7. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2012

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.");
   }
}
```

7. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2012

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. (Oddano rešitev lahko kasneje še spreminjaš.) **Zaradi varnosti priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku** (npr. kopiraj in prilepi v Notepad in shrani v datoteko).

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. Prepletene besede

Napiši podprogram ali funkcijo Prepletene, ki kot parameter dobi niz `s` in iz njega sestavi nov niz (recimo mu `t`) tako, da vse velike črke premakne na začetek niza, vse male črke pa na konec niza. Pri tem naj se vrstni red velikih in malih črk ohrani. Tako dobljeni niz `t` lahko tvoj podprogram vrne kot funkcijsko vrednost ali pa kot drugi parameter po referenci (kar ti je lažje). Predpostavi, da se v nizu pojavljajo le črke angleške abecede.

Primer: `PbREPeseLEdTiEloNO` → `PREPLETENObesedilo`

Tvoj podprogram naj bo takšne oblike:

```
procedure Prepletene(s: string; var t: string);           { v pascalu }
function Prepletene(s: string): string;                   { vrne t }
void Prepletene(const char *s, char *t);     /* v C/C++; predpostavi, da t že kaže na
                                                 primerno velik kos pomnilnika */
string Prepletene(string s);                              // v C++; vrne t
void Prepletene(string s, string& t);                     // v C++
public static String Prepletene(String s);                // v javi
public static string Prepletene(string s);                // v C#; vrne t
public static void Prepletene(string s, out string t);    // v C#
def Prepletene(s): ...                                    # v pythonu; vrne t
```

2. Manjkajoča števila

Na vhodni datoteki imamo zapisana naravna števila v naraščajočem vrstnem redu, vsako število v svoji vrstici. Večinoma gre za zaporedna števila, vendar včasih kakšno manjka, lahko manjka tudi več zaporednih števil.

Napiši program, ki bo bral števila z vhodne datoteke in jih **sproti** prepisoval na izhodno datoteko, pri tem pa izpisal tudi vsa morebitna manjkajoča števila, a ta naj izpiše v oklepajih. Manjkajočih naravnih števil pred prvim prebranim in za zadnjim prebranim ne izpišemo. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `stevila.txt` (kar ti je lažje). Program naj bere vse do konca vhoda (EOF).

Primer vhodnih podatkov:

```
4
5
6
7
9
10
11
17
18
```

Pripadajoči izpis:

```
4
5
6
7
(8)
9
10
11
(12)
(13)
(14)
(15)
(16)
17
18
```

3. Kazenski stavek

V šoli je bilo včasih za posebne dosežke treba ostati v šoli in večkrat na tablo napisati „kazenski stavek“. Tako se je po pol ure vrtenja krede in razmišljanja o tem, kako te nihče ne mara, na tabli pojavilo nekaj takega:

```
NE BOM VEC METAL PAPIRCKOV PO TLEH NE BOM VEC METAL PAPIRCKOV PO
TLEH NE BOM VEC METAL PAPIRCKOV PO TLEH NE BOM VEC METAL PAPIRCKOV
PO TLEH NE BOM VEC METAL PAPIRCKOV PO TLEH NE BOM VEC METAL
PAPIRCKOV PO TLEH NE BOM VEC METAL PAPIRCKOV PO TLEH NE BOM
VEC METAL PAPIRCKOV PO TLEH NE BOM VEC METAL PAPIRCKOV PO TLEH NE
BOM VEC METAL PAPIRCKOV PO TLEH
```

Napiši podprogram ali opiši postopek, ki bo iz zapisa na tabli ugotovil, kolikokrat se v njem pojavi kazenski stavek. Predpostavi, da je zapis na tabli podan kot en sam dolg niz (torej ni razdeljen na več vrstic). Kazenski stavek vzemi kot najkrajši tak kos besedila, za katerega velja, da je mogoče dani zapis s table sestaviti iz več ponovitev tega stavka, ločenih s po enim presledkom. Predpostaviš lahko, da se pisec ni zmotil ter da se v vhodnem nizu pojavljajo le velike črke angleške abecede in presledki.

V zgornjem primeru se kazenski stavek „`NE BOM VEC METAL PAPIRCKOV PO TLEH`“ pojavi 10-krat.

Če pišeš podprogram, naj bo takšne oblike:

```
function KazenskiStavek(s: string): integer;        { v pascalu }
int KazenskiStavek(char *s);                         /* v C/C++ */
int KazenskiStavek(string s);                        // v C++
public static int kazenskiStavek(String s);          // v javi
public static int KazenskiStavek(string s);          // v C#
def KazenskiStavek(s): ...                           # v pythonu; vrne int
```

4. Mase

V neki tovarni na koncu proizvodnega procesa vsak izdelek stehtajo, maso v gramih pa zapišejo v bazo. Kot vsaka tehtnica ima tudi ta znano svojo natančnost, ki je $\pm a$ gramov (torej, če tehtamo izdelek z maso m, lahko tehtnica pokaže katero koli število od vključno $m - a$ do vključno $m + a$).

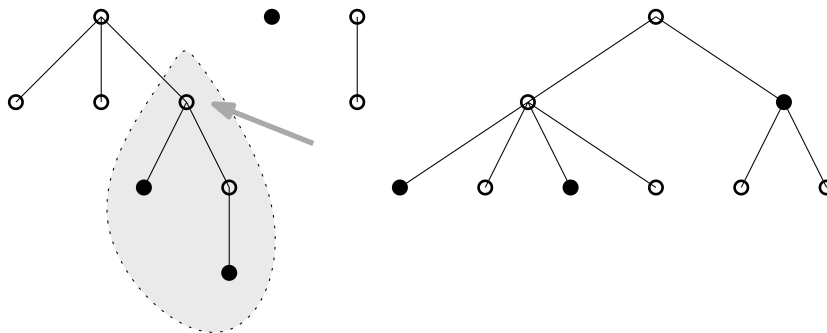
Izdelki so različnih vrst, vsi izdelki iste vrste imajo enako maso, izdelki različnih vrst pa imajo različne mase. Težava je v tem, da ne vemo, koliko vrst izdelkov obstaja in kakšne so njihove prave mase. **Opiši postopek**, ki kot vhodne podatke dobi natančnost tehtnice a in zaporedje meritev (za vsako meritev imamo izmerjeno maso v gramih) ter iz teh podatkov ugotovi, kakšno je najmanjše število različnih vrst izdelkov, pri katerem bi se (glede na dano natančnost tehtnice a) dalo s tehtanjem dobiti takšno zaporedje mas, kot je podano v vhodnih podatkih. Predpostavi, da je vhodno zaporedje meritev podano v naraščajočem vrstnem redu.

Primer: če imamo $a = 5$ in zaporedje meritev 15, 24, 26, je pravilni odgovor 2. Če bi imeli $a = 5$ in zaporedje meritev 15, 24, 25, pa bi bil pravilni odgovor 1 (kajti do vseh teh treh meritev lahko pride med drugim tako, da imamo eno samo vrsto izdelkov z maso 20).

5. V Afganistan!

Imamo n vojakov, oštevilčenih s števili od 0 do $n-1$. Med njimi vlada stroga hierarhija: nekaj vojakov ima najvišji možen čin (in jim ni nihče nadrejen), vsak od preostalih vojakov pa ima natanko enega neposredno nadrejenega vojaka.

Nekatere od vojakov — recimo jim gajstni — želimo poslati v Afganistan. Kateremu koli vojaku (gajstnemu ali ne) lahko damo ukaz, naj se odpravi, in bo to seveda naredil, pri tem pa bo s seboj odpeljal tudi vse sebi podrejene vojake. Za primer glej sliko: vsaka točka predstavlja vojaka, vojaki z višjimi čini so narisani višje. Gajstni vojaki so pobarvani črno. Če damo ukaz vojaku, označenemu s puščico, bo s seboj v Afganistan odpeljal še preostale tri vojake iz črtkano obrobljenega območja.



Na primeru s slike se očitno ne da izdati enega samega ukaza, s katerim bi poslali na pot vse gajstne vojake. Tudi dva ukaza ne zadoščata; lahko pa to naredimo s tremi ukazi, celo na dva različna načina. Zanima nas, koliko ukazov potrebujemo v splošnem.

Napiši program, ki kot podatke dobi opis vojaške hierarhije ter gajstnih vojakov in vrne najmanjše število ukazov, s katerim lahko dosežemo, da bodo v Afganistan odpotovali vsi gajstni vojaki. Če pri tem na pot ukažemo še kakšnim ne-gajstnim vojakom, ni nič narobe. Za dostop do podatkov uporabi naslednji funkciji:

- `Nadrejeni(i)`, ki vrne zaporedno številko vojaka, neposredno nadrejenega vojaku `i`, oziroma -1, če ima vojak `i` najvišji možen čin in mu ni nihče nadrejen;
- `Gajsten(i)`, ki vrne 1, če je vojak `i` gajsten (in ga torej moramo poslati v Afganistan), in 0 sicer.

Predpostavi še, da je na voljo spremenljivka `n`, ki vsebuje število vseh vojakov.

7. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2012

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. (Oddano rešitev lahko kasneje še spreminjaš.) **Zaradi varnosti priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku** (npr. kopiraj in prilepi v Notepad in shrani v datoteko).

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. Ovce

Novozelandski kmet ima n ovac. V času t morajo biti vse ovce ostrižene, saj bo takrat odprodal volno. Ima seznam m strižcev. Za vsakega ima dva podatka:

- p_i je plačilo, ki ga zahteva i-ti strižec za striženje ene ovce;
- t_i je čas, ki ga porabi i-ti strižec za striženje ene ovce.

Aparat za striženje pokuri za c denarnih enot elektrike na časovno enoto. **Opiši postopek**, ki iz teh podatkov ugotovi, kolikšen je najmanjši znesek, ki ga bo kmet plačal za striženje (najem delovne sile + elektrika), če delovno silo izbira optimalno. Vsi strižci lahko delajo hkrati in vsak ima svoj aparat za striženje. Tisti, ki začne striči ovco, jo mora tudi dokončati. Na voljo bo vedno dovolj delavcev, da bodo lahko v danem času ostrigli vse ovce.

Primer: recimo, da imamo $n = 5$ ovac, $t = 12$ enot časa, ceno elektrike $c = 1$ in dva strižca: enega s $p_1 = 6$, $t_1 = 2$ (drag, a hiter) in enega s $p_2 = 2$, $t_2 = 5$ (počasen, a cenejši). Potem se izkaže, da je najmanjši skupni strošek striženja enak 38 (dosežemo ga, če prvi strižec ostriže tri ovce, drugi pa dve).

2. Spričevala

Da bi v prihodnje hitreje odkrili sumljiva (ponarejena) spričevala, so te v javni upravi prosili za pomoč. **Napiši program**, ki bo za vsa spričevala zaposlenih preveril to,

- ali je bil kateri ravnatelj v istem letu podpisan na spričevalih več šol
- ali sta se v istem letu pod katerikoli dve spričevali iste šole podpisali različni osebi,

in v obeh primerih opozoril, da bi bilo koristno preveriti podatke.

Za vsa spričevala so na voljo podatki o šoli, kjer je bilo spričevalo izdano, letu izdaje in podpisanem ravnatelju. Da bi ti bilo lažje, so ti že pripravili vhodne podatke, kjer so šole namesto s polnim imenom predstavljene s števili od 1 do s in ravnatelji s števili od 1 do r.

Tvoj program naj podatke bere s standardnega vhoda ali pa iz datoteke z imenom `spricevala.txt` (kar ti je lažje). V prvi vrstici so števila S (število šol), R (število ravnateljev) in n (število spričeval); veljalo bo $1 \leq S \leq 10^6$ in $1 \leq R \leq 10^6$. Sledi n vrstic, za vsako spričevalo po ena, v njej pa so zaporedoma zapisani leto, šifra šole in šifra ravnatelja. Te vrstice so urejene naraščajoče glede na leto izdaje spričevala.

Če npr. dobi naslednje vhodne podatke:

```
6 5 12
1995 5 2
1995 6 1
1995 5 2
1995 5 3
1998 5 4
2000 3 1
2000 2 5
2000 4 1
2001 4 2
2010 1 1
2010 2 3
2010 2 1
```

mora tvoj program izpisati:

```
Preveri spricevala sole st. 5 v letu 1995.
Preveri spricevala s podpisom ravnatelja st. 1 v letu 2000.
Preveri spricevala sole st. 2 v letu 2010.
Preveri spricevala s podpisom ravnatelja st. 1 v letu 2010.
```

Vrstni red izpisa ni pomemben, izogibaj se le podvojenim izpisom.

3. Razpolovišče lika

Nekoč je živel kralj, ki je umrl. Kraljestvo je v oporoki zapustil svojima sinovoma — vsakemu natanko polovico. Njegova želja je, da se kraljestvo razdeli na dva dela s popolnoma ravno mejo, ki poteka točno v smeri vzhod–zahod. Dvorni geometri imajo težavo določiti primerno lego meje. Pomagaj jim.

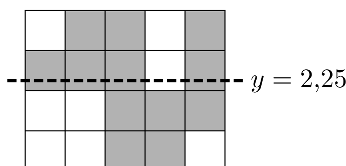
Zemljevid obstoječega kraljestva je znan. Vrisan je na karirasto mrežo širine w in višine h kvadratkov in orientiran tako, da kaže sever navpično navzgor. Na mreži je vsak kvadratek bodisi bel bodisi črn; črni kvadratki sestavljajo kraljestvo.

Napiši podprogram (funkcijo), ki pregleda zemljevid kraljestva in vrne y-koordinato tiste vodoravne črte (premice), za katero sta površini kraljestva pod in nad črto enaki. Če ob razrezu kraljestva „podkraljestvi“ razpadeta na več kosov, nas to ne moti. Koordinatni sistem postavimo tako, da ima izhodišče v spodnjem levem vogalu zemljevida, enota pa je enaka stranici enega kvadratka na karirasti mreži. Pozor — iščemo točno vrednost za y in ta ni nujno celoštevilska. (Predpostavi, da so podatkovni tipi, ki jih tvoj programski jezik uporablja za predstavitev realnih števil (npr. **float** ali **double**) dovolj natančni za potrebe te naloge, torej ti ni treba skrbeti zaradi morebitnih majhnih zaokrožitvenih napak.)

Nalogo lahko rešiš tudi tako, da izpišeš *celoštevilski* y, ki čim bolje (čeprav morda ne čisto natančno) razpolovi kraljestvo. Takšne rešitve bodo vredne največ 10 točk (od 20).

Pri pisanju podprograma privzemi, da imaš nekje že definirani spremenljivki `w` in `h`, ki hranita širino oz. višino zemljevida (v številu kvadratkov), ter funkcijo `Znotraj(x, y)`, ki vrne 1, če je kvadratek s koordinatama (x, y) na zemljevidu črn, sicer pa 0.

Primer:



Na sliki je $w = 5$, $h = 4$. Vrisana je tudi iskana rešitev — meja poteka na višini $y = 2{,}25$. V tem primeru je vrstica razrezana po četrtini in zato južni polovici kraljestva prispeva en kvadratek ploščine, severni polovici pa tri kvadratke. Tako imata severna in južna polovica res enako skupno ploščino, namreč 6 kvadratkov. Rešitev za 10 točk, ki vedno vrača celoštevilske y, pa bi morala pri primeru s slike vrniti $y = 2$.

4. Strukturirani podatki

Na vhodni datoteki imamo strukturirano besedilo v obliki, ki je podobna HTML ali XML, a s preprostejšim zapisom. Lahko si ga predstavljamo kot knjigo, ki vsebuje poglavja, ta vsebujejo podpoglavja, ta odstavke in tako naprej.

Vsaka vrstica vhodne datoteke lahko vsebuje začetno značko, končno značko ali pa neko poljubno vrstico besedila.

Začetna značka je sestavljena iz znaka „`+`“, ki mu sledi neko poljubno ime značke, npr:

```
+uvod
```

Končna značka je sestavljena iz znaka „`-`“, ki mu sledi ime značke, ki se na tem mestu zaključuje, npr:

```
-uvod
```

Predpostavimo lahko, da so vhodni podatki pravilni: začetne in končne značke so lahko gnezdene, pri tem se ime končne značke vedno ujema z imenom zadnje začetne značke, zato ga ni treba preverjati.

Vse ostale vrstice lahko vsebujejo poljubno besedilo. Te vrstice se ne morejo začeti z znakoma „`+`“ ali „`-`“.

Zaradi preglednosti lahko na začetku vsake vrstice stojijo presledki, ki pa jih ignoriramo.

Primer takega besedila na vhodni datoteki:

```
+ena
  +dve
    alfa
  -dve
  +tri
    beta
    gama
    +stiri
      delta
    -stiri
    epsilon
  -tri
  zeta
  +tri
  -tri
-ena
eta
```

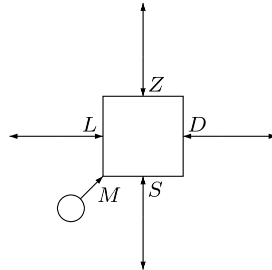
Napiši program, ki bo bral vhodne podatke v tej obliki in sproti izpisoval vse vrstice besedila, ki niso značke; pri tem pa naj pred vsako tako vrstico izpiše vse trenutno aktivne značke (v takem vrstnem redu, kot so gnezdene druga v drugi), med seboj ločene s pikami. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `vhod.txt` (kar ti je lažje). Bere naj vse do konca (EOF). Predpostavi, da je posamezna vrstica dolga največ 100 znakov in da značke niso gnezdene več kot 100 nivojev globoko. Imena značk so sestavljena le iz črk angleške abecede.

Takle naj bo rezultat obdelave zgornjega primera:

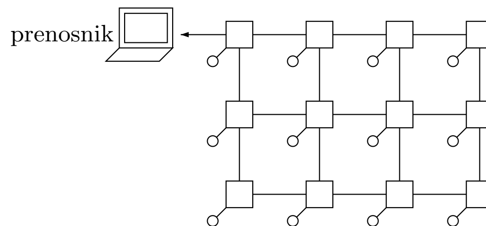
```
ena.dve alfa
ena.tri beta
ena.tri gama
ena.tri.stiri delta
ena.tri epsilon
ena zeta
 eta
```

5. Največji pretok

Imamo mrežo enostavnih merilnih računalnikov. Vsakega sestavlja procesor, senzor in pet komunikacijskih kanalov. Štirje kanali so namenjeni povezavi med računalniki: L (levo), D (desno), Z (zgoraj) in S (spodaj). Peta povezava, M (merilnik) sprejema podatke od senzorja.



Računalniki so s kanali L, D, Z in S povezani v dvodimenzionalno mrežo.



Mrežo potopimo v reko, pri čemer merilniki na posameznih računalnikih merijo pretok vode v reki. Na zgornji levi računalnik priključimo prenosnik, ki bo zajemal podatke.

Na vseh računalnikih je pognan enak program, ki se izvaja v neskončni zanki. Program ima na voljo funkciji Beri in Pisi.

```
function Beri(Kanal: char): integer;                     { v pascalu }
procedure Pisi(Kanal: char; Vrednost: integer);

int Beri(char kanal);                                    /* v C/C++ */
void Pisi(char kanal, int vrednost);

public static int Beri(char kanal);                      // v javi/C#
public static void Pisi(char kanal, int vrednost);

def Beri(kanal): ... # vrne int                          # v pythonu
def Pisi(kanal, vrednost): ...
```

Funkcija Beri prebere podatek iz komunikacijskega kanala (če je na voljo) in vrne njegovo vrednost. Če je kanal prazen ali pa ni nikamor povezan (večina kanalov na robu mreže), funkcija vrne -1.

Podprogram Pisi zapiše podatek na podani kanal. Predpostaviš lahko, da je v kanalu vedno dovolj prostora za zapis podatka. Če ta kanal ni nikamor povezan, podprogram ne naredi ničesar.

S prenosnikom, priključenim na zgornji levi računalnik, bi radi prebrali največji izmerjeni pretok reke v celem omrežju. **Napiši program**, ki se bo izvajal na vseh računalnikih v omrežju in bo poskrbel za to, da bo zgornji levi računalnik na kanal L pošiljal največji doslej zaznani pretok (največji od začetka delovanja celotne mreže). Predpostavi, da so procesorji hitri, komunikacija med njimi tudi, sprejemljivo pa je, če pride podatek o največjem pretoku do zgornjega levega računalnika z majhno zakasnitvijo.

7. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2012

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 imenik `U:_Osebno`, v katerem lahko kreiraš svoje datoteke. 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 1.7 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), `javac` (za javo 1.7), Visual Studio 2010 in druga orodja.

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 tretja 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 %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();
  }
}
```

7. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2012

NALOGE ZA TRETJO SKUPINO

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

1. de-FFT permutacija (`defft.in`, `defft.out`)

Obveščevalna agencija SOVA se je odločila, da za šifriranje svojih sporočil ne bo zaupala ameriškim standardom, kot sta SHA-2 in 3DES. Raje so razvili lastno, novo kodirno funkcijo po imenu FFT (Frišna Funkcija proti Terorju). FFT zakodira podani niz znakov $s_1 s_2 s_3 s_4 \ldots$ tako, da znake premeša po pravilu

$$\text{FFT}(s_1 s_2 s_3 s_4 \ldots) = \text{FFT}(s_1 s_3 s_5 \ldots) + \text{FFT}(s_2 s_4 s_6 \ldots),$$

pri čemer + pomeni navadno stikanje nizov. Zgornja rekurzivna definicija velja za vse vhodne nize dolžine 2 ali več; za vhodne nize dolžine 1 pa nimamo česa premešati in definiramo kar $\text{FFT}(s_1) = s_1$.

Primer 1:

$$\begin{aligned} \text{FFT}(\texttt{sadje}) &= \text{FFT}(\texttt{sde}) + \text{FFT}(\texttt{aj}) \\ &= (\text{FFT}(\texttt{se}) + \text{FFT}(\texttt{d})) + (\text{FFT}(\texttt{a}) + \text{FFT}(\texttt{j})) \\ &= ((\text{FFT}(\texttt{s}) + \text{FFT}(\texttt{e})) + \texttt{d}) + (\texttt{a} + \texttt{j}) \\ &= \texttt{sedaj} \end{aligned}$$

Primer 2 (izpeljavo preskočimo): FFT(`kokain_crv`) = `krik_ovnac`

Dokaži Sovi, da FFT ni prav dobra šifra. **Napiši program**, ki dešifrira poljuben niz, zašifriran s FFT.

Vhodna datoteka: v prvi vrstici je število n ($1 \le n \le 100$), število šifriranih sporočil. Vsaka od naslednjih n vrstic vsebuje po eno šifrirano sporočilo y_i, sestavljeno izključno iz malih črk angleške abecede ter podčrtajev. Sporočila ne bodo daljša od 1024 znakov. V 30 % testnih primerov bodo vse dolžine sporočil potence števila 2.

Izhodna datoteka: vanjo po vrsti izpiši vseh n izvornih sporočil x_i, vsako v svoji vrstici. To, da je x_i „izvorno sporočilo“, pomeni, da velja zveza $\text{FFT}(x_i) = y_i$.

Primer vhodne datoteke:

```
9
sedaj
krik_ovnac
tutla_orask
avion_desno
desna_avtor
enak_nitro
lesk_oktav
uhan_serje
star_mivka
```

Pripadajoča izhodna datoteka:

```
sadje
kokain_crv
tolsta_kura
adonis_oven
danost_reva
enkrat_oni
lokast_vek
usnjar_ehe
smrkav_ati
```

2. Potovanje (`potovanje.in`, `potovanje.out`)

S prijatelji se odpravljate na dolgo potovanje. Pot, ki vas bo vodila od zahoda proti vzhodu, ste že izbrali. Prav tako ste označili vse bencinske črpalke ob poti in količino goriva, ki jo lahko kupite na posamezni črpalki. Izkaže pa se, da je dolžina poti, ki jo boste lahko prevozili, odvisna od začetne lokacije vašega potovanja. Ko vam zmanjka goriva, se bo vaša pot namreč končala. Avto porabi 1 enoto goriva na kilometer in ima neomejeno prostornino tanka za gorivo. Prišlo je do manjšega prepira, saj se nikakor ne morete dogovoriti, kje začeti. Da bi bila odločitev lažja, **napiši program**, ki bo za vsako črpalko izračunal, kakšno razdaljo lahko prepotujete, če svojo pot začnete na tej črpalki.

Vhodna datoteka: v prvi vrstici je število bencinskih črpalk n (velja $n \leq 10^6$). V naslednjih n vrsticah sledijo opisi črpalk; i-ta od teh vrstic vsebuje dve celi števili, x_i in g_i, ločeni s presledkom, pri čemer je x_i oddaljenost i-te črpalke od zahodnega konca poti v kilometrih, g_i pa je količina goriva, ki je na voljo na tej črpalki. Obe števili sta med vključno 1 in 10^9. Črpalke so podane v vrstnem redu, kot si sledijo od zahoda proti vzhodu. V 40 % testnih primerov bo veljalo $n \leq 10^4$.

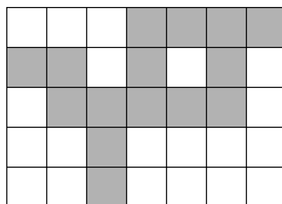
Izhodna datoteka: za vsako črpalko (po vrsti od zahoda proti vzhodu) izpiši po eno vrstico, ki naj vsebuje največjo možno prepotovano razdaljo v kilometrih, če začnete svojo pot na tej črpalki.

(*Nasvet:* skupna količina razpoložljivega goriva je lahko pri nekaterih testnih primerih večja od 2^{32}, zato je pri tej nalogi koristno za nekatere količine uporabiti kakšnega od 64-bitnih celoštevilskih podatkovnih tipov, na primer `int64` v pascalu, **`long long`** v C/C++ in **`long`** v javi/C#.)

Primer vhodne datoteke:	Pripadajoča izhodna datoteka:
5	20
1 10	9
4 5	4
9 4	1
19 1	11
22 11	

3. Leteči pujsi (`pujsi.in`, `pujsi.out`)

Ptice so tako dolgo najedale pujsom, da so se odločili preiti v protinapad in tokrat napasti ptičjo trdnjavo. Ta je predstavljena s karirasto mrežo, pri čemer črna polja predstavljajo opeke, bela polja pa so prosta. Rekli bomo, da je opeka *stabilna*, če bodisi leži v najbolj spodnji vrstici mreže ali pa na svojem desnem, levem ali spodnjem robu meji na neko drugo stabilno opeko. Na primer, v spodnji mreži so vse opeke stabilne:



Leteči pujs si lahko izbere eno od opek (katero koli) in se zaleti vanjo; izbrana opeka pri tem postane nestabilna, zaradi tega pa lahko postanejo nestabilne tudi nekatere druge opeke. **Napiši program**, ki za vsako polje mreže ugotovi, koliko stabilnih opek ostane na mreži, če se pujs zaleti v tisto polje. (Če se zaleti v polje, na katerem sploh ni bilo opeke, se v mreži nič ne spremeni.)

Vhodna datoteka: v prvi vrstici sta dve celi števili, ločeni s presledkom: najprej h (višina mreže) in nato w (širina mreže). Sledi h vrstic, ki opisujejo začetno stanje mreže. Vsaka od njih vsebuje w znakov, ki opisujejo tisto vrstico, pri čemer pika „`.`“ pomeni prazno polje, znak „`#`“ pa opeko. Začetno stanje mreže je pri vseh testnih primerih takšno, da je na mreži prisotna vsaj ena opeka in da so vse opeke stabilne. Veljalo bo $1 \leq w \leq 300$ in $1 \leq h \leq 300$. Pri 50 % testnih primerov bo veljalo tudi $1 \leq w \leq 100$ in $1 \leq h \leq 100$.

Izhodna datoteka: vanjo izpiši h vrstic, v vsako od teh pa w števil, ločenih s po enim presledkom. Vsako od teh števil naj pove, koliko stabilnih opek ostane na mreži, če se pujs zaleti v pripadajoče polje mreže.

Primer vhodne datoteke:

```
5 7
..###.#
##..###
.##.#..
.####..
.##....
```

Pripadajoča izhodna datoteka:

```
18 18 17 16 15 18 17
17 16 18 18 11 15 16
18 15 17 18 10 18 18
18 17 7 8 9 18 18
18 17 17 18 18 18 18
```

4. Nakup parcele (`parcela.in`, `parcela.out`)

V lasti imaš zemljišče, ki je razdeljeno na karirasto mrežo $h \times w$ parcel (h vrstic, w stolpcev; parcele so kvadratne oblike in so vse enako velike). Za vsako parcelo poznamo njeno vrednost, ki je celo število med vključno 1 in 10^9. Za nakup zemlje se zanima k kupcev, ki želijo kupiti različno velika zemljišča: i-ti kupec želi kupiti pravokotno zemljišče, visoko n_i parcel in široko m_i parcel. V navadi je, da kupec za zemljišče plača toliko, kot je vredna najcenejša parcela na njem. Cena, ki jo bo kupec plačal za zemljišče, je torej v splošnem lahko odvisna od tega, kje na naši mreži si bo to zemljišče izbral — v mreži velikosti $h \times w$ si lahko izberemo pravokotnik $n_i \times m_i$ na $(h - n_i + 1) \cdot (w - m_i + 1)$ načinov. Ker mi poznamo le velikost zemljišča, ki ga hoče ta kupec kupiti, ne pa tudi njegovega položaja, bi radi izračunali vsoto cene tako velikega zemljišča (torej $n_i \times m_i$ parcel) po vseh možnih položajih takega zemljišča v naši mreži. **Napiši program**, ki za vsakega kupca posebej izračuna to vsoto cene zemljišča.

Vhodna datoteka: prva vrstica vsebuje višino h in širino w tvojega zemljišča, ločeni s presledkom; veljalo bo $2 \leq h \leq 2000$ in $2 \leq w \leq 2000$. Naslednjih h vrstic vsebuje po w celih števil, ki predstavljajo vrednosti posameznih parcel. Temu sledi vrstica s številom kupcev k (velja $1 \leq k \leq 10$), nato pa k vrstic, ki opisujejo velikosti zemljišč, ki bi jih radi ti kupci kupili; nakup i-tega kupca je opisan z višino n_i in širino m_i (ločeni sta s presledkom), pri čemer velja $2 \leq n_i \leq h$ in $2 \leq m_i \leq w$.

Izhodna datoteka: izpiši k vrstic, ki za vsakega kupca vsebujejo vsoto cene zemljišča danih dimenzij po vseh možnih položajih takega zemljišča.

(*Nasvet:* vsota bo pri nekaterih testnih primerih večja od 2^{32}, zato je pametno zanjo uporabiti kakšnega od 64-bitnih celoštevilskih podatkovnih tipov, na primer `int64` v pascalu, **long long** v C/C++ in **long** v javi/C#.)

Primer vhodne datoteke:

```
3 5
8 5 5 8 1
6 4 8 3 8
8 2 8 7 7
2
2 3
3 3
```

Pripadajoča izhodna datoteka:

```
15
5
```

Razlaga primera: prvi kupec bi lahko kupil 6 različnih zemljišč. Njihove vrednosti od zgoraj navzdol in od leve proti desni so: 4, 3, 1, 2, 2 in 3. Vsota teh vrednosti je 15.

5. Rotacija (`rotacija.in`, `rotacija.out`)

Igralnica je kupila novo igro, ki se imenuje Kolo sreče. Sestavljena je iz velikega kolesa, ki ima na obodu napisane števke od 1 do 9. Igralec kolo zavrti in ko se kolo ustavi, se v smeri urinega kazalca prebere število, ki ga sestavljajo zaporedne števke na obodu. To število predstavlja dobitek igralca. **Napiši program**, ki iz opisa kolesa sreče izračuna največji možni dobitek.

Vhodna datoteka: v prvi vrstici je celo število n ($1 \leq n \leq 10^6$), ki pove, koliko je števk na kolesu. V drugi vrstici je niz n števk, kot si sledijo na kolesu sreče v smeri urinega kazalca. V 40 % testnih primerov bo veljalo $n \leq 10^4$.

Izhodna datoteka: izpiši največji možni dobitek, ki ga lahko zadenemo na opisanem kolesu sreče.

Primer vhodne datoteke:

```
6
425747
```

Pripadajoča izhodna datoteka:

```
747425
```

7. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2012

REŠITVE NALOG ZA PRVO SKUPINO

1. Prepletene besede

Nalogo lahko rešimo tako, da naredimo dva prehoda po vhodnem nizu `s`; v prvem kopiramo v `t` velike črke, v drugem pa male črke. Podrobnosti so načeloma precej odvisne od tega, kako se dela z nizi v našem izbranem programskem jeziku. Spodnja rešitev je v C-ju in se s kazalcem `p` premika naprej po vhodnem nizu; vsakič, ko naleti na primerno črko (veliko v prvem prehodu, malo v drugem), pa jo izpiše v izhodni niz `t` in se premakne naprej tudi po njem (poveča kazalec `t`).

```
void Prepletene(const char *s, char *t)
{
  const char *p;
  for (p = s; *p; p++)
    if ('A' <= *p && *p <= 'Z') *t++ = *p;
  for (p = s; *p; p++)
    if (! ('A' <= *p && *p <= 'Z')) *t++ = *p;
  *t = 0;
}
```

2. Manjkajoča števila

Spodnja rešitev si v spremenljivki `prejsnje` hrani prejšnje prebrano število. To je tudi zadnje število, ki smo ga doslej izpisali. Ko preberemo naslednje število, recimo `n`, gremo z notranjo zanko od `prejsnje` + 1 do `n` − 1 in ta števila izpisujemo v oklepajih. Ob koncu te zanke ima `prejsnje` enak `n`, nato pa moramo `n` le še izpisati (brez oklepajev) in smo že pripravljeni na branje naslednjega vhodnega števila.

```
#include <stdio.h>

int main()
{
  int prejsnje = −1, n;
  while (1 == scanf("%d", &n))
  {
    if (prejsnje < 0) prejsnje = n;
    else while (++prejsnje < n) printf("(%d)\n", prejsnje);
    printf("%d\n", n);
  }
  return 0;
}
```

3. Kazenski stavek

Recimo, da je vhodni niz dolg n znakov, kazenski stavek pa d znakov. Spodnja rešitev gre z zanko po naraščajočih d, pri vsakem preveri, če se dá vhodni niz razumeti kot zaporedje kazenskih stavkov dolžine d (ločenih s po enim presledkom), in vrne prvi (torej najmanjši) tak d, ki ustreza temu pogoju.

Če je vhodni niz sestavljen iz p kopij kazenskega stavka dolžine d mora biti $n = p \cdot d + (p - 1)$, saj naloga pravi, da so posamezne kopije kazenskega stavka med seboj ločene s po enim presledkom. Ta pogoj lahko zapišemo tudi kot $(n + 1) = p \cdot (d + 1)$; z drugimi besedami, v poštev pridejo le takšne dolžine d, pri katerih je $n + 1$ večkratnik števila $d + 1$. Tako lahko večino d-jev takoj zavržemo.

Naslednji pogoj, ki ga je koristno preveriti, je, če je znak s[d] presledek — če imamo opravka s kazenskim stavkom dolžine d, mora prva pojavitev tega stavka pokrivati znake s[0], ..., s[d − 1], nato pa mora biti v s[d] presledek, ki ločuje prvo pojavitev kazenskega stavka od druge.

Nato moramo le še preveriti, če se isti kazenski stavek ponavlja tudi v preostanku niza; znak s[d + 1] mora biti enak s[0], znak s[d + 2] mora biti enak s[1] in tako naprej. To preverimo v notranji zanki; če ta uspešno pride do konca niza s, ne da bi našla kakšno neujemanje, potem vemo, da smo našli primerno dolžino kazenskega stavka. Število pojavitev (to je namreč tisto, po čemer sprašuje naloga) pa je potem, kot smo videli pri zgornji formuli, $(n + 1)/(d + 1)$.

```
#include <stdlib.h>

int KazenskiStavek(char *s)
{
  int n = strlen(s), d, i;
  for (d = 1; d < n; d++)
  {
    if ((n + 1) % (d + 1) != 0) continue;
    if (s[d] != ' ') continue;
    for (i = d + 1; i < n; i++)
      if (s[i] != s[i % (d + 1)]) break;
    if (i == n) break;
  }
  /* Če glavna zanka ni našla ničesar pametnega,
     imamo zdaj d = n in bomo vrnili 1. */
  return (n + 1) / (d + 1);
}
```

4. Mase

Rekli bomo, da neka vrsta izdelkov z maso m *pokrije* meritev x natanko tedaj, ko bi lahko meritev x nastala pri tehtanju izdelka te vrste; z drugimi besedami je to takrat, ko je $m - a \leq x \leq m + a$. Naloga tako pravzaprav zahteva, da vse dane meritve pokrijemo s čim manj izdelki.

Oglejmo si najmanjšo meritev; recimo ji x_1. Pokril bi jo na primer izdelek z maso $x_1 + a$; tak izdelek pokrije celoten interval $[x_1, x_1 + 2a]$. Ali je smiselno razmišljati o rešitvah, pri katerih je najlažji izdelek kaj lažji od $x_1 + a$? Ne, kajti če maso izdelka zmanjšamo pod $x_1 + a$, se interval mas, ki ga pokrije ta izdelek, pomakne navzdol; na spodnjem robu s tem ničesar ne pridobimo (saj meritev, manjših od x_1, sploh nimamo), na zgornjem robu pa mogoče kaj izgubimo (novi izdelek z manjšo maso mogoče ne pokrije nekaterih meritev, ki jih je izdelek $x_1 + a$ pokril). Tako torej vidimo, da ni nobene koristi od tega, da bi bil najlažji izdelek kaj lažji od $x_1 + a$. Po drugi strani pa najlažji izdelek ne sme biti težji od $x_1 + a$, saj bi drugače x_1 ostal nepokrit. Vzemimo torej izdelek z maso točno $x_1 + a$, iz zaporedja meritev pobrišimo tiste, ki jih je ta izdelek pokril (vse do vključno $x_1 + 2a$) in nadaljujemo po enakem postopku, dokler ni pokrito celotno zaporedje.

Zapišimo ta postopek še s psevdokodo:

naj bodo $x_1, \ldots, x_n$ vhodne meritve (urejene naraščajoče);
$i := 0$; $k := 0$; (* *i je števec po meritvah, k je število izdelkov* *)
while $i < n$:
  $k := k + 1$; $m_k := x_i + a$; (* *nova vrsta izdelka z maso* m_k *)
  (* *Preskočimo meritve, ki jih pokrije novi izdelek.* *)
  while $i < n$ **and** $x_i \leq m_k + a$ **do** $i := i + 1$;
return k;

5. V Afganistan!

Naloga pravi, da je vseeno, če poleg gajstnih vojakov pošljemo v Afganistan še kakšnega ne-gajstnega. Potemtakem ni nobene koristi od tega, da bi poslali ukaz vojaku x, če ima le-ta tudi nadrejenega, saj bi lahko ukaz poslali raje temu nadrejenemu in bi tako spravili

v Afganistan tudi vojaka x (in vse njegove posredno in neposredno podrejene). Torej je koristno, če se od vsakega gajstnega vojaka sprehodimo gor po hierarhiji, dokler ne pridemo do vrha; vse tako obiskane vojake označimo (v spodnjem programu imamo zato tabelo poslji). Na koncu tega postopka imamo tako v tej tabeli za vsakega vojaka logično vrednost, ki nam pove, če mora on ali kdo od njegovih (lahko posredno) podrejenih v Afganistan. Ukaze pošljimo tistim vojakom, pri katerih je ta pogoj izpolnjen in ki nimajo svojega nadrejenega.

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

int main()
{
  int i, v, stUkazov;
  /* V tabeli poslji bomo označili gajstne vojake in vse njihove
     nadrejene (posredne in neposredne). */
  bool *poslji = (bool *) malloc(sizeof(bool) * n);
  for (i = 0; i < n; i++) poslji[i] = false;
  for (i = 0; i < n; i++)
    /* Če je i gajsten in ga v tabeli poslji se nismo oznacili,
       moramo zdaj označiti njega in njegove nadrejene.
       Če pa smo ga ze kdaj prej oznacili (npr. ker ima kakšnega
       gajstnega podrejenega), smo takrat označili tudi njegove
       nadrejene in se nam zdaj z njim ni treba še enkrat ukvarjati. */
    if (Gajsten(i) && ! poslji[i])
      for (v = i; v >= 0; v = Nadrejeni(v)) poslji[v] = true;
  /* Ukaze moramo poslati tistim, ki so označeni v tabeli poslji
     in ki nimajo svojega nadrejenega. */
  for (i = 0, stUkazov = 0; i < n; i++)
    if (poslji[i] && Nadrejeni(i) < 0) stUkazov++;
  printf("%d\n", stUkazov); free(poslji); return 0;
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Ovce

Če damo strižcu i postriči eno ovco, je skupni strošek dela in elektrike enak $p_i + c \cdot t_i$; recimo temu q_i. Ni se težko prepričati, da je koristno dati največ dela tistim strižcem, pri katerih je ta skupni strošek q_i najmanjši. Recimo namreč, da imamo dva strižca, i in j, pri čemer je $q_i < q_j$; potem, če damo j-ju ostriči eno ovco manj in i-ju eno ovco več, bo skupni strošek manjši za $q_j - q_i$. Pri tem moramo paziti le na to, da posamezni strižec ne sme dobiti več ovac, kot jih bo lahko v t časa ostrigel, torej $\lfloor t/t_i \rfloor$. Zapišimo postopek še s psevdokodo:

za vsakega strižca izračunajmo $q_i = p_i + c \cdot t_i$
$c := 0$; (* *skupna cena striženja* *)
pregledujmo strižce po naraščajočem vrstnem redu q_i:
  naj bo i trenutni strižec;
  $n_i := \min\{n, \lfloor t/t_i$; (* *ta strižec naj postriže* n_i *ovac* *)
  $c := c + n_i \cdot q_i$; $n := n - n_i$;

Če pade med izvajanjem te zanke n na 0, pomeni, da smo razdelili že vse ovce in lahko takoj končamo. Če pa bi se zgodilo, da bi prišli do konca strižcev, n pa bi bil še vedno večji od 0, bi pomenilo, da vseh ovac v času t sploh ne bomo mogli ostriči; vendar naloga zagotavlja, da do tega ne bo prišlo.

2. Spričevala

Koristno si je v neki tabeli za vsakega ravnatelja zapisovati, na kateri šoli dela (v spodnjem programu je to tabela rs). Če opazimo novo spričevalo, v katerem piše, da ta

ravnatelj dela na neki drugi šoli, kot piše v tisti tabeli, pa vemo, da moramo izpisati opozorilo.

Razmisliti pa moramo še o dveh podrobnosti. Eno je, da moramo opisano tabelo po vsakem letu pobrisati (kajti če isti ravnatelj dela na različnih šolah v različnih letih, to še ni razlog za izpisovanje opozoril). Temu se lahko izognemo tako, da imamo še eno tabelo, v kateri za vsakega ravnatelja piše, v katerem letu smo ga nazadnje videli; v spodnjem programu je to tabela rl. Potem vemo, da moramo podatke v tabeli rs ignorirati, če se ne nanašajo na trenutno leto.

Druga podrobnost je, da nočemo enakega opozorila izpisovati po večkrat. Moramo si torej nekako zapomniti, da smo za nekega ravnatelja v trenutnem letu že izpisali opozorilo. Spodnji program si to zapomni tako, da v ustrezno celico tabele rs zapiše -1.

Doslej smo razmišljali o opozorilih za ravnatelje, enak razmislek pa bi lahko uporabili tudi za šole (da opozorimo, če ima ista šola v istem letu več ravnateljev), le vloge šol in ravnateljev so obrnjene. Spodnji program ima v ta namen še dve tabeli, sr in sl.

```
#include <stdio.h>

#define MaxS 1000000
#define MaxR 1000000

int rl[MaxR], rs[MaxR], sl[MaxS], sr[MaxS];

int main()
{
  int nSol, nRavnateljev, n, s, r, leto;
  scanf("%d %d %d", &nSol, &nRavnateljev, &n);
  for (r = 0; r < nRavnateljev; r++) rl[r] = −1, rs[r] = −1;
  for (s = 0; s < nSol; s++) sl[s] = −1, sr[s] = −1;
  while (n−− > 0)
  {
    scanf("%d %d %d", &leto, &s, &r);
    if (leto != sl[s]) sl[s] = leto, sr[s] = r;
    else if (sr[s] >= 0 && r != sr[s]) {
      sr[s] = −1; printf("Preveri spricevala sole st. %d v letu %d.\n", s, leto); }
    if (leto != rl[r]) rl[r] = leto, rs[r] = s;
    else if (rs[r] >= 0 && s != rs[r]) {
      rs[r] = −1; printf("Preveri spricevala s podpisom ravnatelja "
                    "st. %d v letu %d.\n", r, leto); }
  }
  return 0;
}
```

3. Razpolovišče lika

Najprej preglejmo celotno mrežo in preštejmo, koliko je črnih polj; tako dobimo površino kraljestva. Nato preglejmo mrežo še enkrat, po vrsticah. Recimo, da je celotna površina kraljestva p, v dosedanjih vrsticah (pred trenutno) smo našteli d črnih polj, v trenutni vrstici pa v črnih polj. Mejo moramo potegniti v tisti vrstici, v kateri $d + v$ preseže $p/2$. Recimo, da pokriva trenutna vrstica na y-osi interval $[y, y + 1]$ in da potegnemo mejo na višini $y + t$ (za nek $t \in [0, 1]$). Južno od meje torej ostane $d + t \cdot v$ enot površine; to mora biti enako $p/2$, če hočemo površino res razpoloviti. Tako imamo $d + tv = p/2$ oz. $t = (p/2 - d)/v$ oz. $t = (p - 2d)/(2v)$.

```
double Razpolovisce()
{
  int povrsina = 0, pDoslej, pVrstice, x, y;
  /* V prvem prehodu izračunajmo površino kraljestva. */
  for (y = 0; y < h; y++) for (x = 0; x < w; x++)
    if (Znotraj(x, y)) povrsina++;
  /* V drugem prehodu določimo položaj meje. */
  for (y = 0, pDoslej = 0; y < h; y++)
  {
    for (x = 0, pVrstice = 0; x < w; x++)
```

```
      if (Znotraj(x, y)) pVrstice++;
    /* Ali smo (skupaj s trenutno vrstico) že dosegli polovico površine? */
    if (2 * (pDoslej + pVrstice) >= povrsina)
      /* Izračunajmo točno višino razmejitvene črte. */
      return y + (povrsina − 2 * pDoslej) / (2.0 * pVrstice);
    pDoslej += pVrstice;
  }
}
```

4. Strukturirani podatki

Trenutno odprte značke si je koristno shranjevati na nekakšnem skladu (ki je lahko predstavljen v tabeli ali pa kot seznam, povezan s kazalci). Vhodno datoteko beremo po vrsticah; ko naletimo na začetno značko, jo dodamo na sklad; ko naletimo na končno značko, jo pobrišemo z vrha sklada; ko pa naletimo na običajno vrstico besedila, se sprehodimo po skladu (od dna proti vrhu) in izpišemo imena značk s sklada (ločena s pikami), na koncu pa še trenutno vrstico besedila.

Spodnja rešitev ima sklad predstavljen kot seznam, povezan s kazalci, čisto dobra rešitev pa bi bila tudi s tabelo 100 elementov (saj naloga pravi, da značke ne bodo gnezdene več kot 100 nivojev globoko).

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

typedef struct Znacka_{
  char *znacka;
  struct Znacka_*nasl, *prej;
} Znacka;

int main()
{
  char buf[MaxDolz + 2], *p, n;
  Znacka *sklad = 0, *vrh = 0, *z;
  while (fgets(buf, sizeof(buf), stdin))
  {
    /* Preberimo naslednjo vrstico, preskočimo presledke na začetku in
       odrežimo znak za konec vrstice na koncu. */
    p = &buf[0]; while (*p == ' ') p++;
    n = strlen(p); if (n > 0 && p[n − 1] == '\n') p[−−n] = 0;
    /* Če imamo začetno značko, jo dodajmo (brez znaka +) na vrh sklada. */
    if (*p == '+') {
      z = (Znacka *) malloc(sizeof(Znacka));
      z−>znacka = (char *) malloc(n);
      strcpy(z−>znacka, p + 1);
      z−>prej = vrh; z−>nasl = 0;
      if (vrh) vrh−>nasl = z; else sklad = z;
      vrh = z; }

    /* Če imamo končno značko, pobrišimo zadnji zapis z vrha sklada. */
    else if (*p == '-') {
      free(vrh−>znacka); z = vrh−>prej;
      free(vrh); vrh = z; if (! vrh) sklad = 0; else vrh−>nasl = 0; }

    /* Sicer pa se sprehodimo po skladu, izpišimo trenutno gnezdene značke
       in za njimi še besedilo iz trenutne vrstice. */
    else {
      for (z = sklad; z; z = z−>nasl)
        printf("%s%s", z == sklad ? "" : ".", z−>znacka);
      printf(" %s\n", p); }
  }
  /* sklad mora biti zdaj prazen, če je dokument sintaktično pravilen. */
  return 0;
}
```

5. Največji pretok

Program teče v neskončni zanki in v vsaki iteraciji poskuša prebrati meritev (iz kanala M) in maksimuma, ki mu ju pošiljata njegov desni in spodnji sosed (prek kanalov D in S). Največjo doslej prebrano vrednost hrani v spremenljivki max in jo v vsaki iteraciji pošlje svojemu zgornjemu in levemu sosedu. Tako pri vsakem računalniku max hrani največjo doslej prebrano vrednost v celotnem delu mreže, ki leži spodaj in desno od njega. Računalnik v zgornjem levem kotu mreže tako dobi največjo meritev iz cele mreže in jo izpiše na svoj levi kanal, kjer jo bomo lahko prebrali.

```
int main()
{
  int x, max = 0;
  for ( ; ; )
  {
    if ((x = Beri('M')) > max) max = x;
    if ((x = Beri('D')) > max) max = x;
    if ((x = Beri('S')) > max) max = x;
    Pisi('Z', max);
    Pisi('L', max);
  }
}
```

Primera, ko kakšno branje ne uspe, nam ni treba preverjati posebej, saj naloga pravi, da Beri takrat vrne -1, to pa na naš maksimum ne bo vplivalo (saj je max že od vsega začetka ≥ 0).

REŠITVE NALOG ZA TRETJO SKUPINO

1. de-FFT permutacija

Naloge se lahko lotimo z rekurzijo. Recimo, da imamo kot šifrirano sporočilo nek niz t, dolg 11 znakov. Ker se ob šifriranju s funkcijo FFT dolžina sporočila ne spreminja, je moralo biti tudi prvotno sporočilo (recimo s) dolgo 11 znakov. Iz definicije funkcije FFT vidimo, da bi ta iz niza 11 znakov najprej naredila dva niza, enega dolžine 6 (namreč $s_1 s_3 \ldots s_{11}$) in enega dolžine 5 (namreč $s_2 s_4 \ldots s_{10}$), zakodirala vsakega od njiju in tako dobljena šifrirana niza spet staknila. Torej tvori prvih 6 znakov niza t ravno $\text{FFT}(s_1 s_3 \ldots s_{11})$, preostalih 5 znakov niza t pa tvori niz $\text{FFT}(s_2 s_4 \ldots s_{10})$. Torej lahko z rekurzivnim klicem najprej dešifriramo $t_1 \ldots t_6$, da dobimo $s_1 s_3 \ldots s_{11}$, in dešifriramo $t_7 \ldots t_{11}$, da dobimo $s_2 s_4 \ldots s_{10}$; potem pa moramo njune znake le še preplesti med seboj, da dobimo prvotni niz $s_1 s_2 \ldots s_{10} s_{11}$. Enak razmislek seveda deluje pri vsaki dolžini, ne le 11; če imamo niz dolžine k, ga moramo razbiti na prvi del dolžine $\lceil k/2 \rceil$ in drugi del dolžine $\lfloor k/2 \rfloor$. Robni primer rekurzije pa nastopi pri $k \leq 2$, saj pri tako kratkih nizih vemo, da sta šifrirano in prvotno sporočilo enaki.

Zelo elegantno pa lahko nalogo rešimo tudi z naslednjim razmislekom. Za potrebe tega razmisleka bo lažje, če bomo položaje znakov v nizu šteli od 0 naprej namesto od 1 naprej. Recimo zdaj, da imamo nek niz s dolžine k, vendar ga pred šifriranjem še toliko podaljšajmo (z nekaj posebnimi znaki, ki se v njem sicer ne pojavljajo; recimo #), da bo njegova dolžina potenca števila 2 (recimo 2^b za $b = \lceil \log_2 k \rceil$). Tako podaljšani niz (recimo mu $\hat{s}$) zašifrirajmo in dobimo $\hat{t} = \text{FFT}(\hat{s})$. Kje v nizu $\hat{t}$ je pristal posamezni znak niza $\hat{s}$, recimo s_i? Izkaže se (o tem se lahko prepričamo z indukcijo po b), da če i zapišemo kot b-bitno število v dvojiškem zapisu in potem te bite preberemo od desne proti levi, dobimo ravno indeks, na katerem je v $\hat{t}$ po šifriranju pristal znak s_i. Niz $\hat{t}$ je torej zelo enostavno dešifrirati v $\hat{s}$: vsak indeks i od 0 do $2^b - 1$ zapišemo kot b-bitno število, mu bite obrnemo in dobimo novi indeks, na katerega moramo v $\hat{s}$ zapisati t_i.

Toda naš program kot vhodni niz seveda ne dobi $\hat{t}$, pač pa $t = \text{FFT}(s)$, torej šifro prvotnega nepodaljšanega niza s. Na srečo se izkaže, da če iz $\hat{t}$ pobrišemo vse znake #, dobimo ravno t. Torej si lahko $\hat{t}$ sestavimo „v mislih“: za vsak indeks od 0 do $2^b - 1$ si z obračanjem bitov izračunamo, iz katerega indeksa v nizu $\hat{s}$ bi prišel i-ti znak niza $\hat{t}$; če je ta indeks $\geq k$, vemo, da ima $\hat{t}$ tukaj znak #, sicer pa ima tukaj naslednji znak niza t.

```
#include <stdio.h>

#define MaxDolz 1024

int main()
{
  FILE *f = fopen("defft.in", "rt");
  FILE *g = fopen("defft.out", "wt");
  char x[MaxDolz + 1], y[MaxDolz + 3], *ok2_;
  int ok_, n, ok, d, b, i, ix, iy, bb;
  fscanf(f, "%d\n", &n);
  while (n-- > 0)
  {
    fgets(y, sizeof(y) - 1, f);
    d = 0; while (y[d] == '_' || ('a' <= y[d] && y[d] <= 'z')) d++;
    b = 0; while (d > (1 << b)) b++;
    x[d] = 0;
    for (i = 0, iy = 0; i < (1 << b); i++)
    {
      for (bb = 0, ix = 0; bb < b; bb++)
        if (i & (1 << bb)) ix |= 1 << (b - 1 - bb);
      if (ix >= d) continue;
      x[ix] = y[iy++];
    }
    fprintf(g, "%s\n", x);
  }
  fclose(f); fclose(g); return 0;
}
```

2. Potovanje

Naj bo p_i najbolj vzhodna črpalka, ki jo lahko dosežemo, če začnemo vožnjo na črpalki i s praznim tankom; in naj bo h_i količina goriva, s katero dosežemo to črpalko (preden tankamo na njej). Z „najbolj vhodna“ mislimo to, da tudi če potem na p_i načrpamo vseh g_i enot goriva, ki so tam na voljo, še vseeno ne bomo mogli doseči črpalke $p_i + 1$. Dolžina celotnega potovanja bo v tem primeru $x_{p_i} - x_i + g_i + h_i$. Vprašanje je le še, kako do učinkovito izračunati za vse i.

Izkaže se, da je vrednosti p_i in h_i koristno računati od vzhoda proti zahodu. Recimo, da začnemo pri i s praznim tankom; seveda lahko takoj natočimo g_i enot goriva. Če je $x_{i+1} - x_i > g_i$, ne bomo mogli doseči niti naslednje črpalke, torej je $p_i = i$, $h_i = 0$ in smo s tem i končali. Če pa lahko črpalko $i + 1$ dosežemo (in to z $g' := g_i - (x_{i+1} - x_i)$ goriva v tanku), si lahko v nadaljevanju pomagamo s prej izračunanimi rezultati: če bi začeli v $i + 1$ s praznim tankom, bi lahko prišli do p_{i+1} s h_{i+1} goriva; mi pa smo prišli do $i + 1$ z g' goriva, ne s praznim tankom, torej bomo prišli lahko do p_{i+1} s kar $g'' := g' + h_{i+1}$ goriva. Tam se bomo potem lahko naprej ukvarjali s tem, kako daleč se da priti, če začnemo v p_{i+1} z g'' goriva.

O časovni zahtevnosti tega postopka lahko razmislimo takole. Tabela p_i nam definira kup „skokov“ $i \to p_i$, torej predstavlja nekakšen graf. Na vsaki iteraciji zunanje zanke (tiste, ki pregleduje začetne postaje i od vzhoda proti zahodu) dodamo v graf eno povezavo, $i \to (i + 1)$, nato pa izračunamo p_i in celotno verigo povezav $i \to (i + 1) \to p_{i+1} \to \ldots \to p_i$ zamenjamo z enim samim skokom $i \to p_i$. V nadaljevanju namreč teh vmesnih povezav ne bomo nikoli več uporabili, saj bomo vedno, ko bomo prišli do i, od tam nemudoma skočili na p_i. V vsaki iteraciji dodamo le $O(1)$ povezav, torej je z dodajanjem povezav skupno le $O(n)$ dela; in ker gremo po vsaki povezavi največ enkrat (saj jo bomo takoj zatem pobrisali), je tudi s sprehajanjem po povezavah le $O(n)$ dela. Časovna zahtevnost tega postopka je torej le $O(n)$.

```
#include <stdio.h>

#define MaxX 1000000000
#define MaxG 1000000000
#define MaxN 1000000
```

```
/* Crpalka i je na polozaju xi[i] in na njej je gi[i] goriva. */
int xi[MaxN], gi[MaxN];
/* Ce zacnemo na crpalki i s praznim tankom, je najbolj desna crpalka,
   ki jo se lahko dosecemo, crpalka najP[i], do nje pa pridemo s
   najG[i] enotami goriva v tanku. */
int najP[MaxN];
long long najG[MaxN];

int main()
{
  int i, p, n; long long g;
  FILE *f = fopen("potovanje.in", "rt");
  fscanf(f, "%d\n", &n);
  for (i = 0; i < n; i++)
    fscanf(f, "%d %d\n", &xi[i], &gi[i]);
  fclose(f);

  for (i = n - 1; i >= 0; i--)
  {
    najP[i] = i; najG[i] = 0;
    p = i; g = 0;
    while (p < n - 1)
    {
      /* Ali je znan skok od p do najP[p]? */
      if (najP[p] > p) { g += najG[p]; p = najP[p]; continue; }
      /* Če ne, ali se lahko vsaj pripeljemo od p do p + 1? */
      if (xi[p + 1] - xi[p] > g + gi[p]) break;
      g = g + gi[p] - (xi[p + 1] - xi[p]); p++;
    }
    najP[i] = p; najG[i] = g;
  }

  f = fopen("potovanje.out", "wt");
  for (i = 0; i < n; i++) fprintf(f, "%lld\n", najG[i] + xi[najP[i]] - xi[i] + gi[najP[i]]);
  fclose(f); return 0;
}
```

3. Leteči pujsi

Strnjeni skupini opek, ki ležijo v isti vrstici, bomo rekli *prečka*. Če so vse opeke na prečki stabilne, bomo rekli, da je tudi cela prečka stabilna. V začetnem stanju mreže so bile torej stabilne vse prečke. Pri opekah pravi naloga, da lahko opeka dobi svojo stabilnost z leve, z desne ali pa od spodaj; prečka pa dobi svojo stabilnost le od spodaj, saj z leve in desne meji na prosta polja (ali na rob mreže). Prečka je torej stabilna le, če jo od spodaj podpira vsaj ena druga stabilna prečka (ali pa če leži v najnižji vrstici mreže).

Recimo, da se pujs zaleti v polje (x, y). Če je to polje prosto, se ne zgodi nič zanimivega in lahko takoj končamo. Recimo torej, da je to polje opeka; doslej je bila, tako kot vse opeke na mreži, stabilna, zdaj pa jo je pujs destabiliziral. Kako to vpliva na preostanek mreže?

Ker se stabilnost širi le gor, levo in desno, ne pa tudi dol, so opeke na nižjih vrsticah (pod y) še vedno stabilne. Opeka (x, y) pripada neki prečki, ki leži v vrstici y in pokriva recimo interval $p_l \ldots p_d$. Doslej je bila cela prečka stabilna, zdaj pa polje (x, y) ni več stabilno; kako to vpliva na stabilnost preostanka prečke? Oglejmo si na primer del desno od (x, y). Če je ta del prečke, od $x+1$ do p_d, spodaj podprt s kakšnimi drugimi opekami, je še vedno stabilen; če pa ne, je doslej svojo stabilnost dobival prek opeke (x, y) in je zdaj ta del prečke v ceoti nestabilen. Podobno lahko razmišljamo tudi za levi del prečke, od p_l do $x - 1$. Rezultat je torej ta, da je v vrstici y destabiliziran nek interval opek (ki se začne pri p_l ali x in se konča pri x ali p_d). Druge prečke v vrstici y pa so gotovo še vedno stabilno, saj so morali svojo stabilnost dobiti od spodaj, v spodnjih vrsticah pa se ni nič spremenilo.

Zdaj smo torej videli, da je v vrstici y destabiliziran nek interval, recimo $u..v$. Kaj to pomeni za dogajanje eno vrstico višje, v $y+1$? V tej vrstici so tiste prečke, ki ležijo v

celoti znotraj intervala $u..v$, izgubile vso podporo od spodaj in so torej tudi same postale nestabilne. Levo od njih je mogoče prečka, ki leži le delno nad $u..v$, njen levi konec pa sega levo od u; če ima ta prečka na tem levem delu še kakšno drugo podporo, je še vedno stabilna, drugače pa je tudi ta v celoti nestabilna. Podobno lahko razmišljamo tudi na desnem koncu intervala $u..v$. Tako lahko določimo interval nestabilnosti v novi vrstici $y + 1$.

S tem postopkom lahko zdaj nadaljujemo gor po mreži in tako v vsaki vrstici določimo interval nestabilnosti. Da izračunamo krajišči intervala v naslednji vrstici iz krajišč intervala v prejšnji vrstici, je koristno, če imamo za poljubno polje pri roki podatek o krajiščih prečke, ki jima pripada; če pa je neko polje prosto, leži med dvema prečkama in je koristno imeti podatek o desnem koncu leve prečke in levem koncu desne prečke. Spodnji program ima v ta namen tabeli levo in desno. S temi podatki bomo lahko novi krajišči izračunali v $O(1)$ časa. Poleg tega je koristno imeti za vsako polje tudi podatek o tem, koliko je črnih polj levo od njega v njegovi vrstici (tabela crne); potem moramo le odšteti dve taki vrednosti, pa dobimo število črnih polj na nekem intervalu. To naredimo za interval nestabilnosti v vsaki vrstici, rezultate seštevamo in na koncu dobimo skupno število nestabilnih opek. Tako za vsak možni začetni položaj pujsa porabimo le $O(h)$ časa, da ugotovimo, koliko opek je zdaj nestabilnih. Ker je možnih začetnih položajev $O(wh)$, ima naš postopek časovno zahtevnost $O(w \cdot h^2)$.

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

int h, w;
char t[305][305];

/* crne[i][j] = število črnih polj v t[i][1..j]
   levo[i][j] in desno[i][j] = če je polje (i, j) opeka, sta to
   krajišči njene prečke; sicer sta to koordinati najbližje
   opeke v vrstici i levo in desno od j. */
int levo[305][305], desno[305][305], crne[305][305];

int main()
{
  int i, j, k, nOpek, nNestabilnih, x1, x2, y1, y2;
  FILE *f = fopen("pujsi.in", "rt");

  /* Preberimo vhodne podatke. */
  fscanf(f, "%d %d", &h, &w);
  memset(t, '.', sizeof(t));
  for (j = 1; j <= w; j++) t[0][j] = '#';
  for (i = h; i >= 1; i--) fscanf(f, "%s", t[i] + 1);
  fclose(f);

  /* Pripravimo si tabele levo, desno in crne. */
  nOpek = 0;
  for (i = 1; i <= h; i++)
  {
    crne[i][0]=0;
    levo[i][0]=0;
    for (j = 1; j <= w; j++)
    {
      crne[i][j] = crne[i][j - 1];
      if (t[i][j] == '#') {
        nOpek++; crne[i][j]++;
        if (t[i][j-1] == '#') levo[i][j] = levo[i][j - 1];
        else levo[i][j] = j; }
      else {
        if (j==1 || t[i][j - 1]=='#') levo[i][j] = j - 1;
        else levo[i][j] = levo[i][j - 1]; }
    }
    desno[i][w + 1] = w + 1;
    for (j = w; j >= 1; j--)
    {
```

```
        if (t[i][j] == '#') {
          if (t[i][j + 1] == '#') desno[i][j] = desno[i][j + 1];
          else desno[i][j] = j; }
        else {
          if (j == w || t[i][j + 1] == '#') desno[i][j] = j + 1;
          else desno[i][j] = desno[i][j + 1]; }
      }
    }

  /* Izračunajmo rezultate in jih izpišimo. */
  f = fopen("pujsi.out", "wt");
  for (i = h; i >= 1; i--)
  {
    for (j = 1; j <= w; j++)
    {
      if (j != 1) fprintf(f, " ");
      if (t[i][j] == '.') fprintf(f, "%d", nOpek);
      else
      {
        int x1 = j, x2 = j;
        if (t[i - 1][j - 1] == '.' && levo[i - 1][j - 1] < levo[i][j]) x1 = levo[i][j];
        if (t[i - 1][j + 1] == '.' && desno[i][j] < desno[i - 1][j + 1]) x2 = desno[i][j];
        int nNestabilnih = x2 - x1 + 1;
        for (k = i + 1; k <= h; k++)
        {
          /* Interval nestabilnosti v vrstici k - 1 je x1..x2.
              Poglejmo zdaj, katere prečke ležijo v vrstici k vsaj delno na
              tem intervalu. Naj bo y1 levo krajišče najbolj leve izmed teh
              prečk, y2 pa desno krajišče najbolj desne. */
          if (t[k][x1] == '#') y1 = levo[k][x1]; else y1 = desno[k][x1];
          if (t[k][x2] == '#') y2 = desno[k][x2]; else y2 = levo[k][x2];
          /* Najbolj leva in najbolj desna od prečk na intervalu y1..y2 imata
              mogoče še kakšno podporo zunaj tega intervala; če je tako, ju iz
              intervala izvzamemo. */
          if (y1 < x1 && (t[k - 1][y1] == '#' || desno[k - 1][y1] < x1))
            y1 = desno[k][desno[k][y1] + 1];
          if (x2 < y2 && (t[k - 1][y2] == '#' || x2 < levo[k - 1][y2]))
            y2 = levo[k][levo[k][y2] - 1];
          /* Tako smo dobili pravi interval nestabilnosti v vrstici k. */
          x1 = y1; x2 = y2;
          if (x2 < x1) break;
          nNestabilnih += crne[k][x2] - crne[k][x1 - 1];
        }
        fprintf(f, "%d", nOpek - nNestabilnih);
      }
    }
    fprintf(f, "\n");
  }
  fclose(f); return 0;
}
```

4. Nakup parcele

Recimo, da nas zanimajo pravokotniki višine n in širine m. Razdelimo v mislih naše zemljišče na „celice“ velikosti $(n-1) \times (m-1)$. Za vsako parcelo (x, y) naj bo $zl(x, y)$ minimum vrednosti vseh parcel (x', y'), ki so isti celici kot (x, y) in ležijo zgoraj levo od nje, torej zanje velja $x' \leq x$ in $y' \leq y$. Podobno definirajmo še $zd(x, y)$ za območje zgoraj desno od (x, y) (torej $x' \geq x$, $y' \leq y$), $sl(x, y)$ za območje spodaj levo ($x' \leq x$, $y' \geq y$) in $sd(x, y)$ za območje spodaj desno ($x' \geq x$, $y' \geq y$). Z nekaj pazljivosti lahko vse te minimume izračunamo le v času $O((n-1)(m-1))$ za eno celico oz. $O(wh)$ za celotno zemljišče.

Opazimo lahko, da leži vsak pravokotnik velikosti $n \times m$ v natanko štirih celicah; njegov presek z zgornjo levo od teh štirih celic je eno od območij, o kakšrnih govori

tabela *sd*; njegov presek z zgornjo desno od teh celic je eno ob območij, o kakršnih govori tabela *sl*; in tako naprej. Minimum po celem pravokotniku bomo torej dobili tako, da vzamemo minimum primernih štirih vrednosti iz tabel *sl*, *sd*, *zl* in *zd* (za vogale pravokotnika). Tako imamo torej $O(wh)$ dela s pripravo tabel in potem še $O(1)$ z vsakim pravokotnikom, teh pa je tudi $O(wh)$, tako da ima celotni postopek (za enega kupca) časovno zahtevnost $O(wh)$.

```
#include <stdio.h>

#define MaxW 2000
#define MaxH 2000
#define MaxCena 1000000000
#define MaxKupcev 10

int a[MaxH][MaxW], zl[MaxH][MaxW], zd[MaxH][MaxW], sl[MaxH][MaxW], sd[MaxH][MaxW];

int Min(int a, int b) { return a < b ? a : b; }

int main()
{
  FILE *f = fopen("parcela.in", "rt");
  FILE *g = fopen("parcela.out", "wt");
  int ok_, w, h, x, y, k, wk, hk, r;
  int cxCell, cyCell, xCell, yCell, x0, y0, cw, ch;
  long long vsota;

  fscanf(f, "%d %d", &h, &w);

  for (y = 0; y < h; y++) for (x = 0; x < w; x++)
    fscanf(f, "%d", &a[y][x]);

  fscanf(f, "%d", &k);

  while (k-- > 0)
  {
    fscanf(f, "%d %d", &hk, &wk);
    cxCell = wk - 1; cyCell = hk - 1;
    for (yCell = 0; yCell * cyCell < h; yCell++)
    for (xCell = 0; xCell * cxCell < w; xCell++)
    {
      x0 = xCell * cxCell; y0 = yCell * cyCell;
      cw = Min(cxCell, w - x0);
      ch = Min(cyCell, h - y0);

      for (y = y0; y < y0 + ch; y++) {
        for (x = x0; x < x0 + cw; x++) {
          r = a[y][x];
          if (x > x0) r = Min(r, zl[y][x - 1]);
          if (y > y0) r = Min(r, zl[y - 1][x]);
          zl[y][x] = r; }
        for (x = x0 + cw - 1; x >= x0; x--) {
          r = a[y][x];
          if (x < x0 + cw - 1) r = Min(r, zd[y][x + 1]);
          if (y > y0) r = Min(r, zd[y - 1][x]);
          zd[y][x] = r; }}

      for (y = y0 + ch - 1; y >= y0; y--) {
        for (x = x0; x < x0 + cw; x++) {
          r = a[y][x];
          if (x > x0) r = Min(r, sl[y][x - 1]);
          if (y < y0 + ch - 1) r = Min(r, sl[y + 1][x]);
          sl[y][x] = r; }
        for (x = x0 + cw - 1; x >= x0; x--) {
          r = a[y][x];
          if (x < x0 + cw - 1) r = Min(r, sd[y][x + 1]);
          if (y < y0 + ch - 1) r = Min(r, sd[y + 1][x]);
```

```
        sd[y][x] = r; }}
    }

    vsota = 0;
    for (y = 0; y + hk <= h; y++) for (x = 0; x + wk <= w; x++)
    {
      /* Vogali okna zagotovo pripadajo razlicnim celicam. */
      r = Min(
        Min(sd[y][x], sl[y][x + wk − 1]),
        Min(zd[y + hk − 1][x], zl[y + hk − 1][x + wk − 1]));
      vsota += r;
    }
    fprintf(g, "%lld\n", vsota);
  }

  fclose(f); fclose(g); return 0;
}
```

5. Rotacija

Naj bo $s = s_0 \ldots s_{n-1}$ vhodni niz, ki opisuje naše kolo. Za začetek poiščimo največjo števko v nizu; najboljša možna rotacija našega niza se mora očitno začeti pri eni od pojavitev te števke. Ker pa še ne vemo, pri kateri, si jih vse zapišimo v nek seznam. Nato si za vsako od njih poglejmo, katero števko bi dobili na drugem mestu; med temi števkami si zapomimo največjo in obdržimo v seznamu le tiste kandidatke, ki imajo na drugem mestu res le to števko. Podobno nadaljujemo s števkami na tretjem mestu in tako naprej. Postopek se konča, če ostane le še ena kandidatka ali pa ko dosežemo dolžino n. S psevdokodo ga lahko zapišemo takole:

$m := \max\{s_0, s_1, \ldots, s_{n-1}\}$;
$L := \{i : s_i = m\}$; $d := 1$;
while $|L| > 1$ **and** $d < n$:
  (* *L je množica indeksov, na katerih se v s začne leksikografsko največji podniz dolžine d.* *)
  $m := \max\{s_{i+d} : i \in L\}$;
  $L' := \{i \in L : s_{i+d} = m\}$;
  $L := L'$; $d := d + 1$;

Pri tem si moramo predstavljati, da niz s naslavljamo ciklično: kjer se gornja psevdokoda sklicuje na s_i za $i \geq n$, moramo v resnici uporabiti $s_{i \bmod n}$.

Slabo pri tem postopku je, da ima lahko časovno zahtevnost $O(n^2)$; na primer, če so vse števke niza s enake, bomo šla glavna zanka vse do $d = n$ in v množici L bodo ves čas kar vsi indeksi od 0 do $n - 1$. Rešitev lahko izboljšamo z naslednjim razmislekom. Na začetku vsake iteracije glavne zanke velja naslednje: če poiščemo v s leksikografsko največji podniz dolžine d (pri tem si moramo ves čas predstavljati s kot cikličen niz) — recimo temu podnizu x — so v L vsi indeksi, na katerih se x pojavlja kot podniz v s. Po vsaki iteraciji glavne zanke nekatere pojavitve mogoče izpadejo iz L, tiste, ki ostanejo, pa se podaljšajo za en znak (ker se d poveča za 1). V nekem trenutku se mogoče zgodi, da se dve pojavitvi sprimet skupaj: v L najdemo torej nek i in za njim še $i + d$. Niz s lahko v mislih razdelimo na nekaj delov: $s = ux^kv$, pri čemer je u območje pred indeksom i, v pa območje za $i + kd$. Kakšne rotacije dobimo, če s zarotiramo za i, $i + d$, …, $i + kd$ mest? Pri rotaciji za i dobimo $s^{(i)} = x^kvu$; pri rotaciji za $i + d$ dobimo $s^{(i+d)} = x^{k-1}vux$; in v splošnem pri rotaciji za $i + td$ dobimo $s^{(i+td)} = x^{k-t}vux^t$. Če primerjamo $s^{(i+td)}$ (za $1 \leq t \leq k$) in $s^{(i)}$, vidimo, da se oba začneta na $k - t$ pojavitev niza x, zatem pa ima $s^{(i)}$ še eno pojavitev niza x, niz $s^{(i+td)}$ pa ima tam v (oz. prvih d znakov niza vux^t), kar je gotovo leksikografsko večje od x (saj bi bil drugače tudi indeks $i + (k + 1)d$ tudi še prisoten v L, mi pa smo predpostavili, da ni). Torej $s^{(i+td)}$ gotovo ne bo dal največjega dobitka, torej lahko indeks $i + td$ brez škode pobrišemo iz L.

S takšnim sprotnim brisanjem, čim se začnejo najdaljši podnizi dolžine d sprijemati, bomo tudi zagotovili, da se podnizi dolžine d z začetkom pri indeksih iz L nikoli ne bodo prekrivali. Pri dolžini d lahko torej L vsebuje največ n/d različnih indeksov. Skupna cena vseh podaljševanj vseh podnizov (oz. prenašanja indeksov iz L v L' pri posameznih

iteracijah glavne zanke) je torej $\leq \sum_{d=1}^{n}(n/d) = n\sum_{d=1}^{n}(1/d) \approx n \log n$. Tako smo časovno zahtevnost našega postopka izboljšali z $O(n^2)$ na $O(n \log n)$.

```
#include <stdio.h>

#define MaxN 1000000
char s[2 * MaxN];
int zacetki[MaxN], zacetki2[MaxN];

int main()
{
  FILE *f = fopen("rotacija.in", "rt");
  int n, nZacetkov, nZacetkov2, i, d;
  char cNaj;

  fscanf(f, "%d\n", &n);
  fgets(s, MaxN + 3, f);
  fclose(f);

  nZacetkov = 0;
  for (i = 0, cNaj = 0; i < n; i++) {
    s[n + i] = s[i];
    if (s[i] > cNaj) cNaj = s[i], nZacetkov = 0;
    if (s[i] == cNaj) zacetki[nZacetkov++] = i; }

  d = 1;
  while (nZacetkov > 1)
  {
    /* Pobrisimo neobetavne zacetke. */
    cNaj = 0;
    for (i = 0, nZacetkov2 = 0; i < nZacetkov; i++) {
      if (i > 0 && zacetki[i] == zacetki[i - 1] + d) continue;
      zacetki2[nZacetkov2++] = zacetki[i];
      if (s[zacetki[i] + d] > cNaj) cNaj = s[zacetki[i] + d]; }

    /* Pripravimo nov seznam zacetkov. */
    for (i = 0, nZacetkov = 0; i < nZacetkov2; i++)
      if (s[zacetki2[i] + d] == cNaj)
        zacetki[nZacetkov++] = zacetki2[i];
    d += 1;
  }

  f = fopen("rotacija.out", "wt");
  fwrite(&s[zacetki[0]], n, sizeof(s[0]), f);
  fclose(f); return 0;
}
```

Viri nalog: prepletene besede, ovce — Nino Bašić; največji pretok — Primož Gabrijelčič; leteči pujsi — Matija Grabnar; potovanje, nakup parcele, rotacija — Tomaž Hočevar; mase — Nace Hudobivnik; kazenski stavek — Jurij Kodre; manjkajoča števila, strukturirani podatki — Mark Martinec; spričevala — Mojca Miklavec; v Afganistan!, razpolovišče lika, de-FFT permutacija — Mitja Trampuš. Hvala Tomažu Hočevarju za pomoč pri implementaciji rešitev nalog za 3. skupino.

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