4. tekmovanje IJS v znanju računalništva za srednješolce

28. marca 2009

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.");
   }
}
```

4. tekmovanje IJS v znanju računalništva za srednješolce

28. marca 2009

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.

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

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 številkam 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 številke 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:

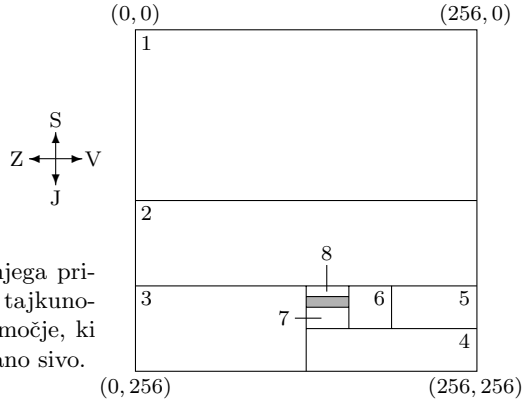
```
procedure Nerazdeljeno(Delitve: string);          { v pascalu }
void Nerazdeljeno(char* delitve);                 /* v C/C++ */
public static void Nerazdeljeno(String delitve);  /* v javi */
def Nerazdeljeno(delitve): ...                    # v pythonu
```

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 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.

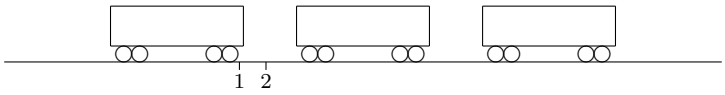


Slika na desni kaže ozemlje po delitvah iz zgornjega primera. Območja, ki so si jih prisvojili posamezni tajkunosmrkci, so označena s številkami od 1 do 8. Območje, ki je na koncu ostalo še nerazdeljeno, pa je pobarvano sivo.

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(): ...
```

4. tekmovanje IJS v znanju računalništva za srednješolce

28. marca 2009

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.

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

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 želeni 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, številka, 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 *n̈* 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:

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 $0, \ldots, 31$.

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
```

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**).

(*Besedilo naloge se nadaljuje na naslednji strani.*)

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.

4. tekmovanje IJS v znanju računalništva za srednješolce

28. marca 2009

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;
}
```

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

- V javi:

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

public class Poskus
{
   public static void main(String[] args) throws IOException
   {
      Scanner fi = new Scanner(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();
   }
}
```

4. tekmovanje IJS v znanju računalništva za srednješolce

28. marca 2009

NALOGE ZA TRETJO SKUPINO

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

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 $1 \leq n \leq 15$, $0 \leq m \leq 1000$ in $0 \leq k \leq 1000$.

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 if, 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
$x_1\ \ldots\ x_n\ n$	copy	$x_1\ \ldots\ x_n\ x_1\ \ldots\ x_n$	podvoji vrhnjih n elementov
$x_n\ \ldots\ x_0\ n$	index	$x_n\ \ldots\ x_0\ x_n$	podvoji $(n+1)$-ti element
$x_{n-1}\ \ldots\ x_0\ n\ i$	roll	$x_{i-1}\ \ldots\ x_0\ x_{n-1}\ \ldots\ x_i$	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
$\circledast\ x_1\ \ldots\ x_n$	counttomark	$\circledast\ x_1\ \ldots\ x_n\ n$	prešteje elemente nad značko
$\circledast\ x_1\ \ldots\ x_n$	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.

(*Besedilo naloge se nadaljuje na naslednji strani.*)

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 $1 \le n \le 100$). Pri petih od desetih testnih primerov bo veljalo $n \le 10$. 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 ... an-1 an n
  push(0)               % a1 a2 ... an-1 an n 0

zacetek:
                        % a1 a2 ... ak-1 ak k (ak+1 + ... + an)
  exch                  % a1 a2 ... ak-1 ak (ak+1 + ... + an) k
  dup push(0)           % a1 a2 ... ak-1 ak (ak+1 + ... + an) k k 0
  % Če je k = 0, končajmo.
  le
  jnz(konec)
                        % a1 a2 ... ak-1 ak (ak+1 + ... + an) k
  push(1) sub           % a1 a2 ... ak-1 ak (ak+1 + ... + an) (k - 1)
  push(3) push(1)       % a1 a2 ... ak-1 ak (ak+1 + ... + an) (k - 1) 3 1
  roll                  % a1 a2 ... ak-1 (k - 1) ak (ak+1 + ... + an)
  add                   % a1 a2 ... ak-1 (k - 1) (ak + ... + an)
  push(1)
  jnz(zacetek)

konec:
                        % (a1 + ... + an) 0
  pop                   % (a1 + ... + an)
```

4. tekmovanje IJS v znanju računalništva za srednješolce

28. marca 2009

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;
}
```

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++)
  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;
}
```

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;
}
```

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 $\langle k, \mathit{TrenutniCas} \rangle$ na konec seznama S
  else if *stNaprave* $= 2$ **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; }
}
```

5. break considered harmful

Stavek **break** pravzaprav naredi dvoje: prekine trenutno iteracijo zanke in nato tudi poskrbi, da se ne bo izvedla nobena iteracije te zanke več, pač pa se začnejo izvajati 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 bomo uporabili namesto pogoja za nadaljevanje 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;
      } } }
  }
```

REŠITVE NALOG ZA TRETJO SKUPINO

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. Ko na ta način pregledamo celoten seznam, gremo še enkrat od začetka proti koncu in izpišemo nize pri tistih ukazih `WRITE`, ki smo jih pred tem označili za izpis.

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

#define MaxN 100000
#define MaxD 100

int main()
{
  FILE *f; int i, n, dolzina, *undo;
  bool *izpisi;
  char **besede, *p;
  char vrstica[MaxD + 8];

  /* Preberimo število ukazov in pripravimo tabele. */
  f = fopen("undo.in", "rt");
  fscanf(f, "%d\n", &n);
  besede = (char **) malloc(n * sizeof(char *));
  undo = (int *) malloc(n * sizeof(int));
  izpisi = (bool *) malloc(n * sizeof(bool));
  /* 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++) {
    fgets(vrstica, MaxD + 8, f);
    dolzina = strlen(vrstica) − 1; vrstica[dolzina] = 0;
    izpisi[i] = false;
    if (strncmp(vrstica, "WRITE ", 6) == 0) {
      undo[i] = −1;
      dolzina −= 6; besede[i] = (char *) malloc(dolzina + 1);
      strcpy(besede[i], vrstica + 6); }
    else sscanf(vrstica, "UNDO %d", &undo[i]); }
  fclose(f);

  /* Poglejmo, katere besede bo treba izpisati. */
  for (i = n − 1; i >= 0; i−−)
    if (undo[i] >= 0) i −= undo[i];
    else izpisi[i] = true;

  /* Izpišimo rezultat. */
  f = fopen("undo.out", "wt");
  for (i = 0; i < n; i++)
```

```
    if (izpisi[i]) fprintf(f, "%s", besede[i]);
  fprintf(f, "\n"); fclose(f);

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

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. Red 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 vrste1,
   dodal v vrsto 2 in jih pobarval v barvo vBarvo2. Za vrsto 2
   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;
  /*printf("Pobarvaj %d->%d ; %d->%d"n", izBarve1, vBarvo1, izBarve2, vBarvo2);*/
  while (glava < r1)
  {
    u = vrsta1[glava++];
    /*if (0) printf(- DeQ %d (%d, %d) barve %d"n", u, u % w, u / w, z[u]);*/
    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, najRed;
  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 reda 0 z barvo 3,
     nato jezera na teh otokih z barvo 4, nato otoke reda 1 z barvo 5,
     nato jezera na teh otokih z barvo 6, nato otoke reda 2 z barvo 7
     in tako naprej. */
  najRed = -1; barva = 2;
  while (n1 > 0) {
    if ((barva % 2) == 1) najRed = (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++; }

  f = fopen("otoki.out", "wt");
  fprintf(f, "%d\n", najRed);
  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;
}
```

Gornji postopek bi se dalo še malo izboljšati: ko poberemo točko u iz vrste, gre gornji postopek po vseh možnih novih strupih i in pri vsakem pregleda tabelo A (da vidi, če bi pacient preživel) in mogoče še B (da vidi, v katero stanje pridemo po izničenju za to primernih kombinacij). To nam da v najslabšem primeru časovno zahtevnost $O(n \cdot (m+k))$ pri vsakem u. Z nekaj razmisleka in eno pomožno tabelo velikosti 2^n bytov lahko postopek predelamo tako, da izvede zanki po tabelah A in B le enkrat (pri tem u), ne pa n-krat (za vsak i po enkrat); s tem se časovna zahtevnost močno zmanjša, na $O(n + m + k)$ pri vsaki točki u. Vendar pa je za naše testne primere že gornja rešitev čisto dovolj hitra.

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.

```
                              % a1 a2 ... an n
zanka:                        % an an−1 ... ak+1 a1 a2 ... ak−1 ak k
  exch                        % an an−1 ... ak+1 a1 a2 ... ak−1 k ak
  push(1) index               % an an−1 ... ak+1 a1 a2 ... ak−1 k ak k
  push(1) add push(1)         % an an−1 ... ak+1 a1 a2 ... ak−1 k ak (k + 1) 1
  roll                        % an an−1 ... ak+1 ak a1 a2 ... ak−1 k
  push(1) sub                 % an an−1 ... ak+1 ak a1 a2 ... ak−1 (k − 1)
  % Oz. če zmanjšamo k za 1:  % an an−1 ... ak+1 a1 a2 ... ak k
  % Če je k > 1, moramo še nadaljevati, sicer lahko končamo.
  dup push(1) gt jnz(zanka)

                              % an an−1 ... a2 a1 1
  pop                         % an an−1 ... a2 a1
```

(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.)

Viri nalog: otoki, strupi — Nino Bašić; 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; smrkci — Anže Žagar; kolikokrat najmanjši, UTF-5, **break** considered harmful, EPS — Janez Brank. Hvala Tomažu Hočevarju za implementacijo rešitev nalog za 3. skupino.

Vprašanja, pripombe, komentarji, popravki ipd. v zvezi z nalogami in rešitvami so dobrodošli: ⟨*janez@brank.org*⟩.