18. tekmovanje ACM v znanju računalništva

25. marca 2023

Bilten

Bilten 18. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, Ljubljana, 2024

Elektronska izdaja

Uredil Janez Brank

Avtorji nalog: Benjamin Bajd, Nino Bašić, Urban Duh, Matija Grabnar, Bor Grošelj Simić, Tomaž Hočevar, Gregor Kikelj, Vid Kocijan, Maks Kolman, Filip Koprivec, Matija Likar, Mark Martinec, Polona Novak, Tim Poštuvan, Jakob Schrader, Jure Slak, Mitja Trampuš, Jasna Urbančič, Jaka Velkaverh, Borut Žnidar, Patrik Žnidaršič, Janez Brank.

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

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

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

COBISS.SI-ID=198699267
ISBN 978-961-264-291-4 (PDF)

KAZALO

STRUKTURA TEKMOVANJA

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

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

V lažjih dveh skupinah traja tekmovanje tri ure; tekmovalci lahko svoje rešitve napišejo na papir ali pa jih natipkajo na računalniku, nato pa njihove odgovore oceni temovalna komisija. (Kot običajno se jih je tudi letos velika večina odločila pisati odgovore na računalniku in ne na papir.) Naloge v teh dveh skupinah večinoma zahtevajo, da tekmovalec opiše postopek ali pa napiše program ali podprogram, ki reši določen problem. Pri pisanju izvorne kode programov ali podprogramov načeloma ni posebnih omejitev glede tega, katere programske jezike smejo tekmovalci uporabljati.

V tretji skupini rešujejo vsi tekmovalci naloge na računalnikih, za kar imajo pet ur časa. Pri vsaki nalogi je treba napisati program, ki prebere podatke s standardnega vhoda, izračuna neki rezultat in ga izpiše na standardni izhod. Programe se potem ocenjuje tako, da se jih na ocenjevalnem računalniku izvede na več testnih primerih, število točk pa je sorazmerno s tem, pri koliko testnih primerih je program izpisal pravilni rezultat. (Podrobnosti točkovanja v 3. skupini so opisane na strani 18.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C#, java, python in rust.

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

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

Na začetku smo tekmovalcem razdelili tudi list z nekaj nasveti in navodili (str. 7–9 za 1. in 2. skupino, str. 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 225–228.

Podobno kot v zadnjih nekaj letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 27. januarja 2023. To je imelo eno samo težavnostno skupino, naloge (ki jih je bilo pet) pa so pokrivale precej širok razpon težavnosti. Tekmovalci so dobili enake strani z nasveti in navodili kot na državnem tekmovanju v 1. in 2. skupini (str. 7–9). Odgovore tekmovalcev na posamezni šoli so ocenjevali mentorji z iste šole, za pomoč pa smo jim pripravili nekaj strani z nasveti in kriteriji za ocenjevanje (str. 210–213). Namen šolskega tekmovanja je bil tako predvsem v tem, da pomaga šolam pri odločanju o tem, katere tekmovalce poslati na državno tekmovanje in v katero težavnostno skupino jih prijaviti. Šolskega tekmovanja se je letos udeležilo 215 tekmovalcev z 31 šol (ena osnovna, ostale srednje ter še osnovnošolska skupina za priprave na računalniška tekmovanja).

Podobno kot leta 2022 (in pred tem v letih 2020–21, ko je tekmovanje zaradi epidemije potekalo prek interneta) so imeli tudi letos tekmovalci na voljo prevajalnike in razvojna orodja v vseh skupinah, ne le v tretji.

Letos je v Sloveniji potekalo tudi srednjeevropsko študentsko tekmovanje v računalništvu (CERC 2023), zato v letošnjem biltenu objavljamo tudi rezultate (str. 223) ter slovenske prevode nalog (str. 35–52) in opise rešitev (str. 123–192) s tega tekmovanja.

Letos smo prvič organizirali tudi poskusno osnovnošolsko računalniško tekmovanje. Potekalo je 25. marca 2023 skupaj s srednješolskim, razdeljeno pa je bilo na dve težavnostni skupini; v prvi, lažji, je ocenjevanje ročno (tako kot v prvi in drugi skupini srednješolskega tekmovanja), v drugi, težji, pa avtomatsko (kot v tretji skupini srednješolskega tekmovanja). Sodelovalo je 11 tekmovalcev v prvi in 8 v drugi skupini. Ker je bilo letošnje osnovnošolsko tekmovanje poskusno, so tekmovalci reševali od doma in rezultati ne bodo objavljeni, pač pa v letošnjem biltenu objavljamo tudi naloge (str. 29) in rešitve (str. 114) tega tekmovanja.

NASVETI ZA 1. IN 2. SKUPINO SŠ

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
#include <stdio.h>

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

Še isti trije primeri v pythonu:

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

a, b = sys.stdin.readline().split()
a = int(a); b = int(b)
print(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 SŠ

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

Tekmovanje bo potekalo na strežniku `https://rtk.fri.uni-lj.si/`, kjer dobiš naloge in oddajaš svoje odgovore. Uporabniška imena in gesla vas bodo čakala na mizi v učilnici. Pri oddaji preko računalnika odpreš dotično nalogo v spletni učilnici in rešitev natipkaš oz. prilepiš v polje za programsko kodo. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Ker je vgrajeni urejevalnik dokaj preprost in ne omogoča označevanja kode z barvami, predlagamo, da rešitev pripraviš v kakšnem drugem urejevalniku na računalniku (Visual Studio Code, Geany, Lazarus) in jo nato prekopiraš v okno spletnega urejevalnika. Naj te ne moti, da se bodo barvne oznake kode pri kopiranju izgubile.

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

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

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

1. Neurejene besede

Mojca in Peter sta urednika šolskega časopisa. Kot vsak zaupanja vreden časopis mora tudi šolski časopis imeti rubriko z ugankami. Po pregledu ostalih časopisov sta se Mojca in Peter odločila, da bosta v šolskem časopisu dijakom v reševanje ponudila „zmešane citate“. Pri tej vrsti ugank je podan stavek, kjer so črke v posameznih besedah premešane, tako da nimajo nobenega smisla, naloga reševalca pa je, da ugotovi pravilen vrstni red črk v besedah, ki mu dajo smiseln citat.

Ker se Mojca in Peter ukvarjata z urejanjem šolskega časopisa, ne pa s programiranjem, sta se obrnila nate, ki obiskuješ programerski krožek. Prosita te, da jima **napišeš program**, ki bo iz citatov generiral uganke. Tvoj program kot vhod prebere citat (lahko ga prebere s standardnega vhoda ali iz datoteke `vhod.txt`, karkoli ti je lažje), izpiše pa naj „zmešani citat“, v katerem so črke vsake besede naključno premešane, ostali znaki citata (presledki in ločila) pa ostanejo nespremenjeni. Predpostavi, da je citat dolg največ 100 znakov, da leži v celoti v eni vrstici in da je posamezna beseda sestavljena le iz črk angleške abecede.

Predpostavi, da je za generiranje naključnih števil na voljo funkcija `Random(n)`, ki vrne naključno celo število od 0 do $n-1$ (pri čemer so vsa števila enako verjetna).

Nekaj primerov:

vhod: `Danes je lep, topel dan.`
možen izhod: `nDsae ej lpe, peotl dan.`

vhod: `Pes, ki laja, ne grize.`
možen izhod: `sPe, ik jaal, ne zireg.`

vhod: `cDdDc cddcdc CCD... cdcdd ddDc? ccddc`
možen izhod: `cDcdD ddcccd DCC... dddcc Dcdd? cdccd`

2. Kibi, mebi

Velikost datotek ali pomnilnika običajno merimo v bajtih (B), pri zapisu večjih vrednosti pa si naredimo število preglednejše oz. lažje razumljivo tako, da uporabimo multiplikativne predpone K (kilo-), M (mega-), G (giga-) itd. Tako predstavlja en kilobajt 1024 bajtov (1 KB = 1024 B), megabajt je 1024 kilobajtov (1 MB = 1024 KB) in tako naprej, vsaka naslednja predpona (kot jih določa in poimenuje npr. industrijski standard JEDEC) je za faktor 1024 večja od prejšnje. Te predpone po vrsti so: K, M, G, T, P (za naš namen se ustavimo pri P, čeprav obstajajo tudi višje).

Napiši podprogram (oz. funkcijo), ki bo dobil kot argument velikost neke datoteke v bajtih kot nenegativno celo število, potem pa izpisal to vrednost, po potrebi okrajšano z uporabo najnižje možne predpone tako, da število števk zapisa ne bo večje kot štiri. Če je število tako veliko, da zanj uporabimo najvišjo predpono, potem za tak primer omejitev na štiri števke ne velja.

Če ti je lažje, lahko namesto podprograma napišeš program, ki naj prebere število iz vhodne datoteke ali s standardnega vhoda.

Izpišemo vedno le celi del (brez morebitnih decimalk), temu naj sledi črka B (= bajt), pred katero naj po potrebi stoji črka predpone.

Če število bajtov ni mnogokratnik vrednosti predpone (in bi pri deljenju ostale decimalke), zaokrožimo število navzgor, npr. 234,03 KB izpišemo kot 235 KB, 234,00 KB pa kot 234 KB.

Za potrebe te naloge lahko predpostaviš, da imajo številski podatkovni tipi tvojega programskega jezika neomejen obseg in natančnost.

Primeri:

podatek	izpis
0	`0 B`
5678	`5678 B`
2097152	`2048 KB`
2097153	`2049 KB`
12897500	`13 MB`
128975000	`124 MB`

3. Lučka

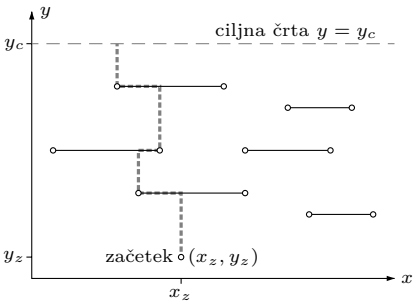
Od podjetja, ki proizvaja namizne lučke, si dobil nalogo napisati program, ki bo nastavil lučko na največjo možno svetlost. Vsaka lučka ima več stopenj svetlosti, zaradi varčevanja pa se je podjetje odločilo, da se svetlost krmili z le eno tipko. Vsakič ko pritisnemo na tipko, se svetlost poveča, razen če je lučka že na največji svetlosti, v tem primeru pa se svetlost ponastavi na najnižjo stopnjo. Ker ne vemo, na kateri stopnji je lučka in koliko stopenj ima, imamo na voljo senzor svetlosti, ki nam pove trenutno svetlost lučke. **Napiši program**, ki bo ob koncu delovanja nastavil lučko na največjo možno svetlost.

Na voljo imaš funkciji PritisniTipko() in PreveriSvetlost(). Funkcija PritisniTipko() simulira pritisk na tipko (in ne vrne ničesar), funkcija PreveriSvetlost() pa vrne trenutno svetlost lučke kot naravno število (za vsako stopnjo svetlosti vedno vrne enako vrednost). Ti dve funkciji sta že napisani in ju ne implementiraš ti. Za vse točke mora tvoj program uporabiti čim manj klicev funkcije PritisniTipko().

4. Oviratlon

Tekmovalec na oviratlonu želi preteči travnik od juga proti severu (torej od manjših y-koordinat proti večjim). Začne na koordinati (x_z, y_z), njegov cilj pa je doseči y-koordinato y_c. Pri tem mu pot ovira n vodoravnih ovir; ovira i (za $i = 1, 2, \ldots, n$) leži na y-koordinati y_i in se po x-koordinati razteza od x_{i1} do x_{i2} (pri čemer je $x_{i1} < x_{i2}$). Vse ovire ležijo po y-koordinati med začetkom in ciljem, torej za vse i velja $y_z < y_i < y_c$.

Tekmovalec bo tekel v ravni črti proti severu, dokler se ne zaleti v oviro. Obšel jo bo po levi ali desni strani, odvisno od tega, katero krajišče mu je bližje (če sta obe krajišči enako oddaljeni od njega, bo izbral levo krajišče, torej tisto z manjšo x-koordinato), in nadaljeval pot na drugi strani od krajišča ovire zopet proti severu. Ovire se med seboj ne dotikajo in se ne prekrivajo (in so dovolj daleč narazen, da gre tekmovalec vedno lahko med njimi oz. okrog njih). Ovire niso nujno podane v kakšnem posebnem vrstnem redu.

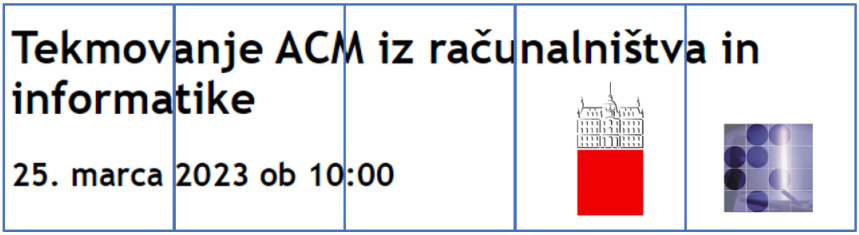


Primer koordinatnega sistema z ovirami (vodoravne daljice) in tekmovalčevo potjo (debela črtkana črta). Tekmovalec začne svojo pot v (x_z, y_z) in nadaljuje, dokler ne doseže ciljne črte $y = y_c$.

Opiši postopek (ali napiši program, če ti je lažje), ki izračuna dolžino poti, ki jo bo tekmovalec pretekel. Kot vhodne podatke tvoj postopek dobi števila x_z, y_z, y_c in n ter (za vsak i od 1 do n) y_i, x_{i1} in x_{i2}. Poleg tega, da opišeš postopek, oceni tudi njegovo časovno zahtevnost (število operacij, ki jih izvede, v odvisnosti od števila ovir n). Ovir je lahko do 100 000, zato je zaželeno, da je tvoj postopek čim bolj učinkovit.

5. Videostena

Na eni od imenitnejših srednjih šol v Ljubljani so se odločili, da bi v telovadnico želeli postaviti velik zaslon, na katerem bi predvajali reklame in druga sporočila, podobno kot vidimo pri športnih tekmovanjih. Ker nimajo denarja za nakup novih, so iz računalniške učilnice nabrali stare, še delujoče računalniške zaslone in jim dodali poceni računalnike, ki krmilijo zaslon in se pogovarjajo z nadzornim računalnikom. Zaslone so pritrdili ob rob telovadnice, pokonci, drugega zraven drugega. Nadzorni računalnik bo celotno sliko „razrezal“ in dele pošiljal na ustrezni zaslon. Primer:



zaslon 140 zaslon 378 zaslon 52 zaslon 6 zaslon 839

Seveda za ta podvig potrebujejo računalniški krožek. Tam so napisali program, ki na posameznem zaslonu ugotovi oba sosednja zaslona in to sporoči nadzornemu računalniku. Te podatke dobiš kot zaporedje trojic števil, kjer drugo število pove številko zaslona, ki sporoča to trojico, prvo število je številka njegovega levega soseda (ali -1, če levega soseda sploh ni), tretje število pa je številka njegovega desnega soseda (ali -1, če desnega soseda sploh ni). Te trojice so navedene v nekem naključnem vrstnem redu, kot kaže naslednji primer za zgornjo sliko:

```
378 52 6
6 839 -1
52 6 839
```

```
-1 140 378
140 378 52
```

Napiši program, ki iz teh podatkov ugotovi vrstni red zaslonov. Številke zaslonov so naravna števila z območja od 1 do 1000, vendar ne nujno točno od 1 do števila zaslonov (kar vidimo tudi na gornjem primeru). Podatke lahko tvoj program bere s standardnega vhoda ali pa iz datoteke `vhod.txt` (karkoli ti je lažje). (Pozor: čeprav je v primeru zgoraj pet zaslonov, naj tvoja rešitev deluje tudi za primere z več ali manj kot petimi zasloni.)

Mogoče je tudi, da v vhodnih podatkih manjka vrstica za kakšen zaslon; v tem primeru naj tvoj program izpiše sporočilo o napaki.

NALOGE ZA DRUGO SKUPINO SŠ

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

1. Stoli

Imamo n stolov v vrsti in n oseb; oboji so oštevilčeni od 1 do n (velja $n \leq 1000$). Osebe bi se rade druga za drugo posedle na stole. O vsaki osebi vemo, na kateri stol bi se najraje usedla (oseba i na stol S_i), poznamo pa tudi njihove želje v primeru, da je ta stol že zaseden. Če je želeni stol že zaseden, se oseba poskuša usesti na najbližji stol, ki ustreza tem zahtevam. Nekatere osebe so se pripravljene premakniti levo od stola, nekatere pa desno od stola. Ker bodo tisti, ki se niso uspeli usesti na želeni stol, slabe volje, imamo za vsako osebo i podano tudi število R_i (ki je večje ali enako 0), ki pove, koliko mest levo in desno od njenega novega sedišča v trenutku, ko se bo usedla, ne sme biti nikogar, da ne bo ta zlovoljna oseba širila negativne energije (oseba, ki se ni mogla usesti na želeni stol, se torej v izbrano smer premika tako daleč, dokler ni v njeni okolici dovolj prostih stolov). Tisti, ki ne morejo zasesti nobenega stola po svojih željah, v jezi odkorakajo domov in ne zasedejo nobenega stola.

Napiši program, ki ugotovi, kakšna je končna razporeditev ljudi na stole. Podatke o osebah naj prebere s standardnega vhoda. V prvi vrstici dobi število oseb n. Sledi n vrstic, za vsako osebo po ena; v i-ti od teh vrstic so s presledkom ločena števila S_i, R_i ter črka `L` ali `D`, ki predstavlja smer, v katero se želi oseba i premakniti, če je njen želeni stol S_i že zaseden.

Tvoj program naj izpiše n s presledkom ločenih števil, kjer i-to število predstavlja i-ti stol. Če na stolu ni nikogar, izpiši 0, sicer pa številko osebe (od 1 do n), ki sedi na njem.

(*Nadaljevanje na naslednji strani.*)

Primer: recimo, da imamo 8 stolov; naslednja tabela kaže, kako bi se posedle prve tri osebe, če bi imele takšne želje, kot je navedeno v levem delu tabele.

oseba i	stol S_i	razmik R_i	smer	Stanje stolov po prihodu te osebe
		začetno stanje →		0 0 0 0 0 0 0 0
1	5	7	L	0 0 0 0 1 0 0 0
2	5	3	L	2 0 0 0 1 0 0 0
3	5	1	L	2 0 3 0 1 0 0 0

2. Tehtnica

Imamo tehtnico z dvema skodelicama. V levo skodelico postavimo paket z neko celoštevilčno težo, za uravnoteženje pa imamo na voljo n uteži, ki jih lahko razporedimo po obeh skodelicah. Uteži imajo teže, ki so potence števila tri, torej npr. za $n = 5$ imamo uteži s težami 1, 3, 9, 27, 81 enot, od vsake po en primerek. Vsako od uteži lahko postavimo ali v nasprotno skodelico od paketa ali v isto skodelico z njim ali pa je ne uporabimo pri uravnoteženju tehtnice.

Napiši program, ki bo prebral število uteži n, potem pa za vsako možno nenegativno celoštevilčno težo paketa, ki se jo še da uravnotežiti z danimi utežmi, izpisal, katere uteži moramo pri tem postaviti v levo skodelico in katere v desno.

Primer izpisa:

```
()     <=> ()     = 0
()     <=> (1)    = 1
(1)    <=> (3)    = 2
()     <=> (3)    = 3
()     <=> (1 3)  = 4
(1 3)  <=> (9)    = 5
(3)    <=> (9)    = 6
...
```

Točen format izpisa ni predpisan, tudi vrstni red ne, važno je le, da je iz izpisa nedvoumno jasno, katere uteži so položene v levo in katere v desno skodelico za vsako možno težo paketa.[1]

3. Konkordanca

Dano je dolgo besedilo v datoteki, razdeljeno na vrstice, dolge po največ 100 znakov. **Napiši program** ali podprogram (oz. funkcijo), ki za dani niz s (sestavljen samo iz črk) poišče vse pojavitve tega niza v besedilu in vsako pojavitev izpiše skupaj z zadnjimi 30 znaki pred njo in prvimi 30 znaki za njo. Pri tem konec vrstice obravnavaj kot presledek. Predpostaviš lahko, da besedilo vsebuje le znake ASCII (črke angleške abecede, številke in ločila). Besedilo lahko bereš s standardnega vhoda ali pa iz datoteke `vhod.txt` (karkoli ti je lažje). Besedilo je lahko dolgo, zato v pomnilniku ni nujno dovolj prostora za hranjenje celotnega besedila.

[1] Zanimiva različica naloge je tudi naslednja: napiši podprogram, ki za dano število uteži n in težo paketa t (ki se nahaja v levi skodelici) izpiše, katere uteži je treba dati v levo in katere v desno skodelico, da bosta uravnoteženi.

Primer za niz „`drevak`“:

```
          (30 znakov)                    (30 znakov)
└──────────────────────────┘ └──────────────────────────┘
rogi bi bil najrajši planil v drevak in se odpeljal v konec Dolge
vredno hoditi iskat.    Nekaj drevakov se je potopilo. Le rilci so
l k vodi. Odbrali so najlepši drevak, izplali vodo, prijeli za ves
bilo jesti."      "Vidim tuje drevake," je spet spregovoril Jelen.
jak je slišal reči Sulca:    "drevak je brez veslača. Je že komu z
opil Ostrorogi v pripravljeni drevak. Vesla so se potopila v skalj
 Sinovi so pa ročno potegnili drevak na pol na breg, ko ga ni bilo
ščave. S trdine pa so veslali drevaki proti koliščem. Veliko je
obrvi. Pogled mu je obstal na drevaku, ki se je pravkar odtrgal iz
```

Opomba: če je najdena pojavitev niza preblizu začetka ali konca datoteke, lahko manjkajoče znake okolice niza nadomestiš s presledki ali pa jih ne izpišeš.

4. Nedeljiva hramba

Mikrokrmilnik periodično odčitava vrednost neke meritve, ki je nenegativno celo število. To število se da shraniti v 32-bitno spremenljivko (to so štirje bajti). Uporabnik lahko kadarkoli izklopi mikrokrmilnik in ga kasneje vklopi nazaj, lahko tudi začasno izpade napajanje. Pomembno pa je, da se vrednost zadnje meritve ohrani tudi ob izpadu, torej da ima mikrokrmilnik ob ponovnem zagonu dostop do zadnjega podatka, ki ga je pred izpadom uspel shraniti.

Lasten hitri pomnilnik za hranjenje podatka prek izpada ni uporaben, ker se njegova vsebina izgubi. Tudi dostopa do diska z datotekami ni. Pač pa imamo na voljo manjši zunanji pomnilnik, ki je sposoben trajno ohranjati svojo vsebino. Ta pomnilnik je velik 1024 bajtov (dovolj in preveč za naš namen, torej ni treba biti pretirano varčen), enota branja in pisanja vanj je en bajt, ta se hrani na naslovu med 0 in 1023.

Konstrukcija zunanjega pomnilnika zagotavlja, da se pisanje bajta izvede celovito (atomično, nedeljivo), tudi če bi sredi operacije pisanja izpadlo napajanje — vrednost bajta se bo torej bodisi zapisala v celoti na zahtevani naslov ali pa se ne bo zapisala in bo ohranjena prejšnja vrednost na tem naslovu. Ne more se torej zgoditi, da bi se zapisalo npr. le nekaj bitov od osmih ali da bi se vsebina kako drugače pokvarila.

Za dostop do trajnega pomnilnika sta na voljo funkciji:

- ShraniBajt(naslov, vrednost)
- PreberiBajt(naslov)

Obe imata kot prvi argument pomnilniški naslov (celo število med 0 in 1023). Funkcija ShraniBajt zapiše na dani naslov en bajt z dano vrednostjo (celo število med 0 in 255), funkcija PreberiBajt pa z danega naslova en bajt prebere in ga vrne (rezultat je torej tudi celo število med 0 in 255). Za tidve funkciji torej predpostavi, da že obstajata, in ju ne poskušaj implementirati sam.

Težava, ki jo moramo rešiti, je, da je podatek z naše meritve dolg štiri bajte in tudi zanj bi potrebovali atomičen način zapisovanja, t.j. da se ob morebitnem izpadu sredi pisanja štirih bajtov ne bi zgodilo, da bi nekaj bajtov obdržalo staro vrednost,

nekaj pa novo, saj bi tako dobili ob branju napačno vrednost (niti staro, niti novo, ampak nekaj pomešanega).

Premisli, kako bi lahko zagotovil atomičen zapis merilne vrednosti v zunanji pomnilnik, in **opiši** podatkovno strukturo in postopek, ki bi to lahko zagotovila.

Napiši tudi tri funkcije:

- NastaviZacetnoStanje() — ta funkcija bo poklicana le ob prvem vklopu mikrokrmilnika in nikoli več. Njena naloga je, da v zunanjem pomnilniku pripravi veljavno začetno podatkovno strukturo, kot si si jo zamislil in opisal.
- ShraniPodatek(podatek) — ta funkcija naj shrani podatek (32-bitno nenegativno celo število) v zunanji pomnilnik in pri tem pazi, da se stari in novi podatek ne bi pomešala, tudi če sredi klica te funkcije izpade delovanje mikrokrmilnika.
- PreberiPodatek() — ta funkcija naj prebere in vrne zadnjo shranjeno vrednost iz zunanjega pomnilnika.

5. Prisotnost

Podjetje je poslalo skupino programerjev na konferenco o najnovejših tehnologijah. Kljub temu, da nekatera izmed n predavanj na konferenci potekajo istočasno, pričakuje direktor podjetja, da bo vsako predavanje poslušala vsaj ena oseba iz delegacije, ki bo lahko pridobljeno znanje predala naprej v podjetju. Za vsako predavanje na konferenci je znano, v katerem časovnem intervalu poteka; i-to predavanje (za $i = 1, 2, \ldots, n$) poteka v intervalu $[z_i, k_i]$, torej od vključno časa z_i do vključno časa k_i.

Zaposleni so se odločili, da si bodo naknadno pogledali posnetke predavanj, pričakovanja direktorja pa bodo zadovoljili z vpisom na seznam prisotnih, ki je na voljo ves čas posameznega predavanja. Raje bodo šli na ogled mesta, med ogledom pa se bo vsake toliko časa nekdo vrnil na prizorišče konference. Kdo bo ta nesrečnež, bodo žrebali vsakič znova. Nesrečni izbranec se bo vpisal med prisotne na vseh predavanjih, ki takrat potekajo, in se takoj vrnil k skupini (za vpis med prisotne torej porabi 0 časa).

Opiši postopek, ki izračuna, najmanj kolikokrat bo moral nekdo od njih tako prekiniti turistični ogled, da bo na vseh predavanjih vsaj eden od njih vpisan med prisotne. Kot vhodne podatke tvoj postopek dobi število predavanj n in čase $z_1, k_1, z_2, k_2, \ldots, z_n, k_n$. Časi so podani kot cela števila v nekih zelo majhnih enotah, zato so lahko to precej velika cela števila. Število predavanj n je lahko do 10^6, zato naj bo tvoj postopek učinkovit.

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO SŠ

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, python ali rust, mi pa jih bomo preverili s prevajalniki FreePascal 3.0.4, GNUjevima `gcc` in `g++` 10.3.0 (ta verzija podpira C++17, novejše različice standarda C++ pa le delno), prevajalnikom za javo iz JDK 17, s prevajalnikom Mono 6.8 za C#, s prevajalnikom `rustc` 1.57 za rust in z interpreterjem za python 3.8.

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2023-3/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi rešitve svojih nalog. Pred začetkom tekmovanja lahko poskusiš oddati katero od nalog iz arhiva `https://putka-rtk.acm.si/tasks/s/test-sistema/list/`. Uporabniško ime in geslo za Putko sta enaki kot za računalnike. Med tekmovanjem lahko vprašanja za tekmovalno komisijo postavljaš prek foruma na Putki (povezava pod „Pogovor o nalogi“ v okvirju „Osnovne informacije“ desno od besedila posamezne naloge).

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

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika (za podrobne nastavitve prevajalnikov na ocenjevalnem strežniku glej `https://putka-rtk.acm.si/info/`. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku in na ocenjevalnem strežniku), prenosnih računalnikov, prenosnih telefonov itd.

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

Ocenjevanje

Vsaka naloga ti lahko prinese od 0 do 100 točk. Vsak oddani program se preizkusi na več testnih primerih; pri tretji nalogi je testnih primerov 20 in vsak je vreden po 5 točk, pri prvi in peti pa je testnih primerov po 10 in vsak je vreden po 10 točk. Pri posameznem testnem primeru dobi program vse točke, če je izpisal pravilen odgovor, sicer pa 0 točk (izjema je tretja naloga, kjer so možne tudi delne točke pri posameznem testnem primeru). Druga in četrta naloga imata točkovanje po

podnalogah, kjer dobi program vse točke za posamezno podnalogo, če pravilno reši vse testne primere tiste podnaloge, sicer pa pri tej podnalogi dobi 0 točk.

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

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

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

Primer naloge (ne šteje k tekmovanju)

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

Primer vhoda:

```
123 456
```

Ustrezen izhod:

```
5790
```

Primeri rešitev:

- V pascalu:

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

- V C-ju:

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

- V C++:

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

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

- 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 SŠ

1. Padalski izlet

Klub športnih padalcev organizira izlet za začetnike, kjer bodo padalsko izkušnjo podali novincem s skokom v tandemu. Cilj izleta je, da čim več začetnikov okusi skok s padalom. Za ta namen mora biti vsak začetnik v paru z enim izkušenim padalcem. Drugi problem izleta je prevoz — za vsakega udeleženca vemo, ali pride z avtom ali ne. Tisti, ki ne pridejo z avtom, se morajo pridružiti nekomu, ki pride z avtom. Udeleženci, ki so brez prevoza ali pa so začetniki brez mentorja, se izleta žal ne morejo udeležiti.

Dobili smo seznam prijavljenih. Za vsakega voznika vemo, koliko ljudi lahko vzame v avto. Prav tako za vsakega izkušenega padalca vemo, s koliko začetniki lahko skoči v tandemu. **Napiši program**, ki ugotovi, največ koliko začetnikov lahko okusi skok s padalom.

Opomba: ni treba, da se oseba pripelje z avtom skupaj z istim soudeležencem, s katerim potem tudi skače; lahko se pripelje z enim in skače z drugim.

Vhodni podatki: v prvi vrstici je naravno število n, število udeležencev. Nato sledi n vrstic; v i-ti od njih je opis i-tega izmed udeležencev z dvema celima številoma a_i in b_i, ločenima s presledkom. Pri tem a_i pove, koliko sopotnikov lahko i-ta oseba vzame v svoj avto. Če je $a_i = -1$, potem ta oseba nima avtomobila in se mora pridružiti nekomu drugemu. Podobno b_i pomeni, koliko so-skakalcev lahko vzame i-ta oseba. Če je $b_i = -1$, je ta oseba začetnik in se mora pridružiti bolj izkušenemu mentorju. Če je $a_i = 0$, to pomeni, da ima oseba prevoz zase, ne more pa vzeti sopotnikov; in podobno $b_i = 0$ pomeni, da ta oseba lahko skoči s padalom, ne more pa skočiti skupaj z nobenim začetnikom.

Omejitve: $1 \leq n \leq 10^5$; za vsak $i = 1, \ldots, n$ velja $-1 \leq a_i \leq 10^5$ in $-1 \leq b_i \leq 10^5$. V prvih 20 % testnih primerov bo veljalo $1 \leq n \leq 15$; v naslednjih 30 % testnih primerov bo veljalo $1 \leq n \leq 1000$; v naslednjih 30 % testnih primerov bo veljalo $a_i, b_i \in \{-1, 1\}$.

Izhodni podatki: izpiši eno samo celo število — koliko največ začetnikov (torej ljudi, za katere je $b_i = -1$) lahko pride na izlet.

Primer vhoda:	Pripadajoči izhod:
5 2 1 -1 1 -1 -1 -1 -1 1 -1	2

Primer vhoda:	Pripadajoči izhod:
8 -1 3 0 -1 -1 2 -1 -1 2 -1 -1 -1 -1 2 -1 -1	3

Komentar: v prvem primeru lahko na izlet vzamemo štiri osebe, izpustimo pa enega izmed začetnikov brez prostora v avtu. V drugem primeru vzamemo 1., 2., 5. osebo in eno izmed 4., 6. in 8. osebe, kar vključuje tri začetnike.

2. Ulične luči

Dana je ulica dolžine m; položaje na njej torej lahko opišemo z x-koordinatami od 0 do m, ki merijo oddaljenost posamezne točke od levega krajišča ulice. Na ulici stoji n luči, pri čemer i-ta od njih (za $i = 1, 2, \ldots, n$) stoji na x-koordinati p_i in ima svetilnost c_i. Luči lahko osvetlimo različno močno: izberemo si neko konstanto a (ki je za vse luči enaka) in potem posamezna luč osvetljuje ulico v razdalji $a \cdot c_i$ od točke, kjer stoji; i-ta luč torej osvetljuje vse točke x z območja $p_i - a \cdot c_i \leq x \leq p_i + a \cdot c_i$. **Napiši program**, ki poišče najmanjši celoštevilski a, pri katerem bo osvetljena celotna ulica (torej: vsako točko x z območja $0 \leq x \leq m$ mora osvetljevati vsaj ena luč).

Vhodni podatki: v prvi vrstici sta celi števili n (število luči) in m (dolžina ulice), ločeni s presledkom. Sledi n vrstic, ki po vrsti opisujejo luči; i-ta od teh vrstic vsebuje celi števili p_i in c_i, ločeni s presledkom. Luči niso nujno podane v kakšnem posebnem vrstnem redu.

Omejitve: $1 \leq n \leq 10^6$; $1 \leq m \leq 10^9$; za vsak $i = 1, 2, \ldots, n$ velja $0 \leq p_i \leq m$ in $1 \leq c_i \leq m$. Vsi p_i so med seboj različni, torej nobeni dve luči nista na isti koordinati.

Točkovanje. Pri tej nalogi je pet podnalog, ki se razlikujejo po dodatnih omejitvah:

- (10 točk) $n \leq 1000$ in $m \leq 1000$;
- (20 točk) $n \leq 1000$;
- (25 točk) $c_i \leq 10$;
- (20 točk) $n \leq 2 \cdot 10^5$;
- (25 točk) brez dodatnih omejitev vhodnih podatkov.

Pri posamezni podnalogi dobiš vse točke, če pravilno rešiš vse testne primere pri njej, sicer pa pri njej ne dobiš nobene točke.

Izhodni podatki: izpiši eno samo vrstico, vanjo pa najmanjše celo število a, pri katerem je osvetljena celotna ulica.

Primer vhoda:	Pripadajoči izhod:
100 3 90 8 10 20 70 30	2

Opomba: pri tem primeru bi bila ulica v celoti osvetljena že pri $a = 5/4$, toda ker naloga zahteva celoštevilsko rešitev, je pravilni odgovor $a = 2$.

3. Špijonaža

V neki tajni službi dela n vohunov, ki so oštevilčeni s celimi števili od 1 do n. Vohuni so organizirani hierarhično: vsak vohun ima natanko enega neposredno nadrejenega, izjema je le vohun številka 1, ki nima nadrejenega in torej stoji na vrhu hierarhije.

Vohun številka 1 je prejel pomemben dokument, z vsebino katerega bi zdaj rad seznanil vse ostale vohune. Zaradi varnosti bodo vohuni širili dokument postopoma, po korakih, pri čemer se v vsakem koraku zgodi ena od naslednjih stvari:

(1) vohun, ki trenutno ima svoj izvod dokumenta, lahko naredi eno kopijo tega dokumenta in to kopijo izroči enemu od svojih neposredno podrejenih vohunov, vendar le takemu, ki takšne kopije ni prejel že kdaj prej;

(2) vohun, ki trenutno ima svoj izvod dokumenta, lahko ta izvod uniči.

V posameznem trenutku lahko torej obstaja več izvodov dokumenta — v koraku tipa 1 se število izvodov poveča za 1, v koraku tipa 2 pa zmanjša za 1. Vohuni bi radi poskrbeli, da bo vsak od njih nekoč prejel svoj izvod dokumenta, pri tem pa želijo, da hkrati obstaja čim manj izvodov (torej: da bi bilo največje število dokumentov, ki bodo kdajkoli obstajali istočasno, najmanjše možno). **Napiši program**, ki prebere podatke o hierarhiji vohunov in izpiše zaporedje korakov, s katerim lahko to dosežejo.[2]

Vhodni podatki: v prvi vrstici je število vohunov n. Sledi $n-1$ vrstic, od katerih i-ta vsebuje številko vohuna, ki je neposredno nadrejen vohunu številka $i+1$. (To so torej podatki o neposredno nadrejenih za vohune od 2 do n; za vohuna 1 takega podatka ni, ker zanj že vemo, da neposredno nadrejenega sploh nima.) Zagotovljeno je, da je vohun številka 1 posredno ali neposredno nadrejen vsem ostalim.

Izhodni podatki: v prvo vrstico izpiši dve celi števili, ločeni s presledkom. Prvo naj bo število korakov v tvojem zaporedju, drugo pa največje število izvodov dokumenta, ki pri tvoji rešitvi kdajkoli obstajajo istočasno. Sledi naj za vsak korak tvojega zaporedja po ena vrstica, ki vsebuje dve celi števili (ločeni s presledkom), ki opisujeta ta korak:

- `1` v — korak tipa 1: vohun v prejme svoj izvod dokumenta od svojega neposredno nadrejenega vohuna;
- `2` v — korak tipa 2: vohun v uniči svoj izvod dokumenta.

Če je možnih več enako dobrih rešitev, je vseeno, katero od njih izpišeš.

Alternativa: če hočeš, lahko izpišeš le prvo vrstico, pri čemer pa namesto števila korakov napišeš -1 (glej drugi primer spodaj). Takšna rešitev dobi pri trenutnem testnem primeru 60 % točk.

[2] Zanimiva je tudi naslednja težja različica naloge: recimo, da lahko na začetku tudi izberemo vohuna r, ki bo kot prvi prejel dokument; da bodo potem sploh lahko vsi vohuni sčasoma prejeli svoj izvod dokumenta, pa moramo korak tipa 1 dopolniti tako, da sme pri njem vohun namesto enemu od svojih podrejenih izročiti kopijo dokumenta svojemu nadrejenemu (če ta doslej še ni prejel kopije dokumenta). Pri prvotni različici naloge je bilo torej vedno $r = 1$, morda pa je mogoče največje število hkrati obstoječih izvodov dokumenta še kaj zmanjšati, če za r namesto 1 izberemo kakšnega drugega vohuna. Opiši postopek, ki poišče najboljši r.

Omejitve: $1 \leq n \leq 10^5$. Pri prvih 30 % testnih primerov velja tudi $n \leq 8$. Pri naslednjih 20 % testnih primerov velja $n \leq 1000$.

Primer vhoda:	Eden od možnih pripadajočih izhodov:

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

Še en primer vhoda:	Izhod za 60 % točk:

`10`
`5`
`1`
`7`
`1`
`5`
`1`
`5`
`4`
`7`
	`-1 3`

Komentar: pri rešitvi prvega primera obstajata po največ dva izvoda dokumenta hkrati. Po prvem koraku imata svoj izvod vohuna 1 in 2; v drugem koraku vohun 2 svoj izvod uniči; po tretjem koraku pa imata svoj izvod vohuna 1 in 3. Tako je vsak vohun nekoč prejel svoj izvod dokumenta, nikoli pa nista obstajala več kot dva izvoda dokumenta hkrati.

4. Valj

Dan je valj, na plašču katerega je karirasta mreža velikosti $n \times m$ (n vrstic in m stolpcev). Vsak kvadratek je pobarvan z neko barvo. Zanima nas, ali lahko valj obkrožimo, če se premikamo samo po kvadratkih ene barve. **Napiši program**, ki ugotovi, za katere barve to lahko storimo.

Valj z neko potjo obkrožimo, če pot začnemo in končamo v isti točki, pri tem pa se premaknemo naokrog celega valja. Pri tem ni nujno, da se gibljemo zgolj v eno smer (glej drugi primer spodaj).

Obhod v desno definiramo kot pot, ki se začne in konča na istem polju in kjer je vsaka navpična oz. „poldnevnik“ (meja med dvema stolpcema v mreži) valja prečkana enkrat več v desno kot v levo. Definicija obhoda v levo je simetrična. Med kvadratki se lahko premikamo čez stranice, ne pa po diagonalah.[3]

Vhodni podatki: v prvi vrstici sta celi števili n in m, višina in širina mreže. Sledi n vrstic s po m celimi števili $a_{i,j}$, kjer vsaka vrednost predstavlja svojo barvo.

Izhodni podatki: izpiši seznam vseh barv, po katerih lahko obkrožimo valj; urejene naj bodo naraščajoče in ločene s presledkom. Če se obhoda ne da narediti po poljih nobene barve, izpiši „`mavrica`“ (brez narekovajev).

Omejitve: $2 \leq n \leq 1000$; $2 \leq m \leq 1000$; $0 \leq a_{i,j} \leq 10^6$.

Točkovanje. Pri tej nalogi so štiri podnaloge, ki se razlikujejo po dodatnih omejitvah:

[3] Zanimivo in malo težjo različico naloge dobimo, če si predstavljamo, da je valj na začetku neobarvan, nato pa barvamo kvadratke enega po enega. Podano je torej zaporedje trojic $(i, j, a_{i,j})$, ki nam povedo, da v naslednjem koraku pobarvamo j-ti kvadratek v i-ti vrstici z barvo $a_{i,j}$. Pri tem nikoli ne barvamo kvadratka, ki smo ga že pobarvali kdaj prej. Naša naloga je ugotoviti, kdaj v tem zaporedju barvanj prvič nastane obhod okrog valja po kvadratkih ene barve (pri tem seveda neobarvani kvadratki ne pridejo v poštev, saj je obhod po njih mogoč že na samem začetku).

Še težjo različico naloge pa dobimo, če zaporedje vseh barvanj ni podano že vnaprej, pač pa moramo po vsakem pobarvanem kvadratku sproti povedati, ali zdaj obhod že obstaja ali še ne.

- (30 točk) $2 \leq n \leq 5$ in $2 \leq m \leq 5$;
- (20 točk) $2 \leq n \leq 100$ in $2 \leq m \leq 100$;
- (20 točk) na valju bosta le dve barvi, 0 in 1;
- (30 točk) brez dodatnih omejitev vhodnih podatkov.

Pri posamezni podnalogi dobiš vse točke, če pravilno rešiš vse testne primere pri njej, sicer pa pri njej ne dobiš nobene točke.

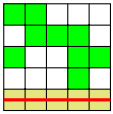
Primer vhoda:

```
5 5
1 1 0 0 0
0 1 1 1 0
1 0 0 1 1
0 0 0 1 0
2 2 2 2 2
```

Pripadajoči izhod:

```
2
```

Slika:



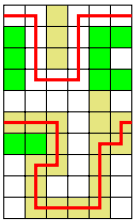
Še en primer vhoda:

```
10 6
0 0 2 0 0 0
1 0 2 0 1 1
1 0 2 0 1 0
1 0 0 0 1 1
0 0 0 0 2 0
2 2 2 0 2 2
1 1 2 0 2 2
0 2 2 0 2 0
0 2 0 0 2 0
0 2 2 2 2 0
```

Pripadajoči izhod:

```
0 2
```

Slika:



Rešitvi obeh primerov sta prikazani na slikah na desni; rdeče črte prikazujejo možne poti okrog valja.

5. Urejanje z medianami

V skladišču je v vrsti n podstavkov, oštevilčenih od 1 do n. Na vsakem podstavku stoji po en zaboj, zaboji pa so različno težki (nobena dva nista enako težka). Zaboje bi radi uredili po teži, pri čemer nam je vseeno, ali bodo urejeni naraščajoče (najlažji zaboj na podstavku 1, drugi najlažji na podstavku 2, ..., najtežji na podstavku n) ali padajoče (najtežji zaboj na podstavku 1, drugi najtežji na podstavku 2, ..., najlažji na podstavku n). Teže posameznih zabojev ne poznamo; za urejanje zabojev bomo morali uporabiti sistem robotskih rok v skladišču, ki podpira le dva tipa operacij:

1. lahko mu podamo številki dveh podstavkov in zahtevamo, naj zamenja zaboja na teh dveh podstavkih (tako da zaboj z enega podstavka pride na drugega in obratno);

2. lahko mu podamo številke treh podstavkov in zahtevamo, naj nam pove, na katerem od teh treh podstavkov stoji zaboj, ki je po teži srednji med zaboji na teh treh podstavkih (torej ki ni niti najlažji niti najtežji izmed njih).

Dodatna težava je, da se podstavki, ki sodelujejo v operaciji drugega tipa, pri tem nekoliko obrabijo. Definirajmo *obrabo* podstavka kot število operacij drugega tipa, pri katerih je bil ta podstavek udeležen. **Napiši program**, ki uredi zaboje po teži

(lahko naraščajoče ali padajoče), pri tem pa pazi, da obraba nobenega podstavka ne bo prevelika.

To je interaktivna naloga; tvoj program se bo z ocenjevalnim strežnikom „pogovarjal“ tako, da bo bral s standardnega vhoda in pisal na standardni izhod. Ta pogovor naj poteka po naslednjih korakih:

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

2. Nato lahko izvedeš 0 ali več operacij. Operacijo izvedeš tako, da na standardni izhod izpišeš vrstico takšne oblike:

   - $a\ b\ -1$ — če hočeš zamenjati zaboja na podstavkih a in b;
   - $a\ b\ c$ — če hočeš poizvedeti, na katerem od podstavkov a, b in c je zaboj, ki je po teži srednji med temi tremi zaboji. Pri tej operaciji moraš nato s standardnega vhoda prebrati vrstico, v kateri boš dobil odgovor (eno od števil a, b in c).

   Pri tem morajo biti a, b in (pri operaciji drugega tipa) c različna cela števila z območja od 1 do n.

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

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

Omejitve: $1 \leq n \leq 1000$. Pri prvih 50 % testnih primerov bo veljalo tudi $n \leq 100$.

Točkovanje: če se tvoj program ne drži zgoraj opisanega protokola, ne dobi pri tem testnem primeru nobene točke; enako velja tudi, če izvede več kot 40 000 operacij ali če na koncu izvajanja zaboji niso urejeni niti naraščajoče niti padajoče. Sicer pa je število točk odvisno od obrabe najbolj obrabljenega podstavka (torej od tega, v koliko največ operacijah drugega tipa je sodeloval kakšen podstavek). Če je ta maksimalna obraba največ 20, dobiš pri tem testnem primeru 10 točk; če je od 21 do 100, dobiš 9 točk; če je od 101 do 500, dobiš 8 točk; če je od 501 do 2000, dobiš 7 točk; če pa je več kot 2000, dobiš 5 točk.

Primer:

Tvoj program izpiše	Sistem izpiše	Komentar
	`4`	v skladišču so štirje zaboji
`2 3 4`	`4`	med podstavki 2, 3, 4 je srednji po teži zaboj na podst. 4
`3 4 -1`		zamenjamo zaboja na podstavkih 3 in 4
`1 2 3`	`1`	med podstavki 1, 2, 3 je srednji po teži zaboj na podst. 1
`2 1 -1`		zamenjamo zaboja na podstavkih 1 in 2
`-1`		končali smo z urejanjem

Zaboji so na koncu tega primera urejeni tako, kot naloga zahteva.

NALOGE ZA ŠOLSKO TEKMOVANJE SŠ

27. januarja 2023

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. Cikcakasti nizi

Recimo, da razdelimo niz znakov na krajše kose tako, da ga prerežemo povsod tam, kjer sta si dva sosednja znaka različna. Na primer:

`aabbacccaab`	$\rightarrow$	`aa` | `bb` | `a` | `ccc` | `aa` | `b`
`ggghddddoouuug`	$\rightarrow$	`ggg` | `h` | `dddd` | `oo` | `uuu` | `g`

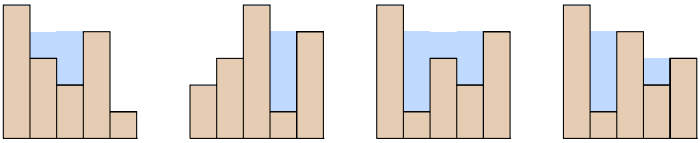
Označimo dolžine teh kosov po vrsti z d_1, d_2, d_3 in tako naprej. Rekli bomo, da je niz *cikcakast*, če te dolžine izmenično naraščajo in padajo; z drugimi besedami, če velja $d_1 < d_2 > d_3 < \ldots$ ali pa $d_1 > d_2 < d_3 > \ldots$.

Napiši program ali podprogram (funkcijo), ki kot vhod dobi niz znakov in ugotovi, ali je niz cikcakast. Niz lahko dobi kot parameter ali pa ga prebere s standardnega vhoda ali iz datoteke (karkoli ti je lažje). Predpostavi, da v nizu nastopajo le male črke angleške abecede (od `a` do `z`), lahko pa je zelo dolg, zato naj bo tvoja rešitev čim bolj učinkovita.

Nekaj primerov: nizi `aabaaa`, `cbbbabb` in `abbcdda` so cikcakasti; nizi `aabbccc`, `abbaaa` in `cdddccdcc` pa niso cikcakasti.

2. Histogram

Histogram je zaporedje n stolpcev, ki so različno visoki, vendar enako široki (po 1 enoto) in se držijo skupaj. Če od zgoraj na histogram pada dež, se zato nabira voda tam, kjer stoji več nižjih stolpcev med dvema višjima. Naslednja slika kaže štiri primere histogramov za $n = 5$:



Vidimo lahko, da je količina vode, ki se nabere v histogramu, lahko različna v odvisnosti od oblike histograma. Pri prvih dveh histogramih na gornji sliki se je nabralo po 3 enote vode, pri tretjem histogramu 6 enot vode, pri četrtem histogramu pa 4 enote vode.

Napiši program ali podprogram (funkcijo), ki kot vhod dobi zaporedje višin stolpcev in izračuna skupno količino vode, ki se nabere v takem histogramu. Višine lahko prebereš s standardnega vhoda ali iz datoteke ali pa predpostaviš, da jih dobiš kot parameter v nekem seznamu, tabeli, vektorju ali čem podobnem (karkoli ti je lažje). Višine stolpcev so naravna števila in nobena dva stolpca nista enako visoka.[4]

Pozor: čeprav so zgoraj primeri histogramov s 5 stolpci, mora tvoja rešitev delovati za poljubno število stolpcev (poljuben n)!

3. Naredimo hitro testiranje zares hitro!

Veliko ljudi se zanima za hitro testiranje na COVID-19, zato so te organizatorji prosili, da jim pomagaš izračunati, koliko točk za testiranje potrebujejo. Testirati začnemo ob 7:00, testirati se želi n ljudi, testiranje posameznega človeka pa traja po t sekund. Vsak človek je podal tudi omejitev, do kdaj hoče zaključiti s testiranjem: testiranje i-tega človeka (za $i = 1, 2, \ldots, n$) mora biti končano najkasneje ob času $h_i{:}m_i$ (v urah in minutah). **Opiši postopek** (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki izračuna, najmanj koliko točk za testiranje potrebujemo, da zadostimo vsem omejitvam. Kot vhodne podatke tvoj postopek dobi n (število ljudi), t (čas testiranja v sekundah) in omejitve $h_1 : m_1$, $\ldots$, $h_n : m_n$. Podrobnosti tega, v kakšni obliki dobi te podatke, si izberi sam in jih v svoji rešitvi tudi opiši. Predpostavi, da so vhodni podatki taki, da rešitev gotovo obstaja. Dobro tudi **utemelji**, zakaj naj bi bil rezultat, ki ga tvoj postopek izračuna, pravilen.

Primer: recimo, da imamo $n = 3$ ljudi, da testiranje enega človeka traja $t = 220$ sekund in da želi biti prvi človek gotov s testiranjem do 7:08, drugi do 7:04 in tretji do 7:05. Potem potrebujemo vsaj dve točki za testiranje (ki obe ob 7:00 začneta s testiranjem, ena z drugim človekom in ena s tretjim, zatem pa ena od njiju testira še prvega človeka); z eno samo točko za testiranje ne bi šlo (ne glede na to, v kakšnem vrstnem redu bi testirali te tri ljudi, gotovo vsaj kakšen od njih ne bi bil testiran tako kmalu, kot je hotel).

4. Palindromi

Palindrom je niz, ki se ne spremeni, če njegove znake preberemo z desne proti levi namesto z leve proti desni, na primer *radar* ali *neradodaren*. **Napiši program ali podprogram** (funkcijo), ki za dani niz prešteje, koliko njegovih podnizov, dolgih vsaj 2 znaka, je palindromov. Niz lahko dobiš kot parameter ali pa ga prebereš s standardnega vhoda ali iz datoteke (karkoli ti je lažje). Predpostavi, da je niz sestavljen le iz malih črk angleške abecede. Zaželeno je, da je tvoj postopek učinkovit, tako da bo deloval hitro tudi za dolge nize (npr. nekaj deset tisoč znakov).

Primer: pri nizu `abccbbba` je pravilni odgovor 5. Palindromni podnizi dolžine vsaj 2 znaka so v tem primeru naslednji: `abccbbba`, `abccbbba`, `abccbbba`, `abccbbba`, `abccbbba`.

[4] Zanimiva je tudi naslednja težja različica naloge: pri izbranem številu stolpcev n obstaja $n!$ takih histogramov, ki imajo n stolpcev s celoštevilskimi širinami od 1 do n (pri čemer nobena dva stolpca nista enako visoka). Opiši postopek, ki za dani števili n in a izračuna, pri koliko histogramih izmed teh $n!$ histogramov se nabere natanko a enot vode. Predpostaviš lahko, da je $n \leq 30$.

5. Čarobne jame

Henrik raziskuje sistem jam v računalniški igrici. Sistem je sestavljen iz n jam, ki so oštevilčene s celimi števili 1 do n. Henrik začne svojo pot v neki konkretni začetni jami s, priti pa želi do končne jame t. Jame so povezane z m prehodi, pri čemer vsak prehod neposredno povezuje dve jami; Henrik se lahko po prehodih premika v obe smeri, kolikorkrat želi. V nekaterih jamah je tudi skrinja z magičnim zvitkom, ki se ob odprtju skrinje takoj uniči, hkrati pa tudi odpre prehod med dvema drugima jamama (ti dve jami nista nujno povezani z jamo, v kateri je zvitek našel).

Opiši postopek (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki za dane podatke o jamah, prehodih in zvitkih ugotovi, ali je sploh mogoče priti od jame s do jame t. Poleg tega tudi oceni časovno zahtevnost svojega postopka oz. približno število operacij, ki jih potrebuje, da pride do rezultata (v odvisnosti od števila jam n in števila prehodov m).

Poleg števil n (število jam), m (število prehodov v začetnem stanju sistema, pred uporabo magičnih zvitkov), s (številka začetne jame) in t (številka končne jame) dobiš kot vhodne podatke še štiri tabele oz. sezname a, b, c in d, ki povedo naslednje: i-ti prehod (za $i = 1, \ldots, m$) neposredno povezuje jami $a[i]$ in $b[i]$; v i-ti jami (za $i = 1, \ldots, m$) je skrinja z zvitkom, ki ustvari prehod med jamama $c[i]$ in $d[i]$ (če i-ta jama sploh ne vsebuje skrinje z zvitkom, bo $c[i] = d[i] = -1$).

NALOGE ZA PRVO SKUPINO OŠ

1. Trikotniki

Učitelj matematike je v šoli za domačo nalogo naročil učencem, naj doma narišejo nekaj trikotnikov in zapišejo dolžine njihovih stranic. Zdaj pregleduje njihove domače naloge in ugotavlja, da so si nekateri učenci oddana števila kar izmislili.

Napiši program, ki prebere tri pozitivna cela števila, urejena po velikosti od najmanjšega do največjega, in izpiše „`trikotnik`“ (brez narekovajev), če je mogoče narisati trikotnik s takšnimi dolžinami stranic, sicer pa naj izpiše „`ponaredek`“.

Trije primeri vhoda:	Pripadajoči izhodi:
`4 4 4`	`trikotnik`
`3 4 5`	`trikotnik`
`1 4 6`	`ponaredek`

2. Daljnovod čez podeželje

Napetost v daljnovodni vrvi (žici) daljnovoda je tako visoka, da jih ni praktično izolirati z gumo tako kot žice, ki jih srečamo vsak dan. Namesto tega jih obesimo zelo visoko in se zanašamo na to, da ni ničesar blizu njih, kamor bi lahko tok stekel; že previsoka veja drevesa lahko povzroča težave.

Imamo niz, dolg n znakov, ki predstavlja neki daljnovod. Znak „`T`“ predstavlja stolp daljnovoda, znak „`_`“ (podčrtaj) prazno polje, ki ne povzroča težav, znak „`o`“ pa krošnjo previsokega drevesa. Daljnovod se začne pri prvem `T`-ju niza in konča pri zadnjem. Znakom med dvema zaporednima stolpoma rečemo *razpetina*.

Napiši program, ki za dani n in niz daljnovoda pove, na koliko razpetinah obstaja kakšna previsoka krošnja.

Primer vhoda:	Pripadajoči izhod:
`20` `__o__T__TT_o_o_ToT__`	`2`

Komentar: v primeru je pred prvim stolpom krošnja, ki nas ne zanima, saj se daljnovod začne šele s prvim stolpom. Razpetini, ki nas motita, sta med tretjim in četrtim stolpom ter med četrtim in petim stolpom.

3. Predor

Malokdo ve, da se policija za lovljenje prehitrih voznikov poleg uporabe tako imenovanih radarjev poslužuje še enega premetenega načina. Na začetku in koncu predora postavijo čitalec registrskih številk, nato pa merijo, kdaj je vozilo prišlo v predor in kdaj je zapeljalo iz njega. Če je izmerjen potovalni čas premajhen, vedo, da je vozilo šlo prehitro, in vozniku izstavijo kazen. Za delovanje sistema poleg čitalcev seveda potrebujejo tudi program, ki je sposoben iz dobljenih podatkov izračunati, katera vozila so šla prehitro. Za pisanje tega programa so se obrnili nate.

Na vhodu se v prvi vrstici nahajata števili n in t, ki predstavljata število zaznanih vozil in minimalni čas, ki ga mora vozilo porabiti za vožnjo skozi predor. Sledi n vrstic, ki opisujejo vozila. V vsaki od njih se nahaja registrska številka vozila, ki je zaporedje šestih malih črk angleške abecede, ter števili z in k, ki povesta, kdaj je vozilo prišlo v predor in kdaj ga je zapustilo. Vsi časi so podani od začetka merjenja. **Napiši program**, ki izpiše registrske številke tistih vozil, ki so skozi predor šla prehitro in jim je sedaj treba napisati kazen.

Primer vhoda:

```
2 3
abcdef 0 4
ghijkl 3 5
```

Pripadajoči izhod:

```
ghijkl
```

4. Besede

Sara bi rada iz kupa modelčkov črk sestavila besede in jih nato nanizala na vrvico. Ker pa ima na voljo le omejeno število modelčkov, bi rada besede nizala tako, da bi zadnjo črko ene besede in prvo črko naslednje besede združila in s tem prihranila en modelček. Za besedi `banana` in `avtomehanik` bi tako na vrvico nanizala `bananavtomehanik`. Sara je že izbrala besede, ki jih želi nanizati, zdaj pa jo zanima, ali so postavljene v pravilen vrstni red, da bo ob vsakem stikanju lahko eno črko na ta način izpustila. Ker je Sarinih besed kar precej, te je prosila, da zanjo **napišeš program**, ki bo ugotovil, ali besede ustrezajo njenim zahtevam.

Na vhodu je v prvi vrstici podan n — število besed, ki jih želi Sara nanizati na vrvico. Sledi n vrstic, v vsaki po ena beseda. Besede bodo vsebovale le male črke angleške abecede, dolge pa bodo kvečjemu 15 znakov.

Tvoj program naj izpiše `ustreza`, če je besede mogoče nanizati v danem zaporedju po Sarinem postopku, sicer pa izpiši `ne ustreza`.

Razmisli in **opiši** še, kako bi ugotovil, ali je mogoče dane besede preurediti, da jih bo Sara lahko nanizala.

Primer vhoda:

```
3
banana
avtomehanik
kenguru
```

Pripadajoči izhod:

```
ustreza
```

Še en primer vhoda:

```
2
kolo
mleko
```

Pripadajoči izhod:

```
ne ustreza
```

NALOGE ZA DRUGO SKUPINO OŠ

1. Napačna imena

Učitelj Uroš uči veliko učencev, zato ima težave pri pomnenju imen vseh učencev. Pogosto se mu zgodi, da koga pokliče po napačnem imenu. Prosil te je, da mu napišeš program, s katerim lahko preveri, če je ime pravilno in kje je morebitna napaka.

Napiši program, ki prebere dva niza; prvi je pravilno ime, drugi pa ime, ki se ga je spomnil Uroš. Izpiši „`pravilno ime`“ (brez narekovajev), če je Uroševo ime pravilno, drugače pa zaporedni položaj prve črke, ki se ne ujema med Uroševim in pravilnim imenom.

Vhodni podatki: v prvi vrstici je nahaja pravilno ime, v drugi pa ime, ki se ga Uroš spomni. Obe imeni imata največ 20 znakov, sta gotovo enake dolžine in sta sestavljeni samo iz črk angleške abecede.

Izhodni podatki: izpiši zaporedni položaj prve črke, v kateri se imeni razlikujeta (prva črka ima položaj 1); če sta enaki, pa izpiši „`pravilno ime`“.

Primer vhoda:	Pripadajoči izhod:
`Janez` `Janez`	`pravilno ime`

Še en primer vhoda:	Pripadajoči izhod:
`Timon` `Timor`	`5`

2. Ceneno potovanje

Letos imamo zaradi povišanih stroškov bivanja manj denarja za dopust. Ker si še vedno želimo iti na potovanje okoli sveta, smo se odločili, da za izbiro lokacij uporabimo posebno strategijo. Začeli bomo na ljubljanskem letališču, kjer bomo kupili najcenejšo možno karto. Ko se bomo nagledali te destinacije, bomo tudi tam kupili najcenejšo letalsko karto in odleteli naprej. Postopek bomo ponavljali, dokler se ne bomo nagledali dovolj mest, pristali nazaj v Ljubljani ali pa na letališču, od koder ne moramo nadaljevati.

Napiši program, ki bo iz podatkov o cenah letov izračunal, kje bomo pristali, če k-krat kupimo najcenejšo letalsko karto in se odpeljemo na to destinacijo. Če pred k-tim letom pristanemo nazaj na začetku, naj se program tam konča.

V nalogi so mesta predstavljena z zaporednimi številkami med vključno 1 in 1000. Ljubljansko letališče, kjer potovanje začnemo, ima vedno številko 1.

Vhodni podatki. V prvi vrstici bosta dani števili n in k. V i-ti od naslednjih n vrstic so podana števila a_i, b_i in c_i, ločena s presledkom; ta zapis pomeni, da enosmerna letalska karta iz mesta a_i v mesto b_i stane c_i evrov.

Zagotovljeno bo, da so vse cene letov izven nekega letališča med seboj različne (lahko pa imata dva leta iz različnih letališč enako ceno).

V 50 % testnih primerov bo na vsakem letališču mogoče kupiti največ eno letalsko karto (torej ne bomo imeli izbire, kam gremo v naslednjem koraku).

Omejitve: $2 \leq n \leq 10^5$; $1 \leq k \leq 10^5$; za vsak $i = 1, 2, \ldots, n$ bo veljalo tudi $1 \leq a_i \leq n$, $1 \leq b_i \leq n$ in $1 \leq c_i \leq 10^9$.

Izhodni podatki. Program naj izpiše dve števili, vsako v svojo vrstico: na katerem letališču s potovanjem končamo ter kolikokrat smo na celotnem potovanju leteli z letalom. Če smo postopek uspešno opravili, mora program torej kot drugo število izpisati k, če se je naša pot končala predčasno, pa manjše število.

Štirje primeri vhodov in pripadajočih izhodov:

Vhod:	Izhod:
8 3 1 2 500 2 3 174 3 1 200 12 7 100 1 12 120 1 3 350 7 3 500 7 2 300	2 3

Vhod:	Izhod:
3 2 1 2 120 2 3 130 3 1 140	3 2

Vhod:	Izhod:
3 500 1 2 2 2 1 2 2 3 3	1 2

Vhod:	Izhod:
5 10 4 3 20 5 4 2 1 5 3 1 5 10 5 4 200	3 3

3. Barvne packe

Na belo platno mečemo balončke, polne različnih barv. Preberi, koliko, kako velike balončke in kam smo jih vrgli, ter nariši, kako izgleda končna risba.

Prazno platno, na primer dolžine 10 in višine 5, si predstavljamo kot mrežo, ki jo lahko predstavimo s poljem samih pik:

```
..........
..........
..........
..........
..........
```

Če na tretji znak tretje vrstice vržemo balon neke barve, ki jo bomo označili z `#`, in velikosti 1, se bo razpočil in naredil packo v obliki kvadrata v razdalji 1 v vsako smer:

```
..........
.###......
.###......
.###......
..........
```

Če nato vržemo še en balon, npr. velikosti 0 in neke druge barve `*`, v četrto vrstico na četrto mesto, dobimo naslednjo risbo:

```
..........
.###......
.###......
.##*......
..........
```

Če vržemo nato še enega velikosti 2 in barve `?` v drugo vrsto, na zadnji znak, dobimo:

```
.......???
.###...???
.###...???
.##*...???
..........
```

Pri zadnjem primeru gre nekaj packe tudi mimo platna, ampak ta del nas ne zanima, saj ga ne bo na končni sliki.

Napiši program, ki prebere podatke o metih balonov in izriše končno stanje platna.

Vhodni podatki. V prvi vrstici so tri s presledkom ločena števila, d, v in p, ki predstavljajo dolžino ter višino platna in število metov. V vsaki izmed naslednjih p vrstic je opis enega meta; i-ta od teh vrstic vsebuje tri cela števila in en znak, x_i, y_i, s_i in b_i. Število y_i pove, v katero vrstico platna smo vrgli balon, število x_i pa, na katero mesto v tej vrstici. Število s_i predstavlja velikost packe, ki jo naredi ta balon, znak b_i pa njeno barvo. Mete izvajamo v takem zaporedju, kot so podani.

Omejitve:

- $1 \leq d \leq 1000$; $1 \leq v \leq 1000$; $0 \leq p \leq 100$;
- za vsak $i = 1, 2, \ldots, p$ bo veljalo $0 \leq s_i \leq 100$, $1 \leq x_i \leq d$, $1 \leq y_i \leq v$, znak b_i pa bo eden izmed znakov „`+`“, „`-`“, „`?`“, „`#`“, „`*`“, „`%`“ in „`$`“.

Dodatne omejitve:

- V prvih 30 % primerov bo višina platna enaka 1 (to pomeni $v = 1$), velikost packe s_i bo vedno 0, barva b_i bo vedno `#` in packa ne bo nikoli gledala prek roba platna.
- V naslednjih 50 % (skupaj 80 %) testnih primerov bo višina platna enaka 1, velikosti, barve in položaji pack pa so lahko poljubni.
- V zadnjih 20 % testnih primerov ni dodatnih omejitev.

Izhodni podatki: izpiši celotno platno po tem, ko smo nanj vrgli vse balone.

Primer vhoda:

```
10 1 3
5 1 0 #
8 1 0 #
1 1 0 #
```

Pripadajoči izhod:

```
#...#..#..
```

Še en primer vhoda:

```
20 1 3
2 1 3 *
20 1 0 $
6 1 1 #
```

Pripadajoči izhod:

```
*******............$
```

Še en primer vhoda:

```
10 5 3
3 3 1 #
4 4 0 *
10 2 2 ?
```

Pripadajoči izhod:

```
.......???
.###...???
.###...???
.##*...???
..........
```

Komentar: prvi primer ustreza dodatnim omejitvam za 30 %, drugi primer omejitvam za 80 %, tretji primer pa je splošen.

4. Dolge skladbe

Razvijamo aplikacijo za predvajanje glasbe z inovativno idejo; uporabili bomo namreč „tok simfonije“, kar pomeni, da se bodo skladbe ena za drugo zlile v eno skupno skladbo, brez očitnih meja med posamičnimi sestavnimi skladbami. Ko uporabnik pritisne tipko Igraj, se bo njegov seznam predvajanja pretvoril v to eno skupno skladbo in začel igrati.

Da prihranimo pri procesorski moči, pa aplikacija ne bo naenkrat pretvorila celotnega seznama, temveč bo združevala skladbo po skladbo. Pri tem pa se pojavi težava: če uporabnik preskoči naprej (ali nazaj) v času predvajanja, moramo hitro izračunati, katera skladba naj bi se takrat predvajala, da jo lahko pretvorimo.

Napiši program, ki bo sprejel podatke o dolžinah skladb in odgovarjal na vprašanja tipa „Katera skladba se predvaja ob času t_j?“

Vhodni podatki. V prvi vrstici se nahajata števili n in q. V i-ti od naslednjih n vrstic je dano število ℓ_i, ki predstavlja dolžino i-te skladbe (te so na vhodu urejene tako, kakor so urejene v seznamu predvajanja). Sledi q vrstic, od katerih j-ta vsebuje število t_j, ki zaznamuje čas od začetka predvajanja, za katerega moraš izračunati, katera skladba se takrat vrti. Vsi časi so podani v sekundah.

Omejitve: $1 \le n \le 10^5$; $1 \le q \le 10^5$; za vsak $i = 1, 2, \ldots, n$ velja $1 \le \ell_i \le 10^9$; za vsak $j = 1, 2, \ldots, q$ velja $0 \le t_j \le \ell_1 + \ell_2 + \ldots + \ell_n$. V 20 % testnih primerov bo dodatno veljalo $q = 1$.

Izhodni podatki. Za vsako od q vprašanj izpiši zaporedno številko skladbe, ki se vrti ob času t_j. Odgovore piši vsakega v svojo vrstico. Skladbe so oštevilčene od 1 do n, kakor so podane na vhodu. Če je čas ravno na meji med dvema skladbama, izpiši tisto, ki se je ravno nehala predvajati.

Primer vhoda:

```
4 4
200
300
100
400
34
600
561
601
```

Pripadajoči izhod:

```
1
3
3
4
```

NALOGE S CERC 2023

Društvo ACM Slovenija je letos sodelovalo tudi pri organizaciji srednjeevropskega študentskega tekmovanja v računalništvu (Central European Regional Contest — CERC), ki je potekalo 9. in 10. decembra 2023 na Fakulteti za računalništvo in informatiko v Ljubljani. Uradna besedila nalog in rešitev (v angleščini) so objavljena na spletni strani tekmovanja, `https://cerc.acm.si/`, v pričujočem biltenu pa objavljamo besedila nalog in rešitev v slovenščini.

Preden si ogledamo naloge, še nekaj opomb o načinu tekmovanja in ocenjevanja na CERC. Tekmovanje poteka podobno kot na slovenskih študentskih tekmovanjih v programiranju (UPM), le da v samo enem kolu: tekmujejo ekipe s po tremi tekmovalci, vsaka ekipa ima en računalnik, svoje rešitve pa oddajajo na ocenjevalni strežnik, ki jih sproti testira in ocenjuje. Tekmovanje obsega en tekmovalni dan (letos je bil to 10. december), 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: E — enaki urniki; B — podajanje žoge; H — kadrovska služba; G — gremo na Luno; J — pomešani skladi; K — ključi; D — sušenje perila; C — torte; I — interaktivna rekonstrukcija; L — najmanjše poti; A — prisotnost; F — filogenetika.

A. Prisotnost

(*Omejitev časa: 8 s. Omejitev pomnilnika: 128 MB.*)

Neki ambiciozen študent se je vpisal k praktično vsem predmetom. Žal pa ti predmeti zahtevajo obvezno prisotnost. Odločil se je, da bo večkrat na dan obiskal kampus univerze, kjer potekajo predavanja. Šel bo na vsako predavanje, ki bo tisti čas v teku, se podpisal na seznam prisotnih in nemudoma odšel zaradi drugih obveznosti. Kasneje istega dne se bo vrnil in ponovil ta postopek, se podpisal na sezname prisotnih pri takratnih predavanjih in tako naprej, dokler ne bo njegovo ime na seznamih prisotnih za vsa predavanja.

Kot da to ne bi bilo dovolj nadležno, se študent sooča s še eno težavo. Urnik predavanj se kar naprej spreminja. Kakšno predavanje dodajo, kakšno pobrišejo. Študent mora zato ves čas prilagajati svoj urnik obiskov univerze, da lahko podpiše sezname prisotnih pri vseh predavanjih.

Napiši program, ki bo začel s praznim urnikom predavanj in prebral zaporedje sprememb (vsaka sprememba je bodisi dodajanje bodisi brisanje enega predavanja). Za vsako spremembo naj izpiše najmanjše število obiskov, ki jih študent potrebuje, da lahko podpiše sezname prisotnih pri vseh predavanjih, ki so takrat (po tej spremembi) na urniku.

Vhodni podatki. V prvi vrstici je število sprememb n; sledi n vrstic, ki opisujejo vsaka po eno spremembo. Dodajanje predavanja je opisano s parom števil a_i in b_i (ločenih s presledkom), ki predstavljata predavanje od vključno časa a_i do vključno časa b_i. Predavanja so oštevilčena od 1 naprej v takem vrstnem redu, v kakršnem so bila dodana na urnik. Brisanje predavanja je predstavljeno z negativnim številom x_i, ki pomeni brisanje predavanja s številko $-x_i$.

Omejitve vhodnih podatkov: $1 \leq n \leq 300\,000$; za vsako dodajanje: $0 \leq a_i \leq b_i \leq 10^9$; vsako brisanje x_i bo veljavno — predavanje x_i bo takrat res prisotno na urniku. Bodi pozoren tudi na omejitev pomnilnika (128 MB).

Izhodni podatki. Za vsako spremembo izpiši vrstico, v kateri bo najmanjše potrebno število obiskov pri trenutnem naboru predavanj (po tisti spremembi).

Primer vhoda:

```
12
2 2
17 26
-2
12 21
0 0
19 21
16 22
14 20
15 19
13 14
-4
13 17
```

Pripadajoči izhod:

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

Komentar: prvo dodano predavanje je $[2, 2]$, ki dobi številko 1. Naslednje dodano predavanje je $[17, 26]$ s številko 2. Takoj zatem je odstranjeno, kar kaže vrstica `-2` v vhodnih podatkih. Nato se doda predavanje $[12, 21]$, ki dobi številko 3, in tako naprej.

B. Podajanje žoge

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

Skupina učencev je pravkar končala z uro matematike in se odpravila ven na telovadbo. Učitelj jim je naročil, naj se razporedijo v krog. Po večminutnem mrzličnem premikanju po igrišču so se končno uspeli razporediti tako, da tvorijo oglišča *strogo konveksnega mnogokotnika*. Morda res ne stojijo na krožnici, vendar je učitelj vesel, da imajo vsaj nekakšno strukturo.

V tej skupini n učencev je sodo število dečkov in tudi sodo število deklet. Vadili bodo podajanje žoge v parih, zato jih mora učitelj razporediti v pare. Pri tem bodo dečki vedno v paru z dečki, dekleta pa z dekleti.

Šolska uprava se je odločila nasloviti upad telesnih zmogljivosti svojih učencev. Zato so vpeljali mero kakovosti vadbe podajanja žoge, ki je definirana kot skupna razdalja, ki jo žoge prepotujejo, če si vsak par otrok enkrat poda žogo. Pomagaj učitelju razdeliti otroke v pare tako, da bo ta mera največja možna.

Vhodni podatki. V prvi vrstici je število otrok n. V drugi vrstici je niz S, sestavljen iz n znakov, ki podajajo spol otrok v takem vrstnem redu, v kakršnem stojijo vzdolž mnogokotnika; pri tem znak `B` pomeni dečka, `G` pa dekle. Sledi n vrstic s koordinatami otrok: i-ta od teh vrstic vsebuje dve s presledkom ločeni celi števili, x_i in y_i, ki povesta koordinati i-tega otroka (v enakem vrstnem redu kot v nizu S).

Omejitve vhodnih podatkov: $2 \leq n \leq 50$. Število dečkov bo sodo, število deklet prav tako; mogoče pa je, da bo eno od teh dveh števil 0. Absolutna vrednost koordinat ne bo presegla 10 000.

Izhodni podatki. Izpiši največjo možno razdaljo podajanja žoge, ki jo je mogoče doseči, če otroke primerno razporedimo v pare. Rešitev bo sprejeta kot pravilna, če bo njena relativna ali absolutna napaka v primerjavi z uradno rešitvijo kvečjemu 10^{-6}.

Trije primeri vhodov in pripadajočih izhodov:

Vhod:	Izhod:
4 BGBG 0 0 0 1 1 1 1 0	2.828427125

Vhod:	Izhod:
4 GGBB 0 0 0 1 1 1 1 0	2

Vhod:	Izhod:
12 GBGBBGBBBBGB 0 -15 6 -14 19 -5 17 7 11 12 1 15 -9 13 -15 10 -17 8 -19 4 -16 -9 -13 -11	186.529031603

C. Torte

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

Bližnja pekarna pripravlja svoj poslovni načrt za naslednjih nekaj mesecev. Peki imajo C receptov za torte, ki potrebujejo vsak svoj nabor sestavin in orodij. Pri izdelavi torte se vse sestavine zanjo potrošijo, orodja pa ne in jih je mogoče uporabiti ponovno pri drugih receptih. Trenutno nima pekarna nobenih sestavin in nobenega orodja — vse jim je bodisi zasegla davkarija bodisi je bilo uničeno v nedavnih poplavah.

Sin glavnega peka je nagovoril célo pekarno, da bodo spekli po največ eno torto po vsakem receptu. Nekateri ljudje na internetu so namreč bojda pripravljeni plačati več, da so potem lahko sami edini lastniki svoje edinstvene „najbolj frišne torte“ (NFT). Pravzaprav je fant že ocenil, za koliko denarja lahko prodajo torto posamezne vrste. Zdaj se peki sprašujejo, katere torte naj spečejo, da bodo imeli čim več dobička. **Napiši program**, ki pri danih cenah vseh sestavin, orodja in tort ugotovi, koliko denarja lahko pekarna največ zasluži.

Vhodni podatki. V prvi vrstici so tri cela števila: G (število vrst sestavin), C (število receptov za torte) in T (število vrst orodij). V drugi vrstici je C s presledki ločenih celih števil, $c_1\ c_2\ \ldots\ c_C$, ki predstavljajo cene tort. V tretji vrstici je G s presledki ločenih celih števil, $g_1\ g_2\ \ldots\ g_G$, ki predstavljajo cene sestavin. V četrti vrstici je T s presledki ločenih celih števil, $t_1\ t_2\ \ldots\ t_T$, ki predstavljajo cene orodij.

Sledi C vrstic; v k-ti od njih je G celih števil, ločenih s presledki; j-to od teh števil, a_{kj}, je količina izdelka j v receptu za torto k.

Nazadnje sledi še C vrstic; k-ta od njih se začne s celim številom n_k, ki pove, koliko orodij je potrebnih za izdelavo torte k. Sledi še n_k celih števil $b_{k\ell}$, ločenih

s presledki; to so številke orodij, potrebnih za izdelavo torte k (vsa ta orodja so različna).

Omejitve vhodnih podatkov:

- $1 \le C \le 200$; $1 \le G \le 200$; $1 \le T \le 200$;
- $0 \le c_k \le 10^9$ (za vse k); $0 \le t_i \le 10^9$ (za vse i);
- $0 \le g_i \le 10^8$ (za vse i); $0 \le a_{kj} \le 10^8$ (za vse k in j);
- $1 \le b_{k\ell} \le T$ (za vse k in ℓ).

Izhodni podatki. Izpiši eno samo celo število, namreč največji dobiček, ki ga pekarna lahko ustvari.

Primer vhoda:

```
5 3 4
14 18 21
1 2 3 1 2
5 6 3 10
0 0 1 2 0
1 2 0 1 2
5 2 1 0 0
2 1 2
2 2 3
2 3 4
```

Pripadajoči izhod:

```
3
```

Opomba: za največji dobiček pri tem primeru morajo peki speči torti 1 in 2, ne pa tudi torte 3.

D. Sušenje perila

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

Bober Harry vodi hotel in naslednjih q tednov, do konca turistične sezone, bo moral vsako nedeljo zvečer oprati posteljnino. Ob koncu j-tega tedna bo imel n sveže opranih rjuh, ki jih bo hotel obesiti na dve vzporedni vrvi dolžine ℓ_j, da se posušijo. Rjuhe lahko obesi eno zraven druge, ne smejo pa se prekrivati. Pri tem je i-ta rjuha (za vse $i = 1, 2, \ldots, n$) široka d_i enot in zelo dolga, zato jo bo pri obešanju na vrv vedno obrnil tako, da bo zasedla d_i enot vrvi. Rjuhe se sušijo različno hitro, pri čemer pa čas sušenja ni nič povezan z njihovo velikostjo, saj je odvisen le od materiala, iz katerega so izdelane. Tako se i-ta rjuha posuši v p_i minutah; če pa jo obesimo čez obe vrvi, se posuši hitreje, v samo h_i minutah, vendar pa s tem tudi zasede prostor na obeh vrveh. Da se rjuhe ne bodo usmradile, jih mora bober Harry vse začeti sušiti takoj, čim pridejo iz pralnega stroja (vse rjuhe mora torej obesiti istočasno).

Bober Harry bi šel rad ob nedeljah spat čim bolj zgodaj, zato te prosi, da mu pomagaš za vsak teden j določiti najmanjši možni čas sušenja ali pa mu poveš, da se rjuh tisti teden sploh ne bo dalo posušiti.

Vhodni podatki. V prvi vrstici sta celi števili n (število rjuh) in q (število tednov do konca turistične sezone). Sledi n vrstic, ki opisujejo rjuhe; i-ta od teh vrstic vsebuje cela števila d_i, h_i in p_i, ki za i-to rjuho povedo njeno širino, hitrejši čas sušenja (čez obe vrvi) in počasnejši čas sušenja (na eni vrvi). Sledi še q vrstic, od katerih j-ta vsebuje število ℓ_j, torej dolžino vrvi za obešanje perila v j-tem tednu.

Omejitve vhodnih podatkov: $1 \le n \le 3 \cdot 10^4$; $1 \le q \le 3 \cdot 10^5$; za vse $i = 1, \ldots, n$ velja $1 \le d_i \le 3 \cdot 10^5$ in $1 \le h_i \le p_i \le 10^9$; za vse $j = 1, \ldots, q$ velja $1 \le \ell_j \le 3 \cdot 10^5$.

Izhodni podatki. Izpiši q vrstic, v j-ti od njih pa naj bo le eno celo število, namreč minimalni potrebni čas sušenja za j-ti teden. Če tisti teden rjuh sploh ni mogoče posušiti, izpiši „`-1`“ (brez narekovajev). (Če je pri kakšnem tednu minimalni potrebni čas sušenja večji od števila minut v tednu, moraš še vseeno izpisati ta čas sušenja in ne „`-1`“.)

Primer vhoda:

```
3 3
1 2 2
1 1 4
2 3 100
3
1
4
```

Pripadajoči izhod:

```
4
-1
3
```

E. Enaki urniki

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

Delaš za ponudnika storitev, ki uporabnikom nudi reševanje programerskih nalog z visoko stopnjo dostopnosti. Kot dobro organizirana ekipa imate urnik pripravljenosti, ki določa, kdo je katero uro zadolžen za zagotavljanje storitve. Sodelavec ti je poslal nov urnik, ti pa bi rad preveril, ali je na njem vsakdo v pripravljenosti enako dolgo časa kot prej, oz. izpisal morebitne razlike.

Urnik pripravljenosti je podan kot zaporedje vrstic oblike s_i e_i t_i, kjer s_i in e_i predstavljata začetek in konec obdobja, ko je v pripravljenosti delavec t_i, merjeno v urah od nekega začetnega trenutka. Primer takšnega urnika je:

```
0 7 jan
7 14 tomaz
14 20 jure
20 24 jan
24 25 tomaz
25 26 jure
```

Pri gornjem urniku vidimo, da je `jan` v pripravljenosti sedem ur (to so ure 0, 1, 2, 3, 4, 5 in 6), `tomaz` naslednjih sedem ur ipd. Vsega skupaj je `jan` v pripravljenosti enajst ur, `tomaz` osem in `jure` sedem.

Vhodni podatki. Na vhodu dobiš dva urnika, ločena z vodoravno vrstico `------`. Vsak urnik je sestavljen iz ene ali več vrstic oblike s_i e_i t_i, kjer celi števili s_i in e_i povesta, da je delavec t_i v pripravljenosti v urah od s_i do zadnje ure pred e_i. Za drugim urnikom je še vrstica z znaki `======`, ki je tudi zadnja vrstica vhodnih podatkov.

Omejitve vhodnih podatkov: za vsak urnik velja $s_1 = 0$, $s_i < e_i \leq 1000$, $s_{i+1} = e_i$. Vsako ime t_i je dolgo vsaj 3 in kvečjemu 20 znakov, vsi znaki pa so male črke angleške abecede.

Izhodni podatki. Izpiši razlike med obema urnikoma kot zaporedje vrstic oblike $t_i \ \pm d_i$, kjer je d_i razlika med drugim in prvim urnikom za delavca t_i. Vrstice morajo biti urejene po abecednem vrstnem redu imen. Tisti delavci, pri katerih med urnikoma ni razlik, morajo biti izpuščeni, drugače pa razliko vedno izpiši s

predznakom `+` ali `-`. Če med urnikoma ni tovrstnih razlik, izpiši „`No differences found.`“ (brez narekovajev).

Trije primeri vhodov in pripadajočih izhodov:

Vhod:
```
0 7 jan
7 14 tomaz
14 20 jure
20 24 jan
24 25 tomaz
25 26 jure
------
0 9 tomaz
9 20 jan
20 26 jure
======
```

Izhod:
```
jure -1
tomaz +1
```

Vhod:
```
0 7 nino
7 14 bgs
14 21 ines
------
0 7 ines
7 14 nino
14 21 bgs
======
```

Izhod:
```
No differences found.
```

Vhod:
```
0 3 vid
3 6 maks
6 9 janez
------
0 1 vid
1 2 vid
2 3 vid
3 4 maks
4 5 maks
5 6 maks
6 7 janez
7 8 janez
======
```

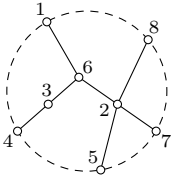
Izhod:
```
janez -1
```

F. Filogenetika

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

Mlada biologinja se je pri študiju evolucijske zgodovine srečala s filogenetskimi drevesi. Takšno drevo prikazuje evolucijska razmerja med različnimi biološkimi vrstami. Predstavljeno je z vložitvijo v ravnino, pri čemer so listi zaradi večje nazornosti razporejeni v krogu. Opravka imamo z drevesom brez korena, v katerem so listi vsa vozlišča s stopnjo 1. Vsa vozlišča drevesa so tudi pobarvana, da je laže ločiti različne vrste.

Naša biologinja uporablja program za vizualizacijo grafov, ki potrebuje nekaj pomoči, da pride do želenega razporeda. Zato se je odločila dodati povezave med listi, ki so si pri vložitvi v ravnino sosedje. Drevo ima vsaj tri liste in povezala jih bo v cikel. Naslednja slika kaže primer takšnega (nepobarvanega) drevesa, v katerem so dodatne povezave med sosednjimi listi narisane s črtkanimi črtami:



Zdaj ko je vizualizacija pripravljena, biologinjo zanima, na koliko načinov je mogoče pobarvati ta graf s k barvami. Vsaki dve sosednji vozlišči morata biti različnih barv, da se jih bo dalo laže prepoznati. **Napiši program**, ki prebere opis njenega grafa in izračuna število barvanj.

Vhodni podatki. V prvi vrstici so cela števila n (število vozlišč), m (število povezav) in k (število barv), ločena s presledkom. Sledi m vrstic, ki podajajo povezave grafa; v i-ti od njih sta s presledkom ločeni celi števili a_i in b_i, ki pomenita številki krajišč i-te povezave. (Vozlišča grafa so oštevilčena z naravnimi števili od 1 do n.)

Zagotovljeno je, da je graf nastal iz ravninske vložitve drevesa (acikličnega povezanega neusmerjenega grafa) tako, da so bili listi drevesa povezani še v cikel. Graf ne

bo vseboval zank (povezav, ki se začnejo in končajo v istem vozlišču) ali vzporednih povezav (torej več povezav med istim parom vozlišč).

Omejitve vhodnih podatkov: $4 \leq n \leq 10^5$; $1 \leq k \leq 10^5$.

Izhodni podatki. Izpiši ostanek po deljenju števila barvanj z 1 000 000 007.

Primer vhoda:	Pripadajoči izhod:
8 12 3 2 5 3 6 2 6 5 4 4 1 1 6 7 5 2 7 3 4 2 8 7 8 1 8	24

G. Gremo na Luno

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

Alica in Bob se igrata v pesku pred svojim dvorcem. Nekje sta narisala krog, ki predstavlja Luno, vsak od njiju pa si je izbral tudi neko točko in se tja postavil (lahko znotraj Lune, lahko zunaj, lahko tudi na njenem robu). Cilj igre je, da eden od igralcev čim hitreje steče do drugega, med tekom pa se tudi dotakne Lune.

Pri danem položaju Lune, Alice in Boba izračunaj dolžino najkrajše poti, ki se začne pri enem od igralcev, se (lahko tudi večkrat) dotakne roba Lune (ali ga seka) in/ali njene notranjosti ter se konča pri drugem igralcu.[5]

Vhodni podatki. V prvi vrstici je T, število testnih primerov. Sledi T vrstic, ki opisujejo vsaka po en testni primer; v vsaki je sedem celih števil, ločenih s presledki: $x_A, y_A, x_B, y_B, x_C, y_C$ in r, ki podajajo koordinate Alice $A = (x_A, y_A)$, Boba $B = (x_B, y_B)$, središče kroga $C = (x_C, y_C)$ in njegov polmer r.

Omejitve vhodnih podatkov: $1 \leq T \leq 10^3$; vse koordinate so po absolutni vrednosti $\leq 10^3$; $0 \leq r \leq 10^3$.

Izhodni podatki. Za vsak testni primer izpiši po eno vrstico, vanjo pa eno samo decimalno število, ki predstavlja dolžino najkrajše take poti od A do B, ki vsebuje tudi neko točko v notranjosti ali na robu kroga s središčem C in polmerom r. Rešitev bo sprejeta kot pravilna, če bo njena relativna ali absolutna napaka v primerjavi z uradno rešitvijo kvečjemu 10^{-6}.

[5] Zanimivo težjo različico naloge dobimo, če zahtevamo, da se mora pot dotakniti roba Lune (torej pot, ki se dotika zgolj notranjosti Lune, ne velja).

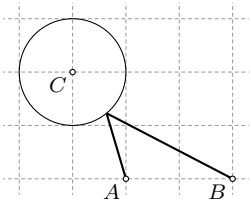
Primer vhoda:

```
2
0 0 2 0 -1 2 1
5 0 3 0 2 0 2
```

Pripadajoči izhod:

```
3.9451754612261913
2
```

Prvi testni primer kaže spodnja slika:



H. Kadrovska služba

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

Delaš za ECorp in vaš kadrovski oddelek seli podatke o zaposlenih z lokalnega sistema, ki ga je nudil Hooli, v oblačno storitev novega zagonskega podjetja Pied Piper. Žal novi sistem še nima vseh zmogljivosti starega, zato potrebujejo tvojo pomoč pri shranjevanju in prikazu celotne upravljalske strukture podjetja. Sistem je varčen in pazljiv pri porabi virov, zato si lahko privoščijo povečati količino shranjenih podatkov le za dva kilobita.

Vhodni podatki. V prvi vrstici je eden od nizov `ENCODE` ali `DECODE`, ki pove, ali naj tvoj program podatke kodira ali dekodira.

Kodiranje: preostale vrstice vhoda opisujejo šefe in njihove neposredno podrejene. Vsaka vrstica se začne z imenom šefa, ki mu sledi dvopičje in nato seznam imen njegovih neposredno podrejenih, urejen od njemu najljubšega do njemu najmanj dragega. Imena v seznamu so ločena s presledkom, pa tudi za dvopičjem je presledek. Noben šef ni naveden pred svojim lastnim šefom (če ga ima).

Dekodiranje: preostale vrstice vsebujejo tisto, kar je tvoj program izpisal pri kodiranju, namreč seznam imen vseh zaposlenih (v nekem poljubnem vrstnem redu), po eno na vrstico, na koncu pa je še vrstica z binarnim nizom B.

Omejitve vhodnih podatkov: število vseh zaposlenih pri ECorp je vsaj 2 in kvečjemu 600. Dolžina niza B je ≤ 2048.

Imena zaposlenih so dolga kvečjemu 10 znakov, sestavljajo pa jih le male in/ali velike črke angleške abecede.

Natanko en zaposleni nima svojega šefa (namreč direktor podjetja), nihče pa nima več kot enega šefa.

Izhodni podatki. — Kodiranje: v prvih n vrsticah (če je n število vseh zaposlenih) moraš izpisati imena vseh zaposlenih, vsako v svoji vrstici, lahko pa so v poljubnem vrstnem redu (tako je zahtevalo vodstvo podjetja). Nato izpiši še vrstico s svojm kodirnim nizom B, ki mora biti sestavljen iz samih ničel in enic, dolg pa mora biti največ 2048 znakov.

Dekodiranje: izpiši prvotno strukturo šefov in podrejenih v enakem formatu, v kakršnem je bila prvotno podana (kot vhodni podatki pri kodiranju). Vrstni red, v katerem so opisani šefi, je lahko tudi drugačen, vendar mora še vedno vsakdo priti za svojim šefom (če ga ima). Vrstni red podrejenih pri posameznem šefu pa mora ostati nespremenjen (od najbolj do najmanj priljubljenega).

Primer vhoda in izhoda za kodiranje in dekodiranje:

Vhod:	Izhod:	Vhod:	Izhod:

`ENCODE`
`Janez: Josip Zofia`
`Zofia: Karolina`
	`Josip`
	`Karolina`
	`Janez`
	`Zofia`
	`00101100`
		`DECODE`
		`Josip`
		`Karolina`
		`Janez`
		`Zofia`
		`00101100`
			`Janez: Josip Zofia`
			`Zofia: Karolina`

Komentar: kodiranje v tem primeru uporablja po dva zaporedna znaka za vsakega zaposlenega (v vrstnem redu, v kakršnem so navedeni v seznamu). Znaka `11` označujeta direktorja (`Janeza`). Znaka `00` pomenita človeka na drugem nivoju hierarhije. Vrstni red teh ljudi v seznamu direktorjevih neposredno podrejenih je tak kot v seznamu zaposlenih v kodirani predstavitvi; na srečo sta v tem primeru taka človeka samo dva (`Josip` in `Zofia`). Znaka `10` označujeta Zofijinega podrejenega, znaka `01` pa bi označevala Josipove podrejene, če bi jih kaj imel.

I. Interaktivna rekonstrukcija

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

To je interaktivna naloga, pri kateri se bo tvoj program sporazumeval z ocenjevalnim sistemom prek standardnega vhoda in izhoda. Tvoja naloga je rekonstruirati označeno drevo z n vozlišči in $n - 1$ povezavami. Vozlišča so označena s števili od 1 do n.

Tvoj program sme izvesti nekaj poizvedb naslednje oblike: izpiše naj niz n znakov, sestavljen le iz ničel in enic, po en znak za vsako vozlišče. Ocenjevalni sistem bo odgovoril z zaporedjem n celih števil (ločenih s presledki), od katerih i-to predstavlja vsoto vrednosti (ničel in enic iz poizvedovalnega niza) po vseh sosedih i-tega vozlišča; če je torej na primer vozlišče j sosed vozlišča i, potem prispeva j-ta števka poizvedovalnega niza k vsoti za i-to število v odgovoru ocenjevalnega sistema (glej tudi primer spodaj).

Vhodni in izhodni podatki. Tvoj program naj najprej prebere vrstico s celim številom n, ki pomeni število vozlišč v drevesu. Nato ima tvoj program dve možnosti:

- poizvedba: izpiši „`QUERY`“ (brez narekovajev), presledek in zaporedje n ničel in enic;
- odgovor: izpiši „`ANSWER`“ (brez narekovajev), konec vrstice in nato še $n - 1$ vrstic, v vsaki od teh pa naj bosta dve s presledkom ločeni celi števili a in b, ki povesta, da obstaja povezava med vozliščema a in b.

Če tvoj program izpiše poizvedbo, bo na standardni vhod nato od ocenjevalnega sistema dobil vrstico z n celimi števili, ločenimi s presledki. Če tvoj program izpiše odgovor, bo ocenjevalni sistem preveril, ali je izpisano drevo pravilno.

Če je v tvojih poizvedbah kakšna napaka, bodisi zaradi napačnega formatiranja ali zaradi prevelikega števila poizvedb, bo ocenjevalni sistem namesto odgovora izpisal „`ERROR`“ (brez narekovajev).

Pozor: pazi na to, da tvoj program po vsaki izpisani poizvedbi ali odgovoru splakne (*flush*) standardni izhod in da se po izpisu dogovora pravilno neha izvajati. O tem, ali bo tvoj program znal obravnavati tudi sporočila ocenjevalnega sistema o napakah

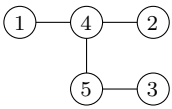
(ERROR) ali ne, se odloči sam; namen teh sporočil je, da se lahko tvoj program v primeru napake preneha izvajati na regularen način in dobi oceno WA (napačen odgovor) namesto TLE (prekoračena časovna omejitev).

Omejitve vhodnih podatkov: $2 \le n \le 3 \cdot 10^4$. Dovoljenih je največ $2 \uparrow\uparrow 3 = 2^{2^2} = 16$ poizvedb; odgovor na koncu ne šteje v to omejitev.

Primer:

Tvoj program izpiše:	Ocenjevalni sistem izpiše:
	`5`
`QUERY 10001`	`0 0 1 2 0`
`QUERY 00010`	`1 1 0 0 1`
`QUERY 10000`	`0 0 0 1 0`
`ANSWER`	
`1 4`	
`4 2`	
`5 4`	
`3 5`	

Komentar. Drevo pri tem primeru je takšno, kot ga prikazuje slika na desni. S tremi poizvedbami iz primera ga je mogoče rekonstruirati enolično.



J. Pomešani skladi

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

Dana je množica n kart, ki so oštevilčene od 1 do n in razdeljene na k skladov $S_1, S_2, \ldots, S_k$. Vsak sklad ima omejeno kapaciteto: i-ti sklad, torej S_i, lahko vsebuje največ C_i kart. Sklade lahko spreminjamo le tako, da poberemo karto z vrha enega sklada in jo odložimo na vrh nekega drugega sklada (če seveda s tem ne prekoračimo kapacitete tega drugega sklada).

Z zaporedjem takšnih potez bi radi prerazporedili karte tako, da bo prvih nekaj (nič ali več) skladov z najmanjšimi indeksi popolnoma zapolnjenih (do svoje kapacitete), naslednji sklad ne bo popolnoma zapolnjen (lahko je celo prazen), vsi preostali skladi pa bodo popolnoma prazni; in poleg tega mora veljati, da če zložimo skupaj vse sklade od S_1 na dnu do S_k na vrhu, bodo karte potem urejene naraščajoče po številkah, od 1 na dnu do n na vrhu. Zagotovljeno je, da velja

$$n \le \left(\sum_{i=1}^{k} C_i \right) - \max_{1 \le i \le k} C_i. \qquad (\star)$$

Primer: recimo, da imamo $n = 6$ kart na $k = 3$ skladih s kapacitetami $C_1 = 4$, $C_2 = C_3 = 3$ in z začetnim stanjem $S_1 = [2, 3, 0, 0]$ (od dna proti vrhu; ničla pomeni prazno mesto), $S_2 = [4, 1, 6]$ in $S_3 = [5, 0, 0]$. Potem je želeno ciljno stanje naslednje: $S_1 = [1, 2, 3, 4]$, $S_2 = [5, 6, 0]$ in $S_3 = [0, 0, 0]$.

Vhodni podatki. V prvi vrstici sta dve celi števili, n (število kart) in k (število skladov), ločeni s presledkom. Preostalih k vrstic opisuje začetno stanje skladov;

i-ta od teh vrstic opisuje sklad S_i in vsebuje $C_i + 1$ celih števil, ločenih s presledki. Prvo od teh števil je C_i (kapaciteta sklada S_i), ostala pa so številke kart na skladu S_i, od dna sklada proti vrhu. Če vsebuje sklad S_i manj kot C_i kart (lahko je celo prazen), bo zadnjih nekaj števil v vrstici enakih 0.

Omejitve vhodnih podatkov: $1 \le n \le 100$ in $3 \le k \le 100$; za vse $i = 1, 2, \ldots, k$ velja $1 \le c_i \le n$.

Izhodni podatki. Izpiši zaporedje potez, ki spravijo sklade v želeno končno stanje. Za vsako potezo izpiši vrstico z dvema celima številoma, ločenima s presledkom: najprej številko sklada, s katerega karto pobiramo, in nato številko sklada, na katerega jo odlagamo (skladi so oštevilčeni od 1 do k; ciljni sklad ne sme biti isti kot izvorni sklad). Število potez ne sme preseči 10^5. Po koncu zaporedja potez izpiši vrstico s številoma „`0 0`“ (brez narekovajev). Če je možnih več rešitev, je vseeno, katero od njih izpišeš.

Primer vhoda:

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

Komentar: to je primer, o katerem smo govorili že prej v besedilu naloge. Primer izhoda kaže zaporedje 14 potez, ki privedejo sklade v želeno končno stanje.

Eden od možnih pripadajočih izhodov:

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

K. Ključi

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

Alica in Bob živita v ogromnem dvorcu z n sobami (ena od teh pravzaprav predstavlja zunanjost), ki jih povezuje m vrat. Vsaka vrata neposredno povezujejo dve sobi ali sobo in zunanjost ter imajo en sam ključ, ki odpira samo ta vrata. Vsaka vrata se, ko greš skoznje, za tabo takoj zaprejo in avtomatsko zaklenejo, zato za prehod skozi vrata vedno potrebuješ ključ. Stavba je tako velika, da uporabljata Alica in Bob eno samo sobo — spalnico. Vse ostale sobe so namenjene zgolj temu, da je hiša videti večja in da so sosedje bolj nevoščljivi.

Ta nenavadni pristop h gradnji hiše zdaj povzroča Alici in Bobu nekaj težav.

Bob se odpravlja na dvotedensko službeno pot. Vendar pa bo čez en teden odpotovala tudi Alica, in to za cel mesec. Ko bo odhajala, bo Alica potrebovala primerne ključe, da bo sploh prišla iz hiše. Toda tudi Bob potrebuje ključe, da bo lahko prišel nazaj v hišo, saj takrat Alice ne bo doma, da bi ga spustila noter. Zdaj poskušata Alica in Bob ugotoviti, kako naj si razdelita ključe vrat, da bo lahko Alica prišla iz sobe 0 (njune spalnice) v sobo 1 (zunanjost), Bob pa bo en teden kasneje lahko prišel iz sobe 1 (zunanjosti) v 0 (spalnico).

Na srečo se je Alica domislila, da lahko na poti iz hiše odloži nekaj ključev, ki jih bo Bob ob vrnitvi pobral. Tako lahko oba prideta skozi ista vrata. Seveda pa ključev ne sme odložiti v sobi 1 (zunanjosti), ker bi jih tam lahko našli sosedje in vdrli v hišo.

Ali lahko Alici in Bobu pomagaš razdeliti ključe in načrtovati njune premike po hiši?

Naloga. Dobil boš opis Aličinega in Bobovega dvorca: m vrat med n sobami, ki so oštevilčene od 0 do $n-1$, pri čemer 1 predstavlja zunanjost, 0 pa spalnico. Oštevilčena so tudi vrata in ključi, in sicer 0 do $m-1$; vrata i odpira ključ številka i.

Najprej izpiši dve vrstici, v katerih so s presledki ločene številke ključev za Alico (v prvi vrstici) in za Boba (v drugi). Nič ni narobe, če ne uporabita vseh ključev, ne smeta pa oba imeti vsak svoje kopije istega ključa (prav tako tudi ne sme eden od njiju imeti po več kopij istega ključa).

Nato izpiši navodila, ki jih bosta Alica in Bob izvedla. Najprej izpiši Aličine premike iz sobe 0 v sobo 1 kot zaporedje ukazov naslednjih dveh vrst:

- „`MOVE` x“ za premik v sobo x (ob predpostavki, da obstajajo vrata med sobo x in sobo, v kateri je Alica trenutno, in da ima ona ključ teh vrat);
- „`DROP` k_1 k_2 $\ldots$“, da Alica odloži ključe $k_1, k_2, \ldots$ (ki so podani kot zaporedje celih števil, ločenih s presledki) v sobi, kjer se trenutno nahaja. To pomeni, da Alica odtlej teh ključev nima več.

Ko so Aličini premiki končani, izpiši v samostojno vrstico niz „`DONE`“. Svoje gibanje mora Alica končati v sobi 1, ni pa nič narobe, če gre med izvajanjem tvojega seznama navodil po večkrat skozi isto sobo, tudi sobo 0 ali sobo 1.

Nato izpiši Bobove premike iz sobe 1 v sobo 0 kot zaporedje ukazov naslednjih dveh vrst:

- „`MOVE` x“ za premik v sobo x (ob predpostavki, da obstajajo vrata med sobo x in sobo, v kateri je Bob trenutno, in da ima on ključ teh vrat);
- „`GRAB`“, da Bob pobere ključe v sobi, v kateri se trenutno nahaja. Bob vedno pobere vse ključe, ki jih je Alica pustila v sobi; če ni v sobi nobenega ključa, pač ne pobere ničesar.

Ko so Bobovi premiki končani, izpiši v samostojno vrstico niz „`DONE`“. Svoje gibanje mora Bob končati v sobi 0, ni pa nič narobe, če gre med izvajanjem tvojega seznama navodil po večkrat skozi isto sobo, tudi sobo 0 ali sobo 1.

Opomba: dovoljeno, čeprav nekoristno, je tudi odložiti (z ukazom `DROP`) prazen seznam ključev ali pa pobrati (z ukazom `GRAB`) ključe v sobi, kjer ni nobenega ključa (ali pa v sobi 1, torej zunaj).

Vhodni podatki. V prvi vrstici sta celi števili n (število sob, vključno z zunanjostjo) in m (število vrat). Sledi m vrstic, ki opisujejo vrata; i-ta od teh vrstic (če jih štejemo od 0 naprej) vsebuje celi števili a_i in b_i, ki povesta, da so med sobama a_i in b_i vrata, ki jih odpira ključ i.

Omejitve vhodnih podatkov: $2 \leq n \leq 10^5$; $2 \leq m \leq 10^5$. Iz vsake sobe se da priti do vsake druge sobe (če imaš vse potrebne ključe). Vsak par sob je neposredno

povezan z največ enimi vrati. Nobena soba ni neposredno povezana sama s sabo. Tvoj program sme izpisati največ $4 \cdot 10^5$ ukazov.

Izhodni podatki. Najprej izpiši dve vrstici, ki opisujeta razdelitev ključev. Nato izpiši vsa navodila za Alico in Boba, kot je določeno zgoraj v opisu naloge. Če rešitev ne obstaja, izpiši „`No solution`“ (brez narekovajev). Če obstaja več veljavnih rešitev, je vseeno, katero od njih izpišeš.

Primer vhoda:

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

Pripadajoči izhod:

```
0 1 2
3 4
MOVE 1
MOVE 2
MOVE 3
DROP 0
MOVE 2
MOVE 1
DONE
MOVE 4
MOVE 3
GRAB
MOVE 4
MOVE 1
MOVE 0
DONE
```

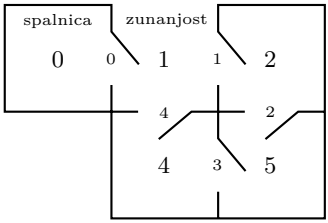
Še en primer:

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

Pripadajoči izhod:

```
No solution
```

Komentar. Prvi primer predstavlja tloris stavbe, kot ga kaže slika na desni (pri čemer majhne številke med sobami pomenijo številko ključa, ki odpira tista vrata). Alica vzame ključe 0, 1 in 2, Bob pa ključa 3 in 4. Alica gre iz sobe 0 v 1, od tam v 2 in od tam v 3, kjer nato odloži ključ 0. Skozi sobo 2 se vrne v 1. Bob začne v sobi 1, gre od tam v 4 in nato v 3, kjer pobere ključ 0. Nato se skozi 4 vrne v 1 in s ključem 0, ki ga je malo prej pobral, odpre vrata v sobo 0.



Pri drugem primeru ni nobenega načina, da bi Alica in Bob dosegla vsak svoj cilj. Še enkrat opozorimo, da Alica ključev ne more puščati v sobi 1.

L. Označene poti

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

Dan je usmerjen acikličen graf z n točkami in m povezavami. Vsaka povezava ima *oznako* (niz malih črk angleške abecede; lahko tudi prazen niz). Pojem oznak lahko zdaj takole posplošimo s povezav na poti: definirajmo *oznako poti* kot niz, ki nastane, če staknemo oznake povezav, ki tvorijo to pot (v takem vrstnem redu, v kakršnem se pojavljajo na poti). *Najmanjša pot* od začetne točke s do končne točke t je tista pot (od s do t), katere oznaka je leksikografsko najmanjša (torej ki je najzgodnejša v leksikografskem vrstnem redu) med vsemi potmi od s do t. **Napiši program**, ki za dani s izpiše najmanjše poti od s do t za vse točke t v grafu.

Vhodni podatki. V prvi vrstici so štiri cela števila, ločena s presledki: n (število točk), m (število povezav), d (dolžina niza A; več o njem v nadaljevanju) in s (številka začetne točke). Točke so oštevilčene z naravnimi števili od 1 do n.

V drugi vrstici je niz A, ki je dolg natanko d znakov; vsi ti znaki so male črke angleške abecede. Vse oznake povezav v našem grafu so podnizi niza A.

Sledi še m vrstic, ki opisujejo povezave grafa; i-ta od teh vrstic opisuje i-to povezavo in vsebuje štiri cela števila, ločena s presledki: u_i (začetno krajišče te povezave), v_i (končno krajišče te povezave), p_i in ℓ_i. Zadnji dve od teh števil povesta, da je oznaka te povezave podniz niza A, ki se začne s p_i-tim znakom niza A in je dolg ℓ_i znakov. (Za znake niza A si predstavljajmo, da so oštevilčeni z indeksi od 1 do d.)

Omejitve vhodnih podatkov:

- $1 \le s \le n \le 600$; $1 \le m \le 2000$;
- $1 \le d \le 10^6$;
- za vse $i = 1, \ldots, m$ velja $1 \le u_i \le n$, $1 \le v_i \le n$ in $u_i \ne v_i$;
- za vse $i = 1, \ldots, m$ velja $1 \le p_i$, $0 \le \ell_i$ in $p_i + \ell_i - 1 \le d$;
- graf je acikličen in nima vzporednih povezav (iz $i \ne j$ sledi $u_i \ne u_j$ in/ali $v_i \ne v_j$).

Izhodni podatki. Izpiši n vrstic, pri čemer t-ta vrstica (za $t = 1, \ldots, n$) opisuje najmanjšo pot od s do t. Če poti od s do t spoh ni, naj ta vrstica vsebuje le število `0` in ničesar drugega. Drugače pa naj se vrstica začne s številom točk na poti (vključno s točkama s in t), nato pa naj bodo po vrsti navedene te točke, ločene s presledki. Če je možnih več rešitev, je vseeno, katero od njih izpišeš.

Primer vhoda:

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

Pripadajoči izhod:

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

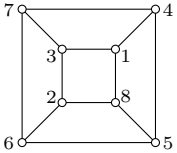
Komentar. Pri tem primeru ima povezava $3 \to 1$ oznako `ab`; povezava $1 \to 4$ ima oznako `a`; najmanjša pot od 3 do 4 je $3 \to 1 \to 4$ z oznako `aba`.

POSKUSNO TEKMOVANJE
(9. decembra 2023)

X. Izgubljena lica

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

Jamie je célo poletje intenzivno preučeval povezane ravninske grafe. Graf je *ravninski*, če obstaja na evklidski ravnini taka slika grafa, pri kateri se nobeni dve povezavi ne sekata. Jamie je v svoj zvezek narisal na stotine slik grafov in to tako, da so bile povezave *ravne*. Poskrbel je celo, da je bil kót med vsakima dvema povezama s skupnim krajiščem strogo večji od 10^{-9} radianov. Primer takšne slike grafa je:



Tom pa je, namesto da bi pomagal Jamieju, celo poletje gledal televizijske nadaljevanke. Za kazen mora zdaj digitalizirati vse slike tako, da bo opisal vsak graf v tekstovni datoteki. Medtem je šel Jamie na pošteno zaslužene počitnice v razkošne toplice, odkoder se bo nocoj vrnil domov. Tom je pripravil primerne datoteke s podatki o povezavah grafov, nato pa šel spet gledat nadaljevanke; šele zdaj se je spomnil, da mu je Jamie naročil, naj v datotekah opiše tudi vsa *lica* grafov. Če bodo podatki o licih manjkali, bo Jamie zelo jezen. Zato Tom zdaj ponuja veliko Milkino čokolado komurkoli, ki mu lahko napiše program, ki bo poiskal vsa lica ravninskega grafa z ravnimi povezavami.

Lica grafa so povezane komponente evklidske ravnine, ki jih dobimo, če jo „razrežemo“ po povezavah grafa. Graf na gornji sliki ima šest lic (pet „notranjih“ in eno „zunanjost“). Vsako lice lahko opišemo z zaporedjem točk, ki ga obdajajo. Naštejemo jih v vrstnem redu, kakršnega dobimo, če se sprehodimo po ciklu na robu lica. Najbolj notranje lice na gornji sliki lahko tako opišemo kot $3, 2, 8, 1$. Če začnemo obhod pri kakšni drugi točki ali pa gremo v nasprotno smer, lahko dobimo drugačen vrstni red istih točk. Še posebej nas zanima leksikografsko najmanjši opis; v gornjem primeru je to $1, 3, 2, 8$. Temu recimo *kanonični opis lica*.

Napiši program, ki prebere podatke o vložitvi ravninskega grafa v evklidsko ravnino in izpiše kanonični opis vseh njegovih lic.

Vhodni podatki. V prvi vrstici sta celi števili n (število točk) in m (število povezav), ločeni s presledkom. Točke grafa so oštevilčene z naravnimi števili od 1 do n. Sledi n vrstic, od katerih i-ta vsebuje celi števili x_i in y_i, ki sta koordinati i-te točke. Sledi še m vrstic, od katerih i-ta vsebuje celi števili u_i in v_i, ki povesta, da obstaja povezava med točkama u_i in v_i.

Omejitve vhodnih podatkov: $1 \leq n \leq 5 \cdot 10^4$; $1 \leq m \leq 5 \cdot 10^4$; za vsako točko i velja $|x_i| \leq 10^7$ in $|y_i| \leq 10^7$; za vsako povezavo i velja $1 \leq u_i \leq n$, $1 \leq v_i \leq n$ in $u_i \neq v_i$. Graf je povezan.

Izhodni podatki. Izpiši toliko vrstic, kolikor ima graf lic. Vsaka vrstica naj vsebuje kanonični opis enega od lic. Te opise izpiši v naraščajočem leksikografskem vrstnem

redu. (To pomeni, da najprej primerjaš dva opisa po prvi točki; če je ta pri obeh enaka, ju primerjaš po drugi točki; in tako naprej.)

Primer vhoda:

```
8 12
1 1
-1 -1
-1 1
2 2
2 -2
-2 -2
-2 2
1 -1
1 3
1 4
1 8
2 3
2 6
2 8
3 7
4 5
4 7
5 6
5 8
6 7
```

Pripadajoči izhod:

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

Ta primer prikazuje slika zgoraj v besedilu naloge.

Primer vhoda:

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

Pripadajoči izhod:

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

Y. Snežna odeja

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

Dana je slika pokrajine, narisana z znaki „`#`“ in „`.`“ na karirasti mreži $w \times h$ celic. Tole je na primer slika drevesa in hiše:

```
............
............
............
.####.......
######...#..
.####...#.#.
..#....#...#
.###...#####
```

Pokrajino nato prekrije odeja snežink, ki jih predstavljajo znaki „`*`“ in ki padajo navpično z zgornjega roba slike. Sneg pade na vsak znak „`#`“, ki ni „pokrit“, torej ki ne stoji pod kakšnim više ležečim znakom „`#`“ v istem stolpcu. Če v nekem stolpcu ni nobenega znaka „`#`“, pade tam sneg na dno stolpca. Snežna odeja je visoka 3 enote. V zgornjem primeru je rezultat naslednji:

```
.****.......
******...*..
******..***.
*#####*.*****
######.**#**
.####.**#.#*
..#...*#...#
.###..*#####
```

Napiši program, ki prebere sliko pokrajine brez snega in izpiše sliko pokrajine po tistem, ko zapade sneg.

Vhodni podatki. V prvi vrstici sta celi števili w (širina slike) in h (višina slike), ločeni s presledkom. Sledi h vrstic z vsebino slike; vsaka od njih je dolga po w znakov, vsebuje pa le znake „`#`“ in „`.`“. Zagotovljeno je, da so v zgornjih treh vrsticah le znaki „`.`“.

Omejitve vhodnih podatkov: $3 \leq w \leq 100$, $3 \leq h \leq 100$.

Izhodni podatki. Izpiši sliko pokrajine po tistem, ko zapade sneg. Slika naj bo sestavljena iz h vrstic, vsaka od teh pa iz w znakov. Debelina snega je vedno enaka: 3 snežinke v vsakem stolpcu.

Primer vhoda:

```
12 8
............
............
............
.####.......
######...#..
.####...#.#.
..#....#...#
.###...#####
```

Pripadajoči izhod:

```
.****.......
******...*..
******..***.
*#####*.*****
######.**#**
.####.**#.#*
..#...*#...#
.###..*#####
```

Z. Časovni napad

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

Upravljalsko geslo sodniškega sistema na CERC je Zelo Varno™ in vesolje ne bo obstajalo dovolj dolgo, da bi lahko preizkusil vsa možna gesla. Ne bomo ti povedali niti dolžine gesla, vendar pa mora biti zaradi omejitev pomnilnika dolgo vsaj en znak in kvečjemu 23 znakov; vsebuje pa lahko črke angleške abecede (velike in male), številke in naslednje posebne znake:

```
_ ? ! @ # $ % ^ & * ( ) + = - > < / [ ] { } \ | ; : ' "
```

Spisali smo celo svoj Preverjalnik Gesel™, ki pazljivo pregleda znake gesla, po vrsti od začetka proti koncu, in jih enega po enega primerja (z Algoritmom Enakosti Znakov™) z znaki tvojega niza, ki si ga vnesel, ko poskušaš uganiti pravo geslo. Preverjanje posameznega znaka vedno traja enako dolgo (en Preverjalni Cikel™), primerjati pa nehamo takoj, ko pridemo do konca enega ali drugega niza (pravega gesla in tvojega niza) ali pa opazimo, da se znaka ne ujemata.

Napiši program, ki ugane upravljalsko geslo CERCovega sodniškega sistema.

Vhodni in izhodni podatki. To je interaktivna naloga, pri kateri se bo tvoj program sporazumeval z ocenjevalnim sistemom prek standardnega vhoda in izhoda.

Tvoj program naj izpisuje nize, s katerimi poskuša uganiti geslo, vsakega v svojo vrstico, ocenjevalni sistem pa bo po vsakem odgovoril s številom Preverjalnih Ciklov™, ki jih je porabil, da je ugotovil, da je tvoje geslo napačno; če pa boš uganil pravo geslo, bo izpisal „`SUCCESS`“.

Po 5000 napačnih poskusih bo ocenjevalni sistem odgovoril z „`BLOCKED`“. To, kako dolge nize smeš pošiljati kot svoja ugibanja in iz kakšnih znakov smejo biti sestavljeni, ni posebej omejeno, vendar pa imej v mislih, da utegne tvoja rešitev prekoračiti časovno omejitev, če bo pošiljala preveč podatkov.

Primer:

Tvoj program izpiše:	Ocenjevalni sistem odgovori:
`CERC`	`1`
`probably_not_the_password`	`2`
`password?!`	`2`
`paprika`	`3`
`pa55`	`4`
`pa55w0rd`	`8`
`pa55w0rD`	`SUCCESS`

Komentar. Pri tej nalogi tvoj program pošilja nize ocenjevalnemu sistemu, ki odgovarja s števili. Pravo geslo je bilo „`pa55w0rD`“. Vsa prejšnja ugibanja so bila napačna.

Ko ocenjevalni sistem izpiše „`SUCCESS`“ ali „`BLOCKED`“, se bo nehal odzivati in tvoj program mora končati z izvajanjem znotraj časovne omejitve, sicer bo tvoja rešitev zavrnjena zaradi prekoračitve časovne omejitve (TLE).

Pazi na to, da po vsaki izpisani vrstici splakneš (*flush*) standardni izhod, da bo besedilo res prišlo do ocenjevalnega sistema. Za navodila, kako to narediti v različnih programskih jezikih, si oglej dokumentacijo (`https://putka-cerc.acm.si/info/#samples`).

NEUPORABLJENE NALOGE IZ LETA 2021

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 16. tekmovanjem ACM v znanju računalništva (leta 2021), 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. 193–209) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Eskalacija

Podjetje VelikiSmo, d. d., je veliko podjetje z ekipami po celem svetu. Vsakič, ko ima kaka ekipa problem, ki zadeva še kakšno drugo ekipo, sprožijo videokonferenco, na katero pokličejo predstavnike prizadetih ekip.

Dežurni predstavniki ekip imajo pet minut, da se oglasijo, sicer se klic ponovi.

Če se dežurni ne javi niti pet minut po drugem klicu, se pokliče šefa ekipe, po enakem postopku, kot velja za dežurne (če se šef ne oglasi po drugem klicu, gre klic naprej na šefovega šefa in tako naprej).

Tvoja naloga je, da napišeš nekaj podprogramov (oz. funkcij), ki sprejemajo dogodke (nova sekunda, nova zahteva) in izvedejo postopek klica in eskalacije. Ne kliči ljudi, ki so že prisotni na sestanku. **Napiši podprograme**:

- Utrip(), ki ga sistem pokliče vsako sekundo;
- Dodaj(**int** uporabnik), ki ga sistem pokliče, kadar potrebujemo danega uporabnika v konferenčnem klicu;
- Prijava(**int** uporabnik), ki ga sistem pokliče, ko se je uporabnik pridružil klicu.

Na voljo imaš naslednja podprograma, za katera lahko predpostaviš, da že obstajata in ju lahko kličeš iz svojih podprogramov:

- Poklici(**int** uporabnik) — pokliče uporabnika s to številko;
- **int** Sef(**int** uporabnik) — vrne uporabniško številko šefa podanega uporabnika.

Funkcija Sef vedno vrne veljavnega uporabnika — šef glavnega direktorja je sam glavni direktor.

2. Mehurčki

Danih je n „mehurčkov“ — skupin ljudi, ki so med seboj vsakodnevno v stiku. Za vsako skupino dobimo seznam imen ljudi v njej (nihče ne pripada več kot eni skupini). Poleg tega je podan tudi seznam s „izrednih“ oz. dodatnih stikov, torej primerov, ko sta si bila dva človeka v stiku, čeprav morda nista iz istega mehurčka. **Napiši podprogram**, ki na podlagi teh podatkov odgovori na p poizvedb; pri vsaki poizvedbi je dano ime nekega človeka, tvoj podprogram pa mora izračunati seznam ljudi, ki morajo v karanteno zaradi tveganega stika, če se tisti človek okuži z nalezljivo boleznijo.

3. Pobeg iz močvare

Neustrašni raziskovalec se je znašel v hudi zagati, saj mu primankuje energije in ne more sam pobegniti iz močvare. Močvara je prevokotne oblike iz $w \times h$ kvadratnih polj, naš raziskovalec pa začne zgoraj levo, na polju $(1, 1)$, in se mora prebiti do polja spodaj desno, torej (w, h). Raziskovalec se po močvari premika s skoki med posameznimi polji. Za skok s polja (x', y') na (x, y) porabi $|x' - x| + |y' - y|$ energije, zato je tak skok mogoč le, če je pred njim imel vsaj toliko energije. Poleg tega ciljno polje (x, y) ne sme biti isto kot začetno polje (x', y'). Po pristanku na polju (x, y) popije raziskovalec napoj, ki njegovi energiji prišteje vrednost p_{xy} (ta vrednost je lahko tudi negativna; če med svojim potovanjem večkrat skoči na isto polje, popije tak napoj po vsakem pristanku na njem, ne le po prvem).

Opiši postopek, ki kot vhodne podatke dobi w, h in vrednosti p_{xy} za vsa polja ($1 \leq x \leq w$, $1 \leq y \leq h$) ter izračuna najmanjše potrebno število skokov, s katerimi lahko raziskovalec pride na polje (w, h). Začetna energija raziskovalca je enaka energiji, ki jo povrne napoj na celici $(1, 1)$, torej p_{11}.

To nalogo si lahko predstavljamo v več različicah, od lažjih do težjih: (*a*) Reši nalogo z dodatno omejitvijo, da lahko raziskovalec skače le dol in desno; z drugimi besedami, skok z (x', y') na (x, y) je mogoč le, če je $x \geq x'$ in $y \geq y'$ in če je vsaj ena od teh dveh neenakosti stroga. (*b*) Reši nalogo, pri čemer lahko predpostaviš, da neko primerno zaporedje skokov od $(1, 1)$ do (w, h) res obstaja. (*c*) Reši nalogo, kot je opisana zgoraj, brez dodatnih zagotovil ali predpostavk.

4. Barvanje zebre

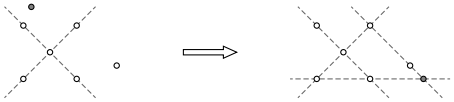
Dana je daljica dolžine D. Na začetku je cela daljica črne barve. Nato dobimo zaporedje več operacij, pri čemer je i-ta operacija oblike „na daljici pobarvamo interval od ℓ_i do d_i z barvo c_i“. Velja seveda $0 \leq \ell_i < d_i \leq D$, barva c_i pa je lahko 0 (črna) ali 1 (bela). **Opiši postopek**, ki po vrsti prebira takšne operacije in po vsaki prebrani operaciji izpiše skupno dolžino tistih delov daljice, ki so trenutno pobarvani belo. Na območjih, kjer se več intervalov $[\ell_i, d_i]$ prekriva, velja seveda zadnja tam uporabljena barva.

5. Prevoz po mreži

Ozemlje neke države ima obliko pravokotne kariraste mreže, sestavljene iz $w \times h$ kvadratnih celic (w stolpcev, h vrstic). V vsaki celici je cesta, ki povezuje dve diagonalno nasprotni oglišči celice; cesta ima torej dve možni orientaciji: / (povezuje spodnje levo in zgornje desno oglišče) in \ (povezuje zgornje levo in spodnje desno oglišče). Za vsako cesto je znana njena trenutna orientacija, pa tudi njena nosilnost: j-ta cesta v i-ti vrstici ima orientacijo o_{ij} in nosilnost c_{ij}. Celicam smemo spreminjati orientacijo (iz / v \ ali obratno), to pa bi radi storili tako, da bo potem obstajala v mreži neprekinjena pot od zgornjega levega do spodnjega desnega kota in da bo minimalna nosilnost po vseh cestah na tej poti večja ali enaka k (recimo, da bi radi od zgornjega levega do spodnjega desnega kota prepeljali tovornjak z maso k). **Opiši postopek**, ki izračuna najmanjše število celic, ki jim je treba spremeniti orientacijo.

6. Šolarkina uganka

Danih je n različnih točk v ravnini. Eno izmed njih smemo premakniti na poljuben nov položaj, to pa bi radi storili tako, da bo potem obstajalo čim več različnih takih premic, na katerih bodo ležale po vsaj tri izmed naših točk. **Opiši postopek**, ki kot vhodne podatke dobi koordinate vseh n točk in ugotovi, katero od njih bi bilo treba premakniti in kakšne naj bodo njene nove koordinate. Za koordinate vhodnih točk lahko predpostaviš, da so cela števila, ki po absolutni vrednosti ne presegajo 10^6.



Zgornja slika kaže primer z $n = 7$ točkami. Črtkane črte kažejo premice, na katerih ležijo po vsaj tri izmed naših točk. S premikom sive točke smo število takih premic povečali z dveh na štiri.

7. Palindromska razbitja

Palindrom je niz, ki se ne spremeni, če njegove znake preberemo z desne proti levi namesto z leve proti desni, na primer *radar* ali *neradodaren*. **Opiši postopek**, ki izračuna najmanjše možno število palindromov, na katere je mogoče razbiti dani niz s. *Primer:* niz `aababb` je mogoče razbiti na tri palindrome (kot `aa|bab|b` ali `a|aba|bb`), ne pa na manj kot tri.

8. Stolp

Dan je stolp n kock; vsaka kocka je v eni od B možnih barv. Poleg tega je podana tudi neka celoštevilska konstanta k. Osnovna operacija na stolpu je, da si izberemo neko barvo b in nato med najnižjimi k kockami na skladu pobrišemo tiste, ki so barve b (sklad se potem seveda malo sesede, tako da v njem ne ostane lukenj; medsebojni vrstni red nepobrisanih kock pa ostane nespremenjen). **Opiši postopek**, ki izračuna najmanjše število takih operacij, s katerimi je mogoče pobrisati vse kocke stolpa.

9. Mediana

Dana je tabela števil $a[1..n]$. Radi bi odgovorili na q poizvedb, pri čemer je i-ta poizvedba oblike „kaj je mediana števil $a[\ell_i], a[\ell_i + 1], \ldots, a[d_i]$?“ Števila ℓ_i in d_i (velja seveda $1 \leq \ell_i \leq d_i \leq n$) so za vse $i = 1, \ldots, q$ podana vnaprej. **Opiši postopek**, ki čim hitreje izračuna odgovore na vse te poizvedbe.

REŠITVE NALOG ZA PRVO SKUPINO SŠ

1. Neurejene besede

Črke posamezne besede lahko naključno premešamo takole: pojdimo v zanki po črkah besede od leve proti desni; ko smo pri k-ti črki, izberimo naključno število r z območja od 1 do k (vsa z enako verjetnostjo) in zamenjajmo k-to ter r-to črko besede (mogoče je torej tudi, da dobimo $r = k$ in torej k-ta črka ostane, kjer je bila). Če imamo n znakov dolgo besedo, je te znake načeloma mogoče premešati na $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$ različnih načinov (ni sicer nujno, da vsi ti načini dajo različne besede — na primer, če v besedi `bob` zamenjamo prvo in tretjo črko, ostane enaka); z indukcijo po n se lahko prepričamo, da lahko prej opisani postopek premeša to besedo na vseh teh $n!$ načinov in da so vsi enako verjetni.

Zdaj torej znamo premešati znake ene besede; ker pa dobimo pri tej nalogi niz, v katerem je lahko tudi več besed, bomo postopek iz prejšnjega odstavka ovili v še eno zanko, ki pregleda celoten niz in obdela vse besede v njem. Oglejmo si implementacijo te rešitve v C++. Zunanja zanka se z i premika po znakih vhodnega niza; notranja zanka obdela trenutno besedo in ob koncu pusti i na naslednjem ne-črkovnem znaku, ki ga potem ++i v zunanji zanki preskoči. V primeru, ko je na indeksu i že ob začetku notranje zanke neki ne-črkovni znak, notranja zanka ne naredi ničesar (zunanja zanka pa se nato premake na naslednji znak).

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

void PremesajCrke(string &s)
{
  // Sprehodimo se po vhodnem nizu.
  for (int i = 0, n = s.length(); i < n; ++i)
    // Premešajmo znake naslednje besede.
    for (int od = i; i < n && isalpha(s[i]); ++i)
      // Postavimo znak i na naključno mesto med „od“ in „i“.
      swap(s[i], s[od + Random(i - od + 1)]);
}

int main()
{
  string s; getline(cin, s); // Preberimo vhodni niz.
  PremesajCrke(s); // Premešajmo črke vsake besede.
  cout << s << endl; return 0; // Izpišimo rezultat.
}
```

Tu smo uporabili funkcijo Random, za katero naloga pravi, da je že podana. Lahko pa namesto tega uporabimo generator naključnih števil iz standardne knjižnice:

```
#include <random>

void PremesajCrke2(string &s)
{
  random_device rnd;
```

```
  // Sprehodimo se po vhodnem nizu.
  for (int i = 0, n = s.length(); i < n; ++i)
    // Premešajmo znake naslednje besede.
    for (int od = i; i < n && isalpha(s[i]); ++i)
      // Postavimo znak i na naključno mesto med „od“ in „i“.
      swap(s[i], s[uniform_int_distribution(od, i)(rnd)]);
}
```

Tudi za mešanje znakov posamezne besede imamo v standardni knjižnici koristno funkcijo, std::shuffle; kot parametra ji moramo podati iteratorja, ki kažeta na prvi znak besede in prvi znak za besedo:

```
void PremesajCrke3(string &s)
{
  random_device rnd;
  // Sprehodimo se po vhodnem nizu.
  for (int i = 0, n = s.length(); i < n; ++i) {
    // Poglejmo, kje se konča trenutna beseda.
    int od = i; while (i < n && isalpha(s[i])) ++i;
    // Premešajmo znake trenutne besede.
    shuffle(s.begin() + od, s.begin() + i, rnd); }
}
```

Oglejmo si še rešitev v pythonu. Ker niza ne moremo spreminjati, ga lahko najprej predelamo v seznam, kjer je vsak znak samostojen element:

```
def PremesajCrke(s):
  i = 0; n = len(s)
  s = list(s) # Spremenimo s v seznam, da ga bomo lahko spreminjali.
  while i < n: # Sprehodimo se po vhodnem nizu.
    # Sprehodimo se po znakih naslednje besede.
    od = i
    while i < n and s[i].isalpha():
      # Postavimo znak s[i] na naključen indeks med „od“ in „i“.
      r = od + Random(i − od + 1)                                    (†)
      s[r], s[i] = s[i], s[r]
      i += 1
    i += 1 # Preskočimo trenutni ne-črkovni znak.
  return "".join(s) # Staknimo znake spet v niz.

# Preberimo niz in ga izpišimo s premešanimi znaki besed.
print(PremesajCrke(input()))
```

Tudi tu bi lahko uporabili generator naključnih števil iz pythonove knjižnice; na začetek programa bi morali dodati **import** random, nato pa vrstico (†) zamenjati z:

```
      r = random.randrange(od, i + 1)
```

Lahko pa uporabimo funkcijo shuffle (iz modula random), ki premeša elemente danega zaporedja; notranjo zanko podprograma PremesajCrke zamenjajmo z:

```
    while i < n and s[i].isalpha(): i += 1
    t = s[od:i]; random.shuffle(t); s[od:i] = t
```

Za ljubitelje pythonove standardne knjižnice je tu še enovrstična rešitev. Besede v vhodnem nizu lahko poiščemo z regularnim izrazom `\w+` (torej zaporedja ene ali več črk); s funkcijo sub iz modula re lahko potem vsako besedo zamenjamo z nizom, ki ga pripravi naša vgnezdena funkcija, ki jo podamo z lambda-izrazom. Ta funkcija dobi kot parameter objekt m tipa Match, ki nam kot m[0] vrne trenutno besedo; s funkcijo sample iz modula random (ki iz danega zaporedja naključno izbira elemente brez vračanja) lahko pripravimo seznam, v katerem so vsi znaki te besede v naključno premešanem vrstnem redu, nato pa jih moramo le še stakniti skupaj v nov niz (z metodo join). Tako smo dobili naslednjo rešitev:

```
import random, re

def PremesajCrke2(s):
  return re.sub(r"\w+", lambda m: "".join(random.sample(m[0], len(m[0]))), s)
```

2. Kibi, mebi

Nalogo lahko načeloma rešimo z nekaj pogojnimi stavki: če je velikost — recimo ji x — manjša ali enaka 9999, jo lahko izpišemo kar v bajtih; sicer, če je manjša ali enaka $9999 \cdot 2^{10}$, jo lahko izpišemo v kilobajtih; sicer, če je manjša ali enaka $9999 \cdot 2^{20}$, jo lahko izpišemo v megabajtih; in tako naprej. Naloga pravi, da če x pri pretvorbi v večje enote ni celo število, ga moramo zaokrožiti navzgor na naslednje celo število. Običajno celoštevilsko deljenje pa zaokroža bodisi navzdol (npr. operator // v pythonu) bodisi proti 0 (npr. v C/C++ in podobnih jezikih), kar v našem primeru tudi pomeni navzdol; da dobimo zaokrožanje navzgor, lahko izkoristimo dejstvo, da (za celoštevilska x in d, kjer je $d > 0$) velja $\lfloor (x + d - 1)/d \rfloor = \lceil x/d \rceil$.

```
  if (x <= 9999) cout << x << " B" << endl;
  else if (x <= 9999 * 1024) cout << ((x + 1024 - 1) / 1024) << " KB" << endl;
  else ...   // in tako naprej.
```

Namesto množenja in deljenja s 1024 (in njegovimi potencami) pa lahko uporabimo tudi operatorja za zamikanje bitov, << in >>, saj se pri množenju s 1024 (kar je 2^{10}) število zamakne za 10 bitov v levo, pri deljenju s 1024 pa za 10 bitov v desno. Tako dobimo naslednjo rešitev:

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

void Izpisi(uintmax_t x)
{
  constexpr uintmax_t M = 9999;
  if (x <= M) cout << x << " B";
  else if (x <= (M << 10)) cout << ((x + (uintmax_t(1) << 10) - 1) >> 10) << " KB";
  else if (x <= (M << 20)) cout << ((x + (uintmax_t(1) << 20) - 1) >> 20) << " MB";
  else if (x <= (M << 30)) cout << ((x + (uintmax_t(1) << 30) - 1) >> 30) << " GB";
  else if (x <= (M << 40)) cout << ((x + (uintmax_t(1) << 40) - 1) >> 40) << " TB";
  else cout << ((x + (uintmax_t(1) << 50) - 1) >> 50) << " PB";
}
```

Uporabili smo največji razpoložljivi celoštevilski tip, **uintmax_t** (ki je načeloma dolg vsaj 64 bitov), da bo naloga delovala tudi za velike vrednosti x (na primer: 10000

PB je približno $2^{63,3}$ bajtov). Naloga sicer tega od nas ne zahteva, saj pravi, da smemo predpostaviti, da imajo številski podatkovni tipi neomejen obseg.

Še ena možnost je, da x pretvarjamo v večje enote postopoma: izkoristimo dejstvo, da je $\lceil x/d^2 \rceil = \lceil \lceil x/d \rceil / d \rceil$, torej lahko v vsakem koraku delimo x s 1024 in rezultat zaokrožimo navzgor. Tako dobimo:

```
void Izpisi2(uintmax_t x)
{
  if (x <= 9999) { cout << x << " B"; return; }
  x = (x + 1023) / 1024; if (x <= 9999) { cout << x << " KB"; return; }
  x = (x + 1023) / 1024; if (x <= 9999) { cout << x << " MB"; return; }
  x = (x + 1023) / 1024; if (x <= 9999) { cout << x << " GB"; return; }
  x = (x + 1023) / 1024; if (x <= 9999) { cout << x << " TB"; return; }
  x = (x + 1023) / 1024; cout << x << " PB";
}
```

Ker pa so si posamezne vrstice naše rešitve tako podobne, jo lahko elegantno zapišemo tudi z zanko:

```
void Izpisi3(uintmax_t x)
{
  // Pripravimo si tabelo predpon.
  static constexpr const char *predpone[] = {"", "K", "M", "G", "T", "P"};
  static constexpr int stPredpon = sizeof(predpone) / sizeof(predpone[0]);

  // Delimo x s 1024 (če se deljenje ne izide, zaokrožimo navzgor),
  // dokler ne pade pod 10000 ali pa ne pridemo do zadnje predpone.
  int i = 0; while (x > 9999 && i + 1 < stPredpon) x = (x + 1023) / 1024, ++i;

  // Izpišimo x s trenutno predpono.
  cout << x << " " << predpone[i] << "B" << endl;
}
```

Zapišimo našo rešitev še v pythonu:

```
def IzpisiVelikost(x):
  for predpona in "|K|M|G|T|P".split("|"):

    # Poglejmo, če je x dovolj kratek (ali pa smo pri zadnji predponi).
    if x <= 9999 or predpona == "P": break

    # Pretvorimo x v naslednjo večjo enoto.
    x = (x + 1023) // 1024

  # Izpišimo x v izbranih enotah.
  print("%d %sB" % (x, predpona))
```

3. Lučka

Ko pritisnemo na tipko, se svetlost lučke poveča — razen če je bila že pred pritiskom na najvišji stopnji, tedaj pa se zmanjša. Najvišjo stopnjo svetlosti prepoznamo torej po tem, da če takrat pritisnemo na tipko, se svetlost zmanjša namesto poveča. Tako lahko torej v zanki pritiskamo na tipko in merimo svetlost, dokler ne opazimo, da se je po zadnjem pritisku svetlost zmanjšala. Takrat vemo, da je bila lučka pred tem zadnjim pritiskom na najvišji možni svetlosti; zdaj, po pritisku, pa je na *najnižji* svetlosti. Toda naloga od nas zahteva, da mora biti ob koncu izvajanja našega programa lučka na najvišji svetlosti, torej moramo še enkrat v zanki pritiskati tipko, dokler spet ne dosežemo najvišje svetlosti.

```
int main()
{
  int maxSvetlost = PreveriSvetlost();

  // Pritiskajmo tipko, dokler se svetlost še povečuje.
  while (true) {
    PritisniTipko();
    if (PreveriSvetlost() <= maxSvetlost) break;
    maxSvetlost = PreveriSvetlost(); }

  // Pritiskajmo tipko, dokler spet ne dosežemo najvišje svetlosti.
  while (PreveriSvetlost() != maxSvetlost) PritisniTipko();

  return 0;
}
```

Obe zanki lahko tudi združimo v eno samo. Če je po pritisku na tipko nova svetlost večja od doslej največje, si jo zapomnimo; če pa je enaka doslej največji, to pomeni, da je bila pred zadnjim pritiskom manjša, torej smo v preteklosti z največje svetlosti že padli na najmanjšo in se zdaj ravnokar spet povzpeli do največje, tako da lahko končamo.

```
int main()
{
  int maxSvetlost = PreveriSvetlost();

  while (true) {
    PritisniTipko();

    // Če je to najvišja svetlost doslej, si jo zapomnimo.
    if (PreveriSvetlost() > maxSvetlost) maxSvetlost = PreveriSvetlost();

    // Če smo že drugič prišli do najvišje svetlosti, lahko končamo.
    else if (PreveriSvetlost() == maxSvetlost) break; }

  return 0;
}
```

Zapišimo prvo od teh dveh rešitev še v pythonu:

```
maxSvetlost = PreveriSvetlost()

# Pritiskajmo tipko, dokler se svetlost še povečuje.
while True:
  PritisniTipko()
  if PreveriSvetlost() <= maxSvetlost: break
  maxSvetlost = PreveriSvetlost()

# Pritiskajmo tipko, dokler spet ne dosežemo najvišje svetlosti.
while PreveriSvetlost() != maxSvetlost: PritisniTipko()
```

4. Oviratlon

Tekmovalec se premika v smeri naraščajočih y-koordinat, zato bo imela vsaka naslednja ovira, na katero bo naletel, višjo y-koordinato kot prejšnja. Zato je koristno za začetek urediti vse ovire naraščajoče po y_i (kajti besedilo naloge pravi, da v vhodnih podatkih niso nujno urejene) in jih potem pregledovati v tem vrstnem redu.

Pri vsaki oviri se zdaj vprašajmo, ali se bo naš tekmovalec zaletel vanjo ali ne. Če je tekmovalec trenutno na (x, y), ovira pa gre od (x_{i1}, y_i) do (x_{i2}, y_i), se bo zaletel vanjo v primeru, ko je $x_{i1} < x < x_{i2}$; takrat moramo pogledati, katero

krajišče mu je bližje — torej katera od razdalj $x - x_{i1}$ in $x_{i2} - x$ je manjša — in tekmovalca premakniti v tisto krajišče. Tako lahko postopoma sledimo njegovi poti in tudi seštevamo dolžine vseh premikov; na koncu pa ne pozabimo prišteti še dolžine premika od zadnjega položaja do ciljne črte.

Zapišimo ta postopek s psevdokodo:

uredi ovire naraščajoče po y-koordinati in jih v tem vrstnem redu oštevilči;
$x := x_z$; $y := y_z$; $d := 0$; (* *Trenutni položaj tekmovalca in dolžina poti.* *)
for $i := 1$ **to** n: (* *Pojdimo v zanki po vseh ovirah.* *)
  (* *Ali se tekmovalec zaleti v to oviro?* *)
  if $y_i < y$ **or** $x_{i1} < x$ **or** $x_{i2} > x$ **then continue**;
  (* *Premaknimo ga do ovire.* *)
  $d := d + y_i - y$; $y := y_i$;
  (* *Premaknimo ga v bližje krajišče ovire* (*oz. v levo, če sta obe enako daleč*). *)
  if $x - x_{i1} \le x_{i2} - x$ **then** $d := d + x - x_{i1}$; $x := x_{i1}$;
                else $d := d + x_{i2} - x$; $x := x_{i2}$;
(* *Pretečeni poti dodajmo še razdaljo do ciljne črte.* *)
return $d + y_c - y$;

Časovna zahtevnost tega postopka je $O(n \log n)$ zaradi urejanja ovir po y-koordinati; ko so ovire enkrat urejene, nam preostanek postopka vzame le $O(n)$ časa, saj imamo v naši zanki z vsako oviro le konstantno mnogo dela.

5. Videostena

Ko prebiramo podatke s standardnega vhoda, lahko v neko tabelo ali vektor zapisujemo za vsak zaslon številko njegovega desnega soseda. Poleg tega si tudi zapomnimo, kateri zaslon ni imel levega soseda (torej je imel tam -1), kajti tisto je potem najbolj levi zaslon. Ko preberemo vse vhodne podatke, začnemo pri najbolj levem zaslonu in se potem s pomočjo prej omenjene tabele na vsakem koraku premaknemo s trenutnega zaslona na njegovega desnega soseda; tako lahko sledimo zaslonom od leve proti desni in jih izpisujemo. Ustavimo se, ko trenutni zaslon nima desnega soseda (torej ko je tam -1).

Naloga pravi, da moramo paziti še na možnost, da podatki za kakšen zaslon manjkajo. Ena možnost je, da manjka najbolj levi zaslon; to bomo prepoznali po tem, da pri nobenem zaslonu kot njegov levi sosed ni navedeno število -1. Če pa manjka kak kasnejši zaslon, bomo to opazili na koncu pri sprehajanju po zaslonih od leve proti desni: kot desnega soseda prejšnjega zaslona lahko dobimo neki zaslon, za katerega nismo prebrali podatka o *njegovem* desnem sosedu. V naši tabeli desnih sosedov moramo torej znati ločiti med primerom, ko neki zaslon nima desnega soseda, in primerom, ko za neki zaslon nimamo podatka o tem, kdo je njegov desni sosed oz. ali ga sploh ima. V ta namen spodnji program uporablja vrednost 0 za manjkajoče podatke in -1 za primere, ko vemo, da desnega soseda ni.

```
#include <cstdio>
#include <vector>                                          // (1)
using namespace std;

int main()
```

```
{
  enum { M = 1000, MANJKA = 0 };
  vector<int> naslednji(M + 1, MANJKA); // naslednji[z] = desni sosed zaslona z     // (2)
  int prvi = MANJKA; // najbolj levi zaslon

  while (true)
  {
    // Preberimo podatke o še enem zaslonu.
    int levi, z, desni; if (scanf("%d %d %d", &levi, &z, &desni) != 3) break;
    if (levi < 0) prvi = z; // Če nima levega soseda, je to najbolj levi zaslon.
    naslednji[z] = desni;   // Zapomnimo si njegovega desnega soseda.
  }

  // Naštejmo zaslone od leve proti desni.
  if (prvi == MANJKA) { fprintf(stderr, "Manjka najbolj levi zaslon.\n"); return 1; }
  for (int z = prvi; z >= 1; z = naslednji[z])
  {
    if (naslednji[z] == MANJKA) {                                               // (3)
      fprintf(stderr, "Manjka zaslon %d.\n", z); return 2; }
    printf("%d ", z);
  }

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

Če bi bile številke zaslonov večje (npr. do 10^9 namesto le do 1000), bi takšna rešitev s tabelo oz. vektorjem porabila preveč pomnilnika. Takrat bi bilo bolje uporabiti razpršeno tabelo oz. slovar, na primer razred map iz C++ove standardne knjižnice. Vse, kar bi morali v zgornji rešitvi spremeniti, sta vrstici (1) in (2):

```
  #include <unordered_map>                                                      // (1)
  ⋮
  unordered_map<int, int> naslednji; // naslednji[z] = desni sosed zaslona z     // (2)
```

Vrstica (3) bi še vedno delovala pravilno, kajti če ključa z takrat v slovarju še ni, ga bo unordered_map::**operator** [] dodal s pripadajočo vrednostjo 0 (kar je ravno enako naši konstanti MANJKA).

Oglejmo si še rešitev v pythonu. Tudi tu bomo uporabili slovar, saj je delo z njimi v pythonu zelo preprosto:

```
import sys
prvi = −1; naslednji = {}

# Preberimo podatke o zaslonih.
for vrstica in sys.stdin:
  [levi, z, desni] = [int(s) for s in vrstica.split()]
  if levi < 0: prvi = z    # Če nima levega soseda, je to najbolj levi zaslon.
  naslednji[z] = desni     # Zapomnimo si njegovega desnega soseda.

# Naštejmo zaslone od leve proti desni.
if prvi < 0: sys.stderr.write("Manjka najbolj levi zaslon.\n"); sys.exit(1)
z = prvi
while z > 0:
  if z not in naslednji: sys.stderr.write("Manjka zaslon %d.\n" % z); sys.exit(2)
  sys.stdout.write("%d " % z); z = naslednji[z]

sys.stdout.write("\n")
```

REŠITVE NALOG ZA DRUGO SKUPINO SŠ

1. Stoli

Ker je n (število stolov in ljudi) pri tej nalogi majhen, lahko dogajanje preprosto odsimuliramo. Pri tem bomo vzdrževali tabelo oz. vektor celih števil, ki za vsak stol povedo, kdo sedi na njem (če sploh kdo; vrednost 0 pomeni prazen stol). V glavni zanki berimo podatke o osebah; pri vsaki najprej preverimo, če je stol S_i prost; če je, lahko sede nanj, sicer pa moramo iti v notranji zanki po stolih od S_i v želeno smer in šteti, koliko prostih stolov skupaj vidimo.

Če zagledamo skupino $2R_i + 1$ strnjenih prostih stolov, lahko oseba i sede na srednjega od teh stolov; če pa pridemo do začetka oz. konca vrste (odvisno od tega, ali smo šli levo ali desno), preverimo, ali smo takrat videli vsaj $R_i + 1$ strnjenih stolov, in če smo jih, lahko oseba i sede na $(R_i + 1)$-vega od njih (gledano iz smeri, iz katere smo prišli). Naloga namreč ne zahteva, da mora biti na vsaki strani stola, kamor bi se i usedel, R_i praznih stolov, pač pa le, da nihče drug ne sme sedeti manj kot R_i mest levo ali desno; temu pogoju lahko ustrežemo bodisi s praznimi stoli bodisi s stem, da stolov sploh ni (ker smo že na začetku oz. koncu naše vrste n stolov).

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

int main()
{
  // Preberimo n in pripravimo vektor praznih stolov.
  int n; cin >> n; vector<int> stoli(n + 1, 0);

  // Preberimo in obdelajmo podatke o osebah.
  for (int i = 1; i <= n; ++i)
  {
    // Preberimo naslednjo osebo.
    int Si, Ri; char smer; cin >> Si >> Ri >> smer;
    int d = (smer == 'D') ? 1 : -1;

    // Ali lahko sede na želeni stol?
    if (stoli[Si] == 0) { stoli[Si] = i; continue; }

    // Sicer se premikajmo v želeno smer.
    for (int stol = Si, stProstih = 0; 1 <= stol && stol <= n; stol += d)
    {
      // Štejmo, kako dolgo strnjeno skupino prostih stolov imamo.
      if (stoli[stol] == 0) ++stProstih; else stProstih = 0;

      // Ali lahko sede tako, da bo imel na vsaki strani Ri prostih stolov?
      if (stProstih >= 2 * Ri + 1) { stoli[stol - d * Ri] = i; break; }

      // Če smo prišli do začetka/konca vrstice, je dovolj že, če je
      // vsaj Ri prostih stolov v smeri, iz katere smo prišli.
      if (stol == (d > 0 ? n : 1) && stProstih >= Ri + 1) {
        stoli[stol - d * (stProstih - Ri - 1)] = i; break; }
    }
  }

  // Izpišimo končno stanje.
  for (int j = 1; j <= n; ++j) cout << stoli[j];
```

```
  cout << endl; return 0;
}
```

Ta rešitev ima časovno zahtevnost $O(n^2)$. Če bi hoteli učinkovitejšo rešitev za večje n, bi bilo koristno vzdrževati strnjene skupine prostih stolov v neki primerno uravnoteženi drevesasti podatkovni strukturi (npr. drevo intervalov ali pa rdeče-črno drevo), kjer bi morali v vsakem notranjem vozlišču tudi vzdrževati dolžino najdaljše skupine v poddrevesu tistega vozlišča. Tako bi lahko v $O(\log n)$ časa poiskali najbližjo dovolj dolgo skupino; ko se oseba i usede na enega od stolov te skupine, pa bi skupina razpadla na dve krajši, torej bi jo morali pobrisati iz drevesa in dodati tisti dve, na kateri je razpadla; tudi to bi šlo v $O(\log n)$ časa. Ker bi morali to narediti pri vsaki osebi, bi imela takšna rešitev časovno zahtevnost $O(n \log n)$.

2. Tehtnica

Za vsako utež imamo tri možnosti: lahko jo damo v levo skodelico, lahko v desno, lahko pa je sploh ne uporabimo (to lahko prikladno opišemo s števili -1, 1 in 0). Ker imamo n uteži, se nam tako nabere $3 \cdot 3 \cdot \ldots \cdot 3 = 3^n$ kombinacij tega, kaj naredimo s katero utežjo. Vse te možnosti lahko pregledamo z rekurzivnim podprogramom, ki za trenutno utež k (s težo 3^k) v zanki preizkusi vse tri možnosti in za vsako izvede vgnezden rekurzivni klic, ki bo pregledal vse možne kombinacije tega, kaj narediti s preostalimi utežmi. Spotoma računajmo tudi vsoto že uporabljenih uteži — tiste na desni strani prištevajmo, tiste na levi pa odštevajmo. Rekurzija se neha gnezditi, ko pridemo do $k = 0$ (utež s težo 1); takrat izpišemo trenutni razpored uteži in vsoto njihovih tež.

Paziti moramo, da ne izpišemo razporedov z negativno vsoto. To bi lahko načeloma preverili tik pred izpisom, še bolje pa je, če si pomagamo z naslednjim opažanjem: najtežja uporabljena utež mora biti na desni strani, da jo bomo prišteli, kajti ona je več kot še enkrat težja od vseh lažjih uteži skupaj (utež k je težka 3^k, vse lažje uteži skupaj pa so težke $1 + 3 + 9 + \ldots + 3^{k-1} = \sum_{i=0}^{k-1} 3^i = (3^k - 1)/2$, torej za slabo polovico uteži k) in če jo odštejemo, niti vse lažje uteži skupaj ne bodo mogle spraviti vsote nazaj nad 0 (niti do 0).

Spodobi se tudi razmisliti, da naša rešitev ne izpiše kakšne teže po večkrat. Ali je mogoče isto vsoto tež dobiti pri dveh različnih razporedih uteži? Pa vzemimo v mislih dva takšna razporeda in naj bo k najtežja utež, ki jo tadva razporeda uporabita različno. Zaradi tega med njunima težama nastopi razlika vsaj 3^k; te razlike pa lažje uteži ne morejo izničiti, kajti tudi če jih en razpored vse uporabi v nasprotni skodelici kot drugi, bo razliko v težah to spremenilo za največ $2 \sum_{i=0}^{k-1} 3^i = 3^k - 1$, torej manj od 3^k. Čim se torej razporeda pri rabi neke uteži razlikujeta, bo med njunima težama nastala razlika, ki je z lažjimi utežmi ne bomo mogli izničiti; torej imata različna razporeda neizogibno tudi različno težo, zato se ne more zgoditi, da bi naš rekurzivni postopek izpisal isto težo po večkrat.

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

int n; // število uteži
vector<int> utezi, kako; // kako[k] pove, kako smo uporabili utež s težo utezi[k]
```

```
void Rekurzija(int k, int vsotaDoslej)
{
  // Preglejmo vse tri možnosti glede tega, kako uporabimo utež k.
  // Pazimo le na to, da prve (= najtežje) uporabljene uteži ne smemo
  // odštevati, pač pa le prištevati, saj bo drugače končna vsota negativna.
  for (kako[k] = (vsotaDoslej == 0 ? 0 : -1); kako[k] <= 1; ++kako[k])
  {
    int vsota = vsotaDoslej + kako[k] * utezi[k];
    if (k > 0) { Rekurzija(k - 1, vsota); continue; }
    // Pri k = 0 izpišimo rezultate.
    cout << "( ";
    for (int i = 0; i < n; ++i) if (kako[i] < 0) cout << utezi[i] << " ";
    cout << ") <=> ( ";
    for (int i = 0; i < n; ++i) if (kako[i] > 0) cout << utezi[i] << " ";
    cout << ") = " << vsota << endl;
  }
}

int main()
{
  // Preberimo število uteži in izračunajmo njihove teže.
  cin >> n; kako.resize(n); utezi.resize(n);
  utezi[0] = 1; for (int i = 1; i < n; ++i) utezi[i] = 3 * utezi[i - 1];
  // Z rekurzijo preglejmo vse možnosti.
  Rekurzija(n - 1, 0); return 0;
}
```

Ker gre naša rekurzija po padajočih k (od težjih uteži k lažjim) in pri vsakem k po naraščajočih kako[k], bodo tudi teže tako dobljenih razporedov nastajale v naraščajočem vrstnem redu, od 0 do največje možne teže $(3^n - 1)/2$.

Namesto z rekurzijo lahko vse možne razporede uteži pregledamo tudi z zanko. Ker imamo n uteži in pri vsaki tri možnosti, je to skupaj 3^n možnih razporedov, ki jih lahko predstavimo s števili od 0 do $3^n - 1$. Če takšno število zapišemo v trojiškem zapisu, recimo $a = (a_{n-1}a_{n-2}\ldots a_2a_1a_0)_3 = \sum_{k=0}^{n-1} a_k \cdot 3^k$, si lahko posamezno števko a_k predstavljamo kot podatek o tem, kako je v tem razporedu uporabljena utež k (tista s težo 3^k): $a_k = 0$ naj pomeni, da na levi, $a_k = 2$, da na desni, $a_k = 1$ pa, da je sploh ne uporabimo. Tak razpored torej potem predstavlja težo

$$\sum_{k=0}^{n-1}(a_k - 1) \cdot 3^k = \sum_{k=0}^{n-1} a_k \cdot 3^k - \sum_{k=0}^{n-1} 3^k = a - (3^n - 1)/2.$$

Ta razmislek nam tudi pove, da ni treba iti z a od 0 do $3^n - 1$; dovolj je, če začnemo pri $a = (3^n - 1)/2$ namesto pri $a = 0$, saj bi pri manjših a-jih nastale negativne teže, ki jih tako ali tako ne smemo izpisati.

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

int main()
{
  // Preberimo število uteži in izračunajmo potence števila 3.
  int n; cin >> n; vector<int> pot3(n + 1), kako(n);
  pot3[0] = 1; for (int i = 1; i <= n; ++i) pot3[i] = 3 * pot3[i - 1];
  // V zanki preglejmo vse možnosti.
```

```
  for (int a = (pot3[n] − 1) / 2; a < pot3[n]; ++a)
  {
    // Določimo razpored uteži iz trojiškega zapisa števila a.
    for (int i = 0, aa = a; i < n; ++i)
      kako[i] = (aa % 3) − 1, aa /= 3;
    // Izpišimo razpored.
    cout << "( ";
    for (int i = 0; i < n; ++i) if (kako[i] < 0) cout << pot3[i] << " ";
    cout << ") <=> ( ";
    for (int i = 0; i < n; ++i) if (kako[i] > 0) cout << pot3[i] << " ";
    cout << ") = " << (a − (pot3[n] − 1) / 2) << endl;
  }
  return 0;
}
```

Bolj za šalo kot zares pa si oglejmo še rešitev v pythonu, ki namesto z rekurzijo pregleda vseh 3^n možnih razporedov z n vgnezdenimi zankami. Ker n-ja ne poznamo vnaprej, bomo izvorno kodo za n vgnezdenih zank pripravili kot niz in jo izvedli s pythonovo funkcijo exec.

```
n = int(input()) # Preberimo število uteži.
kako = [0] * n   # kako[k] = kako uporabimo utež k pri trenutnem razporedu

def Izpisi(): # Izračuna težo trenutnega razporeda in ga izpiše.
  vsota = sum(kako[k] * 3**k for k in range(n))
  if vsota < 0: return
  levo = " ".join(str(3**k) for k in range(n) if kako[k] < 0)
  desno = " ".join(str(3**k) for k in range(n) if kako[k] > 0)
  print(f"({levo}) <=> ({desno}) = {vsota}")

# Pripravimo n vgnezdenih zank, ki pregledajo vse možne razporede uteži.
koda = []
for k in range(n):
  koda.append("%sfor kako[%d] in (-1, 0, 1):" % (" " * k, n − 1 − k))
koda.append(" " * n + "Izpisi()")
exec("\n".join(koda)) # Izvedimo naše vgnezdene zanke.
```

Za konec razmislimo še o različici naloge, ki jo omenja opomba pod črto na koncu besedila: recimo, da bi radi za dano težo t poiskali razpored uteži, pri katerem bo desna skodelica ravno za t težja od leve. Videli smo že, da je najtežja utež težka 3^{n-1}, vse lažje skupaj pa $\sum_{k=0}^{n-2} 3^k = (3^{n-1} - 1)/2$; vse uteži z najtežjo vred pa so skupaj težke $(3^n - 1)/2$.

(1) Če je torej $t > (3^n - 1)/2$, se teže t ne bo dalo dobiti, niti če položimo vse uteži v desno skodelico.

(2) Če je $t \leq (3^n - 1)/2$ in $t > (3^{n-1} - 1)/2$, moramo uporabiti najtežjo utež, kajti z ostalimi se ne dá sestaviti tež, večjih od $(3^{n-1} - 1)/2$. Najtežjo utež pa lahko uporabimo le v desni skodelici: če bi jo dali v levo, bi bila teža na koncu gotovo negativna, saj je najtežja utež več kot še enkrat težja od vseh ostalih skupaj. (2.1) Če je $t \geq 3^{n-1}$, nam zdaj do teže t manjka še $t - 3^{n-1}$, kar moramo sestaviti s preostalimi $n - 1$ utežmi; v nadaljevanju imamo torej problem, ki je prav tak kot prvotni, le z manjšo težo in z eno utežjo manj. (2.2) Če pa je $t < 3^{n-1}$, je zdaj desna skodelica za $3^{n-1} - t$ pretežka; tako težo moramo zdaj sestaviti s preostalimi

$n-1$ utežmi, vendar moramo pri tem zamenjati vlogo obeh skodelic, tako da bodo preostale uteži težo našega razporeda zmanjšale za $3^{n-1}-t$.

(3) Če pa je $t \le (3^{n-1}-1)/2$, najtežje uteži ne smemo uporabiti, kajti če jo položimo v desno skodelico, je potem najnižja vsota, ki jo še lahko sestavimo, tista, pri kateri položimo vse ostale uteži v levo skodelico; takrat dobimo $3^{n-1}-(3^{n-1}-1)/2 = (3^{n-1}+1)/2 > t$. Težo t lahko torej dobimo le tako, da uteži 3^{n-1} sploh ne uporabimo; v nadaljevanju rešujemo enak problem kot doslej, le z eno utežjo manj.

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

// Poišče razpored uteži v levi in desni skodelici, pri katerem je razlika v teži med desno
// in levo enaka t. Vrne logično vrednost, ki pove, ali tak razpored sploh obstaja.
bool Sestavi(int n, int t, vector<int> &levo, vector<int> &desno)
{
  // Pripravimo si tabelo potenc števila 3.
  vector<int> pot3(n + 1); pot3[0] = 1;
  for (int k = 1; k <= n; ++k) pot3[k] = 3 * pot3[k - 1];

  // Poglejmo, ali težo t sploh lahko sestavimo z n utežmi.
  levo.clear(); desno.clear();
  if (t > (pot3[n] - 1) / 2) return false;

  // Določimo razpored uteži.
  bool obrni = false; // Ali je vloga obeh skodelic obrnjena?
  for (int k = n - 1; k >= 0; --k)
  {
    // Morda uteži k sploh ni treba uporabiti.
    if (t <= (pot3[k] - 1) / 2) continue;

    // Položimo jo v ustrezno skodelico.
    (obrni ? levo : desno).emplace_back(pot3[k]);

    // Z ostalimi utežmi bomo morali sestaviti razliko t - 3^k.
    if (t >= pot3[k]) t -= pot3[k];
    else t = pot3[k] - t, obrni = ! obrni;
  }

  return true;
}
```

3. Konkordanca

Recimo, da je niz s, čigar pojavitve nas zanimajo, dolg n znakov in da hočemo pri vsaki pojavitvi izpisati še prejšnjih in naslednjih $d = 30$ znakov. Po vhodni datoteki se bomo sprehajali z „oknom“, dolgim $m = n + 2d$ znakov — na sredi okna je torej n znakov, kjer bomo gledali, ali se tam pojavi niz s ali ne, levo in desno od tega pa je še po d znakov, ki jih potrebujemo za izpis. Na vsakem koraku torej preverimo, ali se s pojavlja pri trenutnem položaju okna; če se, ga izpišemo; nato pa v vsakem primeru premaknemo okno za eno mesto naprej — takrat pride na desni v okno nov znak, na levi pa en znak izpade iz okna.

Da bo ta rešitev učinkovita, je koristno za predstavitev okna uporabiti neke vrste krožno tabelo (*ring buffer*): okno bo sicer tabela m znakov, pri čemer pa zdaj za vsak k velja, da k-ti znak vhodne datoteke (kadar je prisoten v oknu) hranimo v tej tabeli na indeksu k mod m. To pomeni, da ko se okno premakne naprej po vhodni datoteki, bo treba znak, ki na novo pride v okno, hraniti na prav tistem indeksu, kjer

smo prej hranili znak, ki je pri tem premiku ravnokar izpadel z okna; v preostanku tabele pa ne bo treba spreminjati ničesar.

Oglejmo si implementacijo takšne rešitve v jeziku C++. Spremenljivka p bo hranila tisti indeks v tabeli okno, kjer gledamo, ali se tam nahaja pojavitev niza s. Pred p je torej v oknu še d znakov levega kontesta, od p naprej pa $n+d$ znakov (dovolj za eno pojavitev s-ja in za d znakov desnega konteksta); okno takó pravzaprav sestavljajo znaki okno$[(p+i) \bmod m]$ za $-d \le i < n+d$. Nekaj pazljivosti je potrebne še na koncu datoteke, kjer morda od p-ja naprej sploh ni več $n+d$ znakov, ker se datoteka konča že prej. Če je od p-ja naprej celo manj kot n znakov, potem gotovo ne bomo našli nobene pojavitve s-ja več in lahko takoj končamo; sicer pa le pazimo pri izpisu desnega konteksta, da bomo namesto manjkajočih znakov napisali presledke.

Za branje iz vhodne datoteke poskrbimo na začetku vsake iteracije glavne zanke, kjer skušamo poskrbeti, da bomo imeli v oknu od p naprej še $n+d$ znakov (manj kot to pa le, če se datoteka tam že konča). Načeloma to pomeni, da po vsakem premiku okna naprej (torej: po vsakem povečanju p-ja za 1) preberemo po en nov znak iz vhodne datoteke, le na začetku izvajanja programa bomo prebrali do $n+2d$ znakov, da napolnimo celo okno.

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

void Konkordanca(const string &s, int d = 30)
{
  // Pripravimo okno z dovolj prostora za en izvod niza s
  // in še d znakov levo in desno od njega.
  int n = s.length();
  int m = 2 * d + n; string okno(m, ' ');

  // p je trenutni položaj v oknu, kjer preverjamo, ali se tam začne
  // pojavitev s-ja. „stVeljavnih“ je število veljavnih znakov v oknu od p naprej.
  int p = 0, stVeljavnih = 0;
  while (true)
  {
    // Preberimo naslednji znak.
    int c = fgetc(stdin);
    if (c != EOF) {
      okno[(p + stVeljavnih++) % m] = (c == '\n' ? ' ' : c);
      if (stVeljavnih < n + d) continue; }

    // Tu imamo od p-ja naprej bodisi n + d veljavnih znakov
    // bodisi ves preostanek datoteke, če ga je manj kot n + d znakov.
    // Poglejmo, ali se tu začne pojavitev s-ja.
    if (stVeljavnih < n) break;
    int i = 0; while (i < n && s[i] == okno[(p + i) % m]) ++i;

    // Če smo našli pojavitev s-ja, jo izpišimo s kontekstom vred.
    if (i == n) {
      for (int i = -d; i <= n + d; ++i)
        fputc(i >= stVeljavnih ? ' ' : okno[(p + i + m) % m], stdout);
      fputc('\n', stdout); }

    p = (p + 1) % m; --stVeljavnih; // Premaknimo se naprej po oknu.
  }
}
```

To, ali se na sredi okna (pri njegovem trenutnem položaju) nahaja pojavitev niza s, smo preverjali preprosto z zanko, ki primerja istoležne znake v oknu in v s-ju. To bi se dalo seveda še izboljšati s prijemi iz raznih znanih algoritmov, kot so Knuth-Morris-Prattov, Boyer-Mooreov ali Rabin-Karpov, vendar poudarek naše naloge ni na tem.

4. Nedeljiva hramba

Podatek, ki ga želimo shraniti, je dolg štiri bajte, mi pa lahko atomično zapisujemo le po en bajt naenkrat; torej ne moremo zagotoviti, da nas ne bo med shranjevanjem kaj prekinilo, še preden bomo zapisali vse štiri bajte. Če nočemo, da v pomnilniku takrat ostane neka mešanica stare in nove vrednosti, to pomeni, da ne smemo nobenega dela stare vrednosti spremeniti ali povoziti, dokler ni nova vrednost v celoti zapisana v pomnilnik. Koristno je torej, če nove vrednosti ne pišemo čez staro, pač pa nekam drugam; v pomnilniku moramo imeti pripravljen prostor za dve vrednosti hkrati. Eno od njiju hranimo na primer na naslovih od 0 do 3, drugo pa od 4 do 7. Poleg tega moramo nekje — recimo kar na naslovu 8 — hraniti še podatek o tem, katera od obeh vrednosti je novejša.

Podprogram ShraniPodatek bo torej pogledal na naslov 8, katera od vrednosti je novejša, in potem z novim podatkom ne bo povozil nje, pač pa tisto drugo; in šele ko bo novi podatek v celoti shranil, bo na naslovu 8 zapisal, katera vrednost je zdaj novejša. Podprogram PreberiPodatek pa najprej prebere bajt z naslova 8, da vidi, katera vrednost je novejša, in potem prebere vse štiri bajte od tam in jih združi v eno samo 32-bitno celoštevilsko vrednost.

```
void NastaviZacetnoStanje()
{
  ShraniBajt(8, 0);
}

void ShraniPodatek(unsigned int podatek)
{
  // Novi podatek zapišimo na drugo lokacijo od dosedanjega.
  int kam = (PreberiBajt(8) == 0 ? 4 : 0);

  // Zapišimo ga po en bajt naenkrat.
  for (int i = 0; i < 4; ++i, podatek >>= 8)
    ShraniBajt(kam + i, podatek & 0xff);

  // Zapomnimo si, kam smo ga zapisali.
  ShraniBajt(8, kam);
}

unsigned int PreberiPodatek()
{
  // Poglejmo, kje lahko preberemo najnovejši podatek.
  int odKod = PreberiBajt(8);

  // Preberimo ga po en bajt naenkrat.
  unsigned int podatek = 0;
  for (int i = 0; i < 4; ++i)
    podatek = (podatek << 8) | PreberiBajt(odKod + 3 - i);

  return podatek;
}
```

Če uporabnik pokliče PreberiPodatek pred prvim klicem ShraniPodatek, bo dobil neko vrednost, ki je bila od prej pač slučajno v pomnilniku na naslovih od 0 do 3. Še ena možnost bi bila, da bi NastaviZacetnoStanje inicializiral bajt 8 na neko tretjo vrednost (niti 0 niti 4), kar bi PreberiPodatek lahko preveril in v tem primeru javil napako (sprožil izjemo ali kaj podobnega).

5. Prisotnost

Uredimo za začetek predavanja naraščajoče po času konca in jih v tem vrstnem redu oštevilčimo; odslej bomo torej predpostavili, da je $k_1 \leq k_2 \leq \ldots \leq k_n$. Prvi (najzgodnejši) vpis se ne sme zgoditi kasneje kot ob času k_1, saj se kasneje ne bo več mogoče vpisati med prisotne na prvem predavanju. Nobene koristi pa ni od tega, da bi se ta vpis zgodil kaj prej kot ob k_1, saj vsako predavanje, ki bi ga pokrili ob času $t < k_1$, poteka tudi še v času k_1 (kajti če ne bi, bi to pomenilo, da se je končalo pred časom k_1, mi pa smo predavanja uredili po času konca in torej vemo, da se nobeno predavanje ne konča prej kot ob k_1).

Ko smo tako izbrali čas prvega vpisa in z njim pokrili nekaj predavanj (vsaj tisto od z_1 do k_1, morda pa še kakšno drugo), lahko ta predavanja v mislih pobrišemo, saj nam je vseeno, ali jih bodo kasnejši vpisi še kaj pokrivali ali ne; dovolj je že, da so bila pokrita enkrat. Tako nam ostane neka manjša množica predavanj, pri kateri lahko razmišljamo enako kot na začetku: prvi naslednji vpis se mora zgoditi ob najzgodnejšem času, ko se konča kakšno od preostalih predavanj (če bi se zgodil kasneje, bi tisto predavanje zgrešili, če pa bi se zgodil prej, ne bi s tem ničesar pridobili). Ker imamo predavanja urejena po času konca, je dovolj, če poiščemo prvo tako predavanje, ki ima $z_i > k_1$; za ta i potem uporabimo k_i kot čas drugega vpisa. Po tem postopku lahko zdaj nadaljujemo in poiščemo prvo tako predavanje j, ki ima $z_j > k_i$ in za ta j potem uporabimo k_j kot čas tretjega vpisa; in tako naprej, dokler niso pokrita vsa predavanja.

Zapišimo ta postopek še s psevdokodo:

```
uredi tekmovanja naraščajoče po času konca, tako da bo k_1 ≤ k_2 ≤ ... ≤ k_n;
r := 0; v := −∞; (* r je število vpisov, v je čas zadnjega vpisa doslej *)
for i := 1 to n:
  if r = 0 or z_i > v:
    r := r + 1; v := k_i; (* potreben je vpis ob času v *)
return r;
```

Časovna zahtevnost te rešitve je $O(n \log n)$ zaradi urejanja po k_i v prvem koraku. Še ena možnost bi bila, da bi pustili predavanja neurejena in šli po vsakem vpisu z zanko po vseh predavanjih, da bi ugotovili, katero med tistimi, ki se začnejo po tem vpisu, ima najzgodnejši čas konca. Takšna rešitev bi imela časovno zahtevnost $O(n \cdot r)$, kjer je r najmanjše potrebno število vpisov; ta rešitev je lahko boljša od prejšnje, če je r dovolj majhen, v najslabšem primeru pa je seveda r lahko tudi $O(n)$ in bo ta rešitev veliko slabša, reda $O(n^2)$.

REŠITVE NALOG ZA TRETJO SKUPINO SŠ

1. Padalski izlet

Vzemimo za začetek na izlet vse padalce, ki imajo svoj avto, torej ki imajo $a_i \geq 0$; recimo, da je med njimi Z začetnikov, da izkušeni med njimi lahko skočijo z vsega skupaj B začetniki in da oboji skupaj lahko prepeljejo P potnikov:

$$Z = |\{i : a_i \geq 0, b_i < 0\}|, \quad B = \textstyle\sum_{i:a_i\geq 0, b_i \geq 0} b_i, \quad P = \textstyle\sum_{i:a_i \geq 0} a_i.$$

Zdaj lahko izberemo še največ P potnikov. Pri tem lahko na vsakem koraku razmišljamo takole: če je $Z < B$, lahko skočijo vsi dosedanji začetniki in zmogljivosti izkušenih padalcev še niso izkoriščene, zato je bolje vzeti za naslednjega potnika še kakšnega začetnika; če pa je $Z \geq B$, moramo najprej dodati še kakšnega izkušenega potnika, kajti doslej izbrani izkušeni padalci tako ali tako ne morejo poskrbeti za več kot Z začetnikov. Pri drugem primeru izbirajmo izkušene potnike po padajočih b_i, tako da bomo s čim manj izkušenimi potniki lahko poskrbeli za čim več začetnikov. Tako smo dobili naslednji postopek:

while $P > 0$:
  if $Z < B$:
    če ni nobenega začetnika več, končaj;
    sicer vzemi za naslednjega potnika enega od začetnikov;
    $Z := Z + 1$; $P := P - 1$;
  else:
    če ni nobenega izkušenega več, končaj;
    sicer vzemi za naslednjega potnika enega od izkušenih z največjim b_i;
    $B := B + b_i$; $P := P - 1$;

Vidimo lahko, da ta postopek nikoli ne poveča Z-ja tako, da bi bil ta potem večji od B; na koncu postopka lahko torej $Z > B$ velja le v primeru, če je veljala že na začetku. To se torej zgodi takrat, ko smo vsa potniška mesta zapolnili z izkušenimi padalci (ali pa je izkušenih celo zmanjkalo, še preden smo zapolnili vsa potniška mesta), pa jih še vedno ni dovolj, da bi poskrbeli za vse tiste začetnike, ki imajo svoj avto. Naloga pravi, da se začetniki brez mentorja ne morejo udeležiti izleta; torej bomo morali $Z - B$ od naših začetnikov z avtomobili pustiti doma. Zato se bo število potnikov, ki jih lahko prepeljemo, morda kaj zmanjšalo; ali nam bo to povzročilo kakšne težave? Prepričajmo se, da ni tako. Za začetek opazimo, da če je med potniki (ki so v tem scenariju sami izkušeni) kakšen z $b_i = 0$, ni od njega nobene koristi in ga lahko pustimo doma (pravzaprav bi lahko take padalce — ki imajo $b_i = 0$ in $a_i < 0$ — ignorirali že ob branju vhodnih podatkov). V nadaljevanju torej predpostavimo, da imajo vsi potniki $b_i \geq 1$; število potnikov, recimo mu I, je zato $\leq B$. — Če je torej zdaj $Z > B$, uredimo v mislih začetnike z avtomobili po naraščajočih vrednostih a_i. (1) Če je med njimi morda vsaj $Z - B$ takih, ki imajo $a_i = 0$, jih lahko odslovimo, pri čemer se število potnikov, ki jih lahko prepeljemo, nič ne zmanjša, vrednost Z pa se bo s tem izenačila z B in razpored je veljaven. (2) Drugače pa odslovimo vse začetnike z $a_i = 0$; število potnikov, ki jih lahko prepeljemo, se pri tem ne spremeni; Z se zmanjša za število teh odslovljenih začetnikov, vendar pa še vedno ostane večji od B. Zdaj imamo torej Z začetnikov

z $a_i \geq 1$; katerihkoli B od njih bo zmoglo prepeljati vsaj B potnikov, mi pa imamo le $I \leq B$ potnikov, tako da je pravzaprav čisto vseeno, katerih $Z - B$ začetnikov pustimo doma.

Razmislek v prejšnjem odstavku nam je pokazal, da če se naš prej omenjeni postopek konča z $Z > B$, bomo lahko na izlet vzeli največ B začetnikov (tako, da bodo vsi lahko skočili v tandemu s kakšnim izkušenim padalcem); če pa se postopek konča z $Z \leq B$, bomo seveda lahko vzeli vseh tistih Z začetnikov. Tako je torej rezultat, po katerem nas sprašuje naloga, enak $\min\{Z, B\}$ (za vrednosti Z in B ob koncu postopka).

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

int main()
{
  typedef long long int llint;
  llint prostorZaPotnike = 0; // za toliko potnikov je prostora v avtomobilih
  llint stZac = 0; // toliko začetnikov gre na izlet
  llint maxZac = 0; // s toliko začetniki lahko skočijo tisti izkušeni, ki gredo na izlet
  vector<int> izkuseniBrezAvta; // vrednosti b_i izkušenih padalcev brez avtomobila
  llint zacetnikiBrezAvta = 0; // število začetnikov brez avtomobila

  // Preberimo podatke o padalcih. Vse padalce, ki imajo avto, takoj vzemimo na izlet.
  int n; cin >> n;
  for (int i = 0; i < n; ++i)
  {
    int a, b; cin >> a >> b;

    // Če ima avto, ga vzemimo na izlet.
    if (a >= 0) { prostorZaPotnike += a; if (b < 0) ++stZac; else maxZac += b; }

    // Sicer gre na seznam padalcev brez avta.
    else if (b > 0) izkuseniBrezAvta.push_back(b);
    else if (b < 0) ++zacetnikiBrezAvta;
  }

  // Uredimo izkušene padalce brez avta po b_i.
  sort(izkuseniBrezAvta.begin(), izkuseniBrezAvta.end());

  // Izberimo potnike.
  while (prostorZaPotnike > 0)

    if (stZac >= maxZac) {
      // Trenutno je več začetnikov, kot jih lahko skoči z dosedanjimi
      // izkušenimi, zato pošljimo še enega izkušenega.
      if (izkuseniBrezAvta.empty()) break; // Če izkušenih ni več, končajmo.
      --prostorZaPotnike; maxZac += izkuseniBrezAvta.back();
      izkuseniBrezAvta.pop_back(); }

    else {
      // Trenutno je manj začetnikov, kot jih lahko skoči z dosedanjimi
      // izkušenimi, zato pošljimo še kakšnega začetnika.
      llint d = min(zacetnikiBrezAvta, maxZac - stZac);
      if (d <= 0) break; // Če začetnikov ni več, končajmo.
      stZac += d; zacetnikiBrezAvta -= d; prostorZaPotnike -= d; }

  // Izpišimo rezultat.
  cout << min(maxZac, stZac) << endl; return 0;
```

}

Ta rešitev ima časovno zahtevnost $O(n \log n)$, ker mora urediti izkušene padalce brez avtomobila; za potrebe našega tekmovanja je to čisto dovolj dobro, vseeno pa razmislimo še o izboljšavi, ki se urejanju izogne.

Označimo z Z število začetnikov z avtom, z Z' število začetnikov brez avta, z I' število izkušenih brez avta, s P vsoto a_i po vseh padalcih z avtom in z B vsoto b_i po vseh izkušenih padalcih z avtom. Naj bo še B_k vsota vrednosti b_i po tistih k izkušenih padalcih brez avta, ki imajo največje b_i. Če se odločimo izbrati k izkušenih potnikov (za $0 \le k \le \min\{P, I'\}$), bomo lahko izbrali potem še $z(k) := \min\{P - k, Z'\}$ neizkušenih potnikov, končni rezultat pa bo $f(k) := \min\{Z + z(k), B + B_k\}$. Vrednost k bi radi seveda izbrali tako, da bo $f(k)$ čim večja. Ker je $Z + z(k)$ padajoča funkcija k-ja, $B + B_k$ pa naraščajoča, je pri majhnih k vrednost $f(k)$ enaka $B + B_k$ in je zato f tam tudi naraščajoča, pri velikih k pa je $f(k)$ enaka $Z + z(k)$ in je zato tam tudi ona padajoča. Svoj maksimum zato doseže $f(k)$ tam, kjer preide iz rastočega režima v padajočega, se pravi tam, kjer se funkciji $Z + z(k)$ in $B + B_k$ sekata. Dovolj je torej, če poiščemo največji k, pri katerem je $B + B_k \le Z + z(k)$, in najmanjši k, pri katerem je $B + B_k \ge Z + z(k)$.

To lahko naredimo z bisekcijo po k; pomembna podrobnost pa je naslednja: ko smo pri bisekciji omejeni na neki interval možnih k-jev, recimo od k_1 do k_2, vzdržujmo pri tem vsoto B_{k_1} ter seznam b_i-jev za $k_1 \le i \le k_2$ — natančneje rečeno, to so tisti izkušeni padalci brez avta, ki bi bili na indeksih od k_1 do k_2, če bi b_i-je vseh takih padalcev uredili padajoče. Mi pa ne bomo imeli urejenega celega seznama, pač pa le elemente, ki bi bili v takem seznamu na indeksih od k_1 do k_2, pri čemer jih ne bomo imeli nujno urejenih v kakšnem posebnem vrstnem redu.

Ko moramo nato pri trenutnem koraku bisekcije preizkusiti naslednjega kandidata za k, to je $k = \lfloor (k_1 + k_2)/2 \rfloor$, lahko poiščemo ustrezni b_i s quickselectom po našem seznamu v $O(k_2 - k_1)$ časa; pri tem tudi razdelimo seznam na levo in desno polovico, od katerih nam bo ena prišla prav v naslednjem koraku bisekcije; in ko vrednosti B_{k_1} prištejemo vse elemente leve polovice, dobimo ravno B_k, ki ga potrebujemo v trenutnem koraku bisekcije. Ker se širina opazovanega intervala, $k_2 - k_1$, na vsakem koraku bisekcije razpolovi, je vsota teh širin po vseh korakih le $O(n)$ in zato je tudi časovna zahtevnost te rešitve le $O(n)$.

2. Ulične luči

Ko se a povečuje, se tudi interval, ki ga osvetljuje posamezna luč, le povečuje; če je ulica v celoti osvetljena pri nekem a, je tudi pri vsakem večjem a; in če pri nekem a ni v celoti osvetljena, potem ni v celoti osvetljena tudi pri nobenem manjšem. To pomeni, da lahko najmanjši primerni a poiščemo z bisekcijo; začnemo lahko na primer s tem, da vemo, da je $a = 0$ premajhen, $a = m$ pa je gotovo dovolj velik.

Ko moramo pri posameznem a preveriti, ali bi bila ulica pri tem a v celoti osvetljena, lahko razmišljamo takole. Za začetek je koristno luči urediti naraščajoče po p_i; odslej bomo torej predpostavili, da je $0 \le p_1 < p_2 < \ldots < p_n \le m$. Luč i osvetljuje interval od $\ell_i(a) := p_i - ac_i$ do $d_i(a) := p_i + ac_i$; najbolj desna točka, ki jo doseže svetloba luči od 1 do i, je potem $D_i(a) := \max_{1 \le j \le i} d_j(a)$; najbolj leva točka, ki jo doseže svetloba luči od i do n, pa je $L_i(a) := \min_{i \le j \le n} \ell_j(a)$. Naša

ulica je interval $[0, m]$, ki si ga lahko predstavljamo razbitega na krajše podintervale: $[0, p_1], [p_1, p_2], \ldots, [p_{n-1}, p_n], [p_n, m]$. Prvi od teh je v celoti osvetljen, če je $L_1(a) \leq 0$; zadnji je v celoti osvetljen, če je $D_n(a) \geq m$; vmes pa velja, da je interval $[p_{i-1}, p_i]$ v celoti osvetljen, če je $D_{i-1}(a) \geq L_i(a)$. Levo od tega intervala so namreč luči $1, \ldots, i-1$ in svetloba, ki prihaja desno od njih, se mora srečati s svetlobo, ki prihaja levo od luči $i, \ldots n$, ki ležijo desno od našega intervala.

Vrednosti $D_i(a)$ lahko računamo v zanki po naraščajočih i in jih shranjujemo v neko tabelo: $D_0(a) = 0$, $D_i(a) = \max\{D_{i-1}(a), d_i(a)\}$. Podobno lahko potem računamo tudi vrednosti $L_i(a)$ po padajočih i, pri vsakem i pa preberemo še prej shranjeno vrednost $D_{i-1}(a)$ in preverimo pogoj $D_{i-1}(a) \geq L_i(a)$.

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

int main()
{
  // Preberimo vhodne podatke.
  int n, m; scanf("%d %d", &n, &m);
  struct Luc { int p, c; };
  vector<Luc> luci(n);
  for (auto &L : luci) scanf("%d %d", &L.p, &L.c);

  // Uredimo luči od leve proti desni.
  sort(luci.begin(), luci.end(), [] (const auto x, const auto y) { return x.p < y.p; });

  // Najmanjši a poiščimo z bisekcijo.
  int a1 = 0, a2 = m; vector<long long int> desno(n + 1);
  while (a2 - a1 > 1)
  {
    // a1 je premajhen, a2 je dovolj velik; poskusimo na sredi med njima.
    long long int a = (a1 + a2) / 2;

    // desno[i] = desni rob svetlobe prvih i luči
    desno[0] = 0;
    for (int i = 0; i < n; ++i) desno[i + 1] = max(desno[i], luci[i].p + a * luci[i].c);

    // Če svetloba ne doseže desnega krajišča cele ulice, je a premajhen.
    bool ok = (desno[n] >= m);

    // Poglejmo, kako se širi svetloba proti levi.
    long long int levo = m;
    for (int i = n - 1; i >= 0 && ok; --i) {
      levo = min(levo, luci[i].p - a * luci[i].c);
      // Zdaj je „levo“ = levi rob svetlobe luči od i do n - 1.
      // Interval od luči i - 1 do i je pokrit, če se ta svetloba sreča s tisto,
      // ki prihaja desno od luči 0, ... , i - 1. (Pri i = 0 ta pogoj preveri,
      // če svetloba z desne doseže tudi levo krajišče cele ulice.)
      ok = (desno[i] >= levo); }

    // Pripravimo se na naslednjo iteracijo bisekcije.
    if (ok) a2 = a; else a1 = a;
  }

  printf("%d\n", a2); return 0; // Izpišimo rezultat.
}
```

Pri vsakem koraku bisekcije imamo $O(n)$ dela s preverjanjem, ali je ulica v celoti osvetljena, zato ima ta rešitev časovno zahtevnost $O(n \log m)$.

Osvetljenost bi lahko preverili tudi tako, da bi intervale $[p_i - a \cdot c_i, p_i + a \cdot c_i]$ zložili v seznam in jih uredili po začetnem krajišču; potem ne bi bilo težko preveriti, ali bi bila osvetljena celotna ulica:

```
vector<pair<int, int>> intevali;
⋮
sort(intervali.begin(), intervali.end());
int svetloDo = 0;
for (auto [L, D]: intervali)
  // Dosedanji intevali osvetljujejo ulico do x-koordinate svetloDo.
  // Če se trenutni interval začne desno od tam, je vmes neosvetljeno območje.
  if (L > svetloDo) break;
  // Sicer nam trenutni interval morda osvetli ulico še dlje v desno.
  else svetloDo = max(svetloDo, D);
// Preverimo, ali je ulica osvetljena do konca.
if (svetloDo < m) a1 = a; else a2 = a;
```

Ta rešitev je sicer slabša od prejšnje, kajti zaradi urejanja seznama intervalov nam vsak korak bisekcije vzame $O(n \log n)$ časa, celotna rešitev pa ima zato časovno zahtevnost $O(n(\log n)(\log m))$. Pri največjih testnih primerih bo ta rešitev prekoračila časovno omejitev, razen če smo zelo pazljivi pri implementaciji in vključimo v rešitev kakšno hevristiko, ki prihrani nekaj časa, na primer eno od naslednjih dveh:

- za začetno zgornjo mejo bisekcije je koristno namesto $a = m$ vzeti maksimum vrednosti p_1/c_1, $(m - p_n)/c_m$ in $\max_i(p_i - p_{i-1})/(c_{i-1} + c_i)$, kajti to je a, pri katerem bo vsak interval ulice v celoti osvetljen že zgolj zaradi luči, ki stojita na njegovih krajiščih;

- za interval, ki ga osvetljuje luč i, je koristno vzeti $[\max\{0, p_i - ac_i\}, \min\{m, p_i + ac_i\}]$, torej odrezati tisto, kar bi sicer segalo čez rob ulice. Če se nam potem pri več lučeh pojavijo intervali z levim krajiščem 0, obdržimo od njih le tistega, ki sega najdlje na desno; podobno pa tudi med intervali z desnim krajiščem m obdržimo le tistega, ki sega najdlje v levo. Tako lahko zmanjšamo število intervalov, ki jih bo treba urejati.

Od druge hevristike je korist predvsem pri velikih a, kjer bi drugače veliko intervalov štrlelo čez rob ulice; prva hevristika pa se izogne velikim a tako, da že na začetku zmanjša zgornjo mejo bisekcije. Zato ni prave koristi od tega, da bi v rešitvi uporabili obe hevristiki hkrati (čeprav ni narobe, če to naredimo).

Kot zanimivost si oglejmo še eno rešitev, ki je sicer malo bolj zapletena, vendar za razliko od prejšnjih dveh njena časovna zahtevnost ni odvisna od m. Spomnimo se, da je interval $[p_{i-1}, p_i]$ v celoti osvetljen, če velja $D_{i-1}(a) \geq L_i(a)$. (S tem razmislekom lahko pokrijemo tudi intervala $[0, p_1]$ in $[p_n, m]$, če v mislih vpeljemo $p_0 = 0$, $p_{n+1} = m$, $D_0(a) = 0$, $L_{n+1}(a) = m$. Zato intervalov $[0, p_1]$ in $[p_n, m]$ ne bomo obravnavali posebej, pač pa moramo imeti v mislih, da nas bodo intervali $[p_{i-1}, p_i]$ zanimali za $i = 1, 2, \ldots, n + 1$.) Najmanjši a, pri katerem je interval v celoti osvetljen, je torej tisti, pri katerem velja enakost: $D_{i-1}(a) = L_i(a)$. Če rešimo takšno enačbo za vsak i in potem vrnemo maksimum tako dobljenih a-jev, bo pri njem osvetljena celotna ulica, prav po tem pa naloga sprašuje.

Funkcija $D_i(a)$ je definirana kot maksimum nekaj naraščajočih linearnih funkcij (namreč $d_j(a)$ za $1 \le j \le i$), zato je D_i po obliki odsekoma linearna funkcija, v kateri vsak odsek prihaja iz ene od funkcij d_j in vsak naslednji odsek (če jih gledamo po naraščajočih a) narašča hitreje od prejšnjega (D_i je torej konveksna in naraščajoča). Takšno funkcijo lahko predstavimo tako, da imamo v neki uravnoteženi drevesasti strukturi (npr. rdeče-črnem drevesu) naštete njene odseke, urejene naraščajoče po a (in s tem tudi naraščajoče po naklonu). Če hočemo potem predelati D_i v D_{i+1}, ki se razlikuje od D_i le po tem, da ima v maksimumu, s katerim je definirana, še eno linearno funkcijo več (namreč funkcijo d_{i+1}), vidimo, da se lahko v funkciji zaradi tega pojavi nov odsek (z enakim naklonom kot funkcija d_{i+1}) in da zaradi tega lahko nekaj obstoječih odsekov iz funkcije izgine, eden ali dva (na začetku in koncu novega odseka) pa se morda skrajšata. Ker so odseki urejeni po naklonu (in bo to veljalo tudi v D_{i+1}), ni težko poiskati, kje v drevesu bo do te spremembe prišlo. Tako lahko začnemo z D_0 (ki je čisto navadna linearna funkcija) in jo postopoma predelamo v D_1, D_2 in tako naprej do D_n; vsakič pride v drevo največ en nov odsek, največ dva se skrajšata in nekaj jih izpade; ker je dodajanj novih odsekov največ n, je tudi izpadov največ n, skrajševanj pa $2n$; tako imamo $O(n)$ operacij na drevesu, vsaka pa vzame $O(\log n)$ časa, skupaj $O(n \log n)$.[6]

Podobno lahko razmišljamo tudi pri $L_i(a)$, le da so te funkcije definirane kot minimum nekaj padajočih linearnih funkcij (namreč $\ell_j(a)$ za $i \le j \le n$), zato so L_i padajoče in konkavne odsekoma linearne funkcije. Tu lahko začnemo pri L_{n+1} (ki je linearna funkcija) in jo postopoma predelujemo v L_n, L_{n-1} in tako naprej do L_1, za kar spet porabimo skupaj $O(n \log n)$ časa. Vendar pa si tokrat ob vsaki predelavi zapomnimo, kaj točno se je v drevesu spremenilo in kakšno je bilo stanje pred spremembo (ker je bilo sprememb $O(n)$, nam bo hramba teh podatkov vzela $O(n)$ prostora); tako bomo lahko kasneje te spremembe tudi „razveljavili“ oz. jih izvedli v obratni smeri: to nam bo torej omogočilo, da bomo začeli pri L_1 in ga nato postopoma predelovali v L_2, L_3 in tako naprej do L_{n+1}.

V nadaljevanju nam bo prišel prav naslednji postopek: recimo, da gledamo le en odsek funkcije D_{i-1} in nas zanima, ali se (in kje se) tam ta funkcija seka s funkcijo L_i. Opazovani odsek funkcije D_{i-1} pokriva neki interval vrednosti a, recimo $[a_1, a_2]$, funkcija L_i pa ima na tem intervalu morda več odsekov. Razmišljajmo takole:

$u :=$ koren rdeče-črnega drevesa, v katerem hranimo funkcijo L_i;
while $u \neq$ NIL:
  vozlišče u predstavlja neki odsek funkcije L_i;
  naj bo $[a_3, a_4]$ tisti interval a-jev, ki ga pokriva ta odsek;
  (* *Morda leži ta interval v celoti levo ali desno od* $[a_1, a_2]$. *)
  if $a_2 \le a_3$ **then** $u :=$ u-jev desni otrok; **continue**;
  if $a_4 \le a_1$ **then** $u :=$ u-jev levi otrok; **continue**;
  (* *Sicer se intervala vsaj delno prekrivata, recimo na* $[\lambda, \delta]$. *)
  $\lambda := \max\{a_1, a_3\}$; $\delta := \min\{a_2, a_4\}$;
  (* *Spomnimo se, da je* D_{i-1} *naraščajoča,* L_i *pa padajoča.*
  Če je D_{i-1} *že na levem krajišču nad* L_i*, bo desno od tam še bolj nad njo;*

[6] S takšnimi odsekoma linearnimi funkcijami smo se v naših biltenih pred leti že srečali; gl. *Bilten* 2017, str. 116.

presečišče mora torej ležati bolj levo. *)
if $D_{i-1}(\lambda) > L_i(\lambda)$ **then** $u := u$-jev levi otrok; **continue**;

(* *Podobno, če je* D_{i-1} *na desnem krajišču pod* L_i*, bo levo od tam še bolj pod njo; presečišče mora torej ležati bolj desno.* *)
if $D_{i-1}(\delta) < L_i(\delta)$ **then** $u := u$-jev levi otrok; **continue**;

sicer vemo, da je D_{i-1} na levem krajišču pod L_i, na desnem pa nad L_i;
ker sta funkciji zvezni, se morata torej sekati ravno na tem intervalu;
izračunajmo presek teh dveh daljic in ga vrnimo;

Če se zanka ustavi, ne da bi našla presek (pač pa zato, ker se je poskušala z u-jem premakniti v otroka, ki ga prejšnji u sploh ni imel), je to znak, da se opazovani odsek funkcije D_{i-1} sploh ne seka s funkcijo L_i, pač pa leži v celoti nad ali v celoti pod njo.

Ker ima L_i kvečjemu n odsekov, je globina drevesa le $O(\log n)$, zato ima ta postopek časovno zahtevnost $O(\log n)$. Podobno bi se dalo razmišljati tudi, če imamo en odsek funkcije L_i in nas zanima, ali se tam L_i kje seka s funkcijo D_{i-1}.

Recimo torej zdaj, da imamo pred seboj D_0 in L_1 in rešujemo enačbo $D_0(a) = L_1(a)$. Ker ima D_0 le en odsek, jo lahko rešimo tako, da poiščemo presek med njim in L_1 po ravnokar opisanem postopku.

Recimo nato, da smo pravkar rešili enačbo $D_{i-1}(a) = L_i(a)$, v nadaljevanju pa bi si želeli rešiti enačbo $D_i(a) = L_{i+1}(a)$. Funkciji $D_{i-1}(a)$ in $D_i(a)$ pri večini a-jev povsem sovpadata; razlikujeta se le na največ enem intervalu, namreč tistem, kjer ima D_i novi odsek (če ga sploh ima), ki je vanjo prišel zaradi funkcije d_i. Podobno se tudi $L_i(a)$ in $L_{i+1}(a)$ razlikujeta na največ enem intervalu a-jev, namreč tam, kjer ima (če ga ima) L_i novi odsek, ki je vanjo prišel zaradi funkcije ℓ_i. Tako imamo torej dva intervala, enega, kjer se razlikujeta D_{i-1} in D_i, ter drugega, kjer se razlikujeta L_i in L_{i+1}. (1) Če je rešitev enačbe $D_{i-1}(a) = L_i(a)$ nastopila zunaj teh dveh intervalov, je to še vedno tudi rešitev enačbe $D_i(a) = L_{i+1}(a)$. (2) Sicer pa lahko za vsakega od teh dveh intervalov pogledamo, kje na njem velja $D_i(a) = L_{i+1}(a)$. (2.1) Na tistem intervalu, kjer ima D_i novi odsek (iz d_i), lahko uporabimo malo prej opisani postopek, da poiščemo njegov morebitni presek s funkcijo L_{i+1}. (2.2) Na tistem intervalu, kjer ima L_i novi odsek (iz ℓ_i), je v L_{i+1} lahko več odsekov. Za vsakega od njih uporabimo prej opisani postopek, da poiščemo njegov morebitni presek s funkcijo D_i.

Razmislek iz gornjega odstavka moramo ponoviti za $i = 1, 2, \ldots, n$, da rešimo vse enačbe, ki nas zanimajo; kolikokrat bomo vsega skupaj izvedli postopek za iskanje preseka ene odsekoma linearne funkcije z enim odsekom druge funkcije? V vsakem izvajanju točke (2.1) največ enkrat; v točki (2.2) pa se sicer pri posameznem i lahko izvede ta postopek večkrat, ker ima L_{i+1} lahko več odsekov tam, kjer ima L_i le enega. Toda to so odseki, ki so bili pobrisani, ko smo računali L_i iz L_{i+1}, torej so morali biti nekoč prej dodani v drevo; vseh dodajanj skupaj pa je $O(n)$, torej se tudi v točki (2.2) vsega skupaj izvede le $O(n)$ računanj preseka. Vsako računanje preseka pa, kot smo videli, zahteva en spust po drevesu in torej vzame $O(\log n)$ časa; časovna zahtevnost naše rešitve je torej $O(n \log n)$.

3. Špijonaža

Hierarhijo vohunov si lahko predstavljamo kot drevo s korenom 1; *poddrevo* vohuna u tvorijo on in vsi njegovi posredno ali neposredno podrejeni. Označimo s $f(u)$ najmanjše število izvodov, ki morajo hkrati obstajati v u-jevem poddrevesu od trenutka, ko u prejme svoj izvod, do trenutka, ko so vsi vohuni v u-jevem poddrevesu že videli dokument. Naloga torej sprašuje po $f(1)$. Funkcijo f lahko računamo z rekurzivnim razmislekom. Če u sploh nima neposredno podrejenih (torej če je list drevesa), je $f(u) = 1$, saj lahko u uniči svoj izvod takoj po tem, ko ga je prejel.

Če pa u ima neposredno podrejene (torej če je u notranje vozlišče), razmišljajmo takole: ko izroči u nekemu svojemu neposredno podrejenemu, recimo v-ju, kopijo dokumenta, bo sčasoma moralo biti v v-jevem poddrevesu hkrati prisotnih $f(v)$ izvodov dokumenta, preden bodo vsi vohuni v tem poddrevesu videli dokument; s tem pa bodo ti izvodi seveda prisotni tudi v u-jevem poddrevesu. Medtem bo v u-jevem poddrevesu prisoten vsaj še izvod, ki ga bo imel u sam, razen če je bil v zadnji u-jev neposredno podrejeni, ki je prejel svojo kopijo dokumenta; v tem slednjem primeru lahko u uniči svoj izvod, čim v prejme kopijo.

Tako torej vidimo, da za vsakega u-ju neposredno podrejenega v razen za zadnjega, ki prejme svoj izvod, velja $f(u) \geq 1 + f(v)$. Za zadnjega neposredno podrejenega, ki prejme svoj izvod — recimo mu z_u — pa velja $f(u) \geq f(z_u)$, vendar tudi $f(u) \geq 2$, kajti v trenutku, ko je z_u že dobil svoj izvod dokumenta, u pa svojega še ni uničil, sta obstajala vsaj tadva izvoda hkrati (ta robni primer je pomemben le, če ima u enega samega neposredno podrejenega, ta pa je list in ima $f(z_u) = 1$; takrat moramo paziti, da za $f(u)$ dobimo 2 in ne 1).

Najmanjša možna vrednost za $f(u)$ je torej $\max\{2, f(z_u), 1 + \max_v f(v)\}$. Pri tem gre $\max_v$ po vseh u-jevih neposredno podrejenih razen po z_u. Slednji je torej edini, pri katerem se lahko izognemo potrebi po tem, da bi njegovi $f(\cdot)$ prišteli 1; zato je smiselno za z_u vzeti tistega neposredno podrejenega, ki ima največjo vrednost $f(z_u)$.

Zdaj imamo vse, kar potrebujemo za rekurzivno računanje funkcije f. Ko smo pri vohunu u, najprej z vgnezdenimi rekurzivnimi klici obdelamo vse njegove neposredno podrejene; zapomnimo si, pri katerem je nastopila največja vrednost $f(\cdot)$ — to bo naš z_u; poleg tega pa si zapomnimo še največjo vrednost $f(\cdot)$ po ostalih neposredno podrejenih — to bo naš $\max_v f(v)$. Ko imamo to dvoje, lahko izračunamo $f(u)$ po prej omenjeni formuli, poleg tega pa si zapomnimo tudi z_u, ki bo prišel prav pri izpisu zaporedja korakov.

Naloga namreč zahteva, da izpišemo ne le $f(1)$, pač pa tudi zaporedje korakov, s katerim lahko vsi vohuni vidijo dokument, ne da bi kdaj obstajalo več kot $f(1)$ izvodov naenkrat. Tudi za ta izpis lahko poskrbimo z rekurzijo. Ko smo pri vohunu u, najprej obdelamo vse njegove neposredno podrejene razen z_u; vsak neposredno podrejeni dobi svoj izvod dokumenta, nato pa z rekurzivnim klicem obdelamo njegovo poddrevo (v okviru tega vsi vohuni v tem poddrevesu svoje izvode prej ali slej tudi uničijo). Nazadnje dobi svoj izvod še z_u, potem u svoj izvod uniči in z rekurzivnim klicem obdelamo še z_u-jevo poddrevo. (Pazimo seveda tudi na robni primer: če u nima podrejenih, vohun z_u sploh ne obstaja.)

```
#include <cstdio>
#include <vector>
```

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

struct Vohun
{
  vector<int> podrejeni; // seznam neposredno podrejenih
  int f = −1; // najboljši rezultat za poddrevo, ki ga tvorijo u in njegovi podrejeni
  int z = −1; // neposredno podrejeni z največjim f
};
vector<Vohun> vohuni;

// Izračuna f po celem u-jevem poddrevesu in vrne f(u).
int Izracunaj(int u)
{
  // Rekurzivno obdelajmo vse u-jeve neposredno podrejene
  // in si zapomnimo dva z največjim f.
  auto &U = vohuni[u]; int f1 = −1, f2 = −1;
  for (int v : U.podrejeni)
    if (int fv = Izracunaj(v); fv > f1) f2 = f1, f1 = fv, U.z = v;
    else if (fv > f2) f2 = fv;
  // Izračunajmo rezultat tudi za u.
  return U.f = (U.z < 0) ? 1 : max(max(f1, 2), 1 + f2);
}

// Izpiše vse korake v u-jevem poddrevesu od trenutka, ko u prejme svoj izvod.
void Izpisi(int u)
{
  // Vohun u je prejel svoj izvod dokumenta. Najprej rekurzivno obdelajmo
  // vse podrejene razen z.
  auto &U = vohuni[u];
  for (int v : U.podrejeni) if (v != U.z) {
    printf("1 %d\n", v + 1); // Podrejeni v prejme svoj izvod.
    Izpisi(v); } // Rekurzivno obdelajmo v-jevo poddrevo.
  // Končno tudi podrejeni z prejme svoj izvod.
  if (U.z >= 0) printf("1 %d\n", U.z + 1);
  // Zdaj lahko u svoj izvod uniči.
  printf("2 %d\n", u + 1);
  // Rekurzivno obdelajmo poddrevo vohuna z.
  if (U.z >= 0) Izpisi(U.z);
}

int main()
{
  // Preberimo vhodne podatke.
  int n; scanf("%d", &n); vohuni.resize(n);
  for (int u = 1; u < n; ++u) {
    int p; scanf("%d", &p);
    vohuni[−−p].podrejeni.emplace_back(u); }
  // Izračunajmo rezultate in jih izpišimo.
  printf("%d %d\n", 2 * n − 1, Izracunaj(0));
  Izpisi(0); return 0;
}
```

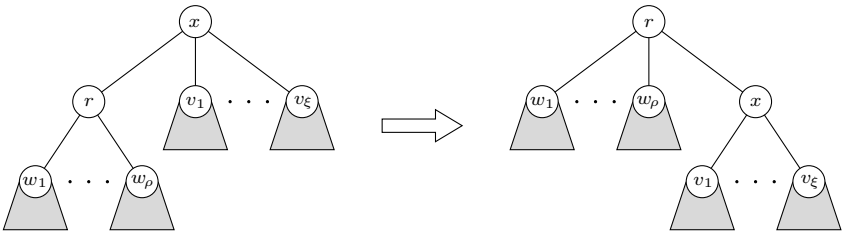
Kot zanimivost razmislimo še, kaj lahko povemo o zvezi med številom vohunov n in številom hkrati obstoječih izvodov dokumenta, torej vrednostjo $f(1)$. Naj bo T_k

najmanjše drevo (torej tako z najmanj vozlišči), pri katerem je $f(1) = k$. Robni primeri so preprosti: T_0 je prazno drevo (z $n = 0$ vozlišči); T_1 je eno samo vozlišče; T_2 je drevo, v katerem ima koren le eno poddrevo, to pa je oblike T_1. Kaj pa T_k za $k \geq 3$? Če ima koren (vozlišče 1) le enega otroka, je ta otrok z_1 in imamo $f(1) = \max\{2, z_1\}$, torej je lahko $f(1) = k$ le, če je tudi $f(z_1) = k$, potem pa naše drevo gotovo ne bo najmanjše s $f(1) = k$, saj bi enak rezultat dobili tudi, če bi se omejili samo na poddrevo s korenom v z_1. Naš koren mora torej imeti vsaj dva otroka; eden bo z_1, drugemu pa recimo v in imamo $f(1) = \max\{2, f(z_1), 1 + f(v)\}$. Pri tem je z_1 po definiciji tisti otrok, ki ima največjo vrednost $f(\cdot)$, torej je $f(z_1) \geq f(v)$. Ker bi radi dobili $f(1) = k$ (in je $k \geq 3$), mora biti torej bodisi $f(z_1) = k$ (toda potem naše drevo spet ne bo najmanjše s $f(1) = k$) bodisi $f(v) = k - 1$; v tem slednjem primeru pa je smiselno potem vzeti tudi $f(z_1) = k - 1$, da bo poddrevo s korenom pri z_1 čim manjše, ob tem pa še vedno izpolnjevalo pogoj $f(z_1) \geq f(v)$. Tako torej vidimo, da ima v drevesu T_k (za $k \geq 3$) koren dve poddrevesi oblike T_{k-1}. Načeloma bi lahko imel koren poleg teh dveh še kakšnega dodatnega otroka, recimo w s $f(w) \leq k - 1$, vendar potem drevo gotovo ne bi bilo najmanjše s $f(1) = k$.

Za najmanjše število vohunov, pri katerem se pojavi potreba po k hkrati obstoječih izvodih dokumenta, smo tako dobili: $|T_0| = 0$; $|T_1| = 1$; $|T_2| = 2$; in za $k \geq 3$ nato $|T_k| = 1 + 2|T_{k-1}|$, iz česar dobimo $|T_k| = 3 \cdot 2^{k-2} - 1$. V splošnem torej, če imamo drevo z n vohuni in $f(1) = k$, vemo, da velja $n \geq |T_k| = 3 \cdot 2^{k-2} - 1$, zato pa $k \leq \log_2(4(n+1)/3)$. Pri n vohunih torej gotovo obstaja vrstni red kopiranja in uničevanja izvodov, pri katerem nikoli ne obstaja več kot $\log_2(4(n+1)/3)$ izvodov hkrati.

Razmislimo zdaj še o težji različici naloge, ki jo omenja opomba pod črto v besedilu naloge. Tu si torej lahko izberemo vohuna r, ki bo na začetku prejel dokument, nato pa se lahko dokument širi tudi gor po drevesu in ne le navzdol kot pri prvotni različici naloge. Vprašanje je, kateri r izbrati, da bo mogoče razširiti dokument po preostanku drevesa in bo pri tem obstajalo čim manj izvodov hkrati.

Naslednja slika kaže, kako lahko premaknemo r en nivo višje v drevesu in se pri tem rešitev nič ne spremeni (x, ki je bil prej r-jev nadrejeni, je zdaj njegov podrejeni, vendar to nič ne vpliva na dejstvo, da lahko r izroči kopijo dokumenta x-u):



Takšni operaciji, pri kateri neko vozlišče zamenja mesto s svojim staršem v drevesu, pravimo tudi *rotacija*. Z rotacijami lahko sčasoma premaknemo r v koren drevesa, nato pa lahko s postopkom za prvotno različico naloge izračunamo, največ koliko dokumentov bo moralo obstajati hkrati.

Toda to nas bo načeloma zanimalo za vsak r, ne bi pa želeli zato n-krat pregledovati celega drevesa. Recimo, da smo nalogo že rešili za primer, ko je v korenu x (kot

na levi sliki zgoraj), torej za vsako vozlišče u poznamo vrednost $f(u)$; in recimo, da potem premaknemo v koren vozlišče r, ki je bilo prej neposredno podrejeno x-u, zdaj pa mu je neposredno nadrejeno (kot na desni sliki zgoraj); nove vrednosti funkcije f označimo s $f'(u)$, da jih bomo lažje ločili od starih. Razmislimo, kakšna je povezava med novimi in starimi vrednostmi. Spomnimo se, da je funkcija f definirana od spodaj gor po drevesu, torej je njena vrednost v nekem vozlišču odvisna le od tega, kar se dogaja v njegovem poddrevesu. Zato se za vozlišča v_i, w_i in njihove potomce ne spremeni nič, saj se tista poddrevesa niso nič spremenila; tam povsod velja $f'(u) = f(u)$.

Pri vozlišču x je bila prej $f(x)$ odvisna od dveh največjih vrednosti izmed $f(r)$, $f(v_1), \ldots, f(v_\xi)$, po novem pa je $f'(x)$ odvisna od dveh največjih vrednosti izmed $f'(v_1), \ldots, f'(v_\xi)$, kar pa je isto kot $f(v_1), \ldots, f(v_\xi)$. Koristno je torej, če si takrat, ko imamo x v korenu, pri izračunu $f(x)$ zapomnimo *tri* največje izmed vrednosti $f(r), f(v_1), \ldots, f(v_\xi)$ (skupaj seveda s tem, pri katerih x-ovih neposredno podrejenih so bile dosežene; vse to si zapomnimo v zapisu, ki ga bomo označili z $M[x]$). Če je ena od njih $f(r)$, bomo še vedno imeli dve največji izmed preostalih, to pa je tudi vse, kar potrebujemo za hiter izračun $f'(x)$.

Ostane še točka r, kjer je bila prej $f(r)$ odvisna od dveh največjih vrednosti izmed $f(w_1), \ldots, f(w_\rho)$, zdaj pa je $f'(r)$ odvisna od dveh največjih vrednosti izmed $f'(x), f'(w_1), \ldots, f'(w_\rho)$, kar pa je isto kot $f'(x), f(w_1), \ldots, f(w_\rho)$. Ko pride r prvič v koren, si lahko privoščimo iti po vseh teh vrednostih; tri največje si zapomnimo (to je naš $M[r]$, ki bo, kot smo videli v prejšnjem odstavku, prišel prav kasneje, ko bo v koren prišlo kakšno drugo vozlišče, r pa se bo spustil en nivo nižje), iz dveh največjih pa izračunajmo $f'(r)$.

Kot smo videli, nas bo na koncu zanimalo, katero vozlišče r ima najmanjšo vrednost $f(r)$ takrat, ko je v korenu drevesa. Drevo moramo torej z rotacijami preoblikovati tako dolgo, da bo vsako vozlišče nekoč prišlo v koren. Če si ogledamo drevo v začetnem stanju, s korenom 1, kakor smo ga dobili v vhodnih podatkih, lahko od tam preiščemo drevo v globino, pa nam bo pri tem nastal sprehod, ki obišče vsa vozlišča in ga lahko uporabimo tudi kot vrstni red rotacij: na vsakem koraku premaknemo v koren drevesa tisto vozlišče, ki je naslednje po vrsti v našem sprehodu. Zapišimo tako dobljeni postopek še s psevdokodo:

podprogram IskanjeVGlobino(vozlišče u, seznam L):
  za vsakega u-jevega otroka v:
    dodaj v na konec L;
    IskanjeVGlobino(v, L);
    dodaj u na konec L;

(* *Pripravimo sprehod, ki obišče vsa vozlišča.* *)
$L :=$ prazen seznam; IskanjeVGlobino(L);

reši nalogo v prvotni obliki, s korenom 1, in si zapomni $f[u]$ vsakega vozlišča;
$x := 1$; za koren x izračunaj $M[x]$;
$r^\star := x$; $f^\star := f[x]$; (* *Najboljša rešitev doslej.* *)

za vsako točko r iz seznama L (po vrsti, kot so v seznamu):
  (* *r je trenutno otrok korena x; izvedimo v mislih rotacijo,*
    s katero pride r v koren. *)

izračunaj novo $f[x]$ iz $M[x]$, pri čemer zdaj upoštevaj,
  da r ni več x-ov otrok (pač pa je njegov starš);
če je r zdaj prvič v korenu:
  zdaj poznamo $f[\cdot]$ za vse r-jeve otroke, tudi za x
    (ki je pravkar postal r-jev otrok);
  iz tega izračunaj $M[r]$;
izračunaj novo $f[r]$ iz $M[r]$;
$x := r$; (* *Zapomnimo si novi koren.* *)
if $f[x] > f^\star$ **then** $r^\star := x$, $f^\star := f[x]$;

Ob koncu tega postopka nam $r^\star$ pove, kateri vohun mora kot prvi prejeti dokument, $f^\star$ pa je največje število hkrati obstoječih dokumentov (pri najboljšem možnem scenariju širjenja dokumenta).

Kakšna je časovna zahtevnost tega postopka? Drevo z n vozlišči ima $n-1$ povezav in naš sprehod gre po vsaki povezavi dvakrat, najprej dol in nato gor; zato imamo $O(n)$ dela s pripravo seznama L na začetku, nato pa ima tudi glavna zanka $O(n)$ iteracij. Vsi koraki v posamezni iteraciji glavne zanke vzamejo po $O(1)$ časa, razen izračuna $M[r]$ iz vrednosti $f[\cdot]$ za vse r-jeve otroke; če ima r recimo n_r otrok, nam ta izračun vzame $O(n_r)$ časa. Na srečo ga moramo izvesti za vsak r le enkrat, namreč ko r prvič pride v koren drevesa, po tistem pa si lahko $M[r]$ zapomnimo. Tako imamo s temi izračuni vsega skupaj le $O(\sum_r n_r) = O(n)$ dela. Celotna rešitev ima torej še vedno časovno zahtevnost $O(n)$ — asimptotično gledano ni nič slabša od rešitve naše prvotne naloge.

4. Valj

Nalogo lahko rešujemo z iskanjem v širino. Začnimo v poljubnem polju (x_0, y_0) in pregledujmo mrežo tako, da se vedno premaknemo iz trenutnega polja na sosednja polja le, če so iste barve. Tako bomo sčasoma pregledali celotno povezano komponento (strnjeno zaplato polj iste barve), ki ji pripada (x_0, y_0). Pri tem veljata dve polji za sosednji, če imata skupno stranico, poleg tega pa sta za si v vsaki vrstici sosednji tudi skrajno levo in skrajno desno polje — s tem upoštevamo dejstvo, da naša mreža ne leži na ravnini, ampak tvori plašč valja.

Definicija obhoda valja v besedilu naloge je koristen namig: za obhod desno mora biti vsaka navpičnica prečkana enkrat več v desno kot v levo. Število teh prečkanj pri meji med stolpcema x in $x+1$ označimo z d_x oz. ℓ_x. Pri obhodu v desno mora torej za vsak x veljati $d_x - \ell_x = 1$; in če to seštejemo po vseh m navpičnicah, mora za obhod kot celoto veljati $d - \ell = m$, kjer je d skupno število korakov v desno, ℓ pa skupno število korakov v levo. Podobno mora pri obhodu v levo veljati $d - \ell = -m$.

Zato je ob pregledovanju mreže z iskanjem v širino koristno, če si za vsako polje zapomnimo tudi razliko med številom korakov v desno in korakov v levo na naši poti od (x_0, y_0) do tistega polja; recimo temu $d(x, y)$. Če potem med pregledovanjem sosedov nekega polja (x', y') vidimo, da smo nekega takega soseda (x, y) obiskali že prej in da se je takratni $d(x, y)$ razlikoval ravno za $\pm m$ od vrednosti, ki bi jo dobil, če bi na tisto polje (ponovno) stopili zdaj — recimo tej vrednosti $d'(x, y)$ —, potem vemo, da imamo obhod okrog valja: sestavlja ga najprej pot, po kateri smo prišli od (x_0, y_0) do (x', y'), nato korak od tam v njegovega soseda (x, y), nato pa pot od tam do (x_0, y_0), ravno obratna od poti, po kateri smo prvič dosegli polje (x, y).

Skupaj je namreč razlika med številom desnih in levih korakov na tem obhodu enaka $d'(x,y) - d(x,y)$, to pa je ravno $\pm m$.[7]

Ko na ta način pregledamo celotno povezano komponento, dosegljivo iz (x_0, y_0), jo v mislih pobrišemo in nadaljujemo pri kakšnem drugem še nepregledanem polju, dokler ni pregledana celotna mreža. V spodnji rešitvi označujemo polja, ki smo jih že odkrili (in dodali v vrsto pri iskanju v širino), tako, da postavimo njihovo barvo na -1. Za polja, ki ne pripadajo trenutni komponenti, imamo v $d(x,y)$ vrednost $n(m+1)$, ki je gotovo za več kot m različna od katerekoli vrednosti, ki jo lahko $d(x,y)$ dobi pri poljih, ki pripadajo trenutni komponenti.

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

int main()
{
  // Preberimo vhodne podatke.
  int w, h; scanf("%d %d", &h, &w);
  vector<int> A(w * h); for (auto &a : A) scanf("%d", &a);

  vector<int> rezultati; // Seznam barv, po katerih je obhod mogoč.
  vector<int> D(w * h, w * (h + 1)); // Zamik (v smeri x) glede na začetno polje.
  vector<int> vrsta; // Vrsta za iskanje v širino.
  const int DX[] = { 1, -1, 0, 0 }, DY[] = { 0, 0, 1, -1 };

  for (int y0 = 0; y0 < h; ++y0) for (int x0 = 0; x0 < w; ++x0)
  {
    int barva = A[y0 * w + x0]; if (barva < 0) continue;

    // Pojdimo iz polja (x0, y0) po poljih iste barve.
    D[y0 * w + x0] = 0; A[y0 * w + x0] = -1;
    vrsta.clear(); int glava = 0; vrsta.emplace_back(y0 * w + x0);
    bool obstajaObhod = false;
    while (glava < vrsta.size())
    {
      int u = vrsta[glava++]; int ux = u % w, uy = u / w;
      // Dosegli smo polje (ux, uy). Preglejmo njegove sosede.
      for (int smer = 0; smer < 4; ++smer)
      {
        int vx = (ux + DX[smer] + w) % w, vy = uy + DY[smer];
        if (vy < 0 || vy >= h) continue;
        int v = vx + w * vy;

        // Če smo tega soseda dosegli že prej pri trenutnem (x0, y0) in je
        // razlika med takratnim in sedanjim zamikom ravno ± w, potem obstaja obhod.
        if (abs(D[u] + DX[smer] - D[v]) == w) obstajaObhod = true;
        if (A[v] != barva) continue;
```

[7] Tu se morda utegne pojaviti pomislek, ali bi se lahko zgodilo, da bi obstajal le obhod, ki gre večkrat okrog valja, ne pa tudi obhod, ki gre okrog natanko enkrat; tedaj bi morali preverjati, ali je razlika $d'(x,y) - d(x,y)$ večkratnik m, ne pa le $\pm m$. Pa recimo, da se to res zgodi. Poskusimo mrežo poplaviti (*flood fill*) od spodnjega roba proti zgornjemu, pri čemer se lahko poplava širi s trenutnega polja na vse njegove sosede, ki imajo z njim skupno vsaj eno oglišče, ne sme pa se širiti na polja, ki pripadajo našemu obhodu. Ker gre obhod vsenaokrog valja, bo naši poplavi prepričil, da bi kjerkoli dosegla zgornji rob mreže. Poplavljeno območje torej na vrhu vsepovsod meji na celice, ki pripadajo našemu obhodu, in iz teh mejnih celic lahko zdaj sestavimo krajši obhod, ki gre okrog valja le enkrat. Torej se res ne more zgoditi, da bi obstajal le obhod, ki gre okrog valja večkrat, ne pa tudi tak, ki gre okrog njega le enkrat.

```
            // Če je sosed enake barve in ga prej še nismo dosegli, ga dodajmo v vrsto.
            A[v] = −1; D[v] = D[u] + DX[smer];
            vrsta.emplace_back(v);
          }
        }
        if (obstajaObhod) rezultati.emplace_back(barva);
        // Počistimo za sabo v tabeli D.
        for (int u : vrsta) D[u] = w * (h + 1);
      }
      // Izpišimo rezultate; pazimo, da izpišemo vsako barvo največ enkrat.
      sort(rezultati.begin(), rezultati.end());
      int prej = −1; for (int barva : rezultati) if (barva != prej) {
        printf("%s%d", prej >= 0 ? " " : "", barva); prej = barva; }
      printf("%s\n", rezultati.empty() ? "mavrica" : ""); return 0;
}
```

Časovna zahtevnost te rešitve je $O(nm)$ za samo iskanje v širino in $O(n \log n)$ za urejanje seznama rezultatov. Vsak obhod je namreč sestavljen iz vsaj m polj, zato lahko obstajajo obhodi za največ $(nm)/m = n$ različnih barv (npr. če je vsaka vrstica v celoti svoje barve).

Kot zanimivost si oglejmo še eno rešitev te naloge. Oprli se bomo na dejstvo, da če obstaja obhod okrog valja, po kakršnem sprašuje naša naloga, to pomeni, da ni mogoče priti od zgornjega roba mreže do spodnjega, ne da bi pri tem stopili na kakšno polje, ki pripada temu obhodu; če pa neka povezana skupina polj enake barve ne tvori obhoda, potem je gotovo mogoče priti od zgornjega roba mreže do spodnjega, ne da bi stopili na kakšno polje te skupine (pri tej poti od zgornjega roba do spodnjega smemo delati tudi diagonalne korake, ne le vodoravnih in navpičnih).

Poiščimo torej spet v naši mreži z iskanjem v širino povezane komponente (maksimalne povezane skupine polj enake barve), nato pa sestavimo graf, v katerem je po ena točka za vsako od teh povezanih komponent; med dvema točkama naj bo (neusmerjena) povezava, če imata tisti dve komponenti kakšno skupno oglišče. Dodajmo še dve točki, recimo jima s in t, ki predstavljata zgornji in spodnji rob mreže; s naj bo povezana s tistimi komponentami, ki vsebujejo kakšno polje v prvi vrstici, t pa s tistimi, ki vsebujejo kakšno polje v zadnji vrstici. Ta graf je povezan; toda če je v mreži obhod, kakršnega iščemo, to pomeni, da se od zgornjega do spodnjega roba mreže ne da priti drugače kot tako, da enkrat prečkamo ta obhod, torej vsaka pot od s do t v našem grafu obišče tudi točko, ki predstavlja komponento, v kateri leži ta naš obhod. Če tisto točko pobrišemo iz grafa, torej ta ne bo več povezan, ker se od s do t ne bo več dalo priti. Točkam, za katere velja, da graf preneha biti povezan, če tisto točko pobrišemo, pravimo *artikulacijske točke* in jih lahko poiščemo v linearno mnogo časa z algoritmom, ki temelji na iskanju v globino.[8] Vse, kar moramo storiti, je torej to, da v našem grafu poiščemo vse artikulacijske točke in izpišemo njihove barve (duplikate pri tem zavržemo).

Za konec razmislimo še o težjih različicah naloge, ki ju omenja opomba pod črto v besedilu naloge. Pri prvi, kjer je podano zaporedje barvanj posameznih polj in nas

[8] Za več o algoritmu za iskanje artikulacijskih točk gl. npr. Wikipedijo *s. v.* Biconnected component.

zanima prvi trenutek, ko obstaja obhod, lahko delamo bisekcijo po zaporedju in na vsakem koraku preverimo, ali obhod takrat že obstaja. Pri tem uporabimo enak algoritem kot pri prvotni nalogi, le da nepobarvanih polj ne upoštevamo, poleg tega pa nam tudi ni treba iskati vseh barv, kjer je obhod mogoč, pač pa le eno (kajti ko bomo z bisekcijo pravilno zadeli najzgodnejši trenutek, bo takrat tako ali tako mogoč obhod le pri eni barvi). Časovna zahtevnost te rešitve je torej $O(nm \log(nm))$.

Malo težja je druga različica naloge, kjer moramo po vsakem barvanju takoj sproti povedati, ali zdaj že obstaja kak obhod ali ne. Recimo, da imamo nalogo že rešeno po določenem številu barvanj; vsako povezano komponento imamo torej v celoti pregledano iz nekega začetnega polja na tej komponenti, za vsako polje (x, y) te komponente pa imamo podatek $d(x, y)$ o razliki v številu desnih in levih korakov na poti, po kateri smo prišli od začetnega polja do nje. Kaj se mora pri tem spremeniti, ko se pobarva neko novo polje, recimo polje (x, y) z barvo c?

Če nima novo polje nobenega soseda barve c, nastane nova komponenta, ki obsega le to polje; razglasimo ga za začetno polje te komponente in mu pripišimo $d(x, y) = 0$.

Če ima novo polje enega soseda barve c, recimo (x', y'), dodajmo novo polje v komponento, ki ji že pripada ta sosed, in novemu polju pripišimo $d(x, y) = d(x', y') + (x - x')$.

Če ima novo polje več kot enega soseda barve c in vsi ti sosedje pripadajo isti komponenti, dodajmo tudi novo polje v to komponento; njegovo $d(x, y)$ določimo s pomočjo enega od sosedov, pri ostalih sosedih pa potem preverimo, če bi nastala za $\pm m$ drugačna vrednost, če bi $d(x, y)$ določili iz tistega drugega soseda. V tem primeru vemo, da bi zdaj pri tej komponenti nastal obhod. Če obhoda take barve doslej še nismo videli, ga zdaj izpišimo.

Ostane še možnost, da ima novo polje več kot enega soseda barve c in da ti sosedje pripadajo več različnim komponentam. V tem primeru dodamo novo polje v največjo od teh komponent (tisto z najmanj polji), nato pa z iskanjem v širino obiščimo novo polje in še vsa polja manjših komponent, jih dodajmo v največjo komponento in jim na novo določimo d (na podlagi začetnega polja največje komponente) ter pri tem tudi preverjajmo, če nastane kak obhod (enako kot pri rešitvi prvotne naloge).

Zaradi takšnega združevanja komponent se lahko zgodi, da med barvanjem mreže posamezno polje sicer pregledamo večkrat in ga selimo iz ene komponente v drugo; vendar pa je nova komponenta vedno vsaj dvakrat tolikšna od prejšnje (ker vedno pripajamo manjšo komponento večji), zato je takšnih selitev lahko kvečjemu $\log_2(wh)$. Časovna zahtevnost tega postopka je tako $O(wh \log(wh))$.

5. Urejanje z medianami

V nalogi so podstavki oštevilčeni od 1 do n, vendar je za potrebe pisanja rešitve v C++ bolj prikladno, če jih oštevilčimo od 0 do $n - 1$ (pri sporazumevanju z ocenjevalnim sistemom pa tem številkam prištejemo 1). Poleg tega bomo oštevilčili tudi zaboje: vsak zaboj naj dobi številko tistega podstavka, na katerem je stal na začetku urejanja, preden smo zaboje začeli premikati. Operacijam prvega tipa bomo rekli na kratko „zamenjave“, operacijam drugega pa „izračuni median“ ali krajše kar „mediane“, če ne bo tveganja za zmedo.

Znanih je veliko algoritmov za urejanje, ki večinoma temeljijo na tem, da je mogoče dva elementa primerjati in povedati, kateri mora v urejenem vrstnem redu priti pred drugim. Pri naši nalogi ne moremo primerjati dveh zabojev, pač pa lahko z izračunom mediane primerjamo tri zaboje; izziv je torej predvsem v tem, kako kakšnega od znanih algoritmov za urejanje prilagoditi tako, da bo namesto primerjave uporabljal izračune median.

Uporabimo lahko na primer **urejanje z vstavljanjem** (*insertion sort*): tu postopoma pripravljamo urejen seznam zabojev, pri čemer začnemo s praznim seznamom, nato pa na vsakem koraku vzamemo neki zaboj ter ga vstavimo v naš seznam na tako mesto, da seznam ostane urejen. Toda če bomo pri tem vrivanju zaboje dejansko premikali po skladišču, bomo izvedli $O(n^2)$ zamenjav, kar je že preveč (spomnimo se, da gre n do 1000, omejeni pa smo na 40 000 operacij). Namesto tega si bomo urejeni seznam zabojev pripravljali v pomnilniku, ko pa bo nared, bomo zaboje zares prerazporedili z največ $O(n)$ zamenjavami (za to v spodnji rešitvi poskrbi podprogram ZakljuciUrejanje).

Razmisliti moramo še o tem, kako ugotoviti, kam v urejeni seznam je treba vriniti novi zaboj, da bo ostal urejen. Če bi imeli možnost primerjati po dva zaboja, bi lahko uporabili bisekcijo; ker pa imamo na voljo le izračun mediane, ki nekako primerja tri zaboje, lahko namesto bisekcije uporabimo „trisekcijo“: naj bo m_1 zaboj približno na eni tretjini našega seznama, m_2 zaboj približno na dveh tretjinah seznama in x naš novi zaboj; tedaj vemo, da če je mediana zabojev m_1, m_2 in x ravno zaboj x, to pomeni, da je x po teži med m_1 in m_2, torej ga moramo vriniti v srednjo tretjino seznama; če je mediana omenjenih treh zaboj m_1, to pomeni, da m_1 leži po teži med x in m_2, torej moramo x vriniti v prvo tretjino seznama, levo od m_1; in podobno, če je mediana zaboj m_2, leži m_2 po teži med m_1 in x, torej moramo x vrniti v zadnjo tretjino seznama, desno od m_2. Tako smo z enim izračunom mediane skrajšali območje, ki še pride v poštev za vrivanje zaboja x, na tretjino prvotne dolžine. V naslednjem koraku trisekcije razdelimo to tretjino spet na tretjine in tako naprej.

Če smo imeli seznam k zabojev, bomo tako v približno $\log_3 k$ korakih našli pravi položaj, kamor je treba vriniti novi zaboj. Tako imamo vsega skupaj približno $\sum_{k=3}^{n} \log_3 k \approx n \log_3 n - O(n)$ računanj mediane. S tem smo že zelo blizu teoretični spodnji meji: ker moramo znati prerazporediti zaboje na $n!$ možnih načinov in ker se lahko obnašanje naše rešitve po vsakem izračunu mediane razveji v tri možna nadaljevanja, moramo gotovo izvesti vsaj $\log_3 n!$ izračunov mediane, kar je tudi približno $n \log_3 n - O(n)$.

Zapišimo dosedanjo rešitev v C++:

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

// Definirajmo tipa Zaboj in Podstavek, da se bo v izvorni kodi
// bolje videlo, katere vrednosti so številke zabojev in katere podstavkov.
typedef int Zaboj, Podstavek;

int n; // število podstavkov in tudi število zabojev
```

```
vector<Zaboj> zaboji;      // zaboji[p] = zaboj na podstavku p
vector<Podstavek> kjeJe; // kjeJe[z] = podstavek, kjer stoji zaboj z

// Zamenja zaboja na podstavkih p1 in p2.
void ZamenjajP(Podstavek p1, Podstavek p2) {
  printf("%d %d -1\n", p1 + 1, p2 + 1); fflush(stdout);
  Zaboj z1 = zaboji[p1], z2 = zaboji[p2];
  swap(zaboji[p1], zaboji[p2]); swap(kjeJe[z1], kjeJe[z2]); }

// Zamenja zaboja s številkama z1 in z2.
void ZamenjajZ(Zaboj z1, Zaboj z2) { ZamenjajP(kjeJe[z1], kjeJe[z2]); }

// Vrne številko podstavka, na katerem je srednji zaboj po teži.
Podstavek MedianaP(Podstavek p1, Podstavek p2, Podstavek p3) {
  printf("%d %d %d\n", p1 + 1, p2 + 1, p3 + 1); fflush(stdout);
  int m; scanf("%d", &m); return m - 1; }

// Kot MedianaP, le da dela s številkami zabojev namesto podstavkov.
Zaboj MedianaZ(Zaboj z1, Zaboj z2, Zaboj z3) {
  return zaboji[MedianaP(kjeJe[z1], kjeJe[z2], kjeJe[z3])]; }

// Razporedi zaboje v vrstni red, kot ga določa „v“.
void ZakljuciUrejanje(const vector<Zaboj> &v) {
  for (Podstavek p = 0; p < n; ++p)
    if (int q = kjeJe[v[p]]; q != p) ZamenjajP(p, q);
  printf("-1\n"); fflush(stdout); }

// Ugotovi, na kateri indeks v „v“ je treba vriniti novi zaboj,
// da bo zaporedje ostalo urejeno po teži.
int Trisekcija(const vector<Zaboj> &v, Zaboj novi)
{
  int L = 0, D = v.size();
  while (D > L)
  {
    // Novi zaboj bo treba vriniti v „v“ na enega od indeksov L, ... , D.
    if (D - L == 1) { if (L > 0) --L; else ++D; }

    // Zdaj območje L, ... , D obsega vsaj tri indekse. Razdelimo ga na tretjine.
    int M1 = (2 * L + D) / 3, M2 = (L + 2 * D) / 3;

    // V katero tretjino sodi novi zaboj?
    Zaboj med = MedianaZ(v[M1], v[M2], novi);
    if (med == v[M1]) D = M1;
    else if (med == v[M2]) L = M2 + 1;
    else L = M1 + 1, D = M2;
  }
  return L;
}

void UrediZVstavljanjem(vector<Zaboj> &v)
{
  // Zaboje bomo pregledovali v naključnem vrstnem redu.
  vector<int> vrstniRed(n); mt19937_64 r;
  for (int i = 0; i < n; ++i) {
    int j = uniform_int_distribution<int>(0, i)(r);
    vrstniRed[i] = vrstniRed[j]; vrstniRed[j] = i; }

  // V vektorju „v“ bo nastajal vrstni red, kjer bodo zaboji urejeni po teži.
  v.clear(); v.reserve(n);
  for (Zaboj z : vrstniRed)
```

```
    if (v.size() < 2) v.emplace_back(z);
    else v.insert(v.begin() + Trisekcija(v, z), z);
}

int main()
{
  scanf("%d", &n); zaboji.resize(n); kjeJe.resize(n);
  for (int i = 0; i < n; ++i) zaboji[i] = i, kjeJe[i] = i;
  vector<Zaboj> vrstniRed; UrediZVstavljanjem(vrstniRed);
  ZakljuciUrejanje(vrstniRed); return 0;
}
```

Slabost te rešitve je, da obraba ni dovolj enakomerno porazdeljena po podstavkih. Recimo, da smo v urejeno zaporedje že dodali k zabojev; ko dodajamo naslednji zaboj, nastopata v prvi iteraciji trisekcije zaboja na mestih (približno) $k/3$ in $2k/3$ v tem urejenem zaporedu. Zanju torej velja, da je približno tretjina izmed dosedanjih k zabojev lažjih oz. težjih od njiju; zato pa lahko pričakujemo, da bo tudi približno tretjina izmed vseh n zabojev lažjih oz. težjih od njiju, saj je bilo dosedanjih k zabojev naključno izbranih izmed vseh n zabojev in bi morale biti njihove teže podobno porazdeljene. V prvi iteraciji vsakega izvajanja trisekcije torej praviloma nastopata dva taka zaboja, ki bosta v končnem vrstnem redu vseh n zabojev v bližini mest $n/3$ in $2n/3$; podstavki, na katerih stojijo takšni zaboji, se zato zelo nadpovprečno obrabijo, saj zabojev medtem nič ne premikamo (to naredimo šele na koncu, v podprogramu ZakljuciUrejanje). Pri naših poskusih z $n = 1000$ je bila maksimalna obraba (po vseh podstavkih) v povprečju okrog 134 in na testnih primerih s tekmovanja bi ta rešitev dobila 86 točk od 100.

Skupno število računanj mediane je, kot smo videli, približno $n \log_3 n$ in pri vsakem se obrabijo trije podstavki; vseh podstavkov pa je n, torej bo povprečna obraba približno $3 \log_3 n$, kar pri $n = 1000$ nanese približno 18,9. Če torej hočemo, da bo maksimalna obraba ≤ 20 (kar naloga zahteva za vse točke), bomo morali poskrbeti, da bodo podstavki obrabljeni zelo enakomerno. (V resnici so sicer naše ocene tukaj nekoliko pesimistične; dejansko število računanj mediane je bilo pri $n = 1000$ pri naših poskusih v povprečju okrog 5579, kar pomeni povprečno obrabo 16,7.)

Pri računanju mediane nas v resnici zanima mediana treh *zabojev*; omejitev, ki nam jo postavlja naloga, pa se nanaša na to, pri koliko računanjih mediane sodeluje posamezni *podstavek*, ne pa zaboj. Če bi se torej slučajno izkazalo, da bi neki zaboj potrebovali pri veliko računanjih mediane, nas nič ne sili, da ga imamo ves čas na istem podstavku in tako povečujemo obrabo tega podstavka; zaboj lahko vsake toliko časa premaknemo na kak drug podstavek, ki je zaenkrat manj obrabljen. Tako lahko z zaboji „kolobarimo“ med podstavki in skrbimo, da so vsi podstavki čim bolj enakomerno obrabljeni (torej da vsi nastopijo v približno enako veliko računanjih mediane).

Dogovorimo se na primer, da bomo poskušali vzdrževati naslednjo lastnost: razlika med maksimalno in minimalno obrabo (po vseh odstavkih) sme biti največ R, razen če je maksimalna obraba $\leq B$. Pri tem sta B in R konstanti, ki si ju bomo izbrali pred začetkom urejanja. Če bi neki zaboj x, ki ga nameravamo uporabiti pri naslednjem izračunu mediane, to pravilo prekršil (ker bi se zaradi povečanja njegove obrabe povečala tudi maksimalna obraba in bila po novem večja od B ter tudi za

več kot R večja od minimalne obrabe), ga bomo zamenjali z enim od zabojev, ki so trenutno na podstavkih z minimalno obrabo.[9]

Namen tega pravila je, da dokler je obraba pri vseh podstavkih nizka (do B), se nam ni treba truditi z uravnoteževanjem obrabe in prerazporejati zabojev po podstavkih; kasneje pa bo posamezni zaboj treba prerazporediti le na vsakih R računanj mediane, v katerih je udeležen. Najmanjšo maksimalno obrabo, vendar tudi največ zamenjav, bomo dobili pri $B = 0$, $R = 1$ (pri naših poskusih je bilo tu pribl. 2,6-krat toliko zamenjav kot računanj mediane); precej zamenjav pa lahko prihranimo, če postavimo B višje. Pri našem postopku s trisekcijo smo na primer videli, da bo povprečna obraba približno $3 \log_3 n$, maksimalna obraba po vseh podstavkih pa tudi ne more biti manjša od povprečne. Postavimo torej B malo pod to mejo, na primer na $3(\lfloor \log_3 n \rfloor - 1)$, parameter R pa naj ostane 1: dobili bomo enako dobro maksimalno obrabo kot pri $B = 0$, $R = 1$, število zamenjav pa bo zdaj manjše kot število računanj mediane.[10]

Oglejmo si implementacijo takšnega uravnoteževanja obrabe. Za vsako možno obrabo si načeloma želimo vzdrževati seznam vseh podstavkov s to obrabo; da bomo lahko podstavek poceni premaknili z enega seznama na naslednjega, ko se mu obraba poveča, bomo za sezname uporabili verige, povezane s kazalci (*linked lists*). Te sezname hranimo v vektorju postavkiPoObrabi, ki ga uporabljamo kot krožno tabelo: postavki z obrabo x so v postavkiPoObrabi[x % M], pri čemer mora biti M dovolj velik, da gotovo ne bomo hkrati potrebovali več kot M takih seznamov. Primerna vrednost je $M = \max\{B, R\} + 2$.[11]

struct Uravnotezevalnik

[9] Natančneje rečeno: med zaboji, ki ne nastopajo v naslednjem izračunu mediane, vzamemo tistega, ki trenutno stoji na najmanj obrabljenem podstavku, in ga zamenjamo z zabojem x. Lahko se tudi zgodi, da primernega kandidata za zamenjavo sploh ni (oz. so sami taki, katerih podstavki so enako obrabljeni kot x-ov, takrat pa od zamenjave ne bi bilo nobene koristi).

[10] Teoretično se sicer lahko pri $B > R$ zgodi, da ko razlika med maksimalno in minimalno obrabo na začetku naraste na B, je opisani postopek uravnoteževanja obrabe s prerazporejanjem zabojev kasneje ne bo uspel zmanjšati na R. Recimo, da je v nekem trenutku maksimalna obraba (po vseh podstavkih) B, minimalna pa 0; in recimo, da v naslednjem izračunu mediane nastopa α takih zabojev, katerih podstavek ima trenutno maksimalno obrabo (torej B); in recimo, da je med podstavki z obrabo, nižjo od maksimalne, trenutno β takih, katerih zaboji ne nastopajo v naslednjem izračunu mediane. Če je $\alpha \leq \beta$, bomo lahko vseh tistih α zabojev premaknili na take podstavke, ki imajo obrabo nižjo od maksimalne in ki jih drugače ne bi uporabili v novem izračunu mediane; zato do povečanja maksimalne obrabe ne bo prišlo. Pri $\alpha > \beta$ pa ni dovolj podstavkov z nižjo porabo, da bi tja lahko premaknili vseh α zabojev, ki nam povzročajo težave; vsaj β pa jih vseeno lahko premaknemo in po tem premiku zdaj prav vsak podstavek, ki je imel obrabo manjšo od maksimalne, sodeluje v novem izračunu mediane. Zato se pri tem izračunu minimalna obraba poveča za 1, prav tako pa maksimalna (zaradi tistih $\alpha - \beta$ zabojev, ki so še sodelovali v trenutnem izračunu mediane in so ostali na podstavkih z obrabo B). Torej se maksimalna obraba poveča (na $B + 1$), razlika med maksimalno in minimalno pa ostane enaka. — Tako torej vidimo, da ko razlika med maksimalno in minimalno obrabo naraste na B, ne moremo zagotoviti, da se bo kasneje spet zmanjšala, pač pa lahko minimalna in maksimalna obraba naraščata obe skupaj s stalno razliko B. Vendar pa pri naših poskusih nikoli ni prišlo do tega, da bi z $B = 3(\lfloor \log_3 n \rfloor - 1)$ in $R = 1$ dobili na koncu slabši rezultat (višjo maksimalno obrabo) kot z $B = 0$ in $R = 1$.

[11] Ko je maksimalna obraba B (ali manj), je lahko minimalna obraba tudi samo 0; takrat imamo hkrati v rabi največ $B + 1$ seznamov. Če potem v naslednjem izračunu mediane nastopata neki zaboj na podstavku z obrabo B in neki zaboj na podstavku z obrabo 0, se lahko zgodi, da želimo prvega premakniti v seznam za obrabo $B + 1$, še preden smo drugega premaknili na seznam za obrabo 1; zato bomo imeli hkrati v rabi celo $B + 2$ seznama. Podoben razmislek velja kasneje, ko je maksimalna obraba $> B$ in se trudimo razliko med maksimalno in minimalno obrabo vzdrževati na največ R. Tako torej vidimo, da hkrati potrebujemo največ $\max\{B + 2, R + 2\}$ seznamov.

```
{
  int B, R, M;

  // Obraba vsakega podstavka ter minimum in maksimum po vseh podstavkih.
  vector<int> obrabaPodstavka; int minObraba, maxObraba;

  // Podstavki z obrabo x tvorijo seznam postavkiPoObrabi[x % M].
  vector<list<int>> podstavkiPoObrabi;

  // kjePodstavek[p] kaže na element z vrednostjo p v seznamu
  // podstavkiPoObrabi[obrabaPodstavka[x] % M].
  vector<list<int>::iterator> kjePodstavek;

  void Init(int n)
  {
    int log3_n = 0, N = n; while (N > 1) N /= 3, ++log3_n;
    B = 3 * max(log3_n - 1, 0); R = 1; M = max(B, R) + 2;

    // Na začetku je obraba vseh podstavkov 0.
    minObraba = 0; maxObraba = 0; obrabaPodstavka.clear(); obrabaPodstavka.resize(n, 0);

    // Dodajmo vse podstavke v seznam podstavkiPoObrabi[0].
    podstavkiPoObrabi.clear(); podstavkiPoObrabi.resize(M);
    kjePodstavek.clear(); kjePodstavek.resize(n);
    auto &L = podstavkiPoObrabi[0];
    for (Podstavek p = 0; p < n; ++p) kjePodstavek[p] = L.emplace(L.begin(), p);
  }

  // Po potrebi skuša premakniti zaboj z na kak drug podstavek, da se
  // maksimalna obraba ne bi povečala. Pri tem ne uporabi podstavkov,
  // na katerih sta trenutno zaboja z1 in z2 (ker bosta tadva zaboja tudi
  // sodelovala v naslednjem izračunu mediane in zato od takšne zamenjave
  // ne bi bilo nobene koristi).
  void Uravnotezi(Zaboj z, Zaboj z1 = -1, Zaboj z2 = -1)
  {
    Podstavek p = kjeJe[z];
    if (obrabaPodstavka[p] == maxObraba && maxObraba >= B &&
                                          maxObraba - minObraba >= R)
      // Premaknimo ta zaboj na kak manj obrabljen podstavek. Vzeli bomo
      // najmanj obrabljen tak podstavek, ki ne vsebuje zabojev z1 ali z2.
      for (int o = minObraba; o < maxObraba; ++o)
        for (Podstavek q : podstavkiPoObrabi[o % M]) {
          if (Zaboj z = zaboji[q]; z == z1 || z == z2) continue;
          ZamenjajP(p, q); p = q;
          o = maxObraba; break; }

    // Zapomnimo si, da bo obraba podstavka p zdaj narasla.
    int &obraba = obrabaPodstavka[p];
    auto &prej = podstavkiPoObrabi[obraba % M],
         &potem = podstavkiPoObrabi[(obraba + 1) % M];
    potem.splice(potem.begin(), prej, kjePodstavek[p]); // Preselimo ga v naslednji seznam.
    if (prej.empty() && obraba == minObraba) ++minObraba;
    if (++obraba > maxObraba) maxObraba = obraba;
  }
} U;
```

V funkcijo main dodajmo klic U.Init(n), lahko takoj po tistem, ko preberemo n. Namesto dosedanje funkcije MedianaZ pa bomo morali (v funkciji Trisekcija) za izračun mediane treh zabojev uporabljati naslednjo:

```
// Kot MedianaZ, vendar z uravnoteževanjem obrabe.
```

```
Zaboj MedianaZU(Zaboj z1, Zaboj z2, Zaboj z3) {
  U.Uravnotezi(z1, z2, z3); U.Uravnotezi(z2, z1, z3); U.Uravnotezi(z3, z1, z2);
  return MedianaZ(z1, z2, z3); }
```

Na testnih primerih s tekmovanja dobi ta rešitev vse točke; pri $n = 1000$ je maksimalna obraba 17.

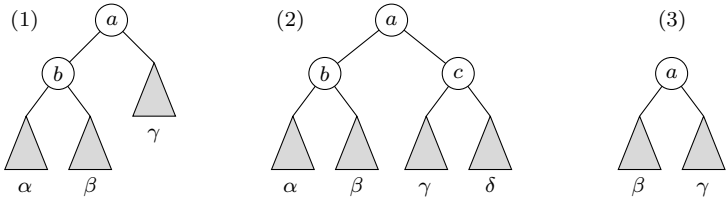
Na našem tekmovanju bi sicer zadoščala že preprostejša implementacija uravnoteževalnika. Zgoraj opisana različica porabi $O(1)$ časa za vsak klic Uravnotezi; ker pa bo pri nas n majhen, bi bilo dovolj že, če bi šla funkcija Uravnotezi v zanki po tabeli obrabaPodstavka in tako v $O(n)$ časa poiskala najmanj obrabljeni podstavek; tako ne bi bilo treba vzdrževati seznamov podstavkiPoObrabi. Poleg tega tudi ni treba veliko razmišljati o parametrih B in R; ker vemo, da obraba ne sme preseči 20, če hočemo dobiti vse točke, bi lahko funkcija Uravnotezi premeščala zaboj z tudi samo takrat, ko bi ta stal na podstavku z obrabo 20.

Nalogo lahko rešimo še na veliko drugih načinov; na kratko si jih oglejmo nekaj. V oglatih oklepajih je pri posameznih rešitvah napisano, koliko točk bi dobile na testnih primerih z našega tekmovanja.

Rešitve z drevesom. Naša dosedanja rešitev porabi načeloma $O(n^2)$ časa za vrivanje zabojev v urejeni seznam. To nas sicer ni motilo, ker je n pri naši nalogi majhen in ker zaboje premikamo le v pomnilniku, ne delamo pa res zamenjav v skladišču. Če pa bi hoteli časovno zahtevnost vendarle zmanjšati, bi morali za predstavitev urejenega seznama namesto tabele oz. vektorja uporabiti kakšno primerno uravnoteženo drevesasto strukturo (npr. rdeče-črno drevo, AVL-drevo, B-drevo, treap). Tam bi vrivanje v drevo vzelo $O(\log n)$ časa, enako pa tudi dostop do elementa na določenem indeksu; ker potrebujemo dva taka dostopa (do zabojev m_1 in m_2) na vsakem koraku trisekcije, teh pa je skupno $O(n \log n)$; bi bila časovna zathevnost te rešitve $O(n(\log n)^2)$.

Še ena možnost pa je, da našo trisekcijo prilagodimo strukturi drevesa. Recimo, da imamo binarno drevo (razmisleka ni težko posplošiti tudi na drevesa, kjer ima lahko posamezno vozlišče več kot dva otroka, npr. B-drevo) in bi radi vanj na primerno mesto dodali novi zaboj x; in recimo, da je koren drevesa zaboj a, ki ima kot levega otroka zaboj b (drevo (1) na sliki na str. 92). Izračunajmo mediano zabojev a, b in x; če je mediana zaboj b, to pomeni, da je x še lažji od b in ga bo treba dodati v poddrevo α; če je mediana zaboj a, to pomeni, da je x težji od a in ga bo treba dodati v poddrevo γ; sicer (če je mediana zaboj x) pa je x težji od b, vendar lažji od a in ga bo treba dodati v poddrevo β. Če poddrevesa, ki smo si ga na ta način izbrali, sploh ni oz. je prazno, lahko novi zaboj x pripnemo v drevo kar tja (po tistem moramo seveda še vedno poskrbeti tudi za uravnoteževanje drevesa, da se nam ne bo izrodilo v seznam ali kaj podobnega), sicer pa se premaknimo v vozlišče, ki predstavlja koren tega poddrevesa, in nadaljujmo tam z enakim razmislekom kot doslej. Tako smo izvedli en izračun mediane in se premaknili za en ali dva nivoja navzdol po drevesu.

Namesto da gledamo koren a in njegovega levega otroka, bi lahko seveda gledali tudi a in njegovega desnega otroka ter razmišljali na podoben način. Še ena možnost, če ima a oba otroka, levega b in desnega c (drevo (2) na sliki), pa je, da izračunamo mediano zabojev b, c in x; če je mediana b, bo treba novi zaboj dodati v poddrevo α; če je mediana c, ga bo treba dodati v δ; če pa je mediana x, ga bo



treba dodati v eno od poddreves β in γ, kar lahko obravnavamo tako, da si začasno mislimo, da sta to poddrevesi manjšega drevesa s korenom a (drevo (3) na sliki). Med temi različnimi možnostmi glede tega, katera dva zaboja iz drevesa uporabiti v naslednjem izračunu mediane, lahko uporabimo tisto, pri kateri je število zabojev v treh delih, na katere nam drevo razpade, če ga razrežemo pri tistih dveh zabojih, čim bolj izenačeno (pri drevesu (1) na sliki so ti trije deli na primer poddrevesa α, β in γ; pri drevesu (2), če uporabimo v izračunu mediane b in c, pa so trije deli potem α, δ ter drevo (3) na sliki) — tako se bomo čim bolj približali idealu, po katerem bi želeli deliti zaporedje zabojev vedno na tretjine (kot pri prvotni različici naše rešitve s trisekcijo na seznamu namesto na drevesu).

Ker torej zdaj pred vsakim izračunom mediane porabimo le $O(1)$ časa, da izberemo zaboja, ki bosta (poleg novega zaboja x) nastopala v njem, in ker se po vsakem izračunu mediane premaknemo za en ali dva nivoja dol po drevesu, nam zdaj trisekcija vzame le $O(\log n)$ časa, časovna zahtevnost celotnega algoritma pa je $O(n \log n)$. Število izračunov mediane je sicer tu malo večje kot pri prvotni rešitvi (trisekciji na seznamu), kajti globina drevesa je sicer reda $O(\log n)$, vendar je večja od $\log_3 n$ (ker je binarno in ker je uravnoteženo le približno, ne pa popolnoma).

Podobno kot pri rešitvi s seznamom je tudi tu obraba precej neenakomerno porazdeljena po zabojih: ker se trisekcija vedno začne na vrhu drevesa, nastopajo zaboji blizu vrha v več izračunih mediane kot niže ležeči zaboji. Zato je tudi pri tej rešitvi pomembno uporabiti prej opisani prijem za uravnoteževanje obrave s premeščanjem pogosto uporabljenih zabojev med podstavki.

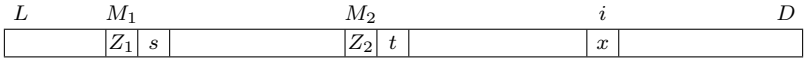
Hitro urejanje (quicksort). Še ena možnost je, da se namesto po urejanju z vstavljanjem zgledujemo po hitrem urejanju (*quicksort*). Spomnimo se, da si pri tem postopku izberemo delilni element in razdelimo tabelo na dva dela glede na to, ali so elementi manjši ali večji od delilnega, nato pa vsakega od teh dveh delov obdelamo z rekurzivnim klicem. Pri naši nalogi nimamo na voljo operacije, ki bi primerjala dva zaboja po teži, pač pa imamo izračun mediane, ki pravzaprav primerja po teži tri zaboje. Zato si namesto enega delilnega elementa izberimo dva in razdelimo tabelo na tri dele glede na to, ali so zaboji lažji od obeh delilnih elementov, težji od obeh ali pa so po teži med njima:

	M_1		M_2	
lažji od Z_1	Z_1	težji od Z_1, lažji od Z_2	Z_2	težji od Z_2

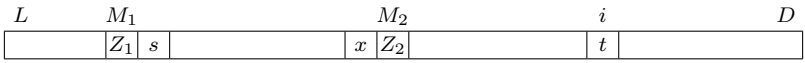
Vsakega od teh treh delov potem uredimo rekurzivno.

Oglejmo si ta postopek deljenja malo pobliže. Recimo, da imamo zaboje v tabeli v in da bi radi na tri dele razdelili območje $v[L..D]$. Indeksa delilnih elementov označimo z M_1 in M_2. Na začetku lahko vzamemo $M_1 = L$ in $M_2 = L + 1$, torej za delilna elementa uporabimo kar prva dva, in vidimo, da je območje $v[L..L + 1]$ že

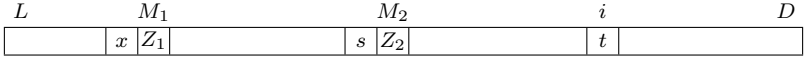
primerno razdeljeno. Nato gremo v zanki po vseh naslednjih elementih od $L+2$ do D. Če je območje $v[L..i-1]$ že primerno razdeljeno in bi radi zdaj to razširili še z elementom i, moramo preveriti, v katerega od treh delov sodi; to naredimo tako, da izračunamo mediano zabojev na indeksih M_1, M_2 in i, torej obeh delilnih elementov ter novega zaboja $v[i] = x$. Če je mediana zaboj $v[M_2]$, to pomeni, da je novi zaboj večji od obeh delilnih in lahko ostane tam, kjer je bil:



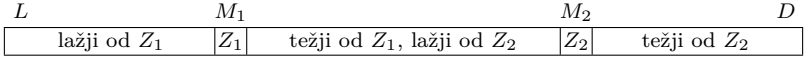
Sicer ga moramo premakniti pred drugi delilni zaboj, kar lahko naredimo z dvema zamenjavama; M_2 se ob tem poveča za 1:



Če je mediana delilnih zabojev in novega zaboja ravno novi zaboj, to pomeni, da je ta po teži med obema delilnima in je zdaj na primernem položaju; sicer pa (če je mediana prvi od delilnih zabojev) moramo zdaj novi zaboj premakniti pred prvega delilnega, kar lahko spet naredimo z dvema zamenjavama (in nato povečamo M_1):



V vsakem primeru smo torej v $O(1)$ časa poskrbeli, da je zdaj primerno razdeljeno že celotno območje $v[L..i]$, ne le $v[L..i-1]$. Tako počasi nadaljujemo po naraščajočih i in v $O(D-L)$ časa razdelimo celotno območje $v[L..D]$, ki nas zanima:



Nato lahko z rekurzivnimi klici ločeno uredimo vsakega od treh delov (od L do M_1-1; od M_1+1 do M_2-1; in od M_2+1 do D), pa bo s tem urejeno tudi območje od L do D v celoti.

Paziti moramo še na naslednjo podrobnost: doslej smo nekako predpostavili, da je delilni zaboj $v[M_1]$ lažji od $v[M_2]$, če pa je težji, bomo pač dobili padajoče urejeno zaporedje namesto naraščajočega. S tem ni nič narobe, težava pa nastopi, ker se podobno lahko zgodi tudi pri vsakem od rekurzivnih klicev; lahko je torej ena tretjina tabele urejena naraščajoče po teži, druga tretjina padajoče in podobno. Tej težavi se izognemo tako, da (če območje $v[L..D]$ ne obsega celotne tabele) najprej izračunamo mediano zabojev $v[L-1]$, $v[M_1]$ in $v[M_2]$; če mediana ni zaboj na M_1, moramo oba delilna zaboja med seboj zamenjati. Če smo na začetku tabele ($L = 0$), uporabimo $v[D+1]$ namesto $v[L-1]$ in če mediana zdaj ni zaboj na M_2, moramo delilna zaboja zamenjati.

```
// Uredi zaboje v[L..D]. Pri tem predpostavi, da so vsi ti zaboji po teži med
// zabojema v[L − 1] in v[D + 1], če obstajata.
void UrediSQuicksortom(vector<Zaboj> &v, int L, int D)
{
  if (D <= L) return; // ni česa urejati

  // Ker smo na začetku urejanja tabelo zabojev naključno premešali, lahko zdaj
  // za delilna delementa vzamemo kar prva dva na opazovanem območju.
  int M1 = L, M2 = L + 1; // indeks delilnih elementov

  // Če je treba, zamenjajmo delilna elementa med seboj, tako da bo zaporedje
```

```
  // v[L − 1], v[M1], v[M2], v[D + 1] urejeno (ni važno, ali naraščajoče ali padajoče).
  if (L > 0) { if (MedianaZU(v[L − 1], v[M1], v[M2]) == v[M2]) swap(v[M1], v[M2]); }
  else if (D < v.size() − 1) if (MedianaZU(v[M1], v[M2], v[D + 1]) == v[M1])
                                                              swap(v[M1], v[M2]);
  // Prerazporedimo ostale zaboje.
  for (int i = L + 2; i <= D; ++i)
  {
    // Na tem mestu velja: zaboji v[L], ..., v[M1 − 1] so lažji od v[M1];
    // zaboji v[M1 + 1], ..., v[M2 − 1] so težji od v[M1] in lažji od v[M2];
    // zaboji v[M2 + 1], ..., v[i − 1] so težji od v[M2].
    // Naloga trenutne iteracije je, da premakne zaboj v[i] tako, da bo
    // ta invarianta veljala tudi za i namesto i − 1.
    Zaboj m = MedianaZU(v[M1], v[M2], v[i]);

    // Če je novi zaboj lažji od v[M2], premaknimo le-tega za eno mesto naprej,
    // na njegovo mesto pa premaknimo novi zaboj.
    if (m != v[M2]) { swap(v[i], v[M2 + 1]); swap(v[M2], v[M2 + 1]); ++M2; }

    // Če je novi zaboj lažji tudi od v[M1], premaknimo še slednjega za eno
    // mesto naprej, na njegovo mesto pa premaknimo novi zaboj.
    if (m == v[M1]) { swap(v[M2 − 1], v[M1 + 1]); swap(v[M1], v[M1 + 1]); ++M1; }
  }
  // Rekurzivno uredimo vsakega od treh delov posebej.
  if (M1 − L > 1) UrediSQuicksortom(v, L, M1 − 1);
  if (M2 − M1 > 2) UrediSQuicksortom(v, M1 + 1, M2 − 1);
  if (D − M2 > 1) UrediSQuicksortom(v, M2 + 1, D);
}

// V „v“ pripravi seznam številk vseh zabojev, urejenih po teži.
void UrediSQuicksortom(vector<Zaboj> &v)
{
  // Postavimo zaboje v naključen vrstni red.
  v.resize(n); mt19937_64 r;
  for (int i = 0; i < n; ++i) {
    int j = uniform_int_distribution<int>(0, i)(r);
    v[i] = v[j]; v[j] = i; }
  // Uredimo jih z rekurzijo.
  UrediSQuicksortom(v, 0, n − 1);
}
```

Uravnoteževanje obrabe je pri tej rešitvi seveda zelo pomembno, saj delilna zaboja pri glavnem klicu rekurzije (tistem, ki gleda celotno tabelo) nastopata v približno n izračunih mediane; zato kliče naša gornja implementacija funkcijo MedianaZU in ne MedianaZ [97 točk].

Še ena možnost, če nočemo pisati prej opisanega uravnoteževalnika obrabe, je, da pred vsakim računanjem mediane izvedemo dve zamenjavi, s katerima premaknemo oba delilna elementa na dva naključno izbrana podstavka [93 točk]; ali pa zabojev ne premeščamo šele na koncu (s podprogramom ZakljuciUrejanje, ki bi mu podali primerno urejen seznam zabojev, dobljen s podprogramom UrediSQuicksortom), pač pa že sproti [87 točk].

V primerjavi s prvotno rešitvijo (urejanje z vstavljanjem s pomočjo trisekcije na seznamu) izvede rešitev s quicksortom nekaj več izračunov mediane, ker opazovanega dela tabele ne uspe razdeliti vedno točno na tretjine, zato je globina rekurzije večja od $\log_3 n$, čeprav je v povprečju še vedno reda $O(\log n)$.

Prevedba na urejanje s primerjavo po dveh elementov. Poiščimo najprej najlažji in najtežji zaboj; to lahko storimo takole: če izračunamo mediano poljubnih treh zabojev, smo zanjo lahko prepričani, da ta zaboj ni niti najlažji od vseh niti najtežji od vseh n zabojev. Ta zaboj torej v mislih pobrišimo, spet izračunajmo mediano poljubnih treh (preostalih) zabojev, pobrišimo tudi njo in tako naprej; ko nam ostaneta le dva zaboja, vemo, da sta to najlažji in najtežji zaboj.[12]

Recimo enemu od njiju m; predpostavili bomo, da je najlažji (če pa je v resnici najtežji, bo učinek le ta, da bomo na koncu dobili padajoče namesto naraščajoče urejeno zaporedje zabojev, kar je tudi povsem sprejemljivo). Zdaj lahko poljubna dva druga zaboja, recimo a in b, primerjamo po teži tako, da izračunamo mediano zabojev m, a in b; mediana bo v tem primeru tisti izmed a in b, ki je lažji izmed teh dveh zabojev. Zdaj znamo torej primerjati dva zaboja po teži, to pa lahko podamo kot primerjalno funkcijo podprogramu sort iz C++ove standardne knjižnice, pa nam bo on uredil zaboje.

```
void UrediSStdSortom(vector<Zaboj> &v)
{
  // Poiščimo najlažji in najtežji zaboj. Najprej so kandidati vsi zaboji od 0 do n − 1.
  v.resize(n); for (Zaboj z = 0; z < n; ++z) v[z] = z;
  mt19937_64 r;
  for (int k = n − 1; k >= 2; −−k)
  {
    // Kandidati so še zaboji v[0..k]. Izberimo naključne tri in jih
    // premaknimo na konec, v v[k − 2..k].
    for (int i = 0; i < 3; ++i) swap(v[k − i], v[uniform_int_distribution(0, k − i)(r)]);
    // Tistega izmed njih, ki mediana teh treh, premaknimo v v[k].
    int m = MedianaZU(v[k − 2], v[k − 1], v[k]);
    if (m == v[k − 1]) swap(v[k − 1], v[k]);
    else if (m == v[k − 2]) swap(v[k − 2], v[k]);
  }
  // Eden od zabojev v[0] in v[1] je najlažji, eden pa najtežji in ga
  // bomo zato premaknili na konec, v v[n − 1].
  swap(v[1], v[n − 1]);
  // Uredimo vmesne zaboje.
  sort(v.begin() + 1, v.end() − 1, [&v] (int z1, int z2) {
    return MedianaZU(v[0], z1, z2) == z1; });
}
```

Seveda nas ne sme presenetiti, če porabi ta rešitev več računanj mediane kot nekatere prejšnje, saj z vsakim klicem v resnici primerja le dva zaboja namesto treh in tako dobi manj informacij. V funkciji sort iz standardne knjižnice se najverjetneje skriva quicksort, ki deli tabelo na dva dela (namesto na tri kot pri naši prej opisani rešitvi s quicksortom) in vsakega obdela rekurzivno. Pri naših poskusih je ta rešitev porabila približno $1{,}63 \cdot n \log_2 n$ računanj mediane.

Poleg tega je pri tej rešitvi zelo pomembno uporabiti uravnoteževalnik obrabe; brez tega bi bila obraba zelo neenakomerna, saj zaboj m nastopa v vsakem od

[12] Kakšna obraba nastane pri tem postopku? Če nam je ostalo še k zabojev in naključno izberemo tri od njih za naslednji izračun mediane, ima vsak kandidat verjetnost $3/k$, da bo izbran; skupaj bo torej posamezni zaboj uporabljen največ $\sum_{k=3}^{n} 3/k = 3 \ln n + O(1)$-krat (tisti, ki izpadejo prej, pa še manjkrat; največ obrabe pričakujemo pri najlažjem in najtežjem zaboju, ker tadva nikoli ne izpadeta iz množice kandidatov za naslednji izračun mediane).

$O(n \log n)$ izračunov mediane [60 točk brez uravnoteževanja obrabe, 90 z njim.]

Urejanje z zlivanjem (*merge sort*). Tudi ta postopek lahko prilagodimo za našo nalogo. Razmislimo najprej o tem, kako lahko dve krajši urejeni zaporedji (ali „četi“ (angl. *runs*), kot jim včasih pravijo v kontekstu urejanja z zlivanjem) zlijemo v eno daljše. Pri urejanju z medianami ne moremo ločiti med primerom, ko je neko zaporedje urejeno naraščajoče, in primerom, ko je urejeno padajoče. Zato se lahko zgodi, da je eno od naših dveh krajših zaporedij urejeno naraščajoče, drugo pa padajoče. Ta primer lahko odkrijemo z največ tremi računanji mediane in po potrebi eno od zaporedij obrnemo, tako da sta potem obe urejeni na enak način: naj bo z_1 (oz. z_2) zaboj na začetku prvega (oz. drugega) zaporedja, k_1 (oz. k_2) pa tisti na koncu; predpostavimo, da je z_1 lažji od k_1; v mislih si lahko predstavljamo, da nam tadva zaboja razdelita vse druge zaboje na tri skupine: (1) tisti, ki so lažji od z_1; (2) tiste, ki so med z_1 in k_1; (3) tiste, ki so težji od k_1. Za vsakega od z_2 in k_2 poglejmo (s po enim izračunom mediane), v katero od teh treh skupin spada. Če z_2 spada v lažjo skupino kot k_2, je vse v redu; če v težjo, je treba drugo zaporedje obrniti; če pa padeta oba v isto skupino, lahko izračunamo še mediano zabojev z_1, z_2 in k_2 in iz rezultata vidimo, ali bo treba drugo zaporedje obrniti ali ne.[13]

Recimo torej zdaj, da imamo dve zaporedji, obe urejeni naraščajoče (ali obe padajoče), in da ju hočemo zliti. Označimo i-ti zaboj v prvem zaporedju z a_i, v drugem pa z b_i. Izračunajmo mediano zabojev a_1, a_2 in b_1; če je to b_1, mora veljati $a_1 < b_1 < a_2$ in lahko v izhodno zaporedje premaknemo a_1 in b_1 (kajti tako a_2 kot b_2 sta težja od b_1, zato so tudi vsi kasnejši zaboji težji in je prav, da stojita a_1 in b_1 na začetku izhodnega zaporedja); če je mediana a_1, mora veljati $b_1 < a_1 < a_2$ in lahko v izhodno zaporedje premaknemo b_1 (kajti a_1 je težji od njega, zato pa so tudi vsi kasnejši a_i težji od b_1 in je prav, da stoji b_1 na začetku izhodnega zaporedja); če pa je mediana a_2, mora veljati $a_1 < a_2 < b_1$ in lahko v izhodno zaporedje premaknemo a_1 in a_2 (kajti b_1 je težji od a_2, zato pa so tudi vsi kasnejši b_i težji od a_2 in je prav, da stojita a_1 in a_2 na začetku izhodnega zaporedja).

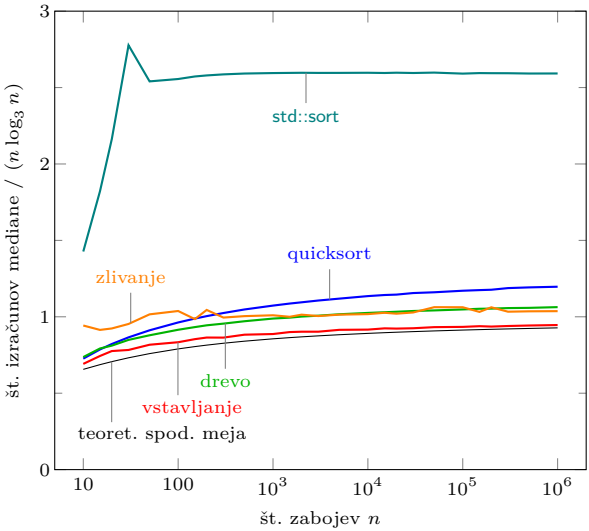
Tako nadaljujemo in počasi prestavljamo zaboje iz vhodnih zaporedij v izhodno, dokler ne pridemo do konca obeh vhodnih zaporedij. Če ostane v prvem le še en element, v drugem pa vsaj dva, lahko razmišljamo enako kot v prejšnjem odstavku, le njuni vlogi se zamenjata. Če pa ostane v vsakem po en element, lahko izračunamo mediano teh dveh ter zadnjega elementa, ki smo ga pred njima premaknili v izhodno zaporedje (ta je gotovo lažji od njiju); ta mediana nam bo povedala, kateri od obeh preostalih elementov je lažji in ga moramo preseliti v izhodno zaporedje najprej, potem pa še drugega.

```
// Zlije četi v[od1..do1 − 1] in v[od2..do2 − 1] in shrani rezultat v „w“ od indeksa od3
// naprej. Funkcija predpostavlja, da ima vsaj ena od vhodnih čet vsaj dva zaboja.
void Zlij(vector<Zaboj> &v, int od1, int do1, int od2, int do2, vector<Zaboj> &w, int od3)
{
  // Najprej poskrbimo, da bosta obe vhodni četi urejeni enako (obe naraščajoče ali
  // obe padajoče). Komentarji spodaj so napisani, kot da je prva četa naraščajoča.
  if (do1 − od1 >= 2 && do2 − od2 >= 2)
```

[13] Če sta z_2 in k_2 oba v prvi skupini, torej oba lažja od z_1, bo mediana teh treh zabojev bodisi z_2 bodisi k_2; če bo mediana z_2, to pomeni, da je z_2 težji od k_2 in bo treba drugo zaporedje obrniti, sicer pa ga ne bo treba. — Razmislek za primer, ko sta z_2 in k_2 oba v drugi ali tretji skupini in sta torej oba težja od z_1, je analogen.

```
  {
    Zaboj z1 = v[od1], k1 = v[do1 − 1], z2 = v[od2], k2 = v[do2 − 1];

    // Zaboja z1 in k1 razdelita številsko os na tri intervale: 0 = manjši od z1;
    // 1 = med z1 in k1; 2 = večji od k1. Poglejmo, na kateri interval padeta z2 in k2.
    Zaboj m = MedianaZU(z1, k1, z2); int m1 = (m == z1) ? 0 : (m == k1) ? 2 : 1;
    m = MedianaZU(z1, k1, k2); int m2 = (m == z1) ? 0 : (m == k1) ? 2 : 1;

    // Če k2 pade na zgodnejši interval kot z2, je treba drugo četo obrniti.
    bool obrni2 = (m1 > m2);

    // Če pa ležita z2 in k2 na istem intervalu, leži tam cela druga četa.
    // Preverili bomo, ali je k2 < z2 — potem je treba drugo četo obrniti.
    // Če sta oba na levem intervalu (m1 == 0), si lahko pomagamo z dejstvom,
    // da sta oba manjša od z1, sicer pa z dejstvom, da sta oba večja od z1.
    if (m1 == m2) obrni2 = (MedianaZU(z2, k2, z1) == (m1 == 0 ? z2 : k2));

    // Če je treba, obrnimo zdaj drugo četo.
    if (obrni2) for (int i = od2, j = do2 − 1; i < j; ) swap(v[i++], v[j−−]);
  }
  // Zlijmo četi.
  int i1 = od1, i2 = od2, i3 = od3;
  while (i1 < do1 || i2 < do2)
  {
    // Če smo na koncu ene čete, moramo le še kopirati zaboje druge na izhod.
    if (i2 == do2) w[i3++] = v[i1++];
    else if (i1 == do1) w[i3++] = v[i2++];

    // Če imamo pri prvi četi še vsaj dva zaboja, izračunajmo mediano naslednjih dveh
    // zabojev prve čete in naslednjega zaboja druge. Najmanjši zaboj lahko skopiramo
    // na izhod, poleg njega pa še mediano, če ne prihaja iz iste čete kot tretji zaboj.
    else if (i1 + 1 < do1) {
      int m = MedianaZU(v[i1], v[i1 + 1], v[i2]);
      if (m == v[i1]) w[i3++] = v[i2++]; // v[i1] še ne smemo skopirati na izhod, kajti
                                         // morda je tudi v[i2 + 1] manjša od njega.
      else if (m == v[i1 + 1]) { w[i3++] = v[i1++]; w[i3++] = v[i1++]; }
      else { w[i3++] = v[i1++]; w[i3++] = v[i2++]; } }

    // Podobno obdelamo primer, ko imamo pri drugi četi še vsaj dva zaboja
    // (pri prvi pa le enega).
    else if (i2 + 1 < do2) {
      int m = MedianaZU(v[i2], v[i2 + 1], v[i1]);
      if (m == v[i2]) w[i3++] = v[i1++]; // v[i2] še ne smemo skopirati na izhod, kajti
                                         // morda je tudi v[i1 + 1] manjša od njega.
      else if (m == v[i2 + 1]) { w[i3++] = v[i2++]; w[i3++] = v[i2++]; }
      else { w[i3++] = v[i2++]; w[i3++] = v[i1++]; } }

    // Ostane še primer, ko imamo pri vsaki četi natanko en zaboj.
    // Ker je imela prvotno vsaj ena četa vsaj dva zaboja, to zdaj pomeni,
    // da smo vsaj en zaboj že skopirali na izhod. Izračunajmo torej mediano
    // tega slednjega in preostalih dveh zabojev naših vhodnih čet.
    else {
      int m = MedianaZU(w[i3 − 1], v[i1], v[i2]);
      if (m == v[i1]) { w[i3++] = v[i1++]; w[i3++] = v[i2++]; }
      else { w[i3++] = v[i2++]; w[i3++] = v[i1++]; } }
  }
}
```

Zdaj torej znamo zlivati krajša urejena zaporedja v daljša. Postopek poženemo tako, da razdelimo naše zaporedje n zabojev na zaporedja s po dvema ali tremi zaboji;



Primerjava različnih rešitev naloge „Urejanje z medianami“.

Graf kaže število računanj mediane v odvisnosti od števila zabojev n. Zaradi preglednosti ni prikazano število računanj mediane sámo, pač pa njegov količnik po deljenju z $n \log_3 n$. Rezultati pri vsakem n so povprečja po več poskusih z naključnim začetnim vrstnim redom zabojev (vsaj 100 poskusov pri vsakem n, pri manjših tudi več).

Od spodaj gor so prikazane: (1) teoretična spodnja meja ($\log_3 n!$ izračunov mediane); (2) urejanje z vstavljanjem, kjer trisekcija vedno deli točno na tretjine (ta rešitev porabi $O(n^2)$ časa pri različici s seznamom oz. $O(n(\log n)^2)$ z drevesom); (3) urejanje z vstavljanjem, kjer se trisekcija prilagodi strukturi drevesa, tako da je časovna zahtevnost le $O(n \log n)$; (4) urejanje z zlivanjem; (5) hitro urejanje oz. quicksort (ki deli trenutno opazovani del seznama na tri dele); (6) rešitev, ki poišče najlažji in najtežji zaboj ter nato pokliče std::sort (pri tej krivulji je čudno obnašanje pri majhnih n posledica tega, da implementacija std::sort v standardni knjižnici, ki smo jo uporabljali, pri manj kot 32 elementih preklopi s quicksorta na urejanje z vstavljanjem).

---

zaporedje dveh zabojev je že samo po sebi urejeno, pri zaporedju s tremi zaboji pa lahko z enim izračunom mediane ugotovimo, katerega je treba postaviti na drugo mesto v zaporedju. Zdaj imamo torej urejena zaporedja dveh ali treh zabojev; nato jih zlivamo po dve in dve v vse daljša in daljša zaporedja, dokler ne ostane eno samo zaporedje z vsemi n zaboji.

```
void UrediZZlivanjem(vector<Zaboj> &v)
{
  // Zaboje bomo pregledovali v naključnem vrstnem redu.
  v.resize(n);
  for (int i = 0; i < n; ++i) {
    int j = uniform_int_distribution(0, i)(r);
    v[i] = v[j]; v[j] = i; }
  if (n <= 2) return;
  // Začnimo s četami dolžine 3.
  for (int i = 0; i + 3 <= n; i += 3) {
```

```
    int m = MedianaZU(v[i], v[i + 1], v[i + 2]);
    if (m == v[i]) swap(v[i], v[i + 1]);
    else if (m == v[i + 2]) swap(v[i + 1], v[i + 2]); }
  // Zlivajmo čete v daljše.
  vector<Zaboj> vNovi(n);
  for (int d = 3; d < n; d *= 2) { // d = dolžina čet v „v“
    for (int i = 0; i < n; i += 2 * d) {
      // Če je četa, ki se začne na indeksu i, zadnja, jo moramo le
      // skopirati v izhodno tabelo.
      if (i + d >= n) for (int j = i; j < n; ++j) vNovi[j] = v[j];
      // Sicer bomo zlili četi z začetkom na i in na i + d.
      // Če bi imela druga le en element, preselimo vanjo še zadnji element prve čete.
      else {
        int j = min(i + d, n - 2);
        Zlij(v, i, j, j, min(j + d, n), vNovi, i); } }
    swap(vNovi, v); }
}
```

Tudi ta postopek izvede $O(n \log n)$ računanj mediane [90 točk brez uravnoteževanja obrabe, 100 z njim].

Primerjavo števila računanj mediane (v odvisnosti od števila zabojev n) za različne tu opisane rešitve kaže graf na str. 98.

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA SŠ

1. Cikcakasti nizi

Vhodni niz lahko pregledujemo v zanki, znak za znakom. Zapomnimo si prvi znak in poglejmo, koliko enakih znakov še pride za njim; tako dobimo dolžino prvega kosa. Nato naredimo podobno od naslednjega znaka naprej in dobimo dolžino drugega kosa; in tako dalje. Dolžine zadnjih treh kosov si zapomnimo v spremenljivkah, recimo d_1, d_2 in d_3. Zanje mora veljati bodisi $d_1 < d_2 > d_3$ bodisi $d_1 > d_2 < d_3$, sicer niz ni cikcakast (in lahko s pregledovanjem niza takoj končamo). Če pridemo do konca niza, ne da bi preverjanje tega pogoja kdaj spodletelo, lahko zaključimo, da niz je cikcakast. Paziti moramo še na robni primer: če ima niz natanko dva enako dolga kosa, bo glavna zanka prišla do konca, niz pa v resnici ni cikcakast. Zato na koncu preverimo, če sta d_1 in d_2 različna ali pa če je $d_1 = 0$ — to slednje poskrbi, da bomo kot cikcakaste pravilno prepoznali tudi nize z le enim kosom ali celo z nobenim (torej prazen niz). Oglejmo si implementacijo te rešitve v jeziku C++:

```
bool JeCikcakast(const char *s)
{
  int d1 = 0, d2 = 0; // Dolžina prejšnjih dveh kosov.
  while (*s) // Preglejmo niz vse do konca.
  {
    char c = *s++; // Iz tega znaka je sestavljen trenutni kos.
    // Poglejmo, kolikokrat se znak c zdaj še ponovi.
    int d3 = 1; while (*s == c) ++s, ++d3;

    if (d1 > 0) // Ali smo že videli vsaj tri kose?
      // Preverimo, ali dolžine zadnjih treh kosov ustrezajo pogojem.
      if (! (d1 < d2 && d2 > d3) && ! (d1 > d2 && d2 < d3)) return false;

    d1 = d2; d2 = d3; // Zapomnimo si dolžini zadnjih dveh kosov.
  }
  return d1 == 0 || d1 != d2; // Če je do konca niza vse v redu, je niz cikcakast.
}
```

Lahko bi tudi brali niz sproti s standardnega vhoda, šteli zaporedne enake znake in preverjali, ali je niz cikcakast; tako sploh ni nujno imeti celotnega niza hkrati v glavnem pomnilniku:

```
bool JeCikcakast2()
{
  int d1 = 0, d2 = 0; // Dolžina prejšnjih dveh kosov.
  int c = getchar(); // Preberimo prvi znak.
  while (c >= 'a' && c <= 'z') // Preglejmo niz vse do konca.
  {
    // Naslednji znak je c; kolikokrat se še ponovi?
    int d3 = 1, cc; while ((cc = getchar()) == c) ++d3;

    if (d1 != 0) // Ali smo že videli vsaj tri kose?
      // Preverimo, ali dolžine zadnjih treh kosov ustrezajo pogojem.
      if (! (d1 < d2 && d2 > d3) && ! (d1 > d2 && d2 < d3)) return false;

    d1 = d2; d2 = d3; c = cc; // Pripravimo se na naslednji kos.
  }
  return d1 == 0 || d1 != d2; // Če je do konca niza vse v redu, je niz cikcakast.
}
```

Zapišimo prvo od gornjih dveh rešitev še v pythonu:

```
def JeCikcakast(s):
    d1 = 0; d2 = 0 # Dolžina prejšnjih dveh kosov.
    i = 0; n = len(s)
    while i < n: # Preglejmo niz vse do konca.
        c = s[i]; i += 1; d3 = 1 # Trenutni kos je iz znaka c.
        # Poglejmo, kolikokrat se znak c zdaj še ponovi.
        while i < n and s[i] == c: i += 1; d3 += 1
        if d1 > 0: # Ali smo že videli vsaj tri kose?
            # Preverimo, ali dolžine zadnjih treh kosov ustrezajo pogojem.
            if not (d1 < d2 > d3 or d1 > d2 < d3): return False
        d1 = d2; d2 = d3 # Zapomnimo si dolžini zadnjih dveh kosov.
    return d1 == 0 or d1 != d2 # Če je do konca niza vse v redu, je niz cikcakast.
```

Malo za šalo, malo zares pa je tu še rešitev za ljubitelje regularnih izrazov:

```
import re
def JeCikcakast(s):
    d1 = 0; d2 = 0 # Dolžina prejšnjih dveh kosov.
    # Preglejmo vse kose niza.
    for kos in re.finditer(r"(.)\1*", s):
        d3 = kos.end() - kos.start() # Dolžina trenutnega kosa.
        # Če smo že videli vsaj tri kose, preverimo, ali dolžine zadnjih
        # treh kosov ustrezajo pogojem.
        if d1 > 0 and not (d1 < d2 > d3 or d1 > d2 < d3): return False
        d1 = d2; d2 = d3 # Zapomnimo si dolžini zadnjih dveh kosov.
    return d1 == 0 or d1 != d2 # Če je do konca niza vse v redu, je niz cikcakast.
```

V regularnem izrazu se pika lahko ujame s poljubnim znakom, ker pa smo jo dali v oklepaje, se lahko kasneje na ta znak sklicujemo z `\1`; na koncu smo dali še zvezdico, tako da naš izraz pobere vse nadaljnje pojavitve istega znaka — to pa je ravno en kos, na kakršne hočemo pri tej nalogi razbiti vhodni niz. Ta regularni izraz je eleganten in rešitev deluje pravilno, vendar je imela pri naših poskusih to slabost, da je porabila približno 160- do 180-krat toliko dodatnega pomnilnika, kolikor je dolg posamezni kos v nizu. Ko je bil na primer s sestavljen iz 5 milijonov samih `a`-jev, je program porabil kar 930 MB pomnilnika. Ta potrata je verjetno povezana s tem, kako je v knjižnici za regularne izraze implementirano sklicevanje nazaj (pri `\1`).

Naš naslednji poskus je bil regularni izraz, ki uporablja sklicevanje nazaj le zato, da določi konec kosa:

```
    for kos in re.finditer(r"(.).*?(?!\1)", s):
```

Tu torej za prvim znakom, ki si ga zapomnimo (zato je pika v oklepajih), poberemo še poljubne nadaljnje znake, vendar čim manj (zato `*?` namesto le `*`), ustavimo pa se na mestu, kjer se nadaljevanje niza *ne ujema* z izrazom znotraj `(?!...)`. V našem primeru imamo tam `\1`, torej se ustavimo ravno takrat, ko naslednji znak ne sodi več v trenutni kos. Pri naših poskusih s tem izrazom je bila poraba dodatnega

pomnilnika neodvisna od dolžine niza s, kar je dobro; je pa bil ta izraz še vedno razmeroma počasen: na nizu, sestavljenem iz 10^9 znakov `a`, se je program izvajal okrog 15 sekund.

Da se izognemo sklicevanju nazaj, zaradi katerega je iskanje pojavitev regularnega izraza počasnejše, se lahko opremo na dejstvo, da vnaprej poznamo vse možne znake, ki se utegnejo pojaviti v naših nizih, namreč male črke od `a` do `z`. Zato lahko sestavimo še preprostejši, a še manj eleganten izraz: `a+|b+|c+|...|z+`, kjer so eksplicitno naštete vse možne oblike kosov.

```
import re, string
⋮
for kos in re.finditer("+|".join(string.ascii_lowercase) + "+", s):
```

Pri tem izrazu je bila ne le poraba dodatnega pomnilnika neodvisna od dolžine niza s, pač pa je bilo tudi naštevanje kosov veliko hitrejše; niz, sestavljen iz 10^9 znakov `a`, je program s tem izrazom obdelal v manj kot pol sekunde. Nauk te zgodbe je torej, da je potrebne pri sestavljanju regularnih izrazov nekaj pazljivosti.

2. Histogram

Označimo višine stolpcev od leve proti desni s $h_1, h_2, \ldots, h_n$. Do katere višine potem pride voda nad stolpcem i? Naj bo $L_i := \max\{h_1, h_2, \ldots, h_i\}$ višina najvišjega stolpca levo od i-tega (vključno z njim samim) in podobno $D_i := \max\{h_i, h_{i+1}, \ldots, h_n\}$ višina najvišjega stolpca desno od i-tega. Potem gladina vode na stolpcu i pride ravno do višine $g_i := \min\{L_i, D_i\}$; dokler je namreč voda nižja od te višine, jo z leve strani zadržuje neki stolpec višine L_i, z desne pa neki stolpec višine D_i in bo zato rasla še naprej; ko pa doseže to višino, je vsaj na eni strani ne zadržuje nič več in se lahko tam razlije čez rob najvišjega stolpca v tisti smeri in sčasoma odteče s histograma. Tako torej vidimo, da se (za vsak i) na stolpcu višine h_i nabere voda do višine g_i, kar pomeni $g_i - h_i$ enot vode; to moramo sešteti po vseh i (od 1 do n), pa dobimo odgovor, po katerem sprašuje naloga.

Oglejmo si implementacijo te rešitve v C++. Vrednosti L_i lahko računamo v zanki od leve proti desni, saj velja zveza $L_i = \max\{L_{i-1}, h_i\}$; podobno lahko računamo D_i v zanki od desne proti levi. Ko imamo oboje, pa gremo lahko še enkrat po stolpcih, računamo količino vode in jo seštevamo.

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

int Histogram(const vector<int> &h)
{
  // Izračunajmo za vsak stolpec višino najvišjega stolpca levo in desno od njega.
  int n = h.size(); vector<int> L(n), D(n);
  for (int i = 0; i < n; ++i) L[i] = max(h[i], (i == 0) ? 0 : L[i - 1]);
  for (int i = n - 1; i >= 0; --i) D[i] = max(h[i], (i == n - 1) ? 0 : D[i + 1]);
  // Zdaj lahko za vsak stolpec določimo višino vode.
  int voda = 0;
  for (int i = 0; i < n; ++i) voda += min(L[i], D[i]) - h[i];
  return voda;
}
```

Ali v pythonu:

```
def Histogram(h):
    # Izračunajmo za vsak stolpec višino najvišjega stolpca levo in desno od njega.
    n = len(h); L = [0] * n; D = [0] * n
    for i in range(n): L[i] = max(h[i], 0 if i == 0 else L[i − 1])
    for i in range(n − 1, −1, −1): D[i] = max(h[i], 0 if i == n − 1 else D[i + 1])
    # Zdaj lahko za vsak stolpec določimo višino vode.
    return sum(min(L[i], D[i]) − h[i] for i in range(n))
```

Še ena možnost pa je, da najprej poiščemo najvišji stolpec; recimo, da je to stolpec m (z višino h_m).[14] Potem za stolpce levo od njega, torej $i < m$, vemo, da najvišji stolpec levo od i ni višji od najvišjega stolpca desno od i (kajti tam je tudi najvišji stolpec sploh), torej gladino vode pri i določa L_i in ne D_i. Desno od najvišjega stolpca, pri $i > m$, pa je ravno obratno. Dovolj je torej, če vrednosti L_i izračunamo le za $i < m$ in jih tudi kar takoj uporabimo kot gladino vode pri teh i, podobno pa potem še vrednosti D_i za $i > m$. Zato nam ni treba hraniti vseh L_i in D_i v tabelah oz. vektorjih, kot je to počela prejšnja rešitev; porabimo le $O(1)$ dodatnega pomnilnika namesto $O(n)$.

```
int Histogram2(const vector<int> &h)
{
    // Naj bo m indeks najvišjega stolpca.
    int n = h.size(), m = 0, voda = 0;
    for (int i = 1; i < n; ++i) if (h[i] > h[m]) m = i;
    // Pojdimo od levega roba do najvišjega stolpca. Gladino vode določa
    // višina najvišjega doslej obiskanega stolpca.
    for (int i = 0, najvisji = 0; i < m; ++i) {
        najvisji = max(najvisji, h[i]); // zdaj je najvisji == L[i] <= D[i] == h[m]
        voda += najvisji − h[i]; }
    // Podobno velja tudi, če gremo od desnega roba do najvišjega stolpca.
    for (int i = n − 1, najvisji = 0; i > m; −−i) {
        najvisji = max(najvisji, h[i]); // zdaj je najvisji == D[i] <= L[i] == h[m]
        voda += najvisji − h[i]; }
    return voda;
}
```

Še v pythonu:

```
def Histogram2(h):
    # Naj bo m indeks najvišjega stolpca.
    n = len(h); m = 0
    for i in range(n):
        if h[i] > h[m]: m = i
    # Pojdimo od levega roba do najvišjega stolpca. Gladino vode določa
    # višina najvišjega doslej obiskanega stolpca.
    najvisji = 0; voda = 0
    for i in range(m):
        najvisji = max(najvisji, h[i]) # zdaj je najvisji == L[i] <= D[i] == h[m]
        voda += najvisji − h[i]
```

[14] Če obstaja več enako visokih najvišjih stolpcev, je vseeno, katerega vzamemo za m. Naša naloga tako ali tako zagotavlja, da bodo stolpci različno visoki, torej bo najvišji stolpec en sam, vendar bi tu opisana rešitev delovala tudi v primerih, ko je lahko več stolpcev enako visokih.

```
# Podobno velja tudi, če gremo od desnega roba do najvišjega stolpca.
najvisji = 0
for i in range(n − 1, m, −1):
    najvisji = max(najvisji, h[i]) # zdaj je najvisji == D[i] <= L[i] == h[m]
    voda += najvisji − h[i]
return voda
```

Preden si ogledamo še eno rešitev, razmislimo malo o obnašanju zaporedij L, D in g. Zaporedje L, če ga gledamo od leve proti desni, ima obliko naraščajočega „stopnišča“: ko pridemo do novega stolpca, ki je najvišji doslej, poskoči L_i na višino tega stolpca, potem pa na tej višini tudi ostane do naslednjega višjega stolpca. Pri najvišjem stolpcu, $i = m$, doseže L_i višino h_m in na njej ostane do konca. Podobno je tudi zaporedje D stopničaste oblike, le da tam višina stopnic raste od desne proti levi; če pa ga gledamo od leve proti desni, je D padajoče zaporedje. Zaporedji L in D se „srečata“ pri najvišjem stolpcu, torej m, kajti tam je $L_m = D_m = h_m$; levo od njega velja $g_i = L_i \leq D_i = h_m$, desno pa $g_i = D_i \leq L_i = h_m$. (Voda, ki pade na histogram levo od m, se sceja od desne proti levi in sčasoma odteče čez levi rob prvega stolpca; voda pa, ki pade desno od m, se sceja od leve proti desni in sčasoma odteče čez desni rob zadnjega (n-tega) stolpca.) Zaporedje g, ki opisuje obliko histograma z vodo vred, si lahko torej predstavljamo kot sestavljeno najprej iz vseh stopnic naraščajočega zaporedja L, nato pa še iz vseh stopnic padajočega zaporedja D; le stopnico višine h_m, ki se pri L razteza od $i = m$ do $i = n$, pri D pa od $i = 1$ do $i = m$, moramo pri g omejiti le na stolpec m.

Nalogo lahko zdaj rešimo tudi tako, da postopoma dodajamo v histogram stolpce (od leve proti desni) in po vsakem izračunamo, koliko dodatne vode se zaradi njega nabere. Kaj se zgodi z zaporedjema L in D, če histogramu s stolpci $h_1, \ldots, h_{n-1}$ na desni dodamo še stolpec višine h_n? Vrednosti $L_1, \ldots, L_{n-1}$ niso odvisne od h_n in se ne bodo nič spremenile; to tudi pomeni, da se v levem delu histograma (do najvišjega stolpca), kjer je $g_i = L_i$, tudi g_i ne bo spremenil; v zaporedju $D_1, \ldots, D_{n-1}$ pa se zdaj lahko zadnjih nekaj vrednosti poveča: tisti D_i, ki so bili prej manjši od h_n, postanejo zdaj enaki h_n. Spomnimo se, da ima zaporedje D obliko padajočega stopnišča; učinek dodajanja novega stolpca je torej ta, da s konca tega stopnišča izginejo tiste stopnice, ki so bile nižje od h_n, namesto njih pa se tam pojavi nova stopnica višine h_n, ki se razteza do vključno novega stolpca n. (To si lahko predstavljamo tudi fizikalno: novi stolpec prepreči odtekanje vode čez desni rob histograma, dokler njena gladina ne naraste do višine novega stolpca.)

Stopnišče D je torej koristno predstaviti s skladom, kjer so stopnice urejene od leve proti desni (in s tem po padajočih višinah); pri tem pa za prvo, najvišjo stopnico (višine h_m), vzemimo, kot da ima širino 1 in obsega le najvišji stolpec sam, ne pa še vseh levo od njega. Ko dodamo nov stolpec, pobrišemo z vrha sklada tiste stopnice, ki so nižje od njega, in dodamo namesto njih novo stopnico v višini novega stolpca; široka pa je nova stopnica toliko, da pokrije tudi vse pravkar pobrisane. Pri brisanju stopnic tudi ni težko računati, koliko dodatne vode se bo na njih nabralo zdaj, ko bo njena gladina narasla do višine novega stolpca. Poseben primer nastopi, če je novi stolpec najvišji doslej (ta primer prepoznamo med drugim po tem, da pri njem pobrišemo čisto vse dosedanje stopnice); takrat voda ne odteka čez desni rob histograma, ampak samo čez levega, in njene gladine levo od novega stolpca ne določa višina tega novega stolpca, ampak višina bivšega najvišjega stolpca. Oglejmo

si še implementacijo te rešitve v C++:

```
#include <stack>

int Histogram3(const vector<int> &h)
{
  struct Stopnica { int sirina, visina; };
  stack<Stopnica> sklad = {}; // padajoče stopnišče D
  int n = h.size(), voda = 0, najvisji = 0;
  for (int i = 0; i < n; ++i)
  {
    int sirina = 0; // širina nove stopnice, ki ji bo v D pripadal novi stolpec i
    while (! sklad.empty() && sklad.top().visina <= h[i])
    {
      // Novi stolpec je vsaj tako visok kot Z (zadnja stopnica na skladu), zato se bo voda
      // na Z dvignila do višine novega stolpca; stopnica Z bo torej postala del nove
      auto Z = sklad.top(); sklad.pop();              // stopnice, ki se konča s stolpcem i.
      sirina += Z.sirina;
      voda += Z.sirina * (h[i] - Z.visina); // toliko dodatne vode se nabere nad bivšo
    }                                                                    // stopnico Z
    // Če je novi stolpec najvišji doslej, gladina levo od njega ne bo šla do njegove višine
    // (kot smo računali v gornji zanki), pač pa do višine drugega najvišjega stolpca doslej.
    // V tem primeru bo novi stolpec sam tvoril (edino) stopnico širine 1.
    if (sklad.empty()) voda -= sirina * (h[i] - najvisji), najvisji = h[i], sirina = 0;
    // Dodajmo na sklad stopnico, ki se konča z novim stolpcem.
    sklad.push({sirina + 1, h[i]});
  }
  return voda;
}
```

In v pythonu:

```
def Histogram3(h):
  sklad = [] # padajoče stopnišče D; pari (širina, višina)
  voda = 0; najvisji = 0
  for hi in h:
    sirina = 0 # širina nove stopnice, ki ji bo v D pripadal novi stolpec hi
    while sklad and sklad[-1][1] <= hi:
      # Novi stolpec je vsaj tako visok kot Z (zadnja stopnica na skladu), zato se bo voda
      # na Z dvignila do višine novega stolpca; stopnica Z bo torej postala del nove
      (zSirina, zVisina) = sklad[-1]; sklad.pop()      # stopnice, ki se konča s stolpcem i.
      sirina += zSirina
      voda += zSirina * (hi - zVisina) # toliko dodatne vode se nabere nad bivšo
                                                                    # stopnico Z
    # Če je novi stolpec najvišji doslej, gladina levo od njega ne bo šla do njegove višine
    # (kot smo računali v gornji zanki), pač pa do višine drugega najvišjega stolpca doslej.
    # V tem primeru bo novi stolpec sam tvoril (edino) stopnico širine 1.
    if not sklad: voda -= sirina * (hi - najvisji); najvisji = hi; sirina = 0
    # Dodajmo na sklad stopnico, ki se konča z novim stolpcem.
    sklad.append((sirina + 1, hi))
  return voda
```

Razmislimo zdaj še o težji različici naloge, ki jo omenja opomba pod črto v besedilu naloge. Tu se torej omejimo na histograme, pri katerih tvorijo višine stolpcev neko

permutacijo števil $1, 2, \ldots, n$ in je n majhen, $n \leq 30$; zanima pa nas, v koliko od teh $n!$ histogramov se nabere natanko a enot vode. Pri histogramih je lažje kot o količini vode razmišljati o skupni površini — vsoti višin vseh stolpcev (to pa je $1 + 2 + \ldots + n = n(n+1)/2$) in vode na njih. Rečemo lahko torej, da nas zanima, pri koliko histogramih je skupna površina enaka $\tilde{a} := a + n(n+1)/2$.

Videli smo že, da je histogram, ko se v njem nabere voda — torej zaporedje g — sestavljen iz zaporedja naraščajočih stopnic (zaporedje L), ki mu sledi najvišji stolpec (v našem primeru ima ta višino n), za njim pa pride še zaporedje padajočih stopnic (zaporedje D). Tako lahko torej govorimo o „levih“ in „desnih“ stopnicah; stolpca višine n pa ne bomo šteli za stopnico. Posamezna stopnica je skupina 1 ali več stolpcev, med katerimi je najvišji tisti, ki je najbolj zunanji (torej najbolj levi, če gre za levo stopnico, in najbolj desni, če gre za desno); ta določa višino stopnice, ostali stolpci pa so nižji (in se na njih nabere voda do višine tistega najvišjega).

Naj bo $f(x, y, \alpha)$ število histogramov, ki imajo v stopnicah skupno x stolpcev (različno visokih), najvišji med njimi (in s tem najvišja stopnica) ima višino y, površina histograma (z vodo vred) pa je α. Ne pozabimo, da ima poleg x stolpcev v stopnicah histograma tudi najvišji stolpec višine n, ki ga ne štejemo za del nobene stopnice. Rezultat, po katerem sprašuje naloga, je potem $f(n-1, n-1, \tilde{a})$. Razmislimo, kako bi funkcijo f računali v splošnem.

Histogram, kakršen nas zanima za $f(x, y, \alpha)$, lahko dobimo tako, da vzamemo neki histogram z manj stolpci, recimo $\hat{x} < x$, in nižjo najvišjo stopnico, recimo $\hat{y} < y$, in dodamo vanj novo stopnico širine $x - \hat{x}$ in višine y. Taka nova stopnica ima površino $(x - \hat{x}) \cdot y$, torej je moral imeti stari histogram površino $\hat{\alpha} := \alpha - (x - \hat{x}) \cdot y$. Na koliko načinov pa lahko dodamo takšno novo stopnico? Nova stopnica vsekakor vsebuje stolpec višine y, ki stoji na zunanjem robu te stopnice in določa nivo vode v njej; poleg njega pa vsebuje še $x - \hat{x} - 1$ stolpcev, nižjih od y. V poštev za višine teh nižjih stolpcev pridejo števila od 1 do $y - 1$, razen tistih $\hat{x}$ višin, ki so že nastopile kot višine stolpcev starega histograma. Tako moramo torej izmed $y - \hat{x} - 1$ višin izbrati $x - \hat{x} - 1$ (različnih) višin in jih postaviti v neki konkreten vrstni red (kajti različen vrstni red teh stolpcev nam bo dal različne histograme, četudi bo gladina vode pri vseh enaka); to pa lahko naredimo na $(y - \hat{x} - 1)!/(y - x)! = (y - \hat{x} - 1)(y - \hat{x} - 2) \cdots (y - x + 1)$ načinov (variacije brez ponavljanja). Poleg tega si lahko izberemo, ali bo nova stopnica leva ali desna, zato se nam število možnosti še podvoji. Da bomo dobili vse možne nove histograme, moramo sešteti vse možnosti glede tega, kako širok je bil histogram pred dodajanjem najvišje stopnice in kako visoka je bila takrat najvišja stopnica; potrebujemo torej vsoto po vseh $\hat{x}$ in $\hat{y}$, pri čemer seveda pazimo na to, da mora biti $\hat{x} \leq \hat{y}$, saj drugače sploh ne bi bilo dovolj različnih višin za vse stolpce starega histograma. Tako smo dobili formulo:

$$f(x, y, \alpha) = \sum_{\hat{x}=0}^{x-1} \sum_{\hat{y}=\hat{x}}^{y-1} 2 \cdot \frac{(y - \hat{x} - 1)!}{(y - x)!} \cdot f(\hat{x}, \hat{y}, \alpha - (x - \hat{x}) \cdot \hat{y}).$$

Robni primer je histogram brez stopnic; tega sestavlja le stolpec višine n, njegova površina pa je zato tudi n; tako imamo $f(0, 0, n) = 1$, pri vseh drugih površinah $\alpha \neq n$ pa je $f(0, 0, \alpha) = 0$. Še en robni primer je seveda tudi $f(x, y, \alpha) = 0$ za $\alpha \leq 0$, saj je površina histograma gotovo pozitivna.

Nalogo lahko torej rešimo z dinamičnim programiranjem: vrednosti funkcije f računamo sistematično s petimi gnezdenimi zankami, namreč po x, y, α, $\hat{x}$ in $\hat{y}$. Za vsako od teh količin je $O(n)$ možnosti, le za α je $O(n^2)$ možnosti (površina histograma z vodo vred je največ $(n-1)^2+n$, do česar pride takrat, ko na začetku in na koncu histograma stojita najvišja dva stolpca, visoka n oz. $n-1$, med njima pa je še $n-2$ nižjih stolpcev, kjer je gladina vode $n-1$). Vrednosti funkcije f moramo računati po naraščajočih x, y in α in si jih zapisovati v neko tabelo, da jih bomo imeli pri roki, ko jih bomo kasneje spet potrebovali. Tudi pri izrazih oblike $a!/b!$ v gornji formuli je za a in b le $O(n)$ možnosti, zato jih lahko vse potabeliramo vnaprej v $O(n^2)$ časa. Tako imamo pri vsaki kombinaciji vrednosti x, y, α, $\hat{x}$ in $\hat{y}$ le $O(1)$ dela; časovna zahtevnost te rešitve je torej $O(n^6)$, kar je pri tako majhnih n, kakršni nas zanimajo tukaj (rekli smo, da je $n \leq 30$), še dovolj hitro.

3. Naredimo hitro testiranje zares hitro!

Nobene koristi ni od tega, da bi na kakšni točki za testiranje med dvema testiranjema zevala časovna vrzel, ko ne bi nekaj časa testirali nikogar; karkšen koli veljaven razpored (tak, ki upošteva vse zahteve ljudi), v katerem so take vrzeli, ostane veljaven tudi, če ljudi za vrzeljo zamaknemo nazaj po času tako daleč, da vrzel izgine.

Razpored za posamezno točko zdaj ni nič drugega kot vrstni red, v katerem bomo na njej testirali ljudi; prvega od njih se bo testiralo od 7:00 do 7:00 + t sekund, drugega od 7:00 + t sekund do 7:00 + $2t$ sekund in tako naprej. Če imamo m točk, pa je razpored za vse skupaj spet le vrstni red, v katerem bomo prvih m ljudi testirali od 7:00 do 7:00 + t sekund, naslednjih m od 7:00 + t sekund do 7:00 + $2t$ sekund in tako naprej.

Poleg tega, ker vsako testiranje traja točno t sekund, če čas $h_i{:}m_i$ nekega človeka pade na sredo takega t-sekundnega območja, to pomeni, da ga v tem območju ne bomo smeli testirati (ker bi se to testiranje zanj končalo že prepozno), kar je torej za nas enako, kot če bi padel njegov čas $h_i{:}m_i$ na začetek tega območja. V nadaljevanju torej lahko v mislih vsak čas $h_i : m_i$ zamenjamo s celim številom, ki pove, koliko celih t-sekundnih intervalov mine od sedmih zjutraj do časa $h_i : m_i$; recimo temu $r_i = \lfloor (60(h_i - 7) + m_i) \cdot 60/t \rfloor$.

Opazimo lahko tudi, da če je neki razpored ljudi med točke za testiranje veljaven (torej če ustreza vsem zahtevam) in v njem nekoč testiramo človeka i in nekoč kasneje človeka j (ne nujno na isti točki) in je $r_i > r_j$, potem bi razpored ostal veljaven tudi, če človeka i in j v razporedu zamenjamo.[15] Omejimo se torej lahko na razporede, v katerih ljudi testiramo v naraščajočem vrstnem redu glede na r_i — najprej testiramo tistega z najmanjšim r_i in tako naprej. Recimo torej, da smo ljudi uredili in oštevilčili v tem vrstnem redu, tako da je $r_1 \leq r_2 \leq \ldots \leq r_n$.

Če imamo eno samo točko, bo človek k (z omejitvijo r_k) prišel na vrsto šele kot k-ti, torej bo končal ob času k (če se testiranje začne ob času 0), kar je v redu, če je $r_k \geq k$, sicer pa ne.

[15] Recimo namreč, da je bil človek i pred zamenjavo testiran do časa t_i, človek j pa do časa $t_j > t_i$; ker je bil razpored takrat veljaven, mora biti $t_i \leq r_i$ in $t_j \leq r_j$. Po zamenjavi je človek j testiran bolj zgodaj kot pred zamenjavo, torej je zanj razpored še vedno veljaven; človek i pa bo po zamenjavi testiran do časa t_j, kar je $\leq r_j$ in zato $< r_i$, saj je $r_i > r_j$; tako bo torej tudi človek i še vedno testiran dovolj zgoraj in razpored je res še vedno veljaven.

Podobno v splošnem, če imamo m točk, bo človek k tudi prišel na vrsto kot k-ti, vendar to pomeni, da bo končal ob času $\lceil k/m \rceil$, ker pač v vsaki časovni enoti obdelamo m ljudi. S tem m-jem torej lahko ustrežemo vsem zahtevam, če je $r_k \geq \lceil k/m \rceil$ (za vse k), sicer pa ne.

Tako dobimo naslednji postopek: začnimo z $m = 1$ in pregledujmo ljudi (načeloma je vseeno, v kakšnem vrstnem redu jih pregledujemo; glavno je, da so oštevilčeni naraščajoče po r_k); pri vsakem človeku k poglejmo, če je $r_k \geq \lceil k/m \rceil$, in če ni, povečujmo m za 1 tako dolgo, dokler ne bo ta pogoj izpolnjen. Ker bo na koncu gotovo $m \leq n$ (z n točkami lahko testiramo vse ljudi takoj ob sedmih, kar bo gotovo rešilo problem, razen če je kakšen od r_k enak 0, takrat pa je problem tako ali tako nerešljiv), je časovna zahtevnost $O(n)$ časa za povečevanje m-ja in pregled ljudi; pred tem pa načeloma $O(n \log n)$ za urejanje, razen če uporabimo urejanje s štetjem, za kar pa gre $O(n)$ časa.[16]

Še ena možnost: opazimo lahko, da če je naloga rešljiva (ob upoštevanju vseh omejitev) z m točkami za testiranje, je gotovo rešljiva tudi z več kot m točkami, saj bo tam prišel vsak človek na vrsto še prej (ali najkasneje ob istem času) kot pri natanko m točkah. Po eni strani vemo, da je naloga gotovo rešljiva z n točkami in da gotovo ni rešljiva z 0 točkami; najmanjši m, pri katerem je naloga še rešljiva, lahko poiščemo z bisekcijo med tema dvema skrajnostma:

$m_1 := 0$; $m_2 := n$;
while $m_2 - m_1 > 1$:
  (* *m_1 točk je gotovo premalo, m_2 točk je gotovo dovolj.* *)
  $m := \lfloor (m_1 + m_2)/2 \rfloor$;
  if je problem rešljiv z m točkami **then** $m_2 := m$ **else** $m_1 := m$;
(* *Ko se zanka konča, vemo, da je najmanjše primerno število točk m_2.* *)

Da preverimo, ali je problem rešljiv z m točkami, pa potrebujemo še vgnezdeno zanko, ki gre po vseh ljudeh (urejenih naraščajoče po r_k) in pri vsakem preveri, ali je $r_k \geq \lceil k/m \rceil$ (ali, z drugimi besedami: ljudi razporejamo med naših m točk, vsakega na tisto točko, ki ji je trenutno dodeljenih najmanj ljudi, potem pa preverimo, če je vsakdo testiran do zahtevanega časa). Tako imamo spet rešitev s časovno zahtevnostjo $O(n \log n)$.

Do minimalnega potrebnega m lahko pridemo tudi s požrešnim postopkom, pri katerem pregledujemo ljudi po naraščajočih k in vsakega dodelimo tisti testirni točki, ki ima zaenkrat najmanj ljudi; če pa bi bil zaradi tega ta človek testiran kasneje kot ob času r_k, odpremo novo testirno točko in ga dodamo tja. V spodnji psevdokodi nam t_i predstavlja število ljudi, ki smo jih že dodelili točki i.

$m := 0$;
za vsakega človeka k (po naraščajočih vrednostih r_k):
  $i :=$ tista točka (od 1 do m), ki ima najmanjšo vrednost t_i;
  if $m = 0$ **or** $t_i + 1 > r_k$:
    $m := m + 1$; $t_m := 1$; (* *Odprimo zanj novo točko.* *)
  else: $t_i := t_i + 1$; (* *Dodelimo tega človeka točki i.* *)

[16] Da bo šlo urejanje s štetjem v $O(n)$ časa, moramo prej še pogledati, če je kakšen od r_k večji od n, in takšne r_k postaviti na n. To smemo narediti, kajti pacient z $r_k > n$ nas tako ali tako nič ne omejuje: v največ n časovnih intervalih lahko testiramo vse ljudi celo na eni sami točki, tako da omejitvi $r_k = n$ ni nič težje ustreči kot omejitvi $r_k > n$.

Razpored, ki ga dobimo s tem postopkom, je gotovo veljaven: človeka k dodelimo obstoječi točki i le, če je pred dodajanjem tega človeka veljalo $t_i \le r_k - 1$, tako da bo on na tej točki testiran najkasneje ob času r_k; drugače pa odpremo zanj novo točko, kjer bo torej testiran že ob času 1, kar je gotovo $\le r_k$ (saj drugače problem sploh ne bi bil rešljiv).

Prepričajmo se zdaj, da nam ta postopek vrne razpored z najmanjšim možnim m (številom testirnih točk). Pa recimo, da kdaj ne bi bilo tako; vzemimo najmanjši tak testni primer (torej takega z najmanjšim številom ljudi n), pri katerem naš požrešni postopek porabi preveč točk; on jih torej porabi m, v resnici pa je mogoče te ljudi testirati že z $m-1$ ali manj točkami. Ali je mogoče, da je naš postopek odprl m-to točko že kaj prej kot šele pri zadnjem, n-tem človeku? To bi pomenilo, da je naš postopek že za prvih $n-1$ ljudi odprl m točk, toda ker je na vseh problemih z manj kot n ljudmi naš postopek optimalen, to pomeni, da se prvih $n-1$ ljudi ne da testirati z manj kot m točkami, zato pa se potem tudi vseh n ljudi ne da testirati z manj kot m točkami; toda to je v protislovju s predpostavko, da naš požrešni razpored teh n ljudi na m točk ni optimalen. — Zdaj torej vemo, da je naš požrešni algoritem odprl m-to točko šele za n-tega človeka. Takrat je moralo biti vsaki od dotedanjih $m-1$ točk že dodeljenih vsaj r_n ljudi, kajti drugače bi lahko človeka n dodelili eni od tistih točk in bi bil še vedno testiran najkasneje do časa r_n (in potem zanj ne bi bilo treba odpirati nove točke). Vsi ti ljudje imajo $r_k \le r_n$, saj smo rekli, da gleda naš postopek ljudi po naraščajočih časih r_k. Skupaj z n-tim človekom imamo torej vsaj $(m-1)\cdot r_n + 1$ ljudi, ki morajo vsi biti testirani najkasneje ob času r_n (nekateri morda celo še prej). Za te ljudi trdi naša predpostavka, da (ker naša požrešna rešitev ni optimalna) jih je mogoče testirati na manj kot m točkah. Toda na $m-1$ točkah je mogoče v r_n korakih testirati največ $(m-1)\cdot r_n$ ljudi, torej je nemogoče, da bi taka rešitev, boljša od naše, res obstajala. □

Pri opisanem požrešnem postopku je načeloma koristne še nekaj pazljivosti, če ga hočemo učinkovito implementirati. Vzdrževali bomo $\tau := \min_i t_i$, torej minimum t_i po vseh doslej odprtih točkah, poleg tega pa bomo za vsak t vzdrževali še množico $S[t]$ vseh točk, ki jim je bilo doslej dodeljenih natanko t ljudi (torej ki imajo trenutno $t_i = t$). To nam bo pri vsakem človeku omogočilo v $O(1)$ časa izbrati točko z najmanjšo t_i. Opišimo zdaj naš postopek podrobneje:

$m := 0$; $\tau := 0$; **for** $t := 1$ **to** n **to** $S[t] := \{\}$;
za vsakega človeka k (po naraščajočih vrednostih r_k):
  if $m = 0$ **or** $\tau \ge r_k$: (* *Treba bo odpreti novo točko.* *)
    $m := m + 1$; $\tau := 1$; dodaj m v $S[1]$;
  $i :=$ poljuben element množice $S[\tau]$;
  (* *Dodelimo tega človeka točki* i. *)
  pobriši i iz $S[\tau]$ in ga dodaj v $S[\tau + 1]$;
  if je $S[\tau]$ prazna **then** $\tau := \tau + 1$;

4. Palindromi

Preprosta, a neučinkovita rešitev je, da gremo z dvema gnezdenima zankama po vseh možnih podnizih vhodnega niza, za vsak podniz pa preverimo, ali je palindrom (in če je, povečamo števec palindromov, ki ga na koncu vrnemo kot rezultat). Da preverimo, ali je podniz palindrom, moramo preveriti, če sta njegov prvi in zadnji

znak enaka, če sta drugi in predzadnji znak enaka in tako naprej; premikajmo se torej po njem hkrati z dvema indeksoma, z enim od leve proti desni in z drugim od desne proti levi, ter primerjajmo znake na teh indeksih; če se indeksa srečata, ne da bi opazili kakšno neujemanje, lahko zaključimo, da je bil ta podniz palindrom. Ker moramo pregledati $O(n^2)$ podnizov in imamo pri vsakem $O(n)$ dela, ima ta rešitev časovno zahtevnost kar $O(n^3)$.

```
int Palindromi1(const string &s)
{
  int n = s.length(), stPalindromov = 0;
  for (int i = 0; i + 1 < n; i++) for (int j = i + 1; j < n; ++j)
  {
    // Preverimo, ali je s[i...j] palindrom.
    bool ok = true;
    for (int ii = i, jj = j; ii < jj; )
      if (s[ii++] != s[jj--]) { ok = false; break; }
    if (ok) ++stPalindromov;
  }
  return stPalindromov;
}
```

Ali v pythonu:

```
def Palindromi1(s):
  n = len(s); stPalindromov = 0
  if n <= 1: return 0
  for i in range(n - 1):
    for j in range(i + 1, n):
      # Preverimo, ali je s[i...j] palindrom.
      ii = i; jj = j
      while ii < jj and s[ii] == s[jj]: ii += 1; jj -= 1
      if ii >= jj: stPalindromov += 1
  return stPalindromov
```

Boljšo rešitev dobimo, če upoštevamo, da se v daljšem palindromu skrivajo krajši. Podniz $s[i]s[i+1]\ldots s[j-1]s[j]$ je lahko palindrom le, če je palindrom tudi malo krajši podniz $s[i+1]s[i+2]\ldots s[j-1]$ (in če sta poleg tega znaka $s[i]$ in $s[j]$ enaka). Takšne podnize je torej koristno pregledovati od krajših proti daljšim; čim opazimo, da nimamo več palindroma, nam daljših podnizov ni več treba gledati.

Postavimo se na primer v mislih v znak $s[i]$ in glejmo podnize s središčem v tem znaku: to so $s[i-d]\ldots s[i+d]$ za $d = 1, 2, \ldots$; če sta $s[i-1]$ in $s[i+1]$ enaka, imamo tu palindrom dolžine 3; če sta poleg tega tudi $s[i-2]$ in $s[i+2]$ enaka, imamo tu palindrom dolžine 5; in tako naprej. Čim opazimo pri kakšnem d neujemanje med $s[i-d]$ in $s[i+d]$, lahko s tem i končamo, saj potem naši podnizi tudi pri večjih d ne bodo več palindromi.

Podobno lahko obravnavamo tudi podnize sode dolžine. Če gledamo recimo podnize s središčem na meji med znakoma $s[i-1]$ in $s[i]$, so to podnizi oblike $s[i-d]\ldots s[i+d-1]$ za $d = 1, 2, \ldots$; če sta $s[i-1]$ in $s[i]$ enaka, imamo palindrom dolžine 2; če sta poleg tega tudi $s[i-2]$ in $s[i+1]$ enaka, imamo palindrom dolžine 4; in tako naprej.

Zdaj potrebujemo torej le dve gnezdeni zanki, eno po i (središče podniza) in eno po d (dolžina podniza). Pri vsakem povečanju d-ja imamo le $O(1)$ dela, da

preverimo, ali se znaka, ki smo ju na levem in desnem koncu dodali v podniz, ujemata; tako imamo rešitev s časovno zahtevnostjo $O(n^2)$.[17]

```
int Palindromi2(const string &s)
{
  int n = s.length(), stPalindromov = 0;
  for (int i = 1; i < n; ++i)
  {
    // Preštejmo palindrome lihe dolžine (vsaj 3) s središčem v s[i].
    L = i - 1; D = i + 1;
    while (L >= 0 && D < n && s[L] == s[D])
      ++stPalindromov, --L, ++D;
    // Preštejmo palindrome sode dolžine (vsaj 2) s središčem med s[i - 1] in s[i].
    int L = i - 1, D = i;
    while (L >= 0 && D < n && s[L] == s[D])
      ++stPalindromov, --L, ++D;
  }
  return stPalindromov;
}
```

Ali v pythonu:

```
def Palindromi2(s):
  n = len(s); stPalindromov = 0
  if n <= 1: return 0
  for i in range(1, n):
    # Preštejmo palindrome lihe dolžine (vsaj 3) s središčem v s[i].
    L = i - 1; D = i + 1
    while L >= 0 and D < n and s[L] == s[D]:
      stPalindromov += 1; L -= 1; D += 1
    # Preštejmo palindrome sode dolžine (vsaj 2) s središčem med s[i - 1] in s[i].
    L = i - 1; D = i;
    while L >= 0 and D < n and s[L] == s[D]:
      stPalindromov += 1; L -= 1; D += 1
  return stPalindromov
```

5. Čarobne jame

Za začetek je koristno predelati naše vhodne podatke tako, da bomo imeli za vsako jamo seznam njenih *sosed* (torej tistih jam, ki so neposredno povezane z njo s prehodi); za jamo u recimo temu seznamu $S[u]$.

Jame lahko pregledujemo sistematično, da ugotovimo, kam vse je mogoče priti iz s: iz s je mogoče priti v njene sosede, iz teh v njihove sosede in tako naprej. Koristno je torej vzdrževati tabelo, v kateri bomo označevali, za katere jame že vemo, da so dosegljive iz s; poleg tega bomo hranili tudi seznam Q z jamami, za katere smo že ugotovili, da so dosegljive iz s, nismo pa še pogledali, kam je mogoče potem priti naprej iz njih — to so torej jame, ki jih bomo morali še pregledati. Tako dobimo približno takšen postopek:

[17] Za namene naše naloge je ta rešitev dovolj učinkovita, kot zanimivost pa omenimo, da obstaja tudi rešitev z linearno časovno zahtevnostjo, $O(n)$; gl. *Bilten* 2013, str. 128–130.

označi jamo s za dosegljivo in jo dodaj v seznam Q;
dokler Q ni prazen:
naj bo u poljubna jama iz Q; pobriši jo iz Q;
za vsako u-jevo sosedo v:
če v še nimamo označene kot dosegljive,
jo označi zdaj in jo dodaj v seznam Q;

Ta postopek bo sčasoma dosegel vse jame, ki jih je sploh mogoče doseči iz s; na koncu moramo torej le še preveriti, ali je med njimi tudi jama t. (Lahko ga tudi prekinemo predčasno, čim se izkaže jama t za dosegljivo.) To, v kakšnem vrstnem redu jemljemo jame iz Q, za naš namen načeloma ni pomembno; če vzamemo vedno tisto jamo, ki je že najdlje v Q, dobimo znani postopek iskanja v širino (*breadth-first search*, BFS).

Doslej še nismo upoštevali čarobnih zvitkov, ki jih omenja besedilo naloge. Recimo, da se v jami u nahaja zvitek, ki ustvari nov prehod med jamama $c[u]$ in $d[u]$. Če jama u ni dosegljiva iz s, Henrik takega zvitka tako ali tako ne bi mogel uporabiti in se zaradi njega pri dosegljivosti nič ne spremeni. Če pa u je dosegljiva, bomo sčasoma prišli do nje v glavni zanki našega postopka in lahko takrat razmislimo o tem, kaj za nas pomeni novi prehod od $c[u]$ do $d[u]$. Tu ločimo dve možnosti: (1) Če niti za $c[u]$ niti za $d[u]$ takrat še ne vemo, ali sta dosegljivi ali ne, je dovolj, če dodamo $c[u]$ na seznam sosedov jame $d[u]$ in obratno; tako bomo novi prehod primerno upoštevali v bodoče, če bomo kdaj prišli do jame $c[u]$ ali $d[u]$. (2) Če pa za vsaj eno od $c[u]$ in $d[u]$ že vemo, da je dosegljiva, potem zdaj zaradi novega prehoda vemo, da sta dosegljivi obe in ju lahko kot taki tudi označimo (če še nista) ter ju dodamo v seznam Q.

Časovna zahtevnost naše rešitve je $O(n+m)$, saj vsako jamo — teh pa je $O(n)$ — le enkrat dodamo v Q (ko jo označimo za dosegljivo), zato jo tudi le enkrat vzamemo iz Q in le enkrat pregledamo njene sosede; s pregledovanjem sosed pa imamo le toliko dela, kolikor je prehodov, torej $O(m)$.

Oglejmo si še primer implementacije tega postopka v C++, pri katerem pa bomo predpostavili, da gredo številke jam (in prehodov) od 0 naprej namesto od 1 naprej:

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

bool JeDosegljiva(int n, int m, int s, int t,
                  const vector<int> &a, const vector<int> &b,
                  const vector<int> &c, const vector<int> &d)
{
  // Pripravimo sezname sosed.
  vector<vector<int>> S(n);
  for (int i = 0; i < m; ++i) S[a[i]].emplace_back(b[i]), S[b[i]].emplace_back(a[i]);
  // Pomožni podprogram, ki označi jamo kot dosegljivo in jo doda v Q.
  vector<bool> dosegljiva(n, false); vector<int> Q;
  auto Oznaci = [&] (int u) { if (! dosegljiva[u]) dosegljiva[u] = true, Q.emplace_back(u); };
  // Na začetku vemo, da je dosegljiva jama s.
  Oznaci(s);
  // Preglejmo preostanek sistema jam.
  while (! Q.empty())
  {
```

```
    int u = Q.back(); Q.pop_back();
    // Ker je u dosegljiva, so dosegljive tudi njene sosede.
    for (int v : S[u]) Oznaci(v);
    // Upoštevajmo čarobni zvitek v jami u, ki odpre prehod med jamama cu in du.
    if (int cu = c[u], du = d[u]; cu >= 0 && du >= 0)
      // Če cu in du še nista dosegljivi, si novi prehod le zapomnimo.
      if (! dosegljiva[cu] && ! dosegljiva[du])
        S[cu].emplace_back(du), S[du].emplace_back(cu);
      // Sicer vemo, da sta dosegljivi obe in ne le ena od njiju.
      else Oznaci(cu), Oznaci(du);
  }
  return dosegljiva[t];
}
```

In še v pythonu:

```
def JeDosegljiva(n: int, m: int, s: int, t: int,
                 a: list[int], b: list[int], c: list[int], d: list[int]) -> bool:
  # Pripravimo sezname sosed.
  S = [[]] * n
  for ai, bi in zip(a, b): S[ai].append(bi); S[bi].append(ai)
  # Pomožni podprogram, ki označi jamo kot dosegljivo in jo doda v Q.
  dosegljiva = [False] * n; Q = []
  def Oznaci(u):
    if not dosegljiva[u]: dosegljiva[u] = True; Q.append(u)
  # Na začetku vemo, da je dosegljiva jama s.
  Oznaci(s)
  # Preglejmo preostanek sistema jam.
  while Q:
    u = Q.pop()
    # Ker je u dosegljiva, so dosegljive tudi njene sosede.
    for v in S[u]: Oznaci(v)
    # Upoštevajmo čarobni zvitek v jami u, ki odpre prehod med jamama cu in du.
    cu = c[u]; du = d[u]
    if cu >= 0 and du >= 0:
      # Če cu in du še nista dosegljivi, si novi prehod le zapomnimo.
      if not (dosegljiva[cu] or dosegljiva[du]):
        S[cu].append(du); S[du].append(cu)
      # Sicer vemo, da sta dosegljivi obe in ne le ena od njiju.
      else: Oznaci(cu); Oznaci(du)
  return dosegljiva[t]
```

Naloge so sestavili: ulične luči — Benjamin Bajd; naredimo hitro urejanje zares hitro! — Urban Duh; stoli — Bor Grošelj Simić; oviratlon, prisotnost — Tomaž Hočevar; lučka — Gregor Kikelj; padalski izlet, valj — Vid Kocijan; kibi, mebi, konkordanca, nedeljiva hramba — Mark Martinec; tehtnica — Polona Novak in Mark Martinec; palindromi — Jakob Schrader; čarobne jame — Jure Slak; neurejene besede — Jasna Urbančič; videostena — Borut Žnidar; špijonaža, urejanje z medianami, cikcakasti nizi, histogram — Janez Brank.

REŠITVE NALOG ZA PRVO SKUPINO OŠ

1. Trikotniki

Označimo stranice našega domnevnega trikotnika z a, b in c, pri čemer naj bo c najdaljša. Spomnimo se, da v trikotniku velja trikotniška neenakost: posamezna stranica je krajša kot ostali dve skupaj. To pomeni $a + b > c$, pa tudi $a + c > b$ in $b + c > a$. Zadnji dve neenakosti za naše a, b in c gotovo veljata že zaradi tega, ker je $c \geq a$ in $c \geq b$ (in ker so števila a, b in c pozitivna); prva neenakost, $a + b > c$, pa ne velja nujno in jo moramo preveriti. Če ne velja, lahko zaključimo, da trikotnik s stranicami a, b in c ne obstaja.

Če pa neenakost $a + b > c$ velja, trikotnik s stranicami a, b in c gotovo obstaja. O tem se lahko prepričamo na primer takole: vpeljimo v ravnini koordinatni sistem in to tako, da bo eno oglišče najdaljše stranice trikotnika v točki $(0, 0)$, drugo pa v $(c, 0)$. Tretje oglišče, recimo mu (x, y), mora potem ležati na oddaljenosti a od $(0, 0)$ in oddaljenosti b od $(c, 0)$. Tako imamo pogoja $x^2 + y^2 = a^2$ in $(x-c)^2 + y^2 = b^2$. Iz prve enačbe izrazimo $y^2 = a^2 - x^2$, kar lahko vstavimo v drugo in dobimo $(x-c)^2 + a^2 - x^2 = b^2$, iz tega pa sčasoma $x = (c^2 - b^2 + a^2)/(2c)$. Ker je $a+b > c$, je $a^2 + b^2 + 2ab > c^2$; ker sta a in b pozitivna, je $2ab > 0$, tako da dobimo $a^2 > c^2 - b^2$; zato pa je $x < (2a^2)/(2c) = a \cdot (a/c) \leq a$. In ker je $b \leq c$, je $c^2 - b^2 \geq 0$; in ker sta a in c pozitivna, je potem $x = (c^2 - b^2 + a^2)/(2c) \geq a^2/2c > 0$. Tako torej vidimo, da je $x \in (0, a)$; zato je v prej omenjeni enačbi $y^2 = a^2 - x^2$ desna stran večja od 0 in lahko y izračunamo kot $y = \sqrt{a^2 - x^2}$. Tako smo torej našli primerno točko (x, y), ki jo lahko vzamemo za tretje oglišče trikotnika in zanj potem vemo, da bo imel stranice dolžine a, b in c.

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

int main()
{
  // Preberimo stranice trikotnika.
  int a, b, c; cin >> a >> b >> c;
  // Izpišimo rezultat.
  cout << (a + b > c ? "trikotnik" : "ponaredek") << endl; return 0;
}
```

Naloga ne pove natančno, kaj naj naredimo v primerih, ko je $a + b = c$. Takrat je načeloma mogoč „trikotnik“ s temi stranicami, ki pa je v resnici izrojen v daljico. Zgornja rešitev take primere razglasi za ponaredke.

2. Daljnovod čez podeželje

Niz z opisom daljnovoda bomo pregledovali znak za znakom od leve proti desni. Pri tem bomo v spremenljivkah vzdrževali dva podatka: ali ima trenutna razpetina kakšno previsoko krošnjo (trenutnaPrevisoka) in koliko razpetin s previsoko krošnjo smo doslej že videli (stPrevisokih).

Spomnimo se, da se prva razpetina ne začne na začetku niza, pač pa šele pri prvem znaku T; zato ima spremenljivka trenutnaPrevisoka v spodnji rešitvi lahko tri

možne vrednosti: -1 pomeni, da sploh še nismo dosegli prve razpetine; 0 pomeni, da v trenutni razpetini še nismo videli nobene previsoke krošnje; 1 pa, da smo jo že.

Ko vidimo znak `T`, lahko pogledamo vrednost te spremenljivke in če je bila 1, to pomeni, da se tu končuje razpetina s previsoko krošnjo in moramo povečati števec takšnih razpetin. V vsakem primeru nato postavimo to spremenljivko na 0, saj se zdaj začenja nova razpetina (ali pa del niza za zadnjo razpetino, če je trenutni `T` zadnji v nizu).

Ko pa vidimo znak `o`, smo pri drevesu s previsoko krošnjo; če ima `trenutnaPrevisoka` takrat vrednost -1, je ne smemo spreminjati, saj še nismo pri prvi razpetini in nima trenutno drevo nobenega učinka. Če pa smo že pri neki razpetini, zanjo zdaj vemo, da vsebuje neko previsoko krošnjo, torej moramo postaviti `trenutnaPrevisoka` na 1.

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

int main()
{
  // Preberimo vhodne podatke.
  int n; string s; cin >> n >> s;

  // Preglejmo vhodni niz.
  int stPrevisokih = 0, trenutnaPrevisoka = -1;
  for (char c : s)
    if (c == 'T') {
      // Trenutna razpetina se končuje; poglejmo, če je imela kakšno previsoko krošnjo.
      if (trenutnaPrevisoka == 1) ++stPrevisokih;

      // Pripravimo se na naslednjo razpetino.
      trenutnaPrevisoka = 0; }

    else if (c == 'o' && trenutnaPrevisoka == 0)
      // Pravkar smo ugotovili, da je v trenutni razpetini previsoka krošnja.
      trenutnaPrevisoka = 1, ++stPrevisokih;

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

Pomembna podrobnost pri tej nalogi je, da moramo števec razpetin s previsoko krošnjo povečati šele, ko pridemo do `T`-ja na koncu razpetine, ne pa že takrat, ko prvič zagledamo kakšen `o` na njej, saj takrat še ne moremo vedeti, ali smo sploh še na razpetini ali pa morda že na območju za zadnjim `T`-jem v nizu (ki ne šteje za razpetino).

3. Predor

Podatke o vozilih berimo v zanki; pri vsakem sproti izračunajmo čas vožnje skozi predor, torej $k - z$, in preverimo, če je morda manjši od najmanjšega dovoljenega časa, torej ali je $k - z < t$. Če to drži, izpišimo registrsko število trenutnega vozila.

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

int main()
{
```

```
  // Preberimo število vozil in minimalni čas vožnje.
  int n, t; cin >> n >> t;

  // Obdelajmo vsa vozila.
  while (n-- > 0)
  {
    // Preberimo podatke o naslednjem vozilu.
    string regSt; int z, k; cin >> regSt >> z >> k;

    // Če je čas vožnje prekratek, izpišimo registrsko številko.
    if (k - z < t) cout << regSt << endl;
  }
  return 0;
}
```

4. Besede

Besede vhodnega zaporedja berimo v zanki; pri vsaki besedi moramo preveriti, če je njen prvi znak enak zadnjemu znaku prejšnje besede, zato je koristno, če si slednjega nekje zapomnimo — v spodnji rešitvi je to spremenljivka prejsnji. Če sta prvi znak trenutne besede in zadnji znak prejšnje različna, lahko takoj zaključimo, da vhodno zaporedje ne ustreza pogojem naloge, ne glede na to, kaj se bo v nadaljevanju zaporedja še dogajalo; za hranjenje tega podatka bomo imeli zato še eno spremenljivko (ustreza), ki nam bo na koncu tudi povedala, kaj moramo izpisati.

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

int main()
{
  bool ustreza = true; // Ali dosedanje zaporedje ustreza pogojem?
  char prejsnji = ' '; // Zadnji znak prejšnje besede.

  int n; cin >> n;      // Preberimo število vhodnih besed.

  // Preglejmo vhodne besede.
  while (n-- > 0)
  {
    // Preberimo naslednjo besedo.
    string beseda; cin >> beseda;

    // Preverimo, ali se njen prvi znak ujema z zadnjim zankom prejšnje.
    if (prejsnji != ' ' && beseda[0] != prejsnji) ustreza = false;

    // Zapomnimo si njen zadnji znak.
    prejsnji = beseda.back();
  }

  // Izpišimo rezultat.
  cout << (ustreza ? "" : "ne ") << "ustreza" << endl; return 0;
}
```

Mimogrede, ko opazimo neujemanje in postavimo ustreza na **false**, bi lahko glavno zanko tudi takoj prekinili, saj takrat že vemo, da je vhodno zaporedje neustrezno in bo takšno tudi ostalo.

Naloga sprašuje še, kako ugotoviti, ali lahko vhodno zaporedje postane ustrezno, če spremenimo vrstni red besed v njem. Za potrebe tega razmišljanja je pri posamezni besedi pomembno le, s katero črko se začne in s katero konča. Predstavimo

zato naš seznam besed z grafom, ki ima po eno točko za vsako črko abecede (v abecedo bomo šteli le tiste črke, na katere se začenja ali končuje vsaj ena beseda našega seznama) in po eno povezavo za vsako besedo z našega seznama; če se beseda začne na črko u in konča na črko v, jo v grafu predstavlja usmerjena povezava $u \to v$. (Natančneje rečeno je to multigraf in ne navaden graf, ker je lahko v njem več povezav z enakim začetnim in enakim končnim krajiščem.)

Vsak sprehod v tem grafu zdaj ustreza nekemu zaporedju besed, v katerem se vsaka naslednja začne na tisto črko, na katero se prejšnja konča. Ker bi radi sestavili tako zaporedje iz vseh n besed, nas torej pravzaprav zanima, ali v grafu obstaja sprehod, ki uporabi vseh n povezav (vsako natanko enkrat) — to pa je ravno Eulerjev sprehod.

Za vsako črko u označimo z d_u razliko med vhodno in izhodno stopnjo točke u (ali, z drugimi besedami, med številom besed, ki se končajo na u, in besed, ki se začnejo na u). Prepričali se bomo, da obstaja Eulerjev sprehod natanko tedaj, ko je naš graf šibko povezan in ko bodisi (1) za vse črke u velja $d_u = 0$ bodisi (2) za eno črko velja $d_u = 1$, za eno $d_u = -1$ in za vse ostale $d_u = 0$.

($\Rightarrow$) Recimo, da obstaja Eulerjev sprehod in da se začne v točki v in konča v točki w. Na začetku torej točko v enkrat zapusti; nato v vsakem koraku vstopi v neko točko in jo v naslednjem koraku spet zapusti; in na koncu še enkrat vstopi v točko w. Pri tem tudi porabi vse povezave, vsako natanko enkrat, torej je razlika med številom vstopov v točko u in izstopov iz nje ravno enaka d_u. Tako torej vidimo, da je $d_v = -1$, $d_w = 1$, za ostale točke pa je $d_u = 0$ — to je lastnost (2). Poseben primer nastopi, če je $v = w$, torej če se sprehod začne in konča v isti točki; tedaj je število vstopov in izstopov tudi pri njej enako in imamo lastnost (1). — Ker Eulerjev sprehod uporabi vse povezave, obišče s tem tudi vse točke, saj smo v naš graf vzeli le take točke, ki imajo tudi kakšno povezavo (= le take črke, na katere se začne ali konča kakšna beseda), torej je graf res šibko povezan.

($\Leftarrow$) Ogledali si bomo Hierholzerjev algoritem za iskanje Eulerjevega sprehoda. Če ima graf lastnost (2), naj bo v tista črka, ki ima $d_v = -1$, in naj bo w tista črka, ki ima $d_w = 1$; če ima graf lastnost (1), vzemimo za v poljubno črko in naj bo $w = v$. Začnimo sprehod v točki v in pojdimo na vsakem koraku naprej po poljubni taki povezavi, ki je doslej še nismo uporabili. Ustavimo se, ko iz trenutne točke ne kaže nobena še neuporabljena povezava. Hitro se vidi, da se to zagotovo zgodi v točki w in ne kje drugje.[18] Če smo s tem porabili že vse povezave, imamo Eulerjev obhod in lahko končamo.

Sicer pa zdaj za vsako točko grafa velja, da je število še neuporabljenih vhodnih povezav vanjo enako številu še neuporabljenih izhodnih povezav. Začnimo v poljubni točki y, ki leži na dosedanjem sprehodu in ima kakšno neuporabljeno izhodno povezavo. Iz nje spet nadaljujmo po neuporabljenih povezavah, dokler je to

[18] Recimo, da se ustavimo v neki točki u. (a) Če je $u \neq v$, je število naših vstopov vanjo za 1 večje od števila izstopov; ker poti iz u ne moremo nadaljevati, to pomeni, da so vse izhodne povezave iz u že uporabljene; zato mora biti vhodnih povezav vsaj za 1 več kot izhodnih, torej $d_u \geq 1$, kar pa je mogoče le tako, da je $d_u = 1$ in $u = w$. (b) Če pa je $u = v$, je število naših vstopov vanjo enako številu izstopov in ker so vse izhodne povezave iz u že uporabljene, mora biti vhodnih vsaj toliko kot izhodnih, torej $d_v \geq 0$, kar pa je mogoče le, če ima graf lastnost (1) (in je $d_v = 0$), tedaj pa je $v = w$, torej spet vidimo, da smo se ustavili ravno v točki w.

[18] Prepričajmo se, da taka točka gotovo obstaja. Vemo, da obstajajo v grafu še neuporabljene povezave; naj bo x neko krajišče ene od njih. Ker je naš graf šibko povezan, obstaja gotovo pot

še mogoče. Podoben razmislek kot v prejšnjem odstavku nam pokaže, da se bomo ustavili spet v točki y; tako bomo dobili neki obhod (z začetkom in koncem v y), ki ga lahko vrinemo v naš dosedanji nastajajoči sprehod in slednji tako pokrije nekaj več povezav kot prej.

Postopek iz prejšnjega odstavka lahko zdaj v zanki ponavljamo, dokler ne uporabimo vseh povezav. □

Na vprašanje, ali je dane nize mogoče prerazporediti v ustrezen vrstni red, lahko torej odgovorimo tako, da pripravimo zgoraj opisani graf; nato izračunamo stopnje točk in preverimo, če ima eno od lastnosti (1) in (2); in končno z iskanjem v širino preverimo še, ali je šibko povezan.

(če zanemarimo smeri povezav) od v (začetne točke našega sprehoda) do x. Naj bo y zadnja točka na tej poti, ki leži na našem sprehodu, in naj bo y' njena naslednica na tej poti. Povezava med y in y' torej še ni bila uporabljena, saj bi sicer tudi y' ležala na našem sprehodu. Če kaže ta povezava iz y v y', ima torej y (ki leži na sprehodu) neuporabljeno izhodno povezavo, kar smo tudi iskali; če pa kaže povezava iz y' v y, ima y neuporabljeno vhodno povezavo, zato pa mora imeti tudi neko neuporabljeno izhodno povezavo, saj že vemo, da ima zdaj vsaka točka enako število obojih.

REŠITVE NALOG ZA DRUGO SKUPINO OŠ

1. Napačna imena

V zanki primerjajmo istoležne znake obeh vhodnih nizov, dokler bodisi ne opazimo neujemanja bodisi ne pridemo do konca nizov. Če smo opazili neujemanje, izpišimo indeks, kjer je do njega prišlo (pri tem pazimo, da se v C/C++ indeksi znakov štejejo od 0 naprej, mi pa ga moramo izpisati od 1 naprej); če pa smo prišli do konca nizov, izpišimo „`pravilno ime`“.

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

int main()
{
  // Preberimo vhodna niza.
  string s, t; cin >> s >> t;

  // Poiščimo prvo neujemanje.
  int i = 0; while (i < s.length() && s[i] == t[i]) ++i;

  // Izpišimo rezultat.
  if (i < s.length()) cout << (i + 1) << endl;
  else cout << "pravilno ime" << endl;
  return 0;
}
```

2. Ceneno potovanje

Ker z vsakega letališča vedno letimo z najcenejšim letom, je dovolj, če si od vseh letov s tistega letališča zapomnimo le najcenejšega. Ob branju vhodnih podatkov si torej pripravimo dve tabeli oz. vektorja: `kam[a]` pove, kam leti najcenejši let iz mesta a, `cena[a]` pa je cena tega leta. Nato lahko s pomočjo tabele `kam` enostavno simuliramo potek našega potovanja: na vsakem koraku gremo s trenutnega letališča v tisto, ki ga zanj določa tabela `kam`. Ustavimo se, če s trenutnega letališča sploh ni nobenega leta (kar je v spodnji rešitvi predstavljeno tako, da je v tabeli `kam` tam vrednost -1), če pridemo spet nazaj v začetno letališče 1 ali pa če smo že naredili k korakov.

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

int main()
{
  // Preberimo vhodne podatke. Za vsako letališče si zapomnimo le
  // ceno najcenejšega leta z njega ter to, kam ta let vodi.
  int n, k; cin >> n >> k;
  const int m = 1000; // število mest oz. letališč
  vector<int> kam(m + 1, -1), cena(m + 1, -1);
  for (int i = 0; i < n; ++i) {
    // Preberimo naslednji let.
    int ai, bi, ci; cin >> ai >> bi >> ci;

    // Če je to najcenejši let z „ai“ doslej, si ga zapomnimo.
```

```
    if (kam[ai] < 0 || ci < cena[ai]) kam[ai] = bi, cena[ai] = ci; }
  // Odsimulirajmo potovanje.
  int kje = 1, stKorakov = 0;
  do {
    // Morda s trenutnega letališča sploh ni nobenega leta.
    if (kam[kje] < 0) break;
    // Naredimo naslednji korak.
    kje = kam[kje]; ++stKorakov;
  // Ustavimo se, ko pridemo nazaj v 1 ali naredimo k korakov.
  } while (kje != 1 && stKorakov < k);
  // Izpišimo rezultat.
  cout << kje << endl << stKorakov << endl; return 0;
}
```

3. Barvne packe

Stanje platna lahko predstavimo z dvodimenzionalno tabelo oz. (kot v spodnji rešitvi) z vektorjem v nizov, ki so dolgi po d znakov in predstavljajo vsak po eno vrstico platna. Na začetku naj bodo vsi znaki pike, nato pa v zanki beremo podatke o packah s standardnega vhoda in jih rišemo na platno. Naloga sicer šteje koordinate od 1 naprej, mi pa jih bomo šteli od 0 naprej, da jih bomo lahko uporabljali kot indekse v vektor in nize; pri branju vhodnih podatkov zato pazimo, da koordinatam x_i in y_i odštejemo 1.

Packa s središčem (x_i, y_i) in velikostjo s_i je načeloma prisotna pri x-koordinatah od $x_i - s_i$ do $x_i + s_i$, razen če ležijo zunaj platna; v resnici moramo torej iti po x od $\max\{x_i - s_i, 0\}$ do $\min\{x_i + s_i, d - 1\}$. Podoben razmislek velja tudi za y-koordinate. Ko tako določimo koordinate pravokotnika, ki ga ta packa na platnu res pokrije, se lahko z dvema gnezdenima zankama sprehodimo po vseh celicah tega pravokotnika in vpisujemo barvo trenutne packe na ustrezna mesta v tabeli oz. vektorju, ki predstavlja platno.

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

int main()
{
  // Preberimo velikost in pripravimo prazno platno.
  int d, v, p; cin >> d >> v >> p;
  vector platno(v, string(d, '.'));
  while (p-- > 0) //Obdelajmo vse packe.
  {
    // Preberimo podatke o naslednji packi.
    int xi, yi, si; char bi; cin >> xi >> yi >> si >> bi; --xi; --yi;
    // Izračunajmo koordinate, ki jih ta packa pokriva.
    int xOd = max(xi - si, 0), xDo = min(xi + si, d - 1);
    int yOd = max(yi - si, 0), yDo = min(yi + si, v - 1);
    // Narišimo packo.
    for (int y = yOd; y <= yDo; ++y) {
```

```
        auto &vrstica = platno[y];
        for (int x = xOd; x <= xDo; ++x) vrstica[x] = bi; }
    }
    // Izpišimo končno stanje platna.
    for (const auto &vrstica : platno) cout << vrstica << endl;
    return 0;
}
```

4. Dolge skladbe

Naj bo $L_i := \ell_1 + \ell_2 + \ldots + \ell_i$ čas, ko se konča i-ta skladba. Ko nas zanima, katera skladba se vrti ob času t_j, torej pravzaprav iščemo tak i, za katerega velja $L_{i-1} < t_j \leq L_i$. Ali še drugače: iščemo najmanjši i, za katerega je $t_j \leq L_i$. (Pri kasnejših skladbah, recimo $k > i$, je pogoj $t_j \leq L_k$ tudi izpolnjen, vendar pa ni več izpolnjen pogoj $L_{k-1} < t_j$.) Ker je zaporedje $L_1, L_2, \ldots, L_n$ naraščajoče, lahko v njem najmanjši element, ki je večji ali enak t_j, poiščemo z bisekcijo; v C++ lahko uporabimo kar funkcijo lower_bound iz standardne knjižnice.

Ker imamo lahko do 10^5 skladb dolžine do 10^9, gredo lahko vrednosti L_i do 10^{14}, zato moramo zanje uporabiti kak 64-bitni podatkovni tip (v spodnji rešitvi je to **long long**). Pri izpisu rezultatov pazimo še na to, da moramo izpisovati številke skladb od 1 do n in ne od 0 do $n-1$.

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

int main()
{
    int n, q; cin >> n >> q;
    // Preberimo dolžine skladb in izračunajmo njihove kumulativne vsote.
    // Tako bo vsote[i] čas, ko se konča skladba i.
    vector<long long> vsote(n);
    for (int i = 0; i < n; ++i) {
        cin >> vsote[i]; if (i > 0) vsote[i] += vsote[i - 1]; }
    while (q-- > 0) // Obdelajmo poizvedbe.
    {
        long long tj; cin >> tj;
        // Z bisekcijo poiščimo prvo skladbo, ki se konča ob času t_j ali kasneje.
        int skladba = lower_bound(vsote.begin(), vsote.end(), tj) - vsote.begin();
        cout << (skladba + 1) << endl; // Izpišimo rezultat.
    }
    return 0;
}
```

Časovna zahtevnost te rešitve je $O(n + q \log n)$, namreč $O(n)$ za izračun vsot L_i in nato pri vsaki od q poizvedb po $O(\log n)$ časa za bisekcijo.

Nalogo lahko rešimo tudi tako, da poizvedbe uredimo naraščajoče po t_j in jih obravnavamo v tem vrstnem redu. Ker zdaj časi t_j ves čas le naraščajo, bodo tudi odgovori na poizvedbe ves čas le naraščali, zato lahko pregledujemo vrednosti L_i po vrsti in pri vsaki poizvedbi nadaljujemo pri tistem i, pri katerem smo se

pri prejšnji poizvedbi ustavili. (To si lahko predstavljamo tudi kot zlivanje dveh naraščajočih zaporedij, namreč $L_1, \ldots, L_n$ in $t_1, \ldots, t_q$.) Vendar pa moramo na koncu izpisati odgovore na poizvedbe v takem vrstnem redu, v kakršnem so bile v vhodnih podatkih, zato moramo pri urejanju poizvedb ob vsakem t_j hraniti še njegov prvotni indeks j.

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

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

  // Preberimo dolžine skladb in izračunajmo njihove kumulativne vsote.
  // Tako bo vsote[i] čas, ko se konča skladba i.
  vector<long long> vsote(n);
  for (int i = 0; i < n; ++i) {
    cin >> vsote[i]; if (i > 0) vsote[i] += vsote[i - 1]; }

  // Preberimo poizvedbe in jih uredimo naraščajoče po t_j.
  vector<pair<long long, int>> poizvedbe(q);
  for (int j = 0; j < q; ++j) {
    auto &P = poizvedbe[j]; cin >> P.first; P.second = j; }
  sort(poizvedbe.begin(), poizvedbe.end());

  // Izračunajmo odgovore na vse poizvedbe.
  vector<int> odgovori(q);
  int i = 0; for (auto [tj, j] : poizvedbe) {
    // Premaknimo se z i do prve skladbe, ki se konča ob času tj ali kasneje.
    while (tj > vsote[i]) ++i;
    odgovori[j] = i; }

  // Izpišimo rezultate.
  for (int odgovor : odgovori) cout << (odgovor + 1) << endl;
  return 0;
}
```

Časovna zahtevnost te rešitve je $O(n + q \log q)$, namreč $O(n)$ za računanje vsot L_i, nato $O(q \log q)$ za urejanje poizvedb po t_j in potem $O(n+q)$ za zlivanje. To, katera od obeh tu opisanih rešitev je boljša, je torej odvisno od tega, ali je število poizvedb veliko v primerjavi s številom skladb ali obratno.

Naloge so sestavili: besede — Bor Grošelj Simić; trikotniki — Tomaž Hočevar; predor — Matija Likar; barvne packe — Jure Slak; daljnovod, napačna imena — Jaka Velkaverh; ceneno potovanje, dolge skladbe — Patrik Žnidaršič.

A. Prisotnost

Če se urnik predavanj ne bi spreminjal, bi lahko najmanjše potrebno število obiskov izračunali s preprostim požrešnim algoritmom. Recimo, da smo z dosedanjimi obiski že pokrili vsa predavanja z začetkom $a_i \le z$. Nepokrita torej ostanejo predavanja z $a_i > z$; med njimi naj bo p tisto z najzgodnejšim koncem, torej najmanjšim b_p. Naslednji obisk ne sme nastopiti kasneje kot ob b_p, saj bi sicer zgrešili predavanje p; nobene koristi pa ni od tega, da bi naslednji obisk nastopil prej kot ob b_p, kajti tak zgodnejši obisk — recimo ob času $t < b_p$ — ne pokrije nobenega (še nepokritega) predavanja, ki ga ne pokrije tudi obisk ob času b_p, kajti táko predavanje bi se moralo končati nekje na intervalu $[t, b_p)$, torej prej kot ob b_p, takšnih predavanj pa ni, saj smo p izbrali tako, da ima najzgodnejši čas konca med vsemi še nepokritimi.

Za implementacijo takega požrešnega postopka je koristno urediti predavanja po času konca, tako da je $b_1 \le b_2 \le \ldots \le b_m$ (recimo, da imamo m predavanj). Potem lahko zapišemo postopek s psevdokodo takole:

algoritem Požrešni:
  $s := 0$; $z := -\infty$; (* *s = število obiskov, z = čas zadnjega obiska.* *)
  for $p := 1$ **to** m:
    (* *Dosedanji obiski pokrijejo vsa predavanja i, ki imajo $a_i \le z$.* *)
    if $a_p > z$ **then** $s := s + 1$, $z := b_p$;

Pri naši nalogi, kjer se predavanja dodajajo in brišejo, lahko pripravimo seznam (urejen po času konca) vseh predavanj, ki se pojavijo v vhodnih podatkih; recimo spet, da jih je m; dodajanje in brisanje si lahko predstavljamo tako, da niso vedno vsa ta predavanja res prisotna. Recimo zdaj, da se spremeni stanje i-tega predavanja v urejenem seznamu (ker je bilo dodano ali pobrisano); pri gornjem požrešnem postopku se za $p < i$ nič ne spremeni, pri $p = i$ pa lahko nastopi sprememba in ta morda vpliva tudi na kasnejše iteracije. Če bomo ta del zanke izvedli ponovno, bo naša rešitev prepočasna.

Vzemimo $B \approx \sqrt{m}$ in razdelimo naš seznam na B blokov s po (približno) B predavanji. Gornji postopek lahko zdaj predelamo v:

algoritem PožrešniPoBlokih:
1 $s := 0$; $z := -\infty$; (* *s = število obiskov, z = čas zadnjega obiska.* *)
2 **for** $\beta := 1$ **to** B:
3 naj bo s' najmanjše potrebno število obiskov, s katerim lahko pokrijemo vsa predavanja iz bloka β ob predpostavki, da so predavanja z $a_p \le z$ že pokrita, in naj bo z' zadnji od teh obiskov;
4 **if** $s' > 0$ **then** $s := s + s'$, $z := z'$;

Vrstico 3 si lahko predstavljamo kot funkcijo $f_\beta(z)$, ki za dani blok β in čas zadnjega obiska z vrne par (s', z') s številom potrebnih dodatnih obiskov, da se pokrije vsa predavanja iz bloka β, in časom zadnjega od teh obiskov (če pa je $s' = 0$, je vrednost z' nedefinirana). Če imamo funkcije f_β primerno potabelirane (več o tem malo kasneje), da lahko do posamezne vrednosti $f_\beta(z)$ pridemo v $O(1)$ časa, bo gornji postopek izračunal potrebno število obiskov v samo $O(B)$ časa.

Poleg tega lahko tudi opazimo, da če pride v bloku β' do spremembe (če se doda ali pobriše eno od predavanj, ki pripadajo temu bloku), se zato spremeni funkcija $f_{\beta'}$, vse ostale f_β za $\beta \neq \beta'$ pa ostanejo nespremenjene. Tu se torej nakazuje način za poceni obravnavo vsake spremembe: na novo bo treba izračunati le eno od funkcij f_β; in kot bomo videli, se dá to izvesti v $O(B)$ časa, ker ima blok le $O(B)$ predavanj.

Za začetek bo koristno čase (vrednosti a_i in b_i), s katerimi delamo, malo skomprimirati. V vhodnih podatkih nastopajo časi z območja od 0 do 10^9; najprej jih uredimo naraščajoče in če kakšen od časov začetka a_i ni hkrati tudi čas konca kakšnega predavanja b_j (lahko tudi $j = i$), ga povečajmo do prvega naslednjega časa, ob katerem se konča kakšno predavanje. To smemo narediti, kajti če pogledamo naš prvotni požrešni algoritem, vidimo, da se v njem začetne čase uporablja le v pogoju „**if** $a_p > z$“, pri čemer je čas zadnjega obiska z vedno enak končnemu času kakšnega izmed prejšnjih predavanj. Če torej neki začetni čas a_p ni enak nobenemu končnemu času in ga povečamo do prvega naslednjega končnega časa, bo pogoj $a_p > z$ zdaj izpolnjen v enakih primerih kot prej.

Zdaj je torej začetni čas vsakega predavanja enak končnemu času nekega (lahko drugega, lahko istega) predavanja. Povsod, kjer se pojavlja najzgodnejši končni čas b_1 (kot končni čas enega ali več predavanj, morda pa tudi kot začetni čas nekaterih predavanj), spremenimo ta čas v 1; kjer koli se pojavlja drugi najzgodnejši končni čas b_2, ga spremenimo v 2; in tako naprej. Odslej bodo torej časi a_i in b_i števila z območja od 1 do m in ne več od 0 do 10^9.

Razmislimo zdaj o tem, kako izračunati in predstaviti funkcijo f_β za posamezen blok. Vrednost $f_\beta(z)$ bi lahko dobili tako, da bi naš prvotni požrešni algoritem pognali le na predavanjih trenutnega bloka: če je bil zadnji obisk ob času z, naj bo med predavanji z $a_i > z$ (torej takimi, ki se začnejo kasneje kot ob z) predavanje p tisto z najmanjšim b_i; naslednji obisk mora biti potem ob času b_p, od tam naprej pa lahko nadaljujemo enako, kot če bi računali $f_\beta(b_p)$.

funkcija $f_\beta(z)$:
1 $p := \arg\min_i\{b_i : i \textit{ je iz bloka } \beta \textit{ in } a_i > z\}$;
2 če takega predavanja sploh ni, vrni $(0, \text{NIL})$;
3 $(s', z') := f_\beta(b_p)$;
4 **if** $s' = 0$ **then** vrni $(1, b_p)$ **else** vrni $(s' + 1, z')$;

(Pogoj v vrstici 2 pride v poštev, če je z prevelik in se nobeno predavanje v bloku β ne začne po času z.)

Množico predavanj, ki v vrstici 1 ustrezajo pogoju „$a_i > z$“, bi bilo lažje opisati, če bi imeli predavanja tega bloka urejena po a_i; potem bi bila ta množica preprosto „zadnjih nekaj predavanj v bloku“ (koliko je ta nekaj, pa bi bilo odvisno od z). Ne bi bilo tudi težko za vsako tako množico vzdrževati minimuma vrednosti b_i po teh zadnjih nekaj predavanjih (to pa je potem b_p, ki ga potrebujemo v vrstici 3). Nadaljevanje postopka pa je potem odvisno le še od p (oz. od b_p), ne pa več neposredno od z-ja, s katerim smo začeli; možnih vrednosti funkcije $f_\beta(z)$ je torej le toliko, na kolikor načinov si lahko izberemo „nekaj“ v besedni zvezi „zadnjih nekaj predavanj v bloku“, to pa je $B + 1$; zato jih bomo lahko vse izračunali in potabelirali v $O(B)$ časa.

Recimo torej, da bi imeli predavanja v bloku urejena po začetnem času, tako da bo $a_1 \leq a_2 \leq \ldots \leq a_B$, če je B število predavanj v bloku. Naj bo potem $I_\beta[z]$

najmanjši i, pri katerem je $a_i > z$ (če takega sploh ni, ker je $z \geq a_B$, pa si mislimo $i = B + 1$). Takšne tabele ni težko pripraviti vnaprej (spomnimo se, da so možne vrednosti z-ja le cela števila od 1 do m, ker smo čase na začetku skomprimirali):

podprogram PripraviBlok(β):
  naj bo $(a_1, b_1), \ldots, (a_B, b_B)$ zaporedje vseh predavanj, ki pripadajo bloku β;
  uredi jih naraščajoče po a_i;
  naj bo $I_\beta[0..m]$ tabela celih števil;
  $i := B + 1$; **for** $z := m$ **downto** 0:
    $I_\beta[z] := i$;
    if $i > 1$ **and** $z = a_{i-1}$ **then** $i := i - 1$;

Zdaj imamo vse, kar potrebujemo, da lahko izračunamo vse možne vrednosti funkcije f_β. Pri tem imejmo v mislih, da od B predavanj, ki tvorijo naš blok, niso nujno vsa tudi res prisotna v vsakem trenutku (nekatera morda še niso bila dodana, nekatera pa so bila morda že pobrisana). V zanki bomo šli po predavanjih bloka od konca proti začetku in pri vsakem predavanju p izračunali $f_\beta(z)$ za tiste z, pri katerih je $a_{p-1} \leq z < p$, torej pri katerih ustrezajo pogoju „$a_i > z$“ predavanja od p-tega naprej. V $b_{\min}$ bomo računali najzgodnejši čas konca po teh predavanjih, kajti to je potem čas naslednjega obiska, če je bil prejšnji obisk ob času z. Ko se p zmanjšuje, se tudi $b_{\min}$ lahko le zmanjšuje, zato ga ni težko popravljati.

podprogram IzračunZaBlok(β):
  naj bo $F_\beta[1..B + 1]$ tabela parov oblike (*št. obiskov, čas zadnjega*);
  $F_\beta[B + 1] := (0, \text{NIL})$;
  $b_{\min} := \infty$; (* *v praksi lahko namesto* ∞ *vzamemo* m *)
  for $p := B$ **downto** 1:
    (* *V tej iteraciji zanke bi radi v* $F_\beta[p]$ *izračunali vrednost funkcije* $f_\beta(z)$ *za tiste* z*, pri katerih je* $I_\beta[z] = p$*, torej pri* $a_{p-1} \leq z < a_p$*.* *)
    if je p-to predavanje tega bloka trenutno prisotno:
      $b_{\min} := \min\{b_{\min}, b_p\}$;
    (* *Zdaj je* b_{min} *najmanjša vrednost* b_i *po prisotnih predavanjih izmed* $p, \ldots, B$*. To je čas naslednjega obiska, če je bil prejšnji ob kakšnem času* z *z območja* $a_{p-1} \leq z < a_p$*.* *)
    if $b_{\min} = \infty$ **then** $i := B + 1$ **else** $i := I_\beta[b_{\min}]$;
    $(s', z') := F_\beta[i]$; (* *to je* $f_\beta(b_{min})$ *)
    if $s' = 0$ **then** $F_\beta[p] := (1, b_{\min})$ **else** $F_\beta[p] := (s' + 1, z')$;

Zdaj imamo vse, kar potrebujemo za hitro računanje funkcije f_β; vrednost $f_\beta(z)$ je preprosto $F_\beta[I_\beta[z]]$.

Ideja rešitve je torej naslednja: za vsak blok izvedemo PripraviBlok, da dobimo tabelo I_β; po vsaki spremembi (dodajanju ali brisanju predavanja) izvedemo IzračunZaBlok za tisti blok, v katerem je prišlo do spremembe, in nato pokličemo PožrešniPoBlokih, da izračuna potrebno število obiskov po tej spremembi. Pri tem je le še ena težava: tabela I_β nam vzame $O(m)$ prostora, kar je v najslabšem primeru $O(n)$; in če bomo hkrati vzdrževali te tabele za vse bloke, ki jih je $O(B) = O(\sqrt{n})$, bodo zasedle skupno $O(n\sqrt{n})$ pomnilnika, s tem pa bomo presegli prostorsko omejitev pri tej nalogi (128 MB, število predavanj n pa gre lahko do 300 000).

Rekli smo, da bi po vsaki spremembi izvedli postopek PožrešniPoBlokih; torej ga bomo izvedli n-krat zaporedoma; toda v resnici ni treba teh n izvedb res izpeljati eno za drugo, lahko jih tudi prepletemo: najprej pri vseh n izvedbah obdelamo prvi blok, nato pri vseh obdelamo drugi blok in tako naprej. Tabele I_β za že obdelane bloke lahko pozabimo, zato bo dovolj imeti v pomnilniku samo tabelo za trenutni blok. Paziti pa moramo na to, da se lahko stanje bloka med različnimi klici postopka PožrešniPoBlokih tudi spreminja (če se ravno v tistem bloku dodajajo in brišejo predavanja); zato si bomo na začetku pripravili za vsak blok β seznam L_β vseh sprememb, ki se nanašajo na ta blok. Zapišimo zdaj glavni podprogram naše rešitve s psevdokodo:

```
 1   uredi vsa predavanja, ki jih omenjajo vhodni podatki, po b_i,
       skomprimiraj vse začetne in končne čase (na območje od 1 do m)
       in razdeli predavanja na B blokov s po približno B predavanji;
 2   for β := 1 to B do L_β := prazen seznam;
 3   for t := 1 to n:
 4     naj bo β blok, ki mu pripada predavanje, na katero se nanaša t-ta
         sprememba (brisanje ali dodajanje predavanja) v vhodnih podatkih;
 5     dodaj t na konec seznama L_β;
 6     s_t := 0; z_t := 0; (* dosedanje št. obiskov in čas zadnjega obiska
                              pri izračunu rezultata po t-ti spremembi *)
 7   for β := 1 to B:   (* po vseh blokih *)
 8     PripraviBlok(β); IzračunZaBlok(β);
 9     recimo, da je L_β = [t_1, t_2, ..., t_k];
10     i := 1; (* indeks naslednje spremembe v L_β *)
11     for t := 1 to n:
12       if i ≤ k and t = t_i:
13         izvêdi t-to spremembo (označi v bloku β ustrezno predavanje
             kot prisotno oz. odsotno);
14         IzračunZaBlok(β); i := i + 1;
15       (s', z') := F_β[I_β[z_t]];
16       if s' > 0 then s_t := s_t + s', z_t := z';
17     pozabi F_β in I_β;
18   for t := 1 to n do izpiši s_t;
```

Glavna zanka (vrstice 7–17) gre torej po blokih, pri vsakem bloku pa z vgnezdeno zanko (vrstice 11–16) dopolnimo vseh n rezultatov, ki jih bomo morali na koncu izpisati (vrstica 18): vsakemu rezultatu s_t prištejemo število obiskov v trenutnem bloku in si v z_t zapomnimo čas zadnjega od njih (vrstici 15–16). Med temi izračuni v primernih trenutkih tudi izvajamo spremembe v bloku in po vsaki na novo izračunamo tabelo F_β (vrstice 12–14). Ko smo z obdelavo trenutnega bloka končali, lahko tabeli F_β in I_β zanj pozabimo oz. ju povozimo s $F_{\beta+1}$ in $I_{\beta+1}$ pri naslednjem bloku.

Imamo B klicev PripraviBlok, ki vzamejo vsak po $O(m)$ časa, in $n + B$ klicev IzračunZaBlok, ki vzamejo vsak po $O(B)$ časa; ker je $B = O(\sqrt{n})$ in $m = O(n)$, je časovna zahtevnost te rešitve $O(n\sqrt{n})$. Prostorska zahtevnost je le $O(n)$: hraniti moramo podatke o vseh predavanjih, nastajajoče rezultate s_t in z_t ter po eno tabelo I_β. Tu opisano rešitev, ki gre v glavni zanki po blokih in tabele I_β sproti pozablja (namesto da bi šla v glavni zanki po spremembah in imela hkrati v pomnilniku

tabele I_β za vse bloke), smo sicer zasnovali na tak način zato, da smo zmanjšali porabo pomnilnika z $O(n\sqrt{n})$ na $O(n)$; izkaže pa se, da je zaradi te spremembe tudi približno trikrat hitrejša.

B. Podajanje žoge

Vhodno zaporedje otrok lahko razdelimo na dve, eno za dečke in eno za deklice, in potem obravnavamo vsake posebej ter na koncu seštejemo rezultate. Otroci v vsakem od tako dobljenih dveh zaporedij naj bodo seveda še vedno v enakem medsebojnem vrstnem redu, v kakršnem so bili v vhodnem zaporedju; tako tvori vsako od obeh novih zaporedij še vedno oglišča strogo konveksnega mnogokotnika. V nadaljevanju našega razmisleka torej predpostavimo, da imamo pred seboj zaporedje otrok enega spola in lahko torej v par povežemo katerakoli dva od njih.

Recimo, da bi bili otroci enakomerno razporejeni na krožnici; črta med dvema otrokoma v paru je potem vedno neka tetiva krožnice, najdaljša možna tetiva pa je tista med dvema točkama na točno nasprotni strani krožnice (180 stopinj narazen), ko je dolžina tetive enaka premeru krožnice. Takrat bi bilo torej pametno vsakega otroka dati v par s tistim na nasprotni strani krožnice.

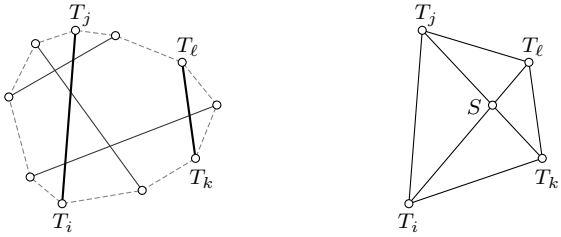
Poskusimo uporabiti to intuicijo tudi na našem primeru. Recimo, da imamo $2m$ otrok (saj naloga zagotavlja, da je otrok vsakega spola sodo mnogo), ki stojijo (po vrsti vzdolž našega konveksnega mnogokotnika) na točkah $T_1, T_2, \ldots, T_{2m}$. Naš razmislek iz prejšnjega odstavka zdaj namiguje, da bi bilo pametno vzeti pare (T_i, T_{i+m}) za $i = 1, 2, \ldots, m$.

Prepričajmo se, da je to res. Pa recimo, da bi bila najboljša rešitev taka, v kateri za vsaj en i otrok T_i ni v paru s T_{i+m}, pač pa z nekim drugim otrokom T_j. Brez izgube za splošnost predpostavimo, da je $j > i$. Med otrokoma T_i in T_j je na eni strani $j - i - 1$ otrok (namreč $T_{i+1}, \ldots, T_{j-1}$), na drugi strani pa $2m - 1 - j + i$ (namreč $T_{j+1}, \ldots, T_{2m}, T_1, \ldots, T_{i-1}$). Na eni strani je torej več otrok kot na drugi (kajti enakost $j - i - 1 = 2m - 1 - j + i$ bi veljala le pri $j = i + m$, mi pa smo predpostavili ravno nasprotno, torej da je $j \neq i + m$). Zato ni mogoče, da bi bili otroci s tiste strani, kjer jih je več, vedno v paru z otroki z druge strani; obstajati mora vsaj en par, kjer sta oba otroka s tiste strani, kjer jih je več. Recimo, da sta to T_k in T_ℓ z območja $i < k < \ell < j$ (razmislek za primer, ko sta k in ℓ na drugi strani, je seveda čisto podoben).

Štirikotnik $T_iT_kT_\ell T_j$ je strogo konveksen, saj so njegova oglišča tudi oglišča našega prvotnega strogo konveksnega mnogokotnika iz vhodnih podatkov. Diagonali T_iT_ℓ in T_kT_j se zato sekata v neki točki, ki ji recimo S in ki tudi leži v notranjosti štirikotnika. Le-tega si lahko torej predstavljamo kot razdeljenega na štiri trikotnike: ST_iT_k, ST_kT_ℓ, $ST_\ell T_j$ in ST_jT_i. Ker je S v notranjosti, so to res pravi trikotniki in gotovo niso izrojeni v daljice; zato v vsakem od njih velja trikotniška neenakost: posamezna stranica je krajša kot ostali dve skupaj. Zato je

$$\begin{aligned} T_iT_j + T_kT_\ell &< (T_iS + ST_j) + (T_kS + ST_\ell) \\ &= (T_iS + ST_\ell) + (T_kS + ST_j) \\ &= T_iT_\ell + T_jT_k. \end{aligned}$$

V prvi vrstici smo torej uporabili trikotniško neenakost, nato pa člene malo preuredili



Primer za $m = 5$. Na levi sliki je T_i v paru s T_j, ki ni točno 5 točk naprej od njega, zato je (če gremo po robu konveksnega mnogokotnika) na eni strani med njima manj točk kot na drugi (3 proti 7) in obstaja par (T_k, T_ℓ), v katerem sta obe točki na isti strani daljice T_iT_j. Na desni sliki vidimo štirikotnik $T_iT_kT_\ell T_j$, v katerem je vsota diagonal $T_iT_\ell + T_jT_k$ daljša od vsote dveh stranic $T_iT_j + T_kT_\ell$.

in upoštevali, da S leži na diagonali T_iT_ℓ, zato je $T_iS + ST_\ell = T_iT_\ell$ (in podobno za T_jT_j).

Ta razmislek nam je torej pokazal, da če bi v naši rešitvi zamenjali para (T_i, T_j) in (T_k, T_ℓ) s paroma (T_i, T_ℓ) in (T_j, T_k), bi se skupna dolžina metov povečala, torej bi bila tako popravljena rešitev boljša kot pred popravkom. To pa je v protislovju s predpostavko, da smo že pred popravkom vzeli najboljšo rešitev.

Tako torej vidimo, da je najboljša rešitev res tista, v kateri je vsak T_i v paru s T_{i+m}. Ni torej treba drugega, kot da izračunamo dolžine daljic T_iT_{i+m} za $i = 1, \ldots, m$ in jih seštejemo. To naredimo posebej za dečke in posebej za deklice ter izpišemo skupno vsoto po obojih.

C. Torte

Za začetek od prodajne cene vsake torte odštejmo ceno sestavin zanjo; prvotno vrednost c_k torej zamenjajmo s $c_k - \sum_{\ell=1}^{G} a_{k\ell} g_\ell$. Če je kakšna od teh razlik negativna, lahko takoj zaključimo, da se tiste torte nikakor ne splača peči, ceno c_k pa lahko postavimo kar na 0. V nadaljevanju našega razmisleka se nam s sestavinami ni več treba ukvarjati, saj smo njihovo ceno že upoštevali v novih vrednostih c_k.

Preostanek naloge lahko prevedemo na problem najmanjšega prereza v grafu. Sestavimo usmerjen graf, ki ima po eno točko x_k za vsako torto, po eno točko y_ℓ za vsako orodje in še dve posebni točki — izvor s in ponor t. V ta graf dodajmo naslednje povezave: $s \to x_k$ s kapaciteto c_k (za vsako torto k); $y_\ell \to t$ s kapaciteto t_ℓ (za vsako orodje ℓ); in $x_k \to y_\ell$ z neskončno kapaciteto za vsako torto k in za vsako orodje ℓ, ki ga potrebujemo pri izdelavi te torte. Množico vseh točk označimo z V.

Spomnimo se, da je *prerez* grafa definiran kot táko razbitje točk na dve skupini S in $T = V - S$, pri katerem prva vsebuje izvor, druga pa ponor; *cena* prereza pa je potem definirana kot vsota kapacitet vseh tistih povezav $u \to v$, ki gredo iz S v T (torej kjer je $u \in S$ in $v \in T$). Izziv pri prerezih je ponavadi to, kako najti čim cenejšega. Kaj to pomeni pri našem grafu? Če torto x_k postavimo v množico S, moramo postaviti v S tudi vsa orodja y_ℓ, ki jih ta torta potrebuje, kajti drugače bo povezava $x_k \to y_\ell$ štela v ceno prereza, ki bo zato postala neskončno velika. Po drugi strani, če je v S neko orodje y_ℓ, ki ga ne potrebuje nobena torta iz S, lahko prerez izboljšamo, če to orodje premaknemo v T, kajti potem povezava $y_\ell \to t$ ne

bo več štela v ceno prereza (povezave, ki kažejo iz tort v y_ℓ, pa tudi ne bodo štele v ceno prereza, saj smo rekli, da v S ni nobene take torte, ki bi zahtevala orodje y_ℓ). Pri najcenejšem prerezu je torej S sestavljena iz neke množice (0 ali več) tort ter iz tistih (in samo tistih) orodij, ki so potrebna za izdelavo teh tort.

V ceno takega prereza štejejo povezave $s \to x_k$ za tiste torte, ki jih ni v S (torej torte, ki jih ne izdelamo), in povezave $y_\ell \to t$ za tista orodja, ki so v S (torej orodja, ki jih moramo kupiti, da lahko izdelamo torte, ki smo se jih namenili izdelati). Cena prereza je torej vsota „oportunitetnih stroškov“ (prihodka, ki bi ga imeli s prodajo tistih tort, ki jih ne bomo naredili), ter (dejanskih) stroškov, ki jih imamo z nakupom orodij (za torte, ki jih bomo naredili). Zapišimo to simbolično:

$$\begin{aligned}[\text{cena prereza}] &= [\text{prihodek od ne-izdelanih tort}] + [\text{cena kupljenih orodij}] \\ &= [\text{prihodek od vseh tort}] - [\text{prihodek od izdelanih tort}] + \\ &\qquad\qquad [\text{cena kupljenih orodij}] \\ &= [\text{prihodek od vseh tort}] - ([\text{prihodek od izdelanih tort}] - \\ &\qquad\qquad [\text{cena kupljenih orodij}]) \\ &= [\text{prihodek od vseh tort}] - [\text{čisti dobiček}].\end{aligned}$$

Razlika med prihodki od izdelanih tort ter ceno kupljenih orodij, ki smo jo tu v zadnji vrstici imenovali „čisti dobiček“, je točno tisto, kar moramo pri tej nalogi maksimizirati. Po drugi strani je prvi člen, prihodek od vseh tort, konstanten (in znaša $\sum_{k=1}^{C} c_k$). Čisti dobiček je torej razlika med prihodkom od vseh tort in ceno prereza; največji možni dobiček dobimo pri najcenejšem možnem prerezu.[19]

Kot je znano, je cena najcenejšega prereza enaka maksimalnemu pretoku skozi graf (od s do t), zato jo lahko poiščemo s kakšnim od znanih algoritmov za iskanje maksimalnega pretoka, na primer Ford-Fulkersonovim oz. z njegovimi raznimi bolj konkretnimi različicami, kot sta Edmonds-Karpov algoritem ter Diničev algoritem.[20]

D. Sušenje perila

Uredimo za začetek rjuhe po počasnem času sušenja p_i in jih v tem vrstnem redu oštevilčimo, tako da bo veljalo $p_1 \le p_2 \le \ldots \le p_n$. Edini razporedi rjuh, o katerih je smiselno razmišljati, so potem tisti, pri katerih prvih nekaj rjuh, recimo od 1 do k, sušimo čez eno vrv, ostale rjuhe, od $k+1$ do n, pa čez obe vrvi. To je posledica dejstva, da če rjuho i sušimo čez obe vrvi, rjuho $i+1$ pa čez eno, se prva od njiju

[19] Lahko si tudi predstavljamo, da smo vnaprej dobili denar za vseh C tort, na koncu pa ga bomo morali vračati za tiste torte, ki jih ne bomo spekli. Pri takem pogledu na nalogo potem ne gre več za maksimizacijo razlike med prihodki in stroški, ampak le še za minimizacijo stroškov: če spečemo torto, se izognemo stroškom z vračanjem denarja, vendar si nakopljemo stroške z nakupom orodja; če pa je ne spečemo, je ravno obratno. Prav te dvoje stroške pa meri cena prereza, ki jo moramo torej minimizirati.

[20] Pri splošnem Ford-Fulkersonovem algoritmu se načeloma lahko zgodi (če imamo dovolj neugoden graf in dovolj smole pri izbiri poti, ki jo bomo zasitili kot naslednjo), da moramo zasititi $O(f)$ poti, preden se postopek ustavi, pri čemer je f maksimalni pretok v našem grafu. Pri naši nalogi so kapacitete povezav dovolj velike, da bi nas lahko skrbelo, da bo takšna rešitev presegla časovno omejitev. Edmonds-Karpov in Diničev algoritem izbirata pot bolj pazljivo in te slabosti nimata. Še ena možnost je *capacity scaling:* začnemo s praznim grafom in poganjamo Ford-Fulkersonov algoritem v več fazah, pri čemer pred k-to fazo dodamo v graf povezave, katerih kapaciteta ni za več kot 2^k-krat manjša od maksimalne kapacitete. V praksi pa sicer ni na testnih primerih z našega tekmovanja nič od tega zares nujno potrebno; tudi ko smo implementirali Ford-Fulkersonov algoritem tako, da je vsakič izbral tisto pot, ki *najmanj* poveča pretok po grafu, je bil program še daleč od tega, da bi prekoračil časovno omejitev.

suši h_i časa, druga pa p_{i+1}; skupni čas sušenja (ki je maksimum časov sušenja posameznih rjuh) je torej vsaj p_{i+1}; in če potem spremenimo naš razpored tako, da rjuho i obesimo samo čez eno vrv, bo razpored še vedno veljaven (in bo celo na eni rjuhi za d_i več prostora kot prej, ker bo rjuha i zasedala prostor le na eni vrvi namesto na obeh), rjuha i pa bo v čas sušenja prispevala p_i namesto h_i; in ker je $p_i \le p_{i+1}$, ne bo skupni čas sušenja nič slabši kot prej, saj je bil že prej vsaj p_{i+1} in rjuha i se bo v tem času zagotovo posušila tudi zdaj. Vsak razpored, ki ni oblike „prvih nekaj rjuh se suši čez eno vrv, ostale pa čez obe“, lahko torej predelamo v razpored te oblike, ne da bi se mu skupni čas sušenja pri tem kaj povečal.

Če sušimo prvih k rjuh čez eno vrv in ostale čez obe, je čas sušenja enak

$$\begin{aligned} c_k &:= \max\{p_1, \ldots, p_k, h_{k+1}, \ldots, h_n\} \\ &= \max\{p_k, h_{k+1}, \ldots, h_n\}, \end{aligned}$$

kjer smo v drugi vrstici upoštevali, da je $p_1 \le p_2 \le \ldots \le p_k$. Možni kandidati za čas sušenja so torej le $c_0, c_1, \ldots, c_n$. Opazimo lahko, da je to zaporedje naraščajoče; o tem se prepričamo takole: če v prvi vrstici gornje formule namesto k uporabimo $k+1$, se v $\max\{\ldots\}$ spremeni le to, da se vrednost h_{k+1} zamenja s p_{k+1}. Ker je $p_{k+1} \ge h_{k+1}$, se maksimum lahko le poveča ali ostane enak, ne more pa se zmanjšati.

Več rjuh ko sušimo čez obe vrvi, več prostora zasedejo in daljše vrvi potrebujemo. Vprašajmo se torej, pri katerih dolžinah vrvi L je kandidat c_k (kjer sušimo prvih k rjuh čez eno vrv, ostale pa čez obe) sploh mogoč. Naj bo $E_k := d_1 + \ldots + d_k$ dolžina rjuh, ki jih sušimo čez eno vrv, in $D_k := E_n - E_k = d_{k+1} + \ldots + d_n$ dolžina tistih, ki jih sušimo čez obe vrvi. Od rjuh, ki jih sušimo čez eno vrv, jih gre nekaj na prvo vrv, nekaj pa na drugo; recimo, da je u skupna dolžina tistih na prvi vrvi; na drugi je potem njihova skupna dolžina $E_k - u$. Brez izgube za splošnost recimo, da je prva vrv tista, na kateri je ta dolžina večja; torej je $u \ge E_k/2$. Poleg teh rjuh je seveda na vsaki vrvi še D_k enot dolžine zasedenih zaradi rjuh, ki so obešene čez obe vrvi. Tako mora biti torej prva vrv dolga vsaj $D_k + u$, druga pa $D_k + (E_k - u)$, kar je manj ali enako kot pri prvi. Če hočemo, da je to izvedljivo s čim krajšima vrvema, mora biti torej u čim manjši. Naj bo torej u_k najmanjša taka skupna dolžina rjuh, ki je še $\ge E_k/2$ in ki jo je mogoče dobiti tako, da seštejemo dolžine nekaterih izmed prvih k rjuh. Poiščemo jo lahko tako, da z dinamičnim programiranjem rešimo problem nahrbtnika. Naj bo $f_k(u)$ logična vrednost (0 ali 1), ki nam pove, ali je mogoče izmed prvih k rjuh izbrati nekaj rjuh tako, da se njihove dolžine seštejejo natanko v u; potem je

$$f_k(u) = f_{k-1}(u) \vee f_{k-1}(u - d_k),$$

kjer prvi člen predstavlja možnost, da k-te rjuhe ne uporabimo, drugi pa, da jo uporabimo. Robni primeri so $f_k(u) = 0$ za $u < 0$, $f_k(0) = 1$ in $f_0(u) = 0$ za $u \ne 0$. Vrednost u_k je potem

$$u_k = \min\{u : u \ge E_k/2 \wedge f_k(u)\},$$

najmanjša dolžina vrvi, pri kateri je tak razpored mogoč, pa je $L_k := D_k + u_k$.

Spomnimo se, da je vsak naslednji kandidat slabši od prejšnjega: $c_0 \le c_1 \le \ldots \le c_n$. Ko se torej pri neki dolžini vrvi ℓ sprašujemo, kako čim hitreje posušiti perilo, moramo vzeti najmanjši k, pri katerem je c_k možen kandidat, torej pri katerem je

$\ell \geq L_k$. Ni se težko prepričati, da je L_k padajoče zaporedje v odvisnosti od k;[21] torej lahko najmanjši k, pri katerem je $\ell \geq L_k$, poiščemo z bisekcijo. Koristno je torej, če si najprej pripravimo seznam vseh kandidatov c_k in njihovih pripadajočih L_k, potem pa bomo lahko z bisekcijo hitro odgovorili na vsako od poizvedb (dolžin $\ell_1, \ldots, \ell_q$) iz vhodnih podatkov. Zdaj imamo vse, kar potrebujemo, da lahko zapišemo psevdokodo naše rešitve:

```
 1  uredi rjuhe naraščajoče po p_i;
 2  E_0 := 0; D_0 := sum_{i=1}^{n} d_i; c_0 := max{h_1, ..., h_n};
 3  ℓ_max := max_j ℓ_j;
 4  f_0 := zaporedje 1 + ℓ_max bitov, vsi so ničle, le prvi je enica;
 5  L := prazno zaporedje;
 6  for k := 0 to n:
 7    if k > 0:
 8      f_k := f_{k-1} or (f_{k-1} shl d_k);
 9      E_k := E_{k-1} + d_k; D_k := D_{k-1} - d_k; c_k := max{c_{k-1}, p_k};
10    u := ⌈E_k/2⌉;
11    while not f_k[u] do u := u + 1;
      (* Zdaj ima spremenljivka u vrednost u_k. *)
12    dodaj D_k + u (to je enako L_k) na konec zaporedja L;
13  for j := 0 to q:
14    z bisekcijo po L poišči najmanjši k, pri katerem je ℓ_j ≥ L_k;
15    če takega sploh ni, izpiši −1, sicer pa izpiši c_k;
```

Časovna zahtevnost tega postopka je načeloma $O(n \cdot \ell_{\max} + q \log n)$, pri čemer močno prevladuje prvi člen (drugi je zaradi bisekcije pri odgovarjanju na poizvedbe). Posamezno funkcijo f_k si lahko predstavljamo kot zaporedje bitov; koristno jo je predstaviti kot bitno karto, torej kot tabelo 64-bitnih celih števil, v kateri vsak element hrani 64 vrednosti funkcije. To nam omogoča, da f_k s pomočjo bitnih operacij hitro izračunamo iz f_{k-1} (v vrstici 8; pri tem si tudi predstavljajmo, da pri zamiku za d_k bitov v levo obdržimo le spodnjih $1+\ell_{\max}$ bitov rezultata), pa tudi pri zanki v vrstici 11 lahko prihranimo čas, če za vsak sklop 64 bitov funkcije f_k najprej preverimo, ali je ta sklop kot celo število različen od 0 tam sploh kak bit prižgan (če ni, se s posameznimi biti ni treba ukvarjati, saj so vsi ugasnjeni). S takšno implementacijo sicer v asimptotičnem smislu ničesar ne pridobimo, je pa naš program hitrejši za neki zajeten konstanten faktor (pri naših poskusih od 10-krat do 30-krat, odvisno od računalnika in prevajalnika); brez te izboljšave bo naša rešitev prekoračila časovno omejitev (3 sekunde, pri čemer gre lahko število rjuh n do 30 000, dolžine vrvi pa do 300 000).

Za lažjo in hitrejšo implementacijo (kar je še posebej pomembno na tekmovanju) pride prav razred `bitset` iz C++ove standardne knjižnice, s katerim lahko vrstico 8 našega gornjega postopka implementiramo preprosto kot `f |= f << d_k`. Za vrstico 11 našega postopka pride prav metoda `_Find_next` razreda `bitset`, ki poišče naslednji

[21] Razmislimo, kaj se zgodi, ko gremo s $k-1$ na k. Spomnimo se, da je u_{k-1} najmanjši $u \geq E_{k-1}/2$, za katerega je $f_{k-1}(u)$ prižgan. Ker je $f_{k-1}(u_{k-1})$ prižgan, je prižgan tudi $f_k(u')$ za $u' = u_{k-1} + d_k$, slednje pa je $\geq E_{k-1}/2 + d_k \geq (E_{k-1} + d_k)/2 = E_k/2$. Torej je u' eden od kandidatov za u_k, torej je $u_k \leq u'$, torej $L_k = D_k + u_k \leq D_k + u' = (D_{k-1} - d_k) + (u_{k-1} + d_k) = D_{k-1} + u_{k-1} = L_{k-1}$, torej je zaporedje $L_0, \ldots, L_k$ res padajoče.

prižgani bit; ta metoda sicer ni standardna, je pa na voljo na primer v g++ovi implementaciji standardne knjižnice. Vendar pa je bilo pri naših poskusih enako hitro tudi iskanje naslednjega prižganega bita v bitsetu z zanko (**while** (! f[L_k]) ++L_k), kar ne uporablja nobene nestandardne razširitve.

E. Enaki urniki

Vhodne podatke berimo po vrsticah in pri tem vzdržujmo slovar, kjer so ključi imena delavcev, pripadajoča vrednost pa je razlika (med drugim in prvim urnikom) v številu dni, ko je ta delavec v pripravljenosti. Ko torej preberemo vrstico „$s_i\ e_i\ u_i$“, moramo v slovarju poiskati zapis za ime u_i (če ga še ni, ga dodajmo z začetno razliko 0) in njegovo razliko nato zmanjšati (če smo pri prvem urniku) oz. povečati (če smo pri drugem urniku) za $e_i - s_i$. Ko pridemo do konca obeh urnikov, se sprehodimo po slovarju in neničelne razlike izpišimo, pri tem pa še v neki logični spremenljivki hranimo podatek o tem, ali smo sploh našli kakšno tako razliko. Če pridemo do konca, ne da bi zaznali razlike, izpišimo „`No differences found.`“.

Ker so testni primeri pri tej nalogi majhni, ni zelo pomembno, kakšno implementacijo slovarja uporabimo; dovolj dobro bi bilo že, če bi v neurejenem seznamu oz. tabeli hranili pare ⟨*ime delavca*, *razlika*⟩, še bolje pa je uporabiti razpršeno tabelo (npr. razred unordered_map v C++) ali drevo (razred map v C++). Slednje ima tudi to prednost, da so imena v njem že urejena abecedno, zato se moramo pri izpisu le sprehoditi po njih; če uporabimo neurejen seznam, ga bomo morali pred izpisom še urediti, če pa uporabimo razpršeno tabelo, bomo morali zapise iz nje (vsaj tiste z neničelno razliko) pred izpisom tudi zložiti v neki seznam in ga urediti.

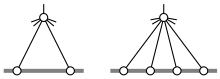
F. Filogenetika

Naloga je sestavljena iz dveh ali treh delov: najprej moramo ugotoviti, katera vozlišča so notranja in katera so listi drevesa; potem si lahko poljubno notranje vozlišče izberemo za koren in otroke vsakega notranjega vozlišča uredimo tako, da bodo na koncu listi prišli v prav tak vrstni red, v kakršnem si sledijo na ciklu povezav, s katerimi so povezani; in končno bomo v tako organiziranem drevesu prešteli barvanja z dinamičnim programiranjem (in pri tem pazili, da bo postopek dovolj učinkovit).

Rekonstrukcija drevesa. Graf, ki ga dobimo na vhodu, je nastal tako, da je nekdo vzel drevo (vloženo v ravnino) in z dodatnimi povezavami povezal vse liste drevesa v cikel. Listi so imeli v drevesu stopnjo 1, nato so dobili vsak po dve povezavi (s prejšnjim in naslednjim listom v ciklu), torej imajo zdaj stopnjo 3; notranjim vozliščem pa se stopnja ni spremenila in imajo torej še vedno stopnjo 2 ali več. Če je v prvotnem drevesu obstajala povezava med listom u in notranjim vozliščem v, bomo rekli, da je u *otrok* vozlišča v, ta pa *starš* vozlišča u; vsak list ima natanko enega starša.

Najmanjši možni graf, ki ga lahko pri tej nalogi dobimo, je tetraeder, torej poln graf na štirih vozliščih: eno notranje vozlišče in trije listi, povezani v cikel (slika 1; z debelimi sivimi črtami bomo risali povezave, ki pripadajo ciklu listov). Če smo na vhodu dobili kak večji graf, ga bomo zdaj v več korakih postopoma, z združevanjem sosednjih vozlišč, skrčili v tetraeder.

Slika 1. Tetraeder.



Slika 2. Trikotniki.

Vozlišča s stopnjo natanko 2 bi nam bila v nadaljevanju nekoliko v nadlego, zato se jih znebimo: če ima u natanko dva soseda, v in w, ga združimo z enim od teh dveh sosedov, recimo z v; povezava (u, v) ob tem izgine, novo združeno vozlišče pa ima povezavo z w (in z vsemi sosedi nekdanjega v-ja).[22] (Te stvari si zapomnimo, da bomo lahko kasneje združena vozlišča spet razdružili.) Odslej torej predpostavimo, da ima vsako vozlišče stopnjo vsaj 3.

Če imata kje dva zaporedna lista s cikla listov istega starša, tvorita skupaj s tem staršem trikotnik; in v splošnem, če ima k zaporednih listov istega starša, tvorijo skupaj z njim „grozd“ $k - 1$ trikotnikov, kot kaže slika 2.

To pa je tudi edini način, da se v našem grafu pojavi trikotnik (razen če se je naš graf že skrčil v tetraeder; takrat je tudi cikel listov samo še trikotnik, toda takrat s postopkom krčenja grafa tudi končamo). V prvotnem drevesu trikotnikov namreč ni bilo, saj je drevo acikličen graf. Trikotnik torej vsebuje vsaj eno povezavo, ki je prišla v graf ob dodajanju cikla listov. Če bi vseboval natanko dve taki povezavi, bi morala tretja (ki bi bila iz drevesa, ne s cikla listov) neposredno povezovati dva lista; toda v drevesu ni nobene povezave, ki bi nesporedno povezovala dva lista. Če pa bi naš trikotnik vseboval tri povezave s cikla listov, bi to pomenilo, da so listi le trije in imamo tetraeder.

Oglejmo si zdaj dva načina, kako bomo krčili graf (gl. sliko 3).

(1) *Združevanje trikotnika.* Če najdemo vozlišče u stopnje 3, ki ima med drugim dva taka soseda v in w, ki sta tudi sama stopnje 3 in neposredno povezana s povezavo, to pomeni, da je eno od teh treh vozlišč notranje, drugi dve pa sta njegova edina otroka, kajti tretja povezava tistega notranjega vozlišča mora voditi naprej po drevesu, do nekega drugega notranjega vozlišča (sicer se v drevesu iz naših treh vozlišč ne bi dalo priti v preostanek drevesa, kar bi bilo v protislovju z dejstvom, da je drevo povezan graf). V tem primeru bomo vozlišča u, v in w združili v eno samo novo vozlišče. Učinek na graf je tak, kot da bi v drevesu poiskali neko notranje vozlišče, ki ima za sosede dva lista in še neko notranje vozlišče, in bi tista dva lista pobrisali, njun dosedanji skupni sosed (in starš) pa bi se s tem iz notranjega vozlišča spremenil v list. Graf smo torej zmanjšali za dve vozlišči. Za cikel listov, ki je doslej tekel skozi pravkar pobrisana lista, si lahko predstavljamo, da teče zdaj skozi njunega nekdanjega starša.

(2) *Združevanje pri zaporedju vsaj treh vozlišč, ki so vsa stopnje 3 in imajo skupnega soseda.* Če najdemo vozlišče u stopnje 3, ki ima dva taka soseda v in w,

[22] Ali se lahko zgodi, da nam zaradi tega nastane v grafu par vzporednih povezav? To bi se zgodilo, če bi bila v in w povezana že od prej. Toda ta domnevna že obstoječa povezava (v, w) je tvorila skupaj s povezavama (u, v) in (v, w) cikel; ker v drevesu ni ciklov, mora torej vsaj ena od teh povezav pripadati ciklu listov, torej sta dve izmed točk u, v in w lista; ker ima u stopnjo 2, ne more biti list, saj imajo ti zdaj stopnjo 3; torej sta lista v in w; ker je bil v drevesu u povezan le z njima, to pomeni, da se iz v in w ni dalo priti nikamor drugam kot v u. Ker pa je bilo drevo povezano, je to mogoče le, če ni obsegalo poleg u, v in w nobenega drugega vozlišča; torej sta bila lista v drevesu le dva, v in w, kar pa je nemogoče, saj naloga zagotavlja, da so bili listi vsaj trije.

Združitev tipa 1. Združitev tipa 2.

Slika 3.

Združitev tipa 1. Združitev tipa 2.

Slika 4. Razveljavljanje združitev.

(*a*) če je p sosed u-ja, ne pa w-ja (*b*) če je p sosed w-ja, ne pa u-ja

Slika 5. Težave pri razveljavljanju združitev drugega tipa.

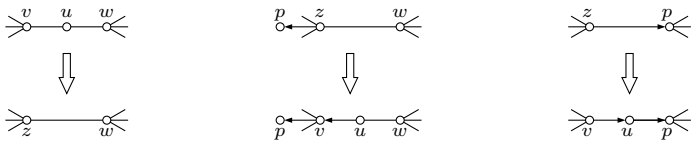
ki sta neposredno povezana s povezavo in je w stopnje 3, vozlišče v pa stopnje > 3; in če je poleg tega w-jev tretji sosed (tisti, ki ni niti u niti v) — recimo mu x — tudi neposredno povezan z v, potem lahko točki u in w združimo v eno samo novo točko. Učinek na graf je tak, kot da bi v drevesu poiskali neko notranje vozlišče, ki ima za otroke med drugim vsaj tri zaporedne liste (zaporedne na ciklu listov), in bi zdaj drugega od teh treh otrok pobrisali. Graf smo torej zmanjšali za eno vozlišče.

Prepričajmo se, da je (če naš graf še ni tetraeder) vedno gotovo mogoče izvesti vsaj eno od teh dveh krčitev. To sledi iz dejstva, da nekje v drevesu gotovo obstaja tako notranje vozlišče v, ki ima za sosede same liste, razen morda enega (ki je notranje vozlišče).[23] Ker ima v stopnjo vsaj 3, sta torej med njegovimi sosedi vsaj dva lista, lahko pa jih je tudi več; v vsakem primeru pa se držijo skupaj na ciklu listov in zato tvorijo skupaj z v-jem trikotnike.

Novo vozlišče, ki nastane s skrčitvijo (na gornjih slikah smo ga imenovali z), lahko kasneje seveda tudi sámo sodeluje v novih skrčitvah. Tako nadaljujemo, dokler nam ne ostanejo le še štiri vozlišča, ki takrat tudi neizogibno tvorijo tetraeder. Izberimo si eno od preostalih vozlišč za koren drevesa; recimo mu r. O tem, kako izbrati r, bomo razmislili malo kasneje. V drevesu obstaja med vsakima dvema vozliščema natanko ena pot; starša poljubnega vozlišča u lahko zdaj definiramo kot neposrednega predhodnika u-ja na tisti edini poti od r do u. (Za liste se ta definicija staršev ujema s tisto, ki smo jo podali že na začetku naše rešitve.)

Pojdimo zdaj po vseh izvedenih krčitvah v obratnem vrstnem redu (od najkasneje izvedenih proti zgodnejšim) in jih razveljavljajmo, torej združena vozlišča spet

[23] Pa recimo, da to ne drži. Vsako notranje vozlišče ima torej med sosedi vsaj dve drugi notranji vozlišči: notranje vozlišče v_1 ima recimo med sosedi notranji vozlišči u_2 in v_2; tudi v_2 ima med sosedi vsaj dve notranji vozlišči; eno je v_1, mora pa obstajati še vsaj eno drugo, recimo v_3; enako vidimo, da mora imeti v_3 za soseda še neko novo notranje vozlišče v_4 in tako naprej. Ker je graf končno velik, se morajo v zaporedju $v_1, v_2, v_3, \ldots$ vozlišča prej ali slej začeti ponavljati; toda to pomeni, da je že v drevesu obstajal cikel, kar pa je protislovje.



Združitev. Razdružitev, p je sosed v-ja. Razdružitev, $p = w$.

Slika 6. Obravnava vozlišč stopnje 2.

razdružimo, pri tem pa primerno računajmo podatke o starših vsakega vozlišča (slika 4; usmerjene povezave kažejo od otrok na starše). Recimo, da ima točka z starša p in da je nastala s krčitvijo. (1) Če je bila to krčitev prvega tipa iz vozlišč u, v in w, poglejmo, katera od teh treh točk je imela p za sosedo. Ona naj dobi p za svojega starša, drugi dve točki pa postaneta njena otroka. (Poseben primer: morda je že r nastal s krčitvijo prvega tipa iz vozlišč u, v in w; tedaj si izberimo poljubno od teh treh in v bodoče njega štejmo za koren drevesa, ostali dve pa za njegova otroka.) (2) Če pa je bila to krčitev drugega tipa iz točk u in w, naj tidve točki obe postaneta otroka točke p. To sicer deluje le, če je p skupna soseda točk u in w. Če je p ena od ostalih dveh sosed točke z — torej tista, ki je po razdružitvi z-ja v u in w soseda le u-ja ali le w-ja, ne pa obeh — bi sicer načeloma lahko zdaj tisto izmed u in w, ki je soseda p-ja, razglasili za starša druge, vendar se v takem primeru kasneje neizogibno izkaže, da dobljena rekonstrukcija ne bo veljavna (povezave, ki ne bodo pripadale rekonstruiranemu drevesu, ne bodo tvorile cikla listov).[24] To je potem znak, da smo začeli z napačnim r in moramo poskusiti s kakšnim drugim.

(3) Na koncu razveljavimo še združitve, ki smo jih izvedli na začetku zaradi točk stopnje 2 (slika 6). Če je imel u le dva soseda, v in w, in je bil zato združen s svojim sosedom v, ju zdaj razklenemo; če je bil starš združenega vozlišča w, naj w zdaj postane u-jev starš, ta pa v-jev; sicer pa je bil starš združenega vozlišča eden od v-jevih sosedov, ki naj zdaj postane v-jev starš, ta pa u-jev.

Razmislimo zdaj o tem, kako izbrati koren r. Težava nastopi predvsem, če izberemo vozlišče, ki v prvotnem drevesu v resnici ni moglo biti drugega kot list; nekakšno drevo bomo s postopkom iz prejšnjega odstavka sicer še vseeno dobili, vendar ostale povezave vhodnega grafa (tiste, ki niso med staršem in otrokom v rekonstruiranem drevesu) ne bodo tvorile cikla listov. Če bi bil naš vhodni graf brez vozlišč stopnje 2, bi se sicer dalo pri združevanju vozlišč vzdrževati podatke o tem, katera vozlišča so zagotovo listi, in bi lahko na koncu, ko nam od grafa ostane le še tetraeder, vzeli poljuben tak r, ki ni označen kot zagotovo list;[25] ker pa imamo lahko v grafu tudi vozlišča stopnje 2, je težko vnaprej reči, kateri r bi bil primeren.

Ena možnost je, da za r poskusimo vzeti vsako od štirih vozlišč našega tetraedra, pri katerem se je postopek krčenja ustavil (pravzaprav lahko tista vozlišča,

[24] Glej sliko 5. (a) Če je p sosed u-ja, ne pa w-ja: po razdružitvi postane u notranje vozlišče, torej bo povezava (u, v) veljavna le, če bo v otrok u-ja; toda potem bosta v in w oba otroka u-ja, zato je povezava med njima veljavna le, če sta oba lista; toda v ima stopnjo > 3, zato ne more biti list. — (b) Če je p sosed w-ja, ne pa u-ja: tu postane w notranje vozlišče, zato bo povezava med njim in v dopustna le, če bo v otrok w-ja; toda potem sta u in v oba otroka w-ja, torej je povezava med njima veljavna le, če sta oba lista; toda v ima stopnjo > 3, zato ne more biti list.

[25] Za podrobnosti gl. str. 337–8 v članku: D. Eppstein, "Simple recognition of Halin graphs and their generalizations", *J. of Graph Algorithms and Applications*, 20(2):323–46 (2016).

ki so nastala z združitvami tipov 1 in 2, brez škode preskočimo, saj smo videli, da predstavljajo liste okleščenega drevesa in mora torej obstajati še neko drugo vozlišče, ki je notranje vozlišče okleščenega drevesa). Če smo si izbrali napačen r, bomo naleteli na eno od naslednjih težav: (a) pri razveljavitvi krčitve tipa 2, ko razklenemo z nazaj na u in w, se lahko izkaže, da vozlišče p, ki je bilo prej starš z-ja in bi moralo zdaj postati starš u-ja in w-ja, v resnici ni sosed obeh teh vozlišč, ampak samo enega od njiju. (b) Lahko se zgodi, da kakšna povezava ne povezuje niti otroka s staršem (kot bi pričakovali od povezav v drevesu) niti dveh listov (kot bi pričakovali od povezav v ciklu listov). (c) Lahko se zgodi, da ima kak list stopnjo 2 namesto 3. — Če pa pridemo z razveljavitvami do konca, ne da bi naleteli na kakšno od teh težav, lahko zaključimo, da imamo pred seboj veljavno rekonstrukcijo. Iz postopka, s katerim smo pri razveljavljanju združitev vzpostavljali povezave med starši in otroki, vidimo, da povezave med starši in otroki res tvorijo drevo (povezan acikličen graf), ostale povezave pa cikel vseh listov.

Druga možnost je, da si poskusimo r izbrati vnaprej, še preden sploh začnemo združevati vozlišča. Med združevanjem se potem držimo načela, da r-ja nikoli ne združimo s kakšnim drugim vozliščem; tako nam bo ostal tudi v tetraedru na koncu in bomo od tam lahko pognali postopek razdruževanja. (1) Če je v vhodnem grafu kakšno vozlišče stopnje > 3, zanj že vemo, da je gotovo notranje in ne list, zato ga lahko uporabimo za r in se nam bo postopek gotovo izšel. (2) Sicer, če imamo kakšno vozlišče stopnje 2: morda je celo več takih zaporednih vozlišč v verigi, toda na krajiščih te verige morata biti vozlišči z višjo stopnjo (sicer bi bil celoten graf le en sam dolg cikel). Tidve vozlišči sta primerna kandidata za r; če se postopek ne izide pri enem, se bo gotovo pri drugem, kajti vsaj eno od njiju gotovo ni list drevesa (sicer bi imeli drevo, sestavljeno iz dveh listov, povezanih z verigo, naloga pa zagotavlja, da so listi vsaj trije). (3) Če pa so vsa vozlišča v vhodnem grafu stopnje 3, lahko razmišljamo takole: vzemimo poljubno vozlišče u in njegove tri sosede; to naj bodo naši kandidati za r. Če je u notranje vozlišče, bo že sámo primeren kandidat; če pa je u list, je eden od njegovih sosedov gotovo notranje vozlišče in se bo postopek izšel pri njem.

Urejanje poddreves. Rekonstruirali smo torej drevo s korenom in za vsako vozlišče poznamo njegovega starša. Ne bi bilo tudi težko za vsako vozlišče pripraviti seznama njegovih otrok; toda pri tem je treba nekaj pazljivosti: za štetje barvanj bo koristno imeti sezname otrok urejene glede na vrstni red, ki ga določa cikel listov. Zaradi povezav v ciklu namreč barvanje enega poddrevesa vpliva na barvanje sosednjih poddreves (prejšnjega in naslednjega na ciklu).

Najprej za vsak list določimo njegovega predhodnika in naslednika na ciklu listov:

naj bo u_0 poljuben list; $u := u_0$; $PrejList[u] :=$ NIL;
ponavljaj:
  naj bo v tisti sosed u-ja, ki ni niti u-jev starš niti $PrejList[u]$;
  $NaslList[u] := v$; $PrejList[v] := u$; $u := v$;
dokler je $u \neq u_0$;

Pripravimo si obratni vrstni red (*postorder*) vozlišč drevesa, torej tak vrstni red, v katerem se pojavi vsako vozlišče šele za svojimi otroki:

$L :=$ prazen seznam; dodaj v L koren drevesa;

```
za vsako vozlišče u iz L (po vrsti):
  dodaj na konec seznama L vse u-jeve otroke;
obrni seznam L;
```

Zdaj lahko obdelamo vozlišča v tem vrstnem redu in pri vsakem primerno uredimo otroke. Rezultate tega urejanja bomo predstavili z nekaj tabelami: prvi otrok vozlišča u bo $Prvi[u]$, naslednji sorojenec (torej u-jev neposredni naslednik v urejenem zaporedju otrok u-jevega starša) vozlišča u pa bo $Nasl[u]$; poleg tega bomo za vsak u izračunali še prvi in zadnji list v u-jevem poddrevesu.

```
    naj bo t[0..n − 1] pomožna tabela;
    za vsako vozlišče u iz L:
      if je u list:
        PrviList[u] := u; ZadnjiList[u] := u;
        Prvi[u] := NIL; Nasl[u] := NIL; continue;
1     za vsakega u-jevega otroka v:
        t[PrejList[PrviList[v]]] := NIL; t[NaslList[ZadnjiList[v]]] := NIL;
2     za vsakega u-jevega otroka v:
        t[PrviList[v]] := v; t[ZadnjiList[v]] := v;
      Prvi[u] := NIL; PrviList[u] := NIL; ZadnjiList[u] := NIL;
3     za vsakega u-jevega otroka v:
        p := t[PrejList[PrviList[v]];
        if p = NIL then Prvi[u] := v, PrviList[u] := PrviList[v];
        Nasl[v] := t[NaslList[ZadnjiList[v]];
        if Nasl[v] = NIL then ZadnjiList[u] := ZadnjiList[v];
4     if Prvi[u] = NIL:
        v := poljuben otrok u-ja;
        Prvi[u] := v; PrviList[u] := PrviList[v];
        w := t[PrejList[PrviList[v]];
        Nasl[w] := NIL; ZadnjiList[u] := ZadnjiList[w];
```

Ko se ukvarjamo z notranjim vozliščem u, v tabeli t pri prvem in zadnjem listu vsakega u-jevega otroka v označimo, da pripadata temu v (zanka 2); poleg tega pa še poskrbimo (zanka 1), da bosta predhodnik prvega lista in naslednik zadnjega lista gotovo imela (v tabeli t) neveljavno vrednost NIL ne glede na to, kaj je morda tam v tej tabeli ostalo od prej, ko smo obdelovali kak drug u. S pomočjo tabele t lahko zdaj za vsakega u-jevega otroka v zlahka določimo, kdo je v-jev neposredni predhodnik in kdo neposredni naslednik v urejenem seznamu u-jevih otrok (zanka 3); naslednika zapišemo v $Nasl[v]$, glede predhodnika pa, če ga v nima, to pomeni, da je v prvi u-jev otrok in ga zapišemo v $Prvi[u]$. Lahko se zgodi, da u-jevo poddrevo obsega prav vse liste in v tem primeru je zanka 3 vse u-jeve otroke povezala v cikel (prek tabele $Nasl$), nobenega od njih pa ni razglasila za prvega. V tem primeru vzamemo (stavek 4) poljubnega u-jevega otroka v in prekinemo cikel med v in njegovim predhodnikom w; tako postane v prvi u-jev otrok, w pa zadnji.

Štetje barvanj. Zdaj so poddrevesa pri vsakem vozlišču primerno urejena (glede na vrstni red listov na ciklu) in lahko barvanja štejemo z dinamičnim programiranjem. Naj bo $T(u)$ poddrevo, ki ga tvorijo u in vsi njegovi potomci; in naj bo $T'(u)$ unija poddreves, ki se začnejo pri u in vseh njegovih desnih sorojencih: $T'(u) =$

$T(u) \cup T'(Nasl[u])$. Naravna ideja je, da si kot podprobleme zastavimo za vsako vozlišče u vprašanje, na koliko načinov je mogoče pobarvati $T(u)$; to poddrevo je sestavljeno iz vozlišča u in iz poddreves vseh u-jevih otrok; toda teh poddreves ne moremo barvati neodvisno enega od drugega: zadnji list posameznega poddrevesa je povezan s prvim listom naslednjega, zato ne smeta biti iste barve; poleg tega pa je koren poddrevesa (torej u-jev otrok) povezan z u-jem in torej tudi onadva ne smeta biti iste barve. V opis podproblema za u-jevo poddrevo moramo torej vključiti barve prvega in zadnjega lista v tem poddrevesu in u-ja samega, kajti ta tri vozlišča imajo povezave z vozlišči zunaj poddrevesa in bodo zato vplivala na barvanje drugih delov grafa.

Naj bo torej $f_u(b_u, b_\ell, b_d)$ število takih barvanj $T(u)$, v katerih dobi najbolj levi list tega poddrevesa barvo b_ℓ, najbolj desni barvo b_d, vozlišče u pa barvo b_u. Da bomo lahko to postopoma računali iz rezultatov za poddrevesa u-jevih otrok, pa je koristno definirati še $g_u(b_p, b_\ell, b_d)$ kot število takih barvanj $T'(u)$, v katerih dobi najbolj levi list v $T'(u)$ barvo b_ℓ, najbolj desni list barvo b_d, vozlišče u in njegovi desni sorojenci pa ne dobijo barve b_p (ker se domneva, da bo take barve njihov starš — recimo mu p).

Razmislimo najprej o tem, kako računati f_u. Če je u list, je stvar preprosta: list u je hkrati najbolj levi in najbolj desni list poddrevesa $T(u)$, zato se barve b_u, b_ℓ in b_d vse nanašajo na isto vozlišče; če so vse enake, je možno eno samo barvanje, v katerem dobi u barvo b_u; če pa niso vse enake, nam postavljajo nemogoče pogoje in je število možnih barvanj 0:

$$f_u(b_u, b_\ell, b_d) = [\![b_u = b_\ell = b_d]\!].$$

Zapis $[\![\cdot]\!]$ pomeni vrednost 1, če je pogoj v oklepajih izpolnjen, sicer pa vrednost 0.

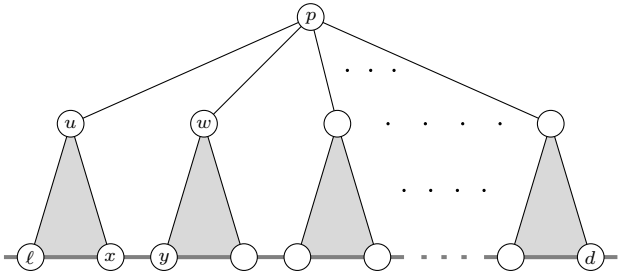
Če u ni list, naj bo v njegov prvi otrok; potem dobimo veljavno barvanje tako, da u pobarvamo z barvo b_u, nato pa pobarvamo poddrevesa vseh njegovih otrok od v-ja naprej, pri tem pa pazimo, da noben otrok ne dobi barve u. Torej je

$$f_u(b_u, b_\ell, b_d) = g_v(b_u, b_\ell, b_d).$$

Razmislimo zdaj o računanju g_u. Če u nima desnega sorojenca — če je torej u zadnji otrok svojega starša ali pa je morda celo koren drevesa in starša sploh nima — je $T'(u) = T(u)$ in si lahko za izračun g_u pomagamo s f_u; paziti moramo le na to, da imamo pri g_u predpisano, kakšne barve u ne sme biti (namreč barve b_p), pri f_p pa, kakšne barve mora biti (namreč barve b_u). V poštev za b_u pridejo torej vse barve razen b_p:

$$g_u(b_p, b_\ell, b_d) = \textstyle\sum_{b_u} [\![b_u \neq b_p]\!]\, f_u(b_u, b_\ell, b_d). \tag{1}$$

Ostane še možnost, da u ima desnega sorojenca, recimo w. Tedaj je $T'(u)$ sestavljen iz $T(u)$ in $T'(w)$; tadva dela pa sta povezana s povezavo med najbolj desnim listom v $T(u)$ in najbolj levim listom v $T'(w)$. Recimo tema listoma x in y; ker sta povezana, morata biti različnih barv (slika 7). Da primerno pobarvamo $T'(u)$, si moramo torej izbrati b_u, b_x in b_y in potem lahko pobarvamo $T(u)$ na $f_u(b_u, b_\ell, b_x)$ načinov ter $T'(w)$ na $g_w(b_p, b_y, b_d)$ načinov. Pri tem moramo seveda paziti, da je $b_u \neq b_p$ (saj je to eden od pogojev, da barvanje celotnega $T'(u)$ pride v poštev za



Slika 7. Siva poddrevesa skupaj tvorijo $T'(u)$.

$g_u(b_p, b_\ell, b_d)$) in $b_x \neq b_y$ (ker morata biti lista x in y različnih barv). Dobili smo torej:

$$g_u(b_p, b_\ell, b_d) = \sum_{b_u, b_x, b_y} [\![b_u \neq b_p]\!][\![b_x \neq b_y]\!]\, f_u(b_u, b_\ell, b_x) g_w(b_p, b_y, b_d). \qquad (2)$$

Paziti pa moramo še na nekaj robnih primerov. Če je u list, je s tem tudi najbolj levi in desni list v svojem poddrevesu, tako da se barve b_u, b_ℓ in b_x vse nanašajo na vozlišče u. Če niso vse te barve enake, veljavnega barvanja sploh ne bo mogoče dobiti; če pa so vse enake, je mogoče $T(u)$ pobarvati na natanko en način. Gornja formula se tedaj poenostavi v:

$$g_u(b_p, b_\ell, b_d) = [\![b_\ell \neq b_p]\!] \sum_{b_y} [\![b_\ell \neq b_y]\!]\, g_w(b_p, b_y, b_d). \qquad (3)$$

Drugi robni primer pri funkciji g_u pa nastopi, če $T'(u)$ obsega vse liste drevesa, kar pomeni, da sta si najbolj levi list (barve b_ℓ) in najbolj desni list (barve b_d) soseda na ciklu listov in ne smeta biti iste barve. Takrat moramo torej v primerih, ko je $b_\ell = b_d$, postaviti $g_u(b_p, b_\ell, b_d)$ na 0 ne glede na to, kaj pravijo prej omenjene formule. Podoben pomislek načeloma velja tudi za funkcijo f_u, vendar le-to tako ali tako računamo iz g_v za u-jevega prvega otroka v; in takrat, če $T(u)$ obsega vse liste, jih tudi $T'(v)$; in ker bomo za $b_\ell = b_d$ dobili $g_v(b_u, b_\ell, b_d) = 0$, bomo dobili brez posebnega dodatnega truda takrat tudi $f_u(b_u, b_\ell, b_d) = 0$.

Zdaj imamo torej načeloma vse, kar potrebujemo za izračun funkcij f_u in g_u. Računamo jih lahko od listov navzgor po drevesu, v obratnem vrstnem redu (*post-order*); ko pridemo do korena r in izračunamo f_r, je potem rezultat, po katerem sprašuje naloga, enak $\sum_{b_r, b_\ell, b_d} f_r(b_r, b_\ell, b_d)$.

Paziti pa moramo, da se tega izračuna ne lotimo preveč naivno. Imamo načeloma $O(n)$ funkcij in pri vsaki moramo izračunati $O(k^3)$ vrednosti, saj ima vsaka funkcija tri argumente, ki so barve in je torej za vsako barvo k možnosti. Poleg tega moramo pri g_u v splošnem (če u ni list in ima desnega sorojenca; to pa se lahko zgodi pri $O(n)$ vozliščih) računati vsoto po b_u, b_x in b_y, torej $O(k^3)$ členov. Če bo naša rešitev porabila $O(n \cdot k^6)$ časa za izračun rezultatov, bo vsekakor presegla časovno omejitev.[26]

[26] Skrbeti bi nas utegnila tudi poraba $O(n \cdot k^3)$ pomnilnika za shranjevanje vseh rezultatov; toda to lahko zmanjšamo na $O(k^3 \log n)$, če pazljivo izberemo vrstni red računanja in sproti pozabljamo rezultate za vozlišča, ki jih ne bomo več potrebovali.

Ena izboljšava, ki ne škoduje, vendar tudi še ne zadošča, temelji na opažanju, da v gornjih formulah pogosto računamo vsote po vseh barvah razen eni. Koristno je zato najprej izračunati vsoto po vseh barvah, nato pa od nje le odšteti barvo, ki je nočemo. Če izračunamo

$$f_u(\bullet, b_\ell, b_d) := \textstyle\sum_{b_u} f_u(b_u, b_\ell, b_d), \tag{4}$$

se nam formula (1) poenostavi v

$$g_u(b_p, b_\ell, b_d) = f_u(\bullet, b_\ell, b_d) - f_u(b_p, b_\ell, b_d).$$

Pri takšnem u (ki nima desnega sorojenca w) porabimo potem $O(k^3)$ časa za izračun vseh vsot $f_u(\bullet, b_\ell, b_d)$ in nato $O(k^3)$ za izračun vseh vrednosti $g_u(b_p, b_\ell, b_d)$; če pa bi vsako od slednjih računali z vsoto po b_u kot v prvotni formuli, bi porabili skupaj $O(k^4)$ časa. Enak prijem lahko uporabimo tudi v splošnem primeru, kjer u ima desnega sorojenca; definirajmo še

$$g_u(b_p, \bullet, b_d) := \textstyle\sum_{b_y} g_u(b_p, b_y, b_d) \tag{5}$$

in formula (2) se poenostavi v

$$g_u(b_p, b_\ell, b_d) = \textstyle\sum_{b_x} \big(f_u(\bullet, b_\ell, b_x) - f_u(b_p, b_\ell, b_x)\big)\big(g_w(b_p, \bullet, b_d) - g_w(b_p, b_x, b_d)\big), \tag{6}$$

če pa je u list, se formula (3) poenostavi v

$$g_u(b_p, b_\ell, b_d) = [\![b_\ell \neq b_p]\!]\big(g_w(b_p, \bullet, b_d) - g_w(b_p, b_\ell, b_d)\big).$$

Pri takšnem u torej porabimo $O(k^3)$ časa za izračun vseh $f_u(\bullet, b_\ell, b_d)$ in $g_u(b_p, \bullet, b_d)$, nato pa $O(k^4)$ za izračun vseh $g_u(b_p, b_\ell, b_d)$, ker imamo pri vsaki od slednjih le še $O(k)$ dela zaradi vsote po b_x (to velja za primere, ko je u notranje vozlišče; če je u list, pa nimamo niti vsote po b_x in imamo le $O(1)$ dela za vsako $g_u(b_p, b_\ell, b_d)$, torej $O(k^3)$ za vse skupaj).

Časovno zahtevnost naše rešitve smo tako zmanjšali z $O(n \cdot k^6)$ na $O(n \cdot k^4)$, vendar je to za našo nalogo še vedno preveč. Naslednja izboljšava pa temelji na opažanju, da če imamo neko veljavno barvanje (torej takšno, v kateri nima nobena povezava dveh krajišč enake barve) in v njem spremenimo barve z neko injektivno preslikavo π — torej takó, da bo vsako vozlišče, ki je bilo prej barve b, po novem barve $\pi(b)$ — bo barvanje ostalo veljavno: povezava, ki je imela prej krajišči dveh različnih barv b in b', ima zdaj krajišči barv $\pi(b)$ in $\pi(b')$, ki sta tudi različni (saj je π injektivna). To pa tudi pomeni, da je $f_u(b_u, b_\ell, b_d) = f_u(\pi(b_u), \pi(b_\ell), \pi(b_d))$ in podobno za g_u. Na primer: če so barve predstavljene s celimi števili od 1 do k in če je k dovolj velik, je $f_u(1,2,3) = f_u(1,3,2) = f_u(3,2,1) = f_u(17,5,23)$ in podobno. V splošnem, če so b_u, b_ℓ in b_d tri različna števila, lahko vzamemo táko preslikavo π, ki preslika $b_u \mapsto 1$, $b_\ell \mapsto 2$ in $b_d \mapsto 3$, in vidimo, da so vrednosti $f_u(b_u, b_\ell, b_d)$ v vseh teh primerih enake $f_u(1,2,3)$. Podobno, če je $b_u = b_\ell \neq b_d$, lahko vzamemo táko π, ki preslika b_u (in b_ℓ) v 1, barvo b_d pa v 2 in vidimo, da so $f_u(b_u, b_\ell, b_d)$ v vseh teh primerih enake $f_u(1,1,2)$. Če tako nadaljujemo, vidimo, da je vrednost funkcije f_u odvisna le od tega, kateri izmed njenih treh argumentov so enaki, kateri pa različni, in da ima torej ta funkcija le pet različnih vrednosti: $f_u(1,1,1)$, $f_u(1,1,2)$,

$f_u(1,2,1)$, $f_u(2,1,1)$ in $f_u(1,2,3)$. Enak razmislek velja tudi za g_u. Pri robnih (marginalnih) vsotah, ki smo jih definirali malo prej, pa sta možni vrednosti celo le po dve: $f_u(\bullet,1,1)$ in $f_u(\bullet,1,2)$, podobno pa tudi $g_u(1,\bullet,1)$ in $g_u(1,\bullet,2)$. In ne le, da moramo izračunati manj vrednosti vseh teh funkcij; tudi pri vsotah, s katerimi so te vrednosti definirane, je veliko členov enakih in ni treba seštevati vsakega posebej. Tako na primer iz formule (4) dobimo:

$$\begin{aligned} f_u(\bullet,1,1) &= \textstyle\sum_{b_u} f_u(b_u,1,1) = f_u(1,1,1) + \sum_{b_u>1} f_u(b_u,1,1) \\ &= f_u(1,1,1) + (k-1) f_u(2,1,1), \end{aligned}$$

kjer smo v zadnjem koraku upoštevali, da ima vsota $k-1$ členov in pri vseh je $b_u \neq 1$, zato so vsi ti členi enaki $f_u(2,1,1)$. Podobno dobimo tudi:

$$\begin{aligned} f_u(\bullet,1,2) &= \textstyle\sum_{b_u} f_u(b_u,1,2) = f_u(1,1,2) + f_u(2,1,2) + (k-2) f_u(3,1,2) \\ &= f_u(1,1,2) + f_u(1,2,1) + (k-2) f_u(1,2,3). \end{aligned}$$

Podobno iz formule (5) dobimo:

$$g_u(1,\bullet,1) = g_u(1,1,1) + (k-1) g_u(1,2,1) \text{ in}$$
$$g_u(1,\bullet,2) = g_u(1,1,2) + g_u(2,1,1) + (k-2) g_u(1,2,3).$$

Tudi vsota po b_x v formuli (6) se nam poenostavi: ker bodo zdaj argumenti funkcije g_u, torej b_p, b_ℓ in b_d, vsi z območja $\{1,2,3\}$, bodo imeli členi vsote za $b_u \geq 4$ vsi enako vrednost, kajti vsak tak b_x je različen od barv b_p, b_ℓ in b_d, na vrednosti naših funkcij pa tako ali tako vpliva le to, kateri argumenti so si enaki in kateri različni. Od vsote po b_x je torej dovolj izračunati le prve štiri člene, četrtega pa pomnožiti s $k-3$. Tako torej vidimo, da moramo izračunati pri vsakem u le konstantno mnogo vrednosti funkcij f_u in g_u ter da imamo z vsako od njih le konstantno mnogo dela. Poraba pomnilnika in poraba časa se s tem obe zmanjšata na $O(n)$. Ker smo poleg tega pred tem porabili $O(n)$ časa in prostora tudi za rekonstrukcijo drevesa in urejanje poddreves, imamo torej rešitev z linearno časovno in prostorsko zahtevnostjo, $O(n)$, od k-ja pa je sploh neodvisna.

G. Gremo na Luno

§1. Uvod. Opomba glede notacije: vektorje bomo pisali brez puščic nad njimi; če imamo na primer točki $U(x_U, y_U)$ in $V(x_V, y_V)$, je potem $U = (x_U, v_U)$ in $UV = V - U = (x_V - x_U, y_V - y_U)$. Dolžino vektorja označimo z $|U|$ oz. $|UV|$, njegov polarni kót pa s ϕ_U oz. ϕ_{UV}; tako je npr. $\phi_U = \operatorname{atan2}(y_U, x_U)$.

Iščemo najkrajšo táko pot od A do B, ki bo imela s krogom skupno vsaj eno točko, recimo T. Pot gre torej od A do T in nato od T do B. Če prvi del poti, od A do T, ni ravno daljica AT, bi se dalo pot skrajšati, če bi ta začetni del zamenjali z daljico AT; podobno razmišljamo pri drugem delu poti, od T do B. Najkrajša pot je torej sestavljena iz daljic AT in TB, vprašanje je le še to, katero točko kroga moramo vzeti za T.

Obravnavajmo najprej nekaj robnih primerov. — Če je $r = 0$, se krog izrodi v točko in edina možna izbira za T je središče kroga C; takrat moramo torej vrniti $|AC| + |CB|$. — Če leži vsaj ena od točk znotraj (ali na robu) kroga, ima že daljica AB skupno neko točko s krogom, poleg tega pa je to tudi najkrajša možna pot

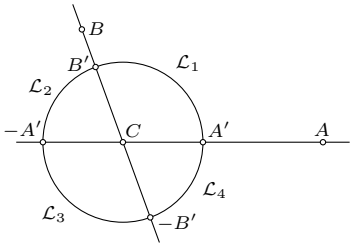
med A in B sploh, zato moramo takrat vrniti $|AB|$. — Če ležita A in B obe zunaj kroga in so točke A, B in C kolinearne, je točka, ki je na krogu najbližja točki A, najbližja tudi točki B, zato moramo za T vzeti njo; to je $T = (A - C)/|A - C|$. — Če ležita A in B obe zunaj kroga, vendar se daljica AB seka s krogom (ali se ga vsaj dotakne), je rešitev spet $|AB|$. To, ali se AB seka s krogom, lahko preverimo tako, da izračunamo pravokotno projekcijo središča C na premico AB. Recimo, da je to točka $U = A + \lambda AB$, pri čemer mora biti $UC \perp AB$, torej mora biti skalarni produkt $\langle UC, AB\rangle = 0$, torej $\langle AC - \lambda AB, AB\rangle = 0$, torej $\lambda = \langle AC, AB\rangle / \langle AB, AB\rangle$. Središču C najbližja točka na daljici AB je potem A, če je $\lambda \leq 0$; B, če je $\lambda \geq 1$; sicer pa U. Če je ta najbližja točka oddaljena od C za $> r$, se daljica ne seka s krogom (niti se ga ne dotika), sicer pa se.

V nadaljevanju torej predpostavimo, da je $r > 0$ in da leži daljica AB v celoti zunaj kroga. Premaknimo koordinatno izhodišče v točko C in delimo vse koordinate z r, tako da bomo v bodoče predpostavili, da je krog enotski, s središčem $(0, 0)$ in polmerom 1 (paziti moramo le še na to, da bomo na koncu pred izpisom pomnožili rezultat z r in ga tako pretvorili nazaj v prvotne enote). Za potrebe našega razmisleka (pri implementaciji na računalniku to ne bo potrebno) nato še zasukajmo koordinatni osi tako, da bo ležala A na pozitivni x-osi (torej da bo $\phi_A = 0$, $x_A > 0$, $y_A = 0$), in po potrebi pomnožimo vse y-koordinate z -1, tako da bo B ležala nad x-osjo (torej bo $0 < \phi_B < 180^\circ$, $y_B > 0$).

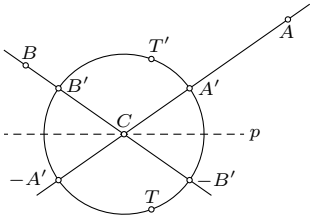
Mislimo si najkrajšo pot od A do B prek neke točke T, ki pripada enotskemu krogu. Videli smo že, da je ta pot sestavljena iz daljic AT in TB. Ali je mogoče, da T leži v notranjosti kroga in ne na njegovem robu? Pa recimo, da je tako; naj bo tedaj U presečišče daljice AT z robom kroga (torej z enotsko krožnico). Daljici AU in UB torej tudi tvorita pot, ki ustreza zahtevam naše naloge. Dolžina naše prvotne poti prek T je $|AT| + |TB| = |AU| + |UT| + |TB| > |AU| + |UB|$, pri čemer smo v zadnjem koraku uporabili trikotniško neenakost za trikotnik $\triangle UTB$. Pot od A do B skozi U je torej krajša kot skozi T, pri tem pa ima še vedno neko skupno točko s krogom (namreč U), kar je protislovje. Tako torej vidimo, da se smemo pri iskanju najkrajše poti omejiti na take točke T, ki ležijo na robu kroga (torej na enotski krožnici) in ne v njegovi notranjosti.

Položaj točke T na naši enotski krožnici lahko opišemo z njenim polarnim kotom $\tau = \phi_T$, tako da je $T = (\cos\tau, \sin\tau)$. Za poljubno točko U, ki ne leži na krožnici, definirajmo $f_U(\tau)$ kot razdaljo med U in točko T (definirano s polarnim kotom τ); ta funkcija ima svoj minimum pri $\tau = \phi_U$ (takrat je T najbližja točki U), od tam počasi narašča (ker se s povečevanjem τ točka T oddaljuje od U) do svojega maksimuma pri $\tau = \phi_U + 180^\circ$ in odtlej pada (ker se T začne spet približevati U-ju po drugi strani krožnice) do $\tau = \phi_U + 360^\circ$, ko spet doseže svoj minimum.

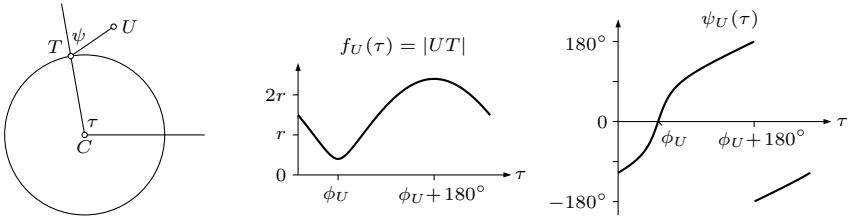
Naša naloga je minimizirati vsoto $f(\tau) := f_A(\tau) + f_B(\tau)$ — to je dolžina poti od A do T (s polarnim kotom τ) in od tam naprej do B. Točko, ki je na krožnici najbližja točki A, označimo z $A' := A/|A|$; in podobno $B' := B/|B|$. Premici CA in CB nam razdelita enotsko krožnico na štiri loke (slika 1; spomnimo se, da je $\phi_A = 0$ in da je ϕ_B med 0 in 180°): ($\mathcal{L}_1$) na območju od A' do B' (to je pri $0 \leq \tau \leq \phi_B$) se, če povečujemo τ, oddaljujemo od A in približujemo B; ($\mathcal{L}_2$) na območju od B' do $-A'$ (pri $\phi_B \leq \tau \leq 180^\circ$) se oddaljujemo od A in od B; ($\mathcal{L}_3$) na območju od $-A'$ do $-B'$ (pri $180^\circ \leq \tau \leq \phi_B + 180^\circ$) se oddaljujemo od B in približujemo A; ($\mathcal{L}_4$) na



Slika 1.



Slika 2.

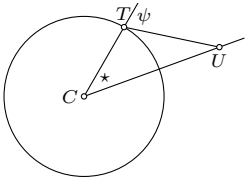


Slika 3.

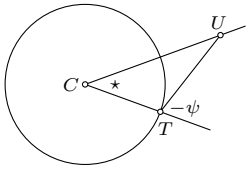
območju od $-B'$ do A' (pri $\phi_B + 180° \leq \tau \leq 360°$) pa se približujemo A in B.

Na loku $\mathcal{L}_2$, kjer se oddaljujemo od obeh, A in B, gotovo ne bomo našli minimuma funkcije f, saj lahko njeno vrednost vedno zmanjšamo, če τ malo zmanjšamo; podobno je na $\mathcal{L}_4$, kjer se približujemo obema točkama in lahko vrednost funkcije zmanjšamo, če τ malo povečamo. Kaj pa lok $\mathcal{L}_3$? Naj bo p premica, ki razpolavlja loka $\mathcal{L}_2$ in $\mathcal{L}_4$; sestavljata jo poltraka s polarnim kotom $90° + \phi_B/2$ in $270° + \phi_B/2$. Zasukajmo v mislih koordinatni sistem tako, da bo p vodoravna (slika 2). Nad njo zdaj ležita točki A in B ter celoten lok $\mathcal{L}_1$; lok $\mathcal{L}_3$ pa leži pod premico p. Recimo, da bi neka točka T na loku $\mathcal{L}_3$ dala najboljšo rešitev; preslikajmo jo čez premico p in dobljeni točki recimo T'. Razdalja od A do T' v vodoravni smeri je enaka kot od A do T; v navpični smeri pa je razdalja od A do T' manjša kot od A do T, (namreč za dvakratnik razdalje od premice p do tiste izmed točk A in T, ki je tej premici bližja); tako je torej $|AT'| < |AT|$. Podobno je tudi $|BT'| < |BT|$, zato točka T' vsekakor pripelje do boljše rešitve kot točka T. Tako vidimo, da tudi na loku $\mathcal{L}_3$ ne bomo našli najkrajše poti; smemo se omejiti na točke T z loka $\mathcal{L}_1$, to je med poltrakoma CA in CB. (Če pa bi nas zanimala na krožnici taka točka T, pri kateri bi bila $|AT| + |BT|$ najdaljša, bi nam analogen razmislek pokazal, da jo bomo gotovo našli na loku $\mathcal{L}_3$.)

Naj bo zdaj (za poljubno točko U zunaj kroga) $\psi_U(\tau) := \tau - \phi_{TU}$ kót v točki T med smerjo TU in normalo na krožnico v točki T (smer te normale je seveda ravno τ). Kót ψ_U merimo vedno v pozitivni smeri (torej nasproti smeri urinega kazalca) od TU k normali, tako da je lahko tudi negativen. Razmislimo, kako se obnaša ψ_U v odvisnosti od τ (slika 3). Če začnemo pri $\tau = \phi_U$ (ko je T ravno projekcija U na krožnico) in počasi povečujemo τ (torej se premikamo po krožnici v smeri, nasprotni urinim kazalcem), se ψ_U počasi povečuje od 0 prek 90° (ki jo doseže takrat, ko gre



Slika 4 ($\star = \tau - \phi_U$).



Slika 5 ($\star = \phi_U - \tau$).

tangenta na krožnico v točki T ravno skozi točko U) do 180° (pri $\tau = \phi_U + 180^\circ$, torej ravno na nasprotni strani krožnice od U), nato preskoči na -180° in se počasi povečuje prek -90° (ko gre tangenta na krožnico v točki T skozi U) nazaj do 0° (pri $\tau = \phi_U$). Absolutna vrednost $|\psi_U(\tau)|$ nam meri „vpadni kót“, pod katerim se žarek iz točke U v smeri proti točki T zaleti v krožnico (res pa je, da pri $|\psi_U| > 90^\circ$ ta interpretacija nima veliko smisla, saj se takrat tak žarek zaleti v krožnico že prej kot v točki T, slednjo pa doseže z notranje strani krožnice in ne z zunanje).

V nadaljevanju se bomo prepričali, da je omenjeni vpadni kót v tesni zvezi z odvodom funkcije f_U, nato pa bomo tudi videli, da v ekstremih funkcije f velja, da sta vpadna kota glede na točki A in B enaka — koristna lastnost, ki nam bo prišla kasneje še večkrat prav.

Funkcijo $f_U(\tau)$, torej razdaljo med U in T, lahko računamo kot

$$f_U(\tau) = |UT| = \sqrt{(x_U - \cos\tau)^2 + (y_U - \sin\tau)^2};$$

hitro lahko vidimo, da je njen odvod

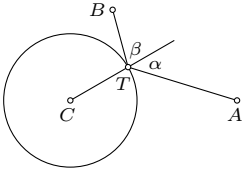
$$\begin{aligned} f'_U(\tau) &= (x_U \sin\tau - y_U \cos\tau)/f_U(\tau) \\ &= |CU| \sin(\tau - \phi_U)/|UT|, \end{aligned} \tag{1}$$

kjer ϕ_U v zadnji vrstici pomeni polarni kót točke U, njena oddaljenost od koordinatnega izhodišča pa je $|CU|$.

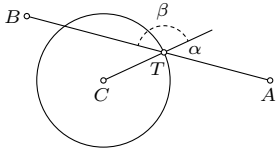
Oglejmo si trikotnik CUT. Vrednost $\psi_U(\tau)$ pišimo krajše kar ψ. Na sliki 4 imamo primer, ko leži T levo od U (gledano iz točke C), zato je $\psi > 0$. Kót pri oglišču C je $\tau - \phi_U$, stranica nasproti tega oglišča pa ima dolžino $|UT|$; podobno je kót pri oglišču T enak $180^\circ - \psi$, stranica nasproti tega oglišča pa je dolga $|CU|$. Spomnimo se na sinusni izrek, ki pravi, da je razmerje med sinusom kota in dolžino njemu nasprotne stranice enako v vseh ogliščih trikotnika; v našem primeru to pomeni, da je $\sin(\tau - \phi_U) : |UT| = \sin(180^\circ - \psi) : |CU|$. Uporabimo to v prej dobljenem izrazu za $f'_U(\tau)$, pa dobimo: $f'_U(\tau) = \sin(180^\circ - \psi) = \sin\psi$.

Podobno je v primeru na sliki 5, ko leži T desno od U in je zato $\psi < 0$. Kót pri oglišču C je $\phi_U - \tau$, pri oglišču T pa $180^\circ + \psi$. Sinusni izrek nam dá $\sin(\phi_U - \tau) : |UT| = \sin(180^\circ + \psi) : |CU|$, torej (če pomnožimo obe strani z -1) $\sin(\tau - \phi_U) : |UT| = \sin\psi : |CU|$, kar nam spet dá $f'_U(\tau) = \sin\psi$.

Funkcija $f(\tau) = f_A(\tau) + f_B(\tau)$, ki jo hočemo pri tej nalogi minimizirati, ima odvod $f'(\tau) = f'_A(\tau) + f'_B(\tau)$; in kot smo pravkar videli, je to naprej enako $\sin\psi_A(\tau) + \sin\psi_B(\tau)$. Da bo manj pisanja, vpeljimo $\alpha := \psi_A(\tau)$ in $\beta = -\psi_B(\tau)$; potem je $f'(\tau) = \sin\alpha - \sin\beta$. Na loku $\mathcal{L}_1$, torej če je $\phi_A \le \tau \le \phi_B$ — in to območje nas najbolj zanima, saj že vemo, da bomo minimum funkcije f našli prav tam — sta α in β oba nenegativna in imata tudi lepo geometrijsko interpretacijo (slika 6): če si



Slika 6.



Slika 7.

predstavljamo žarek, ki gre iz A v T, se tam odbije od krožnice in nadaljuje pot v B, je α njegov vpadni kot (kot med TA in normalo na krožnico v točki T), β pa njegov odbojni kot (kot med TB in normalo na krožnico v točki T).

V točkah, kjer je $f'(\tau) = 0$, ima funkcija f (lokalni) ekstrem. Minimum funkcije f lahko poiščemo tako, da pregledamo vse njene lokalne ekstreme, izračunamo vrednost f v njih in vrnemo najmanjšo od teh vrednosti. Pogoj $f'(\tau) = 0$ pa nam dá $\sin\alpha = \sin\beta$. Kota α in β sta vedno z območja $(-180^\circ, 180^\circ]$, funkcija sin pa na tem območju doseže vsako svojo vrednost največ dvakrat: $\sin\vartheta = \sin(180^\circ - \vartheta)$. Tako je torej $\sin\alpha = \sin\beta$ lahko res le, če je bodisi $\alpha = \beta$ bodisi $\alpha = 180^\circ - \beta$. Toda slednje v našem primeru pomeni (slika 7), da če se postavimo v točko T, tvorita kota med normalo na krožnico ter daljicama TA in TB skupaj ravno iztegnjen kot (180°), torej T v resnici leži na daljici AB; torej ta daljica seka krožnico (ali pa se je vsaj dotika), mi pa smo že pred časom predpostavili, da je ne (kajti primer, ko jo, smo obravnavali že na začetku med robnimi primeri). Tako je torej pogoj $\sin\alpha = \sin\beta$ lahko izpolnjen le tako, da velja $\alpha = \beta$ — vpadni kót mora biti enak odbojnemu.

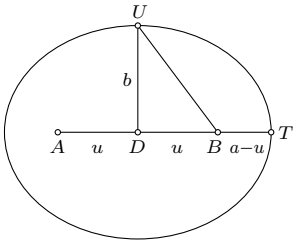
§2. Eksaktna rešitev. Za potrebe tega razdelka je koristno, če si naše točke v ravnini predstavljamo kot kompleksna števila: namesto točke (x, y) si mislimo kompleksno število $z = x + iy$. Osvežimo si še nekaj pojmov: *konjugirana vrednost* z-ja je $\bar{z} = x - iy$; *absolutna vrednost* z-ja je $|z| = \sqrt{x^2 + y^2}$; *argument* z-ja, $\arg z$, pa je polarni kot točke (x, y). Za vsak z z argumentom ϕ velja $z = |z| \cdot e^{i\phi}$, kjer je $e^{i\phi} = \cos\phi + i\sin\phi$. Iz tega sledi, da ima produkt $z \cdot w$ absolutno vrednost $|z| \cdot |w|$ in argument $\arg z + \arg w$; količnik z/w pa ima absolutno vrednost $|z|/|w|$ in argument $\arg z - \arg w$. Iz slednjega tudi sledi, da ima $1/z$ absolutno vrednost $1/|z|$ in argument $-\arg z$. Za konjugirane vrednosti velja $\overline{(z+w)} = \bar{z} + \bar{w}$ in podobno pri odštevanju, množenju in deljenju.

Pri naši nalogi lahko torej v mislih točke A, B, C, T zamenjamo s kompleksnimi števili $a = x_A + iy_A$, $b = x_B + iy_B$, $c = 0$ in $t = e^{i\tau}$. Namesto funkcije $\psi_U(\tau) = \tau - \phi_{TU}$ imamo zdaj (za $u = x_U + iy_U$) $\psi_u(\tau) = \tau - \arg(u - t) = \arg t - \arg(u - t) = \arg\frac{t}{u-t}$. Spomnimo se, da nas zanimajo ekstremi funkcije f, to pa so takšni τ, pri katerih je $\alpha = \beta$, torej $\psi_a(\tau) = -\psi_b(\tau)$, torej $\arg\frac{t}{a-t} + \arg\frac{t}{b-t} = 0$, torej $\arg\frac{t^2}{(a-t)(b-t)} = 0$. Pogoj za ekstrem je torej, da ima število $(a-t)(b-t)/t^2$ polarni kot 0, torej da je realno število, torej da je enako svoji konjugirani vrednosti:

$$(a-t)(b-t)/t^2 = \overline{\big((a-t)(b-t)/t^2\big)}.$$

To lahko predelamo v:

$$(ab - (a+b)t + t^2)\bar{t}^2 = (\bar{a}\bar{b} - (\bar{a}+\bar{b})\bar{t} + \bar{t}^2)t^2$$



Slika 8.

Primer elipse z goriščema A in B, središčem $D = (A + B)/2$, veliko polosjo a in malo polosjo b. Razdalja med goriščema je $2u$. Pri vsaki točki na elipsi je vsota razdalj do obeh gorišč enaka. Za T (na koncu velike polosi) je ta vsota $|TA|+|TB| = (a+u)+(a-u) = 2a$; za U (na koncu male polosi) je ta vsota $|UA| + |UB| = 2\sqrt{b^2+u^2}$. Ker morata biti obe vsoti enaki, dobimo $a = \sqrt{b^2+u^2}$ oz. $b = \sqrt{a^2-u^2}$.

Pomnožimo obe strani s t^2 in upoštevajmo, da je $|t| = 1$, zato $t \cdot \bar{t} = 1$:

$$ab - (a+b)t + t^2 = (\bar{a}\bar{b} - (\bar{a}+\bar{b})\bar{t} + \bar{t}^{\,2})t^4$$
$$ab - (a+b)t + t^2 = \bar{a}\bar{b}t^4 - (\bar{a}+\bar{b})t^3 + t^2$$
$$\bar{a}\bar{b}t^4 - (\bar{a}+\bar{b})t^3 + (a+b)t - ab = 0 \qquad (2)$$

Primerni t-ji so torej ničle polinoma četrte stopnje, ki smo ga dobili v zadnji vrstici gornje izpeljave.[27] Mogoče je sicer tudi, da nekatere ničle tega polinoma ne ležijo na enotski krožnici (torej da $|t| \neq 1$) in zato načeloma ne pridejo v poštev za ekstreme funkcije f; poleg tega je tudi mogoče, da bi neka ničla sicer morala ležati na krožnici, vendar zaradi zaokrožitvenih napak pri računanju dobimo t, ki leži nekoliko stran od nje. Zato je najbolje, če vsako dobljeno ničlo t projiciramo na enotsko krožnico — dobimo $t/|t|$ — in v tej točki izračunamo vrednost funkcije f; tako bomo med drugim dobili vse ekstreme funkcije f in med njimi tudi njen globalni minimum. Najmanjša med tako dobljenimi vrednostmi funkcije f je torej rezultat, po katerem sprašuje naša naloga.

Ničle polinoma četrte stopnje lahko načeloma izračunamo z eksplicitnimi formulami, ki sta jih že v 16. stoletju našla Ferrari in Cardano,[28] lahko pa uporabimo kak postopek za numerično iskanje ničel polinoma s kompleksnimi koeficienti, npr. CPOLY.[29] Za na tekmovanje ta rešitev ni najbolj primerna, ker je z implementacijo vsakega od teh dveh pristopov precej dela.

§3. Rešitev z napihovanjem elipse. Iščemo točko T na enotski krožnici, ki ima najmanjšo vsoto razdalj $|AT| + |BT|$. Množica točk T, pri kateri je vsota $|AT| + |BT| = 2a$, tvori elipso z goriščema A in B ter veliko polosjo a. Naj bo $u := |AB|/2$ polovica razdalje med goriščema; to je potem tudi najmanjša možna vrednost a (pri $a = u$ je elipsa izrojena v daljico AB). Malo polos b lahko potem izrazimo kot $b = \sqrt{a^2-u^2}$ (slika 8).

Naloga torej pravzaprav sprašuje po najmanjši taki elipsi z goriščema A in B, ki še ima kakšno skupno točko s krožnico s središčem C in polmerom r. Za potrebe našega razmisleka v tem razdelku postavimo koordinatno izhodišče v središče elipse (torej razpolovišče daljice AB) in zasukajmo koordinatni osi tako, da bosta imeli gorišči koordinate oblike $A(0,u)$ in $B(0,-u)$. Koordinate potem še vedno delimo z

[27] Za več o tej rešitvi gl. M. Fujimura, P. Hariri, M. Mocanu, M. Vuorinen, "The Ptolemy–Alhazen problem and spherical mirror reflection", *Computational Methods and Function Theory* 19(1):135–155 (March 2019).

[28] Gl. npr. Wikipedijo *s. v.* Quartic equation.

[29] M. A. Jenkins, J. F. Traub, "Algorithm 419: zeros of a complex polynomial", *Communications of the* ACM, 15(2):97–99 (February 1972); in popravek v: D. H. Withers, "Remark on Algorithm 419", *Communications of the* ACM, 17(3):157 (March 1974).

r, tako da v bodoče predpostavimo, da imamo krog s polmerom 1, vendar njegovo središče C zdaj ni nujno v koordinatnem izhodišču.

Vemo, da je pri $a = u$ naša elipsa izrojena v daljico AB in da le-ta nima nobene skupne točke z našim krogom (ker smo primere, ko jih ima, obravnavali posebej že na začetku); in po drugi strani vemo, da bi pri $a = (|AC| + |BC|)/2$ elipsa šla skozi točko C, torej bi gotovo imela neko skupno točko s krogom. Tako imamo torej en a, ki je gotovo premajhen, in enega, ki je gotovo dovolj velik. Med njima lahko zdaj z bisekcijo poiščemo najmanjši a, pri katerem ima elipsa kakšno skupno točko s krožnico s središčem C in polmerom 1 (rekli smo s krožnico, ne s krogom; kajti ker se elipsa začne v celoti zunaj kroga, se bo pri povečevanju a-ja najprej dotaknila roba kroga, to pa je krožnica).

Pri konkretnem a moramo torej znati preveriti, ali ima elipsa kakšno skupno točko s krožnico. Točka (x, y) leži na elipsi, če je $(x/a)^2 + (y/b)^2 = 1$, in na krožnici, če je $(x - x_C)^2 + (y - y_C)^2 = 1$. Tako imamo torej sistem dveh enačb z dvema neznankama, x in y. Za x pride seveda v poštev le presek intervalov $[-a, a]$ in $[x_C - 1, x_C + 1]$; pri vsakem takem x potem obstaja tako na elipsi kot na krožnici vsaj po ena točka, morda celo dve. Iz elipse dobimo $y = \pm b\sqrt{1 - x^2/a^2}$, iz krožnice pa $y = y_C \pm \sqrt{1 - (x - x_C)^2}$. Če dobimo isti y tako pri elipsi kot pri krožnici, smo našli presečišče. Rešujemo torej enačbo

$$s_1 b\sqrt{1 - x^2/a^2} = y_C + s_2\sqrt{1 - (x - x_C)^2},$$

kjer smo s $s_1, s_2 \in \{1, -1\}$ povedali, katero od obeh možnih točk (pri izbranem x) smo vzeli na elipsi in katero na krožnici. Če obe strani kvadriramo in enačbo malo preuredimo, dobimo:

$$Px^2 + Qx + R = 2y_C s_2\sqrt{1 - (x - x_C)^2}$$

za $P = 1 - b^2/a^2$, $Q = -2x_C^2$ in $R = x_C^2 - y_C^2 + b^2 - 1$. Kvadrirajmo obe strani še enkrat in ju še malo preuredimo, pa dobimo:

$$P^2x^4 + 2PQ\,x^3 + (2PR + Q^2 + 4y_C^2)x^2 + \\ 2(QR - 4x_C y_C^2)x + (R^2 - 4y_C^2(1 - x_C^2)) = 0.$$

Spet moramo torej poiskati ničle polinoma četrte stopnje, kar lahko naredimo z enakimi postopki kot pri prejšnji rešitvi. S to rešitvijo je torej tudi veliko dela, tako kot s prejšnjo, poleg tega pa je tudi počasnejša (zaradi bisekcije mora pri vsakem testnem primeru poiskati ničle več polinomov, ne le enega) in ima več težav z zaokrožitvenimi napakami. Prednost te rešitve pred prejšnjo pa je v tem, da nam zanjo ni bilo treba opraviti razmisleka o tem, da sta v iskani točki vpadni kót α in odbojni kót β enaka.

Zaradi zakrožitvenih napak je treba pri implementaciji bisekcije paziti na nekaj podrobnosti. Za vsako ničlo x našega polinoma preverimo, ali je x res realno število (torej ali nima še imaginarne komponente) z območja $[x_C - 1, x_C + 1]$; če je, pripadata krožnici pri tem x točki z y-koordinatama $y = \pm\sqrt{1 - (x - x_C)^2}$; in potem nas načeloma zanima, ali kakšna od tako dobljenih točk (po vseh ničlah polinoma) leži tudi na elipsi z goriščema $(\pm u, 0)$ in glavno polosjo a. Zaradi zaokrožitvenih napak seveda v praksi nobena od teh točk ne bo ležala točno na elipsi, pač pa bo morda

za neki ε oddaljena od nje. Kako velik ε naj še dovolimo, preden rečemo, da točka ne leži na elipsi? Če izberemo prevelik ε, bomo včasih pomotoma mislili, da se elipsa in krožnica dotikata (ali sekata), čeprav se ne, in bomo zato na koncu vrnili premajhno elipso (premajhen a); in podobno, če izberemo premajhen ε, bomo včasih pomotoma mislili, da se elipsa in krožnica ne dotikata, čeprav v resnici se, in bomo zato na koncu vrnili preveliko elipso (prevelik a). Pri naših poskusih nam nikakor ni uspelo izbrati ε tako, da bi dobili pravilen rezultat pri vseh testnih primerih z našega tekmovanja.

Bolje se je obnesel naslednji pristop: namesto da preverjamo, ali se elipsa seka s krožnico, vrnimo neki konkretni a, pri katerem se gotovo seka z njo.

funkcija PreizkusiA(a):
  iz a in globalnih spremenljivk u, x_C, y_C izračunaj še b
    in poišči ničle prej omenjenega polinoma četrte stopnje;
  $\tilde{a} := \infty$;
  za vsako ničlo x:
    if je x realno število z območja $[x_C - 1, x_C + 1]$:
      za $y \in \{y_C + \sqrt{1-(x-x_C)^2}, y_C - \sqrt{1-(x-x_C)^2}\}$:
        $a' := (\sqrt{(x-u)^2+y^2} + \sqrt{(x+u)^2+y^2})/2$;
        if $a' < \tilde{a}$ **then** $\tilde{a} := a'$;
  return $\tilde{a}$;

Pri vsaki točki (x, y) na krožnici, ki smo jo dobili pri kakšni od ničel x našega polinoma (in ki je zato kandidatka za točko na elipsi), torej izračunamo, na kolikšni elipsi (z goriščema $(\pm u, 0)$) v resnici leži: namreč na tisti z glavno polosjo a'. Med tako dobljenimi a' vrnemo najmanjšega; recimo mu $\tilde{a}$. Načeloma bi torej lahko rekli, da če je $\tilde{a}$ enak a (ali vsaj dovolj blizu a), potem se je elipsa z glavno polosjo a sekala s krožnico (ali se je vsaj dotikala — tega v bodoče ne bomo kar naprej ponavljali), sicer pa ne. Toda zaradi numeričnih nenatančnosti tega rezultata raje ne bomo uporabljali na ta način. Razmišljajmo raje takole:

postopek Bisekcija:
  $a_L := u$; $a_D := \left(\sqrt{(x_C-u)^2+y_C)^2} + \sqrt{(x_C+u)^2+y_C^2}\right)/2$;
  while $a_D - a_L > \varepsilon$:
    $a_M := (a_L + a_D)/2$;
    $\tilde{a} :=$ PreizkusiA(a_M);
    if $\tilde{a} \geq a_D$ **then** $a_L := a_M$
    else:
      $a_D := \tilde{a}$;
      if $\tilde{a} \geq (a_L + 3a_D)/4$ **then** $a_L := a_M$; $\qquad (\star)$
  return $2r \cdot a_D$; (* *rezultat, po katerem sprašuje naloga* *)

Vzdržujemo torej vrednost a-ja, ki je gotovo premajhna (a_L; na začetku izberemo $a_L = u$, ko je elipsa izrojena v daljico med goriščema in zato gotovo nima skupnih točk s krožnico), in vrednost, ki je gotovo dovolj velika (a_D; na začetku izberemo táko, pri kateri gre elipsa skozi središče krožnice). Na vsakem koraku bisekcije preizkusimo elipso s polosjo a_M na pol poti med a_L in a_D. Če je ta elipsa premajhna in se ne seka s krožnico, bomo dobili $\tilde{a} = \infty$ ali pa bomo za $\tilde{a}$ dobili glavno polos

neke malo večje elipse, ki pa se seka s krožnico; če je elipsa a_M dovolj velika in se seka s krožnico, pa bomo za $\tilde{a}$ dobili neko vrednost blizu a_M (razlika med njima pa bo le posledica zaokrožitvenih napak). Če je torej $\tilde{a} \geq a_D$, je to zagotovo znak, da je bila elipsa a_M premajhna, zato lahko a_L premaknemo na a_M; sicer pa smo v $\tilde{a}$ našli novo dovolj veliko elipso, manjšo od dosedanje zgornje meje a_D, zato lahko a_D premaknemo na $\tilde{a}$. To je načeloma že dovolj za uspešno rešitev; lahko pa nas žuli dejstvo, da premik a_D-ja na $\tilde{a}$ ne bo nujno razpolovil intervala $[a_L, a_D]$, morda ga bo celo le neznatno zmanjšal (če je $\tilde{a}$ le malo manjši od a_D), zato je težko reči, koliko iteracij bisekcije bo potrebnih. Zato smo dodali v gornjo rešitev še pogoj $(\star)$, ki v primerih, ko je $\tilde{a}$ dovolj velik (bližje a_D kot a_M), sklepa, da je bilo to posledica premajhne elipse (in ne zaokrožitvenih napak pri dovolj veliki elipsi), zato tudi premakne a_L na a_M. To zagotovi, da se bo interval $[a_L, a_D]$ v vsaki iteraciji bisekcije skrčil vsaj za četrtino.

§4. Rešitev s trisekcijo na loku $\mathcal{L}_1$. Vrnimo se spet k razmisleku, kjer je C v koordinatnem izhodišču, krožnica je enotska, točka A leži na pozitivnem delu x-osi, B pa nad x-osjo. Videli smo že, da nam premici CA in CB razdelita krožnico na štiri loke in da bomo globalni minimum funkcije f našli na loku $\mathcal{L}_1$, to je pri $\phi_A \leq \tau \leq \phi_B$. Videli smo tudi, da ima f ekstreme le tam, kjer je vpadni kót α enak odbojnemu kotu β, in da je odvod te funkcije enak $f'(\tau) = \sin\alpha - \sin\beta$.

Kaj se dogaja s kotoma α in β, če točko T počasi premikamo po loku $\mathcal{L}_1$ v pozitivni smeri, torej od polarnega kota ϕ_A do ϕ_B? Pri $\tau = \phi_A$ (ko leži T na A', torej na presečišču krožnice s poltrakom CA) je $\alpha = 0$ in $\beta > 0$; nato, ko se τ povečuje, se α povečuje in β zmanjšuje; nazadnje, pri $\tau = \phi_B$ (ko leži T na B', torej na presečišču krožnice s poltrakom CB), pa je $\beta = 0$ in $\alpha > 0$. Ker je torej na začetku $\alpha < \beta$, na koncu $\alpha > \beta$, ker se oba kota spreminjata zvezno in ker se α le povečuje, β pa le zmanjšuje, se vmes natanko enkrat izenačita; tam je torej edini ekstrem funkcije f na tem območju in tisto ne more biti drugega kot globalni minimum, ki ga iščemo. Preden dosežemo ta ekstrem, je $\alpha < \beta$, zato $f' < 0$ in funkcija f pada, kasneje pa je $\alpha > \beta$, zato $f' > 0$ in funkcija f narašča.

Funkcija f je na loku $\mathcal{L}_1$ torej unimodalna (sprva le pada, nato le narašča), zato lahko njen minimum poiščemo s trisekcijo. Za potrebe lažje implementacije jo lahko namesto s kotom τ parametriziramo s tem, katero točko na daljici AB seka poltrak CT. Recimo, da je to točka $(1-\lambda)A+\lambda B$ za neko $\lambda \in [0,1]$. Zapišimo naš postopek s psevdokodo:

$\lambda_L := 0$; $\lambda_D := 1$; $f^\star := \infty$;
while $\lambda_D - \lambda_L > \varepsilon$:
  for $k := 1$ **to** 2:
    $\lambda_k := ((3-k)\lambda_L + k\lambda_D)/3$;
    $U_k := (1-\lambda_k)A + \lambda_k B$; (* *točka na daljici AB* *)
    $T_k := U_k/|U_k|$; (* *projekcija U_k na krožnico* *)
    $f_k := |AT_k| + |T_k B|$;
  if $f_1 < f_2$ **then** $\lambda_D := \lambda_2$ **else** $\lambda_L := \lambda_1$;
  $f^\star := \min\{f^\star, f_1, f_2\}$;
return $f^\star$; (* *rezultat, po katerem sprašuje naloga* *)

Med trisekcijo torej vzdržujemo interval $[\lambda_L, \lambda_D]$, na katerem še utegne biti minimum, ki ga iščemo. Na vsakem koraku ga razdelimo na tretjine in izračunamo

vrednost funkcije f na začetku (f_1) in koncu (f_2) srednje tretjine intervala. Pri $f_1 < f_2$ vemo, da je funkcija f na srednji tretjini vsaj nekaj časa naraščala; in ker je unimodalna, to pomeni, da ko enkrat začne naraščati, kasneje nikoli več ne pada; torej bo naraščala tudi v zadnji tretjini in tam gotovo ne bomo našli minimuma; zato smemo desni rob intervala, λ_D, postaviti na konec srednje tretjine, torej na λ_2. Podoben razmislek nam pri $f_1 > f_2$ pove, da je funkcija v srednji tretjini vsaj nekaj časa padala, zato je padala tudi celotno prvo tretjino in tam ne bo minimuma, zato lahko levo krajišče λ_L prestavimo na začetek srednje tretjine, torej na λ_1. (Če je $f_1 = f_2$, je vseeno, katero krajišče premaknemo, lahko celo obe.)[30]

To je zdaj preprosta in učinkovita rešitev, ki je ni težko implementirati in ki nima posebnih težav z zaokrožitvenimi napakami. Za na tekmovanje je zelo primerna; vseeno pa si oglejmo še eno ali dve drugi.

§5. Rešitev z grobo silo. Vse naše dosedanje rešitve so temeljile na večji ali manjši količini razmišljanja, s katerim smo o obnašanju funkcije f sčasoma dognali dovolj, da smo potem vedeli, kje in kako se lotiti iskanja njenega minimuma. Toda recimo, da se nam o vsem tem ne bi dalo razmišljati; pred seboj imamo pač neko funkcijo $f(\tau)$ in iščemo njen minimum na območju $\tau \in [0, 2\pi)$. Izberimo si neki n in izračunajmo vrednost funkcije f v n točkah, enakomerno razporejenih vzdolž enotske krožnice, torej $f(\tau_k)$ za $\tau_k = 2\pi k/n$. Vzemimo nato tisto τ_k, ki je dala najmanjšo vrednost $f(\tau_k)$. Lahko si predstavljamo, da je minimum funkcije f nekje v bližini te točke; ne vemo pa zares, ali je med τ_k in τ_{k+1} ali med τ_{k-1} in τ_k. Zato se v nadaljevanju omejimo na interval $[\tau_{k-1}, \tau_{k+1}]$. Na tem intervalu zdaj nadaljujmo po enakem postopku, pri čemer morda ni treba tako velikega n kot na začetku, ker je tudi interval že precej ožji. Konkreten primer implementacije takšnega postopka bi bil na primer:

```
n := n1; τL := 0; τD := 2π;
while τD − τL > ε:
  △τ := (τD − τL)/n; (* razbijmo [τL, τD] na n podintervalov širine △τ *)
  f* := ∞; τ* := NIL;
  for k := 0 to n:
    τk := τL + k△τ; fk := f(τk);
    if fk < f* then f* := fk, τ* := τk;
  τL := τ* − △τ; τD := τ* + △τ;
  n := n2;
return f*; (* rezultat, po katerem sprašuje naloga *)
```

Tu smo torej v prvem koraku, ko imamo pred seboj še celo krožnico, razbili interval $[\tau_L, \tau_D]$ na n_1 delov, kasneje pa le še na n_2. Tako lahko prihranimo čas z manjšim n_2, ko so intervali že tako ali tako ozki. Vzemimo na primer $n_1 = 100$ in $n_2 = 10$.

Ta rešitev je zelo preprosta za implementacijo, ne zahteva veliko razmišljanja in pravilno reši vse testne primere z našega tekmovanja. Seveda pa v splošnem

[30] Še en podoben postopek, ki ga lahko uporabimo namesto trisekcije, je iskanje z zlatim rezom. Namesto da razdelimo interval $[\lambda_L, \lambda_D]$ na tri enake dele (kot smo naredili pri trisekciji), ga razdelimo na tri dele v razmerju $\phi : 1 : \phi$, kjer je $\phi = (1 + \sqrt{5})/2$ razmerje zlatega reza. To bo zagotovilo, da bo ena od delilnih točk $\lambda_{1,2}$ prišla prav tudi v naslednji iteraciji, zato si lahko vrednost funkcije f v tej točki zapomnimo in nam je kasneje ne bo treba računati še enkrat. V vsaki iteraciji bomo morali zato izračunati vrednost f le v eni od točk $\lambda_{1,2}$, v drugi pa jo bomo poznali že od prej.

ta pristop ne bi deloval za vsako funkcijo f; pri naši nalogi deluje le zato, ker se funkcija f obnaša dovolj lepo. Lahko bi nas na primer skrbela možnost, da bi imela f v točkah τ_k in τ_{k+1} neko razmeroma visoko vrednost, med njima pa se bi hitro spustila do svojega globalnega minimuma in se nato spet dvignila z njega; nekje drugje, v neki kasnejši točki $\tau_{k'}$, pa bi morda dosegla vrednost, manjšo od tistih v τ_k in τ_{k+1}, vendar večjo od globalnega minimuma. To bi lahko naš postopek zavedlo, da bi se po prvi iteraciji zunanje zanke osredotočil na okolico točke $\tau_{k'}$ namesto na okolico točk τ_k oz. τ_{k+1}, s tem pa bi zgrešil pravi globalni minimum. Večji ko je n_1, ožji so naši intervali v prvi iteraciji zunanje zanke in bolj nenadno bi se morala f dvigati in spuščati, da bi lahko prišlo do opisanega neugodnega scenarija; ravno zato smo vzeli razmeroma visok n_1.

Na tekmovanju bi se tak šušmarski pristop obnesel, za v bilten pa se spodobi razmisliti o tem, zakaj ta rešitev deluje in pri kakšnih n_1 oz. n_2 deluje. Spomnimo se spet naše razdelitve enotske krožnice na štiri loke; videli smo, da ima f globalni minimum na $\mathcal{L}_1$, kjer je to tudi njen edini ekstrem; ko premikamo T po tem loku v pozitivni smeri, vrednost funkcije f najprej nekaj časa pada in potem narašča. Videli smo tudi, da na loku $\mathcal{L}_2$ vrednost funkcije f ves čas narašča, na $\mathcal{L}_4$ pa ves čas pada. Za lok $\mathcal{L}_3$ pa nam podoben razmislek kot v prejšnjem razdelku za $\mathcal{L}_1$ pokaže, da f na njem najprej nekaj časa narašča in nato nekaj časa pada: na $\mathcal{L}_3$ kót α narašča od -180° (pri $\tau = \phi_A + 180^\circ$) do neke večje (vendar še vedno negativne) vrednosti pri $\tau = \phi_B + 180^\circ$ (torej na koncu loka $\mathcal{L}_3$), kót β pa pada od neke negativne vrednosti, večje od -180° (pri $\tau = \phi_A + 180^\circ$), do vrednosti -180° na koncu loka (pri $\tau = \phi_B + 180^\circ$). Sprva je torej $\alpha < \beta$, na koncu pa $\alpha > \beta$; vmes se torej natanko enkrat zgodi, da je $\alpha = \beta$, zato pa $f' = 0$; tam je torej edini ekstrem funkcije f na $\mathcal{L}_3$ in zanj že vemo, da je globalni maksimum; pred njim torej f pada, za njim pa narašča.

Vidimo torej, da je f v nekem smislu unimodalna ne le na $\mathcal{L}_1$ (kot smo videli že v prejšnjem razdelku), pač pa po celi krožnici: od globalnega minimuma do globalnega maksimuma ves čas le narašča, nato pa od globalnega maksimuma do globalnega minimuma ves čas le pada.

Spomnimo se, da naš postopek začne z nekim intervalom možnih vrednosti τ, ga razdeli na n enako dolgih podintervalov in se v naslednji iteraciji glavne zanke omeji na dva od njih. Podintervali morajo biti torej vsaj trije, sicer bi bil naš novi interval enak prejšnjemu. Recimo torej, da vzamemo $n_1 = n_2 = 3$. Razmislimo o treh vrednostih τ_0, τ_1, τ_2, ki jih pregledamo v prvi iteraciji glavne zanke. Recimo brez izgube za splošnost, da med temi tremi dobimo najmanjšo vrednost funkcije f pri τ_1; v naslednji iteraciji glavne zanke se bomo torej omejili na interval $[\tau_0, \tau_2]$, zavrgli pa bomo območje od τ_2 do τ_0. Za novi interval tudi vemo, da ima na sredi tega intervala (v točki $\tau_1 = (\tau_0 + \tau_2)/2$) funkcija nižjo vrednost kot na njegovih krajiščih.

Prepričajmo se zdaj, da ta lastnost velja tudi v nadaljevanju glavne zanke: na koncu vsake iteracije glavne zanke imamo pred seboj tak interval $[\tau_L, \tau_D]$, za katerega velja, da ima funkcija na sredi intervala manjšo vrednost kot v krajiščih. Za konec prve iteracije smo to pravkar dokazali.

Recimo, da je naša invarianta veljala na koncu prejšnje iteracije in s tem na začetku trenutne; prepričajmo se, da bo veljala tudi na koncu trenutne iteracije. Ker

ima funkcija na sredi intervala manjšo vrednost kot v krajiščih, to pomeni, da na tem intervalu f najprej nekaj časa pada in nato odtlej narašča. V trenutni iteraciji glavne zanke razdelimo interval $[\tau_L, \tau_D]$ na tri dele; pišimo recimo $\tau_a = (2\tau_L + \tau_D)/3$ in $\tau_b = (\tau_L + 2\tau_D)/3$. Naš postopek pogleda, v kateri izmed točk $\tau_L, \tau_a, \tau_b, \tau_D$ je vrednost funkcije f najmanjša. Ali bi se lahko zgodilo, da bi bila vrednost funkcije najmanjša pri τ_L ali τ_D, ne pa pri τ_a ali τ_b? Recimo brez izgube za splošnost, da je najmanjša pri τ_L. Rekli smo, da od tam naprej funkcija nekaj časa pada, toda ker je pri τ_a že višja kot pri τ_L, to pomeni, da začne že na $[\tau_L, \tau_a]$ spet naraščati in odtlej narašča vse do τ_D. Torej je na sredi intervala, pri $(\tau_L + \tau_D)/2$ — kar je hkrati tudi $(\tau_a + \tau_b)/2$ — višja kot pri τ_a in s tem višja kot pri τ_L; to pa je protislovje, saj smo prej rekli, da je vrednost funkcije na sredi intervala $[\tau_L, \tau_D]$ nižja kot v njegovih krajiščih.

Vidimo torej, da bomo v trenutni iteraciji glavne zanke najnižjo vrednost funkcije opazili v eni izmed točk τ_a in τ_b (in ne v τ_L ali τ_D). Recimo brez izgube za splošnost, da v τ_a. Na koncu iteracije se bomo torej omejili na interval $[\tau_L, \tau_b]$, na sredi tega intervala pa je točka τ_a, kjer ima funkcija res nižjo vrednost kot v τ_L in τ_b; tako velja naša invarianta tudi na koncu trenutne iteracije. □

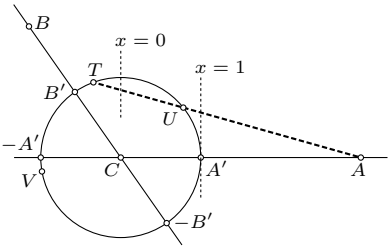
Iz invariante, ki smo jo pravkar dokazali, pa sledi, da funkcija f na intervalu $[\tau_L, \tau_D]$ najprej nekaj časa pada in kasneje nekaj časa narašča, torej ima na tem intervalu svoj minimum. Ker postane interval v vsaki iteraciji za tretjino ožji, se bomo sčasoma temu minimumu približali do poljubne želene natančnosti.

Tako torej vidimo, da naš postopek deluje pravilno že z $n_1 = n_2 = 3$; razmislek je pokazal, da je bila naša previdnost, ko smo sprva vzeli $n_1 = 100$ in $n_2 = 10$, odveč.

§6. Še ena rešitev z grobo silo. Ekstremi funkcije f nastopijo tam, kjer ima njen odvod vrednost 0, torej kjer f' preide iz negativnih vrednosti v pozitivne ali obratno. Zato nam lahko pride na misel, da bi za točke $\tau_k = 2\pi k/n$ $(k = 0, \ldots, n)$ izračunali $f'(\tau_k)$ in gledali, kdaj sta dve zaporedni vrednosti $f'(\tau_k)$ in $f'(\tau_{k+1})$ različno predznačeni. Takrat vemo, da mora na intervalu med njima, $[\tau_k, \tau_{k+1}]$, nastopiti ničla funkcije f' in s tem ekstrem funkcije f. To ničlo oz. ekstrem lahko poiščemo z bisekcijo po intervalu $[\tau_k, \tau_{k+1}]$. Med vrednostmi funkcije f v vseh tako odkritih ekstremih na koncu vrnemo najmanjšo; tisto mora biti globalni minimum.

```
f* := ∞;
for k := 0 to n − 1:
  τ_L := (2πk/n); τ_D := (2π(k + 1)/n);
  if sta f'(τ_L) in f'(τ_D) obe > 0 ali obe < 0 then continue;
  while τ_D − τ_L > ε:
    τ_M := (τ_L + τ_D)/2;
    if sta f'(τ_L) in f'(τ_M) obe > 0 ali obe < 0
    then τ_L := τ_M else τ_D := τ_M;
  f* := min{f*, f(τ_L)};
return f*; (* rezultat, po katerem sprašuje naloga *)
```

V vsaki iteraciji bisekcije torej zavržemo tisto polovico intervala, kjer ima odvod na začetku in na koncu enak predznak. Ko se interval dovolj zoži, da sta krajišči τ_L in τ_D približno enaki, mora biti to naš ekstrem in vrednost funkcije f v njem je eden od kandidatov za globalni minimum (med katerimi na koncu vrnemo najmanjšega).



Slika 9.

Hipotetičen primer, če bi imela f minimum v točki T, maksimum v V in bi si bili tidve točki manj kot 90° narazen (med njima je seveda tudi celoten lok $\mathcal{L}_2$, od B' do $-A'$). Tedaj je (če je A na pozitivni x-osi) T nujno v drugem kvadrantu in daljica AT (debela črtkana črta) se neizogibno seka s krožnico v neki točki U. Navpični črtkani črti kažeta $x = 0$, kjer je krožnica nad daljico AT, in $x = 1$, kjer je krožnica pod daljico AT; vmes se morata torej sekati.

Ta rešitev je, podobno kot tista iz prejšnjega razdelka, preprosta za implementacijo in ne zahteva veliko razmišljanja. Seveda nas lahko skrbi možnost, da ima f' na nekem intervalu $[\tau_k, \tau_{k+1}]$ dve ničli, torej je na začetku in na koncu enako predznačen in se zato naša gornja rešitev s tem intervalom ne bi ukvarjala; toda ker ima f' na njem dve ničli, ima f na njem dva ekstrema, en minimum in en maksimum. Tako lahko kak minimum spregledamo, morda celo globalnega. Da to tveganje zmanjšamo — če smo na tekmovanju in se nam mudi — lahko na primer vzamemo neki kolikor toliko velik n, recimo $n = 100$, oddamo rešitev in vidimo, da pravilno reši vse testne primere. Tu v biltenu pa se vseeno lahko s premislekom prepričamo, da ta rešitev vedno najde pravi rezultat že pri $n = 4$.

Kot smo videli že v prejšnjem razdelku, ima f minimum na $\mathcal{L}_1$, maksimum na $\mathcal{L}_3$, drugih ekstremov pa sploh nima. Težava iz prejšnjega odstavka torej nastopi, če ležita minimum in maksimum oba na istem intervalu $[\tau_k, \tau_{k+1}]$; potem bo f' na začetku in na koncu intervala enako predznačen in naš postopek na njem sploh ne bo izvedel bisekcije. Pa recimo, da se pri $n = 4$ to lahko zgodi. Krožnico smo razdelili na $n = 4$ intervale, torej so dolgi po 90 stopinj. Na istem intervalu ležita minimum in maksimum funkcije f; recimo brez izgube za splošnost, da nastopi minimum pred maksimumom (slika 9). Ker leži minimum na loku $\mathcal{L}_1$, maksimum pa na $\mathcal{L}_3$, mora na našem intervalu ležati tudi celoten lok $\mathcal{L}_2$, ki leži med $\mathcal{L}_1$ in $\mathcal{L}_3$. Ker je interval dolg 90°, mora biti območje od minimuma do konca $\mathcal{L}_2$ krajše od 90°; in ker sta $\mathcal{L}_1$ in $\mathcal{L}_2$ skupaj dolga 180°, mora biti torej območje od začetka $\mathcal{L}_1$ (to je od točke A', kjer poltrak CA seka krožnico) do minimuma (recimo mu T) daljše od 90°. To pa pomeni, da daljica AT gotovo seka krožnico: spomnimo se, da imamo že ves čas koordinatni sistem obrnjen tako, da je A na pozitivni x-osi ($x_A > 0$, $y_A = 0$); T je torej v drugem kvadrantu ($-1 < x_T < 0$ in $0 < y_T < 1$); za vsako x-koordinato med x_T in x_A velja, da je pri njej y-koordinata daljice AT nekje med 0 in y_T; pri $x = 0$ je y-koordinata krožnice enaka $1 > y_T$, torej je krožnica nad daljico; pri $x = 1$ pa je y-koordinata krožnice enaka 0, torej je krožnica pod daljico; nekje vmes se morata torej krožnica in daljica sekati. Recimo temu presečišču U; naša pot od A prek T do B je potem dolga $|AT|+|TB| = |AU|+|UT|+|TB|$, kar je po trikotniški neenakosti (za $\triangle UTB$) $< |AU| + |UB|$, torej je U boljša rešitev od T, to pa je v protislovju s predpostavko, da je imela funkcija f v točki T svoj minimum. □

§7. Težja različica naloge. Razmislimo še o težji različici naloge, ki jo omenja opomba pod črto v besedilu naloge: tu nas torej zanima najkrajša taka pot od A do B, ki ima kakšno skupno točko s krožnico (in ne le s krogom) s središčem v C in polmerom r. Razlika v primerjavi s prvotno nalogo nastopi le v primeru, ko ležita A in B obe znotraj krožnice; takrat smo lahko pri prvotni nalogi uporabili za pot

kar daljico AB, zdaj pa to ne bo veljavna pot (ker se ne dotakne krožnice).

Preden se temu primeru posvetimo v splošnem, je koristno posebej obravnavati primer, ko sta A in B znotraj krožnice in so točke A, B in C kolinearne. Takrat moramo pogledati, katera od točk A in B je bliže krožnici, in iti od nje do najbližje točke na krožnici in nato do druge izmed točk A in B; rezultat, po katerem sprašuje naloga, je torej $|AB| + 2 \cdot (1 - \max\{|A|, |B|\})$.

Recimo torej zdaj, da sta A in B znotraj krožnice in da A, B in C niso kolinearne. Razmislek, s katerim smo se v §1 prepričali, da je $f'_U(\tau) = \sin \psi_U(\tau)$, deluje tudi zdaj, saj se ni nikjer opiral na to, da je U zunaj krožnice namesto znotraj nje. Pač pa je zdaj, ko govorimo o vpadnem in odbojnem kotu, smiselno tadva meriti glede na smer, ki iz T kaže v notranjost krožnice namesto v zunanjost, saj se tudi pot iz A prek T v B vedno odbije od krožnice z notranje strani. Definirajmo torej $\alpha = 180^\circ - \psi_A(\tau)$ in $\beta = \psi_B(\tau) - 180^\circ$. Tudi pri teh definicijah velja $f'(\tau) = f'_A(\tau) + f'_B(\tau) = \sin \psi_A(\tau) + \sin \psi_B(\tau) = \sin \alpha - \sin \beta$. Tudi zdaj je α pozitivna na $\mathcal{L}_{1,2}$ in negativna na $\mathcal{L}_{3,4}$, kót β pa je pozitiven na $\mathcal{L}_{4,1}$ in negativen na $\mathcal{L}_{2,3}$. Razlika v primerjavi s prvotno nalogo, kjer sta bila A in B zunaj krožnice, pa je zdaj ta, da sta kota α in β po absolutni vrednosti zdaj vedno manjša ali enaka 90 stopinj.[31] Zato lahko do $\sin \alpha = \sin \beta$ (in s tem do $f'(\tau) = 0$, torej da ima funkcija f ekstrem v točki τ) pride le tako, da sta α in β enaka oz. (kar je ekvivalentno) da je $\psi_A(\tau) = -\psi_B(\tau)$.

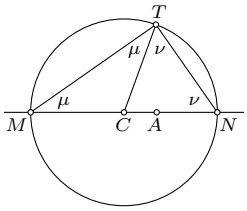
Zato naša eksaktna rešitev iz §2, ki je temeljila na reševanju pogoja $\psi_A(\tau) = -\psi_B(\tau)$, deluje brez sprememb tudi za naš sedanji primer, ko sta A in B znotraj krožnice.

Tudi rešitev z napihovanjem elipse iz §3 deluje skoraj brez sprememb, saj se nič ne zanaša na dejstvo, da sta bili pri prvotni različici naloge A in B zunaj krožnice. Paziti moramo le pri postavljanju začetne zgornje meje a_D pri bisekciji: v §3 smo vzeli takšno a_D, pri kateri elipsa z glavno polosjo a_D (in goriščema A in B) teče skozi C; zdaj pa, ko sta A in B znotraj krožnice, je mogoče, da taka elipsa skozi C tudi sama v celoti leži znotraj krožnice, mi pa moramo na začetku za a_D vzeti dovolj veliko elipso, da vsebuje vsaj neko točko na krožnici. Namesto elipse skozi C lahko izberemo poljubno točko na krožnici, recimo $U = (x_C + 1, y_C)$, in vzamemo elipso skozi U: to je pri $a_D = (\sqrt{(x_C + 1 - u)^2 + y_C^2} + \sqrt{(x_C + 1 + u)^2 + y_C^2})/2$.

§8. Rešitev s trisekcijo pri težji različici naloge. Pri rešitvah iz §§4–6 se stvari malo zapletejo. Še vedno drži razmislek iz §1, da leži globalni minimum funkcije f na $\mathcal{L}_1$, globalni maksimum na $\mathcal{L}_3$ in da na lokih $\mathcal{L}_2$ in $\mathcal{L}_4$ funkcija f nima ekstremov. Ni pa zdaj več nujno res, da se α po loku $\mathcal{L}_1$ le povečuje, β pa le zmanjšuje, zato funkcija f na $\mathcal{L}_1$ ni nujno unimodalna.

Rešitev iz §2 nam je pokazala, da nastopijo ekstremi funkcije f v ničlah nekega polinoma četrte stopnje, torej so ekstremi največ štirje; poleg tega so ekstremi izmenično minimumi in maksimumi; ker se f na krožnici obnaša ciklično, pa mora

[31] Prepričajmo se o tem za α (razmislek za β je analogen). Postavimo za začetek M in N v točki $-A'$ in A', kjer premica CA seka krožnico; narišimo trikotnika $\triangle CTM$ in $\triangle CTN$; kota v ogliščih M oz. N imenujmo μ oz. ν (slika 10). Ker je $|CM| = |CN| = |CT| = 1$, sta trikotnika $\triangle CTM$ in $\triangle CTN$ enakokraka; zato je $\angle CTM = \angle TMC = \mu$ in $\angle CTN = \angle TNC = \nu$; zato je $\angle MTN = \angle MTC + \angle CTN = \mu + \nu$; v trikotniku $\triangle MTN$ je zato vsota kotov enaka $180^\circ = \mu + (\mu + \nu) + \nu$, zato je $\angle MTN = \mu + \nu = 90^\circ$. Če zdaj točki M in N premaknemo bližje skupaj po premici CT, se mora kót v točki T zmanjšati; premaknimo M vse do C, točko N pa vse do A, pa lahko zaključimo, da je $\alpha = \angle CTA < 90^\circ$. □



Slika 10.

biti ekstremov sodo mnogo, sicer bi pri premikanju vzdolž krožnice nekje dobili dva minimuma ali dva maksimuma zaporedoma. Torej ima f dva ali štiri ekstreme. Ker f na $\mathcal{L}_4$ pada, mora biti zadnji ekstrem na $\mathcal{L}_3$ maksimum, prvi ekstrem na $\mathcal{L}_1$ pa minimum; podobno, ker f na $\mathcal{L}_2$ narašča, mora biti zadnji ekstrem na $\mathcal{L}_1$ minimum, prvi ekstrem na $\mathcal{L}_3$ pa maksimum. Na $\mathcal{L}_1$ sta torej prvi in zadnji ekstrem minimuma, torej ima f tam liho mnogo ekstremov — bodisi enega bodisi tri.

Za začetek razmislimo o posebnem primeru, ko je $|A| = |B|$. Takrat je položaj točk A in B simetričen glede na premico, ki razpolavlja kot $\angle ACB$ (in s tem tudi lok $\mathcal{L}_1$). Zato je tudi funkcija f simetrična glede na razpolovišče loka $\mathcal{L}_1$; za vsak υ velja $f(\phi_A+\upsilon) = f(\phi_B-\upsilon)$. Tudi ekstremi funkcije f so zato razporejeni simetrično glede na omenjeno premico. Vemo že, da ima f na $\mathcal{L}_1$ bodisi en ekstrem bodisi tri in da je med temi ekstremi vsekakor tudi globalni minimum. Če je ekstrem na $\mathcal{L}_1$ en sam, mora biti točno na razpolovišču loka (saj drugače razporeditev ekstremov ne bi bila simetrična), torej pri $\tau = (\phi_A + \phi_B)/2$. Če pa so ekstremi trije, mora biti eden na prvi polovici loka, eden na razpolovišču in eden na drugi polovici; ker se morajo minimumi in maksimumi izmenjevati, sta torej prvi in tretji ekstrem minimuma, tisti na razpolovišču pa je maksimum. Ker je f simetrična, sta oba minimuma enako dobra. Vidimo torej, da bi se lahko v tem posebnem primeru (pri $|A| = |B|$) omejili na eno polovico loka $\mathcal{L}_1$, recimo od ϕ_A do $(\phi_A + \phi_B)/2$; tu je funkcija f unimodalna (bodisi ves čas pada, če je minimum na razpolovišču loka $\mathcal{L}_1$; bodisi najprej pada do minimuma in potem narašča do lokalnega maksimuma na razpolovišču) in lahko njen minimum poiščemo s trisekcijo, tako kot v §4.

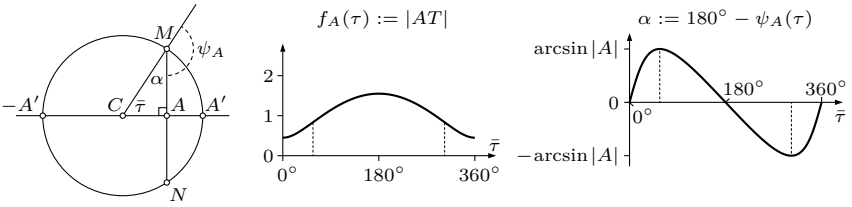
V nadaljevanju lahko brez izgube za splošnost predpostavimo, da je $|A| > |B|$.

§8.1. Drugi odvod funkcije f_U ter obnašanje kotov α in β. V §1 smo že izpeljali prvi odvod funkcije f_U (enačba 1): $f'_U(\tau) = |CU|\sin(\tau-\phi_U)/f_U(\tau)$. Odvajajmo ga zdaj še enkrat:

$$\begin{aligned} f''_U(\tau) &= |CU|\cos(\tau-\phi_U)/f_U(\tau) - |CU|\sin(\tau-\phi_U)/f_U^2(\tau)\cdot f'_U(\tau) \\ &= |CU|\cos(\tau-\phi_U)/|UT| - |CU|^2\sin^2(\tau-\phi_U)/|UT|^3. \end{aligned} \tag{3}$$

Razmislimo zdaj, kako se spreminja kót α, če počasi povečujemo τ (slika 11). Pri $\tau = \phi_A$ je $T = A'$ in $\alpha = 0$. Nato se α nekaj časa povečuje in doseže svoj maksimum takrat, ko je $\triangle CAT$ pravokoten, to je pri $\tau = \phi_A + \arccos|A|$, ko je $\alpha = \arcsin|A|$; nato pa začne α spet padati vse do $\tau = \phi_A + 180^\circ$, ko α spet doseže 0.

O tem, da α res doseže svoj maksimum ravno pri $\phi_A + \arccos|A|$, se prepričamo takole: ker α, kot smo videli, ne preseže 90°, se skupaj z njo ves čas povečuje tudi njen sinus; α ima torej maksimum tam, kjer ga ima tudi $\sin\alpha$; vemo pa, da je



Slika 11.

Pišimo $\bar\tau := \tau - \phi_A$. — *Levo:* če postavimo T v eno od točk M in N (to je pri $\tau = \phi_A \pm \arccos|A|$, ko je trikotnik CAT pravokoten), doseže α svojo največjo oz. najmanjšo vrednost (namreč $\pm\arcsin|A|$). — *Sredina in desno:* grafa kažeta $f_A(\tau) = |AT|$ in vpadni kót $\alpha = 180^\circ - \psi_A(\tau)$. Navpične črtkane črtice kažejo dogajanje pri $\tau = \phi_A \pm \arccos|A|$ (tam ima α ekstrema, f_A pa prevoja).

$\sin\alpha = f'_A(\tau)$, zato $\sin\alpha$ doseže svoj maksimum takrat, ko je $f''_A(\tau) = 0$. Spomnimo se, da smo koordinatni sistem načeloma zasukali tako, da je $\phi_A = 0$ oz. da leži A na pozitivnem delu x-osi; njene koordinate so torej $A = (x, 0)$ za $x = |A|$. Ko je trikotnik $\triangle CAT$ pravokoten, leži T navpično nad A in ima torej koordinate $T = (x, y)$ za $y = \sqrt{1 - x^2}$; obenem je seveda po definiciji vedno $T = (\cos\tau, \sin\tau)$, torej je $x = \cos\tau$ in $y = \sin\tau$. Če vstavimo to v enačbo za f''_A, dobimo $|CA|\cos\tau/|AT| - |CA|^2\sin^2\tau/|AT|^3 = x \cdot x/y - x^2 \cdot y^2/y^3 = 0$, torej ima pri tem τ funkcija f''_A res ničlo, funkcija $f'_A = \sin\alpha$ pa ekstrem (in ima zato tam svoj ekstrem tudi α sama).

Za kot β je razmislek seveda analogen. Recimo, da τ počasi zmanjšujemo; pri $\tau = \phi_B$ je $T = B'$ in $\beta = 0$; ko se τ zmanjšuje, se β povečuje in doseže svoj maksimum, $\beta = \arcsin|B|$, pri $\tau = \phi_B - \arccos|B|$. Od tam naprej se začne β spet zmanjševati in pri $\tau = \phi_B - 180^\circ$ spet doseže 0.

Ker je $|A| > |B|$, je tudi $\arcsin|A| > \arcsin|B|$, torej je maksimalna α večja od maksimalne β; po drugi strani pa je $\arccos|A| < \arccos|B|$, zato je interval kotov τ, kjer α narašča od 0 do svojega maksimuma, ožji od intervala, na katerem β pada od svojega maksimuma do 0.

§8.2. Makov kriterij. V §1 smo si pomagali s sinusnim izrekom v $\triangle CUT$, da smo pokazali, da je $f'_U(\tau) = |CU|\sin(\tau - \phi_U)/|UT|$ naprej enako $\sin\psi$ za $\psi = \psi_U(\tau)$. Uporabimo to ugotovitev tudi pri drugem členu formule (3) za drugi odvod, pa dobimo:

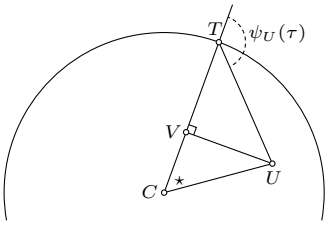
$$\begin{aligned} f''_U(\tau) &= |CU|\cos(\tau - \phi_U)/|UT| - |CU|^2\sin^2(\tau - \phi_U)/|UT|^3 \\ &= |CU|\cos(\tau - \phi_U)/|UT| - \sin^2\psi/|UT| \\ &= (|CU|\cos(\tau - \phi_U) - 1 + \cos^2\psi)/|UT|. \end{aligned}$$

Naj bo V pravokotna projekcija točke U na stranico CT (slika 12). Potem nam pravokotni trikotnik $\triangle CVU$ (s kotom $t - \phi_U$ v oglišču C) pove, da je $|CV| = |CU|\cos(t - \phi_U)$; nadaljujmo torej prejšnjo izpeljavo:

$$= (|CV| - 1 + \cos^2\psi)/|UT|;$$

upoštevajmo, da je $|CV| = |CT| - |VT| = 1 - |VT|$:

$$= (-|VT| + \cos^2\psi)/|UT|.$$



Slika 12 ($\star = \tau - \phi_U$).

Pravokotni trikotnik $\triangle UVT$ (s kotom $180^\circ - \psi$ v oglišču T) nam pove, da je $|VT| = |UT|\cos(180^\circ - \psi) = -|UT|\cos\psi$:

$$\begin{aligned} &= (|UT|\cos\psi + \cos^2\psi)/|UT| \\ &= \cos\psi\,(\cos\psi/|UT| + 1). \end{aligned}$$

Pri $U = A$ imamo $\alpha = 180^\circ - \psi_A(\tau)$, zato $\cos\psi_A(\tau) = -\cos\alpha$ in dobimo $f''_A(\tau) = \cos\alpha\,(\cos\alpha/|AT| - 1)$; pri $U = B$ pa imamo $\beta = \psi_B(\tau) - 180^\circ$, zato $\cos\psi_B(\tau) = -\cos\beta$ in dobimo $f''_B(\tau) = \cos\beta\,(\cos\beta/|BT| - 1)$.

V ekstremih funkcije f sta kota α in β enaka; takrat potem velja

$$\begin{aligned} f''(\tau) &= f''_A(\tau) + f''_B(\tau) \\ &= \cos\alpha\,(\cos\alpha/|AT| - 1) + \cos\beta\,(\cos\beta/|BT| - 1) \\ &= \cos\alpha\,(\cos\alpha(1/|AT| + 1/|BT|) - 2). \end{aligned}$$

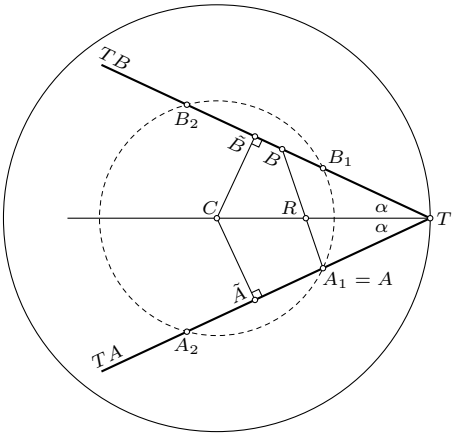
Spomnimo se, da nam v ekstremu funkcije predznak njenega drugega odvoda pove, ali je ta ekstrem minimum ali maksimum: pri minimumu je drugi odvod pozitiven, pri maksimumu pa negativen. V našem primeru, ker je α gotovo na $[0^\circ, 90^\circ)$, je prvi faktor, $\cos\alpha$, gotovo pozitiven, zato je predznak vrednosti $f''(\tau)$ odvisen le od drugega faktorja. Tako torej vidimo: ekstrem τ je minimum (oz. maksimum), če je $m := \cos\alpha(1/|AT| + 1/|BT|)$ večje (oz. manjše) od 2. To nam bo prišlo v nadaljevanju še večkrat prav.[32]

§8.3. Dogajanje na območju, kjer α narašča. Videli smo, da α narašča od $\tau = \phi_A$ (ko je $\alpha = 0$) do $\tau = \phi_A + \arccos|A|$ (ko doseže α svojo maksimalno vrednost, namreč $\arcsin|A|$). To območje leži na začetku loka $\mathcal{L}_1$, morda pa celo pokriva ta lok v celoti in sega deloma še na $\mathcal{L}_2$ (to se zgodi v primeru, če je $\phi_B - \phi_A < \arccos|A|$).

Kaj se dogaja s kotom $\angle CAT$, če točko T počasi premikamo naprej po tem območju? Na začetku leži T na poltraku CA in je $\angle CAT = 180^\circ$, nato pa se ta kót počasi zmanjšuje in na koncu območja (ko je AT pravokotna na CA) pade na 90°. Na območju, kjer α narašča, je torej kot $\angle CAT$ top, kasneje pa (kjer α pada) je oster.

Glede kota β pa vidimo, da je na začetku tega območja (pri $\tau = \phi_A$) gotovo $\beta > 0$, kajti β je bila enaka 0 pri $\phi_B - 180^\circ$ in je od tam počasi naraščala. Na začetku tega območja je torej $\beta > \alpha$, na koncu pa je gotovo $\beta < \alpha$, kajti α je tam

[32] Kriterij, ki smo ga tu izpeljali (oz. ugotovitev, da ima $f''(\tau)$ enak predznak kot $m - 2$), je opisan v: S-y. Mak, "A closer look at Fermat's Principle", *Physics Education* 21(6):365–8 (Nov. 1986), na str. 366.



Slika 13.

Primer, ko ima f v točki T ekstrem in sta zato vpadni in odbojni kót oba enaka α. Debeli črti predstavljata poltraka TA in TB, črtkana krožnica pa ima središče C in polmer $|A|$. Zato mora biti A ena od točk A_1 in A_2 (odvisno od tega, ali je kót $\angle CAT$ top ali oster, oz. — kar je ekvivalentno — od tega, ali α v točki T narašča ali pada); v primeru na sliki je $A = A_1$. Ker je po predpostavki $|B| < |A|$, mora B ležati na daljici B_1B_2 (in to ne v kakšnem od njenih krajišč). Točka R je presečišče daljice AB s poltrakom TC.

na svoji maksimalni vrednosti in ta je večja od maksimalne vrednosti kota β. Vmes torej gotovo obstaja vsaj ena točka T, v kateri je $\alpha = \beta$ in ima funkcija f tam ekstrem.

Zasukajmo koordinatni sistem tako, da leži T na pozitivnem delu x-osi (slika 13). Narišimo iz T poltraka, ki s poltrakom TC oklepata kót α (debeli črti na sliki); na enem od teh poltrakov mora ležati točka A, na drugem pa B. Projekcijama točke C na tadva poltraka recimo $\tilde{A}$ in $\tilde{B}$. Črtkana krožnica na sliki kaže, katere točke so od C oddaljene točno $|CA|$. Na poltraku TA sta dve taki točki; tisti izmed njiju, ki je bližje T, recimo A_1, drugi pa A_2 (točka $\tilde{A}$ potem leži točno na polovici poti med njima). Podobno na poltraku TB dobimo točki B_1 in B_2.

Pri $\tilde{A}$ je $\angle C\tilde{A}T$ pravi kot; če se od tam premikamo proti točki T, se ta kot povečuje, če se od T oddaljujemo, pa se zmanjšuje; kot $\angle CA_1T$ je torej top, kot $\angle CA_2T$ pa oster.

Točka A leži na poltraku TA in je od C oddaljena $|A|$, torej mora biti to ena izmed A_1 in A_2. Ker smo prej videli, da je kot $\angle CAT$ top, mora biti $A = A_1$. Točka B pa leži na poltraku TB in je od C oddaljena $|B| < |A|$, zato mora ležati med točkama B_1 in B_2.

Trikotnik $\triangle C\tilde{A}T$ ima pravi kot v oglišču $\tilde{A}$, hipotenuzo CT dolžine 1 (saj T leži na naši enotski krožnici, C pa je njeno središče), v oglišču T pa ima kot α; zato je $|\tilde{A}T| = \cos\alpha$. Enako dobimo tudi na drugi strani: $|\tilde{B}T| = \cos\alpha$.

Izračunajmo zdaj Makov kriterij:

$$\begin{aligned} m &= \cos\alpha(1/|AT| + 1/|BT|) = \cos\alpha/|AT| + \cos\alpha/|BT| \\ &= |\tilde{A}T|/|AT| + |\tilde{B}T|/|BT|; \end{aligned}$$

upoštevajmo, da je B med B_1 in B_2, zato je $|BT| \le |B_2T|$:

$$\begin{aligned} &\ge |\tilde{A}T|/|AT| + |\tilde{B}T|/|B_2T| \\ &= (|AT| + |A\tilde{A}|)/|AT| + (|B_2T| - |\tilde{B}B_2|)/|B_2T| \\ &= 1 + |A\tilde{A}|/|AT| + 1 - |\tilde{B}B_2|/|B_2T|; \end{aligned}$$

upoštevajmo simetričnost na obeh poltrakih:

$$= 1 + |A\tilde{A}|/|AT| + 1 - |\tilde{A}A_2|/|A_2T|;$$

upoštevajmo, da je $A = A_1$ in da $\tilde{A}$ leži na pol poti med A_1 in A_2, zato je $|A\tilde{A}| = |\tilde{A}A_2|$:

$$= 2 + |A\tilde{A}|(1/|AT| - 1/|A_2T|).$$

To pa je večje od 2, kajti izraz v oklepajih na koncu je pozitiven (točka $A = A_1$ je namreč bližja T-ju kot točka A_2, zato je je $|AT| < |A_2T|$ in $1/|AT| > 1/|A_2T|$).

Ker smo po Makovem kriteriju dobili $m > 2$, ima f v točki T minimum, ne pa maksimuma. Ker to velja za vsak ekstrem T na območju, kjer α narašča, lahko zaključimo, da je na tem območju največ en ekstrem, kajti če bi bila vsaj dva, bi moral biti eden od njiju maksimum (saj vemo, da so ekstremi funkcije f izmenično minimumi in maksimumi). Ker po drugi strani tudi že vemo, da ima f na tem območju vsaj en ekstrem, lahko zdaj zaključimo, da ima tu *natanko en* ekstrem in da je to minimum.

O tem ekstremu lahko povemo še nekaj zanimivega. Če pogledamo še enkrat sliko 13, lahko razmišljamo takole: točko, kjer daljica AB seka poltrak TC, imenujmo R. Spomnimo se, da smo pri rešitvi s trisekcijo na koncu razdelka §4 opisali položaj točke T s parametrom λ, ki je povedal, kje na daljici AB leži njeno presečišče s TC; torej ni v našem primeru λ nič drugega kot $|AR|/|AB|$. To razmerje pa je $< 1/2$, o čemer se lahko prepričamo takole: pri premiku iz A v R se premaknemo po y-koordinati ravno za polovico razlike med y-koordinatama točk $A = A_1$ in B_1; če se nato iz R premaknemo naprej proti B še za nadaljnjih $|AR|$ enot, se bomo torej po y-koordinati premaknili še za eno polovico razlike med A in B_1 in bomo tako prišli ravno na y-koordinato točke B_1; točka B pa, ki leži naprej od B_1 na poltraku TB, ima višjo y-koordinato od točke B_1, torej točke B s tem premikom še nismo dosegli. Zato mora biti $|RB| > |AR|$ in zato $|AR|/|AB| = |AR|/(|AR| + |RB|) < 1/2$.

§8.4. Kdaj je ekstrem iz §8.3 tudi edini na loku $\mathcal{L}_1$? Našli smo torej ekstrem na območju, kjer α narašča; morda to pokrije že celoten lok $\mathcal{L}_1$ (to se zgodi, če je $\phi_B - \phi_A \leq \arccos|A|$) in tedaj je to tudi edini ekstrem na $\mathcal{L}_1$ sploh (in je to tudi globalni minimum funkcije f, ki nas v resnici zanima).

Razmislimo zdaj o primeru, ko je $\phi_B - \phi_A$ sicer večji od $\arccos|A|$, ni pa večji od $\arccos|A| + \arccos|B|$. Slednje pomeni, da začne β padati (pri $\tau = \phi_B - \arccos|B|$) prej kot α (ki začne padati pri $\tau = \phi_A + \arccos|A|$). Območju, kjer α narašča (in ki smo ga že obravnavali v §8.3), sledi tedaj območje, kjer oba kota, α in β, padata (in to traja do konca loka $\mathcal{L}_1$).

Recimo, da ima f na tem območju še neki ekstrem T. Situacija je še vedno taka kot na sliki 13, le s to razliko, da smo tu na območju, kjer α pada, torej je kót $\angle CAT$ oster, torej mora biti $A = A_2$ in ne $A = A_1$. Poleg tega smo tudi na območju, kjer pada β, torej je oster tudi kót $\angle CBT$, torej mora ležati B med $\tilde{B}$ in B_2. Izračunajmo spet Makov kriterij:

$$\begin{aligned} m &= \cos\alpha(1/|AT| + 1/|BT|) = \cos\alpha/|AT| + \cos\alpha/|BT| \\ &= |\tilde{A}T|/|AT| + |\tilde{B}T|/|BT|. \end{aligned}$$

Ker je $A = A_2$, leži A dlje od T kot točka $\tilde{A}$, torej $|\tilde{A}T| < |AT|$; in ker leži B med $\tilde{B}$ in B_2, je B bolj oddaljen od T kot točka $\tilde{B}$, torej $|\tilde{B}T| < |BT|$. V pravkar dobljenem izrazu za m sta torej oba ulomka manjša od 1, zato je $m < 2$ in v točki T ima funkcija f maksimum.

Toda spomimo se, da območje, ki ga gledamo (območje, kjer α in β padata), sega do konca loka $\mathcal{L}_1$, torej je zadnji ekstrem na loku $\mathcal{L}_1$ maksimum; toda loku $\mathcal{L}_1$ sledi lok $\mathcal{L}_2$, kjer f narašča, zato mora biti zadnji ekstrem na $\mathcal{L}_1$ minimum. Tako nas je torej predpostavka, da imamo na območju, kjer α in β padata, še kak ekstrem, pripeljala v protislovje; edini ekstrem na $\mathcal{L}_1$ je minimum iz §8.3 (torej z območja, kjer α raste).

§8.5. Kdaj sta na loku $\mathcal{L}_1$ dva minimuma in kateri je manjši? V §8.3 in §8.4 smo obdelali primere, ko je $\phi_B - \phi_A < \arccos|A| + \arccos|B|$. Ostane še primer, ko je $\mathcal{L}_1$ daljši od tega; to pomeni, da začne α padati prej kot β. Lok $\mathcal{L}_1$ lahko tedaj v mislih razdelimo na tri dele: $\mathcal{L}_{1a}$ od ϕ_A do $\phi_A + \arccos|A|$, nato $\mathcal{L}_{1b}$ od $\phi_A + \arccos|A|$ do $\phi_B - \arcsin|B|$ in končno $\mathcal{L}_{1c}$ od $\phi_B - \arcsin|B|$ do ϕ_B.

Za $\mathcal{L}_{1a}$ (kjer α narašča) smo že v §8.3 videli, da je tam natanko en ekstrem in to minimum.

Na $\mathcal{L}_{1b}$ kót α pada, β pa narašča, zato je lahko tam največ ena točka, kjer sta α in β enaki. Tu je torej največ en ekstrem, ta pa mora biti maksimum, ker je bil prejšnji ekstrem (na $\mathcal{L}_{1a}$) minimum.

Če je torej na $\mathcal{L}_1$ še kak minimum poleg tistega na $\mathcal{L}_{1a}$, se to lahko zgodi le na $\mathcal{L}_{1c}$. Tak minimum na $\mathcal{L}_{1c}$ je lahko največ eden, saj vemo, da so na $\mathcal{L}_1$ lahko največ trije ekstremi (to pa že dosežemo s tem, da imamo en minimum na $\mathcal{L}_{1a}$, enega na $\mathcal{L}_{1c}$ in nekje med njima še maksimum — slednji je lahko na $\mathcal{L}_{1b}$ ali pa na $\mathcal{L}_{1c}$, možno je oboje).

Recimo torej, da imamo na $\mathcal{L}_1$ dva minimuma, enega na $\mathcal{L}_{1a}$ in enega na $\mathcal{L}_{1c}$; kateri od njiju je manjši? Tisto bo globalni minimum, ki ga iščemo.

Mislimo si točko $\hat{B}$, ki ima enak polarni kót ϕ_B kot točka B, vendar je od središča krožnice oddaljena enako kot A, torej $|\hat{B}| = |A|$. Če bi reševali nalogo za točki A in $\hat{B}$ namesto za točki A in B, bi iskali minimume funkcije $\hat{f}(\tau) := f_A(\tau) + f_{\hat{B}}(\tau)$. Ker imata $\hat{B}$ in B enak polarni kot, je razdelitev na loke $\mathcal{L}_1, \ldots, \mathcal{L}_4$ enaka kot prej; in ker je $|\hat{B}| = |A|$, lahko uporabimo razmislek z začetka §8 in zaključimo, da ima $\hat{f}$ lokalni maksimum točno na sredi loka $\mathcal{L}_1$, pri polarnem kotu $\nu := (\phi_A + \phi_B)/2$, ter dva enakovredna minimuma, razporejena simetrično na vsaki polovici loka, pri polarnih kotih $\phi_A + \mu$ in $\phi_B - \mu$ za neki $\mu \in (0, (\phi_B - \phi_A)/2)$.

Razdelitev $\mathcal{L}_1$ na $\mathcal{L}_{1a}$, $\mathcal{L}_{1b}$ in $\mathcal{L}_{1c}$ je sicer pri nalogi z A in $\hat{B}$ malo drugačna kot pri nalogi z A in B: ker je $|\hat{B}| > |B|$, je $\arccos|\hat{B}| < \arccos|B|$, tako da je lok $\mathcal{L}_{1c}$ pri nalogi z A in $\hat{B}$ krajši kot pri nalogi z A in B, lok $\mathcal{L}_{1b}$ pa je zato daljši; lok $\mathcal{L}_{1a}$ pa je pri obeh enak.

Opazimo še, da naš razmislek iz §8.3 deluje tudi za primer, ko je $|B| = |A|$ (in ne le $|B| < |A|$), zato smemo za prvega od teh ekstremov funkcije $\hat{f}$ na $\mathcal{L}_{1a}$, torej za minimum na $\phi_A + \mu$, zaključiti, da leži na loku $\mathcal{L}_{1a}$.

Definirajmo zdaj $g(\tau) := f(\tau) - \hat{f}(\tau) = f_B(\tau) - f_{\hat{B}}(\tau) = |TB| - |T\hat{B}|$. Zasukajmo v mislih koordinatni sistem tako, da ležita B in $\hat{B}$ na pozitivnem delu x-osi: $\phi_B = \phi_{\hat{B}} = 0$, $B = (|B|, 0)$ in $\hat{B} = (|\hat{B}|, 0)$. Uporabimo to v enačbi (3), pa dobimo

$f'_B(\tau) = |B| \sin\tau / |BT|$ in podobno za $\hat{B}$. Potem je

$$g'(\tau) = f'_B(\tau) - f'_{\hat{B}}(\tau) = \sin\tau \cdot (|B|/|BT| - |\hat{B}|/|\hat{B}T|).$$

Zanimali nas bodo primeri, ko leži T na $\mathcal{L}_1$, torej je τ med $\phi_B - 180^\circ$ in ϕ_B (pravzaprav to pokrije celo še $\mathcal{L}_4$, ne le $\mathcal{L}_1$); ker smo zasukali koordinatni sistem in je zdaj $\phi_B = 0$, to pomeni, da bo τ med -180° in 0°, zato bo $\sin\tau < 0$. Kaj pa drugi faktor v g', torej $|B|/|BT| - |\hat{B}|/|\hat{B}T|$? Ali je mogoče, da bi bil kdaj večji ali enak 0? Iz tega bi sledilo:

$$|B|/|BT| \geq |\hat{B}|/|\hat{B}T|$$
$$|B| \cdot |\hat{B}T| \geq |\hat{B}| \cdot |BT|;$$

upoštevajmo, da je $|BT| = \sqrt{(|B| - x_T)^2 + y_T^2} = \sqrt{|B|^2 - 2|B|\cos\tau + 1}$ in podobno pri $|\hat{B}T|$:

$$|B|\sqrt{1 + |\hat{B}|^2 - 2|\hat{B}|\cos\tau} \geq |\hat{B}|\sqrt{1 + |B|^2 - 2|B|\cos\tau};$$

kvadrirajmo obe strani:

$$|B|^2(1 + |\hat{B}|^2 - 2|\hat{B}|\cos\tau) \geq |\hat{B}|^2(1 + |B|^2 - 2|B|\cos\tau)$$
$$|B|^2 + |B|^2|\hat{B}|^2 - 2|B|^2|\hat{B}|\cos\tau \geq |\hat{B}|^2 + |\hat{B}|^2|B|^2 - 2|\hat{B}|^2|B|\cos\tau$$
$$|B|^2 - 2|B|^2|\hat{B}|\cos\tau \geq |\hat{B}|^2 - 2|\hat{B}|^2|B|\cos\tau$$
$$|B|^2 - |\hat{B}|^2 + 2|\hat{B}|^2|B|\cos\tau - 2|B|^2|\hat{B}|\cos\tau \geq 0$$
$$(|B| - |\hat{B}|)(|B| + |\hat{B}|) - 2\,|\hat{B}|\,|B|\cos\tau(|B| - |\hat{B}|) \geq 0;$$

ker je $|\hat{B}| > |B|$, lahko delimo z $|B| - |\hat{B}|$ in se neenačaj obrne:

$$|B| + |\hat{B}| - 2\,|\hat{B}|\,|B|\cos\tau \leq 0;$$
$$|B|(1 - |\hat{B}|\cos\tau) + |\hat{B}|(1 - |B|\cos\tau) \leq 0.$$

Ker je $\cos\tau \leq 1$ in $0 < |B| < 1$, je tudi $|B|\cos\tau < 1$, zato pa $1 - |B|\cos\tau > 0$; podobno bi dobili tudi $1 - |\hat{B}|\cos\tau > 0$. V zgoraj dobljeni neenačbi imamo na levi strani torej vsoto dveh seštevancev, vsak seštevanec pa je produkt dveh pozitivnih faktorjev; zato je tudi vsota pozitivna, neenačba pa zatrjuje, da je manjša ali enaka 0. Tako nas je torej predpostavka, da je drugi faktor v g' kdaj večji ali enak 0, pripeljala v protislovje; torej je ta faktor v resnici vedno < 0; zato je g' zmnožek dveh negativnih faktorjev (prvi je $\sin\tau$, za katerega smo že videli, da je vedno < 0), torej je vedno pozitiven. Zaključimo lahko torej, da je $g(\tau)$ naraščajoča funkcija kota τ.

Funkcijo g smo definirali kot razliko med f in $\hat{f}$, torej je $f(\tau) = \hat{f}(\tau) + g(\tau)$. Funkcijo f smo torej dobili tako, da smo funkciji $\hat{f}$ prišteli naraščajočo funkcijo g; to pomeni, da če je $\hat{f}$ na nekem intervalu naraščala, bo na njem naraščala tudi f. Spomnimo se, da je imela $\hat{f}$ minimuma na $\phi_A + \mu$ in $\phi_B - \mu$ in maksimum na ν, torej je naraščala na $[\phi_A + \mu, \nu]$ in na $[\phi_B - \mu, \phi_B]$; tam torej narašča tudi f, zato na teh dveh intervalih (razen morda v krajiščih) ne more imeti ekstremov.

Ekstremi funkcije f (na loku $\mathcal{L}_1$) lahko ležijo torej le na $[\phi_A, \phi_A + \mu]$ in $[\nu, \phi_B - \mu]$. Spomnimo se, da $\phi_A + \mu$ leži na $\mathcal{L}_{1a}$, zato je interval $[\phi_A, \phi_A + \mu]$ tudi v celoti znotraj

$\mathcal{L}_{1a}$; po drugi strani pa interval $[\nu, \phi_B - \mu]$, ki se začne šele na sredi loka $\mathcal{L}_1$, gotovo nima ničesar skupnega z lokom $\mathcal{L}_{1a}$, kajti ta pokriva manj kot polovico loka $\mathcal{L}_1$. (Spomnimo se namreč, da je $\mathcal{L}_{1a}$ dolg $\arccos|A|$, lok $\mathcal{L}_{1c}$ pa je dolg $\arccos|B|$, kar je — ker je $|B| < |A|$ — daljše od loka $\mathcal{L}_{1a}$; če bi torej že $\mathcal{L}_{1a}$ pokrival vsaj polovico loka $\mathcal{L}_1$, bi bila $\mathcal{L}_{1a}$ in $\mathcal{L}_{1c}$ skupaj že daljša od $\mathcal{L}_1$, to pa bi bilo protislovje.)

Ker vemo, da ima f na $\mathcal{L}_{1a}$ natanko en ekstrem (in sicer minimum), zdaj vidimo, da mora ta minimum ležati na $[\phi_A, \phi_A + \mu]$; morebitni drugi ekstremi te funkcije pa morajo potemtakem ležati na $[\nu, \phi_B - \mu]$. Tam je torej drugi minimum (če ga funkcija ima) in tudi lokalni maksimum med obema minimuma (zapomnimo si torej, da ta lokalni maksimum leži na drugi polovici loka $\mathcal{L}_1$). Če imamo dva minimuma, recimo prvemu τ_1, drugemu pa τ_2. Ker je τ_1 minimum funkcije f na $[\phi_A, \phi_A + \mu]$, je njena vrednost na koncu tega intervala vsaj tolikšna kot v τ_1:

$$f(\tau_1) \le f(\phi_A + \mu) = \hat{f}(\phi_A + \mu) + g(\phi_A + \mu)$$

upoštevajmo, da v $\phi_A + \mu$ minimum funkcije $\hat{f}$ po celem $\mathcal{L}_1$:

$$\le \hat{f}(\tau_2) + g(\phi_A + \mu)$$

upoštevajmo, da je $\phi_A + \mu < \nu \le \tau_2$ in da je g naraščajoča funkcija:

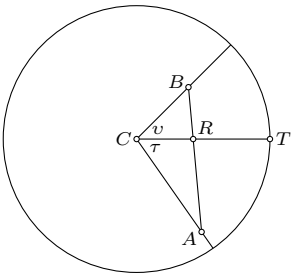
$$< \hat{f}(\tau_2) + g(\tau_2) = f(\tau_2).$$

Vidimo torej, da ima f manjšo vrednost v τ_1 kot v τ_2; v τ_1 je torej njen globalni minimum, medtem ko je τ_2 le lokalni minimum.

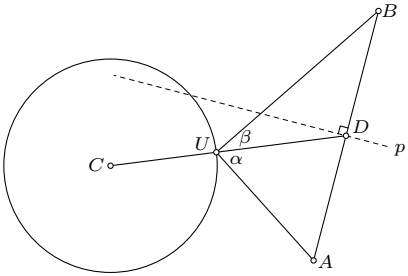
§8.6. Zaključek. Zdaj imamo vse, kar potrebujemo, da s trisekcijo rešimo tudi težjo različico naše naloge. Videli smo (§8.3), da na tistem delu loka $\mathcal{L}_1$, kjer α narašča, gotovo leži natanko en ekstrem in da je to minimum; da ima ta minimum, če ga opišemo s parametrom λ kot v naši rešitvi prvotne naloge v §4, gotovo $\lambda \le 1/2$; in da kasneje na $\mathcal{L}_1$ bodisi ni nobenega minimuma več (§8.4) bodisi je še eden, vendar je slabši od tistega prvega (§8.5).

Videli smo tudi (§8.5), da če imamo na $\mathcal{L}_1$ dva minimuma, potem lokalni maksimum med tema dvema minimumoma leži na drugi polovici loka, torej na $[(\phi_A + \phi_B)/2, \phi_B]$. Z razmislekom, zelo podobnim tistemu s konca §8.3, se lahko prepričamo, da v maksimumu velja $\lambda \ge 1/2$. Naj bo T poljubna točka na drugi polovici loka $\mathcal{L}_1$; poltrak CT torej s poltrakom CA oklepa kót τ, s CB pa kót $\upsilon := (\phi_B - \phi_A) - \tau$, pri čemer je $\upsilon \le \tau$. Zasukajmo koordinatni sistem tako, da leži T na pozitivnem delu x-osi (slika 14). Presečišču daljice AB s poltrakom CT recimo R. Točka B ima y-koordinato $y_B = |B| \sin \upsilon$, točka A pa $y_A = -|A| \sin \tau$; ker je $\upsilon \le \tau$, je $\sin \upsilon \le \sin \tau$, kar nam skupaj z $|B| \le |A|$ dá $y_B \le |y_A|$. Ko se iz B premaknemo v R (ki je na poltraku TC in ima zato y-koordinato 0), se nam je y zmanjšal za y_B; če se še enkrat premaknemo (po daljici AB) enako daleč, se nam bo y še enkrat zmanjšal za y_B; da pa dosežemo točko A, se nam mora y zmanjšati za $|y_A|$, kar je $\ge y_B$; torej mora biti ta drugi premik dolg *vsaj* toliko kot prvi. Tako smo dobili $|AR| \ge |BR|$, torej $\lambda = |AR|/|AB| = |AR|/(|AR| + |BR|) \ge |AR|/(|AR| + |AR|) = 1/2$:

Vidimo torej, da globalni minimum najdemo pri $\lambda \le 1/2$; in da če obstaja še drugi, slabši (lokalni) minimum, se le-ta, skupaj z maksimumom med obema minimumoma, nahaja na $\lambda \ge 1/2$. Na $\lambda \in [0, 1/2]$ je torej f unimodalna in lahko globalni minimum poiščemo s trisekcijo po tem intervalu.



Slika 14.



Slika 15.

Doslej smo ves čas delali s predpostavko, da je $|A| \geq |B|$. V splošnem je lahko tudi obratno; rešitev iz §4 lahko zelo elegantno popravimo v rešitev težje različice naloge tako, da spremenimo le prvo vrstico:

prej: $\lambda_L := 0$; $\lambda_D := 1$; $f^\star := \infty$;
potem: **if** $|1 - |A|\,| \leq |1 - |B|\,|$ **then** $\lambda_L := 0$ **else** $\lambda_L := 1/2$;
$\lambda_D := \lambda_L + 1/2$; $f^\star := \infty$;

Pogoj v stavku **if** torej preverja, ali leži točka A bližje krožnici (ali enako daleč od nje) kot točka B. Zakaj ga nismo zapisali preprosto kot „**if** $|A| \geq |B|$“? Zato, ker v gornji obliki ta rešitev deluje tudi za prvotno (lažjo) različico naloge, pri kateri sta točki A in B zunaj krožnice namesto znotraj nje. Tudi tam torej ni treba pregledati celega območja $0 \leq \lambda \leq 1$ (kot smo naredili v §4), ampak ga je dovolj preiskati le polovico in to isto polovico kot za težjo različico naloge (s točkama A in B znotraj krožnice). Prepričajmo se, da to drži.

§8.7. Nova rešitev s trisekcijo deluje tudi, če sta A in B zunaj krožnice. Recimo torej, da točki A in B ležita zunaj krožnice, pri čemer je A bližja krožnici (ali enako daleč od nje); velja torej $|A| \leq |B|$. Razpolovišče daljice AB imenujmo D; presečišče poltraka CD s krožnico je točka $U = D/|D|$. V tej točki imamo vpadni kot $\alpha = \angle DUA$ in odbojni kot $\beta = \angle DUB$ (slika 15).

Primer, ko je $|A| = |B|$, lahko obravnavamo posebej; takrat je f simetrična glede na razpolovišče loka $\mathcal{L}_1$, prav v tem razpolovišču zato leži tudi njen minimum in naš postopek s trisekcijo bi ga našel pri $\lambda = 1/2$; ker je ta vrednost vsekakor del začetnega intervala $[\lambda_L, \lambda_D]$, minimuma ne bomo spregledali. Odslej predpostavimo, da je $|A| < |B|$.

Naj bo p pravokotnica na AB skozi točko D (črtkana črta na sliki 15). Ta premica nam razdeli ravnino: točkam, ki ležijo na isti strani p-ja kot A, je točka A bližja kot B, za tiste na drugi strani pa je ravno obratno. Za daljico CD vidimo, da eno krajišče (namreč D) leži na p, za drugo krajišče (namreč C) pa velja, da je bližje A-ju kot B-ju, saj smo predpostavili, da je $|A| < |B|$. Torej leži C na isti strani p-ja kot A; in zato leži na tej strani p-ja tudi cela daljica CD (razen točke D, ki leži prav na premici p). Zato pa za vse točke na daljici CD velja, da jim je A bližja kot B (razen za D, ki sta mu obe enako daleč); tudi za točko U: velja torej $|UA| < |UB|$.

Trikotnika $\triangle DUA$ in $\triangle DUB$ imata skupno stranico DU; če merimo njuno višino pravokotno na to stranico, dobimo pri prvem $|DA| \sin \angle UDA$, pri drugem pa

$|DB|\sin\angle UDB$; toda $|DA| = |DB|$, ker je D ravno razpolovišče daljice AD; in kota $\angle UDA$ in $\angle UDB$ skupaj tvorita iztegnjeni kot 180°, zato sta njuna sinusa enaka; zato sta višini obeh trikotnikov enaki; ker smo višino računali pri obeh pravokotno na stranico enake dolžine (namreč stranico DU), pa imata oba trikotnika enako tudi ploščino.

Po drugi strani, če v trikotniku $\triangle DUA$ računamo višino v smeri pravokotno na stranico UA, dobimo $|UD|\sin\alpha$; ploščina tega trikotnika je torej $|UA|\cdot|UD|\sin\alpha$. Podobno v trikotniku $\triangle DUB$ dobimo na stranico UB višino $|UD|\sin\beta$, zato pa ploščino $|UB|\cdot|UD|\sin\beta$. Ker vemo, da sta obe ploščini enaki, sledi $|UA|\sin\alpha = |UB|\sin\beta$ oz. $\sin\alpha/\sin\beta = |UB|/|UA|$; to pa je naprej > 1, saj smo malo prej videli, da je $|UA| < |UB|$. Velja torej $\sin\alpha/\sin\beta > 1$, torej $\sin\alpha > \sin\beta$, torej v točki U funkcija f narašča (spomnimo se, da je $f' = \sin\alpha - \sin\beta$). Na začetku loka $\mathcal{L}_1$ pa je f padala, saj je takrat $\alpha = 0$ in $\beta > 0$ (pa tudi po celotnem loku $\mathcal{L}_4$ pred tem je f padala). Nekje med začetkom loka $\mathcal{L}_1$ in točko U mora torej f preiti iz padanja v naraščanje, tam pa ima minimum. Ta minimum je tudi globalni, saj že iz §4 vemo, da ima f na $\mathcal{L}_1$ le en ekstrem (ki je globalni minimum). Ker naš novi postopek s trisekcijo preišče območje $0 \le \lambda \le 1/2$ (to pa je ravno od začetka $\mathcal{L}_1$ do točke U), bomo tisti minimum tudi našli.

§9. Rešitve z grobo silo pri težji različici naloge. Dejstvo, da imamo na $\mathcal{L}_1$ lahko dva minimuma funkcije f — globalnega na $\mathcal{L}_{1a}$ in še enega zgolj lokalnega na $\mathcal{L}_{1c}$ — lahko naši rešitvi z grobo silo zavede. Za začetek si oglejmo neugoden primer za rešitev iz §5. Vzemimo enotsko krožnico s središčem C v koordinatnem izhodišču; in vzemimo točki A in B, enako oddaljeni od C, recimo $|CA| = |CB| = \rho < 1$. Če sta A in B dovolj blizu krožnice, imamo dva (enakovredna) globalna minimuma, enega na $[\phi_A, \phi_A + \arccos\rho]$ in enega na $[\phi_B - \arccos\rho, \phi_B]$. Če počasi povečujemo ρ proti 1, se A in B približujeta krožnici, minimuma pa se razmikata in se počasi približujeta kotoma ϕ_A in ϕ_B (in $\arccos\rho$ pada proti 0). Spomnimo se, da so točke, ki jih naša rešitev iz §5 pregleda v prvi iteraciji svoje glavne zanke, po $\triangle := 2\pi/n$ radianov narazen. Nastavimo naš ρ tako, da bo razmik med obema minimumoma oblike $(k+\frac{1}{2})\triangle$ za neki celoštevilski k. Nato zavrtimo točki A in B okrog koordinatnega izhodišča za toliko, da dobi eden od ekstremov kót 0, drugi pa $(k+\frac{1}{2})\triangle$. Postopek iz §5 bo torej v prvi iteraciji glavne zanke opazil prvi ekstrem v točki τ_0, drugega pa ne bo opazil, ker leži ravno na pol poti med točkama τ_k in τ_{k+1}. Če zdaj eno od točk A in B premaknemo še malo bližje krožnici, bo „njen“ minimum postal boljši od drugega; tako lahko zagotovimo, da bo pravi globalni minimum ravno tisti, ki ga bo naš postopek iz §5 zgrešil.

Ta konstrukcija je, kot smo videli, delovala za poljuben n; če smo v §5 rekli, da je dovolj že $n = 3$, pa zdaj vidimo, da je pri težji različici naloge mogoče za poljuben n najti primer, kjer vrne naš postopek napačno rešitev. V praksi nas sicer pri sestavljanju neugodnih testnih primerov nekoliko omejuje dejstvo, da naloga zagotavlja, da bodo koordinate točk A, B in C ter radij krožnice cela števila od -1000 do 1000; pri premikanju in sukanju točk v prejšnjem odstavku pa ni nujno, da bomo na koncu dobili celoštevilske koordinate. Vseeno pa nam je pri naših poskusih uspelo brez velikih težav sestaviti protiprimere za vse n do 10^5; pri tako velikih n pa ima rešitev težave že tudi s časovno omejitvijo.

Tudi rešitev iz §6 ima pri težji različici naloge težave. Ko glavna zanka tiste

rešitve računa $f(\tau_k)$ za $k = 0, \ldots, n$, se lahko zgodi, da ima med točkama τ_k in τ_{k+1} funkcija f dva ekstrema; tedaj bo odvod v obeh točkah enako predznačen in naš postopek bo tadva ekstrema spregledal; eden od teh dveh ekstremov pa je neizogibno minimum — morda celo globalni. Podobno, če ima f med τ_k in τ_{k+1} tri ekstreme, sta med njimi dva minimuma, en globalni in eden zgolj lokalni, naš postopek pa bo z bisekcijo našel le enega od njiju, ne nujno globalnega. Vzemimo torej spet točki A in B na oddaljenosti ρ od koordinatnega izhodišča, njuna kota ϕ_A in ϕ_B pa naj si bosta za manj kot $2\pi/n$ radianov narazen; ρ naj bo dovolj blizu 1, da še vseeno dobimo dva minimuma in ne enega. Eno od točk A in B premaknimo malo bližje krožnici, da oba minimuma ne bosta čisto enaka. Koordinatni sistem nato zasukajmo tako, da prideta (za neki k) globalni minimum in lokalni maksimum med τ_k in τ_{k+1}, drugi (zgolj lokalni) minimum pa je v enem od sosednjih intervalov (pred τ_k ali za τ_{k+1}). Naš postopek bo takrat spregledal globalni minimum in vrnil lokalni minimum iz sosednjega intervala.

Vidimo torej, da lahko načeloma tudi za rešitev iz §6 pri vsakem n sestavimo testni primer, ki ga reši narobe; v praksi pa nas pri tem kar precej omejuje dejstvo, da morajo biti koordinate pri tej nalogi cela števila do ± 1000, zato se kota ϕ_A in ϕ_B ne moreta razlikovati za poljubno malo (če nočemo, da sta povsem enaka) in točki tudi ne moreta biti poljubno blizu krožnice (biti pa ji morata zelo blizu, če hočemo dobiti dva minimuma in ne le enega).[33] Pri naših poskusih nam je uspelo najti takšne primere za do vključno $n = 1051$, za $n = 1052$ pa ga kljub precej iskanja nismo našli. S časovno omejitvijo pri takšnih n še ni težav, tudi do približno $n = 10\,000$ bi še šlo.

Obe rešitvi (tisto iz §5 in tisto iz §6) bi lahko seveda spremenili tako, da bi namesto cele krožnice pregledali le lok $\mathcal{L}_{1a}$, kjer f nima drugih ekstremov razen globalnega minimuma; toda s tem bi izgubili glavno prednost teh dveh rešitev, namreč da nam zanju ni bilo treba razmišljati o tem, na katerem delu krožnice leži globalni minimum.

Rešitev iz §5 lahko popravimo tudi tako, da sicer še vedno pregledamo celo krožnico, vendar namesto enega samega intervala $[\tau_L, \tau_D]$ (ki ga v vsaki iteraciji glavne zanke zožimo) vzdržujemo več intervalov; v vsaki iteraciji glavne zanke razbijemo vse dosedanje intervale na manjše podintervale in zavržemo tiste izmed njih, za katere lahko dokažemo, da funkcija f na njih ne doseže minimuma. Recimo, da gledamo interval $[\tau_1, \tau_2]$; in recimo, da poznamo neko spodnjo in zgornjo mejo odvoda f' po tem intervalu, torej $f'_{\min}$ in $f'_{\max}$, pri katerih za vsako $\tau \in [\tau_1, \tau_2]$ velja $f'_{\min} \le f'(\tau) \le f'_{\max}$. Potem vemo, da na našem intervalu funkcija f ves čas narašča ali pada s smernim koeficientom vsaj $f'_{\min}$ in kvečjemu $f'_{\max}$. Če se iz točke τ_1 premaknemo za $\triangle$ v desno, se torej vrednost funkcije f poveča za vsaj $f'_{\min} \cdot \triangle$; če pa se iz τ_2 premaknemo za $\triangle$ v levo, se vrednost funkcije f zmanjša za kvečjemu $f'_{\max} \cdot \triangle$. Vidimo torej, da za vsako $\tau \in [\tau_1, \tau_2]$ velja

$$f(\tau_1) + (\tau - \tau_1) f'_{\min} \le f(\tau) \text{ in } f(\tau_2) - (\tau_2 - \tau) f'_{\max} \le f(\tau).$$

[33] Najmanjšo razliko $\phi_A - \phi_B$, če naj bosta točki še vedno znotraj krožnice s polmerom $r = 1000$ in središčem $(0, 0)$, dobimo, če vzamemo $A(1, 999)$ in $B(1, 998)$, ko je $\phi_A - \phi_B \approx 10^{-6}$ radianov oz. pribl. 0,2 kotne sekunde. Najmanjšo razdaljo med točko (s celoštevilskimi koordinatami do 1000) in krožnico (s celoštevilskim polmerom), znotraj katere ta točka leži, pa dobimo pri $(44, 968)$ in $(616, 748)$, ki ležita znotraj krožnice s središčem $(0, 0)$ in polmerom 969, od nje pa sta oddaljeni le za približno $5{,}16 \cdot 10^{-4}$ enot.

Tako smo torej f na intervalu $[\tau_1, \tau_2]$ omejili od spodaj z maksimumom dveh linearnih funkcij; ni težko izračunati minimuma te spodnje meje, to pa je potem tudi spodnja meja za vrednost funkcije f na tem intervalu.

podprogram DodajInterval(τ_1, τ_2, I, $f^\star$):
  vhodni podatki: interval $[\tau_1, \tau_2]$; množica intervalov I; $f^\star$ je najmanjša doslej znana vrednost f in se prenaša po referenci;
  izračunaj $f(\tau_1)$, $f(\tau_2)$;
  $f^\star := \min\{f^\star, f(\tau_1), f(\tau_2)\}$;
  izračunaj spodnjo mejo L vrednosti $f(\tau)$ za $\tau \in [\tau_1, \tau_2]$;
  if $L < f^\star$ **then** dodaj $[\tau_1, \tau_2]$ v množico I;

glavni blok programa:
  $f^\star :=$ vrednost $f(\tau)$ v poljubni τ;
  $I :=$ prazna množica;
  $\triangle_\tau := (2\pi)/n_1$;
  for $k := 0$ **to** $n_1 - 1$:
    DodajInterval($k\triangle_\tau$, $(k+1)\triangle_\tau$, I, $f^\star$);
  while true:
    $I' :=$ prazna množica;
    za vsak interval $[\tau_1, \tau_2] \in I$:
      $\triangle_\tau := (\tau_2 - \tau_1)/n_2$;
      for $k := 0$ **to** $n_2 - 1$:
        DodajInterval($\tau_1 + k\triangle_\tau$, $\tau_1 + (k+1)\triangle_\tau$, I', $f^\star$);
    if so spodnje meje (L) vseh intervalov v I' večje od $f^\star - \varepsilon$ **then break**;
    $I := I'$;
  return $f^\star$;

Na začetku torej razdelimo krožnico na n_1 enakih delov, kasneje pa vsak interval razdelimo vsakič na n_2 enakih podintervalov. Ustavimo se, ko nobeden od intervalov ne obeta več izboljšave v primerjavi z najmanjšo doslej znano vrednostjo funkcije, torej $f^\star$. Lahko bi dodali še en ustavitveni pogoj: ustavili bi se, ko postanejo intervali v I dovolj kratki.

Nismo pa še povedali, kako za dani $[\tau_1, \tau_2]$ izračunati spodnjo in zgornjo mejo funkcije f' na tem intervalu. Spomnimo se (§1), da je $f'(\tau) = f'_A(\tau) + f'_B(\tau)$, torej lahko izračunamo spodnjo in zgornjo mejo posebej za f_A in f_B in ju nato seštejemo. Za f_A in f_B pa se spomnimo formule $f'_U(\tau) = (x_u \sin\tau - y_U \cos\tau)/f_U(\tau)$. Spodnjo in zgornjo mejo tega količnika dobimo s pomočjo spodnje in zgornje meje števca in imenovalca. Za meje števca upoštevamo, da dosežeta $\sin\tau$ in $\cos\tau$ svoj minimum in maksimum bodisi na krajiščih intervala, torej τ_1 in τ_2, bodisi pri kotih $\pm\pi/2$ (za sinus) oz. 0 in π (za kosinus), kjer imata ekstreme. Za meje imenovalca pa lahko loku, ki ga na krožnici tvorijo točke s polarnimi koti $\tau \in [\tau_1, \tau_2]$, očrtamo pravokotnik (*bounding box*) in vzamemo razdaljo od U do njemu najbližje oz. najbolj oddaljene točke tega pravokotnika.

Ta postopek bo gotovo našel pravo rešitev, ni pa najbolj očitno, koliko intervalov bo moral pri tem pregledati in kako dolgo se bo s tem zamudil. Pri naših poskusih z $n_1 = 10$, $n_2 = 5$ je moral pri večini testnih primerov pregledati morda kakšnih sto

ali dvesto intervalov, v najbolj ekstremnih primerih pa do 26 000. Vsekakor je bila ta rešitev čisto dovolj hitra tudi za časovno omejitev na našem tekmovanju.

H. Kadrovska služba

Ob primeru v besedilu naloge nam lahko pride na misel, da bi vsakemu zaposlenemu pripisali zaporedno številko od 0 do $n-1$ (glede na položaj v seznamu, ki ga izpišemo ob kodiranju oz. ga dobimo kot vhod v dekodiranju), hierarhično strukturo pa bi predstavili tako, da bi za vsakega zaposlenega navedli številko njegovega šefa. Toda za predstavitev števila od 0 do $n-1$ bi potrebovali $\lceil \log_2 n \rceil$ bitov, za vse skupaj torej približno $n \log_2 n$; pri $n = 600$ to pomeni 10 bitov na zaposlenega, skupaj 6000, mi pa smemo porabiti le 2048 bitov. Potrebujemo torej kak varčnejši pristop.

Vhodne podatke, ki jih dobimo za kodiranje, lahko predelamo v drevo, ki ima po eno vozlišče za vsakega zaposlenega; otroci vozlišča, ki predstavlja nekega šefa, so vozlišča zaposlenih, ki predstavljajo neposredno podrejene tistega šefa. Vrstni red otrok naj bo tak kot vrstni red podrejenih v vhodnih podatkih. V korenu drevesa je direktor.

Seznam imen, ki ga moramo izpisati pri kodiranju, lahko dobimo tako, da vozlišča drevesa pregledamo v nekem izbranem vrstnem redu, npr. s premim obhodom (*preorder traversal*), in v tem vrstnem redu tudi izpisujemo imena v njih. Naloga binarnega niza B, ki ga moramo izpisati na koncu, pa je, da predstavi strukturo drevesa. Ob dekodiranju bomo morali potem znati iz niza B rekonstruirati drevo, nato pa bomo s premim obhodom vpisali v vozlišča tudi imena zaposlenih iz seznama, ki smo ga pripravili ob kodiranju.

Po drevesu se lahko sistematično sprehodimo takole: začnemo v korenu, nato pa za vsakega od njegovih otrok (od leve proti desni) ponovimo naslednje: premaknemo se v tega otroka, prehodimo celotno njegovo poddrevo (z rekurzivnim klicem) in se nato premaknemo nazaj v starša. Ob vsakem premiku si zapišimo en bit, ki pove, ali je šlo za premik gor ali dol. Naše drevo ima n vozlišč, zato ima $n-1$ povezav; in ker smo vsako povezavo prehodili dvakrat (prvič v smeri dol in kasneje spet v smeri gor), je bilo premikov vsega skupaj $2n-2$. Tako dobljeni niz je torej dovolj kratek (največ 1198 bitov), da ga lahko uporabimo kot B pri našem kodiranju.

Ko pri tem sprehajanju po drevesu prvič pridemo v neko vozlišče (to je takrat, ko vanj pridemo s korakom dol namesto s korakom gor; izjema je koren, kjer stojimo že na začetku), lahko tudi dodamo ime tega vozlišča v seznam imen, ki ga bomo na koncu skupaj z nizom B izpisali kot rezultat kodiranja — tako bo nastal ravno seznam v premem vrstnem redu, kakršnega smo si želeli. Zapišimo to rešitev s psevdokodo (za vse parametre si mislimo, da se prenašajo po referenci):

podprogram KodirajPoddrevo(vozlišče u, seznam L, niz B):
  dodaj ime vozlišča u na konec seznama L;
  za vsakega u-jevega otroka v od leve proti desni:
    dodaj znak `0` na konec niza B;
    KodirajPodddrevo(v, L, B);
    dodaj znak `1` na konec niza B;

podprogram Kodiraj(koren drevesa u):
  $L :=$ prazen seznam; $B :=$ prazen niz;

```
  KodirajPoddrevo(u, L, B);
  izpiši L in B;

podprogram DekodirajPoddrevo(seznam L, indeks ℓ, niz B, indeks b):
  u := novo vozlišče z imenom L[ℓ]; ℓ := ℓ + 1;
  while B[b] = 0:
    b := b + 1; v := DekodirajPoddrevo(L, ℓ, B, b);
    dodaj v-ju na konec seznama u-jevih otrok;
  b := b + 1; return u;

podprogram Dekodiraj(seznam imen L, niz B):
  ℓ := 0; b := 0; (* začnemo na začetku L-ja in B-ja *)
  u := DekodirajPoddrevo(L, ℓ, B, b);
  vrni drevo s korenom u;
```

Še en način, kako priti do primernega niza B, pa je naslednji: strukturo drevesa lahko opišemo tako, da gremo po vozliščih v premem vrstnem redu in pri vsakem izpišemo, koliko otrok ima; tako dobljeni seznam n celih števil opisuje strukturo drevesa. Da ga predelamo v (dovolj kratek) binarni niz B, lahko vsako število zapišemo v neke vrste eniškem zapisu: če ima trenutno vozlišče k otrok, dodamo na konec niza B najprej k ničel in nato še eno enico; to zagotavlja, da bo mogoče niz brez dvoumnosti tudi dekodirati. Tako dobljen niz B vsebuje toliko ničel, kolikor je skupno število otrok po vseh vozliščih, to pa je $n-1$, saj je vsako vozlišče razen korena otrok enega drugega vozlišča; poleg tega pa vsebuje B še n enic, po eno za vsako vozlišče. Toda ker se tako definiran B v vsakem primeru konča na enico, nam zadnja enica ne pove ničesar koristnega in jo lahko pobrišemo; dobimo niz dolžine $2n-2$, enako kot pri prejšnji rešitvi.

Zanimivo vprašanje je, ali bi se dalo nalogo rešiti s še krajšimi nizi B. Če bi v nizu B iz naše prve rešitve namesto znakov `0` in `1` uporabili `(` in `)`, bi videli, da je B pravilno gnezden oklepajski izraz iz $n-1$ parov oklepajev in zaklepajev. Vsak tak oklepajski izraz predstavlja neki veljaven opis strukture drevesa z n vozlišči. Takih dreves je torej prav toliko kot oklepajskih izrazov. In če po drugi strani namesto `(` in `)` uporabimo znaka `\` in `/`, si lahko tak niz predstavljamo kot opis cikcakaste poti po karirasti mreži, pri čemer `\` predstavlja premik za eno enoto desno in dol, `/` pa za eno enoto desno in gor. Če se taka pot začne v $(0,0)$, se konča v $(2n-2,0)$, kajti vseh premikov je $2n-2$, od tega polovica gor in polovica dol. Takih zaporedij premikov je $\binom{2n-2}{n-1}$, kajti med $2n-2$ koraki si lahko na toliko načinov izberemo, katerih $n-1$ bo šlo gor namesto dol. Ne ustreza pa vsako tako zaporedje premikov veljavnemu oklepajskemu izrazu, ker pri nekaterih premiki obeh tipov niso pravilno gnezdeni. Prepoznamo jih po tem, da takšna pot ne ostane ves čas na koordinatah $y \le 0$, pač pa kdaj doseže tudi $y = 1$ ali še višjo. Če poiščemo na poti prvo točko oblike $(x,1)$ in preostanek te poti prezrcalimo čez premico $y = 1$, se bo pot namesto na višini $y = 0$ končala na višini $y = 2$. Problematičnih poti od $(0,0)$ do $(2n-2,0)$ — se pravi takih, ki se kdaj povzpnejo nad višino $y = 0$ — je torej ravno toliko kot vseh poti od $(0,0)$ do $(2n-2,2)$; vsako táko pot pa sestavlja n korakov gor in $n-2$ dol, zato si lahko na $\binom{2n-2}{n}$ načinov izberemo, kje bodo koraki gor. Primernih poti od $(0,0)$ do $(2n-2,0)$ je torej $a_n := \binom{2n-2}{n-1} - \binom{2n-2}{n} =$

$\binom{2n-2}{n-1}/n = (2n-2)!/\,(n!(n-1)!)$ (mimogrede, tem številom pravimo Catalanova števila); toliko je zato tudi oklepajskih izrazov s pravilno gnezdenimi $n-1$ pari oklepajev in zaklepajev, toliko pa je tudi dreves z n vozlišči. Nemogoče je torej, da bi bili binarni nizi B za vsa ta drevesa krajši od $\log_2 a_n$, ker takih nizov preprosto ni dovolj. Če v a_n upoštevamo Stirlingovo formulo, $k! \approx k^k e^{-k}\sqrt{2\pi k}$, dobimo $a_n \approx 2^{2n-2}/\sqrt{n^3\pi}$, zato pa $\log_2 a_n \approx (2n-2) - \frac{3}{2}\log_2 n - \frac{1}{2}\log_2 \pi$. Naša rešitev z nizom dolžine $2n-2$ je torej le za $O(\log n)$ bitov slabša od te teoretične spodnje meje.

I. Interaktivna rekonstrukcija

Če podamo poizedbo iz samih enic, bomo kot odgovor za vsako vozlišče u našega drevesa dobili ravno število njegovih sosedov, torej stopnjo tega vozlišča; recimo ji d_u. Zdaj torej vemo, katera vozlišča so listi in katera so notranja vozlišča: listi imajo stopnjo 1, notranja vozlišča pa več kot 1. Nato bi lahko ponavljali naslednji postopek:

dokler drevo ni prazno:
  naj bo u poljuben list, torej vozlišče s stopnjo $d_u = 1$;
  oddaj poizvedbo, ki ima enico pri u, drugod pa ničle;
  v odgovoru je enica le pri u-jevem edinem sosedu — recimo mu v;
  izpiši, da je obstajala v drevesu povezava (u, v);
  pobriši to povezavo iz drevesa;

Tako bi sčasoma rekonstruirali celotno drevo. V resnici seveda (v zadnji vrstici gornje psevdokode) povezave ne moremo pobrisati, lahko pa to simuliramo: zmanjšajmo d_u in d_v za 1; stopnja bivšega lista u s tem pade na $d_u = 0$, kar si bomo razlagali kot znak, da je u pobrisan iz drevesa. V kakšni kasnejši iteraciji naše zanke se lahko pri nekem kasnejšem u zgodi, da bo v odgovoru na poizvedbo več enic, namreč ne le pri tistem sosedu, ki ga u trenutno še ima, pač pa tudi pri drugih sosedih, ki jih je imel v prvotnem drevesu, pa smo jih doslej že pobrisali. Enice pri že pobrisanih vozliščih lahko torej v odgovoru ignoriramo in tako še vedno brez težav ugotovimo, kateri je u-jev edini preostali (še nepobrisani) sosed.

Težava tega načrta je, da porabi $O(n)$ poizvedb, toliko pa jih ne smemo izvesti; na voljo imamo le približno $\log_2 n$ poizvedb (največ 16 poizvedb, vozlišč pa je do 30 000). Naj bo $B = \lfloor \log_2 n \rfloor + 1$; številke vozlišč si torej lahko predstavljamo kot B-bitna cela števila. Če bi lahko v poizvedbi za vsako vozlišče namesto enega samega bita oddali poljubno celo število, bi lahko tam uporabili številko vozlišča in v odgovoru bi pri vsakem listu dobili številko njegovega edinega soseda (pri notranjih vozliščih pa vsoto številk njihovih sosedov); tako pa bomo morali izvesti po eno poizvedbo za vsakega od B bitov, iz katerih so sestavljene številke naših vozlišč. Za vsak b od 0 do $B-1$ izvedimo torej po eno poizvedbo, kjer postavimo enice pri tistih vozliščih, ki imajo v svoji številki prižgan bit b. V odgovor na to poizvedbo dobimo za vsako vozlišče u podatek a_{bu}, ki pove, koliko sosedov vozlišča u ima v svoji številki prižgan bit b. S pomočjo teh podatkov lahko, če je u list, izračunamo številko njegovega edinega soseda: ker ima u le enega soseda, je lahko a_{bu} enak le 0 ali 1, odvisno od tega, ali ima tisti edini sosed v svoji številki na bitu b ničlo ali enico. Številka u-jevega soseda je torej $\sum_{b=0}^{B-1} 2^b a_{bu}$.

Razmisliti moramo še o tem, kako popraviti vrednosti a_{bu}, ko brišemo vozlišča iz drevesa. Tik preden smo pobrisali list u, je imel ta le enega soseda, recimo v; spremenijo se torej le vrednosti a_{bv} pri tem v, in sicer se zmanjšajo za 1 pri tistih b, za katere je bil bit b v u prižgan. Druga vozlišča niso imela u-ja za soseda in se jim zato tudi število sosedov (s prižganim bitom b v številki) ni spremenilo. Zapišimo zdaj psevdokodo tako dobljene rešitve:

```
oddaj poizvedbo, ki ima povsod enice; odgovori nanjo so stopnje d_1, ..., d_n;
B := ⌊log_2 n⌋ + 1;
for b := 0 to B − 1:
    oddaj poizvedbo, ki ima enice pri tistih u, v katerih je bit b prižgan;
    odgovori nanjo so števila a_{b1}, ..., a_{bn};

L := prazna množica;
for u := 1 to n do if d_u = 1 then dodaj u v L;
while L ni prazna:
    u := poljubno vozlišče iz L; pobriši ga iz L;
    v := 0; for b := 0 to B − 1 do v := v + 2^b · a_{bu};
    izpiši povezavo (u, v);
    d_v := d_v − 1; if d_v = 1 then dodaj v v L;
    for b := 0 to B − 1 do if je bit b v u prižgan then a_{bv} := a_{bv} − 1;
```

Na začetku torej določimo stopnje točk in vse a_{bu}; to je skupaj največ 16 poizvedb (pri $n = 30\,000$ so števila 15-bitna, torej imamo 15 poizvedb za a_{bu} in pred tem še eno za stopnje). Nato vzdržujemo množico listov L in na vsakem koraku izberemo neki list u, določimo njegovega soseda v, izpišemo povezavo med njima ter jo pobrišemo iz drevesa. Pri tem v lahko tudi sam postane list, v vsakem primeru pa moramo primerno zmanjšati njegove a_{bv}. Časovna zahtevnost te rešitve je $O(n \log n)$.

J. Pomešani skladi

Začnemo lahko z idejo, da bi karte po vrsti spravljali na pravo mesto:

```
x := 1; (* naslednja karta *)
for a := 1 to k do for i := 1 to C_a do if x ≤ n:
    poskrbi, da pride karta x na i-to mesto a-tega sklada
        (gledano od spodaj navzgor) in pri tem ne premikaj kart 1, ..., x − 1,
        ki so že na pravih mestih;
    x := x + 1;
```

To je prikladno, ker skladi dobivajo svojo končno podobo od spodaj navzgor; ko poskušamo spraviti karto x na i-to mesto trenutnega sklada, nam bodo pri tem v napoto karte na višjih mestih tega sklada in jih bomo morali odmakniti; karte na nižjih mestih pa bomo lahko brez škode pustili pri miru. Poleg tega takrat vemo, da so na teh nižjih mestih sklada a, pa tudi na vseh prejšnjih skladih (od 1 do $a - 1$), karte s številkami od 1 do $x - 1$; karto x bomo torej našli bodisi na enem od kasnejših skladov (od $a + 1$ do k) bodisi višje na trenutnem skladu a (od mesta i navzgor), zato bomo tudi tiste prejšnje sklade lahko brez težav pustili pri miru.

Razmislimo zdaj podrobneje o tem, kako spraviti x na želeno mesto. (1) Če je x že na skladu a, vendar višje od položaja i (nižje ne more biti, kajti tam so karte

s številkami $< x$), odmaknimo najprej z a vse karte nad x in nato še karto x sámo. Odmikamo jih lahko načeloma na poljuben sklad, ki ima še kaj praznega prostora (v poštev pridejo le skladi od $a+1$ naprej, saj so skladi od 1 do $a-1$ že v svojem končnem stanju in so torej povsem polni). V nadaljevanju torej predpostavimo, da x leži na nekem skladu $b > a$. (2) Če x še ni na vrhu sklada b, odmaknimo z njega vse karte, ki ležijo nad x. Spet jih lahko načeloma odmikamo kamorkoli, vendar po možnosti ne na a, kajti tam nam bodo pri naslednjem koraku v napoto; toda lahko se zgodi, da bodo vsi ostali skladi (razen a in b) polni in bomo morali nekaj kart vendarle premakniti na a. (3) Odmaknimo z a-ja vse karte razen spodnjih $i-1$. Odmikamo jih lahko kamorkoli razen na b, kajti na vrhu sklada b imamo karto x, ki jo bomo morali v naslednjem koraku premakniti na a in je zato ne bi bilo dobro zametati s kartami, ki jih zdaj odmikamo z a. (4) Zdaj je torej na skladu a natanko $i-1$ kart, karta x pa je na vrhu sklada b; od tam jo premaknimo na a in s tem je prišla na pravo mesto.

V točki (3) se stvari malo zapletejo. Lastnost ($\star$) iz besedila naloge nam zagotavlja, da je število praznih mest na vseh skladih skupaj vsaj tolikšno kot kapaciteta kateregakoli posameznega sklada; ali, z drugimi besedami, da lahko katerikoli posamezni sklad popolnoma izpraznimo tako, da karte z njega preložimo na druge sklade. Toda mi bi radi v točki (3) prelagali karte z a na poljubne druge sklade, *razen na b;* pri tej omejitvi pa ni več nujno, da bo na drugih skladih dovolj prostora za vse karte, ki bi jih radi odmaknili z a-ja. Razmišljamo lahko takole: ko pri odmikanju kart z a-ja vidimo, da bi z naslednjim takšnim odmikom na neki tretji sklad, recimo c, le-tega zapolnili do konca in da bi po tistem ostala prazna mesta le še na skladu b — takrat na tisto zadnje prazno mesto na c raje odmaknimo karto x s sklada b, preostale karte a-ja pa potem odmaknimo na b.

Toda to še ni dovolj; ni nujno, da tak tretji sklad c sploh obstaja. Lahko se zgodi, da smo pri $a = k-1$ in $b = k$, vsi prejšnji skladi pa so že dobili svojo končno podobo in naj jih načeloma ne bi več spreminjali. Ena možnost je, da si začasno izposodimo mesto na vrhu enega od prejšnjih skladov; recimo, da je to sklad c (za $c < a$) in da je tam na vrhu karta z (ki bo morala biti tam tudi na koncu). Položaj kart x, y in z na vrhovih skladov c, a in b lahko opišemo z urejeno trojico (karta y je tista, ki bi jo radi odmaknili z vrha sklada a); trenutno je to (z, y, x); iz nje s premikanjem kart po vrsti dobimo $(\sqcup, y, xz)$, $(y, \sqcup, xz)$, (y, z, x), $(y, zx, \sqcup)$, $(\sqcup, zx, y)$, $(\sqcup, z, yx)$ in $(z, \sqcup, yx)$; tako smo s sedmimi potezami premaknili y na sklad b, karta x je še vedno na vrhu b-ja, karta z pa se je vrnila na svoje mesto na vrhu sklada c.[34] To načeloma deluje, porabi pa neugodno veliko potez; v najslabšem primeru moramo z a-ja na ta način odmakniti $C_a - (i-1)$ kart; ko gre i od 1 do C_a, je to skupaj približno $C_a^2/2$ odmikov, kar je v najslabšem primeru približno $n^2/2$, tako da se teh potez nabere za $\approx \frac{7}{2}n^2$. To sicer ni preveč, saj je pri tej nalogi omejitev števila potez precej darežljiva: pri $n \leq 100$ smo omejeni na 10^5 potez, kar je vsaj $10n^2$. Vseeno pa razmislimo še o boljših rešitvah.

Namesto da z vedno znova vračamo nazaj na vrh c-ja, bi lahko s tremi potezami

[34] Videli smo, da je med temi premiki na skladu a občasno ena karta več kot na začetku. Ali je mogoče, da bi s tem presegli kapaciteto sklada a? To bi pomenilo, da je bil sklad a na začetku tega scenarija čisto poln; poleg tega so bili vsi ostali skladi razen b tudi čisto polni, sicer se s tem scenarijem sploh ne bi ukvarjali; sklad b pa ni čisto prazen, saj je na njem vsaj karta x. To troje skupaj pa je v protislovju z lastnostjo ($\star$), torej se to ne more zgoditi.

$(z, y, x) \to (\sqcup, yz, x) \to (x, yz, \sqcup) \to (x, y, z)$ zamenjali x in z, nato odmaknili na sklad b vse, kar moramo odmakniti z a-ja, in nato premaknili x s c na a; težava je, da je zdaj z zametan globoko na b-ju in moramo najprej vse, kar smo prej odložili nanj, premakniti nazaj na a, potem pa lahko premaknemo z nazaj na c. To je skupaj $2\alpha + 5$ potez, če je α število kart, ki smo jih morali odmakniti z a; teh je kvečjemu $\alpha \le C_a - (i - 1)$; ko gre i od 1 do C_a, se torej nabere kvečjemu $n^2 + O(n)$ potez, kar je veliko bolje od prejšnje rešitve.

Enako dobro, vendar še elegantnejšo rešitev pa dobimo, če se odpovemo dosedanjemu vztrajanju pri tem, da spravljamo karte na njihov končni položaj strogo po naraščajočem vrstnem redu njihovih številk. To je bilo namreč tisto, zaradi česar se nam je lahko sčasoma zgodilo, da sta nam ostala le še zadnja dva sklada (na prejšnjih pa je bilo načeloma vse že zacementirano), mi pa potrebujemo vsaj tri sklade, če hočemo poljubno spreminjati vrstni red kart. Število skladov sme pasti na dva šele takrat, ko nam ostane le še ena karta.

Zamislimo si torej fleksibilnejšo različico dosedanje rešitve. V vhodnih podatkih smo dobili sezname $S_1, \ldots, S_k$, ki nam povedo trenutno vsebino posameznih skladov (pri tem si mislimo, da smo pobrisali morebitne ničle, ki so v vhodnih podatkih predstavljale prazna mesta na koncu skladov). Ko premikamo karte med skladi, bomo seveda te sezname tudi popravljali in tako vzdrževali podatke o trenutnem razporedu kart. Poleg tega pa si pripravimo tudi podoben nabor seznamov $G_1, \ldots, G_k$, ki naj povedo želeno vsebino skladov v končnem stanju, kamor bi radi pripeljali stanje našega sistema. Imamo torej $G_1 = [1, 2, \ldots, C_1]$, nato $G_2 = [C_1 + 1, \ldots, C_1 + C_2]$ in tako naprej, dokler ne zmanjka kart; zadnjih nekaj skladov — vsaj eden, namreč zaradi lastnosti ($\star$) — pa bo na koncu praznih in dobijo tu prazen seznam $G_a = [\,]$. Naš postopek lahko zdaj zapišemo takole:

1 **while** $n > 0$:
2 naj bo a poljuben sklad, čigar G_a ni prazen;
3 naj bo x karta na začetku seznama G_a (torej karta, ki bo v končnem stanju ležala na dnu sklada a);
4 poskrbi, da pride karta x na dno sklada a;
5 zdaj je x prvi element tako seznama G_a kot seznama S_a; pobriši x z začetka obeh teh dveh seznamov;
6 $C_a := C_a - 1$; $n := n - 1$;

V vsaki iteraciji torej spravimo po eno karto na pravo mesto na dnu enega od skladov, nato pa jo v mislih preprosto pobrišemo — ni je več v naših seznamih G_a in S_a, skladu a pa zmanjšamo kapaciteto za 1 in s tem ponazorimo dejstvo, da mesta, ki ga na njem zaseda pravkar obdelana karta x, v bodoče ne bomo več mogli uporabljati. O tem, da se lastnost ($\star$) ohrani tudi po spremembah v vrstici 6, se ni težko prepričati: v neenačbi $n \le (\sum_i C_i) - \max_i C_i$ se je leva stran zmanjšala za 1; na desni strani se je prvi člen $\sum_i C_i$ tudi zmanjšal za 1, drugi člen $\max_i C_i$ pa se je bodisi zmanjšal za 1 ali pa ostal enak (odvisno od tega, ali je obstajal še kak drug sklad z vsaj tolikšno kapaciteto kot sklad a pred trenutno spremembo); zato se desna stran neenačbe bodisi zmanjša za 1 (če se je za 1 zmanjšal le prvi člen) bodisi ostane nespremenjena (če sta se za 1 zmanjšala oba člena). Leva stran se torej zmanjša za 1, desna pa za kvečjemu 1, zato je leva stran še vedno manjša ali enaka desni.

V vrstici 2 smo zgoraj zapisali „poljuben sklad“, vendar moramo dodati še nekaj: če je le mogoče, izberimo med skladi z nepraznim G_a kakšnega s kapaciteto vsaj 2; šele ko takih ni več, se lotimo tistih s kapaciteto 1. Lahko gremo celo še korak dlje in med skladi z nepraznim G_a izberemo vedno tistega z najvišjo kapaciteto; kajti nižji ko so naši skladi, manj dela je s tem, da se dokopljemo do karte x na skladu, kjer je trenutno morda zametana, in da odstranimo s sklada a vse karte, da bomo lahko na dno tega sklada premaknili x. Bolje je torej imeti več skladov z nižjo kapaciteto kot manj z višjo, zato najprej znižujmo višje sklade. V vsakem primeru pa s tem, da si sklade s kapaciteto 1 prihranimo za konec, poskrbimo, da število skladov (se pravi skladov s kapaciteto > 0; za tiste, ki jim kapaciteta pade na 0, si lahko mislimo, da za nas ne obstajajo več) ne bo prekmalu padlo pod 3: kajti število skladov se zmanjša, ko kapaciteta nekega sklada v vrstici 6 pade z 1 na 0; če ostaneta potem samo dva sklada, so morali biti pred tem trije; poleg sklada a recimo še b in c; in če smo izbrali a s kapaciteto $C_a = 1$, to pomeni, da sta morala imeti b in c tudi kapaciteto 1 ali pa prazen seznam G_b oz. G_c. Vsaj eden od njiju je moral imeti prazen seznam, kajti v končnem stanju je zaradi lastnosti ($\star$) vsaj zadnji sklad prazen in naš postopek vedno znižuje kapaciteto skladov, ki v končnem stanju niso prazni, torej je zadnji sklad še vedno prisoten z enako kapaciteto kot na začetku. Od skladov b in c je imel torej največ eden neprazen seznam G_b oz. G_c, poleg tega pa je ta sklad imel kapaciteto 1; tam je torej le ena karta, poleg tega pa je bila še ena na G_a, ki smo jo ravnokar pobrisali. Zdaj sta nam torej ostala dva sklada in ena karta, ta problem pa je enostavno rešljiv: če tista edina karta še ni na pravem skladu, jo pač prestavimo na drugega (ki je trenutno gotovo prazen).

Vrstico 4 opišimo podrobneje kot samostojen podprogram, da bomo lahko temeljiteje razmislili o njenem delovanju:

podprogram SpraviNaDno(karta x, sklad a):
1 če je x že na dnu a-ja, se takoj vrni iz podprograma;
2 dokler je x še na a:
3 premakni karto z vrha a na poljuben drug ne-poln sklad;
4 $b :=$ sklad, kjer je zdaj x;
5 dokler x ni na vrhu b-ja:
6 če obstaja kak ne-poln sklad, ki ni a ali b,
7 premakni karto z vrha b na neki tak sklad,
8 sicer premakni karto z vrha b na a;
9 dokler a ni prazen:
10 če je b edini ne-poln sklad, ki ni a:
11 naj bo c poljuben ne-prazen sklad, ki ni a ali b;
12 premakni karto z vrha c na a;
13 premakni x z vrha b na c; $b := c$;
14 premakni karto z vrha a na poljuben ne-poln sklad, ki ni b;
15 premakni x z vrha b na a;

Prepričajmo se, da ta postopek res deluje. — V vrsticah 2–3 nam lastnost ($\star$) zagotavlja, da je na drugih skladih dovolj prostora za vse karte z a. — V vrsticah 5–8 odmaknemo z b vse karte, pod katerimi leži x. Zaradi ($\star$) je na drugih skladih dovolj prostora zanje, lahko pa se zgodi, da jih bomo nekaj morali odložiti na a.

— Ko pridemo do vrstice 9, je x na vrhu svojega sklada (namreč b-ja); v vrsticah 9–14 bi radi izpraznili a, pri tem pa mora x ostati na vrhu svojega sklada, da ga bomo nato lahko (v vrstici 15) premaknili na a, ko bo le-ta prazen. Zaradi ($\star$) bo na drugih skladih dovolj prostora za karte z a, vendar se lahko zgodi, da jih bo treba nekaj odložiti na b. Slednje postane neizogibno, če so vsi skladi razen a in b že polni (vrstica 10); takrat z enega od polnih skladov, recimo c (vrstica 11), preselimo eno karto na a (vrstica 12) in na prazno mesto preselimo x (vrstica 13). Tako je x zdaj na vrhu nekega polnega sklada in lahko preostale karte a-ja premaknemo na bivši b (vrstica 14). — V vrstici 11 se moramo prepričati, da primeren c res obstaja. Ker smo še v zanki 9–14, sklad a ni prazen; pa tudi b ni prazen, ker je na njem karta x; torej imamo vsaj dve karti; že prej pa smo videli, da število skladov pade pod 3 šele takrat, ko število kart pade na 1; ker imamo dve karti, imamo torej vsaj tri sklade (z neničelno kapaciteto); poleg a in b mora torej obstajati še vsaj neki tretji sklad c; ker smo v vrstici 11, je pogoj iz vrstice 10 izpolnjen, torej je c poln; in ker ima kapaciteto > 0, vsebuje vsaj eno karto in jo bomo zato v vrstici 12 lahko premaknili z njega. — Pri tem premiku v vrstici 12 pa se moramo še prepričati, da je na a res prostor za novo karto. In res: če bi bil a takrat čisto poln, bi to pomenilo, da bi bili polni vsi skladi razen b, slednji pa tudi ni čisto prazen, saj vsebuje karto x; torej bi bilo skupno število prostih mest na vseh skladih (to je $(\sum_i C_i) - n$) manjše od C_b, to pa bi bilo v protislovju z lastnostjo ($\star$), po kateri je število prostih mest $\geq \max_i C_i$. Torej je nemogoče, da bi bil a čisto poln, zato lahko nanj preselimo eno karto s sklada c.

Pri tem postopku bi se sicer dalo še kaj malega izboljšati; na primer, prav mogoče je, da smo karto, ki jo v vrstici 12 selimo s c na a, premaknili na c z a v enem od prejšnjih izvajanj vrstice 14; lahko bi torej v vrstici 14 pazili, če bi s premikom karte z a zapolnili zadnje prosto mesto na zadnjem ne-polnem skladu, ki ni a ali b; in če po tem premiku a še ne bi bil prazen, bi lahko na tisto zadnje prosto mesto takoj preselili x (s sklada b), karte z a pa nato premaknili na b. Tako bi prihranili eno potezo. — Po drugi strani je mogoče tudi, da so se vsi skladi razen a in b zapolnili že ob selitvi kart z b-ja v vrstici 7. V tej vrstici bi lahko torej pazili, če bi s premikom karte z b zapolnili zadnje prosto mesto na zadnjem ne-polnem skladu, ki ni a ali b; in če po tem premiku x še ne bi bil na vrhu b-ja in/ali a ne bi bil prazen, potem bi bilo bolje tisto zadnje prosto mesto „rezervirati“ za x (ko bomo prišli do njega), trenutno karto z vrha b-ja pa že premakniti na a. Tudi tako bi prihranili eno potezo (ker v vrstici 12 ne bo treba šele narediti prostora za x).

Toda ta prihranek ne spremeni ničesar bistvenega. V najslabšem primeru moramo najprej premakniti (skoraj) vse karte z b na a, da pridemo do x, nato pa jih moramo še enkrat premakniti z a na b, da izpraznimo a. To je $2n + O(1)$ premikov; število kart se nam počasi zmanjšuje in če imamo takšno smolo pri vsakem n, se nabere $n^2 + O(n)$ premikov. Konkreten primer: recimo, da imamo tri sklade s kapacitetami $C_1 = C_3 = n$ in $C_2 = 1$; končno stanje je zato $[1, \ldots, n], [\,], [\,]$; in recimo, da je začetno stanje $[\,], [\,], [1, n, n-1, \ldots, 2]$. Nimamo torej druge možnosti, kot da za začetek poskusimo spraviti karto 1 na dno prvega sklada. Če sledimo gornji psevdokodi postopka SpraviNaDno, bomo po $2n + 1$ potezah prišli v stanje $[1], [\,], [2, n, n-1, \ldots, 3]$. Karta 1 je tako na pravem mestu; zdaj jo torej pobrišemo in zmanjšamo C_1 za 1. Tudi število kart je zdaj za 1 manjše; recimo mu $n' = n - 1$.

Zanj je $C_3 = n' + 1$, torej ima tretji sklad zdaj več prostora, kot je vseh kart, tako da mu lahko kapaciteto brez škode zmanjšamo na n'. Ostale so nam karte od 2 do $n' + 1$, kar lahko v mislih preštevilčimo na $1, \ldots, n'$. Zdaj imamo torej pred seboj razpored $[\,], [\,], [1, n', n' - 1, \ldots, 2]$ in kapacitete $(n', 1, n')$, kar je popolnoma enako kot na začetku, le z n' namesto n. Za naslednjo karto bomo torej porabili $2n' + 1 = 2n - 1$ potez, za naslednjo potem $2n - 3$ potez in tako naprej. Šele ko nam ostaneta samo dve karti, porabimo samo dve potezi (in ne pet, kot bi napovedala dosedanja formula), da spravimo prvo od njiju na dno prvega sklada, nato pa še eno potezo za drugo; skupaj je to $(2n + 1) + (2n - 1) + \ldots + 5 + 2 + 1 = n^2 + 2n - 5$ potez.

Ta rešitev je seveda več kot dovolj dobra za potrebe našega tekmovanja, saj smo videli, da imamo na voljo kar $10n^2$ potez; vseeno pa jo poskusimo še izboljšati. Naredimo za začetek majhen poskus: pri majhnih n in k je možnih razporedov kart dovolj malo, da lahko z iskanjem v širino poiščemo optimalno rešitev, torej táko z najmanjšim številom potez. Naj bo $f(n, k, C, S)$ to najmanjše število potez, če imamo n kart, k skladov s kapacitetami $C = (C_1, \ldots, C_k)$ in začetno stanje skladov $S = (S_1, \ldots, S_k)$. Zanimivo vprašanje je, kakšno je to najmanjše število potez v najslabšem primeru (in kateri je ta najslabši primer); definirajmo torej $f(n, k) := \max_C \max_S f(n, k, C, S)$, pri čemer gre prvi maksimum po vseh takih naborih kapacitet $C \in \{1, 2, \ldots, n\}^k$, ki ustrezajo lastnosti $(\star)$, drugi maksimum pa gre po vseh takih začetnih stanjih S, ki so konsistentna s kapacitetami C (torej kjer za vsak sklad i velja $|S_i| \leq C_i$).

Tabela na str. 176 kaže vrednosti $f(n, k)$ za nekaj majhnih n in k. Vidimo, da je — kot bi tudi pričakovali — problem tem težji, čim manj skladov imamo. Pri $k = 3$ lahko pri teh poskusih opazimo, da največje potrebno število potez, $f(n, 3)$, nastopi (med drugim) pri kapacitetah $C = (n, 1, n)$ in začetnem stanju oblike $[\,], [\,], [1, 2, \ldots, n]$ (če je n lih) oz. $[n, n - 1, \ldots, 1], [\,], [\,]$ (če je n sod). Če pogledamo vrednosti $f(n, 3)$ v gornji tabeli, lahko vidimo, da ko n narašča, se te vrednosti po vrsti povečujejo za $3, 5, 5, 7, 7, 9, 9$ in tako naprej. Iz tega lahko z nekaj telovadbe izpeljemo formulo

$$f(n, 3) = \tfrac{1}{2}n^2 + 2n - 2 + (n \bmod 2)/2.$$

Tabela sicer temelji na poskusih na majhnih n in ni nam uspelo dokazati, da velja ta formula tudi pri večjih n; vseeno pa lahko ob teh rezultatih posumimo, da se mora dati sestaviti rešitev, ki v najslabšem primeru porabi le $\frac{1}{2}n^2 + O(n)$ potez, ne pa $n^2 + O(n)$ kot naša dosedanja rešitev.

Doslej smo ciljno stanje $G = (G_1, \ldots, G_k)$ izračunali na začetku in ga kasneje nismo več spreminjali, razen ko smo pobrisali kakšno karto, ki smo jo pravkar spravili na pravo mesto na dnu njenega sklada. Recimo pa, da bi v G-ju premaknili karto z vrha sklada a na vrh sklada b; tako dobimo novo ciljno stanje, ki mu recimo G'. Če v nadaljevanju z resničnim premikanjem kart (torej s premiki, ki jih izpišemo na standardni izhod in ob tem tudi primerno popravimo trenutno stanje kart S) pripeljemo karte v to novo ciljno stanje G', lahko nato še premaknemo karto z vrha sklada b na sklad a, pa bomo prišli v stanje G. Tako bi lahko G spremenili tudi večkrat in na koncu je treba „razveljaviti“ vse te spremembe. Ko izvedemo premik v

n	$k=3$	4	5	6
1	1	1	1	1
2	4	4	4	4
3	9	6	6	6
4	14	11	8	8
5	21	15	13	10
6	28	19	17	15
7	37	24	21	19
8	46			
9	57			

Tabela kaže $f(n,k)$, minimalno potrebno število potez v najslabšem primeru, pri majhnih n in k. „V najslabšem primeru“ tukaj pomeni po vseh naborih kapacitet $(C_1,\ldots,C_k) \in \{1,2,\ldots,n\}^k$ in po vseh začetnih stanjih $(S_1,\ldots,S_k)$, ki so skladna s temi kapacitetami (torej kjer pri vseh i velja $|S_i| \le C_i$).

G, si ga moramo torej v nekem seznamu zapomniti in nato na koncu izvesti obratne premike v obratnem vrstnem redu. Lahko si tudi predstavljamo, da izvaja naš program dve vrsti premikov — resnične premike v S in „odložene“ premike v G — pri čemer je njegov cilj ta, da doseže stanje $S = G$, nato pa odložene premike res izvede (v obratni smeri in obratnem vrstnem redu).

Primer, ko je karta ena sama, lahko rešimo z eno potezo in končamo. V nadaljevanju torej predpostavimo, da sta karti vsaj dve.

Naj bo p sklad z najvišjo kapaciteto (če jih je več, je vseeno, katerega vzamemo). Za začetek z $O(n)$ premiki v G poskrbimo, da bo ta sklad v G prazen (karte z njega pa lahko premaknemo na poljubne druge sklade, kjer je še kaj prostora — zaradi lastnosti ($\star$) je prostora gotovo dovolj). Pri tem pa pazimo, da bosta vsaj dva sklada ne-prazna: če se slučajno znajdejo vse karte na enem skladu, premaknimo kakšno od njih na enega od praznih skladov (razen p). Ker imamo vsaj dve karti, sta zdaj vsaj dva sklada ne-prazna.

Nato z $O(n)$ premiki v S poskrbimo, da bo v S na vsakem skladu enako število kart kot v G, torej da bo $|S_i| = |G_i|$ za vse i. Če zdaj kak sklad i (ki ni p) ni poln, torej če ima $|S_i| = |G_i| < C_i$ kart, mu kapaciteto v mislih zmanjšajmo na $|G_i|$; po tej spremembi so vsi skladi razen p polni, slednji pa je prazen in ima najvišjo kapaciteto, zato lastnost ($\star$) še vedno drži. Za sklade, ki jim kapaciteta (po brisanju karte, ki je prišla na dno pravega sklada) pade na 0, se bomo v nadaljevanju delali, kot da ne obstajajo; tudi v k štejmo vedno le sklade s kapaciteto > 0.

Po tej predpripravi lahko začnemo razmišljati o glavnem delu našega postopka. Naši prejšnji rešitvi se je lahko zgodilo, da je porabila približno $2n$ potez, preden je spravila eno od kart na njeno pravo mesto na dnu nekega sklada in se je tako znebila. Tega si ne moremo privoščiti, če hočemo skupno število potez zmanjšati na $\frac{1}{2}n^2 + O(n)$; bodisi moramo v $2n$ potezah spraviti na pravo mesto vsaj dve karti (in ju potem pobrisati) bodisi moramo spraviti eno karto na pravo mesto v približno n potezah. Glavna zanka našega postopka bo torej takšna:

postopek BoljšaRešitev:
1 **while** $n > 1$:
2 (* *Imamo $k \ge 3$ skladov z neničelno kapaciteto; sklad p je prazen in ima najvišjo kapaciteto, ostali skladi pa so popolnoma polni.* *)
3 v $2n + O(1)$ korakih spravi $r \ge 2$ kart na pravo mesto na dnu r različnih skladov ali pa v $n + O(1)$ korakih spravi 1 karto na pravo mesto na dnu nekega sklada;

4 to karto ali karte pobriši in zmanjšaj n ter kapacitete njihovih skladov;
5 tu imamo še največ eno karto in če ta ni na pravem skladu,
jo z eno potezo spravimo na pravi sklad;

Vprašanje je seveda, kaj storiti v koraku 3, pri čemer moramo paziti, da bo invarianta iz koraka 2 še naprej veljala, torej da bo tudi po spremembah v korakih 3–4 sklad p prazen in da bomo, če število kart ne bo padlo pod 2, še vedno imeli poleg praznega vsaj dva polna sklada. Storili bomo eno od petih stvari.

(I) Če imamo več kot tri karte ($n > 3$), natanko tri sklade ($k = 3$) in ima od dveh polnih skladov eden kapaciteto 1, nastopi neke vrste robni primer. Skladu s kapaciteto 1 recimo a, drugemu polnemu skladu (s kapaciteto ≥ 1) pa b; ker sta to edina neprazna sklada, je na b torej $n - 1$ kart, kar je ≥ 3. Če bi hoteli spraviti pravo karto na dno obeh nepraznih skladov, a in b, in potem tidve karti pobrisati, bi se skladu a kapaciteta s tem zmanjšala na 0, tako da bi nam ostal le še en neprazen sklad, karti na njem pa bi bili vsaj dve; tak problem je lahko sploh nerešljiv in ga ne smemo dopustiti. Po drugi strani, če bi hoteli spraviti pravo karto le na dno sklada b, bi nam to v najslabšem primeru vzelo $2n + O(1)$ potez: če imamo smolo, je prava karta ravno eno mesto nad najnižjim, zato moramo najprej odmakniti z nje $n - 3$ kart nad njo; odmakniti jih nimamo drugam kot na sklad p; ker pa bo moral biti ta na koncu spet prazen, jih bomo morali kasneje spet premakniti s p na b.

Zato bomo naredili nekaj tretjega: na pravo mesto bomo spravili karti, ki morata biti na dnu in na vrhu sklada b. Karto, ki mora priti na dno sklada b, imenujmo x; tisto, ki mora priti na vrh, imenujmo y. Če je recimo x trenutno na skladu a, jo lahko v treh potezah zamenjamo s karto na vrhu sklada b; če pa je bila na vrhu sklada b karta y, zamenjamo x raje (v petih potezah) s karto, ki leži na skladu b tik pod y. Podobno ravnamo tudi, če je na skladu a karta y. Odslej torej predpostavimo, da sta x in y obe na skladu b. Recimo za začetek še, da leži x na tem skladu nižje kot y. Začetno stanje je torej oblike $[u], [\alpha, x, \beta, y, \gamma], [\,]$, pri čemer grške črke pomenijo neke neznane skupine 0 ali več kart; ko se taka skupina pri prevračanju z enega sklada na drugega obrne, jo bomo pisali kot α^r, β^r ipd.; število kart v skupini pa označimo z $|\alpha|$, $|\beta|$ in podobno. Zdaj naredimo takole:

Št. potez	Stanje skladov			Opomba
	a	b	p	
	$[u]$	$[\alpha, x, \beta, y, \gamma]$	$[\,]$	zač. stanje
1	$[\,]$	$[\alpha, x, \beta, y, \gamma]$	$[u]$	
$\lvert\gamma\rvert$	$[\,]$	$[\alpha, x, \beta, y]$	$[u, \gamma^r]$	
1	$[y]$	$[\alpha, x, \beta]$	$[u, \gamma^r]$	
$\lvert\beta\rvert$	$[y]$	$[\alpha, x]$	$[u, \gamma^r, \beta^r]$	(†)
1	$[\,]$	$[\alpha, x]$	$[u, \gamma^r, \beta^r, y]$	
1	$[x]$	$[\alpha]$	$[u, \gamma^r, \beta^r, y]$	
$\lvert\alpha\rvert$	$[x]$	$[\,]$	$[u, \gamma^r, \beta^r, y, \alpha^r]$	
1	$[\,]$	$[x]$	$[u, \gamma^r, \beta^r, y, \alpha^r]$	
$\lvert\alpha\rvert$	$[\,]$	$[x, \alpha]$	$[u, \gamma^r, \beta^r, y]$	
1	$[y]$	$[x, \alpha]$	$[u, \gamma^r, \beta^r]$	
$\lvert\beta\rvert$	$[y]$	$[x, \alpha, \beta]$	$[u, \gamma^r]$	
$\lvert\gamma\rvert$	$[y]$	$[x, \alpha, \beta, \gamma]$	$[u]$	
1	$[\,]$	$[x, \alpha, \beta, \gamma, y]$	$[u]$	
1	$[u]$	$[x, \alpha, \beta, \gamma, y]$	$[\,]$	

Tako smo v $2(|\alpha| + |\beta| + |\gamma|) + O(1) = 2n + O(1)$ potezah spravili x na dno, y pa

na vrh drugega sklada. (Če bi na začetku ležal y nižje na skladu b kot x, bi lahko ravnali enako kot zgoraj, le v stanju (†) bi s še tremi dodatnimi potezami zamenjali karti x in y.) Premaknimo zdaj karto y z vrha sklada b na p ne le v trenutnem stanju S, ampak tudi v ciljnem stanju G; nato jo v obeh pobrišimo z dna p-ja, karto x pa z dna b-ja; kapaciteti C_b in C_p zmanjšajmo za 1, novo število kart pa je $n-2$. Prva dva sklada sta spet polna, p je spet prazen in ima še vedno najvišjo kapaciteto. Pripravljeni smo na naslednjo iteracijo glavne zanke.

Mimogrede, čeprav smo na začetku primera (I) vzeli, da je $n > 3$, pa lahko uporabimo tu opisani scenarij tudi pri $n = 3$; njegov učinek takrat je, da na dno in vrh sklada b (s kapaciteto 2) prideta pravi karti; ta sklad torej dobi svojo končno podobo; za edino preostalo karto pa potem ne more biti drugače, kot da leži (kot edina) na skladu a in da je to tudi njeno končno mesto; problem je torej rešen in naš postopek se lahko konča.

(II) Še en robni primer je, da imamo vsaj dva sklada s kapaciteto 1 in kvečjemu en sklad s kapaciteto, večjo od 1; temu slednjemu (če obstaja) recimo a. Vsi skladi razen a (in praznega p) imajo torej kapaciteto 1 in jih je $k-2$. Z naslednjo zanko lahko vse karte, ki so trenutno na a, morale pa bi biti na ostalih skladih, spravimo na pravo mesto:

while a ni prazen:
  $x :=$ karta na vrhu a-ja;
  $b :=$ sklad, na katerem mora ležati x v končnem stanju;
  premakni x na p;
  if $b \neq a$: (* *potem je b eden od skladov s kapaciteto 1* *)
    premakni karto z b na a, nato premakni x s p na b;
premakni vse karte s p nazaj na a;

Zdaj so na a prav tiste karte, ki bodo morale biti na a tudi v končnem stanju, le da še niso nujno v pravem vrstnem redu; pa tudi na skladih s kapaciteto 1 so le take karte, ki bodo tudi v končnem stanju na skladih s kapaciteto 1, le da še ni nujno vsaka na pravem skladu. Dokončno jih lahko uredimo zdaj:

za vsak sklad b s kapaciteto 1:
  $x :=$ karta na skladu b;
  $c :=$ sklad, na katerem mora ležati x v končnem stanju;
  if $b \neq c$ **then** premakni x na p, nato premakni karto s c na b in nato x na c;

Za vsak sklad s kapaciteto 1 smo izvedli največ dva ali tri premike, da smo nanj spravili pravo karto (dva, če je prišla s sklada a, sicer pa tri); poleg tega smo vsako karto, ki bo v a tudi na koncu, premaknili dvakrat (z a na p in nazaj); to je skupaj manj kot $2n + 3(k-2)$ premikov, na pravo mesto pa smo spravili $k-2$ kart, kar je ≥ 2; te karte bomo zdaj lahko pobrisali, torej je bila cena na vsako pobrisano karto $\leq n + O(1)$. Paziti pa moramo na nekaj: po takšnem brisanju bi nam ostal le sklad a (poleg seveda praznega p); če ima ta potem kapaciteto 1, je tam edina karta že tudi na pravem mestu in je problem rešen. Če pa ima a po brisanju več kot eno karto, lahko problem postane nerešljiv; v tem primeru moramo enega od skladov s kapaciteto 1 pustiti in karte na njem ne pobrisati, čeprav je že na pravem mestu; tako nam ostanejo (s p-jem vred) trije skladi, od tega eden s kapaciteto 1, in lahko problem rešimo do konca po primeru (I).

Če sklada a sploh ni, ker imajo vsi skladi kapaciteto 1, nam že samo drugi del postopka iz primera (II) postavi vse karte na pravo mesto z manj kot $3n$ potezami in lahko potem končamo.

Recimo zdaj, da pogoji za robna primera (I) in (II) niso izpolnjeni. Za vsak neprazni sklad u naj bo x_u karta, ki mora v končnem stanju ležati na dnu tega sklada. Mislimo si usmerjen graf, v katerem je po ena točka za vsak neprazen sklad, povezava $u \to v$ pa pove, da karta x_u trenutno leži na skladu v. Vsaka točka ima torej natanko eno izhodno povezavo. Če začnemo v poljubni točki in sledimo izhodnim povezavam, se morajo točke prej ali slej začeti ponavljati, in sicer bodisi v obliki cikla bodisi zanke (povezave $u \to u$). Takih ciklov in/ali zank je lahko v grafu tudi več; natančneje povedano, vsaka šibko povezana komponenta našega grafa je sestavljena iz enega cikla ali zanke, na katerega ali katero je lahko pripetih še nič ali več dreves, v katerih so vse povezave usmerjene k ciklu oz. zanki.[35]

(III) Če sta v grafu vsaj dve zanki, to pomeni dva sklada a in b, pri čemer karta x_a leži na skladu a, karta x_b pa na skladu b. Brez izgube za splošnost naj bo a tisti izmed njiju, na katerem je manj kart. Ker je vseh kart skupaj n, jih je torej na a lahko kvečjemu $n/2$. Ker sklad a vsebuje karto x_a, je njegova vsebina oblike $S_a = [\alpha, x_a, \beta]$; in ker sklad b ni prazen, imenujmo karto na vrhu tega sklada y, tako da je b potem oblike $S_b = [\gamma, y]$. Zdaj izvedimo naslednje premike:

Št. potez	Stanje skladov			Opomba
	a	b	p	
	$[\alpha, x_a, \beta]$	$[\gamma, y]$	$[\,]$	zač. stanje
$\lvert\beta\rvert$	$[\alpha, x_a]$	$[\gamma, y]$	$[\beta^r]$	
1	$[\alpha, x_a]$	$[\gamma]$	$[\beta^r, y]$	
1	$[\alpha]$	$[\gamma, x_a]$	$[\beta^r, y]$	
$\lvert\alpha\rvert$	$[\,]$	$[\gamma, x_a]$	$[\beta^r, y, \alpha^r]$	
1	$[x_a]$	$[\gamma]$	$[\beta^r, y, \alpha^r]$	
$\lvert\alpha\rvert$	$[x_a, \alpha]$	$[\gamma]$	$[\beta^r, y]$	
1	$[x_a, \alpha]$	$[\gamma, y]$	$[\beta^r]$	
$\lvert\beta\rvert$	$[x_a, \alpha, \beta]$	$[\gamma, y]$	$[\,]$	

Število potez je $2(|\alpha| + |\beta|) + 4 = 2(|S_a| + 1) \leq n + 2$; tako smo torej v $n + O(1)$ potezah spravili karto x_a na dno sklada, kamor tudi spada. Zdaj jo lahko pobrišemo (iz S_a in G_a) ter zmanjšamo n in C_a za 1. Sklad p je spet prazen, tako da smo pripravljeni na novo iteracijo glavne zanke.

(IV) Če je v grafu ena sama šibko povezana komponenta in se ta konča v zanki, potem ta komponenta gotovo vsebuje vsaj dve točki, saj imamo vsaj dva neprazna sklada. Poleg točke z zanko mora torej obstajati še vsaj ena točka, ki kaže v točko z zanko; imamo torej povezavi oblike $a \to b \to b$. To pomeni, da sklad b vsebuje tako karto x_a kot x_b. Recimo za začetek, da leži na b karta x_a nižje kot karta x_b. Sklad a je neprazen; karti na vrhu tega sklada recimo y. Zdaj izvedimo naslednje premike:

[35] S takšnimi grafi smo se na naših tekmovanjih že srečali; gl. npr. nalogo 2018.3.5, str. 79 v *Biltenu* 2018.

Št. potez	Stanje skladov			Opomba
	a	b	p	
	$[\alpha, y]$	$[\beta, x_a, \gamma, x_b, \delta]$	$[\,]$	zač. stanje
$\|\delta\|$	$[\alpha, y]$	$[\beta, x_a, \gamma, x_b]$	$[\delta^r]$	
1	$[\alpha]$	$[\beta, x_a, \gamma, x_b]$	$[\delta^r, y]$	
1	$[\alpha, x_b]$	$[\beta, x_a, \gamma]$	$[\delta^r, y]$	
$\|\gamma\|$	$[\alpha, x_b]$	$[\beta, x_a]$	$[\delta^r, y, \gamma^r]$	(†)
1	$[\alpha, x_b]$	$[\beta]$	$[\delta^r, y, \gamma^r, x_a]$	
$\|\beta\|$	$[\alpha, x_b]$	$[\,]$	$[\delta^r, y, \gamma^r, x_a, \beta^r]$	
1	$[\alpha]$	$[x_b]$	$[\delta^r, y, \gamma^r, x_a, \beta^r]$	
$\|\beta\|$	$[\alpha]$	$[x_b, \beta]$	$[\delta^r, y, \gamma^r, x_a]$	
1	$[\alpha, x_a]$	$[x_b, \beta]$	$[\delta^r, y, \gamma^r]$	
$\|\gamma\|$	$[\alpha, x_a]$	$[x_b, \beta, \gamma]$	$[\delta^r, y]$	
Sedanje stanje p-ja zapišimo v obliki $[z, \epsilon]$:				
	$[\alpha, x_a]$	$[x_b, \beta, \gamma]$	$[z, \epsilon]$	
$\|\delta\|$	$[\alpha, x_a]$	$[x_b, \beta, \gamma, \epsilon^r]$	$[z]$	
1	$[\alpha]$	$[x_b, \beta, \gamma, \epsilon^r, x_a]$	$[z]$	
$\|\alpha\|$	$[\,]$	$[x_b, \beta, \gamma, \epsilon^r, x_a]$	$[z, \alpha^r]$	
1	$[x_a]$	$[x_b, \beta, \gamma, \epsilon^r]$	$[z, \alpha^r]$	
$\|\alpha\|$	$[x_a, \alpha]$	$[x_b, \beta, \gamma, \epsilon^r]$	$[z]$	
1	$[x_a, \alpha]$	$[x_b, \beta, \gamma, \epsilon^r, z]$	$[\,]$	

Skupno število potez je $2(|\alpha| + |\beta| + |\gamma| + |\delta|) + 8 = 2(C_a + C_b) + O(1) \leq 2n + O(1)$, na pravo mesto pa smo spravili dve karti, x_a in x_b, ki ju lahko zdaj pobrišemo ter zmanjšamo kapaciteti skladov a in b za 1, število kart n pa za 2. Če bi na začetku ležala na skladu b karta x_b nižje od x_a, bi lahko še vedno uporabili enak scenarij, le da bi v stanju (†) s še tremi dodatnimi potezami zamenjali karti x_a in x_b.

Ko karto x_a pobrišemo z dna sklada a, se lahko zgodi, da mu kapaciteta s tem pade na 0, zato ta sklad zdaj izgine in k se zmanjša za 1. Toda prej smo imeli bodisi $n \leq 3$ bodisi $k > 3$, kajti drugače bi se znašli v primeru (I) in ne (IV); torej imamo zdaj bodisi še vedno vsaj tri sklade ali pa je število kart padlo na $n \leq 1$, tedaj pa ni nič narobe, če imamo le še dva sklada. Tako bo torej naloga ostala rešljiva, četudi smo sklad a izgubili.

(V) Če ne velja nobeden od primerov (III) in (IV), je neizogibno, da v našem grafu obstaja cikel. Brez izgube za splošnost recimo, da cikel tvorijo skladi od 1 do r in da za njihove kapacitete velja $C_1 \leq C_2 \leq \cdots \leq C_r$ (če drugega ne, lahko sklade začasno preštevilčimo). Vrstni red, v katerem si skladi sledijo na ciklu, lahko predstavimo s permutacijo π nad števili $\{1, \ldots, r\}$, ki nam pove, da se karta x_a, ki bo v končnem stanju ležala na dnu sklada a, trenutno nahaja nekje na skladu $\pi(a)$. Inverz te permutacije nam potem pove, da sklad a vsebuje karto, ki bo v končnem stanju ležala na dnu sklada $\pi^{-1}(a)$.

Skladi $1, \ldots, r$ in p tvorijo zaporedje vse večjih skladov, pri čemer je zadnji trenutno prazen. Naslednik a-ja v tem zaporedju je sklad $N(a)$, kjer je $N(a) = a+1$ za $1 \leq a < r$ in $N(r) = p$. Za začetek bomo vsak sklad a od r do 1 (v padajočem vrstnem redu) prekopicnili na $N(a)$ (kjer je gotovo dovolj prostora), pri tem pa pazili, da pride na vrh tista od kart $x_1, \ldots, x_r$, ki se je nahajala na skladu a. Tako pridejo sčasoma karte $x_1, \ldots, x_r$ (ne nujno v tem vrstnem redu) na vrh skladov $2, \ldots, r, p$ (na vsakem od teh skladov je lahko tudi še kaj praznega prostora), sklad 1 pa je takrat prazen. Poljubni dve karti x_i in x_j lahko v treh potezah zamenjamo (tako, da eno začasno odložimo na sklad 1); z največ $r-1$ takšnimi zamenjavami pa lahko

poskrbimo, da bodo na vrhu skladov $2, \ldots, r, p$ po vrsti karte $x_1, \ldots, x_r$. Potem moramo le še prekopicniti vsakega od skladov $2, \ldots, r, p$ na njegovega predhodnika, pa bodo karte $x_1, \ldots, x_r$ prišle na dno pravih skladov. Zapišimo ta postopek s psevdokodo:

postopek PrimerV:
1 **for** $a := r$ **downto** 1:
2 $b := N(a)$; (* *Tu velja* $C_a \leq C_b$*, sklad* b *pa je trenutno prazen.* *)
3 premakni vse karte s sklada a na sklad b, pri čemer pa naj karta $x_{\pi^{-1}(a)}$ pride na vrh;
(* *Zdaj je sklad 1 prazen, karte* $x_1, \ldots, x_r$ *pa so na vrhu skladov* $2, \ldots, r, p$*, vendar lahko v nekem premešanem vrstnem redu.* *)
4 **for** $a := 1$ **to** r:
5 $b := N(a)$;
6 če x_a ni na skladu b, jo zamenjaj s karto, ki je trenutno na vrhu b-ja;
(* *Zdaj je za vsak* a *karta* x_a *na vrhu sklada* $N(a)$*.* *)
7 **for** $a := 1$ **to** r:
8 $b := N(a)$; (* *Sklad* a *je prazen, na vrhu* b*-ja je karta* x_a*.* *)
9 dokler b ni prazen, premikaj karte z b-ja na a;

O vrstici 3 moramo razmisliti malo podrobneje. Da bo manj pisanja, recimo karti, ki jo je treba spraviti na vrh, preprosto x. Če je x na dnu a-ja, moramo le prekopicniti a na b, pa bo x prišel na vrh b-ja. Recimo torej zdaj, da x ni na dnu a-ja; torej vsebuje a vsaj dve karti. Najlažje je, če lahko karte, ki so bile na a-ju nad x, premaknemo na b, nato začasno odložimo x na neki tretji sklad c, kjer je še kaj prostora, nato premaknemo na b preostanek kart z a-ja in nato premaknemo še x na b; toda ni nujno, da je na kakšnem tretjem skladu še kaj prostora (npr. če naš cikel obsega vse ne-prazne sklade in če imajo vsi skladi enako kapaciteto).

Recimo torej, da smo prisiljeni vzeti za pomožni sklad c neki sklad, ki je že poln. Karti na vrhu c-ja recimo z; karti, ki je dnu a-ja (spomnimo se, da x ni na dnu), pa recimo y. Zdaj lahko naredimo takole:

Št. potez	Stanje skladov			Opomba
	c	a	b	
	$[\gamma, z]$	$[y, \alpha, x, \beta]$	$[\,]$	zač. stanje
$\lvert\beta\rvert$	$[\gamma, z]$	$[y, \alpha, x]$	$[\beta^r]$	
1	$[\gamma]$	$[y, \alpha, x]$	$[\beta^r, z]$	
1	$[\gamma, x]$	$[y, \alpha]$	$[\beta^r, z]$	
$\lvert\alpha\rvert$	$[\gamma, x]$	$[y]$	$[\beta^r, z, \alpha^r]$	
1	$[\gamma, x]$	$[\,]$	$[\beta^r, z, \alpha^r, y]$	
1	$[\gamma]$	$[x]$	$[\beta^r, z, \alpha^r, y]$	
1	$[\gamma, y]$	$[x]$	$[\beta^r, z, \alpha^r]$	
1	$[\gamma, y]$	$[\,]$	$[\beta^r, z, \alpha^r, x]$	

Tako smo vsebino a-ja preselili na b in pri tem pustili x na vrhu, porabili pa smo $C_a + O(1)$ potez. To je dobro, manj dobro pa je, da smo z vrha sklada c ukradli karto z in tam namesto nje pustili y. Če obstaja kak c, ki ni del našega trenutnega cikla, lahko uporabimo njega in potem ni nič narobe, če zamenjamo z in y; toda lahko se zgodi, da so vsi skladi del cikla. Če pa je c na našem ciklu, moramo paziti, da z

njega ne ukrademo ravno tiste karte izmed $x_1, \ldots, x_r$, ki leži na skladu c. Če lahko izberemo tak c, ki svoje x_i nima ravno na vrhu, mu lahko ukrademo karto z vrha in bo vse v redu; toda lahko se zgodi, da imajo vsi skladi cikla vsak svojo x_i ravno na vrhu. Če lahko izberemo tak c, ki vsebuje vsaj dve karti, lahko gornji postopek prilagodimo tako, da c-ju ukrademo drugo najvišjo karto namesto najvišje:

Št. potez	Stanje skladov			Opomba
	c	a	b	
	$[\gamma, z, w]$	$[y, \alpha, x, \beta]$	$[\,]$	zač. stanje
$\lvert\beta\rvert$	$[\gamma, z, w]$	$[y, \alpha, x]$	$[\beta^r]$	
1	$[\gamma, z]$	$[y, \alpha, x]$	$[\beta^r, w]$	
1	$[\gamma, z]$	$[y, \alpha]$	$[\beta^r, w, x]$	
1	$[\gamma]$	$[y, \alpha, z]$	$[\beta^r, w, x]$	
1	$[\gamma, x]$	$[y, \alpha, z]$	$[\beta^r, w]$	
1	$[\gamma, x, w]$	$[y, \alpha, z]$	$[\beta^r]$	
1	$[\gamma, x, w]$	$[y, \alpha]$	$[\beta^r, z]$	
$\lvert\alpha\rvert$	$[\gamma, x, w]$	$[y]$	$[\beta^r, z, \alpha^r]$	
1	$[\gamma, x, w]$	$[\,]$	$[\beta^r, z, \alpha^r, y]$	
1	$[\gamma, x]$	$[w]$	$[\beta^r, z, \alpha^r, y]$	
1	$[\gamma]$	$[w, x]$	$[\beta^r, z, \alpha^r, y]$	
1	$[\gamma, y]$	$[w, x]$	$[\beta^r, z, \alpha^r]$	
1	$[\gamma, y]$	$[w]$	$[\beta^r, z, \alpha^r, x]$	
1	$[\gamma, y, w]$	$[\,]$	$[\beta^r, z, \alpha^r, x]$	

To je še vedno $C_a + O(1)$ potez, kar je dobro. Ali se lahko zgodi, da ne moremo izbrati takega c-ja, ki bi vseboval vsaj dve karti? Do tega pride, če imajo vsi a-jevi predhodniki (skladi $1, \ldots, a-1$) kapaciteto 1 in če je a zadnji na ciklu ($a = r$, $b = p$), kajti drugače bi obstajal v zaporedju $1, \ldots, a, \ldots, r, p$ poleg a-jevega naslednika b še b-jev naslednik c, ki bi imel (ker so kapacitete v zaporedju naraščajoče) $C_c \geq C_a \geq 2$. Toda zdaj se pogovarjamo o scenariju, kjer cikel obsega vse sklade in imajo vsi ti skladi razen enega kapaciteto 1; to pa ne more biti res, kajti potem bi zapadli pod primer (II), ne pa pod primer (V), kjer smo zdaj.

Tako smo dorekli vse potrebno o vrstici 3 postopka PrimerV; razmislimo zdaj o skupnem številu operacij v njem. Za vsak sklad a na ciklu izvedemo $C_a + O(1)$ premikov v vrstici 3 in še C_a v vrstici 9; poleg tega imamo v vrstici 6 še $O(r)$ zamenjav, vsaka zamenjava pa je načeloma sestavljena iz treh premikov. Skupaj imamo torej $2(\sum_a C_a) + O(r) \leq 2n + O(r)$ premikov, učinek celotnega postopka pa je, da na dno vsakega od skladov $1, \ldots, r$ pride prava karta, ki jo lahko nato pobrišemo. Ker je $r \geq 2$, je bila torej cena na vsako pobrisano karto $\leq n + O(1)$.

Postopek BoljšaRešitev torej res v vsaki iteraciji svoje glavne zanke izvede bodisi $2n$ operacij in pobriše vsaj dve karti bodisi n operacij in pobriše eno karto, poleg tega pa izvede še po $O(1)$ operacij za vsako pobrisano karto. Ko se n počasi zmanjšuje od prvotnega števila kart do 1, se nam tako nabere kvečjemu $\frac{1}{2}n^2 + O(n)$ operacij.

V praksi se sicer izkaže, da porabi tale „boljša“ rešitev na mnogih testnih primerih precej več potez kot naša prejšnja rešitev (tista z $n^2 + O(n)$ operacijami); njena prednost pred slednjo je le, da se zanjo ne da najti patoloških primerov, kjer bi število potez naraslo do n^2.

K. Ključi

Dvorec si lahko predstavljamo kot neusmerjen graf, ki ima po eno točko za vsako sobo in eno povezavo za vsaka vrata. Naloga zagotavlja, da je graf povezan. Recimo pa, da v mislih točko 1 (zunanjost) pobrišemo; zdaj se lahko zgodi, da graf razpade na več povezanih komponent. Točka 1 ima v vsaki od teh komponent kakšno sosedo.

Ena od teh komponent vsebuje tudi točko 0 (spalnico); recimo ji C. Ker je bil prvotni graf povezan, je v njem obstajala vsaj ena pot od 0 do 1; vse točke vsake take poti, razen točke 1 na koncu, torej tudi pripadajo komponenti C; zadnji korak take poti pa gre po povezavi iz neke točke C-ja v točko 1. Točka 1 ima torej vsaj eno sosedo, ki pripada komponenti C, lahko pa je takih sosed tudi več.

(1) Recimo, da sta taki sosedi vsaj dve, na primer a in b. Naj bo π poljubna pot (znotraj C) od 0 do a in naj bo ρ poljubna pot (znotraj C) od b do a; naj bo z prva taka točka na π, ki leži tudi na ρ (lahko je tudi $z = 0$ ali $z = a$; s tem ni nič narobe). Pot π torej lahko v mislih razdelimo na dva dela: π_1 od 0 do z ter π_2 od z do a; enako tudi ρ razdelimo na ρ_1 od b do z in ρ_2 od z do a. Zdaj lahko sestavimo naslednji scenarij: Bob naj na potovanje vzame ključe za pot ρ_1 in za vrata med b in 1, Alica pa naj vzame ključe za pot π in za vrata med a in 1. Alica naj gre po π_1 od 0 do z, pusti v točki z vse ključe za π_1 in nadaljuje po π_2 od z do a in od tam v 1. Ko pride Bob domov, naj gre iz 1 v b in od tam po ρ_1 do z, tam pobere ključe za π_1 (ki jih je prej tam pustila Alica) in gre nato po tej poti (v obratni smeri) od z do 0.

(2) Druga možnost pa je, da ima 1 samó eno sosedo v komponenti C; recimo tej sosedi a. To pomeni, da se iz 0 v 1 (ali obratno) ne da priti drugače kot skozi a; Bob bo potreboval ključ povezave $(1, a)$, da bo lahko prišel do spalnice; toda tega ključa ne sme vzeti s seboj na potovanje, saj ga bo Alica potrebovala, da bo lahko sploh prišla iz hiše. Ta ključ mora torej vzeti Alica, Bob pa lahko pride do njega le, če mu ga Alica nekje pusti; ne sme pa mu ga pustiti v komponenti C, saj Bob v to komponento ne more vstopiti drugače kot s tem ključem, ki pa ga še nima.

Če naj torej sploh obstaja scenarij, po kakršnem sprašuje naloga, to pomeni, da bo moral Bob pobrati ključ povezave $(1, a)$ v neki drugi komponenti, recimo C', in da bo v C' vstopil prej kot v C. Recimo, da bo v C' vstopil po povezavi $(1, b)$ za neko točko $b \in C'$. Če je b edina soseda točke 1 v komponenti C', imata Alica in Bob glede ključa te povezave enak problem kot prej s ključem povezave $(1, a)$: Alica potrebuje ključ $(1, b)$, da bo lahko prišla v komponento C' in nekje v njej pustila za Boba ključ $(1, a)$; torej Bob ključa $(1, b)$ ne sme vzeti s seboj na potovanje; torej mu ga mora Alica pustiti v neki tretji komponenti C''. Toda potem bi bilo že vseeno, če bi mu v C'' pustila kar ključ $(1, a)$, v komponento C' pa niti njej niti njemu ne bi bilo treba vstopiti. S komponento, v kateri ima točka 1 le eno sosedo, si torej Alica in Bob ne moreta pomagati.

Potreben pogoj za obstoj scenarija, po kakršnem sprašuje naloga, je torej ta, da obstaja neka komponenta C', v kateri sta vsaj dve sosedi točke 1, recimo b in c; ni pa se težko prepričati, da je ta pogoj tudi zadosten. Pot (po C) od 0 do a imenujmo π, tako kot prej. Ker je C' povezana komponenta, v njej obstaja pot od b do c, ki jo imenujmo ρ. Primeren scenarij je zdaj naslednji: Bob naj vzame na potovanje ključ povezave $(1, b)$, Alica pa naj vzame ključe povezav na π, ρ in še ključa povezav $(1, a)$ in $(1, c)$. S temi ključi naj gre po poti π od 0 do a, od tam skozi vrata v 1,

od tam skozi vrata v c in nato po poti ρ (v obratni smeri) od c do b. V točki b naj pusti ključe vseh povezav na π in še ključ povezave $(1, a)$, nato pa naj se po ρ vrne iz b v c ter izstopi skozi vrata v 1. Ko se Bob vrne s potovanja, naj gre po povezavi $(1, b)$, pobere v točki b ključe, ki jih je tam pustila Alica, izstopi nazaj v 1 in nato s ključi, ki jih je prej pobral, vstopi iz 1 v a in gre od tam po poti π (v obratni smeri) v 0.

Zapišimo zdaj naš postopek še s psevdokodo. Množico vseh točk označimo z $V = \{0, \ldots, n-1\}$, množico vseh sosed točke u pa z $N(u)$. Povezane komponente bomo poiskali z iskanjem v širino iz sosed točke 1; pri vsaki točki u si bomo v $P[u]$ zapomnili njeno neposredno predhodnico na poti od 1 do u (torej točko, iz katere smo pri iskanju prišli v u), v $C[u]$ pa sosedo točke 1, v katero smo vstopili na prvem koraku poti od 1 do u — točka $C[u]$ je torej neke vrste predstavnica celotne povezane komponente, do katere se pride skoznjo. S pomočjo tabele C bomo kasneje zlahka določili točke a, b in c, ki jih potrebujemo za naša dva scenarija, s pomočjo tabele P pa bomo pripravili poti π in ρ.

```
(* Poiščimo povezane komponente. *)
for u ∈ V do P[u] := −1, C[u] := −1;
for u ∈ N(1) do if P[u] < 0:
  Q := prazna množica; dodaj u v Q; P[u] := 1; C[u] := u;
  while Q ni prazna:
    naj bo v poljubna točka iz Q; pobriši jo iz Q;
    for w ∈ N(v) do if w ≠ 1 and P[w] < 0:
      dodaj w v Q; P[w] := v; C[w] := C[v];

(* Poglejmo, če je mogoč scenarij (1). *)
a := C[0]; b := −1; π := [0, P[0], P[P[0]], ..., a];
for u ∈ N(1) do if u ≠ a and C[u] = a:  b := u; break;
if b ≥ 0:
  ρ := [b, P[b], P[P[b]], ..., a];
  naj bo z zadnja točka v π, ki je tudi v ρ;
  razdeli π pri z na π1 in π2, podobno pa ρ na ρ1 in ρ2;
  izpiši scenarij (1) s potmi π1, π2, ρ1 in končaj;

(* Poglejmo, če je mogoč scenarij (2). *)
c := −1; for u ∈ N(1) do if C[u] ≠ a and C[u] ≠ u:  b := u; c := C[u]; break;
if b < 0 then izpiši, da je problem nerešljiv, in končaj;
ρ := [b, P[b], P[P[b]], ..., c];
izpiši scenarij (2) s potema π in ρ;
```

L. Označene poti

Problem, ki ga rešujemo pri tej nalogi, je podoben znanemu problemu najkrajših poti v grafih, vendar s pomembno razliko: pri iskanju najkrajših poti si pogosto pomagamo z dejstvom, da je najkrajšo pot od s do u gotovo mogoče dobiti tako, da začnemo z najkrajšo potjo od s do neke druge točke v in gremo nato v zadnjem koraku po povezavi $v \to u$. Pri našem problemu leksikografsko najmanjših poti pa ni nujno res, da je leksikografsko najmanjša pot od s do u podaljšek leksikografsko najmanjše poti od s do v. Če na primer obstajata od s do v poti z oznakama `a`

in `ab` (leksikografsko manjša je prva od njiju) in če imamo namen pot podaljšati s povezavo, ki ima oznako `c`, bomo dobili boljši rezultat (`abc`), če začnemo z drugo potjo namesto s prvo (kjer dobimo `ac`).

Bolje je, če poti podaljšujemo na začetku namesto na koncu. Recimo, da nas zanima leksikografsko najmanjša pot od u do t; vsaka pot od u do t se začne s korakom po eni od u-jevih izhodnih povezav, recimo $u \to v$, in se nadaljuje kot pot od v do t. Pri vseh poteh, ki se začnejo z istim korakom $u \to v$, se oznaka poti začne z oznako povezave $u \to v$; med temi potmi bo torej leksikografsko najmanjša tista, pri kateri je leksikografsko najmanjši preostanek njene oznake, ta preostanek pa je ravno oznaka preostanka poti (od v do t). Za korakom $u \to v$ moramo torej nadaljevati po leksikografsko najmanjši poti od v do t. Ker vnaprej ne vemo, katera izmed u-jevih izhodnih povezav bo dala najmanjšo pot sploh, moramo preizkusiti vse in si zapomniti najboljšo. Ker je naš graf acikličen, ga lahko preiskujemo v topološkem vrstnem redu iz točke t nazaj (v nasprotni smeri povezav) in tako sčasoma dobimo najmanjše poti od vseh drugih točk do t. V resnici nas seveda zanima le najmanjša pot od s do t, vendar ne samo za en konkreten t, pač pa za vse možne t; zato bomo morali tak postopek pognati $O(n)$-krat, po enkrat za vsak t.

Zapišimo našo rešitev s psevdokodo; množico točk grafa označimo z V, množico povezav z E, množico u-jevih neposrednih predhodnic pa s $P(u)$.

```
1   for t ∈ V:
      (* Za vsako u naj nam R[u] pove, ali je t dosegljiva iz u; D[u] naj bo
         število u-jevih neposrednih naslednic, iz katerih je t dosegljiva;
         N[u] naj bo u-jeva naslednica na doslej najmanjši poti od u do t. *)
2     for u ∈ V do R[u] := FALSE, D[u] := 0, N[u] := NIL;
3     R[t] := TRUE; Q := {t};
4     while Q ni prazna:
5       u := poljubna točka iz Q; pobriši jo iz Q;
6       for v ∈ P(u):
7         D[v] := D[v] + 1;
8         if not R[v]:  R[v] := TRUE; dodaj v v Q;
9     Q := prazna množica; dodaj t v Q;
10    while Q ni prazna:
11      naj bo v poljubna točka iz Q; pobriši jo iz Q;
12      for u ∈ P(u):
13        if N[u] = NIL or je pot [u, v, N[v], N[N[v]], ..., t] leksikografsko
            manjša od poti [u, N[u], N[N[u]], ..., t] then N[u] := v;
14        D[u] := D[u] − 1; if D[u] = 0 then dodaj u v Q;
15    izpiši, da je [s, N[s], N[N[s]], ..., t] najmanjša pot od s do t;
```

Najprej torej (v vrsticah 4–8) z iskanjem v širino nazaj iz točke t označimo, iz katerih točk je t sploh dosegljiva (tabela R), in pri vsaki točki tudi preštejemo, skozi koliko izmed njenih izhodnih povezav se dá priti v t (tabela D). V drugem delu postopka (vrstice 10–14) pa uporabljamo vrednost $D[u]$ kot število tistih u-jevih neposrednih naslednic v, skozi katere je mogoče priti v t in zanje še nismo ocenili poti $u \to v \rightsquigarrow t$. Ko to število pade na 0, vemo, da za u zdaj poznamo najmanjšo pot do t, zato dodamo u v vrsto Q, da bomo kasneje upoštevali to pot pri sestavljanju najmanjših poti od u-jevih predhodnic do t.

Zgoraj je zanka v vrsticah 11–15 zapisana tako, da vedno preišče ves tisti del grafa, iz katerega je dosegljiva točka t, toda v resnici se seveda lahko ustavimo, čim dodamo s v Q; takrat poznamo najmanjšo pot od s do t, to pa je tudi vse, kar nas pri tem t zares zanima.

Kakšna je časovna zahtevnost tega postopka? V najslabšem primeru moramo pri $O(n)$ različnih t-jih preiskati večino grafa; pri vsakem takem t imamo torej načeloma $O(n+m)$ dela s pregledovanjem grafa, poleg tega pa moramo tudi $O(m)$-krat izvesti primerjavo dveh poti v vrstici 14, za tako primerjavo pa ne moremo pričakovati, da se jo bo dalo izvesti v konstantnem času. Predvsem si ne moremo privoščiti, da bi oznake poti predstavili eksplicitno kot nize, saj so za kaj takega predolge; pot je sestavljena iz $O(n)$ povezav, oznaka posamezne povezave pa je dolga do $d \leq 10^6$ znakov, torej je lahko oznaka poti dolga tudi po več sto milijonov znakov.

Pomagati si bomo morali z dejstvom, da oznake povezav pri tej nalogi niso čisto poljubni nizi, pač pa so oznake vedno podnizi niza A (ki smo ga dobili v vhodnih podatkih). Recimo torej, da imamo pred seboj dve oznaki poti, $w = w_1 \dots w_\omega$ in $z = z_1 \dots z_\zeta$, pri čemer so nizi $w_1, \dots, w_\omega, z_1, \dots, z_\zeta$ oznake posameznih povezav in zato podnizi niza A. Da ugotovimo, kateri izmed w in z je leksikografsko manjši, gremo načeloma lahko po nizih od leve proti desni in iščemo prvo neujemanje (torej mesto, kjer istoležna znaka obeh nizov nista enaka; tam moramo potem pogledati, kateri niz ima leksikografsko manjši znak). Po nizih se bomo premikali v zanki, kjer bomo v vsaki iteraciji pogledali, ali med trenutnima kosoma obeh nizov, recimo w_i in z_j, nastopi kakšno neujemanje; če da, smo končali, če ne, pa se lahko premaknemo po nizih tako daleč naprej, da se vsaj eden od obeh kosov konča.

funkcija JeManjši(w, z):
  vhod: niza $w = w_1 \dots w_\omega$, $z = z_1 \dots z_\zeta$;
  izhod: logična vrednost, ki pove, ali je $w < z$;

  $i := 1$; $j := 1$; $p_w := 0$; $p_z := 0$;
  while $i \leq \omega$ **and** $j \leq \zeta$:

    (* *Naš trenutni položaj v nizih w in z pade pri prvem p_w znakov od začetka kosa w_i, pri drugem pa p_z znakov od začetka kosa z_j. Poglejmo, kateri od teh dveh kosov se prej konča.* *)
    $\ell := \min\{|w_i| - p_w, |z_j| - p_z\}$;

    (* *Ali nastopi med kosoma neujemanje in kje?* *)
    $r :=$ dolžina najdaljšega skupnega prefiksa (⋆)
        nizov $w_i[p_w : p_w + \ell]$ in $z_j[p_z : p_z + \ell]$;
    if $r < \ell$ **then return** $w_i[p_w + r] < z_j[p_z + r]$;

    (* *Premaknimo se naprej po nizih, dokler se eden od trenutnih kosov ne konča.* *)
    $p_w := p_w + r$; **while** $i \leq \omega$ **and** $p_w \geq |w_i|$ **do** $p_w := p_w - |w_i|$, $i := i + 1$;
    $p_z := p_z + r$; **while** $j \leq \zeta$ **and** $p_z \geq |z_j|$ **do** $p_z := p_z - |z_j|$, $j := j + 1$;

(* *Če pridemo do sem, je en niz prefiks drugega; torej je $w < z$ v primeru, če je w tisti, ki se je končal prej kot z.* *)
return $j \leq \zeta$;

Ker se v vsaki iteraciji glavne zanke poveča vsaj eden od števcev i in j, izvedemo

največ $O(\omega+\zeta)=O(n)$ iteracij. Vprašanje je le še, kako v vrstici ($\star$) poiskati prvo neujemanje med nizoma $w_i[p_w:p_w+\ell]$ in $z_j[p_z:p_z+\ell]$ (ki sta podniza niza A) ali pa ugotoviti, da sta si enaka. Ogledali si bomo dva pristopa, enega bolj pravovernega in enega bolj šušmarskega, ki pa za potrebe naše naloge deluje dovolj dobro.

Šušmarski pristop temelji na Rabin-Karpovih razprševalnih kodah. Vsako črko angleške abecede si predstavljajmo kot majhno celo število, recimo od 0 do 25; izberimo si bazo b (recimo $b=26$) in delitelj M (recimo kakšno veliko praštevilo, npr. $M=10^9+7$); potem lahko poljubnemu nizu $x=x[1]\,x[2]\ldots x[k]$ pripišemo razprševalno kodo $h(x)=(\sum_{i=1}^{k} b^{k-i}x[i]) \bmod M$. Lepo pri tako definiranih razprševalnih kodah je, da jih lahko poceni računamo za različne podnize niza $A=A[1]\,A[2]\ldots A[d]$. Najprej jih izračunajmo za vse prefikse A-ja: definirajmo $h_i := h(A[1]A[2]\ldots A[i])$; računamo jih lahko po formulah $h_0=0$ in $h_{i+1}=(b\cdot h_i+A[i+1]) \bmod M$. Razprševalno kodo poljubnega podniza lahko potem izračunamo iz kod dveh takih prefiksov: $h(A[i+1]\ldots A[j])=(h_j-b^{j-i}h_i) \bmod M$. Vrstico ($\star$) lahko zdaj implementiramo takole: najprej preverimo, če sta niza enaka — tedaj je najdaljši skupni prefiks enak njuni dolžini, $r=\ell$; če pa nista enaka, lahko najdaljši skupni prefiks poiščemo z bisekcijo po dolžini prefiksa (začnemo z dejstvom, da je najdaljši skupni prefiks dolg vsaj 0 in manj kot ℓ). Ko je treba primerjati niza ali neka njuna prefiksa med sabo, pa tega ne naredimo s primerjanjem znak po znak (ker bi bilo prepočasi), pač pa le izračunamo razprševalni kodi obeh nizov oz. prefiksov in preverimo, če sta enaki ali ne. Ker imamo ves čas opravka s podnizi niza A, lahko, kot smo videli, takšno razprševalno kodo vsakič izračunamo v $O(1)$ časa (pred tem smo morali na začetku sicer porabiti $O(d)$ časa in prostora za izračun vseh h_i, vendar je to le enkraten strošek in se bo utopil v ceni preostanka naše rešitve). Časovna zahtevnost vrstice ($\star$) je torej $O(1)$, če se niza ujemata, in $O(\log d)$, če se ne in moramo izvesti bisekcijo; to slednje pa je potrebno v okviru celega postopka JeManjši le enkrat, kajti po tistem se postopek konča. Časovna zahtevnost postopka JeManjši je torej zdaj $O(n+\log d)$, časovna zahtevnost celotne rešitve pa $O(nm(n+\log d))$, kar v praksi pomeni $O(n^2m)$.

Slabost tega pristopa je, da ne zagotavlja pravilnosti rezultatov; načeloma se lahko zgodi, da imata dva različna niza enako razprševalno kodo in naš postopek bi takšno neujemanje spregledal. Za na tekmovanje je sicer ta rešitev čisto dobra, saj je verjetnost, da bo do napake prišlo ravno na tamkajšnjih testnih primerih, zanemarljivo majhna; vseeno pa se spodobi razmisliti tudi o rešitvi, ki zagotovo najde pravilen rezultat.

Pomagamo si lahko s sufiksno tabelo (*suffix array*); to je tabela $S[0..d]$, v kateri so sufiksi niza A urejeni v leksikografskem vrstnem redu (vsak sufiks je seveda predstavljen samo s številom od 0 do d, ki pove, na katerem indeksu v A se ta sufiks začne). Poleg tega imejmo za vsaka dva zaporedna sufiksa v tej tabeli še dolžino njunega najdaljšega skupnega prefiksa; naj bo torej recimo $P[t]$ najdaljši skupni prefiks nizov $A[S[t]:]$ in $A[S[t+1]:]$. V podrobnosti priprave tabel S in P se tu ne bomo spuščali, saj obstajajo za gradnjo sufiksnih tabel razni dobro znani postopki, ki jih lahko tu uporabimo brez kakršnih koli sprememb.[36] Koristno je imeti še „inverz“

[36] Gl. npr. Wikipedijo *s. v.* Suffix array in tam navedeno literaturo. Sufiksno tabelo S lahko zgradimo v $O(d)$ časa z algoritmom DC3 (J. Kärkkäinen, P. Sanders, S. Burkhardt, “Linear work suffix array construction”, *J. of the* ACM, 53(6):918–36 (November 2006)), prav tako v linearnem času pa tudi tabelo P (T. Kasai *et al.*, “Linear-time longest-common-prefix computation in

tabele S, torej tabelo SI, v kateri nam vrednost $SI[i]$ pove, na katerem indeksu v tabeli S se nahaja vrednost i.

Za poljubna dva A-jeva sufiksa, recimo $A[i{:}]$ in $A[j{:}]$, lahko potem izračunamo najdaljši skupni prefiks tako, da pogledamo, kje v tabeli S se tadva sufiksa nahajata, in vzamemo minimum vrednosti P med njima. Na primer: recimo, da je $S[t] = i$ in $S[u] = j$ (indeksa t in u dobimo s pomočjo tabele SI); in recimo brez izgube za splošnost, da je $t < u$; ker so sufiksi v S urejeni leksikografsko, to pomeni, da če se $A[S[t]{:}]$ in $A[S[u]{:}]$ začneta na neki skupni prefiks, se nanj začnejo tudi vsi sufiksi med njima, $A[S[t+1]{:}]$ do $A[S[u-1]{:}]$. Dolžina tega skupnega prefiksa torej ne more biti večja, kot je dolžina skupnega prefiksa katerihkoli dveh zaporednih sufiksov na tem območju, to pa je $\min\{P[t], P[t+1], \ldots, P[u-1]\}$.

Da bomo lahko poceni računali takšne minimume po več zaporednih elementov tabele P, si lahko zgradimo nad njo še drevesno strukturo za iskanje minimumov na intervalu; z drugimi besedami, pripravimo $\log_2 d$ tabel velikosti $d/2$, $d/4$, $d/8$ in tako naprej, ki vsebujejo minimume po dveh, štirih, osmih itd. elementov tabele P. Ko nas potem zanima minimum nekaj zaporednih elementov P-ja, moramo v vsaki od teh tabel pogledati največ dva elementa; tako dobimo minimum v $O(\log d)$ časa. Te dodatne tabele porabijo $O(d)$ prostora, kar ni nič več od prvotne tabele P. Časovna zahtevnost celotne rešitve je potem $O(n^2 m \log d)$, kar je za testne primere na našem tekmovanju že dovolj dobro.

Še bolje pa je, če si pripravimo $\log_2 d$ tabel velikosti d, kjer v k-ti od teh tabel element $P_k[t]$ vsebuje minimum vrednosti $P[t], P[t+1], \ldots, P[t+2^k-1]$. Ko nas zanima minimum vrednosti $P[t], P[t+1], \ldots, P[u-1]$, vzamemo $k = \lfloor \log_2(u-t) \rfloor$ in vidimo, da lahko interval od t do $u-1$ pokrijemo z dvema intervaloma dolžine 2^k: enim, ki se začne pri t, in enim, ki se konča pri $u-1$ (tadva intervala se lahko tudi prekrivata, s čimer ni nič narobe); iskani rezultat je torej $\min\{P_k[t], P_k[u-2^k]\}$. Tako lahko dolžino najdaljšega skupnega prefiksa izračunamo v $O(1)$ časa namesto $O(\log d)$, časovna zahtevnost celotne rešive pa je le $O(n^2 m)$. Cena za to pa je, da smo porabili $O(d \log d)$ časa in pomnilnika za pripravo tabel P_k; pri omejitvah na našem tekmovanju ($d = 10^6$, pomnilniška omejitev 512 MB) si to še lahko privoščimo.[37]

REŠITVE NALOG POSKUSNEGA TEKMOVANJA

X. Izgubljena lica

Povezave so pari (u_i, v_i); če je pri kakšni $u_i > v_i$, krajišči obrnimo, tako da bo odslej vedno $u_i < v_i$; nato povezave uredimo naraščajoče po u_i, tiste z enakim u_i pa po v_i. To gre načeloma v $O(m)$ časa (stabilno urejanje s štetjem po v_i in nato še po

suffix arrays and is applications", *Combinatorial Pattern Matching: 12th Ann. Symposium* (CPM 2001), LNCS 2089, pp. 181–92). Za namene naše naloge pa je čisto dovolj dober tudi starejši in preprostejši pristop, ki najprej uredi sufikse A-ja glede na prvi znak, nato glede na prva dva, prve štiri, prvih osem in tako naprej; skupaj porabi $O(d \log d)$ časa (U. Manber, G. Myers, "Suffix arrays: a new method for on-line string searches, *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA '90), pp. 319–27).

[37] Obstaja tudi postopek, s katerim lahko še vedno računamo najdaljše skupne prefikse v $O(1)$ časa, pred tem pa porabimo le $O(d)$ časa in prostora za predpripravo podatkov (namesto $O(d \log d)$ kot v naši tu opisani rešitvi); gl. M. A. Bender, M. Farach-Colton, "The LCA problem revisited", *Proc. of the 4th Lat. Am. Symp. on Theoretical Informatics* (LATIN 2000), LNCS 1776, str. 88–94.

u_i), vendar je za naše namene čisto dovolj dober tudi kakšen od splošnih postopkov s časovno zahtevnostjo $O(m \log m)$.

Za vsako točko grafa pripravimo seznam njenih sosedov in jih uredimo naraščajoče po polarnem kotu: če smo pri točki u, vidimo njeno sosedo v v smeri $(x_v - x_u, y_v - y_u)$; polarni kot te smeri lahko izračunamo na primer s funkcijo `atan2` iz C++ove standardne knjižnice, pri čemer nam tudi ni treba skrbeti zaradi numeričnih nenatančnosti, saj naloga zagotavlja, da se bodo te smeri razlikovale za vsaj 10^{-9} radianov.[38] Lahko pa se računanju z ne-celimi števili tudi izognemo, če u-jeve sosede najprej uredimo glede na to, ali imajo y-koordinato večjo ali manjšo od y_u, nato pa za primerjanje smeri uporabimo vektorski produkt: točka w leži levo od v (gledano iz točke u), če je $(x_v - x_u)(y_w - y_u) - (y_v - y_u)(x_w - x_u) > 0$. Za urejanje sosedov točke u porabimo $O(d_u \log d_u)$ časa, če je d_u stopnja točke u; skupaj po vseh u je to $O(\sum_u d_u \log d_u) = O(\sum_u d_u \log n) = O(m \log n)$.

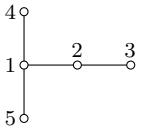
V seznamu u-jevih sosedov (za vsak u) hranimo pri vsakem sosedu tudi indeks povezave (v seznamu vseh m povezav) med tem sosedom in u. Ko so seznami sosedov urejeni, se sprehodimo po njih in si pri vsaki povezavi (u, v) zapišimo, na katerem mestu je v v seznamu u-jevih sosedov in na katerem mestu je u v seznamu u-jevih sosedov.

Vsaka povezava (u, v) načeloma meji na dve lici, eno na levi strani (če se postavimo v u in gledamo proti v) in eno na desni, čeprav moramo paziti tudi na možnost, da je na obeh straneh povezave isto lice (glej sliko na str. 190). V našem seznamu povezav bomo pri vsaki povezavi hranili tudi dve logični vrednosti, ki bosta za vsako stran povezave (levo in desno) povedali, ali smo že pregledali lice na tisti strani te povezave.

Zdaj bomo, dokler ne bodo vse povezave pregledane po obeh straneh, ponavljali naslednji postopek. Vzemimo prvo táko povezavo, kjer vsaj ena stran še ni bila pregledana; spomnimo se, da imamo povezave (u, v) urejene po u, tiste z enakim u pa po v, pri čemer je tudi vedno $u < v$. Če je zdaj (u, v) prva povezava, ki ima vsaj eno nepregledano stran, to pomeni, da lice na tisti strani ne vsebuje nobene točke s številko, nižjo od u; in da, četudi točko u morda obišče še večkrat, iz nje nikoli ne nadaljuje v točko s številko, nižjo od v; torej se bo kanonični opis tega lica začel ravno z (u, v). Zato je bilo koristno urediti povezave: tako bomo lažje brez dodatnega truda prišli do kanoničnih opisov lic.

Začnimo torej v tako izbrani povezavi (u, v) in si izberimo stran, na kateri je še nepregledana (če je na eni že pregledana, pa izbire tako ali tako nimamo). Recimo, da je to leva stran povezave; torej bi želeli licu slediti tako, da bomo imeli povezave ves čas na svoji desni. Ko pridemo iz u v v s povezavo na svoji desni, bomo morali nadaljevati v tistega izmed v-jevih sosedov, ki je v urejenem seznamu v-jevih sosedov tik pred točko u (seznam si moramo predstavljati ciklično; če je u prvi sosed v seznamu v-jevih sosedov, je njegov predhodnik pač zadnji sosed v seznamu). Recimo, da je to sosed w; zdaj se torej premaknemo iz v v w in nato z enakim razmislekom vidimo, da bomo morali v naslednjem koraku iti iz w v točko, ki je v-jev neposredni predhodnik v seznamu w-jevih sosedov. Tako nadaljujemo, dokler ne pridemo nazaj

[38] V resnici še za malo več. Če so koordinate naših točk cela števila z območja $[-a, a]$, bo najmanjši kot med dvema sosedoma nastopil na primer takrat, ko imamo točko $(-a, -a)$ s sosedoma (a, a) in $(a, a-1)$. Pri naši nalogi je $a = 10^7$ in kót med omenjenima sosedoma bi bil približno $2{,}5 \cdot 10^{-8}$.



Primer, kjer je isto lice na obeh straneh povezav. Če začnemo obhod s povezavo $(1,2)$ in imamo povezave ves čas na naši desni, dobimo $1,2,3,2,1,5,1,4$. Vidimo, da je na obhodu korak $2 \to 1$, ko smo povezavo $(1,2)$ obiskali še z druge strani. Če na tistem mestu naš opis obrnemo, dobimo $1,2,3,2,1,4,1,5$, prav to pa je opis, ki bi ga dobili tudi, če bi se že na začetku odločili imeti povezave na naši levi namesto na desni. Drugi opis je leksikografsko manjši od prvega, zato je kanoničen.

---

v u; takrat je naš obhod po licu sklenjen. Obiskane točke si seveda zapisujemo v neki seznam, ki bo tako predstavljal opis našega lica. Poleg tega med obhodom pri vsaki uporabljeni povezavi tudi označimo, da smo ustrezno stran povezave že pregledali.[39]

Če bi začeli s tem, da je nepregledana desna stran začetne povezave in da bomo imeli zato povezave ves čas na svoji levi, ko se bomo delali obhod po licu, bi bil razmislek podoben, le da bi morali ob prihodu v novo točko za naslednji korak vzeti v seznamu njenih sosedov neposrednega naslednika (namesto predhodnika) točke, iz katere smo pravkar prišli.

Opis lica, ki smo ga dobili pri enem in drugem obhodu, je načeloma že kanoničen, razen v naslednjem primeru: če je bila začetna povezava (u,v) prej nepregledana po obeh straneh, se lahko zgodi, da že ob prvem obhodu dobimo lice, ki obdaja to povezavo na obeh straneh. Drugega obhoda potem sploh ne bi naredili, ker bi bila takrat povezava (u,v) že pregledana na obeh straneh; toda opis, ki smo ga dobili pri prvem obhodu, ni nujno kanoničen (glej sliko zgoraj). Takrat moramo torej poiskati v našem obhodu drugi obisk povezave (u,v); ta se gotovo zgodi v nasprotni smeri, torej $v \to u$; pripravimo nato opis, ki se začne pri tem koraku, vendar gre v nasprotno smer, tako da se spet začne z $u \to v$. Tako imamo zdaj dva opisa istega lica; kanoničen je tisti, ki je leksikografsko manjši.

Ostane še možnost, da smo pri isti začetni povezavi (u,v) naredili dva obhoda in dobili dve različni lici (ker prvi obhod še ni obiskal povezave (u,v) po *obeh* straneh). V tem primeru sta opisa, ki smo ju dobili, že kanonična in moramo le še preveriti, kateri je leksikografsko manjši, da ga izpišemo prej kot drugega.

Zapišimo glavni del našega postopka še s psevdokodo:

```
1   za vsako povezavo (u,v) v leksikografskem redu:
2     če je (u,v) na levi še nepregledana, izvedi obhod z začetkom v u → v,
      pri čemer imej povezave ves čas na svoji desni; dobljenemu opisu recimo L1;
3       če je v L1 tudi korak v → u:
4         naj bo L1' opis, ki ga dobimo, če vrstni red točk v L1 obrnemo
          in ga zamaknemo tako, da se spet začne z u → v (bivšim v → u);
5         če je L1' < L1, priredimo L1 := L1';
6     če je (u,v) na desni še nepregledana, izvedi obhod z začetkom v u → v,
      pri čemer imej povezave ves čas na svoji levi; dobljenemu opisu recimo L2;
7     če smo dobili oba opisa, L1 in L2, izpiši najprej leksikografsko manjšega
      od njiju in potem še drugega; če smo dobili le en opis, izpiši pač njega;
```

[39] Spomnimo se, da imamo med obhodom povezave na svoji desni. Če smo šli po povezavi (x,y) od x do y, smo torej pregledali levo stran povezave, če pa smo šli po njej od y do x, smo pregledali njeno desno stran.

Tako sčasoma izpišemo vsa lica v leksikografskem vrstnem redu njihovih kanoničnih opisov. S pregledovanjem in izpisovanjem lic smo imeli $O(m)$ dela, saj je bilo treba vsako povezavo obdelati dvakrat (po vsaki strani enkrat). Časovna zahtevnost celotne rešitve je torej $O(m \log n)$, zaradi urejanja sosedov vsake točke po kotu.

Y. Snežna odeja

Vhodne podatke preberimo v dvodimenzionalno tabelo; recimo, da so stolpci oštevilčeni od 0 do $w-1$, vrstice pa od 0 (na vrhu) do $h-1$ (na dnu). Pojdimo v zanki po stolpcih in v vsakem stolpcu x z vgnezdeno zanko poiščimo najvišji znak `#`. Če ga najdemo na primer v vrstici y, popravimo v trenutnem stolpcu pike v zvezdice v prejšnjih treh vrsticah, torej na poljih $(x, y-1)$, $(x, y-2)$ in $(x, y-3)$. Če v stolpcu ni nobenega znaka `#`, se delajmo, kot da smo ga našli pri $y = h$, torej tik pod spodnjim robom slike. Ko tako obdelamo vse stolpce, moramo sliko le še izpisati.

Ta rešitev načeloma porabi $O(w \cdot h)$ pomnilnika za tabelo s sliko; to pa lahko zmanjšamo, če rezultate izpisujemo sproti že med branjem. Imejmo tabelo s, v kateri element $s[x]$ pove, ali smo v stolpcu x že narisali snežinke ali ne. Ko preberemo novo vrstico, poglejmo za vsak x, ali je v trenutni vrstici na mestu x znak `#` in ali je $s[x]$ še **false**; če je tako, je to najvišji znak `#` v tem stolpcu in moramo v prejšnje tri vrstice na indeksu x vpisati zvezdico. Vidimo torej, da ko preberemo vrstico y, se lahko še kaj spremenijo vrstice $y-1$, $y-2$ in $y-3$; ko bomo nato prebrali $y+1$, se lahko spremenijo y, $y-1$ in $y-2$, vrstica $y-3$ pa ne več, zato jo lahko izpišemo po tistem, ko smo prebrali in obdelali vrstico y, in preden preberemo vrstico $y+1$. Tako moramo torej v pomnilniku hraniti le štiri vrstice naenkrat, poleg njih pa še tabelo s; poraba prostora je le še $O(w)$. Zapišimo to rešitev še s psevdokodo:

naj bo $T[0..3]$ tabela z dovolj prostora za štiri vrstice;
for $x := 0$ **to** $w-1$ **do** $s[x] :=$ **false**;
for $y := 0$ **to** $h+2$:
  if $y < h$ **then** preberi naslednjo vrstico v $T[y \bmod 4]$;
  if $y \leq h$ **then for** $x := 0$ **to** $w-1$:
    if not $s[x]$ **and** ($T[y \bmod 4][x] =$ `'#'` **or** $y = h$):
      $s[x] :=$ **true**; **for** $i := 1$ **to** 3 **do** $T[(y-i) \bmod 4][x] :=$ `'*'`;
  if $y \geq 3$ **then** izpiši vrstico $T[(y-3) \bmod 4]$;

Z. Časovni napad

Ker nam, če predlagamo napačno geslo, sistem pove le položaj prvega neujemanja med njim in pravim geslom, nima smisla predlagati dolgega niza, saj ne bomo izvedeli ničesar koristnega o tem, ali so znaki za prvim neujemanjem pravilni ali napačni. Recimo, da že vemo, da se geslo začne na niz s dolžine n znakov (na začetku je to res za prazen niz s in za $n = 0$) in da bi radi zdaj ugotovili, kakšen je naslednji znak gesla. Težava je, da če si izberemo neki znak c in sistemu predlagamo niz sc, bo njegov odgovor $n+1$ tako v primeru, če je c res naslednji znak gesla (ker se niz tam konča), kot v primeru, če ni (ker tam nastopi neujemanje). Drugačen odgovor bi dobili le v primeru, če bi bilo sc že celotno geslo (takrat bi sistem odgovoril `SUCCESS`, mi pa bi vedeli, da lahko končamo).

Zato je bolje, če sistemu namesto *sc* predlagamo neki daljši niz, ki se začne na *sc*; kako se nadaljuje, ni pomembno — vzemimo na primer *scc*. Če dobimo odgovor $n+1$, potem vemo, da se pravo geslo ne začne na *sc* in moramo poskusiti s kakšnim drugim *c*; če dobimo odgovor $n+2$, potem vemo, da se geslo začne na *sc*, torej lahko *c* v mislih dodamo na konec niza *s*, povečamo *n* za 1 in nadaljujemo po enakem postopku, da bomo uganili spet naslednji znak gesla; če pa po srečnem naključju dobimo odgovor `SUCCESS`, potem vemo, da je pravo geslo ravno *scc* in lahko končamo.

Lahko se zgodi, da ko tako preizkusimo niz *scc* za vse možne *c* (črke, številke in ločila), dobimo vedno odgovor $n+1$. To pomeni, da je geslo dolgo $n+1$ znakov; takrat poskusimo še enkrat vse znake *c*, vendar zdaj z nizom *sc* namesto *scc*. Pri enem bomo zdaj dobili odgovor `SUCCESS` namesto $n+1$ in takrat lahko končamo s postopkom.

Poseben primer nastopi pri $n = 22$: nizov oblike *scc* takrat nima smisla predlagati, saj vemo, da bodo predolgi (24 znakov, geslo pa jih ima največ 23); takrat gremo lahko torej naravnost na nize oblike *sc*.

Zapišimo naš postopek še s psevdokodo:

```
s := prazen niz; n := 0;
while true:
  ok := false;
  if n < 23 then za vsak možni znak c:
    predlagaj sistemu niz scc;
    if je odgovor SUCCESS then končaj postopek
    else if je odgovor n + 2 then
      s := sc; n := n + 1; ok := true; break;
  if not ok then za vsak možni znak c:
    predlagaj sistemu niz sc;
    if je odgovor SUCCESS then končaj postopek;
```

Možnih znakov *c* je 90 (26 velikih črk, 26 malih črk, 10 števk in 28 ločil), tako da bomo izvedli v najslabšem primeru $23 \cdot 90 = 2070$ ugibanj (če bo pravi vedno zadnji znak, ki ga bomo preizkusili), torej smo še daleč od meje 5000 ugibanj, pri kateri bi nas sistem blokiral.

Naloge so sestavili: izgubljena lica — Nino Bašić; filogenetika, podajanje žoge, prisotnost — Tomaž Hočevar; pomešani skladi — Tomaž Hočevar in Janez Brank; torte — Tomaž Hočevar in Vid Kocijan; kadrovska služba — Tomaž Hočevar in Maks Kolman; sušenje perila — Tomaž Hočevar in Tim Poštuvan; interaktivna rekonstrukcija, ključi — Vid Kocijan; gremo na Luno — Vid Kocijan in Jure Slak; časovni napad — Maks Kolman; enaki urniki — Jure Slak; snežna odeja — Mitja Trampuš; označene poti — Janez Brank.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2021

1. Eskalacija

Vzdrževati moramo množico prisotnih na sestanku (da jih ne bomo po nepotrebnem klicali) in seznam ljudi, ki smo jih že poklicali, pa še niso prišli; pri njih moramo za vsakega hraniti tudi podatek o tem, kdaj mine petminutni rok, preden pokličemo drugič ali eskaliramo. Pravzaprav je koristno, če je ta seznam urejen po času, ko omenjeni rok poteče; podprogram Utrip gre potem lahko od začetka tega seznama po zapisih, ki jim je rok ravnokar potekel, in tiste uporabnike bodisi pokliče še drugič bodisi pokliče njihovega šefa (če smo uporabnika že poklicali dvakrat).

Ko se prijavi nov uporabnik, ga dodamo v množico prisotnih, ni pa ga nujno brisati iz seznama ljudi, na katere čakamo (če je bil na njem); lažje je, če Utrip preprosto ignorira zapise za tiste ljudi, ki so se medtem že pojavili na sestanku.

Naloga za precej podrobnosti ne pove natančno, kako naj se naš program obnaša glede njih. Na primer: recimo, da pokličemo uporabnika A; ta ne pride; čez deset minut pokličemo njegovega šefa B; kmalu zatem pride A; ali naj, ko mine pet minut od trenutka, ko smo poklicali B-ja, pokličemo le-tega še enkrat ali naj ga pustimo pri miru, saj ga pravzaprav ne potrebujemo več, ker je medtem A prišel na sestanek (in smo B-ja prej klicali le zato, ker A-ja ni bilo)? — In podobno: recimo, da imata C in D šefa E; pokličemo C-ja in dve minuti za tem še D-ja; nobeden od njiju ne pride; po desetih minutah pokličemo C-jevega šefa E; ali naj dve minuti kasneje pokličemo E-ja še enkrat, ker je tudi D-jev šef in ker se tudi D v desetih minutah, odkar smo ga poklicali, še ni odzval? In ali se odgovor na to vprašanje kaj spremeni, če med časom, ko smo prvič poklicali C-ja in prvič D-ja, ne mineta le dve minuti, pač pa pol ure (predpostavimo, da se E v tem času ni pojavil)?

V naši spodnji rešitvi se držimo načela, da če imamo nekega uporabnika v vrsti, kjer čakamo na to, da se bo bodisi pojavil na sestanku bodisi ga bo treba poklicati drugič ali pa eskalirati na njegovega šefa, ga v tem času ne bomo dodatno klicali (razen običajnega drugega klica pet minut po prvem). Če pa po desetih minutah eskaliramo na njegovega šefa, potem uporabnika samega ne bo več v vrsti in ga lahko kasneje pokličemo ponovno, če se bo pojavila potreba po tem. Tudi se ne trudimo ugotavljati, ali morda nekega šefa ne potrebujemo več (in ga ni treba klicati drugič ali celo eskalirati na *njegovega* šefa), ker se je njegov podrejeni medtem že pojavil na sestanku.

```
#include <queue>
#include <unordered_set>
using namespace std;

extern void Poklici(int uporabnik);
extern int Sef(int uporabnik);

struct Dogodek { int uporabnik, cas; bool eskalacija; };
queue<Dogodek> vrsta;
unordered_set<int> prijavljeni, vVrsti;
int cas = 0;

void Prijava(int uporabnik)
{
  prijavljeni.emplace(uporabnik);
```

```
}

void Dodaj(int uporabnik)
{
  // Morda je ta uporabnik že na sestanku.
  if (prijavljeni.find(uporabnik) != prijavljeni.end()) return;

  // Morda smo ga že klicali in čakamo na odziv, pa ga ni treba klicati še enkrat.
  if (vVrsti.find(uporabnik) != vVrsti.end()) return;

  // Sicer ga pokličimo zdaj.
  Poklici(uporabnik);

  // V vrsto zapišimo, da ga bo treba čez pet minut poklicati še enkrat.
  vrsta.push({uporabnik, cas + 5 * 60, false});
  vVrsti.emplace(uporabnik);
}

void Utrip()
{
  ++cas;
  // Morda je čas, da obdelamo kak dogodek iz vrste.
  while (! vrsta.empty() && vrsta.front().cas <= cas)
  {
    auto D = vrsta.front(); vrsta.pop();

    // Morda se je ta uporabnik medtem že prijavil.
    if (prijavljeni.find(D.uporabnik) != prijavljeni.end()) {
      vVrsti.erase(D.uporabnik); continue; }

    // Morda je čas za drugi klic.
    if (! D.eskalacija) {
      Poklici(D.uporabnik);

      // V vrsto zapišimo, da bo treba čez pet minut eskalirati.
      vrsta.push({D.uporabnik, cas + 5 * 60, true}); continue; }

    // Sicer je čas za eskalacijo.
    vVrsti.erase(D.uporabnik);
    Dodaj(Sef(D.uporabnik));
  }
}
```

2. Mehurčki

Ko se nekdo okuži, morajo iti v karanteno ne le vsi iz njegovega mehurčka, ampak tudi vsi, ki so bili v (izrednem) stiku s kom iz tega mehurčka, nato vsi iz *njihovih* mehurčkov in tako naprej. Mehurčke si lahko predstavljamo kot točke grafa, izredne stike pa kot povezave med njimi: povezava med mehurčkoma obstaja, če obstaja vsaj en izredni stik med kakšnim človekom iz enega in kakšnim človekom iz drugega mehurčka. Če se okuži nekdo iz mehurčka u, morajo iti v karanteno vsi ljudje iz vseh mehurčkov, ki so v našem grafu dosegljivi iz u; z drugimi besedami, iz vseh mehurčkov, ki pripadajo isti povezani komponenti kot u. Koristno je torej, če v grafu najprej poiščemo povezane komponente, potem pa bomo zlahka odgovorili na poljubno poizvedbo. Pri tem je koristno imeti še razpršeno tabelo, ki nam bo za vsakega človeka povedala, kateremu mehurčku pripada, in nato še eno, ki bo za vsak mehurček povedala, kateri povezani komponenti pripada; za vsako komponento pa si moramo pripraviti seznam mehurčkov v njej. (Seznam ljudi v vsakem mehurčku pa

dobimo tako ali tako že kot vhodni podatek.) Oglejmo si implementacijo te rešitve v C++:

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

void Mehurcki(const vector<vector<string>>& mehurcki,
              const vector<pair<string, string>> &dodatni,
              const vector<string> &poizvedbe)
{
  // Najprej pripravimo razpršeno tabelo, ki preslika imena ljudi v številke mehurčkov.
  unordered_map<string, int> kjeOseba;
  int n = mehurcki.size();
  for (int u = 0; u < n; ++i) for (const auto &ime : mehurcki[u])
    kjeOseba.emplace(ime, u);

  // Izračunajmo povezane komponente mehurčkov, ki nastanejo zaradi dodatnih stikov.
  // Najprej pripravimo sezname sosedov vsakega mehurčka v grafu.
  vector<vector<int>> sosedje(n);
  for (const auto &[ime1, ime2] : dodatni) {
    int u = kjeOseba.at(ime1), v = kjeOseba.at(ime2);
    sosedje[u].emplace_back(v); sosedje[v].emplace_back(u); }

  // Določimo komponente z iskanjem v širino.
  vector<vector<int>> komp;       // komp[i] = seznam mehurčkov v komponenti i
  vector<int> kjeMeh(n, -1);      // kjeMeh[u] = komponenta, ki ji pripada mehurček u
  for (int u = 0; u < n; ++u) if (kjeMeh[u] < 0) {
    int stKomp = komp.size(); komp.emplace_back(); auto &K = komp.back();
    // Poglejmo, kateri mehurčki so dosegljivi iz u. Seznam teh mehurčkov
    // bomo pripravili v K, ki ga spotoma uporabljamo tudi kot vrsto
    // mehurčkov, ki jih moramo še pregledati (to so vsi od K[glava] naprej).
    kjeMeh[u] = stKomp; K.emplace_back(u); int glava = 0;
    while (glava < K.size()) {
      int v = K[glava++];
      for (int w : sosedje[v]) if (kjeMeh[w] < 0) {
        kjeMeh[w] = stKomp; K.emplace_back(w); } } }

  // Odgovorimo na poizvedbe.
  for (const string &okuzeni : poizvedbe)
  {
    cout << "Ljudje, ki morajo v karanteno, če se okuži " << okuzeni << ":";
    // V karanteno morajo vsi ljudje iz vseh tistih mehurčkov,
    // ki so v isti povezani komponenti kot mehurček okužene osebe.
    for (int i : komp[kjeMeh[kjeOseba.at(okuzeni)]])
      for (const auto &s : mehurcki[i]) cout << " " << s;
    cout << endl;
  }
}
```

3. Pobeg iz močvare

(*a*) Lahko se zgodi, da do posameznega polja obstaja več poti, ki se razlikujejo po številu skokov in tudi po količini energije, ki jo ima raziskovalec na koncu poti. Ni

vnaprej očitno, katera od teh poti je najprimernejša; na primer, če gledamo poti, ki se končajo v (w, h), nas vsekakor zanima tista z najmanj skoki; če pa gledamo poti, ki se končajo v nekem drugem polju (x, y), bo morda pot z najmanj skoki pustila raziskovalcu premalo energije, da bi od tam sploh lahko nadaljeval pot do spodnjega desnega kota mreže, zato bi morali do (x, y) priti po neki drugi poti, ki ima sicer več skokov, vendar raziskovalcu pusti več energije.

Vprašanje, ki ga moramo pri tej nalogi res reševati, torej ni toliko „kakšno je najmanjše število skokov, s katerim lahko dosežemo polje (x, y)?“, pač pa „kakšna je največja energija, s katero lahko v s skokih dosežemo polje (x, y)?“. Odgovoru na to vprašanje recimo $f(x, y, s)$. Tega načeloma ni težko računati z rekurzivnim razmislekom: da pridemo do (x, y) v s skokih, smo morali najprej priti do nekega drugega polja (x', y') v $s - 1$ skokih in nato od tam skočiti na (x, y). Pred tem zadnjim skokom smo torej imeli (če smo si dotedanji potek poti izbrali optimalno) $f(x', y', s - 1)$ energije; če je to manjše od $|x - x'| + |y - y'|$, potem tisti zadnji skok na (x, y) sploh ne bo mogoč; sicer pa bo mogoč in bomo po pristanku in pitju napoja imeli $f(x', y', s-1) - |x-x'| - |y-y'| + p_{xy}$ energije. Prejšnji položaj (x', y') si moramo seveda izbrati tako, da bo ta nova količina energije čim večja. Tako smo dobili:

$$f(x, y, s) = \max\{\ f(x', y', s - 1) - |x - x'| - |y - y'| + p_{xy} : \\ 1 \le x' \le x,\ 1 \le y' \le y,\ (x, y) \ne (x', y'), \\ f(x', y', s - 1) \ge |x - x'| + |y - y'|\}.$$

Robni primer je $s = 0$, ko je $f(1, 1, 0) = p_{11}$ in $f(x, y, 0) = -\infty$ za vsa ostala polja $(x, y) \ne (1, 1)$. Funkcijo f bi se dalo računati sistematično po naraščajočih s, pri vsakem s pa po vseh poljih (x, y) naše močvare, dokler ne bi pri nekem s dobili $f(w, h, s) \ge 0$; tisto je potem najmanjši s, pri katerem obstaja pot od zgornjega levega do spodnjega desnega kota.[40]

Slabost te rešitve je, da moramo pri izračunu $f(x, y, s)$ pregledati $O(wh)$ možnih vrednosti (x', y'), torej položajev pred zadnjim skokom; časovna zahtevnost te rešitve je zato kar $O(w^2h^2d)$, kjer je d najmanjše potrebno število skokov. Ker se pri vsakem skoku premaknemo vsaj malo dol ali vsaj malo v desno, je $d \le w + h - 2$, tako da je časovna zahtevnost $O(w^2h^2(w + h))$. Če pri kakšnem s že prej opazimo, da so vse $f(x, y, s) < 0$, se lahko ustavimo, saj vemo, da poti s s ali več skoki sploh niso mogoče (in če doslej nismo našli poti do (w, h), je tudi v bodoče ne bomo).

Boljšo rešitev dobimo, če si skok predstavljamo kot sestavljenega iz zaporedja drobnih premikov za eno enoto desno ali dol; vsak tak premik nas stane tudi eno enoto energije, dovoljen pa je le, če smo pred tem premikom imeli vsaj eno enoto energije. Paziti pa moramo na to, da ne popijemo napoja na vsakem polju, na katero se na ta način premaknemo; napoj moramo popiti samo na koncu skoka. V opis stanja torej dodajmo še parameter ℓ, ki pove, ali trenutno letimo ($\ell = 1$) ali smo na tleh ($\ell = 0$). Naj bo torej $g(x, y, s, \ell)$ največja količina energije, s katero

[40] Tu smo predpostavili, da je pot do (w, h) veljavna le, če nam po zadnjem skoku na (w, h) in pitju napoja na tem polju ostane ≥ 0 energije. Nalogo bi se načeloma dalo razumeti tudi tako, da je dovolj že, če smo imeli dovolj energije za skok na (w, h), četudi nam je potem po prištevanju p_{wh} (ki je lahko negativen) energija morda padla pod 0; če bi se odločili za to razumevanje naloge, bi morali naš pogoj $f(w, h, s) \ge 0$ pač spremeniti v $f(w, h, s) \ge p_{wh}$ in podobno tudi drugod kasneje v našem opisu rešitve.

lahko pridemo na polje (x, y), dotlej izvedemo s skokov v celoti (zadnji, morebiti še nedokončan skok pri tem ne šteje) in trenutno letimo oz. smo na tleh, odvisno od parametra ℓ. Tako smo dobili:

$$\begin{aligned} g(x,y,s,0) &= g(x,y,s-1,1) + p_{xy} \\ g(x,y,s,1) &= \max\{\ g(x',y',s,\ell) - 1 \ :\ \ell \in \{0,1\}, \\ &\qquad\qquad (x',y') \in \{(x-1,y),(x,y-1)\}, \\ &\qquad\qquad g(x',y',s,\ell) > 0\ \}. \end{aligned}$$

Prva formula torej odraža dejstvo, da ob pristanku (ko se ℓ spremeni iz 1 v 0) popijemo napoj in povečamo število skokov; druga formula pa odraža premike za eno enoto dol ali desno med letom (ko je $\ell = 1$; sem štejemo tudi prvi premik na začetku leta, ko se ℓ spremeni iz 0 v 1). Robni primer je $s = 0$, ko imamo $g(1,1,0,0) = p_{11}$ in $g(x,y,0,0) = -\infty$ za vse ostale $(x,y) \neq (1,1)$.

Tudi funkcijo g lahko računamo sistematično po naraščajočih s. Najprej iz rezultatov za $g(\cdot,\cdot,s-1,1)$ izračunamo vse $g(\cdot,\cdot,s,0)$, nato pa iz teh izračunamo vse $g(\cdot,\cdot,s,1)$; to slednje moramo početi po naraščajočih x in y, saj se rezultat pri (x,y) opira na tista pri $(x-1,y)$ in $(x,y-1)$. Ko pri nekem s opazimo, da je $g(w,h,s,0) \geq 0$, se ustavimo in vemo, da je trenutni s ravno najmanjše potrebno število skokov. Podobno se seveda ustavimo tudi, če pri kakšnem s opazimo, da so vse $g(x,y,s,0) < 0$. Časovna zahtevnost te rešitve je le še $O(whd)$, torej v najslabšem primeru $O(wh(w+h))$.

Opozorimo še na to, da imata obe rešitvi prostorsko zahtevnost le $O(wh)$, saj lahko rezultate sproti pozabljamo: ko računamo rezultate za s, potrebujemo tiste za $s-1$, ne pa več tistih za $s-2$, $s-3$ in tako naprej.

(b) Razmislimo zdaj o različici, pri kateri smejo iti skoki v poljubno smer, ne le dol in desno. Rešitev s funkcijo f, ki smo jo videli pri podnalogi (a), lahko brez težav prilagodimo za (b), le v formuli za $f(x,y,s)$ moramo pogoja $1 \leq x' \leq x$ in $1 \leq y' \leq y$ zamenjati z $1 \leq x' \leq w$ in $1 \leq y' \leq h$. Raje pa bi seveda uporabili rešitev s funkcijo g, ki je učinkovitejša; toda če bomo dosedanjo formulo za $g(x,y,s,1)$ naivno popravili preprosto tako, da bomo za (x',y') poleg dosedanjih dveh možnosti $(x-1,y)$ in $(x,y-1)$ dovolili tudi dve novi možnosti $(x+1,y)$ in $(x,y+1)$, bo tako popravljena formula omogočala skoke, kjer leti raziskovalec npr. malo levo in malo desno ter pristane na polju, s katerega je skočil — to pa naloga izrecno prepoveduje.

Namesto tega lahko dovolimo, da ima parameter ℓ vrednosti $\{0, \searrow, \nearrow, \nwarrow, \swarrow\}$, pri čemer zadnje štiri povedo, v kateri dve smeri se pri trenutnem skoku leti: $\searrow$ pomeni dol in desno (kar je enako kot $\ell = 1$ pri rešitvi prejšnje različice naloge), $\nearrow$ pomeni gor in desno itd.

$$\begin{aligned} g(x,y,s,0) &= p_{xy} + \max\{g(x,y,s-1,\ell) : \ell \neq 0\} \\ g(x,y,s,\searrow) &= \max\{\ g(x',y',s,\ell) - 1 \ :\ \ell \in \{0,\searrow\}, \\ &\qquad\qquad (x',y') \in \{(x-1,y),(x,y-1)\}, \\ &\qquad\qquad g(x',y',s,\ell) > 0\ \}. \end{aligned}$$

Prva formula zdaj pravi, da lahko pristanemo po letu iz katerekoli kombinacije smeri (katerekoli vrednosti ℓ razen $\ell = 0$) in da bomo med temi možnostmi seveda vzeli tisto z največ energije. Druga formula je čisto taka kot tista za $\ell = 1$ pri različici (a), le ta imamo tukaj $\ell = \searrow$. Poleg nje bi morali zapisati še tri druge zelo podobne,

ki bi se razlikovale le po tem, kaj bi dovolile za (x', y'); na primer, tista za $\ell = \nearrow$ bi za (x', y') dovolila $(x-1, y)$ in $(x, y+1)$, kar odraža dejstvo, da lahko pri $\ell = \nearrow$ letimo le desno in gor.

Tako še vedno porabimo $O(wh)$ časa za izračun $g(\cdot,\cdot,s,\cdot)$ za vsa polja mreže in časovna zahtevnost celotne rešitve je še vedno $O(whd)$, kjer je d potrebno število skokov pri optimalni rešitvi (o tem, kolikšen je d v najslabšem primeru, bomo razmislili malo kasneje). Različica (b) zagotavlja, da taka pot obstaja, tako da lahko preprosto računamo $g(\cdot,\cdot,s,\cdot)$ za vse večje s in se ustavimo, ko pri nekem s prvič dobimo $g(w,h,s,0) \geq 0$.

(c) Ta različica naloge je taka kot (b), le da ne vemo, če primerna pot do (w,h) sploh obstaja. Kako naj ugotovimo, kdaj smemo pri računanju funkcije $g(\cdot,\cdot,s,\cdot)$ za vse večje s odnehati in zaključiti, da iskane poti sploh ni? Mislimo si funkcijo $\hat{g}(x,y,s,\ell)$, ki je definirana čisto tako kot g, le da besedno zvezo „dotlej izvedemo s skokov“ iz definicije g-ja spremenimo v „dotlej izvedemo kvečjemu s skokov“. Formul, ki smo jih prej izpeljali za računanje funkcije g, ne bi bilo težko prilagoditi za $\hat{g}$; še lažje pa je, če računamo g enako kot doslej, po naraščajočem s, nato pa pri vsakem s izračunamo še $\hat{g}$ po formuli $\hat{g}(x,y,s,\ell) = \max\{\hat{g}(x,y,s-1,\ell), g(x,y,s,\ell)\}$.

Če pri nekem s opazimo, da so vrednosti funkcije $\hat{g}$ pri s vse enake tistim pri $s-1$, to pomeni, da se v zadnjem skoku ni nič spremenilo in da se tudi v bodoče ne bo; če doslej nismo našli poti do (w,h), je tudi v bodoče ne bomo in lahko odnehamo. Če pa se to ne zgodi, torej če se pri vsakem povečanju s-ja spremeni vsaj ena od vrednosti $\hat{g}(x,y,\cdot,\ell)$, to pomeni, da se je tista vrednost morala povečati (kajti iz definicije $\hat{g}$ je jasno, da se lahko s povečevanjem s-ja vrednost $\hat{g}(x,y,s,\ell)$ le povečuje ali ostaja enaka, ne more pa se zmanjšati); sčasoma se torej neizogibno ena od vrednosti $\hat{g}(x,y,s,\ell)$ poveča na $w+h-2$, takrat pa imamo dovolj energije, da bomo gotovo lahko dosegli polje (w,h), torej bomo rešitev že z naslednjim skokom gotovo našli. Tako smo dobili naslednji postopek:

za vse x, y, ℓ izračunaj $g(x,y,0,\ell)$ po formulah za robni primer
  in si zapomni tudi $\hat{g}(x,y,0,\ell) = g(x,y,0,\ell)$;
$s := 0$;
while true:
  (* *Izvedimo naslednji skok.* *)
  zdaj poznamo vse $g(\cdot,\cdot,s,0)$; po formulah iz (b),
    izračunaj vse $g(\cdot,\cdot,s,\ell)$ za $\ell \in \{\searrow, \nearrow, \nwarrow, \swarrow\}$;
  $s := s+1$; za vse x, y izračunaj $g(x,y,s,0)$ iz $g(x,y,s-1,\ell)$;
  (* *Ali smo našli rešitev?* *)
  if $g(w,h,s,0) \geq 0$ **then return** s;
  (* *Ali se je $\hat{g}$ nehala spreminjati?* *)
  za vse x, y izračunaj $\hat{g}(x,y,s,0)$ iz $\hat{g}(x,y,s-1,0)$ in $g(x,y,s,0)$;
  if je povsod veljalo $\hat{g}(x,y,s,0) = \hat{g}(x,y,s-1,0)$ **then return** ∞;
  pozabi vse vrednosti $g(\cdot,\cdot,s-1,\cdot)$ in $\hat{g}(\cdot,\cdot,s-1,\cdot)$;

Naš postopek torej vrne število potrebnih skokov, če pa iskana pot ne obstaja, pokaže to tako, da vrne vrednost ∞.

Razmislimo še o tem, koliko skokov potrebuje pri (b) in (c) najkrajša pot v najslabšem primeru. Recimo, da je najkrajša pot do spodnjega desnega kota dolga

vsaj wh skokov; pri prvih wh skokih na poti se torej neizogibno zgodi, da se neko polje pojavi dvakrat. Naj bo u_0 prvo polje, ki se pojavi dvakrat; recimo, da se drugič pojavi k skokov kasneje; zaporedje polj, ki jih na teh k skokih obiščemo, naj bo $u_0, u_1, u_2, \ldots, u_k$, kjer je $u_k = u_0$. Naj bo e_i količina energije, ki smo jo imeli ob pristanku na u_i; naj bo p_i vrednost napoja na polju u_i; in naj bo d_i manhattanska razdalja med poljema u_{i-1} in u_i. Potem za vsak $i = 1, \ldots, n$ velja $e_i = e_{i-1} + p_{i-1} - d_i$. Ob pristanku na u_k — kar v resnici pomeni ob drugem pristanku na u_0 — smo imeli gotovo več energije kot ob prvem pristanku na u_0, kajti drugače bi lahko teh k skokov iz poti preprosto pobrisali in imeli še vedno veljavno, vendar krajšo pot do do spodnjega desnega kota; to bi bilo protislovje, saj smo že na začetku vzeli najkrajšo táko pot. Tako je torej energija ob drugem pristanku na u_0 večja kot ob prvem: $e_k > e_0$. Ta pogoj je, ker velja $e_k = e_0 + \sum_{i=0}^{k-1} p_i - \sum_{i=1}^{k} d_i$, enakovreden pogoju $\sum_{i=0}^{k-1} p_i > \sum_{i=1}^{k} d_i$.

Ali je mogoče, da bi pri vsakem $i = 1, \ldots, k$ veljalo $p_{i-1} + p_i \leq 2d_i$? Potem bi lahko to sešteli po vseh i in dobili (upoštevajmo še, da je $p_0 = p_k$) $2\sum_{i=0}^{k-1} p_i \leq 2\sum_{i=1}^{k} d_i$, to pa bi bilo v protislovju z neenakostjo, ki smo jo dobili na koncu prejšnjega odstavka. Torej mora obstajati neki i, pri katerem je $p_{i-1} + p_i > 2d_i$. Ker je bila prvotna pot veljavna, je naša energija pri vsakem pristanku nenegativna, torej tudi $e_{i-1} \geq 0$; po pristanku na $i-1$ smo popili napoj e_{i-1} in imeli dovolj energije za skok dolžine d_i, torej je $e_{i-1} + p_{i-1} \geq d_i$; skok je porabil d_i energije, po pristanku na u_i smo spili napoj p_i in imeli potem $e_{i-1} + p_{i-1} - d_i + p_i$ energije; to pa je (ker je $p_{i-1} + p_i > 2d_i$) naprej $> e_{i-1} + d_i > d_i$, torej imamo dovolj energije tudi za skok nazaj na u_{i-1}. Če skočimo tja nazaj, bo naša energija ob pristanku znašala $e_{i-1} + p_{i-1} - d_i + p_i - d_i > e_{i-1}$. Tako smo z dvema skokoma, z u_{i-1} na u_i in nazaj na u_{i-1}, pridobili vsaj eno enoto energije. Takšna dva skoka lahko zdaj ponavljamo in z največ $w + h - 2$ pari skokov naša energija doseže $w + h - 2$, kar pa gotovo zadošča za to, da potem v enem skoku dosežemo polje (w, h). Tako torej gotovo obstaja do tega polja pot dolžine kvečjemu $wh + 2(w + h - 2) = O(wh)$.

Tako smo dobili zgornjo mejo za dolžino najkrajše poti. Ker porabi glavna zanka naše rešitve za vsak dodaten skok po $O(wh)$ časa, je časovna zahtevnost celotnega postopka v najslabšem primeru $O(w^2h^2)$.

4. Barvanje zebre

Trenutno stanje daljice lahko predstavimo z urejenim seznamom intervalov, ki so izmenično črni in beli. Ti intervali naj se ne prekrivajo in naj vsi skupaj pokrijejo celotno daljico $[0, D]$. Na primer, če je $D = 10$ in smo izvedli tri operacije: barvanje $[1, 3]$ z belo, nato barvanje $[2, 8]$ z belo in nato barvanje $[4, 5]$ s črno, bi moral naš seznam takrat povedati, da je interval $[0, 1]$ zdaj črn, $[1, 4]$ bel, $[4, 5]$ črn, $[5, 8]$ bel in $[8, 10]$ črn. Če bomo znali vzdrževati tak seznam — recimo mu S —, potem tudi ne bo težko vzdrževati skupne dolžine belih intervalov in jo po vsaki operaciji izpisati.

Razmislimo torej, kaj je treba v seznamu spremeniti, ko pride nova operacija: barvanje intervala $[\ell, d]$ z barvo c. Če ℓ še ni krajišče kakšnega od intervalov v S, to pomeni, da leži v notranjosti nekega intervala; ta interval pri ℓ razbijmo na dva dela. Enako naredimo tudi z d. Zdaj je torej v S eden ali več zaporednih intervalov, ki skupaj pokrijejo natanko območje $[\ell, d]$; če jih je več, jih pobrišimo in jih zamenjajmo z enim samim intervalom za celotno $[\ell, d]$. Zdaj imamo v S torej interval $[\ell, d]$ in

moramo zanj le še označiti, da je barve c. (Po tistem lahko celo pogledamo, če je interval $[\ell, d]$ morda enake barve kot prejšnji in/ali naslednji interval v S, in če je res tako, lahko taka dva sosednja intervala združimo; ni pa nujno, da to res počnemo.)

Ko spreminjamo seznam S, lahko tudi vzdržujemo skupno dolžino belih intervalov v njem; recimo ji B. Ko pobrišemo interval bele barve iz S, moramo vrednost B zmanjšati za dolžino tega intervala; in ko intervalu $[\ell, d]$ v S spremenimo barvo iz črne v belo, moramo B povečati za $d - \ell$.

Kakšna je časovna zahtevnost te rešitve? V vsaki operaciji se število intervalov poveča za največ 2 (ker morda razbijemo neki interval pri ℓ in nekega pri d), nato pa se lahko tudi še zmanjša. Ker je torej skupno število dodajanj intervalov po n operacijah lahko največ $2n$, je tudi skupno število brisanj intervalov po vseh operacijah skupaj lahko največ $2n$; in v seznamu S je lahko hkrati največ $2n + 1$ intervalov. Po posamezni operaciji lahko torej v $O(n)$ časa pregledamo seznam S in po potrebi vrinemo ali pobrišemo nekaj intervalov iz njega; za vseh n operacij skupaj bomo tako porabili $O(n^2)$ časa.

Boljšo rešitev pa dobimo, če S namesto s seznamom predstavimo s primerno uravnoteženim drevesom (na primer rdeče-črnim). Iskanje intervala, ki vsebuje ℓ ali d, nam v takem drevesu vzame $O(\log n)$ časa, enako tudi dodajanje ali brisanje intervala. Ker imamo vsega skupaj $O(n)$ iskanj, $O(n)$ dodajanj in $O(n)$ brisanj, bo časovna zahtevnost te rešitve le $O(n \log n)$.

5. Prevoz po mreži

Vpeljimo v mrežo koordinatni sistem, pri čemer meri x-koordinata razdaljo od levega roba mreže (od 0 do w), y-koordinata pa od zgornjega roba (od 0 do h). Potem ima j-ta celica v i-ti vrstici oglišča $(j-1, i-1)$, (j, i), $(j, i-1)$ in $(j-1, i)$. Cesta v orientaciji \ povezuje prvi dve od teh oglišč, v orientaciji / pa drugi dve.

Opazimo lahko, da za diagonalno nasprotni oglišči celice velja, da ima vsota x- in y-koordinate pri obeh enako parnost. Ker se naša pot po mreži začne v zgornjem levem kotu, to je na koordinatah $(0, 0)$, kjer je vsota koordinat soda, to pomeni, da bo tudi po vsakem nadaljnjem koraku na tej poti vsota koordinat morala biti soda. Pot naj bi se končala v spodnjem desnem kotu mreže, na koordinatah (w, h); če je torej vsota $w + h$ liha, je naloga sploh nerešljiva.

Recimo torej zdaj, da je $w + h$ soda (ali z drugimi besedami, da sta w in h enake parnosti). Dosegljive točke so torej načeloma vse tiste (x, y), pri katerih je $x + y$ soda. Lahko si jih predstavljamo kot točke grafa; povezave v tem grafu pa naj obstajajo med (x, y) in $(x \pm 1, y \pm 1)$, vendar le pod pogojem, da ima cesta med takima dvema točkama nosilnost vsaj k. Takšni povezavi pripišimo dolžino 0, če ima celica, po kateri povezava poteka, že v začetnem stanju pravo orientacijo; če pa jo je treba prej še zasukati, naj ima ta povezava dolžino 1.

V tako dobljenem grafu poiščimo najkrajšo pot od $(0, 0)$ do (w, h); dolžina te poti je ravno najmanjše število celic, ki jim je treba spremeniti orientacijo. Ker imajo vse povezave dolžino 0 ali 1, lahko najkrajšo pot iščemo z rahlo prilagojenim iskanjem v širino. Zapišimo ta postopek s psevdokodo:

```
1  for y := 0 to h do for x := 0 to w do d[x, y] := ∞;
2  d[0, 0] := 0; Q := prazen seznam; dodaj (0, 0) v Q;
3  while Q ni prazen:
```

```
4    (x, y) := prvi element seznama Q; pobriši ga iz Q;
5    for △_x ∈ {1, −1} do for △y ∈ {1, −1}:
6      x′ := x + △_x; y′ := y + △_y;
7      if x′ < 0 or x′ > w or y′ < 0 or y′ > h then continue;
8      i := max{x, x′}; j := max{y, y′}; if c_ij < k then continue;
9      if △_x = △_y then O := \ else O := /;
10     if o_ij = O then c := 0 else c := 1;
11     if d[x, y] + c ≥ d[x′, y′] then continue else d[x′, y′] := d[x, y] + c;
12     if c = 0 then dodaj (x′, y′) na začetek Q else dodaj (x′, y′) na konec Q;
13   return d[w, h];
```

Namesto vrste kot pri običajnem iskanju v širino imamo tu seznam Q, kjer lahko elemente dodajamo tako na koncu kot na začetku (brisali pa jih bomo le na začetku). V tabeli d hranimo za vsako točko dolžino najkrajše doslej znane poti do nje; na začetku poznamo pot le do začetne točke $(0, 0)$. Na vsakem koraku glavne zanke vzamemo iz Q eno od točk z najmanjšo $d[x, y]$ (med vsemi točkami v Q) in pregledamo njene štiri sosede (x', y'); za vsako sosedo najprej preverimo, če sploh obstaja (če ni zunaj mreže; vrstica 7); v vrstici 8 izračunamo koordinati celice, po kateri gre korak iz (x, y) v (x', y'), in preverimo, če ima cesta v tej celici dovolj veliko nosilnost; v vrstici 9 izračunamo orientacijo tega koraka; če celica še nima te orientacije, je dolžina povezave med (x, y) in (x', y') enaka 1, sicer pa 0 (vrstica 10); če do (x', y') že poznamo enako dobro ali boljšo pot kot pravkar odkrito pot skozi (x, y), se nam s točko (x', y') tu ni treba še enkrat ukvarjati (vrstica 11), sicer pa jo dodamo v Q — na začetek, če je $d[x', y'] = d[x, y]$, oz. na konec, če je $d[x', y'] = d[x, y] + 1$ (vrstica 12).

Pri tem postopku se lahko zgodi, da isto točko dodamo v Q dvakrat — prvič z neko dolžino, drugič pa z za 1 krajšo dolžino. Toda ko jo bomo drugič vzeli iz Q, ne bomo izboljšali $d[x', y']$ za nobeno od njenih sosed, zato se takrat ne bo nič spremenilo, tako da od tega, da smo imeli točko (x, y) dvakrat v Q, ni bilo nobene velike škode. Časovna zahtevnost postopka je tako še vedno $O(wh)$.

6. Šolarkina uganka

Množico vhodnih točk imenujmo M. Pripravimo si še množico P vseh premic, ki vsebujejo po vsaj dve točki iz M; za vsako premico $p \in P$ naj bo *Točke*$[p]$ množica tistih točk iz M, ki ležijo na p; in za vsako točko $T \in M$ naj bo *Premice*$[T]$ množica tistih premic iz P, na katerih leži T. Premic je lahko največ $O(n^2)$ in posamezna premica lahko vsebuje $O(n)$ točk, toda posamezna točka lahko leži na največ $n - 1$ različnih premicah, zato je skupna dolžina omenjenih seznamov le $O(n^2)$.

```
P := {};
za vsako T ∈ M naj bo Premice[T] := {};
za vsako točko T ∈ M in za vsako U ∈ M, če U ≠ T:
  p := premica skozi T in U;
  če p še nimamo v P, jo tja dodajmo in inicializirajmo Točke[p] := {};
  dodaj T v Točke[p]; dodaj p v Premice[T];
```

Kako naj predstavimo premico p, da bomo čim lažje opazili, če bomo večkrat (pri različnih parih T in U) prišli do iste premice? Vemo, da lahko premico opišemo na

primer z enačbo $ax+by+c=0$; če je $c \neq 1$, jo delimo s c; odslej torej predpostavimo, da je $c \in \{0,1\}$; če je $c = 0 \neq a$, delimo enačbo še z a; če pa je $c = a = 0$, delimo enačbo z b. Po teh korakih je predstavitev premice enolična: to je trojica (a,b,c) oblike $(a,b,1)$ ali $(1,b,0)$ ali $(0,1,0)$. Ker so bile koordinate vhodnih točk cela števila, sta v tako dobljenih trojicah a in b racionalni števili in ju predstavimo kot okrajšana ulomka. Takšne trojice lahko potem brez težav primerjamo med sabo in jih uporabimo kot ključe v razpršeni tabeli P.

Naloga pravi, da lahko eno točko iz M premaknemo na nov položaj; to pa si lahko predstavljamo tudi tako, da smemo neko $T \in M$ pobrisati in namesto nje v M dodati neko novo točko U. Premicam, ki vsebujejo vsaj tri točke iz M, bomo rekli *ugodne;* naloga torej zahteva, naj maksimiziramo število ugodnih premic. Ko pobrišemo T iz M, se število ugodnih premic lahko zmanjša; tega, za koliko se zmanjša, ni težko ugotoviti: pojdimo v zanki po premicah iz $p \in \mathit{Premice}[T]$ in poglejmo, koliko od njih je vsebovalo natanko tri točke, torej koliko jih je imelo $|\mathit{Točke}[p]| = 3$. Te so bile prej ugodne, po brisanju T-ja pa vsebujejo le še dve točki in so neugodne. Naj bo torej $Z[T]$ število premic, ki prenehajo biti ugodne, ko pobrišemo T iz M. Pripravimo si tudi seznam, v katerem bodo vse točke iz M urejene naraščajoče po svoji $Z[T]$; to bo prišlo prav kasneje.

Pišimo še $M' := M - \{T\}$ in $M'' := M' \cup \{U\}$. Ko v M' dodamo novo točko U (in dobimo M''), se lahko število ugodnih premic kaj poveča. To se zgodi v primerih, ko je neka premica vsebovala natanko dve točki iz M', poleg tega pa vsebuje še U in tako zdaj vsebuje tri točke iz M''; s tem se je spremenila iz neugodne v ugodno. Premica, ki je vsebovala natanko dve točki iz M', pa je vsebovala tudi natanko dve točki iz M in jo torej imamo v množici P. Tako vidimo, da je smiselno U postaviti le na premice iz P, kajti drugače se število ugodnih premic gotovo ne bo povečalo.

(1) Ena možnost je, da postavimo U tako, da leži na natanko eni premici iz P, recimo $p \in P$. To ima smisel le, če p prej še ni bila ugodna, torej če vsebuje natanko dve točki iz M; recimo A_1 in A_2. Poleg tega ima to smisel le, če nobena od A_1 in A_2 ni enaka T, kajti v tem primeru bi p tudi po dodajanju U-ja še vedno imela le dve točki in ne bi bila ugodna. Poleg tega je smiselno za T izbrati le eno od takih točk, pri katerih je $Z[T] = 0$, kajti le tako se lahko število ugodnih premic poveča: z dodajanjem U-ja smo eno ugodno pridobili in če smo prej ob brisanju T-ja (vsaj) eno ugodno izgubili, ne bomo na koncu nič na boljšem kot na začetku.

za vsako $p \in P$, če je $|\mathit{Točke}[p]| = 2$:
  naj bosta A_1 in A_2 točki iz $\mathit{Točke}[p]$;
  če obstaja vsaj ena $T \in M$, ki je različna od A_1 in A_2 in ima $Z[T] = 0$,
    potem je možen scenarij ta, da pobrišemo T in dodamo poljubno tako U,
    ki leži na p in na nobeni drugi premici iz P; po tej spremembi je število
    ugodnih premic za 1 večje kot na začetku;

Da bomo lažje preverili, ali primerna T obstaja, si pomagajmo s seznamom, v katerem imamo točke urejene naraščajoče po $Z[T]$; v tem seznamu poiščimo prvo tako točko T, ki ni niti A_1 niti A_2 (treba bo torej pogledati največ prve tri točke s seznama); če niti tista nima $Z[T] = 0$, potem ga nima sploh nobena, saj so točke v seznamu urejene naraščajoče po $Z[T]$.

Vprašanje je še, kako si izbrati konkretno U tako, da bo ležala samo na p in ne še na kakšni drugi premici iz P. Kmalu bomo videli, da bomo tako ali tako morali

izračunati vsa presečišča med dvema ali več premicami iz P; potem je dovolj na primer to, da postavimo U na p tako, da leži dlje od koordinatnega izhodišča kot katerokoli presečišče dveh premic iz P — potem bo U gotovo ležala samo na p in ne še na kakšni drugi premici iz P.

(2) Druga možnost pa je, da postavimo U tako, da leži na več kot eni premici iz P hkrati; torej na presečišču dveh ali več takih premic. (To ima seveda smisel le, če na tem položaju nismo imeli točke že prej, torej če je $U \notin M$.) Recimo, da je U neko tako presečišče; naj bo P_U množica tistih premic iz P, ki gredo skozi U. (Opazimo lahko, da nobena točka $A \in M$ ne pripada dvema (ali več) premicama iz P_U, saj bi drugače tidve premici imeli dve skupni točki (A in U), potem pa to sploh ne bi bili dve različni premici.)

Če bi U dodali v M, bi pri tem na novo postale ugodne tiste premice iz P_U, ki so imele (v M) natanko dve točki; recimo, da je takih premic y. V resnici pa seveda U-ja ne dodajamo v M, ampak v M', torej v množico, iz katere smo prej pobrisali T. Kaj je zaradi tega drugače? Če neka premica $p \in P_U$ vsebuje točko T (taka premica je gotovo največ ena), se ji število točk ob premiku T-ja na U v resnici nič ne spremeni; če je imela p prej tri točke, jih ima zdaj še vedno, kar pomeni, da število premic, ki zaradi brisanja T-ja postanejo neugodne, v resnici ni $Z[T]$, pač pa le $Z[T]-1$. In podobno, če je imela p prej natanko dve točki, ju ima zdaj še vedno, kar pomeni, da število premic, ki zaradi dodajanja U-ja postanejo ugodne, v resnici ni y, pač pa le $y-1$.

Tako lahko torej ločimo naslednje podprimere:

(2.1) T lahko leži na kakšni taki premici $p \in P_U$, ki ima $|\mathit{Točke}[p]| = 2$. V tem primeru je novo število ugodnih premic (po premiku T-ja na U) za $y - 1 - Z[T]$ večje kot prej.

(2.2) T lahko leži na kakšni taki premici $p \in P_U$, ki ima $|\mathit{Točke}[p]| = 3$. V tem primeru je novo število ugodnih premic (po premiku T-ja na U) za $y - Z[T] + 1$ večje kot prej.

(2.3) Sicer pa je novo število ugodnih premic za $y - Z[T]$ večje kot prej.

Pri vsakem od teh podprimerov je seveda smiselno izbrati T tako, da bo imela čim manjšo $Z[T]$. Zapišimo ta del rešitve s psevdokodo:

postopek ObdelajPresečišče(U, P_U):
1 $M_2 :=$ prazna množica; $M_3 :=$ prazna množica; $y := 0$;
2 za vsako $p \in P_U$:
3 **if** $|\mathit{Točke}[p]| = 2$ **then** $M_2 := M_2 \cup \mathit{Točke}[p]$, $y := y + 1$
4 **else if** $|\mathit{Točke}[p]| = 3$ **then** $M_3 := M_3 \cup \mathit{Točke}[p]$;
5 $c_1 := y - 1 - \min\{Z[T] : T \in M_2\}$;
6 $c_2 := y + 1 - \min\{Z[T] : T \in M_3\}$;
7 v seznamu, kjer so $T \in M$ urejene naraščajoče po $Z[T]$, poišči
prvo tako, ki ni niti iz M_2 niti iz M_3;
8 če take T nismo našli, naj bo $c_3 := -\infty$, sicer $c_3 := y - Z[T]$;
9 zdaj vemo, da če primerno izberemo neko $T \in M$ in jo premaknemo na U,
je mogoče število ugodnih premic povečati za največ $\max\{c_1, c_2, c_3\}$;

Kakšna je časovna zahtevnost tega postopka za posamezno presečišče U? Število premic, ki se sekajo v U, označimo s $k := |P_U|$. Zanka v vrsticah 2–4 izvede torej k iteracij; v vrstici 3 dodamo skupaj največ $2k$ točk v M_2, v vrstici 4 pa skupaj največ

$3k$ točk v M_3; zato imamo tudi v vrsticah 5–6 po $O(k)$ dela, da poiščemo minimum $Z[T]$ po vseh T iz M_2 oz. M_3. V vrstici 7 bo treba pregledati največ $|M_2|+|M_3|+1$ točk iz seznama, preden bomo našli prvo tako, ki ni niti v M_2 niti v M_3 (če ne bo že prej konec seznama). Tako torej vidimo, da imamo s presečiščem U vsega skupaj $O(k)$ dela. Ker si lahko predstavljamo, da nam bo vzelo $O(k)$ časa že to, da to presečišče sploh najdemo oz. pripravimo množico P_U s premicami, ki se sekajo v njem, to pomeni, da gornji postopek časovne zahtevnosti (v asimptotičnem smislu) sploh ne bo nič povečal v primerjavi s samim iskanjem presečišč.

Kako pa naj sploh najdemo vsa možna presečišča U? Preprosta rešitev je z dvema gnezdenima zankama po vseh možnih parih premic; recimo, da imamo $m = |P|$ premic in jih oštevilčimo: $P = \{p_1, \ldots, p_m\}$. Potem naredimo takole:

for $i := 1$ **to** m:
  $h :=$ prazna razpršena tabela, v kateri bo ključ točka U,
    pripadajoča vrednost pa P_U;
  for $j := 1$ **to** m **do if** $j \neq i$:
    if sta p_i in p_j vzporedni **then continue**;
    $U :=$ presečišče premic p_i in p_j;
    if je v h že prisoten ključ U s pripadajočo vrednostjo P_U
    then dodaj j v to P_U
    else dodaj U v h s pripadajočo vrednostjo $P_U = \{i, j\}$;
  za vsako $U \in h$ s pripadajočo vrednostjo P_U:
    if $i = \min P_U$ **then** ObdelajPresečišče(U, P_U);

Pri vsaki premici p_i torej izračunamo njena presečišča z vsemi ostalimi in jih odlagamo v slovar oz. razpršeno tabelo h; tako bomo opazili, če se v isti točki sekata več kot dve premici. Če se v neki točki seka k premic, jo bomo načeloma našli k-krat (pri k različnih p_i); postopek ObdelajPresečišče pa je seveda dovolj klicati le prvič — temu je namenjen pogoj „**if** $i = \min P_U$“ v zadnji vrstici.

Ta postopek torej porabi $O(m^2)$ časa, kar je v najslabšem primeru lahko do $O(n^4)$, saj imamo načeloma lahko do $O(n^2)$ premic. To sicer ni nujno slabo, saj je tudi presečišč v najslabšem primeru $O(n^4)$. Obstajajo pa tudi postopki, kjer je časovna zahtevnost iskanja presečišč odvisna od števila najdenih presečišč, kar bi znalo biti koristno, če je presečišč znatno manj od $O(n^4)$. V računski geometriji je na primer znan Bentley-Ottmannov postopek s preletom ravnine (*plane sweep*), ki v $O((m+q)\log m)$ časa poišče vsa presečišča m daljic, pri čemer je q število najdenih presečišč. Tega postopka ne bi bilo težko prilagoditi za naš problem s premicami, pri čemer bi se celo nekoliko poenostavil: daljice je treba med preletom ravnine dodajati v podatkovno strukturo in brisati iz nje, ko se preletna premica (*sweep line*) premakne mimo enega od krajišč daljice; zato je ta struktura ponavadi rdečečrno drevo ali kaj podobnega; pri nas pa so premice prisotne ves čas preleta in jih ni treba brisati ali dodajati, zato lahko namesto drevesa uporabimo navaden seznam. Časovna zahtevnost naše rešitve se s tem zmanjša na $O((n^2+q)\log n)$; seveda pa je, če je q blizu n^4, to še vedno slabše od preproste rešitve z dvema gnezdenima zankama po vseh premicah.

Kakorkoli že, med vsemi možnostmi, ki smo jih našli pri (1) in pri (2), moramo na koncu vrniti tisto z največjim številom ugodnih premic.[41]

[41] Še zgodovinska opomba: ta naloga je posplošitev uganke Scholar's Puzzle, ki jo je objavil

7. Palindromska razbitja

Naloga je primerna za reševanje z dinamičnim programiranjem. Recimo, da je vhodni niz s dolg n znakov: $s = s[0{:}n]$. (V tej rešitvi bomo uporabljali pythonovski zapis za podnize in tudi znake niza bomo indeksirali od 0 do naprej.) Definirajmo f_k kot najmanjše število palindromov, na katere je mogoče razbiti niz $s[{:}k]$, torej niz, ki ga tvori prvih k znakov niza s; rezultat, po katerem sprašuje naloga, je potem f_n. Funkcijo f lahko računamo po naraščajočih k. Robni primer je $k = 0$, ko imamo prazen niz in je $f_0 = 0$. Pri večjih k pa razmišljajmo takole: v razbitju niza $s[{:}k]$ na palindrome mora biti neki palindrom zadnji; recimo, da je to $s[r{:}k]$; potem palindromi pred njim tvorijo razbitje niza $s[{:}r]$, za tega pa vemo, da je najmanjše število palindromov, na katere ga lahko razbijemo, enako $f(r)$. Tako smo dobili zvezo:

$$f_k = 1 + \min\{f_r : 0 \le r < k,\ \text{niz } s[r{:}k] \text{ je palindrom}\}.$$

Za preverjanje, ali je $s[r{:}k]$ palindrom, lahko uporabimo postopek s str. 110 letošnjega biltena, kjer smo za vsak možni položaj središča palindroma izračunali, kako dolg je najdaljši palindrom s središčem na tistem položaju; s tem ne bi bilo težko preveriti, ali je $s[r{:}k]$ palindrom ali pa je za to že predolg (glede na položaj svojega središča). Opišimo to rešitev s psevdokodo; recimo, da je najdaljši palindrom sode dolžine s središčem med znakoma $s[i-1]$ in $s[i]$ podniz $s[i-a_i{:}i+a_i]$, najdaljši palindrom lihe dolžine s središčem v znaku i pa naj bo podniz $s[i-b_i{:}i+b_i+1]$. Najprej bomo torej izračunali a_i in b_i za vse i, nato pa lahko izračunamo vse f_k (po naraščajočih k) po prej opisani formuli.

```
for i := 0 to n:
  a_i := 0; while i − 1 − a_i ≥ 0 and i + a_i < n and
                  s[i − 1 − a_i] = s[i + a_i] do a_i := a_i + 1;
  b_i := 0; while i − 1 − b_i ≥ 0 and i + b_i + 1 < n and
                  s[i − 1 − b_i] = s[i + b_i + 1] do b_i := b_i + 1;

f_0 := 0;
for k := 1 to n:
  f_k := k; (* Trivialna rešitev — s[:k] lahko razbijemo na posamezne črke. *)
  for r := 0 to k − 1:
    i := ⌊(k + r)/2⌋; (* Središče podniza s[r:k]. *)
    (* Ali je pri tem središču s[r:k] predolg, da bi bil palindrom? *)
    if (k − r) mod 2 = 0 and k − r > 2a_i then continue;
    if (k − r) mod 2 = 1 and k − r > 2b_i + 1 then continue;
    (* Tukaj vemo, da je s[r:k] palindrom. *)
    f_k := min{f_k, 1 + f_r};
return f_n;
```

Dobili smo rešitev s časovno zahtevnostjo $O(n^2)$.

Sam Loyd v svoji knjigi *Cyclopedia of Puzzles* (New York: The Lamb Publishing Co., 1914), str. 59. Mimogrede, z njegovimi ugankami smo se na naših tekmovanjih že srečali: na str. 106 omenjene knjige je uganka Back from the Klondike, katere posplošitev je istoimenska naloga na našem tekmovanju leta 2017 (prva naloga v tretji skupini; str. 25 v *Biltenu* 2017).

8. Stolp

Nalogo lahko rešujemo z dinamičnim programiranjem. V opis stanja vzemimo višino stolpa h (tega, kar je še ostalo od njega po dosedanjih brisanjih) in število kock vsake barve med najnižjimi k kockami (recimo c_b kock barve b, pri čemer je $b \in \{1, \ldots, B\}$). Za te kocke je namreč vseeno, v kakšnem vrstnem redu so, saj na možnosti brisanja to nič ne vpliva; za višje kocke pa vemo, da so še vedno v takem vrstnem redu kot v prvotnem stolpu, saj brisanja tistega dela stolpa še niso mogla prizadeti, zato je o njih vse povedano že z višino h in ni treba v opis podproblema dodajati še podatkov o barvi teh kock.

Imamo torej stanja oblike $(h, \mathbf{c})$ za $\mathbf{c} = (c_1, \ldots, c_B)$, opis začetnega razporeda kock v stolpu pa si lahko predstavljamo kot tabelo $s[1..n]$, kjer je $s[i]$ barva i-te najvišje kocke na skladu. Naj bo zdaj $f(h, \mathbf{c})$ najmanjše potrebno število operacij, da pridemo od tega stanja do praznega stolpa. Za vsako barvo b, če je $c_b > 0$, imamo potem možnost brisanja te barve; novo stanje dobimo takole:

postopek NovoStanje(staro stanje $(h, \mathbf{c})$, pobrisana barva b):
  $d := c_b$; $c_b := 0$;
  while $d > 0$:
    if $h > k$ **then** povečaj $c_{s[h-k]}$ za 1;
    $h := h - 1$; $d := d - 1$;
  return $(h, \mathbf{c})$;

Pobrišemo torej vse kocke barve b in potem pogledamo, kaj po novem pride od zgoraj med najnižjih k kock, ko se sklad spet sesede.

Med vsemi možnimi novimi stanji, ki jih dobimo na ta način (pri različnih b), vzamemo seveda tisto z najmanjšo f; prištejemo še 1, pa imamo f stanja pred brisanjem. Ker ima novo stanje manj kock kot staro, lahko vrednosti funkcije f računamo sistematično po naraščajočih h. Zapišimo ta postopek s psevdokodo:

  $f[0, \mathbf{0}] := 0$;
  for $h := 1$ **to** n:
    za vsako $\mathbf{c} = (c_1, \ldots, c_B)$, kjer je $c_1 + \ldots + c_B = \min\{k, h\}$:
      $f[0, \mathbf{c}] := \infty$;
      for $b := 1$ **to** B do if $c_b > 0$ **then**
        $f[0, \mathbf{c}] := \min\{f[0, \mathbf{c}], f[\text{NovoStanje}(h, \mathbf{c}, b)]\}$;

Rezultat, ki nas na koncu zanima, je $f(n, (c_1, \ldots, c_B))$, kjer je c_b število kock barve b med spodnjimi n kockami v začetnem stanju sklada. V zanki, ki je zgoraj opisana kot „za vsako $\mathbf{c}$“, lahko v praksi vse primerne $\mathbf{c}$ zgeneriramo bodisi z B gnezdenimi zankami ali pa si napišemo rekurziven podprogram (kar je bolj elegantno, ker bo brez sprememb programa delovalo za poljuben B).

Ramislimo še o časovni zahtevnosti te rešitve. Vektor nenegativnih celih števil $c_1, \ldots, c_B$, ki se seštejejo v k, si lahko predstavljamo tako, kot da imamo zaporedje k kroglic in mednje vrinemo $B - 1$ ograjic, tako da je c_1 kroglic pred prvo ograjico, c_2 med prvo in drugo in tako naprej. Tako dobimo zaporedje $k + B - 1$ kroglic in ograjic, v katerem lahko položaj $B - 1$ ograjic izberemo na $\binom{k+B-1}{B-1}$ načinov; toliko je torej možnih vektorjev $\mathbf{c}$. Pri najbolj notranji zanki (po b) imamo z izračunom novih stanj $O(B+c_b)$ dela za vsako možno barvo b, kar je skupaj $O(B^2+k)$. Časovna zahtevnost celotnega postopka je torej $O(n \cdot \binom{k+B-1}{B-1} \cdot (B^2 + k))$.

Še ena različica te rešitve je, da stanje spodnjih k mest v stolpu opišemo tako, da povemo, katere izmed prvotnih n kock so na teh spodnjih k mestih; tu je možnih torej $\binom{n}{k}$ stanj. To je manjše od $\binom{k+B-1}{B-1} = \binom{k+B-1}{k}$ (torej od števila možnih **c** pri prvotni rešitvi), če je $n < k + B - 1$.

9. Mediana

Za začetek si pripravimo tabelo parov $(a[i], i)$ za $i = 1, \ldots, n$ in jo uredimo po $a[i]$. Če je v tako urejeni tabeli na j-tem mestu par $(a[i], i)$, postavimo v tabeli a element $a[i]$ na j. Tako se medsebojni vrstni red elementov tabele a ne bo spremenil in odgovor na vsako poizvedbo bo še vedno isti element tabele a kot prej, le številska vrednost tega elementa je drugačna. (Zato je koristno, če si nekje zapomnimo tudi prvotno vsebino tabele a, da bomo lahko na koncu pri vsaki poizvedbi vrnili pravo vrednost.) Odslej bomo torej predpostavili, da so elementi tabele a cela števila od 1 do n.

Razdelimo v mislih tabelo a na $B \approx \sqrt{n}$ blokov, velikih po približno B elementov. Za vsak blok si pripravimo tudi kopijo bloka, v kateri so elementi bloka urejeni naraščajoče.

Recimo zdaj, da bi radi za poizvedbo (ℓ, d) izračunali mediano elementov $a[\ell..d]$. Če ležita indeksa ℓ in d oba znotraj istega bloka, si preprosto naredimo kopijo zaporedja $a[\ell..d]$ in v njem poiščimo mediano s postopkom quickselect, kar bo vzelo $O(B)$ časa. Če pa ležita ℓ in d v različnih blokih, razmišljajmo takole: na intervalu $a[\ell..d]$ je $k := d - \ell + 1$ elementov; mediana je najmanjše tako število, od katerega je manjših kvečjemu $k/2$ elementov z našega intervala; poiščemo jo lahko z bisekcijo:

funkcija MEDIANA(ℓ, d):
  $k := d - \ell + 1$; $m_1 := 0$; $m_2 := n + 1$;
  while $m_2 - m_1 > 1$:
    (* *Tu velja: na* $a[\ell..d]$ *je kvečjemu* $k/2$ *elementov, ki so manjši od* m_1*, toda več kot* $k/2$ *elementov, ki so manjši od* m_2*.* *)
    $m := \lfloor (m_1 + m_2)/2 \rfloor$;
    if KOLIKOMANJŠIH$(\ell, d, m) \leq k/2$ **then** $m_1 := m$ **else** $m_2 := m$;
  return m_1;

Manjka nam le še funkcija KOLIKOMANJŠIH, ki mora prešteti, koliko elementov z območja $a[\ell..d]$ je manjših od m. To območje morda pokriva nekatere bloke v celoti, na začetku in na koncu pa je lahko še po en delno pokrit blok. Pri delno pokritih blokih preprosto preglejmo vse elemente s preseka tistega bloka in območja $a[\ell..d]$ in preštejmo, koliko jih je manjših od m; pri blokih, ki so pokriti v celoti, pa si pomagajmo s kopijo bloka, v kateri so elementi urejeni naraščajoče: v njej lahko z bisekcijo preštejemo, koliko elementov je manjših od m. Tako imamo z vsakim v celoti pokritim blokom $O(\log B)$ dela, takih blokov pa je največ B; poleg tega imamo še $O(B)$ dela z vsakim delno pokritim blokom; to je skupaj $O(B \log B)$ za funkcijo KOLIKOMANJŠIH, zato pa $O(B(\log B)(\log n))$ za en izračun mediane oz. $O(qB(\log B)(\log n)) = O(q\sqrt{n}(\log n)^2)$ za vseh q poizvedb. K temu moramo prišteti še čas predpriprave podatkov: da smo pripravili urejeno kopijo vsakega bloka, smo morali porabiti $O(B \log B)$ časa na blok, kar za vse bloke skupaj nanese $O(B^2 \log B) = O(n \log n)$.

Če si lahko privoščimo porabiti $O(n\sqrt{n})$ dodatnega pomnilnika in časa, lahko rešitev še izboljšamo: za vsak blok si pripravimo tabelo, v kateri za vsako možno vrednost od 1 do n piše, koliko elementov bloka je manjših od nje. Ko imamo pripravljeno urejeno kopijo bloka, lahko takšno tabelo pripravimo v $O(n)$ časa, kar je skupaj $O(n\sqrt{n})$ za vse bloke. V funkciji KolikoManjših potem ni treba delati bisekcije po vsakem v celoti pokritem bloku, ampak lahko število elementov, ki so v njem manjši od m, preprosto odčitamo iz tabele. Časovna zahtevnost funkcije KolikoManjših se tako zmanjša za faktor $O(\log B)$; časovna zahtevnost vseh q poizvedb skupaj zdaj znaša $O(q\sqrt{n} \cdot \log n)$.

Doslej se nismo še nič opirali na to, da poznamo vse poizvedbe vnaprej. Ker naloga pravi, da jih poznamo vnaprej, lahko uporabimo prijem, ki je v tekmovalnem programiranju znan kot Mo-jev algoritem. Poizvedbe uredimo glede na blok, ki mu pripada ℓ_i, torej na blok, v katerem se začnejo; kjer pa se več poizvedb začne v istem bloku, jih uredimo po d_i. Zdaj naredimo takole:

for $u := 1$ **to** B:
  $M :=$ prazna podatkovna struktura (več o njej kasneje);
  pregleduj poizvedbe (ℓ_i, d_i), ki se začnejo v bloku u, urejene po d_i:
    če je to prva taka poizvedba:
      dodaj v M elemente $a[\ell_i..d_i]$;
    sicer:
      dodaj v M elemente $a[d+1..d_i]$;
      if $\ell_i < \ell$ **then** dodaj v M elemente $a[\ell_i..\ell-1]$
      else pobriši iz M elemente $a[\ell..\ell_i-1]$;
    $\ell := \ell_i$; $d := d_i$;
    odgovor na trenutno poizvedbo je mediana elementov v M;

V podatkovni strukturi M torej hranimo elemente $a[\ell..d]$; pri vsaki naslednji poizvedbi dodamo in pobrišemo nekaj elementov, dokler struktura ne hrani ravno elementov $a[\ell_i..d_i]$. Takrat s pomočjo M izračunamo njihovo mediano, kar je tudi odgovor na trenutno poizvedbo (ℓ_i, d_i). Pri tem nam vrstni red, v katerem obravnavamo poizvedbe, pomaga, da dodajanj in brisanj ni preveč: pri posameznem u se d_i ves čas povečuje, zato je dodajanj na desnem koncu največ $O(n)$, kar je po vseh u skupaj $O(n\sqrt{n})$ dodajanj; poleg tega se pri vsaki poizvedbi lahko levi konec l_i premakne za največ $O(B)$ mest levo ali desno, ker lahko le tako ostane znotraj bloka u; tako je po vseh poizvedbah skupaj še $O(q\sqrt{n})$ dodajanj in/ali brisanj na levem koncu. Za M lahko uporabimo rdeče-črno drevo, v katerem naj vsako vozlišče hrani tudi število elementov v celotnem poddrevesu, ki se začne pri njem; takrat nam dodajanje, brisanje in izračun mediane vzamejo po $O(\log n)$ časa. Še ena možnost je par kopic: ena hrani elemente, manjše od mediane (večji naj bodo pri vrhu), druga pa elemente, večje od mediane (manjši naj bodo pri vrhu); da bomo lahko pobrisali poljuben element, moramo vzdrževati tudi kazalo, ki za vsak element pove, kje točno (in v kateri kopici) se nahaja; dodajanje in brisanje vzame še vedno $O(\log n)$ časa, izračun mediane pa celo samo $O(1)$ časa. Skupaj je časovna zahtevnost vseh poizvedb $O((n+q)\sqrt{n} \cdot \log n)$. Pri časovni zahtevnosti imamo tukaj člen $O(n\sqrt{n} \cdot \log n)$, ki ga prejšnji dve rešitvi nista imeli; če je n velik v primerjavi s q, je ta rešitev slabša od prejšnjih dveh; če pa je (in to je pri tej nalogi bolj naravna predpostavka) q velik v primerjavi z n, združuje naša nova rešitev prednosti prejšnjih dveh: časovna

zahtevnost bo le $O(q\sqrt{n} \cdot \log n)$ in ne $O(q\sqrt{n}(\log n)^2)$, prostorska pa bo le $O(n)$ in ne $O(n\sqrt{n})$.

Naloge so sestavili: šolarkina uganka — Nino Bašić; eskalacija — Matija Grabnar; mehurčki, barvanje zebre, prevoz po mreži — Tomaž Hočevar; mediana — Vid Kocijan; pobeg iz močvare — Filip Koprivec; palindromska razbitja, stolp — Janez Brank.

NASVETI ZA MENTORJE O IZVEDBI ŠOLSKEGA TEKMOVANJA SŠ IN OCENJEVANJU NA NJEM

[Naslednje nasvete in navodila smo poslali mentorjem, ki so na posameznih šolah skrbeli za izvedbo in ocenjevanje šolskega tekmovanja. Njihov glavni namen je bil zagotoviti, da bi tekmovanje potekalo na vseh šolah na približno enak način in da bi ocenjevanje tudi na šolskem tekmovanju potekalo v približno enakem duhu kot na državnem.—*Op. ur.*]

Tekmovalci naj pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom; ocenjevanje teh odgovorov poteka v vsakem primeru tako, da jih pregleda in oceni mentor (in ne npr. tako, da bi se poskušalo izvorno kodo, ki so jo tekmovalci napisali v svojih odgovorih, prevesti na računalniku in pognati na kakšnih testnih podatkih). Čas reševanja je omejen na 180 minut.

Nekatere naloge kot odgovor zahtevajo program ali podprogram v kakšnem konkretnem programskem jeziku, nekatere naloge pa so tipa „opiši postopek“. Pri slednjih je načeloma vseeno, v kakšni obliki je postopek opisan (naravni jezik, psevdokoda, diagram poteka, izvorna koda v kakšnem programskem jeziku, ipd.), samo da je ta opis dovolj jasen in podroben in je iz njega razvidno, da tekmovalec razume rešitev problema.

Glede tega, katere programske jezike tekmovalci uporabljajo, naše tekmovanje ne postavlja posebnih omejitev, niti pri nalogah, pri katerih je rešitev v nekaterih jezikih znatno krajša in enostavnejša kot v drugih (npr. uporaba perla ali pythona pri problemih na temo obdelave nizov).

Kjer se v tekmovalčevem odgovoru pojavlja izvorna koda, naj bo pri ocenjevanju poudarek predvsem na vsebinski pravilnosti, ne pa na sintaktični. Pri ocenjevanju na državnem tekmovanju zaradi manjkajočih podpičij in podobnih sintaktičnih napak odbijemo mogoče kvečjemu eno točko od dvajsetih; glavno vprašanje pri izvorni kodi je, ali se v njej skriva pravilen postopek za rešitev problema. Ravno tako ni nič hudega, če npr. tekmovalec v rešitvi v C-ju pozabi na začetku **#include**ati kakšnega od standardnih headerjev, ki bi jih sicer njegov program potreboval; ali pa če podprogram main() napiše tako, da vrača **void** namesto **int**.

Pri vsaki nalogi je možno doseči od 0 do 20 točk. Od rešitve pričakujemo predvsem to, da je pravilna (= da predlagani postopek ali podprogram vrača pravilne rezultate), poleg tega pa je zaželeno tudi, da je učinkovita (manj učinkovite rešitve dobijo manj točk).

Če tekmovalec pri neki nalogi ni uspel sestaviti cele rešitve, pač pa je prehodil vsaj del poti do nje in so v njegovem odgovoru razvidne vsaj nekatere od idej, ki jih rešitev tiste naloge potrebuje, naj vendarle dobi delež točk, ki je približno v skladu s tem, kolikšen delež rešitve je našel.

Če v besedilu naloge ni drugače navedeno, lahko tekmovalčeva rešitev vedno predpostavi, da so vhodni podatki, s katerimi dela, podani v takšni obliki in v okviru takšnih omejitev, kot jih zagotavlja naloga. Tekmovalcem torej načeloma ni treba pisati rešitev, ki bi bile odporne na razne napake v vhodnih podatkih.

Če oblika vhodnih podatkov ni natančno določena, si lahko podrobnosti tekmovalec izbere sam. Na primer, če naloga pravi, da dobimo seznam parov, je to lahko v praksi tabela (*array*), vektor, *linked list* ali še kaj drugega, pari pa so lahko

bodisi strukture, ki jih je deklarirala tekmovalčeva rešitev, ali pa kaj iz standardne knjižnice (kot je pair v C++ ali tuple v pythonu).

V nadaljevanju podajamo še nekaj nasvetov za ocenjevanje pri posameznih nalogah.

1. Cikcakasti nizi

- Naloga poudarja, da so nizi lahko dolgi in da naj bo rešitev učinkovita. Rešitve s časovno zahtevnostjo, slabšo od linearne, naj dobijo največ 10 točk, če so sicer pravilne.
- V eni od naših rešitev smo brali niz s standardnega vhoda znak po znak in ga sproti obdelovali. Za enako dobre naj štejejo tudi rešitve, ki preberejo celoten niz v glavni pomnilnik, nato pa ga obdelajo.
- Rešitvam, ki porabijo linearno mnogo dodatnega pomnilnika (za povrhu vhodnega niza) — npr. zato, ker si eksplicitno pripravijo celoten seznam kosov, na katere se pri tej nalogi razdeli niz, ali pa njihovih dolžin — naj se zaradi tega odšteje dve točki.
- Za manjše, nebistvene napake pri preverjanju cikcakavosti (npr. če program namesto pogoja $d_1 < d_2 > d_3$ preverja $d_1 \leq d_2 \geq d_3$) naj se odšteje največ dve točki.

2. Histogram

- Za vse točke pričakujemo rešitev s časovno zahtevnostjo $O(n)$. Rešitve s časovno zahtevnostjo $O(n^2)$ naj dobijo največ 15 točk, če so sicer pravilne. Rešitve s časovno zahtevnostjo, slabšo od $O(n^2)$, naj dobijo največ 10 točk, če so sicer pravilne. Med slednje štejejo tudi morebitne rešitve, kjer bi bila časovna zahtevnost kaj odvisna od višine stolpcev (kajti glede teh višin ne daje naloga nobenih zagotovil, razen tega, da so višine pač različna naravna števila).
- V našem opisu rešitve imamo dve različici z linearno časovno zahtevnostjo: eno, ki porabi $O(n)$ dodatnega pomnilnika (poleg vhodnega histograma), in eno ki porabi le $O(1)$ dodatnega pomnilnika. Oboje naj se šteje za enako dobro in lahko dobi vse točke.

3. Naredimo hitro testiranje zares hitro!

- Ključno pri tej nalogi je opažanje, da je smiselno paciente urediti po času, do katerega hočejo biti testirani (v naši rešitvi je ta čas označen z r_i), in jih testirati v tem vrstnem redu. Zaželeno je, da poskuša tekmovalčeva rešitev podati vsaj nekakšno utemeljitev, zakaj je to res, vendar prav veliko v tej smeri ne smemo pričakovati.
- Rešitve, ki porabijo $O(n \log n)$ časa namesto $O(n)$ časa, naj veljajo za enako dobre in lahko tudi dobijo vse točke (če so pravilne). Sem sodijo npr. rešitve, ki ljudi ne urejajo s štetjem, ali pa rešitve, ki določajo najmanjši m z bisekcijo.

- Vse točke lahko dobijo tudi rešitve, ki bi morda predpostavile, da ležijo vsi časi znotraj enega dne in je zato nabor možnih vrednosti zelo omejen, kar lahko olajša urejanje. Od rešitev se tudi ne pričakuje, da se ukvarjajo s podrobnostmi postopka za urejanje.

- Rešitve, ki bi porabile $O(n^2)$ časa (npr. ker gredo morda za vsako možno število točk m po vseh ljudeh, da preverijo, ali se dá sestaviti veljaven razpored za m testirnih točk), naj dobijo največ 15 točk, če so drugače pravilne.

- Pri tej nalogi je načeloma mogoče, da problem sploh ni rešljiv (če hoče biti kdo gotov s testiranjem že v manj kot t sekundah po začetku testiranja ob 7:00), vendar se rešitvam tekmovalcev s tem robnim primerom ni treba ukvarjati (oz. se ga sploh zavedati), saj besedilo naloge zagotavlja, da bodo vhodni podatki taki, da bo problem zagotovo rešljiv.

4. Palindromi

- Pri tej nalogi pričakujemo za vse točke rešitev s časovno zahtevnostjo $O(n^2)$. Rešitve z zahtevnostjo $O(n^3)$ naj dobijo največ 12 točk, če so sicer pravilne.

- Rešitvi, ki bi pomotoma štela tudi palindromne podnize dolžine 1 (in tako povečala število palindromov za n), naj se zaradi tega odšteje dve točki.

- Če bi kakšna rešitev pomotoma štela le palindrome lihe dolžine ali le palindrome sode dolžine, naj se ji zaradi tega odšteje pet točk.

5. Čarobne jame

- Rešitev, ki bi (npr. zaradi kakšne nespametne rekurzije) lahko porabila eksponentno mnogo časa (ali pa celo padla v neskončno zanko zaradi kakšnih ciklov v grafu ipd.), naj dobi največ 10 točk.

- Dodajanje novih povezav zaradi čarobnih zvitkov bi se dalo upoštevati tudi na kak nerodnejši način od tistega v naši rešitvi; lahko bi na primer po vsakem dodajanju povezave začeli preiskovati graf spet od začetka; ali pa bi ga preiskovali v zanki, dokler ne bi enkrat pregledali celega grafa, ne da bi pri tem dodali kakšno novo povezavo. Zaradi takih neučinkovitosti, ki povečajo časovno zahtevnost za faktor $O(n)$, naj se rešitvi odšteje 5 točk. Enako velja tudi za podobne neučinkovitosti pri predstavitvi grafa (npr. če si rešitev ne pripravi seznamov sosedov, pač pa vsakič, ko vzame naslednjo točko iz vrste, pregleda *vse* povezave v grafu; ali pa če uporablja matriko sosednosti namesto seznamov sosedov).

- Možna napaka pri tej nalogi je, da ob dodajanju nove povezave ne obravnavamo posebej primera, ko smo eno od krajišč te povezave že pregledali, drugega pa še ne; če takrat le dodamo novo povezavo, nam ne bo pomagala priti iz že pregledanega krajišča v drugo krajišče. Rešitvam, ki spregledajo ta primer, naj se zaradi tega odšteje dve točki.

- Če se kakšna rešitev za zvitke sploh ne zmeni in pregleda le prvotni sistem jam, naj dobi največ 12 točk (če vsaj to naredi pravilno).
- V naši rešitvi je tudi primer implementacije v C++, vendar še enkrat poudarimo, da naloga zahteva le opis postopka in ne nujno implementacije v kakšnem konkretnem programskem jeziku.
- Naloga pravi, naj tekmovalec tudi oceni časovno zahtevnost svoje rešitve. Tistim rešitvam, ki tega ne naredijo ali pa so v svoji oceni povsem zgrešene, naj se zaradi tega odšteje dve točki.

Težavnost nalog

Državno tekmovanje ACM v znanju računalništva poteka v treh težavnostnih skupinah (prva je najlažja, tretja pa najtežja); na tem šolskem tekmovanju pa je skupina ena sama, vendar naloge v njej pokrivajo razmeroma širok razpon zahtevnosti. Za občutek povejmo, s katero skupino državnega tekmovanja so po svoji težavnosti primerljive posamezne naloge letošnjega šolskega tekmovanja:

Naloga	Kam bi sodila po težavnosti na državnem tekmovanju ACM
1. Cikcakasti nizi	lažja do srednja naloga v prvi skupini
2. Histogram	srednje težka naloga v prvi ali lažja v drugi skupini
3. Hitro testiranje	srednje težka naloga v drugi skupini
4. Palindromi	težka naloga v prvi ali srednja v drugi skupini[42]
5. Čarobne jame	težja naloga v drugi ali lahka v tretji skupini

Če torej na primer neki tekmovalec reši le eno ali dve lažji nalogi, pri ostalih pa ne naredi (skoraj) ničesar, to še ne pomeni, da ni primeren za udeležbo na državnem tekmovanju; pač pa je najbrž pametno, če na državnem tekmovanju ne gre v drugo ali tretjo skupino, pač pa v prvo.

Podobno kot prejšnja leta si tudi letos želimo, da bi čim več tekmovalcev s šolskega tekmovanja prišlo tudi na državno tekmovanje in da bi bilo šolsko tekmovanje predvsem v pomoč tekmovalcem in mentorjem pri razmišljanju o tem, v kateri težavnostni skupini državnega tekmovanja naj kdo tekmuje.

Zadnja leta na državnem tekmovanju opažamo, da je v prvi skupini izrazito veliko tekmovalcev v primerjavi z drugo in tretjo, med njimi pa je tudi veliko takih z zelo dobrimi rezultati, ki bi prav lahko tekmovali tudi v kakšni težji skupini. Mentorjem zato priporočamo, naj tekmovalce, če se jim zdi to primerno, spodbudijo k udeležbi v zahtevnejših skupinah.

[42] Težavnost te naloge je odvisna od tega, kako točkujemo različno učinkovite rešitve. Če bi se dalo dobiti vse točke že za rešitev v času $O(n^3)$, bi postala to morda srednje težka naloga za prvo skupino; če pa bi bilo treba za vse točke najti rešitev v času $O(n)$, bi bila to težka naloga za tretjo skupino.

REZULTATI

Tabele na naslednjih straneh prikazujejo vrstni red vseh tekmovalcev, ki so sodelovali na letošnjem tekmovanju. Poleg skupnega števila doseženih točk je za vsakega tekmovalca navedeno tudi število točk, ki jih je dosegel pri posamezni nalogi. V prvi in drugi skupini je mogoče pri vsaki nalogi doseči največ 20 točk, v tretji skupini pa največ 100 točk.

Načeloma se v vsaki skupini podeli dve prvi, dve drugi in dve tretji nagradi, letos pa so se rezultati izšli tako, da smo v prvi skupini izjemoma podelili štiri tretje nagrade. Poleg nagrad na državnem tekmovanju v skladu s pravilnikom podeljujemo tudi zlata in srebrna priznanja. Število zlatih priznanj je omejeno na eno priznanje na vsakih 25 udeležencev šolskega tekmovanja (teh je bilo letos 215) in smo jih letos podelili sedem. Srebrna priznanja pa se podeljujejo po podobnih kriterijih kot pred leti pohvale; prejmejo jih tekmovalci, ki ustrezajo naslednjim trem pogojem: (1) tekmovalec ni dobil zlatega priznanja; (2) je boljši od vsaj polovice tekmovalcev v svoji skupini; in (3) je tekmoval v prvi ali drugi skupini in dobil vsaj 20 točk ali pa je tekmoval v tretji skupini in dobil vsaj 80 točk. Namen srebrnih priznanj je, da izkažemo priznanje in spodbudo vsem, ki se po rezultatu prebijejo v zgornjo polovico svoje skupine. Podobno prakso poznajo tudi na nekaterih mednarodnih tekmovanjih; na primer, na mednarodni računalniški olimpijadi (IOI) prejme medalje kar polovica vseh udeležencev. Poleg zlatih in srebrnih priznanj obstajajo tudi bronasta, ta pa so dobili najboljši tekmovalci v okviru šolskih tekmovanj (letos smo podelili 86 bronastih priznanj).

V tabelah na naslednjih straneh so prejemniki nagrad označeni z „1“, „2“ in „3“ v prvem stolpcu, prejemniki priznanj pa z „Z“ (zlato) in „S“ (srebrno).

PRVA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Tristan Zore	3	ŠC N. mesto, SEŠTG	20	20	20	17	20	97
1Z	2	Adrian Sebastian Šiška	4	Vegova Ljubljana	20	20	20	16	20	96
2S	3	Gal Končar	2	Vegova Ljubljana	19	20	20	19	16	94
2S		Mitja Ševerkar	1	Gimnazija Bežigrad	18	20	17	19	20	94
3S	5	Matic Babnik	4	Vegova Ljubljana	20	20	20	17	16	93
3S	6	Vid Ošep	3	Gimnazija Vič	18	20	15	18	20	91
3S		Domen Anderlič	3	Vegova Ljubljana	20	19	20	17	15	91
3S		Nejc Sušnik	3	Gimnazija Škofja Loka	19	20	19	18	15	91
S	9	Matic Vernik	4	SERŠ Maribor	20	19	20	16	14	89
S	10	Amon Šincek	2	Vegova Ljubljana	20	20	20	15	13	88
S		Gregor Virant	4	ŠC Celje, Gimn. Lava	15	18	20	20	15	88
S	12	Matic Izak	1	Gimnazija Vič	15	20	20	15	15	85
S	13	Amadej Arh	2	Vegova Ljubljana	15	18	20	13	17	83
S	14	Tom Sabadin	2	Vegova Ljubljana	20	18	20	4	19	81
S		Kristjan Kelvišar	4	ŠC Kranj, Str. gimn.	15	20	20	12	14	81
S		Gašper Nemgar	3	Vegova Ljubljana	20	16	20	9	16	81
S	17	Tilen Zupet	4	SŠ Domžale, PSŠ	20	15	20	9	15	79
S	18	Bor Čarman Djuran	3	ŠC Kranj, STŠ Kranj	20	16	20	7	15	78
S		Tilen Goršek	4	ŠC Celje, SŠ za KER	20	18	19	7	14	78
S	20	Dominik Krašovec	4	ŠC N. mesto, SEŠTG	10	20	15	19	12	76
S	21	Žan Hribar	4	STPŠ Trbovlje	12	15	20	12	15	74
S		Luka Herman	2	I. gimnazija v Celju	20	20	18	3	13	74
S		Aljaž Kos	4	I. gimnazija v Celju	7	20	18	15	14	74
S		Samo Čibej	2	STPŠ Trbovlje	10	20	20	16	8	74
S	25	Nik Deželak	3	ŠC Celje, SŠ za KER	17	15	20	18	3	73
S		Žan Čuden	3	Vegova Ljubljana	15	20	20	5	13	73
S	27	Tine Šuc	3	ŠC Nova Gorica	15	7	15	19	15	71
S		Matej Markuža	4	STŠ Koper	0	18	18	19	16	71
S		Diego Bonaca	4	STŠ Koper	12	19	20	6	14	71
S	30	Peter Mašič	4	Gimnazija Šentvid	17	14	20	6	13	70
S		Matija Bregar	4	STPŠ Trbovlje	15	16	20	4	15	70
S	32	Valentin Ozimic	2	ŠC Velenje, ERŠ	10	17	18	10	14	69
S		Nejc Zalokar	1	ZRI	8	18	18	9	16	69
S	34	Bor Furlan	4	ŠC Kranj, STŠ Kranj	10	16	20	6	16	68
S	35	Tilen Stermecki	3	Vegova Ljubljana	15	12	20	5	15	67
S	36	Maj Donko	3	SERŠ Maribor	20	14	20	9	3	66
S	37	Leo Klarić	3	I. gimnazija v Celju	15	0	20	16	14	65
S		Patrik Turk	3	STŠ Koper	18	20	20	4	3	65
S	39	Urban Krepel	3	ŠC Velenje, ERŠ	10	16	20	5	12	63
S	40	Jaša Gregorič	3	ŠC Nova Gorica	19	16	20	2	5	62
	41	Luka Kuder	4	ŠC Celje, Gimn. Lava	12	17	20	4	8	61
		Darko Sever	4	SERŠ Maribor	15	10	17	5	14	61

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
43	Erik Fajfar	1	Gimnazija Vič	10	16	20	5	9	60
	Teja Nemec	4	SŠ Domžale, Splošna gimnazija	15	16	15	6	8	60
45	Bor Kaučič	1	II. gimnazija Maribor	10	18	10	15	5	58
46	Žiga Novak	3	ŠC Kranj, Strokovna gimnazija	14	20	5	3	15	57
	Aney Vaishnav	3	ŠC Celje, SŠ za KER	10	16	10	9	12	57
	Enej Breskvar	2	ZRI	4	14	20	4	15	57
	Klemen Mežnar	2	Gimnazija Ilirska Bistrica	8	12	20	5	12	57
50	Tilen Gašparič	4	STPŠ Trbovlje	0	16	20	10	10	56
51	Luka Šturbej	3	ŠC Celje, SŠ za KER	15	11	13	3	13	55
	Timotej Lipič	4	SPTŠ Murska Sobota	13	13	10	3	16	55
53	Fedja Razpet	1	Gimnazija Vič	10	14	19	11	0	54
54	Anže Pintar	4	ŠC Novo mesto, SEŠTG	5	19	20	2	5	51
55	Tevž Selčan	3	ŠC Celje, SŠ za KER	15	5	20	5	5	50
56	Enej Fonda	4	ZRI	0	20	20	9	0	49
57	Luka Mavc	2	STŠ Koper	12	16	20	0	0	48
58	Tine Lisec	4	Gimnazija Franceta Prešerna	7	15	15	10	0	47
	Ajda Heric	2	II. gimn. Maribor + ZRI	1	20	0	9	17	47
60	Marko Stanić	4	SPTŠ Murska Sobota	10	16	5	4	6	41
61	Maks Tomšič	1	ZRI	8	7	20	5	0	40
62	Janez Malovrh	4	Gimnazija Poljane	10	14	15	0	0	39
	Jernej Dežman	3	ŠC Kranj, Strokovna gimnazija	12	16	1	10	0	39
64	Tim Rosulnik	3	ŠC Kranj, Strokovna gimnazija	0	17	20	0	0	37
65	Alex Lackovič	2	SPTŠ Murska Sobota	1	18	10	5	0	34
66	Anže Stanonik	3	ŠC Kranj, Strokovna gimnazija	5	14	5	3	5	32
67	Luka Kotnik	4	SŠ Domžale, PSŠ	15	8	8	0	0	31
68	Jon Cvetko	2	STPŠ Trbovlje	10	17	0	0	0	27
69	Sreten Perić	3	SERŠ Maribor	15	5	0	6	0	26
70	Julian Aljaž Loy	1	Škof. klas. gimn. Lj.	1	5	0	3	13	22
	Aleks Detiček	2	ŠC Ptuj, ERŠ	10	5	0	0	7	22
72	Žan Čahuk	2	SPTŠ Murska Sobota	0	8	0	0	13	21
73	Simon Kušar	1	II. gimnazija Maribor	0	20	0	0	0	20
	Nejc Germovšek	1	ZRI	10	5	0	0	5	20
75	Mark Kotnik	2	ŠC Velenje, ERŠ	7	5	5	2	0	19
76	Nik Kocijančič	2	Gimnazija Murska Sobota	0	18	0	0	0	18
77	Lan Lebar	3	ŠC Kranj	11	0	1	0	5	17
78	Silvo Topolovec	2	ŠC Ptuj, ERŠ	1	0	5	2	3	11
79	Mark Šehič	1	Gimnazija Velenje	5	3	0	0	0	8
80	Rene Kolednik	1	ŠC Ptuj, ERŠ	1	1	5	0	0	7
81	Tom Kliner	2	ŠC Velenje, ERŠ	0	5	0	0	0	5
82	Robert Skok	2	ŠC Ptuj, ERŠ	1	1	0	0	0	2

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Rok Perko	3	Vegova Ljubljana	19	20	0	20	10	69
1Z	2	Anton Luka Šijanec	4	Gimnazija Bežigrad	9	10	20	20	6	65
2Z	3	Jan Šuštar	3	Vegova Ljubljana	7	13	3	19	16	58
2S	4	Jure Maček	3	Vegova Ljubljana	15	19	5	8	4	51
3S	5	Nejc Mihelčič	4	ZRI	19	0	15	0	6	40
3S	6	Barbara Makovec	4	Gimnazija Vič	17	7	8	0	5	37
S	7	Tim Strnad	3	ZRI	8	15	0	12	0	35
S		Martin Murko	3	ZRI	15	10	4	0	6	35
S		Tilen Juričan	4	ŠC Kranj, STŠ Kranj	18	0	5	12	0	35
	10	Igor Setnikar	2	Gimnazija Vič	14	0	0	12	6	32
		Maj Zabukovnik	3	ŠC Celje, SŠ za KER	4	7	4	2	15	32
	12	Žan Škorja	3	ŠC Celje, SŠ za KER	11	1	10	1	7	30
	13	Matic Dremelj	4	ZRI	14	5	2	2	6	29
	14	Aljaž Kos	3	ZRI	14	12	0	0	0	26
		Andraž Roth	2	ZRI	0	0	15	5	6	26
	16	Peter Jereb	2	ZRI	9	8	5	0	3	25
	17	Andrej Repič	4	ŠC Nova Gorica	10	0	12	1	0	23
		Domen Korenini	3	Gimnazija Vič	4	5	7	0	7	23
	19	Nik Vodovnik	4	ZRI	8	1	4	0	3	16
	20	Aljaž Marn	4	ZRI	5	0	0	2	7	14
	21	Bojan Brdar Turk	1	Gimnazija Poljane	0	0	12	0	0	12

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Jakob Žorž	3	ZRI	100	88	85	94	51	418
1Z	2	Jošt Smrtnik	4	Gimnazija Vič	87	100	50	88	40	365
2S	3	Bor Brudar	4	ŠC Novo mesto, SEŠTG	84	70	38	88		280
2S	4	Brest Lenarčič	3	ŠC Rogaška Slatina	100	72	37			209
2S	5	Luka Heric	1	II. gim. Maribor + ZRI	90	85	6	18		199
2S	6	Luka Urbanc	3	ZRI	97		40			137
S	7	Klemen Pregelj	4	ŠC Nova Gorica	90	0	9	0		99
S	8	Filip Štamcar	4	ZRI	20	7	60	0		87
	9	Lara Stamać	3	ZRI	70		9	0		79
	10	Špela Gačnik	3	G. Bežigrad, Medn. šola	70		6	0		76
	11	Leon Fišer	4	Vegova Ljubljana	50		25	0		75
	12	Jakob Fajt	4	II. gimnazija Maribor	11	0	20	30	0	61
	13	Žan Pustoslemšek	3	ŠC Ravne na Kor., Gim.	20		27	0		47
	14	Tim Thuma	4	Vegova Ljubljana	17	0	15	0		32
	15	Oskar Rotar	3	ZRI	20					20
	16	Rok Gerič	4	Gimn. Murska Sobota	15	4				19
	17	Bor Kajin	4	ŠC Nova Gorica		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	Vegova Ljubljana	794,2
2	Gimnazija Vič	693
3	ŠC Novo mesto, SEŠTG	392
4	ŠC Celje, SŠ za KER	387
5	II. gimnazija Maribor	281
6	STPŠ Trbovlje	274
7	STŠ Koper	255
8	SERŠ Maribor	242
9	ŠC Nova Gorica	238,4
10	Gimnazija Bežigrad	224
11	ŠC Kranj, STŠ Kranj	216
12	ŠC Kranj, Strokovna gimnazija	214
13	I. gimnazija v Celju	213
14	ŠC Velenje, ERŠ	156
15	SPTŠ Murska Sobota	151
16	ŠC Celje, Gimnazija Lava	149
17	ŠC Rogaška Slatina	125,4
18	SŠ Domžale, Poklicna in strokovna šola	110
19	Gimnazija Škofja Loka	91
20	Gimnazija Šentvid	70
21	Gimnazija Poljane	63
22	SŠ Domžale, Splošna gimnazija	60
23	Gimnazija Ilirska Bistrica	57
24	Gimnazija Franceta Prešerna	47
25	Gimnazija Bežigrad, Mednarodna šola	45,6
26	ŠC Ptuj, ERŠ	42
27	Gimnazija Murska Sobota	29,4
28	ŠC Ravne na Koroškem, Gimnazija	28,2
29	Škofijska klasična gimnazija Ljubljana	22
30	ŠC Kranj	17
31	Gimnazija Velenje	8

NAGRADE

Za nagrado so najboljši tekmovalci vsake skupine prejeli naslednjo strojno opremo in knjižne nagrade:

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Tristan Zore	telefon Samsung Galaxy S21 FE 5G
1	1	Adrian Sebastian Šiška	telefon Samsung Galaxy S21 FE 5G
1	2	Gal Končar	telefon Samsung Galaxy A53 5G
1	2	Mitja Ševerkar	telefon Samsung Galaxy A53 5G
1	3	Matic Babnik	telefon Samsung Galaxy A53 5G
1	3	Vid Ošep	miška Razer DeathAdder V2
1	3	Domen Anderlič	miška Razer DeathAdder V2
1	3	Nejc Sušnik	miška Razer DeathAdder V2
2	1	Rok Perko	telefon Samsung Galaxy S21 FE 5G S. in F. Halim: *Competitive Programming 4*
2	1	Anton Luka Šijanec	telefon Samsung Galaxy S21 FE 5G S. in F. Halim: *Competitive Programming 4*
2	2	Jan Šuštar	telefon Samsung Galaxy A53 5G S. in F. Halim: *Competitive Programming 4*
2	2	Jure Maček	telefon Samsung Galaxy A53 5G
2	3	Nejc Mihelčič	telefon Samsung Galaxy A53 5G
2	3	Barbara Makovec	miška Razer DeathAdder V2
3	1	Jakob Žorž	telefon Samsung Galaxy S21 FE 5G Raspberry Pi 4 model B Cormen *et al.: Introduction to Algorithms*
3	2	Jošt Smrtnik	telefon Samsung Galaxy S21 FE 5G Raspberry Pi 4 model B Cormen *et al.: Introduction to Algorithms*
3	2	Bor Brudar	telefon Samsung Galaxy S21 FE 5G Cormen *et al.: Introduction to Algorithms*
3	2	Brest Lenarčič	telefon Samsung Galaxy A53 5G
3	3	Luka Heric	telefon Samsung Galaxy A53 5G
3	3	Luka Urbanc	telefon Samsung Galaxy A53 5G
Off-line naloga — Sokoban			
	1	Luka Stražišar	Raspberry Pi 4 model B
	2	Mai Rupnik	Raspberry Pi 4 model B

SODELUJOČE ŠOLE IN MENTORJI

II. gimnazija Maribor	Mirko Pešec
Gimnazija Bežigrad, Gimnazija	Andrej Šuštaršič
Gimnazija Bežigrad, Mednarodna šola	Gregor Anželj
Gimnazija Franceta Prešerna	David Konc
Gimnazija Murska Sobota	Romana Zver
Gimnazija Poljane	Janez Malovrh, Boštjan Žnidaršič
Gimnazija Šentvid	Nastja Lasič
Gimnazija Škofja Loka	Alenka Kolenc Krajnik
Gimnazija Velenje	Ivan Jovan
Gimnazija Vič	Klemen Bajec, Marina Trost
I. gimnazija v Celju	Sebastjan Tkavc
Srednja elektro-računalniška šola Maribor (SERŠ)	Dušan Fugina, Slavko Nekrep, Vida Motaln, Branko Potisk, Manja Sovič Potisk
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Dominik Letnar, Igor Kutoš
Srednja šola Domžale, Poklicna in strokovna šola	Bine Iljaš
Srednja šola Domžale, Splošna gimnazija	Bine Iljaš
Srednja tehniška in poklicna šola Trbovlje (STPŠ)	Uroš Ocepek
Srednja tehniška šola Koper (STŠ)	Senka Felicijan, Andrej Florjančič
Šolski center Celje, Gimnazija Lava	Karmen Kotnik
Šolski center Celje, Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Jaka Koren, Žiga Pušelc
Šolski center Kranj, Srednja tehniška šola (STŠ)	Miha Baloh
Šolski center Kranj, Strokovna gimnazija	Lina Dečman Molan, Jan Sušnik
Šolski center Nova Gorica	Barbara Pušnar, Tomaž Mavri, Boštjan Vouk

Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija (SEŠTG)	Simon Vovko, Albert Zorko
Šolski center Postojna, Gimnazija Ilirska Bistrica	Franci Nahtigal
Šolski center Ptuj, Elektro in računalniška šola (ERŠ)	Franc Vrbančič
Šolski center Ravne na Koroškem, Gimnazija	David Ristič
Šolski center Rogaška Slatina	Jože Vajdič
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Miran Zevnik
Škofijska gimnazija Antona Martina Slomška Maribor	Mirko Đukić
Škofijska klasična gimnazija Šentvid	Helena Starc Grlj, Mihael Trajbarič
Vegova Ljubljana	Marko Kastelic, Nataša Makarovič, Aleš Volčini

Zavod za računalniško izobraževanje (ZRI), Ljubljana

REZULTATI CERC 2023

Ker smo letos organizirali srednjeevropsko študentsko tekmovanje v računalništvu (CERC 2023) pri nas v Ljubljani, objavljamo v našem biltenu še rezultate tega tekmovanja. Naloge so na str. 35–52, rešitve pa na str. 123–192.

	Ekipa	Št. rešenih nalog	Čas
1	Kacper Paciorek, Antoni Długosz, Kacper Topolski (Jag. u.)	10	27:41:37
2	Arkadiusz Czarkowski, Bartłomiej Czarkowski, Tomasz Nowak (U. v Varšavi)	9	15:34:22
3	Łukasz Pluta, Krzysztof Boryczka, Antoni Buraczewski (U. v Wrocławu)	9	20:44:21
4	Dorijan Lendvaj, Krešimir Nežmah, Patrick Pavić (U. v Zagrebu)	8	15:21:21
5	Kamil Zwierzchowski, Jan Kwiatkowski, Michał Staniewski (U. v Varšavi)	8	16:57:01
6	Jan Klimczak, Justyna Jaworska, Rafał Pyzik (Jag. u.)	8	20:26:07
7	Adam Ciężkowski, Artur Krzyżyński, Jan Wańkowicz (U. v Wrocławu)	8	23:29:27
8	Péter Gyimesi, Áron Noszály, Péter Varga (ELTE)	7	12:42:39
9	Matīss Kristiņš, Krišjānis Petručeņa, Valters Kalniņš (Latvijska u.)	7	17:23:54
10	Vlatko Borevković, Toni Brajko, Marko Dorčić (U. v Zagrebu)	7	20:46:09
11	Jakub Dziura, Dominik Wawszczak, Piotr Blinowski (U. v Varšavi)	6	12:59:20
12	Tomáš Macháček, Daniel Ilkovič, Michal Staník (Masarykova u.)	6	14:32:23
13	Kamil Szymczak, Jakub Pniewski, Franciszek Witt (U. v Varšavi)	6	17:17:15
14	Jiri Kalvoda, Ondřej Sladký, Tymofii Reizin (Karlova u.)	6	17:42:08
15	Benedek Nádor, Lőrinc Máté, Dávid Gergő Melján (ELTE)	5	8:42:17
16	Jacek Markiewicz, Łukasz Orski, Filip Konieczny (Jag. u.)	5	10:19:16
17	Yasmine Briefs, Jaroslav Urban, Illia Kryvoviaz (CTU)	5	12:12:42
18	Sandra Silina, Kristaps Stals, Kristofers Barkāns (Latvijska u.)	5	13:29:53
19	Hubert Zięba, Tomasz Mazur, Katzper Michno (Jag. u.)	4	4:38:00
20	Ivan Janjić, Ivan Jambrešić, Bartol Markovinović (U. v Zagrebu)	4	5:09:14
21	Michał Januszkiewicz, Wojciech Sobiński, Wojciech Raczuk (U. v Varšavi)	4	5:27:45
22	Igor Hańczaruk, Krzysztof Olejnik, Damian Sosulski (U. v Wrocławu)	4	6:12:46
23	Michal Tkáčik, Ján Priner, Viktor Balan (U. Komenskega)	4	6:51:06
24	Michał Kuśmirek, Paweł Zalewski, Andrzej Jabłoński (U. v Varšavi)	4	8:12:36
25	Joanna Suwaj, Cyryl Szatan, Hubert Dyczkowski (U. v Wrocławu)	4	8:16:51
26	Jozef Číž, Michal Farnbauer, Jan Gottweis (U. Komenskega)	4	8:19:52
27	Matevž Miščič, Jon Mikoš, Domen Hočevar (U. v Ljubljani)	4	8:48:49
28	Miklós Csizmadia, Balázs Makrai-Kis, Csaba Dékány (ELTE)	4	8:50:20
29	Alicja Kluczek, Jerzy Czarkowski, Mateusz Sobkowiak (U. N. Kopernika)	4	9:05:14
30	Łukasz Skabowski, Paweł Czarkowski, Paweł Aniszewski (U. N. Kopernika)	4	9:49:56
31	Job Petrovčič, Luka Horjak, Jakob Schrader (U. v Ljubljani)	4	10:26:29
32	Štěpán Mikéska, Adam Červenka, Petr Slonek (Masarykova u.)	4	10:27:45
33	Daniel Skýpala, Pavlo Tsitsei, Benjamin Swart (Karlova u.)	4	11:08:00
34	Martin Belluš, Martin Štěpánek, Jakub Konc (Karlova u.)	4	12:02:52
35	Bor Grošelj Simić, Patrik Žnidaršič, Ella Potisek (U. v Ljubljani)	4	12:03:37
36	Alen Granda, Mitko Nikov (U. v Mariboru)	4	13:54:29

(*nadaljevanje na naslednji strani*)

REZULTATI CERC 2023 (*nadaljevanje*)

	Ekipa	Št. rešenih nalog	Čas
37	Martin Mlejnecký, Jakub Pelc, Roman Šíp (CTU)	3	3:39:33
38	Bartosz Chomiński, Marcel Szelwiga, Olaf Surgut (U. v Wrocławu)	3	3:47:21
39	Uladzimir Marozau, Ihar Maroz, Aliaksei Mamonau (Jag. u.)	3	4:03:30
40	Martin Nastoupil, Jakub Horák, Hoang Trung Bui (Masarykova u.)	3	5:24:14
41	Andrej Ohrablo, František Sciranka, Tomáš Sládek (CTU)	3	5:59:18
42	Zoltán Szatmáry, Levente Gegő, Soma Szatmári (BUTE)	3	6:41:28
43	Tomáš Prager, Kryštof Rohan, Pavel Holý (CTU)	3	8:14:14
44	Matěj Kříž, Jiří Kvapil, Štěpán Pechman (CTU)	3	9:57:45
45	Matej Uhrin, Branislav Pastula, Norbert Micheľ (UPJŠ)	2	2:08:34
46	Gergő Török, Oliver Regaisz, Péter Dávid Olajos (U. v Szegedu)	2	2:44:51
47	Mário Husár, Tomáš Lokša, Miloš Murín (U. v Žilini)	2	3:36:45
48	Radosław Myśliwiec, Aleksander Trzciński, Michał Dobranowski (AGH)	2	3:37:35
49	Juraj Beňo, Richard Závodský, Erik Zemčík (U. v Žilini)	2	4:32:49
50	Gergely Péter, Zalán Varga, Ákos Kulcsár (U. v Szegedu)	2	5:01:17
51	Alan Bubalo, Rafael Krstačić, Anai Pučić (U. v Pulju)	2	5:19:23
52	Thomas Višvader, Adrián Kabáč, Norbert Vígh (STU)	2	5:33:24
53	Bogusław Banaś, Michał Rusinek, Jakub Gadomski (UMCS)	2	6:00:26
54	Karol Do, Nazar Shcherbyna, Szymon Posiadała (WUT)	2	6:13:53
55	Jakub Patzián, Petr Kladov, Tomáš Bezděk (VŠB)	2	6:42:49
56	Luka Ivanić, Nikolina Rodin, Jakov Tomasić (U. na Reki)	1	0:30:01
57	Benedek Brandschott, Azam Rakhmatillaev, Klevis Imeri (BUTE)	1	1:04:23
58	Miloš Jevtović, Vasilije Božarić, Jovan Pavlović (U. na Primorskem)	1	1:21:59
59	Daniel Dobeš, Ondřej Sivek, Vojtěch Volný (VŠB)	1	2:30:50
60	Anton Horobets, Dzmitry Petukhou, Bohdan Koval (STU)	0	0:00:00

Sodelovale so ekipe z naslednjih univerz:

Češka tehniška univerza (CTU) (Praga, Češka)
Jagielonska univerza (Krakow, Poljska)
Karlova univerza (Praga, Češka)
Latvijska univerza (Riga, Latvija)
Masarykova univerza (Brno, Češka)
Slovaška tehniška univerza (STU) (Bratislava, Slovaška)
Tehniška univerza v Ostravi (VŠB) (Češka)
Varšavska politehnika (WUT) (Poljska)
Univerza Juraja Dobrile v Pulju (Hrvaška)
Univerza Marie Curie-Skłodowske (UMCS) (Lublin, Poljska)
Univerza Komenskega v Bratislavi (Slovaška)
Univerza Loránda Eötvösa (ELTE) (Budimpešta, Madžarska)
Univerza na Primorskem (Slovenija)
Univerza na Reki (Hrvaška)
Univerza Nikolaja Kopernika (Torunj, Poljska)
Univerza Pavola Jozefa Šafárika v Košicah (UPJŠ) (Slovaška)
Univerza v Ljubljani (Slovenija)
Univerza v Mariboru (Slovenija)
Univerza v Szegedu (Madžarska)
Univerza v Varšavi (Poljska)
Univerza v Wrocławu (Poljska)
Univerza v Zagrebu (Hrvaška)
Univerza v Žilini (Slovaška)
Univerza za tehnologijo in ekonomiko (BUTE) (Budimpešta, Madžarska)
Znanstveno-tehnična univerza AGH (Krakow, Poljska)

OFF-LINE NALOGA — SOKOBAN

Na računalniških tekmovanjih, kot je naše, je čas reševanja nalog precej omejen in tekmovalci imajo za eno nalogo v povprečju le slabo uro časa. To med drugim pomeni, da je marsikak zanimiv problem s področja računalništva težko zastaviti v obliki, ki bi bila primerna za nalogo na tekmovanju; pa tudi tekmovalec si ne more privoščiti, da bi se v nalogo poglobil tako temeljito, kot bi se mogoče zmogel, saj mu za to preprosto zmanjka časa.

Off-line naloga je poskus, da se tovrstnim omejitvam malo izognemo: besedilo naloge in testni primeri zanjo so objavljeni več mesecev vnaprej, tekmovalci pa ne oddajajo programa, ki rešuje nalogo, pač pa oddajajo rešitve tistih vnaprej objavljenih testnih primerov. Pri tem imajo torej veliko časa in priložnosti, da dobro razmislijo o nalogi, preizkusijo več možnih pristopov k reševanju, počasi izboljšujejo svojo rešitev in podobno. Opis naloge in testne primere smo objavili decembra 2022, nekaj mesecev po razpisu za tekmovanje v znanju; tekmovalci so imeli čas do 24. marca 2023 (dan pred tekmovanjem), da pošljejo svoje rešitve.

Opis naloge

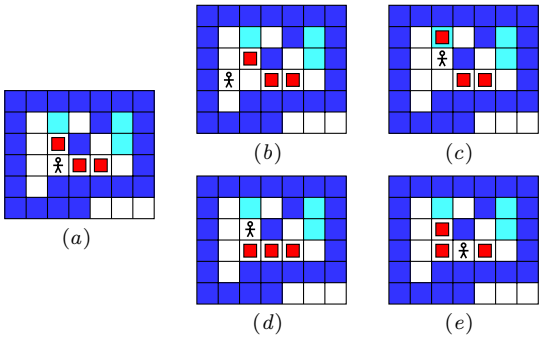
Pri tej nalogi se bomo ukvarjali z malo spremenjeno različico znane igre Sokoban. Dana je karirasta mreža, ki predstavlja tloris skladišča. Vsaka celica v mreži je bodisi prehodna bodisi zazidana. Na nekaterih prehodnih celicah stojijo zaboji (največ po en zaboj na celico). Poleg tega so na nekaterih prehodnih celicah *odlagališča*. Število odlagališč in zabojev je enako.

Na eni od prehodnih celic (taki, kjer ni zaboja) stoji skladiščnik, ki se lahko sprehaja po mreži; v vsakem koraku se lahko premakne za eno celico gor, dol, levo ali desno (seveda le po prehodnih celicah). Skladiščnik lahko tudi premika zaboje, in sicer na naslednja dva načina: (1) ob premiku lahko zaboj rine pred sabo, vendar le, če je naslednja celica v tisti smeri prehodna in brez zaboja; (2) lahko pa zaboj s celice, na katero bi se rad premaknil, prestavi na celico, na kateri zdaj stoji skladiščnik.[43]

Slika na str. 226 kaže primer stanja mreže in možnih premikov. Temno modra polja so zidovi, sinje modra so odlagališča, bela so prehodna polja; manjši rdeči kvadratki predstavljajo zaboje. Če je skladišče v stanju, prikazanem na sliki (a), so možni naslednji premiki: (b) premik levo na polje brez zaboja; (c) premik tipa (1) gor, pri čemer skladiščnik rine zaboj pred sabo; (d) premik tipa (2) gor, pri čemer skladiščnik prestavi zaboj na svoj dosedanji položaj; in (e) premik tipa (2) desno, pri čemer skladiščnik prestavi zaboj na svoj dosedanji položaj. Premik tipa (1) v smeri desno ni mogoč, ker polje onkraj zaboja tam ni prazno — skladiščnik ne more pred seboj porivati dveh ali več zabojev; premik dol tudi ni mogoč, ker je tam zid, ne pa prehodno polje.

Tvoja naloga je spraviti čim več zabojev na odlagališča (pri tem ni pomembno, kateri zaboj stoji na katerem odlagališču), pri tem pa izvesti čim manj premikov zabojev. Ko zaboj premakneš na odlagališče, ni nujno, da tam tudi ostane; lahko

[43] Tale drugi tip premika zabojev je tisto, po čemer se naša naloga razlikuje od običajne igre Sokoban; pri slednji so dovoljeni le premiki zabojev z rinjenjem. Drugi tip premika smo dodali zato, da bi si tekmovalci težje pomagali s raznimi programi za reševanje običajne različice Sokobana in z obstoječo literaturo o tej igri.



ga kasneje premakneš stran z njega. Vse, kar šteje, je to, koliko zabojev stoji na odlagališčih v končnem stanju mreže, po vseh premikih.

Ocenjevanje: boljša je tista rešitev, ki spravi na odlagališča več zabojev. Če sta dve rešitvi po tem kriteriju enaki, je boljša tista, pri kateri skladiščnik manjkrat premakne zaboje (štejejo tako premiki tipa 1 kot premiki tipa 2). Če sta rešitvi tudi po tem kriteriju enaki, je boljša tista z manjšim skupnim številom korakov. (Če se ujemata tudi po tem, veljata rešitvi za enako dobri.)

Rezultati

Sistem točkovanja je bil tak kot pri off-line nalogah v prejšnjih letih. Pripravili smo 30 testnih primerov, pri vsakem testnem primeru smo razvrstili tekmovalce po oceni njegove rešitve, nato pa je prvi tekmovalec (tisti, čigar rešitev je imela najmanjšo oceno) dobil 10 točk, drugi 8, tretji 7 in tako naprej po eno točko manj za vsako naslednje mesto (osmi dobi dve točki, vsi nadaljnji pa po eno). Na koncu smo za vsakega tekmovalca sešteli njegove točke po vseh 30 testnih primerih.

Testni primeri so bili treh različnih velikosti: deset majhnih (w in h med 10 in 20), deset srednjih (w in h med 30 in 40) in deset velikih (w in h med 80 in 100). Majhne primere smo pobrali iz zbirke Davida W. Skinnerja (primeri, ki naj bi bili izziv pri originalni različici igre Sokoban),[44] srednje in velike pa smo sestavili naključno. Pri slednjih začnemo z generiranjem labirinta (ki ima pri različnih primerih različno široke hodnike), nato pa dodamo zaboje in odlagališča; zabojev je pri nekaterih primerih malo (od 15 do 20), pri nekaterih zelo veliko (do 1000); pri nekaterih primerih so zaboji pretežno ločeni od odlagališč (ležijo na drugem koncu mreže), pri nekaterih pa so zaboji in odlagališča pomešana.

Letos so svoje rešitve pri off-line nalogi poslali štirje tekmovalci, od tega trije srednješolci in en študent. Končna razvrstitev je naslednja:

Mesto	Ime	Letnik	Šola	Točke
1	Luka Stražišar	1	FRI	288
2	Mai Rupnik	4	ŠC Nova Gorica, ERŠ	260
3	Aljaž Brodar	4	ŠC Nova Gorica, ERŠ	186
4	Brin Blažko	2	ŠC Nova Gorica, ERŠ	173

[44] `http://www.abelmartin.com/rj/sokobanJS/Skinner/David%20W.%20Skinner%20-%20Sokoban.htm`

Rešitev

Prvotno različico igre Sokoban običajno rešujejo z raznimi postopki za hevristično preiskovanje prostora vseh možnih stanj (npr. A* ali iterativno poglabljanje) in načeloma bi lahko poskusili s čim takim tudi pri naši nalogi, vendar le na najmanjših testnih primerih.

Za večje testne primere lahko poskusimo takole: izberimo si neki zaboj in neko odlagališče in poskusimo spraviti ta zaboj na to odlagališče, pri tem pa naj zaboji, ki že zdaj stojijo na odlagališčih, tam tudi ostanejo. Naslednji primer kaže, kako se lahko skladiščnik premakne mimo skupine zabojev (`S` = skladiščnik, `Z` = zaboj, `.` = prazna celica, `#` = zid):

$$\texttt{SZZZ.} \xrightarrow{2} \texttt{ZSZZ.} \xrightarrow{2} \texttt{ZZSZ.} \xrightarrow{1} \texttt{ZZ.SZ} \longrightarrow \texttt{ZZS.Z} \xrightarrow{2} \texttt{ZSZ.Z} \xrightarrow{1} \texttt{Z.SZZ} \longrightarrow$$
$$\texttt{ZS.ZZ} \xrightarrow{2} \texttt{SZ.ZZ} \xrightarrow{1} \texttt{.SZZZ} \xrightarrow{2} \texttt{.ZSZZ} \xrightarrow{2} \texttt{.ZZSZ} \xrightarrow{2} \texttt{.ZZZS}$$

Tu je `S` prišel mimo strnjene skupine treh `Z`-jev (v 12 korakih, od katerih je 10 premikov zabojev). Če pa na koncu izvedemo še eno potezo drugega tipa, s katero pridemo v stanje `.ZZSZ`, si lahko vse skupaj predstavljamo tako, kot da je par `SZ` prišel mimo strnjene skupine dveh `Z`-jev — z drugimi besedami, skladiščnik je rinil zaboj pred sabo mimo take strnjene skupine (ki je na koncu ostala na istih celicah kot na začetku). Ni si težko predstavljati, da lahko ta primer posplošimo tudi na daljše skupine.

Kaj pa, če se želimo premikati po poti z ovinki, ne le naravnost? Za premik tipa 2 in za premik skladiščnika na prosto celico ovinki niso problem, ker v teh premikih tako ali tako sodelujeta le dve sosednji celici. Zaplete se le pri premiku tipa 1 (ko skladiščnik rine zaboj pred sabo), ki načeloma potrebuje tri celice v ravni črti; kaj naj naredimo, če hočemo riniti zaboj okrog vogala? Izkaže se, da lahko naredimo tudi to, če imamo na zunanji strani ovinka dve prehodni celici, recimo a in b; spodnji primer kaže ovinek v desno okrog zidu `#`:

$$\begin{matrix}\texttt{aZ.}\\ \texttt{bS\#}\end{matrix} \longrightarrow \begin{matrix}\texttt{aZ.}\\ \texttt{Sb\#}\end{matrix} \longrightarrow \begin{matrix}\texttt{SZ.}\\ \texttt{ab\#}\end{matrix} \longrightarrow \begin{matrix}\texttt{.SZ}\\ \texttt{ab\#}\end{matrix} \longrightarrow \begin{matrix}\texttt{S.Z}\\ \texttt{ab\#}\end{matrix} \longrightarrow \begin{matrix}\texttt{a.Z}\\ \texttt{Sb\#}\end{matrix} \longrightarrow \begin{matrix}\texttt{a.Z}\\ \texttt{bS\#}\end{matrix} \longrightarrow \begin{matrix}\texttt{aSZ}\\ \texttt{b.\#}\end{matrix}$$

Na notranji strani ovinka imamo lahko zid `#`, pa smo celice `SZ.` okrog tega zidu vseeno uspešno spremenili v `.SZ`; celici a in b na zunanjem robu pa imata na koncu enako stanje kot na začetku. Ta scenarij deluje ne glede na to, ali sta na a in b zaboja ali ne, pomembno je le, da sta celici prehodni, ne pa zazidani. Podoben scenarij deluje tudi, če imamo prehodni celici na zgornji strani takega ovinka namesto na levi:

$$\begin{matrix}\texttt{ab}\\ \texttt{Z.}\\ \texttt{S\#}\end{matrix} \to \begin{matrix}\texttt{ab}\\ \texttt{S.}\\ \texttt{Z\#}\end{matrix} \to \begin{matrix}\texttt{ab}\\ \texttt{.S}\\ \texttt{Z\#}\end{matrix} \to \begin{matrix}\texttt{aS}\\ \texttt{.b}\\ \texttt{Z\#}\end{matrix} \to \begin{matrix}\texttt{Sa}\\ \texttt{.b}\\ \texttt{Z\#}\end{matrix} \to \begin{matrix}\texttt{.a}\\ \texttt{Sb}\\ \texttt{Z\#}\end{matrix} \to \begin{matrix}\texttt{.a}\\ \texttt{Zb}\\ \texttt{S\#}\end{matrix} \to \begin{matrix}\texttt{Za}\\ \texttt{Sb}\\ \texttt{.\#}\end{matrix} \to \begin{matrix}\texttt{Sa}\\ \texttt{Zb}\\ \texttt{.\#}\end{matrix} \to \begin{matrix}\texttt{aS}\\ \texttt{Zb}\\ \texttt{.\#}\end{matrix} \to \begin{matrix}\texttt{ab}\\ \texttt{ZS}\\ \texttt{.\#}\end{matrix} \to \begin{matrix}\texttt{ab}\\ \texttt{SZ}\\ \texttt{.\#}\end{matrix}$$

Zdaj torej znamo simulirati rinjenje (premik tipa 1) okrog ovinka in lahko prej omenjeni scenarij, ko se je skladiščnik premaknil (ali celo rinil zaboj) mimo strnjene skupine več zabojev, posplošimo na poljubno pot z ovinki, le s to omejitvijo, da moramo imeti pri vsakem ovinku na zunanji strani dve prehodni celici (kot smo pravkar videli).

Pri izbiri poti, po kateri bi šel skladiščnik do zaboja, ki ga hočemo spraviti na odlagališče, in nato pri izbiri poti, po kateri bo ta zaboj dejansko pririnil do

odlagališča, si lahko še vedno pomagamo z neke vrste iskanjem v širino, vendar v opis stanja vključimo poleg trenutnega položaja še smer prejšnjega premika, da bomo lahko zaznali ovinke in v teh primerih preverili, ali imamo na zunanji strani dve prehodni celici, ki ju potrebujemo.

Poleg tega lahko pri izbiri poti tudi upoštevamo ceno premikov. Premik mimo skupine zabojev je dražji kot premik mimo skupine praznih celic; tudi pri rinjenju okrog vogala je celotna operacija dražja, če sta na a in b zaboja, kot če sta tidve celici prazni. To je posledica dejstva, da imajo pri ocenjevanju večjo težo premiki zabojev kot pa premiki skladiščnika na prazne celice (brez premikov zaboja).

Še ena stvar, ki jo lahko upoštevamo pri ocenjevanju možne poti, je, da če zaboji, ki nam stojijo na poti, še niso na odlagališčih, potem ni velike škode, če jih premaknemo, in potem je pot lahko veliko cenejša. Na primer, zgoraj smo videli, da lahko pridemo mimo skupine treh zabojev v 12 korakih; toda če dovolimo, da so ti zaboji na koncu drugje kot na začetku, gre že v 3 korakih:

$$\texttt{SZZZ.} \xrightarrow{2} \texttt{ZSZZ.} \xrightarrow{2} \texttt{ZZSZ.} \xrightarrow{2} \texttt{ZZZS.}$$

Za konec pa omenimo še možnost, da nalogo rešujemo ročno: napišemo si program, s katerim bomo lahko igrali našo prilagojeno različico igre Sokoban, program pa bo tudi sproti shranjeval naše premike. Na ta način se je lotila naloge tudi večina naših letošnjih tekmovalcev.

UNIVERZITETNI PROGRAMERSKI MARATON

Društvo ACM Slovenija sodeluje tudi pri pripravi študenskih tekmovanj v programiranju, ki v zadnjih letih potekajo pod imenom Univerzitetni programerski maraton (UPM, `tekmovanja.acm.si/upm`) in so odskočna deska za udeležbo na ACMovih mednarodnih študentskih tekmovanjih v programiranju (International Collegiate Programming Contest, ICPC). Ker UPM ne izdaja samostojnega biltena, bomo na tem mestu na kratko predstavili to tekmovanje in njegove letošnje rezultate.

Na študentskih tekmovanjih ACM v programiranju tekmovalci ne nastopajo kot posamezniki, pač pa kot ekipe, ki jih sestavljajo po največ trije člani. Vsaka ekipa ima med tekmovanjem na voljo samo en računalnik. Naloge so podobne tistim iz tretje skupine našega srednješolskega tekmovanja, le da so včasih malo težje oz. predvsem predpostavljajo, da imajo reševalci že nekaj več znanja matematike in algoritmov, ker so to stvari, ki so jih večinoma slišali v prvem letu ali dveh študija. Časa za tekmovanje je pet ur, nalog pa je praviloma 6 do 8, kar je več, kot jih je običajna ekipa zmožna v tem času rešiti. Za razliko od našega srednješolskega tekmovanja pri študentskem tekmovanju niso priznane delno rešene naloge; naloga velja za rešeno šele, če program pravilno reši vse njene testne primere. Ekipe se razvrsti po številu rešenih nalog, če pa jih ima več enako število rešenih nalog, se jih razvrsti po času oddaje. Za vsako uspešno rešeno nalogo se šteje čas od začetka tekmovanja do uspešne oddaje pri tej nalogi, prišteje pa se še po 20 minut za vsako neuspešno oddajo pri tej nalogi. Tako dobljeni časi se seštejejo po vseh uspešno rešenih nalogah in ekipe z istim številom rešenih nalog se potem razvrsti po skupnem času (manjši ko je skupni čas, boljša je uvrstitev).

UPM poteka v štirih krogih (dva spomladi in dva jeseni), pri čemer se za končno razvrstitev pri vsaki ekipi zavrže najslabši rezultat iz prvih treh krogov, četrti (finalni) krog pa se šteje dvojno. Najboljše ekipe se uvrstijo na srednjeevropsko regijsko tekmovanje (CERC, ki je potekalo 9.–10. decembra 2023 v Ljubljani), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki je potekalo 15.–20. septembra 2024 v Astani v Kazahstanu).

Na letošnjem UPM je sodelovalo 28 ekip s skupno 81 tekmovalci, ki so prišli s treh slovenskih univerz, nekaj pa je bilo celo srednješolcev. Tabela na naslednji strani prikazuje vse ekipe, ki so se pojavile na vsaj enem krogu tekmovanja.

	Ekipa	Št. rešenih nalog*	Čas
1	Žiga Željko, Marko Hostnik (FRI + FMF), Urban Duh (FMF)	29	46:42:02
2	Benjamin Bajd,[1] Jakob Schrader, Job Petrovčič, Luka Horjak[2] (FMF)	28	44:40:30
3	Bor Grošelj Simić, Patrik Žnidaršič, Ella Potisek (FMF)	23	36:53:50
4	Matevž Miščič, Jon Mikoš (FMF), Domen Hočevar (FRI + FMF)	23	41:20:39

* 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.

[1] Le v prvih dveh krogih. — [2] Le v zadnjih dveh krogih.

(*nadaljevanje na naslednji strani*)

	Ekipa	Št. rešenih nalog*	Čas
5	Bor Brudar (ŠC NM, SEŠTG), Jakob Žorž, Nejc Sušnik (Gim. Šk. Loka)	20	33:31:13
6	Mitko Nikov, Alen Granda (FERI)	17	28:08:31
7	Filip Štamcar, Jakob Kralj, Jošt Smrtnik (Gim. Vič)	16	22:44:47
8	Luka Stražišar, Matej Kralj (FRI + FMF), Anže Hočevar (FMF)	14	18:10:40
9	Filip Trplan, Jernej Oblak, Đorđe Stevanović (FRI + FMF)	13	24:53:59
10	Ana Luetić (FMF), Barbara Makovec, Tine Zaletelj (Gim. Vič)	12	20:22:50
11	Matija Kocbek, Luka Horjak[1] (FMF), Lovro Drofenik (F. za strojništvo, Lj.)	7	14:33:27
12	Nina Sangawa Hmeljak (FRI + FMF), Jan Hrastnik, Jakob Pogačnik Souvent (FMF)	4	6:56:19
13	Miloš Jevtović, Vasilije Božarić, Jovan Pavlović (FAMNIT)	3	2:52:10
14	Aleks Stepančič, Aljaž Medič, Luka Potočnik (FRI + FMF)	3	3:32:25
15	Vili Perše, Sven Ahac, Tilen Gimpelj (FAMNIT)	3	4:35:59
16	Andrej Matos, Rok Hladin, Vojko Hysz (FRI)	3	7:17:32
17	Matjaž Madon, Jakob Beber, Luka Uršič (FAMNIT)	3	7:39:25
18	Nik Čoh, Žan Ferenčak, Tevž Beškovnik (FERI)	3	8:20:39
19	Matej Kodermac, Matevž Kavčič, Etian Križman (FAMNIT)	2	2:13:49
20	Kiryl Buloichyk, Dmytro Tupkalenko, Ryszard Sandak (FAMNIT)	2	4:38:42
21	Patrik Gobec, Gal Habjan, Jaša Marolt (FERI)	2	5:28:45
22	Janko Kondić, Kristjan Kotnik, Patrik Hudi (FAMNIT)	2	7:25:24
23	Matjaž Vuherer (FRI), Alja Mavec Krovinović, Marija Zila Šikovc (FAMNIT)	2	8:59:23
24	Amar Ustavdić, Robert Gavranović, Valeryia Bazhko (FAMNIT)	2	10:11:35
25	Vid Šafranko (FERI)	1	1:48:38
26	Peter Andolšek (Gim. Bežigrad), Alexander Gaydukov (Gim. Koper), Leon Fišer (Vegova Lj.)	1	2:34:08
27	Jakob Lipovec, Merih Hušidić, Gašper Topolinjak (FNM)	0	0:00:00
	Zhivko Stoimchev, Andrej Natev, Danilo Lazić (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.

[1] Le v prvih dveh krogih.

Na srednjeevropskem tekmovanju CERC 2023 so (z nekaterimi spremembami v sestavi) nastopile ekipe 2, 3 in 4 kot predstavnice Univerze v Ljubljani, ekipa 6 kot predstavnica Univerze v Mariboru in ekipa 13 kot predstavnica Univerze na Primorskem. V konkurenci 60 ekip s 25 univerz iz 7 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
27	Matevž Miščič, Jon Mikoš, Domen Hočevar	4	8:48:49
31	Job Petrovčič, Luka Horjak, Jakob Schrader	4	10:26:29
35	Bor Grošelj Simić, Patrik Žnidaršič, Ella Potisek	4	12:03:37
36	Alen Granda, Mitko Nikov	4	13:54:29
58	Miloš Jevtović, Vasilije Božarić, Jovan Pavlović	1	1:21:59

Na srednjeevropskem tekmovanju je bilo 12 nalog, od tega jih je zmagovalna ekipa rešili deset.

ANKETA

V letih 2020 in 2021, ko je tekmovanje potekalo prek interneta, smo na ta način izvedli tudi anketo (prek spletne strani `1ka.si`). Ker se je ta način anketiranja dobro obnesel, smo ga obdržali tudi od 2022 naprej, čeprav je tekmovanje drugače spet potekalo v živo. Vprašanja na anketi so prikazana spodaj in so bila tudi letos približno enaka kot včasih, 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? _____

_____

[Le pri 3. skupini.] Letos v tretji skupini podpiramo reševanje nalog v pascalu, C, C++, C#, javi, pythonu in rustu. Bi rad uporabljal kakšen drug programski jezik? Če da, katerega?

_____

Katere od naslednjih jezikovnih konstruktov in programerskih prijemov znaš uporabljati?	ne poznam	da, slabo	da, dobro
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz standardnega vhoda ali pa ju zapisati na standardni izhod?	☐	☐	☐
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz datoteke ali pa ju zapisati v datoteko?	☐	☐	☐
Tabele (**array**):			
— enodimenzionalne	☐	☐	☐
— dvodimenzionalne	☐	☐	☐
— večdimenzionalne	☐	☐	☐
Znaš napisati svoj podprogram oz. funkcijo	☐	☐	☐
Poznaš rekurzijo	☐	☐	☐

Kazalce, dinamično alokacijo pomnilnika (New/Dispose, GetMem/FreeMem, malloc/free, **new**/**delete**, ...) ☐ ☐ ☐
Zanka **for** ☐ ☐ ☐
Zanka **while** ☐ ☐ ☐
Gnezdenje zank (ena zanka znotraj druge) ☐ ☐ ☐
Naštevni tipi (*enumerated types* — **type** ImeTipa = (Ena, Dve, Tri) v pascalu, **typedef enum** v C/C++) ☐ ☐ ☐
Strukture (**record** v pascalu, **struct**/**class** v C/C++) ☐ ☐ ☐
and, **or**, **xor**, **not** kot aritmetični operatorji (nad biti celoštevilskih operandov namesto nad logičnimi vrednostmi tipa boolean) (v C/C++/C#/javi: &, |, ^, ~) ☐ ☐ ☐
Operatorja **shl** in **shr** (v C/C++/C#/javi: <<, >>) ☐ ☐ ☐
Znaš uporabiti kakšnega od naslednjih razredov iz standardnih knjižnic:
— razpršeno tabelo: hash_map, hash_set, unordered_map, unordered_set (v C++), Hashtable, HashSet (v javi/C#), Dictionary (v C#), dict, set (v pythonu) ☐ ☐ ☐
— iskalna drevesa: map, set (v C++), TreeMap, TreeSet (v javi), SortedDictionary (v C#) ☐ ☐ ☐
— kopico oz. prioritetno vrsto: priority_queue (v C++), PriorityQueue (v javi), heapq (v pythonu) ☐ ☐ ☐

[Naslednja skupina vprašanj se je ponovila za vsako nalogo po enkrat.]
Zahtevnost naloge: ☐ prelahka ☐ lahka ☐ primerna ☐ težka ☐ pretežka ☐ ne vem
Naloga je (ali: bi) vzela preveč časa: ☐ da ☐ ne ☐ ne vem
Mnenje o besedilu naloge:
— dolžina besedila: ☐ prekratko ☐ primerno ☐ predolgo
— razumljivost besedila: ☐ razumljivo ☐ težko razumljivo ☐ nerazumljivo
Naloga je bila: ☐ zanimiva ☐ dolgočasna ☐ že znana ☐ povprečna
Si jo rešil(a)?
☐ nisem rešil(a), ker mi je zmanjkalo časa za reševanje
☐ nisem rešil(a), ker mi je zmanjkalo volje za reševanje
☐ nisem rešil(a), ker mi je zmanjkalo znanja za reševanje
☐ rešil(a) sem jo le delno, ker mi je zmanjkalo časa za reševanje
☐ rešil(a) sem jo le delno, ker mi je zmanjkalo volje za reševanje
☐ rešil(a) sem jo le delno, ker mi je zmanjkalo znanja za reševanje
☐ rešil(a) sem celo
Ostali komentarji o tej nalogi: _____

Katera naloga ti je bila najbolj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? _____

Katera naloga ti je bila najmanj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? _____

Na letošnjem tekmovanju ste imeli tri ure / pet ur časa za pet nalog.
Bi imel(a) raje: ☐ več časa ☐ manj časa ☐ časa je bilo ravno prav
Bi imel(a) raje: ☐ več nalog ☐ manj nalog ☐ nalog je bilo ravno prav

Kakršne koli druge pripombe in predlogi. Kaj bi spremenil(a), popravil(a), odpravil(a), ipd., da bi postalo tekmovanje zanimivejše in bolj privlačno? _____

Kaj ti je bilo pri tekmovanju všeč? _____

Kaj te je najbolj motilo? _____

Če imaš kaj vrstnikov, ki se tudi zanimajo za programiranje, pa se tega tekmovanja niso udeležili, kaj bi bilo po tvojem mnenju treba spremeniti, da bi jih prepričali k udeležbi?

Ali si pri izpolnjevanju ankete prišel/la do sem? ☐ da ☐ ne

Hvala za sodelovanje in lep pozdrav! Tekmovalna komisija

REZULTATI ANKETE

Anketo je izpolnilo 31 tekmovalcev prve skupine, 11 tekmovalcev druge skupine in 10 tekmovalcev tretje skupine. (Opozorimo na to, da je zaradi majhnega števila tekmovalcev v drugi in tretji skupini iz tamkajšnjih anket še posebej težko vleči kakšne pametne posplošitve in zaključke.)

Mnenje tekmovalcev o nalogah

Tekmovalce smo spraševali: kako zahtevna se jim zdi posamezna naloga; ali se jim zdi, da jim vzame preveč časa; ali je besedilo primerno dolgo in razumljivo; ali se jim zdi naloga zanimiva; ali so jo rešili (oz. zakaj ne); in katera naloga jim je bila najbolj/najmanj všeč.

Rezultate vprašanj o zahtevnosti nalog kažejo grafi na str. 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. 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: 3,24 v prvi skupini (v prejšnjih letih 3,35, 3,27, 2,97, 3,47, 3,32), 3,61 v drugi skupini (prejšnja leta 3,39, 3,55, 3,38, 3,17, 3,19) in 3,40 v tretji skupini (prejšnja leta 3,80, 3,63, 3,67, 3,52, 3,59).

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 običajno ($R^2 = 0{,}35$; v prejšnjih letih 0,48, 0,42, 0,68, 0,71, 0,67).

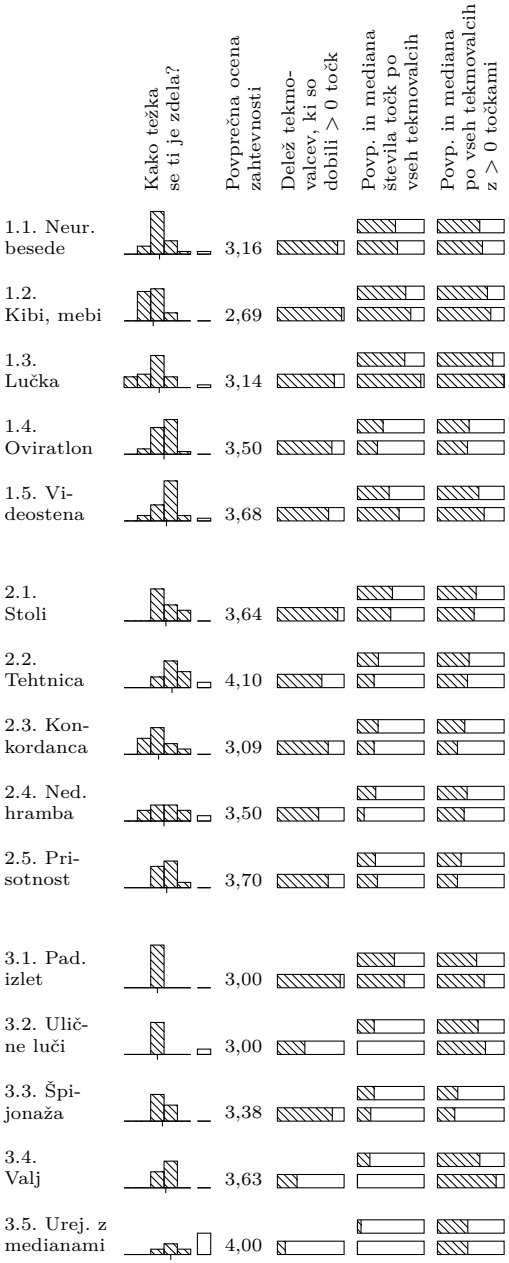
V prvi skupini so tekmovalci kot težjo ocenili predvsem nalogo 1.5 (videostena), deloma pa tudi 1.4 (oviratlon); pri slednji je to lažje razumeti, ker je geometrijska. Kot najlažjo pa so ocenili nalogo 1.2 (kibi, mebi).

V drugi skupini je kot težja izstopala naloga 2.2 (tehtnica), morda zato, ker je za rešitev zahtevala bodisi nekaj matematičnega razmišljanja bodisi rekurzijo. Kot najlažjo pa so v tej skupini ocenili nalogo 2.3 (konkordanca); toda pogled na njihove rešitve kaže, da so jo mnogi razumeli narobe in si jo tako poenostavili (npr. obravnavali vsako vrstico posebej, ločeno od ostalih).

V tretji skupini se jim je zdela težja predvsem naloga 3.5 (urejanje z medianami), najbrž zato, ker je bolj nekonvencionalne oblike (interaktivna naloga). Najlažji sta se jim zdeli nalogi 3.1 (padalski izlet) in 3.2 (ulične luči), ki sta bili tudi mišljeni kot lažji.

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 237. Nad razumljivostjo besedil ni veliko pripomb, podobno kot prejšnja leta, v prvi skupini še malo manj. Kot težje razumljive so ocenili predvsem naloge 3.4 (valj; ta še posebej izstopa kot težko razumljiva), 2.4 (nedeljiva hramba) in 2.2 (tehtnica).

Tudi z dolžino besedil so tekmovalci večinoma zadovoljni; ocene so podobne kot prejšnja leta. Po komentarjih, da je naloga predolga, še najbolj izstopajo 1.4 (oviratlon), 2.4 (nedeljiva hramba), 3.3 (špijonaža) in 3.4 (valj). Mnenj, da je besedilo



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

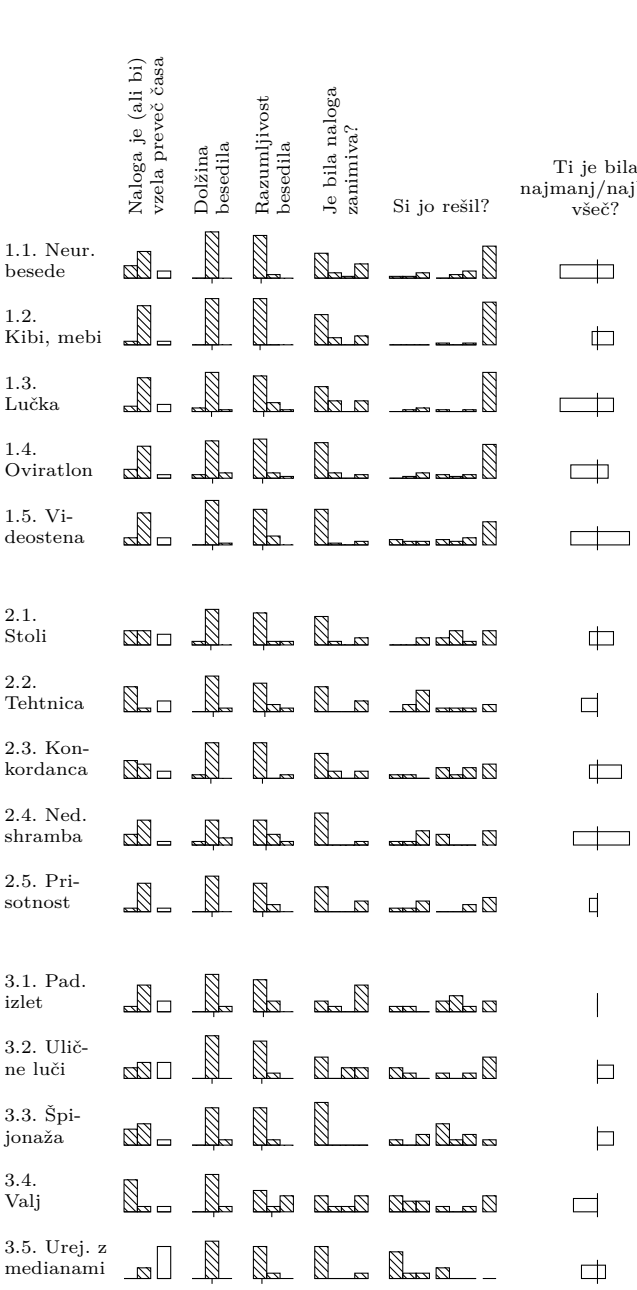
Pomen stolpcev v vsaki vrstici:

Na levi je skupina šestih stolpcev, ki kažejo, kako so tekmovalci v anketi odgovarjali na vprašanje o zahtevnosti naloge. Stolpci po vrsti pomenijo odgovore „prelahka“, „lahka“, „primerna“, „težka“, „pretežka“ in „ne vem“. Višina stolpca pove, koliko tekmovalcev je izrazilo takšno mnenje o zahtevnosti naloge. Desno od teh stolpcev je povprečna ocena zahtevnosti (1 = prelahka, 3 = primerna, 5 = pretežka). Povprečno oceno kaže tudi črtica pod to skupino stolpcev.

Sledi stolpec, ki pokaže, kolikšen delež tekmovalcev je pri tej nalogi dobil več kot 0 točk. Naslednji par stolpcev pokaže povprečje (zgornji stolpec) in mediano (spodnji stolpec) števila točk pri vsej nalogi. Zadnji par stolpcev pa kaže povprečje in mediano števila točk, gledano le pri tistih tekmovalcih, ki so dobili pri tisti nalogi več kot nič točk.

Mnenje tekmovalcev o nalogah

Višina stolpcev pove, koliko tekmovalcev je dalo določen odgovor na neko vprašanje.



Stolpci se od leve proti desni nanašajo na naslednja vprašanja in možne odgovore:

Naloga je (ali: bi) vzela preveč časa:

- da
- ne
- ne vem

Dolžina besedila:

- prekratko
- primerno
- predolgo

Razumljivost besedila:

- razumljivo
- težko razumljivo
- nerazumljivo

Naloga je bila:

- zanimiva
- dolgočasna
- že znana
- povprečna

Si jo rešil?

- nisem, zmanjkalo časa
- nisem, zmanjkalo volje
- nisem, zmanjkalo znanja
- delno, zmanjkalo časa
- delno, zmanjkalo volje
- delno, zmanjkalo znanja
- rešil sem celo

Katera naloga ti je bila najmanj všeč? Katera ti je bila najbolj všeč?

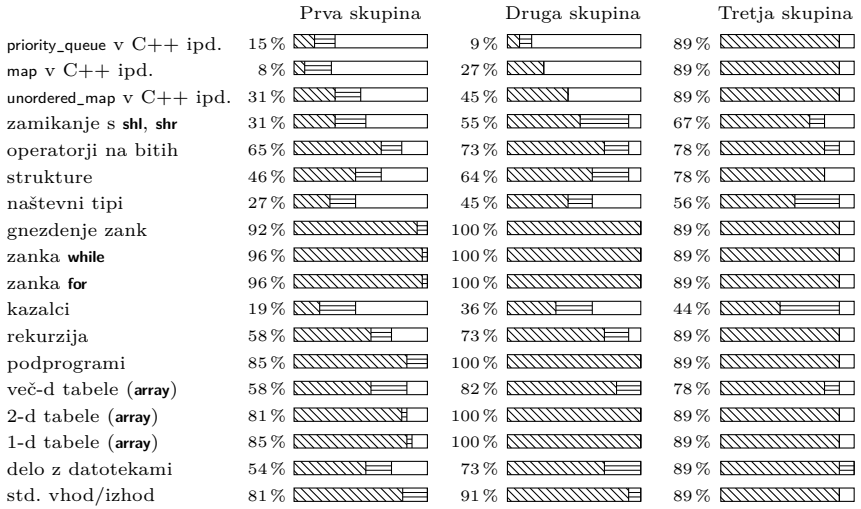


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

prekratko, je bilo malo, še največ pri nalogah 1.3 (lučka) in 1.4 (oviratlon; pri tej nalogi se je sicer nekaterim zdelo besedilo tudi predolgo).

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta. Kot bolj zanimive izstopajo 1.5 (videostena), 2.4 (nedeljiva hramba) in 3.3 (špijonaža), kot manj zanimive pa 3.1 (padalski izlet), 3.4 (valj) in 1.3 (lučka).

Pripomb, da jim je neka naloga že znana, je bilo letos malo: po ena pri nalogah 1.1 (neurejene besede) in 3.4 (valj) ter dve pri 3.2 (ulične luči).

Pripomb, da bi naloga vzela preveč časa, je bilo malo, v drugi skupini sicer več kot ponavadi. Delež takih pripomb je bil velik zlasti pri nalogah 2.2 (tehtnica) in 3.4 (valj), deloma tudi pri 2.1 (stoli) in 2.3 (konkordanca). Pri 2.3 in 3.4 je z implementacijo res nekaj več dela.

Pri vprašanjih, katera naloga je tekmovalcu najbolj všeč in katera najmanj, so bili glasovi letos precej razpršeni med naloge, še posebej v prvi skupini; tako sta na primer nalogi 1.5 (videostena) in 2.4 (nedeljiva hramba) dobili veliko glasov pri obeh vprašanjih. Kot bolj nepriljubljene izstopajo naloge 1.4 (oviratlon), 2.2 (tehtnica) in 3.4 (valj).

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 in tretji skupini pravijo, da znajo malo manj kot tisti v lanski anketi. Stvari, ki jih tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja

	Prva skupina	Druga skupina	Tretja skupina
drevo	47 %	45 %	90 %
hash tabela	29 %	45 %	70 %
seznam (linked list)	47 %	73 %	80 %
sklad (stack)	29 %	64 %	90 %
vrsta (queue)	35 %	82 %	90 %
Evklidov algoritem	90 %	73 %	90 %
Eratostenovo rešeto	65 %	55 %	100 %
vektorski produkt	55 %	36 %	90 %
rekurzija	39 %	36 %	80 %
dinamično prog.	19 %	36 %	90 %
iskanje v širino	19 %	27 %	90 %
O-zapis	43 %	73 %	100 %
urejanje	77 %	82 %	100 %
bubble sort	38 %	55 %	80 %
insertion sort	25 %	18 %	60 %
selection sort	25 %	18 %	60 %
quicksort	31 %	36 %	80 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

leta: kazalci, naštevni tipi in deloma operatorji na bitih, v prvi in drugi skupini tudi naprednejše podatkovne strukture. Poznavanje operatorjev na bitih je letos boljše kot ponavadi.

Uporaba programskih jezikov

Na splošno so razmerja med različnimi jeziki podobna kot v prejšnjih letih. V prvi skupini je tudi letos python daleč najpogostejši, sledita pa mu C/C++ in java; C# je letos razmeroma redek, podobno kot lani. V drugi skupini je tokrat prvič po več letih najpogostejši C/C++, sledi mu python, peščica ljudi pa je uporabljala javo ali C#. V tretji skupini večina tekmovalcev uporablja C++, tako kot ponavadi, vendar je nekaj tudi uporabnikov pythona in jave. Edini jezik, ki se je še pojavil poleg doslej omenjenih, je javascript, ki so ga uporabljali trije tekmovalci v prvi skupini.

Podobno kot prejšnja leta se je tudi letos pojavilo nekaj tekmovalcev, ki oddajajo le ali pretežno rešitve v psevdokodi, tudi tam, kjer naloga sicer zahteva izvorno kodo v kakšnem konkretnem programskem jeziku. Iz tega bi človek mogoče sklepal, da bi bilo dobro dati več nalog tipa „opiši postopek“ (namesto „napiši podprogram“), vendar se v praksi običajno izkaže, da so takšne naloge med tekmovalci precej manj priljubljene in da si večinoma ne predstavljajo preveč dobro, kako bi opisali postopek (pogosto v resnici oddajo dolgovezne opise izvorne kode v stilu „nato bi s stavkom **if** preveril, ali je spremenljivka x večja od spremenljivke y“). Podobno kot prejšnja leta smo tudi letos pri nalogah tipa „opiši postopek“ pripisali „ali napiši podprogram (kar ti je lažje)“ (kjer je bilo to primerno).

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže tabela na str. 240. 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

	Leto in skupina																	
	2023			2022			2021			2020			2019			2018		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal							4			2			2					
C	6	$4\frac{1}{2}$		3	$3\frac{1}{2}$		8	5	$\frac{1}{2}$	$3\frac{1}{2}$	3		10	4	$\frac{1}{2}$	5	4	$\frac{1}{2}$
C++	20	7	$13\frac{1}{2}$	14	6	10	13	$11\frac{1}{2}$	$18\frac{1}{2}$	$26\frac{1}{2}$	8	14	$21\frac{1}{2}$	$7\frac{1}{2}$	18	$18\frac{1}{2}$	13	11
java	$20\frac{1}{2}$	2	2	22			17		5	15	4	3	15	5	1	$21\frac{1}{2}$	$8\frac{1}{2}$	4
PHP			–			–			–	1		–			–			–
C#	2	1		2	1		6	4		6	3		12	2		11	6	
python	$29\frac{1}{2}$	$6\frac{1}{2}$	$1\frac{1}{2}$	29	$16\frac{1}{2}$	4	43	$15\frac{1}{2}$	4	48	20	3	$36\frac{1}{2}$	$26\frac{1}{2}$	$6\frac{1}{2}$	38	11	$\frac{1}{2}$
javascript	3		–	2	1	–	1	2	–	2		–			–		$\frac{1}{2}$	–
swift			–			–			–	1		–			–			–
rust						1			–			–			–			–
psevdokoda	1		–	4	1	–	3		–	2	1	–	5	1	–	3	1	–
nič				3	2		3			2			3			2		1

Število tekmovalcev, ki so uporabljali posamezni programski jezik.

Nekateri uporabljajo po več različnih jezikov (pri različnih nalogah) in se štejejo delno k vsakemu jeziku. (V letu 2023 je en tekmovalec uporabljal python in javo, eden python in C, trije 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“.

---

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 novejše različice C++ (od vključno C++11 naprej) razlikujejo od C++98, je letos uporabljalo kar precej tekmovalcev, še posebej v tretji skupini (npr. ranged **`for`**, **`auto`** v novem pomenu, pri nekaterih tudi lambda-izrazi).

V besedilu nalog za 1. in 2. skupino objavljamo deklaracije tipov, spremenljivk, podprogramov ipd. v pascalu, C/C++, C#, pythonu in javi. Delež tekmovalcev, ki pravijo, da deklaracije razumejo, je letos podoben kot prejšnja leta (27/30 v prvi skupini in 10/11 v drugi). Edini predlog, v katerem jeziku bi še želeli imeti deklaracije, je bila tokrat lua. 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 (23/29 v prvi skupini, 9/11 v drugi, 9/9 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++, sta dva omenila python (oba iz druge skupine), po eden pa javo in zig. 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 skupini so tekmovalci v povprečju malo mlajši kot lani, v drugi in tretji pa približno enako stari. V prejšnjih letih je pogosto nastopilo na tekmovanju tudi nekaj osnovnošolcev, letos pa ni bilo nobenega.

	Št. tekmovalcev po letnikih				Povprečni
Skupina	1	2	3	4	letnik
prva	12	20	24	26	2,8
druga	1	3	9	8	3,1
tretja	1		7	9	3,4

Druga vprašanja

Podobno kot prejšnja leta je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!), je pa bilo letos malo več kot ponavadi takih tekmovalcev, ki so za tekmovanje izvedeli od prijateljev ali na naši spletni strani. 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 slabe tri četrtine); sledijo tisti, ki so se tega naučili v šoli pri pouku (takih je slaba polovica, kar je malo več kot lani), in tisti, ki so se naučili programirati (tudi) na krožkih in tečajih (teh je približno dve petini).

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni; njihov delež je še malo višji kot lani. Med ostalimi so mnenja precej razdeljena, najpogostejša kombinacija je „enako časa, manj nalog“, poleg nje pa še „enako nalog, manj časa“ in „enako nalog, več časa“ (tidve nasprotujoči si mnenji imata obe celo enako število glasov).

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb; tudi posebnih tehničnih težav letos ni bilo. Nekaj tekmovalcev je želelo, da bi lahko svoje odgovore v prvi in drugi skupini oddajali kot datoteke (namesto da jih pišejo ali lepijo v obrazec na spletni strani), eden pa je pogrešal označevanje sintakse z barvami v spletnem obrazcu za oddajo odgovorov (tekmovalcem sicer priporočamo, naj pišejo v kakšnem samostojnem urejevalniku in potem odgovore le skopirajo v obrazec za oddajo).

V preteklosti si je veliko tekmovalcev želelo tudi, da bi imeli v prvi in drugi skupini na računalnikih prevajalnike in podobna razvojna orodja. Razlog, zakaj smo se v teh dveh skupinah izogibali prevajalnikom, je bil predvsem ta, da hočemo s tem obdržati poudarek tekmovanja na snovanju algoritmov, ne pa toliko na lovljenju drobnih napak; in radi bi tekmovalce tudi spodbudili k temu, da se lotijo vseh nalog, ne pa da se zakopljejo v eno ali dve najlažji in potem večino časa porabijo za testiranje in odpravljanje napak v svojih rešitvah pri tistih dveh nalogah. Toda v letih 2020 in 2021, ko so zaradi epidemije vsi reševali naloge doma in torej dostop do prevajalnikov in razvojnih orodij imeli, se je pokazalo, da te težave vendarle niso

	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	29	2	5	3	23	18	7	2	2	2	6	18	2	3	21
II	10	1	2	0	9	4	3	3	1	1	0	10	0	4	7
III	6	1	1	3	6	2	3	2	2	1	0	7	0	0	8

nastopile; tekmovalci so se večinoma lotili vseh nalog in rezultati v prvi skupini so bili še boljši kot ponavadi. Zato smo lani in letos, ko je tekmovanje spet potekalo v živo namesto prek interneta, omogočili tekmovalcem prve in druge skupine tudi uporabo prevajalnikov. To je bilo v anketi večinoma lepo sprejeto, je pa bilo tudi nekaj pripomb, ker razpoložljiva orodja nekaterim niso ustrezala.

Letos za razliko od prejšnjih let ni bilo pripomb, naj se tudi v prvi in drugi skupini uvede avtomatsko ocenjevanje, podobno kot v tretji.

CVETKE

V tem razdelku je zbranih nekaj zabavnih odlomkov iz rešitev, ki so jih napisali tekmovalci. V oklepajih pred vsakim odlomkom sta skupina in številka naloge.

(1.1) Nekdo je z navdušenjem implementiral funkcijo, za katero naloga pravi, da je že podana:

```
// it's real!!!
function Random(n) {
  return Math.floor(Math.random() * n)
}
```

(1.1) Za ljubitelje opojnih substanc:

```
char stavek[420];
```

(1.1) Tale optimist na vsakem koraku naključno izbere po eno črko prvotne besede in očitno upa, da ne bo nikoli po večkrat izbral iste črke:

```
String besedaObrnjena = "";
for (int i = 0; i < beseda.length(); i++)
  besedaObrnjena += beseda.charAt((int) (Math.random() * beseda.length()));
```

(1.1) Spet nekdo, ki spremenljivke imenuje „pomnilniki“:

> 1) rezerviram string pomnilnika, v katerega shranim besedilo od uporabnika
> ⋮
> 6) rezerviram še en pomnilnik, kamor bom prestavljal črke iz prejšnjega pomnilnika, da ne bom izgubil določenih črk v besedi

Podobne primere smo pred leti že imeli (2013, 2015, 2016); vsi so z iste šole in imajo istega mentorja.

(1.1) Odličen prispevek za kategorijo “no, but nice try”:

```
for j in range(len(l[i])):
  if j not in ["a-z"] or j not in ["A-Z"]:
```

Tu je v bistvu več cvetk: `j` je indeks znaka, v drugi vrstici pa ga uporablja, kot da bi bil to znak na tistem indeksu; uporablja zapis `"a-z"`, kot da bi bili to nekakšni regularni izrazi; in v pogoju ima **or** namesto **and**, tako da bi bil izpolnjen pri vsakem znaku (pri mali črki je izpolnjen desni del, pri veliki levi del pogoja, pri drugih znakih pa celo oba).

(1.1) Rešitev za ljubitelje represivnih organov:

```
# če naletite na te znake, bo FBI trkal na vaša vrata
ilegalni_simboli = [",", "!", ".", "?"]
```

Isti tekmovalec pri peti nalogi:

```
# v primeru da ne bi v slovarju našel te vrstice vrne None in mi lahko javimo
# pasivno-agresivno napako
⋮
print("huh? that's impossible. očitno nek podatek manjka :_(. FIX THAT "
                                "RIGHT NOW OR ELSE FBI WILL RAID YOUR HOUSE.")
```

Če je tole zanj pasivno agresivno, si ga ne bi želeli videti resnično agresivnega.

(1.1) Dekadentno: namesto da bi stikal nize z operatorjem +, je uporabil mehanizem za formatiranje nizov (formatted string literals).

```
# prosim, ne me spraševati
# očitno python ne podpira „item assignmenta“ na stringih
if r > k:
  beseda = f"{beseda[:k]}{zamenjanaCrka}{beseda[k+1:r]}{crka}{beseda[r+1:]}"
```

(1.1) Zanimiva sintaktična inovacija. Kako v enem zamahu primerjati znak z več drugimi znaki:

```
if (c[j].equals(',' / '?' / '!' / '.')) {
```

(1.1) Če se je komu zdelo, da v angleščini še ni dovolj težav z besedami na *-ough-*, je tale prispeval še eno:

```
for (int i = 0; i < word, i++) // gets a single word withought end signs
```

(1.1) Tale tekmovalec poskuša zamenjati dva znaka v besedi in pri tem čisto preveč pričakuje od metode replace:

```
char q = beseda.charAt(h);
char o = beseda.charAt(k);
beseda.replace(q, o);
beseda.replace(o, q);
```

(To, da replace ne spremeni niza, na katerem jo pokličemo, ampak vrne rezultat kot nov niz, je pri vsem skupaj še najmanjši problem.)

(1.1) Tale je vse, kar naloga zahteva, prevalil na funkcijo Random:

```
string n;
cout << "Vpiši poved in jo pretvori v zmešani citat:";
getline(cin, n);
Random(n);
cout << "Zmešani citat: " << n << endl;
```

V komentarju pod rešitvijo pa je še na dolgo in široko nakladal:

> Ideja v kodi je, da string n drži vrednost citata. Vrednost citata pa vpišemo s pomočjo getline, ki uporabnika vpraša za vrednost. Ta se potem premeša s pomočjo funkcije Random(n). To bi bila preprosta koda. Če bi želeli kodo poboljšati in narediti bolj uporabno, bi za shranjevanje podatkov uporabili nize (arrays), in sicer tako:
>
> ```
> arrays[n]: {};
> ```
>
> To je enodimenzionalni niz, poznamo tudi multidimenzionalne.

Isti tekmovalec pri četrti nalogi:

> Tekmovalec na oviratlonu teče. Pred sabo ima ovire, menim pa, da ga lahko nadziramo s tipkovnico in pomočjo **if** funkcije.

Če mu računalništvo še ne gre preveč dobro, ima pa vsaj veliko možnosti, da uspe kot politik.

(1.2) Presenetljivo veliko tekmovalcev po nepotrebnem komplicira z deljenjem ali odštevanjem, namesto da bi preprosto primerjali dve števili. Dva primera:

```
while ((stevilo / 9999 != 0) || (i < 6)) { // dokler je število pet- ali več mestno
                                           // ali dokler ne dosežemo največje predpone
while (ikd − 10000 > 0) // dokler je število 5+ mestno
```

Če bi prvi od teh tekmovalcev napisal „stevilo >= 9999“, bi verjetno lažje opazil, da je pomotoma razglasil 9999 za petmestno število. Podobno bi drugi, če bi napisal „ikd > 10000“, lažje opazil, da je pomotoma razglasil 10000 za manj kot petmestno število.

Naslednji tekmovalec je še zmogel napisati „x >= 1024“, pri „x >= 10000“ pa ga je zvilo in je moral uporabiti deljenje:

```
if x >= 1024 and (x / 10000) > 1:
```

Še nekaj prispevkov na to temo:

```
if a − int(a) > 0: # preveri, če ima a decimalke
if ((x − y) > 0.0) { // preverimo, če je karkoli za decimalko
if (int(st / 1024) > 0 && st > 9999) {
```

Pri zadnjem je kompliciranje še toliko bolj odveč, ker je prvi del pogoja (st > 1024) izpolnjen vedno, ko je izpolnjen drugi del (st > 9999).

(1.2) Zanimiv, a površen pristop:

```
dolzina = len(str(bajti))
# Za B
if dolzina <= 4:
   return str(bajti) + " B"
# Za KB
if dolzina > 5 and dolzina < 8:
   return str(math.ceil(bajti / 1024)) + " KB"
```

in tako naprej. Takšen pristop, ki temelji le na dolžini števila (v bajtih) v desetiškem zapisu, ne more delovati, saj je treba na primer osemmestna števila do vključno $9999 \cdot 2^{10} = 10\,238\,976$ zapisati v kilobajtih, večja osemmestna števila pa v megabajtih. Podobno je delalo še nekaj drugih tekmovalcev.

(1.2) Naslednji tekmovalec očitno zna deliti, vseeno pa se je pri pretvorbi iz bajtov v kilobajte odločil za odštevanje v zanki:

```
while (true) {
  // KB
  if ((B / 10240) > 1) { B = B − 10240; K = K + 10; }
  else if ((B / 1024 ) > 1) { B = B − 1024; K = K + 1; }
  else break; // ko ni več bajtov, se program zaključi
```

Podobno je kasneje tudi pretvarjal kilobajte v megabajte in tako naprej.

(1.2) Tale je namesto zanke po predponah uporabil rekurziven podprogram in se na koncu celo pohvalil s tem, da se je izognil zanki:

```
# tukaj sem uporabil "self called function", da se izognem loopa
```

(1.2) Tale tekmovalec je vhodni podatek za začetek pomnožil s tri — za vsak primer, če je bil morda negativen:

```
velikost += 2 * velikost # iz negativnega celega št. spremenimo v naravno št.
```

(1.2) Rešitev za ljubitelje matematike (in tudi deluje skoraj pravilno):

```
def human_readable(a: int) -> str:
  # izhajam iz enačbe log10(a / (2**(10 * n))) < 4
  # ta enačba pove, da mora biti dolžina števila največ štiri mesta
  # iz tega lahko izračunamo n > log2(a / 10**4) / 10
  size = math.ceil(log2(a / 10000) / 10)
  # zatem število zdelimo, zaokrožimo navzgor in dodamo ustrezno predpono
  return f"{math.ceil(ln / (2**(10 * size)))} {predpone[size]}B"
```

(1.2) Predrzno: tale rešitev, namesto da bi sama ugotovila, v kateri enoti je treba izpisati dano vrednost, vpraša o tem kar uporabnika.

```
cout << "Pritisni 1, če želiš pretvoriti v KB, 2 za MB, 3 za GB, 4 za TB "
        "in 5 za PB.";
cin >> odlocitev;
```

(1.2) Za ljubitelje zelo velikih celoštevilskih konstant:

```
else if (input <= 11257873168519397376) { // 11257873168519397376 deljeno s 1024^5
                                          // je 9999 (največje celo 4-mestno število)
```

Od tega pogoja je imel sicer več škode kot koristi — njegov učinek je bil ta, da če se danega števila niti v petabajtih ni dalo izpisati s štirimi števkami ali manj, potem ni izpisal sploh ničesar.

(1.3) Ko svetloba zarjove...

```
if (max_sv > svetlost) { break; } // the loop will break when light level
                                  // goes bellow the max recorded level
```

(1.3) Rešitev za ljudi, ki ne marajo operatorja „!=“:

```
while ((PreveriSvetlost() == num) == False):
  PritisniTipko()
```

(1.3) Nekdo se, kot kaže, ni mogel odločiti, ali bi napisal „predpostavimo“ ali „privzamemo“:

```
// Tipko pritisnemo tolikokrat, kot smo jo prvič, da smo prišli do najvišjega nivoja.
// Predpovzamemo, da je najnižji nivo, na katerega smo nastavili v zgornji vrstici, 1.
```

(1.3) Zanka, od katere je bolj malo haska:

```
int zanka = 0;
int pritiskiTipke = 0;
while (zanka > 0) {
```

(1.3) Tale tekmovalec si je poleg treh funkcij, ki so bile pri tej nalogi podane, zamislil še četrto, ki mu pomaga prekiniti neskončno zanko:

```
while (true) { // to se ugasne, ko se ugasne lučka
  if (Luc() = false) // funkcija, ki je nisem jaz naredil, ampak je dodana, ki samo ustavi
                     // loop, ampak je postavljena na vrh zaradi hitrejšega odziva
    break;
```

(1.4) Ta naloga je zahtevala, da tekmovalec tudi oceni časovno zahtevnost svojega postopka. Nekateri odgovori so bili precej . . . tavtološki:

> Njegova časovna zahtevnost je dokaj kompleksna, saj če ima veliko ovir in so le te zelo dolge potrebuje nekaj časa da jih s procesira in po potreboval malenkost več časa.

Podobno, a še bolj globokoumno napisano:

> Manj kot bi bilo ovir, hitreje bi izračunal dolžino poti in manj bi bilo operacij. To je logično, saj bi za vsako oviro morali izračunati, kateri del je daljši (ko bi se tekmovalec že zaletel vanjo), na kateri višini leži in kako dolga je (od katere do katere x-koordinate leži) in ali se pokriva s trenutno pozicijo/koordinato x tekmovalca. Posledično, če bi bilo več ovir, bi bilo tudi več opreacij, posledično večja časovna zahtevnost.

(1.4) Zanimiv pristop k ocenjevanju časovne zahtevnosti:

> za časovno kompleksnost bi rekel da je solidna

To je bilo v rešitvi, ki je pri premikanju tekača vedno povečevala y-koordinato po 1, namesto da bi ga v enem koraku premaknila do naslednje ovire; tako je bila vse prej kot učinkovita.

(1.4) Pesimistične ideje o časovni zahtevnosti:

> Za vsako dodano oviro bo program nekoliko počasnejši, saj znotraj vsakega klica funkcije premik preverjam vse podane ovire, kar pomeni, da bi se časovna zahtevnost programa eksponentno večala.

Pred tem je opisal postopek, ki se premika v korakih po 1 in po vsakem pregleda, če je zdaj pred njim ovira. Neučinkovito bo res, zaradi premikanja po 1, od števila ovir pa ima le linearno zahtevnost, ne eksponentne.

(1.4) Naslednji pesimist pa je časovne zahtevnosti kar množil, kjer bi jih moral seštevati:

> Časovna kompleksnost je n na kvadrat za sortiranje in n za preverjanje, če se zaletimo v ovire. Skupaj je n na 3.

(1.4) Rešitev z nezaupanjem do javascripta:

> Časovna kompleksnost:
> Moja koda je $O(n)$, ampak najprej sortira ovire, kar je v najslabšem primeru (slaba implementacija Javascripta, nesrečni vrstni red ovir) $O(n^2)$.

(1.4) Pomemben preboj na področju urejanja. Spodnja rešitev nam zagotavlja, da obstaja veliko algoritmov za urejanje v sublinearnem času:

> Časovna zahtevnost je linearno odvisna z številom ovir in to 1:1, dvakrat več ovir = dvakrat več časa. Za razvrstitev po oddaljenosti pa je veliko algoritmov, ki to lahko naredijo v časovni zahtevnosti manj kot 1:1, zato sem to zanemaril.

(1.4) Neroden način, kako opisati zanko, ki gre po ovirah od 1 do n:

> Naredim zanko, ki se bo ponavljala, dokler število 1 ne bo enako n.

(1.5) Samokritični komentarji v eni od rešitev:

```
/* tale program je preklet
:
v moji obrambi sm biu ultra mega turbo lačen
pač dolžan sm se opravičt za tale izgovor za programa */
```

(1.5) Spodnji komentar lahko morda vzamemo kot kompliment, da varnostni ukrepi na tekmovalnih računalnikih le niso tako slabi:

```
// ta del kode bi deloval samo na node.js, ki ga tukaj ni
// (interneta in root dostopa tudi ne :( )
/*
const fs = require('fs');
tekst = fs.readFileSync('vhod.txt').toString();
*/
```

(1.5) Iz rešitve z veliko komentarji vprašljive relevantnosti:

```
če to berete, pomeni, da sem imel preveč časa in sem se dolgočasil :)
```

(2.1) Tale tekmovalec v zapisu $O(\cdot)$ dosledno uporablja ničlo namesto črke O:

```
// Časovna zahtevnost 0(n^2)
:
for (int i = 1; i <=n; i++) // ponavlja za vsako osebo (0(n))
```

(2.2) Prispevek na temo "technically correct, the best kind of correct": rešitev, ki potencam števila 3 pravi „večkratniki".

```
scanf("%d", &n); // prebere število večkratnikov
int veck[100]; // array n-tih večkratnikov
veck[1] = 0; // nastavi prvi večkratnik
for (int i = 3; i <=n; i++) // naredi array večkratnikov števila 3
   veck[i] = veck[i − 1] * 3;
```

(2.3) Naloga: „v pomnilniku ni nujno dovolj prostora za hranjenje celotnega besedila". Tekmovalci:

```
char besede[1000000][100];
⋮
while (fgets("%s", besede[i]) != '') { // Zapomni si vse besede, dokler so nove besede
   i++;
}
```

(2.3) Nekdo si je dajal s konci vrstic več opravka, kot bi bilo nujno:

```
char c = getchar();
if (c == '\r') // hmmm *!#=&/! mikrosoft
   continue;
if (c == '\n')
   c = ' ';
```

(2.5) Rešitev za ljubitelje prekomerno zapletenih podatkovnih struktur in obskurnih kratic v imenih spremenljivk:

> Seznam skupnih trenutnih dogodkov (sstd) naj bo asociativna podatkovna struktura dogodkov s ključem i, ima naj tudi $O(1)$ poizvedbo o dolžini.
>
> Seznam seznamov skupnih trenutnih dogodkov (ssstd) naj bo linked list.
>
> Asociativni seznam dogodkov asd naj ima kot ključ i dogodka in kot vrednost kazalce na sstd, ki so znotraj ssstd.

(3.1) Rešitev za ljubitelje pretiravanja s funkcijskim programiranjem:

```
var sums = new BufferedReader(new InputStreamReader(System.in)).lines()
           .skip(1)
           .map(in -> in.split(" "))
           .map(i -> new int[] { parseInt(i[0]), parseInt(i[1]) })
           .reduce((i1, i2) -> new int[] { i1[0] + i2[0], i1[1] + i2[1] })
           .orElseThrow();
```

(3.1) Priznanje za najbolj obfuscirana imena spremenljivk dobi:

```
for (int nny = 0; nny <= cny; ++nny) {
   long long sny = csny[nny];
   long long bsp = syy_b + sny;
   long long nyn = min(cyn, bsp);
   long long syn = csyn[nyn];
   long long asp = syy_a + syn;
   if (asp < nny) continue;
   long long nnn = min(cnn, min(asp - nny, bsp - nyn));
   long long resc = nyn + nnn;
   if (resc > res) res = resc;
}
```

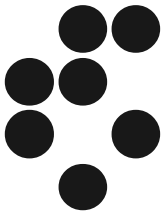
(3.4) Ekscentrično: zakaj bi rekli „**int** main“, če lahko uporabimo sinonim:

```
signed main() {
```

Bojda nekateri ljudje na tekmovanjih počnejo to zato, da imajo lahko na začetku programa makro „**#define int long long**“, česar pa ta program ni imel.

SODELUJOČE INŠTITUCIJE

Institut Jožef Stefan



Institut je največji javni raziskovalni zavod v Sloveniji s skoraj 800 zaposlenimi, od katerih ima približno polovica doktorat znanosti. Več kot 150 naših doktorjev je habilitiranih na slovenskih univerzah in sodeluje v visokošolskem izobraževalnem procesu. V zadnjih desetih letih je na Institutu opravilo svoja magistrska in doktorska dela več kot 550 raziskovalcev. Institut sodeluje tudi s srednjimi šolami, za katere organizira delovno prakso in jih vključuje v aktivno raziskovalno delo. Glavna raziskovalna področja Instituta so fizika, kemija, molekularna biologija in biotehnologija, informacijske tehnologije, reaktorstvo in energetika ter okolje.

Poslanstvo Instituta je v ustvarjanju, širjenju in prenosu znanja na področju naravoslovnih in tehniških znanosti za blagostanje slovenske družbe in človeštva nasploh. Institut zagotavlja vrhunsko izobrazbo kadrom ter raziskave in razvoj tehnologij na najvišji mednarodni ravni.

Institut namenja veliko pozornost mednarodnemu sodelovanju. Sodeluje z mnogimi uglednimi institucijami po svetu, organizira mednarodne konference, sodeluje na mednarodnih razstavah. Poleg tega pa po najboljših močeh skrbi za mednarodno izmenjavo strokovnjakov. Mnogi raziskovalni dosežki so bili deležni mednarodnih priznanj, veliko sodelavcev IJS pa je mednarodno priznanih znanstvenikov.

Tekmovanje sta podprla naslednja odseka IJS:

CT3 — Center za prenos znanja na področju informacijskih tehnologij

Center za prenos znanja na področju informacijskih tehnologij izvaja izobraževalne, promocijske in infrastrukturne dejavnosti, ki povezujejo raziskovalce in uporabnike njihovih rezultatov. Z uspešnim vključevanjem v evropske raziskovalne projekte se Center širi tudi na raziskovalne in razvojne aktivnosti, predvsem s področja upravljanja z znanjem v tradicionalnih, mrežnih ter virtualnih organizacijah. Center je partner v več EU projektih.

Center razvija in pripravlja skrbno načrtovane izobraževalne dogodke kot so seminarji, delavnice, konference in poletne šole za strokovnjake s področij inteligentne analize podatkov, rudarjenja s podatki, upravljanja z znanjem, mrežnih organizacij, ekologije, medicine, avtomatizacije proizvodnje, poslovnega odločanja in še kaj. Vsi dogodki so namenjeni prenosu osnovnih, dodatnih in vrhunskih specialističnih znanj v podjetja ter raziskovalne in izobraževalne organizacije. V ta namen smo postavili vrsto izobraževalnih portalov, ki ponujajo že za več kot 500 ur posnetih izobraževalnih seminarjev z različnih področij.

Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi virtualno učečo se skupnost in pripomoči k učinkovitejšemu povezovanju znanosti in industrije ter večji prepoznavnosti domačega znanja v slovenskem, evropskem in širšem okolju.

E3 — Laboratorij za umetno inteligenco
Področje dela Laboratorija za umetno inteligenco so informacijske tehnologije s poudarkom na tehnologijah umetne inteligence. Najpomembnejša področja raziskav in razvoja so: (*a*) analiza podatkov s poudarkom na tekstovnih, spletnih, večpredstavnih in dinamičnih podatkih, (*b*) tehnike za analizo velikih količin podatkov v realnem času, (*c*) vizualizacija kompleksnih podatkov, (*d*) semantične tehnologije, (*e*) jezikovne tehnologije.

Laboratorij za umetno inteligenco posveča posebno pozornost promociji znanosti, posebej med mladimi, kjer v sodelovanju s Centrom za prenos znanja na področju informacijskih tehnologij (CT3) razvija izobraževalni portal VideoLectures.NET in vrsto let organizira tekmovanja ACM v znanju računalništva.

Laboratorij tesno sodeluje s Stanford University, University College London, Mednarodno podiplomsko šolo Jožefa Stefana ter podjetji Quintelligence, Cycorp Europe, LifeNetLive, Modro Oko in Envigence.

*

Fakulteta za matematiko in fiziko
Fakulteta za matematiko in fiziko je članica Univerze v Ljubljani. Sestavljata jo Oddelek za matematiko in Oddelek za fiziko. Izvaja dodiplomske univerzitetne študijske programe matematike, računalništva in informatike ter fizike na različnih smereh od pedagoških do raziskovalnih.

Prav tako izvaja tudi podiplomski specialistični, magistrski in doktorski študij matematike, fizike, mehanike, meteorologije in jedrske tehnike.

Poleg rednega pedagoškega in raziskovalnega dela na fakulteti poteka še vrsta obštudijskih dejavnosti v sodelovanju z različnimi institucijami od Društva matematikov, fizikov in astronomov do Inštituta za matematiko, fiziko in mehaniko ter Instituta Jožef Stefan. Med njimi so tudi tekmovanja iz programiranja, kot sta Programerski izziv in Univerzitetni programerski maraton.

Fakulteta za računalništvo in informatiko
Glavna dejavnost Fakultete za računalništvo in informatiko Univerze v Ljubljani je vzgoja računalniških strokovnjakov različnih profilov. Oblike izobraževanja se razlikujejo med seboj po obsegu, zahtevnosti, načinu izvajanja in številu udeležencev. Poleg rednega izobraževanja skrbi fakulteta še za dopolnilno izobraževanje računalniških strokovnjakov, kot tudi strokovnjakov drugih strok, ki potrebujejo znanje informatike. Prav posebna in zelo osebna pa je vzgoja mladih raziskovalcev, ki se med podiplomskim študijem pod mentorstvom univerzitetnih profesorjev uvajajo v raziskovalno in znanstveno delo.

Fakulteta za elektrotehniko, računalništvo in informatiko

Fakulteta za elektrotehniko, računalništvo in informatiko (FERI) je znanstveno-izobraževalna institucija z izraženim regionalnim, nacionalnim in mednarodnim pomenom. Regionalnost se odraža v tesni povezanosti z industrijo v mestu Maribor in okolici, kjer se zaposluje pretežni del diplomantov dodiplomskih in podiplomskih študijskih programov. Nacionalnega pomena so predvsem inštituti kot sestavni deli FERI ter centri znanja, ki opravljajo prenos temeljnih in aplikativnih znanj v celoten prostor Republike Slovenije. Mednarodni pomen izkazuje fakulteta z vpetostjo v mednarodne raziskovalne tokove s številnimi mednarodnimi projekti, izmenjavo študentov in profesorjev, objavami v uglednih znanstvenih revijah, nastopih na mednarodnih konferencah in organizacijo le-teh.

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Fakulteta za matematiko, naravoslovje in informacijske tehnologije Univerze na Primorskem (UP FAMNIT) je prvo generacijo študentov vpisala v študijskem letu 2007/08, pod okriljem UP PEF pa so se že v študijskem letu 2006/07 izvajali podiplomski študijski programi Matematične znanosti in Računalništvo in informatika (magistrska in doktorska programa).

Z ustanovitvijo UP FAMNIT je v letu 2006 je Univerza na Primorskem pridobila svoje naravoslovno uravnoteženje. Sodobne tehnologije v naravoslovju predstavljajo na začetku tretjega tisočletja poseben izziv, saj morajo izpolniti interese hitrega razvoja družbe, kakor tudi skrb za kakovostno ohranjanje naravnega in družbenega ravnovesja. V tem matematična znanja, področje informacijske tehnologije in druga naravoslovna znanja predstavljajo ključ do odgovora pri vprašanjih modeliranja družbeno ekonomskih procesov, njihove logike in zakonitosti racionalnega razmišljanja.

ACM Slovenija

ACM je največje računalniško združenje na svetu s preko 80 000 člani. ACM organizira vplivna srečanja in konference, objavlja izvirne publikacije in vizije razvoja računalništva in informatike.

ACM Slovenija smo ustanovili leta 2001 kot slovensko podružnico ACM. Naš namen je vzdigniti slovensko računalništvo in informatiko korak naprej v bodočnost.

Društvo se ukvarja z:

- Sodelovanjem pri izdaji mednarodno priznane revije Informatica — za doktorande je še posebej zanimiva možnost objaviti 2 strani poročila iz doktorata.
- Urejanjem slovensko-angleškega slovarčka — slovarček je narejen po vzoru Wikipedije, torej lahko vsi vanj vpisujemo svoje predloge za nove termine, glavni uredniki pa pregledujejo korektnost vpisov.
- ACM predavanja sodelujejo s Solomonovimi seminarji.
- Sodelovanjem pri organizaciji študentskih in dijaških tekmovanj iz računalništva.

ACM Slovenija vsako leto oktobra izvede konferenco Informacijska družba in na njej skupščino ACM Slovenija, kjer volimo predstavnike.

IEEE Slovenija

Inštitut inženirjev elektrotehnike in elektronike, znan tudi pod angleško kratico IEEE (Institute of Electrical and Electronics Engineers) je svetovno združenje inženirjev omenjenih strok, ki promovira inženirstvo, ustvarjanje, razvoj, integracijo in pridobivanje znanja na področju elektronskih in informacijskih tehnologij ter znanosti.

REPUBLIKA SLOVENIJA
MINISTRSTVO ZA VZGOJO IN IZOBRAŽEVANJE

Ministrstvo za vzgojo in izobraževanje

Skupaj z našimi deležniki soustvarjamo vključujočo, enakopravno, trajnostno naravnano in ustvarjalno družbo vseživljenjskega učenja ter krepimo participacijo mladih in spodbujamo športna udejstvovanja. Zavzemamo se za enakopravno udeležbo vseh deležnikov v dostopnem in prožnem vzgojno-izobraževalnem sistemu, usmerjenem v prihodnost. Iščemo nove poučevalne ter učne strategije in tako posamezniku omogočamo zorenje v odgovornega in samostojnega člana skupnosti.

Zavod
Republike
Slovenije
za šolstvo

Zavod Republike Slovenije za šolstvo

Zavod Republike Slovenije za šolstvo je osrednji nacionalni razvojno-raziskovalni in svetovalni zavod na področju predšolske vzgoje, osnovnega šolstva in splošnega srednješolskega izobraževanja.

SREBRNA POKROVITELJA

Quintelligence
Obstoječi informacijski sistemi podpirajo predvsem procesni in organizacijski nivo pretoka podatkov in informacij. Biti lastnik informacij in podatkov pa ne pomeni imeti in obvladati znanja in s tem zagotavljati konkurenčne prednosti. Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja. IKT (informacijsko-komunikacijska tehnologija) je postavila temelje za nemoten pretok in hranjenje podatkov in informacij. S primernimi metodami je potrebno na osnovi teh informacij izpeljati ustrezne analize in odločitve. Nivo upravljanja in delovanja se tako seli iz informacijske logistike na mnogo bolj kompleksen in predvsem nedeterminističen nivo razvoja in uporabe metodologij. Tako postajata razvoj in uporaba metod za podporo obvladovanja znanja (knowledge management, KM) vedno pomembnejši segment razvoja.

Podjetje Quintelligence je in bo usmerjeno predvsem v razvoj in izvedbo metod in sistemov za pridobivanje, analizo, hranjenje in prenos znanja. S kombiniranjem delnih — problemsko usmerjenih rešitev, gradimo kompleksen in fleksibilen sistem za podporo KM, ki bo predstavljal osnovo globalnega informacijskega centra znanja.

Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja.

Oasis
Oasis Network je inovativna tehnologija veriženja blokov (*blockchain*) s poudarkom na zasebnosti. Ta je zagotovljena s šifriranjem transakcij na obeh koncih, s hrambo podatkov v šifrirani obliki in z izvajanjem programov oz. pametnih pogodb v varnem izvajalnem okolju (npr. Intel SGX). Tovrstno okolje zagotavlja, da lastnik računalnika v porazdeljenem sistemu ne more videti podatkov v nešifrirani obliki, ki jih program obdeluje, četudi ima na voljo fizičen dostop do pomnilnika, diska in procesorja. Tajno izvajanje transakcij na sicer javni verigi blokov posledično omogoča transparentno, preverljivo in robustno delovanje pametnih pogodb, napisanih v običajnem programskem jeziku za Ethereum, brez razkritja občutljivih podatkov. Omenjena tehnologija se danes že uporablja za poganjanje borz brez posrednikov (DeFi), odgovorno trgovanje s podatki, potezne igre, tajne ankete in glasovanja. Fundacija Oasis je bila ustanovljena leta 2019 in podpira raziskave in razvoj porazdeljenih tehnologij na področju zasebnosti in šifriranja ter organizira mednarodne

strokovne delavnice in tekmovanja (hekatone). Prav tako podpira razvoj rešitev za končne uporabnike, ki tečejo na omrežju Oasis in dokazujejo, da je zasebnost možna tudi v praksi. Od leta 2022 fundacija aktivno podpira ACM RTK, saj med najstniki želi spodbujati programersko razmišljanje za reševanje problemov v računalništvu in zavedanje o zasebnosti.

BRONASTA POKROVITELJA