4. tekmovanje IJS v znanju računalništva

Institut Jožef Stefan, Ljubljana, 28. marca 2009

Bilten

Bilten 4. tekmovanja IJS v znanju računalništva
Institut Jožef Stefan, 2009

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Andrej Bauer, Jure Ferlež, Primož Gabrijelčič, Boris Gašperin, Miha Grčar, Tomaž Hočevar, Uroš Jovanovič, Mitja Lasič, Jure Leskovec, Mark Martinec, Polona Novak, Marjan Šterk, Mitja Trampuš, Miha Vuk, Darko Zupanič, Anže Žagar, Klemen Žagar, Janez Brank

Tisk: Present d. o. o., Ljubljana
Naklada: 350 izvodov

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)

TEKMOVANJE IJS v znanju računalništva (4 ; 2009 ; Ljubljana)
Bilten / 4. tekmovanje IJS v znanju računalništva, Ljubljana, 28. marca 2009 ; [avtorji nalog Nino Bašić . . . [et al.] ; uredil Janez Brank]. — Ljubljana : Institut Jožef Stefan, 2009

Dostopno tudi na: `http://rtk.ijs.si/2009/rtk2009-bilten.pdf`

ISBN 978-961-264-014-9
1. Brank, Janez, 1979–
248588800

KAZALO

Struktura tekmovanja 5

Nasveti za 1. in 2. skupino 7
Naloge za 1. skupino 10
Naloge za 2. skupino 14
Navodila za 3. skupino 20
Naloge za 3. skupino 23
Naloge šolskega tekmovanja 30
Neuporabljene naloge iz leta 2007 34

Rešitve za 1. skupino 46
Rešitve za 2. skupino 54
Rešitve za 3. skupino 60
Rešitve šolskega tekmovanja 73
Rešitve neuporabljenih nalog 2007 76

Nasveti za ocenjevanje
in izvedbo šolskega tekmovanja 106

Rezultati 110
Nagrade 114
Šole in mentorji 115

Tekmovanje programov: Labirint 116

Anketa 120
Rezultati ankete 124
Cvetke 131

Sodelujoče inštitucije 135
Pokrovitelji 139

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. Možnost, da tekmovalci v prvi in drugi skupini svoje odgovore natipkajo na računalniku, smo letos ponudili prvič; bila je lepo sprejeta, saj so se zanjo odločili praktično vsi. Da bi bilo tekmovanje pošteno tudi do morebitnih reševalcev na papir, pa so bili na računalnikih za prvo in drugo skupino le urejevalniki besedil, ne pa tudi razvojna orodja, prevajalniki in dokumentacija o programskih jezikih in knjižnicah.

V tretji skupini rešujejo vsi tekmovalci naloge na računalnikih, za kar imajo pet ur časa. Pri vsaki nalogi je treba napisati program, ki prebere podatke iz vhodne datoteke, izračuna nek rezultat in ga izpiše v izhodno datoteko. Programe se potem ocenjuje tako, da se jih na ocenjevalnem računalniku izvede na več testnih primerih, število točk pa je sorazmerno s tem, pri koliko testnih primerih je program izpisal pravilni rezultat. (Podrobnosti točkovanja v 3. skupini so opisane na strani 21.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C# in java.

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

Tekmovalci vseh treh skupin si lahko pri reševanju pomagajo z zapiski in literaturo, v tretji skupini pa tudi z dokumentacijo raznih prevajalnikov in razvojnih orodij, ki so nameščena na tekmovalnih računalnikih.

Na začetku smo tekmovalcem razdelili tudi list z nekaj nasveti in navodili (str. 7–9 za 1. in 2. skupino, str. 20–22 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.

Poleg tekmovanja v znanju računalništva smo organizirali tudi tekmovanje programov, ki je podrobneje predstavljeno na straneh 116–119.

Novost v primerjavi s prejšnjimi leti je šolsko tekmovanje, ki smo ga izvedli 6. februarja 2009. To je potekalo v eni sami težavnostni skupini, 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. 106–109). 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. Čeprav je bil prvotni razlog za uvedbo šolskega tekmovanja zunanjega značaja (novi pravilnik o sofinanciranju šolskih tekmovanj), pa se je izkazalo šolsko tekmovanje tudi sicer za koristno, saj je povečalo zanimanje za naše tekmovanje tudi na nekaterih šolah, ki se ga prej niso udeleževale, in je verjetno precej pripomoglo k temu, da je tudi na državnem tekmovanju letos precej več tekmovalcev kot prejšnja leta (letos 104, lani 87, predlani 79).

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;
```

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). Za manjše sintaktične napake se ne odbije veliko točk. Priporočljivo in zaželeno je, da so tvoje rešitve napisane pregledno in čitljivo. Če je na listih, ki jih oddajaš, več različic rešitve za kakšno nalogo, jasno označi, katera je tista, ki naj jo ocenjevalci upoštevajo.

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

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

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

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

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

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

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

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

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

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

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

```
#include <stdio.h>

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

Še isti trije primeri v pythonu:

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

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

```
# Branje standardnega vhoda po vrsticah:
import sys

i = d = 0
for s in sys.stdin:
  s = s.rstrip('\n') # odrežemo znak za konec vrstice
  i += 1; d += len(s)
  print "%d. vrstica: \"%s\"" % (i, s)
print "%d vrstic, %d znakov." % (i, d)
```

```
# Branje standardnega vhoda znak po znak:
import sys

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

Še isti trije primeri v javi:

```
// Branje dveh števil in izpis vsote:
import java.io.*;
import java.util.Scanner;

public class Primer1
{
    public static void main(String[] args) throws IOException
    {
        Scanner fi = new Scanner(System.in);
        int i = fi.nextInt(); int j = fi.nextInt();
        System.out.println(i + " + " + j + " = " + (i + j));
    }
}
```

```
// Branje standardnega vhoda po vrsticah:
import java.io.*;

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

```
// Branje standardnega vhoda znak po znak:
import java.io.*;

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

NALOGE ZA PRVO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom in oddaš kot tekstovno datoteko (`*.txt`) ali pa oddaš del odgovorov na papirju in del v datoteki. Vse te možnosti so enakovredne. Odgovore, oddane v datotekah, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

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. Liste z nalogami lahko po tekmovanju obdržiš.

1. Kolikokrat najmanjši

Napiši program, ki prebere zaporedje celih števil in izpiše, kolikokrat se v tem zaporedju pojavi najmanjše število.

Primer: če imamo zaporedje $10, 5, 8, 7, 5, 5, 20, 7, 8, 8, 8, 6$, je pravilni rezultat 3 — najmanjše število v zaporedju je 5, ki se v njem pojavi trikrat.

Tvoj program naj bere števila s standardnega vhoda, vsako je v svoji vrstici, vsa pa so cela števila, večja od 0. Predpostaviš lahko, da zaporedje ni prazno. Program naj bere zaporedje vse do konca standardnega vhoda (EOF).

2. Označevanje kovancev

Izdelovalec spominskih kovancev bi rad izdelal serijo dvajsetih milijonov kovancev, vsak pa naj bi imel enolično oznako, sestavljeno iz števk in nekaterih velikih črk. Oznaka naj bi bila dovolj kratka, da si jo je lažje zapomniti ali prepisati, hkrati pa bi bilo pametno, da ne uporabimo v oznaki črk, ki so preveč podobne števkam ali drugim črkam, zato da se izognemo najbolj pogostim zmotam pri odčitavanju oznake. Tako izločimo črke `I`, `L`, `O`, `Q`, `S`, `U`, uporabimo pa vse ostale črke angleške abecede in vse števke od `0` do `9`. Tako nam ostane 30 znakov:

`0123456789ABCDEFGHJKMNPRTVWXYZ`

Izkaže se, da oznaka, sestavljena iz 5 znakov tega nabora, zadošča, da enolično označimo (oštevilčimo) vse kovance.

Napiši program, ki izpiše dvajset milijonov različnih petmestnih oznak, sestavljenih iz gornjih 30 znakov (posamezen znak se sme v posamezni oznaki pojavljati tudi več kot enkrat). Ker je vseh možnih petmestnih oznak več kot dvajset milijonov, je vseeno, katerih dvajset milijonov izmed njih izpiše tvoj program (samo da so vse različne). Vseeno je tudi, v kakšnem vrstnem redu jih izpiše. Vsako oznako naj tvoj program izpiše v svojo vrstico.

3. Citati

Recimo, da bi radi citirali razne strani iz neke knjige; to naredimo tako, da naštejemo številke teh strani v strogo naraščajočem vrstnem redu, na primer 2, 5, 6, 8, 11, 28, 29, 30, 31, 67. Če se v tem seznamu kdaj pojavita dve ali več zaporednih strani, ga lahko zapišemo krajše: obdržimo le prvo in zadnjo številko strani iz take skupine več zaporednih številk, med njiju pa zapišimo vezaj: 2, 5–6, 8, 11, 28–31, 67.

Napiši program, ki prebere seznam številk strani s standardnega vhoda (pri čemer bo vsaka številka v svoji vrstici), na standardni izhod pa izpiše ta seznam v zgoraj določeni obliki (v eni vrstici, z vejicami, presledki in vezaji). V vhodnem seznamu so številke strani že podane v naraščajočem vrstnem redu, vse pa so cela števila, večja od 0. Predpostaviš lahko, da je v vhodnem seznamu vsaj ena številka in da se nobena številka v njem ne pojavi več kot enkrat.

4. Smrkci

Po tisočletjih življenja v komuni se nekega dne tudi v deželo Smrkcev prikrade kapitalizem. V času tranzicije so si postopno razgrabili svojo vasico tako, da si je osem tajkunosmrkcev eden za drugim prilastilo po polovico še nerazdeljenega ozemlja. Vsak je lahko izbiral med severno, južno, zahodno in vzhodno polovico preostanka.

Napiši podprogram Nerazdeljeno, ki ugotovi, katero območje je ostalo nerazdeljeno. Celotna vas Smrkcev se razprostira na kvadratni površini velikosti 256×256 metrov, koordinate vasi pa se merijo od severozahodnega oglišča $(0, 0)$ do skrajnega jugovzhoda $(256, 256)$. Koordinate poljubnega pravokotnega območja znotraj vasi lahko zapišemo kot zaporedje štirih vrednosti (najprej x in y koordinata SZ oglišča in nato x in y koordinata JV oglišča); celotna vas ima torej koordinate $(0, 0, 256, 256)$.

Tvoj podprogram naj bo takšne oblike:

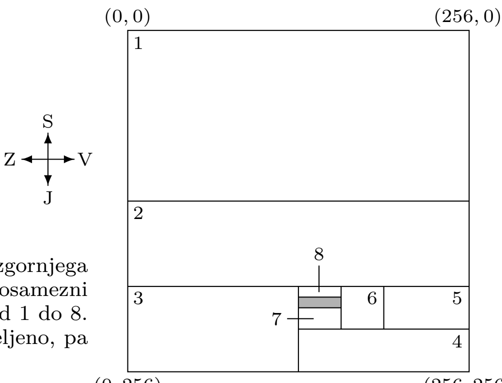
```
procedure Nerazdeljeno(Delitve: string);                { v pascalu }
void Nerazdeljeno(char* delitve);                      /* v C/C++ */
public static void Nerazdeljeno(String delitve);       /* v javi */
def Nerazdeljeno(delitve): ...                         # v pythonu
```

Parameter Delitve je niz znakov (`S`, `J`, `Z` in `V`), ki opisujejo izbire tajkunosmrkcev v takem vrstnem redu, po kakršnem so si delili vas. Tvoj podprogram naj izpiše na standardni izhod koordinate območja, ki ostane nerazdeljeno po vseh delitvah iz niza Delitve.

Opiši tudi, kaj bi bilo treba v tvojem programu spremeniti, če bi se spremenila velikost ozemlja in/ali število tajkunosmrkcev. Zaželeno je, da je tvoja rešitev takšna, da bi v takem primeru potrebovala čim manj sprememb.

Primer: recimo, da dobimo niz `SSZJVVJS`.

Prvi tajkunosmrkec si je tako vzel severno (zgornjo) polovico celotnega ozemlja, torej ostane južna (spodnja) polovica: $(0, 128, 256, 256)$. Drugi izbere severno polovico preostanka, ostane $(0, 192, 256, 256)$. Tretji izbere zahodno (levo) polovico, ostane $(128, 192, 256, 256)$. Nato vzame četrti južno polovico, ostane $(128, 192, 256, 224)$. Peti vzhodno (desno) polovico, ostane $(128, 192, 192, 224)$. Šesti prav tako vzhodno polovico, ostane $(128, 192, 160, 224)$. Sedmi južno polovico, ostane $(128, 192, 160, 208)$. Nazadnje vzame osmi tajkunosmrkec severno polovico, ostane $(128, 200, 160, 208)$, kar je tudi končni odgovor.

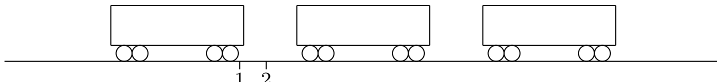


Slika na desni kaže ozemlje po delitvah iz zgornjega primera. Območja, ki so si jih prisvojili posamezni tajkunosmrkci, so označena s številkami od 1 do 8. Območje, ki je na koncu ostalo še nerazdeljeno, pa je pobarvano sivo.

5. Železnica

Po nekem železniškem tiru vozijo vlaki v obe smeri. Hitrost vlakov je različna, od 10 do 100 km/h. Vlaki so sestavljeni iz vagonov in lokomotive, ki so vsi dolgi po 10 m, med njimi pa je 2 m prostora.

Na tiru sta nameščena dva optična senzorja. Razdalja med njima je 1 m. Ko lokomotiva in vagoni potujejo mimo senzorjev, prekinejo svetlobni žarek, sicer pa je žarek neprekinjen (tudi npr. v presledku med dvema vagonoma). (Senzorja zaznavata lokomotivo povsem enako kot vagone, zato bomo v preostanku te naloge tudi lokomotive imenovali vagoni.) S pomočjo teh dveh senzorjev bi radi šteli, koliko vagonov se v nekem času zapelje po našem tiru, in sicer posebej za vsako smer vožnje (levo in desno).



Na zgornji sliki imamo na primer vlak s tremi vagoni; žarek senzorja 1 je trenutno prekinjen, senzorja 2 pa ne. Če bi se vlak premaknil malo proti levi, tudi žarek senzorja 1 ne bi bil več prekinjen.

Napiši tri **podprograme**: Inicializacija, SpremembaSenzorja in Izpis.

- **void** Inicializacija() — sistem lahko ta podprogram pokliče večkrat, pri čemer pa zagotavlja, da ga bo vsaj enkrat poklical še pred prvim klicem podprogramov SpremembaSenzorja in Izpis. Predpostavi, da v času tega klica nobeden od senzorjev nima prekinjenega žarka.

- **void** SpremembaSenzorja(**int** senzor, **bool** prekinjen) — sistem ga bo poklical ob vsaki spremembi stanja senzorja; senzor pove številko senzorja (1 = levi, 2 = desni), prekinjen pa, ali je svetlobni žarek tega senzorja po novem prekinjen ali ne (vrednost **true** pomeni, da je žarek prekinjen in nad senzorjem stoji vagon, vrednost **false** pa, da žarek ni prekinjen in nad senzorjem ni vagona).

- **void** Izpis() — naj izpiše na standardni izhod število vagonov, ki so prevozili naša senzorja po zadnjem klicu podprograma Inicializacija. Posebej naj izpiše število vagonov, ki so peljali v levo, in posebej število vagonov, ki so peljali v desno. Predpostavi, da bo sistem poklical Izpis le v takih trenutkih, ko nobeden od obeh senzorjev nima prekinjenega žarka.

Če hočeš, lahko poleg teh treh podprogramov deklariraš tudi kakšne globalne spremenljivke.

Predpostavi, da se vlaki med vožnjo mimo senzorjev ne ustavljajo in ne spreminjajo smeri vožnje. Predpostaviš lahko tudi, da medtem ko se izvaja kakšen od tvojih podprogramov, sistem ne bo poklical še enega od njih (niti istega podprograma).

Še deklaracije v drugih jezikih:

```
procedure Inicializacija;
procedure SpremembaSenzorja(Senzor: integer; Prekinjen: boolean);
procedure Izpis;

public static void inicializacija();
public static void spremembaSenzorja(int senzor, boolean prekinjen);
public static void izpis();

def Inicializacija(): ...
def SpremembaSenzorja(senzor, prekinjen): ...
def Izpis(): ...
```

NALOGE ZA DRUGO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom in oddaš kot tekstovno datoteko (`*.txt`) ali pa oddaš del odgovorov na papirju in del v datoteki. Vse te možnosti so enakovredne. Odgovore, oddane v datotekah, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

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. Liste z nalogami lahko po tekmovanju obdržiš.

1. Soglasniški podnizi

Na standardnem vhodu imamo zapisane besede, vsako v svoji vrstici. V njih nastopajo le male črke angleške abecede, nobene številke ali ločila ni. Nobena beseda ni daljša od 100 znakov. Zanima nas beseda, v kateri se nahaja najdaljši strnjeni podniz soglasnikov. Za soglasnike štejemo vse črke, ki niso samoglasniki (*a*, *e*, *i*, *o*, *u*).

Tule je nekaj primerov besed, poleg vsake besede smo zapisali iskano dolžino, podzaporedje pa zapisali v krepkem tisku:

i**n**	1
pri**stn**o	3
toreador	1
strjenka	4
a	0
skrbstvo	7
bese**dn**a	2
krt	3

Napiši program, ki bo prebral zaporedje besed s standardnega vhoda in nato izpisal na standardni izhod besedo, ki je vsebovala najdaljši strnjeni podniz soglasnikov. V zgornjem primeru bi torej izpisal besedo „`skrbstvo`“. Če je več besed z enako največjo dolžino podniza, je vseeno, katero od njih bo tvoj program izpisal. Obdela naj vse besede do konca standardnega vhoda (EOF).

2. Kje sem že to posnel?

Na večdnevnem potepanju po naravnih parkih smo z digitalnim fotoaparatom posneli veliko fotografij. K vsaki fotografiji se je zapisal tudi točen čas njenega nastanka. Žal naš model fotoaparata nima možnosti povezave s sprejemnikom GPS za satelitsko navigacijo, ki je ves čas beležil, kje se nahajamo, vendar se je v pomnilniku sprejemnika GPS ohranila pot kot zaporedje točk, kjer vsaka točka hrani zemljepisne koordinate

in točen čas meritve. Te meritve so bile pogoste (denimo v povprečju nekaj točk na minuto, a ne nujno v enakomernih razmikih), vendar niso nujno točno časovno sovpadale s trenutki, ko smo posneli fotografije.

Po vrnitvi s počitnic bi radi v fotografije dodali podatek o zemljepisnih koordinatah njihovega nastanka. Vse fotografije imamo urejene po času, ko so bile posnete; prav tako so tudi vse točke, ki jih je izmeril in shranil sprejemnik za navigacijo, shranjene v datoteki in urejene po času meritve.

Napiši program, ki bo vsaki fotografiji dodal zemljepisne koordinate tiste točke, ki je časovno najbližje času nastanka fotografije. Za dostop do fotografij in dodajanje koordinat vanje sta na voljo podprograma:

```
int PreberiNaslednjoSliko();
void VpisiKoordinateInShrani(double x, double y);
```

Funkcija PreberiNaslednjoSliko prebere v pomnilnik naslednjo fotografijo in vrne čas njenega nastanka; če pa smo že prišli do konca in fotografij ni več, vrne funkcija vrednost 0. Čas se pri tej nalogi meri v sekundah od nekega fiksnega trenutka na koledarju (tako za čase slik kot tudi za čase meritev GPS).

Podprogram VpisiKoordinateInShrani vnese podatek o zemljepisnih koordinatah v fotografijo, ki je trenutno prebrana in čaka v pomnilniku, hkrati pa tako dopolnjeno fotografijo tudi shrani nazaj v njeno datoteko.

Podatke iz sprejemnika GPS dobimo na standardnem vhodu; v vsaki vrstici so tri števila, prvo je čas meritve (celo število, ki predstavlja število sekund od nekega fiksnega trenutka na koledarju; na enak način je zapisan tudi čas v fotografijah), sledita pa mu dve realni števili, ki predstavljata zemljepisne koordinate, kjer smo se nahajali v tistem trenutku. Teh točk je preveč, da bi jih lahko vse hkrati shranili v pomnilnik.

Še deklaracije v pascalu, javi in pythonu:

```
function PreberiNaslednjoSliko: integer;
procedure VpisiKoordinateInShrani(x, y: real);

public static int preberiNaslednjoSliko();
public static void vpisiKoordinateInShrani(double x, double y);

def PreberiNaslednjoSliko(): ... # vrne int
def VpisiKoordinateInShrani(x, y): ...
```

3. Avtocesta

Po nekem enosmernem odseku ceste se vozijo tovornjaki, opremljeni z oddajno-sprejemnimi napravami. Na začetku in na koncu odseka stojita ob cesti merilni postaji. Ko se tovornjak pripelje mimo postaje na začetku odseka, mu le-ta dodeli zaporedno številko in mu jo pošlje, naprava na tovornjaku pa si jo zapomni. Ko pa se tovornjak pripelje mimo postaje na koncu odseka, naprava na tovornjaku sporoči tej postaji zaporedno številko, ki jo je prejela ob zadnji vožnji mimo začetne postaje. S pomočjo vse te mašinerije lahko poskusimo odkrivati tovornjakarje, ki so odsek prevozili prehitro.

Na nekem računalniku imamo možnost pripraviti svoj podprogram, ki ga bo sistem poklical vsakič, ko pride nek tovornjak mimo kakšne od postaj. Kot parametra dobi dve celi števili: številko postaje (1 za začetno postajo, 2 za končno) in zaporedno številko, ki jo je temu tovornjaku dodelila začetna postaja. Številke, ki jih začetna postaja dodeljuje tovornjakom, so naravna števila od 1 naprej, tovornjaki pa jih dobijo v takšnem vrstnem redu, v kakršnem so se pripeljali mimo začetne postaje. Seveda pa ni nujno, da pripeljejo tovornjaki v enakem vrstnem redu tudi mimo končne postaje. Poleg tega so med začetno in končno postajo na cesti tudi križišča, tako da se lahko zgodi, da nek tovornjak pripelje le mimo začetne postaje, mimo končne pa ne, ali pa obratno. Če pride mimo končne postaje tovornjak, ki ni bil še nikoli na začetni postaji, dobimo pri njem kot zaporedno številko vozila vrednost 0. Predpostaviš pa lahko, da sta naši dve postaji edini tovrstni in da če torej pripelje mimo naše končne postaje tovornjak z neko številko, jo je dobil pri prehodu naše začetne postaje (in ne na primer kje drugje).

Radi bi, da bi ta podprogram vsakič, ko ugotovi, da je nek tovornjak za vožnjo od začetne do končne postaje porabil manj kot MinimalniCas sekund, izpisal številko tega tovornjaka na standardni izhod. (Pri tem je MinimalniCas neka konstanta, za katero lahko predpostaviš, da je že definirana.) **Opiši postopek**, ki bi ga moral izvajati tak podprogram, da bi deloval na željeni način. Predpostaviš lahko, da že obstaja neka funkcija TrenutniCas, ki jo lahko kadarkoli pokličeš in od nje izveš trenutni čas, merjen v sekundah od nekega fiksnega začetnega trenutka v preteklosti.

Upoštevaj, da se bo tvoj podprogram izvajal na počasnem računalniku z malo pomnilnika, zato naj bo tvoja rešitev učinkovita in naj pazi, da količina porabljenega pomnilnika ne bo mogla rasti v nedogled (četudi to pomeni, da lahko v primeru zelo gostega prometa kakšnega prehitrega voznika spregleda).

Tvoja rešitev lahko uporablja tudi globalne spremenljivke, ki jih lahko tudi po svoje inicializiraš.

4. UTF-5

V standardu Unicode je vsak znak (črka, števka, ločilo ipd.) predstavljen z enim od celih števil od 0 do 1 114 111; da predstavimo številko posameznega znaka, je torej včasih potrebnih kar 21 bitov. Nekateri avtorji so predlagali postopek UTF-5, ki številko znaka takole predstavi z enim ali več 5-bitnimi števili:

1. Zapišemo številko znaka v dvojiškem zapisu.
2. Če je treba, vrinemo na levi še nekaj ničel, dokler ni skupno število bitov večkratnik števila štiri.
3. Razbijemo zaporedje bitov na skupine po štiri.
4. Pri vsaki skupini vrinemo na levi še eno ničlo, razen pri zadnji skupini, kjer vrinemo enico.

Primer: znaku ṇ pripada v standardu Unicode kodno mesto 7751, kar je v dvojiškem zapisu enako $1\,1110\,0100\,0111_2$. Iz tega torej dobimo četverice bitov 0001, 1110, 0100 in 0111, zato je predstavitev tega znaka po kodiranju UTF-5 naslednja skupina štirih peteric bitov:[1]

00001 01110 00100 10111

torej števila $1, 14, 4, 23$.

Recimo, da nam nekdo po počasnem komunikacijskem kanalu pošilja številke znakov, zakodirane v peterice bitov po postopku UTF-5. **Napiši podprogram** PrejemPeterice, ki ga bo sistem poklical vsakič, ko od pošiljatelja prejme novo peterico bitov. Tvoj podprogram naj iz prejetih podatkov sproti sestavlja številske vrednosti znakov; takoj ko dekodira celotno številko znaka, naj pokliče podprogram PrejemZnaka, za katerega lahko predpostaviš, da že obstaja in bo poskrbel za nadaljnjo obdelavo prejetega znaka. Predpostaviš lahko tudi, da v podatkih, ki prihajajo po komunikacijskem kanalu do tvojega podprograma PrejemPeterice, ni napak.

Tvoj podprogram naj bo takšne oblike:

```
procedure PrejemPeterice(x: 0..31);            { v pascalu }
void PrejemPeterice(int x);                    /* v C/C++ */
public static void prejemPeterice(int x);      // v javi
def PrejemPeterice(x): ...                     # v pythonu
```

Prejeta peterica bitov, x, bo seveda vedno eno od celih števil 0, ..., 31.

Podprogram PrejemZnaka pa je takšne oblike:

```
procedure PrejemZnaka(StZnaka: integer);          { v pascalu }
void PrejemZnaka(int StZnaka);                    /* v C/C++ */
public static void prejemZnaka(int stZnaka);      // v javi
def PrejemZnaka(StZnaka): ...                     # v pythonu
```

[1] To, koliko peteric bitov nastane, je odvisno od vrednosti znaka — včasih zadošča že ena peterica, včasih jih je treba kar šest. Iz gornjega besedila to mogoče ni dovolj jasno razvidno, saj je na tekmovanju nekaj tekmovalcev napisalo rešitve, ki predpostavljajo, da je vsak znak razbit na točno štiri peterice bitov.

Tega podprograma ne piši ti, pač pa predpostavi, da že obstaja in ga lahko preprosto pokličeš, ko ga potrebuješ.

Tvoja rešitev lahko uporablja tudi globalne spremenljivke, za katere lahko tudi predpišeš začetno vrednost (torej vrednost, ki jo bodo imele pred prvim klicem tvojega podprograma PrejemPeterice).

5. break considered harmful

Zamislimo si preprost programski jezik, podoben malo poenostavljenemu in okleščenemu pascalu ali C-ju. Program v njem je sestavljen iz množice podprogramov; na začetku vsakega podprograma so deklaracije spremenljivk, temu pa sledi zaporedje stavkov. Vsak stavek je ene od naslednjih oblik:

- Prireditveni stavek: Spremenljivka = Izraz;
- Klic podprograma: ImePodprograma(Izraz1, ..., IzrazN);
- Zaporedje stavkov: $\{S_1\ S_2\ \ldots\ S_n\}$
- Pogojni stavek: **if** (Pogoj) S — če je izpolnjen pogoj Pogoj, izvede stavek S.
- Pogojni stavek: **if** (Pogoj) S_1 **else** S_2 — če je izpolnjen pogoj Pogoj, izvede stavek S_1, sicer pa stavek S_2.
- Zanka: **while** (Pogoj) S — izvaja stavek S v zanki, pred vsakim izvajanjem pa preveri, če je pogoj Pogoj izpolnjen; če ni, se zanka konča.
- Stavek **continue**; — izvajanje skoči na konec trenutne iteracije najbolj notranje zanke.
- Stavek **break**; — izvajanje skoči ven iz najbolj notranje zanke.

Pri tem v zgornjih primerih Izraz in Pogoj predstavljata poljuben izraz, v katerem lahko nastopajo aritmetični, logični in primerjalni operatorji, konstante, klici funkcijskih podprogramov in podobno.

Izkaže se, da niso vse zgoraj naštete vrste stavkov nujno potrebne; nekaterim med njimi se lahko odpovemo in enak učinek dosežemo tudi kako drugače. Na primer, recimo, da bi se hoteli znebiti pogojnih stavkov z **else** in uporabljati le pogojne stavke brez **else**. Izvorno kodo podprograma, ki uporablja **else**, lahko čisto avtomatsko predelamo v izvorno kodo podprograma, ki deluje popolnoma enako (daje enake rezultate pri enakih vhodnih podatkih), pri tem pa ne uporablja **else**. To lahko naredimo takole: vse stavke **if** ... **else** v našem podprogramu v mislih oštevilčimo od 1 do n (če je n skupno število vseh teh stavkov); nato na začetku programa dodajmo deklaracije n novih logičnih spremenljivk (torej tipa **bool** oz. boolean); recimo, da bo tem spremenljivkam ime Pogoj1, Pogoj2, itd., čeprav seveda lahko uporabimo tudi kakšna drugačna imena, da se le ne tepejo z imeni že obstoječih spremenljivk. Nato pa k-ti stavek **if** ... **else** v našem podprogramu spremenimo iz prvotne oblike

if (P) S_1 **else** S_2

v takšen stavek:

{ Pogoj$k$ = P; **if** (Pogoj$k$) S_1 **if** (! Pogoj$k$) S_2 }

Po tej zamenjavi program deluje enako kot prej, ne uporablja pa več besede **else**. Seveda moramo to narediti za vsakega od stavkov **if** ... **else** v našem podprogramu (torej za vsak k od 1 do n). (V gornjem primeru smo predpostavili, da operator „!" pomeni logično negacijo; v nekaterih jezikih, npr. pascalu, bi namesto „!" pisali „**not**".)

Opiši postopek, ki na podoben način predela poljuben podprogram tako, da bo deloval enako kot prej, vendar ne bo več uporabljal stavka **break** (lahko pa še vedno uporablja **continue** in tudi vse druge zgoraj naštete stvari, vključno z **else**).[2]

Če ne znaš opisati splošnega postopka, ki bi predelal poljuben podprogram v skladu z zahtevami naloge, poskusi vsaj ročno predelati spodnji podprogram tako, da bo deloval enako kot zdaj, ne bo pa več uporabljal stavka **break**. Takšna rešitev lahko dobi pri tej nalogi največ 7 točk (od 20 možnih).

```
podprogram DveZanki;
spremenljivke: i, j, n — cela števila;
{
  n = 100; i = 0;
  while (i < n)
  {
    j = i;
    while (j < n)
    {
      Izpis1(i, j);
      if (Funkcija1(i, j)) break;
      Izpis2(i, j);
      j = j + 1;
    }
    if (Funkcija2(i, j)) break;
    while (j > i)
    {
      Izpis3(i, j);
      j = j − 1;
    }
    i = i + 1;
  }
}
```

Pri tem predpostavi, da so podprogrami Izpis1, Izpis2, Izpis3, Funkcija1 in Funkcija2 že deklarirani, ne moremo pa jih spreminjati in tudi podrobnosti njihovega delovanja nam niso znane.

[2] Zanimivo, vendar malo težjo različico naloge dobimo, če zahtevamo, da se namesto stavka **break** odpravi stavek **continue** ali pa stavek **if**. Še večji izziv je, če hočemo odpraviti dva od teh stavkov hkrati ali pa celo vse tri. V podobnem duhu lahko odpravljamo tudi večnivojski **break** in **continue** ali pa eksotični stavek **redo**, ki skoči na začetek trenutne iteracije najbolj notranje zanke (namesto na konec kot **continue**).

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

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

Delovno okolje

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

Na vsakem računalniku imaš na voljo enoto (disk) `U:`, na kateri lahko kreiraš svoje datoteke (datoteke, ki so tam že od prej, pusti pri miru). Programi naj bodo napisani v programskem jeziku pascal, C, C++, C# ali java, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz JDK 1.6 in s prevajalnikom za C# iz Visual Studia 2008. Za delo lahko uporabiš `FP` oz. `ppc386` (FreePascal), `GCC/G++` (GNU C/C++ — command line compiler), `GCJ` (za javo 1.4), Java 2 SDK (za javo 1.6) in Visual Studio 2008.

Oglej si tudi spletno stran: `http://rtk/`, kjer boš dobil nekaj testnih primerov in program `RTK.EXE`, ki ga lahko uporabiš za oddajanje svojih rešitev. Tukaj si lahko tudi ogledaš anonimizirane rezultate ostalih tekmovalcev.

Preden boš oddal prvo rešitev, boš moral programu za preverjanje nalog sporočiti svoje ime, kar bi na primer Janez Novak storil z ukazom

```
rtk -name JNovak
```

(prva črka imena in priimek, brez presledka, brez šumnikov).

Za oddajo rešitve uporabi enega od naslednjih ukazov:

`rtk` *imenaloge*.`pas`
`rtk` *imenaloge*.`c`
`rtk` *imenaloge*.`cpp`
`rtk` *ImeNaloge*.`java`
`rtk` *ImeNaloge*.`cs`

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Datoteka z izvorno kodo, ki jo oddajaš, ne sme biti daljša od 30 KB. Na spletni strani boš dobil za vsak testni primer obvestilo o tem, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund ali pa porabil več kot 200 MB pomnilnika, 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. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku, razen s predpisano vhodno in izhodno datoteko. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku), prenosnih računalnikov, prenosnih telefonov itd.

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi od 0 do 10 točk (praviloma 10, če je izpisal popolnoma pravilen odgovor, sicer pa 0; izjema je 5. naloga, kjer dobijo boljše rešitve več točk kot slabše), Nato se te točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M - 3(N - 1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

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

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

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

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

Primer vhodne datoteke:

```
123 456
```

Ustrezna izhodna datoteka:

```
5790
```

Primeri rešitev (dobiš jih tudi kot datoteke na `http://rtk/`):

- V pascalu:

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

- V C-ju:

```
#include <stdio.h>

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

- V C++:

```
#include <fstream>

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

- V javi:

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

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

- V C#:

```
using System.IO;

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

NALOGE ZA TRETJO SKUPINO

1. Undo (`undo.in`, `undo.out`)

Napredno podjetje na področju urejevalnikov besedila se je odločilo za povsem nov pristop. Namesto večanja števila operacij, ki so na voljo uporabniku, so se odločili število operacij drastično zmanjšati. Svoj novi produkt nameravajo poimenovati RISE (Reduced Instruction Set Editor), tebe pa so prosili za pomoč pri razvoju.

Odločili so se obdržati le dva ukaza, enega za dodajanje besede na konec trenutnega besedila in enega za razveljavitev nekaj predhodnih ukazov. Ukaz za razveljavitev pa ima neko posebnost, z njim lahko namreč razveljavimo tudi več prejšnjih razveljavitvenih ukazov.

Napiši program, ki prebere zaporedje ukazov in izpiše besedilo, ki nastane po izvedbi vseh prebranih ukazov.

Vhodna datoteka: v prvi vrstici je celo število n, ki pove, koliko ukazov sledi (velja $1 \leq n \leq 100\,000$). Sledi n vrstic, v vsaki je po en ukaz. Ukazi za dodajanje besede bodo imeli obliko `WRITE <beseda>`. Razveljavitveni ukazi pa bodo v obliki `UNDO <k>`, kjer k pomeni število predhodnih ukazov, ki jih želimo razveljaviti. Število k ne bo nikoli večje od števila dotlej izvedenih ukazov. Posamezne besede bodo krajše od 100 znakov in sestavljene samo iz velikih in malih črk angleške abecede, brez ločil ali presledkov.

Izhodna datoteka: vanjo izpiši besedilo, ki nastane po izvedbi vseh prebranih ukazov. Med besedami ne piši presledkov, na koncu besedila pa izpiši znak za konec vrstice.

Primer vhodne datoteke:

```
6
WRITE Danes
WRITE je
UNDO 1
WRITE lep
UNDO 2
WRITE dan
```

Pripadajoča izhodna datoteka:

```
Danesjedan
```

Razlaga: po prvem ukazu `UNDO` nam ostane besedilo „`Danes`“. Drugi ukaz `UNDO` razveljavi dva predhodna ukaza, torej tudi prvi razveljavitveni ukaz. Tako nastane besedilo „`Danesje`“, ki mu na koncu dodamo še niz „`dan`“.

2. Žaba in lokvanji (`zaba.in`, `zaba.out`)

V ribniku plava v ravni vrsti n enakomerno razpostavljenih lokvanjevih listov, ki jih v mislih po vrsti označimo s števili od 1 do n. Na bregu sedi žaba, ki se ji zahoče telovadbe, zato začne skakati z lista na list. Včasih se zgodi, da se list po tistem, ko se žaba odrine od njega, zaradi sunka trajno potopi.

Ker se je nad ribnik spustila megla, vidljivost ni prav dobra: žaba vidi samo liste, ki so od nje oddaljeni največ k. Povedano drugače, ko sedi na listu t, vidi samo liste $t-k, t-k+1, \ldots, t-1, t, t+1, \ldots, t+k-1, t+k$ (če niso že potopljeni). Kljub temu

se žaba tudi daljših skokov ne ustraši. Ne glede na vidljivost in razdaljo je sposobna kadarkoli skočiti s kateregakoli lista na kateregakoli drugega še nepotopljenega.

Napiši program, ki prebere, v kakršnem vrstnem redu žaba obiskuje liste, in izračuna, na katerem lokvanjevem listu bo žaba prvič videla okoli sebe samo vodo (ker bodo ostali listi bodisi potopljeni bodisi predaleč, da bi jih videla).

Vhodna datoteka: v prvi vrstici so tri cela števila, n, k in m, ločena s po enim presledkom. Zanje velja $1 < n \leq 10^6$, $1 \leq k \leq n$ in $1 \leq m \leq 2 \cdot 10^6$. Sledi m vrstic, ki navajajo številke listov v takšnem vrstnem redu, v kakršnem jih žaba obiskuje. V vsaki od teh vrstic sta dve celi števili, ločeni s presledkom: prvo od teh števil je številka lista, drugo število pa pove, ali se ta list potopi, ko žaba odskoči z njega (vrednost 1 pomeni, da se potopi, vrednost 0 pa, da se ne potopi). Po tistem, ko se nek list potopi, ostane trajno potopljen, žaba ga ne vidi več in tudi ne more več skočiti nanj.

V štirih od desetih testnih primerov te naloge bo veljalo tudi $n \leq 1000$ in $m \leq 2000$.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer številko lista, na katerem sedi žaba v prvem takem trenutku, ko ne vidi v svoji okolici nobenega lista. (Testni primeri, ki jih bomo uporabili pri tej nalogi, so sestavljeni tako, da žaba gotovo vsaj enkrat pristane na listu, s katerega ne vidi nobenega drugega lista.)

Primer vhodne datoteke:

```
9 2 11
3 1
6 0
2 1
7 0
6 1
9 0
5 1
4 0
1 1
8 0
4 0
```

Pripadajoča izhodna datoteka:

```
4
```

Razlaga: v zgornjem primeru je $k = 2$, zato lahko žaba z lista 4 vidi le liste 2, 3, 5 in 6, ki pa so pri zgornjem primeru vsi že potopljeni, ko žaba pride do lista 4.

3. Otoki (`otoki.in`, `otoki.out`)

Tuje obveščevalne službe že slutijo, da IJS izvaja jedrski oborožitveni program. Ker bi bilo morebitno razkritje usodno za institut (in posledično za tekmovanje), je treba centrifuge, ki v kletnih prostorih instituta bogatijo uran, nujno premestiti drugam. Na pomoč je priskočil general Pica z otočja San Serriffe in IJS-ju že ponudil velik kos ozemlja v osrčju dežele. Na tem ozemlju IJS išče skrivno lokacijo, kjer bo zgradil podzemne bunkerje.

Podana je karirasta mreža velikosti $w \times h$ ($3 \leq w \leq 3000$ in $3 \leq h \leq 3000$), ki predstavlja zemljevid območja. Vsak kvadratek je bodisi kopno (znak „`#`“) bodisi voda (znak „`.`“). Na tem zemljevidu se nahajajo otoki, ki imajo lahko jezera; na teh jezerih so spet otoki s svojimi jezeri itd. Vsako jezero ali otok je neka množica

kvadratkov, in sicer dva enako označena kvadratka spadata k istemu delu kopnega oz. morja natanko tedaj, če imata skupno celotno stranico. Zunanji rob zemljevida (torej prva in zadnja vrstica ter prvi in zadnji stolpec) je v celoti poplavljen. Vso vodo, do katere lahko pridemo z ladjo z zunanjega roba, bomo imenovali morje.

Stopnja varnosti nekega otoka je najmanjše število jezer, ki jih moramo prečkati, da se z morja prebijemo na ta otok. Kopno, ki je dostopno z morja (to je tisto kopno, po katerem se lahko ljudje, ki jih izkrcamo z ladje na morju, sprehajajo, ne da bi si pri tem zmočili šolne) ima stopnjo varnosti 0. **Napiši program**, ki prebere zemljevid in izpiše najvišjo stopnjo varnosti otoka, ki se pojavlja na tem zemljevidu.

Vhodna datoteka: v prvi vrstici sta celi števili w in h, ločeni s presledkom. Sledi h vrstic, v vsaki pa je w znakov, ki opisujejo zemljevid opazovanega območja; vsak od znakov na zemljevidu je „`.`“ ali „`#`“. Predpostaviš lahko, da je na zemljevidu vsaj en otok.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najvišjo stopnjo varnosti otoka, ki se pojavlja na zemljevidu iz vhodne datoteke.

Primer vhodne datoteke:

```
52 17
....................................................
............................#######........######...
........#############.......########.....###....###.
.......#####..###.######.....#########...##..#...##.
......####............######..####..###..##....###..
....#####..########..######.....#######..########...
....#####..#.....##..#####..........###..########...
.####.....##.###..#.....#####..#.##.###....###..###.
...####...#..#.#..##..#################......#####..
..###....##.####..##..................###.....####..
....##..##..###...#....########...##..#####.....##..
....#..###......###....##.###.#######.######...##...
..###.#############..###########..##..######..##....
..#...#######...########..............####..........
..#.......................#################.........
..###########################.......................
....................................................
```

Pripadajoča izhodna datoteka:

```
2
```

Razlaga: na tem zemljevidu imamo dva otoka stopnje 0, dva stopnje 1 in enega stopnje 2; pravilni odgovor je torej 2.

4. Strupi (`strupi.in`, `strupi.out`)

Primarij Krota z Naše male klinike je predlagal, da bi zaradi velikega števila pacientov z zastrupitvami v kolektiv vzeli kakšnega toksikologa. Poslovna direktorica je ta predlog zavrnila, ker ni ekonomske računice. Mi pa vemo, da lahko delo omenjenega specialista nadomesti dober računalniški program, zato jim ga bomo napisali.

Denimo, da v našem okolju obstaja n ($1 \leq n \leq 15$) strupov, ki so na dolgi rok vsi škodljivi. Ko pripeljejo pacienta v bolnišnico, ima v telesu nekatere od teh strupov. Naloga toksikologa je, da očisti pacienta vseh strupov. Toda kako? Znano je, da se nekateri strupi med seboj izničijo, če jih zmešamo skupaj. Terapija poteka tako, da v pacienta vnašamo ustrezne strupe, s katerimi izničimo tiste, ki jih že ima v organizmu. Naenkrat lahko vnesemo samo en strup. Ko je terapija končana, ne sme imeti pacient v telesu nobenega strupa več. Pri tem početju moramo biti skrajno previdni, ker lahko nekatere kombinacije povzročijo takojšnjo smrt. Pacient je pogubljen, četudi bi se kateri od strupov v tej mešanici izničili (to je počasen kemijski proces).

Na primer: recimo, da poznamo pet strupov, $\{1, 2, 3, 4, 5\}$, in da so znani naslednji podatki o njih. Kombinacije strupov, ki se izničijo: $\{1, 4\}$, $\{2, 3\}$, $\{1, 3, 5\}$.

Kombinacije strupov, ki povzročijo takojšnjo smrt: $\{3, 4\}$, $\{2, 4, 5\}$. Ko pripeljejo pacienta, ima ta v telesu $\{2, 4\}$. Kako lahko ravnamo v tem primeru? Strupa 3 mu ne smemo dati, ker bi dobili mešanico $\{2, 3, 4\}$, ta pa vsebuje $\{3, 4\}$, ki povzroči smrt. Tudi strupa 5 mu iz podobnega razloga ne smemo dodati. Če pa mu dodamo 1, dobimo $\{1, 2, 4\}$. Ta vsebuje $\{1, 4\}$, torej množico strupov, ki se med seboj izničijo, zato ostane v telesu le $\{2\}$. Če dodamo strup 3, dobimo $\{2, 3\}$. To je ravno množica, ki se izniči, in pacient je ozdravljen.

Če se v telesu po vnosu strupa pojavi hkrati več kombinacij, ki bi se lahko izničile, potem se izničijo vse. Npr. denimo, da se izničijo: $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$. V telesu imamo $\{2, 3, 4\}$, mi pa dodamo 1. V tem primeru ne bo v telesu nobenega strupa več, ker pride do izničenja pri vseh kombinacijah.

Naša naloga je, da ugotovimo, katere strupe moramo vnašati v bolnika, da ga bomo pozdravili. To bi radi storili na najkrajši možni način, kar pomeni s čim manj vnosi strupov v bolnika (vnesemo lahko samo po en strup naenkrat).

Vhodna datoteka: v prvi vrstici so cela števila n, m in k, ločena s po enim presledkom. Pri tem je n število strupov (ki so oštevilčeni s številkami od 1 do n), m je število smrtonosnih kombinacij strupov, k pa število kombinacij strupov, ki se med seboj izničijo. Sledi $m + k + 1$ vrstic, v vsaki je po ena kombinacija strupov: najprej je m vrstic s smrtonosnimi kombinacijami, nato k vrstic s kombinacijami, ki se izničijo, in nato še ena vrstica, ki pove kombinacijo strupov, ki jih ima pacient na začetku v telesu. Vsaka kombinacija je navedena tako, da je naprej napisano število strupov v njej (vsaj 1, največ n), nato pa so navedene številke teh strupov v naraščajočem vrstnem redu, ločene s presledki.

Veljalo bo $1 \le n \le 15$, $0 \le m \le 1000$ in $0 \le k \le 1000$.

Izhodna datoteka: vanjo izpiši zaporedje, v katerem bi morali vnašati strupe v pacienta. V prvi vrstici naj bo število strupov, ki jih moramo vnesti v pacienta. V naslednji vrstici pa naj bodo s presledki ločene številke strupov, v takem vrstnem redu, kot jih prejme pacient. Če je več takih zaporedij (minimalne dolžine), lahko izpišeš katerokoli izmed njih. Če primernega zaporedja sploh ni, naj tvoj program izpiše le eno vrstico, vanjo pa število -1.

Primer vhodne datoteke
(za primer iz besedila naloge):

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

Pripadajoča izhodna datoteka:

```
2
1 3
```

5. EPS (Emasculated PostScript) (`eps.txt`)

Mislimo si preprost in eksotičen programski jezik, pri katerem nimamo na voljo spremenljivk, podprogramov, zank while in for ter podobnega razkošja. Vse, kar imamo, je sklad in nekaj ukazov za prekladanje podatkov po njem. Ukazi delujejo tako, da pobirajo operande z vrha sklada in na vrh sklada tudi oddajajo svoje rezultate. Obstaja na primer ukaz `add`, ki pobere z vrha sklada dva operanda (ki morata biti

števili), in odloži na vrh sklada njuno vsoto. Pri tej nalogi predpostavimo, da so lahko elementi sklada le dveh tipov — cela števila (pri tej nalogi bomo ves čas delali s predznačenimi 32-bitnimi celimi števili, torej takimi kot pri tipu **int** oz. integer) ali pa značke (*marks*). Tvoj program je zaporedje takih ukazov in interpreter jih izvaja po vrsti, razen pri ukazu jnz, ki lahko povzroči skok drugam kot na naslednji ukaz v zaporedju. Pred vsakim ukazom lahko stoji labela (enolična oznaka ukaza; ime labele je zaporedje črk, števk in znakov _, od ukaza pa je ločeno z dvopičjem; dolgo je lahko največ 20 znakov, prvi znak pa ne sme biti števka), s katero se lahko na ta ukaz sklicuješ v skokih. Jezik podpira tudi komentarje, in sicer se vse od znaka % do konca vrstice šteje kot komentar.

Ukazi našega programskega jezika so zbrani v spodnji tabeli. V njej je vsak ukaz opisan tako, da je na levi napisan seznam operandov, ki jih pobere z vrha sklada (najbolj desni element je tisti, ki je čisto na vrhu), na desni pa je napisan seznam rezultatov, ki jih ukaz na koncu odloži na vrh sklada (spet je najbolj desni tisti, ki pride čisto na vrh).

operandi	ukaz	rezultati	opis
$x\ y$	exch	$y\ x$	zamenja vrhnja dva elementa
x	dup	$x\ x$	podvoji vrhnji element
x	pop		pobriše vrhnji element
$x_1\ \dots\ x_n\ n$	copy	$x_1\ \dots\ x_n\ x_1\ \dots\ x_n$	podvoji vrhnjih n elementov
$x_n\ \dots\ x_0\ n$	index	$x_n\ \dots\ x_0\ x_n$	podvoji $(n+1)$-ti element
$x_{n-1}\ \dots\ x_0\ n\ i$	roll	$x_{i-1}\ \dots\ x_0\ x_{n-1}\ \dots\ x_i$	ciklično zamakne vrhnjih n elementov za i mest proti vrhu sklada
	push(x)	x	odloži na vrh sklada konstanto x
$x\ y$	add	$(x+y)$	odloži na sklad vsoto
$x\ y$	sub	$(x-y)$	odloži na sklad razliko
$x\ y$	mul	$(x \cdot y)$	odloži na sklad produkt
$x\ y$	lt	0 ali 1	1, če je $x < y$, sicer 0

Poleg lt obstajajo še naslednji ukazi, ki delujejo enako kot lt, le da uporabljajo druge primerjalne operatorje: gt ($>$), le ($\leq$), ge ($\geq$), eq ($=$) in ne ($\neq$).

Pogojni skok: jnz(*labela*)
Ta ukaz pobere element z vrha sklada, recimo mu x; nato pa, če je $x \neq 0$, se izvajanje programa nadaljuje z ukazom za labelo *labela*, sicer pa se izvajanje nadaljuje pri naslednjem ukazu (tako kot običajno).

Označevanje: na sklad je mogoče odložiti tudi posebno vrednost, ki ji pravimo *značka*. Na skladu je lahko v posameznem trenutku nič ali več značk. Za delo z značkami so na voljo naslednji trije ukazi (v spodnji tabeli je značka predstavljena s simbolom $\circledast$):

operandi	ukaz	rezultati	opis
	mark	$\circledast$	odloži na sklad značko
$\circledast\ x_1\ \dots\ x_n$	counttomark	$\circledast\ x_1\ \dots\ x_n\ n$	prešteje elemente nad značko
$\circledast\ x_1\ \dots\ x_n$	cleartomark		pobriše značko in elemente nad njo

Ukaz mark torej odloži novo značko na vrh sklada; counttomark prešteje, koliko elementov je nad najvišjo značko v skladu; cleartomark pa pobriše najvišjo značko in vse elemente, ki so bili nad njo. Če poskušamo izvesti kakšnega od zadnjih dveh ukazov, ko na skladu ni nobene značke, se program sesuje.

Napiši v tem jeziku **program**, ki obrne vrstni red zgornjih n elementov sklada. Torej, če so ob začetku izvajanja na vrhu sklada števila

$$x_1\ x_2\ \ldots\ x_{n-1}\ x_n\ n$$

(pri čemer je vrh sklada na desni, prav na vrhu je torej število n), naj bodo ob koncu izvajanja na vrhu sklada elementi

$$x_n\ x_{n-1}\ \ldots\ x_2\ x_1$$

Pri tem naj se preostala vsebina sklada (torej tisto, kar je bilo prej pod x_1, zdaj pa je pod x_n), nič ne spremeni. Predpostaviš lahko, da so elementi $x_1, \ldots, x_n$, ki jih dobiš na vrhu sklada ob začetku izvajanja svojega programa, cela števila (ne pa značke).

Ocenjevanje: tvoj program bomo preizkusili na desetih testnih primerih z različnimi vrednostmi n (za $1 \leq n \leq 100$). Pri petih od desetih testnih primerov bo veljalo $n \leq 10$. Na posameznem testnem primeru dobi tvoj program 10 točk, če pravilno obrne podatke na skladu in pri tem nikoli ne izvede ukaza mark; če podatke na skladu pravilno obrne in pri tem kdaj izvede tudi ukaz mark, dobi pri tem testnem primeru 7 točk; če pa podatkov ne obrne pravilno (ali pa je sintaktično napačen ali kaj podobnega), ne dobi pri tem testnem primeru nobene točke. Na posameznem testnem primeru sme program izvesti največ 10000 korakov, sicer pri njem ne dobi nobene točke.

Pri tej nalogi sme biti izvorna koda, ki jo oddajaš, dolga največ 10000 bytov (vključno z znaki za konec vrstice), napisana pa mora biti v tu opisanem programskem jeziku (in ne na primer v C-ju, C++u, pascalu ipd.).

Pri reševanju naloge si lahko pomagaš z interpreterjem tu uporabljenega programskega jezika, ki si ga lahko preneseš z ocenjevalnega strežnika.

Primer: spodaj je primer programa, ki rešuje malo drugačen problem — če so ob začetku izvajanja na vrhu sklada števila

$$a_1\ a_2\ \ldots\ a_{n-1}\ a_n\ n$$

jih program zamenja z vsoto $(a_1 + a_2 + \ldots + a_n)$. V komentarjih je prikazano, kakšna je vsebina sklada po izvedbi ukaza ali ukazov v trenutni vrstici. Ukazi med labelama zacetek in konec se bodo izvajali v zanki in komentarji v tistih vrsticah kažejo, kakšna bi bila vsebina sklada v tisti iteraciji, pri kateri nam ostane za seštevanje še k števil (števila od a_{k+1} do a_n pa smo že sešteli).

```
                        % a1 a2 ... a(n-1) an n
  push(0)               % a1 a2 ... a(n-1) an n 0

zacetek:
                        % a1 a2 ... a(k-1) ak k (a(k+1) + ... + an)
  exch                  % a1 a2 ... a(k-1) ak (a(k+1) + ... + an) k
  dup push(0)           % a1 a2 ... a(k-1) ak (a(k+1) + ... + an) k k 0
  % Če je k = 0, končajmo.
  le
  jnz(konec)
                        % a1 a2 ... a(k-1) ak (a(k+1) + ... + an) k
```

```
   push(1) sub            % a_1 a_2 ... a_{k-1} a_k (a_{k+1} + ... + a_n) (k - 1)
   push(3) push(1)        % a_1 a_2 ... a_{k-1} a_k (a_{k+1} + ... + a_n) (k - 1) 3 1
   roll                   % a_1 a_2 ... a_{k-1} (k - 1) a_k (a_{k+1} + ... + a_n)
   add                    % a_1 a_2 ... a_{k-1} (k - 1) a_k (a_k + ... + a_n)
   push(1)
   jnz(zacetek)

konec:
                          % (a_1 + ... + a_n) 0
   pop                    % (a_1 + ... + a_n)
```

NALOGE ZA ŠOLSKO TEKMOVANJE

6. februarja 2009

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). Pri vsaki nalogi lahko dobiš od 0 do 20 točk.

Rešitve bodo objavljene na `http://rtk.ijs.si/`.

1. Vremenoslovje

Na stotih meteoroloških postajah se odčitavajo temperature zraka. V datoteki so zbrane tako dobljene meritve za določeno obdobje. **Napiši program**, ki bo prebral datoteko (recimo, da ji je ime `meritve.txt`) in izpisal najvišje temperature za vsako meteorološko postajo. V vsaki vrstici datoteke so tri cela števila: najprej zaporedna številka dneva, v katerem je bila meritev dobljena; nato zaporedna številka postaje (od 1 do 100); in nato temperatura (merjena v stopinjah Celzija).

Če za neko postajo ni v vhodni datoteki nobenega podatka o temperaturi, zanjo namesto najvišje temperature izpiši niz „`ni podatka`“.

2. Krajšanje imena

Mobilni telefon hrani telefonski imenik z imeni in priimki. Ker so nekatera imena in priimki zelo dolgi, včasih na zaslonu ni dovolj prostora, da bi izpisali polno ime in priimek. V tem primeru se mora prikazati smiselna okrajšava. Možni izpisi imena „Miha Novak“ so (od najbolj do najmanj zaželenih):

1. Miha Novak
2. M. Novak
3. Novak
4. M.N.
5. ⟨prazno⟩

Napiši podprogram, ki sprejme ime, priimek in število razpoložljivih znakov n ter izpiše po potrebi okrajšano ime in priimek. Izpis ne sme prekoračiti n znakov. Predpostaviš lahko, da sta tako ime kot priimek neprazna niza.

3. Registrske številke

V nekem seznamu so po vrsticah zapisane registrske številke (brez vezajev in presledkov) avtomobilov in imena njihovih lastnikov; med registrsko številko in imenom je natanko en presledek. Vsaka registrska številka je dolga natanko sedem znakov, cela vrstica z imenom in priimkom vred pa največ sto znakov. Na primer:

```
LJV502E Miha Novak
MBA60R2 Maja Kovač
LJV511E Ana Kogovšek
```

Delna registrska številka je registrska številka, v kateri so nekateri znaki neznani in so predstavljeni z vprašajem, na primer:

```
LJV5??E
```

Napiši program, ki za dano delno registrsko številko izpiše *imena* vseh tistih lastnikov avtomobilov, ki bi lahko imeli tako številko. Na primer, za zgornjo delno številko in datoteko bi program izpisal:

```
Miha Novak
Ana Kogovšek
```

Tvoj program naj podatke bere s standardnega vhoda ali pa iz datoteke z imenom `stevilke.txt` (kar ti je lažje); v prvi vrstici dobi neko delno registrsko številko, v preostalih vrsticah pa dejanske registrske številke z imeni lastnikov.

4. Največji nihaj navzdol

Ena od stvari, ki nas včasih zanimajo, ko analiziramo gibanje kakšnega vrednostnega papirja (na primer delnice, obveznice ali točke vzajemnega sklada), je tudi *največji nihaj navzdol* (*maximum drawdown*). Recimo, da v nekem trenutku kupimo delnico po ceni c_1 in jo v nekem kasnejšem trenutku prodamo po ceni c_2. Če je prodajna cena nižja od nakupne, smo ustvarili izgubo, in sicer $100 \cdot (1 - c_2/c_1)$ odstotkov. (Pri tej nalogi predpostavimo, da davkov in provizij pri nakupu in prodaji ni.)

Največji nihaj navzdol v določenem obdobju je definiran kot največja izguba, ki jo lahko na ta način naredimo, če kupimo in prodamo v najbolj neugodnih trenutkih.

Če opazujemo nek vrednostni papir v n zaporednih trenutkih in je c_k njegova cena v k-tem izmed teh trenutkov, potem je največji nihaj navzdol največja izmed vrednosti $100 \cdot (1 - c_j/c_i)$ po vseh i in j, za katere je $1 \leq i < j \leq n$. (Če cena ves čas opazovanja narašča, se lahko zgodi, da bo največji nihaj navzdol negativen — to pač pomeni, da v tistem obdobju s tem vrednostnim papirjem ne moremo ustvariti izgube.)

Napiši podprogram NajvecjiNihajNavzdol, ki računa največji nihaj navzdol. Pri tem lahko predpostaviš, da že obstajata naslednja podprograma, ki ti dajeta podatke o opazovanem vrednostnem papirju:

- **function** StTrenutkov: integer; { *v pascalu* }
  int StTrenutkov(); /* *v C/C++* */

  Vrne n, torej število trenutkov, v katerih opazujemo ceno vrednostnega papirja. Predpostaviš lahko, da je n večji od 1.

- **function** Cena(k: integer): real; { *v pascalu* }
  double Cena(**int** k); /* *v C/C++* */

  Vrne c_k, torej ceno opazovanega vrednostnega papirja v trenutku k. Pri tem mora biti k večji ali enak 1 ter manjši ali enak n. Predpostaviš lahko, da so vse cene večje od 0.

Tvoj podprogram pa naj bo takšne oblike:

function NajvecjiNihajNavzdol: real; { *v pascalu* }
double NajvecjiNihajNavzdol(); /* *v C/C++* */

Vrne naj največji nihaj navzdol, izražen v odstotkih.

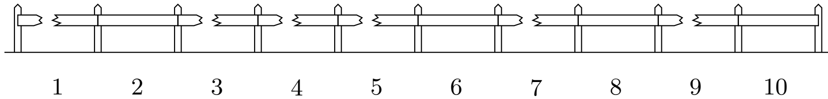
Tvoja rešitev naj bo čim bolj učinkovita (če se le da, naj deluje hitro tudi pri velikih n, npr. $n = 1\,000\,000$ in več).

5. Popravilo ograje

Odkar se je na Kranjskem razcvetel turizem in imajo zato veliko imenitnih obiskovalcev iz tujine, Kranjci veliko bolj skrbijo za samopodobo. Obnovili so že vsa pročelja hiš, sedaj pa so se lotili še cest. Med drugim bodo zamenjali zaščitno ograjo. Ker so nekoliko ekonomični, želijo zamenjati samo poškodovane dele ograje.

Ograja je sestavljena iz deščic dolžine 1 m, ki so druga za drugo pritrjene na stebričke ob cesti. Te deščice tvorijo eno dolgo neprekinjeno ograjo, zato so jih označili s števili od 1 do n. (k-ta deščica se tako na levem koncu stika s $(k-1)$-vo deščico, na desnem pa s $(k+1)$-vo deščico. Izjemi sta le prva in zadnja deščica, ki imata samo eno sosedo.)

Primer ograje z $n = 10$ deščicami, od katerih je šest poškodovanih:



Ograjo so natančno pregledali in naredili seznam oznak poškodovanih deščic. Ko so hoteli naročiti nove deščice, s katerimi bi nadomestili poškodovane, so ugotovili, da so pri dobavitelju na voljo le deske dolžine k metrov ($k > 1$). Te sicer lahko tudi razrežejo, vendar bo od obeh kosov uporaben samo eden — desko lahko skrajšajo, ne morejo je pa razdeliti na dve ali več. Ograjo bodo menjali tako, da bodo iz ograje odstranili l ($l \leq k$) zaporednih metrskih deščic (med njimi so lahko tudi nepoškodovane, ki bodo pri odstranjevanju uničene) in jih nadomestili z novo desko (ki jo bodo po potrebi skrajšali na l metrov). To bodo izvedli na enem ali več mestih, dokler ne bodo zamenjane vse poškodovane deščice. Zanima jih najmanjše število desk, ki jih morajo kupiti, da bodo obnovili celo ograjo.

Opiši postopek, ki za dano dolžino ograje n, dolžino novih desk k in dani seznam poškodovanih deščic (predpostaviš lahko, da so številke poškodovanih deščic naštete v naraščajočem vrstnem redu) ugotovi, kolikšno je najmanjše število novih (k-metrskih) desk, ki jih morajo kupiti, da bodo lahko popravili vse poškodovane dele ograje. Zaželeno je, da je tvoj postopek čim bolj učinkovit, da bo deloval hitro tudi pri velikih n in k. V svojem odgovoru tudi dobro utemelji, zakaj tvoj predlagani postopek daje pravilne rezultate.

Primer: recimo, da imamo ograjo z gornje slike, z $n = 10$ deščicami, od katerih so poškodovane deščice 1, 3, 4, 5, 7 in 9; in recimo še, da je $k = 4$ (torej lahko kupimo le štirimetrske deske). V tem primeru potrebujemo za popravilo ograje najmanj tri nove deske: prvo desko lahko na primer skrajšamo na dolžino 1 in z njo zamenjamo deščico 1; drugo desko skrajšamo na dolžino 3 in z njo zamenjamo deščice 3, 4 in 5; tretjo desko pa skrajšamo na dolžino 3 in z njo zamenjamo deščice 7, 8 in 9 — deščica 8 sicer ni poškodovana, vendar ni nič narobe, če jo vseeno zamenjamo. Le z dvema deskama dolžine $k = 4$ pa se ograje ne da obnoviti, čeprav je poškodovanih samo 6 deščic na ograji.

NEUPORABLJENE NALOGE IZ LETA 2007

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 2. tekmovanjem IJS v znanju računalništva (leta 2007), 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. 76–105) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Nakupovanje

Gospodinja Mici je zelo varčna, zato se pri nakupovanju vedno napoti v trgovino, v kateri bo za svoj nakup plačala najmanj. Pred nakupom si tako iz katalogov izpiše cene izdelkov v posameznih trgovinah, ki jih namerava kupiti, in se glede na nakupovalni seznam določi, v katero trgovino se bo napotila. Vedno pa gre le v eno trgovino, saj tudi njen čas nekaj stane, gorivo za avtomobil pa tudi ni zastonj.

Napiši podprogram NajcenejsiNakup, ki bo Mici izbral trgovino. Tvoj podprogram naj predpostavi, da obstajajo naslednje globalne konstante in spremenljivke s podatki o trgovinah in izdelkih:

- **const** M = ...; — število vseh trgovin.
- **const** N = ...; — število vseh izdelkov.
- **var** Cena: **array** [1..N, 1..M] **of** real; — tabela cen: element Cena[i, j] pove, koliko stane izdelek i v trgovini j.
- **var** Kolicine: **array** [1..N] **of** integer; — nakupovalni seznam: element Kolicine[i] pove, koliko izdelkov i mora gospodinja nakupiti.

Tvoj podprogram naj bo oblike:

```
function NajcenejsiNakup: integer;
```

Vrne naj številko trgovine, kjer je nakup najcenejši.

Še deklaracije v C-ju:

```
#define M ...
#define N ...
double Cene[N][M];
int Kolicine[N];
int NajcenejsiNakup();    /* Podprogram, ki ga moraš napisati ti. */
```

2. Metulji

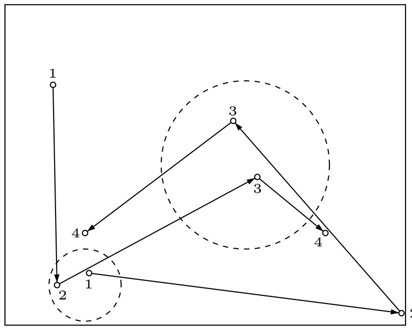
Jožek opazuje dva metulja, ki letata po vrtu pod njegovim oknom. Ker je po naravi pedantnež, se opazovanja loti sistematično: vsako minuto pogleda na vrt in si za vsakega od metuljev zapiše koordinate, na katerih se je nahajal v času opazovanja. Jožek naredi n opazovanj, pri čemer je $n < 1000$. Na koncu bi rad na podlagi svojih

meritev in nekaterih dodatnih podatkov o vrtu izvedel, pri koliko opazovanjih sta metulja sedela na isti roži.

Vrt si pri tem lahko predstavljamo kot pravokotnik širine w in višine h, rože pa kot kroge različnih polmerov znotraj tega pravokotnika. Predpostaviš lahko, da se ti krogi med seboj ne prekrivajo. Znano je število rož r, $r < 1000$, za vsako rožo pa še polmer in koordinati njenega središča. Metulja obravnavamo kot točki. Dimenzije in koordinate niso nujno cela števila.

Napiši program, ki bo s standardnega vhoda prebral rezultate opazovanja metuljev ter informacije o vrtu in na standardni izhod izpisal število, ki zanima Jožka. Za vsakega od metuljev štejemo, da sedi na neki roži, če se v danem trenutku nahaja znotraj kroga, ki ponazarja to rožo.

Namig: razdaljo med točkama A in B s koordinatami (x_A, y_A) in (x_B, y_B) dobiš po pitagorovem izreku: $\sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$.



Primer vrta z $n = 4$, $r = 2$. Črtkani krožnici predstavljata roži, puščice pa označujejo pot metuljev. Vidimo lahko, da sta bila metulja hkrati na isti roži le ob času 3.

Vhodna datoteka: v prvi vrstici sta celi števili w (širina vrta) in h (širina vrta). V drugi vrstici sta celi števili r (število rož) in n (število opazovanj). V naslednjih r vrsticah so podatki o rožah; vsaka vsebuje x-koordinato, y-koordinato in radij posamezne rože. V naslednjih n vrsticah so podatki o položaju metuljev v posameznem trenutku; v vsaki vrstici so x-koordinata prvega metulja, y-koordinata prvega metulja, x-koordinata drugega metulja in nato še y-koordinata drugega metulja. Števila znotraj posameznih vrstic so ločena s presledkom.

Izhodna datoteka: vanjo izpiši število opazovanj, pri katerih sta oba metulja sedela na isti roži.

Primer vhodne datoteke (ki ustreza gornji sliki):

```
10 8
2 4
2.0 1.0 0.9
6.0 4.0 2.1
1.2 6.0 2.1 1.3
1.3 1.0 9.9 0.3
6.3 3.7 5.7 5.1
8.0 2.3 2.0 2.3
```

Pripadajoča izhodna datoteka:

```
1
```

3. Matura

Eden tvojih znancev je našel način, kako vdreti v strežnik Državnega izpitnega centra

(RIC) in priti do rezultatov mature. Sedaj pa bi rad malce premešal ocene, tako da bi najslabše ocenjeni dobil najboljšo oceno, drugi najslabše ocenjeni drugo najboljšo oceno itn., najboljši pa bi dobil najslabšo oceno. Verjame, da bi s tem dejanjem storil dobro delo. Pridni učenci bi tako morali na popravni rok in ga seveda naredili. Tisti, ki pa bi sicer imeli težave tako v prvem, kakor v drugem roku, pa bodo opravili že v prvo. Pri postopku za zamenjavo ocen pa potrebuje tvojo pomoč. Podatki bodo v naslednji obliki:

```
130043 67
130044 90
130041 25
120025 3
120134 35
120156 89
120277 34
```

Številka na levi strani je maturitetna šifra dijaka, številka na desni pa njegov rezultat (število točk na izpitu — to je celo število od 0 do 100). **Opiši postopek**, ki prebere tak seznam in izpiše nov seznam, v katerem bodo šifre dijakov v enakem vrstnem redu, rezultati pa premešani, kakor je zahtevano. To pa zato, da na RICu ne bodo (upajmo) opazili, kaj se je zgodilo.

Opomba: rezultate smeš le premešati, ne pa spreminjati. Če se torej v vhodni datoteki pojavlja npr. 23 dijakov z rezultatom 67, mora biti tudi v izhodni datoteki 23 dijakov s tem rezultatom — ni pa treba, da so to isti dijaki kot v vhodni datoteki.

4. Vojna

Vojna je enostavna in (vsaj med mlajšimi) popularna igra s kartami. Običajni kup kart (od 2 do 10, J, Q, K, A, včasih uporabimo tudi jokerje) enakomerno razdelimo med dva igralca (možna je tudi igra več igralcev, ki pa nas pri tej nalogi ne zanima). Igra poteka tako, da oba igralca potegneta *zgornjo* karto iz kupa. Igralec z močnejšo karto (moč kart narašča 2..10, J, Q, K, A) pobere obe karti in ju da na *dno* svojega kupa. Med igro igralca ne smeta spreminjati vrstnega reda kart v kupu (mešati kart).

Poseben primer nastopi, kadar sta obe karti enako močni. Tedaj pride do „vojne“. V tem primeru igralca potegneta še eno karto (spet z vrha kupa) in jo odložita zaprto (z licem navzdol) vsak na svojo igralno karto. Potem potegneta še tretjo karto (z vrha kupa) in jo odprto (z licem navzgor) odložita vsak na svojo zaprto karto. Igralec z višjo karto na vrhu igralnega kupa zmaga in pobere vseh šest kart (ter jih spravi na dno svojega kupa kart). Če pa sta tudi sedaj karti na vrhu igralnega kupa enaki, nastopi nova vojna in postopek se ponovi (vsak odloži eno karto zaprto in eno odprto, potem pa se zadnji potegnjeni karti primerjata).

Izgubi igralec, ki mu zmanjka kart.

Napiši funkcijski podprogram OdigrajVojno, ki bo odigral igro Vojna in vrnil zmagovalca.

```
int OdigrajVojno(KupKart *igralec1, KupKart *igralec2, int najvecPotez)
```

Podprogram naj odigra največ najvecPotez potez igre Vojna s kupoma kart igralec1 in igralec2. Če na kupu igralec1 zmanjka kart, naj podprogram vrne vrednost 2 (zmagal je drugi igralec). Če zmanjka kart na kupu igralec2, naj podprogram vrne 1 (zmagal je prvi igralec). Če po najvecPotez potezah noben kup kart ni prazen, podprogram vrne 0 (neodločeno), neodvisno od tega, koliko kart je na katerem kupu. (**Ugotovi**, ali je mogoč neodločen izid tudi v kakšnem drugem primeru.)

Uporabljaš lahko pomožna podprograma PoberiKarto in OdloziKarto (za katera lahko predpostaviš, da že obstajata in ju ne rabiš pisati sam):

- **bool** PoberiKarto(KupKart *kup, Karta *vrhnjaKarta);
  Če je kup kart prazen, funkcija vrne **false**, vrednost parametra vrhnjaKarta pa ni definirana. Če kup ni prazen, vzame vrhnjo karto iz kupa in jo vrne v parametru vrhnjaKarta. Vrednost funkcije je v tem primeru **true**, tudi če je novi kup (brez vrhnje karte) zdaj prazen.

- **void** OdloziKarto(KupKart *kup, Karta spodnjaKarta);
  Podprogram doda karto SpodnjaKarta na dno danega kupa kart.

Tip Karta opisuje eno karto. Definiran je kot enostaven številski tip:

```
typedef int Karta;
```

Karte od 2 do 10 so predstavljene s števili od 2 do 10. Karte J, Q, K in A so predstavljene s števili od 11 do 14. Višje število pomeni višjo (močnejšo) karto. Barve ignoriramo, ker v Vojni ne igrajo nobene vloge.

Pojasnila in nasveti:

- Vojna (ali več zaporednih vojn) šteje za eno samo potezo.

- Če igralcu zmanjka kart sredi vojne, igralec izgubi.

- Vseeno je, v kakšnem vrstem redu odlagaš igrane karte pod kup.

- Uporabiš lahko lokalne spremenljivke tipa KupKart. Predpostaviš lahko, da že sama deklaracija spremenljivke naredi prazen kup kart, tako da se ti ni treba ukvarjati z izdelavo kupa in njegovo inicializacijo.

Še deklaracije v pascalu:

```
type Karta = 2..14;
function OdigrajVojno(Igralec1, Igralec2: KupKart; NajvecPotez: integer): integer;
function PoberiKarto(var Kup: KupKart; var VrhnjaKarta: Karta): boolean;
procedure OdloziKarto(var Kup: KupKart; SpodnjaKarta: Karta);
```

5. Disk

Današnji diski so zaradi zanesljivosti branja in pisanja opremljeni s senzorji pospeškov. Z uporabo teh senzorjev ugotavljajo, če se je disk med branjem ali pisanjem prehitro pospeševal (npr. zaradi kakšnega tresljaja iz okolja), da bi bila še diskova bralno-pisalna glava še vedno na pravem mestu (na sredini sledi).

Napiši podprogram PreberiSektor za branje enega sektorja z diska, ki pri delovanju upošteva meritve s senzorjev in vrne prave podatke; podprogram mora, če

opazi, da se je disk med samim branjem podatkov z diska prehitro pospeševal, zavreči prebrane podatke in poskusiti prebrati podatke z istega sektorja znova. Podprogram naj tudi vrne vrednost, ki pove, ali je uspešno ali ni uspešno prebral sektorja:

```
function PreberiSektor(StevilkaSektorja: integer; var Medpomnilnik): boolean;
```

Na voljo imaš naslednje strojne podprograme:

- **procedure** ZacniBratiSektor(StevilkaSektorja: integer; **var** Medpomnilnik);
  Podprogram, ki ukaže diskovni elektroniki, naj prebere sektor z diska in ga zapiše v medpomnilnik.
- **function** Prebrano: integer;
  Funkcija, ki vrne 1, če je elektronika sektor že prebrala, sicer pa vrne 0.
- **function** Pospesek: integer;
  Funkcija vrne maksimalni pospešek v zadnji mikrosekundi.
- **procedure** Cakaj;
  Funkcija čaka eno mikrosekundo (predpostaviš lahko, da branje enega sektorja traja dlje kot eno mikrosekundo).

Pri pisanju upoštevaj,

- da disk ne more obdržati glave v sledi, če pospešek preseže N enot v katerikoli smeri
- da branje ne sme trajati preveč časa, zato odnehaj po M mikrosekundah.

Še deklaracije v C/C++:

```
void ZacniBratiSektor(int StevilkaSektorja, void* Medpomnilnik);
int Prebrano();
int Pospesek();
void Cakaj();
bool PreberiSektor(int StevilkaSektorja, void* Medpomnilnik);
```

6. Prepovedani znaki

Neko besedilo lahko vsebuje prepovedane znake. **Napiši podprogram**, ki v danem besedilu poišče prepovedane znake in jih nadomesti z nadomestnim znakom. Prepovedani znaki so podani v obliki znakovnega niza (torej na enak način kot besedilo). Tvoj podprogram naj ustreza naslednji deklaraciji:

```
procedure NadomestiPrepovedaneZnake(var Besedilo: string;
                                    PrepovedaniZnaki: string; NadomestniZnak: char);
```

ali, v C-ju:

```
void NadomestiPrepovedaneZnake(char* Besedilo,
                               const char* PrepovedaniZnaki, char NadomestniZnak);
```

Tvoja rešitev naj bo čim učinkovitejša.

Primer:

Besedilo	`http://www.google.com/search?hl=en&safe=off&q=rtk+ijs`
PrepovedaniZnaki	`\/*?"<>|`
NadomestniZnak	`_`
Besedilo po klicu podprograma	`http___www.google.com_search_hl=en&safe=off&q=rtk+ijs`

Namig: Pri najpreprostejši rešitvi poraba časa narašča sorazmerno s produktom dolžine obeh nizov (Besedilo in PrepovedaniZnaki). Mogoča pa je tudi učinkovitejša rešitev, pri kateri poraba časa narašča le sorazmerno z vsoto dolžin teh dveh nizov. Če ti ne uspe sestaviti takšne rešitve, poskusi napisati vsaj naivno rešitev ali pa kakšno tretjo, ki je po časovni zahtevnosti nekje vmes med njima.

Dodatno vprašanje: V nalogi nastopajo 8-bitni znaki (to je razvidno iz podatkovnih tipov v deklaraciji). Ali bi ravnal kako drugače, če bi lahko v nizih nastopali tudi poljubni 16-bitni znaki (npr. kot pri standardu Unicode)?

7. Gugl'bot

Spletni gomazač (*Web crawler*) je program, ki se samodejno sprehaja po spletnih straneh. Svoj sprehod začne na eni od strani iz podanega seznama „začetnih strani" ter sistematično sledi vsem povezavam, ki jih najde na strani, na kateri se trenutno nahaja. Njegova naloga je, da shrani vsako spletno stran na svoji poti v podatkovno bazo ali pa kar na trdi disk v obliki datoteke tipa HTML.

V podjetju Gugl so se odločili, da bodo izdelali takega gomazača. Spletne strani so se odločili shranjevati v določen direktorij v obliki datotek tipa HTML. Zadolžili so te, da napišeš podprogram, ki iz naslova spletne strani (tj. njenega URL-ja) tvori ime datoteke. Zaželeno je, da je ime datoteke kar enako naslovu spletne strani, vendar za poimenovanje datotek nekateri znaki, ki nastopajo v spletnih naslovih, niso dovoljeni — ime datoteke namreč ne sme vsebovati znakov `\`, `/`, `:`, `*`, `?`, `"`, `<`, `>` in `|`. V imenu datoteke take znake nadomesti s podčrtajem (`_`). Upoštevaj tudi, da je maksimalna dolžina imena datoteke omejena na 255 znakov. Če je potrebno ime krajšati, mu odreži konec. Nenazadnje upoštevaj tudi, da lahko datoteka z istim imenom v podanem direktoriju že obstaja. V takem primeru k imenu dodaj (na konec) še neko (naključno) številko. Tvoj podprogram naj ustreza naslednji deklaraciji:

```
function PoimenujDatoteko(URL, Direktorij: string): string;
```

ali, v C-ju:

```
char* PoimenujDatoteko(const char* URL, const char* Direktorij);
```

Predpostaviš lahko, da ti je že na voljo naslednji podprogram, ki preveri, ali datoteka z imenom ImeDatoteke že obstaja v direktoriju Direktorij:

```
function DatotekaObstaja(ImeDatoteke, Direktorij: string): boolean;
```

ali, v C-ju:

```
bool DatotekaObstaja(const char* ImeDatoteke, const char* Direktorij);
```

8. Klepetalnica

Program Klepetalnica omogoča dvema uporabnikoma, da si med seboj pošiljata kratka sporočila (dolga največ 100 znakov). Za to uporablja tri podprograme, in sicer:

- Poslji(Sporocilo) — sporočilo drugemu programu.
- Ko drugi program sprejme sporočilo, se pri njem kliče podprogram Sprejem(Sporocilo).
- Prikazi(Niz) — pokaže niz znakov uporabniku.

Trenutno je sporočilo definirano kot:

```
typedef struct {
   char vsebina[100];
} Sporocilo;
```

Ko uporabnik vnese sporočilo in pritisne gumb za pošiljanje, se izvede tale podprogram:

```
void GumbPritisnjen(const char *niz)
{
   Sporocilo sporocilo;
   strcpy(sporocilo.vsebina, niz);
   Poslji(sporocilo);
}
```

Podprogram Sprejem pa je takšen:

```
void Sprejem(Sporocilo sporocilo)
{
   Prikazi(sporocilo.niz);
}
```

Ker uporabnika ločuje internet, sporočila potrebujejo različno dolgo časa, da prispejo od pošiljatelja do prejemnika. Na primer, četudi je bilo sporočilo „Živjo“ poslano pred sporočilom „Adijo“, se lahko zgodi, da bosta sprejeti ravno v nasprotnem vrstnem redu.

Predlagaj popravek podprogramov Sprejem in GumbPritisnjen ter podatkovne strukture Sporocilo, da se bodo sporočila zagotovo prikazovala v takem vrstnem redu, v kakršnem so bila poslana. Pri tem lahko predpostaviš, da so vsa poslana sporočila tudi sprejeta. Da bo naloga lažja, se omejimo tukaj le na komunikacijo v eno smer: sporočila oddaja le en računalnik, sprejema in prikazuje pa jih drugi računalnik.

9. Komparatorji

To nalogo smo na tekmovanju sicer uporabili (2007.2.3), vendar je mogoče pri njej zastaviti še več zanimivih podnalog, za katere takrat žal ni bilo prostora, zato jih objavljamo tule. Naloga pravi na kratko takole: napiši podprogram Kljuc, ki bo takšne oblike:

```
function Kljuc(Z: string): string;     { v pascalu }
char* Kljuc(char* Z);                  /* v C-ju */
```

Ta podprogram mora vračati takšne ključe, da če neko zaporedje vhodnih nizov uredimo leksikografsko po njihovih ključih, bodo s tem tisti vhodni nizi prišli v nek zahtevani vrstni red. Spodnje podnaloge so samostojne in med seboj neodvisne — za vsako napiši po en podprogram:

(e) Vsak vhodni niz je sestavljen iz imena in priimka, ki sta ločena s presledkom. Uredili bi jih radi po leksikografskem vrstnem redu priimkov; če je v več nizih isti priimek, pa še po imenu. (Ta podnaloga je zelo lahka, če rečemo, da so imena in priimki sestavljeni le iz črk. Zanimivejšo in malo težjo nalogo pa dobimo, če rečemo, da dobimo ime in priimek posebej in da lahko v njiju nastopajo čisto poljubni znaki, tudi presledki.)

(f) Vsak vhodni niz predstavlja neko poljubno veliko celo število v desetiškem zapisu (na začetku je lahko predznak `+` ali `-`, sledi pa ena ali več števk od `0` do `9`). Tvoja funkcija `Kljuc` naj vrača take ključe, da bodo z urejanjem po njih naša cela števila urejena naraščajoče v običajnem vrstnem redu ($\ldots < -10 < -9 < \ldots < -1 < 0 < 1 < \ldots 9 < 10 < \ldots$).

(g) Vsak vhodni niz je sestavljen le iz malih črk angleške abecede. Nize bi radi uredili naraščajoče po številu `a`-jev, tiste z enakim številom `a`-jev bi uredili naraščajoče po številu `b`-jev in tako naprej.

(h) Vsak vhodni niz vsebuje dve celi števili, u in v, ločeni s presledkom in zapisani v desetiškem zapisu; zanju velja $0 \leq u < v$. Radi bi jih uredili po naraščajoči vrednosti u/v. (Za ulomke z enako vrednostjo je vseeno, v kakšen vrstni red jih postavi tvoj postopek — npr. ali postavi 2/3 pred 4/6 ali obratno.)

10. Smrkci

Smrkci bi radi svojo vasico obdali s kvadratnim obzidjem. Vsaka hiša v vasici je točkasta (koordinate teh točk so podane) in Smrkci iščejo najmanjši kvadrat, za katerega velja, da ležijo vse hiše v tem kvadratu (lahko tudi na robu kvadrata). Pri tem ni nujno, da bi bile stranice kvadrata vzporedne s koordinatnima osema. **Opiši postopek**, ki poišče najmanjši kvadrat, v katerem ležijo vse hiše.

11. Jezikovni tečaj

Nek organizator jezikovnih tečajev je ugotovil, da so stroški, ki jih ima z izvedbo tečaja, praktično konstantni ne glede na število udeležencev. Zato lahko ponudi svojim strankam tem nižjo ceno, čim več se jih prijavi na tečaj. Strankam je nižja cena seveda všeč, vendar pa si hkrati tudi ne želijo, da bi se na tečaj prijavilo preveč ljudi, saj ima potem vsak posamezni udeleženec manj priložnosti za izpopolnjevanje svojega znanja v pogovoru z učiteljem. Zato organizator ponuja možnost, da udeleženec ob prijavi navede največjo velikost skupine, v kateri je še pripravljen sodelovati. **Napiši program**, ki prebere podatke o prijavah in oblikuje skupino tečajnikov tako, da:

- bo skupina čim večja;
- ne bo v skupini nikogar, ki ni pripravljen sodelovati v tako veliki skupini;
- če je mogoče največjo možno skupino sestaviti na več načinov, naj da prednost tistim udeležencem, ki so se prijavili prej.

Vhodna datoteka. V prvi vrstici je celo število n ($1 \leq n \leq 10000$), ki predstavlja število prijavljenih. V naslednjih n vrsticah so navedene velikosti skupin, ki so jih stranke navedle v obrazcih, v takem vrstnem redu, kakor so se prijavljale. Tako je torej v drugi vrstici datoteke velikost skupine, ki jo je navedel prvi prijavljeni slušatelj, v tretji vrstici je velikost skupine, ki jo je navedel drugi prijavljeni slušatelj in tako naprej. Vsako od teh števil je med vključno 1 in vključno 10000.

Izhodna datoteka. V prvo vrstico izpiši število članov največje skupine, v vsako naslednjo pa še izbrane člane, ki so predstavljeni z zaporednimi številkami njihovih prijav (izpiši jih v naraščajočem vrstnem redu).

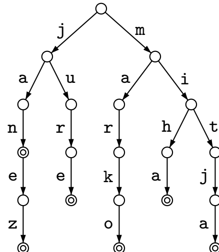
Primer vhodne datoteke:	Pripadajoča izhodna datoteka:	Še en primer vhodne datoteke:	Pripadajoča izhodna datoteka:
10	6	10	5
5	2	13	1
6	4	5	2
2	5	7	3
7	6	5	4
13	7	7	5
7	10	4	
11		5	
5		5	
3		2	
7		3	

12. Urejevalniška razdalja

Podjetje za programiranje d. o. o. bi rado razvilo program za preverjanje pravilnega črkovanja besed. Program mora vrniti vrednost `true`, če je beseda prisotna v slovarju vseh dovoljenih besed, in vrednost `false`, če beseda ni prisotna v tem slovarju. Ker mora program delovati hitro, se glavni programer podjetja odloči, da bo vse besede v slovarju interno predstavil z drevesom besed. Drevo besed je narobe obrnjeno drevo, ki ima po eno vejo za vsako besedo iz slovarja; črke na povezavah vzdolž te veje tvorijo ravno tisto besedo. Če imata dve ali več besed skupnih prvih nekaj črk, imajo tudi njihove veje skupna vozlišča. Nekatera vozlišča so še posebej označena, saj predstavljajo konce besed. Na sliki na str. 43 je primer drevesa, ki opisuje besede `miha`, `jure`, `mitja`, `janez`, `marko`, `jan`; vozlišča, v katerih se konča kakšna beseda, so označena z dvojnim krožcem.

Pri tej nalogi predpostavimo, da smo za dani slovar že zgradili takšno drevo. **Opiši** naslednje **postopke**, ki za dano vhodno besedo s pomočjo drevesa preverijo, ali je ta beseda (ali pa kakšna podobna) prisotna v slovarju:

(*a*) **Opiši postopek**, ki za dano vhodno besedo ugotovi, ali je ta beseda prisotna v slovarju ali ne.



(*b*) **Opiši postopek**, ki bo v primeru, da vhodne besede ni v slovarju, predlagal kakšno tako besedo iz slovarja, ki je enako dolga kot vhodna beseda in se od nje razlikuje le za en znak. (Če primerne besede v slovarju ni, naj postopek pač izpiše, da je ni.)

(*c*) **Opiši postopek**, ki bo v primeru, da vhodne besede ni v slovarju, predlagal kakšno tako besedo iz slovarja, ki se od vhodne besede razlikuje le za en znak: lahko je en znak zbrisan, vrinjen (na poljubno mesto — lahko tudi na začetek ali konec besede) ali pa je eden od obstoječih znakov spremenjen v kakšen drug znak.

Primer: pri slovarju iz gornje slike in pri vhodni besedi `mija` bi postopek (*a*) ugotovil, da je ni v slovarju; postopek (*b*) bi predlagal besedo `miha`; postopek (*c*) pa bi lahko predlagal katerokoli od besed `miha` in `mitja`.

13. Društva

V nekem kraju deluje več društev. Vsakdo se lahko brez kakršnih koli omejitev včlani v nič, eno ali več društev; vsako društvo ima lahko nič, enega ali več članov. Med drugim obstaja tudi Društvo za raziskovanje delovanja društev; njegovi člani (ki jih je tudi lahko nič, eden ali več) so preučevali povezanost članstva posameznih društev in svoja opažanja strnili v nekaj preprostih trditev. Vsaka trditev je v eni od naslednjih oblik:

1. Vsakdo, ki je član društva A, je tudi član društva B.

2. Društvi A in B nimata nobenega skupnega člana.

3. Ni res, da je vsakdo, ki je član društva A, tudi član društva B (z drugimi besedami, obstaja vsaj en človek, ki je član društva A, ne pa tudi društva B).

4. Društvi A in B imata vsaj enega skupnega člana.

Društva so označena s števili od 1 do n ($n \leq 10$). Opozorimo še na to, da če društvo A nima nobenega člana, potem prvi dve trditvi zagotovo držita in to ne glede na to, kakšno je društvo B.

Opiši postopek, ki prebere n in spisek trditev gornje oblike ter odgovori na naslednja vprašanja:

(a) Ali je sploh mogoče, da obstajajo taka društva, ki ustrezajo vsem navedenim trditvam? (Na primer: če imamo trditve „vsakdo, ki je član društva 1, je tudi član društva 2“, „vsakdo, ki je član društva 2, je tudi član društva 3“, „društvi 1 in 3 nimata nobenega skupnega člana“ in „društvi 1 in 4 imata vsaj enega skupnega člana“, se lahko hitro prepričamo, da takih štirih društev, ki bi ustrezala vsem tem trditvam, ni — tisti, ki je trditve pisal, se je očitno zmotil ali pa je lagal.)

(b) Če je mogoče, da taka društva obstajajo, navedi konkreten primer, kakšno bi lahko bilo članstvo teh društev, da bi ustrezalo vsem navedenim trditvam. Prebivalci kraja so oštevilčeni s celimi števili od 1 do 2000, ti pa za vsako društvo (od 1 do n) navedi, kateri prebivalci (če sploh kateri) so v tem tvojem hipotetičnem primeru njegovi člani.

(c) Recimo, da je odgovor na vprašanje iz točke (a) pritrdilen, torej da je mogoče, da primerna društva res obstajajo. Tvoj postopek naj zdaj odgovori še na nekaj dodatnih vprašanj takšne oblike:

- Ali je vsakdo, ki je član društva A, tudi član društva B?
- Ali imata društvi A in B kakšnega skupnega člana?

Pri vsakem vprašanju naj tvoj postopek izbere pravilnega od naslednjih treh možnih odgovorov: to zagotovo drži; to zagotovo ne drži; iz trditev, ki smo jih prebrali na začetku, ni mogoče niti zagotovo sklepati, da to drži, niti zagotovo sklepati, da to ne drži.

14. Elektrika

Prodajalci električne energije le-to ponujajo v paketih, določenih z minimalno in maksimalno količino ter ceno za enoto. Primer takšne ponudbe je 3–7 GWh po 30 €/GWh: lahko kupimo 3 GWh za 90 € ali pa 4 GWh za 120 € ali pa 5 GWh za 150 €, 6 GWh za 180 € lahko pa 7 GWh za 210 €, drugih količin pa iz te ponudbe ne moremo kupiti. Količina elektrike, ki jo kupimo iz neke ponudbe, je torej vedno celo število megavatnih ur in ne sme biti manjše od minimalne ali večje od maksimalne količine pri tej ponudbi (razen če se odločimo, da iz te ponudbe ne kupimo ničesar). Če je ponudb več, lahko nekatere količine sestavimo na različne načine, npr. 5 GWh lahko kupimo tako, da vzamemo 3 GWh iz te ponudbe in 2 GWh iz neke druge.

Napiši program, ki bo iz vhodne datoteke prebral ponudbe in želeno količino ter na izhodno datoteko izpisal način, kako najceneje kupiti točno zahtevano količino. Če obstaja več enakovrednih najcenejših načinov, naj izpiše kateregakoli izmed njih. Predpostaviš lahko, da je ponudb največ 10000, zahtevana količina največ 1000 GWh in cena posamezne ponudbe največ 1000 €/GWh.

Vhodna datoteka v prvi vrstici vsebuje celi števili n in q, ki predstavljata število vseh ponudb in zahtevano količino v GWh. Sledi n vrstic, ki vsebujejo po tri cela števila, ki predstavljajo posamezno ponudbo (minimalna količina v GWh, maksimalna količina v GWh, cena v € za 1 GWh). *Izhodna datoteka* naj vsebuje n vrstic, v vsaki pa po eno celo število — i-ta vrstica predstavlja količino, ki jo kupimo iz i-te ponudbe.

Primer vhodne datoteke:

```
4 10
4 10 40
5 9 5
6 8 20
11 11 10
```

Pripadajoča izhodna datoteka:

```
4
6
0
0
```

Opomba: cena rešitve je $4 \cdot 40 + 6 \cdot 5 = 190$ €. Čeprav bi lahko iz zadnje ponudbe ceneje (za 110 €) kupili 11 GWh, takšna rešitev (0 0 0 11) ni veljavna, ker naloga zahteva točno količino.

15. Bakterije

Mikrobiolog Polde raziskuje novo vrsto bakterij samomorilk. Na začetku ima v svoji petrijevki natanko a bakterij. Te se počasi delijo in ob vsaki delitvi nastaneta iz vsake bakterije dve; tidve se sčasoma tudi sami razdelita in tako naprej. Po prvi delitvi ima torej Polde $2 \cdot a$ bakterij, po drugi delitvi že $4 \cdot a$ bakterij in tako naprej.

Po n delitvah začne Polde obsevati bakterije z ultravijoličnimi žarki. Pod njihovim vplivom se bakterije prenehajo deliti, poleg tega pa se združijo v skupine po k bakterij, da bi se tako bolje branile pred UV žarki (točna vrednost k-ja je odvisna od tega, s katero vrsto bakterij imamo opravka). Ker pa število bakterij ni nujno deljivo s k, se lahko zgodi, da nekaj bakterij pri tem ostane osamljenih. Pomagaj Poldetu in mu **napiši program**, ki za dane vrednosti a, n in k izračuna, koliko bakterij na koncu ostane osamljenih.

Vhodna datoteka: vsebuje eno samo vrstico, v kateri so po vrsti navedena števila a, n in k, ločena s po enim presledkom. Vsa tri so cela števila, zanje pa velja: $1 \leq a \leq 10\,000$, $1 \leq n \leq 1\,000\,000\,000$ in $1 \leq k \leq 181$.

Izhodna datoteka: vanjo vpiši eno samo celo število, in sicer število osamljenih bakterij na koncu poskusa, kakršnega opisuje besedilo naloge.

Primer vhodne datoteke:

```
3 10 5
```

Pripadajoča izhodna datoteka:

```
2
```

Pomembno opozorilo: pri pripravi te naloge nismo trpinčili ali kako drugače pohabili nobene bakterije.

REŠITVE NALOG ZA PRVO SKUPINO

1. Kolikokrat najmanjši

V spremenljivki najmanjse hranimo najmanjše doslej prebrano število, v spremenljivki kolikokrat pa dosedanje število pojavitev števila najmanjse. Vrednost kolikokrat = 0 pa označuje, da nismo prebrali še ničesar. Po vsakem branju novega števila s standardnega vhoda (v spremenljivko stevilo) ga primerjamo z doslej najmanjšim; če je novo število še manjše (ali pa je to sploh prvo prebrano število, kar vidimo iz kolikokrat = 0), si ga zapomnimo v najmanjse in postavimo števec pojavitev na 1. Če je novo število enako dosedanjemu najmanjšemu, pa le povečamo števec pojavitev (kolikorat) za 1.

```
#include <stdio.h>
int main() {
  int kolikokrat = 0, najmanjse, stevilo;
  while (1 == scanf("%d", &stevilo))
    if (kolikokrat == 0 || stevilo < najmanjse) najmanjse = stevilo, kolikokrat = 1;
    else if (stevilo == najmanjse) kolikokrat++;
  printf("%d\n", kolikokrat); return 0;
}
```

Eden od tekmovalcev si je v anketi zaželel tudi rešitev v pietu.[3] Njemu in drugim ljubiteljem modernizma bomo poskusili ustreči s programom na str. 47. Ker v pietu nimamo na voljo posebnega ukaza za preverjanje, ali smo že na koncu standardnega vhoda, smo se oprli na omembo v opisu jezika, ki pravi, da se operacije, ki jih ni mogoče izvesti, preprosto ignorira. Naš program se torej zanaša na to, da inštrukcija za branje vhodnega števila na sklad ne odloži ničesar, če smo pri branju že prišli do konca standardnega vhoda.

2. Označevanje kovancev

Nalogo lahko rešimo na več načinov. Preprosta rešitev je, da s petimi gnezdenimi zankami naštevamo vse možne petmestne oznake in jih izpisujemo. Sproti tudi štejmo, koliko smo jih že izpisali (spremenljivka stIzpisanih), tako da bomo lahko nehali, čim izpišemo dvajset milijonov oznak (to hranimo v konstanti m).

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

int main()
{
  const char *znaki = "0123456789ABCDEFGHJKMNPRTVWXYZ";
  const int m = 20000000, k = strlen(znaki);
  int i1, i2, i3, i4, i5, stIzpisanih = 0;
  for (i1 = 0; i1 < k && stIzpisanih < m; i1++)
  for (i2 = 0; i2 < k && stIzpisanih < m; i2++)
```

[3] Opis tega ezoteričnega programskega jezika in povezave do interpreterjev najdemo na naslovu http://www.dangermouse.net/esoteric/piet.html. V tem jeziku je program pravzaprav dvodimenzionalna slika z omejenim naborom barv, interpreter pa se po določenih pravilih premika po njej in na prehodih iz ene barve v drugo izvaja inštrukcije. Vsi podatki se hranijo na skladu, ukazi za delo z njim pa spominjajo na tiste iz jezika postscript.

```
L0 ·· L0 ## ·· ·· ·· ·· ··
·· ·· ·0 ·· ·· ·· ·· ·· ··
·· ·· ·4 ·· ·· ·· ·· ·· ··
·· ·· ·2 ·· ·· ·· ·· ·· ··
·· ·· ·3 ·· ·· ·· ·· ·· ··
·· ·· D4 ·· ·· ·· ·· ·· ··
·· ·· L4 ·· ·· ·· ·· ·· ··
·· ·· L2 ·· ·· ·· ·· ·· ··
·· ·· ·3 ·· ·· ·· ·· ·· ··
·· ·· ·1 ·· ·· ·· ·· ·· ··
·· ## ·· ·· ·· ·· ·· ·· ··
·· ·· ·· ## ·· ·· ·· ·· ··
·· ·· L0 ·· ·· ·· ·· ·· ··
·· ·· D4 D4 D4 ·· ·· ·· ··
·· ·· L4 L4 ## ·· ·· ·· ··
·· ·· ·4 ·· ·· ·· ·· ·· ··
·· ·· D2 ·· ·· ·· ·· ·· ··
·· ·· D0 ·· ·· ·· ·· ·· ··
·· ·· L0 ·· ·· ·· ·· ·· ··
·· ·· L4 ·· ·· ·· ·· ·· ··
·· ·· ·5 ·5 ·· ·· ·· ·· ··
·· ·· D5 ·· ·· ·· ·· ·· ··
·· ·· L5 ·· ·· ·· ·· ·· ··
·· ·· ·3 ·· ·· ·· ·· ·· ··
·· ·· ·0 ·· ·· ·· ·· ·· ··
·· ·· L3 L3 L3 L3 L3 ·· ··
·· ·· ·· ·· ·· ·· D3 ·· ··
·· ·· L0 L0 L0 ·· ·3 ·· ··
·· ·· L4 ·· ·· ·· D2 ·· ··
·· ·· ·4 ·· ## ## ·· ## ··
·· ·· ·2 ·· ## ·· ·· ## ··
·· ·· D3 ·· ## ## ## ## ··
·· ·· D0 ·· ·· ·· ·· ·· ··
·· ·· ·3 ·3 ·3 ·3 ·3 ·· ··
·· ·· ·· ·· ·· ·· ·· ·· ··
·· ·· L0 ·· ·· ·· L0 L0 L0
·· ·· D0 D0 ·· ·· ·0 ·· ##
·· ·· L0 ·· ·· ·· D0 ·· ··
·· ·· ·0 ·· ·· ·· L4 ·· ··
·· ·· D4 ·· ·· ·· L2 L2 L2
·· ·· ·4 ·· ·· ·· ·2 ·2 ··
·· ·· D4 D4 ·· ·· D2 ·· ··
·· ·· L4 ·· ·· ·· L0 ·· ··
·· ·· ·4 ·· ·· ·· L4 L4 L4
## ·· D2 ·· ·· ·· ·4 ·· ··
·· ·· ## ·· ·· ·· D4 ·· ··
·· ·· L2 L2 L2 L2 L2 ·· ··
·· ·· L5 ·· ·· ·· ·· ·· ··
·· ·· D2 D2 D2 D2 D2 ·· ··
·· ·· ·· ·· ·· ·· ·· ·· ··
·· ·· L0 ·· ·· ·· L0 L0 ··
·· ·· L4 L4 L4 ·· ·0 ## ··
·· ·· ·4 ·4 ·· ·· D0 ·· ··
·· ·· D4 ·· ·· ·· L4 ·· ··
## ·· ·· ·· ·· ·· D4 ·· ··
·· ·· L0 ·· ·· ·· ## ·· ··
·· ·· ·4 ·· ·· ·· ·· ·· ··
·· ·· ·2 ·2 ·2 ·· ·· ·· ··
·· ·· D2 ·· ·· ·· ·· ·· ··
·· ·· L2 ·· ·· ·· ·· ·· ··
·· ·· ·0 ·· ·· ·· ·· ·· ··
·· ·· ·3 ·· ·· ·· ·· ·· ··
·· ·· L0 L0 L0 L0 L0 ·· ··
·· ·· ·· ·· ·· ·· ·· ·· ··
·· ·· L0 ·· ·· ·· L0 L0 L0
·· ·· D0 D0 ·· ·· ·0 ·· ##
·· ·· L0 ·· ·· ·· D0 ·· ··
·· ·· ·0 ·· ·· ·· L4 ·· ··
·· ·· D4 ·· ·· ·· D4 ·· ··
·· ·· L4 ·· ·· ·· ·4 ·· ··
·· ·· L5 L5 ·· ·· D4 D4 ··
·· ·· ·5 ·· ·· ·· L4 ·· ··
·· ·· D5 ·· ·· ·· ·4 ·· ··
·· ·· L3 ·· ·· ·· D2 ·· ··
## ·· ·· ·· ·· D2 D2 ·· ··
·· ·· ## ·· ·· ## ## ·· ··
```

Primer rešitve v pietu
za prvo nalogo prve skupine.

Ker so programi v pietu pravzaprav barvne slike, tiskana verzija našega biltena pa je črno-bela, je program tule prikazan tudi v tekstovni obliki. Vsak piksel je predstavljen z dvema znakoma, pri čemer dve piki pomenita belo barvo, `##` črno, pri ostalih barvah pa prvi znak pomeni svetlost (`L` = svetla, pika = običajna, `D` = temna), drugi znak pa barvni odtenek (od 0 do 5).

```
   for (i3 = 0; i3 < k && stIzpisanih < m; i3++)
   for (i4 = 0; i4 < k && stIzpisanih < m; i4++)
   for (i5 = 0; i5 < k && stIzpisanih < m; i5++, stIzpisanih++)
      printf("%c%c%c%c%c\n", znaki[i1], znaki[i2], znaki[i3], znaki[i4], znaki[i5]);
   return 0;
}
```

Elegantna možnost je tudi, da pogoja stIzpisanih < m v zankah ne preverjamo, pač pa najbolj notranjo zanko spremenimo takole:

```
   for (i5 = 0; i5 < k; i5++) {
      printf("%c%c%c%c%c\n", znaki[i1], znaki[i2], znaki[i3], znaki[i4], znaki[i5]);
      if (++stIzpisanih >= m) return 0; }
```

Namesto gnezdenih zank lahko uporabimo tudi rekurzijo, kar je sicer mogoče malo počasneje, vendar bi bilo takšno rešitev lažje posplošiti na drugačno dolžino oznak (namesto petmestnih):

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

#define dolzina 5
const char *znaki = "0123456789ABCDEFGHJKMNPRTVWXYZ";
int m = 20000000, k, stIzpisanih;
char oznaka[dolzina + 1];

void Izpisuj(int globina)
{
   int i;
   for (i = 0; i < k && stIzpisanih < m; i++) {
      oznaka[globina] = znaki[i];
      if (globina + 1 == dolzina) {
         printf("%s\n", oznaka); stIzpisanih++; }
      else Izpisuj(globina + 1); }
}

int main()
{
   k = strlen(znaki);
   oznaka[dolzina] = 0; stIzpisanih = 0;
   Izpisuj(0); return 0;
}
```

Lahko pa si naše petmestne oznake predstavljamo kot petmestna cela števila v tridesetiškem sestavu; imejmo torej zunanjo zanko z dvajset milijoni iteracij, v vsaki iteraciji pa trenutni števec pretvorimo v tridesetiški zapis in ga zapišimo. Za pretvorbo v tridesetiški zapis poskrbi notranja zanka (po j), ki se opira na dejstvo, da je ostanek po deljenju i s 30 ravno vrednost zadnje (najbolj desne) števke v tridesetiškem zapisu števila i, preostale števke pa skupaj tvorijo število z vrednostjo, ki je ravno celi del količnika po deljenju i s 30.

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

#define dolzina 5
const char *znaki = "0123456789ABCDEFGHJKMNPRTVWXYZ";
```

```
int main()
{
   int i, j, t, m = 20000000, k = strlen(znaki);
   char oznaka[dolzina + 1]; oznaka[dolzina] = 0;
   for (i = 0; i < m; i++) {
      for (j = 0, t = i; j < dolzina; j++) {
         oznaka[dolzina − 1 − j] = znaki[t % k];
         t /= k; }
      printf("%s\n", oznaka); }
   return 0;
}
```

Še ena možnost je, da ne pretvarjamo vsakega i posebej v tridesetiški sestav, pač pa tabelo oznaka uporabljamo kot tridesetiški števec, ki ga v vsaki iteraciji glavne zanke izpišemo in nato povečamo za 1. Vsak od elementov te tabele naj bo število od 0 do 29, ki predstavlja eno od števk trenutne oznake v tridesetiškem zapisu. Da takšno petmestno tridesetiško število povečamo za 1, lahko uporabimo podoben postopek kot pri ročnem seštevanju. Za 1 moramo povečati enice (element oznaka[4]) za 1; če so tako povečane enice še vedno pod 30, je postopek prištevanja s tem končan, v nasprotnem primeru pa pride do prenosa naprej: enice postavimo na 0 in v naslednjem koraku povečamo za 1 tridesetice (torej element oznaka[3]); če pride tudi tam do prenosa, bomo povečali devetstotice (element oznaka[2]) in tako naprej.

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

#define dolzina 5
const char *znaki = "0123456789ABCDEFGHJKMNPRTVWXYZ";

int main()
{
   int i, j, t, m = 20000000, k = strlen(znaki);
   int oznaka[dolzina];
   for (j = 0; j < dolzina; j++) oznaka[j] = 0;
   for (i = 0; i < m; i++) {
      for (j = 0; j < dolzina; j++) putchar(znaki[oznaka[j]]);
      putchar('\n');
      for (j = dolzina − 1; j >= 0; j−−) {
         if (++oznaka[j] < k) break;
         oznaka[j] = 0; }}
   return 0;
}
```

V anketah po tekmovanju imamo vsako leto med drugim vprašanje, ali tekmovalci razumejo kakšnega od programskih jezikov, v katerih objavljamo rešitve, dovolj dobro, da si lahko z izvorno kodo rešitev kaj pomagajo. Vsako leto dobimo tudi nekaj rahlo presenetljivih odgovorov, v katerih nam tekmovalci sporočajo, da C-ja ne razumejo dovolj dobro in da bi radi videli rešitve v C++. Da bi takim ljudem vsaj malo ustregli, si oglejmo še primer rešitve te naloge v C++. Odlikuje se med drugim po tem, da ne uporablja zank in pogojnih stavkov:

```
#include <iostream>
#include <iomanip>
#include <string>
```

```
using namespace std;

enum { B = 30 };
const char znaki[] = "0123456789ABCDEFGHJKMNPRTVWXYZ";

template<int n> struct U;
template<int n, int d> struct U2
{
  U2(string s) { (U2<n, d - 1>) s; U<n - 1>(s + znaki[d]); }
};
template<int n> struct U2<n, -1> { U2(string s) { } };
template<int n> struct U { U(string s) { (U2<n, B - 1>) s; } };
template<> struct U<0> { U(string s) { cout << s << endl; } };

int main()
{
  U2<5, 23>("");
  U2<4, 19>(string(1, znaki[24]));
  U2<3, 21>(string(1, znaki[24]) + znaki[20]);
  U2<2, 5>(string(1, znaki[24]) + znaki[20] + znaki[22]);
  U2<1, 19>(string(1, znaki[24]) + znaki[20] + znaki[22] + znaki[6]);
  return 0;
}
```

Vse dosedanje rešitve so oznake generirale na sistematičen način (in še celo v leksikografskem vrstnem redu). Naloge pa se lahko lotimo tudi drugače in generiramo naključne oznake; tako so nalogo rešili mnogi letošnji tekmovalci. Slabost tega pristopa je, da nam ne zagotavlja, da bo naslednja oznaka drugačna od vseh, ki smo jih doslej že izpisali, zato si moramo vse že izpisane oznake zapomniti in nato vsako novo oznako primerjati z njimi, da se prepričamo, če je res drugačna od njih.

Naj bo n število vseh možnih oznak, m pa število oznak, ki bi jih radi izpisali (pri naši nalogi je $n = 30^5 = 24{,}3 \cdot 10^6$ in $m = 20 \cdot 10^6$); k pa naj bo število oznak, ki smo jih že izpisali. Rešitev z naključnimi oznakami torej porabi $O(k)$ pomnilnika za shranjevanje že izpisanih oznak; ta poraba med delom počasi narašča in na koncu doseže $O(m)$. To je ena od slabosti te rešitve, saj porabi precej več pomnilnika od prej opisanih rešitev.

Druga slabost je, da če naslednja naključno zgenerirana oznaka ni različna od dosedanjih, jo moramo zavreči in poskusiti znova. Če smo že izpisali k oznak, je ostalo neuporabljenih še $n-k$ in verjetnost, da je naslednja oznaka res neuporabljena, je $(n-k)/n$; v povprečju bo torej treba $n/(n-k)$ poskusov, preden bomo dobili primerno oznako. To število je tem večje, čim bolj se k približuje n-ju (in čim manj neuporabljenih oznak nam je torej še ostalo).

Tretja slabost pa je, da tudi preverjanje, ali je nova oznaka drugačna od doslej izpisanih, vzame nekaj časa; ravno tako tudi shranjevanje nove oznake v pomnilnik. Cena teh dveh operacij je odvisna od tega, kako so v pomnilniku shranjene dosedanje oznake. Če so v razpršeni tabeli, je cena posameznega preverjanja približno $O(1)$, ravno tako tudi cena dodajanja nove oznake (ko se prepričamo, da je drugačna od dosedanjih); če so v drevesu (npr. rdeče-črnem), je cena tako preverjanja kot dodajanja $O(\log k)$; če so v urejeni tabeli, je cena preverjanja $O(\log k)$ (z bisekcijo), cena dodajanja pa $O(k)$ (vrivanje v urejen seznam); v neurejeni tabeli pa je

cena preverjanja kar $O(k)$, dodajanja pa le $O(1)$ (na konec seznama). Če torej ceno preverjanja označimo s $T_p(k)$, dodajanja pa s $T_d(k)$, je časovna zahtevnost celega postopka približno $T = \sum_{k=0}^{m-1}(\frac{n}{n-k}T_p(k) + T_d(k))$.

Z nekaj telovadbe se lahko prepričamo, da pri razpršeni tabeli ($T_d(k) = T_p(k) = O(1)$) dobimo $T = O(n \ln \frac{n}{n-m})$; pri drevesu ($T_d(k) = T_P(k) = O(\lg k)$) dobimo $T = O(n \ln \frac{n}{n-m} \lg m)$; pri urejenem seznamu in bisekciji čas T_d močno prevladuje nad T_p in dobimo $T = O(m^2)$; pri neurejenem seznamu pa dobimo kar $T = O(n^2 \ln \frac{n}{n-m})$.

Razmislimo še o tem, kaj bi se zgodilo, če ne bi preverjali, ali je trenutno zgenerirana oznaka drugačna od že izpisanih. Pri posamezni oznaki je verjetnost, da v nekem poskusu ni uporabljena, enaka $1 - 1/n$; verjetnost, da ni bila uporabljena v nobenem od m poskusov, je torej $(1 - 1/n)^m$, kar je približno $e^{-m/n}$. Verjetnost, da je bila izpisana vsaj enkrat, je torej približno $1 - e^{-m/n}$, torej je število različnih oznak, ki smo jih izpisali, približno $n(1 - e^{-m/n})$. Pri naših podatkih ($n = 30^5$ in $m = 20 \cdot 10^6$) je to približno 13,6 milijona, kar je veliko manj od zahtevanih 20 milijonov različnih oznak.

3. Citati

Spodnji program v zanki bere številke strani v spremenljivko stran. Za vsako stran moramo preveriti, ali ta številka strani nadaljuje trenutno skupino več zaporednih številk (na primer: če smo pred tem prebrali številke 28, 29 in 30, v trenutni iteraciji pa številko 31) ali ne. Če ne, moramo trenutno skupino izpisati; začetek te skupine hranimo v spremenljivki intervalOd, konec pa v intervalDo. Če sta začetek in konec enaka, izpišemo le eno število, sicer pa obe in med njima vezaj. Če je trenutna stran enaka intervalDo + 1, je to znak, da se trenutni interval nadaljuje in moramo le povečati intervalDo, izpišemo pa še ničesar. Na začetku postavimo intervalDo na −1; to je znak, da trenutnega intervala sploh še ni. Spremenljivka prvi skrbi za to, da pred prvim intervalom ne pišemo vejice in presledka, pred vsakim nadaljnjim pa. Glavna zanka naredi čisto na koncu seznama še eno iteracijo s stran = −1, kar poskrbi, da bomo izpisali tudi zadnji interval.

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

int main()
{
  int intervalOd, intervalDo = −1, stran;
  bool prvi = true;
  do {
    if (1 != scanf("%d", &stran)) stran = −1;

    /* Mogoče trenutna stran nadaljuje trenutni interval. */
    if (stran == intervalDo + 1) { intervalDo++; continue; }

    /* Sicer pa se začenja nov interval.
       Mogoče moramo najprej izpisati prejšnji interval. */
    if (intervalDo > 0) { /* Izpišimo prejšnji interval. */
      if (! prvi) printf(", "); else prvi = false;
      printf("%d", intervalOd);
      if (intervalDo > intervalOd) printf("-%d", intervalDo); }

    intervalOd = intervalDo = stran; /* Začnimo nov interval. */
```

```
  } while (stran > 0);
  return 0;
}
```

4. Smrkci

Spodnja rešitev se v zanki premika po nizu, ki opisuje delitve, in v vsakem koraku popravi koordinato tistega roba, ki se zaradi te delitve premakne. Na primer, če si trenutni tajkunosmrkec prilasti severno polovico ozemlja, se severni rob nerazdeljenega ozemlja premakne na pol poti med (dosedanjim) severnim in južnim robom.

```
#include <stdio.h>

void Nerazdeljeno(const char* delitve)
{
  int s = 0, j = 256, z = 0, v = 256;
  while (*delitve)
  switch (*delitve++) {
    case 'S': s = (s + j) / 2; break;
    case 'J': j = (s + j) / 2; break;
    case 'Z': z = (z + v) / 2; break;
    case 'V': v = (z + v) / 2; }
  printf("(%d, %d, %d, %d)\n", z, s, v, j);
}
```

Če bi se spremenila začetna velikost ozemlja, bi morali le popraviti vrednosti, s katerima inicializiramo spremenljivki j (višina ozemlja) in v (širina ozemlja). Če se spremeni število tajkunosmrkcev, pa ni treba naše rešitve nič spreminjati, saj že zdaj pregleda niz delitve od začetka do konca, ne glede na to, kako dolg je. Odvisno od dimenzij ozemlja in števila tajkunosmrkcev bi se lahko zgodilo, da koordinate po nekaj razpolovitvah ne bi bile več cela števila; če bi hoteli podpreti tudi ta primer, bi bilo dobro namesto tipa **int** uporabiti **double** in v klicu printf popraviti %d v %g ali kaj podobnega.

5. Železnica

V globalni spremenljivki stanje (ki je tabela dveh logičnih vrednosti) hranimo trenutno stanje obeh senzorjev; ob inicializaciji postavimo oba elementa tabele na **false**, saj naloga pravi, da takrat noben senzor nima prekinjenega žarka. Poleg tega imejmo še globalni spremenljivki, ki hranita število vagonov za vsako smer vožnje (stLevih in stDesnih).

Naloga pravi, da je razmik med dvema zaporednima vagonoma gotovo večji kot razmik med števcema, zato bosta med vagonoma gotovo nekaj časa oba števca imela neprekinjen žarek. Prihod novega vagona lahko torej vedno prepoznamo po tem, da sta bila prej oba števca neprekinjena, nato pa na enem od njiju pride do spremembe: na levem, če prihaja vagon z leve (in se pelje v desno), oz. na desnem, če prihaja vagon z desne (in se pelje v levo). Ta razmislek upošteva podprogram SpremembaSenzorja in ob prihodu novega vagona poveča ustrezni števec vagonov. V vsakem primeru pa nato tudi popravi ustrezni element tabele stanje, da bo odražal novo stanje števca.

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

```
bool stanje[2];
int stLevih, stDesnih;

void Inicializacija() { stanje[0] = false; stanje[1] = false; stLevih = 0; stDesnih = 0; }
void Izpis() { printf("%d vagonov v levo, %d v desno\n", stLevih, stDesnih); }

void SpremembaSenzorja(int senzor, bool prekinjen)
{
  if (! stanje[0] && ! stanje[1])
    if (senzor == 1) stDesnih++; else stLevih++;
  stanje[senzor - 1] = prekinjen;
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Soglasniški podnizi

V zunanji zanki berimo besede, za vsako od njih ugotovimo, kako dolga je v njej najdaljša strnjena skupina soglasnikov (spremenljivka ocena), in če je daljša od najdaljše doslej znane (ki jo hrani spremenljivka najOcena), si trenutno besedo zapomnimo (v spremenljivki najBeseda).

To, kako dolga je najdaljša skupina soglasnikov v trenutni besedi, lahko ugotovimo z zanko po črkah besede. Spremenljivka d šteje, koliko strnjenih soglasnikov smo videli pred trenutno črko. Če je trenutna črka soglasnik, povečamo d za 1, sicer pa (torej če smo pri samoglasniku) ga postavimo na 0, saj je dosedanje skupine soglasnikov konec. Največja vrednost d je tako tudi dolžina najdaljše strnjene skupine soglasnikov v trenutni besedi in si jo zapomnimo v spremenljivki ocena.

```
#include <stdio.h>
#define MaxDolz 100

int main()
{
   char beseda[MaxDolz + 1], najBeseda[MaxDolz + 1] = "";
   int ocena, najOcena = −1, i, j, d;
   while (gets(beseda)) /* Preberimo naslednjo besedo. */
   {
      for (i = 0, d = 0, ocena = 0; beseda[i]; i++) {
         /* Pred črko beseda[i] je d soglasnikov. */
         if (strchr("aeiou", beseda[i])) d = 0;
         else d++;
         /* Črka beseda[i] je na koncu skupine d soglasnikov. */
         if (d > ocena) ocena = d; }
      if (ocena > najOcena) { /* Najboljši rezultat doslej. */
         strcpy(najBeseda, beseda); najOcena = ocena; }
   }
   printf("%s\n", najBeseda); return 0;
}
```

2. Kje sem že to posnel?

Hkrati se bomo sprehajali po zaporedju točk in zaporedju slik. V spremenljivkah t1 in t2 hranimo čas prejšnje in trenutne točke, v x1, y1, x2 in y2 pa njune koordinate.

V vsaki iteraciji zunanje zanke preberimo naslednjo sliko (s časom tSlike). Če pade tSlike med t1 in t2, lahko tej sliki kar takoj pripišemo koordinate (bodisi prejšnje točke ali pa trenutne točke, odvisno, katera ji je po času bližja). Če pa je tSlike večji od t2, se premaknimo naprej po zaporedju točk (trenutna točka postane prejšnja, novo trenutno točko pa preberemo s standardnega vhoda); to ponavljamo, dokler ne pridemo do dveh takih točk, da naša slika po času leži med njima.

Nekaj preglavic nam povzroči še možnost, da je slika posneta pred prvo točko ali pa za zadnjo točko v zaporedju. Zato na začetku obdelave hrani t2 prvo točko zaporedja, t1 pa postavimo na −1 in v okviru tega para točk obdelamo vse slike, ki po času padejo pred prvo točko zaporedja. Podobno na koncu obdelave t1 hrani zadnjo točko zaporedja, t2 pa postavimo na −1 in v okviru tega para točk obdelamo vse slike, ki po času padejo za zadnjo točko zaporedja.

```
#include <stdio.h>

extern int PreberiNaslednjoSliko();
extern void VpisiKoordinateInShrani(double x, double y);

int main()
{
  int t1 = -1, t2, tSlike; double x1 = 0, x2 = 0, y1, y2;
  scanf("%d %lf %lf", &t2, &x2, &y2);
  while ((tSlike = PreberiNaslednjoSliko()) > 0)
  {
    while (t2 > 0 && tSlike >= t2) {
      /* Preberimo naslednjo točko. */
      t1 = t2; x1 = x2; y1 = y2;
      if (3 != scanf("%d %lf %lf", &t2, &x2, &y2)) t2 = -1; }
    /* Zdaj imamo dve točki, za kateri vemo, da naša slika leži
       med njima: t1 <= tSlike < t2. Če smo na začetku zaporedja
       točk (t1 < 0), leve neenakosti seveda ne upoštevamo, če
       pa smo že na koncu zaporedja točk (t2 < 0), pa ne upoštevamo
       desne neenakosti. */
    if (t1 > 0 && (t2 < 0 || tSlike - t1 < t2 - tSlike))
      VpisiKoordinateInShrani(x1, y1);
    else
      VpisiKoordinateInShrani(x2, y2);
  }
  return 0;
}
```

3. Avtocesta

Imejmo seznam S, v katerem vsak element vsebuje številko tovornjaka in čas, ob katerem je ta tovornjak peljal mimo začetne postaje, seznam naj bo urejen po času (oz. po številki tovornjaka, kar je tako ali tako ekvivalentno, ker prva postaja dodeljuje številke v naraščajočem vrstnem redu).

algoritem *ObPrehodu*(*postaja*, k):
  while S ni prazen:
    naj bo t čas prvega elementa v S;
    if je t manj kot MinimalniCas sekund v preteklosti
      in cel seznam S ni predolg (glede na razpoložljivi pomnilnik) **then break**
    pobriši prvi element iz S
  if $postaja = 1$:
    dodaj par $\langle k, \text{TrenutniCas}\rangle$ na konec seznama S
  else if $postaja = 2$ **and** $k \neq 0$ **and** S ni prazen:
    naj bo m številka prvega elementa v S;
    če je $k \geq m$, izpiši k;

Rešitev deluje takole: ko pride tovornjak številka k na končno postajo, imamo v S same take tovornjake, ki so prišli mimo začetne postaje pred manj kot MinimalniCas sekundami. Torej, če je najstarejši zapis v S tisti s številko m in je $k < m$, potem je k vozil dovolj počasi in ga ni treba izpisati, sicer pa ga moramo. Zadeva odpove le v primerih, ko je prometa toliko, da S ne vsebuje vseh tovornjakov, ki so prišli v zadnjih MinimalniCas sekundah mimo začetne postaje; takrat se lahko zgodi, da smo

k že pobrisali iz seznama in je zdaj $k < m$, tako da k-ja ne bomo izpisali, četudi je mogoče vozil prehitro.

Seznam S lahko implementiramo kot verigo elementov, povezanih s kazalci (*linked list*), lahko pa tudi kot krožno tabelo (*ring buffer*).

Zgoraj opisana rešitev ima še tole majhno slabost: če nekdo prehitro pride od začetne do končne postaje in se potem še večkrat pelje mimo končne postaje, ne da bi minilo MinimalniCas časa od zadnjega obiska začetne postaje, ga bomo izpisali po večkrat. Lepo bi bilo, če bi ga lahko zbrisali iz seznama S, ko smo ga prvič izpisali. Toda to bi pomenilo linearni sprehod čez seznam, da preverimo, če je k sploh še v njem (pogoj $k \geq m$ je zdaj le potreben, ne pa tudi zadosten), in ga pobrišemo. Hitrejša različica je, da ga ne pobrišemo, pač pa ga le označimo kot že izpisanega in ga kasneje ne izpisujemo več. Ker je začetna postaja dodeljevala številke po vrsti in nas o vsaki obveščala, vemo, da je tovornjak k v celici $S[k - m]$, če je S shranjen v tabeli in je m številka prvega tovornjaka; tako ni težko priti do tovornjaka k in videti, če je označen kot že izpisan, oz. ga označiti kot že izpisanega, če še ni bil tako označen. Da bo brisanje z začetka seznama cenejše, pa moramo to tabelo seveda uporabljati kot krožno tabelo (*ring buffer*).

4. UTF-5

V globalni spremenljivki znak hranimo doslej prejete bite trenutnega znaka. Ko pride nova peterica bitov, zamaknemo dosedanje bite spremenljivke znak za štri mesta v levo in na spodnja mesta vpišemo spodnje štiri bite nove peterice. Če je najvišji bit peterice prižgan, vemo, da je trenutnega znaka konec in ga lahko pošljemo v nadaljnjo obdelavo (pokličemo PrejemZnaka), spremenljivko znak pa spet postavimo na 0.

```
extern void PrejemZnaka(int x);

int znak = 0;

void PrejemPeterice(int x)
{
  znak <<= 4;
  znak |= x & 15;
  if (x & 16) { PrejemZnaka(znak); znak = 0; }
}
```

Mimogrede, predlog za UTF-5 res obstaja, vendar ni bil deležen kakšnega večjega zanimanja.[4]

5. break considered harmful

Stavek **break** pravzaprav naredi dvoje: prekine trenutno iteracijo zanke in nato tudi poskrbi, da se ne bo izvedla nobena iteracija te zanke več, pač pa se začnejo izvajati

[4] J. Seng, M. Duerst, T. W. Tan: *UTF-5, a transformation format of Unicode and ISO 10646*, Internet Draft, 28 Jan 2000. Naša naloga v eni podrobnosti odstopa od tega predloga: če iz nekega znaka nastane več peteric bitov, je pri nas prižgan zgornji bit v zadnji od teh peteric, v predlogu Senga in soavtorjev pa v prvi od njih. Slednja možnost bi bila za potrebe naše naloge neugodna, ker prejemnik ne bi mogel zanesljivo vedeti, ali je že dobil vse peterice trenutnega znaka ali pride še kakšna.

stavki, ki sledijo zanki. Po drugi strani pa **continue** ravno tako kot **break** prekine trenutno iteracijo zanke, vendar pa (za razliko od **break**) nato nadaljuje z naslednjo iteracijo (če je seveda pogoj za nadaljevanje zanke izpolnjen).

Če torej hočemo stavek **break** odpraviti, lahko namesto njega uporabimo **continue**, vendar moramo nekako zagotoviti, da pogoj za nadaljevanje zanke ne bo izpolnjen. Vpeljemo lahko na primer pomožno logično spremenljivko (recimo ji P, tipa **bool** oz. boolean), ki pove, ali sploh smemo preverjati prvotni pogoj za nadaljevanje zanke. Ko hočemo zanko prekiniti, postavimo P na **false** in zanka ne bo šla v naslednjo iteracijo. Iz zanke

while (Pogoj) S

tako dobimo

{ P = **true**; **while** (P && Pogoj) S' }

Pri tem dobimo S' iz S tako, da v njem vsak stavek **break**, ki se nanaša na našo zanko (ne pa na kakšno bolj notranjo) spremenimo v { P = **false**; **continue**; }. (Temu zadnjemu pogoju najlažje ustrežemo tako, da predelujemo zanke od notranjih proti zunanjim; tako bodo notranje že brez stavkov **break**, preden bomo prišli do zunanjih.) Spremenljivka P seveda ne sme biti za vse zanke enaka, ampak mora imeti vsaka zanka svojo (oz. vsaj vsaka globina gnezdenja svojo).

Zgornja rešitev se zanaša na predpostavko, da v pogoju P && Pogoj program sploh ne bo šel računat pogoja Pogoj, če bo prvi operand, torej P, enak **false**. Tako deluje operator && (logični in) v C-ju in številnih njemu sorodnih jezikih. Ta podrobnost je pomembna, če hočemo, da predelani program res deluje enako kot prvotni. Prvotni program, če je zanko prekinil z **break**, vsekakor ni šel še enkrat računat njenega pogoja za nadaljevanje (torej izraza Pogoj), zato je tudi naš predelani program ne sme. (Konec koncev nič ne vemo, kaj pomeni računanje izraza Pogoj — mogoče se v njem skriva kak zamuden izračun ali pa klic kakšnega podprograma, ki celo kaj izpiše ipd.) Če imamo kakršne koli pomisleke glede tega, ali se operator && res obnaša na opisani način, lahko rešitev spremenimo takole:

```
{ P = true;
  while (P) {
    P = Pogoj; if (! P) continue;
    S' } }
```

Tu res ni več dvoma, da če je P enak **false**, se izraz Pogoj ne računa več.

Če po opisanem postopku predelamo podprogram iz besedila naloge, dobimo nekaj takega:

```
podprogram DveZanki;
spremenljivke: i, j, n — cela števila; p1, p2 — logični spremenljivki;
{
  n = 100; i = 0;
  { p1 = true; while (p1) {
    p1 = (i < n); if (! p1) continue;
    {
      j = i;
      { p2 = true; while (p2) {
        p2 = (j < n); if (! p2) continue;
```

```
            {
               Izpis1(i, j);
               if (Funkcija1(i, j)) { p2 = false; continue; }
               Izpis2(i, j);
               j = j + 1;
            } } }
         if (Funkcija2(i, j)) { p1 = false; continue; }
         { p2 = true; while (p2) {
            p2 = (j > i); if (! p2) continue;
            {
               Izpis3(i, j);
               j = j − 1;
            } } }
         i = i + 1;
      } } }
}
```

Oglejmo si še, kako bi rešili podoben problem: recimo, da bi namesto stavka **break** hoteli odpraviti stavek **continue**. Razmišljamo lahko podobno kot zgoraj; oba stavka najprej prekineta izvajanje trenutne iteracije svoje zanke, vendar pa **continue** potem nadaljuje izvajanje z naslednjo iteracijo, **break** pa ne. Če torej hočemo simulirati **continue** s pomočjo stavka **break**, je koristno, če vsako iteracijo naše zanke obravnavamo kot novo vgnezdeno zanko, ki se vedno izvede v največ eni iteraciji; **continue** lahko potem spremenimo v **break**, ki bo tako prekinil le notranjo zanko (in s tem trenutno iteracijo prvotne zanke). Takšna podvojitev zank pa nam nekoliko zaplete primere, ko bi radi zanko zares prekinili, torej ko je že v prvotnem programu bil stavek **break**; takšen **break** bo zdaj prekinil le notranjo zanko, ne pa tudi zunanje. Pomagamo si lahko s pomožno spremenljivko, v kateri povemo, da naj se zunanja zanka ne nadaljuje. Recept je torej tak: vsako zanko

```
while (Pogoj) S
```

predelamo v

```
while (Pogoj) {
   while (true) {
      P = true;
      S′
      P = false; break; }
   if (P) break; }
```

Pri tem smo S' dobili iz S tako, da smo vsak **continue**, ki se nanaša na trenutno zanko (ne pa na kakšno globlje vgnezdeno) spremenili v { P = **false**; **break**; }. Podobno kot zgoraj tudi tu spremenljivka P ne sme biti ena sama, pač pa po ena za vsako globino gnezdenja. Če se je torej trenutna iteracija izvedla v celoti ali pa se je končala s **continue**, bo imel P vrednost **false** in zunanja zanka bo nadaljevala z delom; če pa se je trenutna iteracija v prvotnem programu prekinila z **break**, bo imel P še vedno vrednost **true** (ki jo je dobil pred začetkom izvajanja te iteracije) in se lahko na to opremo, da tudi zunanjo zanko takoj prekinemo.

Še večji izziv dobimo, če hočemo odpraviti tako stavke **break** kot **continue**. Recimo, da smo se stavkov **break** že znebili po malo prej opisanem postopku; zdaj bi torej radi odpravili še stavke **continue**. Program bomo predelovali od globlje vgnezdenih zank

proti bolj zunanjim; recimo torej zdaj, da se ukvarjamo z neko zanko **while** (Pogoj) S in vemo, da se v S pojavljajo le še taki stavki **continue**, ki se nanašajo na našo trenutno zanko (ker smo tiste stavke **continue**, ki se nanašajo na vgnezdene zanke znotraj S, že odpravili). Uvedli bomo pomožno logično spremenljivko C, ki pove, kdaj je treba preskočiti preostanek trenutne iteracije zanke. Našo zanko torej predelajmo v

while (Pogoj) { C = **false**; $f(S)$ }

Pri tem transformacija f deluje takole:

continue;	$\mapsto$	C = **true**;
if (P) T	$\mapsto$	**if** (P) $f(T)$
if (P) T **else** U	$\mapsto$	**if** (P) $f(T)$ **else** $f(U)$
{ $S_1 \ldots S_n$ }	$\mapsto$	{ $S_1 \ldots S_{k-1}$ $f(S_k)$ **if** (! C) $f($ { $S_{k+1} \ldots S_n$ } $)$ }

V zadnjem primeru je mišljeno, da je k najmanjši tak indeks, pri katerem je S_k bodisi enak **continue** bodisi vsebuje **continue** nekje globlje znotraj sebe. Prireditvene stavke in klice podprogramov naj f pusti nespremenjene. Podobno kot prej mora imeti tudi tu vsaka globina gnezdenja svojo spremenljivko C.

Nazadnje se lahko znebimo tudi stavka **if**. Rešitev za odpravljanje besede **else** smo videli že v besedilu naloge, zato zdaj recimo, da so nam ostali le še navadni **if** brez dela **else**. Stavek **if** (Pogoj) S lahko spremenimo v

{ P = Pogoj;
  while (P) { S P = **false**; } }

Ta rešitev se seveda zanaša na to, da smo pred tem iz S že odpravili stavke **break** in **continue**.

Mimogrede, navdih za naslov te naloge je slavni članek Edsgerja Dijkstre *Go To Statement Considered Harmful* (CACM, 11(3):147–8, March 1968). Za razliko od stavka **go to** pa drugi tu obravnavani stavki nikoli niso bili predmet kakšnih večjih preganjanj.

1. Undo

Koristno je, če seznam ukazov obdelujemo od konca proti začetku; ko pri tem naletimo na `UNDO` k, vemo, da moramo preskočiti predhodnih k ukazov. Za ukaze `WRITE`, ki jih nismo preskočili, pa si moramo nekje zapomniti, da bomo morali izpisati njihove nize (spodnji program shranjuje številke teh ukazov v tabeli izpis). Ko na ta način pregledamo celoten seznam, gremo še enkrat po spisku ukazov, označenih za izpis, in pri vsakem od njih izpišemo pripadajoči niz. Ker smo vhodni seznam pregledovali od konca proti začetku in številke ukazov, pri katerih je potreben izpis, dodajali na konec tabele izpis, so zdaj na koncu te tabele številke ukazov z začetka seznama, na začetku te tabele pa so številke ukazov s konca vhodnega seznama; tudi tabelo izpis moramo torej zdaj obdelati od konca proti začetku.

```
#include <stdio.h>
#include <stdlib.h>
#define MaxN 100000
#define MaxD 100

int main(int argc, char** argv)
{
   FILE *f; int i, n, k, *undo;
   char **besede, **izpis, vrstica[MaxD + 2];

   /* Preberimo število ukazov in pripravimo tabele. */
   f = fopen("undo.in", "rt");
   fscanf(f, "%d", &n);
   besede = (char **) malloc(n * sizeof(char *));
   undo = (int *) malloc(n * sizeof(int));
   /* Preberimo ukaze. Za i-ti ukaz shranimo besedo v besede[i], če gre za ukaz WRITE;
      če pa gre za UNDO, shranimo število razveljavljenih korakov v undo[i]. */
   for (i = 0; i < n; i++) {
      fscanf(f, "%s", vrstica);
      if (vrstica[0] == 'U') fscanf(f, "%d", &undo[i]);  /* UNDO */
      else {  /* WRITE */
         undo[i] = -1;
         fscanf(f, "%s", vrstica);
         besede[i] = (char *) malloc(strlen(vrstica) + 1);
         strcpy(besede[i], vrstica); }}
   fclose(f);

   /* Poglejmo, katere besede bo treba izpisati. Števec k pove, koliko jih je. */
   izpis = (char **) malloc(n * sizeof(char *));
   for (i = n - 1, k = 0; i >= 0; i--)
      if (undo[i] >= 0) i -= undo[i];
      else izpis[k++] = besede[i];

   /* Izpišimo rezultat. */
   f = fopen("undo.out", "wt");
   while (k > 0) fprintf(f, "%s", izpis[--k]);
   fprintf(f, "\n"); fclose(f);

   /* Pospravimo za sabo. */
   for (i = 0; i < n; i++) if (undo[i] < 0) free(besede[i]);
```

```
   free(izpis); free(besede); free(undo);
   return 0;
}
```

2. Žaba in lokvanji

Za vsak lokanj si zapomnimo številko najbližjega še nepotopljenega lokvanja levo in desno od njega (tabeli L in D v spodnjem programu). Spodnji program šteje lokvanje od 0 do $n-1$; pretvarjajmo se, da je levo od lokvanja 0 še lokvanj $-(k+1)$, desno od $n-1$ pa lokvanj $n+k$; takšnih dveh žaba pri vidljivosti k gotovo ne bi videla.

S pomočjo teh dveh tabel lahko v vsakem koraku zelo hitro ugotovimo, ali žaba vidi kak lokvanj v okolici trenutnega ali ne. Ko se lokvanj t potopi, moramo tabeli popraviti: t-jeva soseda, L[t] in D[t], zdaj postaneta soseda drug drugega.

```
#include <stdio.h>
#define MaxN 1000000
#define MaxM 2000000

int main()
{
   FILE *f; int i, t, n, m, k, L, D, *levi, *desni, potopi;
   /* Preberimo število lokvanjev. */
   f = fopen("zaba.in", "rt");
   fscanf(f, "%d %d %d", &n, &k, &m);

   /* Pripravimo tabeli, v katerih za vsak lokvanj piše,
      kateri je najbližji nepotopljeni sosed v vsaki smeri. */
   levi = (int *) malloc(n * sizeof(int));
   desni = (int *) malloc(n * sizeof(int));
   for (t = 0; t < n; t++) { levi[t] = t − 1; desni[t] = t + 1; }

   /* Levo od prvega in desno od zadnjega lokvanja si mislimo
      nek zelo oddaljen lokvanj, ki ga žaba ne vidi. */
   levi[0] = −k − 1; desni[n − 1] = n + k;

   /* Berimo zaporedje skokov. */
   for (i = 0; i < m; i++)
   {
      fscanf(f, "%d %d", &t, &potopi); t −= 1;
      /* Žaba je na lokvanju t. Katera sta najbližja nepotopljena soseda
         in ali ju žaba še lahko vidi? */
      L = levi[t]; D = desni[t];
      if (L < t − k && D > t + k) break;

      /*Če je treba, potopimo lokvanj t; L in D s tem postaneta soseda drug drugemu.*/
      if (potopi) {
         if (L >= 0) desni[L] = D;
         if (D < n) levi[D] = L; }
   }
   fclose(f);

   /* Izpišimo rezultat. */
   f = fopen("zaba.out", "wt");
   fprintf(f, "%d\n", (i == m ? 0 : t + 1)); fclose(f);

   /* Pospravimo za sabo. */
   free(levi); free(desni); return 0;
}
```

Učinek tako uporabljenih tabel L in D je pravzaprav ta, da smo nepotopljene lokvanje organizirali v dvojno povezan seznam (*doubly linked list*). Prednost takšnih tabel pred bolj tradicionalnim seznamom, pri katerem bi bili posamezni členi dinamično alocirani na kopici, pa je v tem, da lahko pri tabelah za poljubni dani t enostavno in učinkovito pridemo do člena seznama, ki vsebuje lovanj t.

3. Otoki

Nalogo lahko rešimo z barvanjem zemljevida. Na začetku so vodna polja barve 0, kopna pa barve 1. Nato pobarvamo z barvo 2 vsa vodna polja, dosegljiva po vodi z roba zemljevida. Nato pobarvamo z barvo 3 vsa še nepobarvana kopna polja, ki so dosegljiva po kopnem s polj barve 2. Nato pobarvamo z barvo 4 vsa še nepobarvana vodna polja, ki so dosegljiva po vodi s polj barve 3; in tako naprej. Barvo 2 dobi torej morje, jezera imajo barve 4, 6, 8 itd., otoki pa barve 3, 5, 7, itd. Stopnja varnosti otoka barve b je torej $(b-3)/2$, kar nam bo prišlo prav pri izpisu rezultatov. Postopek se konča, ko je pobarvan celoten zemljevid.

Za posamezno fazo barvanja skrbi v spodnji rešitvi podprogram Pobarvaj, ki mu lahko nekaj začetnih (in že pobarvanih) polj podamo v vhodni vrsti (vrsta1 z *rep1 elementi), od tam pa bo z iskanjem v širino pobarval vsa polja barve izBarve1 v barvo vBarvo1. Za vsako polje, ki ga pobere iz vhodne vrste, pogleda tudi njegove sosede; če so barve izBarve2, jih pobarva v vBarvo2 in jih doda v izhodno vrsto (vrsta2, ki bo imela ob vrnitvi iz podprograma *rep2 elementov). Ta izhodna vrsta bo prišla prav pri naslednjem barvanju, da bomo vedeli, kje sploh začeti z njim.

```
#include <stdio.h>

#define MaxW 3000
#define MaxH 3000

int w, h;
int *z;
const int dx[4] = { −1, 1, 0, 0 };
const int dy[4] = { 0, 0, −1, 1 };

/* Spodnji podprogram predpostavi, da je v vrsti 1 že rep1 polj, ki so bila nekdaj barve
   izBarve1, zdaj pa so barve vBarvo1. Ta podprogram bo poiskal še vsa druga polja,
   ki so dosegljiva iz njih po poljih barve izBarve1, jih dodal v to vrsto in jih prebarval
   v barvo vBarvo2. Obenem bo vsa tista polja, ki so barve izBarve2 in mejijo na kakšno
   polje iz vrste 1, dodal v vrsto 2 in jih pobarval v barvo vBarvo2. Za vrsto 2 pa
   predpostavi, da je na začetku prazna. */
void Pobarvaj(int *vrsta1, int *rep1, int izBarve1, int vBarvo1,
              int *vrsta2, int *rep2, int izBarve2, int vBarvo2)
{
  int glava = 0, r1 = *rep1, r2 = 0, u, d, x, y, xx, yy, v;
  while (glava < r1)
  {
    u = vrsta1[glava++];
    x = u % w; y = u / w;
    for (d = 0; d < 4; d++)
    {
      xx = x + dx[d]; yy = y + dy[d];
      if (xx < 0 || yy < 0 || xx >= w || yy >= h) continue;
      v = w * yy + xx;
```

```
      if (z[v] == izBarve1) {
        vrsta1[r1++] = v; z[v] = vBarvo1; }
      else if (z[v] == izBarve2) {
        vrsta2[r2++] = v; z[v] = vBarvo2; }
    }
  }
  *rep1 = r1; *rep2 = r2;
}

int main()
{
  char vrstica[MaxW + 2]; int x, y, u, *vrsta1, *vrsta2, *t, n1, n2, barva, najStopnja;
  FILE *f;

  /* Preberimo velikost zemljevida. */
  f = fopen("otoki.in", "rt");
  fscanf(f, "%d %d\n", &w, &h);

  /* Alocirajmo pomnilnik za zemljevid in dve vrsti. */
  z = (int *) malloc(w * h * sizeof(int));
  vrsta1 = (int *) malloc(w * h * sizeof(int)); n1 = 0;
  vrsta2 = (int *) malloc(w * h * sizeof(int));

  /* Preberimo zemljevid. Polja na zunanjem robu dodajmo v vrsto 1. */
  for (y = 0; y < h; y++) {
    fgets(vrstica, w + 2, f);
    for (x = 0; x < w; x++) {
      u = w * y + x;
      if (x == 0 || x == w - 1 || y == 0 || y == h - 1) { vrsta1[n1++] = u; z[u] = 2; }
      else z[u] = (vrstica[x] == '#') ? 1 : 0; }}
  fclose(f);

  /* Trenutno so vsa kopna polja barve 1, vsa vodna pa barve 0, razen tistih na zunanjem
     robu, ki so barve 2. Pobarvajmo zdaj celo morje z barvo 2, nato otoke stopnje 0 z
     barvo 3, nato jezera na teh otokih z barvo 4, nato otoke stopnje 1 z barvo 5, nato
     jezera na teh otokih z barvo 6, nato otoke stopnje 2 z barvo 7 in tako naprej. */
  najStopnja = -1; barva = 2;
  while (n1 > 0) {
    if ((barva % 2) == 1) najStopnja = (barva - 3) / 2;
    Pobarvaj(vrsta1, &n1, barva & 1, barva,
             vrsta2, &n2, (barva ^ 1) & 1, barva + 1);
    t = vrsta1; vrsta1 = vrsta2; vrsta2 = t;
    u = n1; n1 = n2; n2 = n1; barva++; }

  /* Izpišimo rezultat. */
  f = fopen("otoki.out", "wt"); fprintf(f, "%d\n", najStopnja); fclose(f);

  /* Pospravimo za sabo. */
  free(z); free(vrsta1); free(vrsta2); return 0;
}
```

4. Strupi

Nalogo si lahko predstavljamo kot problem najkrajših poti v grafu, pri čemer ima graf po eno točko za vsako možno kombinacijo strupov, povezava od u do v pa obstaja, če lahko iz u z dodajanjem enega strupa pridemo v v (po tistem, ko se izničijo vse kombinacije strupov, ki se pač v teh razmerah lahko). Če bi pacient ob takem dodajanju umrl, povezave ne vzpostavimo. Ker štejemo vse povezave za enako dolge,

lahko za iskanje najkrajših poti uporabimo kar iskanje v širino. Graf ima največ 2^n točk, kar je pri nas (ko je $n \leq 15$) še obvladljivo. Zaradi učinkovitosti je koristno, če kombinacije strupov predstavimo kar s celimi števili, pri katerih vsak od spodnjih n bitov pove, ali je tisti strup v kombinaciji prisoten ali ne.

```
#include <stdio.h>
#define MaxN 15

int main()
{
  FILE *f;
  int n, m, k, s, u, v, i, j, *A, *B, *vrsta, glava, rep, *pred, *kako;
  /* Preberimo n, m in k ter alocirajmo pomnilnik. */
  f = fopen("strupi.in", "rt");
  fscanf(f, "%d %d %d", &n, &m, &k);
  A = (int *) malloc(sizeof(int) * m);
  B = (int *) malloc(sizeof(int) * k);
  vrsta = (int *) malloc(sizeof(int) << n); glava = rep = 0;
  kako = (int *) malloc(sizeof(int) << n);
  pred = (int *) malloc(sizeof(int) << n);
  for (i = 0; i < (1 << n); i++) pred[i] = −1;
  /* Preberimo kombinacije strupov in začetno stanje. */
  for (i = 0; i < m + k + 1; i++) {
    fscanf(f, "%d", &j);
    for (s = 0; j > 0; j−−) { fscanf(f, "%d", &u); s |= 1 << (u − 1); }
    if (i < m) A[i] = s;
    else if (i < m + k) B[i − m] = s;
    else { vrsta[rep++] = s; pred[s] = s; }}
  fclose(f);

  /* Iskanje v širino (začetno stanje imamo že v vrsti). */
  while (glava < rep && pred[0] < 0)
  {
    u = vrsta[glava++];
    for (i = 0; i < n; i++)
    {
      if ((u & (1 << i)) != 0) continue;
      /* Kaj se zgodi, če v stanje u dodamo strup i? Mogoče pacient takoj umre. */
      v = u | (1 << i);
      for (j = 0; j < m; j++) if ((v & A[j]) == A[j]) { v = −1; break; }
      if (v < 0) continue;
      /* Mogoče pride do kakšnih nevtralizacij. */
      for (j = 0; j < k; j++)
        if (((u | (1 << i)) & B[j]) == B[j]) v &= B[j];
      /* Če novega stanja, v, še ne poznamo, ga dodajmo v vrsto. */
      if (pred[v] >= 0) continue;
      pred[v] = u; kako[v] = i;
      vrsta[rep++] = v;
      if (v == 0) break;
    }
  }

  /* Izpišimo rezultat. */
  f = fopen("strupi.out", "wt");
  if (pred[0] < 0) fprintf(f, "-1\n");
  else {
```

```
    /* S pomočjo tabel kako in pred rekonstruirajmo zaporedje dodajanja strupov. */
    u = 0; glava = 0;
    while (u != s) {
      vrsta[glava++] = kako[u]; u = pred[u]; }
    /* Na koncu zaporedja imamo strupe, ki smo jih dodali najprej;
       zdaj torej izpišimo zaporedje od konca proti začetku. */
    fprintf(f, "%d\n", glava);
    rep = glava; while (glava > 0) {
      if (glava < rep) fprintf(f, " ");
      fprintf(f, "%d", vrsta[--glava] + 1); }}
  fclose(f);

  /* Pospravimo za sabo. */
  free(A); free(B); free(vrsta); free(kako); free(pred); return 0;
}
```

Oglejmo si še eno možno izboljšavo gornjega postopka. Ko poberemo točko u iz vrste, gre sedanji postopek po vseh možnih novih strupih i in pri vsakem pregleda tabelo A (da vidi, če bi pacient preživel dodajanje strupa i) in mogoče še B (da ugotovi, v katero stanje pridemo po dodajanju strupa i in izničenju za to primernih kombinacij). To nam da v najslabšem primeru časovno zahtevnost $O(n \cdot (m+k))$ pri vsakem u. Za naše testne primere je ta rešitev že čisto dovolj hitra, vendarle pa jo lahko še malo izboljšamo.

Recimo, da smo iz vrste pravkar vzeli nek u, torej neko kombinacijo strupov ($u \subseteq \{1, \ldots, n\}$). Zgornji algoritem bi šel v zanki po vseh možnih dodatnih strupih i, ki jih še ni v u; in pri vsakem i bi šel v še eni vgnezdeni zanki po vseh možnih smrtonosnih kombinacijah A_j in za vsako preverjal, ali je vsebovana v $u \cup \{i\}$ ali ne. Toda mi že zdaj vemo, da A_j ni podmnožica u-ja, saj sicer do u-ja pri našem pregledovanju grafa sploh ne bi mogli priti. Torej je $A_j \subseteq u \cup \{i\}$ mogoče le v primeru, da je $i \in A_j$ in da je i edini strup, ki je prisoten v A_j, ne pa v u (če bi bilo takih strupov več, bi A_j ne mogla biti podmnožica $u \cup \{i\}$). Pri vsaki smrtonosni kombinaciji A_j lahko torej preverimo, če vsebuje $A_j - u$ en sam element; če to drži, recimo temu elementu i in vemo, da u-ju ne smemo dodati strupa i, saj bi to pacienta ubilo. Tako lahko z enim prehodom čez vse smrtonosne kombinacije ugotovimo, katere strupe i smemo dodati kombinaciji u; ni nam treba iti pri vsakem možnem i posebej čez vse smrtonosne kombinacije.

$C := \{1, \ldots, n\} - u$;
for $j := 1$ **to** m **do**
  if $|A_j - u| = 1$ **then**
    $C := C - (A_j - u)$;

Ob koncu tega postopka je C množica vseh strupov, ki jih lahko (posamično) dodamo v u, ne da bi pacient pri tem nemudoma umrl. Vsako od uporabljenih operacij na množicah lahko izvedemo v času $O(1)$, če imamo množice u, C in A_j predstavljene z n-bitnimi celimi števili (tako kot v gornjem programu) in če je n dovolj majhen (pri naši nalogi je $n \leq 15$). Malo manj očitno je to le pri preverjanju, ali je $|A_j - u| = i$. Pri tem si lahko pomagamo na primer z naslednjim dejstvom: če je x neka celoštevilska spremenljivka, je x ^ (x − 1) vrednost, ki jo dobimo, če v x ugasnemo najnižji prižgani bit; če je ta vrednost 0, pomeni, da je bil v x prižgan en sam bit. Če za x vzamemo

n-bitno število, ki predstavlja množico $A_j - u$, bomo lahko na ta način preverili, ali je $|A_j - u| = 1$.

Podoben razmislek lahko uporabimo tudi pri ugotavljanju, ali se po dodajanju nekega strupa i v kombinacijo u kakšna kombinacija nemudoma izniči. To se zgodi, če je $B_j \subseteq u \cup \{i\}$. Podobno kot prej za A_j tudi tu lahko predpostavimo, da B_j ni podmnožica u-ja (saj bi mi v tem primeru izničenje kombinacije B_j upoštevali že v prejšnjem koraku in sploh ne bi prišli do u), tako da je $B_j \subseteq u \cup \{i\}$ mogoče le, če je $B_j - u = \{i\}$. Če je ta pogoj izpolnjen, vemo, da bo po dodajanju strupa i v kombinacijo u prišlo do izničenja kombinacije B_j. V enem prehodu čez vse B_j lahko za vsak možni $i \in C$ izračunamo, v katero stanje (recimo v_i) bi prišli, če u-ju dodamo strup i:

for $i := 1$ **to** n **do** $v_i := u \cup \{i\}$;
for $j := 1$ **to** k **do**
  if $|B_j - u| = 1$ **then**
    naj bo i ta edini element množice $B_j - u$; $(\star)$
    $v_i := v_i - B_j$;

Na koncu tega postopka imamo v tabeli $(v_1, \ldots, v_n)$ za vsak možni strup i izračunano stanje, v katero pridemo iz u po dodajanju strupa i in po izvedbi vseh možnih izničenj raznih kombinacij B_j. Zdaj se moramo le še sprehoditi z i od 1 do n in za vsak $i \in C$ upoštevati, da v našem grafu obstaja povezava od u do v_i; torej, če v_i-ja dotlej še nismo obiskali, ga moramo zdaj dodati v vrsto in popraviti.

Vprašanje je še, kako v vrstici $(\star)$ učinkovito priti do i-ja. Prej omenjeni trik z `x ^ (x - 1)` nam lahko pove, če je bil v številu prižgan le en bit, ne more pa nam povedati, kateri bit je to bil. Pač pa si lahko te podatke potabeliramo ob inicializaicji programa: alocirajmo tabelo 2^n števil in jo zapolnimo takole:

for $u := 0$ **to** $2^n - 1$ **do** $T[u] := 0$;
for $i := 1$ **to** n **do** $T[2^{i-1}] := i$;

Pri naši nalogi bi morala biti ta tabela dolga največ 2^{15} elementov, kar ni veliko, če pa nam je to preveč, si lahko pripravimo na primer le tabelo 2^8 elementov in vsako 15-bitno število, za katerega moramo preveriti, kateri je edini prižgani bit v njem, razbijemo na dve 8-bitni (oz. eno 8-bitno in eno 7-bitno).

Opisana izboljšava nam načeloma omogoča, da imamo za vsak u, ki ga med pregledovanjem grafa vzamemo iz vrste, le $O(n + m + k)$ dela, ne več $O(n \cdot (m + k))$ dela.

5. EPS (Emasculated PostScript)

Pomagali si bomo z ukazom `roll`, s katerim lahko nek element z vrha sklada zakopljemo nekam daleč v globino. Če imamo na začetku na skladu $a_1\ a_2\ \ldots\ a_n\ n$, lahko z `roll` premaknemo a_n na dno, tako da bo na vrhu potem a_{n-1}. Paziti pa moramo, da pri tem ne izgubimo n-ja — če bomo zakopali na dno tudi tega, kasneje do njega ne bomo več mogli priti. Zato najprej z `exch` premaknemo n tik pod a_n, tako da bo `roll` res zakopal samo a_n. Ko smo na ta način spravili a_n na želeno mesto na skladu, lahko n zmanjšamo za 1 in vse skupaj ponovimo. Tako nadaljujemo v zanki, dokler ne pade n na 1, takrat pa lahko končamo.

```
                                % a_1 a_2 ... a_n n
zanka:                          % a_n a_{n-1} ... a_{k+1} a_1 a_2 ... a_{k-1} a_k k
  exch                          % a_n a_{n-1} ... a_{k+1} a_1 a_2 ... a_{k-1} k a_k
  push(1) index                 % a_n a_{n-1} ... a_{k+1} a_1 a_2 ... a_{k-1} k a_k k
  push(1) add push(1)           % a_n a_{n-1} ... a_{k+1} a_1 a_2 ... a_{k-1} k a_k (k+1) 1
  roll                          % a_n a_{n-1} ... a_{k+1} a_k a_1 a_2 ... a_{k-1} k
  push(1) sub                   % a_n a_{n-1} ... a_{k+1} a_k a_1 a_2 ... a_{k-1} (k-1)
  % Oz. če zmanjšamo k za 1:    % a_n a_{n-1} ... a_{k+1} a_1 a_2 ... a_k k
  % Če je k > 1, moramo še nadaljevati, sicer lahko končamo.
  dup push(1) gt jnz(zanka)

                                % a_n a_{n-1} ... a_2 a_1 1
  pop                           % a_n a_{n-1} ... a_2 a_1
```

(Komentar v predzadnji vrstici ni čisto točen — če je $n = 1$, dobimo po prvi (in edini) iteraciji na vrhu sklada 0, ne pa 1, rezultat programa pa je tudi takrat pravilen.)

Rekurzivna rešitev v času $O(n \log n)$. Doslej opisana rešitev je kratka, elegantna in za potrebe našega tekmovanja čisto dovolj dobra. Kot zanimivost pa si vendarle oglejmo še malo bolj zapleteno, vendar učinkovitejšo rešitev. Naša dosedanja rešitev velikokrat izvede ukaz roll; v prvi iteraciji zavrti del sklada, ki je dolg $n + 1$ elementov, v drugi iteraciji zavrti del sklada, ki je dolg n elementov in tako naprej. Predstavljamo si lahko, da interpreter za roll nad intervalom n elementov porabi $O(n)$ časa, saj mora pač spremeniti vsebino n celic sklada. Skupaj bi nam torej vsi naši rolli vzeli $(n+1) + n + (n-1) + \ldots + 3 = O(n^2)$ časa. Skupna cena ostalih operacij je $O(n)$ in časovna zahtevnost naše rešitve je tako $O(n^2)$.

Do učinkovitejše rešitve lahko pridemo z rekurzivnim razmislekom. Da bo lažje, za začetek predpostavimo, da je n potenca števila 2: $n = 2^s$. Tabelo 2^s elementov lahko obrnemo v treh korakih:

(1) obrnemo zgornjih 2^{s-1} elementov sklada;
(2) zamenjamo zgornjih in spodnjih 2^{s-1} elementov;
(3) obrnemo zgornjih 2^{s-1} elementov sklada.

V točki (2) uporabimo ukaz roll, ki se bo v tem primeru izvajal $O(2^s)$ časa; točki (1) in (3) pa sta pravzaprav rekurzivna klica, ki izvedeta enak postopek na pol manjši tabeli. Označimo časovno zahtevnost takega postopka s $T(2^s)$; vidimo torej, da je $T(2^s) = 2T(2^{s-1}) + O(2^s)$. Rekurzija se izteče pri $s = 0$, ko nam ni treba narediti ničesar („obračanje“ enega samega elementa). Hitro se lahko prepričamo, da iz te zveze sledi $T(2^s) = O(s \cdot 2^s)$ oz. $T(n) = O(n \log n)$. To je vsekakor veliko bolje od prejšnjega postopka, ki je trajal $O(n^2)$ časa. Nekaj truda bomo imeli le še s tem, da novi postopek zapišemo v našem omejenem postscriptu, saj v njem nimamo niti spremenljivk niti ločenega sklada, na katerega bi lahko rekurzivni klici oddajali svoje podatke.

Za boljše razumevanje si oglejmo, kako bi naš postopek obrnil tabelo osmih elementov. Da bo manj pisanja, pišimo namesto push(x) preprosto x (takšen zapis dopušča tudi originalni jezik postscript):

```
                                % a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
  2 1 roll                      % a_1 a_2 a_3 a_4 a_5 a_6 a_8 a_7
  4 2 roll                      % a_1 a_2 a_3 a_4 a_8 a_7 a_5 a_6
  2 1 roll                      % a_1 a_2 a_3 a_4 a_8 a_7 a_6 a_5
```

```
8 4 roll        % a8 a7 a6 a5 a1 a2 a3 a4
2 1 roll        % a8 a7 a6 a5 a1 a2 a4 a3
4 2 roll        % a8 a7 a6 a5 a4 a3 a1 a2
2 1 roll        % a8 a7 a6 a5 a4 a3 a2 a1
```

V splošnem na ta način za obračanje tabele 2^s elementov izvedemo 2^{s-1} klicev roll. (Mimogrede, klice 2 1 roll lahko seveda zamenjamo z exch, kar pa na časovno zahtevnost nič ne vpliva.) Če bi hoteli obrniti le tabelo štirih števil, bi to dosegli s prvimi tremi od gornjih sedmih stavkov; za obračanje tabele 16 števil pa bi morali dodati 16 8 roll in nato še kopijo gornjih sedmih stavkov. Oglejmo si zaporedje števil, ki jih dobivajo stavki roll kot parametre; prvi parameter je vedno dvakratnik drugega, drugi parameter pa, če ga pogledamo po vrsti od enega klica do naslednjega, tvori takšno zaporedje: $1, 2, 1, 4, 1, 2, 1, \ldots$. Naj bo q_t vrednost parametra pri t-tem klicu:

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	$\cdots$
q_t	1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16	$\cdots$

Vidimo torej, da q_t ni nič drugega kot število, ki ga dobimo, če v dvojiškem zapisu števila t ugasnemo vse prižgane bite razen najnižjega. (Na primer: $t = 10_{10} = 1010_2 \mapsto 0010_2 = 2_{10} = q_t$.) Ker se bo dalo q_t preprosto izračunati neposredno iz t, nam postopka sploh ni treba oblikovati kot rekurzijo, ampak lahko klice izvedemo kar v zanki:

for $t := 1$ **to** $2^s - 1$ **do**
  $2q_t$ q_t roll

Tega ni pretežko zapisati v postscriptu:

```
                          % a1 a2 ... an n za n = 2^s
  1                       % a1 a2 ... an n 1
zanka:                    % a'1 ... a'n n t
  % Elemente smo v komentarjih označili s črticami,
  % ker njihov vrstni red že ni več enak prvotnemu.
  2 copy                  % a'1 ... a'n n t n t
  eq jnz(konec)           % če je n = t, končajmo
  dup                     % a'1 ... a'n n t t
  izračunaj qt            % a'1 ... a'n n t qt
```

S tem, kako izračunati q_t, se bomo ukvarjali malo kasneje. Ko ga enkrat izračunamo, bi si načeloma želeli izvesti $2q_t$ q_t roll, če nam ne bi bili v napoto vrednosti n in t na vrhu sklada (ki pa ju ne smemo izgubiti, saj ju bomo v nadaljevanju še potrebovali). V ta namen je potrebne nekaj zvitosti, podobno kot že pri naši prvotni rešitvi. Da bo manj pisanja, označimo $A = a'_1 \ldots a'_{n-2q_t}$, $B = a'_{n-2q_t+1} \ldots a'_{n-q_t}$ in $C = a'_{n-q_t+1} \ldots a'_n$. Nadaljujemo lahko takole:

```
  dup                     % A B C n t qt qt
  4 1 roll                % A B C qt n t qt
  3 add 2                 % A B C qt n t (qt + 3) 2
  roll                    % A B n t C qt
  dup 2 mul 2 add         % A B n t C qt (2qt + 2)
  exch                    % A B n t C (2qt + 2) qt
  roll                    % C A B n t
  1 add                   % C A B n (t + 1)
```

```
  1 jnz(zanka)
konec:                          % a_n a_{n-1} ... a_1 n n
  pop pop                       % a_n a_{n-1} ... a_1
```

Posvetimo se zdaj vprašanju, kako pri danem t izračunati q_t. Videli smo že, da lahko q_t dobimo tako, da v t ugasnemo vse prižgane bite razen najnižjega. Če bi imeli na voljo operatorje za delo s posameznimi biti, bi bila naloga precej preprosta. V C/C++ in sorodnih jezikih je na primer `t & (t − 1)` število, ki ga dobimo, če v t ugasnemo najnižji prižgani bit; ravno vsi biti, ki po taki operaciji ostanejo prižgani, pa so tisti, ki bi jih mi v resnici radi ugasnili, da nam bo ostal prižgan le najnižji bit: `t & ~(t & (t − 1))` je torej ravno naša iskana vrednost q_t. Primerne operatorje imamo na voljo tudi v originalnem postscriptu, kjer bi lahko q_t računali takole:

```
                                % t
  dup dup                       % t t t
  1 sub                         % t t (t − 1)
  and                           % t (t and (t − 1))
  not                           % t (not (t and (t − 1)))
  and                           % q_t
```

Žal pa operatorjev `and` in `not` v okleščenem postscriptu, kot ga definira naša naloga, ni. Če bi imeli vsaj operatorja za celoštevilsko deljenje in ostanek po deljenju (`idiv` in `mod`), bi lahko za t pogledali, ali je v njem bit i, prižgan, s formulo $(t \operatorname{div} 2^i) \bmod 2$. Ker nimamo niti teh, lahko poskusimo takole: računajmo po vrsti vse večje potence števila, dokler ne pridemo do prve take, ki je večja od t. Če je to recimo 2^k, zdaj vemo, da je $2^k > t \geq 2^{k-1}$. Torej je bit $k-1$ v številu t prižgan in ugasnemo ga lahko tako, da od t odštejemo 2^{k-1}. Zdaj lahko ta razmislek ponovimo pri $k-1$, nato pri $k-2$ in tako naprej. S tem po vrsti ugašamo prižgane bite v t-ju od zgoraj navzgol; ko po enem od teh ugašanj opazimo, da je t zdaj enak 0, vemo, da je bit, ki smo ga ravnokar ugasnili, najnižji prižgani bit v prvotni vrednosti t-ja; to bo torej naš q_t.

Najprej potrebujemo torej zanko, ki računa vse večje potence števila 2. Puščala jih bo kar na skladu, saj bodo prišle prav tudi v naslednji zanki:

```
                                % t
  1 0  3 2 roll                 % 1 0 t
zanka1:                         % 2^0 ... 2^k k t
  dup 3 index                   % 2^0 ... 2^k k t t 2^k
  lt jnz(konec1)                % če je t < 2^k, končajmo
                                % 2^0 ... 2^k k t
  2 index 2 mul                 % 2^0 ... 2^k k t 2^{k+1}
  3 1 roll                      % 2^0 ... 2^k 2^{k+1} k t
  exch 1 add exch               % 2^0 ... 2^k 2^{k+1} (k + 1) t
  1 jnz(zanka1)
konec1:                         % 2^0 ... 2^k k t
```

V naslednji zanki lahko s tako izračunanimi potencami števila 2 ugašujemo prižgane bite v številu t od zgoraj navzdol. Označimo s t_k število, ki ga dobimo, če v t ugasnemo vse bite od vključno k navzgor. Prejšnja zanka je izračunala tako visok k, da je $2^k > t$, torej je bit k v številu t ugasnjen, vsi višji biti prav tako in zato pri tem k-ju velja $t = t_k = t_{k+1} = \ldots$ Z ugašanjem bitov bi radi prišli do največjega takega

k, pri katerem bo $t_k = 0$; takrat bomo vedeli, da je $k + 1$ najnižji prižgani bit v t in je zato $q_t = 2^{k+1}$.

```
zanka2:                     % 2^0 ... 2^k k t_{k+1}
  dup 3 index               % 2^0 ... 2^k k t_{k+1} t_{k+1} 2^k
  lt jnz(preskok)           % če je t_{k+1} < 2^k, je bit k v t ugasnjen in t_k = t_{k+1}
  % Sicer pa je bit k v t prižgan in t_k = t_{k+1} − 2^k.
                            % 2^0 ... 2^k k t_{k+1}
  dup 3 index sub           % 2^0 ... 2^k k t_{k+1} t_k
  dup 0                     % 2^0 ... 2^k k t_{k+1} t_k t_k 0
  eq jnz(konec2)            % če je t_k = 0, končajmo
                            % 2^0 ... 2^k k t_{k+1} t_k
  exch pop                  % 2^0 ... 2^k k t_k
preskok:                    % 2^0 ... 2^k k t_k
  exch 1 sub exch           % 2^0 ... 2^k (k − 1) t_k
  3 2 roll pop              % 2^0 ... 2^{k−1} (k − 1) t_k
  1 jnz(zanka2)             % za naslednjo iteracijo v mislih zmanjšajmo k za 1
konec2:                     % 2^0 ... 2^k k t_{k+1} t_k, vendar t_{k+1} = 2^k = q_t, t_k = 0
  pop 1 index               % 2^0 ... 2^k k q_t k
  3 add 1                   % 2^0 ... 2^k k q_t (k + 3) 1
  roll                      % q_t 2^0 ... 2^k k
```

Zdaj moramo le še pobrisati z vrha sklada preostale potence števila 2 (in število k na vrhu). V ta namen zapišimo še eno zanko:

```
zanka3:                     % q_t 2^0 ... 2^k k
  dup 0 eq jnz(konec3)      % če je k = 0, končajmo
                            % q_t 2^0 ... 2^k k
  1 sub exch pop            % q_t 2^0 ... 2^{k−1} (k − 1)
  1 jnz(zanka3)             % za naslednjo iteracijo v mislih zmanjšajmo k za 1
konec3:                     % q_t 2^0 0
  pop pop                   % q_t
```

Tako smo prišli do postopka, ki tudi v okleščenem postscriptu, kakršen je predpisan v naši nalogi, uspešno izračuna q_t in z njim obrne vrstni red elementov poljubne tabele velikosti $n = 2^s$. Kaj pa, če naš n ni potenca števila 2? Če drugega ne, lahko našo tabelo vedno dopolnimo z nekaj (recimo m) dodatnimi elementi, tako da bo njena dolžina vendarle potenca števila 2, recimo 2^s; po obračanju pa bomo morali te elemente pobrisati. Slednje bo najlažje, če bodo po obračanju pristali na vrhu sklada, nad njimi pa mora biti še število teh elementov. To pa pomeni, da morajo biti pred obračanjem ti elementi na dnu sklada:

$$\overbrace{m \quad [m \textit{ dodatnih elementov}] \quad a_1 \quad a_2 \quad \cdots \quad a_n}^{2^s \textit{ elementov}} \quad 2^s$$

Tukaj torej velja $2^s = m + 1 + n$ in zato $2^s > n$; da si ne bomo nakopavali več dela, kot je nujno potrebno, bomo vzeli najmanjši s, ki ustreza temu pogoju.

```
                            % n
  1                         % n 1
zanka4:                     % n 2^s
  2 copy                    % n 2^s n 2^s
  lt jnz(konec4)            % če je n < 2^s, končajmo
```

```
   2 mul                       % n 2^(s+1)
   1 jnz(zanka4)               % v mislih povečajmo s za 1
konec4:                        % n 2^s in 2^s > n
```

Zdaj lahko izračunamo m in v zanki namečemo na sklad še m dodatnih elementov — uporabili bomo kar ničle:

```
                               % a1 ... an n 2^s
   dup 3 2 roll                % a1 ... an 2^s 2^s n
   sub 1 sub                   % a1 ... an 2^s m za m = 2^s − n − 1
   dup                         % a1 ... an 2^s m m
zanka5:                        % a1 ... an [m − k ničel] 2^s m k
   dup 0 eq jnz(konec5)        % če je k = 0, končajmo
   0 4 1 roll                  % a1 ... an [m − (k − 1) ničel] 2^s m k
   1 sub                       % a1 ... an [m − (k − 1) ničel] 2^s m (k − 1)
   1 jnz(zanka5)               % v mislih zmanjšajmo k za 1
konec5:                        % a1 ... an [m ničel] 2^s m 0
   pop                         % a1 ... an [m ničel] 2^s m
```

Nato bomo m in te ničle z ukazom roll odrinili na dno sklada, da bo dobil sklad obliko, kakršno smo videli zgoraj. Najprej moramo poskrbeti, da bo en izvod m-ja prišel še pod blok m ničel, poleg tega pa mora tik za a_n priti še vrednost 2^s; potem bomo lahko blok m ničel skupaj s številom m pred njimi odrinili na dno sklada:

```
                               % a1 ... an [m ničel] 2^s m
   dup 1 add                   % a1 ... an [m ničel] 2^s m (m + 1)
   2 index 1 add exch          % a1 ... an [m ničel] 2^s m (2^s + 1) (m + 1)
   4 2 roll                    % a1 ... an [m ničel] (2^s + 1) (m + 1) 2^s m
   dup 4 add 2                 % a1 ... an [m ničel] (2^s + 1) (m + 1) 2^s m (m + 4) 2
   roll                        % a1 ... an 2^s m [m ničel] (2^s + 1) (m + 1)
   roll                        % m [m ničel] a1 ... an 2^s
```

Sklad je zdaj v želeni obliki in predstavlja tabelo 2^s števil, ki jo lahko obrnemo po prej opisanem postopku. Po obračanju nam bo na skladu nastalo tole:

$$a_n \quad a_{n-1} \quad \cdots \quad a_1 \quad [m \textit{ dodatnih elementov}] \quad m$$

Potrebujemo torej le še zanko, ki bo pobrisala m in naslednjih m elementov pod njim:

```
                               % an ... a1 [m ničel] m
zanka6:                        % an ... a1 [k ničel] k
   dup 0 eq jnz(konec6)        % če je k = 0, končajmo
   exch pop 1 sub              % an ... a1 [k − 1 ničel] (k − 1)
   1 jnz(zanka6)               % v mislih zmanjšajmo k za 1
konec6:                        % an ... a1 0
   pop                         % an ... a1
```

Kakšna je skupna časovna zahtevnost tako dobljenega postopka za obračanje tabele n števil? Videli smo že, da sami ukazi roll pri obračanju tabele 2^s števil vzamejo $O(s \cdot 2^s)$ časa; pri nas je 2^s najmanjša potenca števila 2, ki je večja od n, zato je $n \geq 2^{s-1}$ in zato $2^s < 2n = O(n)$, torej je $O(s \cdot 2^s) = O(n \log n)$. Tudi za izračun q_t iz t porabimo pri vsakem t-ju $O(\log n)$ časa, kar je pri vseh t-jih skupaj $O(n \log n)$. Izračun s-ja na začetku vzame le $O(\log n)$ časa, dodajanje m ničel vzame $O(m)$ časa,

nato dva ukaza roll s po $O(m)$ in $O(2^s)$ časa; brisanje ničel po obračanju spet vzame $O(m)$ časa; vse te reči so $O(n)$, saj je $m < 2^s < 2n$. Skupna časovna zahtevnost je tako še vedno $O(n \log n)$.

Rešitev v času $O(n)$. Še učinkovitejšo rešitev lahko dobimo z naslednjim razmislekom. Pri naši prvotni rešitvi je bila prvotna tabela (oz. to, kar je še ostalo od nje) ves čas na vrhu sklada, obrnjena tabela pa je nastajala pod njo, pri dnu sklada; in v vsaki iteraciji zanke smo premaknili en element z vrha prvotne tabele na vrh obrnjene tabele (tik pod prvotno tabelo). Težava je bila, da je vsak tak premik drag. Postopek bi postal cenejši, če bi obrnjena tabela nastajala na vrhu sklada, prvotna pa bi ostajala pod njo. Vendar pa zdaj ne moremo več poceni brisati posameznih elementov iz prvotne tabele (ker niso več na vrhu sklada); bolje je, če počakamo, da obračanje tabele končano, takrat pa lahko obrnjeno in prvotno tabelo zamenjamo (da pride prvotna na vrh) in prvotno nato pobrišemo.

```
                        % a_1 a_2 ... a_n n
  2                     % a_1 a_2 ... a_n n 2
zanka1:                 % a_1 a_2 ... a_n a_n a_{n-1} ... a_{n-k+1} n 2(k+1)
  dup 2 index           % a_1 a_2 ... a_n a_n a_{n-1} ... a_{n-k+1} n 2(k+1) 2(k+1) n
  2 mul                 % a_1 a_2 ... a_n a_n a_{n-1} ... a_{n-k+1} n 2(k+1) 2(k+1) 2n
  gt jnz(konec1)        % če je k+1 > n, končajmo
  dup                   % a_1 a_2 ... a_n a_n a_{n-1} ... a_{n-k+1} n 2(k+1) 2(k+1)
  index                 % a_1 a_2 ... a_n a_n a_{n-1} ... a_{n-k+1} n 2(k+1) a_{n-k}
  3 1 roll              % a_1 a_2 ... a_n a_n a_{n-1} ... a_{n-k+1} a_{n-k} n 2(k+1)
  2 add                 % a_1 a_2 ... a_n a_n a_{n-1} ... a_{n-k+1} a_{n-k} n 2(k+2)
  1 jnz(zanka1)         % v mislih povečajmo k za 1
konec1:                 % a_1 a_2 ... a_n a_n a_{n-1} ... a_1 n 2(n+1)
```

Obrnjeno tabelo zdaj imamo. V nadaljevanju jo bomo odrinili na dno sklada, da bo na vrh prišla prvotna tabela, ki je trenutno pod obrnjeno. Najprej pa med obrnjeno in prvotno tabelo vrinimo še en n, da ga bomo imeli pri roki na vrhu sklada po premiku obrnjene tabele na dno.

```
  pop                   % a_1 a_2 ... a_n a_n a_{n-1} ... a_1 n
  dup dup               % a_1 a_2 ... a_n a_n a_{n-1} ... a_1 n n n
  2 add 1               % a_1 a_2 ... a_n a_n a_{n-1} ... a_1 n n (n+2) 1
  roll                  % a_1 a_2 ... a_n n a_n a_{n-1} ... a_1 n
  dup 2 mul 1 add exch  % a_1 a_2 ... a_n n a_n a_{n-1} ... a_1 (2n+1) n
  roll                  % a_n a_{n-1} ... a_1 a_1 a_2 ... a_n n
```

Ostane nam le še brisanje prvotne tabele, ki je zdaj na vrhu sklada:

```
zanka2:                 % a_n a_{n-1} ... a_1 a_1 a_2 ... a_{k-1} a_k k
  dup 0 eq jnz(konec2)  % če je k = 0, končajmo
  1 sub                 % a_n a_{n-1} ... a_1 a_1 a_2 ... a_{k-1} a_k (k-1)
  exch pop              % a_n a_{n-1} ... a_1 a_1 a_2 ... a_{k-1} (k-1)
  1 jnz(zanka2)         % v mislih zmanjšajmo k za 1
konec2:                 % a_n a_{n-1} ... a_1 0
  pop                   % a_n a_{n-1} ... a_1
```

Brisanje bi se dalo elegantno izvesti tudi z značkami in ukazom cleartomark, vendar nas pravila točkovanja pri tej nalogi od uporabe značk odvračajo (prav zato, ker je z njimi nekatere stvari mogoče narediti lažje in preprosteje).

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Vremenoslovje

Spodnja rešitev v tabeli najTemp hrani za vsako postajo najvišjo doslej prebrano temperaturo na tej postaji. Na začetku inicializiramo vse elemente te tabele na −274, saj so naše meritve v stopinjah Celzija in smo lahko prepričani, da tako nizkih temperatur v naravi ni. Če bi se hoteli tej predpostavki izogniti, bi si lahko omislili še eno tabelo, v kateri bi bila za vsako postajo po ena logična vrednost (**bool**), ki bi povedala, ali za to postajo že imamo kakšno meritev ali ne.

```
#include <stdio.h>
#define StPostaj 100
#define MinTemp (−274)

int main()
{
   int najTemp[StPostaj + 1], stPostaje, dan, temperatura;
   FILE *f = fopen("meritve.txt", "rt");
   /* V tabeli najTemp označimo, da za nobeno postajo še nimamo podatkov. */
   for (stPostaje = 1; stPostaje <= StPostaj; stPostaje++)
      najTemp[stPostaje] = MinTemp;
   /* Preberimo datoteko z meritvami. */
   while (3 == fscanf(f, "%d %d %d", &stPostaje, &dan, &temperatura))
      /* Če je trenutna temperatura najvišja znana za to postajo, si jo zapomnimo.*/
      if (temperatura > najTemp[stPostaje])
         najTemp[stPostaje] = temperatura;
   fclose(f);
   /* Izpišimo rezultate. */
   for (stPostaje = 1; stPostaje <= StPostaj; stPostaje++)
   {
      printf("Postaja %d: ", stPostaje);
      if (najTemp[stPostaje] == MinTemp) printf("ni podatka\n");
      else printf("najvisja temperatura je %d\n", najTemp[stPostaje]);
   }
   return 0;
}
```

2. Krajšanje imena

S standardno funkcijo strlen lahko pogledamo, kako dolga sta ime in priimek, nato pa za razne možne okrajšave imena računamo, koliko znakov bi zasedle; prvo dovolj kratko obliko nato tudi zares izpišemo.

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

void Izpisi(const char *ime, const char *priimek, int n)
{
   int dolzImena = strlen(ime), dolzPriimka = strlen(priimek);
   if (n >= dolzImena + 1 + dolzPriimka) printf("%s %s", ime, priimek);
   else if (n >= 3 + dolzPriimka) printf("%c. %s", ime[0], priimek);
   else if (n >= dolzPriimka) printf("%s", priimek);
   else if (n >= 4) printf("%c.%c.", ime[0], priimek[0]);
}
```

3. Registrske številke

Spodnja rešitev najprej prebere vrstico z delno registrsko številko in si jo zapomni v tabeli delna. Nato vrstico za vrstico beremo seznam s konkretnimi registrskimi številkami in imeni lastnikov. Delno številko primerjamo z vsako konkretno številko znak za znakom; če nek znak delne številke ni enak '?', se mora ujemati z istoležnim znakom konkretne registrske številke. Če uspešno pridemo skozi vseh sedem znakov, ne da bi opazili kakšno neujemanje, lahko izpišemo pripadajoče ime in priimek.

```
#include <stdio.h>

int main(int argc, char** argv)
{
  /* Vrstica je lahko dolga 100 znakov, v tabeli s pa moramo
     pustiti še prostor za znak '\0' na koncu niza. */
  char delna[8], s[101]; int i;
  gets(delna); /* Iz prve vrstice preberimo delno registrsko številko. */
  while (gets(s)) /* Berimo vrstico za vrstico. */
  {
    /* Poglejmo, v koliko začetnih znakih se ujemata delna registrska številka
       in konkretna številka iz trenutne vrstice (ki je na začetku niza s). */
    for (i = 0; i < 7; i++)
      if (delna[i] != '?' && delna[i] != s[i]) break;
    /* Če se ujemata v vseh sedmih znakih, izpišimo ime (ki se začne v s[8]). */
    if (i == 7) printf("%s\n", &s[8]);
  }
  return 0;
}
```

V nekaterih programskih jezikih je naloga še lažja, ker nam standardna knjižnica ponuja že pripravljene mehanizme za primerjanje nizov. Spodaj je primer rešitve v pythonu, ki uporablja pythonov modul za delo z regularnimi izrazi; upoštevati mora le to, da v regularnih izrazih za ujemanje s poljubnim znakom poskrbi znak '.', ne pa '?'. Na koncu regularnega izraza dodamo še " (.*)", ki se bo ujel s presledkom in preostankom vrstice, pri čemer bo zaradi oklepajev ta preostanek (ime in priimek) kasneje dostopen prek metode group objekta match, tako da ga bomo lahko izpisali.

```
import sys, re

regIzraz = re.compile(sys.stdin.readline().strip().replace('?', '.') + " (.*)")
for vrstica in sys.stdin:    # Beremo po vrsticah.
  match = regIzraz.match(vrstica.strip())
  if match: print match.group(1)
```

4. Največji nihaj navzdol

Naivna rešitev bi pregledovala zaporedje cen z dvema gnezdenima zankama, eno po i in eno po j, in pri vsakem paru (i, j) (če je $1 \leq i < j \leq n$) računala nihaj navzdol ter si zapomnila največjega. Precej učinkoviteje pa je, če opazimo, da pridejo v poštev za največji nihaj navzdol le tisti pari (i, j), pri katerih smo kupili naš vrednostni papir po najvišji možni ceni (torej najvišji ceni, ki jo je imel kdajkoli pred časom j, ko smo ga prodali). Med pregledovanjem zaporedja cen si torej zapomnimo najvišjo doslej doseženo ceno (spodnji podprogram ima v ta namen spremenljivko najCena), iz nje

pa ne bo težko izračunati največjega nihaja navzdol pri prodaji ob trenutnem času j. Največji doslej opaženi nihaj si zapomnimo v spremenljivki najNihaj.

```
double NajvecjiNihajNavzdol()
{
  int n = StTrenutkov(), i;
  double najCena = Cena(1);
  double najNihaj = 1 − Cena(2) / Cena(1), nihaj;
  for (j = 2; j <= n; i++)
  {
    /* Na tem mestu vsebuje najCena najvišjo ceno pred časom j. */
    nihaj = 1 − Cena(j) / najCena;
    if (nihaj > najNihaj) najNihaj = nihaj;
    if (Cena(j) > najCena) najCena = Cena(j);
  }
  return 100 * najNihaj; /* Vrnimo nihaj v odstotkih. */
}
```

5. Popravilo ograje

Recimo, da so poškodovane deščice $a_1, \ldots, a_k$ (in da je $a_1 < a_2 < \ldots < a_k$). Najbolj leva izmed novih desk se ne sme začeti kasneje kot pri a_1 (saj sicer a_1 ne bo pokrita), ni pa treba, da bi se začela kjerkoli bolj levo kot pri a_1 (saj bi jo v tem primeru lahko premaknili bolj desno in bi imeli še vedno pokrite vse tiste poškodovane deščice, ki so bile pokrite že prej). Torej postavimo najbolj levo novo desko tako, da se začne pri a_1. Poleg a_1 pokrije mogoče ta deska še kaj več poškodovanih deščic; če je pokrila vse, smo končali (in imamo očitno pravo rešitev, saj z manj kot eno desko ograje ne bi mogli popraviti).

Drugače pa naj bo a_i prva izmed še nepokritih poškodovanih deščic. Druga najbolj leva deska se ne sme začeti kasneje kot pri a_i (saj sicer a_i ne bo pokrita), ni pa treba, da bi se začela kjerkoli bolj levo kot pri a_i (saj tam ni nobene take poškodovane deščice, ki je ne bi pokrila že prva deska; drugo bi torej lahko v tem primeru lahko premaknili bolj desno, vse do a_i, in bi imeli še vedno pokrite vse tiste poškodovane deščice, ki so bile pokrite že prej). Torej postavimo drugo desko tako, da se bo začela pri a_i; če smo s tem pokrili že vse poškodovane deščice, lahko končamo, sicer pa razmislek iz tega odstavka ponovimo še za tretjo novo desko in tako naprej.

Vidimo lahko, da iz načina, kako smo razporejali deske, sledi (za vsak d) naslednje: z d deskami ni mogoče pokriti več izmed najbolj levih poškodovanih deščic, kot smo jih s prvimi d deskami pokrili mi. Iz tega pa sledi, da, če smo mi na koncu porabili skupno D desk, se z $D - 1$ deskami ne da pokriti vseh poškodovanih deščic, torej je naša rešitev najboljša možna.

Naloge so sestavili: otoki, strupi, popravilo ograje — Nino Bašić; krajšanje imena, registrske številke — Andrej Bauer; avtocesta — Boris Gašperin; undo — Tomaž Hočevar; soglasniški podnizi — Mitja Lasič in Polona Novak; citati — Jure Leskovec; kovanci, kje sem že to posnel? — Mark Martinec; žaba in lokvanji — Mitja Trampuš; železnica — Miha Vuk; vremenoslovje — Darko Zupanič; smrkci — Anže Žagar; kolikokrat najmanjši, UTF-5, **break** considered harmful, EPS, največji nihaj navzdol — Janez Brank.

1. Nakupovanje

Sprehodimo se z zanko po vseh trgovinah in pri vsaki trgovini poglejmo, koliko bi nas stalo, če bi vse nakupe opravili v tej trgovini. V ta namen uporabimo še eno (vgnezdeno) zanko, ki gre po vseh izdelkih. Dobljeno ceno (spremenljivka cena v spodnjem podprogramu) primerjamo z najnižjo doslej znano ceno (najCena) in če je nova cena še nižja, si zapomnimo tako to ceno kot tudi številko trgovine, pri kateri smo jo dosegli (shranimo jo v najTrgovina).

```
int NajcenejsiNakup()
{
  int cena, najCena, trgovina, najTrgovina, i;
  for (trgovina = 0; trgovina < M; trgovina++) {
    for (cena = 0, i = 0; i < N; i++) cena += Kolicine[i] * Cene[i][trgovina];
    if (trgovina == 0 || cena < najCena)
      najCena = cena, najTrgovina = trgovina; }
  return najTrgovina;
}
```

Gornji program vrača številko trgovine od 0 do $N-1$ — če bi hoteli številčenje trgovin začeti pri 1, bi morali vrniti najTrgovina + 1.

2. Metulji

Ker je število rož in opazovanj razmeroma majhno, lahko vsakič, ko Jožek pogleda oba metulja, za prvega metulja pregledamo vse rože, da ugotovimo, na kateri roži metulj sedi (če sploh na kateri). Če sedi na kakšni roži, si zapomnimo njeno zaporedno številko in preverimo, če na njej sedi tudi drugi metulj. Kako preveriti, ali metulj sedi na roži? Recimo, da ima roža polmer r in središče (x, y), metulj pa je na točki (x_m, y_m); potem metulj sedi na roži natanko tedaj, ko je za manj kot r oddaljen od središča: $\sqrt{(x_m - x)^2 + (y_m - y)^2} < r$. Še lažje pa je, če obe strani neenačbe kvadriramo in se tako izognemo potrebi po korenjenju; preverjali bomo torej $(x_m - x)^2 + (y_m - y)^2 < r^2$.

```
#define MaxR 1000

int main()
{
  double w, h, xr[MaxR], yr[MaxR], rr[MaxR], x1, y1, x2, y2;
  int n, r, i, j, stSkupaj = 0;
  /* Preberimo podatke o rožah. */
  scanf("%lf %lf %d %d", &w, &h, &r, &n);
  for (int i = 0; i < r; i++) scanf("%lf %lf %lf", &xr[i], &yr[i], &rr[i]);
  for (int j = 0; j < n; j++) { /* Preberimo potek poti metuljev. */
    scanf("%lf %lf %lf %lf", &x1, &y1, &x2, &y2);
    for (int i = 0; i < r; i++) /* Poglejmo, če sta oba na roži i. */
      if ((x1 − xr[i]) * (x1 − xr[i]) + (y1 − yr[i]) * (y1 − yr[i]) < rr[i] * rr[i] &&
          (x2 − xr[i]) * (x2 − xr[i]) + (y2 − yr[i]) * (y2 − yr[i]) < rr[i] * rr[i])
        { stSkupaj++; break; }}
  printf("%d\n", stSkupaj); return 0;
}
```

Učinkovitejša rešitev bi lahko uporabila kakšno prostorsko podatkovno strukturo, da bi zmanjšala število rož, ki jih mora pregledati v posamezni iteraciji glavne zanke. Primerna struktura bi bilo na primer štiriško drevo (*quadtree*) ali pa R-drevo.

3. Matura

Naloga je podobna tisti s histogrami iz leta poprej (2006.3.3); namesto točk slike imamo dijake, namesto barvnih odtenkov 0..255 imamo rezultate 0..100, cilj pa ni izravnati histogram, pač pa ga obrniti. Kogar torej zanima daljši razmislek o reševanju takšne naloge, naj si ogleda rešitev naloge s histogrami v biltenu za leto 2006, mi pa bomo tu le na kratko zapisali rešitev tokratne naloge.

Recimo, da smo vhodne podatke že prebrali in imamo zaporedje rezultatov shranjeno v tabeli $a = (a_1, \ldots, a_n)$, pri čemer je n število dijakov. Naj bo b_t število dijakov z rezultatom t (za t od 0 do 100). Dijaki, ki so imeli v vhodni tabeli t točk, po predelavi ne bodo imeli nujno vsi enakega števila točk, bodo pa njihovi novi rezultati tvorili nek strnjen interval na osi 0..100. Naj bo p_t največje število točk, ki jih ima po predelavi kakšen od dijakov, ki so imeli v vhodnem zaporedju t točk; in naj bo q_t število dijakov, ki so imeli v vhodnem zaporedju t točk, po predelavi pa jih imajo p_t. Opisane reči lahko izračunamo takole:

for $t := 0$ **to** 100 **do** $b_t := 0$;
for $i := 1$ **to** n **do** $t := a_i$; $b_t := b_t + 1$;
$p := 101$; $q := 0$;
for $t := 0$ **to** 100 **do**
  while $q \leq 0$ **do** $p := p - 1$; $q := q + b_p$;
  $p_t := p$; $q_t := \min\{q, b_t\}$;
  $q := q - b_t$;
end for;

V glavni zanki torej za naraščajoče t računamo, kam se morajo preslikati rezultati s točno t točkami. Trenutno ciljno vrednost nam hrani spremenljivka p, spremenljivka q pa nam pove, koliko rezultatov s p točkami smemo še ustvariti, da jih na koncu ne bo več kot v vhodni tabeli (namreč b_p).

Ko imamo tabeli p_t in q_t pripravljeni, lahko s še enim prehodom skozi tabelo a popravimo vse rezultate:

for $i := 1$ **to** n **do**
  $t := a_i$;
  while $q_t = 0$ **do** $p_t := p_t - 1; q_t := b_{p_t}$;
  $a_i := p_t$; $q_t := q_t - 1$;
end for;

Opisana rešitev ima časovno zahtevnost $O(n + T)$, če je T največje možno število točk (v našem primeru je $T = 100$).

4. Vojna

Postopek je enostaven — iz vsakega kupa poberemo eno karto (pazimo, da zaključimo delo, če na kateremkoli kupu zmanjka kart), poiščemo večjo in obe odložimo na

ustrezni kup. Težave nastopijo pri vojni — v tem primeru moramo iz vsakega kupa potegniti karto, ki bi jo v igri zaprto odložili na mizo („slepa“ karta), potem pa še karto, ki igra proti nasprotnikovi karti („igralna“ karta); nato primerjamo obe „igralni“ karti in vseh šest odložimo na ustrezni kup. Kaj pa, če sta obe „igralni“ karti enaki? V tem primeru moramo ponoviti vojno (in morda še enkrat, in še enkrat . . .).

Pomagamo si tako, da ugotovimo, da nas — ko enkrat nastopi vojna — originalni in „slepi“ karti ne zanimata več, zato ju preprosto porinemo na pomožni kup. Če sta „igralni“ karti enaki, ponovimo postopek — „igralni“ in novi „slepi“ karti odložimo na pomožni kup ter poberemo dve novi „igralni“ karti. Ko sta enkrat „igralni“ karti različni, preložimo pomožni kup in obe aktivni „igralni“ karti na ustrezen igralni kup.

Postopek lahko še malo poenostavimo, tako da vsako iz kupa pobrano karto takoj odložimo na pomožni kup. Ko določimo zmagovalca poteze, le še preložimo pomožni kup na zmagovalčev kup.

Naloga sprašuje tudi po tem, kdaj je še mogoč neodločen izid. Do tega pride v primeru, če med vojno zmanjka kart obema igralcema (ne le enemu od njiju).

```
/* Vzame po eno karto iz vsakega kupa. Če na kateremkoli kupu (ali pa na obeh)
   zmanjka kart, vrne število 0, 1 ali 2, ki pove izid igre;
   sicer pa vrne −1, pobrani karti pa odloži na pomožni kup. */
int IgrajDveKarti(KupKart *igralec1, KupKart *igralec2,
                  KupKart *pomozniKup, Karta *karta1, Karta *karta2)
{
  bool ok1 = PoberiKarto(igralec1, karta1);
  bool ok2 = PoberiKarto(igralec2, karta2);
  if (! ok1) return ok2 ? 2 : 0;
  else if (! ok2) return 1;
  OdloziKarto(pomozniKup, *karta1);
  OdloziKarto(pomozniKup, *karta2);
  return −1;
}

/* Preloži vse karte iz enega kupa na drugega. */
void PreloziKup(KupKart *izKupa, KupKart *naKup)
{
  Karta karta;
  while (PoberiKarto(izKupa, &karta)) OdloziKarto(naKup, karta);
}

int OdigrajVojno(KupKart *igralec1, KupKart *igralec2, int najvecPotez)
{
  KupKart pomozniKup;
  Karta karta1, karta2;
  int poteza, rezultat = 0;
  pomozniKup.vrh = 0; pomozniKup.stKart = 0;

  for (poteza = 0; poteza < najvecPotez; poteza++) {
    rezultat = IgrajDveKarti(igralec1, igralec2, &pomozniKup, &karta1, &karta2);
    if (rezultat >= 0) return rezultat;
    while (karta1 == karta2) { /* Odigrajmo vojno. */
      /* Poberimo „slepi“ karti. */
      rezultat = IgrajDveKarti(igralec1, igralec2, &pomozniKup, &karta1, &karta2);
      if (rezultat >= 0) return rezultat;
```

```
      /* Poberimo „igralni“ karti. */
      rezultat = IgrajDveKarti(igralec1, igralec2, &pomozniKup, &karta1, &karta2);
      if (rezultat >= 0) return rezultat; }
    PreloziKup(&pomozniKup, karta1 < karta2 ? igralec2 : igralec1); }
  /* Možno je, da po natanko najvecPotez potezah zmanjka kart na enem kupu. */
  rezultat = IgrajDveKarti(igralec1, igralec2, &pomozniKup, &karta1, &karta2);
  return rezultat < 0 ? 0 : rezultat;
}
```

5. Disk

Pomagamo si lahko z dvema gnezdenima zankama; zunanja ponavlja poskuse branja, dokler eden od njih ne uspe ali pa zmanjka časa. Pri vsakem poskusu branja najprej preverimo, če je disk trenutno dovolj miren (če ni, bomo čez milisekundo poskusili znova), nato sprožimo branje (klic ZacniBratiSektor) in nato v notranji zanki čakamo, da bo branje končano (funkcija Prebrano). Med čakanjem po enkrat na vsako milisekundo preverimo pospešek in če je le-ta previsok, postavimo spremenljivko napaka na **true**. Ko se notranja zanka konča, nam torej napaka pove, če lahko prebranim podatkom zaupamo (pri napaka == **false**) ali ne; v slednjem primeru bomo pač v naslednji iteraciji glavne zanke poskusili znova. Po vsakem čakanju za 1 ms povečamo števec cas, tako da lahko kasneje preverjamo, če se ni naše branje zavleklo že nad M milisekund.

```
bool PreberiSektor(int StevilkaSektorja, void* Medpomnilnik)
{
  int cas = 0; bool napaka;
  do {
    /* Počakajmo, da se disk umiri, če je treba. */
    napaka = (Pospesek() > N);
    if (napaka) { Cakaj(); cas++; continue; }
    /* Začnimo z branjem. */
    ZacniBratiSektor(StevilkaSektorja, Medpomnilnik);
    /* Počakajmo, da se branje konča, med tem pa spremljajmo pospeške. */
    do {
      if (Pospesek() > N) { napaka = true; break; }
      Cakaj(); cas++;
    } while (! Prebrano() && ! napaka);
    /* Do napake bi lahko prišlo tudi v zadnji milisekundi čakanja. */
    if (Pospesek() > N) napaka = true;
  } while (cas < M && napaka);
  return ! napaka;
}
```

6. Prepovedani znaki

Preprosta, vendar neučinkovita rešitev je, da uporabimo dve gnezdeni zanki; z zunanjo zanko se sprehajamo po znakih besedila, vsakega od njih pa z notranjo zanko primerjamo s prepovedanimi znaki; če ga najdemo med njimi, čezenj zapišemo nadomestni znak. Kot namiguje že besedilo naloge, je časovna zahtevnost te rešitve $O(nm)$, če je n dolžina besedila, m pa število prepovedanih znakov.

```
void NadomestiPrepovedaneZnake(char *Besedilo, const char *PrepovedaniZnaki,
                               char NadomestniZnak)
{
  char *p; const char *q;
  for (p = Besedilo; *p; p++) {
    for (q = PrepovedaniZnaki; *q && *q != *p; q++) ;
    if (*p == *q) *p = NadomestniZnak; }
}
```

Učinkoviteje gre, če si pripravimo tabelo logičnih vrednosti, ki nam za vsak možen znak pove, ali je prepovedan ali ne. Po nizu prepovedanih znakov se moramo tako sprehoditi le enkrat, na začetku obdelave, ne pa po enkrat za vsak znak besedila. Pri vsakem znaku besedila lahko preprosto pogledamo v tabelo in takoj vidimo, če je prepovedan; časovna zahtevnost rešitve je le še $O(n + m)$.

```
void NadomestiPrepovedaneZnake(char *Besedilo, const char *PrepovedaniZnaki,
                               char NadomestniZnak)
{
  char *p; const char *q; int c; bool prepovedan[256];
  for (c = 0; c < 256; c++) prepovedan[c] = false;
  for (q = PrepovedaniZnaki; *q; q++) prepovedan[(unsigned char) *q] = true;
  for (p = Besedilo; *p; p++)
    if (prepovedan[(unsigned char) *p]) *p = NadomestniZnak;
}
```

(Opomba: pretvorbe v **unsigned char** so koristne zato, ker jezik C ne predpisuje, ali je tip **char** sam po sebi predznačen ali ne; to je prepuščeno prevajalnikom in pri marsikaterem je **char** predznačen. V takem primeru bi se znaki z vrednostmi od 128 do 255 obravnavali kot cela števila z vrednostmi od −128 do −1 in ko bi z njimi naslavljali tabelo prepovedan, bi program v resnici pisal po pomnilniku zunaj te tabele; zaradi tega bi se lahko sesul ali pa vsaj napačno deloval.)

Namesto tabele 256 vrednosti **bool** si lahko omislimo tudi bitno karto, v kateri vsakemu znaku pripada le en bit. V spodnji rešitvi smo bitno karto predstavili kot tabelo nepredznačenih celih števil, od katerih je vsako dolgo po IntBits bitov.[5] Znaku c pripada bit c % IntBits v celici z indeksom c / IntBits.

```
#define IntBits (8 * sizeof(unsigned int))
#define Dolzina (256 / IntBits)

void NadomestiPrepovedaneZnake(char *Besedilo, const char *PrepovedaniZnaki,
                               char NadomestniZnak)
{
  char *p; const char *q; int c;
  unsigned int prepovedan[Dolzina];

  /* Označimo v bitni karti prepovedane znake. */
  for (c = 0; c < Dolzina; c++) prepovedan[c] = 0;
  for (q = PrepovedaniZnaki; *q; q++) {
    c = (unsigned char) *q;
    prepovedan[c / IntBits] |= 1u << (c % IntBits); }
```

[5] Če nas slučajno skrbi, da 256 ni večkratnik IntBits, bi morali vrednost Dolzina definirati z zaokrožanjem navzgor, sicer bo tabela za en element prekratka: (256 + IntBits − 1) / IntBits. V praksi so dandanes cela števila ponavadi 32-bitna, tako da je ta skrb odveč ($256 = 8 \cdot 32$).

```
  /* Preglejmo besedilo. */
  for (p = Besedilo; *p; p++) {
    c = (unsigned char) *p;
    if (prepovedan[c / IntBits] & (1u << (c % IntBits)))
      *p = NadomestniZnak; }
}
```

Naloga sprašuje tudi, kako bi problem rešili za 16-bitne znake (namesto 8-bitnih). Prva od zgornjih rešitev bi delovala brez velikih sprememb tudi v tem primeru, le namesto tipa **char** bi morali pač uporabiti **wchar_t**. Pri drugi rešitvi pa je težava predvsem v tem, da bi tabela prepovedan potrebovala $2^{16} = 65536$ elementov namesto le 256 elementov. Inicializacija tolikšne tabele bi vzela tudi nekaj več časa; če je besedilo dovolj dolgo, se to še vseeno splača, pri kratkih besedilih pa bi bilo hitreje uporabiti kar preprostejši postopek iz prve rešitve. Inicializacija postane mogoče malo cenejša, če uporabimo bitno karto, ki bi morala biti zdaj dolga 65536 bitov oz. 8 KB.

Pri 16-bitnih znakih so rešitve s tabelo **bool**ov ali pa z bitno karto videti še nekako za silo sprejemljive; pri dovolj velikem kodnem prostoru pa vendarle postanejo nevzdržne in naloga nas v bistvu poskuša napeljati k razmišljanju o rešitvah, ki niso posebej občutljive na velikost kodnega prostora.[6]

Primer take rešitve je, da tabelo prepovedanih znakov uredimo naraščajoče in nato z bisekcijo za vsak znak besedila preverjamo, ali se nahaja v tabeli prepovedanih znakov ali ne. Takšna rešitev bi porabila $O(m \log m)$ časa za urejanje in nato $O(logm)$ za bisekcijo pri posameznem znaku besedila, skupaj torej $O((n+m) \log m)$. Še ena možnost je, da prepovedane znake shranimo v razpršeno tabelo (*hash table*), kar nam bo načeloma vzelo $O(m)$ časa in $O(m)$ pomnilnika, nato pa bomo lahko za vsak znak besedila v $O(1)$ časa preverili, ali je prepovedan ali ne.

Še ena možnost pa je „redka“ (*sparse*) tabela (ali pa bitna karta) prepovedanih znakov. V standardih, kot je Unicode, so znaki razporejeni približno tako, da znaki, ki se pogosto uporabljajo skupaj (npr. ker so iz iste pisave), ležijo v istem predelu kodnega prostora. Torej je precej verjetno, da tudi naši prepovedani znaki niso razpršeni enakomerno po celem kodnem prostoru, pač pa le v nekem manjšem delu tega prostora. Kodni prostor 2^{16} znakov razdelimo v mislih na manjše bloke, dolge po recimo 128 znakov (tudi Unicode običajno dodeljuje kodna mesta raznim skupinam znakov v blokih, katerih velikost je večkratnik 128). V tabeli prepovedanih znakov (spremenljivka tabela v spodnji rešitvi) hranimo podatke le za tiste bloke, ki vsebujejo vsaj en prepovedan znak; to, ali nek blok vsebuje kakšen prepovedan znak ali ne, nam pove manjša tabela jeBlok, poleg nje pa imamo še tabelo prepovedan, v kateri je za vsak po en kazalec, ki kaže na začetek podatkov za ta blok v tabeli tabela.

```
#include <stddef.h>
#include <wchar.h>

#define LgBlok 7
#define Blok (1 << LgBlok)
#define StBlokov (1 << (16 − LgBlok))
```

[6] Pomislimo na primer na polno različico standarda Unicode, pri katerem kodni prostor ni velik le 2^{16} znakov, pač pa kar $17 \cdot 2^{16}$ znakov, kar je približno 1,1 milijona. Bitna karta s toliko biti bi bila dolga že kar 136 KB.

```
void NadomestiPrepovedaneZnake3(wchar_t *Besedilo, const wchar_t *PrepovedaniZnaki,
                                wchar_t NadomestniZnak)
{
   bool *prepovedan[StBlokov], *tabela, jeBlok[StBlokov], *blok;
   wchar_t *p; const wchar_t *q;
   int c, nBlokov = 0;

   /* Kateri bloki vsebujejo kak prepovedan znak in koliko jih je? */
   for (c = 0; c < StBlokov; c++) jeBlok[c] = false;
   for (q = PrepovedaniZnaki; *q; q++) jeBlok[*q >> LgBlok] = true;
   for (c = 0; c < StBlokov; c++) if (jeBlok[c]) nBlokov++;

   /* Alocirajmo pomnilnik in v tabeli prepovedan pripravimo kazalce na bloke. */
   tabela = (bool *) malloc(sizeof(bool) * Blok * nBlokov);
   nBlokov = 0;
   for (c = 0; c < StBlokov; c++)
      if (jeBlok[c]) prepovedan[c] = tabela + Blok * nBlokov++;
      else prepovedan[c] = 0;

   /* Označimo prepovedane znake. */
   for (c = 0; c < nBlokov * Blok; c++) tabela[c] = false;
   for (q = PrepovedaniZnaki; *q; q++)
      prepovedan[*q >> LgBlok][*q & ((1 << LgBlok) - 1)] = true;

   /* Preglejmo besedilo. */
   for (p = Besedilo; *p; p++) {
      blok = prepovedan[*p >> LgBlok];
      if (blok && blok[*p & (Blok - 1)]) *p = NadomestniZnak; }

   free(tabela);
}
```

7. Gugl'bot

Spodnja rešitev si pomaga s pomožno tabelo Ime, v kateri je prostora le za 255 znakov (in ničelni znak na koncu). Vanjo kopiramo vhodni URL znak po znak, pri tem pa prepovedane znake zamenjamo s podčrtaji. Spremenljivka n šteje, koliko znakov smo že skopirali; ustavimo se najkasneje po 255 znakih, če ni že prej konec niza URL. Nato v zanki preverjamo, če datoteka z imenom Ime že obstaja; če je potrebno, dodamo na konec imena niz oblike _1, _2 ipd. Pri tem nam pride prav spremenljivka n, torej dolžina prvotnega imena: podaljšek _1 skopiramo v Ime od indeksa n naprej, razen če bi ime s tem postalo daljše od 255 znakov. Če moramo preizkusiti več podaljškov, lahko z vsakim naslednjim povozimo prejšnjega, saj številka v podaljšku le narašča in je zato naslednji podaljšek dolg vsaj toliko kot prejšnji.

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

#define MaxDolz 255

char* PoimenujDatoteko(const char *URL, const char *Direktorij)
{
   char Ime[MaxDolz + 1], s[20], c, *Rezultat;
   int n, m, ZapSt;

   /* Skopirajmo URL v Ime in pri tem zamenjajmo prepovedane znake
      s podčrtaji. Če je URL predolg, se bomo ustavili po MaxDolz znakih. */
   for (n = 0; URL[n] && n < MaxDolz; n++)
```

```
  {
    c = URL[n];
    if (c == '\\' || c == '/' || c == ':' || c == '*' || c == '?' ||
        c == '<' || c == '>' || c == '|') c = '_';
    Ime[n] = c;
  }
  /* Preverimo, če datoteka s tem imenom že obstaja; če bo treba,
     bomo dodali podaljšek _1, _2 ipd. */
  Ime[n] = 0; ZapSt = 0;
  while (DatotekaObstaja(Ime, Direktorij))
  {
    sprintf(s, "_%d", ++ZapSt);
    m = strlen(s);
    /* Podaljšek je dolg m znakov, prvotno ime pa n. Če bi bil
       stik tega dvojega predolg, povozimo zadnjih nekaj znakov imena. */
    strcpy(Ime + (n + m > MaxDolz ? MaxDolz - m : n), s);
  }
  /* Skopirajmo ime v novo, dinamično alocirano tabelo, ki jo
     bomo lahko vrnili klicatelju kot rezultat funkcije. */
  n = strlen(Ime); Rezultat = (char *) malloc(n + 1);
  strcpy(Rezultat, Ime); return Rezultat;
}
```

8. Klepetalnica

Pri pošiljanju sporočilom dodelimo zaporedne številke. Pri sprejemu preverimo, če smo uporabniku dostavili že vsa sporočila, ki imajo nižjo številko, kot je številka pravkar prejetega sporočila (številko zadnjega sporočila, ki smo ga dostavili uporabniku, hranimo v globalni spremenljivki zadnjeDostavljeno). Če ne, si sporočilo le zapomnimo; sicer pa sporočilo prikažemo, nato pa še pogledamo, če je med zapomnjenimi sporočili še kakšno, ki ga tudi že lahko dostavimo.

Prvi del, pošiljanje, je enostaven:

```
typedef struct {
  char vsebina[100];
  int stevilka;
} Sporocilo;

int zadnjePoslano = 0;

void GumbPritisnjen(const char *niz)
{
  Sporocilo sporocilo;
  strcpy(sporocilo.vsebina, niz);
  sporocilo.stevilka = ++zadnjePoslano;
  Poslji(sporocilo);
}
```

Sprejem sporočila je malo bolj zapleten; zaenkrat odmislimo vprašanje, kako hraniti sporočila, ki smo jih že sprejeli, nismo pa jih še dostavili uporabniku. Predpostavimo, da obstajata za delo z množico shranjenih nedostavljenih sporočil dva podprograma: Shrani(sporocilo) in in Preberi(stevilka, sporocilo); slednji naj deluje takole: če je v shrambi sporočilo s številko stevilka, ga skopira v spremenljivko *sporocilo, pobriše iz shrambe in vrne **true**; sicer pa vrne **false**.

```
int zadnjeDostavljeno = 0;

void Sprejem(Sporocilo sporocilo)
{
  /* Če prejetega sporočila ne moremo dostaviti, ga shranimo. */
  if (sporocilo.stevilka > zadnjeDostavljeno + 1) { Shrani(sporocilo); return; }
  /* Sicer ga prikažemo, nato pa lahko mogoče prikažemo še več naslednjih sporočil,
     če jih že imamo v shrambi; prikazana sporočila tudi pobrišemo iz shrambe. */
  do { Prikazi(sporocilo.vsebina); ++zadnjeDostavljeno; }
  while (Preberi(zadnjeDostavljeno + 1, &sporocilo));
}
```

Ostane nam še vprašanje, kako organizirati shrambo nedostavljenih sporočil. Ena možnost je na primer seznam s kazalci (linked list), v katerem bi lahko imeli sporočila urejena naraščajoče po številki. Podprogram Sprejem, ko kliče Preberi, vedno sprašuje po sporočilu, ki ima vsaj tako nizko številko kot katero koli drugo še nedostavljeno sporočilo; torej ga bomo, če ga sploh imamo v seznamu, našli na samem začetku seznama. Zato je podprogram Preberi v tem primeru kratek in učinkovit, saj mora pogledati le prvi element seznama in ga po potrebi pobrisati. Več dela pa bi imel v tem primeru podprogram Shrani, ki bi se moral zapeljati po seznamu in primerjati številko novega sporočila s številkami vseh obstoječih sporočil, da bi ugotovil, kam ga je treba vriniti (da bo seznam ostal urejen). Dodajanje ima torej v tem primeru časovno zahtevnost $O(n)$, brisanje pa $O(1)$, če je n trenutno število nedostavljenih sporočil v shrambi.[7]

```
typedef struct Celica_ { Sporocilo s; struct Celica_* nasl; } Celica;
Celica *glava = 0, *rep = 0;

void Shrani(Sporocilo sporocilo)
{
  /* Poglejmo, kam je treba vriniti novo sporočilo. */
  Celica *prej = 0, *tren = glava, *nova;
  while (tren && tren->s.stevilka < sporocilo.stevilka)
    prej = tren, tren = tren->nasl;
  if (tren && tren->s.stevilka == sporocilo.stevilka) /* Čudno, dvakraten prejem. */
    { tren->s = sporocilo; return; }
  /* Pripravimo novo celico seznama in jo dodajmo. */
  nova = (Celica *) malloc(sizeof(Celica));
  nova->s = sporocilo;
  if (prej) nova->nasl = prej->nasl, prej->nasl = nova;
  else nova->nasl = glava, glava = nova;
  if (tren == rep) rep = nova;
}

bool Preberi(int stevilka, Sporocilo* sporocilo)
{
```

[7] Zanimivo vprašanje je, kaj narediti, če seznam postane predolg. Lahko se sprijaznimo z dejstvom, da se občasno kakšno sporočilo tudi izgubi; če postane seznam predolg, uporabniku dostavimo najstarejše še nedostavljeno sporočilo in ga pobrišimo iz seznama. V tem primeru se lahko zgodi, da do nas nekoč kasneje vendarle pride sporočilo s številko, nižjo od zadnjeDostavljeno. Takega sicer mogoče ne bi škodilo prikazati uporabniku, ne smemo pa ob tem povečati zadnjeDostavljeno. Če bi se hoteli bolj resno zavarovati pred izgubljanjem sporočil, bi morali v naš protokol uvesti še potrjevanje prejema, pošiljatelj pa bi moral sporočila po določenem času ponoviti, če še ni dobil potrditve, da jih je prejemnik dobil. S takšnimi rečmi se na primer ukvarja protokol TCP.

```
    Celica *nasl;
    if (! glava || glava->s.stevilka != stevilka) return false;
    nasl = glava->nasl; *sporocilo = glava->s;
    free(glava); glava = nasl; if (! nasl) rep = 0;
    return true;
}
```

Druga možnost je na primer neurejen seznam, pri katerem je shranjevanje poceni (novo sporočilo dodamo npr. vedno na konec seznama), je pa zato dražje brisanje, ker se moramo sprehoditi po celem seznamu, da ugotovimo, ali je iskano sporočilo v njem ali ne.

Uporabimo lahko tudi kopico (*heap*); to je drevesasta podatkovna struktura, v kateri so sporočila z manjšimi številkami bližje korenu, prav v korenu drevesa pa je sporočilo z najmanjšo številko. Tako brisanje kot dodajanje sporočila v kopico imata časovno zahtevnost $O(\log n)$.

Še ena možnost je razpršena tabela (*hash table*), v kateri kot ključ uporabimo številko sporočila. Če se ne bo dogajalo kaj patološkega, bosta tako branje kot dodajanje sporočila zahtevali le $O(1)$ časa.

Možna je tudi rešitev z navadno tabelo (*array*); recimo, da bodo sporočila v njej ne le urejena po številkah, ampak bomo med njimi tudi pustili prazen prostor za sporočila, ki jih še čakamo. Če je recimo v prvi celici sporočilo s številko 123, vemo, da sodi sporočilo s številko 126 v četrto celico tabele. Tako je shranjevanje novega elementa hitro; da pa bo hitro tudi brisanje prvega elementa, ostalih v takem primeru ne bomo selili za eno mesto nazaj po tabeli, ampak si le zapomnimo, da se vsebina tabele začne eno celico kasneje kot prej. Tabelo torej pravzaprav uporabljamo kot krožno tabelo (*ring buffer*). Če so vse celice zasedene, lahko podatke preselimo v novo, dvakrat večjo tabelo ali pa začnemo stara sporočila pozabljati. Slabost te rešitve je potrata prostora, če so med prejetimi sporočili velike vrzeli (mi pa v naši tabeli vseeno hranimo prostor za vsa še ne prejeta sporočila).

9. Komparatorji

(e) Če so imena in priimki sestavljeni le iz črk, lahko primeren ključ dobimo tako, da zapišemo najprej priimek, nato presledek in na koncu ime. Leksikografsko urejanje po tem bo zagotovilo, da bodo prišli skupaj zapisi z enakim priimkom in da bodo ti potem med seboj urejeni leksikografsko po imenu. Zapisi z različnimi priimki pa bodo med seboj tudi urejeni leksikografsko po priimku. O tem se lahko prepričamo takole: dva različna priimka, s in t, se v prvih nekaj znakih sicer mogoče ujemata, prej ali slej pa nastopi neujemanje $s[i] \neq t[i]$ ali pa je eden od njiju podaljšek drugega, npr. $s = tu$. V prvem primeru bo neujemanje na istem mestu nastopilo tudi pri ključih, v drugem primeru pa bo imel ključ zapisa s priimkom t kot svoj $(|t| + 1)$-vi znak presledek, ključ zapisa s priimkom s pa bo imel na tem mestu $(|t| + 1)$-vo črko niza s (oz. prvo črko niza u); ker je presledek leksikografsko pred vsemi črkami, bo ključ zapisa s priimkom t leksikografsko pred ključem zapisa s priimkom s, to pa je tudi tisto, kar smo si želeli. (Če bi bil t podaljšek niza s, ne pa obratno, bi bil razmislek seveda čisto analogen.)

```
char *Kljuc(char *Z)
{
```

```
   int dolzina = strlen(Z);
   char *K = (char*) malloc(dolzina + 1), *k, *z;
   z = Z; while (*z != ' ') z++;  z++;        /* Postavimo z na začetek priimka. */
   k = K; while (*z) *k++ = *z++;             /* Skopirajmo priimek. */
   *k++ = ' ';                                /* Dodajmo presledek. */
   z = Z; while (*z != ' ') *k++ = *z++;      /* Skopirajmo ime. */
   *k = 0; return K;
}
```

Recimo pa, da se lahko v imenu in priimku pojavljajo poljubni znaki, kar pomeni, da ju v ključu ne moremo preprosto ločiti s presledkom (ali kakšnim drugim posebnim znakom), saj se lahko ta pojavlja tudi kot običajen del imena ali priimka. Zdaj lahko problem rešimo takole: pred vsako črko imena vrinimo znak `a`, pred vsako črko priimka pa znak `b`, nato pa staknimo skupaj priimek in ime (v tem vrstnem redu). O tem, da bomo s tako definiranimi ključi zagotovili zahtevani vrstni red, se lahko prepričamo podobno kot v prejšnjem odstavku. V primeru, ko je en priimek podaljšek drugega, npr. $s = tu$, se bosta ključa ujemala v prvih $2|t|$ znakih, nato pa v ključu zapisa s priimkom t prišel znak `a` (ki smo ga vrinili pred prvo črko imena), v ključu zapisa s priimkom s pa znak `b` (ker se tu še nadaljuje priimek, kjer pa je pred vsako črko vrinjen `b`); to bo zagotovilo, da bo ključ zapisa s priimkom t leksikografsko pred ključem zapisa s priimkom s.

```
char *Kljuc(char *ime, char *priimek)
{
   int d1 = strlen(ime), d2 = strlen(priimek), i;
   char *K = (char*) malloc(2 * (d1 + d2) + 1), *p;
   for (i = 0, p = K; i < d2; i++) { *p++ = 'b'; *p++ = priimek[i]; }
   for (i = 0; i < d1; i++) { *p++ = 'a'; *p++ = ime[i]; }
   *p = 0; return K;
}
```

(*f*) Naj bo x naše vhodno celo število. Prvi znak ključa naj bo `0`, če je $x < 0$, sicer pa `1`. To bo zagotovilo, da pridejo pri urejanju vsa negativna števila pred vsa nenegativna. Recimo zdaj, da je $x \geq 0$; zapišimo ga z nizom brez odvečnih ničel na začetku. Daljši nizi zdaj vsekakor predstavljajo večja števila kot krajši nizi (na primer `100` in `99`); med nizi enake dolžine pa predstavljajo večja števila tisti, ki so tudi v leksikografskem vrstnem redu kasnejši (na primer `17546` in `17532`). Slednje je posledica dejstva, da pride znak `9` v leksikografskem vrstnem redu kasneje kot `8`, ta pa kasneje kot `7` in tako naprej. To, kako poskrbeti, da bodo nizi urejeni najprej po dolžini in enako dolgi nato še leksikografsko, pa smo videli že pri podnalogi (*c*) in lahko njeno rešitev uporabimo tudi tu.

Ostanejo nam še negativna števila. Če je naš x manjši od 0, vzemimo njegovo absolutno vrednost; z enakim razmislekom kot v gornjem odstavku bi znali $|x|$ predelati v ključ, ki bi zagotovil, da bodo naša števila urejena naraščajoče po $|x|$. Toda negativni x morajo biti v resnici urejeni padajoče po $|x|$, tako da pride npr. -100 pred -99. Vrstni red iz prejšnjega odstavka moramo torej za negativna števila obrniti; to pa je prav tak problem, kakršnega smo reševali že pri podnalogi (*a*), le da smo imeli tam nize črk, tu pa nize števk.

```
char *Kljuc(char *Z)
{
```

```
  bool minus = (*Z == '-');
  char *z = Z, *K;
  int i, dolzina = 0;

  /* Preskočimo predznak in vodilne ničle. */
  if (*z == '+' || *z == '-') z++;
  while (*z == '0') z++;
  /* Izračunajmo dolžino preostanka števila in pripravimo pomnilnik za ključ. */
  while (z[dolzina]) dolzina++;
  K = (char*) malloc(2 * dolzina + 3);
  /* Prvi znak ključa loči negativna števila od ostalih. */
  K[0] = (minus ? '0' : '1');
  /* Sledijo znaki, ki poskrbijo za urejanje po dolžini. */
  for (i = 0; i < dolzina; i++) K[1 + i] = '1';
  K[1 + dolzina] = '0';
  /* Sledi kopija števk vhodnega števila, ki poskrbi za
     urejanje enako dolgih števil po velikosti. */
  for (i = 0; i < dolzina; i++) K[2 + dolzina + i] = z[i];
  K[2 * dolzina + 2] = 0; /* Znak za konec niza. */
  /* Vrstni red pri negativnih številih obrnemo, da bodo urejena padajoče po
     absolutni vrednosti (in s tem naraščajoče po svoji dejanski vrednosti). */
  if (minus) for (i = 1; i < 2 * dolzina + 2; i++) K[i] = '9' - (K[i] - '0');

  return K;
}
```

Opisana rešitev daje ključe, ki so približno dvakrat daljši od vhodnega števila; če imamo na vhodu recimo n-mestno število in če odmislimo predznak, je v ključu iz n števk vhodnega števila nastalo $2n+1$ znakov. Dalo bi se narediti tudi krajše ključe. Niz n enic in ene ničle, ki smo ga v ključu vrinili pred vhodno število, je pravzaprav eniška predstavitev števila n, torej dolžine vhodnega števila. Namesto v eniškem zapisu bi lahko n predstavili tudi v desetiškem. Ključi morajo biti taki, da bodo poskrbeli za urejanje vhodnih števil najprej po dolžini (torej po n), tistih z enako dolžino pa leksikografsko (oboje skupaj bo zagotovilo, da bodo števila urejena po svoji vrednosti, kot si tudi želimo). Ker imamo n zdaj zapisan v desetiškem zapisu in hočemo naše zapise urediti najprej po n, smo se znašli pred enakim problemom kot v prvotni nalogi. Izberimo si nek znak, ki je leksikografsko za vsemi števkami, na primer strešico `^`); vrinimo pred n toliko strešic, kolikor je števk v n-ju. To bo zagotovilo, da bodo ključi urejeni najprej po dolžini n-ja, tisti z enako dolžino n-ja po samem n-ju, tisti z enakim n pa še po dejanski vrednosti števila. Med n-jem in vhodnim številom v ključu zdaj sploh ne potrebujemo presledka ali kakšnega drugega posebnega znaka. Nekaj primerov: za vhodno število 5 dobimo ključ `^15`, za število 67 dobimo ključ `^267`, za število 9876543210 dobimo ključ `^^109876543210` ipd. Ključ n-mestnega števila je tako od samega števila daljši le za $2l(n)$ znakov, pri čemer je $l(n) = 1 + \lfloor \log_{10} n \rfloor$ dolžina desetiškega zapisa števila n.[8]

[8]Lahko bi šli še korak dlje in prihranili pri strešicah tako, da ne bi dovolili poljubnega n. Mislimo si neko naraščajoče zaporedje $a_1, a_2, \ldots$; če imamo na vhodu n-mestno število, poiščimo v našem zaporedju prvi tak člen, ki je $\geq l(n)$; recimo, da je to a_k. Ključ zdaj sestavimo tako, da pred število n vrinemo toliko ničel, kolikor je treba, da bo dolgo a_k znakov (torej vrinemo $a_k - l(n)$ ničel); pred ta zapis n-ja postavimo še k strešic, na koncu pa pritaknemo kopijo vhodnega števila. Če vzamemo zaporedje, definirano z $a_k = k$, dobimo rešitev, kakršno smo opisali zgoraj v glavnem delu besedila; če pa vzamemo zaporedje, ki narašča hitreje, se lahko zgodi, da bo dobljena rešitev boljša: kar izgubimo zaradi vrivanja ničel, lahko več kot nadoknadimo s tem, da potrebujemo

(g) Za začetek lahko poskusimo ključ sestaviti tako, da črke vhodnega niza uredimo po abecedi. Iz niza `ccbdbc` bi tako nastal ključ `bbcccd`, iz niza `ddcbdbb` pa ključ `bbbcddd`. Ta primer nam že tudi kaže, da z našimi ključi še ni vse v redu: drugi niz ima več `b`-jev kot prvi, torej bi moral v vrstnem redu priti kasneje; toda ključ takega niza se začne na več `b`-jev kot ključ niza z manj `b`-ji, torej bo leksikografsko pred njim. Težava ni v samih `b`-jih, pač pa v tem, kar pride za njimi: niz `bb` je leksikografsko sicer pred `bbb`, če pa jima na koncu pritaknemo `c`, se vrstni red obrne: `bbc` je leksikografsko za nizom `bbbc`. Do tega problema ne bi prišlo, če bi za `b`-ji pritaknili kakšno črko, ki je leksikografsko pred `b`: na primer, `bba` je leksikografsko pred `bbba`. Pri naših ključih bo žal vedno ravno obratno, saj smo črke vhodnega niza uredili po abecedi. Tej težavi se lahko izognemo na primer s tem, da črke po abecednem urejanju spremenimo takole: vsak `a` spremenimo v `z`, vsak `b` v `y`, vsak `c` v `x` in tako naprej. Iz niza `ccbdbc` tako dobimo ključ `yyxxxw`, iz niza `ddcbdbb` pa ključ `yyyxwww`; prvi je leksikografsko pred drugim, to pa tudi hočemo, saj je imel prvi vhodni niz manj `b`-jev kot drugi.

```
char *Kljuc(char *Z)
{
  int dolzina = strlen(Z), f[26], i, j;
  char *K = (char*) malloc(dolzina + 1), *k;

  /* Preštejmo, koliko je pojavitev posamezne črke. */
  for (i = 0; i < 26; i++) f[i] = 0;
  for (i = 0; i < dolzina; i++) f[Z[i] − 'a']++;

  /* Sestavimo primeren ključ. */
  for (i = 0, k = K; i < 26; i++)
    for (j = 0; j < f[i]; j++) *k++ = 'z' − i;

  /* Dodajmo še znak za konec niza. */
  *k = 0; return K;
}
```

(h) Pri tej nalogi nas torej zanimajo ulomki med 0 in 1. Ulomkov z imenovalcem največ v je skupaj $1+2+\ldots+(v-1)+v$; za vsakega od njih, recimo za p/r (pri čemer je $0 \leq p < r \leq v$), lahko pogledamo, če je manjši ali enak u/v ali ne. Te ulomke preglejmo po naraščajočem r, pri vsakem r pa po naraščajočem p; če je $p/r \leq u/v$, mu pripišimo črko `b`, sicer pa črko `a`. Tako dobljeno zaporedje črk staknimo skupaj in dobimo nek ključ. Primer: iz ulomka 3/5 dobimo ključ `b ba bba bbaa bbbaa`.

Tako dobljeni ključi so sicer pravilna rešitev naše naloge,[9] vendar pa so neugodno dolgi — iz ulomka u/v smo naredili ključ dolžine $v(v+1)/2$. Oglejmo si še nekaj krajših rešitev. Za začetek lahko opazimo, da je naša prejšnja rešitev pravzaprav za vsak r (od 1 do v) poiskala največji tak p, pri katerem je bil $p/r \leq u/v$; ta p (recimo mu p_r; velja torej $p_r = \lfloor ru/v \rfloor$)) je potem predstavila z nizom p znakov `b` in $(r-p)$ znakov `a`, leksikografsko urejanje tako dobljenih ključev pa je pravzaprav pomenilo, da smo ulomek u/v predstavili z zaporedjem števil $(p_1, \ldots, p_v)$ in nato leksikografsko

---

manj strešic. Če na primer vzamemo $a_k = qk$ za nek $q > 1$, bomo morali vriniti največ $q-1$ ničel in $\lceil l(n)/q \rceil$ strešic. Ključ n-mestnega števila je tako od samega števila daljši za kvečjemu $l(n)(1+1/q)+q$ znakov. To je bolje od $2l(n)$, če je $l(n)$ dovolj velik. Glej tudi RFC 2550, razdelek 3.4.2, kjer so za a_k vzeli Fibonaccijevo zaporedje: $a_1 = 1$, $a_2 = 2$, $a_k = a_{k-1} + a_{k-2}$.

[9] Dokaz, da leksikografsko urejanje po takšnih ključih res uredi ulomke v naraščajočem vrstnem redu, prepuščamo bralcu za vajo, mi pa ga bomo tu preskočili in kasneje dokazali pravilnost za splošnejšo družino rešitev, med katere sodi tudi pravkar opisana rešitev.

urejali ta zaporedja. Naši ključi so torej števila p_r predstavljali takorekoč v eniškem zapisu; očitno bi bilo krajše, če bi jih predstavljali v dvojiškem, saj bi potem za zapis števila p_r (ki ima vrednost med 0 in $r-1$) porabili le $O(\log r)$ znakov, ne pa več $O(r)$ znakov. Celoten ključ bi bil tako dolg le še $O(\sum_{r=1}^{v} \log r) = O(v \log v)$ znakov, kar je že precej bolje kot prej.

Rešitev lahko še izboljšamo, če opazimo naslednje: vrednosti členov zaporedja $p_1, \ldots, p_v$ počasi naraščajo od $p_1 = 0$ do $p_v = u$. Dva zaporedna člena sta bodisi enaka ali pa je drugi le za 1 večji od prvega, do večjih skokov pa ne more priti: $p_{r+1} - p_r = \lfloor (r+1)u/v \rfloor - \lfloor ru/v \rfloor$ in če bi bilo to > 1, bi pomenilo, da sta med ru/v in $(r+1)u/v$ vsaj dve celi števili, torej je $(r+1)u/v - ru/v \geq 1$, torej je $u/v \geq 1$, kar je v protislovju z besedilom naloge. Če torej gledamo zaporedji p_r-jev za dva različna ulomka, recimo $(p_1, \ldots, p_v)$ za ulomek u/v in zaporedje $(p'_1, \ldots, p'_y)$ za ulomek x/y, in opazimo prvo neujemanje pri nekem konkretnem r — recimo, da je v drugem zaporedju večji člen: $p_r < p'_r$; ker sta se zaporedji pred tem ujemali in ker je lahko naslednji člen od prejšnjega večji kvečjemu za 1, je $p_r < p'_r$ možno le, če je $p_r = p_{r-1}$ in $p'_r = p'_{r-1} + 1$, torej če se je drugo zaporedje na tem mestu povečalo, prvo pa je ostalo enako. Namesto vrednosti p_r in p'_r bi lahko torej primerjali kar vrednosti $p_r - p_{r-1}$ in $p'_r - p'_{r-1}$, pa bi enako dobro opazili neujemanje. Lepo pri tej zamisli je, da so vrednosti $p_r - p_{r-1}$ vse enake ali 0 ali 1, torej za vsako od njih ne potrebujemo $O(\log r)$ znakov, pač pa le en znak. Celoten ključ pri tako dobljeni predstavitvi je dolg le še $v - 1$ znakov. Primer: pri ulomku 5/8 dobimo zaporedje p_r-jev $(\lfloor 5/8 \rfloor, \lfloor 10/8 \rfloor, \lfloor 15/8 \rfloor, \ldots, \lfloor 40/8 \rfloor) = (0, 1, 1, 2, 3, 3, 4, 5)$, razlike med dvema zaporednima členoma pa so torej $(1, 0, 1, 1, 0, 1, 1)$, tako da bi lahko ulomek 5/8 predstavili s ključem `1011011`.

Sestavimo pa lahko še krajše ključe. Izberimo si neko strogo naraščajoče zaporedje $z_1, z_2, \ldots$; pišimo $A_r := \{w/z : 0 \leq w < z \leq z_r\}$ in $A_r(u/v) := \{w/z \in A_r : w/z \leq u/v\}$. Ulomku u/v lahko zdaj pripišemo zaporedje $p_1, p_2, \ldots$, definirano s pravilom $p_r := |A_r(u/v)|$. Dovolj je, če gledamo člene p_r le do vključno prvega takega r, pri katerem je $z_r \geq v$. Izkaže se, da če tako dobljena zaporedja leksikografsko uredimo, bodo pripadajoči ulomki u/v urejeni naraščajoče, tako kot zahteva naša naloga. Število p_r je lahko veliko tja do $O(z_r^2)$, torej potrebujemo $O(\log z_r)$ znakov, da ga zapišemo. Koristno je vzeti takšno zaporedje z_r, ki narašča hitro (da ni treba zapisati preveč členov zaporedja p_r), vendar ne prehitro (sicer je lahko zadnji člen že zelo velik in porabimo preveč prostora, da ga zapišemo). Če vzamemo na primer $z_r = 2^r$, bomo potrebovali $O(\log v)$ členov zaporedja p_r, skupna dolžina ključa pa bo $O((\log v)^2)$ znakov. Še lepše je, če vzamemo $z_r = 2^{2^r}$, ko potrebujemo člene p_r le še do približno $r = \log_2 \log_2 v$ in naš ključ je dolg le še $O(\log v)$ znakov.[10] Asimptotično boljše rešitve pa že ne bi bilo razumno pričakovati, saj je ulomkov z imenovalcem do v približno $O(v^2)$ in torej potrebujemo $O(\log v)$ bitov, da enolično opišemo, kateri od njih nas zanima. Če bi namesto takšnih hitro naraščajočih zaporedij vzeli $z_r := r$, pa bi dobili prav tako rešitev, kot smo jo opisali na začetku (s ključi dolžine $O(v \log v)$ ali celo $O(v^2)$).

Prepričajmo se zdaj o pravilnosti opisane rešitve (deluje za poljubno zaporedje z_r, da je le strogo naraščajoče; je pa seveda od izbora tega zaporedja odvisna dolžina

[10] Glej debato na *comp.theory* 12. in 13. julija 2006. Rešitev z $z_r = 2^r$ je predlagal *neleai@seznam.cz*, rešitev z $z_r = 2^{2^r}$ pa T. Æ. Mogensen.

dobljenih ključev). Mislimo si dva ulomka u/v in x/y; brez izgube za splošnost recimo, da je $u/v < x/y$. Ulomek u/v nam da neko zaporedje $p_1, p_2, \ldots, p_r$, drugi ulomek x/y pa tudi neko zaporedje, ki mu recimo $q_1, \ldots, q_s$. Za začetek opazimo, da iz $u/v < x/y$ sledi, da je vsak ulomek w/z, ki je $\leq u/v$, tudi $\leq x/y$, torej je (pri vsakem t) $A_t(u/v) \subseteq A_t(x/y)$ in zato $p_t \leq q_t$. Poleg tega, če je $t \geq s$, vsebuje A_t med drugim tudi ulomek x/y, ki ni $\leq u/v$, je pa $\leq x/y$, zato pri $t \geq s$ velja celo $A_t(u/v) \subset A_t(x/y)$ in zato $p_t < q_t$.

Ali se lahko zgodi, da sta zaporedji p in q popolnoma enaki? Ne, saj bi to pomenilo, da sta tudi enako dolgi in pri zadnjem členu, s, bi vsekakor držalo $p_s < q_s$ (kot smo videli v prešnjem odstavku).

Torej se zaporedji razlikujeta; mogoče se razlikujeta v kakšnem členu, ki je prisoten še v obeh. Če je tako, naj bo t najmanjši indeks, pri katerem je $p_t \neq q_t$. Ker smo prej videli, da je p_t vedno $\leq q_t$, je neenakost možna le tako, da je $p_t < q_t$; to pa pomeni, da je zaporedje p leksikografsko pred zaporedjem q, točno to pa tudi hočemo (saj je p nastalo iz ulomka u/v, ki je manjši od x/y, iz katerega je nastalo q).

Druga možnost, kako se lahko zaporedji razlikujeta, pa je ta, da je eno zaporedje začetek (prefiks) drugega. To, da bi bilo q začetek zaporedja p, je nemogoče, saj bi bil v tem primeru člen s prisoten še v obeh in bi zato moral biti v obeh enak, mi pa smo prej videli, da je $p_s < q_s$. Torej mora biti p začetek zaporedja q; to pa pomeni, da je q leksikografsko za zaporedjem p, točno to pa tudi hočemo.

Tako torej vidimo, da iz manjšega ulomka v vsakem primeru dobimo zaporedje, ki je leksikografsko pred zaporedjem, dobljenim iz večjega ulomka. Naša rešitev torej ustreza zahtevam naloge.

Za konec omenimo še dve malenkosti: eno je to, da nič ne škoduje, če ulomek pred računanjem ključa okrajšamo — tako bomo včasih dobili še krajše ključe. Drugo pa je to, da bi lahko množice A_r pri zadnji rešitvi definirali tudi kot množice parov (w, z) namesto množice ulomkov w/z; hitro se vidi, da na pravilnost rešitev to nič ne vpliva, prihrani pa nam nekaj dela, ker pri izračunu vrednosti $p_r = |A_r(u/v)|$ ni treba paziti na dejstvo, da lahko več različnih parov (w, z) predstavlja isti ulomek w/z; računamo jo lahko preprosto kot $p_r = \sum_{z=1}^{z_r} \lfloor zu/v \rfloor$.

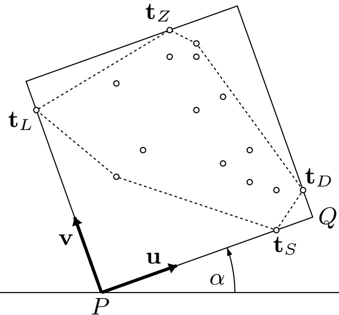
10. Smrkci

Naloga pravi, da so hiše točkaste in da poznamo njihove koordinate; za začetek poiščimo konveksno ovojnico te množice točk.[11] Recimo, da imamo nek kvadrat, ki pokrije vsa oglišča te konveksne ovojnice; ker je kvadrat tudi konveksen lik, vemo, da pokrije naš kvadrat zato tudi vse točke v notranjosti konveksne ovojnice. V nadaljevanju je torej dovolj, če obdržimo le tiste hiše, ki ležijo v ogliščih konveksne ovojnice, ostale pa lahko odmislimo, kajti vsak kvadrat, ki bo pokril vse hiše v ogliščih, bo pokril tudi vse hiše v notranjosti. Koordinate oglišč označimo s $\mathbf{t}_1, \ldots, \mathbf{t}_n$ (mislimo si jih oštevilčene nasproti smeri urinega kazalca).

Stranice našega kvadrata niso nujno vzporedne koordinatnima osema. Mislimo si najnižje oglišče kvadrata; če sta najnižji dve, vzemimo bolj levo med njima; dobljenemu oglišču recimo P. Premaknimo se po robu kvadrata v pozitivni smeri (nasproti

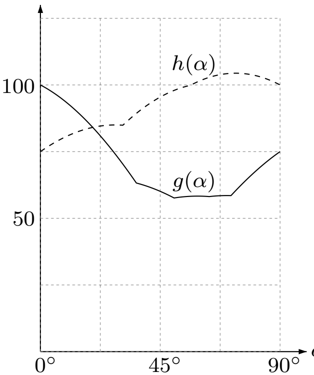
[11] Za to lahko uporabimo katerega koli od več dobro znanih algoritmov, na primer Grahamovega (*Graham scan*, znan tudi kot zavijanje paketa), ki izračuna konveksno ovojnico množice n točk v času $O(n \log n)$. Podrobnosti tega algoritma tu ne bomo opisovali, najdemo pa jih lahko v številnih učbenikih s tega področja.

smeri urinega kazalca) do naslednjega oglišča; recimo mu Q. Orientacijo kvadrata lahko opišemo s kotom α med pozitivno x-osjo in vektorjem $\vec{PQ}$. Primer vidimo na naslednji sliki:



Točke (oz. krožci) predstavljajo posamezne hiše; konveksna ovojnica je označena s črtkano črto. Prikazani kvadrat je najmanjši tak kvadrat, ki pokrije vse hiše.

Kolikšen je najmanjši kvadrat z orientacijo α, ki pokriva vse točke naše konveksne ovojnice? Mislimo si nov koordinatni sistem, ki je zasukan za kot α glede na prvotnega; enotska vektorja v njem sta torej $\mathbf{u} = (\cos\alpha, \sin\alpha)$ in $\mathbf{v} = (-\sin\alpha, \cos\alpha)$. Poljubno točko $\mathbf{t} = (x, y)$ lahko projiciramo na novi koordinatni osi in dobimo projekciji $\mathbf{t} \cdot \mathbf{u}$ in $\mathbf{t} \cdot \mathbf{v}$; to sta zdaj koordinati točke $\mathbf{t}$ v novem koordinatnem sistemu (u, v). Naj bo L indeks tiste točke med vsemi $\mathbf{t}_i$ (za $i = 1, \ldots, n$), ki ima najmanjšo u-koordinato, D pa indeks tiste z največjo u-koordinato; podobno naj bo S tista z najmanjšo v-koordinato, Z pa tista z največjo. To so torej najbolj leva, desna, spodnja in zgornja točka med vsemi, ki nas zanimajo. Kvadrat, ki ima orientacijo α in pokrije te štiri točke, bo pokril tudi vse ostale. Če hočemo pokriti točki L in D, mora biti naš kvadrat v smeri $\mathbf{u}$ širok vsaj $g(\alpha) := \mathbf{t}_D \cdot \mathbf{u} - \mathbf{t}_L \cdot \mathbf{u}$ enot. Podobno, če hočemo pokriti točki S in Z, mora biti naš kvadrat v smeri $\mathbf{v}$ visok vsaj $h(\alpha) := \mathbf{t}_Z \cdot \mathbf{v} - \mathbf{t}_S \cdot \mathbf{v}$ enot. Najmanjša možna stranica je torej $f(\alpha) := \max\{g(\alpha), h(\alpha)\}$.



Graf funkcij $g(\alpha)$ (polna črta) in $h(\alpha)$ (črtkana črta) za nabor hiš s prejšnje slike. Vidimo lahko, da je najmanjša vrednost funkcije $f(\alpha) = \max\{g(\alpha), h(\alpha)\}$ dosežena približno pri $\alpha \approx 19{,}7°$ (to je tudi naklon kvadrata, narisanega na prejšnji sliki).

Naša naloga torej pravzaprav zahteva, naj poiščemo $\min\{f(\alpha) : 0 \leq \alpha < \pi/2\}$. Težava je, da točke L, D, S in Z (od katerih sta odvisni funkciji g in h, s tem pa tudi f) niso pri vseh α enake; zato jih raje označimo kot $L(\alpha)$, $D(\alpha)$ in tako naprej. Oglejmo si zdaj, kako se v odvisnosti od α spreminja „najbolj spodnje“ oglišče naše konveksne ovojnice, torej $S(\alpha)$. Pri $\alpha = 0$ je $S(0)$ očitno kar tisto oglišče, ki ima

najmanjšo y-koordinato; če je takih več, vzemimo najbolj desno od njih. Če zdaj α počasi povečujemo, je točka $S(0)$ nekaj časa še vedno najnižja; ko pa α doseže (in preseže) naklon stranice od $\mathbf{t}_{S(0)}$ do $\mathbf{t}_{S(0)+1}$, postane „najnižje“ oglišče $S(0)+1$, ne več $S(0)$; od tu naprej imamo torej $S(\alpha) = S(0)+1$. Če zdaj α še povečujemo, je točka $S(0)+1$ nekaj časa še najnižja, ko pa α doseže naklon stranice od $\mathbf{t}_{S(0)+1}$ do $\mathbf{t}_{S(0)+2}$, postane najnižja točka $S(0)+2$ in imamo torej od tam naprej $S(\alpha) = S(0)+2$. Tako lahko nadaljujemo, dokler ne pridemo z α do $\pi/2$; večji koti od tega pa nas ne zanimajo. (Če pridemo prej do konca zaporedja oglišč, torej do $\mathbf{t}_n$, moramo seveda nadaljevati pri $\mathbf{t}_1$ in tako naprej.)

Tako vidimo, da je $S(\alpha)$ za $0 \le \alpha < \pi/2$ odsekoma konstantna funkcija in jo lahko predstavimo s seznamom vrednosti α, pri katerih se vrednost $S(\alpha)$ spremeni (pri vsaki od teh α tudi zapišimo novo vrednost $S(\alpha)$). Če vse $\mathbf{t}_i$ zavrtimo okoli koordinatnega izhodišča za 90° v pozitivno smer (torej iz (x, y) dobimo $(-y, x)$) in isti postopek ponovimo, bomo dobili potek funkcije $L(\alpha)$; če jih zavrtimo nato še enkrat za 90° in postopek ponovimo, bomo dobili $Z(\alpha)$, po še eni taki ponovitvi pa $D(\alpha)$.

Vse štiri sezname lahko zdaj zlijemo in imamo tako interval $[0, \pi/2)$ razdrobljen na kratke podintervale oblike $[\alpha_j, \alpha_{j+1})$, za katere velja, da je vsaka od funkcij $L(\alpha)$, $D(\alpha)$, $S(\alpha)$ in $Z(\alpha)$ po celem takem podintervalu konstantna. Če bi znali zdaj za vsak tak podinterval poiskati $\min\{f(\alpha) : \alpha_j \le \alpha < \alpha_{j+1}\}$, bi lahko na koncu vzeli najmanjšo od tako dobljenih vrednosti (prek vseh podintervalov) in bi to bil rezultat, po katerem sprašuje naša naloga.

Oglejmo si torej zdaj problem, kako poiskati minimum funkcije $f(\alpha)$ na intervalu $[\alpha_j, \alpha_{j+1})$, za katerega vemo, da so na njem funkcije $L(\alpha)$, $D(\alpha)$, $S(\alpha)$ in $Z(\alpha)$ konstantne. Pri vseh α s trenutnega intervala uporabljamo torej za izračun funkcij g, h in nato f ista štiri oglišča z naše ovojnice, ki jim recimo kar L, D, S in Z. Spomnimo se, da je $f(\alpha) = \max\{g(\alpha), h(\alpha)\}$. Pri nekaterih α je lahko $g(\alpha) > h(\alpha)$, pri nekaterih $g(\alpha) < h(\alpha)$; kdaj pa se lahko tudi zgodi, da sta $g(\alpha)$ in $h(\alpha)$ enaki. Če upoštevamo, kako smo definirali g in h, lahko pogoj $g(\alpha) = h(\alpha)$ predelamo v

$$
\begin{gathered}
(\mathbf{t}_D - \mathbf{t}_L) \cdot \mathbf{u} = (\mathbf{t}_Z - \mathbf{t}_S) \cdot \mathbf{v} \\
(x_D - x_L)\cos\alpha + (y_D - y_L)\sin\alpha = -(x_Z - x_S)\sin\alpha + (y_Z - y_S)\cos\alpha \\
(y_D - y_L + x_Z - x_S)\sin\alpha = (y_Z - y_S - x_D + x_L)\cos\alpha \\
\operatorname{tg}\alpha = (y_Z - y_S - x_D + x_L)/(y_D - y_L + x_Z - x_S)
\end{gathered}
$$

Ker naš interval $[\alpha_j, \alpha_{j+1})$ leži znotraj $[0, \pi/2)$ in ker je tangens na celem intervalu $[0, \pi/2)$ injektivna funkcija, lahko dobljenemu pogoju ustreza največ ena vrednost α z intervala $[\alpha_j, \alpha_{j+1})$. Če taka α res leži na tem intervalu, ga pri njej razbijmo na dva krajša intervala.

Tako imamo zdaj enega ali dva intervala, za katera velja, da je bodisi po celem intervalu $g(\alpha) > h(\alpha)$ ali pa je po celem intervalu $h(\alpha) > g(\alpha)$. Funkciji g in h sta namreč zvezni in se iz $g > h$ lahko v $h > g$ premaknemo le tako, da imata vmes pri vsaj eni α obe funkciji enako vrednost. Recimo, da zdaj gledamo enega od takih intervalov (označimo ga z A); nas še vedno zanima predvsem $\min_{\alpha \in A} f(\alpha)$. Če je po celem opazovanem intervalu $g(\alpha) > h(\alpha)$, potem je pri vseh teh α tudi $f(\alpha) = g(\alpha)$ in $\min_{\alpha \in A} f(\alpha) = \min_{\alpha \in A} g(\alpha)$; po drugi strani je $\min_{\alpha \in A} g(\alpha) \ge \min_{\alpha \in A} h(\alpha)$, saj je pri vsaki α vrednost $g(\alpha)$ vsaj tolikšna kot $h(\alpha)$. Tako torej vidimo, da je

$\min_{\alpha \in A} f(\alpha) = \max\{\min_{\alpha \in A} g(\alpha), \min_{\alpha \in A} g(\alpha)\}$. Do enake ugotovitve bi prišli s čisto podobnim razmislekom tudi za primer, ko po celem intervalu velja $h(\alpha) > g(\alpha)$ namesto $g(\alpha) > h(\alpha)$.

Tako torej vidimo, da če znamo izračunati $\min_{\alpha \in A} g(\alpha)$ (in podobno za funkcijo h), bomo znali izračunati tudi $\min_{\alpha \in A} f(\alpha)$, to pa je tisto, kar nas za interval A tudi najbolj zanima. Funkciji g in h sta si zelo podobni, obe sta namreč oblike $y(\alpha) = C \cos \alpha + S \sin \alpha$ za neki konstanti C in S. Taka funkcija je odvedljiva, torej bo minimum dosegla ali v enem od krajišč intervala A ali pa pri kakšni taki α, kjer je odvod $y'(\alpha)$ enak 0. Če y odvajamo, vidimo: $y'(\alpha) = -C \sin \alpha + S \cos \alpha$, torej je $y'(\alpha) = 0$ pri $\operatorname{tg} \alpha = S/C$; taka α je spet (podobno kot zgoraj) lahko na našem opazovanem intervalu A največ ena, torej ni težko preveriti, če res leži na njem in kakšna je v tem primeru vrednost $y(\alpha)$. Poleg nje izračunamo y še v obeh krajiščih intervala A in vrnemo najmanjšo vrednost.

11. Jezikovni tečaj

Naj bo $z[k]$ število slušateljev, ki bi bili pripravljeni sodelovati v skupini velikosti k (vsak tak slušatelj je seveda pripravljen sodelovati tudi v manjših skupinah, ni pa nujno, da je pripravljen sodelovati tudi v večjih skupinah — nekateri od njih mogoče so pripravljeni tudi na to, nekateri pa mogoče niso). Do tega podatka ni težko priti, če najprej za vsako velikost skupine preštejemo uporabnike, ki so kot svojo zgornjo mejo navedli točno to velikost; nato pa ta števila seštevamo: v skupini velikosti k bi bili pripravljeni sodelovati vsi tisti, ki bi bili pripravljeni sodelovati tudi v skupini velikosti $k + 1$, poleg tega pa še vsi tisti, ki so kot največjo sprejemljivo velikost skupine navedli točno k.

Nato pa lahko razmišljamo takole. Recimo, da bi radi oblikovali skupino k slušateljev. Kot smo izračunali v prejšnjem odstavku, obstaja le $z[k]$ kandidatov, ki bi bili pripravljeni sodelovati v taki skupini. Torej, če je $z[k] < k$, skupine velikosti k ne bomo mogli oblikovati, sicer pa jo bomo lahko. Ker bi radi čim večjo skupino, pregledujmo k-je od večjih proti manjšim; čim naletimo na takega, pri katerem je $z[k] \geq k$, se ustavimo in vemo, da smo našli primerno velikost skupine. Na koncu se moramo le še sprehoditi po seznamu prijavljenih in izbrati prvih k takih, ki bi bili pripravljeni sodelovati v skupini velikosti k (torej takih, ki so za največjo sprejemljivo velikost svoje skupine navedli število, večje ali enako k).

```
#include <stdio.h>

#define MaxN 10000
#define MaxA 10000

int main()
{
  int ai[MaxN], zk[MaxA + 1], n, m = 0, i, j, k;
  FILE *f = fopen("tecaj.in", "rt");
  /* Preberimo vhodne podatke; ai[i] bo velikost največje skupine, v kateri je pripravljen
     sodelovati slušatelj i. Največjega med vsemi ai[i] shranimo v m. */
  fscanf(f, "%d", &n);
  for (i = 0; i < n; i++) { fscanf(f, "%d", &ai[i]); if (ai[i] > m) m = ai[i]; }
  fclose(f);
  /* Naj bo zk[k] število slušateljev, za katere je ai[i] == k. */
```

```
  for (k = 0; k <= m + 1; k++) zk[k] = 0;
  for (i = 0; i < n; i++) zk[ai[i]]++;
  /* Naj bo zk[k] število slušateljev, za katere je ai[i] >= k. To so ljudje, ki bi bili
     pripravljeni sodelovati v skupini velikosti k (ne pa nujno tudi v večjih skupinah). */
  for (k = m − 1; k >= 0; k−−) zk[k] += zk[k + 1];
  /* Poiščimo največji k, pri katerem je zk[k] >= k. */
  k = m + 1; while (zk[k] < k) k−−;
  /* Izpišimo k in prvih k slušateljev, ki imajo ai[i] >= k. */
  f = fopen("tecaj.out", "wt");
  fprintf(f, "%d\n", k);
  for (i = 0, j = 0; i < n && j < k; i++)
    if (ai[i] >= k) { fprintf(f, "%d\n", i + 1); j++; }
  fclose(f); return 0;
}
```

12. Urejevalniška razdalja

(a) Naj bo $s = s_1 s_2 \ldots s_n$ beseda, ki jo iščemo (s_i je njen i-ti znak). Drevo začnimo pregledovati v korenu. Če je s nekje v drevesu, vodi do njega veja, na kateri ima prva povezava oznako s_1, druga povezava oznako s_2 in tako naprej. Poglejmo torej, če iz korena vodi kakšna povezava z oznako s_1; če je ni, vemo, da besede s ni v drevesu in se lahko takoj ustavimo. Sicer pa se premaknimo v otroka, v katerega vodi ta povezava. Tu zdaj pogledamo, če iz njega vodi kakšna povezava z oznako s_2 in tako naprej. Če uspemo na ta način narediti n korakov, smo v vozlišču, ki predstavlja niz $s_1 s_2 \ldots s_n$; tu moramo le še preveriti, ali je ta beseda tudi res v slovarju — ta podatek pa je tako ali tako shranjen v vsakem vozlišču (na sliki na str. 43) so taka vozlišča označena z dvojnim krožcem namesto z enojnim).

Zapišimo ta postopek še s psevdokodo — je sicer dovolj preprost, da to ni hudo potrebno, bo pa prišlo prav pri naslednjih dveh podnalogah. Za vozlišče q naj bo $q.\mathit{končno}$ logična vrednost, ki pove, ali je beseda, ki jo tvorijo črke na povezavah od korena do tega vozlišča, v slovarju ali ne (z drugimi besedami, ali bi bilo q na sliki narisano z dvojnim krožcem ali ne); $q.\mathit{otroci}$ pa naj bo seznam parov (a_i, q_i), pri čemer je q_i posamezni otrok vozlišča q, a_i pa je znak na povezavi od q do q_i.

```
postopek JeVSlovarju(s1 s2 ... sn):
  q := koren drevesa;
  for i := 1 to n:
    r := NIL;
    za vsak (a, q') v q.otroci:
      if a = si then r := q'; break;
    if r = NIL then return false;
    q := r;
  return q.končno;
```

(b) Za razliko od podnaloge (a) je zdaj pri pregledovanju drevesa včasih treba pregledati več otrok trenutnega vozlišča, ne le enega. Recimo, da smo trenutno pri črki s_i vhodnega niza s; mogoče v slovarju besede s ni, je pa zato kakšna enako dolga beseda, ki se od s razlikuje le v i-tem znaku. Če hočemo preveriti to možnost, moramo pregledati vse otroke trenutnega vozlišča, vendar pa si moramo pri tem tudi

zapomniti, da smo pravico do neujemanja pri enem znaku zdaj že izkoristili in je v nadaljevanju (pri pregledovanju nižje ležečih vozlišč v tistih poddrevesih) ne smemo še enkrat. Ta postopek lahko elegantno zapišemo z rekurzijo:

postopek PoisciBesedo($s_1 s_2 \ldots s_n$):
  naj bosta t in u dve tabeli n znakov;
  naj bo $d_u := \infty$;
  postopek *Rekurzija*(globina i, vozlišče q, število $d \in \{0, 1\}$):
    (* *Predpostavka: q je vozlišče na globini i, znaki na povezavah od korena do q pa tvorijo niz $t_1 t_2 \ldots t_i$, ki se od $s_1 s_2 \ldots s_i$ razlikuje le v d mestih.* *)
    if $i = n$:
      if q.*končno* **and** $d < d_u$ **then** $u = t$; $d_u := d$;
      return;
    za vsak (a, q') **v** q.*otroci*:
      $t_{i+1} := a$;
      if $a = s_{i+1}$ **then** *Rekurzija*($i + 1$, q', d);
      else if $d = 0$ **and** $d_u > 1$ **then** *Rekurzija*($i + 1$, q', $d + 1$);
      if $d_u \leq d$ **then break**;
  Rekurzija(0, *koren*, 0);
  if $d_u = \infty$ **then return** NIL;
  else return u;

Med rekurzijo torej v tabeli t hranimo niz, ki ga tvorijo znaki na povezavah od korena do trenutnega vozlišča, d pa je število mest, na katerih se t ne ujema z istoležnimi znaki s-ja (to je vedno 0 ali 1). Najboljši niz dolžine n, kar smo jih doslej našli, hranimo v u, poleg tega pa v d_u hranimo podatek o tem, na koliko mestih se razlikuje od s (to je 0 ali 1). Preden se spustimo po povezavi od trenutnega vozlišča q do nekega otroka q', preverimo, ali bi to povzročilo dodatno neujemanje; če bi ga, lahko nadaljujemo v tej smeri le, če doslej neujemanja še ni bilo (torej če je $d = 0$), saj bi sicer gledali že nize, ki se od s-ja razlikujejo na vsaj dveh mestih, takšni pa nas pri tej nalogi ne zanimajo. Poleg tega lahko poskusimo postopek malo pospešiti tudi s tem, da rekurzije ne nadaljujemo, če postane očitno, da ne bomo našli kakšne nove boljše rešitve (torej take, zaradi katere bi se d_u še kaj zmanjšal).

(c) Rešitev je podobna kot pri podnalogi (b); drevo lahko še vedno pregledujemo z rekurzijo in imamo parameter d, ki pove, ali smo „pravico“ do enega neujemanja doslej že izkoristili ali ne. Pač pa zdaj globina trenutnega vozlišča v drevesu ni dovolj, da bi vedeli, s katerim znakom vhodnega niza moramo primerjati znake na povezavah, ki izhajajo iz trenutnega vozlišča. Mogoče smo na primer izkoristili pravico do tega, da najdemo besedo, ki jo dobimo tako, da iz s-ja en znak pobrišemo; če smo torej zdaj na globini i, ne smemo gledati črke s_{i+1} (kot pri podnalogi (b)), pač pa črko s_{i+2}. Podobno v primeru, da odkrivamo besedo, ki se od s razlikuje po tem, da ima vrinjen še en dodaten znak, na globini i ne smemo gledati črke s_{i+1}, pač pa s_i. Trenutni položaj v nizu s bomo torej raje prenašali eksplicitno kot nov parameter, recimo j.

postopek PoisciBesedo($s_1 s_2 \ldots s_n$):
  naj bosta t in u dve tabeli z dovolj prostora za $n + 1$ znakov;
  naj bo $d_u := \infty$;

```
postopek Rekurzija(globina i, štev. j ∈ {0, ..., n}, vozlišče q, štev. d ∈ {0, 1}):
  (* Predpostavka: q je vozlišče na globini i, znaki na povezavah od korena
     do q pa tvorijo niz t1t2 ... ti, ki se od s1s2 ... sj razlikuje le v d znakih. *)
  if j = n:
    if q.končno and d < du then u := t1 ... ti; du := d; return;
    if d = 0 then za vsak (a, q') v q.otroci: (* vrivanje na koncu niza s *)
      if q'.končno then u := t1 ... ti a; du := d + 1; break;
    return;
  if d = 0 and du > 1 then (* brisanje znaka iz s *)
    Rekurzija(i, j + 1, q, d + 1);
  za vsak (a, q') v q.otroci:
    ti+1 := a;
    if a = sj+1 then Rekurzija(i + 1, j + 1, q', d);
    else if d = 0:
      if du > 1 then Rekurzija(i + 1, j, q', d + 1); (* vrivanje znaka v s *)
      if du > 1 then Rekurzija(i + 1, j + 1, q', d + 1); (* sprememba znaka *)
    if du ≤ d then break;
Rekurzija(0, 0, koren, 0);
if du = ∞ then return NIL;
else return u;
```

Vrivanje novega znaka v s smo dovolili le na mestih, kjer novi znak ne pride neposredno pred enak znak, ki bi obstajal v s že od prej (vrivanje novega znaka na konec niza je seveda dovoljeno v vsakem primeru). Rešitev bi postala malenkost enostavnejša, če bi za potrebe shranjevanja v drevesu vse besede iz slovarja podaljšali z nekim posebnim znakom, ki se v njih drugače nikoli ne pojavlja (tradicionalno ga označujejo z #).

13. Društva

(a in b) Naj bo U množica vseh prebivalcev našega kraja, A_i pa i-to društvo. Pišimo še $A_{i,1} = A_i$ in $A_{i,0} = U - A_i$. Prebivalce lahko razdelimo na 2^n množic takšne oblike:

$$R_{q_1 q_2 \ldots q_m} = \cap_{i=1}^{n} A_{i,q_i}.$$

Tako je na primer R_{0110} množica ljudi, ki so člani društev A_2 in A_3 in hkrati niso člani društev A_1 ali A_4. Vsak krajan pripada (v odvisnosti od svojega članstva ali nečlanstva v društvih) natanko eni od teh množic $R_\bullet$.

Razmislimo zdaj, kaj pravijo trditve iz besedila naloge o množicah $R_\bullet$:

1. Trditev „vsakdo, ki je član društva A_i, je tudi član društva A_j“ pomeni $A_i \subseteq A_j$, kar je isto kot $A_i - A_j = \emptyset$. Z drugimi besedami, vse $R_{q_1 \ldots q_m}$ za $q_i = 1$, $q_j = 0$ so prazne.

2. Trditev „A_i in A_j nimata nobenega skupnega člana“ (kar je isto kot $A_i \cap A_j = \emptyset$) pomeni, da so prazne vse $R_{q_1 \ldots q_m}$ za $q_i = 1$, $q_j = 1$.

3. Trditev „ni res, da je vsakdo iz A_i tudi član A_j“ (torej $A_i \not\subseteq A_j$) pomeni, da je neprazna vsaj ena od $R_{q_1 \ldots q_m}$ za $q_i = 1$, $q_j = 0$.

4. Trditev „A_i in A_j imata vsaj enega skupnega člana“ (torej $A_i \cap A_j \neq \emptyset$) pomeni, da je neprazna vsaj ena od $R_{q_1 \ldots q_m}$ za $q_i = 1$, $q_j = 1$.

Pripravimo si zdaj neko tabelo 2^n elementov (recimo ji p), v kateri bomo za vsako od množic $R_\bullet$ označili, ali je nujno prazna (zaradi trditev tipa 1 in 2). Nato za vsako trditev tipa 3 ali 4 s pomočjo te tabele poglejmo, če je med množicami $R_\bullet$, na katere se nanaša ta trditev, vsaj kakšna taka, ki ni nujno prazna. Če pri kakšni od trditev tipa 3 ali 4 ugotovimo, da so vse primerne $R_\bullet$ že označene kot prazne (zaradi trditev tipa 1 ali 2), vidimo, da so trditve nekonsistentne, torej ni takih društev, ki bi ustrezala vsem navedenim trditvam. Če pa pri pregledu vseh trditev na ta problem ne naletimo, lahko primeren nabor krajanov (za podnalogo b) sestavimo tako, da damo po enega namišljenega krajana v vsako od tistih množic $R_\bullet$, za katere iz trditev tipa 1 in 2 ni izhajalo, da morajo biti prazne. Zapišimo postopek še s psevdokodo; da bo lažje, označimo s t_i bit $i - 1$ števila t, torej $t_i := \lfloor t/2^{i-1} \rfloor \bmod 2$.

for $t := 0$ **to** $2^n - 1$ **do** $p_t :=$ **false**;
za vsako trditev oblike „$A_i \subseteq A_j$“:
  for $t := 0$ **to** $2^n - 1$ **do if** $t_i = 1$ **and** $t_j = 0$ **then** $p_t :=$ **true**;
za vsako trditev oblike „$A_i \cap A_j = \emptyset$“:
  for $t := 0$ **to** $2^n - 1$ **do if** $t_i = 1$ **and** $t_j = 1$ **then** $p_t :=$ **true**;
$ok :=$ **true**;
za vsako trditev oblike „$A_i \not\subseteq A_j$“:
  $b :=$ **false**;
  for $t := 0$ **to** $2^n - 1$ **do**
    if $t_i = 1$ **and** $t_j = 0$ **and not** p_t **then** $b :=$ **true**; **break**;
  if not b **then** $ok :=$ **false**;
za vsako trditev oblike „$A_i \cap A_j \neq \emptyset$“:
  $b :=$ **false**;
  for $t := 0$ **to** $2^n - 1$ **do**
    if $t_i = 1$ **and** $t_j = 1$ **and not** p_t **then** $b :=$ **true**; **break**;
  if not b **then** $ok :=$ **false**;
if not ok **then** (* *ok je odgovor na podnalogo a* *)
  dane trditve so nekonsistentne; **return**;
$m := 0$; **for** $i := 1$ **to** n **do** $A_i := \{\}$;
for $t := 0$ **to** $2^n - 1$ **do if not** p_t **then**
  (* *Koristno je, če je množica R_t neprazna; izmislimo si novega krajana, $m + 1$, in ga dodajmo vanjo ter v pripadajoča društva.* *)
  $m := m + 1$; **for** $i := 1$ **to** n **do if** $t_i = 1$ **then** $A_i := A_i \cup \{m\}$;
(* *Zdaj imamo v $A_1, \ldots, A_n$ primer konkretnega nabora društev, ki ustrezajo vsem podanim trditvam (za kraj z m prebivalci).* *)

(c) Naloga postane še jasnejša, če jo prevedemo v jezik logike. Označimo z $A = (A_1, \ldots, A_n)$ naš nabor n društev. Najprej smo dobili o društvih neke trditve, recimo jim $P_1(A), \ldots, P_r(A)$, ki naj bi vse veljale; naloga je torej zatrdila njihovo konjunkcijo $P_1(A) \wedge P_2(A) \wedge \ldots \wedge P_r(A)$; da bo krajše, označimo to konjunkcijo kar s $P(A)$. Pri podnalogi (a) smo ugotavljali, ali je P protislovna (torej taka, da ne velja za noben A); in če ni, smo pri podnalogi (b) iskali konkreten primer nabora A, za katerega $P(A)$ drži. Pri podnalogi (c) pa dobimo še neko novo trditev, recimo ji $Q(A)$ (ki je

enake oblike kot posamezne trditve $P_i(A)$), in naloga nas pravzaprav sprašuje, kaj od naslednjega trojega drži: (i) $\forall A : P(A) \Rightarrow Q(A)$; ($ii$) $\forall A : P(A) \Rightarrow \neg Q(A)$; ali ($iii$) nič od tega dvojega.

Pri podnalogi (a) smo se pravzaprav naučili preverjati trditev „$\forall A : \neg P(A)$“ oz., če jo zapišemo daljše: „$\forall A : \neg(P_1(A) \wedge \ldots \wedge P_r(A))$“. Oglejmo si zdaj trditev iz možnosti (i): $\forall A : P(A) \Rightarrow Q(A)$ je isto kot $\forall A : \neg(P(A) \wedge \neg Q(A))$, to pa je isto kot $\forall A : \neg(P_1(A) \wedge \ldots \wedge P_{r+1}(A))$, če si mislimo $P_{r+1}(A) := \neg Q(A)$. Ker je Q enake oblike kot trditve P_i iz podnaloge (a), je tudi njena negacija take oblike. Tako torej vidimo, da lahko možnost (i) preverimo preprosto tako, da med dosedanje trditve $P_1, \ldots, P_r$ dodamo še trditev $\neg Q$ in nato s postopkom za reševanje podnaloge (a) preverimo, ali s to dodatno trditvijo pridemo v protislovje ali ne.

Z analognim razmislekom se lahko prepričamo, da lahko možnost (ii) preverimo tako, da med dosedanje trditve $P_1, \ldots, P_r$ dodamo še trditev Q in spet poženemo postopek za reševanje podnaloge (a).

Če v nobenem od obeh primerov ne bi prišli do protislovja, pomeni, da velja možnost (iii) — z drugimi besedami, za nekatere A-je velja $P(A) \wedge Q(A)$, za nekatere A-je pa velja $P(A) \wedge \neg Q(A)$.

Četrta možnost bi bila, da bi do protislovja prišli obakrat, tako pri preverjanju (i) kot pri preverjanju (ii). To pomeni, da je že $P(A)$ sama po sebi protislovna. (Besedilo naloge pravzaprav pravi, da se nam s to možnostjo ni treba ukvarjati.)

14. Elektrika

Recimo, da imamo n ponudb, pri čemer ima i-ta ponudba minimalno količino a_i, maksimalno b_i in ceno c_i; mi pa bi radi kupili q enot. Naloge se lahko lotimo z dinamičnim programiranjem. Naj bo $f_t(r)$ najnižja cena, za katero lahko dobimo natanko r enot elektrike, če smemo uporabljati le prvih t ponudb. Funkcije f ni težko opisati rekurzivno:

$$f_0(0) = 0$$
$$f_0(r) = \infty \text{ za } r \neq 0$$
$$f_t(r) = \min\{k \cdot c_t + f_{t-1}(r-k) : k = 0 \vee a_t \leq k \leq b_t\} \text{ za } t > 0.$$

To definicijo bi lahko precej neposredno predelali v funkcijski podprogram (v C-ju, pascalu ali kakšnem drugem programskem jeziku) z dvema parametroma, t in r, ki bi računal $f_t(r)$ in pri tem po potrebi rekurzivno klical samega sebe. Računanje minimuma bi izvedli z zanko po k. Hitro bi se izkazalo, da se funkcija kliče po večkrat z istimi vrednostmi parametrov in po nepotrebnem izgublja čas z računanjem vedno istih rezultatov. Zato je koristno računati vrednosti funkcije sistematično in si jih shranjevati v neki tabeli. Računali jih bomo po naraščajočem t, pri vsakem t-ju pa po padajočem r; tako bomo imeli vedno pri roki že izračunane vse tiste vrednosti, ki jih potrebujemo, obenem pa bomo lahko sproti brez škode povozili stare vrednosti, ki jih ne bomo več potrebovali. Vrednosti $f_t(r)$ za $r < 0$ ni treba shranjevati, saj iz definicij sledi, da je vsaka f_t tam enaka ∞:

```
f[0] := 0; for r := 1 to q do f[r] := ∞;
for t := 1 to n:
  for r := q downto 0:
```

```
(* Na tem mestu so v f[r'] za r' ≤ r vrednosti f_{t−1}(r),
   za r' > r pa so v f[r'] vrednosti f_t(r). *)
for k := a_t to b_t:
   if k > r then break;
   f[r] := min{f[r], k · c_t + f[r − k]};
(* Zdaj je tudi v f[r] vrednost f_t(r). *)
```

Na koncu tega postopka je v $f[q]$ vrednost $f_n(q)$, to pa je tudi najnižja cena, za katero je mogoče pri danih ponudbah kupiti q enot elektrike. Vidimo, da je časovna zahtevnost tega postopka v najslabšem primeru $O(q^2 n)$ oz. natančneje $O(q \sum_{i=1}^{n} |b_i - a_i + 1|)$ — slednja ocena utegne biti bolj realistična, če so intervali ponudb $[a_i, b_i]$ razmeroma ozki. Prostorska zahtevnost pa je $O(q)$.[12]

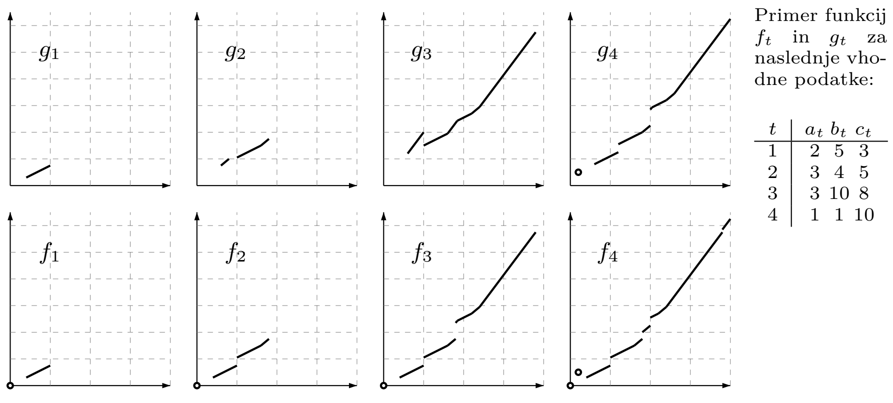
Če bi hoteli na koncu tudi izpisati, koliko elektrike je treba kupiti pri katerem ponudniku, bi bilo koristno med računanjem funkcij f_t v neki tabeli za vsak par (t, r) zapisati, pri katerem k je bil dosežen minimum, s katerim je bila definirana vrednost $f_t(r)$; temu k-ju recimo $k_t^\star(r)$. Za razliko od tega, kar smo zgoraj začeli s tabelo f, tukaj vrednosti za manjše t-je ne smemo povoziti s tistimi za večje, ker jih bomo še potrebovali za rekonstrukcijo rešitve. Prostorska zahtevnost s tem naraste na $O(qn)$. Z nekaj zvitosti lahko prostorsko zahtevnost zmanjšamo na $O(q \log n)$, če si zapomnimo vrednosti f_t in $k_t^\star$ le za $O(\log n)$ primerno izbranih vrednosti t, ostale pa po potrebi računamo ponovno; časovna zahtevnost se pri tem poveča le na $O(q^2 n \log n)$.

Rešitev za ne-cele količine elektrike. Pri naši rešitvi smo doslej predstavljali funkcijo $f_t(r)$ (pri fiksnem t) kar s tabelo $O(q)$ celic, ki podajajo vrednosti funkcije za vse možne r. To bi bilo lahko neučinkovito, če so količine, s katerimi delamo (q, a_t in b_t), velike, povsem pa odpove v primeru, če te količine niso cela števila. Oglejmo si še rešitev, ki bi delovala tudi pri necelih količinah.

Definirajmo poleg funkcije f_t še eno podobno funkcijo: $g_t(r)$ naj bo najnižja cena, za katero lahko kupimo r enot elektrike, če lahko uporabljamo le prvih t ponudb, pri čemer pa ponudbo t nujno moramo uporabiti. Očitno za poljuben t velja $f_t(r) = \min\{f_{t-1}(r), g_t(r)\}$. V nadaljevanju se bomo prepričali, da so vse funkcije f_t in g_t odsekoma linearne; zato f_t ni težko izračunati iz f_{t-1} in g_t z zlivanjem. Videli bomo tudi, kako lahko izračunamo g_t iz f_{t-1}. Pomembno za naš razmislek je, da si mislimo ponudbe oštevilčene po naraščajoči ceni, torej tako, da je $c_1 \le c_2 \le \ldots \le c_n$. Primer funkcij f_t in g_t za različne t kaže slika na str. 100.

Naj bo h ena od teh funkcij (torej f_t ali g_t za nek t); trdimo, da je vsak tak h odsekoma linearna funkcija in ga lahko predstavimo s štirimi zaporedji, r_j, l_j, m_j in d_j za $j = 0, \ldots, u$, ki nam povedo, da je j-ti odsek funkcije daljica od (r_{j-1}, d_{j-1})

[12] Naloga pravi, da so q in vse a_t in b_t cela števila. Naš gornji algoritem se je pri reševanju omejil le na take rešitve, pri katerih je tudi količina elektrike, kupljene pri vsakem posameznem ponudniku, celo število (najbolj notranja zanka gleda celoštevilske k od a_t do b_t); te slednje omejitve pa naloga nikjer posebej ne omenja, zato bi se spodobilo dokazati, da navkljub tej svoji dodatni omejitvi naš postopek ne bo spregledal najcenejšega možnega nakupa. Pri takšnem dokazu si lahko pomagamo z naslednjim razmislekom: če bi bile pri najboljši rešitvi nekatere kupljene količine ne-cele, bi lahko vzeli dve taki in nakup pri enem ponudniku malo zmanjšali, pri drugem malo povečali, ne da bi skupna cena zato kaj povečala, pač pa bi se zmanjšalo število ne-celih nakupov v rešitvi. Podrobnosti dokaza pa prepuščamo bralcu za vajo.



Primer funkcij f_t in g_t za naslednje vhodne podatke:

t	a_t	b_t	c_t
1	2	5	3
2	3	4	5
3	3	10	8
4	1	1	10

do (l_j, d_j). Z drugimi besedami, funkcija h je definirana takole:

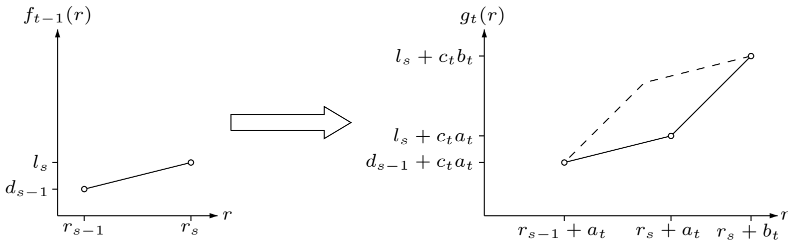
$$\begin{aligned} h(r) &= l_0 && \text{za } r < r_0 \\ h(r) &= d_u && \text{za } r > r_u \\ h(r) &= \min\{l_j, m_j, d_j\} && \text{za } r = r_j \\ h(r) &= d_{j-1} + (l_j - d_{j-1})(r - r_{j-1})/(r_j - r_{j-1}) && \text{za } r_{j-1} < r < r_j. \end{aligned}$$

Vedno bo tudi veljalo $l_0 = d_u = \infty$ in $0 = r_0 < r_1 < \ldots < r_u$. Tukaj torej l_j pomeni vrednost, ki se ji funkcija približa, če se r približa r_j z leve, d_j pa vrednost, ki se ji funkcija približa, če se r približa r_j z desne. (Vrednost m_j pride prav, če imamo v vhodnih podatkih kakšne ponudbe z $a_t = b_t$, zaradi katerih moramo potem na funkciji h mogoče kakšno posamezno vrednost r obravnavati, kot da bi bila samostojen odsek.) Med dvema takima točkama, npr. med r_{j-1} in r_j, pa je h linearna; še več, naklon $(l_j - d_{j-1})/(r_j - r_{j-1})$ je enak eni od cen $c_1, \ldots, c_t$. Neskončne cene predstavljajo območja r-jev, na katerih velja, da se z razpoložljivimi ponudbami sploh ne da kupiti natanko r enot elektrike.

O tem, da so res vse f_t in g_t takšne oblike, kakor tudi o tem, kako jih lahko računamo, se bomo prepričali z indukcijo po t.

Pri $t = 0$ je f_t zelo preprosta: ker sploh nimamo pri kom kupovati elektrike, imamo le možnost, da za 0 enot denarja kupimo 0 enot elektrike, torej $u = r_0 = m_0 = 0$ in $l_0 = d_0 = \infty$. Funkcija g_t je pri $t = 0$ nedefinirana, zanimivo pa si jo je ogledati pri $t = 1$. Takrat imamo le ponudbo številka 1, ki jo nujno moramo uporabiti; dobimo $u = 2$, $r = (0, a_1, a_2)$, $l = (\infty, \infty, c_1 b_1)$, $m = (\infty, c_1 a_1, c_1 b_1)$ in $d = (\infty, c_1 a_1, \infty)$. Ta funkcija je torej na intervalu $[a_1, b_1]$ linearna z naklonom c_1, cena nakupa pa se na tem intervalu povzpne od $c_1 a_1$ do $c_1 b_1$. (Posebej bi morali obravnavati še primere, ko je $a_1 = 0$ in/ali $a_1 = b_1$.) Od tu naprej, torej za funkcije $f_1, g_2, f_2, \ldots$ bomo lahko razmišljali z indukcijo.

Recimo zdaj, da smo našo trditev dokazali že vse do $t - 1$, zdaj pa jo hočemo dokazati še za t; torej moramo za f_t in g_t dokazati, da sta tudi v opisani obliki. Za g_t bomo pokazali postopek, ki izračuna g_t iz f_{t-1} in pri tem dobi g_t spet v zahtevani obliki; za f_t pa je stvar potem enostavna, saj ga lahko zaradi zveze $f_t(r) = \min\{f_{t-1}(r), g_t(r)\}$ izračunamo z zlivanjem zaporedij, s katerima sta predstavljeni funkciji f_{t-1} in g_t.



Primer, kako iz enega odseka funkcije f_{t-1} nastane z dodatkom nove ponudbe (a_t, b_t, c_t) paralelogram možnih nakupov, od katerega sta za funkcijo g_t zanimiva spodnji rob (položni odsek) in desni rob (strmi odsek).

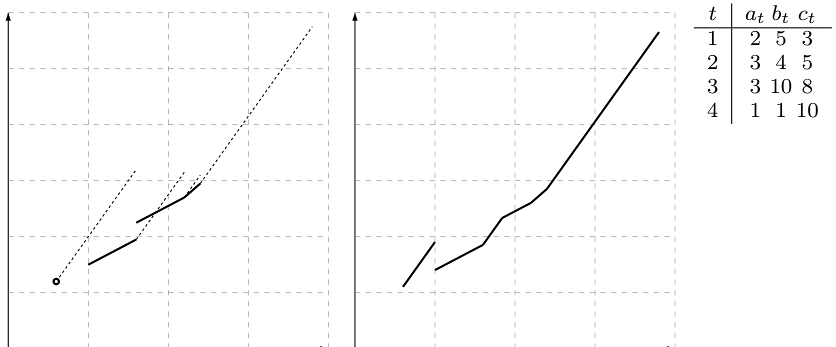
Posvetimo se torej zdaj izračunu funkcije g_t iz f_{t-1}. Po predpostavki že imamo pri roki zaporedja r_j, l_j in d_j za funkcijo f_{t-1}, pri čemer gre indeks j od 0 do u; iz njih bi zdaj radi izračunali ustrezna tri zaporedja še za funkcijo g_t.

Oglejmo si posamezni odsek funkcije f_{t-1}; recimo odsek od r_{s-1} do r_s, pri čemer se funkcija povzpne od vrednosti d_{s-1} do l_s. Vsaka od točk $(r, f_{t-1}(r))$ na tem odseku nam pove, da lahko s ponudbami od 1 do $t-1$ kupimo r enot elektrike za ceno $f_{t-1}(r)$, ceneje pa ne. Vsak tak nakup nas pripelje do več možnih nakupov pri funkciji g_t, namreč do nakupov oblike $(r + \hat{r}, f_{t-1}(r) + \hat{r}c_t)$ za $a_t \leq \hat{r} \leq b_t$ (število $\hat{r}$ torej pove, koliko elektrike kupimo pri novem ponudniku t, ki ga pri funkciji g_t nujno moramo uporabiti). Te točke tvorijo v ravnini paralelogram (glej gornjo sliko); po induktivni predpostavki je imel prvotni odsek (na f_{t-1}) naklon c_i za $i \leq t-1$ in ker imamo ponudbe oštevilčene po naraščajoči ceni, je ta naklon vsekakor manjši ali enak c_t. Zato ima dobljeni paralelogram spodnjo stranico z naklonom c_i in desno stranico z naklonom c_t; ker nas pri vsakem r zanima le najnižja možna cena, pridejo v poštev le nakupi, ki ležijo na teh dveh stranicah paralelograma. Iz našega prvotnega odseka funkcije f_{t-1} smo tako dobili dva odseka, enega od $(r_{s-1} + a_t, d_{s-1} + c_t a_t)$ do $(r_s + a_t, l_s + c_t a_t)$ in enega od $(r_s + a_t, l_s + c_t a_t)$ do $(r_s + b_t, l_s + c_t b_t)$. Prvi ima še vedno naklon c_i in mu bomo rekli *položni odsek*, drugi pa strmejši naklon c_t in mu bomo rekli *strmi odsek*.[13]

Ta razmislek lahko ponovimo pri vsakem odseku funkcije f_{t-1} in iz njega na ta način dobimo dva odseka; težava je le še v tem, da se lahko ti odseki prekrivajo, torej isti r pripada več odsekom. Položni odseki se med seboj sicer ne morejo prekrivati in tvorijo sami zase neko funkcijo, ki ji recimo $\tilde{g}_t$; to ni nič drugega kot funkcija f_{t-1}, zamaknjena za a_t enot desno in $c_t a_t$ enot navzgor. Strmi odseki pa se lahko prekrivajo med sabo (ali pa s pomožnimi); pripravimo si zdaj še pomožno funkcijo $\hat{g}_t$, ki jo sestavljajo le strmi odseki, vendar okleščeni tako, da če je nek r pripadal prej več strmim odsekom, obdržimo pri tem r-ju le tisti odsek, ki nam da najcenejši nakup. Funkcijo g_t bomo lahko potem izračunali z zlivanjem seznamov, saj velja

---

[13]Podobno moramo obravnavati tudi morebitne točkaste odseke na funkciji f_{t-1}. Če je $m_s < \min\{l_s, d_s\}$, bo iz te točke (r_s, m_s) na funkciji f_{t-1} nastal točkast „položni odsek“ $(r_s + a_t, m_s + c_t a_t)$ in od te točke naprej strmi odsek do $(r_s + b_t, m_s + c_t b_t)$; strmega odseka od $(r_s + a_t, l_s + c_t a_t)$ do $(r_s + b_t, l_s + c_t a_t)$ pa v tem primeru ne potrebujemo, saj leži v celoti nad tistim z začetkom pri $(r_s + a_t, m_s + c_t a_t)$. Če je $\min\{l_s, m_s\} < d_s$, pa ne potrebujemo nobenega od teh dveh strmih odsekov, saj oba ležita nad naslednjim položnim odsekom (ki se bo začel na višini $d_s + c_t a_t$) in nad morebitnim strmim odsekom, ki se bo razvil iz njega.



t	a_t	b_t	c_t
1	2	5	3
2	3	4	5
3	3	10	8
4	1	1	10

Levi graf kaže funkciji $\tilde{g}_3$ (polna črta) in $\hat{g}_3$ (črtkana črta), desni pa funkcijo g_3 za vhodne podatke, prikazane v tabeli na desni strani.

$g_t(r) = \min\{\tilde{g}_t(r), \hat{g}_t(r)\}$.[14] Primer takih funkcij kaže gornja slika.

Strmi odseki so, kot smo videli zgoraj, daljice od $(r_s + a_t, l_s + ca_t)$ do $(r_s + b_t, l_s + cb_t)$ za $s = 0, \ldots, u$. Pri vsakem r-ju nas zanima, kateri od teh odsekov leži najnižje (in nam torej omogoči najcenejši nakup r enot elektrike; ta cena bo vrednost funkcije $\hat{g}_t(r)$); ker imajo vsi ti odseki enak naklon (namreč c_t), se med seboj ne sekajo in do sprememb v tem, kateri odsek je najnižje, lahko pride le tam, kjer se kakšen od odsekov začne ali konča, torej v točkah $r_s + a_t$ in $r_s + b_t$ za razne s. Ravnino pregledujmo po naraščajočem r in vzdržujmo množico trenutno aktivnih strmih odsekov: $A(r) = \{s : 0 \leq s \leq u, r_s + a_t \leq r \leq r_s + b_t\}$. Ko pridemo do nekega $r = r_s + a_t$, dodamo odsek s v množico A, pri $r = r_s + b_t$ pa ga pobrišemo iz nje.

Za ugotavljanje tega, kateri odsek (izmed tistih, ki so trenutno v A) leži najnižje, je koristno, če za vsakega izračunamo, pri kateri višini (= ceni) seka abscisno os, torej $r = 0$; to je pri ceni $l_s - c_t r_s$. Množico A lahko hranimo na primer v rdeče-črnem drevesu, v katerem so odseki urejeni po vrednosti $l_s - c_t r_s$, tako da bomo lahko v času $O(\log |A|)$ poiskali trenutno najnižji odsek; prav takšno časovno zahtevnost bo imelo tudi dodajanje in brisanje odseka.[15]

Časovna zahtevnost tega izračuna funkcije $\hat{g}_t$ je torej $O(u \log u)$, če je u (kot smo rekli že zgoraj) število odsekov v funkciji f_{t-1}. Nato imamo še $O(u)$ dela z zlivanjem funkcij $\hat{g}_t$ in $\tilde{g}_t$ v g_t in nato z zlivanjem g_t in f_{t-1} v f_t. Žal se lahko pri neugodno sestavljenih vhodnih primerih zgodi, da število odsekov u narašča eksponentno v odvisnosti od t in je zato celoten postopek še vedno zelo neučinkovit.

Zdaj imamo torej postopek, s katerim lahko sčasoma izračunamo tudi $f_n(q)$, kar je najnižja cena, za katero lahko kupimo q enot elektrike. Razmislimo še o tem, kako

[14] Nekoliko poseben primer nastopi, če je ponudba „točkasta“, torej $a_t = b_t$; takrat strmi odseki na najcenejše nakupe nič ne vplivajo in imamo kar $g_t = \tilde{g}_t$.

[15] Za učinkovito brisanje pride potem prav neko kazalo, tabela $O(u)$ elementov, ki za vsak možni odsek kaže na vozlišče v drevesu, kjer se ta odsek nahaja, če je seveda trenutno sploh prisoten v množici A. Ker pregledujemo odseke od leve proti desni, je možna še naslednja drobna izboljšava: ko dodamo nek odsek v množico A, lahko iz nje takoj pobrišemo vse tiste odseke, ki ležijo nad njim. Ti odseki se namreč, ker so že v množici A, očitno začnejo levo od trenutnega odseka, torej se bodo tudi končali levo od konca trenutnega odseka; in ker leži novi odsek nižje od njih, ne bodo nikoli več imeli možnosti, da bi se izkazali kot najnižji odseki v množici A, torej jih lahko iz nje kar takoj pobrišemo.

bi rekonstruirali tudi konkreten nabor nakupov s to skupno ceno, torej točno količino kupljene elektrike pri vsakem ponudniku. V ta namen si bomo morali pri računanju funkcij g_t in f_t zapomniti še nekatere dodatne podatke. Ko zlivamo f_{t-1} in g_t, da dobimo f_t, si moramo za vsak odsek funkcije f_t zapomniti, ali je prišel iz f_{t-1} ali iz g_t. Pri funkciji g_t pa moramo posvetiti nekaj dodatne pozornosti strmim odsekom. Prvotno je bil vsak strmi odsek širok $b_t - a_t$ enot in je predstavljal različne količine kupljene elektrike pri ponudniku t, od a_t na levem koncu odseka do b_t na desnem koncu. Pri zlivanju odsekov ob izračunu $\hat{g}_t$ in kasneje pri zlivanju $\hat{g}_t$ in $\tilde{g}_t$ v g_t se lahko ti strmi odseki tudi skrajšajo in razdrobijo, zato moramo med zlivanjem ves čas vzdrževati podatke o tem, kolikšen nakup pri ponudniku t predstavlja levi konec posameznega strmega odseka. Opisane podatke lahko za rekonstrukcijo najboljše rešitve uporabljamo takole. Recimo, da nas zanima, kako najceneje kupiti $\hat{q}$ enot elektrike s ponudbami od 1 do t. Poglejmo v funkcijo f_t, ali leži $\hat{q}$ na odseku, ki je prišel iz f_{t-1}, ali na odseku, ki je prišel iz g_t; če velja prvo, nadaljujemo z istim $\hat{q}$ pri funkciji f_{t-1} (in vemo, da pri naši rešitvi od ponudnika t nismo kupili nič elektrike). Če velja drugo, pa poglejmo, na katerem odseku leži $\hat{q}$ v funkciji g_t; če je ta odsek položen, vemo, da smo pri naši rešitvi kupili od ponudnika t le minimalno količino elektrike, to je a_t, postopek pa lahko nadaljujemo pri f_{t-1} s količino $\hat{q} - a_t$. Ostane še možnost, da leži $\hat{q}$ v g_t na nekem strmem odseku; recimo, da je to odsek od r_j do r_{j+1} (pri čemer seveda velja $r_j \leq \hat{q} \leq r_{j+1}$) in da levi konec tega odseka predstavlja nakup p_j enot elektrike pri ponudniku t; v tem primeru vemo, da smo pri naši rešitvi od ponudnika t v resnici kupili $p_j + (\hat{q} - r_j)$ enot elektrike, naš postopek pa lahko nadaljujemo pri funkciji f_{t-1} z zmanjšano količino $\hat{q} - (p_j + (\hat{q} - r_j)) = r_j - p_j$.

NP-težkost. Rešitve, ki bi bila učinkovita tudi v najslabšem možnem primeru, pri tej nalogi pravzaprav ni pametno pričakovati, saj je naš problem NP-težak. O tem se lahko prepričamo na primer tako, da znani problem 0/1-nahrbtnika prevedemo na naš problem nakupovanja elektrike. Recimo, da imamo 0/1-nahrbtnik velikosti q in n predmetov z velikostmi m_i in vrednostmi v_i. Sestavimo zdaj takšen problem nakupovanja elektrike: naj bo $\hat{c}$ poljubno število, večje od vseh v_i/m_i; za vsak predmet i od 1 do n vpeljimo ponudnika elektrike z $a_i = b_i = m_i$ in $c_i = \hat{c} - v_i/m_i$; poleg tega dodajmo še ponudnika številka $n + 1$ s parametri $a_{n+1} = 0$, $b_{n+1} = q$ in $c_{n+1} = \hat{c}$ (ta dodatni ponudnik predstavlja možnost, da nahrbtnik ni čisto poln — problem 0/1-nahrbtnika namreč zahteva le, da skupna velikost izbranih predmetov ne sme preseči q, naš problem nakupovanja elektrike pa zahteva nakup natanko q enot, ne manj kot toliko). Hitro se lahko prepričamo, da če zdaj za ta nabor ponudnikov poiščemo najcenejši način, kako kupiti q enot elektrike, nam bo rešitev povedala tudi to, katere predmete izbrati za v naš nahrbtnik, da bomo maksimizirali njihovo skupno vrednost in pri tem ne presegli velikosti nahbrtnika. Ker je problem 0/1-nahrbtnika NP-težak, nam opisana prevedba kaže, da je NP-težak tudi naš problem nakupovanja elektrike.

15. Bakterije

Če smo začeli z a bakterijami, imamo po prvi delitvi $2a$ bakterij, po drugi $4a$ in tako naprej; po n delitvah imamo torej $a \cdot 2^n$ bakterij. Naloga sprašuje, kolikšen je ostanek po deljenju tega števila s k. Zelo naivna rešitev bi torej lahko najprej izračunala $a \cdot 2^n$ in ga nato delila s k:

$x := a$;
for $i := 1$ **to** n **do** $x := 2 \cdot x$;
$x := x \bmod k$;
return x;

Prva težava, ki pri tem nastopi, je, da je lahko $a \cdot 2^n$ zelo veliko število; naloga pravi, da gre lahko a do 10^4, n pa do 10^9, tako da je $a \cdot 2^n$ lahko približno $10^{300\,000\,000}$. Tako ogromno število ne bo šlo v običajne spremenljivke (npr. tipa `int`), pa tudi računanje s tolikšnimi števili bi bilo zelo počasno. Pomagamo si lahko z lepo lastnostjo ostanka po deljenju:

$$(a_1 \cdot a_2) \bmod k = ((a_1 \bmod k) \cdot (a_2 \bmod k)) \bmod k.$$

Z drugimi besedami, ostanek po deljenju zmnožka $a_1 a_2$ s k se nič ne spremeni, če od obeh faktorjev, a_1 in a_2, pred množenjem obdržimo le *njuna* ostanka po deljenju s k. Zato bi lahko v zanki, s katero računamo x, po vsakem množenju obdržali od zmnožka le ostanek po deljenju s k, pa na končni rezultat to ne bi vplivalo:

$x := a \bmod k$;
for $i := 1$ **to** n **do** $x := (2 \cdot x) \bmod k$;
return x;

Zdaj imamo v zanki ves čas opravka s prijetno majhnimi števili (x je bil pred množenjem že iz $\{0, \ldots, k-1\}$, po množenju z 2 torej dobimo rezultat iz $\{0, \ldots, 2k-2\}$, od tega pa takoj obdržimo le ostanek po deljenju s k).

Ima pa ta rešitev še vedno pomembno slabost, namreč zanko z n iteracijami. Naloga pravi, da je n lahko velik do 10^9, zanka s toliko iteracijami pa bi se izvajala neugodno dolgo. Spomnimo se na lepo lastnost potenciranja:

$$u^{v+w} = u^v \cdot u^w.$$

To pomeni, da bi lahko na primer u^{32} izračunali kot $u^{16} \cdot u^{16}$, pri čemer lahko seveda u^{16} izračunamo kot $u^8 \cdot u^8$, pred tem izračunamo u^8 kot $u^4 \cdot u^4$, še prej u^4 kot $u^2 \cdot u^2$ in čisto na začetku u^2 kot $u \cdot u$. Očitno lahko na ta način hitro računamo potence u-ja za tiste eksponente, ki so potence števila 2. V splošnem lahko to opažanje zapišemo takole: $u^{2^t} = u^{2^{t-1}+2^{t-1}} = u^{2^{t-1}} \cdot u^{2^{t-1}}$. Od u do u^{2^t} lahko pridemo s t zaporednimi kvadriranji.

Če nas zanima eksponent, ki ni potenca števila 2, ga lahko vedno zapišemo kot vsoto potenc števila 2 — ni treba drugega, kot da ga pretvorimo v dvojiški zapis. Na primer: $100 = 64 + 32 + 4 = 2^6 + 2^5 + 2^2$. Torej, če nas zanima u^{100}, moramo najprej izračunati u^{64}, ob tem spotoma dobimo tudi u^{32} in u^4 (in seveda u^{16}, u^8 in u^2, ki pa jih v nadaljevanju ne bomo več potrebovali), nato pa jih moramo le še zmnožiti: $u^{100} = u^{64} \cdot u^{32} \cdot u^6$.

Pri naši nalogi bomo vzeli $u = 2$, za eksponent pa n, tako da bomo lahko s pravkar dobljenim postopkom izračunali 2^n. Nazadnje moramo to vrednost le še pomnožiti z a. Ker nas bo na koncu zanimal le ostanek po deljenju s k, lahko tudi od potenc števila 2 med računanjem po vsakem množenju obdržimo le ostanek po deljenju s k.

$x := 1$; $y := 2$; $t := 0$;
while $2^t \le n$:
  (* *Invarianta:* $y = 2^{2^t} \bmod k$ *in* $x = 2^{n'} \bmod k$,

*pri čemer je n' spodnjih t bitov števila n (torej $n' = n \bmod 2^t$). *)*
if je bit t v dvojiškem zapisu števila n prižgan **then**
$x := (x \cdot y) \bmod k$;
$y := (y \cdot y) \bmod k$; $t := t + 1$;
$x := ((a \bmod k) \cdot x) \bmod k$;
return x;

Zapišimo rešitev še v C-ju. Spremenljivke t niti ne potrebujemo, če n v vsaki iteraciji zamaknemo za en bit v desno. Tisto, kar je bil v prvotni vrednosti n-ja bit t, je tako vedno pri roki v bitu 0 (najnižjem bitu); namesto pogoja $2^t \leq n$ pa je dovolj opazovati le to, ali je v n-ju sploh še kakšen prižgan bit:

```
#include <stdio.h>

int main()
{
   int a, n, k, x = 1, y = 2;
   FILE *f = fopen("bakterije.in", "rt");
   fscanf(f, "%d %d %d", &a, &n, &k); fclose(f);
   while (n > 0) {
      if (n & 1) x = (x * y) % k;
      y = (y * y) % k; n >>= 1; }
   x = ((a % k) * x) % k;
   f = fopen("bakterije.out", "wt");
   fprintf(f, "%d\n", x); fclose(f); return 0;
}
```

Naloge so sestavili: jezikovni tečaj, matura — Nino Bašić; vojna — Primož Gabrijelčič; disk — Boris Gašperin; prepovedani znaki, Gugl'bot — Miha Grčar; urejevalniška razdalja — Miha Grčar in Jure Ferlež; bakterije — Uroš Jovanovič; elektrika — Marjan Šterk; metulji — Mitja Trampuš; Smrkci — Miha Vuk; klepetalnica, nakupovanje — Klemen Žagar; komparatorji, društva — Janez Brank.

NASVETI ZA MENTORJE O IZVEDBI ŠOLSKEGA TEKMOVANJA IN OCENJEVANJU NA NJEM

[Naslednje nasvete in navodila smo poslali mentorjem, ki so na posameznih šolah skrbeli za izvedbo in ocenjevanje šolskega tekmovanja. Njihov glavni namen je bil zagotoviti, da bi tekmovanje potekalo na vseh šolah na približno enak način in da bi ocenjevanje tudi na šolskem tekmovanju potekalo v približno enakem duhu kot na državnem.—*Op. ur.*]

Tekmovalci naj pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom; ocenjevanje teh odgovorov poteka v vsakem primeru tako, da jih pregleda in oceni mentor (in ne npr. tako, da bi se poskušalo izvorno kodo, ki so jo tekmovalci napisali v svojih odgovorih, prevesti na računalniku in pognati na kakšnih testnih podatkih). Pri reševanju si lahko tekmovalci pomagajo tudi z literaturo in/ali zapiski, ni pa mišljeno, da bi imeli med reševanjem dostop do interneta ali do kakšnih datotek, ki bi si jih pred tekmovanjem pripravili sami. Čas reševanja je omejen na 180 minut.

Nekatere naloge kot odgovor zahtevajo program ali podprogram v kakšnem konkretnem programskem jeziku, nekatere naloge pa so tipa „opiši postopek“. Pri slednjih je načeloma vseeno, v kakšni obliki je postopek opisan (naravni jezik, psevdokoda, diagram poteka, izvorna koda v kakšnem programskem jeziku, ipd.), samo da je ta opis dovolj jasen in podroben in je iz njega razvidno, da tekmovalec razume rešitev problema.

Glede tega, katere programske jezike tekmovalci uporabljajo, naše tekmovanje ne postavlja posebnih omejitev, niti pri nalogah, pri katerih je rešitev v nekaterih jezikih znatno krajša in enostavnejša kot v drugih (npr. uporaba perla ali pythona pri problemih na temo obdelave nizov).

Kjer se v tekmovalčevem odgovoru pojavlja izvorna koda, naj bo pri ocenjevanju poudarek predvsem na vsebinski pravilnosti, ne pa na sintaktični. Pri ocenjevanju na državnem tekmovanju zaradi manjkajočih podpičij in podobnih sintaktičnih napak odbijemo mogoče kvečjemu eno točko od dvajsetih; glavno vprašanje pri izvorni kodi je, ali se v njej skriva pravilen postopek za rešitev problema. Ravno tako ni nič hudega, če npr. tekmovalec v rešitvi v C-ju pozabi na začetku **#include**ati kakšnega od standardnih headerjev, ki bi jih sicer njegov program potreboval; ali pa če podprogram main() napiše tako, da vrača **void** namesto **int**.

Pri vsaki nalogi je možno doseči od 0 do 20 točk. Od rešitve pričakujemo predvsem to, da je pravilna (= da predlagani postopek ali podprogram vrača pravilne rezultate), poleg tega pa je zaželeno tudi, da je učinkovita (manj učinkovite rešitve dobijo manj točk).

Če tekmovalec pri neki nalogi ni uspel sestaviti cele rešitve, pač pa je prehodil vsaj del poti do nje in so v njegovem odgovoru razvidne vsaj nekatere od idej, ki jih rešitev tiste naloge potrebuje, naj vendarle dobi delež točk, ki je približno v skladu s tem, kolikšen delež rešitve je našel.

Če v besedilu naloge ni drugače navedeno, lahko tekmovalčeva rešitev vedno predpostavi, da so vhodni podatki, s katerimi dela, podani v takšni obliki in v okviru takšnih omejitev, kot jih zagotavlja naloga. Tekmovalcem torej načeloma ni treba pisati rešitev, ki bi bile odporne na razne napake v vhodnih podatkih.

V nadaljevanju podajamo še nekaj nasvetov za ocenjevanje pri posameznih nalogah.

1. Vremenoslovje

- Če bi kakšna rešitev prebrala datoteko po enkrat za vsako postajo namesto enkrat za vse postaje, naj dobi največ 10 točk (od 20) (če je z njo drugače vse v redu).
- Če bi kakšna rešitev prebrala celotno zaporedje meritev v pomnilnik, naj dobi največ 17 točk (od 20).
- Če se rešitev zanaša na kakšne dodatne predpostavke o vrstnem redu podatkov (ki jih naloga ne zagotavlja) in zaradi teh predpostavk ne bi delovala pravilno, če podatki ne bi bili v tem vrstnem redu, naj se ji odbije 5 točk (npr. če predpostavi, da so podatki urejeni po številki postaje).
- V vhodni datoteki so postaje oštevilčene od 1 naprej, indeksi v tabelah pa so pri C-ju in mnogih sorodnih jezikih od 0 naprej; program mora torej ali pri branju podatkov od številk postaj odšteti 1 ali pa si alocirati tabelo 101 elementov namesto 100 elementov. Obe rešitvi naj se ocenjuje kot enako dobri. Če ima program tabelo 100 elementov z indeksi od 0 do 99, naslavlja pa jo z indeksi od 1 do 100, naj se takšni rešitvi odbije dve točki.
- Za primere, ko za neko postajo ni nobene meritve, je mogoče poskrbeti na različne načine; ena možnost je še ena tabela logičnih vrednosti (**bool** ali boolean), ki povedo, ali za neko postajo že imamo kakšno meritev ali ne, druga možnost pa je, da predpostavimo, da bodo vse meritve nad neko (dovolj nizko) vrednostjo (npr. -274 v naši rešitvi). Obe možnosti naj se ocenjuje kot enako dobri.
- Format izpisa pri tej nalogi ni posebej določen; dovolj dober je vsak izpis, pri katerem so pač našteti rezultati za vse postaje in pri katerem se jasno vidi, če za katero postajo ni bilo nobene meritve.
- V vhodni datoteki je v vsaki vrstici tudi podatek o dnevu, s katerim pri tej nalogi ni treba početi ničesar (razen tega, da ga pač preberemo). Če tekmovalčeva rešitev s tem podatkom vendarle kaj počne, vendar na način, ki ne škoduje pravilnosti rezultatov in ne povzroči kakšne bistvene neučinkovitosti, naj se ji zaradi tega točk ne odbija.

2. Krajšanje imena

- Nič hudega ni, če rešitev na koncu izpiše tudi znak za konec vrstice.
- Nič ni narobe, če si rešitev pripravi nize, ki jih bo morala izpisati, na malo manj učinkovit način, na primer `WriteLn(ime[1] + '.' + priimek[1] + '.')` namesto `WriteLn(ime[1], '.', priimek[1], '.')`.
- Vseeno je, ali rešitev (če je v C-ju ali kakšnem sorodnem jeziku) prenaša ime in priimek kot **const char*** ali kot **char***.

3. Registrske številke

- Glavno pri tej nalogi je, ali rešitev vsebuje pravilno logiko za preverjanje tega, ali se neka številka iz datoteke ujema z dano delno številko.

- Za morebitne napake pri branju vhodnih podatkov naj se ne odbije več kot kakšne tri točke, npr. če je rešitev pisana v C-ju in bere vhodne podatke s `fgets` in pozabi na to, da bo `fgets` pustil na koncu niza tudi znak `'\n'`, ali pa če pusti `fgets`ju premalo prostora (npr. je drugi parameter 100 namesto 101); tako ali tako pri mnogih drugih programskih jezikih ta problem sploh odpade.

- Za morebitne male neučinkovitosti, kot je npr. to, da rešitev prebrani niz pred izpisom razbije na dva niza (npr. v jezikih, ki imajo `string` kot standardni tip), naj se ne odbija točk.

- Če je v kakšnih jezikih mogoče preverjanje, ali se delna in konkretna registrska številka ujemata, izvesti enostavneje kot s preverjanjem znak za znakom, je tudi to sprejemljivo kot enako dobra rešitev (primer je naša rešitev v pythonu, ki uporablja pythonov modul za regularne izraze).

- Če bi rešitev po nepotrebnem prebrala vsa imena naenkrat v pomnilnik, naj dobi največ 17 točk; če bi povrhu tega tudi naredila toge predpostavke o tem, koliko imen bo (npr. da jih odlaga v tabelo fiksne velikosti, ne pa v tabelo, ki jo lahko dinamično povečuje, ali pa v seznam, povezan s kazalci), naj dobi največ 15 točk.

4. Največji nihaj navzdol

- Glavno pri tej nalogi je, ali ima rešitev časovno zahtevnost $O(n^2)$ ali le $O(n)$. Rešitve z zahtevnostjo $O(n^2)$ naj dobijo največ 10 točk.

- Za klic funkcije `Cena()` naj se predpostavi, da je poceni, torej ni nič narobe, če v iteraciji glavne zanke kličemo `Cena(j)` trikrat (kot to stori naša gornja rešitev) namesto npr. le enkrat (kar bi lahko storili, če bi si jo shranili v neko spremenljivko).

- Če rešitev pozabi vrniti rezultat v odstotkih (in ga vrne kot razmerje z intervala $[0, 1]$), naj se odbije eno točko.

- Če je rešitev sicer smiselno pravilna, vendar zaradi pomote pri znakih $<$ in $>$ računa največji nihaj navzgor ali kaj podobnega, naj se odbije največ tri točke.

- Če rešitev pomotoma kliče funkcijo `Cena` z argumenti od 0 do `StTrenutkov()` $- 1$ namesto od 1 do `StTrenutkov()`, naj se odbije največ eno točko.

5. Popravilo ograje

- Postopek je lahko opisan na različne načine — v naravnem jeziku, s psevdokodo, v kakšnem konkretnem programskem jeziku ipd. Načeloma so vse te možnosti enako dobre, glavno je, ali je postopek opisan dovolj jasno in natančno, da se vidi, da je tekmovalec prišel do rešitve in jo razumel.

- Pomembno je tudi, ali se iz odgovora vidi, da ima tekmovalec nekakšen argument za pravilnost svoje rešitve (in je ni zapisal le zato, ker se mu po občutku zdi, da je najbrž pravilna). To seveda ne pomeni, da pričakujemo formalen dokaz z indukcijo (in mogoče še s protislovjem), kakršne se ponavadi uporablja pri požrešnih algoritmih (recimo, da bi obstajala rešitev s še manj deskami; in potem z indukcijo dokažemo, da jo lahko predelamo v rešitev našega algoritma, ne da bi pri tem prenehali pokrivati vse poškodovane deščice; ali pa z indukcijo za vsak d dokažemo, da najbolj levih d desk naše rešitve pokrije vse tiste poškodovane deščice, ki jih pokrije tudi najbolj levih d desk optimalne rešitve). Radi pa bi, da se iz odgovora vidi, da tekmovalec razume, da najbolj leve nove deske ni treba postaviti tako, da bi se začela bolj levo od najbolj leve okvarjene deščice; in da je seveda pametno od vsake nove deske uporabiti čim daljši kos, da pokrije čim več okvarjenih deščic.
- Seveda je enako dobro, če rešitev pokriva ograjo od desne proti levi namesto od leve proti desni.
- Če je rešitev pravilna (v smislu, da dobi pravo minimalno število novih desk), vendar neučinkovita, naj dobi največ 10 točk, če je njena časovna zahtevnost polinomska (vendar ne linearna), in največ 5 točk, če je eksponentna.
- Če rešitev pozabi na robni primer, ko ni poškodovana nobena deščica, naj se točk zaradi tega ne odbija.

Težavnost nalog

Državno tekmovanje IJS 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 IJS
1. Vremenoslovje	lahka naloga v prvi skupini
2. Krajšanje imena	srednja naloga v prvi skupini
3. Registrske številke	težja naloga v prvi ali lahka v drugi skupini
4. Največji nihaj navzdol	srednja naloga v drugi ali lahka v tretji skupini
5. Popravilo ograje	težja naloga v drugi ali lažja v tretji skupini

Če torej na primer nek tekmovalec reši le prvo nalogo in del druge, 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.

Glede na to, da število udeležencev na državnem tekmovanju tudi v prejšnjih letih, ko šolskega tekmovanja še nismo imeli, ni bilo zelo visoko, 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.

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. Poleg nagrad obstajajo tudi pohvale, in sicer jih prejmejo tekmovalci, ki ustrezajo naslednjim trem pogojem: (1) tekmovalec ni dobil nagrade; (2) je boljši od vsaj polovice tekmovalcev v svoji skupini; in (3) je tekmoval v prvi ali drugi skupini in dobil vsaj 30 točk ali pa je tekmoval v tretji skupini in dobil vsaj 80 točk. Namen te novosti 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.

V tabelah so prejemniki nagrad označeni z „1“, „2“ in „3“ v prvem stolpcu, prejemniki pohval pa s „P“.

PRVA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1	1	Tadej Novak	2	Gimnazija Kranj	20	20	17	20	19	96
1		Anže Žitnik	4	ERSŠG Ljubljana	20	17	19	20	20	96
2	3	Jernej Anclin	3	PTERŠ Velenje	20	20	16	20	19	95
2	4	Sandi Majninger	2	II. gimnazija Maribor	20	14	20	20	20	94
3	5	Gašper Medved	2	ERSŠG Ljubljana	19	19	13	20	19	90
3	6	Ernest Beličič	2	ERSŠG Ljubljana	20	20	14	17	14	85
P	7	Rok Slamek	4	SPTŠ Murska Sobota	18	7	20	13	19	77
P	8	Tomaž Turner	4	SPTŠ Murska Sobota	17	10	13	20	15	75
P	9	Martin Šušteršič	1	Gimnazija Bežigrad + ZRI	15	20	16	19	0	70
P	10	Robert Pajek	3	ŠCC Gimnazija Lava	15	15	15	16	5	66
P		Denis Trstenjak	3	SPTŠ Murska Sobota	7	20	12	17	10	66
P	12	Urban Škvorc	1	Gimnazija Bežigrad + ZRI	10	5	13	18	19	65
P	13	Nikolaj Janko	3	ŠC Novo mesto	10	3	14	17	19	63
P	14	Aleš Celar	4	Škof. klas. gimn. Ljubljana	20	10	13	19	0	62
P	15	Jurij Volčič	4	Škof. klas. gimn. Ljubljana	15	17	10	7	11	60
P		Tadej Škvorc	1	Gimnazija Bežigrad + ZRI	10	15	14	14	7	60
P	17	Matevž Černe	4	ERSŠG Ljubljana	10	19	12	13	5	59
P	18	Tadej Kočnik	3	PTERŠ Velenje	18	20	10	10	0	58
P	19	Igor Malič	2	II. gimnazija Maribor	17	20	15	5	0	57
P	20	Matej Žebovec	4	Škof. klas. gimn. Ljubljana	15	12	10	15	0	52
P	21	Miha Rataj	3	ŠC Celje, SŠEK	10	3	11	7	19	50
P	22	Tadej Petreski	1	II. gimnazija Maribor	15	5	14	12	3	49
P	23	Aleš Razpotnik	2	ERSŠG Ljubljana	10	10	11	16	0	47
P	24	Dejan Erjavec	3	ŠC Novo mesto, SEŠTG	20	2	14	8	2	46

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
P	25	Tim Marinšek	4	Škof. klas. gimn. Ljubljana	15	20	0	10	0	45
P		Jože Kulovic	3	ŠC Novo mesto, SEŠTG	10	5	13	7	10	45
P	27	Jasna Urbančič	1	ZRI	10	10	10	9	3	42
P	28	Miha Novak	3	ERSŠG Ljubljana	10	7	13	1	7	38
P	29	Sandi Srkoč	4	SPTŠ Murska Sobota	10	2	16	3	5	36
P	30	Ariadna Štorman	1	Gimnazija Vič	7	15	13	0	0	35
	31	Milan Radakovič	4	ŠCC Gimnazija Lava	12	0	11	8	3	34
		Miha Možina	3	SERŠ Maribor	10	7	10	7	0	34
	33	Zoran Felbar	4	SPTŠ Murska Sobota	5	0	7	16	5	33
		Sven Cerk	1	Gimnazija Bežigrad	10	7	16	0	0	33
	35	Gašper Golob	2	ERSŠG Ljubljana	12	2	12	5	1	32
	36	Uroš Kolenko	3	ERSŠG Ljubljana	5	1	14	8	2	30
		Aljaž Frančič	1	II. gimnazija Maribor	17	3	10	0	0	30
		Gregor Grgurič	3	SPTŠ Murska Sobota	10	5	13	0	2	30
	39	Žiga Gosar	1	Gimnazija Vič	12	0	3	16	0	31
	40	Marko Kavaš	3	SPTŠ Murska Sobota	10	3	15	0	0	28
		Gregor Miklošič	1	II. gimnazija Maribor	7	7	10	0	4	28
		Gregor Šoln	4	ŠCC Gimnazija Lava	10	2	9	2	5	28
	43	Aljaž Blatnik	2	Gimnazija Vič	10	5	1	2	8	26
	44	Aljaž Jesenko	3	ŠCC Gimnazija Lava	5	3	0	10	7	25
	45	Andraž Možina	3	Gimnazija Vič	15	0	1	8	0	24
		Nicola Pinzani	1	Licej Fr. Prešeren, Trst	7	5	12	0	0	24
	47	Primož Mekuč	2	Škof. klas. gimn. Ljubljana	2	0	1	17	0	20
	48	Patrik Huber	3	SPTŠ Murska Sobota	5	0	11	0	0	16
	49	Mezin Prevolšek	1	II. gimnazija Maribor	8	0	7	0	0	15
		Tanis Kodrun	3	SŠEK Celje	10	3	2	0	0	15
	51	Grega Podbregar	4	ŠCC Gimnazija Lava	7	0	1	6	0	14
	52	Vid Kovačec	1	Gimnazija Vič	7	3	1	0	0	11
		Robi Pritržnik	2	PTERŠ Velenje	7	2	2	0	0	11
		Dominik Letnar	3	SPTŠ Murska Sobota	7	3	1	0	0	11
		Blaž Oven	2	Gimnazija Vič	0	1	4	6	0	11
	56	Nives Bricman	1	PTERŠ Velenje	0	0	3	0	5	8
	57	Katja Vrečar	2	PTERŠ Velenje	3	1	2	0	0	6
		Klemen Rahne	3	Gimnazija Vič	1	0	2	3	0	6
	59	Mojca Rozman	3	SPTŠ Murska Sobota	2	1	0	0	0	3
	60	Rok Kališnik	3	ERSŠG Ljubljana	0	1	0	0	0	1
		Vito Zakrajšek	1	II. gimnazija Maribor	0	1	0	0	0	1
	62	Maja Ovčar	2	PTERŠ Velenje	0	0	0	0	0	0

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1	1	Matjaž Leonardis	1	ZRI + Gim. Bežigrad	18	18	10	19	19	84
1	2	Nejc Saje	3	ŠC Novo mesto	20	15	20	10	17	82
2	3	Matija Rezar	3	Gimnazija Kranj	18	15	18	15	5	71
2	4	Igor Lalič	3	ZRI	20	18	18	14	0	70
3	5	Jure Slak	1	Gimnazija Vič	18	15	15	0	10	58
3	6	Jernej Muha	4	ŠCC Gimnazija Lava	15	9	2	15	15	56
P	7	Natan Žabkar	4	Gimnazija Vič	8	10	20	10	4	52
P	8	Gregor Lah	4	ŠC Novo mesto	5	14	0	4	13	36
P	9	Kristian Zupan	4	ŠC Novo mesto	8	5	0	12	13	38
	10	Gregor Cimerman	4	ERSŠG Ljubljana	15	8	6	0	8	37
	11	Peter Žužek	3	Gimnazija Vič	17	15	0	3	0	35
	12	Franci Kaker	4	ŠCC Gimnazija Lava	13	10	0	1	10	34
	13	Simon Vesel	4	ŠC Novo mesto	18	8	6	1	0	33
	14	Anže Novšak	3	ŠC Krško-Sevnica	5	5	6	2	8	26
	15	Jernej Legiša	4	Licej Fr. Prešeren, Trst	15	3	0	2	2	22
	16	David Požar	4	Licej Fr. Prešeren, Trst	8	2	4	0	4	18
	17	Nejc Zupan	3	Gimnazija Vič	0	2	6	0	2	10

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1	1	Nace Hudobivnik	4	Šk. kl. gim. Lj. + ZRI	100	100	80	0	100	380
1	2	Žiga Ham	3	ZRI	87	64	90	0	87	328
2	3	Rok Kralj	3	Gimnazija Vič + ZRI	94	40	71	0		205
2	4	Boris Mitrović	4	ŠC Krško-Sevnica	100	58		0		158
3	5	Andrej Slapnik	4	PTERŠ Velenje	100	37	0		0	137
3	6	Marko Zabreznik	4	PTERŠ Velenje	94	41				135
P	7	Primož Kranjec	4	ŠC Krško-Sevnica	100	34				134
P	8	Dejan Knez	4	ŠCC Gimnazija Lava	78	40	7	0		125
P	9	Mitja Rešek	4	ŠCC Gimnazija Lava	66	28				94
P	10	Gašper Tomažič	3	TŠC Nova Gorica	4	70				74
P	11	Jošt Stergar	3	Gimnazija Bežigrad		21				21
P	12	Blaž Sovdat	3	TŠC Nova Gorica	0	20				20
	13	Žiga Stopinšek	4	ŠCC Gimnazija Lava	8	0	10			18
		Mario Ogorelc	4	ŠC Krško-Sevnica	11	0	0	0	7	18
	15	Jure Gregorin	4	ERSŠG Ljubljana	4	7				11
	16	Tadej Logar	4	TŠC Kranj	0		10			10
	17	Jani Plesničar	4	TŠC Nova Gorica						0
		Demjan Vester	4	TŠC Kranj		0				0
		Matic Jesenovec	4	TŠC Kranj	0					0
		Matevž Ropret	3	TŠC Kranj						0
		Matjaž Črnko	3	SERŠ Maribor	0					0
		Matej Lakota	4	TŠC Nova Gorica	0			0		0
		Matevž Baloh	4	TŠC Nova Gorica			0			0
		Marko Herič	3	SERŠ Maribor	0					0
		Jaka Kramar	3	ŠC Krško-Sevnica		0				0

NAGRADE

Za nagrado so najboljši tekmovalci vsake skupine prejeli naslednjo strojno opremo in knjižne nagrade:

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Tadej Novak	64 GB USB flash disk R. B. Banks: *Ledene gore, padajoče domine*
1	1	Anže Žitnik	8 GB iPod nano R. B. Banks: *Ledene gore, padajoče domine*
1	2	Jernej Anclin	1 TB zunanji disk R. B. Banks: *Ledene gore, padajoče domine*
1	2	Sandi Majninger	750 GB zunanji disk Wilson, Watkins: *Uvod v teorijo grafov*
1	3	Gašper Medved	miška Razer Boomslang Collector's Edition 2007 Wilson, Watkins: *Uvod v teorijo grafov*
1	3	Ernest Beličič	miška Razer Boomslang Collector's Edition 2007 Wilson, Watkins: *Uvod v teorijo grafov*
2	1	Matjaž Leonardis	8 GB iPod nano S. S. Skiena: *The Algorithm Design Manual*
2	1	Nejc Saje	22" monitor S. S. Skiena: *The Algorithm Design Manual*
2	2	Matija Rezar	1 TB zunanji disk J. Strnad: *Mala zgodovina vesolja*
2	2	Igor Lalič	1 TB zunanji disk J. Strnad: *Mala zgodovina vesolja*
2	3	Jure Slak	750 GB zunanji disk M. Danesi: *Paradoks lažnivca in hanojski stolpi*
2	3	Jernej Muha	miška Razer Boomslang Collector's Edition 2007 M. Danesi: *Paradoks lažnivca in hanojski stolpi*
3	1	Nace Hudobivnik	16 GB iPod nano Cormen *et al.: Introduction to algorithms* K. Devlin: *Nova zlata doba matematike*
3	1	Žiga Ham	8 GB iPod nano Cormen *et al.: Introduction to algorithms* K. Devlin: *Nova zlata doba matematike*
3	2	Rok Kralj	1 TB zunanji disk Cormen *et al.: Introduction to algorithms* K. Devlin: *Nova zlata doba matematike*
3	2	Boris Mitrović	64 GB USB flash disk M. Guillemot: *Zgodovina matematike, zgodbe o problemih* J. R. Weeks: *Oblika prostora*
3	3	Andrej Slapnik	1 TB zunanji disk M. Guillemot: *Zgodovina matematike, zgodbe o problemih* J. R. Weeks: *Oblika prostora*
3	3	Marko Zabreznik	1 TB zunanji disk M. Danesi: *Paradoks lažnivca in hanojski stolpi* J. R. Weeks: *Oblika prostora*
Tekmovanje programov — Robot v labirintu			
		Rok Kralj	750 GB zunanji disk S. S. Skiena: *The Algorithm Design Manual*

Poleg tega je vsak od nagrajencev prejel tudi izvod knjige *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004* (v dveh zvezkih, IJS, 2006).

SODELUJOČE ŠOLE IN MENTORJI

Druga gimnazija Maribor — Matej Urbas, Mirko Pešec

Elektrotehniško-računalniška strokovna šola in gimnazija (ERSŠG) Ljubljana
Marko Kastelic, Matjaž Kodela, Nataša Makarovič, Darjan Toth

Gimnazija Bežigrad, Ljubljana — Andrej Šuštaršič

Gimnazija Kranj — Matevž Jekovec

Gimnazija Vič — Klemen Bajec

Srednja elektro-računalniška šola Maribor (SERŠ)
Vida Motaln, Slavko Nekrep, Manja Sovič Potisk

Srednja poklicna in tehniška šola (SPTŠ) Murska Sobota
Simon Horvat, Karel Maček, Milan Petrijan

Škofijska klasična gimnazija, Ljubljana — Helena Medvešek

Šolski center Celje, Splošna in strokovna gimnazija Lava
Borut Slemenšek

Šolski center Celje, Srednja šola za elektrotehniko in kemijo (SŠEK)

Šolski center Krško-Sevnica — Svetlana Novak, Andrej Peklar

Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija
Mile Božič, Tomaž Ferbežar, Robert Jakše, Ivan Slinkar, Simon Vovk

Šolski center Velenje, Poklicna in tehniška elektro in računalniška šola (PTERŠ)
Nedeljko Grabant, Gregor Hrastnik, Miran Zevnik

Tehniški šolski center (TŠC) Kranj — Gabrijela Kranjc

Tehniški šolski center (TŠC) Nova Gorica
Slobodan Marčetič, Boštjan Vouk

Zavod za računalniško izobraževanje (ZRI), Ljubljana
Jelko Urbančič

Znanstveni licej France Prešeren, Trst — Valentina Busechian

TEKMOVANJE PROGRAMOV — ROBOT V LABIRINTU

Podobno kot v prejšnjih letih smo tudi letos organizirali tekmovanje programov. Opis naloge smo objavili novembra 2008 skupaj z razpisom za tekmovanje v znanju, tekmovalci pa so imeli čas do 20. marca 2009 (teden dni pred tekmovanjem), da pošljejo svoje programe. Letošnja naloga je od tekmovalcev zahtevala, da napišejo logiko za krmiljenje robotka po labirintu. Pri tem se mora robot izogibati oviram, omejena je tudi njegova vidljivost, njegov cilj pa je čim hitreje priti do izhoda. Za razliko od prejšnjih let tokrat programi ne tekmujejo neposredno drug proti drugemu, pač pa se spopadajo vsak s svojim labirintom neodvisno od ostalih programov (torej to, kar v labirintu počne en program, nič ne vpliva na druge).

Labirint je predstavljen s karirasto mrežo, pri čemer je vsako polje lahko prosto (prehodno) ali pa zazidano (neprehodno). Polja na zunanjem robu mreže (torej v najbolj levem in najbolj desnem stolpcu ter v najbolj zgornji in najbolj spodnji vrstici) so malo drugačna od ostalih: prosto polje na zunanjem robu imenujemo tudi izhodno polje; zazidano polje na zunanjem robu pa imenujemo zunanji zid.

Na prostih poljih (in tudi na izhodnih poljih) lahko stojijo kocke, vendar največ ena kocka na vsakem takem polju. Začetno stanje labirinta je vedno tako, da nobena kocka ne stoji na izhodnem polju.

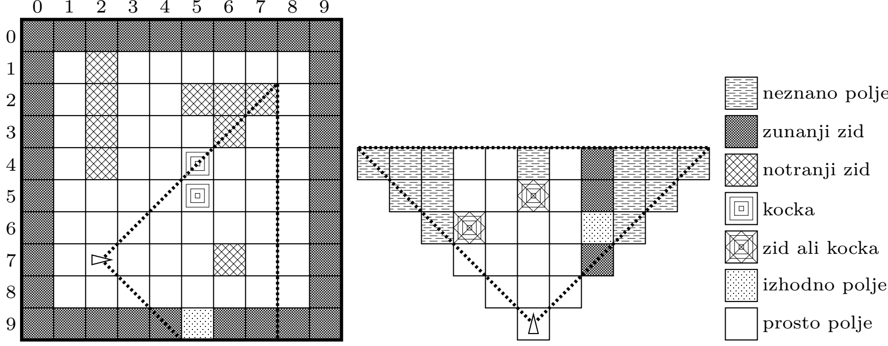
Robot v vsakem trenutku stoji na enem od polj in gleda v eno od štirih možnih smeri: gor, dol, levo ali desno. Na zazidana polja se ne more premakniti, enako tudi ne na polja, na katerih stojijo kocke. Pač pa lahko kocko pobere, če je le-ta stala na polju neposredno pred robotom. Pobrano kocko robot drži nad sabo, tako da ga nič ne ovira pri gledanju in pri gibanju. Kocko lahko robot odloži na polje neposredno pred sabo, če je to polje prosto in če na njem še ne stoji kakšna druga kocka.

Robotovi senzorji mu vračajo le omejene podatke o njegovi okolici. Robot „vidi“ le v smeri 45 stopinj levo in desno od smeri, v katero je trenutno obrnjen. Vidi le polja, ki so največ v enot pred njim (število v dobi podano ob začetku igre). Če na nekem praznem ali izhodnem polju stoji kocka, je ta kocka za robotove senzorje nerazločljiva od (nezunanjega) zidu. Pač pa robotovi senzorji ločijo zunanje zidove od nezunanjih (ali od polj, na katerih stojijo kocke) in ločijo izhodna polja od navadnih prostih polj (kakršna stojijo v notranjosti labirinta).

Če na nekem polju stoji zid ali pa kocka, nam to polje zakriva pogled na polja, ki stojijo za njim. Če je neko polje v celoti zakrito s takšnimi zidovi in/ali kockami, ga robot ne vidi. Primer labirinta in robotovega vidnega polja kaže slika na str. 117.

Možne poteze robota so torej korak naprej, zasuk za 90 ali 180 stopinj levo ali desno ter pobiranje in odlaganje kocke. Vsaka poteza ima svojo **ceno**, in sicer je cena vsake poteze 1, razen če robot nosi kocko in se premika naprej ali se obrača na mestu; te poteze imajo ceno 2. Cena zaporedja potez je definirana kot vsota cen posameznih potez v njem. Naloga robota je čim ceneje priti iz labirinta; igra se konča, ko robot stopi na kakšno od izhodnih polj. Igra se konča tudi, če robot naredi neveljavno potezo (npr. se zaleti v zid ali kocko) ali pa prekorači omejitve glede števila potez ali glede porabe procesorskega časa. Pobiranje kocke je veljavna poteza tudi, če je pred robotom zid in ne kocka, le da v tem primeru pobiranje pač ne uspe (igralni strežnik robota o tem tudi obvesti).

Za **testiranje** smo pripravili nabor 31 labirintov, od katerih sta bila dva velika 49×49 polj, ostali pa 29×29 polj ali manjši. Omejitev porabe procesorskega časa je



Leva slika kaže primer labirinta velikosti 10×10; robot je na položaju $(2, 7)$ in je obrnjen proti desni. Pikčasta črta označuje vidno polje robota, če znaša vidljivost $v = 5$ enot. Desna slika kaže, kaj v tej situaciji zaznava robot. Nekaterih polj v vidnem polju ne zaznava, ker mu kocke in zidovi zakrivajo pogled nanje (označena so kot „neznana polja“), poleg tega pa ne loči med kockami in notranjimi zidovi. Kocke v celici $(4, 5)$ robot ne vidi zato, ker polovica te celice leži zunaj vidnega polja (pikčastega trikotnika), na drugo polovico pa mu zakriva pogled kocka na celici $(5, 5)$.

---

bila 10 000 s na poljih 49×49 in 1000 s na manjših poljih, število robotovih potez pa je bilo omejeno na 100 000. Te omejitve so bile mišljene kot zelo darežljive in tako se je pokazalo tudi v praksi, saj nobeden od prejetih programov na nobenem od testnih labirintov ni porabil več kot dobro minuto časa, na večini labirintov pa še precej manj (tja do deset sekund, pogosto celo manj kot sekundo). Najdaljša veljavna pot je imela 14 856 potez; do 100 000 potez so prišli roboti le v primerih, ko so se zaciklali (dobro napisana rešitev se sicer nikoli ne zacikla, saj so bili testni primeri sestavljeni tako, da je pot do izhoda vedno obstajala).

Točkovanje je pri vsakem labirintu relativno glede na najboljše prejeto zaporedje potez za ta labirint. Če tekmovalec pri nekem labirintu ni pripeljal robota do izhoda (npr. ker je naredil neveljavno potezo ali pa prekoračil časovno omejitev), ne dobi pri tem labirintu nobenih točk. Sicer pa, če je njegov robot opravil zaporedje potez s skupno ceno c, najboljše znano zaporedje potez pri tem labirintu pa ima skupno ceno $c^\star$, bo tekmovalec pri tem labirintu dobil $c^\star/c$ točk. Na koncu točke tekmovalca seštejemo po vseh labirintih.

Rezultati

Čeprav je bilo za nalogo še kar nekaj zanimanja, smo na koncu prejeli rešitve le od treh tekmovalcev. Težava je bila verjetno v tem, da je za kakršno koli netrivialno rešitev pri tej nalogi potrebnega ne tako malo dela — npr. vsaj toliko, da zna program upoštevati podatke, ki mu jih igralni strežnik pošilja o tem, kaj robot vidi. To je mogoče odvrnilo nekatere morebitne tekmovalce od sodelovanja.

Rahlo neugodno pri naši nalogi je tudi to, da je do neke mere odvisna od sreče — robot je nagrajen za to, da pride iz labirinta čim hitreje, vendar pred seboj nima celotne slike labirinta, zato npr. ob prihodu v razvejišče poti ne more zanesljivo vedeti, v katero smer naj nadaljuje, da bo lažje prišel do izhoda. Do neke mere je torej prisiljen ugibati in to se mu lahko izide bolj ali manj srečno. Vpliv sreče na

testiranje bi lahko zmanjšali, če bi močno povečali število testnih labirintov, tega pa si nismo upali, da ne bi ocenjevanje trajalo predolgo (saj smo dali precej darežljive omejitve glede porabe procesorskega časa). Na srečo so razlike med prispelimi tremi rešitvami tolikšne, da ni dvoma o tem, da je to res pravi vrstni red.

Mesto	Ime	Šola	Točke	Rešeni labirinti
1	Rok Kralj	Gimnazija Vič	28,4	31
2	Matjaž Leonardis	ZRI in Gimn. Bežigrad	14,7	18
3	Gašper Medved	ERSŠG Ljubljana	4,4	9

Zadnji stolpec pove, pri koliko od 31 testnih labirintov je program uspešno prišel do izhodnega polja. Prvemu je to uspelo pri vseh labirintih, ostalima dvema pa ne vedno, pri čemer so težave večinoma nastopile zaradi tega, ker se je tekmovalčev program, ki je vodil robota, sesul. Do zaciklanja je prišlo le pri tretjeuvrščenem robotu in to na šestih testnih primerih.

Rešitev

Robot naj v pomnilniku vzdržuje tabelo s podatki o tem, kaj trenutno ve o labirintu. Recimo, da je labirint velik $w \times h$ celic; ker ne vemo, kje v njem se robot na začetku nahaja in v katero smer je obrnjen, je koristno imeti približno dvakrat večjo tabelo, veliko $(2n-1) \times (2n-1)$ celic za $n = \max\{w, h\}$, robota pa v njej na začetku postavimo v sredino in recimo, da je obrnjen navzgor. Tako bo v vse smeri naokoli še dovolj prostora za podatke o labirintu, ne glede na to, kje v njem se robot na začetku nahaja. Če robot v resnici na začetku ni bil obrnjen navzgor, bo v naši tabeli nastajala zasukana podoba pravega labirinta, kar pa na obnašanje našega robota ne bo nič vplivalo, saj so vse poteze opisane relativno glede na robotov trenutni položaj in orientacijo.

Po vsaki potezi lahko popravimo podatek o robotovem položaju in orientaciji (ker vemo, kakšno potezo smo naredili), poleg tega pa dobimo od nadzornega programa podatke o tem, kaj robot v novem položaju vidi, in te podatke moramo zdaj vpisati v našo tabelo v pomnilniku. V tej tabeli se poleg običajnih tipov polj (prazno polje, zid, izhodno polje, zunanji zid, prazno polje s kocko, izhodno polje s kocko) pojavljata še dva posebna tipa polj: neznano polje (za polja, ki jih robot še ni videl) in „zid ali kocka“ (če smo videli, da je na tem polju zid ali kocka, nismo pa je še poskusili pobrati in torej ne vemo, za kaj od tega dvojega gre).

Obnašanje robota lahko v grobem opišemo z nekaj strateškimi pravili:

1. Če že vemo za kakšno izhodno polje, pojdimo po najcenejši poti do njega.

2. Sicer, če vemo za kakšno polje, s katerega bi videli kakšno polje, ki je doslej še neznano, pojdimo po najcenejši poti do njega.

3. Sicer, če vemo za kakšno polje p tipa „zid ali kocka“ (torej vemo, da je tam ali zid ali kocka, ne vemo pa še, za kaj od tega dvojega gre), pojdimo po najcenejši poti do enega od njegovih sosednjih prostih polj (štejejo tudi prosta polja, na katerih trenutno stojijo kocke), se obrnimo proti p in poskušajmo pobrati kocko z njega.

Če pri eni od teh treh točk najdemo več možnih ciljnih polj, vzemimo med njimi tisto, do katerega je mogoča najcenejša pot med vsemi. Ideja teh pravil je torej v tem, da robot raziskuje labirint, dokler ne opazi kakšnega izhodnega polja, takrat pa gre takoj proti njemu. Pri raziskovanju labirinta smo dali prednost gibanju po prostih poljih, razlikovanje med zidovi in kockami pa smo pustili za konec. Motivacija za to je, da ima marsikateri labirint veliko zidov, pa malo ali nič kock in bi bilo neumno izgubljati čas s poskusi pobiranja vseh zidov, namesto da bi raje prehodili labirint po prostih poljih in pri tem našli tudi kak izhod. Seveda pa niso vsi labirinti taki in pri kakšnih bi bilo bolje, če bi vrstni red pravil (2) in (3) v gornjem spisku zamenjali.

Ali se lahko zgodi, da v kakšnem trenutku z nobenim od teh treh pravil ne moremo nadaljevati robotovega gibanja? To bi pomenilo, da je robot v celoti raziskal ves tisti del labirinta, ki ga lahko doseže iz svojega začetnega položaja, pa pri tem ni našel izhoda. Tak robot je torej zazidan v labirint in ne bo mogel iz njega; opis naloge na srečo pravi, da se pri naših testnih primerih to ne bo dogajalo, zato se nam s to možnostjo ni treba ukvarjati.

Oglejmo si še, kako bo robot iskal najkrajše poti do drugih polj. Predstavljajmo si prostor stanj, v katerem posamezno stanje podaja položaj (x, y) in orientacijo s robota v labirintu. Med temi stanji definirajmo povezave: od (x, y, s) do (x, y, s') imejmo povezavo dolžine 1 (kar ponazarja, da se lahko robot obrne na mestu); če je (x', y') sosed polja (x, y) v smeri s in je (x', y') prosto, imejmo povezavo od (x, y, s) do (x', y', s) dolžine 1. Če pa na (x', y') stoji kocka, je premik od (x, y, s) na polje (x', y') sestavljen iz več potez. Vedno je na primer mogoče zaporedje štirih potez: robot stoji na (x, y), gleda v smeri s in pobere kocko s polja (x', y'); v drugi potezi stopi na (x', y'); v tretji se obrne za 180 stopinj; v četrti odloži kocko na (x, y). Tako smo prišli v stanje $(x', y', s + 180°)$ s štirimi potezami za skupno ceno 6; v našem prostoru stanj torej imejmo v takem primeru povezavo dolžine 6 od (x, y, s) do $(x', y', s + 180°)$. Še eno možno zaporedje potez lahko opazimo v primeru, da ima (x, y) kakšno prosto sosednje polje (x'', y''); v tem primeru lahko v stanju (x, y, s) poberemo kocko s polja (x', y'), se obrnemo proti (x'', y''), odložimo kocko na (x'', y''), se obrnemo nazaj proti (x', y') in stopimo na (x', y'); prišli smo v stanje (x', y', s) s petimi potezami za skupno ceno 6.

Na opisani način smo dobili graf, v katerem lahko s kakšnim od primernih algoritmov za iskanje najkrajših poti (na primer z Dijkstrovim) poiščemo najkrajše poti od trenutnega stanja robota do vseh ostalih stanj, vsaka taka pot v grafu pa ustreza tudi neki možni poti robota v labirintu.

Ko si izberemo, kakšen bo naslednji cilj našega robota, lahko najkrajšo pot po pravkar opisanem prostoru stanj predelamo v zaporedje polj našega labirinta, ki jih bomo morali obiskati. Premik s polja na polje pa sestavimo (po enakem razmisleku kot zgoraj) iz ene ali več robotovih potez, odvisno od tega, ali se mora robot najprej zasukati in ali mora odmakniti kocko s ciljnega polja.

ANKETA

Tekmovalcem vseh treh skupin smo na tekmovanju skupaj z nalogami razdelili tudi naslednjo anketo. Rezultati ankete so predstavljeni na str. 124–130.

Letnik: ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Kako si izvedel za tekmovanje?
☐ od mentorja ☐ na spletni strani (kateri? ____________)
☐ od prijatelja/sošolca ☐ drugače (kako? ____________)
Kolikokrat si se že udeležil kakšnega tekmovanja iz računalništva pred tem tekmovanjem? ____________
Katerega leta si se udeležil prvega tekmovanja iz računalništva? ____________
Najboljša dosedanja uvrstitev na tekmovanjih iz računalništva (kje in kdaj)? ____________

Koliko časa že programiraš? ____________
Kje si se naučil? ☐ sam ☐ 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 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 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 v programu uporabiti naslednje podatkovne strukture:

Drevo	☐ da	☐ ne
Hash tabela (asociativna tabela)	☐ da	☐ ne
S kazalci povezan seznam (linked list)	☐ da	☐ ne
Sklad (stack)	☐ da	☐ ne
Vrsta (queue)	☐ da	☐ ne

Ali bi znal 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++ in pythonu.[16]
— 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, da bi objavljali deklaracije (tudi) v kakšnem drugem programskem jeziku? Če da, v katerem? ____________________

V rešitvah nalog trenutno objavljamo izvorno kodo v pascalu in C-ju.
— Ali razumeš kakšnega od teh jezikov dovolj dobro, da si lahko kaj pomagaš z izvorno kodo v naših rešitvah? ☐ da ☐ ne
— Ali bi raje videl, da bi izvorno kodo rešitev pisali v kakšnem drugem jeziku? Če da, v katerem? ____________________

[Le pri 3. skupini.] Doslej smo v tretji skupini podpirali reševanje nalog v pascalu, C, C++, C# in javi. 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 prebrati kakšno celo število in kakšen niz iz standardnega vhoda ali pa ju zapisati na standardni izhod?	☐	☐	☐
Ali bi znal 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	☐	☐	☐

[16] Letos smo v resnici objavili deklaracije tudi v javi, vendar smo to pozabili omeniti v anketnem vprašanju.

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++: &, |, ^, ~) ☐ ☐ ☐
Operatorja **shl** in **shr** (v C/C++: <<, >>) ☐ ☐ ☐
[če pišeš v C++] razred map iz STLa ☐ ☐ ☐
[če pišeš v C++] razred priority_queue iz STLa ☐ ☐ ☐

[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?
☐ nisem rešil, ker mi je zmanjkalo časa za reševanje
☐ nisem rešil, ker mi je zmanjkalo volje za reševanje
☐ nisem rešil, ker mi je zmanjkalo znanja za reševanje
☐ rešil sem jo le delno, ker mi je zmanjkalo časa za reševanje
☐ rešil sem jo le delno, ker mi je zmanjkalo volje za reševanje
☐ rešil sem jo le delno, ker mi je zmanjkalo znanja za reševanje
☐ rešil 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 raje: ☐ več časa ☐ manj časa ☐ časa je bilo ravno prav
Bi imel raje: ☐ več nalog ☐ manj nalog ☐ nalog je bilo ravno prav

Kakršne koli druge pripombe in predlogi. Kaj bi spremenil(a), popravil(a), odpravil(a), ipd., da bi postalo tekmovanje zanimivejše in bolj privlačno? ______________________

____

Kaj ti je bilo pri tekmovanju všeč? ______________________________

____

Kaj te je najbolj motilo? ______________________________________

____

Če imaš kaj vrstnikov, ki se tudi zanimajo za programiranje, pa se tega tekmovanja niso udeležili, kaj bi bilo po tvojem mnenju treba spremeniti, da bi jih prepričali k udeležbi?

____

____

Poleg tekmovanja bi radi tudi v preostalem delu leta organizirali razne aktivnosti, ki bi vas zanimale, spodbujale in usmerjale pri odkrivanju računalništva. Prosimo, da nam pomagate izbrati aktivnosti, ki vas zanimajo in bi se jih zelo verjetno udeležili.

Udeležil bi se oz. z veseljem bi spremljal:

- ☐ izlet v kak raziskovalni laboratorij v Evropi (po možnosti za dva dni)
- ☐ poletna šola računalništva (1 teden na IJS, spanje v dijaškem domu)
- ☐ poletna praksa na IJS
- ☐ predstavitve novih tehnologij (.NET, mobilni portali, programiranje „vgrajenih računalnikov“, strojno učenje, itd.) (1× mesečno)
- ☐ predavanja o algoritmih in drugih temah, ki pridejo prav na tekmovanju (1× mesečno)
- ☐ reševanje tekmovalnih nalog (naloge se rešuje doma in bi bile delno povezane s temo, predstavljeno na predavanju; rešitve se preveri na strežniku) (1× mesečno)
- ☐ tvoji predlogi: ______________________________________

  ____

Vesel bi bil pomoči pri:

- ☐ iskanju štipendije
- ☐ iskanju podjetij, ki dijakom ponujajo njim prilagojene poletne prakse in druge projekte, kjer se ob mentorstvu lahko veliko naučijo.

Ali si pri izpolnjevanju ankete prišel do sem? ☐ da ☐ ne

Hvala za sodelovanje in lep pozdrav! Tekmovalna komisija

REZULTATI ANKETE

Anketo je izpolnilo 55 tekmovalcev prve skupine, 14 tekmovalcev druge in 24 tekmovalcev tretje. Vprašanja so bila pri letošnji anketi enaka kot lani.

Mnenje tekmovalcev o nalogah

Tekmovalce smo spraševali: kako zahtevna se jim zdi posamezna naloga; ali se jim zdi, da jim vzame preveč časa; ali je besedilo primerno dolgo in razumljivo; ali se jim zdi naloga zanimiva; ali so jo rešili (oz. zakaj ne); in katera naloga jim je bila najbolj/najmanj všeč.

Rezultate vprašanj o zahtevnosti nalog kažejo grafi na str. 125. 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. Če pri vsaki nalogi pogledamo povprečje mnenj o zahtevnosti te naloge (1 = prelahka, 3 = primerna, 5 = pretežka) in vzamemo povprečje tega po vseh petih nalogah, dobimo: 3,56 v prvi skupini (lani 3,57; predlani 3,18), 3,46 v drugi skupini (lani 3,62 lani; predlani 3,36) in 3,92 v tretji (lani 3,74; predlani 3,72). Komisija sicer pri pripravi nalog ni imela občutka, da so naloge letos kaj težje kot ponavadi (niti ni bil njen namen, da bi bile).

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 nekaj korelacije, vendar je šibka ($R^2 = 0{,}40$).

Podobno kot ponavadi se je tudi letos pokazalo, da so se zdele tekmovalcem težke predvsem tiste bolj nestandardne naloge, na primer 1.5 (železnica — „realnočasovna“ naloga), 2.5 (**break** considered harmful — kako preoblikovati program v skladu z omejitvami naloge) in 3.5 (EPS — ki zahteva rešitev v jeziku, ki ga tekmovalci od prej najbrž niso poznali). Precej tekmovalcem so tuje tudi operacije z biti (naloga 2.4, UTF-5). Zelo pa nas je presenetilo, da se jim je zdela težka naloga 1.4 (Smrkci).

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 126. Nad razumljivostjo besedil ni veliko pripomb (malenkost več kot lani jih sicer je); najtežje razumljivi sta se jim zdeli nalogi 2.4 (UTF-5) in 3.5 (EPS). To je verjetno delno krivda besedila, delno pa tega, da nalogi govorita o stvareh, ki tekmovalcem niso preveč domače (operacije na bitih, skladovni model računanja). Podobna opažanja smo imeli že pri lanski anketi.

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni; naloge z daljšimi besedili so sicer dobile nekaj več pripomb, češ da je besedilo predolgo, vendar je včasih pač težko na kratko povedati dovolj podrobnosti, da naloga ni preveč dvoumna.

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju v povprečju celo malenkost višje od lanskih in so pri vseh skupinah približno enake. Najbolj dolgočasni sta se jim zdeli nalogi 1.1 (kolikokrat najmanjši) in 1.3 (citati). Mogoče je res, da je v prvi skupini težje dajati zanimive naloge, ker smo pri sestavljanju omejeni s tem, da morajo biti naloge dovolj lahke.

Pripomb, da bi naloga vzela preveč časa, je bilo precej zlasti v tretji skupini (v prvih dveh pa malo, še manj kot lani), najbolj pri nalogah 3.3 (otoki), 3.4 (strupi)

	Kako težka se ti je zdela?	Povprečna ocena zahtevnosti	Delež tekmovalcev, ki so dobili > 0 točk	Povp. in mediana števila točk po vseh tekmovalcih	Povp. in mediana po vseh tekmovalcih z > 0 točkami
1.1. Kolikokrat najm.		2,90			
1.2. Označ. kovancev		3,43			
1.3. Citati		3,27			
1.4. Smrkci		3,95			
1.5. Železnica		4,23			
2.1. Soglas. podnizi		2,83			
2.2. Kje sem to posnel?		3,62			
2.3. Avtocesta		3,25			
2.4. UTF-5		3,83			
2.5. **break**		3,75			
3.1. Undo		3,29			
3.2. Žaba		3,47			
3.3. Otoki		4,07			
3.4. Strupi		4,31			
3.5. EPS		4,45			

Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

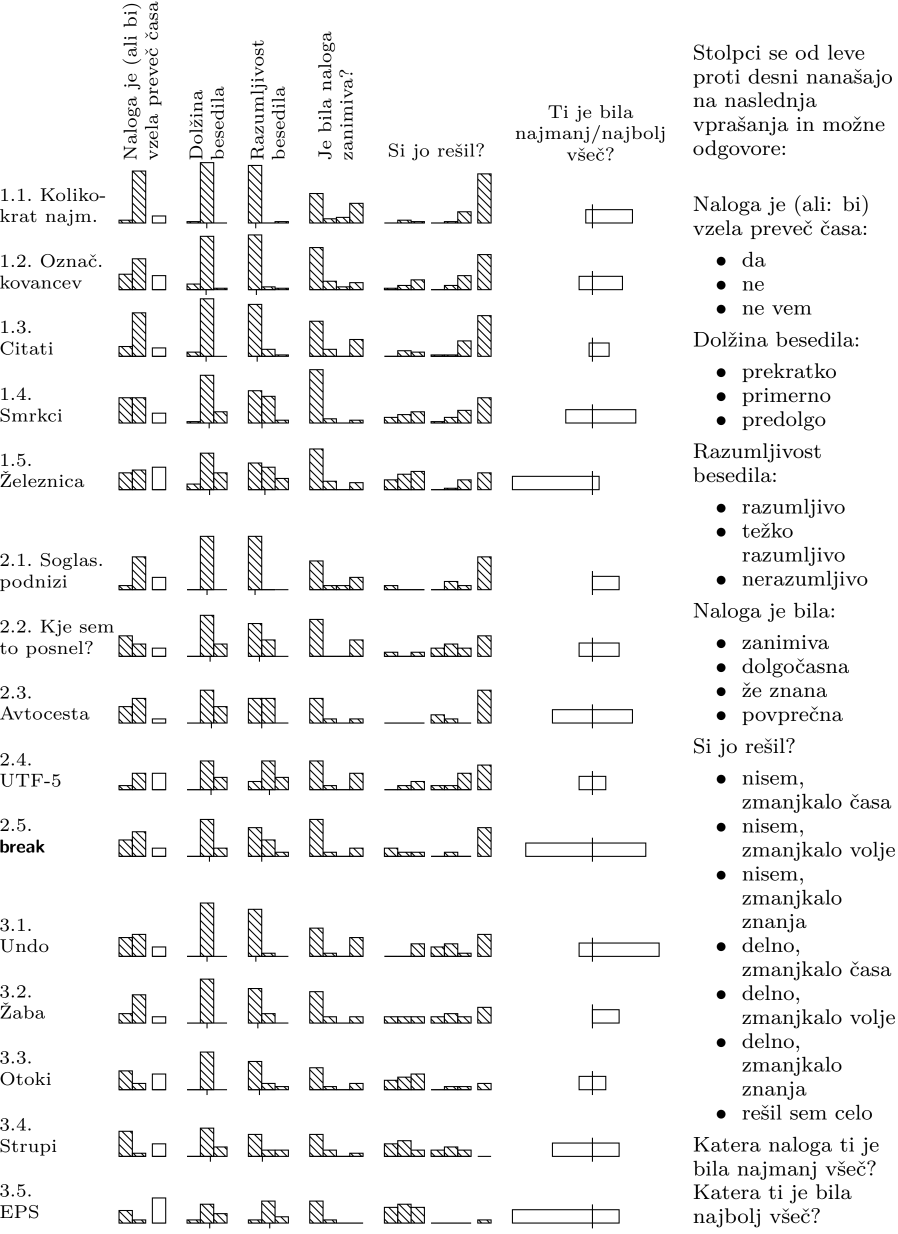
Pomen stolpcev v vsaki vrstici:

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?

in 3.5 (EPS). Dejstvo je, da je pri nalogah 3.3 in 3.4 z rešitvijo nekaj več dela, pri 3.5 pa ne nujno (pač pa jo naredi bolj zamudno dejstvo, da zahteva rešitev v jeziku, ki ga tekmovalci od prej najbrž ne poznajo). Po drugi strani so najboljši tekmovalci v tretji skupini dobro rešili tri ali štiri naloge, tako da nabor nalog kot celota le ni bil izrazito preveč zamuden.

Glasovi o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, kažejo več nasprotujočih si trendov. Mnogim tekmovalcem so najljubše tiste naloge, ki so najlažje, npr. 1.1 (kolikokrat najmanjši) in 3.1 (undo); včasih jih navduši predvsem zgodbica (npr. ker v nalogi 1.4 nastopajo Smrkci). Naloge, ki so jim po vsebini nekaj novega in tujega, pa so včasih zelo nepriljubljene, npr. 1.5 (železnica) in 3.5 (EPS), ali pa dobijo veliko glasov v obeh kategorijah, npr. 2.5 (**break** considered harmful).

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.

	Prva skupina	Druga skupina	Tretja skupina
priority_queue v C++	0 %	0 %	19 %
map v C++	2 %	20 %	18 %
zamikanje s **shl**, **shr**	24 %	29 %	41 %
operatorji na bitih	42 %	57 %	59 %
strukture	43 %	50 %	68 %
naštevni tipi	18 %	29 %	45 %
gnezdenje zank	78 %	86 %	100 %
zanka **while**	88 %	100 %	100 %
zanka **for**	92 %	100 %	100 %
kazalci	26 %	14 %	41 %
rekurzija	31 %	79 %	73 %
podprogrami	64 %	93 %	91 %
več-d tabele (**array**)	38 %	57 %	81 %
2-d tabele (**array**)	66 %	100 %	84 %
1-d tabele (**array**)	78 %	100 %	95 %
delo z datotekami	69 %	79 %	95 %
std. vhod/izhod	80 %	93 %	95 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

Rezultati pri vprašanjih o programerskem znanju so zelo podobni lanskim; zanimiv upad znanja je opaziti pri vprašanju o uporabi kazalcev, kar si mogoče lahko razložimo s tem, da je bilo na letošnjem tekmovanju več ljudi, ki delajo v javi. Stvari, ki jih poznajo slabše, so na splošno približno iste kot prejšnja leta: rekurzija, kazalci, naštevni tipi in operatorji na bitih. Algoritme za urejanje in zapis z velikim O pozna letos v 2. in 3. skupini manj ljudi kot lani, razpršene tabele pa več; je pa vprašljivo, ali smemo iz takšnih sprememb vleči kakšne zaključke, saj je število anketiranih tekmovalcev precej majhno.

	Prva skupina	Druga skupina	Tretja skupina
drevo	42 %	64 %	75 %
hash tabela	17 %	64 %	57 %
seznam (linked list)	40 %	57 %	63 %
sklad (stack)	42 %	50 %	71 %
vrsta (queue)	27 %	57 %	58 %
Evklidov algoritem	74 %	79 %	70 %
Eratostenovo rešeto	58 %	62 %	65 %
vektorski produkt	43 %	54 %	48 %
rekurzija	19 %	54 %	57 %
dinamično prog.	19 %	31 %	35 %
iskanje v širino	6 %	17 %	44 %
O-zapis	34 %	46 %	48 %
urejanje	63 %	85 %	85 %
bubble sort	45 %	64 %	75 %
insertion sort	31 %	36 %	46 %
selection sort	38 %	36 %	46 %
quicksort	22 %	57 %	42 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

Uporaba programskih jezikov

Velika večina tekmovalcev tudi letos uporablja C in C++ (še posebej slednjega), novost v primerjavi z lanskim letom pa je višji delež uporabnikov jave, še posebej v tretji skupini. Pascal podobno kot lani uporablja zelo malo ljudi. Zanimiv pojav, ki smo ga letos opazili prvič, je nezanemarljivo število tekmovalcev (in tekmovalk), ki tekmujejo v 1. skupini in pri vseh nalogah opisujejo svoje rešitve le v psevdokodi, čeprav so bile naloge tipa „napiši program“ in torej načeloma zahtevajo odgovor v kakšnem konkretnem programskem jeziku. Mogoče je torej letošnja prirast v številu tekmovalcev sestavljena predvsem iz začetnikov, ki programiranja še niso vešči. Vsekakor bi bilo dobro, če bi bilo v prvi skupini več nalog tipa „opiši postopek“ namesto „napiši program“, da bi bila bolj prijazna do takih začetnikov.

Podobno kot lani je v anketi precej tekmovalcev napisalo, da dobro poznajo tudi PHP, vendar so ga na tekmovanju uporabljali le trije.

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže tabela na str. 129. Glede štetja C in C++ v tej tabeli je treba pripomniti, da je razlika med njima tako ali tako zelo majhna: tisti, ki delajo v C++, uporabljajo večinoma le malo stvari, ki jih C++ ima, C pa ne. To so ponavadi <iostream> in vhodno/izhodna tokova `cin` in `cout` namesto C-jevih funkcij `scanf`, `printf` in podobnih. Precej pogosto uporabljajo tudi razred `string`.

V besedilu nalog za 1. in 2. skupino objavljamo deklaracije tipov, spremenljivk, podprogramov ipd. v pascalu, C/C++, pythonu in letos po novem tudi v javi. Tekmovalce smo v anketi vprašali, če te deklaracije razumejo ali pa bi morale biti še v kakšnem drugem jeziku; veliki večini sedanje deklaracije zadostujejo (40/51 v prvi skupini in 11/14 v drugi). Žal so letošnji rezultati pri tem vprašanju malo zavajajoči, ker smo v anketnem vprašanju pozabili omeniti, da so deklaracije letos že tudi v javi, torej so nekateri tekmovalci, ki so obkrožili odgovor „ne“, s tem v resnici hoteli

	Leto in skupina															
	2009			2008			2007			2006			2004			2003
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	3
pascal	4	2	1	$1\frac{1}{2}$	2	2	$8\frac{1}{2}$	2	1	6	5	5	23	20	13	17
C	$9\frac{1}{2}$	$3\frac{1}{2}$	$\frac{1}{2}$	$4\frac{1}{2}$	11	$2\frac{1}{2}$	$5\frac{1}{2}$	11	$6\frac{1}{2}$	4	16	$1\frac{1}{2}$	13	$7\frac{1}{2}$	1	4
C++	$26\frac{1}{2}$	2	$12\frac{1}{2}$	$17\frac{1}{2}$	11	$9\frac{1}{2}$	7	14	$15\frac{1}{2}$	13	5	$10\frac{1}{2}$	5		6	5
java	8	8	11	$9\frac{1}{2}$	3			$2\frac{1}{2}$				3		$\frac{1}{2}$		–
PHP	2	1	–		2	–	1		–	1		–			–	–
basic			–			–		1	–		1	–			–	–
C#						3		$\frac{1}{2}$	–			–			–	–
python	4	$\frac{1}{2}$	–	6	1	–			–			–			–	–
psevdokoda	8		–	–	–	–	–	–	–	–	–	–	–	–	–	–
nič	1				1		3			1	2		3	2		

Število tekmovalcev, ki so uporabljali posamezni programski jezik. Nekateri uporabljajo po dva različna jezika (pri različnih nalogah) in se štejejo polovično k vsakemu jeziku. „Nič“ pomeni, da tekmovalec ni napisal nič izvorne kode. Znak „–“ označuje jezike, ki se jih tisto leto v tretji skupini ni dalo uporabljati. Psevdokoda šteje tekmovalce, ki so pisali le psevdokodo, tudi pri nalogah tipa „napiši (pod)program“; pred letom 2009 takih nismo šteli posebej in če je kdo uporabljal le psevdokodo, je štet pod „nič“.

---

povedati, da deklaracij v pascalu, C/C++ in pythonu ne bi razumeli (javanske pa razumejo).

V rešitvah nalog smo do vključno lanskega leta objavljali izvorno kodo v pascalu in C-ju; tekmovalce vseh treh skupin smo v anketi vprašali, če kakšnega od teh dveh jezikov razumejo dovolj, da si lahko kaj pomagajo s to izvorno kodo, in če bi radi videli izvorno kodo rešitev še v kakšnem drugem jeziku. Velika večina je bila s tema dvema jezikoma zadovoljna (32/48 v prvi skupini, 11/13 v drugi in 18/22 v tretji). Pet ljudi je predlagalo, da bi bile rešitve tudi v C++, 10 v javi, dva v pythonu in po eden v PHP in C#. Primerov, ko ljudje pravijo, da sedanjih rešitev (v C-ju) ne razumejo in da bi jih radi videli v C++, je letos kar pet (lani štirje). Ta pojav nas še vedno preseneča, ker pa očitno sam od sebe ne bo izginil, bi bilo mogoče dobro naslednjič vprašati, kaj točno je tisto, zaradi česar ti ljudje C-jaških rešitev ne razumejo, C++ovske pa bi (uporaba malloc/free namesto **new**/**delete**? uporaba printf in scanf namesto cin in cout?).

Ker je iz anket razvidno, da pozna pascal tako malo ljudi, smo se odločili, da letos rešitev v pascalu ne bomo več objavljali. V letošnjem biltenu so zato rešitve le v C-ju, prihodnje leto pa bomo razmislili o tem, ali bi mu dodali še kakšen jezik. Možen kandidat bi bil python, ki je dovolj drugačen od C-ja, da bi bilo imeti rešitve v obeh jezikih lahko celo zares poučno; ker pa ga zna bolj malo tekmovalcev, bi bilo verjetno bolj pametno uvesti rešitve v javi (s tem bi ustregli še največ tekmovalcem).

Letnik

Po pričakovanjih so tekmovalci zahtevnejših skupin v povprečju v višjih letnikih kot tisti iz lažjih skupin. Razmerja so podobna kot lani. Največ tekmovalcev hodi v tretji ali četrti letnik; letos je njihov delež še malo višji kot lani, v drugi in tretji skupini so skoraj sami taki.

Skupina	Št. tekmovalcev po letnikih				Povprečni letnik
	1	2	3	4	
prva	15	13	21	13	2,5
druga	2		6	9	3,3
tretja			9	16	3,6

Druga vprašanja

Podobno kot lani je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!).

Pri vprašanju, kje so se naučili programirati, podobno kot lani prevladujeta odgovora „sam“ in „v šoli“. Samoukov je tokrat malo več kot tistih, ki so se programirati naučili v šoli (podobno kot predlani; lani pa je bilo obratno).

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni. Precej je tudi tekmovalcev, ki želijo manj nalog in/ali več časa.

Skupina	Kje si izvedel za tekmovanje				Kje si se naučil programirati					Čas reše-vanja			Število nalog			Potekmovalne dejavnosti							
	od mentorja	na spletni strani	od prijatelja/sošolca	drugače	sam	pri pouku	na krožkih	na tečajih	poletna šola	hočem več časa	hočem manj časa	je že v redu	hočem več nalog	hočem manj nalog	je že v redu	izlet v tuji laboratorij	poletna šola	praksa na IJS	predstavitve tehnologij	predavanja o algoritmih	reševanje nalog	iskanje štipendije	iskanje podjetij
I	51	0	3	1	25	27	12	4	1	12	4	28	2	15	28	13	11	11	15	13	10	21	20
II	11	1	1	0	12	4	4	1	1	4	0	9	0	4	9	3	3	5	5	3	3	4	6
III	21	0	1	1	15	11	1	1	0	3	3	8	1	8	5	4	4	3	2	2	1	3	1

Iz odgovorov na vprašanje, kakšne potekmovalne dejavnosti bi jih zanimale, je težko zaključiti kaj posebej konkretnega.

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb. Letos so imeli tekmovalci v prvi in drugi skupini prvič možnost pisati svoje odgovore na računalniku namesto na papir (kar so si v prejšnjih letih v anketah že večkrat želeli). Ta možnost je bila zelo dobro sprejeta in so jo izkoristili vsi tekmovalci razen dveh; mnogi so jo tudi pohvalili v anketah, so pa opozorili na nekaj težav pri izvedbi (hrup v računalniški učilnici in čakanje na začetek tekmovanja).

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) Izviren pogled na standardni vhod:

```
file.open("naloga1.txt"); // odpremo datoteko oz. standardni vhod, kjer so števila
```

(Naloga je sicer nalašč zahtevala, naj program bere s standardnega vhoda, ne pa iz datoteke, ker smo hoteli spodbujati rešitve, ki se skozi podatke zapeljejo le enkrat, ne pa dvakrat.)

(1.1) Častna omemba v kategoriji najbolj nepotrebnih pogojev:

```
while (! file.eof()) {
   if (! file.eof()) {
```

Pri tem je file deklariral celo kot ofstream (in kasneje vanj tudi res nekaj zapisoval), tako da bi bil eof() vedno **false**.

(1.1) Letos je v prvi skupini precej tekmovalcev in tekmovalk pisalo rešitve v psevdokodi, kar je načeloma lepo (čeprav so letos naloge v prvi skupini vse zahtevale programe ali podprograme, ne le opisov postopkov), težava pa je, da so takšni opisi v psevdokodi pogosto težko berljivi.

če (if) je prebrano število v vrstici večje od 0 in manjše od najmanjšega
potem (then)

pogleda, kolikokrat je število enako številu, ki je manjše od najmanjšega

(1.2) Ena od slabosti tega, da tekmovalci pišejo na računalnikih, je ta, da imamo več mazohistične kode kot prejšnja leta, ker jo pač lažje generirajo s copy & paste. Tule je primer:

```
char ret(int zap) {
   switch(zap) {
   case 0: return '0';
   case 1: return '1';
   ⋮
   case 29: return 'Z';
   default: exit(1); // napaka
   }
}
```

Žalostno dejstvo je, da v naslednjem podprogramu iste rešitve ta tekmovalec kaže, da zna uporabljati **char*** kot tabelo znakov, kar bi lahko naredil tudi v podprogramu ret in ga tako iz dobrih 30 vrstic skrajšal na eno ali dve.

Še hujši primer je rešitev v naravnem jeziku, dolga kar 7000 znakov, ki jo je oddal nek drug tekmovalec pri nalogi 1.4. Večina tega odpade na osem kopij enega in istega besedila (ki v bistvu predstavlja zanko z osmimi iteracijami).

(1.2) Nagrada za najbolj nepotrebno uporabo stavka **goto** (verjetno celo edino letošnjo uporabo tega stavka sploh):

```
tukaj:
crka = rand() % 25 + 85;
if (crka == 'O' || crka == 'Q' || crka == 'S' || crka == 'U' ||
   crka == 'L' || crka == 'I') goto tukaj;
```

(1.2) Letošnjo nagrado za najglobljo indentacijo dobi:

```
                              e := e + 1;
                           end;
                        d := d + 1;
                     end;
                  c := c + 1;
               end;
            b := b + 1;
         end;
      a := a + 1;
   end;
end;
end.
```

(1.3) Prva nagrada v kategoriji najbolj nepotrebnih pogojev:

```
for (int j = 0; j < stevilo[stevec]; j++) {
   if (stevilo[i] == stevilo[j + 1] || stevilo[i] == stevilo[j − 1]) {
      if (stevilo[i] == stevilo[j + 1] || stevilo[i] == stevilo[j − 1]) {
         if (stevilo[i] == stevilo[j + 1] || stevilo[i] == stevilo[j − 1]) {
            if (stevilo[i] == stevilo[j + 1] || stevilo[i] == stevilo[j − 1]) {
               if (stevilo[i] == stevilo[j + 1] || stevilo[i] == stevilo[j − 1]) {
```

(1.3) Pogumen način, kako prebrati zaporedje števil s standardnega vhoda:

```
int strani[2000]; // predpostavljamo, da knjiga nima več kot 1999 strani
cin >> strani[2000];
```

(1.4) En tekmovalec dosledno piše „tarkunosmrkci" namesto „tajkunosmrkci".

(1.5) Zanimiv način poimenovanja spremenljivk:

```
// Ja... Lahko bi človeka najeli, da bi štel vlake pa to... :D
⋮
void Inicializacija() {
   bool 1 = 0;
   bool 2 = 0;
}
void SpremembaSenzorja(int senzor, bool prekinjen){
   if (senzor == 1) {
      if (1 = 0)
         1 = 1;
      else
         1 = 0;
   }
```

In tako naprej.

(2.1) Še nekdo s predrznim pogledom na to, kaj je standardni vhod in izhod:

```
String vhodna_datoteka = 'Naloga1_vhodna.txt'; // standardni vhod
String izhodna_datoteka = 'Naloga1_izhodna.txt'; // standardni izhod
:
:
BufferedWriter pisi = new BufferedWriter(new FileWriter(izhodna_datoteka));
BufferedReader beri = new BufferedReader(new
                                 FileReader(vhodna_datoteka)); // deklaracija vhoda in izhoda
```

(2.2) Zanimiv nov primerjalni operator (rešitev je sicer v C-ju):

```
if (casn %= cas) { // če je čas nastanka slike podoben času iz GPS-a
```

(2.3) Kako preveriti, ali nek znak ni samoglasnik:

```
if (trnBeseda[k] != ('a' || 'e' || 'i' || 'o' || 'u')) {
```

(2.4) Primer samokritičnega komentarja:

```
/* močno dvomim, da bi tole dejansko delovalo, ampak poskusiti ni greh */
```

Pa v resnici rešitev ni bila tako slaba, le neupravičeno je predpostavila, da bo znak razbit na natanko štiri peterice bitov.

(2.4) Nagrada za najboljšo (mogoče namerno?) pravopisno napako letos:

```
int cunt = 0;
:
:
if (cunt != 4) { // če še ni štirih novih delov znaka, si prejetega zapišemo
                 // v nov del seznama stevila[4]
  stevila[cunt] = x;
  cunt++;
}
```

(2.5) Eden od tekmovalcev je nalogo očitno narobe razumel, kot da bi morali program nekako interpretirati, ne pa le predelati:

> Programu ukažemo, da se premakne za en znak naprej (torej gre mimo zavitega zaklepaja — s tem pa gre tudi ven iz trenutne iteracije najbolj notranje zanke, isto kot bi naredili s stavkom **break**)

(3.1) Eden od tekmovalcev je oddal program, ki ne naredi drugega, kot da prebere vhodno datoteko. Isti program je oddal še nekajkrat slabe pol ure kasneje.

(3.2) Rešitev, ki se izogiba character literalom:

```
char c;
:
:
c = fgetc(datin);
if ((int) c == 87) {
```

(3.2) Rešitev, ki naredi vtis predvsem po razmerju med številom napak in dolžino programa:

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

int main()
{
   int w = 0;
   int h = 0;

   ifstream ifs("otoki.txt");
   cin >> w;
   cin >> h;

   int kopno = w * h;

   ofstream ofs("otoki.txt");
   ofs << kopno;

   system("pause");
   return 0;
}
```

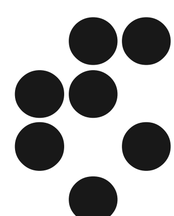
Malo kasneje je oddal tudi različico z **int** kopno = 2.

(3.3) Eden od tekmovalcev je oddal program, ki vedno izpiše 0; nato še program, ki vedno izpiše 1; pa takega, ki vedno izpiše 2; in tako naprej do 20. Najboljši od teh programov je celo dosegel 20 točk (ker je bil pri dveh od naših desetih testnih primerov pravilni odgovor „4“), vendar mu je veliko število oddaj pri tej nalogi zbilo število točk nazaj na 0.

Pri isti nalogi je nek drug tekmovalec oddal program, ki izpiše naključno število od 1 do 9 in z njim pri enem od desetih testnih primerov celo pravilno uganil rešitev. Spet tretji tekmovalec je poskusil nekaj podobnega pri nalogi 3.2 in izpisoval naključno število med 1 in $k/2$, vendar ni nobenkrat zadel pravega odgovora.

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 so podprli naslednji odseki 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.

E2 — Odsek za sisteme in vodenje

Poslanstvo odseka, ki smo si ga postavili kot vodilno nit ob začetku delovanja, smo opredelili kot „premoščanje prepada med teorijo in prakso“.

Osrednji poudarek pri načinu našega dela je na zahtevi, da je treba izhajati iz konkretnih (industrijskih) problemov ter raziskave prilagajati temu, ne pa nasprotno. To dejstvo seveda potegne za seboj potrebo po širokem območju zelo različnih znanj, tako po tipu dela (raziskave, razvoj, inženirstvo itd.) kot tudi stroki (avtomatika, elektronika, računalništvo itd.).

Poslanstvo je izšlo iz potreb domačega okolja, saj so izkušnje pokazale, da spoznanj iz raziskav praktično ne moremo uporabiti pri reševanju konkretnih aplikativnih problemov in je s tem tudi opredelilo način našega dela.

Dejavnosti odseka obsegajo raziskave, razvoj in aplikacije na širšem področju računalniško podprtega vodenja in regulacije (predvsem) tehničnih sistemov, skupaj z ustreznimi storitvami, inženirstvom in izobraževanjem

S svojim znanjem vam lahko pomagamo na področju elektronike, merilne in regulacijske tehnike, računalniške avtomatizacije in informatizacije procesov.

Samo v času od svoje formalne ustanovitve (1986) je odsek delal večje ali manjše projekte za okoli 100 slovenskih in tujih podjetij, večinoma iz sektorja industrijske proizvodnje.

E8 — Odsek za tehnologije znanja

Poslanstvo Odseka za tehnologije znanja IJS je razvoj naprednih informacijskih tehnologij za zajemanje, shranjevanje, upravljanje in odkrivanje znanja s poudarkom na rudarjenju prodatkov oz. strojnem učenju, podpori odločanja in razvoju jezikovnih tehnologij, katerih cilj je prispevati vrhunske znanstvene rezultate v svetovno zakladnico znanja ter pospeševati aplikacije teh tehnologij za razvoj e-znanosti in družbe znanja.

Dolgoročni cilji odseka so razvoj metod inteligentne analize podatkov, upravljanja znanja, podpore odločanja in računalniškega jezikoslovja ter njihova uporaba za reševanje praktičnih problemov na področju ekologije, medicine, zdravstvenega varstva, ekonomije in tržništva. V raziskave vključujemo tudi novejša področja informacijskih tehnologij: semantični splet in upravljanje mrežnih organizacij.

E9 — Odsek za inteligentne sisteme

Osnovni cilji Odseka za inteligentne sisteme so raziskave računalniških osnov inteligence in razvoj naprednih aplikacij s področja inteligentnih informacijskih storitev, analize podatkov, inteligentnega preiskovanja spleta, podpore odločanja, inteligentnih agentov, medicine, ekologije, jezikovnih tehnologij, inteligentne proizvodnje in ekonomije. Z več kot 20-letno tradicijo pri raziskavah in razvoju na širšem področju umetne inteligence, inteligentnih sistemov, medicinske informatike, procesiranja naravnega jezika in kognitivnih znanosti se je Odsek za inteligentne sisteme uveljavil v evropskem in svetovnem merilu. Sodelavci odseka so razvili več delujočih sistemov, ki so pomembni tako v slovenskem kot mednarodnem merilu.

Raziskovalna področja Odseka za inteligentne sisteme:

- induktivno logično programiranje
- evolucijsko računanje

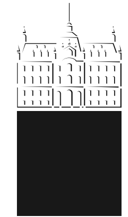
- večstrategijsko učenje in principi mnogoterega znanja
- rudarjenje spletnih podatkov — sinteza znanja za modeliranje in vodenje sistemov
- sistemi za podporo odločanja
- principi inteligence in kognitivnih znanosti
- inteligentni agenti in večagentni sistemi
- umetna inteligenca v medicini
- sinteza govora
- ontologije in semantični splet
- analiza igranja iger.

*

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.

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.

MINISTRSTVO ZA ŠOLSTVO IN ŠPORT

Ministrstvo za šolstvo in šport

Ministrstvo za šolstvo in šport opravlja naloge na področjih vzgoje, osnovnega šolstva, osnovnega glasbenega izobraževanja, srednjega in višjega šolstva, izobraževanja odraslih ter športa. Nastalo je konec leta 2004 po razdružitvi dotedanjega Ministrstva za šolstvo, znanost in šport na Ministrstvo za šolstvo in šport ter Ministrstvo za visoko šolstvo, znanost in tehnologijo.

SREBRNA POKROVITELJA

Quintelligence
Obstoječi informacijski sistemi podpirajo predvsem procesni in organizacijski nivo pretoka podatkov in informacij. Biti lastnik informacij in podatkov pa ne pomeni imeti in obvladati znanja in s tem zagotavljati konkurencne 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.

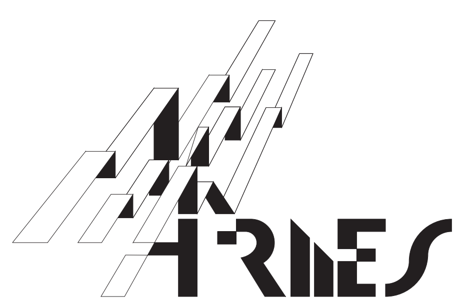
Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja.

Sun d. o. o.

BRONASTI POKROVITELJI

cosylab

CONTROL SYSTEM LABORATORY



IXTLAN
TEAM

K
KOGAST

MARAND
Napredna računalniška hiša
Sun
microsystems

Zemanta
TM

Rtk

09