do u vzamemo torej neko zgodnjo tableto (lahko je tudi $u = z$), na dan $u + 1$ pa ne. Temu obdobju od z do u bomo rekli *blok* zgodnjih tablet.

Če je $u = n$ (torej blok sega vse do konca razporeda), lahko vse tablete z dni od z do $n - 1$ zamaknemo za en dan naprej; razpored ostane veljaven, ima pa eno tableto manj kot prej, kar je protislovje.

Če na dan $u + 1$ ne vzamemo nobene tablete, lahko vse tablete z dni od z do u (cel blok) zamaknemo za en dan naprej; razpored ostane veljaven in ima enako število tablet kot prej, torej je še vedno optimalen; poleg tega pa zdaj na dan z ne vzamemo nobene tablete, torej nastopi prva zelo zgodnja in/ali problematična tableta v njem kasneje kot na dan z;[28] to pa je spet v protislovju s tem, kako smo izbrali naš prvotni razpored.

[28] Pred dnevom z ni mogla zaradi opisanega premika nobena tableta na novo postati zelo zgodnja (saj se tam ni nič spremenilo). Poleg tega pa tudi ni mogla nobena na novo postati problematična; to bi se namreč lahko zgodilo le na dan $z - 1$, če bi pred premikom tableta z dne z ščitila tableto $z - 1$ pred problematičnostjo; a za kaj takega bi morala tableta na dan z biti ne-zgodnja, kar pa ni bila. Enak razmislek velja tudi kasneje pri primerih 1.3, 1.4 in 2.

	(1.1)		(1.2)			(1.3)			(1.4)		
dan	z	n	z			z			z		
prej	B	A	B	A	B	B	A	A	B	A	
	z	n	z	n	?	z	n	z	z	n	
potem		A		A	B	A	B	A		A	B
		n		n	?	z	z	?		n	?

Primer (1). Na dan $z = u$ imamo zelo zgodnjo tableto tipa B, ki ji sledi ne-zgodnja tableta tipa A na dan $u + 1$. Znaki pod tabletami povedo, ali so zgodnje (z) ali ne (n) ali pa v splošnem ne vemo, kaj od tega dvojega so (?).

---

dan	z	u	$u+1$	$u+2$	$u+3$	$\cdots$	$v-1$	v	$(v+1)$
prej	A	B	A	C_{u+2}	C_{u+3}	$\cdots$	C_{v-1}	C_v	
	z	z	n	Z_{u+2}	Z_{u+3}	$\cdots$	Z_{v-1}	Z_v	
potem		A	B	A	C_{u+2}	$\cdots$	C_{v-2}	C_{v-1}	C_v
		n	?	n	Z_{u+2}	$\cdots$	Z_{v-2}	Z_{v-1}	Z_v

Primer (2). Blok tvorita zgodnji tableti A (na dan $z = u - 1$) in B (na dan u), ki jima sledi ne-zgodnja A in potem še nekaj (0 ali več) zaporednih dni s tabletami do vključno dneva v; simbol C_t predstavlja tip tablete, ki jo vzamemo na dan t, simbol pa Z_t pa to, ali je zgodnja ali ne. Stolpec $v + 1$ je prisoten le, če je $v < n$.

Ostane le še možnost, da na dan $u+1$ vzamemo neko ne-zgodnjo tableto. Recimo brez izgube za splošnost, da je to tableta tipa A. Torej smo na dan u vzeli tableto B, sicer bi imeli v dneh u in $u + 1$ dve zaporedni tableti tipa A in bi bila tista na dan $u + 1$ tudi zgodnja (ker je $a \geq 2$).

(1) Če je $z = u$ (torej če je blok dolg en sam dan): na dan z oz. u imamo torej zgodnjo tableto tipa B, na dan $u + 1$ pa ne-zgodnjo tableto tipa A. Zato tableta na dan z ni problematična, torej mora biti zelo zgodnja (saj vemo, da je vsaj nekaj od tega dvojega: problematična in/ali zelo zgodnja). Ločili bomo štiri primere, kot kaže slika zgoraj.

(1.1) Če je $u + 1 = n$: ker je tableta B na dan u zgodnja za vsaj dva dni, je lahko tudi ne vzamemo in imamo še vedno veljaven razpored z eno tableto manj; protislovje.

(1.2) Če na dan $u+2$ vzamemo B: ker je tableta B na dan u zgodnja za vsaj dva dni, je lahko tudi ne vzamemo in imamo še vedno veljaven razpored z eno tableto manj; protislovje.

(1.3) Če na dan $u+2$ vzamemo A: ne-zgodnjo A z dne $u+1$ lahko prestavimo na dan u, s čimer postane zgodnja, vendar le za en dan (torej ne zelo zgodnja); zgodnjo B z dne u (ki je bila zgodnja za več kot en dan) pa prestavimo na dan $u + 1$, kjer je še vedno zgodnja, vendar ne nujno za več kot en dan. Razpored ostane veljaven, vendar se zdaj na dan z začne blok vsaj dveh zgodnjih tablet, A na dan z in B na dan $z + 1$, pri čemer je prva od njiju problematična. Od tu nadaljujemo kot pri primeru (2) spodaj.

(1.4) Če na dan $u+2$ ne vzamemo ničesar: tableto B z dneva u, kjer je bila zgodnja za vsaj dva dni, lahko premaknemo na dan $u+2$; tableta A na dan $u+1$ pa naj ostane, kjer je bila (in je še vedno ne-zgodnja). Razpored ostane veljaven, pri čemer do vključno dne z ni nobene problematične ali zelo zgodnje tablete; protislovje.

(2) Ostane možnost, da je $z < u$, torej je blok dolg vsaj dva dni. Če sta v

bloku kdaj dva zaporedna dneva, ko vzamemo tableto istega tipa, recimo dneva v in $v+1$, lahko vse tablete z dni od z do v zamaknemo za en dan naprej (na dan $v+1$ vzamemo enako tableto kot prej); razpored ostane veljaven, ima pa eno tableto manj kot prej, kar je protislovje. Neizogibno je torej, da v bloku izmenično jemljemo vsak dan tableto drugega tipa.

Ker se blok konča s tableto B na dan u, smo morali na dan $u-1$ vzeti tableto A. Ker smo poleg tega na dan $u+1$ tudi vzeli tableto A in je bila le-ta ne-zgodnja, mora biti $a=2$, zato pa $b>2$ (saj smo primer $a=b=2$ posebej obravnavali že prej). Toda pri $a=2$ je lahko tableta tipa A (na dan $u-1$) zgodnja le tedaj, če jo vzamemo prvi dan celotnega razporeda (kar pomeni, da se blok tam tudi začne: $z=u-1=1$) ali pa takoj naslednji dan po prejšnji tableti tipa A. Ta druga možnost bi pomenila, da smo že na dan $u-2$ vzeli tableto tipa A; če bi tudi ta dan pripadal bloku, bi bila v njem dva zaporedna dneva z enako tableto, za to pa smo že videli, da je nemogoče; tudi v tem primeru je torej neizogibno, da se blok začne šele z dnem $u-1=z$.

Zdaj torej vidimo, da je naš blok neizogibno dolg le dva dni: na dan z vzamemo zgodnjo tableto A, na dan $u=z+1$ pa zgodnjo tableto B. Tista na dan z je problematična (ni pa zelo zgodnja, kajti pri $a=2$ sploh ni mogoče, da bi tableto A vzeli dva ali več dni pred zadnjim možnim dnem). Za blokom pride dan $u+1$, ko vzamemo ne-zgodnjo tableto tipa A. Mogoče sledi potem še več zaporednih dni, ko vsak dan vzamemo tableto; naj bo v zadnji tak dan (lahko je tudi $v=u+1$). To zaporedje se konča bodisi pri $v=n$ bodisi takrat, ko na dan $v+1$ ne vzamemo nobene tablete (gl. sliko na str. 167).

Če to zaporedje tablet z dni od z do v zamaknemo za en dan naprej, razpored ostane veljaven: zaporedje se je začelo z zgodnjima A in B na dneh $u-1$ in u, ki ju torej smemo premakniti za en dan naprej in še ne bosta prepozni; od tam naprej pa je za vsako tableto v zaporedju ostal čas do prejšnje tablete istega tipa enak kot pred premikom, ker se je premaknilo zaporedje kot celota (vse tablete za en dan); za morebitne tablete po dnevu $v+1$ pa se je zdaj čas do prejšnje tablete istega tipa lahko le zmanjšal (ker so se tiste iz zaporedja premaknile za en dan naprej) ali ostal enak.

V primeru $v=n$ zaradi našega premika tableta, ki smo jo prej vzeli na dan v, celo izpade iz razporeda; novi razpored je torej veljaven in ima manj tablet kot prej, kar je protislovje. Ostane le še možnost, da je $v<n$; tableta z dne v ob premiku pride na dan $v+1$, ki je bil prej prazen. Novi razpored je veljaven, pri čemer pa zdaj na nobenem dnevu do vključno z ni problematične ali zelo zgodnje tablete; to pa je spet protislovje.

Vidimo torej, da v vsakem primeru pridemo v protislovje z dejstvom, da je imel že naš prvotni razpored najmanjše število tablet in da je imel med takšnimi razporedi največji z. Predpostavka, da imajo vsi optimalni razporedi kakšno zelo zgodnjo in/ali kakšno problematično tableto, nas je neizogibno pripeljala v protislovje. □

V nadaljevanju se bomo torej omejili na razporede brez zelo zgodnjih in brez problematičnih tablet, saj smo zdaj videli, da je med optimalnimi razporedi gotovo tudi kakšen tak.

Kvadratna rešitev. Pri dinamičnem programiranju se lahko zdaj omejimo na primere, ko smo eno tableto ravnokar vzeli: vpeljimo $f_1(\nu,\beta)=f(\nu,a,\beta)$ in $f_2(\nu,\alpha)=$

$f(\nu, \alpha, b)$. Funkcija f_1 torej rešuje problem, pri katerem imamo pred seboj ν dni, tik pred tem smo vzeli tableto A (in moramo naslednjo šele v prvih a dneh), eno tableto tipa B pa moramo vzeti v prvih β dneh. Pri f_2 je podobno, a z zamenjano vlogo obojih tablet. Razmislimo, kako računati f_1:

- robni primer je $f_1(\nu, 0) = \infty$ za $\nu > 0$, podobno kot pri f (že prejšnji dan bi morali vzeti tableto tipa B, pa je nismo);
- če je $\nu < a$ in $\nu < \beta$, ni treba vzeti nobene tablete več: $f_1(\nu, \beta) = 0$;
- če je $\nu \geq a < \beta$: v prvih $a - 1$ dneh ne smemo vzeti nobene tablete B, kajti ta bi bila zelo zgodnja; iz enakega razloga tudi v prvih $a - 2$ dneh ne smemo vzeti nobene tablete A. Če vzamemo tableto A na dan $a - 1$, bo zgodnja, kar pomeni, da bi morali na dan a vzeti neko ne-zgodnjo tableto, sicer bi bila tista na dan $a - 1$ problematična; toda če na dan a vzamemo še eno A, bo ta zgodnja; če pa na dan a vzamemo tableto B, bo ta tudi zgodnja (kajti ne-zgodnja bi bila šele na dan β, to pa je $> a$). Tablete A torej ne smemo vzeti prej kot na dan a; kasneje kot takrat pa je tudi ne smemo, ker bo že prepozno. Prva tableta, ki jo moramo v tem scenariju vzeti, je torej tableta tipa A na a-ti dan, potem pa nadaljujemo od $(a + 1)$-vega dne naprej: $f_1(\nu, \beta) = 1 + f_1(n - a, \beta - a)$;
- če je $\nu \geq \beta < a$, je stvar podobna kot v prejšnji točki, le da moramo najprej vzeti tableto B na β-ti dan: $f_1(\nu, \beta) = 1 + f_2(\nu - \beta, a - \beta)$;
- ostane še primer, ko je $\nu \geq a = \beta$. V prvih $a - 2$ dneh ne smemo vzeti nobene tablete, ker bi bila zelo zgodnja; najkasneje do dneva a pa moramo vzeti vsaj eno tableto vsakega tipa. Torej moramo v dneh $a - 1$ in a vzeti po eno A in eno B; odločiti se moramo le, katero tableto bomo vzeli dan prej, A ali B:

  $$f_1(\nu, \beta) = 2 + \min\{f_1(\nu - a, b - 1), f_2(\nu - a, a - 1)\}.$$

  Prvi kandidat v $\min\{\ldots\}$ se nanaša na možnost, da na $(a - 1)$-vi dan vzamemo tableto B, na a-ti dan pa tableto A; pri drugem kandidatu pa je ravno obratno.

Razmislek za f_2 je analogen, le vloga obeh vrst tablet se zamenja. Rezultat, po katerem sprašuje naloga, je na koncu $f_1(n, b) = f_2(n, a) = f(n, a, b)$. Funkciji f_1 in f_2 lahko računamo po naraščajočih ν in dobimo rezultat v $O(n(a + b))$ časa. Tudi tu lahko načeloma stare vrednosti do neke mere pozabljamo: ko računamo $f_{1,2}(\nu, \cdot)$, ne potrebujemo več vrednosti $f_1(\mu, \cdot)$ za $\mu < \nu - a$ ali vrednosti $f_2(\mu, \cdot)$ za $\mu < \nu - b$; poraba pomnilnika je zato $O(ab)$, enako kot pri prejšnji rešitvi. (Če starih vrednosti sproti ne pozabljamo, pa porabimo $O(n(a + b))$ pomnilnika.)

Linearna rešitev. Zgoraj pri kvadratni rešitvi smo videli, da ta večino časa ni imela nobene izbire glede tega, kdaj vzeti prvo naslednjo tableto (in katero): naslednjo tableto vzame tako, da ta ne bo zgodnja. Edina izjema nastopi, če bi morali obe tableti vzeti na isti dan (torej če npr. računamo $f(\nu, \alpha, \beta)$ za $\alpha = \beta$); temu pojavu bomo rekli *konflikt*. Konflikt razrešimo tako, da eno od tablet vzamemo en dan prej; pri tem imamo na izbiro dve možnosti glede tega, katero tableto vzeti prej.

Naš postopek je imel torej možnost izbire le pri konfliktih, vmes pa ne. Na začetku reševanja to na primer pomeni, da jemljemo tablete tipa A na dan $a, 2a, 3a, \ldots$ in tablete tipa B na dan $b, 2b, 3b, \ldots$; prvi konflikt nastopi šele na dan $c_0 := \operatorname{lcm}(a, b)$ (najmanjši skupni večkratnik števil a in b). Takrat vzamemo eno tableto dan prej in smo potem v stanju, ko smo ravnokar v zadnjih dveh dneh vzeli obe tableti.

Skupno število tablet, ki smo jih vzeli v teh prvih c_0 dneh, je v vsakem primeru $t_0 := c_0/a + c_0/b$.[29]

Naj bo torej $g_1(\nu) = f(\nu, a, b-1)$ najboljša rešitev za ν dni pri predpostavki, da smo na zadnji dan pred pred začetkom teh ν dni vzeli tableto tipa A, še en dan prej pa tableto tipa B; in podobno $g_2(\nu) = f(\nu, a-1, b)$, le da smo tu zadnji dan vzeli tableto B, dan prej pa A. Rezultat, po katerem sprašuje naloga, je torej enak $t_0 + \min\{g_1(n-c_0), g_2(n-c_0)\}$: v prvih c_0 dneh vzamemo t_0 tablet, nato pa se moramo odločiti, kako razrešiti konflikt na dan c_0 (katero tableto vzeti dan prej), in v skladu s tem rešiti preostalih $n - c_0$ dni z eno od funkcij g_1 in g_2.

Razmislimo zdaj o funkciji $g_1(\nu)$; kdaj nastopi prvi konflikt? Ker smo tableto A vzeli zadnji dan pred opazovanim obdobjem, bomo naslednjo vzeli na dan a, potem $2a, 3a$ in tako naprej; tableto B pa smo vzeli že predzadnji dan pred opazovanim obdobjem, zato bomo naslednjo vzeli že na dan $b-1$, nato pa na $2b-1$, $3b-1$ in tako naprej. Konflikt nastopi torej takrat, ko za neki naravni števili x in y velja $a \cdot x = b \cdot y - 1$ oz. $a \cdot x - b \cdot y = 1$. To je linearna diofantska enačba z dvema neznankama in jo lahko rešimo z razširjenim Evklidovim algoritmom.[30] Pri tem dobimo tudi najmanjši skupni delitelj $d = \gcd(a, b)$. Če je ta slučajno > 1, je leva stran naše diofantske enačbe vedno večkratnik d, desna pa nikoli, torej enačba nima rešitev; konflikt torej nikoli ne nastopi in lahko bomo do konca jemali vsako tableto na zadnji možni dan. Takrat je torej $g_1(\nu) = \lfloor \nu/a \rfloor + \lfloor \nu/b \rfloor$.

Več dela pa imamo pri $d = 1$. Razširjeni Evklidov algoritem nam tam najde neko rešitev, recimo (x_0, y_0), poleg nje pa ima enačba še neskončno drugih rešitev oblike $(x_0 + kb, y_0 + ka)$ za vse $k \in \mathbb{Z}$. Če v formulo za dan konflikta, to je ax (ali $by-1$, kar je isto), vstavimo te rešitve, dobimo $ax_0 + kb$. Nas zanima prvi konflikt, ki nastopi v prihodnosti, torej najmanjši $ax_0 + kb > 0$. Vzeti moramo torej $k := 1 + \lfloor -ax_0/b \rfloor$, pri tem k pa dobimo datum prvega konflikta $c_1 := ax_0 + kb$.

Če je $\nu < c_1$, do konflikta ne pride in lahko neovirano jemljemo tablete vsako na zadnji možni dan; sicer pa na c_1-ti dan nastopi konflikt in se spet lahko odločimo, kako ga bomo razrešili, torej ali bomo dan prej vzeli tableto A ali B. Tako smo dobili:

$$g_1(\nu) = \begin{cases} \lfloor \nu/a \rfloor + \lfloor \nu/b \rfloor, & \text{če } d > 1 \text{ ali } \nu < c_1 \\ t_1 + \min\{g_1(\nu - c_1), g_2(\nu - c_1)\}, & \text{sicer.} \end{cases}$$

Pri tem t_1 pomeni število tablet, ki jih vzamemo do konflikta: $t_1 = c_1/a + (c_1+1)/b$.

Razmislek za funkcijo g_2 je seveda čisto podoben, le vloga obeh vrst tablet je zamenjana; dnevu, ko tedaj nastopi prvi konflikt, recimo c_2. Vidimo torej, da lahko obe funkciji računamo po naraščajočih vrednostih ν od 0 do $n - c_0$, tako da nalogo rešimo v $O(n)$ časa. Pri tem porabimo načeloma tudi $O(n)$ prostora, kar pa lahko zmanjšamo na $O(\min\{n, \max\{c_1, c_2\}\})$, če sproti pozabljamo stare

[29] Poseben primer nastopi, če je $n \leq c_0$. Takrat do konflikta sploh ne pride (če je $n < c_0$) in lahko neovirano jemljemo tablete vsakič na zadnji možni dan; ali pa pride do konflikta zadnji dan in lahko eno tableto brez težav premaknemo en dan nazaj, ne da bi se skupno število pojedenih tablet kaj povečalo; skupaj vzamemo v vsakem primeru $\lfloor n/a \rfloor + \lfloor n/b \rfloor$ tablet.

[30] Gl. npr. knjižico Jožeta Grassellija *Diofantske enačbe* (Ljubljana, DMFA 1984) od str. 29 naprej in Wikipedijo *s. v.* Extended Euclidean algorithm. Testni primeri na našem tekmovanju so sicer tako majhni, da lahko prvi konflikt poiščemo tudi naivno, z zanko: začnimo z $x = 0$, $y = 0$, nato pa na vsakem koraku poglejmo, ali je $a \cdot x$ manjše ali večje od $b \cdot y - 1$; če je manjše, povečajmo x za 1; če je večje, povečajmo y za 1; če pa je $a \cdot x = b \cdot y - 1$, smo našli konflikt.

rezultate, ki jih ne bomo več potrebovali. Poseben primer, kot smo videli, je $d > 1$, ko porabimo le $O(\log n)$ časa za Evklidov algoritem in potem $O(1)$ za sam izračun iskanega rezultata; poraba prostora pa je tedaj $O(1)$.

Ta rešitev je za naše tekmovanje že dovolj hitra, prejšnji dve pa bi bili prepočasni.

Boljša linearna rešitev. Za primere, ko je $n \le c_0$ ali $d > 1$, smo že zgoraj dobili rešitev s časovno zahtevnostjo $O(\log n)$ (zaradi Evklidovega algoritma) in tega ne bomo več poskušali izboljševati. Oglejmo si zdaj malo pobliže primer, ko je $n > c_0$ in $d = 1$. Kot smo videli pri prejšnji rešitvi, je optimalni razpored naslednje oblike: najprej je obdobje c_0 dni, ki se konča s konfliktom in v katerem vzamemo t_0 tablet; preostanek do n dni je razdeljen na obdobja po c_1 dni (ko vzamemo po t_1 tablet) in/ali obdobja po c_2 dni (ko vzamemo po t_2 tablet); prav na koncu pa pride še obdobje recimo r dni, ko ni konflikta. Če smo zadnji konflikt rešili tako, da smo tableto B vzeli dan prej, mora biti $r < c_1$ (sicer bi že prišlo do novega konflikta) in v teh zadnjih r dneh vzamemo $g_1(r) = \lfloor r/a \rfloor + \lfloor (r+1)/b \rfloor$ tablet; če pa smo pri zadnjem konfliktu vzeli dan prej tableto A, mora biti $r < c_2$ (da ne pride do novega konflikt) in v zadnjih r dneh vzamemo $g_2(r) = \lfloor (r+1)/a \rfloor + \lfloor r/b \rfloor$ tablet. Če je r manjši tako od c_1 kot od c_2, imamo na voljo obe možnosti in bomo seveda izbrali manjšo. V zadnjih r dneh torej uporabimo toliko tablet:

$$h(r) = \min\{h_1(r), h_2(r)\} \quad \text{za} \quad h_i(r) = \begin{cases} g_i(r), & \text{če } r < c_i \\ \infty & \text{sicer.} \end{cases}$$

V nadaljevanju razmisleka bomo brez izgube za splošnost predpostavili, da je $c_1 \ge c_2$; če je obratno, je razmislek čisto podoben.

Vrstni red, v katerem si sledijo obdobja dolžine c_1 in obdobja dolžine c_2, pravzaprav ni pomemben; nanj vplivamo z odločitvami o tem, kako razrešiti posamezni konflikt. Vprašanje je torej le, *koliko* obdobij vsake vrste imamo, ne pa tudi, v kakšnem vrstnem redu si sledijo. Recimo, da bo u obdobij dolžine c_1 in v obdobij dolžine c_2. Potem bomo vsega skupaj vzeli $t_0 + t_1 u + t_2 v + h(r)$ tablet. Števil u_0, u_1 in r si seveda ne smemo izbrati poljubno, saj jih povezuje omejitev, da je celotno obdobje dolgo n dni in da v zadnjih r dneh ne sme priti do novega konflikta:

$$n = c_0 + c_1 u + c_2 v + r, \quad 0 \le r < c_1.$$

Če zapišemo ta izraz kot

$$(n - c_0 - c_2 v) = c_1 u + r, \quad 0 \le r < c_1,$$

vidimo, da čim si izberemo v, sta potem u in r že enolično določena: to sta količnik in ostanek pri deljenju $n - c_0 - c_2 v$ s številom c_1. Pišimo $\tilde{n} = n - c_0$; potem imamo

$$r = (\tilde{n} - c_2 v) \bmod c_1, \quad u = \lfloor (\tilde{n} - c_2 v)/c_1 \rfloor = (\tilde{n} - c_2 v - r)/c_1.$$

Skupno število tablet lahko torej zapišemo kot funkcijo v-ja:

$$\begin{aligned} T(v) &= t_0 + t_1 u + t_2 v + h(r) \\ &= t_0 + t_1 (\tilde{n} - c_2 v - r)/c_1 + t_2 v + h(r). \end{aligned}$$

Minimalno število tablet torej dobimo tako, da gremo v zanki po vseh v od 0 do $v_{\max} := \lfloor \tilde{n}/c_2 \rfloor$, pri vsakem v izračunamo r, $h(r)$ in nato $T(v)$ ter med tako dobljenimi $T(v)$ obdržimo najmanjšo vrednost. Časovna zahtevnost te rešitve je $O(n/c_2)$ — kar je v najslabšem primeru še vedno $O(n)$, v praksi pa je še vseeno bolje od prejšnje linearne rešitve — prostorska pa je le $O(1)$.

Hevristika za predčasni konec iskanja. Kot zanimivost si oglejmo še hevristiko, s katero lahko pri tej zadnji rešitvi pogosto, čeprav ne vedno, prihranimo veliko časa. Funkcijo $T(v)$ lahko preuredimo v

$$T(v) = S \cdot v + Z - rt_1/c_1 + h(r) \quad \text{za } S = t_2 - t_1 c_2/c_1,\ Z = t_0 + t_1 \tilde{n}/c_1.$$

Zadnja dva člena v formuli za $T(v)$ se deloma odštejeta. Pri tretjem členu upoštevajmo, da je $t_1 = c_1/a + (c_1+1)/b$, zato je $rt_1/c_1 = r/a + r/b + r/(c_1 b)$. Četrti člen, $h(r)$, je enak enemu od $g_1(r)$ in $g_2(r)$; recimo za začetek, da je enak $g_1(r)$; ta je naprej enak

$$\begin{aligned} g_1(r) &= \lfloor r/a \rfloor + \lfloor (r+1)/b \rfloor \\ &= (r - (r \bmod a))/a + (r + 1 - ((r+1) \bmod b))/b \\ &= r/a - (r \bmod a)/a + r/b - ((r+1) \bmod b)/b + 1/b \end{aligned}$$

Ko vstavimo to v izraz za $T(v)$, se r/a in r/b v tretjem in četrtem členu odštejeta in ostane nam

$$\begin{aligned} T(v) &= S \cdot v + Z - \underbrace{r/(c_1 b)}_{<1/b} - \underbrace{(r \bmod a)/a}_{\le 1-1/a} - \underbrace{((r+1) \bmod b)/b}_{\le 1-1/b} + 1/b \\ &> S \cdot v + Z - 2. \end{aligned}$$

Pri tem smo upoštevali, da je $r < c_1$, zato $r/(c_1 b) < 1/b$; in da je $r \bmod a \le a - 1$, zato $(r \bmod a)/a \le 1 - 1/a$ (in podobno pri b).

Če je $h(r) = g_2(r)$, je razmislek podoben in zaključek enak; dobimo

$$\begin{aligned} g_2(r) &= \lfloor (r+1)/a \rfloor + \lfloor (r+1)/b \rfloor = \ldots \\ &= r/a - ((r+1) \bmod a)/a + r/b - (r \bmod b)/b + 1/a \end{aligned}$$

in ko to vstavimo v $T(v)$, dobimo

$$\begin{aligned} T(v) &= S \cdot v + Z - \underbrace{r/(c_1 b)}_{<1/b} - \underbrace{((r+1) \bmod a)/a}_{\le 1-1/a} - \underbrace{(r \bmod b)/b}_{\le 1-1/b} + 1/a \\ &> S \cdot v + Z - 2. \end{aligned}$$

V vsakem primeru torej velja $T(v) > S \cdot v + Z - 2$. Iz naših formul za $T(v)$ lahko opazimo tudi zgornjo mejo: $T(v) \le S \cdot v + Z + \max\{1/a, 1/b\}$. Z drugimi besedami: vrednost $T(v)$ se nikoli ne odmakne prav daleč od linearne funkcije $S \cdot v + Z$. Ta odmik označimo z $R(v) := T(v) - (S \cdot v + Z)$; pravkar smo videli, da vedno velja $R(v) > -2$. Dogovorimo se, da bomo v-je pregledovali v tisti smeri, v katero vrednost $S \cdot v + Z$ narašča: če je $S \ge 0$, bomo šli po naraščajočih v od 0 do $v_{\max}$, če pa je $S < 0$, bomo šli po padajočih v od $v_{\max}$ do 0.

Recimo zdaj, da smo nekaj v-jev že pregledali in da je najboljša doslej najdena rešitev enaka $T(v^\star)$. Recimo še, da se pri nekem kasnejšem v (torej $v > v^\star$, če je $S \geq 0$, oz. $v < v^\star$, če je $S < 0$) izkaže, da velja

$$S \cdot v + Z \geq T(v^\star) + 1. \qquad (\star)$$

Če na obeh straneh prištejemo $R(v)$, dobimo $T(v) \geq T(v^\star) + 1 + R(v)$; in ker je $R(v) > -2$, dobimo $T(v) > T(v^\star) - 1$. Ker sta $T(v)$ in $T(v^\star)$ celi števili, je ta neenakost možna le tako, da je $T(v) \geq T(v^\star)$. Rešitev pri v torej ni nič boljša od tiste pri $v^\star$, ki smo jo poznali že od prej. Toda če je pogoj $(\star)$ izpolnjen pri trenutnem v, bo tudi pri vsakem naslednjem, saj smo rekli, da pregledujemo v-je v takem vrstnem redu, da funkcija $S \cdot v + Z$ narašča; leva stran pogoja $(\star)$ torej narašča, desna pa se lahko le zmanjšuje (ko odkrijemo kakšno novo rešitev s še manjšim številom tablet). Ker bo pogoj $(\star)$ izpolnjen tudi v prihodnje, bo torej tudi za prihodnje v veljalo $T(v) \geq T(v^\star)$, torej najboljše dosedanje rešitve ne bomo nikoli več izboljšali in preostalih v-jev sploh ni treba pregledati.

Pri preverjanju pogoja $(\star)$ je koristno upoštevati, da je leva stran enaka $T(v) + R(v)$; če nesemo $T(v)$ na desno, dobimo $R(v) \geq T(v^\star) + 1 - T(v)$; v tej obliki ga je laže preverjati s celoštevilsko aritmetiko (na levi strani imamo vsoto nekaj ulomkov z imenovalcem a, b ali $c_1 b$, na desni pa je celo število).

Ta hevristika pogosto močno zmanjša število v-jev, ki jih moramo pregledati, ne pa vedno (npr. ker je včasih S zelo blizu 0).

Logaritemska rešitev. Spet se omejimo na primere, ko je $n > c_0$ in $d = 1$, saj smo za ostale nalogo rešili v logaritemskem času že prej. Če ne bi bilo konfliktov oz. če bi smeli vzeti obe tableti na isti dan, bi vedno porabili $\lfloor n/a \rfloor + \lfloor n/b \rfloor$ tablet. Ko neki konflikt razrešimo tako, da tableto nekega tipa vzamemo dan prej, je učinek na število porabljenih tablet tega tipa enak, kot če bi se opazovano obdobje podaljšalo za en dan (npr. z n dni na $n + 1$ dni in tako naprej). Tako kot doslej recimo, da začetnemu obdobju c_0 dni sledi še u obdobij s po c_1 dnevi (tik pred vsakim od teh obdobij pride konflikt, ki smo ga razrešili tako, da smo tableto tipa B vzeli en dan bolj zgodaj) in v obdobij s po c_2 dnevi (pred vsakim od teh pa je konflikt, pri katerem smo vzeli en dan bolj zgodaj tableto A) ter na koncu še obdobje r dni brez konflikta. Če smo zadnji konflikt pred temi r dnevi razrešili tako, da smo vzeli dan bolj zgodaj tableto B, je skupno število tablet enako:

$$T = \lfloor (n + v)/a \rfloor + \lfloor (n + u + 1)/b \rfloor. \qquad (1)$$

Če pa smo takrat dan bolj zgodaj vzeli tableto A, je skupno število tablet:

$$T = \lfloor (n + v + 1)/a \rfloor + \lfloor (n + u)/b \rfloor. \qquad (2)$$

Pri tem je prva možnost dopustna le, če je $0 \leq r < c_1$, druga pa le, če je $0 \leq r < c_2$.[31]

[31] O pravilnosti teh formul za T se lahko prepričamo tudi bolj formalno; oglejmo si to na primeru formule (1). Naj bo T_1 (oz. T_2) število porabljenih tablet tipa A (oz. B); potem bo $T = T_1 + T_2$. V prvih c_0 dneh vzamemo c_0/a tablet tipa A; v vsakem c_1-dnevnem obdobju c_1/a; v vsakem c_2-dnevnem obdobju $(c_2+1)/a$; in v r-dnevnem obdobju na koncu še $\lfloor r/a \rfloor$. Pišimo $k = \lfloor r/a \rfloor$ in $o = r - ka = r \bmod a$ (tako da je $0 \leq o < a$); potem je $T_1 = c_0/a + u \cdot c_1/a + v \cdot (c_2+1)/a + k = (c_0 + c_1 u + (c_2+1)v + ka)/a$. Upoštevajmo, da je $n = c_0 + c_1 u + c_2 v + r$, pa dobimo $T_1 = (n + v - o)/a$. Vsota $n + v$ je naprej enaka $c_0 + c_1 u + (c_2+1)v + ka + o$; vsi ti členi razen zadnjega so večkratniki

Posvetimo se zdaj možnosti (1); razmislek za (2) je analogen. Izbrati si moramo u in v tako, da bo T čim manjša (pri tem pa seveda $r = n - c_0 - c_1u - c_2v$ ne sme biti $\geq c_1$). Izraz $\lfloor (n+v)/a \rfloor$ ima pri $v = 0, 1, \ldots, a - 1 - (n \bmod a)$ ves čas enako vrednost; prvič naraste (za 1) šele pri naslednjem v, to je $a - (n \bmod a)$; naslednjič naraste (spet za 1) pri $2a - (n \bmod a)$ in tako naprej. Najmanjši v, o katerem je smiselno razmišljati, je torej $v_0 = a - 1 - (n \bmod a)$, saj ne bomo porabili nič manj tablet tipa A, če vzamemo še manjši v. Naslednji najmanjši v, o katerem je smiselno razmišljati, je $v = v_0 + a$, saj bomo pri vseh v od $v_0 + 1$ do $v_0 + a$ porabili enako število tablet tipa A. Omejimo se torej na v-je oblike $v = v_0 + Va$ za $V \geq 0$; pri takem v porabimo $\lfloor (n+v)/a \rfloor = \lfloor n/a \rfloor + V$ tablet tipa A.

Podoben razmislek za izraz $\lfloor (n+1+u)/b \rfloor$ nam pokaže, da se lahko omejimo na $u = u_0 + Ub$ za $U \geq 0$ in $u_0 = b - 1 - ((n+1) \bmod b)$; pri takem u porabimo $\lfloor (n+1+u)/b \rfloor = \lfloor (n+1)/b \rfloor + U$ tablet tipa B.

Skupna poraba tablet je torej zdaj

$$T = \lfloor n/a \rfloor + \lfloor (n+1)/b \rfloor + U + V.$$

Na prva dva člena te vsote seveda ne moremo vplivati; minimizirati moramo torej $U + V$. Pri tem moramo še vedno upoštevati omejitev

$$n \leq c_0 + c_1u + c_2v + r \quad \text{in} \quad 0 \leq r < c_1. \tag{3}$$

Tu torej nismo zahtevali, da mora biti naš scenarij dolg točno n dni, ampak sme biti tudi daljši, kajti ko se omejimo na u-je oblike $u = u_0 + Ub$ in v-je oblike $v = v_0 + Va$, lahko naš scenarij podaljšujemo le v korakih po c_1b ali c_2a dni, torej ni nujno, da nam bo uspelo zadeti takega, kjer bi skupaj z r dnevi na koncu (za neki $0 \leq r < c_1$) dobili točno n dni. Če dobimo scenarij z več kot n dnevi, ni s tem nič narobe, dokler nam to ne poveča vsote $U + V$, ki jo minimiziramo.

Ker r nič ne vpliva na našo kriterijsko funkcijo, je smiselno vzeti največjega možnega, $r = c_1 - 1$; tako bomo najlaže izpolnili v (3) pogoj za n. Tega lahko zdaj zapišemo kot

$$n \leq c_0 + c_1(u_0 + Ub) + c_2(v_0 + Va) + r \tag{4}$$

in ga predelamo v

$$\big(n - c_0 - c_1u_0 - c_2(v_0 + Va) + r\big)/(c_1b) \leq U.$$

Če si torej izberemo neki konkretni V, je najmanjši primerni U enak $\lceil \cdot \rceil$ od leve strani te neenačbe. Večjega U od tega seveda nima smisla uporabljati, saj hočemo minimizirati vsoto $U + V$. Pri tako izbranem U potem vsota $U + V$ postane

$$\begin{aligned} &\Big\lceil \big(n - c_0 - c_1u_0 - c_2(v_0 + Va) + r\big)/(c_1b) \Big\rceil + V \\ =&\Big\lceil \big(n - c_0 - c_1u_0 - c_2v_0 + r + V \cdot (c_1b - c_2a)\big)/(c_1b) \Big\rceil. \end{aligned}$$

---

a (ker so c_0, c_1 in $c_2 + 1$ večkratniki a), zato je $(n+v) \bmod a = o$. Uporabimo to v formuli za T_1, pa dobimo $T_1 = (n + v - o) = ((n+v) - ((n+v) \bmod a))/a = \lfloor (n+v)/a \rfloor$. Podoben razmislek za tablete tipa B bi nas pripeljal do rezultata $T_2 = \lfloor (n+u+1)/b \rfloor$; vsota $T_1 + T_2$ pa je ravno tisto, kar smo za T zapisali v formuli (1). Na prav tak način lahko obravnavamo tudi formulo (2).

V številu je linearna funkcija V-ja, njen smerni koeficient pa je $c_1 b - c_2 a$. Če je ta pozitiven, moramo torej (da bomo minimizirali to funkcijo) vzeti najmanjši možni V, če pa je smerni koeficient negativen, moramo vzeti največji možni V. Tega nam navzgor omejuje dejstvo, da je formula, po kateri smo izračunali U iz V, smiselna le, dokler nam ne začne predlagati negativnih U. Najmanjši možni U je $U = 0$ in če ga vstavimo v neenačbo (4), jo lahko predelamo v

$$\big(n - c_0 - c_1 u_0 - c_2 v_0 - r\big)/(c_2 a) \le V.$$

Največji možni V je torej enak $\lceil \cdot \rceil$ leve strani te zadnje neenačbe; recimu mu $V_{\max}$. Paziti moramo le na to, da če na ta način dobimo negativen V (kar se lahko zgodi, če sta npr. $c_1 u_0$ in $c_2 v_0$ velika), moramo seveda vzeti $V_{\max} = 0$ (takrat bo potem tudi $U = 0$).

Povzemimo našo rešitev s psevdokodo:

(* *Možnost 1: pri zadnjem konfliktu vzamemo B dan bolj zgodaj.* *)
$u_0 = b - 1 - ((n+1) \bmod b)$; $v_0 := a - 1 - (n \bmod a)$; $r := c_1 - 1$;
$V_{\max} := \max\{0, \lceil (n - c_0 - c_1 u_0 - c_2 v_0 - r)/(c_2 a) \rceil\}$;
if $c_1 b \ge c_2 a$ **then** $V := 0$ **else** $V := V_{\max}$;
$U := \max\{0, \lceil (n - c_0 - c_1 u_0 - c_2 (v_0 + Va) + r)/(c_1 b) \rceil\}$;
$T^{(1)} := \lfloor n/a \rfloor + \lfloor (n+1)/b \rfloor + U + V$;

(* *Možnost 2: pri zadnjem konfliktu vzamemo A dan bolj zgodaj.* *)
$u_0 = b - 1 - (n \bmod b)$; $v_0 := a - 1 - ((n+1) \bmod a)$; $r := c_2 - 1$;
izračunaj $V_{\max}$, V in U enako kot pri prvi možnosti;
$T^{(2)} := \lfloor (n+1)/b \rfloor + \lfloor n/b \rfloor + U + V$;

return $\min\{T^{(1)}, T^{(2)}\}$;

Časovna zahtevnost te rešitve je $O(\log n)$ in še to le zaradi Evklidovega algoritma; vse ostalo vzame le $O(1)$ časa.

L. Igra

Pri tej nalogi moramo le pazljivo slediti navodilom, odsimulirati potek igre in izpisati končno stanje. Med simulacijo vzdržujemo seznam kart na kupu, seznam kart v roki in sezname kart za vse štiri vrste, v katere jih odlagamo. V vsaki iteraciji glavne zanke najprej pobiramo karte, dokler jih nimamo v roki osem ali pa se kup ne izprazni; nato izvedemo dve potezi. V spodnji rešitvi uporabljamo spremenljivko *ok*, da označimo, če smo potezo res uspeli narediti; če je nismo, se prekine tudi glavna zanka in igre je konec.

Pri vsaki potezi najprej pogledamo, če bi se dalo uporabiti vzvratni trik; pri tem gremo z zanko po kartah v roki (od leve proti desni) in pri vsaki karti po vrsticah (od zgoraj navzdol); čim naletimo na možno potezo z vzvratnim trikom, jo tudi izvedemo. Če pa prideta tidve zanki do konca, ne da bi izvedli potezo, pogledamo nato z novima dvema zankama, ali bi se dalo izvesti običajno potezo; tu moramo iti spet po kartah v roki (od leve proti desni) in pri vsaki karti po vrsticah (od spodaj navzdol), pri tem pa preverjamo, ali bi se karto dalo dodati v tisto vrstico (pri V_1 in V_2 mora biti nova karta večja od zadnje v vrstici, pri V_3 in V_4 pa manjša) in če da, kakšna bi bila pri tem razlika med novo karto in doslej zadnjo v vrstici.

Najmanjšo razliko si zapomnimo (spremenljivka $d^\star$), ob tem pa tudi karto in vrstico, kjer smo to razliko dosegli ($k^\star$ in $i^\star$). Novo razliko si zapomnimo le, če je strogo manjša od najmanjše doslej; tako poskrbimo, da če minimalno razliko dosežemo pri več kartah, uporabimo med njimi najbolj levo in jo potem damo v najbolj zgornjo možno vrstico.

```
naj bo K seznam kart na kupu (iz vhodnih podatkov);
naj bodo R, V1, V2, V3, V4 prazni seznami;
v V1 in V2 dodaj element 1, v V3 in V4 pa element 100;
do:
  (* Poberimo še nekaj kart. *)
  while K ni prazen and |R| < 8:
    pobriši karto z začetka K in jo dodaj na konec seznama R;
  (* Izvedimo dve potezi. *)
  for poteza := 1 to 2:
    ok := false; (* pove, ali smo naredili potezo *)
    (* Poskusimo uporabiti vzvratni trik. *)
    za vsako karto k ∈ R, od leve proti desni:
      for i := 1 to 4:
        če je k za 10 manjša (pri i = 1, 2) oz. večja (pri i = 3, 4)
        od trenutno zadnje karte v Ri:
          dodaj k na konec Ri in jo pobriši iz K;
          ok := true; break;
      if ok then break;
    if ok then continue;
    (* Poskusimo narediti običajno potezo. *)
    k* := −1; i* := −1; d* := ∞;
    za vsako karto k ∈ R, od leve proti desni:
      for i := 1 to 4:
        d := k − (zadnja karta v Ri); if i ≥ 3 then d := −d;
        (* Pri V1 in V2 mora biti nova karta večja od doslej zadnje, *)
        if d ≤ 0 then continue;          (* pri V3,4 pa manjša. *)
        if d < d* then d* := d, k* := k, i* := i;
    if k* < 0 then break;
    pobriši karto k* iz R in jo dodaj na konec seznama Vi*; ok := true;
while ok;
izpiši K, R, V1, V2, V3 in V4;
```

REŠITVE NALOG POSKUSNEGA TEKMOVANJA

X. Kronogram

Ta naloga je zelo preprosta; ni treba drugega, kot da beremo vhodne podatke vrstico po vrstico, v vsaki prebrani vrstici pa gremo po znakih, pri vsakem znaku preverimo,

če predstavlja kakšno številsko vrednost, in te vrednosti seštevamo, na koncu vrstice pa vsoto izpišemo.

Tudi ni nujno, da beremo vhod po vrsticah; lahko bi ga brali znak po znak in seštevali številske vrednosti znakov, ko pa naletimo na znak za konec vrstice, izpišemo trenutno vsoto in jo postavimo na 0.

Lahko bi si tudi vnaprej pripravili tabelo, v kateri bi za vsak možni znak hranili njegovo številsko vrednost (ali 0, če ta znak nima številske vrednosti); nato bi pri vsakem iz vhodne datoteke prebranem znaku preprosto odčitali njegovo vrednost iz te tabele in jo prišteli k naši vsoti.[32]

Y. Permutacije

Enovrstični zapis permutacije si lahko predstavljamo kot tabelo (*array*), v kateri element $\pi[i]$ (za $i = 1, \ldots, n$) vsebuje vrednost $\pi(i)$, v katero permutacija preslika število i. Naloga sicer zahteva, da na podlagi tabele inverzij izpišemo permutacijo v cikličnem zapisu, vendar bo lažje, če najprej pretvorimo tabelo inverzij v enovrstični zapis, nato pa le-tega v ciklični zapis.

Vrednost b_1 v tabeli inverzij pove, koliko je v enovrstičnem zapisu permutacije levo od števila 1 takih števil, ki so večja od 1. Toda večja od 1 so *vsa* ostala števila, torej b_1 čisto preprosto pove, koliko števil je levo od števila 1; z drugimi besedami, b_1 nam pove, na katerem položaju v enovrstični predstavitvi stoji število 1. Ali še drugače: $\pi[b_1 + 1] = 1$.

Vrednost b_2 nam potem pove, koliko je v enovrstičnem zapisu levo od števila 2 takih števil, ki so večja od 2, to pa so vsa razen števila 1. Lahko si predstavljamo, da so v tabeli π vse celice še prazne, razen celice $\pi[b_1 + 1]$, v katero smo že vpisali vrednost 1; potem nam b_2 pove, da moramo vrednost 2 vpisati v $(b_2 + 1)$-vo prazno celico (gledano z levega konca). Podoben razmislek nam nato pove, da moramo v naslednjem koraku vpisati vrednost 3 v $(b_3 + 1)$-vo prazno celico in tako naprej.

V splošnem torej na k-tem koraku vpišemo vrednost k v $(b_k + 1)$-vo prazno celico. Toda če bomo iskali b_k-to celico tako, da bomo šli v zanki po tabeli π od leve proti desni in šteli prazne celice, bo imel naš postopek na koncu časovno zahtevnost $O(n^2)$, to pa je za našo nalogo že preveč. Boljšo rešitev dobimo, če v mislih razdelimo tabelo π na približno $\sqrt{n}$ blokov s približno $\sqrt{n}$ celicami v vsakem bloku; za vsak blok vzdržujmo števec praznih celic v njem. S pomočjo teh števcev lahko, ko iščemo b_k-to prazno celico, prvih nekaj blokov v celoti preskočimo, podrobneje (celico po celico) pa pregledamo šele tisti blok, ki vsebuje b_k-to prazno celico. Tako imamo pri vsakem k po $O(\sqrt{n})$ dela in časovna zahtevnost celotne rešitve je $O(n^{1,5})$. To je bilo za naše tekmovanje že dovolj dobro.

(* *Inicializacija. $P[i]$ pomeni število praznih celic v i-tem bloku.* *)
$B := \lfloor\sqrt{n}\rfloor$; $N := \lceil n/B \rceil$; (* *Imamo N blokov s po B celicami.* *)
for $i := 1$ **to** N **do** $P[i] := B$;

for $k := 1$ **to** n:
  (* *Poiščimo blok, ki vsebuje b_k-to prazno celico v π.* *)

[32] Mimogrede, latinski primer kronograma v besedilu naloge je napis iz leta 2000 na zvoniku mariborske stolnice (Barbara Damjan (*ur.*), *Posodobitve pouka v gimnazijski praksi — latinščina*, Zavod RS za šolstvo, Ljubljana 2010, str. 29).

```
i := 1; β := 0; (* V prvih i − 1 blokih je β praznih celic, kar je ≤ b_k. *)
while β + P[i] ≤ b_k do β := β + P[i], i := i + 1;

(* (b_k + 1)-va prazna celica leži v i-tem bloku. *)
j := B · (i − 1) + 1; (* Od prvih j − 1 celic je β praznih, kar je ≤ b_k. *)
while π[j] ni prazna or β < b_k:
  if π[j] ni prazna then β := β + 1;
  j := j + 1;

π[j] := k;          (* π[j] je (b_k + 1)-va prazna celica; vpišimo k tja. *)
P[j] := P[j] + 1;   (* Popravimo števec praznih v tem bloku. *)
```

Še boljšo rešitev dobimo, če nad tabelo π zgradimo drevo intervalov: to je skupina tabel Π_ℓ za $0 \le \ell \le L := \lfloor \log_2 n \rfloor$, pri čemer je Π_ℓ dolga $\lfloor n/2^\ell \rfloor$ elementov in v njej element $\Pi_\ell[i]$ pove, koliko je praznih celic $\pi[j]$ na območju $2^\ell \cdot (i-1) < j \le 2^\ell \cdot j$ (na začetku inicializiramo vse elemente $\Pi_\ell[i]$ na 2^ℓ, ker so vse celice prazne). Ko iščemo (b_k+1)-vo prazno celico, moramo na vsakem nivoju ℓ pregledati le en element tabele Π_ℓ; in ko (b_k+1)-vo prazno celico najdemo ter vanjo vpišemo število k, moramo potem na vsakem nivoju ℓ zmanjšati en element tabele Π_ℓ za 1, ker se je število praznih elementov tam zmanjšalo. Tako imamo pri vsakem k po $O(\log n)$ dela in časovna zahtevnost te rešitve je $O(n \log n)$.

```
(* Inicializacija. *)
for ℓ := 0 to L do for i := 1 to ⌊n/2^ℓ⌋ do Π_ℓ[i] := 2^ℓ;

for k := 1 to n:
  i := 1; β := 0; (* Poiščimo (b_k + 1)-vo prazno celico v π. *)
  for ℓ := L downto 0:
    i := 2 · i − 1;
    (* Od celic π[j] za 1 ≤ j ≤ 2^ℓ · (i − 1) je praznih β celic, kar je ≤ b_k.
       Od celic π[j] za 1 ≤ j ≤ 2^ℓ · i pa je praznih več kot b_k celic. *)
    if i ≤ ⌊n/2^ℓ⌋ and β + Π_ℓ[i] ≤ b_k:
      β := β + Π_ℓ[i]; i := i + 1;
  (* i je (b_k + 1)-va prazna celica; vpišimo vanjo k in popravimo tabele Π_ℓ. *)
  π[i] := k;
  for ℓ := 0 to L:
    Π_ℓ[i] := Π_ℓ[i] − 1; i := ⌈i/2⌉;
```

Ko smo (po enem ali drugem postopku) pripravili tabelo π, torej enovrstični zapis permutacije, jo moramo izpisati v cikličnem zapisu. To lahko storimo tako, da gremo v zanki po π od leve proti desni; ko opazimo kak element i, ki ga še nismo izpisali, sledimo povezavam $i \to \pi[i] \to \pi[\pi[i]] \to \ldots$, dokler ne pridemo nazaj v i; to je en cikel in njegove elemente lahko zdaj izpišemo. (To, katere elemente smo že izpisali, si lahko označujemo tako, da jih v tabeli π brišemo oz. postavljamo na -1.) Ker z izpisom naslednjega cikla vedno začnemo pri najbolj levem še neizpisanem elementu, bodo cikli vedno izpisani tako, da se bo vsak cikel začel s svojim najmanjšim elementom, cikli pa bodo urejeni naraščajoče glede na najmanjši element — torej prav tako, kot naloga zahteva.

```
for i := 1 to n do if π[i] ≠ −1:
```

if $i > 1$ **then** izpiši presledek; (* *presledki med cikli* *)
izpiši oklepaj in i; $j := \pi[i]$; $\pi[i] := -1$;
while $j \neq i$:
  izpiši presledek in j; $t := \pi[j]$; $\pi[j] := -1$; $j := t$;
izpiši zaklepaj;

Z. Sokoban

Naj bo V množica tistih polj naše mreže, ki so po samih prostih poljih (med slednje štejemo tudi odlagališča in polja, na katerih stojijo škatle) dosegljiva iz polja, na katerem sprva stoji igralec. Teh polj je največ 36, ker se igra dogaja na mreži velikosti največ 8×8 in ker naloga zagotavlja, da stoji igralec na območju, ki je vse naokoli obdano z zidovi.

Nalogo lahko rešujemo s preiskovanjem prostora stanj, pri čemer stanje igre opišemo s parom (B, p), kjer je $B \subseteq V$, $|B| = b$ (število škatel, ki je hkrati tudi število odlagališč) in $p \in V - B$; množica B pove položaj škatel, p pa položaj igralca. Naloga zahteva rešitev z najmanj premikanji škatel; premik igralca nas nič ne stane, če pri tem ne rine škatle. Zato definirajmo povezave med našimi stanji takole: igralec se lahko najprej nekaj časa premika po poljih iz $V - B$, nato pa stopi na eno od polj iz B in pri tem porine škatlo, ki je prej stala na tem polju. Vse to bomo šteli za en korak pri pregledovanju prostora stanj. Iz posameznega stanja je torej možnih več nadaljevanj odvisno od tega, katero škatlo premaknemo in v katero smer. To, kam je iz p sploh mogoče priti brez rinjenja škatel, bomo raziskali z iskanjem v širino po naši mreži (oz. po množici V).

Naloga potem sprašuje, kako iz začetnega stanja priti s čim manj koraki (rinjenji škatel) v eno od stanj oblike (O, p), kjer je O množica odlagališč, p pa je lahko poljubno polje iz $V - O$; to so torej stanja, kjer so vse škatle na odlagališčih. Ker gledamo le število korakov (vsi koraki veljajo za enako dolge), lahko uporabimo iskanje v širino po prostoru stanj. Tako imamo torej dvoje iskanj v širino, vgnezdenih eno v drugem: zunanje po prostoru stanj in notranje po poljih na mreži V.

$Q :=$ prazna vrsta; $d :=$ prazen slovar oz. razprševalna tabela;
$z :=$ začetno stanje (glede na položaj škatel in igralca iz vhodnih podatkov);
if so v z vse škatle že na odlagališčih **then return** 0;
dodaj z v Q; $d[z] := 0$;
while Q ni prazna:
  $(B, p) :=$ stanje na začetku vrste Q; pobriši ga iz Q;
  $Q' :=$ prazna vrsta; $d' :=$ prazna množica;
  dodaj p v Q' in v d';
  while Q' ni prazna:
    $r :=$ polje z začetka vrste Q'; pobriši ga iz Q';
    za vsakega ne-zazidanega soseda s polja r:
      if $s \neq B$: (* *lahko se premaknemo z r na s* *)
        if $s \notin d'$ **then** dodaj s v d' in na konec Q';
        continue;
      (* *Sicer je na s škatla. Ali jo lahko porinemo?* *)
      $t :=$ sosed s-ja v isti smeri, v kateri je s sosed r-ja;
      if $t \in B$ **or** t je zazidano **then continue**; (* *ne moremo* *)

(* *Lahko se premaknemo z r na s in pri tem škatlo s s porinemo na t.* *)
$B' := B - \{s\} \cup \{t\}$; **if** $B' = O$ **then return** $d[B, p] + 1$;
if je (B', s) že v slovarju in je $d[B', s] \leq d[B, p + 1]$ **then continue**; $(\star)$
$d[B', s] := d[B, p] + 1$; dodaj (B', s) na konec vrste Q;

Pri preiskovanju prostora se lahko ustavimo takoj, ko najdemo pot do kakšnega polja, kjer so vse škatle na odlagališčih ($B' = O$). Če bi se glavna zanka iztekla, ker bi se Q izpraznila, pa bi vedeli, da vseh škatel ni mogoče premakniti na odlagališča; toda naloga tako ali tako zagotavlja, da se to ne bo zgodilo.

Kakšna je časovna zahtevnost te rešitve? Pri vsakem stanju porabimo v najslabšem primeru $O(|V|)$ časa, da določimo vsa možna nadaljevanja. Možnih stanj je kvečjemu $\binom{|V|}{b} \min\{4b, |V|-b\}$, pri čemer prvi faktor pove, na koliko načinov si lahko izberemo izmed $|V|$ polj položaje b škatel, drugi pa, na koliko načinov si lahko potem med preostalimi $|V| - b$ polji izberemo položaj igralca, kjer smo upoštevali tudi to, da (razen v začetnem stanju) igralec vedno stoji zraven ene od škatel (namreč tiste, ki jo je v zadnjem koraku porinil). V najslabšem primeru je to $\binom{36}{4} \cdot 16 = 913\,920$; v praksi nam sicer ni uspelo sestaviti primera, pri katerem bi opisana rešitev preiskala več kot pribl. 487 tisoč stanj, torej dobro polovico vseh možnih.

Rešitev lahko še malo izboljšamo, če za preiskovanje prostora stanj uporabimo algoritem A* namesto iskanja v širino; pri tem kot lahko hevristično oceno (spodnjo mejo) dolžine najkrajše poti (v prostoru stanj) od nekega stanja do najbližjega končnega stanja (torej takega, pri katerem so vse škatle na odlagališčih) uporabimo vsoto dolžin najkrajših poti (na mreži) od vsake škatle trenutnega stanja do njej najbližjega odlagališča. Pri naših pokusih je ta rešitev pri najtežjih testnih primerih pregledala pribl. 272 tisoč stanj.

Še ena koristna izboljšava pa je naslednja: če imata dve stanji (B, s) in (B, s') enak razpored škatel, vendar različen položaj igralca, in če lahko igralec pri tem razporedu škatel pride od s do s', ne da bi premaknil kakšno škatlo, potem bodo nadaljnji koraki iz teh dveh stanj popolnoma enaki in ni nobene koristi od tega, da ju sploh obravnavamo kot dve različni stanji. Zato lahko pred vrstico $(\star)$ najprej pogledamo (s še enim iskanjem v širino), katera polja so dosegljiva iz s, če se omejimo na premike znotraj $V - B'$; med temi polji recimo s' tistemu z najmanjšo številko; in potem v zadnjih dveh vrsticah gornje psevdokode uporabimo s' namesto s.

S to izboljšavo se je število pregledanih stanj pri najtežjem testnem primeru zmanjšalo na približno 59 tisoč pri navadnem iskanju v širino oz. 37 tisoč pri preiskovanju z algoritmom A*).[33]

Naloge so sestavili: denormalizacija, igra, kronogram, permutacije, sokoban — Nino Bašić; kombinacijske ključavnice, požrešni predali, razbojniki — Tomaž Hočevar; razlike — Josip Klepec; hipoteka, spretno tabletno — Vid Kocijan; ozvezdja, razgozdovanje — Maks Kolman in Tomaž Hočevar; pranje denarja — Jure Slak; vrivanje — Janez Brank.

[33] Rešitev z manj kot eksponentno zahtevnostjo se pri tej nalogi težko nadejamo, saj je že vprašanje, ali je pri danem začetnem stanju mreže sploh mogoče spraviti vse škatle na odlagališča, PSPACE-težko (Joseph C. Culberson, *Sokoban is PSPACE-complete*, Technical Report TR 97-02, Dept. of Comp. Science, Univ. of Alberta, April 1997; objavljeno tudi v *Fun with Algorithms: Proceedings of the Int. Conf.* (FUN 1998), str. 65–76). Obstaja pa precej idej na temo tega, kako čim učinkoviteje preiskovati prostor možnih stanj igre; gl. npr. diplomsko nalogo Jake Klančarja *Reševanje problema Sokoban* (FRI, 2016) in tam navedeno literaturo.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2020

1. Futoški

Vrstice oštevilčimo od 0 na vrhu do $n-1$ na dnu mreže, stolpce pa tudi od 0 na levem do $n-1$ na desnem robu mreže. Položaj posameznega polja lahko zdaj opišemo s parom (x, y), ki pove, da leži to polje na preseku vrstice y in stolpca x.

Stanje mreže lahko predstavimo s tremi tabelami: eno za števila v poljih (ta tabela ima n vrstic in n stolpcev, enako kot naša mreža); eno za znake $<$ in $>$ med sosednjima poljema v isti vrstici (ta tabela ima n vrstic in $n-1$ stolpcev); in eno za znake $<$ in $>$ med sosednjima poljema v istem stolpcu (ta tabela ima $n-1$ vrstic in n stolpcev). V spodnji rešitvi torej stevila[y][x] pomeni število v polju (x, y); znakiVrstice[y][x] je znak med poljema (x, y) in $(x+1, y)$; znakiStolpci[y][x] pa je znak med poljema (x, y) in $(x, y+1)$. Če med dvema sosednjima poljema ni nobenega od znakov $<$ ali $>$, pričakujemo na ustreznem mestu v znakiVrstice ali znakiStolpci presledek ali poljuben drug znak, ki ni niti `'<'` niti `'>'`.

Da preverimo, če so vsa števila v vrstici različna, pojdimo v zanki po vrsticah in nato v gnezdeni zanki po poljih trenutne vrstice. V tabeli videno vzdržujemo za vsako število od 1 do n podatek o tem, v kateri vrstici smo ga nazadnje videli; tako lahko pri vsakem polju mreže preverimo, ali smo število v njem videli že kdaj prej v trenutni vrstici (kar pomeni, da mreža ni pravilno izpolnjena). (Za vsak primer prej preverimo še, ali je to število sploh od 1 do n.)

Poleg tega pri vsakem polju tudi preverimo, ali je število na njem v takem razmerju (večje ali manjše) s svojim levim sosedom, kot ga zahteva ustrezni znak iz tabele znakiVrstice. Pri tem si spodnja rešitev pomaga z vgnezdenim podprogramom Primerjaj, ki kot parametra dobi števili iz obeh polj ter znak med njima.

Nato enako kot za vrstice naredimo še za stolpce. Če pri nobenem od teh preverjanj ne odkrijemo napake, lahko na koncu zaključimo, da je mreža pravilno izpolnjena.

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

bool Preveri(const vector<vector<int>> &stevila,
             const vector<vector<char>> &znakiVrstice,
             const vector<vector<char>> &znakiStolpci)
{
  int n = stevila.size();
  // Preverimo, ali so vsa števila z območja od 1 do n.
  for (auto &v : stevila) for (int c : v) if (c < 1 || c > n) return false;
  // Preverimo, ali so v vsaki vrstici vsa števila različna.
  vector<int> videno(n + 1, -1);
  for (int y = 0; y < n; y++) for (int x = 0; x < n; x++)
    if (int c = stevila[y][x]; videno[c] == y) return false;
    else videno[c] = y;
  // Podprogram, ki preveri, ali med številoma a in b velja relacija c.
  auto Primerjaj = [] (char c, int a, int b) {
    return c == '<' ? (a < b) : c == '>' ? (a > b) : true; };
  // Preverimo, ali veljajo omejitve med sosednjimi znaki po vrsticah.
```

```
  for (int y = 0; y < n; y++) for (int x = 0; x < n − 1; x++)
    if (! Primerjaj(znakiVrstice[y][x], stevila[y][x], stevila[y][x + 1])) return false;

  // Preverimo, ali so v vsakem stolpcu vsa števila različna.
  for (int x = 0; x < n; x++) for (int y = 0; y < n; y++)
    if (int c = stevila[y][x]; videno[c] == n + x) return false;
    else videno[c] = n + x;

  // Preverimo, ali veljajo omejitve med sosednjimi znaki po stolpcih.
  for (int y = 0; y < n − 1; y++) for (int x = 0; x < n; x++)
    if (! Primerjaj(znakiStolpci[y][x], stevila[y][x], stevila[y + 1][x])) return false;

  // Če smo prišli do sem, je mreža pravilno izpolnjena.
  return true;
}
```

2. Varnostnik

Za vsako pisarno, pa tudi za vsako skupino desetih pisarn, bomo vdrževali dve celoštevilski vrednosti: trenutno zasedenost (število ljudi v pisarni ali skupini pisarn) in omejitev (maksimalno dovoljeno zasedenost). Te podatke hranimo v dveh vektorjih, pisarne in omejitve; pri branju vhodnih podatkov bodimo pozorni na to, da gredo tam številke pisarn od 1 naprej, indeksi v naša dva vektorja pa od 0 naprej, zato bomo vsako prebrano številko pisarne najprej zmanjšali za 1. Na začetku inicializiramo vse zasedenosti na 0, omejitve pa preberemo s standardnega vhoda; nato pa v zanki beremo podatke o prihodih in odhodih. Ko preberemo odhod iz pisarne, moramo le zmanjšati zasedenost tiste pisarne in njene skupine; ko pa preberemo prihod, moramo najprej preveriti, če ni morda zasedenost te pisarne ali njene skupine že dosegla maksimalno dovoljeno; če jo je, to tudi izpišemo, sicer pa lahko prišlek vstopi, mi pa moramo povečati zasedenost pisarne in skupine za 1.

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

int main()
{
  // Struktura Par hrani trenutno in maksimalno zasedenost pisarne (ali skupine pisarn).
  struct Par { int zasedenost = 0, omejitev; };

  // Preberimo število pisarn in omejitve.
  int n; cin >> n;
  vector<Par> pisarne(n), skupine((n + 9) / 10);
  for (auto &p : pisarne) cin >> p.omejitev;
  for (auto &s : skupine) cin >> s.omejitev;

  // Obdelajmo prihode in odhode.
  int m; cin >> m;
  while (m−− > 0)
  {
    // Preberimo naslednji dogodek.
    int x; cin >> x;
    // Ali je to odhod in na katero pisarno se nanaša?
    bool odhod = (x < 0); x = (odhod ? −x : x) − 1;
    auto &p = pisarne[x], &s = skupine[x / 10];

    // Če je odhod, moramo le zmanjšati podatek o zasedenosti.
    if (odhod) { −−p.zasedenost; −−s.zasedenost; }
```

```
    // Pri prihodu preverimo, če bi prekoračil kakšno omejitev.
    else if (p.zasedenost >= p.omejitev) cout << "POLNA PISARNA" << endl;
    else if (s.zasedenost >= s.omejitev) cout << "POLNA SKUPINA" << endl;

    // Če lahko vstopi, povečajmo podatek o zasedenosti.
    else { ++p.zasedenost; ++s.zasedenost; cout << "OK" << endl; }
  }
  return 0;
}
```

3. Funkcije

Ker so lahko v nizih, s katerimi imamo opravka, klici funkcij rekurzivno vgnezdeni eden v drugem, je naloga primerna za reševanje z rekurzijo. V naši spodnji rešitvi opravi večino dela podprogram ObdelajKlic, ki prebere iz vhodnega niza naslednji klic funkcije, pri tem pa za vsakega od njegovih parametrov, če je le-ta tudi klic (in ne številska konstanta), rekurzivno kliče samega sebe. Pri tem mu spremenljivka p označuje trenutni položaj v nizu (kaže na naslednji znak, ki ga moramo obdelati). Funkcija ObdelajKlic vrne **false**, če naleti v nizu na sintaktično napako, sicer pa **true**.

Številu parametrov rečemo tudi *mestnost* (angl. *arity*) funkcije. Da bomo lahko opazili nedoslednosti v nizu, torej primere tega, da se isto funkcijo večkrat kliče z različnim številom parametrov, moramo nekje hraniti podatke o imenih in mestnosti doslej opaženih funkcij; spodnja rešitev ima v ta namen razpršeno tabelo mestnost. Ko beremo klic funkcije iz vhodnega niza, moramo pri tem šteti parametre in ko pridemo do konca klica (do zaklepaja), lahko preverimo, ali smo funkcijo z enakim imenom prej že videli in ali je imela takrat enako mestnost kot zdaj; če ne, lahko takoj zaključimo, da niz ni dosleden (spremenljivka dosleden v spodnji rešitvi). Če pa funkcije s tem imenom prej še nismo videli, si jo zapomnimo zdaj (in jo vpišimo v slovar mestnost).

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

struct Obdelava
{
  const char *p; // Kaže na naslednji znak, ki ga moramo obdelati (v vhodnem nizu).
  bool dosleden = true; // Ali smo že opazili kakšno nedoslednost?
  unordered_map<string, int> mestnost; // Imena in mestnosti doslej opaženih funkcij.

  void PreskociPresledke() { while (isspace(*p)) ++p; }

  // Prebere naslednji vgnezdeni klic in pusti „p“ takoj za zaklepajem.
  // Vrne false, če je prišlo do sintaktične napake.
  bool ObdelajKlic()
  {
    // Preberimo ime funkcije.
    PreskociPresledke();
    const char *r = p; while (islower(*r)) ++r;
    if (r <= p) return false;
    string ime(p, r);
    p = r; PreskociPresledke();

    // Naslednji znak mora biti oklepaj.
```

```
    if (*p != '(') return false; else ++p;
    // Preberimo parametre.
    int stParametrov = 0;
    while (true)
    {
      PreskociPresledke();
      // Parametri se končajo z zaklepajem.
      if (*p == ')') { ++p; break; }
      // Med parametri morajo biti vejice.
      if (stParametrov > 0) { if (*p != ',') return false;  ++p; PreskociPresledke(); }
      // Parameter je lahko število ali vgnezden klic.
      if (isdigit(*p)) while (isdigit(*++p)) ;
      else if (! ObdelajKlic()) return false;
      ++stParametrov;
    }
    // Zapomnimo si število parametrov te funkcije oz. preverimo, če smo
    // jo prej že videli z drugačnim številom parametrov.
    auto [it, jeNov] = mestnost.emplace(ime, stParametrov);
    if (! jeNov && it->second != stParametrov) dosleden = false;
    return true;
  }
};
```

Glavni podprogram mora le poklicati ObdelajKlic od začetka niza in preveriti, če je prišel do konca (kajti če je v nizu poleg klica še kaj drugega, je to tudi sintaktična napaka). Če je v nizu sintaktična napaka, to tudi sporočimo, sicer pa sporočimo, ali je dosleden ali ne.

```
enum Rezultat { Dosleden, Nedosleden, Napaka };
Rezultat AnalizirajIzraz(const char *p)
{
  Obdelava obdelava; obdelava.p = p;
  if (! obdelava.ObdelajKlic()) return Napaka;
  // Preverimo, da ni v nizu še česa drugega poleg klica.
  obdelava.PreskociPresledke(); if (*obdelava.p) return Napaka;
  return obdelava.dosleden ? Dosleden : Nedosleden;
}
```

4. Sestankovalna sova

Najprej si pripravimo podprogram, ki bo deloval podobno kot ObrniGlavo (torej ima nalogo obrniti glavo za določen kot), vendar bo pri tem tudi pazil na to, da glava ne bo prišla za več kot tri polne kroge ($3 \cdot 360$ stopinj) stran od nevtralnega položaja. Če bi prišlo do tega, moramo kót, za katerega bomo obrnili glavo, zamakniti za 360 stopinj (na primer: namesto za 60 stopinj v levo moramo glavo obrniti za 300 stopinj v desno). Trenutno smer glave hranimo v globalni spremenljivki.

```
extern void ObrniKamero(int kot);
int smerGlave = 0;

void Obrni(int kot)
{
  // Poskrbimo, da bo „kot“ na območju (−180, 180].
```

```
  kot = kot % 360;
  if (kot > 180) kot -= 360; else if (kot <= -180) kot += 360;
  // Če bi bila nova smer za več kot tri kroge od nevtralne smeri,
  // zamaknimo „kot“ za 360 stopinj.
  int novaSmer = smerGlave + kot;
  if (novaSmer > 3 * 360) kot -= 360;
  else if (novaSmer < -3 * 360) kot += 360;
  // Obrnimo kamero in si zapomnimo novo smer.
  ObrniKamero(kot);
  smerGlave += kot;
}
```

Ko imamo takšen podprogram, je potem lažja različica naloge res lahka. V zanki bomo klicali funkcijo SmerZvoka in potem zahtevali obrat glave za razliko med smerjo zvoka in trenutno smerjo glave; tako se bo glava obrnila ravno v smer, iz katere prihaja zvok. Funkcija Obrni bo poskrbela, da pri tem obračanju glava nikoli ne bo prišla za več kot tri kroge od nevtralne smeri.

```
extern int SmerZvoka();

int main()
{
  while (true) Obrni(SmerZvoka() - smerGlave);
  return 0;
}
```

Pri težji različici s tremi mikrofoni ne vemo točno, iz katere smeri prihaja zvok, pač pa lahko razmišljamo takole: če zaznava levi mikrofon močnejši zvok kot desni, je izvor zvoka levo od trenutne smeri glave in jo moramo premakniti malo v levo (recimo za eno stopinjo, saj je to najmanjši možni premik); podobno pa, če zaznava desni mikrofon močnejši zvok kot levi, premaknemo glavo malo v desno. Če zaznavata oba enako močan glas, to pomeni, da prihaja zvok bodisi prav iz tiste smeri, v katero je obrnjena glava (in glave v tem primeru ni treba obračati), bodisi iz nasprotne smeri (tu pa je vseeno, v katero smer obrnemo glavo); tidve možnosti ločimo tako, da preverimo, ali zaznava sprednji mikrofon močnejši zvok kot stranska dva (če da, je izvor zvoka spredaj, v smeri glave).

```
extern int Jakost(int mikrofon);

int main()
{
  while (true)
  {
    int L = Jakost(-1), S = Jakost(0), D = Jakost(1);
    if (L > D) Obrni(-1); // malo v levo
    else if (D > L) Obrni(1); // malo v desno
    else if (S < L) Obrni(smerGlave > 0 ? -1 : 1); // kamorkoli
  }
  return 0;
}
```

Ker se premikamo vedno v korakih po eno stopinjo, je možnih smeri glave le 360; lahko se zgodi, da je izvor zvoka med dvema takima smerema in se bo glava zato

ves čas premikala izmenično levo in desno po eno stopinjo. To bi lahko poskusili preprečiti tako, da bi si zapomnili zadnjih nekaj premikov; če smo na primer v zelo bližnji preteklosti že izvedli premik v eno smer in nato v drugo, potem ta hip morda ni pametno takoj izvesti še enega premika v prvo smer. Še ena možnost je, da bi pogoje za premikanje kamere malo zaostrili: izberimo si neko konstanto $\vartheta > 0$ in se potem pogoj „$L > D$“ (za premik v levo) spremenimo v „$L > D + \vartheta$“, podobno pa tudi ostala dva.

Pri tretji različici naloge, kjer imamo le levi in desni mikrofon, ne pa tudi sprednjega, je težava v tem, da ko sta jakosti na obeh mikrofonih enaki, ne moremo vedeti, ali prihaja zvok iz smeri glave ali iz nasprotne smeri. Recimo, da v prejšnji iteraciji naše zanke zvok ni prihajal točno iz nasprotne smeri glave; glavo smo potem premaknili za eno stopinjo bliže k smeri zvoka, torej tudi v naslednji iteraciji zvok ne prihaja iz nasprotne smeri glave. Do tega, da prihaja zvok iz nasprotne smeri, pride lahko torej le tako, da se izvor zvoka nenadoma premakne za 180 stopinj glede na sovo (npr. ker je dosedanji govorec utihnil, namesto njega pa se je oglasil nekdo iz ravno nasprotne smeri). Spodnja rešitev v primerih, ko je jakost na obeh mikrofonih izenačena, obrne glavo za eno stopinjo; če je prihajal zvok iz smeri glave, jo bomo v naslednji iteraciji tako ali tako obrnili nazaj, če pa je prihajal iz nasprotne smeri, bodo v naslednjih iteracijah sledili še nadaljnji premiki glave, ki jo bodo sčasoma obrnili v smer zvoka.

```
int main()
{
  while (true)
  {
    int L = Jakost(−1), D = Jakost(1);
    if (L > D) Obrni(−1); // malo v levo
    else if (D > L) Obrni(1); // malo v desno
    else Obrni(smerGlave > 0 ? −1 : 1); // kamorkoli
  }
  return 0;
}
```

Podobno kot pri prejšnji rešitvi to pomeni, da bo se bo glava včasih izmenično premikala po eno stopinjo levo in desno okrog smeri zvoka; lahko bi torej dodali logiko, ki premik v tretji vrstici („kamorkoli“) izvede šele, če je jakost na obeh mikrofonih izenačena dovolj dolgo; to bi zagotovilo, da bi bilo „cikcakanje“ okrog smeri zvoka redkejše in zato manj moteče, po drugi strani pa bi sicer sova počasneje odreagirala v primerih, ko se izvor zvoka nenadoma premakne za točno 180 stopinj, vendar lahko domnevamo, da so taki premiki v praksi tako ali tako redki.

5. Ultranet

Za vsako napravo x v omrežju bomo vzdrževali podatek o tem, kakšen status je imelo zadnje prejeto sporočilo oblike $\langle x; s\rangle$ in ob katerem času je prišlo; tema podatkoma recimo s_x (status) in t_x (čas).

Za vsako alarmno napravo a lahko zdaj razmišljamo takole: preglejmo a in njej nadrejene prenosne naprave ter poglejmo, katera od teh naprav (recimo ji x) ima največjo vrednost t_x — z drugimi besedami, za katero od njih imamo najnovejši podatek o stanju. Če je ta podatek (čas t_x) starejši od 5 sekund, je smiselno napravo

a šteti za nedosegljivo (to je načeloma znak, da smo izgubili stik z zgoščevalnikom, prek katerega gre pot od a do NC). Sicer pa lahko pripadajoči status s_x štejemo tudi za stanje alarmne naprave a: ena možnost je namreč ta, da je x zgoščevalnik in $s_x = 0$, torej nam zgoščevalnik sporoča, da je v njemu podrejenem delu mreže vse normalno, nobena naprava ni nedosegljiva ali v alarmu, torej je tudi a v normalnem stanju; druga možnost je, da je $x = a$ in $s_x = 2$, torej vemo, da je naprava a v stanju alarma; tretja možnost pa je, da je $s_x = 1$ (pri čemer je bodisi $x = a$ bodisi je x ena od a-ju nadrejenih naprav), kar pomeni, da je naprava x nedosegljiva, zato pa šteje za nedosegljivo tudi njej podrejena naprava a.

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

typedef enum { OK = 0, Nedosegljiva = 1, Alarm = 2 } Status;

struct Sporocilo
{
  int zg, lv, nv, an; // naslov
  int cas; Status status;
};

struct ZadnjeStanje
{
  int cas = 0; Status status = OK;

  // Skopira v trenutni zapis podatke iz Z, če so tam novejši.
  ZadnjeStanje& operator << (const ZadnjeStanje &Z) {
    if (Z.cas > cas) cas = Z.cas, status = Z.status;
    return *this; }
};

void Obdelaj(const vector<Sporocilo>& sporocila, int trenutniCas)
{
  const int nZG = 128, nLV = 512, nNV = 6, nAN = 8;
  ZadnjeStanje zadnje[nZG + 1][nLV + 1][nNV + 1][nAN + 1];

  // Za vsako napravo si zapomnimo zadnje sporočilo, ki se nanaša nanjo.
  // Predpostavimo, da so sporočila že urejena po času.
  for (auto &S : sporocila) {
    auto &Z = zadnje[S.zg][S.lv][S.nv][S.an];
    Z.cas = S.cas; Z.status = S.status; }

  // Pojdimo po vseh alarmnih napravah.
  for (int zg = 1; zg <= nZG; ++zg) for (int lv = 1; lv <= nLV; ++lv)
  for (int nv = 1; nv <= nNV; ++nv) for (int an = 1; an <= nAN; ++an)
  {
    // Poglejmo zadnje sporočilo, ki se nanaša to napravo ali na eno od njej nadrejenih.
    ZadnjeStanje Z = zadnje[zg][lv][nv][an];
    Z << zadnje[zg][lv][nv][0] << zadnje[zg][lv][0][0] << zadnje[zg][0][0][0];

    // Če je to sporočilo prestaro, mora biti njen zgoščevalnik nedosegljiv
    // in zato tudi ona sama.
    if (Z.cas < trenutniCas - 5) Z.status = Nedosegljiva;

    // Če je nedosegljiva ali v alarmu, to izpišimo.
    if (Z.status != OK) printf("Alarmna naprava (%d, %d, %d, %d) je %s.\n",
      zg, lv, nv, an, (Z.status == Alarm) ? "v alarmu" : "nedosegljiva");
  }
```

}

Podprogram Obdelaj predpostavlja, da se časi štejejo v sekundah od 0 naprej (zato tudi v tabeli zadnji inicializiramo vse čase na 0), da so sporočila že urejena po času in da trenutni čas dobi kot parameter (druga možnost bi bila, da bi čas najnovejšega sporočila šteli kot trenutni čas).

6. Avtomobili

Hitrost v posameznem koraku izračunamo kot razliko v koordinatah dveh zaporednih položajev. Za prvi korak potem ni treba drugega, kot da preverimo, če je bil avtomobil res v prvi prestavi, torej če je $\max\{|\triangle_x|, |\triangle_y|\} = 1$. Za vsak nadaljnji korak pa moramo preveriti, če sme hitrost v tistem koraku slediti hitrosti iz prejšnjega koraka (glede na omejitve, ki jih opisuje besedilo naloge).

Recimo, da je bila hitrost v prejšnjem koraku (u_x, u_y), v trenutnem pa (v_x, v_y). Da nam ne bo treba obravnavati preveč primerov posebej, lahko za začetek v mislih obrnemo koordinatni sistem tako, da bo veljalo $0 \le u_y \le u_x$ (kar ustreza primerom na sliki v besedilu naloge). To pomeni, da če je bil prej u_x negativen, pomnožimo zdaj tako u_x kot v_x z -1 (kar ustreza temu, da obrnemo x-os proti levi namesto proti desni); podobno tudi, če je bil u_y negativen, pomnožimo u_y in v_y z -1; in končno, če je $u_y > u_x$, zamenjamo koordinatni osi: $(u_x, u_y) \mapsto (u_y, u_x)$ in $(v_x, v_y) \mapsto (v_y, v_x)$. Slednje je koristno narediti tudi v primeru, če je $u_x = u_y$ in $v_y > v_x$: ta primer namreč pomeni zavijanje prek diagonale, kjer bi se potem v drugem koraku moralo prestavo meriti v navpični smeri namesto v vodoravni; če pa koordinatni osi zamenjamo, bo drugi korak ostal pod diagonalo in nam kasneje na ta posebni primer ne bo treba paziti.

Ko smo na ta način primerno postavili koordinatni sistem, je prestava v prvem koraku kar u_x, v drugem pa v_x. Preveriti moramo torej, ali je $|v_x - u_x| \le 1$ (ker se od enega koraka do naslednjega prestava ne sme povečati ali zmanjšati za več kot 1) in ali je $1 \le v_x \le 5$ (da prestava ostane od 1 do 5). Potem je tu še omejitev glede zavijanja: naloga pravi, da sme biti hitrost v drugem koraku največ za eno enoto oddaljena od točke, ki bi jo dosegla, če bi nadaljevala v isti smeri. Stara smer je bila (u_x, u_y) in ker je prestava v naslednjem koraku enaka v_x, bi isto smer dosegli s hitrostjo $(v_x, u_y \cdot v_x/u_x)$. Preveriti moramo torej, ali je $|v_y - u_y v_x/u_x| \le 1$; da se izognemo računanju z ne-celimi števili, lahko ta pogoj zapišemo tudi kot $|v_y u_x - u_y v_x| \le u_x$.

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

enum { MaxP = 5 };

// Preveri, ali korak s hitrostjo (vx, vy) lahko sledi koraku s hitrostjo (ux, uy).
bool JeVeljavno(int ux, int uy, int vx, int vy)
{
  // Obrnimo koordinatni sistem tako, da bo 0 <= uy <= ux.
  // Pri ux = uy poskrbimo še, da bo vy <= vx; tako ne bo treba posebej
  // obravnavati zavijanja čez diagonalo.
  if (ux < 0) ux = -ux, vx = -vx;
```

```
  if (uy < 0) uy = −uy, vy = −vy;
  if (ux < uy || ux == uy && vx < vy) { swap(ux, uy); swap(vx, vy); }

  // Prestava se ne sme spremeniti za več kot 1.
  if (vx < ux − 1 || vx > ux + 1) return false;

  // Prestava mora biti med 1 in 5.
  if (vx == 0 || abs(vx) > MaxP) return false;

  // Če bi nadaljevali v isti smeri kot doslej, bi bil drugi
  // korak (vx, vy′) za vy′ = uy vx/ux. Razlika med vy′ in pravim vy
  // sme biti največ 1, torej |vy − uy vx/ux| ≤ 1, torej |vy ux − uy vx| ≤ ux.
  return abs(vy * ux − uy * vx) <= ux;
}

// Preveri, ali je dano zaporedje položajev veljavno.
bool JeVeljavno(const vector<pair<int, int>> &polozaji)
{
  if (polozaji.size() < 2) return true;

  // V prvem koraku mora biti prestava 1.
  int ux = polozaji[1].first − polozaji[0].first, uy = polozaji[1].second − polozaji[0].second;
  if (max(abs(ux), abs(uy)) != 1) return false;

  // Preverimo ostale korake.
  for (int i = 2; i < polozaji.size(); ++i)
  {
    int vx = polozaji[i].first − polozaji[i − 1].first;
    int vy = polozaji[i].second − polozaji[i − 1].second;
    if (! JeVeljavno(ux, uy, vx, vy)) return false;
    ux = vx; uy = vy;
  }
  return true;
}
```

7. BinoXXO

Predpostavimo, da dobimo mrežo v dvodimenzionalni tabeli. Za začetek pojdimo v zanki po vseh poljih mreže; pri vsakem polju preverimo, ali je tam eden od znakov „`X`“ in „`O`“ in ali je znak na trenutnem polju enak kot na prejšnjih dveh levo od njega ali nad njim (če je, je to napaka).

Nekaj več dela je s preverjanjem, ali so vse vrstice različne. Lahko bi imeli tri gnezdene zanke: zunanji dve naj gresta po vseh možnih parih vrstic, notranja pa naj gre po stolpcih in preverja, ali se opazovani dve vrstici v trenutnem stolpcu kaj razlikujeta. Če pridemo do konca, ne da bi opazili kakšno neujemanje, lahko zaključimo, da sta vrstici enaki in mreža ni pravilno izpolnjena. Na enak način lahko preverimo tudi, ali so vsi stolpci različni, le vloga vrstic in stolpcev se zamenja.

```
bool JePravilno(char a[6][6])
{
  for (int y = 0; y < 6; ++y) for (int x = 0; x < 6; ++x)
  {
    // Preverimo, če so na vseh poljih znaki X in O.
    if (a[y][x] != 'X' && a[y][x] != 'O') return false;

    // Preverimo, če niso kje tri zaporedna enaka polja.
    if (x > 1 && a[y][x] == a[y][x − 1] && a[y][x] == a[y][x − 2]) return false;
    if (y > 1 && a[y][x] == a[y − 1][x] && a[y][x] == a[y − 2][x]) return false;
```

```
  }
  for (int y1 = 1; y1 < 6; ++y1) for (int y2 = 0; y2 < y1; ++y2)
  {
    // Preverimo, ali se vrstici y1 in y2 kaj razlikujeta.
    int x = 0; while (x < 6 && a[y1][x] == a[y2][x]) ++x;
    if (x >= 6) return false;
    // Preverimo, ali se stolpca y1 in y2 kaj razlikujeta.
    x = 0; while (x < 6 && a[x][y1] == a[x][y2]) ++x;
    if (x >= 6) return false;
  }
  return true; // Če pridemo do sem, je vse v redu.
}
```

Ker je mreža majhna, je ta rešitev čisto dobra; v splošnem pa, če bi imeli mrežo velikosti $w \times h$, bi opisani postopek poravil $O(w \cdot h^2)$ časa za preverjanje, ali so vse vrstice različne, in podobno $O(w^2 \cdot h)$ za stolpce. Boljša rešitev za večje mreže bi bila, da bi npr. vsako vrstico obravnavali kot niz w znakov (in podobno tudi za stolpce) in te nize shranjevali v razpršeno tabelo (npr. razred `set` iz C++ove standardne knjižnice); ob tem bi se dalo poceni opaziti, ali smo tako vrstico (oz. stolpec) kdaj prej že videli. Elegantna rešitev je tudi, da bi te nize zlagali v drevo (*trie*), v katerem je vsaka povezava označena z enim znakom in vsak niz je predstavljen z vejo od korena drevesa do enega izmed njegovih listov; ob dodajanju niza v tako drevo bi spet opazili, ali smo tak niz nekoč prej že dodali vanj (ker bi prišli do konca niza, ne da bi bilo treba dodati kakšno novo vozlišče). Tako bi bila časovna zahtevnost le $O(wh)$ namesto $O(wh(w+h))$.

8. Trgovski potnik

Naloga pravi, da mora potnik obiskati vse hiše na ulici med cestama, po katerih je prišel na ulico oz. odšel z nje. To pomeni, da če je med dvema zaporednima ulicama več hiš, bo moral vedno obiskati ali vse te hiše ali pa nobene; zato lahko tako skupino hiš v mislih zamenjamo z eno samo, pri kateri je čas obiska enak vsoti časov t_i po vseh hišah v skupini, podobno pa je tudi njen zaslužek enak vsoti zaslužkov p_i po vseh hišah v skupini. V nadaljevanju našega razmisleka bomo torej predpostavili, da je med vsakima dvema zaporednima ulicama natanko ena hiša: med ulicama $j-1$ in j je hiša, pri kateri čas obiska traja t_j, zaslužek pa je p_j. (Če v prvotnih vhodnih podatkih med dvema ulicama ni bilo nobene hiše, si lahko tam mislimo $t_j = p_j = 0$.)

Preprosta in malo manj učinkovita rešitev naloge je potem ta, da z dvema gnezdenima zankama pregledamo vse kombinacije dveh cest, ki ju potnik lahko uporabi. Recimo, da se njegova pot začne z ulico k, za vrnitev pa uporabi ulico ℓ; brez izgube za splošnost se lahko omejimo na primere, ko je $k \le \ell$ (drugače lahko potnik opravi isto pot v nasprotni smeri).

$r^\star := 0$; $k^\star := -1$; $\ell^\star := -1$; (* *najboljša znana rešitev* *)
for $k := 1$ **to** $m-1$:
  $P := 0$; $T := d_k$; (* *skupni zaslužek in trajanje poti* *)
  for $\ell := k+1$ **to** m:
    $P := P + p_\ell$; $T := T + t_\ell$;

$r := P/(T + d_\ell)$; (* *razmerje zaslužek/čas, če uporabimo cesti* k *in* ℓ ***)
if $r \geq r^\star$ **then** $r^\star := r$, $k^\star := k$, $\ell^\star := \ell$;
return $(k^\star, \ell^\star)$;

V zunanji zanki (po k) si torej izberemo cesto, po kateri se potnik odpravi, v notranji (po ℓ) pa tisto, po kateri se vrne. Ko pri danem k postopoma povečujemo ℓ, lahko tudi sproti seštevamo čase obiskov vseh hiš, mimo katerih gremo, ter skupni zaslužek po vseh teh obiskih. Iz tega ni težko izračunati razmerja med zaslužkom in časom (pri čemer ne pozabimo k času šteti tudi d_k in d_ℓ za obe uporabljeni cesti). Najboljše tako dobljeno razmerje si zapomnimo (v spremenljivki $r^\star$), skupaj z njim pa tudi cesti, pri katerih smo to razmerje dosegli ($k^\star$ in $\ell^\star$). Časovna zahtevnost te rešitve je $O(m^2)$, pred tem pa porabimo še $O(n)$ časa, da preberemo vhodni seznam n hiš in jih po potrebi seštejemo, tako da dobimo za vsaki dve zaporedni ulici le eno hišo.

Do boljše rešitve lahko pridemo z geometrijskim razmislekom. Naj bodo $P_j := p_1 + \ldots + p_j$ in $T_j := t_1 + \ldots + t_j$ kumulativne vsote zaslužkov in časov obiska za prvih j hiš. Če trgovski potnik odide po cesti k in se vrne po cesti ℓ (pri čemer je $k < \ell$), porabi za obiske hiš med njima skupno $T_\ell - T_k$ časa in pri tem zasluži $P_\ell - P_i$ denarja. Razmerje med zaslužkom in porabljenim časom je v tem primeru enako $(P_\ell - P_k)/(T_\ell - T_k + d_k + d_\ell)$ — v imenovalcu smo seveda morali prišteti tudi čas potovanja po obeh uporabljenih cestah.

Pišimo $U_i := T_i + d_i$; naše razmerje lahko potem zapišemo kot $(P_\ell - P_k)/(U_\ell - U_k + 2d_k)$. To pa je pravzaprav naklon poltraka iz točke $(U_k - 2d_k, P_k)$ skozi točko (U_ℓ, P_ℓ). Ker bi radi razmerje med zaslužkom in časom maksimizirali, nas torej zanima, pri katerem ℓ bo (za dani k) ta poltrak najstrmejši.

Ta problem je zelo podoben tistemu, s katerim smo se v tem biltenu že srečali pri nalogi „hipoteka“ na str. 161; razlika je le v tem, da smo tam iskali poltrak z najnižjim naklonom, tu pa z najvišjim. Podrobnosti si lahko zato bralec prebere tam, tu pa opišimo preostanek rešitve le na kratko. Če si izberemo neki konkreten k in se vprašamo, katera izmed točk iz neke množice M nam bo dala najstrmejši poltrak, se smemo omejiti na tiste točke, ki ležijo na zgornjem robu konveksne ovojnice množice M; in na tem robu lahko točko, ki nam dá najstrmejši poltrak, poiščemo z neke vrste bisekcijo v $O(\log m)$ časa. Vnaprej si bomo pripravili zgornji rob konveksne ovojnice za skupine po dveh, štirih, osmih itd. zaporednih točk (U_ℓ, P_ℓ). Pri izbranem k nas v resnici zanima najstrmejši poltrak po vseh točkah iz $M = \{(U_\ell, P_\ell) : k < \ell \leq m\}$, kar pa lahko razbijemo na $O(\log m)$ množic s po 2^t zaporednimi točkami (za različne eksponente t), poiščemo najstrmejši poltrak pri vsaki od njih posebej (s pomočjo vnaprej pripravljene konveksne ovojnice) in obdržimo najboljšo izmed tako dobljenih rešitev. Časovna zahtevnost tega postopka je $O(n)$ za začetno predelavo vhodnih podatkov o hišah, nato $O(m \log m)$ za pripravo vseh konveksnih ovojnic in nato $O((\log m)^2)$ pri vsakem k za izračun najstrmejšega poltraka; skupaj je to $O(n + m(\log m)^2)$.

9. Sprehod po mreži I

Nalogo lahko rešimo z iskanjem v širino, vendar ne po prvotni mreži, pač pa po prostoru stanj, kjer je stanje sestavljeno iz našega trenutnega položaja na mreži in iz smeri zadnjega premika. Oba tadva podatka namreč potrebujemo, da lahko določimo, kateri so možni premiki v naslednjem koraku. Smeri lahko predstavimo s

števili od 0 do 7, recimo nasproti smeri urinega kazalca: $0 = \rightarrow$, $1 = \nearrow$, $2 = \uparrow$, ..., $7 = \searrow$ (lahko si torej mislimo, da smeri merimo v enotah po 45°).

Recimo, da nas zanimajo poti od začetnega polja (z_x, z_y) do končnega (k_x, k_y). Potem začnemo iskanje v širino tako, dodamo v vrsto stanja (z_k, z_y, s) za vse možne smeri s, saj naloga pravi, da pri prvem koraku ni nobenih omejitev glede smeri premika. Ustavimo pa iskanje v širino takrat, ko se znajde v vrsti kakšno stanje (k_x, k_y, s) za poljuben s (saj naloga ne daje nobenih omejitev glede tega, iz katere smeri moramo priti v končno polje). Iz stanja (x, y, s) je mogoče v enem koraku narediti premik v vse smeri razen s, $(s + 1) \bmod 8$ in $(s + 7) \bmod 8$, kajti pri teh treh se smer spremeni za 45° ali manj.

Oglejmo si implementacijo takšne rešitve v C++:

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

// Podprogram poišče najkrajšo pot od (zx, zy) do (kx, ky) in jo shrani v „pot“.
// Če take poti sploh ni, bo vektor „pot“ prazen.
// Mrežo opisuje vektor M, v katerem naj M[y * w + x] pove, ali je polje (x, y) prehodno.
void NajkrajsaPot(int w, int h, const vector<bool> &M, int zx, int zy, int kx, int ky,
                  vector<pair<int, int>> &pot)
{
  struct Stanje { int prej = -1, d = -1; }; // prejšnje stanje in dolžina poti do trenutnega
  vector<Stanje> S(w*h*8); // S[(y*w+x)*8+smer] predstavlja stanje (x, y, smer)
  const int DX[8] = {1, 1, 0, -1, -1, -1, 0, 1}; // možne smeri premikanja
  const int DY[8] = {0, 1, 1, 1, 0, -1, -1, -1};
  queue<int> Q; // vrsta stanj, ki jih moramo še pregledati
  int konec = -1; // tu si bomo zapisali prvo doseženo končno stanje

  // Naslednji podprogram doda stanje (x, y, smer) v vrsto, če je novo.
  auto DodajVVrsto = [&, kx, ky, w] (int x, int y, int smer, int prej, int d) {
    int u = (y * w + x) * 8 + smer; auto &U = S[u]; if (U.d >= 0) return;
    Q.emplace(u); U.prej = prej; U.d = d; if (x == kx && y == ky) konec = u; };

  // Najprej dodajmo v vrsto vsa stanja, ki se nanašajo na začetni položaj.
  for (int smer = 0; smer < 8; ++smer) DodajVVrsto(zx, zy, smer, -1, 0);
  while (konec < 0 && ! Q.empty())
  {
    int u = Q.front(); Q.pop(); auto &U = S[u];
    int ux = (u / 8) % w, uy = (u / 8) / w, uSmer = u % 8;
    // Poglejmo, kam se lahko premaknemo iz stanja u = (ux, uy, uSmer).
    // Možne spremembe smeri so od 90 do 270 stopinj.
    for (int dSmeri = 2; dSmeri <= 6; ++dSmeri)
    {
      // Izračunajmo novo stanje (vx, vy, vSmer) pri trenutni spremembi smeri.
      int vSmer = (uSmer + dSmeri + 8) % 8;
      int vx = ux + DX[vSmer], vy = uy + DY[vSmer];
      // Če je polje (vx, vy) prehodno, dodajmo novo stanje v vrsto.
      if (vx >= 0 && vx < w && vy >= 0 && vy < h && M[vy * w + vx])
        DodajVVrsto(vx, vy, vSmer, u, U.d + 1);
    }
  }
  // Rekonstruirajmo pot do končnega položaja. Pri tem si pomagajmo z vektorjem S,
```

```
  // kjer za vsako doseženo stanje piše, iz katerega stanja smo prišli vanj.
  pot.resize(S[konec].d + 1);
  for (int u = konec; u >= 0; u = S[u].prej) pot[S[u].d] = {(u / 8) % w, (u / 8) / w};
}
```

10. Sprehod po mreži II

Našo karirasto mrežo lahko v mislih predelamo v usmerjen graf, pri čemer naj obstaja usmerjena povezava od (x, y) do (x', y') natanko tedaj, če sta točki sosednji (torej če se razlikujeta v eni koordinati za 1, v eni pa za 0) in če je druga višja od prve. V graf vključimo le tiste točke, ki so dosegljive iz (z_x, z_y).

Če v tako dobljenem grafu poiščemo najdaljšo naraščajočo pot (tako, pri kateri se višina na vsakem koraku poveča) z začetkom pri $z := (z_x, z_y)$, lahko potem to pot obrnemo in dobimo najdaljšo padajočo pot s koncem v (z_x, z_y), prav to pa naloga zahteva.

Predstavljajmo si najdaljšo pot od z do u; zadnji korak na njej je recimo povezava $v \to u$. Vse, kar je na tej poti pred tem zadnjim korakom, mora gotovo tvoriti neko najdaljšo pot od z do v, kajti če bi obstajala od z do v kakšna še daljša pot, bi jo lahko podaljšali s korakom $v \to u$ in novo pot od z do u, ki bi bila še daljša od dosedanje. Vidimo torej, da če hočemo najti najdaljšo pot od z do u, je koristno, če najprej poiščemo najkrajše poti od z do vseh tistih točk v, iz katerih obstajajo potem neposredne povezave $v \to u$. Vsako táko pot $z \rightsquigarrow v$ lahko poskusimo potem podaljšati v $z \rightsquigarrow v \to u$ in si med takó dobljenimi potmi od z do u zapomnimo najdaljšo. Za vsako točko bomo pri tem vzdrževali dolžino najdaljše doslej znane poti od z do nje, pa tudi smer zadnjega koraka na tej poti (oz. še bolje: nasprotno smer, kar nam bo na koncu olajšalo izpis poti v nasprotni smeri, da se bo spuščala in se končala pri z, kot zahteva besedilo naloge).

Točke grafa bi radi torej preiskali v takem vrstnem redu, da bo za vsako povezavo $v \to u$ veljalo, da obdelamo začetno krajišče v prej kot končno krajišče u. Tak vrstni red se v teoriji grafov imenuje *topološki vrstni red*; ima ga vsak acikličen usmerjen graf, torej tudi naš (če bi v našem grafu obstajal cikel, bi se dalo od neke točke priti po tem ciklu nazaj v to isto točko po taki poti, pri kateri se višina vsakem koraku poveča; torej bi bila ta točka višja od same sebe, to pa je protislovje).

Naš graf bomo torej pregledali dvakrat: (1) najprej izvedemo iskanje v širino z začetkom pri z; s tem obiščemo vse točke, ki so dosegljive iz z, za vsako od njih pa izračunamo tudi vhodno stopnjo (torej preštejemo povezave, ki kažejo vanjo). (2) Nato pa pregledamo graf v topološkem vrstnem redu. Pri tem vzdržujemo vrsto, v katero odlagamo točke z vhodno stopnjo 0. Na začetku dodamo v vrsto le točko z; nato pa na vsakem koraku vzamemo iz vrste neko točko v, pregledamo povezave $v \to u$ (pri čemer morda odkrijemo novo najdaljšo pot do kakšne od teh točk u) in jih v mislih pobrišemo iz grafa. To slednje pomeni, da se točki u vhodna stopnja zmanjša; in ko se zmanjša na 0, moramo takšno točko u dodati v vrsto.

Oglejmo si primer implementacije takšne rešitve v jeziku C++. Za vsako točko mreže bomo vzdrževali strukturo tipa `Tocka`, ki hrani višino točke (iz vhodnih podatkov), njeno vhodno stopnjo v našem grafu, njeno oddaljenost od z (torej dolžino najdaljše doslej znane poti od z do nje) in smer premika iz te točke do njene neposredne predhodnice na tej najdaljši poti od z.

```
#include <iostream>
#include <vector>
#include <queue>
#include <utility>
using namespace std;

int main()
{
  const int DX[] = {0, 0, 1, −1}, DY[] = {−1, 1, 0, 0}; const char *smeri = "SJVZ";
  // Preberimo vhodne podatke.
  int w, h, zx, zy; cin >> w >> h >> zx >> zy;
  struct Tocka { int visina = −1, stopnja = 0, odd = −1, smer = −1; };
  vector<Tocka> M(w * h); for (auto &tocka : M) cin >> tocka.visina;

  // Namesto poti, ki se spuščajo do (zx, zy), bomo razmišljali o poteh,
  // ki se vzpenjajo iz (zx, zy). Najprej označimo, katere točke so na ta
  // način dosegljive in kakšna je njihova vhodna stopnja.
  queue<pair<int, int>> vrsta; vrsta.emplace(zx, zy);
  while (! vrsta.empty()) {
    const auto [x, y] = vrsta.front(); vrsta.pop(); const auto &t = M[y * w + x];
    // Točka je (x, y) je dosegljiva; kam se lahko premaknemo iz nje?
    for (int smer = 0; smer < 4; ++smer) {
      // Ali je premik iz (x, y) v (xx, yy) veljaven korak navkreber?
      int xx = x + DX[smer], yy = y + DY[smer];
      if (xx < 0 || yy < 0 || xx >= w || yy >= h) continue;
      auto &tt = M[yy * w + xx]; if (tt.visina <= t.visina) continue;
      // Če točko (xx, yy) vidimo prvič, jo dodajmo v vrsto.
      if (++tt.stopnja == 1) vrsta.emplace(xx, yy); } }

  // Da najdemo najdaljšo pot, moramo ta graf preiskati v topološkem vrstnem redu.
  // Pri vsaki točki si bomo v „odd“ zapomnili oddaljenost (po najdaljši poti)
  // od (zx, zy), v „smer“ pa smer premika v prejšnjo točko na tej poti.
  vrsta.emplace(zx, zy); M[zy * w + zx].odd = 0;
  while (! vrsta.empty()) {
    const auto [x, y] = vrsta.front(); vrsta.pop(); const auto &t = M[y * w + x];
    for (int smer = 0; smer < 4; ++smer) {
      // Ali je premik iz (x, y) v (xx, yy) veljaven korak navkreber?
      int xx = x + DX[smer], yy = y + DY[smer];
      if (xx < 0 || yy < 0 || xx >= w || yy >= h) continue;
      auto &tt = M[yy * w + xx]; if (tt.visina <= t.visina) continue;
      // Če je to najdaljša znana pot do (xx, yy), si jo zapomnimo.
      if (t.odd + 1 > tt.odd) tt.odd = t.odd + 1, tt.smer = smer ^ 1;
      // Če smo obdelali že vse povezave, ki kažejo v (xx, yy), jo dodajmo v vrsto.
      if (−−tt.stopnja == 0) vrsta.emplace(xx, yy); } }

  // Poiščimo točko, ki je najdlje od (zx, zy).
  int x = zx, y = zy, odd = 0; for (int i = 0; i < w * h; ++i)
    if (M[i].odd > odd) odd = M[i].odd, x = i % w, y = i / w;
  // Izpišimo pot od te točke do (zx, zy).
  while (odd−− > 0) {
    auto &t = M[y * w + x]; cout << smeri[t.smer];
    x += DX[t.smer]; y += DY[t.smer]; }
  cout << endl; return 0;
}
```

11. Ocenjevanje nalog

Naj bo t_i težavnost, ki jo pripišemo nalogi i. Naloga potem zahteva naslednje: če je neki učenec uspešno rešil nalogo i in nekoč kasneje neuspešno poskusil rešiti nalogo j, potem mora biti $t_i < t_j$. Te omejitve si lahko predstavljamo kot graf, v katerem je po ena točka za vsako nalogo, povezava $i \to j$ pa obstaja natanko v tistih primerih, kjer imamo omejitev $t_i < t_j$.

Če v tem grafu obstaja cikel, mora imeti vsaka naloga na ciklu višjo težavnost od prejšnje naloge na ciklu, ta od predprejšnje in tako nazaj, kar nas sčasoma pripelje spet do naloge, s katero smo začeli; ta mora biti torej težavnejša od same sebe, kar je nemogoče, torej je problem v takem primeru nerešljiv.

Če pa v našem grafu ni cikla, ga lahko topološko uredimo, torej postavimo točke v tak vrstni red, pri katerem za vsako povezavo $i \to j$ velja, da nastopi naloga i v izbranem vrstnem redu prej kot naloga j. Točke lahko pregledamo v tem vrstnem redu in jim sproti pripisujemo težavnost. Če v točko j ne kaže nobena povezava, lahko postavimo težavnost t_j na 1 — manjša ne more biti, saj naloga zahteva, da so težavnosti naravna števila. Če pa v točko j kaže kakšna povezava, recimo $i \to j$, stoji začetno krajišče i te povezave v topološkem vrstnem redu pred točko j, torej bomo točki i že pripisali težavnost, še preden se bomo začeli ukvarjati s točko j. Najmanjša primerna težavnost točke j je potem $1 + \max_{i \in P(j)} t_i$, pri čemer pomeni $P(j)$ množico vseh tistih točk i, za katere obstaja povezava $i \to j$.

Naloga zahteva, da mora biti maksimalna ocena težavnosti (po vseh nalogah) čim manjša. Prepričajmo se, da ocene $t_1, \ldots, t_n$, dobljene po našem zgoraj opisanem postopku, ustrezajo tej zahtevi. Pa recimo, da bi obstajal neki drug nabor ocen $u_1, \ldots, u_n$, ki bi tudi ustrezal vsem omejitvam, obenem pa bi bila najvišja ocena pri njem manjša kot pri našem naboru: $\max_i u_i < \max_i t_i$. Naj bo $j^\star$ tista naloga, ki ima v našem naboru najvišjo oceno; torej je $t_{j^\star} = \max_i t_i > \max_i u_i \geq u_{j^\star}$. To, da je $t_j > u_j$, se torej gotovo zgodi vsaj pri $j = j^\star$, morda pa tudi še pri kakšnem drugem j. Če je takšnih j več, vzemimo med njimi tistega, ki je prvi v topološkem vrstnem redu (tistem, ki ga je uporabljal naš postopek). Ko pride naš postopek do tega j, ločimo dve možnosti. (1) Morda točka j v grafu nima nobene vhodne povezave; torej ji je naš postopek dodelil težavnost $t_j = 1$; toda potem je nemogoče, da bi bilo $t_j > u_j$ (ker tak u_j ne bi mogel biti ≥ 1 in torej ne bi bil naravno število); prišli smo v protislovje. (2) Torej mora točka j imeti kakšno vhodno povezavo. Toda za vsako tako povezavo $i \to j$ velja, da je i v topološkem vrstnem redu pred j, torej gotovo velja $t_i \leq u_i$ in ne $t_i > u_i$ (to slednje se namreč prvič zgodi šele pri točki j, saj smo tako j sploh definirali). Naš postopek je tedaj pripisal točki j težavnost $t_j = 1 + \max_{i \in P(j)} t_i$, kar je potemtakem $\leq 1 + \max_{i \in P(j)} u_i$. Po drugi strani je ocena u_j gotovo večja od vseh u_i za $i \in P(j)$, saj drugače nabor ocen $u_\bullet$ ne bi bil veljaven; torej je $u_j > \max_{i \in P(j)} u_i$; in ker so ocene celoštevilske, to pomeni $u_j \geq 1 + \max_{i \in P(j)} u_i \geq 1 + \max_{i \in P(j)} t_i = t_j$, torej $u_j \geq t_j$, kar je protislovje, saj smo j izbrali tako, da je $t_j > u_j$. — Vidimo torej, da nas predpostavka $\max_i u_i < \max_i t_i$ v vsakem primeru privede v protislovje, torej je maksimalna ocena pri našem naboru najmanjša možna. □

Zapišimo našo rešitev še s psevdokodo. Za začetek potrebujemo postopek, ki pripravi graf; natančneje povedano, za vsako nalogo j potrebujemo množico, ki smo jo zgoraj imenovali $P(j)$, torej tiste naloge i, za katere mora biti $t_i < t_j$.

for $j := 1$ **to** n **do** $P[j] :=$ prazna množica;
za vsakega učenca v vhodnih podatkih:
  $R :=$ prazna množica; (* *naloge, ki jih je že uspešno rešil* *)
  za vsako nalogo j, ki jo je poskušal rešiti (v takem vrstnem redu,
  v kakršnem jih je poskušal reševati):
    če je ta učenec to nalogo rešil, dodaj j v množico R;
    sicer $P[j] := P[j] \cup R$;

Pri topološkem urejanju bo prišlo prav, če bomo imeli poleg seznamov predhodnikov tudi sezname naslednikov: za nalogo i naj bo $N(i)$ množica tistih nalog j, za katere obstaja povezava $i \to j$.

for $i := 1$ **to** n **do** $N[i] :=$ prazna množica;
for $j := 1$ **to** n **do** za vsako $i \in P[j]$: dodaj j v $N[i]$;

Zdaj se lahko lotimo topološkega urejanja. Za vsako nalogo j bomo vzdrževali števec $d[j]$, ki pove, koliko nalogam iz $P(j)$ še nismo pripisali ocene težavnosti. Ko ta števec pade na 0, lahko pripišemo težavnost tudi nalogi j sami.

$Q :=$ prazna vrsta;
for $j := 1$ **to** n:
  $t[j] := 0$; $d[j] := |P[j]|$; **if** $d[j] = 0$ **then** dodaj j v Q;
dokler Q ni prazna:
  naj bo j poljubna naloga iz Q; pobriši j iz Q;
  za vsako $i \in P[j]$: $t[j] := \max\{t[j], t[i]\}$;
  $t[j] := t[j] + 1$; (†)
  za vsako $i \in N[j]$:
    $d[i] := d[i] - 1$; **if** $d[i] = 0$ **then** dodaj i v Q;

Na koncu imamo v tabeli $t[\cdot]$ primeren nabor ocen; če pa je kakšna od vrednosti $t[\cdot]$ še vedno 0, to pomeni, da nam grafa ni uspelo topološko urediti, torej je v njem cikel in problem ni rešljiv.

Razmislimo še o različicah naloge, ki ju omenja besedilo naloge na koncu. Spomnimo se, da če obstaja povezava $i \to j$, to pomeni, da je nekdo uspešno rešil nalogo i in kasneje neuspešno poskusil rešiti nalogo j; toda pri lažji različici naloge zdaj vemo tudi, da je nalogo i gotovo reševal na dan i, nalogo j pa na dan j; torej nastopi dan j kasneje kot dan i, torej je $i < j$. Z drugimi besedami, za vsako povezavo $i \to j$ velja $i < j$; torej se nam ni treba truditi z iskanjem topološkega vrstnega reda, saj že vnaprej vemo, da je primeren topološki vrstni red kar $1, 2, \ldots, n$.

Pri težji različici, kjer so ocene nekaterih nalog predpisane vnaprej, pa moramo v naši rešitvi spremeniti naslednje: ko naša dosedanja rešitev v vrstici (†) izračuna oceno $t[j]$, mora še preveriti, ali je za nalogo j morda v vhodnih podatkih podana že vnaprej predpisana ocena. Če ni, nadaljujemo kot običajno. Če pa je, recimo tej oceni τ_j; takrat razmišljajmo takole: če je $\tau_j \geq t[j]$, priredimo $t[j] := \tau_j$ in nadaljujmo s postopkom; če pa je $\tau_j < t[j]$, je problem nerešljiv, kajti $t[j]$ je najnižja ocena, ki ne prekrši nobene od omejitev na povezavah $i \to j$, torej bi dobili neveljaven nabor ocen, če bi namesto $t[j]$ uporabili predpisano oceno τ_j.

12. Knjižna kazala

Nalogo lahko rešujemo z dinamičnim programiranjem. Naj bo $f(m, P)$ cena najboljšega zaporedja premikov, če se omejimo na prvih m ukazov in bi radi na koncu imeli kazala na položajih $P = \{p_1, \ldots, p_k\}$. Pri tem si moramo P predstavljati ne kot množico, pač pa kot multimnožico (vrečo), ker je lahko več kazal na istem položaju in ima torej P več enakih elementov. Poleg tega mora biti vsaj en element P-ja enak x_m, ostali elementi pa morajo biti iz množice $\{1, x_1, \ldots, x_{m-1}\}$ (kajti na začetku so bila vsa kazala na položaju 1, nato smo eno od njih premaknili na x_1, nato eno na x_2 in tako naprej, v zadnjem premiku pa smo neko kazalo premaknili na x_m).

Če v takšni rešitvi potem premaknemo eno od kazal na x_{m+1}, dobimo veljavno zaporedje premikov za prvih $m + 1$ ukazov: za vsak $p \in P$ je torej $|p - x_{m+1}| + f(m, P)$ kandidat za vrednost $f(m+1, P-\{p\}\cup\{x_{m+1}\})$. Med več tako dobljenimi kandidati si moramo zapomniti najboljšega. Tako lahko sčasoma računamo rešitve vse večjih podproblemov, dokler ne pridemo do $m = u$. Zapišimo dobljeni postopek s psevdokodo; za vsak m bomo imeli slovar (razpršeno tabelo) f_m, v katerem bomo kot ključe hranili multimnožice P, pripadajoče vrednosti pa bodo cene najboljših rešitev $f(m, P)$.

```
f_0 := slovar, ki vsebuje le ključ {1, 1, ..., 1} z ceno 0;
for m := 1 to u:
  f_m := prazen slovar;
  za vsak ključ P (s pripadajočo ceno c) v slovarju f_{m-1}:
    za vsak različen element p ∈ P:
      P' := P − {p} ∪ {x_m};  c' := c + |x_m − p|;
      če ključa P' še ni v f_m, ga tja dodaj s pripadajočo ceno c';
      sicer, če je P' že v f_m, vendar s ceno, višjo od c',
        popravi v f_m ceno ključa P' na c';
  slovar f_{m-1} lahko tu pobrišemo, da porabimo manj pomnilnika;
vrni najmanjšo ceno iz slovarja f_u;
```

Pri posameznem m dosežemo take P-je, pri katerih je vsaj en element enak x_m, drugi pa so iz $\{1, x_1, \ldots, x_{m-1}\}$; torej je možnih kvečjemu $\min\{m, n\}^{k-1}$ različnih P-jev. (Pri tem smo upoštevali, da je za vsako kazalo možnih največ n različnih položajev; to je sicer pomembno le pri $m > n$.) Če to seštejemo po vseh m, vidimo, da ta rešitev pregleda približno $O(u \cdot s^{k-1})$ podproblemov za $s := \min\{n, u\}$. Pri vsakem podproblemu imamo potem notranjo zanko po p (to je $O(k)$ iteracij), v vsaki iteraciji te zanke pa nam bo izračun P' in poizvedba z njim v slovar f_m načeloma vzela $O(k)$ časa. Časovna zahtevnost je torej vsega skupaj $O(k^2 \cdot u \cdot s^{k-1})$. Da bo to obvladljivo, moramo imeti ali zelo majhno število kazal (npr. $k = 2$, da bo eksponent majhen) ali pa zmerno majhno število kazal in majhno osnovo s (torej mora biti knjiga tanka ali pa mora biti število ukazov majhno).

13. Drevesasti izrazi

Definicijo vsakega izraza bomo predstavili s strukturo, ki vsebuje ime izraza, ime funkcije in vektor z argumenti. Argumente lahko predstavimo s celimi števili, pri čemer vrednost $a > 0$ pomeni, da je argument naravno število a, vrednost $a \leq 0$ pa

nam bo pomenila, da je argument izraz, ki se v našem seznamu definicij pojavlja na indeksu $-a$.

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

struct Definicija { string imeIzraza, imeFunkcije; vector<int> argumenti; };
```

Za izpis izraza v razviti obliki lahko uporabimo rekurziven podprogram: izraz izpišemo tako, da izpišemo najprej ime funkcije, nato oklepaj, nato vse argumente (ločene z vejicami) in na koncu zaklepaj. Argumente, ki so naravna števila, izpišemo kar take, kot so; če pa argument predstavlja podizraz, moramo bodisi izpisati ta podizraz z rekurzivnim klicem (če ga doslej še nismo izpisali) bodisi izpisati tri pike (če smo ta podizraz kdaj prej že izpisali). Vzdrževati bomo morali torej tabelo s podatki o tem, katere podizraze smo že izpisali; v spodnji rešitvi je to zeIzpisan.

```
void IzpisiIzraz(const vector<Definicija> &definicije,
                 int stIzraza, vector<bool> &zeIzpisan)
{
  // V vektorju zeIzpisan si označujemo, katere izraze smo pri
  // izpisu že razvili, da jih ne bomo kasneje še enkrat.
  zeIzpisan[stIzraza] = true;

  // Izpišimo ime funkcije in oklepaj.
  auto &D = definicije[stIzraza]; cout << D.imeFunkcije << "(";

  // Izpišimo argumente.
  for (int i = 0; i < D.argumenti.size(); ++i)
  {
    if (i > 0) cout << ", "; // Med argumenti so vejice.

    // Če je argument > 0, ga izpišemo kot naravno število.
    if (int a = D.argumenti[i]; a > 0) cout << a;

    // Sicer je argument številka nekega izraza, pomnožena z -1.
    // Če smo ta izraz kdaj prej že izpisali, izpišemo le tri pike.
    else if (zeIzpisan[-a]) cout << "...";

    // Sicer pa izraz izpišemo z rekurzivnim klicem.
    else IzpisiIzraz(definicije, -a, zeIzpisan);
  }

  cout << ")"; // Izpišimo še zaklepaj na koncu.
}
```

Glavni podprogram IzpisiIzraze pa gre v zanki po vseh definicijah in za vsako izpiše ime izraza, enačaj in nato izraz v razviti obliki (za kar pokliče rekurzivni podprogram IzpisiIzraz).

```
void IzpisiIzraze(const vector<Definicija> &definicije)
{
  for (int stIzraza = 0; stIzraza < definicije.size(); ++stIzraza)
  {
    vector<bool> zeIzpisan(definicije.size(), false);
    cout << definicije[stIzraza].imeIzraza << " = ";
    IzpisiIzraz(definicije, stIzraza, zeIzpisan); cout << endl;
  }
```

```
}

int main()
{
  // Primer iz besedila naloge.
  IzpisiIzraze({
    {"foo", "b", {1, −1}},   // definicije[0]: foo = b(1, bar)
    {"bar", "c", {0, 6}}  }); // definicije[1]: bar = c(foo, 6)
  // Pri gornjem klicu se izpiše:
  //   foo = b(1, c(..., 6))
  //   bar = c(b(1, ...), 6)
  return 0;
}
```

14. Gospodarska rast

Ker je mreža neskončna in ker lahko vojak v okviru ene runde potuje poljubno daleč in pri tem prečka tudi zasedena polja, je pri tej igri popolnoma nepomembno, kje na mreži se kaj nahaja — katera polja so osvojena, kje stojijo vojaki in kje tovarne; pomembno je le, *koliko* je takih polj, koliko vojakov in koliko tovarn. (Zato tudi kot vhodni podatek dobimo le število osvojenih polj k, ne pa tudi položaja oz. zemljevida teh polj).

Opišimo torej stanje igre s trojico (p, v, t), ki pove, da imamo v vojakov, t tovarn in p praznih osvojenih polj. Skupno število osvojenih polj je potem $p+v+t$. Lahko se zgodi, da je isto stanje mogoče doseči na več načinov in imeti pri tem različno količino denarja; za nas so seveda zanimive rešitve s čim več denarja, ker nam te puščajo najbolj proste roke pri nadaljevanju igre. Naj bo torej $f_r(p, v, t)$ največja količina denarja, ki jo lahko imamo, če je igra ob koncu r-te runde v stanju (p, v, t). Funkcijo f_r si lahko predstavljamo kot slovar, v katerem so ključi tista stanja (p, v, t), ki so res dosegljiva po r rundah, pripadajoča vrednost pri takem ključu pa je največje število kovancev, s katerim lahko to stanje dosežemo.

Ta prostor stanj lahko sistematično preiskujemo po naraščajočem številu rund r; ustavili pa se bomo, čim dosežemo kakšno stanje s $p + v + t \geq n$. Pri osvajanju novih polj upoštevajmo še to, da ni nobene koristi od tega, da kakšen vojak v kakšni rundi ne bi osvojil novega polja; to lahko vedno stori, saj je mreža neskončna; tudi stane nas nič več, kot če bi vojak stal pri miru; število praznih osvojenih polj pa se pri tem poveča za 1, kar nam bo dalo v nadaljevanju igre več možnosti glede postavljanja novih vojakov in tovarn.

if $d \geq n$ **then return** 0;
$r := 0$; $f_0 :=$ slovar, v katerem je ključ le $(k, 0, 0)$ z vrednostjo d;

while true:
  $g :=$ prazen slovar;
  za vsako stanje (p, v, t) iz slovarja f_r:
    $z := f_r(p, v, t)$;
    $z := z - v$; **if** $z \leq 0$ **then continue**; (* *Plačajmo vojake.* *)
    $z := z + 5t$; (* *Poberimo denar od tovarn.* *)
    $p := p + v$; (* *Pošljimo vse vojake osvajat nova polja.* *)
    if $p + v + t \geq n$ **then return** $r + 1$; (* *Osvojili smo dovolj ozemlja.* *)

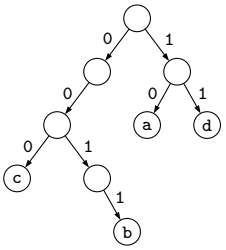
while true: (* *Postavimo nekaj tovarn* (*lahko tudi nobene*). *)
vpiši (p, v, t) z zneskom z v slovar g;
$p := p - 1$; $t := t + 1$; $z := z - 12$;
if $p < 0$ **or** $z < 0$ **then break**;

slovar f_r lahko zdaj pozabimo, da prihranimo nekaj pomnilnika;
$r := r + 1$; $f_r :=$ prazen slovar;
za vsako stanje (p, v, t) iz slovarja g:
$z := g(p, v, t)$;
while true: (* *Postavimo nekaj vojakov* (*lahko tudi nobenega*). *)
vpiši (p, v, t) z zneskom z v slovar f_r;
$p := p - 1$; $v := v + 1$; $z := z - 10$;
if $p < 0$ **or** $z < 0$ **then break**;

V vrsticah, ki vpisujeta v slovar g oz. f_r, je seveda mišljeno, da stanje vpišemo v slovar le, če ga še ni tam; če pa je že tam, moramo pogledati, ali je morda njemu pripadajoči znesek v slovarju manjši od z, in če je tako, ga povečamo na z.

15. Dekodiranje besedila

Pred dekodiranjem je koristno zložiti kode v *trie* — drevo, v katerem ima vsako vozlišče (največ) dva otroka, označena z bitoma 0 in 1. Vsaka pot od korena do kakšnega drugega vozlišča tako predstavlja neki niz bitov. Če je ta niz ravno koda kakšne črke, si to črko zapišimo v vozlišče na koncu te poti. Če se več kod ujema v prvih nekaj bitih, si tudi njim pripadajoče poti po drevesu delijo tistih prvih nekaj vozlišč, kasneje pa se razvejijo.



Primer drevesa, ki nastane, če imamo kode `a` → `10`, `b` → `0011`, `c` → `000` in `d` → `11`.

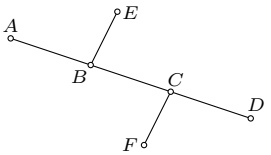
Pri dekodiranju začnemo v korenu drevesa, beremo vhodni niz bit po bit in se spuščamo po povezavah, ki ustrezajo prebranim bitom. Ko pridemo do vozlišča, ki predstavlja kodo kakšne črke, pa moramo to črko dodati v izhodni (dekodirani) niz in začeti pregledovati drevo spet od korena, da bomo razvozlali naslednjo kodo. Tako nadaljujemo vse do konca vhodnega niza.

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

struct Vozlisce {
  char crka = 0;
  Vozlisce *otroka[2] = { nullptr, nullptr }; };

// Doda v drevo dani niz bitov „koda“ in v vozlišče na koncu tega niza vpiše črko „crka“.
```

```
void DodajKodo(Vozlisce *koren, const char *koda, int crka)
{
  Vozlisce *v = koren; // Začnimo pri korenu drevesa.
  while (*koda)
  {
    // Poglejmo, v katerem v-jevem otroku moramo nadaljevati pot.
    auto &otrok = v->otroka[*koda++ - '0'];

    // Če tega otroka še ni, ga ustvarimo zdaj.
    if (! otrok) otrok = new Vozlisce();

    v = otrok; // Premaknimo se v tega otroka.
  }

  // Zapišimo si črko, ki jo predstavlja koda, s katero smo dosegli trenutno vozlišče.
  v->crka = crka;
}

// Dekodira dani niz (zaporedje znakov '0' in '1'). Kode v vektorju „kode"
// naj predstavljajo po vrsti črke 'a', 'b', 'c' in tako naprej.
string Dekodiraj(const char *niz, const vector<string> &kode)
{
  // Pripravimo si drevo in vanj vpišimo vse kode.
  Vozlisce *koren = new Vozlisce();
  for (int i = 0; i < kode.size(); ++i)
    DodajKodo(koren, kode[i].c_str(), 'a' + i);

  // Dekodirajmo niz.
  Vozlisce *v = koren; string dekodiranNiz;
  while (*niz) {

    // Preberimo naslednji bit niza in se spustimo po drevesu.
    v = v->otroka[*niz++ - '0'];

    // Če se v trenutnem vozlišču konča koda kakšne črke, jo dodajmo v izhodni
    // niz in začnimo pregledovati drevo spet od korena.
    if (v->crka) { dekodiranNiz.push_back(v->crka); v = koren; } }

  return dekodiranNiz;
}
```

16. Strešniki

Voda lahko zapusti daljico (v tem opisu rešitve bomo besedi „strešnik" in „daljica" uporabljali bolj ali manj kot sopomenki) le na njenih krajiščih ali pa tam, kjer se te daljice nekje med njenima krajiščema dotika neka druga daljica z enim od *svojih* krajišč. Z vidika naše naloge se torej zanimive stvari dogajajo le v krajiščih.



Recimo na primer, da imamo daljico AD, vendar v njeni notranjosti ležita tudi točki B in C, ki sta krajišči nekih drugih daljic (gl. sliko zgoraj). V mislih si lahko daljico AD predstavljamo kot razdeljeno na tri *člene:* AB, BC in CD. Če se znajde na daljici AD voda kjerkoli na intervalu $[A, B)$, bo od tam dosegla točko B; če se

znajde voda kjerkoli na intervalu $[B, C)$, bo od tam dosegla točko C; in če se znajde kjerkoli na intervalu $[C, D)$, bo od tam dosegla točko D. Iz točke D bo potem padla navpično navzdol in pri tem morda zadela kakšno drugo, niže ležečo daljico (oz. neki člen te daljice); tam bomo potem nadaljevali s podobnim razmislekom. Če pa voda, ki pada iz D, ne naleti na nobeno niže ležečo daljico, to pomeni, da bo tam nastal navpičen curek vode, ki bo v naših končnih rezultatih (po katerih sprašuje naloga) razmejeval dva suha intervala.

Nekaj več pazljivosti pa bo treba pri razmisleku, kako se voda razširi z ene daljice na drugo, kjer se tidve stikata (gl. primere na spodnji sliki na str. 62). (1) Recimo, da je točka T krajišče ene ali več daljic, poleg tega pa še leži v notranjosti daljice UV. Za začetek je koristno daljice s krajiščem T razdeliti glede na to, ali ležijo nad ali pod daljico UV — z drugimi besedami, ali drugo krajišče daljice (tisto, ki ni v točki T) leži nad ali pod premico skozi U in V; tako nastalima skupinama bomo rekli *zgornje* in *spodnje* daljice.

(1.1) Če zdaj voda priteče v točko T po eni od zgornjih daljic ali po daljici UV ali pa pade v točko T od zgoraj z navpičnim curkom, se lahko iz T razširi tudi na druge zgornje daljice (seveda le tiste, ki jim je T zgornje krajišče ali pa so vodoravne, ne pa tudi na tiste, ki jim je T spodnje krajišče), poleg tega pa tudi na daljico UV (po kateri teče iz točke T navzdol, če pa je UV vodoravna, teče seveda voda po njej v obe smeri); ne more pa se razširiti na spodnje daljice, kajti pri tem jo ovira daljica UV, ki — ne pozabimo — v točki T nima krajišča, zato voda tu ne more prodreti skoznjo. Iz enakega razloga voda v T tudi ne more pasti navpično navzdol.

(1.2) Podobno pa, če voda priteče v točko T po eni od spodnjih daljic, se lahko iz T razširi tudi na druge spodnje daljice (tiste, ki jim je T zgornje krajišče ali pa so vodoravne), poleg tega pa tudi pade iz T v curku navpično navzdol; ne more pa se razširiti na zgornje daljice, pa tudi ne na daljico UV, kajti voda v točki T ne more prodreti skozi daljico UV (kar bi morala storiti, da bi lahko potem tekla po zgornji strani te daljice).

Tidve možnosti, (1.1) in (1.2), sta čisto neodvisni; lahko se zgodi ena, druga, obe ali nobena. V mislih si lahko predstavljamo, kot da bi razdelili takšno točko T na dve, „zgornjo“ T_z in „spodnjo“ T_s; za zgornje daljice bi lahko potem rekli, da imajo krajišče T_z, ta točka pa je tudi tista, ki leži na UV (in razmejuje dva izmed njenih členov); spodnje daljice pa imajo krajišče T_s (in ta točka sploh ne leži na UV).

(2) Lažji primer nastopi, če je T krajišče ene ali več daljic, obenem pa ne leži v notranjosti nobene daljice. Tu torej ni nobene daljice, kot je bila prej UV, ki bi vodi ovirala prehajanje med daljicami s krajiščem T ali ji preprečevala, da bi iz T padala v navpičnem curku. Ta primer zato lahko obravnavamo enako kot (1.2); točko T, ki ne leži v notranjosti nobene daljice, si lahko predstavljamo kot spodnjo (brez pripadajoče zgornje točke).

Preden lahko opisani razmislek uporabimo za rešitev naloge (da sledimo gibanju vode in ugotovimo, s katerih krajišč katerih daljic se na koncu sceja v navpičnih curkih, ne da bi še kdaj zadela kak strešnik), moramo doreči še nekaj podrobnosti. Na začetku pada dež z neba (iz neskončne višine) pri vseh x-koordinatah; ugotoviti moramo, katere člene katerih daljic pri tem zadene, poleg tega pa še, na katerih intervalih x-koordinat ne zadene dež nobene daljice (in torej pade neovirano mimo vseh strešnikov). Poleg tega smo videli, da iz spodnjih točk voda pada tudi v curku

navpično navzdol; zanimalo nas bo seveda vedeti, kateri člen katere daljice pri tem zadene. In končno, znati moramo tudi dovolj učinkovito poiskati vse točke, kjer se dotikata dve ali več daljic (še posebej tiste, kjer leži krajišče ene daljice v notranjosti neke druge daljice).

Vse to lahko naredimo s preletom ravnine (*plane sweep*) z navpično premico po naraščajočih x-koordinatah. Med preletom ravnine z navpično premico bomo vzdrževali podatke o tem, katere daljice so prisotne na trenutni x-koordinati; to zaporedje daljic bomo vzdrževali urejeno po y-koordinati od spodaj navzgor (če je več daljic prisotnih na isti y-koordinati, pa morajo biti urejene naraščajoče po naklonu). Ker se daljice ne sekajo, se ta vrstni red ne spreminja drugače kot tako, da vanj občasno vstopi kakšna nova daljica (ko dosežemo x-koordinato njenega levega krajišča) in da občasno kakšna obstoječa daljica izpade iz njega (ko dosežemo x-koordinato njenega desnega krajišča). Da bomo lahko takšno urejeno zaporedje popravljali dovolj učinkovito, ga je koristno hraniti na primer v rdeče-črnem drevesu. Zapišimo psevdokodo našega preleta ravnine:

```
T := množica vseh točk, na katerih ležijo krajišča daljic;
za vsako daljico d naj bo L_d prazen seznam;
D := prazno drevo; x_p := −∞;
za vsak x_0, ki je x-koordinata kakšne točke iz T (v naraščajočem
vrstnem redu):
    če je D trenutno prazno, si zapomni, da voda, ki pada z neba na
      x-koordinatah (x_p, x_0), neovirano pada mimo vseh strešnikov;
    sicer naj bo d najvišja daljica, ki je trenutno v D; zapomni si, da voda,
      ki pada z neba na x-koordinatah (x_p, x_0), najprej pristane na
      |L_d|-tem členu daljice d;
    za vsako daljico d, ki ima desno krajišče na x = x_0:
        dodaj to desno krajišče na konec seznama L_d in pobriši d iz D;
    (* Trenutno so v D natanko tiste daljice, ki se začnejo levo od x_0
       in končajo desno od x_0; torej so prisotne na x = x_0, vendar tam
       nimajo krajišča. *)
    za vsak y_0, kjer je (x_0, y_0) ∈ T:
        če je v D prisotna daljica d, ki gre skozi (x_0, y_0),
          to pomeni, da ta točka leži na daljici d, ni pa krajišče te daljice;
          dodaj (x_0, y_0) na konec seznama L_d;
        med daljicami iz D, ki imajo pri x = x_0 višino y < y_0, naj bo
          y_1 najvišja vrednost y in naj bo d_1 daljica, ki gre skozi (x_0, y_1);
          (če ni nobene take daljice, si mislimo y_1 = −∞);
        naj bo y_2 := max{y : (x_0, y) ∈ T, y < y_0};
        za točko T si zapomni, da voda, ki curlja iz nje navpično navzdol:
            (če je y_1 = y_2 = −∞) pada v nedogled, ne zadene nobenega strešnika;
            (sicer, če je y_2 ≥ y_1) pristane na točki (x_0, y_2);
            (sicer) pristane na |L_{d_1}|-tem členu daljice d_1;
    (* konec zanke po y_0 *)
    naj bo d_1 tista daljica iz D, ki ima pri x = x_0 najvišjo y-koordinato,
      in naj bo y_1 ta y-koordinata (če je D prazen, naj bo y_1 = −∞);
    naj bo y_2 := max{y : (x_0, y) ∈ T};
```

19 če je $y_1 > y_2$, si zapomni, da voda, ki pada z neba pri $x = x_0$,
najprej pristane $|L_{d_1}|$-tem členu daljice d_1;
20 sicer si zapomni, da voda, ki pada z neba pri $x = x_0$, najprej
pristane na točki (x_0, y_2);
21 za vsako daljico d, ki ima levo krajišče na $x = x_0$:
22 dodaj to levo krajišče na konec seznama L_d in dodaj d v D;
23 $x_p := x_0$;
(* *konec zanke po* x_0 *)
24 za vodo, ki pada z neba na x-koordinatah (x_p, ∞), si zapomni,
da neovirano pada mimo vseh strešnikov;

Rezultat tega preleta ravnine je, da za nekatere odprte intervale x-koordinat že vemo, da voda, ki pada z neba pri teh intervalih, neovirano pride mimo vseh strešnikov (vrstici 5 in 24); poleg tega za vsako točko vemo, ali leži na kakšni daljici, ne da bi bila njeno krajišče (vrstica 10); za vsako točko vemo tudi, kje (če sploh kje) bi pristala voda, ki bi curljala iz nje navpično navzdol (vrstice 13–16); poleg tega pa imamo za vsako daljico d seznam L_d vseh točk na njej, od leve proti desni. Poleg tega smo za nekatere člene nekaterih daljic, pa tudi za nekatere točke, izvedeli, da voda nanje pade neposredno z neba (ker nad njimi ni nobenega drugega strešnika; vrstici 19–20).

Zdaj moramo gibanju te vode slediti od zgoraj navzdol po strešnikih, da bomo ugotovili, pri kateri x-koordinati še zadnjič pade s konca kakega strešnika. Da bomo to laže naredili, predstavimo strešnike z usmerjenim grafom (V, E), ki ima po dve točki („spodnjo“ in „zgornjo“) za vsako krajišče $t \in T$, poleg tega pa še po eno točko za vsak člen vsake daljice. Povezave $u \to v$ bodo v tem grafu prisotne natanko tam, kjer lahko voda iz člena ali krajišča u teče neposredno v člen ali krajišče v. Te povezave pokrivajo tako primere, kjer se dve daljici dotikata in lahko voda teče z ene na drugo, kot tudi primere, kjer voda curlja navpično navzdol z ene daljice na drugo (povezave za takšne navpične curke se vedno začnejo v eni od spodnjih točk, končajo pa bodisi v enem od členov ali pa v eni od zgornjih točk). Če kakšna $t \in T$ ne leži v notranjosti nobene daljice (pač pa je le krajišče ene ali več daljic) in torej „zgornje“ točke zanjo ne potrebujemo, jo bomo v spodnjem postopku vendarle ustvarili, nato pa zanjo tudi dodali povezavo v pripadajočo „spodnjo“ točko (tako lahko voda, ki ob navpičnem padanju morda doseže zgornjo točko, pride od tam v spodnjo in je učinek enak, kot če bi imeli le spodnjo točko). Oglejmo si zdaj psevdokodo za pripravo grafa:

25 $V := \{(t, \mathtt{S}), (t, \mathtt{Z}) : t \in T\} \cup \{(d, i) : d \in D, 1 \le i < |L_d|\}$; $E := \{\}$;
26 za vsako $v \in V$ naj bo $N[v]$ prazna množica;
27 za vsako daljico $d \in D$:
28 $k := |L_d|$; $t := L_d[1]$; $u := L_d[k]$;
29 dodaj $(d, 1)$ v $N[t, \mathtt{S}]$ in $(d, k-1)$ v $N[u, \mathtt{S}]$;
30 za vsako daljico $d \in D$, za vsak $i = 2, 3, \ldots, |L_d| - 1$:
31 $t := L_d[i]$; (* *točka* t *leži na* d*, vendar ni njeno krajišče* *)
32 $N[t, \mathtt{Z}] := \{(d, i-1), (d, i)\}$; (* *člena daljice* d*, med katerima je točka* t *)
33 za vsak $(d', i) \in N[t, \mathtt{S}]$:
34 naj bo u tisto od krajišč daljice d', ki ni enako t;
35 če leži u nad nosilko daljice d, premakni (d', i) iz $N[t, \mathtt{S}]$ v $N[t, \mathtt{Z}]$;

36 za vsako $(t, c) \in V$, za vsako $(d, i) \in N[t, c]$:
37 člen (d, i) povezuje točki $L_d[i]$ in $L_d[i + 1]$; naj bo u tista od teh dveh točk, ki ni enaka t;
38 če je $u.y \geq t.y$, dodaj v E povezavo $(d, i) \to (t, c)$;
39 če je $t.y \geq u.y$, dodaj v E povezavo $(t, c) \to (d, i)$;
40 za vsako točko $t \in T$:
41 če je $N[u, \mathtt{Z}]$ prazna, dodaj v E povezavo $(t, \mathtt{Z}) \to (t, \mathtt{S})$;
42 če voda, ki curlja navpično dol iz t, pristane v točki u (vrstica 15):
43 dodaj v E povezavo $(t, \mathtt{S}) \to (u, \mathtt{Z})$;
44 sicer, če ta voda pristane na i-tem členu daljice d (vrstica 16):
45 dodaj v E povezavo $(t, \mathtt{S}) \to (d, i)$;

Množica $N[v]$ predstavlja člene, ki so sosednji točki v (torej imajo točko v za eno od svojih krajišč). Najprej dodamo člene, ki vsebujejo kakšno krajišče kakšne daljice (torej prvi in zadnji člen vsake daljice), v množico $N[v]$ za spodnjo točko tistega krajišča (vrstice 27–29). Nato za vsako točko t v notranjosti kakšne daljice d pogledamo, kateri od členov iz $N[t, \mathtt{S}]$ v resnici ležijo nad daljico d in jih bo treba zato povezati z zgornjo točko, ne s spodnjo; tiste potem premaknemo v $N[t, \mathtt{Z}]$ (vrstice 30–35). Nato lahko dodamo povezave med vsako točko v in njej sosednjimi členi $N[v]$, pri čemer morajo biti te povezave seveda usmerjene tako, da voda ne teče navkreber (vrstice 36–39). Na koncu pa dodamo še povezave, ki predstavljajo navpične curke vode iz vsake spodnje točke (vrstice 42–45) in ki v primerih, ko zgornje točke v resnici sploh ne bi smelo biti, omogočijo vodi, da pride iz nje v pripadajočo spodnjo točko (vrstica 41).

Graf je zdaj pripravljen in v njem lahko z iskanjem v širino ugotavljamo, kam vse se razlije voda; pri tem v množici R hranimo vse točke grafa, za katere že vemo, da jih voda doseže, v vrsti Q pa tiste, za katere še nismo pogledali, kam se potem voda razlije naprej iz njih. Na začetku dodamo v Q in R tiste točke grafa (krajišča in člene), za katere smo že ob preletu ravnine ugotovili, da pada voda z neba neposredno nanje.

46 $Q :=$ prazna vrsta; $R :=$ prazna množica;
47 za vsako točko $t \in T$, v katero pada voda neposredno z neba (vrstica 20):
48 dodaj $(t, \mathtt{Z})$ v Q in v R;
49 kjerkoli pada voda neposredno z neba na i-ti člen daljice d (vrstica 19):
50 dodaj (d, i) v Q in v R;
51 dokler Q ni prazna:
52 naj bo u poljuben element Q; pobriši u iz Q;
53 za vsako povezavo $(u \to v) \in E$, če $v \notin R$:
54 dodaj v v Q in v R;
55 $P :=$ prazen seznam;
56 za vsako $t \in T$, če voda, ki pada navpično dol iz t, ne zadene nobenega strešnika več (vrstica 14):
57 če je $(t, \mathtt{S}) \in R$, dodaj $t.x$ v P;

Zdaj imamo v P zbrane vse tiste x-koordinate, pri katerih se voda sceja s krajišča kakega strešnika in od tam pada navpično navzdol, ne da bi še kdaj zadela kak strešnik. Skupaj z odprtimi intervali, kjer voda pada neovirano mimo vseh strešnikov (vrstici 5 in 24), so to potem vse x-koordinate, kjer voda doseže tla.

58 $O :=$ množica odprtih intervalov, kjer pada voda neovirano z neba,
ne da bi zadela kak strešnik (vrstici 5 in 24);
59 uredi P naraščajoče;
60 za vsaki dve zaporedni koordinati x_i in x_{i+1} iz P:
61 če (x_i, x_{i+1}) ni v O, potem je to suh interval;

Razmislimo še o časovni zahtevnosti naše rešitve. Imamo n strešnikov, zato tudi $O(n)$ krajišč; poleg tega vsako krajišče lahko leži še na največ eni drugi daljici, ki tam nima krajišča (kajti če bi bili taki daljici dve, bi se sekali, naloga pa zagotavlja, da se strešniki ne sekajo). To pomeni, da je skupno število členov (po vseh daljicah skupaj) tudi $O(n)$. Za urejanje točk po x-koordinatah (za vrstico 4) porabimo $O(n \log n)$ časa. Vsako daljico moramo enkrat dodati v drevo (vrstica 22) in jo enkrat pobrisati iz njega (vrstica 7); za vsako točko imamo še $O(1)$ operacij v drevesu v vrsticah 6, 10, 11; to je skupaj $O(n)$ operacij v drevesu, vsaka od teh pa vzame $O(\log n)$ časa. Vrstica 12 nas stane le $O(1)$, če pregledujemo točke (pri vsakem x_0) po naraščajočih y_0, za kar lahko poskrbimo, ko jih na začetku urejamo. Tako nam torej prelet ravnine vzame $O(n \log n)$ časa. Priprava grafa nam vzame le $O(n)$ časa: točk grafa je $O(n)$ (po ena za vsak člen in po dve za vsako krajišče), povezav pa tudi. Vsak člen namreč leži med dvema krajiščema in povezave gredo lahko le iz točk, ki predstavljajo tidve krajišči, v točko, ki predstavlja člen, ali obratno; poleg tega pa gre lahko iz vsakega krajišča še ena povezava, ki predstavlja padanje navpično navzdol. Ker ima graf $O(n)$ točk in povezav, nam tudi iskanje v širino vzame $O(n)$ časa. Na koncu imamo še $O(n \log n)$ časa za urejanje množice P (vrstica 59). Vsega skupaj je torej časovna zahtevnost te rešitve $O(n \log n)$, prostorska pa $O(n)$.

17. Predlaganje naslednje besede

Za začetek razmislimo o lažji različici naloge, pri kateri predpostavimo, da se uporabnik pri prejšnji besedi ni zatipkal. Pri poizvedbi (p_i, c_i) nas potem zanima, katera med besedami, ki se začnejo na c_i in so v preteklosti sledile besedi p_i, je najpogostejša. Koristno bi bilo torej za besedo p_i imeti drevo po črkah (*trie* — primer smo letos že videli pri nalogi „dekodiranje besedila“ na str. 200, le da smo imeli tam v drevesu nize ničel in enic, tukaj pa bomo imeli nize malih črk abecede), v katero bi zložili besede, ki so v preteklosti sledile besedi p_i. Pri vsakem takem vozlišču drevesa, v katerem se konča kakšna od teh besed, pa bi zapisali tudi število njenih pojavitev (torej kolikokrat je v preteklosti ta beseda sledila besedi p_i); temu številu za vozlišče u recimo $N[u]$.

Če imamo takšno drevo — recimo mu $T(p_i)$ — se lahko potem po njem spustimo od korena navzdol po povezavah, ki po vrsti ustrezajo črkam niza c_i. Za vozlišče (recimo mu x), v katerem se ustavimo (ko pridemo do konca c_i), nas potem načeloma zanima, katero vozlišče v med x-ovimi potomci (lahko je tudi $v = x$) ima največjo vrednost $N[v]$; tisto potem ustreza najpogostejši besedi na c_i, ki je doslej sledila besedi p_i. Vidimo torej, da je koristno, če v vsakem vozlišču u našega drevesa poleg vrednosti $N[u]$ hranimo tudi podatek o tem, pri katerem u-jevem potomcu v je dosežena največja vrednost $N[v]$; recimo temu potomcu $M[u]$. V vsakem vozlišču u bomo hranili tudi kazalec na njegovega starša $P[u]$ ter črko $C[u]$ na povezavi od $P[u]$ do u.

Zapišimo s psevdokodo postopek za dodajanje niza $s = s_1 s_2 \ldots s_{|s|}$ v drevo T:

podprogram DODAJ(drevo T, niz s):
  $u :=$ koren drevesa T; DODAJR(u, s, 1);

podprogram DODAJR(vozlišče u, niz s, indeks i):
  if $i > |s|$:
    $N[u] := N[u] + 1$;
    if $N[u] > N[M[u]]$ **then** $M[u] := u$;
    return;
  $v :=$ tisti u-jev otrok, do katerega vodi povezava s črko s_i;
  če takega otroka v še ni, ga ustvari zdaj, ga dodaj pod u in vanj
    vpiši $P[v] := u$, $C[u] := s_i$, $N[v] := 0$ in $M[v] := v$;
  DODAJR(v, s, $i + 1$);
  if $N[M[v]] > N[M[u]]$ **then** $M[u] := M[v]$;

Imamo torej rekurzivni podprogram DODAJR, ki se spušča po drevesu po črkah s-ja in pri tem dodaja morebitna manjkajoča vozlišča; ko pride do konca s-ja, poveča v ustreznem vozlišču $N[u]$ za 1, nato pa ob vračanju gor po drevesu v vsakem vozlišču, če je treba, popravi vrednost $M[u]$.

Še preprostejša je poizvedba v drevesu: spuščamo se od korena navzdol po povezavah, ki ustrezajo črkam poizvedovalnega niza, in ko pridemo do konca le-tega, nam vrednost $M[u]$ v trenutnem vozlišču pokaže na vozlišče, v katerem se konča najpogostejša beseda, ki se začne na poizvedovalni niz. Če hočemo to besedo tudi rekonstruirati, se lahko sprehodimo od $M[u]$ gor do korena (pri tem uporabljamo vrednosti $P[\cdot]$, da se iz trenutnega vozlišča premaknemo v njegovega starša) in stikamo črke na povezavah (vrednosti $C[\cdot]$), na koncu pa seveda tako dobljeni niz obrnemo.

funkcija SPUSTISEPODREVESU(drevo T, niz s):
  $u :=$ koren drevesa T;
  for $i := 1$ **to** $|s|$:
    $u :=$ tisti otrok u-ja, do katerega vodi povezava s črko s_i;
    if takega otroka ni **then return** NIL;
  return u;

funkcija NIZDOVOZLIŠČA(vozlišče u):
  $t :=$ prazen niz;
  while $u \neq$ NIL:
    dodaj $C[u]$ na konec t-ja; $u := P[u]$;
  obrni niz t (z desne na levo) in ga vrni;

funkcija POIZVEDBA(drevo T, niz s):
  $u :=$ SPUSTISEPODREVESU(koren T-ja, s);
  if $u =$ NIL **then return** s
  else return NIZDOVOZLIŠČA($M[u]$);

Pri spuščanju po drevesu se lahko izkaže, da otroka, v katerega bi se morali spustiti po naslednji črki s-ja, sploh ni; to pomeni, da se noben niz v drevesu ne začne na s. Naloga ne določa zares, kaj naj v takem primeru naredimo, zato zgoraj vrnemo kar s (uporabniku torej ne bomo predlagali ničesar razen tega, kar je tako ali tako že sam natipkal).

Doslej smo razmišljali o poenostavljeni nalogi, pri kateri se uporabnik pri prejšnji besedi ne more zatipkati. Razmislimo zdaj, kaj moramo spremeniti, da bomo podprli tudi možnost zatipkanja. Zdaj se torej lahko zgodi, da p_i ni „prava“ prejšnja beseda (tista, ki jo je imel uporabnik v resnici v mislih), pač pa je možna katerakoli taka beseda $\hat{p}_i$, ki se od p_i razlikuje na enem mestu. Če bi za vsako od teh $\hat{p}_i$ izvedli poizvedbo v $T(\hat{p}_i)$ in dobili najverjetnejšo naslednjo besedo, bi morali potem na koncu od teh besed uporabiti tisto z največ pojavitvami. Težava je, da je možnih $\hat{p}_i$ precej (velikost abecede, pomnožena z dolžino niza p_i) in je ta rešitev počasna.

Bolje je, če si podatke vnaprej malo tudi poagregiramo. Nizu, ki ima na enem mestu vprašaj namesto črke, bomo rekli *vzorec*. Vsak vzorec tako predstavlja več nizov: `tr?a` na primer predstavlja nize `trta`, `trma`, `trda` in tako naprej. Če je $\tilde{p}$ neki vzorec, označimo s $P(\tilde{p})$ množico vseh nizov, ki jih dobimo, če vprašaj v $\tilde{p}$ zamenjamo z neko črko. Označimo s $T(p, s)$ tisto vozlišče v $T(p)$, za katero črke na poti od korena do tega vozlišča tvorijo ravno niz s.

Za vsak vzorec $\tilde{p}$ bomo vzdrževali drevo $T(\tilde{p})$, v katerem bodo obstajala vozlišča $T(\tilde{p}, s)$ za vse take nize s, za katere obstaja vozlišče $T(p, s)$ pri vsaj enem $p \in P(\tilde{p})$. Če je zdaj $u = T(\tilde{p}, s)$ neko tako vozlišče v $T(\tilde{p})$, bomo v njem hranili kot $M[u]$ kazalec na tisto vozlišče $v = M[T(p, s)]$, ki ima (po vseh $p \in P(\tilde{p})$) največjo vrednost $N[v]$. Vrednost $M[u]$ nam torej pove, katera je najpogostejša naslednja beseda na s, ki jo lahko dobimo, če vprašaj v vzorcu $\tilde{p}$ zamenjamo z neko konkretno črko po svoji izbiri.

Pri poizvedbi p_i bomo morali pregledati drevesa $T(\tilde{p})$ za vse vzorce $\tilde{p}$, ki jih lahko dobimo, če v p_i en znak spremenimo v vprašaj, in od tako dobljenih kandidatov za naslednjo besedo vrniti najpogostejšo.

Oglejmo si zdaj psevdokodo naše rešitve. Za začetek je tu podprogram, s katerim popravimo drevo $T(\tilde{p})$ po tistem, ko smo v $T(p)$ dodali niz s. Za vzorec $\tilde{p}$ predpostavimo, da ga je mogoče dobiti iz p s tem, da spremenimo en znak v vprašaj.

podprogram DodajV(niz p, vzorec $\tilde{p}$, niz s):
  $u :=$ koren drevesa $T(p)$; $\tilde{u} :=$ koren drevesa $T(\tilde{p})$;
  for $i := 1$ **to** $|s| + 1$:
    if $N[M[u]] > N[M[\tilde{u}]]$ **then** $M[\tilde{u}] := M[u]$;
    if $i > |s|$ **then break**;
    $u :=$ tisti otrok u-ja, do katerega kaže povezava s črko s_i;
    $v :=$ tisti otrok $\tilde{u}$-ja, do katerega kaže povezava s črko s_i;
    če takega otroka v še ni, ga ustvari zdaj, ga dodaj pod $\tilde{u}$ in vanj
      vpiši $P[v] := \tilde{u}$, $C[u] := s_i$ in $M[v] := M[u]$;
    $\tilde{u} := v$;

Naslednji podprogram doda niz s v drevo $T(p)$ in popravi drevesa $T(\tilde{p})$ za vse vzorce, ki jih je mogoče dobiti iz p:

podprogram DodajVVsa(niz p, niz s):
  Dodaj($T(p)$, s);
  for $j := 1$ **to** $|p|$:
    $\tilde{p} :=$ vzorec, ki ga dobimo, če v p spremenimo j-ti znak v vprašaj;
    DodajV(p, $\tilde{p}$, s);

Nekaj, česar zgoraj nismo eksplicitno pisali, na kar pa je vendarle treba paziti, je, da če drevesa $T(p)$ ali $T(\tilde{p})$ sploh še nimamo, ga je treba pred dodajanjem seveda še ustvariti. Prazno drevo vsebuje le koren u, v katerem je $N[u] = 0$ in $M[u] = C[u] = P[u] = \text{NIL}$. Vsa drevesa lahko hranimo v slovarju oz. razpršeni tabeli, kjer so ključi nizi (ali vzorci) p, pripadajoče vrednosti pa drevesa $T(p)$; lahko pa bi nize p zložili celo v še en *trie*, kjer bi potem vozlišče, do katerega pridemo, če od korena navzdol sledimo povezavam, označenim s črkami p-ja, vsebovalo kazalec na koren drevesa $T(p)$.

Pri poizvedbi (p_i, s_i) moramo pogledati v drevesa $T(\tilde{p})$ za vse take vzorce $\tilde{p}$, ki jih je mogoče dobiti iz p_i s spremembo enega znaka v vprašaj. Med rezultati, ki jih tam dobimo, obdržimo tistega z največ pojavitvami:

funkcija PoizvedbaPoVseh(niz p, niz s):
  $u^\star := \text{NIL}$; $n^\star := 0$;
  for $j := 1$ **to** $|p|$:
    $\tilde{p} :=$ vzorec, ki ga dobimo, če v p spremenimo j-ti znak v vprašaj;
    if drevesa $T(\tilde{p})$ nimamo **then continue**;
    $u :=$ SpustiSePoDrevesu($T(\tilde{p})$, s);
    if $u = \text{NIL}$ **or** $N[M[u]] \leq n^\star$ **then continue**;
    $u^\star := M[u]$; $n^\star := N[u^\star]$;
  if $u^\star = \text{NIL}$ **then return** s
  else return NizDoVozlišča(u);

Besedilo naloge pravi, da na začetku dobimo zaporedje z besed, ki jih je uporabnik doslej že natipkal; recimo jim $w_1, w_2, \ldots, w_z$. Iz njih lahko zdaj zelo preprosto inicializiramo naša drevesa, nato pa začnemo odgovarjati na poizvedbe in po vsaki dopolnimo drevesa s pravkar natipkano besedo. Glavni blok našega postopka bo torej takšen:

  for $i := 2$ **to** z **do** DodajVVsa(w_{i-1}, w_i);
  for $i := 1$ **to** q:
    izpiši PoizvedbaPoVseh(p_i, c_i);
    DodajVVsa(p_i, a_i);

Razmislimo še o časovni zahtevnosti te rešitve. Besede so dolge po največ m znakov, zato traja dodajanje v eno drevo $O(m)$ časa; iz besede dolžine m lahko dobimo m vzorcev, torej mora DodajVVsa dodati niz v $O(m)$ dreves, za kar porabi $O(m^2)$ časa. Podobno tudi spuščanje po drevesu vzame $O(m)$ časa in ker mora podprogram PoizvedbaPoVseh narediti to v $O(m)$ drevesih, porabi tudi on skupno $O(m^2)$ časa. Vsega skupaj je torej časovna zahtevnost naše rešitve $O((z+q)m^2)$.

Naloge so sestavili: predlaganje naslednje besede — Urban Duh; sestankovalna sova — Matija Grabnar; funkcije, trgovski potnik, sprehod po mreži I — Gregor Kikelj; ocenjevanje nalog, dekodiranje besedila — Vid Kocijan; varnostnik, sprehod po mreži II — Filip Koprivec; futoši — Samo Kralj; ultranet, avtomobili, binoXXO — Mitja Lasič; gospodarska rast, strešniki — Jure Slak; knjižna kazala, drevesasti izrazi — Janez Brank.

NASVETI ZA MENTORJE O IZVEDBI ŠOLSKEGA TEKMOVANJA IN OCENJEVANJU NA NJEM

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

Tekmovalci naj pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom; ocenjevanje teh odgovorov poteka v vsakem primeru tako, da jih pregleda in oceni mentor (in ne npr. tako, da bi se poskušalo izvorno kodo, ki so jo tekmovalci napisali v svojih odgovorih, prevesti na računalniku in pognati na kakšnih testnih podatkih). 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.

Če oblika vhodnih podatkov ni natančno določena, si lahko podrobnosti tekmovalec izbere sam. Na primer, če naloga pravi, da dobimo seznam parov, je to lahko v praksi tabela (*array*), vektor, *linked list* ali še kaj drugega, pari pa so lahko

bodisi strukture, ki jih je deklarirala tekmovalčeva rešitev, ali pa kaj iz standardne knjižnice (kot je pair v C++ ali tuple v pythonu).

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

1. Seštevanje ulomkov

- Ker je ta naloga mišljena kot lahka, ne pričakujemo, da bodo tekmovalci kaj dosti razmišljali o tem, ali lahko pride pri uporabi aritmetike s plavajočo vejico do kakšnih napak zaradi omejene natančnosti pri predstavitvi ne-celih števil. Rešitev, ki uporablja tip **double** (ali float v pythonu), lahko dobi vse točke prav tako kot rešitev, ki računa le s celimi števili.

- Naloga zahteva, da rezultat zaokrožimo na najbližji kilogram. Rešitvam, ki rezultata ne zaokrožijo ali pa vedno zaokrožijo navzdol, naj se zaradi tega odšteje pet točk. Rešitvam, ki napačno zaokrožijo le polovične kilograme (npr. navzdol namesto navzgor; ali pa če uporabijo pythonovo funkcijo round, ki zaokroža k najbližjemu sodemu številu), naj se zaradi tega odšteje dve točki.

2. Slovar

- V nekaterih programskih jezikih je morda že v standardni knjižnici kakšna funkcija za preverjanje, ali se niz ujema s takim vzorcem, kakršne uporabljamo pri tej nalogi; v tem primeru ni nič narobe, če si rešitev pomaga s takšno funkcijo, namesto da bi sama v zanki primerjala znake niza z znaki vzorca. (Primer: v pythonu lahko uporabimo modul fnmatch, le da moramo v naših vzorcih spremeniti zvezdice v vprašaje; ali pa spremenimo zvezdice v pike in potem uporabimo razrede za delo z regularnimi izrazi, ki so na voljo tako v C++ kot v pythonu).

- Pri podnalogi (b) je težko vnaprej reči, česa vsega se utegnejo tekmovalci domisliti. Želimo si predvsem, da se uspejo nekako izogniti potrebi po preiskovanju celega slovarja pri vsaki poizvedbi. Rešitev, ki vedno pregleda vse tiste besede, ki so enako dolge kot vzorec, naj dobi pri tej podnalogi 4 točke od sedmih.

- V primerih, ko ima vzorec več zaporednih črk, bi se dalo dosedanjo rešitev podnaloge (b) še izboljšati tako, da bi besede slovarja oz. njihove sufikse zlagali v drevesa po črkah (*trie*) in s pomočjo takih dreves prišli do še manjših seznamov kandidatov (dobili bi vse besede, ki se ujemajo z vzorcem v več zaporednih črkah, ne le v eni črki kot pri dosedanji rešitvi). Vendar pa od tekmovalcev ne pričakujemo, da bodo razmišljali o čem takem.

- Rešitev sme narediti razumne predpostavke o tem, kakšne besede so v slovarju; na primer, da niso daljše od nekaj 10 znakov.

- Od tekmovalcev ne pričakujemo, da vedo, koliko črk ima abeceda ali kakšne so njihove kode v tabeli ASCII; rešitvam, ki glede teh stvari vsebujejo napačne konstante, naj se zaradi tega odšteje največ eno točko.

3. Genialno

- Pri tej nalogi pričakujemo večinoma rešitve s časovno zahtevnostjo $O(wh(w+h))$; take naj dobijo največ 15 točk, če so sicer pravilne. Rešitve z manjšo časovno zahtevnostjo, kot je na primer naša rešitev v času $O(wh)$, naj dobijo vse točke, če so sicer pravilne.
- Če bi rešitev pri računanju, koliko točk dobimo ob postavitvi žetona na določeno mesto, prištela zraven tudi število dosedanjih žetonov na tem mestu, naj se ji zaradi tega odšteje tri točke.
- Rešitev lahko prebere opis mreže s standardnega vhoda, iz datoteke ali pa predpostavi, da ga dobi v neki tabeli, seznamu, vektorju ali čem podobnem; vse te možnosti so enako dobre.

4. Parkirišče

- Pri tej nalogi je pomembna predvsem ideja, da sledimo spremembam (prihodom in odhodom tovornjakov) naraščajoče po času. Ni pa mišljeno, da bi se tekmovalčeva rešitev kaj ukvarjala s tem, kakšen postopek uporabiti za urejanje časov sprememb.
- Namesto urejanja časov bi se dalo tudi vsakič preiskati celoten seznam tovornjakov, da ugotovimo, ob katerem času pride do naslednje spremembe. To je ekvivalentno rešitvi, ki bi čase prihodov in odhodov uredila z urejanjem z izbiranjem ali kakšnim podobnim postopkom, ki ima časovno zahtevnost $O(n^2)$. Ker učinkovitost urejanja ni predmet te naloge, lahko tudi take rešitve dobijo vse točke, če so sicer pravilne.
- Naloga posebej omenja, da lahko hkrati pride in odide več tovornjakov in da se v takem primeru šteje, kot da so se odhodi zgodili pred prihodi. Če kakšna rešitev tega ne upošteva in lahko zato dobi napačen rezultat (npr. ker se za hip zdi, da so novi tovornjaki prišli, stari pa še niso odšli in da zato potrebujemo daljše pakririšče), naj se ji zaradi tega odšteje pet točk.
- Cela števila, s katerimi so predstavljeni časi, so razmeroma velika, da bi spodbudili reševalce k urejanju časov. Če bi kakšna rešitev poskušala uporabiti te čase kot indekse v neko gromozansko tabelo (in v njej označevati, kdaj pride do sprememb v dolžini parkirišča), naj se ji zaradi tega odšteje štiri točke.

5. Barvanje stolpnic

- Pri tej nalogi je poudarek predvsem na opažanju, da je koristno stolpnice barvati od višjih proti nižjim. Zaželeno je, da vsebuje tekmovalčev odgovor tudi nekakšno utemeljitev, zakaj je to res, vendar prav veliko v tej smeri ne moremo pričakovati. Če rešitev ne vsebuje niti poskusa take utemeljitve, naj se ji zaradi tega odšteje štiri točke. V naši rešitvi je tudi dokaz, da naš postopek vedno, ko je to mogoče, najde barvanje z dvema barvama namesto s tremi, vendar tovrstnega dokaza od tekmovalčevega odgovora ne pričakujemo.

- Če barvamo stolpnice v kakšnem drugem vrstnem redu, na primer od leve proti desni, se lahko zgodi, da bomo porabili več kot tri barve; takim rešitvam, ki sicer vrnejo veljavno barvanje (tako, pri katerem nista nobeni dve povezani stolpnici pobarvani z isto barvo), vendar porabijo preveč barv, naj se zaradi tega odšteje 5 točk. (To velja seveda za rešitve, ki se vsaj trudijo uporabiti čim manj barv; ni pa mišljeno, da bi dobili ljudje netrivialno število točk npr. za rešitev, ki preprosto pobarva vsako stolpnico s svojo barvo.)

- Rešitve s časovno zahtevnostjo $O(n^2)$ naj dobijo največ 18 točk, če so drugače pravilne; tiste s časovno zahtevnostjo $O(n^3)$ največ 16 točk; in tiste z eksponentno časovno zahtevnostjo največ 10 točk.

Težavnost nalog

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

Naloga	Kam bi sodila po težavnosti na državnem tekmovanju ACM
1. Seštevanje ulomkov	lažja naloga v prvi skupini
2. Slovar	srednje težka naloga v prvi ali lažja v drugi skupini[34]
3. Genialno	srednje težka naloga v prvi ali lažja v drugi skupini[35]
4. Parkirišče	srednje težka naloga v drugi ali lažja v tretji skupini
5. Barvanje stolpnic	težja naloga v drugi ali lažja v tretji skupini

Če torej na primer neki 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.

[34] Podnaloga (a) sama po sebi bi lahko bila lažja naloga za prvo skupino, podnaloga (b) pa bi bila bolj za drugo skupino.

[35] Težavnost te naloge je odvisna od tega, kako točkujemo rešitve odvisno od njihove učinkovitosti. Naloga bi postala težja, če bi zahtevali rešitev v času $O(wh)$, ker pa smo predvideli 15 točk (od dvajsetih) že za veliko preprostejšo rešitev v času $O(wh(w+h))$, ta naloga ni tako težka.

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 tretji skupini izjemoma podelili eno prvo in tri druge 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 188) in smo jih letos podelili sedem. 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 98 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	Nik Jenič	4	ŠC N. mesto, SEŠTG	20	20	18	20	20	98
1Z	2	Bor Brudar	3	ŠC N. mesto, SEŠTG	15	20	17	20	19	91
2S	3	Rok Perko	2	Vegova Ljubljana	20	18	10	20	15	83
2S	4	Jure Maček	2	Vegova Ljubljana	19	20	13	20	8	80
3S	5	Klemen Pregelj	3	ŠC N. Gorica, Teh. g.	15	14	18	20	10	77
3S		Filip Trplan	4	Gimnazija Vič	14	20	18	20	5	77
S	7	Luka Papež	4	Gimnazija Vič	8	20	8	20	19	75
S	8	Jaka Kovač	2	Vegova Ljubljana	18	20	7	20	5	70
S	9	Tristan Zore	2	ŠC N. mesto, SEŠTG	13	14	11	19	12	69
S		Miha Šparl	2	Gimnazija Vič	15	20	4	20	10	69
S	11	Matic Vernik	3	SERŠ Maribor	8	15	17	20	8	68
S	12	Mark Škof	5	SŠTS Šiška	10	13	16	20	8	67
S		Veno Jakomin	4	STŠ Koper	13	20	9	20	5	67
S	14	Lan Bajželj	2	Vegova Ljubljana	15	19	8	19	5	66
S		Gašper Nemgar	2	Vegova Ljubljana	15	17	8	17	9	66
S		Žan Skopec	5	SŠTS Šiška	9	20	17	20	0	66
S	17	Domen Sedlar	3	ŠC N. mesto, SEŠTG	10	17	10	20	8	65
S	18	Enej Breskvar	1	ZRI	15	13	14	16	5	63
S	19	Domen Korenini	2	Gimnazija Vič	10	20	10	20	1	61
S		Luka Logar	4	ŠC Celje, Gim. Lava	15	19	8	19	0	61
S		Maj Zabukovnik	2	ŠC Celje, SŠ za KER	13	18	7	20	3	61
S	22	Sanja Dumenčić	3	ŠC N. Gorica, ERŠ	15	8	15	19	3	60
S	23	Luka Pršina	3	ŠC N. mesto, SEŠTG	10	13	7	19	10	59
S	24	Tomaž Šimunić	4	SERŠ Maribor	15	9	10	19	5	58
S		Domen Trontelj	4	I. gimnazija v Celju	15	17	9	17	0	58
S	26	Tim Nahtigal	3	ŠC N. mesto, SEŠTG	10	10	17	20	0	57
S	27	Tevž Beškovnik	4	SERŠ Maribor	10	7	9	20	9	55
S		Tomaž Černe	4	ŠC N. mesto, SEŠTG	15	14	6	19	1	55
S	29	Bor Kajin	3	ŠC N. Gorica, GZŠ	13	2	18	19	2	54
S		Florijan Tušak	4	ŠC Ptuj, ERŠ	13	12	9	17	3	54
S	31	Gregor Virant	3	ŠC Celje, Gim. Lava	10	16	9	15	3	53
S	32	Martin Jereb	4	Gimnazija Vič	3	10	19	19	1	52
S		Erik Radovičevič	3	ŠC N. mesto, SEŠTG	5	19	8	20	0	52
S	34	Dominik Krašovec	3	ŠC N. mesto, SEŠTG	10	8	7	16	10	51
S	35	Lucijan Škof	1	ZRI	9	9	7	19	5	49
S		Žan Škorja	2	ŠC Celje, SŠ za KER	8	13	3	15	10	49
S		Sebastjan Vidergar	4	Vegova Ljubljana	10	7	9	20	3	49
	38	Tomi Božak	4	ŠC Celje, Gim. Lava	15	2	7	20	4	48
		Anej Predovnik	1	ŠC Ptuj, ERŠ	15	0	8	20	5	48
		Žiga Terbovc	4	ŠC Celje, Gim. Lava	13	15	18	1	1	48

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
41	Danijel Tomić	3	STPŠ Trbovlje	13	13	0	20	1	47
	Teo Škrabič	1	ZRI	9	0	19	19	0	47
43	Lan Gradišek	3	ŠC Celje, Gim. Lava	19	12	10	1	4	46
44	Žan Hribar	3	STPŠ Trbovlje	17	8	7	11	1	44
	Martin Šalamon	4	SŠTS Šiška	8	13	4	19	0	44
46	Vid Ošep	2	Gimnazija Vič	15	0	7	19	2	43
47	Martin Murko	2	ZRI	8	8	7	18	0	41
48	Tjaš Paradiž	4	ŠC Celje, Gim. Lava	15	0	7	17	1	40
49	Domen Brcar	4	ŠC N. mesto, SEŠTG	8	2	20	8	1	39
50	Matej Markuža	3	STŠ Koper	20	0	11	0	7	38
51	Nejc Pestotnik	4	SŠTS Šiška	13	19	1	1	3	37
	Gregor Ahlin	3	STŠ Koper	5	0	2	11	19	37
53	Diego Bonaca	3	STŠ Koper	1	0	7	9	19	36
54	Tomaž Gril	4	ŠC N. mesto, SEŠTG	5	0	10	18	2	35
55	Jan Mušič	5	STŠ Koper	8	0	7	18	0	33
56	Janez Malovrh	3	Gimnazija Poljane	5	8	0	17	0	30
	Jaka Smrkolj	2	Vegova Ljubljana	8	2	0	20	0	30
	Matej Starc	3	ŠC N. Gorica, ERŠ	10	0	8	0	12	30
59	Dominik Brezovšek	2	ŠC Celje, SŠ za KER	10	7	10	1	1	29
	Armen Hodža	3	ŠC N. Gorica, ERŠ	13	7	7	1	1	29
	Jurij Mahne	2	Vegova Ljubljana	15	0	5	9	0	29
62	Matic Bernot	4	STPŠ Trbovlje	8	5	7	5	2	27
63	Maks Jagodič	4	ŠC Celje, Gim. Lava	13	3	10	0	0	26
64	Ajda Heric	1	II. gimnazija Maribor	0	2	5	18	0	25
65	Ferizi Fisnik	4	SERŠ Maribor	5	0	1	13	0	19
	Jaka Lazarevič	3	Gimnazija Poljane	10	2	5	1	1	19
67	Anže Prevodnik	5	STŠ Koper	8	0	5	0	5	18
68	Nik Rozman	1	ŠC Kranj, STŠ Kranj	9	2	4	0	0	15
69	Anej Bregant	3	SERŠ Maribor	1	6	2	1	2	12
	Tine Ivanovič	7	ACM OŠ priprave	10	0	2	0	0	12
	Miha Kos	2	ŠC Celje, SŠ za KER	10	0	2	0	0	12
72	Gašper Macedoni	1	Gimnazija Poljane	3	5	0	0	0	8
73	Gašper Lebar	2	SPTŠ Murska Sobota	1	0	0	5	0	6
74	Matevž Keber	2	GSŠRM Kamnik	5	0	0	0	0	5
75	Luc Nahtigal Hočevar	1	Gim. Novo mesto	2	0	0	1	0	3
76	Blaž Prša	2	SPTŠ Murska Sobota	0	0	0	1	0	1
77	Domen Perčič	8	ACM OŠ priprave	0	0	0	0	0	0
	Jernej Rogač	2	SPTŠ Murska Sobota	0	0	0	0	0	0
	Den Zudič	5	STŠ Koper	0	0	0	0	0	0

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Adam Janko Koležnik	3	II. gimnazija Maribor	18	19	12	14	19	82
1Z	2	Nikola Brković	4	Gimnazija Bežigrad	14	19	5	20	17	75
2Z	3	Luka Svenšek	4	SPTŠ Murska Sobota	10	18	9	20	15	72
2S	4	Tilen Šket	4	I. gimnazija v Celju	13	15	11	14	13	66
3S	5	Luka Stražišar	4	Gimnazija Vič	12	19	0	15	16	62
3S	6	Tim Thuma	3	Vegova Ljubljana	14	10	9	14	13	60
S	7	Marko Gartnar	4	Vegova Ljubljana	2	15	11	13	14	55
S	8	Žan Ambrožič	3	Gimnazija Kranj	7	8	10	9	19	53
S	9	Rene Jelen	3	II. gimnazija Maribor	0	17	8	15	11	51
S		Andrej Matos	4	Vegova Ljubljana	9	15	12	12	3	51
S		Adrian Sebastian Šiška	3	Vegova Ljubljana	14	18	3	12	4	51
S	12	Igor Setnikar	1	Gimnazija Vič	14	1	9	13	10	47
S	13	Tim Hrovat	3	Vegova Ljubljana	7	10	6	10	11	44
S	14	Žiga Kralj	4	Vegova Ljubljana	12	15	9	1	4	41
S		Peter Žavcer	4	Gimnazija Poljane	8	8	9	16	0	41
	16	Anton Luka Šijanec	3	Gimnazija Bežigrad	20	10	1	0	9	40
	17	Tilen Juričan	3	ZRI	16	0	9	6	8	39
	18	Aljaž Travnik	4	Vegova Ljubljana	9	10	0	16	3	38
	19	Luka Lorenci	4	II. gimnazija Maribor	12	12	7	5	0	36
	20	Matic Kovač	3	ZRI	8	12	12	0	0	32
		Urban Krepel	2	ŠC Velenje, ERŠ	16	3	4	9	0	32
	22	Tine Zaletelj	3	Gimnazija Vič	5	8	10	5	3	31
	23	Andrej Anžlovar	3	Gimnazija Vič	14	0	9	6	0	29
	24	Miha Govedič	4	II. gimnazija Maribor	0	11	9	6	0	26
		Nik Vodovnik	3	ZRI	14	0	0	12	0	26
	26	Tim Strnad	2	ZRI	11	0	12	0	0	23
	27	Rok Hladin	4	I. gimnazija v Celju	0	10	0	4	0	14
	28	Željan Kondić	1	Gimnazija Kranj	13	0	0	0	0	13
	29	Aljaž Remec	1	Gimnazija Vič	3	0	0	0	0	3
	30	Anej Žaler	3	GSŠRM Kamnik	0	0	0	0	0	0
		Urban Županič	3	II. gimnazija Maribor	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	Benjamin Bajd	4	ZRI	100	97	97	100	100	494
2Z	2	Luka Urbanc	2	ZRI	100	30	60		0	190
2S	3	Anže Hočevar	4	Gimnazija Vič	80	27	34		30	171
2S	4	Jošt Smrtnik	3	Gimnazija Vič	90		80			170
3S	5	Jakob Žorž	2	ZRI	100	24	31			155
3S	6	Matej Kralj	4	Gimnazija Vič	87		17		50	154
S	7	Lovro Sikošek	4	Gimnazija Brežice		30	40		30	100
	8	Gašper Korbar	4	Vegova Ljubljana	80		15			95
	9	Jakob Kralj	3	Gimnazija Vič	38	20				58
	10	Filip Štamcar	3	ZRI		7			50	57
	11	Tim Tisak	4	Vegova Ljubljana		6	40			46
	12	Lara Stamać	2	ZRI		0	40			40
	13	Samo Golež	4	GSŠRM Kamnik		10	0			10
	14	Jan Mrak	4	GSŠRM Kamnik		0	0	0	0	0
		Luka Hrovatič	4	Gimnazija Brežice	0					0

VRSTNI RED ŠOL

Da bi spodbudili šole k čim večji udeležbi in čim boljšim rezultatom v vseh treh skupinah, smo začeli leta 2018 objavljati tudi vrstni red šol v neke vrste skupnem seštevku. Posamezni šoli prinesejo točke najboljši štirje tekmovalci iz te šole v prvi skupini, najboljši trije v drugi in najboljša dva v tretji skupini. Točke šole so enake vsoti točk njenih tekmovalcev. Točke, ki jih prispeva tekmovalec k vsoti, se izračuna tako, da se delež točk (od vseh možnih točk), ki jih je ta tekmovalec dosegel na tekmovanju, pomnoži z utežjo za skupino, v kateri je tekmoval. Utež za prvo skupino je 100, za drugo skupino 200 in za tretjo skupino 300.

Mesto	Šola	Točke
1	Gimnazija Vič	766,6
2	Vegova Ljubljana	715,6
3	II. gimnazija Maribor	363
4	ŠC Novo mesto, SEŠTG	323
5	Gimnazija Bežigrad	230
6	I. gimnazija v Celju	218
7	SŠTS Šiška	214
8	ŠC Celje, Gimnazija Lava	210
9	SERŠ Maribor	200
10	STŠ Koper	178
11	ŠC Celje, SŠ za KER	151
	SPTŠ Murska Sobota	151
13	Gimnazija Poljane	139
14	Gimnazija Kranj	132
15	ŠC Nova Gorica, ERŠ	119
16	STPŠ Trbovlje	118
17	ŠC Ptuj, ERŠ	102
18	ŠC Nova Gorica, Tehniška gimnazija	77
19	ŠC Velenje, ERŠ	64
20	Gimnazija Brežice	60
21	ŠC Nova Gorica, Gimnazija in zdravstvena šola	54
22	ŠC Kranj, STŠ Kranj	15
23	GSŠRM Kamnik	11
24	Gimnazija Novo mesto	3

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Nik Jenič	telefon Samsung Galaxy S21 FE 5G
1	1	Bor Brudar	telefon Samsung Galaxy S21 FE 5G
1	2	Rok Perko	telefon Samsung Galaxy A52S
1	2	Jure Maček	telefon Samsung Galaxy A52S
1	3	Klemen Pregelj	telefon Samsung Galaxy A52S
1	3	Filip Trplan	telefon Samsung Galaxy A52S
2	1	Adam Janko Koležnik	telefon Samsung Galaxy S21 FE 5G Cormen *et al.: Introduction to Algorithms*
2	1	Nikola Brković	telefon Samsung Galaxy S21 FE 5G S. in F. Halim: *Competitive Programming 4*
2	2	Luka Svenšek	telefon Samsung Galaxy A52S S. in F. Halim: *Competitive Programming 4*
2	2	Tilen Šket	telefon Samsung Galaxy A52S
2	3	Luka Stražišar	telefon Samsung Galaxy A52S
2	3	Tim Thuma	miška Razer DeathAdder V2
3	1	Benjamin Bajd	telefon Samsung Galaxy S21 FE 5G Raspberry Pi 4 model B S. in F. Halim: *Competitive Programming 4*
3	2	Luka Urbanc	telefon Samsung Galaxy S21 FE 5G Raspberry Pi 4 model B Cormen *et al.: Introduction to Algorithms*
3	2	Anže Hočevar	telefon Samsung Galaxy S21 FE 5G Cormen *et al.: Introduction to Algorithms*
3	2	Jošt Smrtnik	telefon Samsung Galaxy A52S
3	3	Jakob Žorž	telefon Samsung Galaxy A52S
3	3	Matej Kralj	telefon Samsung Galaxy A52S
Off-line naloga — Poplavljanje			
	1	Matej Kralj	Raspberry Pi 4 model B
	3	Gregor Kikelj	Raspberry Pi 4 model B

SODELUJOČE ŠOLE IN MENTORJI

ACMova skupina za osnovnošolske priprave na računalniška tekmovanja	Daniel Sami Blažič, Bor Grošelj Simić, Matija Likar, Ella Potisek, Patrik Žnidaršič
II. gimnazija Maribor	Matija Likar, Mitja Osojnik, Mirko Pešec
Gimnazija Bežigrad	Andrej Šuštaršič
Gimnazija Brežice	Tea Habinc
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Gregor Hrastnik, Miran Zevnik
Gimnazija in srednja šola Rudolfa Maistra Kamnik (GSŠRM)	Vinko Kušar
Gimnazija Kranj	Mateja Žepič
Gimnazija Novo mesto	Barbara Strnad
Gimnazija Poljane	Janez Malovrh, Boštjan Žnidaršič
Gimnazija Vič	Klemen Bajec, Marina Trost
I. gimnazija v Celju	Sebastjan Tkavc
Srednja elektro-računalniška šola Maribor (SERŠ)	Dušan Fugina, Slavko Nekrep, Branko Potisk, Manja Sovič Potisk
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Dominik Letnar
Srednja šola tehniških strok Šiška (SŠTS)	Tom Kamin
Srednja tehniška in poklicna šola Trbovlje (STPŠ)	Uroš Ocepek
Srednja tehniška šola Koper (STŠ)	Senka Felicijan, Katarina Novoselec
Šolski center Celje, Gimnazija Lava	Karmen Kotnik
Šolski center Celje, Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Žiga Pušelc
Šolski center Kranj, Srednja tehniška šola (STŠ)	Miha Baloh
Šolski center Nova Gorica, Elektrotehniška in računalniška šola (ERŠ)	Aljaž Gec, Tomaž Mavri

Šolski center Nova Gorica, Gimnazija in zdravstvena šola	Marko Marčetić, Barbara Pušnar
Šolski center Nova Gorica, Tehniška gimnazija	Marko Marčetić, Barbara Pušnar
Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija (SEŠTG)	Simon Vovko, Albert Zorko
Šolski center Ptuj, Elektro in računalniška šola (ERŠ)	Marjan Čeh, David Drofenik, Franc Vrbančič
Šolski center Rogaška Slatina	Jože Vajdič
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Miran Zevnik
Vegova Ljubljana	Marko Kastelic, Melita Kompolšek, Nataša Makarovič, Aleš Volčini, Darjan Toth
Zavod za računalniško izobraževanje (ZRI), Ljubljana	

REZULTATI CERC 2022

Ker smo letos organizirali srednjeevropsko študentsko tekmovanje v računalništvu (CERC 2022) pri nas v Ljubljani, objavljamo v našem biltenu še rezultate tega tekmovanja. Naloge so na str. 35–52, rešitve pa na str. 123–180.

	Ekipa	Št. rešenih nalog	Čas
1	Antoni Długosz, Grzegorz Gawryał, Jacek Salata (Jag. u.)	10	26:55:41
2	Tomasz Nowak, Bartłomiej Czarkowski, Arkadiusz Czarkowski (U. v Varšavi)	10	28:38:21
3	Adam Rajský, Josef Minařík, Jiři Kalvoda (Karlova u.)	8	15:29:53
4	Patrick Pavić, Krešimir Nežmah, Dorijan Lendvaj (U. v Zagrebu)	8	18:29:37
5	Vladyslav Denysiuk, Roman Yanushevskyi, Oleh Naver (Jag. u.)	8	19:19:46
6	Antoni Wiśniewski, Rafał Łyżwa, Kacper Kluk (U. v Varšavi)	8	22:17:47
7	Péter Szente, Péter Varga, Péter Gyimesi (ELTE)	8	22:51:52
8	Attila Gáspár, Bence Deák, Máté Busa (ELTE)	7	16:36:04
9	Dominik Wawszczak, Jakub Dziura, Piotr Blinowski (U. v Varšavi)	7	22:13:07
10	Jan Wańkowicz, Hubert Obrzut, Łukasz Pluta (U. v Wrocławu)	7	22:52:09
11	Rafał Pyzik, Jan Klimczak, Justyna Jaworska (Jag. u.)	7	24:07:14
12	Michał Kępa, Michał Maras, Dominik Kowalczyk (U. v Wrocławu)	7	25:42:45
13	Daniil Zabauski, Bogdan Tolstik, Pavel Sankin (Jag. u.)	6	13:52:34
14	Michał Staniewski, Marek Skiba, Jan Kwiatkowski (U. v Varšavi)	6	15:07:37
15	Mateusz Orda, Bartosz Chomiński, Marcel Szelwiga (U. v Wrocławu)	6	18:20:25
16	Krzysztof Boryczka, Marcin Knapik, Adam Zyzik (U. v Wrocławu)	6	18:33:08
17	Kacper Topolski, Maksym Zub, Andrii Kuts (Jag. u.)	6	22:57:28
18	Félix Moreno Peñarrubia, Tymofii Reizin, Ondřej Sladký (Karlova u.)	6	24:54:40
19	Tomáš Macháček, Michal Staník, Daniel Ilkovič (Masarykova u.)	5	11:46:08
20	Kamil Zwierzchowski, Janusz Partyka, Juliusz Korab-Karpowicz (U. v Varšavi)	5	16:51:21
21	Bojan Štetić, Leon Jurić, Dominik Fistrić (U. v Zagrebu)	4	9:07:40
22	Jaroslav Urban, Xuan Thang Nguyen, Martin Prokopič (CTU)	4	9:24:41
23	Boris Španić, Petar Sruk, Ivan Jambrešić (U. v Zagrebu)	4	10:59:56
24	Domen Hočevar, Benjamin Bajd, Job Petrovčič (U. v Ljubljani)	4	11:38:22
25	Daniel Skýpala, Robert Jaworski, Matouš Šafránek (Karlova u.)	4	12:57:11
26	Csaba Dékány, Miklós Csizmadia, Balázs Makrai-Kis (ELTE)	3	8:12:15
27	Hubert Dyczkowski, Cyryl Szatan, Joanna Suwaj (U. v Wrocławu)	3	9:24:25
28	Krzysztof Jaworski, Adam Pawłowski, Rafał Szubert (PUT)	3	10:58:09
29	Kristofers Barkāns, Krišjānis Petručeņa, Sandra Silina (Latvijska u.)	3	11:01:09
30	Łukasz Skabowski, Paweł Czarkowski, Paweł Aniszewski (U. N. Kopernika)	3	11:15:34
31	Krzysztof Szymański, Jeremiasz Preiss, Michał Opala (U. v Wrocławu)	3	12:22:23
32	Bor Grošelj Simić, Patrik Žnidaršič, Jakob Schrader (U. v Ljubljani)	3	13:04:18
33	Miroslav Matějček, Andrej Pajtaš, Karel Poncar (CTU)	3	14:26:11
34	Illés Tamás Iles, István Megyeri (U. v Szegedu)	2	3:24:53
35	Marcin Zepp, Filip Nikolow, Jan Izydorczyk (AGH)	2	4:01:02
36	Ramal Salha, Filip Kadak, Karlo Iletić (U. v Osijeku)	2	5:00:52
37	Nikolina Rodin, Lucija Žužić, Jakov Tomasić (U. na Reki)	2	5:38:31

(*nadaljevanje na naslednji strani*)

REZULTATI CERC 2022 (*nadaljevanje*)

	Ekipa	Št. rešenih nalog	Čas
38	Jozef Koleda, Aleš Sršeň, Adam Barla (CTU)	2	5:59:40
39	Martin Domajnko, Jakob Kordež (U. v Mariboru)	2	6:15:53
40	Daniil Pastukhov, Artem Savchenko, Nikolay Tsoy (CTU)	2	7:02:33
41	Nóra Bánhidai, Gergely Péter, Nyerges Bálint (U. v Szegedu)	2	7:09:49
42	Erik Rassai, Barna Kirchhof, Ádám Horváth (BUTE)	2	7:13:26
43	Norbert Micheľ, Branislav Pastula, Matej Uhrin (UPJŠ)	2	16:34:45
44	Norbert Vígh, Timotej Králik, Marek Ceľuch (STU)	1	2:16:40
45	Adrián Kabáč, Veronika Paulinyová, Šimon Ukuš (STU)	1	2:51:17
46	Rafael Krstačić, Alan Burić, Alan Bubalo (U. v Pulju)	1	3:01:37
47	Ella Potisek, Urban Duh, Marko Hostnik (U. v Ljubljani)	1	3:44:25
48	Aljaž Žel, Alen Granda, Mitko Nikov (U. v Mariboru)	1	4:07:14
49	Arnis Priedītis, Roberts Cerins, Gunars Abeltins (Latvijska u.)	1	4:25:16
50	Filip Kocháň, Yuliia Teslia, Šimon Brauner (Masarykova u.)	1	4:32:16
51	Michal Kováčik, Tomáš Gerát, Lukáš Mlkvik (U. v Žilini)	1	4:50:58
52	Benedek Brandschott, Gábor Galgóczy, Dániel Kovács (BUTE)	0	0:00:00
	Vojtěch Volný, Ondřej Sivek, Daniel Dobeš (VŠB)	0	0:00:00
	Marek Nieslanik, Vojtěch Mikulenka, Matěj Murgaš (VŠB)	0	0:00:00
	Miloš Murín, Mário Husár, Tobiáš Mitala (U. v Žilini)	0	0:00:00
	Dimitar Pelivanov, Jovan Pavlović, Milan Milivojčević (U. na Primorskem)	0	0:00:00

Sodelovale so ekipe z naslednjih univerz:

Češka tehniška univerza (CTU) (Praga, Češka)
Jagielonska univerza (Krakow, Poljska)
Karlova univerza (Praga, Češka)
Latvijska univerza (Riga, Latvija)
Masarykova univerza (Brno, Češka)
Poznanjska tehnična univerza (PUT) (Poznanj, Poljska)
Slovaška tehniška univerza (STU) (Bratislava, Slovaška)
Tehniška univerza v Ostravi (VŠB) (Češka)
Univerza Josipa Juraja Strossmayerja v Osijeku (Hrvaška)
Univerza Juraja Dobrile v Pulju (Hrvaška)
Univerza Pavola Jozefa Šafárika v Košicah (UPJŠ) (Slovaška)
Univerza Loránda Eötvösa (ELTE) (Budimpešta, Madžarska)
Univerza na Primorskem (Slovenija)
Univerza na Reki (Hrvaška)
Univerza Nikolaja Kopernika (Torunj, Poljska)
Univerza v Ljubljani (Slovenija)
Univerza v Mariboru (Slovenija)
Univerza v Szegedu (Madžarska)
Univerza v Varšavi (Poljska)
Univerza v Wrocławu (Poljska)
Univerza v Zagrebu (Hrvaška)
Univerza v Žilini (Slovaška)
Univerza za tehnologijo in ekonomiko (BUTE) (Budimpešta, Madžarska)
Znanstveno-tehnična univerza AGH (Krakow, Poljska)

OFF-LINE NALOGA — POPLAVLJANJE

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 zmogel, saj mu za to preprosto zmanjka časa.

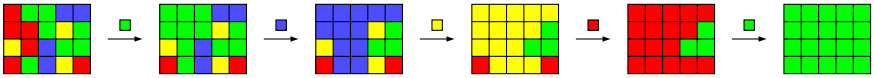
Off-line naloga je poskus, da se tovrstnim omejitvam malo izognemo: besedilo naloge in testni primeri zanjo so objavljeni več mesecev vnaprej, tekmovalci pa ne oddajajo programa, ki rešuje nalogo, pač pa oddajajo rešitve tistih vnaprej objavljenih testnih primerov. Pri tem imajo torej veliko časa in 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 decembra 2021, nekaj mesecev po razpisu za tekmovanje v znanju; tekmovalci so imeli čas do 25. marca 2022 (dan pred tekmovanjem), da pošljejo svoje rešitve.

Opis naloge

Dana je karirasta mreža, v kateri je vsaka celica pobarvana z določeno barvo. Barve celic lahko spreminjamo, vendar le na naslednji način: izberemo si neko barvo in začnemo mrežo „poplavljati“ (*flood fill*) z njo v zgornji levi celici. Nova barva se iz zgornje leve celice razširi tudi na tiste njene sosede, ki so bile prej enake barve kot ona, iz njih pa spet na tiste njihove sosede, ki so bile prej enake barve kot zgornja leva celica, in tako naprej na vse celice, ki so dosegljive na ta način. (Celici veljata za sosednji, če imata skupno eno od stranic.)

Tvoja naloga je s čim manj takšnimi poplavljanji doseči, da bodo vse celice mreže pobarvane z isto barvo (ni važno, katero).

Primer mreže in enega od možnih zaporedij poplavljanj:



Rezultati

Sistem točkovanja je bil tak kot pri off-line nalogah v prejšnjih letih. Pripravili smo 30 testnih primerov, pri vsakem testnem primeru smo razvrstili tekmovalce po oceni njegove rešitve, nato pa je prvi tekmovalec (tisti, čigar rešitev je imela najmanjšo oceno) dobil 10 točk, drugi 8, tretji 7 in tako naprej po eno točko manj za vsako naslednje mesto (osmi dobi dve točki, vsi nadaljnji pa po eno). Na koncu smo za vsakega tekmovalca sešteli njegove točke po vseh 30 testnih primerih.

Testni primeri so bili treh različnih velikosti: deset majhnih (30×30 celic), deset srednjih (100×100 celic) in deset velikih (300×300 celic). Pripravili smo jih naključno, in sicer po naslednjem postopku: vnaprej si izberemo, koliko „zaplat“ (povezanih skupin celic enake barve) bomo imeli, recimo k; nato si izberemo k naključnih celic in vsako razglasimo za zametek po ene zaplate; nato pa na vsakem koraku pogledamo najmanjšo zaplato ter jo povečamo tako, da ji dodamo eno (naključno izbrano) izmed celic, ki mejijo nanjo in ki doslej še niso bile dodeljene k

nobeni od zaplat. Število k je bilo pri majhnih mrežah od 30 do 300, pri srednjih od 300 do 4500, pri velikih pa od 3000 do 50 000. Ko je tako mreža razdeljena na zaplate, si izberemo še število barv b (od 6 do 24) in jo pobarvamo s požrešnim postopkom: na vsakem koraku pobarvamo eno od še nepobarvanih zaplat z naključno izbrano barvo, pri čemer pazimo le na to, da ni iste barve že prej dobila kakšna od zaplat, ki se je dotikajo.

Letos je svoje rešitve pri off-line nalogi poslalo osem tekmovalcev, od tega štirje srednješolci in dva študenta. Končna razvrstitev je naslednja:

Mesto	Ime	Letnik	Šola	Točke
1	Matej Kralj	4	Gimnazija Vič	296
2	Anže Hočevar	4	Gimnazija Vič	250
3	Gregor Kikelj	3	FMF	246
4	Luka Stražišar	4	Gimnazija Vič	238
5	Matjaž Leonardis			188
6	Luka Papež	4	Gimnazija Vič	174
7	Bor Grošelj Simić	1	FMF	156
8	Samo Kralj			125

Rešitev

Za začetek lahko našo karirasto mrežo v mislih predelamo v graf: za vsako zaplato (povezano skupino celic enake barve) imejmo po eno točko, med dvema točkama pa naj bo (neusmerjena) povezava v natanko tistih primerih, kjer njuni dve zaplati mejita druga na drugo (z vsaj eno stranico med dvema sosednjima poljema). Poleg tega si je za vsako zaplato koristno zapomniti morda še to, koliko celic jo sestavlja, po tistem pa se nam s prvotno mrežo ni treba več ukvarjati, ampak za reševanje naloge gledamo le še graf. Poplavljanje zgornje leve zaplate (torej tiste, ki ji pripada zgornja leva celica mreže) se na tem grafu odraža v tem, da se točka, ki predstavlja to zaplato, združi z nekaterimi svojimi sosedami v grafu (tistimi, ki so bile take barve, s kakršno smo pravkar poplavili mrežo); ko ostane le še ena sama točka, je to znak, da se je zgornja leva zaplata razširila že na celo mrežo, torej je cela mreža ene barve in lahko s poplavljanjem končamo.

Za manjše testne primere dobimo precej dobre rešitve že s preprostim požrešnim algoritmom, pri katerem na vsakem koraku poplavimo s tako barvo, pri kateri se bo površina zgornje leve zaplate najbolj povečala; ali pa s tako barvo, pri kateri se zgornja leva zaplata zlije s čim več drugimi zaplatami, ki so bile doslej drugačne barve od nje; ali pa barvo izberemo celó naključno (izmed tistih, pri katerih bi se zgornja leva zaplata potem vsaj malo povečala; postopek poženemo po večkrat ter obdržimo najboljšo izmed tako dobljenih rešitev). Še boljša hevristika je, da v grafu poiščemo najkrajše poti od točke, ki predstavlja zgornjo levo zaplato, do vseh ostalih točk, nato pa izbiramo barve pri poplavljanju tako, da se bo maksimalna dolžina teh najkrajših poti (po vseh ostalih točkah) čim bolj zmanjševala.

Še ena preprosta izboljšava je, da namesto požrešnega algoritma uporabimo iskanje v snopu (*beam search*). Pri tem vzdržujemo hkrati več (recimo N — nekaj sto ali nekaj tisoč) rešitev; na vsakem koraku poskušamo na vsaki od njih izvesti po eno poplavljanje z vsako možno barvo; med vsemi tako dobljenimi novimi rešitvami pa

obdržimo spet le N najboljših (glede na enega od kriterijev iz prejšnjega odstavka). Postopek se konča, čim najdemo kakšno tako rešitev, pri kateri je pobarvana cela mreža.

Še boljšo rešitev pa dobimo, če našo nalogo zapišemo kot optimizacijski problem. Naj bo V množica vseh točk v našem grafu (oz. zaplat na naši mreži), B pa množica vseh barv, ki se pojavljajo v začetnem stanju mreže. Število vseh zaplat označimo z $n = |V|$. Vnaprej si izberimo neko dovolj veliko število T, za katero vemo, da je s T poplavljanji gotovo mogoče doseči, da bo cela mreža ene barve (uporabimo na primer število poplavljanj iz ene od rešitev, ki smo jih dobili s prej opisanimi pristopi).

Za vsak časovni korak t z območja $1 \le t \le T$ in za vsako barvo $b \in B$ vpeljimo celoštevilsko spremenljivko x_{tb}, ki pove, ali smo v t-tem koraku poplavljali z barvo b ($x_{tb} = 1$) ali ne ($x_{tb} = 0$). Poleg tega za vsak t z območja $0 \le t \le T$ in vsako točko $u \in V$ našega grafa vpeljimo celoštevilsko spremenljivko y_{tu}, ki pove, ali je bila zaplata u v prvih t korakih že poplavljena ($y_{tu} = 1$; in se je torej zlila z zgornjo levo zaplato) ali ne ($y_{tu} = 0$).

V posameznem časovnem koraku lahko seveda poplavljamo le z eno barvo; lahko pa v kakšnem tudi sploh ne poplavljamo — tako si bomo pustili možnost, da dobimo rešitev z manj kot T poplavljanji, ne z natanko T poplavljanji. Da pa ne bo prostor možnih rešitev po nepotrebnem prevelik, dodajmo pravilo, da ko enkrat začnemo poplavljati, potem ne smemo več nehati. Vpeljimo torej omejitve:

$$\textstyle\sum_b x_{t-1,b} \le \sum_b x_{tb} \text{ za } t = 2, \ldots, T \text{ in } 0 \le \sum_b x_{1b} \text{ in } \sum_b x_{Tb} \le 1.$$

V prvih nekaj korakih torej morda sploh ne poplavljamo (in je $\sum_b x_{tb} = 0$), odtlej pa vsakič poplavljamo s po eno barvo (in je $\sum_b x_{tb} = 1$).

Na začetku ni še nobena zaplata zlita z zgornjo levo, zato imejmo omejitev $y_{0u} = 0$ za vse $u \in V$ razen za zgornjo levo zaplato sámo, kjer pa je $y_{0u} = 1$. Na koncu pa morajo biti zlite vse zaplate, da bo mreža res v celoti ene barve; vpeljimo torej omejitev $y_{Tu} = 1$ za vse $u \in V$.

V splošnem pa velja, da je zaplata u poplavljena v prvih t korakih (in mora torej biti $y_{tu} = 1$), če (a) je bila bodisi poplavljena že v prvih $t-1$ korakih (torej je bilo že $y_{t-1,u} = 1$) bodisi (b) je bila v prvih $t-1$ korakih poplavljena ena od njenih sosed v grafu (naj bo $N(u)$ množica sosed točke u), v t-tem koraku pa smo poplavljali ravno s tisto barvo, s katero je bila v prvotnem stanju mreže pobarvana zaplata u (tej barvi recimo β_u).

Za možnost (a) lahko poskrbimo tako, da dodamo omejitve $y_{tu} \ge y_{t-1,u}$ za vse $t = 1, \ldots, T$ in $u \in V$: ko je zaplata u enkrat poplavljena (in združena z zgornjo levo zaplato), potem tudi ostane poplavljena.

Da bo manj pisanja, vpeljimo pomožno oznako

$$s_{t-1,u} = \textstyle\sum_{v \in N(u)} y_{t-1,v};$$

zanjo torej velja, da če je bila vsaj ena od u-jevih sosed poplavljena v prvih $t-1$ korakih, potem je $s_{t-1,u}$ večji od 0 in manjši od n, sicer pa je enak 0.

Razmislimo zdaj o možnosti (b). Prehod iz $y_{t-1,u} = 0$ v $y_{tu} = 1$ moramo dovoliti v primerih, ko je $x_{t,\beta_u} = 1$ (da poplavljamo s pravo barvo) in hkrati $s_{t-1,u} > 0$ (da je bil eden od u-jevih sosedov že prej združen z zgornjo levo zaplato). Če seštejemo

$(n-1)\cdot x_{t,\beta_u} + s_{t-1,u}$, ima ta vsota naslednjo lepo lastnost: če sta bila izpolnjena oba pogoja, $x_{t,\beta_u} = 1$ in $s_{t-1,u} > 0$, potem je naša vsota gotovo $\geq n$, sicer pa je gotovo manjša od n. Zdaj lahko torej vpeljemo omejitve:

$$n \cdot (y_{tu} - y_{t-1,u}) \leq (n-1) \cdot x_{t,\beta_u} + s_{t-1,u}$$

za vse $t = 1, \ldots, T$ in vse $u \in V$. Ta omejitev torej poskrbi, da bo $y_{tu} - y_{t-1,u}$ lahko ≥ 1 le v primerih, ko je $x_{t,\beta_u} = 1$ in $s_{t-1,u} > 0$, torej bo le takrat lahko zaplata u prišla iz nepoplavljenega stanja $y_{t-1,u} = 0$ v poplavljeno stanje $y_{tu} = 1$; v vseh ostalih primerih pa bo morala biti razlika $y_{tu} - y_{t-1,u}$ vedno < 1, čemur bo (ker že iz prej vpeljanih omejitev sledi, da bo vedno veljalo $0 \leq y_{t-1,u} \leq y_{tu} \leq 1$) mogoče ustreči le tako, da bo stanje zaplate u ostalo nespremenjeno: $y_{tu} = y_{t-1,u}$.

Tako smo torej dobili optimizacijski problem, pri katerem iščemo takšne vrednosti vseh spremenljivk x_{tb} in y_{tu}, ki vse ležijo na območju $\{0, 1\}$ in ustrezajo doslej vpeljanim omejitvam. Vsak tak nabor vrednosti spremenljivk predstavlja neko veljavno zaporedje T ali manj poplavljanj, ki pobarva celotno mrežo z eno barvo. (Zaporedje barv v posameznih korakih lahko odčitamo tako, da pri vsakem t pogledamo, katera od spremenljivk x_{tb} ima vrednost 1.) Ker pa nas zanima čim krajše zaporedje poplavljanj, moramo v resnici najti takšen nabor vrednosti spremenljivk, ki ima (ob upoštevanju omejitev) čim manjšo vsoto $\sum_t \sum_b x_{tb}$.

Za reševanje tega problema uporabimo kakšno od knjižnic, kot je na primer Googlova *OR-Tools*, ki je prosto dostopna in jo je prav preprosto uporabljati iz npr. pythona. Optimizacijski problemi, kot je tale naš, so praviloma NP-težki in tudi od takšne knjižnice seveda ne smemo pričakovati čudežev.[36] Pri manjših testnih primerih bomo lahko z njo našli optimalno rešitev (torej najkrajše možno zaporedje poplavljanj), pri večjih primerih pa optimizacijskega problema ne bomo mogli rešiti v sprejemljivem času. Takrat pa lahko opisani pristop uporabimo vsaj za to, da poskusimo izboljšati rešitev, ki smo jo dobili s kakšnim drugim postopkom, na primer iskanjem v snopu. Oglejmo si nekaj (recimo m) zaporednih poplavljanj pri tej rešitvi; omejimo se le na tiste zaplate, ki so bile v teh poplavljanjih prvič združene z zgornjo levo zaplato; na tako omejenem problemu poiščimo najkrajše možno zaporedje poplavljanj (tako, da rešimo pravkar opisani optimizacijski problem); če najdemo rešitev z manj kot m poplavljanji, lahko torej z njo skrajšamo našo rešitev celotnega problema. To ponavljamo na raznih mestih v naši rešitvi celotnega problema, dokler še uspemo kje najti kakšno izboljšavo. Večji m ko vzamemo, več možnosti imamo, da bomo res izboljšali rešitev, vendar nam tudi reševanje optimizacijskega problema vzame več časa (ker bo vanj prišlo več zaplat); pri naših poskusih na testnih primerih z letošnjega tekmovanja je bilo smiselno iti z m-jem tja do 15.

[36] Da je problem, ki smo ga dobili, NP-težak, sledi že iz tega, da smo nanj prevedli naš prvotni problem (torej kako s čim manj poplavljanji doseči, da bo celotna mreža ene barve), kajti tudi ta je NP-težak (D. Arthur *et al.*, "The complexity of flood filling games", *Fun with Algorithms, 5th Int. Conf.* (FUN 2010), Springer LNCS 6099, str. 307–18; kot zanimivost omenimo, da so NP-težkost v tem članku dokazali tako, da so na problem poplavljanja prevedli problem najkrajšega skupnega nadniza, s katerim smo se že srečali tudi na naših tekmovanjih — imeli smo ga za off-line nalogo leta 2017).

UNIVERZITETNI PROGRAMERSKI MARATON

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

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

UPM poteka v štirih krogih (dva spomladi in dva jeseni), pri čemer se za končno razvrstitev pri vsaki ekipi zavrže najslabši rezultat iz prvih treh krogov, četrti (finalni) krog pa se šteje dvojno. Najboljše ekipe se uvrstijo na srednjeevropsko regijsko tekmovanje (CERC, ki je potekalo 26.–27. novembra 2022 v Ljubljani), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bi moralo v normalnih razmerah potekati spomladi 2023, vendar je bilo zaradi zakasnitev, povezanih z epidemijo v prejšnjih letih, preloženo in je potekalo 14.–19. aprila 2024 v Luksorju v Egiptu).

Na letošnjem UPM je sodelovalo 23 ekip s skupno 63 tekmovalci, ki so prišli s treh slovenskih univerz, nekaj pa je bilo celo srednješolcev. Tabela na naslednji strani prikazuje vse ekipe, ki so se pojavile na vsaj enem krogu tekmovanja.

	Ekipa	Št. rešenih nalog*	Čas
1	Job Petrovčič (FMF), Benjamin Bajd (Gim. Kranj), Domen Hočevar (FRI + FMF)	26	50:35:09
2	Žiga Željko, Marko Hostnik (FRI + FMF), Urban Duh (FMF)	24	34:51:26
3	Jakob Schrader, Patrik Žnidaršič, Daniel Sami Blažič (FMF)	20	36:11:41
4	Mitko Nikov, Alen Granda, Aljaž Žel (FERI)	15	25:14:27
5	Bor Grošelj Simić, Ella Potisek (FMF), Matija Likar (II. gimn. Maribor)	13	28:38:59
6	Vid Drobnič, Matej Marinko (FRI + FMF), Žiga Patačko Koderman (FRI)	12	10:37:38
7	Matevž Miščič, Jon Mikoš (FMF)	10	12:45:41
8	Jakob Kordež, Martin Domajnko (FERI)	10	17:34:59
9	Matija Kocbek, Luka Horjak (FMF), Lovro Drofenik (F. za strojništvo, Lj.)	10	18:08:05
10	Gregor Kikelj (FMF), Luka Grbec, Mia Grbec (FRI)	8	19:39:57
11	Tadej Tomažič (FMF), Tomaž Tomažič (—), Veronika Tomažič (Škof. gimn. Vipava)	6	7:03:32
12	Jelena Glišić, Milan Milivojčević, Jovan Pavlović (FAMNIT)	4	4:48:31
13	Bor Brecelj, Laura Guzelj Blatnik (FRI + FMF), Zala Erič (FRI)	3	4:09:08
14	Amadej Šenk, Luka Tomažič, Martin Dagarin (FRI)	3	4:39:53
15	Filip Štamcar, Jakob Kralj, Jošt Smrtnik (Gim. Vič)	3	5:29:13
16	Tina Poštuvan (FRI), Sara Veber (FRI + FMF)	3	7:26:32
17	Tjaž Silovšek (FRI + FMF), Tim Vučina, Marcel Tori (FRI)	3	8:26:36
18	Jure Pustoslemšek, Jure Mihelčič (FRI + FMF), Ana Arnež (F. za farm.)	3	11:34:44
19	Luka Lonec, Rene Klement (FERI)	2	3:29:27
20	Nina Sangawa Hmeljak, Ema Leila Grošelj (FRI + FMF), Jan Hrastnik (FMF)	2	6:36:07
21	Dimitar Pelivanov, Todor Antić (FAMNIT)	2	9:01:47
22	Iztok Bajcar, Zala Pregelj, Gašper Spagnolo (FRI)	1	4:00:07
23	Domen Hribernik, Žan Kazar (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 CERC 2022 so (z nekaterimi spremembami v sestavi) nastopile ekipe 1, 2 in 3 kot predstavnice Univerze v Ljubljani, ekipi 4 in 8 kot predstavnici Univerze v Mariboru in ekipa 12 kot predstavnica Univerze na Primorskem. V konkurenci 56 ekip s 24 univerz iz 7 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
24	Domen Hočevar, Benjamin Bajd, Job Petrovčič	4	11:38:02
32	Bor Grošelj Simić, Patrik Žnidaršič, Jakob Schrader	3	13:04:18
39	Martin Domajnko, Jakob Kordež	2	6:15:53
47	Ella Potisek, Urban Duh, Marko Hostnik	1	3:44:25
48	Aljaž Žel, Alen Granda, Mitko Nikov	1	4:07:14
52	Dimitar Pelivanov, Jovan Pavlović, Milan Milivojčević	0	0:00:00

Na srednjeevropskem tekmovanju je bilo 12 nalog, od tega sta jih najboljši ekipi rešili po deset.

ANKETA

Lani in predlani, ko je tekmovanje potekalo prek interneta, smo na ta način izvedli tudi anketo (prek spletne strani `1ka.si`). Ker se je ta način anketiranja dobro obnesel, smo ga obdržali tudi letos, čeprav je tekmovanje drugače spet potekalo v živo. Vprašanja na anketi so prikazana spodaj in so bila približno enaka kot včasih, ko je bila anketa na papirju. Rezultati ankete so predstavljeni na str. 237–244.

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++ (v 1. skupini pa tudi v pythonu).
— Ali razumeš C++ (oz. python) 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? _____

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, pythonu in rustu. 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 oz. funkcijo	☐	☐	☐
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:
— razpršeno tabelo: hash_map, hash_set, unordered_map, unordered_set (v C++), Hashtable, HashSet (v javi/C#), Dictionary (v C#), dict, set (v pythonu) ☐ ☐ ☐
— iskalna drevesa: map, set (v C++), TreeMap, TreeSet (v javi), SortedDictionary (v C#) ☐ ☐ ☐
— kopico oz. prioritetno vrsto: 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?

Ali si pri izpolnjevanju ankete prišel/la do sem? ☐ da ☐ ne

Hvala za sodelovanje in lep pozdrav! Tekmovalna komisija

REZULTATI ANKETE

Anketo je izpolnilo 37 tekmovalcev prve skupine, 12 tekmovalcev druge skupine in 7 tekmovalcev tretje skupine. (Opozorimo na to, da je zaradi majhnega števila tekmovalcev v tretji skupini iz tamkajšnjih anket še posebej težko vleči kakšne pametne posplošitve in zaključke.)

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. 238. 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 približno tako težke kot ponavadi, v tretji skupini mogoče malce težje. 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: 3,35 v prvi skupini (v prejšnjih letih 3,27, 2,97, 3,47, 3,32, 3,11), 3,39 v drugi skupini (prejšnja leta 3,55, 3,38, 3,17, 3,19, 3,51) in 3,80 v tretji skupini (prejšnja leta 3,63, 3,67, 3,52, 3,59, 3,73).

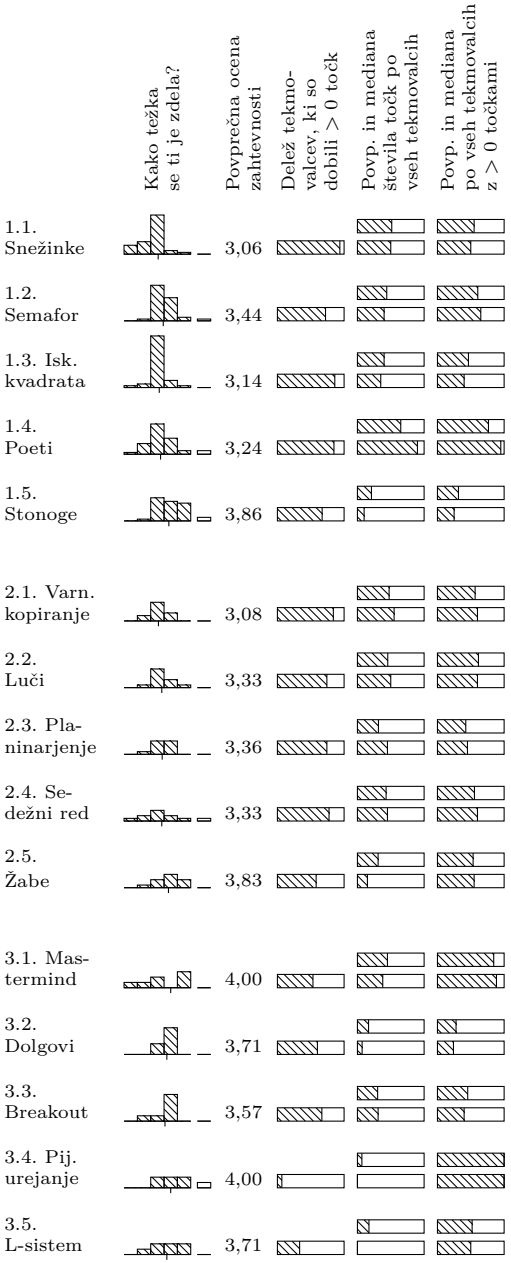
Med tem, kako težka se je naloga zdela tekmovalcem, in tem, kako dobro so jo zares reševali (npr. merjeno s povprečnim številom točk pri tej nalogi), je ponavadi (šibka) negativna korelacija; letos je bila šibkejša kot v prejšnjih nekaj letih ($R^2 =$ 0,48; v prejšnjih letih 0,42, 0,68, 0,71, 0,67, 0,70).

V prvi skupini so tekmovalci kot težjo ocenili predvsem nalogo 1.5 (stonoge), ki sicer ni zelo težka, je pa malo nekonvencionalna. Kot najlažjo pa so ocenili nalogo 1.1 (snežinke), kar je zanimivo, saj se takšne „realnočasovne“ naloge tekmovalcem ponavadi zdijo težke.

V drugi skupini je izstopala naloga 2.5 (žabe), ki se je večini ljudi zdela težka ali pretežka, kar ima sicer morda več zveze s formulacijo naloge kot s samim algoritmom, ki se ga je treba domisliti (tudi besedilo te naloge se je mnogim zdelo dolgo in težje razumljivo). Kot najlažjo pa so v tej skupini ocenili nalogo 2.1 (varnostno kopiranje).

V tretji skupini sta se jim zdeli težji predvsem nalogi 3.1 (mastermind) in 3.4 (pijansko urejanje), kar morda kaže na to, da se pri tako majhnem številu prejetih anket iz te skupine na njihove rezultate ne smemo preveč zanašati. Naloga 3.1 je namreč v nekem smislu najlažja, saj so tekmovalci pri njej dosegli več točk kot pri katerikoli drugi. Za nalogo 3.4 je manj presenetljivo, da se jim je zdela težka, saj gre za interaktivno nalogo, na kakršne so tekmovalci manj navajeni. Najlažja v tej skupini se jim je zdela naloga 3.3 (breakout).

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 239. Nad razumljivostjo besedil ni veliko pripomb, podobno kot prejšnja leta, v tretji skupini še malo manj. Kot težje razumljive so ocenili predvsem naloge 1.2 (semafor), 2.5 (žabe) in 3.1 (mastermind); pri slednji je sicer zanimivo to, da so jo kljub temu reševali bolje kot katerokoli drugo nalogo v tretji skupini.



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

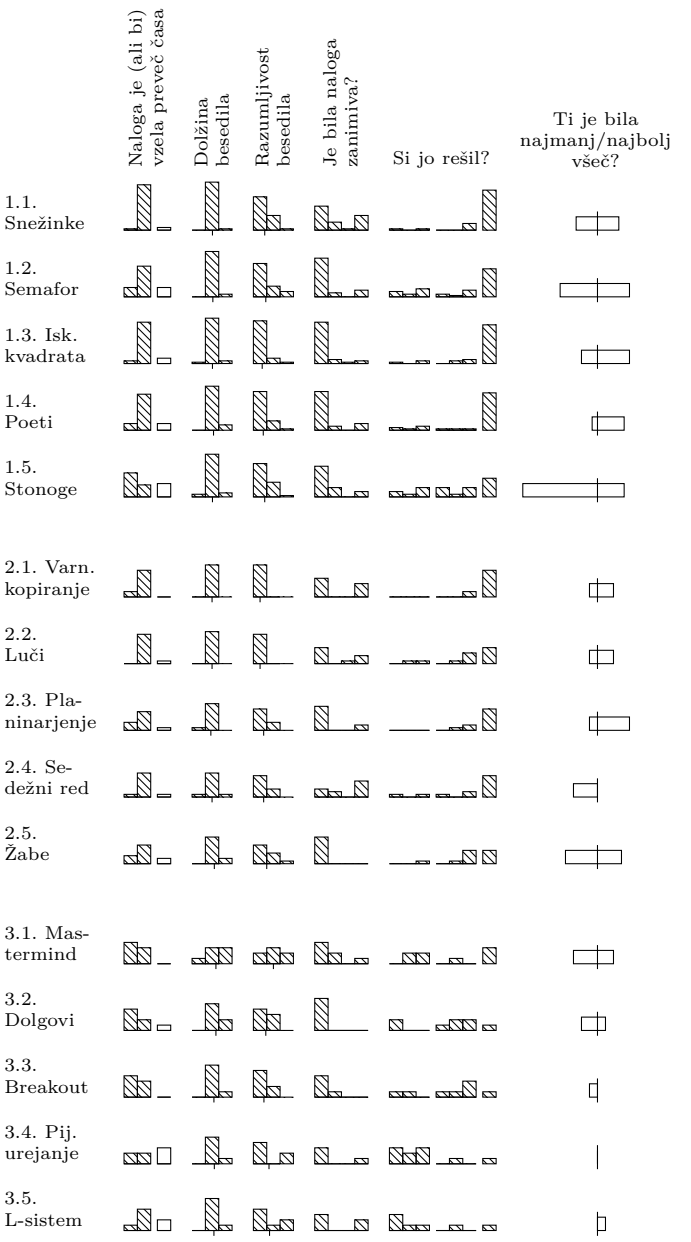
Pomen stolpcev v vsaki vrstici:

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?

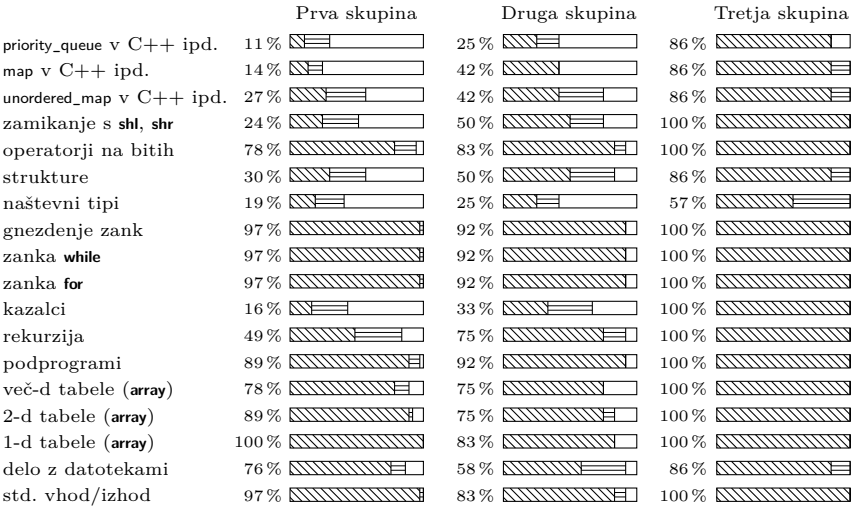


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

Tudi z dolžino besedil so tekmovalci večinoma zadovoljni; ocene so podobne kot prejšnja leta. Po komentarjih, da je naloga predolga, še najbolj izstopajo naloge 1.4 (neprevidni poeti), 3.1 (mastermind) in 3.2 (dolgovi). Mnenj, da je besedilo prekratko, je bilo malo, še največ pri nalogi 1.5 (žabe); pri tej nalogi se je sicer nekaterim zdelo besedilo tudi predolgo.

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta. Kot bolj zanimive izstopajo 1.3 (iskanje kvadrata), 2.3 (planinarjenje), 2.5 (žabe) in 3.2 (dolgovi), kot manj zanimivi pa 2.4 (sedežni red) in 3.1 (mastermind). Pripomb, da jim je neka naloga že znana, je bilo letos zelo malo: po ena pri nalogah 1.1 (snežinke), 1.3 (iskanje kvadrata) in 2.2 (luči).

Pripomb, da bi naloga vzela preveč časa, je bilo malo, v tretji skupini sicer več kot ponavadi. Največ takih pripomb je bilo pri nalogah 1.5 (stonoge; to je tudi naloga, ki se jim je zdela v tej skupini najtežja) in 3.2 (dolgovi), pa tudi 3.1 (mastermind) in 3.3 (breakout). Pri zadnjih treh je z implementacijo rešitve res še kar nekaj dela (tudi če gledamo npr. dolžino izvorne kode naših rešitev).

Pri vprašanjih, katera naloga je tekmovalcu najbolj všeč, so bili glasovi letos precej razpršeni med naloge, še posebej v prvi skupini; v drugi pa kot bolj priljubljena izstopa 2.3 (planinarjenje). Pri vprašanju, katera naloga jim je najmanj všeč, pa izrazito izstopa naloga 1.5 (žabe); verjetno ni naključje, da je to naloga, ki se je mnogim zdela tudi težka in slabo razumljiva. Nekaj nalog je dobilo veliko glasov pri obeh vprašanjih, npr. 1.2 (semafor).

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

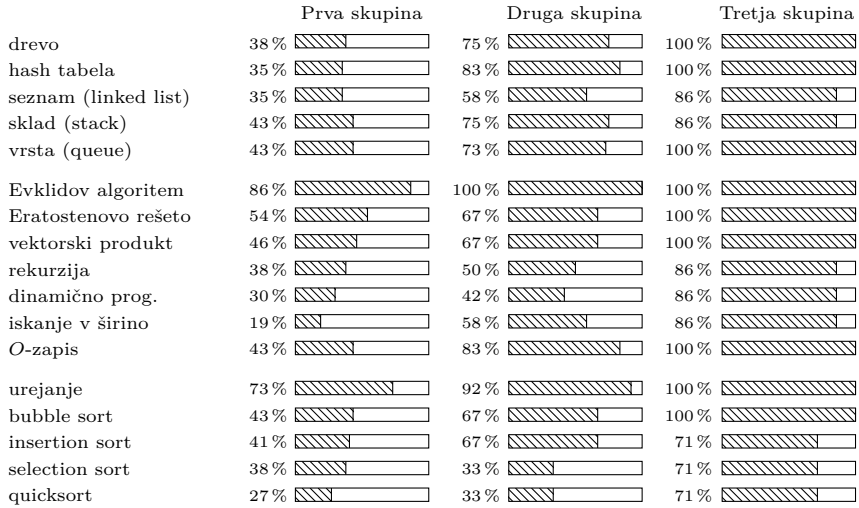


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

podatkovno strukturo. Zato jih v anketah zadnjih nekaj let sprašujemo, če te reči poznajo in bi jih znali uporabiti v svojih programih.

Rezultati pri vprašanjih o programerskem znanju so podobni tistim iz prejšnjih let. V prvi in tretji skupini pravijo, da znajo malo več kot tisti v lanski anketi. Stvari, ki jih tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja leta: kazalci, naštevni tipi in operatorji na bitih, v prvi in drugi skupini tudi strukture in rekurzija. Poznavanje operatorjev na bitih in rekurzije je letos boljše kot ponavadi.

Uporaba programskih jezikov

Na splošno so razmerja med različnimi jeziki podobna kot v prejšnjih letih. V prvi skupini je tudi letos python daleč najpogostejši, na drugem mestu pa je tudi letos java; sledita jima C++ in C, C# pa je razmeroma redek, podobno kot lani. Tudi v drugi skupini je python najpogostejši, s precejšnjim zaostankom pa mu sledi C/C++. V tretji skupini je še vedno najpogostejši C++, nekaj tekmovalcev pa je uporabljalo python. Jave ni v drugi in tretji skupini uporabljal nihče. Edina jezika, ki sta se še pojavila poleg doslej omenjenih, sta javascript (ki sta ga uporabljala dva tekmovalca v prvi in eden v drugi skupini) in rust (ki ga je uporabljal en tekmovalec v tretji skupini). Slednji se je letos na naših tekmovanjih pojavil prvič (in je bil tudi prvič podprt v tretji 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

	Leto in skupina																	
	2022			2021			2020			2019			2018			2017		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal				4			2			2						4		
C	3	$3\frac{1}{2}$		8	5	$\frac{1}{2}$	$3\frac{1}{2}$	3		10	4	$\frac{1}{2}$	5	4	$\frac{1}{2}$	4	3	$2\frac{1}{2}$
C++	14	6	10	13	$11\frac{1}{2}$	$18\frac{1}{2}$	$26\frac{1}{2}$	8	14	$21\frac{1}{2}$	$7\frac{1}{2}$	18	$18\frac{1}{2}$	13	11	23	10	$15\frac{1}{2}$
java	22			17		5	15	4	3	15	5	1	$21\frac{1}{2}$	$8\frac{1}{2}$	4	28	3	2
PHP			–			–	1		–			–			–			–
C#	2	1		6	4		6	3		12	2		11	6		7	6	
python	29	$16\frac{1}{2}$	4	43	$15\frac{1}{2}$	4	48	20	3	$36\frac{1}{2}$	$26\frac{1}{2}$	$6\frac{1}{2}$	38	11	$\frac{1}{2}$	42	11	–
javascript	2	1	–	1	2	–	2		–			–		$\frac{1}{2}$	–			–
julia			–			–			–			–			–	1		–
swift			–			–	1		–			–			–			–
rust			1			–			–			–			–			–
psevdokoda	4	1	–	3		–	2	1	–	5	1	–	3	1	–	5		–
nič	3	2		3			2			3			2		1			

Število tekmovalcev, ki so uporabljali posamezni programski jezik.

Nekateri uporabljajo po več različnih jezikov (pri različnih nalogah) in se štejejo delno k vsakemu jeziku. (V letu 2022 je en tekmovalec uporabljal python in C, dva pa python in C++.) „Nič“ pomeni, da tekmovalec ni napisal nič izvorne kode (niti psevdokode, pač pa morda rešitve v naravnem jeziku). 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“.

---

stilu „nato bi s stavkom **if** preveril, ali je spremenljivka x večja od spremenljivke y“). Podobno kot prejšnja leta smo tudi letos pri nalogah tipa „opiši postopek“ pripisali „ali napiši podprogram (kar ti je lažje)“ (kjer je bilo to primerno).

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže gornja tabela. Glede štetja C in C++ v tej tabeli je treba pripomniti, da je razlika med njima majhna in včasih pri kakšnem krajšem kosu izvorne kode že težko rečemo, za katerega od obeh jezikov gre. Je pa po drugi strani videti, da se raba stvari, po katerih se C++ loči od C-ja, sčasoma povečuje; zdaj že veliko tekmovalcev na primer uporablja `string` namesto **`char *`** in tip `vector` namesto tradicionalnih tabel (*arrays*). Novosti, po katerih se zadnje različice C++ (od vključno C++11 naprej) razlikujejo od C++98, je letos uporabljalo kar precej tekmovalcev, še posebej v tretji skupini (npr. ranged **`for`**, **`auto`** v novem pomenu, pri enem tekmovalcu tudi lambda-izrazi).

Pri pythonu zdaj 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 podoben kot prejšnja leta (36/37 v prvi skupini in 11/12 v drugi). Kot običajno so pri vprašanju, ali bi želeli deklaracije še v kakšnem jeziku, nekateri tekmovalci navedli jezike, v katerih deklaracije že imamo, na primer C++ ali C#; od originalnih predlogov pa sta se po enkrat pojavila javascript in rust. V vsakem primeru pa se poskušamo zadnja leta v besedilih nalog izogibati deklaracijam v konkretnih programskih jezikih in jih zapisati bolj na splošno, na primer „napiši funkcijo `foo(x, y)`“ namesto „napiši funkcijo **`bool`** `foo(`**`int`** `x,`

int y)“.

V rešitvah nalog objavljamo od 2017 izvorno kodo v C++, pri prvi skupini pa tudi v pythonu. Tekmovalce smo v anketi vprašali, če razumejo C++ (ali, v prvi skupini, python) 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++ (oz. pythonom) zadovoljna (28/37 v prvi skupini, 12/12 v drugi, 7/7 v tretji); ta delež je podoben kot lani. 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 javo (predvsem v prvi skupini, kar ni čudno, ker je tam veliko tekmovalcev tudi reševalo v javi) ter python in rust. Vendar je s pripravo rešitev v dveh jezikih precej dela, zato bomo do nadaljnjega objavljali rešitve v pythonu (poleg v C++) še vedno le v prvi skupini.

Letnik

Običajno so tekmovalci zahtevnejših skupin večinoma v višjih letnikih kot tisti iz lažjih skupin. Razmerja so podobna kot prejšnja leta; v prvi in drugi skupini so tekmovalci v povprečju malo starejši kot lani, v tretji pa približno enako stari. Letos sta nastopila tudi dva osnovnošolca, oba v prvi skupini.

	Št. tekmovalcev po letnikih								Povprečni
Skupina	7	8	9	1	2	3	4	5	letnik
prva	1	1		8	20	22	22	5	3,0
druga				3	2	14	12		3,1
tretja					3	3	9		3,4

Druga vprašanja

Podobno kot prejšnja leta je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!), je pa bilo letos malo več kot ponavadi takih tekmovalcev, ki so za tekmovanje izvedeli od prijateljev. V smislu širitve zanimanja za tekmovanje in večanja števila tekmovalcev se zelo dobro obnese šolsko tekmovanje, ki ga izvajamo zadnjih nekaj let, saj se odtlej v tekmovanje vključuje tudi nekaj šol, ki prej na našem državnem tekmovanju niso sodelovale.

Pri vprašanju, kje so se naučili programirati, je podobno kot prejšnja leta najpogostejši odgovor, da so se naučili programirati sami (takih so dobre tri četrtine); sledijo tisti, ki so se tega naučili v šoli pri pouku, in tisti, ki so se naučili programirati (tudi) na krožkih in tečajih (obojih je po približno dve petini).

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni; njihov delež je še malo višji kot lani. Med ostalimi so mnenja precej razdeljena, najpogostejši kombinaciji pa sta „več časa, manj (ali enako) nalog“ in (redkeje) „enako časa, manj nalog“.

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb; tudi posebnih tehničnih težav letos ni bilo. Nekaj tekmovalcev je želelo, da bi lahko svoje odgovore v prvi in drugi skupini oddajali kot datoteke (namesto da jih pišejo ali lepijo v obrazec na spletni strani), nekaj pa, da bi želeli imeti možnost po tekmovanju priti do svojih rešitev (slednje sicer brez težav pošljemo tistim, ki nam po tekmovanju pišejo, da bi jih radi dobili).

	Kje si izvedel za tekmovanje				Kje si se naučil programirati					Čas reševanja			Število nalog		
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
I	34	0	7	1	26	16	7	5	4	8	3	26	3	9	25
II	12	0	1	0	12	4	5	2	1	2	0	10	1	1	10
III	7	0	0	0	6	2	3	2	2	1	1	4	0	4	3

V preteklosti si je veliko tekmovalcev želelo tudi, da bi imeli v prvi in drugi skupini na računalnikih prevajalnike in podobna razvojna orodja. Razlog, zakaj smo se v teh dveh skupinah izogibali prevajalnikom, je bil 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. Toda v letih 2020 in 2021, ko so zaradi epidemije vsi reševali naloge doma in torej dostop do prevajalnikov in razvojnih orodij imeli, se je pokazalo, da te težave vendarle niso nastopile; tekmovalci so se večinoma lotili vseh nalog in rezultati v prvi skupini so bili še boljši kot ponavadi. Zato smo letos, ko je tekmovanje spet potekalo v živo namesto prek interneta, omogočili tekmovalcem prve in druge skupine tudi uporabo prevajalnikov. To je bilo v anketi večinoma lepo sprejeto, je pa bilo tudi nekaj pripomb, ker razpoložljiva orodja nekaterim niso ustrezala; zlasti je bilo, kot kaže, slabše poskrbljeno za tiste, ki so želeli programirati v javascriptu.

Nekaj tekmovalcev si je v anketi tudi želelo, da bi v prvi in drugi skupini uvedli avtomatsko ocenjevanje, podobno kot v tretji. Toda pri takem načinu ocenjevanja postane tekmovanje težje, ker lahko dobiš več kot nič točk šele, če napišeš delujoč program, ki pravilno reši vsaj nekatere testne primere; rešitve, ki jih oddajajo tekmovalci prve in druge skupine, pa so pogosto še daleč od tega. Če bi torej uvedli avtomatsko ocenjevanje v prvi dve skupini, bi morali dajati še lažje naloge kot doslej (ali pa bi imeli v rezultatih veliko tekmovalcev s po 0 točkami, kar bi delovalo nespodbudno), poudarek tekmovanja pa bi se s tem premaknil z razmišljanja o algoritmih na zgolj programiranje; tega pa si ne želimo, zato bomo zaenkrat še ostali pri ročnem ocenjevanju.

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) Nekaj letošnjih prispevkov na temo nepotrebnega kompliciranja pri pogojih v neskončnih zankah:

```
while (true = true) // program ves čas ponavljamo

while (1 = 1) // ponavlja

bool a = true; // spremenljivka narejena z namenom, da se lahko naslednji while stavek
while (a) {      // nadaljuje v neskončno
```

(1.1) Zakaj bi prirejali 0, če lahko z njo množimo:

```
cout << counter;
counter *= 0;
```

(1.1) Tale tekmovalec je bil pripravljen na tekmovanje z 999 nalogami:

```
public class N001 {
```

Pri drugi nalogi je imel N002 in tako naprej.

(1.1) Zanimiva sintaktična inovacija: zanka znotraj pogoja v stavku **if**, ki naj bi preverila, če pogoj velja pri vseh n.

```
if (for ((int n = 1; n < 100; n++) { Senzor(n) }) == false) // če senzor ne zazna
                                                            // snežinke, spet pregleduje, dokler ne zazna
```

(1.1) Brutalno: temu tekmovalcu neskončna zanka ni bila dovolj, moral je dodati še neskončno rekurzijo...

```
public static void Main(string[] args)
{
   while (true)
   {
      Meri(); // pokličem metodo Meri
   }
}

public static void Meri()
{
   ⋮
   Main(null); // Kličem nazaj Main, kjer se vse ponovi
}
```

(1.2) Nekdo je ročno deklariral devet ArrayListov. Ko bi le obstajala kakšna primerna podatkovna struktura, v katero bi se jih dalo zložiti...

```
ArrayList <Integer> ena = new ArrayList<Integer>();
⋮
ArrayList <Integer> devet = new ArrayList<Integer>();
```

(1.2) Komentar na koncu popolnoma zgrešene rešitve:

```
# mislim, da moj program v redu deluje, saj zelo hitro najde rešitev
```

(1.2) Komentar tekmovalca, ki je znal poklicati funkciji scanf in printf, funkcije BeriStevec pa ne:

> Ne vem, kako delujejo funkcije v Cju, saj jih nikoli ne uporabljam, zato sem podatke vzel kot integerje na standardnem vhodu.

(1.3) Tale tekmovalec uporablja besedo „kvadrat“ malo drugače, kot smo navajeni:

> Nato dobimo 2 vrednosti: dolžina in širina.
> Če je kvadrat pravokoten, ali podoben pravokotniku, lahko preverimo tako, da izračunamo razliko med njima — absolutno vrednost pri odštevanju.

(1.3) Rešitev z velikimi pričakovanji od spremenljivke š, ki naj bi hkrati vsebovala širine posameznih stolpcev in njihove vsote:

```
for š in š_stolpcev: # hodi skozi seznam š_stolpcev
  opcija = š / v_vrstic.index(i) # vsak int iz š_stolpcev se deli z prvim int iz v_vrstic
  opcije.append(opcija) # opcija se pripne v seznam opcije
  š += š # int v š_stolpcev se seštevajo in se v 7. vrstici ponovno delijo
```

(1.3) Dober letošnji prispevek v žanr opisov postopkov, ki so v resnici opisi izvorne kode:

> Potem naredim **for**(k) od 0 do števila višin krat števila širin. Po koncu tega fora naredim nov **for**(i) od 0 do števila širin, v tem **for**u pa še en **for**(j) od 0 do števila višin.

(1.4) Nekdo je pisal logični ALI s poševnimi črtami namesto navpičnimi:

```
if (crke[i] == 'a' \\ crke[i] == 'e' \\ crke[i] == 'i' \\ crke[i] == 'o' \\ crke[i] == 'u') {
```

(1.4) Rešitev, ki opazi bolj malo velikih črk:

```
if ((pesem[j].charAt(i) == 'a') || (pesem[j].charAt(i) == 'e') ||
    (pesem[j].charAt(i) == 'i') || (pesem[j].charAt(i) == 'o') ||
    (pesem[j].charAt(i) == 'u'))
{
  if (pesem[j].charAt(i).isUpperCase())
```

(1.4) Dekadenten način, kako za vsako črko niza preveriti, ali je samoglasnik:

```
for i in s:
  st = (i.lower().count("a") + i.lower().count("e") + i.lower().count("i") +
        i.lower().count("o") + i.lower().count("u"))
  sez.append(st) # dodamo vrednosti v seznam
```

(1.4) Nagrado za prispevke k jezikoslovju dobi:

```
def samostalniki(s: str):
    out = []
    for i in s:
        if i in ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U']: out.append(i)
    return out
⋮
# najprej iz besedila ločimo samostalnike, ker so to edine črke, ki nas zanimajo
```

(1.4) Prispevek na temo „zavajanje sovražnika“: pogoj `pravilno == 1` pomeni, da pesem *nima* ritma.

```
if (pravilno == 1)
    printf("NE");
```

(1.4) Presenetljiva potrata pri podatkovni strukturi: tale tekmovalec je uporabil `map` (kar je ponavadi nekakšno rdeče-črno drevo) za nekaj, kar bi prav lahko bil vektor ali niz:

```
std::map<int, bool> zaporedje;
int zaporedje_stev = 0;
for (int i = 0; i < max_velikost_vrstice; i++) {
    if (vrstica[i] == 'A' || vrstica[i] == 'E' || vrstica[i] == 'I' ||
        vrstica[i] == 'O' || vrstica[i] == 'U')
    {
        zaporedje.insert(std::pair<int, bool>(zaporedje_stev, 1));
        zaporedje_stev++;
```

(1.5) Neki tekmovalec je namesto obrnjene poševnice `\` pisal `a`, najbrž nato, ker ni vedel, kako obrnjeno poševnico pravilno vključiti v string literal (torej kot `'\\'`). To še ni cvetka; cvetka je način, kako je poskušal spremeniti znake `/` v `\` (oz. `a`) in obratno:

```
noge2 = noge2.replace('/', "a")
noge2 = noge2.replace("a", '/') # prezrcalili spodnje noge in če so enake zgornjim nogam
```

(1.5) Tale rešitev pri stonogah, ki imajo dve nožici usmerjeni v levo, predpostavi, da gre za napako v vhodnih podatkih, in eno pobriše:

```
self.vrsta1 = vrsta1.replace("\\\\", "\\") # neki je šlo narobe pr escapanju
```

(2.1) Zapleten način, kako pobrisati prazne nize iz seznama:

```
def rek(x):
    x.remove('')
    a = ''
    if a in x:
        x = rek(vnos)
    return x
⋮
vnos1 = input()
vnos = vnos1.split("/")
if '' in vnos:
    vnos = rek(vnos)
```

(2.1) Boleče: namesto da bi imel tabelo kazalcev na nize in pri urejanju prerazporejal le te kazalce, premešča spodnji program kar cele 4096-bytne vrstice v tabeli:

```
char line[10000][4096];
⋮
qsort(line, n, sizeof(char) * 4096, cmpfunc); // sortira array line, tako da bodo vrstice
                                              // urejene od najkrajše od najdaljše
```

(2.1) Komentar na začetku ene od rešitev:

```
/* Vse poti, ki jih zelimo izpisati, bomo hranili v set-u, da lahko
 * delamo bisekcijo pri iskanju.
```

„Bisekcija“ pri iskanju:

```
set<string> poti;
⋮
for (auto it = set.begin(); it != set.end(); it++) {
  if (*it.substr(0, mapa.size()) == mapa) {
    set.erase(it); // upam, da je sintaksa prav  :)
```

(Glede komentarja na koncu: sintaksa je sicer prav, semantika pa ne povsem; po brisanju bo iterator it neveljaven, pač pa erase vrne veljaven iterator na naslednji element po vrsti, zato bi bilo bolje narediti it = set.erase(it), le da potem zanj ne smemo izvesti še it++.)

(2.2) Celotna „rešitev“ enega od tekmovalcev:

> Program bi preizkušal vse možne kombinacije ukazov in izpisal najkrajšo.
> (upam, da za to dobim nekaj točk)

(2.4) Nekdo je „spregledal“, da je število vrst podano (n), in si je zato nalogo malo olajšal:

```
# ker količina vrst ni omejena, sklepam, da je učilnica neskončno dolga :D, zato je m = 2
# V vsak stolpec vstavim eno punco in enega fanta
```

(2.4) Za rubriko „tekmovalci čestitajo in pozdravljajo“:

```
// note: kdor misli, da je 5c zloglasni, še ni videl R 2.D v polni moči.
```

(3.1) Nekdo se ne zna odločiti, ali naj si prebrane vhodne podatke zapomni ali ne:

```
b = str(input())
⋮
for j in range (0, n):
  ugibanje.append(b[j * 2])
  ugibanje = []
```

(3.3) Posrečena mnemonika:

```
enum Stran {
  S,  // si
  V,  // včeraj
  J,  // jedel
  Z,  // zelje
}
```

(3.3) Zakaj bi imeli običajne spremenljivke, če pa jih lahko zavijemo v slovar, da bo dostop do njih čim dražji in počasnejši:

```
zoga = { "x": a − 0.5, "y": 0 }
meje = { "leva": 1, "desna": s − 1, "zgornja": v − 1, "spodnja": 0 }
while (meje["leva"] <= zoga["x"] <= meje["desna"] and
       meje["spodnja"] <= zoga["y"] <= meje["zgornja"]):
```

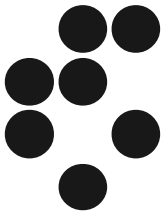
(3.5) Optimističen način, kako pripraviti vse anagrame p-ja: znake niza p je $|p|!$-krat naključno premešal in očitno upal, da bo po srečnem naključju vsakič nastal drugačen anagram.

```
pl = list(p)
⋮
while True:
  for _ in range(factorial(len(p))):
    q = ""
    shuffle(pl)
    for i in pl: q += i
    anagrami.add(q)
  break
```

Bonus točke si zasluži tudi za zelo koristno in zelo neskončno zunanjo zanko **while**.

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.

Zavod Republike Slovenije za šolstvo

Zavod Republike Slovenije za šolstvo

Zavod Republike Slovenije za šolstvo je osrednji nacionalni razvojno-raziskovalni in svetovalni zavod na področju predšolske vzgoje, osnovnega šolstva in splošnega srednješolskega izobraževanja.

ZLATI 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.