15. tekmovanje ACM v znanju računalništva

28. marca 2020

Bilten

Bilten 15. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2021

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Primož Gabrijelčič, Tomaž Hočevar, Boris Horvat, Klemen Kenda, Vid Kocijan, Samo Kralj, Mitja Lasič, Matjaž Leonardis, Mark Martinec, Polona Novak, Jure Slak, Boštjan Slivnik, Jasna Urbančič, Janez Brank.

Naklada: 170 izvodov
Bilten je brezplačen.

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

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

CIP — Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

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

TEKMOVANJE ACM v znanju računalništva (15 ; 2020 ; online)
15. tekmovanje ACM v znanju računalništva : 28. marca 2020 : bilten / [avtorji nalog Nino Bašić ... [et al.] ; uredil Janez Brank]. — Ljubljana : Institut Jožef Stefan, 2021

ISBN 978-961-264-214-3
COBISS.SI-ID 80583171

KAZALO

Struktura tekmovanja 5

Nasveti za 1. in 2. skupino 7
Naloge za 1. skupino 11
Naloge za 2. skupino 17
Navodila za 3. skupino 23
Naloge za 3. skupino 27
Naloge šolskega tekmovanja 33
Neuporabljene naloge iz leta 2018 37

Rešitve za 1. skupino 47
Rešitve za 2. skupino 55
Rešitve za 3. skupino 63
Rešitve šolskega tekmovanja 81
Rešitve neuporabljenih nalog 2018 91

Nasveti za ocenjevanje
in izvedbo šolskega tekmovanja 129

Rezultati 133
Nagrade 140
Šole in mentorji 141
Off-line naloga: Stiskanje nizov 143

Univerzitetni programerski maraton 147

Anketa 150
Rezultati ankete 155
Cvetke 163

Sodelujoče inštitucije 169
Pokrovitelji 174

STRUKTURA TEKMOVANJA

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

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

V lažjih dveh skupinah traja tekmovanje tri ure; tekmovalci lahko svoje rešitve napišejo na papir ali pa jih natipkajo na računalniku, nato pa njihove odgovore oceni temovalna komisija. Naloge v teh dveh skupinah večinoma zahtevajo, da tekmovalec opiše postopek ali pa napiše program ali podprogram, ki reši določen problem. Pri pisanju izvorne kode programov ali podprogramov načeloma ni posebnih omejitev glede tega, katere programske jezike smejo tekmovalci uporabljati. Ponavadi lahko tekmovalci v teh dveh skupinah pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom in velika večina se jih odloči za slednje; letos pa, ker je tekmovanje v celoti potekalo prek interneta, je bilo reševanje na papir možno le v izjemnih primerih (če je tekmovalec sam poskeniral odgovore in jih poslal po elektronski pošti, kar je eden tudi res naredil).

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 nek 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 23.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C#, java in python.

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

Na začetku smo tekmovalcem poslali tudi nekaj navodil in nasvetov (str. 7–9 za 1. in 2. skupino, str. 23–25 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 143–145.

Podobno kot v zadnjih nekaj letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 24. januarja 2020. To je imelo eno samo težavnostno skupino, naloge (ki jih je bilo pet) pa so pokrivale precej širok razpon težavnosti. Tekmovalci so pisali odgovore na papir in dobili enak list z nasveti in navodili kot na državnem tekmovanju v 1. in 2. skupini (str. 7–9). Odgovore tekmovalcev na posamezni šoli so ocenjevali mentorji z iste šole, za pomoč pa smo jim pripravili nekaj strani z nasveti in kriteriji za ocenjevanje (str. 129–132). 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 288 tekmovalcev z 28 šol (ena od njih je bila osnovna, ostale pa srednje).

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

NASVETI ZA 1. IN 2. SKUPINO

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

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

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

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

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

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

Pri ocenjevanju so vse naloge vredne enako število točk. Svoje odgovore dobro utemelji. Prizadevaj si predvsem, da bi bile tvoje rešitve pravilne, ob tem pa je zaželeno, da so tudi čim bolj učinkovite; take dobijo več točk kot manj učinkovite (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). Za manjše sintaktične napake se ne odbije veliko točk. Priporočljivo in zaželeno je, da so tvoje rešitve napisane pregledno in čitljivo.

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

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

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

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

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

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

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

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

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

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

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

```
#include <stdio.h>

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

Še isti trije primeri v pythonu:

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

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

# Branje standardnega vhoda po vrsticah:
import sys

i = d = 0
```

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

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

Še isti trije primeri v javi:

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

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

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

NALOGE ZA PRVO SKUPINO

Naloge rešuj samostojno; ne sprašuj drugih ljudi za nasvete ali pomoč pri reševanju (niti v živo niti prek interneta ali kako drugače), ne kopiraj v svoje odgovore tuje izvorne kode in podobno. Tekmovalna komisija si pridržuje pravico, da tekmovalce diskvalificira, če bi se kasneje izkazalo, da nalog niso reševali sami. Internet lahko uporabljaš, če ni v nasprotju s prejšnjimi omejitvami (npr. za branje dokumentacije), vendar za reševanje nalog ni nujno potreben. Tvoje odgovore bomo pregledali in ocenili ročno, zato manjše napake v sintaksi ali pri klicih funkcij standardne knjižnice niso tako pomembne, kot bi bile na tekmovanjih z avtomatskim ocenjevanjem.

Tekmovanje bo potekalo na strežniku `https://rtk.fri.uni-lj.si/`, kjer dobiš naloge in oddajaš svoje odgovore. Uporabniška imena in gesla (bo)ste dobili po elektronski pošti. Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na svojem lokalnem računalniku.

Med reševanjem lahko vprašanja za tekmovalno komisijo postavljaš prek zasebnih sporočil na tekmovalnem strežniku (ikona oblačka zgoraj desno), izjemoma pa tudi po elektronski pošti na `rtk-info@ijs.si`.

Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave, lahko izjemoma pošlješ svoje odgovore po elektronski pošti na `rtk-info@ijs.si`, vendar nas morajo doseči pred koncem tekmovanja; odgovorov, prejetih po koncu tekmovanja, ne bomo upoštevali.

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

1. Vesoljske vsote

Preučevalec vesoljske matematike, Tim, je na eni izmed svojih odprav v vesolje odkril skrivnosten tuj zapis vsot. Niz „`*---**-*-----*`“ predstavlja vesoljski zapis vsote $1 + 4 + 4 + 5 + 10$. Po podrobnem pregledu nekaj zapisov je ugotovil, da za zapisovanje vsot v vesolju obstajajo naslednja pravila:

- Na začetku postavimo trenutno število na 1.
- Znak „`*`“ doda trenutno število v zapis vsote (na konec).
- Znak „`-`“ poveča trenutno število za 1.

Tim ni najbolj spreten pri programiranju, zato te prosi, da mu **napišeš program** (ali podprogram oz. funkcijo), ki vesoljski zapis vsote pretvori v človeku berljiv račun. Na standardni vhod bo tvoj program dobil niz znakov „`*`“ in „`-`“, ki naj ga pretvori v človeku berljiv račun (na koncu naj doda tudi enačaj in končno vrednost vsote) in izpiše na standardni izhod (ali pa v datoteko `vsota.txt`, karkoli ti je lažje). Predpostaviš lahko, da vsebuje vhodni niz vsaj eno zvezdico „`*`“. Zgornji niz naj tvoj program izpiše kot „`1 + 4 + 4 + 5 + 10 = 24`“.

Tim je pripravil tudi dva primera, da mu boš lažje pomagal.

Primer 1:

Vhod: `*---**-*-----*`
Izhod: `1 + 4 + 4 + 5 + 10 = 24`

Naslednja tabela kaže vrednost trenutnega števila po vsakem prebranem znaku:

Znak		*	-	-	-	*	*	-	*	-	-	-	-	-	*
Število	1	1	2	3	4	4	4	5	5	6	7	8	9	10	10

Primer 2:

Vhod: `-----*-*`
Izhod: `6 + 7 = 13`

Naslednja tabela kaže vrednost trenutnega števila po vsakem prebranem znaku:

Znak		-	-	-	-	-	*	-	*
Število	1	2	3	4	5	6	6	7	7

2. Ključ

Ključ za običajno cilindrično ključavnico ima šest zarez, globina vsake se mora ujemati z dolžino istoležnega zatiča v ključavnici, da se ključavnica lahko odklene. Zareze niso poljubno globoke, ampak proizvajalec predpisuje določene pogoje, ki jih morajo globine zarez izpolnjevati, da lahko nek tip ključavnice deluje pravilno in zanesljivo ter da se lahko ključ vanjo vstavi in izvleče brez zatikanja. Možnih je deset različnih globin, označenih s številko med 0 in 9. Pogoji so naslednji:

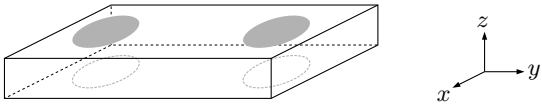
1. dovoljene so le globine med 1 in 8;
2. globini sosednjih dveh zarez se lahko razlikujeta za največ 5 (globina 2 na primer ne sme biti sosednja globini 8; to pravilo zagotavlja, da se ključ ne more zatakniti pri vstavljanju ali odstranjevanju);
3. dve sosednji zarezi ne smeta imeti enake globine;
4. nobena globina zareze se ne sme pojaviti več kot dvakrat.

Napiši podprogram (funkcijo), ki bo kot argument prejel celoštevilčno kodo ključa (na primer: 597969). Desetiške števke tega števila predstavljajo šest globin zarez. Lahko je prvih nekaj števk enakih 0 (na primer: število 123 predstavlja kodo 000123). Za vsako od naštetih štirih pravil naj podprogram izpiše, ali je pravilo izpolnjeno ali ne.

Primer: 597969 ustreza drugemu in tretjemu pogoju, ne pa prvemu (ker v njem niso samo števke od 1 do 8) in četrtemu (ker se števka 9 pojavlja trikrat).

3. Obračanje jogija

Imamo jogi v obliki kvadra. Odvisno od tega, kako ga obrnemo, se lahko naša glava znajde na enem od štirih možnih mest, kot kaže leva slika spodaj:



Sivo pobarvani elipsi kažeta dva možna položaja glave na eni strani jogija, črtkani elipsi pa še dva možna položaja glave na drugi strani jogija.

Spati želimo tako, da imamo glavo na najmanj obrabljenem delu, zato smo pripravljeni jogi občasno obrniti — zavrtimo ga za 180 stopinj okrog ene od osi x, y ali z (desna slika zgoraj kaže, kam je usmerjena katera os), tako da bo potem naša glava na kakšnem od ostalih treh možnih mest na jogiju.

Napiši funkcijo ObrniJogi(n), ki jo bo uporabnik poklical, ko bo pripravljen obrniti jogi, ona pa mu mora primerno svetovati, po kateri osi naj obrne jogi za 180 stopinj, da bo imel glavo na najmanj obrabljenem delu (torej na tistem, na katerem je doslej spal najmanj dni). Funkcija naj vrne enega od znakov `'x'`, `'y'`, `'z'`; če pa je bolje, da uporabnik trenutno jogija sploh ne obrača, naj tvoja funkcija vrne znak `'n'`. Uporabnik potem obrne jogi v skladu z rezultatom, ki ga vrne tvoja funkcija, in vse odtlej do naslednjega klica vsak dan spi na njem v njegovem sedanjem položaju (jogija torej nikoli ne obrača na lastno pest).

Kot parameter n bo tvoja funkcija dobila število dni od zadnjega klica (toliko dni je uporabnik spal na dosedanjem mestu).

Predpostavi, da uporabnik tvojo funkcijo prvič pokliče z $n = 0$ in da dotlej na jogiju ni še nikoli spal. Poleg funkcije ObrniJogi lahko deklariraš tudi poljubne globalne spremenljivke (oz. spremenljivke zunaj svoje funkcije) in jih po želji inicializiraš.

4. Zobna ščetka

Električno zobno ščetko poganja elektromotor, kot uporabniški vmesnik pa služi tipka; ta je lahko pritisnjena ali spuščena. Delovanje motorja upravlja preprost računalnik, ki lahko odčitava trenutno stanje tipke in lahko vklaplja ali izklaplja motor. Izklopljeno ščetko spravimo v pogon s pritiskom na tipko. Trajanje pritiska na tipko ne vpliva na delovanje, važen je le trenutek začetka pritiska tipke. Da uporabnik ne pretirava s čiščenjem zob, se mora ščetka samodejno izklopiti po 120 sekundah od zadnjega vklopa, lahko pa jo uporabnik izklopi že pred iztekom tega časa s (ponovnim) pritiskom na tipko.

Napiši program, ki bo upravljal z motorjem zobne ščetke tako, kot je zgoraj predpisano. Na razpolago so naslednje funkcije:

- za odčitavanje stanja tipke:

  Tipka() — funkcija vrne **true**, če je tipka pritisnjena, sicer **false**. (Funkcija ne čaka na pritisk tipke, ampak se vrne takoj in sporoči trenutno stanje tipke. Če uporabnik dlje časa drži tipko pritisnjeno, funkcija v tem času ob vsakem klicu vrne **true**.)

- za vklop ali izklop motorja:

  Motor(vklop) — če ima parameter vklop vrednost **true**, bo funkcija vklopila motor, če ima vrednost **false**, pa ga bo izklopila.

- štoparica, ki meri čas v sekundah:

  PozeniUro() — postavi čas na 0 in požene štoparico;
  UstaviUro() — ustavi štoparico;
  OdcitajUro() — vrne čas štoparice v sekundah kot celo število (tipa **int** oz. integer).

(Kdor piše v pythonu, naj si namesto **true** in **false** misli True in False.)

5. Plonkanje

Zaradi pojava virusa so morali organizatorji nekega tekmovanja iz znanja izvesti leto prek interneta, kjer pa tekmovalna komisija ni mogla nadzirati, če so si tekmovalci med seboj kaj pomagali („plonkali“).

Po natančni analizi vseh oddanih nalog več sto tekmovalcev je tekmovalni komisiji uspelo (seveda s pomočjo računalnika) natančno rekonstruirati, kdo je plonkal od koga.

Podatki o plonkanju so (zaradi varstva osebnih podatkov) anonimizirani in predstavljeni v tabeli z dvema stolpcema, kjer prva številka pomeni številko tekmovalca, ki je bil *pomočnik*, druga pa številko *prepisovalca*, torej tekmovalca, ki je prepisoval (plonkal). Tisti, ki je plonkal, je seveda lahko bil ob neki drugi priložnosti pomočnik in je pomagal novemu prepisovalcu in tako naprej. Pomočnik je lahko pomagal več prepisovalcem (v prvem stolpcu bodo lahko tudi enake številke), medtem ko je prepisovalec vedno lahko plonkal samo od enega pomočnika (v drugem stolpcu bodo same različne številke). Primer:

(*Nadaljevanje na naslednji strani.*)

pomočnik	prepisovalec (plonkar)
15	18
41	62
15	29
47	50
29	47
15	41
33	21
91	55
41	37
21	12
12	72
12	33

Iz zgornje tabele ugotovimo na primer, da je tekmovalec 41 prepisoval od tekmovalca 15 in da je tekmovalec 41 pomagal tekmovalcu 62 in tekmovalcu 37. Tekmovalec 15 pa je pomagal tudi tekmovalcu 29 in tekmovalcu 18.

Opiši postopek, ki bo iz tako podanih podatkov ugotovil:

(a) kateri so bili tisti tekmovalci, ki niso plonkali od nikogar, ampak so bili izključno pomočniki — imenujmo jih *izvirni tekmovalci*;

(b) kateri so bili *plonkarji prvega reda*, torej tekmovalci, ki so plonkali od izvirnih tekmovalcev;

(c) kateri so bili *plonkarji drugega reda*, torej tekmovalci, ki so plonkali od nekoga, ki je sam plonkal od izvirnega tekmovalca.

V zgornjem primeru sta izvirna tekmovalca 15 in 91, plonkarji prvega reda so 18, 41, 29 in 55, plonkarji drugega reda pa so 37, 62 in 47.

NALOGE ZA DRUGO SKUPINO

Naloge rešuj samostojno; ne sprašuj drugih ljudi za nasvete ali pomoč pri reševanju (niti v živo niti prek interneta ali kako drugače), ne kopiraj v svoje odgovore tuje izvorne kode in podobno. Tekmovalna komisija si pridržuje pravico, da tekmovalce diskvalificira, če bi se kasneje izkazalo, da nalog niso reševali sami. Internet lahko uporabljaš, če ni v nasprotju s prejšnjimi omejitvami (npr. za branje dokumentacije), vendar za reševanje nalog ni nujno potreben. Tvoje odgovore bomo pregledali in ocenili ročno, zato manjše napake v sintaksi ali pri klicih funkcij standardne knjižnice niso tako pomembne, kot bi bile na tekmovanjih z avtomatskim ocenjevanjem.

Tekmovanje bo potekalo na strežniku `https://rtk.fri.uni-lj.si/`, kjer dobiš naloge in oddajaš svoje odgovore. Uporabniška imena in gesla (bo)ste dobili po elektronski pošti. Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na svojem lokalnem računalniku.

Med reševanjem lahko vprašanja za tekmovalno komisijo postavljaš prek zasebnih sporočil na tekmovalnem strežniku (ikona oblačka zgoraj desno), izjemoma pa tudi po elektronski pošti na `rtk-info@ijs.si`.

Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave, lahko izjemoma pošlješ svoje odgovore po elektronski pošti na `rtk-info@ijs.si`, vendar nas morajo doseči pred koncem tekmovanja; odgovorov, prejetih po koncu tekmovanja, ne bomo upoštevali.

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

1. Metanje na koš

Profesionalni košarkarji se morajo med treningom vaditi tudi v metanju prostih metov na koš, saj so prosti meti pomemben del košarkaških tekem in je važno, da so pri tem čim bolj zanesljivi.

Organiziramo torej tekmovanje v metanju prostih metov na koš, kjer se tekmovalci preizkušajo, kdo ima najboljše živce in bo največkrat vrgel na koš, ne da bi enkrat samkrat zgrešil koš.

Pravila so takšna: Vsak tekmovalec bo žogo na koš vrgel velikokrat, recimo tisočkrat (seveda lahko vmes tudi počiva, spije vodo, ali kaj prigrizne `;-)`). Zmagovalec tekmovanja bo tisti, ki bo naredil najdaljši neprekinjeni niz zadetkov v koš.

Če pa bo imelo več tekmovalcev enak najdaljši neprekinjeni niz zadetkov, potem bo zmagovalec tisti od njih, ki ima najdaljši niz zadetkov, v katerem je en sam met mimo koša.

In če bo tudi pri tem pogoju še vedno več tekmovalcev imelo enako dolg niz zadetkov z enim zgrešenim metom, potem bo zmagovalec tisti, ki bo imel najdaljši niz zadetkov z dvema zgrešenima metoma. In tako dalje, s tremi, štirimi, petimi zgrešenimi meti v nizu, dokler ne bo število zgrešenih metov v zaporednem nizu enako številu vseh zgrešenih metov tekmovalca. Če bo tudi takrat več tekmovalcev imelo enak rezultat (pri tisoč metih je sicer zelo malo verjetno, da bi do tega prišlo), bo zmagovalca pač določila tekmovalna komisija z žrebom.

Podatki za enega tekmovalca so predstavljeni z nizom enic in ničel (enica za zadetek v koš, ničla za met mimo koša), na primer takole (35 metov, 26 zadetkov, 9 mimo koša):

```
11101111001101111101111001011111110
```

Opiši postopek ali napiši podprogram (funkcijo), ki za dani niz in celo število k izračuna dolžino najdaljšega takega niza zadetkov, med katerimi je največ k metov mimo koša. (Če tega ne znaš rešiti v splošnem, reši nalogo vsaj za primere, ko je $k = 0$ ali $k = 1$, in boš dobil delne točke.)

Primer: za gornji niz 35 metov bi pri $k = 0$ dobili rezultat 7, pri $k = 1$ rezultat 9 (zaporedje petih zadetkov, nato en met mimo koša in nato še zaporedje štirih zadetkov), pri $k = 3$ pa rezultat 12.

2. Ne odlašaj na jutri, kar lahko storiš pojutrišnjem

Učenci v šoli imajo n obveznosti, ki jih morajo izpolniti. Pri tem jim i-ta obveznost vzame d_i dni in jo morajo opraviti najkasneje na dan k_i (mislimo si, da so dnevi oštevilčeni z naravnimi števili od začetka šolskega leta). V posameznem dnevu se lahko ukvarjajo samo z eno obveznostjo, lahko pa počivajo in se ne ukvarjajo z nobeno. Ko se enkrat lotijo neke obveznosti, jo morajo potem opraviti v neprekinjenem sklopu d_i zaporednih dni (torej jim ta obveznost vzame dan, ko so začeli z njo, in še naslednjih $d_i - 1$ dni). Če je na primer $d_i = 3$, to pomeni, da se morajo začeti z i-to obveznostjo ukvarjati najkasneje na dan $k_i - 2$.

Kot tipični učenci si seveda želijo vse te obveznosti opraviti čim kasneje. Torej, če lahko nek dan počivajo in se obveznosti lotijo jutri in še zmeraj opravijo vse obveznosti pravočasno, potem danes seveda raje počivajo.

Natančneje povedano, dva možna razporeda tega, kdaj opravijo kakšno obveznost, primerjamo takole: poiščemo najzgodnejši dan, na katerega pri enem razporedu počivajo, pri drugem pa delajo; če takega dne ni, štejemo razporeda za enakovredna in sta nam oba enako dobra; sicer pa nam je boljši tisti razpored, pri katerem na tisti dan počivajo.

Napiši program ali podprogram (funkcijo), ki kot vhodne podatke dobi podatke o obveznostih (števila $n, k_1, d_1, k_2, d_2, \ldots, k_n, d_n$) in izračuna najboljši možni razpored (za vsako obveznost i naj torej ugotovi, na kateri dan z_i se morajo začeti ukvarjati z njo), ali pa ugotovi, da za te vhodne podatke sploh ni nobenega veljavnega razporeda. Če je možnih več enako dobrih razporedov, je vseeno, katerega poiščeš. Podrobnosti glede oblike vhodnih in izhodnih podatkov si izberi sam in jih tudi opiši. Tvoja rešitev naj bo čim učinkovitejša, tako da bo uporabna tudi za večje vhodne primere (recimo, da gre lahko n do 10^6, datum k_i pa do 10^9).

3. Lenoba

V službi želimo preživeti čim manj časa, ne da bi to kdorkoli opazil. Za vse sodelavce natanko vemo čas prihoda in odhoda (vsakdo pride in odide zgolj enkrat v dnevu; vsi podatki so znotraj enega dneva, nihče ne ostane v službi čez polnoč). **Opiši postopek**, ki izračuna, kdaj moramo priti v službo in koliko časa moramo tam preživeti, da nobena oseba ne bo prisotna takrat, ko pridemo, in še vedno prisotna takrat, ko odidemo. Edini dodatni pogoj je, da želimo priti v službo pred dvanajsto in oditi po dvanajsti (ker je točno ob dvanajstih kosilo).

Kot vhodne podatke tvoj postopek dobi število sodelavcev in za vsakega sodelavca čas njegovega prihoda in odhoda. Vsi časi se merijo v nanosekundah od polnoči, tako da so to sicer nenegativna cela števila, vendar so lahko precej velika. (Ena sekunda ima 1 000 000 000 nanosekund.) Sodelavec, ki pride ali odide ob istem času kot mi, nas vidi priti ali oditi — če se hočemo temu izogniti, moramo priti vsaj eno nanosekundo pred njim ali oditi vsaj eno nanosekundo za njim. Podrobnosti glede predstavitve vhodnih podatkov si izberi sam in jih v svoji rešitvi tudi opiši.

4. Semafor

Semafor na prehodu za pešce ima dvomestni prikazovalnik sekund do spremembe luči. Prikazuje lahko torej poljubno celo število od 0 do 99, lahko pa je tudi ugasnjen. **Napiši podprogram** oz. funkcijo VsakoSekundo(n), ki jo bo sistem poklical enkrat na sekundo, da mu bo pomagala upravljati s prikazovalnikom na semaforju. Prek parametra n ti sistem pove, kaj se dogaja z lučjo semaforja: če se je v zadnji sekundi stanje luči spremenilo (iz zelene v rdečo ali obratno), ti parameter n pove, koliko sekund bo luč semaforja v svojem novem stanju; če pa se v zadnji sekundi stanje luči ni spremenilo, boš dobil $n = -1$.

Na voljo imaš funkcijo Prikazi(n), ki jo lahko pokličeš, da na prikazovalniku prikažeš število n. Število n mora biti od 0 do 99, lahko pa je -1, če hočeš, da naj bo prikazovalnik prazen (ne prikazuje nobenega števila, niti ničle).

Deklariraš lahko tudi globalne spremenljivke in jih po svoji želji inicializiraš. Predpostavi, da bo vrednost parametra n pri prvem klicu funkcije VsakoSekundo gotovo večja od 0.

Če čas, ko bi se morala luč spremeniti, mine, sistem pa te še ni obvestil o spremembi stanja luči (in trajanju novega stanja), moraš poskrbeti, da bo prikazovalnik prazen, dokler te sistem ne obvesti o spremembi stanja luči.

Prikazovalnik na semaforju je le dvomesten, stanje luči pa včasih traja več kot 99 sekund. Zato naj tvoj podprogram poskrbi, da če je do spremembe stanja luči več kot 99 sekund, naj se prikazuje število 99, vendar naj utripa (eno sekundo gori, eno sekundo je ugasnjeno).

5. Prelom besedila

Dano imamo besedilo, dolgo z znakov, ki ga želimo prikazati v okencu, ki ima prostora za w znakov na vrstico. Ko besedilo pišemo v okence, lahko vrstico prelomimo le na presledku pred začetkom nove besede. Presledki ostanejo na prejšnji vrstici in lahko gledajo preko roba okna, nova vrstica pa se začne s prvim naslednjim znakom, ki ni presledek.

Za vsako širino w od 1 do z izračunaj, koliko vrstic bo besedilo imelo, če ga želimo napisati v okence širine w. Besedilo bo vsebovalo le črke, števke, ločila in presledke. Lahko predpostaviš, da drugih znakov za prazen prostor, kot na primer tabulatorjev ali znakov za novo vrstico, ne bo. Besedilo se tudi ne bo začelo s presledkom.

Napiši podprogram (funkcijo), ki sprejme besedilo kot niz znakov dolžine z in vrne ali izpiše seznam z števil, kjer števila po vrsti predstavljajo, koliko vrstic bo besedilo imelo, če ga želimo napisati v okence širine $1, 2, 3, 4, \ldots, z$. Če besedila v okence neke širine ni mogoče spraviti, ne da kakšna beseda gledala prek meja, na tisto mesto napiši -1.

Primer. Recimo, da dobimo takšen vhodni niz (spodaj je napisan v dveh vrsticah, vendar si moramo obe skupaj predstavljati kot en sam niz, brez kakšnih vmesnih znakov za konec vrstice ali česa podobnega; presledki so predstavljeni s simbolom ␣, da se jih bolje vidi):

```
Na␣začetku␣je␣bilo␣ustvarjeno␣vesolje.␣␣To␣je␣povzročilo␣
mnogo␣hude␣krvi␣in␣na␣splošno␣velja␣za␣zelo␣slabo␣potezo.
```

Pravilen izhodni seznam za ta niz je:

$-1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 12, 12, 10, 10, 8, 8, 8, 7, 6, 6,$
$6, 6, 6, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,$
$3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,$
$2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,$
$2, 2, 2, 2, 2, 2, 2, 2, 2, 1.$

Na primer, če je $w = 19$, spravimo besedilo v 6 vrstic (zato je 19. element v gornjem seznamu enak 6):

```
Na␣začetku␣je␣bilo␣
ustvarjeno␣vesolje.␣␣
To␣je␣povzročilo␣
mnogo␣hude␣krvi␣in␣
na␣splošno␣velja␣za␣
zelo␣slabo␣potezo.
```

(Primer očitno povzet po: Douglas Adams, *Restavracija ob koncu Vesolja*, *Štoparski vodnik po galaksiji*, prevod: Alojz Kodre.)

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

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

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

Tvoji programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali python, mi pa jih bomo preverili s prevajalniki FreePascal, GNUjevima `gcc` in `g++` 7.4.0 (ta verzija podpira C++17), prevajalnikom za javo iz JDK 8, s prevajalnikom Mono 4.6 za C# in z interpreterjema za python 2.7 in 3.6.

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2020-3/` 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/test-sistema/`. Uporabniško ime in geslo za Putko boš dobil po elektronski pošti. Med tekmovanjem lahko vprašanja za tekmovalno komisijo postavljaš prek foruma na Putki (povezava „Diskusija“ na dnu besedila posamezne naloge), izjemoma pa tudi po elektronski pošti na `rtk-info@ijs.si`.

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

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

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

Ocenjevanje

Vsaka naloga ti lahko prinese od 0 do 100 točk. Vsak oddani program se preizkusi na več testnih primerih; pri vsakem od njih dobi vse točke, če je izpisal pravilen

odgovor, sicer pa 0 točk (izjema je 1. naloga, kjer je možno tudi delno točkovanje). Pri prvi in tretji nalogi je testnih primerov po 20 in vsak je vreden po 5 točk, pri četrti in peti nalogi je testnih primerov po 10 in vsak je vreden po 10 točk, pri drugi nalogi pa je testnih primerov 6, število točk za posamezni primer pa je navedeno v besedilu naloge.

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

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

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

Primer naloge (ne šteje k tekmovanju)

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

Primer vhoda:

```
123 456
```

Ustrezen izhod:

```
5790
```

Primeri rešitev:

- V pascalu:

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

- V C++:

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

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

- V C-ju:

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

- V pythonu:

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

- V javi:

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

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

- V C#:

```
using System;

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

NALOGE ZA TRETJO SKUPINO

1. Vaje v slogu

Avantgardni pisatelj Polde ne uporablja presledkov, ločil, velikih začetnic in podobnih stvari. Njegovi spisi zato niso nič drugega kot dolgi nizi samih malih črk. Iznašel je tudi novo retorično figuro, ki jo je z veliko skromnostjo poimenoval *Poldetova trojica*. Tvorijo jo tri pojavitve enakega podniza v dolgem nizu, ki predstavlja njegov spis, pri čemer se te tri pojavitve med seboj ne smejo prekrivati, smejo pa biti med njimi tudi kakšne črke, ki ne pripadajo nobeni od teh treh pojavitev.[1]

Poldetovo trojico lahko na kratko opišemo s četverico števil (i, j, k, d), pri čemer d pove dolžino uporabljenega podniza, števila i, j in k pa so indeksi prvega znaka vsake od njegovih treh pojavitev, ki tvorijo to trojico (znake niza, ki predstavlja celoten spis, si mislimo oštevilčene od 1 do n, pri čemer je n dolžina niza). Pojavitve naštejmo vedno od leve proti desni, tako da bo $i < j < k$.

Oglejmo si nekaj primerov. V nizu `ababaccabab` se med drugim pojavljajo naslednje Poldetove trojice:

Podniz	Pojavitve	Opis (i, j, k, d)
`ab`	**a b** a b a c c **a b** a b	$(1, 3, 8, 2)$
`ab`	**a b** a b a c c a b **a b**	$(1, 3, 10, 2)$
`ba`	a **b a** b a c c a **b a** b	$(2, 4, 9, 2)$
`a`	**a** b **a** b **a** c c a b a b	$(1, 3, 5, 1)$
`a`	a b **a** b **a** c c **a** b a b	$(3, 5, 8, 1)$
`a`	a b a b **a** c c **a** b **a** b	$(5, 8, 10, 1)$
`a`	a b **a** b a c c **a** b **a** b	$(3, 7, 10, 1)$
`a`	**a** b a b **a** c c a b **a** b	$(1, 5, 10, 1)$
`b`	a **b** a **b** a c c a **b** a b	$(2, 4, 9, 1)$

Poleg zgornjih je v istem nizu še več drugih Poldetovih trojic. V tem nizu se trikrat pojavlja tudi podniz `aba`, vendar pa njegove tri pojavitve ne tvorijo Poldetove trojice, ker se prvi dve pojavitvi tega podniza prekrivata.

Polde ima rad trojice, ki jih tvorijo čim daljši podnizi. **Napiši program**, ki mu jih bo pomagal odkrivati. Med vsemi Poldetovimi trojicami poišči tisto z največjo dolžino podniza d oz. preštej, koliko je trojic s tem d.

Vhodni podatki: v prvi vrstici je celo število n (veljalo bo $1 \leq n \leq 10^5$). V drugi vrstici je niz s; dolg je n znakov in vsi ti znaki so male črke angleške abecede.

Pri 70 % testnih primerov bo $n \leq 1000$.

Izhodni podatki: v prvo vrstico izpiši števila i, j, k in d, ločena s po enim presledkom. Zanje mora veljati $1 \leq i$, $i + d \leq j$, $j + d \leq k$, $k + d \leq n + 1$ in ta števila morajo predstavljati opis tiste Poldetove trojice v vhodnem nizu, ki ima največji d. Če obstaja več trojic s tem d, je vseeno, katero izpišeš. Pri naših testnih primerih bo vedno obstajala vsaj ena Poldetova trojica.

[1] Zanimivo (in malenkost težjo) različico naloge dobimo tudi, če v definicijo Poldetove trojice dodamo še naslednji pogoj: med tremi pojavitvami podniza, ki tvorijo Poldetovo trojico, sme biti kvečjemu g znakov, ki ne pripadajo tem trem pojavitvam. Pri tem je g neka vnaprej podana celoštevilska konstanta. Tako na primer za $g \leq 2$ tri pojavitve podniza `ab` v nizu `abcabdcab` ne tvorijo Poldetove trojice, saj so med njimi še trije drugi znaki (`c`, `d` in še en `c`).

V drugo vrstico izpiši eno samo celo število, namreč število Poldetovih trojic z največjim d. Pravzaprav, ker zna biti to število precej veliko, izpiši ostanek po deljenju tega števila z 1 000 037.

Točkovanje: če števila v prvi vrstici tvojega izpisa ustrezajo zahtevam naloge, število v drugi vrstici pa je napačno, dobiš pri tistem testnem primeru 60 % vseh možnih točk. Če sta pravilni obe vrstici, dobiš vse točke.

Primer vhoda:	Eden od možnih pripadajočih izhodov:
`11` `ababaccabab`	`2 4 9 2` `5`

Komentar: v tem vhodnem nizu je pet Poldetovih trojic s podnizi dolžine 2 (štiri trojice iz nizov `ab` in ena iz nizov `ba`).

2. Zamik

Podano je naravno število n. Tvoj program naj izpiše poljubno zaporedje n ničel in enic. Ocenjevalni računalnik ga prebere in na skrivaj krožno zamakne za k mest v desno, nato pa ga mogoče tudi prezrcali z desne na levo (mogoče pa ne). Tvoja naloga je ugotoviti, za koliko mest je ocenjevalni računalnik zamaknil tvoj niz in ali ga je nato tudi prezrcalil. Uporabiš lahko poizvedbe tipa „kakšen je po tem zamiku in morebitnem zrcaljenju i-ti znak niza?“ za različne i. Ugotovi pravi zamik (in to, ali je bil niz po zamiku tudi prezrcaljen ali ne) s čim manj poizvedbami.

V okviru vsakega testnega primera bo moral tvoj program rešiti več takih ugank (vendar ne več kot 100), lahko za različne n. To je interaktivna naloga — tvoj program se bo „pogovarjal“ z ocenjevalnim računalnikom tako, da bo bral s standardnega vhoda in pisal na standardni izhod. Ta pogovor naj poteka po naslednjih korakih:

1. Preberi vrstico, v kateri je celo število n za naslednjo uganko. Če je $n < 0$, je to znak, da je ugank konec in da naj se tvoj program neha izvajati. Sicer bo $n \geq 6$ in moraš nadaljevati z naslednjim korakom.

2. Izpiši vrstico, v kateri je niz n ničel in enic, ki bi ga rad uporabljal v nadaljevanju te uganke.

3. Nato lahko izvedeš 0 ali več (največ 20) poizvedb. Poizvedbo izvedeš tako, da izpišeš vrstico, v kateri je eno samo celo število i z območja $1 \leq i \leq n$, in nato prebereš odgovor ocenjevalnega računalnika — vrstico, v kateri je eno samo celo število, `0` ali `1`, ki ti pove, kakšen je i-ti znak v nizu po tistem, ko ga je sistem zamaknil in mogoče prezrcalil.

4. Ko ugotoviš, za koliko mest je sistem zamaknil tvoj niz v desno (recimo k) in ali ga je nato tudi prezrcalil ali ne, izpiši vrstico z dvema številoma, ločenima s presledkom: prvo število naj bo k, drugo pa naj bo 0, če sistem tvojega niza ni prezrcalil, oz. 1, če ga je prezrcalil.

5. Nato se vrni na korak 1.

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

Omejitve. Pri tej nalogi je šest testnih primerov, ki imajo različne omejitve in so vredni različno število točk:

(1) $n \leq 10$, sistem nikoli ne prezrcali niza (10 točk);

(2) $n \leq 10$, sistem lahko tudi prezrcali niz (10 točk);

(3) $n \leq 100$, sistem nikoli ne prezrcali niza (15 točk);

(4) $n \leq 100$, sistem lahko tudi prezrcali niz (15 točk);

(5) $n \leq 1024$, sistem nikoli ne prezrcali niza (25 točk);

(6) $n \leq 1024$, sistem lahko tudi prezrcali niz (25 točk).

Še enkrat poudarimo, da bo tvoj program v okviru vsakega od teh šestih testnih primerov moral rešiti po več ugank. Točke za tisti testni primer dobi le, če ustrezno reši vse uganke v njem.

Če tvoj program pri kakšni uganki izpiše napačen odgovor, postavi več kot 20 poizvedb ali pa se v kakšnem drugem pogledu ne drži zgoraj opisanega protokola, ga bo sistem takoj ustavil in pri tem testnem primeru ne bo dobil nobenih točk.

Sicer je število točk pri tistem testnem primeru odvisno od maksimalnega števila poizvedb, ki jih je tvoj program postavil (pri katerikoli od ugank v tem testnem primeru); recimo temu maksimumu m. Če je $m \leq 11$, dobiš pri tem testnem primeru vse točke, sicer pa $m - 11$ točk manj kot vse.

Primer:

Tvoj program izpiše	Sistem izpiše	Komentar
	`7`	rešiti bomo morali uganko za $n = 7$
`1010001`		sistem ta niz pri sebi zamakne in morda prezrcali
`1`	`1`	naš program deluje precej naivno:
`2`	`0`	sistematično pregleduje zamaknjeni niz
`3`	`1`	znak za znakom
`4`	`1`	
`5`	`0`	
`6`	`0`	
`7`	`0`	
`4 1`		pravilno smo ugotovili, da je sistem zamaknil naš niz za 4 znake desno in da ga je nato tudi prezrcalil
	`-1`	sistem pravi, da je testnega primera konec

V gornjem primeru je sistem najprej zamaknil naš niz `1010001` za štiri mesta v desno (nastalo je `0001101`) in ga nato še prezrcalil (in dobil `1011000`).

3. Zlaganje slik

V nekem muzeju moderne umetnosti želijo natisniti plakat, ki bo v pomanjšani obliki prikazoval vse slike v njihovi zbirki. Pripravili so torej zaporedje n pravokotnih slik, pri čemer je i-ta od njih široka w_i enot in visoka h_i enot.

Te slike bi zdaj radi zložili v vrstice; širina vrstice je definirana kot vsota širin slik v njej, višina vrstice pa kot maksimum njihovih višin. Zlagati jih moramo po vrsti (v takem vrstnem redu, v kakršnem so podane v vhodnem zaporedju), torej prvih nekaj slik v prvo vrstico, naslednjih nekaj v drugo vrstico in tako naprej. Da plakat ne bo preširok, ne sme biti nobena vrstica širša od s enot.

Ker so slike zelo abstraktne, so videti enako dobro tudi, če jih zavrtimo za 90 stopinj, zato se smemo pri vsaki sliki posebej odločiti, ali bi jo mogoče tako zavrteli ali ne (če sliko i zavrtimo za 90 stopinj, bo potem široka h_i enot namesto w_i, visoka pa bo w_i enot namesto h_i).

Znotraj teh omejitev si želimo, da bi bila vsota višin vseh vrstic čim manjša. **Napiši program**, ki izračuna najmanjšo možno vsoto višin vseh vrstic, ki jo je mogoče doseči na ta način.

Vhodni podatki: v prvi vrstici sta dve celi števili, n in s, ločeni s presledkom. Sledi n vrstic, od katerih i-ta vsebuje celi števili w_i in h_i, ločeni s presledkom. Veljalo bo $1 \leq n \leq 10^4$, $1 \leq w_i \leq 10^9$, $1 \leq h_i \leq 10^9$, $s \leq 10^{13}$. Poleg tega bo s zagotovo večji ali enak dolžini krajše stranice vsakega pravokotnika, tako da bo slike zagotovo mogoče razporediti v vrstice, široke največ s enot.

- Pri prvih 20 % testnih primerov bo $n \leq 20$.
- Pri naslednjih 30 % testnih primerov bo $n \leq 1000$.
- Pri preostalih 50 % testnih primerov bo $n \leq 10^4$.

V vsaki od zgornjih treh skupin bodo pri polovici testnih primerov vse slike kvadratne ($w_i = h_i$).

Izhodni podatki: izpiši najmanjšo možno vsoto višin vrstic, ki jo je mogoče doseči, če slike razporedimo v vrstice v skladu s pravili iz besedila naloge.

Primer vhoda:

```
6 24
14 4
8 3
6 11
4 13
11 9
3 14
```

Pripadajoči izhod:

```
18
```

4. Janko in Metka

Janko in Metka sta v hiši zlobne čarovnice našla n sladkarij. Vrednost i-te sladkarije sta ocenila s c_i. Čarovnici bosta ukradla k sladkarij. Težavo pa imata, ker se ne strinjata vedno, ali je posamezna sladkarija dobra ali ne. Sklenila sta kompromis, da mora biti v izbrani množici k sladkarij *vsaj* x dobrih sladkarij po mnenju Janka in *vsaj* x dobrih po mnenju Metke ($x \leq k$).

Napiši program, ki bo izračunal, kakšna je največja možna vsota vrednosti sladkarij, ki jih lahko odneseta Janko in Metka ob upoštevanju svojega kompromisa.

Vhodni podatki: v prvi vrstici so s presledkom ločena števila n, k in x. V drugi vrstici so podane vrednosti sladkarij $c_1, c_2, \ldots, c_n$, ki so prav tako ločene s presledki. Tretja vrstica opisuje sladkarije, ki so dobre po Jankovem mnenju, četrta pa po Metkinem mnenju. Seznama sladkarij se začneta s številom sladkarij v seznamu, ki

mu sledijo s presledki ločene številke sladkarij. Vsi vhodni podatki so pozitivna cela števila.

Omejitve: veljalo bo $1 \le k \le n \le 10^6$ in (za vsak i) $1 \le c_i \le 10^9$.

V prvih 20 % testnih primerov bo $n \le 20$. V naslednjih 40 % testnih primerov bo $n \le 10^4$.

Pozor, pazite na hitrost branja, ker ima naloga velike vhodne podatke!

Izhodni podatki: izpiši iskano vsoto vrednosti sladkarij, ki jih bosta odnesla Janko in Metka. Zagotovljeno je, da bo rešitev obstajala.

Primer vhoda:	Pripadajoči izhod:

`11 5 2`
`15 6 14 1 16 7 90 14 3 4 88`
`5 5 3 1 4 7`
`4 9 10 7 4`
	`213`

Komentar: v danem primeru se jima najbolj splača vzeti sladkarije 1, 5, 7, 10 in 11. Janku so všeč 1, 5 in 7, Metki pa 7 in 10.

5. Ključavničarstvo

Načrtujemo računalniško igrico, v kateri mora igralec hoditi po stavbi in obiskati vse njene sobe. Stavba obsega n sob in $n - 1$ hodnikov med njimi. Vsak hodnik neposredno povezuje natanko dve sobi. Prek teh hodnikov je vsaka soba (v enem ali več korakih) povezana z vsako drugo, vendar natanko na en način (z drugimi besedami, hodniki ne tvorijo ciklov).

Sobe so oštevilčene od 1 do n. Na začetku igre je v vsaki sobi nekaj ključev, vsak hodnik med dvema sobama pa je zaklenjen z 0, 1 ali več ključavnicami. Ko igralec prvič vstopi v neko sobo, pri tem avtomatsko pobere vse ključe v njej. Igralec začne svoj sprehod v sobi 1; pred tem ni imel ključev, vendar pa takoj na začetku pobere ključe, ki so bili v sobi 1.

Igralec lahko z vsakim ključem odpre katero koli ključavnico, ki nato ostane odklenjena, uporabljeni ključ pa se pri tem zlomi in ni več uporaben. Po hodniku lahko gre igralec iz ene sobe v drugo šele, če odklene vse ključavnice v njem (ni pa nujno, da odklene vse; dokler je v njem kakšna ključavnica zaklenjena, je hodnik neprehoden). Ko je hodnik odklenjen, se sme igralec po njem sprehoditi tudi po večkrat in to v obe smeri; tako se lahko torej tudi vrača v sobe, ki jih je prej že obiskal.

Želimo se izogniti scenariju, kjer igralec zaradi nespametnega zaporedja obiska sob in odklepanja ključavnic ne more končati igre, ker bi, še preden je obiskal vse sobe, porabil vse ključe in obstal pred zaklenjenim hodnikom. **Napiši program**, ki ugotovi, ali je tak scenarij mogoč.

Vhodni podatki: v okviru enega testnega primera bo moral tvoj program obdelati več stavb. V prvi vrstici je število stavb T. Sledijo opisi stavb, pred vsakim pa je prazna vrstica.

Vsaka stavba je opisana takole: v prvi vrstici je celo število n (število sob). Sledi vrstica z n celimi števili, $k_1, k_2, \ldots, k_n$, ločenimi s po enim presledkom; pri tem število k_i pove, koliko ključev je na začetku igre v i-ti sobi. Sledi še $n - 1$ vrstic, ki opisujejo hodnike; j-ta od teh vrstic vsebuje tri cela števila, u_j, v_j in ℓ_j (ločena s

po enim presledkom), ki povedo, da j-ti hodnik neposredno povezuje sobi u_j in v_j in da je ob začetku igre zaklenjen s ℓ_j ključavnicami.

Omejitve: veljalo bo $1 \leq n \leq 10^5$, $0 \leq k_i \leq 10^4$, $1 \leq u_j < v_j \leq n$, $0 \leq \ell_j \leq 10^4$ in $1 \leq T \leq 10$.

- Pri prvih 20 % testnih primerov bo $n \leq 10$.
- Pri naslednjih 20 % testnih primerov bo $n \leq 1000$, sobe bodo s hodniki povezane v eno samo dolgo zaporedje (seznam) brez razvejitev, skupno število ključev pa ne bo enako skupnemu številu ključavnic.
- Pri naslednjih 20 % testnih primerov bo $n \leq 1000$.
- Pri naslednjih 20 % testnih primerov bodo vsi $\ell_j = 1$ in skupno število ključev ne bo enako skupnemu številu ključavnic.
- Pri zadnjih 20 % testnih primerov ni posebnih dodatnih omejitev.

Izhodni podatki: po vrsti izpiši po eno vrstico za vsako stavbo; ta vrstica naj vsebuje le niz „`da`“ ali „`ne`“ (brez narekovajev), ki pove, ali je pri tisti stavbi mogoč tak scenarij, po kakršnem sprašuje naloga.

Primer vhoda:

```
2

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

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

Pripadajoči izhod:

```
da
ne
```

Komentar: v prvem primeru lahko igralec iz sobe 1 obišče sobo 2, nato pa odklene ena izmed vrat na hodniku $(1, 3)$ in ena izmed vrat na hodniku $(2, 4)$. Ostane brez ključev, ne da bi obiskal sobi 3 in 4. V drugem primeru bo igralec vedno lahko obiskal vse sobe.

Omenimo še, da je prvi tloris veljaven primer zaporedja (seznama) brez razvejitev, karkšni se pojavljajo v drugi izmed podnalog.

NALOGE ZA ŠOLSKO TEKMOVANJE

24. januarja 2020

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. PINi

Primož si ne uspe zapomniti varnostne kode (PIN) za svojo bančno kartico, zato si jo bo napisal kar na zadnjo stran kartice. Ker pa ima nekaj malega pojma o varovanju podatkov, je sklenil, da bo številko prikril s postopkom, ki ga je izumil sam.

Varnostna koda je štirimestno število z vrednostjo od 0000 do 9999. Zapišemo jo lahko tudi kot zaporedje štirih števk *abcd*, kjer lahko vsaka števka zavzame vrednosti od 0 do 9.

Namesto kode *abcd* bi rad Primož na bančno kartico zapisal štirimestno število *wxyz*, iz katerega bo izračunal originalno kodo *abcd* po naslednjem postopku:

$a = z$;
$b = w \oplus x$;
$c = x \oplus y$;
$d = y \oplus z$;

Računsko operacijo $\oplus$ je definiral takole: operacija sešteje dve števili, nato pa izračuna ostanek pri deljenju z 11. Če je rezultat 10, ga spremeni v 0.

Če bi imel Primož varnostno kodo 6910, bi si na kartico napisal 1846. Iz te številke lahko z nekaj enostavnimi operacijami dobi originalno varnostno kodo:

$wxyz = 1846$, torej $w = 1$, $x = 8$, $y = 4$, $z = 6$;

$a = z = 6$;
$b = w \oplus x = 1 \oplus 8 = 9$;
$c = x \oplus y = 8 \oplus 4 = 1$;
$d = y \oplus z = 4 \oplus 6 = 0$;

Koda je torej $abcd = 6910$.

Kmalu pa je ugotovil, da za svojo varnostno kodo ne more najti ustrezne začetne kombinacije. Malo je še poskušal in ugotovil, da obstaja kar nekaj takih štirimestnih kod, ki nikakor ne morejo biti rezultat njegovega postopka.

Namesto da bi si izmislil boljši postopek prikrivanja številke, je sklenil, da bo stopil do bankomata in spremenil varnostno kodo svoje kartice. Še pred tem pa bi rad ugotovil, katerih varnostnih kod ne sme uporabiti.

Napiši program, ki pregleda vse možne varnostne kode od 0000 do 9999 in izpiše vse, ki se jih ne da izračunati po zgoraj opisanem postopku. Tvoja rešitev naj bo čim učinkovitejša.

2. INTERCAL

V ezoteričnem programskem jeziku INTERCAL se vsak ukaz začne z „DO“, „PLEASE“ ali „PLEASE DO“ (v nadaljevanju ukaza se te fraze ne pojavljajo). Tistim, ki se začnejo s „PLEASE“ ali „PLEASE DO“, rečemo *vljudni*, ostalim (tistim, ki se začnejo z „DO“) pa *nevljudni*. To, na katero od teh treh fraz se ukaz začne, nič ne vpliva na njegov pomen ali delovanje, vendar pa prevajalnik kode ne bo prevedel, če se vljudni ukazi v njem pojavljajo preredko ali prepogosto (ker uporabnik ni vljuden ali pa je pretirano vljuden). Natančneje povedano, delež vljudnih ukazov (glede na vse ukaze) mora biti vsaj a in kvečjemu b. Pri tem sta a in b podana kot deleža, ne v odstotkih, torej velja $0 \leq a \leq b \leq 1$.

Napiši program ali podprogram (funkcijo), ki kot vhodne podatke dobi ali prebere obe meji (a in b) in zaporedje ukazov (ukazov je največ 10 000, vsak je v svoji vrstici, ta se začne na eno od zgornjih treh fraz in presledek in je dolga največ 100 znakov). Izpiše naj primerno popravljeno zaporedje ukazov, pri čemer sme spreminjati le začetno frazo („DO“, „PLEASE“ ali „PLEASE DO“) posameznega ukaza. V popravljenem zaporedju naj bo delež vljudnih ukazov vsaj a in kvečjemu b. Če se zaporedja ne da popraviti v skladu s temi omejitvami, naj tvoj program izpiše, da je problem nerešljiv.[2]

Ukaze lahko bereš s standardnega vhoda ali iz datoteke `vhod.txt` ali pa predpostaviš, da jih dobiš podane v neki tabeli, seznamu, vektorju ali čem podobnem.

Primer. Recimo, da imamo $a = 0{,}5$ in $b = 0{,}8$ ter da dobimo naslednje zaporedje ukazov:

```
DO ,1 <- #13
PLEASE DO ,1 SUB #1 <- #238
DO ,1 SUB #2 <- #108
DO ,1 SUB #3 <- #112
DO ,1 SUB #4 <- #0
```

Tu je vljuden le en ukaz od petih; delež vljudnih ukazov je torej 1/5, kar je manj od a. Primerno popravljeno zaporedje ukazov je na primer takšno (to ni edina možna rešitev):

```
DO ,1 <- #13
PLEASE DO ,1 SUB #1 <- #238
DO ,1 SUB #2 <- #108
PLEASE ,1 SUB #3 <- #112
PLEASE ,1 SUB #4 <- #0
```

Zdaj so vljudni trije ukazi od petih, torej je delež vljudnih ukazov 3/5, kar je res večje ali enako a ter manjše ali enako b.

3. Najdaljša pot

Na pravokotni plošči s karirasto mrežo imamo v vsakem polju ploščico z vrisano puščico, ki lahko kaže v eno od štirih smeri ($\leftarrow, \rightarrow, \uparrow, \downarrow$). Ploščice so naključno postavljene na vsa polja. Mreža ima največ 1000 stolpcev in 1000 vrstic.

[2] Zanimivo in rahlo težjo različico naloge dobimo, če dodamo še zahtevo, da hočemo spremeniti čim manj ukazov vhodnega zaporedja.

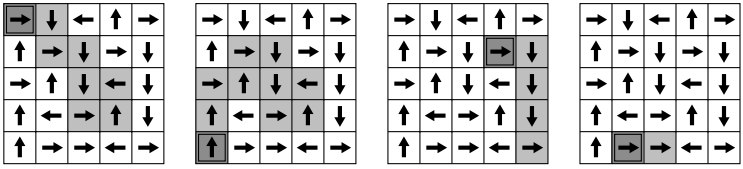
Postavimo se na izbrano začetno polje in se začnemo premikati. Iz polja, na katerem smo trenutno, se lahko premaknemo samo na sosednje polje, na katero kaže puščica našega polja. Na sosednje polje se ne moremo premakniti:

- če se nahajamo na polju, ki je na robu igralne plošče, in puščica našega polja kaže izven igralne plošče,
- če je v sosednjem polju puščica, ki kaže nazaj na naše polje,
- če smo v tem sosednjem polju že bili (križanja in zanke niso dovoljeni).

Opiši postopek (ali napiši program ali podprogram oz. funkcijo — karkoli ti je lažje), ki kot vhodni podatek dobi opis mreže in izračuna dolžino najdaljše poti, ki jo lahko prehodimo, če si smemo začetno polje izbrati poljubno.

Če ti je ta naloga pretežka, lahko za 14 točk od 20 rešiš lažjo različico, pri kateri si začetnega polja ne smemo izbrati poljubno, pač pa vedno začnemo v zgornjem levem polju.

Primer: spodnja slika kaže štiri poti na eni in isti mreži, vsakič z drugim začetnim poljem. Začetno polje je pobarvano temno sivo in ima dvojni rob, ostala polja na poti pa so svetlo siva.



Dolžina poti: 8 10 5 2

Izkaže se, da je pri tej mreži največja možna dolžina poti 10.

4. Domače naloge

Imamo seznam n domačih nalog, ki jih bomo po vrsti reševali d dni. Prvi dan bomo torej rešili prvih nekaj nalog, drugi dan naslednjih nekaj in tako naprej. Vsak dan lahko rešimo poljubno število nalog. Za vsako nalogo poznamo njeno težavnost t_i, ki je realno število med 0 in 10. Učinkovitost vaje za posamezen dan je enaka največji težavnosti naloge, ki smo jo rešili ta dan. **Opiši postopek**, ki ugotovi, kako naj seznam nalog razdelimo po dnevih, da bo vsota učinkovitosti po vseh dnevih največja možna. Kot vhodne podatke dobi tvoj postopek števila n, d, $t_1, \ldots, t_n$.

Primer: če imamo $n = 8$ nalog s težavnostmi $[6, 9, 5, 8, 3, 4, 5, 8]$ in bi jih radi razdelili na $d = 3$ dni, je največja možna vsota učinkovitosti enaka 25; dosežemo jo na primer z razdelitvijo $[6, 9], [5, 8, 3], [4, 5, 8]$ (ni pa to edina primerna razdelitev).

5. Razbijanje permutacijske šifre

Začasno smo dobili dostop do super tajne naprave za šifriranje, ki jo za komunikacijo uporabljata KGB in ameriški predsednik. Naprava je pravzaprav permutacijska šifra: kot vhod dobi niz n znakov in izpiše te znake v premešanem vrstnem redu, pri čemer jih vedno premeša na enak način. Ugotovili bi radi, za katero permutacijo gre (torej:

kateri vhodni znak pride na katero mesto na izhodu). Pri tem lahko v napravo spuščamo različne nize, dolge po n znakov (ki si jih sami izberemo), in opazujemo, kaj pride ven. Abeceda, iz katere so sestavljeni naši nizi, ima največ b različnih znakov — recimo, da so znaki predstavljeni kar s celimi števili od 0 do $b - 1$.

Opiši postopek, ki za podana n in b sestavi zaporedje vhodnih nizov za šifrirno napravo, nato pa vrnjene zašifrirane nize uporabi, da razbije šifro. Radi bi torej, da bi tvoj postopek na koncu vedel o delovanju naprave dovolj, da bi znal za poljuben niz n znakov napovedati, v kaj ga bo naprava zašifrirala. Za n in b predpostavi, da sta celi števili z območja od 1 do 1000.

NEUPORABLJENE NALOGE IZ LETA 2018

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 13. tekmovanjem ACM v znanju računalništva (leta 2018), 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. 91–128) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Križci in krožci

(To je rahlo težja različica naloge, ki smo jo na tekmovanju 2018 uporabili kot prvo v drugi skupini.) Mislimo si posplošeno različico igre „križci in krožci“. Namesto dveh igralcev jih je lahko več, pa tudi namesto le dveh vrst simbolov (križcev in krožcev) jih je lahko več, zato bomo simbole predstavili kar z naravnimi števili. Igralci začnejo z veliko karirasto mrežo (njene velikosti naš program ne pozna), ki je na začetku prazna, nato pa igralci vanjo vpisujejo števila. Število smejo vpisati le v celico, ki je bila dotlej prazna. Če z vpisom novega števila nastane na mreži skupina petih zaporednih enakih števil (v isti vrstici, stolpcu ali diagonali), dobi igralec, ki je tisto število vpisal, točko; lahko dobi tudi več točk hkrati, če z vpisom tistega števila nastane več takih skupin hkrati.

Napiši podprogram (funkcijo) `Poteza(x, y, n)`, ki ga bo sistem poklical, ko bo igralec vpisal naravno število n v celico (x, y). Koordinate celic so naravna števila. Tvoja funkcija mora vrniti število, ki pove, koliko točk dobi igralec s to potezo; če pa je poteza neveljavna (ker celica ni bila prazna), naj vrne -1. Za shranjevanje podatkov o stanju igre lahko uporabljaš globalne spremenljivke in jih pred prvim klicem funkcije `Poteza` tudi po želji inicializiraš.

2. Taborniki

Taborniki vsako leto priredijo tekmovanje v signalizaciji z Morsejevo abecedo. Črke in številke so v Morsejevi abecedi predstavljene s pomočjo pik in črtic, na primer:

$$\text{A} = \text{. -}, \quad \text{B} = \text{. - - -}, \quad 7 = \text{- - . . .}, \quad \text{Z} = \text{- - . .} \text{ ipd.}$$

Taborniki sporočila na daljši razdalji prenašajo ponoči s pomočjo svetlobnih bliskov. Piko oddajo s pomočjo kratkega bliska, črto pa s pomočjo dolgega. Zmaga tista ekipa, ki najhitreje in čimbolj pravilno odda vse znake.

Ker organizatorji ne želijo, da bi ekipe, ki tekmujejo prej, svojim vrstnikom, ki tekmujejo kasneje, sporočile zaporedja znakov na listku, so te prosili za pomoč. Želijo, da jim pripraviš program, ki bo za vsako ekipo pripravil drugačen listek z naključnim zaporedjem znakov. Pri tem pa morajo vse ekipe na listkih imeti enake znake (le v drugačnem vrstnem redu), saj se časi oddajanja različnih znakov med seboj razlikujejo, smisel tekmovanja pa je prav v primerljivosti rezultatov.

Napiši podprogram, ki bo izpisal listke za n ekip (n je med 2 in 100). Posamezen listek vsebuje 5 vrstic po 6 znakov. Listek naj vsebuje med 3 in 7 števk, ostali znaki morajo biti črke (velike črke angleške abecede).

Različica: tvoj podprogram naj pazi tudi na to, da so si vsi listki res različni in da noben listek ne vsebuje dveh ali več enakih vrstic.

Primer izpisa za $n = 2$:

```
EKIPA 1
2   J   O   T   8   P
A   C   X   R   O   D
W   J   6   Z   M   E
L   5   S   K   I   P
Q   M   O   U   C   N

EKIPA 2
8   O   L   Q   J   C
M   C   2   I   T   P
J   A   X   6   W   M
P   E   5   N   Z   S
O   K   D   O   U   R
```

3. Kratkovidnež

Imamo pravokotno mrežo različno visokih stolpcev, ki predstavljajo neko diskretno hribovje. Nekje na pobočju smo izgubili očala, ki so se odkotalila navzdol in pristala nekje v dolini. Sedaj vidimo samo še sosednje štiri celice in bi radi poiskali očala. Vemo, da so se očala odkotalila tako, da so na vsakem koraku padla s trenutnega polja na najnižje izmed sosednjih polj; če je takih več, pa so padla naključno. Kotaliti se nehajo, ko so vsa sosednja polja višja od trenutnega.

Za vsako od naslednjih podnalog **napiši podprogram**, ki jo reši, pri čemer kot vhodne podatke dobi dvodimenzionalno tabelo z višinami vseh stolpcev in koordinati točke, kjer smo izgubili očala.

(a) Najdi očala, če imaš zagotovljeno enoličnost padca.

(b) Kratkovidnež izvaja enako strategijo, kot so jo očala. Ali jih bo gotovo našel (ali pa je na poti kje neenolična možnost)?

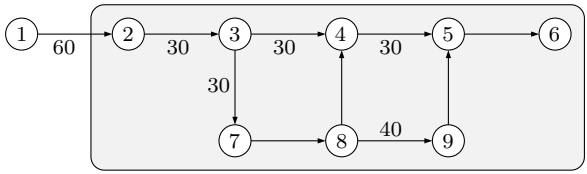
(c) Kratkovidnež ima pri roki še drobtinice, ki jih lahko pušča na poljih, da se zna vrniti nazaj po že prehojeni poti (in pri tem drobtinice spet pobira, da jih lahko kasneje uporabi kje drugje). Drobtinic ima dovolj za največ n korakov dolgo pot. Koliko polj izmed tistih, kamor bi očala potencialno lahko pristala, bo pogledal, ne da tvega, da se izgubi?

4. Omejitve hitrosti

Na otoku, na katerega vodi samo en most, je cestno omrežje sestavljeno iz samih enosmernih cest. Na mostu stoji prometni znak, ki določa *privzeto* omejitev hitrosti v mestu. (Tudi most je enosmeren, cesta na njem pelje na otok, z otoka pa ne pelje sploh nobena cesta.) Trenutna prometna ureditev določa, da mora po vsakem križišču stati znak za omejitev hitrosti, *razen* če je na tem delu ceste hitrost enaka za mesto privzeti omejitvi. V tem primeru znaka za omejitev hitrosti zaradi varčevanja ni.

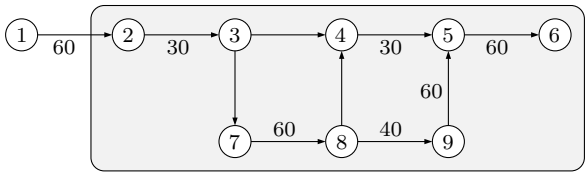
Na spodnji sliki je prikazan primer zelo majhnega cestnega omrežja takšnega tipa. Na cesti, ki vodi na most ($1 \to 2$), stoji znak za privzeto omejitev 60 km/h. Na odsekih od križišča 7 do križišča 8, od 8 do 4 in od 9 do 5 ni znaka za omejitev

hitrosti, zato je tam hitrost omejena na 60 km/h (privzeta hitrost). Vsi drugi cestni odseki so opremljeni z ustreznimi prometnimi znaki.



Mestni svet se je v svoji modrosti odločil, da je treba prometni sistem spremeniti, ker imajo turisti z njim preveč težav. Po novem bo na vsakem cestnem odseku stal znak za omejitev hitrosti, a le, če se hitrost na tem odseku razlikuje od hitrosti na cestah, ki prihajajo v križišče. Na vhodni cesti (na mostu) mora znak za omejitev hitrosti ostati, ker sicer turisti ne bi vedeli, kako hitro naj vozijo po mostu.

Spodnja slika prikazuje stanje po spremembi. Oglejmo si primer. Hitrost na odseku od 3 do 4 se ne razlikuje od hitrosti na odseku od 2 do 3, zato znaka za omejitev tam ne potrebujemo. Potrebujemo pa znak na odseku od 5 do 6, ker se hitrost razlikuje od tiste na odseku od 4 do 5.



Opiši postopek, ki prebere podatke o cestnem omrežju, zgradi interno predstavitev podatkov, ugotovi, kako naj bodo znaki postavljeni, in izpiše novo postavitev.

Vhodni podatki: v prvi datoteki sta števili n (število križišč) in m (število cestnih odsekov). Sledi m vrstic, v vsaki sta oznaka začetnega in končnega križišča in omejitev (če znak obstaja; sicer je tam število -1). Križišča so označena s celimi števili od 1 do n. Križišči, ki ju povezuje most, ki vodi na otok, imata številki 1 (začetek mostu, na celini) in 2 (konec mostu, na otoku).

Izhodni podatki: program naj izpiše novo stanje v enaki obliki.

Primer vhodnih podatkov za zgornje cestno omrežje:

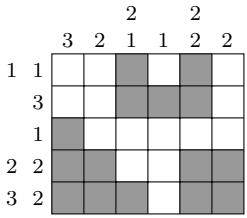
```
9 10
1 2 60
2 3 30
3 4 30
4 5 30
5 6 -1
3 7 30
7 8 -1
8 4 -1
8 9 40
9 5 -1
```

Pripadajoči izhodni podatki:

```
10
1 2 60
2 3 30
3 4 -1
4 5 30
5 6 60
3 7 -1
7 8 60
8 4 -1
8 9 40
9 5 60
```

5. Nonogram

Nonogram je vrsta uganke, pri kateri moramo v pravokotni karirasti mreži, ki je na začetku prazna (vsa polja so bela), pobarvati nekatera polja črno glede na številke, ki so na levi strani vsake vrstice in nad vsakim stolpcem. Število številk pred vrstico oziroma nad stolpcem nam pove, koliko strnjenih skupin črnih polj moramo imeti v tej vrstici oziroma stolpcu; vsaka številka pa pove, koliko črnih polj mora vsebovati vsaka skupina. Naslednja slika kaže primer pravilno izpolnjene mreže:



Napiši podprogram, ki kot vhodne podatke dobi že izpolnjeno mrežo in vse omejitve (številke levo od mreže in nad njo) ter preveri, ali je mreža pravilno izpolnjena (v skladu z omejitvami). Podrobnosti tega, v kakšnih podatkovnih strukturah bi tvoj podprogram te podatke dobil, definiraj sam in jih v svoji rešitvi tudi opiši.

6. Varovanje podatkov

V podjetju imamo v produkciji n datotečnih strežnikov. Datoteke iz teh strežnikov zaradi varnosti kopiramo na m rezervnih (*backup*) strežnikov, ki so razpršeni na zelo oddaljenih lokacijah. Število produkcijskih strežnikov je večje od števila rezervnih strežnikov.

Za kopiranje skrbi sistem za varnostno kopiranje. Njegova naloga je, da je vsaka nova različica vsake datoteke čim prej skopirana na enega od rezervnih strežnikov. Pravila za kopiranje so naslednja:

- Kopija iste verzije datoteke mora biti vedno v celoti kopirana samo na en strežnik.
- Sistem ne sme istočasno kopirati dveh datotek iz istega datotečnega strežnika, ker s tem zmanjšuje prenosno hitrost.
- Sistem ne sme istočasno kopirati dveh datotek na isti rezervni strežnik, ker s tem zmanjšuje prenosno hitrost.

Sistem za kopiranje ima za čim boljše delovanje na voljo naslednje podatke:

- Sprotni seznam vseh datotek za kopiranje na vseh datotečnih strežnikih.
- Dolžino vsake datoteke, ki jo je treba skopirati (MB).
- Sprotno informacijo o tem, kdaj je datoteka uspešno skopirana na rezervni strežnik.
- Povprečno hitrost kopiranja (MB/s) v zadnji minuti za vsakega od rezervnih strežnikov.

Opiši postopek, ki se bo na podlagi podatkov sproti odločal, katero datoteko iz katerega datotečnega strežnika bo kopiral na kateri rezervni strežnik.

7. Kotaljenje kocke

Dana je kocka, ki ima na vsaki ploskvi po eno črko. Razporeditev črk po kocki poznamo in tudi vemo, kako je na začetku obrnjena. Kotalimo jo po karirasti mreži (celice mreže so kvadrati enake velikosti kot ploskve kocke) po neki znani poti. Ko kocka pade na celico mreže, nanjo zapiše črko tiste ploskve, s katero se je je dotaknila. Pred začetkom kotaljenja je bila mreža prazna; če kocka isto celico obišče po večkrat, ostane na njej najkasneje vpisana črka. **Napiši podprogram**, ki izpiše končno stanje mreže. Vhodni podatki so: začetno stanje kocke (katera črka je na kateri ploskvi), velikost mreže in zaporedje položajev kocke.

8. Gen

Zanimajo nas geni dolžine n, torej nizi n znakov, sestavljeni le iz črk $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Podanih je nekaj omejitev oblike „i-ti in j-ti znak niza morata biti enaka“ za različne pare (i, j). Genu, ki ustreza vsem danim omejitvam, rečemo, da je *ugoden*.

(a) Opiši postopek, ki izračuna, koliko je ugodnih genov.

(b) Opiši postopek, ki našteje (izpiše) vse ugodne gene v leksikografskem vrstnem redu.

(c) Opiši postopek, ki za dani m vrne m-ti ugodni gen v leksikografskem vrstnem redu.

9. Okrajšave

Aditya živi in Indiji in dela za založbo Elsevier za majhne denarce. Njegova naloga je, da sestavlja BibTeX-datoteke. To počne po cele dneve, za \$0,02 na vsako referenco. Aditya dobro ve, da avtorji često uporabljajo okrajšana imena revij, npr. *IEEE Trans. Rob.* namesto *IEEE Transactions on Robotics* in podobno.

Aditya meni, da okrajšave niso optimalne, zato bi rad napisal računalniški program, ki sestavi optimalne okrajšave. Pri tem lahko besedo okrajša tako, da vzame nek neprazen prefiks (predpono oz. začetni del niza, torej prvih nekaj črk niza) in doda piko (razen če vzame celo besedo, potem pike ne doda). Na primer, besedo „Journal“ lahko krajša kot „J.“, „Jo.“, „Jou.“, „Jour.“, „Journ.“, „Journa.“ ali „Journal“.

Ima dolg seznam revij. Za vsako ime revije bi rad poiskal okrajšavo, tako da:

- vsaka okrajšana beseda enolično določa originalno besedo (torej se ne sme ista okrajšava v različnih naslovih uporabljati za različne besede) in
- je skupna dolžina vseh okrajšanih imen vseh revij minimalna. (Pika tudi šteje pri dolžini niza!)

Opiši postopek, ki poišče najboljše okrajšave po teh kriterijih. Kot vhodne podatke dobiš seznam imen revij. Besede so ločene s presledki; kratke besedice, ki v okrajšavah ne nastopajo (*on*, *of*, *the*, ...), so že izpuščene.

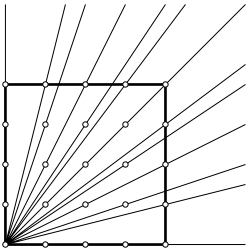
10. Beg iz zapora

V nekem zaporu so se zaporniki odločili za pobeg. Zapor je kvadratne oblike velikosti $n \times n$, zaporniki pa so zaprti v celicah v točki $(0, 0)$. Načrt zapornikov je tak:

- Vsak izmed zapornikov si bo izbral poljubno celoštevilsko točko znotraj ali na robu zapora, razen točke $(0, 0)$.

- Zapornik se bo nato obrnil proti tej točki in nato neprestano tekel naravnost, dokler ne zapusti območja zapora.

Na nesrečo zapornikov je za njihov načrt izvedel direktor zapora. On si seveda ne želi, da bi mu katerikoli izmed zapornikov pobegnil, in bo storil vse, da to prepreči. Odločil se je, da bo na vseh točkah na robu zapora, kjer bi zapornik lahko pobegnil, postavil paznika. Ker pa sam ni dovolj sposoben, je prosil tebe, da mu poveš minimalno število paznikov, ki jih bo moral postaviti.



Primer za $n = 4$. Poltraki, ki izhajajo iz točke $(0, 0)$, torej iz spodnjega levega kota, predstavljajo smeri pobega zapornikov. Paznike bi bilo treba postaviti tam, kjer ti poltraki sekajo zgornji in desni rob zapora. Vidimo, da je poltrakov trinajst, torej potrebujemo trinajst paznikov.

Napiši podprogram (funkcijo), ki kot parameter dobi velikost zapora n in izračuna minimalno potrebno število paznikov, ki jih mora direktor postaviti, tako da nobenemu zaporniku pobeg gotovo ne bo uspel. Primer: ko je velikost zapora $n = 4$, je odgovor 13.

Namig: Ne računaj z realnimi števili.

11. Okleščeni CSS

Pri tej nalogi bomo delali z dokumenti v majhni podmnožici jezika HTML, za opis njihovega položaja pa bomo uporabljali majhno podmnožico jezika CSS. Dokument je podan kot drevesasta struktura vgnezdenih elementov ⟨div⟩ . . . ⟨/div⟩. Če je en element neposredno vgnezden v drugem, rečemo, da je prvi element *otrok* drugega, drugi pa *starš* prvega. Za dva elementa rečemo, da sta *sorojenca*, če imata istega starša. Elemente, ki nimajo nobenih otrok, imenujemo *listi* drevesa.

Pri prikazu (npr. na zaslonu ali na papirju) zaseda vsak element neko pravokotno območje. Pri tej nalogi nas širine in x-koordinate teh pravokotnikov ne bodo zanimale in jih bomo zanemarili; ukvarjali se bomo le z višinami in y-koordinatami.

Za vsak element drevesa poznamo tri atribute:

- height: višina elementa, ki je bodisi nenegativno celo število v pikslih bodisi ima vrednost auto, ki pove, da se višino elementa izračuna glede na višino njegovih otrok. V listih se vrednost auto ne uporablja.

- top: celo število v pikslih, ki se uporablja pri določanju y-koordinate zgornjega roba elementa.

- position: ta ima eno od treh možnih vrednosti, relative, absolute in static.

Opiši postopek, ki izračuna višino (y-koordinato) zgornjega levega kota vsakega elementa, če so pravila razporejanja in zlaganja elementov taka kot v CSS-u:

- Element tipa static se začne tik pod koncem prejšnjega sorojenca tipa relative ali static; če ni nobenega takega sorojenca, pa se začne takoj na vrhu svojega starša.
- Element tipa relative se najprej postavi na tak položaj, kot če bi bil static, nato pa se zamakne za top pikslov navzdol (če je vrednost top negativna, se pač zamakne navzgor), kar pa ne vpliva na položaj naslednjega sorojenca tipa relative ali static (tisti se računa, kot da bi bil tu top enak 0).
- Element tipa absolute se začne top pikslov pod začetkom najbližjega takega prednika, ki je tipa absolute ali relative (pri slednjem se upošteva njegov položaj po zamiku za njegovih top pikslov). Če takega prednika ni, se element začne na y-koordinati top.
- Višina elementa, če ni bila eksplicitno definirana (torej če je height = auto), je točno enaka vsoti višin vseh njegovih otrok tipa relative ali static.

12. Krajšanje matematičnega izraza

Matematiki svoje izraze zelo radi olepšajo tako, da nek podniz izraza nadomestijo z eno novo črko. Podan je niz s, ki ga sestavljajo same male črke angleške abecede. **Opiši postopek**, ki poišče dolžino najkrajšega niza, ki ga lahko dobimo, če si izberemo en podniz niza s in vsako neprvo pojavitev tega podniza nadomestimo z eno samo črko — recimo s črko φ, ki se sicer v s gotovo ne pojavlja; tako ne bo zapletov zaradi možnosti, da bi izraz vseboval svojo lastno okrajšavo. Pojavitve, ki jih nadomestimo s φ, se ne smejo prekrivati. Prva pojavitev podniza se ne izbriše, saj mora biti okrajšava konec koncev nekje definirana.

Primer: če v nizu *abaababababa* za krajšanje uporabimo podniz *aba*, dobimo $\underline{aba}\varphi b\varphi ba$ (prvo pojavitev smo podčrtali); če uporabimo podniz *ab*, se nam niz skrajša v $\underline{ab}a\varphi\varphi\varphi\varphi a$; če uporabimo podniz *baba*, dobimo $aba\underline{ababa}\varphi$. So še druge možnosti, najkrajši niz pa dasta tisti dve, kjer za krajšanje uporabimo podniz *ab* ali *aba*.

13. Karte

Imamo kup kart, R rdečih in B črnih (vse so različne). Izmed njih jih bomo naključno izbrali k (spet samih različnih, torej izbiramo brez vračanja). **Opiši postopek**, ki izračuna, kakšna je verjetnost, da je v takem izboru vsaj m rdečih. Kolikšen je najmanjši k, pri katerem je ta verjetnost vsaj q?

14. Sedežni red

Kinodvorana ima v vsaki izmed v vrst na voljo s sedežev. Sedeži so v sosednjih vrstah poravnani eden za drugim v pravokotno mrežo dimenzij $v \times s$ ($v \leq 100$, $s \leq 10000$). Nekateri sedeži so že zasedeni, drugi pa so prosti. Rezervirati želimo strnjeno skupino n sedežev ($n \leq s$) v čim bolj zadnji vrsti. Če imamo znotraj

izbrane vrste več možnosti, izberemo tako skupino sedežev, ki je čim bolj na sredini celotne vrste. Če še vedno obstaja več možnosti, izberemo bolj levo skupino. **Napiši program** ali podprogram, ki čim učinkoviteje poišče tako skupino sedežev.

Rahlo težja različica: reši nalogo še za primer, ko je mreža zelo velika, vendar je zasedenih sedežev malo v primerjavi z velikostjo mreže. Poleg v, s in n je podano število zasedenih sedežev m in njihove koordinate (številka vrste in številka sedeža v vrsti).

15. Sklicevanje

Dan imamo zakon z veliko členi.

```
(1) Slovenija je demokratična republika.

(2) Slovenija je pravna in socialna država.

(3) Člena (1) in (2) ne veljata vedno in
ne nujno za vse ljudi.

(4) ...
```

Vsak člen je sestavljen iz ene ali več zaporednih nepraznih vrstic; med seboj so členi ločeni s praznimi vrsticami, vsak člen pa se začne z zaporedno številko člena v oklepajih (ki je na začetku vrstice, v kateri stoji). Druge pojavitve številk v oklepajih predstavljajo sklice na druge člene, kot na primer v zgornjem primeru, kjer se člen (3) sklicuje na člena (1) in (2). Zaporedne številke členov so naravna števila od (1) naprej, v naraščajočem vrstnem redu in brez vrzeli (manjkajočih oz. neuporabljenih števil). Predpostavi, da so vsi sklici veljavni, torej da člen, na katerega se sklic nanaša, tudi v resnici obstaja. **Napiši program**, ki prebere besedilo zakona s standardnega vhoda in ugotovi naslednje stvari:

- Ali sklici tvorijo kakšne cikle (npr. tako, da bi se en člen skliceval na drugega, ta na tretjega, ta pa spet na prvega)?
- Če jih ne, kako dolga je najdaljša veriga sklicev?
- Ali se kdaj zgodi, da se člen sklicuje na enega od kasnejših členov?

16. Urejanje zaimkov

Besedilo je sestavljeno iz stavkov. Vsak stavek vsebuje definicije novih pojmov in reference na pojme iz prejšnjih stavkov. Stavke za našo najnovejšo knjigo smo že spisali, zdaj pa jih moramo urediti v smiseln vrstni red.

Da bo besedilo razumljivo, se mora vsaka referenca pojaviti šele za pojmom, na katerega se nanaša. Poleg tega se želimo ogniti situacijam, kjer se reference nanašajo na stavke predaleč nazaj po besedilu. Zato želimo poleg prejšnjega pogoja dodatno minimizirati vsoto razlik razdalj med pojmi in referencami, ki se na njih nanašajo.

Vsak pojem predstavimo z naravnim številom. Vsak stavek predstavimo z dvema seznamoma: seznam novo vpeljanih pojmov in seznam referenc na pojme, definirane v drugih stavkih besedila (torej naloga kljub dosedanji zgodbici ne obsega dela z

nizi). **Opiši postopek**, ki iz teh podatkov poišče tak vrstni red stavkov, pri katerem se vsaka referenca pojavi šele po definiciji in je vsota razdalj med referencami in pojmi, na katere se sklicujejo, minimalna.

17. Goljufije pri telefoniranju

Nekateri ljudje goljufajo pri telefoniranju tako, da zajedajo uporabniško ime nekoga drugega. To pomeni, da pravzaprav različni ljudje z različnih lokacij kličejo tako, kot da bi šlo (z vidika telefonskega operaterja) za istega naročnika. Te goljufije skušamo odkriti tako, da iščemo primere, ko bi npr. nekdo v 1 uri moral prepotovati 300 km ali kaj podobno nerealističnega.

Za dano uporabniško ime imamo seznam parov $\langle \mathit{lokacija}, \mathit{čas} \rangle$, ki predstavljajo vse klice, narejene pod tem uporabniškim imenom. Čas klica je zaokrožen na pol ure, vrstni red parov pa je lahko poljubno premešan. Če leži več zapisov znotraj istega polurnega intervala, zato ne vemo točno, v kakšnem vrstnem redu je človek te lokacije res obiskal. **Opiši postopek**, ki izračuna dolžino najkrajše poti, ki je skladna s temi podatki.

18. Učenje odločitvenih seznamov

Odločitveni seznam je oblika logičnega izraza, ki si ga lahko predstavljamo kot verigo stavkov **if** ... **else if** ... **else**:

if pogoj_1 **then return** b_1;
else if pogoj_2 **then return** b_2;
$\vdots$
else if pogoj_{r-1} **then return** b_{r-1};
else return b_r;

Pri tej nalogi se bomo omejili na take odločitvene sezname, ki imajo v pogojih zgolj disjunkcije (dovolimo negirane spremenljivke, ne pa negiranih disjunkcij), npr. $x_1 \vee \neg x_3 \vee \neg x_5$. V pogojih nastopajo le spremenljivke $x_1, \ldots, x_n$, posamezni pogoj pa lahko vsebuje poljubno število teh spremenljivk.

Podan imamo seznam vhodov in izhodov, ki so bili generirani z neznanim odločitvenim seznamom, ki ga želimo rekonstruirati. **Opiši postopek**, ki za dani seznam vhodov in izhodov (torej vrednosti logičnih spremenljivk x_i in pripadajoči izhod, enega od b_j, ki so tudi logične vrednosti) najde odločitveni seznam, ki se z njimi ujema (ali ugotovi, da se tega ne da) in ima čim manj pogojev, torej čim manjši r.

REŠITVE NALOG ZA PRVO SKUPINO

1. Vesoljske vsote

Vhodni niz bomo brali znak po znak in pri tem v dveh spremenljivkah hranili trenutno število in dosedanjo vsoto. Za potrebe izpisa pa je koristno hraniti še podatek o tem, ali smo že izpisali kak seštevanec ali ne — z drugimi besedami, ali smo v vhodnem nizu doslej že naleteli na kakšno zvezdico „*“ ali ne. To pride prav zato, ker moramo med seštevanci napisati znak + in presledke; to lahko preprosto izvedemo tako, da izpišemo + pred vsakim seštevancem razen pred prvim. Ko pridemo do konca niza, moramo izpisati le še enačaj in vsoto na koncu izraza.

Oglejmo si takšno rešitev v C/C++:

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

void VesoljskeVsote(const char *s)
{
  int trenutno = 1, vsota = 0;
  bool prvi = true;
  while (*s) // Sprehodimo se po znakih vhodnega niza.
    if (*s++ == '-') // Pri minusu povečamo trenutno število.
      trenutno++;
    else // Pri zvezdici prištejemo trenutno število k vsoti.
    {
      // Pred vsakim seštevancem, razen pred prvim, izpišimo +.
      if (prvi) prvi = false; else printf(" + ");
      printf("%d", trenutno); // Izpišimo trenutni seštevanec.
      vsota += trenutno; // Prištejmo ga k vsoti.
    }
  // Na koncu izpišimo še enačaj in vsoto.
  printf(" = %d\n", vsota);
}
```

Še prav taka rešitev v pythonu:

```
import sys

def VesoljskeVsote(s):
  trenutno = 1; vsota = 0; prvi = True
  for c in s: # Sprehodimo se po znakih vhodnega niza.
    # Pri minusu povečamo trenutno število.
    if c == "-": trenutno += 1
    else: # Pri zvezdici prištejemo trenutno število k vsoti.
      # Pred vsakim seštevancem, razen pred prvim, izpišimo +.
      if prvi: prvi = False
      else: sys.stdout.write(" + ")
      sys.stdout.write(str(trenutno))    # Izpišimo trenutni seštevanec.
      vsota += trenutno                  # Prištejmo ga k vsoti.
  # Na koncu izpišimo še enačaj in vsoto.
  sys.stdout.write(" = %d\n" % vsota)
```

Šlo bi tudi brez spremenljivke prvi, saj lahko to, da smo pri prvem seštevancu, prepoznamo tudi po tem, da je takrat vsota še vedno 0 (kasneje bo vsota vedno večja od 0, saj so tudi seštevanci vsi večji od 0).

Nalogo lahko rešimo tudi malo drugače. Spodnja rešitev v pythonu med branjem vhodnega niza ničesar ne izpisuje, pač pa dodaja seštevance v seznam cleni. Ko pride do konca niza, s pomočjo tega seznama izpiše najprej vse člene vsote (ločene s plusi) in na koncu še vrednost vsote (ki jo izračuna s pythonovo funkcijo sum).

```
def VesoljskeVsote2(s):
  trenutno = 1; cleni = []
  for c in s:
    # Pri minusu povečamo trenutno število.
    if c == "-": trenutno += 1
    # Pri zvezdici dodamo trenutno število v seznam členov.
    else: cleni.append(trenutno)
  # Izpišimo člene, ločene s plusi, in nato še enačaj in vsoto.
  print("%s = %d" % (" + ".join(str(clen) for clen in cleni), sum(cleni)))
```

2. Ključ

Naloga pravi, da dobimo opis ključa kot celo število (od 0 do 999 999). Da pridemo do posameznih števk, si lahko pomagamo z dejstvom, da je ostanek po deljenju števila z 10 ravno najbolj desna števka tega števila (enice), celi del količnika pri tem deljenju pa je tisto, kar od števila ostane, če mu zadnjo števko pobrišemo. Tako lahko v zanki pregledujemo zareze ključa od desne proti levi.

Pri vsaki zarezi preverimo, če je od 1 do 8 (prvi pogoj iz besedila naloge). Pri vsaki zarezi razen prve tudi preverimo, če je različna od prejšnje (tretji pogoj), vendar ne za več kot pet (drugi pogoj); pri tem si moramo torej prejšnjo zarezo zapomniti (v spodnji rešitvi jo shranimo v spremenljivko prejsnja).

Za četrti pogoj je koristno imeti tabelo, v kateri štejemo, kolikokrat smo kakšno števko že videli; na začetku postavimo vse elemente na 0, nato pa pri vsaki prebrani zarezi ustrezni element tabele povečamo za 1 in pogledamo, če je zdaj večji od 2.

Za vsak pogoj imamo še eno logično spremenljivko (ok1, ..., ok4), ki nam pove, ali je ključ ta pogoj prekršil ali ne; vrednosti teh spremenljivk na koncu izpišemo.

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

enum { Dolzina = 6, MaxRazlika = 5 };

void Preveri(int kljuc)
{
  bool ok1 = true, ok2 = true, ok3 = true, ok4 = true;
  int stPojavitev[10] = { }, zareza = -1;
  for (int i = 0; i < Dolzina; i++)
  {
    // Izluščimo naslednjo zarezo iz spremenljivke 'kljuc'.
    int prejsnja = zareza; zareza = kljuc % 10; kljuc /= 10;

    // Pogoj 1: ali so vse zareze od 1 do 8?
    if (zareza < 1 || zareza > 8) ok1 = false;

    if (i > 0) { // Ker to ni prva zareza, jo lahko primerjamo s prejšnjo.
      int razlika = zareza - prejsnja;

      // Pogoj 2: sosednji zarezi se ne smeta preveč razlikovati.
      if (razlika < -MaxRazlika || razlika > MaxRazlika) ok2 = false;

      // Pogoj 3: sosednji razrezi ne smeta biti enaki.
```

```
        else if (razlika == 0) ok3 = false; }
      // Pogoj 4: ne smemo imeti treh ali več enakih zarez.
      if (++stPojavitev[zareza] > 2) ok4 = false;
    }
    // Izpišimo rezultate.
    printf("Ključ ustreza naslednjim pravilom: 1 %s, 2 %s, 3 %s, 4 %s.\n",
      ok1 ? "da" : "ne", ok2 ? "da" : "ne", ok3 ? "da" : "ne", ok4 ? "da" : "ne");
}
```

Zapišimo to rešitev še v pythonu. Za spremembo bomo namesto štirih logičnih spremenljivk uporabili tabelo (list v pythonu) s štirimi elementi:

```
Dolzina = 6; MaxRazlika = 5

def Preveri(kljuc):
    ok = [True] * 4; stPojavitev = [0] * 10; zareza = 1
    for i in range(Dolzina):
        # Izluščimo naslednjo zarezo iz spremenljivke „kljuc“.
        prejsnja = zareza; zareza = kljuc % 10; kljuc //= 10
        # Pogoj 1: ali so vse zareze od 1 do 8?
        if zareza < 1 or zareza > 8: ok[0] = False
        if i > 0: # Ker to ni prva zareza, jo lahko primerjamo s prejšnjo.
            razlika = zareza − prejsnja
            # Pogoj 2: sosednji zarezi se ne smeta preveč razlikovati.
            if abs(razlika) > MaxRazlika: ok[1] = False
            # Pogoj 3: sosednji razrezi ne smeta biti enaki.
            elif razlika == 0: ok[2] = False
        # Pogoj 4: ne smemo imeti treh ali več enakih zarez.
        stPojavitev[zareza] += 1
        if stPojavitev[zareza] > 2: ok[3] = False
    # Izpišimo rezultate.
    print("Ključ ustreza naslednjim pravilom: 1 %s, 2 %s, 3 %s, 4 %s." %
        tuple("da" if b else "ne" for b in ok))
```

3. Obračanje jogija

Jogi ima dve strani (zgornjo in spodnjo — to sta tisti dve ploskvi, ki sta pravokotni na os z), na vsaki strani pa imamo lahko glavo na enem od dveh koncev (pri enem od krajših robov tiste strani jogija, torej tistih, ki sta vzporedna z osjo x). Tako lahko vsako od štirih mest, kjer imamo lahko glavo, opišemo s parom bitov (s, k), ki povesta stran in konec. Drug pogled na tak par bitov pa je seveda ta, da si ga predstavljamo kot število $2s + k$, torej eno od števil 0, 1, 2 ali 3.

S pomočjo tega številčenja mest lahko razmislimo, kaj se zgodi pri vrtenju jogija (pri tem je koristno gledati na sliko, ki kaže, kam so usmerjene koordinatne osi). Če smo doslej spali na mestu (s, k) in jogi nato zavrtimo za 180 stopinj okrog osi z, bo pod našo glavo prišlo mesto $(s, 1 - k)$, torej na isti strani jogija, vendar na drugem koncu. Podobno, če ga zavrtimo okrog osi y, bo pod našo glavo prišlo mesto $(1-s, k)$, torej na istem koncu jogija, vendar na drugi strani. Če pa ga zavrtimo okrog osi x, pride pod našo glavo mesto $(1 - s, 1 - k)$.

Naša rešitev bo morala v globalnih spremenljivkah za vsa štiri mesta hraniti podatek o tem, kolikokrat je uporabnik doslej spal na njih. Ko uporabnik pokliče našo funkcijo, moramo najprej ustrezno povečati števec obrabljenosti za tisto mesto, na katerem je spal doslej, nato pa pogledati, katero mesto je zdaj najmanj obrabljeno (če je tu več enako dobrih možnosti, je vseeno, katero uporabimo), in s pomočjo razmisleka iz prejšnjega odstavka svetovati, kako je treba zasukati jogi, da bo prišlo pod uporabnikovo glavo najmanj obrabljeno mesto: če je uporabnik doslej spal na (s, k), najmanj obrabljeno pa je $(1 - s, k)$, mu moramo svetovati vrtenje okrog osi y in podobno. Če imamo mesta predstavljena z dvobitnimi celimi števili od 0 do 3, je dovolj že, če pogledamo, v katerih bitih se razlikujeta stara in nova številka mesta (pri tem lahko uporabimo operator xor oz. ^): če v nižjem (konec), je treba obrniti jogi okrog osi z; če v višjem (stran), okrog y; če v obeh, okrog x; če v nobenem, pa jogija ni treba obračati.

Na začetku sicer ne vemo, kako je jogi obrnjen, vendar to za nas tudi ni pomembno, saj so vsa mesta popolnoma neobrabljena. Tisto mesto, na katerem je uporabnikova glava na začetku izvajanja programa, lahko preprosto razglasimo za $(0, 0)$, ostale kombinacije dveh bitov pa potem pač predstavljajo ostala tri mesta odvisno od tega, ali ležijo na istem ali na drugem koncu/strani kot začetno mesto.

```
int obrabljenost[4] = { }, trenutno = 0;

char ObrniJogi(int n)
{
  // Popravimo števec obrabljenosti trenutnega mesta.
  obrabljenost[trenutno] += n;

  // Poglejmo, katero mesto je zdaj najmanj obrabljeno.
  int novo = 0; for (int i = 1; i < 4; i++)
    if (obrabljenost[i] < obrabljenost[novo]) novo = i;

  // Poglejmo, kako obrniti jogi, da pride mesto „novo“ tja,
  // kjer je bilo doslej mesto „trenutno“.
  char kakoObrniti = "nzyx"[novo ^ trenutno];

  // Zapomnimo si, da bo uporabnik odslej spal na novem mestu.
  trenutno = novo; return kakoObrniti;
}
```

Zapišimo to rešitev še v pythonu:

```
obrabljenost = [0] * 4; trenutno = 0

def ObrniJogi(n):
  global trenutno

  # Popravimo števec obrabljenosti trenutnega mesta.
  obrabljenost[trenutno] += n;

  # Poglejmo, katero mesto je zdaj najmanj obrabljeno.
  novo = 0
  for i in range(1, 4):
    if obrabljenost[i] < obrabljenost[novo]: novo = i

  # Poglejmo, kako obrniti jogi, da pride mesto „novo“ tja,
  # kjer je bilo doslej mesto „trenutno“.
  kakoObrniti = "nzyx"[novo ^ trenutno];

  # Zapomnimo si, da bo uporabnik odslej spal na novem mestu.
  trenutno = novo; return kakoObrniti
```

4. Zobna ščetka

Naš program bo tekel v neskončni zanki. Vsakič preverimo stanje tipke; pritisk na tipko prepoznamo po tem, da je trenutno pritisnjena, ob prejšnjem preverjanju pa še ni bila. Poleg tega si zapomnimo tudi, ali je motor trenutno prižgan ali ne (v spodnji rešitvi je to spremenljivka prizgana).

Če je uporabnik pritisnil na tipko in je bil motor doslej ugasnjen, ga moramo prižgati; ugasniti pa ga moramo, če je bil doslej prižgan in če je uporabnik zdaj pritisnil na tipko ali pa če motor teče že 120 sekund. Če ni izpolnjen nobeden od teh pogojev, moramo motor pustiti v dosedanjem stanju (stavek **continue** v spodnji rešitvi), sicer pa mu stanje spremenimo (ga vklopimo, če je bil prej izklopljen, oz. izklopimo, če je bil vklopljen). Ko motor poženemo, moramo tudi prižgati štoparico, ko pa ga ugasnemo, moramo štoparico ustaviti.

```
int main()
{
  bool prizgana = false, tipka = false;
  while (true)
  {
    // Zapomnimo si prejšnje stanje tipke in poglejmo sedanje.
    bool prejTipka = tipka; tipka = Tipka();

    // Če uporabnik pritisne tipko in je ščetka ugasnjena,
    // jo bo treba prižgati. Pri tem tudi poženimo štoparico.
    if (tipka && ! prejTipka && ! prizgana) PozeniUro();

    // Če je ščetka prižgana in uporabnik pritisne tipko ali pa je prižgana že
    // več kot 120 sekund, jo bo treba ugasniti. Pri tem tudi ustavimo štoparico.
    else if (tipka && ! prejTipka || prizgana && OdcitajUro() > 120) UstaviUro();

    // Sicer lahko ščetka ostane v sedanjem stanju.
    else continue;

    // Postavimo motor v novo stanje in si ga zapomnimo.
    prizgana = ! prizgana; Motor(prizgana);
  }
}
```

Zapišimo to rešitev še v pythonu:

```
prizgana = False; tipka = False
while True:
  # Zapomnimo si prejšnje stanje tipke in poglejmo sedanje.
  prejTipka = tipka; tipka = Tipka()

  # Če uporabnik pritisne tipko in je ščetka ugasnjena,
  # jo bo treba prižgati. Pri tem tudi poženimo štoparico.
  if tipka and not prejTipka and not prizgana: PozeniUro()

  # Če je ščetka prižgana in uporabnik pritisne tipko ali pa je prižgana že
  # več kot 120 sekund, jo bo treba ugasniti. Pri tem tudi ustavimo štoparico.
  elif tipka and not prejTipka or prizgana and OdcitajUro() > 120: UstaviUro()

  # Sicer lahko ščetka ostane v sedanjem stanju.
  else: continue

  # Postavimo motor v novo stanje in si ga zapomnimo.
  prizgana = not prizgana; Motor(prizgana)
```

5. Plonkanje

Vhodne podatke si lahko predstavljamo kot zaporedje parov $\langle pomočnik, plonkar \rangle$. Vsako od treh podnalog lahko rešimo s po enim prehodom po tem seznamu.

V prvem prehodu si pripravimo množico vseh pomočnikov in množico vseh plonkarjev; izvirni tekmovalci so potem preprosto tisti, ki so v množici pomočnikov, ne pa tudi v množici plonkarjev. (Namesto tega lahko naredimo tudi dva prehoda: v prvem sestavimo množico vseh pomočnikov, v drugem pa iz nje pomečemo vse plonkarje, pa nam ostanejo ravno vsi izvirni tekmovalci; ali pa v prvem prehodu sestavimo množico vseh plonkarjev, v drugem pa dodamo v množico izvirnih tekmovalcev le tiste pomočnike, ki niso v množici plonkarjev iz prvega prehoda.)

V drugem prehodu pripravimo množico plonkarjev prvega reda tako, da pri vsakem paru z vhodnega seznama pogledamo, če je pomočnik eden od izvirnih tekmovalcev, in če je, dodamo plonkarja v množico plonkarjev prvega reda.

V tretjem prehodu pripravimo množico plonkarjev drugega reda tako, da pri vsakem paru pogledamo, če je pomočnik eden od plonkarjev prvega reda, in če je, dodamo plonkarja med plonkarje drugega reda.

Množice lahko predstavimo z razpršenimi tabelami ali pa tudi z navadno tabelo logičnih vrednosti (za vsakega tekmovalca po ena, ki pove, če tekmovalec pripada tej množici ali ne), če so številke tekmovalcev dovolj majhne, da jih lahko uporabljamo kot indekse v tabelo.

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

struct Par { int pomocnik, plonkar; };
typedef unordered_set<int> Mnozica;

void Prepisovanje(const vector<Par> &pari,
                  Mnozica &izvirni, Mnozica &plonkarji1, Mnozica &plonkarji2)
{
  // Pripravimo množico vseh plonkarjev.
  Mnozica plonkarji; for (auto &P : pari) plonkarji.insert(P.plonkar);
  // Izvirni tekmovalci so pomočniki, ki niso plonkarji.
  izvirni.clear(); for (auto &P : pari)
    if (! plonkarji.count(P.pomocnik)) izvirni.insert(P.pomocnik);
  // Pripravimo množico plonkarjev prvega reda.
  plonkarji1.clear(); for (auto &P : pari)
    if (izvirni.count(P.pomocnik)) plonkarji1.insert(P.plonkar);
  // Pripravimo množico plonkarjev drugega reda.
  plonkarji2.clear(); for (auto &P : pari)
    if (plonkarji1.count(P.pomocnik)) plonkarji2.insert(P.plonkar);
}
```

Še bolj jedrnati smo lahko v pythonu:

```
def Prepisovanje(pari):
  plonkarji = set(plonkar for (pomocnik, plonkar) in pari)
  izvirni = set(pomocnik for (pomocnik, plonkar) in pari if pomocnik not in plonkarji)
  plonkarji1 = set(plonkar for (pomocnik, plonkar) in pari if pomocnik in izvirni)
  plonkarji2 = set(plonkar for (pomocnik, plonkar) in pari if pomocnik in plonkarji1)
  return (izvirni, plonkarji1, plonkarji2)
```

Ni si težko predstavljati, da bi lahko definicije iz naše naloge še posplošili in govorili o plonkarjih tretjega reda, pa četrtega in tako naprej. Našo dosedanjo rešitev bi lahko nadaljevali na podoben način kot doslej in računali še plonkarje višjih redov, vendar pa bi se ob tem pokazala slabost te rešitve: za vsak naslednji red potrebuje po en prehod čez celoten seznam vhodnih parov. Če imamo n tekmovalcev in s tem tudi $O(n)$ parov v vhodnem seznamu (ker je le toliko parov, kolikor je vseh plonkarjev), porabimo tako $O(n)$ časa za vsak red, ki ga hočemo računati; in ker bi šli lahko redovi v najslabšem primeru do $n-1$, bi ta postopek porabil takrat $O(n^2)$ časa.

Boljšo rešitev dobimo, če v prvem prehodu čez vhodni seznam predelamo podatke v drugačno obliko: za vsakega tekmovalca pripravimo seznam tistih, ki so plonkali od njega. Spotoma lahko za vsakega označimo še, ali je sam od koga plonkal ali ne. To je dovolj, da s še enim prehodom čez vse tekmovalce pripravimo seznam izvirnih (to so tisti, ki niso od nikogar plonkali). Nato lahko seznam plonkarjev prvega reda pripravimo tako, da gremo po vseh izvirnih tekmovalcih in staknemo skupaj sezname vseh tistih, ki so prepisovali od njih. Pomembna razlika v primerjavi s prvotno rešitvijo je torej ta, da nam ni treba iti še enkrat po vseh podatkih, ampak le po izvirnih tekmovalcih. V nadaljevanju gremo lahko na enak način po seznamu tekmovalcev prvega reda in staknemo skupaj sezname tistih, ki so prepisovali od njih, pa dobimo seznam tekmovalcev drugega reda. Tako bi lahko nadaljevali še za vse višje redove; vsakega tekmovalca največ enkrat dodamo na en tak seznam (pri njegovem redu) in se nato enkrat sprehodimo po seznamu tistih, ki so prepisovali od njega (ko računamo za eno višji red); časovna zahtevnost celotnega postopka, za vse redove skupaj, je tako le še $O(n)$. Lepo je tudi to, da nam ni več treba delati z množicami (ki so implementirane npr. z razpršenimi tabelami; pri prejšnji rešitvi smo jih potrebovali, da smo lahko učinkovito preverjali, ali pomočnik v nekem paru pripada prejšnjemu redu, ko računamo naslednji red), ampak so dovolj že navadni seznami (npr. vektorji).

Oglejmo si implementacijo takšne rešitve v C++:

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

void Prepisovanje(const vector<Par> &pari,
                  vector<int> &izvirni, vector<int> &plonkarji1, vector<int> &plonkarji2)
{
  struct Tekmovalec
  {
    bool jePlonkar = false; // ali je on plonkal
    vector<int> plonkarji; // kdo je plonkal od njega
  };
  // Pripravimo za vsakega tekmovalca seznam, kdo je plonkal od njega.
  unordered_map<int, Tekmovalec> T;
  for (auto &P : pari) {
    T[P.pomocnik].plonkarji.push_back(P.plonkar);
    T[P.plonkar].jePlonkar = true; }
  izvirni.clear(); plonkarji1.clear(); plonkarji2.clear();
  // Začeli bomo s seznamom izvirnih tekmovalcev.
  for (auto &[u, U] : T) if (! U.jePlonkar) izvirni.push_back(u);
```

```
  // Kdor je plonkal od njih, je plonkar prvega reda.
  for (auto u : izvirni) for (int v : T[u].plonkarji) plonkarji1.push_back(v);

  // Kdor je plonkal od teh, pa je plonkar drugega reda.
  for (auto u : plonkarji1) for (int v : T[u].plonkarji) plonkarji2.push_back(v);
}
```

Zapišimo to rešitev še v pythonu:

```
def Prepisovanje2(pari):
  class Tekmovalec:
    def __init__(self): self.jePlonkar = False; self.plonkarji = []

  # Pripravimo za vsakega tekmovalca seznam, kdo je plonkal od njega.
  T = {}
  for (pomocnik, plonkar) in pari:
    if pomocnik not in T: T[pomocnik] = Tekmovalec()
    if plonkar not in T: T[plonkar] = Tekmovalec()
    T[pomocnik].plonkarji.append(plonkar)
    T[plonkar].jePlonkar = True

  # Začeli bomo s seznamom izvirnih tekmovalcev.
  izvirni = [u for (u, U) in T.items() if not U.jePlonkar]

  # Kdor je plonkal od njih, je plonkar prvega reda.
  plonkarji1 = [v for u in izvirni for v in T[u].plonkarji]

  # Kdor je plonkal od teh, pa je plonkar drugega reda.
  plonkarji2 = [v for u in plonkarji1 for v in T[u].plonkarji]

  return (izvirni, plonkarji1, plonkarji2)
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Metanje na koš

Vhodni niz lahko v mislih razdelimo na strnjene skupine enic, med katerimi so ničle. Kjer stojita dve ničli skupaj, si mislimo med njima prazno skupino enic. Za niz 35 metov iz besedila naloge dobimo na primer:

$$1\,1\,1\,0\,1\,1\,1\,1\,0\ \cdot\ 0\,1\,1\,0\,1\,1\,1\,1\,1\,0\,1\,1\,1\,1\,0\ \cdot\ 0\,1\,0\,1\,1\,1\,1\,1\,1\,1\,1\,0\,\cdot,$$

pri čemer smo prazne skupine enic označili s pikami „·“. Zapišimo dolžine tako dobljenih skupin enic kot seznam:

$$3, 4, 0, 2, 5, 4, 0, 1, 7, 0.$$

Vsaka skupina $k+1$ zaporednih členov v tem seznamu predstavlja zaporedje metov na koš, v katerem je k ničel (metov mimo koša), ostalo pa so enice (zadetki); to pa je prav táko zaporedje, po kakršnem spraševanje naša naloga. Skupno število zadetkov v takem zaporedju dobimo tako, da seštejemo tistih $k+1$ zaporednih členov našega seznama. To lahko počnemo v zanki: gremo po zaporedju, na vsakem mestu izračunamo vsoto zadnjih $k+1$ členov in si zapomnimo največjo od tako dobljenih vsot. Pri tem pa je koristno upoštevati še to, da nam vsote ni treba računati vsakič od začetka; ko se premaknemo za en člen naprej, pridobi vsota na desni en člen, na levi pa najstarejšega izgubi, tako da lahko staro vsoto preprosto in poceni popravimo v novo.

```
int NajdaljsiNizZadetkov(const char *s, int k)
{
  // Pripravimo seznam z dolžinami strnjenih skupin enic.
  // Kjer sta dve ničli skupaj, si mislimo med njima skupino 0 enic.
  // Prvi element seznama predstavlja enice pred prvo ničlo.
  vector<int> skupine; skupine.push_back(0);
  for (int i = 0; s[i]; i++)
    // Ko pridemo do ničle, začnemo novo skupino enic.
    if (s[i] == '0') skupine.push_back(0);
    // Ko smo pri enici, se trenutna skupina enic podaljša.
    else ++skupine.back();

  // Računajmo vsote po k + 1 zaporednih elementov seznama.
  int vsota = 0, naj = 0;
  for (int i = 0; i < skupine.size(); i++)
  {
    // Prištejmo trenutni element.
    vsota += skupine[i];
    // Če ima vsota zdaj že k + 2 členov, najstarejšega odštejmo.
    if (i > k) vsota -= skupine[i - (k + 1)];
    // Največjo vsoto si zapomnimo.
    naj = max(naj, vsota);
  }
  return naj; // Vrnimo najboljšo rešitev.
}
```

Ta rešitev v najslabšem primeru porabi $O(n)$ pomnilnika za seznam z dolžinami skupin enic, če je n dolžina vhodnega niza. Lahko bi jo še malo izboljšali, če bi seznam gradili sproti, medtem ko se premikamo po njem, in iz njega tudi sproti brisali stare elemente, ki jih ne bomo več potrebovali (tiste, ki so več kot k mest levo od trenutnega); poraba pomnilnika bi se s tem zmanjšala na $O(k)$.

Na podobni ideji temelji tudi naslednja rešitev, ki namesto dolžin skupin enic gleda položaje ničel. Ko se premikamo po nizu znak po znak od leve proti desni, vzdržujmo v tabeli oz. vektorju nicle položaje zadnjih $k+1$ ničel, ki smo jih doslej videli. Podniz, ki se začne takoj za najbolj levo izmed teh ničel in traja vse do našega trenutnega položaja, vsebuje torej k ničel, ostalo pa so enice; zato je to podniz take oblike, kakršno zahteva naša naloga, število enic v njem pa je ravno za k manjše od dolžine podniza. Med vsemi tako dobljenimi možnostmi si spet zapomnimo najdaljšo.

Za inicializacijo tega postopka se je koristno pretvarjati, da levo od začetka našega vhodnega niza stoji še $k+1$ ničel (če indekse v vhodnem nizu štejemo od 0 do $n-1$, pri čemer je n dolžina niza, si mislimo, da stojijo ničle tudi na indeksih $-(k+1), -k, \ldots, -2, -1$).

Vektor zadnjih $k+1$ ničel je koristno uporabljati kot krožno tabelo (*ring buffer*): zapomnimo si, na katerem indeksu v njej je najbolj leva izmed teh $k+1$ ničel (spremenljivka prva); in ko naletimo na naslednjo ničlo, vpišemo njen položaj čez tisto najbolj levo, indeks prva pa pomaknemo za eno mesto naprej, ker je tam zdaj najbolj leva izmed preostalih ničel. Tako imamo pri vsaki ničli le $O(1)$ dela, da ustrezno popravimo vektor nicle.

```
int NajdaljsiNizZadetkov2(const char *s, int k)
{
  // V vektorju „nicle“ bomo hranili indekse zadnjih k + 1 ničel; uporabljali jo
  // bomo kot krožno tabelo, najbolj leva od teh ničel je nicle[prva].
  vector<int> nicle(k + 1); int prva = 0;

  // Za začetek se pretvarjajmo, da je levo od niza še k + 1 ničel.
  for (int i = 0; i <= k; i++) nicle[k - i] = -i - 1;

  // Sprehodimo se po nizu.
  int naj = 0;
  for (int i = 0; s[i]; i++)
    if (s[i] == '1')
      // Znaki na indeksih od nicle[prva] + 1 do k tvorijo zaporedje
      // s k ničlami, ostalo pa so enice.
      naj = max(naj, i - nicle[prva] - k);
    else { // Najbolj levo ničlo v seznamu „nicle“ povozimo s trenutno ničlo.
      nicle[prva] = i; prva = (prva + 1) % (k + 1); }
  return naj; // Vrnimo najboljšo rešitev.
}
```

Oglejmo si še eno zelo preprosto in elegantno rešitev. Če imamo vhodno zaporedje $s_1 s_2 \ldots s_n$, je podzaporedje, po katerem sprašuje naloga, oblike $s_\ell s_{\ell+1} \ldots s_{d-1} s_d$ za neka ℓ in d. Pri tem imamo omejitev, da mora biti v tem podzaporedju kvečjemu k ničel in čim več enic.

Če v mislih fiksiramo d, torej desni konec podzaporedja, je s tem najboljši ℓ (pri tem d) že enolično določen: ℓ je treba postaviti tako daleč na levo, da bi pri $\ell - 1$

v podzaporedje že prišla $(k+1)$-va ničla (če pa na območju od 1 do d v vhodnem zaporedju sploh ni toliko ničel, lahko vzamemo $\ell = 1$).

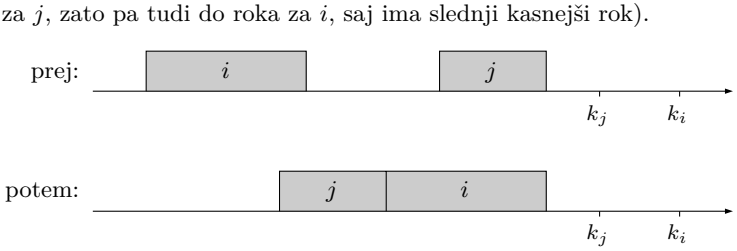
Razmislimo zdaj o tem, kaj se bo zgodilo, ko bomo desni rob podzaporedja postavili pri $d+1$ namesto pri d. Kako se bo zaradi tega spremenil najboljši ℓ? V levo se gotovo ne more premakniti, kajti če je bilo že v podnizu $s_{\ell-1} \ldots s_d$ preveč ničel, jih bo preveč tudi v $s_{\ell-1} \ldots s_{d+1}$. Ko se torej d povečuje, se tudi ℓ lahko le povečuje oz. ostaja enak; ali, z drugimi besedami: ko se desni rob podzaporedja premika v desno, se tudi levi rob podzaporedja premika v desno (kadar se sploh premakne), nikoli pa v levo.

Lahko torej v zanki počasi povečujemo d za 1, po vsakem takem povečanju pa v vgnezdeni zanki povečamo ℓ za toliko, kolikor je treba, da število ničel v podzaporedju $s_\ell \ldots s_d$ ne preseže k. Po vsakem takem popravku je podzaporedje $s_\ell \ldots s_d$ kandidat za najboljše podzaporedje, po katerem sprašuje naloga; med temi kandidati si zapomnimo najboljšega. Pazimo pa na to, da naloga ne sprašuje po dolžini podzaporedja, ampak po številu zadetkov (enic) v njem.

```
int NajdaljsiNizZadetkov3(const char *s, int k)
{
  int naj = 0, od = 0, nicle = 0;
  for (int Do = 0; s[Do]; ++Do)
  {
    // Med znaki s[od... Do − 1] je „nicle“ ničel.
    if (s[Do] == '0') ++nicle;
    // Med znaki s[od... Do] je „nicle“ ničel. Če je to več kot k,
    // povečujmo „od“, dokler se število ničel ne zmanjša na k.
    while (od <= Do && nicle > k) if (s[od++] == '0') −−nicle;
    // Če je območje od... Do najboljša rešitev doslej, si ga zapomnimo.
    if (nicle <= k) naj = max(naj, Do − od + 1 − nicle);
  }
  return naj; // Vrnimo najboljšo rešitev.
}
```

2. Ne odlašaj na jutri, kar lahko storiš pojutrišnjem

Hitro se dá videti, da je smiselno obveznosti opravljati v takem vrstnem redu, v kakršnem morajo biti končane (torej po naraščajočih vrednostih k_i). O tem se lahko prepričamo takole: recimo, da opravimo obveznost i in nato (mogoče z nekaj dnevi premora) še obveznost j, pri čemer pa je $k_j < k_i$. Če je med njima premor, lahko i zamaknemo toliko naprej, da se konča takrat, ko se j začne, pa bo razpored še vedno veljaven (i se konča pred j, njegov rok pa je za j-jevim; če se j tu konča pravočasno, se i tudi), odlašanja pa je še več. Zdaj smo torej v stanju, ko se i konča takrat, ko se j začne; lahko ju enostavno zamenjamo, pa se bo j končal bolj zgodaj kot prej (in torej še vedno do roka), i pa se bo končal takrat, ko se je prej j (kar je do roka za j, zato pa tudi do roka za i, saj ima slednji kasnejši rok).



Tako lahko torej razpored, v katerem opravila niso urejena po naraščajočem roku k_i, predelamo v takega, kjer so tako urejena, ne da bi se kaj poslabšal.

Za začetek lahko torej obveznosti uredimo po k_i in jih nato obdelujemo od konca proti začetku, torej po padajočih k_i. Zadnjo obveznost (tisto z največjim k_i) je smiselno začeti ob času $z_i := k_i - d_i$; kasneje je ne smemo (ker potem ne bi bila končana do roka), bolj zgodaj pa tudi ne (ker bi v tem primeru razpored lahko še izboljšali, če bi z začetkom te obveznosti še malo počakali). Razmislimo zdaj o predzadnji obveznosti — recimo ji j: ne smemo je začeti kasneje kot ob $k_j - d_j$ (ker potem ne bi bila končana do roka), pa tudi ne kasneje kot ob $z_i - d_j$ (ker potem ne bi bila končana do takrat, ko moramo začeti zadnjo obveznost (katere z_i smo določili malo prej). Obveznost j moramo torej začeti najkasneje ob $z_j := \min\{k_j, z_i\} - d_j$; bolj zgodaj kot to pa je nima smisla začenjati, saj bi se v tem primeru dalo razpored še izboljšati, če bi njen začetek malo odložili.

Enako lahko razmišljamo tudi pri vseh zgodnejših obveznostih; vsaka se mora torej končati do svojega roka in tudi do začetka naslednje obveznosti. Tako lahko načeloma za vsako obveznost i določimo njen začetni čas z_i. Naloga pravi, da so dnevi oštevilčeni z naravnimi števili od začetka leta, tako da, če na koncu dobimo pri najzgodnejši obveznosti $z_i \leq 0$, je to znak, da je problem nerešljiv (saj dni s številkami, manjšimi od 1, ni), sicer pa smo dobili primeren razpored, po kakršnem sprašuje naloga.

Oglejmo si še implementacijo tega postopka v jeziku C++. Vhodne podatke bomo predstavili z vektorjem majhnih struktur, ki naj ob klicu za vsako obveznost povedo njeno dolžino d_i in rok k_i, do katerega mora biti končana, naša funkcija pa bo v vsako zapisala primeren dan začetka z_i. Funkcija vrne logično vrednost, ki pove, ali obstaja veljaven razpored (tak, v katerem so vsi časi pozitivni). Začetek naslednje obveznosti hranimo v spremenljivki zacNaslednje; ob koncu nam ta pove začetek prve (najzgodnejše) obveznosti, za katero moramo le še preveriti, če je večja od 0.

```
struct Obveznost { int rok, dolzina, zacetek; };

bool PoisciRazpored(vector<Obveznost>& O)
{
  int n = O.size();
  // Pripravimo si vrstni red, v katerem so obveznosti urejene padajoče po roku.
  vector<int> vrstniRed(n);
  for (int i = 0; i < n; i++) vrstniRed[i] = i;
  sort(vrstniRed.begin(), vrstniRed.end(), [&] (int i, int j) {
    return O[i].rok > O[j].rok; });
  // V tem vrstnem redu jih pregledujmo in določajmo čas začetka.
  int zacNaslednje = 1;
  for (int i = 0; i < n; i++) {
    auto &o = O[vrstniRed[i]];
    int konec = (i == 0) ? o.rok : min(o.rok, zacNaslednje - 1);
    // Ta obveznost se mora končati najkasneje na dan „konec“.
    o.zacetek = zacNaslednje = konec - O.dolzina + 1; }
  return zacNaslednje > 0;
}
```

3. Lenoba

Naloga pravi, da moramo priti pred poldnem in oditi po njem. Za začetek lahko torej v mislih pobrišemo sodelavce, ki pridejo šele opoldne ali kasneje (kajti oni nas gotovo ne bodo videli priti na delo), in tudi tiste, ki odidejo najkasneje opoldne (kajti oni nas gotovo ne bodo videli oditi).

Recimo, da nam ostane n sodelavcev in da i-ti od njih pride ob času z_i in odide ob času k_i. Kot smo videli v prejšnjem odstavku, so vsi z_i pred poldnevom, vsi k_i pa po poldnevu. Uredimo sodelavce naraščajoče po času prihoda, tako da bo $z_1 \leq z_2 \leq \ldots \leq z_n$.

Recimo našemu času prihoda p. Opazimo lahko, da če p leži na območju $z_i \leq p < z_{i+1}$, nas bo videlo priti prvih i sodelavcev, ostali pa ne; takrat torej ne smemo oditi prej kot ob času $\max\{k_1, \ldots, k_i\} + 1$ (ni pa tudi nobenega razloga, da bi odšli kasneje). Ker je ta čas odhoda enak za vse prihode na območju $z_i \leq p < z_{i+1}$, je najbolje, če pridemo čisto na koncu tega intervala, ob $p = z_{i+1} - 1$ — tako bomo v službi najkrajši čas.

Poseben primer je še možnost, da pridemo pred prvim sodelavcem, torej ob $p < z_1$. Takrat nas nihče ne vidi priti, zato je vseeno, kdo nas vidi oditi; torej lahko odidemo že eno nanosekundo po poldnevu. Ker je ta čas odhoda enak za vse $p < z_1$, je med njimi spet smiselno vzeti najkasnejšega, torej $p = z_1 - 1$.

Zdaj torej poznamo vse možne čase prihoda in za vsakega od njih tudi čas odhoda; med temi možnostmi bomo na koncu seveda vrnili tisto, pri kateri smo v službi najmanj časa. Zapišimo našo rešitev še v C++:

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

typedef long long int llint;
struct Par { llint prihod, odhod; };

Par Lenoba(vector<Par>& sodelavci)
{
  const llint M = 12'000'000'000LL; // poldne

  // Pripravimo seznam sodelavcev, ki nas lahko ovirajo.
  vector<Par> v; for (auto &P : sodelavci)
    if (P.prihod < M && M < P.odhod) v.push_back(P);

  // Če takih sodelavcev ni, je problem trivialen.
  if (v.empty()) return {M - 1, M + 1};

  // Uredimo jih naraščajoče po času prihoda.
  sort(v.begin(), v.end(), [] (auto x, auto y) { return x.prihod < y.prihod; });
  v.push_back({M, M}); // stražar na koncu zaporedja

  // Preglejmo možnost, da pridemo pred vsemi sodelavci.
  llint odhod = M + 1;
  Par naj = {v[0].prihod - 1, odhod}; // najboljša rešitev doslej

  // Preglejmo še možnosti kasnejšega prihoda.
  for (int i = 0; i + 1 < v.size(); i++)
  {
    // Na naslednjem intervalu nas vidi priti tudi sodelavec i,
    // kar nas dodatno omejuje pri odhodu.
    odhod = max(v[i].odhod + 1, odhod);
```

```
    // Če več sodelavcev pride ob istem času, moramo upoštevati vse,
    // da dobimo pravi čas odhoda.
    if (v[i + 1].prihod == v[i].prihod) continue;

    // Sicer je pametno priti tik pred sodelavcem i + 1.
    llint prihod = v[i + 1].prihod − 1;

    // Če je to najboljša rešitev doslej, si jo zapomnimo.
    if (odhod − prihod < naj.odhod − naj.prihod) naj = {prihod, odhod};
  }
  return naj;
}
```

4. Semafor

Imeli bomo dve globalni spremenljivki: stevec, ki odšteva sekunde do spremembe luči, in prizgan, ki jo uporabljamo za izvedbo utripanja in nam pove, ali je bil prikazovalnik v prejšnji sekundi prižgan.

Naš podprogram VsakoSekundo ob vsakem klicu zmanjša števec za 1 (pri tem pazimo, da ne pade pod -1; vrednost -1 bomo uporabljali kot znak, da je čas do naslednje pričakovane spremembe luči že potekel in da mora biti prikazovalnik zdaj prazen) in, dokler je števec nad 99, vsako sekundo tudi obrne vrednost prizgan. Poseben primer je, ko dobimo novo pozitivno vrednost parametra n, takrat pa to shranimo v stevec.

Nato lahko popravimo stanje prikazovalnika takole: če je števec nad 99, mora prikazovalnik ali kazati število 99 ali pa biti prazen, odvisno od spremenljivke prizgan; sicer pa prikažemo kar vrednost števca samo (pri vrednosti -1 bo prikazovalnik prazen, kar je točno to, kar takrat tudi hočemo).

```
int stevec = 0;
bool prizgan = false;

void VsakoSekundo(int n)
{
  if (n >= 0) // Postavimo števec na n. Spremenljivko „prizgan“ uporabljamo le,
              // ko je števec večji od 99.
    stevec = n, prizgan = true;
  else {
    // Zmanjšajmo števec za 1 (če ni bil že pod 0, kar pomeni ugasnjen prikazovalnik).
    if (stevec >= 0) stevec−−;

    // Če je števec nad 99, poskrbimo za utripanje.
    if (stevec > 99) prizgan = ! prizgan; }

  // Če je števec nad 99, prikažimo 99 ali prazen prikazovalnik.
  if (stevec > 99) Prikazi(prizgan ? 99 : −1);

  // Sicer prikažimo trenutno vrednost števca.
  else Prikazi(stevec);
}
```

5. Prelom besedila

Recimo, da lomimo besedilo na vrstice dolžine največ w. Če se je neka vrstica začela z i-tim znakom vhodnega niza, so lahko v njej znaki največ do $(i + w - 1)$-vega; najkasneje pri znaku $i + w$ pa bo treba iti v novo vrstico. Toda ne nujno točno

pri njem; če je ta znak ne-prvi znak neke besede, bo novo vrstico treba začeti že na začetku te besede; če pa je $(i + w)$-ti znak presledek, bo treba novo vrstico začeti šele na začetku naslednje besede, saj naloga pravi, da presledki pri prelomu vrstice ostanejo v prejšnji vrstici, tudi če štrlijo čez rob okenca.

Koristno bi bilo torej za vsak znak vedeti, kje se začne beseda, ki ji ta znak pripada (če ni presledek), oz. kje se začne naslednja beseda (če je ta znak presledek). Temu podatku za znak i recimo N_i. Tega ni težko računati z dvema prehodoma po nizu. Najprej gremo od leve proti desni in računamo N_i za ne-presledke: če i ni presledek, potem, če je znak $i - 1$ presledek, je $N_i = i$, sicer pa je $N_i = N_{i-1}$. Nato gremo še od desne proti levi in ga računamo za presledke: če je znak i presledek, je $N_i = N_{i+1}$.

S pomočjo tabele N torej vemo, da če lomimo besedilo na vrstice dolžine največ w in se ena vrstica začne pri i, se mora naslednja začeti pri N_{i+w}. Tako lahko hitro skačemo od začetka ene vrstice do začetka naslednje, dokler ne pridemo do konca besedila. Sproti lahko vrstice še štejemo in rezultat na koncu zapišemo v zaporedje (vektor), ki ga na koncu vrnemo.

Poseben primer je, če je širina vrstice w ožja od dolžine najdaljše besede. Takrat se bo včasih zgodilo, da bo $N_{i+w} \leq i$; na to moramo paziti ne le zato, ker moramo takrat vrniti -1, pač pa tudi zato, ker bi se naš siceršnji algoritem za lomljenje vrstic (iz prejšnjega odstavka) zaciklal.

Oglejmo si implementacijo te rešitve v C++:

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

vector<int> PrelomBesedila(const string& s)
{
  int z = s.length();
  vector<int> rezultati(z), N(z);

  // Za vsak ne-presledek izračunajmo začetek njegove besede.
  for (int i = 0; i < z; i++) if (s[i] != ' ')
    N[i] = (i > 0 && s[i - 1] != ' ') ? N[i - 1] : i;

  // Za vsak presledek izračunajmo začetek naslednje besede.
  for (int i = z - 1; i >= 0; i--)
    if (s[i] == ' ') N[i] = (i == z - 1) ? z : N[i + 1];
    else i = N[i]; // skočimo na začetek trenutne besede

  // Za vsako širino izračunajmo število vrstic.
  for (int w = 1; w <= z; w++)
  {
    int stVrstic = 0, zacVrstice = N[0];
    while (zacVrstice < z) {

      // Določimo začetek naslednje vrstice.
      stVrstic++; int konVrstice = zacVrstice + w;
      if (konVrstice >= z) break;
      int zacNaslednje = N[konVrstice];

      // Pazimo na primer, ko je širina preozka za to besedo.
      if (zacNaslednje <= zacVrstice) { stVrstic = -1; break; }
      zacVrstice = zacNaslednje; }

    rezultati[w - 1] = stVrstic;
```

```
  }
  return rezultati;
}
```

Razmislimo še o časovni zahtevnosti te rešitve. Na začetku porabimo $O(z)$ časa za oba prehoda čez vhodni niz, s katerima izračunamo tabelo N. Pri lomljenju besedila imamo z vsako vrstico $O(1)$ dela, tako da za prelom pri širini w porabimo toliko časa, kolikor vrstic nastane, to pa je približno $O(z/w)$ (ker imamo niz dolžine z, v vsaki vrstici pa je prostora za w znakov).[3] Če to seštejemo po vseh w od 1 do z, dobimo $\sum_{w=1}^{z}(z/w) = zH_z$, kjer H_z pomeni z-to harmonično število; zanj velja $H_z \approx \ln z$, torej ima naš postopek časovno zahtevnost $O(z \ln z)$.

[3] Natančnejši razmislek: naj bo x_i število znakov v vrstici i (brez presledkov na koncu) in p_i število presledkov na koncu te vrstice. Če bi v dveh zaporednih vrsticah bilo, skupaj s presledki na koncu prve od teh dveh vrstic, le w ali manj znakov, bi ju lahko združili v eno. Imamo torej $x_i + p_i + x_{i+1} \geq w + 1$ za $i = 1, \ldots, h-1$. Če vse to seštejemo in upoštevamo, da je $\sum_{i=1}^{h}(x_i+p_i) = z$ (dolžina celotnega vhodnega niza), dobimo $2z - x_1 - x_h - 2p_h \geq (h-1)/(w+1)$, torej $h \leq 1 + (2z - x_1 - x_h - 2p_h)/(w+1) \leq 1 + 2z/w$.

REŠITVE NALOG ZA TRETJO SKUPINO

1. Vaje v slogu

Mislimo si neko konkretno dolžino d in recimo za začetek, da nas zanimajo Poldetove trojice iz podnizov te dolžine. Ker je vhodni niz dolg n znakov, se lahko podniz dolžine d začne na enem od indeksov $1, 2, \ldots, n-d+1$. Niso pa nujno vsi ti podnizi različni; recimo, da za vsak različen podniz pripravimo množico — pravzaprav bo še bolj koristno imeti urejen seznam — indeksov, na katerih se začenjajo pojavitve tega podniza. Vsak indeks od 1 do $n-d+1$ pripada natanko eni od teh množic, tako da te množice tvorijo razbitje (particijo) množice $\{1, \ldots, n-d+1\}$. Temu razbitju recimo R_d. Na primer, pri $s =$ `ababaccabab` (primer iz besedila naloge) za $d = 2$ dobimo $R_d = \{\{1, 3, 8, 10\}, \{2, 4, 9\}, \{5\}, \{6\}, \{7\}\}$. Podmnožice, ki tvorijo to razbitje, se nanašajo (po vrsti, kot so tu napisane) na podnize `ab`, `ba`, `ac`, `cc` in `ca`.

Takšno razbitje pride zelo prav pri iskanju ali štetju Poldetovih trojic. Ker tvorijo trojico trije enaki podnizi, se lahko ukvarjamo z vsako množico v razbitju posebej. Recimo torej, da gledamo množico $\{t_1, t_2, \ldots, t_\ell\}$, ki nam pove, da se neki podniz dolžine d pojavlja ℓ-krat v nizu s, in sicer se njegove pojavitve začenjajo na indeksih $t_1, t_2, \ldots, t_\ell$. Recimo še, da so ti indeksi urejeni naraščajoče, torej je $t_1 < t_2 < \ldots < t_\ell$.

Če nas zanima, ali sploh obstaja kakšna Poldetova trojica, temelječa na tem podnizu, moramo torej le preveriti, ali se kakšne tri od teh pojavitev ne prekrivajo. Če se sploh kakšni dve od njih ne prekrivata, sta to gotovo prva in zadnja; in če se res ne, moramo potem le še preveriti, ali se kakšna od vmesnih pojavitev ne prekriva z njima, torej ali za kak t_i (za $1 < i < \ell$) velja $t_1 + d \le t_i$ in $t_i + d \le t_\ell$. To nam vzame $O(\ell)$ časa, kjer je ℓ število pojavitev tega podniza; in ker imajo vsi podnizi dolžine d skupaj $n-d+1$ pojavitev, nam preverjanje, ali kakšna trojica pri tem d obstaja, vzame $O(n)$ časa (če imamo razbitje že pripravljeno).

Podobno, le za odtenek bolj zapleteno, je štetje trojic. Pri vsakem indeksu t_j se lahko vprašamo: če bi hoteli vzeti tisto pojavitev našega podniza, ki se začne na indeksu t_j, kot srednji člen neke Poldetove trojice, na koliko načinov bi si lahko izbrali levi člen t_i in desni člen t_k, ne da bi se prekrivala s t_j? Zanima nas torej, do katerega i velja $t_i + d \le t_j$ in od katerega k naprej velja $t_j + d \le t_k$. Ko počasi povečujemo j, se počasi povečujeta tudi največji primerni i in najmanjši primerni k, zato je dovolj, če gremo v zanki enkrat po zaporedju $t_1, \ldots, t_\ell$ in sproti povečujemo vse tri števce:

$N := 0$; (* *tu bomo izračunali število trojic* *)
$i := 0$; $k := 1$;
for $j := 1$ **to** ℓ:
  while $t_{i+1} + d \le t_j$ **do** $i := i + 1$;
  while $k \le n$ **and** $t_j + d \ge t_k$ **do** $k := k + 1$;
  (* *Če za srednji člen trojice vzamemo pojavitev, ki se začne na* t_j,
    se mora levi člen začeti na enem od indeksov $t_1, \ldots, t_i$,
    desni člen pa na enem od indeksov $t_k, \ldots, t_n$. *)
  $N := N + i \cdot (n - k + 1)$;

Na koncu tega postopka imamo v N število vseh trojic, temelječih na tistem podnizu, na katerega se nanaša množica $\{t_1, \ldots, t_\ell\}$. Pravzaprav naloga pravi, da bomo

morali izpisati ostanek po deljenju N z 1 000 037; lahko torej vsakič, ko v zadnji vrstici gornjega postopka povečamo N, obdržimo le ostanek po deljenju nove vrednosti z 1 000 037, lahko pa tudi računamo N v 64-bitni spremenljivki in ostanek po deljenju izračunamo šele na koncu, ob izpisu. (V nizu dolžine n je lahko največ $O(n^3)$ trojic, temelječih na podnizih dolžine d.)

Pri naši nalogi nas bo zanimal največji d, pri katerem sploh še obstaja kakšna Poldetova trojica. Opazimo lahko, da če obstaja neka trojica iz podnizov dolžine d, se v njej skrivajo tudi trojice iz podnizov dolžine $d-1$, $d-2$ in tako naprej. Zato lahko največji d poiščemo z bisekcijo: pri $d=0$ trojica gotovo obstaja (če si jo predstavljamo iz treh praznih nizov), pri $d=\lfloor n/3\rfloor+1$ gotovo ne obstaja (ker je niz s prekratek, da bi v njem sploh lahko bili trije neprekrivajoči se podnizi dolžine d), vmes pa bomo najmanjši d iskali z bisekcijo.

Razmisliti pa moramo še o tem, kako za nek konkreten d sploh pripraviti razbitje R_d. Pri $d=1$ je stvar enostavna; podnizi so tu kar posamezne črke, zato lahko za vsako črko abecede pripravimo po eno množico (pravzaprav urejen seznam) vseh indeksov, na katerih se pojavlja ta črka. Za daljše podnize lahko razmišljamo takole: recimo, da nas zanimajo podnizi dolžine d; zapišimo d kot vsoto dveh manjših dolžin, $d=d_1+d_2$. Potem lahko tudi vsak podniz t dolžine d zapišemo kot stik dveh krajših podnizov, $t=t_1t_2$, kjer je t_1 dolg d_1 znakov, t_2 pa d_2 znakov. Pojavitve podniza t se začenjajo natanko na tistih indeksih i, za katere velja, da se na i začenja tudi neka pojavitev podniza t_1 in da se poleg tega na $i+d_1$ začenja neka pojavitev podniza t_2. Začnemo lahko torej z množico indeksov, na katerih se pojavlja t_1 (ta množica je del razbitja R_{d_1}), in jo razdrobimo na manjše podmnožice tako, da za vsak indeks i v tej množici pogledamo, kateri podniz dolžine d_2 se pojavlja na indeksu $i+d_1$ (z drugimi besedami: pogledamo, kateri množici v razbitju R_{d_2} pripada indeks $i+d_1$). Tako nam množica pojavitev niza t_1 razpade na podmnožice, ki predstavljajo različne take podnize dolžine d, ki se začnejo na t_1. Ko naredimo to za vse množice iz R_{d_1}, nam sčasoma nastane razbitje R_d. Da lahko to počnemo učinkovito, je koristno imeti vsako razbitje predstavljeno ne le s seznamom urejenih seznamov (ki predstavljajo posamezne množice v razbitju), ampak tudi s tabelo, ki za vsak indeks od 1 do $n-d+1$ pove, kateri podmnožici v razbitju pripada; recimo tej tabeli $\tilde{R}_d$; potem moramo le za vsak indeks i vzeti par $(\tilde{R}_{d_1}[i], \tilde{R}_{d_2}[i+d_1])$ in kjer nastanejo enaki pari, dodamo pripadajoče i-je v isto podmnožico nastajajočega R_d. Za preverjanje, kje nastanejo enaki pari, lahko uporabimo razpršeno tabelo ali pa urejanje, npr. urejanje s štetjem, tako da bomo imeli z vsakim parom v povprečju le $O(1)$ dela in bomo za izračun R_d iz R_{d_1} in R_{d_2} porabili $O(n)$ časa. Drobna izboljšava, ki tudi ne škodi, je, da pri pripravi razbitij R_d podnize z manj kot tremi pojavitvami (torej množice z manj kot tremi elementi) kar sproti zavržemo, saj iz njih gotovo ne bo nastala nobena Poldetova trojica, pa tudi pri nadaljnjem drobljenju (ko bomo računali razbitja za večje d) bodo iz njih nastale prav tako premajhne podmnožice.[4]

S tem postopkom lahko iz R_1 dobimo R_2, nato R_4, R_8 in tako naprej za dolžine, ki so potence števila 2. Tudi pri bisekciji je zato koristno paziti na to, da je širina intervala d-jev, ki so pri bisekciji še aktualni, vedno potenca števila 2, tako da bomo lahko enostavno v $O(n)$ časa izračunali razbitje za dolžino na sredi intervala. Zato

[4] Posebej enostaven primer tega nastopi, če vzamemo $d_2=1$ (torej gledamo tam le posamezne črke) in naš zgoraj opisani postopek izračuna R_d iz R_{d-1}. S podobnim prijemom smo se že srečali pri nalogi 2012.3.5; gl. str. 66–7 v *Biltenu* 2012.

je koristno, če na začetku zgornjo mejo namesto na $\lfloor n/3 \rfloor + 1$, kot smo pisali zgoraj, postavimo na najmanjšo potenco števila 2, ki je večja ali enaka $\lfloor n/3 \rfloor + 1$. Bisekcija bo tako potrebovala $O(\log n)$ korakov, v vsakem pa porabimo $O(n)$ časa za izračun novega razbitja in za to, da ga pregledamo, če je v njem kaj trojic (oz. koliko jih je). Tudi izračun R_1, R_2, R_4, R_8 itd. na začetku nam vzame $O(n \log n)$ časa, tako da je časovna zahtevnost celotne rešitve $O(n \log n)$.

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

int n; string s; // Vhodni podatki.

struct Trojica { int i, j, k; };

struct Razbitje
{
  int d;                          // dolžina podnizov pri tem razbitju
  vector<vector<int>> m;          // množice
  vector<int> p;                  // pripadnost; x je element množice m[p[x]]

  void Init1() // Pripravi razbitje za d = 1.
  {
    d = 1; m.resize(26); p.resize(n);
    for (int i = 0; i < n; i++) {
      p[i] = s[i] - 'a'; m[p[i]].push_back(i); }
  }

  // Iz R1 in R2 izračuna razbitje za d = R1.d + R2.d.
  void Zdruzi(const Razbitje &R1, const Razbitje &R2)
  {
    d = R1.d + R2.d; p = vector<int>(n - d + 1, -1); m.clear(); m.reserve(n - d + 1);
    unordered_map<long long, int> h;
    for (int a1 = 0; a1 < R1.m.size(); a1++) if (R1.m[a1].size() >= 3)
      for (int i : R1.m[a1])
      {
        if (i + d > n) continue;
        int a2 = R2.p[i + R1.d]; if (a2 < 0 || R2.m[a2].size() < 3) continue;
        // Indeks i predstavimo s parom (R1.p[i], R2.p[i + R1.d]) — no, pravzaprav
        // kar s številom R1.p[i] * n + R2.p[i + R1.d]. Če ima več indeksov
        // enak par, predstavljajo enak podniz dolžine d in jih moramo torej
        // dodati v isti seznam znotraj m. Te pare hranimo kot ključe
        // v h, kot spremljevalno vrednost pa indeks ustreznega seznama v m.
        auto [it, nov] = h.emplace(a1 * (long long) n + a2, -1);
        if (nov) { it->second = m.size(); m.push_back({}); }
        p[i] = it->second; m[p[i]].push_back(i);
      }
    // Podnize z manj kot tremi pojavitvami lahko ignoriramo.
    for (auto &v : m) if (v.size() < 3) { for (int i : v) p[i] = -1;  v.clear(); }
  }

  Trojica PrimerTrojice() const // Vrne primer trojice, če ta obstaja.
  {
```

```
    for (const auto &v : m) { // Preglejmo vse podnize dolžine d.
      // Če ima podniz manj kot tri pojavitve, iz njega ne bo trojic.
      int vd = v.size(); if (vd < 3) continue;
      // Sicer uporabimo prvo in zadnjo pojavitev.
      int i = v[0], k = v[vd - 1];
      // Poglejmo, če je vmes še kakšna, ki se ne prekriva z njima.
      for (int j : v) if (i + d <= j) if (j + d <= k) return {i, j, k}; else break; }
    return {-1, -1, -1};
  }
};

int main()
{
  // Preberimo vhodne podatke.
  cin >> n >> s;
  // Pripravimo razbitja za d-je, ki so potence števila 2. Potrebujemo jih do
  // takega 2^g, dokler d = 1 + 2^g še vedno ustreza pogoju 3d <= n.
  int lgN3 = 0; while (3 * (1 + (1 << lgN3)) < n) lgN3++;
  vector<Razbitje> RP2(lgN3);
  RP2[0].Init1(); for (int i = 0; i < lgN3 - 1; i++) RP2[i + 1].Zdruzi(RP2[i], RP2[i]);
  // Z bisekcijo poiščimo največji d, pri katerem še obstaja kakšna trojica.
  int L = 1; Razbitje RL = RP2[0];
  for (int g = lgN3 - 1; g >= 0; --g)
  {
    // Tu velja, da obstaja trojica z d = L, ne obstaja pa taka z d = L + 2^(g+1).
    int M = L + (1 << g);
    Razbitje RM; RM.Zdruzi(RL, RP2[g]);
    if (RM.PrimerTrojice().i >= 0) { L = M; RL = move(RM); }
  }
  // Izpišimo eno od najdaljših trojic.
  auto T = RL.PrimerTrojice();
  cout << (T.i + 1) << " " << (T.j + 1) << " " << (T.k + 1) << " " << L << endl;
  // Preštejmo trojice dolžine L.
  int stTrojic = 0;
  for (const auto &v : RL.m)
  {
    int vd = v.size(), pi = 0, pk = 0; if (vd < 3) continue;
    for (int j : v)
    {
      while (pi < vd && v[pi] + L <= j) pi++;
      while (pk < vd && v[pk] < j + L) pk++;
      // Pojavitve, ki se začnejo na v[0], ..., v[pi - 1], ležijo v celoti
      // levo od tiste, ki se začne na j; tiste pa, ki se začnejo na
      // v[pk], ..., v[vd - 1], ležijo v celoti desno od nje.
      stTrojic = (stTrojic + pi * (long long) (vd - pk)) % 1000037;
    }
  }
  cout << stTrojic << endl; return 0;
}
```

Razmislimo zdaj še o različici naloge, ki jo omenja opomba pod črto v besedilu naloge: v definicijo Poldetove trojice torej dodajmo omejitev, da sme biti med po-

javitvami treh podnizov, ki tvorijo trojico, kvečjemu g drugih znakov. Pri prvotni definiciji smo videli, da se v Poldetovi trojici, ki jo tvorijo trije podnizi dolžine d, vedno skrivajo tudi krajše trojice s podnizi dolžin $1, 2, \ldots, d-1$, pri novi definiciji pa to ne drži nujno, saj se lahko zgodi, da je med pojavitvami teh krajših podnizov zdaj že preveč (več kot g) drugih znakov. Na primer: pri $g = 2$ tvorijo tri pojavitve podniza `ab` v nizu `abcabdcab` Poldetovo trojico, tri pojavitve podniza `a` pa ne (ker je med njimi kar 5 drugih znakov: `bc` in `bdc`), prav tako pa tudi ne tri pojavitve podniza `b` (tudi med temi je kar 5 drugih znakov: `ca` in `dca`).

To pomeni, da najdaljšega d, pri katerem še obstaja kakšna Poldetova trojica, zdaj ne moremo več iskati z bisekcijo, saj se je ta postopek opiral na predpostavko, da obstajajo trojice pri *vseh* d od 0 do neke maksimalne dolžine, zdaj pa to ni več nujno res. Tudi to, da pri nekem d nismo našli nobene trojice, zdaj ne pomeni, da je ne bomo našli pri kakšnem večjem d. Pri novi različici naloge torej ne bo šlo drugače, kot da pregledamo vse možne d. Razbitje R_d lahko izračunamo iz R_{d-1} in R_1 enako kot pri prvotni različici naloge. Ustavimo se šele pri tistem d, ko nobena množica v R_d nima vsaj treh elementov; takrat vemo, da se noben podniz te dolžine ne pojavi vsaj trikrat, torej Poldetovih trojic te (ali večje) dolžine gotovo ne more biti.

Kako lahko zdaj pri znanem razbitju R_d preštejemo, koliko je Poldetovih trojic s tem d? Podobno kot pri prvotni različici naloge bomo gledali vsako množico razbitja posebej in indekse v njej uredili; recimo, da imamo pred seboj seznam indeksov $t_1, \ldots, t_\ell$ (urejen naraščajoče), ki predstavljajo vse pojavitve nekega podniza dolžine d v vhodnem nizu s.

Recimo, da gledamo trojice z levim členom t_i. Za desni člen t_k mora potem veljati $t_k \geq t_i + 2d$, saj imata levi in srednji člen trojice skupaj $2d$ znakov in bi se desni člen, če bi se začel prej kot pri $t_i + 2d$, gotovo prekrival s srednjim; po drugi strani pa mora veljati tudi $t_k \leq t_i + 2d + g$, saj bi sicer bilo med temi tremi členi gotovo več kot g znakov in to potem ne bi bila Poldetova trojica. Naj bo u najmanjši tak indeks, pri katerem je $t_u \geq t_i + 2d$, v pa največji tak indeks, pri katerem je $t_v \leq t_i + 2d + r$. Možni k-ji so torej (pri izbranem i) le $u \leq k \leq v$ (ta interval je lahko tudi prazen; tedaj pač ni pri trenutnem d nobene Poldetove trojice z levim členom t_i). Za vsakega od njih nas zdaj zanima, koliko je pri njem primernih položajev srednjega člena, recimo t_j.

V poštev pridejo vsi j z območja $i < j < k$ (teh je $k - i - 1$) razen tistih, pri katerih bi se srednji člen prekrival z levim ali z desnim. Z levim se prekrivajo tisti, pri katerih je $t_j < t_i + d$, z desnim pa tisti, pri katerih je $t_k < t_j + d$. (Ne more pa se zgoditi, da bi se kak srednji člen prekrival tako z levim kot z desnim, saj smo si k izbrali tako, da je $t_k \geq t_i + 2d$, zato sta si levi in desni člen predaleč narazen, da bi se lahko srednji prekrival z obema.) Naj bo L_k število pojavitev, ki se začnejo levo od t_k in se prekrivajo s tisto na t_k (torej se začnejo nekje od $t_k - d + 1$ do $t_k - 1$) in naj bo D_i število pojavitev, ki se začnejo desno od t_i in se prekrivajo s tisto na t_i (torej se začnejo nekje od $t_i + 1$ do $t_i + d - 1$). Če si za desni člen izberemo t_k, je torej možnih izborov srednjega člena $(k - i - 1) - D_i - L_k$.

To moramo zdaj sešteti po vseh primernih k (od u do v), pa bomo dobili število vseh trojic z levim členom i; to je torej $\sum_{k=u}^{v} ((k - i - 1) - D_i - L_k) = (v - u + 1)((u + v)/2 - i - 1 - D_i) - \sum_{k=u}^{v} L_k$. Vsoto $\sum_{k=u}^{v} L_k$ bomo lahko računali zelo

poceni, če si vnaprej pripravimo delne vsote zaporedja L_k, torej $L'_k := L_1 + \ldots + L_k$; potem je $\sum_{k=u}^{v} L_k = L'_v - L'_{u-1}$.

Vse možne i lahko pregledujemo v zanki; ko se i poveča za 1, ni težko popraviti tudi števcev u in v (tadva se lahko ali povečata ali pa ostaneta nespremenjena). Zapišimo dobljeni postopek s psevdokodo:

algoritem PreštejTrojice:
vhod: urejen seznam $t_1, t_2, \ldots, t_\ell$;
(* *Izračunajmo tabele L, D in L'.* *)
$i := 1$; $k := 1$; $L'_0 := 0$;
for $j := 1$ **to** ℓ:
  while $r_i + d \le r_j$ **do** $i := i + 1$;
  while $k < \ell$ **and** $r_{k+1} < t_j + d$ **do** $k := k + 1$;
  (* *S pojavitvijo na t_j se prekrivajo tiste na $t_i, \ldots, t_{j-1}$ levo od nje in tiste na $t_{j+1}, \ldots, t_k$ desno od nje.* *)
  $L_j := j - i$; $D_j := k - j$; $L'_j := L'_j + L_j$;
(* *Preštejmo zdaj vse Poldetove trojice.* *)
$N := 0$; (* *tu bomo izračunali število trojic* *)
$u := 1$; $v := 1$;
for $i := 1$ **to** $\ell - 2$:
  while $u \le \ell$ **and** $t_u < t_i + 2d$ **do** $u := u + 1$;
  if $u > \ell$ **then break**; (* *preostale pojavitve so preblizu trenutne* *)
  while $v + 1 \le \ell$ **and** $t_{v+1} \le t_i + 2d + g$ **do** $v := v + 1$;
  if $v < u$ **then continue**; (* *pri tem i ni možna nobena trojica* *)
  (* *Če je levi člen trojice t_i, je lahko desni člen le eden od $t_u, \ldots, t_v$.* *)
  $N := N + (v - u + 1)((u + v)/2 - i - 1 - D_i) - (L'_v - L'_{u-1})$;
return N;

Časovna zahtevnost tega postopka je $O(\ell)$, zato lahko vse množice v razbitju R_d pregledamo v skupno $O(n)$ časa. Prav toliko časa potrebujemo tudi za to, da iz R_{d-1} izračunamo R_d. Ker zdaj ne moremo uporabiti bisekcije, ampak moramo pregledati vse d-je po vrsti, bo časovna zahtevnost celotne rešitve zdaj $O(n^2)$.

2. Zamik

Najprej opomba o notaciji: v tej rešitvi bomo zapis, ki spominja na potence, uporabljali za zaporedja več enakih znakov; tako na primer $\mathtt{0}^a\mathtt{1}^b$ pomeni niz, v katerem je naprej a zaporednih ničel in nato b zaporednih enic; kot običajno, če eksponent manjka, si mislimo, da je enak 1.

Recimo zdaj za začetek, da bi pripravili niz samih ničel, le z eno enico na koncu: $\mathtt{0}^{n-1}\mathtt{1}$. Hitro lahko opazimo, da s takim nizom ne moremo ugotoviti, ali ga je sistem prezrcalil ali ne: če ga zamakne za eno mesto v desno in ga ne prezrcali, je rezultat enak, kot če ga zamakne za 0 mest in ga prezrcali. Na enako težavo naletimo, če imamo eno daljšo skupino enic, $\mathtt{0}^a\mathtt{1}^b$; ali pa če imamo dve enako dolgi skupini enic, ločeni z ničlo: $\mathtt{0}^a\mathtt{1}^b\mathtt{0}\mathtt{1}^b$.

Šele če imamo dve različno dolgi skupini enic, lahko zaznamo zrcaljenje: če imamo na primer niz $\mathtt{0}^{n-4}\mathtt{1011}$, lahko zrcaljenje prepoznamo po tem, da bo po

njem v nizu nastopal podniz 1101 namesto 1011; zamik pa lahko določimo iz tega, kje v nizu zdaj eden ali drugi od teh dveh podnizov stoji.

Naslednja težava je, da moramo znati te enice v nizu najti z zelo malo poizvedbami, to pa bo težko, če predstavljajo tako kratek del celotnega niza. Tako nam lahko pride na misel, da bi imeli dve daljši skupini enic, ločeni z ničlo: $0^a1^b01^d$, pri čemer naj bo $b \neq d$. Toda ko bomo s poizvedbami v zamaknjenem (in mogoče prezrcaljenem) nizu našli enice, bomo morali najti tudi tisto ničlo med obema skupinama enic; šele iz položaja te ničle (relativno glede na položaj enic) bomo lahko vedeli, da imamo v novem nizu 1^b pred 1^d (in torej niz ni bil prezrcaljen) ali 1^d pred 1^b (in je torej niz bil prezrcaljen). Najti eno samcato ničlo v dolgem zaporedju enic pa bo spet težko, zato je bolje, če obe skupini enic tudi ločimo z neko daljšo skupino ničel.

Tako smo pri nizu oblike $0^a1^b0^c1^d$ z $b \neq d$. Paziti pa moramo še na to, da morata biti različno dolgi ne le obe skupini enic, ampak tudi obe skupini ničel: kajti če bi bilo $a = c$, potem (zaradi cikličnega značaja našega niza pri zamikanju) ne bi mogli reči, ali leži 1^b levo ali desno od 1^d, torej spet ne bi mogli zaznati, ali je bil niz prezrcaljen ali ne. Dodati moramo torej še pogoj $a \neq c$, ne samo $b \neq d$.

Znotraj teh omejitev je koristno, če so vsi štirje deli približno enako dolgi, torej okrog $n/4$. Ker n ni nujno večkratnik 4, pišimo $n = 4k + r$; vzemimo za začetek $a = b = k - 1$ in $c = d = k + 1$, nato pa prvih r izmed števil a, b, c povečajmo za 1. Tako dobimo niz naslednje oblike:

n	a	b	c	d	niz s
$n = 4k$	$k-1$	$k-1$	$k+1$	$k+1$	$0^{k-1}1^{k-1}0^{k+1}1^{k+1}$
$n = 4k+1$	k	$k-1$	$k+1$	$k+1$	$0^k1^{k-1}0^{k+1}1^{k+1}$
$n = 4k+2$	k	k	$k+1$	$k+1$	$0^k1^k0^{k+1}1^{k+1}$
$n = 4k+3$	k	k	$k+2$	$k+1$	$0^k1^k0^{k+2}1^{k+1}$

Da se ta načrt obnese, nobena od skupin ničel ali enic ne sme biti prazna, sicer niz ne bo prave oblike. Pri $n = 4k$ in $4k+1$ je najkrajša skupina dolga $k-1$, torej mora biti $k \geq 2$, torej $n \geq 8$; pri $n = 4k + 2$ in $4k + 3$ pa je najkrajša skupina dolga k, zato je dovolj že $k \geq 1$, torej $n \geq 6$. Tako torej vidimo, da naš pristop deluje za vse $n \geq 6$, to pa so ravno vsi, ki lahko nastopijo v naši nalogi.[5]

Razmislimo zdaj o tem, kako točno pri takšnem nizu s čim manj poizvedbami določiti zamik in zrcaljenje. Naš cilj bo, da za dve sosednji skupini znakov, torej eno skupino ničel in eno skupino enic, točno določimo njuno dolžino in položaj. Ko namreč enkrat poznamo to, lahko določimo tudi zamik in ali je prišlo do zrcaljenja. Na primer, če najdemo skupino 0^c in desno od nje skupino 1^b, potem vemo, da je

[5] Hitro se lahko prepričamo, da pri manjših n problem sploh ni rešljiv. Če hočemo rešiti uganko ne glede na to, kakšen zamik si sistem izbere in ali prezrcali naš niz, mu moramo podati tak niz, pri katerem s temi $2n$ kombinacijami zamika in morebitnega zrcaljenja nastane $2n$ različnih nizov. Jasno pa je, da zamik in zrcaljenje nič ne spremenita števila enic in ničel v nizu. Recimo, da ima naš niz k enic. Takih nizov dolžine n je $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Če je $\binom{n}{k} < 2n$, je naloga gotovo nerešljiva, saj sploh ne obstaja dovolj različnih nizov dolžine n s k enicami. Hitro se lahko prepričamo, da do $n = 5$ to velja povsod (za $0 \leq k \leq n \leq 5$), razen pri $n = 5$ in $k = 2$ ali 3, kjer je $\binom{5}{2} = \binom{5}{3} = 10$. Tam bi bil problem rešljiv, če bi bilo vseh tistih 10 nizov dolžine 5 s po dvema enicama (in podobno za nize s po tremi enicami) takih, da lahko nastanejo z zamikanjem in zrcaljenjem enega niza. Toda jasno je, da to ni res, saj imajo nekateri eno skupino enic (npr. 00110), nekateri pa dve ločeni (npr. 01010), taki nizi pa ne morejo nastati eden iz drugega z zrcaljenjem in zamikanjem.

bil niz prezrcaljen, saj bi drugače bila 1^b levo od 0^c; nato lahko izračunamo, kje sta bili tidve skupini pred zrcaljenjem; iz tega pa lahko, ko njun položaj primerjamo s tistim v začetnem nizu, izračunamo tudi zamik.

Označimo položaje v našem nizu z indeksi od 0 do $n-1$. S prvo poizvedbo poglejmo, ali je na indeksu 0 ničla ali enica. Recimo, da je ničla; za enico bi bil razmislek zelo podoben. Zanimivo bi bilo vedeti, kje je prva enica desno od te ničle; potem bi vedeli, da se tam začenja neka skupina enic, in bi lahko s še eno poizvedbo ugotovili, ali je dolga b enic ali d enic; nato bi pa s še eno poizvedbo ugotovili, ali za njo pride skupina a ničel ali c ničel; potem pa imamo, kot smo ugotovili v prejšnjem odstavku, že vse, kar potrebujemo, da rešimo uganko.

Vprašanje je torej zdaj, koliko ničel še pride desno od tiste na indeksu 0, preden nastopi prva enica. Lahko ni nobene (in je enica že na indeksu 1), lahko pa jih je največ $c-1$ (če se je na indeksu 0 ravno začela skupina c ničel — spomnimo se, da je to daljša od obeh skupin ničel, saj je $a < c$). Med temi c možnostmi lahko pravo poiščemo z bisekcijo, za kar porabimo $\lceil \log_2 c \rceil$ poizvedb. Če prištejemo še eno poizvedbo na začetku (na indeksu 0) in dve na koncu (da določimo dolžino naslednje skupine enic in naslednje skupine ničel), imamo skupaj $3 + \lceil \log_2 c \rceil = \lceil \log_2 8c \rceil$ poizvedb.

Ali je to že najboljša rešitev? Skoraj, ne pa še čisto. Pri nizu dolžine n obstaja $2n$ možnosti glede tega, kaj lahko sistem naredi — n zamikov brez zrcaljenja in še n zamikov, ki jim sledi zrcaljenje. Ker lahko z vsako poizvedbo ločimo med dvema možnostma, lahko z m poizvedbami ločimo med 2^m možnostmi. Če naj se dá z m poizvedbami rešiti našo uganko, mora torej veljati $2^m \geq 2n$, torej je najmanjši primerni m enak $\lceil \log_2 2n \rceil$. Naša rešitev ima $\lceil \log_2 8c \rceil$ poizvedb, kar je, če upoštevamo, da je $c \approx n/4$, res približno enako $\lceil \log_2 2n \rceil$. Toda v resnici je c vedno malo večji od $n/4$, zato se včasih lahko zgodi, da je $\lceil \log_2 8c \rceil > \lceil \log_2 2n \rceil$. Do tega pride, če je na območju od $2n$ do $8c-1$ kakšna potenca števila 2, recimo 2^{t+1}; takrat je $\lceil \log_2 2n \rceil = t+1$ in $\lceil \log_2 8c \rceil = t+2$.

Imamo torej pogoj $2n \leq 2^{t+1} < 8c$, kar lahko predelamo v $n \leq 2^t < 4c$. Če je $r = 0, 1$ ali 2, je $c = k+1$ in naš pogoj je naprej enak $4k + r \leq 2^t < 4k+4$; edina potenca števila 2 na območju od $4k+r$ do $4k+3$ je lahko $4k$, pa še ta je dosegljiv le, če je $r = 0$; in takrat je $4k = n$. Pri $r = 3$ pa je $c = k+2$ in naš pogoj je naprej enak $4k+3 \leq 2^t < 4k+8$, edina potenca števila 2 na območju od $4k+3$ do $4k+7$ pa je lahko $4k+4 = n+1$.

Tako torej vidimo, da do neugodne situacije, ko je $\lceil \log_2 8c \rceil > \lceil \log_2 2n \rceil$, pride le tedaj, ko je n oblike 2^t ali $2^t - 1$. Hitro se lahko prepričamo, da je takrat $r = 0$ ali $r = 3$, v vsakem primeru pa je zato $c = 2^{t-2} + 1$ in $a = c-2$. Razmislimo o tem, kako lahko v takih primerih število potrebnih poizvedb zmanjšamo za 1. Ko smo ugotavljali, kako dolga je skupina ničel na začetku niza (po tistem, ko ga je sistem zamaknil in mogoče prezrcalil), smo rekli, da moramo z bisekcijo ločiti med c različnimi možnostmi, zato potrebujemo $\lceil \log_2 c \rceil$ poizvedb, kar je naprej enako $\lceil \log_2(2^{t-2}+1) \rceil = t-1$. Teh c možnosti se nanaša na to, ali je desno od začetne ničle še $0, 1, \ldots, c-2$ ali $c-1$ ničel; zadnji dve lahko v mislih združimo v eno samo, „vsaj $c-2$ ničel“. Zdaj imamo le $c-1$ možnosti in bisekcija med njimi nam vzame le $\lceil \log_2(c-1) \rceil = \lceil \log_2 2^{t-2} \rceil = t-2$ poizvedb, torej eno manj kot prej. Če se izkaže, da je teh ničel od 0 do $c-3$, nadaljujemo enako kot v prvotni različici rešitve. Če

pa se izkaže, da je teh ničel vsaj $c-2$, potem že vemo, da imamo tukaj opravka s skupino 0^c in ne 0^a, le tega še ne vemo, kje točno se začne: ali na indeksu 0 (in je potem desno od njega še $c-1$ ničel) ali na indeksu $n-1$ (in se od tam nadaljuje na indeksu 0, desno od tega pa je še $c-2$ ničel). Med tema dvema možnostma lahko ločimo s še eno poizvedbo. Zdaj za eno skupino števk že poznamo njen položaj in dolžino, tako da potrebujemo le še eno poizvedbo, da preverimo, ali desno od nje stoji skupina 1^b ali 1^d, potem pa bomo že vedeli dovolj, da bomo določili zamik in zrcaljenje. Tako smo porabili $t-2$ poizvedb za bisekcijo, eno pred njo in dve za njo, skupaj $t+1$, kar je ravno enako $\lceil \log_2 2n \rceil$, tako kot smo želeli.

Zdaj imamo torej rešitev, ki v vsakem primeru porabi največ $\lceil \log_2 2n \rceil$ poizvedb. Manj od tega, kot smo videli že zgoraj, niti ni razumno pričakovati. Besedilo naloge pravi, da gre lahko n do 1024 in da smemo porabiti največ 11 poizvedb, če hočemo vse točke, to pa je ravno $\log_2(2 \cdot 1024)$, tako da naša rešitev tudi v tem smislu ustreza pričakovanjem naloge.

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

int Poizvedba(int i) { // Pošlje sistemu poizvedbo in vrne njegov odgovor.
  cout << (i + 1) << endl;
  int odgovor; cin >> odgovor; return odgovor; }

int main()
{
  while (true)
  {
    // Preberimo n.
    int n; cin >> n; if (n < 0) return 0;

    // Pripravimo niz.
    int a[4]; for (int i = 0; i < 4; i++) a[i] = (n / 4) - 1 + (i & 2) + (i < (n % 4) ? 1 : 0);
    if (a[0] == a[2]) a[0]--, a[2]++;
    if (a[1] == a[3]) a[1]--, a[3]++;
    cout << string(a[0], '0') << string(a[1], '1')
         << string(a[2], '0') << string(a[3], '1') << endl;

    // Poglejmo, s kakšnimi vrednostmi se zamaknjeni niz začne in do kod segajo.
    // Naj bo b0 vrednost bita na indeksu 0 in recimo, da se ista vrednost pojavlja na
    // indeksih 0, ..., d0' - 1, na indeksu d0' pa je vrednost že drugačna.
    // Namesto d0' bomo v resnici računali d0 = min(d0', a[b0] + 1).
    int b0 = Poizvedba(0);
    int L = 0, D = a[b0] + 1;
    while (D - L > 1)
      // Zdaj vemo, da je L < d0 ≤ D.
      if (int M = (L + D) / 2; Poizvedba(M) == b0) L = M; else D = M;
    int d0 = D;

    // Če smo dobili d0 > a[b0], je to gotovo blok dolžine a[b0 + 2] in ne a[b0].
    int z1, d1, b1; // z = začetek, d = dolžina, b = vrednost bitov v tem bloku
    if (d0 > a[b0])
    {
      // Če se dolžini a[b0] in a[b0 + 2] razlikujeta za 2 (in ne le za 1),
      // imamo zdaj d0 = a[b0] + 1, blok pa je dolg a[b0] + 2 in zato še ne vemo,
      // ali se začne na indeksu 0 ali na indeksu n - 1.
```

```
      if (a[b0 + 2] == a[b0] + 2) if (Poizvedba(d0) == b0) d0++;
      d1 = a[b0 + 2]; z1 = (d0 − d1 + n) % n; b1 = b0;
    }
    else
    {
      // Sicer imamo bite z vrednostjo b0 na indeksih od 0 do d0 − 1,
      // pri d0 pa se začne nov blok. Poglejmo, kako dolg je.
      z1 = d0; b1 = 1 − b0;
      d1 = (Poizvedba(z1 + a[b1]) == b1) ? a[b1 + 2] : a[b1];
    }
    // Zdaj en blok poznamo; poglejmo, kako dolg je naslednji.
    int z2 = (z1 + d1) % n, b2 = 1 − b1;
    int d2 = (Poizvedba(z2 + a[b2]) == b2) ? a[b2 + 2] : a[b2];
    // Ali je bil niz prezrcaljen?
    int i1 = b1 + (d1 == a[b1] ? 0 : 2), i2 = b2 + (d2 == a[b2] ? 0 : 2);
    bool prezrcaljen = (i2 == (i1 + 3) % 4);
    // Če da, poglejmo, kakšen bi bil položaj teh dveh blokov brez zrcaljenja.
    if (prezrcaljen) { z1 = n − 1 − (z1 + d1 − 1) % n; z2 = n − 1 − (z2 + d2 − 1) % n;
      swap(z1, z2); swap(d1, d2); swap(b1, b2); swap(i1, i2); }
    // Kje se je blok i1 začel pred zamikom?
    int p1 = 0; for (int i = 0; i < i1; i++) p1 += a[i];
    int zamik = (z1 − p1 + n) % n;
    // Pošljimo svoj odgovor.
    cout << zamik << " " << (prezrcaljen ? 1 : 0) << endl;
  }
}
```

3. Zlaganje slik

Naloga je primerna za reševanje z dinamičnim programiranjem. Naj bo $f(k)$ najmanjša skupna višina vrstic, v katere je moč zložiti prvih k slik. Ko računamo to funkcijo, lahko razmišljamo takole: v zadnjo od teh vrstic bomo zložili zadnjih nekaj slik, recimo od j do k; potem nam ostane še problem, kako čim bolje zložiti v vrstice prvih $j-1$ slik, za to pa že vemo, da bo skupna višina teh vrstic $f(j-1)$. Ker ne moremo vnaprej vedeti, koliko slik je pametno dati v zadnjo vrstico, bomo morali preizkusiti vse možne j in med tako dobljenimi rešitvami obdržati najboljšo. Dobimo torej zvezo $f(k) = \min_j\{f(j-1)+v(j,k)\}$, kjer $v(j,k)$ predstavlja najnižjo možno višino vrstice, ki jo tvorijo slike od j do k. Z j moramo iti tako daleč navzdol (nazaj po zaporedju slik), dokler je še mogoče zlagati slike od j do k v eno vrstico; ko pa postanejo preširoke za v eno vrstico (ne glede na to, kako jih obračamo), se ustavimo. Funkcijo f bomo računali po naraščajočih k-jih, pri vsakem bomo šli v notranji zanki po padajočih j-jih, ko pa enkrat izračunamo $f(k)$, si to vrednost zapomnimo v neki tabeli, da nam bo kasneje pri roki, ko jo bomo potrebovali.

Vprašanje je zdaj še, kako računati $v(j,k)$, torej najmanjšo možno višino vrstice, v kateri so slike od j do k. Naloga pravi, da so pri polovici testnih primerov vse slike kvadratne; takrat torej ni nobene koristi od tega, da jih obračamo za 90 stopinj, in višina vrstice je preprosto enaka najdaljši stranici vseh slik v njej: $v(j,k) = \max\{h_i : j \le i \le k\}$. Ta maksimum lahko računamo sproti, ko zmanjšujemo j (pri fiksnem k): $v(j,k) = \max\{v(j+1,k), h_j\}$. Tako imamo pri vsakem j le $O(1)$ dela in časovna zahtevnost naše rešitve je v tej obliki $O(n^2)$.

Nekaj več dela pa je s primeri, ko so slike lahko poljubni pravokotniki. Pri i-ti sliki označimo krajšo od obeh stranic z a_i, daljšo pa z b_i. Rekli bomo, da slika *stoji*, če je obrnjena tako, da ima višino b_i in širino a_i, in da *leži*, če ima širino a_i in višino b_i. Kako naj zdaj obrnemo slike od j do k, da bo višina vrstice, v kateri so te slike, čim manjša? Najbolje seveda je, če vse slike ležijo; takrat bo višina najmanjša, vendar bo tudi širina največja, tako da si lahko to privoščimo le, dokler širina takega razporeda ne preseže s.

Ko dodajamo vse več slik v vrstico, utegnemo sčasoma priti v položaj, ko ne morejo več vse ležati, ampak jih moramo nekaj postaviti stoje. Če postavimo sliko i stoje, bo vrstica zaradi tega visoka vsaj b_i. Če je pri neki drugi sliki ℓ daljša stranica krajša od tega, torej $b_\ell \le b_i$, potem se višina vrstice ne bo nič povečala, če postavimo pokonci tudi sliko ℓ, pač pa se bo širina vrstice zaradi tega zmanjšala (ker bo slika ℓ potem v širino merila le a_ℓ namesto b_ℓ). Vidimo torej, da če postavimo pokonci eno sliko, se splača postaviti pokonci tudi vse, katerih daljša stranica je krajša od njene (ali pa enaka njeni). Lahko si predstavljamo, da imamo neko zgornjo mejo v, ki nam pove, da bodo vse slike z $b_i \le v$ stale, vse z $b_i > v$ pa ležale. Uporabiti pa hočemo seveda najmanjši tak v, pri katerem širina tako nastale vrstice še ne preseže največje dovoljene širine s.

Kaj se zgodi, ko gremo z j na $j-1$, torej poskušamo v vrstico spraviti še eno novo sliko več poleg vseh dosedanjih (ki jih imamo še vedno v vrstici)? Ali je mogoče, da bi bil zdaj dovolj dober že kakšen manjši v kot prej? Gotovo ne, kajti pri tem nižjem v bi že slike $j, \ldots, k$ presegle širino s, torej bo ta širina presežena tudi, ko jim bomo dodali še sliko $j-1$, pa ne glede na to, ali bo ta slika pri tem v stala ali ležala. Ko se torej j zmanjšuje in prihajajo v vrstico nove slike, se lahko v (torej višina, do katere slike še stojijo) le povečuje ali pa ostaja enak, ne more pa se zmanjševati. Ko sliko enkrat postavimo pokonci, je (pri kasnejšem zmanjševanju j-ja) ne bomo nikoli več vrnili v ležeči položaj.

To pa pomeni, da nam ni več treba hraniti podrobnih podatkov o slikah, ki že stojijo; vse, kar nas še zanima o njih, bo zajeto v podatek o skupni širini trenutne vrstice. Slike, ki še vedno ležijo, pa je koristno hraniti urejene po b_i. Ko se j zmanjša in pride v vrstico nova slika, pogledamo, če je $b_j \le v$, da vidimo, če jo moramo dodati med ležeče ali med stoječe; nato pa pogledamo, če je vrstica zdaj preširoka, in počasi spreminjamo ležeče slike v stoječe (pri čemer jih gledamo naraščajoče po b_i), dokler njena širina ne pade na s ali manj. Primerna podatkovna struktura za to je kopica, v kateri naj bodo ležeče slike urejene tako, da je v korenu tista z najmanjšo b_i; pri vsaki sliki pa kot spremljevalni podatek hranimo še razliko $b_i - a_i$, ki nam pove, za koliko se vrstica zoži, če to sliko premaknemo iz ležečega položaja v stoječega.

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

typedef long long int llint;

int main()
{
```

```
  // Preberimo vhodne podatke.
  int n; lint s; scanf("%d %lld", &n, &s);
  vector<int> a(n), b(n);
  for (int i = 0; i < n; i++) {
    int w, h; scanf("%d %d", &w, &h);
    a[i] = min(w, h); b[i] = max(w, h); }
  // Rešimo problem po naraščajočem številu slik.
  vector<lint> f(n + 1); f[0] = 0; lint INF = 1;
  for (int k = 0; k < n; k++)
  {
    // Poiščimo najboljšo rešitev za slike 0, ..., k.
    // V zadnjo vrstico bomo poskusili dati slike j, ..., k za različne j.
    lint &naj = f[k + 1] = INF += b[k], sirina = 0; int visina = 0;
    priority_queue<pair<int, int>> lezece;
    for (int j = k; j >= 0; j--)
    {
      // Dodajmo sliko j med ležeče ali stoječe, odvisno od daljše stranice.
      if (b[j] <= visina) sirina += a[j];
      else visina = max(visina, a[j]), lezece.emplace(-b[j], b[j] - a[j]), sirina += b[j];
      // Poglejmo, če je treba kaj ležečih slik postaviti pokonci.
      while (sirina > s && ! lezece.empty()) {
        auto p = lezece.top(); visina = max(visina, -p.first);
        sirina -= p.second; lezece.pop(); }
      // Mogoče so slike že preširoke za eno vrstico, četudi vse stojijo.
      if (sirina > s) break;
      // Če je to najboljša rešitev doslej, si jo zapomnimo.
      naj = min(naj, visina + f[j]);
    }
  }
  printf("%lld\n", f[n]); return 0; // Izpišimo rezultat.
}
```

Pri dodajanju v kopico smo uporabili $-b_i$ namesto b_i, ker C++ov razred priority_queue v korenu kopice hrani največji element namesto najmanjšega; to bi lahko rešili tudi tako, da bi kot tretji template argument podali greater<pair<int, int>> ali kaj podobnega. Vrednost INF uporabljamo za inicializacijo vrednosti f[k + 1], ker je večja od največje možne skupne višine vseh slik od 0 do k in zato tudi večja od prave vrednosti najboljše rešitve za prvih $k + 1$ slik.

4. Janko in Metka

Razdelimo v mislih sladkarije na štiri množice, A, B, C in D, pri čemer so v A tiste, ki so všeč tako Janku kot Metki, v B tiste, ki so všeč le Janku, v C tiste, ki so všeč le Metki, in v D tiste, ki niso všeč nobenemu od njiju.

Naloga pravi, da moramo izbrati k sladkarij, od tega vsaj x takih, ki so všeč Janku, in vsaj x takih, ki so všeč Metki. Z vidika teh omejitev je vseeno, *katere* sladkarije iz množice A izberemo, pomembno je le, *koliko* jih izberemo; in enako seveda tudi pri B, C in D. Ker pa želimo izbor s čim večjo vsoto vrednosti, je smiselno, da vsako od teh štirih množic uredimo padajoče po vrednosti in iz vsake izberemo prvih nekaj sladkarij (torej tistih z najvišjo vrednostjo).

Recimo zdaj, da iz množice A izberemo natanko p sladkarij (seveda tistih z najvišjo vrednostjo). Te so všeč tako Janku kot Metki; toda če je $p < x$, moramo

izbrati nujno vsaj še $x-p$ takih, ki so všeč samo Janku, in $x-p$ takih, ki so všeč samo Metki, sicer naš izbor ne bo ustrezal zahtevam naloge. Pišimo $q := \max\{0, x-p\}$; dodajmo torej v naš izbor najvrednejših q sladkarij iz B in najvrednejših q iz C. (Lahko se izkaže, da v kakšni od množic B in C sploh ni q sladkarij; to pomeni, da smo vzeli premajhen p in z njim do veljavnega izbora sploh ni mogoče priti. Po drugi strani se lahko izkaže, da smo izbrali že preveč sladkarij, torej da je $p+2q > k$; tudi to je znak, da je naš sedanji p premajhen.)

Tako nam ostane še $|B| - q$ sladkarij iz B, pa $|C| - q$ sladkarij iz C in vse iz D. Tej množici neizbranih sladkarij recimo R. Našemu izboru do velikosti k manjka še $r := k - (p + 2q)$ sladkarij. Vprašanje je torej, kako poceni določiti vsoto vrednosti najdražjih r sladkarij v množici R. (Lahko se tudi izkaže, da je $r > |R|$, torej sploh ni ostalo dovolj sladkarij — to je še en znak, da smo vzeli premajhen p.) Pri tem je dobro imeti v mislih, da smo doslej ves čas razmišljali o eni konkretni vrednosti p, toda v resnici pravega p vnaprej ne poznamo, zato bomo morali preizkusiti vse in si zapomniti najboljšo od tako dobljenih rešitev. Ko počasi povečujemo p, se q počasi zmanjšuje, zato se množica R počasi povečuje (vanjo prihajajo novi elementi); želena velikost izbora r pa se pri povečevanju p-ja sprva povečuje (dokler se q še zmanjšuje), kasneje pa zmanjšuje (ko q doseže vrednost 0 in tam ostane).

Množico R lahko predstavimo na primer tako, da vse sladkarije v $B \cup C \cup D$ uredimo padajoče po vrednosti in nad tem seznamom zgradimo polno binarno drevo. Vsako vozlišče naj hrani dve števili: koliko je v R sladkarij iz poddrevesa, ki se začne pri tem vozlišču, ter kakšna je vsota njihovih vrednosti. Na začetku so v vseh vozliščih ničle, ko pa dodamo novo sladkarijo, recimo i, v množico R, povečamo v i-tem listu in vseh njegovih prednikih število sladkarij za 1 in vsoto njihovih vrednosti za c_i. Ko nas potem zanima vsota najvrednejših r sladkarij v množici R, lahko razmišljamo takole: začnimo v korenu drevesa in postavimo v na 0; če je števec sladkarij v levem otroku vsaj r, se premaknimo vanj, sicer pa povečajmo v za vsoto vrednosti sladkarij v levem otroku, zmanjšajmo r za število sladkarij v levem otroku in se premaknimo v desnega otroka. Ko pridemo v list, povečajmo v še za vrednost sladkarije v njem in imamo iskano vsoto. Vsaka operacija na takem drevesu (poizvedba ali dodajanje elementa) nam vzame $O(\log n)$ časa in pri vsaki vrednosti p-ja imamo največ dve dodajanji in eno poizvedbo, tako da nam ta rešitev vzame $O(n \log n)$ časa. (Toliko porabimo tudi za urejanje sladkarij po vrednosti na začetku.)

Še ena možnost je, da R predstavimo z dvema kopicama (prioritetnima vrstama), recimo R_1 in R_2; pri tem naj R_1 hrani najvrednejših r sladkarij in to tako, da je v korenu najmanj vredna med njimi, R_2 pa hrani ostale sladkarije in to tako, da je v korenu najvrednejša med njimi. Poleg tega vzdržujmo tudi vsoto vrednosti vseh sladkarij v R_1 — to je ravno vsota najvrednejših r sladkarij iz R, ki nas zanima za potrebe našega izbora. Ko dodajamo nove elemente v R, to v praksi pomeni, da jih dodamo v R_1, če so vsaj tako vredni kot najmanj vredni element v R_1, sicer pa jih dodamo v R_2; in nato po potrebi premestimo nekaj elementov iz R_1 v R_2 ali obratno, tako da ima R_1 spet natanko r elementov. (Tudi če nismo dodali nobenega elementa, se je mogoče spremenil r in je premeščanje potrebno že zaradi tega.) Tudi tu imamo z vsako operacijo na kopicah $O(\log n)$ dela in konstantno mnogo takih operacij pri vsakem p, tako da ima tudi ta rešitev časovno zahtevnost $O(n \log n)$. Potencialna

prednost v primerjavi s prejšnjo rešitvijo je, da ima marsikateri programski jezik primerno implementacijo kopice že v svoji standardni knjižnici.

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

typedef long long int llint;
priority_queue<int, vector<int>, greater<int>> R;
priority_queue<int, vector<int>, less<int>> S;
llint vr; // vsota elementov R-ja

void Prerazporedi(int r) { // poskrbi, da bo v R res točno r elementov
  while (R.size() > r) { int v = R.top(); S.push(v); R.pop(); vr -= v; }
  while (R.size() < r) { int v = S.top(); R.push(v); S.pop(); vr += v; } }

int main()
{
  // Preberimo vhodne podatke.
  int n, k, x; scanf("%d %d %d", &n, &k, &x);
  vector<int> C(n); for (int&ci : C) scanf("%d", &ci);
  vector<bool> DJ(n, false), DM(n, false); // dobri Janku/Metki
  for (int kdo = 0; kdo < 2; kdo++) {
    auto &v = (kdo == 0) ? DJ : DM; int d; scanf("%d", &d);
    while (d-- > 0) { int i; scanf("%d", &i); v[i - 1] = true; } }

  // Pripravimo sezname vrednosti sladkarij, ki so dobre Janku, Metki, obema ali nikomur.
  vector<int> D[4]; // dobri nikomur / samo Metki / samo Janku / obema
  for (int i = 0; i < n; i++) D[(DJ[i] ? 2 : 0) + (DM[i] ? 1 : 0)].push_back(C[i]);

  // Uredimo jih padajoče po ceni.
  for (auto &v : D) sort(v.begin(), v.end(), greater<int>());

  // Recimo, da uporabimo natanko p takih sladkarij, ki so všeč obema.
  // Naj bo vp vsota njihovih vrednosti. Potem moramo uporabiti še
  // q := max(0, x - p) takih, ki so všeč le Janku, in prav toliko
  // takih, ki so všeč Metki; naj bo vq vsota njihovih vrednosti.
  // Tiste, ki jih še nismo izbrali in niso všeč obema, bomo hranili v
  // kopicah R in S, in sicer največjih r v kopici R (njihova vsota bo vr),
  // ostale pa v S. Za r moramo vzeti r := k - p - 2q, da dobimo izbor k sladkarij.
  // — Za začetek pripravimo vse za p = 0.
  llint vp = 0, vq = 0, naj = -1; vr = 0; int q = x;
  for (int j = 0; j <= 2; j++) for (int i = 0; i < D[j].size(); i++)
    if (i >= (j == 0 ? 0 : q)) S.push(D[j][i]);
    else if (j > 0) vq += D[j][i];

  // Preglejmo vse možne p.
  for (int p = 0; p <= D[3].size() && p <= k; p++)
  {
    int qPrej = q; q = max(0, x - p); if (p > 0) vp += D[3][p - 1];

    // Ko se q zmanjša za 1, izpadeta dva elementa iz vsote vq
    // in se premakneta v kopici R oz. S.
    if (q < qPrej) for (int j = 1; j <= 2; j++) if (q < D[j].size()) {
      int v = D[j][q]; vq -= v;
      if (R.empty() || v < R.top()) S.push(v);
      else { vr += v; R.push(v); } }
```

```
        // Če je q prevelik (ker je p premajhen), izbor pri tem p sploh ni mogoč.
        if (q > D[1].size() || q > D[2].size()) continue;

        // Izmed še neizbranih sladkarij, ki niso všeč obema, moramo izbrati največjih r.
        int r = k - p - 2 * q;

        // Če jih sploh ni toliko, je p premajhen in izbor pri njem ni mogoč.
        if (r < 0 || r > R.size() + S.size()) continue;

        // Poskrbimo, da bo v R res točno r sladkarij (ostale pa v S).
        Prerazporedi(r);

        // Najboljšo rešitev si zapomnimo.
        naj = max(naj, vp + vq + vr);
    }
    // Izpišimo rezultat.
    printf("%lld\n", naj); return 0;
}
```

5. Ključavničarstvo

Ker je vsaka soba prek hodnikov povezana z vsako drugo, hodniki pa ne tvorijo ciklov, ima tloris stavbe obliko drevesa, torej povezanega acikličnega neusmerjenega grafa, v katerem točke predstavljajo sobe, povezave pa hodnike med njimi. Lažje si ga predstavljamo, če ga narišemo v hierarhični obliki: točko 1, v kateri naš igralec začne svoj sprehod, vzemimo za koren drevesa, vse ostale povezave pa naj visijo navzdol od tam. Za primera iz besedila naloge na primer dobimo (ob vsaki točki smo napisali število ključev v njej, ob vsaki povezavi pa število ključavnic na njej):

Če imamo povezavo med u in v in je u bližje korenu kot v, temu z drugimi besedami rečemo, da je v *otrok* točke u, ta pa je *starš* točke v. Ker naš igralec začne svoj sprehod v korenu, vemo, da preden prvič obišče točko v, mora gotovo že obiskati tudi njenega starša.

Ko pride igralec prvič v točko u, pobere k_u ključev v njej. Spomnimo se, da nas zanima scenarij, pri katerem on porabi čim več ključev, po možnosti toliko, da mu jih zmanjka, še preden uspe obiskati vse točke v grafu. Recimo, da je povezava od u do nekega njegovega otroka v zaklenjena z ℓ ključavnicami in da je $\ell > 1$. V tem primeru lahko naš igralec, ko prvič pride v u, takoj porabi $\ell - 1$ ključev, da odklene na tej povezavi vse ključavnice razen ene; s tem je svoje število ključev nekoliko zmanjšal, povezava pa je še vedno zaklenjena in s tem neprehodna. Zato lahko v mislih kar zmanjšamo k_u (število ključev v u) za $\ell - 1$, nato pa število ključavnic na prej omenjeni povezavi zmanjšamo z ℓ na 1. Po tej spremembi imajo vse povezave 0 ali 1 ključavnico, število ključev v posamezni točki pa je lahko po novem tudi negativno (kar pomeni, da lahko takoj po prihodu v tisto točko z odklepanjem

ključavnic na povezavah do otrok porabimo več ključev, kot smo jih v tisti točki dobili).

Za poddrevo, ki se začne pri točki u (torej obsega to točko in vse njene potomce), naj bo $f(u)$ največja možna razlika med številom odklenjenih ključavnic in številom pobranih ključev pri sprehajanju po tem poddrevesu (z začetkom v točki u), če si primerno izberemo, katere dele poddrevesa bi obiskali. Če je $f(u) > 0$, to pomeni, da lahko s sprehajanjem po tem poddrevesu porabimo več ključev (za odklepanje ključavnic), kot jih pridobimo z obiski sob v njem; če pa je $f(u) < 0$, to pomeni, da v tem poddrevesu neizogibno poberemo več ključev, kot jih porabimo.

Funkcijo f lahko računamo od spodaj navzgor po drevesu: preden jo izračunamo za točko u, jo je koristno poznati za njene otroke. Recimo, da ima u otroke $v_1, v_2, \ldots, v_k$ in da do otroka v_i vodi povezava z $\ell_i \in \{0, 1\}$ ključavnicami. Pri izračunu $f(u)$ lahko razmišljamo takole: za začetek ob vstopu v u pridobimo k_u ključev, torej za začetek postavimo $f(u)$ na $-k_u$; nato pa moramo za vsakega otroka v_i razmisliti, ali se ga splača obiskati ali ne. Pravzaprav, če povezava od u do v_i nima nobene ključavnice ($\ell_i = 0$), potem tistega otroka naš igralec neizogibno lahko obišče, če je obiskal tudi u (z drugimi besedami, ni takega scenarija, v katerem bi on obiskal u, točka v_i pa bi mu bila nedosegljiva); in ko takega otroka obišče, si lahko (s sprehajanjem po njegovem poddrevesu) število ključev zmanjša za $f(v_i)$, ne pa za več kot toliko. Za take otroke moramo torej nujno prišteti $f(v_i)$ k vrednosti $f(u)$.

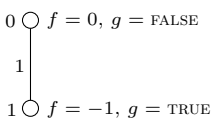
Pri tistih otrocih v_i pa, do katerih vodi povezava s ključavnico ($\ell_i = 1$), se lahko odločimo: lahko v otroka v_i ne gremo, s čimer se nam število ključev ne spremeni, lahko pa vanj gremo, ob čemer porabimo en ključ in nato s sprehajanjem po v_i-jevem poddrevesu zmanjšamo število svojih ključev še za $f(v_i)$. Od teh dveh možnosti je boljša tista, ki nam število ključev najbolj zmanjša. Za take otroke moramo torej prišteti k vrednosti $f(u)$ maksimum $\max\{0, 1 + f(v_i)\}$.

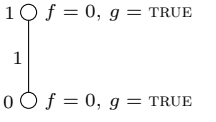
Na koncu nas bo zanimalo, kakšna je vrednost funkcije f v korenu, torej v točki 1. Spomnimo se, da naš igralec začne sprehod v tej točki z 0 ključi (preden pobere tistih k_1 ključev, ki ga čakajo v tej točki). Če je $f(1) \geq 0$, to pomeni, da je mogoč tak sprehod, pri katerem mu število ključev sčasoma pade nazaj na začetno vrednost, torej na 0 (če je $f(1) = 0$) ali celo pod 0 (če je $f(1) > 0$). Takrat pa igralec ne more odkleniti nobene nove povezave več, saj je na vsaki zaklenjeni povezavi vsaj ena ključavnica, torej ne more obiskati celotnega drevesa. (Povezave, ki nimajo ključavnic, smo upoštevali že pri izračunu funkcije f, saj smo takrat rekli, da je otroka, do katerega vodi povezava brez ključavnic, vedno nujno treba obiskati; spremembe v številu ključev zaradi obiskovanja takih otrok in njihovih poddreves so torej že zajete v vrednosti funkcije f.) Po drugi strani pa, če je $f(1) < 0$, to pomeni, da se po drevesu ni mogoče sprehajati tako, da bi porabili več ključev, kot jih poberemo, torej scenarij, po kakršnem sprašuje naloga, ne obstaja.

V primeru, ko je $f(1) = 0$, nastopi pravzaprav še ena drobna komplikacija. Mogoče je, da število ključev pade nazaj na 0 šele takrat, ko že obiščemo celotno drevo. Takrat scenarij, po karkšnem sprašuje naloga, tudi ne obstaja, čeprav je $f(1) = 0$. Zato si moramo pri izračunu funkcije $f(u)$ zapomniti še to, ali smo njeno vrednost dobili tako, da smo obiskali vse točke v poddrevesu z začetkom pri u ali ne. Recimo tej vrednosti $g(u)$. Izračunamo jo takole: če u nima otrok, je $g(u) =$ TRUE; če ima otroke in smo pri izračunu $f(u)$ kakšnega od njih preskočili, je $g(u) =$ FALSE;

sicer pa je $g(u)$ konjunkcija vrednosti $g(v_i)$ po vseh u-jevih otrocih v_i.

Primer te težave kažeta naslednji sliki. Koren je v obeh primerih zgornja točka in vrednost funkcije f v njej je 0. Pri levi stavbi se naš igralec lahko zatakne (celo mora se, saj nima ključa, da bi odklenil ključavnico na povezavi), pri desni pa se ne more (saj v korenu dobi ključ, s katerim lahko odklene povezavo in tako obišče še drugo sobo):





Ta razmislek se nekoliko zaplete le v primeru, ko do nekega otroka v_i pelje povezava z eno ključavnico in je $f(v_i) = -1$ (do tega pride na primer pri levi stavbi na gornji sliki); če torej gremo vanj, najprej porabimo en ključ, da odklenemo ključavnico na povezavi do njega, nato pa se nam število ključev pri sprehodu po v_i-jevem poddrevesu poveča za 1, zato smo z vidika izračuna vrednosti $f(u)$ na istem, kot če ga sploh ne bi obiskali; z vidika izračuna $g(u)$ pa ne, kajti če otroka v_i preskočimo, bo $g(u)$ takoj postala FALSE, sicer pa ne nujno. Kaj naj naredimo? Razmišljati moramo takole: če to, ali obiščemo otroka v_i ali ne, nič ne vpliva na vrednost $f(u)$, potem tudi ne vpliva nič na vrednost $f(1)$ v korenu. In to, ali smo nekega otroka obiskali ali ne, je pomembno le v primeru, ko je $f(1) = 0$, kajti le takrat nam je pomembno, ali smo obiskali celotno drevo (preden nam je število ključev padlo na 0) ali ne. Tedaj pa — ker, kot smo pravkar videli, ostane vrednost $f(1)$ enaka ne glede na to, ali v_i obiščemo ali ne — lahko torej $f(1) = 0$ dosežemo tudi tako, da otroka v_i ne obiščemo; torej je mogoč tak sprehod, ki obišče nekaj točk (mogoče tudi v_i-jevega starša u, ne pa nujno), ostane na koncu pri 0 ključih, točke v_i pa še ni obiskal in vanjo tudi ne more, ker je na povezavi od u do v_i še vedno zaklenjena ključavnica. V takem primeru je torej pravilni odgovor ta, da scenarij, po kakršnem sprašuje naloga, obstaja; da bomo pri $f(1) = 0$ dali tak odgovor, moramo torej imeti v korenu $g(1) =$ FALSE, to pa lahko zagotovimo le, če pri izračunu $f(u)$ in $g(u)$ štejemo, da smo otroka v_i preskočili, tako da bo $g(u)$ zagotovo FALSE, to pa se bo kasneje preneslo dalje navzgor po drevesu vse do korena.

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

struct Tocka
{
  int stars = -1;                    // starš te točke
  vector<pair<int, int>> sosedje;    // pari ⟨sosed, št. ključavnic na hodniku⟩
  int kljuci;                        // število ključev v tej sobi
  int maxPoraba; bool obisceVse;     // f(u) in g(u) za to točko
};

int main()
{
  int T; scanf("%d", &T);
  while (T-- > 0)
  {
```

```
    // Preberimo naslednjo stavbo.
    int n; scanf("%d", &n);
    vector<Tocka> G(n); for (auto &U : G) scanf("%d", &U.kljuci);
    for (int j = 0; j < n − 1; j++) {
      int u, v, l; scanf("%d %d %d", &u, &v, &l); u−−; v−−;
      G[u].sosedje.emplace_back(v, l);
      G[v].sosedje.emplace_back(u, l); }
    // Preglejmo graf iz korena navzven.
    const int koren = 0; auto &R = G[koren]; R.stars = −1;
    vector<int> q; q.push_back(koren); int glava = 0;
    while (glava < q.size()) {
      int u = q[glava++]; auto &U = G[u];
      for (auto &[v, l] : U.sosedje) if (v != U.stars) {
        auto &V = G[v]; V.stars = u; // Označimo, da je u starš točke v.
        // Na povezavi od starša u do otroka v odklenimo vse ključavnice razen ene
        // in ustrezno zmanjšajmo število ključev v staršu u.
        if (l > 1) { U.kljuci −= l − 1; l = 1; }
        q.push_back(v); } }
    // Rešimo problem od spodaj navzgor.
    for (int i = q.size() − 1; i >= 0; i−−) {
      int u = q[i]; auto &U = G[u];
      // Ob vstopu v u poberemo vse ključe, ki nas čakajo v njem.
      U.maxPoraba = −U.kljuci; U.obisceVse = true;
      for (auto [v, l] : U.sosedje) if (v != U.stars) { auto &V = G[v];
        // Otroka obiščemo, če je povezava do njega brez ključavnic ali pa
        // če se nam z obiskom tega otroka število ključev zmanjša. Sicer ga preskočimo.
        if (l > 0 && V.maxPoraba + 1 <= 0) { U.obisceVse = false; continue; }
        U.maxPoraba += V.maxPoraba + l;
        U.obisceVse = U.obisceVse && V.obisceVse; } }
    printf("%s\n", R.maxPoraba > 0 || R.maxPoraba == 0 && !R.obisceVse ? "da" : "ne");
  }
  return 0;
}
```

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. PINi

Napišimo si najprej podprogram, ki iz zapisanih števk $wxyz$ izračuna varnostno kodo $abcd$ po pravilih iz besedila naloge. Ker v običajnih programskih jezikih nimamo operatorja $\oplus$, si bomo pomagali z operatorjem za ostanek po deljenju (npr. % ali **mod**): operanda moramo sešteti, izračunati ostanek po deljenju z 11, nato pa obdržati le ostanek po deljenju tega ostanka z 10.

```
int IzracunajKodo(int w, int x, int y, int z)
{
  int a = z;
  int b = ((w + x) % 11) % 10;
  int c = ((x + y) % 11) % 10;
  int d = ((y + z) % 11) % 10;
  return a * 1000 + b * 100 + c * 10 + d;
}
```

Če nas zdaj za neko konkretno kodo $abcd$ zanima, ali jo je mogoče dobiti iz kakšne kombinacije $wxyz$, gremo lahko z nekaj gnezdenimi zankami po vseh $wxyz$, pri vsakem izračunamo kodo in preverimo, če se ujema z našo:

```
bool JeDosegljiva(int koda)
{
  for (int w = 0; w <= 9; w++) for (int x = 0; x <= 9; x++)
  for (int y = 0; y <= 9; y++) for (int z = 0; z <= 9; z++)
    if (IzracunajKodo(w, x, y, z) == koda) return true;
  return false;
}
```

Glavni blok programa lahko kliče to funkcijo v zanki po vseh možnih kodah in sproti izpisuje tiste, ki se izkažejo za nedosegljive:

```
int main()
{
  for (int koda = 0; koda <= 9999; ++koda)
    if (! JeDosegljiva(koda)) printf("%04d\n", koda);
  return 0;
}
```

Ta rešitev je precej neučinkovita, saj za vsako od 10000 kod pregleda v najslabšem primeru do 10000 kombinacij $wxyz$, da se prepriča, če je tista koda res nedosegljiva. Boljšo rešitev dobimo, če gremo po vseh kombinacijah $wxyz$ le enkrat, pri vsaki izračunamo njej pripadajočo kodo in v neki tabeli označujemo, katere kode smo na ta način dosegli. Na koncu se moramo le še enkrat sprehoditi po tej tabeli in izpisati kode, ki jih nismo dosegli nobenkrat:

```
int main()
{
  bool dosegljiva[10000];
  for (int koda = 0; koda <= 9999; ++koda) dosegljiva[koda] = false;
  for (int w = 0; w <= 9; w++) for (int x = 0; x <= 9; x++)
  for (int y = 0; y <= 9; y++) for (int z = 0; z <= 9; z++)
```

```
      dosegljiva[IzracunajKodo(w, x, y, z)] = true;
   for (int koda = 0; koda <= 9999; ++koda)
      if (! dosegljiva[koda]) printf("%04d\n", koda);
   return 0;
}
```

Izkaže se, da je nedosegljivih 2008 od vseh 10000 možnih PINov.

Oglejmo si še rešitev v pythonu:

```
def IzracunajKodo(w, x, y, z):
   a = z
   b = ((w + x) % 11) % 10
   c = ((x + y) % 11) % 10
   d = ((y + z) % 11) % 10
   return a * 1000 + b * 100 + c * 10 + d

def JeDosegljiva(koda):
   for w in range(10):
      for x in range(10):
         for y in range(10):
            for z in range(10):
               if IzracunajKodo(w, x, y, z) == koda: return True
   return False

for koda in range(10000):
   if not JeDosegljiva(koda): print("%04d" % koda)
```

In še učinkovitejša različica:

```
dosegljiva = [False] * 10000
for w in range(10):
   for x in range(10):
      for y in range(10):
         for z in range(10):
            dosegljiva[IzracunajKodo(w, x, y, z)] = True
for koda in range(10000):
   if not dosegljiva[koda]: print("%04d" % koda)
```

2. INTERCAL

Za začetek lahko iz števila ukazov (recimo mu n) in števil a in b izračunamo minimalno in maksimalno število vljudnih ukazov, pri katerih se program še prevede. Če označimo število vljudnih ukazov z v, je delež vljudnih glede na vse ukaze enak v/n, torej mora veljati $v/n \geq a$ in $v/n \leq b$. Iz prvega pogoja dobimo $v \geq a \cdot n$, iz drugega pa $v \leq b \cdot n$. Poleg tega mora biti v celo število, torej je najmanjša primerna vrednost v resnici $\lceil a \cdot n \rceil$ (to je vrednost $a \cdot n$, zaokrožena navzgor na celo število), največja pa $\lfloor b \cdot n \rfloor$ (to je vrednost $b \cdot n$, zaokrožena navzdol na celo število). Pri tem se lahko izkaže, da primernega v sploh ni; na primer, če bi imeli $n = 7$ ukazov in meji $a = 0{,}6$ in $b = 0{,}7$, nam spodnja meja $a = 0{,}6$ pove, da potrebujemo vsaj $\lceil a \cdot n \rceil = \lceil 0{,}6 \cdot 7 \rceil = \lceil 4{,}2 \rceil = 5$ vljudnih ukazov, zgornja meja $b = 0{,}7$ pa nam pove, da smemo imeti največ $\lfloor b \cdot n \rfloor = \lfloor 0{,}7 \cdot n \rfloor = \lfloor 4{,}9 \rfloor = 4$ vljudne ukaze. V takem primeru lahko le zaključimo, da je problem nerešljiv, saj obema pogojema hkrati ne bomo mogli ustreči.

Drugače pa si lahko izberemo poljuben v z območja od $\lceil a \cdot n \rceil$ do $\lfloor b \cdot n \rfloor$ in popravimo dano zaporedje ukazov tako, da bo vsebovalo natanko v vljudnih ukazov. Pri tem se nam ni nujno treba ozirati na to, kateri ukazi so bili že od prej vljudni in koliko jih je bilo, saj naloga ne predpisuje nobenih omejitev glede tega, koliko ukazov spremenimo in katere. Obstoječe fraze „DO“, „PLEASE“ ali „PLEASE DO“ lahko preprosto porežemo z začetka ukazov in nato pri prvih v ukazih na začetek postavimo „PLEASE“, pri ostalih pa „DO“ — tako bo nastal program z v vljudnimi ukazi.

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

void Popravi(double a, double b, const vector<string>& ukazi)
{
  // Iz deležev a in b izračunajmo najmanjše in največje
  // dovoljeno število vljudnih ukazov.
  int stUkazov = ukazi.size();
  int minVljudnih = (int) ceil(a * stUkazov);
  int maxVljudnih = (int) floor(b * stUkazov);

  // Če je zgornja meja manjša od spodnje, je problem nerešljiv.
  if (maxVljudnih < minVljudnih) { printf("Problem je neresljiv.\n"); return; }

  // Sprehodimo se po zaporedju ukazov.                                    (⋆)
  static const vector<string> prefiksi = {"PLEASE DO", "PLEASE", "DO"};
  for (int i = 0; i < ukazi.size(); i++)
  {
    const char *s = ukazi[i].c_str();

    // Poglejmo, kateri prefiks se pojavlja na začetku tega ukaza.
    int dolzinaPrefiksa = 0;
    for (const string &p : prefiksi) {
      int d = p.length();
      if (strncmp(s, p.c_str(), d) == 0) { dolzinaPrefiksa = d; break; } }

    // Izpišimo preostanek ukaza s primernim prefiksom.
    printf("%s%s\n", (i < minVljudnih) ? "PLEASE" : "DO", s + dolzinaPrefiksa);
  }
}
```

Zapišimo to rešitev še v pythonu:

```
import math

def Popravi(a, b, ukazi):
  # Iz deležev a in b izračunajmo najmanjše in največje
  # dovoljeno število vljudnih ukazov.
  stUkazov = len(ukazi)
  minVljudnih = int(math.ceil(a * stUkazov))
  maxVljudnih = int(math.floor(b * stUkazov))

  # Če je zgornja meja manjša od spodnje, je problem nerešljiv.
  if maxVljudnih < minVljudnih: print("Problem je neresljiv."); return

  # Sprehodimo se po zaporedju ukazov.                                    (⋆)
  for i, ukaz in enumerate(ukazi):
```

```
# Odrežimo prefiks, ki se pojavlja na začetku ukaza.
for p in ["PLEASE DO", "PLEASE", "DO"]:
  if ukaz.startswith(p): ukaz = ukaz[len(p):]; break

# Izpišimo ukaz s primernim prefiksom.
print("%s%s" % ("PLEASE" if i < minVljudnih else "DO", ukaz))
```

Razmislimo zdaj še o težji različici naloge, ki jo omenja opomba pod črto v besedilu naloge: radi bi torej število vljudnih ukazov spravili v predpisane okvire in pri tem spremenili čim manj ukazov prvotnega vhodnega zaporedja. Zdaj bomo morali, preden karkoli spreminjamo, najprej prešteti, koliko je vljudnih ukazov v vhodnem zaporedju; recimo, da jih je v. Spomnimo se, da bomo morali na koncu imeti vsaj $\lceil a \cdot n \rceil$ in kvečjemu $\lfloor b \cdot n \rfloor$ vljudnih ukazov. Če je $v < \lceil a \cdot n \rceil$, je vljudnih ukazov premalo (oz. je nevljudnih preveč) in bomo morali $\lceil a \cdot n \rceil - v$ nevljudnih ukazov spremeniti v vljudne; podobno, če je $v > \lfloor b \cdot n \rfloor$, je vljudnih ukazov preveč in bomo morali $v - \lfloor b \cdot n \rfloor$ vljudnih ukazov spremeniti v nevljudne; sicer pa (torej če je $\lceil a \cdot n \rceil \leq v \leq \lfloor b \cdot n \rfloor$) je število vljudnih ukazov že primerno in ni treba spreminjati ničesar.

Lahko si omislimo dve spremenljivki, v katerih na začetku izračunamo, koliko preveč je v vhodnem zaporedju vljudnih in koliko nevljudnih ukazov. Pri vsakem vljudnem ukazu potem pogledamo, če je števec prevecVljudnih še vedno večji od 0; če je, spremenimo ukaz v nevljudnega in števec zmanjšamo za 1. Podobno razmišljamo tudi pri nevljudnih ukazih. V naši dosedanji rešitvi v C++ moramo na primer glavno zanko (od vrstice ($\star$) naprej) zamenjati z nečim takšnim:

```
// Preštejmo, koliko ukazov je vljudnih.
int stVljudnih = 0; for (const string &s : ukazi) if (s[0] == 'P') ++stVljudnih;
// Izračunajmo, koliko preveč vljudnih/nevljudnih ukazov imamo.
int prevecVljudnih = stVljudnih - maxVljudnih;
int prevecNevljudnih = minVljudnih - stVljudnih;

// Izpišimo primerno popravljeno zaporedje ukazov.
static const vector<string> prefiksi = {"PLEASE DO", "PLEASE", "DO"};
for (const string &S : ukazi)
{
  // Mogoče ga sploh ni treba spreminjati.
  const char *s = S.c_str(); bool vljuden = (*s == 'P');
  if (vljuden && prevecVljudnih <= 0 || ! vljuden && prevecNevljudnih <= 0) {
    printf("%s\n", s); continue; }

  // Poglejmo, kateri prefiks se pojavlja na začetku tega ukaza.
  int dolzinaPrefiksa = 0;
  for (const string &p : prefiksi) {
    int d = p.length();
    if (strncmp(s, p.c_str(), d) == 0) { dolzinaPrefiksa = d; break; } }

  // Izpišimo ukaz s spremenjenim prefiksom.
  if (vljuden) { printf("DO%s\n", s + dolzinaPrefiksa); --prevecVljudnih; }
  else { printf("PLEASE%s\n", s + dolzinaPrefiksa); --prevecNevljudnih; }
}
```

Podobno lahko v naši dosedanji rešitvi v pythonu zamenjamo vse od vrstice ($\star$) naprej z nečim takšnim:

```
# Preštejmo, koliko ukazov je vljudnih.
stVljudnih = sum(1 for ukaz in ukazi if ukaz.startswith("PLEASE"))
```

```
    # Izračunajmo, koliko preveč vljudnih/nevljudnih ukazov imamo.
    prevecVljudnih = stVljudnih − maxVljudnih; prevecNevljudnih = minVljudnih − stVljudnih

    # Izpišimo primerno popravljeno zaporedje ukazov.
    for ukaz in ukazi:

        # Mogoče ga sploh ni treba spreminjati.
        vljuden = ukaz.startswith("PLEASE")
        if vljuden and prevecVljudnih <= 0 or not vljuden and prevecNevljudnih <= 0:
            print(ukaz); continue

        # Odrežimo prefiks, ki se pojavlja na začetku ukaza.
        for p in ["PLEASE DO", "PLEASE", "DO"]:
            if ukaz.startswith(p): ukaz = ukaz[len(p):]; break

        # Izpišimo ukaz s spremenjenim prefiksom.
        if vljuden: print("DO" + ukaz); prevecVljudnih −= 1
        else: print("PLEASE" + ukaz); prevecNevljudnih −= 1
```

3. Najdaljša pot

Če si izberemo neki začetni položaj, lahko v zanki sledimo poteku poti v skladu s pravili naloge. Na vsakem koraku izračunamo koordinati polja, proti kateremu kaže puščica na trenutnem polju. Če je novi položaj zunaj mreže, je premik neveljaven in pot se konča; podobno tudi, če se izkaže, da puščica na novem polju kaže nazaj na trenutno polje ali pa da smo novo polje obiskali že kdaj prej. Če pa ne drži nič od tega, je premik veljaven in lahko s sledenjem poti nadaljujemo na novem polju. Da lahko poceni preverimo, ali smo neko polje prej že obiskali, si pomagamo s tabelo (v spodnjem podprogramu je to vektor zeBili), v kateri označujemo, katera polja smo že obiskali.

Ker naloga sprašuje po najdaljši poti sploh, po vseh možnih začetnih položajih, moramo razmislek iz gornjega odstavka oviti še v eno zanko, ki preizkusi vse možne začetne položaje in si zapomni najdaljšo tako odkrito pot. Zapišimo takšno rešitev v C++:

```
#include <vector>
using namespace std;
typedef enum { Gor, Dol, Levo, Desno } Smer;
const int DX[] = { 0, 0, −1, 1 };
const int DY[] = { −1, 1, 0, 0 };

int NajdaljsaPot(const vector<vector<Smer>> &mreza)
{
  const int w = mreza[0].size(), h = mreza.size();
  vector<int> zeBili(w * h, −1);
  int najDolzina = 0;
  for (int z = 0; z < w * h; z++)
  {
    int x = z % w, y = z / w, dolzina = 1;
    // Izračunajmo dolžino poti z začetkom na polju (x, y).
    // Že obiskana polja bomo v tabeli zeBili označili z vrednostjo z.
    zeBili[z] = z;
    while (true)
    {
      // Izračunajmo novi koordinati pri tem premiku.
      Smer s = mreza[y][x]; x += DX[s]; y += DY[s];
```

```
        // Ali bi padli iz mreže?
        if (x < 0 || x >= w || y < 0 || y >= h) break;
        // Ali kaže novo polje nazaj na prejšnje?
        Smer s2 = mreza[y][x];
        if (DX[s2] == -DX[s] && DY[s2] == -DY[s]) break;
        // Ali smo v novem polju že bili?
        if (zeBili[x + y * w] == z) break;
        // Če ne, se premaknimo vanj.
        zeBili[x + y * w] = z; ++dolzina;
      }
      // Če je to najdaljša pot doslej, si jo zapomnimo.
      if (dolzina > najDolzina) najDolzina = dolzina;
    }
  return najDolzina;
}
```

Za označevanje že obiskanih polj bi lahko imeli tabelo oz. vektor logičnih vrednosti (**bool**-ov), ki bi jih pri vsakem začetnem položaju najprej vse inicializirali na **false**. To je rahlo potratno, saj je prav mogoče, da večine polj pri prejšnji poti sploh nismo obiskali in so zato tisti elementi tabele že imeli vrednost **false**. Druga možnost je, da ko določimo dolžino poti (pri nekem začetnem položaju), sledimo isti poti še enkrat in postavljamo vrednosti v zeBili na **false**. Tretja možnost, ki smo jo uporabili v gornjem podprogramu, pa je, da je zeBili vektor celoštevilskih vrednosti (**int** namesto **bool**), v katerem uporabljamo vrednost z (trenutni začetni položaj) kot **true**, katerokoli manjšo vrednost (ki je tam ostala še od prejšnjih začetnih položajev) pa si razlagamo kot **false**.

Še ena možnost pa je, da namesto tabele uporabimo množico oz. razpršeno tabelo, v katero shranjujemo le polja, ki smo jih že obiskali. Ta pristop uporablja spodnja pythonovska različica naše rešitve:

```
Gor = 0; Dol = 1; Levo = 2; Desno = 3
DX = [0, 0, -1, 1]; DY = [-1, 1, 0, 0]

def NajdaljsaPot(mreza):
  w = len(mreza[0]); h = len(mreza)
  najDolzina = 0
  for y0 in range(h):
    for x0 in range(w):
      # Preglejmo pot z začetkom v (x0, y0).
      x = x0; y = y0; zeBili = set(); zeBili.add((x, y))
      while True:
        # Izračunajmo novi koordinati pri tem premiku.
        s = mreza[y][x]; x += DX[s]; y += DY[s]
        # Ali bi padli iz mreže?
        if x < 0 or x >= w or y < 0 or y >= h: break
        # Ali kaže novo polje nazaj na prejšnje?
        s2 = mreza[y][x]
        if DX[s2] == -DX[s] and DY[s2] == -DY[s]: break
        # Ali smo v novem polju že bili?
        if (x, y) in zeBili: break
        # Če ne, se premaknimo vanj.
```

```
            zeBili.add((x, y))
        najDolzina = max(najDolzina, len(zeBili))
    return najDolzina
```

Pri mreži velikosti $w \times h$ je lahko pot dolga največ $w \cdot h$ polj (npr. če lepo sistematično cikcaka po celi mreži). V takem primeru bomo pri $O(wh)$ začetnih položajih morali slediti poti vsaj $O(wh)$ korakov daleč, preden se bomo ustavili, zato je časovna zahtevnost naše rešitve v najslabšem primeru kar $O(w^2h^2)$. Razmislimo še o učinkovitejši rešitvi.

Dolžino poti, ki se začne na polju u, bomo označili z $d[u]$. Naloga torej sprašuje po $\max_u d[u]$, pri čemer gre u po vseh poljih mreže.

Videli smo, da se pot konča bodisi zato, ker naslednja poteza s trenutnega polja sploh ni možna (ker bi padli iz mreže ali pa se premaknili v polje, ki kaže nazaj na trenutno polje) ali pa ker bi nas pripeljala na neko že prej obiskano polje — v tem slednjem primeru torej vidimo, da možni premiki tvorijo cikel. Če tak cikel sestavlja k polj, lahko za vsako od njih takoj zaključimo, da je dolžina poti z začetkom na tistem polju natanko k (ker bi taka pot sledila ciklu, dokler ne bi prišla nazaj na začetno polje).

Kaj pa, če u ne leži na ciklu? Ena možnost je, da premik z njega sploh ni mogoč; tedaj je pač $d[u] = 1$. Sicer pa nas premik s polja u pripelje v enega od njegovih sosedov, recimo v; nadaljevanje poti točke u gotovo ne bo več doseglo (saj bi to pomenilo, da u leži na ciklu, kar pa ni res), torej bo nadaljevanje poti potekalo popolnoma enako, kot če bi s potjo začeli šele pri v namesto pri u. Dolžina tega preostanka poti je torej $d[v]$, celotna pot skupaj z začetnim korakom od u do v pa je še za en korak daljša; torej imamo $d[u] = d[v] + 1$.

Recimo, da začnemo s potjo pri u_1 in izvedemo $k-1$ korakov: $u_1 \to \ldots \to u_k$; in recimo, da iz u_k naslednji korak ni mogoč zato, ker bi padli čez rob mreže ali pa prišli v polje, ki kaže nazaj na u_k. Potem lahko takoj zaključimo, da je $d[u_k] = 1$, $d[u_{k-1}] = 2$ in tako nazaj do $d[u_1] = k$. Za vsa ta polja torej zdaj poznamo dolžino poti z začetkom na njih in se nam z njimi ne bo treba več ukvarjati.

Podobno je, če se nam pot sčasoma zacikla: recimo, da pri u_k opazimo, da nas premik od tam pelje v u_i za nek $i < k$. Polja $u_i, u_{i+1}, \ldots, u_k$ torej tvorijo cikel dolžine $c := k - i + 1$. Zanje torej lahko zaključimo, da je $d[u_i] = d[u_{i+1}] = \ldots = d[u_k] = c$; za polja pred njimi na poti pa, da je $d[u_{i-1}] = c + 1$ in tako nazaj do $d[u_1] = k$.

Če pa med sledenjem poti iz u_1 pridemo (recimo po $k-1$ korakih) do nekega polja u_k, za katero že poznamo $d[u_k]$, lahko takoj zaključimo, da je $d[u_{k-1}] = d[u_k] + 1$ in tako nazaj do $d[u_1] = d[u_k] + k - 1$. Nadaljevanju poti zato v tem primeru ni treba slediti.

V vsakem primeru torej za vsak korak, ki ga pri sledenju poti naredimo, uspemo tudi določiti $d[\cdot]$ za eno novo polje. Zato pri sledenju vseh poti skupaj (po vseh možnih začetnih položajih) naredimo le toliko korakov, kolikor je vseh polj v mreži; časovna zahtevnost te rešitve je tako le še $O(wh)$. Zapišimo še to rešitev:

```
def NajdaljsaPot2(mreza):
  w = len(mreza[0]); h = len(mreza); n = w * h

  # Polja bomo predstavili s števili: (x, y) predstavlja število x + y * w.
  # Funkcija Nasl(u) vrne številko polja, v katero se premaknemo iz polja u.
```

```
# Če premik iz u ni veljaven, vrne −1.
def Nasl(u):
    # Pretvorimo številko polja v koordinate.
    x = u % w; y = u // h; s = mreza[y][x]
    # Izračunajmo novi položaj.
    x += DX[s]; y += DY[s]
    # Ali smo padli iz mreže?
    if x < 0 or x >= w or y < 0 or y >= h: return −1
    # Ali puščica v novem polju kaže nazaj v staro?
    s2 = mreza[y][x]
    if DX[s2] == −DX[s] and DY[s2] == −DY[s]: return −1
    # Vrnimo številko novega polja.
    return x + y * w

# V tabeli d bomo hranili dolžine poti z začetkom pri posameznem polju.
d = [0] * n
for z in range(n):
    # Mogoče že poznamo dolžino poti z začetkom na polju z.
    if d[z]: continue
    # Sicer sledimo poti iz z, dokler se pot ne konča, zacikla ali pa naleti na polje,
    # pri katerem dolžino poti od tam naprej že poznamo. k šteje prehojena polja,
    # u je trenutni položaj; v d[u] vpisujemo števila −1, −2, . . . , ki povedo
    # položaj polja na poti (z njihovo pomočjo bomo lažje določili dolžino cikla).
    k = 1; u = z; d[u] = −k
    while True:
        v = Nasl(u)
        if v < 0 or d[v] != 0: break
        k += 1; u = v; d[u] = −k

    # Če je v < 0, to pomeni, da se je pot ustavila, ker premik naprej iz polja u
    # ni mogoč. Polja na poti (od z do u) dobijo dolžine poti od k do 1.
    if v < 0: d[z] = k; c = 0

    # Če je d[v] > 0, to pomeni, da je pot dosegla polje v, za katero že
    # poznamo pravo vrednost d[v]. Polja na poti od z do u dobijo
    # dolžine poti od d[v] + k do d[v] + 1.
    elif d[v] > 0: d[z] = k + d[v]; c = 0

    # Sicer je d[v] < 0, kar pomeni, da se naša pot zacikla: iz u se pot nadaljuje na v,
    # ki pa smo jo že obiskali nekoč prej, namreč d[v] − d[u] korakov prej. Cikel torej
    # pokriva c = d[v] − d[u] + 1 polj. Polja na poti od z do v dobijo dolžine poti
    # od k do c, vsa ostala polja na poti (od v do u) pa dobijo dolžino poti c.
    else: d[z] = k; c = d[v] − d[u] + 1

    # Pojdimo še enkrat po poti od z naprej in vpisujmo v d dolžine, ki jih zdaj poznamo.
    u = z
    for i in range(k − 1):
        v = Nasl(u); d[v] = max(d[u] − 1, c); u = v
return max(d)
```

4. Domače naloge

Učinkovitost posameznega dne je enaka težavnosti ene od nalog; in vsaka naloga lahko vpliva le na učinkovitost enega dne (namreč tisti dan, ko jo rešujemo); učinkovitost po d dneh torej ne more biti večja od vsote težavnosti najtežjih d nalog. To zgornjo mejo pa tudi res lahko dosežemo, če razporedimo naloge med dni tako, da vsak dan dobi eno od najtežjih d nalog. Za ostale naloge je potem vseeno, kako jih razporedimo med dneve. Če na primer v vhodnem zaporedju $t_1, \ldots, t_n$ nastopi

d najtežjih nalog na indeksih $m_1 < m_2 < \ldots < m_d$, lahko prvi dan rešimo naloge od 1 do $m_2 - 1$, drugi dan naloge od m_2 do $m_3 - 1$ in tako naprej, predzadnji dan rešimo naloge od m_{d-1} do $m_d - 1$, zadnji dan pa naloge od m_d do n.

Poseben primer nastopi, če je $d > n$, torej če je dni več kot nalog. Takrat je pač najbolje v n dneh rešiti vsak dan po eno nalogo, $d - n$ dni pa lahko lenarimo.

Vprašanje je zdaj le še to, kako poiskati d največjih števil v seznamu $t_1, \ldots, t_n$. Če je d majhen, gremo lahko preprosto z zanko po seznamu in sproti vzdržujemo spisek največjih d doslej prebranih elementov; vsakič ko preberemo nov element iz seznama, ga primerjamo z najmanjšim izmed d doslej največjih in slednjega zamenjamo z novim, če je le-ta večji. Zapišimo ta postopek s psevdokodo:

$M :=$ prazen seznam;
for $i := 1$ **to** n:
  if ima M manj kot d elementov:
    dodaj nalogo i v M;
  else if je naloga i težja od najlažje naloge v M:
    najlažjo nalogo v M zamenjaj z nalogo i;

Koristno je, če vzdržujemo podatek o tem, kje v seznamu M je trenutno najlažja naloga (izmed tistih, ki so v M); tako jo bomo imeli pri roki, ko jo bo treba primerjati z novo nalogo i. Ko pa najlažjo nalogo v M zamenjamo z i, se bomo morali sprehoditi po seznamu M, da bomo ugotovili, katera je zdaj najlažja naloga v njem; to vzame $O(d)$ časa in se v najslabšem primeru lahko zgodi pri vsakem i (npr. če je vhodno zaporedje $t_1, \ldots, t_n$ naraščajoče), zato ima ta rešitev časovno zahtevnost $O(n \cdot d)$.[6]

Če za vzdrževanje najtežjih d doslej prebranih nalog namesto seznama (kot je M zgoraj) uporabimo kopico, rdeče-črno drevo ali kaj podobnega, lahko časovno zahtevnost zmanjšamo na samo $O(n \log d)$. Še boljša rešitev je postopek quickselect, ki poišče največjih d elementov v samo $O(n)$ časa (v C++ovi standardni knjižnici lahko v ta namen uporabimo funkcijo `nth_element`).

5. Razbijanje permutacijske šifre

Preprosta rešitev je, da pošljemo napravi niz, ki ima na vseh mestih ničle, razen na enem mestu, kjer je enica. Tudi na izhodu bo torej niz z eno enico (drugod pa bodo ničle). Če smo recimo na vhodu poslali enico kot i-ti znak niza, na izhodu pa smo jo dobili kot j-ti znak, zdaj vemo, da šifrirnik preslika i-ti vhod v j-ti izhod. Ta poskus izvedimo n-krat, vsakič z drugim i-jem, pa bomo za vsak vhod vedeli, v kateri izhod ga naprava preslika. (Pravzaprav vemo vse že po $n - 1$ poskusih, saj potem za edini še preostali vhod ne more biti drugače, kot da se preslika na edini še preostali izhod.)

To rešitev lahko poskusimo na razne načine še izboljšati; na primer, ker smemo uporabljati znake od 0 do $b-1$ namesto le 0 in 1, lahko v enem poskusu pošljemo niz,

[6] Koristno bi bilo pregledovati elemente vhodnega zaporedja v naključnem vrstnem redu; potem velja, da več nalog ko smo že pregledali, težje so zdaj naloge v M in manj verjetno je, da bi bila naloga i težja od najlažje naloge v M; zato prihaja do sprememb v M vse redkeje. Pričakovano število takih sprememb je potem le $O(d \ln(n/d))$, kar je (če je število dni d majhno v primerjavi s številom nalog n) precej manj od $O(n)$, ki smo jih imeli zgoraj v najslabšem primeru (pri naraščajočem vhodnem zaporedju). S tem prijemom smo se že srečali pri 4. nalogi za prvo skupino leta 2003; za več o tem glej str. 550 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004*.

ki ima na enem mestu znak z vrednostjo 1, na naslednjem mestu znak z vrednostjo 2 in tako naprej do $b-1$, drugod pa ima ničle; tako bomo lahko za $b-1$ vhodov naenkrat ugotovili, v katere izhode se preslikajo. Tako bomo za razbijanje šifre potrebovali približno $n/(b-1)$ poskusov namesto n poskusov.

Še boljša rešitev pa je naslednja. Če pošljemo na vhod d nizov, dolgih po n znakov, si jih lahko predstavljamo kot zapisane v tabelo z n stolpci in d vrsticami. Vse, kar šifrirnik naredi, je to, da premeša stolpce tabele. Če hočemo biti zmožni ugotoviti, kateri stolpec je prišel kam, morajo biti vsi stolpci različni med seboj. Ker imajo stolpci po d znakov in za vsak znak je b možnosti, je možnih b^d različnih stolpcev. Vzeti moramo torej dovolj velik d, da bo $b^d \geq n$, torej $d \geq \log_b n$, torej $d = \lceil \log_b n \rceil$.

Za stolpce zdaj lahko vzamemo poljubne nize, samo da so vsi različni; še posebej elegantno je, če vzamemo kar zapis števil od 0 do $n-1$ v b-iškem sestavu (pri številih, ki imajo manj kot d števk, vrinimo na začetku ničle). Pokličimo torej kodirnik d-krat, in sicer pri i-tem klicu na j-tem vhodu podajmo i-to števko števila j v b-iškem zapisu. Nato moramo za vsak izhod le stakniti skupaj števke, ki jih je šifrirnik dajal na tem izhodu pri vseh d klicih, pa dobimo številko pripadajočega vhoda (v b-iškem zapisu).

Naloge so sestavili: PINi — Primož Gabrijelčič; domače naloge, Janko in Metka — Tomaž Hočevar; plonkanje, metanje na koš — Boris Horvat; INTERCAL, razbijanje permutacijske šifre, obračanje jogija, lenoba, zamik, ključavničarstvo — Vid Kocijan; ne odlašaj na jutri — Samo Kralj; najdaljša pot — Mitja Lasič; ključ, zobna ščetka — Mark Martinec; prelom besedila — Jure Slak; vesoljske vsote — Jasna Urbančič; semafor, vaje v slogu, zlaganje slik — Janez Brank.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2018

1. Križci in krožci

Ker velikosti mreže ne poznamo (in je potencialno lahko tudi zelo velika), je ne bomo hranili v običajni tabeli (*array*), pač pa v razpršeni tabeli (*hash table*). V njej bo za vsako ne-prazno celico mreže po en element, pri čemer bomo kot ključ uporabili par koordinat (x, y), kot pripadajočo vrednost pa število, ki so ga igralci vpisali v tisto celico mreže. V C++ lahko za razpršeno tabelo uporabimo razred unordered_map iz standardne knjižnice, v pythonu pa dict.

Naš podprogram Poteza mora najprej preveriti, če je trenutno opazovana celica že v razpršeni tabeli; če je, to pomeni, da so vanjo že prej vpisali neko število in je sedanja poteza neveljavna. Sicer pa moramo izračunati, koliko novih skupin D zaporednih enakih števil nastane s trenutno potezo (pri naši nalogi je $D = 5$, vendar nič ne škodi, če rešitev zapišemo za splošen D). Možne so štiri smeri take skupine: vodoravno, navpično, diagonalno gor in diagonalno gor; vse štiri bomo pregledali v zanki.

Za vsako smer poglejmo, kako dolga je strnjena skupina celic z vrednostjo n pred našo celico in kako dolga je taka skupina za našo celico. To dvoje lahko preverimo z zankama, pri čemer gremo največ $D-1$ celic daleč, saj tiste bolj oddaljene ne morejo tvoriti strnjene skupine D celic skupaj z našo trenutno. Če je pred našo celico $a-1$ takih z enako vrednostjo in za njo $b-1$ takih, tvorijo vse te celice z našo vred strnjeno skupino dolžine $a+b-1$, v kateri se skriva $(a+b-1)-(D-1) = a+b-D$ strnjenih skupin dolžine D; za toliko se bo torej povečalo število točk trenutnega igralca. To moramo sešteti po vseh smereh.

Oglejmo si primer implementacije take rešitve v pythonu:

```
mreza = {}
D = 5

def Poteza(x, y, n):
    if (x, y) in mreza: return -1
    mreza[x, y] = n
    tocke = 0
    for (dx, dy) in [(1, 0), (0, 1), (1, 1), (1, -1)]:
        a = 0; b = 0
        while a < D and mreza.get((x - a * dx, y - a * dy), 0) == n: a += 1
        while b < D and mreza.get((x + b * dx, y + b * dy), 0) == n: b += 1
        tocke += max(a + b - D, 0)
    return tocke
```

2. Taborniki

Recimo v splošnem, da moramo generirati listke s po w stolpci in h vrsticami. Naloga pravi, da morajo biti na vseh enaki znaki, le v različnem vrstnem redu; za začetek lahko torej sestavimo naključno zaporedje $w \cdot h$ znakov, nato pa ga za vsak listek naključno premešamo. Pri pripravi začetnega naključnega zaporedja si najprej naključno izberimo, koliko bo števk (od 3 do 7); nato dodajmo toliko naključnih števk; na koncu pa dodajmo še dovolj naključnih črk, da bo niz na koncu dolg $w \cdot h$ znakov.

Znake osnovnega zaporedja lahko potem za vsak listek naključno premešamo s preprosto zanko, ki na vsakem koraku zamenja trenutni znak z naključno izbranim znakom od trenutnega mesta do konca niza:

vhod: niz $s[1..wh]$;
for $i := 1$ **to** $w \cdot h$:
  (* *Na indeksih* $s[1], \ldots, s[i-1]$ *so znaki, ki smo jih za trenutni listek že uporabili, na indeksih* $s[i], \ldots, s[w \cdot h]$ *pa tisti, ki jih še nismo. Izmed slednjih naključno izberimo enega in ga zamenjajmo s tistim na* $s[i]$. *)
  $j :=$ naključno število izmed $\{i, i+1, \ldots, w \cdot h\}$;
  zamenjaj $s[i]$ in $s[j]$;

Če hočemo poleg tega še zagotoviti, da na listku ne bosta nobeni dve vrstici enaki, lahko na koncu vsake vrstice (ko je v gornji zanki i večkratnik w-ja) preverimo, če je ta vrstica različna od dosedanjih, in če ni, se vrnemo nazaj na začetek vrstice. V naši spodnji rešitvi bomo vrstice hranili v razpršeni tabeli, ker pa jih je malo (pri naši nalogi bo $h = 5$), bi lahko uporabili tudi še eno vgnezdeno zanko, ki bi trenutno vrstico primerjala z vsemi dosedanjimi.

Podobno je tudi, če hočemo zagotoviti, da so vsi listki različni. Lahko jih hranimo v razpršeni tabeli, lahko pa v navadnem seznamu oz. tabeli in novi listek primerjamo v zanki z vsemi dosedanjimi. Če je novi listek enak kakšnemu od že obstoječih, ga zavržemo, sicer pa ga izpišemo in dodamo v razpršeno tabelo ali seznam.

Oglejmo si implementacijo takšne rešitve v C++:

```
#include <random>
#include <cstdio>
#include <unordered_set>
#include <string>
#include <string_view>

using namespace std;
const int w = 6, h = 5;                    // širina in višina listka
const int minStevk = 3, maxStevk = 7; // dovoljeno število števk na listku (ostalo so črke)

template<typename R>
void Premesaj(R&& r, string &s)
{
  unordered_set<string_view> vrstice;
  for (int y = 0; y < h; )
  {
    // Sestavimo y-to vrstico.
    for (int x = 0; x < w; x++)
      swap(s[y * w + x], s[uniform_int_distribution<>{y * w + x, y * w + w - 1}(r)]);
    // Preverimo, če je različna od dosedanjih.
    auto [it, jeNova] = vrstice.emplace(&s[y * w], w);
    if (jeNova) y++;
  }
}

void IzpisiListke(int n)
{
  mt19937_64 r; // Generator naključnih števil.
  uniform_int_distribution<> dNStevk{minStevk, maxStevk}, dStevke{0, 9}, dCrke{0, 25};
```

```
  // Pripravimo osnovno zaporedje znakov.
  string s0;
  const int nStevk = dNStevk(r);
  for (int i = 0; i < nStevk; i++) s0.push_back('0' + dStevke(r));
  for (int i = nStevk; i < w * h; i++) s0.push_back('A' + dStevke(r));

  // Pripravimo n premešanih zaporedij.
  unordered_set<string> listki;
  for (int i = 0; i < n; i++)
  {
    // Premešajmo zaporedje in pazimo, da bo različno od dosedanjih.
    string s = s0; do { Premesaj(r, s); } while (listki.find(s) != listki.end());
    listki.emplace(s);

    // Izpišimo ga v obliki listka.
    printf("%sEKIPA %d\n", (i > 0 ? "\n" : ""), i + 1);
    for (int y = 0; y < h; y++) for (int x = 0; x < w; x++)
      printf("%c%s", s[y * w + x], (x == w - 1) ? "\n" : " ");
  }
}
```

Gornjo rešitev bi se dalo še malo izboljšati; v trenutni obliki obstaja možnost (sicer zelo majhna), da se bo zaciklala. Začetni niz `s0` ima od 3 do 7 števk, ostalo so črke; lahko se na primer zgodi, da so vse števke enake in vse črke enake, npr. `000AAA...AA`. Pri generiranju listka se potem lahko zgodi, da vse tri ničle pridejo v prvo vrstico, ostale vrstice pa si potem ne morejo biti različne med seboj, ker bodo vse sestavljene iz samih `A`-jev. Podprogram `Premesaj` bi torej lahko izboljšali tako, da bi štel, kolikokrat je že neuspešno poskusil sestaviti trenutno vrstico, in se v primeru nekaj neuspelih poskusov vrnil nazaj na začetek listka.

Če bi nas zanimalo tudi večje število listkov (naloga sicer pravi, da jih bo največ 100), bi prišlo do podobne težave tudi pri zagotavljanju različnosti listkov. Pri začetnem nizu, ki ima le tri ničle in drugod same `A`-je, je možno položaj ničel na listku izbrati na $\binom{w \cdot h}{3}$ načinov, kar je na primer pri $w = 6$, $h = 5$ enako $\binom{30}{3} = 4060$, torej več kot toliko različnih listkov ne bi bilo mogoče sestaviti. Da bi se z gotovostjo izognili tudi temu, bi moral tudi `IzpisiListke` šteti, kolikokrat mu je spodletelo pri sestavljanju novega listka, in po nekaj neuspelih poskusih zavreči vse dosedanje listke in začeti znova z novim osnovnim zaporedjem `s0`; izpisati pa bi moral listke šele takrat, ko mu jih res uspe sestaviti n različnih.

3. Kratkovidnež

Recimo, da je mreža velika $w \times h$ in da so višine podane v tabeli `v`, pri čemer element `v[y * w + x]` hrani višino celice (x, y). Za začetek si pripravimo pomožni podprogram `NaslednjiKorak`, ki za dani položaj (x, y) ugotovi, v katere smeri lahko očala nadaljujejo pot iz te točke. V zanki moramo pregledati vse štiri možne smeri premika (gor, dol, levo, desno); korake, ki bi nas premaknili ven iz mreže, pri tem seveda preskočimo. Med višinami sosednjih polj si zapomnimo najnižjo, skupaj s tem pa si zapomnimo tudi, v katerih smereh ležijo sosedje s to najnižjo višino. Te smeri bomo vrnili kot celo število od 0 do 15, v katerem vsak od najnižjih štirih bitov ustreza eni od štirih možnih smeri in s svojo vrednostjo pove, ali očala lahko nadaljujejo pot v tisto smer ali ne. Paziti moramo še na možnost, da je najnižji

sosed višji od trenutnega položaja; v tem primeru vrnemo 0, saj naloga pravi, da se očala v takem primeru ustavijo.

```
#include <vector>

using namespace std;
int w, h;
vector<int> v;

// Štiri možne smeri premika so (DX[i], DY[i]) za i = 0, 1, 2, 3.
const int DX[] = { -1, 1, 0, 0 }, DY[] = { 0, 0, -1, 1 };

int NaslednjiKorak(int x, int y)
{
  int vNaslednja, rezultat = 0;
  for (int d = 0; d < 4; d++)
  {
    // Kam bi nas premaknil korak v smeri d?
    int xx = x + DX[d], yy = y + DY[d];

    // Če bi pri tem padli iz mreže, premik ni mogoč.
    if (xx < 0 || xx >= w || yy < 0 || yy >= h) continue;

    // Če je to najnižji sosed doslej, si ga zapomnimo.
    int vSoseda = v[yy * w + xx];
    if (rezultat == 0 || vSoseda < vNaslednja) vNaslednja = vSoseda, rezultat = 0;

    // Zapomnimo si tudi, v katerih smereh se pride do sosedov s tako višino.
    if (vSoseda == vNaslednja) rezultat |= (1 << d);
  }
  // Če so vsi sosedje višji od trenutnega položaja, premik ni mogoč.
  int vZdaj = v[y * w + x];
  return (vNaslednja > vZdaj) ? 0 : rezultat;
}
```

Če bi nas zanimali le podnalogi (a) in (b), bi se dalo gornji podprogram poenostaviti; takrat ne bi bilo treba, da vrača kombinacijo bitov, ki predstavljajo vse možne smeri, pač pa bi lahko vrnil le številko smeri naslednjega koraka, če je ta edina možna; če ni možna nobena smer, bi vrnil na primer -1, če je možnih več smeri, pa -2 (slednje možnosti pri (a) ne potrebujemo, saj smemo tam predpostaviti, da do nje ne bo prišlo).

Razmislimo zdaj o podnalogi (b), saj pri tem ne bo težko spotoma rešiti tudi (a). V zanki kličemo NaslednjiKorak; če je možen premik v natanko eno smer, se premaknemo tja in nadaljujemo po enakem postopku. Ustavimo pa se, ko nam NaslednjiKorak pove, da naslednji korak ni mogoč v nobeno smer (takrat smo očala našli) ali pa da je možnih več smeri (v tem primeru lahko le zaključimo, da pot do očal ni enolična); funkcija vrne logično vrednost, ki pove, ali je očala našla ali ne, parametra x in y pa ob vrnitvi iz funkcije povesta položaj, na katerem se je njena pot ustavila. Razlika med (a) in (b) je le v tem, da pri (a) do primera, ko je možnih več smeri in očal zato ne najdemo, ne more priti.

```
bool NajdiOcala(int &x, int &y)
{
  while (true)
  {
```

```
    // Poglejmo, kakšni so naslednji možni koraki očal.
    int korak = NaslednjiKorak(x, y);

    // Če ni nobenega, se očala tu ustavijo.
    if (korak == 0) return true;

    // Sicer poglejmo, v katero smer gre naslednji korak.
    int d = 0; while (((korak >> d) & 1) == 0) d++;

    // Če je možnih več naslednjih korakov, javimo napako.
    if ((korak >> d) != 1) return false;

    // Sicer se premaknimo v smer naslednjega koraka.
    x += DX[d]; y += DY[d];
  }
}
```

Podnalogo (c) pa lahko rešujemo z iskanjem v širino. Pri tem vzdržujemo vrsto s polji, za katera že vemo, da jih je mogoče doseči, nismo pa še pregledali, kako je mogoče pot z njih nadaljevati. Na vsakem koraku vzamemo prvo polje iz vrste in dodamo vanjo tiste njegove sosede, na katere se lahko očala z njega premaknejo (pri tem si pomagamo s podprogramom NaslednjiKorak). Poleg tega imamo še tabelo, v katero za vsako polje zapišemo, ali smo ga že dosegli in v koliko korakih (od začetnega polja). Tako lahko poskrbimo, da vsako polje dodamo v vrsto le enkrat (ko ga dosežemo prvič) in da ne naredimo več kot n korakov dolgih poti. Naloga na koncu sprašuje po tem, na koliko tako dosegljivih poljih bi se očala potencialno lahko ustavila; to so tista, pri katerih NaslednjiKorak ugotovi, da ni mogoč noben naslednji korak. Ta polja lahko sproti štejemo in na koncu vrnemo njihovo število.

```
#include <utility>

int PrestejDosegljivaPolja(int x0, int y0, int n)
{
  // dolzina[y * w + x] pove dolžino najkrajše poti od (x0, y0) do (x, y);
  // če pot še ni znana, je tam −1.
  vector<int> dolzina; dolzina.resize(w * h, −1);

  // Pripravimo si vrsto za iskanje v širino.
  vector<pair<int, int>> vrsta; int glava = 0;
  // Dodajmo začetni položaj (x0, y0) v vrsto.
  vrsta.emplace_back(x0, y0); dolzina[y0 * w + x0] = true;

  // Preglejmo, kaj vse je dosegljivo v n korakih.
  int rezultat = 0;
  while (glava < vrsta.size())
  {
    auto [x1, y1] = vrsta[glava++]; int d1 = dolzina[y1 * w + x1];

    // Položaj (x1, y1) je dosegljiv v d1 korakih. Kako je mogoče od tam nadaljevati pot?
    int korak = NaslednjiKorak(x1, y1);

    // Če nadaljevanj ni, je to eno od polj, po katerih sprašuje naloga.
    if (korak == 0) ++rezultat;

    // Če je pot že do sem dolga n korakov, je ne smemo nadaljevati.
    if (d1 >= n) continue;

    // Sicer preglejmo vsa možna nadaljevanja poti.
    for (int d = 0; d < 4; d++) if ((korak >> d) & 1)
    {
      int x2 = x1 + DX[d], y2 = y1 + DY[d];
```

```
        // Če do (x2, y2) prej še nismo prišli, si zapomnimo, da je mogoče
        // do njega priti v d1 + 1 korakih, in ga dodajmo v vrsto.
        int &d2 = dolzina[y2 * w + x2];
        if (d2 >= 0) continue; else d2 = d1 + 1;
        vrsta.emplace_back(x2, y2);
      }
    }
    return rezultat;
}
```

4. Omejitve hitrosti

V novi postavitvi znakov morajo znaki stati na tistih odsekih, kjer obstaja možnost, da se je omejitev hitrosti za voznika spremenila, ko je začel voziti po tem odseku. Če gledamo na primer odsek $u \to v$, nas torej zanima, kakšne so omejitve hitrosti na odsekih, ki vodijo v križišče u; če imajo vsi ti odseki enako omejitev kot odsek $u \to v$, potem na slednjem znaka ne potrebujemo, sicer pa ga moramo tam postaviti, tako da bodo vsi vozniki videli novo omejitev, četudi so se v u pripeljali po kakšnem odseku z drugačno omejitvijo.

Podatke o odsekih bomo hranili v seznamu (v spodnji rešitvi je to vektor odseki), pri čemer za vsak odsek hranimo začetno in končno križišče ter omejitev hitrosti. Spotoma, ko beremo podatke, lahko za vsak odsek še pogledamo, če je to tisti na mostu na otok (odsek $1 \to 2$); v tem primeru si zapomnimo omejitev hitrosti na njem, saj je to privzeta omejitev tudi za tiste odseke, kjer ni znaka.

Potem moramo določiti vhodno omejitev hitrosti za vsako križišče (v spodnji rešitvi jih hranimo v vektorju vhodnaOmejitev). V zanki gremo spet čez odseke; pri vsakem določimo omejitev hitrosti (uporabimo tisto z znaka, če slednjega ni, pa vzamemo privzeto omejitev) in ustrezno popravimo podatek o vhodni omejitvi za križišče, v katero pelje trenutni odsek. Če za tisto križišče še ne poznamo vhodne omejitve (ker nismo obdelali še nobenega odseka, ki bi vodil v to križišče), si le zapomnimo omejitev s trenutnega odseka; če že poznamo kakšno omejitev za to križišče in je ta drugačna od tiste na trenutnem odseku, pa si zapomnimo, da so vhodne omejitve v to križišče različne.

Zdaj vemo vse, kar je treba, da lahko izpišemo rezultate. Še enkrat pojdimo v zanki po odsekih in pri vsakem poglejmo, če se njegova omejitev ujema z vhodno omejitvijo za križišče, pri katerem se ta odsek začne; če se ujemata, znaka ne potrebujemo, sicer pa ga.

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

int main()
{
  int n, m; cin >> n >> m;
  // Preberimo podatke o odsekih in določimo privzeto omejitev (iz znaka na mostu).
  struct Odsek { int od, Do, omejitev; };
  vector<Odsek> odseki(m); int privzetaOmejitev;
  for (auto &o : odseki) {
    cin >> o.od >> o.Do >> o.omejitev;
    if (o.od == 1 && o.Do == 2) privzetaOmejitev = o.omejitev; }
```

```
  // Za vsako križišče določimo omejitev na vhodnih odsekih
  // (če je na vseh vhodnih odsekih enaka).
  enum { NEZNANA = −1, RAZLICNA = −2 };
  vector<int> vhodnaOmejitev(n, NEZNANA);
  for (auto &o : odseki)
  {
    // Če na tem odseku ni znaka, velja privzeta omejitev hitrosti.
    if (o.omejitev < 0) o.omejitev = privzetaOmejitev;

    // Ali je to prvi odsek v križišče o.Do?
    auto &vo = vhodnaOmejitev[o.Do − 1];
    if (vo == NEZNANA) vo = o.omejitev;

    // Ali se omejitev na tem odseku kaj razlikuje od tistih na
    // že pregledanih odsekih v križišče o.Do?
    else if (vo != o.omejitev) vo = RAZLICNA;
  }
  // Izpišimo rezultate.
  cout << n << ' ' << m << endl;
  for (auto &o : odseki)
    // Znak potrebujemo, če se omejitev na tem odseku razlikuje od
    // omejitve na vhodnih odsekih v njegovo začetno krajišče (o.od).
    cout << o.od << ' ' << o.Do << ' '
         << (o.omejitev == vhodnaOmejitev[o.od − 1] ? −1 : o.omejitev) << endl;
  return 0;
}
```

5. Nonogram

Stanje mreže lahko predstavimo kar s tabelo (ali vektorjem) nizov, v katerih vsak niz predstavlja eno vrstico mreže, vsak znak takega niza pa eno polje mreže; pri tem nam bo znak `'#'` pomenil črno polje, znak `'.'` pa belo polje. Omejitve po vrsticah predstavimo s seznamom, ki ima toliko elementov, kolikor je vrstic v mreži; vsak element pa naj bo seznam celih števil, ki povedo zahtevane dolžine strnjenih skupin črnih polj v tisti vrstici. Na analogen način predstavimo tudi omejitve po stolpcih.

Razmislimo zdaj o tem, kako preveriti omejitve po vrsticah. Omejitve za eno vrstico so neodvisne od omejitev za kakšno drugo vrstico, zato gremo lahko preprosto v zanki po vrsticah in preverjamo vsako vrstico posebej. Postavimo se na začetek vrstice ($x = 0$) in po vrsti pregledujmo števila iz seznama zahtevanih dolžin strnjenih skupin črnih polj v tej vrstici. Pri vsakem številu se premaknemo z x do naslednjega črnega polja v vrstici in preštejemo, kako dolga je naslednja skupina črnih polj in če se ta dolžina ujema s predpisano. Če se ne ujema, lahko takoj zaključimo, da nonogram ni pravilno izpolnjen, enako pa tudi v primeru, če naslednje črne skupine sploh ni. Če na ta način pridemo do konca seznama zahtevanih dolžin skupin črnih polj za to vrstico, ne da bi našli kakšno napako, moramo potem preveriti le še to, ali so od trenutnega x do konca vrstice vsa polja bela (če niso, to pomeni, da je v trenutni vrstici več skupin črnih polj, kot določajo omejitve, kar je spet napaka).

Omejitve po stolpcih lahko preverjamo na enak način, le da se pač vlogi x-in y-koordinat zamenjata. Da ne bomo imeli dveh blokov skoraj enake izvorne kode, smo namesto tega v spodnji implementaciji uporabili zanko, ki v prvi iteraciji poskrbi za preverjanje po vrsticah in v drugi za preverjanje po stolpcih. Na začetku vsake iteracije mora le primerno nastaviti dimenzije mreže (spremenljivki `U` in `V` v spodnji

rešitvi) ter pri dostopanju do mreže po potrebi obrniti koordinate (v spodnji rešitvi zato skrbi parameter obrni podprograma Crno).

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

bool Preveri(const vector<string>& mreza, const vector<vector<int>>& vrstice,
             const vector<vector<int>>& stolpci)
{
  // Pomožni podprogram, ki pove, ali je neko polje črno.
  auto Crno = [&] (int u, int v, bool obrni) {
    return (obrni ? mreza[v][u] : mreza[u][v]) == '#'; };
  // Naj bo w širina mreže, h pa njena višina.
  int w = mreza[0].length(), h = mreza.size();
  // Preverimo zdaj vse omejitve.
  for (int poStolpcih = 0; poStolpcih <= 1; poStolpcih++)
  {
    auto &omejitve = poStolpcih ? stolpci : vrstice;
    int U = poStolpcih ? w : h, V = poStolpcih ? h : w;
    // Imamo U vrstic/stolpcev in vsak(a) od njih ima V polj.
    for (int u = 0; u < U; u++)
    {
      // Preglejmo u-to vrstico/stolpec.
      int v = 0; // v je naš trenutni položaj v tej vrstici/stolpcu
      for (int d : omejitve[u])
      {
        // Poiščimo naslednji črni blok v tej vrstici/stolpcu.
        while (v < V && ! Crno(u, v, poStolpcih)) v++;
        int vOd = v; while (v < V && Crno(u, v, poStolpcih)) v++;
        // Preverimo, če je ta blok dolg toliko, kot zahtevajo omejitve.
        if (v - vOd != d) return false;
      }
      // Preverimo, da ni v tej vrstici/stolpcu več črnih blokov kot v omejitvah.
      while (v < V && ! Crno(u, v, poStolpcih)) v++;
      if (v < V) return false;
    }
  }
  return true;
}
```

6. Varovanje podatkov

Naloga pravi, da imamo na voljo sprotni seznam vseh datotek za kopiranje na vseh datotečnih strežnikih. Predpostavimo, da so v njem tako datoteke, ki se trenutno že kopirajo (niso pa se še skopirale do konca), kot tiste, ki jih še nismo začeli kopirati (pa jih še bomo). Ko od rezervnega strežnika izvemo, da se je neka datoteka uspešno skopirala nanj, jo izbrišemo iz omenjenega seznama za kopiranje. Predpostavimo še, da nas datotečni strežniki obvestijo, ko se na njih pojavi nova datoteka, ki jo bo treba skopirati; takrat tisto datoteko dodamo na seznam za kopiranje.

Vsakič, ko pride v seznamu za kopiranje do kakšne spremembe (ko pride nanj nova datoteka ali pa ga neka datoteka zapusti, ker je bila uspešno skopirana), se mora naš postopek odločiti, ali bi zdaj začel kakšno novo kopiranje (in od kod in

kam). Če so vsi rezervni strežniki zasedeni (na vsakega se že nekaj kopira), novega kopiranja ne moremo začeti; ravno tako tudi ne, če je seznam za kopiranje prazen ali pa so na njem same take datoteke, ki se že kopirajo. Sicer pa se moramo odločiti, katero izmed datotek, ki še čakajo na kopiranje, bi začeli kopirati (pri tem pridejo v poštev le datoteke s tistih datotečnih strežnikov, s katerih se trenutno nič ne kopira), in na kateri rezervni strežnik (če jih je več takih, na katere se trenutno nič ne kopira) bi jo poslali.

Lahko bi na primer vsakič izbrali tisto datoteko, ki že najdlje čaka v vrsti na začetek kopiranja; poslali pa bi jo na tistega izmed trenutno prostih rezervnih strežnikov, ki ima največjo hitrost kopiranja. Potencialna slabost te rešitve se pokaže v primerih, ko so hitrosti strežnikov zelo različne: mogoče bomo neko zelo veliko datoteko poslali v kopiranje na neki zelo počasen strežnik, ker je ta pač edini, ki je trenutno prost; in mogoče bi bilo bolje, če bi s kopiranjem te datoteke še malo počakali, da se sprosti kak hitrejši strežnik.

Lahko bi torej za vsak rezervni strežnik vzdrževali ločen seznam datotek, ki jih nameravamo poslati v kopiranje nanj. Ko se pojavi nova datoteka, ki jo je treba skopirati, se moramo odločiti, na kateri rezervni strežnik bi jo poslali, in jo dodati v njegov seznam. Za vsak rezervni strežnik lahko iz dolžine datotek v njegovem seznamu in iz hitrosti tega strežnika ocenimo, kdaj bo trenutna datoteka prišla na vrsto in kdaj se bo do konca skopirala, če jo pošljemo na ta strežnik; in po tem razmisleku jo dodamo v seznam tistega rezervnega strežnika, pri katerem kaže, da bo lahko ta datoteka skopirana najbolj zgodaj. Ko nas kak rezervni strežnik obvesti, da je končal s kopiranjem neke datoteke, jo pobrišemo iz njegovega seznama in mu v kopiranje pošljemo naslednjo datoteko iz tistega seznama (tisto, ki že najdlje čaka na kopiranje).

V mislih je dobro imeti še naslednje: to, kdaj bo neka datoteka skopirana, ni odvisno le od rezervnega strežnika, ampak tudi od izvornega datotečnega strežnika; nič nam ne pomaga, če je na voljo neki hiter rezervni strežnik, če pa je mogoče datotečni strežnik, s katerega prihaja ta datoteka, trenutno še zaseden s kopiranjem neke druge datoteke (ki se bo mogoče še dolgo kopirala). Datoteka se torej ne bo mogla začeti kopirati prej kot takrat, ko bo končano kopiranje drugih datotek z istega datotečnega strežnika (tistih, ki čakajo na kopiranje dlje kot ona).

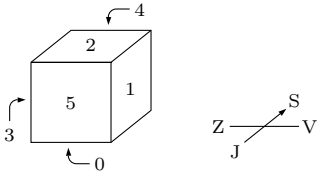
To bi se dalo še izboljšati: recimo, da na nekem datotečnem strežniku čaka na kopiranje neka dolga datoteka, ki jo nameravamo skopirati na neki hiter rezervni strežnik, ko bo le-ta prost; in recimo, da se medtem pojavi na istem datotečnem strežniku še neka druga, kratka datoteka. Mogoče lahko njo skopiramo na neki drug (četudi morda počasnejši) rezervni strežnik, medtem ko prej omenjena dolga datoteka še vedno čaka na kopiranje (pa bo to mogoče še vseeno bolje, kot če bi namesto tega začeli na počasnejši strežnik takoj kopirati tisto dolgo datoteko). Lahko bi torej za vsak strežnik (datotečni in rezervni) vzdrževali seznam časovnih intervalov v prihodnosti, ko bo tisti strežnik predvidoma zaposlen s kopiranjem; naj bo D_i tak seznam za i-ti datotečni strežnik, R_j pa za j-ti rezervni strežnik. Ko pride v kopiranje nova datoteka z i-tega datotečnega strežnika in razmišljamo o tem, da bi jo poslali na j-ti rezervni strežnik, lahko rečemo takole: iz dolžine datoteke in hitrosti rezervnega strežnika lahko ocenimo, da se bo kopirala recimo t sekund; in zdaj v mislih zlijemo seznama D_i in R_j ter na njuni uniji iščemo najzgodnejši prost interval,

dolg vsaj t sekund; konec tega intervala je torej najzgodnejši možni trenutek, ko bi bila ta datoteka lahko skopirana, če jo bomo poslali na rezervni strežnik j. Ko pregledamo vse j, uporabimo tistega, pri katerem je ta čas najzgodnejši, in dodamo ustrezen interval za to datoteko v seznama D_i in R_j.

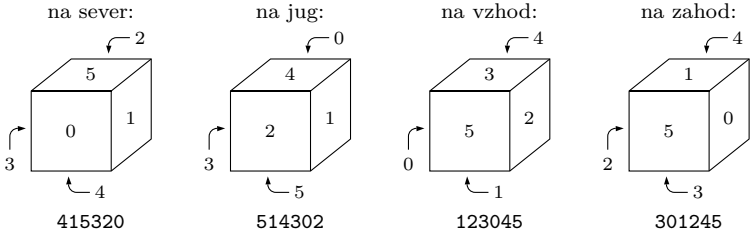
Ko se nek rezervni strežnik sprosti, pogledamo, katera je naslednja datoteka (recimo d) v njegovem seznamu; če je izvorni datotečni strežnik tiste datoteke trenutno še zaseden (ker se je mogoče neko zgodnejše kopiranje izkazalo za bolj zamudno, kot smo sprva računali), naj rezervni strežnik počaka, da izvorni strežnik konča s kopiranji, ki bi jih glede na svoj seznam moral opraviti pred kopiranjem datoteke d. Tu pa že vidimo slabost takšne rešitve: če se hitrosti rezervnih strežnikov skozi čas občutneje spreminjajo, se lahko takšni seznami izkažejo za neugodne; mogoče smo v časih, ko je bil nek strežnik hiter, dodali na njegov seznam veliko datotek za kopiranje, nato pa je njegova hitrost padla in bi bilo zdaj dobro nekaj teh datotek poslati na druge strežnike. Zato lahko vsake toliko časa vse datoteke, ki še čakajo na kopiranje, na novo razporedimo med rezervne strežnike (po enakem razmisleku kot v prejšnjih nekaj odstavkih).

7. Kotaljenje kocke

Recimo, da ploskve kocke oštevilčimo od 0 do 5, kot kaže naslednja slika; 0 je spodnja ploskev, 1 je vzhodna, 2 je zgornja, 3 je zahodna, 4 je severna in 5 je južna.



Razmislimo, kako se ploskve prerazporedijo, če se kocka zakotali prek ene od štirih stranic svoje spodnje ploskve. Po vsakem kotaljenju zapišimo vsebino ploskev v enakem vrstnem redu kot prej: najprej spodnjo, nato vzhodno, nato zgornjo, nato zahodno, nato severno in nazadnje južno:



Pri vsakem kotaljenju se štiri ploskve ciklično zamaknejo za eno mesto (pri kotaljenju na zahod in vzhod so to ploskve 0123, pri kotaljenju na sever in jug pa ploskve 0425), ostali dve pa ostaneta na dosedanjih položajih.

Stanje mreže predstavimo s tabelo oz. vektorjem nizov; če ima mreža w stolpcev in h vrstic, bomo imeli h nizov, dolgih po w znakov. Na začetku naj bodo v nizih vsi znaki pike, kar bo predstavljalo prazna polja.

Stanje kocke, torej to, katera črka je na kateri ploskvi, lahko predstavimo z nizom 6 znakov, pri čemer znak na indeksu i (za $i = 0, \ldots, 5$) predstavlja ploskev i v skladu z zgoraj omenjenim številčenjem ploskev. Naš podprogram se mora sprehoditi v zanki po položajih kocke; vsakega primerja s prejšnjim, da vidi, katera koordinata se je spremenila (x ali y) in ali se je spremenila za $+1$ ali za -1. Iz tega lahko ugotovimo, v katero smer se je kocka zakotalila, in ustrezno preuredimo črke v nizu, ki opisuje stanje kocke. Prvo črko niza — to je tista, ki predstavlja spodnjo ploskev, torej tisto, ki leži na mreži — potem vpišemo v tabelo nizov, ki predstavlja stanje mreže.

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

void KotaljenjeKocke(int w, int h, string crke, const vector<pair<int, int>> &polozaji)
{
  // Na začetku je mreža prazna.
  vector<string> mreza(h, string(w, '.'));
  // Sprehodimo se po poti kocke. (xp, yp) je prejšnji položaj.
  int xp, yp; bool prvi = true;
  for (auto [x, y] : polozaji)
  {
    if (prvi) prvi = false;
    else
    {
      // V katero smer smo se zakotalili?
      const char *p =
        (y == yp - 1) ? "415320" : // na sever
        (y == yp + 1) ? "514302" : // na jug
        (x == xp - 1) ? "301245" : // na zahod
        (x == xp + 1) ? "123045" : // na vzhod
        nullptr; // napaka v vhodnih podatkih
      // Pripravimo novo stanje kocke.
      string noveCrke = string(6, '.');
      for (int i = 0; i < 6; i++) noveCrke[i] = crke[p[i] - '0'];
      crke = move(noveCrke);
    }
    // Vpišimo črko na spodnji ploskvi v trenutno celico mreže.
    mreza[y][x] = crke[0];
    // Trenutni položaj postane prejšnji položaj za naslednjo iteracijo.
    xp = x; yp = y;
  }
  // Izpišimo rezultate.
  for (const auto &s : mreza) cout << s << endl;
}
```

8. Gen

(a) Mislimo si neusmerjen graf na n točkah; za vsako omejitev, ki pravi, da morata biti u-ti in v-ti znak niza enaka, dodajmo v naš graf povezavo med točkama u in v. Zdaj vidimo, da če med dvema točkama obstaja pot, recimo $u_0, u_1, \ldots, u_\ell$, to

pomeni, da mora biti u_0-ti znak niza enak u_1-vemu, ta u_2-emu, ta u_3-emu in tako naprej, torej morajo biti v resnici vsi ti znaki enaki. Če si za neko točko u izberemo, kakšen znak bo na indeksu u v nizu, mora biti potem enak znak tudi na indeksih v za vse tiste v, pri katerih v grafu obstaja pot od u do v. V teoriji grafov taki množici točk pravijo *povezana komponenta* grafa. Za vsako povezano komponento si lahko torej poljubno izberemo enega od štirih možnih znakov, nato pa moramo ta znak uporabiti na vseh indeksih, ki pripadajo tej povezani komponenti. Ugodnih genov je torej 4^k, če je k število povezanih komponent.

Prešteti jih ni težko z iskanjem v širino ali v globino ali čim podobnim. Poiščimo prvo točko, ki je še nismo pripisali nobeni povezani komponenti; zanjo zdaj začnimo novo komponento in preiščimo vse točke, ki so dosegljive iz nje; tudi one pripadajo isti komponenti kot ona. Ta postopek nadaljujemo, dokler še obstaja kakšna točka, ki je še nismo pripisali nobeni komponenti. Spotoma lahko komponentam tudi dodeljujemo zaporedne številke od 0 naprej in si za vsako točko zapomnimo, kateri komponenti pripada (spodnja rešitev vrne to v vektorju komp); to bo prišlo prav pri naslednjih podnalogah.

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

int n;
vector<pair<int, int>> omejitve;

// Funkcija vrne število komponent, v komp[u] pa številko komponente,
// ki ji pripada točka u.
int PovezaneKomponente(vector<int>& komp)
{
  vector<vector<int>> sosede(n);
  for (auto [u, v] : omejitve) { sosede[u].push_back(v); sosede[v].push_back(u); }
  komp.resize(n, -1); int stKomp = 0;
  for (int u = 0; u < n; ++u) if (komp[u] < 0)
  {
    // u je prva točka neke doslej še neznane povezane komponente.
    // Poglejmo, kaj vse je še dosegljivo iz nje.
    stack<int> toDo; toDo.push(u); komp[u] = stKomp;
    while (! toDo.empty()) {
      int v = toDo.top(); toDo.pop();
      // v je dosegljiva iz u, zato so dosegljive tudi v-jeve sosede.
      for (int w : sosede[v]) if (komp[w] < 0) {
        toDo.push(w); komp[w] = stKomp; } }
    ++stKomp;
  }
  return stKomp;
}
```

(b) Videli smo že, da obstaja 4^k ugodnih genov, če je k število povezanih komponent v našem grafu; da jih dobimo, moramo našteti vse možne kombinacije tega, katerega od štirih možnih znakov pripišemo posamezni komponenti. Vsak tak nabor znakov lahko predstavimo kot k-terico števil $(a_1, \ldots, a_k)$, pri čemer število $a_i \in \{0, 1, 2, 3\}$ pove, kateri znak uporabimo za i-to komponento. Ker hočemo gene izpisati v abece-

dnem vrstnem redu, bomo tudi števila od 0 do 3 interpretirali v abecednem vrstnem redu: 0 pomeni znak `A`, število 1 pomeni znak `C`, število 2 znak `G` in število 3 znak `T`.

Recimo zdaj, da primerjamo dva različna nabora znakov, $a = (a_1, \ldots, a_k)$ in $b = (b_1, \ldots, b_k)$, in da je a leksikografsko manjši od b. Na prvih nekaj mestih se mogoče ujemata, prej ali slej pa nastopi prva razlika; recimo, da je to na indeksu i, tam je torej $a_i < b_i$ (saj smo rekli, da je a leksikografsko manjši od b). Izmed tistih točk grafa, ki tvorijo komponento i, naj bo u točka z najmanjšo številko. Potem velja, da točke $0, \ldots, u-1$ pripadajo le komponentam $1, \ldots, i-1$, saj smo v naši rešitvi podnaloge (a) novo komponento začeli vedno pri točki z najmanjšo številko izmed vseh točk, ki jih še nismo dodelili nobeni od dotlej odkritih komponent. Torej se gena (niza dolžine n), ki nastaneta iz izborov a in b — recimo tema nizoma g_a in g_b — ujemata v prvih u znakih, razlikujeta pa se v u-tem znaku. Znak u je določen z i-to komponento našega izbora; ta je pri a manjša kot pri b, saj smo rekli, da je $a_i < b_i$; in zato predstavlja a_i znak, ki je po abecedi pred znakom, ki ga predstavlja b_i; torej bo $g_a[u]$ po abecedi pred $g_b[u]$ in ker je to prvi indeks, na katerem se niza g_a in g_b razlikujeta, bo s tem g_a leksikografsko manjši od g_b.

Tako torej vidimo, da se leksikografski vrstni red naborov $(a_1, \ldots, a_k)$ popolnoma ujema z leksikografskim vrstnim redom ugodnih genov (tega slednjega pa zahteva naša naloga). Vse, kar moramo narediti, je, da naštejemo vse nabore $(a_1, \ldots, a_k)$ v leksikografskem vrstnem redu, za vsakega sestavimo pripadajoči gen in ga izpišemo. Naši nabori so k-terice števil iz $\{0, 1, 2, 3\}$, zato si jih lahko predstavljamo kot k-mestna naravna števila, zapisana v štiriškem zapisu (namesto bolj običajnega desetiškega). Na vsakem koraku moramo táko število povečati za 1, pa dobimo naslednji nabor. Takega števila ni težko povečati za 1: gremo po števkah od desne proti levi, trojke spreminjamo v ničle, ko pa dosežemo prvo ne-trojko, jo povečamo za 1 in se ustavimo. Če je bilo celo število sestavljeno iz samih trojk, pa je bil to zadnji možni izbor in je postopek končan.

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

// Znaki so tu v abecednem vrstnem redu, kar je pomembno.
const char znaki[] = {'A', 'C', 'G', 'T'};
const int stZnakov = sizeof(znaki) / sizeof(znaki[0]);

void NastejGene(const vector<int>& komp, int stKomp)
{
  // izbor[i] pove, da bomo za indekse iz komponente i uporabili znak znaki[i].
  vector<int> izbor(stKomp, 0);

  // V nizu „gen“ bomo sestavili vsakokratni gen, preden ga izpišemo.
  int n = komp.size(); string gen(n, '.');

  // Preglejmo vse možne izbore znakov.
  while (true)
  {
    // Sestavimo in izpišimo gen za trenutni izbor.
    for (int u = 0; u < n; ++u) gen[u] = znaki[izbor[komp[u]]];
    cout << gen << endl;

    // Premaknimo se na naslednji izbor.
```

```
    int u = stKomp − 1; while (u >= 0)
      if (++izbor[u] < stZnakov) break;
      else izbor[u−−] = 0;
    if (u < 0) break;
  }
}
```

Kot parametra ji moramo podati rezultata, ki ju vrne funkcija PovezaneKomponente, torej število komponent in vektor, ki pove, kateri komponenti pripada posamezna točka.

(*c*) Kot smo videli v rešitvi prejšnje podnaloge, nastane m-ti ugodni gen v leksikografskem vrstnem redu iz m-tega nabora števil $(a_1, \ldots, a_k) \in \{0, 1, 2, 3\}^k$, te nabore pa si lahko predstavljamo tudi kot štiriške zapise celih števil od 0 do $4^k - 1$. Število m moramo torej le zapisati v štiriškem zapisu, pa bomo dobili posamezne a_i, vsak od teh pa nam bo povedal, kateri znak uporabiti na tistih mestih v genu, ki pripadajo i-ti povezani komponenti. Pretvoriti m v štiriški zapis je zelo enostavno; v spodnji rešitvi smo to naredili tako, da smo število z zanki delili s 4 in gledali ostanke pri teh deljenjih (tako dobimo števke od desne proti levi), lahko pa bi namesto tega uporabili operatorje na bitih: vsaka števka v štiriškem zapisu ustreza ravno dvema bitoma, torej dvema števkama v dvojiškem zapisu, tako da lahko i-to števko z desne dobimo po formuli (m ≫ (2 * i)) & 3.

```
string MtiUgodniGen(const vector<int>& komp, int stKomp, int m)
{
  string izbor(stKomp, '.');
  for (int i = stKomp − 1; i >= 0; i−−) {
    izbor[i] = znaki[m % stZnakov];
    m /= stZnakov; }
  int n = komp.size(); string gen(n, '.');
  for (int u = 0; u < n; u++)
    gen[u] = izbor[komp[u]];
  return gen;
}
```

9. Okrajšave

Nalogo lahko prevedemo na problem ujemanja v dvodelnem grafu (*bipartite matching*). Definirajmo graf z dvema skupinama točk: leve točke naj predstavljajo besede iz naših naslovov revij, desne pa vse možne okrajšave teh besed. Če je niz t ena od možnih okrajšav besede s, pri čemer se s pojavlja $\#_s$-krat v našem seznamu imen revij, dodajmo med levo točko s in desno točko t povezavo s težo $(|s| - |t|) \cdot \#_s$. Pri tem zapis $|\cdot|$ pomeni dolžino niza med navpičnima črtama (število znakov — pri okrajšavi t šteje sem tudi pika na koncu okrajšave). Teža povezave torej pove, za koliko črk se bo naš seznam skrajšal, če besedo s okrajšamo v t.

Vsak veljaven nabor okrajšav, ki ustreza pogojem naloge, lahko v mislih predstavimo z neko podmnožico povezav našega grafa; povezavo (s, t) vzamemo v to množico natanko tedaj, če je bila v našem naboru okrajšav beseda s okrajšana v t. Ker sme biti v takem naboru posamezna beseda okrajšana le na en način in ker se posamezne okrajšave ne sme uporabljati pri več različnih besedah, to pomeni, da v naši podmnožici povezav ne bosta imeli nobeni dve skupnega krajišča, niti levega

(ker bi to pomenilo, da sta za neko besedo v rabi dve okrajšavi) niti desnega (ker bi to pomenilo, da je bila neka okrajšava uporabljena za dve različni besedi). Taki podmnožici povezav, v kateri nimata nobeni dve povezavi skupnega krajišča, se v teoriji grafov reče *ujemanje* (*matching*).

Vsakemu veljavnemu naboru okrajšav torej ustreza neko ujemanje v našem grafu; hitro pa vidimo, da velja tudi obratno: vsako ujemanje predstavlja nek nabor okrajšav. Dobimo ga tako, da če je v ujemanje vključena neka povezava (s, t), potem besedo s okrajšamo v okrajšavo t, če pa za nek s ni v ujemanju nobene povezave, pustimo to besedo neokrajšano. Ker v ujemanju nobeni dve povezavi nimata skupnega krajišča, ne bo imela nobena beseda dveh različnih okrajšav in nobena okrajšava ne bo uporabljena pri dveh različnih besedah, torej bo naš nabor okrajšav res ustrezal pogojem naloge.

Teža ujemanja je definirana kot vsota tež povezav v njem. Ker nam teža povezave pove, koliko črk prihranimo, če tisto okrajšavo uporabimo v našem seznamu naslovov, nam teža ujemanja pove, koliko črk prihranimo s celotnim naborom okrajšav, ki ga predstavlja to ujemanje. Ker naloga sprašuje po naboru, ki bo kar najbolj skrajšal naš seznam, moramo torej poiskati ujemanje z največjo težo. To lahko naredimo s kakšnim od algoritmov za največji pretok v grafu, na primer Floyd-Fulkersonovim.

10. Beg iz zapora

Opazimo lahko, da je situacija povsem simetrična glede na diagonalo (premico $y = x$); na zgornjem robu zapora potrebujemo prav toliko paznikov kot na desnem robu. Dovolj bo torej, če preštejemo paznike na desnem robu, potem pa njihovo število podvojimo in odštejemo 1 (da ne bomo tistega v zgornjem desnem kotu šteli dvakrat).

Na desnem robu imamo vsekakor paznika v spodnjem desnem kotu $(n, 0)$, glede ostalih pa razmišljajmo takole. Da se poltrak iz $(0, 0)$ seka z desnim robom zapora in to ne ravno v spodnjem desnem kotu, mora iti skozi eno od točk (x, y) za $1 \leq y \leq x \leq n$. Desni rob potem seka v točki $(n, n \cdot y/x)$. Potrebujemo torej toliko različnih paznikov, kolikor je različnih možnih vrednosti y/x.

Zelo naivna rešitev bi bila v času $O(n^4)$ — en par zank gleda vse x in y, drugi par zank pa vse x' in y' ter preverja, ali je y/x enak kakšnemu od že prej obravnavanih ulomkov y'/x', torej takih z manjšim imenovalcem $x' < x$.

```
int Beg1(int n) // O(n^4) časa, O(1) prostora
{
  int r = 1;
  for (int x = 1; x <= n; x++) for (int y = 1; y <= x; y++)
  {
    // Ali smo ulomek z vrednostjo y/x srečali že pri kakšnem manjšem imenovalcu?
    bool nov = true;
    for (int xx = 1; xx < x && nov; xx++) for (int yy = 1; yy <= xx; yy++)
      if (y * xx == yy * x) { nov = false; break; }
    if (nov) r++;
  }
  return 2 * r - 1;
}
```

Malo boljša možnost je tale: če naj bo y/x enak y'/x' za nek manjši imenovalec x', to pomeni, da je $y' = x' \cdot y/x$; vprašati se moramo torej, ali je ta vrednost celoštevilska,

saj imajo naši ulomki vedno celoštevilske števce in imenovalce. Tako potrebujemo znotraj zank po x in y še gnezdeno zanko po x', v njej pa vsakič preverimo, ali je $x' \cdot y/x$ celo število ali ne. Ta rešitev ima časovno zahtevnost $O(n^3)$.

```
int Beg2(int n) // O(n^3) časa, O(1) prostora
{
  int r = 1;
  for (int x = 1; x <= n; x++) for (int y = 1; y <= x; y++)
  {
    // Ali smo ulomek z vrednostjo y/x srečali že pri kakšnem manjšem imenovalcu?
    bool nov = true;
    for (int xx = 1; xx < x && nov; xx++)
      // Če naj bo yy/xx = y/x, mora biti yy = xx * y/x; ali je to celo število?
      if ((xx * y) % x == 0) { nov = false; break; }
    if (nov) r++;
  }
  return 2 * r - 1;
}
```

Še bolje je, če se spomnimo, da če sta dva ulomka enaka, se bosta okrajšala v en in isti ulomek. Lahko bi torej šli po vseh y/x in preverjali, ali je ulomek že okrajšan, torej ali sta si x in y tuja. Če to počnemo z Evklidovim algoritmom, dobimo rešitev s časovno zahtevnostjo $O(n^2 \log n)$.

```
int gcd(int a, int b)
{
  while (a > 0) { int c = b % a; b = a; a = c; }
  return b;
}

int Beg3(int n) // O(n^2 log n) časa, O(1) prostora
{
  int r = 1;
  for (int x = 1; x <= n; x++) for (int y = 1; y <= x; y++)
    // Ali sta si x in y tuja, torej je ulomek y/x že okrajšan?
    if (gcd(x, y) == 1) r++;
  return 2 * r - 1;
}
```

Če si lahko privoščimo nekaj podobnega Eratostenovemu rešetu, kjer v $O(n^2)$ prostora označujemo, katere ulomke smo že spoznali za neokrajšane, gremo lahko po ulomkih in vsakič, ko vidimo kakšnega okrajšanega y/x, označimo vse $(yd)/(xd)$ za neokrajšane (za $d = 2, 3, \ldots$, dokler xd ne preseže n). Tako porabimo le $O(n^2)$ časa.

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

int Beg4(int n) // O(n^2) časa, O(n^2) prostora
{
  int r = 1;
  // okrajsan[x * (x - 1) / 2 + y - 1] pove, ali je ulomek x/y okrajšan ali ne.
  vector<bool> okrajsan(n * (n + 1) / 2, true);
  for (int x = 1; x <= n; x++) for (int y = 1; y <= x; y++)
  {
    if (! okrajsan[x * (x - 1) / 2 + y - 1]) continue;
    r++; // Postavimo paznika.
```

```
    // Označimo, da ulomki (d * y) / (d * x) za d = 2, 3, ... niso okrajšani.
    for (int xx = x + x, yy = y + y; xx <= n; xx += x, yy += y)
      okrajsan[xx * (xx − 1) / 2 + yy − 1] = false;
  }
  return 2 * r − 1;
}
```

Izkaže se, da je pri danem x število primernih y-ov, ki so tuji temu x, enako $\phi(x) = x\prod_p(1-1/p)$, pri čemer gre p po vseh praštevilih, ki delijo x.[7] Torej gremo lahko v eni zanki po x, nato vsak x razcepimo na prafaktorje in izračunamo $\phi(x)$. Sproti, ko gremo z zanko po x-ih, si lahko delamo tudi seznam praštevil do x; da razcepimo x, ga moramo poskusiti deliti s praštevili do $\sqrt{x}$, teh pa je približno $O(\sqrt{x}/\log x)$. Časovna zahtevnost take rešitve je potem $O(n\sqrt{n}/\log n)$.

```
// Vrne število, ki pove, koliko izmed števil {1, 2, ..., a} je tujih številu a.
// V seznamu „prastevila“ morajo biti praštevila vsaj do √a, urejena naraščajoče.
int phi(int a, const vector<int> &prastevila) // O(√a/ log a) časa, O(1) prostora
{
  // V r bo nastal rezultat, φ(a) = a∏_p(1 − 1/p) po vseh a-jevih prafaktorjih p.
  int r = a;
  for (int p : prastevila)
  {
    if (p * p > a) break; // ostala praštevila so prevelika, da bi bila a-jevi prafaktorji
    if (a % p != 0) continue; // p ni a-jev prafaktor

    // Pomnožimo r z (1 − 1/p).
    r = (r / p) * (p − 1);

    // Pobrišimo p iz razcepa a-ja na prafaktorje.
    while (a % p == 0) a /= p;
  }
  // Ker smo šli s praštevili le do √a, ne do a samega, je mogoče,
  // da nam je zdaj v a ostal še zadnji prafaktor (ki ima gotovo stopnjo 1).
  if (a > 1) r = (r / a) * (a − 1);
  return r;
}

int Beg5(int n) // O(n√n/ log n) časa, O(√n/ log n) prostora
{
  vector<int> prastevila;
  int r = 1;
  for (int x = 1; x <= n; x++)
  {
    // Koliko okrajšanih ulomkov y/x nastane pri tem imenovalcu x?
    int fi = phi(x, prastevila);
    r += fi; // Postavimo paznike zanje.

    // Praštevila do √n dodajajmo v seznam.
    if (fi == x − 1 && x * x <= n) prastevila.push_back(x);
  }
  return 2 * r − 1;
}
```

Še boljša rešitev je Eratostenovo rešeto, ki išče praštevila od 1 do n, ko pa kakšno najde (recimo p) in mora iti po njegovih večkratnikih (da jih označi kot sestavljena

[7] Gl. npr. Wikipedijo *s. v.* Euler's totient function.

števila), lahko v neki tabeli še pomnoži njihovo $\phi(x)$ z $1 - 1/p$. Tako nam bodo sproti nastale vse $\phi(x)$ in jih bo treba na koncu le še sešteti. Časovna zahtevnost je torej sorazmerna z $n \sum_{p \le n} 1/p$, pri čemer gre vsota po vseh praštevilih p, manjših od n. Matematiki vedo povedati, da je to reda $O(n \log \log n)$.[8]

```
int Beg6(int n) // O(n log log n) časa, O(n) prostora
{
  int r = 1;
  vector<int> phi(n + 1); for (int x = 1; x <= n; x++) phi[x] = x;
  for (int x = 1; x <= n; x++)
  {
    // Zdaj je phi[x] = x ∏_p(1 − 1/p) po vseh praštevilih p,
    // ki delijo x in so manjša od x. Če je to še vedno enako x (in je x > 1),
    // to pomeni, da je x tudi sam praštevilo in moramo v produkt dodati
    // še člen (1 − 1/x), da dobimo pravo vrednost φ(x) = x − 1.
    if (phi[x] == x && x > 1) −−phi[x];
    // Postavimo paznike zaradi okrajšanih ulomkov y/x.
    r += phi[x];
    // Če je x praštevilo, obdelajmo njegove večkratnike.
    if (phi[x] != x − 1) continue;
    for (int xx = x + x; xx <= n; xx += x)
      phi[xx] = (phi[xx] / x) * (x − 1);
  }
  return 2 * r − 1;
}
```

Če bi nas zanimalo število paznikov le približno oceniti, pa si lahko pomagamo z naslednjo ugotovitvijo iz teorije števil: vsota $r = 1 + \sum_{x=1}^{n} \phi(x)$, ki nam pove število paznikov na eni stranici, je približno enaka $(3/\pi^2)n^2 + O(n \log n)$,[9] torej je skupno potrebno število paznikov $2r - 1$ približno $(6/\pi^2)n^2 \approx 0{,}6079\, n^2$.

11. Okleščeni CSS

Položaj elementov lahko računamo z rekurzijo po drevesu, od zunanjih elementov proti notranjim. Pri tem je koristno vzdrževati dva podatka: na kateri y-koordinati se začne zadnji prednik tipa absolute ali relative in na kateri y-koordinati se začne zadnji sorojenec tipa relative ali static (če takega ni, pa naj bo tam y-koordinata zgornjega roba starša). Prva od teh dveh vrednosti pride prav pri določanju položaja elementov tipa absolute, druga pa pri ostalih elementih. Spodnja rešitev ju prenaša kot parametra yZaAbs in yZaNeAbs pri rekurzivnih klicih.

Ko tako določimo y-koordinato zgornjega roba nekega elementa, se lahko z rekurzivnimi klici lotimo še njegovih otrok. Pri tem tudi seštevamo višine tistih otrok, ki niso tipa absolute; tako nastane vsota, ki jo potrebujemo za izračun višine trenutnega vozlišča, če je le-to imelo prej pri atributu height vrednost auto. Sproti pride ta vsota prav tudi pri določanju položaja tistih otrok, ki niso tipa absolute (pove nam, koliko pod zgornjim robom trenutnega vozlišča se mora začeti naslednji tak otrok).

Preden se vrnemo iz rekurzivnega klica, si višino trenutnega vozlišča nekam zapišimo — spodnja rešitev ima v ta namen atribut trueHeight, kamor skopira height,

[8] Gl. npr. Wikipedijo *s. v.* Meissel-Mertens constant.
[9] Gl. npr. Wikipedijo *s. v.* Totient summatory function.

če le-ta ni auto, sicer pa uporabi višino, dobljeno s seštevanjem višin otrok (razen tistih tipa absolute).

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

enum { AUTO = −1 }; // za atribut height
typedef enum { STATIC, RELATIVE, ABSOLUTE } Position;

struct Element
{
  // Vhodni podatki.
  int height; // višina v pikslih (≥ 0) ali AUTO
  int top;
  Position position;
  vector<Element> otroci;
  // y-koordinato vrha in pravo višino (če je height = AUTO) bomo izračunali mi.
  int y, trueHeight;
  void IzracunajY(int yZaAbs = 0, int yZaNeAbs = 0);
};

void Element::IzracunajY(int yZaAbs, int yZaNeAbs)
{
  // Določimo y-koordinato zgornjega roba trenutnega elementa.
  y = (position == ABSOLUTE ? yZaAbs : yZaNeAbs) + (position == STATIC ? 0 : top);

  // yZaAbsOtroke naj bo y-koordinata, glede na katero se pozicionirajo
  // tisti naši otroci, ki imajo position = ABSOLUTE.
  int visinaOtrok = 0, yZaAbsOtroke = (position != STATIC) ? y : yZaAbs;

  // Preglejmo vse otroke.
  for (Element &otrok : otroci)
  {
    // Z rekurzivnim klicem obdelajmo tega otroka in vse znotraj njega.
    otrok.IzracunajY(yZaAbsOtroke, y + visinaOtrok);

    // Seštevajmo višine otrok (razen tistih s position = ABSOLUTE).
    if (otrok.position != ABSOLUTE) visinaOtrok += otrok.trueHeight;
  }

  // Če je height = AUTO, si zapomnimo višino, izračunano iz višin otrok.
  trueHeight = (height == AUTO) ? visinaOtrok : height;
}
```

Postopek poženemo tako, da na korenskem elementu pokličemo IzracunajY().

12. Krajšanje matematičnega izraza

Recimo, da imamo vhodni niz dolžine n, $s = s[1]s[2]\ldots s[n]$, in da razmišljamo o tem, da bi za okrajšavo uporabili kak podniz dolžine d. Taki podnizi se začnejo na indeksih $\{1, 2, \ldots, n - d + 1\}$ (kasneje pa ne, ker bi podniz potem že štrlel čez konec niza s). Te indekse lahko razdelimo na podmnožice glede na to, če so njim pripadajoči podnizi enaki. Vsaka podmnožica torej predstavlja po en različen podniz dolžine d (s prav takimi podmnožicami smo se v letošnjem biltenu že srečali, pri prvi nalogi za tretjo skupino; gl. str. 63).

Da ugotovimo, za koliko bi se niz skrajšal, če bi za okrajšavo uporabili recimo podniz t (dolžine d), moramo ugotoviti, koliko neprekrivajočih se pojavitev niza t

lahko okrajšamo. To lahko storimo s požrešnim algoritmom. Uredimo indekse, na katerih se začenjajo pojavitve t-ja v s, naraščajoče. Prve pojavitve (recimo, da se začne na indeksu i_1), kot pravi naloga, ne smemo okrajšati. Vzemimo nato izmed naslednjih pojavitev prvo táko, ki se ne prekriva s prvo, torej táko, ki se ne začne prej kot na $i_1 + d$; recimo, da se začne na i_2. Nato vzamemo prvo táko, ki se ne začne prej kot na $i_2 + d$ in tako naprej. Od tega, da bi kdaj vzeli kakšno kasnejšo pojavitev namesto prve primerne (prve take, ki se ne prekriva s prejšnjo uporabljeno pojavitvijo), ne more biti nobene koristi; recimo, da je prva primerna na indeksu j, mi pa bi namesto nje vzeli tisto na nekem $j' > j$; toda če nam tista na j ni všeč, je to lahko le zato, ker se neugodno prekriva z neko kasnejšo; toda z njo se bo tista na j' tudi prekrivala (in bi lahko namesto pojavitve na j' uporabili tisto na j, pa ne bi bili nič na slabšem) ali pa celo ležala čisto desno od nje (pa bi lahko poleg pojavitve na j' uporabili tudi tisto na j in bi rešitev s tem celo še izboljšali) — v obeh primerih torej ni nobene škode od tega, da uporabimo kar pojavitev, ki se začne na j, torej prvo primerno.

Ker vnaprej ne vemo, pri kateri dolžini podniza k bo nastala najboljša rešitev, bomo pregledali vse možne d (od 1 do $\lfloor n/2 \rfloor$, saj pri daljših podnizih ne moreta nastopiti niti dve neprekrivajoči se pojavitvi). Podmnožice z indeksi, na katerih se pojavljajo podnizi dolžine $d + 1$, lahko enostavno izračunamo iz podmnožic za podnize dolžine d. Recimo, da imamo neko tako podmnožico $A = \{i_1, \ldots, i_\ell\}$, ki nam pove, da se neki podniz t dolžine k pojavlja v nizu s tako, da se njegove pojavitve začnejo na indeksih $i_1, \ldots, i_\ell$. Če hočemo ta podniz podaljšati na dolžino $d+1$, bo na koncu pridobil enega od znakov $s[i_1 + d], \ldots, s[i_\ell + d]$. Ti znaki niso nujno vsi enaki, torej moramo množico razbiti glede na ta naslednji znak, pa dobimo podmnožice za take podnize dolžine $d + 1$, ki se začnejo na t. Če na ta način razdrobimo vse podmnožice za podnize dolžine d, bomo dobili sčasoma podmnožice za vse podnize dolžine $d + 1$, saj je mogoče vsak tak podniz dobiti tako, da neki podniz dolžine d podaljšamo za en znak.

Oglejmo si še implementacijo te rešitve v C++u. Za množice, ki za vse podnize določene dolžine povedo, na katerih indeksih se pojavljajo v s, bi lahko uporabili vektor vektorjev; lahko pa imamo namesto tega le dva vektorja, od katerih eden vsebuje indeks prve pojavitve vsakega podniza, drugi pa za vsak indeks i pove, kje v s se naslednjič pojavi tisti podniz trenutne dolžine, ki se začne pri i. Množice smo torej pravzaprav predstavili s seznami v obliki verig (*linked list*), le da so členi povezani z indeksi namesto s kazalci.

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

int NajboljsaOkrajsava(const string& s)
{
  int n = s.length(), naj = n;
  vector<int> prvi, nasl(n), Prvi, Nasl;
  for (int i = 0; i < n - 1; ++i) nasl[i] = i + 1;
  prvi.push_back(0); nasl[n - 1] = -1;
  for (int d = 0; d + 1 <= n / 2; d++)
  {
    // Za vse različne podnize dolžine d so v „prvi“ indeksi njihovih prvih pojavitev;
```

```
    // v „nasl[i]“ pa je indeks naslednje pojavitve tistega podniza dolžine d,
    // ki se začne na indeksu i. Pripravimo podobno razbitje za podnize dolžine d + 1.
    Prvi.clear(); Nasl.clear(); Nasl.resize(n, −1);
    for (int i : prvi)
    {
      // Pojavitve trenutnega podniza (prva se začne na i) bomo razdelili
      // glede na (d + 1)-vo črko; v z[c] naj bo indeks zadnje doslej odkrite
      // take pojavitve, pri kateri je (d + 1)-va črka z.
      int z[26]; for (int c = 0; c < 26; ++c) z[c] = −1;

      // Preglejmo vse pojavitve trenutnega podniza.
      for ( ; i >= 0 && i + d < n; i = nasl[i]) {
        int c = s[i + d] − 'a'; // d-ta črka tega podniza.

        // Če takega z d-to črko „c“ še nismo videli, je to nov podniz
        // dolžine d in ga dodajmo v seznam Prva.
        if (z[c] < 0) Prvi.push_back(i);

        // Sicer ga dodamo na konec ustreznega seznama — prejšnja
        // pojavitev takega podniza je bila na indeksu z[c].
        else Nasl[z[c]] = i;

        // Zapomnimo si to kot zadnjo pojavitev podniza s tem 'c' doslej.
        z[c] = i; }
    }
    swap(prvi, Prvi); swap(nasl, Nasl);

    // Za vsak podniz dolžine d poglejmo, koliko neprekrivajočih se
    // pojavitev lahko najdemo.
    for (int i : prvi)
    {
      // „Do“ je indeks, pri katerem se konča zadnja uporabljena pojavitev.
      int Do = i + d, st = 1; // „st“ je število uporabljenih pojavitev.

      for ( ; i >= 0; i = nasl[i])
        // Trenutno pojavitev (z začetkom na i) uporabimo, če se ne
        // prekriva z zadnjo uporabljeno (ki se konča pri Do − 1).
        if (i > Do) { ++st; Do = i + d; }

      // Uporabili smo „st“ pojavitev, vsaka razen prve se skrajša z d + 1 znakov
      // na 1 znak. Če to pripelje do najboljše rešitve doslej, si jo zapomnimo.
      naj = min(naj, n − (st − 1) * d);
    }
  }
  return naj;
}
```

13. Karte

Slabost te naloge (z vidika računalniških tekmovanj) je, da je bolj matematična kot računalniška. Spomnimo se za začetek na nekaj pojmov iz matematike, ki nam bodo prišli prav: fakulteta $a! = a \cdot (a-1) \cdot \ldots \cdot 2 \cdot 1$, padajoča potenca $a^{\underline{b}} = a \cdot (a-1) \cdot \ldots \cdot (a-b+1) = a!/(a-b)!$ in binomski koeficient $\binom{a}{b} = \frac{a!}{b!(a-b)!} = a^{\underline{b}}/b!$.

Vseh izborov k kart izmed $n = R + B$ kart je $\binom{n}{k}$. Če pa hočemo, da ima izbor natanko r rdečih kart, moramo izbrati teh r kart izmed vseh R rdečih, nato pa še $k-r$ kart izmed B modrih; to lahko torej naredimo na $\#_r := \binom{R}{r} \cdot \binom{B}{k-r}$ načinov. To ima seveda smisel le, če velja $0 \le r \le k$, $r \le R$ in $k - r \le B$, saj sicer sploh ni dovolj rdečih (če je $r > R$) ali črnih kart (če je $k - r > B$), da bi bil tak izbor

mogoč. Omenjene pogoje lahko združimo v $\max\{0, k-B\} \le r \le \min\{k, R\}$, zunaj tega območja pa je $\#_r = 0$.

Število izborov, pri katerih je rdečih kart vsaj m, dobimo potem tako, da seštejemo $\#_r$ po vseh $r \ge m$; recimo tej vsoti $\#_{\ge m} = \sum_{r=m}^{\min\{k,R\}} \#_r$. Če to število ugodnih izborov delimo s številom vseh izborov, torej z $\binom{n}{k}$, dobimo verjetnost, da je v izboru vsaj m rdečih kart, po čemer nas sprašuje naloga.

Tega razmisleka ni težko implementirati v C++:

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

uintmax_t PadPot(int a, int b)
{
  uintmax_t r = 1;
  for (int i = 0; i < b; i++) r *= a - i;
  return r;
}

uintmax_t Fakt(int a) { return PadPot(a, a); }
uintmax_t Binom(int a, int b) { return PadPot(a, b) / Fakt(b); }

double VerjetnostVsajMRdecih(int R, int B, int k, int m)
{
  uintmax_t stUgodnihIzborov = 0;
  for (int r = max(m, k - B); r <= k && r <= R; ++r)
    stUgodnihIzborov += Binom(R, r) * Binom(B, k - r);
  uintmax_t stVsehIzborov = Binom(R + B, k);
  return stUgodnihIzborov / double(stVsehIzborov);
}
```

Pri vsakem r porabimo $O(k)$ časa za izračun obeh binomskih koeficientov, r pa mora iti v najslabšem primeru do k, zato je časovna zahtevnost te rešitve $O(k^2)$. To bi se dalo izboljšati, če upoštevamo, da se binomska koeficienta le počasi spreminjata, ko se r povečuje za 1: $\binom{R}{r} = \binom{R}{r-1}\frac{R-r+1}{r}$ in $\binom{B}{k-r} = \binom{B}{k-(r-1)}\frac{k-r}{B-k+r}$; zato ju ni treba računati vsakič od začetka, ampak lahko le popravimo tista dva iz prejšnje iteracije glavne zanke (pri $r-1$). S tem bi se časovna zahtevnost rešitve zmanjšala na $O(k)$, vendar so vsa ta razmišljanja o časovni zahtevnosti tako ali tako zgrešena, saj do prav velikih k v nobenem primeru ne bomo prišli. Gornja rešitev dela z največjim celoštevilskim tipom, ki ga prevajalnik podpira (**uintmax_t**), kar dandanes praviloma pomeni 64 bitov; in če izbiramo $k = 15$ kart izmed $R + B = 30$ kart, bo v funkciji Binom prišlo do napake, ker bo padajoča potenca $(R+B)^{\underline{k}}$ takrat večja od 2^{64}, torej bo prevelika za naš celoštevilski tip.

To bi se dalo še malo izboljšati. Spomnimo se, da gornja funkcija Binom računa vrednost $\binom{a}{b}$ tako, da najprej izračuna $a^{\underline{b}} = a(a-1)\cdots(a-b+1)$ in to potem deli z $b! = b(b-1)\cdots 2\cdot 1$; lahko bi najprej, preden faktorje v številcu zmnožimo med seboj, pogledali, kako jih je mogoče pokrajšati s tistimi v imenovalcu, tako da bi se to, kar bi od njih ostalo, na koncu zmnožilo točno v $\binom{a}{b}$. Še ena možnost je, da bi si binomske koeficiente pripravili vnaprej v tabeli, pri čemer bi jih računali s seštevanjem: $\binom{a}{b} = \binom{a-1}{b-1} + \binom{a-1}{b}$; tudi tako ne bi bilo treba imeti opravka z vrednostmi, večjimi od $\binom{a}{b}$. Toda s tem ne bi veliko pridobili; pri $R+B = 68$ kartah se prvič lahko zgodi (če vzamemo $31 \le k \le 37$), da je $\binom{R+B}{k} \ge 2^{64}$, torej takrat

število vseh izborov k kart izmed $R + B$ kart postane preveliko za naš celoštevilski tip.

Če bi hoteli računati s celimi števili pri večjem številu kart, bi morali napisati svoje podprograme za delo z velikimi celimi števili (ali pa uporabiti kakšno od knjižnic, ki že obstajajo v ta namen; mimogrede, pri nekaterih drugih programskih jezikih, npr. pythonu ali javi, je takšna funkcionalnost že del jezika oz. njegove standardne knjižnice), takrat pa potem tudi osnovne aritmetične operacije na njih nimajo več konstantne časovne zahtevnosti.

Lažje pa je, če si pri izračunu pomagamo z ne-celimi števili, torej z aritmetiko s plavajočo vejico. Namesto z velikimi števili bomo delali z njihovimi logaritmi; tako imamo na primer $\ln a^{\underline{b}} = \ln\big(a \cdot (a-1) \cdot \ldots \cdot (a-b+1)\big) = \sum_{i=0}^{b-1} \ln(a-i)$ in podobno pri $\ln a!$ in $\ln \binom{a}{b}$. Iz tega potem ni težko izračunati $\ln \#_r = \ln \binom{R}{r} + \ln \binom{B}{k-r}$. Pri izračunu števila vseh ugodnih izborov $\#_{\geq m}$ pa se stvari malo zapletejo; to je definirano kot vsota $\#_r$ za $r \geq m$, mi pa te vsote ne bi radi računali eksplicitno, saj je lahko zelo velika; raje bi ostali pri njenem logaritmu, ki pa ga ne moremo enostavno izračunati iz logaritmov posameznih seštevancev, torej iz $\ln \#_r$. Namesto tega si pomagajmo z dejstvom, da bomo morali vsoto $\#_{\geq m}$ na koncu tako ali tako deliti s številom vseh možnih izborov, $\binom{R+B}{k}$, da bomo dobili verjetnost, po kateri sprašuje naloga; tako imamo $p = \#_{\geq m} / \binom{R+B}{k} = \sum_{r \geq m} \#_r / \binom{R+B}{k} = \sum_{r \geq m} \exp(\ln \#_r - \ln \binom{R+B}{k})$. V tej zadnji vsoti so vsi seštevanci med 0 in 1, tako da težav z velikimi števili ne bo.

```
double VerjetnostVsajMRdecih2(int R, int B, int k, int m)
{
  // Izračunajmo ln (R+B choose k).
  double logStVseh = 0;
  for (int i = 1; i <= k; i++) logStVseh += log(R + B - i + 1) - log(i);

  // Izračunajmo najmanjše in največje možno število rdečih kart v našem izboru.
  int rMin = max(0, k - B), rMax = min(R, k);

  // Izračunajmo ln (R choose rMin) in ln (B choose k-rMin).
  double logBinR = 0, logBinB = 0;
  for (int i = 1; i <= rMin; i++) logBinR += log(R - i + 1) - log(i);
  for (int i = 1; i <= k - rMin; i++) logBinB += log(B - i + 1) - log(i);

  // Pojdimo po vseh možnih r in računajmo log #_r.
  double p = 0;
  for (int r = rMin; r <= rMax; r++)
  {
    if (r > rMin) {

      // Popravimo logBinR z ln (R choose r-1) na ln (R choose r).
      logBinR += log(R - r + 1) - log(r);

      // Popravimo logBinB z ln (B choose k-r+1) na ln (B choose k-r).
      logBinB -= log(B - k + r) - log(k - r + 1); }

    // Če je izbranih vsaj m rdečih kart, prištejmo verjetnost tega izbora k rezultatu.
    if (r >= m) p += exp(logBinR + logBinB - logStVseh); }

  return p;
}
```

Pri predstavitvi števil s plavajočo vejico in računanju z njimi seveda prihaja do

raznih zaokrožitvenih napak. Pri naših poskusih do $R + B = 29$ kart (kjer tudi celoštevilska rešitev še nima težav s prekoračitvijo obsega 64-bitnih celih števil) se rezultati obeh rešitev nikoli niso razlikovali za več kot pribl. 10^{-14}, torej so napake še zelo majhne; pri okrog 150 kartah naraste napaka rešitve s plavajočo vejico do približno 10^{-13}, pri okrog 500 kartah pa do 10^{-12}.

Za konec naloga sprašuje še, kakšen je najmanjši k, pri katerem verjetnost, da bo med izbranimi kartami vsaj m rdečih, doseže ali preseže neko mejo q. Ta verjetnost seveda narašča s k-jem: več kart ko izberemo, bolj verjetno je, da bo med njimi vsaj m rdečih. Vemo na primer, da je pri $k = m - 1$ ta verjetnost gotovo 0, saj sploh ne izberemo m kart; in da je pri $k = B + m$ ta verjetnost gotovo 1, saj četudi izberemo vse črne karte, bomo morali izbrati še vsaj m rdečih; vmes pa lahko najmanjši primerni k poiščemo z bisekcijo.

```
int NajmanjsiKZaVerjetnostQ(int R, int B, int m, double q)
{
  // Verjetnosti, večje od 1, ne moremo doseči pri nobenem k.
  if (q > 1) return -1;

  // Pri vsakem k je verjetnost izbora m rdečih kart ≥ 0;
  // za q = 0 je torej najmanjši primerni k že k = 0.
  if (q <= 0) return 0;

  // Če je m <= 0, nam ni treba izbrati nobene rdeče karte, kar nam bo
  // zagotovo uspelo že pri k = 0 (ko ne izberemo sploh nobene karte).
  if (m <= 0) return 0;

  // Če je m večji od R, je nemogoče izbrati m rdečih kart.
  if (m > R) return -1;

  // Sicer vemo, da je k = m - 1 premajhen (verjetnost izbora vsaj m rdečih kart je tam 0),
  // k = m + B pa dovolj velik (verjetnost izbora vsaj m rdečih kart je takrat 1).
  int k1 = m - 1, k2 = m + B;
  while (k2 - k1 > 1)
  {
    int k = (k1 + k2) / 2;

    // Ali je k dovolj velik ali premajhen?
    if (VerjetnostVsajMRdecih(R, B, k, m) >= q) k2 = k; else k1 = k;
  }
  // Tu je k1 = k2 - 1 in k1 je še premajhen, k2 pa dovolj velik, zato je k2 najmanjši
  return k2;                                                    // primerni k.
}
```

14. Sedežni red

Pojdimo po vrsticah od zadaj naprej, v vsaki vrstici pa od leve proti desni. Pri tem spremljajmo maksimalne strnjene skupine prostih sedežev v trenutni vrstici (maksimalne v tem smislu, da jih ni mogoče še podaljšati, ker na levi in desni mejijo na zaseden sedež ali pa na rob mreže). Če je taka skupina dolga vsaj n in če primerne skupine še nismo našli ali pa ne tako na sredi vrstice, si jo zapomnimo. Ko pridemo do konca vrstice, v kateri smo prvič našli kakšno skupino n sedežev, pa lahko takoj končamo.

Če je trenutna skupina praznih sedežev daljša od n sedežev, moramo še malo poračunati, da vidimo, kje v njej pridemo najbliže sredini vrstice. Recimo, da pripišemo sedežem v vrstici x-koordinate od 0 do $s - 1$; sredina vrstice je potem pri

$x = (s-1)/2$. (To je lahko tudi ne-celo število, če je s sod; s tem ni nič narobe in nam pač pove, da je sredina vrstice na meji med srednjima dvema stoloma.) Podobno, če imamo skupino n praznih sedežev, ki obsega sedeže od $x = k$ do $x = k+n-1$, je sredina te skupine pri $x = k+(n-1)/2$. Položaj te skupine je tem ugodnejši, čim manjša je razdalja med $k+(n-1)/2$ in $(s-1)/2$, torej čim manjša je $|k+(n-1)/2-(s-1)/2|$. Ta izraz je naprej enak $|k-(s-n)/2|$, kar pa je ravno razdalja med k in $(s-n)/2$. Začetek skupine k je torej tem ugodnejši, čim bliže x-koordinate $k^\star := (s-n)/2$ leži.

Če imamo zdaj neko daljšo skupino več kot n praznih sedežev, recimo od ℓ do d, pri čemer je $d-\ell+1 \geq n$, lahko za začetek skupine n praznih sedežev izberemo katerikoli k, za katerega velja $\ell \leq k$ (da se ne začne prezgodaj) in $k+n-1 \leq d$ (da se ne konča prepozno). Možni k-ji so torej tisti z intervala $[\ell, d-n+1]$. Če na tem intervalu leži tudi $k^\star$, je najbolje, če za k vzamemo kar $k^\star$ (če to ni celo število, ga zaokrožimo navzdol, saj sta meji našega intervala celoštevilski, tako da če vsebuje neko ne-celo število, vsebuje tudi obe celi števili, med katerima to ne-celo število leži; navzdol namesto navzgor pa moramo zaokrožiti zato, ker naloga pravi, da imamo raje bolj levo od dveh možnih skupin sedežev, če sta obe enako blizu sredine vrste); sicer pa za k vzemimo tisto krajišče intervala, ki je bližje $k^\star$: če je $k^\star < \ell$, vzemimo $k = \ell$, sicer pa (če je $k^\star > d+n-1$) vzemimo $k = d+n-1$.

Oglejmo si implementacijo te rešitve v C++. Spodnji podprogram vrne logično vrednost, ki pove, ali je našel kakšno skupino n prostih sedežev; koordinati začetka najboljše take skupine pa vrne v `y` in `k`. Številke vrst gredo 0 do $h-1$, pri čemer manjše številke pomenijo vrste bolj zadaj.

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

bool PoisciSedeze(vector<vector<bool>>& prost, int n, int &y, int &k)
{
  int v = prost.size(), s = prost[0].size(); k = -1;
  int dvaKOpt = s - n; // = 2·k*, da ne bo težav z ne-celimi števili
  for (y = 0; y < v; y++)
  {
    for (int x = 0; x < s; )
    {
      if (! prost[y][x]) { ++x; continue; }
      int L = x++; while (x < s && prost[y][x]) x++;

      // Imamo strnjeno skupino prostih sedežev od L do vključno x - 1.
      if (x - L < n) continue; // Skupina je prekratka.

      // Skupina n sedežev znotraj te daljše skupine se lahko začne
      // s sedežem k za kMin ≤ k ≤ kMax.
      int kMin = L, kMax = x - n;

      // Izračunajmo najboljši k na tem intervalu.
      int kKand = (dvaKOpt < 2 * kMin) ? kMin :
                  (dvaKOpt > 2 * kMax) ? kMax : dvaKOpt / 2;

      // Če je to najboljša rešitev doslej, si jo zapomnimo.
      if (k < 0 || abs(2 * kKand - dvaKOpt) < abs(2 * k - dvaKOpt))
        k = kKand;
    }
```

```
    // Če smo v tej vrsti našli kakšno rešitev, lahko končamo.
    if (k >= 0) return true;
  }
  // Če pridemo do sem, to pomeni, da ni nobene skupine n prostih sedežev.
  y = -1; return false;
}
```

Pri težji različici naloge imamo veliko mrežo, zasedenih sedežev pa je malo, zato bi bilo potratno s časom in prostorom, če bi pregledovali vse sedeže v mreži enega za drugim (in jih hranili v tabeli velikosti $v \times s$), tako kot prejšnja rešitev. Namesto tega bomo pregledovali seznam, v katerem naj bodo zasedeni sedeži urejeni po vrstah (od zadaj naprej) in v vsaki vrsti od leve proti desni. Pri vsakem zasedenem sedežu obdelamo skupino praznih sedežev med njim in prejšnjim zasedenim sedežem; če pa prejšnji ni v isti vrsti kot trenutni, moramo pregledati skupino praznih sedežev za prejšnjim v njegovi vrsti, nato morebitno prazno vrsto med prejšnjim in sedanjim zasedenim sedežem (če je vmes več kot ena prazna vrsta, bomo rešitev našli v prvi od njih), nato pa prazne sedeže pred trenutnim v njegovi vrsti.

```
#include <algorithm>

struct Sedez { int y, x; };

bool PoisciSedeze2(int v, int s, vector<Sedez>& zasedeni, int n, int &y, int &k)
{
  y = -1; k = -1;
  if (s < n) return false; // Če so vrste prekratke, je problem nerešljiv.
  int dvaKOpt = s - n;

  // Uredimo zasedene sedeže naraščajoče po vrstah in
  // nato v vsaki vrsti po številkah sedežev.
  sort(zasedeni.begin(), zasedeni.end(), [] (const auto &a, const auto &b) {
    return a.y < b.y || a.y == b.y && a.x <= b.x; });

  // Podprogram, ki obdela skupino prostih sedežev (od L do D v vrsti Y).
  auto Skupina = [=, &y, &k] (int Y, int L, int D)
  {
    if (D - L + 1 < n) return; // Skupina je prekratka.

    // Skupina n sedežev znotraj te daljše skupine se lahko začne
    // s sedežem k za kMin <= k <= kMax.
    int kMin = L, kMax = D - n + 1;

    // Izračunajmo najboljši k na tem intervalu.
    int kKand = (dvaKOpt < 2 * kMin) ? kMin :
                (dvaKOpt > 2 * kMax) ? kMax : dvaKOpt / 2;

    // Če je to najboljša rešitev doslej, si jo zapomnimo.
    if (k < 0 || abs(2 * kKand - dvaKOpt) < abs(2 * k - dvaKOpt))
      k = kKand, y = Y;
  };

  // Da bo postopek preprostejši, si bomo mislili še en zaseden sedež
  // tik pred prvo vrsto in enega takoj za zadnjo vrsto.
  Sedez prej { 0, -1 };
  for (int i = 0; i <= zasedeni.size(); ++i)
  {
    Sedez S = (i < zasedeni.size()) ? zasedeni[i] : Sedez{v - 1, s};
```

```
    // Obdelajmo prazne sedeže na koncu vrste, v kateri je bil prejšnji zasedeni sedež.
    if (S.y > prej.y) { Skupina(prej.y, prej.x + 1, s − 1);
      // S tem je tista vrsta zaključena; če smo našli rešitev, končajmo.
      if (k >= 0) return true; }
    // Če je med prejšnjim in trenutnim zasedenim sedežem kakšna prazna vrsta,
    // bomo v prvi taki gotovo našli rešitev.
    if (S.y > prej.y + 1) { Skupina(prej.y + 1, 0, s − 1); return true; }
    // Obdelajmo prazne sedeže pred trenutnim. Če je trenutni zasedeni sedež prvi
    // v svoji vrsti, so prazni vsi pred njim, sicer pa vsi med prejšnjim in njim.
    Skupina(S.y, (prej.y == S.y ? prej.x + 1 : 0), S.x − 1);
    prej = S;
  }
  return k >= 0;
}
```

15. Sklicevanje

Sklice med členi si lahko predstavljamo kot usmerjen graf, v katerem je po ena točka za vsak člen in po ena povezava za vsak sklic — če se člen u sklicuje na v, imejmo povezavo $u \to v$. Za izračun dolžin verig in preverjanje obstoja ciklov si lahko pomagamo s topološkim urejanjem. Naj bo d_u dolžina najdaljše take verige sklicev, ki se konča s členom u.

Razmišljajmo takole: členi, na katere se ne sklicuje noben drug, gotovo niso del nobenega cikla in najdaljša veriga sklicev, ki se konča pri takem členu, je dolga 1 člen. Členi, na katere se sklicujejo le taki členi, na katere se ne sklicuje noben drug, tudi gotovo niso del nobenega cikla; najdaljša veriga sklicev, ki se konča pri takem členu, je dolga 2 člena. Tako lahko nadaljujemo in v splošnem rečemo: če se na člen v sklicuje neki tak člen u, ki ni na nobenem ciklu in ki ima $d_u = k$, in če se na v ne sklicuje noben tak člen u', ki bi imel $d_{u'} > k$, potem v tudi ni na nobenem ciklu in ima $d_v = k + 1$.

Koristno je torej pregledovati člene po naraščajoči d_u. Vzdrževali bomo vrsto, v katero na začetku dodajmo člene, na katere se ne sklicuje noben drug in ki imajo torej $d_u = 1$. Na vsakem koraku vzemimo po en člen u iz vrste in preglejmo tiste člene v, na katere se u sklicuje; pri vsakem od njih je $d_u + 1$ kandidat za vrednost d_v in si ga zapomnimo, če za v še nismo videli nobene daljše verige. Če za nek v vidimo, da smo zdaj obdelali (vzeli iz vrste) že vse člene, ki se sklicujejo na v, potem vemo, da v ni na ciklu in zanj zdaj poznamo pravo vrednost d_v.

Opazimo lahko, da preden ta postopek doda v vrsto člen v, je že pobral iz vrste vse člene, iz katerih je mogoče priti v v. Če recimo u in v ležita na nekem ciklu, bi moral postopek dodati v vrsto u prej kot v in v prej kot v, kar je nemogoče, torej v resnici ne doda v vrsto nobenega takega člena, ki leži na kakšnem ciklu. Ko se naš postopek ustavi, moramo torej le pogledati, ali je obdelal vse člene ali ne; če jih je, ciklov ni, sicer pa so. Če ciklov ni, imamo takrat za vsak člen v že tudi pravo vrednost d_v, torej dolžine najdaljše verige s koncem pri tem členu, in med temi moramo zdaj izpisati najdaljšo.

Naloga sprašuje še po tem, ali obstajajo sklici naprej, kar je še lažje preveriti; za vsak sklic $u \to v$ moramo le pogledati, ali je $u < v$ — tisto je potem sklic naprej.

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

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

struct Clen
{
  int stVhodnih = 0;      // toliko drugih členov se sklicuje na tega
  vector<int> izhodni;    // na te druge člene se sklicuje ta člen
  int maxVeriga = 1;      // najdaljša veriga s koncem pri tem členu
};

int main()
{
  // Preberimo vhodno besedilo.
  vector<Clen> cleni(1);
  bool vClenu = false, skliciNaprej = false; int trenClen = 0;
  while (true)
  {
    string s; getline(cin, s); // Preberimo naslednjo vrstico.
    if (cin.fail()) break; // Očitno smo bili že na koncu vhodnih podatkov.
    if (s.empty()) { vClenu = false; continue; } // Prazna vrstica med dvema členoma.
    if (! vClenu) { ++trenClen; cleni.emplace_back(); } // Začenja se nov člen.

    // Poiščimo sklice v trenutni vrstici.
    const char *p0 = s.c_str();
    for (const char *p = p0; *p; )
    {
      // Poiščimo naslednji oklepaj.
      if (*p != '(') { ++p; continue; }

      // Preberimo števke za njim.
      const char *r = p + 1; int n = 0;
      while (*r >= '0' && *r <= '9') { n = 10 * n + (*r++ - '0'); }

      // Ali smo imeli vsaj eno števko in za njo zaklepaj?
      if (*r != ')' || r - p <= 1) { p = r; continue; }

      // Ali je to številka na začetku člena (in torej ni sklic)?
      if (p == p0 && ! vClenu) { p = r + 1; continue; }

      // Dodajmo ta člen, če ga še nimamo.
      if (cleni.size() <= n) cleni.resize(n + 1);

      // Zapomnimo si podatke o tem sklicu.
      ++cleni[n].stVhodnih;
      cleni[trenClen].izhodni.push_back(n);

      // Če najdemo kak sklic naprej, si to zapomnimo.
      if (n > trenClen) skliciNaprej = true;
      p = r + 1; // Nadaljujmo za sklicem.
    }
    vClenu = true;
  }
  // Pripravimo topološki vrstni red. Atribut stVhodnih pri vsakem členu
  // nam bo zdaj povedal, koliko izmed tistih členov, ki se sklicujejo na dani
  // člen, še nismo vzeli iz vrste.
  vector<int> vrsta; int glava = 0;
  for (int u = 1; u < cleni.size(); ++u)
    if (cleni[u].stVhodnih == 0) vrsta.push_back(u);
  while (glava < vrsta.size())
  {
```

```
    const auto &U = cleni[vrsta[glava++]];
    for (int v : U.izhodni) {
      auto &V = cleni[v];
      if (--V.stVhodnih == 0) vrsta.push_back(v);
      // Najdaljšo verigo do u lahko podaljšamo v verigo do v;
      // če je to najdaljša veriga do v doslej, si jo zapomnimo.
      V.maxVeriga = max(V.maxVeriga, U.maxVeriga + 1); }
  }
  // Če v topološki vrstni red nismo mogli postaviti vseh členov,
  // to pomeni, da obstajajo cikli.
  if (vrsta.size() < cleni.size() - 1) cout << "Ciklicni sklici obstajajo." << endl;
  else {
    // Če ciklov ni, izpišimo tudi dolžino najdaljše verige.
    int maxVeriga = 0; for (auto &C : cleni) maxVeriga = max(maxVeriga, C.maxVeriga);
    cout << "Ciklicnih sklicev ni. Najdaljsa veriga je dolga "
         << maxVeriga << " clenov." << endl; }
  // Izpišimo še, ali je bilo kaj sklicev naprej.
  cout << "Sklici naprej " << (skliciNaprej ? "" : "ne ") << "obstajajo." << endl;
  return 0;
}
```

16. Urejanje zaimkov

Nalogo lahko rešujemo s topološkim urejanjem, podobno kot prejšnjo. Sestavimo graf, v katerem je po ena točka za vsak stavek, povezavo $u \to v$ pa dodajmo tam, kjer stavek v vsebuje referenco na kakšnega od pojmov, definiranih v stavku u. Obstoj te reference namreč pomeni, da se mora v vrstnem redu, kakršnega zahteva naloga, stavek u pojaviti pred stavkom v. Ko dodajamo stavke enega po enega v naš vrstni red, smemo stavek v dodati v vrstni red šele po tistem, ko smo v vrstni red že dodali vse tiste stavke, na katere se v sklicuje. Recimo, da imamo n stavkov in da so oštevilčeni od 1 do n. Tako dobimo naslednji postopek:

for $v := 1$ **to** n **do** $d_v := 0$;
za vsako povezavo $u \to v$: $d_v := d_v + 1$;
$L :=$ prazen seznam; $Q :=$ prazna množica;
for $v := 1$ **to** n:
  if $d_v = 0$ **then** dodaj v v Q;
while Q ni prazna:
  naj bo v nek stavek iz Q; $(\star)$
  pobriši v iz Q in ga dodaj na konec L;
  za vsako povezavo $v \to u$:
    $d_u := d_u - 1$;
    if $d_u = 0$ **then** dodaj u v Q;

Vrstni red torej tvorimo v seznamu L. Za vsak stavek v vzdržujemo vrednost d_v, ki pove, na koliko takih stavkov, ki jih še nismo dodali v L, se stavek v sklicuje. V množici Q hranimo stavke, ki jim je d_v že padel na 0, nismo pa jih še dodali v vrstni red L. Na vsakem koraku izberemo enega od njih in ga dodamo v L, nato pa ustrezno zmanjšamo vrednosti d_u pri tistih u, ki se sklicujejo na pravkar dodani stavek; če jim pri tem d_u pade na 0, jih dodamo v Q.

Na ta način bomo sčasoma dobili enega od možnih vrstnih redov, pri katerih se vsaka referenca pojavi šele po definiciji (če se nam Q izprazni, še preden smo uspeli dodati v vrstni red L vse stavke, to pomeni, da reference ponekod tvorijo cikle in zato primernega vrstnega reda sploh ni). To, katerega od teh vrstnih redov dobimo, je odvisno od tega, kako smo izbirali $v \in Q$ v vrstici ($\star$), kadar je tam imela množica Q po več kot en element; toda naloga pravi, da moramo najti tisti vrstni red, ki minimizira vsoto razdalj med referencami in pojmi, na katere se sklicujejo. Ker vnaprej ne vemo, kateri v izmed več možnih kandidatov v Q nas bo pripeljal do takšne optimalne rešitve, moramo preizkusiti vse te možne izbire in pri vsaki od njih z rekurzijo nadaljevati postopek; tako sčasoma dobimo vse topološke vrstne rede in lahko pri vsakem izračunamo vsoto razdalj ter si zapomnimo najboljšega.

Predelajmo torej našo gornjo rešitev v rekurzivni podprogram:

podprogram NADALJUJ(L, Q):
  če je Q prazna :
    oceni vrstni red L in če je najboljši doslej, si ga zapomni;
  za vsak $v \in Q$:
    pobriši v iz Q in ga dodaj na konec L;
    za vsako povezavo $v \to u$:
      $d_u := d_u - 1$;
      if $d_u = 0$ **then** dodaj u v Q;
    NADALJUJ(L, Q);
    za vsako povezavo $v \to u$:
      if $d_u = 0$ **then** pobriši u iz Q;
      $d_u := d_u + 1$;
    pobriši v s konca L in ga dodaj v Q;

Glavni del programa bi moral izračunati vhodne stopnje d_v, enako kot prvotna rešitev, in nato poklicati NADALJUJ s praznim seznamom L in z množico $Q = \{v : d_v = 0\}$.

Rekurzija se ustavi, ko je množica Q prazna; takrat lahko izračunamo oceno vrstnega reda L (koristno je prej še preveriti, če so v L sploh prisotni vsi stavki, torej če je dolg n členov — če ni, je to znak, da nekatere reference tvorijo cikel in primeren vrstni red sploh ne obstaja) in si L zapomnimo (npr. v neki globalni spremenljivki), če je to najboljša rešitev doslej. Da izračunamo oceno vrstnega reda, moramo za vsak stavek vedeti, kje v L se nahaja:

  for $i := 1$ **to** n **do** $kje[L[i]] := i$;
  $ocena := 0$;
  za vsako povezavo $u \to v$:
    $ocena := ocena + |kje[u] - kje[v]|$;

Opazimo lahko, da seznama L drugače sploh ne potrebujemo; če pa bi ga mogoče na koncu želeli vrniti ali izpisati, ga lahko še vedno rekonstruiramo iz tabele *kje*. Zato si lahko prihranimo nekaj dela, če že med rekurzijo ves čas delamo s tabelo *kje* namesto s seznamom L. Kjer je prvotna rešitev dodala v na konec L, bi morala naša izboljšana rešitev vpisati novo dolžino L-ja v $kje[v]$, namesto seznama L bi kot parameter prenašali le njegovo dolžino (in tabelo *kje*, če je nimamo v globalni spremenljivki).

Naša rekurzivna rešitev je v najslabšem primeru lahko zelo počasna: za n stavkov lahko obstaja do $n!$ različnih vrstnih redov (če ni nobenih povezav, ki bi nas omejevale) in naša rešitev bi sčasoma zgenerirala in pregledala vse te vrstne rede. Razmislimo o tem, kako jo lahko izboljšamo.

Na začetku vsakega rekurzivnega klica lahko množico točk $V = \{1, \ldots, n\}$ v mislih razdelimo na dva dela: *uporabljene* (tiste, ki smo jih že postavili v seznam L) in *neuporabljene* (ki jih še nismo postavili v L). Tidve podmnožici označimo z V_U in V_N. Naloga rekurzivnega klica je, da poišče vse možne vrstne rede točk iz V_N in z njimi podaljša dosedanji seznam L.

Lahko se zgodi, da isto podmnožico V_U zložimo v vrstni red na več različnih načinov, torej dobimo več različnih L; pri vsakem od njih bi potem izvedli po en rekurzivni klic, ki naj bi poiskal vsa možna nadaljevanja vrstnega reda. Toda ker ima vsak od teh klicev na voljo enako množico neuporabljenih točk V_N, bo vsak od njih našel popolnoma enak nabor možnih nadaljevanj. Z drugimi besedami: to, v kakšne vrstne rede je mogoče zložiti točke iz V_N, ni nič odvisno od tega, v kakšen vrstni red smo zložili točke iz V_U, pač pa le od tega, *katere* točke so to (kajti od tega je odvisno, katere nam potem še ostanejo: $V_N = V \setminus V_U$).

Tudi drugi podatki, s katerimi je delala naša dosedanja rešitev, so odvisni le od V_N. Na primer, stanje tabele d, ki smo jo zgoraj med rekurzijo uporabljali kot nekakšno globalno spremenljivko, vedno sledi naslednjemu pravilu: d_v pove, koliko povezav $u \to v$ kaže v v iz točk $u \in V_N$; množica kandidatov Q pa vedno vsebuje natanko tiste $v \in V_N$, ki imajo $d_v = 0$.

Koristno bi torej bilo, če bi si lahko rezultat naše rekurzivne funkcije pri posameznem V_N nekako zapomnili in ga kasneje ponovno uporabili, če bomo spet naleteli na isti V_N. Toda rezultat naše funkcije je načeloma to, da na vse možne načine podaljša seznam L z vsemi točkami iz V_N in pogleda, kateri od tako podaljšanih seznamov ima najboljšo oceno. Tu pa je seveda res, da ocena najboljšega tako podaljšanega seznama *ni* odvisna le od V_N, ampak tudi od tega, v kakšen vrstni red smo pred tem razporedili točke iz V_U. Toda za vsako povezavo, ki prečka mejo med V_U in V_N v končnem vrstnem redu, torej táko povezavo $u \to v$, ki ima $u \in V_U$ in $v \in V_N$, lahko njen prispevek k oceni razdelimo na tri dele: razdalja med u in zadnjo točko iz V_U; nato 1 kot razdalja med zadnjo točko iz V_U in prvo točko iz V_N; in nato razdalja med prvo točko iz V_N in točko v sámo.

In če rekurzivnemu klicu za V_N ne naložimo, naj računa celotno oceno vsakega končnega vrstnega reda, pač pa le toliko, kolikor v oceno prispevajo tisti deli povezav, ki ležijo znotraj V_N, bo imel zdaj rekurzivni klic pred seboj res nalogo, ki je odvisna le od V_N in nič od vrstnega reda točk iz V_U; preostali del ocene pa bo lahko prištel klicatelj sam, če med rekurzijo vzdržujemo podatek o tem, koliko povezav prečka mejo med V_U in V_N — recimo temu številu p. Tega števila ni težko vzdrževati: ko izberemo neko točko $v \in Q \subseteq V_N$ in jo postavimo v naš nastajajoči vrstni red L, se točka v s tem premakne iz V_N v V_U. Povezave, ki so kazale vanjo (iz že prej uporabljenih točk iz V_U), zato zdaj ne štejejo več v p, povezave pa, ki kažejo iz v (v še vedno neuporabljene točke iz V_N), po novem štejejo v p (prej pa niso); p-ju moramo torej prišteti razliko med izhodno in vhodno stopnjo točke v. Te razlike, recimo jim $\triangle_v$, lahko za vse v izračunamo vnaprej in jih hranimo v neki tabeli.

Tako smo prišli do naslednje rešitve:

```
globalne spremenljivke: tabela d[1..n], množici Q in V_N ter celo število p;
                        slovar h za pomnjenje že izračunanih rezultatov;
funkcija NADALJUJ():
  if V_N = {} then return (0, NIL);
  if imamo v h rezultat za V_N then return h[V_N];
  r* := ∞; (* to bo ocena najboljšega razporeda točk iz V_N *)
  v* := NIL; (* to bo v, pri katerem dosežemo oceno r* *)
  za vsako točko v ∈ Q:
    (* Dodajmo v v naš vrstni red. *)
    pobriši v iz Q in V_N;  p := p + △[v];
    za vsako povezavo v → u:
      d[u] := d[u] − 1;
      if d[u] = 0 then dodaj u v Q;
    (* Z rekurzijo preizkusimo vsa možna nadaljevanja vrstnega reda. *)
    (r, v') := NADALJUJ();
    if p + r < r* then r* := p + r in v* := v;
    (* Pobrišimo v iz našega vrstnega reda. *)
    za vsako povezavo v → u:
      if d[u] = 0 then pobriši u iz Q;
      d[u] := d[u] + 1;
    dodaj v v Q in V_N;  p := p − △[v];
  h[V_N] := (r*, v*); (* shranimo rezultat v slovar h *)
  return (r*, v*);
```

Za vsak v smo torej izvedli po en vgnezden klic, ki vrne oceno najboljšega možnega nadaljevanja vrstnega reda, pri čemer so od povezav, ki se začnejo v v točki v ali levo od nje, upoštevane le razdalje od prve naslednje točke (tiste, ki bo v vrstni red prišla kot naslednja za v). Mi pa moramo našemu klicatelju vrniti oceno, v kateri so upoštevane razdalje od prve točke, ki smo jo postavili mi, torej od v; ta razlika je pomembna pri tistih povezavah, ki se začnejo pri v ali levo od nje, končajo pa se desno od nje; takih povezav je p in pri vsaki moramo njen prispevek k oceni povečati za 1, zato vrednosti, ki jo vrne vgnezdeni klic, prištejemo p. Funkcija NADALJUJ poleg ocene najboljšega razporeda $r^\star$ vrne tudi vrednost v-ja, pri kateri jo je dosegel; to bo prišlo prav kasneje, če nas bo poleg ocene najboljšega vrstnega reda zanimal tudi ta vrstni red sam.

Če se NADALJUJ kliče večkrat pri istem V_N, so takrat tudi vrednosti d, Q in p vsakič enake, zato bo tudi rezultat funkcije pri vseh klicih enak; funkcija zato rezultate shranjuje v slovar h (zanj lahko uporabimo npr. razpršeno tabelo, lahko pa celo navadno tabelo, pri čemer množico V_N predstavimo z n-bitnim številom, v katerem vsak bit pove, ali je neka točka prisotna v V_N ali ne, in taka n-bitna števila uporabljamo kot indekse v tabelo h) in ob vsakem klicu najprej preveri, če ima rezultat že v slovarju, da ga ne bo po nepotrebnem računala še enkrat.

Glavni del programa mora najprej inicializirati globalne spremenljivke, nato pa pognati rekurzijo:

```
for v := 1 to n do d[v] := 0 in △[v] := 0;
za vsako povezavo u → v:
  d[v] := d[v] + 1; △[v] := △[v] − 1; △[u] := △[u] + 1;
```

$Q :=$ prazna množica; $V_N :=$ prazna množica; $p := 0$;
for $v := 1$ **to** n:
  dodaj v v V_N; **if** $d[v] = 0$ **then** dodaj v v Q;
$(r^\star, v) :=$ NADALJUJ();

Tako smo v $r^\star$ dobili oceno najboljšega vrstnega reda; če pa hočemo najti ta vrstni red sam, bo s tem še nekaj dodatnega dela. Pri naši prvotni rešitvi smo na koncu rekurzije imeli pred seboj celoten vrstni red in smo si ga lahko zapomnili, če je bil najboljši doslej; tu pa tega ne moremo, saj NADALJUJ nič ne ve o tistem delu vrstnega reda, ki se nanaša na točke iz V_U — med globalnimi spremenljivkami ni niti seznama L niti tabele *kje*, saj ju NADALJUJ za svoje delo ne potrebuje. Pač pa lahko na koncu, ko imamo v h že shranjene rezultate za vse možne podprobleme, iz njih rekonstruiramo rešitev. Gornji postopek se je končal s tem, da smo dobili v r oceno najboljšega vrstnega reda, v v pa prvo točko tistega vrstnega reda. To točko moramo torej postaviti na začetek našega nastajajočega vrstnega reda L; s tem ona postane uporabljena in jo moramo pobrisati iz V_N; zdaj pa lahko pogledamo v slovar h, kakšna je rešitev za V_N, iz česar bomo izvedeli, katera je naslednja točka v optimalnem vrstnem redu; in tako naprej. Gornji postopek torej nadaljujmo takole:

$L :=$ prazen seznam;
while $v \neq$ NIL:
  dodaj v na konec L-ja; pobriši v iz V_N;
  $(r, v) := h[V_N]$;

Vse poizvedbe v slovar h se nanašajo na take množice V_N, s katerimi se je med rekurzijo že srečal podprogram NADALJUJ, zato so zanje v slovarju gotovo že shranjene rešitve. Na koncu imamo v L najboljši vrstni red (oz. enega od njih, če jih obstaja več enako dobrih), v $r^\star$ pa je še od prej njegova ocena.

17. Goljufije pri telefoniranju

Vhodni seznam uredimo po času, tako da pridejo zapisi za isti polurni interval skupaj. Recimo, da nastopa v vhodnih podatkih n različnih polurnih intervalov; za i-ti polurni interval v tem vrstnem redu naj bo potem L_i množica lokacij, ki se pojavljajo skupaj s tem polurnim intervalom. Vidimo lahko, da je pot, ki jo iščemo, odvisna le od $L_1, L_2, \ldots, L_n$; konkretni časi posameznih L_i na pot nič ne vplivajo, vrstni red zapisov za posamezni polurni interval tudi ne (saj ne vemo, v kakšnem vrstnem redu je uporabnik res obiskal tiste lokacije); in če se pri kakšnem polurnem intervalu ista lokacija pojavi večkrat, lahko vse ne-prve pojavitve zavržemo, saj bo pri najkrajši poti (in to je tisto, kar moramo pri tej nalogi poiskati) uporabnik táko lokacijo gotovo obiskal večkrat zaporedoma, ne da bi vmes šel še kam drugam; zato smemo vsako L_i gledati kot množico namesto kot seznam (v katerem bi se lahko ista lokacija pojavila po večkrat).

Iščemo torej najkrajšo táko pot, ki najprej obišče vse lokacije iz L_1, nato vse iz L_2 in tako naprej. Če bi imeli opravka le z eno množico L_k, bi bil to pravzaprav znani problem trgovskega potnika, le s to razliko, da se mora trgovski potnik na koncu poti vrniti nazaj tja, kjer je svojo pot začel, pri naši nalogi pa se lahko pot konča na kakšni drugi lokaciji, kot pa se je začela. Ker je že problem trgovskega

potnika NP-težak, je naša naloga tudi NP-težka in ne moremo pričakovati, da bomo našli res učinkovito rešitev.

Nalogo lahko rešujemo z dinamičnim programiranjem, pri čemer si zastavimo podprobleme naslednje oblike: naj bo $f(k, A, u)$ dolžina najkrajše take poti, ki najprej obišče vse lokacije iz L_1, nato vse iz L_2, ..., nato vse iz L_{k-1} in končno še vse iz $A \subseteq L_k$, pri čemer kot zadnjo od teh obišče lokacijo $u \in A$.[10] Rezultat, po katerem sprašuje naloga, je potem $\min\{f(n, L_n, u) : u \in L_n\}$, ker moramo obiskati vse lokacije iz vseh n različnih časovnih intervalov in ker nam je vseeno, na kateri lokaciji u (v zadnjem intervalu) se naša pot konča.

Če vsebuje A eno samo lokacijo, $A = \{u\}$, moramo tja priti iz ene od lokacij prejšnjega časovnega intervala, torej L_{k-1}; med temi bomo seveda izbrali tisto, ki nam dá najkrajšo pot:

$$f(k, \{u\}, u) = \min\{f(k-1, L_{k-1}, v) + d(v, u) : v \in L_{k-1}\}.$$

To ima smisel pri $k > 1$; pri $k = 1$ pa prejšnjega intervala ni in se naša pot pri u šele začne, zato ima dolžino 0: $f(1, \{u\}, u) = 0$.

Če pa A vsebuje več kot eno lokacijo, smo morali v u (ki ga moramo obiskati na koncu) očitno priti iz neke druge lokacije iz A, recimo iz v; med temi možnostmi spet izberimo tisto, ki nam dá najkrajšo pot:

$$f(k, A, u) = \min\{f(k, A \setminus \{u\}, v) + d(v, u) : v \in A \setminus \{u\}\}.$$

Kot je običajno pri dinamičnem programiranju, si je koristno vrednosti funkcije f, ko jih enkrat izračunamo, nekje zapomniti, da jih ne bo treba računati ponovno, če jih bomo kasneje še kdaj potrebovali. Računamo jih lahko sistematično po naraščajočih k, pri vsakem k pa od manjših množic A proti večjim. Opazimo lahko, da ko pri nekem k računamo rezultate za množice A z recimo ℓ elementi, potrebujemo le rezultate za množice A z $\ell - 1$ elementi, za manjše A pa ne več, torej jih lahko sproti pozabljamo in tako prihranimo nekaj pomnilnika.

Zapišimo naš postopek s psevdokodo:

for $k := 1$ **to** n **do for** $\ell := 1$ **to** $|L_k|$:
  za vsako $A \subseteq L_k$, ki ima ℓ elementov:
    za vsak $u \in A$:
      if $\ell > 1$ **then**
        $f(k, A, u) = \min\{f(k, A \setminus \{u\}, v) + d(v, u) : v \in A \setminus \{u\}\}$;
      else if $k > 1$ **then**
        $f(k, A, u) = \min\{f(k-1, L_{k-1}, v) + d(v, u) : v \in L_{k-1}\}$;
      else
        $f(k, A, u) = 0$;
  if $\ell > 1$
  then pozabi vrednosti funkcije $f(k, A, u)$ za $|A| = \ell - 1$, $u \in A \subseteq L_k$;
  else pozabi vrednosti funkcije $f(k-1, L_{k-1}, u)$ za $u \in L_{k-1}$;
return $\min\{f(k, L_k, u) : u \in L_k\}$;

[10] Podoben prijem smo uporabili že pri prejšnji nalogi, le da tam stavki niso bili razdeljeni na podmnožice; bilo je tako, kot da bi pri naši tukajšnji nalogi imeli $k = 1$.

Razmislimo še o časovni zahtevnosti tega postopka. Recimo, da imamo v posameznem časovnem intervalu po m lokacij. Potem moramo pri vsakem k (od 1 do n) pregledati 2^m množic A, pri vsaki od teh imamo $|A| = O(m)$ različnih u-jev in pri vsakem od njih moramo v zanki pregledati $O(m)$ različnih v-jev, da izračunamo minimum. Časovna zahtevnost postopka je tako $O(n \cdot m^2 \cdot 2^m)$. To je sicer veliko, vendar še vseeno tudi veliko manj od števila vseh možnih poti; pri vsakem k lahko vrstni red, v katerem obiščemo m lokacij tistega intervala, izberemo na $m!$ načinov, tako da je vseh možnih poti kar $(m!)^n$.

18. Učenje odločitvenih seznamov

Recimo, da imamo seznam z m primeri vhodov in pripadajočih izhodov, pri čemer nam k-ti primer pove, da so bile vrednosti vhodnih spremenljivk po vrsti enake $x_{k1}, x_{k2}, \ldots, x_{kn}$, izhod pa je bil y_k. Naj bo $P_{ic} = \{k : 1 \leq k \leq n, x_{ki} = c\}$ množica tistih primerov, pri katerih ima i-ta spremenljivka vrednost c.

Poskusimo sestaviti prvi pogoj našega odločitvenega seznama; to bo disjunkcija enega ali več členov, pri čemer je vsak člen oblike x_i ali $\neg x_i$. Če v pogoj vključimo člen x_i, mu bodo med drugim ustrezali vsi vhodi iz P_{i1}, če pa v pogoj vključimo člen $\neg x_i$, mu bodo med drugim ustrezali vsi vhodi iz P_{i0}. Če za pogoj uporabimo disjunkcijo več takih členov, pa lahko množico vhodov, ki ustrezajo temu pogoju, opišemo kot unijo nekaj takih P_{ic} (tistih, ki pripadajo členom, iz katerih smo sestavili naš pogoj).

Za posamezno množico P_{ic} se lahko zgodi, da za vse primere v njej velja $y_k = 1$; ali pa, da za vse velja $y_k = 0$; ali pa, da za nekatere velja $y_k = 1$ in za nekatere $y_k = 0$. V prvem primeru bomo rekli, da je P_{ic} *pozitivna*, v drugem, da je *negativna*, in v tretjem, da je *mešana*.

Spomnimo se, da odločitveni seznam vrne isti rezultat b_1 za vse vhode, ki ustrezajo prvemu pogoju. Če hočemo vrniti $b_1 = 1$, smemo torej v pogoj vključiti le take člene, katerih pripadajoča P_{ic} je pozitivna, sicer se obnašanje našega odločitvenega seznama ne bo ujemalo s primeri, ki smo jih dobili. Podobno, če hočemo vrniti $b_1 = 0$, smemo vključiti v pogoj le člene, katerih P_{ic} je negativna. Členov z mešano P_{ic} pa v pogoj sploh ne smemo vključiti.

In če nek člen smemo vključiti v ta pogoj, potem ni nobene koristi od tega, da ga ne bi vključili; kajti če ga ne vključimo, bodo nekateri vhodni primeri, ki bi jih sicer obdelali že v prvem stavku našega odločitvenega seznama, ostali neobdelani in se bomo morali nanje ozirati še v kasnejših stavkih; tam ne bo naše delo zaradi njih nič lažje, kvečjemu imamo lahko zaradi njih bolj zapleten problem (ki bo mogoče zahteval več pogojev), kot bi ga imeli brez njih.

Tako imamo torej za prvi pogoj pravzaprav le dve možnosti: ali bo vrnil $b_1 = 0$ ali $b_1 = 1$; ko se odločimo za to, je tudi že jasno, iz katerih členov moramo sestaviti pogoj (iz vseh, katerih P_{ic} so negativne oz. pozitivne, odvisno od b_1). Primeri, ki jih tako izbrani pogoj pokrije, do kasnejših pogojev ne bodo prišli, zato jih lahko zdaj v mislih pobrišemo iz množic P_{ic}. (Ob tem brisanju se lahko zgodi, da kakšna množica, ki je bila prej mešana, postane pozitivna ali negativna.)

Ko potem razmišljamo o drugem pogoju, lahko za začetek opazimo, da ni nobene koristi od tega, da bi vračal enak rezultat kot prvi, torej da bi bilo $b_2 = b_1$, kajti v tem primeru bi lahko njegove člene dodali kar k prvemu pogoju in naš odločitveni

seznam skrajšali za en pogoj. Pri drugem pogoju torej nimamo nobene izbire: vzeli bomo $b_2 = 1 - b_1$ in vzeli v pogoj vse tiste člene, katerih P_{ic} je zdaj negativna (če je $b_2 = 0$) oz. pozitivna (če je $b_2 = 1$). Primere, ki jih je pokril tako dobljeni drugi pogoj, lahko zdaj tudi pobrišemo iz vseh P_{ic}, ker do kasnejših pogojev ti primeri ne bodo prišli.

Podobno je potem pri tretjem pogoju, kjer je za b_3 smiselno vzeti le $b_3 = 1 - b_2$ in s tem $b_3 = b_1$. Tako nadaljujemo in sestavljamo pogoje, dokler ne pokrijemo vseh primerov ali pa pridemo v stanje, ko so vse preostale množice P_{ic} mešane ali pa prazne. Takšno stanje pomeni, da vsak pogoj, ki pokrije kakšnega od še preostalih primerov, nujno pokrije tako kak primer z $y_k = 1$ kot kak primer z $y_k = 0$; če bi torej tak pogoj uporabili v našem odločitvenem seznamu, bi bili njegovi rezultati gotovo neskladni s primeri.[11]

Vidimo lahko, da je edini trenutek, ko smo imeli v našem razmišljanju sploh kakšno izbiro, tisti, v katerem smo izbrali vrednost b_1. Ker ne vemo, ali nas bo do boljše rešitve pripeljal $b_1 = 0$ ali $b_1 = 1$, bomo morali pač preizkusiti obe možnosti in vrniti boljšo od dobljenih rešitev.

Razmislimo še o tem, kako učinkovito implementirati ta postopek. Za vsako množico P_{ic} je koristno hraniti podatek o tem, koliko njenih elementov ima $y_k = 1$ in koliko $y_k = 0$; tako bomo vedno vedeli, ali je množica pozitivna, negativna ali mešana. Poleg tega imejmo še dva seznama, enega za pozitivne množice P_{ic} in enega za negativne.

Za vsak primer k je koristno hraniti tudi podatek o tem, ali je že pokrit ali ne. Ko se odločimo dodati nov člen v pogoj, moramo iti po vseh primerih iz množice P_{ic}, ki ustreza temu členu; za vsakega od teh primerov pogledamo, če je že pokrit, in če ni, ga zdaj označimo za pokritega in pri vseh ostalih množicah $P_{i'c'}$, ki jim primer tudi pripada, ustrezno zmanjšamo števec elementov, ki imajo takšno vrednost y_k kot pravkar obravnavani element. Po vsaki taki spremembi pogledamo, če se je množica $P_{i'c'}$ s tem spremenila iz mešane v pozitivno ali negativno, in če se je, jo dodamo v ustrezni seznam pozitivnih oz. negativnih množic.

Tadva seznama prideta prav, ko tvorimo pogoj v naslednjem stavku našega odločitvenega seznama. Če bo ta stavek vračal rezultat $b_j = 1$, moramo v pogoj vzeti člene, ki ustrezajo vsem pozitivnim množicam P_{ic}, sicer pa člene za vse negativne množice P_{ic}.

Vsako množico P_{ic} enkrat dodamo v nek pogoj; takrat pregledamo vse njene elemente; skupna cena tega je enaka skupni (začetni) velikosti vseh P_{ic}, torej $n \cdot m$ (vsak od m primerov pride v eno P_{ic} za $i = 1, 2, \ldots, n$). Ko nek element prvič pokrijemo, imamo $O(n)$ dela s tem, da popravimo števce za množice, ki jim ta element pripada; to je skupaj $O(n \cdot m)$. Vsako množico P_{ic} največ enkrat dodamo v seznam pozitivnih ali negativnih in jo nato poberemo iz njega, ko naslednjič tvorimo stavek z rezultatom $b_j = 1$ oz. $b_j = 0$; to je skupaj $O(n)$ dela, toliko je namreč množic P_{ic}. Skupaj je torej časovna zahtevnost tega postopka $O(n \cdot m)$.

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

[11] Primer, kjer nastopi ta težava: imejmo $n = 2$ spremenljivki in štiri primere z vhodi 00, 01, 10, 11, izhod pa naj bo vedno $y_k = x_{k1} \operatorname{xor} x_{k2}$. Hitro se lahko prepričamo, da so vse množice P_{ic} mešane.

```
// Struktura Mnozica predstavlja posamezno množico P_ic.
struct Mnozica
{
  vector<int> primeri; // indeksi k, ki tvorijo to P_ic
  int st[2] = {0, 0};    // st[z] = število primerov k ∈ P_ic, ki imajo y_k = z
};

// Struktura Clen predstavlja en člen v pogoju, namreč „x_i = c“.
struct Clen { int i, c; };

// Struktura Stavek predstavlja en stavek v odločitvenem seznamu:
// „pogoj“ je disjunkcija členov, „rezultat“ pa je vrednost b_j, ki jo
// odločitveni seznam vrne, če je pogoj izpolnjen.
struct Stavek { vector<Clen> pogoj; int rezultat; };

// V „seznam“ sestavi najkrajši tak odločitveni seznam, ki v prvem stavku
// vrne „b“. Če ne obstaja, vrne „false“, sicer pa „true“.
bool OdlocitveniSeznam(const vector<vector<bool>>& x,
                       const vector<bool>& y,
                       int b, vector<Stavek>& seznam)
{
  // Pripravimo množice P_ic.
  int m = x.size(), n = x[0].size();
  vector<Mnozica> P[2];
  P[0].resize(n); P[1].resize(n);
  for (int k = 0; k < m; k++) for (int i = 0; i < n; i++) {
    auto &p = P[x[k][i]][i];
    p.primeri.push_back(k); ++p.st[y[k]]; }

  // Pripravimo seznama pozitivnih in negativnih množic.
  vector<Clen> ciste[2]; // ciste[0] = negativne, ciste[1]= pozitivne
  for (int c = 0; c < 2; c++) for (int i = 0; i < n; i++) {
    auto &p = P[c][i];
    for (int bb = 0; bb < 2; bb++)
      if (p.st[bb] > 0 && p.st[1 - bb] == 0)
        ciste[bb].push_back({i, c}); }

  // Dodajajmo stavke, dokler ne pokrijemo vseh primerov.
  // Vsak naslednji stavek vrača nasprotno vrednost kot prejšnji.
  vector<bool> pokrit(m, false); seznam.clear();
  for (int stNepokritih = m; stNepokritih > 0; b = 1 - b)
  {
    int prejNepokritih = stNepokritih;

    // Obdelajmo člene, ki jih lahko dodamo v ta pogoj.
    for (auto clen : ciste[b])
    {
      auto &p = P[clen.c][clen.i];

      // Primeri iz te P_ic postanejo s tem pokriti (če še niso bili).
      for (int k : p.primeri) if (! pokrit[k])
      {
        pokrit[k] = true; --stNepokritih;

        // Primer k je zdaj pokrit; vrzimo ga iz vseh množic P_i'c', ki jim je pripadal.
        for (int ii = 0; ii < n; ii++) {
          int cc = x[k][ii]; auto &pp = P[cc][ii];

          // Če se množica s tem spremeni iz mešane v pozitivno/negativno,
          // jo dodajmo na ustrezni seznam.
          if (--pp.st[b] == 0 && pp.st[1 - b] > 0)
```

```
                ciste[1 − b].push_back({ii, cc}); }
            }
        }
        // Če nismo mogli pokriti nobenega novega primera, je problem nerešljiv.
        if (prejNepokritih == stNepokritih) return false;
        // Sicer dodajmo novi stavek v seznam.
        seznam.push_back({ciste[b], b});
        ciste[b].clear();
    }
    return true;
}
```

Glavni del programa bi moral klicati OdlocitveniSeznam dvakrat, enkrat z $b = 0$ in enkrat z $b = 1$; če najdeta rešitev oba, moramo obdržati krajšega od obeh seznamov.

Naloge so sestavili: Nino Bašić — okrajšave; Primož Gabrijelčič — omejitve hitrosti; Tomaž Hočevar — sedežni red; Klemen Kenda — taborniki; Vid Kocijan — krajšanje matematičnega izraza, urejanje zaimkov, učenje odločitvenih seznamov; Samo Kralj — beg iz zapora; Mitja Lasič — nonogram, varovanje podatkov, kotaljenje kocke; Matjaž Leonardis — karte; Matjaž Leonardis in Jure Slak — okleščeni CSS; Polona Novak — križci in krožci; Jure Slak — kratkovidnež, sklicevanje; Boštjan Slivnik — gen; Janez Brank — goljufije pri telefoniranju.

NASVETI ZA MENTORJE O IZVEDBI ŠOLSKEGA TEKMOVANJA IN OCENJEVANJU NA NJEM

[Naslednje nasvete in navodila smo poslali mentorjem, ki so na posameznih šolah skrbeli za izvedbo in ocenjevanje šolskega tekmovanja. Njihov glavni namen je bil zagotoviti, da bi tekmovanje potekalo na vseh šolah na približno enak način in da bi ocenjevanje tudi na šolskem tekmovanju potekalo v približno enakem duhu kot na državnem.—*Op. ur.*]

Tekmovalci naj pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom; ocenjevanje teh odgovorov poteka v vsakem primeru tako, da jih pregleda in oceni mentor (in ne npr. tako, da bi se poskušalo izvorno kodo, ki so jo tekmovalci napisali v svojih odgovorih, prevesti na računalniku in pognati na kakšnih testnih podatkih). Pri reševanju si lahko tekmovalci pomagajo tudi z literaturo in/ali zapiski, ni pa mišljeno, da bi imeli med reševanjem dostop do interneta ali do kakšnih datotek, ki bi si jih pred tekmovanjem pripravili sami. Čas reševanja je omejen na 180 minut.

Nekatere naloge kot odgovor zahtevajo program ali podprogram v kakšnem konkretnem programskem jeziku, nekatere naloge pa so tipa „opiši postopek“. Pri slednjih je načeloma vseeno, v kakšni obliki je postopek opisan (naravni jezik, psevdokoda, diagram poteka, izvorna koda v kakšnem programskem jeziku, ipd.), samo da je ta opis dovolj jasen in podroben in je iz njega razvidno, da tekmovalec razume rešitev problema.

Glede tega, katere programske jezike tekmovalci uporabljajo, naše tekmovanje ne postavlja posebnih omejitev, niti pri nalogah, pri katerih je rešitev v nekaterih jezikih znatno krajša in enostavnejša kot v drugih (npr. uporaba perla ali pythona pri problemih na temo obdelave nizov).

Kjer se v tekmovalčevem odgovoru pojavlja izvorna koda, naj bo pri ocenjevanju poudarek predvsem na vsebinski pravilnosti, ne pa na sintaktični. Pri ocenjevanju na državnem tekmovanju zaradi manjkajočih podpičij in podobnih sintaktičnih napak odbijemo mogoče kvečjemu eno točko od dvajsetih; glavno vprašanje pri izvorni kodi je, ali se v njej skriva pravilen postopek za rešitev problema. Ravno tako ni nič hudega, če npr. tekmovalec v rešitvi v C-ju pozabi na začetku **#include**ati kakšnega od standardnih headerjev, ki bi jih sicer njegov program potreboval; ali pa če podprogram main() napiše tako, da vrača **void** namesto **int**.

Pri vsaki nalogi je možno doseči od 0 do 20 točk. Od rešitve pričakujemo predvsem to, da je pravilna (= da predlagani postopek ali podprogram vrača pravilne rezultate), poleg tega pa je zaželeno tudi, da je učinkovita (manj učinkovite rešitve dobijo manj točk).

Če tekmovalec pri neki nalogi ni uspel sestaviti cele rešitve, pač pa je prehodil vsaj del poti do nje in so v njegovem odgovoru razvidne vsaj nekatere od idej, ki jih rešitev tiste naloge potrebuje, naj vendarle dobi delež točk, ki je približno v skladu s tem, kolikšen delež rešitve je našel.

Če v besedilu naloge ni drugače navedeno, lahko tekmovalčeva rešitev vedno predpostavi, da so vhodni podatki, s katerimi dela, podani v takšni obliki in v okviru takšnih omejitev, kot jih zagotavlja naloga. Tekmovalcem torej načeloma ni treba pisati rešitev, ki bi bile odporne na razne napake v vhodnih podatkih.

V nadaljevanju podajamo še nekaj nasvetov za ocenjevanje pri posameznih nalogah.

1. PINi

- Neučinkovite rešitve, ki za vsako šifro *abcd* pregledujejo vse možne kombinacije *wxyz* (tako kot naša prva rešitev), naj dobijo največ 13 točk, če so drugače pravilne.
- Format izpisa pri tej nalogi ni posebej predpisan, pomembno je le, da je iz njega razvidno, katerih šifer *abcd* po postopku iz naloge ni mogoče dobiti. Za morebitne drobne napake pri izpisu (npr. če manjkajo vodilne ničle, tako da se `0042` pomotoma izpiše kot `42`) naj se odšteje največ dve točki.
- Naši rešitvi imata po štiri gnezdene zanke, ki gredo z w, x, y in z od 0 do 9; enako dobro bi bilo tudi, če bi šli z eno zanko od 0 do 9999 in iz tega števca izluščili števke w, x, y in z z deljenjem.
- Za morebitne ne-elegantnosti pri izračunu operacije $\oplus$ naj se rešitvi ne odšteva točk, če je izračun sicer pravilen. Na primer, namesto `b = ((w + x) % 11) % 10` bi lahko naredili `b = (w + x) % 11; if (b == 10) b = 0` in podobno.

2. INTERCAL

- Pri tej nalogi ni poudarek na branju vhodnih podatkov ali izpisu rezultatov, zato je vseeno, kako rešitev počne te stvari. Kot pravi že besedilo naloge, lahko rešitev tudi predpostavi, da dobi ukaze podane v tabeli ali seznamu.
- Naša rešitev se nič ne ozira na obstoječe prefikse „DO“, „PLEASE“ ali „PLEASE DO“ v vhodnih podatkih, ampak jih preprosto poreže in namesto njih vrine nove, ki poskrbijo, da je delež vljudnih ukazov primeren. Nič pa ni narobe, če poskuša tekmovalčeva rešitev uporabiti obstoječe prefikse v ukazih (in npr. popraviti le minimalno število ukazov, da bo delež vljudnih ukazov prišel v predpisane meje), dokler je rezultat v skladu z zahtevami naloge.
- Naša rešitev poskuša pazljivo brisati prefikse od daljših proti krajšim in ko odkrije pravega, z brisanjem preneha. Toda ker naloga pravi, da se te fraze kasneje v ukazih ne pojavljajo več, ni nič narobe, če poskuša tekmovalčeva rešitev brisati bolj agresivno, npr. če po tistem, ko je že pobrisala prefiks „DO“, poskuša vseeno pobrisati še prefiks „PLEASE“; ali pa celo, če poskuša pobrisati vse pojavitve podnizov „DO“ in „PLEASE“ v ukazu.
- Rešitvam, ki ne preverijo možnosti, da je problem nerešljiv (ker sta a in b tako blizu skupaj, da pri nobenem številu vljudnih ukazov ne dobimo deleža vljudnih ukazov na intervalu $[a, b]$), naj se zaradi tega odšteje pet točk.
- Rešitve, ki bi delovale le za primer iz besedila naloge, ali pa le za programe neke fiksne majhne dolžine (npr. pet ukazov, ker ima primer v nalogi le pet ukazov), naj dobijo največ 5 točk.

3. Najdaljša pot

- Pri tej nalogi je poudarek bolj na pravilnosti kot na učinkovitosti. Rešitev s časovno zahtevnostjo $O(w^2h^2)$, če je w širina in h višina mreže, lahko dobi največ 18 točk (če je drugače pravilna), rešitev s časovno zahtevnostjo $O(wh)$ pa vseh 20 točk.

- Kot pravi že besedilo naloge, naj se rešitvam, ki izračunajo dolžino poti le za en začetni položaj (namesto minimuma po vseh možnih začetnih položajih), zaradi tega odšteje 6 točk.

- Iz primerov v besedilu naloge je razvidno, da si pod „dolžino“ poti tu predstavljamo število obiskanih polj, ne števila premikov (ki je za 1 manjše od števila obiskanih polj). Če tekmovalčeva rešitev kot dolžino poti vzame število premikov, naj se ji zaradi tega odšteje največ 2 točki.

- Naš primer rešitve računa premike s pomočjo tabel DX in DY, enako dobro pa je tudi, če ima rešitev za vsako možno smer po en stavek **if**.

- Naloga pravi, da je plošča pravokotna, primer na sliki pa ima kvadratno mrežo. Če tekmovalčeva rešitev predpostavi, da je mreža vedno kvadratna, naj se ji zaradi tega odšteje največ 2 točki.

4. Domače naloge

- Poudarek pri tej nalogi je na ideji, da je treba poiskati d najtežjih nalog in vsako razporediti v svoj dan, ne pa toliko na učinkoviti implementaciji. Dovolj dobro je že, če rešitev uredi vhodno zaporedje $t_1, \ldots, t_n$, da določi najtežjih d nalog. Rešitev se sme tudi sklicevati na algoritme in podatkovne strukture, kakršne najdemo v standardni knjižnici kakšnega programskega jezika.

- Če ima rešitev težave v primeru, ko je $d > n$ (več dni kot nalog), naj se ji zaradi tega odšteje največ štiri točke.

- Rešitvam, ki lahko vračajo napačne rezultate, če ima več nalog enako težavnost, naj se zaradi tega odšteje največ štiri točke. (Na primer: recimo, da imamo $d = 3$ in zaporedje težavnosti $7, 7, 8, 8$. Kakšna neprevidna rešitev bi lahko ugotovila, da ima d-ta najtežja naloga težavnost 7, in pomislila, da lahko torej razdeli zaporedje tako, da poišče prvih $d - 1$ nalog s težavnostjo vsaj 7 in zaporedje prereže po vsaki od njih; tako nastanejo trije dnevi: $[7], [7], [8, 8]$. Ta rešitev je napačna, saj obstaja boljša razdelitev: $[7, 7], [8], [8]$.)

- Ni pa mišljeno, da bi se rešitev kaj ukvarjala s tem, da pride pri predstavitvi realnih števil v računalniku do raznih zaokrožitvenih napak (in da bi mogoče lahko dve težavnosti šteli za enaki že, če se razlikujeta za neko dovolj majhno vrednost).

5. Razbijanje permutacijske šifre

- Rešitve, ki pošljejo napravi $O(n)$ nizov, lahko dobijo največ 15 točk (če so drugače pravilne). Če pošlje napravi manj kot $O(n)$ nizov, lahko dobi vse točke, četudi porabi npr. $O(n^2)$ časa za obdelavo rezultatov (npr. ker za vsakega od n izhodov pregleda vse možne vhode, da ugotovi, kateri vhod se je preslikal vanj).

- Pri tej nalogi ni mišljeno, da bi morali vse vhodne nize pripraviti vnaprej, še preden jih začnemo pošiljati šifrirni napravi. Rešitev sme na primer sestaviti nek vhodni niz, ga poslati šifrirni napravi, potem pa nekako upoštevati od nje prejeti izhodni niz pri sestavljanju naslednjega vhodnega niza. (Kakšne velike koristi od tega sicer ni.)

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. PINi	lažja naloga v prvi skupini
2. INTERCAL	srednje težka naloga v prvi ali lažja v drugi skupini
3. Najd. pot	težja naloga v prvi ali lažja v drugi skupini
4. Dom. nal.	težka naloga v prvi ali srednje težka v drugi skupini
5. Razb. šifre	težja naloga v drugi skupini

Če torej na primer nek tekmovalec reši le eno ali dve lažji nalogi, pri ostalih pa ne naredi (skoraj) ničesar, to še ne pomeni, da ni primeren za udeležbo na državnem tekmovanju; pač pa je najbrž pametno, če na državnem tekmovanju ne gre v drugo ali tretjo skupino, pač pa v prvo.

Podobno kot prejšnja leta si tudi letos želimo, da bi čim več tekmovalcev s šolskega tekmovanja prišlo tudi na državno tekmovanje in da bi bilo šolsko tekmovanje predvsem v pomoč tekmovalcem in mentorjem pri razmišljanju o tem, v kateri težavnostni skupini državnega tekmovanja naj kdo tekmuje.

Zadnja leta na državnem tekmovanju opažamo, da je v prvi skupini izrazito veliko tekmovalcev v primerjavi z drugo in tretjo, med njimi pa je tudi veliko takih z zelo dobrimi rezultati, ki bi prav lahko tekmovali tudi v kakšni težji skupini. Mentorjem zato priporočamo, naj tekmovalce, če se jim zdi to primerno, spodbudijo k udeležbi v zahtevnejših skupinah.

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 tri tretje, v drugi skupini pa eno prvo in tri druge nagrade. Poleg nagrad na državnem tekmovanju v skladu s pravilnikom podeljujemo tudi zlata in srebrna priznanja. Število zlatih priznanj je omejeno na eno priznanje na vsakih 25 udeležencev šolskega tekmovanja (teh je bilo letos 288) in smo jih letos podelili deset. 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 123 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	Kristjan Komloši	3	ŠC Kranj, Str. gimn.	20	20	20	18	20	98
1Z		Peter Lekše	2	Škof. klas. gimn. Lj.	20	20	20	20	18	98
2S	3	Žiga Kralj	2	Vegova Ljubljana	20	16	18	20	20	94
2S		David Panić	5	SŠTS Šiška	20	18	16	20	20	94
3S	5	Petja Furlan	3	ŠC Kranj, STŠ Kranj	20	20	16	19	18	93
3S		Ema Leila Grošelj	4	Gimnazija Vič	20	17	20	17	19	93
3S		Miha Meglič	4	ŠC Kranj, Str. gimn.	20	20	20	20	13	93
S	8	Iztok Bajcar	4	Gimnazija Vič	19	17	20	16	20	92
S	9	Jakob Kralj	1	Gimnazija Vič	20	20	19	18	14	91
S		Luka Peršolja	3	Gimnazija Vič	20	15	20	20	16	91
S	11	Žiga Bradaš	4	SŠTS Šiška	18	14	18	20	19	89
S		Jure Pospišil	3	II. gimnazija Maribor	19	20	16	15	19	89
S		Tim Prapotnik	4	SERŠ Maribor	17	14	20	20	18	89
S		Jaka Velkaverh	3	Gimnazija Vič	20	19	14	20	16	89
S	15	Jernej Avsec	4	ŠC Novo mesto, SEŠTG	20	18	18	15	17	88
S		Gašper Korbar	2	Vegova Ljubljana	15	17	20	20	16	88
S		Tim Thuma	1	Vegova Ljubljana	20	20	18	14	16	88
S	18	Anja Laharnar	3	STPŠ Trbovlje	15	17	15	20	20	87
S		Tim Tisak	2	Vegova Ljubljana	15	18	19	17	18	87
S		Jakob Žorž	9	ZRI	20	16	17	15	19	87
S	21	Domen Kralj	4	Vegova Ljubljana	20	16	18	15	17	86
S		Luka Svenšek	2	SPTŠ Murska Sobota	20	15	20	14	17	86
S	23	Matic Dremelj	1	ZRI + Gimn. in SŠ R. Maistra Kamnik	15	18	18	17	17	85
S		Rok Hladin	2	I. gimnazija v Celju	18	18	12	17	20	85
S		Luka Leskovšek	1	Vegova Ljubljana	15	18	12	20	20	85
S	26	Igor Zevnik	3	Vegova Ljubljana	15	16	20	17	16	84
S	27	Jure Dolar	4	ŠC Celje, Gimn. Lava	17	18	16	16	16	83
S		Timotej Gregorič	5	SŠTS Šiška	19	18	15	11	20	83
S		Nejc Mihelčič	1	Gimnazija Vič	17	16	15	20	15	83
S		Matjaž Vuherer	4	ŠC Celje, Gimn. Lava	15	15	18	15	20	83
S	31	Vid Ošep	9	OŠ Blaža Arniča Luče	20	17	12	15	18	82
S		Timotej Robavs	4	STPŠ Trbovlje	12	18	19	13	20	82
S		Nina Sangawa Hmeljak	4	Gimnazija Vič	15	20	19	9	19	82
S		Katja Schrader	1	ZRI	16	18	16	17	15	82
S	35	Matija Mučič	1	Gimnazija Vič	15	16	14	19	17	81
S		Adrian Šiška	1	Vegova Ljubljana	19	10	20	12	20	81
S	37	Jan Wolf	4	SERŠ Maribor	15	17	20	9	19	80
S	38	Tilen Anzeljc	3	Vegova Ljubljana	10	18	20	15	16	79
S		Anže Kocjančič	4	ŠC Novo mesto, SEŠTG	18	20	18	15	8	79
S		Matija Pilko	3	I. gimnazija v Celju	19	20	8	15	17	79
S		Žan Starašinič	3	ŠC Novo mesto, SEŠTG	19	18	15	9	18	79

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
S	42	Oskar Rotar	9	ZRI	20	14	15	9	20	78
S		Lara Stamać	9	ZRI	15	14	18	15	16	78
S		Aljoša Vertot	3	SPTŠ Murska Sobota	14	17	15	16	16	78
S		Chris Zerjavic	4	SERŠ Maribor	15	19	16	14	14	78
S	46	Gašper Arzenšek	4	ŠC Kranj, Str. gimn.	14	19	16	10	18	77
S		Lan Dolenc	4	ŠC Novo mesto, SEŠTG	15	18	12	16	16	77
S		Vida Mlinar	1	Zavod sv. Frančiška Saleškega, Gimnazija Želimlje	20	10	15	15	17	77
S	49	Luka Lah	4	ŠC Velenje, ERŠ	13	16	16	14	17	76
S		Marko Zupan	4	Gimnazija Kranj	19	18	14	9	16	76
	51	Jure Brenčič Jazbec	5	SŠTS Šiška	10	18	10	20	17	75
		Tim Povše	3	ŠC Velenje, ERŠ	15	14	14	12	20	75
		Anton Luka Šijanec	1	Gimnazija Bežigrad	20	13	13	15	14	75
		Tilen Šket	2	I. gimnazija v Celju	10	19	17	9	20	75
		Matic Šulc	4	SERŠ Maribor	15	19	10	15	16	75
	56	Martin Belušič	1	Gimnazija Bežigrad	14	20	7	16	16	73
		Jošt Smrtnik	1	Gimnazija Vič	20	16	5	12	20	73
	58	Žan Ambrožič	1	Gimnazija Kranj	15	18	16	14	8	71
		Bor Križaj	3	ŠC Nova Gorica, ERŠ	16	14	15	10	16	71
	60	Tilen Juričan	1	ZRI	17	18	10	12	13	70
	61	Elvis Okić	4	STPŠ Trbovlje	15	16	3	15	20	69
		Tina Poštuvan	4	Gimnazija Vič	18	20	2	9	20	69
	63	Klemen Šuštar	4	STPŠ Trbovlje	14	14	4	19	17	68
	64	Mark Muravec	3	Gimnazija Vič	15	12	16	4	20	67
		Janez Sedeljšak	4	ŠC Velenje, ERŠ	20	18	1	19	9	67
		Luka Urbanc	9	ZRI	20	16	5	17	9	67
	67	Ahac Rafael Bela	3	SERŠ Maribor	4	18	16	13	15	66
		Nik Jan Špruk	4	Gimnazija Šiška	15	20	0	15	16	66
	69	Jakob Čerič	1	II. gimnazija Maribor	13	15	1	16	20	65
	70	Rene Klement	3	II. gimnazija Maribor	20	19	15	0	10	64
		Luka Pavčnik	4	ŠC Velenje, ERŠ	15	15	2	15	17	64
	72	Rok Stepić	5	SŠTS Šiška	14	16	16	6	11	63
		Timotej Šušteršič	3	Vegova Ljubljana	15	14	4	16	14	63
	74	Matic Boc	4	Gimnazija Šiška	13	18	0	15	16	62
		Urban Vesel	3	Gimnazija Velenje	15	13	2	15	17	62
	76	Jakob Golubić	1	Gimnazija Vič	15	15	15	0	16	61
	77	Aljaž Marn	1	ZRI	18	18	0	8	16	60
		Nemanja Raduljica	3	ŠC Kranj, STŠ Kranj	14	16	11	4	15	60
	79	Luka Ivančič	2	SPTŠ Murska Sobota	15	16	0	9	19	59
	80	Andraž Šošterič	2	I. gimnazija v Celju	14	12	16	9	7	58
	81	Alen Petek	4	ŠC Celje, Gimn. Lava	13	17	0	15	11	56
	82	Peter Jereb	8	ZRI	18	19	2	0	16	55
	83	Žan Luka Artič	2	ŠC Velenje, ERŠ	15	16	5	10	8	54

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
84	Matej Medvešček	3	ŠC Nova Gorica, ERŠ	10	19	5	8	11	53
	David Čadež	2	ŠC Nova Gorica, ERŠ	12	16	0	7	18	53
86	Daniel Bartolič	3	ŠC Nova Gorica, ERŠ	16	18	0	0	18	52
	Samo Pungaršek Pritržnik	4	ŠC Velenje, ERŠ	15	16	0	6	15	52
	Blaž Matija Samotorčan	3	Gimnazija Vič	10	14	0	9	19	52
89	Nik Pirc	1	Gimnazija Vič	15	14	0	12	9	50
90	Matic Bernot	2	STPŠ Trbovlje	17	16	5	1	10	49
	Gašper Podbregar	3	STPŠ Trbovlje	10	18	0	9	12	49
92	Jaša Krevh	1	Gimnazija Vič	15	16	2	2	12	47
	Anja Rupnik	2	Gimnazija Vič	10	14	12	7	4	47
94	Miha Govedič	2	II. gimnazija Maribor	15	20	2	9	0	46
95	Blaž Mežnar	3	ŠC Velenje, ERŠ	17	15	5	6	0	43
96	Mark Ivanič	2	SPTŠ Murska Sobota	20	16	5	0	0	41
97	Martin Gubina	1	Gimnazija Bežigrad	0	0	0	20	19	39
98	Nik Vodovnik	1	ZRI	20	18	0	0	0	38
99	Peter Voušek	1	Gimnazija Vič	8	11	0	10	8	37
100	Aleks Marinič	4	SPTŠ Murska Sobota	20	16	0	0	0	36
101	Matija Derganc	1	Gimnazija Vič	3	10	1	5	14	33
102	Lovro Lotrič	1	ZRI	18	14	0	0	0	32
103	Aljaž Kokol	3	SERŠ Maribor	15	16	0	0	0	31
	Nejc Kozar	2	SPTŠ Murska Sobota	17	14	0	0	0	31
105	Mark Žnidar	3	Gimnazija Kranj	4	12	0	8	0	24
106	Julijan Zakonjšek	1	I. gimnazija v Celju	0	18	0	0	0	18
107	Nejc Pajenk	1	Gimnazija Poljane	15	0	0	0	0	15
108	Žiga Kovačič	2	II. gimnazija Maribor	2	0	0	0	0	2
109	Jaka Zupanc	2	II. gimnazija Maribor	0	0	0	0	0	0

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Janez Koprivec	4	Gimnazija Vič	20	17	8	16	20	81
2Z	2	Matej Kralj	2	Gimnazija Vič + ZRI	10	18	15	20	13	76
2Z	3	Domen Ogorevc	3	Gimnazija Vič + ZRI	5	20	15	18	15	73
2Z	4	Ella Potisek	3	ZRI	18	15	5	19	15	72
3S	5	Blaž Čerenak	2	I. gimnazija v Celju	15	18	3	20	15	71
3S	6	Mihael Golob	3	Vegova Ljubljana	12	15	4	19	17	67
S	7	Tadej Strah	4	Gimnazija Vič	15	20	0	17	14	66
S	8	Vili Perše	4	STŠ Koper	10	18	0	20	17	65
S		Matej Remc	4	Škof. klas. gimn. Lj.	20	7	5	19	14	65
S	10	Lenart Arvo Kos	4	Vegova Ljubljana	8	17	6	20	13	64
S	11	Eva Juvanc	3	ZRI	20	12	0	20	11	63
S		Urh Robič	4	Škof. klas. gimn. Lj.	15	16	14	10	8	63
S	13	Domen Kastelic	3	ZRI	20	0	3	19	13	55
S	14	Anže Hočevar	2	Gimnazija Vič	12	18	4	20	0	54
S	15	Staš Horvat	4	II. gimnazija Maribor	10	13	0	19	8	50
S	16	Janez Ignacij Jereb	3	ZRI	8	12	3	20	5	48
S		Filip Štamcar	1	ZRI + Gimnazija Vič	3	8	5	20	12	48
	18	Gal Gantar	3	Gimnazija Vič	9	6	5	17	9	46
		Enej Lah	2	Gimnazija Bežigrad	5	9	3	15	14	46
		Luka Skeledžija	4	Gimnazija Vič	12	0	5	20	9	46
	21	Maja Budna	4	Škof. klas. gimn. Lj.	9	20	4	7	5	45
		Gabrijel Pflaum	3	Gimnazija Bežigrad	7	0	4	18	16	45
	23	Jan Hrastnik	4	Gimnazija Vič	10	9	5	7	13	44
		David Ošlaj	4	Škof. klas. gimn. Lj.	10	0	5	19	10	44
	25	Jani Bangiev	4	STŠ Koper	2	20	0	15	3	40
		Nikola Brković	2	Gimnazija Bežigrad	9	12	0	19	0	40
	27	Jaša Knap	3	Gimnazija Bežigrad	9	20	5	0	5	39
	28	Tjaž Eržen	4	Gimnazija Kranj	5	8	3	7	13	36
	29	Gregor Kržmanc	4	Gimnazija Vič	10	6	4	12	0	32
	30	Samo Hribar	3	Gimnazija Bežigrad	8	0	0	10	13	31
	31	Sebastijan Trojer	4	Gimnazija Poljane	7	0	0	18	3	28
	32	Jan Zajc	3	Gimnazija Novo mesto	7	0	5	2	11	25
	33	Elias Podbregar	4	Gimnazija Poljane	7	6	3	0	7	23
		Grega Potočnik	3	ŠC Ravne, SŠ Ravne	5	7	2	6	3	23
	35	Zdenko Šušmelj	4	ŠC Nova Gorica, ERŠ	5	4	3	3	7	22
	36	David Krajnc	2	ŠC Ravne, SŠ Ravne	5	4	5	0	7	21
	37	Matic Kovač	1	ZRI	10	0	0	0	0	10
	38	Matija Pajenk	2	ŠC Ravne, SŠ Ravne	5	0	1	0	0	6
	39	Anej Repnik	2	ŠC Ravne, SŠ Ravne	5	0	0	0	0	5

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Tevž Lotrič	4	Gimnazija Kranj		44	47	64	97	252
1Z	2	Benjamin Bajd	2	ZRI	88	0	50	30	77	245
2Z	3	Jakob Schrader	3	ZRI + Gimnazija Vič	81		100		40	221
2Z	4	Job Petrovčič	4	Gimnazija Vič		0	44	80	94	218
3S	5	Domen Hočevar	3	Gimn. Novo mesto	12	0	39	87	0	138
3S	6	Lan Sevčnikar	3	II. gimnazija Maribor	70	14	25	7		116
S	7	Matija Kocbek	4	I. gimnazija v Celju		0	47	47		94
S	8	Erik Červek Roškarič	1	II. gimnazija Maribor	0	37	25	30		92
S	9	Nadezhda Komarova	2	Gimnazija Bežigrad	0	10	0	80	0	90
S	10	Matija Likar	2	II. gimnazija Maribor	42	0	30	17		89
	11	Luka Lonec	3	II. gimnazija Maribor			0	77		77
	12	Gregor Kovač	4	ZRI	48	0	19	7	0	74
	13	Klemen Klopčič	2	Gimnazija Bežigrad	12		0	60		72
	14	Patrik Žnidaršič	3	Gimnazija Vič	28			27	0	55
	15	Daniel Blažič	3	ZRI + Gimnazija Vič	51			0		51
	16	Tadej Tomažič	3	Vegova Ljubljana			0		34	34
	17	Žan Žnidar	4	Gimnazija Kranj	0		25	7		32
	18	Aleksander Piciga	4	Gimnazija Vič	0			17		17
	19	Aljaž Medič	4	ŠC Kranj, Str. gimn.	15				0	15
	20	Alen Leban	3	ŠC N. Gorica, ERŠ	6			0		6

VRSTNI RED ŠOL

Da bi spodbudili šole k čim večji udeležbi in čim boljšim rezultatom v vseh treh skupinah, smo začeli leta 2018 objavljati tudi vrstni red šol v neke vrste skupnem seštevku. Posamezni šoli prinesejo točke najboljši štirje tekmovalci iz te šole v prvi skupini, najboljši trije v drugi in najboljša dva v tretji skupini. Točke šole so enake vsoti točk njenih tekmovalcev. Točke, ki jih prispeva tekmovalec k vsoti, se izračuna tako, da se delež točk (od vseh možnih točk), ki jih je ta tekmovalec dosegel na tekmovanju, pomnoži z utežjo za skupino, v kateri je tekmoval. Utež za prvo skupino je 100, za drugo skupino 200 in za tretjo skupino 300.

Mesto	Šola	Točke
1	Gimnazija Vič	1090,4
2	Vegova Ljubljana	639,4
3	Gimnazija Bežigrad	546,2
4	I. gimnazija v Celju	495,4
5	II. gimnazija Maribor	488,8
6	Škofijska klasična gimnazija Ljubljana	444
7	Gimnazija Kranj	413,4
8	SŠTS Šiška	341
9	ŠC Novo mesto, SEŠTG	323
10	SERŠ Maribor	322
11	STPŠ Trbovlje	306
12	ŠC Velenje, ERŠ	282
13	ŠC Kranj, Strokovna gimnazija	277
14	ŠC Nova Gorica, ERŠ	276,6
15	SPTŠ Murska Sobota	264
16	ŠC Celje, Gimnazija Lava	222
17	Srednja tehniška šola Koper	210
18	ŠC Kranj, Srednja tehniška šola	153
19	Gimnazija Novo mesto	132,8
20	Gimnazija Šiška	128
21	Gimnazija Poljane	117
22	ŠC Ravne na Koroškem, Srednja šola	100
23	Gimnazija in srednja šola Rudolfa Maistra Kamnik	85
24	OŠ Blaža Arniča Luče	82
25	Zavod sv. Frančiška Saleškega, Gimnazija Želimlje	77
26	Gimnazija Velenje	62

NAGRADE

Za nagrado so najboljši tekmovalci vsake skupine prejeli naslednjo strojno opremo in knjižne nagrade:

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Kristjan Komloši	telefon Samsung Galaxy Note 10 Lite
1	1	Peter Lekše	telefon Samsung Galaxy Note 10 Lite
1	2	Žiga Kralj	telefon Samsung Galaxy A51
1	2	David Panić	telefon Samsung Galaxy A51
1	3	Petja Furlan	telefon Samsung Galaxy A20e
1	3	Ema Leila Grošelj	telefon Samsung Galaxy A20e
1	3	Miha Meglič	telefon Samsung Galaxy A20e
2	1	Janez Koprivec	telefon Samsung Galaxy Note 10 Lite Dasgupta *et al.: Algorithms*
2	1	Matej Kralj	telefon Samsung Galaxy Note 10 Lite Dasgupta *et al.: Algorithms*
2	2	Domen Ogorevc	telefon Samsung Galaxy A51 Dasgupta *et al.: Algorithms*
2	2	Ella Potisek	telefon Samsung Galaxy A51
2	3	Blaž Čerenak	telefon Samsung Galaxy A51
2	3	Mihael Golob	telefon Samsung Galaxy A20e
3	1	Tevž Lotrič	telefon Samsung Galaxy Note 10 Lite Raspberry Pi 4 model B Cormen *et al.: Introduction to Algorithms*
3	1	Benjamin Bajd	telefon Samsung Galaxy Note 10 Lite Raspberry Pi 4 model B Cormen *et al.: Introduction to Algorithms*
3	2	Jakob Schrader	telefon Samsung Galaxy A51 Cormen *et al.: Introduction to Algorithms*
3	2	Job Petrovčič	telefon Samsung Galaxy A51
3	3	Domen Hočevar	telefon Samsung Galaxy A20e
3	3	Lan Sevčnikar	telefon Samsung Galaxy A20e
Off-line naloga — Stiskanje nizov			
	1	Miloš Ljubotina	Raspberry Pi 4 model B
	3	Domen Hočevar	Raspberry Pi 4 model B

SODELUJOČE ŠOLE IN MENTORJI

II. gimnazija Maribor	Mitja Osojnik, Mirko Pešec, Lan Sevčnikar
Gimnazija Bežigrad	Andrej Šuštaršič
Gimnazija in srednja šola Rudolfa Maistra Kamnik	Vinko Kušar
Gimnazija Kranj	Mateja Žepič
Gimnazija Novo mesto	Barbara Strnad
Gimnazija Poljane	Janez Malovrh, Boštjan Žnidaršič
Gimnazija Šiška	Edi Kuklec
Gimnazija Velenje	Ivan Jovan
Gimnazija Vič	Klemen Bajec, Filip Koprivec, Alenka Krapež, Marina Trost
I. gimnazija v Celju	Luka Zlatečan
Osnovna šola Blaža Arniča Luče	Alenka Kos
Srednja elektro-računalniška šola Maribor (SERŠ)	Dušan Fugina, Slavko Nekrep
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Dominik Letnar
Srednja šola tehniških strok Šiška	Tom Kamin, Peter Krebelj, Maruša Perič Vučko, Boris Ribaš
Srednja tehniška in poklicna šola Trbovlje (STPŠ)	Uroš Ocepek
Srednja tehniška šola Koper	Andrej Florjančič
Šolski center Celje, Gimnazija Lava	Karmen Kotnik
Šolski center Kranj, Srednja tehniška šola	Miha Baloh
Šolski center Kranj, Strokovna gimnazija	Gašper Strniša
Šolski center Nova Gorica, Elektrotehniška in računalniška šola (ERŠ)	Aljaž Gec, Marko Marčetič, Tomaž Mavri, Barbara Pušnar, Boštjan Vouk
Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija (SEŠTG)	Albert Zorko, Simon Vovko

Šolski center Ptuj, Elektro in računalniška šola (ERŠ)	Franc Vrbančič
Šolski center Ravne na Koroškem, Srednja šola Ravne	Aleksa Marković
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Miran Zevnik
Škofijska klasična gimnazija Šentvid	Helena Starc Grlj
Vegova Ljubljana	Marko Kastelic, Melita Kompolšek, Nataša Makarovič, Darjan Toth
Zavod sv. Frančiška Saleškega, Gimnazija Želimlje	Benjamin Tomažič

Zavod za računalniško izobraževanje (ZRI), Ljubljana

OFF-LINE NALOGA — STISKANJE NIZOV

Na računalniških tekmovanjih, kot je naše, je čas reševanja nalog precej omejen in tekmovalci imajo za eno nalogo v povprečju le slabo uro časa. To med drugim pomeni, da je marsikak zanimiv problem s področja računalništva težko zastaviti v obliki, ki bi bila primerna za nalogo na tekmovanju; pa tudi tekmovalec si ne more privoščiti, da bi se v nalogo poglobil tako temeljito, kot bi se mogoče lahko, saj mu za to preprosto zmanjka časa.

Off-line naloga je poskus, da se tovrstnim omejitvam malo izognemo: besedilo naloge in testni primeri zanjo so objavljeni več mesecev vnaprej, tekmovalci pa ne oddajajo programa, ki rešuje nalogo, pač pa oddajajo rešitve tistih vnaprej objavljenih testnih primerov. Pri tem imajo torej veliko časa in priložnosti, da dobro razmislijo o nalogi, preizkusijo več možnih pristopov k reševanju, počasi izboljšujejo svojo rešitev in podobno. Opis naloge in testne primere smo objavili novembra 2019, nekaj mesecev po razpisu za tekmovanje v znanju; tekmovalci so imeli čas do 27. marca 2020 (dan pred tekmovanjem), da pošljejo svoje rešitve.

Opis naloge

Dan je niz znakov s, ki ga sestavljajo same male črke angleške abecede. Tak niz lahko včasih opišemo kot stik več krajših nizov, pri čemer se smejo posamezni krajši nizi pojavljati po večkrat. Na primer: `zaplapolalo` lahko zapišemo kot `zap` + `la` + `po` + `la` + `lo` (niz `la` se pojavlja dvakrat) ali pa kot `z` + `ap` + `l` + `ap` + `olalo` (niz `ap` se pojavlja dvakrat) ali še na razne druge načine.

Pri tej nalogi bomo ocenili takšno razbitje niza na podnize tako, da bomo sešteli dolžine vseh različnih nizov v razbitju in tej vsoti prišteli število vseh nizov v razbitju. Za drugega izmed zgornjih primerov, `zaplapolalo` = `z` + `ap` + `l` + `ap` + `olalo`, imamo na primer v razbitju pet nizov (`z`, `ap`, `l`, še en `ap` in na koncu še `olalo`), različni pa so štirje (`z`, `ap`, `l`, `olalo`) s skupno dolžino $1+2+1+5=9$. Ocena tega razbitja je torej $9+5=14$.

Tvoja naloga je za dan vhodni niz poiskati čim boljše razbitje na podnize (torej tako, pri katerem je ocena po definiciji iz prejšnjega odstavka čim nižja).

Še en primer: recimo, da imamo niz s = `kanetkananet`. Nekaj možnih rešitev je potem:

- `k` + `anet` + `k` + `an` + `anet` z oceno $(1+4+2)+5=12$;
- `kan` + `et` + `kan` + `an` + `et` z oceno $(3+2+2)+5=12$;
- `kanetkananet` z oceno $(12)+1=13$;
- `k` + `ane` + `tkan` + `ane` + `t` z oceno $(1+3+4+1)+5=14$.

Možne so seveda tudi še drugačne rešitve. Med zgornjimi štirimi sta najboljši prvi dve (z oceno 12). Kot vidimo iz tretje alineje, je možna rešitev tudi ta, da niza sploh ne razbijemo na več krajših.

Rezultati

Sistem točkovanja je bil tak kot pri off-line nalogah v prejšnjih letih. Pripravili smo 10 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 10 testnih primerih.

Nizi v testnih primerih so bili dolgi od približno 635 tisoč do 5,2 milijona znakov, večina pa jih je bila dolga okrog 1 milijon znakov. Nekateri so bili besedila v naravnih jezikih (iz katerih smo pobrisali vse znake, ki niso bili črke), nekatere pa smo zgenerirali naključno.

Letos je svoje rešitve pri off-line nalogi poslalo pet tekmovalcev, od tega dva srednješolca in trije študentje. Končna razvrstitev je naslednja:

Mesto	Ime	Letnik	Šola	Točke
1	Miloš Ljubotina	2	FE	91
2	Samo Kralj	5	FMF	79
3	Domen Hočevar	3	Gimn. Novo mesto	74
4	Matej Kralj	2	ZRI	70
5	Gregor Kikelj	2	FMF	50

Rešitev

Nalogo si lahko predstavljamo tako, kot da imamo nek „slovar“ — množico kratkih nizov M, iz katerih bomo s stikanjem sestavili vhodni niz s. Če smo si slovar nekako že izbrali, ostane le še vprašanje, kako razdeliti s na čim manj podnizov iz slovarja. To lahko ugotovimo z dinamičnim programiranjem. Recimo, da je s dolg n znakov, $s = s_1 s_2 \ldots s_n$. Naj bo $f(k)$ najmanjše število podnizov iz slovarja, na katere je mogoče razdeliti niz $s_1 \ldots s_k$. To funkcijo lahko računamo po naraščajočih k; če že poznamo $f(k)$ in če je podniz $s_{k+1} \ldots s_m$ tudi prisoten v slovarju, potem lahko razbitje niza $s_1 \ldots s_k$ na $f(k)$ nizov iz slovarja podaljšamo s tem podnizom in dobimo razbitje niza $s_1 \ldots s_m$ na $f(k)+1$ nizov iz slovarja; torej je $f(k)+1$ eden od kandidatov za vrednost $f(m)$. Če je mogoče do istega m priti pri več k-jih, bomo seveda uporabili najmanjšo od tako dobljenih vrednosti. Poleg najmanjše vrednosti $f(m)$ si zapomnimo še, pri katerem k smo jo dobili; recimo temu $g(m)$. Zapišimo ta postopek s psevdokodo:

```
f[0] := 0; for k := 1 to n do f[k] := ∞;
for k := 0 to n − 1:
  m := k + 1;
  while m ≤ n:
    if s_{k+1} s_{k+2} ... s_m ∈ M:
      if f[k] + 1 < f[m]:
        f[m] := f[k] + 1; g[m] := k;
    m := m + 1;
```

Da ne bomo zapravljali časa, lahko notranjo zanko prekinemo, čim postane jasno, da se noben niz v slovarju ne začne na $s_{k+1} \ldots s_m$. Preprost kriterij za to je, da je $m-k$ večje od dolžine najdaljšega niza v slovarju — nizi v naših slovarjih bodo večinoma kratki, zato se bo ta kriterij dobro obnesel. Še boljša možnost pa je, da nize slovarja zložimo v drevo (*trie*), ki ima črke na povezavah, vsakemu nizu slovarja pa ustreza neka pot od korena drevesa navzdol. Vsakič, ko se z m premaknemo za eno črko

naprej po nizu, se premaknemo tudi navzdol po drevesu po povezavi, označeni s črko s_m; če take povezave ni, pa vemo, da se noben niz v slovarju ne začne na $s_{k+1} \ldots s_m$, zato lahko notranjo zanko takoj prekinemo.

Na koncu tega postopka imamo torej v $f(n)$ število besed iz slovarja, na katere smo razbili niz s. Če je $f(n) = \infty$, to pomeni, da se niza sploh ni dalo razbiti, toda temu se bomo zlahka izognili tako, da bomo v slovar med drugim dodali tudi vse posamezne črke abecede (podnize dolžine 1). Z vidika naše naloge je torej ocena te rešitve $f(n) + \sum_{w \in M} |w|$, torej vsota števila besed, na katere smo razbili s, in skupne dolžine vseh besed v slovarju M. Mogoče pa je tudi, da je v slovarju M (odvisno od tega, kako smo ga sestavili) kakšna beseda odveč, torej je pri razbijanju niza s na podnize nismo uporabili. V tem primeru bomo bolj pravo oceno rešitve dobili, če takih besed ne štejemo v vsoto $\sum_{w \in M} |w|$. S pomočjo tabele g, ki smo jo izračunali v zgornjem postopku, lahko rekonstruiramo dejansko zaporedje besed, na katere smo razbili niz s:

$L :=$ prazen seznam; $m := n$;
while $m > 0$:
  $k := g[m]$; dodaj v L besedo $s_{k+1} s_{k+2} \ldots s_m$;
  $m := k$;

V oceno rešitve moramo zdaj poleg $f(n)$ vzeti še vsoto dolžin vseh tistih besed iz slovarja, ki se vsaj enkrat pojavijo v seznamu L.

Ostane nam še vprašanje, kako si izbrati slovar M. Vsekakor je smiselno vanj sprejeti le take besede, ki so podnizi niza s. Poleg tega si lahko predstavljamo, da so bolj obetavni tisti podnizi, ki se v s-ju pojavijo čim večkrat. Če pogledamo, kakšni so nizi s v naših testnih primerih, lahko vidimo, da pogosti podnizi najbrž ne bodo zelo dolgi; na primer, če je niz s nastal s stikanjem besed iz nekega besedila v nekem naravnem jeziku, se v takem besedilu lahko po večkrat pojavijo posamezne besede ali kratke fraze, zelo dolga podzaporedja besed pa najbrž ne. Zato se lahko odločimo, da bomo v slovar sprejeli le podnize niza s do neke maksimalne dolžine d, na primer 20 znakov. Tako imamo $O(nd)$ podnizov, ki pridejo v poštev za v slovar — v praksi jih bo manj kot $n \cdot d$, ker se mnogi podnizi ponavljajo. Uredimo jih padajoče po številu pojavitev v s. Slovar lahko zdaj tvorimo na primer z neke vrste požrešnim postopkom: za začetek dodajmo v slovar le podnize odlžine 1, nato pa pregledujmo daljše podnize (po padajočem številu pojavitev podniza v s) in pri vsakem poglejmo, ali se dosedanja rešitev kaj izboljša, če ta niz dodamo v slovar; če se, ga obdržimo, sicer pa ga vržemo iz slovarja, nato pa v vsakem primeru nadaljujemo z naslednjim možnim podnizom.

UNIVERZITETNI PROGRAMERSKI MARATON

Društvo ACM Slovenija sodeluje tudi pri pripravi študenskih tekmovanj v programiranju, ki v zadnjih letih potekajo pod imenom Univerzitetni programerski maraton (UPM, `tekmovanja.acm.si/upm`) in so odskočna deska za udeležbo na ACMovih mednarodnih študentskih tekmovanjih v programiranju (International Collegiate Programming Contest, ICPC). Ker UPM ne izdaja samostojnega biltena, bomo na tem mestu na kratko predstavili to tekmovanje in njegove letošnje rezultate.

Na študentskih tekmovanjih ACM v programiranju tekmovalci ne nastopajo kot posamezniki, pač pa kot ekipe, ki jih sestavljajo po največ trije člani. Vsaka ekipa ima med tekmovanjem na voljo samo en računalnik. Naloge so podobne tistim iz tretje skupine našega srednješolskega tekmovanja, le da so včasih malo težje oz. predvsem predpostavljajo, da imajo reševalci že nekaj več znanja matematike in algoritmov, ker so to stvari, ki so jih večinoma slišali v prvem letu ali dveh študija. Časa za tekmovanje je pet ur, nalog pa je praviloma 6 do 8, kar je več, kot jih je običajna ekipa zmožna v tem času rešiti. Za razliko od našega srednješolskega tekmovanja pri študentskem tekmovanju niso priznane delno rešene naloge; naloga velja za rešeno šele, če program pravilno reši vse njene testne primere. Ekipe se razvrsti po številu rešenih nalog, če pa jih ima več enako število rešenih nalog, se jih razvrsti po času oddaje. Za vsako uspešno rešeno nalogo se šteje čas od začetka tekmovanja do uspešne oddaje pri tej nalogi, prišteje pa se še po 20 minut za vsako neuspešno oddajo pri tej nalogi. Tako dobljeni časi se seštejejo po vseh uspešno rešenih nalogah in ekipe z istim številom rešenih nalog se potem razvrsti po skupnem času (manjši ko je skupni čas, boljša je uvrstitev).

UPM poteka v štirih krogih (dva spomladi in dva jeseni), pri čemer se za končno razvrstitev pri vsaki ekipi zavrže najslabši rezultat iz prvih treh krogov, četrti (finalni) krog pa se šteje dvojno. Najboljše ekipe se uvrstijo na srednjeevropsko regijsko tekmovanje (CERC, ki bi moralo biti jeseni leta 2020 v Pragi, vendar je bilo zaradi epidemije koronavirusa večkrat preloženo in so ga končno izvedli 25.–26. septembra 2021 prek interneta), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bi moralo v normalnih razmerah potekati spomladi 2021, vendar je bilo preloženo in bo predvidoma izvedeno novembra 2022).[12]

Na letošnjem UPM je sodelovalo 41 ekip s skupno 117 tekmovalci, ki so prišli s treh slovenskih univerz, nekaj pa je bilo celo srednješolcev. Tabela na naslednjih dveh straneh prikazuje vse ekipe, ki so se pojavile na vsaj enem krogu tekmovanja.

[12] Prejšnje zaključno svetovno tekmovanje, za katero smo v *Biltenu* 2019 zapisali, da „bo od 21. do 26. junija 2020 v Moskvi“, je bilo v omenjenem mestu izvedeno šele 1.–6. oktobra 2021.

	Ekipa	Št. rešenih nalog*	Čas
1	Žiga Željko, Tim Poštuvan, Marko Hostnik (FRI + FMF)	23	38:12:59
2	Žiga Vene, Aljaž Eržen, Marko Rus (FRI)	19	24:19:16
3	Benjamin Bajd, Tevž Lotrič (Gim. Kranj), Bor Grošelj Simić (FMF)	17	21:04:20
4	Vid Drobnič, Matej Marinko (FMF), Žiga Patačko Koderman (FRI)	17	27:04:26
5	Urban Duh, Andraž Maier, Gregor Kikelj (FMF), Miha Rajter (FRI + FMF)**	17	29:47:26
6	Vid Keršič, Mitja Žalik, Matic Rašl (FERI)	16	23:45:55
7	Matevž Miščič, Jon Mikoš (FMF)	16	28:02:56
8	M. Besher Massri, Nemanja Torbica, Mirza Redžić (FAMNIT)	16	36:45:35
9	Jakob Schrader, Daniel Sami Blažič, Patrik Žnidaršič (Gim. Vič)	14	16:56:18
10	Lan Sevčnikar, Matija Likar, Luka Lonec (II. gim. Maribor)	13	18:05:26
11	Jakob Kordež, Martin Domajnko, Tilen Koren (FERI)	13	19:49:29
12	Tadej Petrič, Žan Bajuk, Nejc Zajc (FMF)	11	16:13:33
13	Ilija Tavchioski, Boshko Koloski, Josif Tepegjozov (FRI)	11	18:21:24
14	Matija Kocbek, Luka Horjak, Lovro Drovenik (I. gim. v Celju)	7	15:19:48
15	Domen Vreš (FRI + FMF), Timen Stepišnik Perdih (FRI)	6	6:28:28
16	Domen Hočevar (Gim. Novo mesto)	6	6:56:34
17	Bor Brecelj (FRI + FMF), Zala Erič, Miha Benčina (FRI)	6	9:09:32
18	Rok Strah, Gašper Golob, Jakob Zmrzlikar (FMF)	6	11:18:45
19	Ines Meršak (—), Matic Oskar Hajšen, Jan Rozman (FMF)	6	15:07:38
20	Tadej Tomažič (Vegova Lj.), Tomaž Tomažič, Blaž Blokar	6	16:32:21
21	Žiga Šmelcer (FE), Žiga Gradišar (FMF), Lojze Žust (FRI)	5	5:23:09
22	Mirza Krbezlija, Hasan Mahmutagić, Arber Avdullahu (FAMNIT)	5	6:56:21
23	Andrej Kolar-Požun, Jan Jezeršek, Miha Rot (FMF)	5	8:00:54
24	Alexei Drake (FRI), Mitja Vodnik, Luka Medic (FMF)	5	9:11:03
25	Davor Ornik, Urban Knuples, Janko Gruden (FERI)	5	9:50:46
26	Tjaž Silovšek (FMF), Marcel Tori, Tim Vučina (FRI)	5	11:22:09
27	Jaka Vrhovec, Janez Justin, Sara Veber (FRI + FMF)	5	11:31:57
28	Juš Hladnik, Miha Torkar, Matic Fučka (FRI + FMF)	5	22:21:33
29	Ella Potisek, Katja Schrader (Gim. Vič), Jaka Basej (FMF)	5	23:52:56
30	Filip Štamcar, Jakob Kralj, Jošt Smrtnik (Gim. Vič)	4	5:20:23
31	Žan Magerl, Domen Drzin (FRI), Urban Cör (FRI + FMF)	4	5:49:54

* 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.

** Gregor Kikelj je nastopil le v prvih treh krogih, Miha Rajter pa (namesto njega) le v finalnem.

(*nadaljevanje na naslednji strani*)

	Ekipa	Št. rešenih nalog*	Čas
32	Mai Praskalo, Urban Arbajter, David Mikek (FERI)	4	5:52:57
33	Nadezhda Komarova, Klemen Klopčič, Gregor Alič (Gim. Bežigrad)	4	9:08:14
34	Iris Ulčakar, Maruša Lekše, Anja Pirnat (FMF)	4	9:53:42
35	Matic Erznožnik (FMF), Gal Petkovšek (FRI + FMF), Mark Bogataj (FRI)	4	12:02:26
36	Jovana Murdjeva, Sunaj Nedjip (FERI)	3	4:35:21
37	Aljaž Žel, Anže Ferčec, Nejc Jeušnik (FERI)	3	4:36:54
38	Anže Marinko, Jernej Jezeršek (FRI + FMF), Luka Marinko (F. za strojništvo, Lj.)	3	5:52:18
39	Matija Gubanec Hančič (FMF), Andraž Novak (FRI)	2	5:46:05
40	Marko Kužner, Luka Kobale, Jani Kaukler (FERI)	1	0:45:24
41	Matjaž Vuherer, Jure Dolar, Domen Višnar (ŠCC, Gim. Lava)	0	0:00:00

* Opomba: naloge z najslabšega od prvih treh krogov se ne štejejo, naloge z zadnjega kroga pa se štejejo dvojno. Enako je tudi pri času, le da se čas zadnjega kroga ne šteje dvojno.

Na srednjeevropskem tekmovanju so nastopile ekipe 1, 4, 5 in 7 (malo spremenjena) kot predstavnice Univerze v Ljubljani, ekipi 6 (malo spremenjena) in 11 kot predstavnici Univerze v Mariboru in ekipa 8 kot predstavnica Univerze na Primorskem. V konkurenci 88 ekip s 25 univerz iz 6 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
25	Gregor Kikelj, Urban Duh, Andraž Maier	5	12:59
34	Nemanja Torbica, Mirza Redžić, M. Besher Massri	3	5:28
37	Žiga Patačko Koderman, Vid Drobnič, Matej Marinko	3	7:40
38	Matevž Miščič, Miha Rajter, Jon Mikoš	3	8:51
42	Tilen Koren, Jakob Kordež, Martin Domajnko	3	11:29
44	Marko Hostnik, Žiga Željko, Tim Poštuvan	2	1:30
80	Vid Keršič, Niko Uremović, Mitja Žalik	1	2:07

Na srednjeevropskem tekmovanju je bilo 11 nalog, od tega so jih najboljše ekipe rešile po devet.

ANKETA

Ker je letošnje tekmovanje v celoti potekalo prek interneta, smo na ta način izvedli tudi anketo (prek spletne strani `1ka.si`). Vprašanja na anketi so prikazana spodaj in so bila približno enaka kot prejšnja leta, ko je bila anketa na papirju. Rezultati ankete so predstavljeni na str. 155–162.

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? _____

[Le pri 1. in 2. skupini.] Kakšno je tvoje mnenje o sistemu za oddajanje odgovorov prek računalnika? _____

_____

[Le pri 3. skupini.] Letos v tretji skupini podpiramo reševanje nalog v pascalu, C, C++, C#, javi in pythonu. Bi rad uporabljal kakšen drug programski jezik? Če da, katerega? _

_____

Katere od naslednjih jezikovnih konstruktov in programerskih prijemov znaš uporabljati?	ne poznam	da, slabo	da, dobro
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz standardnega vhoda ali pa ju zapisati na standardni izhod?	☐	☐	☐
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz datoteke ali pa ju zapisati v datoteko?	☐	☐	☐
Tabele (**array**):			
— enodimenzionalne	☐	☐	☐
— dvodimenzionalne	☐	☐	☐
— večdimenzionalne	☐	☐	☐
Znaš napisati svoj podprogram (**procedure**, **function**)	☐	☐	☐

Poznaš rekurzijo	☐	☐	☐
Kazalce, dinamično alokacijo pomnilnika (New/Dispose, GetMem/FreeMem, malloc/free, **new**/**delete**, ...)	☐	☐	☐
Zanka **for**	☐	☐	☐
Zanka **while**	☐	☐	☐
Gnezdenje zank (ena zanka znotraj druge)	☐	☐	☐
Naštevni tipi (*enumerated types* — **type** ImeTipa = (Ena, Dve, Tri) v pascalu, **typedef enum** v C/C++)	☐	☐	☐
Strukture (**record** v pascalu, **struct**/**class** v C/C++)	☐	☐	☐
and, **or**, **xor**, **not** kot aritmetični operatorji (nad biti celoštevilskih operandov namesto nad logičnimi vrednostmi tipa boolean) (v C/C++/C#/javi: &, |, ^, ~)	☐	☐	☐
Operatorja **shl** in **shr** (v C/C++/C#/javi: <<, >>)	☐	☐	☐
Znaš uporabiti kakšnega od naslednjih razredov iz standardnih knjižnic:			
hash_map, hash_set, unordered_map, unordered_set (v C++), Hashtable, HashSet (v javi/C#), Dictionary (v C#), dict, set (v pythonu)	☐	☐	☐
map, set (v C++), TreeMap, TreeSet (v javi), SortedDictionary (v C#)	☐	☐	☐
priority_queue (v C++), PriorityQueue (v javi), heapq (v pythonu)	☐	☐	☐

[Naslednja skupina vprašanj se je ponovila za vsako nalogo po enkrat.]
Zahtevnost naloge: ☐ prelahka ☐ lahka ☐ primerna ☐ težka ☐ pretežka ☐ ne vem
Naloga je (ali: bi) vzela preveč časa: ☐ da ☐ ne ☐ ne vem
Mnenje o besedilu naloge:
— dolžina besedila: ☐ prekratko ☐ primerno ☐ predolgo
— razumljivost besedila: ☐ razumljivo ☐ težko razumljivo ☐ nerazumljivo
Naloga je bila: ☐ zanimiva ☐ dolgočasna ☐ že znana ☐ povprečna
Si jo rešil(a)?

- ☐ nisem rešil(a), ker mi je zmanjkalo časa za reševanje
- ☐ nisem rešil(a), ker mi je zmanjkalo volje za reševanje
- ☐ nisem rešil(a), ker mi je zmanjkalo znanja za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo časa za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo volje za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo znanja za reševanje
- ☐ rešil(a) sem celo

Ostali komentarji o tej nalogi: _____

_____

Katera naloga ti je bila najbolj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? _____

_____

Katera naloga ti je bila najmanj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? _____

_____

Na letošnjem tekmovanju ste imeli tri ure / pet ur časa za pet nalog.
Bi imel(a) raje: ☐ več časa ☐ manj časa ☐ časa je bilo ravno prav
Bi imel(a) raje: ☐ več nalog ☐ manj nalog ☐ nalog je bilo ravno prav

Kakršne koli druge pripombe in predlogi. Kaj bi spremenil(a), popravil(a), odpravil(a), ipd., da bi postalo tekmovanje zanimivejše in bolj privlačno? _____

_____

Kaj ti je bilo pri tekmovanju všeč? _____

_____

Kaj te je najbolj motilo? _____

_____

Če imaš kaj vrstnikov, ki se tudi zanimajo za programiranje, pa se tega tekmovanja niso udeležili, kaj bi bilo po tvojem mnenju treba spremeniti, da bi jih prepričali k udeležbi?

_____

_____

Ali si pri izpolnjevanju ankete prišel/la do sem? ☐ da ☐ ne

Hvala za sodelovanje in lep pozdrav! Tekmovalna komisija

REZULTATI ANKETE

Anketo je izpolnilo 76 tekmovalcev prve skupine, 27 tekmovalcev druge skupine in 15 tekmovalcev tretje skupine.

Mnenje tekmovalcev o nalogah

Tekmovalce smo spraševali: kako zahtevna se jim zdi posamezna naloga; ali se jim zdi, da jim vzame preveč časa; ali je besedilo primerno dolgo in razumljivo; ali se jim zdi naloga zanimiva; ali so jo rešili (oz. zakaj ne); in katera naloga jim je bila najbolj/najmanj všeč.

Rezultate vprašanj o zahtevnosti nalog kažejo grafi na str. 156. Tam so tudi podatki o povprečnem številu točk, doseženem pri posamezni nalogi, tako da lahko primerjamo mnenje tekmovalcev o zahtevnosti naloge in to, kako dobro so jo zares reševali.

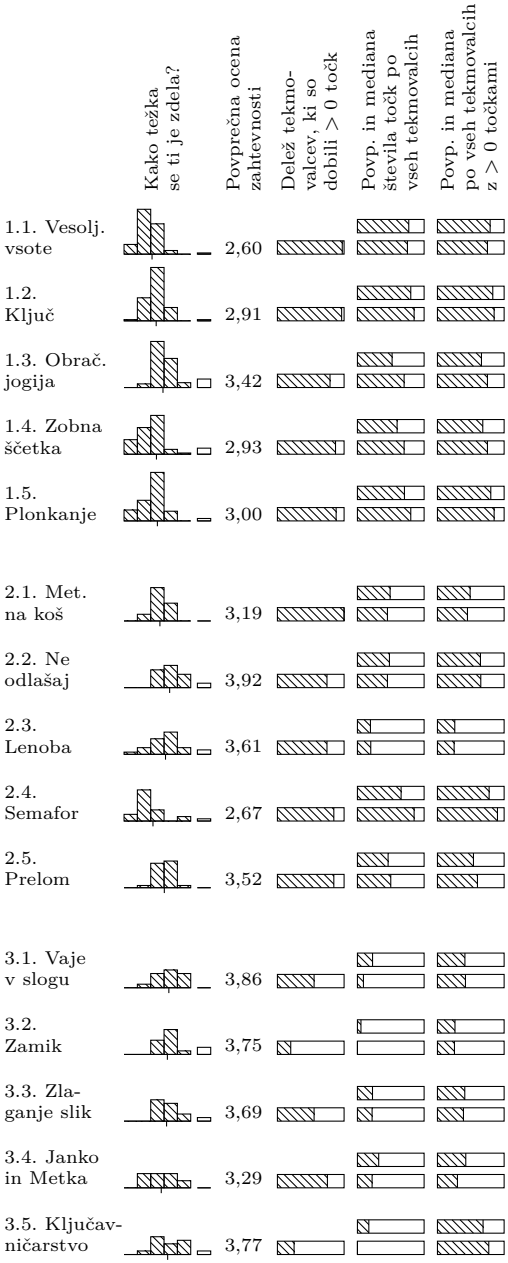
V povprečju so se zdele tekmovalcem v vseh skupinah naloge še kar težke, vendar so številke podobne kot v prejšnjih letih, le v prvi skupini so se jim zdele lažje kot običajno. To slednje se pozna tudi pri rezultatih, saj so dosegli tekmovalci v prvi skupini v povprečju znatno več točk kot npr. lani. Zanimivo je, da po drugi strani v tretji skupini, kjer smo dali težje naloge kot ponavadi in se je to poznalo tudi na doseženih točkah, v anketah tekmovalci niso dajali občutka, da bi se jim zdele naloge kaj dosti težje kot prejšnja leta. 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: 2,97 v prvi skupini (v prejšnjih letih 3,47, 3,32, 3,11, 3,31, 3,41), 3,38 v drugi skupini (prejšnja leta 3,17, 3,19, 3,51, 3,65, 3,33) in 3,67 v tretji skupini (prejšnja leta 3,52, 3,59, 3,73, 3,43, 3,61).

Med tem, kako težka se je naloga zdela tekmovalcem, in tem, kako dobro so jo zares reševali (npr. merjeno s povprečnim številom točk pri tej nalogi), je ponavadi (šibka) negativna korelacija; letos je bila precej močna, podobno kot lani in predlani ($R^2 = 0{,}68$; v prejšnjih letih 0,71, 0,67, 0,70, 0,39, 0,56, 0,14, 0,52, 0,21, 0,11, pred tem okoli 0,4).

V prvi skupini so tekmovalci kot težjo ocenili predvsem nalogo 1.3 (obračanje jogija), ki sicer ni posebej težka, je pa malo neobičajna in zahteva nekaj razmisleka; pri njej je bilo tudi nekaj pripomb, češ da je težko razumljiva. V drugi skupini sta se jim zdeli težki predvsem nalogi 2.2 (ne odlašaj na jutri) in 2.3 (lenoba), mogoče zato, ker je bil poudarek pri njiju na snovanju in opisu postopka, ne pa na implementaciji nečesa, kar bi bilo takoj očitno že iz opisa naloge. V tretji skupini se jim je zdela najtežja naloga 3.1 (vaje v slogu), dokaj težke pa tudi 3.2, 3.3 in 3.5.

Kot najlažji so tekmovalci v prvi skupini ocenili nalogo 1.1 (vesoljske vsote), ki je bila tudi mišljena kot lahka. V drugi skupini se jim je zdela najlažja naloga 2.4 (semafor), kar je nekoliko presenetljivo, saj gre za „realnočasovno“ nalogo in take se zdijo tekmovalcem ponavadi težke. V tretji skupini se jim je zdela najlažja naloga 3.4 (Janko in Metka); nam sicer ni posebej očitno, zakaj bi bila ravno ta kaj lažja od ostalih.

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 157. Nad razumljivostjo besedil ni veliko pripomb, čeprav malo več kot prejšnja leta. Kot težje razumljive so ocenili predvsem naloge 1.3 (obračanje jogija), 2.2 (ne odlašaj na jutri)



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

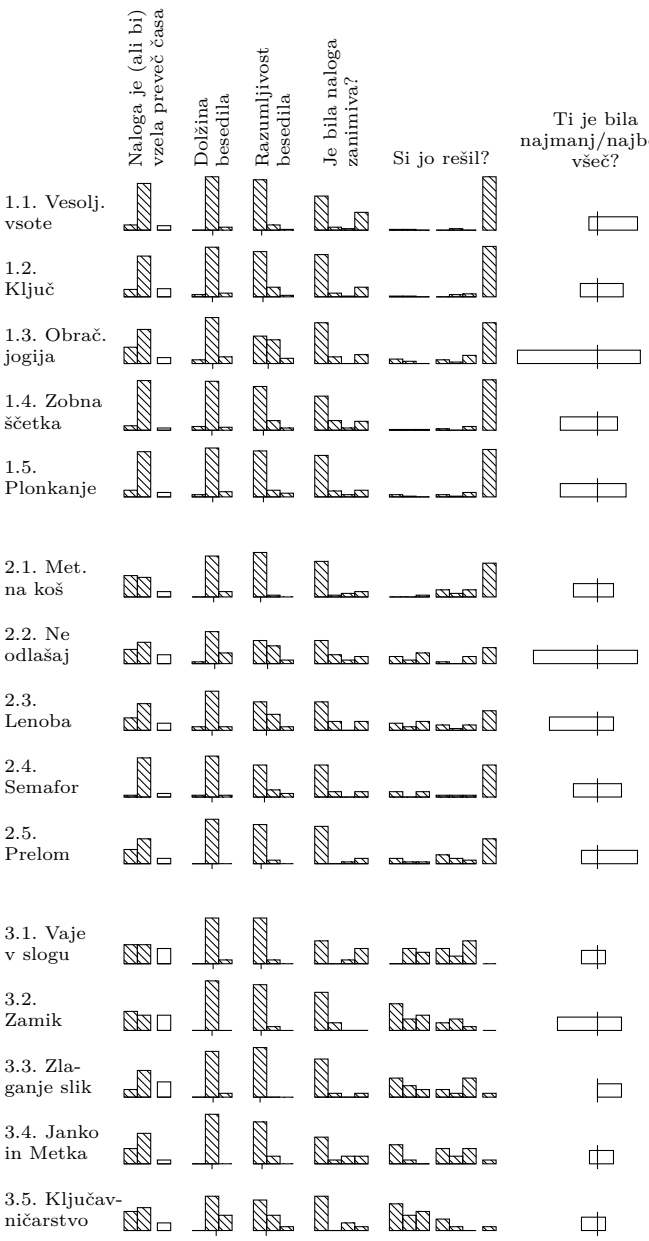
Pomen stolpcev v vsaki vrstici:

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?

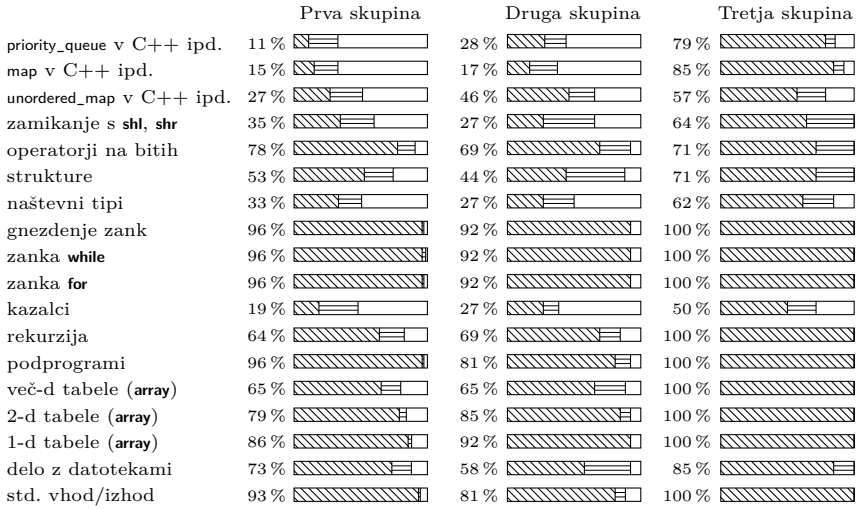


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

in 3.5 (ključavničarstvo). To so približno iste naloge, ki so se jim tudi sicer zdele težke, je pa najbrž težko reči, kaj je vzrok in kaj posledica.

Tudi z dolžino besedil so tekmovalci večinoma zadovoljni; ocene so podobne kot prejšnja leta, v prvi skupini še malenkost boljše. Pri tem še najbolj odstopajo naloge 2.1 (metanje na koš), 2.2 (ne odlašaj na jutri) in 3.5 (ključavničarstvo). Slednja ima res nekoliko daljše besedilo, prvi dve pa ne toliko. Mnenj, da je kakšno besedilo prekratko, je bilo letos malo, vendar več kot lani, največ pri nalogah 1.3 (obračanje jogija) in 1.4 (zobna ščetka).

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta. Kot bolj zanimive izstopajo 2.5 (prelom besedila), 3.3 (zlaganje slik) in 3.5 (ključavničarstvo), kot manj zanimive pa 2.2 (ne odlašaj na jutri) in 2.3 (lenoba). Pripomb, da jim je neka naloga že znana, je bilo letos malo več kot lani, so pa precej enakomerno razpršene med naloge.

Pripomb, da bi naloga vzela preveč časa, je bilo malo, podobno kot prejšnja leta. Največ takih pripomb je bilo pri nalogah 1.3 (obračanje jogija), 2.2 (ne odlašaj na jutri), 3.2 (zamik) in 3.3 (zlaganje slik). Mogoče je to povezano s tem, da so se jim zdele večinoma iste naloge tudi najzahtevnejše, ker drugače po dolžini rešitve ne izstopajo izrazito.

Pri vprašanjih, katera naloga je tekmovalcu najbolj in katera najmanj všeč, so bili glasovi letos precej razpršeni med naloge, večkrat pa se je tudi zgodilo, da je ista naloga dobila veliko glasov pri obeh vprašanjih (npr. 1.5, plonkanje). Kot nepriljubljene izstopajo 1.3 (obračanje jogija), 2.3 (lenoba) in 3.2 (zamik), kot bolj priljubljene pa 1.1 (vesoljske vsote), 2.5 (prelom besedila), 3.3 (zlaganje slik).

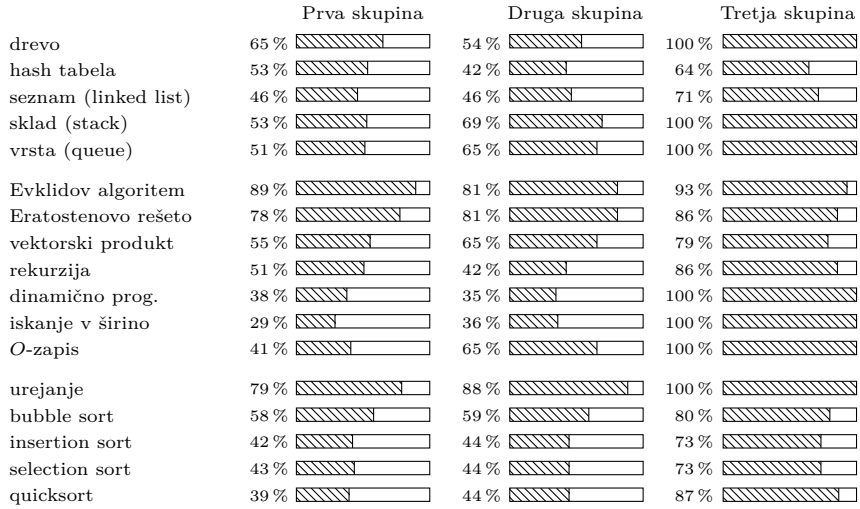


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

Programersko znanje, algoritmi in podatkovne strukture

Ko sestavljamo naloge, še posebej tiste za prvo skupino, nas pogosto skrbi, če tekmovalci poznajo ta ali oni jezikovni konstrukt, programerski prijem, algoritem ali podatkovno strukturo. Zato jih v anketah zadnjih nekaj let sprašujemo, če te reči poznajo in bi jih znali uporabiti v svojih programih.

Rezultati pri vprašanjih o programerskem znanju so podobni tistim iz prejšnjih let. V prvi skupini pravijo, da znajo malo več kot tisti v lanski anketi. Stvari, ki jih tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja leta: kazalci, naštevni tipi in operatorji na bitih, v prvi in drugi skupini tudi strukture in rekurzija. Poznavanje operatorjev na bitih in rekurzije je letos boljše kot ponavadi, še posebej v prvi skupini. Kazalce pozna letos še manj ljudi kot lani (kar sicer najbrž ni čudno, saj jih veliko dela v jezikih, kjer s kazalci nimajo veliko opravka).

Uporaba programskih jezikov

Na splošno so razmerja med različnimi jeziki podobna kot v prejšnjih letih. V prvi skupini je ima letos python še večjo prednost pred C++ kot lani, sledi jima java; C# je letos redkejši. Nekaj ljudi je pisalo v C-ju, pa tudi med uporabniki C++ je nekaj takih, katerih C++ je skoraj C. Tudi v drugi skupini je python najpogostejši, C++ pa drugi najpogostejši; razmerje med njima je podobno kot lani; sledijo java, C in C#, vsak s peščico tekmovalcev. V tretji skupini je še vedno najpogostejši C++, manj kot lani je bilo uporabnikov pythona, več pa uporabnikov jave. Basica ni letos uporabljal nihče, pascal pa le dva v prvi skupini.

Podobno kot prejšnja leta se je tudi letos pojavilo nekaj tekmovalcev (in tekmovalk), ki oddajajo le rešitve v psevdokodi ali pa celo naravnem jeziku, tudi tam, kjer naloga sicer zahteva izvorno kodo v kakšnem konkretnem programskem jeziku. Iz tega bi človek mogoče sklepal, da bi bilo dobro dati več nalog tipa „opiši po-

	Leto in skupina																	
	2020			2019			2018			2017			2016			2015		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal	2			2						4			$\frac{1}{3}$	3		5	2	
C	$3\frac{1}{2}$	3		10	4	$\frac{1}{2}$	5	4	$\frac{1}{2}$	4	3	$2\frac{1}{2}$	$4\frac{1}{3}$	1	2	3		1
C++	$26\frac{1}{2}$	8	14	$21\frac{1}{2}$	$7\frac{1}{2}$	18	$18\frac{1}{2}$	13	11	23	10	$15\frac{1}{2}$	28	8	9	27	9	$9\frac{1}{2}$
java	15	4	3	15	5	1	$21\frac{1}{2}$	$8\frac{1}{2}$	4	28	3	2	24	6	5	22	6	$3\frac{1}{2}$
PHP	1		–			–			–			–			–	3		–
basic			–			–			–									1
C#	6	3		12	2		11	6		7	6		12	5	1	16	5	
python	48	20	3	$36\frac{1}{2}$	$26\frac{1}{2}$	$6\frac{1}{2}$	38	11	$\frac{1}{2}$	42	11	–	$29\frac{1}{3}$	12	–	26	1	–
javascript	2		–			–		$\frac{1}{2}$	–			–	1		–	1		–
julia			–			–			–	1		–			–			–
swift	1		–			–			–			–			–			–
batch			–			–			–			–			–			–
psevdokoda	2	1	–	5	1	–	3	1	–	5		–	5		–	6	1	–
nič	2			3			2		1				2	3		4	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 2020 je en tekmovalec uporabljal C in C++, dva C in python, štirje 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“.

stopek“ (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).

Letos je prvič nekdo pisal odgovore v swiftu; prvič po nekaj letih je nekdo uporabljal PHP, dva pa javascript.

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže gornja tabela. Glede štetja C in C++ v tej tabeli je treba pripomniti, da je razlika med njima majhna in včasih pri kakšnem krajšem kosu izvorne kode že težko rečemo, za katerega od obeh jezikov gre. Je pa po drugi strani videti, da se raba stvari, po katerih se C++ loči od C-ja, sčasoma povečuje; zdaj že veliko tekmovalcev na primer uporablja `string` namesto **`char *`** in tip `vector` namesto tradicionalnih tabel (*arrays*). Novosti, po katerih se zadnje različice C++ (od vključno C++11 naprej) razlikujejo od C++98, je letos uporabljalo kar precej tekmovalcev, še posebej v tretji skupini (npr. ranged **`for`**, **`auto`** v novem pomenu, metoda `emplace_back`).

Pri pythonu zdaj praktično vsi uporabljajo python 3 in ne python 2; je pa res, da je pri tako preprostih programih, s kakršnimi se srečujemo na našem tekmovanju, razlika večinoma le v tem, ali `print` uporabljajo kot stavek ali kot funkcijo.

V besedilu nalog za 1. in 2. skupino objavljamo deklaracije tipov, spremenljivk, podprogramov ipd. v pascalu, C/C++, C#, pythonu in javi. Delež tekmovalcev, ki

pravijo, da deklaracije razumejo, je letos še malo višji kot lani (70/76 v prvi skupini in 26/26 v drugi). Kot običajno so pri vprašanju, ali bi želeli deklaracije še v kakšnem jeziku, nekateri tekmovalci navedli jezike, v katerih deklaracije že imamo, na primer javo ali C#; najpogostejši originalni predlog je bil javascript. V vsakem primeru pa se poskušamo zadnja leta v besedilih nalog izogibati deklaracijam v konkretnih programskih jezikih in jih zapisati bolj na splošno, na primer „napiši funkcijo `foo(x, y)`“ namesto „napiši funkcijo `bool foo(int x, int y)`“.

V rešitvah nalog objavljamo od 2017 izvorno kodo v C++, pri prvi skupini pa tudi v pythonu. Tekmovalce smo v anketi vprašali, če razumejo C++ (ali, v prvi skupini, python) dovolj, da si lahko kaj pomagajo s izvorno kodo v rešitvah, in če bi radi videli izvorno kodo rešitev še v kakšnem drugem jeziku. Večina je s C++ (oz. pythonom) zadovoljna (62/76 v prvi skupini, 25/26 v drugi, 13/14 v tretji); ta delež je podoben kot lani, v drugi skupini še malo višji. Zanimivo vprašanje je, ali bi s kakšnim drugim jezikom dosegli večji delež tekmovalcev (koliko tekmovalcev ne bi razumelo rešitev v javi? ali v pythonu?). Med jeziki, ki bi jih radi videli namesto (ali poleg) C++, jih največ omenja javo (predvsem v prvi skupini) in python (predvsem v drugi skupini). Vendar je s pripravo rešitev v dveh jezikih precej dela, zato bomo do nadaljnjega objavljali rešitve v pythonu (poleg v C++) še vedno le v prvi skupini.

Letnik

Običajno so tekmovalci zahtevnejših skupin večinoma v višjih letnikih kot tisti iz lažjih skupin. Razmerja so podobna kot prejšnja leta; v prvi skupini so tekmovalci v povprečju malo starejši kot lani, v tretji pa malo mlajši. Letos je nastopilo tudi nekaj osnovnošolcev, in sicer vsi v prvi skupini.

	Št. tekmovalcev po letnikih							Povprečni
Skupina	8	9	1	2	3	4	5	letnik
prva	1	5	26	18	26	29	4	2,8
druga			2	8	12	17		3,1
tretja			1	4	8	7		3,0

Druga vprašanja

Podobno kot prejšnja leta je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!), je pa bilo letos malo več kot ponavadi takih tekmovalcev, ki so za tekmovanje izvedeli od prijateljev. V smislu širitve zanimanja za tekmovanje in večanja števila tekmovalcev se zelo dobro obnese šolsko tekmovanje, ki ga izvajamo zadnjih nekaj let, saj se odtlej v tekmovanje vključuje tudi nekaj šol, ki prej na našem državnem tekmovanju niso sodelovale.

Pri vprašanju, kje so se naučili programirati, je podobno kot prejšnja leta najpogostejši odgovor, da so se naučili programirati sami (takih sta približno dve tretjini); sledijo tisti, ki so se tega naučili v šoli pri pouku (takih je dobra tretjina). Približno ena tretjina tekmovalcev pa se je naučila programirati (tudi) na krožkih in tečajih.

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni; takih je še več kot prejšnja leta. Med tistimi, ki niso, so mnenja precej razdeljena, najpogostejši kombinaciji pa sta „več časa, manj nalog“ in (redkeje) „enako časa, manj nalog“.

	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	69	0	15	5	56	36	22	11	6	8	2	66	4	6	66
II	26	0	2	2	18	10	10	5	4	11	1	14	0	8	18
III	13	0	3	0	12	1	6	2	4	8	1	5	0	2	12

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb; tudi posebnih tehničnih težav letos ni bilo. Pri sistemu za oddajo odgovorov v prvi in drugi skupini je precej tekmovalcev želelo, da bi bilo okno za vnos besedila večje in da bi podpiralo avtomatsko zamikanje vrstic ter označevanje sintakse z barvami; toda tem težavam se je najlažje izogniti tako, da pišemo odgovore v svojem priljubljenem urejevalniku ali razvojnem okolju (to je bilo letos še toliko lažje, ker so tekmovalci reševali naloge doma), nato pa jih le skopiramo in prilepimo v spletni obrazec za oddajo odgovorov.

V preteklosti si je veliko tekmovalcev želelo tudi, da bi imeli v prvi in drugi skupini na računalnikih prevajalnike in podobna razvojna orodja. Razlog, zakaj smo se v teh dveh skupinah izogibali prevajalnikom, je bil predvsem ta, da hočemo s tem obdržati poudarek tekmovanja na snovanju algoritmov, ne pa toliko na lovljenju drobnih napak; in radi bi tekmovalce tudi spodbudili k temu, da se lotijo vseh nalog, ne pa da se zakopljejo v eno ali dve najlažji in potem večino časa porabijo za testiranje in odpravljanje napak v svojih rešitvah pri tistih dveh nalogah. Toda letošnje tekmovanje, ko so vsi reševali naloge doma in torej dostop do prevajalnikov in razvojnih orodij imeli, je pokazalo, da te težave vendarle niso nastopile; tekmovalci so se večinoma lotili vseh nalog in rezultati v prvi skupini so bili še boljši kot ponavadi. Zato bomo predvidoma tudi v prihodnje, ko bo tekmovanje spet potekalo v živo namesto prek interneta, omogočili tekmovalcem prve in druge skupine tudi uporabo prevajalnikov.

CVETKE

V tem razdelku je zbranih nekaj zabavnih odlomkov iz rešitev, ki so jih napisali tekmovalci. V oklepajih pred vsakim odlomkom sta skupina in številka naloge.

(1.1) Za ljubitelje nepotrebne rekurzije:

```
private static int stSestevancev(String koda, int index) {
  if (index >= koda.length()) return 0;
  int st = koda.charAt(index) == '*' ? 1 : 0;
  return st + stSestevancev(koda, index + 1);
}
```

(1.1) Posrečena tipkarska napaka:

```
// bere podatke iz vodne datoteke
```

(1.1) Rešitev z močnimi stališči glede syntax highlightinga:

```
// upam da si zlomi nogo tisti, ki je v urejevalnik
// pozabil dodati syntax highlighting
```

Glede na to, da so tekmovalci reševali od doma, bi lahko sicer pisal v kakšnem drugem urejevalniku in na koncu le skopiral izvorno kodo v obrazec na spletni strani tekmovalnega strežnika.

(1.1) Nekateri tekmovalci so za pluse med seštevanci poskrbeli tako, da so najprej za vsakim seštevancem dodali plus, na koncu pa so zadnji plus pobrisali. Naslednji je to poskušal storiti kar z znakom backspace:

```
System.out.print("\b\b = " + vsota);
// "\b\b" samo zaradi formatiranja izpisa, da pobrisemo zadnji presledek ter +
```

Ta rešitev se sicer ne obnese, če je standardni izhod preusmerjen v datoteko.

(1.1) Namesto da bi vsoto izračunal sam, spodnji program počaka, da ima v nizu `r` že vse seštevance (ločene s plusi), nato pa ga poda pythonovi funkciji `eval`, da iz njega izračuna vsoto:

```
r += ' = ' + str(eval(r))
```

(1.2) Za ljubitelje nepotrebnih pretvorb: namesto da bi primerjala dva zaporedna znaka niza, ju spodnja rešitev najprej pretvori v niza, nato v celi števili in ju šele nato primerja.

```
if (Integer.parseInt(kljuc1.charAt(i) + "") == Integer.parseInt(kljuc1.charAt(i − 1) + "")) {
```

(1.2) Pri tej nalogi je nekaj tekmovalcev štelo pojavitve števk od 1 do 8 tako, da so imeli 8 različnih spremenljivk. Cvetko pa dobimo, če to naredi tekmovalec, ki sicer očitno zna čisto dobro uporabljati tudi tabele:

```
int[] zareze = stevilo.ToString().ToCharArray().Select(Convert.ToInt32).ToArray();
for (int i = 0; i < zareze.Length + 1; i++)
{
  if (zareze[i] == 1) ena++;
  if (zareze[i] == 2) dve++;
  :
  :
  if (ena > 2 || dve > 2 || tri > 2 || stiri > 2 || pet > 2 || sest > 2 || sedem > 2 || osem > 2)
```

(1.2) Rešitev, ki pazi, da spremenljivke ne bi po nepotrebnem postavila na **true** več kot enkrat:

```
if ((int) koda.charAt(i) - prejGlobina == 0 && i != 0 && tri == false)
  tri = true;
```

(1.2) Rešitve tega tekmovalca so takšne, kot da bi nekdo mehanično prevedel kodo iz kakšnega programskega jezika v slovenščino:

> [...] Nato nastavi n = 1. Nato, če je n element v števke enak m elementu v števke, potem nastavi 3. znak v stringu x, potem pa spremeni n za 1, m za 1, sicer: če je absolutna vrednost n element v števke − m element v števke večja od 5, potem nastavi 2. znak v stringu x na 0 in spremeni n za 1, m za 1. Prejšnji ukaz ponavljaj do n = 6. Nato nastavi n na 1, m na 2. [...]

Potem bi bilo že bolje, če bi oddal kar izvorno kodo.

(1.2) Zanimiva optimizacija na začetku rešitve:

```
if (kljuc < 112233 || kljuc > 998877) { // testiramo ključ za 000 na začetku in
  // osnovno veljavnost (najmanjša in največja številka glede na pogoje)
  System.out.print("Neveljaven ključ");
```

Zgornjo mejo bi sicer lahko še znižal, na 887766, saj ključi z devetkami niso veljavni.

(1.2) Zakaj bi pisali stevec[kombinacija[i]] += 1, če lahko zakompliciramo:

```
switch (kombinacija[i])
{
  case 1: stevec[1] += 1; break;
  case 2: stevec[2] += 1; break;
```

in tako naprej do 9.

(1.2) Rešitev z nezaupanjem do operatorja / v C++:

```
a.push_back((d - d % 100000) / 100000);
a.push_back((d % 100000 - d % 10000) / 10000);
a.push_back((d % 10000 - d % 1000) / 1000);
```

in tako naprej. V resnici ne bi bilo treba odštevati, kajti operator / bo rezultat zaokrožil navzdol na celo število in pri tem se bo izgubil prav tisti neceli del, ki se mu poskuša gornja rešitev izogniti z odštevanjem.

(1.2) Naslednji rešitvi očitno ni dovolj preveriti, ali je trenutna števka za 5 manjša od naslednje, ampak poleg tega preveri še, ali je sploh manjša od naslednje:

```
if (a[x] < a[x+1] && (a[x]+5) < a[x+1]) // Preverjam ali je sosednja številka vecja za 5
   dovoljeno = false;
```

(1.2) Dekadentno: zakaj bi za štetje pojavitev vsake števke uporabili tabelo (list v pythonu) z 10 elementi, če pa lahko uporabimo slovar:

```
pojavitve = { 0:0, 1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:0, 8:0, 9:0 }
```

(1.2) Naslednji tekmovalec je v vseh zankah **for** na koncu ročno povečeval števec. Koristi to seveda ne, na srečo pa v pythonu tudi ne škoduje.

```
for k in range(6):
   if seznam[k] == seznam[n]: # 4. izjava
      d = d + 1
   k = k + 1
```

(1.2) Rešitev za ljubitelje nepotrebnega množenja:

```
if (st == 0 || st == 9) { pog[0] *= 0; }
else pog[0] *= 1;
```

(1.2) Tale tekmovalec števkam pravi „števniki“:

```
short X[6]; // posamezni števniki
for (short & i : X) { // pojdi skozi števila
   i = short(input % 10); // shrani samo prvi števnik števila
   input /= 10; // zamakni število za en števnik
```

(1.3) Za ljubitelje ročnega dela:

```
// Najbolj smiselno reševanje je, da na roke napišemo vse možnosti;
```

(1.3) Tale reševalec je pomislil tudi na to, da bi utegnil na jogiju spati še kdo drug:

```
if n == 0: # če uporabnik na jogiju še ni spal (in predpostavljamo, da tudi kdo drug,
           # ki bi ga lahko obrabil, ni), jogija ni treba obračati.
```

(1.3) Lahko bi le preveril, če je st < 100000, a je raje zakompliciral:

```
if ((int) (Math.log10(st) + 1) < 6) // ker ne sme vsebovati globine 0, to pomeni,
      // da se stevilo ne more zaceti z 0, zato s pomocjo logaritma preverimo dolzino
```

(1.3) Rešitev z veliko nepotrebnimi dvopičji:

```
int trenutnaPozicija = 0, niz[4] = { 0, 0, 0, 0 };
char ObrniJogi(n) {
   ::niz[::trenutnaPozicija] += n;
   int novaPozicija = ::trenutnaPozicija;
```

Ni ravno očitno, zakaj so se mu zdela potrebna, saj razen globalnih spremenljivk nima ničesar drugega s tema imenoma.

(1.4) Iz komentarja na začetku rešitve:

> Najprej sem naredil spremenljivki true in false, saj nisem siguren, da ju ima C vgrajeni.

In nato za komentarjem:

```
const int true = 0 == 0;
const int false = 0 != 0;
```

Prav vgrajenih res nima, sta pa v headerju <stdbool.h> iz standardne knjižnice.

(1.4) Prispevek za rubriko „tekmovalci čestitajo in pozdravljajo“:

```
if OdcitajUro() >= 120: # Če motor deluje že 120 sekund ali več, ga je treba
   # izklopiti. Nekateri zobozdravniki sicer priporočajo ščetkanje vsaj 3 minute,
   # proizvajalci očitno ne.
   # Cvetke, here I come!
```

(1.4) Nekdo ne mara operatorja >=:

```
if (OdcitajUro == 120 || OdcitajUro > 120) break;
```

(1.4) Ko zagledaš „**for** (; ;)“, pomisliš, da bo to neskončna zanka, vendar je spodnja bolj kratke sape:

```
for ( ; ; )
{
   if (vklop == true)
   {
      Motor(vklop);
      PozeniUro();
      break;
   }
   break;
}
```

(1.4) Zanimiva sintaktična inovacija: „!=“ namesto „= !“ (po zgledu „+=“ in podobnih operatorjev).

```
if (tipka and last != tipka):
   prizig != prizig
```

(1.5) Vprašljiva skrb za učinkovitost:

> Sklepam, da so podatki podani v obliki 2d tabele, vendar bi lahko za višjo učinkovitost uporabili SQL databazo

(2.1) Pri tekmovalnem programiranju nekateri radi uporabljajo makre, s katerimi naj bi prihranili nekaj časa pri tipkanju izvorne kode. Nekateri pri tem malo pretiravajo:

```
#define INC(p) for (int i = 0; p; i++)
#define INC2(p) for (int j = 0; p; j++)
```

(2.1) Rešitev z veliko zaupanja v košarkaše:

```
# program dela na predvidevanju da bodo več košev zadeli kot zgrešili
# za to ne preverja dolžino katerega simbola dela
```

(2.2) Posrečena tipkarska napaka:

```
# Najdemo najbolsji razpored
def najboljsi_razpored(obveznosti):
```

(2.2) Še ena posrečena tipkarska napaka:

```
// Če pa je več objeznosti, pa razporedi tako, da imajo hitrejše prednost
```

Lahko si predstavljamo obveznosti, ki so jezne, ker imajo nekatere druge prednost pred njimi `:)`

(2.2) Za ljubitelje samocenzure:

```
if (freeUntil < 0) {
  cout << "Ni rešljivo, za***** si" << endl;
```

(2.3) Rešitev z veliko lenobe:

```
// ta in naslednja naloga, iskreno — ta RTK me spodbuja k lenobi
#include <stdio.h>
int k[10000000], d[1000000], dnjeviljeta[10000000]; // število ničel je completely
                                                    // random sori ne da se mi štet
```

Kasneje v isti rešitvi:

> // za moje osebno zadovoljstvo bomo predpostavili, da so obveznosti že urejene po vrsti od prve do zadnje glede na rok oddaj. Obrazložitev 1: Nikjer ne piše, da niso. Obrazložitev 2: Ne da se mi ukvarjati s sortiranjem structov ker je to rak in trenutno že imamo koronavirus, hvala. Obrazložitev 3: Kakšen krompir nima sortiranih svojih obveznosti po datumih?

(2.4) Zakaj bi preverjali „maks − c >= 100“, če lahko zakompliciramo:

```
elif ((maks − c) // 100 > 0): Prikazi(−1)
```

(3.1) Še en primer potrate pri podatkovnih strukturah: namesto tabele s 26 elementi uporablja spodnja rešitev drevo (`map` v C++) s ključi `'a'`, ..., `'z'`.

```
map<char, vector<int>> a;
for (char i = 'a'; i <= 'z'; i++)
  a[i] = {};
```

(3.2) Prispevek na temo „zavajanje sovražnika“:

```
typedef int ll;
```

Je pa res, da je v zgodnejši različici programa to bil „**typedef long long** ll“.

(3.3) Še en zanimiv primer makrov, ki naj bi reševalcu prihranili čas pri tipkanju:

```
#define loop(i, a, b) for (long long i = a; i < b; i++)
#define pool(i, a, b) for (long long i = a − 1; i >= b; i−−)
⋮
loop(i, 1, n + 1) {
  ll dol = 0, vis = llmin;
  pool(j, i + 1, 1) {
```

(3.4) Rešitev z veliko kumulativnimi vsotami:

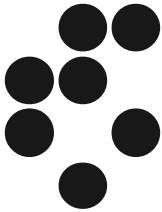
```
a_cum = make_cum(a);
b_cum = make_cum(b);
inpt_cum = make_cum(inpt);
intrs_cum = make_cum(intrs);
```

Za konec pa še ena cvetka iz anket (in sicer iz prve skupine):

> (*Vprašanje v anketi.*) Kaj bi spremenil(a), da bi postalo tekmovanje zanimivejše in bolj privlačno?
> (*Odgovor.*) Več težavnih stvari. [...] Npr. Napiši funkcijo, ki med 0 in 1 (double) randomly izbere številko. Izračunaj število pi. Ali pa tortise and harem algoritm.

SODELUJOČE INŠTITUCIJE

Institut Jožef Stefan



Institut je največji javni raziskovalni zavod v Sloveniji s skoraj 800 zaposlenimi, od katerih ima približno polovica doktorat znanosti. Več kot 150 naših doktorjev je habilitiranih na slovenskih univerzah in sodeluje v visokošolskem izobraževalnem procesu. V zadnjih desetih letih je na Institutu opravilo svoja magistrska in doktorska dela več kot 550 raziskovalcev. Institut sodeluje tudi s srednjimi šolami, za katere organizira delovno prakso in jih vključuje v aktivno raziskovalno delo. Glavna raziskovalna področja Instituta so fizika, kemija, molekularna biologija in biotehnologija, informacijske tehnologije, reaktorstvo in energetika ter okolje.

Poslanstvo Instituta je v ustvarjanju, širjenju in prenosu znanja na področju naravoslovnih in tehniških znanosti za blagostanje slovenske družbe in človeštva nasploh. Institut zagotavlja vrhunsko izobrazbo kadrom ter raziskave in razvoj tehnologij na najvišji mednarodni ravni.

Institut namenja veliko pozornost mednarodnemu sodelovanju. Sodeluje z mnogimi uglednimi institucijami po svetu, organizira mednarodne konference, sodeluje na mednarodnih razstavah. Poleg tega pa po najboljših močeh skrbi za mednarodno izmenjavo strokovnjakov. Mnogi raziskovalni dosežki so bili deležni mednarodnih priznanj, veliko sodelavcev IJS pa je mednarodno priznanih znanstvenikov.

Tekmovanje sta podprla naslednja odseka IJS:

CT3 — Center za prenos znanja na področju informacijskih tehnologij

Center za prenos znanja na področju informacijskih tehnologij izvaja izobraževalne, promocijske in infrastrukturne dejavnosti, ki povezujejo raziskovalce in uporabnike njihovih rezultatov. Z uspešnim vključevanjem v evropske raziskovalne projekte se Center širi tudi na raziskovalne in razvojne aktivnosti, predvsem s področja upravljanja z znanjem v tradicionalnih, mrežnih ter virtualnih organizacijah. Center je partner v več EU projektih.

Center razvija in pripravlja skrbno načrtovane izobraževalne dogodke kot so seminarji, delavnice, konference in poletne šole za strokovnjake s področij inteligentne analize podatkov, rudarjenja s podatki, upravljanja z znanjem, mrežnih organizacij, ekologije, medicine, avtomatizacije proizvodnje, poslovnega odločanja in še kaj. Vsi dogodki so namenjeni prenosu osnovnih, dodatnih in vrhunskih specialističnih znanj v podjetja ter raziskovalne in izobraževalne organizacije. V ta namen smo postavili vrsto izobraževalnih portalov, ki ponujajo že za več kot 500 ur posnetih izobraževalnih seminarjev z različnih področij.

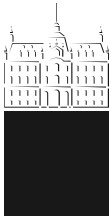
Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi virtualno učečo se skupnost in pripomoči k učinkovitejšemu povezovanju znanosti in industrije ter večji prepoznavnosti domačega znanja v slovenskem, evropskem in širšem okolju.

E3 — Laboratorij za umetno inteligenco
Področje dela Laboratorija za umetno inteligenco so informacijske tehnologije s poudarkom na tehnologijah umetne inteligence. Najpomembnejša področja raziskav in razvoja so: (*a*) analiza podatkov s poudarkom na tekstovnih, spletnih, večpredstavnih in dinamičnih podatkih, (*b*) tehnike za analizo velikih količin podatkov v realnem času, (*c*) vizualizacija kompleksnih podatkov, (*d*) semantične tehnologije, (*e*) jezikovne tehnologije.

Laboratorij za umetno inteligenco posveča posebno pozornost promociji znanosti, posebej med mladimi, kjer v sodelovanju s Centrom za prenos znanja na področju informacijskih tehnologij (CT3) razvija izobraževalni portal VideoLectures.NET in vrsto let organizira tekmovanja ACM v znanju računalništva.

Laboratorij tesno sodeluje s Stanford University, University College London, Mednarodno podiplomsko šolo Jožefa Stefana ter podjetji Quintelligence, Cycorp Europe, LifeNetLive, Modro Oko in Envigence.

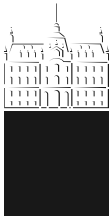
*



Fakulteta za matematiko in fiziko
Fakulteta za matematiko in fiziko je članica Univerze v Ljubljani. Sestavljata jo Oddelek za matematiko in Oddelek za fiziko. Izvaja dodiplomske univerzitetne študijske programe matematike, računalništva in informatike ter fizike na različnih smereh od pedagoških do raziskovalnih.

Prav tako izvaja tudi podiplomski specialistični, magistrski in doktorski študij matematike, fizike, mehanike, meteorologije in jedrske tehnike.

Poleg rednega pedagoškega in raziskovalnega dela na fakulteti poteka še vrsta obštudijskih dejavnosti v sodelovanju z različnimi institucijami od Društva matematikov, fizikov in astronomov do Inštituta za matematiko, fiziko in mehaniko ter Instituta Jožef Stefan. Med njimi so tudi tekmovanja iz programiranja, kot sta Programerski izziv in Univerzitetni programerski maraton.



Fakulteta za računalništvo in informatiko
Glavna dejavnost Fakultete za računalništvo in informatiko Univerze v Ljubljani je vzgoja računalniških strokovnjakov različnih profilov. Oblike izobraževanja se razlikujejo med seboj po obsegu, zahtevnosti, načinu izvajanja in številu udeležencev. Poleg rednega izobraževanja skrbi fakulteta še za dopolnilno izobraževanje računalniških strokovnjakov, kot tudi strokovnjakov drugih strok, ki potrebujejo znanje informatike. Prav posebna in zelo osebna pa je vzgoja mladih raziskovalcev, ki se med podiplomskim študijem pod mentorstvom univerzitetnih profesorjev uvajajo v raziskovalno in znanstveno delo.

Fakulteta za elektrotehniko, računalništvo in informatiko

Fakulteta za elektrotehniko, računalništvo in informatiko (FERI) je znanstveno-izobraževalna institucija z izraženim regionalnim, nacionalnim in mednarodnim pomenom. Regionalnost se odraža v tesni povezanosti z industrijo v mestu Maribor in okolici, kjer se zaposluje pretežni del diplomantov dodiplomskih in podiplomskih študijskih programov. Nacionalnega pomena so predvsem inštituti kot sestavni deli FERI ter centri znanja, ki opravljajo prenos temeljnih in aplikativnih znanj v celoten prostor Republike Slovenije. Mednarodni pomen izkazuje fakulteta z vpetostjo v mednarodne raziskovalne tokove s številnimi mednarodnimi projekti, izmenjavo študentov in profesorjev, objavami v uglednih znanstvenih revijah, nastopih na mednarodnih konferencah in organizacijo le-teh.

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Fakulteta za matematiko, naravoslovje in informacijske tehnologije Univerze na Primorskem (UP FAMNIT) je prvo generacijo študentov vpisala v študijskem letu 2007/08, pod okriljem UP PEF pa so se že v študijskem letu 2006/07 izvajali podiplomski študijski programi Matematične znanosti in Računalništvo in informatika (magistrska in doktorska programa).

Z ustanovitvijo UP FAMNIT je v letu 2006 je Univerza na Primorskem pridobila svoje naravoslovno uravnoteženje. Sodobne tehnologije v naravoslovju predstavljajo na začetku tretjega tisočletja poseben izziv, saj morajo izpolniti interese hitrega razvoja družbe, kakor tudi skrb za kakovostno ohranjanje naravnega in družbenega ravnovesja. V tem matematična znanja, področje informacijske tehnologije in druga naravoslovna znanja predstavljajo ključ do odgovora pri vprašanjih modeliranja družbeno ekonomskih procesov, njihove logike in zakonitosti racionalnega razmišljanja.

ACM Slovenija

ACM je največje računalniško združenje na svetu s preko 80 000 člani. ACM organizira vplivna srečanja in konference, objavlja izvirne publikacije in vizije razvoja računalništva in informatike.

ACM Slovenija smo ustanovili leta 2001 kot slovensko podružnico ACM. Naš namen je vzdigniti slovensko računalništvo in informatiko korak naprej v bodočnost.

Društvo se ukvarja z:

- Sodelovanjem pri izdaji mednarodno priznane revije Informatica — za doktorande je še posebej zanimiva možnost objaviti 2 strani poročila iz doktorata.
- Urejanjem slovensko-angleškega slovarčka — slovarček je narejen po vzoru Wikipedije, torej lahko vsi vanj vpisujemo svoje predloge za nove termine, glavni uredniki pa pregledujejo korektnost vpisov.
- ACM predavanja sodelujejo s Solomonovimi seminarji.
- Sodelovanjem pri organizaciji študentskih in dijaških tekmovanj iz računalništva.

ACM Slovenija vsako leto oktobra izvede konferenco Informacijska družba in na njej skupščino ACM Slovenija, kjer volimo predstavnike.

IEEE Slovenija

Inštitut inženirjev elektrotehnike in elektronike, znan tudi pod angleško kratico IEEE (Institute of Electrical and Electronics Engineers) je svetovno združenje inženirjev omenjenih strok, ki promovira inženirstvo, ustvarjanje, razvoj, integracijo in pridobivanje znanja na področju elektronskih in informacijskih tehnologij ter znanosti.

REPUBLIKA SLOVENIJA
MINISTRSTVO ZA IZOBRAŽEVANJE, ZNANOST IN ŠPORT

Ministrstvo za izobraževanje, znanost in šport
Ministrstvo za izobraževanje, znanost in šport opravlja upravne in strokovne naloge na področjih predšolske vzgoje, osnovnošolskega izobraževanja, osnovnega glasbenega izobraževanja, nižjega in srednjega poklicnega ter srednjega strokovnega izobraževanja, srednjega splošnega izobraževanja, višjega strokovnega izobraževanja, izobraževanja otrok in mladostnikov s posebnimi potrebami, izobraževanja odraslih, visokošolskega izobraževanja, znanosti, ter športa.

Zavod
Republike
Slovenije
za šolstvo

Zavod Republike Slovenije za šolstvo
Zavod Republike Slovenije za šolstvo je osrednji nacionalni razvojno-raziskovalni in svetovalni zavod na področju predšolske vzgoje, osnovnega šolstva in splošnega srednješolskega izobraževanja.

ZLATI POKROVITELJ

Quintelligence

Obstoječi informacijski sistemi podpirajo predvsem procesni in organizacijski nivo pretoka podatkov in informacij. Biti lastnik informacij in podatkov pa ne pomeni imeti in obvladati znanja in s tem zagotavljati konkurenčne prednosti. Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja. IKT (informacijsko-komunikacijska tehnologija) je postavila temelje za nemoten pretok in hranjenje podatkov in informacij. S primernimi metodami je potrebno na osnovi teh informacij izpeljati ustrezne analize in odločitve. Nivo upravljanja in delovanja se tako seli iz informacijske logistike na mnogo bolj kompleksen in predvsem nedeterminističen nivo razvoja in uporabe metodologij. Tako postajata razvoj in uporaba metod za podporo obvladovanja znanja (knowledge management, KM) vedno pomembnejši segment razvoja.

Podjetje Quintelligence je in bo usmerjeno predvsem v razvoj in izvedbo metod in sistemov za pridobivanje, analizo, hranjenje in prenos znanja. S kombiniranjem delnih — problemsko usmerjenih rešitev, gradimo kompleksen in fleksibilen sistem za podporo KM, ki bo predstavljal osnovo globalnega informacijskega centra znanja.

Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja.