5. tekmovanje ACM in IJS v znanju računalništva za srednješolce

27. marca 2010

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.");
   }
}
```

5. tekmovanje ACM in IJS v znanju računalništva za srednješolce

27. marca 2010

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.

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

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. 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
```

5. tekmovanje ACM in IJS v znanju računalništva za srednješolce

27. marca 2010

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.

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

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.

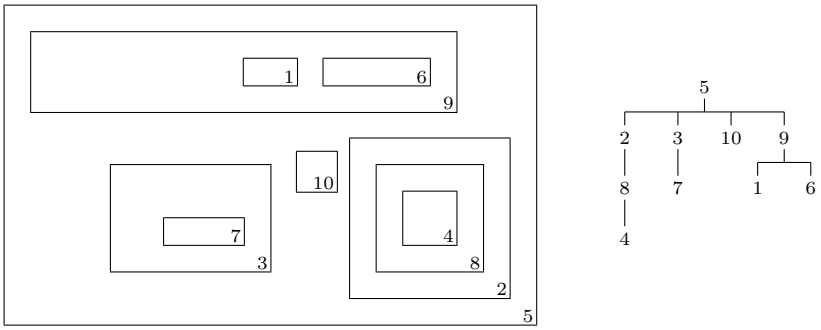
Še deklaracije v drugih jezikih:

```
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.

5. tekmovanje ACM in IJS v znanju računalništva za srednješolce

27. marca 2010

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` (Free Pascal), `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;
}
```

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

- V javi:

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

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

- V C#:

```
using System.IO;

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

5. tekmovanje ACM in IJS v znanju računalništva za srednješolce

27. marca 2010

NALOGE ZA TRETJO SKUPINO

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

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 $1 \leq n \leq 1000$ in $1 \leq k \leq 1\,000\,000$. 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 $k \leq 2000$.

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:

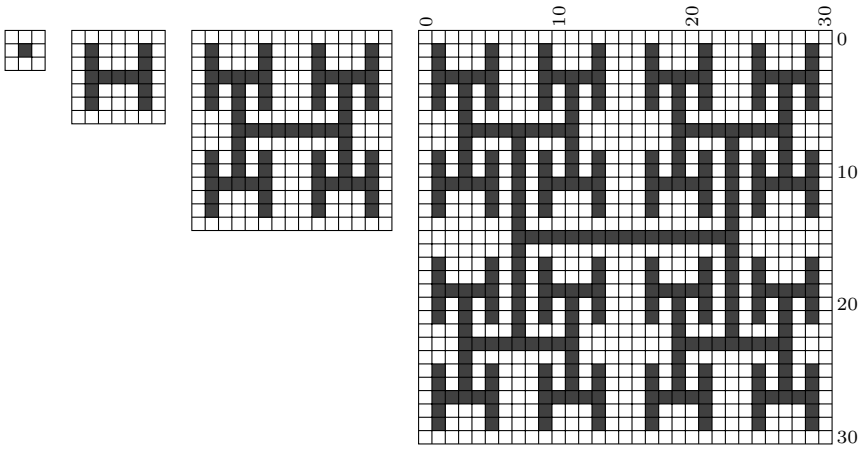
```
4 3
2
3
2
```

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. Spodnja slika 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 $0 \le k \le 28$, $x \ge 0$, $y \ge 0$, $1 \le w \le 100$, $1 \le h \le 100$. V 40 % testnih primerov bo veljalo tudi $k \le 10$.

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:

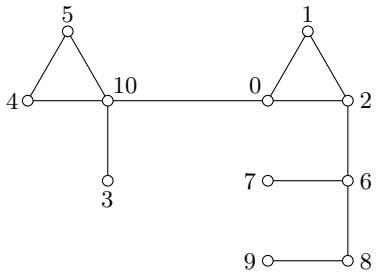
```
3 10 7 12 5
```

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 $3 \le n \le 500\,000$ in $3 \le m \le 500\,000$. 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 $2 \leq w \leq 100$, $2 \leq h \leq 100$, $1 \leq n < wh$.

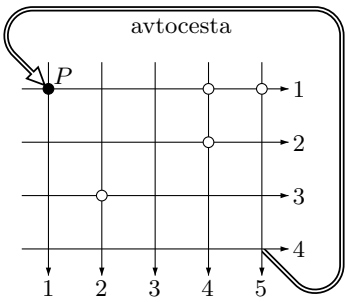
Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše potrebno število voženj po avtocesti, ki jih mora poštar opraviti, da lahko dostavi vse pošiljke in se na koncu vrne na pošto.

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.

5. tekmovanje ACM in IJS v znanju računalništva za srednješolce

27. marca 2010

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.[1]

```
#include <stdio.h>

int main()
{
  int znak, prejZnak = -1, dolzina = 0, naj = 0;
  while ((znak = fgetc(stdin)) != EOF)
  {
    if (znak < 'a' || znak > 'z') dolzina = 0;
```

[1] To je načeloma koristno, da ne bi kot abecednega podniza prepoznali niza „`a“ (saj je krativec „`“ 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.

```
    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');
}
```

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); }
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Poštne številke

V tabeli `variante` imejmo za vsako števko (od 0 do 9) niz z vsemi možnimi števkami, 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 števke z deljenjem: ostanek po deljenju k z 10 je ravno najbolj desna števka, količnik pa je v tem primeru število, ki ga dobimo, če k-ju pobrišemo najbolj desno števko. Ta količnik lahko nato spet delimo z 10, da dobimo predzadnjo števko k-ja in tako naprej. Ko imamo vse štiri števke (`d1`, `d2`, `d3` in `d4`), se v štirih gnezdenih zankah sprehodimo po vseh kombinacijah števk 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);
}
```

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 izračunu 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.

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) continue;
    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 $O(n^2)$. Obstajajo tudi učinkovitejše (a precej bolj zapletene) rešitve s časovno zahtevnostjo $O(n \log n)$; da pa se tudi pokazati, da hitreje od tega ne gre.

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.

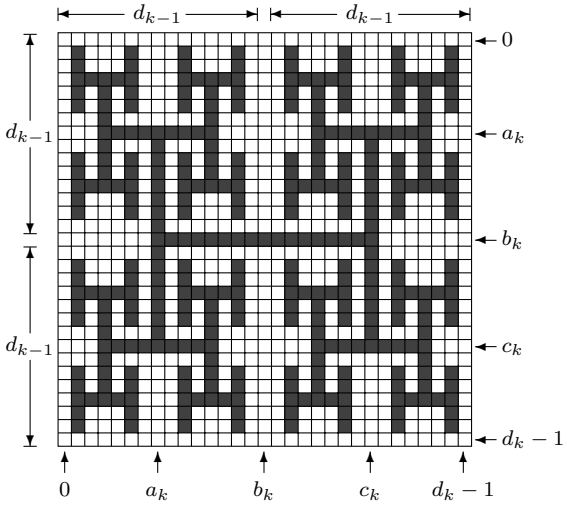
Na koncu se lahko sprehodimo po seznamu od spodnjega zvezka proti vrhu (pri tem sledimo povezavam iz tabele nad) in pri tem štejemo, mimo koliko zvezkov smo že šli. Tako za trenutni zvezek vemo, koliko zvezkov je že pod njim; to vrednost si nekje zapomnimo in na koncu te vrednosti izpišemo. Spodnji program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

```
#include <stdio.h>

#define MaxN 10000
#define MaxK 1000000

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 */
    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.

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] = 1; 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,...,w, for r_h := 1,...,h:
  for r_x := 0,...,w − r_w − 1, for r_y := 0,...,h − r_h − 1:
    ok := false;
    for x := r_x,...,r_x + r_w − 1, for y := r_y,...,r_y + r_h − 1:
      if črno[y, x] then ok := false; break;
    if ok then M := max{M, r_w · 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,...,h − 1, for r_x := 0,...,w − 1:
  for r_w := 1,...,w − r_x:
    for r_h := 1,...,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** $r_y := 0, \ldots, h-1$, **for** $r_x := 0, \ldots, w-1$:
  for $r_w := 1, \ldots, w - r_x$:
    for $r_h := 1, \ldots, h - r_y$:
      $ok := (\mathit{desno}[r_y + r_h - 1, r_x] \geq r_w)$;
      if not ok **then break**
      else $M := \max\{M, r_w \cdot r_h\}$;

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 $x := 9, \ldots, w-1$: $\mathit{pod}[h, x] := 0$;
for $y := h-1, \ldots, 0$:
  for $x := 0, \ldots, w-1$:
    $\mathit{pod}[y, x] := \mathit{pod}[y+1, x]$;
    if $\mathit{črno}[y, x]$ **then** $\mathit{pod}[y, x] := \mathit{pod}[y, x] + 1$;

$M := 0$; **for** $r_y := 0, \ldots, h-1$, **for** $r_x := 0, \ldots, w-1$:
  for $r_h := 1, \ldots, h - r_y$:
    for $r_w := 1, \ldots, w - r_x$:
      $ok := (\mathit{pod}[r_y, r_x + r_w - 1] \geq r_h)$;
      if not ok **then break**
      else $M := \max\{M, r_w \cdot r_h\}$;

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 tako, pri kateri je pravokotnik še popolnoma bel).

$M := 0$; **for** $r_y := 0, \ldots, h-1$, **for** $r_x := 0, \ldots, w-1$:
  $r_h := \infty$;
  for $r_w := 1, \ldots, w - r_x$:
    $r_h := \min\{r_h, \mathit{pod}[r_y, r_x + r_w - 1]\}$;
    if $r_h = 0$ **then break**;
    else $M := \max\{M, r_w \cdot r_h\}$;

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 $O(w^2h)$.

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 tistih, 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 $x := 0, \ldots, w-1$: $nad[0, x] := 0$;
for $y := 1, \ldots, h-1$:
  for $x := 0, \ldots, w-1$:
    $nad[y, x] := nad[y+1, x]$;
    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 $r_y := 0, \ldots, h-1$:
  $r_x := 0$;
  while $r_x < w$:
    if $x > 0$ **then if not** $črno[y, x-1]$ **then** $r_x := r_x + 1$; **continue**;
    $h_{nad} := \infty$; $h_{pod} := \infty$; $r_w := 0$;
    while $r_x + r_w < w$:
      if $črno[r_y, r_x + r_w]$ **then break**;
      $h_{nad} := \min\{h_{nad}, nad[r_y, r_x + r_w]\}$;
      $h_{pod} := \min\{h_{pod}, pod[r_y, r_x + r_w]\}$;
      $r_w := r_w + 1$;
      $M := \max\{M, r_w \cdot (h_{nad} + h_{pod})\}$;
    $r_x := r_x + r_w$;

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.[2] 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 $r_y := 0, \ldots, h-1$:
  $S :=$ prazen sklad, na katerem bodo pari oblike $\langle višina, x \rangle$;
  dodaj $\langle 0, -1 \rangle$ na S;
  for $r_x := 0, \ldots, w$:
    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;
      $M := \max\{M, (r_x - x) \cdot v'\}$;
    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 $O(wh)$.

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

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 $O(wh)$.

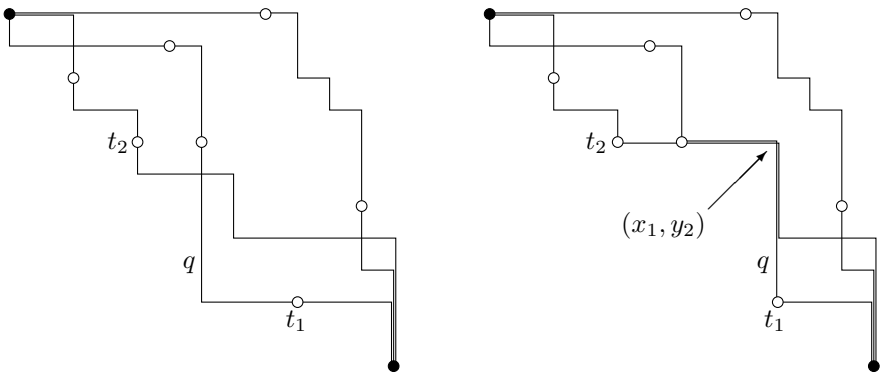
```
#include <stdio.h>
#include <stdbool.h>

#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 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 $y \leq y_2$. Za q to med drugim pomeni, da gre tudi skozi (x_1, y_2), saj pride do $x = x_1$ pri $y \leq y_2$ 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 $y \leq y_2$ 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 „$y \leq y_2$“ 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 $Q^\star$.

Viri nalog: H-fraktal — Nino Bašić; poštne številke, delitev kamenja — Andrej Bauer; vodilni elementi — Primož Gabrijelčič; parktronic — Boris Gašperin; poštar — Tomaž Hočevar; pravokotniki — Peter Keše; kvadrat s seštevanjem, potok presledkov — Mitja Lasič; skrivanje tipk — Mark Martinec; abecedni podnizi — Polona Novak; cikel, kup knjig, največji pravokotnik, slepe ulice — Mitja Trampuš. Hvala Ninu Bašiću za implementacijo rešitev nalog za 3. skupino. Primer pri nalogi „potok presledkov“ je iz 25. poglavja Gibbonove *Zgodovine zatona in propada Rimskega cesarstva*.

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