5. tekmovanje ACM v znanju računalništva

Institut Jožef Stefan, Ljubljana, 27. marca 2010

Bilten

Bilten 5. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2011

Uredil Janez Brank

Avtorji nalog: Andrej Bauer, Nino Bašić, Matej Črepinšek, Primož Gabrijelčič, Boris Gašperin, Marko Grobelnik, Tomaž Hočevar, Peter Keše, Jurij Kodre, Mitja Lasič, Borut Lesjak, Matija Lokar, Mojca Miklavec, Polona Novak, Mark Martinec, Matej Šprogar, Mitja Trampuš, Matjaž Urlep, Anže Žagar, Klemen Žagar, Janez Brank.

Tisk: Potens d. o. o., Podolnica 40a, Horjul
Naklada: 350 izvodov

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 ACM in IJS v znanju računalništva (5 ; 2010 ; Ljubljana)
Bilten [Elektronski vir] / 5. tekmovanje ACM in IJS v znanju računalništva, Ljubljana, 27. marca 2010 ; avtorji nalog Andrej Bauer ... [et al.] ; uredil Janez Brank. — Ljubljana : Institut Jožef Stefan, 2011

Način dostopa (URL): `http://rtk.ijs.si/2010/rtk2010-bilten.pdf`

ISBN 978-961-264-033-0
1. Bauer, Andrej 2. Brank, Janez, 1979–
255355904

KAZALO

STRUKTURA TEKMOVANJA

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

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

V lažjih dveh skupinah traja tekmovanje tri ure; tekmovalci lahko svoje rešitve napišejo na papir ali pa jih natipkajo na računalniku, nato pa njihove odgovore oceni temovalna komisija. 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. Podobno kot lani smo tudi letos ponudili možnost, da tekmovalci v prvi in drugi skupini svoje odgovore natipkajo na računalniku; tudi tokrat so se zanjo skoraj vsi tekmovalci. 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 19.) 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. 18–20 za 3. skupino).

Omenimo še, da so rešitve, objavljene v tem biltenu, večinoma obsežnejše od tega, kar na tekmovanju pričakujemo od tekmovalcev, saj je namen tukajšnjih rešitev pogosto tudi pokazati več poti do rešitve naloge in bralcu omogočiti, da bi se lahko iz razlag ob rešitvah še česa novega naučil.

Poleg tekmovanja v znanju računalništva smo organizirali tudi tekmovanje programov, ki je podrobneje predstavljeno na straneh 129–130.

Podobno kot lani smo izvedli tudi šolsko tekmovanje, ki je potekalo 29. januarja 2010. To je imelo eno samo težavnostno skupino, naloge (ki jih je bilo pet) pa so pokrivale precej širok razpon težavnosti. Tekmovalci so pisali odgovore na papir in dobili enak list z nasveti in navodili kot na državnem tekmovanju v 1. in 2. skupini (str. 7–9). Odgovore tekmovalcev na posamezni šoli so ocenjevali mentorji z iste šole, za pomoč pa smo jim pripravili nekaj strani z nasveti in kriteriji za ocenjevanje (str. 119–122). 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. Tako kot lani se je šolsko tekmovanje izkazalo za zelo 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 se rast števila tekmovalcev na državnem tekmovanju nadaljuje tudi letos (letos je bilo skupaj 134 tekmovalcev, lani 104, predlani 87, pred tem 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;
```

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

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

Pri ocenjevanju so vse naloge vredne enako število točk. Svoje odgovore dobro utemelji. Prizadevaj si predvsem, da bi bile tvoje rešitve pravilne, ob tem pa je zaželeno, da so tudi čim bolj učinkovite (take dobijo več točk kot manj učinkovite). 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 ali pa oddaš del odgovorov na papirju in del v datoteki. Vse te možnosti so enakovredne. Odgovore, oddane prek računalnika, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev samodejno shranjuje in na zaslonu ti bo pisalo, kdaj se bo shranjevanje naslednjič zgodilo. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko bi rad začasno prekinil pisanje rešitve naloge ter se lotil druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu.

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. Vodilni elementi

Za posamezni element v tabeli števil (*array*) rečemo, da je *vodilni*, če velja, da so vsi njegovi nasledniki (elementi z večjim indeksom) manjši od njega.

Primer: v tabeli $[10, 8, 11, 4, 8, 2, 8, 5, 7, 3, 1]$ so vodilni elementi 11, 8, 7, 3 in 1.

Napiši program, ki s standardnega vhoda prebere število elementov tabele, nato vse elemente (vsak podatek je zapisan v svoji vrstici), ter izpiše, koliko ima ta tabela vodilnih elementov. Predpostavi, da so podatki cela števila, da tabela nikoli ne bo imela več kot sto elementov ter da v podatkih ni nobenih napak (ni potrebno preverjati pravilnosti vhoda).

Primer (za enako zaporedje kot zgoraj): če so na standardnem vhodu podatki

```
11
10
8
11
4
8
2
8
5
7
3
1
```

mora program izpisati vrednost 5.

2. Abecedni podnizi

V nekaterih besedah se zgodi, da si več zaporednih črk sledi v abecednem vrstnem redu in to celo tako, da vmes nobena ne manjka. Takemu zaporedju črk pravimo *abecedni podniz*. Nekaj primerov: *ab* v besedi *nabava, jkl* v *primanjkljaj, hij* v *monarhija, mnop* v *limnoplankton, abc* v *vrabci* (podniz *abci* pa ni abecedni podniz, ker je sicer *i* v abecedi za *c*, vendar so vmes v abecedi še druge črke).

Napiši program, ki prebere zaporedje besed in izpiše dolžino najdaljšega abecednega podniza, ki se pojavlja v kakšni od teh besed. Program naj bere besede s standardnega vhoda vse do konca (EOF); vsaka beseda je v svoji vrstici, zapisane pa so samo z malimi črkami. Posamezna beseda je dolga največ sto znakov. Da bo lažje, uporabljamo pri tej nalogi angleško abecedo:

a b c d e f g h i j k l m n o p q r s t u v w x y z

3. Skrivanje tipk

Navigacijska naprava ima na dotik občutljiv zaslon, na katerem se izriše tipkovnica, prek katere lahko vnesemo ime kraja, kamor smo namenjeni. Izbira med kraji je omejena na sto krajev.

Da olajšamo vnos in zmanjšamo možnost napake, so na narisani (virtualni) tipkovnici ob vnosu vsake naslednje črke imena prikazane le tiste tipke/črke, ki še lahko pridejo v poštev na tem mestu glede na omejen nabor krajev in glede na dosedaj vnesene črke.

Napiši podprogram NaslednjeCrke, ki bo kot parameter dobil niz, ki ga je uporabnik doslej že vnesel, tvoj podprogram pa naj izpiše vse možne črke, ki pridejo v poštev kot naslednja črka v imenu kraja. Predpostaviš lahko, da je vseh 100 možnih krajevnih imen že shranjeno v neki tabeli nizov (njeno ime si lahko izbereš sam). Ta tabela se med klici tvojega podprograma ne bo spreminjala. Uporabljaš lahko tudi globalne spremenljivke in jim po svoje določiš začetne vrednosti. Da se izognemo zapletom, predpostavimo, da nastopajo v imenih le male črke angleške abecede in nobeni drugi znaki, namesto znakov s strešicami pa stojijo črke `c`, `s`, `z`.

Naslednji primer ilustrira vnos črk `c`, `e`, `l`, `j`, `e` in možen vsakokratni izpis naslednjih črk — pri tem smo si pomagali s seznamom imen slovenskih krajev:

NaslednjeCrke("")	izpiše	`abcdefghijklmnoprstuvz`
NaslednjeCrke("c")	izpiše	`aeimoruv`
NaslednjeCrke("ce")	izpiše	`bcdghklmnprstz`
NaslednjeCrke("cel")	izpiše	`eijo`
NaslednjeCrke("celj")	izpiše	`e`
NaslednjeCrke("celje")	izpiše prazen niz	

Tvoj podprogram naj bo takšne oblike:

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

4. Cikel

Prihajava počasi na prvi peron,
tam priključi nama se še eden vagon,
pa še drugi in pa tretji, kmalu nas je sto-o-o,
kaj če mašino razneslo bo?
— Bepop, *Lokomotiva*

Na zabavi s super muziko se znajde n ljudi. Prevzame jih plesalska strast, zato vsak plesalec zagrabi natanko enega soudeleženca zabave za boke; tako vsi skupaj naredijo enega ali več „vlakcev“. („Vlakec“ torej nima „lokomotive“.) Vsako osebo drži natanko en človek.

Nekaj časa traja, da se organizirajo, nato pa vlakec le začne pohod. Skozi režo v vratih jih opazuje vratar, ki sicer ne more videti vseh plesalcev naenkrat, ga pa vseeno zanima, v kakšni formaciji plešejo. Zato je v mislih oštevilčil plesalce s številkami od 1 do n in si na listek napisal svoja opažanja. Uredil jih je v tabelo z n števili: v i-to polje v tabeli je zapisal številko plesalca, ki ga drži plesalec i.

Opiši postopek, ki s pomočjo vratarjeve tabelice ugotovi, ali vsi plesalci plešejo v enem samem velikem krogu. V tem primeru naj postopek izpiše „`V KROGU`“, sicer pa „`PO SVOJE`“.

5. Kvadrati s seštevanjem

Napiši podprogram Kvadrati(n), ki izpiše kvadrate števil od 1 do n, pri tem pa ne sme uporabljati drugih aritmetičnih operacij kot seštevanje.[1] Tvoj podprogram naj bo takšne oblike:

```
void Kvadrati(int n);                  /* v C/C++ */
procedure Kvadrati(n: integer);        { v pascalu }
public static void kvadrati(int n);    // v javi
def Kvadrati(n): ...                   # v pythonu
```

[1] Zelo zanimivo različico naloge dobimo tudi, če zahtevamo le izračun n^2, ne pa tudi kvadratov manjših števil.

NALOGE ZA DRUGO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del v datoteki. Vse te možnosti so enakovredne. Odgovore, oddane prek računalnika, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev samodejno shranjuje in na zaslonu ti bo pisalo, kdaj se bo shranjevanje naslednjič zgodilo. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko bi rad začasno prekinil pisanje rešitve naloge ter se lotil druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu.

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. Poštne številke

Pošta Slovenije pogosto dobi pisma z nečitljivimi poštnimi številkami, saj ljudje pišejo tako, da je nekatere števke težko razbrati. Še posebej pogosto prihaja do zamenjave

- med 3 in 8,
- med 1, 4 in 7 ter
- med 5 in 6.

Napiši podprogram Variante(k), ki sprejme štirimestno poštno številko k in na zaslon izpiše vse možne poštne številke, ki jih lahko dobimo z naštetimi zamenjavami števk. Na primer, Variante(1035) na zaslon izpiše naslednje poštne številke, ne nujno v tem vrstnem redu: 1035, 1036, 1085, 1086, 4035, 4036, 4085, 4086, 7035, 7036, 7085, 7086.

Tvoj podprogram naj bo takšne oblike:

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

2. Reka presledkov

Imamo besedilo, zapisano tako, da so vsi znaki (tudi presledki, ločila ipd.) enako široki. Besedilo je raztegnjeno čez več vrstic.

Včasih se zgodi, da se presledki v več zaporednih vrsticah neugodno poravnajo in tvorijo „reko“ presledkov (*river*); če stran besedila pogledamo od daleč, je takšna reka videti kot moteča bela lisa. Za potrebe naše naloge bomo reko definirali kot zaporedje presledkov, za katere velja, da je vsak presledek v naslednji vrstici kot prejšnji in leži bodisi tik pod prejšnjim ali pa največ eno mesto levo ali desno od njega.

Napiši program, ki prebere besedilo s standardnega vhoda in izpiše dolžino najdaljše reke presledkov v njem. Besedilo naj bere vse do konca (EOF). Posamezna vrstica besedila je dolga največ 100 znakov. (Prazen prostor desno od konca posamezne vrstice ne šteje za presledke in zato ne more postati del reke.)

Primer: v besedilu na desni se najdaljša reka razteza prek 10 vrstic (od pete do štirinajste vrstice).

```
Two fierce and enormous bears, distinguished
by the appellations of Innocence and Mica
Aurea, could alone deserve to share the
favour of Maximin. The cages of those trusty
guards were always placed near the
bed-chamber of Valentinian, who frequently
amused his eyes with the grateful spectacle
of seeing them tear and devour the bleeding
limbs of the malefactors who were abandoned
to their rage. Their diet and exercises were
carefully inspected by the Roman emperor;
and, when Innocence had earned her discharge
by a long course of meritorious service, the
faithful animal was again restored to
the freedom of her native woods.
```

3. Delitev kamenja

Butalci in Tepanjčani so se skregali zaradi kupa kamenja, ki leži na meji med obema vasema. Butalci so trdili, da je kamenje njihovo, Tepanjčani pa prav tako. Na koncu so se zedinili, da je treba iz enega kupa narediti dva, po teži enaka. Ker pa je bil vroč dan in so imeli eno samo macolo, je butalski župan odredil, da se bode največ en kamen na dva kosa razbijal, ostali naj celi ostanejo.

Pomagaj razrešiti spor in jim **opiši postopek** (algoritem), s katerim bodo kup razdelili na dva enako težka kupa, upoštevaje Butalskega župana. Predpostaviti smeš, da je znana teža vsakega kamna (kamni niso nujno vsi enako težki) in da smemo en kamen razdeliti na dva v poljubnem razmerju. Postopek naj bo razložen podrobno in jasno, da ga bodo tudi Butalci razumeli.

4. Parktronic

Vsak bolj moderen avto ima vgrajene čuda veliko tehnike. Med drugim imajo marsikaj, da se ga ne razbije med parkiranjem, recimo ultrazvočne merilnike razdalje. Ko tak avto vozimo vzvratno, piska, ko zazna oviro na neki razdalji, in bližje ko smo, bolj tečno piska. **Napiši program**, ki nenehno meri razdaljo in primerno piska po spodaj opisanih navodilih.

1. Dokler je ovira oddaljena več kot en meter, ne piska;
2. ko je med 1 m in 0,5 m, piska s 400 Hz;
3. ko je med 0,5 m in 0,25 m, piska s 1000 Hz;
4. ko je bližje kot 0,25 m, pa piska z 2000 Hz.

Na voljo imaš naslednje sistemske funkcije/podprograme (zanje torej predpostavi, da že obstajajo in jih lahko tvoj podprogram pokliče, ko jih potrebuje):

- **void** Zvocnik(**int** frekvenca), ki sproži piskanje s podano frekvenco; če je podana vrednost 0, ugasne piskanje;
- **long** Cas(), ki pove trenutni sistemski čas v mikrosekundah od trenutka, ko se je prižgal računalnik v avtomobilu, na katerem teče tvoj program;
- **void** Ping(**unsigned char** x), ki odda ultrazvočni pulz, namenjen meritvi. V pulz ta funkcija zakodira vrednost x (število od 0 do 255), funkcija Poslusaj pa zna to število izluščiti iz signala, ki pride nazaj po odboju.
- **int** Poslusaj(), ki vrne -1, če v času od zadnjega klica ni slišala odboja nobenega ultrazvočnega merilnega pulza, oddanega s Ping; če pa je tak odboj slišala, vrne vrednost x, ki je bila ob pošiljanju tistega signala podana funkciji Ping. (Če je slišala več odbojev, vrne x za tistega, ki ga je slišala kot zadnjega.)

Predpostaviš lahko, da je hitrost zvoka 340 m/s.

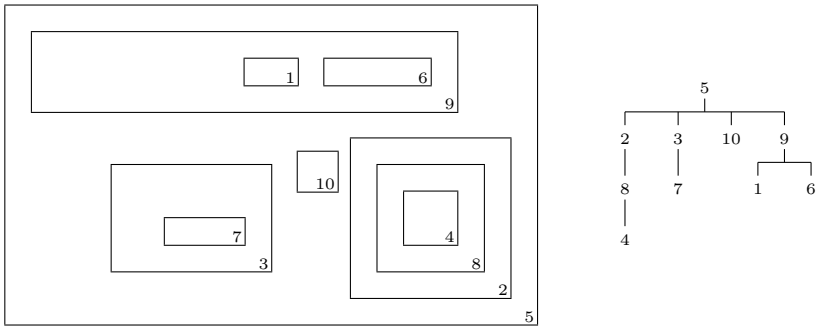
Še deklaracije v drugih jezikih:

```
procedure Zvocnik(Frekvenca: integer);        { v pascalu }
function Cas: integer;
procedure Ping(x: byte);
function Poslusaj: integer;

public static void zvocnik(int frekvenca);    // v javi
public static long cas();
public static void ping(byte x);
public static int poslusaj();

def Zvocnik(frekvenca): ...                   # v pythonu
def Cas(): ...
def Ping(x): ...
def Poslusaj(): ...
```

5. Pravokotniki

V ravnini imamo podanih n pravokotnikov. Njihove stranice so vzporedne koordinatnima osema, znane pa so tudi njihove koordinate. Za i-ti pravokotnik sta (x_{i1}, y_{i1}) koordinati njegovega spodnjega levega oglišča, (x_{i2}, y_{i2}) pa koordinati njegovega zgornjega desnega oglišča. Za vsak par pravokotnikov velja naslednje: bodisi je prvi vsebovan v drugem bodisi je drugi vsebovan v prvem bodisi nimata nobene skupne točke. Ne more se torej zgoditi, da bi se dva pravokotnika delno prekrivala, dotikala ali sekala, lahko pa leži eden v drugem. Poleg tega velja tudi, da obstaja med temi pravokotniki en tak, ki vsebuje vse ostale.

Iz teh omejitev sledi, da lahko pravokotnike uredimo hierarhično glede na to vsebovanost. Primer kaže naslednja slika:



Opiši postopek, ki za vsak pravokotnik našteje, kateri so njegovi neposredni podrejeni v tej hierarhiji vsebovanosti.

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.

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

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk. (Izjema je prva naloga, kjer je testnih primerov 20 in za pravilen odgovor pri posameznem testnem primeru dobiš 5 točk.) Nato se točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M - 3(N-1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

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

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

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. Kup knjig (`kup.in`, `kup.out`)

O'Reilly je pred kratkim izdal knjigo *Understanding ActionScript 2: Quirks and Annoyances*. Ker je o zadevi dosti povedati, je knjiga izšla v n zvezkih, oštevilčenih od 1 do n. Mitja, ki zadnje čase navdušeno programira v ActionScriptu 2, si je kupil vseh n. Ker na knjižnih policah nima prostora zanje, si jih je zložil kar v skladovnico na tla: čisto na dno je položil zvezek 1, nanj zvezek 2, nato zvezek 3 itd.

Med programiranjem je dostikrat potreboval kakšen zvezek nekje iz sredine kupa. V takšnem primeru ga je pač potegnil iz kupa; ko pa je v njem prebral, kar ga je zanimalo, se mu ga ni dalo tlačiti nazaj na njegovo mesto v kup, temveč ga je odložil kar na vrh. Vedno pa je zvezek vrnil na skladovnico, še preden je začel brati novega.

Napiši program, ki prebere zaporedje, v kakršnem je Mitja jemal zvezke in jih sproti vračal na vrh skladovnice, ter na koncu za vsak zvezek izpiše, kje v kupu se po tem zaporedju operacij nahaja.

Vhodna datoteka: v prvi vrstici sta dve celi števili, n (število zvezkov) in k (število pobiranj zvezkov); zanju velja $1 \le n \le 1000$ in $1 \le k \le 1\,000\,000$. Sledi k celih števil, vsako v svoji vrstici, ki po vrsti opisujejo, katere zvezke je Mitja jemal s kupa (vsaka od teh vrstic pove številko pobranega zvezka; to je celo število, večje ali enako 1 in manjše ali enako n).

V 40 % testnih primerov bo dodatno veljalo $k \le 2000$.

Izhodna datoteka: program naj izpiše n vrstic, pri tem pa naj i-ta vrstica vsebuje število zvezkov, ki po končanem prekladanju zvezkov ležijo med zvezkom i in tlemi.

Primer vhodne datoteke:

```
4 3
2
3
2
```

Pripadajoča izhodna datoteka:

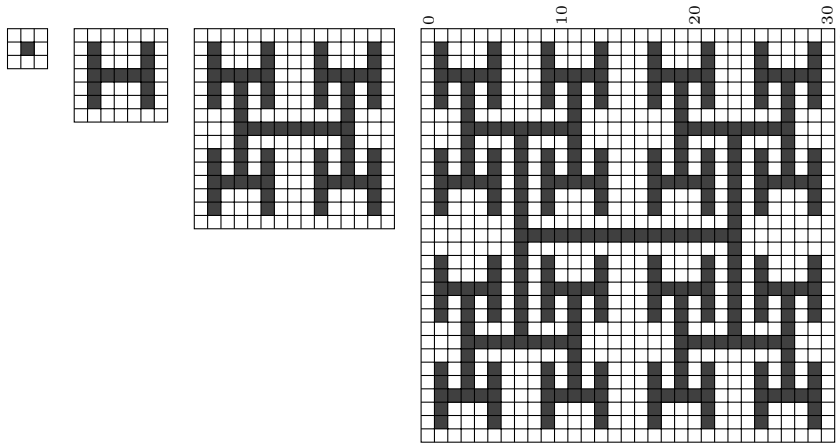
```
0
3
2
1
```

2. *H*-fraktal (`fraktal.in`, `fraktal.out`)

Pri tej nalogi bomo risali fraktale na kvadratni karirasti mreži. Vsako polje mreže je lahko črno ali belo. H-fraktal reda 0 je na mreži 3×3, pri čemer je srednje polje črno, ostala pa bela. H-fraktal reda $k+1$ dobimo tako, da vzamemo štiri kopije H-fraktala reda k, jih razporedimo v kvadrat, mednje postavimo eno vrstico in en stolpec belih polj, nato pa jih povežemo s črko H. Slika na str. 22 kaže H-fraktale reda 0, 1, 2 in 3. V taki mreži lahko vpeljemo koordinatni sistem: vrstice oštevilčimo od zgoraj navzdol (najbolj zgornja vrstica ima številko 0, tista tik pod njo ima številko 1 in tako naprej), stolpce pa od leve proti desni (najbolj levi stolpec ima številko 0, tisti tik desno ob njem ima številko 1 in tako naprej). Nekaj številk vrstic in stolpcev je prikazanih tudi na gornji sliki za H-fraktal reda 3.

Napiši program, ki prebere k in izriše nek pravokoten del H-fraktala reda k z znaki „`#`“ (za črna polja) in „`.`“ (za bela polja). V vhodni datoteki bosta podani koordinati zgornjega levega polja zahtevanega pravokotnika in njegova velikost.

Vhodna datoteka: vsebuje eno samo vrstico, v kateri je pet celih števil: k, x, y, w in h. Ločena so s po enim presledkom. Pri tem k pove red H-fraktala, ki nas zanima;



(x, y) sta koordinati zgornjega levega polja v pravokotniku, ki ga je treba izrisati, w je širina tega pravokotnika, h pa njegova višina. Koordinate in velikost pravokotnika so takšne, da pravokotnik ne gleda čez rob fraktala. Veljalo bo $0 \leq k \leq 28$, $x \geq 0$, $y \geq 0$, $1 \leq w \leq 100$, $1 \leq h \leq 100$. V 40 % testnih primerov bo veljalo tudi $k \leq 10$.

Izhodna datoteka: vanjo izpiši h vrstic, vsaka od teh naj vsebuje w znakov, vsak od teh znakov pa naj bo „`#`“ ali „`.`“. Izpisani znaki naj predstavljajo pravokotni del H-fraktala reda k, po katerem sprašuje vhodna datoteka.

Primer vhodne datoteke:

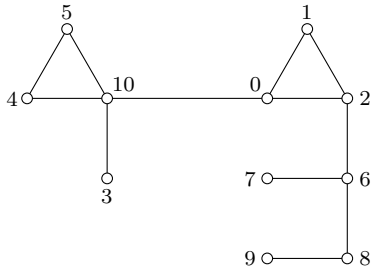
```
3 10 7 12 5
```

Pripadajoča izhodna datoteka:

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

3. Slepe ulice (`slepe.in`, `slepe.out`)

Ceste v Klozeljskem so nadvse ozke. Obračanje za 180 stopinj z avtom sredi ceste ali celo v križišču je zelo neugodno, zato je treba zelo paziti, da ne zapelješ v kakšno slepo ulico. Še posebej neprijetno je to za turiste, ki lokalnih cest ne poznajo dobro. Končno so to uvideli tudi mestni možje in sklenili, da kaže na začetke slepih ulic postaviti opozorilne table. Ker vsaka takšna tabla stane precej denarja, bi jih želeli postaviti samo toliko, kolikor je neobhodno nujno. Izognili bi se radi tablam na začetku slepih ulic, do katerih se lahko pripelješ samo skozi kakšno drugo slepo ulico. V tem primeru si namreč tablo gotovo že videl in nima smisla, da se postavlja še ena. (Natančnejša definicija: ulica je slepa natanko tedaj, ko obstaja taka smer vožnje po njej, za katero velja, da če se po ulici zapeljemo v tej smeri, se od takrat naprej po tisti ulici ne bomo več mogli peljati, ne da bi nekoč sredi kakšne ulice ali pa v križišču naredili obrat za 180 stopinj.)



Primer cestnega omrežja z 11 križišči in 12 ulicami. Slepe so ulice (10, 3), (2, 6), (6, 7), (6, 8) in (8, 9); tabli pa je treba postaviti le na (10, 3) in (2, 6).

Cestno omrežje Klozeljskega lahko opišemo tako, da vseh n križišč, ki jih imajo, oštevilčimo od 0 do $n - 1$, nato pa podamo m parov oblike (a_i, b_i), ki pomenijo, da sta križišči a_i in b_i povezani s cesto. Za potrebe takšnega označevanja rečemo „križišče“ tudi točkam, v katerih se konča le ena slepa ulica. Vse ceste so dvosmerne. Noben par križišč ni neposredno povezan z dvema ali več cestami. Cestno omrežje je povezano, torej se lahko iz vsakega križišča pripeljemo do vsakega drugega. Zagotovo obstajajo vsaj naslednje tri ulice: ulica med križiščema 0 in 1, ulica med 1 in 2 ter ulica med 0 in 2. Z drugimi besedami, križišča 0, 1 in 2 so povezana v majhno krožišče.

Napiši program, ki prebere opis cestnega omrežja in izpiše, na katere ceste je treba nujno postaviti opozorilno tablo za slepo ulico.

Vhodna datoteka: v prvi vrstici sta celi števili n (število križišč) in m (število ulic), ločeni s presledkom. Zanju bo veljalo $3 \leq n \leq 500\,000$ in $3 \leq m \leq 500\,000$. Sledilo bo m vrstic s po dvema številoma a_i in b_i, ki označujeta cestno povezavo med križiščema a_i in b_i.

Izhodna datoteka: vanjo izpiši ceste, na katere je treba postaviti table. Cesto opiši s številkama križišč, ki jo določata, ločenima s presledkom; pri tem najprej navedi križišče, *iz* katerega bi se avtomobili zapeljali v slepo ulico. Vsako cesto izpiši v svojo vrstico; vrstice naj bodo urejene naraščajoče glede na prvo številko križišča, v primeru enakosti pa še glede na drugo številko križišča.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
2 6
10 3
```

(Ta primer ustreza omrežju na gornji sliki.)

4. Križanka (`krizanka.in`, `krizanka.out`)

28. avgusta letos bo minilo okroglih sto let od razglasitve samostojnega kraljestva Črne gore. Ob tej priložnosti bo tamkajšnji časopis *Le Monde Negro* objavil gigantsko spominsko križanko velikosti $w \times h$ kvadratkov ($1 \leq w \leq 3000$, $1 \leq h \leq 3000$). Dežurni ugankar se je že odločil, kje v križanki bodo stali črni kvadratki, ni pa še izbral prostora za reklamno sliko. Želi si, da bi bila reklama čim večja — tako bo imel potem on sam manj dela, pa še oglaševalca lahko bolj pomolzejo.

Napiši program, ki pomaga ugankarju in za dano predlogo križanke (t.j. pravokotno mrežo belih in črnih kvadratkov) najde največji vsebovan bel pravokotnik.

Vhodna datoteka: v prvi vrstici sta celi števili w in h, ločeni s presledkom. Zanju velja $1 \leq w \leq 3000$, $1 \leq h \leq 3000$. Sledi h vrstic s po w znaki, ki na naraven način opisujejo križanko: vsak od teh znakov je bodisi pika („`.`“) in predstavlja istoležen bel kvadratek v križanki bodisi lojtra („`#`“) in predstavlja črn kvadratek.

V 20 % testnih primerov bo dodatno veljalo $w, h \leq 200$, v 60 % primerov bo $w, h \leq 500$ in v 80 % primerov bo $w, h \leq 1250$.

Izhodna datoteka: program naj vanjo izpiše eno samo celo število: največjo možno ploščino pravokotnega območja v križanki, ki sestoji izključno iz belih kvadratkov.

Primer vhodne datoteke:

```
7 6
....###
.#.....
.....#.
.#.....
##....#
..#...#
```

Pripadajoča izhodna datoteka:

```
12
```

5. Poštar (`postar.in`, `postar.out`)

Poštar je dobil zaposlitev v občini, kjer imajo zelo nenavadno prometno ureditev. Vse ulice so enosmerne in tvorijo pravokotno mrežo z w ulicami, po katerih je promet dovoljen v smeri od severa proti jugu, in h ulicami s prometom v smeri od zahoda proti vzhodu. Poleg te mreže ulic imajo tudi enosmerno avtocesto, ki povezuje jugovzhodno križišče s severozahodnim križiščem, kjer se nahaja tudi pošta. Poštar je dobil vrečo n pošiljk, ki jih mora iz poštne stavbe dostaviti na različna križišča v občini in se na koncu vrniti na pošto. Pošiljke lahko dostavlja v poljubnem vrstnem redu. Ker plačuje cestnino iz lastnega žepa, želi čim manjkrat uporabiti avtocesto. **Napiši program**, ki izračuna, najmanj kolikokrat se bo moral poštar peljati po avtocesti, da bo razvozil vse pošiljke.

Vhodna datoteka: v prvi vrstici so tri cela števila, w, h in n, ločena s po enim presledkom. Pri tem je w število navpičnih ulic, h število vodoravnih ulic, n pa število križišč, na katera mora poštar dostaviti pošiljke. Ta križišča so opisana v naslednjih n vrsticah; vsaka od teh vrstic vsebuje dve celi števili, ločeni s presledkom: najprej številko navpične ulice (od 1 do w) in nato številko vodoravne ulice (od 1 do h), ki se križata v tem križišču. Ta križišča so si paroma vsa različna (z drugimi besedami, nobeno križišče se v tem seznamu n križišč ne pojavi več kot enkrat). Pošta se nahaja na križišču $(1, 1)$; poštarju ne bo nikoli treba dostaviti pošiljke na križišče, kjer se nahaja pošta. Veljalo bo $2 \leq w \leq 100$, $2 \leq h \leq 100$, $1 \leq n < wh$.

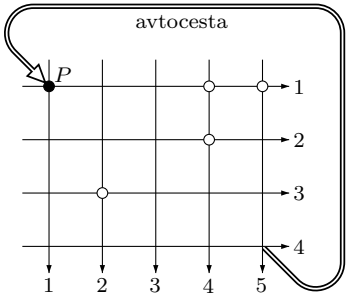
Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše potrebno število voženj po avtocesti, ki jih mora poštar opraviti, da lahko dostavi vse pošiljke in se na koncu vrne na pošto.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
3
```



Slika na desni prikazuje zemljevid, ki ustreza zgornjemu primeru vhodne datoteke. Pika P označuje položaj pošte, beli krožci $\circ$ pa križišča, na katera mora poštar dostaviti pošiljke.

NALOGE ZA ŠOLSKO TEKMOVANJE

29. januarja 2010

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.

1. Ruleta

Janez rad igra ruleto. Ker nima rad pretiranega tveganja, uporablja le stavo na barvo. Pri stavi na barvo je važno le, ali se kroglica ustavi na rdeči ali črni barvi. Če se ustavi na zeleni barvi (zelena je na ruleti le številka 0), vsi izgubijo. Če se kroglica ustavi na barvi, na katero smo stavili, dobimo dvojno vsoto tiste, ki smo jo stavili (torej je naš dobiček enak stavljenemu znesku), če pa na nasprotni (ali pa na zeleni 0), stavljeno vsoto izgubimo.

Pri odločanju, na katero barvo bo stavil, Janez uporablja naslednjo strategijo: vedno stavi žeton za 10 evrov na barvo, različno od tiste, ki je bila izžrebana v prejšnjem krogu. V prvem krogu ali pa, če je v prejšnjem krogu bila izžrebana zelena 0, vedno stavi na rdečo barvo. Prav tako nikoli ne igra več kot 12 iger.

Denimo, da potek dogajanja opišemo s številom. To je sestavljeno iz števk 1, 2 in 3. Števka 1 na i-tem mestu števila (od leve) pomeni, da je bila v i-tem krogu izžrebana rdeča, 2 črna in 3 zelena.

Napiši program, ki prebere število, sestavljeno iz števk 1, 2 in 3, in zanj izpiše, koliko denarja je Janez priigral (oziroma izgubil).

Primera:

- `1112232212` $\longrightarrow$ zaslužek $= -20$
  Razlaga: zaslužki so bili: $+10, -10, -10, +10, -10, -10, -10, -10, +10, +10$.

- `12213331212` $\longrightarrow$ zaslužek $= 30$
  Razlaga: zaslužki so bili: $+10, +10, -10, +10, -10, -10, -10, +10, +10, +10, +10$.

(Opomba: tvoja rešitev naj izpiše le skupni zaslužek po koncu vseh krogov, ne pa tudi zaslužkov v posameznih krogih.)

2. Glava e-pošte

Dolge vrstice (polja) v glavi e-poštnih sporočil se lahko razlomi na krajše pred kakšnim od obstoječih presledkov. Tako nastale nadaljevalne vrstice se zato vedno začnejo s presledkom; tiste nastale vrstice, ki nimajo na začetku presledka, torej začenjajo novo polje (novo prvotno vrstico).

Zgornje besedilo bi na primer lahko na ta način razlomili takole:

```
Dolge vrstice (polja) v glavi
 e-poštnih sporočil
 se lahko razlomi na krajše pred kakšnim
od obstoječih presledkov. Tako nastale nadaljevalne vrstice se zato vedno začnejo s
presledkom;
 tiste
 nastale
 vrstice, ki nimajo...
```

Napiši program, ki bo prebral tako razlomljeno glavo e-poštnega sporočila in jo prepisal na izhodno datoteko tako, da bodo morebitne razlomljene vrstice spet združene v prvotne dolge vrstice (polja). Na vhodni datoteki je le glava enega samega sporočila (lahko tudi prazna), ki se konča s koncem datoteke. Če ti je lažje, lahko predpostaviš, da nobena vrstica vhodne datoteke ni daljša od 100 znakov; vrstice, ki bodo nastale po združevanju, pa so seveda lahko tudi veliko daljše.

3. Trajekt

Na nekem otočku se ob izteku zadnjega sončnega dne pred napovedanimi dolgotrajnimi padavinami večina turistov odloči oditi domov. Seveda vsi izkoristijo zadnje minute dneva, nakar se z avtomobili optimistično odpravijo na pomol, kjer jih čaka zadnji večerni trajekt. Prostor na trajektu je seveda omejen, zato je potrebno narediti izbor vozil, ki bodo lahko še ta dan nadaljevala pot po celini. Vsi ostali bodo žal primorani podaljšati dopust vsaj še za eno noč.

Kapitan se odloči, da bo izbral vozila tako, da jih bo uspel na trajekt stlačiti čim več. Omejitve so vezane na njihovo skupno dolžino, saj se vozila naložijo eno za drugim vzdolž celotne dolžine trajekta. Tvoja naloga je **opisati postopek**, ki bo poiskal največje možno število vozil, ki jih lahko izberemo, ne da bi njihova skupna dolžina presegla dolžino trajekta. Poleg tega tudi utemelji, zakaj je tvoja rešitev pravilna (torej zakaj je število vozil, ki jih uspe spraviti na trajekt, res največje možno).

Kot vhodni podatek dobi tvoj postopek dolžino trajekta ter seznam dolžin vseh vozil.

4. Ledene dobe

Geolog na podlagi raziskav polarnega ledu predpostavlja, da povečanju koncentracije ogljikovega dioksida v zraku najkasneje čez pet tisoč let sledi obdobje zelo nizkih temperatur (ledena doba).

Geologi dobijo iz ledu naslednje podatke:

- seznam let, ko je bila vsebnost ogljikovega dioksida zelo visoka
- seznam let, ko je bil zaznan začetek ledene dobe.

Opiši postopek, ki bo geologu povedal, v koliko primerih njegova predpostavka drži (torej: za koliko let z visoko koncentracijo ogljikovega dioksida velja, da je v naslednjih 5000 letih nastopila kakšna ledena doba). Kot vhodni podatek dobi oba seznama, vsak od njiju pa je že urejen naraščajoče (od zgodnejših letnic proti kasnejšim). Predpostaviš lahko, da sta oba seznama že podana v pomnilniku, vsak v svoji tabeli ali kakšni drugi primerni podatkovni strukturi; torej se ti ni treba ukvarjati s tem, kako bi seznama bral iz kakšne datoteke. Vhodna seznama sta lahko dolga, zato naj bo tvoja rešitev učinkovita (bolj učinkovite rešitve dobijo več točk).

Primer:

- visoke koncentracije ogljikovega dioksida: $-130000, -100000, -20000$
- začetki ledenih dob: $-200000, -127000, -50000, -18000$

Rezultat: 2 (letoma -130000 in -20000 sta sledili ledeni dobi -127000 in -18000 v roku kvečjemu 5000 let, letu -100000 pa ni sledila nobena ledena doba znotraj tega roka).

(Poučna povezava: `http://en.wikipedia.org/wiki/Ice_core`)

5. Fora

Mislimo si k vgnezdenih zank; i-ta zanka med njimi izvede n_i iteracij, v vsaki iteraciji pa najprej izpiše število i in nato izvede naslednjo zanko. Njihovo delovanje bi torej lahko zapisali takole:

V pascalu:

```
for a1 := 1 to n1 do begin
  WriteLn(1);
  for a2 := 1 to n2 do begin
    WriteLn(2);
    ...
      for ak := 1 to nk do begin
        WriteLn(k);
      end; {for ak}
    ...
  end; {for a2}
end; {for a1}
```

V C-ju:

```
for (a1 = 0; a1 < n1; a1++) {
  printf("%d\n", 1);
  for (a2 = 0; a2 < n2; a2++) {
    printf("%d\n", 2);
    ...
      for (ak = 0; ak < nk; ak++) {
        printf("%d\n", k);
      } /* for ak */
    ...
  } /* for a2 */
} /* for a1 */
```

V pythonu:

```
for a1 in range(n1):
  print 1
  for a2 in range(n2):
    print 2
    ...
      for ak in range(nk):
        print k
```

V javi:

```
for (a1 = 0; a1 < n1; a1++) {
  System.out.println(1);
  for (a2 = 0; a2 < n2; a2++) {
    System.out.println(2);
    ...
      for (ak = 0; ak < nk; ak++) {
        System.out.println(k);
      } /* for ak */
    ...
  } /* for a2 */
} /* for a1 */
```

Opiši postopek, ki prebere zaporedje števil, kot ga izpišejo zgornje zanke, in iz njega ugotovi, kakšne so bile vrednosti k, n_1, n_2, ..., n_k. Predpostaviš lahko, da bo $1 \leq k \leq 100$ in da bo vhodno zaporedje res takšne oblike, kakršno lahko izpišejo zgoraj predstavljene vgnezdene zanke. Vhodno zaporedje lahko tvoj postopek bere iz datoteke ali pa predpostavi, da je že na voljo v pomnilniku v neki tabeli ali kakšni drugi primerni podatkovni strukturi. Vhodno zaporedje je lahko precej dolgo; bolj učinkovite rešitve bodo dobile več točk.

NEUPORABLJENE NALOGE IZ LETA 2008

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 3. tekmovanjem IJS v znanju računalništva (leta 2008), 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. 79–117) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Statistika

Napiši program, ki prebere besedilo, v katerem so besede med seboj ločene z znakom `' '`, stavki pa s katerim od znakov `'.'`, `'?'` in `'!'`. Program naj izpiše povprečno število črk in zlogov v besedi besedila in povprečno število besed v stavku besedila. Da bo naloga lažja, predpostavi, da je zlogov toliko kot samoglasnikov in da se v besedilu ne pojavljajo drugi znaki kot črke, presledki in zgoraj našteta končna ločila.

2. Brez ene črke

Napiši program, ki s standardnega vhoda prebira stavke (vsak je v svoji vrstici) in na koncu za vsako črko abecede izpiše, kako dolg je bil najdaljši izmed tistih stavkov, v katerem se ta črka ni nikoli pojavila.

Pri tem definiramo dolžino stavka kot število črk v njem — presledkov, ločil in morebitnih ostalih znakov ne štejemo.

Da bo naloga lažja, lahko predpostaviš, da se v stavkih ne pojavljajo druge črke kot male črke angleške abecede (od `'a'` do `'z'`).

3. Izpis HTMLja

Mislimo si preprost dokument v jeziku HTML — poleg besedila vsebuje še oznake oblike `<ime>` in `</ime>`, pri čemer je „`ime`“ ime enega od elementov jezika HTML. (Jezik HTML dovoli v oznakah sicer tudi še presledke in atribute, v besedilu pa komentarje, vendar predpostavimo, da v naših dokumentih teh reči ne bo.)

Napiši program, ki prebere s standardnega vhoda nek takšen dokument v jeziku HTML in za tisti del dokumenta, ki leži med oznakama `<body>` in `</body>`, izpiše vse besedilo (ne pa tudi oznak). Pri izpisu naj tudi skoči v novo vrstico na vsakem takem mestu, kjer se v vhodnem dokumentu pojavi oznaka `<br>`.

4. Volitve

Galaktična federacija voli svojega predsednika po naslednjem postopku. Vsak planet pošlje na volitve enega predstavnika, ki odda svoj glas za enega od kandidatov za predsednika. Za vsak planet je tudi znano, okoli katere zvezde kroži. Za posameznega kandidata pravimo, da je dobil podporo določene zvezde, če je med predstavniki planetov, ki krožijo okoli te zvezde, dobil več glasov kot kateri koli drugi kandidat.

Kandidat, ki je dobil podporo več kot polovice vseh zvezd v federaciji, je zmagovalec volitev. (Posebej opozorimo na to, da ta pravila dopuščajo možnost, da podpore določene zvezde ne dobi noben kandidat, pa tudi možnost, da volitve sploh nimajo zmagovalca.)

Napiši program, ki prebere podatke o glasovanju in izpiše, kdo je na volitvah zmagal. V prvi vrstici vhodnih podatkov je zapisano število kandidatov (recimo n), število zvezd v federaciji (recimo m) in število planetov (recimo k). Sledijo podatki o tem, kako so glasovali predstavniki posameznih planetov; za vsakega od njih je ena vrstica, v kateri je številka zvezde, okoli katere kroži ta planet (od 1 do m) in številka kandidata, za katerega je glasoval ta predstavnik (od 1 do n). Tvoj program naj izpiše številko kandidata, ki je zmagal na volitvah; če pa po gornjih pravilih zmagovalca ni, naj izpiše 0. Veljalo bo $n \leq 10$, $m \leq 1000$, $k \leq 10000$.

5. Polinomi

V butalski osnovni šoli vsi računajo na prste ali z računalnikom, zato srednje šole, kjer je treba znati premetavati x-e, pa tega kalkulator ne zna, nihče ne izdela. Pa je nekemu Tepanjčanu prišlo na misel, da bi začel proizvajati računalnike, ki bi poznali osnovne operacije računanja z veččleniki.

Primer: $(2 \cdot x - 3) + (3 \cdot x^2 - 2 \cdot x + 1)$ bi morali znati sešteti v $3 \cdot x^2 - 2$.

Napiši program, ki prebere podatke o dveh takih veččlenikih in izpiše njuno vsoto ali razliko.

Vhodni podatki:

- prva vrstica: računska operacija (znak „`+`“ ali „`-`“) in dve števili m in n;
- druga vrstica: n parov (celih) števil a_i b_i, ki predstavljajo prvi veččlenik $a_i \cdot x^{b_i}$;
- tretja vrstica: m parov (celih) števil, ki po istem kopitu predstavljajo drugi veččlenik.

Zgornji primer bi bil predstavljen takole:

```
+ 2 3
2 1 -3 0
3 2 -2 1 1 0
```

Izhodni podatki:

- prva vrstica: število l
- druga vrstica: l parov celih števil a_i b_i, kjer a_i ne sme biti 0.

Vsoto iz zgornjega primera bi zapisali takole:

```
2
3 2 -2 0
```

Ker imajo Butalci samo 20 prstov, lahko predpostaviš, da sta $1 \leq m, n \leq 20$.

6. Kalkulator

Napiši podprogram, ki bo v neskončni zanki upravljal kalkulator:

- na začetku naj izpiše 0;
- potem bere tipke, sestavlja števila, računa in izpisuje rezultat, v neskončnost.

Na voljo imaš naslednja podprograma:

```
char PocakajTipko(); /* vrne '0'..'9', '+', 'C' */
void PrikaziNaZaslonu(int stevilo);
```

Kalkulator naj deluje takole:

- dokler uporabnik tipka številke, naj program sestavlja število;
- ko uporabnik pritisne `+`, shrani število, ki je na zaslonu, v nek vmesni pomnilnik; število pusti na zaslonu, prva naslednja številka ga mora izbrisati;
- potem spet sprejema številke;
- ko uporabnik naslednjič pritisne `+`, sešteje vmesni pomnilnik in tisto, kar je na zaslonu, in izpiše rezultat; rezultat shrani v vmesni pomnilnik;
- ko uporabnik pritisne `C`, pobriše zaslon in vmesni pomnilnik (torej postavi oboje na 0).

Zaslon je širok 6 znakov; če uporabnik natipka več kot toliko številk, odvečne ignoriramo; če ima vsota več kot 6 številk, naj izpiše `999999`.

Primer:

Uporabnik pritisne:	Vmesni pomnilnik:	Zaslon:
(začetek)	0	0
2	0	2
3	0	23
5	0	235
+	235	235
1	235	1
4	235	14
+	249	249
1	249	1
+	250	250
+	250	250
C	0	0

7. Tečajnica

Na borzi kotira veliko število delnic, ki jih označujejo nizi znakov. Cene delnic se spreminjajo, tako da posamezna cena velja le na nekem časovnem intervalu.

V pomnilniku našega računalnika bi radi hranili podatke o cenah delnic v preteklosti. Vsak zapis bo torej sestavljen iz štirih delov: (1) oznaka delnice, (2) cena, (3) datum začetka veljavnosti te cene in (4) datum konca veljavnosti te cene.

Nad temi podatki bi bili radi sposobni izvajati naslednji dve operaciji:

- dodajanje novih zapisov;

- iskanje po zapisih, in sicer tako, da je podana oznaka delnice in nek datum, nas pa zanima, kakšna je bila cena delnice na tisti datum.

Opiši, kako bi predstavil podatke v pomnilniku, da bi lahko potem na njih izvajal zgornji dve operaciji, in za vsako od teh dveh operacij **opiši postopek**, ki jo izvede. Pri tem pazi predvsem na to, da naj bo postopek za iskanje čim bolj učinkovit. Če zmoreš, poskusi poskrbeti, da bo tudi postopek za dodajanje zapisov učinkovit, vendar to ni tako nujno.

Predpostaviš lahko, da se bo zapise dodajalo po naraščajočem času; če je torej pred nekim dodajanjem najnovejši zapis za neko delnico določal njeno ceno do datuma d, zdaj pa se dodaja nov zapis za to delnico, ki določa njeno ceno od datuma d' naprej, lahko predpostaviš, da datum d' ni zgodnejši od datuma d.

Pri dodajanju se lahko zgodi, da kakšno časovno obdobje pri kakšni delnici ni pokrito (za tisto obdobje torej ne poznamo cene te delnice). **Opiši**, kako se tvoj postopek za iskanje obnaša v tem primeru.

Opomba: pri tej nalogi ni treba pisati izvorne kode za konkretno implementacijo tvojih postopkov. Če pa hočeš, si lahko za vodilo vzameš naslednji deklaraciji (tvoja dva postopka morata opisovati delovanje takšnih dveh funkcij):

```
procedure DodajZapis(Delnica: string; Zacetek, Konec: Datum; Cena: real);
function CenaDelnice(Delnica: string; Kdaj: Datum): real;
```

ali, v C-ju:

```
void DodajZapis(char *delnica, Datum zacetek, Datum konec, double cena);
double CenaDelnice(char *delnica, Datum kdaj);
```

8. Okvarjen pomnilnik

Imamo nek pomnilniški čip, ki hrani 2^n bytov pomnilnika. S procesorjem je povezan z n naslovnimi linijami in 8 podatkovnimi linijami. Procesor pri branju in pisanju prek n naslovnih linij sporoči pomnilniku naslov celice, iz katere bi rad bral ali vanjo pisal (naslov je torej n-bitno celo število od 0 do $2^n - 1$); pri branju mu potem pomnilnik na osmih podatkovnih linijah vrne vrednost byta, ki je trenutno shranjen v zahtevani celici; pri pisanju pa na podatkovne linije procesor pošlje vrednost, ki jo mora pomnilnik potem zapisati v zahtevano celico.

(*a*) Ker se je na čipu zaradi tresljajev razrahljala ena od nožic, se pomnilnik zdaj obnaša tako, kot da bi bila ena od naslovnih linij ves čas enaka 0 — torej pomnilnik

deluje tako, kot da bi imel zahtevani naslov na tistem mestu bit z vrednostjo 0, ne glede na to, ali jo ima res ali ne.

Primer: recimo, da je okvarjena linija 2 in hoče procesor dostopati do pomnilniške celice z naslovom 69. V dvojiškem zapisu je $69_{10} = 1000\underline{1}01_2$ (bit 2 je podčrtan). Zaradi okvare linije 3 bo naš pomnilniški čip zahtevano operacijo (branje ali pisanje) v resnici izvedel na celici z naslovom $1000\underline{0}01_2$, torej 65_{10}.

Napiši podprogram Preveri, ki ugotovi, katera naslovna linija se je pokvarila, da jo bo serviser lahko prilotal nazaj. Za potrebe te naloge predpostavi, da vidi tvoj program naš pomnilniški čip kot neko tabelo, veliko 2^n bytov, njen naslov pa dobi kot parameter:

```
const n = ...;
type byte= 0..255;
     PomnilnikT = packed array [0..(1 shl n) − 1] of byte;
{ Tvoj podprogram naj bo takšne oblike: }
procedure Preveri(var M: PomnilnikT);
```

Ali, v C/C++:

```
#define n ...
typedef unsigned char PomnilnikT[1 << n];
void Preveri(PomnilnikT M);
```

(*b*) **Opiši postopek**, ki bi rešil problem tudi v primerih, ko je lahko okvarjenih več naslovnih linij in so lahko okvarjene tudi nekatere podatkovne linije. (Ni pa ti treba pisati implementacije v kakšnem konkretnem programskem jeziku.) Ali obstajajo primeri, ko ne moreš ugotoviti, katere linije so okvarjene?

(*c*) Doslej smo ves čas predpostavljali, da so okvarjene linije „zataknjene“ na vrednosti 0. Kaj pa, če se lahko zgodi tudi, da so nekatere linije „zataknjene“ na vrednosti 1? **Opiši postopek** za testiranje pomnilnika pod temi pogoji, in razloži, kaj se da v tem primeru ugotoviti o tem, katere linije so zataknjene in na kakšnih vrednostih.

9. Tiskalnik

Na neki šoli želijo dijakom omogočiti tiskanje določenega števila strani na leto preko centralnega strežnika, pri čemer lahko dijaki, ki bi radi tiskali še za lastne potrebe, dokupijo kvoto.

Vsak dijak ob začetku leta dobi svojo zaporedno številko od 1 do n (in seveda geslo, sicer bi Janez lahko tiskal na Metkin račun). Prosili so te, če jim lahko pomagaš pri pisanju programa, ki bo skrbel za vodenje števila izpisanih strani in kupljenih kvot. Sistem mora poskrbeti, da dijaki ne bodo tiskali več od dovoljene kvote, razen če dokupijo ustrezno število strani. Če velikost dokumenta, ki ga pošlje dijak na tiskalnik, presega preostalo kvoto, naj tiskalnik ne natisne ničesar.

Predpostaviš lahko, da je dijakov manj kot 1000 in uporabljaš globalne spremenljivke (seveda jih deklariraj!). (V praksi bi se te podatke zapisovalo v bazo namesto v globalne spremenljivke.)

Prosili so te, da **napišeš** naslednje **podprograme** (klical jih bo glavni program):

- **procedure** ZacniSolskoLeto(SteviloDijakov, SteviloStrani: integer);
  void ZacniSolskoLeto(**int** SteviloDijakov, **int** SteviloStrani);

  Ta podprogram bo poklican na začetku šolskega leta in naj vsem dijakom (ki jih je SteviloDijakov) nastavi kvoto tiskanja na SteviloStrani.

- **function** NatisnjenihStrani: integer;
  int NatisnjenihStrani();

  Ta funkcija naj vrne skupno število natisnjenih strani od začetka šolskega leta.

- **function** KvotaDijaka(IdDijaka: integer): integer;
  int KvotaDijaka(**int** IdDijaka);

  Ta funkcija naj dijaku z z zaporedno številko IdDijaka (te zaporedne številke so med 1 in SteviloDijakov) pove, koliko strani še lahko natisne.

- **procedure** PovisajKvoto(IdDijaka, StStrani: integer);
  void PovisajKvoto(**int** IdDijaka, **int** StStrani);

  Ta funkcija naj dijaku IdDijaka poviša kvoto tiskanja za StStrani.

- **function** NatisniDokument(IdDijaka, IdDokumenta, StStrani: integer): integer;
  int NatisniDokument(**int** IdDijaka, **int** IdDokumenta, **int** StStrani);

  Ta funkcija naj pošlje dokument z identifikacijsko številko IdDokumenta na tiskalnik in poskrbi, da se dijaku ustrezno zniža kvota tiskanja. Če tiskanje dokumenta ni mogoče, naj funkcija vrne -1, sicer pa število natisnjenih strani.

Na voljo imaš funkcijo:

function PosljiTiskalniku(IdDokumenta: integer): boolean;
bool PosljiTiskalniku(**int** IdDokumenta);

Ta funkcija vrne true, če je bil dokument uspešno natisnjen, in false, če je med tiskanjem prišlo do napake (okvara na tiskalniku, zmanjkalo papirja, ...).

Opiši tudi, kaj se zgodi, če na tiskalnik en dijak naenkrat pošlje večje število dokumentov. Ali tvoja rešitev poskrbi, da dijak kljub temu ne more preseči kvote?

10. Pobiranje smeti

Imamo robota za pobiranje smeti na neki ravni cesti (premici). Na cesti je n smeti in njihovi položaji so znani (podana je njihova oddaljenost v metrih relativno glede na začetni položaj robota; pozitivna števila pomenijo desno od začetnega položaja, negativna pa levo od njega). Robot mora pobrati vseh n smeti po nepadajoči oddaljenosti od svojega začetnega položaja in se na koncu vrniti nazaj na začetni položaj. Torej, če so smeti razporejene takole:

$$-8 \quad -5 \quad -4 \quad -3 \quad -2 \quad 1 \quad 4 \quad 6 \quad 14$$

gre najprej po 1, potem po -2, potem po -3, potem mora pobrati -4 in 4 (lahko najprej -4 in nato 4, lahko pa obratno), potem -5, potem 6, potem -8, potem 14. **Opiši postopek**, ki iz položajev smeti izračuna dolžino najkrajše poti, s katero lahko robot pobere vse smeti in pri tem upošteva zgoraj navedene omejitve.

11. Koščki in SG-1

Igitur qui desiderat pacem, praeparet bellum.

Ekipa SG-1 ponovno rešuje planet P2X-416 (in ubogo ljudstvo, ki živi na njem) pred pogubo, ki jo predstavljajo Goa'uldi. Sejalci smrti se že spuščajo v atmosfero in napadajo, zato je treba hitro ukrepati. Na srečo SG-1 in domačinov je slavni Thor na tem planetu skril mogočno orožje, ki ga lahko uporabijo za obrambo, vendar je vhod v orožarno primerno zavarovan. Ker je hotel, da ljudje dosežejo dovolj visoko stopnjo inteligence, preden jim bo zaupal to orožje (sicer bi ga kaj hitro zlorabili in se sami pogubili), vhod varuje primerno težka naloga.

Daniel Jackson jo je že prevedel, gre pa takole. Imaš ploščice (res, našli so n ploščic) z nekimi desetiškimi števili. Število na vsaki ploščici je brez vodilnih ničel in ima lahko do 100 števk. Zlepi ploščice skupaj v takem vrstnem redu, da bo število, ki ga dobiš, najmanjše med vsemi možnimi, ki jih je mogoče tako dobiti.

Primer: če imaš štiri ploščice s številkami 354, 57, 910 in 574, jih lahko zlepiš v število 57435457910 ali pa v število 35491057574 in tako naprej. Nas pa zanima najmanjši tak zlepek; v tem primeru se izkaže, da je to 91057574354. **Opiši postopek**, ki za dani seznam ploščic (oz. številk na njih) poišče največji tak zlepek. Predpostaviš lahko, da je $n \leq 50000$ in da so števila na ploščicah pozitivna cela števila s po največ 100 števkami in brez vodilnih ničel.

12. Kolesarjeva pot

Kolesar se odpravi na pot s kolesom. S seboj vzame merilec srčnega utripa, ki je hkrati tudi brzinomer, in ki meri naslednje podatke:

- dolžina doslej prevožene poti (ob pričetku poti 0);
- porabljen čas od začetka kolesarjenja;
- srčni utrip.

Ko se kolesar vrne, vnese kolesarsko pot na spletni strani z zemljevidi (na primer *geopedia.si*), tako da zariše mnogokotnik, po katerem se je peljal. Ker ima spletna stran tudi podateke o nadmorski višini, lahko kolesar izvozi datoteko, kjer vsaka vrstica predstavlja eno oglišče mnogokotnika, in sicer zemljepisno dolžino (prvi stolpec), širino (drugi stolpec) in nadmorsko višino (tretji stolpec).

Pomagaj kolesarju **napisati podprogram** ZdruziPodatke, ki podatke iz merilca srčnega utripa združi s podatki iz zemljevida. Podprogram naj za vsako meritev najde tisto točko na poti, začrtani na zemljevidu, ki je od začetka poti (torej prve točke) oddaljena toliko, kot je bil v tisti točki odmerek prevožene poti. Tvoj podprogram naj bo takšne oblike:

```
void ZdruziPodatke(MeritevT *meritve, int stMeritev, TockaT* tocke, int stTock);
```

V pomoč naj ti bosta tale podprograma, za katera lahko predpostaviš, da že obstajata:

```
/* vrne razdaljo med točkama A in B */
double Razdalja(TockaT A, TockaT B);

/* vrne točko, ki je na daljici AB za x oddaljena od A */
TockaT NajdiTocko(TockaT A, TockaT B, double x);
```

in podatkovne strukture

```
typedef struct {
   double x;          { zemljepisna dolžina (v metrih) }
   double y;          { zemljepisna širina (v metrih) }
   double z;          { nadmorska višina (v metrih) }
} TockaT;

typedef struct {
   double x;          { zemljepisna dolžina (v metrih) }
   double y;          { zemljepisna širina (v metrih) }
   double z;          { nadmorska višina (v metrih) }
   double pot;        { prevožena pot (v metrih) }
   double cas;        { čas od začetka vožnje (v sekundah) }
   double utrip;      { srčni utrip (v udarcih na minuto) }
} MeritevT;
```

Ob klicu tvojega podprograma ZdruziPodatke bodo v tabeli meritve veljavne vrednosti le v poljih pot, cas in utrip, tvoj podprogram pa naj za vsako meritev izračuna ustrezne koordinate in jih shrani v polja x, y in z.

13. Igra

Igralca A in B se igrata naslednjo igro: najprej si izbereta naključno število n in na tablo zapišeta število 1. Nato pa izmenoma vzameta število na tabli, ga pomnožita z enim od števil med 2 in 9 in rezultat napišeta na tablo. Zmaga tisti igralec, ki prvi zapiše na tablo število, ki je večje od na začetku izbranega n. **Opiši postopek**, ki za dano število n odgovori, kdo bo zmagal, če privzame, da bosta oba igralca igrala optimalno.

14. Bézierjeva krivulja

Bézierjeva krivulja reda d je opisana z $d+1$ kontrolnimi točkami, $A_0, \ldots, A_d$. Definirajmo zdaj $A_{ii}(t) = A_i$ (za $i = 0, \ldots, d$) in $A_{ij}(t) = (1-t) \cdot A_{i,j-1}(t) + t \cdot A_{i+1,j}(t)$ za $0 \le i < j \le d$. Krivuljo tvorijo točke $A_{0d}(t)$ za vse t z intervala $[0, 1]$.

Primer za $d = 3$ in $t = 2/5$ vidimo na sliki na str. 39.

Vpliv parametra t se kaže pri razmerju, v katerem smo postavljali nove točke na daljice:

$$\frac{A_0A_{01}}{A_{01}A_1} = \frac{A_1A_{12}}{A_{12}A_2} = \frac{A_2A_{23}}{A_{23}A_3} = \frac{t}{1-t}$$

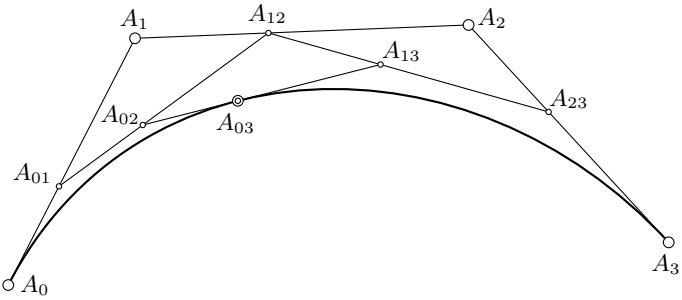
$$\frac{A_{01}A_{02}}{A_{02}A_{12}} = \frac{A_{12}A_{13}}{A_{13}A_{23}} = \frac{t}{1-t}$$

$$\frac{A_{02}A_{03}}{A_{03}A_{13}} = \frac{t}{1-t}$$

Napiši podprogram, ki za dane $d, A_0, \ldots, A_d$ in t izračuna $A_{0d}(t)$.

15. Loto

Hazarder Srečko ima recept za igranje Lota. Pravi, da zmagovalna „sedmica“ sestoji iz najpogosteje izžrebanih številk Lota, ki jim dodaš najmanjšo številko, ki je bila v zgodovini izžrebana največkrat zapored!



Primer Bézierjeve krivulje za $d = 3$. Če kontrolne točke A_0, A_1, A_2, A_3 povežemo z daljicami in jih delimo v razmerju $t : (1 - t)$, pridemo sčasoma do $A_{03}(t)$, ki leži na krivulji. Na sliki vidimo primer za $t = 2/5$. Ko preleti t celoten interval od 0 do 1, pa dobimo ravno celo krivuljo (ki je narisana z debelo črto).

Igra Loto se igra z 39 številkami, od katerih jih sedem tvori zmagovalno kombinacijo. Na standardnem vhodu je v vsaki vrstici podana po ena zmagovalna „sedmica“ iz preteklosti; številke so ločene s presledki. **Napiši program**, ki prebere te podatke in izpiše dobitno kombinacijo.

16. Poplavljanje labirinta

V labirintu (zidovi so označeni z „`X`“, prazna polja pa s pikami „`.`“) je z zvezdico „`*`“ označen položaj vira vode. Nekega dne pride do poplave. **Napiši program**, ki prebere tak opis labirinta in izpiše isti labirint, v katerem pa naj z znaki „`~`“ označi vsa polja, ki jih bo zalila voda. Primer:

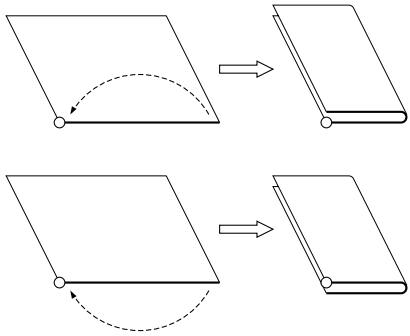
```
XXXXXXXXX          XXXXXXXXX
X..X..X.X    →     X~~X~~X.X
X.*...XXX          X~~~~~XXX
XXXXXXX..          XXXXXXX..
```

Predpostavi, da se voda lahko širi z enega polja na drugo le, če sta obe prazni in imata skupno stranico.

17. Prepogibanje papirja

Vzemimo pravokoten list papirja in ga prepognimo čez pol, tako da se oba prepognjena konca prekrivata. En konec lista držimo v levi roki (na spodnjih slikah je ta konec označen z ∘), drugega v desni in list zapognemo tako, da desni konec primaknemo k levemu. Tako lahko ločimo prepogib gor in prepogib dol, kot kaže slika na str. 40.

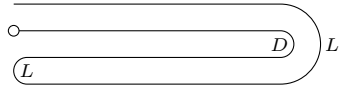
Če bi list razgrnili, bi se na sredini poznala črta, po kateri smo list prepognili. Toda mi ga ne bomo razgrnili, ampak ga bomo prepognili še enkrat na isti način, tako da se bodo sedaj prekrivale že štiri plasti. Ta postopek lahko še večkrat ponovimo. Ko list na koncu razgrnemo, se bodo na papirju poznale črte, po katerih smo ga prepognili. Če dovolj dobro pogledamo, bomo opazili, da je papir po teh črtah včasih upognjen v eno, včasih pa v drugo stran. (Glej sliko spodaj.) Vzorce, ki



Ilustracija k nalogi Prepogibanje papirja. Zgornja slika prikazuje prepogib gor, spodnja pa prepogib dol. V nadaljevanju te naloge se bomo ukvarjali le z robom lista, ki je na sliki prikazan z debelo črto. S krožcem ∘ je označen tisti konec lista, ki smo ga na začetku prepogibanja držali v levi roki.

---

nastanejo pri tem, bomo popisali z nizom znakov, kjer bo `D` pomenil, da je papir upognjen v desno, `L` pa bo pomenil, da je upognjen v levo. En tak popis, ko papir prepognemo štirikrat, je videti npr. takole: `LLDDLDDLLLDLLDD`. Vseh možnih nizov, ki jih lahko takole dobimo, je več, odvisno od tega, v katero smer smo upognili list v posameznem koraku. Naslednja slika kaže primer, kaj se zgodi, če list najprej prepognemo dol, nato pa gor; dobimo niz `DLL`:



Čez nekaj časa je nekdo poskusil z obratnim postopkom. Najprej si je izmislil nek tak niz, ki opisuje prepogibanje, nato pa je poskušal papir zložiti tako, da bo ustrezal temu popisu. Po nekaj poskusih se je zelo prestrašil, ko je ugotovil, da ni mogoče prepogniti papirja tako, da bi ustrezal poljubnemu nizu, ki bi si ga izmislili. Ker je čisto v šoku, te prosi za pomoč.

(*a*) **Opiši postopek**, ki za dani niz ugotovi, s kakšnim zaporedjem prepogibov (na vsakem koraku lahko izvedemo prepogib gor ali pa prepogib dol) ga lahko dobimo, ali pa ugotovil, da do tega niza ne moremo priti z nobenim zaporedjem prepogibov.

(*b*) **Opiši postopek**, ki za dano zaporedje prepogibov (gor in/ali dol) izpiše niz znakov `L` in `D`, ki opisuje končno obliko lista po vseh teh prepogibih.

(*c*) **Opiši postopek**, ki za dano zaporedje prepogibov (gor in/ali dol) izriše obliko lista papirja po teh prepogibih z z ASCII znaki „`/`“, „`\`“, „`-`“ in „`|`“. Točko, v kateri smo list papirja na začetku držali z levo roko, prikaži z znakom „`o`“.

Po n prepogibanjih ima papir 2^n plasti, zato naj ima slika 2^n vrstic, široka pa naj bo $2^n + 10$ stolpcev.

Primeri:

- Papir pred prepogibanjem:

```
o----------
```

- Po enem prepogibu gor:

```
----------\
o----------/
```

- Prepogib gor, ki mu sledi prepogib dol:

```
------------\
o-----------\|
/-----------/|
\------------/
```

- Prepogib gor, ki mu sledita dva prepogiba dol:

```
-----------------\
o---------------\|
/--------------\||
\-------------\|||
/-------------/|||
|/-------------/||
|\--------------/|
\----------------/
```

18. Tkanina

V času razcveta informacijskih tehnologij so se v podjetju Tkanina d. o. o. odločili, da bodo izdali posebno serijo tkanin, ki bodo imele motive sestavljene iz ničel in enic. Digitalni stroj za tisk tkanine dobi na vhodu kolut nepotiskanega blaga in motiv za tisk. Blago na kolutu ima podano širino (w) in dolžino (h) v centimetrih. Vzorec, ki bi ga radi natisnili, je sestavljen iz pravokotnega osnovnega vzorca, ki se potem po višini in širini ponovi tolikokrat, kolikor je potrebno glede na velikost blaga. Osnovni vzorec je širok t stolpcev in visok 2^t vrstic, na njem pa so napisana števila od $2^t - 1$ do 0 v dvojiškem zapisu (vsaka številka pokriva območje 1×1 cm).

Zgornji levi kot osnovnega vzorca ni nujno poravnan z zgornjim levim kotom našega pravokotnega kosa blaga, pač pa sta podani števili $\triangle x$ in $\triangle y$, ki povesta, da je zgornji levi kot osnovnega vzorca zamaknjen $\triangle x$ stolpcev levo in $\triangle x$ vrstic navzgor glede na zgornji levi kot našega kosa blaga.

Primer za $t = 3$ kaže slika na str. 42.

(a) **Napiši podprogram**, ki za dana cela števila t, w, h, $\triangle x$ in $\triangle y$ ter nariše vzorec, ki nastane na blagu pri tiskanju pod zgoraj omenjenimi pravili.

(b) **Opiši postopek**, ki za dane t, w, h, $\triangle x$, $\triangle y$ prešteje, koliko enic bo na tako potiskanem kosu blaga. Postopek naj bo učinkovit tudi za velike w in h.

19. Prelom besedila

Neko besedilo (ki je podano v eni dolgi vrstici) bi radi razbili v več vrstic in to tako, da bo v novem besedilu najdaljša vrstica čim krajša. Pri tem pa moramo upoštevati omejitev, da sme biti novo besedilo dolgo največ k vrstic. Besedilo lahko prelomimo le pri presledku (presedek se pri tem izgubi) — posamezne besede torej ne smemo razdeliti med dve vrstici. V dolžino vrstice štejejo vsi znaki v njej, tudi presledki (ne pa znaki za konec vrstice).

Vhodna datoteka: v njej sta dve vrstici. Prva vsebuje besedilo (dolgo največ 200 znakov), ki bi ga radi razbili. Druga vrstica vsebuje število k (celo število, za katero velja $1 \leq k \leq 50$), torej največje število vrstic, ki jih smemo uporabiti pri razbijanju besedila.

```
                                    00100100
111                                 11011011
110                                 10010010
101                                 01001001
100                                 00000000
011                                 11111111
010                                 10110110
001                                 01101101
000                                 00100100
                                    11011011
```

Primer za nalogo Tkanina. Leva slika kaže osnovni vzorec pri $t = 3$, desna slika pa rezultat, ki ga dobimo, če ta vzorec uporabimo na tkanini velikosti $w = 8$, $h = 10$ z zamikom $\triangle x = 1$, $\triangle y = 3$.

Besedilo vsebuje le črke angleške abecede, presledke in osnovna ločila (pika, klicaj, vprašaj). V besedilu ne bo podvojenih presledkov. Presledkov ne bo niti na začetku niti na koncu besedila, vselej bodo vmes.

Izhodna datoteka: vanjo izpiši dožino najdaljše vrstice v najboljšem možnem razbitju besedila.

Primer treh vhodnih datotek: Pripadajoče izhodne datoteke:

```
Danes je lep soncen dan.                    12
2


EnaSamaInEdinaBeseda                        20
37


Ena zeloooooooooooooooooo dolga beseda      22
3
```

Opomba: najboljša možna razbitja pri teh treh testnih primerih so takšna:

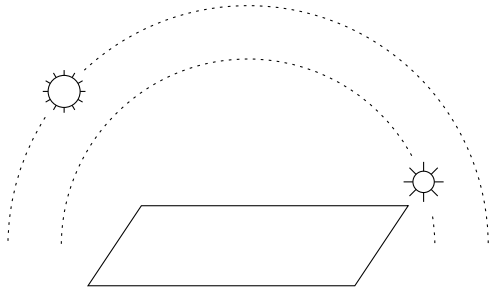
```
Danes je lep
soncen dan.


EnaSamaInEdinaBeseda


Ena
zeloooooooooooooooooo
dolga beseda
```

V splošnem se lahko zgodi, da obstaja za neko besedilo več enako dobrih optimalnih razbitij (dolžina najdaljše vrstice pa je pri vseh enaka).

20. Mnogosončje

Mislimo si ploščat planet, okrog katerega kroži n sonc s podanimi obhodnimi časi; i-to sonce obkroži naš planet v $2p_i$ urah. Polovico tega časa osvetljuje zgornjo stran našega planeta, polovico časa pa spodnjo. Na začetku opazovanja za vsako sonce

vemo, pred koliko časa je nazadnje vzšlo (gledano z zgornje strani planeta; če je na primer i-to sonce nazadnje vzšlo pred a_i urami in je $a_i \geq p_i$, to pomeni, ta trenutno osvetljuje spodnjo stran planeta, ne zgornje). Opazujemo naslednjih t ur. **Opiši postopek**, ki ugotovi, koliko ur bo zgornja stran planeta neosvetljena, koliko ur bo osvetljena z enim soncem, koliko z dvema itd.

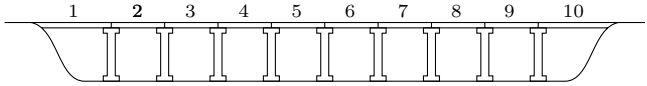


21. Dvižni most

Dan je dvižni most, ki je a sekund dvignjen, nato b sekund spuščen, nato spet a sekund dvignjen, b sekund spuščen in tako naprej v nedogled. Ob časih $t_1, \ldots, t_n$ pripeljejo pred most avtomobili; če avtomobil prispe v času, ko je most spuščen, zapelje čez, če pa je most takrat dvignjen, bo avtomobil počakal do trenutka, ko se most spusti. (Most se dvigne ali spusti v trenutku; in če avtomobil pripelje do njega ravno v istem trenutku, mu ni treba čakati.) Čase merimo v sekundah po polnoči. Vprašanje je, ali naj bo ob času 0 most dvignjen ali spuščen in koliko sekund po tem času naj se mu stanje prvič spremeni (iz dvignjenega v spuščeno ali obratno). **Opiši postopek**, ki to dvoje izbere tako, da bo skupno število sekund, ki jih bodo moralo naših n avtomobilov prečakati pred mostom, čim manjše.

22. Viadukt

Podjetje RBŠ je ravnokar zgradilo največji avtocestni viadukt na svetu. Sestavljajo ga podporni stebri in cestni odseki, ki so nameščeni na teh stebrih. Na vsakem podpornem stebru se stikata po dva cestna odseka, oz. povedano drugače, oba konca odseka ležita na enem podpornem stebru, le prvi in zadnji sta na zunanjih koncih podprta s krajnima opornikoma. (Glej sliko.) Cestni odseki so oštevilčeni s števili od 1 do n.



Čeprav je ta viadukt najmodernejši med vsemi, kar s(m)o jih ljudje zgradili do sedaj, ima seveda svoje omejitve. Še preden ga bodo odprli za promet, morajo izračunati, koliko avtomobilov je lahko na viaduktu naenkrat (zaradi varnosti). Zato so inženirji izmerili, koliko avtomobilov prenesejo posamezni odseki. Ker je viadukt zelo velik, je to počela velika ekipa inženirjev, ki je bila slabo organizirana. Ker se jim je zelo mudilo (tako ali tako že zamujajo z otvoritvijo), so svoje delo opravili zelo

površno. Zaradi naglice so mnogi inženirji namesto na posameznih odsekih izmerili največjo dovoljeno obremenitev kar na več zaporednih odsekih skupaj, npr. nekdo je izmeril, da se lahko na odsekih od # 13 do # 49 (to vključuje oba navedena in vse vmesne) nahaja naenkrat največ 140 avtomobilov. Ker pa so bili neorganizirani, se mnoge meritve prekrivajo. Nekdo drug je npr. izmeril, da odseki od # 23 do # 67 prenesejo največ 230 avtomobilov.

Nastala je velika zmeda, ker sedaj nihče pri RBŠ ne zna izračunati, koliko avtomobilov sme biti na viaduktu naenkrat, zato potrebujejo tvojo pomoč. **Opiši postopek**, ki prebere podatke o meritvah in izračuna največje možno število avtomobilov, ki bi lahko hkrati stali na viaduktu, ne da bi bila prekoračena kakšna omejitev.

Bolj matematično lahko nalogo opišemo takole: naj bo a_i število avtomobilov na i-tem odseku; danih je m omejitev, j-ta med njimi je opisana s števili u_j, v_j in w_j in nam pove, da mora biti vsota $a_{u_j} + a_{u_j+1} + \ldots + a_{v_j-1} + a_{v_j}$ manjša ali enaka w_j. Mi pa iščemo tak nabor nenegativnih celih števil $a_1, \ldots, a_n$, ki bo imel največjo možno vsoto $a_1 + \ldots + a_n$, seveda ob pogoju, da mora ustrezati vsem m omejitvam.

23. Meje med stavki

Eno od opravil, s katerimi se srečujemo pri računalniški obdelavi besedil, je tudi razdeljevanje besedila na posamezne povedi. Ta problem sploh ni tako preprost, kot se zdi na prvi pogled. Res je sicer, da nam konec povedi ponavadi označuje eno od končnih ločil (na primer pika ali klicaj), vendar pa se stvari zapletejo zaradi prisotnosti narekovajev, oklepajev, več končnih ločil skupaj in zaradi primerov, ko pika ne označuje konca povedi, pač pa okrajšavo.

Pri tej nalogi si bomo ogledali malo poenostavljeno različico pravil, ki so jih za razdeljevanje besedila na povedi objavili v standardu Unicode. Za potrebe naše naloge se dogovorimo, da bomo za končna ločila šteli le znake `? ! .` (vprašaj, klicaj in piko); za oklepaje bomo šteli znake `( ) [ ] { }`, za narekovaje pa znaka `"` (dvojni narekovaj) in `'` (enojni narekovaj). Predpostavimo tudi, da se poleg omenjenih znakov v našem vhodnem besedilu pojavljajo le črke angleške abecede (velike in male), številke (od `0` do `9`), ločila `, : ; -`, presledki in znaki za konec vrstice.

Pravila za določanje mej med stavki so opisana s trojicami oblike „$L \times R$“ ali „$L \div R$“. Za nek položaj v besedilu (mesto med dvema zaporednima znakoma) pravimo, da se ujema s pravilom „$L \times R$“ ali „$L \div R$“, če se konec besedila pred opazovanim položajem ujema z vzorcem L, začetek besedila za opazovanim položajem pa se ujema z vzorcem R.

Pri tej nalogi bomo uporabljali naslednji seznam pravil:

$$
\begin{array}{rcl}
P & \times & D \\
V\ P & \times & V \\
P\ \langle O\ \textit{ali}\ N\rangle^\star\ S^\star & \times & \langle \textit{poljuben znak, ki ni } V,\ M\ \textit{ali}\ L\rangle^\star\ \mathrm{M} \\
L\ \langle O\ \textit{ali}\ N\rangle^\star\ S^\star & \times & L \\
L\ \langle O\ \textit{ali}\ N\rangle^\star & \times & \langle O\ \textit{ali}\ N\ \textit{ali}\ S\rangle \\
L\ \langle O\ \textit{ali}\ N\rangle^\star\ S^\star & \times & S \\
L\ \langle O\ \textit{ali}\ N\rangle^\star\ S^\star & \div & \langle \textit{poljuben znak ali pa konec besedila}\rangle
\end{array}
$$

Pri tem okrajšave pomenijo: L = končno ločilo; P = pika; D = številka; V = velika črka; M = mala črka; O = oklepaj; N = narekovaj; S = presledek. Tako na primer

drugo pravilo v gornjem seznamu pravi, da če stoji pika med dvema velikima črkama, potem za to piko ne smemo postaviti meje med povedima (ker najbrž pika tukaj ne označuje konca povedi, pač pa gre za kratico). Zvezdica * v vzorcu pomeni, da se sme na tistem mestu pojaviti nič, eden ali več takšnih znakov, kot jih opisuje izraz pred zvezdico (na primer: $S^\star$ pomeni nič ali več presledkov, $\langle O\ ali\ N\rangle^\star$ pomeni poljubno zaporedje nič ali več oklepajev in/ali narekovajev).

Postopek za odločanje o tem, ali na nekem položaju v besedilu nastopa meja med dvema povedima ali ne, pa je definiran takole. Za opazovani položaj poiščemo v seznamu pravil prvo tako pravilo, s katerim se naš položaj ujema. Če takega pravila sploh ni, na opazovani položaj ne postavimo meje med povedima. Če pa tako pravilo najdemo, postavimo na opazovani položaj mejo med povedima natanko v tistih primerih, ko najdeno pravilo vsebuje simbol $\div$, ne pa simbola $\times$.

Napiši program, ki prebere besedilo iz vhodne datoteke in določi v njem vse meje med povedmi v skladu z gornjimi pravili. Besedilo naj izpiše v izhodno datoteko in to tako, da na vsako mesto, kjer stoji meja med dvema povedma, vrine znak `|`. Besedilo v vhodni datoteki je dolgo največ 1 000 000 znakov in je zapisano kot ena sama dolga vrstica.

Opomba: v spodnjem primeru je besedilo razdeljeno čez dve vrstici, ker bi bilo sicer preširoko za ta list papirja. V vhodnih datotekah, s katerimi bo delal tvoj program, bo besedilo vedno v eni sami vrstici.

Primer vhodne datoteke:

```
G. Kovac pravi: "Kaj?  Ti si bil!"  ("On ze ne -- g. Kovac laze!" si
mislim.  "Mogoce je bil npr. kar sam... ali pa...")  In je sel.
```

Pripadajoča izhodna datoteka:

```
G. |Kovac pravi: "Kaj?  |Ti si bil!"  |("On ze ne -- g. |Kovac laze!" |si
mislim.  |"Mogoce je bil npr. kar sam... ali pa...")  |In je sel.|
```

24. Družinska drevesa

Podane imaš sorodstvene vezi, opisane z relacijo „je prednik od“. Ni nujno, da te sorodstvene vezi natančno opredeljujejo celotno družinsko drevo, je pa zagotovljeno, da so smiselne (torej ni možno, da bi nekdo bil hkrati prednik in potomec nekoga drugega). **Opiši postopek**, ki za dane posameznike ugotovi njihove zanesljive prednike ter njihove potencialne prednike. *Zanesljivi predniki* in *zanesljivi potomci* so tisti, za katere je to razvidno že na osnovi podanih sorodstvenih vezi. *Potencialni predniki* pa so tisti, ki hkrati niso niti zanesljivi predniki niti zanesljivi potomci.

25. Poker

Napiši program za iskanje zmagovalca pri igri pokra z dvema ali več igralci. Igra se z enim kompletom igralnih kart (2–10, J, Q, K, A v štirih barvah); vsak igralec dobi po 5 kart. V vsaki vrstici vhodne datoteke so podatki o igralčevih petih kartah v obliki parov $\langle barva, številka\rangle$, kjer so barve: `s` (srce), `k` (karo), `p` (pik), `x` (križ), številke pa so 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 (= fant), 12 (= dama), 13 (= kralj) in 14 (= as). Primer: oznaka `k12` predstavlja karovo damo. V izhodno datoteko zapiši zmagovalno kombinacijo kart.

Igre pri pokru so naslednje (razvrščene od najvišje do najnižje):

1. barvna lestvica (zaporedje v isti barvi, npr. 7–6–5–4–3 v srcu; zmaga višja lestvica);
2. štiri enake (npr. 8–8–8–8–A; zmaga najvišja četvorka);
3. tris + par (tri enake + en par, npr. 7–7–7–14–14; zmaga najvišji tris);
4. ena barva (vseh 5 kart iste barve; zmaga najvišja karta);
5. lestvica (zaporedje, npr. 11–10–9–8–7; zmaga višja lestvica);
6. dva para (npr. 11–11–7–7–10; zmaga najvišji prvi/drugi par ali najvišja karta);
7. en par (npr. 13–13–7–8–2; zmaga najvišji par ali najvišja karta);
8. najvišja karta (npr. 14–13–6–4–3; zmaga najvišja karta).

As se lahko upošteva kot najvišja karta (nad kraljem) ali kot 1 v lestvici (najnižja možna lestvica). V vhodnih podatkih bo as vedno naveden kot 14. V primeru, da imata dva igralca enako igro, je zmagovalec določen po pravilu najvišje karte, sicer je igra neodločena. Na primer:

igralec 1: `s8 x8 p8 k8 s14` igralec 2: `s9 x9 p9 k9 p3`	Oba igralca imata po štiri enake karte (igra 2); zmaga igralec 2, ker ima višji četverec.
igralec 1: `s8 x8 p9 k6 p3` igralec 2: `k8 p8 s9 s6 s4`	Oba igralca imata isti par (8) in isti dve dodatni karti, odloča tretja dodatna karta; zmaga igralec 2, ker je 4 več kot 3.

Primer vhodne datoteke:

```
s3 s4 s5 s6 k7
p2 p3 p4 p5 p8
k3 k4 k5 s11 p11
x3 x4 x5 x7 x12
```

Pripadajoča izhodna datoteka:

```
x3 x4 x5 x7 x12
```

Komentar: igralec 1 ima lestvico (igra 5), igralec 3 ima par fantov (igra 7), igralca 2 in 4 imata oba po eno barvo (igra 4), vendar zmaga igralec 4, ker ima višjo karto (damo) kot igralec 2 (ki ima le osmico).

26. Brisanje podnizov

Dana sta dva niza, w in x. Če se v w-ju pojavlja x kot strnjen podniz, smemo kakšno tako pojavitev x-a v w-ju pobrisati; tako iz w-ja nastane nek nov, krajši niz. To lahko ponavljamo, dokler se pač da. **Opiši postopek**, ki ugotovi, kako dolg je najkrajši niz, ki se ga da na ta način dobiti iz niza w.

Primer: iz $w =$ `baaabcbcabccb` je mogoče z brisanjem $x =$ `abc` dobiti `baaabcbccb`, nato `baabccb` in končno `bacb`.

27. Skiroji

Imamo n skirojev in n otrok. Za vsakega otroka vemo, kako rad ima posamezen skiro (recimo temu številu a_{ij} za otroka i in skiro j). Vsakemu želimo dodeliti skiro na tak

način, da bo najbolj nezadovoljen otrok čim bolj zadovoljen. Torej **opiši postopek**, ki poišče maksimum vrednosti $\min\{a_{i,\pi(i)} : 1 \leq i \leq n\}$ po vseh permutacijah π množice $\{1, \ldots, n\}$.

28. Zeleni val

Dana je pravokotna mreža cest z $m \times n$ križišči; v vsakem križišču je semafor. Ulice so konstantne dolžine in pot od enega križišča do poljubnega sosednjega vedno vzame enako časa (ki je podan). Vsi semaforji se periodično prižigajo in ugašajo, vsi imajo enako periodo: p sekund so prižgani, p sekund ugasnjeni in tako naprej. Vsak semafor pa ima svoj časovni zamik (naravno število od 0 do $2p-1$, ki nam pove, kje v svojem ciklu prižiganja in ugašanja je bil ta semafor v nekem izbranem trenutku, ko smo celoten sistem začeli opazovati). Začnemo v nekem podanem križišču A in želimo priti do podanega križišča B, ne da bi se enkrat samkrat ustavili na rdeči luči. Se to da? Se to da, če štartamo takoj? Kaj pa, če enkrat v prvih 10 sekundah? **Opiši postopek**, ki poišče odgovore na ta vprašanja.

REŠITVE NALOG ZA PRVO SKUPINO

1. Vodilni elementi

Hitro opazimo, da je „vodilnost“ nekega elementa odvisna le od elementov z večjim indeksom. Zato vodilne elemente najlažje preštejemo tako, da začnemo preiskovati polje na koncu, pri zadnjem elementu.

Zadnji element je vedno vodilni, ker je zagotovo večji od vseh elementov z večjim indeksom, saj takih elementov ni. Vrednost tega elementa shranimo v spremenljivko, ki vsebuje največji dosedaj najdeni maksimum v polju (najvecji) ter nadaljujemo s preiskovanjem proti začetku.

Na vsakem koraku pogledamo, ali je trenutni element večji od trenutnega maksimuma. Če to drži, je trenutni element tudi vodilni, saj je večji od svojega največjega naslednika, kar pomeni, da je večji od vseh svojih naslednikov. V tem primeru vredno elementa shranimo v spremenljivko najvecji in povečamo število vodilnih elementov. Če pa je trenutni element manjši od trenutnega maksimuma, pa med njegovimi nasledniki obstaja večji element, zato trenutni element ni vodilni.

```
#include <stdio.h>

int main()
{
  int n, i, najvecji, stVodilnih = 0, tabela[100];
  scanf("%d", &n);
  for (i = 0; i < n; i++) scanf("%d", &tabela[i]);
  for (i = n - 1; i >= 0; i--)
    if (i == n - 1 || tabela[i] > najvecji)
      najvecji = tabela[i], stVodilnih++;
  printf("%d\n", stVodilnih);
  return 0;
}
```

2. Abecedni podnizi

Vhodne podatke lahko beremo znak za znakom; poleg trenutnega znaka (spremenljivka znak) si zapomnimo še prejšnjega (spremenljivka prejZnak). Spremenljivka dolzina hrani dolžino trenutnega abecednega podniza. Če je trenutni znak ravno naslednik prejšnjega v abecedi, povečamo dolzina za 1, sicer pa jo postavimo na 1. Na vsakem koraku pogledamo, če je nova dolžina daljša od najdaljše doslej znane, in če je, si jo zapomnimo (v spremenljivki naj, ki jo na koncu tudi izpišemo). Vrednost dolzina = 0 uporabljamo v primerih, ko prejšnji znak ni bil črka.[2]

```
#include <stdio.h>

int main()
{
  int znak, prejZnak = -1, dolzina = 0, naj = 0;
  while ((znak = fgetc(stdin)) != EOF)
```

[2] To je načeloma koristno, da ne bi kot abecednega podniza prepoznali niza „`a“ (saj je kativec „`“ v ASCII ravno eno mesto pred črko „a“. Res pa je, da pri naši nalogi to ni zares potrebno, saj iz besedila naloge sledi, da v vhodnih podatkih ne kaže pričakovati drugih znakov razen črk in znakov za konec vrstice.

```
  {
    if (znak < 'a' || znak > 'z') dolzina = 0;
    else if (dolzina == 0 || znak != prejZnak + 1) dolzina = 1;
    else dolzina++;
    if (dolzina > naj) naj = dolzina;
    prejZnak = znak;
  }
  printf("%d\n", naj);
  return 0;
}
```

3. Skrivanje tipk

Recimo, da so imena krajev shranjena v tabeli imena. Preglejmo vsa imena in pri vsakem poglejmo, ali se začne ravno z nizom, ki smo ga mi dobili kot parameter s. To naredimo tako, da primerjamo istoležne znake obeh nizov, dokler ne pridemo do neujemanja ali pa do konca kakšnega od nizov. Če pridemo do konca niza s, ne da bi opazili neujemanje, in če se trenutno ime še nadaljuje (torej da ni čisto enako s, pač pa se le začne na s), vemo, da bomo morali izpisati njegovo naslednjo črko. Ker se lahko zgodi, da na isto črko naletimo pri več imenih, izpisali pa bi jo radi le enkrat, si bomo pomagali s tabelo mozna, v kateri za vsako črko hranimo podatek o tem, ali jo bo treba izpisati ali ne. Med pregledovanjem imen torej le postavljamo posamezne elemente te tabele na **true**, izpišemo pa jih šele na koncu.

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

#define StImen 100
extern const char *imena[StImen];

void NaslednjeCrke(char *s)
{
  bool mozna[26]; int i, j;
  /* Inicializirajmo tabelo mozna. */
  for (i = 0; i < 26; i++) mozna[i] = false;
  /* Preglejmo vsa imena. */
  for (i = 0; i < StImen; i++) {
    /* Poglejmo, do kod se ujemata imena[i] in niz s. */
    for (j = 0; s[j] == imena[i][j] && s[j]; j++) ;
    /* Če se imena[i] začne na niz s in se potem še nadaljuje... */
    if (! s[j] && imena[i][j])
      /* ... označimo naslednjo črko tega imena kot možno. */
      mozna[imena[i][j] - 'a'] = true; }
  /* Izpišimo možne črke. */
  for (i = 0; i < 26; i++) if (mozna[i]) putchar('a' + i);
  putchar('\n');
}
```

Še elegantnejša rešitev pa je, da imena mest zložimo v drevo (*trie*); na vsaki povezavi naj bo ena črka in za vsako mesto bomo imeli v drevesu vejo (zaporedje povezav z začetkom v korenu drevesa), na kateri črke tvorijo ravno ime tega mesta. Če se več imen mest ujema v prvih nekaj znakih, si ustrezne povezave delijo, drevo pa se razveji šele tam, kjer med imeni pride do prvih neujemanj. S takšnim drevesom se mora podprogram NaslednjeCrke le sprehoditi od korena navzdol po povezavah, ki ustrezajo

črkam iz niza `s`; ko pride do konca `s`, pa mora le pogledati, katere črke so prisotne na povezavah, ki izhajajo iz vozlišča, do katerega je prišel.

4. Cikel

Naj bo $a[k]$ številka plesalca, ki ga drži plesalec k. Naloga pravi, da imamo te vrednosti podane v tabeli; obenem pa zagotavlja tudi, da vsakega plesalca drži natanko en plesalec. Začnimo pri poljubnem plesalcu, recimo k_0; on drži plesalca k_1, ta drži plesalca k_2, slednji drži plesalca k_3 in tako naprej. Ker imamo le n različnih plesalcev, se začnejo v tem zaporedju plesalci prej ali slej ponavljati. Recimo, da je v tem zaporedju k_j prvi tak plesalec, ki se je pojavil v njem že drugič, recimo na indeksu i (torej $i < j$ in $k_j = k_i$). Če je $i > 0$, pomeni, da tega plesalca držita tako k_{i-1} kot k_{j-1}, kar je mogoče le, če sta tudi tadva en in isti plesalec. To pa bi bilo v protislovju s predpostavko, da je k_j prvi plesalec, ki se je v zaporedju pojavil drugič. Ostane torej le možnost, da je $i = 0$; z drugimi besedami, prvi plesalec, ki se v našem zaporedju pojavi drugič, je kar plesalec k_0, pri katerem smo začeli. Naš postopek je torej lahko takšen: začnemo pri poljubnem k_0 in sledimo povezavam iz tabele a, dokler ne pridemo spet do k_0 (iz pravkar opisanega razmisleka vemo, da se bo to prej ali slej gotovo res zgodilo). Ob tem še štejemo, koliko plesalcev smo pregledali. Če pridemo nazaj na k_0 šele po n korakih, vemo, da imamo vseh n plesalcev v enem samem velikem krogu; če pa pridemo do k_0 že prej, vemo, da je ta plesalec del nekega manjšega kroga, ki ne zajema vseh plesalcev, torej mora obstajati poleg njega še kakšen drug krog.

Zapišimo naš postopek še s psevdokodo:

$k_0 := 1$; $k := k_0$; $i := 0$;
ponavljaj
  $k := a[k]$; $i := i + 1$;
dokler velja $k \neq k_0$;
če $i = n$, izpiši „`V KROGU`“,
sicer izpiši „`PO SVOJE`“;

5. Kvadrati s seštevanjem

Število k^2 lahko namesto z množenjem ($k^2 = k \cdot k$) računamo s seštevanjem: začnemo s številom 0 in mu k-krat prištejemo k.

$$k^2 = k \cdot k = \underbrace{k + k + \ldots + k}_{k\text{-krat}}.$$

Za izračun k^2 torej potrebujemo zanko s k iteracijami. Ker moramo izpisati kvadrate vseh števil od 1 do n, pa potrebujemo poleg tega še zunanjo zanko, ki gre s k od 1 do n.

```
#include <stdio.h>

void Kvadrati(int n)
{
  int i, k, kvadrat;
  for (k = 1; k <= n; k = k + 1) {
```

```
    for (kvadrat = 0, i = 1; i <= k; i = i + 1)
      kvadrat = kvadrat + k;
    printf("%d\n", kvadrat); }
}
```

Lahko pa tudi opazimo, da nam kvadratov ni treba računati vsakič znova. Velja namreč

$$(k+1)^2 = (k+1)(k+1) = k \cdot k + k \cdot 1 + 1 \cdot k + 1 \cdot 1 = k^2 + 2k + 1.$$

Če torej že imamo pri roki vrednost k^2, lahko iz nje dobimo $(k+1)^2$ preprosto tako, da ji prištejemo $k + k + 1$.

```
void Kvadrati2(int n)
{
  int k, kvadrat = 0;
  for (k = 0; k < n; k = k + 1) {
    /* Na tem mestu je kvadrat = k * k. */
    kvadrat = kvadrat + k + k + 1;
    /* Na tem mestu je kvadrat = (k + 1) * (k + 1). */
    printf("%d\n", kvadrat); }
}
```

Lepo pri tej rešitvi je, da porabi le $O(n)$ časa, prejšnja pa ga je $O(n^2)$.

Kaj pa, če bi hoteli za dani n poiskati samo kvadrat n-ja samega, ne pa tudi kvadratov od 1 do $(n-1)^2$? Seveda bi lahko uporabili npr. funkcijo Kvadrati2 in jo prilagodili, da bi izpisala le n^2; toda to bi še vedno vzelo $O(n)$ časa. Do hitrejše rešitve pridemo, če si predstavljamo n v dvojiškem zapisu:

$$n = b_k \cdot 2^k + b_{k-1} \cdot 2^{k-1} + \ldots + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0,$$

pri čemer so vsi $b_i \in \{0, 1\}$. Očitno velja

$$n^2 = b_k \cdot 2^k n + b_{k-1} \cdot 2^{k-1} n + \ldots + b_2 \cdot 2^2 n + b_1 \cdot 2n + b_0 \cdot n.$$

Do števil $2^i n$ ni težko priti s seštevanjem: $n + n = 2n$, nato $2n + 2n = 4n$ in tako naprej. Vprašanje je le, kako priti do posameznih bitov v dvojiškem zapisu n-ja, če nočemo uporabljati drugih operacij kot seštevanje. Ena možnost je, da računamo potence števila 2, torej 2^k za vse večje k, dokler ne pridemo do takega k, pri katerem je $2^{k+1} > n$. Takrat vemo, da je bit k ravno najvišji prižgani bit v dvojiškem zapisu n-ja; spremenljivko, v kateri računamo kvadrat n-ja, torej lahko zdaj povečamo za $2^k n$, bit k v številu n pa ugasnemo. To ponavljamo, dokler je v n še kaj prižganih bitov. Ker prvotno vrednost n-ja pravzaprav ves ta čas še potrebujemo (za računanje $2^k n$), bomo za to ugašanje bitov uporabljali pomožno spremenljivko m.

```
int Kvadrat3a(int n)
{
  int m = n, kvadrat = 0, p, pn;
  while (m > 0) {
    /* Invarianta: m je bil dobljen iz n z ugašanjem zgornjih nekaj bitov;
       kvadrat = pn * (n - m). */
    /* Poiščimo najvišji prižgani bit v m. */
    p = 1; pn = n;
    while (p + p <= m) p = p + p, pn = pn + pn;
```

```
    /* Zdaj je p potenca števila 2, ki pove največji prižgani bit v m; in velja pn = n * p.
       Povečajmo kvadrat in ugasnimo bit p v m. */
    kvadrat = kvadrat + pn; m = m − p; }
  return kvadrat;
}
```

Neugodno pri tej rešitvi je, da smo za ugašanje bitov v m uporabili odštevanje, naloga pa načeloma pravi, da ne smemo uporabiti drugih aritmetičnih operacij kot seštevanje. Pomagamo si lahko tako, da od m-ja v resnici ne odštevamo, pač pa si v neki drugi spremenljivki (recimo d) zapomnimo, kaj vse bi morali doslej že odšteti od njega, pa nismo. Vedno bo torej veljala zveza $m = n - d$; namesto da zmanjšujemo m, bomo povečevali d; pa tudi pogoje, v katerih nastopa m, lahko zdaj predelamo tako, da uporabljajo d in ne potrebujejo odštevanja.

```
int Kvadrat3b(int n)
{
  int d = 0, kvadrat = 0, p, pn;
  while (n > d) {
    /* Invarianta: d je bil dobljen iz n z ugašanjem spodnjih nekaj bitov;
       kvadrat = pn * d. */
    /* Poiščimo najvišji prižgani bit v n − d. */
    p = 1; pn = n;
    while (p + p + d <= n) p = p + p, pn = pn + pn;
    /* Zdaj je p potenca števila 2, ki pove največji prižgani bit v n − d; in velja pn = n * p.
       Povečajmo kvadrat in prižgimo bit p v d. */
    kvadrat = kvadrat + pn; d = d + p; }
  return kvadrat;
}
```

Zunanja zanka naredi toliko iteracij, kolikor je v n prižganih bitov; notranja zanka pa naredi vsakič največ $\lfloor \log_2 n \rfloor$ iteracij, saj je po toliko iteracijah $p = 2^{\lfloor \log_2 n \rfloor}$ in zato pogoj $2p \leq n$ gotovo ni več izpolnjen (za vsak x velja $x < \lfloor x \rfloor + 1$, zato je $n = 2^{\log_2 n} < 2^{\lfloor \log_2 n \rfloor + 1} = 2 \cdot 2^{\lfloor \log_2 n \rfloor} = 2p$). Časovna zahtevnost te rešitve je torej le $O((\log n)^2)$.

Gornja rešitev je potence števila 2 (in njihove n-kratnike) računala vsakič znova (za vsak prižgan bit v dvojiškem zapisu n-ja). Do še učinkovitejše rešitve pridemo, če jih izračunamo le enkrat in jih nato shranimo v tabeli.

```
  ps[0] = 1; pns[0] = n; s = 0;
  while (ps[s] + ps[s] <= n) {
    ps[s + 1] = ps[s] + ps[s];
    pns[s + 1] = pns[s] + pns[s];
    s = s + 1; }
```

Na koncu te zanke imamo $s = \lfloor \log_2 n \rfloor$, tabeli pa hranita vrednosti $ps[i] = 2^i$ in $pns[i] = 2^i \cdot n$. Podobno kot prej se moramo zdaj sprehajati od večjih potenc proti manjšim in ugašati bite v n-ju:

```
  m = n; kvadrat = 0;
  for (i = 0; i <= s; i = i + 1)
    if (ps[s − i] <= m) {
      m = m − ps[s − i];
      kvadrat = kvadrat + pns[s − i]; }
```

Ta zanka še ni dobra, saj vsebuje odštevanje. Potrebi po odštevanju od spremenljivke m se lahko izognemo z uvedbo spremenljivke $d = n - m$, enako kot prej pri Kvadrat3b. Bolj nadležen problem je zdaj ta, kako brez odštevanja pregledovati potence v padajočem vrstnem redu; v gornji zanki se ta problem odraža v tem, da v tabelah ps in pns dostopamo do indeksa $s-i$; druga možnost bi bila, da bi se v zanki i zmanjševal od s do 0 — v vsakem primeru torej potrebujemo odštevanje. Temu se lahko izognemo tako, da si v še eni tabeli pripravimo indekse od 0 do s v padajočem vrstnem redu:

```
a[0] = 0; r = 0;
for (r = 0; ps[r] <= s; r = r + 1)
   for (i = 0; i < ps[r]; i = i + 1)
      a[i + ps[r]] = a[i], a[i] = a[i] + ps[r];
```

Po prvi iteraciji zunanje zanke imamo $a = \langle 1, 0\rangle$, po drugi $a = \langle 3, 2, 1, 0\rangle$ in tako naprej. Po r iteracijah imamo $a[i] = 2^r - 1 - i$ za $i = 0, \ldots, 2^r - 1$. Ob koncu te zanke je $r = 1 + \lfloor \log_2 s \rfloor$. Časovna zahtevnost r-te iteracije je $O(2^r)$, časovna zahtevnost vseh iteracij od prve do r-te pa torej $O(2^1 + 2^2 + \ldots + 2^r) = O(2^r) = O(s) = O(\log n)$.

Zdaj lahko glavno zanko, ki bo pregledovala potence števila 2 v padajočem vrstnem redu, popravimo tako, da bo namesto števila $s - i$ za indeksiranje tabel ps in pns uporabljala vrednosti $a[i]$ za naraščajoče i. Tiste i, pri katerih je $a[i] > s$, pa preskočimo, saj tako velikih potenc števila 2 v tabeli ps ni, pripadajoči biti v n pa tudi gotovo niso prižgani.

```
d = 0; kvadrat = 0;
for (i = 0; i < ps[r]; i = i + 1) if (a[i] <= s)
   if (ps[a[i]] + d <= n) {
      d = d + ps[a[i]];
      kvadrat = kvadrat + pns[a[i]]; }
```

Ker je $ps[r]$ najmanjša potenca števila 2, ki je večja od s, je $ps[r] \le 2s$, torej je časovna zahtevnost celotnega postopka le $O(s)$, kar je $O(\log n)$. Asimptotično boljše rešitve pa že ni razumno pričakovati; od aritmetičnih operacij imamo na voljo le seštevanje, pri seštevanju pa je vsota kvečjemu dvakratnik večjega od obeh seštevancev; ker imamo na začetku le spremenljivko z vrednostjo n, je največja vrednost, ki jo lahko dobimo po k seštevanjih, enaka $n \cdot 2^k$; ker bi radi prišli do n^2, mora torej veljati $n \cdot 2^k \ge n^2$, torej $2^k \ge n$, torej $k \ge \log_2 n$. Rešitev, ki uporablja le seštevanje, torej potrebuje vsaj $O(\log n)$ operacij.

REŠITVE NALOG ZA DRUGO SKUPINO

1. Poštne številke

V tabeli variante imejmo za vsako številko (od 0 do 9) niz z vsemi možnimi številkami, s katerimi jo je mogoče zamenjati (na primer: iz 4 lahko nastane 1, 4 ali 7, zato je variante[4] == "147"). Iz danega števila k lahko izluščimo posamezne številke z deljenjem: ostanek po deljenju k z 10 je ravno najbolj desna številka, količnik pa je v tem primeru število, ki ga dobimo, če k-ju pobrišemo najbolj desno številko. Ta količnik lahko nato spet delimo z 10, da dobimo predzadnjo številko k-ja in tako naprej. Ko imamo vse štiri številke (d1, d2, d3 in d4), se v štirih gnezdenih zankah sprehodimo po vseh kombinacijah številk iz nizov variante[d1], ..., variante[d4] in vsako kombinacijo izpišimo.

```
#include <stdio.h>

void Variante(int k)
{
  static const char *variante[10] = { "0", "147", "2", "38", "147",
                                      "56", "56", "147", "38", "9" };
  int d1 = k / 1000, d2 = (k / 100) % 10, d3 = (k / 10) % 10, d4 = k % 10;
  const char *p1, *p2, *p3, *p4;
  for (p1 = variante[d1]; *p1; p1++)
  for (p2 = variante[d2]; *p2; p2++)
  for (p3 = variante[d3]; *p3; p3++)
  for (p4 = variante[d4]; *p4; p4++)
    printf("%c%c%c%c ", *p1, *p2, *p3, *p4);
}
```

Namesto z gnezdenimi zankami bi lahko nalogo rešili tudi z rekurzijo, kar bi sicer utegnilo biti malo počasneje, vendar pa bi bilo takšno rešitev lažje pripraviti do tega, da bi delovala tudi v primerih, ko k nima natanko štirih številk.

Bolj za šalo kot zares še enovrstična rešitev v pythonu (natisnjena čez dve vrstici, ker je stran sicer preozka zanjo):

```
def Variante(k): print " ".join(reduce(lambda X, Y: [y + x for x in X for y in
                        "0 147 2 38 147 56 56 147 38 9".split()[int(Y)]], str(k)[::-1], [""]))
```

2. Reka presledkov

Naloga pravi, da je vsaka vrstica vhodnega besedila dolga največ 100 znakov. Imejmo tabelo reka, v kateri za vsak položaj v vrstici piše, kako dolga reka se konča pri tistem znaku. Če je trenutni vrstici i-ti znak ni presledek ali pa je vrstica krajša od i znakov, bo reka[i] enaka 0.

Ko preberemo novo vrstico (v tabelo s), moramo popraviti tudi tabelo reka. Če s[i] ni presledek, bomo morali postaviti reka[i] na 0. Drugače pa lahko ta presledek nadaljuje kakšno reko iz prejšnje vrstice, če je tam kakšna reka, ki se konča na indeksih $i-1$, i ali $i+1$. Med dolžinami teh treh rek (ki so še v tabeli reka) vzemimo največjo, ji prištejmo 1 in tako dobimo novo vrednost reka[i]. Če je nova dolžina daljša od najdaljše doslej znane reke (spremenljivka najReka), si jo zapomnimo in na koncu najdaljšo dobljeno dolžino izpišemo.

Tabela reka ima 101 element, kar je en več, kot je lahko znakov v najdaljši možni vrstici; to je koristno, da nam ni treba pred dostopom do celice reka[i + 1] posebej preverjati, če nismo mogoče v stotem znaku trenutne vrstice.

Paziti moramo na naslednjo podrobnost: ker obdelujemo vrstico od leve proti desni, se v času, ko pridemo do i-tega znaka, v celici reka[i − 1] že nahaja podatek za trenutno vrstico, mi pa pri izačunu nove vrednosti reka[i] potrebujemo staro vrednost reka[i − 1] (tisto iz prejšnje vrstice). Zato si tisto staro vrednost zapomnimo v spremenljivki prejReka, kamor smo jo vpisali ob koncu prejšnje iteracije naše notranje zanke.

Vrstice vhodnega besedila so lahko različno dolge. Če je neka vrstica dolga le n znakov, moramo v tabeli reka na indeksih od n naprej postaviti ničle, sicer bi utegnili pri kakšni kasnejši vrstici (če bo spet daljša od trenutne) zmotno misliti, da se tam še lahko nadaljujejo kakšne zgodnejše reke (čeprav se ne morejo, ker je bila vmes neka krajša vrstica). Da ne bomo vsakič brisali tabele vse do konca, si v spremenljivki prejZadnja zapomnimo indeks zadnjega neničelnega elementa v tabeli; ničle moramo tako zapisati le do tega indeksa (če je trenutna vrstica krajša), saj so za njim v tabeli že od prej same ničle.

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

int main()
{
  int reka[MaxDolz + 1], najReka = 0, prejReka, prejReka2, zadnja, prejZadnja = 0, i;
  char s[MaxDolz + 2];
  for (i = 0; i <= MaxDolz; i++) reka[i] = 0;
  while (! feof(stdin)) {
    fgets(s, MaxDolz + 2, stdin);
    prejReka = 0; zadnja = 0;
    for (i = 0; i < MaxDolz && s[i] != '\n' && s[i]; i++, prejReka = prejReka2) {
      prejReka2 = reka[i];
      if (s[i] != ' ') { reka[i] = 0; continue; }
      if (prejReka > reka[i]) reka[i] = prejReka;
      if (reka[i + 1] > reka[i]) reka[i] = reka[i + 1];
      reka[i] += 1; zadnja = i;
      if (reka[i] > najReka) najReka = reka[i]; }
    while (i <= prejZadnja) reka[i++] = 0;
    prejZadnja = zadnja; }
  printf("%d\n", najReka); return 0;
}
```

3. Delitev kamenja

Kamne dajemo na dva kupa, pri čemer naslednjega vedno damo na trenutno lažji kup. Ko kamnov zmanjka, poiščemo na težjem kupu zadnji kamen, ki smo ga tja dodali, ga razbijemo na dva ustrezna kosa in en kos prestavimo v drugi kup.

Pri tej rešitvi mogoče na prvi pogled ni čisto očitno, da bomo na ta način res vedno lahko dobili dva enako težka kupa. Ali se lahko zgodi, da se teža kupov razlikuje za toliko, da ju z razbijanjem tistega zadnjega kamna s težjega kupa ne bomo mogli izenačiti? Prepričajmo se, da se to ne more zgoditi. Z indukcijo po številu kamnov bomo dokazali, da je razlika v teži kupov kvečjemu tolikšna, kolikor tehta nazadnje dodani kamen na težjem od obeh kupov.

Pri enem samem kamnu je očitno, da trditev drži: lažji kup je prazen, na težjem kupu je en kamen in razlika v teži kupov je ravno enaka teži tega kamna (ki je hkrati edini in zadnji kamen na težjem kupu). Recimo zdaj, da smo trditev dokazali za $k-1$ kamnov. Vzemimo k-ti kamen; njegovo maso označimo z m_k, skupno maso lažjega kupa z a, skupno maso težjega kupa z b, masa nazadnje dodanega kamna na težjem kupu pa naj bo x. Po induktivni predpostavki torej velja $0 \le b-a \le x$. Naš postopek pravi, da damo posamezni kamen vedno na lažji kup, v našem primeru torej na kup a; temu se masa poveča na $a+m_k$. Ločimo dve možnosti: (1) mogoče je z dodatkom kamna m_k ta kup postal težji od drugega, torej je $a+m_k \ge b$. V tem primeru je razlika v teži kupov zdaj $a+m_k-b = m_k-(b-a)$, kar je $\le m_k$, saj je $b-a \ge 0$. Torej je razlika v teži kupov kvečjemu tolikšna kot teža zadnjega dodanega kamna na težjem kupu (to je zdaj namreč kamen m_k); prav to smo tudi hoteli dokazati. (2) Mogoče pa tudi po dodatku kamna m_k na kup a le-ta še ostane lažji od drugega kupa. Kupa se zdaj razlikujeta za $b-(a+m_k) = b-a-m_k \le b-a \le x$, torej še vedno za manj, kot je teža zadnjega kamna na težjem kupu (namreč x). Tako torej vidimo, da v obeh primerih opisana lastnost velja tudi po k kamnih. Zato bomo lahko na koncu, ne glede na to, koliko kamnov smo imeli, s primerno razdelitvijo zadnjega kamna na težjem kupu vsekakor vedno dosegli, da bosta imela oba kupa enako težo.

Nalogo lahko rešimo tudi še kako drugače, na primer takole. Najprej seštejmo maso vseh kamnov; recimo, da je njihova skupna masa enaka M. Nato odlagajmo kamne v poljubnem vrstnem redu na en kup, dokler njihova skupna teža ne postane večja ali enaka $M/2$. Takrat od zadnjega dodanega kamna odrežemo toliko, za kolikor teža kupa presega $M/2$, in damo to na drugi kup. Nato dodamo na drugi kup še vse preostale kamne.

4. Parktronic

Naš program izvaja neskončno zanko; v vsaki iteraciji pošlje signal in nekaj časa čaka na odboj; če ga dobi, izračuna oddaljenost ovire in po potrebi prilagodi frekvenco zvočnika.

Ko pošljemo signal (s funkcijo Ping), si v spremenljivki t1 zapomnimo čas, ko smo to storili. Nato v zanki kličimo Poslusaj in merimo čas; ustavimo se, ko zaslišimo odboj našega pravkar poslanega signala ali pa ko mine MaxCasOdboja časa (to je čas, v katerem bi moral priti do nas odboj od 1 m oddaljenega predmeta). Če smo odboj prejeli, lahko iz razlike v času prejema (t2) in oddaje signala (t1) izračunamo oddaljenost ovire, od katere se je naš signal odbil (spremenljivka d). Iz te oddaljenosti določimo po navodilih iz besedila naloge frekvenco, s katero mora piskati zvočnik (novaFrek). Če je ta frekvenca drugačna od tiste, s katero smo piskali doslej, pokličemo funkcijo Zvočnik z novo frekvenco. (Če odboja nismo prejeli, lahko sklepamo, da je ovira oddaljena vsaj 1 m in lahko piskanje ugasnemo.)

Osembitno vrednost, ki jo pošljemo v posameznem signalu (signal), v vsaki iteraciji glavne zanke povečamo za 1, tako da, če zaslišimo odboj od kakšnega prejšnjega signala, na katerega smo se medtem že naveličali čakati, bo imel ta odboj zelo verjetno različno številko od nazadnje poslanega signala, tako da bomo vedeli, da to ni odboj, ki ga čakamo. Po 256 iteracijah glavne zanke se začnejo številke ponavljati, zato si je vsekakor mogoče zamisliti scenarij, ko bi ta rešitev delovala napačno (zaslišala odboj

nekega zgodnejšega signala, ki pa bi imel enako številko kot nazadnje poslani, in bi zaradi tega čisto narobe ocenila oddaljenost ovire); vendar pa bi bil tak scenarij že zelo nerealističen.

```
int main()
{
  const double HitrostZvoka = 340.0; /* v metrih na sekundo */
  const double MaxCasOdboja = 2e6 / HitrostZvoka; /* v mikrosekundah */
  int signal = rand() % 256, odboj, d, frek = 0, novaFrek; long t1, t2;
  for ( ; ; )
  {
    signal = (signal + 1) % 256;
    t1 = Cas(); Ping((unsigned char) signal);
    do { odboj = Poslusaj(); t2 = Cas(); }
    while (odboj != signal && t2 - t1 <= MaxCasOdboja);
    if (odboj != signal) novaFrek = 0;
    else {
      d = HitrostZvoka * (t2 - t1) / 2.0;
      novaFrek = d > 1 ? 0 : d >= 0.5 ? 400 : d >= 0.25 ? 1000 : 2000; }
    if (frek != novaFrek) { frek = novaFrek; Zvocnik(frek); }
  }
}
```

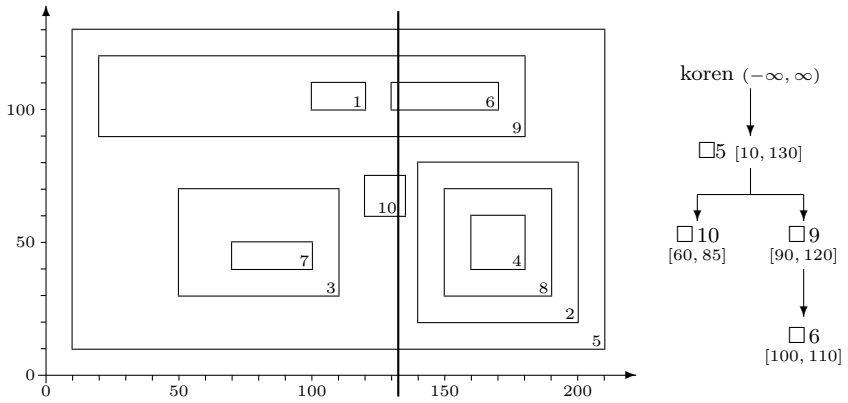
5. Pravokotniki

Za vsak pravokotnik poglejmo, kateri ga vsebujejo; to so njegovi nadrejeni v hierarhiji. Neposredno nadrejen mu je tisti, ki je med vsemi nadrejenimi najmanjši. Preglejmo vse pravokotnike in za vsakega v neko tabelo zapišimo, kateri mu je neposredno nadrejen. Potem pojdimo še enkrat čez vse pravokotnike, pri vsakem s pomočjo tabele ugotovimo, kateri so mu neposredno podrejeni, in takšne neposredno podrejene izpišimo. Zapišimo ta postopek še s psevdokodo:

za vsak $i := 1, 2, \ldots, n$:
  $k := 0$;
  za vsak $j := 1, 2, \ldots, n$:
    če $j \neq i$ in $x_{j1} \leq x_{i1}$ in $x_{i2} \leq x_{j2}$ in $y_{j1} \leq y_{i1}$ in $y_{i2} \leq y_{j2}$:
      če $k = 0$ ali $(x_{j2} - x_{j1})(y_{j2} - y_{j1}) < (x_{k2} - x_{k1})(y_{k2} - y_{k1})$:
        $k := j$;
  $nad[i] := k$;
za vsak $i := 1, 2, \ldots, n$:
  izpiši „Neposredni podrejeni pravokotnika i so:“
  za vsak $j := 1, 2, \ldots, n$:
    če $nad[j] = i$, potem izpiši j;
  izpiši konec vrstice;

Drugemu paru gnezdenih zank (ki skrbi le za izpis rezultatov) se lahko brez veliko truda izognemo, na primer tako, da namesto tabele nadrejenih (nad v zgornji psevdokodi) pripravljamo tabelo seznamov neposredno podrejenih; ko izvemo, da je k neposredno nadrejen pravokotniku i, dodamo i v seznam k-jevih neposrednih podrejenih. Prvi dve gnezdeni zanki pa nam tudi po tej spremembi ostaneta in je časovna zahtevnost celotnega postopka zaradi njiju $O(n^2)$.

Do učinkovitejše rešitve lahko pridemo s preletom ravnine (*plane sweep*). Predstavljajmo si, da v ravnino položimo navpično premico in pogledamo njen presek z našimi pravokotniki. Ta presek je množica intervalov, kot kaže primer na naslednji sliki (za premico $x = 133$); lahko si jih predstavljamo zložene v drevo, pri čemer vsako vozlišče vsebuje številko pravokotnika in interval na y-osi, ki ga ta pravokotnik pokriva (primer je na spodnji sliki desno).



Če našo navpično premico počasi premikamo od leve proti desni, v drevo intervalov včasih pride nov pravokotnik (ko naša premica naleti na njegov levi rob), včasih pa eden od pravokotnikov pade iz drevesa (ko naša premica naleti na njegov desni rob). Intervali v drevesu so v čisto enakem hierarhičnem odnosu kot pravokotniki, iz katerih so ti intervali nastali, le da v drevesu pač niso vedno prisotni vsi pravokotniki, pač pa le tisti, ki jih trenutno seka naša navpična premica. Tako bomo postopoma (ko se bo naša premica premikala od leve proti desni) odkrili vse hierarhične odnose med pravokotniki, po katerih sprašuje naša naloga. Postopek je torej nekako takšen:

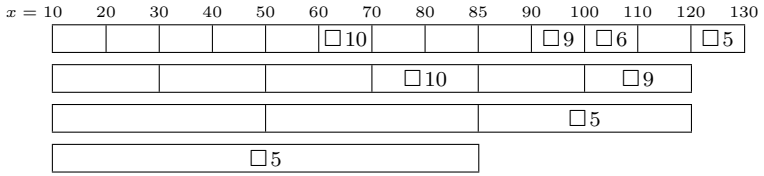
1 Naj bo T drevo intervalov, ki je na začetku prazno (oz. vsebuje le interval $(-\infty, \infty)$ kot koren drevesa).
2 Pregledujmo ravnino z navpično premico od leve proti desni.
3 Ko naletimo na levi rob nekega pravokotnika P:
4 Naj bo $[y_1, y_2]$ interval, ki ga P pokriva na y-osi.
5 Poiščimo v T najgloblji interval, ki vsebuje $[y_1, y_2]$.
Naj bo P' pravokotnik, iz katerega je nastal ta interval.
Potem vemo, da je P neposredno podrejen pravokotniku P'. $(\star)$
6 Dodajmo v T novo vozlišče za pravokotnik P z intervalom $[y_1, y_2]$, ki postane otrok vozlišča za pravokotnik P'.
7 Ko naletimo na desni rob nekega pravokotnika P:
8 Poiščimo v drevesu T njegov pripadajoči interval in ga pobrišimo (vemo tudi, da zagotovo nima otrok).

Opomba: če se na isti y-koordinati začne in konča več pravokotnikov, najprej obdelajmo tiste, ki se tam končajo (od ožjih proti širšim), nato pa tiste, ki se tu začnejo (od širših proti ožjim).

Rezultate, po katerih sprašuje naloga, dobimo, če si zapomnimo vse primere podrejenosti, ki smo jih opazili v vrstici ($\star$).

Oglejmo si še, kako lahko učinkovito implementiramo drevo T. Vse y-koordinate, ki se kje pojavljajo kot koordinate spodnjih ali zgornjih robov naših pravokotnikov, uredimo naraščajoče in intervale med njimi oštevilčimo: $A_0, \ldots, A_{k-1}$; pri tem gre lahko k tja do $2n-1$, če imamo v vhodnem seznamu n pravokotnikov. Vsak interval, ki ga potem dejansko pokriva kakšen od naših pravokotnikov, je potem unija enega ali več zaporednih A_i-jev. Drevo T lahko predstavimo kot $1 + \lfloor \log_2 k \rfloor$ tabel, pri čemer je prva dolga k elementov, druga $\lfloor k/2 \rfloor$, tretja $\lfloor k/4 \rfloor$ in tako naprej; v drugi tabeli predstavlja vsak element po dva zaporedna intervala, v tretji po štiri in tako naprej.

Za ilustracijo si oglejmo množico pravokotnikov iz gornjega primera. Njihovi spodnji in zgornji robovi imajo 14 različnih y-koordinat, tako da imamo $k = 13$ osnovnih intervalov. Zato potrebujemo za predstavitev našega drevesa 4 tabele: eno s 13 elementi, eno s 6 elementi, eno s tremi in eno z enim elementom. Drevo iz gornjega primera (pri premici $x = 123$) bi bilo videti takole:



Ko hočemo dodati v T nek interval, ki je v bistvu unija $A_i \cup \ldots \cup A_j$ (za $j \geq i$), vpišemo podatke o tem pravokotniku (recimo P) le v najmanjše potrebno število elementov naših tabel (v vsaki tabeli največ v dva elementa). To nam vzame $O(\log k)$ časa. Še preden to naredimo, pogledamo za poljubnega od teh intervalov — recimo za A_i — ali pripada kakšnemu pravokotniku P', ki je trenutno že v T; to nam vzame še $O(\log k)$ časa in nam bo povedalo, kdo bo oče novega pravokotnika P v hierarhiji. Brisanje intervala iz drevesa pa lahko izvedemo tako, da v iste elemente tabel, v katere smo ga vpisali pri dodajanju, zdaj napišemo, da so prazni; nato pa pogledamo njegovega očeta v hierarhiji in gremo čez vse elemente tabel, v katere smo ob dodajanju vpisali tega očeta; če je kateri od elementov zdaj prazen, vpišemo vanj oznako tega očeta. To pride prav v primerih, ko je sin povozil kakšne od očetovih vpisov; na primer, če imamo interval 4..7 in njegovega očeta 3..8; oče se je vpisal v celice 3..3, 4..7 in 8..8, sin pa je nato njegov vpis v srednji od njih povozil in je treba ob brisanju sina ta očetov vpis obnoviti.

Tako torej vidimo, da imamo ob preletu ravnine z vsakim levim ali desnim robom kakšnega pravokotnika $O(\log n)$ dela, tako da je skupna časovna zahtevnost našega postopka $O(n \log n)$ (v tem času lahko tudi uredimo x- in y-koordinate, preden začnemo s preletom ravnine). Asimptotično hitrejšega postopka pa pri tem problemu že ni več razumno pričakovati; če bi znali našo nalogo rešiti v manj kot $O(n \log n)$ časa, bi lahko tudi tabelo n realnih števil uredili naraščajoče v manj kot $O(n \log n)$ časa, kar pa je načeloma nemogoče.[3]

[3]Vsaj če predpostavimo, da z elementi tabele ne počnemo drugega kot to, da jih primerjamo med sabo in jih premikamo po tabeli. Prevedba problema urejanja na našo nalogo bi šla takole:

REŠITVE NALOG ZA TRETJO SKUPINO

1. Kup knjig

Številke knjig je koristno hraniti v dvojno povezanem seznamu (*doubly linked list*) — z drugimi besedami, za vsak zvezek hranimo podatek o tem, kateri je tik nad njim in kateri je tik pod njim. Spodnji program ima v ta namen tabeli nad in pod. Če takega zvezka sploh ni (npr. zvezka pod tistim, ki je na dnu kupa), hranimo v ustrezni celici tabele vrednost 0. Zapomnimo si tudi, kateri zvezek je na dnu kupa (spodnji) in kateri na vrhu (zgornji).

Ko povlečemo iz kupa zvezek i, postaneta zvezka nad[i] in pod[i] po novem neposredna soseda, torej mora pod[nad[i]] pokazati na pod[i], podobno pa mora nad[pod[i]] pokazati na nad[i]. Če spodnjega zvezka sploh ni, ker je ležal i na dnu kupa, postane zvezek, ki je bil doslej nad i, po novem spodnji, zato postavimo spodnji na nad[i].

Ko vrnemo zvezek i na vrh kupa, leži tisti, ki je bil doslej zgornji, tik pod njim, torej moramo z nad[zgornji] pokazati na i, s pod[i] pa na zgornji; in nato i postane novi zgornji.

Koristno je posebej obravnavati primer, ko je i pred odstranitvijo ležal na vrhu kupa; tedaj se kup s tem, ko zvezek i odstranimo in nato odložimo na vrh, sploh ne spremeni in nam ni treba v takem primeru narediti ničesar.

Na koncu se lahko sprehodimo po seznamu od spodnjega zvezka proti vrhu (pri tem sledimo povezavam iz tabele nad) in pri tem štejemo, mimo koliko zvezkov smo že šli. Tako za trenutni zvezek vemo, koliko zvezkov je že pod njim; to vrednost si nekje zapomnimo in na koncu te vrednosti izpišemo. Spodnji program v ta namen zlorablja kar tabelo pod, saj je takrat ne potrebuje več.

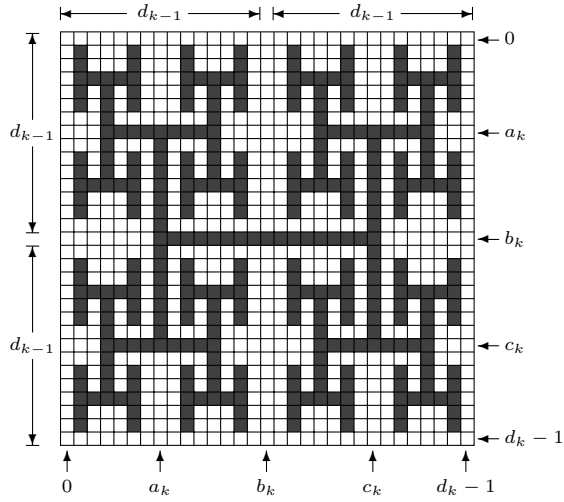
```
#include <stdio.h>

#define MaxN 10000

int main()
{
  int n, k, i, j, nad[MaxN + 1], pod[MaxN + 1], spodnji, zgornji;
  FILE *f = fopen("kup.in", "rt");
  fscanf(f, "%d %d", &n, &k);
  /* Inicializirajmo kup. */
  for (i = 1; i <= n; i++) nad[i] = (i == n) ? -1 : i + 1, pod[i] = (i == 1) ? -1 : i - 1;
  spodnji = 1; zgornji = n;
  /* Berimo vhodno datoteko. */
  while (k-- > 0)
  {
    fscanf(f, "%d", &i);
    /* Vzamemo i iz kupa. */
    if (nad[i] < 0) continue; /* i je bil zgornji zvezek */
```

če je $(a_1, \ldots, a_n)$ vhodna tabela, ki bi jo radi uredili, vzemimo $M > 2 \max_i a_i$ in definirajmo za vsak i pravokotnik, pravzaprav kvadrat, s koordinatami $[a_i, M - a_i] \times [a_i, M - a_i]$. Na tako dobljeni množici n pravokotnikov poženimo poljuben algoritem, ki reši problem naše naloge, torej poišče hierarhične odnose med pravokotniki. Vsak pravokotnik (razen najmanjšega) ima natanko enega neposredno podrejenega; če jih zdaj pregledamo od nadrejenih proti podrejenim in izpisujemo x-koordinate njihovih levih robov, smo dobili elemente vhodne tabele, $a_1, \ldots, a_n$, urejene v naraščajočem vrstnem redu.

```
      else pod[nad[i]] = pod[i];
      if (pod[i] < 0) spodnji = nad[i];
      else nad[pod[i]] = nad[i];
      /* Odložimo i na vrh kupa. */
      nad[i] = −1; pod[i] = zgornji;
      nad[zgornji] = i; zgornji = i;
    }
    fclose(f);
    /* Za vsak zvezek preštejmo, koliko jih je pod njim. */
    for (i = 0; i <= n; i++) pod[i] = k + 1;
    for (i = spodnji, k = 0; i > 0; i = nad[i]) pod[i] = k++;
    /* Izpišimo rezultate. */
    f = fopen("kup.out", "wt");
    for (i = 1; i <= n; i++) fprintf(f, "%d\n", pod[i]);
    fclose(f); return 0;
}
```

2. H-fraktal

Da bo manj pisanja, H-fraktal reda k imenujmo na kratko kar H_k. Naši fraktali so vsi kvadratne oblike; naj bo d_k dolžina stranice pri fraktalu H_k. Iz definicije v besedilu naloge sledi, da je $d_0 = 3$ in $d_{k+1} = 2d_k + 1$; iz tega lahko opazimo splošno formulo $d_k = 2^{k+2} - 1$. Stolpci pri H_k so torej oštevilčeni od 0 do $d_k - 1$. Ta fraktal je sestavljen iz štirih izvodov fraktala H_{k-1}, od katerih ima vsak obliko kvadrata s stranico d_{k-1}. Leva dva od njih se torej v H_k raztezata prek stolpcev od 0 do $d_{k-1} - 1$; nato sledi en pretežno prazen stolpec (s številko $b_k := d_{k-1}$), ki je ravno na sredi fraktala H_k; nato pa sledijo stolpci od $d_{k-1} + 1$ do $d_k - 1$, v katerih ležita desna dva izvoda fraktalov H_{k-1}. Srednji stolpec v levi polovici fraktala ima številko $a_k := d_{k-2}$, srednji stolpec v desni polovici pa številko $c_k := d_{k-1} + 1 + d_{k-2}$. Enak razmislek lahko seveda naredimo tudi pri vrsticah. Primer kaže naslednja slika:



Zdaj lahko razmišljamo takole: nekatera polja so na H_k črna zato, ker so del velikega H-ja, ki povezuje štiri kopije fraktala H_{k-1}. Navpični prečki tega H-ja sta na x-koordinatah a_k in c_k, po y-koordinati pa se raztezata od vključno a_k do vključno c_k. Vodoravna prečka tega H-ja je na y-koordinati b_k, po x-koordinati pa se razteza od vključno a_k do vključno c_k. S temi pogoji ni težko preveriti, ali neko polje leži na tem H-ju ali ne.

Nadalje lahko iz definicije v opisu naloge vidimo, da so vsa ostala polja v srednji vrstici in stolpcu (torej tista, pri katerih je ena od koordinat enaka b_k) prazna.

Ostala polja pa pripadajo eni od štirih kopij fraktala H_{k-1}, ki smo jih uporabili pri sestavljanju fraktala H_k. Za ta polja lahko z rekurzivnim klicem preverimo, ali so črna v okviru fraktala H_{k-1}. Paziti moramo le na to, da če je na primer x-koordinata našega opazovanega polja večja od b_k (kar pomeni, da leži v eni od desnih dveh kopij fraktala H_{k-1}), ji moramo pred rekurzivnim klicem odšteti $b_k + 1$ (kajti podprogram, ki se bo ukvarjal s fraktalom H_{k-1}, bo seveda predpostavljal, da ima opravka s koordinatami od 0 do $d_k - 1$), podobno pa je tudi z y-koordinato.

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

bool JeCrno(int x, int y, int k)
{
  /* a = sredina leve polovice, b = sredina celega fraktala, c = sredina desne polovice. */
  int a = (1 << k) - 1, b = (1 << (k + 1)) - 1, c = 3 * (1 << k) - 1;
  /* Pri k = 0 je črno le srednje polje, ostala so bela. */
  if (k == 0) return x == 1 && y == 1;
  /* Polja na vodoravni prečki H-ja so črna. */
  if (y == b && a <= x && x <= c) return true;
  /* Polja na navpičnih prečkah H-ja so črna. */
  if ((x == a || x == c) && a <= y && y <= c) return true;
  /* Ostala polja v srednji vrstici in stolpcu so bela. */
  if (x == b || y == b) return false;
  /* Za ostala polja lahko barvo ugotovimo z rekurzivnim klicem. */
  if (x > b) x -= b + 1;
  if (y > b) y -= b + 1;
  return JeCrno(x, y, k - 1);
}

int main()
{
  FILE *f = fopen("fraktal.in", "rt");
  int x, y, w, h, k, i, j;
  fscanf(f, "%d %d %d %d %d", &k, &x, &y, &w, &h);
  fclose(f);
  f = fopen("fraktal.out", "wt");
  for (i = 0; i < h; i++) {
    for (j = 0; j < w; j++)
      fputc(JeCrno(x + j, y + i, k) ? '#' : '.', f);
    fputc('\n', f); }
  fclose(f); return 0;
}
```

Naloge se lahko lotimo tudi tako, da si pripravimo v pomnilniku tabelo velikosti $d_k \times d_k$ in vanjo narišemo cel fraktal, nato pa zahtevani del fraktala izpišemo v izhodno datoteko. Tudi v tem primeru si lahko pomagamo z rekurzijo (fraktal H_k

narišemo tako, da najprej z rekurzivnim klicem narišemo štiri kopije H_{k-1} in nato pobarvamo še polja, ki tvorijo povezovalni H). Vendar pa ta rešitev ni kaj dosti preprostejša od zgornje, poleg tega pa pri večjih k-jih porabi neobvladljivo veliko pomnilnika. Celoten fraktal H_k ima $d_k \times d_k \approx 4^{k+2}$ polj in četudi za vsako polje porabimo le en bit pomnilnika, bi pri $k = 15$ potrebovali že 2 GB pomnilnika, pri $k = 20$ pa kar 2 TB.

Oglejmo si še, koliko ima fraktal H_k črnih polj. Označimo to število z u_k. Fraktal H_k je sestavljen iz štirih kopij fraktala H_{k-1}, povezanih z velikim H; slednjega tvorijo tri prečke s po $c_k - a_k - 1 = 2^{k+1} - 1$ črnimi polji. Tako dobimo rekurzivno zvezo $u_k = 4u_{k-1} + 3(2^{k+1} - 1)$ z robnim primerom $u_0 = 1$. Iz tega lahko izpeljemo tudi eksplicitno formulo $u_k = 3 \cdot 2^{k+1}(2^k - 1) + 1$. Delež črnih polj je tako $u_k/(d_k)^2 \approx 3 \cdot 2^{2k+1}/2^{2k+4} = 3/8$.

3. Slepe ulice

Če pripada neko križišče le eni ulici, je ta ulica gotovo slepa (če se po njej pripeljemo v tisto križišče, ne bomo mogli nazaj drugače kot z obratom za 180 stopinj). Podobno, če se v nekem križišču sicer stika več ulic, vendar smo vse razen ene že prepoznali za slepe, potem je tudi tista ena preostala ulica slepa. O tem se lahko prepričamo takole: naj bo u naše križišče in naj bodo $e_1, \ldots, e_d$ ulice, ki se stikajo v njem; recimo, da smo vse razen e_1 že spoznali za slepe. Recimo, da e_1 ni slepa. To pomeni, da če se pripeljemo po njej v u, lahko pot nadaljujemo tako, da se nekoč kasneje spet peljemo po e_1, ne da bi vmes kdaj naredili obrat za 180 stopinj. Brez izgube za splošnost recimo, da se ta pot nadaljuje po e_2. Glede tega, kako ta pot ponovno prepelje ulico e_1, pa ločimo dve možnosti: ali se pelje po njej v u ali pa iz u. Če se pelje po njej v u, lahko nadaljujemo pot po e_2, kar je v protislovju s predpostavko, da je e_2 slepa. Če pa se pelje po njej iz u, pomeni, da se je morala najprej nekako pripeljati v u; po e_2 se ni mogla, ker bi bilo to v protislovju s predpostavko, da je e_2 slepa; če pa je to storila po eni od ulic $e_3, \ldots, e_d$, bi se lahko pot iz u namesto po e_1 nadaljevala po e_2, kar bi bilo spet v protislovju s predpostavko, da je e_2 slepa. V vsakem primeru torej vidimo, da nas domneva, da e_1 ni slepa, pripelje v protislovje, torej je e_1 slepa.

Postopek za odkrivanje slepih ulic je torej takšen: na začetku razglasimo vse ulice za ne-slepe; če obstaja kakšno križišče, do katerega pelje le ena ne-slepa ulica (in nič ali več slepih), razglasimo tisto ulico za slepo; ta korak ponavljamo, dokler se da.

Ob koncu tega postopka za vsako križišče velja, da se v njem dotikata ali vsaj dve ne-slepi ulici ali pa nobena (če bi bila ne-slepa ulica v tem križišču ena sama, bi jo naš postopek razglasil za slepo). Križišče, pri katerem so vse ulice slepe, bomo imenovali slepo križišče, ostala pa so ne-slepa križišča. Table moramo postaviti na natanko tiste ulice, ki povezujejo slepo križišče z ne-slepim.

Spodnji program hrani podatke o omrežju v več tabelah: najprej prebere krajišča ulic v tabeli `us` in `vs`; v tabeli `deg` izračuna za vsako križišče njegovo stopnjo, torej koliko ulic se stika v njem; v `sdeg` pa piše, koliko od teh ulic je slepih. Tabela `ns` hrani za vsako križišče seznam sosedov, in sicer so sosedje križišča `u` shranjeni na indeksih od `fn[u]` do `fn[u] + deg[u] - 1`.

Za učinkovito pregledovanje omrežja si pomagamo z vrsto, v katero odlagamo križišča, ki smo jih prepoznali kot slepa. V vsaki iteraciji glavne zanke vzamemo neko križišče iz vrste in dodamo vanjo njegovega morebitnega ne-slepega soseda (če

tak sosed obstaja; več kot eden pa zagotovo ni). Spotoma v tabelo slepa vpisujemo podatke o tem, katera križišča so slepa.

Na koncu se le še sprehodimo po seznamu povezav in izpišemo tiste, pri katerih je eno krajišče slepo, drugo pa ne. Ker naloga zahteva izpis v naraščajočem vrstnem redu, smo seznam povezav pred nadaljnjo obdelavo uredili naraščajoče.

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

#define MaxN 500000
#define MaxM 500000

int deg[MaxN], sdeg[MaxN], us[2 * MaxM], vs[2 * MaxM], ns[2 * MaxM],
    fn[MaxN], vrsta[MaxN];

int main()
{
  int i, u, v, n, m, mm, glava, rep;
  bool slepa[MaxN];
  FILE *f = fopen("slepe.in", "rt");
  fscanf(f, "%d %d", &n, &m);
  printf("%d %d\n", n, m);

  /* Preberimo krajišča povezav v (us, vs) in izračunamo stopnje točk. */
  for (u = 0; u < n; u++) deg[u] = 0;
  for (i = 0; i < m; i++) {
    fscanf(f, "%d %d", &u, &v);
    us[i] = u; vs[i] = v; deg[u]++; deg[v]++; }

  /* fn[u] je indeks prve sosede točke u v tabeli ns. */
  fn[0] = 0; for (u = 1; u < n; u++) fn[u] = fn[u - 1] + deg[u - 1];

  /* Pripravimo sezname sosed v tabeli ns. */
  for (u = 0; u < n; u++) deg[u] = 0;
  for (i = 0; i < m; i++) {
    u = us[i]; v = vs[i]; ns[fn[u] + deg[u]++] = v; ns[fn[v] + deg[v]++] = u; }

  /* S pomočjo seznamov sosed pripravimo v (us, vs) seznam krajišč,
     pri čemer zdaj vsaka povezava nastopa dvakrat (kot (u, v) in (v, u)),
     urejene pa so naraščajoče po drugi komponenti (v). */
  for (u = 0, mm = 0; u < n; u++) for (i = 0; i < deg[u]; i++, mm++)
    us[mm] = ns[fn[u] + i], vs[mm] = u;

  /* Zdaj povezave na novo prenesimo v sezname sosed; to bo zagotovilo,
     da so sosede v vsakem seznamu urejene naraščajoče (kar bo prišlo prav
     pri izpisu rezultatov). */
  for (u = 0; u < n; u++) deg[u] = 0;
  for (i = 0; i < mm; i++) { u = us[i]; v = vs[i]; ns[fn[u] + deg[u]++] = v; }

  /* Dodajmo v vrsto vse točke s stopnjo 1. */
  for (u = 0, glava = 0, rep = 0; u < n; u++) {
    sdeg[u] = 0; slepa[u] = false; if (deg[u] == 1) vrsta[rep++] = u; }

  /* Pregledujmo graf. */
  while (glava < rep) {

    /* Iz vrste vzemimo točko u, jo razglasimo za slepo in v mislih pobrišimo povezave
       do njenih neslepih sosed. Če kakšni sosedi pade stopnja na 1, jo dodajmo v vrsto. */
    u = vrsta[glava++]; slepa[u] = true;
    for (i = 0; i < deg[u]; i++) {
      v = ns[fn[u] + i]; sdeg[v]++;
```

```
        if (slepa[v]) continue;
        if (sdeg[v] == deg[v] − 1) vrsta[rep++] = v; } }
    /* Izpišimo povezave, ki imajo prvo krajišče neslepo, drugo pa slepo.
       Način, kako smo sestavili sezname sosedov, nam zagotavlja, da bomo
       povezave izpisali urejene tako, kot zahteva naloga. */
    f = fopen("slepe.out", "wt");
    for (u = 0; u < n; u++) {
      deg[u] = (u == n − 1 ? mm : fn[u + 1]) − fn[u];
      for (i = 0; i < deg[u]; i++) {
        v = ns[fn[u] + i];
        if (slepa[v] && ! slepa[u]) fprintf(f, "%d %d\n", u, v); } }
    fclose(f);
    return 0;
}
```

Pri omrežju z n križišči in m ulicami ima ta postopek časovno zahtevnost le $O(n+m)$.

4. Križanka

Zelo naivna rešitev je, da gremo v gnezdenih zankah po vseh možnih velikostih pravokotnika in po vseh možnih koordinatah zgornjega levega kota; pri vsakem pravokotniku pregledamo vsa njegova polja, da vidimo, če je med njimi kakšno črno; največji povsem bel pravokotnik pa si zapomnimo v spremenljivki M. V spodnji psevdokodi predstavlja $črno[y,x]$ logično vrednost, ki nam pove, ali je polje (x,y) črno ali ne.

$M := 0$; **for** $r_w := 1,\ldots,w$, **for** $r_h := 1,\ldots,h$:
  for $r_x := 0,\ldots,w-r_w-1$, **for** $r_y := 0,\ldots,h-r_h-1$:
    $ok :=$ **false**;
    for $x := r_x,\ldots,r_x+r_w-1$, **for** $y := r_y,\ldots,r_y+r_h-1$:
      if $črno[y,x]$ **then** $ok :=$ **false**; **break**;
    if ok **then** $M := \max\{M, r_w \cdot r_h\}$;

Vendar pa je ta rešitev zaradi toliko gnezdenih zank zelo počasna; v najslabšem primeru (na popolnoma beli mreži) bi se izvajala kar $O(w^3h^3)$ časa. Preprosta izboljšava je, da pravokotnike, ki niso večji od največjega doslej znanega popolnoma belega, kar preskočimo (še preden začnemo z zankama po r_x in r_y, preverimo, ali je $r_w \cdot r_h > M$). To lahko precej pomaga, vendar je tudi takšna rešitev prepočasna (od naših desetih testnih primerov ne bi nobenega rešila dovolj hitro).

Pravokotniki, ki jih pregledujemo, imajo seveda marsikaj skupnega; na primer, če fiksiramo zgornji levi kot na (r_x, r_y) in povečamo višino r_h za 1, vsebuje novi pravokotnik vsa polja, ki jih je vseboval prejšnji, poleg njih pa še nekaj dodanih polj v svoji najbolj spodnji vrstici. Če pravokotnik pri višini r_h ni bil povsem bel, tudi pri r_h+1 (in tako naprej) ne bo, torej nam takih sploh ni treba gledati; če pa je bil pri r_h povsem bel, ga lahko pri r_h+1 preverimo že tako, da pogledamo samo polja v zadnji vrstici pravokotnika; za vsa ostala že od prej vemo, da so bela.

$M := 0$; **for** $r_y := 0,\ldots,h-1$, **for** $r_x := 0,\ldots,w-1$:
  for $r_w := 1,\ldots,w-r_x$:
    for $r_h := 1,\ldots,h-r_y$:
      $ok :=$ **false**;

```
for x := r_x, ..., r_x + r_w − 1:
  if črno[r_y + r_h − 1, x] then ok := false; break;
if not ok then break
else M := max{M, r_w · r_h};
```

Ta rešitev ima v najslabšem primeru časovno zahtevnost $O(w^3h^2)$. Če je w veliko večji od h, je koristno vhodne podatke pred obdelavo transponirati. Zanki po r_w in r_h bi lahko tudi obrnili in se torej spraševali, kaj se zgodi, če pravokotnik razširimo za en stolpec (namesto za eno vrstico); takšen postopek bi imel zahtevnost $O(w^2h^3)$; vendar pa je zaradi lokalnosti pri dostopih do pomnilnika bolje, če pravokotnik pregledujemo bolj po vrsticah kot po stolpcih. Še ena preprosta, vendar zelo koristna izboljšava te rešitve je tale: pri trenutnih r_y in r_w vemo, da ima pravokotnik lahko ploščino največ $r_w \cdot (h - r_y)$, saj se dlje kot do dna mreže ne more raztegniti. Če je ta zmnožek manjši ali enak M, lahko s trenutnim r_w takoj zaključimo in se posvetimo naslednjemu.

V prejšnji rešitvi je najbolj notranja zanka (po x) porabila precej časa za ugotavljanje, ali stoji $(r_x, r_y + r_h - 1)$ na levem koncu vodoravnega bloka vsaj r_w belih polj. Ista polja bomo morali pregledati po večkrat, saj bomo nanje naleteli pri različnih kombinacijah r_x in r_w. Opazimo lahko, da je mogoče te reči izračunati vnaprej in si jih shraniti v tabeli:

```
for y := 0, ..., h − 1:
  desno[y, w] := 0;
  for x := w − 1, ..., 0:
    desno[y, x] := desno[y, x + 1];
    if črno[y, x] then desno[y, x] := desno[y, x] + 1;
```

Ta izračun nam je vzel $O(wh)$ časa in zdaj imamo v $desno[y, x]$ vrednost, ki nam pove, koliko polj od (x, y) naprej (vključno z njim), gledano v desni smeri, je belih (do prvega črnega). S pomočjo te tabele se lahko znebimo najbolj notranje zanke v prejšnjem postopku:

```
M := 0; for r_y := 0, ..., h − 1, for r_x := 0, ..., w − 1:
  for r_w := 1, ..., w − r_x:
    for r_h := 1, ..., h − r_y:
      ok := (desno[r_y + r_h − 1, r_x] ≥ r_w);
      if not ok then break
      else M := max{M, r_w · r_h};
```

Tako smo prišli do postopka s časovno zahtevnostjo $O(w^2h^2)$. Enako kot prej bi lahko tudi tu dodali še pogoj, ki bi nad trenutnim r_w takoj obupal, če bi veljalo $r_w \cdot (h - r_y) \leq M$.

Vidimo lahko, da ta postopek tabelo *desno* pregleduje bolj po vrsticah kot po stolpcih, torej bi bilo zaradi večje učinkovitosti spet koristno obrniti vrstni red zank: najprej po r_h, nato po r_w, namesto tabele *desno* pa bi imeli tabelo *pod*, ki bi nam povedala, koliko belih polj leži pod (x, y), vključno z njim samim. Na asimptotično časovno zahtevnost taka sprememba nič ne vpliva, je pa v praksi zaradi nje program lahko nekajkrat hitrejši.

```
for x := 0,...,w−1: pod[h,x] := 0;
for y := h−1,...,0:
  for x := 0,...,w−1:
    pod[y,x] := pod[y+1,x];
    if črno[y,x] then pod[y,x] := pod[y,x] + 1;
```

```
M := 0; for r_y := 0,...,h−1, for r_x := 0,...,w−1:
  for r_h := 1,...,h−r_y:
    for r_w := 1,...,w−r_x:
      ok := (pod[r_y, r_x + r_w − 1] ≥ r_h);
      if not ok then break
      else M := max{M, r_w · r_h};
```

Zdaj lahko tudi opazimo, da ni prave potrebe po tem, da se zapičimo v nek fiksen r_h. Če smo že fiksirali zgornji levi kot pravokotnika, bi bilo smiselno pri posamezni širini r_w gledati le največjo dopustno višino pravokotnika (torej največjo táko, pri kateri je pravokotnik še popolnoma bel).

```
M := 0; for r_y := 0,...,h−1, for r_x := 0,...,w−1:
  r_h := ∞;
  for r_w := 1,...,w−r_x:
    r_h := min{r_h, pod[r_y, r_x + r_w − 1]};
    if r_h = 0 then break;
    else M := max{M, r_w · r_h};
```

V najbolj notranji zanki lahko ustavitveni pogoj $r_h = 0$ zamenjamo z $(w - r_x) \cdot r_h \leq M$, kar bo še bolje (odnehamo, če je pravokotnik že zdaj tako nizek, da po ploščini ne bi presegel M niti tedaj, če bi ga raztegnili čisto do desnega roba križanke). V vsakem primeru pa ima ta postopek časovno zahtevnost le še $O(w^2 h)$.

Lahko pa gremo še korak naprej. Za največji beli pravokotnik vsekakor velja, da se na levem robu tišči kakšnega črnega polja ali pa levega roba križanke — če to ne bi bilo res, bi ga lahko na levi strani razširili še za en stolpec in tako dobili nek še večji bel pravokotnik. (Enak razmislek lahko seveda ponovimo tudi za ostale tri robove tega pravokotnika.) Naši dosedanji postopki so pregledali vse pare (r_x, r_y) in se pri vsakem vprašali: „kako velik je največji beli pravokotnik z zgornjim levim kotom (r_x, r_y)?“ Zdaj pa vidimo, da bi se lahko vprašali drugače: „kako velik je največji beli pravokotnik, ki se na levem robu dotika polja (r_x, r_y)?“ Tu ni treba pregledati vseh parov (r_x, r_y), pač pa le tiste, ki predstavljajo črna polja ali pa ležijo tik levo od roba križanke. To vprašanje lahko preoblikujemo še takole: „kako velik je največji beli pravokotnik, ki v svojem levem stolpcu pokriva tudi polje (r_x, r_y)?“ Zdaj mora iti (r_x, r_y) po vseh poljih, ki imajo na levi črnega soseda ali pa ležijo v najbolj levem stolpcu križanke.

Zdaj torej ni nujno, da se pravokotnik v celoti razteza pod r_y, ampak lahko štrli tudi nad to y-koordinato, saj (r_x, r_y) ni več zgornje levo polje pravokotnika, pač pa le neko polje iz njegovega najbolj levega stolpca. Poleg tabele `pod` bomo zato potrebovali tudi tabelo `nad`:

```
for x := 0,...,w−1: nad[0,x] := 0;
for y := 1,...,h−1:
```

```
for x := 0, ..., w − 1:
  nad[y, x] := nad[y + 1, x];
  if črno[y − 1, x] then nad[y, x] := nad[y, x] + 1;
```

Za razliko od tabele pod torej tabela nad ne šteje polja (x, y) samega, pač pa le tista nad njim. Glavni del našega postopka je zdaj lahko takšen:

```
M := 0;
for r_y := 0, ..., h − 1:
  r_x := 0;
  while r_x < w:
    if x > 0 then if not črno[y, x − 1] then r_x := r_x + 1; continue;
    h_nad := ∞; h_pod := ∞; r_w := 0;
    while r_x + r_w < w:
      if črno[r_y, r_x + r_w] then break;
      h_nad := min{h_nad, nad[r_y, r_x + r_w]};
      h_pod := min{h_pod, pod[r_y, r_x + r_w]};
      r_w := r_w + 1;
      M := max{M, r_w · (h_nad + h_pod)};
    r_x := r_x + r_w;
```

Namesto ene višine r_h imamo zdaj dve, h_{nad} (za tisto, kar leži nad r_y) in h_{pod} (za tisto, kar leži na višini r_y ali pod njo). Časovna zahtevnost tega postopka je le še $O(wh)$, saj tista polja, ki smo jih pregledali v najbolj notranji zanki, nato preskočimo ($r_x := r_x + r_w$). Podobno kot prej bi lahko v najbolj notranjo zanko dodali še pogoj, ki bi jo prekinil, če bi se izkazalo, da je $(w - r_x) \cdot (h_{nad} + h_{pod}) \leq M$. Lahko bi celo obudili k življenju tabelo *desno* in namesto $(w - r_x)$ uporabili $desno[r_y, r_x]$, ki je še tesnejša ocena za največjo možno širino pravokotnika pri trenutnem (r_x, r_y).

Oglejmo si še malo drugačen algoritem s časovno zahtevnostjo $O(wh)$, ki ga je predlagal D. Vandevoorde.[4] Recimo, da se omejimo na pravokotnike, ki se končajo v vrstici r_y. Vrednost $nad[r_y, r_x]$ naj nam tokrat označuje število belih polj nad (r_x, r_y), vključno s tem poljem samim. Pregledujmo trenutno vrstico od leve proti desni; ko se višina (iz tabele *nad*) poveča, dodajmo novo vrednost na sklad, ob njej pa si zapomnimo še trenutni r_x. Ko pa se višina iz tabele *nad* zmanjša, pobiramo s sklada zapise, pri katerih je višina večja od trenutne; ker imamo pri vsakem od njih tudi x-koordinato, pri kateri se je ta višina začela, lahko iz nje vidimo, da bi se pravokotnik s takšno višino lahko začel pri tisti x-koordinati in končal tik pred trenutno x-koordinato.

```
M := 0;
for r_y := 0, ..., h − 1:
  S := prazen sklad, na katerem bodo pari oblike ⟨višina, x⟩;
  dodaj ⟨0, −1⟩ na S;
  for r_x := 0, ..., w:
    if r_x = w then v := 0 else v := nad[r_y, r_x];
    while ima element na vrhu S višino, večjo od v:
      pobriši vrhnji element (v′, x) s sklada;
```

[4] David Vandevoorde, *The maximal rectangle problem*, Dr. Dobb's Journal, April 1998.

$M := \max\{M, (r_x - x) \cdot v'\}$;
if ima element na vrhu S višino, manjšo od x:
dodaj $\langle v, x\rangle$ na S;

Ker dodamo pri vsakem polju križanke na sklad največ en element, je tudi brisanj s sklada toliko in časovna zahtevnost celotnega postopka je $O(wh)$. Pri naših poskusih je bil ta algoritem približno dvakrat hitrejši od prejšnjega algoritma z zahtevnostjo $O(wh)$.

Še implementacija Vandevoordejevega algoritma v C-ju:

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

#define MaxW 3000
#define MaxH 3000

char crno[MaxH][MaxW + 2];
int w, h;

int Resi()
{
  int hSklad[MaxW], xSklad[MaxW], sp, nad[MaxW + 1], x, y, xOd, naj = 0, kand;
  for (x = 0; x < w; x++) nad[x] = 0;
  for (y = 0; y < h; y++)
  for (x = 0, sp = 0; x <= w; x++)
  {
    nad[x] = (x == w || crno[y][x]) ? 0 : (y == 0 ? 0 : nad[x]) + 1;
    xOd = x;
    while (sp > 0 && hSklad[sp - 1] > nad[x])
    {
      --sp; xOd = xSklad[sp]; kand = (x - xOd) * hSklad[sp];
      if (kand > naj) naj = kand;
    }
    if (! (sp > 0 && hSklad[sp - 1] == nad[x])) {
      hSklad[sp] = nad[x]; xSklad[sp++] = xOd; }
  }
  return naj;
}

int main()
{
  int x, y;
  FILE *f = fopen("krizanka.in", "rt");
  fscanf(f, "%d %d\n", &w, &h);
  for (y = 0; y < h; y++) {
    fgets(crno[y], MaxW + 2, f);
    for (x = 0; x < w; x++)
      crno[y][x] = (crno[y][x] == '#' ? 1 : 0); }
  fclose(f);
  f = fopen("krizanka.out", "wt"); fprintf(f, "%d\n", Resi()); fclose(f); return 0;
}
```

5. Poštar

Poiščimo najnižje križišče, ki ga je treba obiskati (tisto z najvišjo y-koordinato; pri tej nalogi so vodoravne ulice oštevilčene tako, da kaže y-os dol, ne pa gor); recimo

mu (x_1, y_1). Neka pot bo vsekakor morala obiskati to križišče, pa recimo, da naj bo to kar prva pot. Med križišči, ki ležijo levo od njega, poiščimo najnižje; recimo temu (x_2, y_2). Neka pot bo vsekakor morala obiskati to križišče in tista, ki ga bo, ne bo mogla med x-koordinatama x_2 in x_1 obiskati ničesar drugega. Pa recimo, da bo (x_2, y_2) obiskala kar naša prva pot in da se bo od tam nadaljevala vodoravno do (x_2, y_1), od tam pa se bo spustila dol do (x_1, y_1). Poiščimo potem med križišči, ki ležijo levo od x_2, najnižje; recimo temu (x_3, y_3); tudi njega mora obiskati neka pot in recimo, da naj ga obišče kar naša prva pot. Tako nadaljujemo, dokler ne pridemo do najbolj levega križišča. Tako smo sestavili neko pot; križišča, ki jih je obiskala, pobrišimo in poženimo postopek znova, da nam sestavi naslednjo pot. Tako nadaljujemo, dokler ne obiščemo vseh križišč.

Ker delamo z mrežo velikosti največ 100×100, jo lahko predstavimo kar s tabelo logičnih vrednosti, v kateri vsaka celica pove, ali je treba tisto križišče še obiskati ali ne. Da ne bo treba že obdelanih delov tabele pregledovati po večkrat, si v tabeli rob zapomnimo potek prejšnje poti: to pomeni, da so v vrstici y križišča do x-koordinate rob[y] že obiskana (ker so jih obiskale dosedanje poti), torej lahko pri iskanju naslednjega neobiskanega križišča v tej vrstici začnemo na x-koordinati rob[y], ne pa že na $x = 1$. To nam zagotavlja, da časovna zahtevnost celotnega postopka ne bo presegla $O(wh)$.

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

#define MaxW 100
#define MaxH 100

int main()
{
  bool a[MaxH][MaxW]; int w, h, n, i, x, y, stPoti = 0, rob[MaxH + 1], xDo, yDo;
  FILE *f = fopen("postar.in", "rt");
  fscanf(f, "%d %d %d", &w, &h, &n);
  for (y = 0; y < h; y++) for (x = 0; x < w; x++) a[y][x] = false;
  for (i = 0; i < n; i++) { fscanf(f, "%d %d", &x, &y); x--; y--; a[y][x] = true; }
  fclose(f);

  for (y = 0; y < h; y++) rob[y] = 0; rob[h] = w; yDo = h - 1;
  /* rob[y] pove, da so v tabeli a vse vrednosti a[y][0..rob[y] - 1] enake false.
     yDo je najmanjše tako število, za katerega je rob[y] < w. */
  while (yDo >= 0)
  {
    for (y = yDo, xDo = rob[y + 1]; y >= 0; y--)
    {
      /* xDo je najbolj leva pošiljka, ki jo trenutna pot dostavi v vrstici y + 1.
         To pomeni, da v trenutni vrstici ne smemo iti bolj desno od te koordinate.
         Dosedanje poti so v vrstici y dostavile vse pošiljke na x-koordinatah, manjših
         od rob[y]. Trenutna pot lahko torej v tej vrstici dostavi vse pošiljke na
         x-koordinatah od rob[y] do xDo. Novi rob[y] se lahko po tem postavi na xDo.
         Novi xDo pa mora postati koordinata najbolj leve pošiljke, ki smo jo zdaj
         dostavili v vrstici y; če nismo dostavili nobene, ostane xDo nespremenjen. */
      x = rob[y]; while (x < xDo && ! a[y][x]) x++;
      rob[y] = x; while (x < xDo) a[y][x++] = false;
      xDo = rob[y]; rob[y] = x; if (x >= w) yDo = y - 1; else a[y][x] = false;
    }
    if (xDo < w) stPoti++;
```

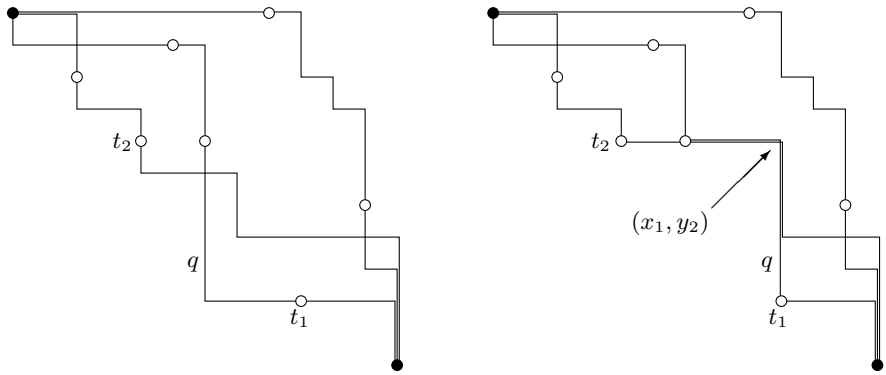
```
    if (stPoti > h) break;
  }

  f = fopen("postar.out", "wt");
  fprintf(f, "%d\n", stPoti); fclose(f); return 0;
}
```

Dokaz pravilnosti. Naj bo A množica točk, ki jih je treba pokriti. Dokazovali bomo z indukcijo po $|A|$. Pri $|A| = 1$ je očitno, da je požrešna rešitev optimalna, saj zahteva le eno pot, z manj kot eno potjo pa se ene točke ne da pokriti.

Naj bo zdaj $|A| = n > 1$ in recimo, da smo dokazali pravilnost požrešnega algoritma že za vse množice $n - 1$ ali manj točk. Vzemimo torej zdaj poljubno množico A z n točkami; naj bo $Q^\star$ najmanjša množica poti, ki pokrijejo vse točke iz množice A.

Poiščimo najnižjo točko iz A; če je več takih, vzemimo med njimi najbolj levo. Tej točki recimo $t_1 = (x_1, y_1)$. Vsako pot $p \in Q^\star$ predelajmo takole: če se spusti pod $y = y_1$ že pri $x < x_1$, je jasno, da od tam naprej ne bo obiskala nobene točke več, torej jo speljimo po $y = y_1$ v desno vse do $x = w$.

Neka pot iz $Q^\star$ vsekakor obišče tudi t_1; recimo tej poti q. Med točkami, ki so levo od t_1, vzemimo najnižjo; če je takih več, vzemimo med njimi najbolj levo; tej točki recimo $t_2 = (x_2, y_2)$. Iz tega sledi, da ni v A nobene točke, ki bi imela $x < x_1$ in hkrati $y > y_2$. Zdaj lahko vsako pot $p \in Q^\star$ (tudi $p = q$) predelamo takole: če se p kdaj spusti pod $y = y_2$ pri $x < x_1$ (recimo v točki (x', y_2)), potem poglejmo, kakšno y-koordinato ima pri $x = x_1$ (recimo, da je to (x_1, y'); seveda je $y \le y_1$); in potem jo popravimo tako, da gre od (x', y_2) desno do (x_1, y_2) in od tam dol v (x_1, y'). Primer takšne predelave kaže naslednja slika:



Po teh predelavah (ki ohranijo veljavnost rešitve) vemo, da vsaka pot prvič doseže $x = x_1$ pri $y \le y_2$. Za q to med drugim pomeni, da gre tudi skozi (x_1, y_2), saj pride do $x = x_1$ pri $y \le y_2$ in se mora nato spustiti do točke $t_1 = (x_1, y_1)$.

Če q ne obišče t_2, jo pa vsekakor obišče neka druga pot iz $Q^\star$, recimo p. Kot vsaka druga pot tudi p (kot smo videli na koncu prejšnjega odstavka) ostane pri $y \le y_2$ vse do $x = x_1$; no, ker je šla skozi (x_2, y_2), se kasneje ne more premakniti na nižje y-koordinate, torej lahko „$y \le y_2$“ spremenimo kar v „$y = y_2$“. Pot p gre torej

skozi točko (x_1, y_2), prav tako kot q (za slednjo smo to ugotovili na koncu prejšnjega odstavka). Potemtakem lahko potek poti p in q do točke (x_1, y_2) zamenjamo. Rešitev s tem ostane veljavna, pot q pa zdaj obišče tudi točko t_2.

Zdaj lahko vzamemo med točkami, ki so levo od t_2, najnižjo; če je takih več, pa med njimi najbolj levo; tej točki recimo $t_3 = (x_3, y_3)$. Z enakim razmislekom kot prej lahko vse poti predelamo tako, da se pod $y = y_3$ ne spustijo prej kot pri $x = x_2$; z enakim razmislekom kot prej lahko tudi vidimo, da če q še ne gre skozi t_3, lahko zamenjamo del te poti z neko drugo potjo, tako da bo potem q obiskala t_3. S tem razmislekom lahko nadaljujemo za točke t_4, t_5 in tako naprej; prej ali slej se zgodi, da ni nobene točke, ki bi bila bolj levo od $t_k = (x_k, y_k)$. Vse poti lahko tedaj predelamo tako, da gredo najprej od $(1, 1)$ do $(x_k, 1)$ in se šele tam začnejo spuščati (če sploh). Pot q je po vseh teh predelavah ravno prva pot, ki bi jo našel naš požrešni algoritem.

Naj bo A' množica tistih točk iz A, ki jih q ne obišče. Naj bo $r = |Q^\star|$. V $Q^\star$ je torej poleg q še $r - 1$ drugih poti, ki očitno vse skupaj obiščejo ravno vse točke iz A'. Po drugi strani bi požrešni algoritem, ko bi reševal problem A, najprej sestavil pot q, nato bi preostanek rešitve sestavil tako, da bi rešil problem A'. Ker je $|A'| < |A|$, iz induktivne predpostavke sledi, da bi bila požrešna rešitev za problem A' optimalna. Ker je $Q^\star$ uspela pokriti vse točke iz A' z $r - 1$ potmi, je jasno, da jih bo tudi naš požrešni algoritem uspel pokriti s kvečjemu $r - 1$ potmi. Skupaj s potjo q bo torej naš požrešni algoritem sestavil rešitev za A, ki bo vsebovala kvečjemu r poti, torej ni nič slabša od optimalne rešitve $Q^\star$.

Rešitev, ki gradi vse poti naenkrat. Nalogo lahko elegantno rešimo tudi tako, da poti ne tvorimo eno za drugo, ampak vse naenkrat. Pripravimo si seznam vseh križišč, ki jih je treba obiskati; uredimo jih od spodaj navzgor, tiste z isto y-koordinato pa od desne proti levi. Tudi poti bomo tvorili od spodaj navzgor; v vsakem trenutku bomo vzdrževali seznam poti, pri čemer bo vsaka predstavljena z najvišjim križiščem, ki smo ga doslej postavili na to pot (recimo (x_i, y_i) za i-to pot), poti v tem seznamu pa bodo urejene od leve proti desni. Ko se moramo za neko novo križišče, npr. (x, y), odločiti, na katero pot ga bomo postavili; v poštev pridejo le take z $x_i \geq x$ (saj bi drugače pot od (x, y) do (x_i, y_i) naredila premik v levo, tega pa naloga ne dovoli), med njimi pa moramo vzeti najbolj levo, torej tisto z najmanjšim x_i. To pot lahko poiščemo z bisekcijo. Če pa primerne poti sploh ni, torej če je $x > x_i$ za vse trenutno odprte poti, to pomeni, da moramo za (x, y) začeti novo pot (in jo dodati na konec seznama poti). Zapišimo ta postopek s psevdokodo:

```
1  P := prazen seznam;
2  uredi križišča, ki jih je treba obiskati, od spodaj navzgor,
     tista z isto y-koordinato pa od desne proti levi;
3  pregleduj križišča v tem vrstnem redu:
4     naj bo (x, y) trenutno križišče;
5     naj bo i najmanjši indeks v seznamu P, za katerega je P[i].x ≥ x;
6     če takega i sploh ni,
7        potem dodaj (x, y) na konec seznama P,
8        sicer P[i] := (x, y);
9  vrni dolžino seznama P;
```

Če imamo vsega skupaj n križišč, ki jih je treba obiskati, nam urejanje v vrstici 2

vzame $O(n \log n)$ časa, glavna zanka (vrstice 3–8) ima n iteracij, v vrstici 5 uporabimo bisekcijo in za to porabimo pri vsaki iteraciji $O(\log n)$ časa, tako da je časovna zahtevnost celotnega postopka $O(n \log n)$. Za primerjavo: naša prejšnja rešitev je porabila $O(wh)$ časa; naš novi postopek je torej slabši, če je treba obiskati veliko križišč (ko je n blizu wh), boljši pa, če je število križišč, ki jih moramo obiskati, majhno v primerjavi s številom vseh možnih križišč ($n \ll wh$).

Povezava s problemom najdaljšega padajočega podzaporedja. Ta naloga je v zelo zanimivi povezavi s problemom najdaljšega padajočega podzaporedja. Uredimo naše točke po naraščajočih x-koordinatah, tiste z enakim x pa naraščajoče po y. V tem vrstnem redu jih zdaj oštevilčimo: (x_i, y_i) za $i = 1, \ldots, n$. Recimo, da nas zanima najdaljše padajoče podzaporedje (lahko nestrnjeno) v zaporedju $Y := \langle y_1, \ldots, y_n \rangle$. Poiščemo ga lahko tako, da pregledujemo zaporedje Y od leve proti desni in za vsak k v spremenljivki p_k hranimo največji doslej obdelani y_i, pri katerem se konča kakšno padajoče podzaporedje dolžine k. Postopek je torej takšen:

1 $d := 0$; $p_{d+1} := -\infty$;
2 **for** $i := 1$ **to** n:
3 $\quad k :=$ najmanjši indeks (večji ali enak 1), pri katerem je $y_i \geq p_k$;
4 $\quad$ **if** $k = d + 1$ **then** $d := d + 1$; $p_{d+1} := -\infty$; $P_d :=$ prazen seznam;
5 $\quad p_k := y_i$; dodaj (x_i, y_i) na konec seznama P_k;

Na začetku vsake iteracije te zanke velja naslednja invarianta: najdaljše padajoče podzaporedje v doslej pregledanem delu zaporedja Y ima dolžino d in za vsak k od 1 do d je p_k največje število, pri katerem se (v doslej pregledanem delu zaporedja Y) konča kakšno nepadajoče podzaporedje dolžine k; o tem, da ta invarianta res drži, se lahko prepričamo z indukcijo po i.

Za začetek opazimo, da če invarianta v nekem trenutku velja, je zaporedje p nenaraščajoče. To sledi iz definicije: če se v neki točki z $y = p_k$ konča padajoče podzaporedje dolžine k, se v njej konča tudi padajoče zaporedje dolžine $k - 1$, torej mora biti p_{k-1} (ki je največja y-koordinata, pri kateri se konča kakšno padajoče zaporedje dolžine $k - 1$) večji ali enak p_k.

Pri $i = 1$ invarianta očitno velja. Preverimo zdaj, da jo zanka ohranja. Naj bo k vrednost, ki jo poišče vrstica 3, torej takšna, da je $p_{k-1} > y_i \geq p_k$. (Mislimo si $p_0 = \infty$, če je treba.) Pri $y = p_{k-1}$ se torej konča neko padajoče zaporedje dolžine $k - 1$, torej se pri $y = y_i$ konča neko padajoče zaporedje dolžine k; torej je y_i kandidat za novo vrednost p_k (in ker je $y_i \geq p_k$, je prav, da p_k popravimo na y_i). Ali je mogoče, da se pri y_i konča kakšno padajoče zaporedje dolžine t za $t > k$? To bi pomenilo, da se pri prejšnjem členu (recimo y') konča padajoče zaporedje dolžine $t - 1$, torej imamo $p_{t-1} \geq y' > y_i \geq p_k$, torej $p_{t-1} > p_k$; zaradi $t > k$ pa imamo $t - 1 \geq k$, torej $p_{t-1} \leq p_k$ (saj je zaporedje p nenaraščajoče), tako da smo prišli v protislovje. Torej zaradi elementa i ni treba popravljati vrednosti p_{k+1}, p_{k+2} itd. Kaj pa krajša podzaporedja? Videli smo, da se v točki i konča padajoče zaporedje dolžine k; seveda se v njem zato končajo tudi padajoča zaporedja dolžine t za $t < k$. Zato je y_i kandidat za novo vrednost p_t za vse te t. Todaj ker je $y_i < p_{k-1}$, je y_i manjši tudi od p_t za vse $t \leq k - 1$, zato nam teh p_t ni treba popravljati. □

Iz te invariante sledi, da je na koncu postopka d ravno dolžina najdaljšega padajočega podzaporedja, tako da vidimo, da opisani postopek res pravilno reši problem najdaljšega padajočega podzaporedja.

Zanimivo pa je še naslednje: če pogledamo seznam P_k za poljuben k od 1 do d, vidimo, da ima vsaka točka v njem vsaj tolikšno x-koordinato kot prejšnja (ker pregledujemo točke po naraščajočih i in torej tudi po naraščajočih x_i) in kvečjemu tolikšno y-koordinato kot prejšnja (ker smo k izbrali v vrstici 3 tako, da je y-koordinata nove točke, y_i, večja ali enaka kot y-koordinata prejšnje točke v seznamu, p_k). Vsak seznam P_k torej predstavlja neko veljavno pot poštarja, ki obišče eno ali več izmed zahtevanih križišč. Vsako križišče smo dodali v enega od teh seznamov, tako da teh d poti skupaj $(P_1, \ldots, P_d)$ obišče ravno vsa zahtevana križišča.

Iz tega torej vidimo, da je vsekakor mogoče obiskati vsa križišča z d potmi, pri čemer je d dolžina najdaljšega padajočega podzaporedja v zaporedju Y. Ali je mogoče, da obstaja še kakšna rešitev z manj kot d potmi? Mislimo si točke, ki tvorijo tisto najdaljše padajoče podzaporedje; to so recimo (x_{i_j}, y_{i_j}) za neke $1 \le i_1 < i_2 < \ldots < i_d \le n$. Ker so indeksi naraščajoči, velja $x_{i_1} \le x_{i_2} \le \ldots \le x_{i_d}$; in ker je bilo to padajoče podzaporedje v Y, velja $y_{i_1} > y_{i_2} > \ldots > y_{i_d}$. Vse te točke, $i_1, \ldots, i_d$, mora poštar nekoč obiskati; tu je d točk in če je poštar opravil manj kot d poti, mora vsaj ena pot obiskati več kot eno od teh točk; recimo torej, da neka pot obišče tako točko i_j kot točko i_t za $j < t$. Za tidve točki torej velja $x_{i_j} \le x_{i_t}$ in $y_{i_j} > y_{i_t}$; če bi veljalo $x_{i_j} = x_{i_t}$, bi iz $i_j < i_t$ sledilo $y_{i_j} < y_{i_t}$ (saj smo rekli, da točke z enakim x uredimo naraščajoče po y), kar bi bilo protislovje; torej je $x_{i_j} < x_{i_t}$; torej leži točka i_t desno in gor od i_j, tako da poštar ne more obiskati obeh na isti poti (če najprej obišče i_j, ne bo mogel gor do i_t, če pa najprej obišče i_t, ne bo mogel levo do i_j). Torej rešitev z manj kot d potmi ni mogoča.

Ta povezava med obema problemoma je zanimiva med drugim iz naslednjega razloga. Recimo, da bi imeli nek algoritem, ki bi rešil naš problem poštarja v manj kot $O(n \log n)$ časa. Potem bi lahko sestavili takšen algoritem za iskanje najdaljšega padajočega podzaporedja: če imamo vhodno zaporedje $Y = \langle y_1, \ldots, y_n \rangle$, ga predelamo v množico točk (i, y_i) za $i = 1, \ldots, n$ in na tako dobljeni množici točk poženemo algoritem za problem poštarja. Rezultat, ki ga vrne slednji, je obenem že tudi dolžina najdaljšega padajočega podzaporedja v zaporedju Y. Tako smo torej rešili problem najdaljšega padajočega podzaporedja v času, manjšem od $O(n \log n)$; to pa je protislovje, saj je znano, da se tega problema ne da rešiti v manj kot $O(n \log n)$ časa.[5] Tako vidimo, da se tudi problema poštarja ne da rešiti v manj kot $O(n \log n)$ časa, torej je rešitev, ki smo jo videli zgoraj, v asimptotičnem smislu že najboljša možna.

[5] M. L. Fredman, *On computing the length of the longest increasing subsequence*, Discrete Mathematics 11(1):29–35 (January 1975); je pa njegov razmislek temeljil na predpostavki, da z elementi zaporedja ne moremo početi drugega kot to, da jih primerjamo po velikosti (z relacijo $\le$). Če smemo predpostaviti še kaj drugega, npr. to, da so elementi zaporedja cela števila od 1 do u, je mogoče najkrajše padajoče podzaporedje poiskati v času $O(n \log \log u)$ z uporabo van Emde Boasovih dreves. Za več o tem problemu glej rešitev naloge 1992.3.3 na str. 148–152 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004* in literaturo, omenjeno tam v opombi na str. 152; zelo zanimivo branje je na primer Knuth, *The Art of Computer Programming*, 3. knjiga, § 5.1.4.

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Ruleta

Naša rešitev v zanki bere vhodno zaporedje znak za znakom. V vsaki iteraciji primerja trenutno izžrebano barvo (ki je v spremenljivki c) z barvo, na katero smo stavili (spremenljivka stava), in primerno popravi spremenljivko stanje (jo poveča za 10, če smo zadeli, in zmanjša za 10, če smo izgubili). Nato na podlagi trenutno izžrebane barve c določi, na katero barvo bomo stavili v naslednji iteraciji, in si to zapomni v spremenljivki stava.

```
#include <stdio.h>

int main()
{
  enum { Rdeca = '1', Crna = '2', Zelena = '3' };
  int c, stava = Rdeca, stanje = 0;
  while ((c = getc(stdin)) != EOF && '1' <= c && c <= '3') {
    if (c == stava) stanje += 10; else stanje -= 10;
    if (c == Rdeca) stava = Crna; else stava = Rdeca; }
  printf("%d\n", stanje); return 0;
}
```

2. Glava e-pošte

Vhodno besedilo lahko beremo znak za znakom. Preden nek znak izpišemo, pogledamo še, kakšen je naslednji znak; če je trenutni znak '\n' (znak za novo vrstico), naslednji pa presledek, trenutnega znaka ne izpišemo, drugače pa ga. Tako se bo vrstica, ki se začne na presledek, spojila s predhodno vrstico, saj ne bomo izpisali znaka za konec vrstice, ki je v vhodni datoteki stal med njima.

```
#include <stdio.h>

int main()
{
  int znak = fgetc(stdin), naslednji;
  while (znak != EOF)
  {
    naslednji = fgetc(stdin);
    if (! (znak == '\n' && naslednji == ' '))
      fputc(znak);
    znak = naslednji;
  }
  return 0;
}
```

Naloge bi se seveda lahko lotili tudi še drugače, na primer z branjem po vrsticah namesto po znakih.

3. Trajekt

Največje možno število vozil lahko spravimo na trajekt tako, da vozila najprej uredimo naraščajoče po dolžini in jih nato v tem vrstnem redu dodajamo na trajekt,

dokler se to še da; ustavimo se, ko pridemo do vozila, pri katerem bi skupna dolžina doslej izbranih vozil presegla dolžino trajekta.

O tem, da na ta način na trajekt res lahko spravimo največje možno število vozil, se lahko prepričamo takole. Naj bo R najboljša rešitev naloge, torej največji tak izbor vozil, pri katerem skupna dolžina vozil še ne preseže dolžine trajekta. Naj bo k število vozil v R. Če izbor R ni sestavljen ravno iz najkrajših k vozil, vzemimo najdaljše vozilo iz R in ga zamenjajmo z najkrajšim takim vozilom, ki ga doslej še ni bilo v R. Število vozil v R tako ostane enako, njihova skupna dolžina pa se še zmanjša, torej je novi izbor še vedno veljaven (ne presega dolžine trajekta). Če ta postopek nadaljujemo, pridemo sčasoma do izbora, v katerih je najkrajših k vozil. Torej najboljšega izbora ne bomo spregledali, četudi se že vnaprej omejimo na izbore, v katerih nastopa najkrajših nekaj vozil. Ravno te izbore pa pregleduje naša rešitev iz prvega odstavka; ker med njimi najde tistega z največ vozili (ki še ne preseže dolžine trajekta), je to tudi najboljši možni izbor sploh.

4. Ledene dobe

Preprosta rešitev je, da pregledujemo seznama z dvema gnezdenima zankama. Pri vsakem letu z visoko koncentracijo ogljikovega dioksida pregledamo celoten seznam ledenih dob, da vidimo, če se je katera zgodila v naslednjih 5000 letih. Recimo, da je $(a_1, \ldots, a_n)$ seznam let z visoko koncentracijo CO_2, $(b_1, \ldots, b_m)$ pa seznam let, ko so se začele ledene dobe. Postopek je torej tak:

```
r := 0;
for i := 1 to n:
  ok := false;
  for j := 1 to m:
    if a_i ≤ b_j ≤ a_i + 5000 then ok := true;
  if ok then r := r + 1;
```

Na koncu imamo v r število let, ki potrjujejo geologovo domnevo. Slabost tega postopka je, da ima časovno zahtevnost $O(n \cdot m)$, kar je lahko počasi, če sta seznama dolga. Razmislimo o tem, kako bi ga izboljšali. Pri notranji zanki nas pravzaprav ne zanimajo vse možne ledene dobe, ampak le prva taka, ki nastopi v letu a_i ali po njem. Če taka ledena doba obstaja in je najkasneje v letu $a_i + 5000$, je za našega geologa ugodna, sicer pa ne. Ker sta obe zaporedji urejeni naraščajoče, lahko razmišljamo takole: prva ledena doba, ki se začne v letu a_{i+1} ali po njem, je bodisi ista kot prva ledena doba, ki se začne v letu a_i ali po njem, ali pa ena od kasnejših ledenih dob; gotovo pa ne ena od zgodnejših. Če si torej zapomnimo, kje v zaporedju b smo bili, ko smo se ukvarjali z letom a_i, se moramo, ko preidemo z a_i na a_{i+1}, po zaporedju b bodisi premakniti za eno ali več mest naprej bodisi ostati na istem mestu. Tako se premikamo po obeh zaporedjih hkrati:

```
r := 0; j := 1;
for i := 1 to n:
  while j ≤ m:
    if b_j ≥ a_i then break else j := j + 1;
  if j ≤ m then if b_j ≤ a_i + 5000 then r := r + 1;
```

Ko se i premika po zaporedju a, se j počasi premika po zaporedju b in opravi vsega skupaj en prehod po njem, ne več po en prehod pri vsakem i. Časovna zahtevnost tega postopka je le še $O(n + m)$.

5. Fora

Naloge se lahko lotimo na več načinov. Ena od možnih rešitev je na primer naslednja: zaporedje prebiramo od začetka proti koncu in med tem v neki tabeli sproti vzdržujemo podatek o tem, v kateri zanki (v spodnji rešitvi je to spremenljivka k) je bil postopek iz besedila naloge, ko je izpisal trenutni člen zaporedja, in v kateri iteraciji posamezne zanke je takrat bil (recimo $n[i]$ za i-to vgnezdeno zanko). Ko se globina zanke poveča (naslednji vhodni element, recimo x, je večji od k), vemo, da se je začela nova vgnezdena zanka in njene iteracije štejemo od 1 naprej (spodnji program postavi n[x] na 0 in ga bo takoj zatem povečal za 1); v nasprotnem primeru pa se nadaljuje neka že odprta zanka in moramo njen števec samo povečati za 1. V vsakem primeru pa novi element x tudi pove, kakšna je po novem globina, na kateri se je program iz besedila naloge med izpisovanjem tega elementa nahajal.

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

int main()
{
  int k = 0, n[MaxK + 1], x;
  FILE *f = fopen("zaporedje.txt", "rt");
  while (1 == fscanf(f, "%d", &x)) {
    if (x > k) n[x] = 0;
    k = x; n[k]++; }
  printf("%d", k); for (x = 1; x <= k; x++) printf(" %d", n[x]); printf("\n");
  fclose(f); return 0;
}
```

Nalogo lahko rešimo tudi tako, da začnemo na koncu zaporedja. Zadnji element zaporedja je ravno enak k, globini gnezdenja zank. Premikajmo se od konca zaporedja proti začetku in štejmo, koliko k-jev vidimo, preden prvič naletimo na $k - 1$; število teh k-jev je ravno n_k. Od takrat naprej štejmo pojavitve $k - 1$, dokler prvič ne naletimo na $k - 2$; število teh pojavitev $k - 1$ je ravno n_{k-1}. Tako lahko nadaljujemo, dokler ne rekonstruiramo vseh n-jev.

Še ena možnost je, da za vsako število od 1 do k preštejemo, kolikokrat se pojavi v celem zaporedju; naj bo a_i število pojavitev vrednosti i. Potem lahko računamo takole: $n_1 = a_1$; $n_2 = a_2/n_1$; $n_3 = a_3/(n_1 n_2)$; $n_4 = a_4/(n_1 n_2 n_3)$ in tako naprej.

Naloge so sestavili: poštne številke, delitev kamenja — Andrej Bauer; H-fraktal — Nino Bašić; vodilni elementi — Primož Gabrijelčič; Parktronic — Boris Gašperin; poštar — Tomaž Hočevar; pravokotniki — Peter Keše; kvadrati s seštevanjem, reka presledkov — Mitja Lasič; fora — Borut Lesjak; ruleta — Matija Lokar; abecedni podnizi — Polona Novak; skrivanje tipk, glava e-pošte — Mark Martinec; cikel, kup knjig, slepe ulice, križanka — Mitja Trampuš; trajekt — Anže Žagar; ledene dobe — Klemen Žagar. Primer pri nalogi „Reka presledkov“ je iz 25. poglavja Gibbonove *Zgodovine zatona in propada Rimskega cesarstva*.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2008

1. Statistika

Spodnja rešitev bere vhodno besedilo znak po znak. Ko naleti na konec besede (presledek), poveča števec besed (stBesed), pri končnem ločilu pa tako števec besed kot števec stavkov (stStavkov). Če pa je trenutni znak črka, povečamo števec črk (stCrk); če je samoglasnik, povečamo še števec zlogov (stZlogov). Na koncu lahko iz teh spremenljivk izračunamo povprečja, po katerih sprašuje naloga.

Če bi povečali števec besed pri vsakem presledku, bi naleteli na težave v primeru, ko sta dve besedi ločeni z več kot enim presledkom. Podobno velja za štetje stavkov v primeru, ko je na koncu stavka več kot eno ločilo (npr. `?!` ali pa tri pike). Zato ima naša rešitev še spremenljivki vBesedi in vStavku, ki povesta, ali se trenutno sploh nahajamo v besedi ali stavku. Ko na primer naletimo na presledek, povečamo števec stBesed le, če smo trenutno bili v besedi; in ker presledek pomeni, da se je trenutna beseda končala, postavimo takrat vBesedi na **false**. S tem smo zagotovili, da če bo naslednji znak tudi presledek, pri njem števca besed ne bomo povečali še enkrat.

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

int main()
{
  int stCrk = 0, stZlogov = 0, stBesed = 0, stStavkov = 0, c;
  bool vBesedi = false, vStavku = false;
  do {
    c = fgetc(stdin);
    if (c == EOF || c == '.' || c == '?' || c == '!' || c == ' ') {
      if (vBesedi) stBesed++, vBesedi = false;
      if (c != ' ' && vStavku) stStavkov++, vStavku = false; }
    else if (isalpha(c)) {
      vBesedi = true; vStavku = true; stCrk++;
      c = tolower(c);
      if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') stZlogov++; }
  }
  while (c != EOF);
  printf("Povprečna beseda ima %.1f črk in %.1f zlogov.\n",
    stCrk / (double) (stBesed <= 0 ? 1 : stBesed),
    stZlogov / (double) (stBesed <= 0 ? 1 : stBesed));
  printf("Povprečni stavek ima %.1f besed.\n",
    stBesed / (double) (stStavkov <= 0 ? 1 : stStavkov));
  return 0;
}
```

Pogoj isalpha(c) je pravzaprav rahlo odveč, saj naloga pravi, da se v vhodnem besedilu tako ali tako ne bodo pojavljali drugi znaki kot črke, presledki in končna ločila.[6] Do primera, ko je c == EOF, pa pride na koncu vhodnih podatkov; to vrednost obravnavamo enako kot končna ločila, tako da bomo pravilne rezultate dobili tudi, če se zadnji stavek ne konča s končnim ločilom.

[6] V praksi je lahko rezanje besedila na besede in stavke precej zapletena zadeva; gl. npr. Unicode Standard Annex #29 in nalogo 2008.X.23 na str. 44.

Pri računanju povprečij je koristno biti pazljiv še na to, da se izognemo deljenju z 0, če bi nam slučajno kdo podtaknil prazno vhodno besedilo.

2. Brez ene črke

Imejmo dve tabeli; v eni, imaCrko, bo za vsako črko pisalo, ali je prisotna v trenutnem stavku ali ne; v drugi, najBrez, pa bomo za vsako črko hranili dolžino najdaljšega doslej najdenega stavka, ki ni vseboval te črke. Vhodno besedilo lahko prebiramo znak z znakom. Če je trenutni znak črka, postavimo ustrezni element tabele imaCrko na **true** in povečamo števec črk v trenutnem stavku (dolzina). Ko pa pridemo do konca vrstice (znak `'\n'` ali pa celo konec vhodnih podatkov sploh, EOF), za vsako črko pogledamo, ali je bila prisotna v trenutnem stavku; če ni bila in če je to najdaljši tak stavek doslej, vpišemo njegovo dolžino v tabelo najBrez. Ob tem še postavimo vse vrednosti v imaCrko na **false** in spremenljivko dolzina nazaj na 0, tako da smo pripravljeni na obdelavo naslednjega stavka. Na koncu moramo le še izpisati vsebino tabele najBrez.[7]

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

int main()
{
  int najBrez[26], dolzina = 0, i, c;
  bool imaCrko[26];
  for (i = 0; i < 26; i++) najBrez[i] = 0, imaCrko[i] = false;
  do {
    c = fgetc(stdin);
    if ('A' <= c && c <= 'Z') dolzina++, imaCrko[c - 'A'] = true;
    else if ('a' <= c && c <= 'z') dolzina++, imaCrko[c - 'a'] = true;
    else if (c == EOF || c == '\n') {
      for (i = 0; i < 26; i++) {
        if (imaCrko[i]) imaCrko[i] = false;
        else if (dolzina > najBrez[i]) najBrez[i] = dolzina; }
      dolzina = 0; }
  }
  while (c != EOF);
  for (i = 0; i < 26; i++)
    printf("Najdaljši stavek brez %c je dolg %d znakov.\n", 'A' + i, najBrez[i]);
  return 0;
}
```

3. Izpis HTMLja

Vhodno besedilo bomo brali znak po znak in sproti izpisovali tiste dele, ki jih zahteva naloga. V spremenljivki body hranimo podatek o tem, ali se trenutno nahajamo med oznakama `<body>` in `</body>` ali ne; ko naletimo na `<body>`, povečamo števec body za 1, pri `</body>` pa ga zmanjšamo za 1. Tako smo pripravljeni celo na primer, če bi bilo več parov `<body> </body>` vgnezdenih eden v drugem, čeprav se v HTMLju to načeloma ne sme zgoditi.

[7] Več o besedilih, ki se na razne načine izogibajo uporabi posameznih črk, najdemo na primer v Wikipediji *s. v.* Constrained Writing.

Ko torej preberemo nek znak, moramo preveriti, če je `body` > 0, in če je, ga moramo izpisati. Posebej pa moramo obravnavati primer, ko naletimo na znak `<`, ki pomeni začetek neke oznake. Oznake ne izpisujemo, pač pa jo preberemo v niz `tag`; ko pridemo do konca oznake (znak `>`), pogledamo, katera oznaka je to bila in po potrebi nanjo odreagiramo: pri `<body>` in `</body>` spremenimo števec `body`, pri `<br>` pa izpišemo znak za konec vrstice. Pravzaprav si oznake ni treba zapomniti v celoti, dovolj je že prvih 7 znakov — že to je dovolj, da bomo lahko na koncu oznake ugotovili, ali je to kakšna od tistih treh oznak, ki nas zanimajo.

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

int main()
{
  enum { MaxTagLen = 7 };
  char tag[MaxTagLen + 1];
  int body = 0, tagLen, c;
  while ((c = fgetc(stdin)) != EOF)
  {
    if (c != '<') {
      if (body > 0) fputc(c, stdout);
      continue; }
    /* Smo na začetku oznake; preberimo preostanek. */
    tagLen = 0; tag[tagLen++] = c;
    while ((c = fgetc(stdin)) != EOF)
    {
      if (tagLen < MaxTagLen) tag[tagLen++] = tolower(c);
      if (c == '>') break;
    }
    tag[tagLen] = 0;
    if (strcmp(tag, "<body>") == 0) body++;
    else if (strcmp(tag, "</body>") == 0) body--;
    else if (strcmp(tag, "<br>") == 0 && body > 0) fputc('\n', stdout);
  }
  return 0;
}
```

4. Volitve

Imejmo tabelo `glasovi`, v kateri za vsako zvezdo in vsakega kandidata hranimo podatek o tem, koliko planetov, ki krožijo okoli te zvezde, je glasovalo za tega kandidata. Na začetku postavimo vse elemente te tabele na 0, med branjem vhodnih podatkov pa za vsak prejeti glas povečajmo ustrezni element tabele za 1. Nato moramo za vsako zvezdo ugotoviti, katerega kandidata je podprla. V ta namen pregledamo vse kandidate in si v spremenljivki `naj` hranimo številko tistega, ki je med doslej pregledanimi kandidati dobil največ glasov od planetov te zvezde. Poleg tega je pomemben podatek še, ali je ta kandidat edini s toliko glasovi ali pa si deli prvo mesto s še kakšnim drugim kandidatom. Ko pregledamo vse kandidate, dobi podporo zvezde tisti z največ glasovi, vendar le, če je edini s toliko glasovi. Podatek o tem, koliko zvezd podpira posameznega kandidata, vzdržujemo v tabeli `podpora`. Na koncu se moramo le še sprehoditi po tej tabeli in poiskati kandidata, ki je dobil podporo več kot polovice vseh zvezd; ta je (če sploh obstaja) zmagovalec naših volitev.

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

#define MaxKand 10
#define MaxZvezd 1000

int main()
{
  int glasovi[MaxZvezd][MaxKand], podpora[MaxKand];
  int stKand, stZvezd, stPlanetov, k, p, z, naj, zmagovalec;
  bool edini;
  scanf("%d %d %d", &stKand, &stZvezd, &stPlanetov);

  /* Inicializirajmo tabelo glasovi. */
  for (z = 0; z < stZvezd; z++)
    for (k = 0; k < stKand; k++) glasovi[z][k] = 0;

  /* Preberimo glasove in jih štejmo v tabeli glasovi. */
  for (p = 0; p < stPlanetov; p++) {
    scanf("%d %d", &z, &k); glasovi[z - 1][k - 1]++; }

  /* Za vsako zvezdo poglejmo, katerega kandidata podpira. */
  for (k = 0; k < stKand; k++) podpora[k] = 0;
  for (z = 0; z < stZvezd; z++) {
    /* Spremenljivka „naj“ hrani kandidata z največ glasovi doslej,
       „edini“ pa nam pove, ali je to edini kandidat s toliko glasovi. */
    naj = 0; edini = true;
    for (k = 1; k < stKand; k++)
      if (glasovi[z][k] > glasovi[z][naj]) naj = k, edini = true;
      else if (glasovi[z][k] == glasovi[z][naj]) edini = false;
    if (edini) podpora[naj]++; }

  /* Zmagovalec volitev je kandidat, ki ga podpira več kot polovica zvezd. */
  zmagovalec = 0;
  for (k = 0; k < stKand; k++)
    if (podpora[k] > stZvezd - podpora[k]) zmagovalec = k + 1;
  printf("%d\n", zmagovalec); return 0;
}
```

Mimogrede, sistem glasovanja, kot ga opisuje naša naloga, ni čisto izmišljen; približno takšen sistem predvideva ustava ZDA za volitve predsednika, če bi bilo število elektorskih glasov izenačeno in bi moral o predsedniku odločati kongres. Namesto zvezd si moramo misliti posamezne države znotraj ZDA, namesto planetov pa kongresna okrožja znotraj držav.

5. Polinomi

Za začetek preberimo vhodne podatke v tabele — `a1` in `b1` za prvi polinom ter `a2` in `b2` za drugega. Še dve tabeli, `a3` in `b3`, pa imejmo pripravljeni za rezultat (vsoto ali razliko vhodnih polinomov). Pojdimo v zanki po prvem polinomu in za vsak člen poglejmo, ali obstaja v drugem člen z isto stopnjo (`b2[j] == b1[i]`); če obstaja, izračunamo vsoto ali razliko njunih koeficientov, sicer pa ostanemo le pri koeficientu iz prvega polinoma. Tako dobimo člen, ki ga moramo shraniti v izhodni polinom, razen če je ob seštevanju ali odštevanju nastal koeficient 0 (naloga pravi, naj takih členov ne izpisujemo). Členom drugega polinoma, ki smo jih na ta način že upoštevali pri izračunu, postavimo `b2[j]` na -1 in jih s tem označimo kot neveljavne

(predpostavili smo, da se v naših vhodnih podatkih pojavljajo le členi z nenegativno stopnjo). Na koncu tako ostanejo v drugem polinomu le še taki členi, za katere velja, da se v prvem polinomu ni pojavil noben člen z enako stopnjo. Te preostale člene lahko prenesemo naravnost v izhodni polinom (pri tem jih še pomnožimo z -1, če računamo razliko namesto vsote). Sledi le še zanka, s katero izhodni polinom (ki ga imamo zdaj v tabelah `a3` in `b3`) izpišemo.

```
#include <stdio.h>
#define MaxClenov 20

int main()
{
  int a1[MaxClenov], a2[MaxClenov], a3[2 * MaxClenov];
  int b1[MaxClenov], b2[MaxClenov], b3[2 * MaxClenov];
  /* Preberimo vhodne podatke. */
  int n1, n2, n3 = 0, i, j, a; char op; scanf("%c %d %d", &op, &n1, &n2);
  for (i = 0; i < n1; i++) scanf("%d %d", &a1[i], &b1[i]);
  for (j = 0; j < n2; j++) scanf("%d %d", &a2[j], &b2[j]);
  /* Za vsak člen prvega polinoma poglejmo, če obstaja člen z isto stopnjo tudi v
     drugem polinomu; v tem primeru izračunajmo vsoto ali razliko njunih koeficientov. */
  for (i = 0; i < n1; i++) {
    a = a1[i];
    for (j = 0; j < n2; j++)
      if (b2[j] == b1[i]) a += (op == '+' ? 1 : -1) * a2[j], b2[j] = -1;
    if (a != 0) { a3[n3] = a; b3[n3] = b1[i]; n3++; } }
  /* Dodajmo še člene, ki so prisotni le v drugem polinomu, v prvem pa ne. */
  for (j = 0; j < n2; j++) if (a2[j] != 0 && b2[j] >= 0) {
    a3[n3] = (op == '+' ? 1 : -1) * a2[j]; b3[n3] = b2[j]; n3++; }
  /* Izpišimo rezultat. */
  printf("%d\n", n3); for (i = 0; i < n3; i++) printf("%d %d ", a3[i], b3[i]);
  printf("\n"); return 0;
}
```

Ena od slabosti te rešitve je, da členov izhodnega polinoma ne izpiše nujno urejenih po stopnji (niti naraščajoče niti padajoče); res pa je, da naloga tega ne zahteva eksplicitno.

Če bi se smeli zanesti na to, da bodo členi v vhodnih podatkih urejeni padajoče po stopnji, bi lahko nalogo rešili še bolj elegantno, z zlivanjem: hkrati se sprehodimo po obeh polinomih; če vidimo v obeh člen z isto stopnjo, izračunajmo vsoto ali razliko koeficientov pri teh členih in se premaknimo naprej po obeh polinomih; drugače pa se premaknimo naprej le po tistem polinomu, ki je imel na trenutnem položaju člen z višjo stopnjo (ta člen ob tem tudi prenesemo v izhodni polinom). Posebej je treba paziti na primere, ko smo pri enem polinomu že prišli do konca, pri drugem pa še ne. Namesto dveh gnezdenih zank (po `i` in `j`) iz zgornje rešitve imamo zdaj le eno zanko, pri kateri se povečujeta oba števca:

```
  for (i = 0, j = 0; i < n1 || j < n2; n3++) {
    /* Naslednji spremenljivki povesta, po katerem polinomu se bomo premaknili naprej. */
    bool premik1 = false, premik2 = false;
    /* Če smo že na koncu enega polinoma, se premaknemo le po drugem. */
    if (i >= n1) premik2 = true; else if (j >= n2) premik1 = true;
    /* Če smo pri obeh na členu z isto stopnjo, se premaknemo po obeh. */
```

```
    else if (b1[i] == b2[j]) premik1 = true, premik2 = true;
    /* Sicer se premaknemo po tistem, ki ima trenutno člen z višjo stopnjo. */
    else if (b1[i] > b2[j]) premik1 = true; else premik2 = true;
    /* Izračunajmo novi člen izhodnega polinoma. */
    a3[n3] = 0;
    if (premik1) { a3[n3] = a1[i]; b3[n3] = b1[i]; i++; }
    if (premik2) { a3[n3] += (op == '+' ? 1 : -1) * a2[j]; b3[n3] = b2[j]; j++; } }
```

Lepo pri tej rešitvi je, da ima časovno zahtevnost le $O(n_1 + n_2)$ namesto $O(n_1 \cdot n_2)$, poleg tega pa ima pri njej tudi rezultat (izhodni polinom v tabelah `a3` in `b3`) člene lepo urejene po stopnji. Če ne bi imeli pri tej nalogi opravka s tako kratkimi polinomi (besedilo naloge pravi, da imajo vhodni polinomi po največ 20 členov), bi se splačalo celo najprej urediti člene vhodnih polinomov po stopnji (če še niso urejeni) in potem uporabiti postopek z zlivanjem.

6. Kalkulator

Spodnji program hrani vsebino vmesnega pomnilnika v spremenljivki `vsota`; spremenljivka `stevilo` pa hrani število, ki ga uporabnik trenutno tipka. Po vnosu nove števke prikažemo `stevilo`, po pritisku na + pa izračunamo novo vsoto in jo prikažemo. Če vsota preseže 6 znakov, jo postavimo na 999999; podobno pa pri vnosu števila upoštevamo le prvih 6 števk, nadaljnje pa ignoriramo.

```
int main()
{
  int vsota = 0, stevilo = 0; char c;
  PrikaziNaZaslonu(stevilo);
  while (true)
  {
    c = PocakajTipko();
    if (c == 'C') {
      vsota = 0; stevilo = 0;
      PrikaziNaZaslonu(stevilo); }
    else if (c == '+') {
      vsota += stevilo; stevilo = 0;
      if (vsota > 999999) vsota = 999999;
      PrikaziNaZaslonu(vsota); }
    else {
      if (stevilo <= 99999) stevilo = stevilo * 10 + (c - '0');
      PrikaziNaZaslonu(stevilo); }
  }
}
```

Besedilo naloge je malo nejasno glede tega, kaj naj se zgodi, če uporabnik večkrat zapored pritisne +, ne da bi vmes pritisnil kakšno števko. Naša rešitev v tem primeru pusti tako vsoto kot prikaz na zaslonu nespremenjen (enak kot po prvem pritisku na +).

7. Tečajnica

Ker se vsaka operacija (dodajanje ali povpraševanja) nanaša le na eno delnico in je čisto neodvisna od vseh drugih delnic, je smiselno imeti za vsako delnico ločeno podatkovno strukturo. Te strukture lahko potem povežemo v razpršeno tabelo (*hash*

table), v kateri so ključi kar naši nizi z oznakami delnic, pripadajoča vrednost k posameznemu ključu pa je podatkovna struktura za tisto delnico.

Tako nam ostane le še vprašanje, kakšna je primerna podatkovna struktura za vodenje podatkov o posamezni delnici. Primerna možnost je kar navadna tabela (*array*), v kateri naj bodo zapisi urejeni naraščajoče po času. Dodajati bomo morali le na konec tabele, tako da je dodajanje hitro in učinkovito (ko se tabela napolni, lahko alociramo novo, dvakrat večjo, in prenesemo podatke vanjo; tako je povprečna cena enega dodajanja še vedno le $O(1)$). Ker je tabela urejena kronološko, lahko podprogram CenaDelnice po njej išče z bisekcijo, ko mora ugotoviti ceno delnice na nek dan; iskanje bo torej vzelo $O(\log n)$ časa, če imamo v tabeli n zapisov. Bisekcija ponavadi deluje tako, da nam najde npr. zadnji zapis, ki ima datum začetka manjši ali enak od datuma, ki nas zanima (Kdaj v deklaraciji podprograma CenaDelnice), tako da moramo po bisekciji še preveriti, ali tako dobljeni zapis res pokriva naš poizvedovalni datum (ali pa se konča že pred njim, kar pomeni, da je naš poizvedovalni datum padel v eno od časovnih vrzeli, za katere v tečajnici nimamo cene te delnice).

```
if (t.size() == 0 || kdaj < t[0].zacetek) return −1;
int L = 0, D = t.size();
while (D − L > 1) {
  // Invarianta: t[L].zacetek <= kdaj < t[D].zacetek.
  int mid = (L + D) / 2;
  if (t[mid].zacetek <= kdaj) L = mid; else D = mid; }
// Zdaj torej velja: t[L].zacetek <= kdaj < t[L + 1].zacetek
if (kdaj <= t[L].konec) return L; else return −1;
```

8. Okvarjen pomnilnik

(a) Če je okvarjena naslovna linija t in poskusimo nekaj zapisati v celico $M[2^t]$, se bo sprememba namesto v tej celici poznala v celici $M[0]$ (kajti v dvojiškem zapisu števila 2^t je prižgan samo bit t in če je naslovna linija t zataknjena na vrednosti 0, bo naš pomnilniški čip ta naslov videl kot 0). Lahko se torej sprehodimo v zanki po t, vsakič postavimo $M[0]$ na neko znano vrednost in nato poskusimo postaviti $M[2^t]$ na neko drugo vrednost; če se je $M[0]$ pri tem kaj spremenila, vemo, da je bila linija t okvarjena.

```
#define n ...
typedef unsigned char PomnilnikT[1 << n];

void Preveri(PomnilnikT M)
{
  int linija; unsigned char c;
  for (linija = 0; linija < n; linija++)
  {
    M[0] = 0;
    M[1 << linija] = 255;
    if (M[0] != 0) printf("Naslovna linija %d je okvarjena.\n", linija);
  }
}
```

(b) Primer, ko je okvarjenih več naslovnih linij, ne le ena, reši že naša rešitev iz točke (a). Okvare na podatkovnih linijah pa najlažje odkrijemo tako, da poskusimo nekaj

narediti s celico $M[0]$. Če vanjo vpišemo 255 (vrednost, v kateri so vsi biti prižgani) in nato to celico spet preberemo, nam bodo ugasnjeni biti v prebrani vrednosti povedali, katere podatkovne linije so bile okvarjene.

```
M[0] = 255;
c = M[0];
for (linija = 0; linija < 8; linija++)
  if ((c & (1 << linija)) == 0)
    printf("Podatkovna linija %d je okvarjena.\n", linija);
```

Če so okvarjene vse podatkovne linije, pa ne bomo mogli testirati morebitnih okvar na naslovnih linijah, saj nam pomnilnik ob vsakem branju vrne 0, ne glede na to, iz katere celice beremo in kakšna je bila njena prava vrednost.

(c) Podatkovne linije, ki so zataknjene na 0, lahko odkrijemo enako kot zgoraj pri (b); tiste, ki so zataknjene na 1, pa odkrijemo tako, da v $M[0]$ vpišemo 0, nato vrednost $M[0]$ preberemo in pogledamo, če je v dobljeni vrednosti kakšen bit prižgan.

S testiranjem naslovnih linij pa je tako: če se dva naslova, recimo a_0 in a_1, razlikujeta le v enem bitu in je pripadajoča naslovna linija okvarjena, bomo pri dostopu do $M[a_0]$ in $M[a_1]$ dostopali obakrat do iste pomnilniške celice, vendar pa ne vemo, do katere. Zato lahko ugotovimo, katere naslovne linije so okvarjene, ne pa tudi, ali so zataknjene na vrednosti 0 ali na 1.

Pri testiranju naslovnih linij moramo biti zdaj pozorni na naslednje. Pri (a) smo v celico $M[0]$ vpisali 0, v celico $M[2^t]$ pa 255 in nato z branjem preverili, ali je $M[0]$ še vedno 0 ali ne. Zdaj pa, ko so lahko nekatere podatkovne linije zataknjene na vrednost 1, ne smemo pričakovati, da bomo z branjem iz $M[0]$ dobili 0; če je z naslovno linijo t vse v redu, bomo iz $M[0]$ prebrali vrednost, v kateri so prižgani natanko tisti biti, katerih pripadajoče podatkovne linije so zataknjene na vrednosti 1.

```
void Preveri(PomnilnikT M)
{
  unsigned char d0, d1; int linija;
  M[0] = 255; d0 = M[0] ^ 255;
  M[0] = 0; d1 = M[0];
  for (linija = 0; linija < 8; linija++) {
    if ((d0 >> linija) & 1) printf("Pod. linija %d je zataknjena na 0.\n", linija);
    if ((d1 >> linija) & 1) printf("Pod. linija %d je zataknjena na 1.\n", linija); }
  assert((d0 & d1) == 0); /* sicer so okvare nekonsistentne */
  if ((d0 | d1) == 255) {
    printf("Vse podatkovne linije so okvarjene, "
         "zato naslovnih ne moremo testirati.\n"); return; }
  for (linija = 0; linija < n; linija++) {
    M[0] = 0; /* v celico se zares vpiše vrednost d1 */
    M[1 << linija] = 255; /* v celico se zares vpiše vrednost d0 ^ 255 */
    /* Če je ta naslovna linija okvarjena, sta oba gornja
       stavka spreminjala isto celico. */
    assert(M[0] == d1 || M[0] == (d0 ^ 255)); /* sicer so okvare nekonsistentne */
    if (M[0] != d1) printf("Naslovna linija %d je okvarjena.\n", linija); }
}
```

9. Tiskalnik

Najprej deklarirajmo nekaj globalnih spremenljivk, v katerih bomo hranili število dijakov, skupno število natisnjenih strani in razpoložljivo kvoto vsakega dijaka:

```
#define MaxDijakov 1000
int stDijakov, kvota[MaxDijakov], stNatisnjenih;
```

Nekateri od podprogramov, ki jih moramo napisati, ne delajo drugega, kot da vračajo ali spreminjajo vrednosti teh globalnih spremenljivk:

```
int NatisnjenihStrani() { return stNatisnjenih; }
int KvotaDijaka(int IdDijaka) { return kvota[IdDijaka − 1]; }
void PovisajKvoto(int IdDijaka, int StStrani) { kvota[IdDijaka − 1] += StStrani; }
```

Na začetku šolskega leta postavimo stNatisnjenih na 0, si zapomnimo število dijakov in postavimo njihove kvote na SteviloStrani:

```
void ZacniSolskoLeto(int SteviloDijakov, int SteviloStrani)
{
  int d; stDijakov = SteviloDijakov; stNatisnjenih = 0;
  for (d = 0; d < stDijakov; d++) kvota[d] = SteviloStrani;
}
```

Najbolj zanimiva pa je funkcija NatisniDokument. Če je dijaku ostalo premalo kvote, tiskanje zavrnemo; drugače pa kvoto zmanjšamo za dolžino dokumenta in dokument pošljemo tiskalniku. Če se uspešno natisne, povečamo globalno spremenljivko stNatisnjenih in vrnemo število strani v dokumentu. Če pa je pri tiskanju prišlo do napake, dijakovo kvoto povečamo nazaj in vrnemo −1.

```
int NatisniDokument(int IdDijaka, int IdDokumenta, int StStrani)
{
  if (kvota[IdDijaka − 1] < StStrani) return −1;
  kvota[IdDijaka − 1] −= StStrani;
  if (! PosljiTiskalniku(IdDokumenta)) { kvota[IdDijaka − 1] += StStrani; return −1; }
  else { stNatisnjenih += StStrani; return StStrani; }
}
```

Če bi kvoto zmanjšali šele po uspešno končanem tiskanju, bi lahko prišlo do težav, če dobimo od istega dijaka hkrati več zahtevkov za tiskanje: če je kvota na primer 13 strani in pridejo trije zahtevki po 10 strani, bi na začetku vsi videli kvoto 13, začeli s tiskanjem in na koncu zmanjšali kvoto vsak za 10, tako da bi bila nazadnje enaka −17.

Naša sedanja rešitev sicer še vedno ni povsem odporna na težave pri več hkratnih zahtevkih (še vedno se lahko zgodi, da med trenutkom, ko preverimo kvoto, in trenutkom, ko jo zmanjšamo, nek drug zahtevek preveri kvoto istega dijaka, kar lahko pripelje do prav take napake kot v prejšnjem odstavku); za res zanesljivo rešitev bi morali poskrbeti, da enega zahtevka med branjem in spreminjanjem kvote ne bo mogel prekiniti noben drug zahtevek. Pri tem bi si lahko pomagali z mehanizmi za sinhronizacijo, ki so na voljo v mnogih operacijskih sistemih in programskih jezikih (npr. **synchronized** v javi).

Napake pri tiskanju (če funkcija PosljiTiskalniku vrne **false**) si naša rešitev razlaga tako, kot da ne bi bilo natisnjeno nič, saj ne more vedeti, ali je kakšna stran vendarle bila natisnjena (in koliko).

10. Pobiranje smeti

Omejitve iz besedila naloge robotu pravzaprav ne puščajo veliko izbire glede tega, v kakšnem vrstnem redu mora pobirati smeti. Izbira nastopi le v primerih, ko imamo dve smeti na enaki oddaljenosti od koordinatnega izhodišča, recimo $+x$ in $-x$; takrat si lahko robot sam izbere, ali bo najprej pobral $+x$ in nato $-x$ ali obratno.

Recimo, da smeti oštevilčimo po naraščajoči oddaljenosti od koordinatnega izhodišča kot $a_1, a_2, \ldots, a_n$. Naj bo $f(x)$ dolžina najkrajše veljavne poti robota, pri kateri pobere vse smeti, ki so od izhodišča oddaljene za največ $|x|$, in konča v točki x. Rezultat, po katerem sprašuje naša naloga, je potemtakem bodisi $f(a_n) + |a_n|$ (ker se moramo po pobiranju zadnje smeti vrniti nazaj v izhodišče) bodisi $f(-a_n) + |a_n|$, pri čemer pride slednja možnost v poštev le, če imamo smet tudi na točki $-a_n$. (V tem primeru je seveda $a_{n-1} = -a_n$.)

Dovolj bo, če bomo znali računati f le za tiste x, na katerih leži kakšna smet, saj ni nobene koristi od tega, da bi robot spreminjal smer gibanja še kje drugje kot v točkah, kjer je pravkar pobral kakšno smet. Naj bo torej x neka smet in naj bo y tista smet, ki je najdlje od izhodišča med vsemi, ki so od izhodišča oddaljene za manj kot $|x|$. Ločimo nekaj možnosti:

- če ne obstajata niti smet $-x$ niti $-y$, potem je $f(x) = f(y) + d(x, y)$;
- če obstaja smet $-y$, ne pa $-x$, potem je $f(x) = \min\{f(y) + d(y, x), f(-y) + d(-y, x)\}$;
- če obstaja smet $-x$, ne pa y, potem je $f(x) = f(y) + d(y, -x) + d(-x, x)$;
- če obstajata tako smet $-y$ kot $-x$, potem je $f(x) = \min\{f(y) + d(y, -x) + d(-x, x), f(-y) + d(-y, -x) + d(-x, x)\}$.

(Pri tem smo pisali $d(u, v)$ za dolžino poti od u do v; torej $d(u, v) = |u - v|$.) Z drugimi besedami, če obstajata tako smet y kot $-y$, si lahko izberemo, katero bomo obiskali prej in katero kasneje; če pa obstajata tako $-x$ kot x, nas definicija funkcije f sili, da (ko računamo $f(x)$) obiščemo najprej $-x$ in nato x (ko bomo računali $f(-x)$, pa bo seveda ravno obratno).

S temi formulami lahko izračunamo $f(x)$ za vse položaje x, kjer leži kakšna smet, po naraščajoči oddaljenosti od izhodišča. Opazimo lahko, da ko računamo $f(x)$ in $f(-x)$, potrebujemo pri tem vrednosti $f(y)$ in $f(-y)$ (slednjo le, če na $-y$ res leži kakšna smet), ne pa tudi vrednosti f za smeti, ki so od koordinatnega izhodišča oddaljene za manj kot $|y|$; stare vrednosti f lahko torej sproti pozabljamo, tako da nam ni treba alocirati nove tabele z n elementi za vse vrednosti funkcije f. Spodnja rešitev v vsaki iteraciji glavne zanke izračuna f za eno ali dve zaporedni smeti (dve takrat, če sta enako oddaljeni od koordinatnega izhodišča), pri tem pa potrebuje le rezultate iz prejšnje iteracije (ki si jih zapomni v spremenljivkah f1 in f2), zgodnejših pa ne. Ta podprogram tudi predpostavlja, da so smeti v tabeli a že podane po naraščajoči oddaljenosti od izhodišča.

```
int d(int x, int y) { return abs(x − y); }
int min(int x, int y) { return x < y ? x : y; }

int NajkrajsaPot(int *a, int n)
{
  int f1 = 0, f2 = 0;
  for (int k = 0; k < n; k)
  {
    int x = a[k];
    int y = 0, fy = f2, fmy = f1;
    if (k > 0) y = a[k − 1];
    bool minusY = (k > 1 && a[k − 2] == −y);
    /* Zdaj imamo v fy vrednost f(y), v fmy pa f(−y).
       Izračunajmo f(−x) in f(x) in ju shranimo v f1 in f2. */
    if (k + 1 < n && a[k + 1] == −a[k])
    {
      if (minusY) {
        f1 = min(fy + d(y, −x) + d(−x, x), fmy + d(y, −x) + d(−x, x));
        f2 = min(fy + d(y, x) + d(x, −x), fmy + d(y, x) + d(x, −x)); }
      else {
        f1 = fy + d(y, −x) + d(−x, x);
        f2 = fy + d(y, x) + d(x, −x); }
      k += 2;
    }
    else
    {
      if (minusY) f2 = min(fy + d(y, x), fmy + d(−y, x));
      else        f2 = fy + d(y, x);
      k += 1;
    }
  }
  if (n <= 0) return 0;
  int naj = f2;
  if (n > 1 && a[n − 2] == −a[n − 1])
    if (f1 < naj) naj = f1;
  return naj + abs(a[n − 1]);
}
```

Zelo elegantno pa lahko do optimalne poti pridemo tudi z naslednjim razmislekom. Recimo, da je od izhodišča najbolj oddaljena smet a_n in da na točki $-a_n$ smeti ni. Robot bo torej pot moral končati v a_n (in se od tam vrniti v izhodišče). Druga najbolj oddaljena smet je a_{n-1}.

(1) Če na točki $-a_{n-1}$ ni smeti, bo zadnja smet, ki jo bo robot pobral, preden bo šel na a_n, morala biti smet a_{n-1} — druge možnosti ni.

Če pa je smet tudi na točki $-a_{n-1}$, lahko razmišljamo takole.

(2) Recimo, da sta a_n in a_{n-1} enako predznačeni. Robot bo vsekakor moral nekje na svoji poti obiskati $-a_{n-1}$ in to še prej, preden bo šel do a_n. Toda od $-a_{n-1}$ do a_n se ne da priti drugače kot tako, da gremo pri tem tudi mimo a_{n-1}. Torej ne more biti nobene koristi od tega, da bi robot pobral a_{n-1} pred $-a_{n-1}$, ker bo lahko a_{n-1} pobral spotoma na svoji poti od $-a_{n-1}$ do a_n.

(3) Če bi bili a_n in a_{n-1} nasprotno predznačeni, bi z analognim razmislekom prišli do tega, da ni nobene koristi od tega, da bi robot pobral $-a_{n-1}$ pred a_{n-1}.

Z eno od opisanih treh možnosti torej lahko rekonstruiramo zadnji del poti robota

in vemo, da se pred tem zadnjim delom nahaja pri eni od smeti a_{n-1} in $-a_{n-1}$ (in tudi vemo, kateri) in da je pred tem že pobral vse smeti, ki so od izhodišča oddaljene manj kot $|a_{n-1}|$. To pa je zdaj problem iste oblike kot na začetku, le robot je malo bližje izhodišču in smeti je nekaj manj. Preostanek poti torej lahko rekonstruiramo s ponavljanjem prav takega razmisleka.

Na začetku smo predpostavili, da na točki $-a_n$ ni smeti; če je, lahko najboljšo rešitev poiščemo tako, da gornji postopek ponovimo dvakrat: enkrat se pretvarjamo, da smeti $-a_n$ ni, drugič pa, da ni smeti a_n. Vsako od dobljenih poti potem še podaljšamo tako, da robot pobere smet, ki smo jo prej odmislili, in se vrne v izhodišče (to podaljšanje pomeni, da mora iti od $\pm a_n$ do $\mp a_n$ in od tam do 0, tako da je skupna dolžina tega podaljšanja v vsakem primeru $3|a_n|$). Od obeh tako dobljenih poti izberemo najkrajšo.

```
/* Predpostavka: tabela a je urejena naraščajoče po absolutni vrednosti
   in vsa števila v njej so različna. */
int NajkrajsaPot(int *a, int n)
{
  int kand1, kand2, pot, k;
  if (n == 0) return 0;
  if (n > 1 && a[n - 1] == -a[n - 2])
  {
    kand1 = NajkrajsaPot(a, n - 1);
    a[n - 2] = -a[n - 2];
    kand2 = NajkrajsaPot(a, n - 1);
    return (kand1 < kand2 ? kand1 : kand2) + 2 * abs(a[n - 2]);
  }
  pot = abs(a[n - 1]);
  for (k = n - 1; k > 0; )
    if (k == 1 || a[k - 1] != -a[k - 2]) {
      pot += abs(a[k - 1] - a[k]);
      k -= 1; }
    else {
      /* Ena od smeti a[k - 2] in a[k - 1] leži na poti med drugo smetjo in a[k].
         Poskrbimo, da bo v a[k - 2] tista druga. */
      if ((a[k - 2] < 0 && a[k] < 0) || (a[k - 2] > 0 && a[k] > 0))
        a[k - 2] = a[k - 1];
      pot += abs(a[k - 2] - a[k]);
      k -= 2; }
  return pot + abs(a[0]);
}
```

11. Koščki in SG-1

Vhodna števila si lahko predstavljamo kot nize in jih označimo $s_1, s_2, \ldots, s_n$. Radi bi jih v nekem primernem vrstnem redu staknili skupaj in to tako, da bi bil dobljeni niz leksikografsko največji. Naloge se lahko lotimo z rekurzijo:[8]

naj bo $s^\star$ globalna spremenljivka, ki vsebuje najboljši
doslej dobljeni zlepek;

[8] Opomba glede notacije: če sta t_1 in t_2 dva niza, pomeni t_1t_2 stik teh dveh nizov; $|t|$ pomeni dolžino niza t (število znakov v njem); ε pa pomeni prazen niz. Če obstaja nek u, tako da je $s = tu$, potem rečemo, da je t *prefiks* niza s, ta pa je *podaljšek* niza t.

```
podprogram Rekurzija(s, S):
  (* Vhod: s je nek zlepek doslej uporabljenih nizov,
     S pa je množica vseh še neuporabljenih nizov. *)
  for each w ∈ S:
    naj bo s' := sw;
    if se s' začne na s*:
      s* := s';
    else if s' > s*:
      (* s' je leksikografsko večji od s* in ni njegov podaljšek; torej ni važno,
         v kakšnem vrstnem redu bi dodali na konec s' še preostale nize iz s*,
         rezultat bi bil vedno večji od s*; ker torej iz s' ne bomo dobili
         najboljše rešitve, rekurzije tu ne bomo nadaljevali. *)
      continue;
    else if s' < s* and s* se ne začne na s':
      s* := s';
    Rekurzija(s', S − {w});

glavni blok programa:
  s* := ε;
  Rekurzija(ε, {s_1, ..., s_n});
  izpiši s*;
```

Podprogram *Rekurzija* stika nize v vseh možnih vrstnih redih, pri tem pa si zapomni najboljši doslej dobljeni zlepek in z rekurzijo preneha, čim ugotovi, da trenutni zlepek ne more voditi do najboljših rešitev (boljših od najboljše doslej znane rešitve).

Očitno je časovna zahtevnost te rešitve precej odvisna od tega, kako kmalu najde res dobre zlepke, ki ji bodo v nadaljevanju omogočili, da se bo čim bolj izognila nepotrebnim rekurzivnim klicem in pregledovanju neobetavnih delov prostora. To pa je odvisno od vrstnega reda, v katerem zanka „**for each** $w \in S$“ pregleduje nize iz množice S. Ali bi se mogoče dalo s primerno izbranim vrstnim redom priti do najboljše rešitve celo kar takoj, brez rekurzije?

Prva stvar, ki nam mogoče pride na misel, je, da bi nize uredili leksikografsko in jih staknili v tem vrstnem redu; toda hitro lahko vidimo, da na ta način ne bomo vedno dobili najboljše rešitve. Primer: $s_1 = 78$, $s_2 = 787$; leksikografski vrstni red nam da $s_1 s_2 = 78787$, boljša rešitev pa je $s_2 s_1 = 78778$.

Pri tem primeru je videti težava v tem, da je bil en niz (s_2) podaljšek drugega (s_1). Mogoče bi morali podaljške postaviti pred njihove prefikse. To bi lahko v praksi implementirali tako, da bi pri urejanju vsakemu nizu na konec pritaknili še nek poseben znak (recimo #), ki bi bil leksikografsko za vsemi znaki, ki se sicer pojavljajo v naših nizih. Tako na primer iz nizov 78, 787, 79 dobimo 78#, 787#, 79# in vrstni red 787#, 78#, 79#. (Pred stikanjem znake # seveda pobrišemo.) Toda tudi za to zamisel hitro najdemo protiprimer: $s_1 = 76$, $s_2 = 767$; naš pravkar opisani vrstni red bi dal rešitev $s_2 s_1 = 76776$, boljša rešitev pa je $s_1 s_2 = 76767$.

V obeh gornjih primerih smo uspeli suboptimalni vrstni red izboljšati tako, da smo zamenjali vrstni red dveh sosednjih nizov. Očitno je, da velja to tudi v splošnem: če imamo nize staknjene v vrstnem redu $s_{\pi(1)} s_{\pi(2)} \ldots s_{\pi(n)}$ in je $s_{\pi(i+1)} s_{\pi(i)} < s_{\pi(i)} s_{\pi(i+1)}$, potem lahko naš vrstni red izboljšamo, če niza $s_{\pi(i)}$ in $s_{\pi(i+1)}$ med seboj

zamenjamo.

Da bo manj pisanja, definirajmo relacijo $\preceq$ takole: $x \preceq y$ natanko tedaj, ko $xy \leq yx$. Zdaj lahko torej začnemo s poljubnim vrstnim redom naših nizov $s_1, \ldots, s_n$ in če opazimo dva zaporedna niza, ki si nista v relaciji $\preceq$, ju lahko med seboj zamenjamo in tako vrstni red izboljšamo. Prej ali slej pridemo torej do vrstnega reda, v katerem je vsak niz v relaciji $\preceq$ s tistim, ki stoji v vrstnem redu takoj za njim. Ali je takšen vrstni red že tudi najboljši možni vrstni red sploh?

Recimo, da naše nize oštevilčimo v takem vrstnem redu, kot smo ga pravkar dobili; torej imamo $s_1 \preceq s_2 \preceq \ldots \preceq s_n$. Pa recimo, da bi obstajal nek še boljši vrstni red, $s_{\pi(1)}, \ldots, s_{\pi(n)}$, za katerega je torej $s_{\pi(1)}s_{\pi(2)}\ldots s_{\pi(n)} < s_1 s_2 \ldots s_n$. V prvih nekaj nizih se mogoče oba vrstna reda ujemata, prej ali slej pa mora nastopiti med njima neka razlika; naj bo torej i najmanjši indeks, pri katerem $\pi(i) \neq i$. Ker se pred tem vrstna reda ujemata, imamo $\pi(1) = 1, \ldots, \pi(i-1) = i-1$. Vrednost i torej v π ne nastopi niti na indeksu i niti pred njim, zato mora v π nastopiti nekje kasneje, na primer na indeksu j; torej imamo $\pi(j) = i$ za nek $j > i$. Na mestih $\pi(i), \ldots, \pi(j-1)$ pa imamo potemtakem vrednosti, večje od i; označimo $k = \pi(j-1)$; ker je torej $k > i$, vemo, da velja $s_i \preceq s_{i+1} \preceq \cdots \preceq s_k$. Upajmo, da je $\preceq$ tranzitivna in da zato sledi $s_i \preceq s_k$. To pomeni, da je $s_i s_k \leq s_k s_i$ oz. $s_{\pi(j)}s_{\pi(j-1)} \leq s_{\pi(j-1)}s_{\pi(j)}$. Če torej v vrstnem redu π zamenjamo j-ti in $(j-1)$-vi niz, se vrstni red nič ne poslabša, mogoče se celo izboljša. Po tej spremembi se element i pojavlja na mestu $\pi(j-1)$, ne več na $\pi(j)$; na tem novem položaju lahko isti razmislek ponovimo in tako premikamo element i korak za korakom proti levi, vse dokler ne pristane na položaju $\pi(i)$. Zaporedje π se zdaj z našim vrstnim redom ujema že v prvih i elementih, ne le v prvih $i-1$ elementih. Zdaj lahko poiščemo naslednje neujemanje in z enakim postopkom odpravimo tudi njega; tako lahko sčasoma predelamo optimalno zaporedje π v naše zaporedje $s_1, \ldots, s_n$, ne da bi se pri tem zaporedje na kakšnem koraku poslabšalo. Torej ni naša rešitev nič slabša od optimalne.

Ta razmislek nam je pokazal, da lahko do optimalne rešitve pridemo tako, da nize uredimo z relacijo $\preceq$, toda le, če je ta relacija tranzitivna. Izkaže se, da je res tranzitivna, vsaj dokler jo gledamo le nad neprazними nizi. Prepričajmo se, da to res drži. Za začetek opazimo, da za $\preceq$ veljajo naslednje lastnosti (dokaze prepuščamo bralcu za vajo):

- Za $x \neq y$ velja $x \preceq y$ natanko tedaj, ko velja nekaj od naslednjega trojega:
  1. obstaja neprazen u, tako da je $x = yu$ in $u \preceq y$;
  2. obstaja neprazen v, tako da je $y = xv$ in $x \preceq v$;
  3. nobeden od x in y ni podaljšek drugega in $x_d < y_d$, če vzamemo $d = \min\{|x|, |y|\}$ in z x_d oz. y_d označimo prvih d znakov niza x oz. y. (Zaradi $x_d < y_d$ velja seveda tudi $x < y$.)
- Za vsaka x, y velja $x \preceq y \iff x \preceq xy \iff yx \preceq y$.
- Za vsake x, y, z velja $x \preceq z \wedge y \preceq z \Rightarrow xy \preceq z$ in $x \preceq y \wedge x \preceq z \Rightarrow x \preceq yz$.

Radi bi dokazali, da je $\preceq$ tranzitivna, torej da za vsako trojico neprazних nizov x, y, z velja $x \preceq y \preceq z \Rightarrow x \preceq z$. Dokazujmo z indukcijo po njihovi skupni dolžini, $|xyz|$.

Baza indukcije je $|xyz| = 3$, ko so nizi dolgi le po en znak. Ker niz dolžine 1 ne more biti podaljšek drugega niza dolžine 1, sledi iz $x \preceq y$ tudi $x < y$, iz $y \preceq z$ pa $y < z$, zato $x < z$, iz tega pa, ker nobeden od njiju ni podaljšek drugega, sledi $x \preceq z$.

Recimo zdaj, da smo tranzitivnost preverili za vse primere, ko je $|xyz| < m$. Poglejmo neko trojico x, y, z, za katero je $|xyz| = m$ in $x \preceq y \preceq z$. Če je $x = y$, sledi $x \preceq z$ iz $y \preceq z$; če je $y = z$, sledi $x \preceq z$ iz $x \preceq y$; če je $x = z$, sledi $x \preceq z$ iz definicije relacije $\preceq$. Ostane nam še primer, ko so x, y in z sami različni nizi. Videli smo, da je $x \preceq y$ možno na naslednje načine:

(1) Lahko je $x = yu$ in $u \preceq y$. Potem je $|u| < |x|$ in zato po induktivni predpostavki iz $u \preceq y \preceq z$ sledi $u \preceq z$, kar nam skupaj z $y \preceq z$ da $yu \preceq z$, torej $x \preceq z$.

(2) Lahko je $y = xv$ in $x \preceq v$. Recimo zdaj, da ne bi veljalo $x \preceq z$; torej je $xz > zx$; torej je $xzv > zxv = zy \geq yz = xvz$ (prva neenakost sledi iz $xz > zx$, druga iz $y \preceq z$), torej $zv > vz$, torej $v \preceq z$, kar nam skupaj z $x \preceq v$ in induktivno predpostavko (ki jo lahko uporabimo, ker je $|v| < |y|$) dá $x \preceq z$, kar je protislovje.

(3) Lahko pa nobeden od x in y ni podaljšek drugega (in velja $x < y$). Zaradi $y \preceq z$ zdaj vemo, da velja nekaj od naslednjega:

(3.1) Lahko je $y = zu$ in $u \preceq z$. Recimo zdaj, da ne bi veljalo $x \preceq z$; torej je $zx < xz$; torej je $zxu < xzu = xy \leq yx = zux$, torej tudi $xu < ux$, torej $x \preceq u$; to nam (ker je $|u| < |y|$) skupaj z $u \preceq z$ po induktivni predpostavki dá $x \preceq z$, kar je protislovje.

(3.2) Lahko je $z = yv$ in $y \preceq v$. Slednje nam (ker je $|v| < |z|$) skupaj z $x \preceq y$ po induktivni predpostavki dá $x \preceq v$, zato pa tudi $x \preceq yv$, torej $x \preceq z$.

(3.3) Lahko pa nobeden od y in z ni podaljšek drugega in velja $y < z$. Zdaj imamo torej $x < y < z$; ker je $x < z$, niz x ne more biti podaljšek z-ja (saj bi bil potem $x > z$); in če bi bil z podaljšek niza x, bi moral biti tudi y podaljšek niza x (saj so leksikografsko med nekim nizom in njegovimi podaljški sami taki nizi, ki so tudi sami njegovi podaljški), kar pa je protislovje. Torej nobeden od x in z ni podaljšek drugega, kar nam skupaj z $x < z$ dá $x \preceq z$. □

Tako torej vidimo, da je $\preceq$ tranzitivna in zato res lahko pridemo do najboljšega vrstnega reda tako, da nize uredimo po relaciji $\preceq$. Pripomnimo lahko še, da $\preceq$ vrstnega reda ne določa nujno enolično: če je $xy = yx$, velja tako $x \preceq y$ kot $y \preceq x$ in naša relacija nam ne pove, katerega naj postavimo v vrstnem redu prej. S tem ni nič narobe, saj iz $xy = yx$ sledi tudi, da je naš zlepek enak ne glede na to, ali postavimo x pred y ali obratno.[9] Bolj matematično bi lahko rekli, da $\preceq$ ni linearna urejenost; je sicer tranzitivna, sovisna in refleksivna, ni pa antisimetrična. Kljub temu jo lahko brez težav uporabimo kot podlago za kakšnega od dobro znanih postopkov urejanja (npr. quicksort, bubble sort ipd.). Če bi hoteli doseči tudi antisimetričnost (in s tem linearno urejenost), bi lahko definicijo popravili takole: $x \preceq' y$ ntk. je $xy < yx$ ali pa $(xy = yx$ in $x \leq y)$.

12. Kolesarjeva pot

Spodnja rešitev predpostavlja, da so meritve v tabeli meritve podane v takem vrstnem redu, kot so bile posnete, torej po naraščajoči vrednosti polja pot. Zato se lahko

[9] Dokazati bi se dalo, da do $xy = yx$ pride natanko v tistih primerih, ko obstaja nek niz u in celi števili k in l, tako da je $x = u^k$ in $y = u^l$.

hkrati sprehajamo po tabeli meritev (s števcem i) in po tabeli točk (z indeksom t). V spremenljivkah pot1 in pot2 hranimo prevoženo pot (od začetka vožnje) za točki tocke[t] in tocke[t + 1]. Če pot naše trenutne meritve ne leži med pot1 in pot2, se premikamo po zaporedju točk naprej, dokler ta pogoj ni izpolnjen (premik nazaj po zaporedju točk pa gotovo ne bo potreben, saj smo predpostavili, da so meritve podane po naraščajoči vrednosti polja pot). Takrat vemo, da je bila trenutna meritev posneta nekje med točkama tocke[t] in tocke[t + 1] in lahko njene koordinate izračunamo iz koordinat teh dveh točk s pomočjo funkcije NajdiTocko.

```
typedef struct { double x, y, z; } TockaT;
typedef struct { double x, y, z, pot, cas, utrip; } MeritevT;

extern double Razdalja(TockaT A, TockaT B);
extern TockaT NajdiTocko(TockaT A, TockaT B, double x);

void ZdruziPodatke(MeritevT *meritve, int stMeritev, TockaT *tocke, int stTock)
{
  double pot1 = 0, pot2 = Razdalja(tocke[0], tocke[1]);
  int t = 0, i; TockaT C;
  for (i = 0; i < stMeritev; i++)
  {
    /* Invarianta: pot1 je prevožena pot od začetka vožnje do tocke[t],
       pot2 pa od začetka vožnje do tocke[t + 1]. Poleg tega je meritve[i].pot ≥ pot1. */
    /* Če je treba, se premaknimo po zaporedju točk naprej, dokler ne
       pridemo tako daleč, da leži trenutna meritev med točkama t in t + 1. */
    while (t + 1 < stTock && pot2 < meritve[i].pot) {
      pot1 = pot2; t++; pot2 += Razdalja(tocke[t], tocke[t + 1]); }
    /* Zdaj lahko izračunamo koordinate meritve in jih shranimo. */
    C = NajdiTocko(tocke[t], tocke[t + 1], meritve[i].pot − pot1);
    meritve[i].x = C.x; meritve[i].y = C.y; meritve[i].z = C.z;
  }
}
```

Opisani podprogram tudi predpostavlja, da imamo podani vsaj dve točki.

13. Igra

Recimo, da je na tabli število x; igralec, ki je zdaj na potezi, lahko razmišlja takole:

- Če je $n < x$, je očitno nasprotnik pravkar zmagal.
- Če je $n/9 < x \leq n$, lahko x pomnožim z 9 in bom dobil rezultat, ki je $> 9 \cdot (n/9) = n$, torej bom zmagal jaz.
- Če je $n/18 < x \leq n/9$, bo po moji potezi, ne glede na to, s čim množim, nastalo število y, za katerega bo veljalo $2 \cdot (n/18) < y \leq 9 \cdot (n/9)$, torej $n/9 < y \leq n$, torej bo prišlo v interval, v katerem bo zmagal nasprotnik.
- Če je $n/162 < x \leq n/18$, lahko množim z 9 in dobim rezultat na intervalu iz prejšnje točke, kar bo nasprotnika prisililo v poraz.
- Če je $n/324 < x \leq n/162$, bo rezultat, ne glede na to, s čim ga množim, prišel v interval iz prejšnje točke, torej bom igro izgubil.

- In tako naprej. V splošnem torej vidimo: če za x velja $n/18^k < x \leq 2n/18^k$, bo zmagal nasprotnik, sicer pa jaz.

Recimo zdaj, da se je igra pravkar začela, na tabli je $x = 1$ in igralec A je prvi na potezi. Vprašanje je torej, ali velja $n/18^k < 1 \leq 2n/18^k$ za neko celo število $k \geq 0$ — to bi bil namreč znak, da bo igralec A igro izgubil. Ta pogoj lahko zapišemo tudi kot $n < 18^k \leq 2n$, iz česar tudi vidimo, kako ga lahko preverjamo:

```
void KdoBoZmagal(int n)
{
  int p = 1;
  while (p <= n) p *= 18;
  if (p <= 2 * n) printf("Zmagal bo drugi igralec.\n");
  else printf("Zmagal bo prvi igralec.\n");
}
```

14. Bézierjeva krivulja

V nalogi je A_{ij} definirana rekurzivno: za izračun A_{ij} potrebujemo $A_{i,j-1}$ in $A_{i+1,j}$, tadva pa dobimo po enakem postopku iz $A_{i,j-2}$, $A_{i+1,j-1}$, $A_{i+2,j}$ in tako naprej. To rekurzivno formulo lahko zelo preprosto prelijemo v rekurziven podpogram:[10]

```
Tocka A[d + 1];

Tocka Bezier1(int i, int j, double t)
{
  if (i == j) return A[i];
  return (1 − t) * Bezier1(i, j − 1, t) + t * Bezier1(i + 1, j, t);
}
```

Slabost te rešitve je, da je neučinkovita. Rekurzija gre tu vedno $j - i$ nivojev globoko in vsakič iz enega klica nastaneta dva klica na naslednjem globljem nivoju. Ker nas v resnici zanima rezultat za $i = 0$, $j = d$, se bo vsega skupaj izvedlo $2^0 + 2^1 + 2^2 + \ldots + 2^d = 2^{d+1} - 1$ klicev naše funkcije.

Rezultat naše funkcije je (pri fiksnem naboru kontrolnih točk A in pri fiksnem t) odvisen le od i in j; zaradi omejitve $0 \leq i \leq j \leq d$ je za (i, j) le $(n + 1)(n + 2)/2$ možnosti, torej je tudi različnih vrednosti, ki bi jih funkcija utegnila vrniti, kvečjemu toliko. Če se izvede $2^{d+1} - 1$ klicev, to pomeni, da se po večkrat računajo točke za ene in iste pare (i, j). Torej lahko prihranimo veliko časa, če si že izračunane rezultate zapomnimo:

```
Tocka temp[d + 1][d + 1];
bool zePoznamo[d + 1][d + 1];

Tocka Bezier2a(int i, int j, double t)
{
  if (i == j) return A[i];
  if (! zePoznamo[i][j]) {
    temp[i][j] = (1 − t) * Bezier2a(i, j − 1, t) + t * Bezier2a(i + 1, j, t);
```

[10] Predpostavili smo, da lahko uporabimo operatorja * in + za množenje vektorjev (struktur tipa Tocka) s skalarji (števili tipa **double**) in za seštevanje takih vektorjev. To bi šlo na primer v C++, v C-ju pa ne in bi morali za te operacije napisati funkcijske podprograme v stilu Tocka Sestej(Tocka x, Tocka y) in Tocka Pomnozi(**double** x, Tocka y).

```
    zePoznamo[i][j] = true; }
  return temp[i][j];
}

Tocka Bezier2(double t)
{
  int i, j;
  for (i = 0; i <= d; i++) for (j = i; j <= d; j++) zePoznamo[i][j] = false;
  return Bezier2a(0, d, t);
}
```

Časovna zahtevnost je zdaj le še $O(d^2)$, žal pa smo zdaj porabili $O(d^2)$ pomnilnika za shranjevanje pomožnih rezultatov (pri prejšnji rešitvi pa le $O(d)$ za hranjenje parametrov vgnezdenih rekurzivnih klicev na skladu).

Do še bolj elegantne rešitve pridemo, če opazimo, da lahko računamo vrednosti A_{ij} bolj sistematično, namreč po naraščajoči vrednosti $j - i$. Pišimo $B_{ki} = A_{i,i+k}$; potem vidimo, da je B_{ki} neposredno odvisen od $B_{k-1,i}$ in $B_{k-1,i+1}$. Ko računamo B_{ki} za vse i pri nekem k, bomo pri tem izračunu potrebovali le $B_{k-1,i}$ za vse i; vse $B_{k',i}$ za $k' < k-1$ pa lahko takrat že pozabimo. Na koncu nas zanima A_{0d}, kar je isto kot B_{d0}.

```
Tocka B[d + 1];

Tocka Bezier3(double t)
{
  int i, k;
  /* Najprej izračunajmo B_ki za k = 0. */
  for (i = 0; i <= d; i++) B[i] = A[i];
  /* Zdaj lahko izračunamo B_ki za večje k. */
  for (k = 1; k <= d; k++) for (i = 0; i + k <= d; i++)
    /* Na tem mestu velja: za j < i vsebuje B[j] vsebuje vrednost B_kj(t),
       za j ≥ i pa vsebuje B[j] vrednost B_{k-1,j}(t). */
    B[i] = (1 - t) * B[i] + t * B[i + 1];
  /* Zdaj B[0] vsebuje B_d0(t), kar je A_0d(t). */
  return B[0];
}
```

Ta rešitev porabi še vedno le $O(d^2)$ časa (verjetno v praksi še manj kot prejšnja, ker nima toliko dela z rekurzivnimi klici), poleg tega pa tudi le $O(d)$ dodatnega pomnilnika.

Mimogrede, funkcijo $B_{d0}(t)$ lahko izrazimo tudi neposredno s kontrolnimi točkami: izkaže se, da je $B_{d0}(t) = \sum_{j=0}^{d} b_{jk}(t)A_j$, pri čemer so b_{jk} Bernsteinovi polinomi: $b_{jk}(t) = \binom{k}{j} t^j (1-t)^{k-j}$. Rekurzivne zveze, kakršne smo uporabljali doslej (in ki so najprikladnejši način za izračun funkcije B_{d0}), pa se imenujejo de Casteljeaujev algoritem.

15. Loto

Med branjem vhodnih podatkov bomo v tabeli izzrebana za vsako številko od 1 do 39 šteli, kolikokrat je bila doslej izžrebana; v tabeli zapored pa bomo šteli, v koliko zadnjih kombinacijah je bila izžrebana. Ko je izžrebana neka številka, recimo i, povečamo njena števca v obeh tabelah za 1; po vsaki prebrani kombinaciji pa postavimo

v tabeli zapored na 0 števce za vsa tista števila, ki niso bila izžrebana v pravkar prebrani kombinaciji. Poleg tega imamo še tabelo maxZapored, ki za vsako številko hrani največjo doslej doseženo vrednost iz tabele zapored.

Ko pridemo do konca vhodnih podatkov, lahko sestavimo zmagovalno kombinacijo po zahtevah naloge: s pomočjo tabele izzrebana poiščemo šest največkrat izžrebanih številk, nato pa s pomočjo tabele maxZapored poiščemo tisto, ki je bila izžrebana največkrat zapovrstjo (naloga zahteva najmanjšo tako — naš program tudi res vrne najmanjšo tako, ker pregleduje številke v naraščajočem vrstnem redu in si zapomni novo številko le, če je bila izžrebana večkrat kot dosedanja). Pri tem si pomagamo s tabelo prejsnja, da ne vključimo v našo zmagovalno kombinacijo iste številke po večkrat.

Naloga je neugodno nejasna glede nekaterih podrobnosti: govori o najpogosteje izžrebanih številkah, ne pove pa, kaj naj naredimo, če si sedem ali celo več številk deli prvo mesto (in so bile vse izžrebane enako pogosto); naš program v tem primeru uporabi najmanjših šest med najpogosteje izžrebanimi številkami.

Sedma številka zmagovalne kombinacije pa naj bi bila najmanjša taka številka, ki je bila izžrebana največkrat zapored; naloga žal ne pove natančno, kaj naj naredimo, če je ta številka hkrati tudi ena od šestih, ki smo jih izbrali v prejšnjem odstavku. Naš program tistih šest številk preprosto preskoči in za sedmo izbere najmanjšo tako, ki je bila izžrebana največkrat zapored in ni prišla med tistih prvih šest po skupnem številu izžrebanj.

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

#define N 39

int main()
{
  int izzrebana[N], zapored[N], maxZapored[N], a[7], i, j; bool prejsnja[N];
  for (i = 0; i < N; i++) prejsnja[i] = false, izzrebana[i] = 0,
                          zapored[i] = 0, maxZapored[i] = 0;
  while (! feof(stdin)) {
    /* Preberimo naslednjo kombinacijo. */
    for (j = 0; j < 7; j++) {
      if (1 != scanf("%d", &a[j])) break;
      a[j]--; izzrebana[a[j]]++; zapored[a[j]]++; }
    if (j < 7) break;
    /* Popravimo tabelo prejsnja, da bo odražala trenutno kombinacijo. */
    for (i = 0; i < N; i++) prejsnja[i] = false;
    for (j = 0; j < 7; j++) prejsnja[a[j]] = true;
    /* Številkam, ki jih v trenutni kombinaciji ni, postavimo zapored[i] na 0.
       Popravimo tabelo maxZapored. */
    for (i = 0; i < N; i++) {
      if (! prejsnja[i]) zapored[i] = 0;
      if (zapored[i] > maxZapored[i]) maxZapored[i] = zapored[i]; } }

  /* Poiščimo šest najpogosteje izžrebanih številk. Tabelo prejsnja
     uporabimo za označevanje tega, katere številke smo že uporabili. */
  for (i = 0; i < N; i++) prejsnja[i] = false;
  for (j = 0; j < 6; j++) {
    a[j] = -1;
```

```
    for (i = 0; i < N; i++) if (! prejsnja[i])
      if (a[j] < 0 || izzrebana[i] > izzrebana[a[j]]) a[j] = i;
    prejsnja[a[j]] = true; }

  /* Poiščimo še najmanjšo številko, ki je bila izžrebana največkrat zapored. */
  a[6] = −1;
  for (i = 0; i < N; i++) if (! prejsnja[i])
    if (a[6] < 0 || maxZapored[i] > maxZapored[a[6]]) a[6] = i;

  /* Izpišimo rezultat. */
  for (j = 0; j < 7; j++) printf("%d ", a[j] + 1);
  return 0;
}
```

16. Poplavljanje labirinta

Naloge se lahko lotimo na primer z iskanjem v globino. Na začetku se postavimo v polje, kjer izvira voda; nato ponavljamo naslednje korake:

1. Če ima trenutno polje kakšnega soseda, ki je prazno in na katerem še nismo bili, se premaknemo tja.

2. Če pa takega polja ni, se vrnemo iz trenutnega polja v tisto polje, iz katerega smo prvič prišli vanj.

Ko se nam primer 2 zgodi pri polju, kjer izvira voda, se lahko ustavimo, saj smo takrat pregledali že vsa polja, ki jih lahko voda doseže.

O vsakem polju moramo torej hraniti podatek o tem, ali smo ga kdaj že obiskali ali ne (da se nam postopek ne bo zaciklal in obiskoval istih polj po večkrat), in o tem, iz katere smeri smo prišli vanj (da se lahko kasneje vrnemo iz njega in s pregledovanjem labirinta nadaljujemo drugod). Spodnji program uporablja za vse to kar isto tabelo, v katero na začetku prebere labirint. Za vsako polje labirinta imamo vrednost tipa **char**, ki je v vhodni datoteki lahko `'*'` (za izvir vode), `'.'` (prazno polje) ali `'X'` (zid). Med pregledovanjem labirinta pa uporabljamo še vrednosti od 0 do 3, da povemo, iz katere smeri smo v neko polje prišli, in vrednost `'~'`, s katero označimo polje za poplavljeno, ko se zadnjič premaknemo iz njega.

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

#define MaxW 100
#define MaxH 100
const int dx[4] = { −1, 1, 0, 0 };
const int dy[4] = { 0, 0, −1, 1 };

int main()
{
  int w, h = 0, x, y, i, xx, yy;
  char a[MaxH][MaxW + 2];
  while (! feof(stdin)) {
    fgets(a[h], MaxW + 2, stdin);
    if (h == 0) { w = strlen(a[h]); if (a[h][w] == '\n') w−−; }
    for (i = 0; i < w; i++) if (a[h][i] == '*') { x = i; y = h; break; }
    h++; }
```

```
  while (true)
  {
    /* Poglejmo, če ima trenutno polje (x, y) kakšnega
       soseda, kamor se voda še lahko razširi. */
    for (i = 0; i < 4; i++) {
      xx = x + dx[i]; yy = y + dy[i];
      if (xx < 0 || xx >= w || yy < 0 || yy >= h) continue;
      if (a[yy][xx] == '.') {
        /* Našli smo primernega soseda; premaknimo se vanj in tja
           tudi zapišimo podatek, iz katere smeri smo vanj prišli. */
        x = xx; y = yy; a[yy][xx] = i; break; }}
    if (i < 4) continue;
    /* Če smo prišli do sem, pomeni, da trenutno polje nima nobenega soseda, kamor se
       lahko voda še razširi. Označimo ga kot poplavljeno in se vrnimo tja, od koder
       smo vanj najprej sploh prišli. Če pa je trenutno polje tisto z izvirom, potem
       vemo, da smo poplavili že vse, kar se je dalo poplaviti, in lahko končamo. */
    i = a[y][x];
    a[y][x] = ' '; /* Označimo trenutno polje kot poplavljeno. */
    if (i == '*') break;
    x -= dx[i]; y -= dy[i];
  }
  for (y = 0; y < h; y++) printf("%s", a[y]);
  return 0;
}
```

17. Prepogibanje papirja

Rešimo najprej podnalogo (b). Če sledimo listu od tistega roba, ki smo ga na začetku držali v levi roki, do nasprotnega roba (ki smo ga na začetku držali v desni roki, vendar je že po prvem prepogibu prišel v levo roko), gredo plasti izmenično v levo in desno (prva, tretja, peta itd. plast gredo v desno, ostale pa v levo). Če list prepognemo navzgor, dobi vsaka desna plast nov upogib tipa `L`, vsaka leva plast pa nov upogib tipa `D`. Če list prepognemo navzgol, je ravno obratno. Po k pregibih imamo 2^k plasti (in zato $2^k - 1$ upogibov, ko list razgrnemo) in zato nastane z naslednjim pregibom na listu 2^k novih upogibov.

$t :=$ prazen niz;
for $k := 1$ **to** n:
  (* i-ti znak niza t označimo s t_i *)
  če je k-ti prepogib gor:
    $t :=$ `L`t_1`D`t_2`L`t_3`D`t_4`L`$\ldots$`L`t_{2^k-1}`D`;
  sicer (če je k-ti prepogib dol):
    $t :=$ `D`t_1`L`t_2`D`t_3`L`t_4`D`$\ldots$`D`t_{2^k-1}`L`;

Na koncu tega postopka vsebuje t ravno niz, ki opisuje končno zaporedje upogibov na listu. Pri $k = 1$ moramo postaviti t na `L`, če imamo prepogib gor, oz. `D`, če imamo prepogib dol.

Zdaj lahko rešimo tudi podnalogo (a), saj moramo ta postopek le obrniti. Če je $t = t_1 t_2 \ldots t_N$ naš vhodni niz, mora biti $N = 2^n - 1$ za neko celo število n; veljati mora $t_1 = t_5 = t_9 = \ldots = t_{N-2}$ in $t_3 = t_7 = t_{11} = \ldots = t_N$ in $t_1 \neq t_3$; če je vse to res, bi lahko t nastal s prepogibom lista, ki ga opisuje krajši niz $t' = t_2 t_4 t_6 \ldots t_{N-1}$ (in sicer bi šlo za prepogib gor, če je $t_1 =$ `L`, in prepogib dol, če je $t_1 =$ `D`). Zdaj

moramo torej po enakem postopku preveriti le še, ali je tudi t' veljavne oblike. Niza t' nam ni treba sestaviti eksplicitno, saj lahko njegove znake sproti prebiramo iz t.

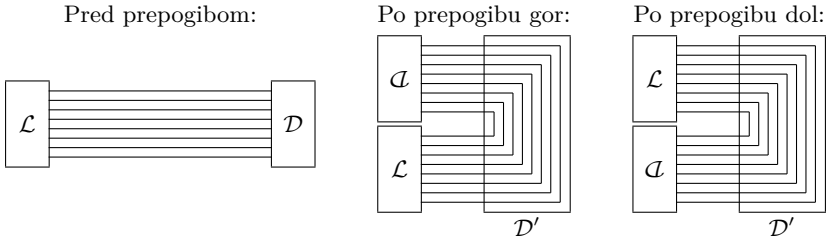
```
(* vhod: niz t = t_1 t_2 ... t_N *)
if N + 1 ni potenca števila 2:
  return; (* niz t ni prave oblike *)
d := 1; p := prazen niz;
while d ≤ N:
  i := d;
  while i + d < N:
    if t_i = t_{i+d}:
      return; (* niz t ni prave oblike *)
    i := i + 2d;
  if t_d = L then p := Gp else p := Dp;
  d := 2d;
```

Na koncu tega postopka je niz p zaporedje `G`-jev in `D`-jev, ki povedo, v kakšnem zaporedju je treba izvajati pregibe gor in dol, da pride naš list papirja v stanje, ki ga opisuje niz t.

(*c*) Po n prepogibih imamo 2^n plasti, torej bo imela naša slika 2^n vrstic. Pred zadnjim od teh prepogibov je imel list 2^{n-1} plasti in ob zadnjem prepogibu so se vse te plasti prepognile in so zdaj na desnem robu lista. Desni del slike (desnih 2^{n-1} stolpcev) je torej vedno takšne oblike:

```
        ---\
        --\|
    -\  -\||
\   \|  \|||
/   /|  /|||   itd.
    -/  -///
        --//
        ---/
```

Levi del slike pa lahko sestavimo z naslednjim razmislekom: ob vsakem prepogibu lista se bivši desni del preseli na levo, lahko nad ali pa pod dosedanji levi del (odvisno od smeri prepogiba), pri tem pa se tudi zasuka za 180 stopinj. Primer vidimo na naslednji sliki (stari levi in desni del sta označena z $\mathcal{L}$ in $\mathcal{D}$, novi desni del pa z $\mathcal{D}'$):



Če torej levi in desni del slike hranimo v tabelah velikosti $2^n \times 2^{n-1}$, lahko po vsakem prepogibu brez posebnih težav izračunamo novi levi in desni del. Novi levi del dobimo tako, da stari desni del zasukamo za 180 stopinj in ga postavimo nad ali pod stari

levi del; novi desni del pa narišemo čisto sistematično, da predstavlja 2^n vgnezdenih prepognjenih plasti papirja. Ker naloga pravi, da mora biti naša slika široka $2^n + 10$ stolpcev, moramo med levi in desni del narisati še deset stolpcev iz samih znakov -.

Oglejmo si še primer implementacije tega postopka v pythonu. Levi in desni del slike hranimo v tabelah L in D, pri čemer je L[i] niz, ki predstavlja i-ti najbolj levi stolpec slike, D[i] pa niz, ki predstavlja i-ti najbolj desni stolpec slike. To, da nizi predstavljajo stolpce namesto vrstic, se izkaže za bolj prikladno predvsem pri tvorbi novega desnega dela slike po prepogibu (DD v spodnjem podprogramu):

```
def Narisi(prepogibi):
    def rev(s): return s[::-1]
    prepogibi = prepogibi.lower()
    L = ["o"]; D = ["-"]        # L[0] = skrajni levi stolpec; D[0] = skrajni desni; ipd.
    for k in range(len(prepogibi)):
        # Naredili smo že k prepogibov. Imamo torej 2^k plasti, v vsaki od
        # tabel L in D pa imamo 2^(k-1) stolpcev.
        if prepogibi[k] == 'g': LL = [rev(D[i]) + L[i] for i in range(len(L))]
        else: LL = [L[i] + rev(D[i]) for i in range(len(L))]
        if k > 0: LL += ["-" * (2 ** (k + 1))] * (2 ** (k - 1))
        DD = ["-" * i + "\\" + "|" * (2 ** (k + 1) - 2 * i - 2) + "/" + "-" * i
                                                                  for i in range(0, 2 ** k)]
        L = LL; D = DD
    for i in range(len(L[0])):
        print "%s----------%s" % ("".join(s[i] for s in L), "".join(s[i] for s in reversed(D)))
Narisi("gdd") # nastane zadnja slika iz besedila naloge
```

18. Tkanina

(a) Vpeljimo v naš pravokotni kos blaga velikosti $w \times h$ koordinatni sistem: stolpce oštevilčimo od 0 do $w - 1$, vrstice pa od 0 do $h - 1$. Podobno bi lahko naredili tudi za osnovni vzorec. Naloga pravi, da je izhodišče (zgornji levi kot) pri osnovnem vzorcu zamaknjeno za $\triangle x$ stolpcev levo in $\triangle y$ vrstic gor glede na izhodišče našega kosa blaga; torej celica (x, y) na našem kosu blaga ustreza celici $(x + \triangle x, y + \triangle y)$ na osnovnem vzorcu. Ker se osnovni vzorec ponavlja, lahko od x-koordinate vzamemo le ostanek po deljenju s t (širina osnovnega vzorca), od y-koordinate pa ostanek po deljenju z 2^t (višina osnovnega vzorca). Paziti moramo še na to, da v vrstici y osnovnega vzorca ni dvojiški zapis števila y, pač pa dvojiški zapis števila $2^t - 1 - y$ (ta ima ničle tam, kjer ima y enice, in obratno).

```
void Narisi(int t, int w, int h, int dx, int dy)
{
    int x, y, xx, yy;
    for (y = 0; y < h; y++)
    {
        yy = (y + dy) % (1 << t);
        yy = (1 << t) - 1 - yy;
        for (x = 0; x < w; x++)
        {
            xx = (x + dx) % t;
            printf("%d", (yy >> (t - 1 - xx)) & 1);
        }
        printf("\n");
    }
}
```

(b) Vrstica y naše tkanine ustreza vrstici $(y+\triangle y) \bmod 2^t$ osnovnega vzorca. Če torej pogledamo 2^t zaporednih vrstic naše tkanine, se pojavijo v njih ravno vse vrstice osnovnega vzorca, vsaka natanko enkrat. Opazimo lahko, da je v vsakem stolpcu osnovnega vzorca 2^{t-1} ničel in 2^{t-1} enic. Zato tudi v naši tkanini, če pogledamo 2^t zaporednih vrstic, lahko vidimo, da v vsakem stolpcu nastopi 2^{t-1} ničel in 2^{t-1} enic. V prvih 2^t vrsticah naše tkanine je torej $w \cdot 2^{t-1}$ enic, prav toliko tudi v naslednjih 2^t vrsticah in tako naprej. Takšnih skupin po 2^t vrstic je $\lfloor h/2^t \rfloor$ in nam torej prispevajo skupaj $\lfloor h/2^t \rfloor w 2^{t-1}$ enic.

Ostane nam še zadnjih $h' := h \bmod 2^t$ vrstic našega kosa tkanine, od vrstice $y' := h - h' = \lfloor h/2^t \rfloor 2^t$ do vrstice $h-1$. Koliko enic je na tem območju? Tako kot vse vrstice, ki so večkratniki 2^t, ustreza tudi vrstica y' naše tkanine vrstici $\triangle y$ osnovnega vzorca. Opazovano območje tkanine torej pokriva na osnovnem vzorcu h' vrstic z začetkom pri $\triangle y$. (Če je $\triangle y + h' > 2^t$, bi bilo pravzaprav bolj točno reči, da pokriva opazovano območje na osnovnem vzorcu vrstice od $\triangle y$ do $2^t - 1$ in od 0 do $\triangle y + h' - 2^t - 1$.)

Naj bo $f(x, y)$ število enic, ki se pojavljajo v osnovnem vzorcu v stolpcu x v prvih y vrsticah. Če bomo znali izračunati to, lahko izračunamo vse ostalo, kar nas zanima; na primer, število enic v vrsticah od $\triangle y$ do $\triangle y + h' - 1$ stolpca x je potem enako $f(x, \triangle y + h') - f(x, \triangle y)$. Oglejmo si zdaj, kako izračunati $f(x, y)$. Pri $x = t - 1$ imamo skrajno desni stolpec osnovnega vzorca, ki je oblike $10101010\ldots$; pri $x = t - 2$ imamo predzadnji stolpec, ki je oblike $11001100\ldots$; v splošnem torej vidimo, da se v stolpcu x izmenjujejo skupine 2^{t-x-1} enic in skupine 2^{t-x-1} ničel. V prvih y vrsticah imamo torej $\lfloor y/2^{t-x} \rfloor$ skupin vsake vrste, tako da iz njih dobimo $2^{t-x-1} \lfloor y/2^{t-x} \rfloor$ enic. Ostane še zadnjih $y \bmod 2^{t-x}$ vrstic; če je to manj kot 2^{t-x-1}, imamo v njih same enice, sicer pa tu po 2^{t-x-1} enicah nastopijo ničle.

Zapišimo dobljeni postopek še v C-ju. Glavna funkcija je PrestejEnice, pomožni podprogram EniceVStolpcu pa računa funkcijo $f(x, y)$.

```
/* Vrne število enic v prvih y vrsticah stolpca x osnovnega vzorca.
   Deluje tudi, če je x ≥ t in/ali y > 2^t. */
int EniceVStolpcu(int t, int x, int y)
{
  int stEnic;
  x %= t; /* pri x ≥ t je stolpec x v osnovnem vzorcu enak stolpcu x mod t */
  /* Najprej imamo nekaj skupin po 2^(t-x) vrstic, ki so prisotne
     v celoti in jih sestavlja pol ničel in pol enic. */
  stEnic = (y >> (t - x)) << (t - x - 1);
  /* Naslednja skupina ni prisotna v celoti. Poglejmo, koliko vrstic te skupine je prisotnih;
     prva polovica skupine je sestavljena iz enic, druga iz ničel in slednjih ne smemo šteti. */
  y %= (1 << (t - x));
  if (y > (1 << (t - x - 1))) y = 1 << (t - x - 1);
  return stEnic + y;
}

int PrestejEnice(int t, int w, int h, int dx, int dy)
{
  int x, stEnic, k;
  /* Najprej imamo na tkanini h - h mod 2^t vrstic,
     v katerih je polovica ničel in polovica enic. */
  stEnic = ((h >> t) << (t - 1)) * w;
  /* Ostane še zadnjih h mod 2^t vrstic. */
```

```
  h %= (1 << t);
  for (x = 0; x < t; x++)
  {
    /* Stolpec x naše tkanine ustreza stolpcu x + dx osnovnega vzorca. Koliko je
       na tkanini vseh stolpcev, ki ustrezajo temu stolpcu osnovnega vzorca? */
    k = (w / t) + (x < w % t ? 1 : 0);
    /* Gledamo h vrstic naše tkanine, ki ustrezajo vrsticam od dy do dy + h
       na osnovnem vzorcu. Poglejmo, koliko je tam enic. */
    stEnic += k * (EniceVStolpcu(t, x + dx, h + dy) - EniceVStolpcu(t, x + dx, dy));
  }
  return stEnic;
}
```

19. Prelom besedila

Če si v mislih predpišemo največjo še sprejemljivo dolžino vrstice (recimo največ t znakov), ni težko preveriti, ali je mogoče dano besedilo zapisati z največ k vrsticami ali ne. Sprehodimo se po njem in dodajajmo besede v trenutno vrstico; če bi njena dolžina presegla največjo dovoljeno dolžino vrstice, pa začnimo novo vrstico. Če presežemo k vrstic, še preden smo prišli do konca besedila, vemo, da smo si postavili prehudo omejitev.

Najmanjši primeren t lahko poiščemo z bisekcijo:

$t_l :=$ (dolžina najdaljše besede v vhodnem besedilu) $- 1$;
$t_d :=$ (dolžina celotnega besedila);
while $t_d - t_l > 1$:
  (* *Invarianta: besedilo se da zapisati z največ k vrsticami dolžine največ t_d, ne pa tudi z največ k vrsticami dolžine največ t_l.* *)
  $t_m := \lfloor (t_l + t_d)/2 \rfloor$;
  if lahko besedilo zapišemo z največ k vrsticami dolžine največ t_m
    then $t_d := t_m$;
    else $t_l := t_m$;
return t_d;

Če je n skupna dolžina besedila, bo imela glavna zanka tega postopka $O(\log n)$ iteracij, pri vsaki pa bomo imeli $O(n)$ dela, da preverimo, ali je mogoče besedilo prelomiti na k vrstic ali ne.

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#define MaxN 200

/* Preveri, ali je mogoče besedilo s razbiti na največ k vrstic dolžine največ t. */
bool JePrelomMogoc(char *s, int t, int k)
{
  int dolzina = 0, presledki, beseda;
  while (*s)
  {
    /* Poglejmo, koliko presledkov sledi in kako dolga je naslednja beseda. */
    presledki = 0; while (*s == ' ') s++, presledki++;
    beseda = 0; while (*s && *s != ' ') s++, beseda++;
    if (beseda > t) return false;
```

```
      /* Če ne gre oboje v trenutno vrstico, bo treba začeti novo vrstico. */
      dolzina += presledki + beseda;
      if (dolzina > t) {
         if (--k <= 0) return false; /* Porabili smo že preveč vrstic. */
         dolzina = beseda; }
   }
   return true;
}

int main()
{
   char s[MaxN + 2]; int k, tl, td, tm; FILE *f;
   f = fopen("prelom.in", "rt");
   fgets(s, MaxN + 2, f);
   k = strlen(s); if (k > 0 && s[k - 1] == '\n') s[k - 1] = 0;
   fscanf(f, "%d", &k); fclose(f);

   tl = 0; td = strlen(s);
   while (td - tl > 1) {
      tm = (tl + td) / 2;
      if (JePrelomMogoc(s, tm, k)) td = tm; else tl = tm; }

   f = fopen("prelom.out", "wt"); fprintf(f, "%d\n", td); fclose(f); return 0;
}
```

20. Mnogosončje

Če predpostavimo, da so obhodni časi p_i celoštevilski in da t ni prevelik, lahko nalogo rešimo zelo preprosto s simulacijo: v zanki za vsako uro pogledamo, koliko sonc je tisto uro na nebu, in povečamo števce a_i, ki povedo, koliko časa je minilo od zadnjega vzhoda posameznega sonca (tako da vemo, katera trenutno osvetljujo zgornjo stran planeta, katera pa ne). Rezultate hranimo v tabeli, ki jo imenujmo d; tako torej d_s pove, koliko ur je bil planet osvetljen z natanko s sonci.

Vhodni podatki: $a = (a_1, \ldots, a_n)$, $p = (p_1, \ldots, p_n)$, t

for $s := 0$ **to** n **do** $d_s := 0$;
for $k := 1$ **to** t:
  $s := 0$;
  for $i := 1$ **to** n:
    if $a_i < p_i$ **then** $s := s + 1$;
    $a_i := (a_i + 1) \bmod 2p_i$;
  $d_s := d_s + 1$;

Ob koncu tega postopka vsebuje tabela d rezultate, po katerih sprašuje naša naloga. Toda če je t velik, bo ta postopek zelo počasen; če časi p_i niso celi, pa sploh ne bo vračal pravih rezultatov, pač pa le neke približke.

Boljšo rešitev dobimo, če upoštevamo, da do spremembe v številu sonc na nebu (vrednost s v gornjem postopku) pride le v tistih trenutkih, ko kakšno sonce vzide ali zaide, vmes pa ne. Naši zanki, ki naj bi pregledala časovno obdobje t ur, se torej ni treba ustavljati na vsako uro in šteti sonc: dovolj je že, če pogleda stanje ob sončnih vzhodih in zahodih. Naj bo torej t_i naslednji trenutek, ko bo i-to sonce vzšlo ali zašlo; z_i pa naj bo **true**, če bo zašlo, in **false**, če bo vzšlo.

for $s := 0$ **to** n **do** $d_s := 0$;

$s := 0$;
for $i := 1$ **to** n:
  if $a_i < p_i$ **then** $z_i :=$ **true**; $t_i := p_i - a_i$; $s := s + 1$;
    else $z_i :=$ **false**; $t_i := 2p_i - a_i$;

$k := 0$;
while $k < t$:
  (* *Ob času k je na nebu s sonc. Kdaj nastopi naslednja sprememba?* *)
  $i :=$ planet z najmanjšo t_i;
  $k' := \min\{t_i, t\}$;
  $d_s := d_s + (k' - k)$;
  if z_i **then** $s := s - 1$ **else** $s := s + 1$;
  $z_i :=$ **not** z_i; $t_i := t_i + p_i$; $k := k'$;

Kako poiskati planet z najmanjšo t_i (torej tistega, pri katerem bo nastopila prva sprememba po trenutnem času k)? Seveda se lahko sprehodimo v zanki po vseh planetih; učinkovitejša rešitev pa je, da planete hranimo v kopici, tako da bo tisti z najmanjšo t_i vedno pri roki v korenu kopice, po vsaki spremembi (povečanju t_i) pa bomo imeli $O(\log n)$ dela s tem, da kopico popravimo. Tako nam bo vsaka iteracija glavne zanke vzela le še $O(\log n)$ časa, ne pa $O(n)$.

Možna je še ena izboljšava. Naj bo P najmanjši skupni večkratnik vseh števil $p_1, \ldots, p_n$. Potem vemo, da se po $2P$ urah nahajajo vsa sonca v enakem položaju kot na začetku opazovanja. Torej lahko naše opazovano obdobje t ur razdelimo takole: najprej pride $\lfloor t/2P \rfloor$ obdobij, ki so dolga vsako po $2P$ ur in so si med seboj glede osvetljenosti popolnoma enaka; torej je dovolj, če odsimuliramo eno tako obdobje in potem rezultate v tabeli d pomnožimo s $\lfloor t/2P \rfloor$. Na koncu pa pride še zadnjih $t \bmod 2P$ ur — to obdobje lahko odsimuliramo posebej ali pa si rezultate zanj celo pripravimo že med prvo simulacijo (tisto, ki je pokrila obdobje $2P$ ur); pri tem moramo paziti le, da se moramo potem ustaviti tudi pri $k = t \bmod 2P$, četudi takrat ne zaide ali vzide nobeno sonce.

21. Dvižni most

Obnašanje mostu se ponavlja s periodo $a + b$; če torej nek avtomobil pripelje do mostu ob času t_i, je njegova usoda popolnoma enaka, kot če bi pripeljal ob času $t_i - k(a+b)$ za poljubno celo število k. Torej je dovolj, če od vsakega t_i vzamemo le njegov ostanek po deljenju z $a + b$. V nadaljevanju bomo zato predpostavili, da vsi t_i ležijo na intervalu $[0, a + b)$. V tem intervalu se naš most natanko enkrat dvigne in enkrat spusti. Naj bo d čas, ko se most dvigne; spusti se potemtakem ob času $(d + a) \bmod (a + b)$. Število d je v bistvu že rezultat, po katerem sprašuje naloga; če je $d < b$, je most ob času 0 spuščen in se bo dvignil čez d sekund, sicer pa je most ob času 0 dvignjen in se bo čez $d - b$ sekund spustil.

Recimo, da izberemo d tako, da nobeden od avtomobilov ne pripelje na most ob trenutku, ko se ta ravno dviguje ali spušča. Za vsak avtomobil naj bo ε_i čas, ki mine med njegovim prihodom (t_i) in prvo naslednjo spremembo mostu. Po predpostavki so vsi $\varepsilon_i > 0$; potemtakem je tudi $\varepsilon := \min_i \varepsilon_i$ večji od 0. Če bi zdaj d zmanjšali za ε, bi se most dvignil in spustil za ε sekund bolj zgodaj; avtomobili, ki so morali

pri d čakati na dvig mostu, bodo morali zdaj čakati ε sekund manj; tisti pa, ki jim pri d ni bilo treba čakati, ker so prišli v času, ko je bil most spuščen, bodo tudi zdaj še vedno prišli pred spuščen most in jim ne bo treba čakati. Tako torej vidimo, da če imamo rešitev, pri kateri noben avtomobil ne pripelje tik v trenutku, ko se most dviguje ali spušča, ta rešitev gotovo ni optimalna, saj jo lahko izboljšamo, če čas dviga mostu premaknemo malo nazaj. Torej se lahko pri iskanju optimalne rešitve omejimo na tiste možnosti, pri katerih dvig ali spust mostu sovpada s prihodom kakšnega od avtomobilov. Možni kandidati za d so torej vsi t_i (kar pomeni, da avtomobil i pripelje takrat, ko se most dvigne) in vsi $(t_i + b) \bmod (a + b)$ (kar pomeni, da avtomobil i pripelje takrat, ko se most spusti). Za d imamo torej največ $2n$ možnosti, ki jih oštevilčimo v naraščajočem vrstnem redu kot $d_1, d_2, \ldots, d_{2n}$.

Mislimo si nek konkreten čas dviga mostu d. Število avtomobilov, ki pri takšnem d pripeljejo pred most takrat, ko je dvignjen, in morajo zato čakati, označimo s $f(d)$; njihov skupni čas čakanja pa označimo z $g(d)$. Za začetek lahko izračunamo $f(d_1)$ in $g(d_1)$ preprosto tako, da v zanki pregledamo vse avtomobile in pri vsakem preverimo, če bo moral čakati in kako dolgo.

Kaj pa, če se zdaj d poveča od $d = d_1$ na $d = d_2$? Ko se d poveča z d_1, morajo nekateri avtomobili začeti čakati, prej pa jim ni bilo treba; to so tisti, pri katerih je $t_i = d_1$; čas čakanja je pri vsakem od njih b, nato pa se počasi zmanjšuje, če d povečujemo. Po drugi strani pa, ko se d poveča z d_1, nekaterim avtomobilom ni več treba čakati, prej pa so morali; to so tisti, pri katerih je $t_i = (d_1 - a) \bmod (a + b)$. Vsakemu avtomobilu, ki mora pri $d \in (d_1, d_2)$ čakati na prečkanje mostu, se čas čakanja zmanjša za $d_2 - d_1$, ko se d premakne od d_1 do d_2. Enako lahko razmišljamo tudi v nadaljevanju, ko premaknemo d od d_2 do d_3 in tako naprej. Vidimo, da je koristno, če pri vsakem d_k vemo, ali nek avtomobil pri tem d začne čakati ($d_k = t_i$) ali neha čakati ($d_k = t_i + b$ ali $d_k = t_i - a$); temu podatku recimo s_k ($s_k = 1$, če začne čakati, sicer pa $s_k = 0$).

Zdaj imamo torej takšen postopek:

$L :=$ prazen seznam;
for $i := 1$ **to** n:
  $t_i := t_i \bmod (a + b)$;
  dodaj $(t_i, 1)$ in $((t_i + b) \bmod (a + b), 0)$ v seznam L;
uredi seznam L naraščajoče;
v tem vrstnem redu označimo k-ti člen seznama z (d_k, s_k);

$f := 0$; $g := 0$;
for $i := 1$ **to** n:
  if $t_i < d_1 < t_i + a$ **then** $f := f + 1$; $g := g + t_i + a - d_1$
  else if $t_i < d_1 + a + b < t_i + a$ **then** $f := f + 1$; $g := g + t_i - d_1 - b$;

$d^\star := d_1$; $g^\star := g$;
for $k := 1$ **to** $2n - 1$:
  if $s_k = 1$ **then** $f := f + 1$; $g := g + a$;
  else $f := f - 1$;
  $g := g - (d_{k+1} - d_k) \cdot f$;
  if $g > g^\star$ **then** $d^\star := d_{k+1}$; $g^\star := g$;

return $d^\star$;

V seznamu L se lahko zgodi, da je več zaporednih časov d_k enakih in da nekateri od njih izvirajo od avtomobilov, ki bi pri $d = d_k$ pripeljali do mostu ravno v času dvigovanja ($s_k = 1$), nekateri pa od takih, ki bi pri $d = d_k$ pripeljali do mostu ravno v času spuščanja ($s_k = 0$). Pomembno je, da tiste s $s_k = 0$ obdelamo prej kot tiste s $s_k = 1$, ker se pri slednjih g že povečuje, čeprav pri $d = d_k$ ti avtomobili še ne čakajo (ker pripeljejo ravno v trenutku, ko se most dviguje). Načeloma naj bi za to poskrbelo že urejanje seznama L, kjer se razume, da če se več zapisov ujema v prvi komponenti, se jih uredi še po drugi, tako da tisti s $s_k = 0$ pridejo pred tiste s $s_k = 1$.

Z urejanjem imamo $O(n \log n)$ dela, ostale zanke pa nam vzamejo le še $O(n)$ časa, tako da je časovna zahtevnost celotnega postopka $O(n \log n)$.

22. Viadukt

Za optimalno rešitev prav gotovo velja naslednje: vse omejitve so izpolnjene, če pa bi katerokoli a_i še kaj povečali (torej če bi dodali še kak avtomobil na i-ti odsek), bi prekoračili vsaj kakšno od omejitev, ki pokrivajo odsek i. (Obratno sicer ne velja nujno: če ne moremo povečati nobenega a_i, ne da bi prekoračili kakšno omejitev, to še ne pomeni nujno, da je naša trenutna rešitev optimalna.) Lahko bi torej poskusili rešiti nalogo s takšnim postopkom:

1 postavi vse a_i na 0;
2 poišči kakšen tak a_i, ki se ga še da povečati,
  ne da bi pri tem prekoračili kakšno omejitev;
3 če takega ni, končaj;
4 sicer povečaj tisti a_i toliko, kolikor se da, in se vrni v točko 2;

Hitro lahko najdemo primere, ko ta algoritem ne najde optimalne rešitve. Če imamo na primer tri odseke a_1, a_2, a_3 in dve omejitvi $a_1 + a_2 \leq 1$ in $a_2 + a_3 \leq 1$, bi se lahko zgodilo, da bi naš algoritem najprej povečal a_2 na 1 in nato ugotovil, da po tem ne more povečati nobene spremenljivke več; tako bi našel rešitev $(0, 1, 0)$, optimalna rešitev pa je $(1, 0, 1)$.

Torej ni vseeno, kateri a_i izberemo (če je več takih, ki bi jih v danem trenutku mogli povečati). Poskusimo biti sistematični: naš postopek dopolnimo tako, da v točki 2 vzamemo vedno najmanjši tak i, pri katerem je mogoče a_i še kaj povečati. Z drugimi besedami, po viaduktu se sprehajamo od leve proti desni in na vsak odsek dodamo toliko avtomobilov, kolikor je le mogoče, ne da bi prekoračili kakšno omejitev.

Tako popravljeni algoritem bi lahko zdaj preizkušali na raznih majhnih testnih primerih, kjer lahko optimalno rešitev poiščemo tudi z grobo silo, in videli bi, da naš algoritem vedno vrača optimalne rešitve. Torej lahko posumimo, da jih vrača tudi v splošnem, in poskusimo to dokazati.

Naj bo zdaj $a = (a_1, \ldots, a_n)$ rešitev, ki jo je našel naš pravkar opisani požrešni algoritem; in naj bo $b = (b_1, \ldots, b_n)$ optimalna rešitev. Če je možnih več enako dobrih rešitev, vzemimo za b tisto, ki je leksikografsko največja (torej ima največjo b_1; med tistimi s takšno b_1 vzemimo tisto z največjo b_2 in tako naprej). Recimo zdaj, da naš algoritem ni našel optimalne rešitve; naj bo i najmanjši tak indeks, pri katerem je $a_i \neq b_i$. Ali je takrat mogoče, da je $b_i > a_i$? Definirajmo $c := (a_1, \ldots, a_{i-1}, 0, \ldots, 0)$

— takšen je bil videti vektor a, preden je naš požrešni algoritem izbral vrednost a_i. Naj bo $f_k := w_k - \sum_{j=u_k}^{\min\{v_k, i-1\}} c_j$ število dodatnih avtomobilov, ki bi se jih takrat še dalo dodati k omejitvi k. Naš požrešni algoritem je vzel $a_i := \min\{f_k : u_k \le i \le v_k\}$. Čim bi na a_i postavili neko večjo vrednost, bi bila neka omejitev že takrat prekršena; torej je (če je $b_i > a_i$) že vektor $(a_1, \ldots, a_{i-1}, b_i, 0, \ldots, 0)$ neveljaven, kaj šele vektor $(a_1, \ldots, a_{i-1}, b_i, b_{i+1}, \ldots, b_n) = b$.

Torej je $b_i < a_i$. Ker pa je skupno število avtomobilov na b večje kot na a, mora biti nekje kasneje še nek indeks k, pri katerem je $b_k > a_k$ in torej $b_k > 0$. Naj bo zdaj j najmanjši tak indeks, ki je večji od i in pri katerem je $b_j > 0$. Torej imamo:

$$\begin{array}{lccccccc} a = (\ldots, & a_i, & a_{i+1}, & a_{i+2}, & \ldots, & a_{j-1}, & a_j, & \ldots) \\ b = (\ldots, & b_i, & 0, & 0, & \ldots, & 0, & b_j, & \ldots) \end{array}$$

in $a_i > b_i$ in $b_j > 0$. Ker je b optimalna rešitev, pomeni, da se b_i gotovo ne da povečati (če pustimo ostale komponente pri miru), torej je neka omejitev, ki pokriva b_i, že zdaj zasičena. Recimo, da je to omejitev (u_k, v_k, w_k). Seveda velja $u_k \le i \le v_k$ (ker smo rekli, da ta omejitev pokriva b_i). To, da je omejitev zasičena, pomeni, da je $\sum_{t=u_k}^{v_k} b_t = w_k$. Ta omejitev gotovo ne pokriva le indeksov $1, \ldots, i$ (z drugimi besedami: gotovo ni $v_k \le i$), ker bi v tem primeru bila ta ista omejitev v vektorju a že presežena (ker je $a_i > b_i$). Torej pokriva ta omejitev še vsaj nekaj indeksov, večjih od i. Toda če bi bil $v_k < j$, bi ta omejitev v vektorju a zdaj dobila vsoto

$$\begin{array}{cl} & (a_{u_k} + \ldots + a_{i-1} + a_i + a_{i+1} + \ldots + a_{v_k}) \\ = & (b_{u_k} + \ldots + b_{i-1} + a_i + a_{i+1} + \ldots + a_{v_k}) \\ > & (b_{u_k} + \ldots + b_{i-1} + b_i + 0 + \ldots + 0) \\ = & (b_{u_k} + \ldots + b_{i-1} + b_i + b_{i+1} + \ldots + b_{v_k}) = w_k, \end{array}$$

torej bi bila ta omejitev v a že prekršena. (Neenakost med drugo in tretjo vrstico sledi iz tega, da je $a_i > b_i$ in da so $a_{i+1}, \ldots, a_{v_k}$ vsi večji ali enaki 0, obenem pa so $b_{i+1}, \ldots, b_{v_k}$ vsi enaki 0, saj je $v_k < j$.) Torej mora ta omejitev pokrivati vsaj še indeks j. Enak razmislek lahko ponovimo za vsako v b-ju zasičeno omejitev, ki pokriva indeks i — vse te omejitve morajo pokrivati tudi indeks j. Torej, če zdaj b_j zmanjšamo za 1, b_i pa povečamo za 1, so te omejitve po novem še vedno zasičene (nobena pa ni prekršena); tiste, ki pokrivajo i in prej niso bile zasičene, zdaj mogoče sicer nekatere so, nobena pa ni prekršena; tiste, ki pokrivajo j, ne pa i, pa so zdaj nekatere mogoče celo nezasičene, čeprav so prej bile. Kakorkoli že, rešitev je še vedno veljavna in enako dobra kot b, obenem pa je leksikografsko večja kot b, kar je protislovje, ker smo rekli, da za b izberemo leksikografsko največjo med optimalnimi rešitvami.

Torej ni mogoče, da bi se b (leksikografsko največja med optimalnimi rešitvami) kaj razlikovala od a (t.j. rešitve našega požrešnega algoritma). □

Zdaj torej vemo, da požrešni algoritem vrača optimalne rešitve; razmislimo še o tem, kako ga lahko učinkovito implementiramo. V glavni zanki bomo šli po viaduktu od leve proti desni (torej z i od 1 do n); v vsakem trenutku nas bo zanimalo, katere so trenutno odprte omejitve (torej tiste k, ki pokrivajo trenutni odsek: $u_k \le i \le v_k$). Za vsako od teh omejitev bi nam prišel prav podatek o tem, koliko dodatnih avtomobilov lahko še dodamo na levo stran omejitve, preden bo postala enaka desni: to je $f_k := w_k - (a_{u_k} + \ldots + a_{v_k})$. Na a_i hočemo zdaj dodati čim več avtomobilov,

vendar tako, da pri tem ne bomo prekoračili prav nobene omejitve; torej moramo med vsemi trenutno odprtimi omejitvami vzeti tisto, ki ima najmanjši f_k. Ko povečamo a_i z 0 na neko novo vrednost, se seveda f_k za vse odprte omejitve zmanjša za toliko, kolikor se je a_i povečal.

```
1 H := prazna množica;
2 for i := 1 to n:
3   za vsak k, ki ima u_k = i:
4     postavi f_k := w_k in dodaj k v H;
5   a_i := min{f_k : k ∈ H};
6   za vsak k ∈ H:
7     f_k := f_k − a_i;
8   za vsak k, ki ima v_k = i:
9     zbriši k iz H;
```

Zanko v vrsticah 3–4 lahko izvedemo učinkovito in elegantno tako, da si vnaprej pripravimo seznam, v katerem so omejitve urejene po u_k. Pri vsakem i potem nadaljujemo s pregledovanjem seznama tam, kjer smo končali pri $i-1$. Tako ima zanka v vrsticah 3–4 vsega skupaj le $O(m)$ iteracij (pri vseh i skupaj). Podobno je z zanko v vrsticah 8–9, za katero potrebujemo še seznam, v katerem so omejitve urejene po v_k.

Primeren kandidat za podatkovno strukturo H je na primer kopica. V obeh primerih nam operacije, kot so dodajanje, brisanje in spreminjanje vrednosti f_k (v vrstici 7) vzamejo po $O(\log m)$ časa. Iskanje minimuma (v vrstici 5) pa nam vzame celo le $O(1)$ časa (ker je iskani element že v korenu vrstice in ker ga nimamo namena kar takoj pobrisati iz nje).

Najbolj zaskrbljujoč del tega postopka je zanka v vrsticah 6–7. Pri vsakem i se moramo sprehoditi po vseh odprtih omejitvah; ker je teh lahko $O(m)$, bi se vrstica 7 lahko izvedla vsega skupaj $O(nm)$-krat, kar je že neugodno veliko. Toda ker od vseh omejitev v H odštevamo isto vrednost, namreč a_i, to nič ne vpliva na njihov medsebojni vrstni red po f_k (tista, ki je imela prej najmanjši f_k, ga ima še zdaj ipd.). Torej ni zares nujno, da popravljamo vsak f_k posebej; lahko si kar v neko globalno spremenljivko zapišemo, koliko bi morali odšteti (pa nismo). Spodnja različica postopka pravi tej globalni spremenljivki D, v kopici H pa zdaj ne hrani več vrednosti f_k, pač pa vrednosti $g_k := f_k + D$.

```
1 H := prazna množica; D := 0;
2 for i := 1 to n:
3   za vsak k, ki ima u_k = i:
4     postavi g_k := w_k + D in dodaj k v H;
5   a_i := min{g_k : k ∈ H} − D;
6   D := D + a_i;
7   za vsak k, ki ima v_k = i:
8     zbriši k iz H;
```

Časovna zahtevnost postopka je zdaj takšna: $O(m \log m)$ za pripravo seznamov, kjer so omejitve urejene po u_k oz. v_k; zanki v vrsticah 3–4 in 7–8 imata vsega skupaj $O(m)$ iteracij, v vsaki se izvede ena operacija nad kopico H, kar je torej skupaj $O(m \log m)$;

vrstica 5 prispeva še $O(n)$ operacij, ki le poškilijo v koren kopice, vsaka torej vzame le $O(1)$ časa; časovna zahtevnost celotnega postopka je torej $O(n + m \log m)$.

23. Meje med stavki

Pravila iz besedila naloge pravijo, da lahko pride meja med povedima le za končnim ločilom, vendar ne nujno takoj za njim; preskočiti moramo morebitne oklepaje in/ali narekovaje, nato pa še morebitne presledke: mejo lahko postavimo šele za vso to navlako. Oglejmo si zdaj znak, ki pride za mestom, na katerega poskušamo postaviti mejo. Če je ta naslednji znak tudi ločilo, potem meje ne smemo postaviti (to pravilo skrbi za to, da ne postavljamo odvečnih mej v primerih, ko se stavek konča npr. s tremi pikami ali čim podobnim). Če je bilo naše ločilo pika, pa moramo preveriti še nekaj dodatnih pravil: pika tik pred številko ali pika med dvema velikima črkama ne pomeni konca povedi; in če za piko poiščemo prvi tak znak, ki je črka ali ločilo, in se izkaže, da je ta znak mala črka, potem tista pika tudi ni pomenila konca povedi (to pravilo poskrbi za nekatere primere, ko je pika uporabljena v okrajšavah, ne kot končno ločilo).

Spodnji program ima celotno besedilo v tabeli `s` in se po njej sprehaja s številcem `i`; ko naleti na končno ločilo, se pomakne s številcem `j` mimo morebitnih narekovajev, oklepajev in presledkov za ločilom. Nato lahko preveri pravila iz prejšnjega odstavka in se odloči, ali je treba pri `j` potegniti mejo med povedima ali ne. Če je bilo končno ločilo pri `i` pika, se je potrebno zapeljati še naprej od `j` do prvega naslednjega znaka, ki je črka ali ločilo (to naredimo s številcem `k`); če je tisti znak mala črka, meje pri `j` tudi ne smemo postaviti. V vsakem primeru drži, da znaki med ločilom `i` in položajem `j` oz. `k` niso ločila, torej nam jih v nadaljevanju ne bo treba pregledovati še enkrat, ne glede na to, ali smo zdaj pri `j` postavili mejo ali ne; zato lahko naslednja iteracija glavne zanke nadaljuje kar pri `j` oz. `k`, ne šele pri `i + 1`. To nam tudi zagotavlja, da je časovna zahtevnost celotnega postopka le $O(n)$.

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#define MaxN 1000000

bool Locilo(int c) { return c == '.' || c == '!' || c == '?'; }
bool Stevka(int c) { return '0' <= c && c <= '9'; }
bool Velika(int c) { return 'A' <= c && c <= 'Z'; }
bool Mala(int c) { return 'a' <= c && c <= 'z'; }
bool OaliN(int c) { return 0 != strchr("()[]{}\'\"", c); }
bool Presledek(int c) { return c == ' '; }

int main()
{
  char *s = (char *) malloc(MaxN + 2);
  bool meja; int i, j, k, t, n;
  FILE *f = fopen("meje.in", "rt"); fgets(s, MaxN + 2, f); fclose(f);
  n = strlen(s); if (n > 0 && s[n - 1] == '\n') s[--n] = 0;
  f = fopen("meje.out", "wt");
  for (int i = 0; i < n; i++)
  {
    fputc(s[i], f);
    if (! Locilo(s[i])) continue;
```

```
    if (s[i] == '.') {
      /* Če je pika pred številko ali med dvema velikima črkama, tu ni meje. */
      if (i + 1 < n && Stevka(s[i + 1])) continue;
      if (i > 0 && i + 1 < n && Velika(s[i − 1]) && Velika(s[i + 1])) continue; }
    /* Premaknimo se mimo oklepajev, narekovajev in presledkov; meja bo nastopila,
       če sploh bo nastopila, šele za njimi, zato jih lahko tudi že izpišemo. */
    j = i + 1;
    while (j < n && OaliN(s[j])) j++;
    while (j < n && Presledek(s[j])) j++;
    for (t = i + 1; t < j; t++) fputc(s[t], f);
    /* Med dvema ločiloma ni meje. */
    if (j < n && Locilo(s[j])) { i = j − 1; continue; }
    /* Za piko ni meje, če je naslednja črka mala, ne pa velika. */
    k = j; meja = true;
    if (s[i] == '.' && j < n) {
      while (k < n && ! Velika(s[k]) && ! Mala(s[k]) && ! Locilo(s[k])) k++;
      if (k < n && Mala(s[k])) meja = false; }
    /* Zdaj nam spremenljivka meja pove, ali imamo mejo ali ne; če jo imamo, je med j − 1
       in j; vse do j smo tudi že izpisali; in vemo, da od j do k ni nobene druge meje. */
    if (meja) fputc('|', f);
    for (t = j; t < k; t++) fputc(s[t], f);
    i = k − 1;
  }
  fclose(f); return 0;
}
```

Slabost te rešitve je, da prebere celotno besedilo naenkrat v pomnilnik; po svoje bi bilo bolj elegantno, če bi se temu izognili in zagotovili, da poraba pomnilnika ni sorazmerna z dolžino besedila. Glavno težavo nam pri tem povzročajo primeri, ko moramo besedilo pregledati daleč naprej, preden se lahko odločimo, ali moramo na nekem zgodnejšem mestu postaviti mejo ali ne. Pri naši nalogi je edino pravilo tega tipa tisto, ki pravi, da za piko ni konec povedi, če je prva naslednja črka v besedilu mala (ne pa velika). Če ne bi imeli celotnega besedila v pomnilniku, ampak bi ga brali znak za znakom iz neke vhodne datoteke, bi si morali v tem primeru zapomniti v pomnilniku vsaj znake od s[j] do s[k], kajti šele ko preberemo vse te znake, se lahko odločimo, ali je treba pred s[j] izpisati |, ki označuje mejo med povedima. Še ena možnost je, da bi se v vhodni datoteki po potrebi premaknili nazaj (s položaja k na položaj j) in ta del besedila kasneje prebrali še enkrat; to pa je neugodno, če besedilo prihaja npr. iz pipe ali s standardnega vhoda.

24. Družinska drevesa

Osebe, ki nastopajo v naših podatkih, si lahko predstavljamo kot točke grafa, relacija „je prednik od“ pa nam določa povezave tega grafa (povezava $a \to b$ pomeni, da je a prednik osebe b). Naj bo zdaj s oseba, za katero iščemo potencialne prednike. Množico zanesljivih potomcev lahko določimo preprosto tako, da pregledujemo graf (npr. v širino ali pa v globino) in označimo vse točke, ki so dosegljive iz s. Na enak način lahko določimo množico zanesljivih prednikov, le povezavam moramo slediti v nasprotni smeri. Vse točke, ki jih nismo označili pri nobenem od obeh pregledov grafa, pa so potencialni predniki.

Vhod: usmerjen acikličen graf $G = (V, E)$, točka $s \in V$
Izhod: tabela p, ki za vsako točko iz V pove, ali je potencialni prednik s-ja.

for each $u \in V$: $p[u] :=$ **true**;

$Q := \{s\}$; $p[s] :=$ **false**;
while $Q \neq \emptyset$:
  $u :=$ poljubna točka iz Q; $Q := Q - \{u\}$;
  for each $v \in V$:
    if $(u \to v) \in V$ **and** $p[v]$:
      $p[v] :=$ **false**; $Q := Q \cup \{v\}$;

$Q := \{s\}$;
while $Q \neq \emptyset$:
  $u :=$ poljubna točka iz Q; $Q := Q - \{u\}$;
  for each $v \in V$:
    if $(v \to u) \in V$ **and** $p[v]$:
      $p[v] :=$ **false**; $Q := Q \cup \{v\}$;

V praksi bi bilo seveda koristno graf predstaviti tako, da bi imeli za vsako točko seznam neposrednih prednikov in potomcev, tako da se v notranjih zankah ne bi bilo treba sprehajati po vseh točkah v, pač pa le po tistih, ki so res neposredne prednice ali potomke točke u. S tem dopolnilom je časovna zahtevnost našega postopka le $O(|V| + |E|)$. Množico Q lahko v praksi implementiramo na primer kot vrsto (in dobimo iskanje v širino) ali pa kot sklad (in dobimo iskanje v globino).

25. Poker

Ko primerjamo dve kombinaciji kart, je glavni kriterij seveda tip igre — če ima ena kombinacija višji tip igre kot druga (npr. ena barvno lestvico, druga pa tris in par), zmaga tista z višjim tipom igre. Če pa imata obe enak tip igre, ju moramo primerjati še po nečem drugem; ta dodatni kriterij pa je pri različnih tipih igre različen. Pri igrah tipa „štiri enake“ je glavni kriterij tista vrednost, ki je pojavi štirikrat; če bi se obe kombinaciji ujemali tudi v tem, pa kot sekundarni kriterij pogledamo še vrednost preostale karte (tiste, ki ne nastopa v skupini štirih enakih). Podobno pri igrah tipa „dva para“ najprej gledamo vrednost višjega para; če se kombinaciji ujemata v tem, gledamo vrednost nižjega para; če se ujemata tudi v tem, gledamo vrednost pete karte (tiste, ki ne nastopa v nobenem paru).

Prav bi torej prišel nek eleganten način, kako te različne kriterije zapakirati v eno samo številsko vrednost, na podlagi katere bomo lahko primerjali kombinacije kart med seboj. Poleg tipa igre potrebujemo še do največ pet dodatnih kriterijev, ki jih označimo s $k_1, \ldots, k_5$ (od najmanj do najbolj pomemtnih). Vrednosti kriterijev so vedno kar vrednosti kart, torej števila od 2 do 14. Za tip igre pa uporabimo števila od 1 do 8, vendar tako, da višje število pomeni višjo igro (torej ravno obratno, kot so igre oštevilčene v besedilu naloge). Te vrednosti zdaj lahko skombiniramo v oceno naše kombinacije petih kart na primer takole:

$$ocena = igra \cdot 15^5 + k_5 \cdot 15^4 + k_4 \cdot 15^3 + k_3 \cdot 15^2 + k_2 \cdot 15 + k_1.$$

(V spodnjem programu to računa funkcija Rezultat.) Tako za vsako peterico kart

dobimo preprosto celoštevilsko oceno, s pomočjo katere lahko kombinacije kart primerjamo med seboj in ugotovimo, katera je zmagovalna.

Glavnino dela v spodnji rešitvi opravi podprogram Oceni, ki za vse možne igre (od najvišje do najnižje) preveri, če jim dana kombinacija kart ustreza. Da je preverjanje lažje, si karte uredimo naraščajoče po vrednosti (za to poskrbi podprogram Uredi). Pri lestvicah moramo posebej paziti na to, da kot lestvico prepoznamo tudi kombinacijo $14-5-4-3-2$, saj naloga pravi, da so asi v vhodnih podatkih vedno predstavljeni z vrednostjo 14, za potrebe lestvic pa si jih smemo razlagati tudi kot vrednost 1.

```
#include <stdio.h>
#include <stdbool.h>
typedef struct { char b; int n; } Karta;

/* Uredi karte po naraščajoči vrednosti n.
   Ker jih je malo, uporabimo kar urejanje z vstavljanjem. */
void Uredi(Karta *karte)
{
  int i, j; Karta k;
  for (i = 1; i < 5; i++) {
    k = karte[i];
    for (j = i - 1; j >= 0 && karte[j].n > k.n; j--)
      karte[j + 1] = karte[j];
    karte[j + 1] = k; }
}

int Rezultat(int igra, int k5, int k4, int k3, int k2, int k1)
{
  const int p = 15;
  return ((((igra * p + k5) * p + k4) * p + k3) * p + k2) * p + k1;
}

int Oceni(Karta *karte)
{
  char b[5]; int n[5], i;
  Uredi(karte);
  for (i = 0; i < 5; i++) b[i] = karte[i].b, n[i] = karte[i].n;
  /* Barvna lestvica. */
  if (b[0] == b[1] && b[1] == b[2] && b[2] == b[3] && b[3] == b[4])
    if (n[1] == n[0] + 1 && n[2] == n[1] + 1 && n[3] == n[2] + 1)
      if (n[4] == n[3] + 1) return Rezultat(8, n[4], 0, 0, 0, 0);
      else if (n[0] == 2 && n[4] == 14) return Rezultat(8, n[3], 0, 0, 0, 0);
  /* Štiri enake. */
  if (n[1] == n[2] && n[2] == n[3])
    if (n[0] == n[1]) return Rezultat(7, n[1], n[4], 0, 0, 0);
    else if (n[3] == n[4]) return Rezultat(7, n[1], n[0], 0, 0, 0);
  /* Tris + par. */
  if (n[0] == n[1] && n[1] == n[2] && n[3] == n[4]) return Rezultat(6, n[2], n[3], 0, 0, 0);
  if (n[2] == n[3] && n[3] == n[4] && n[0] == n[1]) return Rezultat(6, n[2], n[0], 0, 0, 0);
  /* Ena barva. */
  if (b[0] == b[1] && b[1] == b[2] && b[2] == b[3] && b[3] == b[4])
    return Rezultat(5, n[4], n[3], n[2], n[1], n[0]);
```

```
  /* Lestvica. */
  if (n[1] == n[0] + 1 && n[2] == n[1] + 1 && n[3] == n[2] + 1)
    if (n[4] == n[3] + 1) return Rezultat(4, n[4], 0, 0, 0, 0);
    else if (n[0] == 2 && n[4] == 14) return Rezultat(4, n[3], 0, 0, 0, 0);
  /* Dva para ali en par. */
  if (n[4] == n[3])
    if (n[2] == n[1]) return Rezultat(3, n[3], n[1], n[0], 0, 0);
    else if (n[1] == n[0]) return Rezultat(3, n[3], n[1], n[2], 0, 0);
    else return Rezultat(2, n[3], n[2], n[1], n[0], 0);
  if (n[3] == n[2])
    if (n[1] == n[0]) return Rezultat(3, n[3], n[1], n[4], 0, 0);
    else return Rezultat(2, n[3], n[4], n[1], n[0], 0);
  if (n[2] == n[1]) return Rezultat(2, n[1], n[4], n[3], n[0], 0);
  if (n[1] == n[0]) return Rezultat(2, n[1], n[4], n[3], n[2], 0);
  /* Najvišja karta. */
  return Rezultat(1, n[4], n[3], n[2], n[1], n[0]);
}

bool Preberi(FILE *f, Karta *karte)
{
  int i; char b[2]; for (i = 0; i < 5; i++) {
    if (2 != fscanf(f, "%1s%d", b, &karte[i].n)) return false;
    karte[i].b = b[0]; }
  return true;
}

int main()
{
  FILE *f = fopen("karte.in", "rt");
  Karta karte[5], najKarte[5]; int ocena, najOcena = -1, i;
  /* Preberimo kombinacije kart in si zapomnimo najboljšo. */
  while (Preberi(f, karte))
    if ((ocena = Oceni(karte)) > najOcena)
      for (i = 0, najOcena = ocena; i < 5; i++) najKarte[i] = karte[i];
  /* Izpišimo rezultate. */
  fclose(f); f = fopen("karte.out", "wt");
  for (i = 0; i < 5; i++) fprintf(f, "%c%d ", najKarte[i].b, najKarte[i].n);
  fprintf(f, "\n"); fclose(f); return 0;
}
```

26. Brisanje podnizov

Naloge se lahko lotimo z dinamičnim programiranjem. Recimo, da je w dolg n znakov, x pa m znakov; i-ti znak označimo z $w[i]$ oz. $x[i]$, podniz od i-tega do vključno j-tega znaka pa z $w[i..j]$ oz. $x[i..j]$. Za začetek opazimo, da če je $m > n$, se x prav gotovo ne pojavlja kot podniz v w, zato je odgovor, ki ga išče naloga, kar enak n. V nadaljevanju torej predpostavimo, da je $m \leq n$.

Zdaj si lahko zastavimo podprobleme oblike: „ali je mogoče iz $w[i..j]$ z brisanjem pojavitev niza x dobiti niz $x[1..k]$?“ Recimo temu $f(i, j, k)$, pri čemer je $1 \leq i \leq j \leq n$ in $0 \leq k \leq m$. Funkcije f ni težko definirati rekurzivno. Če je iz $w[i..j]$ mogoče dobiti $x[1..k]$, mora eden od znakov v $w[i..j]$ biti enak $x[k]$ (in ostati nedotaknjen pri vseh brisanjih); recimo, da je to $w[t]$. Potemtakem se je moralo dati iz $w[i..t-1]$ dobiti

$x[1..k-1]$, iz $w[t+1..j]$ pa celo prazen niz. Torej imamo:

$$f(i,j,k) \Leftrightarrow \exists t : i \le t \le j \wedge w[t] = x[k] \wedge f(i,t-1,k-1) \wedge f(t+1,j,0).$$

Ko enkrat izračunamo $f(i,j,k)$ za neko konkretno trojico (i,j,k), si rezultat zapomnimo v neki tabeli, da ga ne bomo kasneje računali še večkrat. Opazimo lahko tudi, da se pri izračunu $f(i,j,k)$ sklicujemo le na rezultate za krajše dele nizov w in x, tako da lahko funkcijo računamo sistematično po naraščajočih $j-i$. Robni primer rekurzije je $k=0$, ko imamo $f(i,j,0) \Leftrightarrow j < i \vee f(i,j,n)$.

Naj bo zdaj $g(i,k)$ dolžina najkrajšega niza, ki se konča na $x[1..k]$ in ga lahko iz $w[1..i]$ dobimo z brisanjem podniza x. To funkcijo lahko računamo z naslednjim razmislekom: nekje v $w[1..i]$ se mora pojavljati $x[k]$, recimo na indeksu t ($w[t] = x[k]$); pred tem indeksom, torej iz $w[1..t-1]$, moramo potemtakem dobiti čim krajši niz, ki se konča na $x[1..k-1]$; vse za indeksom t, torej celoten $w[t+1..i]$, pa moramo pobrisati. Tako imamo:

$$\begin{aligned} g(i,k) &= \min\{g(t-1,k-1)+1 \ : \ 1 \le t \le i,\ f(t+1,i,0),\ w[t] = x[k]\} \\ g(i,0) &= \min\{g(t-1,0) \ : \ 1 \le t \le i,\ f(t+1,i,0)\}. \end{aligned}$$

Robni primeri so $g(i,k) = \infty$ za $k < i$ in $g(0,0) = 0$. Na koncu je rezultat, po katerem sprašuje naloga, ravno vrednost $g(n,0)$. Podobno kot f lahko tudi g računamo sistematično, po naraščajočih k, pri vsakem k pa po naraščajočih i.

Vidimo, da imamo z izračunom posamezne vrednosti $f(i,j,k)$ in $g(i,k)$ vsakič po $O(n)$ dela, tako da nam izračun celotne funkcije f vzame $O(n^3 m)$ časa, izračun funkcije g pa $O(n^2 m)$ časa. Zahtevnost celotne rešitve je tako $O(n^3 m)$.

Na nalogo lahko pogledamo tudi malo drugače. Predstavljajmo si graf s točkami $v_0, \ldots, v_{n+1}$; od v_i do v_{i+1} imejmo usmerjeno povezavo z oznako $w[i]$; pri $i = 0$ si mislimo, da je $w[0] = \#$ (nek poseben znak, ki v w in x sicer ne nastopa). Zdaj gremo lahko v zanki po naraščajočih i in pri vsakem i s sledenjem povezavam v obratni smeri pogledamo, iz katerih v_j se da priti v v_i z zaporedjem m povezav, katerih oznake tvorijo ravno niz $x[1..m]$; za vsak tako dobljen v_j potem pogledamo vse njegove vhodne povezave in za vsako tako povezavo, recimo $(v_k \to v_j)$ z oznako c, dodamo v graf novo povezavo $(v_k \to v_i)$ z isto oznako c. Ko je vse to končano, moramo le še poiskati dolžino najkrajše poti od v_0 do v_{n+1} (za povezave z oznako $\#$ štejmo, kot da imajo dolžino 0).

Ideja tega postopka je torej v tem, da niz predstavlja pot skozi graf, brisanje podniza x pa pomeni, da lahko del te poti preskočimo oz. zaobidemo (kar smo ponazorili z dodajanjem novih povezav, ki preskočijo del poti, iz katerega bi sicer nastal podniz x). Vendar pa se nam ob tem lahko povezave tako namnožijo, da v posamezno točko grafa kaže tudi do $O(n)$ povezav za vsako črko abecede. Pri vsakem i moramo iti $O(m)$ korakov nazaj po grafu; na vsaki oddaljenosti od i je potencialno dosegljivih $O(n)$ točk, pri vsaki od teh pa moramo pregledati $O(n)$ vhodnih povezav s pravo oznako; tako imamo torej pri vsakem i-ju do $O(n^2 m)$ dela, kar nas spet pripelje do $O(n^3 m)$ za celoten postopek.

V primerih, ko w vsebuje kakšno tako črko, ki se v x ne pojavlja, lahko w razrežemo na krajše kose, namreč take, ki so sestavljeni le iz takih črk, ki se pojavljajo v x; zdaj lahko obdelamo vsak tak kos posebej. Ker časovna zahtevnost našega postopka

narašča s kubom n-ja, je ceneje obdelati več krajših podnizov w-ja kot pa celoten niz naenkrat.

27. Skiroji

Naivna rešitev je, da pregledamo vseh $n!$ možnih permutacij; pri vsaki imamo $O(n)$ dela, da izračunamo $\min\{a_{i,\pi(i)} : 1 \le i \le n\}$; tako v času $O(n \cdot n!)$ pridemo do rešitve. Z nekaj pazljivosti bi lahko minimum deloma računali sproti med rekurzijo, s katero generiramo permutacije, in časovno zahtevnost s tem zmanjšali na $O(n!)$.

Do hitrejše (čeprav še vedno zelo neučinkovite) rešitve pridemo z dinamičnim programiranjem. Zastavimo si podproblem, pri katerem imamo le prvih i otrok in neko podmnožico vseh skirojev, recimo $S \subseteq \{1, \ldots, n\}$. Naj bo $f(i, S)$ rešitev tega podproblema. Če se odločimo, da bomo otroku i dali skiro $j \in S$, nam ostane podproblem $f(i-1, S - \{j\})$. Tako vidimo, da lahko f računamo rekurzivno:

$$f(i, S) = \max\{\min\{f(i-1, S - \{j\}), a_{ij}\} : j \in S\}.$$

Rekurzijo lahko ustavimo pri $i = 1$, ko je $f(1, \{j\}) = a_{1j}$ (ali pa celo pri $i = 0$, ko je smiselno definirati $f(0, \emptyset) = \infty$). Rezultat, po katerem sprašuje naša naloga, je potem $f(n, \{1, \ldots, n\})$.

Pri tej rekurzivni zvezi lahko vidimo, da je v S vedno natanko i skirojev. Pri posameznem i imamo torej $\binom{n}{i}$ možnih S, tako da moramo vsega skupaj rešiti $\sum_{i=0}^{n} \binom{n}{i} = 2^n$ podproblemov. Z vsakim podproblemom imamo $O(i)$ dela, tako da je časovna zahtevnost postopka zdaj $O(n \cdot 2^n)$. To je precej manj kot $O(n!)$, čeprav je še vedno prepočasi za velike n. Slabost te rešitve pa je, da ko računamo f za nek konkreten i, moramo imeti v pomnilniku rešitve za $i - 1$; poraba pomnilnika je tako $O(\max_i \binom{n}{i}) = O(\binom{n}{n/2}) = O(2^n/\sqrt{n})$.

28. Zeleni val

Naj bo d čas vožnje od enega križišča do naslednjega križišča; za križišče u naj bo $z_u \in \{0, \ldots, 2p-1\}$ njegov zamik (število, ki pove, kje v svojem ciklu prižiganja in ugašanja je bil ta semafor ob času, ko naš avtomobil začne svojo pot). Potem vemo, da če je u dosegljiv ob času t, je vsak od njegovih sosedov v dosegljiv ob času $t + d$, vendar le, če takrat na semaforju v ne gori rdeča luč (saj čakanja pri rdeči luči ne maramo, razen če je v ciljno križišče, v katerem se naša pot konča). Ta pogoj lahko preverimo tako, da izračunamo $(t+d+z_v) \bmod 2p$; če je ta rezultat na intervalu $[0, p)$, je na semaforju zelena luč v eni smeri (vodoravni oz. navpični — naloga žal ne pove jasno, v kateri), če je na intervalu $[p, 2p)$, pa v drugi smeri (navpični oz. vodoravni). Zdaj si torej lahko predstavljamo, da namesto prvotnega grafa preiskujemo nek nov, večji graf, v katerem je vsaka točka par (u, t), ki predstavlja prihod v križišče u ob času t. Ker se stanje vseh semaforjev ponovi na vsakih $2p$ sekund, je dovolj, če čase vedno gledamo le po modulu $2p$. Zdaj imamo torej takšen postopek:

za vsako križišče u: $D_u := \emptyset$;
$Q :=$ prazna vrsta; dodaj $(A, 0)$ v vrsto in dodaj 0 v D_A;
while $D_B = \emptyset$ **and** $Q \neq \emptyset$:
  vzemi prvi element (u, t) iz vrste Q;
  $t' := (t + d) \bmod 2p$;

za vsako u-jevo sosedo v:
if $(v = B$ **or** $(t' + z_v) \bmod 2p < p)$[11] **and** $t' \notin D_v$:
dodaj (v, t') v vrsto Q in dodaj t' v D_v;

Za vsako križišče u torej vzdržujemo množico časov D_u, ob katerih je dosegljivo. Preiskovanje lahko končamo, čim dosežemo ciljno križišče B, sicer pa nadaljujemo, dokler ne preiščemo vseh dosegljivih vozlišč.

To je bila rešitev za primer, ko z vožnjo začnemo ob času 0. Če te omejitve ni, lahko na začetku postopka poleg $(A, 0)$ dodamo v vrsto (in v D_A) še vse (A, t) za $t \in \{1, \ldots, 2p - 1\}$. Če je omejitev ta, da moramo vožnjo začeti v prvih desetih sekundah, pa naredimo to za vse $t \in \{0, \ldots, 9\}$.

Križišč je $m \cdot n$, možnih parov (u, t) je torej $m \cdot n \cdot 2p$, tako da je časovna zahtevnost tega postopka $O(mnp)$. Množice D_u lahko predstavimo na primer z bitnimi kartami dolžine $2p$ bitov ali pa z razpršeno tabelo (*hash table*).

Naloge so sestavili: pobiranje smeti, koščki in SG-1, prepogibanje papirja, prelom besedila, viadukt — Nino Bašić; tkanina — Matej Črepinšek; okvarjen pomnilnik — Boris Gašperin; meje med stavki — Marko Grobelnik in Janez Brank; skiroji — Tomaž Hočevar; izpis HTMLja — Jurij Kodre statistika — Mitja Lasič; tiskalnik — Mojca Miklavec; loto, poker — Matej Šprogar; mnogosončje, dvižni most, zeleni val — Mitja Trampuš; igra, poplavljanje labirinta — Matjaž Urlep; Bézierjeva krivulja, družinska drevesa — Anže Žagar; kalkulator — Anže Žagar in Primož Gabrijelčič; polinomi — Anže Žagar in Mojca Miklavec; tečajnica, kolesarjeva pot — Klemen Žagar; brez ene črke, volitve, brisanje podnizov — Janez Brank.

[11] Ali pa $(t' + z_v) \bmod 2p \geq p$, odvisno od tega, ali je pot od u do v na naši mreži v vodoravni ali v navpični smeri.

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. Ruleta

- Vseeno je, ali rešitev prebere vhodne podatke z datoteke ali s standardnega vhoda in ali pri tem še kaj izpiše na zaslon (npr. navodilo uporabniku, naj vnese rešitev). Vseeno je tudi, ali predpostavi, da se vhodni podatki končajo s koncem vrstice ali s koncem datoteke.

- Rešitev lahko prebira vhodne podatke znak za znakom ali pa kot cel niz naenkrat; oboje je enako dobro. Če bi kakšna rešitev prebrala celotno vhodno zaporedje kot eno veliko celo število in pri tem uporabila premajhen celoštevilski tip (npr. 32-bitni **int**, ki ne bi zmogel hraniti 12-mestnih celih števil), naj se ji odbije 3 točke.

- Če rešitev poleg skupnega zaslužka izpiše še kaj drugega, česar sicer naloga ne zahteva (npr. zaslužke po posameznih krogih), naj se ji zaradi tega ne odbija točk.

2. Glava e-pošte

- Vseeno je, ali rešitev bere vhodne podatke po znakih ali po vrsticah.

- 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.

- Naloga posebej pravi, da so vrstice po združevanju lahko poljubno dolge. Če rešitev naredi kakšne neupravičene predpostavke o tem, kakšna je največja možna dolžina vrstice po združevanju, naj se ji odbije 5 točk.

- Če si rešitev ob združevanju vrstic najprej pripravi celo izhodno vrstico v pomnilniku, preden jo izpiše (namesto da bi izpisovala že posamezne vhodne vrstice, le da brez znakov za konec vrstice), naj se ji zaradi tega ne odbija točk (četudi je taka rešitev potratna s pomnilnikom).

3. Trajekt

- Ker naloga zahteva opis postopka, se rešitvi ni treba ukvarjati s tem, od kod dobi vhodno zaporedje in kako ga prebere; torej je vseeno, ali rešitev kaj govori o tem ali ne.

- Največ vozil lahko pri tej nalogi spravimo na trajekt tako, da izberemo najkrajših nekaj vozil. Poudarek pri tej nalogi je na tem, da tekmovalec to opazi in po možnosti tudi dobro utemelji, ne pa na podrobnostih tega, kako njegova rešitev ta vozila dejansko poišče. Dovolj dobro je že, če rešitev pravi, da seznam dolžin vozil uredimo, ni pa treba, da govori o podrobnostih postopka za urejanje.

- 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). Glavna ideja takega argumenta je načeloma ta, da če smo uporabili neko daljše vozilo, nekega krajšega pa ne, lahko tisto daljše zamenjamo s tem krajšim in izbor vozil ostane dopusten (torej če prej skupna dolžina ni bila predolga, tudi zdaj ni). Seveda pa to ne pomeni, da pričakujemo formalen dokaz z indukcijo (in mogoče še s protislovjem), kakršne se ponavadi uporablja pri požrešnih algoritmih (recimo, da bi obstajal veljaven izbor s še več vozili; in potem z indukcijo dokažemo, da jo lahko predelamo v veljaven izbor, ki vsebuje najkrajših nekaj vozil, s tem pa pridemo v protislovje, saj je naša rešitev našla največji veljavni izbor take oblike).

4. Ledene dobe

- Glavno pri tej nalogi je, ali ima rešitev časovno zahtevnost $O(n \cdot m)$ ali le $O(n + m)$. Rešitve z zahtevnostjo $O(n \cdot m)$ naj dobijo največ 10 točk.
- Navodilo, da „v naslednjih 5000 letih nastopi ledena doba“, si je mogoče načeloma razlagati na različne načine: če je v letu x povečana koncentracija CO_2, v letu y pa ledena doba, lahko zapišemo ta pogoj kot $x \leq y \leq x+5000$ ali pa mogoče kot $x < y \leq x+5000$ ali pa $x \leq y < x+5000$ ali pa celo $x < y < x+5000$. Naša rešitev je uporabila prvo od teh možnosti, enako dobro pa je tudi, če tekmovalčeva rešitev uporabi katero od ostalih treh.
- Če se rešitev (po nepotrebnem) ukvarja z branjem vhodnih podatkov, naj se ji zaradi tega ne odbija točk.
- Če rešitev naredi kakšne predpostavke o največji možni dolžini vhodnih zaporedij, naj se ji odbije največ 5 točk.

5. Fora

- V naših rešitvah navajamo tri primere postopkov, ki rešijo nalogo in se pri tem sprehodijo skozi vhodno zaporedje le enkrat. Vseeno je, kaj od tega uporabi tekmovalčeva rešitev (ali pa še kak drug podoben postopek, ki se ga nismo domislili). Ni torej pomembno, ali rešitev pregleda vhodno zaporedje od začetka proti koncu ali od konca proti začetku ali še v kakšnem drugačnem vrstnem redu. (V nekem smislu je seveda boljša rešitev, ki pregleduje samo od začetka proti koncu, saj bi taka lahko obdelala tudi primere, ko je zaporedje v datoteki in ga ne moremo ali nočemo celega prebrati v pomnilnik, ker je predolgo; vendar pa nočemo, da bi bila naloga pretežka, zato bomo kot enako dobre obravnavali tudi rešitve, ki se sprehodijo po zaporedju v kakšnem drugačnem vrstnem redu kot od začetka proti koncu.)
- Želimo pa, da rešitev vhodnega zaporedja ne bi pregledala več kot enkrat. Če se po njem sprehodi dvakrat, naj dobi največ 15 točk, če še večkrat (npr. k-krat), pa največ 10 točk.

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. Ruleta	lahka naloga v prvi skupini
2. Glava e-pošte	srednje težka naloga v prvi skupini
3. Trajekt	lažja ali srednje težka naloga v drugi skupini
4. Ledene dobe	srednja naloga v drugi skupini
5. Fora	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, letos pa so se rezultati izšli tako, da smo v drugi skupini izjemoma podelili tri prve, eno drugo in tri tretje nagrade. Poleg nagrad na državnem tekmovanju v skladu s pravilnikom podeljujemo tudi zlata in srebrna priznanja. Število zlatih priznanj je omejeno na eno priznanje na vsakih 25 udeležencev šolskega tekmovanja, tako da smo jih letos podelili osem. Srebrna priznanja pa se podeljujejo po podobnih kriterijih kot v prejšnjih letih pohvale; prejmejo jih tekmovalci, ki ustrezajo naslednjim trem pogojem: (1) tekmovalec ni dobil zlatega priznanja; (2) je boljši od vsaj polovice tekmovalcev v svoji skupini; in (3) je tekmoval v prvi ali drugi skupini in dobil vsaj 20 točk ali pa je tekmoval v tretji skupini in dobil vsaj 80 točk. Namen srebrnih priznanj je, da izkažemo priznanje in spodbudo vsem, ki se po rezultatu prebijejo v zgornjo polovico svoje skupine. Podobno prakso poznajo tudi na nekaterih mednarodnih tekmovanjih; na primer, na mednarodni računalniški olimpijadi (IOI) prejme medalje kar polovica vseh udeležencev. Poleg zlatih in srebrnih priznanj obstajajo tudi bronasta, ta pa so dobili najboljši tekmovalci v okviru šolskih tekmovanj.

V spodnjih tabelah so prejemniki nagrad označeni z „1“, „2“ in „3“ v prvem stolpcu, prejemniki priznanj pa z „Z“ (zlato) in „S“ (srebrno).

PRVA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Nejc Grenc	3	Škof. klas. gimn. Lj.	18	19	17	15	15	84
1Z	2	Žiga Gosar	2	Gimnazija Vič	15	17	19	15	15	81
2S	3	Robi Maren	4	ERSŠG Ljubljana	14	19	15	15	15	78
2S	4	Marko Novak	1	ERSŠG Lj. + ZRI	20	0	20	15	20	75
3S	5	Denis Celcer	3	SERŠ Maribor	15	15	15	12	15	72
3S	6	Bojan Rajh	4	ERŠ Velenje	12	16	15	15	11	69
3S	7	Nejc Drašček Dekleva	4	ERSŠG Ljubljana	12	15	19	15	5	66
S	8	Tadej Kočnik	4	ERŠ Velenje	19	0	12	14	20	65
S	9	Aljaž Frančič	2	II. gimnazija Maribor	12	17	18	0	15	62
S	9	Gregor Miklošič	2	II. gimnazija Maribor	15	20	12	0	15	62
S	11	Erik Langerholc	1	Gimnazija Bežigrad	2	9	20	15	15	61
S	12	Rok Lampret	1	Škof. klas. gimn. Lj.	7	10	8	15	20	60
S	13	Ariadna Štorman	2	Gimnazija Vič	7	7	15	15	15	59
S	14	Tadej Petreski	2	II. gimnazija Maribor	8	15	14	5	15	57
S		Mihael Polanec	3	SERŠ Maribor	3	10	19	10	15	57
S		Jan Markočič	3	TŠC Nova Gorica	10	5	12	15	15	57
S	17	Matija Skala	1	ERSŠG Ljubljana	1	2	18	15	20	56

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
S	18	Tadej Ciglarič	1	Gimnazija Bežigrad	2	3	20	15	15	55
S		Uroš Kolenko	4	ERSŠG Ljubljana + ZRI	15	20	15	0	5	55
S	20	Živa Urbančič	1	ZRI	15	8	9	15	5	52
S		Klemen Drev	4	ERŠ Velenje	15	1	5	16	15	52
S	22	Leon Lončarič	1	Gimnazija Piran	10	2	11	14	13	50
S		Martin Davorin Kržišnik	1	Gimnazija Vič	12	0	8	15	15	50
S	24	Alec Smrekar	1	Gimnazija Piran	8	1	10	18	11	48
S	25	Tadej Vengust	3	SERŠ Maribor	0	2	8	13	20	43
S	26	Ines Meršak	1	Gimnazija Vič	7	0	15	5	15	42
S	27	Marko Balažic	3	SPTŠ Murska Sobota	2	1	0	18	20	41
S		Marion Antonia van Midden	3	SŠJJ Ivančna Gorica	1	10	0	15	15	41
S	29	Luka Hrvatin	3	Sr. teh. šola Koper	7	0	17	0	15	39
S	30	Jakob Kastelic	1	Gimnazija Bežigrad	8	5	8	4	12	37
S		Primož Mekuč	3	Škof. klas. gimn. Lj.	14	3	5	5	10	37
S	32	Jan Aleksandrov	9	OŠ Dravlje	8	0	12	0	15	35
S		Mitja Zidar	3	SŠJJ Ivančna Gorica	0	0	0	15	20	35
	34	Tadej Vajdl	3	ERŠ Velenje	4	1	10	8	11	34
	35	Rok Bevc	2	ERSŠG Ljubljana	15	0	0	1	15	31
		Marko Pranjić	4	ERŠ Velenje	1	1	12	3	14	31
		Sašo Markovič	2	II. gimnazija Maribor	2	10	2	6	11	31
	38	Marko Ljubotina	3	SŠJJ Ivančna Gorica	2	0	0	15	12	29
	39	Rene Vidonja	2	SPTŠ Murska Sobota	12	8	2	5	0	27
	40	Žiga Kajzer	3	SERŠ Maribor	11	0	0	0	15	26
	41	Robi Pritržnik	3	ERŠ Velenje	7	1	5	1	11	25
		Marko Vidic	2	Gimnazija Vič	4	2	4	10	5	25
	43	Tilen Pugelj	2	Gimnazija Vič	0	1	5	13	5	24
	44	Andrea Gostinčar	1	Gimnazija Vič	1	1	4	15	2	23
	45	Denis Smej	2	SPTŠ Murska Sobota	1	0	0	0	20	21
		Vito Meznarič	1	ERŠ Ptuj	5	1	1	3	11	21
		Nives Bricman	2	ERŠ Velenje	2	1	1	8	9	21
	48	Luka Pušić	2	Gimnazija Ledina	0	1	7	0	12	20
		Matej Lajh	3	ERŠ Ptuj	2	0	0	0	18	20
		Gal Kranjc	1	Gimnazija Vič	2	1	0	1	16	20
		Tadej Pregl	3	SERŠ Maribor	5	3	0	0	12	20
	52	Klemen Plazar	2	ERŠ Velenje	3	1	3	0	12	19
	53	Rajko Lukačič	4	Gimnazija Ormož	0	0	0	0	18	18
		Tadej Štadler	2	ERŠ Velenje	0	0	3	0	15	18
		Vid Peterson	2	ZRI	3	1	0	14	0	18
	56	Andrej Čuber	1	Gimnazija Piran	2	0	0	1	11	14
	57	Primož Godec	3	Škof. klas. gimn. Lj.	5	1	0	2	5	13
	58	Beno Šircelj	2	Sr. teh. šola Koper	10	0	0	0	0	10
	59	Jure Domajnko	2	SPTŠ Murska Sobota	8	0	0	0	0	8
	60	Tomaž Petek	2	ERŠ Velenje	2	1	3	0	0	6
	61	Natalija Špur	4	Gimnazija Ormož	0	0	0	0	5	5
		Grega Durmanič	4	Gimnazija Ormož	2	0	1	0	2	5
		Jan Keber	3	Sr. teh. šola Koper	0	0	0	0	5	5
		Tilen Božič	4	Sr. teh. šola Koper	0	0	0	0	5	5
	65	Blaž Vidovič	1	ERŠ Ptuj	3	0	1	0	0	4
		Fabian Meško	1	ERŠ Ptuj	0	1	3	0	0	4
	67	Rene Križman	2	Gimnazija Piran	0	0	0	0	0	0

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Gašper Medved	3	ERSŠG Ljubljana	20	18	20	20	3	81
1Z	1	Klemen Kloboves	3	ZRI	20	16	7	20	18	81
1Z	1	Ernest Beličič	3	ERSŠG Ljubljana	19	13	15	16	18	81
2S	4	Maks Kolman	2	Gimnazija Vič	16	11	20	15	15	77
3S	5	Sandi Majninger	3	II. gimnazija Maribor	15	7	20	19	13	74
3S	6	Rok Kaufman	2	Gimnazija Vič	19	7	19	14	14	73
3S	7	Rok Povšič	4	SEŠTG Novo Mesto	5	13	20	19	13	70
S	8	Urban Škvorc	2	Gimnazija Bežigrad	10	7	12	19	19	67
S	9	Jože Kulovic	4	ŠC Novo mesto	20	8	20	15	3	66
S	10	Simon Žlender	3	ERŠ Ptuj	20	4	10	15	16	65
S	11	Ludvik Zobec	5	Licej France Prešeren, Trst	15	5	12	20	12	64
S	11	Črt Jaklič	4	SEŠTG Novo Mesto	17	5	20	12	10	64
S	13	Jasna Urbančič	2	ZRI	20	14	20	2	5	61
S	13	Matej Vehar	3	ŠC Novo mesto, SEŠTG	12	9	10	14	16	61
S	15	Jurij Podgoršek	3	ERŠ Ptuj	20	17	20	0	0	57
S	15	Tadej Škvorc	2	Gimnazija Bežigrad	5	9	10	16	17	57
S	17	Matic Vrenko	2	ERSŠG Ljubljana	15	0	5	16	18	54
S	18	Aleš Razpotnik	3	ERSŠG Ljubljana	17	6	10	15	5	53
S	19	Boštjan Budna	2	Srednja šola Ravne	19	0	20	0	13	52
S	19	Robert Pajek	4	ŠCC Gimnazija Lava	5	12	7	15	13	52
S	21	Dejan Rjavec	4	ŠC Novo mesto, SEŠTG	5	12	3	18	12	50
S	22	Jernej Legiša	5	Licej France Prešeren, Trst	7	17	1	15	8	48
S	23	Miha Mohorčič	4	SEŠTG Novo Mesto	5	15	10	11	2	43
	24	Miran Višner	4	Srednja šola Ravne	19	0	19	0	4	42
	25	Albin Jordan	3	ŠC Novo mesto, SEŠTG	10	6	10	2	10	38
	26	Matej Korošec	4	SPTŠ Murska Sobota	3	7	20	2	5	37
	27	Martin Šušterič	2	ZRI in Gimnazija Bežigrad	0	15	0	15	5	35
	28	Dominik Letnar	4	SPTŠ Murska Sobota	8	10	12	2	2	34
	29	Tomaž Petrovič	3	ERŠ Ptuj	8	15	0	10	0	33
	29	Miran Helbl	3	Srednja šola Ravne	10	1	5	2	15	33
	31	Nejc Zupan	4	Gimnazija Vič	10	0	10	12	0	32
	31	Denis Trstenjak	4	SPTŠ Murska Sobota	17	0	10	2	3	32
	33	Aljaž Jesenko	4	ŠCC Gimnazija Lava	0	16	3	0	12	31
	34	Marko Kavaš	4	SPTŠ Murska Sobota	10	10	10	0	0	30
	34	Nejc Sever	4	SEŠTG Novo Mesto	10	0	20	0	0	30
	34	Blaž Kostanjšek	4	ERSŠG Ljubljana	0	0	20	0	10	30
	34	Tomaž Cebek	4	SERŠ Maribor	10	3	0	12	5	30
	34	Tomi Raguž	3	Sr. teh. šola Koper	0	0	20	0	10	30
	39	Jan Vidic	4	Gimnazija Vič	3	5	20	0	1	29
	39	Gašper Primožič	3	Srednja šola Ravne	5	0	10	2	12	29
	41	Kristian Žarn	3	ŠC Krško-Sevnica	0	0	20	0	0	20
	42	Mitja Suša	4	ŠC Novo mesto, SEŠTG	8	7	2	0	2	19
	43	Miha Novak	4	ERSŠG Ljubljana	2	0	5	2	5	14
	44	Oskar Korošec	3	ERŠ Ptuj	0	0	8	0	0	8
	45	Sandi Krivec	4	Srednja šola Ravne	5	0	0	0	0	5
	46	Blaž Celarc	2	ERSŠG Ljubljana	0	0	0	0	0	0

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Žiga Ham	4	ZRI in ERSŠG Lj.	100	90	60	94	100	444
1Z	2	Matjaž Leonardis	2	ZRI	100	0	10	80	100	290
2Z	3	Matej Aleksandrov	4	ZRI	97	0		80	97	274
2S	4	Peter Koželj	4	ZRI + Gim. Bežigrad	100	100		63		263
3S	5	Andraž Bajt	3	TŠC Nova Gorica	100	37	0	22	100	259
3S	6	Nejc Saje	4	SEŠTG Novo Mesto	94	40		0	100	234
S	7	Rok Kralj	4	Gimnazija Vič	100	91		20	0	211
S	8	Gašper Tomažič	4	TŠC Nova Gorica	92	100				192
S	9	Igor Lalič	4	ERSŠG Ljubljana	100			57	0	157
S	10	Tadej Novak	3	Gimnazija Kranj	94	37			0	131
	11	Jure Slak	2	Gimnazija Vič	100			20	4	124
	12	Matej Biberovič	3	Srednja šola Ravne	97					97
	13	Matija Rezar	4	Gimnazija Kranj	92		0		0	92
	14	David Ogorevc	3	ŠC Krško-Sevnica	82				0	82
	15	Blaž Sovdat	4	TŠC Nova Gorica	72					72
	16	Danijel Duraković	4	ERSŠG Ljubljana	5		0	0	0	5
	17	Blaž Pavlica	4	TŠC Nova Gorica	0			0		0
	17	Alen Roguljič	4	ŠC Krško-Sevnica	0					0
	17	Jaka Kramar	4	ŠC Krško-Sevnica		0				0
	17	Denis Kelbič	3	ŠC Krško-Sevnica	0					0
	17	Anže Novšak	4	ŠC Krško-Sevnica						0

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Nejc Grenc	16 GB iPod nano
1	1	Žiga Gosar	8 GB iPod nano
1	2	Robi Maren	8 GB iPod nano
1	2	Marko Novak	1.5 TB zunanji disk
1	3	Denis Celcer	1 TB zunanji disk
1	3	Bojan Rajh	1 TB zunanji disk
2	1	Gašper Medved	2 GB zunanji disk
2	1	Klemen Kloboves	8 GB iPod nano
2	1	Ernest Beličič	1.5 TB zunaji disk
2	2	Maks Kolman	1.5 TB zunaji disk
2	3	Sandi Majninger	1.5 TB zunaji disk
2	3	Rok Kaufman	1.5 TB zunaji disk
2	3	Rok Povšič	1 TB zunanji disk
3	1	Žiga Ham	2 TB zunanji disk Cormen *et al.: Introduction to algorithms*
3	1	Matjaž Leonardis	8 GB iPod nano Cormen *et al.: Introduction to algorithms*
3	2	Matej Aleksandrov	1.5 TB zunanji disk Cormen *et al.: Introduction to algorithms*
3	2	Peter Koželj	64 GB USB flash disk
3	3	Andraž Bajt	1.5 TB zunanji disk
3	3	Nejc Saje	1.5 TB zunanji disk
Tekmovanje programov — Tarok			
		Jure Slak	1 TB zunanji disk

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	Marko Kokol, Mirko Pešec
Elektrotehniško-računalniška strokovna šola in gimnazija (ERSŠG) Ljubljana	Matjaž Kodela, Nataša Makarovič, Darjan Toth
Gimnazija Bežigrad, Ljubljana	Andrej Šuštaršič, Jurij Železnik
Gimnazija Kranj	Matevž Jekovec
Gimnazija Ledina	Gregor Anželj
Gimnazija Ormož	Lenka Keček Vavpotič
Gimnazija Piran	Janez Urevc
Gimnazija Vič	Klemen Bajec, Andreja Likar Cerc, Marina Trost
Osnovna šola Dravlje	
Srednja elektro-računalniška šola Maribor (SERŠ)	Slavko Nekrep, Manja Sovič Potisk
Srednja poklicna in tehniška šola (SPTŠ) Murska Sobota	Simon Horvat, Karel Maček
Srednja šola Ravne	Gorazd Geč, Zdravko Pavleković
Srednja šola Josipa Jurčiča, Ivančna Gorica	Darko Pandur
Srednja tehniška šola Koper	Andrej Florjančič
Škofijska klasična gimnazija, Ljubljana	Helena Medvešek, Nace Hudobivnik
Šolski center Celje, Splošna in strokovna gimnazija Lava	Borut Slemenšek
Šolski center Krško-Sevnica	Svetlana Novak, Andrej Peklar
Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija	Mile Božič, Tomaž Ferbežar, Ivan Slinkar, Simon Vovko
Šolski center Ptuj, Elektro in računalniška šola	Marjan Čeh, Zoltan Sep, Franc Vrbančič
Šolski center Velenje, Elektro in računalniška šola	Gregor Hrastnik, Miran Zevnik
Tehniški šolski center Nova Gorica	Barbara Pušnar, Boštjan Vouk
Zavod za računalniško izobraževanje (ZRI), Ljubljana	Jelko Urbančič
Znanstveni licej France Prešeren, Trst	Valentina Busechian, Walter Auber

TEKMOVANJE PROGRAMOV — TAROK

Podobno kot v prejšnjih letih smo tudi letos organizirali tekmovanje programov. Opis naloge smo objavili novembra 2009 skupaj z razpisom za tekmovanje v znanju, tekmovalci pa so imeli čas do 12. marca 2010 (dva tedna pred tekmovanjem), da pošljejo svoje programe. Letošnja naloga je od tekmovalcev zahtevala, da napišejo logiko za igranje taroka s precej poenostavljenimi pravili.

Igralci, pripomočki. Tarok je igra s kartami. Igrajo jo štirje igralci s 54 kartami: 22 tarokov, oštevilčenih od 1 do 22, in še po 8 kart vsake barve (srce, karo, križ, pik): 7, 8, 9, 10, fant, kaval, kraljica, kralj.

Predpriprava. Strežnik naključno premeša karte. Šest jih zadrži (tem kartam pravimo *talon*), po 12 pa jih razdeli vsakemu igralcu. Strežnik določi tudi vrstni red, v katerem igralci „sedijo za (okroglo) mizo“.

Tipi igre. Vsako igro je eden od igralcev „glavni“. Ta igralec se odloči za enega od kraljev (srce, karo, pik, križ) in za enega od šestih tipov igre: 3, 2, 1, solo 3, solo 2 ali solo 1.

Glavnega igralca se določi tako, da vsak igralec pove, kakšen tip igre želi igrati, nato pa glavni igralec postane tisti, ki je predlagal navišje točkovan tip igre (glej razdelek o točkovanju spodaj). Če je takšnih igralcev več, glavni postane tisti, ki v sedežnem redu nastopa prej.

Če se odloči za igro „3“, se karte iz talona razdeli v dve skupini po 3 karte; glavni igralec pred začetkom igre vzame poljubno izmed teh dveh skupin v roke (jih doda svojim 12 kartam) ter poljubne 3 karte iz svoje roke (razen kraljev) pospravi na svoj kupček. (V tej fazi igre sicer še nobeden od igralcev nima svojega kupčka; na kupčku nabirajo karte, ki jih pridobijo med rednim delom igre.)

Igralci se nato organizirajo v dve skupini. Igralec, ki ima v roki kralja, ki si ga je izbral glavni igralec, igra v skupini z glavnim igralcem, preostala dva igralca pa tvorita drugo skupino. Če je bil izbrani kralj v talonu, igra glavni igralec sam, preostali trije igralci pa tvorijo drugo skupino.

Podobni sta igri „2“ in „1“, le da se talon razdeli na skupine s po dvema oz. po eno karto, zato glavni igralec dobi le dve oz. eno karto in zato tudi na kupček pospravi le dve oz. eno karto. V „solo“ različicah iger vsa pravila ostanejo enaka, le da glavni igralec ne izbere kralja, temveč igra sam proti vsem trem igralcem.

Redni del igre. Igra ima 12 krogov. Prvi krog začne prvi igralec po sedežnem redu. Preostale kroge začne tisti, ki je pobral karte v predhodnem krogu.

Igralec, ki začne krog, vrže poljubno karto. V vrstnem redu, določenim na začetku igre, vržejo po eno karto še preostali igralci. Če je prvi igralec vrgel netaroka, mora vsak od preostalih igralcev vreči karto enake barve; če takšne karte nima, mora vreči taroka; če pa nima niti taroka niti karte enake barve, lahko vrže poljubno karto. Če pa je prvi igralec vrgel taroka, mora tudi vsak od preostalih igralcev vreči taroka, če ga le ima, in poljubno karto sicer. Na koncu kroga vse štiri karte pobere in spravi na svoj kupček igralec, ki je vrgel najvišjo karto. Vsak tarok pobere vsakega netaroka. Netaroki, ki so enake barve kot prva v krogu vržena karta, poberejo netaroke drugih barv. Med kartami iste barve je kralj najvišja karta, 7 pa najnižja.

Točkovanje. Igralci znotraj vsake skupine združijo kupčke. Kupčka se ovrednotita tako, da se seštejejo vrednosti kart v kupčku. Naj bo a vrednost kupčka tiste skupine,

v kateri je glavni igralec, b pa vrednost kupčka druge skupine. Potem vsi igralci skupine, v kateri igra glavni igralec, dobijo točke po formuli $|a-b|/2+(\textit{vrednost tipa igre})$, če je $a > b$; v nasprotnem primeru (torej če je $a \leq b$) pa takšno število točk izgubijo.

Vrednosti tipov iger so:

Tip igre	Vrednost igre
3	30
2	60
1	90
solo 3	120
solo 2	150
solo 1	180

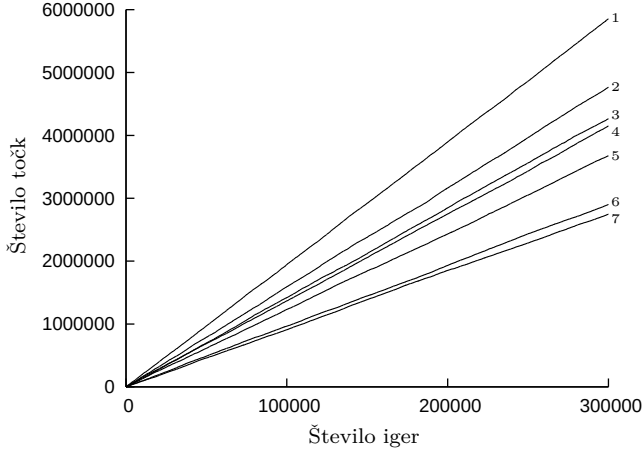
Vrednosti kart:

Karta	Vrednost
kralj	13
kraljica	10
kaval	7
fant	4
taroki 1, 21 in 22	13
ostale karte	1

Rezultati

Za nalogo je bilo precej zanimanja in na koncu smo prejeli rešitve sedmih tekmovalcev; eden je bil dijak, ostali pa študentje FRI. Za ocenjevanje smo izvedli približno 300 000 iger in to tako, da so nastopile vse možne kombinacije štirih tekmovalcev enako pogosto. Končni rezultati (skupno število točk po vseh igrah) so:

Mesto	Ime	Letnik	Šola	Točke
1	Jure Slak	2	Gimnazija Vič	6 073 483
2	Žan Kafol	3	FRI	4 936 612
3	Tomaž Kariž	3	FRI	4 425 627
4	Primož Kariž	3	FRI	4 298 834
5	Klavdij Oberstar	2	FRI	3 807 152
6	Domen Mladovan	3	FRI	3 008 550
7	Vid Tadel	3	FRI	2 839 440



Skupno število točk v odvisnosti od preigranega števila iger.
1 = Jure, 2 = Žan, 3 = Tomaž, 4 = Primož, 5 = Klavdij, 6 = Vid, 7 = Domen

UNIVERZITETNI PROGRAMERSKI MARATON

Društvo ACM Slovenija sodeluje tudi pri pripravi študenskih tekmovanj v programiranju, ki v zadnjih letih potekajo pod imenom Univerzitetni programerski maraton (UPM, `www.upm.si`) in so odskočna deska za udeležbo na ACMovih mednarodnih študentskih tekmovanjih v programiranju (International Collegiate Programming Contest, ICPC). Ker UPM ne izdaja samostojnega biltena, bomo na tem mestu na kratko predstavili to tekmovanje in njegove letošnje rezultate.

Na študentskih tekmovanjih ACM v programiranju tekmovalci ne nastopajo kot posamezniki, pač pa kot ekipe, ki jih sestavljajo po največ trije člani. Vsaka ekipa ima med tekmovanjem na voljo samo en računalnik. Naloge so podobne tistim iz tretje skupne našega srednješolskega tekmovanja, le da so včasih malo težje oz. predvsem predpostavljajo, da imajo reševalci že nekaj več znanja matematike in algoritmov, ker so to stvari, ki so jih večinoma slišali v prvem letu ali dveh študija. Časa za tekmovanje je pet ur, nalog pa je praviloma 6 do 8, kar je več, kot jih je običajna ekipa zmožna v tem času rešiti. Za razliko od našega srednješolskega tekmovanja pri študentskem tekmovanju niso priznane delno rešene naloge; naloga velja za rešeno šele, če program pravilno reši vse njene testne primere. Ekipe se razvrsti po številu rešenih nalog, če pa jih ima več enako število rešenih nalog, se jih razvrsti po času oddaje. Za vsako uspešno rešeno nalogo se šteje čas od začetka tekmovanja od uspešne oddaje pri tej nalogi, prišteje pa se še po 20 minut za vsako neuspešno oddajo pri tej nalogi. Tako dobljeni časi se seštejejo po vseh uspešno rešenih nalogah in ekipe z istim številom rešenih nalog se potem razvrsti po skupnem času (manjši ko je skupni čas, boljša je uvrstitev).

UPM poteka v štirih krogih (dva spomladi in dva jeseni), pri čemer se za končno razvrstitev pri vsaki ekipi zavrže najslabši rezultat iz prvih treh krogov, četrti (finalni) krog pa se šteje dvojno. Najboljše ekipe se uvrstijo na srednjeevropsko regijsko tekmovanje (CERC, tokrat v Wrocławu na Poljskem), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki je bilo tokrat v Harbinu na Kitajskem).

Na letošnjem UPM je sodelovalo 23 ekip s skupno 61 tekmovalci, ki so prišli z vseh treh slovenskih univerz, nekaj pa je bilo celo srednješolcev. Tabela na naslednji strani prikazuje ekipe, ki so rešile vsaj eno nalogo.

	Ekipa	Št. rešenih nalog	Čas
1	Tomaž Hočevar (FRI), Gašper Zadnik (FMF), Matjaž Leonardis (Gim. Bežigrad)	26	32:42:27
2	Žiga Ham (ERSŠG / FRI), Nace Hudobivnik (FMF), Matej Aleksandrov (Gim. Bežigrad / FMF)	21	30:28:38
3	Primož Koželj, Nino Bašić, Jan Berčič (FMF)	21	33:42:56
4	Peter Koželj (Gim. Bežigrad / FMF), Klemen Kloboves (Gim. Škofja Loka)	9	14:39:08
5	Ernest Beličič, Aleš Razpotnik, Gašper Medved (ERSŠG)	8	17:59:37
6	Aleksander Kelevc, Tim Kos, Matej Hren (FNM Maribor)	7	9:43:06
7	Matic Potočnik, Nino Ostrc (FRI)	6	3:55:12
8	Andrej Cimperšek, Uroš Vovk, Nastja Maršič (FAMNIT)	6	6:31:12
9	Jure Slak, Žiga Gosar, Maks Kolman (Gim. Vič)	6	6:38:50
10	Gašper Černevšek, Neža Čeč, Dean Gostiša (FRI)	6	6:56:11
11	Jernej Štrasner, Matevž Črnilogar (FAMNIT)	6	9:14:38
12	Žan Kafol, Primož Kariž, Tomaž Kariž (FRI)	6	9:21:54
13	Aleksandar Tošić, Duško Topić, Simon Mezgec (FAMNIT)	6	13:53:20
14	Rok Kralj (Gim. Vič)	4	2:29:49

Na srednjeevropsko tekmovanje so se uvrstile ekipe 1 (brez Matjaža Leonardisa, ki je bil še srednješolec), 2 in 3 kot predstavnice Univerze v Ljubljani, 6 kot predstavnica Univerze v Mariboru in 8 kot predstavnica Univerze na Primorskem.

ANKETA

Tekmovalcem vseh treh skupin smo na tekmovanju skupaj z nalogami razdelili tudi naslednjo anketo. Rezultati ankete so predstavljeni na str. 137–143.

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++, pythonu in javi.
— Ali razumeš kakšnega od teh jezikov dovolj dobro, da razumeš te deklaracije v besedilu naših nalog? ☐ da ☐ ne
— So ti prišle deklaracije v pythonu kaj prav? ☐ da ☐ ne
— Ali bi raje videl, da bi objavljali deklaracije (tudi) v kakšnem drugem programskem jeziku? Če da, v katerem? _____

V rešitvah nalog trenutno objavljamo izvorno kodo v C-ju.
— Ali razumeš C 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 1. in 2. skupini.] Kakšno je tvoje mnenje o sistemu za oddajanje odgovorov prek računalnika? _____

_____

[Le pri 3. skupini.] Letos v tretji skupini podpiramo reševanje nalog v pascalu, C, C++, C# 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	☐	☐	☐
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 33 tekmovalcev prve skupine in 12 tekmovalcev druge skupine; rezultatov ankete v tretji skupini pa žal nimamo, ker so se anketni listi tretje skupine izgubili neznano kam. Vprašanja so bila pri letošnji anketi približno enaka kot lani, dodali pa smo vprašanje za tekmovalce 1. in 2. skupine o tem, kakšen se jim je zdel sistem za oddajo odgovorov prek računalnika.

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. 138. Tam so tudi podatki o povprečnem številu točk, doseženem pri posamezni nalogi, tako da lahko primerjamo mnenje tekmovalcev o zahtevnosti naloge in to, kako dobro so jo zares reševali.

V povprečju so se zdele tekmovalcem v vseh skupinah naloge še kar težke, vendar malo lažje kot lani. 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: 3,34 v prvi skupini (lani 3,56; predlani 3,57) in 3,38 v drugi skupini (lani 3,46; predlani 3,62).

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 ($R^2 = 0{,}39$).

Največ pripomb o tem, kako da je naloga težka, je bilo pri nalogah 1.3 (skrivanje tipk) in 2.2 (reka presledkov); ponavadi sicer naloge o nizih in besedilu tekmovalcem niso bile tako težke, ampak letos je očitno drugače. Težka se jim je zdela 2.1 (poštne številke), mogoče zato, ker pri njej pride prav rekurzija, ta pa mnogim tekmovalcem dela težave.

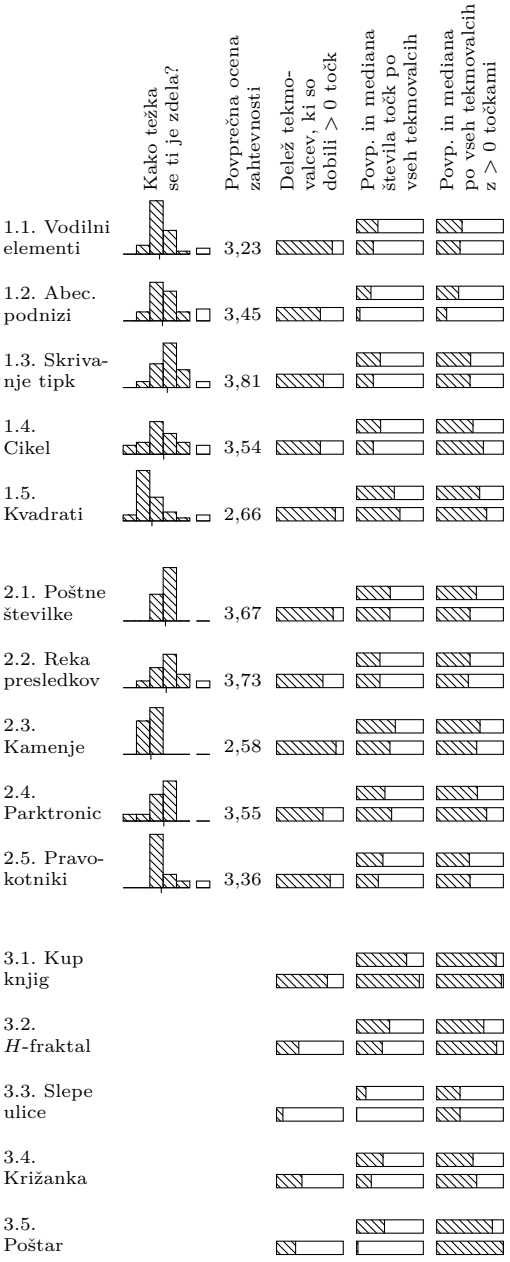
Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 139. Nad razumljivostjo besedil ni veliko pripomb (celo malo manj kot lani); najtežje razumljive so se jim zdele naloge 1.4 (cikel), 2.4 (parktronic) in 2.5 (pravokotniki); 2.4 mogoče zato, ker je realnočasovna in so take tekmovalcem vedno malo težje.

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, še bolj kot lani. Pripombe, češ da je besedilo prekratko, in tiste, češ da je predolgo, so letos bolj uravnotežene kot prejšnja leta.

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot lani, v povprečju malo nižje. Več pripomb glede dolgočasnosti nalog je bilo v drugi skupini, še posebej pri nalogi 2.3 (delitev kamenja), za katero so nekateri pripomnili tudi, da jim je že znana.

Pripomb, da bi naloga vzela preveč časa, je bilo manj kot lani, še največ pa jih je bilo pri je nalogah 1.1 (vodilni elementi), 1.3 (skrivanje tipk) in 2.1 (poštne številke). Pri 1.1 in 2.1 nas ta rezultat pravzaprav preseneča, saj se nam ni zdelo, da bi ti dve nalogi zahtevali kakšno posebej zamudno rešitev ali dolgotrajen razmislek.

Glasovi o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, kažejo letos izrazito zmagovalko v prvi skupini: naloga 1.5 (kvadrati s seštevanjem) je



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

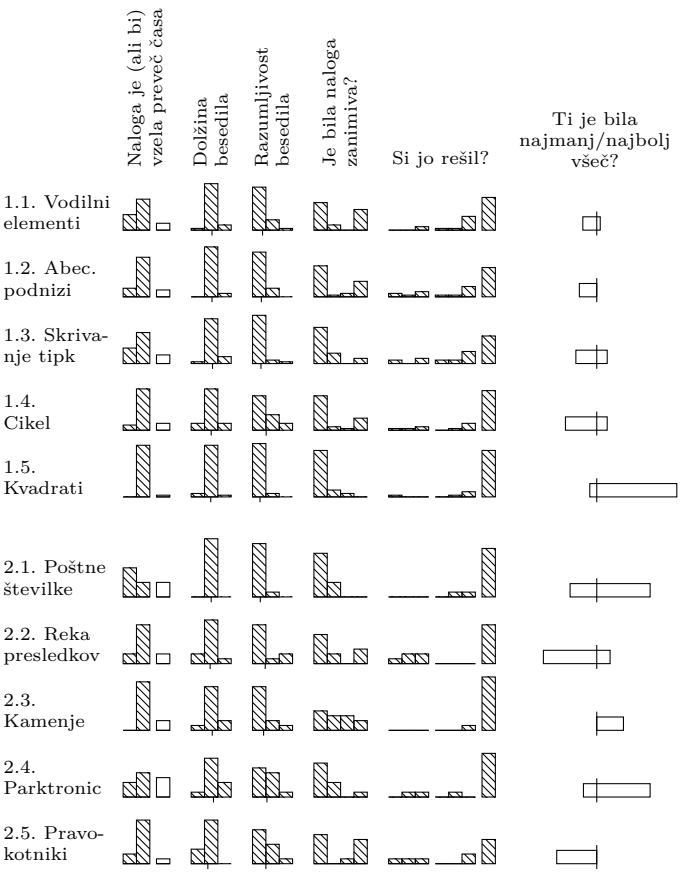
Pomen stolpcev v vsaki vrstici:

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?

daleč najbolj priljubljena (ta naloga se jim je zdela tudi najlažja, tako da njena priljubljenost ni presenetljiva). V drugi skupini so glasovi bolj razpršeni med več nalog. Priljubljeni sta bili še nalogi 2.1 (poštne številke) in 2.4 (parktronic), nepriljubljeni pa 1.4 (cikel) in 2.2 (reka presledkov).

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.

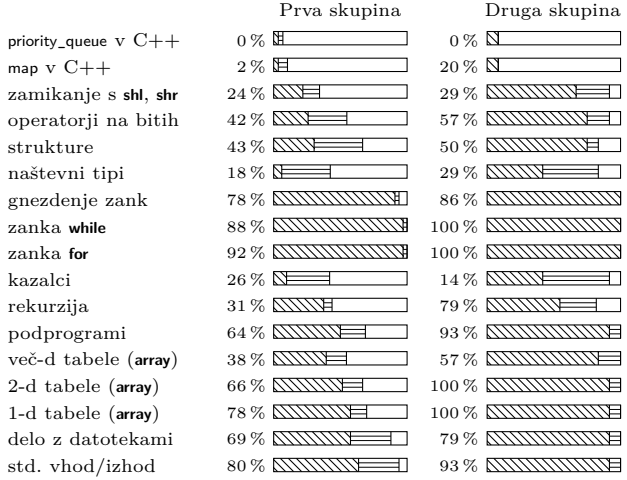


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

Rezultati pri vprašanjih o programerskem znanju so zelo podobni lanskim. 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 1. skupini manj ljudi kot lani, v 2. pa približno enako; je pa vprašljivo, ali smemo iz takšnih sprememb vleči kakšne zaključke, saj je število anketiranih tekmovalcev precej majhno (še posebej v drugi skupini).

Uporaba programskih jezikov

Velika večina tekmovalcev tudi letos uporablja C in C++ (slednji prevladuje nad navadnim C-jem še bolj kot lani), uporaba jave je malo upadla, več kot lani pa je uporabnikov pythona. Pascal se drži na približno istem nivoju kot lani. Podobno kot lani se je tudi letos pojavilo nezanemarljivo število tekmovalcev (in tekmovalk), ki oddajajo le rešitve v psevdokodi ali pa celo naravnem jeziku, tudi tam, kjer naloga sicer zahteva izvorno kodo v kakšnem konkretnem programskem jeziku. Po eni strani to mogoče pomeni, da bi morali dati več nalog tipa „opiši postopek“ in manj „napiši

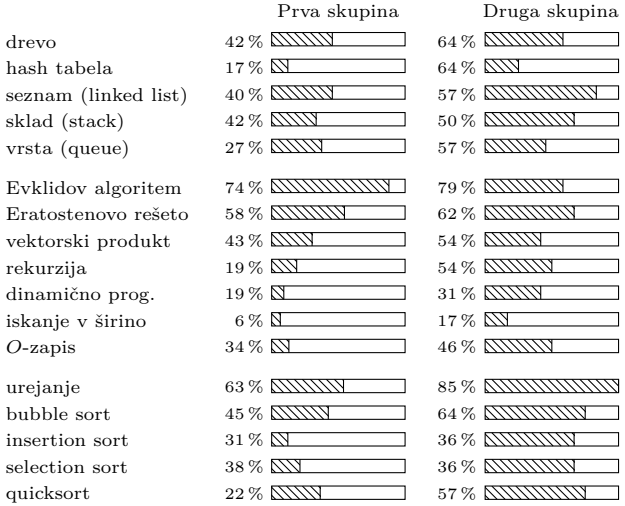


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

---

podprogram“ (letos so bile v 1. skupini štiri naloge tipa „napiši podprogram“); po drugi strani pa se v praksi izkaže, da so odgovori tekmovalcev pri nalogah tipa „opiši postopek“ pogosto precej nejasni, tako da jih je težje ocenjevati.

Podobno kot v prejšnjih letih je v anketi precej tekmovalcev napisalo, da dobro poznajo tudi PHP, vendar sta ga na tekmovanju uporabljala le dva.

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže tabela na str. 142. 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 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 (26/33 v prvi skupini in 11/12 v drugi). Presenetljivo je, da nekaj tekmovalcev posebej želi deklaracije v C++; nekatere mogoče pri obstoječih deklaracijah zmede to, da nize podajamo kot **char *** namesto string.

V rešitvah nalog od lani naprej objavljamo izvorno kodo le v C-ju; tekmovalce smo v anketi vprašali, če razumejo C dovolj, da si lahko kaj pomagajo s to izvorno kodo, in če bi radi videli izvorno kodo rešitev še v kakšnem drugem jeziku. Večina je s C-jem zadovoljna (18/33 v prvi skupini, 10/12 v drugi), vendar je v prvi skupini zdaj že kar skoraj polovica ljudi, ki pravijo, da ga ne razumejo. Med jeziki, ki bi jih radi videli namesto C-ja, jih največ omenja C++ in python, za javo pa je bil letos en sam glas (lani jih je bilo precej).

Kot kažejo ti rezultati, ni ravno očitno, kateri jezik bi bilo pametno v rešitvah

	Leto in skupina																		
	2010			2009			2008			2007			2006			2004			2003
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	3
pascal	$4\frac{1}{2}$	5	2	4	2	1	$1\frac{1}{2}$	2	2	$8\frac{1}{2}$	2	1	6	5	5	23	20	13	17
C	6	6	1	$9\frac{1}{2}$	$3\frac{1}{2}$	$\frac{1}{2}$	$4\frac{1}{2}$	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++	33	$17\frac{1}{2}$	13	$26\frac{1}{2}$	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	5	9	4	8	8	11	$9\frac{1}{2}$	3			$2\frac{1}{2}$				3		$\frac{1}{2}$		–
PHP	1	1	–	2	1	–		2	–	1		–	1		–			–	–
basic			–			–			–		1	–		1	–			–	–
C#		$\frac{1}{2}$	1						3		$\frac{1}{2}$	–			–			–	–
python	12	2	–	4	$\frac{1}{2}$	–	6	1	–			–			–			–	–
psevdokoda	4		–	8		–	–	–	–	–	–	–	–	–	–	–	–	–	–
nič	1	5	–	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“.

uporabiti poleg C-ja (ali pa celo namesto C-ja), saj se glasovi iz leta v leto precej spreminjajo. Koristno bi bilo poizvedeti, kaj je pravzaprav tisto, zaradi česar nekateri rešitev v C-ju ne razumejo in hkrati pravijo, da bi rešitve v C++u znali razumeti.

Letnik

Po pričakovanjih so tekmovalci zahtevnejših skupin v povprečju v višjih letnikih kot tisti iz lažjih skupin. Razmerja so podobna kot lani, v povprečju pa so se tekmovalci letos rahlo pomladili, še posebej v prvi skupini.

	Št. tekmovalcev po letnikih				Povprečni
Skupina	1	2	3	4	letnik
prva	17	20	18	11	2,3
druga		9	15	20	3,3
tretja		2	5	14	3,6

Druga vprašanja

Podobno kot lani je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!). V smislu širitve zanimanja za tekmovanje in večanja števila tekmovalcev se zelo dobro obnese šolsko tekmovanje, ki ga izvajamo zadnjih nekaj let, saj se odtlej v tekmovanje vključuje tudi nekaj šol, ki prej na našem državnem tekmovanju niso sodelovale.

Pri vprašanju, kje so se naučili programirati, podobno kot lani prevladujeta odgovora „sam“ in „v šoli“; obojih je približno enako, novost pa je letos precejšnje število tekmovalcev prve skupine, ki pravijo, da so se programirati naučili na krožkih.

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni. Nekaj je tudi tekmovalcev, ki želijo manj nalog in/ali več časa.

Iz odgovorov na vprašanje, kakšne potekmovalne dejavnosti bi jih zanimale, je težko zaključiti kaj posebej konkretnega.

	Kje si izvedel za tekmovanje				Kje si se naučil programirati					Čas reševanja			Število nalog			Potekmovalne dejavnosti							
Skupina	od mentorja	na spletni strani	od prijatelja/sošolca	drugače	sam	pri pouku	na krožkih	na tečajih	poletna šola	hočem več časa	hočem manj časa	je že v redu	hočem več nalog	hočem manj nalog	je že v redu	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

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb. Od lani imajo tekmovalci v prvi in drugi skupini možnost pisati svoje odgovore na računalniku namesto na papir (kar so si prej v anketah že večkrat želeli). Letos so vsi razen enega oddajali odgovore na računalniku. V anketo smo dodali tudi vprašanje o tem, kakšen se jim je zdel sistem za oddajo nalog; z njim so bili večinoma zadovoljni, so pa opozorili na nekatere tehnične težave (bili so celo primeri, ko se je del odgovora pri oddaji izgubil).

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) Nek tekmovalec je deklariral približno sto spremenljivk in bral vhodne podatke vanje takole:

```
a, b, c, d, ..., z, aa, ab, ..., cy = sys.stdin.readline().split()
a=int(a); b=int(b); c=int(c); d=int(d); e=int(e), f=int(f) ... # itd do cy (vključno s cy)
tabela=[a, b, c, d, e, f, g, h...] # spet vse do cy
for zz in tabela:
  print zz
```

(1.1) Koristen stavek:

```
if (j < j + 1) j++;
```

(1.1) Še boljši primer:

```
if v > v + 1:
  z = z + 1
  v = v + 1
else :
  v = v + 1
```

(1.2) Iz ene od rešitev:

```
najdalsi = 0 # najdaljši je sicer moj
```

(1.3) Rešitev z izgovarjanjem na pomanjkljivosti standardne knjižnice:

```
/* ta rešitev ni pravilna zaradi tega ker funkcija find() vrne za kjer koli v nizu, ne
   samo od začetka */
```

Mišljena je string::find v C++; ampak saj ta poišče prvo pojavitev iskanega podniza in vrne indeks, na katerem se začne ta pojavitev; če nas torej zanimajo pojavitve le na začetku, bi bilo dovolj preveriti, če find vrne 0.

(1.4) Komentarji na začetku ene od rešitev:

```
/* more bit pa hudo pametn vratar da si lahko zapomni tudi zelo velike številke; npr.
   kaj če je plesalcev 1000? in on si vse zapomni :0
   kako je lahko tako pameten človek le vratar... */
```

(1.4) Nekdo si je navodilo „opiši postopek“ razlagal tako, kot da je treba opisati, kako napišemo izvorno kodo programa, ki reši problem iz naloge:

> Najprej dodamo v program standardno knjižnico in uporabimo std input in output. Začnemo program z int main. Najprej bi deklariral vse potrebne integer-je in polja.

Še dobro, da ni predlagal tudi system("PAUSE") na koncu :P

(1.4) Geometrijski pristop k nalogi:

zato, da bi lahko vsak udeleženec zgrabil natanko enega soudeleženca zabave, morajo plesati v krogu oz. neki sklenjeni formaciji (tvoriti morajo konveksno množico/lik)

(1.5) Prispevek za rubriko „tekmovalci čestitajo in pozdravljajo“:

```
# Rad bi se zahvalil Juretu Slaku, ker me je naučil Pythona. Upam, da pride to v Cvetke.
```

(1.5) Eden od tekmovalcev se je skušal množenju izogniti z uporabo funkcije za potenciranje iz standardne knjižnice:

```
//Sem pozabil funkcijo za potenciranje... V kodi sem jo označil kakor „pwr“
⋮
x = pwr(n,2);
cout << x << endl;
```

Pravo ime te funkcije je sicer pow, ampak v vsakem primeru to ni rešitev, ki smo jo imeli v mislih pri tej nalogi.

(2.1) Letošnjo nagrado za najglobljo indentacijo dobi:

```
                        Write(t1, s1, d1, e1, ' ')
                    e1 := e1 + 1;
                until e1 = 10;
            d1 := d1 + 1;
        until d1 = 10;
    s1 := s1 + 1;
until s1 = 10;
t1 := t1 + 1;
until t1 = 10;
end;
```

Vsi ti nivoji indentacije so bili tudi res upravičeni, saj je bila vsaka zanka zavita v pogoj (**if** pogoj **then repeat** ... **until**).

(2.1) Zanimiv način, kako izluščiti enice iz števila:

```
cetrta[3] = k % 1000 % 100 % 10;
```

(2.1) Pri tej nalogi smo dobili tudi najdaljšo prejeto rešitev letošnjega tekmovanja; ima okoli 380 vrstic, deluje pa tako, kot da bi imel rekurzijo in nato izvorno kodo eksponentno razpihnil, tako da rekurzija ni več potrebna. Je pa to uspel narediti le delno; če bi s tem nadaljeval, bi bila koda še bistveno daljša.

Tudi nekaj drugih tekmovalcev je oddajalo tu zelo dolge rešitve, po sto vrstic in več, eden celo okoli 320 vrstic. To je ena od slabosti tekmovanja na računalnikih; če bi bili prisiljeni pisati na papir, bi jih to mogoče pripeljalo do razmisleka o elegantnejši rešitvi.

(2.1) Odlična ideja za nov primerjalni operator:

```
if (a || b || c || d == 3 || 8 || 1 || 4 || 7 || 5 || 6)
```

(2.3) Nek tekmovalec je celotno rešitev zapisal kot pismo butalskemu županu.

> Še nasvet za konec — če Tepanjčani ne bodo iz vaše vasi odpeljali kamna jim zaračunajte za to, da kamenje stoji na vaši zemlji in morda še zaslužite in če ne plačajo, si lahko kamenje prisvojite.

(2.4) Podprogram, ki nima veliko dela:

```
void main()
{
  char x = '0';
  if (Cas() == Cas() + 1)
  {
    ⋮
  }
}
```

Tudi spremenljivke x nikjer ne uporabi.

(2.5) Ena od slabosti nalog tipa „opiši postopek“ je, da so odgovori včasih zelo megleni:

> Nalogo rešujemo z for in if stavki. Z for povečujemo x in y koordinate, ki jih z if-i preverjamo. Ustvarimo temp spremenljivko v kateri bomo shranili trenutno spremenljivko, ki jo bomo nato izpisali.

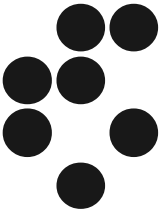
(Nadaljuje se v približno istem slogu.)

(3.1) Pogoji, ki redko pridejo prav:

```
if (n <= 1 && n >= 1000)
  return 0;
else if (k <= 1 && k >=1000000)
  return 0;
```

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.

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.

Fakulteta za elektrotehniko, računalništvo in informatiko

Fakulteta za elektrotehniko, računalništvo in informatiko (FERI) je znanstveno-izobraževalna institucija z izraženim regionalnim, nacionalnim in mednarodnim pomenom. Regionalnost se odraža v tesni povezanosti z industrijo v mestu Maribor in okolici, kjer se zaposluje pretežni del diplomantov dodiplomskih in podiplomskih študijskih programov. Nacionalnega pomena so predvsem inštituti kot sestavni deli FERI ter centri znanja, ki opravljajo prenos temeljnih in aplikativnih znanj v celoten prostor Republike Slovenije. Mednarodni pomen izkazuje fakulteta z vpetostjo v mednarodne raziskovalne tokove s številnimi mednarodnimi projekti, izmenjavo študentov in profesorjev, objavami v uglednih znanstvenih revijah, nastopih na mednarodnih konferencah in organizacijo le-teh.

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Fakulteta za matematiko, naravoslovje in informacijske tehnologije Univerze na Primorskem (UP FAMNIT) je prvo generacijo študentov vpisala v študijskem letu 2007/08, pod okriljem UP PEF pa so se že v študijskem letu 2006/07 izvajali podiplomski študijski programi Matematične znanosti in Računalništvo in informatika (magistrska in doktorska programa).

Z ustanovitvijo UP FAMNIT je v letu 2006 je Univerza na Primorskem pridobila svoje naravoslovno uravnoteženje. Sodobne tehnologije v naravoslovju predstavljajo na začetku tretjega tisočletja poseben izziv, saj morajo izpolniti interese hitrega razvoja družbe, kakor tudi skrb za kakovostno ohranjanje naravnega in družbenega ravnovesja. K temu bo fakulteta v prihodnjih letih (2009–2013) z razvojem kakovostnega oblikovanja in izvajanja naravoslovnih študijskih programov tudi stremela. V tem matematična znanja, področje informacijske tehnologije in druga naravoslovna znanja predstavljajo ključ do odgovora pri vprašanjih modeliranja družbeno ekonomskih procesov, njihove logike in zakonitosti racionalnega razmišljanja.

ACM Slovenija

ACM je največje računalniško združenje na svetu s preko 80 000 člani. ACM organizira vplivna srečanja in konference, objavlja izvirne publikacije in vizije razvoja računalništva in informatike.

ACM Slovenija smo ustanovili leta 2001 kot slovensko podružnico ACM. Naš namen je vzdigniti slovensko računalništvo in informatiko korak naprej v bodočnost.

Društvo se ukvarja z:

- Sodelovanjem pri izdaji mednarodno priznane revije Informatica — za doktorande je še posebej zanimiva možnost objaviti 2 strani poročila iz doktorata.
- Urejanjem slovensko-angleškega slovarčka — slovarček je narejen po vzoru Wikipedije, torej lahko vsi vanj vpisujemo svoje predloge za nove termine, glavni uredniki pa pregledujejo korektnost vpisov.
- ACM predavanja sodelujejo s Solomonovimi seminarji.
- Sodelovanjem pri organizaciji študentskih in dijaških tekmovanj iz računalništva.

ACM Slovenija vsako leto oktobra izvede konferenco Informacijska družba in na njej skupščino ACM Slovenija, kjer volimo predstavnike.

IEEE Slovenija

Inštitut inženirjev elektrotehnike in elektronike, znan tudi pod angleško kratico IEEE (Institute of Electrical and Electronics Engineers) je svetovno združenje inženirjev omenjenih strok, ki promovira inženirstvo, ustvarjanje, razvoj, integracijo in pridobivanje znanja na področju elektronskih in informacijskih tehnologij ter znanosti.

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

ADACTA

CONSIDER IT DONE.

arnes

cosylab

CONTROL SYSTEM LABORATORY

Fortheia

Pametne rešitve

XLAB

NOT IDLE