16. tekmovanje ACM v znanju računalništva

27. marca 2021

Bilten

Bilten 16. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2023

Elektronska izdaja

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Urban Duh, Gašper Fijavž, Luka Fürst, Primož Gabrijelčič, Matija Grabnar, Tomaž Hočevar, Branko Kavšek, Gregor Kikelj, Vid Kocijan, Filip Koprivec, Samo Kralj, Mitja Lasič, Matija Lokar, Mark Martinec, Polona Novak, Patrik Pavić, Tim Poštuvan, Jure Slak, Mitja Trampuš, Jasna Urbančič, 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`.

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

COBISS.SI-ID=152918019
ISBN 978-961-264-269-3 (PDF)

KAZALO

Struktura tekmovanja 5

Nasveti za 1. in 2. skupino 7
Naloge za 1. skupino 10
Naloge za 2. skupino 14
Navodila za 3. skupino 18
Naloge za 3. skupino 20
Naloge šolskega tekmovanja 25
Naloge s CERC 2021 28
Neuporabljene naloge iz leta 2019 45

Rešitve za 1. skupino 56
Rešitve za 2. skupino 62
Rešitve za 3. skupino 74
Rešitve šolskega tekmovanja 100
Rešitve nalog s CERC 2021 116
Rešitve neuporabljenih nalog 2019 157

Nasveti za ocenjevanje
in izvedbo šolskega tekmovanja 207

Rezultati 212
Nagrade 219
Šole in mentorji 220
Rezultati CERC 2021 222

Off-line naloga: Pokrajina iz kock 224

Univerzitetni programerski maraton 228

Anketa 231
Rezultati ankete 235
Cvetke 243

Sodelujoče inštitucije 251
Pokrovitelji 255

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. (Ker je tekmovanje letos v celoti potekalo prek interneta, je bilo reševanje na papir možno le v izjemnih primerih — če bi tekmovalec sam poskeniral odgovore in jih poslal po elektronski pošti; vendar pa te možnosti letos ni izkoristil nihče.) 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 18.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C#, java in python.

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

Na začetku smo tekmovalcem poslali tudi nekaj navodil in nasvetov (str. 7–9 za 1. in 2. skupino, str. 18–20 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 224–227.

Podobno kot v zadnjih nekaj letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 22. januarja 2021. 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. 207–211). 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 230 tekmovalcev s 25 šol (vse so bile srednje).

Državno tekmovanje je tudi letos zaradi epidemije novega koronavirusa v celoti potekalo prek interneta. Tekmovalci so reševali naloge od doma, vsak na svojem računalniku, in pošiljali odgovore na naš ocenjevalni strežnik. To med drugim pomeni, da so lahko tudi v prvi in drugi skupini uporabljali prevajalnike in druga razvojna orodja, česar sicer pred letom 2020 na tekmovalnih računalnikih v prvi in drugi skupini niso imeli.

Društvo ACM Slovenija je sodelovalo tudi pri organizaciji srednjeevropskega študentskega tekmovanja v računalništvu (CERC 2021, ki je sicer zaradi epidemije poteklo šele aprila 2022 in še to le prek interneta), zato v letošnjem biltenu objavljamo tudi rezultate (str. 222) ter slovenske prevode nalog (str. 28–44) in opise rešitev (str. 116–156) 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

Naloge rešuj samostojno; ne sprašuj drugih ljudi za nasvete ali pomoč pri reševanju (niti v živo niti prek interneta ali kako drugače), ne kopiraj v svoje odgovore tuje izvorne kode in podobno. Tekmovalna komisija si pridržuje pravico, da tekmovalca diskvalificira, če bi se kasneje izkazalo, da nalog ni reševal sam. Internet lahko uporabljaš, če ni v nasprotju s prejšnjimi omejitvami (npr. za branje dokumentacije), vendar za reševanje nalog ni nujno potreben. Tvoje odgovore bomo pregledali in ocenili ročno, 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 (bo)ste dobili po elektronski pošti. Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Ker je vgrajeni urejevalnik dokaj preprost in ne omogoča označevanja kode z barvami, predlagamo, da rešitev pripraviš v urejevalniku na svojem računalniku 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 in zapri“ 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), izjemoma pa tudi po elektronski pošti na `rtk-info@ijs.si`. 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 pri oddaji odgovorov prek spletnega strežnika kakšne težave, lahko izjemoma pošlješ svoje odgovore po elektronski pošti na `rtk-info@ijs.si`, vendar nas morajo doseči pred koncem tekmovanja; odgovorov, prejetih po koncu tekmovanja, ne bomo upoštevali.

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.

1. Gesla

Marjan je pozabil geslo za Apple ID in se ga ne spomni. Kot mnogi drugi ljudje ima tudi on nekaj različnih gesel, ki jih ponavadi uporablja za vse možne storitve (Gmail, Facebook, Instagram itd.). Ta gesla vsebujejo samo male črke angleške abecede in števke (na primer: „`iec4oovi`“, „`eipe9thu`“ in podobno); vsako geslo vsebuje vsaj eno črko. Marjan ni prepričan, katero geslo je sprva nameraval uporabiti za Apple ID, spomni pa se, da je geslo moralo biti „varno“, kar pomeni, da je moral Marjan v svojem geslu uporabiti tudi en znak, ki ni črka ali števka in pa vsaj eno veliko črko. Spomni se le še tega, da je nekje znotraj gesla (mogoče celo čisto na začetku ali na koncu) dodal piko in spremenil eno od obstoječih črk v veliko, ne spomni pa se natančno, kje je dodal piko in katero črko je spremenil v veliko. **Napiši podprogram** (funkcijo) MoznaGesla(geslo), ki kot parameter dobi niz geslo z Marjanovim prvotnim geslom iz samih malih črk in števk ter izpiše vse možne nize, ki bi lahko bili Marjanovo geslo za Apple ID. (Na primer: iz `eipe9thu` lahko dobimo `eip.E9thu` ali `e.ipe9tHu` ali `.eiPe9thu` ali še marsikaj drugega.)

2. Marsovci

Vsak marsovec se specializira za natanko 5 opravil. Če je za izvedbo naloge treba več kot 5 opravil, se povežejo v skupine. Imamo skupino m marsovcev in za vsakega marsovca imamo podatke o tem, katerih 5 opravil zna opravljati. Opravila so predstavljena s celimi števili od 1 do 100. **Napiši program**, ki za podano skupino marsovcev ugotovi, ali so vsa tista opravila, ki jih opravlja vsaj en marsovec v skupini, približno enako zastopana; natančneje povedano, preveriti moraš, ali se število marsovcev, ki so specializirani za posamezno opravilo, od enega opravila do drugega razlikuje največ za 1. Podatke naj tvoj program prebere s standardnega vhoda ali pa iz datoteke `marsovci.txt` (karkoli ti je lažje); v prvi vrstici je število marsovcev m, v vsaki od naslednjih m vrstic pa je po 5 števil, ki povedo, katera opravila obvlada posamezni marsovec. Če so vsa opravila približno enako zastopana, naj izpiše `da`, sicer pa `ne`.

Primer vhodnih podatkov:

```
4
75 12 96 57 28
96 28 12 75 9
96 9 57 28 75
12 57 9 28 75
```

Pripadajoči izhod:

```
da
```

Še en primer vhoda:

```
4
75 12 96 57 28
96 28 12 75 9
96 9 57 28 75
12 57 96 28 75
```

Pripadajoči izhod:

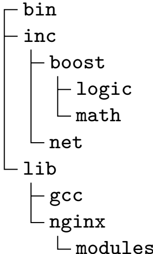
```
ne
```

Komentar: v prvem primeru se vsako opravilo pojavlja pri treh ali štirih marsovcih, zato so približno enakomerno zastopana. V drugem primeru pa se opravilo 9 pojavlja

le pri dveh marsovcih, nekatera opravila pa pri štirih, zato niso približno enakomerno zastopana (glede na definicijo iz besedila naloge).

3. Rekonstrukcija poti

Direktorije oziroma mape na disku si pogosto predstavljamo kot zložene v drevesasto hierarhično strukturo, na primer takole:



Če bi hoteli takšno drevo direktorijev predstaviti samo z besedilom, brez črt, nam lahko prideta na misel naslednja dva načina:

(1) Pri vsakem imenu direktorija lahko zapišemo njegovo globino v drevesu. Pri zgornjem drevesu bi tako dobili:

```
bin 1
inc 1
boost 2
logic 3
math 3
net 2
lib 1
gcc 2
nginx 2
modules 3
```

(2) Lahko pa za vsak direktorij izpišemo polno pot od korena do njega. Pri zgornjem drevesu bi tako dobili:

```
/bin
/inc
/inc/boost
/inc/boost/logic
/inc/boost/math
/inc/net
/lib
/lib/gcc
/lib/nginx
/lib/nginx/modules
```

Napiši program, ki prebere predstavitev drevesa v prvi obliki (torej z imeni direktorijev in njihovimi globinami v drevesu) in ga izpiše v drugi obliki (torej s polnimi potmi). Delovati mora seveda za poljuben vhod, ne le za tistega iz gornjega primera. Če v vhodnem seznamu manjka kakšen direktorij in zato v nekem trenutku poti ni več mogoče rekonstruirati, naj program izpiše „`Napaka!`“ in se neha izvajati. Podatke lahko bereš s standardnega vhoda in pišeš na standardni izhod ali pa bereš iz datoteke `vhod.txt` in pišeš na `izhod.txt` (karkoli ti je lažje). Imena direktorijev so sestavljena le iz črk, brez presledkov ali kakšnih drugih posebnih znakov.

Primer vhoda, kjer rekonstrukcija ni mogoča:

```
abc 1
def 3
```

Tvoj program bi moral tu izpisati:

```
/abc
Napaka!
```

4. Kako dobri so virusni testi?

Prebivalce testiramo na okužbo z virusom covida s hitrimi testi in s testi PCR. Prvi so, kot že ime pove, hitri (in poceni) in dajo rezultat v nekaj minutah, so pa nezanesljivi, drugi, tako imenovani testi PCR, pa so zanesljivejši, vendar precej dražji in je na rezultate treba čakati en dan.

Da bi ugotovili kvaliteto hitrih testov, občasno testiramo skupino ljudi hkrati z obema vrstama testov, hitrimi in testi PCR. Rezultate lahko predstavimo z dvema enako dolgima nizoma znakov, pri čemer i-ti znak prvega niza pove rezultat hitrega testa na i-tem pacientu (`1` = okužen in `0` = neokužen), i-ti znak drugega niza pa rezultat testa PCR na istem pacientu (enako `1` = okužen in `0` = neokužen). Primerjava obeh nizov nam pokaže kvaliteto hitrih testov, ki smo jih pri tem poskusu uporabili.

Napiši podprogram (funkcijo) `Primerjava(s, t, n)`, ki kot parametra dobi dva enako dolga niza `s` (rezultati hitrih testov) in `t` (rezultati testov PCR) in ugotovi, pri katerih n zaporednih pacientih je bilo največ razhajanj med hitrimi in testi PCR. Tvoja funkcija naj vrne indeks, na katerem se začne ta skupina n zaporednih pacientov; če je takšnih skupin več, vrni indeks najbolj leve od njih (tiste z najmanjšim začetnim indeksom). Tvoja rešitev naj bo čim bolj učinkovita, da bo delovala hitro tudi za zelo dolge nize in velike n. Predpostavi, da sta `s` in `t` dolga po vsaj n znakov, tako da rešitev gotovo obstaja.

5. Zlaganje loncev

V kuhinjsko omaro zlagamo lonce. Lonci so v obliki odprtih valjev različnih premerov. Zaradi prihranka prostora lahko natanko en manjši lonec položimo v večjega, kadar ima manjši premer osnovne ploskve kot večji lonec. V ta manjši lonec pa lahko kasneje položimo še en manjši lonec in tako naprej, da dobimo nekakšen sklad loncev. Ne želimo pa v en lonec neposredno postaviti dveh ali več manjših (npr. da bi v lonec premera 20 cm postavili neposredno lonca premerov 5 cm in 3 cm; v tem primeru bi v lonec premera 20 cm postavili lonec premera 5 cm, v slednjega pa potem lonec premera 3 cm). Preveriti želimo, ali je naša kuhinjska omara dovolj prostorna, da lahko vanjo na ta način postavimo vse svoje lonce.

Opiši postopek (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki kot vhodni podatek dobi seznam premerov vseh loncev na kuhinjski mizi ter izračuna najmanjše število skladov, ki jih lahko sestavimo iz teh loncev. Izračuna pa naj tudi najnižjo možno vsoto, ki jo lahko dobimo, če vzamemo premer najbolj spodnjega lonca v vsakem skladu in te premere seštejemo po vseh skladih. Dobro tudi utemelji pravilnost svojega postopka.

Primer: če imamo lonce s premeri

$$28, 17, 14, 29, 12, 22, 28, 28, 13, 20, 30, 18, 4, 18, 4,$$

potrebujemo najmanj tri sklade, najmanjša možna vsota premerov pa je 86. (Eden od možnih načinov, kako lahko zložimo lonce na optimalen način, so takšni trije skladi: $[4, 14, 28, 29, 30]$, $[13, 17, 18, 28]$ in $[4, 12, 18, 20, 22, 28]$; vsota premerov najbolj spodnjih loncev je takrat $30 + 28 + 28 = 86$.)

NALOGE ZA DRUGO SKUPINO

[Pred nalogami so bila navodila, enaka tistim v prvi skupini (str. 10), zato jih tu ne bomo ponavljali.—*Op. ur.*]

1. Sredinec

V šoli je pri športni vzgoji navada, da se pred začetkom šolske ure učenci postavijo v vrsto od najmanjšega do največjega. Prav tako je navada, da učenci zamujajo. Vsak učenec, ki vstopi v telovadnico, se vrine na svoje mesto v vrsti glede na velikost. Učitelj športne vzgoje se med tem zamudnim procesom zabava z opazovanjem, kdo se po vsakem novem prihodu nahaja na sredini vrste. Učenci vstopajo posamično, učitelja pa zanima, kako visok je tisti izmed n prisotnih učencev, ki se trenutno nahaja na $\lceil n/2 \rceil$-tem mestu v vrsti od najmanjšega do največjega. (Zapis $\lceil n/2 \rceil$ pomeni, da rezultat po deljenju n z 2 zaokrožimo navzgor. Na primer: pri $n = 5$ in $n = 6$ ga zanima tretji po vrsti, pri $n = 7$ in $n = 8$ četrti po vrsti in podobno.)

Opiši postopek, ki to nalogo reši čim bolj učinkovito (recimo, da učencev ni le nekaj deset, ampak na milijone): prebira naj višine učencev v takem vrstnem redu, kakor vstopajo v telovadnico, in po vsakem prebranem učencu sproti izpiše višino tistega, ki je zdaj srednji po višini. Oceni tudi časovno zahtevnost svoje rešitve, torej kako se povečuje čas izvajanja v odvisnosti od števila učencev. Višine učencev so podane v obliki seznama po vrsti, tako kot vstopajo v telovadnico. Višine niso večje od dveh metrov in so podane s celimi števili, ki predstavljajo višino v centimetrih.[1]

2. Svetilka

Žepna baterijska svetilka je lahko ugasnjena ali pa sveti v dveh možnih načinih: sveti stalno ali pa utripa tako, da vsako sekundo posveti za eno desetinko sekunde (in je potem devet desetink sekunde ugasnjena). Za preklop med temi tremi stanji služi tipka.

Takoj ko pritisnemo tipko (t.j. ob začetku pritisnjenosti tipke) naj se svetilka vklopi: če je bila prej ugasnjena, naj se vklopi v stalni način, če je bila v stalnem načinu, naj se preklopi v utripanje, in če je utripala, naj se preklopi v stalni način. Če je tipka pritisnjena tri sekunde ali več, naj se po teh treh sekundah svetilka izklopi.

Podana je funkcija `Luc(vklop)`, s katero lahko program upravlja svetilo: vrednost **true** vklopi svetilo, **false** ga izklopi.

Napiši naslednji dve **funkciji**, ki ju bo operacijski sistem malega računalnika v svetilki avtomatsko klical takole:

- `Tiktak()` — ta funkcija bo poklicana vsako desetinko sekunde;
- `Tipka(pritisnjena)` — ta funkcija bo poklicana vsakokrat, ko se bo stanje pritisnjenosti tipke spremenilo; vrednost argumenta bo **true**, če je bila tipka pravkar pritisnjena (t.j. začetek pritiska), in **false**, če je bila tipka pravkar spuščena.

[1] Zanimivo in rahlo težjo različico naloge dobimo, če dovolimo tudi ne-celoštevilske višine (ali pa npr. rečemo, da so višine sicer cela števila, vendar v nanometrih, tako da so ta števila precej velika).

Za ohranitev stanja programa lahko uporabiš poljubne globalne spremenljivke in jih tudi po svoje inicializiraš. Na začetku delovanja programa je luč ugasnjena, tipka pa spuščena.

Glede na to, da nimamo možnosti merjenja časa z večjo ločljivostjo od desetinke sekunde, ne bo nič narobe, če ob preklopu na utripanje prvi blisk ne traja točno eno desetinko sekunde, prav tako lahko trosekundni interval (za ugašanje svetilke) odstopa za malenkost.

Če si želiš poenostaviti nalogo, lahko opustiš stanje utripanja in poskrbiš le za vklop in izklop svetilke — pri tem boš dobil največ polovico točk naloge.

3. Pletenje puloverja

Neža se je med karanteno lotila novega konjička. Naučila se je plesti. Nekaj preglavic pa ji povzročajo sheme za pletenje vzorcev. V knjigah so pogosto narisane velike sheme, na primer:

```
---0-0---0-0---0-0---0-0---0-0
---000---000---000---000---000
---0-0---0-0---0-0---0-0---0-0
---000---000---000---000---000
---0-0---0-0---0-0---0-0---0-0
---000---000---000---000---000
---0-0---0-0---0-0---0-0---0-0
---000---000---000---000---000
```

Zgornja shema predstavlja osnovni vzorec

```
---0-0
---000
```

Neža je hitro ugotovila, da si mora pri pletenju izdelka zapomniti oziroma zapisati le osnovni vzorec in ne celotne velike sheme iz knjige. Prosi te, da **napišeš program** ali podprogram (funkcijo), ki v poljubni dani shemi poišče osnovni vzorec. Osnovni vzorec je najmanjši (po površini) tak vzorec, iz katerega lahko s ponavljanjem sestavimo celotno shemo (pri čemer se mora vzorec lepo zaključiti na vseh robovih sheme). Program naj izpiše širino in višino osnovnega vzorca. Za primer zgoraj je rešitev `6 2`. Če je možnih več enako dobrih rešitev, je vseeno, katero od njih izpišeš. Shemo lahko tvoj program prebere iz datoteke ali s standardnega vhoda ali pa predpostavi, da je že podana v neki tabeli ali seznamu nizov (ali dvodimenzionalni tabeli znakov). Shemo sestavljajo le znaki „`-`“ in „`0`“.

4. Pangramski podniz

Pangram je niz, ki vsebuje vsako črko abecede vsaj enkrat; pri tej nalogi pa nas bodo zanimali malo bolj posebni pangrami — taki, ki vsebujejo vsako črko abecede vsaj k-krat. **Napiši podprogram** oz. funkcijo, ki za dani niz s in naravno število k vrne dolžino najkrajšega takega strnjenega podniza niza s, v katerem se vsaka črka abecede pojavi vsaj k-krat. Če takega podniza sploh ni, naj funkcija vrne -1. Niz s je sestavljen le iz malih črk angleške abecede, lahko pa je zelo dolg, zato naj bo tvoja rešitev čim bolj učinkovita.

Primer: če bi namesto cele abecede gledali le črke {a, b, c} in če bi imeli $k = 2$, bi bil najkrajši primerni podniz v nizu s = aabaaccabaacccbabb dolg 7 znakov. Taki podnizi so celo trije: baaccab, baacccb, acccbab. Poudarimo pa, da je to samo primer in da mora tvoja rešitev delovati za celotno abecedo in za poljuben k in poljubno dolg niz s.

5. Tetris

Imamo ploščo, sestavljeno iz 8×8 kvadratnih polj, ki bi jo radi pokrili s ploščki v obliki raznih likov, podobnih tistim iz igre Tetris. **Napiši program** ali podprogram (funkcijo), ki bo ploščo v celoti pokril s ploščki, pri čemer se le-ti med seboj ne smejo prekrivati ali štrleti čez rob plošče. Na voljo so ploščki n različnih oblik, ki so oštevilčene od 1 do n; v vsaki obliki pa je na voljo le omejeno število ploščkov. Za delo s ploščki naj tvoj program uporablja naslednje funkcije (zanje torej predpostavi, da že obstajajo in ni mišljeno, da jih ti implementiraš sam):

- **int** StOblik() — vrne n, torej število, ki pove, koliko različnih oblik ploščkov je na voljo.

- **int** StPlosckov(**int** oblika) — vrne število razpoložljivih ploščkov oblike oblika. To je celo število, večje od 0, vanj pa so všteti tudi tisti ploščki, ki jih je tvoj program mogoče že postavil na ploščo.

- **bool** JePokrito(**int** x, **int** y) — vrne logično vrednost, ki pove, ali je polje (x, y) na plošči trenutno pokrito (torej ali ga pokriva kakšen od že doslej postavljenih ploščkov). Na začetku izvajanja tvojega programa je plošča prazna (torej ni na njej še nobenega ploščka). Koordinate polj na plošči gredo od $x = 0$ (levo) do $x = 7$ (desno) in od $y = 0$ (zgoraj) do $y = 7$ (spodaj).

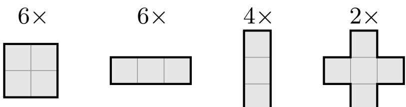
- **bool** PreveriPloscek(**int** oblika, **int** x, **int** y) — vrne logično vrednost, ki pove, ali je mogoče na ploščo dodati plošček oblike oblika tako, da najbolj levo polje v najbolj zgornji vrstici tega ploščka pokrije polje (x, y) na plošči. (Funkcija preverja le obliko, ne pa tudi tega, ali imaš še na voljo kaj ploščkov te oblike ali pa si jih morda že vse postavil na ploščo.)

- **void** PostaviPloscek(**int** oblika, **int** x, **int** y, **bool** b) — če je b == **true**, ta funkcija položi plošček oblike oblika na ploščo tako, da najbolj levo polje v najbolj zgornji vrstici tega ploščka pokrije polje (x, y) na plošči. Če je b == **false**, pa funkcija ta plošček s tega položaja odstrani.

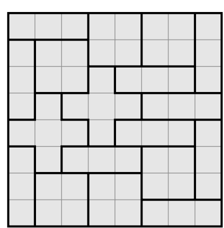
Funkcija PostaviPloscek prekine izvajanje tvojega programa, če zahtevaš od nje operacijo, ki je ni mogoče izvesti (npr. dodajanje ploščka neke oblike, če si vse razpoložljive ploščke te oblike že položil na mrežo; ali dodajanje ploščka tako, da bi se prekrival z že obstoječimi ali štrlel čez rob mreže; ali brisanje ploščka, ki ga v resnici ni tam).

Ploščkov se pri tej nalogi ne dá obračati ali vrteti in funkciji PreveriPloscek in PostaviPloscek tega tudi ne poskušata početi. To pomeni, da štejeta na primer in za dve različni obliki in ploščkov ene oblike ne moremo zasukati in uporabiti kot ploščke druge oblike.

Primer. Recimo, da imamo ploščke naslednjih štirih oblik v naslednjih količinah:



Potem lahko ploščo pokrijemo takole:



Še deklaracije gornjih funkcij v drugih jezikih:

```
{ V pascalu: }
function StOblik: integer;
function StPlosckov(oblika: integer): integer;
function JePokrito(x, y: integer): boolean;
function PreveriPloscek(oblika, x, y: integer): boolean;
procedure PostaviPloscek(oblika, x, y: integer; b: boolean);
```

```
// V javi: deklaracije so kot v besedilu naloge, le z boolean namesto bool.
```

```
# V pythonu:
def StOblik() -> int: ...
def StPlosckov(oblika: int) -> int: ...
def JePokrito(x: int, y: int) -> bool: ...
def PreveriPloscek(oblika: int, x: int, y: int) -> bool: ...
def PostaviPloscek(oblika: int, x: int, y: int, b: bool) -> None: ...
```

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Naloge rešuj samostojno; ne sprašuj drugih ljudi za nasvete ali pomoč pri reševanju (niti v živo niti prek interneta ali kako drugače), ne kopiraj v svoje odgovore tuje izvorne kode in podobno. Tekmovalna komisija si pridržuje pravico, da tekmovalce diskvalificira, če bi se kasneje izkazalo, da nalog niso reševali sami. Internet lahko uporabljaš, če ni v nasprotju s prejšnjimi omejitvami (npr. za branje dokumentacije). V rešitvah lahko uporabljaš manjše fragmente izvorne kode, ki si jih napisal sam že pred tekmovanjem.

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++` 7.4.0 (ta verzija podpira C++17), prevajalnikom za javo iz JDK 8, s prevajalnikom Mono 4.6 za C# in z interpreterjema za python 2.7 in 3.6.

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2021-3/` 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 boš dobil po elektronski pošti. Med tekmovanjem lahko vprašanja za tekmovalno komisijo postavljaš prek foruma na Putki (povezava „Diskusija“ na dnu besedila posamezne naloge), izjemoma pa tudi po elektronski pošti na `rtk-info@ijs.si`.

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/help/programming/`). 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. Pri tretji in četrti nalogi je testnih primerov po 20 in vsak je vreden po 5 točk, pri ostalih 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++:

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

int main()
{
  int i, j; cin >> i >> j;
  cout << 10 * (i + j) << '\n';
}
```

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

- 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 pythonu:

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

- 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. Kapniki

Jamarji so odkrili dolgo nizko jamo s številnimi kapniki. Tla in strop jame sta vzporedna. Jama je višine v in dolžine n, na vsakem metru jame pa raste s tal stalagmit ali s stropa stalaktit. Za vsak kapnik poznamo njegov tip (stalagmit ali stalaktit) in velikost kapnika k_i ($1 \leq k_i < v$). V jami bi radi postavili turistično železnico, ki bo potekala vzporedno s tlemi in stropom na neki celoštevilski višini y ($1 \leq y \leq v$). Ker so stalagmiti velikosti $k_i \geq y$ in stalaktiti velikosti $k_i > v - y$ taki železnici v napoto, jih bo treba podreti. **Napiši program**, ki bo poiskal višine železnice y, pri katerih bi bilo treba podreti čim manj kapnikov.

Vhodni podatki: v prvi vrsti sta celi števili v in n, ločeni s presledkom. V drugi vrstici je podan niz n znakov `'M'` ali `'T'`, kjer i-ti znak v nizu predstavlja tip kapnika, ki raste na i-tem metru jame. Če je enak `'M'`, gre za stalagmit, ki raste s tal, če je enak `'T'`, pa za stalaktit, ki raste s stropa. V tretji vrstici je podan s presledki ločen seznam n celih števil k_i, kjer i-to število predstavlja velikost i-tega kapnika.

Omejitve: veljalo bo $1 \leq v \leq 10^{18}$ in $1 \leq n \leq 10^5$.

- V prvih 20 % testnih primerov bo $n \leq 1000$ in $v \leq 1000$.
- V naslednjih 40 % testnih primerov bo $v \leq 10^6$.

Izhodni podatki: izpiši najmanjše število kapnikov, ki jih bo treba podreti, in število višin železnice, pri katerih lahko dosežemo to število podrtih kapnikov. Števili izpiši v isti vrstici, ločeni pa naj bosta z enim presledkom.

Primer vhoda:

```
8 9
TTMTMMTTM
2 1 6 5 2 5 3 7 2
```

Pripadajoči izhod:

```
3 1
```

2. Socialno omrežje

V neki demokratični deželi daleč daleč stran se je blaženi vodja odločil prepovedati socialna omrežja tehnoloških gigantov, kot sta npr. Twitter in Facebook, zaradi

morebitnega širjenja neprimernih vsebin. Namesto tega pa so zgradili svoje lastno socialno omrežje, kjer lahko uporabniki med seboj sklepajo prijateljstva in sovraštva. Vemo, da uporabniki omenjenega omrežja prijateljstva in sovraštva sklepajo po naslednjih pravilih: „prijatelj mojega prijatelja je moj prijatelj“, „sovražnik mojega prijatelja je moj sovražnik“ in „sovražnik mojega sovražnika je moj prijatelj“. Omenjena pravila so bolj formalno definirana kasneje. Dobili smo dostop do podatkov o prijateljih in sovražnikih na tem omrežju, zanima pa nas, ali so omenjena pravila dosledno spoštovana, torej ali niti nikoli niso kršena niti z njihovim upoštevanjem ne moremo skleniti novih prijateljstev ali sovraštev.

Vhodni podatki: v prvi vrstici je podano celo število t, to je število omrežij v tem testnem primeru. Sledijo opisi vseh t omrežij. Vsako omrežje se začne z vrstico, ki vsebuje celi števili n (število ljudi v omrežju) in m (število prijateljstev ali sovraštev med njimi). Uporabniki so označeni s števili od 1 do n. Sledi m vrstic; vsaka izmed njih vsebuje tri števila a_i, b_i in p_i. To nam pove, da sta osebi a_i in b_i povezani med seboj. Če je p_i enak 0, sta a_i in b_i sovražnika, če je p_i enak 1, pa prijatelja. Čustva so obojestranska in vsako je navedeno samo enkrat, prav tako ni mogoče, da bi bili dve osebi hkrati prijatelja in sovražnika.

Za vsako podano omrežje želimo preveriti, ali dosledno spoštuje naslednja tri pravila:

1. Prijatelj mojega prijatelja je moj prijatelj: če sta osebi A in B prijatelja in osebi B in C prijatelja, potem morata biti tudi osebi A in C prijatelja.
2. Sovražnik mojega prijatelja je moj sovražnik: če sta osebi A in B prijatelja in osebi B in C sovražnika, potem morata biti osebi A in C sovražnika.
3. Sovražnik mojega sovražnika je moj prijatelj: če sta osebi A in B sovražnika in osebi B in C sovražnika, potem morata biti osebi A in C prijatelja.

Izhodni podatki: za vsako izmed t omrežij izpiši „`DA`“, če omenjena pravila dosledno veljajo, sicer pa „`NE`“. Vsak odgovor naj bo podan v svoji vrstici.

Omejitve vhodnih podatkov: vedno bo veljalo $1 \leq n \leq 10^5$, $1 \leq m \leq 10^5$ in $t \leq 10$.

Podnaloge:

- V prvih 20 % testnih primerov velja $n \leq 100$ in $m \leq 100$.
- V naslednjih 40 % testnih primerov velja $n \leq 1000$ in $m \leq 1000$.
- Pri preostalih 40 % testnih primerov ni dodatnih omejitev.

Primer vhoda:

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

Pripadajoči izhod:

```
DA
NE
```

3. Proizvodnja cepiva

Za farmacevtsko podjetje, ki proizvaja cepiva proti COVID-19, moramo sestaviti načrt proizvodnje cepiva. Trenutno proizvajamo 0 odmerkov cepiva na dan. Vsak dan znova se odločimo, ali bomo proizvajali cepivo ali pa nadgrajevali proizvodnjo.

Če se odločimo za nadgradnjo proizvodnje, ta dan ne proizvedemo nič cepiva, dnevno proizvodnjo odmerkov cepiva pa povečamo za 1. Če se odločimo za proizvodnjo, pa ta dan proizvedemo toliko odmerkov, kolikor je naša trenutna dnevna proizvodnja cepiva.

Naš cilj je v čim krajšem času ustvariti k odmerkov cepiva. A to še ni vse! Zaradi cepljenja najbolj ranljivih skupin moramo nekaj cepiva dostaviti že vnaprej. Natančneje rečeno, podanih imamo d omejitev, vsaka od njih pa pravi, da moramo v roku x_i dni skupno ustvariti vsaj y_i odmerkov cepiva, kjer so dnevi oštevilčeni začenši z 1.

Napiši program, ki izračuna, koliko najmanj dni potrebujemo, da ustvarimo k odmerkov cepiva, če upoštevamo vse omejitve in optimalno izbiramo strategijo za nadgradnjo in proizvodnjo cepiva.

Vhodni podatki: v prvi vrstici bosta podani dve števili, k (število odmerkov cepiva, ki jih moramo proizvesti) in d (število omejitev, ki jih moramo pri tem upoštevati). Nato sledi d vrstic, ki opisujejo omejitve, v vsaki izmed njih pa sta dani števili x_i in y_i. V prvih x_i dneh je treba skupno proizvesti vsaj y_i odmerkov cepiva.

Izhodni podatki: tvoj program naj izpiše eno število, namreč minimalno število dni, ki jih potrebujemo, da ustvarimo dovolj odmerkov cepiva. Zagotovljeno je, da bo rešitev vedno obstajala.

Omejitve podatkov: $0 \leq d \leq 100$, $1 \leq k \leq 10^6$, $1 \leq y_i \leq k$, $1 \leq x_i \leq 10^6$. Pri prvih 20 % testnih primerov bo $d = 0$; pri naslednjih 30 % testnih primerov bo $k \leq 10^4$.

Primer vhoda:

```
100 1
4 4
```

Pripadajoči izhod:

```
22
```

4. Virus v Timaniji

V deželi Timaniji so slišali, da po svetu razsaja nov smrtonosni virus. Oseba, ki je okužena, postane tudi sama kužna po natanko k dneh, v natanko ℓ dneh po nastopu kužnosti pa oseba v strašnih krčih in mukah umre. (Primer: če je $k = 2$ in $\ell = 3$ in se je nekdo okužil v ponedeljek, bo sam okuževal druge v sredo, četrtek in petek, umrl pa bo v soboto in tisti dan ne bo okužil nikogar.) Drugih simptomov pred smrtjo ni, tako da živih okuženih državljanov ni mogoče poslati v osamo.

V strahu pred izbruhom epidemije je vlada Timanije sprejela ukrepe, kjer je omejila srečanja med državljani, tako da se vsak državljan lahko na vsak dan v tednu sreča le z enim preostalim državljanom, skupno z največ 7 državljani v tednu. Vsak državljan je moral na seznam napisati, koga bo srečal kateri dan v tednu, seznama pa kasneje ne smejo spreminjati in se ga morajo vsi državljani strogo držati (seznam je torej vsak teden enak).

Napiši program, ki bo preveril, ali bo po vnosu virusa celotno prebivalstvo izumrlo ali ne. Vlado zanima, ali za dan urnik srečanj za celotno prebivalstvo in določena k in ℓ obstaja scenarij, kjer se bo na točno določen dan d okužil (od zunaj) točno določen državljan in se bo tako sčasoma okužilo (in pomrlo) celotno prebivalstvo (število prebivalcev je n). Če tak scenarij ne obstaja, pa poišči scenarij z najmanjšim številom preživelih državljanov po epidemiji. Dni v tednu je 7, kjer nedeljo predstavlja število 0, ponedeljek število 1, soboto pa število 6. Predpostaviš lahko, da se vsa srečanja zgodijo ob istem času zjutraj in da na dan smrti okuženi ne okuži nikogar več.

Vhodni podatki: v prvi vrstici so števila n, k in ℓ, ločena s po enim presledkom. Sledi n vrstic s po sedmimi številkami, vsaka od njih pa predstavlja urnik srečanj po enega državljana: j-ta številka v i-ti vrstici predstavlja številko državljana, ki ga bo i-ti državljan srečal v j-tem dnevu vsakega tedna. (Državljani so oštevilčeni s številkami od 0 do $n - 1$.)

Veljalo bo $1 \leq k \leq 15$, $1 \leq \ell \leq 15$. Veljalo bo še:

- v prvih 20 % primerov: $\ell = 1$ in $n \leq 1000$;
- v naslednjih 40 % primerov: $\ell = 1$, $n \leq 10^5$;
- v preostalih 40 % primerov: $\ell \leq 15$, $n \leq 1000$.

Izhodni podatki: izpiši štiri cela števila i, d, p in r, ločena s po enim presledkom. Pri tem naj bo i številka državljana in d številka dneva v tednu za tisti scenarij, pri katerem umre največ ljudi, če se okužba začne s tem, da se državljan i okuži na dan d; število p naj bo za ta scenarij število preživelih po epidemiji; število r pa naj pove, koliko scenarijev s tem številom preživelih obstaja, torej za koliko parov (i', d') velja, da na koncu ostane p preživelih, če se epidemija začne s tem, da se človek i' okuži na dan d'. Če obstaja več enako dobrih rešitev, je vseeno, katero od njih izpišeš.

Primer vhoda:

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

Pripadajoči izhod:

```
1 5 0 24
```

Komentar. Scenarij, na katerega se sklicuje izhod v gornjem primeru, je naslednji:

- V petek (dan 5) prvega tedna se okuži državljan 1.
- Drugi teden, dan 1: državljan 1 okuži državljana 3. (Naslednji dan se spet srečata, ampak je 3 že okužen.)
- Drugi teden, dan 4: državljan 3 okuži državljana 0.
- Drugi teden, dan 5: državljan 3 okuži državljana 2.
- Tretji teden, dan 1: državljan 0 okuži državljana 6, državljan 2 pa okuži državljana 4.
- Tretji teden, dan 4: državljan 6 okuži državljana 7.

- Četrti teden, dan 1: državljan 7 okuži državljana 5.

Tako so se okužili vsi državljani in preživelih ni. To je eden od 24 možnih scenarijev, pri katerih za te vhodne podatke umrejo vsi ljudje.

Še en primer vhoda:

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

Pripadajoči izhod:

```
3 4 0 16
```

5. Tja in spet nazaj

Hobit se odpravlja na dogodivščino. V roki ima zemljevid, na katerem je označil n točk, ki jih želi obiskati vsaj enkrat. Trenutno se nahaja doma na najbolj zahodni točki (tisti z najmanjšo x-koordinato), kamor se želi na koncu tudi vrniti. Odločil se je, da ga bo njegova pot najprej vodila ves čas proti vzhodu v smeri naraščajočih x-koordinat točk, nato pa se bo obrnil in se ves čas premikal nazaj proti zahodu v smeri padajočih x-koordinat točk. **Napiši program**, ki bo izračunal dolžino najkrajše hobitove poti, na kateri obišče vse točke vsaj enkrat in se vrne domov.

Vhodni podatki: v prvi vrstici je število točk n, ki so podane v sledečih n vrsticah. V vsaki vrstici sta podani s presledkom ločeni koordinati x_i in y_i neke točke na zemljevidu. Vse koordinate x_i bodo med seboj različne.

Omejitve: veljalo bo $2 \leq n \leq 5000$. Koordinate točk x_i in y_i bodo celoštevilske z intervala $[0, 100\,000]$.

- V prvih 30 % testnih primerov bo $n \leq 20$.
- V naslednjih 40 % testnih primerov bo $n \leq 500$.

Izhodni podatki: izpiši dolžino najkrajše hobitove poti. Rešitev bo sprejeta, če se bo od uradne razlikovala za največ 10^{-4}.

Primer vhoda:

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

Eden od možnih pripadajočih izhodov:

```
20.013352
```

NALOGE ZA ŠOLSKO TEKMOVANJE

22. januarja 2021

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. Križci in krožci

Dva igralca sta se igrala križce in krožce na karirasti mreži nenavadne oblike: sestavlja jo ena sama vrstica, v njej pa je n polj. Stanje mreže lahko zato opišemo z nizom n znakov, v katerem črka `'x'` predstavlja križec, črka `'o'` pa krožec. **Napiši podprogram** (funkcijo) `Izenaceno(s)`, ki kot vhodni podatek dobi niz `s` in preveri, če ta niz predstavlja takšno stanje mreže, v katerem se je igra končala z izenačenim izidom. Z drugimi besedami, preveriti je treba, če veljajo vsi naslednji pogoji:

- Na vsakem od polj mora biti ali križec ali krožec, drugih znakov v nizu ne sme biti.
- Število križcev mora biti enako številu krožcev.
- Nikjer se ne smejo pojavljati po trije (ali več) enaki znaki skupaj.

Tvoja rešitev naj bo učinkovita, tako da bo delovala tudi za velike n (dolge vhodne nize).

Nekaj primerov: niza `xxooxo` in `oxox` predstavljata izenačene izide, nizi `xxooox`, `xxcoox` in `ooxooxxo` pa ne.

2. Kovanci

Janezek rad obiskuje dedka. Ne le, da dedek ve toliko zanimivih zgodb, vsakič, ko ga obišče, se njegov hranilnik-prašiček odebeli. Zadnjič pa ga je čakalo presenečenje. Dedek mu je na mizo postavil več kupčkov kovancev in mu naročil: vzameš lahko, kolikor želiš kupčkov, samo nikoli ne smeš vzeti dveh sosednjih. Če so torej na mizi kupčki z vrednostmi

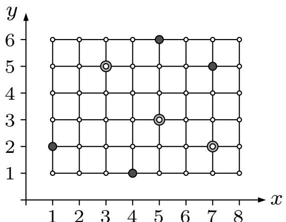
$$2,\ 4,\ 1,\ 3,\ 4,$$

lahko Janezek izbere prvi, tretji in peti kupček, lahko vzame drugega in petega, lahko drugega in četrtega itd. — možnih je še nekaj drugih kombinacij. Janezkov prašiček je seveda lačen, zato bi rad Janezek pobral z mize karseda veliko kovancev. Pomagaj mu in **opiši postopek** (ali napiši podprogram oz. funkcijo, če ti je lažje), ki za dano tabelo vrednosti kupčkov vrne največjo vsoto, ki jo je mogoče na ta način doseči. Pri zgornjem primeru je pravilni rezultat 8 (to vsoto dobimo, če poberemo drugi in peti kupček).

Pozor: čeprav je v zgornjem primeru pet kupčkov, naj tvoja rešitev deluje za poljubno število kupčkov, tudi če jih je npr. več tisoč.[2]

3. Taksi

V nekem mestu ima cestno omrežje obliko pravokotne kariraste mreže. Položaj vsake točke (križišča) lahko zato opišemo s parom celoštevilskih koordinat (x, y), kot kaže naslednja slika:



Ker potekajo ceste samo vodoravno in navpično, je dolžina poti med dvema točkama, recimo (x_1, y_1) in (x_2, y_2), pri tem cestnem omrežju enaka $|x_1 - x_2| + |y_1 - y_2|$.

Taksist, ki vozi za n stalnih strank, bi rad na eni od točk omrežja postavil svojo centralo, pri čemer se mora odločiti med m možnimi položaji centrale. Med raziskavo profitabilnosti je že določil koordinate (x, y) vseh n strank in vseh m možnih položajev centrale. **Opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki ugotovi, na katerega izmed možnih položajev naj postavi centralo, da bo vsota razdalj od centrale do strank čim manjša. (Če obstaja več enako dobrih najboljših položajev, je vseeno, katerega od njih vrne tvoja rešitev.) Kot vhodne podatke tvoj postopek dobi n, m ter koordinate vseh n stalnih strank in vseh m možnih položajev centrale. Tvoj postopek naj ne predpostavi, da je mreža majhna, kot na primer tista na gornji sliki — deluje naj tudi za velike mreže.

Primer: zgornja slika kaže mrežo, na kateri so štiri stalne stranke (črne pike) in trije možni položaji centrale (dvojni krogi). Med temi tremi položaji je najboljši tisti na $(5, 3)$, pri katerem je vsota razdalj do stalnih strank enaka (če gledamo stranke od leve proti desni) $5 + 3 + 3 + 4 = 15$.

4. Preusmerjanje

Spletni strežnik ima možnost, da ko odjemalec od njega zahteva vsebino z nekega naslova (URLja), te vsebine odjemalcu ne pošlje, pač pa ga obvesti, da se ta vsebina zdaj nahaja na nekem drugem naslovu — temu pravimo z drugimi besedami, da ga je *preusmeril*. Takšne preusmeritve lahko tvorijo verige: z enega naslova nas preusmerijo na drugega, s tega na tretjega in tako naprej; v najslabšem primeru pa se lahko taka veriga preusmeritev celo zacikla (na primer: s tretjega naslova nas preusmerijo nazaj na drugega).

Napiši podprogram oz. funkcijo, ki za dani začetni naslov z sledi verigi preusmeritev in vrne naslov, pri katerem se ta veriga konča, oz. ugotovi, če se veriga

[2] Zanimiva je tudi naslednja težja različica naloge: Janezkov nagajivi brat Štefan bi rad z mize izmaknil en kupček kovancev tako, da bo Janezkov izkupiček potem čim manjši. Opiši postopek, ki ugotovi, kateri kupček naj izmakne in kakšen bo takrat izkupiček.

zacikla. (Mogoče je seveda tudi, da se veriga konča kar pri z-ju samem, če s tega naslova ni nobene preusmeritve.) Da bo naša naloga lažja, bomo naslove namesto z nizi znakov predstavili kar z naravnimi števili od 1 do n. Kot vhodne podatke dobi tvoj podprogram števili n in z ter seznam parov (s_i, t_i), ki povedo, da obstaja z naslova s_i preusmeritev na naslov t_i. Posamezna stran lahko nastopa kot prvi element v največ enem takem paru — z drugimi besedami, ne more se zgoditi, da bi z neke strani s obstajali preusmeritvi na dve ali več drugih strani.

Primer: če imamo $n = 6$ in preusmeritve $(1, 2)$, $(2, 4)$, $(3, 1)$, $(6, 5)$, je pri $z = 1$ pravilni odgovor 4.

5. Odstranjevanje črk

Angleška beseda *splatters* ima zanimivo lastnost. Če iz nje črke brišemo v pravem vrstnem redu, bomo imeli do konca na vsakem koraku pred seboj neko angleško besedo: *splatters* → *splatter* → *platter* → *latter* → *later* → *late* → *ate* → *at* → *a*.

Napiši podprogram oz. funkcijo, ki kot vhodni podatek dobi seznam nizov in v njem poišče najdaljši niz z zgoraj opisano lastnostjo, torej najdaljši tak niz, v katerem bi se dalo enega po enega brisati znake (če na vsakem koraku primerno izberemo, kateri znak pobrišemo) tako dolgo, da bi ostal niz dolžine 1, pri čemer bi po vsakem brisanju imeli niz, ki je tudi prisoten v vhodnem seznamu. Če obstaja več enako dolgih najdaljših nizov, je vseeno, katerega od njih vrne tvoja rešitev.

Predpostavi, da so nizi sestavljeni le iz malih črk angleške abecede (od `a` do `z`) in dolgi vsaj 1 ter kvečjemu 30 znakov; nize dobiš v neki tabeli, vektorju, seznamu ali čem podobnem, torej se ti ni treba ukvarjati z branjem nizov iz datoteke. Nizov v vhodnem seznamu je lahko veliko, zato naj bo tvoja rešitev učinkovita.[3]

[3] Zanimivo različico naloge dobimo, če dovolimo, da se po brisanju črke tudi spremeni vrstni red preostalih črk. Tako lahko na primer iz besede *kramp* v enem koraku naredimo *park*.

NALOGE S CERC 2021

Društvo ACM Slovenija je letos sodelovalo tudi pri organizaciji srednjeevropskega študentskega tekmovanja v računalništvu (Central European Regional Contest — CERC). Ta tekmovanja so bila ponavadi v novembru tekočega leta, zaradi epidemije pa so se v zadnjem času nekoliko zamaknila in CERC 2021 je potekal 23. in 24. aprila 2022. 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. Zaradi epidemije je letošnji CERC potekal prek interneta; ekipe so reševale naloge vsaka na svoji univerzi pod nadzorstvom lokalnih mentorjev. Tekmovanje obsega en tekmovalni dan (letos je bil to 24. april), 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: F — črke; H — radar; A — letalska družba; K — enotirna železnica; B — gradnja na Luni; L — sistematični trgovski potnik; E — ribolov; I — pokrajinski razvoj; D — DJ Darko; G — premice na mreži; J — ponovitve; C — rezanje kaktusov.

A. Letalska družba

(*Omejitev časa: 15 s.* *Omejitev pomnilnika: 512 MB.*)

Neka letalska družba ponuja redne lete med n različnimi letališči. Vsak let neposredno povezuje dve letališči in omogoča potovanje med njima v obe smeri, vmes pa se ne ustavlja na nobenem drugem letališču. Leti so organizirani tako, da za vsako možno kombinacijo začetnega letališča s in ciljnega letališča t obstaja natanko eno táko zaporedje letov, ki potnike pripeljejo od s do t, ne da bi kakšno letališče obiskali več kot enkrat. Številu letov v tem zaporedju pravimo *razdalja* med s in t.

Če bi letalska družba dodala še en let, na primer med letališčema x in y, bi se utegnilo zgoditi, da bi za nekatere pare (s, t) potem obstajalo še neko novo zaporedje letov od s do t, ki bi bilo krajše od dosedanjega. Pri več parih ko se to zgodi, tem obetavnejša se zdi nova povezava med x in y. Pomagaj letalski družbi in **napiši program**, ki oceni več možnih novih povezav (x, y) glede na ta kriterij.

Vhodni podatki. V prvi vrstici sta dve celi števili, n (število letališč) in q (število možnih novih povezav (x, y), ki jih bo treba oceniti).

Naslednjih $n-1$ vrstic opisuje prvotne polete (tiste, ki so obstajali že pred dodajanjem nove povezave); i-ta od teh vrstic vsebuje celi števili u_i in v_i, ki povesta, da obstaja neposreden let med letališčema u_i in v_i.

Preostalih q vrstic opisuje možne dodatne lete, o katerih razmišlja letalska družba; i-ta od teh vrstic vsebuje celi števili x_i in y_i, ki povesta, da bi v i-tem scenariju prvotnim $n-1$ letom dodali še novo neposredno letalsko povezavo med letališčema x_i in y_i.

Omejitve vhodnih podatkov:

- $2 \leq n \leq 10^6$; $1 \leq q \leq 10^5$
- $1 \leq u_i \leq n$; $1 \leq v_i \leq n$; $u_i \neq v_i$ (za $i = 1, 2, \ldots, n-1$)
- $1 \leq x_i \leq n$; $1 \leq y_i \leq n$; $x_i \neq y_i$ (za $i = 1, 2, \ldots, q$)
- če s k_i označimo razdaljo med x_i in y_i v prvotnem omrežju letov (torej pred dodajanjem nove povezave), bo veljalo $\sum_{i=1}^{q} k_i \leq 10^7$.

Izhodni podatki. Izpiši q vrstic; v i-to od njih izpiši število parov (s, t), za katere je $1 \leq s < t \leq n$ in bi se razdalja med s in t zmanjšala, če bi v prvotno omrežje $n-1$ letov dodali še neposredno letalsko povezavo med letališčema x_i in y_i.

Primer vhoda:

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

Pripadajoči izhod:

```
10
4
```

B. Gradnja na Luni

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

Mentorji z ICPCja se nikoli zares ne upokojijo. Ko oznanijo svojo „upokojitev“, začno v resnici delati za neko tajno agencijo (nadaljnjih podrobnosti ne smemo razkriti), ki na temni strani Lune gradi monumentalne zgradbe. Trenutno je v teku en tak projekt.

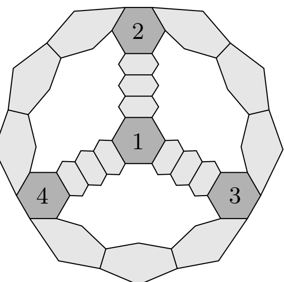
Pri gradnji te monumentalne zgradbe uporabljajo šestkotne gradnike dveh vrst:

- *Dvorana* ima vrata na treh stranicah (od katerih nobeni dve nimata skupnega krajišča).
- *Člen* ima vrata na dveh stranicah, ki ležita na nasprotnih straneh šestkotnika.

Dva člena ali pa člen in dvorano je mogoče spojiti skupaj vzdolž takih stranic, kjer imata oba uporabljena gradnika vrata. Gradnika potem zvarijo skupaj, da zgradba pri spoju ne bo puščala zraka.

Načrtovana zgradba bo imela n dvoran na Luninem površju. Vsaka od teh dvoran bo s *hodniki* povezana z natanko tremi drugimi dvoranami. Vsak hodnik je sestavljen iz L zaporedno spojenih členov. Vsak od obeh koncev hodnika (tam,

kjer so vrata) je spojen z dvorano. Na primer: recimo, da imamo $n = 4$ dvorane (oštevilčene od 1 do 4) in da je $L = 3$. Eno od možnih zgradb pri teh n in L kaže naslednja slika (dvorane so temno sive, hodniki pa svetlo sivi):



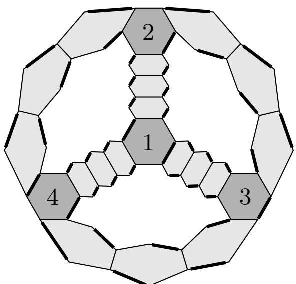
Zagotovljeno je, da vsak par dvoran povezuje največ en hodnik in da se noben hodnik ne začne in konča pri isti dvorani. Poleg tega je iz vsake dvorane mogoče po hodnikih priti do vsake druge dvorane (morda prek ene ali več vmesnih dvoran). Ker je načrt zgradbe pripravil nekdanji mentor z ICPCja, je zagotovljeno, da se hodniki med seboj ne sekajo (spomnimo se, da bodo zgradbo zgradili na površju Lune). Tak načrt lahko opišemo kot zaporedje trojic

$$(c_{11}, c_{12}, c_{13}), (c_{21}, c_{22}, c_{23}), \ldots, (c_{n1}, c_{n2}, c_{n3}).$$

To pomeni, da je dvorana i povezana z dvoranami c_{i1}, c_{i2} in c_{i3}. Če se postavi človek v dvorano i in se zavrti v smeri urinega kazalca, bo najprej videl hodnik, ki vodi v dvorano c_{i1}, nato hodnik, ki vodi v c_{i2}, in končno še hodnik, ki vodi v c_{i3}. Načrt z gornje slike lahko opišemo z naslednjim zaporedjem:

$$(2, 3, 4), (1, 4, 3), (1, 2, 4), (1, 3, 2).$$

Ker je temna stran Lune temna (kot prikladno namiguje že njeno ime), bodo na vsako stranico vsakega gradnika (dvorane ali člena) namestili neonsko luč. Na tistih stranicah, kjer sta dva gradnika zvarjena skupaj, bo seveda le po ena luč (in ne po ena na vsaki strani). Ker bo stala ta zgradba na Luni, ne smemo preveč razsipati z energijo, zato luči na dveh stranicah ne smeta biti prižgani hkrati, če imata tidve stranici skupno krajišče. Da bo stavba dovolj razsvetljena, so se mentorji odločili prižgati *maksimalno število* luči, kolikor jih je le mogoče prižgati ob upoštevanju omejitve zaradi varčevanja z energijo. Takšnemu razporedu bomo rekli *veljavna osvetlitev*. Eno možnost (za naš dosedanji primer zgradbe) kaže naslednja slika:



Mentorjem se zdi, da je mogoče do veljavne osvetlitve priti še na veliko drugih načinov. Zanima jih, koliko je vseh veljavnih osvetlitev. Ker so preleni, da bi sami sprogramirali rešitev, bodo ta problem zastavili kot nalogo na tekmovanju, tako da bodo učinkovite rešitve poiskali študentje. **Napiši program**, ki prebere opis monumentalne zgradbe in izračuna število vseh veljavnih osvetlitev. Ker utegne biti to število zelo veliko, izpiši le njegov ostanek po deljenju z $10^6 + 3$.

Vhodni podatki. V prvi vrstici sta števili n (število dvoran) in L (število členov, ki tvorijo hodnik), ločeni s presledkom. Sledi n vrstic; i-ta od njih vsebuje števila c_{i1}, c_{i2} in c_{i3}, ločena s po enim presledkom.

Omejitve vhodnih podatkov:

- $4 \leq n \leq 16$
- $1 \leq L \leq 100$
- $1 \leq c_{ij} \leq n$ za vse $i = 1, 2, \ldots, n$ in $j = 1, 2, 3$.

Izhodni podatki. Izpiši eno samo število, namreč ostanek po deljenju števila vseh veljavnih osvetlitev z $10^6 + 3$.

Primer vhoda:

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

Pripadajoči izhod:

```
4400
```

C. Rezanje kaktusov

(*Omejitev časa: 15 s. Omejitev pomnilnika: 256 MB.*)

Gospod Malnar je za spremembo opustil svojo obsedenost z drevesi in si našel nekaj še bolj zanimivega: kaktuse! Formalno je kaktus definiran kot povezan neusmerjen graf, v katerem vsaka povezava pripada največ enemu ciklu. Cikel je definiran kot zaporedje dveh ali več različnih povezav, v katerem imata vsaki dve zaporedni povezavi skupno krajišče, pa tudi prva in zadnja povezava imata skupno krajišče; drugače pa nimata nobeni dve povezavi na ciklu skupnega krajišča.

Žal je kaktus, ki ga je gospod Malnar kupil, precej velik, zato ga želi razrezati na ločene palice. Palica je definirana kot par povezav, ki imata skupno eno od krajišč. Gospod Malnar je pedanten človek, zato ga zanima, na koliko načinov lahko razreže svoj kaktus na palice.

Vhodni podatki. V prvi vrstici je število točk n in število povezav m. Vsaka od naslednjih m vrstic vsebuje dve različni celi števili, a_i in b_i, ki povesta, da obstaja med tema dvema točkama povezava. Vsaka povezava se bo v vhodnih podatkih pojavila natanko enkrat.

Omejitve vhodnih podatkov:

- $1 \leq n, m \leq 100\,000$
- $1 \leq a_i, b_i \leq n$ za vse $i = 1, 2, \ldots, m$

Izhodni podatki. Izpiši število različnih načinov, na katere lahko gospod Malnar razreže svoj kaktus na palice. Pravzaprav, ker je to število lahko precej veliko, izpiši le njegov ostanek po deljenju z $10^6 + 3$.

Primer vhoda:	Pripadajoči izhod:

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

D. DJ Darko

(*Omejitev časa: 4 s.* *Omejitev pomnilnika: 256 MB.*)

V mestu je nov didžej. DJ Darko mora pripraviti svoje zvočnike. V vrsti ima n zvočnikov, pri čemer je glasnost i-tega od njih trenutno nastavljena na a_i. Spreminjanje glasnosti je precej težavno, zato je pri i-tem zvočniku treba porabiti b_i enot energije, da se mu glasnost poveča ali zmanjša za 1.

Žal Darka rad zafrkava njegov zlobni brat dvojček Karko. Zgodilo se bo q dogodkov naslednjih dveh vrst:

$$1 \;\; \ell \;\; r \;\; x$$
$$2 \;\; \ell \;\; r$$

Pri dogodku prve vrste spremeni Karko glasnost vseh zvočnikov od vključno ℓ-tega do vključno r-tega za x (glasnosti zvočnikov se torej pri tem prišetje x). Pri dogodku druge vrste nastavi Darko glasnost vseh zvočnikov od vključno ℓ-tega do vključno r-tega na enako vrednost, in sicer na táko, pri kateri porabi za to nastavljanje najmanj energije. Če je mogoče to storiti na več načinov, si izbere med njimi tistega z najnižjo končno glasnostjo. Zvočniki po tej spremembi ostanejo na novi glasnosti, dokler jih kak kasnejši dogodek spet ne spremeni.

Tebe kot gledalca zdaj zanima, na katero glasnost je Darko nastavil zvočnike pri vsakem od dogodkov druge vrste.

Vhodni podatki. V prvi vrstici sta dve števili, število zvočnikov n in število dogodkov q. V drugi vrstici je n števil, $a_1, \ldots, a_n$, ki podajajo trenutne glasnosti zvočnikov. V tretji vrstici je n števil, $b_1, \ldots, b_n$, ki povedo, koliko energije je treba za spremembo glasnosti posameznega zvočnika za 1. V preostalih q vrsticah je q dogodkov, podanih tako, kot je prikazano zgoraj. Vsa števila v vhodnih podatkih so cela.

Omejitve vhodnih podatkov:

- $1 \le n, q \le 200\,000$
- $1 \le a_i, b_i \le 10^9$
- pri vsakem dogodku: $1 \le \ell \le r \le n$ in $-10^9 \le x \le 10^9$.

Izhodni podatki. Za vsak dogodek drugega tipa izpiši glasnost, na katero je Darko pri njem nastavil zvočnike.

Primer vhoda:

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

Pripadajoči izhod:

```
1
0
-7
9
```

Še en primer vhoda:

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

Pripadajoči izhod:

```
-3
-7
```

E. Ribolov

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

Ob obali Jadranskega morja stoji majhna vas. Tamkajšnji ribiči modelirajo morje kot karirasto mrežo $w \times h$ celic, katere prva vrstica se dotika obale, zadnja vrstica pa leži najdlje od obale na odprtem morju. Spremljajo gibanje rib in drugih reči, ki plavajo v morju. Morje je večinoma prazno, zanimivih pa je k celic, katerih položaj je opisan s številko vrstice r_i in stolpca c_i. Ribiči so ocenili, da če lovijo v i-ti celici, bo njihov ulov vreden v_i denarja. Število v_i je lahko tudi manjše ali enako 0, če so v tisti celici večinoma neželeni predmeti. Vse ostale celice obravnavamo tako, kot da so prazne in imajo vrednost 0.

Vaški svèt vsak dan določi pravokotno območje, v katerem bo ribolov tisti dan dovoljen; to območje obsega stolpce od ℓ do d in se razteza b vrstic od obale. Za ribolov v njem bodo ribiči bodo pripravili mrežo, dolgo natanko b enot. Njena dolžina je torej fiksna, dá pa se razviti na poljubno širino do največ $d - \ell + 1$. Na podlagi tega, kar vedo o morju, bodo mrežo položili nekje v pravokotnem območju, kjer je ribolov tisti dan dovoljen, pri čemer pa želijo maksimizirati skupno vrednost ulova, torej vsoto vseh celic, ki jih mreža pokriva.

Z vprašanjem, kako izbrati najboljši položaj mreže za ribolov, se ribiči ukvarjajo vsak dan. **Napiši program**, ki bo za naslednjih q dni izračunal vrednost ulova, pri čemer je za vsak dan podano območje, kjer bo ribolov takrat dovoljen. Predpostaviš lahko, da so vrednosti celic konstantne, torej se ne izčrpajo, četudi so tam ribarili že v prejšnjih dneh.

Vhodni podatki. V prvi vrstici so število vrstic h, število stolpcev w in število nepraznih celic k. Te celice so opisane v naslednjih k vrsticah, pri čemer i-ta od njih vsebuje številko vrstice r_i, stolpca c_i in vrednost v_i, ločene s po enim presledkom. Vrstice so oštevilčene od 1 do h, stolpci pa od 1 do w. Vse vrednosti v_i so cela števila.

V naslednji vrstici je število poizvedb q. Sledi q vrstic, ki opisujejo posamezne poizvedbe; j-ta od njih je opisana s tremi celimi števili b'_j, ℓ'_j in d'_j. Da bo tvoja rešitev prisiljena reševati poizvedbe po vrsti, so le-te podane v zakodirani obliki. Pravi opis poizvedbe lahko izračunaš po formulah

$$b_j = b'_j \oplus A_{j-3}, \quad \ell_j = \ell'_j \oplus A_{j-2}, \quad d_j = d'_j \oplus A_{j-1},$$

pri čemer A_j pomeni rezultat pri j-ti poizvedbi (oz. 0 pri $j \leq 0$), znak $\oplus$ pa predstavlja operacijo XOR (izključni ali) po bitih. Tvoj program naj poišče največji možni ulov po pravokotnem območju, ki se razteza po prvih b_j vrsticah in po nekem podintervalu stolpcev od ℓ_j do d_j.

Omejitve vhodnih podatkov:

- $1 \leq w, h, k, q \leq 300\,000$
- $|v_i| \leq 1000$

Izhodni podatki. Za vsako poizvedbo izpiši po eno vrstico, vanjo pa največjo možno vrednost ulova. Pazi na to, da se lahko ribiči odločijo ostati tudi pri prazni mreži z vrednostjo 0.

Primer vhoda:

```
10 7
12
2 6 -5
3 3 3
4 2 -2
4 6 2
5 3 -1
5 5 5
7 1 8
7 7 4
8 4 -3
8 5 1
9 6 -4
10 3 2
6
5 1 5
10 1 0
7 1 11
15 15 6
9 1 0
3 7 1
```

Pripadajoči izhod:

```
7
13
0
6
3
0
```

Opomba. Dekodirano zaporedje poizvedb pri tem primeru je:

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

F. Črke

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

Martin posluša predavanje o linearni algebri. Profesor, ki predava, je kajpada najdolgočasnejši človek v celem vesolju. Na tabli je matrika reda $n \times m$. Nekateri elementi matrike so črke (angleške abecede), drugod pa so prazna mesta. Tu je

primer takšne matrike reda 6×8:

$$\begin{bmatrix} k & & l & & n & d & i & \\ & & & & & c & & \\ & & & & & & i & h \\ j & & & a & & & & \\ & & c & b & & & & \\ & & c & & & & e & f \end{bmatrix}.$$

Martinu se še sanja ne, kaj ta matrika predstavlja. Tako mu je dolgčas, da predavanju že pol ure ne sledi več. Vendar pa ima Martin zelo bujno domišljijo. Zamišlja si, da na matriko vpliva težnost in da vse črke v njej polzijo navzdol, dokler vsaka črka bodisi ne doseže dna ali pa pristane na črki pod seboj. V prvi fazi se tako gornja matrika spremeni v:

$$\begin{bmatrix} & & & & & & & \\ & & & & & & & \\ & & & & & & & \\ & & l & & & & i & \\ k & & c & a & & d & i & h \\ j & & c & b & n & c & e & f \end{bmatrix}.$$

Nato težnost spremeni svojo smer in zdaj vleče črke v levo. Zdaj smo v drugi fazi. Spet vse črke polzijo v levo, dokler vsaka črka bodisi ne doseže levega oklepaja ali pa se ustavi ob črki na svoji levi. Prejšnja matrika bi se tako spremenila v:

$$\begin{bmatrix} & & & & & & & \\ & & & & & & & \\ & & & & & & & \\ l & i & & & & & & \\ k & c & a & d & i & h & & \\ j & c & b & n & c & e & f & \end{bmatrix}.$$

Martin v mislih ponavlja ta postopek vse do konca dolgočasnega predavanja. Seveda se lahko po vsaki fazi (torej ko črke dosežejo vsaka svoj končni položaj) spremeni smer težnosti (možne smeri so štiri: levo, desno, gor in dol).

Napiši program, ki določi končni položaj črk v matriki. Seveda dobiš tudi točno zaporedje smeri težnosti.

Vhodni podatki. V prvi vrstici so tri cela števila, n, m in k, pri čemer je $n \times m$ velikost matrike, k pa je število faz.

V drugi vrstici je niz dolžine k, sestavljen iz črk `L`, `R`, `U` in `D`, ki predstavljajo smer gravitacije v posameznih fazah (`L` = levo, `R` = desno, `U` = gor in `D` = dol).

Preostalih n vrstic predstavlja matriko. V vsaki od njih je m znakov. Ti znaki so male črke angleške abecede ter pike „`.`“, ki predstavljajo prazna polja.

Omejitve vhodnih podatkov: $1 \leq n \leq 100$, $1 \leq m \leq 100$, $0 \leq k \leq 100$.

Izhodni podatki. Izpiši matriko, ki jo Martin dobi na koncu predavanja. Format matrike naj bo enak kot v vhodnih podatkih.

Primer vhoda:

```
6 8 5
DLURD
k.l.ndi.
.....c..
......ih
j..a....
..cb....
..c...ef
```

Pripadajoči izhod:

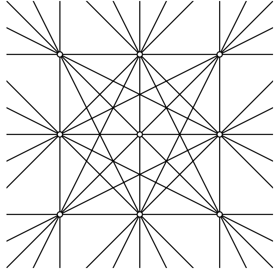
```
........
........
........
......hf
..iadice
.lkcbnjc
```

G. Premice na mreži

(*Omejitev časa: 8 s. Omejitev pomnilnika: 1024 MB.*)

Dana je karirasta mreža $n \times n$ točk s koordinatami (i, j) za $i = 0, 1, \ldots, n-1$ in $j = 0, 1, \ldots, n-1$. Naj bo ℓ_n število različnih premic, ki jih je mogoče postaviti tako, da na vsaki premici ležita vsaj dve točk mreže.

Pri $n = 3$ je mogoče na ta način postaviti 20 različnih premic, kot kaže naslednja slika:



Napiši program, ki za dane vrednosti n izračuna ℓ_n.

Vhodni podatki. V prvi vrstici je celo število q. V drugi vrstici je q celih števil $n_1, \ldots, n_q$, ločenih s presledki.

Omejitve vhodnih podatkov:

- $1 \leq q \leq 1000$
- $1 \leq n_i \leq 10^7$ za vse $i = 1, 2, \ldots, q$

Izhodni podatki. Izpiši q števil $\ell_{n_1}, \ldots, \ell_{n_q}$, vsako v svojo vrstico. Pravzaprav, ker utegnejo biti števila ℓ_n precej velika, izpiši le njihove ostanke po deljenju z $10^6 + 3$.

Primer vhoda:

```
3
1 3 2
```

Pripadajoči izhod:

```
0
20
6
```

H. Radar

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

Neko območje pregledujemo s posebnim radarjem. Temu lahko podamo seznam oddaljenosti, na primer $\langle 2, 4, 1 \rangle$, in seznam kotov oz. smeri, na primer $\langle 100°, 270°, 180°, 10°, 300° \rangle$, in pregledal bo točke na vseh teh oddaljenostih v vsaki od teh smeri. Kako blizu nekaterim drugim točkam, ki nas zanimajo, bomo lahko na ta način prišli?

Vhodni podatki. V prvi vrstici so tri cela števila, ločena s presledki: R (število oddaljenosti), F (število smeri oz. kotov) in N (število točk, ki nas zanimajo). Sledi R vrstic; i-ta od njih vsebuje celo število r_i, ki predstavlja oddaljenost od radarja do točk, ki jih bo pregledoval. Sledi F vrstic; i-ta od njih vsebuje dve s presledkom ločeni celi števili $(f_x)_i$ in $(f_y)_i$, ki sta kartezični koordinati točke, ki določa i-ti kot. Sledi še N vrstic, od katerih vsaka vsebuje dve s presledkom ločeni celi števili x_i in y_i, ki sta kartezični koordinati i-te točke, ki nas zanima.

Kót, ki ga določa točka $((f_x)_i, (f_y)_i)$, je tisti od pozitivne x-osi do poltraka, ki se začne v koordinatnem izhodišču in gre skozi točko $((f_x)_i, (f_y)_i)$.

Omejitve vhodnih podatkov:

- $1 \leq R, F, N \leq 10^5$
- $|x_i|, |y_i|, |(f_x)_i|, |(f_y)_i|, r_i$ so vse $< 10^6$
- $(f_x)_i^2 + (f_y)_i^2 > 0$ in $r_i > 0$
- vsi r_i so med seboj različni
- vsi poltraki, ki jih določajo $(f_x)_i, (f_y)_i$, so med seboj različni.

Izhodni podatki. Izpiši N vrstic; i-ta od njih naj vsebuje razdaljo med točko (x_i, y_i) in najbližjo tako točko, ki jo radar pregleda. Rezultat bo veljal za pravilnega, če ima absolutno ali relativno napako največ 10^{-6}.

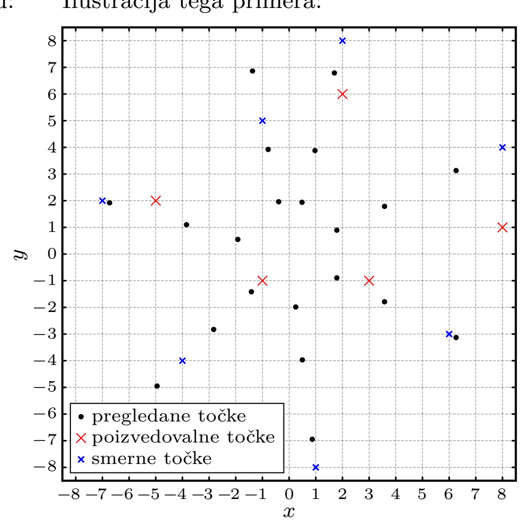
Primer vhoda:

```
3 7 5
2
4
7
8 4
2 8
-1 5
-7 2
-4 -4
1 -8
6 -3
3 -1
8 1
2 6
-5 2
-1 -1
```

Pripadajoči izhod:

```
0.977772290466
2.750120773895
0.846777708005
1.464071052924
0.585786437627
```

Ilustracija tega primera:



I. Pokrajinski razvoj

(*Omejitev časa: 4 s. Omejitev pomnilnika: 256 MB.*)

Kralja je doseglo več pritožb, češ da so nekatere pokrajine njegovega kraljestva gospodarsko zapostavljene. Na cestah med nekaterimi vasmi niso prebivalci že zelo dolgo videli niti enega trgovca. Da bi rešil ta problem ter kraljestvu povrnil bogastvo in blagostanje, je kralj zaupal svojemu dvornemu matematiku nalogo, naj pripravi primeren načrt poti trgovcev.

Načrt bo za vsako cesto določil pozitivno število trgovcev, ki bodo potovali po njej (vsi v isto smer). Za vsako vas mora biti število trgovcev, ki po cestah vstopajo vanjo, enako številu trgovcev, ki jo zapuščajo. Da bo število trgovcev približno enakomerno razporejeno po celem kraljestvu, je kralj naročil, naj bo na vsaki cesti število trgovcev, ki potujejo po njej, večje ali enako 1 ter manjše od m.

Dvorni matematik je bil poklican pred kralja, da mu predstavi svoje rezultate. Njegova prihodnost je negotova, saj mu problema ni uspelo rešiti. Nekaj pa je vendarle naredil: pripravil je načrt, pri katerem po vsaki cesti potuje veljavno število trgovcev (vsaj 1 in manj kot m). Težava je le, da se število prihajajočih in odhajajočih trgovcev za vsako vas ne ujemata čisto: razlika med številom prihajajočih in številom odhajajočih ni nujno pri vsaki vasi enaka 0, je pa gotovo njen ostanek po deljenju z m enak 0. Svoje ugotovitve je pripravljen deliti s teboj, če mu **napišeš program**, ki poišče veljaven načrt ali pa ugotovi, da ne obstaja.

Vhodni podatki. V prvi vrstici so cela števila n (število vasi), r (število cest) in m. Sledi r vrstic, ki opisujejo ceste; i-ta od njih vsebuje cela števila a_i, b_i in c_i, ki povedo, da po cesti, ki neposredno povezuje vasi a_i in b_i, potuje c_i trgovcev v smeri od a_i proti b_i. Vasi so oštevilčene od 1 do n. Za vsak par vasi obstaja največ ena cesta med njima; nobena cesta se ne začne in konča v isti vasi. Za vsako vas je razlika med skupnim številom prihajajočih in odhajajočih trgovcev enaka 0 po modulu m.

Omejitve vhodnih podatkov:

- $1 \leq n \leq 1000$; $0 \leq r \leq 10\,000$; $2 \leq m \leq 1000$
- $1 \leq a_i, b_i \leq n$ in $0 < c_i < m$ za vse $i = 1, 2, \ldots r$.

Izhodni podatki. Za vsako cesto izpiši število trgovcev, ki v tvojem načrtu potujejo po njej. Izpiši jih vsako v svojo vrstico in to v enakem vrstnem redu, v kakršnem so bile ceste podane v vhodnih podatkih. Če pri tvojem načrtu trgovci po neki cesti potujejo v nasprotni smeri kot v vhodnih podatkih, predstavi to z negativnim številom (na primer: če v tvojem načrtu potuje x trgovcev od vasi b_i do vasi a_i, izpiši v i-ti vrstici izhoda število $-x$).

Če obstaja več rešitev, je vseeno, katero od njih izpišeš. Če ne obstaja nobena rešitev, izpiši eno samo vrstico z besedo „`IMPOSSIBLE`“ (brez narekovajev).

Primer vhoda:

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

Pripadajoči izhod:

```
2
3
2
-1
3
```

J. Ponovitve

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

Polde je nadebudni avantgardni pisatelj, ki gleda zviška na rabo presledkov, ločil, velikih začetnic in podobnega; zato niso njegove zgodbe nič drugega kot dolgi nizi samih malih črk angleške abecede. Kritiki so opazili tudi, da je za njegov slog značilna nagnjenost k ponavljanju, namreč tako, da se včasih enak podniz pojavi v besedilu dvakrat zaporedoma, ne da bi vmes stali še kakršnikoli drugi znaki.

Svojo najnovejšo mojstrovino, niz dolžine n znakov, je Polde poslal q literarnim revijam v upanju, da jo bo vsaj ena od njih pripravljena objaviti. Odziv je bil mnogo ugodnejši, kot si je bil upal pričakovati. Uredniki vseh q revij so bili pripravljeni objaviti nek del (torej podniz) njegove zgodbe, vendar le pod pogojem, da poiščče v njem najdaljšo ponovitev (torej tak krajši podniz, ki se pojavi dvakrat zaporedoma). Tisto ponovitev nameravajo uredniki namreč izpustiti, da zgodba ne bo preveč dolgočasna. Polde zdaj potrebuje tvojo pomoč, da bo urednikom odgovoril na ta vprašanja.

Napiši program, ki za dani niz n znakov, $s[1]s[2]\ldots s[n]$, odgovori na q poizvedb oblike „dana sta a_i in b_i; kako dolg je najdaljši niz t, za katerega se tt pojavlja kot podniz v $s[a_i]s[a_i+1]\ldots s[b_i-1]s[b_i]$ in kje v slednjem se začne prva (najbolj leva) taka pojavitev?“

Vhodni podatki. V prvi vrstici sta dve celi števili, n in q. V drugi vrstici je niz s, ki je dolg n znakov; vsi ti znaki so male črke angleške abecede. Preostalih q vrstic opisuje poizvedbe; i-ta od teh vrstic vsebuje celi števili a_i in b_i, ločeni s presledkom.

Omejitve vhodnih podatkov:

- $1 \le n \le 10^6$
- $1 \le q \le 100$
- $1 \le a_i \le b_i \le n$ za vse $i = 1, 2, \ldots, q$

Izhodni podatki. Izpiši q vrstic; i-ta od teh vrstic mora vsebovati dve celi števili ℓ_i in c_i, ločeni s presledkom. Pri tem naj bo ℓ_i dolžina najdaljšega niza t, za katerega se tt pojavlja kot podniz v $s[a_i]s[a_i+1]\ldots s[b_i-1]s[b_i]$; število c_i pa naj bo indeks, na katerem se začne najbolj leva takšna pojavitev, torej najmanjše celo število, za katero velja $a_i \le c_i$ in $c_i + 2\ell_i - 1 \le b_i$ in $s[c_i]\ldots s[c_i+\ell_i-1] = s[c_i+\ell_i]\ldots s[c_i+2\ell_i-1]$. (Če je $\ell_i = 0$, je c_i po definiciji enako a_i.)

Primer vhoda:

```
10 4
cabaabaaca
4 8
1 9
5 9
8 10
```

Pripadajoči izhod:

```
1 4
3 2
1 7
0 8
```

Opomba: štiri poizvedbe v gornjem primeru se nanašajo na podnize

*a*abaa, *caba*abaac, ab*a*ac in aca;

v ležečem tisku je podniz, o katerem govori odgovor na tisto poizvedbo (podniz dolžine ℓ_i z začetkom na indeksu c_i). Pri četrti poizvedbi takega podvojenega podniza sploh ni, zato je $\ell_4 = 0$.

K. Enotirna železnica

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

Vlaki, ki vozijo po enotirni železniški progi, se lahko srečujejo le na postajah. Če se dva vlaka odpeljeta hkrati, eden z začetne in eden s končne postaje (torej z začetne postaje za nasprotno smer vožnje), bo eden od njiju običajno moral počakati drugega na eni od postaj vzdolž proge. Vlaka se vedno srečata na tisti postaji, kjer je čas čakanja najmanjši.

Čase vožnje po vsakem odseku med dvema zaporednima postajama poznamo; ti časi so vedno enaki za obe smeri vožnje. Zaradi gradbenih del na progi pa se časi vožnje nenehno spreminjajo. Poleg začetnih časov vožnje je podan tudi novi čas vožnje po vsaki spremembi; **napiši program**, ki izpiše najmanjši potrebni čas čakanja, če odpeljeta vlaka hkrati z nasprotnih koncev proge, in sicer tako za začetno stanje kot po vsaki izmed sprememb.

Vhodni podatki. V prvi vrstici je število postaj n. V drugi vrstici je $n-1$ števil, ki povedo začetne čase vožnje med dvema zaporednima postajama (i-to število je čas vožnje med i-to in $(i+1)$-vo postajo). V tretji vrstici je število sprememb k. Vsaka od preostalih k vrstic vsebuje po dve celi števili: prvo, $j \in [1, n-1]$, je številka postaje, drugo pa je novi čas vožnje med postajama j in $j+1$. Upoštevaj, da ima prva postaja številko 1 in ne 0.

Omejitve vhodnih podatkov:

- $2 \leq n \leq 200\,000$
- $0 \leq k \leq 200\,000$
- Vsi časi vožnje med dvema zaporednima postajama (tako začetni kot spremenjeni) so cela števila z intervala $[1, 10^6]$.

Izhodni podatki. Izpiši $k+1$ vrstic. V prvo vrstico izpiši najkrajši možni čas čakanja v začetnem stanju prog. V drugi, tretji, četrti, ... vrstici izpiši najkrajši možni čas čakanja po prvi, drugi, tretji, ... spremembi.

Primer vhoda:	Pripadajoči izhod:

`6`
`20 70 40 10 50`
`2`
`4 80`
`2 30`
	`10`
	`0`
	`40`

Komentar. V začetnem stanju se morata vlaka, ki odpeljeta istočasno z nasprotnih koncev proge, srečati na postaji 3. Prvi vlak jo bo dosegel v 90 minutah, drugi pa v 100 minutah; čas čakanja bo torej 10 minut. Po prvi spremembi postane najprimernejši kraj za srečanje postaja 4. Oba vlaka jo bosta dosegla v 130 minutah, zato ne bo nobenemu treba čakati. Tudi po drugi spremembi se bosta srečala na postaji 4, vendar pa bo moral tokrat vlak, ki bo prišel tja kot prvi, čakati 40 minut.

L. Sistematični trgovski potnik

(*Omejitev časa: 6 s.* *Omejitev pomnilnika: 256 MB.*)

Trgovski potnik je dobil seznam mest, ki jih mora obiskati na naslednjem potovanju. Vseeno je, v katerem mestu začne svojo pot, da le obišče vsako mesto vsaj enkrat; tudi ni treba, da konča v mestu, kjer je začel. Trgovski potnik je opazil, da porabljajo njegovi kolegi, drugi trgovski potniki, precej preveč časa za načrtovanje in iskanje optimalne poti. Zato se je odločil izbrati svojo pot z drugačnim, bolj sistematičnim pristopom.

Naprej bo razdelil vsa mesta na levo in desno polovico. Če je število mest liho, bo vsebovala desna polovica eno mesto več kot leva. Nato bo izbral eno od polovic in obiskal vsa mesta v tisti polovici, preden bo obiskal katerokoli mesto iz druge polovice.

Da obišče mesta v izbrani levi ali desni polovici, pa bo to množico mest razdelil na zgornjo in spodnjo polovico. Če bo v množici liho število mest, bo dobila zgornja polovica eno mesto več kot spodnja. Spet bo obiskal vsa mesta v eni od polovic, preden bo obiska katerokoli mesto druge polovice.

Po tem postopku bo nadaljeval in izmenično delil mesta na polovico — enkrat vodoravno, enkrat navpično — dokler ne bo sestavil načrta celotnega potovanja. **Napiši program**, ki poišče najkrajšo pot, ki jo lahko trgovski potnik dobi na ta način.

Vhodni podatki. V prvi vrstici je n, število mest, ki jih želi trgovski potnik obiskati. Položaj teh mest je podan v naslednjih n vrsticah. Vsako mesto opisujeta celoštevilski koordinati x_i in y_i (ločeni s presledkom), ki podajata njegov položaj na ravnini. Zagotovljeno je, da so x_i vseh mest med seboj različni; ravno tako so tudi y_i vseh mest med seboj različni.

Omejitve vhodnih podatkov:

- $1 \leq n \leq 1000$
- $0 \leq x_i, y_i \leq 10^6$

Izhodni podatki. V prvo vrstico izpiši najmanjšo možno dolžino potnikove poti. Dolžina bo veljala pravilno, če se bo od uradne rešitve razlikovala za kvečjemu 10^{-4}. V drugo vrstico izpiši številke mest (ločene s presledki) v takem vrstnem redu, v kakršnem jih trgovski potnik obišče. Mesta so oštevilčena od 1 do n v takem vrstnem redu, v kakršnem se pojavljajo v vhodnih podatkih. Če je možnih več enako dobrih rešitev, lahko izpišeš katerokoli od njih.

Primer vhoda:

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

Eden od možnih pripadajočih izhodov:

```
13.142182
3 4 1 6 5 2
```

Komentar. Trgovski potnik najprej obišče levo polovico (mesta 1, 3 in 4), nato pa desno polovico (mesta 2, 5 in 6).

Da obišče mesta 1, 3 in 4, obišče najprej zgornjo polovico (mesti 3 in 4) in nato spodnjo polovico (mesto 1). Mesti v zgornji polovici razdeli na levo polovico (mesto 3), ki jo obišče najprej, in desno polovico (mesto 4), ki jo obišče zatem.

Mesta 2, 5 in 6 razdeli na spodnjo polovico (mesto 6) in zgornjo polovico (mesti 2 in 5). Tu trgovski potnik najprej obišče spodnjo polovico. Svojo pot nadaljuje v zgornji polovici, kjer najprej obišče desno polovico (mesto 5) in zaključi z levo polovico (mesto 2).

POSKUSNO TEKMOVANJE
(23. aprila 2022)

X. Anagram
(*Omejitev časa: 4 s. Omejitev pomnilnika: 256 MB.*)

Besedi sta *anagrama*, če je mogoče vrstni red črk prve besede premešati tako, da nastane druga beseda. Primer anagramov sta besedi *enajst* in *stanje*.

Dobil boš seznam besed, ki so sestavljene le iz malih črk. Tvoja naloga je prečistiti ta seznam tako, da posamezno besedo izpustiš, če se je kdaj prej v seznamu že pojavil kak njen anagram.

Vhodni podatki. V prvi vrstici je n, število besed v seznamu. Sledi n vrstic, ki vsebujejo vsaka po eno besedo.

Omejitve vhodnih podatkov:

- $1 \leq n \leq 10^5$
- V besedah nastopajo le male črke angleške abecede.
- Dolžina posamezne besede je vsaj 1 in največ 100 znakov.

Izhodni podatki. Izpiši seznam besed brez anagramov, vsako besedo v svojo vrstico. Besede morajo stati v enakem vrstnem redu kot v vhodnih podatkih.

Primer vhoda:

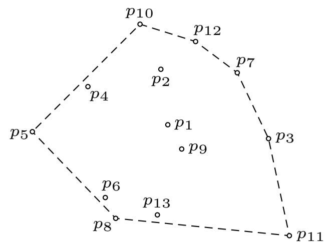
```
5
listen
santa
satan
silent
cat
```

Pripadajoči izhod:

```
listen
santa
cat
```

Y. E(dolžina(KO))
(*Omejitev časa: 2 s. Omejitev pomnilnika: 256 MB.*)

Danih je n točk na ravnini; i-ta točka bo *aktivirana* z verjetnostjo p_i. Te verjetnosti so del vhodnih podatkov. **Napiši program**, ki izračuna pričakovano vrednost obsega konveksne ovojnice aktiviranih točk.



Vhodni podatki. V prvi vrstici sta celo število n (število točk) in realno število $p^\star$. Vsaka od točk je aktivirana z verjetnostjo $p_i = p^\star$, le prve tri točke so vedno aktivirane ($p_1 = p_2 = p_3 = 1$); tako bo konveksna ovojnica vedno dobro definirana.

Sledi n vrstic, pri čemer i-ta od njih vsebuje koordinati i-te točke, x_i in y_i, ločeni s presledkom.

Omejitve vhodnih podatkov:

- $3 \leq n \leq 1000$; $0 \leq p^\star \leq 1$
- $0 \leq x_i, y_i \leq 10\,000$; x_i in y_i sta celi števili.
- Nobene tri točke ne ležijo na isti premici. Nobeni dve točki nimata niti enake x-koordinate niti enake y-koordinate.

Izhodni podatki. Izpiši eno samo število, namreč pričakovani obseg konveksne ovojnice. Da bo tvoja rešitev veljala za pravilno, se mora od uradne rešitve razlikovati za manj kot 0,001.

Primer vhoda:

```
4 0.281250
6 6
5 8
8 3
7 10
```

Pripadajoči izhod:

```
12.816849
```

Še en primer vhoda:

```
5 0.561523
2 11
7 8
13 10
9 9
13 3
```

Pripadajoči izhod:

```
27.943471
```

Z. Robin Hood

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

Vaški starešine pričakujejo hudo zimo in Robina Hooda skrbi, kako bodo shajali revnejši ljudje. Kot običajno bo bogastvo po kraljestvu malo prerazporedil — z drugimi besedami, kradel bo bogatim. Ocenil je, da bo treba izvesti k tatinskih podvigov. Vendar pa se Robin Hood drži moralnega zakonika, ki določa, kdo je najprimernejša tarča takega podviga. Vedno krade od najbogatejšega človeka; če je takih več, izbere izmed njih prvega na seznamu. Pri vsakem podvigu ukrade le

100 denarnih enot in nikoli ne okrade človeka, ki bi mu po kraji ostalo 0 ali manj denarnih enot.

Dobil boš podatke o premoženju n ljudi, poleg tega pa še število tatvin k. **Napiši program**, ki za vsakega človeka izračuna, koliko denarja mu bo ostalo po koncu teh tatvin, pri čemer so bile le-te izvedene v skladu z opisanim moralnim zakonikom.

Vhodni podatki. V prvi vrstici sta dve celi števili, n in k, ločeni s presledkom. V drugi vrstici je n s presledki ločenih celih števil $p_1, \ldots, p_n$, kjer je p_i premoženje i-tega človeka (potencialne tarče Robinovih tatvin).

Omejitve vhodnih podatkov:

- $1 \leq n, k \leq 10^5$
- $1 \leq p_i \leq 10^9$ za vse $i = 1, 2, \ldots, n$

Izhodni podatki. Izpiši premoženje vseh n ljudi po izvedenih k tatvinah; to naj bodo cela števila, ločena s po enim presledkom, vsa v eni vrstici, vrstni red ljudi pa naj bo enak kot v vhodnih podatkih. Če Robin Hood toliko tatvin sploh ne more izvesti, pa izpiši „`impossible`“ (brez narekovajev).

Primer vhoda:

```
4 2
100 120 250 13
```

Pripadajoči izhod:

```
100 120 50 13
```

Še en primer vhoda:

```
4 4
100 120 250 13
```

Pripadajoči izhod:

```
impossible
```

Še tretji primer vhoda:

```
3 4
200 300 300
```

Pripadajoči izhod:

```
100 100 200
```

NEUPORABLJENE NALOGE IZ LETA 2019

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 14. tekmovanjem ACM v znanju računalništva (leta 2019), 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. 157–206) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Predstavitve

Pri predstavitvah (prezentacijah) pogosto želimo prikazovati vsebino posamezne prosojnice postopoma, tako da se ne prikaže cela vsebina prosojnice naenkrat. Nekateri formati (na primer PDF) pa takšnega prikazovanja ne podpirajo, zato moramo pri pretvorbi v tak format iz ene izvorne prosojnice narediti več strani, ki vsebujejo vse večji del vsebine tiste izvorne prosojnice.

Predstavitev imamo opisano z zaporedjem ukazov „nova stran“, „nova alineja“, „pavza“, med njimi pa je poljubno besedilo. Pri pretvorbi v PDF bi se pri vsakem ukazu „pavza“ (in tudi na koncu vsake prosojnice) naredilo novo stran, ki vsebuje vso vsebino od zadnjega ukaza „nova stran“ do trenutne pavze. Pri tem pa ven vržemo več sledečih si ukazov „nova alineja“ in „pavza“, če vmes ni besedila (ne dovolimo praznih alinej in nespremenjenih prosojnic).

Napiši program, ki prebere opis predstavitve in ga izpiše v tako predelani obliki. Da bo izpis preglednejši, izpiši med strani še po eno prazno vrstico.

Primer vhoda:

```
nova stran
ena
nova alineja
dve
nova alineja
pavza
tri
nova stran
stiri
```

Predelana predstavitev:

```
nova stran
ena
nova alineja
dve

nova stran
ena
nova alineja
dve
nova alineja
tri

nova stran
stiri
```

2. Urnik

Dan je seznam predmetov na urniku kot seznam četveric oblike ⟨matematika, torek, učilnica 31, 3. ura⟩. Predlagaj podatkovno strukturo, ki hitro odgovarja na poizvedbe oblike „Katere učilnice so proste ob torkih 4. uro?“ **Opiši** tudi **postopek**, ki predela vhodni seznam v to podatkovno strukturo, in postopek, ki odgovarja na omenjene poizvedbe.

3. Pravokotnik iz kvadratov

Danih je n kvadratov s celoštevilskimi stranicami $c_1, c_2, \ldots, c_n$. Radi bi sestavili pravokotnik, ki ima enako ploščino kot vsi ti kvadrati skupaj, pri tem pa morajo biti njegove stranice celoštevilske in biti mora čim bolj kvadraten (torej: razmerje med dolžino daljše in krajše stranice mora biti čim bližje 1). **Opiši postopek**, ki poišče najbolj kvadraten tak pravokotnik.

Težja različica: ni ti treba uporabiti vseh n vhodnih kvadratov, pač pa si lahko izbereš neko podmnožico njih (vendar vsaj dva) in potem iščeš pravokotnik s celoštevilskimi stranicami, ki ima tako ploščino kot izbrani kvadrati skupaj. Opiši postopek, ki poišče najbolj kvadraten pravokotnik, ki ga je mogoče sestaviti na ta način.

4. Seznama

V neki cirkuški točki nastopa $2n$ akrobatov, od katerih jih n nosi zelene hlače, n pa oranžne. Akrobati se morajo razporediti v n parov, pri čemer bo v vsakem paru po en akrobat z zelenimi hlačami in eden z oranžnimi. Zaradi varnosti in ravnotežja si želimo, da bi si bila akrobata v vsakem paru čim bolj podobna po teži. Natančneje povedano, če v i-tem paru (za $i = 1, \ldots, n$) sodelujeta akrobata s težama a_i in b_i, bomo celotni razpored akrobatov v pare ocenili z vsoto $\sum_{i=1}^{n} (a_i - b_i)^2$. Teže vseh akrobatov so znane in so podane kot realna števila.

(a) **Opiši postopek**, ki poišče najboljši možni razpored akrobatov na pare. Poleg tega tudi utemelji pravilnost svojega postopka.

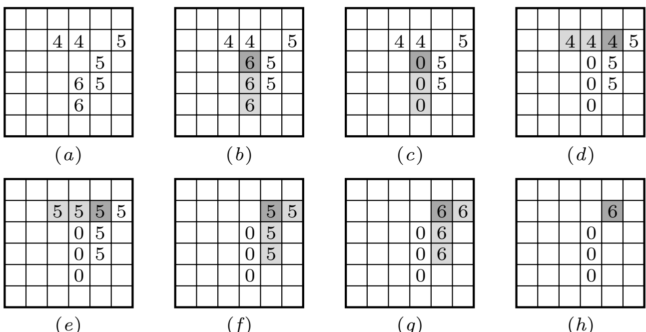
(b) Kaj pa, če si akrobati še niso oblekli hlač in lahko, preden jih razdelimo v pare, tudi določimo, kateri med njimi bodo oblekli zelene in kateri oranžne hlače? Reši nalogo še za ta primer.

5. Napredovanje števil

Iz nabora števil od 1 do 6 naključno izbiramo števila in jih polagamo v polja mreže 6×6. Pravila igre so naslednja:

- Na začetku igre so vsa polja prazna. Vsaka poteza igre se sestoji iz tega, da dobimo neko naključno število od 1 do 6 in ga postavimo v eno od praznih polj mreže (to polje si izberemo mi).
- *Skupino* predstavljajo najmanj tri sosednja polja, v katerih je enako število. Pri tem veljata dve polji za sosednji, če imata skupno stranico.
- Ko imamo skupino najmanj treh sosednjih polj, v katerih se nahaja enako število iz nabora od 1 do 5, celotna skupina samodejno napreduje v naslednje višje število. Novo višje število ostane v polju, v katero smo v to skupino nazadnje vpisali kakšno število, vsa ostala polja te skupine pa se izpraznijo.
- Ko imamo skupino najmanj treh sosednjih polj, v katerih se nahaja število 6, vsa polja celotne skupino samodejno napredujejo v končno število 0. Polja, v katerih se nahaja število 0, ne morejo biti več uporabljena pri tvorjenju skupin.
- Igra se konča, ko je celotna mreža zapolnjena; cilj je takrat imeti na čim več poljih ničle.

Napiši program ali podprogram, ki za dano zaporedje potez ugotovi, ali je predstavlja veljaven potek igre od začetka (prazne mreže) do konca (popolnoma polne mreže) in izračuna končno stanje mreže ter število ničel v njej. Kot podatke o posamezni potezi dobi izbrano naključno število (od 1 do 6) ter koordinati (torej številko vrstice in stolpca) polja, v katero smo to število postavili. Podrobnosti glede oblike vhodnih in izhodnih podatkov si izberi sam.



Primer dveh zaporednih potez. — (a) Stanje mreže pred prvo od teh dveh potez. — (b) Dobili smo naključno število 6 in ga dodali v temno sivo polje; skupaj z dvema svetlo sivima poljema je nastala skupina treh polj s številko 6. — (c) Skupina je napredovala v 0; prve poteze je tu konec. — (d) Začetek druge poteze: dobili smo naključno število 4 in ga dodali v temno sivo polje; skupaj z dvema svetlo sivima poljema je nastala skupina treh polj s številko 4. — (e) Skupina je napredovala v 5. — (f) V temno sivem polju, kamor smo nazadnje postavili število, je le-to ostalo, preostanek dosedanje skupine se je izpraznil. Temno sivo polje zdaj tvori skupaj s še tremi svetlo sivimi novo skupino z vrednostjo 5. — (g) Nova skupina je napredovala v 6. — (h) V temno sivem polju, kamor smo nazadnje postavili število, je le-to ostalo, preostanek dosedanje skupine se je izpraznil. Druga poteza je s tem končana.

6. Študentski servis

Metka se odpravlja na potovanje po Evropi. A kaj, ko je za vse njene cilje prihranjenega denarja premalo. Zato se odpravi na študentski servis, da bi si našla delo. Na študentskem servisu imajo vrsto del. Vsako je določeno z časom začetka, časom konca in zaslužkom. Seveda lahko vsako delo Metka prevzame le v celoti in v tistem obdobju ne more delati ničesar drugega. Pomagaj Metki in **napiši podprogram**, ki za dano tabelo trojic ⟨začetek, konec, zaslužek⟩ vrne največji znesek, ki ga Metka lahko zasluži.

7. Pandemija

V paralelnem vesolju se nahaja planet Virus, ki je na las podoben Zemlji. Na njem je m mest, ki so oštevilčena od 1 do m, ter p dvosmernih cest med njimi. Ceste prebivalcem planeta služijo za premikanje med posameznimi mesti. Žal pa na planetu Virus ni vse tako rožnato kakor pri nas na Zemlji. Planetu Virus namreč grozi množičen izbruh štirih bolezni: gripe, ošpic, kuge in kolere.

Kot skrbni preučevalci izvenzemeljskih civilizacij opazujemo širjenje bolezni na planetu Virus. Opazovanja s preizkušenimi znanstvenimi metodami izvajamo v enakomernih časovnih intervalih. Razširjenost posamezne bolezni v vsakem mestu opisujemo s štirimi stopnjami: 1 = ni okužb; 2 = nizka prisotnost bolezni; 3 = visoka prisotnost bolezni; in 4 = epidemija. Iz dolgotrajnega opazovanja dinamike izbruhov bolezni smo se naučili nekaj pravil:

- bolezni so med seboj neodvisne;
- najvišja stopnja razširjenosti bolezni v posameznem mestu je epidemija;
- če je vsota stopenj razširjenosti bolezni v sosednjih mestih večja kot 6, se v opazovanem mestu razširjenost poveča za 1 (razen če je že bila najvišja možna, torej 4); in
- če v vsaj treh zaporednih časovnih intervalih nismo zaznali povečanja razširjenosti bolezni v posameznem mestu, lahko znižamo stopnjo razširjenosti bolezni za 1.

Če izbruh katerekoli bolezni v vseh mestih na planetu Virus doseže epidemične razsežnosti, civilizacija na planetu Virus izumre. Glede na to, da nam zmanjkuje denarja za draga opazovanja s preizkušenimi znanstvenimi metodami, te prosimo, da nam na podlagi zgornjega opisa širjenja bolezni izdelaš simulacijo za naslednjih 100 časovnih intervalov ter na izhod napišeš „`SE SO ZIVI`“, če civilizacija preživi, oziroma časovni interval izumrtja, če civilizacija izumre.

Na standardni vhod boš dobil podatke o številu mest, m, in številu povezav, p. V naslednjih m vrsticah te čakajo zadnje znane ocene o razširjenosti vsake izmed 4 bolezni (4 številke, ločene s presledki) za vsako mesto. V naslednjih p vrsticah se nahajajo pari mest, ki so povezani s cesto.

8. Tretji tir

Potem ko je drugi tir Divača–Koper žalostno propadel zaradi afere z njegovo maketo, je politika ugotovila, da potrebujemo popolnoma novo in neobremenjeno traso, ki bo najbolj optimalno povezala oba kraja.

Na razpolago imaš digitalni model reliefa v mreži kvadratov 50×50 metrov in naloga je najti tako traso, katere izgradnja je najcenejša med vsemi takimi, kjer naklon proge na nobenem mestu ne preseže 2 % (kolikor lokomotive še lahko zvlečejo težke tovorne kompozicije). Traso proge lahko opišemo z zaporedjem kvadratov v mreži, ki se stikajo s stranicami. Kar se zavojev tiče, je omejitev le ta, da se smer trase znotraj enega kvadrata v mreži ne sme spremeniti za več kot 90 stopinj (torej bo proga primerna tudi za vlakec smrti).

Cena za izgradnjo je naslednja:

- Če gre proga do 5 metrov nad ali pod površjem, stane to 1 DENEN (to je denarna enota na dolžinsko enoto v prostoru);
- Če gre proga več kot vključno 5 metrov nad površjem, potem stane 1 DENEN in še po 1 DENEN za vsakih 10 metrov nad površjem (podražitev zaradi gradnje nasipa ali mostu). Višinsko razliko nad površjem zaokrožimo na 10 metrov.

- Če gre proga več kot vključno 5 metrov pod površjem, potem stane 7 DENEN (podražitev zaradi gradnje predora).

Na standardni vhod boš v prvi vrstici dobil dimenziji digitanega reliefa w in h, za kateri velja $5 \leq w \leq 500$ in $5 \leq h \leq 500$. V drugi vrstici se bosta nahajali koordinati Divače, x_d in y_d, in Kopra, x_k in y_k. Sledila bo dvodimenzionalna matrika s podatki o nadmorski višini površja h_{xy}, ki predstavlja digitalni relief. Za vsak element velja $0 < h_{xy} < 500$. Digitalni model je poenostavljen, kar pomeni, da je nadmorska višina h_{xy} konstantna znotraj kvadrata, naklon pa se nanaša na spremembo nadmorske višine med sosednjimi kvadrati. (Z drugimi besedami: ker so kvadrati veliki 50×50 metrov in je največji dopustni naklon trase 2 %, to pomeni, da sme biti nadmorska višina trase v enem kvadratu kvečjemu za 1 meter višja ali nižja kot v sosednjem kvadratu. Nadmorska višina trase naj bo povsod celo število od 0 do 500.)

9. Tabela števil

Dobimo tabelo $n \times n$ celic, ki vsebuje števila od 1 do vključno n. Posamezno število se lahko pojavi enkrat, večkrat ali pa tudi nikoli. **Opiši postopek**, ki elemente te tabele prerazporedi tako, da bosta v vsaki vrsti največ dve različni števili.

Primer:

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

Gornja tabela ni ustrezna, saj prva in tretja vrstica vsebujeta po tri različna števila. Ena izmed pravilnih preureditev bi bila:

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

Odebeljena so mesta, kjer smo tabelo spreminjali. Ni treba minimizirati števila sprememb, le najti rešitev ali izpisati, da ne obstaja, če nam to ne uspe.

10. Film

Želiš prenesti film prek interneta, kjer poznaš vozlišča in kapacitete povezav med njimi. Znano je, na katerem vozlišču se film trenutno nahaja in katero vozlišče predstavlja tvoj računalnik. **Opiši postopek**, ki najde pot z največjo kapaciteto prenosa in ugotovi, koliko je ta kapaciteta. Podatke boš moral prenesti vse po eni poti, torej jih ne boš mogel npr. razdeliti na več delov in jih prenašati vzporedno po več različnih poteh hkrati. Med dvema vozliščema je lahko tudi več neposrednih povezav, vendar gre lahko tok podatkov le po eni.

Težja različica: povezave s kapaciteto, večjo od C, so pod nadzorom. Ko preneseš film prek ene take, zbudiš pozornost agentov Sove, ki te naslednjič, ko uporabiš

kakšno od velikih povezav, vržejo v zapor. Koliko je sedaj maksimalna hitrost prenosa?

11. Kodiranje besedila

Imamo besedilo, ki ga želimo kodirati. V vhodnem besedilu se pojavljajo le male črke angleške abecede in ne-črkovni znaki, ne pa tudi velike črke. Osnovni korak kodiranja je, da si izberemo neki par malih črk in potem v vsaki pojavitvi tega para v besedilu pobrišemo prvo črko para, drugo pa spremenimo iz male v veliko (npr. `kr` $\rightarrow$ `R`); tej veliki črki pravimo *koda* tega para. Posamezna velika črka se lahko uporabi kot koda pri največ enem paru malih črk. Kodiranje besedila poteka tako, da na vsakem koraku poiščemo prvi (torej najbolj levi) tak par malih črk, za katerega je ustrezna koda še prosta (torej tiste velike črke še nismo uporabili za kodiranje kakšnega drugega para malih črk), in tisti par (oz. vse njegove pojavitve) potem zakodiramo. Ko ni več nobenega para malih črk, ki bi se ga še dalo zakodirati, se postopek konča.

(*a*) **Napiši podprogram**, ki bere vhodno besedilo in ga sproti izpisuje v kodirani obliki, kakršna nastane na koncu zgoraj opisanega postopka.

(*b*) **Napiši podprogram**, ki kot parameter dobi tabelo (ali niz), ki za vsako veliko črko pove, katera je bila prva mala črka v paru, ki je bil zakodiran s tisto veliko črko (če sploh katera); tvoj podprogram naj bere kodirano besedilo in ga sproti izpisuje v dekodirani obliki.

Primer: naslednji primer kaže korake pri kodiranju niza „`ce bi cebela ne bila cebula`“.

Stanje besedila	Uporabljena koda
`ce bi cebela ne bila cebula`	(prvotno besedilo)
`E bi Ebela ne bila Ebula`	`ce` $\rightarrow$ `E`
`E I Ebela ne Ila Ebula`	`bi` $\rightarrow$ `I`
`E I EbLa ne Ila Ebula`	`el` $\rightarrow$ `L`
`E I EbLa ne IA EbuA`	`la` $\rightarrow$ `A`
`E I EbLa ne IA EUA`	`bu` $\rightarrow$ `U`

V tretjem koraku na primer nismo mogli zakodirati para `be`, ker je bila koda `E` že zasedena; prvi primerni par za kodiranje je bil zato `el`.

12. Rudarji

Vodja rudarskega sindikata v rudniku z d delavci in k rovi je pod pritiskom vodstva, ki želi zapreti nedobičkonosne rove in odpustiti nekaj delavcev. Pogaja se o najmanjši količini rude m, ki jo mora izkopati vsak rov do nekega roka, da ne pride do odpuščanja. Za vsak rov poznamo količino rude v zaporednih še neizkopanih kubičnih metrih. Vsoto dolžin rovov označimo z n. Če v rovu koplje x delavcev, bodo do roka izkopali x kubičnih metrov. **Opiši postopek**, ki izračuna, kakšna je največja kvota m, ki jo lahko dosežejo ob optimalni razporeditvi delavcev.

Primer: recimo, da imamo $d = 6$ delavcev in $k = 3$ rove, količina rude v njih pa je $[10, 20, 10, 50]$, $[30, 100, 10, 0, 5]$ in $[25, 10, 70]$; skupna dolžina teh seznamov je $n = 12$. Rešitev je potem $m = 30$: če v prvem rovu zaposlimo 3, v drugem 1, v

tretjem pa 2 delavca, bodo izpleni po rovih $10 + 20 + 10 = 40$, 30 in $25 + 10 = 35$. Višje kvote ne morejo doseči.

13. Največji xor

Dano je zaporedje n nenegativnih celih števil: $a_1, a_2, \ldots, a_n$. Zanima nas táko njegovo strnjeno podzaporedje, katerega XOR je maksimalen. (Operacija XOR (izključni ali) deluje tako, da je v rezultatu u xor v posamezni bit prižgan natanko tedaj, če je istoležni bit v enem od operandov u ali v prižgan, v enem pa ugasnjen.) Maksimizirati želimo torej vrednost a_ℓ xor $a_{\ell+1}$ xor $\cdots$ xor a_{d-1} xor a_d po vseh ℓ in d, kjer je $1 \leq \ell \leq d \leq n$. Preden začnemo, smemo še spremeniti do k bitov v vhodnih številih (1 v 0 ali obratno) — k bitov skupno, ne k v vsakem številu. **Opiši postopek**, ki poišče največji možni XOR pri pravilno izbranem strnjenem podzaporedju in tem, katere bite bi na začetku spremenili.

Nalogo si lahko predstavljamo v lažji in težji različici, kar je povezano s tem, da so vhodna števila lahko različno dolga. (Na primer, število $17 = 10001_2$ ima pet bitov, število $6 = 110_2$ pa samo tri bite.)

(a) Lažjo različico naloge dobimo, če si mislimo, da krajša števila vhodnega zaporedja dopolnimo na levi s toliko vodilnimi ničlami, da postanejo enako dolga kot najdaljše število vhodnega zaporedja, in da smejo tudi te vodilne ničle priti med tistih k bitov, ki jih spremenimo, preden začnemo XORati števila med sabo. (Pri gornjem primeru to na primer pomeni, da si število 6 predstavljamo kot 00110_2 in da lahko s spreminjanjem enega bita dobimo iz njega med drugim tudi $10110_2 = 22$ in $01110_2 = 14$.)

(b) Težjo različico naloge pa dobimo, če takih vodilnih ničel ne dovolimo; edino število, pri katerem lahko nastopi vodilna ničla, je število 0 (ki ga smemo s spremembo enega bita spremeniti v 1). (Če nadaljujemo primer od prej: iz števila $6 = 110_2$ lahko pri tej različici naloge, če smemo spremeniti en bit, dobimo le $100_2 = 4$, $010_2 = 2$ in $111_2 = 7$, ne pa tudi 22 in 14.)

Neodvisno od tega, katero od obeh gornjih različic vzamemo, lahko naredimo nalogo lažjo tudi tako, da dodamo eno od naslednjih omejitev: (c) vsi a_i so ≤ 3; (d) $k = 0$; (e) $k = 1$.

Primer: če imamo zaporedje $\langle 1, 1, 3, 4, 8, 2, 10 \rangle$ in $k = 0$, je najbolje vzeti $\ell = 3$ in $d = 5$, torej števila od tretjega do petega; njihov XOR je 3 xor 4 xor $8 = 11_2$ xor 100_2 xor $1000_2 = 1111_2 = 15$.

14. Prefiksna in postfiksna oblika

Pri tej nalogi se bomo ukvarjali z aritmetičnimi izrazi, v katerih nastopajo operatorji `+ - * /` in naravna števila; vsi operatorji se vedno uporabljajo z dvema operandoma (torej ni npr. unarnega minusa). Takšne izraze smo običajno navajeni pisati tako, da operator pišemo med njegovima operandoma, čemur pravimo *infiksni zapis*. Lahko pa tak izraz zapišemo tudi tako, da operator vedno postavimo pred njegova operanda (temu pravimo *prefiksni zapis*) ali pa vedno za njegova operanda (čemur pravimo *postfiksni zapis*). Ena od prednosti prefiksnega in postfiksnega zapisa pred infiksnim je, da pri njiju ne potrebujemo oklepajev. Za primer si oglejmo dva izraza v vseh treh oblikah:

infiksni zapis:	`(12 + 34) * ((56 - 78) * 9)`
prefiksni zapis:	`* + 12 34 * - 56 78 9`
postfiksni zapis:	`12 34 + 56 78 - 9 * *`
infiksni zapis:	`(12 + 34 * 56) - 78 * 9`
prefiksni zapis:	`- + 12 * 34 56 * 78 9`
postfiksni zapis:	`12 34 56 * 78 9 * -`

Napiši podprogram, ki prebere izraz v prefiksnem zapisu in ga izpiše v postfiksnem zapisu. Deluje naj učinkovito tudi za zelo dolge izraze.

15. Zbiratelj

Ivan se je odločil, da bo postal zbiratelj najnovejših sličic, ki se jih dobi ob nakupu nad 10 EUR v trgovini Intermarket. Obstaja $k \leq 15$ različnih sličic, med katerimi je zadnja najredkejša, zaradi česar si Ivan želi imeti le to in nobene druge (da se bo pred prijatelji lahko pohvalil, da je že v prvem poskusu dobil najredkejšo sličico). Na žalost pa je Ivan ob nakupu v Intermarketu dobil prvo, najpogostejšo sličico.

Na srečo je v Sloveniji veliko fanatičnih zbirateljev sličic, ki tvorijo $n \leq 100$ zbirateljskih kartelov. Vsak kartel ima natanko eno trgovino, v njej pa prodaja in kupuje le določene sličice; prav tako pa vsak kartel zaradi svoje fanatičnosti zahteva, da ima njihov kupec v lasti določeno sličico, v nasprotnem primeru potencialnih kupcev niti ne spusti v svojo trgovino.

Ivan mora pri svojih nakupih paziti tudi, v kakšnem vrstnem redu obišče trgovine različnih kartelov, saj so določeni med seboj skregani in zaradi tega v svojo trgovino ne spustijo kupcev, ki so nazadnje kupovali v trgovini njihovih sovražnikov. Na srečo pa pri preverjanju tega karteli niso preveč pozorni, dovolj je le da Ivan tik pred obiskom obišče trgovino nekega kartela, ki ni skregan s kartelom, katerega trgovino želi obiskati (mora pa seveda paziti, da ima pravo sličico, da ga bodo v to trgovino sploh spustili).

Ivan ima na začetku samo prvo sličico in se nahaja v trgovini prvega kartela. Pomagaj mu ugotoviti, v koliko najmanj korakih (kjer so možni koraki: obisk trgovine, nakup sličice, prodaja sličice) lahko kupi zadnjo sličico in proda vse ostale, ki jih je mogoče pridobil med svojo nalogo.

Vhodni podatki: prva vrstica: k n m (število sličic; število kartelov; število parov kartelov, ki so skregani); naslednjih n vrstic: z_i r_i c_{i1} c_{i2} $\ldots$ c_{i,r_i} (sličica, ki jo kartel i zahteva ob vstopu v trgovino; število sličic, ki jih i-ti kartel prodaja/kupuje; in zaporedne številke teh sličic); naslednjih m vrstic: a_i b_i (zaporedni številki kartelov, ki sta skregana).

Izhodni podatki: izpiši najmanjše število korakov.

16. Stave

Metka navdušeno spremlja nogomet in vsakič stavi na eno od ekip skupaj s svojimi $n - 1$ prijatelji. Skupaj gledajo tekme in vsi, ki so pravilno napovedali rezultat, dobijo po eno točko. Metka je na razpredelnici trenutno v vodstvu (nihče nima več točk od nje).

Na tem mestu se želi obdržati, zato goljufa. Vsakič izvohuni, kaj so stavili drugi, preden položi svojo stavo. Nato stavi enako, kot je stavil tisti, ki ima na razpredelnici

največ točk. Če je takih ljudi več, izbere večinski glas (med najboljšimi). V primeru izenačenja stavi na naključno izmed ekip. **Opiši postopek**, ki ugotovi, koliko časa bo Metka v vodstvu v najslabšem scenariju, če vedno stavi po tem postopku. Velja $2 < n < 10^5$, trenutne vrednosti v razpredelnici so manj kot 10^{16}.

17. Zamik

V skladišču stoji v vrsti n zabojev. Mesta, na katerih stojijo, si mislimo oštevilčena od leve proti desni s celimi števili od 0 do $n - 1$. Za premikanje zabojev je na voljo podprogram `Zamenjaj(a, b)`, ki krmili sistem robotskih rok v skladišču in z njimi zamenja zaboja na mestih a in b, tako da po tej zamenjavi stoji na mestu b tisti zaboj, ki je pred njo stal na mestu a, in obratno. **Napiši podprogram** `Zamakni(n, k)`, ki kot parameter dobi število zabojev n in še celo število k, ki je z območja $0 \leq k < n$. Tvoj podprogram naj poskrbi, da se vsi zaboji ciklično zamaknejo za k mest v levo. (To pomeni, naj se najbolj levih k zabojev premakne na konec zaporedja, vsi preostali zaboji pa se premaknejo k mest v levo).

Primer: če imamo $n = 5$ zabojev in jih oštevilčimo od 0 do 4 glede na to, na katerem mestu so stali na začetku, bo stanje po zamiku za $k = 2$ mesti takšno: $[2, 3, 4, 0, 1]$.

18. Človeške ribice

Ministrstvo za Uspešno in Zadovoljivo urejanje Akvatnega življa (na kratko MUZA) se je odločilo narediti umetno jamo, v katero si želijo namestiti človeške ribice. Jama bo bo sestavljena iz sistema n podzemeljskih soban, ki so povezane z $n - 1$ hodniki, tako da tvorijo drevesasto razvejeno strukturo, po kateri je mogoče iz vsake sobane priti do vsake druge sobane po natanko eni poti (hodniki torej ne tvorijo ciklov). Sobane so na znanih celoštevilskih globinah. Ena od soban je na globini 0; njej pravimo *koren* jame. Za vsak hodnik velja, da sta sobani, ki ju ta hodnik neposredno povezuje, na različnih globinah; za globljo od njiju pravimo, da je *otrok* druge sobane (tiste, ki je na nižji globini, torej bliže površja), za to drugo pa, da je *starš* tiste prve, globlje sobane. Vsaka sobana razen korena ima natanko enega starša, koren pa nobenega. Vsaka sobana ima lahko nič ali več otrok; tistim brez otrok pravimo *listi* jame. Za vsako sobano poznamo njeno prostornino (koliko vode lahko sprejme), enako pa tudi za vsak hodnik.

Ker pa se je dostop do korenske sobane zaprl, bodo jamo napolnili tako, da bodo v določene listne sobane injicirali vodo pod pritiskom. **Opiši postopek**, ki ugotovi, koliko soban bo po končanih dotakanjih vode popolnoma poplavljenih.

Če ℓ litrov vode injiciramo v neko listno sobano, se bo voda obnašala, kot pričakujemo: voda bo napolnila celotno sobano in povezavo nad njo (če je vode dovolj), preostanek vode pa bo napolnil starševsko sobano, in sicer takole:

- Če so vse sobane in povezave pod njo polne, se bo začela polniti ta sobana, kar pa bo ostalo vode, gre navzgor.
- Če obstaja nepolna sobana pod starševsko sobano, se bo najprej v celoti napolnil najglobلji otrok starševske sobane (in povezava do njega), po enakih pravilih kot starševska sobana; s preostankom vode se bo nato podobno napolnil drugi najgloblji otrok, nato tretji, ..., dokler ne bodo vse sobane pod

staršEvsko polne; nato se bo s preostankom polnila starševska sobana. Če ima starševska sobana več otrok na enaki globini, se bodo ti otroci polnili v takem vrstnem redu, v kakršnem so omenjeni v vhodnih podatkih.

Vhodni podatki: v prvi vrstici je sta n (število soban) in k (število injiciranj vode); nato sledi n vrstic, kjer i-ta predstavlja i-to sobano in vsebuje: p_i (številka starša i-te sobane), g_i (globino i-te sobane), v_i (prostornino i-te sobane), v'_i (prostornino hodnika med i in p_i); nato sledi k vrstic, kjer i-ta opisuje i-to injiciranje vode in vsebuje števili a_i (indeks listne sobane, v katero injiciramo vodo), ℓ_i (količino vode). Vsa števila so cela števila; sobane so oštevilčene s številkami od 1 do n. Pri korenu, ki starša nima, je $p_i = v'_i = 0$.

Omejitve: $2 \leq n \leq 10^6$; prostornine v_i, v'_i, ℓ_i so ≥ 0; vsota vseh v_i in v'_i je $\leq 10^9$, ravno tako vsota vseh ℓ_i.

Izhodni podatki: izpiši k vrstic, pri čemer naj i-ta od njih vsebuje indekse tistih soban, ki so po i-tem injiciranju vode že čisto zapolnjene z vodo, pred njim pa še niso bile.

19. Cenena konferenca

Na konferenco smo poslali n znanstvenikov, ki so konferenco vzeli zelo resno. Glavni del konference je predstavitev plakatov, ki so široki vsak po 1 m in postavljeni v ravni vrsti brez presledkov eden ob drugem.

Znanstveniki si plakate ogledujejo tako, da i-ti znanstvenik začne na začetku in si ogleda vse plakate od prvega plakata do k_i-tega (pri tem prehodi k_i metrov). Ker pa so znanstveniki nerodni, moramo za vsak prehojen meter plačati zavarovanje, ki nas stane 1 evro na meter.

Novi direktor želi čim bolj zmanjšati ceno zavarovanja, hkrati pa še vedno dovoliti ogled določenega števila plaktov; tu nastopiš ti. Edini način, da si znanstveniki ogledajo manj plakatov, je, da za b-tim plakatom potegnemo trak, ki prepoveduje prehod. Tako si znanstvenik i ogleda $\min\{b, k_i\}$ plakatov. Skupna cena zavarovanja, ki jo bo treba plačevati, je potem enaka vsoti vrednosti $\min\{b, k_i\}$ po vseh i od 1 do n.

Direktor je pripravil seznam m različnih zneskov $s_1, s_2, \ldots, s_m$ in sedaj ga za vsakega od teh zneskov zanima, največ kolikšen je lahko b, tako da bo skupna cena zavarovanja še vedno manjša ali enaka tistemu znesku. **Opiši postopek**, ki to ugotovi (kot vhodne podatke dobi števili n in m ter seznama $k_1, k_2, \ldots, k_n$ in $s_1, s_2, \ldots, s_m$). Plakatov je več, kot si jih katerikoli znanstvenik želi ogledati, zato njihovo točno število ni pomembno in tudi ni podano.

Primer: recimo, da imamo $n = 3$ znanstvenike in $k_1 = 5$, $k_2 = 10$ in $k_3 = 8$. Potem, če na primer hočemo, da je cena zavarovanja kvečjemu 20, moramo vzeti $b = 7$ ali manj (pri $b = 7$ bo cena zavarovanja 19, pri $b = 8$ pa bi bila že 21).

Težja različica: poizvedbe s_j so prepletene z dodajanji novih znanstvenikov k_i.

Še težja različica: posamezni k_i je podan relativno glede na rezultat prejšnje poizvedbe, torej niso vsi k_i znani naprej, ampak moramo res sproti računati rezultate poizvedb.

20. Transakcijski računi

(To je različica naloge, ki smo jo na tekmovanju 2019 uporabili kot tretjo v 3. skupini.) Že dlje časa nadzoruješ aktivnosti lokalne kriminalne organizacije in imaš bazo številk bančnih računov, s katerimi pogosto poslujejo. Večinoma so to računi članov organizacije, občasno pa za izboljšanje javnega ugleda kaj denarja nakažejo tudi določeni dobrodelni ustanovi. V bazi je n številk računov, pri čemer omenjeni dobrodelni ustanovi pripada prvi od teh računov. Številke računov so zaporedja 7 števk, pri čemer je zadnja števka kontrolna; zanjo velja, da je enaka ostanku po deljenju vsote ostalih števk z 10.

Prestregel si seznam m nakazil, ki jih bodo kriminalci izvedli naslednji dan. Vsako nakazilo je sestavljeno iz številke računa r (to je vedno eden od n računov iz baze) in zneska z ter pomeni, da bodo na račun r plačali z enot denarja. Na koncu seznama je še kontrolna vsota celotnega seznama; to je zaporedje 7 števk, v katerem (za vsak i) izračunamo i-to števko tako, da seštejemo i-te števke številk računov pri vseh n nakazilih v seznamu in obdržimo ostanek po deljenju tako dobljene vsote z 10.

Seznam hočemo spremeniti tako, da bo dobrodelna ustanova dobila čim več denarja, pri čemer pa lahko spreminjamo le številke računov pri posameznih nakazilih in še to le tako, da številko računa zamenjamo z eno od n številk računov iz baze. Ne smemo pa spreminjati zneskov nakazil ali kontrolne vsote na koncu seznama; slednja mora ostati tudi po naših spremembah še vedno veljavna. **Opiši postopek**, ki izračuna največji možni znesek, ki ga lahko dobrodelna organizacija prejme, če spremenimo seznam v skladu z opisanimi omejitvami.

REŠITVE NALOG ZA PRVO SKUPINO

1. Gesla

Nalogo lahko rešimo z dvema gnezdenima zankama. Zunanja zanka bo pregledala vse možne položaje pike; na začetku dodamo piko recimo na konec niza, nato pa po vsaki iteraciji te zanke premaknemo piko za en znak nazaj (tisti znak pa, ki je bil prej tik pred piko, se pri tem premakne tik za piko). Pazimo le na to, da po zadnji iteraciji, ko je pika že na začetku niza, ne poskušamo premakniti pike še bolj nazaj.

V notranji zanki pa bomo (pri vsakem položaju pike) poskušali na vse možne načine spremeniti po eno malo črko v veliko. Ker niz ni pretirano dolg in je verjetno večina znakov v njem črk, gremo lahko v tej drugi zanki kar po vseh znakih niza in pri vsakem najprej preverimo, ali je črka; če ni, gremo takoj na naslednji znak. Če pa je trenutni znak (mala) črka, jo spremenimo v veliko, niz izpišemo in spremenimo črko nazaj v malo, da povrnemo niz v prejšnje stanje.

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

void MoznaGesla(string geslo)
{
  geslo.push_back('.'); // Dodajmo piko na konec niza.
  // Z zanko preizkusimo vse možne položaje pike.
  for (int pika = geslo.length() − 1; pika >= 0; −−pika)
  {
    // Na vse možne načine spremenimo eno malo črko v veliko.
    for (char &c : geslo) if (isalpha(c))
    {
      c = toupper(c); // Spremenimo to črko v veliko.
      cout << geslo << endl;
      c = tolower(c); // Spremenimo črko nazaj v malo.
    }
    // Premaknimo piko eno mesto nazaj.
    if (pika > 0) swap(geslo[pika], geslo[pika − 1]);
  }
}
```

Oglejmo si še primer rešitve v pythonu. Tu niza ne moremo spreminjati, zato bomo morali delati kopije niza. V zunanji zanki bomo šli po vseh znakih niza; ne-črke preskočimo, če pa je trenutni znak črka, pripravimo kopijo niza, pri kateri to (malo) črko zamenjamo z ustrezno veliko črko. Nato izvedemo še notranjo zanko, ki gre po vseh možnih položajih pike in izpiše različico niza, v kateri je pika vrinjena na ta položaj.

```
def MoznaGesla(geslo):
  n = len(geslo)

  # Na vse možne načine spremenimo po eno črko v veliko.
  for velika in range(n):

    # Ne-črkovne znake preskočimo.
    if not geslo[velika].isalpha(): continue
```

```
    # Pripravimo kopijo gesla, v kateri je trenutna črka velika.
    geslo2 = geslo[:velika] + geslo[velika].upper() + geslo[velika + 1:]
    # Na vse možne načine dodajmo piko.
    for pika in range(n + 1):
        # Izpišimo različico gesla, v kateri je pika na indeksu „pika“.
        print(geslo2[:pika] + "." + geslo2[pika:])
```

2. Marsovci

Ker so opravila oštevilčena od 1 do 100, lahko uporabimo tabelo 100 elementov (ali 101, ker gredo indeksi od 0, nam pa bo lažje uporabljati indekse do 100), v kateri bomo šteli, koliko marsovcev se specializira za posamezno opravilo. Na začetku vse elemente te tabele inicializiramo na 0, nato pa v zanki beremo podatke o marsovcih in ustrezno povečujemo števce v tabeli. Na koncu se sprehodimo po celotni tabeli in poiščemo najmanjši in največji element, pri tem pa pazimo, da tiste z vrednostjo 0 preskočimo, saj ni nujno, da se vsa števila od 1 do 100 res pojavljajo v naših vhodnih podatkih. Če je razlika med največjim in najmanjšim elementom največ 1, so opravila približno enako zastopana, sicer pa ne.

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

int main()
{
    int zastopanost[101] = { };
    int m; cin >> m; // Preberimo število marsovcev.
    while (m-- > 0)
        // Preberimo opravila naslednjega marsovca.
        for (int i = 0; i < 5; i++)
        {
            int opravilo; cin >> opravilo; // Preberimo naslednje opravilo.
            ++zastopanost[opravilo];      // Povečajmo števec zastopanosti tega opravila.
        }
    // Poiščimo najmanjšo in največjo zastopanost.
    int min = -1, max = -1;
    for (int z : zastopanost)
    {
        if (z == 0) continue; // Ta številka opravila sploh ni v rabi.
        if (min < 0 || z < min) min = z;
        if (max < 0 || z > max) max = z;
    }
    // Izpišimo rezultat.
    cout << (max - min <= 1 ? "da" : "ne") << endl; return 0;
}
```

Zapišimo podobno rešitev še v pythonu. Za iskanje največje in najmanjše zastopanosti po vseh opravilih lahko uporabimo pythonovi funkciji min in max, če iz tabele prej pobrišemo ničle (ki predstavljajo neuporabljene številke opravil).

```
import sys

zastopanost = [0] * 101
m = int(sys.stdin.readline()) # Preberimo število marsovcev.
for i in range(m):
```

```
    # Preberimo opravila naslednjega marsovca.
    for opravilo in sys.stdin.readline().split():
        # Povečajmo števec zastopanosti tega opravila.
        zastopanost[int(opravilo)] += 1

# Pobrišimo ničle iz tabele, ker predstavljajo številke
# opravil, ki se v vhodnih podatkih sploh ne pojavljajo.
zastopanost = [z for z in zastopanost if z > 0]

# Izpišimo rezultat.
print("da" if max(zastopanost) - min(zastopanost) <= 1 else "ne")
```

3. Rekonstrukcija poti

Recimo, da pri branju vhoda preberemo podatek, da imamo direktorij s na globini g. Da dobimo polno pot do njega, moramo vzeti polno pot do njegovega starša (naddirektorija) in ji pritakniti poševnico / ter niz s. Koristno je torej, če imamo takrat to pot do starša že nekje pri roki. Ker pa ne moremo vnaprej vedeti, kakšen g bomo v naslednji vrstici dobili, moramo pravzaprav imeti pri roki poti do trenutnega direktorija in vseh njegovih prednikov. Hranili jih bomo v nekakšnem seznamu, ki ga uporabljamo bolj ali manj kot sklad, torej elemente dodajamo in brišemo le na koncu.

Ko potem preberemo ime direktorija s na globini g, moramo z vrha sklada pobrisati toliko elementov, da jih ostane le $g-1$; zadnji med temi je potem neposredni naddirektorij našega pravkar prebranega direktorija in iz polne poti do tega naddirektorija lahko izračunamo polno pot do pravkar prebranega direktorija ter jo dodamo na vrh sklada. (V praksi ni treba brisati toliko elementov, da jih ostane $g-1$, ampak jih lahko pustimo g in potem g-tega povozimo z novo potjo do pravkar prebranega direktorija).

Poseben primer nastopi, če je na skladu že zdaj manj kot $g-1$ elementov; takrat poti do pravkar prebranega direktorija ni mogoče določiti (kot npr. pri drugem primeru v besedilu naloge), zato lahko le še javimo napako in končamo z izvajanjem programa.

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

int main()
{
    // Sprva bo na skladu le prazen niz, ki predstavlja koren drevesa (na globini 0).
    vector<string> sklad = { "" };
    while (true)
    {
        // Preberimo naslednji direktorij.
        string s; int globina;
        cin >> s >> globina; if (! cin.good()) break;

        // Če je globina prevelika, sporočimo napako.
        if (globina > sklad.size()) { cout << "Napaka!" << endl; break; }

        // Če je globina za 1 večja od dosedanje, dodajmo na sklad nov element
        // s polno potjo do pravkar prebranega direktorija.
        else if (globina == sklad.size())
```

```
        sklad.push_back(sklad.back() + "/" + s);
      else {
        // Sicer pobrišimo toliko elementov, da bo zadnji tisti na indeksu „globina“.
        while (sklad.size() > globina + 1) sklad.pop_back();

        // Vanj vpišimo polno pot do pravkar prebranega direktorija.
        sklad[globina] = sklad[globina − 1] + "/" + s; }

      // Izpišimo polno pot do trenutnega direktorija.
      cout << sklad.back() << endl;
    }
    return 0;
}
```

Zapišimo to rešitev še v pythonu:

```
import sys
# Sprva bo na skladu le prazen niz, ki predstavlja koren drevesa (na globini 0).
sklad = [""]
for vrstica in sys.stdin:
  # Preberimo naslednji direktorij.
  s, globina = vrstica.split(); globina = int(globina)

  # Če je globina prevelika, sporočimo napako.
  if globina > len(sklad): print("Napaka!"); break

  # Če je globina za 1 večja od dosedanje, dodajmo na sklad nov element
  # s polno potjo do pravkar prebranega direktorija.
  elif globina == len(sklad):
    sklad.append(sklad[−1] + "/" + s)

  else:
    # Sicer pobrišimo toliko elementov, da bo zadnji tisti na indeksu „globina“.
    del sklad[globina + 1:]

    # Vanj vpišimo polno pot do pravkar prebranega direktorija.
    sklad[−1] = sklad[−2] + "/" + s

  # Izpišimo polno pot do trenutnega direktorija.
  print(sklad[−1])
```

4. Kako dobri so virusni testi?

V mislih lahko po obeh nizih hkrati pomikamo „okno“ širine n znakov. Pri tem bomo v neki spremenljivki (v spodnji rešitvi je to razlik) vzdrževali število mest znotraj okna, kjer se istoležna znaka nizov s in t razlikujeta. Ko se okno premakne za en znak naprej (v desno), tega števila ni težko popraviti: če je zadnji indeks v oknu zdaj recimo i, to pomeni, da je ta indeks zdaj na novo prišel v okno in moramo števec razlik povečati za 1, če se niza na tem indeksu razlikujeta (torej če sta $s[i]$ in $t[i]$ različna). In če je zadnji indeks v oknu i, okno pa je dolgo n znakov, to pomeni, da je prvi indeks v oknu $i-n+1$; indeks $i-n$ pa, ki je bil malo prej še v oknu, je zdaj na levi izpadel iz okna, tako da moramo števec razlik zmanjšati za 1, če je bila na tistem mestu med nizoma razlika (torej če sta bila $s[i-n]$ in $t[i-n]$ različna). Tako lahko po vsakem premiku okna izračunamo novo število razlik s samo konstantno mnogo operacijami, torej v $O(1)$ časa, neodvisno od širine okna n.

Po vsakem premiku okna moramo novo število razlik primerjati z največjim doslej in če je novo večje, si ga zapomnimo, skupaj z njim pa tudi i, pri katerem

smo ga dobili (v spodnji rešitvi je to spremenljivka najKje). Pomembno je, da najKje popravimo le, če je novo število razlik strogo večje od največjega doslej, ne pa, če je enako; s tem bomo zagotovili, da bomo med več enako dobrimi položaji okna vrnili najbolj levega, tako kot zahteva naloga. Paziti moramo še na to, da naloga zahteva indeks najbolj levega znaka v oknu, naša spremenljivka i pa je indeks najbolj desnega, tako da moramo na koncu še odšteti $n - 1$.

Ker smo imeli pri vsakem možnem položaju okna le $O(1)$ dela, je časovna zahtevnost tega postopka $O(d)$, če je d dolžina nizov s in t.

```
int Primerjava(const char *s, const char *t, int n)
{
  int najRazlik = −1, najKje = −1, razlik = 0;
  // Z oknom širine n znakov se pomikajmo v desno po obeh nizih in v spremenljivki
  // „razlik“ hranimo število mest (v oknu), kjer se niza razlikujeta.
  for (int i = 0; s[i]; ++i)
  {
    // Desni rob okna premaknimo na znak i.
    if (s[i] != t[i]) ++razlik;

    // Na levem robu zato znak i − n izpade iz okna.
    if (i >= n && s[i − n] != t[i − n]) −−razlik;

    // Najboljšo rešitev si zapomnimo.
    if (razlik > najRazlik) najRazlik = razlik, najKje = i;
  }
  // Vrnimo rezultat, vendar indeks na levem koncu okna, ne na desnem.
  return najKje − n + 1;
}
```

Zapišimo to rešitev še v pythonu:

```
def Primerjava(s, t, n):
  najRazlik = −1; najKje = −1; razlik = 0

  # Z oknom širine n znakov se pomikajmo v desno po obeh nizih in v spremenljivki
  # „razlik“ hranimo število mest (v oknu), kjer se niza razlikujeta.
  for i in range(len(s)):

    # Desni rob okna premaknimo na znak i.
    if s[i] != t[i]: razlik += 1

    # Na levem robu zato znak i − n izpade iz okna.
    if i >= n and s[i − n] != t[i − n]: razlik −= 1

    # Najboljšo rešitev si zapomnimo.
    if razlik > najRazlik: najRazlik = razlik; najKje = i

  # Vrnimo rezultat, vendar indeks na levem koncu okna, ne na desnem.
  return najKje − n + 1
```

5. Zlaganje loncev

Največji lonec ne more biti drugje kot na dnu svojega sklada; imeti moramo torej vsaj en sklad s takšnim premerom, kot ga ima največji lonec. Če zdaj pogledamo drugi največji lonec, ga lahko položimo v prvega in tako nadaljujemo isti sklad; podobno položimo tretji največji lonec v drugega in tako naprej. Edino, kar nam lahko pri tem postopku povzroči težave, je, če naletimo na dva ali več loncev z enakim premerom. Ker taki lonci ne gredo eden v drugega, lahko damo na prvi sklad

le enega od njih, za ostale pa bomo morali načeti nove sklade (za vsak tak lonec po enega). Pri naslednjem manjšem polmeru lahko lonec spet damo v prvi sklad in tako naprej; sčasoma mogoče spet naletimo na več loncev z enakim premerom in jih damo po vsakega v en sklad; če imamo skladov premalo, pa za preostale take lonce začnemo nove sklade. Tako nadaljujemo, dokler ne razporedimo vseh loncev.

Da bo preglednejе, zapišimo ta postopek še s psevdokodo. V spremenljivki s bomo hranili število skladov, v v pa vsoto polmerov najnižjih loncev v njih. Pri pregledovanju loncev bo p polmer prejšnjega lonca, t pa število doslej pregledanih loncev s tem polmerom.

```
v := 0; s := 0; p := −1; t := 0;
pregleduj lonce padajoče po polmeru:
   naj bo r polmer trenutnega lonca;
   if r ≠ p then p := r, t := 1
   else t := t + 1;
   (* To je že t-ti lonec s polmerom r; potrebujemo torej vsaj t skladov.
      Če jih še nimamo toliko, začnimo nov sklad. *)
   if t > s then s := s + 1, v := v + r;
```

Na koncu tega postopka sta s in v rezultata, po katerih sprašuje naloga.

Vidimo lahko, da odpre ta postopek t skladov le, če vidi t loncev z enakim polmerom; na koncu bo torej skladov toliko, kolikor je največ loncev z enakim polmerom, tako da je število skladov res minimalno. Da je minimalna tudi vsota njihovih polmerov, pa se lahko prepričamo takole. Naš postopek odpira sklade po padajočem (oz. natančneje: nenaraščajočem) polmeru: vsak naslednji sklad ima na dnu kvečjemu tako velik lonec kot prejšnji sklad; in t-ti sklad odpre pri največjem takem polmeru r, pri katerem imamo vsaj t loncev enakega polmera. Če bi bil polmer t-tega največjega sklada manjši od tega r, bi bili polmeri vseh nadaljnjih skladov tudi manjši od r, torej bi obstajalo kvečjemu $t-1$ skladov s polmerom vsaj r, to pa je premalo za naših (vsaj) t loncev s polmerom r. Tako torej vidimo, da če bi polmer kateregakoli sklada zmanjšali, bi rešitev postala neveljavna, torej naš postopek res najde najmanjšo možno vsoto polmerov.

REŠITVE NALOG ZA DRUGO SKUPINO

1. Sredinec

Ker so pri tej nalogi podane višine v centimetrih in ker učenci niso večji od dveh metrov, je možnih razmeroma malo višin — to so cela števila od 1 do 200. Četudi je učencev na milijone, imajo lahko največ 200 različnih višin; vrsta, v katero se učenci razporejajo v telovadnici, ima torej vedno takšno obliko: najprej nekaj učencev z višino 1, nato nekaj učencev z višino 2, ... in končno nekaj učencev z višino 200. (Pri vsakem od teh „nekaj" je seveda mogoče tudi, da ni nobenega s tisto višino.) Učencev z enako višino nam ni treba nikakor ločiti med seboj, saj nas zanima vedno le to, kako visok je srednji učenec v vrsti, ne pa, kdo točno je ta srednji učenec. Vrste nam torej ni treba predstaviti s seznamom višin, ki bi vseboval po en element za vsakega učenca, pač pa je dovolj že tabela, ki za vsako možno višino od 1 do 200 pove, koliko učencev s to višino je trenutno v telovadnici. Lepo pri tem je, da ko vstopi nov učenec, moramo le povečati en element te tabele za 1, kar je veliko ceneje, kot če bi hoteli vzdrževati urejen seznam višin vseh učencev in vrivati novega učenca na pravo mesto v tem seznamu.

Višino srednjega, torej $\lceil n/2 \rceil$-tega učenca, bi lahko zdaj določili tako, da bi šli v zanki po višinah od 1 naprej in seštevali število učencev posamezne višine. Pri tisti višini, kjer ta vsota doseže ali preseže $\lceil n/2 \rceil$, vemo, da je v skupini učencev s to višino tudi srednji ($\lceil n/2 \rceil$-ti) učenec in moramo to višino izpisati.

Toda ko je v telovadnici že veliko učencev in jih ima tudi po več enako višino kot srednji učenec, se lahko pogosto zgodi, da ostane višina srednjega učenca nespremenjena tudi po prihodu novega učenca. Na primer: če imamo učence $[10, 20, 20, 20, 30]$, je višina srednjega učenca 20; in če vstopi zdaj en nov učenec, bo višina srednjega še vedno 20 ne glede na višino novega učenca.

Zato je koristno, če višine srednjega ne računamo vsakič znova z zanko po višinah od 1 naprej, ampak le pogledamo, če je treba dosedanjo višino srednjega kaj popraviti. V ta namen bomo poleg višine srednjega vzdrževali še skupno število učencev, ki so manjši od srednjega; recimo, da je višina srednjega učenca m, da ima táko višino v_m učencev, manjših od te višine pa je $v_{<m}$ učencev. Ko pride nov učenec, za začetek pustimo m pri miru in le povečamo v_m ali $v_{<m}$ za 1, če je novi učenec visok m ali $< m$; za 1 povečamo tudi števec vseh učencev, torej n; nato pa preverimo, ali ni zdaj slučajno $v_{<m} \geq \lceil n/2 \rceil$ — če je, to pomeni, da je učencev, manjših od m, že preveč, zato je m že previsok, da bi bila to lahko višina srednjega učenca; tedaj m zmanjšujmo po 1 (in ustrezno zmanjšujmo $v_{<m}$), dokler ta pogoj ni izpolnjen. Podobno preverimo tudi, ali ni zdaj slučajno $v_{<m} + v_m < \lceil n/2 \rceil$ — če je, to pomeni, da je zdaj preveč učencev večjih od m in je višina m prenizka, da bi bila to višina srednjega učenca; tedaj m povečujmo po 1 (in ustrezno povečujmo $v_{<m}$), dokler ta pogoj ni izpolnjen. Tako bomo v večini primerov dobili primerno novo vrednost m že brez popravkov ali pa mogoče z enim povečanjem ali zmanjšanjem za 1.[4]

[4]Mogoče pa je sestaviti patološke primere, kjer ta izboljšava nič ne pomaga; na primer, če dobimo zaporedje učencev z višinami 200, 1, 200, 1, 200, 1 in tako naprej, nam tudi višina srednjega učenca enako preskakuje z 200 na 1 in nazaj, zato mora naša zanka, ki popravlja m, vsakič pravzaprav iti po vseh možnih višinah. Če bi se hoteli izogniti tej težavi, bi morali imeti način, da preskočimo višine, ki jih nima noben učenec; namesto tabele bi lahko uporabili kakšno (primerno uravnoteženo) drevo, npr. rdeče-črno, ali pa bi vzdrževali par kopic, kar si

Oglejmo si še implementacijo te rešitve v C++. Višine učencev bomo brali s standardnega vhoda in po vsakem prebranem učencu izpisali na standardni izhod višino srednjega učenca:

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

int main()
{
  int stZVisino[201] = { }; // št. učencev s posamezno višino
  int mediana = 0;          // višina srednjega učenca
  int stPodMediano = 0;     // št. učencev z višino < mediana
  int n = 0;                // število doslej prebranih učencev

  while (true)
  {
    // Preberimo višino naslednjega učenca.
    int visina; cin >> visina;
    if (! cin.good()) break;
    // Povečajmo števec vseh učencev in učencev te višine.
    ++n; ++stZVisino[visina];
    // Če je manjši od mediane, povečajmo tudi števec takih.
    if (visina < mediana) ++stPodMediano;
    // Če je mediana zdaj previsoka, jo zmanjšajmo.
    while (stPodMediano >= (n + 1) / 2)
      stPodMediano -= stZVisino[--mediana];
    // Če pa je mediana zdaj prenizka, jo povečajmo.
    while (stPodMediano + stZVisino[mediana] < (n + 1) / 2)
      stPodMediano += stZVisino[mediana++];
    // Izpišimo višino srednjega učenca.
    cout << mediana << endl;
  }
  return 0;
}
```

Časovna zahtevnost te rešitve je $O(n)$ za obdelavo zaporedja n učencev, saj imamo z vsakim novim le konstantno mnogo dela, neodvisno od n; bolj natančno pa bi morali reči, da je zahtevnost v najslabšem primeru $O(n \cdot V)$, kjer je V število vseh možnih višin — v našem primeru 200.

Razmislimo zdaj še o težji različici naloge, ki jo omenja opomba pod črto na koncu besedila naloge: tu višine niso nujno le cela števila od 1 do 200, morda niti niso cela števila; torej je število vseh možnih višin V potencialno zelo veliko, večje od števila učencev n. Zato bi tabela V elementov (za vse možne višine) zasedla preveč prostora; bolje je hraniti le podatke o višinah doslej prispelih učencev. Poleg tega si zdaj tudi ne moremo privoščiti, da bi pri vsakem prihodu novega učenca porabili za določitev sredinca po $O(V)$ ali $O(n)$ časa.

Učence lahko v mislih razdelimo na dve skupini; prvo naj tvori najmanjših $\lceil n/2 \rceil$ učencev, drugo pa vsi preostali. Sredinec je potem vedno največji učenec v prvi skupini. Ko pride nov učenec, za začetek poglejmo, ali je večji od dosedanjega sredinca; če je, ga dajmo v drugo skupino, sicer pa v prvo. Potem je težava lahko

bomo ogledali v nadaljevanju naše rešitve.

le še v tem, da skupini zdaj morda nista več primerno veliki. Ko se število učencev poveča z n na $n+1$, se velikost prve skupine spremeni z $\lceil n/2 \rceil$ na $\lceil (n+1)/2 \rceil$. Pri sodem n to pomeni, da mora biti prva skupina po novem enako velika kot prej; če smo torej novega učenca dali v prvo skupino, je ta zdaj prevelika in moramo enega učenca (največjega) preseliti iz nje v drugo skupino. Pri lihem n pa mora biti prva skupina po novem za 1 večja kot prej; če smo torej novega učenca dali v drugo skupino, je prva zdaj premajhna in moramo najmanjšega učenca iz druge skupine preseliti v prvo skupino. Po tem popravku je sredinec spet tisti učenec, ki je zdaj največji v prvi skupini.

Koristno je torej imeti podatkovno strukturo, pri kateri bomo lahko poceni dodajali ali brisali elemente iz skupine in imeli pri roki največjega oz. najmanjšega med njimi. Zelo primerna struktura za to je kopica (*heap*), šlo pa bi tudi s kakšnim primerno uravnoteženim binarnim iskalnim drevesom, npr. rdeče-črnim. Prvo skupino bomo torej predstavili s kopico, pri kateri je v korenu največji element (ta je naš sredinec), drugo pa s táko, pri kateri je v korenu najmanjši element. Tako bomo imeli vedno pri roki tistega, ki ga bo treba seliti iz ene skupine v drugo; brisanje elementa iz ene skupine in dodajanje v drugo pa nam bo vzelo $O(\log n)$ časa.[5] Oglejmo si implementacijo te rešitve v C++, kjer je kopica na voljo v standardni knjižnici kot razred priority_queue:

```
#include <iostream>
#include <queue>
#include <vector>
#include <functional>
using namespace std;

int main()
{
  priority_queue<int> majhni;                                  // prva skupina
  priority_queue<int, vector<int>, greater<int>> veliki;  // druga skupina
  int n = 0;  // število doslej prebranih učencev
  int s = 0;  // indeks sredinca (= zahtevana velikost prve skupine)
  while (true)
  {
    // Preberimo višino naslednjega učenca.
    int visina; cin >> visina;
    if (! cin.good()) break;

    // Povečajmo števec vseh učencev in indeks sredinca.
    ++n; s = (n + 1) / 2;

    // Dodajmo novega v eno od skupin.
    if (majhni.empty() || visina <= majhni.top()) majhni.push(visina);
    else veliki.push(visina);

    // Popravimo velikost skupin.
    if (majhni.size() > s) { veliki.push(majhni.top()); majhni.pop(); }
    else if (majhni.size() < s) { majhni.push(veliki.top()); veliki.pop(); }

    // Največji v prvi skupini je zdaj novi sredinec.
    cout << majhni.top() << endl;
  }
  return 0;
```

[5] Podoben prijem z dvema kopicama smo videli že leta 2020 pri 4. nalogi v tretji skupini (gl. str. 75–76 v *Biltenu* 2020).

```
}
```

2. Svetilka

Razmislimo najprej, kakšne globalne spremenljivke bomo potrebovali. Funkcija Tiktak mora vedeti, ali je tipka pritisnjena in kako dolgo, da bo lahko po treh sekundah držanja tipke ugasnila luč. V spodnji rešitvi imamo v ta namen spremenljivki tipkaPritisnjena in casPritiska (ki šteje čas pritiska v desetinkah sekunde), šlo pa bi tudi z eno samo spremenljivko (pri čemer bi npr. vrednost casPritiska $== -1$ pomenila, da tipka sploh ni pritisnjena). Poleg tega moramo poznati tudi trenutni način delovanja, saj je od tega odvisno, kaj se zgodi ob naslednjem pritisku in ali moramo skrbeti za utripanje. Pri utripanju pa moramo vedeti še, čez koliko časa naj se luč spet prižge; spodnja rešitev ima za to spremenljivko casDoUtripa.

```
typedef enum { Ugasnjena, Sveti, Utripa } Nacin;
Nacin nacin = Ugasnjena;
bool tipkaPritisnjena = false;
int casPritiska, casDoUtripa;
```

Oglejmo si zdaj funkcijo Tipka, ki je enostavnejša. Novo stanje tipke si zapomnimo v tipkaPritisnjena; če je tipka zdaj spuščena, je to tudi vse, sicer pa določimo novi način delovanja luči: če je prej svetila stalno, mora zdaj utripati, sicer pa mora zdaj svetiti. V slednjem primeru lahko luč takoj tudi prižgemo; pri utripanju pa bi bila škoda, če bi jo zdaj prižgali in potem pri naslednjem klicu funkcije Tiktak ugasnili, saj lahko do takrat mine manj kot desetinka sekunde. Namesto tega bomo raje postavili casDoUtripa na 1 in tako zagotovili, da bo luč prižgala funkcija Tiktak ob naslednjem klicu (in jo potem še en klic kasneje spet ugasnila; tako bo luč gotovo gorela eno desetinko sekunde).

```
void Tipka(bool pritisnjena)
{
  // Zapomnimo si novo stanje tipke.
  tipkaPritisnjena = pritisnjena;
  if (! pritisnjena) return;

  // Ob pritisku začnemo meriti čas pritiska.
  casPritiska = 0;

  // Preklopimo na novo stanje.
  nacin = (nacin == Sveti) ? Utripa : Sveti;

  // Pri preklopu na utripanje bomo prižgali luč v naslednji
  // desetinki namesto takoj, da bomo lažje odmerili čas.
  Luc(nacin == Sveti);
  casDoUtripa = 1;
}
```

Funkcija Tiktak mora skrbeti za utripanje luči in za izklop po treh sekundah držanja na tipko. Za to slednje poskrbimo s števcem casPritiska, ki ga ob vsakem klicu povečamo, ko pa doseže 31, luč ugasnemo. Ker ga je Tipka postavila na 0, ko je uporabnik pritisnil tipko, in ker ne vemo točno, kje v času med dvema klicema funkcije Tiktak je prišel klic Tipka, to pomeni, da se bo luč ugasnila po vsaj treh sekundah (gotovo pa manj kot 3,1 sekunde) držanja na tipko.

Za utripanje poskrbimo tako, da zmanjšujemo števec casDoUtripa; ko pade na 0, prižgemo luč in postavimo števec na 10; ko pa pade števec na 9 (torej eno desetinko sekunde po tistem, ko smo luč prižgali in postavili števec na 10) luč spet ugasnemo. Tako bo luč res gorela eno desetinko sekunde in bo potem devet desetink ugasnjena.

```
void Tiktak()
{
  if (tipkaPritisnjena && casPritiska <= 30)
    // Povečajmo števec, ki meri čas pritiska tipke.
    if (++casPritiska > 30) {
      // Po treh sekundah luč ugasnemo.
      Luc(false); nacin = Ugasnjena; }
  // Poskrbimo za utripanje.
  if (nacin == Utripa)
    // Zmanjšajmo čas do utripa za 1; ko pade na 0, luč prižgemo.
    if (−−casDoUtripa == 0) { Luc(true); casDoUtripa = 10; }
    // Ko je do naslednjega utripa še 9 desetink sekunde, luč spet ugasnemo.
    else if (casDoUtripa == 9) Luc(false);
}
```

3. Pletenje puloverja

Recimo, da je shema široka w stolpcev in visoka h vrstic; naj bo $s(x, y)$ znak na preseku x-tega stolpca in y-te vrstice.

Recimo zdaj, da je v shemi prisoten vzorec velikosti $w_p \times h_p$, ki se lepo zaključi na robovih sheme. Iz tega sledi, da je prvih h_p vrstic sheme (ki tvorijo prvo vrsto pojavitev vzorca) enakih naslednjim h_p vrsticam (ki tvorijo drugo vrsto pojavitev vzorca) in potem spet naslednjim h_p vrsticam in tako naprej. Z drugimi besedami, velja torej $s(x, y) = s(x, y-h_p)$ za vse x in y (natančneje povedano: za vse $1 \le x \le w$ in $h_p < y \le h$; tovrstnih pogojev v nadaljevanju ne bomo posebej pisali, jih pa imejmo v mislih). Poleg tega vidimo tudi, da je višina sheme h večkratnik višine vzorca h_p (torej da h_p deli h), saj se sicer vzorec na spodnjem robu ne bi lepo zaključil, pač pa bi bila zadnja vrsta pojavitev vzorca delno odrezana. Podobno lahko razmišljamo tudi za stolpce, kjer ugotovimo, da velja $s(x, y) = s(x - w_p, y)$ za vse x in y ter da w_p deli w.

Kaj pa obratno? Recimo, da v naši shemi pri nekem w_p, ki deli w, velja $s(x, y) = s(x - w_p, y)$ (za vse x in y) in da pri nekem h_p, ki deli h, velja $s(x, y) = s(x, y - h_p)$ (za vse x in y). Iz prve od teh dveh predpostavk vidimo, da se vsebina pravokotnika $w_p \times h_p$ v zgornjem levem kotu sheme potem spet ponavlja, če jo zamikamo po w_p enot desno; in ker w_p deli w, bomo s ponavljanjem tega pravokotnika ravno zapolnili prvih h_p vrstic mreže po celi širini. Druga predpostavka pa nam potem pove, da se nam vsebina teh prvih h_p vrstic v nadaljevanju ponavlja v vsakih naslednjih h_p vrsticah in (ker h_p deli h) tako sčasoma točno zapolni celo shemo. Tako torej vidimo, da je v shemi prisoten vzorec velikosti $w_p \times h_p$.

Če oba prejšnja odstavka združimo, lahko zaključimo, da je v shemi prisoten vzorec $w_p \times h_p$ natanko tedaj, ko w_p deli w, h_p deli h in ko za vse primerne x in y velja $s(x, y) = s(x - w_p, y)$ in $s(x, y) = s(x, y - h_p)$. Za to zadnjo skupino pogojev pa vidimo, da se eni nanašajo samo na w_p, eni pa samo na h_p. Tako vidimo, da nam pri iskanju vzorcev ni treba preverjati para (w_p, h_p) skupaj, ampak lahko iščemo

primerne w_p posebej in primerne h_p posebej. Najmanjši vzorec — in to je tisti, po katerem nas sprašuje naloga — bomo torej dobili tako, da bomo vzeli najmanjši primerni w_p in najmanjši primerni h_p. Tako se tudi ne bo moglo zgoditi, da bi obstajalo več enako dobrih rešitev; vedno je en sam vzorec najmanjši.

Pojdimo torej v zanki po naraščajočih w_p in pri vsakem najprej preverimo, ali deli w; če je to res, preglejmo, če se vsebina sheme ponavlja na vsakih w_p vrstic. Najmanjši w_p, pri katerem se to izide, je potem širina našega osnovnega vzorca (če ne prej, bo ta pogoj gotovo izpolnjen pri $w_p = w$). Nato podobno naredimo še za h_p, kjer preverjamo, če h_p deli h in če se vsebina sheme ponavlja na vsakih h_p stolpcev.

Oglejmo si implementacijo te rešitve v C++. Ker je preverjanje po vrsticah in po stolpcih zelo podobno, smo si pomagali z zanko z dvema iteracijama; v prvi iščemo najmanjši w_p, v drugi pa najmanjši h_p, vmes pa v mislih zamenjamo vrstice in stolpce, da lahko potem obakrat uporabimo isto kodo.

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

void OsnovniVzorec(const vector<string>& shema)
{
  int w = shema[0].length(), h = shema.size(), w0, h0;
  // Pri smer == 0 iščemo širino osnovnega vzorca, pri smer == 1 pa višino.
  for (int smer = 0; smer < 2; ++smer)
  {
    // Naslednji podprogram prebere en znak sheme.
    auto Znak = [&shema, smer] (int x, int y) { return smer ? shema[x][y] : shema[y][x]; };

    // Če ne bomo našli ožjega, bo osnovni vzorec pokrival celo širino sheme.
    w0 = w;

    // Preizkusimo ožje širine, seveda le take, ki delijo širino sheme.
    for (int d = 1; d < w; ++d) if (w % d == 0)
    {
      // Preverimo, če se vzorec s to širino ponavlja po celi shemi.
      bool ok = true;
      for (int y = 0; y < h && ok; ++y) for (int x = d; x < w; ++x)
        if (Znak(x, y) != Znak(x - d, y)) { ok = false; break; }

      // Če se, smo našli širino osnovnega vzorca.
      if (ok) { w0 = d; break; }
    }

    // Obrnimo osi, da bomo v naslednji iteraciji našli še višino.
    swap(w, h); swap(w0, h0);
  }
  cout << w0 << ' ' << h0 << endl; // Izpišimo rezultate.
}
```

To rešitev bi se dalo še izboljšati s kakšnimi hevristikami, ki bi nam pomagale čim prej in čim ceneje prepoznati neobetavne w_p ali h_p. Recimo, da za vsak stolpec izračunamo nekakšno kontrolno vsoto ali zgoščevalno kodo: $k[1], k[2], \ldots, k[w]$ (če drugega ne, lahko preštejemo ničle v stolpcu in to vzamemo za kontrolno vsoto). Ko nas kasneje zanima, ali se shema ponavlja na vsakih w_p stolpcev, bi lahko za začetek preverili, ali se tako ponavljajo tudi te kontrolne vsote, torej ali velja $k[x] = k[x - w_p]$

za vse x; šele če se to izide, je smiselno preverjati vse znake sheme, kot to počne gornji podprogram. Lahko gremo še korak naprej: če se kontrolne vsote res ponavljajo na vsakih w_p stolpcev, to pomeni, da se vsaka od vsot $k[1], \ldots, k[w_p]$ pojavi (w/w_p)-krat. Če bi torej za vsako različno kodo prešteli, kolikokrat se pojavi, bi morala biti vsa ta števila pojavitev večkratniki vrednosti w/w_p; ali še drugače, w_p je lahko kandidat za širino vzorca le, če w/w_p deli vsa števila pojavitev kontrolnih vsot stolpcev; to pa pomeni, da mora deliti njihov najmanjši skupni delitelj; slednjemu recimo D. Tega lahko izračunamo na začetku, še preden se začnemo ukvarjati s posameznimi w_p, in potem pri vsakem w_p najprej preverimo, če w_p deli w in če w/w_p deli D; če se to izide, preverimo, ali se kontrolne vsote $k[1], \ldots, k[w]$ ponavljajo na vsakih w_p stolpcev; in šele nato preverimo, ali se tudi vsebina stolpcev ponavlja na vsakih w_p stolpcev. Podobno lahko seveda naredimo tudi pri vrsticah.

Še ena možna izboljšava je naslednja: recimo, da smo najmanjši primerni w_p že našli in da zdaj iščemo najmanjši primerni h_p. Ko moramo pri nekem kandidatu za h_p preveriti, ali res povsod velja $s(x,y) = s(x, y-h_p)$, nam tega zdaj ni treba preverjati za vse x od 1 do w, ampak je dovolj že do $x = w_p$; če je pogoj $s(x,y) = s(x, y-h_p)$ veljal povsod v prvih w_p stolpcih, bo veljal tudi povsod desno od tam, saj se odtlej stolpci le še periodično ponavljajo.

Boljša rešitev s pomočjo Knuth-Morris-Prattovega algoritma. O nizu s dolžine n bomo rekli, da je *periodičen* s periodo t, če je oblike $s = t^k$ za neki $k > 1$; in da je *semiperiodičen* s periodo t, če je oblike $s = t^k u$ za $k \geq 1$ in je pri tem u neprazen prefiks t-ja.

Niz je lahko (semi)periodičen pri več različnih periodah; na primer, niz *abababa* je semiperiodičen s periodama *ab* in *abab*; niz *abababab* pa je periodičen s periodama *ab* in *abab*.

Najkrajši periodi, s katero je s periodičen, recimo *osnovna perioda* tega niza. Prepričajmo se, da so dolžine vseh ostalih period večkratniki dolžine osnovne periode. Recimo, da je $|s| = n$ in da je s periodičen s periodama dolžine p in q; torej $s = t^k = u^\ell$ za $|t| = p$, $k = n/p$, $|u| = q$, $\ell = n/q$. Naj bo $r = \gcd(p, q)$; števili $P = p/r$ in $Q = q/r$ sta si potemtakem tuji. Naj bo $N = n/r$. Potem je $N = n/r = (pk)/r = Pk$ in hkrati $N = n/r = (ql)/r = Q\ell$; torej $Pk = Q\ell$; leva stran je večkratnik P, torej mora biti desna tudi; ker pa sta si P in Q tuja, je lahko desna stran večkratnik P-ja le tako, da je ℓ večkratnik P-ja; zato pa je $N = Q\ell$ tudi večkratnik produkta PQ.

Razdelimo vse tri nize na kose dolžine r: $s = s_0 s_1 \ldots s_{N-1}$, $t = t_0 t_1 \ldots t_{P-1}$ in $u = u_0 u_1 \ldots u_{Q-1}$. V enakosti $s = t^k = u^\ell$ primerjajmo zdaj istoležne kose: $s_i = t_{i \bmod P} = u_{i \bmod Q}$. Ko gre i od 0 do $PQ - 1$ (in spomnimo se, da je N večkratnik PQ), dobimo za $(i \bmod P, i \bmod Q)$ vse možne pare ostankov $\{0, \ldots, P-1\} \times \{0, \ldots, Q-1\}$; kajti če bi imela dva različna i-ja enak par ostankov, bi to pomenilo, da se hkrati razlikujeta za neki večkratnik P-ja in tudi za neki večkratnik Q-ja, kar pa je (ker sta si P in Q tuja) mogoče le, če se razlikujeta za neki večkratnik PQ-ja, to pa se ne moreta, če sta oba z območja od 0 do $PQ - 1$. Pri nekaterih i torej dobimo kombinacije $(0,0), (0,1), ..., (0, Q-1)$, ki nam povedo, da je t_0 enak nizom $u_0, u_1, \ldots, u_{Q-1}$; pri nekaterih drugih i pa dobimo kombinacije $(0,0), (1,0), ..., (P-1, 0)$, ki nam povedo, da je u_0 enak nizom $t_0, t_1, \ldots, t_{P-1}$. Kosi $t_0, \ldots, t_{P-1}, u_0, \ldots, u_{Q-1}$ so si torej vsi enaki, vsi so en in isti niz; recimo mu

v; torej je $t = v^P$, $u = v^Q$ in $s = v^N$. Torej je s periodičen s periodo dolžine $r = \gcd(p, q) \leq p$.

Če zdaj v tem razmisleku za p vzamemo osnovno periodo, ki je najkrajša med vsemi periodami, je nemogoče, da bi bila perioda r še krajša; v neenakosti $r \leq p$ mora torej veljati stroga enakost, torej $\gcd(p, q) = p$, to pa je mogoče le tako, da je q večkratnik p. Torej je dolžina vsake daljše periode res večkratnik dolžine osnovne periode. □

Spomnimo se, da pri naši nalogi iščemo najmanjši tak w_p, za katerega se stolpci naše sheme ponavljajo s periodo dolžine w_p. (Za h_p je stvar podobna, le da moramo v mislih zamenjati stolpce in vrstice.) Pri takem w_p je vsaka vrstica sheme periodična s periodo dolžine w_p. Če je osnovna perioda vrstice y recimo dolga p_y, to pomeni, da mora biti w_p večkratnik p_y; ker to velja za vse y, mora biti w_p skupni večkratnik vseh števil $p_1, \ldots, p_h$. Ker nas zanima najmanjši primerni w_p, bomo morali vzeti najmanjši skupni večkratnik, to je $w_p = \operatorname{lcm}(p_1, \ldots, p_h)$.

Naloge torej ne bo težko rešiti, če bomo znali učinkovito poiskati osnovno periodo vsake vrstice. Oglejmo si zdaj, kako lahko to naredimo.

V nadaljevanju nam bo prišlo prav naslednje opažanje: če je neki niz u hkrati prefiks in sufiks niza s (dolžine n) in je $|u| \geq n/2$, potem je s (semi)periodičen s periodo dolžine $p := n - |u|$. (Če je p delitelj n-ja, bo s periodičen, sicer pa semiperiodičen.)

Prepričajmo se, da je to res. Ker se s začne na u, ga lahko zapišemo kot $s = ut$; in ker se konča na u, ga lahko zapišemo kot $s = vu$. Pri tem je $|t| = |v| = p$. Ker je po predpostavki $|u| \geq n/2$, mora biti $p = n - |u| \leq n/2$, torej sta t in v krajša od u-ja (ali kvečjemu enako dolga kot u). Naj bo $r = n \bmod p$, tako da je $n = kp + r$ za neki $k \geq 2$ (to, da je $k \geq 2$, sledi iz dejstva, da je $p \leq n/2$). Razdelimo u v mislih na kose, dolge po p znakov, le zadnji naj ima le r znakov: $u = u_1 u_2 \cdots u_{k-1} w$. Zdaj imamo

$$\begin{array}{rccccc} s = ut = & u_1 & u_2 & \cdots & u_{k-1} & wt \\ s = vu = & v & u_1 & \cdots & u_{k-2} & u_{k-1}w \end{array}$$

Če v obeh vrsticah primerjamo istoležne (in enako dolge) kose niza s, vidimo, da velja $v = u_1$, $u_1 = u_2$, ..., $u_{k-2} = u_{k-1}$; na koncu pa še $wt = u_{k-1}w$. Tako je torej $v = u_1 = u_2 = \ldots = u_{k-1}$ in zato $u = v^{k-1}w$; iz tistega na koncu pa dobimo $wt = u_{k-1}w = vw$. Ker je $|w| = r$ in $|v| = |t| = p$, je torej w krajši od v in t (ali kvečjemu enako dolg kot onadva); enakost $wt = vw$ nam torej pove, da se v začne na w in da se t konča na w, torej $v = w\hat{v}$ in $t = \hat{t}w$; ko to nesemo v $wt = vw$, dobimo $w\hat{t}w = w\hat{v}w$, torej $\hat{t} = \hat{v}$; recimo temu nizu z, pa dobimo: $v = wz$, $t = zw$, $s = vu = vv^{k-1}w = v^k w = (wz)^k w$. Torej je s res (semi)periodičen s periodo dolžine $|wz| = |v| = n - |u| = p$, pri čemer od zadnje kopije periode wz nastopi le prvih $|w| = r = n \bmod p$ znakov (če se deljenje izide in je $r = 0$, je s periodičen s to periodo, sicer pa le semiperiodičen). □

Videli smo torej, da če se v s neki (dovolj dolg) prefiks u pojavlja hkrati tudi kot sufiks in če je $p := n - |u|$ delitelj n, potem je s periodičen s periodo dolžine p; velja pa tudi obratno: če je $s = t^k$, je niz $u = t^{k-1}$ hkrati prefiks in sufiks s-ja. Če bi torej pregledali vse take u, ki so hkrati prefiksi in sufiksi s-ja in za katere je $n - |u|$ delitelj n-ja, bi s tem pregledali tudi vse periode s-ja. Nas bo seveda zanimala osnovna, to je najkrajša perioda; to je tista z najmanjšim $n - |u|$, torej z najdaljšim u.

Imejmo torej niz s dolžine n in naj bo $f(k)$ indeks, na katerem se v s začne druga pojavitev podniza $s[1..k]$ (prva se očitno začne na indeksu 1). Če take druge pojavitve sploh ni, si mislimo $f(k) = n+1$ ali kaj podobnega. Vrednosti $f(k)$ za vse k od 1 do n lahko izračunamo v $O(n)$ časa s postopkom, ki je del znanega Knuth-Morris-Prattovega algoritma za iskanje podnizov v nizih, zato se s podrobnostmi tega tu ne bomo ukvarjali. Oglejmo pa si, kako si lahko s funkcijo f pomagamo pri iskanju osnovne periode niza s.

Recimo, da je s periodičen z osnovno periodo t, torej je $s = t^k$ za neki $k > 1$ (in t je najkrajši niz, pri katerem tak k obstaja). Označimo dolžino periode s $p = |t|$. Potem se niz t^{k-1} (to je prefiks s-ja, dolg $n - p$ znakov) pojavi v s tako na začetku (indeks 1) kot še na indeksu $p + 1$. Takrat bo torej $f(n - p) \leq p + 1$. Če velja tu enakost, se pravi $f(n-p) = p+1$, lahko iz tega zaključimo, da se prefiks dolžine $n-p$ pojavlja tudi kot sufiks s-ja in da je s zato periodičen s periodo p (dokaz tega smo si ogledali malo prej). Vprašanje pa je, ali se lahko zgodi, da velja stroga neenakost in nas ovira pri tem zaključku (ker iz vrednosti $f(n-p)$ tedaj ne bomo mogli vedeti, ali se t^{k-1} pojavlja tudi na koncu niza s ali ne); torej: ali je lahko $f(n - p) < p + 1$? Ali se torej lahko t^{k-1} pojavi v s še nekje vmes med tisto pojavitvijo na začetku niza in tisto p znakov kasneje?

Pa recimo, da bi se to res zgodilo, torej da je $f(n - p) = r + 1 < p + 1$. Torej je s oblike $s = ut^{k-1}v$, pri čemer je $|u| = r$, niz v pa mora biti potem dolžine $n - r - (k - 1)p = kp - r - (k - 1)p = p - r$. Toda obenem se spomnimo, da je $s = t^k$; velja torej $t^k = ut^{k-1}v$. Če primerjamo začetka obeh strani te enakosti, vidimo, da se mora t začeti na u (saj je t daljši od u-ja); če pa primerjamo konca, vidimo, da se mora t končati na v (saj je t daljši tudi od v-ja). Toda u in v skupaj sta dolga natanko toliko kot t; torej je $t = uv$. Če to nesemo v $t^k = ut^{k-1}v$ (kar je oboje enako s), dobimo $(uv)^k = u(uv)^{k-1}v$. Leva stran je naprej enaka $u(vu)^{k-1}v$. Obe strani te enakosti se torej začneta na u in končata na v; če to dvoje na obeh straneh odrežemo, dobimo $(vu)^{k-1} = (uv)^{k-1}$, iz česar sledi $vu = uv = t$.

Iz tega med drugim sledi, da je $s = t^k = (vu)^k = v(uv)^{k-1}u = vt^{k-1}u$. Tako smo našli še eno pojavitev niza t^{k-1} kot podniza v s: poleg tiste na začetku (kot prefiks) in tiste $r = |u|$ znakov po začetku imamo zdaj še eno pojavitev $|v|$ znakov po začetku. Ker smo r definirali tako, da se nanaša na drugo pojavitev (gledano od leve proti desni), mora tale nova pravkar odkrita pojavitev ležati bolj desno, torej mora biti v daljši od u (ali pa sta enako dolga in smo v resnici našli še enkrat drugo pojavitev).

Oglejmo si spet $t = uv = vu$. V nizu t (dolžine p) je torej u hkrati prefiks in sufiks, enako pa tudi v; ker je (kot smo pravkar videli) v vsaj tako dolg kot u, je v dolg vsaj $p/2$. Opažanje, ki smo ga dokazali malo prej, nam zdaj pove, da je t semiperiodičen s periodo u. Torej je $t = u^\ell w$, pri čemer je w neki prefiks u-ja primerne dolžine (namreč dolžine $p \bmod r$). Če bi bil $|w| = 0$, bi bil t že sam zase periodičen, torej bi bila njegova perioda tudi perioda s-ja, kar je v protislovju s predpostavko, da je t najkrajša možna perioda niza s; to se torej ne more zgoditi. Ker je w prefiks u-ja, lahko pišemo $u = wz$ in zato $t = (wz)^\ell w$ (iz slednjega zaradi $t = uv$ dobimo še $v = (wz)^{\ell-1}w)$; torej se t konča na zw. Toda zaradi $t = vu$ vemo, da se t konča na $u = wz$. Ker se torej t konča na zw in tudi na wz in ker sta tadva niza enako dolga, morata biti enaka: $zw = wz$ (in oboje je enako u).

Enakost $zw = wz$ ima med drugim to koristno posledico, da lahko v kateremkoli nizu, dobljenem s stikanjem z-jev in w-jev, poljubno spremenimo njihov vrstni red, ne da bi se ta niz kaj spremenil; na primer: $wzwzw = w(zw)(zw) = w(wz)(wz) = ww(zw)z = ww(wz)z = wwwzz$, pri čemer vsaka enakost tu velja zato, ker smo uporabili $zw = wz$ na enem ali več podnizih v oklepajih. V našem primeru je to koristno zato, ker znamo s stikanjem w-jev in z-jev sestaviti u, v, nato t in končno s; zato je na primer tudi $zt = tz$ in $zt^{k-1} = t^{k-1}z$ in podobno.

Imamo torej $s = t^k = (uv)^k = ut^{k-1}v = wzt^{k-1}v$; zdaj uporabimo pravkar ugotovljeno dejstvo, da je $zt^{k-1} = t^{k-1}z$, pa dobimo $s = wt^{k-1}zv$. Torej se t^{k-1} pojavi kot podniz v s-ju že $|w|$ znakov naprej od začetka niza; mi pa smo na začetku predpostavili, da se druga pojavitev t^{k-1} v s (prva je na samem začetku, ker je t^{k-1} tudi prefiks s-ja) pojavi šele $r = |u|$ znakov naprej od začetka. Ker je $|u| > |w|$, smo prišli v protislovje. Naša začetna predpostavka, da se t^{k-1} pojavlja v s še kje vmes kot le na začetku in na koncu niza, je bila torej napačna. Primer $f(n-p) < p+1$ se torej pri tistem p, ki pomeni dolžino osnovne periode s-ja, ne more zgoditi; takrat bo gotovo veljalo $f(n-p) = p+1$. Osnovno periodo lahko torej poiščemo preprosto tako, da po naraščajočih p preverjamo ta pogoj in se ustavimo, čim je pri enem od njih izpolnjen.

Na ta način lahko v $O(n)$ časa poiščemo osnovno periodo niza dolžine n (spomnimo se, da tudi za izračun funkcije f po postopku iz Knuth-Morris-Prattovega algoritma potrebujemo le $O(n)$ časa). Za našo nalogo to pomeni, da lahko v $O(wh)$ časa poiščemo osnovne periode vseh vrstic in potem izračunamo njihov najmanjši skupni večkratnik; to je iskani w_p, širina osnovnega vzorca. Podobno nato v $O(wh)$ časa obdelamo še stolpce in dobimo h_p, višino osnovnega vzorca. Boljše rešitve od $O(wh)$ pa si pri tej nalogi ne moremo želeti, saj porabimo toliko časa že samo za branje vhodnih podatkov.

4. Pangramski podniz

Recimo, da je naš vhodni niz s dolg n črk, $s = s_1 s_2 \ldots s_n$. Podniz, ki nas zanima, bo oblike $s_i s_{i+1} \ldots s_{j-1} s_j$ za neka i in j. Najkrajši pangramski podniz lahko najdemo tako, da gremo v zanki po vseh možnih j (od 1 do n) in se pri vsakem vprašamo, kateri je najkrajši pangramski podniz, ki se konča pri tem j; to pa je seveda tisti, ki ima največji i. Kaj se dogaja s tem i, torej položajem levega konca podniza, če počasi povečujemo j, torej položaj desnega konca podniza? Če je bil $s_i \ldots s_j$ pangram in če potem premaknemo desni konec na s_{j+1}, bo niz $s_i \ldots s_{j+1}$ še vedno pangram, tako da se prav gotovo ne bo moglo zgoditi, da bi bilo treba kdaj pomakniti i nazaj v levo; mogoče pa je, da bo zdaj pangram tudi $s_{i+1} \ldots s_{j+1}$ in da smemo torej premakniti i v desno (in podniz tako še kaj skrajšati).

Tako imamo torej naslednji postopek: povečujemo j za 1 in po vsakem povečanju j-ja pogledamo, kako daleč smemo še povečati i, ne da bi podniz $s_i \ldots s_j$ prenehal biti pangram. Med vsemi tako dobljenimi podnizi si zapomnimo dolžino najkrajšega in jo na koncu vrnemo.

Stvar se malo zaplete le na začetku, kjer se lahko zgodi, da pri kakšnem j sploh ni nobenega pangrama, ker mogoče niti pri $i = 1$ podniz $s_i \ldots s_j$ še ne vsebuje vsake črke vsaj k-krat (prav gotovo se to zgodi npr. pri $j < 26 \cdot k$). Na začetku moramo torej pustiti i na 1 (da se podniz začne na začetku niza s) in povečevati j tako dolgo,

dokler $s_1 \ldots s_j$ ne postane pangram (mogoče je tudi, da se to ne zgodi nikoli, ker morda niti celoten s ni pangram).

Vprašanje je še, kako lahko poceni preverjamo, ali je opazovani podniz $s_i \ldots s_j$ pangram ali ne. V ta namen je koristno vzdrževati tabelo, ki za vsako črko abecede vsebuje število pojavitev te črke v podnizu. Ko povečamo j za 1, pride v podniz nova črka in zato ustrezni element tabele povečamo za 1; ko pa povečamo i za 1, izpade ena črka iz podniza in zato ustrezni element tabele zmanjšamo za 1.

Podniz je pangram, če se vsaka črka pojavlja vsaj k-krat. Ko torej po vsakem povečanju j-ja razmišljamo o tem, ali smemo zdaj tudi i povečati za 1, vidimo, da ga smemo povečati, če se črka s_i pojavlja v podnizu več kot k-krat, saj bo v tem primeru podniz ostal pangram, četudi eno pojavitev te črke izgubimo.

Na začetku, ko je nekaterih črk manj kot k, naš podniz sploh še ni pangram; da bomo laže ugotovili, kdaj postane pangram, je koristno vdzrževati še podatek o tem, koliko črk abecede se pojavlja manj kot k-krat. V spodnjem podprogramu imamo v ta namen spremenljivko premalo; vzdrževati je ni težko: ko se število pojavitev neke črke poveča s $k-1$ na k, zmanjšamo premalo za 1; in ko pade premalo na 0, vemo, da je naš podniz pangram (in bo odtlej to tudi ostal).

```
int PangramskiPodniz(const char *s, int k)
{
  enum { Abeceda = 26 };
  int n[Abeceda] = { };      // število pojavitev vsake črke v podnizu s[i...j]
  int premalo = Abeceda;  // koliko črk ima manj kot k pojavitev
  int naj = −1;               // najboljša rešitev doslej
  for (int i = 0, j = 0, c; s[j]; ++j)
  {
    // Trenutno imamo v tabeli „n“ števila pojavitev črk v s[i. . . j− 1].
    // Popravimo jih, da se bodo nanašala na s[i. . . j].
    if (++n[s[j] − 'a'] == k) −−premalo;

    // Če se kakšna črka pojavlja premalokrat, bomo morali podniz
    // na desni še podaljšati.
    if (premalo > 0) continue;

    // Mogoče lahko levi konec podniza premaknemo proti desni:
    // če ima črka s[i] več kot k pojavitev, jo smemo vreči iz podniza.
    while (n[c = s[i] − 'a'] > k) ++i, −−n[c];

    // Če je to najboljša rešitev doslej, si jo zapomnimo.
    if (naj < 0 || j − i + 1 < naj) naj = j − i + 1;
  }
  return naj; // Vrnimo najboljšo rešitev.
}
```

5. Tetris

Nalogo lahko rešujemo z rekurzijo. Ploščo bomo pokrivali sistematično: na vsakem koraku bomo poiskali najvišje nepokrito polje (če je takih več, pa najbolj levo med njimi) in ga na vse možne načine poskušali pokriti z enim od še razpoložljivih ploščkov. Pri tem moramo najprej preveriti, ali je plošček sploh mogoče postaviti tja; če da, ga postavimo in nadaljujemo z rekurzivnim klicem, ki bo poskušal s postavljanjem ostalih ploščkov do konca zapolniti ploščo. Če se je to posrečilo, lahko končamo, sicer pa pravkar postavljeni plošček spet odstranimo, da bomo poskusili še s kakšnim drugim.

Med rekurzijo je torej koristno imeti podatke o tem, do kod v plošči smo že pokrili vsa polja, tako da bomo lahko z iskanjem prvega nepokritega nadaljevali od tam in nam ne bo treba iti vsakič znova od začetka mreže. Poleg tega potrebujemo tudi tabelo oz. vektor števil, ki nam povedo, koliko ploščkov posamezne oblike je še na voljo (torej da jih še nismo položili na ploščo); ko položimo plošček, zmanjšamo ustrezni števec v tem vektorju, ko pa ga poberemo s plošče, njegov števec spet povečamo.

```
enum { W = 8, H = 8 }; // širina in višina plošče

// Funkcija vrne true, če je uspela v celoti pokriti ploščo.
// Sicer vrne false, plošča pa je ob vrnitvi iz funkcije v enakem stanju
// kot na začetku klica.
bool NadaljujPokrivanje(int x, int y, vector<int> &prosti)
{
  // Poiščimo naslednje nepokrito polje.
  while (y < H && JePokrito(x, y))
    if (++x == W) x = 0, ++y;
  if (y >= H) return true; // Če je vse že pokrito, smo končali.

  // Na vse možne načine ga poskusimo pokriti.
  for (int oblika = 0; oblika < prosti.size(); ++oblika)
  {
    // Ali lahko sem postavimo plošček te oblike?
    if (prosti[oblika] <= 0) continue;
    if (! PreveriPloscek(oblika + 1, x, y)) continue;

    // Postavimo ga in nadaljujmo z rekurzijo.
    PostaviPloscek(oblika + 1, x, y, true); --prosti[oblika];
    if (NadaljujPokrivanje(x, y, prosti)) return true;

    // Če nismo uspeli pokriti cele plošče, plošček spet odstranimo.
    PostaviPloscek(oblika + 1, x, y, false); ++prosti[oblika];
  }
  return false; // Če pridemo do sem, se plošče ni dalo pokriti do konca.
}
```

Glavna funkcija mora le inicializirati vektor z začetnim številom ploščkov vsake oblike in pognati rekurzijo od začetka mreže (torej od zgornjega levega kota):

```
bool PokrijPlosco()
{
  int n = StOblik();
  vector<int> prosti(n);
  for (int oblika = 0; oblika < n; ++oblika) prosti[oblika] = StPlosckov(oblika + 1);
  return NadaljujPokrivanje(0, 0, prosti);
}
```

Ob vrnitvi iz funkcije je plošča bodisi v celoti pokrita (tedaj funkcija vrne **true**) bodisi v enakem stanju kot na začetku, torej prazna (če se je sploh ni dalo pokriti; tedaj funkcija vrne **false**).

REŠITVE NALOG ZA TRETJO SKUPINO

1. Kapniki

Preprosta, a neučinkovita rešitev je, da gremo v zanki po vseh možnih višinah železnice (od 1 do v) in pri vsaki od njih s še eno vgnezdeno zanko pregledamo vseh n kapnikov ter preštejemo, koliko od njih bi bilo treba pri tej višini železnice podreti. Ta rešitev ima časovno zahtevnost $O(nv)$ in bi pri testnih primerih z našega tekmovanja dobila 20 % točk.

Poskusimo to rešitev izboljšati. Kako se spreminja število motečih kapnikov, ko v zanki počasi dvigujemo železnico za 1? Stalagmit višine k_i nam je v napoto pri $1 \leq y \leq k_i$, stalaktit višine k_i pa pri $v - k_i + 1 \leq y \leq v$. Na začetku, na višini $y = 1$, nas motijo vsi stalagmiti in noben stalaktit; nato pa, ko počasi dvigujemo železnico, nam je počasi v napoto vse manj stalagmitov in vse več stalaktitov, dokler nas na koncu pri $y = v$ ne motijo vsi stalaktiti in noben stalagmit. Ko dvignemo železnico z višine $y - 1$ na višino y, nas nehajo motiti stalagmiti višine $y - 1$, začnejo pa nas motiti stalaktiti višine $v - y + 1$.

Lahko bi imeli torej tabelo, v kateri bi za vsako možno višino označili, koliko kapnikov nas tam začne ali neha motiti; to tabelo lahko pripravimo med branjem kapnikov, nato pa gremo v zanki po vseh višinah od 1 do v in te spremembe v številu motečih kapnikov seštevamo, pa bomo v vsakem trenutku točno vedeli, koliko kapnikov nas na tej višini moti. Ta rešitev porabi $O(n + v)$ časa in prostora, kar bo pri naših testnih primerih dovolj za 60 % točk.

Opazimo pa lahko, da ko pregledujemo možne višine železnice od 1 do v, lahko posamezni kapnik povzroči spremembo le pri eni višini: pri tisti, kjer se začne (če je stalaktit) ali konča (če je stalagmit). Čeprav imamo lahko morda 10^{18} višin, je možnih sprememb le toliko kot kapnikov, kar je največ 10^5. Na območju med dvema zaporednima spremembama je število motečih kapnikov ves čas enako, zato se nam ni treba ukvarjati z vsako višino na tem območju posebej.

Pripravimo si torej seznam parov (y, d), ki povedo, da se število motečih kapnikov zmanjša za d, ko višina železnice naraste z $y - 1$ na y. Stalagmit višine k_i torej v ta seznam prispeva par $(k_i + 1, 1)$, stalaktit višine k_i pa par $(v - k_i + 1, -1)$. Nazadnje dodajmo v seznam še par $(v + 1, 0)$, ki ponazarja konec jame, tako da bomo lahko upoštevali tudi višine od zadnje spremembe v številu motečih kapnikov do stropa jame ($y = v$).

Nato te pare uredimo, tako da tisti z isto višino pridejo skupaj, med njimi pa pridejo najprej tisti za stalaktite (ki število motečih kapnikov povečujejo), nato pa tisti za stalagmite (ki to število zmanjšujejo). Tako bomo, če pride do več sprememb na isti višini, najprej prekomerno povečali število motečih kapnikov, ko bomo prištevali stalaktite, in ga nato zmanjšali na pravo vrednost, ko bomo odštevali stalagmite. Na ta način ne bomo nikoli imeli premajhnega števila motečih kapnikov, na koncu take skupine sprememb na isti višini pa bomo imeli točno pravo število motečih kapnikov na tej višini. Tako lahko po vsaki spremembi preverimo, če je trenutno število motečih kapnikov najnižje doslej, in če je, si ga zapomnimo. Razlika v višini med prejšnjo in trenutno spremembo pa nam pove, na koliko višinah velja tisto število motečih kapnikov, ki je veljalo pred trenutno spremembo; to bo prišlo

prav, ker moramo izpisati ne le najmanjšega možnega števila motečih kapnikov, ampak tudi to, na koliko višinah ga dosežemo.

Ta rešitev ima časovno zahtevnost $O(n \log n)$, namreč zaradi urejanja sprememb; vse ostalo je $O(n)$.

```
#include <iostream>
#include <utility>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
typedef long long int llint;

int main()
{
  // Preberimo število in tip kapnikov.
  llint v; int n, m = 0; string s; cin >> v >> n >> s;
  // Preberimo višine kapnikov in pripravimo tabelo parov (y, d), ki povedo, da nas neki
  // kapnik na novo (d < 0 ? začne : neha) motiti, ko višina železnice zraste na y.
  vector<pair<llint, int>> spremembe(n + 1);
  for (int i = 0; i < n; ++i)
  {
    llint ki; cin >> ki;
    // Če je to stalagmit, nas neha motiti pri višini ki + 1.
    if (s[i] == 'M') ++m, spremembe[i] = {ki + 1, 1};
    // Če je stalaktit, nas začne motiti pri višini v − ki + 1.
    else spremembe[i] = {v − ki + 1, −1};
  }
  spremembe[n] = {v + 1, 0};
  // Pregledujmo spremembe po naraščajoči višini. „m“ pove, koliko
  // kapnikov nas moti; trenutno (pri y = 1) so to vsi stalagmiti.
  sort(spremembe.begin(), spremembe.end());
  int naj = n + 1; llint koliko = 0;
  for (int i = 0; i <= n; i++)
  {
    llint dv = spremembe[i].first − (i == 0 ? 1 : spremembe[i − 1].first);
    // Od prejšnje spremembe do trenutne je dv možnih višin, kjer nas moti m kapnikov.
    if (m < naj) naj = m, koliko = dv; // Nova najboljša rešitev.
    else if (m == naj) koliko += dv;   // Izenačena dosedanja najboljša rešitev.
    // Upoštevajmo spremembo v številu motečih kapnikov.
    m −= spremembe[i].second;
  }
  printf("%d %lld\n", naj, koliko); return 0;
}
```

Namesto urejanja parov (y, d) bi lahko uporabili kakšno primerno uravnoteženo drevesasto podatkovno strukturo, na primer rdeče-črno drevo (v C++ lahko uporabimo razred map iz standardne knjižnice); kot ključe v njem bi uporabili vrednosti y, pripadajoča vrednost pa bi bilo število, ki pove, za koliko se pri tem y spremeni število kapnikov, ki so nam v napoto. Na koncu se lahko s pomočjo te strukture sprehodimo po vseh takih višinah v naraščajočem vrstnem redu in sproti primerno popravljamo število kapnikov, ki so nam trenutno v napoto. Časovna zahtevnost te rešitve je še vedno $O(n \log n)$, ker nam vsaka operacija na drevesu vzame $O(\log n)$ časa.

2. Socialno omrežje

Naj bo $P(u)$ množica, ki jo sestavljajo oseba u in vsi njeni prijatelji; in naj bo $S(u)$ množica, ki jo sestavljajo vsi sovražniki osebe u. Razmislimo zdaj, kako preverjati vsako od treh pravil, ki jih omenja besedilo naloge.

(1) Pravilo „prijatelj mojega prijatelja je tudi moj prijatelj“ pomeni, da če sta u in v prijatelja, mora biti vsak u-jev prijatelj tudi v-jev prijatelj in obratno, torej mora biti $P(u) = P(v)$. To velja za vsakega u-jevega prijatelja v; vidimo torej, da morajo vsi u-jevi prijatelji imeti enako množico prijateljev kot u.

Ta pogoj je torej potreben, da pravilo (1) velja, hitro pa vidimo, da je tudi zadosten: recimo, da za vsak u in za vse $v \in P(u)$ velja $P(v) = P(u)$. Vzemimo poljubne take konkretne u, v in w, kjer je $v \in P(u)$ in $w \in P(v)$. Iz $v \in P(u)$ potem sledi, da je $P(u) = P(v)$, torej je $w \in P(v) = P(u)$, torej je w (ki je prijatelj u-jevega prijatelja, namreč v-ja) tudi u-jev prijatelj.

Pravilo (1) lahko torej preverimo tako, da gremo pri vsakem u po vseh njegovih prijateljih v in preverjamo, če velja $P(u) = P(v)$. Za vsak v si tudi označimo, da smo ga že videli in da nam kasneje ni treba iti še zanj po *njegovih* prijateljih w in preverjati, če zanje velja $P(v) = P(w)$, kajti takrat že vemo, da ima v iste prijatelje kot u in če je preverjanje pogoja uspelo pri u, bo tudi pri v, torej ga pri v ni treba preverjati še enkrat.

(2) Pravilo „sovražnik mojega prijatelja je tudi moj sovražnik“ pomeni, da če sta u in v prijatelja, mora biti vsak u-jev sovražnik tudi v-jev sovražnik in obratno, torej mora biti $S(u) = S(v)$. Vsi u-jevi prijatelji morajo torej imeti ne le enako množico prijateljev kot u (zaradi prvega pravila), pač pa tudi enako množico sovražnikov kot u (zaradi drugega pravila). O tem, da je ta pogoj tudi zadosten, se lahko prepričamo čisto podobno kot pri prvem pravilu. Ta pogoj lahko tudi preverjamo v isti sapi s tistim za pravilo (1).

Preden se lotimo tretjega pravila, imejmo v mislih naslednje: naloga od nas zahteva le, da ugotovimo, če omrežje ustreza vsem trem pravilom skupaj, ne pa za vsako pravilo posebej. Če na primer omrežje ne ustreza pravilu (1) ali (2), bomo že zaradi tega morali odgovoriti, da pravilom ne ustreza, in takrat bo vseeno, ali pravilo (3) sploh preverjamo (oz. ali ga preverjamo pravilno ali ne). Zato smemo pri preverjanju pravila (3) predpostaviti, da omrežje praviloma (1) in (2) ustreza.

(3) Zdaj nam torej ostane še pravilo „sovražnik mojega sovražnika je moj prijatelj“. To pomeni, da če ima na primer u sovražnika v in w, sta si slednja v odnosu „sovražnik mojega sovražnika“, torej si morata biti prijatelja; vsi u-jevi sovražniki morajo biti v-jevi prijatelji: $S(u) \subseteq P(v)$. Po drugi strani za vsakega v-jevega prijatelja x velja, da je u sovražnik x-ovega prijatelja (namreč v-ja), torej mora biti tudi x-ov sovražnik, saj smo rekli, da bomo pri preverjanju pravila (3) predpostavili, da omrežje ustreza pravilu (2); torej je vsak v-jev prijatelj tudi u-jev sovražnik: $P(v) \subseteq S(u)$. Oboje skupaj nam dá $S(u) = P(v)$.

Ali je treba ta pogoj preveriti za *vse* u-jeve sovražnike? Rekli smo, da bomo pri preverjanju pravila (3) predpostavili tudi, da omrežje ustreza pravilu (1); zaradi slednjega imajo vsi v-jevi prijatelji enako množico prijateljev; če smo torej preverili, da je $S(u) = P(v)$, potem za poljubnega drugega u-jevega sovražnika, recimo w, velja, da je $w \in S(u)$, zato $w \in P(v)$, zato je w prijatelj v-ja in ima torej tudi sam enako množico prijateljev kot v: $P(w) = P(v)$, torej $S(u) = P(w)$, torej nam tega

pogoja ni treba preveriti še za w, če smo ga že preverili za v. Dovolj je torej pri vsakem u pogledati le *enega* njegovega sovražnika v (katerega koli) in zanj preveriti, ali je $S(u) = P(v)$.

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

bool ObdelajOmrezje()
{
  // Preberimo podatke o omrežju.
  int n, m; scanf("%d %d", &n, &m);
  vector<unordered_set<int>> P(n), S(n); // prijatelji in sovražniki vsakega človeka
  for (int u = 0; u < n; ++u) P[u].insert(u);
  for (int i = 0; i < m; ++i) {
    int ai, bi, pi; scanf("%d %d %d", &ai, &bi, &pi);
    auto &V = pi ? P : S; V[--ai].insert(--bi); V[bi].insert(ai); }

  // Preverimo, če ustreza vsem trem pravilom.
  vector<bool> pregledan(n, false);
  for (int u = 0; u < n; ++u) if (! pregledan[u])
  {
    // Preverimo, ali imajo vsi u-jevi prijatelji enaki množici
    // prijateljev in sovražnikov kot u (pravili 1 in 2).
    for (int v : P[u])
      if (P[v] != P[u] || S[v] != S[u]) return false;
      else pregledan[v] = true;

    // Če ima u kaj sovražnikov, morajo biti oni prijatelji med sabo
    // (pravilo 3). Dovolj je, če to preverimo za enega od njih.
    for (int v : S[u])
      if (P[v] != S[u]) return false;
      else break;
  }
  return true;
}

int main()
{
  // Preberimo število omrežij in jih obdelajmo v zanki.
  int t; scanf("%d", &t);
  while (t-- > 0) printf("%s\n", ObdelajOmrezje() ? "DA" : "NE");
  return 0;
}
```

Razmislimo še o časovni zahtevnosti te rešitve. Najbolj neugodna poraba časa nastopi, če omrežje ustreza vsem pravilom, in sicer zato, ker se postopek izvede do konca le takrat (sicer pa se prekine, čim opazi, da omrežje kakšnemu pravilu ne ustreza), pa tudi zato, ker preverjanje, ali sta dve množici enaki, vzame največ časa ravno takrat, ko sta res enaki — za množici A in B nam to vzame $O(\min\{|A|, |B|\})$ časa.

Recimo torej zdaj, da omrežje ustreza vsem trem pravilom. Takšno omrežje je razdeljeno na eno ali več skupin prijateljev, pri čemer so si v vsaki skupini vsi ljudje med seboj prijatelji (in vsi imajo enako množico sovražnikov). Od vsake take skupine se naša glavna zanka ukvarja le z enim u, kajti pri vseh nadaljnjih članih skupine

vidi, da so bili že pregledani. Recimo, da ima i-ta skupina k_i članov in da imajo po s_i sovražnikov; potem porabimo pri tej skupini za vsakega od k_i članov po $O(k_i)$ časa, da preverimo $P(u) = P(v)$, in še $O(s_i)$ časa, da preverimo $S(u) = S(v)$; nato pa porabimo še $O(s_i)$ časa, da preverimo $S(u) = P(v)$ za enega od u-jevih sovražnikov v. Vsega skupaj torej pri tej skupini porabimo $O(k_i^2 + k_i s_i)$ časa; časovno zahtevnost celotnega postopka dobimo, ko to seštejemo po vseh skupinah.

Toda spomnimo se, da je $k_i^2 + k_i s_i$ ravno skupno število prijateljstev in sovraštev, v katerih so udeleženi člani te skupine; vse te odnose, za vse skupine, smo morali na začetku prebrati iz vhodnih podatkov, opis naloge pa pravi, da jih je (po vseh skupinah) največ m. Natančneje povedano, vsota $\sum_i (k_i^2 + k_i s_i)$ po vseh skupinah i je enaka $2m + n$, saj smo si vsak odnos, prebran iz vhodnih podatkov, zapisali na dveh mestih (npr. da je u prijatelj v-ja in da je v prijatelj u-ja), poleg tega pa smo v vsako $P(u)$ dodali še u-ja samega. Časovna zahtevnost postopka je torej $O(n + m)$.

3. Proizvodnja cepiva

V tej rešitvi bomo besedo *proizvodnja* vedno uporabljali za skupno količino proizvedenega cepiva do vključno določenega dne, številu opravljenih nadgradenj do vključno določenega dne pa bomo rekli *zmogljivost* (ker določa, koliko cepiva bomo odtlej lahko proizvedli v enem dnevu).

Nalogo lahko rešujemo z dinamičnim programiranjem. Najprej si bomo ogledali preprosto rešitev, ki je za testne primere na našem tekmovanju že čisto dovolj dobra, kasneje pa bomo videli, da jo lahko tudi še precej izboljšamo.

Naj bo $f(t, z)$ največja možna proizvodnja, ki jo je mogoče doseči v prvih t dneh, če smo v tem času izvedli natanko z nadgradenj in če smo ob tem spoštovali vse omejitve, ki zapadejo v tem času (torej ki imajo $x_i \leq t$). Naloga potem pravzaprav sprašuje po najmanjšem takem t, pri katerem funkcija f doseže vrednost k ali več; to, pri katerem z jo doseže, ni pomembno, saj nam je vseeno, s koliko nadgradnjami dosežemo zahtevano proizvodnjo; tudi ni pomembno, če je ta t manjši od roka kakšne omejitve, saj smo s proizvodnjo k gotovo izpolnili tudi vse morebitne preostale omejitve (besedilo naloge namreč zagotavlja, da je pri vseh omejitvah $y_i \leq k$).

Vrednosti $f(t, z)$ lahko računamo s preprostim rekurzivnim razmislekom: na dan t bodisi proizvajamo bodisi nadgrajujemo. Če proizvajamo, delamo to pri zmogljivosti z, torej bomo proizvedli z enot cepiva, pred tem pa smo morali v prvih $t - 1$ dneh izvesti z nagradenj in smo torej proizvedli največ $f(t - 1, z)$ enot; tako imamo skupno proizvodnjo $f(t-1, z) + z$. Če pa na dan t nadgrajujemo, ne proizvedemo ničesar, v prvih $t-1$ dneh pa smo morali imeti $z-1$ nadgradenj in smo torej proizvedli največ $f(t - 1, z - 1)$ enot. Za $f(t, z)$ bomo vzeli boljšo od teh dveh možnosti:

$$f(t, z) = \max\{f(t - 1, z - 1), f(t - 1, z) + z\}.$$

Ta funkcija je načeloma definirana za $0 \leq z \leq t$, saj v t dneh ne moremo izvesti več kot t nadgradenj. Robni primer nastopi pri $z = 0$: začetna zmogljivost je 0 in če smo pri njej tudi ostali, bo tudi proizvodnja 0, torej je $f(t, 0) = 0$. Funkcijo lahko računamo po naraščajočih t, pri vsakem t pa z vgnezdeno zanko po z. Opazimo lahko tudi, da ko računamo vrednosti funkcije f za dan t, potrebujemo le tiste za dan $t - 1$, starejše vrednosti (za dneve od 0 do $t - 2$) pa lahko sproti pozabljamo in tako privarčujemo nekaj pomnilnika.

Omejitve (x_i, y_i) lahko zdaj upoštevamo takole: če na dan t zapade neka taka omejitev, torej če je $x_i = t$, potem za vsak z, pri katerem je dosedanja formula dala $f(t,z) < y_i$, v mislih postavimo $f(t,z)$ na $-\infty$. Vrednost $f(t,z) < y_i$ namreč pomeni, da z natanko z nadgradnjami v prvih t dneh ni mogoče izpolniti omejitve (x_i, y_i), torej se na razpored nadgradenj, s katerim je bila vrednost $f(t,z)$ dosežena, tudi ne smemo sklicevati v nadaljevanju pri računanju funkcije f za kasnejše dneve (na primer za dan $t+1$). Ravno to pa dosežemo s tem, ko smo postavili njeno vrednost na $-\infty$, saj bo tako postala popolnoma neprivlačna v formuli za izračun $f(t+1,z)$ in $f(t+1,z+1)$.

Oglejmo si zdaj implementacijo te rešitve v C++. Vrednosti funkcije f za trenutni dan t bomo hranili v vektorju f, tiste za dan $t+1$ pa računali v vektorju ff. Namesto vrednosti $-\infty$ uporabljamo -1, pazimo pa seveda na to, kako jo uporabljamo v $\max\{\ldots\}$ v formuli za $f(t,z)$. Za lažje upoštevanje omejitev bomo le-te na začetku uredili po času x_i. V zunanji zanki gremo potem po omejitvah, znotraj nje pa v vgnezdeni zanki po dnevih t do roka x_i trenutne omejitve in računamo vrednosti funkcije f; ko pridemo do x_i, pa še preverimo, pri katerih z je res dosežena zahtevana proizvodnja y_i. Čim dosežemo proizvodnjo k, se takoj ustavimo in izpišemo trenutni dan t, saj so s tem gotovo dosežene že tudi vse morebitne preostale omejitve. Ostane še možnost, da je proizvodnja k dosežena šele po roku zadnje omejitve; za to lahko poskrbimo tako, da na konec seznama omejitev dodamo še eno namišljeno omejitev z dovolj poznim rokom (npr. $k+1$ dni za rokom zadnje prave omejitve), da bo do takrat gotovo dosežena proizvodnja k.

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

int main()
{
  // Preberimo vhodne podatke.
  int k, d; cin >> k >> d;
  vector<pair<int, int>> omejitve(d);
  for (auto &O : omejitve) cin >> O.first >> O.second;

  // Uredimo omejitve po času.
  sort(omejitve.begin(), omejitve.end());

  // Dodajmo na konec še eno omejitev, v kateri bo proizvodnja k gotovo dosežena.
  omejitve.emplace_back((omejitve.empty() ? 0 : omejitve.back().first) + k, k);

  // Naj bo f(t, z) največja možna proizvodnja v prvih t dneh,
  // če izvedemo natanko z nadgradenj in upoštevamo vse omejitve.
  vector<int> f, ff; int t = 0; f.push_back(0); ff.push_back(0);
  for (auto [xi, yi] : omejitve)
  {
    // Izračunajmo možnosti proizvodnje do vključno dneva xi.
    while (t < xi)
    {
      f.push_back(-1); ff.push_back(-1); ++t;
      // f[z] hrani vrednosti f(t - 1, z); izračunajmo f(t, z) in jih shranimo v ff[z].
      for (int z = 1; z <= t; ++z)
        if ((ff[z] = max(f[z - 1], f[z] < 0 ? -1 : f[z] + z)) >= k) {
```

```
            // Če dosežemo proizvodnjo k, lahko takoj končamo.
            cout << t << endl; return 0; }
        swap(f, ff);
      }
      // Razveljavimo rezultate, ki ne dosežejo trenutne omejitve.
      for (auto &fi : f) if (fi < yi) fi = −1;
   }
   // Če pridemo sem, je naloga zaradi omejitev nerešljiva.
   return −1;
}
```

Zunanja zanka (po omejitvah) se načeloma vedno prekine predčasno s tem, da dosežemo proizvodnjo k. Edini način, da pridemo do stavka **return** -1 na koncu, je ta, da je problem nerešljiv, ker ni mogoče ustreči vsem omejitvam (na primer: morda imamo omejitev, ki zahteva že v prvih treh dneh proizvodnjo 10 enot); toda naloga zagotavlja, da se pri naših testnih primerih to ne bo zgodilo.

Razmislimo o časovni zahtevnosti te rešitve. Če je T rezultat, ki ga na koncu vrnemo, torej število dni, v katerih smo proizvedli k enot, smo do takrat izračunali $f(t, z)$ za vse $0 \leq z \leq t \leq T$, kar nam vzame $O(T^2)$ časa. Pred tem je še $O(d \log d)$ časa za urejanje omejitev po roku x_i, kar je sicer v praksi zanemarljivo, saj je število omejitev d majhno. Tega, kakšen je v najslabšem primeru T, sicer ni čisto trivialno oceniti, lahko pa si predstavljamo, da je T bolj reda $O(\sqrt{k})$ kot $O(k)$, kajti proizvodnjo k lahko dosežemo v $2\sqrt{k}$ dneh tako, da najprej $\sqrt{k}$ dni nadgrajujemo in nato $\sqrt{k}$ dni proizvajamo pri zmogljivosti $\sqrt{k}$ enot na dan. Kasneje bomo videli, da je $T = O(\sqrt{kd})$, torej ima naša dosedanja rešitev zahtevnost $O(kd + d \log d)$. (Videli bomo tudi, da za najboljši rezultat ne potrebujemo več kot $\lfloor\sqrt{k}\rfloor$ nadgradenj, torej nam funkcije $f(t, z)$ ne bi bilo treba računati za vse $z \leq t$, ampak le do $z = \min\{t, \lfloor\sqrt{k}\rfloor\}$, kar nam zmanjša čas računanja funkcije f na $O(T\sqrt{k})$, torej skupaj $O(k\sqrt{d} + d \log d)$ za celoten postopek.)

Dosedanja rešitev je za testne primere na našem tekmovanju že več kot dovolj dobra. V nadaljevanju si bomo ogledali še eno ali dve boljši rešitvi, ob tem pa se bomo tudi bolje seznanili s strukturo problema, kar nam bo omogočilo oceniti časovno zahtevnost naših rešitev.

Boljša rešitev z razmislekom po intervalih. Uredimo spet, tako kot že pri prejšnji rešitvi, omejitve (x_i, y_i) naraščajoče po času x_i. Če imamo več omejitev z istim rokom x_i, obdržimo le tisto z najvišjo zahtevano proizvodnjo y_i, saj bo vsak razpored nadgradenj, ki ustreza tej omejitvi, ustrezal tudi tistim z istim rokom in nižjo zahtevano proizvodnjo. Podobno, če opazimo, da ima naslednja omejitev sicer kasnejši rok kot prejšnja, vendar nižjo ali enako zahtevano proizvodnjo, lahko to naslednjo omejitev pobrišemo, saj bo vsak razpored, ki bo ustrezal prejšnji, s tem že ustregel tudi tej naslednji. V nadaljevanju torej predpostavimo, da smo omejitve na ta način uredili in prečistili, tako da zdaj velja $0 < x_1 < x_2 < \ldots < x_d$ in $0 < y_1 < y_2 < \ldots < y_d$. Za potrebe opisa naše rešitve si mislimo še $x_0 = y_0 = 0$.

Recimo, da imamo dva zaporedna dneva, t in $t + 1$, da dan t ni rok nobene omejitve in da na dan t proizvajamo cepivo, naslednji dan pa nadgrajujemo tovarno. Če bi to dvoje zamenjali in raje nadgrajevali na dan t, proizvajali pa na dan $t+1$, bi bila zmogljivost na koncu teh dveh dni enaka kot pred to zamenjavo, proizvodnja pa za 1 višja, ker bi na dan $t+1$ po nadgradnji proizvedli eno enoto več kot v prvotnem

scenariju na dan t pred nadgradnjo. To, da se je proizvodnja, do katere bi sicer prišlo na dan t, zdaj zamaknila na dan $t+1$, bi lahko bila težava le, če bi na dan t padel rok kakšne omejitve, kar pa se, kot smo predpostavili, ne zgodi.

Roki omejitev nam razdelijo časovno premico na intervale, pri čemer r-ti omejitvi ustreza interval, ki ga tvorijo dnevi $x_{r-1}+1, \ldots, x_r$. Dolžino r-tega intervala označimo s $t_r := x_r - x_{r-1}$. Če imamo nek veljaven razpored nadgradenj (tak, ki ustreza vsem omejitvam) in če znotraj nekega intervala ne nastopijo vse nadgradnje na začetku intervala, lahko razpored izboljšamo, če jih premaknemo na začetek intervala, saj smo v prejšnjem odstavku videli, da se pri tem proizvodnja poveča, zmogljivost pa ostane enaka, torej bo tako spremenjen razpored še vedno ustrezal omejitvam. Ni se nam torej treba bati, da bi spregledali optimalno rešitev, če se v nadaljevanju omejimo na take razporede, ki znotraj vsakega intervala vedno najprej nekaj dni (lahko tudi 0 dni) samo nadgrajujejo, potem pa odtlej do konca intervala samo proizvajajo.

Pri iskanju optimalnega razporeda vsaj do konca zadnjega intervala, se pravi za prvih x_d dni (z vprašanjem, kako od tam potem najhitreje priti do proizvodnje k, se bomo ukvarjali malo kasneje), je torej vprašanje predvsem, *koliko* nadgradenj izvesti v vsakem od teh intervalov — to, *kdaj* jih izvesti, pa smo že videli: na začetku vsakega intervala. Na misel nam lahko pride, da bi razpored gradili postopoma: najprej izberemo število nadgradenj v prvem intervalu, nato v drugem in tako naprej. Toda kaj je pravzaprav najboljše število nadgradenj v prvem intervalu? To ni nujno tisto, ki maksimizira proizvodnjo v prvem intervalu; včasih je bolje posvetiti več časa nadgrajevanju in proizvesti le toliko, kolikor je nujno treba, da dosežemo omejitev y_1, zato pa imamo potem višjo zmogljivost za kasnejšo proizvodnjo. Po drugi strani tudi ni nujno najbolje izvesti največjega možnega števila nadgradenj, s katerim še lahko dosežemo omejitev y_1, ker bo tako dosežena proizvodnja mogoče preslabo izhodišče za naslednjo omejitev (ki ima mogoče rok kmalu za prvo, vendar občutno višjo y_2). Ker torej vnaprej ne moremo vedeti, koliko nadgradenj je pametno izvesti v prvem intervalu, bomo preizkusili vse možnosti in si za vsako zapomnili doseženo proizvodnjo.

Ker velja podoben razmislek tudi pri kasnejših intervalih, si zastavimo podprobleme takšne oblike: naj bo $f_r(z)$ največja proizvodnja, ki jo je mogoče doseči do konca r-tega intervala (torej v prvih x_r dneh), če moramo pri tem ustreči prvim $r-1$ omejitvam in opraviti natanko z nadgradenj. Če ta problem sploh ni rešljiv, si mislimo $f_r(z) = -\infty$. Tudi če je rešljiv, še ni nujno, da doseže proizvodnjo y_r, ki jo zahteva r-ta omejitev; definirajmo $z_r^{\min}$ in $z_r^{\max}$ kot najmanjšo in največjo vrednost z-ja, pri kateri je $f_r(z) \geq y_r$.

Ko rešujemo tak podproblem — torej ko računamo vrednost $f_r(z)$ — se moramo med drugim odločiti, koliko nadgradenj bi opravili v zadnjem (torej r-tem) intervalu; recimo, da jih opravimo u. V prejšnjih $r-1$ intervalih moramo torej opraviti $z-u$ nadgradenj in ustreči vsem tamkajšnjim omejitvam; največja proizvodnja, ki jo je mogoče pri tem doseči, je $f_{r-1}(z-u)$. V r-tem intervalu potem porabimo u dni za nadgradnje, s čimer dosežemo zmogljivost z, tako da potem v preostalih $t_r - u$ dneh tega intervala proizvedemo še $(t_r - u) \cdot z$ enot. Tako dosežemo proizvodnjo

$$F_r(z, u) := f_{r-1}(z-u) + (t_r - u) \cdot z.$$

Ker hočemo za $f_r(z)$ čim večjo proizvodnjo, bomo seveda vzeli tak u, pri katerem

je $F_r(z, u)$ čim večja. Toda kateri u sploh pridejo v poštev? Seveda mora biti $0 \leq u \leq t_r$, saj ima r-ti interval le t_r dni; poleg tega pa mora biti $z^{\min}_{k-1} \leq z - u \leq z^{\max}_{k-1}$, saj drugače razpored za prvih $r - 1$ intervalov z $z - u$ nadgradnjami sploh ne more ustreči prvim $r - 1$ omejitvam.[6] Tako lahko zaključimo:

$$f_r(z) = \max\{F_r(z, u) : \max(0, z - z^{\max}_{r-1}) \leq u \leq \min(t_r, z - z^{\min}_{r-1})\}.$$

Za katere z ima sploh smisel razmišljati o tem? Z manj kot $z^{\min}_{r-1}$ nadgradnjami ne moremo ustreči niti prvim $r - 1$ omejitvam, torej tudi prvim r omejitvam ne; po drugi strani, če bi hoteli več kot $z^{\max}_{r-1} + t_r$ nadgradenj, bi lahko porabili za nadgradnje celoten r-ti interval (t_r dni), pa bi jih še vedno ostalo več kot $z^{\max}_{r-1}$ za prejšnjih $r-1$ intervalov, s čimer spet ne bo mogoče ustreči prvim $r-1$ omejitvam. Tako torej vidimo, da je $f_r(z)$ smiselno računati za $z^{\min}_{r-1} \leq z \leq z^{\max}_{r-1} + t_r$. (Tema dvema mejama recimo $\hat{z}^{\min}_r := z^{\min}_{r-1}$ in $\hat{z}^{\max}_r = z^{\max}_{r-1} + t_r$; to bo prišlo prav kasneje. Zunaj tega območja je f_r nedefinirana (oz. enaka $-\infty$), kajti tam nam, kot smo videli, ostane za prvih $k - 1$ intervalov premalo ali preveč nagradenj, da bi z njimi lahko ustregli prvim $k - 1$ omejitvam.)

Vidimo, da za izračun funkcije f_r potrebujemo vrednosti funkcije f_{r-1}, zato je koristno te funkcije računati naraščajoče po r. Pri vsakem r gremo v gnezdeni zanki po z in v še eni po u, izračunane vrednosti $f_r(z)$ pa shranjujemo v tabelo, da nam bodo pri roki kasneje za izračun funkcije f_{r+1}. Začnemo pa pri $r = 0$, kjer imamo le $z = 0$ in $f_0(0) = 0$.

Ta postopek se konča, če ne prej, takrat, ko pridemo do konca omejitev in izračunamo tudi funkcijo f_d; tedaj nam ostane še vprašanje, kako od tam čim hitreje priti do želene končne proizvodnje k. Lahko pa se že prej pri nekih r in z zgodi, da dobimo $f_r(z) \geq k$; to pomeni, da lahko že v r-tem intervalu dosežemo zahtevano proizvodnjo k, s tem pa gotovo ustrežemo tudi vsem preostalim omejitvam $(y_r, \ldots, y_d)$, saj besedilo naloge pravi, da za vse omejitve velja $y_i \leq k$. V tem primeru se lahko s trenutno omejitvijo in vsemi preostalimi nehamo ukvarjati in se delamo, da je bilo omejitev le $r - 1$ (v mislih postavimo d na $r - 1$) ter takoj skočimo na vprašanje, kako nadaljevati po tej zadnji omejitvi, da bomo čim hitreje dosegli proizvodnjo k.

Recimo torej zdaj, da smo ustregli vsem omejitvam in v prvih x_d dneh izvedli z nadgradenj ter proizvedli $f_d(z)$ enot cepiva. Kako od tu čim hitreje doseči proizvodnjo k — z drugimi besedami, kako čim prej proizvesti še $k - f_d(z)$ enot? Ker nas od tu naprej ne vežejo več roki omejitev, je tudi zdaj smiselno najprej nekaj časa le nadgrajevati, nato pa le proizvajati. Če porabimo u dni za nadgradnje, bomo imeli zmogljivost $z + u$ in za izdelavo $k - f_d(z)$ enot bomo potrebovali še $\lceil (k - f_d(z))/(z + u) \rceil$ dni. Skupaj z u dnevi za nadgradnje smo tako porabili

[6] Načeloma bi morali dodati še pogoj, da mora biti $f_{r-1}(z - u) \geq y_{r-1}$, saj drugače razpored, ki ga dobimo pri tem u, ne bo ustrezal omejitvi $r - 1$. Toda kot bomo videli kasneje, je funkcija f_{r-1} konkavna, kar pomeni, da najprej nekaj časa samo narašča, od nekega trenutka naprej pa ves čas samo še pada. Spomnimo se, da ima f_{r-1} pri $z^{\min}_{k-1}$ in $z^{\max}_{k-1}$ vrednost vsaj y_r ter da $z - u$ leži nekje na območju od $z^{\min}_{k-1}$ do $z^{\max}_{k-1}$; torej, če bi bila $f_{r-1}(z - u) < y_r$, bi to pomenilo, da je funkcija f_{r-1} od $z^{\min}_{r-1}$ do $z - u$ nekje padala, kasneje pa je od $z - u$ do $z^{\max}_{r-1}$ nekje naraščala. To pa je pri konkavni funkciji nemogoče, saj taka funkcija, ko enkrat začne padati, kasneje nikoli več ne narašča. Toda tudi če tega razmisleka ne opravimo, lahko v našem programu, ko bomo z zanko pregledovali možne u, pri vsakem brez težav še pogledamo, če zanj velja $f_{r-1}(z - u) \geq y_{r-1}$; ta pogoj bo sicer vedno izpolnjen in z njim bomo le zapravili nekaj časa, vendar ne toliko, da bi se nam bilo treba zaradi tega vznemirjati.

$g(u) := u + \lceil (k - f_d(z))/(z + u) \rceil$ dni. Iščemo seveda tak u, pri katerem bo $g(u)$ najmanjša. Preprosta rešitev je, da v zanki povečujemo u po 1, računamo $g(u)$ in si zapomnimo najmanjšo doslej doseženo vrednost te funkcije; recimo ji $g^\star$. Sčasoma nam u doseže $g^\star$ in takrat vemo, da bo odtlej $g(u) > u \geq g^\star$, torej odtlej ne bomo več našli rešitve, boljše od $g^\star$ (z drugimi besedami, pri tem u bi porabili že samo za nadgradnje toliko časa, kolikor ga pri najboljši rešitvi porabimo za nadgradnje in proizvodnjo skupaj).

Pri razmisleku v prejšnjem odstavku smo začeli z zmogljivostjo z in proizvodnjo $f_d(z)$. Ker vnaprej ne moremo vedeti, katera z bo dala najboljšo rešitev, moramo seveda v zanki preizkusiti vse (od $z_d^{\min}$ do $z_d^{\max}$).

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

int main()
{
  // Preberimo vhodne podatke.
  int k, d, r; cin >> k >> d;
  struct Omejitev { int x, y; }; // zahteva proizvodnjo y enot v prvih x dneh
  vector<Omejitev> omejitve; omejitve.reserve(d + 1); omejitve.resize(d);
  for (auto &O : omejitve) cin >> O.x >> O.y;
  omejitve.push_back({0, 0}); ++d; // Dodajmo še „omejitev“ (0, 0) na začetku.

  // Uredimo omejitve po času.
  sort(omejitve.begin(), omejitve.end(), [] (const auto &a, const auto &b) {
    return a.x < b.x || a.x == b.x && a.y > b.y; });

  // Izračunajmo možne kombinacije zmogljivosti in proizvodnje na koncu vsakega intervala.
  vector<int> f = { 0 }, ff;
  for (r = 1; r < d; ++r)
  {
    if (f.empty()) break;   // V f[z] so vrednosti f_{r−1}(z);
    ff.clear();             // v ff[z] bomo izračunali vrednosti f_r(z).
    const Omejitev &O = omejitve[r], &OP = omejitve[r − 1];
    int tr = O.x − OP.x;   // t_r = trajanje trenutnega intervala
    if (tr <= 0) continue; // Od omejitev z istim x upoštevamo le prvo (z največjim y).
    if ((tr / 2) * (tr − tr / 2) >= k − OP.y)
      break; // Če je interval dovolj dolg, bomo na njem gotovo dosegli k.
    int zMax = f.size() − 1; // z^max_{r−1}
    bool konec = false;
    for (int z = 0; z <= zMax + tr; ++z)
    {
      // Recimo, da hočemo na koncu intervala doseči zmogljivost z.
      // Izračunajmo največjo možno proizvodnjo.
      int p = −1;
      for (int u = max(0, z − zMax); u <= min(tr, z); ++u)

        // Preizkusimo možnost, da v tem intervalu izvedemo u nadgradenj.
        if (f[z − u] >= OP.y) p = max(p, f[z − u] + (tr − u) * z);

      if (p < O.y) p = −1; // nismo dosegli omejitve
      ff.push_back(p);
      if (p >= k) { konec = true; break; }
    }
    if (konec) break; // Ali se dá doseči proizvodnjo k?
```

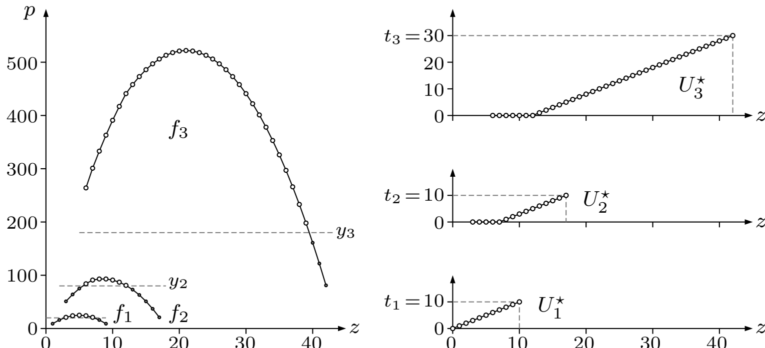
```
      while (! ff.empty() && ff.back() < 0) ff.pop_back();
      swap(f, ff);
   }
   // Zdaj imamo v f možne vrednosti proizvodnje bodisi na koncu zadnjega intervala
   // ali pa na koncu nekega takega intervala, kjer lahko v naslednjem že dosežemo k.
   // Vprašanje je torej le še, kako čim hitreje doseči k.
   int tMin = −1, tZdaj = omejitve[r − 1].x;
   for (int z = 0; z < f.size(); ++z)
      // Če začnemo v (z, p) in nadgrajujemo u dni, bomo porabili skupaj
      // f(u) = u + ⌈(k − p) / (z + u)⌉ dni. Poiščimo minimum tega.
      if (int p = f[z]; p >= 0) for (int u = (z == 0 ? 1 : 0); ; ++u)
      {
         // Mogoče pri tem u že samo nadgrajevanje traja dlje kot
         // nadgrajevanje + proizvodnja pri najboljši doslej znani rešitvi.
         // Če je tako, bo pri večjih u samo še slabše in lahko končamo.
         if (tMin >= 0 && tZdaj + u > tMin) break;
         int zSkupaj = z + u;
         int kand = tZdaj + u + (k − p + zSkupaj − 1) / zSkupaj;
         if (tMin < 0 || kand < tMin) tMin = kand;
      }
   cout << tMin << endl; return 0;
}
```

Tudi to rešitev bi se dalo na razne načine še izboljšati. Po urejanju omejitev lahko omejitve r, pri katerih je $x_r > x_{r-1}$ in $y_r \leq y_{r-1}$, preprosto zavržemo, saj vsak razpored, ki ustreže omejitvi $r-1$, gotovo ustreže tudi omejitvi r (ki ima kasnejši rok in nič višjo zahtevano proizvodnjo). Z zanko po z smo šli v gornji rešitvi od 0 do $z_{r-1}^{\max} + t_r$, toda lahko bi začeli pri $z_{r-1}^{\min}$ namesto pri 0, kajti če prvim $r-1$ omejitvam ni mogoče ustreči z manj kot toliko nadgradnjami, potem tudi prvim r-omejitvam ne bo; še več, lahko bi zanko začeli pri tistem z, kjer $f_{r-1}(z)$ doseže svoj maksimum, kajti pri manjših z vstopimo v r-ti interval z nižjo zmogljivostjo *in* nižjo proizvodnjo, od tega pa ne more biti nobene koristi. Še nekaj drugih izboljšav pa zahteva malo več razmisleka, da se prepričamo o njihovi pravilnosti, zato si jih bomo ogledali v naslednjih razdelkih.

Najboljši u pri izračunu $f_r(z)$. Naša dosedanja rešitev porabi največ časa za izračun vrednosti funkcije $f_r(z)$, ki je, kot se spomnimo, definirana kot $\max_u F_r(z)$, mi pa smo ta maksimum iskali z zanko po vseh primernih u, od $\max(0, z - z_{r-1}^{\max})$ do $\min(t_r, z - z_{r-1}^{\min})$. Toda izkaže se, da je ta maksimum vedno dosežen pri najmanjšem možnem u — recimo mu $U_r(z) := \max(0, z - z_{r-1}^{\max})$. Ko se bomo prepričali, da to res drži, se bomo lahko zanki po u odpovedali in tako prihranili ogromno časa.

Oglejmo si za začetek majhen primer: recimo, da imamo tri omejitve, $(10, 20)$, $(20, 80)$ in $(50, 120)$, in da po dosedanjem postopku izračunamo zanje funkcije f_1, f_2, f_3. Spomnimo se, da je $f_r(z) = \max_u F_r(z, u)$; zapomnimo si vsakič, pri katerem u je bil ta maksimum dosežen; recimo mu $U_r^\star(z)$; zanj je torej $f_r(z) = F_r(z, U_r^\star(z))$. Levi graf na naslednji sliki prikazuje funkcije f_r, pri čemer črni krožci predstavljajo tiste vrednosti, ki ležijo pod omejitvijo y_r in jih zato v nadaljevanju postopka (pri izračunu f_{r+1}) ne uporabljamo; vodoravna črtkana črta kaže proizvodnjo y_r. Grafi na desni strani naslednje slike pa prikazujejo funkcije U_r, ki nam povedo, koliko nadgradenj ima v r-tem intervalu najboljši razpored za $f_r(z)$; vodoravna črtkana

črta kaže $u = t_r$, navpična pa $z = \hat{z}_r^{\max}$.



Na levem grafu vidimo, kako lepo konkavne oblike so funkcije f_r; vsaka torej najprej nekaj časa le narašča (in to vse počasneje), nato pa odtlej le še pada (in to vse hitreje). Še zanimivejši pa so grafi na desni, kjer vidimo, kako preprosta je funkcija $U_r^\star(z)$: sprva je ves čas 0, vzpenjati pa se začne v korakih po 1 in to šele takrat, da potem najvišjo možno vrednost t_r doseže pri najvišjem možnem argumentu, $z = z_{r-1}^{\max} + t_r$. Z drugimi besedami, velja torej $U_r^\star(z) = \max\{0, z - z_{r-1}^{\max}\} = U_r(z)$, tako da bomo najboljši u res lahko računali po preprosti formuli za $U_r(z)$.

Prepričajmo se zdaj, da ta opažanja res držijo v splošnem. Vpeljimo zapis $\triangle f_r(z) := f_r(z-1) - f_r(z)$, ki nam pove, za koliko se vrednost funkcije zmanjša, ko se ji argument poveča z $z-1$ na z. Neformalni povzetek našega razmisleka je naslednji: recimo, da bi radi prvih r intervalov končali z natanko z nadgradnjami (in čim večjo proizvodnjo) in da razmišljamo o tem, da bi na zadnjem od teh intervalov izvedli kakšno nadgradnjo več, kot je nujno potrebno, npr. $u+1$ namesto le u nadgradenj. Z vsako tako dodatno nadgradnjo izgubimo na r-tem intervalu en dan proizvodnje, ko bi lahko proizvedli z enot; toda po drugi strani na predhodnih intervalih, kjer moramo zdaj izvesti eno nadgradnjo manj, morda kaj pridobimo — natančneje, pridobimo $\triangle f_{r-1}(z-u)$ enot. Vprašanje je torej, ali pridobimo več, kot smo izgubili; z drugimi besedami, ali funkcija f_{r-1} kdaj pada tako hitro, da s tem, ko se ji argument zmanjša z $z-u$ na $z-u-1$, njena vrednost naraste za z ali več. Pokazali bomo, da niti pri največji vrednosti argumenta ne pada tako hitro (čeprav je že zelo blizu tega); da je konkavne oblike, zato pri nižjih vrednostih argumenta pada še počasneje; in da iz tega res sledi, da je najmanjši dopustni u tudi najboljši.

Zapišimo zdaj naš razmislek bolj formalno. Spomnimo se, da je funkcija $f_r(z)$ definirana na območju $\hat{z}_r^{\min}$ od $\hat{z}_r^{\max}$. Trdimo, (a) da pri $z = \hat{z}_r^{\max}$ velja $\triangle f_r(z) = z-1$ in (b) da pri $\hat{z}_r^{\min} < z < \hat{z}_r^{\max}$ velja $\triangle f_r(z) \leq \triangle f_r(z+1) - 2$ (kar lahko zapišemo tudi kot $\triangle f_r(z+1) - \triangle f_r(z) \geq 2$); in še, (c) da pri vseh z velja $f_r(z) = F_r(z, U_r(z))$.

Preden se lotimo dokazovanja te trditve, omenimo njeno koristno posledico: če neenačbo $\triangle f_r(z) \leq \triangle f_r(z+1) - 2$ uporabimo po večkrat zapored, dobimo $\triangle f_r(z) \leq \triangle f_r(z+1) - 2 \leq \triangle f_r(z+2) - 4 \leq \triangle f_r(z+3) - 6$ in tako naprej; v splošnem torej $\triangle f_r(z) \leq \triangle f_r(z+w) - 2w$. Če gremo do $w = \hat{z}_r^{\max} - z$, dobimo

$$\begin{aligned} \triangle f_r(z) \leq\ & \triangle f_r(\hat{z}_r^{\max}) - 2(\hat{z}_r^{\max} - z) = \\ =\ & (\hat{z}_r^{\max} - 1) - 2(\hat{z}_r^{\max} - z) = 2z - \hat{z}_r^{\max} - 1. \end{aligned} \qquad (\dagger)$$

Lotimo se zdaj dokaza naše trditve. Dokazovali bomo z indukcijo po r; za začetek se prepričajmo, da trditev velja pri $r = 1$. Glede (c) je tako, da je v formuli $f_1(z) = \max_u F_1(z, u)$ tako ali tako mogoč en sam u, namreč $u = z$, saj moramo vseh z nadgradenj izvesti v edinem intervalu, ki ga imamo; najboljši in edini u je torej $u = z$, ravno tega pa nam tudi priporoči funkcija $U_1(z) = \max(0, z - z_0^{\max}) = \max(0, z - 0) = z$. (To, da je $z_0^{\max} = 0$, sledi iz dejstva, da v 0 intervalih ne moremo izvesti več kot 0 nadgradenj.) Pri $r = 1$ imamo torej $f_1(z) = F_1(z, z) = f_0(0) + (t_1 - z)z = (t_1 - z)z$, funkcija pa je definirana od $\hat{z}_1^{\min} = z_0^{\min} = 0$ do $\hat{z}_1^{\max} = z_1^{\max} + t_1 = t_1$. Ta formula za $f_1(z)$ nam dá $\triangle f_1(z) = f_1(z-1) - f_1(z) = (t_1 - (z-1))(z-1) - (t_1 - z)z = 2z - 1 - t_1$. Pri $z = t_1 = \hat{z}_1^{\max}$ imamo torej $\triangle f_1(z) = z - 1$, torej (a) drži. Pri manjših z pa imamo $\triangle f_1(z+1) - \triangle f_1(z) = [2(z+1) - 1 - t_1] - [2z - 1 - t_1] = 2$, torej velja tudi (b).

Recimo zdaj, da smo našo trditev že dokazali do vključno $r - 1$; preverili bi radi, da velja tudi za r. Oglejmo si najprej (c), ki govori o tem, pri katerem u je dosežen maksimum $\max_u F_r(z, u)$ v formuli za $f_r(z)$. Primerjajmo pri fiksnem z vrednosti $F_r(z, u)$ za dva zaporedna u: razlika med njima je $F_r(z, u+1) - F_r(u) = [f_{r-1}(z - u - 1) + (t_r - u - 1)z] - [f_{r-1}(z - u) + (t_r - u)z] = \triangle f_{r-1}(z - u) - z = (\star)$. Seveda nas pri izračunu $f_r(z) = \max_u F_r(z, u)$ zanimajo le taki u, ki so $\geq \max(0, z - z_{r-1}^{\max})$, zato je $z - u \leq z_{r-1}^{\max} \leq \hat{z}_{r-1}^{\max}$, zato lahko za $z - u$ uporabimo posledico (†) naše trditve, kajti le-ta po induktivni predpostavki že velja za $r - 1$; imamo torej $\triangle f_{r-1}(z - u) \leq 2(z - u) - \hat{z}_{r-1}^{\max} - 1$, zato pa $F_r(z, u+1) - F_r(u) = (\star) \leq 2(z-u) - \hat{z}_{r-1}^{\max} - 1 - z = (z - u - \hat{z}_{r-1}^{\max}) - (u+1)$; od teh dveh členov je prvi ≤ 0, ker je $z - u \leq \hat{z}_{r-1}^{\max}$, drugi pa je > 0, ker je $u \geq 0$; zato je razlika manjša od 0. Tako torej vidimo, da je $F_r(z, u+1) - F_r(u) < 0$, torej se $F_r(z, u)$ zmanjša vsakič, ko povečamo u za 1, tako da bo maksimum dosežen pri najmanjšem možnem u, to je pri $u = \max(0, z - z_{r-1}^{\max}) = U_r(z)$, torej (c) res velja.

S tem zdaj tudi vemo nekaj več o vrednostih funkcije $f_r(z)$. Na območju $z_{r-1}^{\max} \leq z \leq z_{r-1}^{\max} + t_r$ imamo $U_r(z) = z - z_{r-1}^{\max}$, na območju $z_{r-1}^{\min} \leq z \leq z_{r-1}^{\max}$ pa $U_r(z) = 0$. Če to vstavimo v $f_r(z) = F_r(z, U_r(z)) = f_{r-1}(z - U_r(z)) + (t_r - U_r(z))z$, dobimo:

$$f_r(z) = \begin{cases} f_{r-1}(z_{r-1}^{\max}) + (t_r - z + z_{r-1}^{\max})z & : z_{r-1}^{\max} \leq z \leq z_{r-1}^{\max} + t_r \\ f_{r-1}(z) + t_r z & : z_{r-1}^{\min} \leq z \leq z_{r-1}^{\max}. \end{cases}$$

Če potem izračunamo razliko dveh zaporednih $f_r(z)$, dobimo:

$$\triangle f_r(z) = \begin{cases} 2z - 1 - t_r - z_{r-1}^{\max} & : z_{r-1}^{\max} < z \leq z_{r-1}^{\max} + t_r \\ \triangle f_{r-1}(z) - t_r & : z_{r-1}^{\min} < z \leq z_{r-1}^{\max}. \end{cases}$$

Pri $z = \hat{z}_r^{\max} = z_{r-1}^{\max} + t_r$ nam ta formula pove, da je $\triangle f_r(z) = z_{r-1}^{\max} + t_r - 1 = z - 1$, torej velja (a). — Pri z z območja $z_{r-1}^{\max} < z < z_{r-1}^{\max} + t_r$ lahko izračunamo $\triangle f_r(z+1) - \triangle f_r(z) = \ldots = 2$, torej tu velja (b). — Pri $z = z_{r-1}^{\max}$ lahko izračunamo $\triangle f_r(z+1) - \triangle f_r(z) = \ldots = z_{r-1}^{\max} + 1 - \triangle f_{r-1}(z_{r-1}^{\max}) = (\star)$; ker po induktivni predpostavki naša trditev velja za $r-1$, nam (†) pove, da je $\triangle f_{r-1}(z_{r-1}^{\max}) \leq 2z_{r-1}^{\max} - \hat{z}_{r-1}^{\max} - 1$; zato je $(\star) \geq z_{r-1}^{\max} + 1 - [2z_{r-1}^{\max} - \hat{z}_{r-1}^{\max} - 1] = 2 + (\hat{z}_{r-1}^{\max} - z_{r-1}^{\max})$, kar je gotovo ≥ 2, saj je $z_{r-1}^{\max}$ po definiciji $\leq \hat{z}_{r-1}^{\max}$; tako torej vidimo, da (b) velja tudi pri $z = z_{r-1}^{\max}$. — In končno, pri z z območja $z_{r-1}^{\min} < z < z_{r-1}^{\max}$ lahko izračunamo $\triangle f_r(z+1) - \triangle f_r(z) = [f_{r-1}(z+1) - (z+1)] - [f_{r-1}(z) - z] = [f_{r-1}(z+1) - f_{r-1}(z)] + 1$;

že prvi člen sam je po induktivni predpostavki (ker naša trditev velja za $r-1$) večji ali enak 2, torej velja (b) tudi tukaj. □

Ta razmislek je koristen še iz nekega drugega razloga poleg tega, da nam omogoča izogniti se notranji zanki po u pri izračunu $f_r(z)$: ob njem smo namreč potek funkcije f_r spoznali dovolj dobro, da lahko izpeljemo zgornjo mejo vrednosti $\hat{z}_r^{\max}$. Naj bo $z^\star$ tista vrednost z-ja, kjer doseže funkcija $f_r(z)$ svoj maksimum. Če je ta maksimum $\geq k$, se bo naš postopek tako ali tako nemudoma ustavil in preostanka funkcije niti ne bo računal; omejimo se zdaj na primer, ko je maksimum vendarle $< k$. Če zdaj z postopoma povečujemo od $z^\star$ navzgor, začne funkcija padati; in to padanje je vse hitrejše, saj smo videli, da je $\triangle f_r(z+1) \geq \triangle f_r(z) + 2$. Prvi padec je torej vsaj za 2, naslednji vsaj za 4, še naslednji vsaj za 6 in tako naprej. Po w korakih funkcija pade že vsaj za $2 + 4 + \ldots + 2w = w(w+1)$. Toda ker je bila vrednost funkcije že na maksimumu $< k$ in ker nikoli ne more pasti pod 0, morajo biti tudi vsi padci skupaj manjši od k; torej imamo $w(w+1) < k$, torej $w \leq \sqrt{k}$.

Podobno je tudi, če z postopoma zmanjšujemo od $z^\star$ navzdol; ker velja neenakost $\triangle f_r(z-1) \leq \triangle f_r(z) - 2$, pada funkcija vse hitreje, ko se z zmanjšuje. Enak razmislek kot prej nam tudi tokrat pokaže, da imamo lahko le $\sqrt{k}$ padcev.

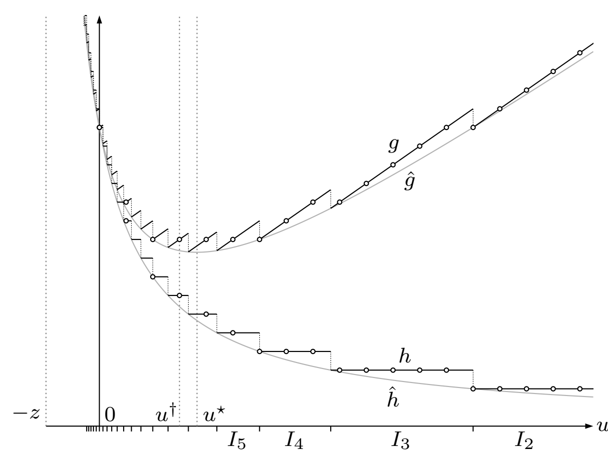
Zaradi zveze $\triangle f_r(z) \leq \triangle f_r(z+1) - 2$ tudi ni mogoče, da bi imela f_r pri treh zaporednih z-jih enako vrednost. Možen razpon z-jev, pri katerih je f_r sploh definirana, je torej širok le $2\sqrt{k} + O(1)$ elementov. Ker so poleg tega vsi z-ji pozitivni, lahko zaključimo, da je $\hat{z}_r^{\max} \leq 2\sqrt{k} + O(1)$.

Najboljše število nadgradenj po koncu zadnje omejitve. Spomnimo se, da se moramo po tistem, ko dosežemo konec zadnjega intervala, odločiti še, koliko nadgradenj naj izvedemo po njem, da bomo potem čim hitreje dosegli želeno proizvodnjo k. Videli smo že, da pri tem iščemo minimum funkcije $g(u) = u + h(u)$ za $h(u) = \lceil (k-p)/(z+u) \rceil$, kjer je z dosedanja zmogljivost, $p = f_d(z)$ pa dosedanja proizvodnja; funkcija h torej pove, koliko dni proizvodnje potrebujemo, da dosežemo ali presežemo k. Doslej smo minimum g-ja iskali z zanko po u, v tem razdelku pa si bomo ogledali, kako lahko to počnemo učinkoviteje. Da bo manj pisanja, pišimo $\tilde{p} := k - p$; to je torej proizvodnja, ki nam še manjka do k.

Če ne bi bilo tistega $\lceil \cdot \rceil$, bi imeli funkcijo $\hat{g}(u) = u + \hat{h}(u)$ za $\hat{h}(u) = \tilde{p}/(z+u)$; in takšna $\hat{g}$ je odvedljiva: $\hat{g}'(u) = 1 - \tilde{p}/(z+u)^2$, kar je enako 0 pri $u^\star = \sqrt{\tilde{p}} - z$. Tam ima funkcija $\hat{g}$ svoj minimum; če se oddaljujemo levo ali desno od $u = u^\star$, vrednost funkcije $\hat{g}$ narašča.

Ker sta si funkciji $\hat{g}$ in g tako podobni, bi pričakovali, da bomo tudi minimum funkcije $g(u)$ našli nekje v bližini $u^\star$. Omejiti se moramo seveda na celoštevilske u, saj nam u pomeni število nadgradenj, to pa je vedno celo. Obetaven kandidat je torej $u^\dagger := \lfloor u^\star \rfloor = \lfloor \sqrt{\tilde{p}} \rfloor - z$. Prepričajmo se, da funkcija $g(u)$ res doseže svoj minimum ravno pri $u = u^\dagger$.

Razmislimo najprej, kaj se dogaja s funkcijo g pri majhnih u. Ko se u zmanjšuje, se $h(u)$ in $\hat{h}(u)$ povečujeta, in to še celo vse hitreje. Ker nas zanimajo le celoštevilski u, si oglejmo, kaj se zgodi, ko se premaknemo iz u v $u-1$. Tam se $\hat{h}$ poveča s $\hat{h}(u)$ na $\hat{h}(u-1)$ in če se tidve vrednosti razlikujeta za ≥ 1, potem gotovo nimata istega $\lceil \cdot \rceil$; takrat je torej $h(u-1) > h(u)$, torej se je v funkciji $g(u) = u + h(u)$ prvi seštevanec sicer zmanjšal za 1, drugi pa povečal za vsaj 1, torej se vrednost g pri tem ni zmanjšala: $g(u-1) \geq g(u)$. Takrat torej pri iskanju minimuma funkcije



Primer za $z = 2$, $\tilde{p} = 32$. Graf kaže funkcije (od zgoraj navzdol) g, $\hat{g}$, h in $\hat{h}$ (g in h sta narisani z debelejšima črnima črtama, $\hat{g}$ in $\hat{h}$ pa s tanjšima sivima). Krožci kažejo vrednosti $g(u)$ in $h(u)$ za celoštevilske u. Navpične črtkane črte kažejo $u = -z$ (kjer imajo omenjene funkcije pol), $u = u^\dagger$ (kjer ima g svoj minimum) in $u = u^\star$ (kjer ima $\hat{g}$ svoj minimum). Oznake na vodoravni osi kažejo intervale I_h, na katerih je $h(u) = h$.

$g(u)$ nima smisla iti z u na $u - 1$, kaj šele dlje v levo. Kdaj točno se to zgodi? Videli smo, da takrat, ko je $\hat{h}(u-1) - \hat{h}(u) \geq 1$; pišimo $w = z + u$, pa naš pogoj postane $\tilde{p}/(w-1) - \tilde{p}/w \geq 1$, torej $p \geq w(w-1)$, torej $w^2 - w - \tilde{p} \leq 0$, kar je pri $w \leq \frac{1}{2} + \sqrt{\tilde{p} + 1/4}$, torej $u \leq \frac{1}{2} + \sqrt{\tilde{p} + 1/4} - z$. Naš $u^\dagger$ temu pogoju ustreza, kar pomeni, da manjših u-jev od njega ni treba gledati.

Kaj pa večji u-ji? Oglejmo si funkcijo $h(u)$ še malo pobliže. Zaradi $\lceil\cdot\rceil$ je ta funkcija odsekoma konstantna; pri katerih u je dosežena neka konkretna vrednost h? Z malo telovadbe se hitro vidi, da je to pri $u \in I_h := [\tilde{p}/h - z, \tilde{p}/(h-1) - z)$. Vodoravno os u lahko v mislih razdelimo na takšne intervale: $\ldots, I_{h+1}, I_h, I_{h-1}, \ldots, I_1$; vsak naslednji je širši od prejšnjega, zadnji od njih pa se celo razteza na desni v neskončnost. Na vsakem takem intervalu ima $h(u) = h$ konstantno vrednost, zato bo $g(u) = u + h(u)$ svojo najmanjšo vrednost (na tem intervalu) dosegla pri najmanjšem u s tega intervala. Toda spomnimo se, da nas zanimajo le celoštevilski u. Če je I_h širok vsaj 1 enoto, vsebuje gotovo vsaj eno celo število; naj bo u_h najmanjše celo število na tem intervalu; minimum funkcije g na tem intervalu je torej $u_h + h$. Na naslednjem intevalu, I_{h-1}, ki je še širši od I_h in torej gotovo tudi vsebuje vsaj eno celo število, je najmanjše celo število recimo u_{h-1}, ki je gotovo vsaj za 1 večje od u_h; najmanjša vrednost funkcije g na tem intervalu je potem $u_{h-1} + (h-1) \geq (u_h + 1) + (h-1) = u_h + h$, torej minimum g-ja na I_{h-1} ni nič boljši kot na I_h. To pomeni, da čim dosežemo interval I_h, ki je širok vsaj 1 enoto, naslednjih intervalov (torej manjših h) ni več treba gledati. Pri katerih h pa nastopijo takšni intervali širine vsaj 1? I_h je širok $\tilde{p}(1/(h-1) - 1/h) = \tilde{p}/[h(h-1)]$, kar je ≥ 1 pri $h \leq h_D := \frac{1}{2} + \sqrt{\tilde{p} + 1/4}$. Opazimo, da je $h_D > \sqrt{\tilde{p}}$, zato bomo namesto pogoja $h \leq h_D$ uporabljali strožji, a preprostejši pogoj $h \leq \sqrt{\tilde{p}}$.

Kmalu bomo videli, da pri $h = h(u^\dagger)$ ta pogoj še ni nujno izpolnjen; preden pridemo do dovolj majhnih h, moramo u še malo povečati. Toda videli bomo tudi, da pri tem povečevanju funkcija $g(u)$ nikoli ne doseže vrednosti, manjše od $g(u^\dagger)$. Pišimo $q := \lfloor\sqrt{\tilde{p}}\rfloor$; z drugimi besedami, q je celo število, za katero velja $q^2 \leq \tilde{p} < (q+1)^2$. Potem je $u^\dagger = q - z$ in $h(u^\dagger) = \lceil\tilde{p}/q\rceil$. V nadaljevanju razmisleka bomo

območje $q^2 \le \tilde{p} < (q+1)^2$ razdelili na tri dele in obravnavali vsakega posebej:

(a) Če je $\tilde{p} = q^2$, je $\tilde{p}/q = q$, torej celo število, zato $h(u^\dagger) = \lceil \tilde{p}/q \rceil = q$; takrat torej že $h = h(u^\dagger)$ ustreza pogoju $h \le \sqrt{\tilde{p}}$, zato je I_h širok vsaj 1 in pri večjih u ne bomo našli nobene manjše vrednosti funkcije g.

(b) Če je $q^2 < \tilde{p} \le q(q+1)$, je $q < \tilde{p}/q \le q+1$, torej je $h(u^\dagger) = \lceil \tilde{p}/q \rceil = q+1$. Ker je $(q-1)(q+1) = q^2 - 1 < q^2$, velja tudi $q - 1 < q^2/(q+1) < \tilde{p}/(q+1) \le q$, zato je $h(u^\dagger + 1) = \lceil \tilde{p}/(q+1) \rceil = q$. Pri premiku iz $u^\dagger$ v $u^\dagger + 1$ se je torej u za 1 povečal, h pa za 1 zmanjšal, zato je vrednost $g(u)$ ostala nespremenjena; pri $u = u^\dagger + 1$ pa $h(u)$ že ustreza pogoju $h \le \sqrt{p}$, torej je interval I_h širok vsaj 1 in od tu naprej ne bomo našli nobene manjše vrednosti funkcije g.

(c) Ostane še primer, ko je $q(q+1) < \tilde{p} < (q+1)^2$. Ker sta $\tilde{p}$ in q cela, lahko desno neenakost zapišemo kot $\tilde{p} \le (q+1)^2 - 1 = q(q+2)$. Velja torej $q+1 < \tilde{p}/q \le q+2$, zato je $h(u^\dagger) = \lceil \tilde{p}/q \rceil = q+2$. Velja tudi $q < \tilde{p}/(q+1) < q+1$, zato je $h(u^\dagger + 1) = \lceil \tilde{p}/(q+1) \rceil = q+1$. Velja pa tudi $(q-1)(q+2) = q^2 + q - 2 < q(q+1) < \tilde{p} \le q(q+2)$, zato $q - 1 < \tilde{p}/(q+2) \le q$, tako da je $h(u^\dagger + 2) = \lceil \tilde{p}/(q+2) \rceil = q$. Pri premiku iz $u^\dagger$ v $u^\dagger + 1$ in nato iz $u^\dagger + 1$ v $u^\dagger + 2$ se torej u vsakič poveča za 1, h pa zmanjša za 1, zato $g(u)$ ostane nespremenjena; pri $u = u^\dagger + 2$ pa imamo že $h(u) = q \le \sqrt{\tilde{p}}$, torej smo na intervalu I_h širine vsaj 1 in vemo, da od tu naprej ne bomo našli nobene manjše vrednosti funkcije g.

V vseh treh primerih torej vidimo, da pri $u > u^\dagger$ funkcija g nikoli ne pade pod vrednost $g(u^\dagger)$, torej smo pri iskanju minimuma funkcije g lahko zadovoljni že z $u^\dagger$. □

Zdaj torej vemo, da doseže $g(u)$ svoj minimum pri $u = u^\dagger = \lfloor \sqrt{k-p} \rfloor - z$; lahko pa se zgodi, da je to število negativno (če dosedanja zmogljivost z dovolj velika v primerjavi z manjkajočo proizvodnjo $k - p$), mi pa negativnega števila nadgradenj ne moremo izvesti, zato moramo takrat pač vzeti $u = 0$. Tako lahko v naši rešitvi zanko, v kateri smo proti koncu programa pregledovali različne u in iskali najmanjšo vrednost $g(u)$, zamenjamo z enim samim stavkom, ki preprosto izračuna $u = \max\{0, \lfloor \sqrt{k-p} \rfloor - z\}$.

Zgornja meja za število nadgradenj v najboljšem razporedu. Če ne bi imeli nobenih omejitev, bi lahko k odmerkov cepiva proizvedli tako, da bi najprej u dni nadgrajevali in nato $\lceil k/u \rceil$ dni proizvajali, torej v skupaj $g(u) = u + \lceil k/u \rceil$ dneh. V prejšnjem razdelku smo videli, da doseže ta funkcija minimum pri $u = \lfloor \sqrt{k} \rfloor$. Več kot toliko nadgradenj torej ne potrebujemo.

Ali bi se to kaj spremenilo, če bi imeli v nalogi tudi omejitve? Prepričali se bomo, da tudi takrat ni treba več kot $\lfloor \sqrt{k} \rfloor$ nadgradenj. Mislimo si neki nabor omejitev in zanj poiščimo najboljšo rešitev, torej táko, ki doseže proizvodnjo k v najmanj dneh; če je takih več, pa vzemimo med njimi tisto, ki ima najmanjše število nadgradenj, recimo z. Po zadnji nadgradnji nastopi v njej gotovo še nekaj proizvodnih dni, saj sicer od tiste zadnje nadgradnje ni bilo nobene koristi in bi se dalo rešitev za en dan skrajšati, če bi se tistemu zadnjemu dnevu z nadgradnjo popolnoma odpovedali.

Recimo torej, da po zadnji nadgradnji pride še t proizvodnih dni. Ali je mogoče, da je $t < z$? Potem bi lahko namesto z-te nadgradnje tisti dan proizvajali in takrat proizvedli $z - 1$ odmerkov cepiva; v vsakem od naslednjih t dni pa bi po novem proizvedli en odmerek manj kot prej. Skupna proizvodnja od začetka razporeda do vključno i-tega od tistih t proizvodnih dni bi se torej povečala za $z - 1 - i$, kar

je ≥ 0, torej bi novi razpored še vedno ustrezal vsem omejitvam, ki jim je tudi prvotni. Zato je novi razpored še vedno optimalen (enako dolg kot prvotni), ima pa eno nadgradnjo manj; toda mi smo na začetku predpostavili, da je imel prvotni razpored najmanjše možno število nadgradenj (med vsemi optimalnimi razporedi). Tako smo v protislovju, torej mora biti $t \geq z$.

Za zadnjo, z-to nadgradnjo torej pride še vsaj z dni proizvodnje, v njih pa se proizvede vsaj z^2 odmerkov cepiva. Pišimo zdaj $q = \lfloor\sqrt{k}\rfloor$, tako da k leži na območju $q^2 \leq k < (q+1)^2$. Če je $z \leq q$, potem trditev, ki jo dokazujemo (namreč da za optimalno rešitev ni treba več kot $\lfloor\sqrt{k}\rfloor$ nadgradenj), velja tudi za naš trenutni nabor omejitev, torej je vse v redu. Ali se lahko zgodi, da bi bilo $z \geq q+2$? To bi pomenilo, da za zadnjo, $(q+2)$-go nadgradnjo pride še vsaj $q+2$ dni proizvodnje, takrat pa se proizvede $(q+2)^2$ enot cepiva; toda če bi se enemu od teh dni proizvodnje odpovedali, bi se takrat še vedno proizvedlo $(q+2)(q+1)$ enot, kar je že sámo po sebi večje od k; torej prvotni razpored ni mogel biti optimalen, kar je protislovje.

Tako torej $z \geq q+2$ ni mogoče; kaj pa $z = q+1$? Tedaj pride po zadnji, $(q+1)$-vi nadgradnji še vsaj $q+1$ dni proizvodnje; toda več kot $q+1$ jih ne more biti, saj že v $q+1$ dneh proizvedemo $(q+1)^2$ odmerkov, kar je več kot k; če bi imeli $q+2$ ali več dni proizvodnje, bi lahko razpored skrajšali in še vedno ustregli vsem omejitvam, kar pa je nemogoče, saj smo začeli z optimalnim razporedom. Po zadnji nadgradnji imamo torej natanko $q+1$ dni proizvodnje.

Skupna proizvodnja v teh dneh je $(q+1)^2$, kar je $\geq k+1$. Če bi zdaj $(q+1)$-vo nadgradnjo zamenjali z dnevom proizvodnje, bi takrat proizvedli q odmerkov in za vsakega od naslednjih q proizvodnih dni bi veljalo, da je skupna proizvodnja od začetka razporeda do konca tistega dne vsaj tolikšna kot pred to spremembo. Po zadnjem, $(q+1)$-vem proizvodnem dnevu iz te skupine pa bi bila zdaj skupna proizvodnja za 1 manjša kot prej (v vsakem od $q+1$ starih proizvodnih dni smo izgubili eno enoto proizvodnje, pred tem pa smo pridobili q enot na tisti dan, ko smo nadgradnjo zamenjali s proizvodnjo). Toda to je tudi zadnji dan celotnega razporeda in omejitev takrat gotovo ni mogla biti tesna, saj bi v tem primeru ta omejitev znašala vsaj $(q+1)^2$, to pa je večje od k. Zato novi razpored gotovo še vedno ustreza vsem omejitvam, čeprav ima skupno proizvodnjo za 1 manjšo kot prej. Je enako dolg kot prvotni, torej je tudi novi razpored optimalen; ima pa tudi eno nadgradnjo manj kot prvotni, kar pa je v protislovju s predpostavko, da smo imeli že prej najmanjše možno število nadgradenj. Tako nas torej tudi $z = q+1$ pripelje v protislovje in lahko zaključimo, da je $z \leq q$. Z drugimi besedami, če ima naloga pri danem k in danem naboru omejitev sploh kakšno rešitev, potem med optimalnimi razporedi (takimi z najmanjšim številom dni) gotovo obstaja kak tak, ki ima kvečjemu $\lfloor\sqrt{k}\rfloor$ nadgradenj. □

To opažanje lahko uporabimo za drobno izboljšavo v naši rešitvi: doslej smo računali vrednosti funkcije $f_r(z)$ vse do $\hat{z}_r^{\max} = z_{r-1}^{\max} + t_r$, zdaj pa vidimo, da gotovo ne bomo spregledali nobene optimalne rešitve, če razporede z več kot $\lfloor\sqrt{k}\rfloor$ nadgradnjami ignoriramo. Dovolj je že, če tam, kjer imamo zdaj za nadaljevanje zanke po z pogoj `z <= zMax + tr`, dodamo še `&& z * z <= k`.

Prav veliko koristi od te izboljšave sicer ni, saj smo prej videli, da je $\hat{z}_r^{\max} \leq 2\sqrt{k} + O(1)$, torej smo zgornjo mejo naše zanke zdaj približno razpolovili (z $2\sqrt{k}$ na $\sqrt{k}$), red zahtevnosti pa je ostal enak.

Zgornja meja za dolžino najboljšega razporeda v odvisnosti od k in d. Pri danih k in d obstaja veliko primerov naše naloge; ti se razlikujejo med seboj po svojih omejitvah in imajo zato lahko tudi različne rešitve, torej porabijo različno veliko dni, da dosežejo proizvodnjo k ob upoštevanju vseh omejitev. Poskusimo najti neko zgornjo mejo za to; vprašajmo se torej, koliko dni porabimo za proizvodnjo k enot v najslabšem primeru, če imamo d omejitev, ki so razporejene na najbolj neugoden način.

Preden se lotimo tega vprašanja, zapišimo drobno opažanje, ki nam bo prišlo kasneje prav: recimo, da najboljši razpored vstopi v nek interval pri zmogljivosti z, interval pa je dolg t dni; če izvede razpored v tem intervalu še u nadgradenj, bo v preostanku intervala proizvedel $P(u) := (t-u)(z+u)$ enot cepiva. Odvod te funkcije je $P'(u) = t - z - 2u$, kar je enako 0 pri $u = (t - z)/2$; ker pa nas negativni u tu ne zanimajo, lahko zaključimo, da je največja proizvodnja na tem intervalu dosežena pri $u = \max\{0, \lceil (t - z)/2 \rceil\}$ nadgradnjah. Manj kot toliko nadgradenj se zagotovo ne splača narediti, ker bosta v tem primeru na koncu intervala tako proizvodnja kot zmogljivost manjši ali enaki kot pri u nadgradnjah. Omejitve na koncu kasnejših intervalov nas morda lahko prisilijo v to, da izvedemo več kot u nadgradenj, ne pa manj kot toliko.

Razmislimo torej zdaj o zgornji meji za dolžino najboljšega razporeda. Omejimo se za začetek na naloge, pri katerih ni odvečnih omejitev, torej imamo $x_1 < x_2 < \ldots < x_d$ in $y_1 < y_2 < \ldots < y_d$, poleg tega pa predpostavimo še, da je $k = y_d$ in da te proizvodnje ni mogoče doseči prej kot na dan x_d.

Naših d omejitev nam razdeli časovno premico na d intervalov; v vsakem je pri optimalni rešitvi najprej nekaj (lahko tudi nič) nadgradenj in nato nekaj (lahko tudi nič) proizvodnih dni. Izberimo si neko število a; intervalom, pri katerih je zmogljivost ob koncu večja ali enaka a, bomo rekli *visoko produktivni* intervali, ostalim pa *nizko produktivni*. V visoko produktivnih intervalih se na vsak dan, ko poteka proizvodnja (in ne nadgradnja), izdela vsaj a izdelkov; zato je takih dni največ $\lceil k/a \rceil$. Poleg tega je v visoko produktivnih intervalih vsega skupaj tudi največ $\lfloor \sqrt{k} \rfloor$ dni nadgrajevanja, saj smo videli, da optimalna rešitev za določen k ne potrebuje več kot toliko nadgradenj.

Razmislimo zdaj še o nizko produktivnih intervalih; recimo, da jih je d_N (velja seveda $d_N \leq d$). V nekaterih od njih mogoče potekajo nadgradnje; takih intervalov je recimo q. Naj bo a_r število nadgradenj v r-tem od njih, naj bo $z_r = a_1 + \ldots + a_r$ skupno število nadgradenj v prvih r takih intervalih in naj bo t_r dolžina r-tega od njih. V r-tega torej vstopimo z zmogljivostjo z_{r-1}, zgoraj pa smo videli, da v takem primeru optimalna rešitev gotovo ne izvede manj kot $(t_r - z_{r-1})/2$ nadgradenj; torej je $a_r \geq (t_r - z_{r-1})/2$, torej $t_r \leq 2a_r + z_{r-1} = a_r + z_r$, torej je skupna dolžina teh intervalov $\sum_r t_r \leq \sum_r a_r + \sum_r z_r$; prva od zadnjih dveh vsot je enaka skupnemu številu nadgradenj v nizko produktivnih intervalih, kar je $< a$; druga vsota ima q členov, vsak od njih pa je $< a$; tako imamo $\sum_r t_r \leq a + q(a - 1) = qa$.

Ostanejo še nizko produktivni intervali brez nadgradenj. Vsak od njih je lahko dolg kvečjemu a dni, saj bi sicer pri vstopni zmogljivosti $z < a$ in dolžini, večji od a in s tem tudi večji od z, v njem gotovo prišlo do vsaj ene nadgradnje.

Vsi nizko produktivni intervali skupaj so torej dolgi kvečjemu $qa + (d_N - q)a = d_N a \leq da$ dni; vsi visoko produktivni intervali skupaj pa so dolgi kvečjemu $k/a +$

$\sqrt{k}+2$ dni (prišteli smo 2, da se nam ni treba ukvarjati z zaokrožanjem navzgor pri k/a in $\sqrt{k}$). Skupna dolžina razporeda, recimo ji T, je torej $T \leq da + k/a + \sqrt{k} + 2$. To velja za poljuben a; predstavljajmo si desno stran te neenačbe kot funkcijo a-ja in jo odvajajmo: dobimo $f'(a) = d - k/a^2$, kar je enako 0 pri $a = \sqrt{k/d}$, takrat pa dobimo $T \leq 2\sqrt{kd} + \sqrt{k} + 2$, torej $T = O(\sqrt{kd})$. □

Da je ta meja vsaj v asimptotičnem smislu tesna, se lahko prepričamo tako, da sestavimo primeren nabor omejitev, pri katerem bo najboljša rešitev res porabila $O(\sqrt{kd})$ dni. Pri naših poskusih, da bi za različne k in d našli tak nabor omejitev, pri katerem bi optimalna rešitev zahtevala čim več časa (več o tem v naslednjem razdelku), je največ časa vedno zahteval nabor omejitev naslednje oblike: naj bo $b := \sqrt{k/(d+3)}$; imejmo najprej en interval dolžine $2b$ z zahtevano proizvodnjo b^2; nato imejmo $d-2$ intervala dolžine b z zahtevano proizvodnjo b^2 v vsakem intervalu; in končno imejmo še en interval dolžine $3b$ z zahtevano proizvodnjo $4b^2$. Tako imamo d intervalov s skupno proizvodnjo $(d+3)b^2$, kar je ravno k; rešitev izvede b nadgradenj v prvem in še b v zadnjem intervalu, drugače pa ves čas le proizvaja; in trajanje tega razporeda je $(d+3)b = \sqrt{k(d+3)}$ dni.

Na začetku tega razdelka smo se omejili na naloge brez odvečnih omejitev, s $k = y_d$ in pri katerih tega k ni mogoče doseči prej kot na dan x_d. Če dodamo odvečne omejitve, s tem seveda najboljši razpored ni nič daljši, kot če teh omejitev ne bi bilo, torej zanj velja meja za d' namesto d, če je d' število ne-odvečnih omejitev; in ker je naša meja $2\sqrt{kd}$ naraščajoča funkcija d-ja, ne moremo ničesar pridobiti, če namesto d vzamemo manjšo vrednost d'. Podobno je, če imamo nabor omejitev, v katerem je mogoče doseči k prej kot na dan x_d; bodisi lahko rok te zadnje omejitve skrajšamo ali pa, če lahko k dosežemo celo na dan x_{d-1} ali prej, lahko zadnjo omejitev pobrišemo, saj je odvečna; tako smo spet pri manjšem številu omejitev, s tem pa, kot smo videli, ne moremo ničesar pridobiti (pri iskanju nabora omejitev, ki bi kot optimalno rešitev zahteval čim daljši razpored). In končno, če imamo $k > y_d$ in je mogoče proizvodnjo k doseči šele po roku zadnje omejitve, torej po x_d, je učinek tak, kot da bi imeli $d+1$ omejitev namesto d, pri čemer bi nova omejitev zahtevala proizvodnjo k z najzgodnejšim rokom, na katerega je to proizvodnjo mogoče doseči. Našo zgornjo mejo za trajanje razporeda pri najbolj neugodnem razporedu lahko še vedno uporabimo, le da moramo vzeti $d+1$ namesto d; dobimo $2\sqrt{k(d+1)}$. Med to mejo in $2\sqrt{kd}$ pa tako ali tako ni velike razlike: $2\sqrt{k(d+1)} = 2\sqrt{kd}\sqrt{1+1/d}$, tale zadnji faktor pa je blizu 1 in to tem bliže, čim večji je d.

Iskanje čim daljšega optimalnega razporeda pri danih k in d z dinamičnim programiranjem. V prejšnjem razdelku smo izpeljali zgornjo mejo za dolžino optimalnega razporeda v odvisnosti k in d v najslabšem primeru, torej po vseh možnih naborih omejitev za ta k in d. Zdaj pa si oglejmo še, kako lahko za dana k in d z dinamičnim programiranjem poiščemo konkreten primer čim daljšega optimalnega razporeda oz. nabora omejitev, pri katerem ta razpored nastane.

Omejili se bomo na primere, ko ni odvečnih omejitev (in imamo $x_1 < x_2 < \ldots < x_d$ in $y_1 < y_2 < \ldots < y_d$) in so vse omejitve tesne (optimalni razpored ravno doseže zahtevano proizvodnjo, preseže pa je ne) in je zadnja omejitev ravno enaka $y_d = k$, optimalni razpored pa jo doseže na dan x_d. (Če bi hoteli pokriti tudi primere, ko je $k > y_d$, bi lahko preprosto vzeli za 1 večji d in si mislili, da je tam na koncu še ena

omejitev več.)

Naj bo $h(z, \ell, d)$ najmanjši p, pri katerem je mogoče sestaviti tak nabor d omejitev (pod pogoji iz prejšnjega odstavka), ki bo imel $y_d = p$, njegova optimalna rešitev bo dolga $\ell = x_d$ dni, od tega pa bo porabila z dni za nadgradnje, proizvedla pa bo natanko p enot cepiva. Potem pravzaprav iščemo (pri danih k in d) največji tak ℓ, pri katerem za neki z velja $h(z, \ell, d) \leq k$.

Teh reči ni težko računati po naraščajočih d. Pri $d = 0$ je problem rešljiv le za $z = \ell = 0$, ko imamo $h(0, 0, 0) = 0$; drugod pa si mislimo $h(z, \ell, 0) = \infty$. Nato pa, ko poznamo rešitve za d, lahko razmišljamo takole: nabor omejitev, ki nas je pripeljal do $h(z, \ell, d)$, lahko podaljšamo s še enim intervalom dolžine t; na njem se bo izvedlo še $u := \max\{0, \lceil (t - z)/2 \rceil\}$ nadgradenj[7] in proizvedlo še $q := (t - u)(z + u)$ enot cepiva; tako torej vemo, da je vrednost $h(z, \ell, d) + q$ kandidat za vrednost $h(z + u, \ell + t, d + 1)$. To moramo ponoviti za vse z in ℓ, pri katerih je stanje (z, ℓ, d) dosegljivo, torej pri katerih je $h(z, \ell, d) < \infty$; pri vsakem pa moramo preizkusiti t-je do 1 naprej tako daleč, dokler $h(z, \ell, d) + q$ ne preseže k, kajti nabori omejitev, ki zahtevajo več kot k enot proizvodnje, nas ne bodo zanimali. Če se nam pri istem stanju $(z', \ell', d+1)$ nabere več kandidatov, obdržimo med njimi seveda najmanjšega (tistega z najmanjšo skupno proizvodnjo); koristno si je zraven tudi zapisati, pri kateri vrednosti z in t smo ga dobili — tako bomo lahko na koncu tudi rekonstruirali nabor omejitev, ki pripelje do tega stanja.

Kot smo omenili že v prejšnjem razdelku, so najbolj neugodni nabori omejitev, ki smo jih s tem postopkom našli, zahtevali rešitve dolžine približno $\sqrt{k(d + 3)}$ dni. Pri $k = 10^6$ in $d = 101$ (kar ustreza omejitvam iz naše naloge, pri čemer smo vzeli 101 namesto 100) nastane tako nabor omejitev, pri katerem je najboljša rešitev dolga 10198 dni.

Časovna zahtevnost naše rešitve. Razmislimo za konec še o časovni zahtevnosti naše rešitve s strani 83. V prvotni različici moramo najprej za vsako omejitev r izračunati $f_r(z)$ vse do $z = \hat{z}_r^{\max}$, pri vsakem z pa imamo zanko po u, ki izvede v najslabšem primeru $O(t_r)$ iteracij. Spomnimo se, da je $\hat{z}_r^{\max} = O(\sqrt{k})$; če namesto tega uporabimo izboljšavo, da gledamo le z-je do $\sqrt{k}$, smo še vedno pri $O(\sqrt{k})$, zato o tej izboljšavi ne bomo govorili posebej. Pri vsakem r torej porabimo $O(\sqrt{k} \cdot t_r)$ časa, kar bo skupaj po vseh intervalih dalo $O(\sqrt{k} \sum_r t_r)$.

V tej zadnji formuli nastopa $\sum_r t_r$, vsota dolžin vseh intervalov; ta je potencialno lahko zelo velika, toda spomnimo se, da po največ $O(\sqrt{kd})$ dneh gotovo že lahko dosežemo končno proizvodnjo k, takrat pa se glavna zanka našega programa ustavi; poleg tega pa, če bi bil posamezni t_r večji od $2\sqrt{k}$, bi se tam ustavili, ne da bi sploh poskusili računati f_r. Tako torej vidimo, da od vsote $\sum_r t_r$ na našo časovno zahtevnost v resnici vpliva le $O(\sqrt{kd}) + O(\sqrt{k})$ dni; časovna zahtevnost izračuna vseh potrebnih f_r je torej $O(\sqrt{k} \cdot \sqrt{kd}) = O(k\sqrt{d})$.

[7]Spomnimo se, da z omejitvami ne moremo prisiliti optimalnega razporeda, da bi na tem intervalu izvedel manj kot toliko nadgradenj; lahko pa bi ga prisilili, da jih izvede več, vendar smo se tej možnosti odpovedali. Pri sestavljanju takega nabora omejitev, ki zahteva čim daljši razpored, nam tako ali tako ni v interesu, da bi razpored prehitro nadgrajeval zmogljivost, kajti prej ko doseže visoko zmogljivost, manj proizvodnih dni se bo dalo dodati v razpored; ključno pri doseganju dolgega razporeda pa je ravno to, da imamo veliko dni proizvodnje (pri nizki zmogljivosti).

Po tistem, ko izračuna vse f_r, mora naša rešitev pri vsakem z še razmisliti, koliko nadgradenj bi bilo najbolje izvesti po koncu zadnje omejitve. V prvotni različici gremo tu do $z = z_d^{\max}$ in pri vsakem z imamo zanko po u. Ta zanka po največ $\sqrt{k}$ iteracijah opazi rešitev, ki $\sqrt{k}$ dni nadgrajuje in $\sqrt{k}$ dni proizvaja; po največ $2\sqrt{k}$ iteracijah pa opazi, da zdaj še samo za nadgradnje porabi več časa kot pri najboljši rešitvi za nadgradnje in proizvodnjo skupaj, zato se ustavi. Ta zanka torej porabi $O(\sqrt{k})$ časa pri vsakem z; po vseh z skupaj (do $z_d^{\max}$, kar je $\leq \hat{z}_d^{\max} = O(\sqrt{k})$) je to $O(\sqrt{k} \cdot \sqrt{k}) = O(k)$. Ta del postopka je torej poceni v primerjavi z glavnim delom, ki je računal funkcije $f_1, \ldots, f_d$.

Razmislimo zdaj o glavnih dveh izboljšavah, ki smo si ju ogledali. Ena je, da pri izračunu $f_r(z) = \max_u F_r(z, u)$ ne uporabimo zanke po u, pač pa upoštevamo, da je maksimum vedno dosežen pri najmanjšem u, to je $u = \max\{0, z - \hat{z}_{r-1}^{\max}\}$. Tako imamo pri vsakem z le $O(1)$ dela, skupaj $O(\hat{z}_r^{\max}) = O(\sqrt{k})$ pri vsakem r, kar nanese $O(d\sqrt{k})$ pri vseh r skupaj.

Druga izboljšava je, da na koncu pri vsakem z ne iščemo števila nadgradenj po koncu zadnjega intervala z zanko po u, pač pa ga izračunamo po formuli $\max\{0, \lfloor k - f_d(z) \rfloor - z\}$. To pomeni, da imamo pri vsakem z le $O(1)$ dela, pri vseh skupaj pa zato $O(\sqrt{k})$.

Zaključimo torej lahko, da ima osnovna različica naše rešitve s strani 83 časovno zahtevnost $O(k\sqrt{d})$; z obema tu omenjenima izboljšavama pa se to zmanjša na $O(d\sqrt{k})$. V obeh primerih se spodobi prišteti še $O(d \log d)$ za urejanje omejitev na začetku postopka.

4. Virus v Timaniji

V opisu naloge vidimo, da bomo imeli opravka z dvema precej različnima skupinama testnih primerov: pri nekaterih bo ljudi veliko, vendar bo vsak okužil samo enega drugega, nato pa takoj umrl ($\ell = 1$, $n \leq 10^5$); pri drugih pa bo ljudi malo, vendar lahko vsak okuži po več drugih ($\ell \leq 15$, $n \leq 1000$). Ti dve različni vrsti problemov bomo tudi reševali na dva različna načina. Naj bo $S(u, d)$ človek, s katerim se sreča oseba u na dan d — to je tisto, kar dobimo kot vhodne podatke.

Pri $\ell = 1$ okuži vsakdo samo enega drugega človeka; vendar pa je to, koga točno okuži, odvisno od tega, kdaj se je sam okužil: če se u okuži na dan d, bo nato sam na dan $d' := (d + k) \bmod 7$ okužil človeka $S(u, d')$. Te povezave si lahko predstavljamo kot graf, v katerem so točke vsi možni pari oblike (u, d), torej človek ter dan v tednu, ko se je okužil; povezave v tem grafu pa naj bodo $(u, d) \to (S(u, d'), d')$, kot smo videli zgoraj. Iz vsake točke torej kaže ena sama povezava; hitro pa vidimo, da tudi v vsako točko kaže le ena povezava: človek u' se lahko na dan d' okuži le od tistega, s komer se ta dan sreča, torej od $S(u', d')$; in ker je ta človek kužen le en dan (nato pa umre), se je moral sam okužiti na dan $(d' - k) \bmod 7$. V točko (u', d') torej kaže povezava le iz točke $(S(u', d'), (d' - k) \bmod 7)$.

Začnimo v poljubni točki grafa in sledimo izhodnim povezavam: $q_0, q_1, q_2, \ldots$, toda ker ima graf le končno mnogo točk, moramo prej ali slej priti v neko točko, ki smo jo že obiskali; recimo, da se to prvič zgodi tako, da je $q_t = q_i$ za $i < t$. Če bi bila $q_i > 0$, bi to pomenilo, da smo v q_i prišli tako iz q_t kot iz q_{i-1}, kar je nemogoče, ker v vsako točko kaže samo ena povezava. Ostane le možnost $q_i = 0$, torej smo prišli nazaj v točko, kjer smo začeli; točke, ki smo jih prehodili, tvorijo cikel. Ves graf je

sestavljen iz takšnih ciklov, ki so drug od drugega seveda ločeni (saj točke nimajo drugih izhodnih povezav kot naprej po svojem ciklu, tako da ni povezav med cikli).

Epidemija se torej ne bo mogla širiti iz enega cikla v drugega; cikle lahko obdelujemo vsakega posebej v zanki, dokler ne obiščemo vseh točk grafa. Recimo torej zdaj, da imamo pred seboj cikel $q_0, q_1, \ldots, q_{t-1}, q_t = q_0$. Če se epidemija začne v točki q_i na tem ciklu, se od tam razširi v q_{i+1}, pa v q_{i+2} in tako naprej. Edino, zaradi česar se to širjenje ustavi, je, če doseže nekega človeka, ki se je okužil že prej: če imamo recimo $q_i, q_{i+1}, \ldots, q_j, q_{j+1}, \ldots, q_{k-1}, q_k$ in se točki q_j in q_k nanašata na istega človeka (vendar na dva različna dneva v tednu), to pomeni, da se je ta človek nalezel bolezni v stanju q_j, nato jo je v stanju q_{j+1} prenesel na nekoga drugega in takoj zatem umrl (ker je $\ell = 1$); ko torej pride čas za prenos $q_{k-1} \to q_k$, do njega ne more priti, ker je tisti, ki bi se moral v stanju q_k okužiti, že mrtev.

Na ciklu moramo torej poiskati čim daljši strnjen podniz točk, ki se nanašajo na same različne ljudi. To lahko naredimo tako, da gremo v zanki naraščajoče po i, nato pa pri vsakem i pogledamo, kako daleč pride okužba, če se začne pri q_i. Tega ni treba računati pri vsakem i od začetka; če se okužba začne pri q_{i+1} namesto pri q_i, se bo razširila vsaj tako daleč, kot se je prej, ko se je še začela pri q_i; preveriti moramo le, če se zdaj mogoče razširi kaj dlje. Zato je koristno v neki tabeli hraniti podatke o tem, kateri ljudje so prisotni v trenutno opazovanem podnizu; na desnem koncu lahko podniz podaljšujemo tako dolgo, dokler se naslednja točka nanaša na človeka, ki ga še ni v podnizu.

Oglejmo si zdaj še drugi del naloge, pri katerem je ℓ lahko večji od 1, vendar pa je ljudi malo. Takrat si lahko privoščimo preizkusiti vseh $7n$ možnih scenarijev (torej vseh možnih kombinacij tega, kdo se je prvi okužil in na kateri dan v tednu) in za vsakega od njih preprosto odsimulirati dogajanje, dokler se epidemija ne neha širiti.

Vzdrževali bomo tabelo s podatki o tem, kdo je še zdrav (tisti, ki niso, so ali že umrli ali pa še bodo); poleg tega potrebujemo še nekakšno čakalno vrsto okužb, do katerih bo prišlo v prihodnosti. Ko na primer med simulacijo pridemo do dneva d in vidimo, da se tisti dan okuži človek u, s tem tudi vemo, da bo ta človek v dnevih od $d + k$ do $d + k + \ell - 1$ okužil tiste, s katerimi bo takrat v stiku, tako da moramo te okužbe dodati v čakalno vrsto pod ustreznimi dnevi.

V spodnji rešitvi je ta čakalna vrsta implementirana tako, da imamo po en seznam za vsakega od prihodnjih $k + \ell$ dni. Ker nam ne bo treba hraniti več kot $k + \ell$ takih seznamov naenkrat, jih bomo hranili kar v krožni tabeli (*ring buffer*), kjer seznam za dan d hranimo na indeksu $d \bmod (k + \ell)$. Vsak dan se sprehodimo po seznamu za tisti dan in ustrezno popravimo stanje ljudi na njem.

Simulacija se konča, ko so vsi seznami za prihodnjih $k + \ell$ dni prazni — ali, z drugimi besedami, takrat, ko ni več okuženih ali kužnih ljudi, ampak samo še zdravi in mrtvi.

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

int n, k, L; vector<array<int, 7>> T;                  // vhodni podatki
int uNaj = −1, dNaj = −1, pNaj = −1, nNaj = 0;  // rezultati
```

```
void Cikli()
{
  vector<bool> obiskan(n * 7, false), vNizu(n, false);
  vector<int> cikel;
  for (int u0 = 0; u0 < n; ++u0) for (int d0 = 0; d0 < 7; ++d0) if (! obiskan[u0 * 7 + d0])
  {
    // Začnimo okužbo z človekom u0 na dan d0.
    int u = u0, d = d0; cikel.clear();
    do {
      // u se je okužil na dan d; kdaj in koga bo okužil on?
      d = (d + k) % 7; u = T[u][d];
      cikel.push_back(u * 7 + d); obiskan[u * 7 + d] = true;
    } while (u != u0 || d != d0);

    // Poiščimo v ciklu najdaljši podniz, ki ne vsebuje po večkrat istega človeka.
    int m = cikel.size(), r = 0;
    for (int i = 0; i < m; ++i, --r)
    {
      // Če se podniz začne pri i in je dolg r členov, še ne vsebuje nobenega
      // človeka po večkrat. vNiz[u] nam pove, ali ta podniz vsebuje človeka u.
      // Podaljšajmo r, kolikor je le mogoče.
      while (! vNizu[cikel[(i + r) % m] / 7])
        vNizu[cikel[(i + r++) % m] / 7] = true;

      // Zapomnimo si najboljšo rešitev.
      int preziveli = n − r;
      if (pNaj < 0 || preziveli < pNaj)
        pNaj = preziveli, uNaj = cikel[i] / 7, dNaj = cikel[i] % 7, nNaj = 1;
      else if (preziveli == pNaj) ++nNaj;

      // Ko bomo premaknili i, torej začetek podniza, za en znak naprej,
      // bo s tem en človek izpadel iz podniza.
      vNizu[cikel[i % m] / 7] = false;
    }
    while (r > 0) vNizu[cikel[−−r] / 7] = false; // pospravimo za sabo
  }
}

void Simulacija()
{
  vector<bool> zdrav(n, true); vector<int> mrtvi, kuzni;
  // vrste[d % M] je seznam ljudi, ki se bodo na dan d okužili.
  const int M = k + L; vector<vector<int>> vrste(M);

  for (int u0 = 0; u0 < n; ++u0) for (int d0 = 0; d0 < 7; ++d0)
  {
    // Začnimo okužbo z človekom u0 na dan d0.
    mrtvi.clear(); vrste[d0 % M].push_back(u0);

    // Simulirajmo, dokler se ima še kaj spremeniti.
    for (int dan = d0, toDo = 1; toDo > 0; ++dan)
    {
      // Kdo vse se danes okuži?
      auto &vrsta = vrste[dan % M];
      for (int u : vrsta) if (zdrav[u]) {
        zdrav[u] = false; mrtvi.push_back(u);

        // Koga vse bo u okužil in kdaj?
        for (int d = dan + k; d < dan + k + L; ++d)
          vrste[d % M].push_back(T[u][d % 7]), ++toDo; }
```

```
      toDo −= vrsta.size(); vrsta.clear();
    }
    // Zapomnimo si najboljšo rešitev.
    int preziveli = n − mrtvi.size();
    if (pNaj < 0 || preziveli < pNaj) pNaj = preziveli, uNaj = u0, dNaj = d0, nNaj = 1;
    else if (preziveli == pNaj) ++nNaj;
    for (int u : mrtvi) zdrav[u] = true; // pospravimo za sabo
  }
}

int main()
{
  // Preberimo vhodne podatke.
  scanf("%d %d %d", &n, &k, &L); T.resize(n);
  for (int u = 0; u < n; ++u) for (int d = 0; d < 7; ++d) scanf("%d", &T[u][d]);
  // Rešimo nalogo s primerno izbranim algoritmom.
  if (L == 1) Cikli(); else Simulacija();
  // Izpišimo rezultate.
  printf("%d %d %d %d\n", uNaj, dNaj, pNaj, nNaj); return 0;
}
```

5. Tja in spet nazaj

Hobitova pot je sestavljena iz dveh delov: tja (od zahoda proti vzhodu) in spet nazaj (od vzhoda proti zahodu). Če se uspemo nekako odločiti, katere točke bi obiskali v prvem delu, je s tem določena že cela pot: prvi del mora obiskati te točke v naraščajočem vrstnem redu njihovih x-koordinat, drugi del pa mora obiskati vse ostale točke v padajočem vrstnem redu x-koordinat.

Ker je torej vrstni red točk po x-koordinatah pomemben pri reševanju te naloge, je koristno, če si vhodne podatke za začetek uredimo po x-koordinati; v nadaljevanju bomo torej predpostavili, da velja $x_1 < x_2 < \ldots < x_n$.

Recimo, da bi želeli prvi del poti sestavljati postopoma po korakih in vanj dodajati točke eno po eno od zahoda proti vzhodu; in recimo, da je točka i zadnja, ki smo jo doslej dodali v prvi del poti. To pomeni, da kasneje nobene od prvih točk $\{1, 2, \ldots, i\}$ ne bomo več dodali v prvi del poti; tiste, ki jih doslej še nismo dodali v prvi del, bomo morali torej obiskati v drugem delu. Zato bi se dalo že zdaj izračunati, kako dolgo pot bo drugi del opravil, da bo obiskal tiste točke; tako bomo pravzaprav sestavljali oba dela poti naenkrat, pri čemer bomo drugi del gradili v nasprotni smeri od tiste, v kateri ga bo hobit kasneje zares prehodil.

Vidimo lahko tudi, da pri tem razmisleku ni zares pomembno, kateri del poti je prvi in kateri drugi, kajti ko imamo enkrat pripravljen cel obhod (tja in spet nazaj), ga lahko hobit prehodi tudi v nasprotni smeri, pa bo rezultat enako dober. Rečemo lahko torej, da preprosto sestavljamo dve poti od zahoda proti vzhodu hkrati, pri čemer bo šla vsaka nova točka v natanko eno od teh dveh poti.

To je torej tisto, glede česar se bomo morali pri vsaki točki odločiti: ali bi z njo podaljšali eno ali drugo od naših dveh nastajajočih poti. Da pa bomo lahko izračunali, za koliko se pri tem poveča dolžina tiste poti, moramo vedeti, katera točka je bila doslej zadnja na tisti poti.

Naj bo torej $f(i, j)$ skupna dolžina obeh poti za najboljšo tako rešitev, pri kateri smo na obe poti že razporedili prvih i točk (ostalih pa še ne), pri čemer je zadnja

točka na eni poti bila točka j (za $j < i$), zadnja točka na drugi poti pa je bila točka i. To slednje pomeni, da tista pot, ki vsebuje točko i, gotovo pred tem vsebuje tudi točke $j+1, j+2, \ldots, i-1$; brez točke i imamo torej pred seboj rešitev problema za prvih $i-1$ točk, pri čemer ima druga pot še vedno j kot zadnjo točko; najboljša rešitev tega pa je $f(i-1, j)$. Tako torej vidimo, da pri $j < i-1$ velja $f(i, j) = f(i-1, j) + d(i-1, i)$, pri čemer $d(\cdot, \cdot)$ pomeni razdaljo med točkama v oklepajih.

Malo drugače pa moramo obravnavati primer, ko je $j = i-1$. Pot, ki bo zdaj vsebovala točko i, torej ne vsebuje točke $i-1$ (kajti z njo se konča druga pot); zadnja točka pred i na njej je torej neka točka $k < i-1$. Brez točke i imamo torej pred sabo poti, ki pokrijeta prvih $i-1$ točk, pri čemer se ena konča pri k (druga pa seveda pri $i-1$); najboljša rešitev tega podproblema pa je $f(i-1, k)$. Ko potem našo pot do k podaljšamo s korakom od k do i, naraste skupna dolžina na $f(i-1, k) + d(k, i)$. Ker ne moremo vnaprej vedeti, pri katerem k bo ta skupna dolžina najmanjša, moramo preizkusiti vse. Tako smo dobili:

$$f(i, i-1) = \min\{f(i-1, k) + d(k, i) : 1 \le k < i-1\}.$$

Poseben primer je še $i = 2$, ko imamo le dve točki in sploh nimamo nobene izbire: $f(1, 0) = d(0, 1)$.

Funkcijo f je koristno računati po naraščajočih i in shranjevati njene vrednosti v tabelo, da jih bomo imeli pri roki, ko jih bomo kasneje spet potrebovali. Ko računamo vrednosti funkcije za neki i, potrebujemo rezultate za $i-1$, ne pa več tistih za $i-2$, $i-3$ itd., zato lahko tiste sproti pozabljamo. Tako je prostorska zahtevnost naše rešitve le $O(n)$, časovna pa $O(n^2)$.

Na koncu nas zanima dolžina celotnega obhoda. Recimo, da smo vse točke že razdelili med obe poti in da se je ena torej končala s točko n, druga pa s točko j za neki $j < n$. Najkrajša možna dolžina takih dveh poti je $f(n, j)$; da pa dobimo dolžino obhoda, moramo pot do j še podaljšati s korakom do n. Med tako dobljenimi obhodi moramo vrniti najkrajšega, to je $\max\{f(n, j) + d(j, n) : 1 \le j < n\}$.

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

int main()
{
  // Preberimo vhodne podatke.
  int n; scanf("%d", &n);
  struct Tocka { int x, y; };
  vector<Tocka> T(n);
  for (auto &t : T) scanf("%d %d", &t.x, &t.y);

  // Uredimo točke naraščajoče po x.
  sort(T.begin(), T.end(), [] (const auto &t, const auto &u) { return t.x < u.x; });

  // Funkcija, ki vrne razdaljo med dvema točkama.
  auto D = [&T] (int i, int j) { double dx = T[i].x - T[j].x, dy = T[i].y - T[j].y;
                                 return sqrt(dx * dx + dy * dy); };
```

```
  // Rešimo nalogo.
  vector<double> f(n − 1), ff(n − 1);
  f[0] = D(0, 1); // Rešitev za prvi dve točki.
  for (int i = 2; i < n; i++)
  {
    // f[j] = najboljša rešitev za točke 0, . . . , i − 1, pri čemer se ena pot konča v točki j,
    // druga pa v točki i − 1. Izračunajmo zdaj rešitve za točke 0, . . . , i in jih shranimo v ff.
    ff[i − 1] = numeric_limits<double>::infinity();
    for (int j = 0; j < i − 1; ++j) {
      // V rešitvi, pri kateri se ena pot konča v i − 1, druga pa v j,
      // lahko podaljšamo prvo pot s korakom iz i − 1 v i.
      ff[j] = f[j] + D(i − 1, i);
      // Lahko pa podaljšamo drugo pot s korakom iz j v i.
      ff[i − 1] = min(ff[i − 1], f[j] + D(j, i)); }
    swap(f, ff);
  }
  // Poiščimo dolžino najkrajšega obhoda.
  double r = numeric_limits<double>::infinity();
  for (int j = 0; j < n − 1; ++j)
    // Rešitev, pri kateri se konča ena pot v n − 1 in druga v j,
    // lahko podaljšamo s korakom iz j v n − 1 in dobimo obhod.
    r = min(r, f[j] + D(j, n − 1));
  // Izpišimo rezultat.
  printf("%.6f\n", r); return 0;
}
```

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Križci in krožci

Nalogo lahko rešimo z zanko, ki pregleduje znake niza enega po enega. Pri vsakem znaku preverimo, če je križec ali krožec; če ni nič od tega dvojega, lahko takoj zaključimo, da je niz neveljaven. Podobno tudi preverimo, če je trenutni znak slučajno enak prejšnjima dvema; če je, imamo tri zaporedne enake znake in vemo, da je niz neveljaven. Na koncu moramo preveriti še, če je število križcev enako številu krožcev; spodnja rešitev to počne tako, da med pregledovanjem niza sproti računa razliko med številom doslej prebranih križcev in številom doslej prebranih krožcev. Če je ta razlika na koncu enaka 0, je bilo križcev in krožcev enako veliko, sicer pa ne.

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

bool Izenaceno(const string &s)
{
   int razlika = 0; // Razlika v dosedanjem številu križcev in krožcev.
   char c1 = ' ', c2 = ' '; // Prejšnja dva znaka.
   // Pojdimo po znakih niza.
   for (char c : s)
   {
      // Ali so vsi znaki križci in krožci?
      if (c != 'x' && c != 'o') return false;
      // Ali so kdaj trije zaporedni znaki enaki?
      if (c == c1 && c1 == c2) return false;
      // Popravimo razliko med številom križcev in krožcev.
      razlika += (c == 'x') ? 1 : −1;
      // Prejšnja dva znaka si zapomnimo.
      c2 = c1; c1 = c;
   }
   // Na koncu še preverimo, če je križcev in krožcev enako mnogo.
   return razlika == 0;
}
```

Zapišimo to rešitev še v pythonu:

```
def Izenaceno(s):
   razlika = 0 # Razlika v dosedanjem številu križcev in krožcev.
   c1 = ' '; c2 = ' ' # Prejšnja dva znaka.
   # Pojdimo po znakih niza.
   for c in s:
      # Ali so vsi znaki križci in krožci?
      if c != 'x' and c != 'o': return False
      # Ali so kdaj trije zaporedni znaki enaki?
      if c == c1 and c1 == c2: return False
      # Popravimo razliko med številom križcev in krožcev.
      razlika += 1 if c == 'x' else −1
      # Prejšnja dva znaka si zapomnimo.
```

```
    c2 = c1; c1 = c
  # Na koncu še preverimo, če je križcev in krožcev enako mnogo.
  return razlika == 0
```

Tej rešitvi se pozna, da smo jo najprej napisali v C++ in nato prevedli v python. Zapišimo jo še bolj pythonično:

```
def Izenaceno2(s):
  krizci = s.count('x'); krozci = s.count('o')
  return krizci == krozci and krizci + krozci == len(s) and "xxx" not in s and "ooo" not in s
```

2. Kovanci

Recimo, da je na mizi n kupčkov, pri čemer na i-tem kupčku leži a_i kovancev (za $i = 1, 2, \ldots, n$). Razmišljamo lahko od konca proti začetku. Pri zadnjem kupčku imamo dve možnosti: ali ga vzamemo ali pa ne. Če zadnji kupček vzamemo, potem predzadnjega ne smemo (saj naloga pravi, da ne smemo vzeti dveh sosednjih kupčkov), kar pomeni, da nam ostane le še vprašanje, kako sestaviti čim večjo vsoto iz prvih $n - 2$ kupčkov. Po drugi strani pa, če zadnjega kupčka ne vzamemo, potem predzadnjega smemo vzeti, kar pomeni, da imamo zdaj pred seboj vprašanje, kako sestaviti čim večjo vsoto iz prvih $n - 1$ kupčkov.

V obeh primerih smo torej prišli do problema, ki je enake oblike kot prvotni, le da namesto vseh kupčkov gledamo samo prvih nekaj (natančneje prvih $n - 2$ ali $n - 1$). Ker vnaprej ne moremo vedeti, katera od obeh možnosti bo dala boljšo rešitev, moramo preizkusiti obe in uporabiti boljšo od njiju.

Tega ni težko zapisati z rekurzivno zvezo. Naj bo $f(k)$ največja vsota, ki jo lahko sestavimo z izbiranjem izmed prvih k kupčkov. Končni rezultat, po katerem sprašuje naloga, je potem $f(n)$. Dosedanji razmislek pa lahko strnemo v formulo:

$$f(k) = \max\{a_k + f(k-2), f(k-1)\}.$$

Robni primer nastopi na začetku zaporedja, kjer si lahko mislimo $f(k) = 0$ za $k \leq 0$ (če sploh ni nobenega kupčka kovancev, bo tudi naš pobrani znesek lahko zgolj 0).

Iz te formule lahko vidimo, da je pri računanju $f(k)$ koristno, če tedaj že poznamo $f(k-1)$ in $f(k-2)$. Torej je pametno računati te vrednosti od leve proti desni, po naraščajočih k. Že izračunane vrednosti funkcije f bi lahko shranjevali v tabelo, vendar pravzaprav v vsakem trenutku potrebujemo le prejšnji dve vrednosti: ko računamo $f(k)$, potrebujemo $f(k-1)$ in $f(k-2)$, ne pa več tudi $f(k-3)$ in tako naprej. Zato je dovolj, če vedno hranimo le zadnji dve vrednosti funkcije.

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

int NajvecjaVsota(const vector<int> &kupi)
{
  int f = 0, fPrej = 0; // Rešitvi za 0 kupov.
  // Pregledujmo kupe od leve proti desni.
  for (int kup : kupi)
  {
    // V spremenljivki f je zdaj najboljša rešitev za vse kupe do
```

```
      // vključno prejšnjega, v fPrej pa za vse kupe pred prejšnjim.
      // Izračunajmo najboljšo rešitev za vse kupe vključno s trenutnim.
      int fNova = max(fPrej + kup, f);

      // Zadnji dve rešitvi si zapomnimo.
      fPrej = f; f = fNova;
   }
   return f; // Vrnimo rešitev za vse kupe.
}
```

Zapišimo to rešitev še v pythonu:

```
def NajvecjaVsota(kupi):
   f = 0; fPrej = 0 # Rešitvi za 0 kupov.

   # Pregledujmo kupe od leve proti desni.
   for kup in kupi:

      # V spremenljivki f je zdaj najboljša rešitev za vse kupe do
      # vključno prejšnjega, v fPrej pa za vse kupe pred prejšnjim.
      # Izračunajmo najboljšo rešitev za vse kupe vključno s trenutnim.
      fNova = max(fPrej + kup, f)

      # Zadnji dve rešitvi si zapomnimo.
      fPrej = f; f = fNova

   # Vrnimo rešitev za vse kupe.
   return f
```

Razmislimo še o težji različici naloge, ki jo omenja opomba pod črto na koncu besedila: vprašanje je torej, kateri kupček naj pobrišemo, da bo potem Janezkov izkupiček čim manjši. Če pobrišemo kupček p, si lahko preostale kupčke predstavljamo kot razdeljene na dve skupini, levo (od 1 do $p-1$) in desno (od $p+1$ do n — mislimo si torej, da imajo kupčki enake zaporedne številke kot prej, torej še vedno do n, čeprav smo enega pobrisali).

Če zdaj Janezek na primer izbere kup $p+1$, ima to dve posledici: kupa $p-1$ ne sme izbrati in njegov največji možni izkupiček iz leve skupine je $f(p-2)$; podobno pa tudi kupa $p+2$ ne sme izbrati in njegov največji možni izkupiček iz desne skupine je tak, kot če bi se omejil le na kupčke od $p+3$ naprej.

Če pa kupa $p+1$ ne izbere, potem sme izbrati kup $p-1$ in največji možni izkupiček iz leve skupine je $f(p-1)$; ravno tako pa sme tudi izbrati kup $p+2$ in največji možni izkupiček iz desne skupine je tak, kot če bi se omejil na kupčke od $p+2$ naprej.

Iz tega razmisleka vidimo, da je koristno definirati še eno funkcijo, podobno funkciji f: naj bo $g(k)$ največji izkupiček, ki ga lahko dobimo, če se omejimo na kupčke od k do n. Prejšnja dva odstavka lahko zdaj povzamemo takole: Janezkov najboljši izkupiček, če smo mu prej pobrisali kupček p, je enak

$$h(p) := \max\{f(p-2) + a_{p+1} + g(p+3), f(p-1) + g(p+2)\}.$$

Pri izračunu te funkcije moramo biti nekoliko pazljivi pri robnih primerih; koristno je vzeti $f(k) = 0$ pri $k < 1$ in $g(k) = 0$ pri $k > n$; pri $p = n$ si mislimo še $a_{n+1} = 0$.

Funkcijo g pa lahko računamo po podobnem razmisleku kot f: če izberemo kup k, potem kupa $k+1$ ne smemo in lahko nadaljujemo šele pri $k+2$, sicer pa ga lahko

izberemo in torej nadaljujemo pri $k+1$; tako dobimo

$$g(k) = \max\{a_k + g(k+2), g(k+1)\}.$$

Računamo jo lahko po padajočih k, podobno kot smo f računali po naraščajočih k.

Da rešimo nalogo, moramo zdaj le izračunati $h(p)$ za vse p in pogledati, kdaj doseže svoj minimum. Oglejmo si implementacijo takšne rešitve v C++:

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

pair<int, int> NajvecjaVsota2(const vector<int> &kupi)
{
  int n = kupi.size();

  // f[k] = najboljši izkupiček za prvih k kupov
  vector<int> f_(n + 1); auto f = f_.begin() + 1; f[-1] = 0; f[0] = 0;
  for (int k = 1; k < n; ++k) f[k] = max(f[k - 1], f[k - 2] + kupi[k - 1]);

  // g[k] = najboljši izkupiček za vse kupe razen prvih k
  vector<int> g(n + 3); g[n + 2] = 0; g[n + 1] = 0; g[n] = 0;
  for (int k = n - 1; k >= 0; --k) g[k] = max(g[k + 1], g[k + 2] + kupi[k]);

  // Izračunajmo h(p) za vse p in si zapomnimo njen minimum.
  int pNaj = -1, hNaj = 0;
  for (int p = 0; p < n; ++p) {
    int h = max(f[p - 1] + (p < n - 1 ? kupi[p + 1] : 0) + g[p + 3], f[p] + g[p + 2]);
    if (pNaj < 0 || h < hNaj) hNaj = h, pNaj = p; }

  return { pNaj, hNaj };
}
```

Funkcija vrne par celih števil, od katerih prvo pove, kateri kup p naj Štefan pobriše, drugo pa, kakšen bo potem Janezkov izkupiček. Da imamo pri računanju $h(p)$ manj dela z robnimi primeri, je koristno, če obstajajo pri funkcijah f in g tudi vrednosti $f(-1)$, $g(n)$, $g(n+1)$ in $g(n+2)$, zato imata v gornji rešitvi vektorja f_ in g malo več kot n elementov; in ker števila -1 ne moremo uporabiti kot indeksa v vektor, smo za f vzeli iterator, ki kaže na *drugi* element vektorja f_.

Zapišimo to rešitev še v pythonu:

```
def NajvecjaVsota2(kupi):
  n = len(kupi); f = [0] * (n + 2); g = [0] * (n + 3)

  # f[k] = najboljši izkupiček za prvih k kupov
  for k in range(1, n + 1): f[k] = max(f[k - 1], f[k - 2] + kupi[k - 1])

  # g[k] = najboljši izkupiček za vse kupe razen prvih k
  for k in range(n - 1, -1, -1): g[k] = max(g[k + 1], g[k + 2] + kupi[k])

  # Izračunajmo h(p) za vse p in si zapomnimo njen minimum.
  pNaj = -1; hNaj = 0
  for p in range(n):
    h = max(f[p - 1] + (kupi[p + 1] if p < n - 1 else 0) + g[p + 3], f[p] + g[p + 2])
    if pNaj < 0 or h < hNaj: hNaj = h; pNaj = p

  return pNaj, hNaj
```

Tu z negativnimi indeksi ni težav, saj so v pythonu veljavni (in se nanašajo na elemente na koncu seznama).

Pri prvotni različici naloge smo lahko vrednosti funkcije f sproti pozabljali, tako da je porabila naša rešitev le $O(1)$ dodatnega pomnilnika (poleg tistega, ki ga že tako ali tako zaseda vhodna tabela a oz. kupi). Pri težji različici pa to ni tako preprosto; da izračunamo $h(p)$ za neki konkreten p, potrebujemo vredosti $f(k)$ in $g(k)$ za nekaj k-jev v bližini tega p. Če računamo h po naraščajočih p-jih, bomo torej tudi f in g potrebovali po naraščajočih k-jih; funkcije $f(k)$ ni težko računati po naraščajočih k-jih (in sproti pozabljati tiste $f(k)$, ki jih ne bomo več potrebovali), funkcije $g(k)$ pa ne, saj je slednja definirana tako, da jo lahko računamo le po *padajočih* k-jih. Če pa bi želeli računati h po padajočih p-jih namesto po naraščajočih, bi naleteli na podoben problem pri funkciji f namesto pri g.

Naša dosedanja rešitev, funkcija NajvecjaVsota2, je porabila $O(n)$ dodatnega pomnilnika za tabeli oz. vektorja, v katerih je hranila vrednosti funkcij f in g. V mislih si lahko predstavljamo, da tako tabelo razdelimo na bloke dolžine B; teh blokov je torej približno n/B. Recimo, da bomo računali $h(p)$ po naraščajočih p; videli smo, da bomo pri tem potrebovali tudi $f(k)$ po naraščajočih k, torej lahko te vrednosti računamo sproti in jih tudi sproti pozabljamo, tabele za funkcijo f pa sploh ne potrebujemo. Glede funkcije g pa naredimo takole: najprej izračunajmo vse $g(k)$ po padajočih g, vendar jih večino tudi sproti pozabimo; zapomnimo si le zadnji dve iz vsakega bloka. Ko začnemo nato računati vrednosti $h(p)$ po naraščajočih p in zato potrebujemo tudi vrednosti $g(k)$ po naraščajočih k, pa lahko vsakič, ko pridemo do novega bloka, na novo izračunamo vse vrednosti $g(k)$ v tem bloku iz zadnjih dveh (pri tem gremo seveda padajoče po k-jih tega bloka). Ko se kasneje p že toliko poveča, da nekega bloka ne bomo več potrebovali, ga lahko v celoti pozabimo. Poraba pomnilnika je tako le $O(B + n/B)$, ker v vsakem trenutku hranimo le en ali dva zaporedna bloka, poleg tega pa še po zadnja dva elementa vsakega bloka. Najbolje je torej vzeti $B \approx \sqrt{n}$, ko imamo prostorsko zahtevnost $O(\sqrt{n})$, časovna pa je še vedno le $O(n)$, četudi je konstantni faktor zdaj malo večji (ker smo morali funkcijo g pravzaprav računati dvakrat).

Še bolj pa lahko porabo pomnilnika zmanjšamo, če smo pripravljeni v zameno sprejeti malo večjo časovno zahtevnost. Opazimo lahko, da za izračun $h(p)$ potrebujemo le vrednosti $f(k)$ in $g(k)$ za nekaj k-jev blizu tistega p; naj bo $F(p) = \langle f(p-2), f(p-1) \rangle$ in $G(p) = \langle g(p+2), g(p+3) \rangle$, pa lahko $h(p)$ izračunamo iz $F(p)$ in $G(p)$ (in a_{p+1}, toda tabela a z vhodnimi podatki je tako ali tako vedno na voljo). Obenem tudi vidimo, da lahko $F(p)$ izračunamo iz $F(p-1)$ (in a_p), $G(p)$ pa iz $G(p+1)$ (in a_p). Če nas zanimajo vrednosti $h(p)$ za več zaporednih p-jev — recimo za območje $\ell \leq p \leq d$ — je koristno, če imamo za začetek pri roki $F(\ell)$ in $G(d)$. Zdaj lahko razmišljamo rekurzivno: območje razdelimo na dve polovici; izračunajmo F in G na koncu prve in na začetku druge polovice; in vsako polovico obdelajmo z rekurzivnim klicem. Robni primer je, ko je $d = \ell$, takrat pa imamo torej $F(\ell)$ in $G(\ell)$ in lahko iz njiju izračunamo $h(\ell)$. Tako sčasoma izračunamo h za vse možne vrednosti njegovega argumenta, sproti pa lahko tudi gledamo, katera od njih ima najmanjšo vrednost (to hranimo npr. v neki globalni spremenljivki). Zapišimo to rešitev s psevdokodo:

vhodni podatki: vrednosti kupov $a_1, \ldots, a_n$;

globalni spremenljivki: $p^\star$ in $h^\star$;

podprogram Rekurzija(ℓ, d, $F(\ell)$, $G(d)$):
  if $\ell = d$ **then**
    izračunaj $h(\ell)$ iz $F(\ell)$ in $G(\ell)$;
    if $h(\ell) < h^\star$ **then** $h^\star := h(\ell)$, $p^\star := \ell$;
  else:
    $m := \lfloor(\ell + d)/2\rfloor$;
    for $p := \ell + 1$ **to** $m + 1$ **do** izračunaj $F(p)$ iz $F(p-1)$;
    for $p := d - 1$ **downto** m **do** izračunaj $G(p)$ iz $G(p+1)$;
    (* *vrednosti* $F(p)$ *in* $G(p)$ *sproti pozabljaj, razen pri* $p = m$ *in* $p = m + 1$ *)
    Rekurzija(ℓ, m, $F(\ell)$, $G(m)$); Rekurzija($m + 1$, d, $F(m+1)$, $G(d)$);

glavni del programa:
  $h^\star := \infty$; $p^\star := -1$; Rekurzija(1, n);

Na koncu imamo v $p^\star$ in $h^\star$ vrednosti, po katerih sprašuje naloga. Pri tem smo porabili $O(\log n)$ dodatnega pomnilnika, kolikor znaša globina rekurzije; glede porabe časa pa lahko razmišljamo takole: da obdelamo območje n možnih vrednosti p-ja, porabimo najprej $O(n)$ časa za izračun vrednosti F in G do sredine območja, nato pa izvedemo dva rekurzivna klica za polovico manjši območji; imamo torej časovno zahtevnost $T(n) = O(n) + 2\,T(n/2)$ in hitro se lahko prepričamo, da to pomeni $T(n) = O(n \log n)$. Tako smo torej nekaj izgubili pri časovni, veliko pa pridobili pri prostorski zahtevnosti.

3. Taksi

Preprosta, vendar manj učinkovita rešitev je z dvema gnezdenima zankama. Z zunanjo zanko pojdimo po vseh možnih položajih centrale; pri vsakem položaju centrale pojdimo potem z notranjo zanko po strankah, računajmo razdaljo od centrale do posamezne stranke in te razdalje seštevajmo. Ko za trenutni položaj centrale poznamo vsoto razdalj do vseh strank, lahko pogledamo, če je to najmanjša vsota doslej, in če je, si jo zapomnimo (in tudi to, pri kateri centrali smo jo dobili). Na koncu zunanje zanke bomo tako poznali najmanjšo vsoto sploh in tudi položaj centrale, pri kateri smo jo dosegli.

Zapišimo to rešitev v C++:

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

struct Tocka { int x, y; };

// Vrne indeks izbrane centrale.
int IzberiCentralo1(const vector<Tocka>& stranke, const vector<Tocka>& centrale)
{
  int najVsota = 0, najCentrala = −1;
  // Preglejmo vse možne položaje centrale.
  for (int i = 0; i < centrale.size(); ++i)
  {
    // Izračunajmo vsoto razdalj od i-te centrale do vseh strank.
    int vsota = 0; Tocka C = centrale[i];
```

```
    for (const auto &S : stranke) vsota += abs(S.x − C.x) + abs(S.y − C.y);
    // Če je to najboljši rezultat doslej, si ga zapomnimo.
    if (najCentrala < 0 || vsota < najVsota)
      najVsota = vsota, najCentrala = i;
  }
  return najCentrala;
}
```

Zapišimo to rešitev še v pythonu. Tu ne bomo definirali svoje strukture oz. razreda za točko, pač pa bomo predpostavili, da dobimo točke kot urejene pare (pythonov tip tuple):

```
def IzberiCentralo1(stranke, centrale):
  najVsota = 0; najCentrala = −1
  # Preglejmo vse možne položaje centrale.
  for i, (cx, cy) in enumerate(centrale):
    # Izračunajmo vsoto razdalj od i-te centrale do vseh strank.
    vsota = 0
    for (sx, sy) in stranke: vsota += abs(sx − cx) + abs(sy − cy)
    # Če je to najboljši rezultat doslej, si ga zapomnimo.
    if najCentrala < 0 or vsota < najVsota:
      najVsota = vsota; najCentrala = i
  return najCentrala # Vrnimo najboljšo rešitev.
```

Ali, krajše:

```
def IzberiCentralo1b(stranke, centrale):
  return min((sum(abs(sx − cx) + abs(sy − cy) for (sx, sy) in stranke), i)
    for (i, (cx, cy)) in enumerate(centrale))[1]
```

Časovna zahtevnost te rešitve je $O(n \cdot m)$, če imamo n strank in m možnih položajev centrale.

Do hitrejše rešitve lahko pridemo, če si pomagamo z naslednjim opažanjem: pri takšni meri razdalje, kot jo uporabljamo pri naši nalogi (torej manhattanska razdalja namesto bolj znane evklidske razdalje), je tisto, kar k razdalji med dvema točkama prispeva razlika njunih x-koordinat, popolnoma neodvisno od tistega, kar k isti razdalji prispeva razlika njunih y-koordinat. Ta dva prispevka lahko računamo ločeno in ju nato seštejemo. Razmislimo, kaj to pomeni za našo nalogo. Naj bo $\mathbf{s}_i = (x_i, y_i)$ položaj i-te stranke (za $i = 1, \ldots, n$); in recimo, da razmišljamo o tem, da bi centralo postavili na točko $\mathbf{c} = (\tilde{x}, \tilde{y})$. Da ocenimo to možnost, moramo izračunati vsoto razdalj od $\mathbf{c}$ do vseh strank $\mathbf{s}_i$:

$$\begin{aligned} J(\mathbf{c}) &= \textstyle\sum_{i=1}^n \big(|x_i - \tilde{x}| + |y_i - \tilde{y}|\big) \\ &= \textstyle\sum_{i=1}^n |x_i - \tilde{x}| + \sum_{i=1}^n |y_i - \tilde{y}|. \end{aligned}$$

Vsoto smo torej razdelili na dve vsoti, eno za x-koordinate in eno za y-koordinate; recimo jima $J_x(\tilde{x})$ in $J_y(\tilde{y})$. Oglejmo si pobliže prvo od njiju; za drugo bo razmislek podoben. V mislih uredimo stranke naraščajoče po x-koordinati in jih v tem vrstnem redu oštevilčimo. Poglejmo zdaj, koliko jih leži levo od $\tilde{x}$, torej da zanje velja $x_i < \tilde{x}$; recimo, da je to prvih k v tem vrstnem redu, preostalih $n - k$ pa ima $x_i \geq \tilde{x}$. Videli

smo, da vsaka stranka k vsoti prispeva člen $|x_i - \tilde{x}|$; pri desnih strankah je to kar enako $x_i - \tilde{x}$, pri levih pa je ta prispevek enak $\tilde{x} - x_i$. Našo vsoto lahko zdaj razdelimo na dve, eno po levih in eno po desnih strankah:

$$\begin{aligned} J_x(\tilde{x}) &= \textstyle\sum_{i=1}^n |x_i - \tilde{x}| \\ &= \textstyle\sum_{i=1}^k (\tilde{x} - x_i) + \sum_{i=k+1}^n (x_i - \tilde{x}). \end{aligned}$$

V vsakem seštevancu se pojavlja $\tilde{x}$, ki pa ni odvisen od i, zato ga lahko nesemo ven in ga enostavno pomnožimo s številom takih seštevancev:

$$J_x(\tilde{x}) = k\tilde{x} - \left(\textstyle\sum_{i=1}^k x_i\right) + \left(\textstyle\sum_{i=k+1}^n x_i\right) - (n-k)\tilde{x}.$$

Vsoti $\sum_{i=1}^k x_i$ recimo S_k; to ni nič drugega kot vsota x-koordinat najbolj levih k točk. Druga vsota, $\sum_{i=k+1}^n x_i$, je potem enaka $S_n - S_k$. Našo formulo lahko zdaj zapišemo kot

$$\begin{aligned} J_x(\tilde{x}) &= k\tilde{x} - S_k + (S_n - S_k) - (n-k)\tilde{x} \\ &= (2k - n)\tilde{x} + S_n - 2S_k. \end{aligned}$$

Za potrebe iskanja najboljšega položaja centrale lahko v tej zadnji formuli člen S_n tudi izpustimo, saj je pri vseh $\mathbf{c}$ enak in torej nič ne vpliva na to, kateri položaj $\mathbf{c}$ bo imel najmanjšo vsoto $J(\mathbf{c})$.

Po tej formuli lahko $J_x(\tilde{x})$ računamo zelo poceni in enostavno. Ko imamo stranke enkrat urejene po x-koordinatah, lahko z bisekcijo hitro ugotovimo, koliko jih leži levo od $\tilde{x}$; tako dobimo k. Vsote S_k za vse k od 0 do n si lahko izračunamo vnaprej in jih hranimo v tabeli. Tako bomo imeli z izračunom vsake $J_x(\tilde{x})$ le $O(\log n)$ dela (zaradi bisekcije), pred tem pa še $O(n \log n)$ dela za urejanje strank po x-koordinatah. Z enakim razmislekom lahko seveda obdelamo tudi y-koordinate. Skupaj je torej časovna zahtevnost te rešitve $O((n + m) \log n)$. Zapišimo to rešitev v C++:

```
#include <algorithm>

int IzberiCentralo2(const vector<Tocka>& stranke, const vector<Tocka>& centrale)
{
   // Pripravimo urejeni tabeli x- in y-koordinat vseh strank.
   int n = stranke.size();
   vector<int> xi(n), yi(n);
   for (int i = 0; i < n; i++) xi[i] = stranke[i].x, yi[i] = stranke[i].y;
   sort(xi.begin(), xi.end()); sort(yi.begin(), yi.end());

   // Pripravimo delne vsote obeh tabel.
   vector<int> sxi(n + 1), syi(n + 1); sxi[0] = 0; syi[0] = 0;
   for (int i = 0; i < n; i++) sxi[i + 1] = sxi[i] + xi[i], syi[i + 1] = syi[i] + yi[i];

   // Preglejmo vse možne položaje centrale.
   int najVsota = 0, najCentrala = −1;
   for (int i = 0; i < centrale.size(); ++i)
   {
      // Izračunajmo vsoto razdalj od i-te centrale do vseh strank.
      Tocka C = centrale[i];
      int kx = lower_bound(xi.begin(), xi.end(), C.x) − xi.begin();
      int ky = lower_bound(yi.begin(), yi.end(), C.y) − yi.begin();
      int Jx = (2 * kx − n) * C.x − 2 * sxi[kx];
      int Jy = (2 * ky − n) * C.y − 2 * syi[ky];
      int vsota = Jx + Jy;
```

```
    // Če je to najboljši rezultat doslej, si ga zapomnimo.
    if (najCentrala < 0 || vsota < najVsota)
      najVsota = vsota, najCentrala = i;
  }
  return najCentrala;
}
```

In v pythonu:

```
import bisect

def IzberiCentralo2(stranke, centrale):
  n = len(stranke)
  # Pripravimo urejeni tabeli x- in y-koordinat vseh strank.
  xi = [x for (x, y) in stranke]; xi.sort()
  yi = [y for (x, y) in stranke]; yi.sort()
  # Pripravimo delne vsote obeh tabel.
  sxi = [0]; syi = [0]
  for x in xi: sxi.append(sxi[−1] + x)
  for y in yi: syi.append(syi[−1] + y)
  # Preglejmo vse možne položaje centrale.
  najVsota = 0; najCentrala = −1
  for i, (cx, cy) in enumerate(centrale):
    # Izračunajmo vsoto razdalj od i-te centrale do vseh strank.
    kx = bisect.bisect_left(xi, cx)
    ky = bisect.bisect_left(yi, cy)
    Jx = (2 * kx − n) * cx − 2 * sxi[kx]
    Jy = (2 * ky − n) * cy − 2 * syi[ky]
    vsota = Jx + Jy
    # Če je to najboljši rezultat doslej, si ga zapomnimo.
    if najCentrala < 0 or vsota < najVsota:
      najVsota = vsota; najCentrala = i
  return najCentrala # Vrnimo najboljšo rešitev.
```

Še ena možnost je, da možne položaje central uredimo po x-koordinati in nato istočasno pregledujemo po naraščajočih x-koordinatah tako seznam strank kot seznam položajev centrale. S tem postopkom, torej neke vrste zlivanjem dveh urejenih seznamov, bomo lahko za vsak položaj centrale določili, koliko strank je levo od nje, od tam naprej pa lahko razmišljamo enako kot zgoraj, da izračunamo J_x tega položaja centrale. Nato podobno naredimo še za y-koordinate. Porabimo torej $O(n \log n)$ časa za urejanje strank, $O(m \log m)$ časa za urejanje central in nato $O(n + m)$ za zlivanje obeh seznamov ter izračun vseh $J_x(\tilde{x})$ in $J_y(\tilde{y})$. Časovna zahtevnost te rešitve je $O(n \log n + m \log m)$, kar je bolje od prejšnje, če je $m < n$.

```
int IzberiCentralo2b(const vector<Tocka>& stranke, const vector<Tocka>& centrale)
{
  int n = stranke.size(), m = centrale.size();
  vector<int> vsote(m, 0);
  for (int os = 0; os < 2; os++) // os 0 = x, os 1 = y
  {
    // Uredimo stranke po koordinati.
    vector<int> ti(n);
    for (int i = 0; i < n; i++) ti[i] = (os == 0) ? stranke[i].x : stranke[i].y;
```

```
    sort(ti.begin(), ti.end());

    // Uredimo centrale po koordinati.
    vector<pair<int, int>> ci(m);
    for (int i = 0; i < m; i++) ci[i] = {(os == 0) ? centrale[i].x : centrale[i].y, i};
    sort(ci.begin(), ci.end());

    // Za vsako centralo izračunajmo vsoto razdalj do strank v trenutni smeri.
    int k = 0, sk = 0;
    for (auto [tc, j] : ci[i]) // centrale[j] ima koordinato tc
    {
      // Premaknimo se mimo strank, ki imajo manjšo koordinato kot trenutna centrala.
      while (k < n && ti[k] < tc) sk += ti[k++];

      // Zdaj so ti[0..k − 1] manjši od tc (in sk je njihova vsota),
      // ti[k..n − 1] pa so večji ali enaki tc.
      // Prištejmo prispevek razdalj v trenutni smeri k oceni j-tega
      // možnega položaja centrale.
      vsote[j] += (2 * k − n) * tc − 2 * sk;
    }
  }
  // Poiščimo najboljši položaj centrale.
  int najVsota = 0, najCentrala = −1;
  for (int i = 0; i < m; ++i)
    if (najCentrala < 0 || vsote[i] < najVsota)
      najVsota = vsote[i], najCentrala = i;
  return najCentrala;
}
```

Zapišimo to rešitev še v pythonu:

```
def IzberiCentralo2b(stranke, centrale):
  n = len(stranke); m = len(centrale)
  vsote = [0] * m

  for os in range(2): # os 0 = x, os 1 = y

    # Uredimo stranke po koordinati.
    ti = [s[os] for s in stranke]; ti.sort()

    # Uredimo centrale po koordinati.
    ci = [(c[os], i) for i, c in enumerate(centrale)]; ci.sort()

    # Za vsako centralo izračunajmo vsoto razdalj do strank v trenutni smeri.
    k = 0; sk = 0
    for (tc, j) in ci: # centrale[j][os] == tc

      # Premaknimo se mimo strank, ki imajo manjšo koordinato kot trenutna centrala.
      while k < n and ti[k] < tc: sk += ti[k]; k += 1
      # Zdaj so ti[0..k − 1] manjši od tc (in sk je njihova vsota),
      # ti[k..n − 1] pa so večji ali enaki tc.
      # Prištejmo prispevek razdalj v trenutni smeri k oceni j-tega
      # možnega položaja centrale.
      vsote[j] += (2 * k − n) * tc − 2 * sk

  # Poiščimo najboljši položaj centrale in ga vrnimo.
  najVsota = 0; najCentrala = −1
  for i in range(m):
    if najCentrala < 0 or vsote[i] < najVsota:
      najVsota = vsote[i]; najCentrala = i
  return najCentrala
```

4. Preusmerjanje

Ko sledimo verigi preusmeritev, je koristno, če znamo za trenutno stran hitro ugotoviti, ali obstaja z nje preusmeritev in kam. Naloga pravi, da kot vhod dobimo seznam parov (s_i, t_i), kar za ta namen ni najbolj prikladno — morali bi se sprehoditi po celem seznamu in za vsak par preverjati, ali je njegova s_i ravno tista stran, s katero se trenutno ukvarjamo. Bolje bo, če ta seznam predelamo v tabelo: naloga pravi, da so strani oštevilčene od 1 do n, zato imamo lahko tabelo z n elementi, v kateri številko strani uporabimo kot indeks; vrednost posameznega elementa pa naj nam pove, kam nas tista stran preusmeri (če nikamor, lahko tja zapišemo na primer -1).

Zdaj znamo hitro in preprosto slediti verigi preusmeritev; paziti pa moramo še na možnost, da se veriga zacikla. Ena možnost, kako odkriti tak cikel, je, da si nekje shranjujemo podatke o tem, katere strani smo med našim sledenjem verigi že obiskali. Preden sledimo preusmeritvi na neko novo stran, moramo potem preveriti, ali smo tisto stran kdaj prej že obiskali; če da, potem vemo, da smo se znašli na ciklu. Tudi za podatke obiskanosti lahko uporabimo tabelo n logičnih vrednosti, ki za vsako stran povedo, ali smo jo že obiskali ali ne.

Druga možnost je, da se ne zmenimo za to, ali smo neko stran že obiskali ali ne, pač pa štejemo, koliko korakov smo naredili pri sledenju verigi. Če naredimo n korakov, ne da bi se veriga končala, potem vemo, da se vsaj ena stran na verigi pojavi več kot enkrat (ker bi drugače morali imeti $n + 1$ različnih strani, v resnici pa jih je samo n), torej na verigi obstaja cikel. Ta rešitev porabi manj pomnilnika kot prejšnja (ker ne potrebuje tabele s podatki o obiskanosti), pač pa več časa (ker naredi n korakov, četudi se cikel mogoče pojavi že veliko prej).

Oglejmo si implementacijo prve od teh dveh možnosti v C++:

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

struct Preusmeritev { int s, na; };

int KonecVerige(int n, int z, const vector<Preusmeritev> &preusmeritve)
{
  // Pripravimo si tabelo, ki za vsako stran pove, kam nas
  // od tam preusmerijo; če nikamor, bo tam −1.
  vector<int> kam(n + 1, −1);
  for (const auto &p : preusmeritve) kam[p.s] = p.na;

  // Pripravimo si tabelo za označevanje že obiskanih strani.
  vector<bool> obiskana(n + 1, false);

  // Sledimo verigi preusmeritev od z naprej.
  while (kam[z] >= 0 && ! obiskana[z])
  {
    obiskana[z] = true; // Označimo trenutno stran za obiskano
    z = kam[z];         // in se premaknimo na naslednjo.
  }

  // Če smo se ustavili zato, ker smo prišli na neko že obiskano stran,
  // to pomeni, da se je veriga zaciklala; sicer pa vrnimo stran, pri kateri
  // se je veriga končala.
  return obiskana[z] ? −1 : z;
}
```

Vektorja kam in obiskana smo inicializirali na velikost $n + 1$ elementov, da lahko kot indekse uporabljamo števila od 1 do n, kot se pojavljajo v vhodnih podatkih — elementov na indeksu 0 tako nikoli ne uporabimo. Če nas ta majhna potrata pomnilnika moti, bi morali pač paziti na to, da indekse v vhodnih podatkih pred obdelavo zmanjšamo za 1, na koncu pa številko strani, pri kateri se je veriga končala, spet povečamo za 1, preden jo vrnemo.

Namesto tabel lahko za kam in obiskana uporabimo razpršeni tabeli; s tem lahko potencialno prihranimo nekaj časa in pomnilnika, kajti v zgornji rešitvi imata obe tabeli vedno po n elementov, četudi je preusmeritev mogoče malo in četudi mogoče obiščemo le majhno število strani; pri razpršeni tabeli pa ima lahko kam le toliko elementov, kolikor je preusmeritev v vhodnem seznamu, obiskana pa le toliko elementov, kolikor strani obiščemo pri sledenju verigi. V C++ si lahko pomagamo z razredoma unordered_map in unordered_set. Še ena prednost tega pristopa je, da ga lahko skoraj brez sprememb uporabimo tudi, če so strani namesto z zaporednimi številkami od 1 do n predstavljene z nizi (npr. z naslovi, torej URLji). Oglejmo si še to rešitev:

```
#include <unordered_set>
#include <unordered_map>

int KonecVerige2(int n, int z, const vector<Preusmeritev> &preusmeritve)
{
  // Pripravimo si slovar preusmeritev.
  unordered_map<int, int> kam;
  for (const auto &p : preusmeritve) kam.emplace(p.s, p.na);

  // Pripravimo si množico že obiskanih strani.
  unordered_set<int> obiskana;

  // Sledimo verigi preusmeritev od z naprej.
  while (obiskana.find(z) == obiskana.end())
  {
    // Poglejmo, ali obstaja s strani „z“ preusmeritev kam drugam.
    auto it = kam.find(z);

    // Če ne obstaja, se veriga tu konča.
    if (it == kam.end()) return z;

    obiskana.emplace(z); // Označimo trenutno stran za obiskano
    z = it->second;      // in se premaknimo na naslednjo.
  }
  // Tu vemo, da se je veriga zaciklala.
  return -1;
}
```

Zapišimo obe različici rešitve še v pythonu. Tu predpostavimo, da so v vhodnih podatkih preusmeritve predstavljene kar z urejenimi pari (tip tuple v pythonu):

```
def KonecVerige(n, z, preusmeritve):
  # Pripravimo si tabelo, ki za vsako stran pove, kam nas
  # od tam preusmerijo; če nikamor, bo tam -1.
  kam = [-1] * (n + 1)
  for (s, na) in preusmeritve: kam[s] = na

  # Pripravimo si tabelo za označevanje že obiskanih strani.
  obiskana = [False] * (n + 1)
```

```
    # Sledimo verigi preusmeritev od z naprej.
    while kam[z] >= 0 and not obiskana[z]:
        obiskana[z] = True # Označimo trenutno stran za obiskano
        z = kam[z]         # in se premaknimo na naslednjo.

    # Če smo se ustavili zato, ker smo prišli na neko že obiskano stran,
    # to pomeni, da se je veriga zaciklala; sicer pa vrnimo stran, pri kateri
    # se je veriga končala.
    return −1 if obiskana[z] else z

def KonecVerige2(n, z, preusmeritve):
    # Pripravimo si slovar preusmeritev.
    kam = {s: na for (s, na) in preusmeritve}

    # Pripravimo si množico že obiskanih strani.
    obiskana = set()

    # Sledimo verigi preusmeritev od z naprej.
    while z not in obiskana:

        # Poglejmo, ali obstaja s strani „z“ preusmeritev kam drugam.
        # Če ne obstaja, se veriga tu konča.
        if z not in kam: return z

        obiskana.add(z) # Sicer označimo trenutno stran za obiskano
        z = kam[z]      # in se premaknimo na naslednjo.

    # Tu vemo, da se je veriga zaciklala.
    return −1
```

5. Odstranjevanje črk

Besedam, ki imajo v nalogi opisano lastnost, bomo rekli, da so *ugodne*. Iz definicije v besedilu naloge vidimo, da je beseda ugodna, če je dolga eno samo črko ali pa če lahko iz nje z brisanjem ene črke dobimo kakšno drugo ugodno besedo; sicer pa ni ugodna. Z brisanjem ene črke seveda nastane malo krajša beseda; ko se na primer ukvarjamo z neko besedo dolžine k znakov, bomo z brisanjem dobili besede dolžine $k-1$ znakov. Koristno bo torej, če bomo takrat za tiste krajše besede že vedeli, ali so ugodne ali ne, saj je od tega potem odvisno, ali je tudi daljša beseda ugodna. Naš vhodni seznam besed je torej pametno za začetek urediti naraščajoče po dolžini in jih potem pregledovati v tem vrstnem redu.

Pri vsaki besedi moramo na vse možne načine pobrisati eno črko in preveriti, če je kakšna od tako dobljenih krajših besed že znana kot ugodna. V ta namen je koristno, če ugodne besede shranjujemo v razpršeno tabelo ali kakšno podobno podatkovno strukturo, pri kateri bomo lahko poceni preverili, ali vsebuje neko besedo ali ne. V C++ lahko uporabimo na primer razred unordered_set, v pythonu pa set.

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

string Splatters(vector<string> besede)
{
    // Uredimo besede po naraščajoči dolžini.
    sort(besede.begin(), besede.end(), [] (const string &s, const string& t) {
        return s.length() < t.length(); });
```

```
  // Pripravimo razpršeno tabelo, v katero bomo shranjevali ugodne besede.
  unordered_set<string> ugodne;
  int n = besede.size(), zadnjaUgodna = −1;
  // Preglejmo besede po naraščajoči dolžini.
  for (int i = 0; i < n; i++)
  {
    const string &s = besede[i];
    // Enočrkovne besede so ugodne že same po sebi.
    int k = s.length();
    if (k <= 1) { ugodne.emplace(s); continue; }
    // Pri daljših besedah poglejmo, če lahko z brisanjem ene črke dobimo ugodno besedo.
    for (int j = 0; j < k; j++)
    {
      string t = s.substr(0, j) + s.substr(j + 1);
      // t je beseda, ki nastane, če v s pobrišemo črko s[j].
      if (ugodne.find(t) == ugodne.end()) continue;
      // t je ugodna, torej je s tudi.
      ugodne.emplace(s); zadnjaUgodna = i; break;
    }
  }
  // Vrnimo zadnjo ugodno besedo, ki smo jo našli (ta je tudi najdaljša).
  return zadnjaUgodna < 0 ? string() : besede[zadnjaUgodna];
}
```

Oglejmo si še rešitev v pythonu:

```
def Splatters(besede):
  ugodne = set(); zadnjaUgodna = None
  # Pregledujmo besede po naraščajoči dolžini.
  for k, s in sorted((len(s), s) for s in besede):
    # Enočrkovne besede so ugodne že same po sebi.
    if k <= 1: ugodne.add(s); continue
    # Pri daljših besedah poglejmo, če lahko z brisanjem ene črke dobimo ugodno besedo.
    for j in range(k):
      # Poglejmo, ali je beseda, ki nastane iz s z brisanjem črke s[j], ugodna.
      if s[:j] + s[j + 1:] not in ugodne: continue
      # Če da, potem je tudi s ugodna.
      ugodne.add(s); zadnjaUgodna = s; break
  # Vrnimo zadnjo ugodno besedo, ki smo jo našli (ta je tudi najdaljša).
  return zadnjaUgodna
```

Kakšna je časovna zahtevnost te rešitve? Recimo, da imamo n vhodnih nizov, pri čemer je i-ti dolg d_i znakov, najdaljši je dolg $m = \max_i d_i$ znakov, vsi skupaj pa $D = \sum_i d_i$ znakov. Za urejanje nizov po dolžini v resnici ni treba premikati samih nizov, lahko bi imeli le njihove indekse ali kazalce nanje; urejanje bi potem vzelo $O(n \log n)$ časa ali pa celo le $O(n+m)$ časa, če uporabimo urejanje s štetjem (kar je koristno, če nizi niso predolgi). Potem imamo pri vsaki besedi s zanko po vseh črkah; krajše besede t, ki nastanejo iz nje, bi lahko z nekaj pazljivosti računali tako, da bi imeli vsakič le $O(1)$ dela, da popravimo niz $s[1..i-1]s[i+1..k]$ v niz $s[1..i]s[i+2..k]$. Toda za poizvedbo v razpršeno tabelo ugodne bo še vseeno treba $O(k)$ časa; in ker imamo pri posameznem nizu s po k takih poizvedb, je to $O(k^2)$. Ko to seštejemo

po vseh vhodnih nizih, imamo $O(\sum_i d_i^2) = O(nm^2)$. V tem se potem utopi tudi čas za dodajanje ugodnih besed v razpršeno tabelo.

Bolje bi šlo z Rabin-Karpovimi razprševalnimi kodami, kjer lahko v $O(k)$ časa pripravimo razprševalne kode vseh prefiksov in sufiksov s-ja, iz tega pa v $O(k)$ časa tudi razprševalne kode vseh t-jev (torej besed, ki nastanejo iz s-ja z brisanjem ene črke). Predstavljamo si lahko, da če za razprševalno kodo niza t opazimo, da je v tabeli ugodne, je potem zelo verjetno, da je tudi tisti niz t res v tabeli ugodne (torej da med razprševalnimi kodami ni trkov oz. da jih je zelo malo), torej bomo sicer še lahko porabili $O(k)$ časa za preverjanje, ali je t v tabeli ugodne, vendar le enkrat pri vsakem s, ker bo tisto preverjanje pokazalo, da je t res ugoden, zato je tudi s ugoden in se bomo z njim nehali ukvarjati. Tako porabimo le $O(k)$ časa pri vsakem s-ju, kar je skupaj $O(\sum_i d_i) = O(D)$.

Razmislimo zdaj še o različici naloge, ki jo omenja opomba pod črto na koncu besedila: recimo torej, da se sme po vsakem brisanju črke tudi poljubno spremeniti vrstni red ostalih črk. Preprost način, da to novo različico naloge prevedemo na prvotno, je ta, da v seznamu besede, ki ga dobimo kot vhodni podatek, na začetku pri vsaki besedi uredimo njene črke po abecedi. Namesto besed *kramp* in *park* bi tako dobili *akmpr* in *akpr*, tu pa lahko drugo dobimo iz prve že samo z brisanjem ene črke in ni treba po tem še spreminjati vrstnega reda črk.

Ko razmišljamo o brisanju ene črke iz besede s, lahko opazimo še naslednje: če je v s več zaporednih enakih črk, je vseeno, katero od njih pobrišemo, saj bo rezultat vsakič enak (na primer: če v *bcc* pobrišemo drugo ali tretjo črko, v obeh primerih dobimo *bc*); torej je dovolj, če od več zaporednih enakih črk poskusimo brisati le eno. (To drobno izboljšavo bi lahko uporabili že v prvotni različici naloge, le da se tam verjetno precej redkeje zgodi, da ima beseda več zaporednih enakih črk; pri naši novi različici pa se bo to zgodilo pogosteje, ker smo v vsaki besedi najprej uredili črke po abecedi, tako da so enake črke gotovo prišle skupaj.)

Če so naše vhodne besede zelo dolge, je koristna potem še naslednja izboljšava: namesto da besedo (s črkami, urejenimi po abecedi) predstavimo kot niz, jo lahko predstavimo z urejeno c-terico celih števil, pri čemer je c število različnih črk v abecedi, iz katerih so sestavljene naše besede; v taki c-terici vsako število pove, kolikokrat se posamezna črka abecede pojavlja v naši besedi. Besede, ki iz nje nastanejo z brisanjem ene črke, dobimo potem tako, da v taki c-terici po en pozitivni element naenkrat zmanjšamo za 1. (Primer: če imamo abecedo s tremi črkami, recimo a, b in c, potem besedo *bcc* predstavlja trojica $(0, 1, 2)$; iz nje lahko z brisanjem ene črke dobimo $(0, 0, 2)$, torej *cc*, ali pa $(0, 1, 1)$, torej *bc*.) V razpršeno tabelo ugodne zdaj ne shranjujemo več besed, pač pa take urejene c-terice. Prednost te rešitve je v tem, da računanje vseh možnih t, ki nastanejo iz s z brisanjem ene črke, vzame le $O(c)$ časa; ravno tako tudi preverjanje, ali je beseda t v ugodne ali ne, vzame le $O(c)$ časa; naš postopek tako postane neodvisen od dolžine vhodnih besed (razen na samem začetku, ko jih predela v urejene c-terice). To je koristno, če so besede dolge v primerjavi z velikostjo abecede.

Lahko bi šli še korak dlje in izkoristili dejstvo, da so si c-terice za različne t-je, ki nastanejo iz istega s z brisanjem ene črke, med seboj zelo podobne, in ravnali podobno kot pri prej omenjenem pristopu z Rabin-Karpovimi razprševalnimi kodami. Če besedo s predstavlja c-terica $(u_1, \ldots, u_c)$, potem besedo t, ki nastane iz s

z brisanjem enega izvoda i-te črke abecede, predstavlja c-terica $(u_1, \ldots, u_{i-1}, u_i - 1, u_{i+1}, \ldots, u_c)$. Če torej vnaprej pripravimo Rabin-Karpove kode za prefikse in sufikse s-jeve c-terice, torej za zaporedja $(u_1, \ldots, u_j)$ in $(u_j, \ldots, u_c)$ za $1 \leq j \leq c$, lahko iz dveh takih kod izračunamo v $O(1)$ časa tudi razprševalno kodo t-jeve c-terice. Tako potrebujemo le $O(c)$ časa za izračun razprševalnih kod *vseh* t-jev pri danem s; časovna zahtevnost celotne rešitve bo le še $O(D + nc)$.

Naloge so sestavili: gesla — Nino Bašić; sredinec, kapniki, tja in spet nazaj, taksi — Tomaž Hočevar; kako dobri so virusni testi? — Boris Horvat; socialno omrežje, proizvodnja cepiva, odstranjevanje črk — Vid Kocijan; zlaganje loncev, virus v Timaniji — Jurij Kodre; križci in krožci — Mitja Lasič; kovanci — Matija Lokar; svetilka — Mark Martinec; marsovci — Polona Novak; preusmerjanje — Jure Slak; pletenje puloverja — Jasna Urbančič; rekonstrukcija poti — Anže Žagar; tetris — Borut in Peter Žnidar; pangramski podniz — Janez Brank.

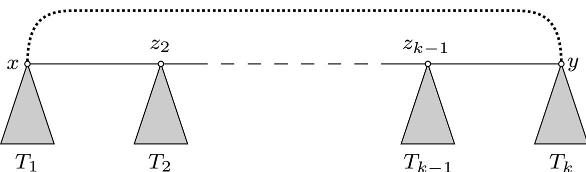
REŠITVE NALOG S CERC 2021

A. Letalska družba

Letališča si lahko predstavljamo kot točke grafa, lete pa kot neusmerjene povezave med njimi. Naloga pravi, da je med vsakima dvema točkama natanko ena pot, torej je graf povezan in acikličen — z drugimi besedami, gre za drevo. Razdaljo med točkama s in t (torej dolžino najkrajše poti med njima, merjeno s številom povezav) bomo označili z $d(s,t)$.

Recimo, da nas zanima, pri koliko parih (s,t) se dolžina najkrajše poti skrajša, če prvotnemu drevesu dodamo še povezavo (x,y). (Naloga bo zahtevala od nas, da izračunamo to za več parov (x_i, y_i), vendar se zaenkrat osredotočimo le na enega od njih.) V prvotnem drevesu je že morala obstajati neka (natanko ena) pot od x do y; recimo, da je to $\rho = \langle z_1, z_2, \ldots, z_k \rangle$, pri čemer je $z_1 = x$ in $z_k = y$, pot ρ pa je torej dolga $k-1$ korakov.

Če za poljubno točko u pogledamo razdalje $d(u, z_i)$ za $i = 1, \ldots, k$, je najmanjša izmed teh razdalj dosežena pri natanko enem i.[8] Množico tistih u, ki jim je od vseh točk na ρ najbližja ravno z_i, označimo s T_i (med njimi je seveda tudi z_i sama). Drevo si lahko torej predstavljamo v takšni obliki (debela pikčasta črta kaže načrtovano novo povezavo med x in y):



Za poljuben par točk $\{s,t\}$ lahko zdaj razmišljamo takole: če obe pripadata isti T_i, potem dodajanje povezave (x,y) ne bo nič skrajšalo poti od s do t, kajti taka pot bi morala najprej iti iz s v z_i, preden bi lahko od tam prišla do x ali y; in na koncu bi morala iti iz x ali y v z_i, preden bi lahko od tam prišla v t; potem pa bi bilo že bolje tisti vmesni del poti med prvim in zadnjim obiskom točke z_i povsem izrezati.

Ostane še možnost, da s in t ne pripadata isti T_i; recimo, da je $s \in T_i$ in $t \in T_j$ za $i \neq j$. Brez izgube za splošnost recimo, da je $i < j$. (Parov $\{s,t\}$, pri katerih se to zgodi, je potem $|T_i| \cdot |T_j|$.) V prvotnem drevesu je torej morala pot iz s v t iti najprej od s do z_i, nato po poti ρ do z_j in nato od tam v t:

$$s \rightsquigarrow z_i \to z_{i+1} \to \ldots \to z_{j-1} \to z_j \rightsquigarrow t.$$

[8] O tem se lahko prepričamo takole: recimo, da je pri i dosežena najmanjša razdalja $d(u, z_i)$; pot od u do z_i gotovo ne gre skozi nobeno drugo točko z_j, saj bi v tem primeru bila $d(u, z_j) < d(u, z_i)$. To pa pomeni, da lahko pot od u do z_i podaljšamo z enim ali več korakov po poti ρ in tako pridemo do katerekoli druge točke na ρ, na primer do z_j. Tako smo dobili pot od u do z_j, v kateri se nobena točka ne pojavi več kot enkrat; naloga zagotavlja, da je taka pot od u do z_j ena sama, njena dolžina pa je zato po definiciji enaka $d(u, z_j)$. Ker smo to pot dobili tako, da smo pot od u do z_i podaljšali za enega ali več korakov, mora biti $d(u, z_j) > d(u, z_i)$, torej je res, da je minimalna razdalja od u do točk na poti ρ dosežena samo pri z_i in ne tudi pri kakšni drugi točki z_j. □

Ko bomo dodali povezavo (x, y) oz. (z_0, z_k), bo iz poti ρ nastal cikel in osrednji del poti od s do t bo mogoče speljati po drugi strani cikla:

$$s \rightsquigarrow z_i \to z_{i-1} \to \ldots \to z_2 \to z_1 \to z_k \to z_{k-1} \to \ldots \to z_{j+1} \to z_j \rightsquigarrow t.$$

V prvotnem grafu je torej naša pot porabila $j - i$ korakov, da je prišla od z_i do z_j, v novem pa imamo še to drugo možnost, ki za to porabi $k - (j - i)$ korakov. Nova pot je torej krajša od prvotne, če je $k - (j - i) < (j - i)$, torej če je $j > i + k/2$.

Naj bo $n_i := |T_i|$; število parov $\{s, t\}$, ki se jim najkrajša pot od s do t po uvedbi nove povezave (x, y) skrajša, je torej

$$\sum_{i=1}^{k} \sum_{j>i+k/2} n_i n_j = \sum_{i=1}^{\lceil k/2 \rceil - 1} n_i \sum_{j=i+\lfloor k/2 \rfloor + 1}^{k} n_j,$$

kar lahko izračunamo v $O(k)$ časa, če najprej izračunamo delne vsote oblike $S[k] = n_k$, $S[j] = n_j + S[j+1]$ (za $j < k$) in nato seštejemo $\sum_{i=1}^{\lceil k/2 \rceil - 1} n_i \, S[i + \lfloor k/2 \rfloor + 1]$.

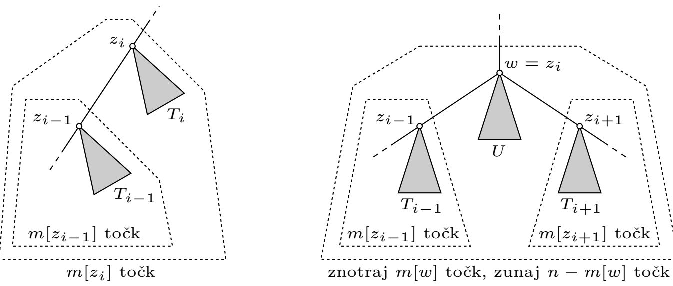
Vprašanje pa je še, kako lahko dovolj poceni izračunamo vse n_i, pri čemer moramo imeti v mislih, da bo treba v resnici odgovoriti na več poizvedb za različne (x, y). Za posamezen par (x, y) bi sicer lahko z iskanjem v širino poiskali pot ρ od x do y in nato za vsako točko z_i na njej tudi prešteli, koliko točk je dosegljivih iz nje po povezavah, ki ne pripadajo poti ρ, vendar bi nam to vzelo $O(n)$ časa in če to naredimo pri vsaki od q poizvedb, dobimo časovno zahtevnost $O(qn)$, kar bo že preveč.

Namesto tega si raje na začetku izberimo poljubno točko za koren drevesa in iz nje poženimo iskanje v širino, da določimo vsaki točki u starša $p[u]$ glede na izbrani koren. Ob tem za vsako točko u tudi izračunajmo število točk $m[u]$ v poddrevesu, ki se začne pri u:

```
naj bo Q[1..n] tabela n elementov;
i := 1; j := 1; Q[i] := koren; p[koren] := NIL;
while i ≤ j:
  u := Q[i]; m[u] := 1; i := i + 1;
  za vsako u-jevo sosedo v:
    if v = p[u] then continue;
    p[v] := u; j := j + 1; Q[j] := v;
for i := n downto 2:
  u := Q[i]; m[p[u]] := m[p[u]] + m[u];
```

Tabelo Q torej uporabljamo kot vrsto (v kateri so trenutno elementi $Q[i \ldots j]$), na koncu pa $m[u]$ vsake točke izračunamo tako, da začnemo z 1 in ji prištejemo vrednosti $m[v]$ vseh u-jevih otrok v.

Recimo zdaj, da bi radi ocenili par (x, y). Pot od x do y ne more potekati drugače kot tako, da gre najprej 0 ali več korakov gor po drevesu do najglobljega skupnega prednika točk x in y, nato pa 0 ali več korakov dol po drevesu do točke y. Če je pot od x do y dolga k korakov, je tudi skupni prednik največ k nivojev nad točkama x in y; najti ga je sicer mogoče v $O(\log n)$ časa, vendar je za naš namen dovolj dobro že, če ga najdemo v $O(k)$ časa, saj bomo toliko časa porabili tudi za preostanek izračuna pri tem paru (x, y). Lahko se torej hkrati vzpenjamo (s pomočjo tabele staršev $p[\cdot]$, ki smo jo pripravili na začetku) iz točk x in y in si označujemo že obiskana vozlišča; ko se prvič zgodi, da na neko vozlišče — recimo



Dva primera izračuna vrednosti $n_i = |T_i|$.

Levo: z_i leži na poti med x in w, zato tvorijo množico T_i tiste točke, ki so v poddrevesu z začetkom pri z_i, ne pa tudi v tistem z začetkom pri z_{i-1}; tako velja $n_i = m[z_i] - m[z_{i-1}]$.

Desno: $z_i = w$, zato tvorijo množico točk T_i vse tiste točke, ki bodisi sploh niso v w-jevem poddrevesu (takih je $n - m[w]$) bodisi so v njem, ne pa v poddrevesih njegovih otrok z_{i-1} in z_{i+1}. Ta druga skupina je na sliki označena z U in torej vsebuje $m[w] - m[z_{i-1}] - m[z_{i+1}]$ točk.

---

w — naletimo že drugič, mora biti to najgloblji skupni prednik točk x in y. Poti $x \rightsquigarrow w$ in $y \rightsquigarrow w$ lahko sestavimo s pomočjo tabele p, nato pa drugo pot obrnemo in jo staknemo s prvo, pa dobimo $x \rightsquigarrow w \rightsquigarrow y$, torej ravno pot ρ od x do y, ki smo jo iskali.

Na tej poti moramo zdaj za vsako točko z_i izračunati njen n_i, torej število točk, ki jim je z_i bližja kot katerakoli druga točka na poti ρ. (1) Pri $z_0 = x$ pridejo v poštev ravno vse točke iz x-ovega poddrevesa, torej $n_0 = m[x]$; podobno je tudi na drugem koncu poti: $n_k = m[y]$. (2) Če gledamo z_i nekje na poti med x in w, pridejo v poštev vse točke iz z_i-jevega poddrevesa razen tistih, ki so v poddrevesu točke z_{i-1} (ki je otrok točke z_i in njen predhodnik na poti ρ), torej je takrat $n_i = m[z_i] - m[z_{i-1}]$. Podobno je tudi na drugem koncu poti: če je z_i nekje na poti med y in w, je $n_i = m[z_i] - m[z_{i+1}]$. (3) Ostane še primer $z_i = w$; takrat pridejo v poštev vse točke drevesa razen tistih iz poddreves točk z_{i-1} in z_{i+1}, ki sta otroka točke z_i in njegova prednik ter naslednik na poti ρ; takrat je torej $n_i = n - m[z_{i-1}] - m[z_{i+1}]$. (Za primera (2) in (3) glej sliko zgoraj.)

Tako lahko torej za vsak par (x, y) v $O(k)$ časa (kjer je $k = d(x, y)$ razdalja med točkama x in y) poiščemo najgloblǰega skupnega prednika točk x in y, sestavimo pot ρ od x do y ter za vsako točko z_i na tej poti določimo n_i, nato izračunamo delne vsote $S[i]$ in končno izračunamo število vseh parov (s, t), pri katerih se najkrajša pot med s in t skrajša, če dodamo povezavo (x, y). Vsega skupaj je torej časovna zahtevnost naše rešitve $O(n + \sum_i k_i)$, kjer gre vsota po vseh q poizvedbah iz vhodnih podatkov. V najslabšem primeru je sicer posamezen k_i lahko $O(n)$, vendar naloga zagotavlja, da njihova vsota ne bo prevelika.

B. Gradnja na Luni

Dogovorimo se, da bomo luči na tistih stranicah, kjer je hodnik spojen z dvorano, vedno šteli k hodniku in ne k dvorani. Pri dvorani tako pridejo v poštev le tiste tri stranice, na katerih ni vrat. Recimo, da bi si za vse take stranice vseh dvoran

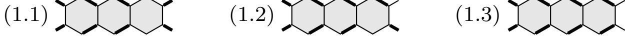
nekako izbrali, ali bodo imele prižgano luč ali ne. Kakšno je potem največje skupno število prižganih luči, ki ga lahko dosežemo, če primerno izberemo, katere luči bomo prižgali v hodnikih?

Tiste dvoranske stranice, ki nimajo vrat (in smo torej zanje že vnaprej določili, ali bodo imele prižgano luč ali ne), vedno stojijo med dvema takima stranicama, ki imata vrata (in ki ju bomo, kot smo se dogovorili, šteli k hodniku namesto k dvorani). Lahko bi torej rekli, da dvoranska stranica brez vrat povezuje dva hodnika; pa prerežimo vsako tako stranico na pol in prištejmo vsako od nastalih polovic k enemu od obeh hodnikov. (Tudi pri štetju prižganih luči bo potem taka stranica, če je luč na njej prižgana, prispevala po pol luči k vsakemu od obeh hodnikov.) Hodnik ima potem takšno obliko (na slikah bomo kazali primere za $L = 3$, razmislek pa seveda velja tudi na splošno):

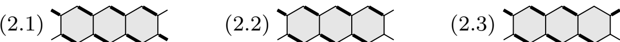
a_1 b_2
a_2 b_1

Zdaj je torej cela zgradba sestavljena le iz takšnih dopolnjenih hodnikov, pri čemer za polovične stranice, ki smo jih na gornji sliki označili z a_1, a_2, b_1 in b_2, že vemo, ali imajo prižgano luč ali ne. Na ostale stranice hodnika vplivajo le te štiri polovične stranice, nič pa ne vpliva na hodnik tisto, kar pripada drugim hodnikom. Maksimalno število luči lahko torej določimo za vsak hodnik posebej, neodvisno od drugih. Stanje polovičnih stranic bomo predstavili s števili: $a_1 = 1$ pomeni, da je na stranici a_1 luč prižgana, $a_1 = 0$ pa, da je ugasnjena; in podobno za ostale tri stranice. Razmislimo zdaj, kakšno je največje možno število prižganih luči v odvisnosti od a_1, a_2, b_1 in b_2.

(1) Recimo, da je $a_1 = a_2 = 1$. (1.1) Če je tudi $b_1 = b_2 = 1$, lahko prižgemo na hodniku največ $2L - 2$ luči; skupaj s tistimi na polovičnih stranicah (ki jih, ne pozabimo, štejemo le polovično) je to skupaj $2L$. (1.2) Če je $b_1 = 1$ in $b_2 = 0$, lahko prižgemo $2L - 1$ luči, kar skupaj s polovičnimi dá $2L + \frac{1}{2}$. Pri $b_1 = 0$ in $b_2 = 1$ je razmislek podoben. (1.3) Če pa je $b_1 = b_2 = 0$, lahko prižgemo na hodniku $2L$ luči, in sicer na en sam način. Skupaj z lučema na polovičnih stranicah a_1 in a_2 je to $2L + 1$ luči.



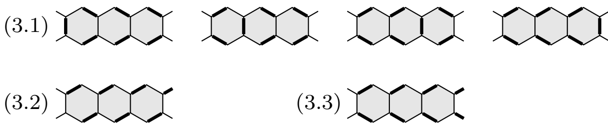
(2) Recimo zdaj, da je $a_1 = 1$ in $a_2 = 0$. (2.1) Če je tudi $b_1 = 1$ in $b_2 = 0$, lahko prižgemo na hodniku največ $2L$ luči, in sicer na en sam način; skupaj s polovičnimi je to $2L + 1$. (2.2) Tudi če je $b_1 = b_2 = 0$, lahko prižgemo na hodniku največ $2L$ luči; skupaj s polovičnimi pa je to zdaj le $2L + \frac{1}{2}$. (2.3) Če pa je $b_2 = 1$, lahko prižgemo na hodniku največ $2L - 1$ luči (ne glede na to, ali je b_1 enak 0 ali 1); skupaj s polovičnimi lučmi tako dobimo $2L$ (če je $b_1 = 0$; to je primer na sliki) ali $2L + \frac{1}{2}$ (če je $b_1 = 1$).



Primer, ko je $a_1 = 0$ in $a_2 = 1$, lahko obravnavamo po enakem kopitu kot (2), zato tega ne bomo pisali posebej.

(3) Ostane še možnost, da je $a_1 = a_2 = 0$. (3.1) Če je tudi $b_1 = b_2 = 0$, lahko na hodniku prižgemo največ $2L + 1$ luči, in sicer na $L + 1$ načinov. Hodnik ima namreč

$L+1$ stranic, ki so pravokotne na smer hodnika (na naši sliki so to navpične stranice, ker je hodnik vodoraven) in luč lahko prižgemo na eni od njih; v členih levo od tam se potem ponavlja vzorec , desno pa vzorec . V vsakem primeru hodnik tu prispeva $2L+1$ luči, saj so polovične tu vse ugasnjene. (3.2) Če je $b_1 = 0$ in $b_2 = 1$, lahko na hodniku prižgemo največ $2L$ luči (sicer na več načinov, na sliki je le eden od njih); skupaj s polovičnimi lučmi dobimo $2L + \frac{1}{2}$. Na enak način obravnavamo tudi primer, ko je $b_1 = 1$ in $b_2 = 0$. (3.3) Ostane še primer, ko je $b_1 = b_2 = 1$. To je le zrcalna slika primera (1.3); tudi tu lahko torej na hodniku prižgemo največ $2L$ luči, in sicer na en sam način; skupaj s polovičnimi lučmi dobimo $2L + 1$.



Vidimo torej, da lahko hodnik (s polovičnimi stranicami sosednjih dvoran vred) prispeva največ $2L + 1$ luči. Ker ima vsaka od n dvoran po tri hodnike in ker vsak hodnik povezuje dve dvorani, je hodnikov skupaj $m = 3n/2$; število luči torej ne more preseči $(2L + 1)m$. To število pa lahko tudi res dosežemo, na primer tako, da pustimo v tistih stranicah dvoran, ki nimajo vrat, vse luči ugasnjene; potem pridejo vsi hodniki pod primer (3.1) in lahko torej v vsakem prižgemo $2L + 1$ luči.

Veljavne osvetlitve so torej tiste z $(2L + 1)m$ lučmi; in ker imamo m hodnikov, vsak od njih pa lahko prispeva kvečjemu $2L+1$ luči, lahko skupno število $(2L+1)m$ luči dosežemo le tako, da prav vsak hodnik prispeva po $2L + 1$ luči (in ne manj kot toliko). Upoštevati pa moramo še, da lahko nekateri hodniki prispevajo toliko luči na več različnih načinov; ker so hodniki neodvisni med sabo, moramo njihove prispevke zmnožiti med seboj, da zajamemo vse možne veljavne osvetlitve pri danem izboru osvetlitev dvoranskih stranic brez vrat. Posamezen hodnik prispeva $2L + 1$ luči na toliko načinov:

$$G(a_1, a_2, b_1, b_2) := \begin{cases} 1, & \text{če je } a_1 = a_2 \neq b_1 = b_2 \text{ — primera (1.3) in (3.3);} \\ 1, & \text{če je } a_1 = b_1 \neq a_2 = b_2 \text{ — primer (2.1);} \\ \text{L + 1}, & \text{če je } a_1 = a_2 = b_1 = b_2 = 0 \text{ — primer (3.1);} \\ 0 & \text{sicer.} \end{cases}$$

Ker so naši testni primeri majhni, je dovolj dobra rešitev že ta, da vse možne osvetlitve dvoran pregledujemo z rekurzijo, pri čemer za hodnike med njimi že sproti preverjamo, ali ustrezajo pravkar omenjenim pogojem, in množimo med seboj število načinov, na katero lahko posamezni hodniki prispevajo po $2L+1$ luči. Če je to število pri kakšnem hodniku 0, z rekurzijo ni treba nadaljevati, saj to pomeni, da tisti hodnik prispeva manj kot $2L + 1$ luči in na ta način ne bomo dobili veljavne osvetlitve. Zapišimo to rešitev s psevdokodo:

globalne spremenljivke:
  $a[1..n]$ = vrstni red, v katerem obiskujemo dvorane;
  $s[1..n]$ = osvetlitve posameznih dvoran;
  r = rezultat, ki ga iščemo; na začetku 0;

podprogram Rekurzija(k, w):

```
1   u := a[k];   (* Dvorana, s katero se zdaj ukvarjamo. *)
2   za vsako od 2^3 možnih kombinacij tega, na katerih stranicah brez vrat
    prižgemo luči v dvorani u:
3      zapiši trenutno kombinacijo v s[u];
4      f := 1;
5      za vsak hodnik, ki povezuje u z neko dvorano v, ki je
       v vrstnem redu a pred u:
6         iz s[u] in s[v] izračunaj za ta hodnik a1, a2, b1, b2, iz tega pa število
          načinov g = G(a1, a2, b1, b2), kako lahko ta hodnik prispeva 2L + 1 luči;
7         f := f · g;
8      if k = n then r := r + f · w;
9      else if f ≠ 0 then Rekurzija(k + 1, f · w);
```

Rekurzivni podprogram torej po vrsti izbira osvetlitev dvoran, v parametru w pa prenaša zmnožek števila načinov, na katerega lahko prispevajo $2L + 1$ luči tisti hodniki, ki povezujejo dve že obdelani dvorani (taki, ki smo jima že izbrali osvetlitev). Če ta zmnožek kdaj pade na 0, z rekurzijo nima smisla nadaljevati; če pa uspemo izbrati osvetlitev vseh n dvoran, nam ta zmnožek takrat pove, koliko veljavnih osvetlitev celotne zgradbe lahko dobimo pri tej osvetlitvi dvoran — to potem prištejemo globalni spremenljivki r, v kateri na koncu dobimo rezultat, po katerem sprašuje naloga. (V resnici moramo seveda ves čas računati po modulu $10^6 + 3$, česar gornja psevdokoda ne prikazuje posebej.)

Za vrstni red, v katerem bomo obiskovali dvorane, smo zgoraj predpostavili, da ga imamo v globalni tabeli a; rezultat bo enak ne glede na ta vrstni red, saj rekurzija prej ali slej pregleda vse dvorane in zato tudi vse hodnike. Je pa koristno, če čim prej pregledamo čim več hodnikov, ker imamo tako več možnosti, da bomo že zgodaj ugotovili, če kakšen od hodnikov ne more prispevati $2L + 1$ luči in je zato osvetlitev že doslej obdelanih dvoran brezupna. Lahko bi torej vrstni red sestavili tako, da bi na vsakem koraku dodali vanj tisto dvorano, ki ima hodnike do največ takih dvoran, ki smo jih v vrstni red postavili že prej. (Tudi ni nujno, da je vrstni red zares fiksen in enak v vseh vejah rekurzije; lahko bi v vrstici 1 podprograma Rekurzija vsakič sproti izbrali za u tisto izmed še neobdelanih dvoran, pri kateri bo v vrstici 9 potem nastalo najmanjše število vgnezdenih klicev.)

Za majhne testne primere, kakršne smo imeli na našem tekmovanju, je dosedanja rešitev čisto dovolj dobra; vseeno pa si oglejmo še, kako jo lahko izboljšamo z dinamičnim programiranjem. Izberimo si spet vrstni red, v katerem bomo obravnavali dvorane: $u_1, u_2, \ldots, u_n$ (o tem, kako ga izbrati, bomo razmislili malo kasneje). Naj bo $U_k := \{u_1, \ldots, u_k\}$ množica prvih k dvoran v tem vrstnem redu; naj bo N_k množica tistih hodnikov, ki imajo obe krajišči v U_k (notranji hodniki), Z_k pa množica tistih hodnikov, ki imajo eno krajišče v U_k, drugo pa ne (zunanji hodniki).

Kot smo že videli, si lahko pri vsaki dvorani za tri stranice (tiste brez vrat) izberemo, ali bodo osvetljene ali ne; tako imamo za vsako dvorano osem možnih osvetlitev, recimo jim $S = \{0, 1\}^3$. Če izberemo osvetlitev prvih k dvoran, lahko to opišemo s funkcijo $h : U_k \to S$. Množica vseh možnih osvetlitev prvih k dvoran je torej $\mathcal{H}_k := S^{U_k}$. Pri taki osvetlitvi h lahko za poljuben notranji hodnik $e \in N_k$ že izračunamo a_1, a_2, b_1 in b_2 (ker poznamo osvetlitev obeh dvoran, ki ju ta hodnik povezuje) in iz tega število načinov (lahko tudi 0), na katere ta hodnik

prispeva $2L+1$ luči; temu številu recimo $G(h,e)$. Zmnožek tega po vseh notranjih hodnikih označimo z $G_k(h) := \prod_{e \in N_k} G(h,e)$ — na toliko načinov lahko (pri tem h) notranji hodniki iz N_k prispevajo maksimalno število luči. Rezultat, po katerem na koncu sprašuje naloga, je potem $\sum_{h \in \mathcal{H}_n} G_n(h)$. Težava je seveda, da vseh $h \in \mathcal{H}_n$ ne moremo pregledati eksplicitno, saj jih je preveč, 8^n; prav to pa počne naša prej omenjena rekurzivna rešitev (le s to izboljšavo, da se poskuša čim prej nehati ukvarjati s tistimi h, za katere je že jasno, da bodo dali $G_n(h)=0$; ta izboljšava sicer kar precej pomaga[9]). Opazimo lahko, da če neko osvetlitev $h \in \mathcal{H}_{k-1}$ dopolnimo tako, da izberemo še osvetlitev naslednje dvorane u_k — recimo novi osvetlitvi $h' \in \mathcal{H}_k$ — lahko zdaj $G_k(h')$ nove osvetlitve izračunamo iz $G_{k-1}(h)$ stare osvetlitve, iz stanja stranic (v h) ob vseh zunanjih hodnikih Z_{k-1} in iz stanja nove dvorane $h'(u_k)$. Hodniki $e \in N_{k-1}$, ki so bili že pred dopolnitvijo notranji, so zdaj še vedno in imajo enako stanje stranic ob krajiščih, zato je zanje $G(h',e) = G(h,e)$ — oni torej v $G_k(h')$ prispevajo enak faktor kot v $G_{k-1}(h)$, zmnožek vseh tek faktorjev pa je ravno $G_{k-1}(h)$. Ostanejo še tisti hodniki, ki pred dopolnitvijo niso bili notranji, zdaj pa so to postali, torej $e \in N_k - N_{k-1}$; tak hodnik ima eno krajišče v u_k, drugo pa v nekem $v \in U_{k-1}$, zato je z vidika množice U_{k-1} to zunanji hodnik: $e \in Z_{k-1}$. Za izračun njegovega $G(h',e)$ pa potrebujemo stanje stranic ob njem v v in v u_k. Vidimo torej, da je res, kot smo rekli: da dobimo novo $G_k(h')$ iz stare $G_{k-1}(h)$, moramo poznati osvetlitev nove dvorane $h'(u_k)$ in stanje stranic v h ob zunanjih hodnikih — ni torej treba poznati celega h-ja. Ti dodatni hodniki iz $N_k - N_{k-1}$ torej prispevajo v $G_k(h')$ nekaj faktorjev, ki jih v $G_{k-1}(h)$ ni bilo; in ti faktorji se torej zmnožijo ravno v $G_k(h')/G_{k-1}(h)$.

Če se torej zdaj več različnih h-jev iz $\mathcal{H}_{k-1}$ ujema glede stanja stranic ob zunanjih hodnikih in če potem vse te h-je dopolnimo tako, da v vseh na enak način osvetlimo naslednjo dvorano u_k, bomo dobili sicer različne h', vendar so podatki, ki so podlaga za izračun $G_k(h')/G_{k-1}(h)$, v vseh teh primerih enaki, torej je razmerje $G_k(h')/G_{k-1}(h)$ za vse enako. Ker se torej vsi ti $G_{k-1}(h)$ pomnožijo z enakim razmerjem, ko iz njih računamo $G_k(h')$, ni treba, da to počnemo za vsakega posebej, ampak lahko vzdržujemo njihovo vsoto in pomnožimo s tem razmerjem célo vsoto naenkrat.

Za $h \in \mathcal{H}_k$ definirajmo $z_k(h)$ kot stanje stranic ob vseh zunanjih hodnikih iz Z_k. Ker sta ob vsakem dve stranici, si lahko $z_k(h)$ predstavljamo kot zaporedje $2 \cdot |Z_k|$ bitov ali pa kot funkcijo oblike $Z_k \to \{0,1\}^2$. Množico vseh takih funkcij označimo z $\mathcal{Z}_k := (\{0,1\}^2)^{Z_k}$. Naj bo $\mathcal{H}_k(\zeta) = \{h \in \mathcal{H}_k : z_k(h) = \zeta\}$ — tako smo torej množico $\mathcal{H}_k$ razdelili na več podmnožic glede na to, kakšno stanje ζ imajo stranice ob zunanjih hodnikih. Robni primer nastopi pri $k=0$ in $k=n$, ko zunanjih

---

[9] Če smo vrstni red točk izbrali razumno, je vsaka dvorana u povezana z vsaj eno tako dvorano v, ki je v vrstnem redu pred njo. Ko obdelujemo u, označimo z a_1, a_2 stranici ob tem hodniku v dvorani v, z b_1, b_2 pa v dvorani u; potem je pogoj $G(a_1,a_2,b_1,b_2) > 0$ izpolnjen le v naslednjih primerih: če je $a_1 = a_2 = 1$, mora biti $b_1 = b_2 = 0$; če je $a_1 \neq a_2$, mora biti $b_1 = a_1$ in $b_2 = a_2$; če pa je $a_1 = a_2 = 0$, mora biti $b_1 = b_2$. V tem slednjem primeru imamo torej za b_1 in b_2 dve možnosti, drugače pa celó samo eno. Toda stranici b_1 in b_2 sta že dve od treh brezvratnih stranic dvorane u; za tretjo stranico sta potem tudi še dve možnosti, torej za vse tri stranice skupaj največ štiri (če je $a_1 = a_2 = 0$, sicer pa samo dve možnosti). Časovna zahtevnost naše prvotne rekurzivne rešitve bo torej bolj $O(4^n)$ kot $O(8^n)$, pa še ta ocena je pesimistična: če pri neki dvorani u pride do neugodnega scenarija s štirimi možnostmi, imajo pri nekaterih od njih nekatere stranice prižgano luč, zato na drugi strani hodnikov ob njih ne bo veljalo $a_1 = a_2 = 1$ in tam torej gotovo ne bo prišlo do scenarija s štirimi možnostmi.

hodnikov sploh ni: takrat je $Z_k = \emptyset$ in zato je edina funkcija z domeno Z_k tudi sama prazna množica, $\mathcal{Z}_k = \{\emptyset\}$.

Definirajmo $G_k(\zeta) := \sum_{h\in\mathcal{H}_k(\zeta)} G_k(h)$ — to je torej vsota števila ugodnih razporedov luči po vseh tistih delnih osvetlitvah iz $\mathcal{H}_k$, ki imajo ob zunanjih hodnikih ravno stanje ζ. Zdaj smo pripravljeni na izračun takšnih vsot z dinamičnim programiranjem:

$G_0[\emptyset] := 0$;
for $k := 1$ **to** n:
  za vsako $\zeta' \in \mathcal{Z}_k$: $G_k[\zeta'] := 0$;
  za vsako $\zeta \in \mathcal{Z}_{k-1}$ in vsako $s \in \{0,1\}^3$:
    (* *Kaj se zgodi, če osvetlitve $h \in \mathcal{H}_{k-1}(\zeta)$ dopolnimo v $h' \in \mathcal{H}_k$ tako, da osvetlitev naslednje dvorane, u_k, postavimo na s? Izračunati moramo novo stanje stranic ob zunanjih hodnikih Z_k — recimo mu ζ' — ter zmnožek faktorjev, ki ga v $G_k(h')$ prispevajo tisti hodniki, ki so zaradi dopolnitve postali notranji.* *)
    $c := 1$;
    za vsak hodnik e, ki ima eno krajišče v u_k:
      iz s (tj. osvetlitve dvorane u_k) izlušči stanje stranic ob stiku med dvorano u_k in hodnikom e; recimo temu stanju $\sigma \in \{0,1\}^2$;
      $v :=$ drugo krajišče hodnika e;
      if $v \notin U_{k-1}$:
        (* *Z vidika h' bo to zunanji hodnik, $e \in Z_k$.* *)
        $\zeta'[e] := \sigma$; **continue**;
      (* *Sicer je e zdaj notranji: $e \in Z_{k-1} \cap N_k$.* *)
      izračunaj $G(h', e)$ iz $\zeta[e]$ in σ (to je stanje stranic ob obeh krajiščih e-ja, v in u_k)
      $c := c \cdot G(h', e)$;
    za vsak hodnik $e \in Z_k$:
      $v :=$ tisto krajišče e-ja, ki ni v U_k;
      if $v = u_k$ **then continue**; (* *Take smo že obdelali.* *)
      (* *Sicer je e zunanji hodnik ne le z vidika h, ampak tudi h',* *)
      $\zeta'[e] := \zeta[e]$; (* *torej $e \in Z_k \cap Z_{k+1}$.* *)
    (* *Zdaj imamo v ζ' stanje stranic (v h') ob vseh zunanjih hodnikih iz Z_k.* *)
    $G_k[\zeta'] := G_k[\zeta'] + c \cdot G_{k-1}[\zeta]$;
return $G_n[\emptyset]$;

Pri vsaki kombinaciji ζ in s torej izračunamo tudi, kakšno je novo stanje ζ' stranic ob zunanjih hodnikih, če osvetlitev h s stanjem ζ dopolnimo v h' s tem, da dvorani u_k postavimo osvetlitev na s. Podobno kot lahko razmerje $G_k(h')/G_{k-1}(h)$ izračunamo že samo iz ζ in s, lahko tudi ζ' izračunamo že samo iz ζ in s. Večina hodnikov, ki so bili zunanji za h, je namreč zunanjih tudi za h' in moramo njihova stanja le prepisati iz ζ v ζ'; za tiste hodnike pa, ki so na novo prišli med zunanje (ker imajo eno krajišče v u_k, drugo pa zunaj U_k), lahko stanje stranic ob stiku med u_k in temi hodniki izluščimo iz osvetljave s dvorane u_k.

Na koncu, kot smo že rekli, pri $k = n$ zunanjih hodnikov sploh ni, zato je edina možna ζ le $\zeta = \emptyset$ in v $G_n(\emptyset)$ dobimo vsoto vrednosti $G_n(h)$ po vseh možnih osvetlitvah $h \in \mathcal{H}_n$ — prav to pa je rezultat, po katerem sprašuje naloga.

Pri implementaciji je koristno stanje zunanjih hodnikov, recimo $\zeta \in \mathcal{Z}_k$, predstaviti kot zaporedje $2 \cdot |Z_k|$ bitov, torej kot celo število od 0 do $4^{|Z_k|} - 1$; funkcijo $G_k(\zeta)$ pa zato predstavimo kot tabelo $4^{|Z_k|}$ elementov. Časovna zahtevnost te rešitve je $O(nd \cdot 4^d)$, če je $d = \max_k |Z_k|$ največje število zunanjih hodnikov, s katerimi imamo kdaj opravka.

Tako torej vidimo, da je za časovno zahtevnost naše rešitve ugodno, če si izberemo tak vrstni red obdelave dvoran, $u_1, u_2, \ldots, u_n$, pri katerem bo nastalo čim manj zunanjih hodnikov naenkrat. Lahko si vnaprej postavimo neko zgornjo mejo za d (začnemo na primer pri $d = 3$) in poskusimo primeren vrstni red poiskati z rekurzijo (dodajamo dvorane eno po eno v vrstni red na vse možne načine, če pa število zunanjih hodnikov kdaj preseže d, rekurzije v tisto smer seveda ne nadaljujemo); če ga ne najdemo, povečamo d za 1 in poskusimo znova in tako naprej. Pri testnih primerih na našem tekmovanju se je dalo vedno najti vrstni red, pri katerem je bilo zunanjih hodnikov največ 6 naenkrat.[10]

C. Rezanje kaktusov

Namesto da kaktus režemo na palice (poti, sestavljene iz dveh povezav), si lahko tudi predstavljamo, da ga pokrivamo z njimi. Rekli bomo, da gre palica *skozi* točko v, če je v skupna obema povezavama, ki tvorita palico.

Rešitev za drevesa. Kaktus je v nekem smislu posplošitev drevesa — pri kaktusu sme posamezna povezava pripadati največ enemu ciklu, pri drevesu pa sploh nobenemu. Razmislimo najprej o tem, kako rešiti našo nalogo za drevesa, kasneje pa si bomo ogledali, kako to rešitev razširiti na poljubne kaktuse.

V našem drevesu si za začetek izberimo poljubno točko za koren in od tam raziščimo graf (na primer z iskanjem v globino), tako da lahko med točkami vzpostavimo običajne „sorodstvene“ odnose (točka u je otrok točke v, če je v neposredni predhodnik u-ja na poti od korena do u). Poddrevesu, ki ga tvorijo točka u in vsi njeni potomci (ter povezave med njimi), recimo T_u. Množico u-jevih otrok označimo z D_u. Če je $v \in D_u$ eden od u-jevih otrok, bomo povezavi med v in u, s katero je poddrevo T_v pripeto na točko u, rekli *pecelj* tega poddrevesa.

Vprašajmo se zdaj, na koliko načinov se dá s palicami pokriti T_u. Pecelj med u in njegovim otrokom v lahko pokrijemo bodisi (1) s palico skozi u bodisi (2) s palico skozi v. V primeru (1) je potem druga povezava te palice lahko le neki drug pecelj, recimo med u in nekim $v' \in D_u$; v primeru (2) pa se mora drugi korak te palice nadaljevati iz v nekam navzdol v T_v; taka palica torej vpliva na to, kako (in na koliko načinov) se lahko pokrije T_v.

Recimo, da smo zdaj nekako pokrili vse peclje. V nadaljevanju moramo s palicami pokriti še preostanek dreves T_v za vse $v \in D_u$; toda tu ne bo nobena palica mogla hkrati ležati v dveh takih poddrevesih, saj je mogoče iz enega poddrevesa v drugo priti le prek pecljev, te pa smo že pokrili. To, kako pokrijemo T_v, torej ne

[10] Najmanjši d, ki ga je mogoče pri danem grafu (dvoran in hodnikov med njimi) doseči, se v teoriji grafov imenuje „točkovno ločilno število“ (*vertex separation number*). Pri naši nalogi imamo opravka s kubičnimi grafi (vsaka točka ima natanko tri povezave) in za dovolj velike kubične grafe na n točkah je točkovno ločilno število največ $n/6$, vendar grafi pri naši nalogi niso tako veliki, da bi si lahko s to mejo kaj pomagali. Gl. npr. Wikipedijo *s. vv.* “Pathwidth”, “Cubic graph” in tam navedeni članek: F. V. Fomin, K. Høie, “Pathwidth of cubic graphs and exact algorithms”, *Inf. Proc. Lett.*, 97(5):191–196 (2006).

more nič vplivati na to, kako lahko pokrijemo $T_{v'}$ (če sta v in v' dva različna otroka u-ja), tako da lahko za vsako poddrevo posebej izračunamo, na koliko načinov se ga dá pokriti, in potem pokritja posameznih T_v poljubno skombiniramo v pokritje celotnega T_u.

Če je poddrevo T_v sestavljeno iz liho mnogo povezav (ne vštevši peclja), ga s palicami gotovo ne moremo v celoti pokriti, saj ima vsaka palica dve povezavi; lahko pa to drevo pokrijemo tako, da skupaj z njim pokrijemo tudi njegov pecelj, saj je skupaj s slednjim število povezav potem sodo. Po drugi strani pa, če ima T_v že sam sodo mnogo povezav, ga lahko pokrijemo v celoti brez peclja, ne moremo pa ga pokriti skupaj s pecljem (ker bi imeli potem liho mnogo povezav).

Za poljubno točko u naj bo a_u število načinov, kako je mogoče pokriti s palicami poddrevo T_u (brez peclja, če ima T_u sodo mnogo povezav, oz. skupaj s pecljem, če ima T_u liho mnogo povezav); s_u pa naj bo enako 1, če pri tem pecelj ostane nepokrit, oz. 0, če je bil pokrit tudi on. Število povezav v T_u (brez peclja med u in njegovim staršem) označimo s τ_u; potem je $s_u = 1 - (\tau_u \bmod 2)$ in $\tau_u = \sum_{v \in D_u}(1 + \tau_v)$.

Če je u list in torej nima otrok, obsega T_u le vozlišče u in nobene povezave, zato velja $\tau_u = 0$ (in zato $s_u = 1$) in $a_u = 1$ (kar predstavlja pokritje z 0 palicami).

Razmislimo zdaj o tem, kako izračunati a_u za notranja vozlišča; pri tem si bomo pomagali z vrednostmi a_v za u-jeve otroke $v \in D_u$. Potem ko smo pokrili poddrevesa T_v in skupaj z njimi tudi nekatere peclje med vozlišči $v \in D_u$ in njihovim staršem u, nam ostane nepokritih še $\tilde{s}_u := \sum_{v \in D_u} s_v$ takih pecljev. Te moramo pokriti s palicami skozi u. Če je $\tilde{s}_u$ sodo število, lahko teh $\tilde{s}_u$ pecljev pokrijemo s palicami na $(\tilde{s}_u - 1)!!$ načinov;[11] T_u je s tem v celoti pokrit, torej peclja med u in njegovim staršem ne bomo pokrili in takrat je $s_u = 1$. Po drugi strani, če je $\tilde{s}_u$ liho število, bomo morali skupaj s T_u pokriti še njegov pecelj, da bomo imeli sodo mnogo povezav (namreč $\tilde{s}_u + 1$) in jih bomo lahko razdelili na pare (in sicer na $\tilde{s}_u!!$ načinov); takrat bo torej $s_u = 0$. Tako smo dobili:

$$a_u = (\tilde{s}_u - s_u)!! \prod_{v \in D_u} a_v \quad \text{za} \quad \tilde{s}_u = \sum_{v \in D_u} s_v.$$

Na koncu nas zanimajo razporedi, ki pokrijejo celotno drevo; če z r zdaj označimo koren drevesa, nam a_r pove število takih razporedov, toda le, če je $s_r = 1$; če pa je $s_r = 0$, to pomeni, da ima drevo liho število povezav in ga sploh ni mogoče razdeliti na palice (vrednost a_r pa šteje razporede, ki poleg drevesa pokrijejo še pecelj med r in njegovim staršem, ki seveda v resnici sploh ne obstaja). Iskani rezultat je torej $s_r \cdot a_r$.

Vrednosti a_u, s_u in tudi τ_u ne bi bilo težko računati od spodaj navzgor. Iz korena poženimo iskanje v globino, ki rekurzivno pregleda celotno drevo; ko se pri nekem u končajo rekurzivni klici za vse njegove otroke $v \in D_u$ in smo torej zanje že izračunali a_v in s_v, lahko zdaj izračunamo tudi a_u in s_u ter se vrnemo iz rekuzivnega klica za u nazaj v klic za njegovega starša.

Posplošitev na kaktuse. Razmislimo zdaj o tem, kako našo rešitev posplošiti z dreves na kaktuse. Tako kot pri drevesu lahko poljubno točko razglasimo za koren

[11] O tem se lahko prepričamo takole: za pecelj do otroka z najvišjo številko se lahko na $\tilde{s}_u - 1$ načinov odločimo, s katerim od preostalih pecljev bo tvoril palico; potem pa nam ostane še $\tilde{s}_u - 2$ pecljev, na katerih moramo ponoviti enak razmislek. Če označimo število razporeditev $\tilde{s}_u$ pecljev v palice s $f(\tilde{s}_u)$, velja torej $f(\tilde{s}_u) = (\tilde{s}_u - 1) \cdot f(\tilde{s}_u - 2)$, robni primer pa je $f(2) = 1$. Tako dobimo $f(\tilde{s}_u) = (\tilde{s}_u - 1) \cdot (\tilde{s}_u - 3) \cdot \ldots \cdot 3 \cdot 1 = (\tilde{s}_u - 1)!!$.

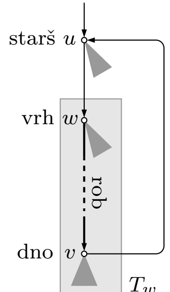
in iz nje poženemo iskanje v globino, da raziščemo celoten graf; vendar pa se lahko zdaj zgodi, da med sosedi nekega vozlišča poleg njegovega starša in otrok zagledamo tudi kakšnega bolj oddaljenega prednika. To kaže na prisotnost cikla:

```
podprogram Obišči(v):
  obiskana[v] := 1; (* Obisk je v teku. *)
  za vsako v-jevo sosedo u:
    if u = starš[v] then continue;
    else if obiskana[u] = 0:
      (* Točka u je otrok točke v. *)
      starš[u] := v; Obišči(u);
    else if obiskana[u] = 1:
      (* Točke v, starš[v], starš[starš[v]], ..., u tvorijo cikel. *)      (†)
    else: (* torej če je obiskana[u] = 2 *)
      (* Točke u, starš[u], starš[starš[u]], ..., v tvorijo cikel. *)      (‡)
      (* Z njim se nam tu ni treba ukvarjati, ker smo vse v-jeve potomke
         na tem ciklu že obiskali. *)
  obiskana[v] := 2; (* Obisk je zaključen. *)

glavni del programa:
  for v := 1 to n do obiskana[v] := 0;
  koren := poljubna točka od 1 do n; Obišči(koren);
```

Povezave med točkami in njihovimi starši še vedno tvorijo drevo in za vsako točko v lahko definiramo T_v enako kot pri drevesu; razlika je le v tem, da naš kaktus poleg teh povezav zdaj vsebuje še nekatere druge, ki povezujejo dve točki, od katerih je ena prednik (vendar ne starš) druge. Taka je povezava med v in u v vrstici (†) gornjega postopka. V ciklu, ki smo ga odkrili v tisti vrstici (gl. sliko na str. 127), je vsaka naslednja točka starš prejšnje; naj bo w zadnja točka pred u, tako da je w otrok u-ja. Toda ker imamo na koncu cikla še povezavo med u in v, je tudi v skorajda kot nekakšen otrok u-ja, le da tadva u-jeva „otroka", v in w, ne predstavljata vsak po enega poddrevesa, pač pa oba skupaj predstavljata en *podkaktus*, ki obsega pot od w do v ter vse, kar visi dol s točk na tej poti; skupaj je to ravno T_w. Pot od w do v bomo imenovali *rob* tega podkaktusa, točko w (ki je na tej poti najvišja) njegov *vrh*, točko v (ki je na tej poti najnižja) njegovo *dno*, za točko u pa bomo rekli, da je *starš* tega podkaktusa. (Vrh in dno sta gotovo dve različni točki, saj mora imeti cikel vsaj tri točke.) Za razliko od poddrevesa, ki je na starša pripeto z enim samim pecljem, je podkaktus pripet na starša u z dvema pecljema — to sta povezavi med u in vrhom ter dnom podkaktusa (točkama w in v). (Zato bomo isti cikel kot v vrstici (†) kasneje opazili še enkrat, namreč ko bomo v okviru klica Obišči(u) med naštevanjem u-jevih sosed v vrstici (‡) opazili točko v; ta primer prepoznamo po tem, da je bil obisk v-ja do takrat že zaključen in se zato s tem ciklom ni treba ukvarjati še enkrat.)

Ko smo reševali nalogo na drevesu in za neko točko u računali, na koliko načinov se dá pokriti njeno poddrevo T_u, je vsako od poddreves njegovih otrok $v \in D_u$ prispevalo k temu bodisi en nepokrit pecelj bodisi nobenega, odvisno od tega, ali je bilo število povezav v T_v liho ali sodo; in vse tako dobljene nepokrite peclje smo morali na vse možne načine združiti v pare, da smo jih pokrili s palicami skozi u.



Primer podkaktusa T_w, ki obsega vse znotraj svetlo sivega pravokotnika. Točka w je vrh podkaktusa, v je njegovo dno, pot med njima tvori rob podkaktusa; u je starš podkaktusa. Temno sivi trikotniki predstavljajo vse tiste potomce točk u, v in w, ki niso narisani posebej.

Tu pri kaktusu je stvar podobna, le da imamo poleg poddreves tudi podkaktuse. Recimo, da ima u med drugim neki podkaktus z vrhom w in dnom v. Če ima ta podkaktus T_w liho mnogo povezav (ne vštevši pecljev, torej povezav med u in w ter med u in v), se ga prav gotovo ne da pokriti s palicami, pač pa se ga mogoče da pokriti tako, da skupaj z njim pokrijemo še enega od obeh pecljev, drugi pecelj pa ostane nepokrit. To lahko obravnavamo enako kot poddrevo, če postavimo $s_w = 1$ (ker en pecelj ostane nepokrit) in s T_w označimo število načinov, kako pokriti to poddrevo z enim pecljem vred.

Če pa ima podkaktus sodo mnogo povezav, se ga prav gotovo ne da pokriti tako, da bi skupaj z njim pokrili še enega od pecljev; pač pa se ga mogoče da pokriti tako, da skupaj z njim pokrijemo oba peclja ali pa nobenega. To pa pomeni, da število nepokritih pecljev v u, torej $\tilde{s}_u$, zdaj ni več enolično določeno (kot je bilo pri drevesu), ampak ima lahko različne možne vrednosti (odvisno od tega, ali so posamezni podkaktusi prispevali po dva nepokrita peclja ali nobenega).

Oznaka D_u nam bo, tako kot doslej, pomenila množico tistih u-jevih otrok, ki so koreni poddreves; poleg tega pa vpeljimo zdaj še oznaki S_u oz. L_u za množici tistih u-jevih otrok, ki so vrhovi podkaktusov s sodim oz. lihim številom povezav (takim podkaktusom bomo krajše rekli kar „sodi“ oz. „lihi“ podkaktusi). Ostane morda še en otrok: če je u na robu podkaktusa, vendar ne na dnu, ima enega otroka, ki tudi pripada robu istega podkaktusa; naj bo R_u množica s tem otrokom (če pa takega ni, naj bo R_u prazna).[12]

S τ_u tudi zdaj označimo število povezav v T_u; ker so podkaktusi pripeti na u z dvema pecljema namesto z enim, velja zdaj

$$\tau_u = \sum_{v \in D_u \cup R_u} (1 + \tau_v) + \sum_{v \in S_u \cup L_u} (2 + \tau_v).$$

Za otroke $v \in D_u \cup L_u \cup R_u$ vemo, da je mogoče T_v pokriti na a_v načinov, pri čemer naj nam $s_v \in \{0, 1\}$ pove, koliko pecljev pri tem ostane nepokritih (pri $v \in D_u \cup R_u$ je $s_v = 1 - (\tau_v \bmod 2)$, pri $v \in L_u$ pa je s_v vedno 1). Vsi ti otroci skupaj pustijo $\tilde{s}_u := \sum_{v \in D_u \cup L_u \cup R_u} s_v$ nepokritih pecljev, pokritja posameznih T_v pa je mogoče skombinirati skupaj (ker je dogajanje v različnih T_v med seboj neodvisno) na $\tilde{a}_u := \prod_{v \in D_u \cup L_u \cup R_u} a_v$ načinov.

[12] Morebitni otrok iz R_u gotovo ni hkrati tudi vrh kakšnega podkaktusa, torej ni skrbi, da bi moral ta otrok hkrati pripadati tudi eni od množic S_u in L_u; nobena točka namreč ne more ležati na robu dveh podkaktusov hkrati, kajti v tem primeru bi povezava med njo in njenim staršem pripadala dvema cikloma hkrati, to pa je v kaktusu nemogoče.

Za otroke $v \in S_u$ (vrhove sodih podkaktusov) pa bomo morali poleg a_v (ki nam, ker je število povezav v T_v sodo, pove, na koliko načinov ga je mogoče pokriti brez pecljev) izračunati tudi število načinov, na katere je mogoče pokriti T_v z obema pecljema vred; recimo temu a'_v. Kako lahko ta različna pokritja skombiniramo med seboj, ko razmišljamo o pokrivanju celotnega T_u? Naj bo $P \subseteq S_u$ množica, ki pove, pri katerih $v \in S_u$ uporabimo pokritje brez pecljev; pri tistih iz $S_u - P$ potem uporabimo pokritje z obema pecljema. Tovrstna pokritja otrok iz S_u je mogoče skombinirati na toliko načinov:

$$f_u(P) := \left(\prod_{v \in P} a_v\right) \cdot \left(\prod_{v \in S_u - P} a'_v\right).$$

Če smo pri $t := |P|$ otrocih pustili peclja nepokrita, bo tak izbor pri u-ju pustil $2t$ nepokritih pecljev, ki se pridružijo $\tilde{s}_u$ nepokritim pecljem od otrok iz D_u, L_u in R_u. Ker imajo torej enako velike množice P podoben vpliv na pokrivanje pecljev s palicami skozi u, jih je koristno obravnavati skupaj; definirajmo zato:

$$f_u(t) := \sum_{P \subseteq S_u \,:\, |P| = t} f_u(P).$$

To, ali je nepokritih pecljev sodo ali liho mnogo (od tega je odvisno, ali bomo pri pokrivanju T_u-ja morali vključiti zraven še povezavo med u in njegovim staršem), je seveda neodvisno od t, saj se zaradi njega število nepokritih pecljev vedno spreminja v korakih po 2. Število s_u, ki naj nam pove, ali ostane (pri pokrivanju T_u-ja) povezava med u in njegovim staršem nepokrita ($s_u = 1$) ali pokrita ($s_u = 0$), lahko torej računamo enako kot pri drevesu: $s_u = 1 - (\tau_u \bmod 2)$. Število vseh pokritij T_u-ja je torej:

$$a_u = \tilde{a}_u \sum_{t=0}^{|S_u|} (\tilde{s}_u + 2t - s_u)!!\, f_u(t).$$

Kako učinkovito računati vrednosti $f_u(t)$? Lahko bi jih računali z dinamičnim programiranjem, pri čemer bi gledali podprobleme, ki nastanejo, če se omejimo na prvih nekaj otrok iz S_u; vendar bi imela taka rešitev časovno zahtevnost $O(k^2)$ za $k = |S_u|$. Pri testnih primerih na našem tekmovanju bi bilo to sicer dovolj hitro, ker gre lahko k le do približno 25 000;[13] vseeno pa si oglejmo, kako lahko pridemo do učinkovitejše rešitve.

Opazimo lahko, da zmnožki, s kakršnimi smo definirali $f_u(P)$, nastanejo, ko razvijemo produkt $\prod_{v \in S_u} (a_v + a'_v)$ v vsoto. Iz takega produkta nastane vsota 2^k zmnožkov, ker lahko pri vsakem od k faktorjev oblike $a_v + a'_v$ na dva načina izberemo, katerega od obeh členov bomo uporabili v trenutnem zmnožku. Če nam množica P pove, pri katerih faktorjih smo uporabili člen a_v in ne a'_v, ima nastali zmnožek vrednost ravno $f(P)$. Vidimo torej, da je $\prod_{v \in S_u} (a_v + a'_v) = \sum_{P \subseteq S_u} f(P)$. Vsota na desni je že precej podobna temu, kar potrebujemo za $f(t)$, vendar se moramo za te slednje znati omejiti na tiste P-je, ki imajo natanko t elementov. To najlaže dosežemo, če naše dvočlene faktorje spremenimo v polinome; definirajmo torej

$$p_u(x) := \prod_{v \in S_u} q_v(x) \quad \text{za} \quad q_v(x) := (a_v x + a'_v).$$

[13] Podkaktus mora imeti vsaj eno povezavo, saj bi bila sicer vrh in dno ena in ista točka in bi ta morala imeti dve povezavi s staršem podkaktusa, kar pa je nemogoče. Najmanjše možno sodo število povezav podkaktusa je torej 2; skupaj z obema pecljema (ki povezujeta starša podkaktusa z vrhom ter z dnom podkaktusa) pa so povezave štiri. Pri naši nalogi ima lahko graf največ $m = 10^5$ povezav, torej sodih podkaktusov ne more biti več kot $m/4 = 25\,000$.

Če podobno kot prej predelamo ta zmnožek k dvočlenih faktorjev v vsoto 2^k zmnožkov, pri čemer nam množica P pove, pri katerih otrocih smo v zmnožek sprejeli člen $a_v x$ in ne člena a'_v, nam zdaj pri množici P nastane zmnožek $f(P)x^{|P|}$; torej je $p_u(x) = \sum_{P\subseteq S_u} f(P)x^{|P|}$. Če v tej vsoti združimo skupaj tiste P-je, ki imajo natanko t elementov, lahko pri njih x^t izpostavimo, njihovi $f(P)$ pa se seštejejo v $f(t)$; dobimo torej

$$p_u(x) = \sum_{t=0}^k f(t)x^t.$$

Vrednosti $f(t)$, ki jih potrebujemo za izračun a_u, so torej ravno koeficienti polinoma $p_u(x)$, ta pa je produkt k polinomov prve stopnje (linearnih funkcij), torej $p_u(x) = \prod_{v\in S_u} q_v(x)$. Spomnimo se, da lahko s pomočjo hitrega postopka za računanje diskretne Fourierjeve transformacije (FFT — *fast Fourier transform*) zmnožimo dva polinoma dolžine k v $O(k \log k)$ časa, medtem ko bi naivna rešitev, ki bi množila vsak člen z vsakim, porabila $O(k^2)$ časa. V našem primeru lahko najprej množimo polinome q_v med seboj po dva in dva ter dobimo $k/2$ polinomov s po štirimi koeficienti; nato te množimo po dva in dva ter dobimo $k/4$ polinomov s po osmimi koeficienti; in tako naprej. Za vsak s od 1 do $r := \lceil \log_2 k\rceil$ moramo izvesti $k/2^s$ množenj polinomov stopnje 2^{s-1}, vsako tako množenje pa vzame $O(s\cdot 2^s)$ časa; skupaj je to $O(\sum_{s=1}^r (k/2^s)2^s \cdot s) = O(kr^2) = O(k(\log k)^2)$ časa. Taka je torej cena za izračun vrednosti a_u; to moramo narediti za vsako vozlišče u, pri čemer je seveda število sodih podkaktusov $k = |S_u|$ lahko pri različnih u-jih različno. Pišimo ga zato kot k_u namesto le k; časovna zahtevnost se nam zdaj po vseh u-jih sešteje v $O(\sum_u k_u(\log k_u)^2) = O(\sum_u k_u(\log n)^2) = O(n(\log n)^2)$, pri čemer smo upoštevali, da je vsota $\sum_u k_u$ manjša od n, ker je vsako vozlišče lahko vrh sodega podkaktusa pri največ enem u (koren pa sploh pri nobenem).

Pri računanju FFT je treba nekaj pazljivosti, ker imamo opraviti s precej velikimi števili. Naloga bo na koncu zahtevala rezultat po modulu $M = 10^6 + 3$ in seveda je smiselno, da že med računanjem delamo le z ostanki po deljenju z M. To pomeni, da ko množimo med seboj dva polinoma, bodo njuni koeficienti že taki ostanki, torej cela števila od 0 do $M - 1$; in tudi od koeficientov njunega zmnožka bomo obdržali le ostanke po deljenju z M. Vendarle pa moramo te koeficiente najprej pravilno izračunati s FFTjem in pri tem ne moremo računati po modulu M. Da pridemo do našega $p(x)$ kot zmnožka $\leq n$ polinomov prve stopnje, bomo v zadnjem koraku morali zmnožiti med seboj dva polinoma stopnje $\leq n/2$ (s koeficienti od 0 do $M-1$), zato bodo koeficienti njunega zmnožka veliki do približno $(n/2)M^2$.

Če računamo FFT v obsegu kompleksnih števil (kot je običajno), nam bodo napake zaradi omejene natančnosti pri računanju s plavajočo vejico prej ali slej pokvarile rezultat, če bosta n in M dovolj velika. Videli smo že, da ima lahko točka pri omejitvah iz besedila naloge največ $k = 25\,000$ sodih podkaktusov; če poskusimo množiti dva polinoma stopnje $k/2$ pri različnih M, se izkaže, da so, če uporabimo 64-bitni tip **double**, rezultati pravilni do približno $M = 7\,000$; če pa imamo na voljo 80-bitni tip s plavajočo vejico (ki ga na primer prevajalnik **g++** podpira pod imenom **long double**), gremo lahko do približno $M = 600\,000$. Oboje je torej manj od $M = 10^6 + 3$, ki ga načeloma potrebujemo pri naši nalogi.

Pomagamo si lahko tako, da koeficiente polinoma razbijemo na dva dela: vzemimo $\mu := \lceil\sqrt{M}\,\rceil = 1001$, pa lahko vsako celo število s z območja od 0 do $M - 1$ predstavimo kot $s = s_1 \cdot \mu + s_2$, pri čemer sta s_1 in s_2 celi števili od 0 do $\mu - 1$. Ko

množimo dva polinoma, recimo $g(x)$ in $h(x)$, s koeficienti z območja od 0 do $M-1$, ju torej lahko najprej razbijemo na $g(x) = g_1(x)\cdot\mu + g_2(x)$ in $h(x) = h_1(x)\cdot\mu + h_2(x)$, pri čemer imajo polinomi g_1, g_2, h_1, h_2 zdaj koeficiente od 0 do $\mu-1$ namesto od 0 do $M-1$. Potem je $g(x)\cdot h(x) = g_1(x)h_1(x)\mu^2 + (g_1(x)h_2(x) + g_2(x)h_1(x))\mu + g_2(x)h_2(x)$. Tu imamo torej štiri množenja polinomov, katerih koeficienti so le od 0 do $\mu - 1$, torej so dovolj majhni, da lahko uporabimo FFT na kompleksnih številih in računamo s tipom **double**; ko pa dobimo te štiri zmnožke, imamo od tam naprej le še celoštevilsko aritmetiko po modulu M.[14]

Izračun vrednosti a'_z za točke na robu podkaktusa. Recimo, da je $w \in S_u$, torej da je vrh podkaktusa s sodim številom povezav in da ima starša u. Potem bomo pri izračunu a_u potrebovali ne le a_w, pač pa tudi a'_w, ki pove, na koliko načinov lahko pokrijemo T_w z obema pecljema vred. Eden od teh dveh pecljev je povezava med w in njegovim staršem u, drugi pecelj pa je povezava med u in dnom podkaktusa, recimo mu v, ki leži v drevesu morda precej nivojev niže od w. Pri pokritjih, ki jih šteje a'_w, je ta drugi pecelj pokrit s palico skozi v (namesto s táko skozi u), to pa vpliva na število pokritij pri T_v in še drugih vozliščih na robu podkaktusa (torej na poti med w in v). Podatek, ali smo povezavo med v in u že pokrili ali ne, smo torej morali imeti pri roki takrat, ko smo se ukvarjali s točko v; in če ga hočemo imeti pri roki tudi kasneje, ko se na vrhu podkaktusa ukvarjamo s točko w, to pomeni, da ga bo treba prenašati gor po podkaktusu. Zato bomo vrednosti $a'_\bullet$ računali ne le za točko w na vrhu podkaktusa, pač pa tudi za vsako drugo točko na njegovem robu (in to ne le za podkaktuse s sodo mnogo povezavami, pač kar pa za vse, saj pri nižje ležečih točkah na robu podkaktusa še ne moremo vedeti, koliko povezav bo imel na koncu); če je z točka na robu podkaktusa, čigar dno je v in čigar starš je u, naj bo a'_z število načinov, na katere je mogoče pokriti s palicami celotno $T(z)$ in tudi povezavo med v in u; in če ima to dvoje skupaj liho število povezav, vključimo v pokritje še pecelj med z in njegovim staršem. (Pri $z = w$ se ta definicija točno ujema s tem, kako smo prvotno vpeljali a'_w za točko na vrhu sodega podkaktusa.)

Pri izračunu a'_z je koristno ločiti dva primera. (1) Če je $z = v$, torej dno podkaktusa (kar prepoznamo tudi po tem, da je R_z takrat prazna množica), ima zahteva, da moramo pokriti tudi povezavo med v in u, enak učinek, kot da bi imel v še enega otroka (ki pa sam ne bi imel nobenih nadaljnjih potomcev); učinek takega otroka pa bi bil, da bi se $\tilde{s}_v$ povečala za 1. To uporabimo v formuli za a_v, pa bomo kot rezultat dobili a'_v. (2) Če pa leži z nekje više gor na robu podkaktusa, naj bo $y \in R_u$ tisti njegov otrok, ki leži na robu istega podkaktusa kot z. Ta otrok je v formulo za $\tilde{a}_z$ (in s tem v a_z) prispeval faktor a_y; zdaj, pri izračunu a'_z, pa bo namesto tega prispeval faktor a'_y; nastane zmnožek $\tilde{a}'_z = a'_y \prod_{v \in D_u \cup L_u} a_v$. Poleg tega pa, ker je v T_y zdaj pokrita ena povezava več (namreč tista med v in u), kot je bila pri izračunu a_z, se je parnost števila povezav spremenila in pecelj med y in z je pokrit natanko v primeru, če prej ni bil; število nepokritih pecljev med z-jem in njegovimi otroci torej zdaj ni $\tilde{s}_z$, pač pa $\tilde{s}'_z = \tilde{s}_z - s_y + (1 - s_y)$. To, ali pri pokrivanju T_z-ja

[14] Še ena možnost bi bila tudi, da bi računali FFT v obsegu $\mathbb{Z}_P$ (cela števila od 0 do $P - 1$, pri čemer seštevamo in množimo po modulu P), vendar je s tem več dela; poiskati moramo primeren P (večji od $(k/2)\mu^2$) in zanj najti primeren koren enote; in če hočemo, da se bo dalo množiti v $\mathbb{Z}_P$ že samo s 64-bitnimi celimi števili, mora biti $P < 2^{32}$, kar nas pri $k = 25\,000$ omeji na $\mu \leq 586$; to je manj od $\sqrt{M}$, torej bi morali koeficiente naših polinomov razbiti na tri dele namesto na dva (in potem množiti vsakega z vsakim, tako da bi bilo z množenjem še več dela — devet množenj namesto štirih, torej bi bil čas izvajanja približno dvakrat daljši).

ostane pecelj med z-jem in njegovim staršem nepokrit, pa nam zdaj pove število $s'_z = 1 - (\tilde{s}'_z \bmod 2) = 1 - s_z$. Če zdaj tako spremenjene količine uporabimo v formuli za a, dobimo:

$$a'_z = \tilde{a}'_z \sum_{t=0}^{|S_z|} (\tilde{s}'_z + 2t + s'_z)!!\, f(t).$$

Zdaj imamo vse, kar potrebujemo, da lahko psevdokodo našega postopka opišemo podrobneje:

```
podprogram OBIŠČI(v):
  obiskana[v] := 1;               (* Obisk je v teku. *)
  sp[v] := NIL;                   (* To bo starš podkaktusa, na čigar robu leži v. *)
  naj bodo D_v, L_v, S_v, R_v prazne množice;

  za vsako v-jevo sosedo u:
    if u = starš[v] then continue;
    else if obiskana[u] = 1:
      (* Točke v, starš[v], starš[starš[v]], . . . , u tvorijo cikel. *)
      sp[v] := u;                 (* Točka u je starš podkaktusa, čigar dno je v. *)
    else if obiskana[u] = 0:      (* Točka u je otrok točke v. *)
      starš[u] := v; OBIŠČI(u);
      if sp[u] = v:               (* u je vrh nekega v-jevega podkaktusa. *)
        če je τ_u sod, dodaj u v S_v, sicer pa ga dodaj v L_v;
      else if sp[u] = sp[v]:
        dodaj u v R_v;            (* u in v sta na robu istega podkaktusa *)
      else: dodaj u v D_v;        (* u je koren nekega v-jevega poddrevesa. *)

  obiskana[v] := 2;               (* Obisk v-ja bo zdaj zaključen. *)
  iz rezultatov za v-jeve otroke izračunaj koeficiente polinoma p_v(x),
    nato pa tudi τ_v, ã_v, a_v, s̃_v, s_v;
  if sp[v] ≠ NIL:                 (* v je na robu podkaktusa *)
    izračunaj ã'_v, a'_v, s̃'_v, s'_v;
    (* Če v ni vrh podkaktusa, prenesimo v njegovega
       starša podatek o tem, kdo je starš tega podkaktusa. *)
    u := starš[v]; if sp[v] ≠ u then sp[u] := sp[v];

glavni del programa:
  če je m (število vseh povezav) liho, takoj vrni 0;
  for v := 1 to n do obiskana[v] := 0;
  r := poljubna točka drevesa; starš[r] := NIL;
  OBIŠČI(r); return a_r;
```

V glavnem delu programa si torej za koren izberemo poljubno točko r in iz nje poženemo iskanje v globino. Za koren r vrednost a_r pomeni število načinov, kako pokriti celoten graf (in po tem sprašuje naša naloga), vendar le, če ima sodo mnogo povezav (pri lihem številu povezav bi a_r štel pokritja, v katera je vključena tudi povezava med r in njegovim staršem, ki pa v resnici seveda ne obstaja).

D. DJ Darko

Razmislimo najprej o dogodku druge vrste, pri katerem moramo določiti novo kon-

stantno glasnost x za vse zvočnike od a_ℓ do a_d. Pri takem dogodku nastane skupina več zaporednih zvočnikov z enako glasnostjo a_i; dogodek prve vrste pa takih skupin tudi ne razbija pretirano hitro (saj prišteje več zaporednim a_i enako konstanto). Zato bo imelo pogosto po več zaporednih zvočnikov enako glasnost; táko skupino lahko (za potrebe določanja nove glasnosti pri dogodku druge vrste) obravnavamo, kot da bi bila en sam zvočnik, pri čemer cene b_i zvočnikov v skupini seštejemo, da dobimo ceno spreminjanja celotne skupine.

Recimo torej zdaj, da imamo pred seboj k skupin zvočnikov, pri čemer ima i-ta skupina glasnost α_i in skupno ceno spremembe β_i; in recimo še, da bi jih uredili po glasnosti, tako da bo $\alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_k$. Označimo s $f(x)$ ceno tega, da glasnost vseh teh zvočnikov postavimo na x:

$$f(x) = \textstyle\sum_{i=1}^k \beta_i |\alpha_i - x|.$$

Hitro se lahko prepričamo, da se smemo pri iskanju minimuma te funkcije omejiti na primere, ko je x enak kakšni od glasnosti α_i. Če je $x < \alpha_1$, prispevajo vse skupine člen $\beta_i(\alpha_i - x)$ in ker so cene β_i nenegativne, se skupna cena $f(x)$ zmanjša (ali vsaj ostane enaka), če x povečamo. Podobno tudi pri $x > \alpha_k$ vidimo, da se $f(x)$ zmanjša ali ostane enaka, če x zmanjšamo. Tam torej prav gotovo ne bomo našli minimuma funkcije f. In končno, če leži x med glasnostma dveh skupin, recimo na $\alpha_i < x < \alpha_{i+1}$, velja tam:

$$\begin{aligned} f(x) &= \textstyle\sum_{j=1}^i \beta_j (x - \alpha_j) + \sum_{j=i+1}^k \beta_j (\alpha_j - x) \\ &= \left(\textstyle\sum_{j=1}^i \beta_j - \sum_{j=i+1}^k \beta_j\right) x + \left(\textstyle\sum_{j=i+1}^k \beta_j \alpha_j - \sum_{j=1}^i \beta_j \alpha_j\right). \end{aligned}$$

Na tem območju je torej $f(x)$ linearna funkcija, kar pomeni, da se bo njena vrednost zmanjšala ali vsaj ostala enaka, če x zmanjšamo ali povečamo za 1 (odvisno od predznaka pri koeficientu, s katerim je pomnožen x). Minimuma funkcije f potemtakem gotovo ne bomo spregledali, če se omejimo na take x, ki so enaki kakšni od glasnosti α_i, saj lahko povsod drugod vrednost $f(x)$ zmanjšamo (ali je vsaj ne povečamo), če x premaknemo bliže k eni od α_i.

Glejmo torej $f(x)$ za primere, ko je $x = \alpha_i$. Kaj se zgodi, če se iz $x = \alpha_i$ premaknemo v $x = \alpha_{i+1}$? Pri skupinah $1, 2, \ldots, i$ moramo zdaj glasnost spremeniti za $\alpha_{i+1} - \alpha_i$ več kot prej, pri skupinah $i+1, \ldots, k$ pa za prav toliko manj kot prej. Označimo skupno vsoto zvočnikov levo od i z $L_i := \sum_{j=1}^{i-1} \beta_j$, tistih desno od i z $D_i := \sum_{j=i+1}^k \beta_j$, vseh skupaj pa z $B := L_i + \beta_i + D_i$. Sprememba cene pri premiku iz α_i v α_{i+1} je torej

$$\begin{aligned} f(\alpha_{i+1}) - f(\alpha_i) &= \left(\textstyle\sum_{j=1}^i \beta_j - \sum_{j=i+1}^k \beta_j\right)(\alpha_{i+1} - \alpha_i) \\ &= (L_i + \beta_i - D_i)(\alpha_{i+1} - \alpha_i) = (B - 2D_i)(\alpha_{i+1} - \alpha_i). \end{aligned}$$

Vidimo torej, da če je $B - 2D_i < 0$ oz. $D_i > B/2$, se splača iti iz $x = \alpha_i$ v $x = \alpha_{i+1}$, ker se s tem vrednost funkcije f zmanjša (ali pa vsaj ne poveča). Podobno se lahko tudi prepričamo, da če je $L_i > B/2$, se splača iti iz $x = \alpha_i$ v $x = \alpha_{i-1}$. Minimum funkcije f bomo torej našli tam, kjer ne velja nič od tega dvojega; torej tam, kjer je

$L_i \le B/2$ in tudi $D_i \le B/2$; slednji pogoj lahko zapišemo tudi kot $B - L_i - \beta_i \le B/2$, torej $L_i + \beta_i \ge B/2$; oba pogoja lahko torej združimo v

$$L_i \le B/2 \le L_i + \beta_i \text{ oz. } L_i \le B/2 \le L_{i+1}. \qquad (\star)$$

Naloga pravi, da če je minimum dosežen pri več različnih glasnostih, moramo uporabiti najmanjšo od njih; torej iščemo najmanjši i, pri katerem je izpolnjen gornji pogoj. Če je pri nekem i pogoj izpolnjen, vendar s strogo enakostjo na levi strani (torej $L_i = B/2 \le L_{i+1}$), bo izpolnjen tudi pri $i - 1$ in bo treba najnižjo glasnost iskati tam — pogoj lahko torej spremenimo v $L_i < B/2 \le L_{i+1}$, torej iščemo največji tak L_i, ki je še manjši od $B/2$. (Tak L_i gotovo obstaja, kajti naloga zagotavlja, da so vsi $b_\bullet > 0$, zato je $B > 0$ in pogoj $L_i < B/2$ je gotovo izpolnjen vsaj pri $i = 1$, kjer je $L_1 = 0$.) Tako smo prišli do naslednjega postopka za iskanje minimuma funkcije f:

vzemi največji tak i, pri katerem je L_i še vedno $< B/2$;
funkcija $f(x)$ doseže svoj minimum pri $x = \alpha_i$;

Če si res pripravimo urejen seznam parov (α_i, β_i), kot smo rekli na začetku, lahko zdaj minimum funkcije $f(x)$ zelo preprosto poiščemo v $O(k)$ časa: seštejmo vse β_i, da dobimo B, nato pa pojdimo v zanki po naraščajočih i in povečujmo L_i ter preverjajmo pogoj $L_i < B/2 \le L_i + \beta_i$; ko je ta izpolnjen, vrnimo $x = \alpha_i$. Toda zaradi urejanja na začetku bo časovna zahtevnost tega postopka vendarle $O(k \log k)$. To sicer ni nič hudega, kajti kot bomo videli kasneje, bomo tako ali tako že za pripravo seznama parov (α_i, β_i) (pred urejanjem) porabili $O(k \log n)$ časa.

Kljub temu pa si kot zanimivost oglejmo še malo boljši postopek za iskanje minimuma funkcije f, ki seznama parov ne ureja in ima časovno zahtevnost $O(k)$. Ko hočemo poiskati največji i, pri katerem je $L_i < B/2$, bi lahko to počeli z bisekcijo, saj je zaporedje L_i-jev naraščajoče:

$\ell := 1$; $d := k$;
while $d > \ell$:
  (* *Tu velja:* $L_\ell < B/2 \le L_{d+1}$. *)
  $m := \lfloor (\ell + d)/2 \rfloor$;
  if $L_m < B/2$ **then** $\ell := m$ **else** $d := m$;
(* *Tu velja:* $L_\ell < B/2 \le L_{\ell+1}$. *)
return α_ℓ;

Vprašanje je le še, kako računati vrednosti L_m, ne da bi na začetku uredili vse pare (α_i, β_i) in potem izračunali zaporedja L_i kot kumulativnih vsot zaporedja β_i. Pri tem si lahko pomagamo z algoritmom quickselect (v C++ uporabimo na primer funkcijo `nth_element` iz standardne knjižnice), ki nam v $O(d - \ell)$ časa razdeli pare (α_i, β_i) za $i = \ell, \ldots, d$ na levo in desno polovico glede na α_i. Naš postopek z bisekcijo lahko zdaj podrobneje opišemo takole:

funkcija NovaGlasnost:
  vhod: $S[1..k]$ = tabela vseh k parov (α_i, β_i) v poljubnem vrstnem redu;

$\ell := 1$; $d := k$; $L := 0$; $B := \beta_1 + \ldots + \beta_k$;
while $d > \ell$:

(* *Tu velja:* $L_\ell < B/2 \le L_{d+1}$. *Spremenljivka* L *ima vrednost* L_ℓ.
V tabeli S *so na indeksih od* ℓ *do* d *prav tisti pari, ki bi bili tam tudi v primeru, če bi bila tabela urejena po* α, *le da so mogoče premešani.* *)
$m := \lfloor(\ell + d)/2\rfloor$;
s quickselectom prerazporedi pare $S[\ell..d]$ tako, da bodo imeli pari na mestih $S[\ell..m-1]$ manjšo ali enako vrednost α kot tisti na mestih $S[m..d]$;
$L' := L +$ vsota β po vseh parih iz $S[\ell..m-1]$; (†)
(* Zdaj ima L' vrednost L_m. *)
if $L' < B/2$ **then** $\ell := m$; $L' := L$ **else** $d := m$;
(* *Tu po koncu zanke velja:* $L_\ell < B/2 \le L_{\ell+1}$ *in v* $S[\ell]$ *je prav tisti par, ki bi bil tam tudi, če bi bila tabela urejena po* α. *)
vrni $S[\ell].\alpha$; (* *to je* α_ℓ *)

Časovna zahtevnost vrstice (†) je $O(d-\ell)$ in ker se ta interval v vsaki iteraciji glavne zanke razpolovi, se tega po vseh iteracijah skupaj nabere za $O(k)$; toliko pa stane tudi izračun B (kot vsote vseh β_i) v prvi vrstici.

Doslej smo govorili le o dogodkih druge vrste; razmislimo zdaj še o tem, kako sploh predstaviti glasnosti zvočnikov in kako te podatke popravljati pri dogodkih prve vrste. Ker se ob vsakem dogodku spremeni veliko vrednosti a_i, si ne moremo privoščiti, da bi jih hranili eksplicitno, saj bi nam njihovo popravljanje vzelo preveč časa. Opazimo lahko, da se manj kot glasnosti a_i spreminjajo razlike med njimi, torej vrednosti $r_i := a_i - a_{i-1}$ (mislimo si še $a_0 = 0$). Kaj obe vrsti dogodkov pomenita za takšne razlike? Pri dogodku prve vrste, ko se $a_\ell, \ldots, a_d$ povečajo za x, se spremenita le dve razliki: r_ℓ se poveča za x, razlika r_{d+1} pa se zmanjša za x. Pri dogodku druge vrste, ko $a_\ell, \ldots, a_d$ dobijo vsi enako vrednost x, pa padejo razlike $r_{\ell+1}, \ldots, r_d$ na 0, razlika r_ℓ je po novem enaka $x - a_{\ell-1}$, razlika r_{d+1} pa je po novem enaka $a_{d+1} - x$.

Če torej namesto glasnosti a_i hranimo razlike r_i, bomo imeli manj dela pri spremembah glasnosti. Iz razlik r_i se dá potem izraziti a_i po formuli $a_i = r_1 + r_2 + \ldots + r_i$; da pa bomo lahko to računali dovolj hitro, je koristno vrednosti r_i organizirati v Fenwickovo drevo. Iz podatkov v njem lahko kadarkoli v $O(\log n)$ časa izračunamo posamezno vrednost r_i ali a_i, pa tudi spremenimo vrednost posamezne r_i.

Videli smo že, da bodo zaradi dogodkov druge vrste pogosto nastale skupine po več zaporednih zvočnikov z enako glasnostjo a_i; za naše razlike r_i pa to pomeni, da bo nekaj zaporednih r_i enakih 0. Nova skupina se začne pri vsakem neničelnem r_i. Videli smo tudi, da naš postopek za obravnavo dogodka druge vrste pričakuje kot vhod seznam skupin na območju, ki ga dogodek zadeva; znati moramo torej poceni našteti vse take skupine, torej vse neničelne r_i na danem območju. V ta namen je koristno npr. v rdeče-črnem drevesu hraniti tiste i, za katere je r_i trenutno neničeln (v C++ lahko uporabimo razred `set` iz standardne knjižnice); tega ni težko vzdrževati, ko se vrednosti r_i spreminjajo.

Zapišimo zdaj psevdokodo za obravnavo dogodka prve vrste. Rdeče-črno drevo bomo označili s T, Fenwickovo drevo pa s F.

podprogram Spremeni(i, x):
if $x = 0$ **then return**;
če je i v T, ga od tam pobriši;

popravi F tako, da se r_i-ju prišteje x;
izračunaj iz F novo vrednost r_i;
če je nova vrednost r_i enaka 0, dodaj i v T;

podprogram DogodekPrveVrste(ℓ, d, x):
Spremeni(ℓ, x); **if** $d < n$ **then** Spremeni($d + 1$, $-x$);

Pri dogodku druge vrste moramo najprej s pomočjo drevesa T našteti vse neničelne r_i na obravnavanem območju; nato lahko izračunamo novo glasnost, uporabljene r_i-je pa postavimo na 0, ker bo zdaj glasnost po celem obravnavanem območju enaka. Nazadnje moramo še popraviti r_ℓ in r_{d+1}. Ko pripravljamo pare (α, β) za skupine več zaporednih zvočnikov z enako glasnostjo, moramo znati hitro izračunati vsoto cen b_i za vse zvočnike v skupini; to je najlaže narediti tako, da si vnaprej izračunamo kumulativne vsote $B_1 = 0$, $B_{i+1} = B_i + b_i$, potem pa lahko vsoto več zaporednih cen $b_i + \ldots + b_{j-1}$ izračunamo kot $B_j - B_i$.

podprogram DogodekDrugeVrste(ℓ, d):
izračunaj iz F sedanjo vrednost a_ℓ in si jo zapomni v A;
S := prazen seznam; $i := \ell$; $a_\ell^{\text{stara}} := A$;
while true:
j := najmanjši tak element drevesa T, ki je $> i$;
if takega ni ali pa je večji od d **then** $j := d + 1$;
(* *Zvočniki od i do $j - 1$ imajo glasnost A.* *)
dodaj v S par $(A, B_j - B_i)$;
if $j > d$ **then break**;
$i := j$; izračunaj iz F vrednost r_i in si jo zapomni v R;
$A := A + R$; (* *Zdaj dobi A vrednost a_j.* *)
Spremeni(i, $-R$); (* *S tem postane $r_i = 0$ in i izgine iz T.* *)
$a_d^{\text{stara}} := A$;
α := NovaGlasnost(S);
Spremeni(ℓ, $\alpha - a_\ell^{\text{stara}}$); **if** $d < n$ **then** Spremeni($d + 1$, $a_d^{\text{stara}} - \alpha$);
return α;

Kakšna je časovna zahtevnost te rešitve? Posamezna operacija na F ali T vzame po $O(\log n)$ časa; tudi obravnava dogodka prve vrste zato vzame $O(\log n)$ časa; pri dogodku druge vrste prevladuje zanka za pripravo seznama S, ki porabi $O(k \log n)$ časa, če je v seznamu na koncu k elementov.

V najslabšem primeru je seveda k lahko velik $O(n)$, torej lahko za dogodek druge vrste porabimo $O(n \log n)$ časa. Na srečo pa se to ne more zgoditi posebej pogosto: neničelne r_i, ki jih gornji postopek prebere in uporabi pri izračunu, potem tudi postavi na 0. Dolgoročno je torej skupni čas obdelave več dogodkov druge vrste omejen s tem, koliko neničelnih r_i uspemo na novo pridelati, število le-teh pa je omejeno: vsak dogodek (prvega ali drugega tipa) spremeni največ dva r_i tako, da imata možnost postati neničelna. Recimo torej, da smo obdelali zaporedje več dogodkov, od tega q_1 dogodkov prve vrste in q_2 druge vrste; skupaj je torej nastalo največ $O(q_1 + q_2)$ neničelnih razlik, zato je cena vseh dogodkov druge vrste skupaj tudi največ $O((q_1 + q_2) \log n)$. Poleg tega je tudi cena vseh dogodkov prve vrste skupaj le $O(q_1 \log n)$. Če prištejemo še $O(n)$ časa za inicializacijo Fenwickovega

drevesa, lahko zaključimo, da je časovna zahtevnost naše rešitve $O(n + q \log n)$, kjer je q skupno število obojih dogodkov.

E. Ribolov

Ker moramo na poizvedbe odgovarjati sproti, se bomo vedno ukvarjali le z eno poizvedbo naenkrat, zato indeksa j ne bomo pisali; poizvedbo torej opisujejo števila b, ℓ in d. Razmislimo za začetek, kako bi rešili lažjo različico naloge, pri kateri nam ni treba razmišljati o y-koordinatah — recimo, da imajo vse poizvedbe $b = h$, tako da se mora naš pravokotnik vedno raztezati po celi višini mreže.

Naj bo zdaj v_x skupna vrednost vseh celic v stolpcu x; če se odločimo uporabiti pravokotnik od L do D, bo skupna vrednost celic v njem potem enaka $v_L + v_{L+1} + \ldots + v_D$. Poizvedba nas torej pravzaprav sprašuje, kakšna je največja možna vsota te oblike pri omejitvi $\ell \leq L \leq D \leq d$; recimo temu rezultatu $f(\ell, d)$.

To lahko ugotovimo z rekurzivnim razmislekom. Razdelimo poizvedovalno območje $[\ell, d]$ na levi del, recimo stolpce od ℓ do m (za neki m z območja $\ell \leq m < d$), in desni del, torej stolpce od $m + 1$ do d; najboljši pravokotnik za našo poizvedbo leži (1) bodisi celoti v levem delu (2) bodisi v celoti v desnem (3) bodisi malo v enem in malo v drugem. Prvi dve možnosti lahko obdelamo z rekurzivnima klicema za levi oz. desni del, pri čemer bomo vsakega od njiju spet razdelili na dva dela in tako naprej. Ta rekurzija se ustavi, ko ima pred seboj le še en sam stolpec ($\ell = d$) in je najboljša rešitev zanj $\max\{0, v_\ell\}$ (lahko ga vzamemo v celoti ali pa ničesar).

Tretja možnost, torej da se pravokotnik začne v levem in konča v desnem delu, pa pomeni, da bo vsota $v_L + \ldots + v_D$ sestavljena iz leve vsote $v_L + \ldots + v_m$ in desne vsote $v_{m+1} + \ldots + v_D$. Pri tem dogajanje v levem delu nič ne vpliva na desnega in obratno; vzeti moramo torej tak L (z območja $\ell \leq L \leq m$), pri katerem je $v_L + \ldots + v_m$ največja, in tak D (z območja $m + 1 \leq D \leq d$), pri katerem je $v_{m+1} + \ldots + v_D$ največja. V levem delu nas torej zanima največja *sufiksna* (priponska) vsota, se pravi vsota v_x za zadnjih nekaj stolpcev v njem; recimo ji $s(\ell, m)$. V desnem delu pa nas zanima največja *prefiksna* (predponska) vsota, torej vsota v_x za prvih nekaj stolpcev v njem; recimo ji $p(m + 1, d)$. Naš dosedanji razmislek lahko povzamemo s formulo:

$$f(\ell, d) = \max\{f(\ell, m), f(m + 1, d), s(\ell, m) + p(m + 1, d)\}$$

z robnim primerom $f(x, x) = s(x, x) = p(x, x) = \max\{0, v_x\}$, kajti ko imamo samo en stolpec, imajo ribiči na izbiro le, da ga bodisi vzamejo v celoti bodisi ostanejo s prazno mrežo.

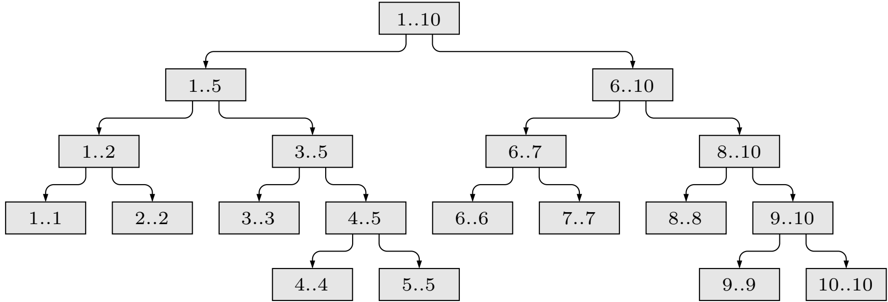
Tudi največje prefiksne in sufiksne vsote lahko računamo z rekurzivnim razmislekom. Največja sufiksna vsota območja $[\ell, d]$ se bodisi začne v desnem delu (v tem primeru je najbolje vzeti kar največjo sufiksno vsoto desnega dela) bodisi v levem (v tem primeru pa je najbolje vzeti največjo sufiksno vsoto levega dela, nato pa ji moramo v vsakem primeru prešteti še vsoto *celotnega* desnega dela). Za prefiksne vsote je razmislek podoben. Vidimo torej, da je koristno vpeljati še vsoto po vseh stolpcih nekega območja, recimo ji $v(\ell, d) = v_\ell + v_{\ell+1} + \ldots + v_d$. Tako smo dobili

zveze:

$$s(\ell, d) = \max\{s(m+1, d), s(\ell, m) + v(m+1, d)\}$$
$$p(\ell, d) = \max\{p(\ell, m), v(\ell, m) + p(m+1, d)\}$$
$$v(\ell, d) = v(\ell, m) + v(m+1, d).$$

Ker bomo morali omenjene funkcije f, s, p in v običajno računati vse štiri skupaj, jih bo občasno prikladno združiti v eno; vpeljimo v ta namen oznako $\mathbf{r}(\ell, d) = (f(\ell, d), s(\ell, d), p(\ell, d), v(\ell, d))$.

Doslej nismo rekli še ničesar o tem, kako izbrati m, torej kje razdeliti neko območje na levi in desni del. Ker bomo morali odgovoriti na veliko poizvedb, je smiselno uporabljati taka pod-območja, ki bodo prišla prav pri več poizvedbah, tako da nam vrednosti f, p, s in v zanje ne bo treba računati po večkrat. Začnimo na primer z območjem $[1, w]$, ki pokriva celotno mrežo; razdelimo ga na dva približno enako široka dela, nato vsakega od njiju spet na dva približno enako široka dela in tako rekurzivno naprej; nastane nam nekakšno drevo intervalov (v listih, kjer se rekurzija konča, imamo intervale, ki pokrivajo en sam stolpec). Naslednja slika kaže primer za $w = 10$:



Za vsako vozlišče u hranimo krajišči intervala, ki ga pokriva — recimo $[\ell_u, d_u]$ — ter vrednost $\mathbf{r}[u] := \mathbf{r}(\ell_u, d_u)$. Vsako vozlišče ima tudi kazalca (ali indeksa) na svoja dva otroka; recimo jima *levi*$[u]$ in *desni*$[u]$.

Ko pride potem poizvedba $[\ell, d]$, začnemo v korenu drevesa in spet razmišljamo rekurzivno. Če se trenutno vozlišče razdeli na levega in desnega otroka pri x-koordinati m, lahko tam razdelimo tudi naše poizvedovalno območje; z rekurzivnima klicema poiščemo rezultat za vsak del območja posebej, nato pa oba rezultata združimo s prav takim razmislekom, kot smo ga videli zgoraj pri funkcijah f, s, p in v. Pri tem pa, če poizvedovalno območje pokriva celotno trenutno vozlišče, lahko uporabimo rezultate, ki jih imamo že shranjene v tem vozlišču. Zapišimo ta postopek s psevdokodo:

funkcija POIZVEDBA(vozlišče u, interval $[\ell, d]$):
  $\ell' = \max\{\ell_u, \ell\}$; $d' = \min\{d_u, d\}$;
  (* $[\ell', d']$ *je tisti del poizvedovalnega intervala, ki ga pokriva vozlišče* u*;*
    naloga naše funkcije je izračunati $\mathbf{r}(\ell', d')$*.* *)
  (* *Mogoče se poizvedovalni interval sploh ne prekriva z našim vozliščem.* *)
  if $d' < \ell'$ **then return** (0, 0, 0, 0);

```
(* Mogoče poizvedovalni interval pokriva naše vozlišče v celoti. *)
if ℓ' = ℓ_u and d_u = d' then return r[u];
(* Tu že vemo, da je d_u > ℓ_u, torej ima u dva otroka.
   Razdelimo tudi poizvedbo na levi in desni del. *)
r_1 := Poizvedba(levi[u], [ℓ, d]);
r_2 := Poizvedba(desni[u], [ℓ, d]);
return Združi(r_1, r_2);
```

Koren drevesa pokriva celoten interval $[1, w]$ in njegov presek z našim poizvedovalnim območjem $[\ell, d]$ je kar to območje v celoti; če torej pokličemo Poizvedba(*koren*, $[\ell, d]$), bo takrat $\ell' = \ell$ in $d' = d$, tako da bomo kot rezultat med drugim dobili $f(\ell, d)$, prav to pa je odgovor na našo poizvedbo.

Za združevanje rezultatov iz poizvedbe po levem in po desnem otroku uporabimo naslednji podprogram, ki ni nič drugega kot implementacija prej omenjenih rekurzivnih zvez za funkcije f, s, p in v. Prav nam bo prišel tudi kasneje pri gradnji dreves.

```
funkcija Združi(r_1, r_2):
  (* Kot vhod pričakujemo četverici r_1 = r(ℓ, m) in r_2 = r(m + 1, d) za neka
     števila ℓ ≤ m < d. Funkcija vrne r(ℓ, d). *)
  naj bo (f_1, s_1, p_1, v_1) = r_1 in (f_2, s_2, p_2, v_2) = r_2;
  f_3 := max{f_1, f_2, s_1 + p_2}; s_3 := max{s_2, s_1 + v_2}; p_3 := max{p_1, v_1 + p_2};
  v_3 := v_1 + v_2; return (f_3, s_3, p_3, v_3);
```

Kakšna je časovna zahtevnost postopka Poizvedba? Pri vozliščih, ki jih naše poizvedovalno območje bodisi pokriva v celoti bodisi ima z njimi neprazen presek, porabimo le konstantno mnogo časa in lahko čas, porabljen zanje, štejemo k nadrejenemu rekurzivnemu klicu. Vprašanje je torej, koliko je takih vozlišč, ki jih naše poizvedovalno območje prekriva le delno, ne pa v celoti. To se lahko zgodi na tri načine: (1) poizvedovalno območje pokriva levi konec našega vozlišča, ne pa desnega ($\ell \leq \ell_u$ in $d < d_u$); (2) ali pa pokriva desni konec in ne levega ($\ell_u < \ell$ in $d_u \leq d$); (3) ali pa ne pokriva nobenega ($\ell_u < \ell$ in $d < d_u$). Vozlišče tipa 1 ali 2 ima lahko največ enega otroka enakega tipa (drugi otrok pa je bodisi pokrit v celoti ali pa sploh nič); vozlišče tipa 3 pa ima lahko največ enega otroka tipa 3 (drugi je v tem primeru popolnoma nepokrit) ali pa ima največ enega otroka tipa 2 (to je lahko njegov levi otrok) in največ enega tipa 1 (to bo desni otrok). Vidimo torej, da če je bil koren tipa 1 ali 2, bomo tudi na vsakem od nižjih nivojev imeli po največ eno vozlišče istega tipa (in sploh nobenega vozlišča tipa 3); če pa je bil koren tipa 3, bomo še na nekaj naslednjih nivojih morda imeli po eno vozlišče tipa 3, od tam naprej pa po največ eno vozlišče tipa 1 in največ eno tipa 2. Na vsakem nivoju drevesa torej obiščemo po največ dve vozlišči; in ker vsako vozlišče razcepimo približno na polovico, je drevo globoko približno $\log_2 w$ nivojev, zato je časovna zahtevnost poizvedbe $O(\log w)$.

Doslej smo razmišljali le o x-koordinatah, torej kot da je pri vsaki poizvedbi $b = h$. V resnici bomo morali znati odgovarjati na poizvedbe za poljubne b od 1 do h. Lepo bi bilo, če bi lahko imeli za vsak možen b takšno drevo, kot smo ga zgradili zgoraj za $b = h$; pri drevesu T_b bi upoštevali vse tiste celice, ki imajo y-koordinato na območju $1 \leq y \leq b$. Vsako tako drevo zasede $O(w)$ prostora, zato si ne moremo

privoščiti $O(h)$ dreves; na srečo pa so si mnoga med njimi zelo podobna: T_b lahko dobimo iz T_{b-1} tako, da vanj dodamo tiste celice, ki imajo y-koordinato natanko b. Pri dodajanju celice (x, b) v drevo pa se spremenijo le tista vozlišča u, ki pokrivajo x-koordinato te celice, torej tista, za katera velja $\ell_u \leq x \leq d_u$; to je natanko eno vozlišče na vsakem nivoju drevesa, skupno $O(\log w)$ vozlišč.

Tako bi se dalo poceni predelati T_{b-1} v T_b, toda ker bomo kasneje pri odgovarjanju na poizvedbe potrebovali obe drevesi, pri dodajanju v resnici ne bomo spreminjali vozlišč drevesa T_{b-1}, pač pa bomo tam, kjer pride do sprememb, ustvarjali nova vozlišča; tam pa, kjer je novo drevo enako staremu, bo še vedno uporabljalo vozlišča starega drevesa. Tako sčasoma nastane struktura, v kateri se skriva več med seboj prekrivajočih se dreves, namreč T_b za vse b-je od 1 do h; vsako ima svoj koren, spodaj pa si mnoga vozlišča deli po več dreves. Takšnim podatkovnim strukturam, ki ob spreminjanju ohranijo tudi svojo staro verzijo, pravimo „obstojne“ (*persistent*). Postopek dodajanja je torej takšen:

funkcija DODAJ(vozlišče u, koordinata x, vrednost v):
  (* *Predpostavimo, da u pokriva x, torej da je $\ell_u \leq x \leq d_u$.* *)
  $\tilde{u}$:= novo vozlišče z $\ell_{\tilde{u}} = \ell_u$ in $d_{\tilde{u}} = d_u$;
  if $\ell_u = d_u$: (* *torej če je u list* *)
    $(f_u, p_u, s_u, v_u) := \mathbf{r}[u]$;
    $v_{\tilde{u}} := v_u + v$; $p_{\tilde{u}} := \max\{0, v_{\tilde{u}}\}$;
    $s_{\tilde{u}} := p_{\tilde{u}}$; $f_{\tilde{u}} := p_{\tilde{u}}$;
    $\mathbf{r}[\tilde{u}] := (f_{\tilde{u}}, s_{\tilde{u}}, p_{\tilde{u}}, v_{\tilde{u}})$;
  else:
    z := *levi*$[u]$; w := *desni*$[u]$; $m := d_z$;
    if $x \leq m$ **then** z := DODAJ(z, x_i, v) **else** w := DODAJ(w, x_i, v);
    levi$[\tilde{u}] := z$; *desni*$[\tilde{u}] := w$;
    $\mathbf{r}[\tilde{u}]$:= ZDRUŽI$(\mathbf{r}[z], \mathbf{r}[w])$;
  return $\tilde{u}$;

Da dobimo drevesa $T_1, T_2, \ldots, T_h$, bomo začeli s praznim drevesom intevalov — takim, ki ima pravo strukturo vozlišč, vendar imajo vsa vozlišča $\mathbf{r}[u] = (0, 0, 0, 0)$ — in potem dodajali vanj celice po naraščajočem y. V spodnji psevdokodi si predstavljamo T kot tabelo, ki hrani korene vseh tako dobljenih dreves:

  for $y := 1$ **to** h **do** $L[y]$:= prazen seznam;
  za vsako celico (x_i, y_i) z vrednostjo $v_i \neq 0$: dodaj (x_i, v_i) v seznam $L[y_i]$;
  ustvari prazno drevo intervalov (s pravo strukturo, a ničelnimi vrednostmi);
  naj bo u koren tega drevesa;
  for $b := 1$ **to** h:
    za vsako (x_i, v_i) s seznama $L[b]$: u := DODAJ(u, x_i, v_i);
    $T[b] := u$;

Prazno drevo ima $O(w)$ vozlišč, nato pa dodamo vanj $O(k)$ novih celic in pri vsakem takem dodajanju nastane $O(\log w)$ novih vozlišč; časovna zahtevnost tega postopka je torej $O(w + k \log w)$, prostorska pa $O(w + h + k \log w))$.

Za naše namene je ta rešitev dovolj dobra, dalo pa bi se jo še malo izboljšati. Na primer, če sta w in h veliko večja od k, torej če je naša mreža zelo redka, bi bilo dobro koordinate najprej „skompresirati“ — obdržimo le tiste vrstice in stolpce

mreže, ki vsebujejo kakšno neničelno celico; x-koordinate celic tako preslikamo z območja $\{1, \ldots, w\}$ v $\{1, \ldots, k\}$, podobno pa tudi y-koordinate; ustrezno pa potem preslikamo tudi koordinate poizvedovalnih območij.

Po drugi strani, če je k velik v primerjavi z w in h, torej če je naša mreža gosta, je potratno, da funkcija DODAJ vedno ustvarja nova vozlišča; če imamo v isti vrstici b več celic in se pri dodajanju teh celic v drevo večkrat spremeni isto vozlišče, ni treba vsakič delati nove kopije tega vozlišča, saj nas bo zanimalo stanje drevesa šele po vseh teh dodajanjih, ne pa med njimi. Takrat bi bilo torej koristno v vsakem vozlišču hraniti podatek o tem, pri katerem b smo tisto vozlišče ustvarili, in če smo še vedno pri istem b, smemo to vozlišče spremeniti in zanj ni treba delati kopije. Tako se poraba prostora za drevesa zmanjša z $O(w + k \log w)$ na $O(w + \min\{hw, k \log w\})$, ker se v najslabšem primeru pri vsakem b spremenijo vsa vozlišča drevesa, teh pa je $O(w)$.

F. Črke

Ker so testni primeri majhni, lahko nalogo rešimo preprosto tako, da odsimuliramo dogajanje. Vsebino matrike lahko hranimo v dvodimenzionalni tabeli, pri čemer element $a[i, j]$ predstavlja j-ti znak v i-ti vrstici. Če na primer v neki fazi črke polzijo navzdol, lahko stanje matrike na koncu te faze izračunamo tako, da gremo po vsakem stolpcu od spodaj navzgor in premikamo črke na dno stolpca:

```
for j := 1 to m:
  k := n; (* vrstica, v katero bo šla naslednja črka *)
  for i := n downto 1:
    if a[i, j] = '.' then continue;
    a[i, k] := a[i, j]; k := k − 1;
  while k > 0:
    a[i, k] := '.'; k := k − 1;
```

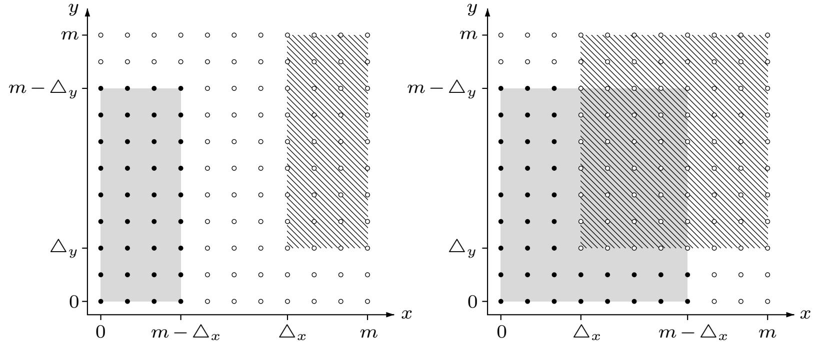
Polzenje gor (namesto dol) lahko simuliramo na enak način, le da moramo v gornjem postopku, kjerkoli dostopa do elementa $a[u, v]$, zdaj namesto tega dostopati do $a[n - 1 - u, v]$. Podobno lahko polzenje desno ali levo simuliramo tako, da v mislih zamenjamo n in m, nato pa, kjerkoli postopek za polzenje dol ali gor dostopa do $a[u, v]$, zdaj namesto tega dostopamo do $a[v, u]$.

Možna drobna izboljšava je še tale: če v več zaporednih fazah polzijo črke le levo in/ali desno, je končni učinek enak, kot če bi imeli le zadnjo od teh faz, prejšnje pa lahko preskočimo; podobno tudi, če v več zaporednih fazah polzijo črke le gor in/ali dol.

G. Premice na mreži

Imamo n navpičnih in n vodoravnih premic, ostale pa so poševne. Naklon poševne premice lahko opišemo s parom celih števil $(\triangle_x, \triangle_y)$, pri čemer je $\triangle_x > 0$ in $\triangle_y \neq 0$, ki nam povesta, da se y-koordinata spremeni za $\triangle_y$, ko se x-koordinata poveča za $\triangle_x$. Pri tem imamo še omejitev, da si morata biti $\triangle_x$ in $\triangle_y$ tuja, saj bi drugače po več parov opisovalo enak naklon in bi nekatere premice šteli po večkrat.

Premice z $\triangle_y > 0$ imenujmo *rastoče*, tiste z $\triangle_y < 0$ pa *padajoče*. Vidimo lahko, da če ravnino prezrcalimo prek vodoravne simetrale našega kvadrata (premice



Dva primera določanja, katere točke (x, y) so lahko prve, ki jih v naši karirasti mreži zadene premica z naklonom $(\triangle_x, \triangle_y)$. Sivi pravokotnik je območje, ki ustreza pogoju (1), šrafirani pravokotnik pa območje, ki ustreza pogoju (2). Črne pike predstavljajo tiste pare (x, y), ki ustrezajo prvemu pogoju, ne pa tudi drugemu. Na levi sliki je primer za $m - \triangle_x < \triangle_x$, na desni pa za $m - \triangle_x \geq \triangle_x$.

$y = (n-1)/2$), se vsaka rastoča premica preslika v neko padajočo in obratno. Obojih je torej enako število, zato bomo v nadaljevanju šteli le rastoče in na koncu njihovo število podvojili.

Rastoče premice lahko razdelimo na *položne* ($\triangle_y \leq \triangle_x$) in *strme* ($\triangle_y \geq \triangle_x$). Vidimo lahko, da če ravnino prezrcalimo prek diagonalne simetrale našega kvadrata (premice $y = x$), se vsaka strma premica preslika v neko položno in obratno; tudi tu je torej obojih enako število, zato bomo v nadaljevanju šteli le položne in na koncu njihovo število podvojili. Pazimo le na to, da smo diagonalne premice ($\triangle_y = \triangle_x = 1$) šteli pri obojih, zato jih bomo morali po podvajanju enkrat spet odšteti (diagonalnih premic, ki gredo skozi vsaj dve točki našega kvadrata, je $2n - 3$).

V nadaljevanju bomo vpeljali $m = n - 1$, da bo manj pisanja. Možni nakloni so zdaj torej $(\triangle_x, \triangle_y)$ za $1 \leq \triangle_y \leq \triangle_x \leq m$, pri čemer si morata biti $\triangle_x$ in $\triangle_y$ tuja. Za posamezni tak naklon $(\triangle_x, \triangle_y)$ nas zdaj zanima, koliko premic s tem naklonom pokrije vsaj dve točki našega kvadrata. Če se po taki premici premikamo v smeri gor in desno, naj bo (x, y) prva točka kvadrata, na katero naletimo; naslednja bo potem $(x + \triangle_x, y + \triangle_y)$. Veljati mora torej

$$0 \leq x \leq m - \triangle_x \quad \text{in} \quad 0 \leq y \leq m - \triangle_y, \tag{1}$$

sicer bo ta naslednja točka že zunaj našega kvadrata. Od tako dobljenih parov (x, y) pa moramo odšteti tiste, pri katerih (x, y) ni prva točka, na katero naletimo, torej tiste, pri katerih velja

$$x \geq \triangle_x \quad \text{in} \quad y \geq \triangle_y. \tag{2}$$

Parov (x, y), ki ustrezajo pogoju (1), je torej $(m - \triangle_x + 1) \cdot (m - \triangle_y + 1)$. Koliko moramo zdaj od tega odšteti zaradi pogoja (2)?

Če je $m - \triangle_x < \triangle_x$ (torej če je $\triangle_x > m/2$ — levi primer na gornji sliki), bo iz pogoja $x \leq m - \triangle_x$ sledilo tudi $x < \triangle_x$, torej pari, ki ustrezajo pogoju (1), gotovo ne bodo izpolnili pogoja (2) in tam ne bo treba odštevati ničesar.

Če pa je $m - \triangle_x \geq \triangle_x$ (torej če je $\triangle_x \leq m/2$ — desni primer na sliki), potem iz pogoja $\triangle_y \leq \triangle_x$ sledi, da je tudi $\triangle_y \leq m/2$, torej bo nekaj parov (x, y) pri tem paru $(\triangle_x, \triangle_y)$ gotovo res lahko imelo $x \geq \triangle_x$ in $y \geq \triangle_y$; to so tisti, ki imajo

$$\triangle_x \leq x \leq m - \triangle_x \quad \text{in} \quad \triangle_y \leq y \leq m - \triangle_y,$$

torej jih je $(m - 2\triangle_x + 1) \cdot (m - 2\triangle_y + 1)$.

Skupno število položnih premic je torej $Q = Q_1 - Q_2$, pri čemer je Q_1 število vseh, ki ustrezajo pogoju (1), Q_2 pa je število tistih, ki jih moramo odšteti zaradi pogoja (2):

$$Q_1 = \sum_{\triangle_x=1}^{m} \sum_{\triangle_y=1}^{\triangle_x} [\![\triangle_y \text{ je tuj } \triangle_x]\!](m - \triangle_x + 1)(m - \triangle_y + 1)$$

$$Q_2 = \sum_{\triangle_x=1}^{\lfloor m/2 \rfloor} \sum_{\triangle_y=1}^{\triangle_x} [\![\triangle_y \text{ je tuj } \triangle_x]\!](m - 2\triangle_x + 1)(m - 2\triangle_y + 1).$$

Ta dva izraza sta si zelo podobna in si ju lahko predstavljamo kot posebna primera splošnejšega izraza: $Q_1 = q(m, 1)$ in $Q_2 = q(\lfloor m/2 \rfloor, 2)$, pri čemer je

$$\begin{aligned} q(a, b) &= \sum_{\triangle_x=1}^{a} \sum_{\triangle_y=1}^{\triangle_x} [\![\triangle_y \text{ je tuj } \triangle_x]\!](n - b\triangle_x)(n - b\triangle_y) \\ &= \sum_{\triangle_x=1}^{a} \sum_{\triangle_y=1}^{\triangle_x} [\![\triangle_y \text{ je tuj } \triangle_x]\!](n^2 - nb\triangle_x - nb\triangle_y + b^2\triangle_x\triangle_y) \\ &= \sum_{\triangle_x=1}^{a} \left[n^2\phi(\triangle_x) - nb\triangle_x\phi(\triangle_x) - nbF(\triangle_x) + b^2\triangle_x F(\triangle_x)\right]. \end{aligned}$$

Pri tem je $\phi(u)$ (Eulerjeva funkcija) število tistih števil izmed $1, 2, \ldots, u$, ki so tuja številu u; s $F(u)$ pa smo označili vsoto teh števil. Pri $u > 1$ velja $F(u) = (u/2)\phi(u)$,[15] zato pri $\triangle_x > 1$ dobimo $\triangle_x\,\phi(\triangle_x) + F(\triangle_x) = 3F(\triangle_x)$, pri $\triangle_x = 1$ pa je $\triangle_x\,\phi(\triangle_x) + F(\triangle_x) = 2 = 3F(\triangle_x) - 1$. Če uporabimo to v zadnjem izrazu za $q(a, b)$, dobimo:

$$q(a, b) = \sum_{\triangle_x=1}^{a} \left[n^2\phi(\triangle_x) - 3nbF(\triangle_x) + b^2\triangle_x F(\triangle_x)\right] \; + \; nb.$$

Pišimo še $G(u) = u \cdot F(u)$. Koristno si je vnaprej v tabelah pripraviti delne (kumulativne) vsote funkcij ϕ, F in G; naj bo torej $S_\phi(v) = \sum_{u=1}^{v} \phi(u)$ in podobno za $S_F(v)$ in $S_G(v)$. Potem dobimo

$$q(a, b) = n^2 S_\phi(a) - 3nbS_F(a) + b^2 S_G(a) + nb.$$

[15] O tem se lahko prepričamo takole: za $u = 2$ ročno preverimo, da formula drži (edino primerno tuje število je 1, zato je $\phi(2) = F(2) = 1$). Za $u > 2$ opazimo, da sta števili v in $u - v$ (za $v = 1, 2, \ldots, u - 1$) bodisi obe tuji u-ju ali pa nobeno; vsota, s katero je definirana $F(u)$, ima $\phi(u)$ seštevancev, ki jih lahko v mislih združimo v pare $\{v, u - v\}$; imamo torej $\phi(u)/2$ parov, vsak par pa ima vsoto u, torej $F(u) = u \cdot \phi(u)/2$.

Ko imamo funkcije S_ϕ, S_F in S_G pripravljene v tabelah, lahko v $O(1)$ časa izračunamo $q(a,b)$, zato pa tudi Q (število položnih premic) in iz tega $2Q-(2n-3)$ (število vseh poševnih premic; podvojili smo položne in odšteli diagonalne) in končno $4Q-2n+6$ (število vseh premic; poševne smo podvojili in dodali še n vodoravnih in n navpičnih).

Preden lahko računamo funkcije F, G, S_ϕ, S_F in S_G, moramo najprej izračunati ϕ; najbolje, da kar do $\phi(10^7)$, ker gredo lahko vrednosti n pri tej nalogi do 10^7. Kot je znano, je $\phi(u) = u\prod_p(1-1/p)$, pri čemer gre p po vseh u-jevih prafaktorjih.[16] Zato lahko ϕ računamo z Eratostenovim rešetom:

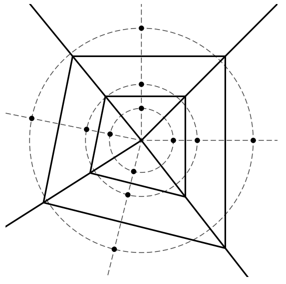
for $u := 1$ **to** n **do** $\phi[u] := u$;
for $u := 2$ **to** n **do if** $\phi[u] = u$:
  $v := u$; **while** $v \le n$:
    $\phi[v] := \phi[v] - \phi[v]/u$; $v := v + u$;

Na začetku torej inicializiramo $\phi[u]$ na u (za vse u); potem pa gremo po naraščajočih u in če pri nekem u vidimo, da je $\phi[u]$ še vedno enak u, to pomeni, da je u praštevilo (kajti če bi imel u kak prafaktor $p < u$, bi vrednost $\phi[u]$ že takrat, ko je glavna zanka dosegla tisti p, pomnožili z $1-1/p$, zato bi bila $\phi[u]$ zdaj manjša od u); to pa pomeni, da za vsak u-jev večkratnik v v formuli za $\phi(v)$, torej $\phi(v) = v\prod_p(1-1/p)$, nastopa tudi faktor $1-1/u$; zato je zdaj primeren trenutek, da vrednost $\phi[v]$ pomnožimo z $1-1/u$. Tako bomo sčasoma naredili z vsemi v-jevimi prafaktorji in bo na koncu v $\phi[v]$ res prava vrednost $\phi(v)$. Časovna zahtevnost tega postopka je $O(\sum_p n/p)$, pri čemer gre p po vseh praštevilih, manjših ali enakih n; izkaže se, da je to reda $O(n\log\log n)$.

H. Radar

Smerne točke $((f_x)_i, (f_y)_i)$ določajo F poltrakov, ki izhajajo iz koordinatnega izhodišča; oddaljenosti r_i pa določajo R krožnic s središčem v koordinatnem izhodišču (in polmeri r_i). Točke, ki jih radar pregleda, so ravno vsa presečišča teh F poltrakov s temi R krožnicami. Na sliki na str. 144 vidimo primer s štirimi poltraki in tremi krožnicami; narisani so s črtkanimi črtami, pregledane točke pa s črnimi pikami. Če bi se zdaj v mislih za vsako točko ravnine vprašali, katera izmed pregledanih točk ji je najbližja, bi nam ravnina razpadla na prav toliko območij, kolikor je pregledanih točk; meje med temi območji so na sliki narisane z debelimi črtami. Vidimo, da lahko te meje dobimo tako, da ravnino najprej razrežemo s F novimi poltraki, ki

[16] Do te formule lahko pridemo na primer z načelom vključitev in izključitev. Naj bo $P = \{p_1, \ldots, p_t\}$ množica vseh u-jevih prafaktorjev. Da dobimo $\phi(u)$, lahko začnemo s številom vseh števil od 1 do u in od tega odštevamo število tistih, ki niso tuja u-ju: tu je u/p_1 večkratnikov p_1, pa u/p_2 večkratnikov p_2 in tako naprej; toda tiste, ki so večkratniki dveh prafaktorjev hkrati, smo zdaj odšteli dvakrat in jih moramo enkrat prišteti nazaj: torej $u/(p_ip_j)$ za vse pare i,j; večkratnike treh prafaktorjev smo zdaj prišteli prevečkrat in jih moramo spet odšteti in tako naprej; dobimo torej $\phi(u) = \sum_{r=0}^t u(-1)^r \sum_{A\subseteq P:|A|=r}\prod_{p\in A}\frac{1}{p} = u\sum_{r=0}^t\sum_{A\subseteq P:|A|=r}\prod_{p\in A}(-\frac{1}{p}) = u\sum_{A\subseteq P}\prod_{p\in A}(-\frac{1}{p}) = u\prod_{p\in P}(1-\frac{1}{p})$. Zadnji enačaj je upravičen zato, ker imamo v $\prod_{p\in P}(1-\frac{1}{p})$ produkt t dvočlenih izrazov, ki ga lahko razvijemo v vsoto 2^t zmnožkov; vsaka $A \subseteq P$ predstavlja eno od možnih 2^t izbir tega, pri katerih izmed t dvočlenih izrazov vzamemo v zmnožek prvi člen (torej 1), pri katerih pa drugega (torej $-\frac{1}{p}$).



Primer radarja s tremi oddaljenostmi r_i (polmeri črtkanih krožnic) in štirimi smermi (črtkani poltraki), tako da radar pregleda 12 točk (črne pike). Za vsako od teh točk lahko določimo del ravnine, v katerem je ta točka bližja od ostalih pregledanih točk; meje med temi območji so narisane z debelimi črtami.

---

ležijo ravno na pol poti med dvema zaporednima izmed prvotnih (črtkanih) poltrakov; nastane F izsekov, po eden za vsakega od črtkanih poltrakov; nato pa vsak tak izsek razrežemo na R območij tako, da čez njegov črtkani poltrak potegnemo pravokotnice ravno na pol poti med vsakima dvema zaporednima pregledanima točkama tega poltraka.

Da bomo torej za dano poizvedovalno točko določili najbližjo pregledano točko, bomo morali najprej ugotoviti, kateri črtkani poltrak je tej poizvedovalni točki najbližji (tako bomo izvedeli, na katerem izseku leži ta poizvedovalna točka), nato pa jo bomo morali pravokotno projicirati na tisti poltrak in ugotoviti, katera izmed pregledanih točk na njem je najbližja poizvedovalni.

O tem se lahko prepričamo tudi z algebraičnim razmislekom. Recimo za začetek, da imamo en sam poltrak, ki ga določa točka $\mathbf{f} = (f_x, f_y)$, in da nas zanima, katera točka na njem je najbližja poizvedovalni točki $\mathbf{q} = (q_x, q_y)$. Točke na poltraku so oblike $\lambda\mathbf{f}$ (za $\lambda \geq 0$) in oddaljenost take točke od $\mathbf{q}$ lahko zapišemo kot funkcijo parametra λ:

$$g(\lambda) = ||\lambda\mathbf{f} - \mathbf{q}||^2 = (\lambda f_x - q_x)^2 + (\lambda f_y - q_y)^2.$$

Da poiščemo njen minimum, poglejmo, kje ima odvod 0:

$$\begin{aligned} g'(\lambda) &= 2(\lambda f_x - q_x)f_x + 2(\lambda f_y - q_y) \\ &= 2\lambda(f_x^2 + f_y^2) - 2(q_x f_x + q_y f_y) = 2\lambda||\mathbf{f}||^2 - 2\mathbf{q}^T\mathbf{f}, \end{aligned}$$

torej $g'(\lambda) = 0$ pri $\lambda = \mathbf{q}^T\mathbf{f}/||\mathbf{f}||^2$. Za to λ je $\lambda\mathbf{f}$ najbližja naši poizvedovalni točki (ni se tudi težko prepričati, da je $\lambda\mathbf{f}$ ravno pravokotna projekcija točke $\mathbf{q}$ na poltrak skozi $\mathbf{f}$). Če nam ta formula dá $\lambda < 0$, moramo vzeti $\lambda = 0$, saj imamo opravka s poltrakom in ne premico.

Naš radar seveda ne pregleda vseh točk poltraka, pač pa le tiste, ki so od koordinatnega izhodišča $\mathbf{0}$ oddaljene za eno od vrednosti r_i iz vhodnih podatkov; to so točke oblike $(r_i/||\mathbf{f}||)\mathbf{f}$. Med njimi je točki $\lambda\mathbf{f}$ najbližja tista, za katero je $r_i/||\mathbf{f}||$ najbližji številu λ. Koristno je urediti oddaljenosti naraščajoče, tako da bo $r_1 < r_2 < \ldots < r_R$, potem pa lahko v tem zaporedju z bisekcijo iščemo vrednost $\lambda||\mathbf{f}||$; če je enaka enemu od r_i, vzamemo njega, če pa $\lambda||\mathbf{f}||$ pade med r_i in r_{i+1}, moramo pač pogledati, kateri od njiju je bližji številu $\lambda||\mathbf{f}||$.

Doslej smo razmišljali o enem poltraku, v resnici pa jih imamo več. Na katerem izmed njih bomo našli $\mathbf{q}$-ju najbližjo točko? Smer poltraka skozi $\mathbf{f}$ lahko opišemo tudi s kotom α, za katerega ima poltrak potem smerni vektor $(\cos\alpha, \sin\alpha) = \mathbf{f}/||\mathbf{f}||$.

Recimo, da se osredotočimo na točke na oddaljenosti r od izhodišča; med njimi potem na poltraku v smeri α dobimo točko $r \cdot (\cos\alpha, \sin\alpha)$. Kakšna je razdalja med to točko in $\mathbf{q}$ v odvisnosti od smeri α?

$$\begin{aligned}
h(\alpha) &= ||r(\cos\alpha, \sin\alpha) - \mathbf{q}||^2 \\
&= (r\cos\alpha - q_x)^2 + (r\sin\alpha - q_y)^2 \\
&= r^2\cos^2\alpha - 2rq_x\cos\alpha + q_x^2 + r^2\sin^2\alpha - 2rq_y\sin\alpha + q_y^2 \\
&= r^2(\cos^2\alpha + \sin^2\alpha) - 2r(q_x\cos\alpha + q_y\sin\alpha) \\
&= r^2 - 2r(q_x\cos\alpha + q_y\sin\alpha).
\end{aligned}$$

Zapišimo točko $\mathbf{q}$ v polarni obliki: $\mathbf{q} = c \cdot (\cos\beta, \sin\beta)$ za $c = ||\mathbf{q}||$. Nesimo to v formulo za h:

$$\begin{aligned}
h(\alpha) &= r^2 - 2cr(\cos\alpha\cos\beta + \sin\alpha\sin\beta) \\
&= r^2 - 2cr\cos(\alpha - \beta).
\end{aligned}$$

Ker sta c in r konstantna in večja od 0, je vrednost $h(\alpha)$ tem manjša, čim večji je $\cos(\alpha-\beta)$, to pa je takrat, ko je $\alpha-\beta$ čim manjša. Najbližjo točko bomo torej dobili na tistem poltraku, čigar smer α je najbližja polarnemu kotu naše poizvedovalne točke — in ta zaključek je neodvisen od oddaljenosti r, pri kateri bomo najbližjo točko dejansko našli.

Tudi poltrake je torej koristno urediti po njihovi smeri α; dobimo na primer $\alpha_1 < \alpha_2 < \ldots < \alpha_F$. V tako dobljenem zaporedju potem z bisekcijo poiščimo vrednost β; če ta pade med α_i in α_{i+1}, preglejmo tistega izmed obeh poltrakov, ki je bližji β, lahko pa tudi enostavno pregledamo oba. Paziti moramo še na to, da so koti ciklični; če imamo $\beta < \alpha_1$, se lahko zgodi, da je najbližji poltrak v resnici α_F (in podobno, če je $\beta > \alpha_F$, je najbližji poltrak mogoče α_1). Kakorkoli že, ko se tako omejimo na enega ali dva poltraka, lahko potem na njem ali njiju poiščemo najbližjo točko z bisekcijo po oddaljenostih r_i, kot smo videli malo prej. Časovna zahtevnost našega postopka je torej $O(R\log R + F\log F)$ za urejanje oddaljenosti in poltrakov na začetku, nato pa $O(\log R + \log F)$ za vsako od N poizvedb (zaradi bisekcije po oddaljenostih in po poltrakih). Za izračun polarnih kotov α_i in β lahko uporabimo na primer funkcijo `atan2` iz standardne knjižnice jezikov C/C++.

Primere, ko je $\mathbf{q} = (0, 0)$ in poizvedovalni točki ne moremo določiti polarnega kota β, lahko obravnavamo posebej; taki točki so najbližje tiste na oddaljenosti r_1 (ne glede na smer poltraka).

Z nekaj truda bi se dalo $\mathbf{q}$-ju najbližjo točko najti tudi brez operacij s plavajočo vejico,[17] vendar napake pri računanju s plavajočo vejico pri tej nalogi niso tolikšne, da bi bilo to potrebno.

[17] Namesto da bi računali polarne kote, lahko za primerjanje smeri (pri urejanju poltrakov, pa tudi kasneje pri bisekciji na poltrakih) gledamo naklone f_y/f_x kot racionalna števila (in pazimo še na predznak pri f_x); pri iskanju najbližje točke na poltraku lahko $\lambda = \mathbf{q}^T\mathbf{f}/||\mathbf{f}||^2$ predstavimo kot racionalno število, saj so q_x, q_y, f_x, f_y cela števila; ker nimamo polarnih kotov, je težje reči, kateri od dveh poltrakov, med katerima leži naša točka, ji je najbližji, zato izračunajmo najbližjo točko na obeh in poglejmo, katera od teh je bližja $\mathbf{q}$. Nekoliko se zaplete le pri računanju razdalje med poizvedovalno točko $\mathbf{q}$ in točkami $(r_i/||\mathbf{f}||)\mathbf{f}$ na poltraku, kajti taka razdalja je lahko iracionalno število; je pa njen kvadrat oblike $u + \sqrt{v}$ za racionalna u in v, taka števila pa lahko z nekaj pazljivosti primerjamo po velikosti tudi brez računanja s plavajočo vejico.

I. Pokrajinski razvoj

Cestno omrežje lahko predstavimo z grafom in pretokom na njem: točke predstavljajo vasi, povezave med njimi ceste, pretok po povezavah pa trgovce. Naj bo $f(u,v)$ pretok (torej število trgovcev) po povezavi od u do v, če taka povezava obstaja in če tok res teče od u do v; če pa teče v obratno smer ali če povezave ni, naj bo $f(u,v) = 0$. Funkcijo f lahko potem posplošimo tudi na množice točk:

$$f(A,B) := \sum_{u \in A} \sum_{v \in B} f(u,v) \text{ in } f(A,v) := f(A,\{v\}) \text{ in } f(u,B) = f(\{u\},B).$$

Množico vseh točk (vasi) označimo z V. Vhodni tok v točko u je potem enak $f(V,u)$, izhodni pa $f(u,V)$; razliko med njima označimo z $g(u) := f(V,u) - f(u,V)$. Na začetku dobimo tak pretok po grafu, pri katerem je $g(u)$ vedno večkratnik M; pretok bomo postopoma popravljali, dokler ne bodo vse te $g(u)$ postale enake 0 (naloga omenja tudi možnost, da rešitev morda sploh ne obstaja, vendar bomo videli, da do tega v resnici ne more priti).

Če še niso vse $g(u) = 0$, mora obstajati neka točka u, pri kateri sta izhodni in vhodni tok različna; recimo, da je izhodni tok večji od vhodnega, torej $g(u) < 0$ (razmislek za drugo možnost, torej da je vhodni tok večji od izhodnega, je povsem analogen). Naj bo U množica vseh točk, ki so dosegljive iz te u tako, da gremo vedno le po povezavah v smeri toka (torej gremo lahko od v do w le, če je $f(v,w) > 0$). To pomeni, da po povezavah, ki imajo eno krajišče v U, drugo pa zunaj njega, tok teče vedno v U, nikoli ven iz U (kajti sicer bi prišlo tudi tisto drugo krajišče v množico U). Zato izhodni tok iz točk U-ja lahko teče le v druge točke U-ja, ne pa ven iz U-ja (v točke iz $V - U$): $f(U, V - U) = 0$ in zato $f(U,V) = f(U,U)$. Vhodni tok v točke U-ja prihaja deloma iz drugih točk U-ja, poleg tega pa lahko tudi še iz točk zunaj U-ja: $f(V,U) = f(U,U) + f(V - U, U)$. Če obe opažanji združimo, vidimo, da je $f(V,U) = f(U,V) + f(V - U, U)$, torej $f(V,U) - f(U,V) = f(V - U, U)$. Na levi strani imamo ravno razliko med vhodnim tokom v vse točke U-ja ter izhodnim tokom iz vseh točk U-ja: velja torej $\sum_{v \in U} g(v) = f(V - U, U)$. Po predpostavki za vsaj eno točko U-ja, namreč u, velja $g(u) < 0$ (torej ima večji izhodni tok kot vhodni). Ali je mogoče, da bi za vse ostale točke $v \in U$ veljalo $g(v) \leq 0$, torej da bi bil izhodni tok pri vseh teh točkah vsaj tolikšen kot vhodni? Potem bi bilo $\sum_{v \in U} g(v) < 0$, torej $f(V - U, U) < 0$, kar pa je nemogoče, saj smo funkcijo f definirali tako, da je vedno ≥ 0. Nujno mora torej obstajati neka taka točka $v \in U$, pri kateri je $g(v) > 0$, torej je vhodni tok v v večji od izhodnega toka iz nje.

Ker so vse točke U-ja dosegljive iz u tako, da hodimo po povezavah v smeri toka, mora veljati to tudi za v. Primerno pot od u do v lahko poiščemo z iskanjem v širino; naj bo torej $u_0, u_1, \ldots, u_k$ neka taka pot, kjer je $u_0 = u$, $u_k = v$, za vsak i pa je $f(u_{i-1}, u_i) > 0$. Pošljimo zdaj dodatnih m enot pretoka po tej poti v obratni smeri. Kaj se zgodi? Prej je po povezavi med u_{i-1} in u_i teklo na primer c_i enot toka iz u_{i-1} v u_i, zdaj pa po tej povezavi teče $m - c_i$ enot toka iz u_i v u_{i-1}. Ker so bili prej vsi tokovi večji od 0 in manjši od m, so zdaj tudi (spremenili pa so smer). Vmesne točke na poti, torej $u_1, \ldots, u_{k-1}$, so pridobile m enot vhodnega in m izhodnega toka, zato se jim $g(u_i)$ ni spremenila, po drugi strani pa se je $g(u)$ (ki je bila manjša od 0) povečala za m, vrednost $g(v)$ (ki je bila večja od 0) pa se je zmanjšala za m. Tako smo torej dvema točkama premaknili g za m bliže k 0, ostalim točkam pa ga nismo spremenili.

Ta postopek lahko zdaj ponavljamo v zanki, dokler ne pridejo vse $g(0)$ na 0. Kako dolgo bo to trajalo? Ker so bili v začetnem grafu tokovi na posameznih povezavah z območja od 1 do $m-1$, je imela točka stopnje d lahko kvečjemu $(m-1)\cdot d$ vhodnega in izhodnega toka; torej je bilo $|g(u)| < m\cdot d$. Taka točka bo torej potrebovala manj kot d popravkov, da pride $g(u)$ na 0. Če to seštejemo po vseh točkah, se njihove stopnje seštejejo v dvakratnik števila povezav v grafu; skupaj potrebujemo torej $O(r)$ popravkov, vsak popravek pa nam vzame $O(n + r)$ časa (zaradi iskanja v širino).

Rešitev lahko še izboljšamo, če opazimo njeno sorodnost s problemom največjega pretoka v grafu. Definirajmo nov graf G', ki ga dobimo iz prvotnega G takole: vzemimo vse G-jeve točke in jim dodajmo še dve novi, izvor s in ponor t; za vsaki dve točki u in v, kjer je v G-ju $f(u, v) > 0$, dodajmo v G' usmerjeno povezavo $u \to v$ s kapaciteto 1; za vsako točko u, ki ima (v G-ju) več izhodnega toka kot vhodnega, dodajmo v G' povezavo $s \to u$ s kapaciteto $-g(u)/m$; in za vsako točko, ki ima več vhodnega toka kot izhodnega, dodajmo v G' povezavo $u \to t$ s kapaciteto $g(u)/m$. Iščimo v tem novem grafu maksimalni pretok od s do t s Ford-Fulkersonovim algoritmom. Vsakič ko ta algoritem najde v G' neko novo nezasičeno pot od s do t in jo zasiti s tem, ko pošlje po njej eno dodatno enoto pretoka, bi lahko naš prvotni algoritem v G po isti poti (le brez prvega koraka iz s ter zadnjega koraka v t) poslal m enot pretoka v nasprotni smeri in obratno; oba algoritma torej pravzaprav rešujeta isti problem na enak način.

Nalogo lahko torej rešimo tako, da v G' poiščemo maksimalni pretok in potem za vsako povezavo, po kateri v njem res teče tok, pošljemo v G po m enot toka v nasprotni smeri. Če za iskanje maksimalnega pretoka v G' namesto Ford-Fulkersonovega algoritma uporabimo kakšnega drugega, lahko časovno zahtevnost zmanjšamo z $O(r^2)$ na $O(n \cdot r)$.[18] Vendar pa so pri naši nalogi grafi dovolj majhni, da takšnih izboljšav ne potrebujemo.

J. Ponovitve

Uporabljali bomo pythonovsko notacijo za podnize: če je w niz dolžine n, so njegovi posamezni znaki $w[0], w[1], \ldots, w[n-1]$; $w[i:j]$ je podniz, ki ga tvorijo znaki $w[i], \ldots, w[j-1]$; $w[:j] = w[0:j]$ in $w[i:] = w[i:n]$; če je kakšen od indeksov negativen, pa mu v mislih prištejmo n. Tako na primer $w[-3:]$ predstavlja podniz, ki ga tvorijo zadnji trije znaki niza w. Dolžino niza w pišemo tudi kot $|w|$. Dolžino najdaljšega v, za katerega se vv pojavlja kot podniz v w, označimo s $f(w)$; to je funkcija, ki jo bomo morali znati pri tej nalogi učinkovito računati.

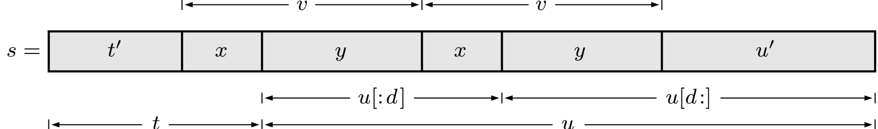
Za začetek odmislimo poizvedbe in si mislimo, da iščemo najdaljši dvojni podniz v celotnem nizu s. Nalogo lahko rešujemo z rekurzivnim razmislekom: razdelimo s na dva približno enako dolga dela, $s = tu$. Najdaljši dvojni podniz v s leži bodisi v celoti v t bodisi v celoti v u bodisi prečka mejo med t in u. Tiste, ki ležijo v celoti v t ali v celoti v u, lahko poiščemo z rekurzivnim klicem; ostane torej še vprašanje, kako poiskati najdaljši tak dvojni podniz vv, ki prečka mejo med t in u. Dolžini tega podniza recimo $M(t, u)$ — to nam bo prišlo prav, da se bomo kasneje sklicevali

[18] Gl. npr. James B. Orlin, “Max flows in $O(nm)$ time, or better”, v *Proc. of the 45th Annual* ACM *Symposium on the Theory of Computing* (STOC 2013), str. 765–774.

na postopek, s katerim bomo znali poiskati ta podniz. Doslej smo torej ugotovili, da je $f(tu) = \max\{f(t), f(u), M(t, u)\}$.

Če torej vv prečka mejo med t in u, sta glede položaja te meje dve možnosti: lahko leži v prvem v-ju ali pa v drugem.

(1) Oglejmo si najprej prvo možnost: takrat meja razdeli tisti prvi v na dva dela, recimo x in y. Imamo torej $v = xy$, oba dela niza s pa sta oblike $t = t'x$ in $u = yxyu'$, kot kaže slika:



Pišimo $d = |v| = |xy| = |yx|$; potem vidimo, da se niza t in $u[:d]$ oba končata na x, niza u in $u[d:]$ pa se oba začneta na y. Koristno bi bilo torej poznati dolžino najdaljšega skupnega sufiksa nizov t in $u[:d]$ — recimo ji d' — in dolžino najdaljšega skupnega prefiksa nizov u in $u[d:]$ — recimo ji d''. Če je $d' + d'' \geq d$, potem lahko res vzamemo nek dovolj dolg sufiks x nizov t in $u[:d]$ ter nek dovolj dolg prefiks y nizov u in $u[d:]$, da bo t oblike $t'x$ in da bo u oblike $yxyu'$, torej bomo imeli tu dvojni podniz $xyxy$ za $|xy| = d$, čigar druga polovica v celoti leži v u. Če pa je $d' + d'' < d$, takega podniza pri tem d ni. V zanki moramo iti po vseh d (od 1 do $|u|$) in poiskati najdaljšega, pri katerem dvojni podniz obstaja. (Najbolj leva pojavitev tako dolgega dvojnega podniza je potem tista, ki se začne d' znakov pred koncem t-ja.)

Za izračun d' in d'' si lahko pomagamo z znano Z-funkcijo. Za dani niz w je mogoče v $O(|w|)$ časa izračunati tabelo $Z(w)$, v kateri element $Z(w)[i]$ za $i = 0, \ldots, |w|$ pove dolžino najdaljšega skupnega prefiksa nizov w in $w[i:]$.[19] V našem primeru lahko torej izračunamo $Z(u)$, pa bomo v $Z(u)[d]$ dobili ravno tisto, kar potrebujemo za d''. Za d' moramo biti malo bolj zviti: izračunajmo $Z(t^R\#u^R)$, pri čemer ^{R} pomeni, da niz obrnemo z desne na levo; za simbol # vzemimo poljuben tak znak, ki se ne pojavlja v t in u. Dobili bomo tabelo z dolžinami najdaljših skupnih prefiksov med $t^R\#u^R$ in njegovimi sufiksi, med drugim tudi tistimi oblike $u^R[i:]$; ker pa slednji ne vsebuje #, bodo taki skupni prefiksi obenem tudi skupni prefiksi nizov t^R in $u^R[i:]$; to pa so obenem tudi skupni sufiksi nizov t in $u[:-i]$. Prav to pa potrebujemo mi za d'; vzeti moramo torej $d' = Z(t^R\#u^R)[|tu| + 1 - d]$.

(2) Pri drugi možnosti, torej ko meja med t in u leži v drugem v-ju namesto v prvem, je vse čisto analogno kot pri prvi. Zdaj nam meja razdeli drugi v na dva dela in imamo $v = xy$, $t = t'xyx$ in $u = yu'$. Pišimo spet $d = |v| = |xy| = |yx|$ in vidimo, da se niza $t[:-d]$ in t oba končata na x, niza $t[-d:]$ in u pa se oba začneta na y. Naj bo torej d' dolžina najdaljšega skupnega sufiksa nizov $t[:-d]$ in t; in naj bo d'' dolžina najdaljšega skupnega prefiksa nizov $t[-d:]$ in u. Potem primeren dvojni podniz vv z $|v| = d$ tukaj obstaja natanko tedaj, če je $d' + d'' \geq d$; podobno kot prej moramo v zanki pregledati vse d od 1 do $|t|$ in si zapomniti največji d,

[19] Gl. npr. D. Gusfield, *Algorithms on Strings, Trees, and Sequences* (1997), str. 7–10. Pod imenom „Z-algoritem“ ali „Z-funkcija“ je opisana tudi na mnogih spletnih straneh, povezanih s tekmovalnim programiranjem, npr. `https://cp-algorithms.com/string/z-function.html`; M. Crochemore in W. Rytter (*Jewels of Stringology* (2002), str. 36–39) pa jo imenujeta preprosto „tabela prefiksov“.

pri katerem je omenjena neenačba izpolnjena; najbolj leva pojavitev tako dolgega dvojnega podniza pa je spet tista, ki se začne d' znakov pred koncem t-ja.

Za izračun d' in d'' lahko tudi zdaj uporabimo Z-funkcijo: $d' = Z(t^R)[d]$ in $d'' = Z(u\#t)[|tu| + 1 - d]$. Do enakega rezultata lahko pridemo tudi tako, da niz s gledamo od desne proti levi, $s^R = u^R t^R$; meja med u^R in t^R torej zdaj leži v prvem v-ju in lahko razmišljamo naprej enako kot pri točki (1).

Kakšna je časovna zahtevnost tega postopka za izračun $f(tu)$? Pri iskanju dvojnih podnizov, ki prečkajo mejo med t in u, smo porabili $O(|s|)$ časa za izračun vseh Z-funkcij in tudi $O(|s|)$ časa za pregled vseh možnih d-jev pri eni in drugi možnosti glede položaja meje. Poleg tega pa smo izvedli še dva rekurzivna klica za pol krajša niza (namreč t in u). Tako imamo časovno zahtevnost $T(n) = O(n) + 2T(n/2)$, kar pomeni $T(n) = O(n \log n)$.

Kot smo že rekli, bomo v resnici morali odgovoriti na več poizvedb za različne podnize niza s. Če bi za vsako od q poizvedb posebej pognali doslej opisani postopek za ustrezni podniz, bi imela naša rešitev časovno zahtevnost $O(qn \log n)$, kar bi bilo že prepočasi. Pomagati si moramo z dejstvom, da se vse naše poizvedbe nanašajo na podnize enega in istega niza s.

Razdelimo naš s na podnize dolžine 2^ℓ za $\ell = 0, 1, 2, \ldots, \lfloor \log_2 |s| \rfloor$. Pišimo $s^\ell(i) = s[i \cdot 2^\ell : (i+1) \cdot 2^\ell]$; in naj bo $R^\ell[i] = f(s^\ell(i))$. Vrednosti R si bomo izračunali na začetku in jih shranili v tabele, ker nam bodo prišle prav pri odgovarjanju na poizvedbe. Pri $\ell = 0$ imamo podnize dolžine 1, v katerih se lahko dvojni podniz pojavlja le, če je prazen, torej je $R^0[i] = 0$. Pri večjih ℓ pa lahko uporabimo naš dosedanji rekurzivni razmislek: niz $s^\ell(i)$ je sestavljen iz leve polovice $s^{\ell-1}(2i)$ in desne polovice $s^{\ell-1}(2i+1)$. Dvojni podniz v njem leži bodisi v levi polovici bodisi v desni bodisi prečka mejo med njima, torej imamo

$$R^\ell[i] = \max\{R^{\ell-1}[2i], R^{\ell-1}[2i+1], M(s^{\ell-1}(2i), s^{\ell-1}(2i+1))\}.$$

Tako lahko računamo vse vrednosti R po naraščajočih ℓ in pri vsakem ℓ po vseh i. Ker gre pri posameznem ℓ lahko i le od 0 do $\lfloor n/2^\ell \rfloor - 1$ in ker za vsak i porabimo $O(2^\ell)$ časa, da izračunamo $R^\ell[i]$ (zaradi klica funkcije M na dveh podnizih dolžine $2^{\ell-1}$), nam izračun vseh R^ℓ vzame $O(n)$ časa pri vsakem ℓ, za vse ℓ skupaj pa torej $O(n \log n)$.

Ko potem dobimo poizvedbo za podniz $s[i:j]$, si lahko tega predstavljamo kot sestavljenega iz več kosov — nizov oblike $s^\ell(\cdot)$, in sicer po največ dveh pri vsakem ℓ, enega na levi in enega na desni; kajti če bi hoteli uporabiti po tri ali več zaporedne nize na istem nivoju ℓ, bi lahko vsaj dva od njih zamenjali z enim daljšim na nivoju $\ell+1$. Najdaljši dvojni podniz v $s[i:j]$ potem bodisi v celoti leži v enem od teh kosov (in dolžine teh podnizov že imamo v tabelah R^ℓ) bodisi prečka kakšno mejo med dvema takima kosoma (take dvojne podnize pa poiščemo s prej omenjeno funkcijo M). Zapišimo ta postopek s psevdokodo:

$i' := i$; $j' := j$; $r_L := 0$; $r_D := 0$; $\ell := 0$;
while $i' < j'$:
  (* *Tu sta* i' *in* j' *večkratnika* 2^ℓ; $r_L = f(s[i:i'])$ *in* $r_D = f(s[j':j])$. *)
  if je $i'/2^\ell$ liho:
    $r_L := \max\{r_L, R^\ell[i'/2^\ell], M(s[i:i'], s[i':i'+2^\ell])\}$;
    $i' := i' + 2^\ell$;

if je $j'/2^\ell$ liho:
  $r_D := \max\{r_D, R^\ell[j'/2^\ell - 1], M(s[j' - 2^\ell : j'], s[j' : j])\}$;
  $j' := j' - 2^\ell$;
$\ell := \ell + 1$;
(* *Zdaj je* $i' = j'$ *in še vedno* $r_L = f(s[i:i'])$ *in* $r_D = f(s[j':j])$. *)
return $\max\{r_L, r_D, M(s[i:i'], s[j':j])\}$; (* *to je* $f(s[i:j])$ *)

Zanka je izvedla le $O(\log n)$ iteracij; dolžina nizov, na katerih pri posameznem ℓ kličemo funkcijo M, pa je največ 2^ℓ, zato tista dva klica porabita $O(2^\ell)$ časa; vsota tega po vseh ℓ je $O(j - i)$. Tako lahko na posamezno poizvedbo odgovorimo v $O(j - i)$ časa, pri čemer je $j - i$ dolžina tistega podniza s-ja, na katerega se nanaša ta poizvedba. Časovna zahtevnost celotnega postopka s predpripravo (računanjem vseh tabel R^ℓ) vred je tako v najslabšem primeru $O(n \log n + nq)$.

K. Enotirna železnica

Naj bo $m = n - 1$; železnica je torej sestavljena iz m odsekov, pri čemer trajanje vožnje po i-tem odseku (med postajama i in $i + 1$) označimo z a_i. Vsoto prvih p odsekov označimo s $s_p := \sum_{i=1}^{p} a_i$. Skupni čas vožnje po celi progi je potem s_m.

Recimo, da se hočeta vlaka srečati na postaji $p + 1$. Levi vlak (tisti, ki pelje od postaje 1 proti postaji n), bo postajo $p + 1$ dosegel, ko bo prevozil prvih p odsekov, kar mu vzame skupno s_p časa; desni vlak (tisti, ki pelje od postaje n proti postaji 1) pa jo bo dosegel, ko bo prevozil zadnjih $m - p$ odsekov, kar mu vzame skupno $s_m - s_p$ časa.

Če je $s_p \leq s_m - s_p$ (ali, z drugimi besedami, če je $s_p \leq s_m/2$), pride levi vlak prvi na to postajo in mora čakati desnega; čaka ga torej $(s_m - s_p) - s_p = s_m - 2s_p$ časa. Vidimo, da je ta čas čakanja tem manjši, čim večji je s_p; najkrajši čas čakanja za ta primer bomo torej dobili, če poiščemo največjo s_p, ki še ne preseže $s_m/2$.

Podobno pa, če je $s_p > s_m - s_p$ (torej če je $s_p > s_m/2$), pride desni vlak prvi na postajo in mora čakati levega, in sicer čaka $s_p - (s_m - s_p) = 2s_p - s_m$ časa. Tu je torej čas čakanja tem manjši, čim manjši je s_p; najkrajši čas za ta primer bomo torej dobili, če poiščemo najmanjšo s_p, ki še ne pade pod $s_m/2$.

Če oba prejšnja odstavka združimo, lahko zaključimo takole: poiskati moramo največji p, pri katerem je s_p še $\leq s_m/2$ (zanj je potemtakem $p + 1$ najmanjši tak indeks, pri katerem je $s_{p+1} > s_m/2$); najmanjši čas čakanja je potem $\min\{s_m - 2s_p, 2s_{p+1} - m\}$. Vprašanje je le še, kako čim hitreje poiskati pravi p.

Ko se čas vožnje a_i spremeni, se spremenijo tudi vsote $s_i, s_{i+1}, \ldots, s_m$; zato si ne moremo privoščiti, da bi jih hranili eksplicitno, ker bi nam vzelo popravljanje vseh teh vsot preveč časa. Pomagamo si lahko z neke vrste drevesom segmentov, v katerem vozlišča hranijo vsote po dveh, štirih, osmih itd. zaporednih vrednosti a_i. Imejmo torej tabele $A_0, A_1, \ldots, A_R$ za $R = \lceil \log_2 m \rceil$; tabela A_r naj ima $\lceil m/2^r \rceil$ elementov, pri čemer element $A_r[i]$ (indeksi i naj gredo od 0 naprej, ne od 1 naprej) hrani vsoto števil a_j za $2^r i < j \leq 2^r(i + 1)$ (pri tem si za $j > m$ seveda mislimo $a_j = 0$). Zadnja tabela ima en sam element, ki je ravno vsota vseh a_i, torej: $A_R[0] = s_m$.

V tabeli A_0 imamo torej ravno posamezne vrednosti a_i, v više ležečih tabelah pa je vsak element vsota dveh v naslednji nižji tabeli: $A_r[i] = A_{r-1}[2i] + A_{r-1}[2i + 1]$.

Tako lahko tabele na začetku tudi inicializiramo: začetne vrednosti a_i vpišemo v $A_0[i-1]$, nato pa s seštevanjem izračunamo še vse ostale A_r.

Teh tabel tudi ni težko popraviti, ko se kakšna vrednost a_i spremeni. Recimo, da se a_i poveča za d (če se zmanjša, vzemimo pač negativen d); za vsak r moramo zdaj prišteti d k vrednosti $A_r[\lfloor(i-1)/2^r\rfloor]$, kajti le tam nastopa vsota, ki obsega tudi člen a_i. Tako imamo z vsako spremembo posamezne vrednosti a_i le $O(\log n)$ dela.

Največje vrednosti s_p, ki še ne presega $s_m/2$, zdaj ni težko računati s spuščanjem po drevesu. Na vsakem nivoju se premaknemo tako daleč naprej, kolikor se še lahko, ne da bi presegli $s_m/2$:

$r := R$; $i := 0$; (* *r je trenutni nivo; i je indeks v A_r* *)
$L := 0$; $V := A_R[0]$; (* *L je vsota levo od i; V je vsota vseh a_i* *)
while $r > 0$:
  (* *Tu velja: $V = s_m$; $L = s_p$ za $p = 2^r \cdot i$;*
    s_p je še $\leq s_m/2$, če pa mu prištejemo še $A_r[i]$ in tako iz njega
    naredimo s_q za $q = 2^r \cdot (i+1)$, bo že prevelik: $s_q > s_m/2$. *)
  $r := r - 1$; $i := 2i$;
  (* *Tu je L še vedno enak s_p za $p = 2^r \cdot i$; in $s_p \leq s_m/2$. Toda zdaj*
    mu lahko mogoče prištejemo $A_r[i]$, ne da bi vsota presegla $s_m/2$. *)
  if $L + A_r[i] \leq V/2$:
    $L := L + A_r[i]$; $i := i + 1$;
(* *Tu velja: $V = s_m$ in $L = s_i \leq s_m/2$, medtem ko je s_{i+1} že $> s_m/2$.* *)
$L' := L + A_0[i]$; (* *Zdaj je $L' = s_{i+1} > s_m/2$.* *)
return $\min\{V - 2L, 2L' - V\}$;

Rezultat, ki ga vrne ta postopek na koncu, je ravno najmanjši potrebni čas čakanja. Ker smo imeli na vsakem nivoju drevesa le konstantno mnogo dela, ima ta postopek časovno zahtevnost $O(\log n)$. Tako imamo torej rešitev, ki porabi skupno $O(n + k \log n)$ časa (namreč $O(n)$ za inicializacijo in potem $O(\log n)$ po vsaki od k sprememb) in $O(n)$ prostora (za tabele A_r).

L. Sistematični trgovski potnik

Nalogo lahko rešujemo z dinamičnim programiranjem. Recimo, da imamo pred seboj množico n točk A, ki jo po x-koordinati razdelimo na levo polovico B (levih $\lfloor n/2\rfloor$ točk) in desno polovico C (desnih $\lceil n/2\rceil$ točk), in da mora zdaj naš trgovski potnik bodisi obiskati najprej vse točke iz B in potem vse iz C bodisi najprej vse iz C in potem vse iz B. Med tema dvema možnostma za nas pravzaprav ni velike razlike, kajti če imamo na primer pot, ki najprej obišče vse točke iz C in potem vse iz B, gremo lahko po isti poti tudi v obratni smeri in torej najprej obiščemo vse točke iz B in potem vse iz C.

Recimo torej, da se naša pot začne v točki $b \in B$, obišče nato vse točke iz B, nato prestopi v C, obišče vse točke še tam in se konča v točki $c \in C$. Dolžino najkrajše take poti imenujmo $f_A(b, c)$. Prestop iz B v C se zgodi tako, da iz neke točke $b' \in B$ odpotujemo v neko točko $c' \in C$; prvi del poti torej obišče vse točke iz B ter se pri tem začne v b in konča v b'; najkrajša taka pot pa je dolga $f_B(b, b')$. Podobno drugi del poti obišče vse točke iz C in se začne v c ter konča v c'; najkrajša taka pot je

dolga $f_C(c', c)$. Na koncu moramo seveda b' in c' izbrati tako, da bo skupna dolžina poti čim manjša:

$$f_A(b, c) = \min\{f_B(b, b') + d(b', c') + f_C(c', c) : b' \in B,\ c' \in C\}.$$

Pri tem smo z $d(\cdot, \cdot)$ označili razdaljo (evklidsko) med dvema točkama.

Vidimo torej, da lahko nalogo rešujemo rekurzivno: preden začnemo računati f_A, z rekurzivnima klicema rešimo nalogo za podmnožici B in C (le da bomo njiju delili na spodnjo in zgornjo polovico namesto na levo in desno) ter tako dobimo f_B in f_C, ki ju potrebujemo za izračun f_A. Paziti pa moramo na naslednje: če bomo pri vsaki kombinaciji b in c pregledali vse možne kombinacije b' in c', bomo za izračun f_A porabili $O(n^4)$ časa, kar bo gotovo prekoračilo časovno omejitev (spomnimo se, da gre n pri tej nalogi do 1000). Boljšo rešitev dobimo, če izračun f_A razbijemo na dva dela: opazimo, da je to, pri katerem b' je najbolje končati prvi del naše poti (po B), odvisno le od tega, pri katerem c' bomo začeli drugi del naše poti (po C), ne pa tudi od tega, pri katerem c bomo ta drugi del poti končali. Definirajmo torej $g_A(b, c')$ kot dolžino najkrajše poti, ki se začne v $b \in B$, obišče vse točke iz B ter na koncu naredi en korak iz B v $c' \in C$:

$$g_A(b, c') = \min\{f_B(b, b') + d(b', c') : b' \in B\},$$

kar lahko izračunamo v $O(n^3)$ časa za vse b in c'. S pomočjo te funkcije pa lahko računamo f_A po formuli

$$f_A(b, c) = \min\{g_A(b, c') + f_C(c', c) : c' \in C\}.$$

Ker bomo morali na koncu tudi izpisati dejanski potek najkrajše poti, je seveda koristno, če si ob izračunu vsakega minimuma tudi zapišemo, pri katerem b' oz. c' je bil dosežen.

Kakšna je časovna zahtevnost te rešitve? Pri množici A velikosti n smo izvedli rekurzivna klica za dve podmnožici velikosti $n/2$, nato pa porabili $O(n^3)$ časa za izračun g_A in f_A. Tako imamo časovno zahtevnost $T(n) = 2T(n/2) + O(n^3)$, iz česar dobimo $T(n) = O(n^3)$.

Ko računamo $g_A(b, c')$, pazimo še na to, da potnik pri obiskovanju točk iz B seveda razdeli tudi to množico na dve polovici (spodnjo in zgornjo) in obišče eno polovico pred drugo; pot $b \rightsquigarrow b' \to c'$, kakršno iščemo, je torej možna le za take b', ki ne pripadajo isti polovici množice B kot točka b. Podobno tudi pri izračunu $f_A(b, c)$ pridejo za pot $b \rightsquigarrow c' \rightsquigarrow c$ v poštev le take točke c', ki ne pripadajo isti polovici množice C kot točka c. Ta problem se reši sam od sebe, če si predstavljamo, da je $f_B(b, b') = \infty$ v primerih, ko sta b in b' oba iz iste polovice množice B; in podobno za f_C. Če pa se odločimo hraniti vse funkcije $f_\bullet$ v eni sami tabeli velikosti $n \times n$ elementov, moramo omenjeni pogoj preverjati eksplicitno; če tam namreč na primer vzamemo b' iz iste polovice množice B kot točko b — recimo tej polovici B_1 — bomo v $f[b, b']$ našli neki veljaven rezultat (in ne ∞), le da se ne bo nanašal na funkcijo f_B, pač pa na f_{B_1} ali morda celo na rešitev kakšne še manjše podmnožice množice B_1.

vhod: tabela točk $t[1..n]$
globalne spremenljivke: tabele f, $\tilde{b}'$, $\tilde{c}'$ velikosti $n \times n$

podprogram $\textsc{Rekurzija}(i,\ k,\ os)$:
(* *Rešiti moramo problem za* $A = \{t[i], \ldots, t[i+k-1]\}$, *pri čemer „os“ pove, ali točke urejamo po osi* x *ali po osi* y. *)

if $k = 1$: (* *Robni primer: ena sama točka.* *)
$f[k,k] := 0$; **return;**

uredi točke $t[i], \ldots, t[i+k-1]$ po x-koordinati, če je $os = 0$,
oz. po y-koordinati, če je $os = 1$;
$k_B := \lfloor k/2 \rfloor$; $k_C := \lceil k/2 \rceil$; $i_B := i$; $i_C := i + k_B$; $i_D := i + k$;
$m_B := i_B + \lfloor k_B/2 \rfloor$; $m_C := i_C + \lfloor k_C/2 \rfloor$;
(* *B obsega točke od i_B do $i_C - 1$; njegova druga polovica se začne pri m_B.*
C obsega točke od i_C do $i_D - 1$; njegova druga polovica se začne pri m_C. *)
$\textsc{Rekurzija}(i_B,\ k_B,\ 1 - os)$;
$\textsc{Rekurzija}(i_C,\ k_C,\ 1 - os)$;

for $b := i_B$ **to** $i_C - 1$:
  for $c' := i_C$ **to** $i_D - 1$:
    $g[c'] := \infty$; $j := i_B$; $j' := i_C$;
    if $k_1 > 1$ **then if** $b < m_B$ **then** $j := m_B$ **else** $j' := m_B$;
    (* *Tista polovica B-ja, ki ne vsebuje b, obsega točke od j do $j' - 1$.* *)
    for $b' := j$ **to** $j' - 1$:
      $r := f[b, b'] + d(t[b], t[b'])$;
      if $r < g[c']$ **then** $g[c'] := r$, $\tilde{g}[c'] := b'$;
  for $c := i_C$ **to** $i_D - 1$:
    $f[b,c] := \infty$; $j := i_C$; $j' := i_D - 1$;
    if $k_2 > 1$ **then if** $c < m_C$ **then** $j := m_C$ **else** $j' := m_C$;
    (* *Tista polovica C-ja, ki ne vsebuje c, obsega točke od j do $j' - 1$.* *)
    for $c' := j$ **to** $j' - 1$:
      $r' := g[c'] + f[c', c]$;
      if $r < f[b,c]$ **then** $f[b,c] := r$, $\tilde{b}'[b,c] := \tilde{g}[c']$, $\tilde{c}'[b,c] := c'$;
    $f[c,b] := f[b,c]$; $\tilde{b}'[c,b] := \tilde{b}'[b,c]$; $\tilde{c}'[c,b] := \tilde{c}'[b,c]$

glavni klic: $\textsc{Rekurzija}(1,\ n,\ 0)$;

Vrednosti $g_A(b, \cdot)$ potrebujemo le pri izračunu $f_A(b, \cdot)$, kasneje pa jih lahko pozabimo, zato smo za shranjevanje vrednosti funkcije g_A uporabili le enodimenzionalno tabelo: vrednost $g_A(b, c')$ shranimo v $g[c']$, spotoma pa si v $\tilde{g}[c']$ še zapišemo, pri katerem b' je bila ta rešitev dosežena. Ko pa izračunamo $f_A(b, c)$, jo shranimo v $f[b, c]$, spotoma pa si v $\tilde{b}'[b, c]$ in $\tilde{c}'[b, c]$ zapišemo, pri katerih b' in c' je bila vrednost funkcije $f_A(b, c)$ dosežena. Rešitve za (b, c) lahko uporabimo tudi za (c, b), saj lahko pot prehodimo tudi v nasprotni smeri in ostane veljavna in enako dolga; na koncu torej skopiramo $f[b, c]$ v $f[c, b]$ in podobno za $\tilde{b}'$ in $\tilde{c}'$.

S pomočjo tabel $\tilde{b}'$ in $\tilde{c}'$ lahko na koncu rekonstruiramo celoten potek najkrajše poti. Tudi to lahko naredimo z rekurzijo:

globalna spremenljivka: tabela $p[1..n]$;

podprogram $\textsc{Pot}(i,\ k,\ u,\ v,\ j)$:
(* *V $p[j..j+k-1]$ moramo shraniti potek najkrajše poti od u do v,*

ki obišče vse točke iz $A = \{t[i], \ldots, t[i+k-1]\}$. *)
if $k = 1$ **then** $p[j] := i$; **return**; (* *robni primer* *)

(* *Razdelimo v mislih* A *na* B *in* C*, enako kot zgoraj.* *)
$k_B := \lfloor k/2 \rfloor$; $k_C := \lceil k/2 \rceil$; $i_B := i$; $i_C := i + k_B$; $i_D := i + k$;

(* *Rekurzivno pripravimo potek poti v vsaki polovici posebej.* *)
if $u < v$: (* *torej je* $u \in B$, $v \in C$ *)
  Pot(i_B, k_B, u, $\tilde{b}'[u,v]$, j);
  Pot(i_C, k_C, $\tilde{c}'[u,v]$, v, $j + k_B$);

else: (* *torej je* $u \in C$, $v \in B$ *)
  Pot(i_C, k_C, u, $\tilde{c}'[u,v]$, j);
  Pot(i_B, k_B, $\tilde{b}'[u,v]$, v, $j + k_C$);

Ta postopek poženemo tako, da za A vzamemo množico vseh n točk, jo razdelimo na B in C ter poiščemo tista $b \in B$ in $c \in C$, pri katerih je $f[b,c]$ najmanjša; nato pa poženemo Pot$(1, n, b, c, 1)$.

Pri izpisu tako dobljene poti moramo paziti na to, da bomo izpisali indekse točk v vhodnih podatkih, ne pa v našem preurejenem vrstnem redu; prvotne indekse je zato koristno hraniti v tabeli t skupaj s koordinatami točk.

REŠITVE NALOG POSKUSNEGA TEKMOVANJA

X. Anagram

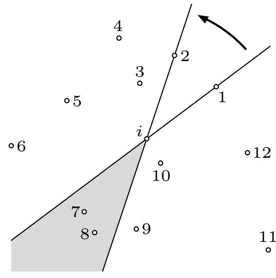
Preprost način za odkrivanje anagramov je, da v vsaki besedi uredimo črke po abecedi. Če sta si bili dve besedi anagrama, to pomeni, da sta vsebovali enake črke, le v različnem vrstnem redu; po urejanju črk po abecedi pa je vrstni red v obeh besedah enak, torej sta si povsem enaki. Primer: iz anagramov *enajst* in *stanje* nastane po urejanju črk obakrat enak niz *aejnst*.

Naš program lahko torej v zanki bere besede, pri vsaki uredi njene črke in za tako dobljeni niz preveri, če smo enakega že dobili pri kakšni prejšnji besedi. V ta namen je koristno nize hraniti v razpršeni tabeli (v C++ uporabimo na primer razred `unordered_set` iz standardne knjižnice). Če niza, ki je nastal iz trenutne besede po urejanju črk, v tej razpršeni tabeli ne najdemo, ga vanjo dodamo, trenutno besedo pa izpišemo.

Ker so besede kratke, ni posebej pomembno, kako urejamo njihove črke; če pa bi bile zelo dolge, bi bilo koristno namesto urejanja črk pripraviti za vsako besedo kar tabelo 26 števil, ki bi za vsako črko abecede povedala, kolikokrat se pojavlja v besedi. Če sta dve besedi anagrama, bomo iz njiju dobili tabeli z enako vsebino, sicer pa ne.

Y. E(dolžina(KO))

Množico aktiviranih točk označimo z $A \subseteq T := \{1, 2, \ldots, n\}$. Verjetnost, da je aktivirana prav A, je $p(A) := \left(\prod_{i \in A} p_i\right)\left(\prod_{i \notin A}(1 - p_i)\right)$. Rob konveksne ovojnice sestavlja nekaj (vsaj tri) daljic, ki povezujejo po dve aktivirani točki; naj bo $f_{ij}(A)$ funkcija, ki ima vrednost 1, če je daljica med točkama i in j del roba konveksne



Primer določanja leve in desne množice. Točke smo uredili po kotu glede na i in jih v tem vrstnem redu oštevilčili. Ko je šla premica skozi i in $j = 1$, so bile levo od nje točke od 2 do $k_1 = 6$, desno pa točke od 7 do 12. Ko se j poveča na 2, točka 2 preneha biti leva, na novo pa levi postaneta točki 7 in 8 (v sivem območju), tako da so zdaj leve točke od 3 do $k_2 = 8$, desne pa od 9 do 1.

---

ovojnice množice A, sicer pa naj bo $f_{ij}(A) = 0$. Dolžino daljice med točkama i in j označimo z d_{ij}. Obseg konveksne ovojnice množice A je potem $d(A) := \sum_{1 \le i < j \le n} d_{ij} f_{ij}(A)$. Nalogo zanima povprečje (uteženo z verjetnostmi $p(A)$) tega obsega po vseh množicah A, torej

$$\begin{aligned} & \sum_{A \subseteq T} d(A) p(A) \\ = & \sum_{A \subseteq T} \sum_{1 \le i < j \le n} d_{ij} f_{ij}(A) p(A) \\ = & \sum_{1 \le i < j \le n} \sum_{A \subseteq T} d_{ij} f_{ij}(A) p(A) \\ = & \sum_{1 \le i < j \le n} d_{ij} p_{ij} \quad \text{za} \quad p_{ij} := \sum_{A \subseteq T} f_{ij}(A) p(A). \end{aligned}$$

V zadnji vrstici p_{ij} pomeni verjetnost, da je daljica med i in j del roba konveksne ovojnice množice A. Kdaj se to zgodi, torej kdaj je $f_{ij}(A) = 1$? To se zgodi v primeru, če sta krajišči i in j obe aktivirani (torej če sta $i \in A$ in $j \in A$) in če vse ostale točke iz A ležijo na isti strani premice skozi i in j — bodisi vse na levi bodisi vse na desni (recimo, da se postavimo v točko i in gledamo v smeri točke j). Označimo z L_{ij} oz. D_{ij} tiste točke iz T, ki ležijo na levi oz. desni strani te premice; to, da ležijo vse aktivirane točke levo od premice skozi i in j, se zgodi takrat, ko ni aktivirana nobena od desnih, in obratno. Oboje hkrati se ne more zgoditi, kajti potem bi bili aktivirani le dve točki (i in j), naloga pa zagotavlja, da so vedno aktivirane vsaj tri. Tako torej vidimo, da je

$$p_{ij} = p_i p_j \left(\prod_{k \in D_{ij}} (1 - p_k) + \prod_{k \in L_{ij}} (1 - p_k) \right).$$

Tega ne bi bilo težko računati, če bi za vsak par i in j šli v zanki po vseh k in preverjali, ali leži k ali desno od premice skozi i in j, ter ob tem počasi računali zmnožka vrednosti $1 - p_k$ za leve in za desne točke. Vendar bi imela taka rešitev časovno zahtevnost $O(n^3)$, kar je za nas že preveč.

Učinkovitejšo rešitev dobimo, če za vsak i najprej uredimo vseh ostalih $n-1$ točk po njihovem kotu glede na točko i.[20] Točke levo od premice skozi i in j potem tvorijo neko strnjeno podzaporedje tega vrstnega reda, recimo od $j + 1$ do k_j (zaporedje si predstavljajmo kot ciklično — če gredo ti indeksi čez $n - 1$, jim v mislih odštejmo

[20] Pri tem ni treba zares računati kotov, da ne bo težav z natančnostjo in ne-celimi števili. Ko primerjamo dve točki, poglejmo najprej, če ležita obe nad y_i ali obe pod y_i; če da, ju primerjajmo s pomočjo vektorskega produkta, sicer pa naj tista, ki leži pod y_i, pride v vrstnem redu za tisto, ki leži nad y_i.

$n-1$). Če zdaj počasi povečujemo j za 1, se tudi k_j vsakič lahko le poveča (ali pa ostane enak); po vsakem povečanju j-ja moramo torej povečevati k_j v korakih po 1, dokler ne pridemo spet do točke, ki leži desno od premice skozi i in j (glej sliko na str. 155). Ko naredi j cel krog, naredi tudi k_j cel krog; tako bomo v $O(n)$ časa za vse j dobili k_j. Ni težko tudi vzdrževati podatka o tem, koliko izmed levih točk ima verjetnost aktivacije 1; recimo, da je levih točk ℓ_j, od tega ℓ'_j z verjetnostjo aktivacije 1. Če je $\ell'_j > 0$, je nemogoče, da ne bi bila nobena leva točka aktivirana, sicer pa se to zgodi z verjetnostjo $(p^\star)^{\ell_j}$. Podobno razmišljamo tudi glede desnih točk; teh je $r_j := n-2-\ell_j$, od tega pa jih ima $r'_j := 3-\ell'_j - [\![p_i = 1]\!] - [\![p_j = 1]\!]$ verjetnost aktivacije 1. (Tu smo predpostavili, da imajo verjetnost aktivacije 1 le tri točke, kar je res pri $p^\star < 1$; pri $p^\star = 1$ pa je pač $r'_j = r_j$.) Dobili smo torej

$$p_{ij} = p_i p_j \left([\![r'_j = 0]\!] \, (1-p^\star)^{r_j} + [\![\ell'_j = 0]\!] \, (1-p^\star)^{\ell_j} \right).$$

Tako lahko pri danem i v $O(n \log n)$ časa uredimo točke po kotu in nato v $O(n)$ časa izračunamo vse k_j, k'_j, ℓ_j, ℓ'_j in p_{ij}. Ker moramo to storiti za vsak i, bo časovna zahtevnost te rešitve $O(n^2 \log n)$.

Z. Robin Hood

Nalogo lahko rešimo s simulacijo: za vsako od k tatvin moramo ugotoviti, kdo je takrat najbogatejši človek; če ima 100 ali manj enot denarja, se postopek ustavi in moramo izpisati „`impossible`“, sicer pa mu premoženje zmanjšamo za 100 enot in nadaljujemo. Da bomo lahko v vsakem trenutku hitro poiskali najbogatejšega človeka, je koristno ljudi hraniti v kopici (v C++ lahko uporabimo razred `priority_queue` iz standardne knjižnice), urejene po premoženju, tako da je najbogatejši vedno pri vrhu kopice.[21] Na vsakem koraku vzamemo najbogatejšega iz kopice, mu zmanjšamo premoženje in ga dodamo nazaj v kopico; ta operacija vzame $O(\log n)$ časa, zato je časovna zahtevnost celotne rešitve $O(k \log n)$. Šlo pa bi tudi s kakšno drugo podatkovno strukturo, npr. rdeče-črnim drevesom ali čim podobnim (v C++ bi lahko uporabili razred `map`).

Naloge so sestavili: gradnja na Luni, črke — Nino Bašić; pokrajinski razvoj — Gašper Fijavž in Tomaž Hočevar; enotirna železnica — Luka Fürst; ribolov, sistematični trgovski potnik, Robin Hood — Tomaž Hočevar; ponovitve — Tomaž Hočevar in Janez Brank; anagram — Vid Kocijan; rezanje kaktusov, DJ Darko — Patrik Pavić; premice na mreži, radar — Jure Slak; E(dolžina(KO)) — Mitja Trampuš; letalska družba — Janez Brank.

[21] Natančneje povedano, urediti jih je treba po parih $\langle$*premoženje*, *indeks*$\rangle$, naraščajoče po premoženju in pri enakem premoženju padajoče po indeksu; tako bomo izpolnili zahtevo naloge, da med več najbogatejšimi oropamo tistega z najmanjšim indeksom v vhodnih podatkih.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2019

1. Predstavitve

Besedilo, ki tvori posamezno alinejo, si lahko predstavljamo kot zaporedje nizov (vsak za eno vrstico). Posamezna stran lahko vsebuje več alinej, torej bo primerna podatkovna struktura seznam seznamov nizov (v spodnji rešitvi je to spremenljivka alineje). Ko pride ukaz „nova stran“ ali „pavza“, si s tem seznamom pomagamo, da vsebino strani izpišemo (podprogram Izpisi v spodnji rešitvi); morebitne prazne alineje pri tem preskočimo.

Ko pride ukaz „nova alineja“, dodamo novo (sprva prazno) alinejo na konec seznama alineje. Če pa je bila na koncu že od prej prazna alineja, ni treba dodajati še ene, saj praznih alinej tako ali tako ne bomo izpisovali.

Ko pride nov niz z besedilom (torej vrstica, ki ni ukaz), ga preprosto dodamo na konec seznama, ki predstavlja zadnjo alinejo doslej.

Naloga pravi še, naj ukaz „pavza“ ne izpiše strani še enkrat, če se od zadnje pavze ni spremenila (pri čemer dodajanje prazne alineje ne šteje za spremembo). Vpeljimo torej še globalno spremenljivko spremenjena, ki pove, ali se je stran od zadnjega izpisa že kaj spremenila. To postavimo na **true** vsakič, ko dodamo novo vrstico besedila v trenutno alinejo; ko stran izpišemo, pa postavimo spremenjena na **false**. Ko se začne nova stran (z ukazom „nova stran“), postavimo spremenjena na **true**, da se bo to stran sčasoma gotovo izpisalo, četudi bo morda takrat še prazna (brez besedila).

Za izpis prazne vrstice med stranmi lahko poskrbimo tako, da si v globalni spremenljivki prvilzpis zapomnimo, ali smo kakšno stran že izpisali ali ne; pred vsako stranjo razen prve pa izpišimo še prazno vrstico.

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

vector<vector<string>> alineje;
bool spremenjena = false, prvilzpis = true;

void Izpisi()
{
  if (! spremenjena) return;

  // Pred vsako izpisano stranjo razen prve izpišimo prazno vrstico.
  if (prvilzpis) prvilzpis = false; else cout << endl;

  cout << "nova stran" << endl;
  bool prva = true;
  for (const auto &alineja : alineje) {
    if (alineja.empty()) continue; // Prazne alineje preskočimo.

    if (prva) prva = false; else cout << "nova alineja" << endl;
    cout << "nova alineja" << endl;
    for (const auto &s : alineja) cout << s << endl; }

  spremenjena = false;
}

int main()
```

```
{
  string s;
  while (! getline(cin, s).fail())
  {
    if (s == "nova stran") {
      Izpisi(); alineje.clear();  // Izpišimo staro stran in začnimo novo,
      spremenjena = true; }  // ki jo bo treba tudi še izpisati.
    else if (s == "nova alineja") {
      // Dodajmo novo alinejo, razen če je na koncu trenutne strani že prazna alineja.
      if (alineje.empty() || ! alineje.back().empty())
        alineje.push_back({}); }
    else if (s == "pavza")
      Izpisi(); // Izpišimo trenutno stanje strani.
    else {
      // Če je stran še čisto prazna, dodajmo prvo alinejo.
      if (alineje.empty()) alineje.push_back({});
      // Dodajmo novi niz in si zapomnimo, da je stran spremenjena.
      alineje.back().push_back(s); spremenjena = true; }
  }
  Izpisi(); // Izpišimo še zadnjo stran.
  return 0;
}
```

2. Urnik

Za vsako kombinacijo dneva v tednu in ure v dnevu si lahko pripravimo množico učilnic, ki so takrat zasedene. Če ima teden na primer D dni, dan pa T ur, imamo lahko tabelo $D \times T$ elementov, od katerih vsak kaže na takšno množico zasedenih učilnic za tisto uro tistega dne. Recimo tej tabeli a. Na začetku naj bodo vse množice prazne, nato pa se sprehodimo po vhodnem seznamu in za vsak zapis $\langle p, d, u, t \rangle$ (ki nam pove, da na uro t dneva d poteka predmet p v učilnici u) dodamo u v množico $a[d, t]$. Spotoma si pripravimo še množico vseh učilnic sploh, recimo U.

Kasneje, ko pride poizvedba, ki sprašuje, katere učilnice so proste ob uri t dneva d, moramo vrniti razliko $U - a[d, t]$. Lahko gremo na primer v zanki po vseh učilnicah $u \in U$ in za vsako preverimo, ali je zasedena, torej ali je $u \in a[d, t]$. Lahko pa tudi že pred odgovarjanjem na poizvedbe gremo po vseh množicah v a in vsako predelamo v njen komplement, torej spremenimo $a[d, t]$ v $U - a[d, t]$; po tistem bodo množice v a ravno tisto, kar moramo vračati pri naših poizvedbah. Ta druga možnost je koristna, če so učilnice v povprečju bolj zasedene kot proste, ker je takrat bolje pri odgovarjanju na poizvedbo imeti že pripravljen seznam prostih učilnic (ki bo razmeroma kratek) kot pa iti po vseh učilnicah in za vsako preverjati, ali je zasedena ali ne.

Če števil D in T ne poznamo vnaprej, lahko a predstavimo tudi z razpršeno tabelo, v kateri so ključi pari (d, t), pripadajoča vrednost pa je seznam takrat zasedenih (ali prostih, kakor se pač odločimo) učilnic. Če pride poizvedba za tak par (d, t), ki ga v a sploh nimamo, pa lahko vrnemo množico U — če v vhodnem urniku ni bilo podatkov o tem, da bi bila t-to uro d-tega dne kakšna učilnica zasedena, potem so takrat vse učilnice proste.

Oglejmo si primer implementacije v pythonu:

```
class Urnik:
  __slots__= ["a", "U"]

  def __init__(self, seznam):
    # Pripravimo množice zasedenih učilnic za vsako kombinacijo dneva in ure
    # ter množico vseh učilnic sploh.
    self.a = {}; self.U = set()
    for (p, d, u, t) in seznam:
      # Če te kombinacije dneva in ure v a še nimamo, jo dodajmo.
      if (d, t) not in self.a: self.a[d, t] = set()
      self.a[d, t].add(u); self.U.add(u)

    # Množice zasedenih učilnic predelajmo v množice prostih.
    for d, t in self.a:
      self.a[d, t] = self.U − self.a[d, t]

  def ProsteUcilnice(self, d, t):
    # Vrnimo množico prostih učilnic za to kombinacijo dneva in ure;
    # če pa take nimamo, so tedaj proste vse učilnice.
    return self.a.get((d, t), self.U)
```

3. Pravokotnik iz kvadratov

Oglejmo si najprej lažjo različico naloge, pri kateri moramo uporabiti vseh n vhodnih kvadratov. Njihovo skupno ploščino označimo s $s = c_1^2 + c_2^2 + \ldots + c_n^2$. Krajšo stranico pravokotnika, ki ga iščemo, imenujmo a, daljšo pa $b = s/a$. Da bosta stranici celoštevilski, morata biti a in b seveda delitelja števila s. Da bo pravokotnik čim bolj kvadraten, mora biti b/a čim bližje 1; to je naprej enako $b/a = s/a^2$, torej mora biti a čim bližje $\sqrt{s}$ (ne sme pa ga preseči, kajti potem a ne bi več mogel biti krajša stranica pravokotnika, ampak kvečjemu daljša). Preprosta rešitev je torej ta, da v zanki preizkušamo vse manjše a od $\lfloor\sqrt{s}\rfloor$ navzdol, dokler ne najdemo takega, ki deli s; tisti je potem iskana rešitev (skupaj z drugo stranico $b = s/a$).

Še ena možnost je, da s najprej razcepimo na prafaktorje, npr. $s = \prod_i p_i^{r_i}$; s-jevi delitelji imajo potem iste prafaktorje p_i kot s, vendar mogoče z manjšo stopnjo od r_i. Vse delitelje dobimo torej tako, da na vse možne načine izberemo stopnje $q_i \in \{0, \ldots, r_i\}$ in za vsak izbor stopenj izračunamo $\prod_i p_i^{q_i}$. Za vsakega od njih lahko, ko ga izračunamo, pogledamo, ali je njegov kvadrat $\leq s$, in si med takimi zapomnimo največjega.

Razmislimo zdaj še o težji različici naloge, pri kateri si lahko izberemo dva ali več kvadratov. Načeloma je možnih izborov torej $2^n - n - 1$ (odšteli smo izbore z enim ali nobenim kvadratom); pri vsakem lahko določimo vsoto ploščin s in zanjo poženemo postopek, ki smo ga videli zgoraj pri lažji različici. Lahko se zgodi, da do istega s pridemo pri več izborih, zato je koristno, če si pripravimo množico vseh dosegljivih s-jev, da ne bomo potem istega s obravnavali po večkrat. Pri tem pazimo na to, da se nam v množico ne prikradejo izbori z enim samim kvadratom; spodnji postopek zato ločeno hrani s-je, ki smo jih dobili z izborom vsaj dveh kvadratov (množica S_2), in tiste, ki smo jih dobili samo z enim (množica S_1):

$S_1 := \{\}$; $S_2 := \{\}$;
for $i := 1$ **to** n:
  $T := \{s + c_i^2 \ : \ s \in S_1 \cup S_2\}$;
  dodaj vse $s \in T$ v S_2;

dodaj c_i^2 v S_1;

Za vsak $s \in S_2$ moramo zdaj pognati postopek, ki smo si ga ogledali pri lažji različici naloge. Če bomo s-je razbijali na prafaktorje, si lahko nekaj stvari, ki pridejo prav pri tem (npr. seznam praštevil do $\sqrt{s}$ za $s = \sum_{i=1}^{n} c_i^2$), pripravimo na začetku in jih potem uporabljamo pri vsakem s.

4. Seznama

Nalogo lahko rešujemo s požrešnim algoritmom in tudi dokaz pravilnosti gre po čisto takem kopitu, kot je običajno za požrešne algoritme. Rešitvi obeh različic naloge sta si zelo podobni:[22]

(a) Uredimo obe skupini akrobatov po teži in jih v tem vrstnem redu oštevilčimo, tako da imajo npr. akrobati z zelenimi hlačami mase $a_1 \leq a_2 \leq \ldots \leq a_n$, tisti z oranžnimi pa $b_1 \leq b_2 \leq \ldots \leq b_n$. Potem jih je najbolje razporediti v pare tako, da najlažji zeleni akrobat pride skupaj z najlažjim oranžnim, podobno drugi najlažji zeleni z drugim najlažjim oranžnim in tako naprej; torej v splošnem, da pride a_i v par z b_i.

Prepričajmo se, da je to res najboljši razpored. Pa recimo, da bi bil neki drugi razpored še boljši. Pri najlažjih nekaj akrobatih se morda ta boljši razpored še ujema z našim, prej ali slej pa mora nastopiti razlika; recimo, da v boljšem razporedu nastopajo pari $(a_1, b_1), \ldots, (a_{i-1}, b_{i-1})$, nato pa akrobat a_i ni v paru z b_i (kot pri našem razporedu), pač pa z b_j za neki $j > i$ (možnost $j < i$ odpade, saj so akrobati $b_1, \ldots, b_{i-1}$ že razporejeni v prej naštete pare). Akrobat b_i pa, ker ni v paru z a_i, mora biti v paru z nekim a_k za $k > i$ (spet $k < i$ odpade, ker so tudi akrobati $a_1, \ldots, a_{i-1}$ že razporejeni v prej naštete pare).

Omenjena para (a_i, b_j) in (a_k, b_i) prispevata k vsoti, s katero ocenjujemo razporede, člena $U := (a_i - b_j)^2 + (a_k - b_i)^2$. Če bi akrobata b_i in b_j zamenjali, tako da bi se omenjena para spremenila v (a_i, b_i) in (a_k, b_j), pa bi njun prispevek k oceni razporeda znašal $V := (a_i - b_i)^2 + (a_k - b_j)^2$. S tem se je torej ocena spremenila za $V - U = \ldots = 2(a_i - a_k)(b_j - b_i)$. Ker velja $i < k$, je $a_i \leq a_k$, zato je $a_i - a_k \leq 0$; in ker velja tudi $i < j$, je $b_i \leq b_j$, zato je $b_j - b_i \geq 0$. Naša razlika $V - U$ je torej ≤ 0, torej je se je ocena razporeda s to spremembo zmanjšala ali ostala enaka; in ker hočemo oceno minimizirati, je novi razpored vsaj tako dober kot stari; se pa zdaj ujema z našim urejenim razporedom v enem paru več kot stari razpored.

Tako lahko tisti domnevno boljši razpored, čigar obstoj smo predpostavili na začetku, korak za korakom predelamo v naš urejeni razpored, ne da bi se njegova ocena pri tem kdaj poslabšala. Torej je tudi naš urejeni razpored najboljši.

(b) Pri tej različici akrobati še niso razdeljeni na dve skupini po n, zato uredimo vseh $2n$ akrobatov naraščajoče po teži in jih v tem vrstnem redu oštevilčimo: $a_1 \leq a_2 \leq \ldots \leq a_{2n}$. Najboljši razpored v pare je spet po vrsti: (a_1, a_2), (a_3, a_4) in tako naprej do (a_{2n-1}, a_{2n}).

Dokaz, da je to res, je čisto podoben tistemu kot pri (a). Recimo, da bi obstajal neki še boljši razpored od našega; v prvih nekaj parih se mogoče ujema z našim, prej ali slej pa mora nastopiti prvo neujemanje. Recimo, da se to zgodi pri i-tem

[22] Še eno zelo podobno nalogo smo pred leti že imeli: 2001.U.4, pri kateri je bilo treba razdeliti $2n$ števil v n parov (a_i, b_i) in pri tem minimizirati vsoto zmnožkov $\sum_i a_i b_i$ (gl. *Rešene naloge s srednješolskih računalniških tekmovanj, 1988–2004*, str. 687–8 in rešitev na str. 709–10).

paru, kjer akrobat a_{2i-1} ni v paru z a_{2i} kot pri nas, pač pa z a_j za neki $j > 2i$; in akrobat a_{2i}, ker ni v paru z a_{2i-1}, je z a_k za neki $k > 2i$. Ta dva para, (a_{2i-1}, a_j) in (a_{2i}, a_k), prispevata k oceni razporeda vsoto $U := (a_{2i-1} - a_j)^2 + (a_{2i} - a_k)^2$.

Če zdaj te štiri akrobate prerazporedimo v para (a_{2i-1}, a_{2i}) in (a_j, a_k), bo njun prispevek k oceni znašal $V := (a_{2i-1} - a_{2i})^2 + (a_j - a_k)^2$. Sprememba je $V - U = \ldots = 2(a_{2i-1} - a_k)(a_j - a_{2i})$, kar je ≤ 0, ker je $a_{2i-1} \leq a_k$ (zaradi $k > 2i - 1$) in $a_j \geq a_{2i}$ (zaradi $j > 2i$). Po tej spremembi torej razpored ni nič slabši, se pa z našim urejenim razporedom ujema v enem paru več kot prej. Tako lahko najboljši razpored korak za korakom spremenimo v našega, ne da bi se kdaj poslabšal, torej je tudi naš razpored najboljši.

5. Napredovanje števil

Stanje mreže lahko predstavimo s tabelo 6×6 celih števil (v spodnji rešitvi je to tip Mreza). Vrednost -1 nam bo pomenila prazno polje; na začetku torej inicializirajmo vse elemente tabele na -1.

Nato v zanki pregledujmo poteze. Pri vsaki preverimo, če sta koordinati res znotraj mreže (predpostavimo, da nastopajo v vhodnih podatkih številke vrstic in stolpcev od 1 do 6), če je novo število od 1 do 6 in če je polje, kamor ga želimo vpisati, res še prazno. Če je vse to v redu, vpišimo novo število v tabelo, potem pa moramo pogledati, ali pripada kakšni skupini vsaj treh polj z enako vrednostjo. Če ji ne, smo s to potezo končali; če pa polje pripada kakšni skupini, moramo poskrbeti za napredovanje skupine, po njem pa se lahko zgodi, da polje zdaj pripada neki drugi skupini (to smo videli tudi v primeru na koncu besedila naloge). Zato moramo preverjanje in napredovanje skupin izvajati v zanki, ki se ustavi šele, ko trenutno polje ne pripada nobeni skupini.

Ko pridemo do konca seznama potez, moramo še preveriti, če je mreža zdaj polna; spotoma lahko tudi štejemo ničle v njej. To število ničel na koncu vrnemo, v primeru napake pa vrnemo -1. Oglejmo si implementacijo te rešitve v C++:

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

struct Poteza { int x, y, n; };
typedef int Mreza[6][6];

int NapredovanjeStevil(const vector<Poteza>& poteze, Mreza m)
{
  // Na začetku je mreža prazna.
  for (int y = 0; y < 6; ++y) for (int x = 0; x < 6; ++x) m[y][x] = −1;
  // Odsimulirajmo vse poteze.
  for (const auto &poteza : poteze)
  {
    int x = poteza.x − 1, y = poteza.y − 1;
    // Ali je poteza sploh veljavna?
    if (poteza.n < 1 || poteza.n > 6) return −1;
    if (x < 0 || y < 0 || x > 5 || y > 5) return −1;
    if (m[y][x] >= 0) return −1; // Polje ni bilo prazno.
    // Vpišimo novo število v mrežo.
    m[y][x] = poteza.n;
```

```
      // Poglejmo, ali je nastala skupina; če da, poskrbimo za napredovanje
      // števil v njej. Po tistem lahko nastane nova skupina in tako naprej.
      while (ObdelajSkupino(x, y, m)) ;
   }
   // Na koncu mora biti mreža polna. Preštejmo tudi ničle v njej.
   int nicle = 0;
   for (int y = 0; y < 6; ++y) for (int x = 0; x < 6; ++x)
      if (m[y][x] < 0) return −1;
      else if (m[y][x] == 0) ++nicle;
   return nicle;
}
```

Manjka nam še podprogram ObdelajSkupino, čigar naloga je, da preišče skupino polj, ki ji pripada dano začetno polje (x_0, y_0). Če se izkaže, da je to res skupina vsaj treh polj, mora poskrbeti za napredovanje polj v njej; če skupina napreduje na kaj drugega kot 0, pa mora njena polja (razen začetnega) tudi pobrisati. Za pregled polj v skupini lahko uporabimo iskanje v širino: na začetek dodamo v seznam s začetno polje, nato pa na vsakem koraku pogledamo naslednji element seznama in dodamo v seznam njegove sosede, če je na njih enaka številka. Spotoma, ko dodajamo polja v seznam, jih na mreži brišemo; to je koristno predvsem zato, da ne bomo istega polja dodali v seznam po večkrat. Če bomo na koncu ugotovili, da skupine ne bi smeli pobrisati, bomo pobrisana števila vpisali nazaj. Podprogram vrne logično vrednost, ki pove, ali je polje (x_0, y_0) pripadalo kakšni skupini in napredovalo ali ne (če je, to pomeni, da ga bo treba pognati še enkrat, ker je mogoče, da isto polje zdaj pripada neki drugi skupini in bo napredovala tudi ta).

```
bool ObdelajSkupino(int x0, int y0, Mreza m)
{
   const int DX[] = { −1, 1, 0, 0 }, DY[] = { 0, 0, −1, 1 };
   // Z iskanjem v širino bomo pregledali skupino, ki ji pripada polje (x0, y0).
   int n = m[y0][x0]; // Vrednost polj v skupini.
   // Polja z vrednostjo 0 ne morejo več tvoriti skupin in napredovati.
   if (n == 0) return false;
   // Seznam polj skupine nastaja v „s“. Ko polja dodajamo v vrsto, jih v
   // „m“ postavljamo na −1, da se ne bomo zaciklali.
   vector<int> s; s.push_back(x0 + y0 * 6); m[y0][x0] = −1;
   for (int glava = 0; glava < s.size(); ++glava)
   {
      // Preberimo naslednje polje (x, y) iz „s“ in dodajmo v „s“ njegove sosede,
      // če imajo tudi oni vrednost „n“.
      int x = s[glava] % 6, y = s[glava] / 6;
      for (int smer = 0; smer < 4; ++smer)
      {
         int xx = x + DX[smer], yy = y + DY[smer];
         if (xx < 0 || xx > 5 || yy < 0 || yy > 5) continue;
         if (m[yy][xx] != n) continue;
         s.push_back(xx + yy * 6); m[yy][xx] = −1;
      }
   }
   // Če je to res skupina vsaj treh polj, bo napredovala.
   bool jeSkupina = (s.size() > 2);
   if (jeSkupina) n = (n + 1) % 7;
```

```
  // V „m“ smo ob pregledovanju pobrisali polja te skupine. Če to v resnici ni
  // bila skupina ali pa je bila in je napredovala v 0, moramo števila vpisati nazaj.
  if (n == 0 || ! jeSkupina) for (int p : s) m[p / 6][p % 6] = n;

  // Na začetnem polju (x0, y0), kamor smo vpisali število v trenutni potezi,
  // pa števila v nobenem primeru ne smemo pobrisati.
  m[y0][x0] = n;
  return jeSkupina;
}
```

6. Študentski servis

Za začetek je koristno urediti vhodni seznam opravil naraščajoče po času zaključka. V tem vrstnem redu jih zdaj oštevilčimo; naj bodo naj bodo z_i, k_i in p_i začetek, konec in plačilo pri i-tem opravilu. Skupno število opravil označimo z n.

Recimo zdaj, da se Metka odloči prevzeti zadnje opravilo; ker se to začne ob času z_n, to pomeni, da ne more prevzeti nobenega takega opravila, ki se konča kasneje kot ob z_n. Ker imamo opravila urejena po času konca, ustreza temu pogoju prvih nekaj opravil (koliko, je odvisno od z_n) v našem seznamu, ostala pa ne. Metki torej v nadaljevanju preostane le še to, da poišče čim boljši izbor opravil izmed teh prvih nekaj.

Po drugi strani, če bi se Metka odločila, da zadnjega opravila ne prevzame, ji potem ostane le še, da poišče najboljši izbor izmed prvih $n - 1$ opravil.

V obeh primerih smo torej naš problem na n opravilih prevedli na malo manjši podproblem, pri katerem gledamo le prvih nekaj opravil namesto vseh. Pri reševanju vsakega takega podproblema bi razmišljali na enak način. Naj bo torej $f(i)$ največji znesek, ki ga Metka lahko zasluži, če izbira opravila le izmed prvih i. Dosedanji razmislek nam je pokazal, da velja naslednja zveza:

$$f(i) = \max\{p_t + f(r(i)), f(i-1)\},$$

pri čemer je $r(i)$ indeks zadnjega opravila, ki se konča najkasneje ob času z_i, torej

$$r(i) = \max\{j \ : \ k_j \le z_i\}.$$

Če ni sploh nobenega takega opravila, si mislimo $r(i) = 0$. Poleg tega velja tudi $f(0) = 0$ (če ni na voljo nobenih opravil, ne more Metka zaslužiti ničesar).

Vidimo lahko, da pri izračunu vrednosti $f(i)$ potrebujemo vrednost $f(i-1)$ in še $f(r(i))$, pri čemer je $r(i) < i$. Koristno je torej vrednosti funkcije f računati po naraščajočih i in jih hraniti v neki tabeli; tako bomo imeli vedno pri roki vrednosti, ki jih potrebujemo.

Razmisliti moramo še o tem, kako računati vrednosti $r(i)$, torej kako za vsako opravilo i ugotoviti, koliko opravil se konča najkasneje ob z_i. Ker imamo opravila že urejena po času konca, lahko po tem seznamu iščemo z bisekcijo in tako v $O(\log n)$ časa poiščemo zadnje opravilo s koncem $\le z_i$. Za vse i lahko tako dobimo $r(i)$ v skupnem času $O(n \log n)$; to je čisto sprejemljiva cena, saj nam bo $O(n \log n)$ časa vzelo tudi urejanje opravil po času konca.

Še ena možnost za izračun vrednosti $r(i)$ pa je, da na začetku vržemo čase začetkov in čase koncev v en sam seznam in ga uredimo. Potem se sprehodimo po seznamu in hranimo največji j, za katerega smo doslej že videli k_j; vsakič pa, ko

pridemo do nekega časa začetka, na primer z_i, si lahko takrat zapomnimo, da je $r(i) = j$ (za tisti j, ki ga imamo trenutno shranjenega). Pri tem pazimo še na to, da če se ob istem času nekaj opravil začne in nekaj konča, morajo priti konci v našem urejenem seznamu pred začetki, sicer ta postopek ne bo dajal pravilnih rezultatov.

Oglejmo si primer implementacije rešitve z bisekcijo (v C++ si lahko pomagamo s funkcijo upper_bound iz standardne knjižnice):

```
#include <vector>
#include <algorithm>
using namespace std;
struct Delo { int z, k, p; }; // začetek, konec, plačilo

int NajvecjiZasluzek(vector<Delo> dela)
{
  // Uredimo dela po času konca.
  auto comp = [ ] (const auto &x, const auto &y) { return x.k < y.k; };
  sort(dela.begin(), dela.end(), comp);

  // f[i] = največji zaslužek, če se omejimo na prvih i del.
  int n = dela.size(); vector<int> f(n + 1); f[0] = 0;
  for (int i = 0; i < n; ++i) {

    // Naj bo ri število del, ki se končajo najkasneje takrat, ko se delo i začne.
    int ri = upper_bound(dela.begin(), dela.end(),
                         Delo{0, dela[i].z, 0}, comp) − dela.begin();

    // Izračunajmo najboljšo rešitev za prvih i + 1 del (od 0 do i).
    f[i + 1] = max(f[i], f[ri] + dela[i].p); }

  return f[n];
}
```

7. Pandemija

Za vsako mesto bomo vzdrževali strukturo z naslednjimi podatki: trenutna stopnja vsake epidemije v tem mestu; kako dolgo je mesto že na tej stopnji (za vsako epidemijo posebej); in seznam sosedov, torej mest, ki so z opazovanim mestom neposredno povezana prek ene od povezav. Te sezname pripravimo ob branju vhodnih podatkov: ko na vhodu dobimo povezavo med u in v, moramo dodati u na seznam v-jevih sosedov in v na seznam u-jevih sosedov.

V vsakem časovnem intervalu simulacije imamo zanko po vseh štirih epidemijah, saj so te med seboj neodvisne in pravila širjenja so pri vseh enaka. Najprej, še preden začnemo stopnje spreminjati, izračunamo za vsako mesto vsoto stopenj sosedov. Pri vsakem mestu potem trajanje trenutne stopnje povečamo za 1, nato pa začnemo gledati, ali je treba mestu stopnjo zvišati ali znižati; če da, resetiramo podatek o trajanju nazaj na 0. Spotoma še preverjamo, če je nova stopnja kakšnega mesta nižja od 4; v tem primeru vemo, da zaradi te epidemije trenutno še ne bo izumrtja. Če pa pri kakšni epidemiji v kakšnem koraku zaznamo izumrtje, simulacijo takoj prekinemo.

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

int main()
```

```
{
  // Preberimo število mest in povezav.
  int m, p; cin >> m >> p;

  // Pripravimo si vektor s podatki o mestih.
  struct Mesto { int stopnja[4], vsota, trajanje[4] = {}; vector<int> sosedje; };
  vector<Mesto> mesta(m);

  // Preberimo začetne stopnje epidemije v vseh mestih.
  for (auto &M : mesta) for (int b = 0; b < 4; ++b) cin >> M.stopnja[b];

  // Preberimo povezave in pripravimo sezname sosedov vsakega mesta.
  for (int i = 0; i < p; ++i) {
    int u, v; cin >> u >> v; --u; --v;
    mesta[u].sosedje.push_back(v); mesta[v].sosedje.push_back(u); }

  // Simulirajmo dogajanje.
  const int minStopnja = 1, maxStopnja = 4, maxCas = 100;
  int casIzumrtja = -1;
  for (int cas = 1; cas <= maxCas && casIzumrtja < 0; ++cas)
    for (int b = 0; b < 4 && casIzumrtja < 0; ++b)
    {
      // Za vsako mesto izračunajmo vsoto sosedov na podlagi starih stopenj.
      for (auto &M : mesta) {
        M.vsota = 0; for (int v : M.sosedje) M.vsota += mesta[v].stopnja[b]; }

      // Poglejmo, kako se stopnje spremenijo.
      casIzumrtja = cas;
      for (auto &M : mesta) {
        ++M.trajanje[b]; // Tako dolgo smo na trenutni stopnji.

        // Ali se mora stopnja zvišati?
        if (M.vsota > 6 && M.stopnja[b] < maxStopnja)                    (⋆)
          ++M.stopnja[b], M.trajanje[b] = 0;

        // Ali se mora stopnja znižati?
        else if (M.trajanje[b] >= 3 && M.stopnja[b] > minStopnja)
          --M.stopnja[b], M.trajanje[b] = 0;

        // Če kakšno mesto ni v najvišji stopnji, izumrtja še ne bo.
        if (M.stopnja[b] < maxStopnja) casIzumrtja = -1; }
    }

  // Izpišimo rezultat.
  if (casIzumrtja < 0) cout << "SE SO ZIVI" << endl;
  else cout << "Izumrli po " << casIzumrtja << " korakih." << endl;
  return 0; }
```

Naloga pravi, naj se stopnja zniža za 1, če se prej tri časovne intervale ni zvišala. Vprašanje je, ali velja to tudi v primeru, če se ni zvišala le zato, ker je bila epidemija tam že na najvišji stopnji. Če bi hoteli dodati to kot izjemo, bi morali stavek (⋆) spremeniti v nekaj takega:

```
  if (M.vsota > 6) { M.trajanje[b] = 0; if (M.stopnja[b] < maxStopnja) ++M.stopnja[b]; }
```

8. Tretji tir

Naloga pravi, da je nadmorska višina trase vedno celo število od 0 do $v = 500$. Če to združimo z dejstvom, da si traso v tlorisu predstavljamo kot zaporedje sosednjih kvadratov na karirasti mreži $w \times h$, lahko zaključimo, da se vlak pravzaprav premika po trodimenzionalni karirasti (oz. kockasti) mreži $w \times h \times v$, pri čemer se lahko

vlak iz polja (x, y, z) premakne v sosednja polja $(x \pm 1, y, z')$ in $(x, y \pm 1, z')$ za $z' \in \{z-1, z, z+1\}$. Cena tega, da trasa obišče polje (x, y, z), je odvisna od razlike med z in nadmorsko višino površja tam, torej h_{xy}. Naša naloga je znotraj teh omejitev najti najcenejšo pot od začetnega polja (x_d, y_d, h_{x_d,y_d}) do končnega polja (x_k, y_k, h_{x_k,y_k}).

Pomagamo si lahko z Dijkstrovim algoritmom za iskanje najkrajših poti po grafih. Vzdrževali bomo množico polj R, za katera že poznamo najcenejšo pot od začetnega polja do njih, in množico Q tistih polj, ki še niso v R, so pa dosegljiva iz kakšnega polja v R v enem koraku in zanje tudi poznamo najkrajšo tako pot od začetnega polja do njih, ki se ves čas razen v zadnjem koraku giblje po poljih v R. Pokazati je mogoče, da za tisto polje v Q, pri katerem je ta pot najkrajša (med vsemi polji, ki so trenutno v Q), velja, da je tista pot že tudi najkrajša pot sploh od začetnega polja do njega, zato lahko to polje premaknemo iz Q v R, nato pa v Q dodamo njegove sosede (če še niso v R). Ta postopek pregleduje polja po naraščajoči ceni poti od začetnega; sčasoma se v R znajde tudi končno polje in takrat lahko postopek končamo.

Pri implementaciji tega algoritma lahko za R uporabimo razpršeno tabelo, v kateri so ključi koordinate polj, pripadajoča vrednost pri posameznem ključu pa je cena najkrajše poti do tistega polja (koristno je zraven hraniti še smer, iz katere smo v to polje prišli — tako bomo lahko na koncu rekonstruirali potek trase od začetnega polja do končnega). Za Q pa je koristno uporabiti kopico, v kateri so polja organizirana po ceni najcenejše doslej znane poti do njih (tista z cenejšo potjo so bolj pri vrhu kopice).

Koristna je še naslednja izboljšava. Za polje $u \in Q$ naj bo $f(u)$ najcenejša doslej znana pot od začetnega polja do njega, $g(u)$ pa najcenejša pot od u do končnega polja. Doslej smo rekli, da iz Q vsakič vzamemo polje u z najmanjšo $f(u)$. Posledica tega je, da algoritem jemlje polja po naraščajoči oddaljenosti (ceni poti) od začetnega in da bo, preden bo iz Q vzel končno polje, pregledal že vsa druga polja, ki so od začetnega oddaljena manj kot končno. Takih polj pa utegne biti zelo veliko. Če bi namesto tega vzeli iz Q vsakič polje z najmanjšo vsoto $f(u) + g(u)$, bi se algoritem osredotočil na polja, ki res ležijo na najkrajši poti od začetnega do končnega, druga polja pa bi prišla na vrsto kasneje (oz. sploh ne bi, ker postopek končamo, čim vzamemo iz Q končno polje). Težava seveda je, da vrednosti $g(u)$ (torej najcenejše poti od u do končnega polja) ne poznamo. Pokazati pa je mogoče, da če namesto $g(u)$ vzamemo neki približek oz. hevristiko, recimo $\tilde{g}(u)$, za katerega velja $0 \le \tilde{g}(u) \le g(u)$ (torej naš približek lahko podceni ceno poti od u do končnega polja, ne sme pa je preceniti), bo algoritem še vedno dajal pravilne rezultate in bo mogoče vendarle hitrejši od prvotnega, ki je vedno jemal iz Q polje z najmanjšo $f(u)$. Takšna različica Dijkstrovega algoritma, izboljšana s hevristiko $\tilde{g}(u)$, se imenuje A*.

V našem primeru lahko preprosto hevristiko $\tilde{g}(u)$ računamo takole: če mora vlak priti od $u = (x_u, y_u, z_u)$ do končnega polja (x_k, y_k, z_k), mora gotovo narediti vsaj $|x_u - x_k| + |y_u - y_k|$ korakov, saj lahko v enem koraku spremeni bodisi x-koordinato za 1 bodisi y-koordinato za 1, ne pa obeh. Gotovo mora tudi narediti vsaj $|z_u - z_k|$ korakov, saj lahko v enem koraku spremeni z-koordinato največ za ± 1. Vsak korak pa stane vsaj 1 DENEN; tako je torej $\tilde{g}(u) = \max\{|x_u - x_k| + |y_u - y_k|, |z_u - z_k|\}$ gotovo manjša ali enaka od prave $g(u)$, torej cene najcenejše poti od u do končnega

polja. Pri naših poskusih z mrežami do $w = h = 500$ je ta hevristika v povprečju pospešila delovanje algoritma za več desetkrat, pri čemer je bilo na večjih primerih od nje še več koristi kot na manjših.

```
#include <iostream>
#include <queue>
#include <utility>
#include <vector>
#include <unordered_map>
using namespace std;

const int DX[] = { −1, 1, 0, 0 }, DY[] = { 0, 0, −1, 1 }, MinZ = 0, MaxZ = 500;

// Vhodni podatki.
int w, h, xd, yd, zd, xk, yk, zk;
vector<int> relief;

// Struktura, ki predstavlja koordinate polja. operator == in razred hash<Polje>
// potrebujemo, da bomo lahko Polje uporabljali kot ključ v razpršeni tabeli (unordered_set).
struct Polje { int x, y, z;
  int Hevristika() { return max(abs(x − xk) + abs(y − yk), abs(z − zk)); } // g̃(u)
  bool operator ==(const Polje& p) const { return x == p.x && y == p.y && z == p.z; } };
template<> struct hash<Polje> {
  size_t operator ()(const Polje& p) const { return ((p.z * h) + p.y) * w + p.x; } };

// Cena poti do nekega polja in smer, iz katere smo prišli vanj.
struct CenaSmer { int c, s; };

// Polje s ceno f(u) (in smerjo vstopa) in hevristično oceno f(u) + g̃(u).
struct PoljeC : Polje, CenaSmer { int ch = c + Hevristika();
  // Primerjalni operator poskrbi, da bodo v kopici na vrhu polja z najmanjšo f(u) + g̃(u).
  bool operator < (const PoljeC &p) const { return ch > p.ch; }; };

int Resi(vector<Polje> &pot) // Najcenejšo pot zapiše v „pot“ in vrne njeno ceno.
{
  zd = relief[yd * w + xd]; zk = relief[yk * w + xk];
  Polje pd { xd, yd, zd }, pk { xk, yk, zk }; // Začetno in končno polje.

  // R = polja, do katerih že poznamo najcenejšo pot. Q = polja, dosegljiva
  // iz R v enem koraku; zanje poznamo najcenejšo tako pot, ki gre ves čas po R,
  // razen v zadnjem koraku. Na začetku je v Q le začetno polje, R pa je prazna.
  unordered_map<Polje, CenaSmer> R;
  priority_queue<PoljeC> Q; Q.push({pd, 1, 0});

  // Pregledujmo prostor, dokler ne pregledamo vsega ali ne dosežemo končnega polja.
  while (! Q.empty())
  {
    // Polje z najcenejšo potjo bomo iz Q premaknili v R. Isto polje se lahko v Q
    // znajde večkrat, če ga dosežemo iz več sosedov; vse pojavitve razen prve ignoriramo.
    PoljeC p = Q.top(); Q.pop();
    auto [pi, jeNov] = R.emplace(p, p); if (! jeNov) continue;
    if (p == pk) break; // Ko pride v R končno polje, lahko končamo.

    // Preglejmo vse sosede polja p.
    for (int d = 0; d < 4; ++d) for (int dz = −1; dz <= 1; ++dz)
    {
      int xx = p.x + DX[d], yy = p.y + DY[d], zz = p.z + dz;

      // Ali je ta sosed sploh v mreži?
      if (xx < 0 || yy < 0 || zz < MinZ || xx >= w || yy >= h || zz > MaxZ) continue;

      // Kakšna je cena proge v tem sosednjem polju?
```

```
            int v = zz − relief[yy * w + xx]; // višina nad/pod površjem
            int cena = p.c + (v <= −5 ? 7 : v >= 5 ? 1 + ((v + 9) / 10) : 1);
            // Dodajmo to sosednje polje v Q.
            Q.push({xx, yy, zz, cena, 4 * (dz + 1) + d});
        }
    }
    // Zdaj lahko rekonstruiramo najcenejšo pot (od konca proti začetku).
    pot.clear(); pot.push_back(pk);
    while (! (pot.back() == pd))
    {
        // Zdaj smo v polju p, v katero smo prišli iz smeri s.
        Polje p = pot.back(); int s = R[p].s;
        int dz = (s / 4) − 1, d = s % 4;
        // Premaknimo se v obratni smeri, da dobimo prejšnje polje na poti.
        p.x −= DX[d]; p.y −= DY[d]; p.z −= dz;
        pot.push_back(p);
    }
    // Pot obrnimo v pravi vrstni red in vrnimo njeno skupno ceno.
    reverse(pot.begin(), pot.end()); return R[pk].c;
}

int main()
{
    // Preberimo velikost mreže in koordinate začetka in konca.
    cin >> w >> h >> xd >> yd >> xk >> yk;
    // Preberimo tabelo višin (relief).
    relief.resize(w * h);
    for (int y = 0; y < h; ++y) for (int x = 0; x < w; ++x) cin >> relief[y * w + x];
    // Poiščimo najcenejšo traso.
    vector<Polje> pot; int cena = Resi(pot);
    // Izpišimo rezultate.
    cout << cena << " " << pot.size() << endl; // cena poti in število korakov
    for (const Polje &p : pot) cout << p.x << " " << p.y << " " << p.z << endl;
    return 0;
}
```

9. Tabela števil

Ker naloga pravi, da ni treba minimizirati števila sprememb, si lahko predstavljamo, da na začetku poberemo vsa števila iz tabele in jih potem zložimo nazaj v prazno tabelo v takem vrstnem redu, kot si ga bomo zamislili. Vse, kar si moramo od prvotnega stanja tabele zapomniti, je to, kolikokrat se je pojavljalo posamezno število v njej.

Recimo v splošnem, da ima tabela n stolpcev in h vrstic in da se v njej pojavlja skupno največ h različnih števil (vsa pa so naravna števila z območja od 1 do n). Preštejmo za vsako število od 1 do n, kolikokrat se pojavi v tabeli. Ker imamo največ h različnih števil in skupno $h \cdot n$ pojavitev, je nemogoče, da bi se vsako število pojavilo več kot n-krat. Naj bo torej a neko tako število, ki se pojavi kvečjemu n-krat. Vse pojavitve števila a zložimo v prvo vrstico.

Če se a pojavi natanko n-krat, smo zdaj prvo vrstico ravno zapolnili in ob tem porabili vse a-je, tako da nam ostane problem s $h - 1$ vrsticami in $h - 1$ različnimi

števili, ki ga rešujemo po enakem postopku (le da bo pač h za 1 manjši kot doslej).

Če pa se a pojavi manj kot n-krat, mora obstajati neko drugo število b, ki se pojavi več kot n-krat (saj se drugače ne bi moglo nabrati za $h \cdot w$ pojavitev vseh števil skupaj). Ker prve vrstice z a-ji še nismo čisto zapolnili, dodajmo torej v preostanek vrstice b-je (ki jih pri tem zagotovo še ne bo zmanjkalo). Tako nam spet ostane problem s $h - 1$ vrsticami in $h - 1$ različnimi števili (ker smo porabili vse a-je, ne pa še vseh b-jev).

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

void Preuredi(int n, vector<int> &T)
{
  // Preštejmo pojavitve števil v T.
  vector<int> f(n + 1, 0);
  for (int t : T) ++f[t];
  // Na novo zapolnimo celotno tabelo.
  for (int y = 0; y < n; ++y)
  {
    // Poiščimo najredkejše število a in najpogostejše b.
    int a = −1, b = −1;
    for (int t = 1; t <= n; ++t) if (f[t] > 0) {
      if (a < 0 || f[t] < f[a]) a = t;
      if (b < 0 || f[t] > f[b]) b = t; }
    // Naslednjo vrstico zapolnimo z a-ji, ko jih zmanjka, pa z b-ji.
    for (int x = 0; x < n; ++x) {
      int t = (f[a] > 0) ? a : b;
      T[y * n + x] = t; −−f[t]; }
  }
}
```

10. Film

Omrežje z vozlišči in povezavami med njimi si lahko predstavljamo kot neusmerjen graf. Problem poti z največjo kapaciteto je zelo podoben bolj znanemu problemu najkrajše poti, pri katerem ima vsaka povezava znano dolžino, mi pa iščemo pot z najmanjšo vsoto dolžin povezav. Pri našem problemu je stvar podobna, le da ima vsaka povezava kapaciteto namesto dolžine, iščemo pa pot, na kateri je minimalna kapaciteta po vseh povezavah na poti čim večja. Povezava z minimalno kapaciteto je namreč tista, ki predstavlja „ozko grlo“ in s tem določa kapaciteto celotne poti. (Kot zanimivost omenimo, da če bi naloga dovoljevala, da podatke pošiljamo vzporedno po več poteh hkrati, bi imeli problem maksimalnega pretoka po grafu, take pa se potem rešuje z drugimi algoritmi.)

Načeloma lahko torej katerikoli algoritem za iskanje najkrajših poti po grafu prilagodimo tako, da kjer prvotni algoritem sešteva dolžine povezav, bo moral naš spremenjeni algoritem računati minimum njihovih kapacitet; in kjer prvotni algoritem minimizira dolžino poti, mora naš algoritem maksimizirati njeno kapaciteto. Oglejmo si na primer tako predelano obliko Dijkstrovega algoritma (s katerim smo se v letošnjem biltenu že srečali — gl. str. 166 — in se zato vanj tu ne bomo še bolj poglabljali):

vhod: vozlišča V, povezave E, njihove kapacitete $c(u, v)$
začetno vozlišče s, ciljno vozlišče t;

za vsako $u \in V$: $c[u] := 0$, $p[u] :=$ NIL;
$c[s] := \infty$; $Q :=$ prazna kopica; dodaj s v Q;
while Q ni prazna:
  pobriši iz Q tisto vozlišče (recimo mu u), ki ima največjo $c[u]$;
  (* *Tu vemo, da je $c[u]$ kapaciteta najboljše poti od s to u sploh.* *)
  if $u = t$ **then break**;
  za vsako povezavo (u, v) s krajiščem v u:
    $k := \min\{c[u], c(u, v)\}$;
    if $k \leq c[v]$ **then continue**;
    $p[v] := u$;
    if $c[v] = 0$:
      $c[v] := k$; dodaj v v kopico Q;
    else:
      $c[v] := k$; premakni v gor po kopici, kot zahteva njegova
        nova (večja) vrednost $c[v]$;
pot := prazen seznam; $u := t$; dodaj t v *pot*;
while $u \neq s$:
  $u := p[u]$; dodaj u na začetek seznama *pot*

Za vsako vozlišče u nam torej $c[u]$ pove najnižjo kapaciteto na najboljši doslej znani poti od s do t, pri čemer je $p[u]$ neposredni predhodnik u-ja na tej poti. Vozlišča, za katera še ne poznamo nujno najboljše poti, pač pa jo poznamo za kakšnega njihovega soseda, hranimo v kopici Q, ki mora biti organizirana tako, da so više v njej elementi z večjo $c[u]$. Ko pregledujemo u-jeve sosede v, upoštevamo, da lahko pot od s do u podaljšamo s korakom $u \to v$; kapaciteta take poti je $k = \min\{c[u], c(u, v)\}$; če do v še ne poznamo boljše poti, si to novo pot zapomnimo; pri tem v dodamo v kopico (če ga še ni bilo tam) oz. popravimo njegov položaj v kopici (ker se mu je povečala $c[v]$). Na koncu tega postopka imamo v $c[t]$ kapaciteto najboljše poti od s do t (če pa bi se slučajno izkazalo, da taka pot ne obstaja, bo $c[t] = 0$), s pomočjo tabele p pa lahko rekonstruiramo potek poti.

Razmislimo zdaj še o težji različici naloge, pri kateri smemo na poti največ enkrat uporabiti kakšno od povezav s kapaciteto C ali več. V tem primeru moramo pri tvorbi poti naš položaj opisati s parom (u, b), pri čemer u pove, v katerem vozlišču se trenutno nahajamo, b pa, ali smo že uporabili kakšno povezavo s kapaciteto vsaj C (to predstavimo z $b = 1$) ali ne ($b = 0$). Predstavljamo si lahko, da smo naredili nov graf, ki ga sestavljata dve rahlo spremenjeni kopiji prvotnega: če je imel prvotni graf vozlišče u, ima novi graf dve vozlišči $(u, 0)$ in $(u, 1)$; če je imel prvotni graf povezavo $u \to v$ s kapaciteto vsaj C, ima novi graf povezavo $(u, 0) \to (v, 1)$ z enako kapaciteto; in če je imel prvotni graf povezavo $u \to v$ s kapaciteto manj kot C, ima novi graf povezavi $(u, 0) \to (v, 0)$ in $(u, 1) \to (v, 1)$ z enako kapaciteto. Če smo v prvotnem grafu iskali najboljšo pot (torej tako z največjo kapaciteto) od s do t, moramo v novem iskati najboljšo pot od $(s, 0)$ do $(t, 0)$ ali $(t, 1)$ (torej pravzaprav dve najboljši poti, vrnemo pa boljšo izmed njiju, ker nam na koncu ni pomembno, ali smo uporabili kakšno povezavo s kapaciteto vsaj C ali ne).

11. Kodiranje besedila

Pri kodiranju lahko beremo vhodno besedilo znak po znak; preden se odločimo, kaj narediti s trenutnim znakom, pa poglejmo še naslednjega. Če sta oba mali črki, lahko izračunamo, v kaj bi se morala zakodirati (namreč v veliko različico druge črke). Če je ta koda še prosta, jo lahko zdaj dodelimo temu paru črk in ga zakodiramo; prav tako jo uporabimo tudi, če je bila že od prej dodeljena prav temu paru črk (ker smo ga npr. videli že nekoč prej v vhodnem besedilu). Če kodiranje iz kakršnega koli razloga ni uspelo (npr. ker trenutni in naslednji znak nista oba mali črki ali pa ker je bila koda že zasedena itd.), pa izpišemo trenutni znak brez sprememb.

Podatke o tem, katere kode smo že uporabili in za kaj, bomo hranili v tabeli 26 znakov; če na primer vpeljemo kodo `ce` $\rightarrow$ `E`, bomo v peti element tabele (ker je `E` peta črka abecede) vpisali `c` (prvo črko para, ki ga predstavlja ta koda; druga črka je le mala različica kode in je ni treba posebej shranjevati). Spodnja rešitev pričakuje tabelo, v katero bo shranjevala podatke o kodah, kar kot parameter, saj so ti podatki koristni tudi za klicatelja (potreboval jih bo za dekodiranje).

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

void Kodiraj(istream &is, ostream &os, char kode[26])
{
  // Na začetku so vse kode še proste.
  for (int c = 0; c < 26; ++c) kode[c] = '.';

  while (true)
  {
    // Preberimo trenutni znak.
    int c = is.get(); if (! is) break;

    // Če sta ta in naslednji znak črki, ju poskusimo kodirati.
    int cn = is.peek();
    if (c >= 'a' && c <= 'z' && cn >= 'a' && cn <= 'z')
    {
      int koda = cn - 'a';

      // Če je ta koda še prosta, jo definirajmo zdaj.
      if (kode[koda] == '.') kode[koda] = c;

      // Če lahko ta in naslednji znak zakodiramo, ju dajmo.
      if (kode[koda] == c)
      {
        is.get();              // Uporabili smo tudi naslednji vhodni znak.
        os.put('A' + koda); // Izpišimo kodo.
        continue;
      }
    }

    // Če nismo uspeli zakodirati para znakov, samo izpišimo trenutni znak.
    os.put(c);
  }
}
```

Dekodiranje je še lažje. Kodirano besedilo berimo znak po znak; če je trenutni znak velika črka, pogledamo v tabelo kod, kjer izvemo, katera je prva črka para, ki ga kodira ta velika črka; druga črka para je potem mala različica te velike črke. Če pa je bil trenutni znak kaj drugega kot velika črka, ga brez sprememb izpišemo.

```
void Dekodiraj(istream &is, ostream &os, char kode[26])
{
  while (true)
  {
    // Preberimo naslednji znak.
    int c = is.get(); if (! is) break;

    // Če je velika črka, jo dekodirajmo v dve mali.
    if (c >= 'A' && c <= 'Z')
      os.put(kode[c - 'A']).put(c - 'A' + 'a');

    // Sicer izpišimo trenutni znak brez sprememb.
    else os.put(c);
  }
}
```

12. Rudarji

Pri določenem razporedu rudarjev med rove je dosežena kvota m, če v vsakem rovu izkopljejo vsaj m enot rude. Največji m, ki ga dosežejo, je torej minimalni izkop po vseh rovih: rov, v katerem izkopljejo najmanj rude, je tisti, ki preprečuje, da bi dosegli večji m.

Preprosta rešitev naloge je torej ta, da delavce razporejamo v rove enega po enega, pri čemer vsakič dodamo naslednjega delavca v tisti rov, ki ima trenutno najmanjši izkop — ta rov je namreč tisti, zaradi katerega kvota m trenutno ne more biti večja, kot je. Če ima več rovov enak najmanjši izkop, je vseeno, v katerega od njih damo novega delavca, saj bomo tako ali tako morali dodati po enega delavca v vse take rove, preden se bo kvota lahko kaj povečala.

Da poiščemo rov z najmanjšim izkopom, lahko uporabimo zanko po vseh rovih, kar nam vzame $O(k)$ časa; ker moramo to narediti pri vsakem delavcu, teh pa je d, imamo rešitev s časovno zahtevnostjo $O(kd)$. Lahko pa hranimo rove v kopici (prioritetni vrsti), urejene tako, da je tisti z najmanjšim izkopom na vrhu (v korenu kopice). Tako v vsakem trenutku vemo, v kateri rov poslati naslednjega rudarja; s tem se izkop v tem rovu poveča in ga moramo pogrezniti globlje v kopici, da le-ta ostane primerno urejena. Kopica je globoka $O(\log k)$ nivojev, zato tudi pogrezanje traja toliko časa; ker ga moramo izvesti po dodajanju vsakega rudarja, imamo rešitev s časovno zahtevnostjo $O(d \log k)$.

Obstaja pa še boljša rešitev. Izkop v posameznem rovu je vsota prvih nekaj (toliko, kolikor rudarjev pošljemo v ta rov) števil v seznamu, ki opisuje ta rov, torej neka delna vsota tega seznama. Dosežena kvota m (pri določenem razporedu rudarjev med rove) je minimum izkopa po vseh rovih. Tako so torej možne vrednosti m le delne vsote seznamov, ki opisujejo vhodne podatke. Ti seznami imajo vsega skupaj n elementov, torej je tudi različnih delnih vsot lahko največ n.

Recimo, da imamo urejen seznam vseh teh n delnih vsot iz vseh rovov. Naj bo s_d v tem seznamu d-ta najmanjša vsota. Za to vsoto in vse pred njo lahko pogledamo, na kateri rov se posamezna od njih nanaša; recimo, da se (za vsak i) d_i od teh d vsot nanaša na rov i. Če bi torej delavce razporedili med rove tako, da bi (za vsak i) v i-ti rov prišlo d_i delavcev, bi dobili razpored, pri katerem bi v tistem rovu, na katerega se nanaša d-ta najmanjša vsota, izkopali s_d enot rude, v vsakem od ostalih rovov pa $\leq s_d$. Če v vsakega od rovov, v katerih se izkoplje strogo manj kot s_d, dodamo še enega rudarja, pa bo v tem rovu gotovo izkopano vsaj s_d rude, saj bi

drugače videli pred s_d v urejenem seznamu še eno delno vsoto več iz tega rova, tedaj pa bi namesto d_i rudarjev že na začetku razporedili vanj $d_i + 1$ rudarjev.[23]

Tako smo torej dobili razpored, ki porabi največ $d + k - 1$ rudarjev in doseže v vsakem rovu izkop vsaj s_d. Zdaj lahko število rudarjev zmanjšamo (na d) z neke vrste požrešnim algoritmom: na vsakem koraku pobrišemo zadnjega rudarja iz tistega rova, pri katerem bomo s tem najmanj pokvarili minimalni izkop po vseh rovih (natančneje povedano: če bi za vsak rov pogledali, kakšna je pri njem delna vsota z enim rudarjem manj, kot jih je zdaj v tistem rovu, potem moramo pobrisati zadnjega rudarja iz tistega rova, pri katerem je ta delna vsota največja).

Prepričajmo se, da je ta rešitev res optimalna. Naj bo $D = (d_1, \ldots, d_k)$ poljuben razpored rudarjev v rove in naj bo s minimum izkopa po posameznih rovih; rekli bomo, da je D *ekonomičen*, če zanj velja, da če iz poljubnega rova odslovimo zadnjega rudarja, bo izkop v tem rovu po novem $\leq s$, ne pa $> s$.

Razpored z $d+k-1$ (ali manj) rudarji, ki smo ga najprej poiskali pri naši rešitvi, je vsekakor bil ekonomičen: minimalni izkop je bil s_d; tisti rov, ki mu pripada delna vsota s_d v urejenem seznamu vseh delnih vsot, je imel pri tem razporedu izkop natanko s_d in z enim rudarjem manj bi imel izkop $\leq s_d$; za vsakega od ostalih rovov pa velja, da je imel pred dodajanjem zadnjih $k - 1$ rudarjev v razpored izkop $\leq s_d$, nato pa je dobil enega novega rudarja in ima zdaj izkop $\geq s_d$; če bi mu torej enega rudarja odvzeli, bi imel izkop spet $\leq s_d$. Tako torej vidimo, da bi vsak rov imel izkop $\leq s_d$, če bi mu odvzeli zadnjega rudarja, torej je razpored res ekonomičen.

Prepričajmo se zdaj, da naš požrešni postopek ohranja ekonomičnost razporeda, ko briše rudarje. To naredimo z indukcijo. Recimo, da imamo pred brisanjem nekega rudarja razpored $D = (d_1, \ldots, d_k)$, ki je ekonomičen; naj bodo $v_1, \ldots, v_k$ izkopi po posameznih rovih; njihov minimum je $s := \min_i v_i$. Naj bo v'_i izkop v rovu i, če iz njega odslovimo zadnjega rudarja in ostane le $d_i - 1$ rudarjev (pri $d_i = 0$ si mislimo $v'_i = -\infty$). Naš požrešni postopek odslovi rudarja iz tistega rova, ki ima največji v'_i. Po predpostavki je bil D ekonomičen, torej je $v'_i \leq s$ (in ne $> s$).

(1) Če je $v'_i = s$, je novi minimum izkopa po vseh rovih $s' = s$, torej enak kot prej. Za ostale rove še vedno velja, da če jim enega rudarja odvzamemo, se jim izkop zmanjša na $\leq s$ in s tem tudi na $\leq s'$; očitno pa tudi za rov i velja, da če mu enega rudarja odvzamemo, bo njegov izkop potem $\leq s = s'$. Torej je ekonomičen tudi novi razpored.

(2) Če pa je $v'_i < s$, je novi minimum izkopa po vseh rovih dosežen ravno tu, torej pri $s' = v'_i$. Če iz tega rova odslovimo še enega rudarja, bo izkop pri njem seveda $\leq s'$. Kaj pa, če odslovimo po enega rudarja iz nekega drugega rova j? Pri njem izkop po definiciji pade na v'_j, to pa je $\leq v'_i = s'$, ker smo i izbrali tako, da je imel največji v'_i. Novi izkop v rovu j je torej $\leq s'$, prav kot smo hoteli dokazati. □

Izkaže se, da če obstaja neki ekonomičen razpored D z r rudarji in minimalnim izkopom s, potem z r rudarji ni mogoče sestaviti razporeda, ki bi imel minimalni izkop (po vseh rovih) večji od s.

Pa recimo, da bi se to vendarle dalo; takemu razporedu recimo $D' := (d'_1, \ldots, d'_k)$

[23] Tu je treba paziti še na robni primer: lahko se zgodi, da v kak rov ne moremo dodati še enega rudarja, ker bo že pri dosedanjem razporedu izkopan do konca. Naš razmislek bo še vseeno deloval, le da moramo namesto s_d uporabiti skupno količino rude v tistem rovu, pri katerem je ta skupna količina najmanjša. (Ko smo že pri robnih primerih: drugi je še ta, da pri $d < k$ ne moremo poslati niti po enega rudarja v vsak rov, zato bo rezultat gotovo 0.)

in minimalni izkop pri njem naj bo s'. Ker je imel D minimalni izkop s, je bil v vsaj enem rovu tam izkop točno s; recimo, da je to rov i. Ker ima v D' vsak rov izkop vsaj s', kar je $> s$, to pomeni, da je v rovu i zdaj zaposlenih več rudarjev kot prej, torej $d'_i > d_i$. Ker pa je skupno število rudarjev še vedno r, mora biti v vsaj enem drugem rovu zaposlenih manj rudarjev kot prej; recimo, da je to rov j, kjer je torej $d'_j < d_j$. Ker je D ekonomičen, vemo, da če v rovu j zaposlimo manj kot d_j rudarjev, bo izkop v njem $\leq s$, ne pa $> s$. Ker pa je $s < s'$, bo s tem tudi minimalni izkop v D' kvečjemu s in s tem manjši od s', tako da smo v protislovju. □

Povzemimo, kar smo videli doslej: naš požrešni algoritem začne z ekonomičnim razporedom in iz njega briše rudarje tako, da razpored ostane ekonomičen; na koncu ima ekonomičen razpored z d rudarji in nekim minimalnim izkopom s; in pravkar smo pokazali, da to slednje pomeni, da ne obstaja razpored z d rudarji in minimalnim izkopom $> s$. Torej je razpored, ki ga je našel naš požrešni algoritem, res optimalen.

Razmislimo še o tem, kako ta postopek učinkovito implementirati. Na začetku nam ni treba zares urejati zaporedja vseh delnih vsot $s_1, \ldots, s_n$ (kar bi vzelo $O(n \log n)$ časa); dovolj je že, če poiščemo d-to največjo od njih, kar lahko naredimo v $O(n)$ časa z algoritmom quickselect; potem lahko v $O(n)$ časa pregledamo, katere delne vsote spadajo med d najmanjših in koliko od njih pripada kateremu rovu; to slednje pa je vse, kar potrebujemo, da lahko potem v $O(d)$ časa pripravimo začetni razpored d rudarjev. V $O(k)$ časa ga potem dopolnimo v ekonomični razpored $d + k - 1$ rudarjev z minimalnim izkopom s_d; nato zložimo rove v kopico glede na v'_i in zato za vsako brisanje rudarja porabimo $O(\log k)$ časa, skupaj torej $O(k \log k)$. Časovna zahtevnost te rešitve je torej $O(n + k \log k)$. Oglejmo si še njeno implementacijo v C++:

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

int Rudarji(int d, const vector<vector<int>> &rovi)
{
  int k = rovi.size(); // število rovov
  if (d < k || k == 0) return 0;
  vector<vector<int>> vsote(k); // seznami delnih vsot po rovih

  // S naj vsebuje vse delne vsote kot pare ⟨vsota, številka rova⟩.
  vector<pair<int, int>> S;
  int minVsota = numeric_limits<int>::max();
  for (int i = 0; i < k; ++i) {
    int vsota = 0; vsote[i].push_back(0);
    for (int x : rovi[i]) {
      vsote[i].push_back(vsota += x); S.emplace_back(vsota, i); }
    minVsota = min(minVsota, vsota); }

  // Poiščimo najmanjših d delnih vsot.
  nth_element(S.begin(), S.begin() + d − 1, S.end());

  // Pripravimo začetni razpored D z natanko d rudarji.
  vector<int> D(k, 0); for (int i = 0; i < d; ++i) ++D[S[i].second];

  // Dodajmo po enega rudarja v rove, ki izkopljejo < sd rude.
  int sd = min(S[d − 1].first, minVsota), r = d;
```

```
  for (int i = 0; i < k; ++i) if (vsote[i][D[i]] < sd) ++D[i], ++r;
  // Pripravimo kopico.
  priority_queue<pair<int, int>> kopica;
  for (int i = 0; i < k; ++i) if (D[i] > 0) kopica.emplace(vsote[i][D[i] - 1], i);
  // Pobrišimo toliko rudarjev, da jih ostane le d.
  while (r-- > d) {
    int i = kopica.top().second; kopica.pop();
    if (--D[i] > 0) kopica.emplace(vsote[i][D[i] - 1], i); }
  // Zdaj imamo optimalni razpored; določimo minimalni izkop po vseh rovih.
  int kvota = numeric_limits<int>::max();
  for (int i = 0; i < k; ++i) kvota = min(kvota, vsote[i][D[i]]);
  return kvota;
}
```

13. Največji xor

Operacija XOR je asociativna in komutativna — podobno kot na primer seštevanje ali množenje. Za razliko od njiju dveh pa ima XOR še to zanimivo lastnost, da je u xor $u = 0$ za vsak u, kajti če sta oba operanda enaka, se ne bo nikoli zgodilo, da bi bil neki bit v enem prižgan, v drugem pa bi bil istoležni bit ugasnjen, kar pa je pogoj za to, da bi bil tisti bit v rezultatu prižgan. Za vsak u velja tudi u xor $0 = u$, kajti če so v drugem operandu vsi biti ugasnjeni, dobimo v rezultatu prižgan bit le tam, kjer je bil istoležni bit prižgan že v prvem operandu.

Te lastnosti pomenijo, da če nas zanima XOR več zaporednih števil v vhodnem zaporedju, na primer $\mathrm{xor}_{i=\ell}^{d} a_i$, lahko najprej XORamo med seboj prvih d števil in jih nato še enkrat XORamo s prvimi $\ell - 1$ števili. Primer za $\ell = 3$, $d = 5$:

$$\begin{aligned} & (a_1 \text{ xor } a_2 \text{ xor } a_3 \text{ xor } a_4 \text{ xor } a_5) \text{ xor } (a_1 \text{ xor } a_2) \\ = \; & (a_1 \text{ xor } a_1) \text{ xor } (a_2 \text{ xor } a_2) \text{ xor } (a_3 \text{ xor } a_4 \text{ xor } a_5) \\ = \; & 0 \text{ xor } 0 \text{ xor } (a_3 \text{ xor } a_4 \text{ xor } 5) \\ = \; & a_3 \text{ xor } a_4 \text{ xor } 5. \end{aligned}$$

Vpeljimo torej novo zaporedje b_i, ki ga dobimo iz vhodnega zaporedja a_i tako, da po vrsti XORamo njegove člene med seboj: $b_i = \mathrm{xor}_{j=1}^{i} a_j$. Računamo ga lahko tako, da začnemo z $b_0 = 0$ in nadaljujemo po formuli $b_i = b_{i-1}$ xor a_i. Vrednosti oblike $\mathrm{xor}_{i=\ell}^{d} a_i$, kakršne nas zanimajo pri naši nalogi, lahko potem računamo po formuli b_d xor $b_{\ell-1}$. Naloga sprašuje po največji taki vrednosti; poiščemo jo lahko tako, da gremo v zanki po vseh možnih d (od 1 do n) in se pri vsakem d vprašamo: katera od vrednosti $b_0, b_1, \ldots, b_{d-1}$ dá največji rezultat po XORanju z b_d? Če bomo znali hitro odgovarjati na takšna vprašanja, bomo lahko hitro prišli tudi do rezultata, po katerem sprašuje naloga.

Razmislimo najprej o lažji obliki naloge, pri kateri je $k = 0$, torej ne dovolimo predhodnega spreminjanja bitov v vhodnih številih. V tem primeru med različicama (a) in (b) ni nobene razlike. Če so vhodna števila različno dolga (v dvojiškem zapisu), jih v mislih podaljšajmo s toliko vodilnimi ničlami, da bodo na koncu vsa enako dolga — recimo, da so dolga m bitov. Števila a_i in zato tudi b_i si lahko zdaj predstavljamo tudi kot nize m znakov (ničel in enic).

Recimo torej zdaj, da za neki konkretni b_d razmišljamo, kateri b_i bi dal skupaj z njim največjo vrednost b_i xor b_d. Če se niza b_i in b_d razlikujeta v prvem (najvišjem,

najbolj levem) bitu, bo v rezultatu tisti bit prižgan in njegova vrednost bo vsekakor večja, kot če bi bil ta bit ugasnjen (ne glede na to, kaj se bo zgodilo na nižjih bitih). Če je le mogoče, se torej pri izboru b_i omejimo na tiste nize, ki se v najvišjem bitu razlikujejo od b_d; če pa ni nobenega takega, nam pač ostanejo le tisti, ki se v najvišjem bitu ujemajo z b_d. V nadaljevanju imamo zdaj pred seboj neko mogoče malo manjšo množico kandidatov b_i in podoben razmislek ponovimo za drugi najvišji bit: spet se omejimo na tiste b_i, ki se v tem bitu razlikujejo od b_d, če pa ni nobenega takega, pa pač ostanemo pri teh, ki se v njem ujemajo z b_d. Tako nadaljujemo vse do najnižjega bita.[24]

Za učinkovito implementacijo tega postopka je koristno zložiti vse nize $b_0, b_1, \ldots, b_{d-1}$ v drevo (*trie*), v katerem je na vsaki povezavi en bit in vsakemu b_k ustreza neka veja od korena drevesa do enega od listov, pri čemer biti na povezavah te veje tvorijo ravno niz b_k. Če se več nizov b_k ujema v prvih nekaj bitih, si tam tudi delijo ista vozlišča. Naš postopek začne pri korenu drevesa in se na vsakem koraku poskuša spustiti navzdol po povezavi, ki je označena z nasprotnim bitom od trenutnega bita niza b_d, če pa take povezave ni, pa gre pač po tisti drugi (ki se ujema s trenutnim bitom niza b_d). Tako lahko pri posameznem d najdemo najprimernejši b_i v $O(m)$ časa, celotna rešitev pa vzame $O(nm)$ časa. Na začetku imejmo prazno drevo, nato pa pojdimo z zanko naraščajoče po d in pri vsakem b_d najprej poiščimo najprimernejši b_i v dosedanjem drevesu, nato pa dodajmo b_d v drevo.

Doslej smo razmišljali o $k = 0$; oglejmo si zdaj primer, ko je $k = 1$, torej lahko en bit nekje v vhodnih podatkih tudi spremenimo. Ostanimo zaenkrat pri lažji različici (*a*), kjer smemo spremeniti kateregakoli od m bitov kateregakoli števila, tudi vodilne ničle. Na rezultat $\mathrm{xor}_{i=\ell}^{d} a_i$ vpliva sprememba enega bita seveda le, če smo ga spremenili v enem od števil $a_\ell, \ldots, a_d$, ne pa nekje pred a_ℓ ali za a_d; je pa za omenjeni rezultat vseeno, v *katerem* od števil $a_\ell, \ldots, a_d$ smo ta bit spremenili — pomembno je le, kateri od m bitov je to bil. Zaradi te spremembe se potem spremeni tudi istoležni bit v rezultatu $\mathrm{xor}_{i=\ell}^{d} a_i$. Taka sprememba je za nas koristna, če z njo prižgemo neki bit, ki bi bil sicer v rezultatu ugasnjen.

V našem postopku s spuščanjem po drevesu moramo zdaj pravilo za odločanje o tem, v katero poddrevo se spustiti, prilagoditi takole: če lahko spuščanje nadaljujemo tako, da bo v rezultatu operacije XOR na trenutnem mestu nastala enica, to naredimo (če lahko enico dobimo na več načinov, moramo preizusiti vse, ker ne moremo vnaprej vedeti, kateri bo pripeljal do najboljšega reultata); sicer pa pač nadaljujemo spuščanje tako, da bo v rezultatu na trenutnem mestu nastala ničla (in spet, če lahko to naredimo na več načinov, moramo preizkusiti vse). (Preprostejše pravilo, ki smo ga videli pri rešitvi za $k = 0$, je le poseben primer tega tukaj.)

Recimo, da je trenutni bit števila b_d enak c. Pri $k = 1$ lahko do enice na trenutnem mestu v rezultatu pridemo na dva načina: če se spustimo po povezavi, označeni z $1 - c$; ali če se spustimo po povezavi, označeni s c, in pri tem izkoristimo možnost, da en bit v vhodnih podatkih spremenimo. (Ni sicer nujno, da oba načina res obstajata; npr. morda iz trenutnega vozlišča ne izhaja primerna povezava ali pa smo možnost spremembe enega bita izkoristili že nekoč prej pri spuščanju do trenutnega vozlišča.) Do ničle na trenutnem mestu v rezultatu pa lahko pridemo le

[24] S tem postopkom smo se že srečali leta 2016 pri tretji nalogi v tretji skupini; gl. str. 57–58 v *Biltenu* 2016.

na en način: če se spustimo po povezavi, označeni s c. Možnost, da bi se spustili po povezavi $1 - c$ in pri tem izkoristili možnost spremembe enega bita, odpade zato, ker če taka povezava obstaja, bi se lahko spustili po njej brez spremembe enega bita in tako dobili na trenutnem mestu v rezultatu enico, ne pa ničle.

Ostane nam še primer, ko je $k > 1$. Tu opazimo, da ni nobene koristi od tega, da bi spremenili istoležni bit pri več različnih a_i-jih, saj bi pri XORanju po dve taki spremembi druga drugo izničili. Podobno kot pri $k = 1$ je tudi zdaj za rezultat vseeno, pri katerem a_i (z območja $a_\ell, \ldots, a_d$) določen bit spremenimo; vprašanje je torej le, na katerih mestih (od 0 do $m - 1$) naj spremenimo en bit v enem od vhodnih števil. Enako kot pri $k = 1$ se taka sprememba pri spuščanju po drevesu pokaže v tem, da sledimo taki veji, ki bi sicer (če bita ne bi spremenili) povzročila, da bi bil istoležni bit v rezultatu ugasnjen. Razlika v primerjavi s $k = 1$ je le ta, da namesto podatka o tem, ali smo možnost spremembe bita že izkoristili nekoč prej med spuščanjem po drevesu, potrebujemo podatek o tem, *koliko* takih sprememb nam je še ostalo.

Kakšna je časovna zahtevnost te rešitve? Pri $k = 1$ se od poti, po kakršni bi se spuščala prvotna rešitev (za $k = 0$), na vsakem nivoju odcepi stranska veja (ki izkoristi možnost spremembe), ki pa se v nadaljevanju ne cepi več (ker je možnost spremembe samo ena). Tako nastane $O(m)$ vej dolžine $O(m)$, torej imamo z vsakim b_d po $O(m^2)$ dela, za vse skupaj $O(n \cdot m^2)$. V splošnem (za večje k) pa opazimo, da lahko pri posameznem b_d dosežemo največ $\binom{m}{k}$ listov, kajti na toliko načinov se lahko odločimo, na katerih nivojih drevesa bomo izkoristili možnost spremembe enega bita. Ker do vsakega lista vodi veja dolžine $O(m)$ in ker moramo to storiti pri vsakem b_d, imamo časovno zahtevnost $O(nm \cdot \binom{m}{k})$; če je $k \ll m$, je to približno $O(n \cdot m^{k+1})$. Zapišimo dobljeno rešitev s psevdokodo:

funkcija REK(v, h, k, b_d):
  (* *Vhodni podatki: v je trenutno vozlišče drevesa in leži h nivojev nad listi;*
    k je število sprememb, ki so nam še na voljo;
    $b_d = b_d[m-1]b_d[m-2]\ldots b_d[0]$ je drugi operand XORa. *)
  if $v =$ NIL **then return** -1;
  if $h = 0$: (* *torej če je v list* *)
    $b_i :=$ vrednost, ki jo predstavlja pot od korena do v;
    $c := b_i$ xor b_d;
    če je $k > 0$ in je najnižji bit c-ja ugasnjen, ga prižgi;
    return c;
  $\beta := b_d[h]$; (* *trenutni bit b_d-ja* *)
  (* *Poskusimo nadaljevati tako, da dobimo v rezultatu enico.* *)
  $c :=$ REK($v.otrok[1-\beta], h-1, k, b_d$);
  (* *Morda jo lahko dobimo tako, da tu spremenimo en bit.* *)
  if $k > 0$ **then** $c := \max\{c,$ REK$(v.otrok[\beta], h-1, k-1, b_d \text{ xor } 2^h)\}$;
  (* *Sicer nadaljujmo tako, da dobimo v rezultatu ničlo.* *)
  if $c < 0$ **then** $c :=$ REK($v.otrok[\beta], h-1, k, b_d$);

glavni del programa:
  $c^\star := 0$;
  pripravi drevo (*trie*) s korenom r, na začetku dodaj vanj le niz b_0;

```
for d := 1 to n:
    c* := max{c*, Rek(r, m − 1, k, b_d)};
    dodaj niz b_d v drevo;
return c*;
```

Pri implementaciji je treba sicer še nekaj pazljivosti, da funkcija Rek ne bi porabila po $O(m)$ časa za vsako računanje c-ja ali primerjanje rezultatov iz dveh rekurzivnih klicev. O podrobnostih tega gl. *Bilten* 2016, str. 58–60.

Doslej smo se ukvarjali z različico (a) naše naloge, pri kateri smemo v okviru spreminjanja k bitov v vhodnih podatkih spreminjati tudi vodilne ničle, ki smo jih vhodnim nizom dodali, da so vsi postali enako dolgi. Oglejmo si zdaj še različico (b), pri kateri vodilnih ničel ne smemo spreminjati.

Recimo, da smo prišli do nekega d in se sprašujemo, pri katerem $\ell \le d$ začeti, da bo rezultat $\mathrm{xor}_{i=\ell}^{d} a_i$ čim večji. Recimo, da je najdaljši niz med $a_1, \ldots, a_d$ dolg μ bitov in da je zadnji niz te dolžine nastopil na indeksu i. Ločimo dva primera.

(1) Če je a_d dolg manj kot μ bitov: to pomeni, da so vsi nizi $a_{i+1}, \ldots, a_d$ krajši od μ bitov, tako da, če postavimo ℓ kamorkoli na to območje, bo tudi rezultat krajši od μ bitov in s tem manjši od kateregakoli števila, dolgega vsaj μ bitov — na primer od rezultata, ki ga dobimo, če postavimo $\ell = i$. V tem primeru se torej smemo omejiti na $\ell \le i$; pri takih ℓ pa vemo, da bo med števili $a_\ell, a_{\ell+1}, \ldots, a_d$ tudi število a_i, ki je dolgo μ bitov, zato lahko spremenimo katerihkoli k bitov hočemo (izmed najnižjih μ), saj jih bomo lahko, če ne drugje, spremenili v vhodnem številu a_i, ki bo zagotovo prisotno na območju od ℓ do d. V tem primeru bi bilo torej dobro imeti drevo, zgrajeno nad nizi $a_1, \ldots, a_i$ (oz. natančneje: drevo, ki vsebuje nize $b_0, \ldots, b_{i-1}$), na njem pa bi pognali enak postopek kot pri različici (a), pri čemer bi vse nize obravnavali kot dolge μ bitov.

(2) Če je a_d dolg natanko μ bitov: tu lahko že brez omejitve na $\ell \le i$ spreminjamo katerekoli bite (izmed najnižjih μ), saj jih bomo lahko, če ne drugje, spremenili v številu a_d. Tu bi bilo torej dobro imeti drevo, zgrajeno nad vsemi dosedanjimi nizi, prav kakor pri (a), in na njem tudi pognati enak postopek kot pri (a).

Naš postopek mora torej zdaj vzdrževati dve drevesi, eno z vsemi dosedanjimi nizi in eno do zadnjega najdaljšega niza; v slednje, ko pridemo do novega niza take dolžine, dodamo vse nize, ki jih še nismo. Ko pridemo do niza, ki je daljši od vseh doslej (torej ko se μ poveča), moramo obe drevesi tudi poglobiti na novi μ (nizom v mislih dodamo vodilne ničle, kar se v drevesu pozna tako, da od novega korena do starega vodi zaporedje ene ali več povezav z oznako 0). Zapišimo ta postopek s psevdokodo:

```
μ := 1; i := 1; c* := 0;
pripravi drevesi (trie) s korenoma r in r̃, na začetku dodaj vanju le niz 0;
for d := 1 to n:
  (* Na tem mestu velja: drevo s korenom r vsebuje nize b_0, ..., b_{d−1},
     drevo s korenom r̃ pa nize b_0, ..., b_{i−1}. Če je d > 1, je niz a_i dolg μ.
     Nizi a_{i+1}, ..., a_{d−1} so krajši od μ. *)
  if |a_d| > μ:
    poglôbi obe drevesi za |a_d| − μ nivojev; μ := |a_d|;
  if |a_d| < μ:
    c* := max{c*, Rek(r̃, μ − 1, k, b_d)};
```

else: (* *torej pri* $|a_d| = \mu$ *)
$c^\star := \max\{c^\star, \text{Rek}(r, \mu - 1, k, b_d)\}$;
while $i < d$:
dodaj niz b_i v drevo s korenom $\tilde{r}$; $i := i + 1$;
dodaj niz b_d v drevo s korenom r;
return $c^\star$;

Za konec razmislimo še o različici (c), pri kateri so vsi a_i z območja $\{0, 1, 2, 3\}$. Tedaj je tudi XOR več zaporednih a_i lahko le 0, 1, 2 ali 3. Če je kakšen $a_i = 3$, vzemimo njega samega in imamo optimalni rezultat 3. Sicer, če obstajata kakšna dvojka in kakšna enica, uporabimo poljubno podzaporedje oblike $1, 0, \ldots, 0, 2$ ali $2, 0, \ldots, 0, 1$ (koliko je ničel, ni važno, lahko ni tudi nobene) in imamo spet rezultat 3. Sicer, če obstaja kakšna dvojka, nam ta sama po sebi da optimalni rezultat, ki je 3 (pri $k > 0$) ali 2 (pri $k = 0$). Sicer, če obstaja kakšna enica, nam sama po sebi da optimalni rezultat 1. Sicer imamo zaporedje samih ničel; katerakoli od njih nam sama po sebi da optimalni rezultat, ki je 1 (pri $k > 0$) ali 0 (pri $k = 0$). Povzetek teh rezultatov kaže naslednja tabela:

Največji možni XOR, če obstaja kak a_i z vrednostjo				in če je	
0	1	2	3	$k = 0$	$k > 0$
*	*	*	da	3	3
*	da	da	ne	3	3
*	ne	da	ne	2	3
*	da	ne	ne	1	1
da	ne	ne	ne	0	1

14. Prefiksna in postfiksna oblika

Predelavo izraza iz prefiksne oblike v postfiksno lahko opišemo s preprostim rekurzivnim razmislekom. Izraz je bodisi sestavljen iz enega samega naravnega števila ali pa iz operanda in njegovih dveh operatorjev. V prefiksni obliki tega dvojega ni težko ločiti, saj moramo le pogledati, če je na začetku izraza operator ali število. Če je število, je le-to že samo po sebi izraz in je v postfiksni obliki enako kot v prefiksni, torej ga lahko takoj tudi izpišemo. Če pa je na začetku operator, potem vemo, da mu sledita podizraza, ki v prefiksni obliki opisujeta njegova operanda. Vsakega od teh lahko z rekurzivnim klicem predelamo v prefiksno obliko, na koncu pa izpišemo še operator, ki smo ga prebrali na začetku:

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

void PredelajIzraz(istream &is, ostream &os)
{
  char c; is >> c; // Preberimo prvi znak izraza.
  if (c == '+' || c == '-' || c == '*' || c == '/') {
    // Če je prvi znak operator, preberimo oba operanda in ju izpišimo v postfiksni
    // obliki. Med njima izpišimo presledek, na koncu pa še operator.
    PredelajIzraz(is, os); os << ' ';
```

```
      PredelajIzraz(is, os); os << ' ' << c; }
   else {
      // Sicer pa je izraz le število; preberimo ga do konca in ga izpišimo.
      is.putback(c);
      int n; is >> n; os << n; }
}
```

Če je izraz dolg n znakov, je časovna zahtevnost te rešitve $O(n)$, prostorska pa tudi, kajti v najslabšem primeru je lahko rekurzija gnezdena $O(n)$ nivojev globoko. To se zgodi na primer pri izrazu oblike `+ 1 + 1 + 1`...`+ 1 + 1 1` (ki ga bo treba predelati v `1 1`...`1 + +`...`+`); tak izraz ima k plusov in $k + 1$ enic, rekurzija pa bo šla $k+1$ nivojev globoko. Pri takih izrazih utegnejo nastopiti težave s pomanjkanjem prostora na skladu, zato je takrat bolje, če rekurzivno rešitev predelamo v iterativno, pri kateri sami vzdržujemo sklad. Naša iterativna rešitev bo v zanki pobirala ukaze s sklada in jih izvajala. Ukaze predstavimo preprosto z znaki, pri čemer nam bo znak `!` (klicaj) pomenil, da moramo prebrati naslednji podizraz v prefiksni obliki in ga izpisati v postfiksni obliki (to je torej ekvivalent vgnezdenega rekurzivnega klica pri prvotni rešitvi); poleg teh klicajev pa se lahko kot ukazi pojavijo še operatorji in presledki, ki jih moramo le izpisati na izhodni tok.

Ko pridemo do ukaza `!`, obdelamo naslednji izraz podobno kot prvotna rešitev: pogledamo njegov prvi znak; če ni operator, je cel izraz le eno število in ga moramo samo prebrati do konca in izpisati brez sprememb; če pa je prvi znak operator, moramo predelati prvi podizraz, izpisati presledek, predelati drugi podizraz, izpisati presledek in končno izpisati tisti operator, ki je bil v prefiksni obliki na začetku, v postfiksni pa mora biti na koncu. To je torej pet novih ukazov, ki jih dodamo na sklad (seveda v obrnjenem vrstnem redu, da bodo bliže vrhu sklada tisti ukazi, ki jih je treba izvesti prej), da jih bo naša glavna zanka sčasoma izvedla.

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

void PredelajIzraz2(istream &is, ostream &os)
{
   // Na skladu so ukazi, ki jih izvaja glavna zanka te funkcije. Klicaj pomeni,
   // da mora obdelati podizraz, druge znake pa le izpiše na izhodni tok.
   stack<char> sklad; sklad.push('!');

   // Ukaze bomo izvajali v zanki, dokler se sklad ne izprazni.
   while (! sklad.empty())
   {
      // Preberimo naslednji ukaz s sklada.
      char c = sklad.top(); sklad.pop();

      // Če je kaj drugega kot klicaj, ga le izpišemo.
      if (c != '!') { os << c; continue; }

      // Sicer moramo obdelati podizraz. Ali se začne na operator?
      is >> c;
      if (c == '+' || c == '-' || c == '*' || c == '/') {
         // Podizraz se res začne na operator. Treba bo predelati prvi operand,
         // izpisati presledek, predelati drugi operand in izpisati še en presledek
         // in operator. Dodajmo te ukaze na sklad.
         sklad.push(c);
```

```
            sklad.push(' '); sklad.push('!');
            sklad.push(' '); sklad.push('!'); }
        else {
            // Podizraz je le število. Preberimo ga do konca in ga izpišimo.
            is.putback(c);
            int n; is >> n; os << n; }
    }
}
```

Nalogo lahko rešimo tudi tako, da pregledujemo vhodni niz od konca proti začetku. To pomeni, da preden naletimo na neki operator, smo že v celoti prebrali tisti del niza, ki predstavlja njegova dva operanda. Koristno je torej, če smo do takrat že tudi predelali oba operanda in ju imamo nekje pri roki v postfiksni obliki; potem je zdaj primeren trenutek, da pripravimo postfiksno obliko celotnega podizraza, na katerega se nanaša pravkar prebrani operator: stakniti moramo torej niz s postfiksno obliko levega operanda, presledek, niz s postfiksno obliko desnega operanda, še en presledek in na koncu še pravkar prebrani operator. Vzdrževali bomo sklad, na katerega bomo odlagali tako predelane nize. To namreč pomeni, da ko pridemo do operatorja, smo ravnokar v celoti prebrali in predelali njegova operanda, zato sta njuna niza (v postfiksni obliki) na vrhu sklada, torej prav tam, kjer najlaže pridemo do njiju.

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

string PredelajIzraz3(const string &s)
{
    stack<string> sklad;
    for (int i = s.size() - 1; i >= 0; --i)
    {
        char c = s[i];
        if (c == '+' || c == '-' || c == '*' || c == '/') {
            // Trenutni znak je operator; na vrhu sklada sta njegova operanda,
            // že predelana v postfiksno obliko. Prav na vrhu je levi operand,
            // ker smo tega prebrali nazadnje, pod njim pa je desni operand.
            string levi = sklad.top(); sklad.pop();
            string desni = sklad.top(); sklad.pop();
            // Dodajmo na sklad postfiksno obliko izraza, ki se v prefiksni obliki
            // začne s pravkar prebranim operatorjem.
            sklad.push(levi + " " + desni + " " + c); }
        else if (c >= '0' && c <= '9')
        {
            // Trenutni znak je števka. Preberimo celotno število, ki mu pripada.
            int j = i; while (j >= 0 && s[j] >= '0' && s[j] <= '9') --j;
            // Znaki s[j + 1..i] tvorijo število. Dodajmo ga na sklad.
            sklad.push(s.substr(j + 1, i - j));
            i = j + 1; // stavek for bo izvedel še --i in tako nadaljeval pri i = j
        }
    }
    return sklad.top();
}
```

Slabost pri tej rešitvi je, da utegne na vhodnem nizu dolžine n porabiti $O(n^2)$ časa. Pri nizu `+ 1 + 1 ... + 1 + 1 1` bomo na primer v zadnjem koraku stikali niza `1` (levi operand) in `1 1 ... 1 + + ... +` (desni operand v postfiksni obliki) ter še zadnji `+` na koncu; desni operand je dolg $O(n)$, tako da bo stikanje že v tem zadnjem koraku vzelo $O(n)$ časa. Ker se to zgodi tudi na vsakem nivoju gnezdenja znotraj desnega operanda, se vsega skupaj nabere za $O(n^2)$. Temu se lahko izognemo, če krajših nizov ne stikamo sproti, pač pa jih hranimo v seznamih, povezane v verige (*linked lists*), kajti dva taka seznama lahko staknemo v $O(1)$ časa. V C++ovi standardni knjižnici lahko uporabimo razred `list` in stikamo z njegovo metodo `splice`. Krajše nize bomo zares staknili v en sam dolg niz šele čisto na koncu:

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

string PredelajIzraz3b(const string &s)
{
  stack<list<string>> sklad;
  for (int i = s.size() - 1; i >= 0; --i)
  {
    char c = s[i];
    if (c == '+' || c == '-' || c == '*' || c == '/') {
      // Trenutni znak je operator; na vrhu sklada sta njegova operanda,
      // že predelana v postfiksno obliko. Prav na vrhu je levi operand,
      // ker smo tega prebrali nazadnje, pod njim pa je desni operand.
      list<string> levi = sklad.top(); sklad.pop();
      list<string> desni = sklad.top(); sklad.pop();

      // Dodajmo na sklad postfiksno obliko izraza, ki se v prefiksni obliki
      // začne s pravkar prebranim operatorjem.
      levi.push_back(" "); levi.splice(levi.end(), desni);
      levi.push_back({' ', c}); sklad.push(levi); }
    else if (c >= '0' && c <= '9')
    {
      // Trenutni znak je števka. Preberimo celotno število, ki mu pripada.
      int j = i; while (j >= 0 && s[j] >= '0' && s[j] <= '9') --j;

      // Znaki s[j + 1..i] tvorijo število. Dodajmo ga na sklad.
      sklad.push({s.substr(j + 1, i - j)});
      i = j + 1; // stavek for bo izvedel še --i in tako nadaljeval pri i = j
    }
  }
  // Staknimo skupaj nize v seznamu, ki smo ga dobili kot rezultat.
  size_t dolzina = 0; for (auto &t : sklad.top()) dolzina += t.size();
  string u; u.reserve(dolzina); for (auto &t : sklad.top()) u += t;
  return u;
}
```

15. Zbiratelj

Nalogo si lahko predstavljamo kot iskanje najkrajše poti po prostoru stanj, pri čemer so stanja pari (t, A), ki povedo, da se Ivan nahaja v trgovini kartela t in ima množico kartic A. Možni prehodi pred stanji so potem naslednji:

- prodaja sličice: $(t, A) \to (t, A - \{x\})$, če je $x \in A$ in če je kartel t pripravljen trgovati s sličico x;
- nakup sličice: $(t, A) \to (t, A \cup \{x\})$, če $x \notin A$ in je kartel t pripravljen trgovati s sličico x;
- premik v drugo trgovino: $(t, A) \to (u, A)$, če kartela t in u nista skregana in če je $z_u \in A$ (sličica z_u je namreč pogoj za vstop v trgovino kartela u).

Stanj oblike (t, A) je $n \cdot 2^k$ (ker si lahko t izberemo na n načinov in ker sta za vsako od k sličic dve možnosti — lahko je v množici A ali pa ne), kar je še obvladljivo, ker je k pri tej nalogi majhen. Začetno stanje je $(1, \{1\})$, radi pa bi prišli v eno od stanj $(t, \{k\})$ za poljuben t.

Ker nas zanima le najmanjše število korakov, lahko prostor preiskujemo z iskanjem v širino. Pri tem vzdržujemo vrsto, v kateri hranimo stanja, do katerih že poznamo najkrajšo pot, nismo pa še pogledali, kako se dá pot iz njih nadaljevati. Na začetku dodamo v vrsto začetno stanje, nato pa v vsakem koraku vzamemo stanje z začetka vrste in dodamo na konec vrste njegove sosede (če jih še nismo). Da ne bomo istega stanja obiskali po večkrat, si moramo nekje zapisati, da smo neko stanje že dodali v vrsto; dovolj je že en sam bit za vsako možno stanje.

Ker nam na koncu ne bo treba izpisati poteka poti, ampak le število korakov na njej, tudi ni potrebe, da bi si za vsako stanje hranili dolžino najkrajše poti do njega; dovolj je že, da imamo ta podatek za prvo stanje v vrsti in da si zapomnimo, pri katerem stanju v vrsti se dolžina najkrajše poti (od začetnega stanja do njega) poveča za 1. (Med iskanjem v širino namreč vedno velja, da so oddaljenosti stanj v vrsti od začetnega stanja urejene naraščajoče in da se razlikujejo med seboj največ za 1.)

Da bomo lahko učinkoviteje našteli stanja, v katera se lahko iz nekega (t, A) premaknemo, si je koristno vnaprej pripraviti za vsako kombinacijo stanja t in sličice x seznam — recimo mu $S(t, x)$ — tistih kartelov u, ki zahtevajo to sličico ($z_u = x$) in niso skregani s t. Potem lahko trgovine, v katere se lahko iz (t, A) premaknemo, naštejemo preprosto tako, da gremo v zanki po vseh $x \in A$ in znotraj nje v vgnezdeni zanki po vseh $u \in S(t, x)$.

Iskanje v širino se načeloma konča, ko je vrsta prazna; takrat vemo, da smo preiskali vsa stanja, dosegljiva iz začetnega. Ker pa nas zanima le dolžina najkrajše poti do najbližjega končnega stanja, lahko postopek prekinemo takoj, ko dodamo kakšno končno stanje v vrsto, saj takrat že poznamo rezultat, ki ga bomo morali na koncu izpisati.

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

int main()
{
  int k, n, m; cin >> k >> n >> m;
  // z[t] = sličica, potrebna za vstop v trgovino kartela t.
  // C[t] = seznam sličic, s katerimi se tam trguje.
  vector<int> z(n); vector<vector<int>> C(n);
  for (int t = 0; t < n; ++t) {
    int r; cin >> z[t] >> r; --z[t];
```

```
    while (r-- > 0) { int x; cin >> x; C[t].push_back(x - 1); } }
  // skregan[t * k + u] = ali sta kartela t in u skregana.
  vector<bool> skregan(n * n, false);
  for (int i = 0; i < m; ++i) {
    int t, u; cin >> t >> u; --t; --u;
    skregan[t * n + u] = true; skregan[u * n + t] = false; }
  // sosedje[t * k + x] = seznam kartelov u, ki imajo z[u] = x in niso skregani s t.
  vector<vector<int>> sosedje(n * k);
  for (int t = 0; t < n; ++t) for (int u = 0; u < n; ++u)
    if (t != u && ! skregan[t * n + u]) sosedje[t * k + z[u]].push_back(u);

  // Dodajmo začetno stanje v vrsto. Stanja predstavimo s celimi števili;
  // spodnjih k bitov pove, katere sličice imamo, višji biti pa, v kateri trgovini smo.
  queue<int> vrsta; vector<bool> znano(n << k, false);
  vrsta.push(1); znano[1] = true;
  // Prvih nekaj stanj v vrsti je na oddaljenosti d korakov od začetnega,
  // tista od vključno stanja dNasl naprej pa na oddaljenosti d + 1 korakov.
  int d = 0, dNasl = -1, rezultat = (k <= 1) ? 0 : -1;

  // Podprogram za dodajanje novega stanja v vrsto.
  // Vrne „true“, če je to eno od končnih stanj.
  auto Dodaj = [&, k] (int U) -> bool {
    // Morda smo to stanje našli že prej.
    if (znano[U]) return false;
    // Dodajmo ga v vrsto.
    vrsta.push(U); znano[U] = true;
    if (dNasl < 0) dNasl = U;
    // Ali je to eno od končnih stanj?
    if ((U & ((1 << k) - 1)) == (1 << (k - 1))) { rezultat = d + 1; return true; }
    return false; };

  // Preiščimo prostor stanj v širino.
  while (! vrsta.empty() && rezultat < 0)
  {
    // Iz vrste poberimo stanje T, kjer smo v trgovini t.
    int T = vrsta.front(); vrsta.pop();
    int t = T >> k, slicice = T & ((1 << k) - 1);
    // Ali je to stanje, pri katerem se oddaljenost od začetnega poveča?
    if (T == dNasl) { ++d; dNasl = -1; }
    // Katere sličice lahko tu kupimo ali prodamo?
    for (int x : C[t])
      // V katero stanje pridemo s prodajo/nakupom sličice x?
      if (Dodaj(T ^ (1 << x))) break;
    if (rezultat >= 0) break;
    // V katere trgovine gremo lahko iz tega stanja?
    for (int x = 0; x < k && rezultat < 0; ++x) if (T & (1 << x))
      // Imamo sličico x, torej gremo lahko h kartelom u,
      // ki imajo z[u] = x in niso skregani s t.
      for (int u : sosedje[t * k + x])
        if (Dodaj((u << k) | slicice)) break;
  }

  // Izpišimo rezultat.
```

```
  cout << rezultat << endl; return 0;
}
```

16. Stave

Nalogo lahko rešujemo z neke vrste simulacijo, vendar moramo dobro razmisliti o tem, kakšen točno je za Metko najslabši scenarij, po katerem sprašuje naloga, da ga bomo potem lahko simulirali dovolj učinkovito, saj iz omejitev v besedilu naloge vidimo, da je udeležencev lahko veliko in tudi število točk je veliko.

Označimo začetne točke v padajočem vrstnem redu kot $a_1 \geq a_2 \geq \ldots \geq a_n$ (Metka je torej a_1).

Osebi a_2 in vsem nadaljnjim, ki imajo enako število točk kot a_2, bomo rekli „zasledovalna skupina“. Metka torej vedno stavi enako kot večina zasledovalne skupine (oz. kot ena od polovic, če se zasledovalci glede na stavo razdelijo na dve enako veliki skupini). Edini način, da se njena prednost pred zasledovalci zmanjšuje, je ta, da ona in „njena“ polovica (oz. večina) zasledovalne skupine stavijo narobe, druga polovica (oz. manjšina) pa pravilno. Da bo scenarij za Metko čim neugodnejši, mora biti njena polovica oz. večina zasledovalne skupine čim manjša (tako da bo čim več ljudi zmanjšalo zaostanek za Metko); najmanjša možna pa je $\lceil k/2 \rceil$ članov, če ima zasledovalna skupina k članov. Manjša polovica zasledovalne skupine (tista, ki v tem scenariju zmanjša svoj zaostanek za Metko za eno točko) pa je potem velika $k - \lceil k/2 \rceil = \lfloor k/2 \rfloor$ članov. Za ostale ljudi (tiste, ki niso v zasledovalni skupini), pa je koristno (in torej velja v za Metko najslabšem scenariju, po katerem sprašuje besedilo naloge), če tudi oni vsi stavijo pravilno, saj se bo tako tudi njihov zaostanek za Metko zmanjševal.

Primer: recimo, da ima zasledovalna skupina 4 člane in da je začetno stanje točk táko, kot ga kaže prva vrstica spodaj. Najneugodnejši scenarij bi šel potem takole:

8 2 2 2 2 0 0 ...
8 3 3 2 2 1 1 ...
8 4 3 3 3 2 2 ...
8 4 4 4 4 3 3 ...

Tu se je cela zasledovalna skupina v 3 korakih povečala za 2, Metka za 0, ostali ljudje pa za 3 točke.

Če pišemo $c = \lfloor \log_2 k \rfloor$, lahko zdaj v splošnem rečemo, da (pri najslabšem scenariju) zasledovalna skupina s k člani v $c + 1$ korakih pridobi po c točk, Metka nobene, vsi ostali udeleženci pa pridobijo v tem času $c + 1$ točk.[25] Če je bila na začetku takega cikla razlika med zasledovalno skupino in naslednjim udeležencem 1,

[25] O tem se lahko prepričamo z indukcijo po c. Oglejmo si najprej $c = 0$; to se zgodi le pri $k = 1$ (kar je tudi najmanjša možna velikost zasledovalne skupine). Tam bo Metka stavila enako kot ta edini zasledovalec in neugodni scenarij je, da se onadva zmotita, kasnejši zasledovalci pa pridobijo eno točko. Metka torej ni dobila nobene točke; zasledovalec je dobil $c = 0$ točk; kasnejši zasledovalci pa 1 točko, kar je $c + 1$; prav to pa zatrjuje naša trditev. — Recimo zdaj, da trditev drži do vključno $c - 1$, torej za vse $k < 2^c$. Preverimo, da velja potem tudi za c. Vzemimo torej poljuben k z območja $2^c \leq k < 2^{c+1}$. Največja možna velikost „manjše polovice“ zasledovalne skupine — torej tiste, s katero Metka ne potegne in ki potem zmanjša zaostanek za njo (in pride v prednost pred večjo polovico) — je po definiciji $k' := \lfloor k/2 \rfloor$. V prvem koraku se torej zasledovalna skupina velikosti k razcepi na dve manjši, pri čemer skupina velikosti k' dobi eno točko in s tem postane nova zasledovalna skupina, preostanek (velikosti $k - k'$) pa ne dobi točke; tudi Metka ne, vsi kasnejši zasledovalci pa jo dobijo. Zdaj imamo zasledovalno skupino

se bo do konca tega cikla zasledovalna skupina zlila z naslednjo in bo nastala večja zasledovalna skupina.

Člani zasledovalne skupine torej pridobijo povprečno $c/(c+1)$ točk na korak, ostali pa 1 točko na korak. Zato je koristno, če je zasledovalna skupina čim večja, v ta namen pa morajo vsi ostali tudi pridobivati točke, da jo bodo čim hitreje dohiteli. To je argument v prid temu, da naj ljudje, ki niso v zasledovalni skupini, vsi vedno stavijo pravilno.

Recimo torej, da imamo Metko z a_1 točkami, nato zasledovalno skupino k ljudi z a_2 točkami, nato pa osebo a_{k+2} z manj kot a_2 točkami. Po $\lfloor(a_1 - a_2 - 1) \cdot c\rfloor$ ciklih (s po $c+1$ koraki v vsakem ciklu) se lahko zasledovalna skupina Metki toliko približa, da jo bodo nekateri v naslednjem ciklu že ujeli; tisto je potem bolje simulirati posebej. Po drugi strani pa bo v $a_2 - a_{k+2} - 1$ ciklih zasledovalec a_{k+2} prišel tako blizu zasledovalne skupine, da bi jo v naslednjem ciklu že ujel in bi se zasledovalna skupina povečala. Pogledati moramo torej, kaj od tega dvojega se zgodi prej, potem lahko odsimuliramo ustrezno število ciklov in nato še en korak (da vidimo, kako se zasledovalna skupina spremeni), nato pa simulacijo po enakem postopku spet nadaljujemo.

Kakšna je časovna zahtevnost te rešitve? Število ciklov (s po $c+1$ koraki v vsakem ciklu), ki se izvedejo v celoti, lahko preprosto izračunamo in jih tako tudi odsimuliramo vse v enem zamahu, kot da bi šlo za en sam korak; temu sledi cikel, ki se izvede le delno, torej ga sestavlja $\leq c$ korakov, in na koncu tega delnega cikla bodisi nekdo iz zasledovalne skupine prehiti Metko (in naša simulacija se s tem lahko konča) bodisi naslednja skupina (tista za zasledovalno) dohiti zasledovalno skupino in se obe združita v eno, večjo zasledovalno skupino. Tako je torej prejšnji odstavek odsimuliral $O(c)$ korakov in zmanjšal število skupin za 1 (skupino tukaj tvorijo vsi, ki imajo enako število točk). Če smo imeli na začetku s skupin, smo izvedli torej $O(c \cdot s)$ korakov; in to je naprej reda $O(c \cdot n)$, saj skupin ne more biti več kot ljudi, teh pa je n.

Skupine lahko hranimo v seznamu kot pare $\langle$število ljudi, število točk$\rangle$ in enega koraka simulacije potem ni težko izvesti v $O(s)$ časa. Z nekaj pazljivosti pri implementaciji pa lahko to še izboljšamo. Videli smo, da v posameznem koraku večina ljudi dobi točko — edini izjemi sta Metka in večja polovica zasledovalne skupine. Namesto da gremo v takem primeru z zanko po vseh skupinah (razen ene) in jim povečujemo število točk za 1, imamo lahko neko globalno spremenljivko in v tem primeru povečamo za 1 le njo, nato pa število točk pri tisti skupini, ki *ne* dobi točke, zmanjšamo za 1. Tako imamo pri vsakem koraku le $O(1)$ dela in časovna zahtevnost celotne rešitve je $O(c \cdot n)$. Spomnimo se, da je $c = O(\log k)$, pri čemer je k število ljudi v zasledovalni skupini, kar je seveda $\leq n$; tako je torej časovna zahtevnost naše rešitve $O(n \log n)$.

Oglejmo si še implementacijo te rešitve v C++. Predpostavili bomo, da so podatki že organizirani v skupine in urejeni padajoče po številu točk; pravzaprav pa,

---

velikosti k', kar leži na območju $2^{c'} \leq k' < 2^{c'+1}$ za $c' := c - 1$; zato po induktivni predpostavki v naslednjih $c' + 1$ korakih ta skupina pridobi c' točk, Metka nobene, vsi ostali udeleženci pa $c' + 1$ točk. Če to združimo z dogajanjem v prvem koraku in upoštevamo, da je $c' + 1 = c$, dobimo ravno rezultat, o katerem govori naša trditev: skupaj je bilo $1 + (c' + 1) = c + 1$ korakov; Metka ni dobila nobene točke; manjša polovica zasledovalne skupine je dobila $1 + c' = c$ točk; večja polovica zasledovalne skupine je dobila $0 + (c' + 1) = c$ točk; ostali udeleženci so dobili $1 + (c' + 1) = c + 1$ točk. □

ker za nas absolutno število točk ni pomembno, pač pa le zaostanek za Metko, bomo predpostavili, da so v podatkih ti zaostanki (in da so skupine urejene naraščajoče po zaostanku). To pa tudi pomeni, da Metke v podatkih ni treba predstavljati eksplicitno; spodnja rešitev pričakuje v seznamu skupin le druge udeležence, ne pa Metke same.

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

typedef int_fast64_t myint;
struct Skupina { myint k, z; }; // k ljudi, z točk zaostanka

// Predpostavimo, da dobimo skupine urejene naraščajoče po zaostankih za Metko
// in da Metka ni všteta v prvo skupino. Funkcija vrne število korakov, v katerih
// Metke gotovo ne bo nihče prehitel; če je ne bo nikoli prehitel nihče, vrne −1.
myint Stave(vector<Skupina> v)
{
  const myint inf = numeric_limits<myint>::max();
  int n = v.size(); // število skupin

  // Robni primer: če je poleg Metke samo en (ali celo noben) udeleženec,
  // se vrstni red ne bo nikoli spremenil.
  if (n == 0 || n == 1 && v[0].k == 1) return −1;

  // Sicer izvedimo simulacijo. „dz“ pove, da se je zaostanek vseh udeležencev doslej
  // že zmanjšal za „dz“ in da v seznamu „v“ to ni upoštevano.
  myint stKorakov = 0, dz = 0;

  // Na začetek in konec tabele vrinimo stražarja; veljavne skupine so na indeksih i...n.
  v.insert(v.begin(), Skupina{}); v.push_back({}); int i = 1;

  while (true)
  {
    Skupina &s1 = v[i], &s2 = v[i + 1];
    myint z1 = s1.z − dz; // dejanski zaostanek prve skupine

    // Če imata vsaj dva človeka enako točk kot Metka, jo lahko že v naslednjem
    // koraku eden od njiju prehiti.
    if (z1 == 0 && s1.k > 1) break;

    // Izračunajmo dolžino cikla.
    int c = 0; while ((s1.k >> c) > 1) ++c;

    // V vsakih c + 1 korakih pridobi skupina s1 po c točk, Metka nobene,
    // ostale skupine po c + 1 točk.
    // Koliko ciklov lahko naredimo v celoti, ne da bi kdo iz prve skupine ujel Metko?
    myint nc = (c == 0) ? inf : (z1 == 0) ? 0 : (z1 − 1) / c;

    // Koliko ciklov lahko naredimo v celoti, ne da bi naslednja skupina ujela prvo?
    if (i + 1 <= n) nc = min(nc, s2.z − s1.z − 1);

    // Izvedimo te cikle.
    stKorakov += (c + 1) * nc; dz += (c + 1) * nc; s1.z += nc;

    // Ker zdaj ne moremo odsimulirati še enega celega cikla, odsimulirajmo en korak.
    // Med izvajanjem tega „delnega cikla“ korak za korakom se lahko število skupin
    // začasno poveča za 1, ker prva skupina razpade na večjo in manjšo polovico. Na srečo
    // je „v[i − 1]“ gotovo veljaven, ker smo na začetek vektorja „v“ vrinili stražarja.
    // To povečanje števila skupin je le začasno, saj bo na koncu tega „delnega cikla“
    // bodisi nekdo prehitel Metko (in bo konec simulacije) ali pa bo druga skupina
```

```
    // dohitela prvo (in se bosta združili v eno samo skupino).
    myint manjsina = s1.k / 2; myint vecina = s1.k − manjsina;
    if (manjsina == 0) {
      // Prvo skupino tvori en sam človek. Ali ga ujame druga skupina?
      if (s2.z − s1.z == 1) { s2.k += vecina; ++i; } }
    else {
      // Manjšina prehiti večino za 1 točko. Ali pri tem prehiti Metko?
      if (z1 == 0) break;
      // Če druga skupina pri tem koraku dohiti večino, si lahko mislimo,
      // da se je ta večina le premaknila iz prve skupine v drugo.
      if (i + 1 <= n && s2.z − s1.z == 1) { s2.k += vecina; s1.k = manjsina; }
      // Sicer pa prva skupina res razpade na dve manjši.
      else { v[−−i] = { manjsina, s1.z }; s1.k = vecina; ++s1.z; } }
    // V tem koraku se zaostanek za Metko vsem zmanjša za 1 (razen tistim, pri
    // katerih ostane nespremenjen, tem pa smo že povečali z za 1).
    ++stKorakov; ++dz;
  }
  return stKorakov;
}
```

Na začetek in konec zaporedja skupin (vektor v) smo vrinili še po eno skupino kot stražarja; stražar na koncu zagotavlja, da bo referenca s2 na drugo skupino vedno veljavna, četudi je v resnici skupina samo ena; stražar na začetku pa nam olajša vrivanje nove skupine na začetku seznama (kajti med simulacijo posameznih korakov se lahko število skupin poveča za 1, ko zasledovalna skupina razpade na manjšo in večjo polovico; je pa to povečanje le začasno, kajti na koncu tega delnega cikla bo bodisi manjša polovica prehitela Metko in bo simulacije konec bodisi se bo večja polovica zlila z naslednjo skupino in se bo število skupin spet zmanjšalo za 1).

Razmislimo za konec še o tem, kolikšna števila dobimo kot rezultate pri tej nalogi. Videli smo, da pri ciklu dolžine c dobijo člani zasledovalne skupine povprečno $c/(c+1)$ točk na cikel, člani ostalih skupin po 1 točko na cikel, Metka pa nobene. Najpočasneje jo bodo torej dohitevali pri $c = 1$, kar se zgodi pri skupini dolžine $k = 2^c = 2$. Če ima Metka na začetku simulacije t točk, poleg nje pa sta le še dva udeleženca s po 0 točkami, bo trajalo $2t$ korakov, preden jo bosta dohitela (v naslednjem koraku pa jo lahko eden od njiju že prehiti). Besedilo pravi, da je $t \leq 10^{16}$, tako da bodo 64-bitna cela števila dovolj velika za naše potrebe.

17. Zamik

Preprosta, a neučinkovita rešitev je, da vsakič zamenjamo po dva zaporedna zaboja. Če to delamo po vrsti od leve proti desni, se vsi zaboji zamaknejo ciklično za eno mesto v levo. Primer za $n = 5$ (v vsaki vrstici sta podčrtana tista dva zaboja, ki ju bomo zamenjali v naslednjem koraku):

$$\begin{array}{ccccc} \underline{0} & \underline{1} & 2 & 3 & 4 \\ 1 & \underline{0} & \underline{2} & 3 & 4 \\ 1 & 2 & \underline{0} & \underline{3} & 4 \\ 1 & 2 & 3 & \underline{0} & \underline{4} \\ \underline{1} & \underline{2} & 3 & 4 & 0 \end{array}$$

Če to ponovimo k-krat, bomo zamaknili zaboje za k mest, kot zahteva naloga. Tega ni težko zapisati z dvema gnezdenima zankama:

```
void Zamakni(int n, int k)
{
  for (int i = 0; i < k; ++i) for (int j = 1; j < n; ++j) Zamenjaj(j - 1, j);
}
```

Slabo pri tej rešitvi je, da porabimo $O(n \cdot k)$ časa, kar je v najslabšem primeru $O(n^2)$.

Še ena preprosta rešitev je, da vzdržujemo par tabel s podatki o tem, kje se zaboji nahajajo. Vsak zaboj v mislih oštevilčimo s številko mesta, na katerem je stal, preden smo zaboje začeli premikati. Pojdimo potem v zanki od leve proti desni po mestih in se pri vsakem mestu vprašajmo: na mestu i mora v končnem stanju tabele stati zaboj številka $(i + k) \bmod n$; kje stoji ta zaboj zdaj? To nam pove ena od naših tabel, tako da ga lahko zdaj zamenjamo s tistim, ki na mestu i stoji trenutno.

```
void Zamakni2(int n, int k)
{
  vector<int> zaboji(n);    // zaboji[i] = številka zaboja na mestu i.
  vector<int> kjeJe(n);     // kjeJe[i] = mesto, na katerem je zaboj številka i.
  for (int i = 0; i < n; ++i) kjeJe[i] = i, zaboji[i] = i;

  for (int i = 0; i < n; ++i)
  {
    // Kje je trenutno tisti zaboj, ki mora na koncu priti na mesto i?
    int j = kjeJe[(i + k) % n];
    if (j != i) Zamenjaj(i, j);

    // zi in zj sta zaboja, ki smo ju pravkar zamenjali.
    int zi = zaboji[i], zj = zaboji[j];

    // Vpišimo njun novi položaj v naši dve tabeli.
    kjeJe[zi] = j; kjeJe[zj] = i;
    zaboji[i] = zj; zaboji[j] = zi;
  }
}
```

Tako moramo izvesti največ $n - 1$ zamenjav, saj z vsako zamenjavo vsaj en zaboj pride na svoje pravo končno mesto in ga odtlej ne bomo več premikali. (Ko pa je $n - 1$ zabojev na pravih mestih, je tudi preostali n-ti zaboj na pravem mestu.) Časovna zahtevnost te rešitve je le $O(n)$, vendar pa porabi tudi $O(n)$ prostora za pomožni tabeli.

Videli smo, da je treba načeloma zaboje premakniti za k mest v levo; na mesto i pride zaboj z mesta $i+k$, dokler nam desni rob skladišča ne začne povzročati komplikacij. Recimo, da za začetek sistematično kličemo Zamenjaj(i, i + k) po naraščajočih i. Oglejmo si primer za $n = 11$, $k = 3$.

$$\begin{array}{ccccccccccc}
\underline{0} & 1 & 2 & \underline{3} & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
3 & \underline{1} & 2 & 0 & \underline{4} & 5 & 6 & 7 & 8 & 9 & 10 \\
3 & 4 & \underline{2} & 0 & 1 & \underline{5} & 6 & 7 & 8 & 9 & 10 \\
3 & 4 & 5 & \underline{0} & 1 & 2 & \underline{6} & 7 & 8 & 9 & 10
\end{array}$$

V prvih k korakih smo torej lepo zamenjali prvih k zabojev z naslednjimi k zaboji (ki so pri tem že prišli na svoje pravo mesto). V naslednjih k korakih bi podobno prišlo na pravo mesto še naslednjih k zabojev, prvih k pa bi se pomaknilo še za k mest v desno:

$$\begin{array}{ccccccccccc l}
3 & 4 & 5 & \underline{0} & 1 & 2 & \underline{6} & 7 & 8 & 9 & 10 & \text{(prejšnje stanje)} \\
3 & 4 & 5 & 6 & \underline{1} & 2 & 0 & \underline{7} & 8 & 9 & 10 & \\
3 & 4 & 5 & 6 & 7 & \underline{2} & 0 & 1 & \underline{8} & 9 & 10 & \\
3 & 4 & 5 & 6 & 7 & 8 & \underline{0} & 1 & 2 & \underline{9} & 10 &
\end{array}$$

Zdaj pa imamo desno od skupine prvih k zabojev (torej $[0, 1, 2]$ v gornjem primeru) nepopolno skupino, dolgo le $r := n \bmod k$ zabojev. Če z zamenjavami nadaljujemo po enakem postopku kot doslej, bomo lahko izvedli le še r zamenjav, nato pa trčili ob desni rob skladišča; pri tem pride zadnjih r zabojev (v gornjem primeru sta to zaboja $[9, 10]$) na prava mesta, prvih r zabojev (pri nas sta to $[0, 1]$) pride na konec skladišča, preostalih $k - r$ izmed prvih k zabojev (pri nas je to $[2]$) pa se ne premakne:

$$\begin{array}{ccccccccccc l}
3 & 4 & 5 & 6 & 7 & 8 & \underline{0} & 1 & 2 & \underline{9} & 10 & \text{(prejšnje stanje)} \\
3 & 4 & 5 & 6 & 7 & 8 & 9 & \underline{1} & 2 & 0 & \underline{10} & \\
3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 2 & 0 & 1 &
\end{array}$$

Na levih $n - k$ mestih so že pravi zaboji. Prvih k zabojev pa je prišlo na najbolj desnih k mest, kar je tudi dobro, vendar so se pri tem malo pomešali; morali bi biti v enakem vrstnem redu, kot so bili na začetku: $[0, 1, 2]$, ne pa $[2, 0, 1]$, kar smo dobili zgoraj. Težava je bila v tem, da je prišlo prvih r zabojev čisto na konec skladišča, mi pa bi radi, da bi desno od njih stalo še naslednjih $k - r$ zabojev. Če zdaj to skupino k zabojev na desnem koncu skladišča zamaknemo ciklično za $k - r$ mest v levo, bo prišla ravno v tako stanje, kot ga želimo. V našem primeru imamo $r = n \bmod k = 11 \bmod 3 = 2$, zato $k - r = 3 - 2 = 1$ in dobimo:

$$\begin{array}{lccc l}
[3\ 4\ 5\ 6\ 7\ 8\ 9\ 10] & \underline{2} & \underline{0} & 1 & \text{(prejšnje stanje)} \\
[3\ 4\ 5\ 6\ 7\ 8\ 9\ 10] & 0 & \underline{2} & \underline{1} & \\
[3\ 4\ 5\ 6\ 7\ 8\ 9\ 10] & 0 & 1 & 2 & \text{(končno stanje)}
\end{array}$$

Z oglatimi oklepaji smo označili del skladišča, ki v tem novem cikličnem zamiku ni sodeloval (torej levih $n-k$ mest). S tem zamikom je prišlo skladišče v želeno končno stanje. Novi ciklični zamik lahko seveda izvedemo na enak način kot prvotnega: počasi povečujemo i in vsakič zamenjamo zaboja na mestih i in $i + k'$, pri čemer je k' naš novi zamik, torej $k - r$. Na koncu tabele se morda pojavi potreba po še manjšem tretjem zamiku in tako naprej; prej ali slej pa se enkrat zgodi, da je $r = 0$ in takrat vemo, da je tudi zadnjih nekaj zabojev na pravih mestih (naslednji zamik bi zahteval, da zamaknemo zadnjih k zabojev za $k - r = k$ mest v levo, pri čemer pa se seveda ne bi nič spremenilo).

```
void Zamakni3(int n, int k)
{
  // d = dolžina območja (mesta n − d, . . . , n − 1), na katerem trenutno
  // izvajamo ciklični zamik za k mest v levo.
  for (int i = 0, d = n; i < n; i++)
  {
```

```
      if (i + k == n) {
        // Prišli smo do konca skladišča. Zadnjih k zabojev bo treba ciklično
        // zamakniti za r mest v desno oz. k − r v levo. Poseben primer:
        // pri r = 0 ni treba zamikati za k mest v levo, saj se s tem nič ne spremeni.
        int r = d % k;
        d = k;
        k = (k − r) % d; }
      if (k == 0) break;
      Zamenjaj(i, i + k);
    }
}
```

Izkaže se, da izvede ta rešitev popolnoma enako zaporedje zamenjav kot prejšnja (Zamakni2), pri tem pa porabi le $O(1)$ dodatnega pomnilnika namesto $O(n)$.

Še ena elegantna ideja pa je naslednja. Recimo, da zamenjamo zaboja na mestih 0 in k, nato na k in $2k$, nato na $2k$ in $3k$ in tako naprej. S prvo od teh zamenjav pride zaboj z mesta k na mesto 0, torej prav tja, kamor ga hočemo premakniti. Podobno druga zamenjava premakne zaboj z mesta $2k$ na mesto k, kot si tudi želimo; in tako naprej. Vsaka od teh zamenjav pa premakne za k mest v desno tisti zaboj, ki je bil prvotno na mestu 0. Ko tako gledamo po vrsti mesta 0, k, $2k$, $3k, \ldots$, ti indeksi seveda sčasoma postanejo $\geq n$; recimo na primer, da je ak prvi večkratnik k-ja, ki je $\geq n$; zato je $ak < n + k$ in $ak \bmod n < k$. Zaboj z mesta $ak \bmod n$ se mora ob zamiku za k mest v levo skočiti z levega konca skladišča na desni konec in se premikati naprej v levo od tam; pristane na mestu $(ak \bmod n) - k + n$, kar je ravno enako $(a-1)k \bmod n = (a-1)k$ — prav to pa je mesto, pri katerem smo izvedli zamenjavo z mestom $ak \bmod n$. Z našimi zamenjavami med mesti, ki so zaporedni večkratniki števila k, lahko torej nadaljujemo tudi po tistem, ko dosežejo in presežejo n, le da moramo takrat od njih gledati samo ostanke po deljenju z n:

$$0,\ k \bmod n,\ 2k \bmod n,\ 3k \bmod n,\ \ldots.$$

Ker imamo le n različnih mest, se začno ti ostanki prej ali slej ponavljati. Recimo, da je prvi ostanek, ki se pojavi drugič, $bk \bmod n$, ki je recimo enak $ck \bmod n$ za neki $c > b$. Toda to pomeni, da je $(c-b)k \bmod n = 0$, torej se je moral že ostanek 0 pojaviti znova (po $c-b$ korakih od začetka). Prvi ostanek, ki se pojavi drugič, je torej ostanek 0; recimo, da se prvič ponovi po u korakih: $uk \bmod n = 0$. V tem primeru naj bo zadnja zamenjava, ki jo še izvedemo, tista med mestoma $(u-2)k \bmod n$ in $(u-1)k \bmod n$. Ta premakne zaboj z mesta $(u-1)k \bmod n$ za k mest v levo, kot je prav, na njegovo mesto pa premakne zaboj 0 (spomnimo se, da so vse naše zamenjave doslej premikale zaboj 0 za k mest v desno). Zaboj 0 je zdaj na mestu $(u-1)k \bmod n$, če pa bi se od tod premaknil še za k mest v desno, bi prišel na $uk \bmod n = 0$, torej stoji zdaj ravno k mest *levo* od svojega prvotnega položaja — to pa je točno tam, kjer ga hočemo.

Tako torej vidimo, da smo z zanko, ki smo jo ravnokar opisali, uspešno zamaknili za k mest v levo vse zaboje na mestih, ki smo jih obiskali. Katera mesta pa so to in katera so nam še ostala? Število oblike $ak \bmod n$ dobimo tako, da od ak odštejemo neki večkratnik n-ja; torej lahko to število zapišemo kot $ak - bn$. Naj bo d najmanjši skupni večkratnik k in n; pišimo $k = d \cdot K$ in $n = d \cdot N$ (in za K in N potem vemo, da sta si tuja, kajti če bi imela kak skupen prafaktor, bi ta postal del skupnega

večkratnika d); potem je $ak - bn = (aK - bN)d$. Indeksi mest, s kateri smo imeli doslej opravka, so torej sami večkratniki d-ja. Na območju od 0 do $n-1$ je takih večkratnikov N, saj smo rekli, da je $n = d \cdot N$. V prejšnjem odstavku smo videli, da smo obiskali u takih večkratnikov in da so bili vsi različni; torej je $u \leq N$. Ali je mogoče, da bi bil $u < N$? Vemo, da je $uk \bmod n = 0$, torej za neki v velja $uk = vn$, torej $uK = vN$; desna stran je večkratnik N, torej mora biti leva tudi; in ker je K tuj N-ju, je lahko uK večkratnik N-ja le tako, da je že u sam po sebi večkratnik N-ja. Torej je nemogoče, da bi bil $u < N$; veljati mora $u = N$. Vidimo torej, da smo obiskali vseh N takih indeksov (z območja od 0 do $n-1$), ki so večkratniki d-ja, in pravilno premaknili njihove zaboje za k mest v levo.

Naše skladišče z n mesti si lahko predstavljamo kot razdeljeno na N skupin s po d mesti. Vidimo, da smo zaenkrat uspešno obdelali prvo mesto v vsaki skupini. Če bi našo zanko namesto pri 0 začeli pri 1, bi bilo vse skupaj čisto podobno, le da bi obdelali drugo mesto v vsaki skupini. Podobno bi potem lahko začeli zanko še pri 2 in tako obdelali tretje mesto v vsaki skupini; in tako naprej. Tako smo dobili naslednjo rešitev:

```
for z := 0 to d − 1:
  i := z;
  while true:
    j := (i + k) mod n; if j = z break;
    Zamenjaj(i, j); i := j;
```

Število d je, kot smo rekli, najmanjši skupni delitelj n in k; lahko bi ga torej računali z Evklidovim algoritmom, vendar gre tudi brez tega. Spomnimo se, da v prvi iteraciji naše zunanje zanke, torej pri $z = 0$, obiščemo natanko vse take indekse, ki so večkratniki d; najmanjši neničelni indeks med njimi je potemtakem ravno d sam; lahko si ga torej takrat zapomnimo in ga ne bo treba posebej računati. Zapišimo še implementacijo te rešitve v C++:

```
void Zamakni4(int n, int k)
{
  for (int z = 0, d = n; z < d; ++z)
    // Zamaknimo vse zaboje na mestih oblike a * gcd(n, k) + z.
    for (int i = z; ; )
    {
      // Zaboj na mestu i bomo zamenjali s tistim k mest naprej.
      int j = (i + k) % n;

      // Ko pridemo nazaj na začetni indeks z, končamo.
      if (j == z) break;

      // Pri z = 0 je najmanjši neničelni indeks, ki ga bomo dosegli,
      // ravno gcd(n, k); zapomnimo si ga v d.
      if (z == 0 && j < d) d = j;

      Zamenjaj(i, j); i = j; // Zamenjajmo zaboja na mestih i in j.
    }
}
```

Tudi ta rešitev porabi $O(n)$ časa in $O(1)$ dodatnega pomnilnika.

18. Človeške ribice

Jamski sistem pri tej nalogi ima obliko drevesa, kakršnih smo navajeni tudi iz teorije grafov: sobane so točke oz. vozlišča drevesa, hodniki pa povezave med njimi. Zato bomo uporabljali tudi druge običajne izraze s tega področja, kot so poddrevesa, starši in otroci ipd.

Rekli bomo, da je točka u *polna*, če je zapolnjen že ves prostor v tej točki (v_u enot prostornine), na povezavi od nje do njenega starša (v'_u enot) in če so polni tudi vsi njeni otroci.

Izziv pri tej nalogi je predvsem, kako učinkovito ugotoviti, kam se steka voda, ko jo črpamo v točko a. Če je a polna, moramo iti načeloma od nje gor po drevesu, dokler ne naletimo na prvega takega prednika, ki ni poln — recimo mu b; če a že sama ni polna, pa vzemimo kar $b := a$. Iz b zdaj teče voda navzdol v najglobljega ne-polnega otroka, iz tega spet v najglobljega njegovega ne-polnega otroka in tako naprej, dokler ne pristane v neki taki točki (recimo ji c), ki sama še ni polna, nima pa nobenih ne-polnih otrok.

Težava je, da je lahko drevo precej izrojene oblike in pot od a prek b do c je lahko dolga $O(n)$ korakov. Če se bomo tega lotili preveč naivno, bo časovna zahtevnost naše rešitve na koncu $O(n \cdot k)$. Učinkovito moramo torej znati odgovarjati na naslednji dve vrsti vprašanj: (1) Kateri je najbližji a-jev ne-poln prednik? (2) Katera je naslednja točka v b-jevem poddrevesu, v katero se bo stekala voda?

Vrstnega reda, v katerem se voda nabira v točkah, ni težko določiti po pravilih iz besedila naloge: preden se začne voda nabirati v u, mora zapolniti vse njegove otroke, to pa počne po padajoči globini teh otrok (najprej gre v najglobljega). Tega ni težko zapisati z rekurzivnim postopkom:

globalna spremenljivka: seznam R;

podprogram Rekurzija(u):
  za vsakega u-jevega otroka t po padajoči globini g_t:
    Rekurzija(t);
  dodaj u na konec seznama R;

glavni del postopka:
  R := prazen seznam;
  Rekurzija(*koren*);

Na koncu tega postopka bomo imeli v seznamu R vrstni red, v katerem se voda nabira v točkah. Za vsak u naj bo zdaj z_u položaj točke u v tem vrstnem redu. Odgovor na vprašanje (2) je torej: voda se steka v tistega b-jevega ne-polnega potomca c (mimogrede opozorimo, da štejemo med b-jeve potomce tudi b-ja samega), ki ima najmanjšo vrednost z_c.

Koristno bi bilo torej imeti podatkovno strukturo, v kateri bi hranili ne-polne točke, urejene po z, torej po njihovem položaju v vrstnem redu R. Obenem se mora dati v tej strukturi hitro poiskati, kateri med b-jevimi potomci v njej ima najmanjši z. Na srečo lahko iz postopka, s katerim smo določili vrstni red R in iz njega izhajajoče z-je, vidimo, da točka b in njeni potomci pokrivajo neko strnjeno podzaporedje vrstnega reda R (kajti ko je naš podprogram Rekurzija vstopil v b, je potem najprej obiskal vse b-jeve potomce in jih dodal v R, nato pa je dodal tja še b-ja

samega; vmes nikoli ni izstopil iz b-jevega drevesa in dodajal v R še kakšnih drugih točk, ki bi ležale zunaj tega poddrevesa). Naj bo torej zdaj z'_u minimum vrednosti z po vseh potomcih točke u. Tega ni težko računati rekurzivno (in ta izračun lahko kar vključimo v zgoraj omenjeni rekurzivni postopek, s katerim določimo vrstni red R): za liste je $z'_u = z_u$, za notranja vozlišča pa je $z'_u := \min_t z'_t$ po vseh otrocih t vozlišča u.

V naši poizvedbi po podatkovni strukturi ne-polnih točk lahko torej vprašanje „kateri med b-jevimi potomci ima najmanjši z?“ izrazimo kot „katera izmed točk, ki imajo z na območju $z'_b \leq z \leq z_b$, ima najmanjši z?“ Primerna struktura za naš namen je kakšna od uravnoteženih drevesastih struktur, na primer rdeče-črno drevo (v C++ lahko uporabimo razred `map` iz standardne knjižnice); recimo mu N, ker bomo v njem hranili z-je *n*e-polnih točk. Potem lahko v $O(\log n)$ časa poiščemo v N najmanjši tak element z, ki je $\geq z'_b$ (tak element zagotovo obstaja in pripada enemu od b-jevih potomcev, saj b ni poln, torej bomo v N, če ne drugega, našli b-ja samega).

Razmislimo zdaj še o vprašanju (1), torej kako poiskati najbližjega a-jevega nepolnega prednika b. Če a že sam ni poln, vzamemo za b kar njega in smo končali. Če pa je a poln, poiščimo najprej njegovega zadnjega (najvišje ležečega) polnega prednika, recimo b'; starš tega b' je potem prvi ne-polni prednik, torej naš iskani b. (Lahko se tudi izkaže, da je b' koren jame in starša sploh nima; tedaj je celoten jamski sistem že poln in vanj sploh ni mogoče več dodajati vode.) Možne vrednosti b' so le take točke, ki so polne, nimajo pa polnega starša; to so torej točke, ki bi jih bilo dobro hraniti v nekakšni podatkovni strukturi, kjer bi jih lahko dobili kot odgovor na poizvedbe tipa (1).

Če si mislimo neko táko točko b' in vzamemo za a katerega koli njenega potomca, mora biti odgovor na vprašanje „kateri je zadnji a-jev polni prednik?“ v vsakem primeru ravno b'. Spomnimo se, da v našem vrstnem redu R (ki smo ga vpeljali malo prej) vsi ti a-ji, torej vsi potomci točke b', stojijo skupaj, kot zadnja med njimi pa stoji ravno točka b'. Torej so z-ji teh točk zaporedna cela števila, največji med njimi pa je $z_{b'}$. Imejmo torej še eno rdeče-črno drevo, recimo mu P (ker vsebuje *p*olne točke) in hranimo v njem z-je tistih polnih točk, ki nimajo polnega starša. Ko nas potem zanima najvišji polni prednik polne točke a, moramo le poiskati v P najmanjši tak z, ki je $\geq z_a$. Zanj potem vemo, da mora pripadati pravemu b' (torej vzamemo $b' := R[z]$), kajti izmed točk, ki imajo z nekje vmes med z_a in $z_{b'}$, nobene nimamo v P (vse te točke so namreč potomci točke b' in zato prav gotovo nimajo ne-polnega starša).

Zdaj imamo vse, kar potrebujemo, da lahko opišemo našo rešitev s psevdokodo:

pripravi vrstni red R in z-je (z_u = položaj točke u v R);
dodaj vse ne-polne točke v N;
dodaj v P vse polne točke, ki nimajo polnega starša;
za vsako polnjenje (a, ℓ):
  while $\ell > 0$:
    if a ni polna: $b := a$;
    else:
      b' := najvišji polni prednik a-ja (poizvedba v P);
      if b' je koren **then break;** (* *jama je čisto polna* *)

$b :=$ starš točke b';
$c :=$ ne-polni potomec b-ja z najmanjšim z (poizvedba v N);
$\triangle := \min\{\ell, v_c\}$; zmanjšaj ℓ in v_c za $\triangle$;
if $\triangle > 0$ **and** $v_c = 0$ **then** izpiši, da se je sobana c zdaj zapolnila;
$\triangle := \min\{\ell, v'_c\}$; zmanjšaj ℓ in v'_c za $\triangle$;
if je c zdaj polna **then** pobriši jo iz N, dodaj jo v P
in od tam pobriši njene otroke;

Kakšna je časovna zahtevnost te rešitve? Vsaka iteracija notranje zanke (**while** $\ell > 0$) bodisi zapolni neko točko c bodisi zaključi trenutno polnjenje (ker pade ℓ na 0). Prvo se lahko zgodi le $O(n)$-krat (ker se vsaka točka zapolni največ enkrat), drugo le k-krat (enkrat za vsako polnjenje), torej se izvede največ $O(n + k)$ iteracij te zanke. Pri vsaki iteraciji imamo po največ eno poizvedbo v P in N, kar traja $O(\log n)$ časa; skupaj (po vseh iteracijah) je to $O((n + k) \log n)$. Poleg tega vsako točko največ enkrat dodamo v N, pobrišemo iz N, dodamo v P in pobrišemo iz P, kar je skupaj še $O(n)$ operacij na rdeče-črnih drevesih, ki trajajo vsaka po $O(\log n)$ časa; skupaj $O(n \log n)$. Poleg tega smo morali pri pripravi vrstnega reda R urediti otroke vsake točke po globini, kar pri točki z n_i otroki vzame $O(n_i \log n_i)$ časa; če seštejemo to po vseh točkah, dobimo $O(\sum_i n_i \log n_i) = O(\sum_i n_i \log n) = O(n \log n)$ časa. Vsega skupaj je tako časovna zahtevnost naše rešitve $O((n + k) \log n)$.

Preden si ogledamo konkretno implementacijo v C++, omenimo še eno ali dve malenkosti, na kateri je dobro biti pozoren pri implementaciji. Pri tvorbi vrstnega reda R smo doslej opisali rekurzivni postopek, vendar gre lahko ta rekurzija $O(n)$ nivojev globoko, zato je v praksi varneje, če to rekurzijo predelamo v iteracijo, da ne bo težav s prekoračitvijo prostora na skladu. Naša spodnja rešitev vzdržuje med tem postopkom svoj sklad s pari (u, b), ki ji povedo, da zdaj začenja (če je $b =$ true) oz. zaključuje (če je $b =$ false) obdelavo poddrevesa, ki se začne pri točki u. Ko vstopi v tako poddrevo, doda na sklad zapise za zaključek obdelave b-jevega poddrevesa in za začetek obdelave b-jevih otrok; otroke mora dodati na sklad tako, da pridejo na vrh sklada tisti, kamor voda odteče najprej. Ob zaključku obdelave poddrevesa pa mora naš postopek dodati trenutno točko u v vrstni red R in tudi izračunati z'_b (v spodnji rešitvi vrednosti z' hranimo v vektorju z1). Takrat je tudi primeren trenutek, da izračunamo, ali je u v začetnem stanju jame polna ali ne.

Pri izpisu rezultatov moramo paziti na to, da smo mi definirali točko kot polno šele takrat, ko se zapolni tudi hodnik nad njo (med njo in staršem) — v gornji psevdokodi je to takrat, ko padeta na 0 tako v_c kot tudi v'_c; naloga pa zahteva, da jo izpišemo že takrat, ko se zapolni sama sobana (torej ko pade v_c na 0), četudi je v hodniku nad njo morda še kaj prostora.

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

int main()
{
  int n, k, koren = −1; cin >> n >> k;
```

```
// Preberimo podatke o sobanah in pripravimo za vsako seznam otrok.
vector<int> p(n), g(n), v(n), vp(n); // vhodni podatki
vector<vector<int>> otroci(n);       // seznami otrok posameznih soban
for (int u = 0; u < n; ++u) {
   cin >> p[u] >> g[u] >> v[u] >> vp[u]; --p[u];
   if (p[u] < 0) koren = u; else otroci[p[u]].push_back(u); }

// Oštevilčimo sobane po vrstnem redu polnjenja; soba u = vrstniRed[i] dobi
// številko z[u] = i, minimum z-ja po vseh u-jevih potomcih pa bomo shranili v z1[u].
vector<int> vrstniRed, z(n), z1(n); vrstniRed.reserve(n);
vector<bool> polna(n); // u je polna, če je v[u] = vp[u] = 0 in so vsi otroci polni

// Vrstni red določimo z rekurzivnim postopkom, ki ga simuliramo z iteracijo.
// Na skladu hranimo ukaze (u, b), pri čemer b pove, ali vstopamo v poddrevo u
// ali izstopamo iz njega.
stack<pair<int, bool>> sklad; sklad.emplace(koren, true);
while (! sklad.empty())
{
   auto [u, vstop] = sklad.top(); auto &O = otroci[u];

   // Če smo vstopili v list, bomo takoj spet izstopili iz njega.
   if (O.empty()) vstop = false;

   // Ob vstopu v poddrevo dodamo na sklad ukaze za izstop iz njega
   // in za vstop v njegove otroke. Otroke pred tem uredimo v obratnem
   // vrstnem redu polnjenja (najglobljі pridejo na konec seznama in
   // s tem na vrh sklada, da jih bomo obiskali kot prve).
   if (vstop) {
      sort(O.begin(), O.end(),
           [&] (int x, int y) { return g[x] < g[y] || g[x] == g[y] && x > y; });
      sklad.top().second = false;
      for (int t : O) sklad.emplace(t, true); }

   // Ob izstopu iz poddrevesa dodamo u v vrstni red,
   // izračunamo z1[u] in preverimo, ali je u polna.
   else {
      sklad.pop(); z1[u] = z[u] = vrstniRed.size(); vrstniRed.push_back(u);
      polna[u] = (v[u] == 0 && vp[u] == 0);
      for (int t : O) {
         z1[u] = min(z1[u], z1[t]);
         if (! polna[t]) polna[u] = false; } }
}
// Pripravimo začetno stanje množic „polne“ in „nepolne“.
set<int> polne;    // z-ji polnih soban, ki nimajo polnega starša
set<int> nepolne; // z-ji ne-polnih soban
for (int u = 0; u < n; ++u)
   if (! polna[u]) nepolne.emplace(z[u]);
   else if (! (p[u] >= 0 && polna[p[u]])) polne.emplace(z[u]);

// Preberimo zdaj podatke o vbrizgavanju vode in simulirajmo dogajanje.
while (k-- > 0)
{
   int a, L; cin >> a >> L; --a;
   bool prvi = true; // za presledke pri izpisu
   while (L > 0)
   {
      // Naj bo b najbližji ne-poln a-jev prednik. To je kar a sam,
      // če ni poln; sicer pa moramo v „polne“ poiskati najbolj oddaljenega
      // a-jevega polnega prednika in nato za b vzeti njegovega starša.
```

```
      int b = (! polna[a]) ? a : p[vrstniRed[*polne.lower_bound(z[a])]];
      if (b < 0) break; // celoten jamski sistem je že poln

      // Naj bo c tisti potomec b-ja, v katerega zdaj teče voda (morda kar b sam).
      int c = vrstniRed[*nepolne.lower_bound(z1[b])];

      // Koliko vode odteče v c?
      int dv = min(L, v[c]); v[c] -= dv; L -= dv;
      if (dv > 0 && v[c] == 0) { cout << (prvi ? "" : " ") << (c + 1); prvi = false; }

      // Koliko vode odteče v hodnik med c in njegovim staršem?
      dv = min(L, vp[c]); vp[c] -= dv; L -= dv;

      // Če c še ni polna, mora biti L zdaj 0 in lahko končamo.
      if (v[c] > 0 || vp[c] > 0) break;

      // Sicer se je c zdaj napolnila. Premaknimo jo iz „nepolne“ v „polne“,
      // od tam pa pobrišimo njene otroke.
      polna[c] = true; nepolne.erase(z[c]); polne.emplace(z[c]);
      for (auto u : otroci[c]) polne.erase(z[u]);
    }
    // Če je tu L > 0, je L enot vode ostalo po tistem, ko se je jamski sistem
    cout << endl;                                                // povsem zapolnil.
  }
  return 0;
}
```

19. Cenena konferenca

Vrstni red znanstvenikov v vhodnih podatkih je načeloma lahko poljuben in za našo nalogo ni pomemben, saj bo vsota zavarovalnin enaka ne glede na vrstni red seštevancev. Zato je koristno znanstvenike za začetek urediti naraščajoče po k_i; odslej bomo torej predpostavili, da velja $k_1 \leq k_2 \leq \ldots \leq k_n$.

Če zdaj v mislih pregledujemo znanstvenike v tem vrstnem redu, lahko opazimo, da (če za hip odmislimo morebitno omejitev b) plakate od 1 do k_1 obiščejo vsi znanstveniki; plakate od $k_1 + 1$ do k_2 obiščejo vsi znanstveniki razen prvega; plakate od $k_2 + 1$ do k_3 obiščejo vsi razen prvih dveh znanstvenikov; in tako naprej. V splošnem za vsak i velja, da plakate od vključno $k_{i-1} + 1$ do vključno k_i obiščejo le znanstveniki $i, i + 1, \ldots, n$, torej $n - i + 1$ znanstvenikov. Ta interval plakatov zato k skupni zavarovalni vsoti prispeva znesek $z_i := (k_i - k_{i-1}) \cdot (n - i + 1)$. (Pri $i = 1$ si mislimo $k_0 = 0$, pa bo ta formula delovala tudi tam.) Vsi plakati od 1 do k_i skupaj pa potem prispevajo znesek $Z_i := z_1 + z_2 + \ldots + z_i$.

Recimo zdaj, da hočemo za neki konkreten znesek s določiti, kam postaviti mejo b, da skupna zavarovalna vsota ne bo presegla s. Poiščimo največji i, pri katerem je Z_i še $\leq s$; tam torej velja $Z_i \leq s < Z_{i+1}$. To pomeni, da s plakati od 1 do k_i še ne presežemo zneska s, pač pa ga bomo presegli, če jim nato dodamo še plakate od $k_i + 1$ do k_{i+1}. Plakate na tem intervalu obiščejo znanstveniki od $i + 1$ do n, torej $n - i$ znanstvenikov, torej se za vsak dodatni plakat na tem intervalu naša zavarovalna vsota poveča za $n - i$. Dodamo lahko torej še $\lfloor (s - Z_i)/(n - i) \rfloor$ plakatov s tega intervala, pri naslednjem pa bi zavarovalna vsota že presegla s. Odgovor, ki ga iščemo, je torej $b := k_i + \lfloor (s - Z_i)/(n - i) \rfloor$. Pazimo še na robni primer: če je $s \geq Z_n$, dobimo $i = n$ in zato v naši formuli za b pride do deljenja z 0; pri tako velikih s je denarja dovolj ne glede na to, kakšen b izberemo; zato največji b, po

kakršnem naloga sprašuje, sploh ne obstaja (spodnja implementacija rešitve vrne v takih primerih $b = k_n$).

To, pri katerem i leži s na intervalu $Z_i \leq s < Z_{i+1}$, lahko poiščemo z bisekcijo po $Z_1, \ldots, Z_n$, saj je to zaporedje naraščajoče. To nam vzame pri vsakem s po $O(\log n)$ časa in ker moramo rešiti nalogo za m takih zneskov, bo imela ta rešitev časovno zahtevnost $O(n \log n)$ za urejanje znanstvenikov po k_i in nato $O(m \log n)$ za obdelavo vseh m zneskov, skupaj torej $O((n+m) \log n)$. Oglejmo si še implementacijo te rešitve v C++:

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

vector<int> Konferenca(vector<int> k, const vector<int>& s)
{
  sort(k.begin(), k.end()); // Uredimo znanstvenike po k_i.
  int n = k.size(); vector<int> Z(n);

  // Izračunajmo kumulativne stroške zavarovanja po intervalih.
  for (int i = 0; i < n; ++i)
  {
    // Interval od k[i − 1] + 1 do k[i] obiščejo znanstveniki od i do n − 1.
    int zi = (k[i] − (i == 0 ? 0 : k[i − 1])) * (n − i);    // z_i
    Z[i] = (i == 0 ? 0 : Z[i − 1]) + zi;                     // Z_i
  }

  // Odgovorimo na poizvedbe.
  vector<int> odgovori; odgovori.reserve(s.size());
  for (int sj : s)
  {
    // Poiščimo interval, za katerega je Z[i − 1] ≤ sj < Z[i];
    int i = upper_bound(Z.begin(), Z.end(), sj) − Z.begin();

    // Koliko nam ostane po prvih i − 1 intervalih?
    if (i > 0) sj −= Z[i − 1];

    // Do katerega posterja lahko pridemo s tem denarjem?
    int b = (i > 0) ? k[i − 1] : 0; // prvih i − 1 intervalov
    if (i < n) b += sj / (n − i);  // del i-tega intervala

    odgovori.push_back(b);
  }
  return odgovori;
}
```

Preprostejšo in malo manj učinkovito rešitev dobimo, če namesto z bisekcijo iščemo primeren i kar z zanko po vseh i od 1 do n; tako ima naša rešitev časovno zahtevnost $O(n \cdot m)$.

Elegantna in učinkovita rešitev je tudi ta, da zneske $s_1, \ldots, s_m$ uredimo naraščajoče in se nato hkrati sprehodimo po obeh urejenih seznamih, $Z_1, \ldots, Z_n$ in $s_1, \ldots, s_m$: ko se začnemo ukvarjati s poizvedbo s_i, ni treba iti po seznamu Z-jev od začetka, ampak lahko nadaljujemo kar od tam, kjer smo pri s_{i-1} končali. Tako dobimo rešitev s časovno zahtevnostjo $O(n \log n + m \log m)$.

Razmislimo zdaj o težji različici naloge, pri kateri lahko med poizvedbami s_j tudi dodajamo nove znanstvenike k_i, vendar pa so še vedno vse poizvedbe in dodajanja znani vnaprej. V tem primeru lahko še vedno pripravimo (že na začetku) urejen

seznam $k_1 \leq k_2 \leq \ldots \leq k_n$, v katerem naj bodo k-ji vseh znanstvenikov (duplikate pobrišimo), ne le tistih, ki so prisotni že od začetka, ampak tudi tistih, ki jih bomo dodali šele kasneje. Pravzaprav si lahko predstavljamo, da začnemo s prazno množico znanstvenikov in potem občasno kakšnega dodamo (nekatere morda že pred prvo poizvedbo). Naš seznam $k_1, \ldots, k_n$ nam spet razdeli plakate na intervale oblike $P_i := \{k_{i-1} + 1, \ldots, k_i\}$, le da se zdaj število znanstvenikov, ki obiščejo določen interval, počasi povečuje. Dolžino območja P_i označimo z $d_i = |P_i| = k_i - k_{i-1}$; število znanstvenikov, ki bi ga radi prehodili, pa s c_i (sem štejmo le tiste znanstvenike, ki smo jih že dodali, ne pa tistih, ki jih morda šele bomo!). Podobno kot pri prvotni različici naloge je skupna cena zavarovanja za i-ti interval enaka $z_i = d_i \cdot c_i$, skupna cena za prvih i intervalov pa $Z_i = z_1 + z_2 + \ldots + z_i$.

Težava je zdaj v tem, da ko pride nov znanstvenik, ki bi rad obiskal recimo prvih t intervalov, se povečajo vrednosti $c_1, \ldots, c_t$ za 1, zato se povečajo $z_1, \ldots, z_t$ in čisto vse vsote $Z_1, \ldots, Z_n$. Če bi hoteli vse te vrednosti vzdrževati eksplicitno, bi nam vzelo pri vsakem novem znanstveniku $O(n)$ časa, da jih popravimo, in naša rešitev bi bila zelo neučinkovita. Potrebujemo način, da spremembe, ki prizadenejo po več zaporednih intervalov (kot je na primer povečanje vrednosti $c_1, \ldots, c_t$ za 1), zapišemo le na enem mestu (ali vsaj na majhnem številu mest) in to tako, da jih bomo kasneje lahko pravilno upoštevali pri odgovarjanju na poizvedbe.

Razdelimo v mislih intervale $P_1, \ldots, P_n$ na dve skupini, levo (prvih nekaj intervalov) in desno (vsi preostali intervali); nato vsako skupino spet na dve manjši skupini in tako naprej, dokler ni nazadnje vsak interval sam v svoji skupini; tako si lahko mislimo, da smo nad intervali zgradili binarno drevo. V korenu drevesa je vozlišče, ki predstavlja vse intervale skupaj, v listih pa je n vozlišč, ki predstavljajo vsako po en posamični interval. Podrobnosti tega, kako točno delimo skupine intervalov na manjše podskupine, niti niso tako zelo pomembne, važno je le, da drevo na koncu ne bo globoko več kot $O(\log n)$ nivojev.[26] Takšna drevesasta struktura nam bo omogočila, da spremembe, ki vplivajo na daljše skupine intervalov, zapišemo v višje ležečih vozliščih (ki jih je malo), njihov vpliv na nižje ležeča (ki jih je veliko) pa računamo sproti med spuščanjem po drevesu.

Vsako vozlišče tega drevesa, recimo u, torej predstavlja neko strnjeno skupino intervalov; taka strnjena skupina je zato tudi sama interval oblike $P_u = \{r_u - d_u + 1, \ldots, r_u - 1, r_u\}$, pri čemer smo z r_u označili desni rob intervala (številko najbolj desnega posterja na njem), z d_u pa dolžino intervala (število posterjev na njem). Naj bo c_u število takih znanstvenikov, ki bi radi obiskali celoten interval P_u, med njimi pa naj bo γ_u število tistih, ki ne bi radi obiskali celotnega intervala u-jevega starša. V korenu, ki starša nima, je $c_u = \gamma_u$; niže v drevesu pa, če ima u starša p, je $c_u = \gamma_u + c_p$. Vrednosti c_u zato ne bomo hranili eksplicitno, pač pa jih bomo računali sproti pri spuščanju po drevesu; hranili pa bomo le vrednosti γ_u. To je prikladno zato, ker je treba, ko pride nov znanstvenik, povečati γ_u za 1 pri največ enem vozlišču na vsakem nivoju drevesa. Oglejmo si zametek tega postopka v psevdokodi; vrednosti, ki jih hranimo eksplicitno, bomo pisali z oglatimi oklepaji,

[26] Elegantna možnost je na primer tale: oštevilčimo v mislih nivoje drevesa od $R = \lceil \log_2 n \rceil$ (koren drevesa) do 0 (listi); in i-to vozlišče na nivoju r naj predstavlja skupino, ki jo tvorijo intervali P_j za $(i-1) \cdot 2^r < j \leq i \cdot 2^r$. Potem vemo, da sta otroka tega vozlišča $(2i-1)$-vo in $(2i)$-to vozlišče na nivoju $r - 1$. Ker ima drevo tako pravilno strukturo, ga lahko predstavimo kar z zaporedjem $R + 1$ tabel, po eno za vsako plast.

npr. $r[u]$ namesto r_u; podobno smo z $levi[u]$ in $desni[u]$ označili otroka vozlišča u.

```
podprogram Dodaj(k):    (* osnutek *)
  (* Dodajamo znanstvenika, ki želi obiskati posterje od 1 do k. *)
  u := koren drevesa;
  while k < r[u]:
    if k ≤ r[levi[u]] then u := levi[u];
                      else: γ[levi[u]] += 1; u := desni[u];
  (* Tu velja k = r[u]. *)
  γ[u] += 1;
```

Naj bo Z_u skupna cena vseh prehojenih poti od začetka razstave do konca P_u. (Tiste, ki so krajše, se pač končajo prej; tiste, ki so daljše, v mislih odrežemo za posterjem r_u.) Če bi znali računati takšne Z_u, bi lahko na poizvedbe odgovarjali takole:

```
funkcija Poizvedba(s):    (* osnutek *)
  u := koren drevesa;
  if s ≥ Z[u] then
    return ∞; (* Denarja je dovolj za vse, omejitve b sploh ne potrebujemo. *)
  while u ni list:
    (* Tu velja, da je s dovolj denarja za vse intervale levo od P_u,
       torej za prvih r_u − d_u posterjev, ne pa tudi za celoten interval P_u, torej
       do vključno r_u-tega posterja; velja s < Z_u. *)
    if s ≥ Z[levi[u]] then u := desni[u] else u := levi[u];
  (* Tu velja gornja invarianta in u je list. *)
  b := nekje na območju od r_u − d_u do r_u − 1;
  return b;
```

Ceno Z_u lahko v mislih razdelimo na L_u (ceno vseh prehojenih poti levo od P_u) in z_u (ceno prehojenih poti znotraj P_u). Torej je $Z_u = L_u + z_u$. Če ima u otroka x (levega) in y (desnega), je $z_u = z_x + z_y$; poleg tega je $L_x = L_u$ in $L_y = L_x + z_x$. Če poznamo vrednosti z, lahko L-je in Z-je računamo sproti ob spuščanju po drevesu. Pravkar omenjeno psevdokodo lahko zdaj zapišemo natančneje:

```
funkcija Poizvedba(s):
  u := koren drevesa;
  c := γ[u]; L := 0;
  Z_u := z[u];                                                        (†)
  if s ≥ Z_u then return ∞;
  while u ni list:
    (* Tu je L = L_u in c = c_u; in L_u ≤ s < Z_u. *)
    Z_x := L + z[levi[u]];                                            (‡)
    if s ≥ Z_x then u := desni[u]; L := Z_x else u := levi[u];
    c := c + γ[u];
  (* Še vedno je L = L_u, c = c_u, L_u ≤ s < Z_u in u je list. *)
  return (r[u] − d[u]) + ⌊(s − L)/c⌋;
```

Na koncu zanke vemo dovolj, da lahko izračunamo primerno mejo b: za posterje levo od P_u (ki jih je $r_u - d_u$) potrebujemo L enot denarja, ostane nam še $s - L$ enot;

za vsak dodatni poster na območju P_u pa potrebujemo po c_u enot, ker se želi po intervalu P_u sprehoditi c_u znanstvenikov.

Opisani postopek se opira na vrednosti z_u; razmislimo torej, kako jih lahko računamo. Znanstvenike, ki prispevajo k strošku zavarovanja z_u (torej tiste, katerih želena pot sega vsaj delno na območje P_u; ali še drugače: to so tisti, katerih k je $> r_u - d_u$), lahko razdelimo na tiste, ki hočejo območje P_u prehoditi v celoti (torej ki imajo $k \geq r_u$; takih znanstvenikov je c_u in torej k z_u prispevajo znesek $c_u \cdot d_u$), in tiste, ki ga hočejo prehoditi le delno; prispevek teh slednjih k z_u pa imenujmo β_u.

Če je u list, pokriva en sam interval in je nemogoče, da bi ga kdo želel prehoditi le delno, zato je pri listih $\beta_u = 0$. Če pa je u notranje vozlišče, ima dva otroka, recimo x (levega) in y (desnega); pot, ki pokrije del intervala u (ne pa celega), pokrije bodisi del intervala x in nič intervala y (take poti prispevajo β_x) ali pa pokrije celoten x in del intervala y (takih poti je γ_x, vsaka od njih prispeva strošek d_x v levem delu, vse skupaj pa še β_y v desnem delu). Za notranja vozlišča je torej $\beta_u = \gamma_x \cdot d_x + \beta_x + \beta_y$.

Pri dodajanju novega znanstvenika v drevo se nekaj vozliščem poveča γ, zato se prednikom takih vozlišč poveča β. Koliko dela bomo imeli, da pregledamo vse te prednike in popravimo njihove β? Podprogram DODAJ, s katerim pri dodajanju znanstvenika povečujemo γ nekaterih vozlišč, je do teh vozlišč prišel s spuščanjem po drevesu in je torej vse njihove prednike že obiskal. Lahko ga torej dopolnimo tako, da ob spuščanju tudi popravlja vrednosti β:

podprogram DODAJ(k):
  $u :=$ koren drevesa;
  while $k < r[u]$:
    (* *Novi znanstvenik želi prehoditi P_u le delno,*
      od levega konca (poster št. $r_u - d_u + 1$) do vključno k. *)
    $\beta[u]$ += $k - (r[u] - d[u])$;
    if $k \leq r[levi[u]]$ **then** $u := levi[u]$;
                else $\gamma[levi[u]]$ += 1; $u := desni[u]$;
  (* *Tu velja $k = r[u]$. Ta znanstvenik torej želi prehoditi P_u v celoti,*
    zato k β_u ne prispeva. *)
  $\gamma[u]$ += 1;

S to različico postopka DODAJ bomo imeli v drevesu vedno prave vrednosti γ_u in β_u. V nazadnje opisani različici funkcije POIZVEDBA pa bi morali spremeniti le še to, da bo funkcija računala vrednosti z sama po prej omenjeni formuli $z_u = c_u \cdot d_u + \beta_u$. Vrstica (†) tako postane:

  $Z_u := c \cdot d[u] + \beta[u]$;

Vrstica (‡) pa postane:

    $c_x := \gamma[x] + c$;
    $z_x := c_x \cdot d[x] + \beta[x]$;
    $Z_x := L + z_x$;

Za vsako vozlišče u moramo eksplicitno hraniti vrednosti d_u, r_u, γ_u, β_u (in kazalca na otroka, če nimamo drevesa predstavljenega s skladovnico tabel), ostalo pa lahko naša dva postopka DODAJ in POIZVEDBA računata sproti. Videli smo, da se oba samo spuščata po drevesu in ker je le-to globoko $O(\log n)$ nivojev, je časovna zahtevnost

vsake operacije le $O(\log n)$. Če imamo n znanstvenikov in m poizvedb, je časovna zahtevnost cele rešitve $O((n+m)\log n)$, nič slabša kot pri prvotni, lažji različici naloge.

Za konec nam je ostala še najtežja različica naloge, pri kateri moramo na poizvedbe odgovarjati sproti, ko še ne vemo, katere znanstvenike bo kasneje treba dodati v našo podatkovno strukturo. Še vedno lahko uporabimo našo pravkar opisano rešitev z binarnim drevesom, le da zdaj intervalov ne poznamo vnaprej. Začeti bomo morali z drevesom, ki ima eno samo vozlišče, ki predstavlja neskončen interval $\{1, 2, \ldots\}$. Pri dodajanju novega znanstvenika se potem lahko zgodi, da je njegov k različen od dosedanjih, zato bo treba enega od obstoječih intervalov razcepiti na dva dela. Recimo, da je bil to interval P_u, pri čemer je bil u doslej list drevesa; novi znanstvenik želi torej obiskati posterje od 1 do k za neki k z območja $r_u - d_u < k < d_u$. Ustvarimo torej dve novi vozlišči in ju dodajmo v drevo kot otroka vozlišča u: levi otrok (recimo mu x) naj pokriva interval od $r_u - d_u + 1$ do k, desni otrok (recimo mu y) pa od $k+1$ do r_u.

Spomnimo se, da smo β_u definirali kot ceno poti, prehojenih na območju P_u, vendar le od tistih znanstvenikov, ki nočejo prehoditi celotnega P_u (torej ki imajo $k < r_u$). V našem primeru, ko smo ob dodajanju novega znanstvenika razcepili vozlišče u, je dosedanja vrednost $\beta[u]$ še vedno dobra. Za u-jeva nova otroka pa nam definicija β pove, da morata oba dobiti $\beta[x] = \beta[y] = 0$, kajti noben znanstvenik ne želi prehoditi njunih intervalov le delno: novi znanstvenik želi iti točno do k, torej prehoditi celoten levi interval in nič od desnega; vsi dosedanji znanstveniki pa so hoteli prehoditi ali oba v celoti ali nič od njiju, saj bi drugače interval P_u razcepili že prej.

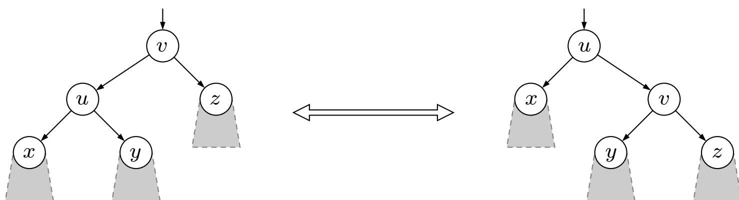
Glede γ_u se spomnimo, da smo ga definirali kot število znanstvenikov, ki bi radi obiskali celoten P_u, ne pa tudi celotnega intervala njegovega starša. Pri razcepu vozlišča u na x in y to pomeni, da je dosedanja vrednost $\gamma[u]$ še vedno dobra; levi otrok ima $\gamma[u] = 1$ (pravkar dodani znanstvenik je edini, ki želi obiskati P_x, ne pa tudi celega P_u, saj vsi predhodni znanstveniki bodisi obiščejo P_u v celoti bodisi ga sploh ne obiščejo); desni otrok pa ima $\gamma[y] = 0$ (to je splošna lastnost vseh desnih otrok v drevesu — nemogoče je, da bi kdo želel obiskati celoten interval desnega otroka, ne pa tudi celotnega intervala starša, saj se oba končata z istim posterjem: $r[u] = r[desni[u]]$).

Zdaj torej načeloma znamo dodajati znanstvenike tudi pri tej različici naloge in za nova vozlišča v drevesu izračunati vse, kar moramo vedeti o njih. V postopku za poizvedovanje po drevesu pa nam ni treba spreminjati ničesar.

Ta rešitev je že skoraj dobra, ne pa še čisto. Videli smo, da se pri dodajanju novega znanstvenika lahko neki list v drevesu razcepi in se spremeni v notranje vozlišče, pod seboj pa dobi dva otroka. Pri naslednjem znanstveniku bi se lahko razcepil eden od teh dveh novih otrok in tako naprej; po n dodajanjih bi bilo lahko drevo globoko že $O(n)$ nivojev in tudi cena vsakega naslednjega dodajanja, pa tudi vsake poizvedbe, bi bila lahko $O(n)$. Temu rečemo, da se je drevo *izrodilo* — njegova globina ni več $O(\log n)$, kot bi si želeli.

Da to preprečimo, moramo pri dodajanju novih znanstvenikov drevo po potrebi malo preurediti in ga spet uravnotežiti. Za to lahko uporabljamo čisto enake prijeme, kot se uporabljajo pri rdeče-črnih drevesih, AVL-drevesih in drugih podobnih

podatkovnih strukturah. V podrobnosti se tu ne bomo spuščali, saj so opisane v številnih učbenikih in na mnogih spletnih straneh. Spodobi pa se razmisliti, ali bomo lahko ob uravnoteževanju drevesa primerno popravljali tudi podatke, ki jih moramo vzdrževati o vsakem vozlišču, torej d_u, r_u, γ_u in β_u. Tako pri rdeče-črnih kot pri AVL-drevesih je osnovna operacija uravnoteževanja tako imenovana *rotacija*, pri kateri neko vozlišče pride na mesto svojega starša, bivši starš pa postane njegov otrok:



Sivi liki predstavljajo poddrevesa, ki se začnejo pri vozliščih x, y in z ter se brez sprememb premaknejo skupaj s temi vozlišči.

Razmislimo, kako moramo popraviti naše podatke pri rotaciji v desno (kjer bivši levi otrok pride na mesto svojega starša, slednji pa postane njegov desni otrok), primer rotacije v levo pa prepustimo bralcu za vajo. Da ne bo zmede, bomo nove vrednosti naših atributov označili s črticami; tako je na primer d_u vrednost atributa d pri vozlišču u pred rotacijo, d'_u pa po rotaciji.

Za desna krajišča intervalov in njihove dolžine je stvar enostavna: pri x, y in z se nič ne spremenijo; desno krajišče intervala pri nekem vozlišču je po definiciji enako kot pri njegovem desnem otroku, dolžina pa je vsota dolžin otrok, tako da imamo $r'_u = r'_v = r_z$, $d'_v = d_y + d_z$ in $d'_u = d_x + d'_v$.

Naj bo $N(a, b)$ število znanstvenikov, ki imajo k na območju $a \leq k < b$. Potem, če ima t starša t', je po definiciji $\gamma_t = N(r_t, r_{t'})$. V našem primeru to pomeni, da je bilo pred rotacijo $\gamma_x = N(r_x, r_y)$, $\gamma_y = \gamma_z = 0$, $\gamma_u = N(r_y, r_z)$ in $\gamma_v = N(r_z, r_p)$, če je p vozlišče, ki je bilo pred rotacijo starš v-ja, po rotaciji pa je starš u-ja. (Če p-ja ni, ker je bil v koren, si mislimo $r_p = \infty$.) Po rotaciji pa imamo $\gamma'_x = N(r_x, r_z) = \gamma_x + \gamma_u$, $\gamma'_y = N(r_y, r_z) = \gamma_u$, $\gamma'_z = \gamma'_v = 0$ in $\gamma'_u = N(r_z, r_p) = \gamma_v$.

Glede β vidimo, da se ta po definiciji nanaša le na poti znotraj intervala posamezne točke (in le za tiste znanstvenike, ki tega intervala ne prehodijo v celoti); zato se β_x, β_y in β_z nič ne spremenijo, saj se tudi intervali teh točk niso spremenili. Točka u ima po rotaciji enak interval, kot ga je imela v pred rotacijo, zato je $\beta'_u = \beta_v$; za v pa lahko izračunamo novi β po formuli, ki smo jo že videli: $\beta'_v = \gamma'_y d'_y + \beta'_y + \beta'_z$, kar je naprej enako $\gamma_u d_y + \beta_y + \beta_z$.

Tako torej vidimo, da tudi po rotaciji lahko popravimo vse podatke, ki jih naš postopek potrebuje. Tako lahko zagotovimo, da bo globina drevesa ostala $O(\log n)$ in časovna zahtevnost naše rešitve bo na koncu še vedno taka kot prej: po $O(\log n)$ časa za vsako dodajanje in za vsako poizvedbo.

20. Transakcijski računi

Najprej vpeljimo nekaj oznak: $d = 6$ naj bo dolžina številk računov brez kontrolne števke; številke računov si bomo predstavljali kot vektorje oblike $\mathbf{r} = (r_1, \ldots,$

r_d, r_{d+1}); pri i-tem nakazilu v vhodnem seznamu naj bo $\mathbf{r}_i$ račun prejemnika, z_i pa znesek; r_{ij} (za $j = 1, \ldots, d+1$) je torej j-ta števka v $\mathbf{r}_i$; kontrolno vsoto celotnega seznama imenujmo $\mathbf{s}$, njeno j-to števko pa s_j. Zapis $x \equiv y$ nam bo pomenil, da imata x in y enak ostanek po deljenju z 10 (če sta vektorja, pa naj velja to za istoležne komponente obeh).

Naloga pravi, da je zadnja števka vsake številke računa v bazi (in zato tudi v seznamu nakazil) kontrolna: torej velja $r_{i,d+1} = \sum_{i=1}^{d} r_{ij} \bmod 10$. Za kontrolno vsoto na koncu seznama nakazil pa velja $\mathbf{s} = \sum_{i=1}^{m} \mathbf{r}_i \bmod 10$ (pri tem zapisu z vektorji si moramo seveda predstavljati, da seštevamo istoležne komponente vektorjev $\mathbf{r}_i$ in na koncu od vsake vsote obdržimo ostanek po deljenju z 10). Pri $j = d+1$ to slednje pomeni $s_{d+1} = \sum_{i=1}^{m} r_{i,d+1} \bmod 10 = \sum_{i=1}^{m} \sum_{j=1}^{d} r_{ij} \bmod 10 = \sum_{j=1}^{d} \sum_{i=1}^{m} r_{ij} \bmod 10 = \sum_{j=1}^{d} s_j \bmod 10$. To pomeni, da če pri spreminjanju seznama poskrbimo za to, da bo imela njegova kontrolna vsota enake vrednosti $s_1, \ldots, s_d$ kot pri prvotnem seznamu, bo potem tudi vrednost s_{d+1} enaka kot v prvotnem seznamu. To pomeni, da se nam z $(d+1)$-vo števko pri tej nalogi sploh ni treba ubadati, niti pri številkah računov niti pri kontrolni vsoti seznama. Odslej bomo o obojem govorili, kot da ima le po d števk.

Opazimo tudi, da potrebujemo številke $\mathbf{r}_i$ iz vhodnega seznama le za izračun $\mathbf{s}$-ja, po tistem pa lahko vse $\mathbf{r}_i$ pozabimo, saj jih lahko poljubno spremenimo, dokler ima spremenjeni seznam še vedno kontrolno vsoto $\mathbf{s}$.

Kar se tiče kontrolnih vsot, je vseeno, v kakšnem vrstnem redu se pojavljajo računi v seznamu; pomembno je le to, kolikokrat se pojavi kateri račun. V našem primeru je potem smiselno razporediti račune tako, da dobrodelna organizacija dobi najvišjih nekaj nakazil (torej tistih z največjimi z_i), za ostale račune pa je vseeno, kako so razporejeni med nakazila. Ostane nam torej le še vprašanje, kolikokrat naj uporabimo kateri račun, da bo na koncu kontrolna vsota seznama še vedno $\mathbf{s}$ in da bo račun dobrodelne organizacije uporabljen čim večkrat.

V bazi imamo n računov, ki jih lahko uporabljamo; i-temu od njih recimo $\mathbf{a}_i$, njegovi j-ti števki pa a_{ij}. Recimo, da račun $\mathbf{a}_i$ uporabimo pri k_i nakazilih. Rešitev $\mathbf{k} = (k_1, k_2, \ldots, k_n)$ je torej veljavna, če je $\sum_{i=1}^{n} k_n = m$ in če je $\mathbf{s} \equiv \sum_{i=1}^{n} k_i \mathbf{a}_i$.

Če enega od k_i povečamo ali zmanjšamo za 10 ali za nek večkratnik 10, se tudi v vsoti $\sum_i k_i \mathbf{a}_i$ vsaka komponenta poveča ali zmanjša za nek večkratnik števila 10, torej se njen ostanek po deljenju z 10 nič ne spremeni; rešitev ima še vedno enako kontrolno vsoto kot prej. To pomeni, da če imamo veljavno rešitev, pri kateri je kak k_i (za $i \geq 2$) večji ali enak 10, lahko ta k_i zmanjšamo za 10 in namesto tega povečamo k_1 za 10; kontrolna vsota bo enaka kot prej, dobrodelna organizacija pa bo dobila več nakazil. V nadaljevanju se torej lahko omejimo na rešitve, pri katerih je $0 \leq k_i \leq 9$ za vse $i = 2, \ldots, n$. Poleg tega iz omejitve $\sum_{i=1}^{n} k_n = m$ sledi, da je k_1 enolično določen, ko enkrat izberemo $k_2, \ldots, k_n$. Pri iskanju rešitve pravzaprav torej iščemo le vrednosti $(k_2, \ldots, k_n) \in \{0, 1, \ldots, 9\}^{n-1}$.

Takih $(n-1)$-teric je veliko, 10^{n-1}; toda na srečo nam ni treba pregledati vseh. Pri posamezni $(n-1)$-terici, recimo $\mathbf{k} = (k_2, \ldots, k_n)$, je pomembno predvsem dvoje: (1) skupno število nakazil $v(\mathbf{k}) := \sum_{i=2}^{n} k_i$, ki vpliva na to, koliko nakazil bo dobila dobrodelna organizacija — namreč $k_1 = m - v(\mathbf{k})$; in (2) kontrolne vsote $\mathbf{t}(\mathbf{k}) = \sum_{i=2}^{n} k_i \mathbf{a}_i \bmod 10$, ki vplivajo na to, ali bo rešitev sploh veljavna; ko bomo namreč tem nakazilom dodali še k_1 nakazil na račun $\mathbf{a}_1$, moramo priti do kontrolne vsote $\mathbf{s}$.

Tako imamo pogoj $\mathbf{s} \equiv \mathbf{t}(\mathbf{k}) + k_1\mathbf{a}_1$.

Če se več rešitev $\mathbf{k}$ ujema v $v(\mathbf{k})$ in $\mathbf{t}(\mathbf{k})$, nam med njimi ni treba razlikovati, saj so si za naše namene popolnoma enakovredne. Kaj pa, če se dve rešitvi ujemata samo v kontrolni vsoti $\mathbf{t}$? Če imata različno v, bosta imeli tudi različna k_1 in zato se lahko zgodi, da bo pogoj $\mathbf{s} \equiv \mathbf{t}(\mathbf{k}) + k_1\mathbf{a}_1$ pri eni izpolnjen, pri drugi pa ne. Toda če se v dveh rešitev razlikuje za neki večkratnik 10, potem se bosta tudi pripadajoča k_1 razlikovala za večkratnik 10 in izraza $\mathbf{t}(\mathbf{k}) + k_1\mathbf{a}_1$ se bosta v vsaki komponenti razlikovala za večkratnik 10, torej bosta imeli obe rešitvi enako kontrolno vsoto; pogoj $\mathbf{s} \equiv \mathbf{t}(\mathbf{k}) + k_1\mathbf{a}_1$ bo zato izpolnjen ali pri obeh ali pri nobeni. Takšni rešitvi sta si torej s tega vidika enakovredni, za nas pa je seveda boljša tista z manjšo $v(\mathbf{k})$, ker nam bo omogočila več nakazil preusmeriti dobrodelni organizaciji.

Koristno je torej, če si za vsako možno $\mathbf{t} \in \{0, \ldots, 9\}^d$ in za vsak možni ostanek p od 0 do 9 zapomnimo najmanjšo vrednost $v(\mathbf{k})$ po vseh takih rešitvah $\mathbf{k}$, ki imajo $\mathbf{t}(\mathbf{k}) = \mathbf{t}$ in $v(\mathbf{k}) \bmod 10 = p$. To je namreč vse, kar moramo vedeti o taki skupini rešitev. Lepo pri tej ugotovitvi je, da je možnih kombinacij $(\mathbf{t}, p)$ le 10^{d+1} (v našem primeru je to 10 milijonov, ker imamo $d = 6$), kar je veliko manj od vseh možnih rešitev, ki jih je 10^{n-1}.

Lahko si predstavljamo, da pregledujemo prostor stanj oblike $(\mathbf{t}, p)$. Če za neko rešitev $\mathbf{k}$ velja $\mathbf{t}(\mathbf{k}) = \mathbf{t}$ in $v(\mathbf{k}) \bmod 10 = p$, bomo rekli, da rešitev $\mathbf{k}$ *pripada* stanju $(\mathbf{t}, p)$. Če zdaj vzamemo poljubno rešitev $\mathbf{k}$, ki pripada stanju $(\mathbf{t}, p)$, in v njej povečamo k_i za 1 (torej preusmerimo še eno nakazilo več na račun $\mathbf{a}_i$), nastane rešitev $\mathbf{k}'$, ki pripada stanju $((\mathbf{t} + \mathbf{a}_i) \bmod 10, (p+1) \bmod 10)$ — novo stanje je torej neodvisno od tega, s *katero* $\mathbf{k}$ iz starega stanja smo začeli.

Na začetku vemo, da je pri $\mathbf{k} = \mathbf{0}$ dosegljivo stanje $(\mathbf{0}, 0)$, in sicer že z 0 porabljenimi nakazili: $v(\mathbf{k}) = 0$. Nato lahko za vsak i od 2 do n devetkrat ponovimo naslednje: v vsakem trenutno dosegljivem stanju poskusimo za 1 povečati k_i in izračunajmo, v katero stanje nas to pripelje; če tega stanja doslej še nismo dosegli ali pa smo ga zdaj dosegli z manjšo $v(\mathbf{k})$ od najmanjše doslej znane, si to novo rešitev zapomnimo. (Rešitve z $v(\mathbf{k}) \geq m$ lahko tudi ignoriramo, saj iz njih kasneje tako ali tako ne bo mogoče dobiti veljavne rešitve, ker bi bil k_1 tam negativen.)

Ob koncu tega postopka poznamo vsa stanja, ki jih je mogoče doseči s poljubno kombinacijo vrednosti $k_2, \ldots, k_n \in \{0, \ldots, 9\}$. Za vsako od njih tudi vemo, kakšno je najmanjše število uporabljenih nakazil $v(\mathbf{k})$, s katerim ga je mogoče doseči. Za vsako tako stanje, recimo $(\mathbf{t}, p)$, lahko potem izračunamo, koliko nakazil ostane na voljo za dobrodelno organizacijo: $k_1 = n - v(\mathbf{k})$; če je to ≥ 0, je rešitev načeloma smiselna, njena kontrolna vsota pa bo $\mathbf{t} + k_1\mathbf{a}_1 \bmod 10$; če je to enako $\mathbf{s}$, je rešitev veljavna (ustreza vsem našim pogojem). Med tako odkritimi veljavnimi rešitvami moramo na koncu vrniti tisto z največjo k_1 (oz. natančneje povedano, pri tem k_1 moramo potem vrniti vsoto k_1 najvišjih zneskov izmed $z_1, z_2, \ldots, z_m$).

Zapišimo naš postopek preiskovanja prostora stanj še s psevdokodo:

h := prazna razpršena tabela, v kateri bomo kot ključe uporabljali pare $(\mathbf{t}, p)$, pripadajoča vrednost pa bo najmanjša $v(\mathbf{k})$, s katero nam je doslej uspelo doseči stanje $(\mathbf{t}, p)$; za pripadajočo vrednost h ključu $(\mathbf{t}, p)$ bomo uporabljali oznako $h[\mathbf{t}, p]$, če pa tega ključa ni v h, si mislimo $h[\mathbf{t}, p] = \infty$;
dodaj v h ključ $(\mathbf{0}, p)$ s pripadajočo vrednostjo 0;
for $i := 2$ **to** m:

```
  L := seznam vseh ključev, ki so trenutno v h;
  za vsako stanje (t, p) v L:
    for d := 1 to 9:
      t' := (t + d a_i) mod 10; p' := (p + d) mod 10;
      v' := h[t, p] + d; if v' > m then break;
      if h[t', p] ≤ v' then break;                                        (⋆)
      h[t', p] := v'; (* dodaj nov ključ ali spremeni pripadajočo vrednost pri
                                                          že obstoječem *)
k1* := 0;
za vsak ključ (t, p) iz h:
  k1 := m − h[t, p];
  if k1 > k1* and s = (t + k1 a1) mod 10 then k1* := k;
vrni vsoto k1* največjih vrednosti izmed z_1, ..., z_m;
```

V vrstici ($\star$) takoj prekinemo najbolj notranjo zanko, če pridemo v stanje $(\mathbf{t}', p')$, za katero že od prej poznamo enako dobro ali boljšo rešitev. Prepričajmo se, da to res smemo narediti. (1) Če je bila ta rešitev v h že prej, preden smo se začeli ukvarjati s trenutnim i, to pomeni, da imamo tisto stanje $(\mathbf{t}', p')$ tudi v L in bomo prej ali slej naredili iz njega 9 korakov naprej z računom $\mathbf{a}_i$ (ali pa smo to morda celo že naredili); mi pa bomo zdajle, ko smo do njega prišli iz $(\mathbf{t}, p)$ v d korakih, naredili od njega le še $9-d$ korakov naprej, torej ne bomo dosegli ničesar novega, pa tudi boljših rešitev ne bomo dobivali, saj že pri $(\mathbf{t}', p')$ naša nova rešitev v' ni boljša od dosedanje $h[\mathbf{t}', p]$. (2) Druga možnost je, da smo rešitev $h[\mathbf{t}', p']$ dosegli šele pri trenutnem i, vendar pri nekem drugem začetnem stanju, recimo po $\tilde{d}$ korakih iz $(\tilde{\mathbf{t}}, \tilde{p})$; imamo torej $(\mathbf{t}', p') = (\mathbf{t}, p) + d(\mathbf{a}_1, 1) \bmod 10$ in $(\mathbf{t}', p') = (\tilde{\mathbf{t}}, \tilde{p}) + \tilde{d}(\mathbf{a}_1, 1) \bmod 10$; torej $(\tilde{\mathbf{t}}, \tilde{p}) = (\mathbf{t}, p) + (d - \tilde{d})(\mathbf{a}_1, 1) \bmod 10$; torej sta d in $\tilde{d}$ gotovo različna, saj bi bili sicer stanji $(\mathbf{t}, p)$ in $(\tilde{\mathbf{t}}, \tilde{p})$ enaki. (2.1) Če je $\tilde{d} > d$, to pomeni, da se iz $(\tilde{\mathbf{t}}, \tilde{p})$ po $\tilde{d} - d$ korakih pride v $(\mathbf{t}, p)$; torej so rešitve, do katerih pridemo po $10 - (\tilde{d} - d)$ ali več korakih iz $(\mathbf{t}, p)$, hkrati tudi rešitve, do katerih pridemo po 10 ali več korakih iz $(\tilde{\mathbf{t}}, \tilde{p})$; take rešitve pa ne morejo biti optimalne, saj bi se dalo potem 10 korakov z računom $\mathbf{a}_i$ izrezati in imeti enako kontrolno vsoto, vendar manjšo $v(\mathbf{k})$, tako da bi ostalo več prostora za nakazila dobrodelni organizaciji. Torej nima smisla, da sploh delamo 9 korakov iz $(\mathbf{t}, p)$, kajti pri stanjih, ki bi jih pri tem dosegli in ki jih nismo dosegli ob delanju 9 korakov iz $(\tilde{\mathbf{t}}, \tilde{p})$, gotovo ne bomo dobili optimalnih rešitev. (2.2) Ostane še možnost $\tilde{d} < d$, toda ta je sploh nemogoča: ta namreč pomeni, da smo morali, še preden smo zdaj po d korakih iz $(\mathbf{t}, p)$ dosegli $(\mathbf{t}', p')$, že po $\tilde{d}$ korakih doseči $(\tilde{\mathbf{t}}, \tilde{p})$ in bi že takrat opazili, da imamo v h že vsaj tako dobro ali boljšo rešitev, torej bi zanko prekinili že tam. □

Naloge so sestavili: zbiratelj — Urban Duh; rudarji — Tomaž Hočevar; tabela števil — Gregor Kikelj; predstavitve, seznama, stave — Vid Kocijan; človeške ribice, cenena konferenca — Filip Koprivec; pravokotnik iz kvadratov, napredovanje števil, kodiranje besedila — Mitja Lasič; študentski servis — Matija Lokar; največji XOR — Tim Poštuvan; urnik, film, prefiksna in postfiksna oblika, transakcijski računi — Jure Slak; pandemija, tretji tir — Jasna Urbančič; zamik — Janez Brank.

NASVETI ZA MENTORJE O IZVEDBI ŠOLSKEGA TEKMOVANJA IN OCENJEVANJU NA NJEM

[Naslednje nasvete in navodila smo poslali mentorjem, ki so na posameznih šolah skrbeli za izvedbo in ocenjevanje šolskega tekmovanja. Njihov glavni namen je bil zagotoviti, da bi tekmovanje potekalo na vseh šolah na približno enak način in da bi ocenjevanje tudi na šolskem tekmovanju potekalo v približno enakem duhu kot na državnem.—*Op. ur.*]

Tekmovalci naj pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom; ocenjevanje teh odgovorov poteka v vsakem primeru tako, da jih pregleda in oceni mentor (in ne npr. tako, da bi se poskušalo izvorno kodo, ki so jo tekmovalci napisali v svojih odgovorih, prevesti na računalniku in pognati na kakšnih testnih podatkih). Pri reševanju si lahko tekmovalci pomagajo tudi z literaturo in/ali zapiski, ni pa mišljeno, da bi imeli med reševanjem dostop do interneta ali do kakšnih datotek, ki bi si jih pred tekmovanjem pripravili sami. Čas reševanja je omejen na 180 minut.

Nekatere naloge kot odgovor zahtevajo program ali podprogram v kakšnem konkretnem programskem jeziku, nekatere naloge pa so tipa „opiši postopek". Pri slednjih je načeloma vseeno, v kakšni obliki je postopek opisan (naravni jezik, psevdokoda, diagram poteka, izvorna koda v kakšnem programskem jeziku, ipd.), samo da je ta opis dovolj jasen in podroben in je iz njega razvidno, da tekmovalec razume rešitev problema.

Glede tega, katere programske jezike tekmovalci uporabljajo, naše tekmovanje ne postavlja posebnih omejitev, niti pri nalogah, pri katerih je rešitev v nekaterih jezikih znatno krajša in enostavnejša kot v drugih (npr. uporaba perla ali pythona pri problemih na temo obdelave nizov).

Kjer se v tekmovalčevem odgovoru pojavlja izvorna koda, naj bo pri ocenjevanju poudarek predvsem na vsebinski pravilnosti, ne pa na sintaktični. Pri ocenjevanju na državnem tekmovanju zaradi manjkajočih podpičij in podobnih sintaktičnih napak odbijemo mogoče kvečjemu eno točko od dvajsetih; glavno vprašanje pri izvorni kodi je, ali se v njej skriva pravilen postopek za rešitev problema. Ravno tako ni nič hudega, če npr. tekmovalec v rešitvi v C-ju pozabi na začetku **#include**ati kakšnega od standardnih headerjev, ki bi jih sicer njegov program potreboval; ali pa če podprogram main() napiše tako, da vrača **void** namesto **int**.

Pri vsaki nalogi je možno doseči od 0 do 20 točk. Od rešitve pričakujemo predvsem to, da je pravilna (= da predlagani postopek ali podprogram vrača pravilne rezultate), poleg tega pa je zaželeno tudi, da je učinkovita (manj učinkovite rešitve dobijo manj točk).

Če tekmovalec pri neki nalogi ni uspel sestaviti cele rešitve, pač pa je prehodil vsaj del poti do nje in so v njegovem odgovoru razvidne vsaj nekatere od idej, ki jih rešitev tiste naloge potrebuje, naj vendarle dobi delež točk, ki je približno v skladu s tem, kolikšen delež rešitve je našel.

Če v besedilu naloge ni drugače navedeno, lahko tekmovalčeva rešitev vedno predpostavi, da so vhodni podatki, s katerimi dela, podani v takšni obliki in v okviru takšnih omejitev, kot jih zagotavlja naloga. Tekmovalcem torej načeloma ni treba pisati rešitev, ki bi bile odporne na razne napake v vhodnih podatkih.

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. Križci in krožci

- Neučinkovite rešitve, ki bi za niz dolžine n porabile več kot $O(n)$ časa, naj dobijo največ 13 točk, če so drugače pravilne.
- V naši rešitvi smo za preverjanje tega, ali je število križcev v nizu enako številu krožcev, uporabili eno spremenljivko, ki računa razliko med obema številoma. Za enako dobro naj šteje tudi rešitev, ki šteje križce posebej in krožce posebej, vsake v svoji spremenljivki, in ju na koncu primerja med seboj.
- V naši rešitvi smo si pri pregledovanju niza zapomnili prejšnja dva znaka v spremenljivkah, za enako dobro pa naj šteje tudi rešitev, ki vsakič znova pogleda v prejšnja dva elementa niza (ali pa enega prejšnjega in enega naslednjega ipd.). Če pride na začetku ali koncu niza do kakšnih napak pri indeksih (npr. v C++ branje znaka na indeksu -1 ali kaj podobnega), naj se rešitvi zaradi tega odšteje dve točki.
- Trije pogoji, navedeni v besedilu naloge, so mišljeni kot približno enakovredni in preverjanje vsakega od njih je vredno eno tretjino točk.
- Naloga ne predpisuje posebej, kaj naj funkcija naredi z rezultatom; lahko ga vrne (npr. kot logično vrednost tipa **bool** ali kaj podobnega), lahko ga izpiše na zaslon ipd. — vse te možnosti naj štejejo za enako dobre, glavno je, da je iz rešitve razvidno, da je do (pravilnega) rezultata v nekem trenutku res prišla.

2. Kovanci

- Besedilo naloge pravi, da je kupčkov lahko veliko. Rešitve, ki imajo eksponentno časovno zahtevnost, npr. ker delajo z rekurzijo in si ne shranjujejo že izračunanih rezultatov, naj dobijo največ 10 točk, če so sicer pravilne.
- Naša rešitev pri računanju funkcije f poleg trenutne vrednosti hrani le dve prejšnji, tako da porabi le $O(1)$ pomnilnika. Za enako dobre naj se štejejo tudi rešitve, ki porabijo $O(n)$ pomnilnika, ker shranjujejo vse že izračunane rezultate, četudi večine izmed njih ne bodo več potrebovale.

3. Taksi

- Pri tej nalogi pričakujemo večinoma rešitve s časovno zahtevnostjo $O(n \cdot m)$; take naj dobijo največ 12 točk, če so sicer pravilne. Rešitve z manjšo časovno zahtevnostjo, kot sta na primer rešitvi v času $O(n \log n + m \log m)$ ali $O((n + m) \log n)$, naj dobijo vse točke, če so sicer pravilne.

- Naloga pravi, da je mreža lahko velika. Če bi kakšna rešitev slučajno imela časovno zahtevnost, ki je odvisna od velikosti mreže (za kar sicer ni kakšnega posebej dobrega razloga) namesto le od n in m, naj dobi največ 6 točk.
- Besedilo naloge ne daje nobenih zagotovil glede tega, da so vsi možni položaji central v vhodnih podatkih različni, toda če tekmovalčeva rešitev predpostavi, da so različni, naj se ji zaradi tega ne odšteva točk (če je drugače pravilna).
- Lahko se zgodi, da ležita kakšna stranka in kakšen od možnih položajev centrale na isti točki (x, y); ali pa, da leži na isti točki več strank. Če bi kakšna rešitev predpostavila, da se to ne more zgoditi, in bi zato v takih primerih vračala napačne rezultate, naj se ji zaradi tega odšteje dve točki.

4. Preusmerjanje

- Rešitve, ki porabijo $O(n^2)$ časa, npr. ker na vsakem koraku znova pregledajo celoten vhodni seznam, da ugotovijo, ali s trenutne strani vodi kakšna preusmeritev (in kam), naj dobijo največ 10 točk, če so drugače pravilne.
- V naših rešitvah smo objavili dve različici, eno s tabelo (oz. vektorjem) in eno z razpršeno tabelo (oz. slovarjem). Pri točkovanju naj se oboje šteje za enako dobro, ravno tako tudi rešitve, ki bi namesto razpršenih tabel uporabile kakšne uravnotežene drevesaste strukture (npr. map in set iz C++ove standardne knjižnice).
- Rešitvam, ki ne shranjujejo podatkov o že obiskanih straneh, pač pa v primeru cikla vedno naredijo n ali več korakov, preden ugotovijo, da obstaja cikel, naj se zaradi te neučinkovitosti odšteje dve točki.
- Če rešitev sploh ne ugotovi obstoja cikla, pač pa se na njem tudi sama zacikla v neskončni zanki, naj se ji zaradi tega odšteje polovico točk, kolikor bi jih sicer dobila glede na časovno zahtevnost postopka.
- Za morebitne drobne napake pri indeksiranju (npr. povezane s tem, ali so številke strani od 1 do n ali od 0 do $n - 1$) naj se odšteje največ dve točki.
- Naloga ne predpisuje podrobnosti tega, v kakšni obliki naš podprogram dobi vhodne podatke, zato lahko tekmovalčeva rešitev podrobnosti tega določi sama. V naših primerih smo na primer enkrat imeli vektor struktur, enkrat smo uporabili pythonov tip tuple, dalo bi se imeti tudi dva seznama (recimo s in na, ki bi nam povedala, da obstaja preusmeritev s strani s[i] na stran na[i]) in podobno.

5. Odstranjevanje črk

- Pri tej nalogi je poudarek predvsem na opažanju, da je koristno besede pregledovati od krajših proti daljšim. Rešitve, ki tega ne počno in poskušajo mogoče z rekurzijo na vse možne načine brisati znake enega za drugim ter imajo zaradi tega eksponentno časovno zahtevnost, naj dobijo največ 5 točk, če so drugače pravilne.

- Naša rešitev shranjuje ugodne besede v razpršeno tabelo; za enako dobro naj velja tudi, če tekmovalčeva rešitev uporabi kakšno uravnoteženo drevesasto strukturo (npr. razred set iz C++ove standardne knjižnice). Če pa bi kakšna rešitev hranila ugodne besede v navadnem seznamu ali tabeli ali čem podobnem, kjer bi torej vsako preverjanje, ali je neka beseda ugodna, trajalo $O(n)$ časa, naj se ji zaradi tega odšteje 4 točke.

- Naša rešitev porabi pri nizu s dolžine k načeloma $O(k^2)$ časa, da sestavi vse možne nize, ki nastanejo iz s z brisanjem ene črke. To bi se dalo z nekaj pazljivosti zmanjšati na $O(k)$, enako tudi pri poizvedbah v razpršeno tabelo, če bi npr. uporabili Rabin-Karpove hash kode; vendar ni mišljeno, da bi se rešitve tekmovalcev ukvarjale s čim takim.

- Lahko se zgodi, da v vhodnem seznamu ni nobene ugodne besede. Naloga ne predpisuje natančno, kaj naj rešitev naredi v takem primeru, zato naj se za pravilno šteje karkšno koli obnašanje, iz katerega je razvidno, da rešitev ni našla ugodne besede. Lahko na primer vrne prazen niz, vrednost None (v pythonu) ali **null** (v javi ipd.), indeks -1 (če vrača indeks niza v vhodnem seznamu), lahko izpiše, da niza ni, lahko sproži izjemo in podobno.

- V naši rešitvi v C++ smo vektor besede prenašali po vrednosti namesto po referenci, tako da dobimo svojo kopijo vektorja in lahko besede v njem uredimo (naraščajoče po dolžini), ne da bi se to pri klicatelju kaj poznalo. Rešitvam, ki namesto tega prenašajo vektor po referenci in ga morebiti spremenijo tako, da to spremembo vidi tudi klicatelj, naj se tega ne šteje v slabo in naj se jim zaradi tega ne odšteva 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. Križci in krožci	lažja naloga v prvi skupini
2. Kovanci	srednje težka naloga v prvi ali lažja v drugi skupini
3. Taksi	težka naloga v prvi ali srednje težka v drugi skupini[27]
4. Preusmerjanje	srednje težka naloga v prvi ali lažja v drugi skupini
5. Odstranjevanje črk	težja naloga v prvi ali lažja v drugi skupini

[27] Opomba: težavnost te naloge je zelo odvisna od tega, kako točkujemo rešitve odvisno od njihove učinkovitosti. Rešitev s časovno zahtevnostjo $O(nm)$ je zelo preprosta, do učinkovitejših rešitev pa je znatno težje priti. Ker smo predvideli 12 točk (od dvajsetih) že za rešitev v času $O(nm)$, lahko štejemo to nalogo za srednje težko.

Če torej na primer nek tekmovalec reši le eno ali dve lažji nalogi, pri ostalih pa ne naredi (skoraj) ničesar, to še ne pomeni, da ni primeren za udeležbo na državnem tekmovanju; pač pa je najbrž pametno, če na državnem tekmovanju ne gre v drugo ali tretjo skupino, pač pa v prvo.

Podobno kot prejšnja leta si tudi letos želimo, da bi čim več tekmovalcev s šolskega tekmovanja prišlo tudi na državno tekmovanje in da bi bilo šolsko tekmovanje predvsem v pomoč tekmovalcem in mentorjem pri razmišljanju o tem, v kateri težavnostni skupini državnega tekmovanja naj kdo tekmuje.

Zadnja leta na državnem tekmovanju opažamo, da je v prvi skupini izrazito veliko tekmovalcev v primerjavi z drugo in tretjo, med njimi pa je tudi veliko takih z zelo dobrimi rezultati, ki bi prav lahko tekmovali tudi v kakšni težji skupini. Mentorjem zato priporočamo, naj tekmovalce, če se jim zdi to primerno, spodbudijo k udeležbi v zahtevnejših skupinah.

REZULTATI

Tabele na naslednjih straneh prikazujejo vrstni red vseh tekmovalcev, ki so sodelovali na letošnjem tekmovanju. Poleg skupnega števila doseženih točk je za vsakega tekmovalca navedeno tudi število točk, ki jih je dosegel pri posamezni nalogi. V prvi in drugi skupini je mogoče pri vsaki nalogi doseči največ 20 točk, v tretji skupini pa največ 100 točk.

Načeloma se v vsaki skupini podeli dve prvi, dve drugi in dve tretji nagradi in letos so se rezultati izšli tako, da od tega ni bilo treba odstopati. 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 230) in smo jih letos podelili osem. 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 126 bronastih priznanj).

V tabelah na naslednjih straneh so prejemniki nagrad označeni z „1“, „2“ in „3“ v prvem stolpcu, prejemniki priznanj pa z „Z“ (zlato) in „S“ (srebrno).

PRVA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1Z	1	Anton Luka Šijanec	2	Gimnazija Bežigrad	20	20	20	20	17	97
1Z	2	Luka Peršolja	4	Gimnazija Vič	17	20	18	20	18	93
2S	3	Tilen Juričan	2	ZRI	19	20	20	10	19	88
2S		Tim Thuma	2	Vegova Ljubljana	20	20	12	18	18	88
3S	5	Jaša Knap	4	Gimnazija Bežigrad	20	20	20	10	16	86
3S		Luka Stražišar	3	Gimnazija Vič	19	20	19	10	18	86
S	7	Adrian Sebastian Šiška	2	Vegova Ljubljana	16	20	20	10	18	84
S	8	Gabrijel Pflaum	4	Gimnazija Bežigrad	20	15	20	10	17	82
S	9	Luka Leskovšek	2	Vegova Ljubljana	20	17	20	9	15	81
S		Nikola Brković	3	Gimnazija Bežigrad	20	15	20	10	16	81
S	11	Galileo Pellizer	4	STŠ Koper	20	16	20	7	17	80
S	12	Gašper Dobrovoljc	2	Vegova Ljubljana	15	19	20	10	15	79
S		Žan Ambrožič	2	Gimnazija Kranj	20	18	18	8	15	79
S	14	Lan Bajželj	1	Vegova Ljubljana	20	18	11	10	18	77
S	15	Luka Papež	3	Gimnazija Vič	20	20	17	0	17	74
S	16	Gorazd Kotnik	1	ŠC Velenje, ERŠ	18	20	19	0	16	73
S		Tine Zaletelj	2	Gimnazija Vič	20	18	10	9	16	73
S	18	Anja Laharnar	4	STPŠ Trbovlje	20	13	10	9	20	72
S	19	Filip Gerdina	4	ŠC Celje, Gim. Lava	20	15	7	9	19	70
S		Tim Nahtigal	2	ŠC N. mesto, SEŠTG	18	20	8	8	16	70
S	21	Rene Klement	4	II. gimnazija Maribor	20	15	18	0	16	69
S	22	Jaka Kovač	1	Vegova Ljubljana	18	15	20	0	15	68
S	23	Aljaž Travnik	3	Vegova Ljubljana	15	10	20	5	17	67
S	24	Domen Lisjak	3	Gimnazija Bežigrad	20	20	14	0	12	66
S		Tomo Testen	4	ŠC N. mesto, SEŠTG	14	9	15	17	11	66
S	26	Ožbej Pavc	4	Škof. klas. gimn. Lj.	19	18	10	0	18	65
S	27	Samo Hribar	4	Gimnazija Bežigrad	20	9	20	0	13	62
S		Špela Gačnik	1	Gimnazija Bežigrad	15	18	20	0	9	62
S	29	Aljaž Kokol	4	SERŠ Maribor	20	18	10	0	12	60
S		Tim Rezelj	4	ŠC N. mesto, SEŠTG	7	18	8	10	17	60
S	31	Gregor Gračnar	4	SC Celje, SŠ za KER	13	14	12	0	17	56
S	32	Urban Krepel	1	ŠC Velenje, ERŠ	17	18	17	0	3	55
S	33	Matevž Bizjak	3	ŠC Nova Gorica	17	12	20	0	5	54
S	34	Matjaž Pogačnik	3	Gimnazija Bežigrad	18	15	6	0	14	53
S	35	Gašper Lukman	3	SERŠ Maribor	10	10	18	0	14	52
S		Leon Sovič	4	SERŠ Maribor	18	18	2	9	5	52
S		Matic Bernot	3	STPŠ Trbovlje	15	10	10	0	17	52
S		Rebeka Stres	3	Škof. klas. gimn. Lj.	19	13	10	0	10	52
S	39	Martin Jereb	3	Gimnazija Vič	20	16	0	0	15	51
S		Teo Lah	4	SERŠ Maribor	16	10	9	0	16	51
S		Peter Jereb	9	ZRI	12	10	20	9	0	51

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
S	42	Žan Starašinič	4	ŠC Novo mesto, SEŠTG	15	10	10	0	15	50
S	43	Goro Modic	3	Gimnazija Vič	18	20	0	9	0	47
S		Tit Šober	3	ŠC Novo mesto, SEŠTG	18	10	5	0	14	47
S		Tjaš Paradiž	3	ŠC Celje, Gimnazija Lava	18	12	17	0	0	47
S	46	Luka Logar	3	ŠC Celje, Gimnazija Lava	20	0	0	10	16	46
S		Tim Hrovat	2	Vegova Ljubljana	16	8	0	9	13	46
	48	Matija Pilko	4	I. gimnazija v Celju	17	15	0	0	13	45
		Tomi Božak	3	ŠC Celje, Gimnazija Lava	0	18	18	9	0	45
		Žiga Terbovc	3	ŠC Celje, Gimnazija Lava	15	18	0	0	12	45
	51	Dejan Pajsar	5	ŠC Kranj, STŠ Kranj	17	10	17	0	0	44
		Urban Dopudja	3	ŠC Kranj, STŠ Kranj	15	10	2	0	17	44
		Žan Skopec	4	SŠTS Šiška	7	19	18	0	0	44
	54	Tian Ključanin	3	Gimnazija Vič	9	3	13	0	16	41
	55	Lucijan Škof	9	ZRI	0	17	15	0	8	40
	56	Marino Vuk	3	SERŠ Maribor	20	2	2	0	15	39
	57	Anže Prevodnik	4	STŠ Koper	15	10	10	0	3	38
		Domen Brcar	3	ŠC Novo mesto, SEŠTG	15	10	0	0	13	38
		Žan Hribar	2	STPŠ Trbovlje	14	5	7	0	12	38
	60	Andraž Velušček	8	ZRI	0	20	0	0	17	37
		Kevin Poredoš	4	ŠC Novo mesto, SEŠTG	18	0	1	0	18	37
		Tevž Peče	1	ZRI + Gimnazija Vič	19	18	0	0	0	37
		Tilen Gašparič	2	STPŠ Trbovlje	15	14	0	0	8	37
		Tit Podhraški	3	ŠC Celje, Gimnazija Lava	17	0	0	20	0	37
		Vid Kranjec	4	Gimnazija Murska Sobota	10	1	8	10	8	37
	66	Mark Škof	1	SŠTS Šiška	18	18	0	0	0	36
	67	Nik Javor	1	Gimnazija Bežigrad	20	0	0	0	15	35
	68	Vanja Ivačić	3	Vegova Ljubljana	14	14	5	0	0	33
	69	Domen Korenini	1	Gimnazija Vič	19	10	0	0	3	32
		Miha Mirt	3	ŠC Nova Gorica	17	0	5	10	0	32
	71	Arja Ela Hvala	1	Gimnazija Vič	14	9	0	0	8	31
		Nejc Zalokar	8	ZRI	11	0	20	0	0	31
	73	Nik Vodovnik	2	ZRI	0	20	0	10	0	30
	74	Hana Perman	9	ZRI	15	14	0	0	0	29
		Luka Wernig	4	Škof. klas. gimn. Lj.	20	9	0	0	0	29
	76	Žan Seničar	4	ŠC Celje, SŠ za KER	11	1	4	0	12	28
	77	Anže Kejžar	1	Vegova Ljubljana	8	10	3	0	4	25
		Barbara Kastelic	4	ŠC Novo mesto, SEŠTG	15	0	10	0	0	25
		Kevin Jerebica	1	STŠ Koper	12	13	0	0	0	25
	80	Danijel Tomić	2	STPŠ Trbovlje	10	10	3	0	1	24
	81	Enej Breskvar	9	ZRI	2	3	17	0	0	22
		Jan Mušič	4	STŠ Koper	7	0	15	0	0	22

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
83	Vid Jurkovič	1	Gimnazija Vič	7	10	1	0	2	20
84	Niko Pozderec	2	SPTŠ Murska Sobota	16	3	0	0	0	19
85	Marko Đukič	1	ŠC Novo mesto, SEŠTG	1	10	4	3	0	18
86	Matevž Terziev	3	ŠC Kranj, STŠ Kranj	0	0	0	0	14	14
	Monika Simičak	2	Gimnazija Vič	8	2	0	0	4	14
88	Gregor Špan	3	ŠC Celje, Gimnazija Lava	0	1	0	0	12	13
	Klaudija Jakše	3	ŠC Novo mesto, SEŠTG	3	8	2	0	0	13
	Robi Mudri	3	ŠC Nova Gorica	12	0	1	0	0	13
91	Ela Roš	3	Gimnazija Murska Sobota	5	1	6	0	0	12
92	Blaž Osredkar	1	ŠC Velenje, ERŠ	4	0	1	0	1	6
	Enej Tinauer	1	ZRI	6	0	0	0	0	6
	Filip Lupscha	2	SPTŠ Murska Sobota	6	0	0	0	0	6
95	Martin Murko	1	ZRI + Gimnazija Vič	0	0	3	0	0	3
96	Enej Fonda	2	ZRI	0	0	0	0	0	0
	Jakob Fir	4	Vegova Ljubljana	0	0	0	0	0	0
	Tim Strnad	1	ZRI + Gimnazija Vič	0	0	0	0	0	0

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1Z	1	Jakob Kralj	2	Gimnazija Vič	19	20	18	5	20	82
1Z		Jakob Žorž	1	ZRI	12	20	15	15	20	82
2S	3	Jošt Smrtnik	2	Gimnazija Vič	20	20	5	17	12	74
2S	4	Anže Hočevar	3	Gimnazija Vič	10	19	18	5	20	72
3S	5	Janez Ignacij Jereb	4	ZRI	14	8	17	18	10	67
3S	6	Tim Tisak	3	Vegova Ljubljana	16	9	18	12	7	62
S	7	Petja Furlan	5	ŠC Kranj, STŠ Kranj	12	20	15	12	0	59
S	8	Bor Pangeršič	4	Vegova Ljubljana	11	18	13	15	0	57
S	9	Lara Stamać	1	ZRI	4	6	10	16	17	53
S	10	Jure Pospišil	4	II. gimnazija Maribor	15	15	10	12	0	52
S		Žiga Bradaš	5	SŠTS Šiška	17	20	10	5	0	52
S		Žiga Kralj	3	Vegova Ljubljana	14	9	10	16	3	52
S	13	Aljoša Vertot	4	SPTŠ Murska Sobota	14	20	5	11	0	50
S		Oskar Rotar	1	ZRI	14	19	0	17	0	50
S	15	Jaka Velkaverh	4	Gimnazija Vič	10	9	13	14	0	46
S		Lovro Lotrič	2	Gimnazija Kranj + ZRI	15	5	8	18	0	46
S	17	Eva Juvanc	4	ZRI	16	8	2	18	0	44
S		Luka Urbanc	1	ZRI	9	20	15	0	0	44
S	19	Andrej Matos	3	Vegova Ljubljana	10	9	5	16	0	40
	20	Filip Trplan	3	Gimnazija Vič	11	20	8	0	0	39
		Kristjan Komloši	4	ŠC Kranj, Str. gimn.	13	12	2	12	0	39
		Matic Kovač	2	ZRI	10	12	3	14	0	39
		Vida Mlinar	2	Zavod sv. Frančiška Saleškega, Gimnazija Želimlje	8	9	5	12	5	39
	24	Nejc Mihelčič	2	Gimnazija Vič	12	9	2	14	0	37
	25	Denis Balant	2	ŠC Velenje, Gimnazija	5	5	5	10	3	28
		Miha Govedič	3	II. gimnazija Maribor	3	6	0	0	19	28
	27	Anej Repnik	3	ŠC Ravne na Kor., SŠ	10	0	2	15	0	27
		Lenart Frankovič	2	ŠC Velenje, Gimnazija	9	8	10	0	0	27
	29	David Krajnc	3	ŠC Ravne na Kor., SŠ	7	1	2	14	0	24
	30	Aljaž Marn	2	ZRI	10	3	0	8	0	21
	31	Dominik Brezovšek	1	ŠC Celje, SŠ za KER	10	4	3	2	0	19
	32	Grega Potočnik	4	ŠC Ravne na Kor., SŠ	10	5	2	0	0	17
		Žiga Kovačič	3	II. gimnazija Maribor	10	0	7	0	0	17
	34	Maj Zabukovnik	1	ŠC Celje, SŠ za KER	12	1	2	0	0	15
	35	Peter Lekše	3	Škof. klas. gimn. Lj.	9	0	0	0	0	9
		Vid Ošep	1	ZRI + Gimnazija Vič	0	9	0	0	0	9
	37	Matic Dremelj	2	ZRI	0	7	0	0	0	7
	38	Matija Pajenk	3	ŠC Ravne na Kor., SŠ	3	1	1	1	0	6

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj)					
					1	2	3	4	5	∑
1Z	1	Benjamin Bajd	3	ZRI	97	100	100	100	97	494
1Z	2	Matija Likar	3	II. gimn. Maribor	97	54	89	69	50	359
2Z	3	Jakob Schrader	4	ZRI	100	100	20	79		299
2Z	4	Patrik Žnidaršič	4	Gimnazija Vič	94	91	17		24	226
3S	5	Lan Sevčnikar	4	II. gimn. Maribor	65	30	17	57	27	196
3S	6	Domen Hočevar	4	Gimn. Novo mesto	97	47	14		30	188
S	7	Lovro Sikošek	3	Gimnazija Brežice	100	57	0			157
S	8	Brest Lenarčič	1	ŠC Rogaška Slatina	60	94				154
S	9	Mihael Golob	4	Vegova Ljubljana	11	100	14		0	125
S	10	Domen Kastelic	4	ZRI	20	97				117
S	11	Matej Kralj	3	Gimnazija Vič	88		20		0	108
S	12	Nadezhda Komarova	3	ZRI	60	20			27	107
S	13	Luka Lonec	4	II. gimn. Maribor	60	0	8		17	85
S	14	Tadej Tomažič	4	Vegova Ljubljana	60	0	20			80
	15	Tjaž Valentinčič	4	ŠC Nova Gorica	4	54	20			78
	16	Klemen Klopčič	1	Gimn. Bežigrad	14	20	30			64
	17	Ella Potisek	4	ZRI	57		5			62
	18	Daniel Blažič	4	ZRI	51					51
	19	Tilen Tomšič	4	ŠC Nova Gorica	20	7	20			47
	20	Lan Vrčkovnik	3	ŠC Velenje, Gimn.	17	0	10	0	7	34
	21	Aljaž Vetrih	3	ŠC Velenje, Gimn.	17	0	10			27
	22	Filip Štamcar	2	ZRI	17		2		0	19
	23	Alen Leban	4	ŠC Nova Gorica	17	0				17
	24	Bor Brudar	2	ŠC N. mesto, SEŠTG	0	4	10		0	14
	25	Blaž Špacapan	4	ŠC Nova Gorica	0	0				0
		Daniel Bartolič	4	ŠC Nova Gorica	0					0
		Miha Lazić	4	ŠC Nova Gorica		0			0	0
		Tim Cvetko	3	Gimnazija Brežice	0	0	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	982,4
2	Vegova Ljubljana	797
3	II. gimnazija Maribor	596
4	Gimnazija Bežigrad	384,4
5	ŠC Novo mesto, SEŠTG	254,4
6	ŠC Kranj, STŠ Kranj	220
7	SERŠ Maribor	215
8	ŠC Celje, Gimnazija Lava	208
9	STPŠ Trbovlje	199
10	SŠTS Šiška	184
11	ŠC Nova Gorica	174
12	Gimnazija Kranj	171
13	STŠ Koper	165
14	Škofijska klasična gimnazija Ljubljana	164
15	ŠC Celje, SŠ za KER	152
16	ŠC Velenje, Gimnazija	146,6
17	ŠC Ravne na Koroškem, Srednja šola	136
18	ŠC Velenje, ERŠ	134
19	SPTŠ Murska Sobota	125
20	Gimnazija Novo mesto	112,8
21	Gimnazija Brežice	94,2
22	ŠC Rogaška Slatina	92,4
23	ŠC Kranj, Strokovna gimnazija	78
	Zavod sv. Frančiška Saleškega, Gimnazija Želimlje	78
25	Gimnazija Murska Sobota	49
26	I. gimnazija v Celju	45

NAGRADE

Za nagrado so najboljši tekmovalci vsake skupine prejeli naslednjo strojno opremo in knjižne nagrade:

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Anton Luka Šijanec	telefon Samsung Galaxy S20FE
1	1	Luka Peršolja	telefon Samsung Galaxy S20FE
1	2	Tilen Juričan	telefon Realme 7 Pro
1	2	Tim Thuma	telefon Samsung Galaxy A12
1	3	Jaša Knap	telefon Samsung Galaxy A12
1	3	Luka Stražišar	miška Razer DeathAdder V2
2	1	Jakob Kralj	telefon Samsung Galaxy S20FE Dasgupta *et al.: Algorithms*
2	1	Jakob Žorž	telefon Samsung Galaxy S20FE Dasgupta *et al.: Algorithms*
2	2	Jošt Smrtnik	telefon Realme 7 Pro Dasgupta *et al.: Algorithms*
2	2	Anže Hočevar	telefon Realme 7 Pro
2	3	Janez Ignacij Jereb	telefon Samsung Galaxy A12
2	3	Tim Tisak	miška Razer DeathAdder V2
3	1	Benjamin Bajd	telefon Samsung Galaxy S20FE Raspberry Pi 4 model B Steven S. Skiena: *The Algorithm Design Manual*
3	1	Matija Likar	telefon Samsung Galaxy S20FE Raspberry Pi 4 model B Steven S. Skiena: *The Algorithm Design Manual*
3	2	Jakob Schrader	telefon Realme 7 Pro Steven S. Skiena: *The Algorithm Design Manual*
3	2	Patrik Žnidaršič	telefon Realme 7 Pro
3	3	Lan Sevčnikar	telefon Samsung Galaxy A12
3	3	Domen Hočevar	telefon Samsung Galaxy A12
Off-line naloga — Pokrajina iz kock			
	2	Domen Hočevar	Raspberry Pi 4 model B
	4	Gregor Kikelj	Raspberry Pi 4 model B

SODELUJOČE ŠOLE IN MENTORJI

Šola	Mentorji
II. gimnazija Maribor	Luka Lonec, Mitja Osojnik, Lan Sevčnikar
Gimnazija Bežigrad	Andrej Šuštaršič
Gimnazija Brežice	Tea Habinc
Gimnazija Kranj	Mateja Žepič
Gimnazija Murska Sobota	Romana Zver
Gimnazija Novo mesto	Barbara Strnad
Gimnazija Vič	Dušan Bajec, Klemen Bajec, Marina Trost
I. gimnazija v Celju	Luka Zlatečan
Srednja elektro-računalniška šola Maribor (SERŠ)	Slavko Nekrep, Manja Sovič Potisk
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Dominik Letnar
Srednja šola tehniških strok Šiška	Tom Kamin
Srednja tehniška in poklicna šola Trbovlje (STPŠ)	Uroš Ocepek
Srednja tehniška šola Koper	Andrej Florjančič, Senka Sabotin
Šolski center Celje, Gimnazija Lava	Karmen Kotnik
Šolski center Celje, Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Boštjan Lubej, Timej Pirš, Davor Zupanc
Šolski center Kranj, Srednja tehniška šola	Miha Baloh
Šolski center Kranj, Strokovna gimnazija	Gašper Strniša
Šolski center Nova Gorica	Aljaž Gec, Marko Marčetić, Tomaž Mavri, Barbara Pušnar
Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija (SEŠTG)	Simon Vovko, Albert Zorko
Šolski center Ptuj	Franc Vrbančič
Šolski center Ravne na Koroškem, Srednja šola Ravne	Gorazd Geč

Šolski center Rogaška Slatina	Jože Vajdič
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Miran Zevnik
Šolski center Velenje, Gimnazija	Ivan Jovan
Škofijska klasična gimnazija Šentvid	Helena Starc Grlj, Gašper Žajdela
Vegova Ljubljana	Marko Kastelic, Melita Kompolšek, Nataša Makarovič, Aleš Volčini, Darjan Toth
Zavod sv. Frančiška Saleškega, Gimnazija Želimlje	Benjamin Tomažič
Zavod za računalniško izobraževanje (ZRI), Ljubljana	

REZULTATI CERC 2021

Ker smo letos organizirali srednjeevropsko študentsko tekmovanje v računalništvu (CERC 2021) pri nas v Ljubljani, objavljamo v našem biltenu še rezultate tega tekmovanja. Naloge so na str. 28–44, rešitve pa na str. 116–156.

	Ekipa	Št. rešenih nalog	Čas
1	Marcin Martowicz, Adam Górkiewicz, Anadi Agrawal (U. v Wrocławu)	8	16:08:50
2	Krzysztof Potępa, Bartosz Podkanowicz, Krzysztof Pióro (Jag. u.)	8	21:42:44
3	Krzysztof Ziobro, Jacek Salata, Grzegorz Gawryał (Jag. u.)	8	22:35:06
4	Rafał Łyżwa, Kacper Kluk, Jakub Kądziołka (U. v Varšavi)	7	20:36:35
5	Arkadiusz Czarkowski, Michał Staniewski, Tomasz Nowak (U. v Varšavi)	6	10:14:48
6	Rafał Pyzik, Jan Klimczak, Justyna Jaworska (Jag. u.)	6	13:13:44
7	Kacper Topolski, Maksym Zub, Andrii Orap (Jag. u.)	6	13:33:06
8	Adam Zyzik, Marcin Knapik, Krzysztof Boryczka (U. v Wrocławu)	6	13:38:57
9	Attila Gáspár, Péter Gyimesi, Péter Varga (ELTE)	6	14:08:40
10	Michał Maras, Dominik Kowalczyk, Michał Kępa (U. v Wrocławu)	6	14:22:59
11	Josef Minařík, Eldar Urmanov, Adam Rajský (Karlova u.)	6	15:08:05
12	Máté Busa, Bence Deák, Áron Noszály (ELTE)	6	16:10:55
13	Jan Wańkowicz, Łukasz Pluta, Hubert Obrzut (U. v Wrocławu)	6	16:29:08
14	Katzper Michno, Tomasz Mazur, Hubert Zięba (Jag. u.)	6	18:15:13
15	Josip Klepec, Marin Kišić, Jurica Horvat (U. v Zagrebu)	6	19:50:54
16	Marcel Szelwiga, Mateusz Orda, Oskar Fiuk (U. v Wrocławu)	5	8:48:12
17	Bojan Štetić, Leon Jurić, Dominik Fistrić (U. v Zagrebu)	5	10:53:17
18	Mateusz Opala, Aleksander Pogoda, Krzysztof Łukasiewicz (U. v Wrocławu)	5	10:58:33
19	Marko Khasin, Andrii Kovryhin, Maksym Tur (Jag. u.)	5	11:13:44
20	Vilim Lendvaj, Martin Josip Kocijan, Leonard Inkret (U. v Zagrebu)	5	12:10:54
21	Bartol Markovinović, Pavel Kliska, Gabrijel Jambrošić (U. v Zagrebu)	5	12:14:04
22	András Csertán, Péter Szente, Bálint Horcsin (ELTE)	5	13:09:41
23	Piotr Kępczyński, Andrzej Radzimiński, Antoni Wiśniewski (U. v Varšavi)	5	14:00:44
24	Gregor Kikelj, Job Petrovčič, Domen Hočevar (U. v Ljubljani)	5	14:37:06
25	Jiří Kalvoda, Václav Janáček, Magdaléna Mišínová (Karlova u.)	5	15:14:12
26	Marko Hostnik, Urban Duh, Žiga Željko (U. v Ljubljani)	5	15:46:36
27	Aleksejs Naumovs, Roberts Leonārs Svarinskis, Ingus Jānis Pretkalniņš (Latvijska u.)	4	8:44:38
28	Jan Kwiatkowski, Juliusz Korab-Karpowicz, Kamil Zwierzchowski (U. v Varšavi)	4	9:04:53
29	Sandra Siliņa, Aleksejs Jeļisejevs, Aleksandrs Zajakins (Latvijska u.)	4	11:17:24
30	Adam Pawłowski, Kamil Piechowiak, Paweł Woźniak (PUT)	4	11:48:49
31	Paweł Sankin, Andrei Mishchanka, Nazarii Denha (Jag. u.)	3	2:41:52
32	Mitko Nikov, Mitja Žalik, Vid Keršič (U. v Mariboru)	3	5:42:15
33	Daniel Sami Blažič, Jakob Schrader, Patrik Žnidaršič (U. v Ljubljani)	3	6:28:50
34	Dominik Farhan, Ondřej Sladký, Jonáš Havelka (Karlova u.)	3	6:48:29
35	Jakub Podolak, Paweł Putra, Dawid Sula (U. v Varšavi)	3	6:53:13
36	Kacper Karoń, Jakub Szczugieł, Arkadiusz Kraus (AGH)	3	7:07:24
37	Marcin Mordecki, Jakub Zarzycki, Kacper Harasimowicz (U. v Varšavi)	3	7:09:38

(*nadaljevanje na naslednji strani*)

REZULTATI CERC 2021 (*nadaljevanje*)

	Ekipa	Št. rešenih nalog	Čas
38	Lucija Žužić, Marina Banov, Jakov Tomasić (U. na Reki)	3	8:00:17
39	Krzysztof Jaworski, Rafał Szubert, Marcin Zatorski (PUT)	3	9:26:45
40	Nguyen Xuan Thang, Martin Prokopič, Jan Pokorný (CTU)	3	11:11:13
41	Illés Iles, István Megyeri (U. v Szegedu)	2	1:35:25
42	Luka Tomić, Luka Jovanović, Mislav Brnetić (U. v Zagrebu)	2	1:57:28
43	Karlo Iletić, Filip Kadak, Tomislav Prusina (U. v Osijeku)	2	2:03:00
44	Mateusz Kamiński, Patryk Kisielewski, Marcin Wojdat (U. N. Kopernika)	2	2:45:04
45	Bor Grošelj Simić, Matevž Miščič, Jon Mikoš (U. v Ljubljani)	2	3:03:50
46	Timo Zikeli, Otto Winter, Sebastian Schulze (TUW)	2	3:12:57
47	Jakub Kamiński, Artur Krzyżyński, Adam Ciężkowski (U. v Wrocławu)	2	3:14:20
48	Paweł Aniszewski, Paweł Czarkowski, Łukasz Skabowski (U. N. Kopernika)	2	4:49:12
49	Piotr Aksamit, Jan Izydorczyk, Jan Chyczynski (AGH)	2	4:57:49
50	Klaus Kraßnitzer, Christian Stippel, Sebastian Steiner (TUW)	2	7:03:49
51	Daniel Koren, Adam Barla, Jozef Koleda (CTU)	1	0:33:26
52	Jan Lukáš, Linda Šindelářová, Vít Břichňáč (CTU)	1	0:35:31
53	Kateřina Kubecová, Anna Ibatullina, Leonid Golovyrin (CTU)	1	0:39:22
54	Petr Šťastný, Marie Kalousková, Filip Danielsson (CTU)	1	0:43:08
55	Martin Domajnko, Jakob Kordež (U. v Mariboru)	1	0:51:09
56	Daniel Král, Martin Koutenský, Adam Procházka (CTU)	1	1:00:37
57	Józsué Majthényi-Wass, Lorenzo Fiorini, Miklós Bartos-Elekes (BUTE)	1	1:32:43
58	Alex Krizsan, Daniel Kaposvari, Andras Balogh (U. v Szegedu)	1	2:17:42
59	Barna Kirchhof, Ádám Horváth, Abdelrahman Desoki (BUTE)	1	2:30:07
60	Ajla Šehović, Jelena Glišić, Ina Bašić (U. na Primorskem)	1	3:49:31
61	Wojciech Holf, Tomasz Rahn, Dawid Stopczynski (U. N. Kopernika)	1	4:01:53

Sodelovale so ekipe z naslednjih univerz:

Češka tehnična univerza (CTU) (Praga, Češka)
Jagielonska univerza (Krakow, Poljska)
Karlova univerza (Praga, Češka)
Latvijska univerza (Riga, Latvija)
Poznanjska tehnična univerza (PUT) (Poznanj, Poljska)
Tehnična univerza na Dunaju (TUW) (Avstrija)
Univerza Josipa Juraja Strossmayerja v Osijeku (Hrvaška)
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 za tehnologijo in ekonomiko (BUTE) (Budimpešta, Madžarska)
Znanstveno-tehnična univerza AGH (Krakow, Poljska)

OFF-LINE NALOGA — POKRAJINA IZ KOCK

Na računalniških tekmovanjih, kot je naše, je čas reševanja nalog precej omejen in tekmovalci imajo za eno nalogo v povprečju le slabo uro časa. To med drugim pomeni, da je marsikak zanimiv problem s področja računalništva težko zastaviti v obliki, ki bi bila primerna za nalogo na tekmovanju; pa tudi tekmovalec si ne more privoščiti, da bi se v nalogo poglobil tako temeljito, kot bi se mogoče lahko, saj mu za to preprosto zmanjka časa.

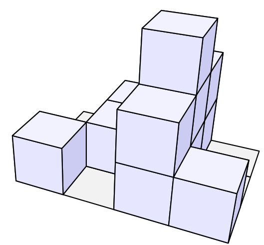
Off-line naloga je poskus, da se tovrstnim omejitvam malo izognemo: besedilo naloge in testni primeri zanjo so objavljeni več mesecev vnaprej, tekmovalci pa ne oddajajo programa, ki rešuje nalogo, pač pa oddajajo rešitve tistih vnaprej objavljenih testnih primerov. Pri tem imajo torej veliko časa in priložnosti, da dobro razmislijo o nalogi, preizkusijo več možnih pristopov k reševanju, počasi izboljšujejo svojo rešitev in podobno. Opis naloge in testne primere smo objavili decembra 2020, nekaj mesecev po razpisu za tekmovanje v znanju; tekmovalci so imeli čas do 26. marca 2021 (dan pred tekmovanjem), da pošljejo svoje rešitve.

Opis naloge

Iz kock, podobnih lego kockam, bi radi sestavili trodimenzionalni model pokrajine. Pokrajina je opisana z višinskim zemljevidom oz. tlorisom; območje, ki nas zanima, ima v tlorisu obliko pravokotnika, ki ga v mislih razdelimo na karirasto mrežo enotskih kvadratov. Ta mreža ima w stolpcev in h vrstic. Za vsako celico mreže je podana višina pokrajine v tej celici, recimo $v[x, y]$ za celico na preseku stolpca x in vrstice y. Vse te višine so cela števila, večja ali enaka 0.

Primer višinskega zemljevida ($w = 4$, $h = 3$) in pripadajoče pokrajine:

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

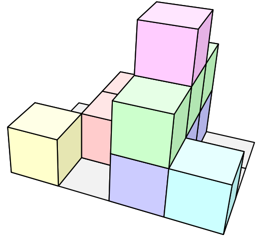
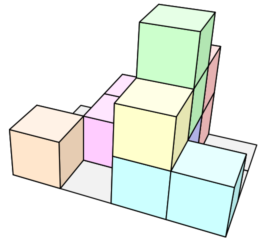


„Kocke“, iz katerih sestavljamo naš model pokrajine, so v resnici kvadri različnih velikosti. Možne velikosti kvadrov so podane, za vsako velikost pa imamo na voljo neomejeno število kvadrov tiste velikosti. Kvadre moramo zlagati tako, da se med seboj ne prekrivajo, da ne štrlijo ven iz pokrajine in da pokrijejo celotno pokrajino. Kvadre smemo vrteti v korakih po 90° okrog navpične osi (torej osi z), tako da lahko na primer kvader velikosti $a_x \times a_y \times a_z$ uporabimo tudi kot kvader velikosti $a_y \times a_x \times a_z$.

Tvoja naloga je sestaviti tak model pokrajine, ki ustreza tem omejitvam in pri tem porabi čim manj kvadrov.

Primer: recimo, da imamo višinski zemljevid z zgornje slike in da so na voljo kvadri naslednjih velikosti: $1 \times 1 \times 1$, $2 \times 1 \times 1$, $3 \times 1 \times 1$, $1 \times 1 \times 2$.

Naslednji dve sliki kažeta dva izmed možnih načinov, kako lahko iz kvadrov takih velikosti sestavimo pokrajino s tega višinskega zemljevida (kvadre na slikah smo pobarvali z različnimi barvami, vendar le zato, da se jih laže razloči, drugače pa barve ne pomenijo ničesar posebnega).

Rešitev na levi sliki sestavlja 6 kvadrov, rešitev na desni sliki pa 7 kvadrov, zato je leva rešitev boljša od desne.

Rezultati

Sistem točkovanja je bil tak kot pri off-line nalogah v prejšnjih letih. Pripravili smo 26 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 26 testnih primerih.

Pokrajine v testnih primerih so bile različnih oblik, nekatere v resnici bolj eno- ali dvo- kot trodimenzionalne, njihova prostornina pa je segala od nekaj sto do približno milijona enotskih kockic. Kvadri so bili pri večini testnih primerov majhni; njihove stranice niso presegale dolžine 5 enot.

Letos so svoje rešitve pri off-line nalogi poslali štirje tekmovalci, od tega dva srednješolca in en študent. Končna razvrstitev je naslednja:

Mesto	Ime	Letnik	Šola	Točke
1	Samo Kralj			258
2	Domen Hočevar	4	Gimn. Novo mesto	213
3	Jošt Smrtnik	3	Gimnazija Vič	182
4	Gregor Kikelj	3	FMF	170

Rešitev

Prva dva testna primera sta bila v resnici enodimenzionalna (pri enem je bilo $w = h = 1$, torej je bil tloris en sam enotski kvadrat; pri enem pa je bil $h = 1$ in višina pokrajine je bila povsod ≤ 1); takrat je torej pri kvadrih, s katerimi imamo opraviti, pomembna ena sama dimenzija — recimo, da je to dolžina, torej da so oblike $d\times1\times1$. Pokrajina je sestavljena iz enega ali več takih kvadrov (ločenih med sabo, tako da lahko rešujemo problem za vsakega posebej), pa tudi naši kvadri, iz katerih skušamo pokrajino sestaviti, so take oblike. Množico dolžin naših kvadrov označimo z A; potem smo torej pred vprašanjem, kako s čim manj kvadri iz A sestaviti kvader neke

želene dolžine d (ki predstavlja enega od kosov ciljne pokrajine). To lahko rešujemo z dinamičnim programiranjem: naj bo $f(d)$ najmanjše število kvadrov, potrebnih za kos dolžine d; potem je $f(0) = 0$ in $f(d) = 1 + \min\{f(d - a) : a \in A,\ a \leq d\}$. To lahko rešujemo po naraščajočih d in tako v $O(d)$ časa pridemo do optimalne rešitve. Spotoma si pri vsakem d še zapomnimo, pri katerem a je bil dosežen minimum, da bomo lahko na koncu rekonstruirali tudi konkreten nabor a-jev, ki se seštejejo v d.

Naslednjih šest testnih primerov je bilo v resnici dvodimenzionalnih: bodisi je imela pokrajina tloris oblike $w \times b$ in višino največ 1 bodisi je imela tloris oblike $w \times 1$ in višino največ b; in pri tem je bila dolžina w sicer velika, druga dimenzija b pa majhna. Poleg tega so bili majhni tudi kvadri (stranice so imeli dolge največ a). V tem primeru lahko nalogo še vedno rešujemo z dinamičnim programiranjem. Recimo, da postavljamo kvadre v pokrajino od leve proti desni, po naraščajočih x. To pomeni, da preden pokrijemo kakšno kockico pri $x = d$, mora biti v pokrajini vse na $x \leq d - a$ že popolnoma pokrito, saj so kvadri v vsako smer dolgi največ a enot, tako da noben kvader, ki je prisoten v stolpcu $x = d$, ne more hkrati segati tudi v $x = d - a$, kaj šele na manjše x. Naj bo torej $f(d, A)$ najmanjše število kvadrov, s katerimi lahko pokrijemo prvih d stolpcev naše pokrajine (torej $x \leq d$), pri čemer naj bodo v prvih $d-a$ stolpcih pokrite vse kockice naše pokrajine, v zadnjih a stolpcih pa le tiste iz množice A. Te vrednosti lahko računamo po naraščajočih d in pri vsakem d naraščajoče po $|A|$: če v A ni nobene kockice z $x = d$, je $f(d, A) = f(d - 1, A')$, pri čemer A' dobimo tako, da v A dodamo vse kockice, ki pripadajo naši pokrajini pri $x = d - a$. Sicer pa je $f(d, A) = 1 + \min_K f(d, A - K)$, pri čemer možne K dobimo tako, da na vse možne načine postavimo en kvader v mrežo, pri čemer mora biti prisoten v stolpcu $x = d$, ne pa tudi desno od njega in pokrivati mora le take kockice, ki so prisotne v A. Ker pokriva A le a stolpcev in ker je druga dimenzija naše mreže enaka b, imamo 2^{ab} možnih vrednosti A-ja, torej moramo rešiti vsega skupaj $O(w \cdot 2^{ab})$ podproblemov. Pri naših dvodimenzionalnih testnih primerih je bilo $a \cdot b \leq 20$, dolžina w pa do 10^5.

Za primere, ki so res trodimenzionalni, lahko našo nalogo zapišemo kot optimizacijski problem. Pokrajino si lahko predstavljamo kot množico enotskih kockic; recimo ji P. Vsak kvader, ki ga postavimo v prostor, pokrije neko podmnožico teh kockic, recimo K. Pri tem se omejimo seveda le na kvadre takih oblik, ki so nam pri našem testnem primeru na voljo, in le na take položaje teh kvadrov, pri katerih noben del kvadra ne štrli ven iz pokrajine P. Tako dobimo nek nabor možnih K-jev, ki ga označimo s $\mathcal{K}$.

Vpeljimo zdaj za vsak $K \in \mathcal{K}$ po eno neznanko u_K, ki nam pove, ali smo tak kvader na takem položaju res uporabili ($u_K = 1$) ali ne ($u_K = 0$); imamo torej omejitev $0 \leq u_K \leq 1$.

Za vsako kockico $p \in P$ naše pokrajine pa vpeljimo pogoj

$$\sum_{K \in \mathcal{K}\,:\,p \in K} u_K = 1;$$

z drugimi besedami, vsako kockico mora pokriti natanko eden izmed uporabljenih kvadrov. S tem poskrbimo, da se kvadri ne smejo prekrivati in da ne sme noben del pokrajine ostati nepokrit.

V okviru teh omejitev moramo zdaj najti tak nabor vrednosti u_K (za vse $K \in \mathcal{K}$),

pri katerem je vsota $\sum_{K \in \mathcal{K}} u_K$ čim manjša — to je namreč število uporabljenih kvadrov in naloga zahteva, da jih uporabimo čim manj. Optimizacijskim problemom te oblike (neznanke so celoštevilske, omejitve so linearne (ne)enačbe, pa tudi kriterijska funkcija, ki jo minimiziramo, je linearna funkcija; in koeficienti v omejitvah in kriterijski funkciji so cela števila) pravimo *celoštevilsko linearno programiranje.* To je sicer NP-težak problem, vendar obstajajo zanj razne knjižnice in programi, ki pri marsikaterem manjšem primeru tega problema vendarle najdejo optimalno rešitev v obvladljivo majhnem času. Tako lahko rešimo srednje velike primere naše naloge (recimo naslednjih deset primerov; pokrajina ni v nobeno dimenzijo merila več kot 50 enot, posamezni kvadri pa so imeli stranice do 4 enote in prostornino največ 16 kockic).

Zadnjih osem primerov je prevelikih, da bi jih reševali optimalno v enem zamahu (velikosti kvadrov so bile kot pri srednje velikih primerih, vendar so bile dimenzije pokrajine do 300 enot), lahko pa na primer razdelimo tloris pokrajine na manjša pravokotna območja (npr. do 15×15) in zlagamo kvadre nad vsakim od teh območij posebej. S tem seveda rešitev na koncu ni več nujno optimalna, ker izgubimo priložnost, da bi kakšen od kvadrov segal prek meje med dvema takima območjema. Nazadnje lahko poskusimo rešitev izboljšati še z neke vrste lokalno optimizacijo: izberimo si neko kocko v prostoru (npr. do velikosti $15 \times 15 \times 15$), pobrišimo v mislih tiste kvadre, ki v celoti ležijo znotraj nje, in poskusimo optimalno pokriti tako izpraznjene dele pokrajine; če se število uporabljenih kvadrov s tem zmanjša, rešitev obdržimo. To ponavljamo, dokler se rešitev še kaj izboljšuje.

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 bi bilo v normalnih razmerah verjetno novembra 2021, vendar je bilo zaradi epidemije in zamika *prejšnjega* CERCa preloženo na 23.–24. april 2022, potekalo pa je prek interneta), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bi moralo v normalnih razmerah potekati spomladi 2022, vendar je bilo zaradi zakasnitev, povezanih z epidemijo v prejšnjih letih, preloženo in bo predvidoma izvedeno novembra 2023).

Na letošnjem UPM je sodelovalo 41 ekip s skupno 118 tekmovalci, ki so prišli s treh slovenskih univerz, nekaj pa je bilo celo srednješolcev. Tabela na naslednjih dveh straneh prikazuje vse ekipe, ki so se pojavile na vsaj enem krogu tekmovanja.

	Ekipa	Št. rešenih nalog*	Čas
1	Benjamin Bajd (Gim. Kranj), Domen Hočevar (Gim. N. mesto), Job Petrovčič (FMF)	26	22:51:40
2	Žiga Željko, Marko Hostnik (FRI + FMF), Urban Duh (FMF)	23	31:24:21
3	Jakob Schrader, Patrik Žnidaršič, Daniel Sami Blažič (Gim. Vič)	20	31:40:07
4	Mitja Žalik, Vid Keršič, Matic Rašl (FERI)	19	34:24:56
5	M. Besher Massri, Nemanja Torbica, Mirza Redžić (FAMNIT)	18	29:52:44
6	Matevž Miščič, Jakob Zmrzlikar, Jon Mikoš (FMF)	18	30:49:04
7	Gregor Kikelj (FMF)	17	20:54:56
8	Vid Drobnič, Matej Marinko (FRI + FMF), Žiga Patačko Koderman (FRI)	17	22:41:46
9	Lan Sevčnikar, Matija Likar, Luka Lonec (II. gimn. Maribor)	16	20:47:47
10	Ilija Tavchioski, Josif Tepegjozov (FRI), Boshko Koloski (MPŠJS)	16	25:47:22
11	Jakob Kordež, Martin Domajnko, Tilen Koren (FERI)	15	29:17:59
12	Filip Štamcar, Jakob Kralj, Jošt Smrtnik (Gim. Vič)	14	18:39:51
13	Tadej Tomažič (Vegova Lj.), Tomaž Tomažič, Blaž Blokar	11	20:10:11
14	Urban Cör, Mark Žakelj (FRI + FMF) Domen Grzin (FRI)	11	25:34:51
15	Matija Kocbek, Luka Horjak (FMF), Lovro Drofenik (I. gim. v Celju)	10	7:57:50
16	Jaka Vrhovec, Adrijan Rogan (FRI + FMF)	9	11:26:27
17	Bor Grošelj Simić (FMF), Ella Potisek (Gim. Vič)	9	11:49:42
18	Mitko Nikov, Kristijan Mitrov, Stefan Srnjakov (FERI)	9	12:11:56
19	Anna Sidorova, Aljaž Žel, Alen Granda (FERI)	6	14:49:00
20	Tjaž Silovšek (FMF), Tim Vučina, Marcel Tori (FRI)	5	7:16:33
21	Sara Veber, Tim Poštuvan (FRI + FMF), Tina Poštuvan (FRI)	5	10:30:17
22	Tijan Veingerl, Gregor Šraj (FRI + FMF), Vid Rebol (FRI)	5	14:24:57
23	Nejc Zajc, Tadej Petrič, Žan Bajuk (FMF)	4	3:10:56
24	Ina Bašić, Jelena Glišić, Ajla Šehović (FAMNIT)	4	13:49:44
25	Nik Pangeršič, Gašper Pišek, Aleksandar Georgiev (FRI)	3	1:41:17
26	Miha Rajter (FRI + FMF), Domen Vreš, Timen Stepišnik Perdih (FRI)	3	2:43:48
27	Matic Šutar, Enei Sluga, Ana Strmičnik (FRI)	3	2:49:53
28	Miha Bastl, Urban Kocmut (FRI), Jan Geršak (FRI + FMF)	3	4:22:06
29	Bor Brecelj (FRI + FMF), Zala Erič, Miha Benčina (FRI)	3	5:22:26
30	Vili Perše (FAMNIT), Jani Bangiev, Aleš Špeh (FRI)	3	9:24:11

* Opomba: naloge z najslabšega od prvih treh krogov se ne štejejo, naloge z zadnjega kroga pa se štejejo dvojno. Enako je tudi pri času, le da se čas zadnjega kroga ne šteje dvojno.

(*nadaljevanje na naslednji strani*)

	Ekipa	Št. rešenih nalog⋆	Čas
31	Davor Ornik, Urban Knupleš, Janko Gruden (FERI)	2	2:08:18
32	Andraž Pauko, Primož Jozič (FMF), Jaša Dimič (FNM)	2	8:58:07
33	Žiga Kovačič, Erik Červek Roškarič, Adam Janko Koležnik (II. gimn. Maribor)	1	0:39:12
34	Žiga Ivanšek, Anja Krleža, Brin Pšunder (FERI)	1	0:43:41
35	David Šeruga, Anže Kocjančič, Jakob Dorn (FRI)	1	0:59:41
36	Milan Milivojčević, Jana Ristovska, Dušan Bjelica (FAMNIT)	1	1:09:58
37	Mateja Žvegler, Gašper Funda Povše, Marko Šimunović (FERI)	1	1:28:30
38	Marko Kričej, Andraž Valentinčič, Kristjan Šuligoj (FERI)	1	3:53:05
39	Andrej Perković, Amir Hadžipašić (FAMNIT)	0	0:00:00
	Ema Leila Grošelj, Nina Sangawa Hmeljak, Iztok Bajcar (FRI)	0	0:00:00
	Inga Raič, Nedeljko Bošković, Marko Janković (FAMNIT)	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 2021 so (z nekaterimi spremembami v sestavi) nastopile ekipe 1, 2, 3 in 6 kot predstavnice Univerze v Ljubljani, ekipi 4 in 11 kot predstavnici Univerze v Mariboru in ekipa 24 kot predstavnica Univerze na Primorskem. V konkurenci 61 ekip z 19 univerz iz 7 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
24	Gregor Kikelj, Job Petrovčič, Domen Hočevar	5	14:37:06
26	Marko Hostnik, Urban Duh, Žiga Željko	5	15:46:36
32	Mitko Nikov, Mitja Žalik, Vid Keršič	3	5:42:15
33	Daniel Sami Blažič, Jakob Schrader, Patrik Žnidaršič	3	6:28:50
45	Bor Grošelj Simić, Matevž Miščič, Jon Mikoš	2	3:03:50
55	Martin Domajnko, Jakob Kordež	1	0:51:09
60	Ajla Šehović, Jelena Glišić, Ina Bašić	1	3:49:31

Na srednjeevropskem tekmovanju je bilo 12 nalog, od tega so jih najboljše ekipe rešile po osem.

ANKETA

Ker je letošnje tekmovanje v celoti potekalo prek interneta, smo na ta način izvedli tudi anketo (prek spletne strani `1ka.si`). Vprašanja na anketi so prikazana spodaj in so bila približno enaka kot prejšnja leta, ko je bila anketa na papirju. Rezultati ankete so predstavljeni na str. 235–242.

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? ____________________

Katere od naslednjih jezikovnih konstruktov in programerskih prijemov znaš uporabljati?	ne poznam	da, slabo	da, dobro
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz standardnega vhoda ali pa ju zapisati na standardni izhod?	☐	☐	☐
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz datoteke ali pa ju zapisati v datoteko?	☐	☐	☐
Tabele (**array**):			
— enodimenzionalne	☐	☐	☐
— dvodimenzionalne	☐	☐	☐
— večdimenzionalne	☐	☐	☐
Znaš napisati svoj podprogram (**procedure**, **function**)	☐	☐	☐
Poznaš rekurzijo	☐	☐	☐
Kazalce, dinamično alokacijo pomnilnika (New/Dispose, GetMem/FreeMem, malloc/free, **new**/**delete**, ...)	☐	☐	☐
Zanka **for**	☐	☐	☐
Zanka **while**	☐	☐	☐

Gnezdenje zank (ena zanka znotraj druge) ☐ ☐ ☐
Naštevni tipi (*enumerated types* — **type** ImeTipa = (Ena, Dve, Tri) v pascalu, **typedef enum** v C/C++) ☐ ☐ ☐
Strukture (**record** v pascalu, **struct**/**class** v C/C++) ☐ ☐ ☐
and, **or**, **xor**, **not** kot aritmetični operatorji (nad biti celoštevilskih operandov namesto nad logičnimi vrednostmi tipa boolean) (v C/C++/C#/javi: &, |, ^, ~) ☐ ☐ ☐
Operatorja **shl** in **shr** (v C/C++/C#/javi: <<, >>) ☐ ☐ ☐
Znaš uporabiti kakšnega od naslednjih razredov iz standardnih knjižnic:
— 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 68 tekmovalcev prve skupine, 25 tekmovalcev druge skupine in 16 tekmovalcev tretje skupine.

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. 236. 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.

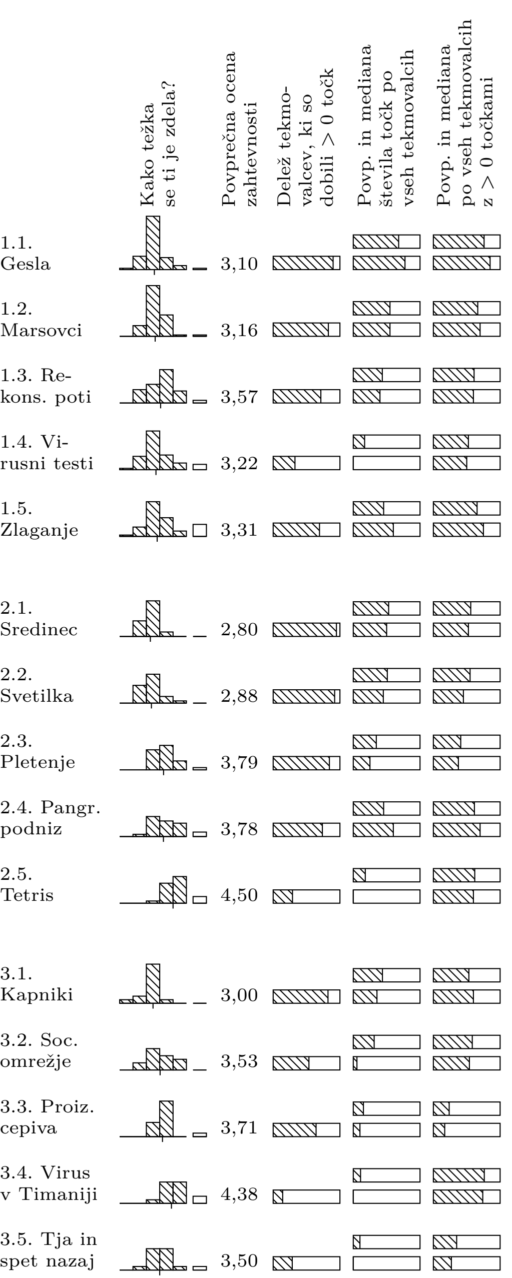
V povprečju so se zdele tekmovalcem v vseh skupinah naloge približno tako težke kot ponavadi, v drugi skupini mogoče malce težje. Zanimivo je, da čeprav smo dali letos v tretji skupini lažje naloge kot lani, v anketah tekmovalci niso dajali vtisa, da bi se jim zdele naloge kaj dosti lažje kot v lanski anketi. 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: 3,27 v prvi skupini (v prejšnjih letih 2,97, 3,47, 3,32, 3,11, 3,31), 3,55 v drugi skupini (prejšnja leta 3,38, 3,17, 3,19, 3,51, 3,65) in 3,63 v tretji skupini (prejšnja leta 3,67, 3,52, 3,59, 3,73, 3,43).

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 ($R^2 =$ 0,42; v prejšnjih letih 0,68, 0,71, 0,67, 0,70, 0,39).

V prvi skupini so tekmovalci kot težjo ocenili predvsem nalogo 1.3 (rekonstrukcija poti), ki sicer ni posebej težka, je pa malo neobičajna in zahteva nekaj razmisleka. V drugi skupini je izstopala naloga 2.5 (tetris), ki se je večini ljudi zdela pretežka; rešitev te naloge je sicer zelo šolski primer rekurzije, vendar le-ta tekmovalcem v drugi skupini mogoče še ni tako domača. V tretji skupini se jim je zdela najtežja naloga 3.4 (virus v Timaniji), ki se je večini ljudi zdela težka ali pretežka. Konceptualno sicer ni tako težka, je pa z njo precej dela, ker moramo dve vrsti testnih primerov reševati na dva precej različna načina (pri tej nalogi je veliko tekmovalcev tudi reklo, da bi vzela preveč časa).

Kot najlažjo so tekmovalci v prvi skupini ocenili nalogo 1.1 (gesla), v drugi skupini 2.1 (sredinec) in v tretji 3.1 (kapniki). To se ujema z našimi pričakovanji, saj se načeloma trudimo naloge v posamezni skupini oštevilčiti približno od lažjih proti težjim.

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 237. Nad razumljivostjo besedil ni veliko pripomb, podobno kot prejšnja leta, v drugi skupini še malo manj. Kot težje razumljive so ocenili predvsem naloge 1.2 (marsovci), 1.4 (kako dobri so virusni testi?), 2.5 (tetris) in 3.4 (virus v Timaniji). Pri tem še zlasti izstopa 1.4, ki je redek primer naloge, pri kateri je več ljudi reklo, da je težko razumljiva ali celo nerazumljiva, kot pa, da je razumljiva. To se je pokazalo že med tekmovanjem, ko je bilo ravno o tej nalogi največ vprašanj (predvsem o pomenu



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.

Naloga je (ali bi) vzela preveč časa
Dolžina besedila
Razumljivost besedila
Je bila naloga zanimiva?
Si jo rešil?
Ti je bila najmanj/najbolj všeč?

1.1. Gesla
1.2. Marsovci
1.3. Rekons. poti
1.4. Virusni testi
1.5. Zlaganje

2.1. Sredinec
2.2. Svetilka
2.3. Pletenje
2.4. Pangr. podniz
2.5. Tetris

3.1. Kapniki
3.2. Soc. omrežje
3.3. Proiz. cepiva
3.4. Virus v Timaniji
3.5. Tja in spet nazaj

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?

	Prva skupina	Druga skupina	Tretja skupina
priority_queue v C++ ipd.	11 %	20 %	56 %
map v C++ ipd.	9 %	12 %	69 %
unordered_map v C++ ipd.	29 %	32 %	69 %
zamikanje s **shl**, **shr**	23 %	36 %	75 %
operatorji na bitih	68 %	76 %	81 %
strukture	36 %	64 %	81 %
naštevni tipi	21 %	28 %	56 %
gnezdenje zank	89 %	96 %	100 %
zanka **while**	95 %	92 %	100 %
zanka **for**	95 %	96 %	100 %
kazalci	15 %	28 %	44 %
rekurzija	43 %	68 %	94 %
podprogrami	95 %	92 %	100 %
več-d tabele (**array**)	48 %	61 %	100 %
2-d tabele (**array**)	69 %	84 %	100 %
1-d tabele (**array**)	80 %	92 %	100 %
delo z datotekami	62 %	72 %	94 %
std. vhod/izhod	88 %	96 %	100 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

---

parametra n; precej tekmovalcev ga je čisto ignoriralo in iskalo najdaljšo strnjeno skupino ujemanj).

Tudi z dolžino besedil so tekmovalci večinoma zadovoljni; ocene so podobne kot prejšnja leta. Po komentarjih, da je naloga predolga, še najbolj izstopajo naloge 1.5 (zlaganje loncev), 2.5 (tetris), 3.2 (socialno omrežje) in 3.4 (virus v Timaniji). Mnenj, da je besedilo prekratko, je bilo malo, še največ pri nalogi 1.4 (kako dolgi so virusni testi?).

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta. Kot bolj zanimive izstopajo 1.3 (rekonstrukcija poti), 2.5 (tetris) in 3.3 (proizvodnja cepiva), kot manj zanimivi pa 1.4 (kako dobri so virusni testi?) in 2.3 (pletenje puloverja). Pripomb, da jim je neka naloga že znana, je bilo letos skoraj pol manj kot lani; največ jih je bilo pri nalogi 1.5 (zlaganje loncev).

Pripomb, da bi naloga vzela preveč časa, je bilo malo, vendar več kot ponavadi. Največ takih pripomb je bilo pri nalogah 1.3 (rekonstrukcija poti), 2.5 (tetris) in 3.4 (virus v Timaniji). Še posebej izstopa slednja, kjer je rešitev res nekoliko daljša kot običajno. Verjetno ni naključje, da so se jim zdele večinoma iste naloge tudi najzahtevnejše.

Pri vprašanjih, katera naloga je tekmovalcu najbolj in katera najmanj všeč, so bili glasovi letos precej razpršeni med naloge, večkrat pa se je tudi zgodilo, da je ista naloga dobila veliko glasov pri obeh vprašanjih (npr. nekaj nalog v prvi skupini). Kot nepriljubljene izstopajo naloge 1.4 (kako dobri so virusni testi?), 2.5 (Tetris) in 3.4 (virus v Timaniji), kar so večinoma tudi tiste naloge, ki so se jim zdele težke in težko razumljive. Kot bolj priljubljene pa izstopajo naloge 1.3 (rekonstrukcija poti), 2.1 (sredinec) in 3.1 (kapniki).

	Prva skupina	Druga skupina	Tretja skupina
drevo	38 %	52 %	88 %
hash tabela	31 %	44 %	81 %
seznam (linked list)	41 %	56 %	75 %
sklad (stack)	40 %	48 %	75 %
vrsta (queue)	43 %	52 %	94 %
Evklidov algoritem	75 %	83 %	100 %
Eratostenovo rešeto	66 %	68 %	88 %
vektorski produkt	28 %	52 %	63 %
rekurzija	46 %	36 %	81 %
dinamično prog.	29 %	28 %	81 %
iskanje v širino	20 %	36 %	69 %
O-zapis	45 %	67 %	88 %
urejanje	74 %	80 %	100 %
bubble sort	40 %	64 %	83 %
insertion sort	26 %	36 %	56 %
selection sort	25 %	20 %	50 %
quicksort	26 %	36 %	72 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

Programersko znanje, algoritmi in podatkovne strukture

Ko sestavljamo naloge, še posebej tiste za prvo skupino, nas pogosto skrbi, če tekmovalci poznajo ta ali oni jezikovni konstrukt, programerski prijem, algoritem ali podatkovno strukturo. Zato jih v anketah zadnjih nekaj let sprašujemo, če te reči poznajo in bi jih znali uporabiti v svojih programih.

Rezultati pri vprašanjih o programerskem znanju so podobni tistim iz prejšnjih let. V prvi skupini pravijo, da znajo malo manj 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. Kazalce pozna letos še manj ljudi kot lani (kar sicer najbrž ni čudno, saj jih veliko dela v jezikih, kjer s kazalci nimajo veliko opravka).

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 tokrat java; sledita jima C++ in C (tudi med uporabniki C++ je nekaj takih, katerih C++ je skoraj C), C# pa je letos redkejši. Tudi v drugi skupini je python najpogostejši, sledi pa mu C++, ki letos zaostaja manj kot lani; nekaj tekmovalcev je uporabljalo še C in C#, nihče pa letos ni v drugi skupini uporabljal jave. V tretji skupini je še vedno najpogostejši C++, več tekmovalcev kot običajno je uporabljalo javo, nekaj pa tudi python. Basica ni letos uporabljal nihče, pascal pa štirje v prvi skupini. Edini jezik, ki se je še pojavil poleg doslej omenjenih, je javascript, ki so ga letos uporabljali trije tekmovalci.

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

	Leto in skupina																	
	2021			2020			2019			2018			2017			2016		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal	4			2			2						4			$\frac{1}{3}$	3	
C	8	5	$\frac{1}{2}$	$3\frac{1}{2}$	3		10	4	$\frac{1}{2}$	5	4	$\frac{1}{2}$	4	3	$2\frac{1}{2}$	$4\frac{1}{3}$	1	2
C++	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	$15\frac{1}{2}$	28	8	9
java	17		5	15	4	3	15	5	1	$21\frac{1}{2}$	$8\frac{1}{2}$	4	28	3	2	24	6	5
PHP			–	1		–			–			–			–			–
C#	6	4		6	3		12	2		11	6		7	6		12	5	1
python	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	–	$29\frac{1}{3}$	12	–
javascript	1	2	–	2		–			–		$\frac{1}{2}$	–			–	1		–
julia			–			–			–			–	1		–			–
swift			–	1		–			–			–			–			–
psevdokoda	3		–	2	1	–	5	1	–	3	1	–	5		–	5		–
nič	3			2			3			2		1				2	3	

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 2021 je en tekmovalec uporabljal C in C++, eden 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“.

---

mogoče sklepal, da bi bilo dobro dati več nalog tipa „opiši postopek“ (namesto „napiši podprogram“), vendar se v praksi običajno izkaže, da so takšne naloge med tekmovalci precej manj priljubljene in da si večinoma ne predstavljajo preveč dobro, kako bi opisali postopek (pogosto v resnici oddajo dolgovezne opise izvorne kode v stilu „nato bi s stavkom **if** preveril, ali je spremenljivka x večja od spremenljivke y“). Podobno kot prejšnja leta smo tudi letos pri nalogah tipa „opiši postopek“ pripisali „ali napiši podprogram (kar ti je lažje)“ (kjer je bilo to primerno).

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, metoda emplace_back; pri enem tudi structured bindings iz C++17).

Pri pythonu zdaj praktično 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 lani (61/65 v prvi skupini in 25/25 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

javo ali C#; najpogostejši originalni predlog je bil javascript. 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 (50/63 v prvi skupini, 23/25 v drugi, 14/16 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) ter python in C# (predvsem v drugi skupini). 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 mlajši kot lani, v tretji pa malo starejši. Letos je nastopilo tudi nekaj osnovnošolcev, in sicer vsi v prvi skupini.

	Št. tekmovalcev po letnikih							
Skupina	8	9	1	2	3	4	5	Povprečni letnik
prva	2	4	18	18	29	26	1	2,6
druga			7	10	11	8	2	2,8
tretja			2	2	8	16		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 približno tri četrtine); sledijo tisti, ki so se tega naučili v šoli pri pouku (takih je slaba polovica). Približno dve petini tekmovalcev pa sta se naučili programirati (tudi) na krožkih in tečajih.

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni, vendar je njihov delež letos manjši kot prejšnja leta. Med ostalimi so mnenja precej razdeljena, najpogostejši kombinaciji pa sta „več časa, enako (ali manj) 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. Pri sistemu za oddajo odgovorov v prvi in drugi skupini je precej tekmovalcev želelo, da bi bilo okno za vnos besedila večje in da bi podpiralo avtomatsko zamikanje vrstic

	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	64	3	5	1	53	34	13	9	8	24	2	37	2	17	43
II	22	1	3	2	16	11	6	6	7	12	1	12	1	10	14
III	12	1	4	0	14	3	7	3	2	2	2	11	0	3	12

ter označevanje sintakse z barvami; toda tem težavam se je najlaže izogniti tako, da pišemo odgovore v svojem priljubljenem urejevalniku ali razvojnem okolju (to je bilo letos še toliko lažje, ker so tekmovalci reševali naloge doma), nato pa jih le skopiramo in prilepimo v spletni obrazec za oddajo odgovorov.

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 letošnje (in tudi lansko) tekmovanje, ko so vsi reševali naloge doma in torej dostop do prevajalnikov in razvojnih orodij imeli, 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 bomo predvidoma tudi v prihodnje, ko bo tekmovanje spet potekalo v živo namesto prek interneta, omogočili tekmovalcem prve in druge skupine tudi uporabo prevajalnikov.

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) Za ljubitelje nepotrebnih pretvorb: naslednja rešitev kar naprej pretvarja nize v tabele znakov, tudi za operacije, ki bi jih čisto lahko naredila kar na samem nizu.

```
foreach (string g in geslo) {
   for (int y = 0; y < g.ToCharArray().Count(); y++) {
      char c = g.ToCharArray()[y];
```

(1.1) Zanimivost: nekdo v C++ uporablja alternativno predstavitev operatorja &&, torej besedo **and**. Morda je na C++ preklopil iz pythona.

```
if (v[i] > 64 and v[i] < 123) {
   v[i] = v[i] − 32;
```

(1.1) Namen je bil dober, ampak zakaj vendar ni na levi strani preprosto s[i]?

```
(Chr(Ord(s[i]))) := Chr(Ord(s[i] − 32)); // 32 = Ord(a) − Ord(A), spremeni male črke
                                          // v velike črke
```

(1.1) Zakaj bi napisal pika = '.', če pa lahko pokažeš svoje poznavanje ASCIIja:

```
char pika = 46; // 46 je ASCII vrednost pike
```

(1.1) Boleč način za preverjanje, ali je trenutni znak gesla črka:

```
if (geslo1.charAt(i) != '.' && geslo1.charAt(i) != '0' && geslo1.charAt(i) != '1' &&
    geslo1.charAt(i) != '2' && geslo1.charAt(i) != '3' && geslo1.charAt(i) != '4' &&
    geslo1.charAt(i) != '5' && geslo1.charAt(i) != '6' && geslo1.charAt(i) != '7' &&
    geslo1.charAt(i) != '8' && geslo1.charAt(i) != '9')
```

Pohvalno pa je, da je bil dovolj zvit, da je preverjal teh 11 pogojev, kaj znak ne sme biti, namesto 26 pogojev, katera črka je lahko :)

(1.1) Še en neugoden način za preverjanje, ali je trenutni znak niza črka:

```
case izpis[n] of
   a, b, c, d, e, f, g, j, k, l, m, n, o, p, r, s, t, v, u, z, w, y, x: daNe := 1;
end;
```

Bolj kot pomanjkanje narekovajev nas lahko moti diskriminacija črk *h*, *i* in *q*. To je še toliko bolj obžalovanja vredno, ker je v pascalu za take reči na voljo zelo elegantna sintaksa oblike 'a'..'z'.

(1.1) Rešitev s pomisleki glede vračanja iz podprogramov:

```
return 1; /* vrnemo se iz podprograma, po želji zamenjaj z while(1); če se
             bojiš vračanja iz podprogramov */
```

(1.2) Zakaj bi uporabili običajno zanko **for**, če lahko uporabimo list comprehension in spotoma pridelamo še popolnoma nekoristen seznam samih vrednosti None:

```
enaplusenajekurapecena = [c.append(int(y)) for y in u]
```

(1.2) Prispevek na temo „zavajanje sovražnika“:

```
// Array 100 opravil, z 0 marsovci, ki jih opravljajo
// pomoje da je hitrejše kot s slovarjem (opravilo → število marsovcev)
HashMap<Integer, Integer> opravila = new HashMap<>();
```

Array bi bil najbrž res hitrejši; zakaj je potem vendarle uporabil slovar?

(1.2) Ne gre in ne gre:

```
datoteka = [] # ustavrimo listo
⋮
opravila = [] # ustravimo listo
```

(1.2) Presenetljivo: nekdo v pythonu ne ve za operator **in**, pač pa ve za metodo __contains__ (ali pa ve za oboje in vendarle raje uporablja slednjo?).

```
if not seznam.__contains__(opravilo):
```

(1.2) Ob deklaraciji bufferja, v katerega bo prebral število marsovcev:

```
char buf[100]; /* vhodni niz ne more biti večji od 100, razen če je marsovcev več
                  kot 1e100, v tem primeru imamo večje probleme */
```

(1.2) Za ljubitelje umazanih trikov:

```
with open("marsovci.txt") as datoteka: # odpremo datoteko, iz katere beremo
  next(datoteka) # grd trik v pythonu, ki nam omogoči, da spustimo 1. vrstico,
                 # ker je nepotrebna
```

Lepše bi se sicer enak učinek dalo doseči z „datoteka.readline()“.

(1.3) Tale direktorijem pravi „direktorji“:

```
if (st − prejsnji) > 1: # če je med zaporednima številoma razlika več kot 1, pomeni,
                        # da en direktor manjka in je to napaka, ker je prvi v seznamu
                        # dikretor, lahko prejsnji naprej pustimo na 0
```

Tudi naslednji tekmovalec očitno ni bil zadovoljen z zgodbico o direktorijih in namesto tega govori o zaposlenih in njihovih šefih:

```
# imena si shranimo, če rabimo kasneje dostopati
employee = boss + '/' + imena[i]
output.write(employee + '\n')
employees.append(employee)
```

(1.3) Zanimiva sintaktična inovacija: operator −= za brisanje stvari s konca niza.

```
boss −= '/'
boss −= imena[n]
```

(1.3) Odlična tipkarska napaka:

> Tukaj je tudi zanka za ugotavljanje narobe zapisane hirarhije.

Lahko si predstavljamo hirajočo hierarhijo :)

(1.3) Boleče neroden način računanja poti:

```
if (nivo[i] == 1) ime[1] = "/" + pot[i];
else if (nivo[i] == 2) ime[2] = ime[1] + "/" + pot[i];
else if (nivo[i] == 3) ime[3] = ime[2] + "/" + pot[i];
⋮
else if (nivo[i] == 9) ime[9] = ime[8] + "/" + pot[i];
else if (nivo[i] == 10) ime[10] = ime[9] + "/" + pot[i];
```

To je še toliko bolj presenetljivo, ker drugod v svoji rešitvi čisto lepo uporablja konstrukte oblike ime[nivo[i]].

(1.3) Neka rešitev porabi najprej skoraj celo stran kode za branje vhodnih podatkov, potem pa se zaključi takole... antiklimaktično:

```
ofstream write("izhod.txt");
for (int i = 0; i < stVrstic; i++) {
   for (int j=0; j < vhod[i].st; j++) {
      // tukaj bi dal kodo, ki vpisuje po pravem zaporedju + preverja
   }
   write << "\n";
}
write.close();
```

(1.4) Naslednja rešitev gre v zanki po znakih niza in jih skuša primerjati z istoležnimi znaki drugega niza. Ker pa ima v spremenljivki i trenutni znak, ne pa njegovega indeksa, poskuša indeks določiti tako, da trenutni znak išče v prvem nizu:

```
for i in s:
   if i != t[s.index(i)]:
```

To seveda odpove, če je v nizu s več enakih znakov, in ker imamo pri tej nalogi nize samih ničel in enic, se bo to zgodilo skoraj vedno.

(1.4) Rešitev z nepotrebno aritmetiko:

```
for (int i = 0; i < d; i++) r[i] = (s[i] − '0') − (t[i] − '0');
```

(1.4) Prijetno dekadenten način za primerjanje istoležnih znakov obeh vhodnih nizov: pretvoril ju je v celi števili, xoral in nato pretvoril nazaj v niz ničel in enic.

```
def Primerjava(s, t, n):
   # lista xoramo
   errors = bin(int(s, 2) ^ int(t, 2))[2:]
```

(1.4) Tole je napisano tako, kot da bi obstajala še kakšna tretja možnost poleg tega, da se znaka razlikujeta ali ujemata:

```
if (s.charAt(i) != t.charAt(i))
   tab[j]++;
else if (s.charAt(i) == t.charAt(i))
   j++;
```

(1.4) Impresivne komplikacije pri preverjanju, ali sta s[i] in t[i] različna:

```
if len(set([s[i], t[i]])) == 2:
```

(1.5) Eden od bolj nekoherentnih opisov postopka letos:

> Najprej bi iz tabele vzel ven tri največje vrednosti in jih dal v tri tabele. Nato bi vsaki vrednosti pripisal vse vrednosti iz glavne tabele, ki so ji najbližje. Ko najde vsaka tabela vse vrednosti, ki so večje od mediane (sredine) vseh vrednosti, bi vsaki tabeli pripisal še vrednosti, ki so manjše od mediane. Vse tri tabele bi seštel in ven dobil rezultat.

(1.5) Težava nalog z zgodbico (kot je na primer tale z lonci) je, da se jih nekateri ljudje lotevajo preveč zdravorazumsko:

> Program poskuša spraviti lonce v največ štiri različne kupe (odvisno od števila in višine loncev). Pri tem poskuša razdeliti lonce v enako visoke kupe in iz večjih loncev v manjše. Torej sproti računa, kateri so največji lonci, in iz teh izbere 4 največje itd. Če višina kupa ni velika (mogoče manj kot 30 cm višine), nato program poskuša spraviti lonce v 3 kupe. Če gre še manj, v 2 in nato v 1.

(1.5) Nekateri pa si navodilo „opiši postopek“ razlagajo tako, da opišejo, kako bi napisali program:

> Napisal bi program, ki bi najprej prebral vse vnose od uporabnika. Nato bi se lotil pisati program, ki bi preveril vse možnosti zlaganja posod (gnezdenje zank), nato bi napisal pogoj v zanke, ki bi izpisoval najboljše moznosti na ponujene velikosti posod.

(1.5) Kratko in jedrnato ... in popolnoma neuporabno:

Uredimo lonce po premerih.
Po velikosti padajoče premere seštevamo.
Vsoto primerjamo z najmanjšo možno vsoto.
Dobimo optimalne sklade.

(2.1) Hudo pesimističen pogled na hitrost izvajanja:

> Tak program bi pri 100 učencih potreboval nekaj sekund, da bi izpisal višine vseh učencev, ki so bili na sredini [...]

Opisal je rešitev, ki bi po prihodu n-tega učenca potrebovala $O(n)$ časa, da bi ugotovila, kdo je zdaj sredinec — neučinkovito, ampak tako zelo pa spet ne.

(2.1) Pri tej nalogi veliko rešitev vzdržuje urejen seznam višin vseh učencev. Mnogi tekmovalci so se pri dodajanju novega učenca domislili, da bi mesto, kamor ga je treba vriniti, poiskali z bisekcijo in tako prihranili čas. Cvetka je v tem, da ni skoraj nihče od njih pomislil, da potem še vedno porabimo $O(n)$ časa za premikanje ostalih učencev za eno mesto naprej po tabeli, torej z bisekcijo v resnici nismo ničesar bistvenega pridobili.

(2.1) Iz psevdokode, ki dodaja novega učenca v urejen seznam (spodnji vrstici se nanašata na primer, ko ga je treba dodati na konec):

učencu, ki smo ga zadnjega gledali, odvzamemo prestižen naslov
   zadnjega v seznamu;

(2.1) Prispevek na temo “technically correct, the best kind of correct”:

// *Rešitev je* $O(n^n)$

V resnici je bila $O(n^2)$, kar je seveda obenem res tudi $O(n^n)$.

(2.1) Presenetljivo veliko tekmovalcev se je pri računanju indeksa, na katerem se v urejenem zaporedju pojavlja srednji element, zateklo h kompliciranju z ne-celimi števili, na primer:

```
printf("%d\n", visina[(int) ceil((double) i / 2)]); // uporabim ceil za zaokroževanje navzgor
                                                     // ter (int) in (double) za pravilno delovanje
```

Enak učinek bi dosegli z „visina[(i + 1)/2]“, pri čemer smo izkoristili dejstvo, da operator / na celih številih zaokroža proti 0.

(2.1) Prispevek na temo „ima se, može se“. Dodajanje v urejen seznam je za šleve, pravi frajerji urejajo celo zaporedje vsakič znova:

> Na konec arraya dodajam višine, potem pa s pomočjo qsorta sortiram vsakič, ko se doda nova višina.

(2.1) Rešitev z velikimi pričakovanji do stavkov **if**:

> Program je sestavljen samo iz vgnezdnjenih **if** stavkov, kar samo po sebi ne bi smelo biti zahtevno, kakor če bi uporabljali zanke.

Primer enega od njegovih stavkov **if**:

> — če je novi dijak večji od sredinskega in hkrati večji od prvega večjega od sredinskega → takrat je dijak, ki je bil prvi po vrsti večji od sredinskega, novi sredinski

S tem, kako bo vzdrževal vrstni red dijakov po višinah in koliko časa mu bo to vzelo, pa se ni ukvarjal :)

(2.2) Zanimiv primer eksotične sintakse: **while** namesto **if**.

```
# ko je z enak 0, se „z“ zviša za 1 in vrne vrednost True
while (z == 0):
    z += 1
    return True
# drugače, vrne vrednost False
else:
    return False
```

To deluje pravilno — če/ko pogoj v **while** ni izpolnjen, se izvede blok **else**.

(2.2) Ena od rešitev je imela v funkciji Tipka neskončno zanko, težave s tem pa je preložila kar na neznanega programerja:

```
while (true) { // ponavljano utripanje luči, dokler ne pritisnemo ponovno na tipko,
    // predpostavimo, da je programer, ki bo uporabil to funkcijo, dovolj vešč, da
    // bo ob ponovnem klicu Tipka(pritisnjena) uredil, da se ta while preneha izvajati
```

(2.2) Posebno nagrado za vprašljive prispevke h kozmologiji dobi:

```
// timer bo deloval približno pol miljarde let, kar je več kot starost vesolja,
// dokler ne pride spet na 0 in bo sigurno dovolj
```

Tisti timer je bil sicer spremenljivka tipa **unsigned long long**, ki ga njegova rešitev v funkciji TikTak poveča za 1. Tako bo prišel na 0 po $2^{64}/10$ sekundah, kar je približno 58 milijard let.

(2.4) Tale podnizom pravi „permutacije“:

```
# dobimo vse permutacije
for i in range(len(s)):
  for j in range(i + 1, len(s) + 1):
    if j − i >= min_št_znakov:
      permutacije.append(s[i:j])
```

(2.4) Nekateri ne vedo dobro, koliko črk ima abeceda:

```
# Recimo, da imamo znake slovenske abecede shranjene v neki tabeli z dolžino 24 znakov
```

In nekdo drug:

```
const int da = 30; // velikost abecede
```

(2.4) Več tekmovalcev namesto „pangram“ piše „panagram“:

```
int panagram(char* s, int k)
```

Lahko si predstavljamo pangram, ki si je odprl slamnato podjetje v Panami, da se bo izmikal plačevanju davkov :)

(2.4) Zakaj bi odštevali `'a'` (torej 97), če lahko odštevamo `2 * '0'` (torej $2 \cdot 48 = 96$):

```
shrt = s[i] − 2 * '0'; // crke pretvorim v v vrednosti od 1 do 25
```

(2.4) Rešitev z udarjanjem po tipkovnici:

```
// malo sem udarjal po tipkovnici, zdi se mi, da razumete, kaj je to
let test = "dfahdlkfhadbfjajdsncladjtfgvjbgnaoimejtelepdanmnfoaisnvbfaoisnfbvi
                                   asnfboiajsnfbaijnfaojsnfvoiasnfbas";
```

Je pa impresivno, da je z udarjanjem po tipkovnici med drugim napisal „imejte lep dan“...

(2.4) Nekdo res ne mara besede „pregledovanje“:

> Program bi lahko še dalje optimizirali z omejevanjem pregledevanja zadnjih $26 \cdot k - 1$ znakov (angleška abeceda ima 26 znakov in vsak se mora pajaviti k-krat, kar pomeni da moramo pri $k = 2$ preveriti vsaj 52 znakov, zato je preglejevanje začeti na 51. indexu pred koncem pri $k = 2$ nepotrebno).

(2.5) Eden od tekmovalcev je najprej preučil vse oblike ploščkov (postavljal je v mrežo po en plošček in s funkcijo JePokrita gledal, katera polja so pokrita). Po tem zanimivem, vendar povsem nekoristnem delu (koristno bi bilo le, če bi hotel sam implementirati funkciji PreveriPloscek in PostaviPloscek) pa se program zaključi takole:

```
    // zdaj ko ima program vse možne informacije, začne z neznanim algoritmom
    // postavljati oblike
}
```

(3.1) Nagrado za prispevke k speleologiji dobi:

```
// M raste iz tal
// T razste iz zraka
```

Če rastejo kapniki iz zraka, zakaj jih moramo hoditi potem gledat v jame, namesto da bi jih gojili lepo nad zemljo...

(3.1) Prispevek na temo „zavajanje sovražnika“. Ko v rešitvi (napisani v C++) vidimo

```
sort(all(pairs));
reverse(all(pairs));
```

bomo najprej morda pomislili, da uporablja std::ranges::sort in std::views::all iz C++20, in bomo prijetno presenečeni, da tekmovalci to že poznajo in da prevajalnik to že podpira. Toda izkazalo se je, da je na začetku programa definiran makro:

```
#define all(n) n.begin(),n.end()
```

(3.1) Današnja mladina je res razvajena; naslednja rešitev pričakuje skoraj 7 eksabytov pomnilnika:

```
int t[1000000000000000002], m[1000000000000000002];
```

Zaradi te deklaracije se program sploh ne prevede, zato je nekaj minut kasneje prišla na ocenjevalni strežnik še popravljena verzija, ki zahteva le dobrih 700 terabytov:

```
int t[100000000000002], m[100000000000002];
```

Seveda se tudi ta ni prevedla. Bodisi uporabljajo nekateri doma zelo čudne prevajalnike ali pa oddajajo kodo, ki je niso niti poskušali prevesti...

(3.4) Pri tej nalogi smo dobili najdaljšo oddajo letos, dolgo kar 308 vrstic (8.9 KB); druga najdaljša je skoraj polovico krajša od nje.

(3.5) Naslednja rešitev poskuša do časovne omejitve generirati naključne razporede in na koncu vrniti najboljšega:

```
while time.time() − start < 1.9:
    random.shuffle(coords)
    sum = all(coords)
    if sum < best:
        best = sum
    else:
        pass
```

Žal je pri večini primerov časovno omejitev (2 sekundi) vseeno prekoračila; očitno so ostale stvari (predvsem branje vhodnih podatkov) trajale več kot 0,1 sekunde.

(3.5) Zanimiv primer povsem zgrešene rešitve. Ena od rešitev pri tej nalogi sestavi pot tako, da uredi točke po orientaciji glede na najbolj levo in jih obišče v tem

vrstnem redu. Toda v tem vrstnem redu si lahko med seboj sledijo točke na zelo različnih oddaljenostih od začetne, tako da lahko pot precej cikcaka, namesto da bi šla najprej le v desno in odtlej le v levo, kot zahteva naloga.

(3.5) Tale hobit pa je bil malo bolj temeljit: na poti tja je obiskal prav vse točke (po naraščajočem x), nato pa vse še enkrat na poti nazaj. Seveda so bili zato vsi odgovori napačni:

```
tocke = sorted(tocke, key = lambda x: x[0])
for i in range(n − 1):
    pot += math.sqrt((tocke[i + 1][0] − tocke[i][0]) ** 2 +
                     (tocke[i + 1][1] − tocke[i][0]) ** 2)
print(pot * 2)
```

Kmalu zatem je isti tekmovalec poslal popravljeno različico, pri kateri se hobit iz najbolj desne točke vrne naravnost v začetno (najbolj levo), kar je sicer še vedno narobe.

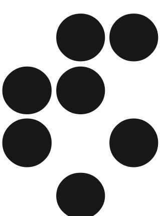
(3.5) Prispevek na temo „ziher je ziher“:

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

Rešitev je sicer kasneje uporabljala le stvari iz std, tako da je prvi **#include** odveč.

SODELUJOČE INŠTITUCIJE

Institut Jožef Stefan



Institut je največji javni raziskovalni zavod v Sloveniji s skoraj 800 zaposlenimi, od katerih ima približno polovica doktorat znanosti. Več kot 150 naših doktorjev je habilitiranih na slovenskih univerzah in sodeluje v visokošolskem izobraževalnem procesu. V zadnjih desetih letih je na Institutu opravilo svoja magistrska in doktorska dela več kot 550 raziskovalcev. Institut sodeluje tudi s srednjimi šolami, za katere organizira delovno prakso in jih vključuje v aktivno raziskovalno delo. Glavna raziskovalna področja Instituta so fizika, kemija, molekularna biologija in biotehnologija, informacijske tehnologije, reaktorstvo in energetika ter okolje.

Poslanstvo Instituta je v ustvarjanju, širjenju in prenosu znanja na področju naravoslovnih in tehniških znanosti za blagostanje slovenske družbe in človeštva nasploh. Institut zagotavlja vrhunsko izobrazbo kadrom ter raziskave in razvoj tehnologij na najvišji mednarodni ravni.

Institut namenja veliko pozornost mednarodnemu sodelovanju. Sodeluje z mnogimi uglednimi institucijami po svetu, organizira mednarodne konference, sodeluje na mednarodnih razstavah. Poleg tega pa po najboljših močeh skrbi za mednarodno izmenjavo strokovnjakov. Mnogi raziskovalni dosežki so bili deležni mednarodnih priznanj, veliko sodelavcev IJS pa je mednarodno priznanih znanstvenikov.

Tekmovanje sta podprla naslednja odseka IJS:

CT3 — Center za prenos znanja na področju informacijskih tehnologij

Center za prenos znanja na področju informacijskih tehnologij izvaja izobraževalne, promocijske in infrastrukturne dejavnosti, ki povezujejo raziskovalce in uporabnike njihovih rezultatov. Z uspešnim vključevanjem v evropske raziskovalne projekte se Center širi tudi na raziskovalne in razvojne aktivnosti, predvsem s področja upravljanja z znanjem v tradicionalnih, mrežnih ter virtualnih organizacijah. Center je partner v več EU projektih.

Center razvija in pripravlja skrbno načrtovane izobraževalne dogodke kot so seminarji, delavnice, konference in poletne šole za strokovnjake s področij inteligentne analize podatkov, rudarjenja s podatki, upravljanja z znanjem, mrežnih organizacij, ekologije, medicine, avtomatizacije proizvodnje, poslovnega odločanja in še kaj. Vsi dogodki so namenjeni prenosu osnovnih, dodatnih in vrhunskih specialističnih znanj v podjetja ter raziskovalne in izobraževalne organizacije. V ta namen smo postavili vrsto izobraževalnih portalov, ki ponujajo že za več kot 500 ur posnetih izobraževalnih seminarjev z različnih področij.

Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi virtualno učečo se skupnost in pripomoči k učinkovitejšemu povezovanju znanosti in industrije ter večji prepoznavnosti domačega znanja v slovenskem, evropskem in širšem okolju.

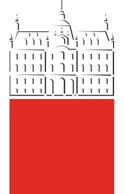
E3 — Laboratorij za umetno inteligenco

Področje dela Laboratorija za umetno inteligenco so informacijske tehnologije s poudarkom na tehnologijah umetne inteligence. Najpomembnejša področja raziskav in razvoja so: (*a*) analiza podatkov s poudarkom na tekstovnih, spletnih, večpredstavnih in dinamičnih podatkih, (*b*) tehnike za analizo velikih količin podatkov v realnem času, (*c*) vizualizacija kompleksnih podatkov, (*d*) semantične tehnologije, (*e*) jezikovne tehnologije.

Laboratorij za umetno inteligenco posveča posebno pozornost promociji znanosti, posebej med mladimi, kjer v sodelovanju s Centrom za prenos znanja na področju informacijskih tehnologij (CT3) razvija izobraževalni portal VideoLectures.NET in vrsto let organizira tekmovanja ACM v znanju računalništva.

Laboratorij tesno sodeluje s Stanford University, University College London, Mednarodno podiplomsko šolo Jožefa Stefana ter podjetji Quintelligence, Cycorp Europe, LifeNetLive, Modro Oko in Envigence.

*



Fakulteta za matematiko in fiziko

Fakulteta za matematiko in fiziko je članica Univerze v Ljubljani. Sestavljata jo Oddelek za matematiko in Oddelek za fiziko. Izvaja dodiplomske univerzitetne študijske programe matematike, računalništva in informatike ter fizike na različnih smereh od pedagoških do raziskovalnih.

Prav tako izvaja tudi podiplomski specialistični, magistrski in doktorski študij matematike, fizike, mehanike, meteorologije in jedrske tehnike.

Poleg rednega pedagoškega in raziskovalnega dela na fakulteti poteka še vrsta obštudijskih dejavnosti v sodelovanju z različnimi institucijami od Društva matematikov, fizikov in astronomov do Inštituta za matematiko, fiziko in mehaniko ter Instituta Jožef Stefan. Med njimi so tudi tekmovanja iz programiranja, kot sta Programerski izziv in Univerzitetni programerski maraton.

Fakulteta za računalništvo in informatiko

Glavna dejavnost Fakultete za računalništvo in informatiko Univerze v Ljubljani je vzgoja računalniških strokovnjakov različnih profilov. Oblike izobraževanja se razlikujejo med seboj po obsegu, zahtevnosti, načinu izvajanja in številu udeležencev. Poleg rednega izobraževanja skrbi fakulteta še za dopolnilno izobraževanje računalniških strokovnjakov, kot tudi strokovnjakov drugih strok, ki potrebujejo znanje informatike. Prav posebna in zelo osebna pa je vzgoja mladih raziskovalcev, ki se med podiplomskim študijem pod mentorstvom univerzitetnih profesorjev uvajajo v raziskovalno in znanstveno delo.

Fakulteta za elektrotehniko, računalništvo in informatiko

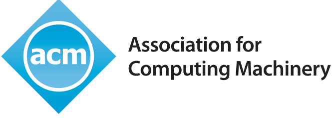
Fakulteta za elektrotehniko, računalništvo in informatiko (FERI) je znanstveno-izobraževalna institucija z izraženim regionalnim, nacionalnim in mednarodnim pomenom. Regionalnost se odraža v tesni povezanosti z industrijo v mestu Maribor in okolici, kjer se zaposluje pretežni del diplomantov dodiplomskih in podiplomskih študijskih programov. Nacionalnega pomena so predvsem inštituti kot sestavni deli FERI ter centri znanja, ki opravljajo prenos temeljnih in aplikativnih znanj v celoten prostor Republike Slovenije. Mednarodni pomen izkazuje fakulteta z vpetostjo v mednarodne raziskovalne tokove s številnimi mednarodnimi projekti, izmenjavo študentov in profesorjev, objavami v uglednih znanstvenih revijah, nastopih na mednarodnih konferencah in organizacijo le-teh.

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Fakulteta za matematiko, naravoslovje in informacijske tehnologije Univerze na Primorskem (UP FAMNIT) je prvo generacijo študentov vpisala v študijskem letu 2007/08, pod okriljem UP PEF pa so se že v študijskem letu 2006/07 izvajali podiplomski študijski programi Matematične znanosti in Računalništvo in informatika (magistrska in doktorska programa).

Z ustanovitvijo UP FAMNIT je v letu 2006 je Univerza na Primorskem pridobila svoje naravoslovno uravnoteženje. Sodobne tehnologije v naravoslovju predstavljajo na začetku tretjega tisočletja poseben izziv, saj morajo izpolniti interese hitrega razvoja družbe, kakor tudi skrb za kakovostno ohranjanje naravnega in družbenega ravnovesja. V tem matematična znanja, področje informacijske tehnologije in druga naravoslovna znanja predstavljajo ključ do odgovora pri vprašanjih modeliranja družbeno ekonomskih procesov, njihove logike in zakonitosti racionalnega razmišljanja.

ACM Slovenija



ACM je največje računalniško združenje na svetu s preko 80 000 člani. ACM organizira vplivna srečanja in konference, objavlja izvirne publikacije in vizije razvoja računalništva in informatike.

ACM Slovenija smo ustanovili leta 2001 kot slovensko podružnico ACM. Naš namen je vzdigniti slovensko računalništvo in informatiko korak naprej v bodočnost.

Društvo se ukvarja z:

- Sodelovanjem pri izdaji mednarodno priznane revije Informatica — za doktorande je še posebej zanimiva možnost objaviti 2 strani poročila iz doktorata.
- Urejanjem slovensko-angleškega slovarčka — slovarček je narejen po vzoru Wikipedije, torej lahko vsi vanj vpisujemo svoje predloge za nove termine, glavni uredniki pa pregledujejo korektnost vpisov.
- ACM predavanja sodelujejo s Solomonovimi seminarji.
- Sodelovanjem pri organizaciji študentskih in dijaških tekmovanj iz računalništva.

ACM Slovenija vsako leto oktobra izvede konferenco Informacijska družba in na njej skupščino ACM Slovenija, kjer volimo predstavnike.

IEEE Slovenija

Inštitut inženirjev elektrotehnike in elektronike, znan tudi pod angleško kratico IEEE (Institute of Electrical and Electronics Engineers) je svetovno združenje inženirjev omenjenih strok, ki promovira inženirstvo, ustvarjanje, razvoj, integracijo in pridobivanje znanja na področju elektronskih in informacijskih tehnologij ter znanosti.

Ministrstvo za izobraževanje, znanost in šport

Ministrstvo za izobraževanje, znanost in šport opravlja upravne in strokovne naloge na področjih predšolske vzgoje, osnovnošolskega izobraževanja, osnovnega glasbenega izobraževanja, nižjega in srednjega poklicnega ter srednjega strokovnega izobraževanja, srednjega splošnega izobraževanja, višjega strokovnega izobraževanja, izobraževanja otrok in mladostnikov s posebnimi potrebami, izobraževanja odraslih, visokošolskega izobraževanja, znanosti, ter športa.

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.

Rtk