C példatár

Juhász, István

Kósa, Márk

Pánovics, János

Édelkraut, Róbert

A tananyag a TÁMOP-4.1.2-08/1/A-2009-0046 számú Kelet-magyarországi Informatika Tananyag Tárház projekt keretében készült. A tananyagfejlesztés az Európai Unió támogatásával és az Európai Szociális Alap társfinanszírozásával valósult meg.

Nemzeti Fejlesztési Ügynökség http://ujszechenyiterv.gov.hu/ *06 40 638-638*

2005

Tartalom

Az ábrák listája

16.7.
16.8.
17.1.
17.2.
18.1.
18.2.
18.3.
18.4.
18.5.
18.6.
19.1.
19.2.
19.3.
19.4.
19.5.

Előszó

Minden programozási nyelvet egyszerűbb és bonyolultabb feladatok **önálló** megoldásával, saját programok megírásával, lefuttatásával, tesztelésével lehet a legkönnyebben elsajátítani. Ugyanakkor egy programozási nyelv megtanulásánál sokat segítenek a mintaprogramok, receptek, programozási fogások. Különösen igaz ez a C-re, köszönhetően annak, hogy a C a magas szintű programozási nyelvek között a leginkább gépközeli nyelv.

A C már hosszú évek óta a procedurális jellegű alkalmazásfejlesztés nyelve, a gyakorlatban igen nagy népszerűségnek örvend. Ugyanakkor a felsőoktatásban is általánosan használt, és az utóbbi években a különböző szintű programozási versenyek egyik hivatalos nyelvévé vált.

A C példatár tankönyv, a C programozási nyelv gyakorlati alkalmazásába nyújt betekintést. A legegyszerűbb példáktól kezdve a nemzetközi programozói versenyeken előforduló feladatokig vezeti végig az olvasót, bemutatva a nyelv kínálta lehetőségeket egy-egy probléma vagy algoritmus hatékony kódolására. A példatár nem magát a nyelvet ismerteti. A feladatok megoldásaiban szereplő programkódok feltételezik a C nyelv ismeretét, nem térünk ki külön a nyelvi elemek tárgyalására, csupán az alkalmazásukra mutatunk példákat. Egyes feladatoknál azonban -- különösen a versenyfeladatoknál, ahol az algoritmusok bonyolultsága indokolja -- részletesen tárgyaljuk a választott algoritmus működését.

Ajánljuk ezt a könyvet a közép- vagy felsőoktatásban programozást tanuló diákok és hallgatók számára, valamint mindazoknak, akik jelenleg vagy a közeljövőben ismerkednek a programozással, és ehhez a C nyelvet választják. Hasznos ismereteket találhatnak benne azok a gyakorló szakemberek is, akik már jelentős programozói háttérrel most térnek át a C használatára.

Jelen tankönyv az utasítások használatát bemutató egyszerű feladatokat, a C egyéb nyelvi elemeit részletesen ismertető példákat, illetve a komplex algoritmikus gondolkodást igénylő versenyfeladatokat tartalmaz, s mindezekhez C nyelvű mintamegoldásokat is ad. Részleteiben az egyes fejezetek a következő témaköröket tárgyalják:

- Az első részben a C nyelv eszközeinek felhasználásával különböző egyszerű algoritmusokat kódolunk. A példák végigvezetnek az egyszerű és származtatott adattípusokon, a legtöbb utasításon, és bemutatják az állománykezelési technikákat, valamint az előfordítói direktívák és a változó paraméterszámú függvények használatát.
- A második rész különböző típusú versenyfeladatok megoldásával illusztrálja a C nyelv széles körű alkalmazási lehetőségét. A feladatokat a következő témák közül választottuk: matematikai, szimulációs, sakk- és gráfproblémák (labirintusokkal kapcsolatos feladatok),

dinamikus programozási példák és adott formátumú kimenetet előállító programok.

- A harmadik rész tartalmazza a 2002-es és 2003-as évek közép-európai és nemzetközi diákolimpiai döntőinek, valamint az ACM programozói versenyek közép-európai döntőinek teljes feladatsorait. Ezekhez a feladatokhoz nem készítettünk példamegoldásokat, azok elkészítését -- bízva felkészültségében és lelkesedésében -- az olvasóra bízzuk.

A könyv megoldásai egyfajta programozási stílust vallanak. Természetesen bárki találhat jobb, gyorsabb, elegánsabb megoldást. Az is előfordulhat, hogy a közölt kódok hibásak. A tartalommal kapcsolatban minden javító szándékú megjegyzést, kiegészítést, helyesbítést szívesen fogadunk az `mkosa@inf.unideb.hu` vagy a `panovics@inf.unideb.hu` e-mail címen. A könyvben közölt programrészek kódjai letölthetők a

`http://infotech.inf.unideb.hu/konyvek/cpeldatar/index.html`

címről, ugyanitt jelenik majd meg (a reményeink szerint kevés számú) hibalista is.

Debrecen, 2004. szeptember 8.

Juhász István

Kósa Márk

Pánovics János

1. fejezet - Bevezetés

Tartalom

Minden programozási nyelvet egyszerűbb és bonyolultabb feladatok önálló megoldásával lehet a legkönnyebben elsajátítani. Különösen igaz ez a C nyelvre, amely az 1990-es évek elejétől kezdve terjedt el robbanásszerűen, köszönhetően általánosságának, egyszerűségének, tömörségének, és annak, hogy a magas szintű programozási nyelvek között a leginkább gépközeli nyelv.

A felsorolt tulajdonságok miatt a C szokatlan lehet azok számára, akik korábban más, magasabb szintű nyelveken programoztak. A példatár első részében igyekszünk nagyon egyszerű programkódokat bemutatni, amelyek illusztrálják a nyelv egyszerűségét, tömörségét, és mégis érthetőek a kezdő C-programozók számára is.

Mindenekelőtt azonban a nyelv tömörségét bizonyítandó, álljon itt a C-programozói berkekben[1] jól ismert rövid programkód, amely remélhetőleg nem fog elriasztani senkit a könyv további részeinek tanulmányozásától:

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#\
n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw\
' i;# ){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk \
nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{\
n' ')# }'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),
a+1);}
```

Bármennyire is hihetetlen, ez a program -- fordítás és futtatás után -- a következő, angolszász nyelvterületen népszerű gyermekverset írja a képernyőre:

```
On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a pear tree.

On the fourth day of Christmas my true love gave to me
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the fifth day of Christmas my true love gave to me
five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the sixth day of Christmas my true love gave to me
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the seventh day of Christmas my true love gave to me
seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the eigth day of Christmas my true love gave to me
eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the ninth day of Christmas my true love gave to me
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the tenth day of Christmas my true love gave to me
ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the eleventh day of Christmas my true love gave to me
eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the twelfth day of Christmas my true love gave to me
twelve drummers drumming, eleven pipers piping, ten lords a-leaping,
```

```
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.
```

Hasznos programozási tanácsok

A példatárban -- a *6. C-implementációk* című fejezetet kivéve -- az ANSI-szabványnak megfelelő programkódokat mutatunk be. Ezért nincsenek a példák között olyanok, amelyek az ANSI-szabványban nem rögzített eszközöket (például konzol I/O-t, grafikát, hálózati szolgáltatásokat, multiprogramozást) használnak.

Az ANSI-szabvány az eredeti C nyelvet sok új eszközzel bővítette, bizonyos területeken viszont szűkítette a szemantikát. A következőkben néhány praktikus programozási tanácsot adunk az ANSI-szabványnak leginkább megfelelő kódok elkészítéséhez [8].

Ötletek a hordozható kód előállításához

- A standard külső függvények deklarációjához használjuk a megfelelő header állományokat. Ezzel elkerülhetjük ugyanazon függvény különböző inkonzisztens deklarációit.
- Mindig használjunk függvényprototípusokat, és a függvényfejléceket a prototípusnak megfelelő alakban írjuk meg.
- Használjuk az `offsetof()` makrót a struktúratagok offseteinek meghatározására. Az `offsetof()` makró az `<stddef.h>`-ban van definiálva.
- Típuskonverzió esetén mindig használjunk explicit típuskonverziót.
- Vigyázzunk az olyan minősített objektumokkal, mint például a `volatile` és a `const`. Az ilyen objektumok címeit mindig csak hasonlóan minősített mutatókhoz rendeljük hozzá.
- Minden egyes visszatérési pontnál adjunk vissza értéket minden nem-void típusú függvény esetén.
- Csak megfelelő típusú struktúrát használjunk a `.` és a `->` operátorok bal oldalán (azaz ügyeljünk rá, hogy a jobb oldal egy létező mezője legyen a bal oldalon álló struktúrának).
- Minden bitmezőnél írjuk ki a `signed` vagy `unsigned` kulcsszót.

Kerülendő veszélyforrások

- Ne keverjük ugyanannak a függvénynek a prototípusos és nem prototípusos deklarációit.
- Sose hívjunk meg egy függvényt a deklarációja előtt. Ez inkompatibilis automatikus deklarációhoz vezethet.
- A paraméterek kiértékelési sorrendje nem definiált. (Mi lesz például a második paraméter értéke a foo(a++, a, ...) kódrészletben?)
- Kerüljük a mellékhatással rendelkező kifejezések függvényparaméterként valóhasználatát.
- Kerüljük az ugyanarra az adatra vonatkozó, közvetlenül egymás mellett elhelyezkedő mellékhatások sorozatát (például: `x=++x;`).
- Kerüljük a lokális környezetben való függvénydeklarációt, főleg ha a függvény rendelkezik prototípussal.
- Sose probáljunk meg hozzáférni olyan paraméterhez, amit nem tüntettünk fel a paraméterlistában, kivéve az `stdarg` eszközkészletet használva. Az `stdarg` eszközkészletet csak változó paraméterszámú függvények esetén használjuk (azaz csak

`...`-ra végződő paraméterlista esetén).

- Egy mutató típust sose konvertáljunk explicit típuskonverzióval más típusra, mint egy másik mutató típus, vagy egy vele azonos méretű egész típus (`unsigned long`), se fordítva. Ha a mutató bitmintáját nem egész és nem is mutató típusúként (hanem `char`-ok tömbjeként) akarjuk felhasználni, használjunk `union` típust.
- Ne trükközzünk az előfordító tokenjeivel (például `FOO/**/BAR`).
- Sose módosítsuk a sztring literálokat.
- Ne hagyatkozzunk az idézőjelek között megadott `include` állományok keresési sorrendjére.

A feladatok forrásai

A példatár feladatait több forrásból válogattuk.

Az 1. rész feladatainak nagy részét személyes, a Debreceni Egyetemen végzett oktatói és gyakorlatvezetői munkánk tapasztalatai alapján állítottuk össze. Az 1. részben más forrásból származnak a következő feladatok:

3.1. alfejezet, 3.7. feladat.

Középiskolai Matematikai Lapok, 2003. februári szám, I.43. feladat,

`http://www.komal.hu/verseny/2003-02/inf.h.shtml`

3.3. alfejezet, 3.40. feladat.

Programozási feladatok I., Kossuth Kiadó, Budapest, 1997 [6]

152. oldal, 4.39. feladat

A 2. rész feladatainak szövegei hazai és nemzetközi programozói versenyek feladatsoraiból, illetve internetes programozói feladatgyűjteményekből származnak:

7. 1 alfejezet, Pi.

Problem Set Archive, Universidad de Valladolid,

`http://acm.uva.es/p/v4/412.html`

7. 2 alfejezet, Goldbach sejtése.

Problem Set Archive, Universidad de Valladolid,

`http://acm.uva.es/p/v6/686.html`

7. 3 alfejezet, Vonatok.

Problem Set Archive, Universidad de Valladolid,

`http://acm.uva.es/p/v7/783.html`

7. 4 alfejezet, Egyiptomi törtek.

Középiskolai Matematikai Lapok, 2002. októberi szám, I.31. feladat,

`http://www.komal.hu/verseny/2002-10/inf.h.shtml`

7. 5 alfejezet, Számrendszerváltás.

Középiskolai Matematikai Lapok, 2002. januári szám, I.13. feladat,

`http://www.komal.hu/verseny/2002-01/szt.h.shtml`

8. 1 alfejezet, Josephus.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v3/305.html

8. 2 alfejezet, Veremváros.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v5/514.html

9. 1 alfejezet, NyargaLó.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v4/439.html

9. 2 alfejezet, Hány huszár.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v6/696.html

9. 3 alfejezet, A nyolc királynő problémája.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v7/750.html

10. 1 alfejezet, Jill kerékpározik.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v5/507.html

10. 2 alfejezet, Maximális összeg.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v1/108.html

11. 1 alfejezet, Az útvonal feltérképezése.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v6/614.html

11. 2 alfejezet, Labirintus.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v7/784.html

12. 1 alfejezet, Háromszöghullám.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v4/488.html

12. 2 alfejezet, LCD kijelző.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v7/706.html

13. 1 alfejezet, Szelektív hulladékgyűjtés.

Problem Set Archive, Universidad de Valladolid,

http://acm.uva.es/p/v1/102.html

13. 2 alfejezet, Szerelvényrendezés.

Problem Set Archive, Universidad de Valladolid,

`http://acm.uva.es/p/v2/299.html`

13. 3 alfejezet, Óramutatók.

Problem Set Archive, Universidad de Valladolid,

`http://acm.uva.es/p/v5/579.html`

13. 4 alfejezet, Milyen nap van?

Problem Set Archive, Universidad de Valladolid,

`http://acm.uva.es/p/v6/602.html`

13. 5 alfejezet, DNS rendezés.

Problem Set Archive, Universidad de Valladolid,

`http://acm.uva.es/p/v6/612.html`

A 3. részben található feladatsorok a Közép-európai Informatikai Diákolimpia, a Nemzetközi Informatikai Diákolimpia és az ACM által szervezett nemzetközi programozó verseny közép-európai döntőinek 2002. és 2003. évi versenyein szerepeltek. Eredeti angol nyelvű forrásaik megtalálhatók a következő címeken:

Közép-európai Informatikai Diákolimpia, 2002

Kassa, Szlovákia

`http://cs.science.upjs.sk/ceoi/`

Közép-európai Informatikai Diákolimpia, 2003

Münster, Németország

`http://www.ceoi2003.de/`

Nemzetközi Informatikai Diákolimpia, 2002

Yong-In, Dél-Korea

`http://olympiads.win.tue.nl/ioi/ioi2002/index.html`

Nemzetközi Informatikai Diákolimpia, 2003

Kenosha, USA

`http://www.ioi2003.org/`

ACM közép-európai döntő, 2002

Varsó, Lengyelország

`http://cepc.mimuw.edu.pl/2002/pages/problems.html`

ACM közép-európai döntő, 2003

Varsó, Lengyelország

`http://cepc.mimuw.edu.pl/pages/problems.html`

A feladatok tanulmányozásához és önálló megoldásához nagy türelmet, kitartást és sok sikert kívánunk!

[1] Lásd a *The International Obfuscated C Code Contest* honlapot a `http://www.ioccc.org` címen.

2. fejezet - Egyszerű feladatok

Tartalom

Egyszerű adattípusok és vezérlési szerkezetek

2.1. FELADAT. *Írjunk olyan programot, amely beolvas a billentyűzetről két számot: a-t és b-t, és ha b a nagyobb, akkor megcseréli őket!*

```
#include <stdio.h>

main()
{
  int a, b;
  printf( "a=" );  scanf( "%d", &a );
  printf( "b=" );  scanf( "%d", &b );
  if ( a < b )
  {
    int tmp = a;
    a = b;
    b = tmp;
  }
}
```

2.2. FELADAT. *Írjunk programot, amely egy zárthelyi dolgozatra adott pontszám alapján eldönti, hogy sikeres volt-e a dolgozat! A dolgozat akkor sikeres, ha a pontszám az elérhető maximumnak legalább a 60 százaléka.*

```
#include <stdio.h>

main()
{
  int elerheto, pontszam;
  printf( "elérhető=" );  scanf( "%d", &elerheto );
  printf( "pontszám=" );  scanf( "%d", &pontszam );
  if ( 100 * pontszam >= 60 * elerheto )
    puts( "A dolgozat sikeres." );
  else
    puts( "A dolgozat sikertelen." );
}
```

2.3. FELADAT. *Írjunk programot, amely egy beolvasott évszámról eldönti, hogy szökőév-e!*

```
#include <stdio.h>

main()
{
  int evszam;
  printf( "Évszám: " );  scanf( "%d", &evszam );
  if ( evszam % 4 == 0 && evszam % 100 != 0 || evszam % 400 == 0 )
    puts( "Szökőév." );
  else
    puts( "Nem szökőév." );
}
```

2.4. FELADAT. *Írjunk programot, amely eldönti, hogy három szakaszból szerkeszthető-e háromszög, és ha igen, akkor megadja a háromszög területét!*

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

main()
{
  double a, b, c;
  printf( "a=" );  scanf( "%lf", &a );
  printf( "b=" );  scanf( "%lf", &b );
  printf( "c=" );  scanf( "%lf", &c );
  if ( a > 0 && b > 0 && c > 0 && a + b > c && a + c > b && b + c > a )
  {
    double s = ( a + b + c ) / 2;
    puts( "Szerkeszthető háromszög a szakaszokból." );
    printf( "A háromszög területe: %lf.\n",
            sqrt( s * ( s - a ) * ( s - b ) * ( s - c ) ) );
  }
  else
    puts( "Nem szerkeszthető háromszög a szakaszokból." );
}
```

2.5. FELADAT. *Írjunk programot, amely értékel egy dolgozatot a rá adott pontszám alapján! Az értékelés a következő táblázat alapján történjen:*

Pontszám	*Értékelés*
0--42	elégtelen
43--57	elégséges
58--72	közepes
73--87	jó
88--100	jeles

```
#include <stdio.h>

main()
{
  int pont;
  printf( "Pontszám: " );  scanf( "%d", &pont );
  if ( pont < 0 || pont > 100 )
    puts( "Érvénytelen pontszám." );
  else if ( pont <= 42 )
    puts( "Elégtelen." );
  else if ( pont <= 57 )
    puts( "Elégséges." );
  else if ( pont <= 72 )
    puts( "Közepes." );
  else if ( pont <= 87 )
    puts( "Jó." );
  else
    puts( "Jeles." );
}
```

2.6. FELADAT. *Írjunk programot, amely az* a,b,c *értékek ismeretében képernyőre írja az* $ax^2+bx+c=0$ *másodfokú egyenlet megoldásait!*

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

main()
{
  double a, b, c;
  printf( "a=" );  scanf( "%lf", &a );
  printf( "b=" );  scanf( "%lf", &b );
  printf( "c=" );  scanf( "%lf", &c );
  if ( a == 0 )
  {
    if ( b == 0 )
    {
      if ( c == 0 )
        puts( "Az egyenlet azonosság. Megoldása minden valós szám." );
      else
        puts( "Az egyenlet ellentmondásos, nincs megoldása." );
    }
    else
    {
      puts( "Az egyenlet elsőfokú." );
      printf( "Megoldása: x=%lf\n", -c / b );
    }
  }
  else
  {
    double diszkriminans = b * b - 4 * a * c;
    if ( diszkriminans < 0 )
      puts( "Az egyenletnek a valós számok körében nincs megoldása." );
    else if ( diszkriminans == 0 )
    {
      puts( "Az egyenletnek két egybeeső megoldása van." );
      printf( "Ezek a következők: x1 = x2 = %lf\n", -b / ( 2 * a ) );
    }
    else
    {
      puts( "Az egyenletnek két különböző megoldása van." );
      printf( "Ezek a következők: x1 = %lf, x2 = %lf\n",
              ( -b + sqrt( diszkriminans ) ) / ( 2 * a ),
              ( -b - sqrt( diszkriminans ) ) / ( 2 * a ) );
    }
  }
}
```

2.7. FELADAT. *Írjunk programot, amely kiírja egy dolgozat szöveges értékelését az érdemjegy alapján!*

```
#include <stdio.h>

main()
{
  int jegy;
  printf( "Érdemjegy: " );  scanf( "%d", &jegy );
  switch ( jegy )
  {
    case 1:  puts( "elégtelen" ); break;
    case 2:  puts( "elégséges" ); break;
    case 3:  puts( "közepes" );   break;
```

```
    case 4:  puts( "jó" );        break;
    case 5:  puts( "jeles" );     break;
  }
}
```

2.8. FELADAT. *Írjunk programot, amely a hónapok valamelyikének sorszámát beírva kiírja a hónap nevét!*

1. Először elkészítjük a `switch`-szerkezetet használó programot.

```
#include <stdio.h>

main()
{
  int ho;
  printf( "A hónap sorszáma: " );  scanf( "%d", &ho );
  switch ( ho )
  {
    case  1:  puts( "január" );      break;
    case  2:  puts( "február" );     break;
    case  3:  puts( "március" );     break;
    case  4:  puts( "április" );     break;
    case  5:  puts( "május" );       break;
    case  6:  puts( "június" );      break;
    case  7:  puts( "július" );      break;
    case  8:  puts( "augusztus" );   break;
    case  9:  puts( "szeptember" );  break;
    case 10:  puts( "október" );     break;
    case 11:  puts( "november" );    break;
    case 12:  puts( "december" );    break;
    default:  puts( "Nincs ilyen sorszámú hónap" ); break;
  }
}
```

2. Most lássuk azt a megoldást, ahol a hónapok neveit egy tömbben tároljuk.

```
#include <stdio.h>

main()
{
  char *honapnev[] = { "január",  "február",   "március",
                       "április", "május",     "június",
                       "július",  "augusztus", "szeptember",
                       "október", "november",  "december" };
  int ho;
  printf( "A hónap sorszáma: " );  scanf( "%d", &ho );
  if ( 1 <= ho && ho <= 12 )
    puts( honapnev[ ho - 1 ] );
  else
    puts( "Nincs ilyen sorszámú hónap" );
}
```

2.9. FELADAT. *Írjunk programot, amely kiírja az első 10 természetes számot és azok négyzetét!*

```
#include <stdio.h>

main()
{
  int szam;
  for ( szam = 0; szam < 10; ++szam )
    printf( "%d %d\n", szam, szam * szam );
```

```
}
```

2.10. FELADAT. *Írjunk programot, amely a standard inputját átmásolja a standard outputjára!*

```
#include <stdio.h>

main()
{
  int ch;
  while ( ( ch = getchar() ) != EOF )
    putchar( ch );
}
```

2.11. FELADAT. *Írjunk programot, amely meghatározza az első n természetes szám összegét!*

1. Nézzük először azt a megoldást, amely egy ciklust használ az összeg meghatározására.

```
#include <stdio.h>

main()
{
  unsigned n, osszeg = 0, i;
  printf( "N = " );  scanf( "%u", &n );
  for ( i = 1; i <= n; osszeg += i++ )
    ;
  printf( "Az első %u természetes szám összege: %u\n", n, osszeg );
}
```

2. A feladatot meg lehet oldani egyszerűbben is, ha ismerjük az első n természetes szám összegének zárt képletét. Talán mondanunk sem kell, ez gyorsabb program, mint az előző.

```
#include <stdio.h>

main()
{
  unsigned n;
  printf( "N = " );  scanf( "%u", &n );
  printf( "Az első %u természetes szám összege: %u\n",
          n, n * ( n + 1 ) / 2 );
}
```

2.12. FELADAT. *Írjunk függvényt, amely meghatározza egy n ($n > 1$) természetes számnál nem nagyobb legnagyobb négyzetszámot!*

```
int negyzet( int n )
{
  int i;
  for ( i = 1; i * i <= n; ++i )
    ;
  --i;
  return i * i;
}
```

2.13. FELADAT. *Írjunk programot, amely meghatározza egy szám legnagyobb valódi osztóját!*

```
#include <stdio.h>

main()
{
  unsigned szam, i;
  printf( "Szám: " );  scanf( "%u", &szam );
```

```
  for ( i = szam / 2; i >= 2 && szam % i != 0; --i )
    ;
  if ( i <= 1 )
    puts( "A számnak nincs valódi osztója." );
  else
    printf( "%u legnagyobb valódi osztója: %u.\n", szam, i );
}
```

2.14. FELADAT. *Írjunk programot, amely kiszámítja egy 0 és 12 közötti egész szám faktoriálisát! (Azért csak ekkoráét, mert a 12 faktoriálisa még tárolható egy* `unsigned long` *típusban.)*

```
#include <stdio.h>

main()
{
  unsigned szam, i;
  unsigned long fakt = 1;
  printf( "Szám: " );  scanf( "%u", &szam );
  for ( i = 2; i <= szam; ++i )
    fakt *= i;
  printf( "%u faktoriálisa: %lu.\n", szam, fakt );
}
```

2.15. FELADAT. *Írjunk programot, amely kiszámítja a jól ismert Fibonacci-sorozat n-edik elemének értékét, ahol n egy nem túl nagy természetes szám!*

1.

```
#include <stdio.h>

main()
{
  unsigned szam, i;
  long akt = 1, elozo = 1;
  printf( "Szám: " );  scanf( "%u", &szam );
  for ( i = 3; i <= szam; ++i )
  {
    long uj = elozo + akt;
    elozo = akt;
    akt = uj;
  }
  printf( "A Fibonacci-sorozat %u. eleme: %ld.\n", szam, akt );
}
```

2.

```
#include <stdio.h>

main()
{
  unsigned szam, i;
  long akt = 1, elozo = 1;
  printf( "Szám: " );  scanf( "%u", &szam );
  for ( i = 3; i <= szam; ++i )
  {
    akt += elozo;
    elozo = akt - elozo;
  }
  printf( "A Fibonacci-sorozat %u. eleme: %ld.\n", szam, akt );
}
```

2.16. FELADAT. *Írjunk programot, amely egész számokat olvas be a billentyűzetről mindaddig, míg 0-t nem gépelünk, és közben minden beolvasott számról eldönti, hogy páros-e vagy páratlan!*

```
#include <stdio.h>

main()
{
  int szam;
  puts( "Kérem a számokat:" );
  scanf( "%d", &szam );
  while ( szam )
  {
    if ( szam % 2 == 0 )
      printf( "%d páros.\n", szam );
    else
      printf( "%d páratlan.\n", szam );
    scanf( "%d", &szam );
  }
}
```

2.17. FELADAT. *Írjunk programot, amely meghatározza két pozitív egész szám legnagyobb közös osztóját!*

1.

```
#include <stdio.h>

main()
{
  unsigned a, b;
  printf( "a = " );  scanf( "%u", &a );
  printf( "b = " );  scanf( "%u", &b );
  while ( a != b )
    if ( a > b )
      a -= b;
    else
      b -= a;
  printf( "A legnagyobb közös osztó: %u.\n", a );
}
```

2.

```
#include <stdio.h>

main()
{
  unsigned a, b, r;
  printf( "a = " );  scanf( "%u", &a );
  printf( "b = " );  scanf( "%u", &b );
  while ( r = a % b )
  {
    a = b;
    b = r;
  }
  printf( "A legnagyobb közös osztó: %u.\n", b );
}
```

2.18. FELADAT. *Írjunk programot, amely egy billentyűzetről beolvasott természetes számról eldönti, hogy prímszám-e!*

```
#include <math.h>
```

```
#include <stdio.h>

main()
{
  int szam, oszto;
  printf( "Szám: " );  scanf( "%d", &szam );
  for ( oszto = 2; oszto <= sqrt( szam ) && szam % oszto != 0; ++oszto )
    ;
  if ( oszto <= sqrt( szam ) || szam == 1 )
    puts( "Nem prím." );
  else
    puts( "Prím." );
}
```

2.19. FELADAT. *Írjunk programot, amely megadja egy billentyűzetről beolvasott természetes szám prímtényezős felbontását!*

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

main()
{
  int szam;
  printf( "Szám: " );  scanf( "%d", &szam );
  if ( szam == 1 )
    puts( "1" );
  else
  {
    while ( szam != 1 )
    {
      int oszto = 2;
      for ( ; oszto <= sqrt( szam ) && szam % oszto != 0; ++oszto )
        ;
      if ( oszto > sqrt( szam ) )
        oszto = szam;
      printf( "%d ", oszto );
      szam /= oszto;
    }
    putchar( '\n' );
  }
}
```

2.20. FELADAT. *Írjunk programot, amely a billentyűzetről karaktereket olvasmindaddig, amíg azok az angol ábécé betűi, majd a beolvasás után kiírja, hogy hány volt ezek közül kisbetű!*

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

main()
{
  int szamlalo = 0;
  char ch;
  puts( "Kérem a karaktereket:" );
  do
    if ( islower( ch = getchar() ) )
      ++szamlalo;
  while ( isalpha( ch ) );
  printf( "A kisbetűk száma: %d.\n", szamlalo );
}
```

2.21. FELADAT. *Írjunk programot, amely a billentyűzetről beolvas egy szabályos dátumot (év,*

hónap, nap sorrendben), majd meghatározza és képernyőre írja, hogy az adott nap hányadik napja az adott évnek!

```
#include <stdio.h>

int napszam( int ev, int ho, int nap )
{
  int szokoev = ev % 4 == 0 && ev % 100 != 0 || ev % 400 == 0;
  int ho_nap[ 12 ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  int i, osszeg = 0;
  for ( i = 0; i < ho - 1; ++i )
    osszeg += ho_nap[ i ];
  return osszeg + nap + ( ho > 2 && szokoev ? 1 : 0 );
}

main()
{
  int ev, ho, nap;
  printf( "Kérek egy dátumot: " );
  scanf( "%d %d %d", &ev, &ho, &nap );
  printf( "Ez az év %d. napja.\n", napszam( ev, ho, nap ) );
}
```

2.22. FELADAT. *Írjunk programot, amely egy billentyűzetről beolvasott karaktersorozatban megszámolja, hogy hány betűt, hány számjegyet és hány egyéb karaktert tartalmaz!*

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

#define MERET 1000

main()
{
  char s[ MERET ];
  int i, betu = 0, szam = 0, egyeb = 0;
  printf( "Kérem a sztringet: ");  fgets( s, MERET, stdin );
  for ( i = 0; i < strlen( s ); ++i )
  {
    if ( isalpha( s[ i ] ) )
      ++betu;
    else if ( isdigit( s[ i ] ) )
      ++szam;
    else
      ++egyeb;
  }
  printf( "betű: %d   szám: %d   egyéb: %d\n", betu, szam, egyeb );
}
```

2.23. FELADAT. *Írjunk programot, amely egyesével beolvas egész számokat mindaddig, amíg a számok váltakozó előjelűek, majd kiírja a képernyőre a beolvasott értékek darabszámát! A nulla egy speciális érték, amelyet a negatív számok közé sorolunk, ha előtte pozitív érték szerepel (beleértve a pozitív számok közé sorolt nullát is), illetve a pozitívak közé, ha előtte negatív érték áll (beleértve a negatív számok közé sorolt nullát is). A sorozat elején álló nulla értékek előjele a sorozat első nem nulla értékének előjelétől függ.*

```
#include <stdio.h>

main()
{
  enum { SEMLEGES, POZITIV, NEGATIV, VEG } allapot = SEMLEGES;
```

```
  int szam, darab = 0;

  do
  {
    ++darab;
    scanf( "%d", &szam );
    switch ( allapot )
    {
      case SEMLEGES:
        if ( szam < 0 )
          allapot = NEGATIV;
        else if ( szam > 0 )
          allapot = POZITIV;
        break;
      case POZITIV:
        allapot = szam > 0 ? VEG : NEGATIV;
        break;
      case NEGATIV:
        allapot = szam < 0 ? VEG : POZITIV;
        break;
    }
  } while ( allapot != VEG );
  printf( "A beolvasott értékek száma: %d.\n", darab );
}
```

2.24. FELADAT. *Írjunk programot, amely az ötöslottó számsorsolását modellezi!*

A feladatot a könyvtári véletlenszám-generáló függvényekkel (`srand()` és `rand()`) kétféleképpen is megoldjuk.

1. Először egy ötelemű tömböt használunk, amely a már kihúzott számokat fogja tartalmazni. Minden új szám előállításakor megnézzük, hogy szerepel-e már a tömbben. Ha nem, megjegyezzük, ha igen, újat generálunk.

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

#define DARAB 5

main()
{
  int kihuzott[ DARAB ], db, i;
  srand( ( unsigned int )time( NULL ) );
  for ( db = 0; db < DARAB; ++db )
    do
    {
      kihuzott[ db ] = 1 + ( int )( 90.0*rand() / ( RAND_MAX+1.0 ) );
      for ( i = 0; i < db; ++i )
        if ( kihuzott[ i ] == kihuzott[ db ] )
          break;
    } while ( i < db );
  for ( i = 0; i < DARAB; ++i )
    printf( "%d ", kihuzott[ i ] );
}
```

2. A második megoldásban egy 90 elemű tömböt használunk. A tömb elemei a kihúzható számokat jelentik, és a 90-ből az első 5 lesz az, amit kihúzottnak tekintünk. Így nincs más teendőnk, mint az első 5 számot sorban kicserélni a maradék számok közül véletlenszerűen választottakkal.

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

```
#include <time.h>

#define MAXDB 90
#define DARAB 5

main()
{
  int szamok[ MAXDB ], i;
  srand( ( unsigned int )time( NULL ) );
  for ( i = 0; i < MAXDB; ++i )
    szamok[ i ] = i + 1;
  for ( i = 0; i < DARAB; ++i )
  {
    int index =
      i + ( int )( ( double )( MAXDB-i )*rand() / ( RAND_MAX+1.0 ) );
    int seged = szamok[ index ];
    szamok[ index ] = szamok[ i ];
    szamok[ i ] = seged;
    printf( "%d ", seged );
  }
}
```

3. fejezet - Származtatott adattípusok

Tartalom

Tömbök

3.1. FELADAT. *Írjunk programot, amely a billentyűzetről látható karaktereket olvas mindaddig, amíg a @ karaktert meg nem kapja! A program határozza meg és írja képernyőre a beolvasott különböző karaktereket és azok gyakoriságát!*

```
#include <stdio.h>

main()
{
  int c;
  int gyak[ 256 ] = { 0 };   /* az egész tömböt nullázza */
  while ( ( c = getchar() ) != '@' )
    ++gyak[ c ];
  for ( c = 0; c < 256; ++c )
    if ( gyak[ c ] )
      printf( "%c: %d\n", c, gyak[ c ] );
}
```

3.2. FELADAT. *Írjunk eljárást, amely paraméterként megkap egy tetszőleges méretű, egészeket tartalmazó egydimenziós tömböt! Az eljárás határozza meg a tömbben lévő pozitív, negatív és nulla elemek darabszámát! Az eljárás nem írhat a képernyőre!*

A feladat többféleképpen is megoldható.

1. Nézzük először azt a megoldást, amikor az eljárás globális változókban határozza meg a pozitív, negatív és nulla elemek darabszámát:

```
int pozitiv, negativ, nulla;
```

```
void poznegnull( int *t, int meret )
{
  int i;
  pozitiv = negativ = nulla = 0;
  for ( i = 0; i < meret; ++i )
    if ( t[ i ] > 0 )
      ++pozitiv;
    else if ( t[ i ] < 0 )
      ++negativ;
    else
      ++nulla;
}
```

2. Egy másik megoldási lehetőség, ha az eljárásnak átadunk még három paramétert, azoknak a memóriaterületeknek a címét, ahol a keresett értékeket tárolni szeretnénk:

```
void poznegnull( int *t, int meret, int *poz, int *neg, int *nulla )
{
  int i;
  *poz = *neg = *nulla = 0;
  for ( i = 0; i < meret; ++i )
    if ( t[ i ] > 0 )
      ++*poz;
    else if ( t[ i ] < 0 )
      ++*neg;
    else
      ++*nulla;
}
```

3.3. FELADAT. *Írjunk logikai függvényt, amely a paraméterként megkapott, egész számokat tartalmazó tömbről eldönti, hogy van-e az elemei között két olyan, amelyeknek a szorzata megegyezik egy szintén paraméterként kapott számmal!*

```
#define HAMIS 0
#define IGAZ  ( !HAMIS )

int van( int t[], int meret, int szam )
{
  int i, j;
  for ( i = 0; i < meret - 1; ++i )
    for ( j = i + 1; j < meret; ++j )
      if ( t[ i ] * t[ j ] == szam )
        return IGAZ;
  return HAMIS;
}
```

3.4. FELADAT. *Írjunk programot, amely a billentyűzetről egész értékeket olvas be a 0 végjelig! A program írja képernyőre azokat az értékeket, amelyek megyegyeznek az előző két érték összegével!*

```
#include <stdio.h>

#define HAMIS 0
#define IGAZ  ( !HAMIS )
#define DARAB 3

main()
{
  int t[ DARAB ], idx = -1, megvan = HAMIS;
  for ( ; ; )
```

```
  {
    scanf( "%d", &t[ idx = ( idx + 1 ) % DARAB ] );
    if ( !t[ idx ] )
      break;
    if ( !megvan && idx == DARAB - 1 )
      megvan = IGAZ;
    if ( megvan &&
         t[ idx ] == t[ ( idx+1 ) % DARAB ] + t[ ( idx+2 ) % DARAB ] )
      printf( "%d\n", t[ idx ] );
  }
}
```

3.5. FELADAT. *Írjunk programot, amely nullától különböző egész értékeket olvas be a billentyűzetről a 0 végjelig! A program határozza meg és írja képernyőre azt a három értéket, amelynek átlaga maximális!*

Vegyük észre, hogy a feladat nem más, mint a begépelt számok közül a három legnagyobb értékű szám kiválasztása. A feladat nem szól arról, hogy mi történjen, ha háromnál kevesebb értéket olvasunk be, így a következő programok ilyen esetekben csak egy figyelmeztető üzenetet írnak a képernyőre.

1. Az első megoldás tárolja az összes beolvasott értéket, majd a 0 érték beolvasása után csökkenő sorrendbe rendezi őket, és kiírja közülük az első hármat (ha van legalább három érték).

```
#include <stdio.h>

void csokkeno_rendez( int t[], int meret )
{
  int i, j;
  for ( i = meret - 2; i >= 0; --i )
    for ( j = 0; j <= i; ++j )
      if ( t[ j ] < t[ j + 1 ] )
      {
        int seged = t[ j ];
        t[ j ] = t[ j + 1 ];
        t[ j + 1 ] = seged;
      }
}

main()
{
  int *tomb = NULL, meret = 0;
  for ( ; ; )
  {
    tomb = ( int * )realloc( tomb, ( meret + 1 ) * sizeof( int ) );
    scanf( "%d", &tomb[ meret ] );
    if ( tomb[ meret ] == 0 )
      break;
    ++meret;
  }
  if ( meret < 3 )
    puts( "Nincs három érték." );
  else
  {
    csokkeno_rendez( tomb, meret );
    printf( "%d, %d, %d\n", tomb[ 0 ], tomb[ 1 ], tomb[ 2 ] );
  }
  free( tomb );
}
```

2. A második megoldás csak a három legnagyobb értéket tárolja úgy, hogy mindig a legkisebb tárolt

értéket írja felül, ha egy nála nagyobb értéket olvas.

```
#include <stdio.h>

#define DARAB 3

main()
{
  int tomb[ DARAB ], meret = 0, szam;
  for ( ; ; )
  {
    scanf( "%d", &szam );
    if ( szam == 0 )
      break;
    if ( meret < DARAB )
      tomb[ meret++ ] = szam;
    else
    {
      int index = tomb[ 0 ] < tomb[ 1 ] ? tomb[ 0 ] < tomb[ 2 ] ?
        0 : 2 : tomb[ 1 ] < tomb[ 2 ] ? 1 : 2;
      if ( szam > tomb[ index ] )
        tomb[ index ] = szam;
    }
  }
  if ( meret < DARAB )
    puts( "Nincs három érték." );
  else
    printf( "%d, %d, %d\n", tomb[ 0 ], tomb[ 1 ], tomb[ 2 ] );
}
```

3. A harmadik megoldás, a második megoldáshoz hasonlóan, a három legnagyobb értéket tárolja, amelyek sorrendje azonban a beolvasás szerinti sorrend marad.

```
#include <stdio.h>

#define DARAB 3

main()
{
  int tomb[ DARAB ], meret = 0, szam;
  scanf( "%d", &szam );
  while ( szam )
  {
    if ( meret < DARAB )
      tomb[ meret++ ] = szam;
    else
    {
      int index = 0, i;
      for ( i = 1; i < DARAB; ++i )
        if ( tomb[ i ] < tomb[ index ] )
          index = i;
      if ( tomb[ index ] < szam )
      {
        while ( index < DARAB - 1 )
        {
          tomb[ index ] = tomb[ index + 1 ];
          ++index;
        }
        tomb[ DARAB - 1 ] = szam;
      }
    }
    scanf( "%d", &szam );
```

```
  }
  if ( meret < DARAB )
    printf( "Nem volt %d érték\n", DARAB );
  else
  {
    int i;
    printf( "%d", tomb[ 0 ] );
    for ( i = 1; i < DARAB; ++i )
      printf( ", %d", tomb[ i ] );
    putchar( '\n' );
  }
}
```

3.6. FELADAT. *Írjunk eljárást, amely megcseréli a paraméterként kapott két egész típusú értéket!*

```
void cserel( int *a, int *b )
{
  int seged = *a;
  *a = *b;
  *b = seged;
}
```

3.7. FELADAT. *Írjunk függvényt, amely egy paraméterként adott* `cel` *értékre (*$2 \le cel \le 10000000$*) meghatározza azt a legnagyobb* [illegible] *értéket, amelyre*

$F_{n-1} \cdot x + F_n \cdot y = cel,$

ahol [illegible] az [illegible] -edik Fibonacci-szám ($F_0 = 1$, $F_1 = 1$ és $F_n = F_{n-1} + F_{n-2}$, ha [illegible] > 1), továbbá [illegible] és [illegible] nemnegatív egészek!

```
int legnagyobb( unsigned long cel )
{
  unsigned long fib[ 36 ];   /* fib[ 35 ] = 14930352 */
  int n;
  fib[ 0 ] = fib[ 1 ] = 1;
  n = 2;
  do
    fib[ n ] = fib[ n - 1 ] + fib[ n - 2 ];
  while ( fib[ n++ ] <= 10000000 );
  while ( --n )
  {
    unsigned long y;
    for ( y = 0; fib[ n ] * y <= cel &&
                 ( cel - fib[ n ] * y ) % fib[ n - 1 ] != 0; ++y )
      ;
    if ( fib[ n ] * y <= cel )
      return n;
  }
}
```

3.8. FELADAT. *Írjunk programot, amely megoldja a jól ismert ,,nyolc királynő" problémát, azaz elhelyez egy sakktáblán nyolc királynőt úgy, hogy azok ne üssék egymást, és kiírja a képernyőre a probléma összes megoldását!*

1. Elsőként egy iteratív visszalépéses keresést megvalósító program kódját mutatjuk be. Érdekességképpen jegyezzük meg, hogy a program első részét (a főprogram kivételével) a következő megoldásban is felhasználjuk majd.

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

#define N 8
#define HAMIS 0
#define IGAZ  ( !HAMIS )

int tabla[ N ];   /* automatikus nullázás van */

int elofeltetel( int sor, int oszlop )
{
  int i;
  for ( i = 0; i < oszlop; ++i )
    if ( tabla[i] == sor || abs( tabla[i]-sor ) == abs( i-oszlop ) )
      return HAMIS;
  return IGAZ;
}

main()
{
  int index = 0;
  while ( index >= 0 )
  {
    if ( index == N )
    {
      int i;
      for ( i = 0; i < N; ++i )
        if ( i )
          printf( " %c%d", 'a' + i, tabla[ i ] + 1 );
        else
          printf( "%c%d", 'a' + i, tabla[ i ] + 1 );
      putchar( '\n' );
      ++tabla[ --index ];
    }
    while ( tabla[index] < N && !elofeltetel( tabla[index], index ) )
      ++tabla[ index ];
    if ( tabla[ index ] == N )
    {
      tabla[ index ] = 0;
      ++tabla[ --index ];
    }
    else
      ++index;
  }
}
```

2. Lássuk most -- felhasználva az előző megoldás első részét -- a feladat rekurzív megoldását:

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

#define N 8
#define HAMIS 0
#define IGAZ  ( !HAMIS )

int tabla[ N ];

int elofeltetel( int sor, int oszlop )
```

```
{
  int i;
  for ( i = 0; i < oszlop; ++i )
    if ( tabla[i] == sor || abs( tabla[i]-sor ) == abs( i-oszlop ) )
      return HAMIS;
  return IGAZ;
}

void kiralyno( int darab )
{
  int i;
  if ( darab == 0 )
  {
    for ( i = 0; i < N; ++i )
      if ( i )
        printf( " %c%d", 'a' + i, tabla[ i ] + 1 );
      else
        printf( "%c%d", 'a' + i, tabla[ i ] + 1 );
    putchar( '\n' );
  }
  else
  {
    for ( i = 0; i < N; ++i )
      if ( elofeltetel( i, N - darab ) )
      {
        tabla[ N - darab ] = i;
        kiralyno( darab - 1 );
      }
  }
}

main()
{
  kiralyno( N );
}
```

A tömb és a mutató

3.9. FELADAT. *Írjunk logikai függvényt, amely egy paraméterként megkapott,sztringeket tartalmazó négyzetes mátrixról eldönti, hogy szimmetrikus-e!*

```
#include <string.h>

#define HAMIS 0
#define IGAZ  ( !HAMIS )

int negyzetes( char *s[], int meret )
{
  int i, j;
  for ( i = 0; i < meret - 1; ++i )
    for ( j = i + 1; j < meret; ++j )
      if ( strcmp( s[ i * meret + j ], s[ j * meret + i ] ) )
        return HAMIS;
  return IGAZ;
}
```

3.10. FELADAT. *Írjunk eljárást, amely egy paraméterként megkapott, tetszőleges méretű, egészeket tartalmazó négyzetes mátrixot tükröz a mellékátlójára!*

```
void tukroz( int *t, int meret )
```

```
{
  int i, j;
  for ( i = 0; i < meret - 1; ++i )
    for ( j = 0; j < meret - 1 - i; ++j )
    {
      int seged = t[ i * meret + j ];
      t[ i * meret + j ] = t[ ( meret-j-1 ) * meret + ( meret-i-1 ) ];
      t[ ( meret-j-1 ) * meret + ( meret-i-1 ) ] = seged;
    }
}
```

3.11. FELADAT. *Írjunk eljárást, amely paraméterként megkap egy bitmátrixot és egy bitvektort, majd a bitmátrix minden oszlopa és a bitvektor között kizáró vagy műveletet végez! Az eredeti bitmátrixra a továbbiakban nincs szükség.*

A feladat megoldásában bitmátrix és bitvektor alatt -- az egyszerűség kedvéért -- olyan mátrixot és vektort értünk, amely csak 0 és 1 értékeket tartalmazhat. Az, hogy az eredeti bitmátrixra nincs szükség, azt jelenti, hogy a benne tárolt értékek felülírhatók.

```
void xor( int *m, int *v, int sor, int oszlop )
{
  int i, j;
  for ( j = 0; j < oszlop; ++j )
    for ( i = 0; i < sor; ++i )
      m[ i * oszlop + j ] ^= v[ i ];
}
```

3.12. FELADAT. *Írjunk függvényt, amely tetszőleges méretű, valós értékeket tartalmazó kétdimenziós tömböt kap paraméterként! A függvény határozza meg azon oszlop indexét, amelyben van olyan elem, amelynek az értéke megegyezik az oszlop elemeinek átlagával! Ha több ilyen oszlop is van, akkor a legnagyobb indexértéket adja vissza!*

Íme egy helyesnek tűnő megoldás:

```
int atlagindex( double *t, int sor, int oszlop )
{
  int j;
  for ( j = oszlop - 1; j >= 0; --j )
  {
    double atlag = 0;
    int i;
    for ( i = 0; i < sor; ++i )
      atlag += t[ i * oszlop + j ];
    atlag /= sor;
    for ( i = 0; i < sor; ++i )
      if ( t[ i * oszlop + j ] == atlag )
        return j;
  }
  return -1;
}
```

Megjegyezzük, hogy a `t[ i * oszlop + j ] == atlag` kifejezés értéke az oszlopátlag lebegőpontos ábrázolásának pontatlansága miatt igen gyakran akkor is hamis, ha matematikailag megegyezik vele a tömbelem értéke. Ezért a következő megoldást javasoljuk:

```
#include <float.h>  /* DBL_EPSILON */
#include <math.h>

int atlagindex( double *t, int sor, int oszlop )
{
  int j;
```

```
  for ( j = oszlop - 1; j >= 0; --j )
  {
    double atlag = 0;
    int i;
    for ( i = 0; i < sor; ++i )
      atlag += t[ i * oszlop + j ];
    atlag /= sor;
    for ( i = 0; i < sor; ++i )
      if ( fabs( t[ i * oszlop + j ] - atlag ) < DBL_EPSILON )
        return j;
  }
  return -1;
}
```

A fentiek illusztrálására íme egy példaprogram, amellyel tesztelni lehet mindkét változatot:

```
main()
{
  double tomb[ 3 ][ 4 ] = { {  2.5,  0.0,  5.5, 4.9 },
                            { -4.2,  1.0,  0.2, 2.6 },
                            {  2.7, -1.0, -5.1, 0.4 } };
  printf( "A legnagyobb index: %d\n",
          atlagindex( ( double * )tomb, 3, 4 ) );
}
```

Ez a program az elsőként megadott `atlagindex()` függvénnyel 1-et, a másodikként megadottal 2-t ír eredményképpen a képernyőre.

3.13. FELADAT. *Írjunk függvényt, amely egy paraméterként kapott, egészeket tartalmazó kétdimenziós tömb azon oszlopának indexét adja vissza, amelyben a legkevesebb pozitív elem van!*

Amennyiben több oszlopban is annyi pozitív elem van, mint abban, amelyikben a legkevesebb pozitív elem található, akkor az alábbi függvény a legelső ilyen oszlop indexét határozza meg.

```
int kevespozoszlop( int *t, int sor, int oszlop )
{
  int j, min = sor, minoszlop = 0;
  for ( j = 0; j < oszlop; ++j )
  {
    int i, poz = 0;
    for ( i = 0; i < sor; ++i )
      if ( t[ i * oszlop + j ] > 0 )
        ++poz;
    if ( poz < min )
    {
      min = poz;
      minoszlop = j;
    }
  }
  return minoszlop;
}
```

3.14. FELADAT. *Írjunk függvényt, amely egy paraméterként megkapott, egészeket tartalmazó kétdimenziós tömb esetén megadja azon oszlopok számát, amelyekben egy szintén paraméterként megadott értéknél csak nagyobb értékek szerepelnek!*

```
int csaknagyobb( int *t, int sor, int oszlop, int ertek )
{
  int j, db = 0;
  for ( j = 0; j < oszlop; ++j )
  {
    int i;
```

```
    for ( i = 0; i < sor; ++i )
      if ( t[ i * oszlop + j ] <= ertek )
        break;
    if ( i == sor )
      ++db;
  }
  return db;
}
```

3.15. FELADAT. *Írjunk eljárást, amely paraméterként megkap egy tetszőleges méretű, valósakat tartalmazó kétdimenziós tömböt, és előállít egy olyan egydimenziós tömböt, amely a sorok átlagát tartalmazza! Az eljárás a képernyőre nem írhat!*

```
#include <stdlib.h>

double *soratl;

void soratlagok( double *t, int sor, int oszlop )
{
  int i;
  soratl = ( double * )calloc( sor, sizeof( double ) );
  for ( i = 0; i < sor; ++i )
  {
    int j;
    for ( j = 0; j < oszlop; ++j )
      soratl[ i ] += t[ i * oszlop + j ];
    soratl[ i ] /= oszlop;
  }
}
```

3.16. FELADAT. *Írjunk eljárást, amely egy paraméterként megkapott, egészeket tartalmazó kétdimenziós tömb oszlopait úgy rendezi át, hogy az első sor elemei nagyság szerint csökkenő sorrendben legyenek! Feltehetjük, hogy az első sor elemei különbözőek.*

A feladat többféle módon is megoldható, itt buborékrendezéssel rendeztük a tömb első sorának elemeit:

```
void csokelsosor( int *t, int sor, int oszlop )
{
  int korlat = oszlop - 1, utolsocsere;
  do
  {
    int j;
    utolsocsere = -1;
    for ( j = 0; j < korlat; ++j )
      if ( t[ j ] < t[ j + 1 ] )
      {
        int i;
        /* az oszlopok minden elemét cseréljük */
        for ( i = 0; i < sor; ++i )
        {
          int seged = t[ i * oszlop + j ];
          t[ i * oszlop + j ] = t[ i * oszlop + j + 1 ];
          t[ i * oszlop + j + 1 ] = seged;
        }
        utolsocsere = j;
      }
    korlat = utolsocsere;
  } while ( utolsocsere != -1 );
}
```

3.17. FELADAT. *Írjunk eljárást, amely egy paraméterként megkapott, tetszőleges méretű, valósakat tartalmazó kétdimenziós tömb sorait úgy rendezi át, hogy az utolsó oszlop értékei csökkenő sorrendben legyenek!*

A feladat többféle módon is megoldható, itt maximumkiválasztásos rendezéssel rendeztük a tömb utolsó oszlopának elemeit:

```
void atrendez( double *t, int sor, int oszlop )
{
  int i;
  for ( i = 0; i < sor - 1; ++i )
  {
    int j, k, index = i;
    for ( k = i + 1; k < sor; ++k )
      if ( t[ index * oszlop + oszlop-1 ] < t[ k * oszlop + oszlop-1 ] )
        index = k;
    /* a sorok minden elemét cseréljük */
    for ( j = 0; j < oszlop; ++j )
    {
      double seged = t[ index * oszlop + j ];
      t[ index * oszlop + j ] = t[ i * oszlop + j ];
      t[ i * oszlop + j ] = seged;
    }
  }
}
```

3.18. FELADAT. *Írjunk eljárást, amely egy paraméterként megkapott, tetszőleges méretű, egészeket tartalmazó kétdimenziós tömb oszlopátlagai közül meghatározza a legnagyobbat (több ilyen is lehet)! Az eljárás nem írhat képernyőre és állományba!*

Vegyük észre, hogy a feladat megfogalmazása csalafinta: hiába van esetleg több azonos legnagyobb oszlopátlag, az eljárásnak csak ezek egyikét, egyetlen értéket kell meghatároznia.

A feladat többféleképpen is megoldható.

1. Lássuk először azt a megoldást, amikor az eljárás egy globális változóban határozza meg a keresett (legnagyobb) oszlopátlagot:

```
double atlag;

void atlagol( int *t, int sor, int oszlop )
{
  int i, j;
  atlag = 0.0;
  for ( i = 0; i < sor; ++i )
    atlag += t[ i * oszlop ];
  for ( j = 1; j < oszlop; ++j )
  {
    double seged = 0.0;
    for ( i = 0; i < sor; ++i )
      seged += t[ i * oszlop + j ];
    if ( seged > atlag )
      atlag = seged;
  }
  atlag /= sor;
}
```

2. Egy másik megoldási lehetőség, ha az eljárásnak átadunk még egy paramétert, annak a memóriaterületnek a címét, ahol a keresett átlagértéket tárolni szeretnénk:

```
void atlagol( int *t, int sor, int oszlop, double *atlag )
{
```

```
  int i, j;
  *atlag = 0.0;
  for ( i = 0; i < sor; ++i )
    *atlag += t[ i * oszlop ];
  for ( j = 1; j < oszlop; ++j )
  {
    double seged = 0.0;
    for ( i = 0; i < sor; ++i )
      seged += t[ i * oszlop + j ];
    if ( seged > *atlag )
      *atlag = seged;
  }
  *atlag /= sor;
}
```

3.19. FELADAT. *Írjunk függvényt, amely paraméterként megkap egy tetszőleges méretű, egészeket tartalmazó négyzetes mátrixot, és visszaadja a főátló maximális és minimális elemét!*

```
typedef struct { int max, min; } MAXMIN;

MAXMIN foatlo( int *m, int meret )
{
  MAXMIN mm = { *m, *m };
  int i;
  for ( i = 1; i < meret; ++i )
    if ( m[ i * ( meret + 1 ) ] > mm.max )
      mm.max = m[ i * ( meret + 1 ) ];
    else if ( m[ i * ( meret + 1 ) ] < mm.min )
      mm.min = m[ i * ( meret + 1 ) ];
  return mm;
}
```

3.20. FELADAT. *Írjunk függvényt, amely paraméterként megkap egy tetszőleges méretű, valósakat tartalmazó kétdimenziós tömböt, és visszatérési értékként meghatározza a sorok átlagának minimumát és az oszlopok átlagának maximumát!*

```
typedef struct { double min, max; } MINMAX;

MINMAX sorminoszmax( double *m, int sor, int oszlop )
{
  MINMAX mm;
  int i, j;
  for ( i = 0; i < sor; ++i )
  {
    double osszeg = 0;
    for ( j = 0; j < oszlop; ++j )
      osszeg += m[ i * oszlop + j ];
    if ( i == 0 || osszeg < mm.min )
      mm.min = osszeg;
  }
  mm.min /= oszlop;
  for ( j = 0; j < oszlop; ++j )
  {
    double osszeg = 0;
    for ( i = 0; i <sor; ++i )
      osszeg += m[ i * oszlop + j ];
    if ( j == 0 || osszeg > mm.max )
      mm.max = osszeg;
  }
  mm.max /= sor;
  return mm;
}
```

3.21. FELADAT. *Írjunk eljárást, amely egy paraméterként megkapott, egészeket tartalmazó kétdimenziós tömbben meghatározza azon oszlopok indexét (akárhány ilyen lehet), amelyekben a negatív elemek száma legalább kétszerese a nulla értékű elemek számának! A tömb mérete tetszőleges.*

A feladat többféleképpen is megoldható.

1. Először is lássuk azt a megoldást, amely két globális változót használ: egyik a feltételnek megfelelő oszlopok darabszámát fogja tartalmazni, a másik pedig arra a memóriaterületre mutat, ahol a keresett oszlopindexeket tároljuk.

```
#include <stdlib.h>

int *dupneg, darab;

void negketszer( int *t, int sor, int oszlop )
{
  int j;
  dupneg = NULL;
  darab = 0;
  for ( j = 0; j < oszlop; ++j )
  {
    int i, negativ = 0, nulla = 0;
    for ( i = 0; i < sor; ++i )
      if ( t[ i * oszlop + j ] < 0 )
        ++negativ;
      else if ( t[ i * oszlop + j ] == 0 )
        ++nulla;
    if ( negativ >= 2 * nulla )
    {
      dupneg = ( int * )realloc( dupneg, darab + 1 );
      dupneg[ darab++ ] = j;
    }
  }
}
```

2. Az eljárás természetesen paraméterlistán keresztül is kommunikálhat a hívó programegységgel. Ekkor a megoldás a következő lehet:

```
#include <stdlib.h>

void negketszer( int *t, int sor, int oszlop, int *db, int **dupneg )
{
  int j;
  *dupneg = NULL;
  *db = 0;
  for ( j = 0; j < oszlop; ++j )
  {
    int i, negativ = 0, nulla = 0;
    for ( i = 0; i < sor; ++i )
      if ( t[ i * oszlop + j ] < 0 )
        ++negativ;
      else if ( t[ i * oszlop + j ] == 0 )
        ++nulla;
    if ( negativ >= 2 * nulla )
    {
      *dupneg = ( int * )realloc( *dupneg, *db + 1 );
      ( *dupneg )[ ( *db )++ ] = j;
    }
  }
}
```

3. Álljon itt végül az a megoldás, amely egy globális mutatóval dolgozik: a mutató egy olyan memóriaterületre mutat, amelynek első eleme az ezt követő elemek (a keresett oszlopindexek) darabszámát adja meg.

```
#include <stdlib.h>

int *dupneg;

void negketszer( int *t, int sor, int oszlop )
{
  int j;
  dupneg = ( int * )malloc( sizeof( int ) );
  *dupneg = 0;
  for ( j = 0; j < oszlop; ++j )
  {
    int i, negativ = 0, nulla = 0;
    for ( i = 0; i < sor; ++i )
      if ( t[ i * oszlop + j ] < 0 )
        ++negativ;
      else if ( t[ i * oszlop + j ] == 0 )
        ++nulla;
    if ( negativ >= 2 * nulla )
    {
      dupneg = ( int * )realloc( dupneg, *dupneg + 2 );
      dupneg[ ++*dupneg ] = j;
    }
  }
}
```

3.22. FELADAT. *Írjunk eljárást, amely egy paraméterként megadott kétdimenziós, egészeket tartalmazó tömb azon oszlopát határozza meg, amelyben benne van az egész tömb legnagyobb eleme (csak egy ilyen van)!*

A feladatot többféle módon is lehet értelmezni.

1. Elsőként nézzük azt a változatot, amelyben csak a keresett oszlop indexét határozzuk meg.

```
void legoszlop( int *t, int sor, int oszlop, int *idx )
{
  int i, j, maxelem = *t;
  *idx = 0;
  for ( i = 0; i < sor; ++i )
    for ( j = 0; j < oszlop; ++j )
      if ( t[ i * oszlop + j ] > maxelem )
      {
        maxelem = t[ i * oszlop + j ];
        *idx = j;
      }
}
```

2. A második megoldásban már nemcsak egy oszlopindexet keresünk, hanem előállítunk -- a tömbétől különböző memóriaterületen -- egy olyan vektort, amely a keresett oszlop elemeit tartalmazza.

```
#include <stdlib.h>

void legoszlop( int *t, int sor, int oszlop, int **oszl )
{
  int i, j, maxelem = *t, maxoszlop = 0;
  *oszl = ( int * )malloc( sor * sizeof( int ) );
  for ( i = 0; i < sor; ++i )
    for ( j = 0; j < oszlop; ++j )
```

```
      if ( t[ i * oszlop + j ] > maxelem )
      {
        maxelem = t[ i * oszlop + j ];
        maxoszlop = j;
      }
  for ( i = 0; i < sor; ++i )
    ( *oszl )[ i ] = t[ i * oszlop + maxoszlop ];
}
```

3.23. FELADAT. *Írjunk eljárást, amely egy paraméterként megkapott, tetszőleges méretű, egészeket tartalmazó négyzetes mátrix főátlójában elhelyezi soronként a főátló fölötti elemek összegét!*

```
void atlossz( int *t, int meret )
{
  int j;
  for ( j = 0; j < meret; ++j )
  {
    int i;
    t[ j * meret + j ] = 0;
    for ( i = 0; i < j; ++i )
      t[ j * meret + j ] += t[ i * meret + j ];
  }
}
```

3.24. FELADAT. *Írjunk eljárást, amely előállít egy olyan kétdimenziós tömböt, amely egy paraméterként megkapott, egészeket tartalmazó, tetszőleges méretű kétdimenziós tömb elemeit tartalmazza, de minden olyan elemének a 0 értéket adja, amelynek a tömb elemeinek átlagától való eltérése az átlag felénél nagyobb! Az eredeti tömböt változatlanul kell hagyni!*

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

void nullaz( int *t, int sor, int oszlop, int **ujtomb )
{
  *ujtomb = ( int * )malloc( sor * oszlop * sizeof( int ) );
  int i, j;
  double atlag;
  for ( i = 0; i < sor; ++i )
    for ( j = 0; j < oszlop; ++j )
      atlag += t[ i * oszlop + j ];
  atlag /= sor * oszlop;
  for ( i = 0; i < sor; ++i )
    for ( j = 0; j < oszlop; ++j )
      if ( fabs( t[ i * oszlop + j ] - atlag ) > atlag / 2 )
        ( *ujtomb )[ i * oszlop + j ] = 0;
      else
        ( *ujtomb )[ i * oszlop + j ] = t[ i * oszlop + j ];
}
```

Sztringkezelés, könyvtári sztringkezelő függvények használata

3.25. FELADAT. *Írjunk eljárást, amely paraméterként megkap egy karaktereket tartalmazó, tetszőleges méretű egydimenziós tömböt, és a tömb nem betű karaktereit kicseréli szóközre!*

```
#include <ctype.h>

void nembetu( char t[], int meret )
{
  int i;
```

```
  for ( i = 0; i < meret; ++i )
    if ( !isalpha( t[ i ] ) )
      t[ i ] = ' ';
}
```

3.26. FELADAT. *Írjunk függvényt, amely egy paraméterként megkapott sztringben szóközzel felülírja a nem betű karaktereket és visszaadja az új sztringet!*

A feladatot többféleképpen is meg lehet oldani. Mi itt a sztring kezelésének módjától függően két megoldást mutatunk be.

1. Elsőként tömbelemekként kezeljük a sztring karaktereit, amelyekre indexeléssel hivatkozunk.

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

char *nembetu( char *s )
{
  char *uj = ( char * )malloc( ( strlen( s ) + 1 ) * sizeof( char ) );
  int i;
  for ( i = 0; s[ i ]; ++i )
    uj[ i ] = isalpha( s[ i ] ) ? s[ i ] : ' ';
  uj[ i ] = '\0';
  return uj;
}
```

2. A második megoldásban a paraméterként kapott mutató segítségével járjuk végig a sztringet, miközben egy másik mutató az új sztringen mozog.

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

char *felulir( char *s )
{
  char *uj = ( char * )malloc( ( strlen( s ) + 1 ) * sizeof( char ) );
  char *p = uj;
  while ( *p = *s )
  {
    if ( !isalpha( *p ) )
      *p = ' ';
    ++s;
    ++p;
  }
  return uj;
}
```

3.27. FELADAT. *Írjunk logikai függvényt, amely paraméterként megkap egy sztringet, és igaz értéket ad vissza, ha a sztring tükörszimmetrikus (például: `kosarasok`, `görög`)!*

```
#include <string.h>

int tukor( char *s )
{
  int i, j;
  for ( i = 0, j = strlen( s ) - 1; i < j && s[i] == s[j]; ++i, --j )
    ;
  return i >= j;
}
```

3.28. FELADAT. *Írjunk eljárást, amely a paraméterként kapott sztringet helyben megfordítja!*

```
#include <string.h>

void fordit( char *s )
{
  int also, felso;
  for ( also = 0, felso = strlen(s) - 1; also < felso; ++also, --felso )
  {
    char c = s[ also ];
    s[ also ] = s[ felso ];
    s[ felso ] = c;
  }
}
```

3.29. FELADAT. *Írjunk függvényt, amely paraméterként megkap egy sztringet, és annak tükörképét adja vissza új sztringként!*

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

char *tukroz( char *s )
{
  char *uj = ( char * )malloc( ( strlen( s ) + 1 ) * sizeof( char ) );
  int also, felso;
  for ( also = 0, felso = strlen( s ) - 1; s[ also ]; ++also, --felso )
    uj[ also ] = s[ felso ];
  uj[ also ] = '\0';
  return uj;
}
```

3.30. FELADAT. *Írjunk logikai függvényt, amely egy paraméterként megkapott magyar szóról eldönti, hogy magánhangzóra vagy mássalhangzóra végződik-e!*

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

int maganveg( char *s )
{
  return !strchr( "bcdfghjklmnpqrstvwxyz",
                  tolower( s[ strlen( s ) - 1 ] ) );
}
```

3.31. FELADAT. *Írjunk logikai függvényt, amely paraméterként megkap egy angol nyelvű szöveget tartalmazó sztringet, és igaz értékkel tér vissza, ha a sztringben a betűk száma nagyobb, mint a nem betű karakterek száma, és hamissal tér vissza egyébként!*

1. Az elsőként bemutatott megoldásban a kapott sztringet tömbként kezeljük, a karaktereit indexeléssel érjük el és vizsgáljuk meg.

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

int tobb_betu( char *s )
{
  int i, betu = 0;

  for ( i = 0; i < strlen( s ); ++i )
    if ( isalpha( s[ i ] ) )
      ++betu;
  return betu > strlen( s ) - betu;
}
```

2. A következő megoldásban nem indexelünk, hanem a paraméterként megkapott mutató segítségével lépkedünk végig a sztring karakterein.

```
#include <ctype.h>

int tobb_betu( char *s )
{
  int hossz, betu = 0;

  for ( hossz = 0; *s; ++s, ++hossz )
    if ( isalpha( *s ) )
      ++betu;
  return betu > hossz - betu;
}
```

3.32. FELADAT. *Írjunk logikai függvényt, amely egy paraméterként megkapott angol szó esetén igaz értékkel tér vissza, ha a szóban nincs egymás mellett két mássalhangzó!*

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

#define HAMIS 0
#define IGAZ  ( !HAMIS )

int massal( char *s )
{
  int i;
  char *mgh = "aeiou";

  for ( i = 0; i + 1 < strlen( s ); ++i )
    if ( !strchr( mgh, tolower( s[ i ] ) ) &&
         !strchr( mgh, tolower( s[ i + 1 ] ) ) )
      return HAMIS;

  return IGAZ;
}
```

A `for` ciklus feltételében üres sztring esetén `i < strlen( s ) - 1` nem jó, mert ennek a kifejezésnek a bal oldalán `int`, a jobb oldalán `unsigned` típusú érték áll, ugyanis az `strlen` függvény `unsigned` típusú értékkel tér vissza. Az implicit konverzió során az `int` típusú érték `unsigned` típusúvá alakul, és emiatt nem -1 lesz a `<` operátor jobb oldali operandusának értéke.

3.33. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott két sztringről eldönti, hogy anagrammák-e! Két sztring akkor anagramma, ha ugyanazokból a karakterekből állnak, és az egyes karakterek ugyanannyiszor fordulnak elő a két sztringben.*

```
#include <string.h>

int anagramma( char *egyik, char *masik )
{
  int egykar[ 256 ] = { 0 }, maskar[ 256 ] = { 0 };
  for ( ; *egyik && *masik; ++egyik, ++masik )
  {
    ++egykar[ *egyik ];
    ++maskar[ *masik ];
  }
  return *egyik == *masik &&
         !memcmp( egykar, maskar, 256 * sizeof( int ) );
}
```

3.34. FELADAT. *Írjunk eljárást, amely a paraméterként megkapott sztring elejéről és végéről eltávolítja a szóköz karaktereket!*

```
void lenyes( char *s )
{
  int i, j, utolso = -1;
  for ( i = 0; s[ i ] && s[ i ] == ' '; ++i )
    ;
  for ( j = 0; s[ i ]; ++i, ++j )
  {
    s[ j ] = s[ i ];
    if ( s[ j ] != ' ' )
      utolso = j;
  }
  s[ utolso + 1 ] = '\0';
}
```

3.35. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott sztring elejéről és végéről eltávolítja a szóköz karaktereket, és visszaadja az új sztringet! Az eredeti sztring nem módosulhat!*

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

char *lenyes( char *s )
{
  char *uj = ( char * )malloc( ( strlen( s ) + 1 ) * sizeof( char ) );
  int i, j, utolso = -1;
  for ( i = 0; s[ i ] && s[ i ] == ' '; ++i )
    ;
  for ( j = 0; s[ i ]; ++i, ++j )
  {
    uj[ j ] = s[ i ];
    if ( uj[ j ] != ' ' )
      utolso = j;
  }
  uj[ utolso + 1 ] = '\0';
  return uj;
}
```

3.36. FELADAT. *Írjunk logikai függvényt, amely paraméterként megkap egy sztringet, és igaz értéket ad vissza, ha a sztringben van olyan 4 elemű részsztring, amely legalább háromszor ismétlődik!*

```
#include <string.h>

#define HAMIS 0
#define IGAZ  ( !HAMIS )

int negyismet( char *s )
{
  int i, j, darab;
  for ( i = 0; i + 3 < strlen( s ); ++i )
  {
    darab = 1;
    for ( j = i + 1; j + 3 < strlen( s ); ++j )
      if ( s[ i ] == s[ j ] && s[ i + 1 ] == s[ j + 1 ] &&
           s[ i + 2 ] == s[ j + 2 ] && s[ i + 3 ] == s[ j + 3 ] )
        if ( ++darab == 3 )
          return IGAZ;
  }
```

```
  return HAMIS;
}
```

3.37. FELADAT. *Írjunk függvényt, amely paraméterként megkap egy sztringet, és visszaadja azt az új sztringet, amely az eredeti sztringben leggyakrabban előforduló karaktereket tartalmazza (mindegyikből egyet)!*

```
#include <stdlib.h>

char *leggyakkar( char *s )
{
  char *uj;
  int i, max, darab, gyak[ 256 ] = { 0 };
  for ( ; *s; ++s )
    ++gyak[ *s ];
  for ( max = gyak[ 1 ], darab = max ? 1 : 0, i = 2; i < 256; ++i )
    if ( gyak[ i ] == max && max )
      ++darab;
    else if ( gyak[ i ] > max )
    {
      max = gyak[ i ];
      darab = 1;
    }
  uj = ( char * )malloc( ( darab + 1 ) * sizeof( char ) );
  for ( darab = 0, i = 1; i < 256; ++i )
    if ( gyak[ i ] == max && max )
      uj[ darab++ ] = i;
  uj[ darab ] = '\0';
  return uj;
}
```

3.38. FELADAT. *Írjunk függvényt, amely paraméterként megkap egy sztringet, és visszaadja azt az új sztringet, amely azt a karaktert tartalmazza, amelyik az eredeti sztringben az ASCII-kódtábla szerinti utolsó karakter, és annyiszor, ahányszor ez a karakter előfordul az eredeti sztringben!*

```
#include <stdlib.h>

char *utolsokar( char *s )
{
  char *uj, maxkar;
  int darab;
  for ( darab = 0, maxkar = *s; *s; ++s )
    if ( *s == maxkar )
      ++darab;
    else if ( *s > maxkar )
    {
      maxkar = *s;
      darab = 1;
    }
  uj = ( char * )malloc( ( darab + 1 ) * sizeof( char ) );
  uj[ darab ] = '\0';
  while ( darab )
    uj[ --darab ] = maxkar;
  return uj;
}
```

3.39. FELADAT. *Írjunk függvényt, amely összeadja a paraméterként megkapott sztringben található természetes számokat!*

1. Első megoldásunkban a nem számjegy karaktereket szóközökkel helyettesítjük, majd az

`strtoul()` konverziós függvénnyel átkonvertáljuk a sztring számjegy karaktereit előjel nélküli egész számokká mindaddig, amíg lehetséges.

```
unsigned long termszam( char *s )
{
  unsigned long osszeg = 0;
  int i;
  for ( i = 0; s[ i ]; ++i )
    if ( !isdigit( s[ i ] ) )
      s[ i ] = ' ';
  while ( 1 )
  {
    char *p;
    osszeg += strtoul( s, &p, 10 );
    if ( s == p )
      break;
    s = p;
  }
  return osszeg;
}
```

2. Második megoldásunkban karakterenként haladunk végig a sztringen. Ha számjegyet érintünk, akkor a Horner-séma szerint vagy egy új számot kezdünk, vagy pedig rászorzunk a számrendszer alapszámával (10-zel) az eddigi utolsó számra. Ha véget ért a számjegysorozat, akkor hozzáadjuk az összeghez a menet közben kiszámolt számot.

```
unsigned long termszam( char *s )
{
  unsigned long osszeg = 0, szam;
  enum { SEMLEGES, SZAM, VEGE } allapot = SEMLEGES;
  while ( allapot != VEGE )
  {
    switch ( allapot )
    {
      case SEMLEGES: if ( !*s )
                       allapot = VEGE;
                     else if ( isdigit( *s ) )
                     {
                       szam = *s - '0';
                       allapot = SZAM;
                     }
                     break;
      case SZAM:     if ( !*s || !isdigit( *s ) )
                     {
                       osszeg += szam;
                       allapot = *s ? SEMLEGES : VEGE;
                     }
                     else
                     {
                       szam *= 10;
                       szam += *s - '0';
                     }
                     break;
    }
    ++s;
  }
  return osszeg;
}
```

3.40. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott mondat minden második szavát megfordítja, és visszaadja az új mondatot (az eredeti mondatot pedig változatlanul hagyja)!*

Vegyük észre, hogy a feladat nem határozza meg a nyelvet, amelynek szavait (betűit) a mondat tartalmazza, ezért mi sem szorítkozhatunk csupán az angol vagy a magyar nyelvre. Áthidaló megoldásként ismertnek tekintjük a nyelv szavakat elhatároló írásjeleit (karaktereit), ezeket a későbbiekben igény szerint átdefiniálhatjuk. Így megoldásaink a következők:

1. Elsőként a kapott mondat karaktereinek minősítése (határolójel vagy sem) szerint dolgozzuk fel a sztringet. Határolójelek olvasásakor, ha páros számú szónál tartunk, elindulunk fordított sorrendben összegyűjteni a karaktereket az aktuális pozíciótól visszafelé haladva.

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

char *masod_fordit( char *s )
{
  char *hatarolok = " \t\n().!?,;:-";
  char *uj = ( char * )malloc( ( strlen( s ) + 1 ) * sizeof( char ) );
  int paros = 0, eleje = 1;
  char *p = uj;
  for ( ;; ++s )
  {
    if ( !*s || strchr( hatarolok, *s ) )
    {
      if ( paros )
      {
        char *seged = s - 1;
        while ( !strchr( hatarolok, *seged ) )
          *p++ = *seged--;
      }
      *p++ = *s;
      if ( !*s )
        break;
      if ( !eleje && !strchr( hatarolok, *( s + 1 ) ) )
        paros = !paros;
    }
    else
    {
      eleje = 0;
      if ( !paros )
        *p++ = *s;
    }
  }
  return uj;
}
```

2. Második megoldásunkban azt figyeljük, hogy páros vagy páratlan szónál tartunk-e. A páratlan szavak karaktereit (és az őket követő karaktereket a legközelebbi határolójelig) változatlan sorrendben átmásoljuk az új sztringbe, a páros szavaknál pedig az őket követő első határolójeltől visszafelé haladva tesszük ugyanezt.

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

char *masod_fordit( char *s )
{
  char *hatarolok = " \t\n().!?,;:-";
  char *uj = ( char * )malloc( ( strlen( s ) + 1 ) * sizeof( char ) );
  int paros = 0, eleje = 1;
  char c, *p = uj;
  for ( ; *s; ++s )
  {
    if ( paros )
```

```
    {
      if ( !strchr( hatarolok, *s ) &&
           ( strchr( hatarolok, *( s + 1 ) ) || !*( s + 1 ) ) )
      {
        char *seged = s;
        while ( !strchr( hatarolok, *seged ) )
          *p++ = *seged--;
      }
      else if ( strchr( hatarolok, *s ) )
      {
        *p++ = *s;
        if ( !strchr( hatarolok, *( s + 1 ) ) )
          paros = 0;
      }
    }
    else
    {
      *p++ = *s;
      if ( !eleje && strchr( hatarolok, *s ) &&
           !strchr( hatarolok, *( s + 1 ) ) )
        paros = 1;
      if ( eleje && !strchr( hatarolok, *s ) )
        eleje = 0;
    }
  }
  *p = '\0';
  return uj;
}
```

3. A harmadik megoldásban egy állapotátmenet-gráf alapján készítjük el a kódot. Az állapotok a következők:

1	semleges	éppen nem dolgozunk fel se páros, se páratlan szót
2	páratlan	egy páratlan szó karaktereit olvassuk
3	kétköz	két szó -- egy páratlan és egy páros -- között járunk
4	páros	egy páros szó karaktereit olvassuk
5	vége	elérkeztünk a mondat

		végére

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

char *masod_fordit( char *s )
{
  char *uj = ( char * )malloc( ( strlen( s ) + 1 ) * sizeof( char ) );
  char *p = uj, *hatarolok = " \t\n().!?,;:-";
  enum { SEMLEGES, PARATLAN, KETKOZ, PAROS, VEGE } allapot = SEMLEGES;
  while ( allapot != VEGE )
  {
    switch ( allapot )
    {
      case SEMLEGES: *p++ = *s;
                     if ( !*s )
                       allapot = VEGE;
                     else if ( !strchr( hatarolok, *s ) )
                       allapot = PARATLAN;
                     break;
      case PARATLAN: *p++ = *s;
                     if ( !*s )
                       allapot = VEGE;
                     else if ( strchr( hatarolok, *s ) )
                       allapot = KETKOZ;
                     break;
      case KETKOZ:   if ( !*s )
                     {
                       *p++ = *s;
                       allapot = VEGE;
                     }
                     else if ( strchr( hatarolok, *s ) )
                       *p++ = *s;
                     else
                       allapot = PAROS;
                     break;
      case PAROS:    if ( !*s || strchr( hatarolok, *s ) )
                     {
                       char *seged = s - 1;
                       while ( !strchr( hatarolok, *seged ) )
                         *p++ = *seged--;
                       allapot = ( *p++ = *s ) ? SEMLEGES : VEGE;
                     }
                     break;
    }
    ++s;
  }
  return uj;
}
```

4. Végezetül, negyedikként megmutatjuk azt a megoldást, amely az előző jelöléseket felhasználva egy veremben gyűjti össze azokat a karaktereket, amelyeknek a mondatbeli sorrendjét meg kell fordítani.

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

typedef struct veremelem {
  char ch;
  struct veremelem *kovetkezo;
} VEREMELEM;
```

```
VEREMELEM *verem;

void push( char ch )
{
  VEREMELEM *uj = ( VEREMELEM * )malloc( sizeof( VEREMELEM ) );
  uj->ch = ch;
  uj->kovetkezo = verem;
  verem = uj;
}

char pop()
{
  char ch = verem->ch;
  VEREMELEM *seged = verem;
  verem = verem->kovetkezo;
  free( seged );
  return ch;
}

int ures()
{
  return !verem;
}

char *masod_fordit( char *s )
{
  char *uj = ( char * )malloc( ( strlen( s ) + 1 ) * sizeof( char ) );
  char *p = uj, *hatarolok = " \t\n().!?,;:-";
  enum { SEMLEGES, PARATLAN, KETKOZ, PAROS, VEGE } allapot = SEMLEGES;
  while ( allapot != VEGE )
  {
    switch ( allapot )
    {
      case SEMLEGES: *p++ = *s;
                     if ( !*s )
                       allapot = VEGE;
                     else if ( !strchr( hatarolok, *s ) )
                       allapot = PARATLAN;
                     break;
      case PARATLAN: *p++ = *s;
                     if ( !*s )
                       allapot = VEGE;
                     else if ( strchr( hatarolok, *s ) )
                       allapot = KETKOZ;
                     break;
      case KETKOZ:   if ( !*s )
                     {
                       *p++ = *s;
                       allapot = VEGE;
                     }
                     else if ( strchr( hatarolok, *s ) )
                       *p++ = *s;
                     else
                     {
                       push( *s );
                       allapot = PAROS;
                     }
                     break;
      case PAROS:    if ( !*s || strchr( hatarolok, *s ) )
                     {
                       while ( !ures() )
```

```
                        *p++ = pop();
                      allapot = ( *p++ = *s ) ? SEMLEGES : VEGE;
                    }
                    else
                      push( *s );
                    break;
    }
    ++s;
  }
  return uj;
}
```

3.41. FELADAT. *Írjunk függvényt, amely meghatározza, hogy a paraméterként kapott két magyar szó közül melyik van előrébb ábécé sorrend szerint! A függvény negatív értéket adjon vissza, ha az ábécé szerint az első szó van előrébb, pozitívat, ha a második szó van előrébb, és nullát, ha a két szó megegyezik (hasonlóan a könyvtári* `strcmp()` *függvényhez)!*

Bizonyos esetekben nem lehet eldönteni a szó ábécé szerinti helyét a szó elemzése nélkül. Például az *egyezség* szóban a *zs* betűkapcsolatot nem zs betűnek kell tekinteni, de ezt csak a szóelemző írásmód figyelembevételével lehet észrevenni. A mi `strcmphun()` függvényünk viszont minden szót úgy tekint, mintha az a kiejtés szerinti írásmóddal lenne megadva.

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

char *kodol( const char *s )
{
  char *kod = ( char * )malloc( ( strlen( s ) + 1 ) * sizeof( char ) ),
       *k = kod;
  while ( *s )
  {
    switch ( *s )
    {
      case 'a': case 'A':
        *k++ = 1;
        break;
      case 'á': case 'Á':
        *k++ = 2;
        break;
      case 'b': case 'B':
        *k++ = 3;
        break;
      case 'c': case 'C':
        if ( tolower( *(s+1) ) == 's' )
        {
          *k++ = 5;
          ++s;
        }
        else if ( tolower( *(s+1) ) == 'c' && tolower( *(s+2) ) == 's' )
        {
          *k++ = 5;
          *k++ = 5;
          s += 2;
        }
        else
          *k++ = 4;
        break;
      case 'd': case 'D':
        if ( tolower( *(s+1) ) == 'z' )
        {
```

```
      *k++ = 7;
      ++s;
    }
    else if ( tolower( *(s+1) ) == 'd' && tolower( *(s+2) ) == 'z' )
    {
      *k++ = 7;
      *k++ = 7;
      s += 2;
    }
    else if ( tolower( *(s+1) ) == 'z' && tolower( *(s+2) ) == 's' )
    {
      *k++ = 8;
      s += 2;
    }
    else if ( tolower( *(s+1) ) == 'd' && tolower( *(s+2) ) == 'z'
              && tolower( *(s+3) ) == 's' )
    {
      *k++ = 8;
      *k++ = 8;
      s += 3;
    }
    else
      *k++ = 6;
    break;
  case 'e': case 'E':
    *k++ = 9;
    break;
  case 'é': case 'É':
    *k++ = 10;
    break;
  case 'f': case 'F':
    *k++ = 11;
    break;
  case 'g': case 'G':
    if ( tolower( *(s+1) ) == 'y' )
    {
      *k++ = 13;
      ++s;
    }
    else if ( tolower( *(s+1) ) == 'g' && tolower( *(s+2) ) == 'y' )
    {
      *k++ = 13;
      *k++ = 13;
      s += 2;
    }
    else
      *k++ = 12;
    break;
  case 'h': case 'H':
    *k++ = 14;
    break;
  case 'i': case 'I':
    *k++ = 15;
    break;
  case 'í': case 'Í':
    *k++ = 16;
    break;
  case 'j': case 'J':
    *k++ = 17;
    break;
  case 'k': case 'K':
    *k++ = 18;
```

```
  break;
case 'l': case 'L':
  if ( tolower( *(s+1) ) == 'y' )
  {
    *k++ = 20;
    ++s;
  }
  else if ( tolower( *(s+1) ) == 'l' && tolower( *(s+2) ) == 'y' )
  {
    *k++ = 20;
    *k++ = 20;
    s += 2;
  }
  else
    *k++ = 19;
  break;
case 'm': case 'M':
  *k++ = 21;
  break;
case 'n': case 'N':
  if ( tolower( *(s+1) ) == 'y' )
  {
    *k++ = 23;
    ++s;
  }
  else if ( tolower( *(s+1) ) == 'n' && tolower( *(s+2) ) == 'y' )
  {
    *k++ = 23;
    *k++ = 23;
    s += 2;
  }
  else
    *k++ = 22;
  break;
case 'o': case 'O':
  *k++ = 24;
  break;
case 'ó': case 'Ó':
  *k++ = 25;
  break;
case 'ö': case 'Ö':
  *k++ = 26;
  break;
case 'ő': case 'Ő':
  *k++ = 27;
  break;
case 'p': case 'P':
  *k++ = 28;
  break;
case 'q': case 'Q':
  *k++ = 29;
  break;
case 'r': case 'R':
  *k++ = 30;
  break;
case 's': case 'S':
  if ( tolower( *(s+1) ) == 'z' )
  {
    *k++ = 32;
    ++s;
  }
  else if ( tolower( *(s+1) ) == 's' && tolower( *(s+2) ) == 'z' )
```

```
  {
    *k++ = 32;
    *k++ = 32;
    s += 2;
  }
  else
    *k++ = 31;
  break;
case 't': case 'T':
  if ( tolower( *(s+1) ) == 'y' )
  {
    *k++ = 34;
    ++s;
  }
  else if ( tolower( *(s+1) ) == 't' && tolower( *(s+2) ) == 'y' )
  {
    *k++ = 34;
    *k++ = 34;
    s += 2;
  }
  else
    *k++ = 33;
  break;
case 'u': case 'U':
  *k++ = 35;
  break;
case 'ú': case 'Ú':
  *k++ = 36;
  break;
case 'ü': case 'Ü':
  *k++ = 37;
  break;
case 'ű': case 'Ű':
  *k++ = 38;
  break;
case 'v': case 'V':
  *k++ = 39;
  break;
case 'w': case 'W':
  *k++ = 40;
  break;
case 'x': case 'X':
  *k++ = 41;
  break;
case 'y': case 'Y':
  *k++ = 42;
  break;
case 'z': case 'Z':
  if ( tolower( *(s+1) ) == 's' )
  {
    *k++ = 44;
    ++s;
  }
  else if ( tolower( *(s+1) ) == 'z' && tolower( *(s+2) ) == 's' )
  {
    *k++ = 44;
    *k++ = 44;
    s += 2;
  }
  else
    *k++ = 43;
  break;
```

```
    }
    ++s;
  }
  *k = '\0';
  return kod;
}

void ekezettelenit( char *kod )
{
  while ( *kod )
  {
    switch ( *kod )
    {
      case 2:
        *kod = 1;
        break;
      case 10:
        *kod = 9;
        break;
      case 16:
        *kod = 15;
        break;
      case 25:
        *kod = 24;
        break;
      case 27:
        *kod = 26;
        break;
      case 36:
        *kod = 35;
        break;
      case 38:
        *kod = 37;
        break;
    }
    ++kod;
  }
}

int strcmphun( const char *s1, const char *s2 )
{
  char *kod1 = kodol( s1 ), *kod2 = kodol( s2 );
  int elteres = strcmp( kod1, kod2 ), ujelteres;
  if ( !elteres )
  {
    free( kod1 );
    free( kod2 );
    return 0;
  }
  ekezettelenit( kod1 );
  ekezettelenit( kod2 );
  ujelteres = strcmp( kod1, kod2 );
  free( kod1 );
  free( kod2 );
  return !ujelteres ? elteres : ujelteres;
}
```

3.42. FELADAT. *Írjunk programot, amely a billentyűzetről angol szavakat olvas be a végjelig, majd képernyőre írja azokat a szavakat, amelyekben a magánhangzók száma több, mint a mássalhangzók száma!*

Mivel a feladat szerint először be kell olvasni az összes szót a végjelig, és csak utána szabad kiírni a megadott feltételnek eleget tevőket, az egyszerűség kedvéért a programot rekurzívan írjuk meg. Így ugyanis a szavak feldolgozása csak az utolsó szó beolvasása után kezdődik el. A rekurzív megoldás hatására azonban a kiírt szavak sorrendje fordítottja lesz a beolvasottakénak. A program érdekessége továbbá, hogy a rekurziót nem külön függvény, hanem maga a `main()` valósítja meg.

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

main()
{
  char szo[ 80 ];
  printf( "Kérem a szót: " );  scanf( "%s", szo );
  if ( strcmp( szo, "*" ) )
  {
    int mgh = 0, msh = 0;
    char *p = szo;
    main();
    for ( ; *p; ++p )
      if ( strchr( "aeiou", tolower( *p ) ) )
        ++mgh;
      else if ( strchr( "bcdfghjklmnpqrstvwxyz", tolower( *p ) ) )
        ++msh;
    if ( mgh > msh )
      puts( szo );
  }
}
```

4. fejezet - A C nyelv további eszközei

Tartalom

Állománykezelés

`4.1. FELADAT.` *Írjunk programot, amely angol szavakat kér be billentyűzetről *** végjelig, és kiírja egy szöveges állományba közülük azokat, amelyek tartalmazzák a b, c, x, y karaktereket!*

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

main()
{
  FILE *f = fopen( "ki.txt", "w" );
  char szo[ 100 ];
  scanf( "%s", szo );
  while ( strcmp( szo, "***" ) )
  {
    if ( strchr( szo, 'b' ) && strchr( szo, 'c' ) &&
```

```
        strchr( szo, 'x' ) && strchr( szo, 'y' ) )
      fprintf( f, "%s\n", szo );
    scanf( "%s", szo );
  }
  fclose( f );
}
```

4.2. FELADAT. *Írjunk programot, amely a billentyűzetről angol szavakat olvas mindaddig, amíg üres sztringet nem kap! A program írja egy szöveges állományba azokat a szavakat, amelyekben egymás mellett van legalább három mássalhangzó!*

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

#define angmsh( c ) ( strchr( "bcdfghjklmnpqrstvwxyz", tolower( c ) ) )

main()
{
  FILE *f = fopen( "szavak.txt", "w" );
  char szo[ 100 ];
  for ( ; ; )
  {
    char szo[ 100 ];
    int i;
    gets( szo );
    if ( !szo[ 0 ] )
      break;
    for ( i = 0; i + 2 < strlen( szo ); ++i )
      if ( angmsh( szo[ i ] ) && angmsh( szo[ i + 1 ] ) &&
           angmsh( szo[ i + 2 ] ) )
      {
        fprintf( f, "%s\n", szo );
        break;
      }
  }
  fclose( f );
}
```

4.3. FELADAT. *Adva van egy szöveges állomány, amely egymástól egy szóközzel elválasztott különböző angol szavakat tartalmaz. Írjunk programot, amely képernyőre írja azokat a szavakat (ezekből akármennyi lehet), amelyekben a legtöbb magánhangzó van!*

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

int mghdarab( char *s )
{
  int darab = 0;
  while ( *s )
  {
    if ( strchr( "aeiou", tolower( *s ) ) != NULL )
      ++darab;
    s++;
  }
  return darab;
}

main()
{
```

```
  FILE *fin;
  char input[ 256 ], szo[ 100 ];
  int max = 0;
  printf( "Az input állomány: " );  scanf( "%s", input );
  fin = fopen( input, "r" );
  while ( fscanf( fin, "%s", szo ) != EOF )
  {
    int mgh = mghdarab( szo );
    if ( mgh > max )
      max = mgh;
  }
  rewind( fin );
  while ( fscanf( fin, "%s", szo ) != EOF )
  {
    int mgh = mghdarab( szo );
    if ( mgh == max )
      printf( "%s\n", szo );
  }
  fclose( fin );
}
```

4.4. FELADAT. *Adott egy szöveges állomány, amelyben magyar szavak vannak, minden szó után egy szóköz áll. Írjunk eljárást, amely képernyőre írja azokat a sorokat (több ilyen is lehet), amelyekben a legkevesebb szó van!*

Feltételezve, hogy a feldolgozandó állomány neve `be.txt`, és az állomány egyetlen sora sem tartalmaz 2000-nél több karaktert, egy lehetséges megoldás a következő:

```
#include <stdio.h>

#define SORHOSSZ 2000

void kevesszosorok()
{
  FILE *f = fopen( "be.txt", "r" );
  char sor[ SORHOSSZ ];
  int i, min = SORHOSSZ;
  while ( fgets( sor, SORHOSSZ, f ) )
  {
    int szoszam = 0;
    for ( i = 0; sor[ i ]; ++i )
      if ( sor[ i ] == ' ' )
        ++szoszam;
    if ( szoszam < min )
      min = szoszam;
  }
  rewind( f );
  while ( fgets( sor, SORHOSSZ, f ) )
  {
    int szoszam = 0;
    for ( i = 0; sor[ i ]; ++i )
      if ( sor[ i ] == ' ' )
        ++szoszam;
    if ( szoszam == min )
      printf( sor );
  }
  fclose( f );
}
```

4.5. FELADAT. *Adva van egy szöveges állomány, amely soraiban egymástól egyetlen szóközzel elválasztott magyar szavak állnak. Írjunk eljárást, amely meghatározza az állományban előforduló*

szavak gyakoriságát! Feltételezhetjük, hogy maximum 200 különböző szó fordul elő.

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

typedef struct tablaelem
{
  char *kulcs;
  int  gyakorisag;
} TABLA[ 200 ];

TABLA tomb;
int darab;

void gyakszamolo( char *allomany )
{
  FILE *f;
  char szo[ 30 ];
  darab = 0;
  f = fopen( allomany, "r" );
  while ( fscanf( f, "%s", szo ) != EOF )
  {
    int i;
    for ( i = 0; i < darab; ++i )
      if ( strcmp( tomb[ i ].kulcs, szo ) == 0 )
        break;
    if ( i == darab )
    {
      ++darab;
      tomb[ i ].kulcs =
        ( char * )malloc( ( strlen( szo ) + 1 ) * sizeof( char ) );
      strcpy( tomb[ i ].kulcs, szo );
      tomb[ i ].gyakorisag = 1;
    }
    else
      ++tomb[ i ].gyakorisag;
  }
  fclose( f );
}
```

4.6. FELADAT. *Adva van egy olyan szöveges állomány, amelynek sorai egyetlen szóközzel elválasztott angol szavakat tartalmaznak. Írjunk programot, amely meghatározza és képernyőre írja a szövegben előforduló szavak gyakoriságát!*

Vegyük észre, hogy a feladat nagyon hasonlít a 3.5. feladatban megfogalmazottakhoz, mindössze annyi a különbség, hogy most nem ismerjük a szöveges állományban található, egymástól különböző szavak maximális darabszámát. Ezért a szavakat és a gyakoriságukat tartalmazó táblázatot célszerű dinamikusan létrehozni.

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

typedef struct tablaelem
{
  char *kulcs;
  int  gyakorisag;
} TABLA;

main()
{
```

```
  FILE *fin;
  char input[ 256 ], szo[ 30 ];
  TABLA *tabla = NULL;
  int darab = 0, i;

  printf( "Az input állomány: " );  scanf( "%s", input );

  fin = fopen( input, "r" );
  while ( fscanf( fin, "%s", szo ) != EOF )
  {
    for ( i = 0; i < darab; ++i )
      if ( strcmp( tabla[ i ].kulcs, szo ) == 0 )
        break;
    if ( i == darab )  /* még nem szerepelt a szó a táblázatban */
    {
      ++darab;
      tabla = ( TABLA * )realloc( tabla, darab * sizeof( TABLA ) );
      tabla[ i ].kulcs =
        ( char * )malloc( ( strlen( szo ) + 1 ) * sizeof( char ) );
      strcpy( tabla[ i ].kulcs, szo );
      tabla[ i ].gyakorisag = 1;
    }
    else               /* a szó benne volt a táblázatban */
      ++tabla[ i ].gyakorisag;
  }
  fclose( fin );

  for ( i = 0; i < darab; ++i )
    printf( "%s: %d\n", tabla[ i ].kulcs, tabla[ i ].gyakorisag );

 for ( i = 0; i < darab; ++i )
   free( tabla[ i ].kulcs );
  free( tabla );
}
```

4.7. FELADAT. *Írjunk eljárást, amely paraméterként megkapja két szöveges állomány nevét és egy sztringet, majd az első állomány azon sorait, melyeknek a vége azonos a sztringgel, átírja a másik állományba! Az első állomány létezik, a másodikat most kell létrehozni.*

A feladatot többféleképpen is meg lehet oldani. Azonban minden megoldáshoz érdemes felhasználni a következő függvényt, amely meghatározza, hogy milyen hosszú egy (létező) állomány leghosszabb sora:

```
#include <stdio.h>

int sorhossz( char *allomany )
{
  int hossz = 0, maxhossz = 0, ch;
  FILE *f = fopen( allomany, "r" );

  while ( ( ch = fgetc( f ) ) != EOF )
  {
    if ( ch == '\n' )
    {
      if ( hossz > maxhossz )
        maxhossz = hossz;
      hossz = 0;
    }
    else
      ++hossz;
  }
```

```
  if ( hossz > maxhossz )
    maxhossz = hossz;

  fclose( f );

  return maxhossz;
}
```

A fenti függvény segítségével pontosan akkora memóriaterületet foglalhatunk le egy állománybeli sor beolvasásához, amennyire ahhoz maximálisan szükség lehet.

1. Ezek után lássuk azt a megoldást, amely az állományból beolvasott sorokat karakterenként hátulról előrefelé haladva dolgozza fel:

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

void sorvegir( char *innev, char *outnev, char *s )
{
  int hossz = sorhossz( innev );
  char *sor = ( char * )malloc( ( hossz + 2 ) * sizeof( char ) );
  FILE *in = fopen( innev, "r" ), *out = fopen( outnev, "w" );
  while ( fgets( sor, hossz + 2, in ) )
  {
    char *sorp = sor + strlen( sor ) - 1, *sp = s + strlen( s ) - 1;
    if ( *sorp == '\n' )
      --sorp;
    while ( sorp >= sor && sp >= s && *sorp == *sp )
    {
      --sorp;
      --sp;
    }
    if ( sp < s )
      fputs( sor, out );
  }
  fclose( in );
  fclose( out );
  free( sor );
}
```

2. Most pedig következzék az a megoldás, amely a sorokat a karaktersorozatok összehasonlítására használt `strcmp()` és `strncmp()` könyvtári függvények segítségével dolgozza fel:

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

void sorvegir( char *innev, char *outnev, char *s )
{
  int hossz = sorhossz( innev );
  char *sor = ( char * )malloc( ( hossz + 2 ) * sizeof( char ) );
  FILE *fin = fopen( innev, "r" ), *fout = fopen( outnev, "w" );
  while ( fgets( sor, hossz + 2, fin ) )
  {
    int sorh = strlen( sor ), szth = strlen( s ), diff = sorh - szth;
    if ( sor[ sorh - 1 ] == '\n' &&
         diff > 0 && !strncmp( s, sor + diff - 1, szth )
         ||
         sor[ sorh - 1 ] != '\n' &&
         diff >= 0 && !strcmp( s, sor + diff ) )
      fputs( sor, fout );
  }
  fclose( fin );
```

```
  fclose( fout );
  free( sor );
}
```

4.8. FELADAT. *Írjunk programot, amely egy létező szöveges állomány minden sorát 80 karakter hosszúságúra egészíti ki szóközökkel, ha a sorok 80 karakternél rövidebbek, és csonkítja őket a végükön, ha 80 karakternél hosszabbak! Az új sorokat egy új szöveges állományba kell írni.*

A 80 karakternél rövidebb sorokat sok helyen ki lehet egészíteni szóközökkel, a leglátványosabb (és legkönnyebben ellenőrizhető) módon a sorok elején, ahogy azt a következő program is csinálja:

```
#include <stdio.h>

#define HATAR 80

main()
{
  FILE *fin, *fout;
  char input[ 256 ], output[ 256 ], sor[ HATAR + 1 ];
  enum { BELUL, TUL, VEGE } allapot = BELUL;
  int darab = 0;

  printf( "Az input állomány: " );   scanf( "%s", input );
  printf( "Az output állomány: " );  scanf( "%s", output );
  fin = fopen( input, "r" );
  fout = fopen( output, "w" );

  while ( allapot != VEGE )
  {
    int ch = fgetc( fin );
    switch ( allapot )
    {
      case BELUL: if ( ch == EOF )
                  {
                    sor[ darab ] = '\0';
                    if ( darab != 0 )
                    {
                      int i;
                      for ( i = 0; i < HATAR - darab; ++i )
                        fputc( ' ', fout );
                      fprintf( fout, "%s", sor );
                    }
                    allapot = VEGE;
                  }
                  else if ( ch == '\n' )
                  {
                    int i;
                    sor[ darab ] = '\0';
                    for ( i = 0; i < HATAR - darab; ++i )
                      fputc( ' ', fout );
                    fprintf( fout, "%s\n", sor );
                    darab = 0;
                  }
                  else
                  {
                    sor[ darab++ ] = ch;
                    if ( darab == HATAR )
                    {
                      sor[ darab ] = '\0';
                      allapot = TUL;
                    }
                  }
```

```
                  break;
      case TUL:   if ( ch == EOF )
                  {
                    fprintf( fout, "%s", sor );
                    allapot = VEGE;
                  }
                  else if ( ch == '\n' )
                  {
                    fprintf( fout, "%s\n", sor );
                    allapot = BELUL;
                    darab = 0;
                  }
                  break;
    }
  }
  fclose( fin );
  fclose( fout );
}
```

4.9. FELADAT. *Írjunk programot, amely egy létező szöveges állomány sorainak mindegyikét 100 hosszúságúra egészíti ki a sorok végén szóközöket szúrva be, ha rövidebb, illetve elhagyva a fölösleges karaktereket, ha hosszabb! Az új sorokat egy most létrehozott szöveges állományba kell kiírni.*

A feladat többféleképpen is megoldható.

1. Első megoldásunkban állapotátmenet-gráfot használunk, hasonlóan a 3.8. feladatban szereplő megoldáshoz.

```
#include <stdio.h>

#define HATAR 100

main()
{
  FILE *fin, *fout;
  char input[ 256 ], output[ 256 ], sor[ HATAR + 1 ];
  enum { BELUL, TUL, VEGE } allapot = BELUL;
  int darab = 0;

  printf( "%s\n", __func__ );
  printf( "Az input állomány: " );   scanf( "%s", input );
  printf( "Az output állomány: " );  scanf( "%s", output );
  fin = fopen( input, "r" );
  fout = fopen( output, "w" );

  while ( allapot != VEGE )
  {
    int ch = fgetc( fin );
    switch ( allapot )
    {
      case BELUL: if ( ch == EOF )
                  {
                    sor[ darab ] = '\0';
                    if ( darab != 0 )
                    {
                      int i;
                      fprintf( fout, "%s", sor );
                      for ( i = 0; i < HATAR - darab; ++i )
                        fputc( ' ', fout );
                    }
                    allapot = VEGE;
```

```
                }
                else if ( ch == '\n' )
                {
                  int i;
                  sor[ darab ] = '\0';
                  fprintf( fout, "%s", sor );
                  for ( i = 0; i < HATAR - darab; ++i )
                    fputc( ' ', fout );
                  fputc( '\n', fout );
                  darab = 0;
                }
                else
                {
                  sor[ darab++ ] = ch;
                  if ( darab == HATAR )
                  {
                    sor[ darab ] = '\0';
                    allapot = TUL;
                  }
                }
                break;
      case TUL:   if ( ch == EOF )
                {
                  fprintf( fout, "%s", sor );
                  allapot = VEGE;
                }
                else if ( ch == '\n' )
                {
                  fprintf( fout, "%s\n", sor );
                  allapot = BELUL;
                  darab = 0;
                }
                break;
    }
  }
  fclose( fin );
  fclose( fout );
}
```

2. Második megoldásunk -- a 3.7. feladathoz hasonlóan -- először meghatározza az input állomány leghosszabb sorának hosszát, majd ezt az adatot felhasználva soronként olvassa végig újra a feldolgozandó állományt.

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

#define max( a, b )   ( ( ( a ) > ( b ) ) ? ( a ) : ( b ) )

#define HATAR 100

int sorhossz( char *allomany )
{
  int hossz = 0, maxhossz = 0, ch;
  FILE *f = fopen( allomany, "r" );

  while ( ( ch = fgetc( f ) ) != EOF )
  {
    if ( ch == '\n' )
    {
      if ( hossz > maxhossz )
        maxhossz = hossz;
```

```
      hossz = 0;
    }
    else
      ++hossz;
  }
  if ( hossz > maxhossz )
    maxhossz = hossz;
  fclose( f );
  return maxhossz;
}

main()
{
  char input[ 256 ], output[ 256 ], *sor;
  int hossz;
  FILE *fin, *fout;
  printf( "Az input állomány: " );   scanf( "%s", input );
  printf( "Az output állomány: " );  scanf( "%s", output );
  hossz = sorhossz( input );
  sor = ( char * )malloc( max( hossz+2, HATAR+2 ) * sizeof( char ) );

  fin = fopen( input, "r" );
  fout = fopen( output, "w" );
  while ( fgets( sor, hossz + 2, fin ) )
  {
    int sorh = strlen( sor );
    int utolso = sor[ sorh - 1 ] != '\n';
    if ( !utolso )
      sor[ --sorh ] = '\0';
    if ( sorh < HATAR )
    {
      int i;
      for ( i = 0; i < HATAR - sorh; ++i )
        strcat( sor, " " );
    }
    else
      sor[ HATAR ] = '\0';
    if ( !utolso )
      strcat( sor, "\n" );
    fputs( sor, fout );
  }

  free( sor );

  fclose( fin );
  fclose( fout );
}
```

4.10. FELADAT. *Írjunk programot, amely egy magyar szavakat tartalmazó szöveges állomány szavait ábécé sorrendben írja át egy másik állományba!*

A feladat megoldásához felhasználjuk a 2.41. feladatban szereplő `strcmphun()` függvényt.

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

int strcmphun( const char *, const char * );

main()
{
  FILE *in = fopen( "be.txt", "r" ), *out = fopen( "ki.txt", "w" );
```

```
  char szo[ 100 ], **tomb = NULL;
  int meret = 0, i, j;
  while ( fscanf( in, "%s", szo ) != EOF )
  {
    tomb = ( char ** )realloc( tomb, ++meret * sizeof( char * ) );
    tomb[ meret - 1 ] =
      ( char * )malloc( ( strlen( szo ) + 1 ) * sizeof( char ) );
    strcpy( tomb[ meret - 1 ], szo );
  }
  fclose( in );
  for ( i = meret - 2; i >= 0; --i )
    for ( j = 0; j <= i; ++j )
      if ( strcmphun( tomb[ j ], tomb[ j + 1 ] ) > 0 )
      {
        char *seged = tomb[ j ];
        tomb[ j ] = tomb[ j + 1 ];
        tomb[ j + 1 ] = seged;
      }
  for ( i = 0; i < meret; ++i )
    fprintf( out, "%s\n", tomb[ i ] );
  fclose( out );
}
```

Az előfordító és a makrók

4.11. FELADAT. *Írjunk egy `MAX( a, b )` alakú egysoros makrót, amely megadja két szám maximumát!*

```
#define MAX( a, b )   ( ( ( a ) > ( b ) ) ? ( a ) : ( b ) )
```

4.12. FELADAT. *Írjunk egy `HAROMMAX( a, b, c )` alakú makrót, amely megadja három szám maximumát!*

A megoldásban felhasználjuk az előző feladatban megírt `MAX( a, b )` makrót.

```
#define HAROMMAX( a, b, c )   ( ( MAX(a, b) < (c) ) ? (c) : MAX(a, b) )
```

4.13. FELADAT. *Írjunk egy `KIIR( x, format )` alakú egysoros makrót, amely képernyőre írja egy változó értékét a megadott formátumban! Például ha `x` egy egész típusú változó, amelynek 24 az értéke, akkor a `KIIR( x, %d )` makró az `x = 24` szöveget írja a képernyőre!*

```
#define KIIR( x, format )   printf( # x " = " # format "\n", x );
```

Változó paraméterszámú függvények

4.14. FELADAT. *Írjunk egy változó paraméterszámú függvényt, amely összefűz tetszőleges számú sztringet, és az így kapott új sztringgel tér vissza! A függvény első paramétere az összefűzendő sztringek darabszáma, további paraméterei pedig maguk a sztringek.*

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

char *konkat( int n, ... )
{
  va_list ap;
  int osszhossz = 0;
  char *sztring = ( char * )malloc( sizeof( char ) );
```

```
  *sztring = '\0';
  va_start( ap, n );
  while ( n-- )
  {
    char *arg = va_arg( ap, char * );
    sztring =
      ( char * )realloc( sztring, ( osszhossz += strlen( arg ) ) + 1 );
    strcat( sztring, arg );
  }
  va_end( ap );
  return sztring;
}
```

Íme egy példa a függvény meghívására:

```
main()
{
  puts( konkat( 5, "Ez ", "egy ", "jó ", "hosszú ", "sztring!" ) );
}
```

4.15. FELADAT. *Írjunk egy változó paraméterszámú függvényt, amely összefűz tetszőleges számú sztringet, és az így kapott új sztringgel tér vissza! A függvény legalább egy sztringet vár paraméterként, utolsó paraméterének a* `NULL` *pointernek kell lennie, ezzel jelezzük a paraméterlista végét.*

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

char *konkat( char *elso, ... )
{
  va_list ap;
  int osszhossz = strlen( elso );
  char *arg;
  char *sztring = ( char * )malloc( ( osszhossz+1 ) * sizeof( char ) );
  strcpy( sztring, elso );
  va_start( ap, elso );
  while ( arg = va_arg( ap, char * ) )
  {
    sztring =
      ( char * )realloc( sztring, ( osszhossz += strlen( arg ) ) + 1 );
    strcat( sztring, arg );
  }
  va_end( ap );
  return sztring;
}
```

Íme egy példa a függvény meghívására:

```
main()
{
  puts( konkat( "Ez ", "egy ", "jó ", "hosszú ", "sztring!", NULL ) );
}
```

A program paraméterei és visszatérési értéke

4.16. FELADAT. *Írjunk programot, amely az első paraméterként megkapott sztringet megkeresi a további paraméterekként megkapott állományok soraiban, és képernyőre írja azokat a sorokat, amelyekben megtalálta! Ha a programnak nem adunk meg második paramétert, akkor a*

standard inputon kell elvégeznie a keresést. Ha egy paramétert sem adunk meg, vagy nem nyitható meg valamelyik állomány, akkor a program írjon hibaüzenetet a standard hibakimenetre! Ha a program több állománynevet kap, akkor az egyes állományok nevét is írja ki, mielőtt megkezdené bennük a keresést! A program visszatérési értéke sikeres lefutás esetén 0, paraméterezési hiba esetén 1, állománymegnyitási hiba esetén pedig 2 legyen! Az egyszerűség kedvéért feltehetjük, hogy egyik állományban sincs 1000 karakternél hosszabb sor. (A feladat tehát a UNIX-ban jól ismert `grep` *parancs egyszerűsített változatának elkészítése.)*

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

#define MAX 1002

int main( int argc, char *argv[] )

  enum  NINCS, EGY, TOBB  all_szam;
  FILE *f;
  char sor[ MAX ];
  int akt;

  switch ( argc )

    case 1:
      fprintf( stderr, "Használat: %s sztring [állományok]", argv[ 0 ] );
      return 1;
    case 2:
      all_szam = NINCS;
      f = stdin;
      break;
    case 3:
      all_szam = EGY;
      break;
    default:
      all_szam = TOBB;


  for ( akt = 2; ; )

    if ( all_szam != NINCS )
      if ( !( f = fopen( argv[ akt ], "r" ) ) )

        fprintf( stderr, "Állománymegnyitási hiba: %s", argv[ akt ] );
        return 2;

    if ( all_szam == TOBB )

      if ( akt != 2 )
        putchar( '' );
      printf( "%s:", argv[ akt ] );

    while ( fgets( sor, MAX, f ) )
      if ( strstr( sor, argv[ 1 ] ) )

        printf( sor );
        if ( sor[ strlen( sor ) - 1 ] != '' )
          putchar( '' );

    if ( all_szam != NINCS )
      fclose( f );
    if ( all_szam != TOBB || ++akt == argc )
      break;
```

```
  return 0;


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

#define MAX 1002

int main( int argc, char *argv[] )
{
  FILE *f;
  char sor[ MAX ];
  int akt;

  switch ( argc )
  {
    case 1:
      fprintf( stderr,
               "Használat: %s sztring [állományok]\n", argv[ 0 ] );
      return 1;
    case 2:
      while ( fgets( sor, MAX, stdin ) )
        if ( strstr( sor, argv[ 1 ] ) )
        {
          printf( sor );
          if ( sor[ strlen( sor ) - 1 ] != '\n' )
            putchar( '\n' );
        }
      break;
    case 3:
      if ( !( f = fopen( argv[ 2 ], "r" ) ) )
      {
        fprintf( stderr, "Állománymegnyitási hiba: %s\n", argv[ 2 ] );
        return 2;
      }
      while ( fgets( sor, MAX, f ) )
        if ( strstr( sor, argv[ 1 ] ) )
        {
          printf( sor );
          if ( sor[ strlen( sor ) - 1 ] != '\n' )
            putchar( '\n' );
        }
      fclose( f );
      break;
    default:
      for ( akt = 2; akt < argc; ++akt )
      {
        if ( !( f = fopen( argv[ akt ], "r" ) ) )
        {
          fprintf( stderr,
                   "Állománymegnyitási hiba: %s\n", argv[ akt ] );
          return 2;
        }
        if ( akt != 2 )
          putchar( '\n' );
        printf( "%s:\n", argv[ akt ] );
        while ( fgets( sor, MAX, f ) )
          if ( strstr( sor, argv[ 1 ] ) )
          {
            printf( sor );
```

```
            if ( sor[ strlen( sor ) - 1 ] != '\n' )
              putchar( '\n' );
          }
        fclose( f );
      }
  }

  return 0;
}
```

A függvény mint típus

4.17. FELADAT. *Írjunk programot, amely a billentyűzetről beolvas egy egyszerű kifejezést, és képernyőre írja annak értékét! A kifejezés csak a* `+`*, a* `-`*, a* `*` *és a* `/` *operátorok valamelyikét és két valós operandust tartalmazhat infix alakban.*

A feladat -- szokás szerint -- sokféleképpen megoldható. Mi most két olyan megoldást mutatunk be, amelyekben függvénymutatókat használunk. Mindkét esetben szükségünk lesz az alábbi függvényekre:

```
double osszead( double a, double b )
{
  return a + b;
}

double kivon( double a, double b )
{
  return a - b;
}
double szoroz( double a, double b )
{
  return a * b;
}

double oszt( double a, double b )
{
  return a / b;
}
```

1. Az első megoldásban felveszünk egy függvénymutatót, amelynek értékét a beolvasott operátortól függően állítjuk be egy `switch` utasításban:

```
#include <stdio.h>

main()
{
  double a, b, ( *muv )( double, double );
  char op;

  scanf( "%lf%c%lf", &a, &op, &b );
  switch( op )
  {
    case '+':
      muv = osszead;
      break;
    case '-':
      muv = kivon;
      break;
    case '*':
      muv = szoroz;
```

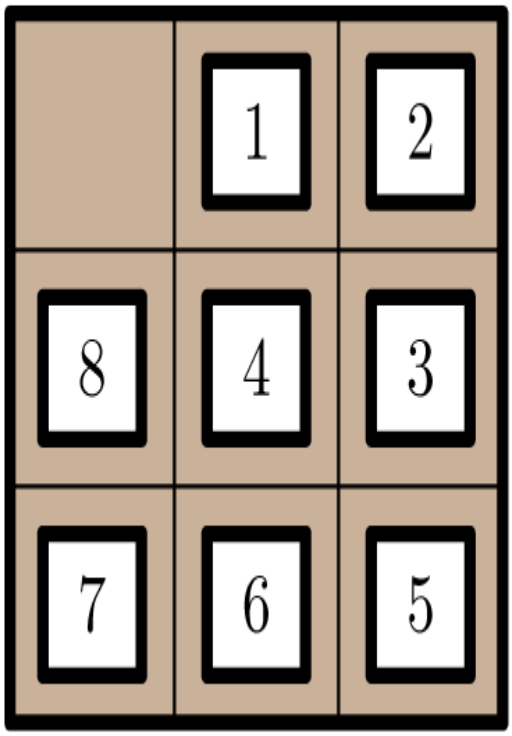
```
      break;
    case '/':
      muv = oszt;
      break;
  }
  printf( "%lf\n", muv( a, b ) );
}
```

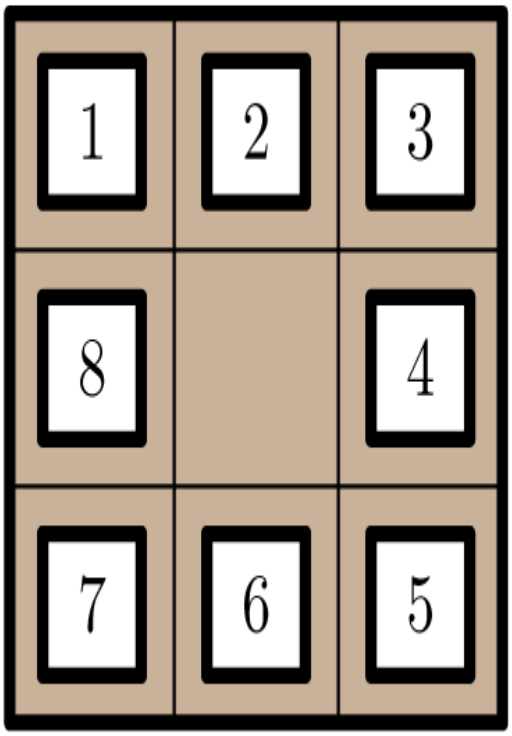
2. A másik megoldásban egy függvénymutatókból álló négyelemű tömböt deklarálunk, amelynek mindjárt kezdőértéket is adunk. A `switch` utasításban most a tömbindexet állítjuk be:

```
#include <stdio.h>

main()
{
  double a, b, ( *muv[ 4 ] )( double, double ) =
              { osszead, kivon, szoroz, oszt };
  char op;

  scanf( "%lf%c%lf", &a, &op, &b );
  switch( op )
  {
    case '+':
      op = 0;
      break;
    case '-':
      op = 1;
      break;
    case '*':
      op = 2;
      break;
    case '/':
      op = 3;
      break;
  }
  printf( "%lf\n", muv[ op ]( a, b ) );
}
```

4.18. FELADAT. *Adott nyolc számozott lapocskának a 4.1. ábra bal oldalán látható elrendezése. Írjunk programot, amely a lapocskák vízszintes, illetve függőleges irányú tologatásaival előállítja az ábra jobb oldalán látható elrendezést!*

4.1. ábra -





A feladatot a következőképpen oldjuk meg. A tologatások feltételeit és hatásait az üres mezőhöz viszonyítva fogalmazzuk meg, ugyanis egy üres mezőnek mindig kell lennie a táblán.

Az üres mezőt mozgathatjuk felfelé, ha nem az első sorban van, mozgathatjuk balra, ha nem az első oszlopban van, mozgathatjuk lefelé, ha nem a harmadik sorban van, és mozgathatjuk jobbra, ha nem a harmadik oszlopban van.

Az üres mező felfelé történő mozgatásakor a felette lévő elem, balra mozgatásakor a tőle balra lévő elem, lefelé mozgatásakor a tőle egy sorral lejjebb lévő elem, míg jobbra mozgatásakor a tőle jobbra lévő elem kerül az üres mező helyére.

A kiinduló állapotból indulva, a lehetséges tologatások mindegyikét végrehajtva, szélességi kereséssel keressük a célállapotot. Az előállított állapotokat egy egyirányban láncolt listába fűzzük, és figyeljük, hogy egyszer már elért állapotot még egyszer ne vegyünk fel a listába. Ha elérjük a célállapotot, megállunk, és kiírjuk a megoldást: a tologatások irányát és az előálló közbenső állapotokat.

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

typedef enum { semerre = -1, balra, fel, jobbra, le } IRANY;
typedef int TABLA[ 3 ][ 3 ];

TABLA kezdo = { { 0, 1, 2 }, { 8, 4, 3 }, { 7, 6, 5 } };
TABLA cel   = { { 1, 2, 3 }, { 8, 0, 4 }, { 7, 6, 5 } };

void ures_pozicio( TABLA tabla, int *x, int *y )
{
  int i, j;
  for ( i = 0; i < 3; ++i )
    for ( j = 0; j < 3; ++j )
      if ( tabla[ i ][ j ] == 0 )
      {
```

```
        *x = i;
        *y = j;
        return;
      }
}

int balra_mehet( TABLA tabla )
{
  int x, y;
  ures_pozicio( tabla, &x, &y );
  return y > 0;
}

int fel_mehet( TABLA tabla )
{
  int x, y;
  ures_pozicio( tabla, &x, &y );
  return x > 0;
}

int jobbra_mehet( TABLA tabla )
{
  int x, y;
  ures_pozicio( tabla, &x, &y );
  return y < 2;
}

int le_mehet( TABLA tabla )
{
  int x, y;
  ures_pozicio( tabla, &x, &y );
  return x < 2;
}

int ( *elofeltetel[ 4 ] )( TABLA ) =
  { balra_mehet, fel_mehet, jobbra_mehet, le_mehet };

void balra_megy( TABLA tabla )
{
  int x, y, seged;
  ures_pozicio( tabla, &x, &y );
  tabla[ x ][ y ] = tabla[ x ][ y - 1 ];
  tabla[ x ][ y - 1 ] = 0;
}

void fel_megy( TABLA tabla )
{
  int x, y, seged;
  ures_pozicio( tabla, &x, &y );
  tabla[ x ][ y ] = tabla[ x - 1 ][ y ];
  tabla[ x - 1 ][ y ] = 0;
}

void jobbra_megy( TABLA tabla )
{
  int x, y, seged;
  ures_pozicio( tabla, &x, &y );
  tabla[ x ][ y ] = tabla[ x ][ y + 1 ];
  tabla[ x ][ y + 1 ] = 0;
}

void le_megy( TABLA tabla )
```

```
{
  int x, y, seged;
  ures_pozicio( tabla, &x, &y );
  tabla[ x ][ y ] = tabla[ x + 1 ][ y ];
  tabla[ x + 1 ][ y ] = 0;
}

void ( *hatas[ 4 ] )( TABLA ) = { balra_megy,  fel_megy,
                                  jobbra_megy, le_megy };

typedef struct listaelem
{
  TABLA             tabla;
  IRANY             irany;
  struct listaelem *szulo;
  struct listaelem *kovetkezo;
} LISTAELEM;

LISTAELEM *zart, *nyilt, *utolso;

void inicializal()
{
  zart = nyilt = utolso = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  memcpy( nyilt->tabla, kezdo, sizeof( TABLA ) );
  nyilt->irany = semerre;
  nyilt->szulo = nyilt->kovetkezo = NULL;
}

int volt( TABLA tabla )
{
  LISTAELEM *seged = zart;
  while ( seged && memcmp( seged->tabla, tabla, sizeof( TABLA ) ) )
    seged = seged->kovetkezo;
  return seged != NULL;
}

void kiterjeszt()
{
  IRANY i;
  for ( i = balra; i <= le; ++i )
    if ( elofeltetel[ i ]( nyilt->tabla ) )
    {
      TABLA tabla;
      memcpy( tabla, nyilt->tabla, sizeof( TABLA ) );
      hatas[ i ]( tabla );
      if ( !volt( tabla ) )
      {
        LISTAELEM *uj = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
        if ( !uj )
        {
          fprintf( stderr, "Nincs elég memória!\n" );
          break;
        }
        memcpy( uj->tabla, tabla, sizeof( TABLA ) );
        uj->irany = i;
        uj->szulo = nyilt;
        uj->kovetkezo = NULL;
        utolso->kovetkezo = uj;
        utolso = uj;
      }
    }
  nyilt = nyilt->kovetkezo;
```

```
}

void kiir( LISTAELEM *mutato )
{
  if ( mutato )
  {
    int i, j;
    kiir( mutato->szulo );
    switch ( mutato->irany )
    {
      case balra:  puts( "balra\n" );   break;
      case fel:    puts( "fel\n" );     break;
      case jobbra: puts( "jobbra\n" );  break;
      case le:     puts( "le\n" );      break;
    }
    for ( i = 0; i < 3; ++i )
    {
      for ( j = 0; j < 3; ++j )
        printf( j ? " %d" : "%d", mutato->tabla[ i ][ j ] );
      putchar( '\n' );
    }
    putchar( '\n' );
  }
}

void felszabadit()
{
  while ( zart )
  {
    LISTAELEM *seged = zart->kovetkezo;
    free( zart );
    zart = seged;
  }
}

main()
{
  inicializal();
  while ( nyilt )
  {
    if ( !memcmp( nyilt->tabla, cel, sizeof( TABLA ) ) )
      break;
    kiterjeszt();
  }
  if ( nyilt )
  {
    puts( "A feladat egy lehetséges megoldása:\n" );
    kiir( nyilt );
  }
  else
    puts( "A probléma nem oldható meg." );
  felszabadit();
}
```

A program kimenete a következő:

```
A feladat egy lehetséges megoldása:

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

```
jobbra

1 0 2
8 4 3
7 6 5

jobbra

1 2 0
8 4 3
7 6 5

le

1 2 3
8 4 0
7 6 5

balra

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

5. fejezet - Adatszerkezetek megvalósítása

Tartalom

Láncolt listák

Ebben az alfejezetben az egyirányban láncolt listával, a kétirányban láncolt listával és a cirkuláris (körkörös) listával foglalkozunk.

Egyirányban láncolt lista

A következőkben az alábbi típusdefiníciót fogjuk használni a listaelemek kezelésére (a listában egész értékeket tárolunk):

```
typedef struct listaelem {
  int adat;
  struct listaelem *kov;
} LISTAELEM;
```

5.1. FELADAT. *Írjunk eljárást, amely bejárja a paraméterként megkapott egyirányban láncolt listát, és képernyőre írja az elemeit!*

Paraméterként az eljárásunk a lista első elemére mutató pointert, a listafejet kapja meg. Mivel minden listaelem mutatórésze tekinthető az első elem elhagyásával képzett részlista fejének, ezért előrefelé a lista iteratívan és rekurzívan is könnyen bejárható, míg hátrafelé a bejárást rekurzívan célszerű elvégezni.

1.

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

```
void bejar_elore( LISTAELEM *fej )
{
  LISTAELEM *seged;
  for ( seged = fej; seged; seged = seged->kov )
    printf( "%d\n", seged->adat );
}
```

2.

```
#include <stdio.h>

void bejar_elore_rek( LISTAELEM *fej )
{
  if ( fej )
  {
    printf( "%d\n", fej->adat );
    bejar_elore_rek( fej->kov );
  }
}
```

3.

```
#include <stdio.h>

void bejar_hatra_rek( LISTAELEM *fej )
{
  if ( fej )
  {
    bejar_hatra_rek( fej->kov );
    printf( "%d\n", fej->adat );
  }
}
```

5.2. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott egyirányban láncolt lista elejére beszúrja a szintén paraméterként megkapott értéket, és visszatér az új lista fejével!*

```
#include <stdlib.h>

LISTAELEM *beszur_eleje( LISTAELEM *fej, int adat )
{
  LISTAELEM *ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  ujelem->adat = adat;
  ujelem->kov = fej;
  return ujelem;
}
```

5.3. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott egyirányban láncolt lista végére beszúrja a szintén paraméterként megkapott értéket, és visszatér az új lista fejével!*

1.

```
#include <stdlib.h>

LISTAELEM *beszur_vege( LISTAELEM *fej, int adat )
{
  LISTAELEM *ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  ujelem->adat = adat;
  ujelem->kov = NULL;
  if ( !fej )
    return ujelem;
  else {
    LISTAELEM *seged = fej;
```

```
    while ( seged->kov )
      seged = seged->kov;
    seged->kov = ujelem;
  }
  return fej;
}
```

2.

```
#include <stdlib.h>

LISTAELEM *beszur_vege_rek( LISTAELEM *fej, int adat )
{
  if ( !fej )
  {
    fej = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
    fej->adat = adat;
    fej->kov = NULL;
  }
  else
    fej->kov = beszur_vege_rek( fej->kov, adat );
  return fej;
}
```

5.4. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott, növekvően rendezett egyirányban láncolt listába beszúrja a szintén paraméterként megkapott értéket úgy, hogy a lista továbbra is rendezett maradjon, és visszatér az új lista fejével!*

1.

```
#include <stdlib.h>

LISTAELEM *beszur_rendezve( LISTAELEM *fej, int adat )
{
  LISTAELEM *ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  LISTAELEM *akt = fej, *elozo = NULL;
  ujelem->adat = adat;
  for ( ; akt && akt->adat < adat; elozo = akt, akt = akt->kov )
    ;
  ujelem->kov = akt;
  ( elozo ? elozo->kov : fej ) = ujelem;
  return fej;
}
```

2.

```
#include <stdlib.h>

LISTAELEM *beszur_rendezve_rek( LISTAELEM *fej, int adat )
{
  if ( !fej || fej->adat >= adat )
  {
    LISTAELEM *ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
    ujelem->adat = adat;
    ujelem->kov = fej;
    return ujelem;
  }
  fej->kov = beszur_rendezve_rek( fej->kov, adat );
  return fej;
}
```

5.5. FELADAT. *Írjunk függvényt, amely törli a paraméterként megkapott egyirányban láncolt*

lista első elemét (ha létezik), és visszatér az új lista fejével!

```
#include <stdlib.h>

LISTAELEM *torol_eleje( LISTAELEM *fej )
{
  LISTAELEM *elso = fej;
  if ( fej )
  {
    fej = fej->kov;
    free( elso );
  }
  return fej;
}
```

5.6. FELADAT. *Írjunk függvényt, amely törli a paraméterként megkapott egyirányban láncolt lista utolsó elemét (ha létezik), és visszatér az új lista fejével!*

1.

```
#include <stdlib.h>

LISTAELEM *torol_vege( LISTAELEM *fej )
{
  if ( fej )
  {
    LISTAELEM *akt = fej, *elozo = NULL;
    for ( ; akt->kov; elozo = akt, akt = akt->kov )
      ;
    ( elozo ? elozo->kov : fej ) = NULL;
    free( akt );
  }
  return fej;
}
```

2.

```
#include <stdlib.h>

LISTAELEM *torol_vege_rek( LISTAELEM *fej )
{
  if ( fej )
  {
    if ( !fej->kov )
    {
      free( fej );
      return NULL;
    }
    fej->kov = torol_vege_rek( fej->kov );
  }
  return fej;
}
```

5.7. FELADAT. *Írjunk függvényt, amely megkeresi és törli a paraméterként megkapott, növekvően rendezett egyirányban láncolt listából a szintén paraméterként megkapott értéket (ha létezik), és visszatér az új lista fejével! Ha a keresett érték többször is előfordul a listában, akkor csak egy előfordulását kell törölni.*

1.

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

```
LISTAELEM *torol_rendezve( LISTAELEM *fej, int adat )
{
  LISTAELEM *akt = fej, *elozo = NULL;
  while ( akt && akt->adat < adat )
  {
    elozo = akt;
    akt = akt->kov;
  }
  if ( akt && akt->adat == adat )
  {
    ( elozo ? elozo->kov : fej ) = akt->kov;
    free( akt );
  }
  return fej;
}
```

2.

```
#include <stdlib.h>

LISTAELEM *torol_rendezve_rek( LISTAELEM *fej, int adat )
{
  if ( fej && fej->adat <= adat )
    if ( fej->adat == adat )
    {
      LISTAELEM *seged = fej;
      fej = seged->kov;
      free( seged );
      return fej;
    }
    else
      fej->kov = torol_rendezve_rek( fej->kov, adat );
  return fej;
}
```

5.8. FELADAT. *Írjunk rekurzív függvényt, amely visszaad egy olyan egyirányban láncolt listát, amely a paraméterként megkapott lista elemeit tartalmazza fordított sorrendben!*

```
#include <stdlib.h>

LISTAELEM *fordit( LISTAELEM *lista )
{
  LISTAELEM *reszlista, *ujelem;
  if ( !lista )
    return NULL;
  reszlista = fordit( lista->kov );
  ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  ujelem->adat = lista->adat;
  ujelem->kov = NULL;
  if ( !reszlista )
    reszlista = ujelem;
  else
  {
    LISTAELEM *seged = reszlista;
    while ( seged->kov )
      seged = seged->kov;
    seged->kov = ujelem;
  }
  return reszlista;
}
```

Kétirányban láncolt lista

A következőkben az alábbi típusdefiníciókat fogjuk használni a listaelemek és a listafejek kezelésére (a listában egész értékeket tárolunk):

```
typedef struct listaelem {
  int adat;
  struct listaelem *elozo, *kov;
} LISTAELEM;

typedef struct {
  LISTAELEM *elso, *utolso;
} FEJEK;
```

5.9. FELADAT. *Írjunk eljárást, amely bejárja a paraméterként megkapott kétirányban láncolt listát, és képernyőre írja az elemeit!*

Paraméterként az eljárásunk a listafejeket tartalmazó struktúrát kapja meg. Mivel a kétirányban láncolt lista teljesen szimmetrikus, ezért minden műveletét (így a bejárást is) kétféleképpen valósíthatjuk meg: az első elemtől az utolsó felé, vagy az utolsó elemtől az első felé haladva.

1.

```
#include <stdio.h>

void bejar_elore( FEJEK fejek )
{
  LISTAELEM *seged;
  for ( seged = fejek.elso; seged; seged = seged->kov )
    printf( "%d\n", seged->adat );
}
```

2.

```
#include <stdio.h>

void bejar_hatra( FEJEK fejek )
{
  LISTAELEM *seged;
  for ( seged = fejek.utolso; seged; seged = seged->elozo )
    printf( "%d\n", seged->adat );
}
```

5.10. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott kétirányban láncolt lista elejére beszúrja a szintén paraméterként megkapott értéket, és visszatér az új listafejekkel!*

```
#include <stdlib.h>

FEJEK beszur_eleje( FEJEK fejek, int adat )
{
  LISTAELEM *ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  ujelem->adat = adat;
  ujelem->elozo = NULL;
  ujelem->kov = fejek.elso;
  ( fejek.utolso ? ujelem->kov->elozo : fejek.utolso ) =
    fejek.elso = ujelem;
  return fejek;
}
```

5.11. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott kétirányban láncolt lista végére beszúrja a szintén paraméterként megkapott értéket, és visszatér az új listafejekkel!*

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

```
FEJEK beszur_vege( FEJEK fejek, int adat )
{
  LISTAELEM *ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  ujelem->adat = adat;
  ujelem->kov = NULL;
  ujelem->elozo = fejek.utolso;
  ( fejek.elso ? ujelem->elozo->kov : fejek.elso ) =
    fejek.utolso = ujelem;
  return fejek;
}
```

5.12. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott, növekvően rendezett kétirányban láncolt listába beszúrja a szintén paraméterként megkapott értéket úgy, hogy a lista továbbra is rendezett maradjon, és visszatér az új listafejekkel!*

```
#include <stdlib.h>

FEJEK beszur_rendezve( FEJEK fejek, int adat )
{
  LISTAELEM *ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  LISTAELEM *akt, *elozo;
  ujelem->adat = adat;
  for ( akt = fejek.elso, elozo = NULL; akt && akt->adat < adat;
        elozo = akt, akt = akt->kov )
    ;
  ( ( ujelem->elozo = elozo ) ? elozo->kov : fejek.elso ) =
    ( ( ujelem->kov = akt ) ? akt->elozo : fejek.utolso ) = ujelem;
  return fejek;
}
```

5.13. FELADAT. *Írjunk függvényt, amely törli a paraméterként megkapott kétirányban láncolt lista első elemét (ha létezik), és visszatér az új listafejekkel!*

```
#include <stdlib.h>

FEJEK torol_eleje( FEJEK fejek )
{
  LISTAELEM *seged = fejek.elso;
  if ( seged )
  {
    ( ( fejek.elso = seged->kov ) ? fejek.elso->elozo : fejek.utolso ) =
      NULL;
    free( seged );
  }
  return fejek;
}
```

5.14. FELADAT. *Írjunk függvényt, amely törli a paraméterként megkapott kétirányban láncolt lista utolsó elemét (ha létezik), és visszatér az új listafejekkel!*

```
#include <stdlib.h>

FEJEK torol_vege( FEJEK fejek )
{
  LISTAELEM *seged = fejek.utolso;
  if ( seged )
  {
    ( ( fejek.utolso = seged->elozo ) ?
      fejek.utolso->kov : fejek.elso ) = NULL;
    free( seged );
  }
```

```
  return fejek;
}
```

5.15. FELADAT. *Írjunk függvényt, amely megkeresi és törli a paraméterként megkapott, növekvően rendezett kétirányban láncolt listából a szintén paraméterként megkapott értéket (ha létezik), és visszatér az új listafejekkel! Ha a keresett érték többször is előfordul a listában, akkor csak egy előfordulását kell törölni.*

```
#include <stdlib.h>

FEJEK torol_rendezve( FEJEK fejek, int adat )
{
  LISTAELEM *seged = fejek.elso;
  while ( seged && seged->adat < adat )
    seged = seged->kov;
  if ( seged && seged->adat == adat )
  {
    ( seged->elozo ? seged->elozo->kov : fejek.elso ) = seged->kov;
    ( seged->kov ? seged->kov->elozo : fejek.utolso ) = seged->elozo;
    free( seged );
  }
  return fejek;
}
```

Cirkuláris lista

A következőkben az alábbi típusdefiníciót fogjuk használni a listaelemek kezelésére (a listában egész értékeket tárolunk):

```
typedef struct listaelem {
  int adat;
  struct listaelem *kov;
} LISTAELEM;
```

5.16. FELADAT. *Írjunk eljárást, amely bejárja a paraméterként megkapott cirkuláris listát, és képernyőre írja az elemeit!*

```
#include <stdio.h>

void bejar_elore( LISTAELEM *fej )
{
  LISTAELEM *seged = fej;
  if ( fej )
    do
    {
      printf( "%d\n", seged->adat );
      seged = seged->kov;
    } while ( seged != fej );
}
```

5.17. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott cirkuláris lista elejére beszúrja a szintén paraméterként megkapott értéket, és visszatér az új lista fejével!*

```
#include <stdlib.h>

LISTAELEM *beszur_eleje( LISTAELEM *fej, int adat )
{
  LISTAELEM *ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  ujelem->adat = adat;
  if ( !fej )
    ujelem->kov = ujelem;
  else
```

```
  {
    LISTAELEM *seged = fej;
    while ( seged->kov != fej )
      seged = seged->kov;
    seged->kov = ujelem;
    ujelem->kov = fej;
  }
  return ujelem;
}
```

5.18. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott cirkuláris lista végére beszúrja a szintén paraméterként megkapott értéket, és visszatér az új lista fejével!*

```
#include <stdlib.h>

LISTAELEM *beszur_vege( LISTAELEM *fej, int adat )
{
  LISTAELEM *ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  ujelem->adat = adat;
  if ( !fej )
    fej = ujelem->kov = ujelem;
  else
  {
    LISTAELEM *seged = fej;
    while ( seged->kov != fej )
      seged = seged->kov;
    seged->kov = ujelem;
    ujelem->kov = fej;
  }
  return fej;
}
```

5.19. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott, növekvően rendezett cirkuláris listába beszúrja a szintén paraméterként megkapott értéket úgy, hogy a lista továbbra is rendezett maradjon, és visszatér az új lista fejével!*

```
#include <stdlib.h>

LISTAELEM *beszur_rendezve( LISTAELEM *fej, int adat )
{
  LISTAELEM *ujelem = ( LISTAELEM * )malloc( sizeof( LISTAELEM ) );
  ujelem->adat = adat;
  if ( !fej )
     fej = ujelem->kov = ujelem;
  else
  {
    LISTAELEM *akt, *elozo;
    for ( akt = fej, elozo = NULL; akt->kov != fej && akt->adat < adat;
          elozo = akt, akt = akt->kov )
      ;
    if ( akt->adat >= adat )
    {
      ujelem->kov = akt;
      if ( elozo )
        elozo->kov = ujelem;
      else
      {
        while ( akt->kov != fej )
          akt = akt->kov;
        fej = akt->kov = ujelem;
      }
    }
```

```
    else
    {
      ujelem->kov = fej;
      akt->kov = ujelem;
    }
  }
  return fej;
}
```

5.20. FELADAT. *Írjunk függvényt, amely törli a paraméterként megkapott cirkuláris lista első elemét (ha létezik), és visszatér az új lista fejével!*

```
#include <stdlib.h>

LISTAELEM *torol_eleje( LISTAELEM *fej )
{
  if ( fej )
  {
    LISTAELEM *regifej = fej;
    if ( fej->kov == fej )
      fej = NULL;
    else
    {
      LISTAELEM *seged = fej->kov;
      while ( seged->kov != fej )
        seged = seged->kov;
      fej = seged->kov = fej->kov;
    }
    free( regifej );
  }
  return fej;
}
```

5.21. FELADAT. *Írjunk függvényt, amely törli a paraméterként megkapott cirkuláris lista utolsó elemét (ha létezik), és visszatér az új lista fejével!*

```
#include <stdlib.h>

LISTAELEM *torol_vege( LISTAELEM *fej )
{
  if ( fej )
  {
    LISTAELEM *akt = fej;
    if ( fej->kov == fej )
      fej = NULL;
    else
    {
      LISTAELEM *elozo;
      for ( ; akt->kov != fej; elozo = akt, akt = akt->kov )
        ;
      elozo->kov = fej;
    }
    free( akt );
  }
  return fej;
}
```

5.22. FELADAT. *Írjunk függvényt, amely megkeresi és törli a paraméterként megkapott, növekvően rendezett cirkuláris listából a szintén paraméterként megkapott értéket (ha létezik), és visszatér az új lista fejével! Ha a keresett érték többször is előfordul a listában, akkor csak egy előfordulását kell törölni.*

```
#include <stdlib.h>

LISTAELEM *torol_rendezve( LISTAELEM *fej, int adat )
{
  if ( fej )
  {
    LISTAELEM *akt, *elozo;
    for ( akt = fej, elozo = NULL; akt->kov != fej && akt->adat < adat;
          elozo = akt, akt = akt->kov )
      ;
    if ( akt->adat == adat )
    {
      if ( elozo )
        elozo->kov = akt->kov;
      else
        if ( akt->kov == fej )
          fej = NULL;
        else
        {
          elozo = akt->kov;
          while ( elozo->kov != fej )
            elozo = elozo->kov;
          fej = elozo->kov = akt->kov;
        }
      free( akt );
    }
  }
  return fej;
}
```

Rendezések és keresések egydimenziós tömbben

Minden adatszerkezet folytonos reprezentálásakor alkalmazhatjuk az egydimenziós tömböt. Az így reprezentált adatszerkezetekben a rendező és kereső algoritmusok könnyen implementálhatók a C tömb típusának segítségével.

Rendezések

A következőkben a leggyakrabban használt rendezési algoritmusokat tekintjük át az egyszerűbbektől a hatékonyabbakig. Az eljárások a rendezendő tömböt, valamint annak méretét veszik át paraméterként. A tömb mindig egész típusú értékeket tartalmaz, és mindig növekvő sorrendbe rendezzük.

Az alfejezetben többször hivatkozunk az alábbi eljárásra, amely megcseréli egy egészeket tartalmazó tömb két, adott indexű elemét:

```
void csere( int tomb[], int i, int j )
{
  int seged = tomb[ i ];
  tomb[ i ] = tomb[ j ];
  tomb[ j ] = seged;
}
```

5.23. FELADAT. *Írjunk eljárást, amely növekvő sorrendbe rendezi a paraméterként megkapott, egészeket tartalmazó tömböt maximumkiválasztásos rendezéssel!*

```
void maxkival( int tomb[], int meret )
{
  int j;
  for ( j = meret - 1; j > 0; --j )
```

```
  {
    int max = j, i;
    for ( i = 0; i < j; ++i )
      if ( tomb[ i ] > tomb[ max ] )
        max = i;
    csere( tomb, max, j );
  }
}
```

5.24. FELADAT. *Írjunk eljárást, amely növekvő sorrendbe rendezi a paraméterként megkapott, egészeket tartalmazó tömböt minimumkiválasztásos rendezéssel!*

```
void minkival( int tomb[], int meret )
{
  int j;
  for ( j = 0; j < meret - 1; ++j )
  {
    int min = j, i;
    for ( i = j + 1; i < meret; ++i )
      if ( tomb[ i ] < tomb[ min ] )
        min = i;
    csere( tomb, min, j );
  }
}
```

5.25. FELADAT. *Írjunk eljárást, amely növekvő sorrendbe rendezi a paraméterként megkapott, egészeket tartalmazó tömböt beszúrásos rendezéssel!*

```
void beszurasos( int tomb[], int meret )
{
  int j;
  for ( j = 1; j < meret; ++j )
  {
    int kulcs = tomb[ j ], i = j - 1;
    while ( i >= 0 && tomb[ i ] > kulcs )
    {
      tomb[ i + 1 ] = tomb[ i ];
      --i;
    }
    tomb[ i + 1 ] = kulcs;
  }
}
```

5.26. FELADAT. *Írjunk eljárást, amely növekvő sorrendbe rendezi a paraméterként megkapott, egészeket tartalmazó tömböt buborékrendezéssel!*

```
void buborek1( int tomb[], int meret )
{
  int i, j;
  for ( i = meret - 1; i > 0; --i )
    for ( j = 0; j < i; ++j )
      if ( tomb[ j + 1 ] < tomb[ j ] )
        csere( tomb, j, j + 1 );
}
```

A rendezés gyorsítható, ha a külső ciklusban figyeljük, hogy teljesült-e a belső ciklusban lévő `if` utasítás feltétele. Ha ugyanis egyszer sem végeztünk cserét a belső ciklusban, akkor a tömbünk már rendezett, tehát feleslegesek a további vizsgálatok.

```
#define HAMIS 0
#define IGAZ  ( !HAMIS )
```

```
void buborek2( int tomb[], int meret )
{
  int i, j, voltcsere = IGAZ;
  for ( i = meret - 1; i > 0 && voltcsere; --i )
  {
    voltcsere = HAMIS;
    for ( j = 0; j < i; ++j )
      if ( tomb[ j + 1 ] < tomb[ j ] )
      {
        csere( tomb, j, j + 1 );
        voltcsere = IGAZ;
      }
  }
}
```

Még tovább gyorsíthatjuk a rendezést, ha nem csak azt jegyezzük meg, hogy történt-e csere, hanem azt is, hogy hol történt az utolsó csere a külső ciklus előző iterációjában. A következő iterációban ugyanis elegendő addig az elemig végezni az összehasonlításokat.

```
void buborek3( int tomb[], int meret )
{
  int i, j, utolsocsere;
  for ( i = meret - 1; i > 0; i = utolsocsere )
  {
    utolsocsere = 0;
    for ( j = 0; j < i; ++j )
      if ( tomb[ j + 1 ] < tomb[ j ] )
      {
        csere( tomb, j, j + 1 );
        utolsocsere = j;
      }
  }
}
```

5.27. FELADAT. *Írjunk eljárást, amely növekvő sorrendbe rendezi a paraméterként megkapott, egészeket tartalmazó tömböt Shell-rendezéssel!*

A Shell-rendezés egy összetett algoritmus, ami azt jelenti, hogy a rendezendő tömböt halmazokra osztja, amelyeket külön-külön rendez valamilyen egyszerű rendező algoritmussal. Esetünkben ez a beszúrásos rendezés.

```
void shell( int tomb[], int meret )
{
  int lk[] = { 6, 3, 1 };
  int lkindex;
  for ( lkindex = 0; lkindex < sizeof( lk ) / sizeof( int ); ++lkindex )
  {
    int lepeskoz = lk[ lkindex ];
    int eltolas, j;
    for ( eltolas = 0; eltolas < lepeskoz; ++eltolas )
      for ( j = lepeskoz + eltolas; j < meret; j += lepeskoz )
      {
        int i = j - lepeskoz;
        int kulcs = tomb[ j ];
        while ( i >= 0 && tomb[ i ] > kulcs )
        {
          tomb[ i + lepeskoz ] = tomb[ i ];
          i -= lepeskoz;
        }
        tomb[ i + lepeskoz ] = kulcs;
      }
```

```
  }
}
```

Lerövidíthetjük a kódot, ha észrevesszük, hogy a második ciklus elhagyható, ha a harmadik ciklusban egyesével lépkedünk. Bár így eggyel kevesebb ciklus lesz, a futási idő ezzel nem változik, hiszen csak az összehasonlítások sorrendjét változtattuk meg.

```
void shell2( int tomb[], int meret )
{
  int lk[] = { 6, 3, 1 };
  int lkindex;
  for ( lkindex = 0; lkindex < sizeof( lk ) / sizeof( int ); ++lkindex )
  {
    int lepeskoz = lk[ lkindex ];
    int j;
    for ( j = lepeskoz; j < meret; ++j )
    {
      int i = j - lepeskoz;
      int kulcs = tomb[ j ];
      while ( i >= 0 && tomb[ i ] > kulcs )
      {
        tomb[ i + lepeskoz ] = tomb[ i ];
        i -= lepeskoz;
      }
      tomb[ i + lepeskoz ] = kulcs;
    }
  }
}
```

5.28. FELADAT. *Írjunk eljárást, amely növekvő sorrendbe rendezi a paraméterként megkapott, egészeket tartalmazó tömböt gyorsrendezéssel!*

```
void gyors( int tomb[], int bal, int jobb )
{
  if ( bal < jobb )
  {
    int also = bal, felso = jobb + 1, kulcs = tomb[ bal ];
    for ( ; ; )
    {
      while ( ++also < felso && tomb[ also ] < kulcs )
        ;
      while ( tomb[ --felso ] > kulcs )
        ;
      if ( also >= felso )
        break;
      csere( tomb, also, felso );
    }
    csere( tomb, felso, bal );
    gyors( tomb, bal, felso - 1 );
    gyors( tomb, felso + 1, jobb );
  }
}
```

Az alábbi megoldás abban különbözik az előzőtől, hogy most a felosztást külön eljárásban végezzük egy másik módszerrel, és nem a bal szélső, hanem a jobb szélső elemet választjuk rendezési kulcsnak.

```
int feloszt( int tomb[], int bal, int jobb )
{
  int kulcs = tomb[ jobb ], hatar = bal - 1, akt;
  for ( akt = bal; akt < jobb; ++akt )
```

```
    if ( tomb[ akt ] <= kulcs )
      csere( tomb, ++hatar, akt );
  csere( tomb, hatar + 1, jobb );
  return hatar + 1;
}

void gyors2( int tomb[], int bal, int jobb )
{
  if ( bal < jobb )
  {
    int kozepso = feloszt( tomb, bal, jobb );
    gyors2( tomb, bal, kozepso - 1 );
    gyors2( tomb, kozepso + 1, jobb );
  }
}
```

Keresések

5.29. FELADAT. *Írjunk függvényt, amely teljes kereséssel megkeresi a paraméterként megkapott tömbben a szintén paraméterként megkapott értéket, és az első olyan elem indexével tér vissza, amelynek értéke a keresett érték! Ha a keresett érték nincs a tömbben, akkor* -1 *-et kell visszaadni.*

```
int teljes( int tomb[], int meret, int ertek )
{
  int i;
  for ( i = 0; i < meret && tomb[ i ] != ertek; ++i )
    ;
  return i < meret ? i : -1;
}
```

5.30. FELADAT. *Írjunk függvényt, amely lineáris kereséssel megkeresi a paraméterként megkapott, növekvően rendezett tömbben a szintén paraméterként megkapott értéket, és az első olyan elem indexével tér vissza, amelynek értéke a kapott érték! Ha a keresett érték nincs a tömbben, akkor* -1 *-et kell visszaadni.*

```
int linearis( int tomb[], int meret, int ertek )
{
  int i;
  for ( i = 0; i < meret && tomb[ i ] < ertek; ++i )
    ;
  return i < meret && tomb[ i ] == ertek ? i : -1;
}
```

5.31. FELADAT. *Írjunk függvényt, amely bináris kereséssel megkeresi a paraméterként megkapott, növekvően rendezett tömbben a szintén paraméterként megkapott értéket, és egy olyan elem indexével tér vissza, amelynek értéke a kapott érték! Ha a keresett érték nincs a tömbben, akkor* -1 *-et kell visszaadni.*

1.

```
int binaris( int tomb[], int meret, int ertek )
{
  int also = 0, felso = meret - 1;
  while ( also <= felso )
  {
    int kozepso = ( also + felso ) / 2;
```

```
    if ( tomb[ kozepso ] == ertek )
      return kozepso;
    if ( tomb[ kozepso ] > ertek )
      felso = kozepso - 1;
    else
      also = kozepso + 1;
  }
  return -1;
}
```

2.

```
int binaris_rek( int tomb[], int also, int felso, int ertek )
{
  if ( also <= felso )
  {
    int kozepso = ( also + felso ) / 2;
    if ( tomb[ kozepso ] == ertek )
      return kozepso;
    return tomb[ kozepso ] > ertek ?
           binaris_rek( tomb, also, kozepso - 1, ertek ) :
           binaris_rek( tomb, kozepso + 1, felso, ertek );
  }
  return -1;
}
```

Fák

Az alfejezetben használt FAELEM típus definíciója a következő:

```
typedef struct faelem {
  int adat;
  struct faelem *bal, *jobb;
} FAELEM;
```

5.32. FELADAT. *Írjunk függvényt, amely meghatározza egy bináris fa elemszámát!*

```
int elemszam( FAELEM *fa )
{
  return fa ? 1 + elemszam( fa->bal ) + elemszam( fa->jobb ) : 0;
}
```

5.33. FELADAT. *Írjunk függvényt, amely meghatározza egy bináris fa magasságát!*

```
#define MAX( a, b )   ( ( a ) > ( b ) ? ( a ) : ( b ) )

int magassag( FAELEM *fa )
{
  return fa ? 1 + MAX( magassag( fa->bal ), magassag( fa->jobb ) ) : 0;
}
```

5.34. FELADAT. *Írjunk függvényt, amely meghatározza egy bináris fa levélelemeinek számát!*

```
int levelszam( FAELEM *fa )
{
  if ( !fa )
    return 0;
  else if ( !fa->bal && !fa->jobb )
    return 1;
  else
    return levelszam( fa->bal ) + levelszam( fa->jobb );
```

```
}
```

5.35. FELADAT. *Írjunk függvényt, amely meghatározza egy bináris fában azon elemek számát, amelyeknek nincs jobb oldali rákövetkezőjük!*

```
int nincs_jobb( FAELEM *fa )
{
  if ( !fa )
    return 0;
  else if ( !fa->jobb )
    return 1 + nincs_jobb( fa->bal );
  else
    return nincs_jobb( fa->bal ) + nincs_jobb( fa->jobb );
}
```

5.36. FELADAT. *Írjunk függvényt, amely meghatározza egy bináris fában azon elemek számát, amelyeknek két rákövetkezőjük van!*

```
int ket_kov( FAELEM *fa )
{
  if ( !fa )
    return 0;
  else if ( fa->bal && fa->jobb )
    return 1 + ket_kov( fa->bal ) + ket_kov( fa->jobb );
  else
    return ket_kov( fa->bal ) + ket_kov( fa->jobb );
}
```

5.37. FELADAT. *Írjunk függvényt, amely egy bináris fáról eldönti, hogy szimmetrikus-e!*

```
#define HAMIS 0
#define IGAZ  ( !HAMIS )

int szimmetrikus( FAELEM *fa )
{
  if ( !fa )
    return IGAZ;
  else if ( !fa->bal && !fa->jobb)
    return IGAZ;
  else if ( !fa->bal || !fa->jobb)
    return HAMIS;
  else
    return fa->bal->adat == fa->jobb->adat &&
           szimmetrikus( fa->bal ) && szimmetrikus( fa->jobb );
}
```

5.38. FELADAT. *Írjunk eljárást, amely helyben tükröz egy bináris fát!*

```
void helyben_tukroz( FAELEM *fa )
{
  if ( fa )
  {
    FAELEM *seged = fa->bal;
    fa->bal = fa->jobb;
    fa->jobb = seged;
    helyben_tukroz( fa->bal );
    helyben_tukroz( fa->jobb );
  }
}
```

5.39. FELADAT. *Írjunk függvényt, amely tükrözi a paraméterként megkapott bináris fát, és*

visszaadja az új fát!

```
#include <stdlib.h>

FAELEM *tukroz( FAELEM *fa )
{
  if ( !fa )
    return NULL;
  else
  {
    FAELEM *uj = ( FAELEM * )malloc( sizeof( FAELEM ) );
    uj->adat = fa->adat;
    uj->bal = tukroz( fa->jobb );
    uj->jobb = tukroz( fa->bal );
    return uj;
  }
}
```

5.40. FELADAT. *Írjunk függvényt, amely két bináris fáról eldönti, hogy megegyeznek-e (elemről elemre azonosak-e)!*

```
#define HAMIS 0
#define IGAZ  ( !HAMIS )

int megegyezik( FAELEM *egyik, FAELEM *masik )
{
  if ( !egyik && !masik)
    return IGAZ;
  else if ( !egyik || !masik)
    return HAMIS;
  else
    return egyik->adat == masik->adat &&
           megegyezik( egyik->bal, masik->bal ) &&
           megegyezik( egyik->jobb, masik->jobb );
}
```

5.41. FELADAT. *Írjunk függvényt, amely paraméterként egy nem üres bináris fa gyökérmutatóját kapja meg, és visszaadja, hogy mennyi a legalacsonyabb szintszámú, bal oldali rákövetkezővel nem rendelkező elem szintszáma! A gyökér szintszáma 0.*

```
#define MIN( a, b )   ( ( a ) < ( b ) ? ( a ) : ( b ) )

int szintszam( FAELEM *fa )
{
  if ( !fa->bal )
    return 0;
  else if ( !fa->jobb )
    return 1 + szintszam( fa->bal );
  else
    return 1 + MIN( szintszam( fa->bal ), szintszam( fa->jobb ) );
}
```

5.42. FELADAT. *Írjunk logikai függvényt, amely egy bináris fáról eldönti, hogy kiegyensúlyozott-e! Használjuk a 4.33. feladatban definiált* `magassag` *függvényt!*

```
#include <stdlib.h>

#define HAMIS 0
#define IGAZ  ( !HAMIS )

int kiegyensulyozott( FAELEM *fa )
{
```

```
  return fa ? kiegyensulyozott( fa->bal ) &&
         kiegyensulyozott( fa->jobb ) &&
         abs( magassag( fa->bal ) - magassag( fa->jobb ) ) <= 1 : IGAZ;
}
```

5.43. FELADAT. *Írjunk logikai függvényt, amely egy bináris fáról eldönti, hogy tökéletesen kiegyensúlyozott-e! Használjuk a 4.32. feladatban definiált* `elemszam` *függvényt!*

```
#include <stdlib.h>

#define HAMIS 0
#define IGAZ  ( !HAMIS )

int tok_kiegy( FAELEM *fa )
{
  return fa ? tok_kiegy( fa->bal ) && tok_kiegy( fa->jobb ) &&
         abs( elemszam( fa->bal ) - elemszam( fa->jobb ) ) <= 1 : IGAZ;
}
```

5.44. FELADAT. *Írjunk eljárást, amely*

1. preorder

2. inorder

3. postorder

módon bejár egy bináris fát, és a bejárás sorrendjében kiírja a képernyőre a csomópontokban tárolt adatokat!

1.

```
#include <stdio.h>

void preorder( FAELEM *fa )
{
  if ( fa )
  {
    printf( "%d\n", fa->adat );
    preorder( fa->bal );
    preorder( fa->jobb );
  }
}
```

2.

```
#include <stdio.h>

void inorder( FAELEM *fa )
{
  if ( fa )
  {
    inorder( fa->bal );
    printf( "%d\n", fa->adat );
    inorder( fa->jobb );
  }
}
```

3.

```
#include <stdio.h>

void postorder( FAELEM *fa )
```

```
{
  if ( fa )
  {
    postorder( fa->bal );
    postorder( fa->jobb );
    printf( "%d\n", fa->adat );
  }
}
```

5.45. FELADAT. *Adva van egy bináris fa folytonos reprezentációja három egydimenziós tömbben (`adat`, `bal`, `jobb`). Az első elem a gyökérelem. Ha egy elemnek nem létezik bal vagy jobb oldali rákövetkezője, akkor a megfelelő tömbben az elemhez tartozó érték 0. Írjunk függvényt, amely paraméterként megkapja a fa elemeinek számát és a három tömböt! A függvény állítsa elő a fa szétszórt reprezentációját, és adja vissza a fa gyökérmutatóját!*

```
#include <stdlib.h>

FAELEM *felepit( int index, int adat[], int bal[], int jobb[] )
{
  FAELEM *uj = ( FAELEM * )malloc( sizeof( FAELEM ) );
  uj->adat = adat[ index ];

  uj->bal =
    bal[ index ] ? felepit( bal[ index ], adat, bal, jobb ) : NULL;
  uj->jobb =
    jobb[ index ] ? felepit( jobb[ index ], adat, bal, jobb ) : NULL;
  return uj;
}

FAELEM *szetszor( int elemszam, int adat[], int bal[], int jobb[] )
{
  return elemszam == 0 ? NULL : felepit( 0, adat, bal, jobb );
}
```

5.46. FELADAT. *Írjunk függvényt, amely paraméterként megkap egy olyan nem bináris fát, amelyben minden elemnek legfeljebb 5 gyereke lehet, és visszaadja a fa bináris megfelelőjét!*

```
#include <stdlib.h>

#define N 5

typedef struct binfa {
  int ertek;
  struct binfa *bal, *jobb;
} BINELEM;

typedef struct nembinfa {
  int ertek;
  struct nembinfa *gyerek[ N ];
} NEMBINELEM;

void beilleszt( BINELEM *binelem, BINELEM *reszfa )
{
  if ( binelem->bal )
  {
    BINELEM *seged = binelem->bal;
    while ( seged->jobb )
      seged = seged->jobb;
    seged->jobb = reszfa;
  }
  else
```

```
    binelem->bal = reszfa;
}

BINELEM *binarizal( NEMBINELEM *nembinfa )
{
  BINELEM *binfa;
  int i;
  if ( !nembinfa )
    return NULL;
  binfa = ( BINELEM* )malloc( sizeof( BINELEM ) );
  binfa->ertek = nembinfa->ertek;
  binfa->bal = binfa->jobb = NULL;
  for ( i = 0; i < N; ++i )
    beilleszt( binfa, binarizal( nembinfa->gyerek[ i ] ) );
  return binfa;
}
```

5.47. FELADAT. *Írjunk programot, amely a billentyűzetről bekér egy szabályos postfix kifejezést, majd kiírja a képernyőre a kifejezés prefix alakját! A kifejezésben az operandusokat és az operátorokat fehér karakterek választják el egymástól. Operátor csak a* `+`*, a* `-`*, a* `*` *és a* `/` *karakterek egyike lehet. Minden operátor kétoperandusú.*

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

typedef struct faelem {
  char *op;
  struct faelem *bal, *jobb;
} FAELEM;

typedef struct veremelem {
  FAELEM *elem;
  struct veremelem *kov;
} VEREMELEM;

VEREMELEM *verem;

void push( FAELEM *elem )
{
  VEREMELEM *ujelem = ( VEREMELEM * )malloc( sizeof( VEREMELEM ) );
  ujelem->elem = elem;
  ujelem->kov = verem;
  verem = ujelem;
}

FAELEM *pop()
{
  FAELEM *seged = verem->elem;
  VEREMELEM *torlendo = verem;
  verem = verem->kov;
  free( torlendo );
  return seged;
}

void preorder( FAELEM *gyoker )
{
  if ( gyoker )
  {
    printf( "%s ", gyoker->op );
    free( gyoker->op );
    preorder( gyoker->bal );
```

```
    preorder( gyoker->jobb );
    free( gyoker );
  }
}

main()
{
  char kif[ 256 ];
  puts( "Kérem a postfix kifejezést:" );
  while ( scanf( "%s", kif ) != EOF )
  {
    FAELEM *uj = ( FAELEM * )malloc( sizeof( FAELEM ) );
    uj->op = ( char * )malloc( ( strlen( kif ) + 1 ) * sizeof( char ) );
    strcpy( uj->op, kif );
    if ( strlen( kif ) == 1 && strchr( "+-*/", *kif ) )
    {
      uj->jobb = pop();
      uj->bal = pop();
      if ( !verem )
      {
        preorder( uj );
        break;
      }
    }
    else
      uj->bal = uj->jobb = NULL;
    push( uj );
  }
  putchar( '\n' );
}
```

5.48. FELADAT. *Írjunk eljárást, amely egy paraméterként megkapott, egészeket tartalmazó keresőfába beszúrja a szintén paraméterként megadott értéket! Ha az érték már szerepel a fában, az eljárás ne csináljon semmit!*

```
#include <stdlib.h>

void beszur( FAELEM **gym, int ertek )
{
  if( !*gym )
  {
    *gym = ( FAELEM * )malloc( sizeof( FAELEM ) );
    ( *gym )->adat = ertek;
    ( *gym )->bal = ( *gym )->jobb = NULL;
  }
  else if ( ( *gym )->adat != ertek )
    beszur( ( *gym )->adat > ertek ? &( *gym )->bal : &( *gym )->jobb,
            ertek );
}
```

5.49. FELADAT. *Írjunk eljárást, amely egy paraméterként megkapott, egészeket tartalmazó keresőfából kitöröl egy szintén paraméterként megadott értéket! Az eljárás írjon megfelelő hibaüzenetet a képernyőre, ha a törlés valamilyen okból nem hajtható végre!*

Egy elem törlésénél a következő lehetőségek fordulhatnak elő:

• A törlendő elem nincs benne a keresőfában. Ekkor hibaüzenetet kell a képernyőre írni.

• A törlendő elem levélelem, azaz nincs egyetlen rákövetkezője sem.

• A törlendő elemnek csak egy rákövetkezője van.

• A törlendő elemnek két rákövetkezője van.

A fentieket figyelembe véve a feladat rekurzívan és iteratívan is megoldható. Az alábbiakban három megoldást adunk meg.

1. Az első megoldás rekurzívan oldja meg a feladatot. Egy külön rekurzív eljárásban kezeltük benne azt az esetet, amikor a törlendő elemnek két rákövetkezője van.

```
#include <stdlib.h>

void torol_2rakov( FAELEM *torlendo, FAELEM **legjobb )
{
  if( ( *legjobb )->jobb )
    torol_2rakov( torlendo, &( *legjobb )->jobb );
  else
  {
    FAELEM *seged = *legjobb;
    torlendo->adat = ( *legjobb )->adat;
    *legjobb = ( *legjobb )->bal;
    free( seged );
  }
}

void torol( FAELEM **gym, int ertek )
{
  if( !*gym )
    fputs( "Nincs ilyen érték a fában!\n", stderr );
  else if( ( *gym )->adat != ertek )
    torol( ( *gym )->adat < ertek ? &( *gym )->jobb : &( *gym )->bal,
           ertek );
  else if( !( *gym )->bal || !( *gym )->jobb )
  {
    FAELEM *torlendo = *gym;
    *gym = ( *gym )->bal ? ( *gym )->bal : ( *gym )->jobb;
    free( torlendo );
  }
  else
    torol_2rakov( *gym, &( *gym )->bal );
}
```

2. Másodikként lássuk az iteratív megoldást:

```
#include <stdlib.h>

void torol( FAELEM **gym, int ertek )
{
  FAELEM *akt = *gym, *szulo = NULL;
  while ( akt != NULL && akt->adat != ertek )
  {
    szulo = akt;
    akt = akt->adat > ertek ? akt->bal : akt->jobb;
  }
  if ( !akt )
    fputs( "Nincs ilyen érték a fában!\n", stderr );
  else if ( !akt->bal || !akt->jobb )
  {
    if( !szulo )
      *gym = akt->bal ? akt->bal : akt->jobb;
    else if ( akt->adat < szulo->adat )
      szulo->bal = akt->bal ? akt->bal : akt->jobb;
    else
      szulo->jobb = akt->bal ? akt->bal : akt->jobb;
    free( akt );
```

```
  }
  else
  {
    FAELEM *seged;
    szulo = akt;
    for ( seged = akt->jobb; seged->bal; seged = seged->bal )
      szulo = seged;
    ( szulo != akt ? szulo->bal : szulo->jobb ) = seged->jobb;
    akt->adat = seged->adat;
    free( seged );
  }
}
```

3. Végül álljon itt egy olyan rekurzív megoldás, amely iteratív elemeket is tartalmaz (azoknál az elemeknél, amelyeknek két rákövetkezőjük is van):

```
#include <stdlib.h>

void torol( FAELEM **gym, int ertek )
{
  if( !*gym )
    fputs( "Nincs ilyen érték a fában!\n", stderr );
  else if( ( *gym )->adat != ertek )
    torol( ( *gym )->adat < ertek ? &( *gym )->jobb : &( *gym )->bal,
           ertek );
  else if( !( *gym )->bal || !( *gym )->jobb )
  {
    FAELEM *torlendo = *gym;
    *gym = ( *gym )->bal ? ( *gym )->bal : ( *gym )->jobb;
    free( torlendo );
  }
  else
  {
    FAELEM *seged = ( *gym )->jobb;
    while ( seged->bal )
      seged = seged->bal;
    ( *gym )->adat = seged->adat;
    torol( &( *gym )->jobb, seged->adat );
  }
}
```

6. fejezet - C-implementációk

Ebben a fejezetben a különböző C-fordítók eltérő működésére, és -- ehhez kapcsolódóan -- kezdő C-programozók által elkövetett tipikus hibákra mutatunk be néhány példát. A lista korántsem teljes, csak az általunk tapasztalt és legfontosabbnak vélt implementációfüggő eszközökre térünk ki. Most is feladatokat fogalmazunk meg, a helyes megoldás mellett azonban bővebb magyarázatot, valamint nem szabványos (néhol helytelen) megoldásokat is közlünk.

6.1. FELADAT. *Írjunk függvényt, amely paraméterként megkap egy tetszőleges méretű, valósakat tartalmazó kétdimenziós tömböt, és visszaad egy olyan egydimenziós tömböt, amely a sorok átlagát tartalmazza!*

```
double *soratlagok( double *t, int sor, int oszlop )
{
  double soratl[ sor ] = { 0 };  // a teljes tömböt nullázza
  int i;
  for ( i = 0; i < sor; ++i )
  {
    int j;
```

```
    for ( j = 0; j < oszlop; ++j )
      soratl[ i ] += t[ i * oszlop + j ];
    soratl[ i ] /= oszlop;
  }
  return soratl;
}
```

Ha a fenti kódot egy főprogrammal kiegészítjük, lefordítjuk, majd lefuttatjuk, a legtöbb esetben azt tapasztaljuk, hogy szabályosan lefut, és helyes eredményt ad. A kódban azonban van egy hiba, valamint két olyan eszközt is alkalmaztunk, amelyet az ANSI-szabvány nem tartalmaz. A hiba nem szintaktikai, hanem szemantikai, és a `return soratl;` utasítás okozza. Mint tudjuk, a C-ben egy tömb neve önállóan (szögletes zárójelek nélkül) a tömb első elemét címző mutatót (a tömb kezdőcímét) jelenti (ezért is adtunk meg `double *`-ot a függvény típusaként). Emiatt nem a tömböt adja vissza a függvény, hanem „csak" egy tárbeli címet, ahol a tömb elhelyezkedik. Mindez nem lenne probléma, ha a tömb globális lenne a függvényre nézve. Mivel azonban a `soratl` a `soratlagok()` függvény lokális változója, ezért abban a pillanatban megszűnik (azaz felszabadul az általa lefoglalt memória), amikor a függvény befejezi működését. Így tehát egy olyan mutatót adunk vissza, amely egy szabad memóriaterületre mutat. Szerencsés esetben a futtató rendszer nem írja felül ezt a memóriaterületet addig, amíg fel nem használjuk az ott tárolt adatokat, ezért fordulhat elő, hogy helyes eredményt tapasztalunk. Ha azonban a hívó függvényben memóriafoglalási műveleteket végzünk, mielőtt hivatkoznánk a visszaadott címre, nagy valószínűséggel már nem az általunk létrehozott tömböt találjuk ott.

Most pedig nézzük meg a nem szabványos eszközöket! Az egyik a `//` többkarakteres szimbólum, amely az ANSI-szabványban nem szerepel. Mivel a legtöbb maiC-fordító átvette a C++-ból ezt a szimbólumot az egysoros megjegyzések jelölésére, ezért le is fordítják ezt a kódot. A másik nem szabványos eszköz a

```
double soratl[ sor ] = { 0 };
```

utasításban van. Nem a kezdőértékadásról van szó, hiszen a szabvány kimondja, hogy ha kevesebb kezdőértéket adunk meg egy tömb deklarációjakor, mint ahány eleme van a tömbnek, akkor az első elemek a megadott értékekkel, a fennmaradó elemek pedig 0-val inicializálódnak. A tömb elemszámát viszont az ANSI-szabvány szerint csak konstanskifejezéssel (azaz fordítási időben kiértékelhető kifejezéssel) adhatjuk meg, márpedig a `sor` paraméter nyilván nem az.

Ezek után lássuk azt a kódot, amely teljesen szabványos és hibamentes:

```
#include <stdlib.h>

double *soratlagok( double *t, int sor, int oszlop )
{
  double *soratl = ( double * )calloc( sor, sizeof( double ) );
         /* a calloc() függvény nullázza a lefoglalt memóriaterületet */
  int i;
  for ( i = 0; i < sor; ++i )
  {
    int j;
    for ( j = 0; j < oszlop; ++j )
      soratl[ i ] += t[ i * oszlop + j ];
    soratl[ i ] /= oszlop;
  }
  return soratl;
}
```

A hibát tehát úgy küszöböltük ki, hogy nem lokális tömböt hoztunk létre a sorátlagok tárolására, hanem egy mutatót, és a `calloc()` függvény segítségével pontosan akkora területet foglaltunk, mint amennyit a tömb foglalna el. Azért ezt a függvényt választottuk a `malloc()` helyett, mert ez

mindjárt nullázza is a lefoglalt memóriaterületet. Ezzel megoldódott az a probléma is, hogy nem konstanskifejezéssel adtuk meg a tömb elemszámát. A megjegyzést pedig az ANSI-szabvány által is ismert `/*` és `*/` szimbólumok közé tettük.

6.2. FELADAT. *Írjunk függvényt, amely a paraméterként megkapott, egészeket tartalmazó egydimenziós tömb harmadik legnagyobb elemével tér vissza! Feltehetjük, hogy a tömbnek legalább 3 eleme van.*

```
#define N 3

int min_index( int t[], int meret )
{
  int i, legk = 0;
  for ( i = 1; i < meret; ++i )
    if ( t[ i ] < t[ legk ] )
      legk = i;
  return legk;
}

int harmadik( int t[], int meret )
{
  int i, max[ N ] = { t[ 0 ], t[ 1 ], t[ 2 ] };
  for ( i = N; i < meret; ++i )
  {
    int legk = min_index( max, N );
    if ( t[ i ] > max[ legk ] )
      max[ legk ] = t[ i ];
  }
  return max[ min_index( max, N ) ];
}
```

A megoldáshoz felvettünk egy háromelemű tömböt (`max`), ami a paraméterként kapott `t` tömb három legnagyobb elemét tartalmazza. Ezt úgy érjük el, hogy kezdetben `t` első három elemét tároljuk el, majd végigmegyünk a tömb további elemein, és mindig a `max` tömb legkisebb elemét írjuk felül `t` aktuális elemével, ha az nagyobb nála. A végén csak a `max` tömb legkisebb elemét kell visszaadni.

A feladatnak természetesen nem ez a legegyszerűbb megoldása, viszont jól szemlélteti, hogyan nem szabad kezdőértéket adni egy tömbnek. Az ANSI-szabvány ugyanis előírja, hogy egy tömb kezdőértékei csak konstanskifejezések lehetnek. A legtöbb mai fordító azonban lehetővé teszi, hogy egy tömbelem kezdőértéke a fenti példában láthatóhoz hasonló, futási időben kiértékelt kifejezés legyen. A függvény tehát úgy lesz szabványos, ha az

```
int i, max[ N ] = { t[ 0 ], t[ 1 ], t[ 2 ] };
```

sort kicseréljük a következő kódrészletre:

```
int i, max[ N ];
for ( i = 0; i < N; ++i )
  max[ i ] = t[ i ];
```

6.3. FELADAT. *Írjunk programot, amely képernyőre írja az első 10 természetes számot, valamint négyzetüket!*

```
#include <stdio.h>

main()
{
  int i = 1;
  while ( i <= 10 )
    printf( "%3d  %3d\n", i, i * i++ );
```

```
}
```

A legtöbb fordító figyelmeztetést is küld a fenti kód fordítása közben, mely szerint definiálatlan lesz az `i`-t tartalmazó kifejezések értéke. A probléma abban rejlik, hogy az ANSI-szabvány nem határozza meg a paraméterek kiértékelési sorrendjét. A legtöbb esetben ennek nincs is jelentősége, csak akkor, ha a paraméterek nem függetlenek, és valamelyikük mellékhatással rendelkező operátort tartalmaz. A példában feltételeztük, hogy a paraméterek balról jobbra fognak kiértékelődni. A legtöbb implementáció azonban nem ezt vallja, hanem -- a változó paraméterszámú függvények támogatása céljából -- a paramétereket jobbról balra értékelik ki és teszik a verembe. Így az eredmény is hibás lesz:

```
  2    1
  3    4
  4    9
  5   16
  6   25
  7   36
  8   49
  9   64
 10   81
 11  100
```

Más fordító használata esetén azonban előfordulhat az is, hogy helyes eredményt kapunk. Hogy a megoldás ne legyen implementációfüggő, az `i` változó növelését a kiírás után kell elvégezni, például így:

```
#include <stdio.h>

main()
{
  int i = 1;
  while ( i <= 10 )
  {
    printf( "%3d  %3d\n", i, i * i );
    ++i;
  }
}
```

Szebb és tömörebb kódot kapunk, ha `for` ciklust alkalmazunk:

```
#include <stdio.h>

main()
{
  int i;
  for ( i = 1; i <= 10; ++i )
    printf( "%3d  %3d\n", i, i * i );
}
```

6.4. FELADAT. *Írjunk programot, amely pozitív egész számokat olvas be a billentyűzetről a 0 végjelig, és minden beolvasott szám esetén képernyőre írja, hogy az prímszám-e!*

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

#define HAMIS 0

main()
{
```

```
  int prim( unsigned szam )
  {
    int oszto;
    for ( oszto = 2; oszto <= sqrt( szam ); ++oszto )
      if ( szam % oszto == 0 )
        return HAMIS;
    return szam != 1;
  }

  puts( "Kérem a számokat!" );

  unsigned szam;

  for ( ; ; )
  {
    scanf( "%u", &szam );
    if ( !szam )
      break;
    puts( prim( szam ) ? "Prím." : "Nem prím." );
  }

}
```

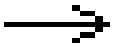
Egyes fordítók minden további nélkül lefordítják ezt a kódot, holott két helyen is ellentmond az ANSI-szabványnak. A szabvány szerint ugyanis függvényt definiálni csak külső szinten (az állomány szintjén) lehet, márpedig mi a `prim()` függvény törzsét a `main()` függvény belsejében adtuk meg. Másrészt deklarációs utasítás csak az állomány szintjén, valamint a blokkok „elején" (a blokk első végrehajtható utasítása előtt) állhat, a `szam` változó deklarációja viszont közvetlenül egy végrehajtható utasítás után áll.

Mindezek alapján egy szabványos megoldás a következő lehet:

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

#define HAMIS 0

int prim( unsigned szam )
{
  int oszto;
  for ( oszto = 2; oszto <= sqrt( szam ); ++oszto )
    if ( szam % oszto == 0 )
      return HAMIS;
  return szam != 1;
}

main()
{
  unsigned szam;
  puts( "Kérem a számokat!" );
  for ( ; ; )
  {
    scanf( "%u", &szam );
    if ( !szam )
      break;
    puts( prim( szam ) ? "Prím." : "Nem prím." );
  }
}
```

6.5. FELADAT. *Írjunk függvényt, amely a paraméterként kapott, valós számokat tartalmazó egydimenziós tömb elemeiről eldönti, hogy monoton növekvő számsorozatot alkotnak-e!*

```
#define HAMIS 0
#define IGAZ  ( !HAMIS )

int mon_nov( double t[], int meret )
{
  int i;
  for ( i = 0; i < meret; ++i )
    if ( t[ i ] < t[ i - 1 ] )
      return HAMIS;
  return IGAZ;
}
```

A kezdő programozók egyik tipikus hibáját fedezhetjük fel a fenti kódban: az ember hajlamos mindig ugyanúgy „végigmenni" egy tömbön, nem törődve azzal, hogy így nem létező tömbelemekre hivatkozik. Jelen esetben például `i = 0` esetén `t[ -1 ]`-re hivatkozunk az összehasonlítás jobb oldalán. A fordító számára ez nem hiba, mivel nem végez indexhatár-ellenőrzést. Ennek az az oka, hogy minden `[]` operátort * operátorrá alakít a következő szabály szerint: `t[ i ]`  `*( t + i )`. Ez a mutató pedig bárhová mutathat a memóriában, nem szükséges, hogy a tömb egy elemét címezze. Így tehát a fentivel egyenértékű kódot kapunk, ha kicseréljük az

```
if ( t[ i ] < t[ i - 1 ] )
```

sort a következőre:

```
if ( *( t + i ) < *( t + i - 1 ) )
```

Szintén egyenértékű kódot kapunk, ha nem egy indexszel, hanem egy mutatóval megyünk végig a tömbön:

```
#define HAMIS 0
#define IGAZ  ( !HAMIS )

int mon_nov( double t[], int meret )
{
  double *p;
  for ( p = t; p < t + meret; ++p )
    if ( *p < *( p - 1 ) )
      return HAMIS;
  return IGAZ;
}
```

Ez a kód tehát ugyanúgy hibás, mint az előző. Az ANSI-szabvány szerint a `p` mutató értéke definiálatlan, ha `p < t` vagy `p > t + meret`. Csúnyán fogalmazva ez azt jelenti, hogy `p` mutathat a tömb „utolsó utáni" elemére is, ebben az esetben még garantált, hogy a tömb utolsó eleme által elfoglalt tárhelyet követő memóriaterületre mutat. Ha azonban `p = t - 1`, akkor nem biztos, hogy `p` a tömb első eleme előtti tárhelyet címzi.

Persze a konkrét feladat esetén nem lenne jó a következő megoldás sem:

```
#define HAMIS 0
#define IGAZ  ( !HAMIS )

int mon_nov( double t[], int meret )
{
  int i;
  for ( i = 0; i < meret; ++i )
    if ( t[ i + 1 ] < t[ i ] )
      return HAMIS;
```

```
  return IGAZ;
}
```

Most ugyan nem címzünk olyan területet, ami kívül esne a fent megadott tartományon (azaz a kód szabványos), viszont továbbra sem tudjuk, mi van a memóriában az utolsó tömbelem után. A helyes megoldás természetesen a következő:

```
#define HAMIS 0
#define IGAZ  ( !HAMIS )

int mon_nov( double t[], int meret )
{
  int i;
  for ( i = 1; i < meret; ++i )
    if ( t[ i ] < t[ i - 1 ] )
      return HAMIS;
  return IGAZ;
}
```

6.6. FELADAT. *Írjunk programot, amely képernyőre írja egy kétbájtos, fixpontosan ábrázolt egész érték felső (alacsonyabb memóriacímen lévő) és alsó (magasabb memóriacímen lévő) bájtját!*

• Első megoldásként két mutatót használunk, az egyikkel a felső, a másikkal az alsó bájtot címezzük:

```
#include <stdio.h>

main()
{
  short int ertek = 1;
  char *felso = ( char * )&ertek, *also = ( char * )&ertek + 1;
  printf( "felső: %3d, alsó: %3d\n", *felso, *also );
}
```

• A következő megoldásban egy olyan struktúra típusra „húzzuk rá" az értéket explicit típuskonverzióval, amelynek a két mezője az egész érték felső, illetve alsó bájtjára „illeszkedik":

```
#include <stdio.h>

main() {
  short int ertek = 1;
  struct stip
  {
    char felso, also;
  } *mut = ( struct stip * )&ertek;
  printf( "felső: %3d, alsó: %3d\n", mut->felso, mut->also );
}
```

• Végül ugyanezt csináljuk típuskonverzió nélkül, a `union` típus segítségével:

```
#include <stdio.h>

main()
{
  union utip
  {
    short int ertek;
    char bajt[ 2 ];
  } u;
  u.ertek = 1;
  printf( "felső: %3d, alsó: %3d\n", u.bajt[ 0 ], u.bajt[ 1 ] );
```

```
}
```

Mindhárom megoldás ugyanazt az eredményt adja, viszont hogy mi is ez az eredmény, az platformfüggő. Intel-kompatibilis processzorokkal ellátott gépeken lefuttatva a fenti programokat, az alsó bájt 0 lesz, a felső pedig 1. Ennek oka az úgynevezett bájtfordított tárolási mód, azaz a tárban előrébb helyezkedik el az alacsonyabb helyiértékű bájt, mint a magasabb helyiértékű. Ezt a tárolási módot angolul *little-endian* architektúrának nevezik (ilyen volt például a PDP-11 is, amelyen a C nyelvet kifejlesztették, de ilyenek a VAX-család gépei is). A *big-endian* architektúrájú gépeken ezzel szemben éppen fordított eredményt kapunk, azaz az alsó bájt lesz 1, a felső pedig 0, mert ezek a gépek a nagyobb helyiértékű bájtot tárolják a kisebb memóriacímen (a rendszerek többsége -- például az IBM 370-család, a PDP-10, a Motorola-processzorok vagy a legtöbb RISC architektúrájú gép is -- ide sorolható).

6.7. FELADAT. *Az 4.28. feladat megoldásai közül joggal hiányolhatja az olvasó azt a megoldást, amely az* `stdlib.h`*-ban deklarált* `qsort()` *könyvtári függvényt használja a tömbelemek rendezéséhez. Lássuk most ezt:*

```
#include <stdlib.h>

int hasonlit( const void *egyik, const void *masik )
{
  return *( int * )egyik - *( int * )masik;
}

void gyors3( int tomb[], int bal, int jobb )
{
  qsort( tomb + bal, jobb - bal + 1, sizeof( int ), hasonlit );
}
```

Ez a függvény azt csinálja, amit várunk tőle: növekvő sorrendbe rendezi a paraméterként megadott tömb elemeit. A feladat azonban túl egyszerű volt, nem tudtuk rajta bemutatni `qsort()` függvény egy érdekes implementációfüggő tulajdonságát.

Tegyük fel ugyanis, hogy a rendezendő elemeket csak valamely résztulajdonságuk alapján szeretnénk sorba rakni: esetünkben például előre azokat a számokat, amelyek 2-vel osztva 0 maradékot adnak (azaz párosak), utánuk azokat, amelyek 2-vel osztva 1 maradékot adnak (azaz páratlanok). Erre egy lehetséges megoldás a következő:

```
#include <stdlib.h>

int hasonlit( const void *egyik, const void *masik )
{
  return *( int * )egyik % 2 - *( int * )masik % 2;
}

void parosgyors( int tomb[], int bal, int jobb )
{
  qsort( tomb + bal, jobb - bal + 1, sizeof( int ), hasonlit );
}
```

Látható, hogy a könyvtári `qsort()` függvény első paramétere a rendezendő tömb memóriabeli kezdőcíme, második paramétere a rendezendő elemek darabszáma, harmadik paramétere a rendezendő tömb egy elemének mérete bájtokban számolva, míg a negyedik paraméter egy mutató egy olyan függvényre, amely a tömb két elemét tudja összehasonlítani.

A „hasonlító" függvényt a `qsort()` függvény fogja majd időnként meghívni, az elemek összehasonlítási sorrendje azonban implementációfüggő. Ez a hasonlító függvény két mutatót kap paraméterül: az összehasonlítandó elemek címeit. A programozónak csak annyi a dolga, hogy az összehasonlítás „algoritmusát" kódolja le benne. A mi esetünkben ez két egész típusú érték

összehasonlítását jelenti. A függvénynek negatív értéket kell visszaadnia, ha az első összehasonlítandó elem kisebb a másodiknál,  értéket, ha a két elem egyenlő, és pozitív értéket, ha az első összehasonlítandó elem nagyobb a másodiknál.

Írjuk meg a `main()` függvényt ehhez a két függvényhez a következőképpen:

```
#include <stdio.h>

main()
{
  int t[] = { 4, 9, 5, 7, 2, 3, 8, 10, 1, 6 };
  int i, darab = sizeof( t ) / sizeof( int );
  parosgyors( t, 0, darab - 1 );
  for ( i = 0; i < darab; ++i )
    printf( "%d ", t[ i ] );
  putchar( '\n' );
}
```

Ha mindennel elkészültünk, fordítsuk le a forrásállományt különböző fordítóprogramokkal, és futtassuk a lefordított programokat! Mi a következőt tapasztaltuk:

• Windows XP operációs rendszerre telepített Dev-C++ fordítóprogrammal (4.9.8.9-es verzió) a futási eredmény:

```
4 10 8 6 2 5 9 1 3 7
```

• Linux operációs rendszerre (Mandrake 9.1-es disztribúció) telepített 3.2.2-es `gcc` fordítóval a futási eredmény:

```
4 2 8 10 6 9 5 7 3 1
```

• AIX 4.3.3-as operációs rendszerre telepített 3.0.1-es `gcc` fordítóval a futási eredmény:

```
2 4 6 10 8 3 7 5 9 1
```

A fenti példák bizonyítják, hogy a könyvtári `qsort()` függvény használata nem garantálja a rendezés szempontjából azonos tulajdonságú elemek eredeti sorrendjének megőrzését a rendezést követően. Ezt csak a Linux operációs rendszeren működő 3.2.2-es `gcc` fordítóval fordított program esetében tapasztaltuk (a fenti példában).

A 3.2.2-es `gcc` fordítóval fordított, `qsort()` függvényhívást tartalmazó programok hasonlóan működtek olyan, általunk ismeretlen bemeneti adatokat felhasználó feladatok megoldásában is, ahol pedig az egyező tulajdonságú adatok eredeti sorrendjének megőrzése a helyes megoldás egyik feltétele volt (lásd a 13.5 alfejezetben ismertetett feladatot). Az említett feladatra adott megoldásaink tesztelése során azt tapasztaltuk, hogy a nemzetközi online zsűri elfogadott olyan -- elvileg hibás -- programokat is, ahol a helyes megoldást implementációfüggő rendezéssel (konkrétan a `qsort()` függvény használatával) sikerült előállítani. Azonban félreértés ne essék: egy-két példa, amely alátámasztja a 3.2.2-es `gcc` fordítónak ezt a tulajdonságát, még nem erősíthet meg bennünket abban a hitben, hogy ez minden esetben így működik. Sőt! Sokkal gyanakvóbbá kell tegyen bennünket, hiszen az így megírt programjaink majdnem biztosan nem a várt eredményt szolgáltatják majd más platformokon.

`6.8. FELADAT.` *Utolsóként vegyük szemügyre a 1.24. feladatot, ahol a megoldásainkban felhasználtuk az* `stdlib.h` *header állományban deklarált* `srand()` *és* `rand()` *függvényeket.*

Mindkét megoldásban, amit bemutattunk, az `srand()` függvényhívás a `main()` függvény első végrehajtható utasításában szerepel. Ezzel a függvényhívással mindkét esetben a véletlenszám-generátor kezdeti értékét állítottuk be egy, az aktuális rendszeridőtől függő értékre. Az ehhez

felhasznált `time()` függvény szintén szabványos könyvtári függvény (a `time.h` header állományban deklarálták), amely általában valamely rögzített időponttól -- például 1970. január 1-jétől -- eltelt időt adja meg másodpercekben számolva. Milyen problémák merülhetnek fel az itt alkalmazott megoldással kapcsolatban?

Nos, képzeljük el, hogy a lefordított programunkat sokszor szeretnénk lefuttatni egymás után. Amennyiben két egymást követő programindítás ugyanabban a másodpercben történik, a véletlenszám-generátor kezdeti értéke ugyanaz lesz mindkét esetben, hiszen a rögzített időponttól eltelt másodpercek száma is azonos mindkét esetben. Ez viszont azt eredményezi, hogy a programok futási eredménye is azonos lesz.

Ami viszont érdekesebb ennél, hogy különböző implementációk különbözőképpen végezhetik a véletlenszám-generátor kezdeti értékének beállítását az idő függvényében. Mire is gondolunk konkrétan? Nyilvánvaló, hogy ha nem ugyanabban a másodpercben futtatjuk le a programunkat többször egymás után, hanem mondjuk ennél valamivel (de nem túl sokkal) nagyobb időközönként, akkor várhatóan az eredményeknek is különbözniük kell legalább néhány értékben. Tulajdonképpen ez így is van, tapasztalataink ezt támasztják alá. Azonban az eltérés mértéke igencsak különböző az egyes implementációk esetén. A kipróbált fordítóprogramok közül legkevésbé a Windows XP operációs rendszerre telepített Dev-C++ fordítóprogrammal (4.9.8.9-es verzió) voltunk elégedettek ilyen szempontból: az ezzel fordított programok eredményeiben a számötösök első számjegye és az eltelt (másod)percek száma között erős kapcsolatot véltünk felfedezni.

7. fejezet - Matematikai feladatok

Tartalom

Pi

Az angliai Birminghamben, az Aston Egyetem Alkalmazott Matematika és Számítástudományi Tanszékén dolgozó Robert A. J. Matthews professzor nemrégiben leírta, hogyan használható fel az éjszakai égbolt csillagainak helyzete a π értékének meglepően pontos meghatározásához. Ez az eredmény a számelmélet bizonyos tételeinek alkalmazásából következett.

Nekünk ugyan nem áll rendelkezésünkre az éjszakai égbolt, de felhasználhatjuk ugyanezt az elméleti alapot a π egy becslésének meghatározására:

• Adva van egész számoknak egy nagy, véletlenszerű gyűjteménye. Ebből tetszőlegesen kiválasztva két számot, $6/\pi^2$ annak a valószínűsége, hogy a két számnak a legnagyobb közös osztója 1.

A 2, 3, 4, 5, 6 számok *kis* gyűjteményét használva például 10 számpár állítható elő: $(2,3)$, $(2,4)$ stb. Ebből a 10 párból a következő hatnak a legnagyobb közös osztója 1: $(2,3)$,

$(2,5)$, $(3,4)$, $(3,5)$, $(4,5)$ és $(5,6)$. Ezen darabszámok arányát mint valószínűséget felhasználva, a következőt kapjuk:

$6/\pi^2 \approx 6/10$

$\pi \approx 3.162$

Ebben a feladatban adathalmazok egy sorozatát kell feldolgoznod. Minden adathalmaz pszeudovéletlen pozitív egész számoknak egy halmazát tartalmazza. Az egyes adathalmazok esetén keresd meg azokat a számpárokat, amelyeknek a legnagyobb közös osztója 1, a fent ismertetett módszer felhasználásával adj egy becslést a π értékére, majd írasd ki ezt a becslést.

Input

A bemenet adathalmazok sorozatából áll.

Minden adathalmaz első sora egy N pozitív egész értéket tartalmaz, amely nagyobb 1-nél és kisebb 50-nél.

A következő N sor mindegyikében egy darab pozitív egész szám szerepel. Ezek alkotják azt a halmazt, amelyben a számpárokat kell vizsgálni. Ezek az egész számok nagyobbak 0-nál és kisebbek 32768-nál.

A bemeneti adatfolyamban szereplő számok első számjegye a sorok első karaktere.

Az adatok végét a halmaz méretét megadó N szám nulla értéke jelzi.

Output

Minden feldolgozott bemeneti adathalmaz esetén egy sort kell kiírnod, amely egyetlen valós értéket tartalmaz. Ez az érték a π-nek az adott adathalmaz alapján kapott becslése. A példában látható formátumot kell használni. A válaszokat a tizedespont után hat számjegyre kell kerekíteni.

Néhány adathalmaz esetén nem lehetséges becslést adni a π értékére. Ez akkor fordul elő, ha nincs egyetlen olyan számpár sem, amelynek 1 a legnagyobb közös osztója. Ezekben az esetekben a következő egysoros üzenetet írd ki:

```
Nincs becslés erre az adathalmazra.
```

A sor első karaktere legyen az üzenet első karaktere, az `N`.

Példa input

```
5
2
3
4
5
6
2
13
39
```

```
0
```

Példa output

```
3.162278
Nincs becslés erre az adathalmazra.
```

Megoldás

Az egyes adathalmazok feldolgozásakor a beolvasott elemek mindegyikét párba állítjuk az eddig beolvasottakkal, és ha a legnagyobb közös osztójuk 1, akkor megnövelünk egy számlálóértéket, amely az adathalmazban lévő relatív prímpárokat számolja.

Amennyiben az adathalmaz feldolgozása végén a számláló értéke nulla, akkor nem tudunk becslést adni a  értékére, egyébként pedig a megadott képletből a  -t kifejezve kiírjuk a becslést.

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

#define MAXN 50

int n, tomb[ MAXN ], i, j, szamlalo;
char sor[ 10 ];

int lnko( int a, int b )
{
  int maradek;
  while ( maradek = a % b )
  {
    a = b;
    b = maradek;
  }
  return b;
}

int main()
{
  gets( sor );
  while ( n = atoi( sor ) )
  {
    szamlalo = 0;
    for ( i = 0; i < n; ++i )
    {
      gets( sor );
      tomb[ i ] = atoi( sor );
      for ( j = 0; j < i; ++j )
        if ( lnko( tomb[ i ], tomb[ j ] ) == 1 )
          ++szamlalo;
    }
    if ( szamlalo )
      printf( "%0.6lf\n", sqrt( 3.0 * n * ( n - 1 ) / szamlalo ) );
    else
      puts( "Nincs becslés erre az adathalmazra." );
    gets( sor );
  }
  return EXIT_SUCCESS;
}
```

Goldbach sejtése

• **Goldbach sejtése:** bármely páros $n \geq 4$ számhoz létezik legalább egy p_1, p_2 prímpár, amelyre $n = p_1 + p_2$.

Ezt a sejtést még nem bizonyították, de nem is cáfolták. Senki sem biztos abban, hogy ez a sejtés valóban igaz-e. Találhatunk viszont ilyen prímpárt bármely adott páros számhoz, ha létezik. A feladatod az, hogy írj egy olyan programot, amely megadja az összes olyan prímpár számát, amely kielégíti a sejtésben megfogalmazott feltételt egy adott páros szám esetén.

A bemenet páros számok sorozata. Minden egyes számhoz a programnak meg kell adnia a fent említett prímpárok számát. Vedd figyelembe, hogy a **különböző** párok számára vagyunk kíváncsiak, és a (p_1, p_2) és a (p_2, p_1) nem számítanak különböző pároknak.

Input

A bemenet minden sorában egy egész szám található. Feltételezheted, hogy minden szám páros, és nagyobb vagy egyenlő, mint 4, és kisebb, mint 2^{15}. A bemenet végét egy 0 szám jelöli, amit már nem kell feldolgoznod.

Output

A kimenet minden sorában egy egész számnak kell lennie. Semmilyen más karakter nem szerepelhet a kimeneten.

Példa input

```
6
10
12
0
```

Példa output

```
1
2
1
```

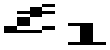
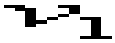
Megoldás

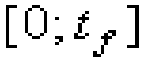
A feladat megoldásakor -- a gyorsabb működés érdekében -- előállítunk egy olyan tömböt, amely a megadott intervallumba eső számok esetén megadja, hogy azok prímek-e vagy sem. Ezt a tömböt felhasználva már könnyen össze tudjuk számolni azokat a számpárokat, amelyekben az értékek összege éppen a beolvasott páros szám.

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

int prim( long szam )
{
  long i;
  for ( i = 2; szam % i && i * i <= szam; ++i )
    ;
  return i * i > szam;
}

int main()
```

```
{
  int paros, i, darab;
  char tomb[ 32768 ];

  for ( i = 2; i < 32768; ++i )
    tomb[ i ] = prim( i );

  scanf( "%d", &paros );
  while ( paros ) {
    darab = 0;
    for ( i = 2; i <= paros / 2; ++i )
      if ( tomb[ i ] && tomb[ paros - i ] )
        ++darab;
    printf( "%d\n", darab );
    scanf( "%d", &paros );
  }

  return EXIT_SUCCESS;
}
```

Vonatok

Két város (T1 és T2) két sínpárral van összekötve. A T1 és T2 közötti távolság  méter. T1-ből T2-be a vonatok  másodpercenként, T2-ből T1-be pedig  másodpercenként indulnak. A T1-ből T2-be haladó vonatok sebessége  m/s, a T2-ből T1-be tartó vonatok sebessége pedig  m/s.

A feladatod az, hogy írj egy programot, amely kiszámolja, hogy hányszor találkoznak a vonatok a T1-et és T2-t összekötő vonalon a  másodperces időintervallum alatt.

Feltételezzük a következőket:

1. a 0 időpillanatban két vonat indul (egy T1-ből T2-be és egy T2-ből T1-be),

2. az input és output adatok egészek.

Input

A programod az adatokat a standard inputról olvassa. Adathalmazokat kell beolvasnod, soronként egyet, a következő formában:

Output

A program a standard outputra írja ki a találkozások számát, minden adathalmazra külön sorban.

Példa input

```
10 5 5 1 1 2
```

Példa output

```
6
```

Megoldás

A feladat megoldásához mindössze a fizikából jól ismert $v = s / t$ képlet ismeretére van szükség. A számítások gyorsítása és a pontos értékek megtartása érdekében egész aritmetikai műveleteket használtunk.

Első megoldásunkban összepárosítjuk az összes T1 városbeli indulási időpontot az összes T2 városbeli indulási időponttal. A T1-ből az  időpillanatban induló vonat akkor találkozik a T2-ből a  időpillanatban induló vonattal, ha

• az  időpontban induló és T2-be $i + d / v_1$ időpillanatban érkező vonat találkozhat a T2-ből a  időpillanatban induló vonattal, azaz

$$i + \frac{d}{v_1} \geq j,$$

amiből

$$d \geq v_1 \cdot (j - i);$$

• a  időpontban induló és T1-be $j + d / v_2$ időpillanatban érkező vonat találkozhat a T1-ből az  időpillanatban induló vonattal, azaz

$$j + \frac{d}{v_2} \geq i,$$

amiből

$$d \geq v_2 \cdot (i - j);$$

• a két vonat által a befejezési időig megtett utak hosszának összege nagyobb vagy egyenlő a T1 és T2 közötti távolságnál, azaz

$$v_1 \cdot (t_f - i) + v_2 \cdot (t_f - j) \geq d.$$

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

int main()
{
  int d, v1, v2, t1, t2, tf;
  int i, j, szamlalo;

  while ( scanf( "%d %d %d %d %d %d",
                 &d, &v1, &v2, &t1, &t2, &tf ) != EOF )
  {
    szamlalo = 0;
    for ( i = 0; i <= tf; i += t1 )
      for ( j = 0; j <= tf; j += t2 )
```

```
        if ( v1 * ( j - i ) <= d && v2 * ( i - j ) <= d &&
             v1 * ( tf - i ) + v2 * ( tf - j ) >= d )
          ++szamlalo;
    printf( "%d\n", szamlalo );
  }

  return EXIT_SUCCESS;
}
```

Az egymásba ágyazott `for` ciklusokban sok olyan vonatpárt is megvizsgálunk, amelyek eleve nem találkozhatnak (az egyik vonat túl korán érkezik a másik vonat indulásához képest, vagy túl későn indul a másik vonat érkezéséhez képest). Ezeket a vizsgálatokat kiküszöbölhetjük úgy, hogy csak az egyik városból induló vonatokat vesszük sorra, és mindegyikre meghatározzuk azt a legkorábbi és legkésőbbi indulási időpontot, amikor a másik városból induló vonatok találkozhatnak ezzel a vonattal.

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

#define min( a, b )   ( ( ( a ) < ( b ) ) ? ( a ) : ( b ) )

int d, v1, v2, t1, t2, tf;
int i, j, szamlalo, seged;
int t2v2, t2v1v2, tfv1, tfv2, v1v1, v1v2, dv1, dv2, tfv1_v1v2;
int jmin, jmax;

int main()
{
  while ( scanf( "%d %d %d %d %d %d",
                 &d, &v1, &v2, &t1, &t2, &tf ) != EOF )
  {
    if ( t1 < t2 )
    {
      seged = t1;
      t1 = t2;
      t2 = seged;
      seged = v1;
      v1 = v2;
      v2 = seged;
    }
    t2v2 = t2 * v2;
    t2v1v2 = t2 * v1 * v2;
    tfv1 = tf * v1;
    tfv2 = tf * v2;
    v1v1 = v1 * v1;
    v1v2 = v1 * v2;
    dv1 = d * v1;
    dv2 = d * v2;
    tfv1_v1v2 = tf * v1 * v1 + tf * v1 * v2;
    szamlalo = 0;
    for ( i = 0; i <= tf; i += t1 )
    {
      jmin = 0;
      if ( i * v2 > d )
        jmin = i * v2 - d;

      seged = jmin / t2v2;
      jmin = jmin % t2v2 ? seged + 1 : seged;

      jmax = min( i * v1v2 + dv2, tfv1_v1v2 - i * v1v1 - dv1 );
      jmax /= t2v1v2;
```

```
      if ( jmax >= jmin )
        szamlalo += jmax - jmin + 1;
    }
    printf( "%d\n", szamlalo );
  }

  return EXIT_SUCCESS;
}
```

Egyiptomi törtek

Az ókori Egyiptomban a 0 és 1 közötti racionális számokat egységtörtek összegeként,

$$\frac{1}{x_1}+\frac{1}{x_2}+\ldots+\frac{1}{x_k}$$

alakban adták meg, ahol az x_i -k különböző pozitív egész számok. Például

$$\frac{2}{5}=\frac{1}{3}+\frac{1}{15} \qquad \frac{9}{11}=\frac{1}{2}+\frac{1}{4}+\frac{1}{15}+\frac{1}{660} \qquad \frac{19}{20}=\frac{1}{2}+\frac{1}{4}+\frac{1}{5}$$

Feladat

Készíts programot, amely adott N ($2 \le N \le 30$) és M ($1 \le M < N$) természetes számokra megadja az M / N tört egységtörtekre bontását!

Megoldás

A feladat megoldásakor figyeljünk a következőkre:

• Az M / N törtet, amennyiben M és N nem relatív prímek, érdemes egyszerűsíteni. Így -- ha az egyszerűsítés után a tört számlálója 1 -- lehet, hogy egyből megvan a keresett (egyelemű) egységtörtsorozat.

• A keresett egységtörtsorozat egy szigorúan monoton csökkenő sorozat. (Ebből következik, hogy -- mivel a törtek számlálója mindig 1 -- a törtek nevezői egy szigorúan monoton növekvő sorozatot alkotnak.) Legyenek ugyanis a és b relatív prímek úgy, hogy $a \neq 1$, és keressük azt a legkisebb c természetes számot, melyre teljesül az

$$\frac{a}{b}=\frac{1}{c}+m$$

egyenlőség, ahol $c \ge 2$ és $a/b > 1/c$, azaz $ac > b$. Ekkor

$$0<m=\frac{a}{b}-\frac{1}{c}=\frac{ac-b}{bc}<\frac{ac}{bc}=\frac{a}{b}$$

A továbbiakban már csak ezzel az m ($< a/b$) maradékkal kell tovább dolgoznunk, hasonlóan az előzőleg leírtakhoz.

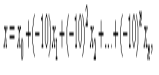
• Az egységtörtsorozat véges. A végességet az garantálja, hogy az  maradék számlálójának értéke sohasem lehet kisebb 1-nél, hiszen  . Mivel  ,  és  egész számok,  számlálója előbb-utóbb 1 lesz.

Ezek alapján íme a programkód:

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

#define HAMIS 0
#define IGAZ  ( !HAMIS )

unsigned long lnko( unsigned long a, unsigned long b )
{
  unsigned long maradek;
  while ( maradek = a % b )
  {
    a = b;
    b = maradek;
  }
  return b;
}

int main()
{
  unsigned long m, n, oszto;
  printf( "m = " );  scanf( "%lu", &m );
  printf( "n = " );  scanf( "%lu", &n );
  printf( "%lu/%lu =", m, n );
  oszto = lnko( m, n );
  m /= oszto;
  n /= oszto;
  while ( m != 1 )
  {
    unsigned long c = 2;
    while ( m * c <= n )
      ++c;
    printf( " 1/%lu +", c );
    m *= c;
    m -= n;
    n *= c;
    oszto = lnko( m, n );
    m /= oszto;
    n /= oszto;
  }
  printf( " 1/%lu\n", n );
  return EXIT_SUCCESS;
}
```

Számrendszerváltás

Egész számokat a 10-es mellett  -es számrendszerben is felírhatunk, a következőképpen:



ahol $0 \leq x_i \leq 9$ minden $0 \leq i \leq n$ esetén. Ebben a számrendszerben minden egész szám felírható előjel nélküli egész számként.

Feladat

Írd meg azokat a függvényeket, amelyek a paraméterként megadott 10-es, illetve -10-es számrendszerbeli számokat -10-es, illetve 10-es számrendszerbeli számokká konvertálják!

Példa

10-es számrendszerben	-10-es számrendszerben	
3	3	
-6	14	$(= -10 + 4)$
34	174	$(= 100 - 70 + 4)$
-72	88	$(= -80 + 8)$
163	243	$(= 200 - 40 + 3)$
-527	1533	$(= -1000 + 500 - 30 + 3)$
1526	19686	$(= 10000 - 9000 + 600 - 80 + 6)$
1994	18014	$(= 10000 - 8000 + 0 - 10 + 4)$

Megoldás

A feladatot legegyszerűbben rekurzív algoritmusokkal oldhatjuk meg.

```
int tizesbe( int szam )
{
  return szam ? -10 * tizesbe( szam / 10 ) + szam % 10 : 0;
}

int minuszba( int szam )
{
  if ( !szam )
    return 0;
  else if ( szam > 0 )
    return 10 * minuszba( -szam / 10 ) + szam % 10;
  else
  {
    int maradek = ( 10 - -szam % 10 ) % 10;
    return 10 * minuszba( -( szam - maradek ) / 10 ) + maradek;
  }
}
```

A 10-es számrendszerbe történő konverzió iteratívan is nagyon egyszerű a Horner-séma alkalmazásával (ugyanúgy történik, mint bármely más alapú számrendszer esetén).

A -10-es számrendszerbe történő átváltáskor mindig az 1-es helyiértéken lévő számjegyet vizsgáljuk. Ha a szám pozitív, akkor egyszerű maradékképzéssel dolgozunk, ha negatív, akkor figyelembe vesszük azt is, hogy a maradék (az 1-es helyiértéken lévő érték) nulla-e vagy sem.

```
int itertizesbe( int szam )
{
  char s[ 11 ], *p = s;
  int ertek = 0;
  sprintf( s, "%d", szam );
  do
    ertek = -10 * ertek + *p - '0';
  while ( *++p );
  return ertek;
}

int iterminuszba( int szam )
{
  int ertek = 0, hatvany = 1;
  while ( szam )
  {
    int maradek;
    if ( szam > 0 )
    {
      maradek = szam % 10;
    }
    else
    {
      maradek = -szam % 10;
      if ( maradek )
      {
        maradek = 10 - maradek;
      }
      szam -= maradek;
    }
    szam /= -10;
    ertek += hatvany * maradek;
    hatvany *= 10;
  }
  return ertek;
}
```

8. fejezet - Szimuláció

Tartalom

Josephus

Josephus Flavius problémája elég közismert. Azok kedvéért, akik nem ismerik az eredeti problémát: körben álló n ember közül, akiket az $1, 2, \ldots, n$ számokkal jelölünk meg, minden


-ediket kivégzik, és csak az utolsónak megmaradó személy életét kímélik meg. Josephus elég okos volt ahhoz, hogy az utolsónak megmaradó személy helyére álljon, ezért megmenekült, és hírt adhatott nekünk a történtekről. Amikor például $n = 6$ és $m = 5$, akkor az emberek az 5, 4, 6, 2, 3 sorrendben fognak meghalni, míg az 1-es megmenekül.

Tegyük fel, hogy van  jó fiú és  rossz fiú. A körben az első  helyen állnak a jó fiúk, az utolsó  helyen a rosszak. Meg kell határoznod azt a minimális  számot, amelyre az összes rossz fiú hamarabb fog meghalni, mint a jók közül bármelyik.

Input

Az input  értékeket tartalmazó sorokból áll. Az utolsó sor az input állományban egy 0-t tartalmaz, ezt a sort már nem kell feldolgoznod. Feltételezheted, hogy $0 < k < 14$.

Output

Az output sorai az input  értékeinek megfelelő  értékeket tartalmazzák.

Példa input

```
3
4
0
```

Példa output

```
5
30
```

Megoldás

A probléma megoldásánál vegyük észre a következőket:

• Minden  input esetén  lehetséges legkisebb értéke $k + 1$.

• Minden kiszámolási sorozat előtt $2k$ élő ember áll a körben. Egy kiszámolási sorozat addig tart, amíg  -nál több ember él.

• A kiszámolásnál csak az élő embereket kell figyelembe venni.

• A kiszámolási sorozatokban az élő emberek száma viszonylag hamar  értéke alá csökken, ezért a kiszámolás gyorsítható, ha  helyett az

értékig végezzük a kiszámolást.

• Ha már csak  ember él, és utolsóként nem egy jó fiút öltünk meg, akkor megtaláltuk a keresett  értéket.

Ezek alapján az első megoldásunk a következő:

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

#define ELO    1
#define HALOTT 0

int main()
{
  int k;
  scanf( "%d", &k );
  while ( k )
  {
    long m = k;
    int i, *emberek = ( int * )malloc( 2 * k * sizeof( int ) );
    do
    {
      int elok = 2 * k;
      ++m;
      for ( i = 0; i < 2 * k; ++i ) /* kezdetben mindenki él */
        emberek[ i ] = ELO;

      i = -1;
      while ( elok > k )
      {
        int szamolas = 0, darab = m % elok ? m % elok : elok;
        /* keressük az m-edik kiszámolandó embert */
        while ( szamolas < darab )
        {
          ++i;
          i %= 2 * k;
          /* csak az élő embereket figyeljük */
          if ( emberek[ i ] == ELO )
            ++szamolas;
        }
        emberek[ i ] = HALOTT; /* megtaláltuk, az i-edik ember meghal */
        --elok;                /* az élők száma eggyel csökken */
        if ( i < k )
          break;               /* ha jó fiút öltünk meg, megállunk */
      }

    } while ( i < k );

    printf( "%ld\n", m );
    free( emberek );

    scanf( "%d", &k );
  }
  return EXIT_SUCCESS;
}
```

A program $0 < k < 14$ értékekre az alábbi táblázatban látható eredményeket adja:

k	m
1	2
2	7
3	5
4	30
5	169
6	441
7	1 872
8	7 632
9	1 740
10	93 313
11	459 901
12	1 358 657
13	2 504 881

Ha meggondoljuk, ennek a táblázatnak a segítségével az eredeti problémára adhatunk egy rendkívül gyors megoldást: egyszerűen csak ki kell olvasni ebből a táblázatból a  input adathoz tartozó  értéket. Íme az ezt megvalósító program:

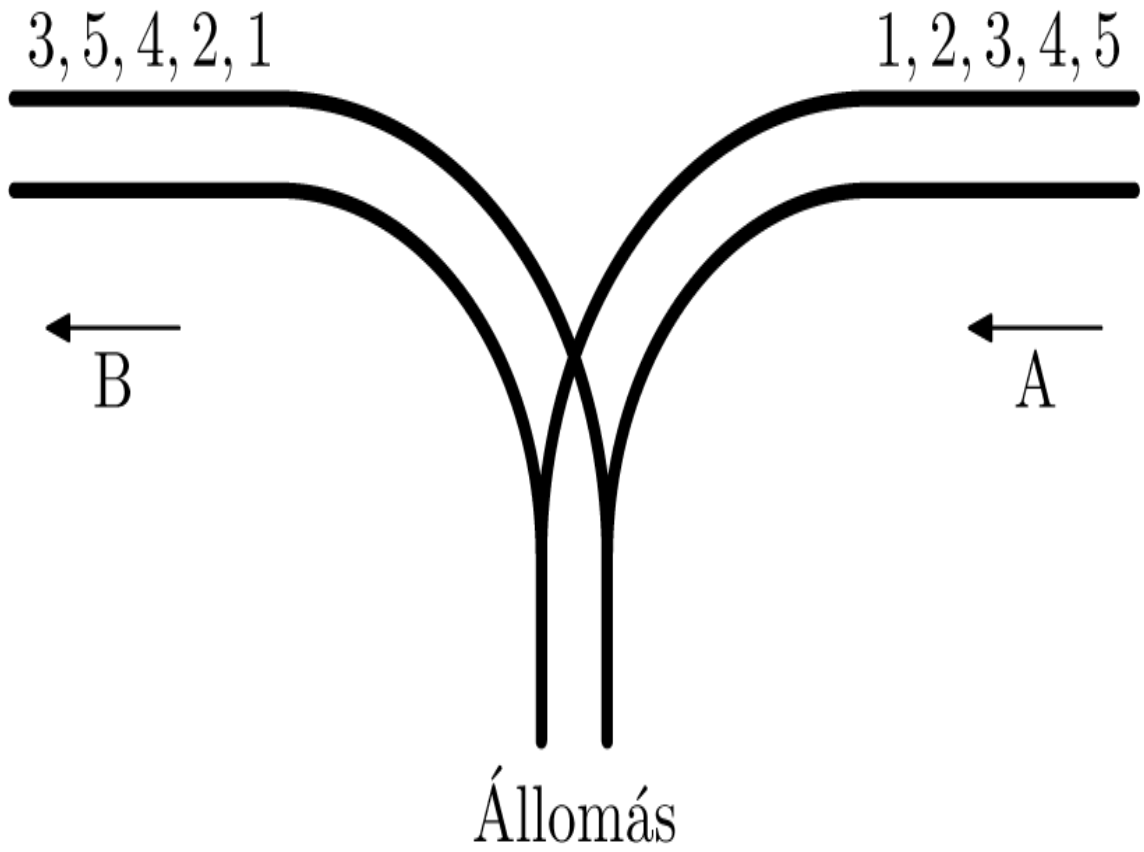
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  long tomb[] = { 0, 2, 7, 5, 30, 169, 441,
                  1872, 7632, 1740, 93313, 459901, 1358657, 2504881 };
  int k;
  scanf( "%d", &k );
  while ( k )
  {
    printf( "%ld\n", tomb[ k ] );
    scanf( "%d", &k );
```

```
  }
  return EXIT_SUCCESS;
}
```

Veremváros

Van egy nevezetes vasútállomás Veremvárosban. A táj arrafelé hihetelenül dimbes-dombos. Az állomás még a XIX. században épült. Sajnos az anyagi eszközök rendkívül korlátozottak voltak abban az időben. Csak egy felszíni pálya kiépítésére volt lehetőség. Ráadásul kiderült, hogy az állomásnak csak vakvágánya lehet (lásd a 8.1. ábrát) és a helyhiány miatt abból is csak egyetlenegy.

8.1. ábra -



A helyi hagyomány az, hogy minden A irányból érkező vonat a B irányban folytatja az útját a vasúti kocsik valamilyen módon történő átrendezésével. Tegyük fel, hogy az A irányból érkező vonatnak $N \leq 1000$ számozott kocsija van, 1-től N-ig növekvő sorrendben. A kocsirendezőnek tudnia kell, hogy vajon lehetséges-e átrendezni a vasúti kocsikat úgy, hogy útjukat a B irányban folytatva a sorrendjük $a_1, a_2, \ldots, a_N$ legyen. Segíts neki, és írj egy programot, amely eldönti, hogy elő lehet-e állítani a vasúti kocsik megadott sorrendjét. Feltételezheted, hogy az egyes kocsik lekapcsolhatók a vonatról, mielőtt beérkeznek az állomásra, és hogy egyedül is képesek mozogni, amíg a B irányba menő vágányra nem állnak. Feltételezheted továbbá azt is, hogy bármikor annyi vasúti kocsit lehet tárolni az állomáson, amennyit csak szükséges. Azonban ha egy vasúti kocsi begördült az állomásra, nem tud visszatérni az A irányból érkező vágányra, és hasonlóan, ha

egyszer elhagyta az állomást a B irányban, akkor nem tud visszatérni az állomásra.

Input

A bemenet blokkok sorozatából áll. Az utolsó kivételével minden blokk egy vagy több kimenő szerelvény leírását tartalmazza. A blokk első sorában a fent ismertetett N egész szám szerepel (ennyi kocsiból áll a szerelvény). A blokk minden további sorában az $1,2,\ldots,N$ számok egy permutációja áll. A blokk utolsó sora csak egy 0-t tartalmaz.

Az utolsó blokk egy sorból áll, amely egy 0-t tartalmaz.

Output

A bemenet minden permutációt tartalmazó sorához ki kell írnod egy sort. A kimenet egy sora az `Igen` szót tartalmazza, ha a vagonok átrendezhetők a bemenet megfelelő sorában megadott módon. Különben a `Nem` szót kell kiírni. A bemenet minden blokkja után egy üres sort kell kiírni (az utolsó nem 0-t tartalmazó blokk után is!). Az utolsó 0-t tartalmazó blokkhoz nem tartozik sor a kimeneten.

Példa input

```
5
1 2 3 4 5
5 4 1 2 3
0
6
6 5 4 3 2 1
0
0
```

Példa output

```
Igen
Nem

Igen
```

Megoldás

A vagonok kimeneti sorrendjét a `sorrend` tömbben tároljuk. Az A irányból érkező, növekvő sorszámú vagonokat közvetlenül a B irányba távozó vagonokkal hasonlítjuk össze. Amennyiben pontosan az a vagon érkezik, amelyik illik a távozó vagonok sorába, akkor beillesztjük, és megpróbáljuk hozzácsatolni a városban található vagonokat. Egyébként meg „vermeljük" az érkező vagont.

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

#define N 1000

int rendezheto( int sorrend[], int kocsiszam )
{
  int verem[ N ], vm = -1, i, sor = 0;
  for ( i = 1; i <= kocsiszam; i++ )
    if ( sorrend[ sor ] == i ) {
      sor++;
      while ( vm >= 0 && verem[ vm ] == sorrend[ sor ] ) {
        vm--;
        sor++;
      }
    }
    else
      verem[ ++vm ] = i;
```

```
  return sor == kocsiszam;
}

int main()
{
  int kocsik_szama;

  scanf( "%d", &kocsik_szama );
  while ( kocsik_szama ) {
    int elso;
    scanf( "%d", &elso );
    while ( elso ) {
      int szerelveny[ N ], i;
      szerelveny[ 0 ] = elso;
      for ( i = 1; i < kocsik_szama; i++ )
        scanf( "%d", &szerelveny[ i ] );

      puts( rendezheto( szerelveny, kocsik_szama ) ? "Igen" : "Nem" );

      scanf( "%d", &elso );
    }
    putchar( '\n' );
    scanf( "%d", &kocsik_szama );
  }

  return EXIT_SUCCESS;
}
```

9. fejezet - Sakk

Tartalom

NyargaLó

Egy barátod éppen a *NyargaLó* problémáját kutatja, amelynek lényege, hogy meg kell találni lóugrások azon legrövidebb sorozatát, amely a sakktábla adott n^2 mezőjének mindegyikét pontosan egyszer érinti. A barátod szerint a probléma legnehezebb része annak meghatározása, hogy minimálisan hány lóugrás szükséges ahhoz, hogy eljussunk egy adott mezőről egy másikra, és úgy gondolja, hogy ezek után az utat a két mező között már könnyű lesz megtalálni.

Természetesen te tudod, hogy ez fordítva van, így felajánlod neki, hogy írsz egy programot, amely megoldja a „nehezebb" részt.

Írj programot, amely beolvas két mezőt, és meghatározza, hogy hány lóugrást tartalmaz az a legrövidebb út, amely az egyik mezőből a másikba vezet!

Input

A bemenet egy vagy több tesztesetet tartalmaz. Minden teszteset egy sorból áll, amely két mezőt tartalmaz egy szóközzel elválasztva. Egy mező egy betűből (`a--h`) és egy számjegyből (`1--8`) áll, amelyek rendre a sakktábla egy oszlopát, illetve sorát jelölik.

Output

Minden tesztesethez tartozzon egy sor a következő formában:

xx és yy között n lóugrás

Példa input

```
e2 e4
a1 b2
b2 c3
a1 h8
a1 h7
h8 a1
b1 c3
f6 f6
```

Példa output

```
e2 és e4 között 2 lóugrás
a1 és b2 között 4 lóugrás
b2 és c3 között 2 lóugrás
a1 és h8 között 6 lóugrás
a1 és h7 között 5 lóugrás
h8 és a1 között 6 lóugrás
b1 és c3 között 1 lóugrás
f6 és f6 között 0 lóugrás
```

Megoldás

A feladatot elsőként a jól ismert szélességi kereséssel oldjuk meg. Minden tesztesetnél a 8×8-as tábla mezőit kezdetben -1-es értékekkel töltjük fel, a kiinduló mezőbe pedig egy 0 értéket írunk. Mindaddig, amíg a célmező értéke -1, azt nézzük, hogy melyek azok a még be nem járt mezők, ahová a már bejártakból eljuthatunk. Ezeknek az értékét mindig arra az értékre állítjuk be, ahány lépésben a kiinduló mezőről eljuthatunk hozzájuk. Ha a célmező is kap ilyen módon értéket, akkor abbahagyjuk a keresést, és kiírjuk a kapott lépésszámot.

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

#define MERET 8

char tabla[ MERET ][ MERET ];

int main()
{
  int i, j, kezdo_sor, cel_sor, szamlalo;
  char kezdo_oszlop, cel_oszlop;
  while ( scanf( " %c%d %c%d", &kezdo_oszlop,
                 &kezdo_sor, &cel_oszlop, &cel_sor ) != EOF )
  {
    for ( i = 0; i < MERET; ++i )
      for ( j = 0; j < MERET; ++j )
        tabla [ i ][ j ] = -1;
    tabla[ kezdo_sor - 1 ][ kezdo_oszlop - 'a' ] = szamlalo = 0;

    while ( tabla[ cel_sor - 1 ][ cel_oszlop - 'a' ] == -1 )
    {
      for ( i = 0; i < MERET; ++i )
        for ( j = 0; j < MERET; ++j )
          if ( tabla[ i ][ j ] == -1 &&
             ( i > 0 && j > 1 &&
                 tabla[ i-1 ][ j-2 ] == szamlalo ||
               i > 0 && j < MERET-2 &&
                 tabla[ i-1 ][ j+2 ] == szamlalo ||
               i > 1 && j > 0 &&
```

```
                tabla[ i-2 ][ j-1 ] == szamlalo ||
              i > 1 && j < MERET-1 &&
                tabla[ i-2 ][ j+1 ] == szamlalo ||
              i < MERET-1 && j > 1 &&
                tabla[ i+1 ][ j-2 ] == szamlalo ||
              i < MERET-1 && j < MERET-2 &&
                tabla[ i+1 ][ j+2 ] == szamlalo ||
              i < MERET-2 && j > 0 &&
                tabla[ i+2 ][ j-1 ] == szamlalo ||
              i < MERET-2 && j < MERET-1 &&
                tabla[ i+2 ][ j+1 ] == szamlalo ) )
            tabla[ i ][ j ] = szamlalo + 1;
      ++szamlalo;
    }
    printf( "%c%d és %c%d között %d lóugrás\n",
            kezdo_oszlop, kezdo_sor, cel_oszlop, cel_sor, szamlalo );
  }

  return EXIT_SUCCESS;
}
```

Amennyiben összepárosítjuk az összes lehetséges kiinduló és célmezőt, és feljegyezzük, hogy hány lépésben lehet az egyikről a másikra eljutni, akkor ezekből az adatokból mindenféle további számolgatás nélkül meg tudjuk határozni a kérdéses lépéstávokat. Az adatokat nem `int`, hanem `char` típusú tömbben tároljuk, mivel így kevesebb helyett foglalnak.

További gyorsítást érhetünk el, ha nem a formátumozott adatbeviteli és -kiviteli függvényeket (`scanf()` és `printf()`) használjuk, hanem a `gets()` és `puts()` függvényekkel dolgozunk.

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

char tabla[] = { 0, 3, 2, 3, 2, 3, 4, 5, 3, 4, 1, 2, 3, 4, 3, 4,
                 2, 1, 4, 3, 2, 3, 4, 5, 3, 2, 3, 2, 3, 4, 3, 4,
                 2, 3, 2, 3, 4, 3, 4, 5, 3, 4, 3, 4, 3, 4, 5, 4,
                 4, 3, 4, 3, 4, 5, 4, 5, 5, 4, 5, 4, 5, 4, 5, 6,
                 3, 0, 3, 2, 3, 2, 3, 4, 2, 3, 2, 1, 2, 3, 4, 3,
                 1, 2, 1, 4, 3, 2, 3, 4, 2, 3, 2, 3, 2, 3, 4, 3,
                 3, 2, 3, 2, 3, 4, 3, 4, 4, 3, 4, 3, 4, 3, 4, 5,
                 3, 4, 3, 4, 3, 4, 5, 4, 4, 5, 4, 5, 4, 5, 4, 5,
                 2, 3, 0, 3, 2, 3, 2, 3, 1, 2, 3, 2, 1, 2, 3, 4,
                 4, 1, 2, 1, 4, 3, 2, 3, 3, 2, 3, 2, 3, 2, 3, 4,
                 2, 3, 2, 3, 2, 3, 4, 3, 3, 4, 3, 4, 3, 4, 3, 4,
                 4, 3, 4, 3, 4, 3, 4, 5, 5, 4, 5, 4, 5, 4, 5, 4,
                 3, 2, 3, 0, 3, 2, 3, 2, 2, 1, 2, 3, 2, 1, 2, 3,
                 3, 4, 1, 2, 1, 4, 3, 2, 2, 3, 2, 3, 2, 3, 2, 3,
                 3, 2, 3, 2, 3, 2, 3, 4, 4, 3, 4, 3, 4, 3, 4, 3,
                 3, 4, 3, 4, 3, 4, 3, 4, 4, 5, 4, 5, 4, 5, 4, 5,
                 2, 3, 2, 3, 0, 3, 2, 3, 3, 2, 1, 2, 3, 2, 1, 2,
                 2, 3, 4, 1, 2, 1, 4, 3, 3, 2, 3, 2, 3, 2, 3, 2,
                 4, 3, 2, 3, 2, 3, 2, 3, 3, 4, 3, 4, 3, 4, 3, 4,
                 4, 3, 4, 3, 4, 3, 4, 3, 5, 4, 5, 4, 5, 4, 5, 4,
                 3, 2, 3, 2, 3, 0, 3, 2, 4, 3, 2, 1, 2, 3, 2, 1,
                 3, 2, 3, 4, 1, 2, 1, 4, 4, 3, 2, 3, 2, 3, 2, 3,
                 3, 4, 3, 2, 3, 2, 3, 2, 4, 3, 4, 3, 4, 3, 4, 3,
                 5, 4, 3, 4, 3, 4, 3, 4, 4, 5, 4, 5, 4, 5, 4, 5,
                 4, 3, 2, 3, 2, 3, 0, 3, 3, 4, 3, 2, 1, 2, 3, 2,
                 4, 3, 2, 3, 4, 1, 2, 1, 3, 4, 3, 2, 3, 2, 3, 2,
                 4, 3, 4, 3, 2, 3, 2, 3, 5, 4, 3, 4, 3, 4, 3, 4,
                 4, 5, 4, 3, 4, 3, 4, 3, 5, 4, 5, 4, 5, 4, 5, 4,
                 5, 4, 3, 2, 3, 2, 3, 0, 4, 3, 4, 3, 2, 1, 4, 3,
                 5, 4, 3, 2, 3, 4, 1, 2, 4, 3, 4, 3, 2, 3, 2, 3,
```

5, 4, 3, 4, 3, 2, 3, 2, 4, 5, 4, 3, 4, 3, 4, 3,
5, 4, 5, 4, 3, 4, 3, 4, 6, 5, 4, 5, 4, 5, 4, 5,
3, 2, 1, 2, 3, 4, 3, 4, 0, 3, 2, 3, 2, 3, 4, 5,
3, 2, 1, 2, 3, 4, 3, 4, 2, 1, 4, 3, 2, 3, 4, 5,
3, 2, 3, 2, 3, 4, 3, 4, 2, 3, 2, 3, 4, 3, 4, 5,
3, 4, 3, 4, 3, 4, 5, 4, 4, 3, 4, 3, 4, 5, 4, 5,
4, 3, 2, 1, 2, 3, 4, 3, 3, 0, 3, 2, 3, 2, 3, 4,
2, 3, 2, 1, 2, 3, 4, 3, 1, 2, 1, 4, 3, 2, 3, 4,
2, 3, 2, 3, 2, 3, 4, 3, 3, 2, 3, 2, 3, 4, 3, 4,
4, 3, 4, 3, 4, 3, 4, 5, 3, 4, 3, 4, 3, 4, 5, 4,
1, 2, 3, 2, 1, 2, 3, 4, 2, 3, 0, 3, 2, 3, 2, 3,
1, 2, 3, 2, 1, 2, 3, 4, 4, 1, 2, 1, 4, 3, 2, 3,
3, 2, 3, 2, 3, 2, 3, 4, 2, 3, 2, 3, 2, 3, 4, 3,
3, 4, 3, 4, 3, 4, 3, 4, 4, 3, 4, 3, 4, 3, 4, 5,
2, 1, 2, 3, 2, 1, 2, 3, 3, 2, 3, 0, 3, 2, 3, 2,
2, 1, 2, 3, 2, 1, 2, 3, 3, 4, 1, 2, 1, 4, 3, 2,
2, 3, 2, 3, 2, 3, 2, 3, 3, 2, 3, 2, 3, 2, 3, 4,
4, 3, 4, 3, 4, 3, 4, 3, 3, 4, 3, 4, 3, 4, 3, 4,
3, 2, 1, 2, 3, 2, 1, 2, 2, 3, 2, 3, 0, 3, 2, 3,
3, 2, 1, 2, 3, 2, 1, 2, 2, 3, 4, 1, 2, 1, 4, 3,
3, 2, 3, 2, 3, 2, 3, 2, 4, 3, 2, 3, 2, 3, 2, 3,
3, 4, 3, 4, 3, 4, 3, 4, 4, 3, 4, 3, 4, 3, 4, 3,
4, 3, 2, 1, 2, 3, 2, 1, 3, 2, 3, 2, 3, 0, 3, 2,
4, 3, 2, 1, 2, 3, 2, 1, 3, 2, 3, 4, 1, 2, 1, 4,
4, 3, 2, 3, 2, 3, 2, 3, 3, 4, 3, 2, 3, 2, 3, 2,
4, 3, 4, 3, 4, 3, 4, 3, 5, 4, 3, 4, 3, 4, 3, 4,
3, 4, 3, 2, 1, 2, 3, 4, 4, 3, 2, 3, 2, 3, 0, 3,
3, 4, 3, 2, 1, 2, 3, 2, 4, 3, 2, 3, 4, 1, 2, 1,
3, 4, 3, 2, 3, 2, 3, 2, 4, 3, 4, 3, 2, 3, 2, 3,
5, 4, 3, 4, 3, 4, 3, 4, 4, 5, 4, 3, 4, 3, 4, 3,
4, 3, 4, 3, 2, 1, 2, 3, 5, 4, 3, 2, 3, 2, 3, 0,
4, 3, 4, 3, 2, 1, 2, 3, 5, 4, 3, 2, 3, 4, 1, 2,
4, 3, 4, 3, 2, 3, 2, 3, 5, 4, 3, 4, 3, 2, 3, 2,
4, 5, 4, 3, 4, 3, 4, 3, 5, 4, 5, 4, 3, 4, 3, 4,
2, 1, 4, 3, 2, 3, 4, 5, 3, 2, 1, 2, 3, 4, 3, 4,
0, 3, 2, 3, 2, 3, 4, 5, 3, 2, 1, 2, 3, 4, 3, 4,
2, 1, 4, 3, 2, 3, 4, 5, 3, 2, 3, 2, 3, 4, 3, 4,
2, 3, 2, 3, 4, 3, 4, 5, 3, 4, 3, 4, 3, 4, 5, 4,
1, 2, 1, 4, 3, 2, 3, 4, 2, 3, 2, 1, 2, 3, 4, 3,
3, 0, 3, 2, 3, 2, 3, 4, 2, 3, 2, 1, 2, 3, 4, 3,
1, 2, 1, 4, 3, 2, 3, 4, 2, 3, 2, 3, 2, 3, 4, 3,
3, 2, 3, 2, 3, 4, 3, 4, 4, 3, 4, 3, 4, 3, 4, 5,
4, 1, 2, 1, 4, 3, 2, 3, 1, 2, 3, 2, 1, 2, 3, 4,
2, 3, 0, 3, 2, 3, 2, 3, 1, 2, 3, 2, 1, 2, 3, 4,
4, 1, 2, 1, 4, 3, 2, 3, 3, 2, 3, 2, 3, 2, 3, 4,
2, 3, 2, 3, 2, 3, 4, 3, 3, 4, 3, 4, 3, 4, 3, 4,
3, 4, 1, 2, 1, 4, 3, 2, 2, 1, 2, 3, 2, 1, 2, 3,
3, 2, 3, 0, 3, 2, 3, 2, 2, 1, 2, 3, 2, 1, 2, 3,
3, 4, 1, 2, 1, 4, 3, 2, 2, 3, 2, 3, 2, 3, 2, 3,
3, 2, 3, 2, 3, 2, 3, 4, 4, 3, 4, 3, 4, 3, 4, 3,
2, 3, 4, 1, 2, 1, 4, 3, 3, 2, 1, 2, 3, 2, 1, 2,
2, 3, 2, 3, 0, 3, 2, 3, 3, 2, 1, 2, 3, 2, 1, 2,
2, 3, 4, 1, 2, 1, 4, 3, 3, 2, 3, 2, 3, 2, 3, 2,
4, 3, 2, 3, 2, 3, 2, 3, 3, 4, 3, 4, 3, 4, 3, 4,
3, 2, 3, 4, 1, 2, 1, 4, 4, 3, 2, 1, 2, 3, 2, 1,
3, 2, 3, 2, 3, 0, 3, 2, 4, 3, 2, 1, 2, 3, 2, 1,
3, 2, 3, 4, 1, 2, 1, 4, 4, 3, 2, 3, 2, 3, 2, 3,
3, 4, 3, 2, 3, 2, 3, 2, 4, 3, 4, 3, 4, 3, 4, 3,
4, 3, 2, 3, 4, 1, 2, 1, 3, 4, 3, 2, 1, 2, 3, 2,
4, 3, 2, 3, 2, 3, 0, 3, 3, 4, 3, 2, 1, 2, 3, 2,
4, 3, 2, 3, 4, 1, 2, 1, 3, 4, 3, 2, 3, 2, 3, 2,
4, 3, 4, 3, 2, 3, 2, 3, 5, 4, 3, 4, 3, 4, 3, 4,

5, 4, 3, 2, 3, 4, 1, 2, 4, 3, 4, 3, 2, 1, 2, 3,
5, 4, 3, 2, 3, 2, 3, 0, 4, 3, 4, 3, 2, 1, 2, 3,
5, 4, 3, 2, 3, 4, 1, 2, 4, 3, 4, 3, 2, 3, 2, 3,
5, 4, 3, 4, 3, 2, 3, 2, 4, 5, 4, 3, 4, 3, 4, 3,
3, 2, 3, 2, 3, 4, 3, 4, 2, 1, 4, 3, 2, 3, 4, 5,
3, 2, 1, 2, 3, 4, 3, 4, 0, 3, 2, 3, 2, 3, 4, 5,
3, 2, 1, 2, 3, 4, 3, 4, 2, 1, 4, 3, 2, 3, 4, 5,
3, 2, 3, 2, 3, 4, 3, 4, 2, 3, 2, 3, 4, 3, 4, 5,
2, 3, 2, 3, 2, 3, 4, 3, 1, 2, 1, 4, 3, 2, 3, 4,
2, 3, 2, 1, 2, 3, 4, 3, 3, 0, 3, 2, 3, 2, 3, 4,
2, 3, 2, 1, 2, 3, 4, 3, 1, 2, 1, 4, 3, 2, 3, 4,
2, 3, 2, 3, 2, 3, 4, 3, 3, 2, 3, 2, 3, 4, 3, 4,
3, 2, 3, 2, 3, 2, 3, 4, 4, 1, 2, 1, 4, 3, 2, 3,
1, 2, 3, 2, 1, 2, 3, 4, 2, 3, 0, 3, 2, 3, 2, 3,
1, 2, 3, 2, 1, 2, 3, 4, 4, 1, 2, 1, 4, 3, 2, 3,
3, 2, 3, 2, 3, 2, 3, 4, 2, 3, 2, 3, 2, 3, 4, 3,
2, 3, 2, 3, 2, 3, 2, 3, 3, 4, 1, 2, 1, 4, 3, 2,
2, 1, 2, 3, 2, 1, 2, 3, 3, 2, 3, 0, 3, 2, 3, 2,
2, 1, 2, 3, 2, 1, 2, 3, 3, 4, 1, 2, 1, 4, 3, 2,
2, 3, 2, 3, 2, 3, 2, 3, 3, 2, 3, 2, 3, 2, 3, 4,
3, 2, 3, 2, 3, 2, 3, 2, 2, 3, 4, 1, 2, 1, 4, 3,
3, 2, 1, 2, 3, 2, 1, 2, 2, 3, 2, 3, 0, 3, 2, 3,
3, 2, 1, 2, 3, 2, 1, 2, 2, 3, 4, 1, 2, 1, 4, 3,
3, 2, 3, 2, 3, 2, 3, 2, 4, 3, 2, 3, 2, 3, 2, 3,
4, 3, 2, 3, 2, 3, 2, 3, 3, 2, 3, 4, 1, 2, 1, 4,
4, 3, 2, 1, 2, 3, 2, 1, 3, 2, 3, 2, 3, 0, 3, 2,
4, 3, 2, 1, 2, 3, 2, 1, 3, 2, 3, 4, 1, 2, 1, 4,
4, 3, 2, 3, 2, 3, 2, 3, 3, 4, 3, 2, 3, 2, 3, 2,
3, 4, 3, 2, 3, 2, 3, 2, 4, 3, 2, 3, 4, 1, 2, 1,
3, 4, 3, 2, 1, 2, 3, 2, 4, 3, 2, 3, 2, 3, 0, 3,
3, 4, 3, 2, 1, 2, 3, 2, 4, 3, 2, 3, 4, 1, 2, 1,
3, 4, 3, 2, 3, 2, 3, 2, 4, 3, 4, 3, 2, 3, 2, 3,
4, 3, 4, 3, 2, 3, 2, 3, 5, 4, 3, 2, 3, 4, 1, 2,
4, 3, 4, 3, 2, 1, 2, 3, 5, 4, 3, 2, 3, 2, 3, 0,
4, 3, 4, 3, 2, 1, 2, 3, 5, 4, 3, 2, 3, 4, 1, 2,
4, 3, 4, 3, 2, 3, 2, 3, 5, 4, 3, 4, 3, 2, 3, 2,
2, 3, 2, 3, 4, 3, 4, 5, 3, 2, 3, 2, 3, 4, 3, 4,
2, 1, 4, 3, 2, 3, 4, 5, 3, 2, 1, 2, 3, 4, 3, 4,
0, 3, 2, 3, 2, 3, 4, 5, 3, 2, 1, 2, 3, 4, 3, 4,
2, 1, 4, 3, 2, 3, 4, 5, 3, 2, 3, 2, 3, 4, 3, 4,
3, 2, 3, 2, 3, 4, 3, 4, 2, 3, 2, 3, 2, 3, 4, 3,
1, 2, 1, 4, 3, 2, 3, 4, 2, 3, 2, 1, 2, 3, 4, 3,
3, 0, 3, 2, 3, 2, 3, 4, 2, 3, 2, 1, 2, 3, 4, 3,
1, 2, 1, 4, 3, 2, 3, 4, 2, 3, 2, 3, 2, 3, 4, 3,
2, 3, 2, 3, 2, 3, 4, 3, 3, 2, 3, 2, 3, 2, 3, 4,
4, 1, 2, 1, 4, 3, 2, 3, 1, 2, 3, 2, 1, 2, 3, 4,
2, 3, 0, 3, 2, 3, 2, 3, 1, 2, 3, 2, 1, 2, 3, 4,
4, 1, 2, 1, 4, 3, 2, 3, 3, 2, 3, 2, 3, 2, 3, 4,
3, 2, 3, 2, 3, 2, 3, 4, 2, 3, 2, 3, 2, 3, 2, 3,
3, 4, 1, 2, 1, 4, 3, 2, 2, 1, 2, 3, 2, 1, 2, 3,
3, 2, 3, 0, 3, 2, 3, 2, 2, 1, 2, 3, 2, 1, 2, 3,
3, 4, 1, 2, 1, 4, 3, 2, 2, 3, 2, 3, 2, 3, 2, 3,
4, 3, 2, 3, 2, 3, 2, 3, 3, 2, 3, 2, 3, 2, 3, 2,
2, 3, 4, 1, 2, 1, 4, 3, 3, 2, 1, 2, 3, 2, 1, 2,
2, 3, 2, 3, 0, 3, 2, 3, 3, 2, 1, 2, 3, 2, 1, 2,
2, 3, 4, 1, 2, 1, 4, 3, 3, 2, 3, 2, 3, 2, 3, 2,
3, 4, 3, 2, 3, 2, 3, 2, 4, 3, 2, 3, 2, 3, 2, 3,
3, 2, 3, 4, 1, 2, 1, 4, 4, 3, 2, 1, 2, 3, 2, 1,
3, 2, 3, 2, 3, 0, 3, 2, 4, 3, 2, 1, 2, 3, 2, 1,
3, 2, 3, 4, 1, 2, 1, 4, 4, 3, 2, 3, 2, 3, 2, 3,
4, 3, 4, 3, 2, 3, 2, 3, 3, 4, 3, 2, 3, 2, 3, 2,
4, 3, 2, 3, 4, 1, 2, 1, 3, 4, 3, 2, 1, 2, 3, 2,

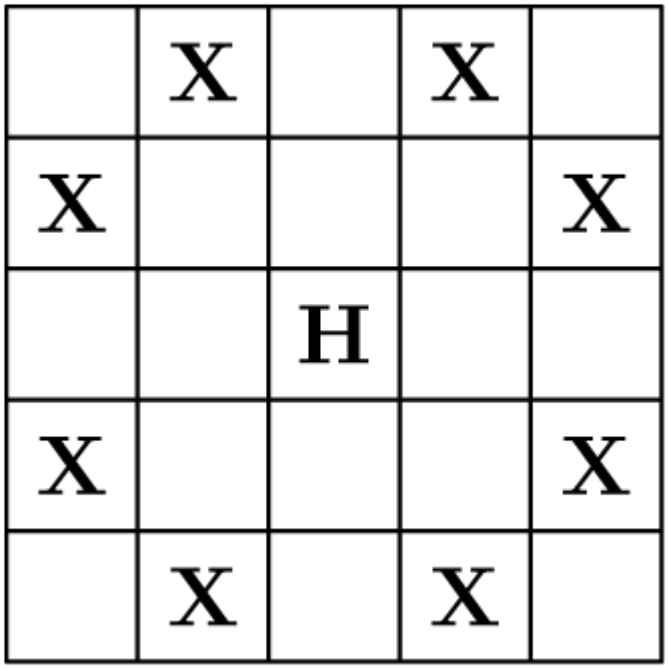
4, 3, 2, 3, 2, 3, 0, 3, 3, 4, 3, 2, 1, 2, 3, 2,
4, 3, 2, 3, 4, 1, 2, 1, 3, 4, 3, 2, 3, 2, 3, 2,
5, 4, 3, 4, 3, 2, 3, 2, 4, 3, 4, 3, 2, 3, 2, 3,
5, 4, 3, 2, 3, 4, 1, 2, 4, 3, 4, 3, 2, 1, 2, 3,
5, 4, 3, 2, 3, 2, 3, 0, 4, 3, 4, 3, 2, 1, 2, 3,
5, 4, 3, 2, 3, 4, 1, 2, 4, 3, 4, 3, 2, 3, 2, 3,
3, 4, 3, 4, 3, 4, 5, 4, 2, 3, 2, 3, 4, 3, 4, 5,
3, 2, 3, 2, 3, 4, 3, 4, 2, 1, 4, 3, 2, 3, 4, 5,
3, 2, 1, 2, 3, 4, 3, 4, 0, 3, 2, 3, 2, 3, 4, 5,
3, 2, 1, 2, 3, 4, 3, 4, 2, 1, 4, 3, 2, 3, 4, 5,
4, 3, 4, 3, 4, 3, 4, 5, 3, 2, 3, 2, 3, 4, 3, 4,
2, 3, 2, 3, 2, 3, 4, 3, 1, 2, 1, 4, 3, 2, 3, 4,
2, 3, 2, 1, 2, 3, 4, 3, 3, 0, 3, 2, 3, 2, 3, 4,
2, 3, 2, 1, 2, 3, 4, 3, 1, 2, 1, 4, 3, 2, 3, 4,
3, 4, 3, 4, 3, 4, 3, 4, 2, 3, 2, 3, 2, 3, 4, 3,
3, 2, 3, 2, 3, 2, 3, 4, 4, 1, 2, 1, 4, 3, 2, 3,
1, 2, 3, 2, 1, 2, 3, 4, 2, 3, 0, 3, 2, 3, 2, 3,
1, 2, 3, 2, 1, 2, 3, 4, 4, 1, 2, 1, 4, 3, 2, 3,
4, 3, 4, 3, 4, 3, 4, 3, 3, 2, 3, 2, 3, 2, 3, 4,
2, 3, 2, 3, 2, 3, 2, 3, 3, 4, 1, 2, 1, 4, 3, 2,
2, 1, 2, 3, 2, 1, 2, 3, 3, 2, 3, 0, 3, 2, 3, 2,
2, 1, 2, 3, 2, 1, 2, 3, 3, 4, 1, 2, 1, 4, 3, 2,
3, 4, 3, 4, 3, 4, 3, 4, 4, 3, 2, 3, 2, 3, 2, 3,
3, 2, 3, 2, 3, 2, 3, 2, 2, 3, 4, 1, 2, 1, 4, 3,
3, 2, 1, 2, 3, 2, 1, 2, 2, 3, 2, 3, 0, 3, 2, 3,
3, 2, 1, 2, 3, 2, 1, 2, 2, 3, 4, 1, 2, 1, 4, 3,
4, 3, 4, 3, 4, 3, 4, 3, 3, 4, 3, 2, 3, 2, 3, 2,
4, 3, 2, 3, 2, 3, 2, 3, 3, 2, 3, 4, 1, 2, 1, 4,
4, 3, 2, 1, 2, 3, 2, 1, 3, 2, 3, 2, 3, 0, 3, 2,
4, 3, 2, 1, 2, 3, 2, 1, 3, 2, 3, 4, 1, 2, 1, 4,
5, 4, 3, 4, 3, 4, 3, 4, 4, 3, 4, 3, 2, 3, 2, 3,
3, 4, 3, 2, 3, 2, 3, 2, 4, 3, 2, 3, 4, 1, 2, 1,
3, 4, 3, 2, 1, 2, 3, 2, 4, 3, 2, 3, 2, 3, 0, 3,
3, 4, 3, 2, 1, 2, 3, 2, 4, 3, 2, 3, 4, 1, 2, 1,
4, 5, 4, 3, 4, 3, 4, 3, 5, 4, 3, 4, 3, 2, 3, 2,
4, 3, 4, 3, 2, 3, 2, 3, 5, 4, 3, 2, 3, 4, 1, 2,
4, 3, 4, 3, 2, 1, 2, 3, 5, 4, 3, 2, 3, 2, 3, 0,
4, 3, 4, 3, 2, 1, 2, 3, 5, 4, 3, 2, 3, 4, 1, 2,
4, 3, 4, 3, 4, 5, 4, 5, 3, 4, 3, 4, 3, 4, 5, 4,
2, 3, 2, 3, 4, 3, 4, 5, 3, 2, 3, 2, 3, 4, 3, 4,
2, 1, 4, 3, 2, 3, 4, 5, 3, 2, 1, 2, 3, 4, 3, 4,
0, 3, 2, 3, 2, 3, 4, 5, 3, 2, 1, 2, 3, 4, 3, 4,
3, 4, 3, 4, 3, 4, 5, 4, 4, 3, 4, 3, 4, 3, 4, 5,
3, 2, 3, 2, 3, 4, 3, 4, 2, 3, 2, 3, 2, 3, 4, 3,
1, 2, 1, 4, 3, 2, 3, 4, 2, 3, 2, 1, 2, 3, 4, 3,
3, 0, 3, 2, 3, 2, 3, 4, 4, 3, 2, 1, 2, 3, 4, 3,
4, 3, 4, 3, 4, 3, 4, 5, 3, 4, 3, 4, 3, 4, 3, 4,
2, 3, 2, 3, 2, 3, 4, 3, 3, 2, 3, 2, 3, 2, 3, 4,
4, 1, 2, 1, 4, 3, 2, 3, 1, 2, 3, 2, 1, 2, 3, 4,
2, 3, 0, 3, 2, 3, 2, 3, 1, 2, 3, 2, 1, 2, 3, 4,
3, 4, 3, 4, 3, 4, 3, 4, 4, 3, 4, 3, 4, 3, 4, 3,
3, 2, 3, 2, 3, 2, 3, 4, 2, 3, 2, 3, 2, 3, 2, 3,
3, 4, 1, 2, 1, 4, 3, 2, 2, 1, 2, 3, 2, 1, 2, 3,
3, 2, 3, 0, 3, 2, 3, 2, 2, 1, 2, 3, 2, 1, 2, 3,
4, 3, 4, 3, 4, 3, 4, 3, 3, 4, 3, 4, 3, 4, 3, 4,
4, 3, 2, 3, 2, 3, 2, 3, 3, 2, 3, 2, 3, 2, 3, 2,
2, 3, 4, 1, 2, 1, 4, 3, 3, 2, 1, 2, 3, 2, 1, 2,
2, 3, 2, 3, 0, 3, 2, 3, 3, 2, 1, 2, 3, 2, 1, 2,
5, 4, 3, 4, 3, 4, 3, 4, 4, 3, 4, 3, 4, 3, 4, 3,
3, 4, 3, 2, 3, 2, 3, 2, 4, 3, 2, 3, 2, 3, 2, 3,
3, 2, 3, 4, 1, 2, 1, 4, 4, 3, 2, 1, 2, 3, 2, 1,
3, 2, 3, 2, 3, 0, 3, 2, 4, 3, 2, 1, 2, 3, 2, 1,

```
                4, 5, 4, 3, 4, 3, 4, 3, 5, 4, 3, 4, 3, 4, 3, 4,
                4, 3, 4, 3, 2, 3, 2, 3, 3, 4, 3, 2, 3, 2, 3, 2,
                4, 3, 2, 3, 4, 1, 2, 1, 3, 4, 3, 2, 1, 2, 3, 2,
                4, 3, 2, 3, 2, 3, 0, 3, 3, 4, 3, 2, 1, 2, 3, 4,
                5, 4, 5, 4, 3, 4, 3, 4, 4, 5, 4, 3, 4, 3, 4, 3,
                5, 4, 3, 4, 3, 2, 3, 2, 4, 3, 4, 3, 2, 3, 2, 3,
                5, 4, 3, 2, 3, 4, 1, 2, 4, 3, 4, 3, 2, 1, 2, 3,
                5, 4, 3, 2, 3, 2, 3, 0, 4, 3, 4, 3, 2, 1, 2, 3,
                5, 4, 5, 4, 5, 4, 5, 6, 4, 3, 4, 3, 4, 5, 4, 5,
                3, 4, 3, 4, 3, 4, 5, 4, 2, 3, 2, 3, 4, 3, 4, 5,
                3, 2, 3, 2, 3, 4, 3, 4, 2, 1, 4, 3, 2, 3, 4, 5,
                3, 4, 1, 2, 3, 4, 3, 4, 0, 3, 2, 3, 2, 3, 4, 5,
                4, 5, 4, 5, 4, 5, 4, 5, 3, 4, 3, 4, 3, 4, 5, 4,
                4, 3, 4, 3, 4, 3, 4, 5, 3, 2, 3, 2, 3, 4, 3, 4,
                2, 3, 2, 3, 2, 3, 4, 3, 1, 2, 1, 4, 3, 2, 3, 4,
                2, 3, 2, 1, 2, 3, 4, 3, 3, 0, 3, 2, 3, 2, 3, 4,
                5, 4, 5, 4, 5, 4, 5, 4, 4, 3, 4, 3, 4, 3, 4, 5,
                3, 4, 3, 4, 3, 4, 3, 4, 2, 3, 2, 3, 2, 3, 4, 3,
                3, 2, 3, 2, 3, 2, 3, 4, 4, 1, 2, 1, 4, 3, 2, 3,
                1, 2, 3, 2, 1, 2, 3, 4, 2, 3, 0, 3, 2, 3, 2, 3,
                4, 5, 4, 5, 4, 5, 4, 5, 3, 4, 3, 4, 3, 4, 3, 4,
                4, 3, 4, 3, 4, 3, 4, 3, 3, 2, 3, 2, 3, 2, 3, 4,
                2, 3, 2, 3, 2, 3, 2, 3, 3, 4, 1, 2, 1, 4, 3, 2,
                2, 1, 2, 3, 2, 1, 2, 3, 3, 2, 3, 0, 3, 2, 3, 2,
                5, 4, 5, 4, 5, 4, 5, 4, 4, 3, 4, 3, 4, 3, 4, 3,
                3, 4, 3, 4, 3, 4, 3, 4, 4, 3, 2, 3, 2, 3, 2, 3,
                3, 2, 3, 2, 3, 2, 3, 2, 2, 3, 4, 1, 2, 1, 4, 3,
                3, 2, 1, 2, 3, 2, 1, 2, 2, 3, 2, 3, 0, 3, 2, 3,
                4, 5, 4, 5, 4, 5, 4, 5, 5, 4, 3, 4, 3, 4, 3, 4,
                4, 3, 4, 3, 4, 3, 4, 3, 3, 4, 3, 2, 3, 2, 3, 2,
                4, 3, 2, 3, 2, 3, 2, 3, 3, 2, 3, 4, 1, 2, 1, 4,
                4, 3, 2, 1, 2, 3, 2, 1, 3, 2, 3, 2, 3, 0, 3, 2,
                5, 4, 5, 4, 5, 4, 5, 4, 4, 5, 4, 3, 4, 3, 4, 3,
                5, 4, 3, 4, 3, 4, 3, 4, 4, 3, 4, 3, 2, 3, 2, 3,
                3, 4, 3, 2, 3, 2, 3, 2, 4, 3, 2, 3, 4, 1, 2, 1,
                3, 4, 3, 2, 1, 2, 3, 2, 4, 3, 2, 3, 2, 3, 0, 3,
                6, 5, 4, 5, 4, 5, 4, 5, 5, 4, 5, 4, 3, 4, 3, 4,
                4, 5, 4, 3, 4, 3, 4, 3, 5, 4, 3, 4, 3, 2, 3, 2,
                4, 3, 4, 3, 2, 3, 2, 3, 5, 4, 3, 2, 3, 4, 1, 2,
                4, 3, 4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 3, 2, 3, 0 };

int main()
{
  char sor[ 10 ], szoveg[] = "a1 és a1 között 0 lóugrás";
  while ( gets( sor ) )
  {
    szoveg[ 0 ] = sor[ 0 ];
    szoveg[ 1 ] = sor[ 1 ];
    szoveg[ 6 ] = sor[ 3 ];
    szoveg[ 7 ] = sor[ 4 ];
    szoveg[ 16 ] = tabla[ 8 * ( 8 * ( 8 * ( sor[ 0 ] - 'a' ) +
                          sor[ 1 ] - '0' - 1 ) + sor[ 3 ] - 'a' ) +
                          sor[ 4 ] - '0' - 1 ] + '0';
    puts( szoveg );
  }

  return EXIT_SUCCESS;
}
```

Hány huszár?

A huszár egy olyan sakkfigura, amely a sakktáblán a pozíciójától vagy (a) két sorral és egy oszloppal, vagy (b) egy sorral és két oszloppal távolabb lévő mezőket támad, ahogy a 9.1. ábrán látható. Az **H**-val jelölt mező jelzi a huszár pozícióját, az **X**-szel jelöltek pedig az ütés alatt álló mezők.

9.1. ábra -



Ebben a feladatban meg kell határoznod, hogy legfeljebb hány huszárt lehet elhelyezni egy M sorból és N oszlopból álló táblán úgy, hogy ne üssék egymást. M és N értéke nem nagyobb, mint 500.

Input

A bemenet az M és N egész értékek párjaiból áll. Az utolsó számpár két darab 0, amit már nem kell feldolgoznod.

Output

Minden bemeneti számpárra írd ki a tábla sorainak és oszlopainak számát, valamint a szabályosan elhelyezhető huszárok maximális számát a példa outputban látható módon.

Példa input

```
2 3
5 5
4 7
0 0
```

Példa output

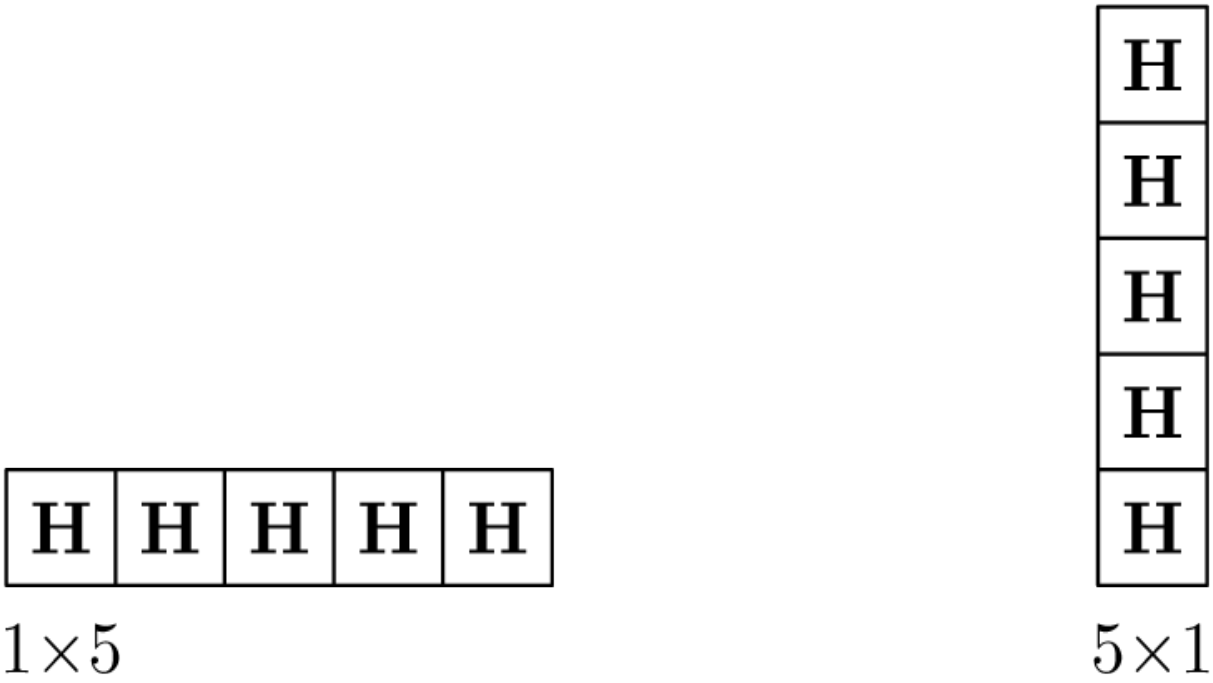
```
4 huszár helyezhető el egy 2 soros és 3 oszlopos táblán.
13 huszár helyezhető el egy 5 soros és 5 oszlopos táblán.
14 huszár helyezhető el egy 4 soros és 7 oszlopos táblán.
```

Megoldás

Mivel a huszár ütéstávolsága az egyik irányban két mezőnyi, ezért külön meg kell vizsgálni azokat a táblákat, ahol a sorok vagy oszlopok száma nem éri el a hármat.

• Ha a tábla vagy csak egy sorból, vagy csak egy oszlopból áll (a másik dimenziója akármekkora lehet), akkor ebbe az egy sorba vagy egy oszlopba pontosan annyi huszár helyezhető el, amekkora a tábla (lásd a 9.2. ábrát).

9.2. ábra -



• Ha valamelyik irányban két mező a tábla mérete, akkor a 9.3. ábrán látható elrendezések lehetségesek. Vegyük észre, hogy ezek az elrendezésminták csak a tábla másik dimenziójának méretétől (illetve annak 4-es maradékától) függnek.

9.3. ábra -

H	H
H	H

$2\times2:\quad 2+\mathbf{2}=4$ darab

H	H			H	H
H	H			H	H

$2\times6:\quad 6+\mathbf{2}=8$ darab

H	H	
H	H	

$2\times3:\quad 3+\mathbf{1}=4$ darab

H	H			H	H	
H	H			H	H	

$2\times7:\quad 7+\mathbf{1}=8$ darab

H	H		
H	H		

$2\times4:\quad 4+\mathbf{0}=4$ darab

H	H			H	H		
H	H			H	H		

$2\times8:\quad 8+\mathbf{0}=8$ darab

H	H			H
H	H			H

$2\times5:\quad 5+\mathbf{1}=6$ darab

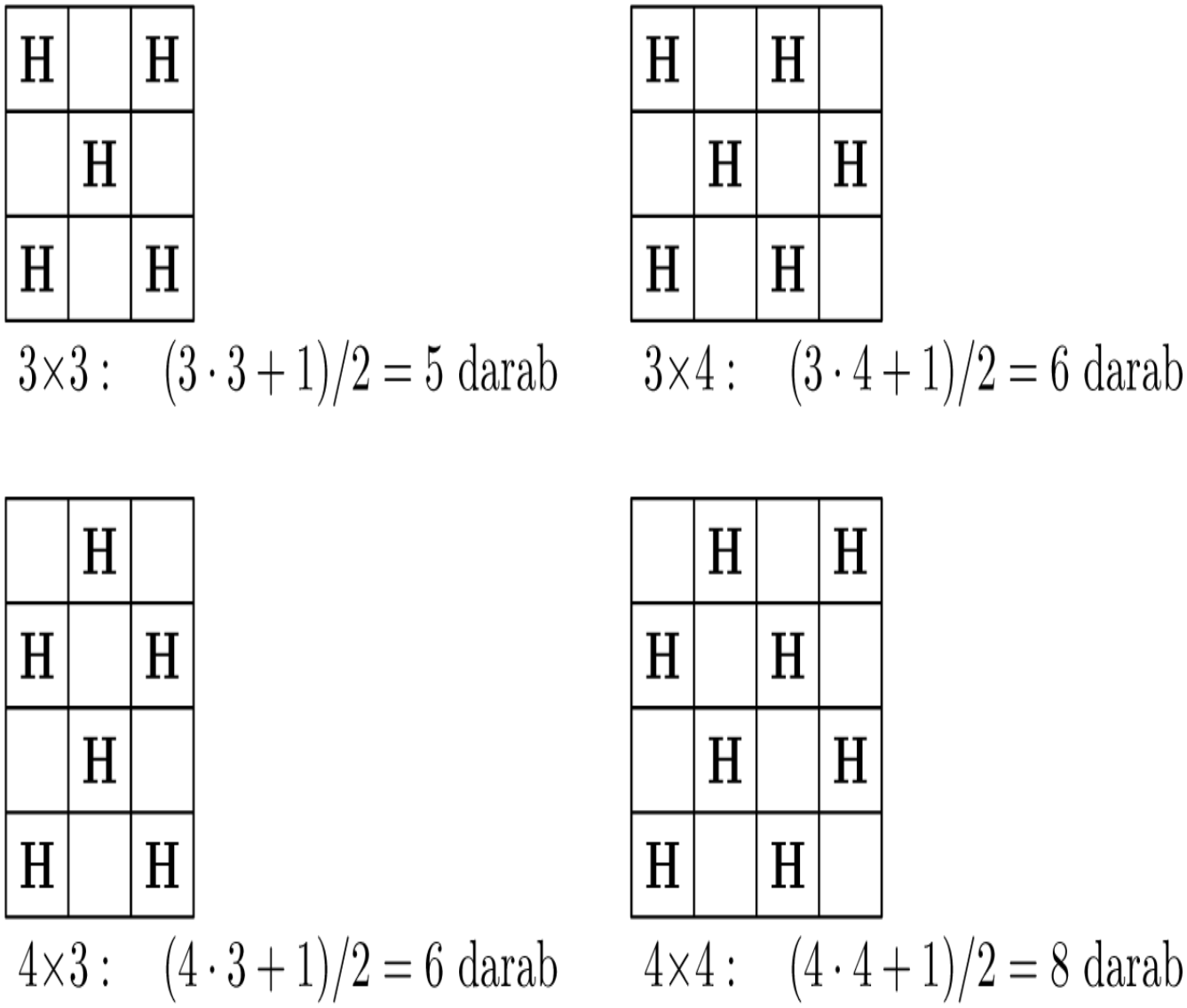
H	H			H	H			H
H	H			H	H			H

$2\times9:\quad 9+\mathbf{1}=10$ darab

• Ha a tábla legalább három sorból és három oszlopból áll (mint az a 9.4. ábrán látható), akkor -- a huszárokat azonos színű mezőkön elhelyezve -- az alábbi képlettel számolhatunk:

$$\frac{sor \cdot oszlop + 1}{2}.$$

9.4. ábra -



Mindezeket figyelembe véve egy lehetséges megoldás a következő:

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

int sor, oszlop, s, o, seged;
```

```
char sztring[ 10 ], *p, plusz[ 4 ] = { 0, 1, 2, 1 };
long lerakhato;

int main()
{
  scanf( "%d %d", &sor, &oszlop );
  while ( sor || oszlop )
  {
    s = sor;
    o = oszlop;
    if ( sor > oszlop )
    {
      seged = sor;
      sor = oszlop;
      oszlop = seged;
    }
    if ( sor == 1 )
      lerakhato = oszlop;
    else if ( sor == 2 )
      lerakhato = oszlop + plusz[ oszlop % 4 ];
    else
      lerakhato = ( sor * oszlop + 1 ) / 2;
    printf( "%ld huszár helyezhető el egy %d soros és "
            "%d oszlopos táblán.\n", lerakhato, s, o );
    scanf( "%d %d", &sor, &oszlop );
  }
  return EXIT_SUCCESS;
}
```

A nyolc királynő problémája

A sakkban el lehet helyezni nyolc királynőt a táblán úgy, hogy egyik se álljon ütésben. Írj programot, amely meghatározza a nyolc királynő összes lehetséges elrendezését, ha adott az egyik királynő pozíciója.

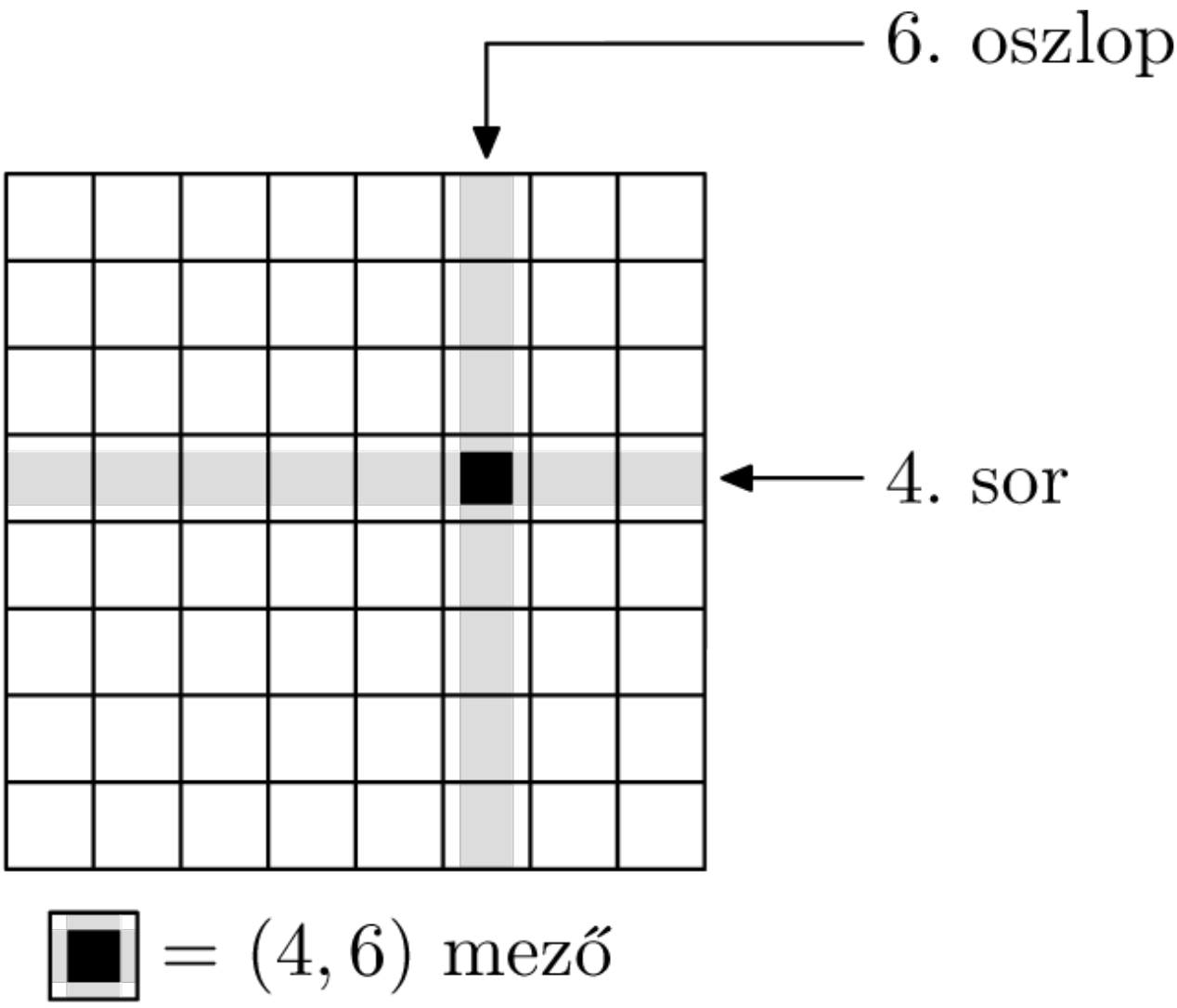
Ne próbálj olyan programot írni, amely a 8 királynő 8 lehetséges helyéből adódó összes állást kiértékeli. Ez 8^8 kiértékelést igényelne, ami térdre kényszerítené a rendszert. A programod futási idejére ésszerű megszorítást fogunk alkalmazni.

Input

A programod bemenete két számból áll, egy szóközzel elválasztva. A számok azt a mezőt jelölik, amelyiken a nyolc királynő egyike elhelyezkedik. Érvényes mezőt adnak meg; a bemenetet nem szükséges ellenőrizni.

Az egységes jelölés érdekében legyen a tábla bal felső sarka az $(1,1)$ pozíció. A sorok vízszintesen követik egymást, a felső sor az első sor. Az oszlopok függőlegesek, és az első oszlop a legbaloldalibb oszlop. Egy mezőre először a sorával, majd az oszlopával hivatkozunk; a $(4,6)$ mező tehát a 4. sor 6. oszlopát jelenti.

9.5. ábra -



Output

A programodnak a bemeneti adatokhoz tartozó összes megoldást elő kell állítania.

A megoldásokat 1-től kezdődően, egyesével haladva sorszámoznod kell. Minden megoldás 8 számból áll (egy-egy szóközzel elválasztva), amelyek az adott megoldáshoz tartozó SOR koordináták. Az oszlop-koordinátákat a nyolc kiírt szám sorrendje fogja megadni. Így tehát az első szám az a SOR, amelyikbe az első oszlopban lévő királynő kerül; a második szám az a SOR, amelyikbe a második oszlopban lévő királynő kerül; stb.

Az alábbi példa inputhoz négy megoldás tartozik. Az egyes megoldásokhoz tartozó 8×8 -as táblák a következők (ez nem azonos az előállítandó kimenettel!):

```
  1. MEGOLDÁS       2. MEGOLDÁS       3. MEGOLDÁS       4. MEGOLDÁS

1 0 0 0 0 0 0 0   1 0 0 0 0 0 0 0   1 0 0 0 0 0 0 0   1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0   0 0 0 0 0 0 1 0   0 0 0 0 0 1 0 0   0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0   0 0 0 1 0 0 0 0   0 0 0 0 0 0 0 1   0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1   0 0 0 0 0 1 0 0   0 0 1 0 0 0 0 0   0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0   0 0 0 0 0 0 0 1   0 0 0 0 0 0 1 0   0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0   0 1 0 0 0 0 0 0   0 0 0 1 0 0 0 0   0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0   0 0 0 0 1 0 0 0   0 1 0 0 0 0 0 0   0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0   0 0 1 0 0 0 0 0   0 0 0 0 1 0 0 0   0 0 0 1 0 0 0 0
```

A példában szereplő 1. megoldás azt mutatja, hogy van egy királynő az 1. sor 1. oszlopában, az 5. sor 2. oszlopában, a 8. sor 3. oszlopában, az 6. sor 4. oszlopában, az 3. sor 5. oszlopában, az 7. sor 6. oszlopában, az 2. sor 7. oszlopában és a 4. sor 8. oszlopában.

Az oszlopfejlécek két sorát is add hozzá a kimenethez, ahogy a példa outputban látható, a megoldásokat pedig lexikografikus sorrendben írd ki.

Példa input

```
1 1
```

Példa output

```
MEGOLDÁS        OSZLOP
   #       1 2 3 4 5 6 7 8

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

Megoldás

Elsőként egy rekurzív algoritmust mutatunk be, amely a visszalépéses megoldáskeresés egy szép példája. A kiinduló helyzete az az állás, amikor a táblán kizárólag a beolvasott pozíción található egy királynő. A táblára a bal oldali oszloptól a jobb oldali oszlopig haladva megpróbálunk egyesével további királynőket elhelyezni úgy, hogy egyik se üssön korábban már a táblára rakott királynőket.

• Ha minden oszlopba sikerült elhelyezni egy-egy királynőt, akkor kiírjuk a megoldást.

• Ha ahhoz az oszlophoz érünk, amelyikbe a kiinduló helyzetben elhelyeztük az első királynőt, akkor ezt az oszlopot érintetlenül hagyjuk, és továbblépünk a következőre.

• Ha üres oszlophoz érünk, akkor minden sort végigpróbálunk. Ha találunk egy olyan sort, ahová a királynő lerakható, akkor oda lerakjuk, és megyünk tovább a következő oszlopra. Amennyiben végigpróbáltuk az összes lehetőséget, levesszük ezt a királynőt a tábláról, és egy korábbi állással folytatjuk a keresést. (Ezt a hívási láncon történő visszalépéssel érjük el.)

A keresés akkor ér véget, amikor a hívási láncon visszatérünk a főprogramba. Ekkorra már a rekurzív hívások során az összes megoldás a kimenetre íródott.

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

#define HAMIS 0
#define IGAZ  ( !HAMIS )

#define N 8

int tabla[ N ];
int sor, oszlop;

int elofeltetel( int sor, int oszlop )
{
  int i;
  for ( i = 0; i < N; ++i )
    if ( tabla[ i ] && ( tabla[ i ] == sor ||
         abs( tabla[ i ] - sor ) == abs( i - oszlop + 1 ) ) )
      return HAMIS;
  return IGAZ;
}
```

```
void kiralyno( int darab )
{
  int i;
  static int sorszam = 0;
  if ( darab == 0 )
  {
    if ( tabla[ oszlop - 1 ] == sor )
    {
      printf( "%4d      ", ++sorszam );
      for ( i = 0; i < N; ++i )
        printf( "%2d", tabla[ i ] );
      putchar( '\n' );
    }
  }
  else if ( tabla[ N - darab ] )
    kiralyno( darab - 1 );
  else
    for ( i = 1; i <= N; ++i )
      if ( elofeltetel( i, N - darab + 1 ) )
      {
        tabla[ N - darab ] = i;
        kiralyno( darab - 1 );
        tabla[ N - darab ] = 0;
      }
}

int main()
{
  scanf( "%d %d", &sor, &oszlop );

  puts( "MEGOLDÁS       OSZLOP" );
  puts( "   #       1 2 3 4 5 6 7 8\n");

  tabla[ oszlop - 1 ] = sor;
  kiralyno( N );

  return EXIT_SUCCESS;
}
```

A rutinos versenyzők találkozhattak már a „nyolc királynő problémával". Talán azzal sem mondunk újat, hogy ennek a problémának ilyen kis N értékre nincs is olyan sok megoldása ($N = 8$ esetén összesen 92). Megfelelő módon tárolva ezeket a megoldásokat, sokat tudunk gyorsítani a keresés algoritmusán.

A következő megoldásunkban a lehetséges állásokat már a kiírandó formátumban tároljuk, sztring literálként. Ezek közül -- a beolvasott pozíciónak megfelelően -- csak ki kell választanunk egy egyszerű mintaillesztéssel a számunkra szükségeseket.

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

const char *allas[] = { "1 5 8 6 3 7 2 4",
                        "1 6 8 3 7 4 2 5",
                        "1 7 4 6 8 2 5 3",
                        "1 7 5 8 2 4 6 3",
                        "2 4 6 8 3 1 7 5",
                        "2 5 7 1 3 8 6 4",
                        "2 5 7 4 1 8 6 3",
                        "2 6 1 7 4 8 3 5",
                        "2 6 8 3 1 4 7 5",
```

"2 7 3 6 8 5 1 4",
"2 7 5 8 1 4 6 3",
"2 8 6 1 3 5 7 4",
"3 1 7 5 8 2 4 6",
"3 5 2 8 1 7 4 6",
"3 5 2 8 6 4 7 1",
"3 5 7 1 4 2 8 6",
"3 5 8 4 1 7 2 6",
"3 6 2 5 8 1 7 4",
"3 6 2 7 1 4 8 5",
"3 6 2 7 5 1 8 4",
"3 6 4 1 8 5 7 2",
"3 6 4 2 8 5 7 1",
"3 6 8 1 4 7 5 2",
"3 6 8 1 5 7 2 4",
"3 6 8 2 4 1 7 5",
"3 7 2 8 5 1 4 6",
"3 7 2 8 6 4 1 5",
"3 8 4 7 1 6 2 5",
"4 1 5 8 2 7 3 6",
"4 1 5 8 6 3 7 2",
"4 2 5 8 6 1 3 7",
"4 2 7 3 6 8 1 5",
"4 2 7 3 6 8 5 1",
"4 2 7 5 1 8 6 3",
"4 2 8 5 7 1 3 6",
"4 2 8 6 1 3 5 7",
"4 6 1 5 2 8 3 7",
"4 6 8 2 7 1 3 5",
"4 6 8 3 1 7 5 2",
"4 7 1 8 5 2 6 3",
"4 7 3 8 2 5 1 6",
"4 7 5 2 6 1 3 8",
"4 7 5 3 1 6 8 2",
"4 8 1 3 6 2 7 5",
"4 8 1 5 7 2 6 3",
"4 8 5 3 1 7 2 6",
"5 1 4 6 8 2 7 3",
"5 1 8 4 2 7 3 6",
"5 1 8 6 3 7 2 4",
"5 2 4 6 8 3 1 7",
"5 2 4 7 3 8 6 1",
"5 2 6 1 7 4 8 3",
"5 2 8 1 4 7 3 6",
"5 3 1 6 8 2 4 7",
"5 3 1 7 2 8 6 4",
"5 3 8 4 7 1 6 2",
"5 7 1 3 8 6 4 2",
"5 7 1 4 2 8 6 3",
"5 7 2 4 8 1 3 6",
"5 7 2 6 3 1 4 8",
"5 7 2 6 3 1 8 4",
"5 7 4 1 3 8 6 2",
"5 8 4 1 3 6 2 7",
"5 8 4 1 7 2 6 3",
"6 1 5 2 8 3 7 4",
"6 2 7 1 3 5 8 4",
"6 2 7 1 4 8 5 3",
"6 3 1 7 5 8 2 4",
"6 3 1 8 4 2 7 5",
"6 3 1 8 5 2 4 7",
"6 3 5 7 1 4 2 8",

```
                        "6 3 5 8 1 4 2 7",
                        "6 3 7 2 4 8 1 5",
                        "6 3 7 2 8 5 1 4",
                        "6 3 7 4 1 8 2 5",
                        "6 4 1 5 8 2 7 3",
                        "6 4 2 8 5 7 1 3",
                        "6 4 7 1 3 5 2 8",
                        "6 4 7 1 8 2 5 3",
                        "6 8 2 4 1 7 5 3",
                        "7 1 3 8 6 4 2 5",
                        "7 2 4 1 8 5 3 6",
                        "7 2 6 3 1 4 8 5",
                        "7 3 1 6 8 5 2 4",
                        "7 3 8 2 5 1 6 4",
                        "7 4 2 5 8 1 3 6",
                        "7 4 2 8 6 1 3 5",
                        "7 5 3 1 6 8 2 4",
                        "8 2 4 1 7 5 3 6",
                        "8 2 5 3 1 7 4 6",
                        "8 3 1 6 2 5 7 4",
                        "8 4 1 3 6 2 7 5" };

int main()
{
  int oszlop, i, sorszam = 0;
  char sor;

  scanf( "%c %d\n", &sor, &oszlop );

  puts( "MEGOLDÁS       OSZLOP" );
  puts( "   #       1 2 3 4 5 6 7 8\n");

  for ( i = 0; i < 92; ++i )
    if ( allas[ i ][ 2 * ( oszlop - 1 ) ] == sor )
      printf( "%4d%22s\n", ++sorszam, allas[ i ] );

  return EXIT_SUCCESS;
}
```

10. fejezet - Dinamikus programozás

Tartalom

Jill kerékpározik

Jill szeret kerékpározni, de mióta a lakhelye, Greenhills szép városa megnőtt, gyakran használja a kitűnő buszközlekedést utazásai során. Van egy összecsukható kerékpárja, ezt viszi magával, ha az útja első részét busszal teszi meg. Amikor a busz a város egy szép részéhez ér, Jill leszáll, és kerékpárral folytatja útját. A busz útvonalát követi, amíg el nem éri a célját, vagy a város olyan részéhez nem ér, amelyet nem szeret. Utóbbi esetben felszáll a buszra, és befejezi az útját.

Az évek tapasztalata alapján Jill minden útszakaszt rangsorolt egy „szépségskálán". A pozitív értékek azokat az útszakaszokat jelölik, amelyeket szeret, és negatív értékeket használ azoknál, amelyeket nem szeret. Nulla értékek nincsenek. Jill megtervezi, hol szálljon le a buszról, és kezdjen kerékpározni, és azt is, hogy hol hagyja abba a kerékpározást, és szálljon vissza a buszra úgy, hogy

azon útszakaszok szépségértékeinek az összege, amelyeket kerékpárral tesz meg, maximális legyen. Ez azt jelenti, hogy néha olyan útszakaszon is kerékpározni fog, amelyet nem szeret, feltéve hogy ez a szakasz az útjának két olyan részét köti össze, amelyeknek bizonyos útszakaszait eléggé szereti ahhoz, hogy ezt ellensúlyozzák. Elképzelhető, hogy az útvonal egyik része sem megfelelő kerékpározásra, ekkor Jill a teljes útvonalat busszal teszi meg. Másrészről viszont az is lehetséges, hogy a teljes útvonal annyira szép, hogy Jill egyáltalán nem fog buszra szállni.

Mivel sok különböző buszjárat létezik, mindegyik számos megállóval, ahol Jill le- és felszállhat, ezért úgy érzi, hogy egy számítógépes program segíthetne neki kiválasztani az egyes buszjáratok útvonalainak kerékpározásra legalkalmasabb részeit.

Input

A bemenet a buszjáratokról tartalmaz információkat. Az első sor egy b egész számot tartalmaz, amely a járatleírások száma. Egy járat r azonosítója a bemeneten elfoglalt helye szerinti sorszám ($1 \leq r \leq b$). Az egyes járatok leírása egy s egész értékkel, a járat útvonalán található megállók számával kezdődik, amely önmagában áll a sorban ($2 \leq s \leq 20000$). A megállók számát $s-1$ sor követi; az i -edik sor ($1 \leq i < s$) egy n_i egészet tartalmaz, amely az i -edik és $(i+1)$ -edik megállók közötti útszakasz szépségének Jill szerinti értékét jelenti.

Output

A programodnak a bemenet minden r járatára ki kell választania azt az i -edik kezdő megállót és j -edik befejező megállót, amelyek a járatnak a maximális szépségösszegű ($m = n_i + n_{i+1} + \ldots + n_{j-1}$) szakaszát jelölik. Ha egynél több maximális szépségű útszakasz létezik, válaszd a leghosszabbat (ahol $j-i$ a legnagyobb). Ha több ilyen is van, válaszd azt a szakaszt, amely a legkorábbi (legkisebb i indexű) megállónál kezdődik. A bemenet minden r járatára írj ki egy sort a következő alakban:

A(z) *r*. járat legszebb része a(z) *i*. és a(z) *j*. megálló között van.

Ha azonban a maximális összeg nem pozitív, a progamod a következőt írja ki:

A(z) *r*. járatnak nincs szép része.

Példa input

```
3
3
  -1
   6
10
   4
  -5
   4
  -3
   4
   4
  -4
   4
  -5
4
  -2
  -3
  -4
```

Példa output

```
A(z) 1. járat legszebb része a(z) 2. és a(z) 3. megálló között van.
```

```
A(z) 2. járat legszebb része a(z) 3. és a(z) 9. megálló között van.
A(z) 3. járatnak nincs szép része.
```

Megoldás

Első megoldásként tekintsük a mezítlábas algoritmust, amely minden kezdő megállót minden befejező megállóval összepárosít, kiszámolja a részösszegeket, és kiválasztja közülük a legnagyobbat, illetve ha több is van, akkor azok közül a leghosszabbat.

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

#define MERET 20000

int szepseg[ MERET ];

int main()
{
  int b, szamlalo;
  scanf( "%d", &b );
  for ( szamlalo = 1; szamlalo <= b; ++szamlalo )
  {
    int megallok, i, kezdo = 1, veg = 1, k, v;
    long maxosszeg = 0;
    scanf( "%d", &megallok );
    for ( i = 1; i < megallok; ++i )
      scanf( "%d", &szepseg[ i ] );
    for ( k = 1; k < megallok; ++k )
    {
      for ( v = k + 1; v <= megallok; ++v )
      {
        long osszeg = 0;
        for ( i = k; i < v; ++i )
          osszeg += szepseg[ i ];
        if ( osszeg > maxosszeg ||
             osszeg == maxosszeg && v - k > veg - kezdo )
        {
          maxosszeg = osszeg;
          kezdo = k;
          veg = v;
        }
      }
    }
    if ( maxosszeg > 0 )
      printf( "A(z) %d. járat legszebb része a(z) %d. és a(z) "
              "%d. megálló között van.\n", szamlalo, kezdo, veg );
    else
      printf( "A(z) %d. járatnak nincs szép része.\n", szamlalo );
  }
  return EXIT_SUCCESS;
}
```

Pár kísérlet után látható, hogy a program nagy s értékekre rendkívül sokáig számol. Hogy gyorsítsuk a számításokat, az adatok beolvasásakor készítünk egy olyan tömböt, amelyben azt tartjuk nyilván, hogy az első megállótól kezdve az i-edik megállóig (ahol $1 \leq i \leq s$) mennyire szép egy-egy útvonal. Ekkor tetszőleges, az i-edik megállótól a j-edik megállóig tartó útvonal szépségét úgy kapjuk meg, hogy kivonjuk a j-edik tömbelem értékéből az i-edikét.

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

```
#define MERET 20000

long szepseg[ MERET + 1 ];

int main()
{
  int b, szamlalo;
  scanf( "%d", &b );
  for ( szamlalo = 1; szamlalo <= b; ++szamlalo )
  {
    int megallok, i, kezdo = 1, veg = 1, k, v, szam;
    long maxosszeg = 0;
    scanf( "%d", &megallok );
    for ( i = 1; i < megallok; ++i )
    {
      scanf( "%d", &szam );
      szepseg[ i + 1 ] = szepseg[ i ] + szam;
    }
    for ( k = 1; k < megallok; ++k )
    {
      for ( v = k + 1; v <= megallok; ++v )
      {
        long osszeg = szepseg[ v ] - szepseg[ k ];
        if ( osszeg > maxosszeg ||
             osszeg == maxosszeg && v - k > veg - kezdo )
        {
          maxosszeg = osszeg;
          kezdo = k;
          veg = v;
        }
      }
    }
    if ( maxosszeg > 0 )
      printf( "A(z) %d. járat legszebb része a(z) %d. és a(z) "
              "%d. megálló között van.\n", szamlalo, kezdo, veg );
    else
      printf( "A(z) %d. járatnak nincs szép része.\n", szamlalo );
  }
  return EXIT_SUCCESS;
}
```

Az előzőnél is gyorsabb megoldást kapunk, ha nem rögzítjük kiinduló megállóként az első megállót, hanem optimista szemlélettel mindaddig hozzáadjuk a következő útszakasz szépségét az eddigiekhez, amíg a részösszeg nem negatív. Ha a részösszeg negatívvá válik, akkor az adott megállótól újrakezdjük az összegzést.

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

int main()
{
  int b, szamlalo;
  scanf( "%d", &b );
  for ( szamlalo = 1; szamlalo <= b; ++szamlalo )
  {
    int megallok, i, kezdo = 1, veg = 1, k, v, szam;
    long maxosszeg = 0;
    scanf( "%d", &megallok );
    k = 1;
    for ( i = 1; i < megallok; ++i )
    {
      long osszeg;
```

```
      scanf( "%d", &szam );
      if ( i == 1 || osszeg < 0 )
      {
        osszeg = szam;
        k = i;
      }
      else
        osszeg += szam;
      if ( osszeg > maxosszeg ||
           osszeg == maxosszeg && i + 1 - k > veg - kezdo )
      {
        maxosszeg = osszeg;
        kezdo = k;
        veg = i + 1;
      }
    }
    if ( maxosszeg > 0 )
      printf( "A(z) %d. járat legszebb része a(z) %d. és a(z) "
              "%d. megálló között van.\n", szamlalo, kezdo, veg );
    else
      printf( "A(z) %d. járatnak nincs szép része.\n", szamlalo );
  }
  return EXIT_SUCCESS;
}
```

A 10.2. fejezetben olvasható feladat ennek a feladatnak az általánosítása kétdimenziós esetre.

Maximális összeg

Háttér

Egy problémát, amelynek egyszerű a megoldása egy dimenzióban, gyakran sokkal bonyolultabb megoldani egynél több dimenzióban. Tekintsük egy konjunktív normálformában lévő logikai kifejezés kielégíthetőségét, amelyben minden egyes konjunkció pontosan 3 diszjunkcióból áll. Ez a probléma (3-SAT) NP-teljes. A 2-SAT probléma viszont elég hatékonyan megoldható. Vannak olyan problémák is, amelyek ugyanabba a bonyolultsági osztályba tartoznak, tekintet nélkül a problémák dimenzióinak a számára.

A probléma

Adott egy pozitív, negatív és nulla értékű egész számokat tartalmazó kétdimenziós tömb, amelynek meg kell keresned a legnagyobb összszegű részmátrixát! Egy részmátrix összege a részmátrixban lévő összes elem összege. Ebben a feladatban a legnagyobb összegű részmátrixra *maximális részmátrixként* hivatkozunk. Egy részmátrix, amely 1×1-es vagy nagyobb méretű összefüggő téglalap alakú területe a mátrixnak, bárhol elhelyezkedhet a teljes mátrixon belül. Például a

```
 0  -2  -7   0
 9   2  -6   2
-4   1  -4   1
-1   8   0  -2
```

mátrix maximális részmátrixa a bal alsó sarokban található, és az összege 15:

```
 9  2
-4  1
-1  8
```

Input és output

A bemenet egy egész számokat tartalmazó $N \times N$-es mátrixból áll. A bemenet egy olyan sorral kezdődik, amely csak egyetlen egy pozitív egész N értéket tartalmaz, ami a kétdimenziós négyzetes mátrix méretét jelzi. Ezt követi N^2 darab egész szám, egymástól fehér karakterekkel (szóközökkel, tabulátorokkal és újsor karakterekkel) elválasztva. Ezt az N^2 egész számot a mátrix sorfolytonosan tartalmazza (azaz az első N szám az első sorban balról jobbra, a következő N szám a második sorban balról jobbra helyezkedik el stb.). N értéke legfeljebb 100 lehet. A mátrixban lévő értékek a $[-127;127]$ tartományba esnek.

A kimenet a maximális részmátrix összege.

Példa input

```
4
0 -2 -7  0 9  2 -6  2
-4  1 -4  1 -1
8  0 -2
```

Példa output

```
15
```

Megoldás

A feladatot többféleképpen is meg lehet oldani. Minden megoldásunkat úgy írjuk meg, hogy mátrixainkat egységesen kezelhessük (1-től N-ig indexelhessük mindkét dimenzióban). Kezdjük talán a legegyszerűbb, mezítlábas algoritmussal, amelyik a mátrix elemeinek beolvasása után négy, egymásba ágyazott ciklussal végighalad a lehetséges részmátrixok bal felső és jobb alsó sarkain, aztán újabb két -- az előzőekbe és -- egymásba ágyazott ciklussal összeadja az éppen vizsgált részmátrix elemeit, és ezek közül kiválasztja és megjegyzi a maximális részmátrix összegét.

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

#define MERET 100

int matrix[ MERET + 1 ][ MERET + 1 ];

int main()
{
  int n, bal, felso, jobb, also, i, j;
  long max;

  scanf( "%d", &n );
  for ( i = 1; i <= n; ++i )
    for ( j = 1; j <= n; ++j )
      scanf( "%d", &matrix[ i ][ j ] );
  max = matrix[ 1 ][ 1 ];
  for ( bal = 1; bal <= n; ++bal )
    for ( felso = 1; felso <= n; ++felso )
      for ( jobb = bal; jobb <= n; ++jobb )
        for ( also = felso; also <= n; ++also )
        {
          long osszeg = 0;
          for ( i = bal; i <= jobb; ++i )
            for ( j = felso; j <= also; ++j )
              osszeg += matrix[ i ][ j ];
          if ( osszeg > max )
            max = osszeg;
        }
```

```
  printf( "%ld\n", max );
  return EXIT_SUCCESS;
}
```

Ez a megoldás -- különösen nagyméretű mátrixok esetén -- rendkívül lassú. Keressünk hát másik, hatékonyabb megoldást. Az eredeti mátrix beolvasása után építsünk fel egy másik mátrixot, amelynek az (i,j) koordinátapárral jelölt eleme azt adja majd meg, hogy -- a mátrixok sorait és oszlopait 1-től N -ig számozva -- az eredeti mátrix $(1,1)$ bal felső sarkú és (i,j) jobb alsó sarkú részmátrixában mennyi az elemek összege. A példaként megadott mátrix esetén ez a következőképpen fog kinézni:

Eredeti:

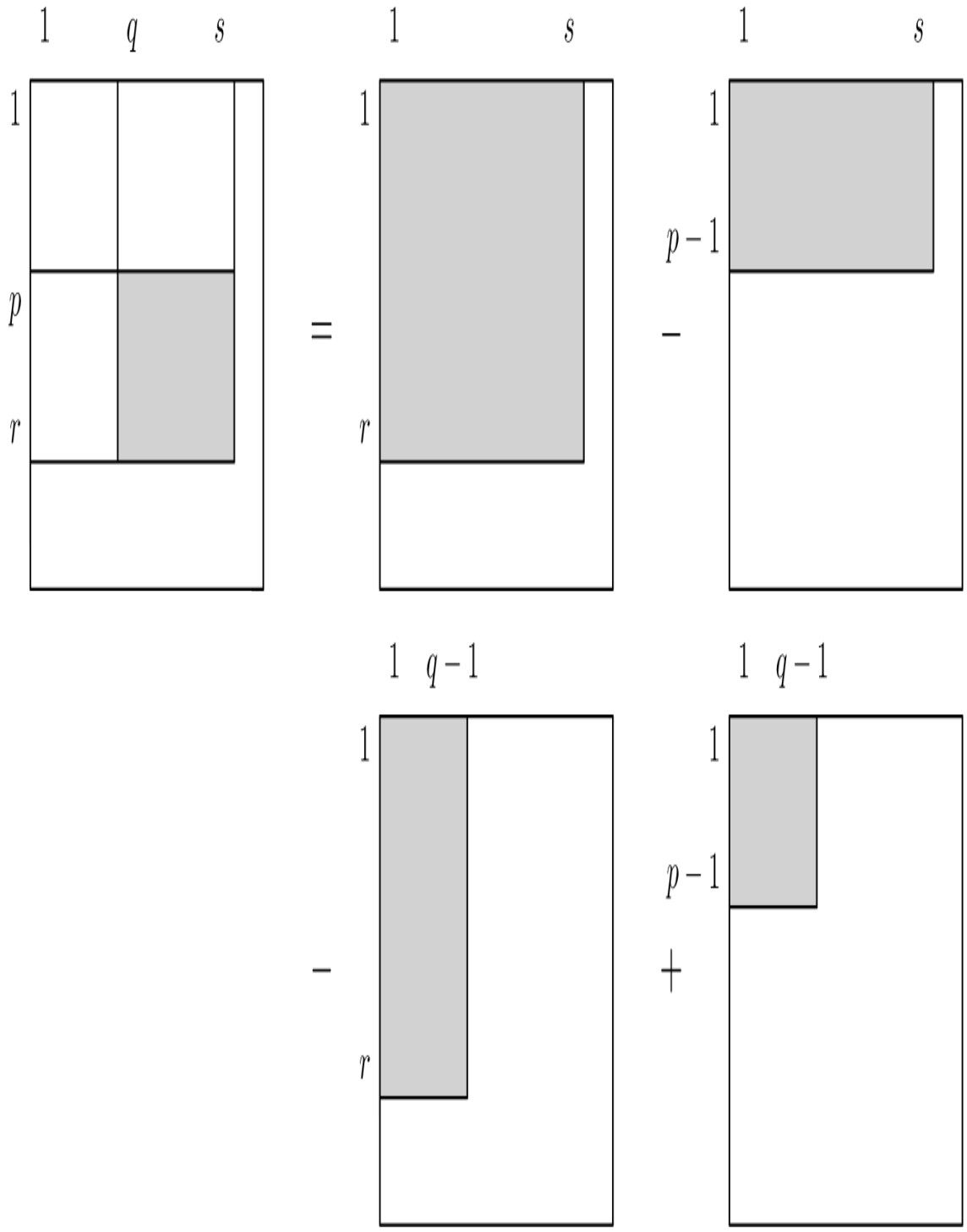
0	−2	−7	0
9	2	−6	2
−4	1	−4	1
−1	8	0	−2

Új:

0	−2	−9	−9
9	9	−4	−2
5	6	−11	−8
4	13	−4	−3

Hogyan tudjuk ebből az új mátrixból egy tetszőleges (p,q) bal felső és (r,s) jobb alsó koordinátájú részmátrix összegét meghatározni? Ehhez tekintsük a 10.1. ábrát.

10.1. ábra -



Látható, hogy bármely részmátrix összege kiszámítható négy olyan részmátrix összegének az ismeretében, amelyeknek a bal felső koordinátája az $(1,1)$. Az előzőleg definiált mátrixunkban pontosan ilyen típusú részmátrixok összegét tároljuk, így onnan könnyen ki tudjuk olvasni a keresett összegeket. A programunk a következőképpen egyszerűsödik:

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

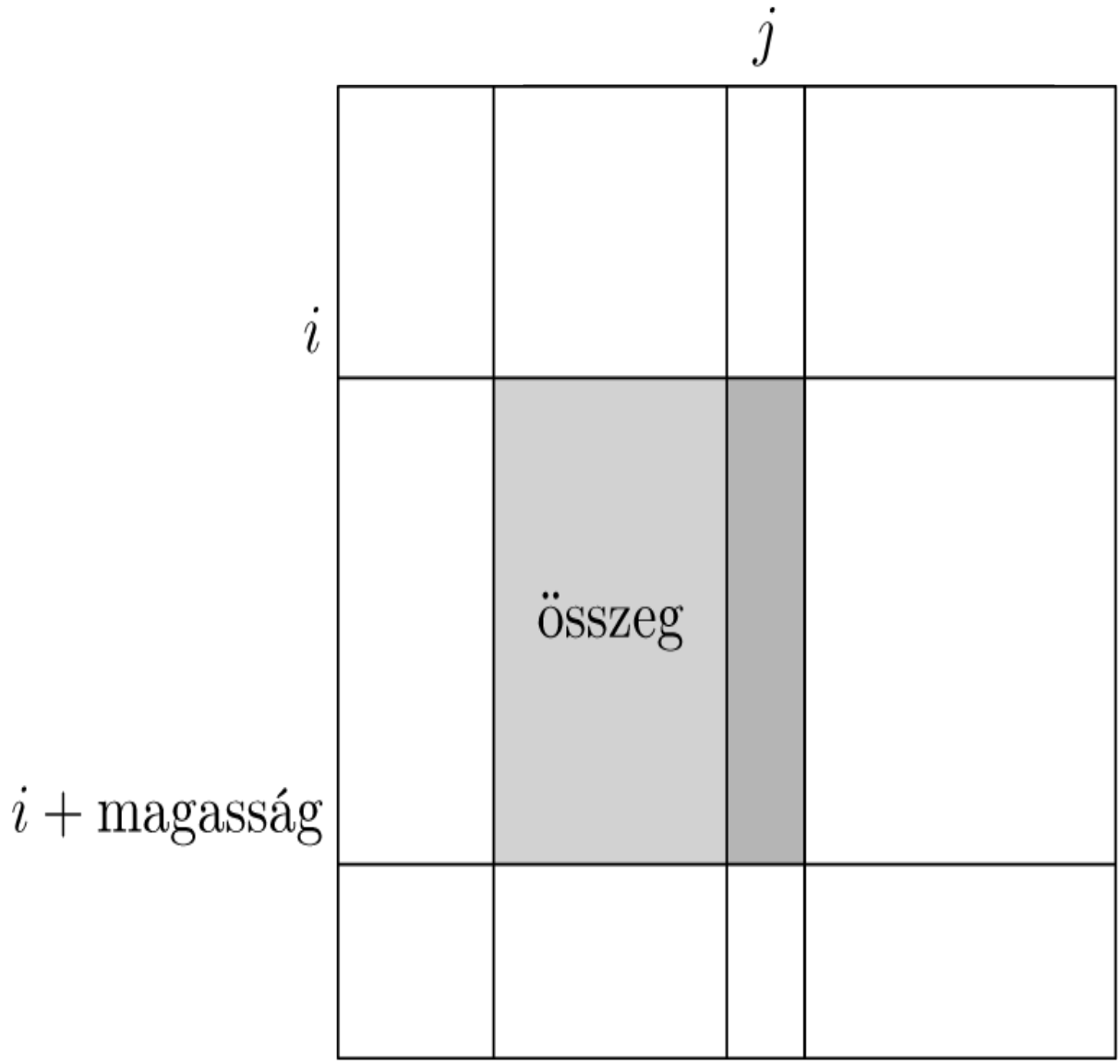
```
#include <stdlib.h>

int matrix[ MERET + 1 ][ MERET + 1 ];

int main()
{
  int n, bal, felso, jobb, also, i, j;
  long max;

  scanf( "%d", &n );

  for ( i = 1; i <= n; ++i )
    for ( j = 1; j <= n; ++j )
    {
      scanf( "%d", &matrix[ i ][ j ] );
      matrix[ i ][ j ] += matrix[ i - 1 ][ j ] + matrix[ i ][ j - 1 ]
                        - matrix[ i - 1 ][ j - 1 ];
    }

  max = matrix[ 1 ][ 1 ];

  for ( bal = 1; bal <= n; ++bal )
    for ( felso = 1; felso <= n; ++felso )
      for ( jobb = bal; jobb <= n; ++jobb )
        for ( also = felso; also <= n; ++also )
        {
          long osszeg = matrix[ jobb ][ also ] -
                        matrix[ jobb ][ felso - 1 ] -
                        matrix[ bal - 1 ][ also ] +
                        matrix[ bal - 1 ][ felso - 1 ];
          if ( osszeg > max )
            max = osszeg;
        }

  printf( "%ld\n", max );

  return EXIT_SUCCESS;
}
```

Ez már egészen jó megoldás, azonban még mindig nem az igazi. Ugyanis még tovább gyorsíthatjuk a keresést, ha nem ragaszkodunk ahhoz, hogy tetszőleges részmátrixaink összegét mindig $(1,1)$ bal felső koordinátájú részmátrixok összegének lineáris kombinációjaként határozzuk meg. Készítsünk el egy olyan mátrixot, ahol az (i,j) koordinátájú elem azt fogja megadni, hogy mennyi a j-edik oszlopban az első sortól az i-edik sorig található elemek összege. A példaként megadott mátrix esetén ez a következőképpen fog kinézni:

Eredeti: $\begin{matrix} 0 & -2 & -7 & 0 \\ 9 & 2 & -6 & 2 \\ -4 & 1 & -4 & 1 \\ -1 & 8 & 0 & -2 \end{matrix}$ Új: $\begin{matrix} 0 & -2 & -7 & 0 \\ 9 & 0 & -13 & 2 \\ 5 & 1 & -17 & 3 \\ 4 & 9 & -17 & 1 \end{matrix}$

Hogyan határozzuk meg ennek a mátrixnak a segítségével a maximális részmátrix összegét?

10.2. ábra -



A mátrixban N különböző magasságú részmátrixot tudunk képezni. Egy konkrét magasság esetén $N \times (N - \text{magasság} + 1)$ darab részmátrix létezik. Az adott részmátrixok összegét úgy számítjuk, hogy haladunk a mátrix első oszlopától az N-edik oszlopáig, és azt vizsgáljuk, hogy az addig kiszámított részösszeg nemnegatív-e. Ha nemnegatív, akkor az aktuális részoszlop összegét is hozzáadjuk (reménykedve, hogy még nagyobb összeget kapunk). Ha negatív az eddigi részösszeg, akkor a részösszeg számítását az aktuális (j-edik) oszloptól újrakezdjük.

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

int matrix[ MERET + 1 ][ MERET + 1 ];

int main()
{
  int n, i, j, magassag;
  long max;

  scanf( "%d", &n );
```

```
  for ( i = 1; i <= n; ++i )
    for ( j = 1; j <= n; ++j )
    {
      scanf( "%d", &matrix[ i ][ j ] );
      matrix[ i ][ j ] += matrix[ i - 1 ][ j ];
    }
  max = matrix[ 1 ][ 1 ];
  for ( magassag = 1; magassag <= n; ++magassag )
    for ( i = 0; i <= n - magassag; ++i )
    {
      long osszeg = 0;
      for ( j = 1; j <= n; ++j )
      {
        if ( osszeg >= 0 )
          osszeg += matrix[ i + magassag ][ j ] - matrix[ i ][ j ];
        else
          osszeg = matrix[ i + magassag ][ j ] - matrix[ i ][ j ];
        if ( osszeg > max )
          max = osszeg;
      }
    }
  printf( "%ld\n", max );
  return EXIT_SUCCESS;
}
```

11. fejezet - Labirintus

Tartalom

Az útvonal feltérképezése

Egy labirintusból kivezető út megtalálása népszerű számítógépes feladat. Ebben a feladatban a labirintust négyzet alakú cellák téglalap alakú tömbje alkotja, amelyben minden cellának lehet fala északról, délről, keletről és/vagy nyugatról. Az egyik cella lesz a kezdőpont, egy másik pedig a végpont. A feladatod, hogy megtalálj egy utat a kezdőpontból a végpontba, megcímkézd az útvonal minden egyes celláját a cella útvonalbeli sorszámával, azonosítsd azokat a cellákat, amelyeket érintettél, de nincsenek az útvonalon, és kirajzold a labirintust.

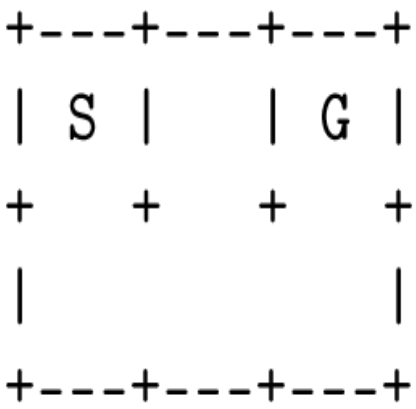
A labirintusban az útvonal megtalálásához használt algoritmusnak az alább ismertetettnek kell lennie. Tegyük fel, hogy egy robot a kezdő cellában van. A robot először nyugatra próbál menni abból a cellából, majd északra, majd keletre, majd délre. A robot akkor mozoghat a kiválasztott irányba, ha

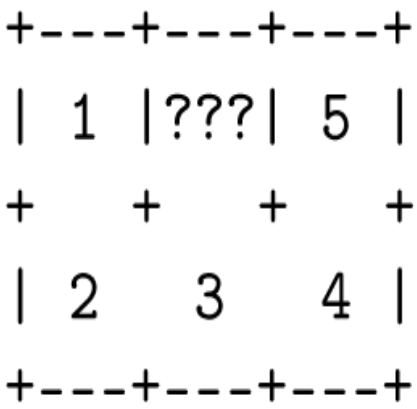
1. nincs fal, amely megakadályozná, hogy abba az irányba mozogjon, és

2. még nem volt az adott irány szerinti szomszédos cellában.

Ha a robot eléri a célt, az útja véget ér. Ha a robot egy olyan cellába ér, amelyből nem lehet továbbhaladni, visszatér abba a cellába, ahonnan idejött, és megkísérel a következő, még nem próbált irányba mozogni.

11.1. ábra -





Vegyük például a 11.1. ábra bal oldalán látható egyszerű labirintust, amely két cella magas és három cella széles. A kezdő cella S-sel, a célcella G-vel van jelölve. Amikor a robot elindul, először nyugatra (balra) próbál mozogni, de ott falat talál. Ezután megpróbál északra (felfelé) menni, de arra is falba ütközik. Szintén egy fal akadályozza meg keleti irányú (jobbra történő) mozgását is. Ezért végül délre (lefelé) próbál mozogni, ami sikerül. Az új cellából végül keletre megy tovább. Itt megismétli a mozgási algoritmusát. Bár nyugati irányban, amerre továbbhaladhatna, nincsen fal, már érintette az abban az irányban lévő cellát, ezért megpróbál északra menni, ami sikerül. Sajnos, miután északra mozgott, nem tudja tovább folytatni az útját, ezért visszalép abba a cellába, ahol előzőleg állt. Ezután megpróbál kelet felé menni, ami sikerül. Abból a cellából észak felé megy, ahol megtalálja a célt. A kimeneten megjelenítendő labirintus a 11.1. ábra jobb oldalán látható. Figyeld meg, hogy a kezdő cella 1-essel van jelölve, az útvonal minden cellája a célig (beleértve a célt tartalmazó cellát is) a sorszámával van jelölve, és minden cella, amelyet érintettünk, de nincs az útvonalon, kérdőjellel van jelölve.

Input

Tekintsük a labirintust cellák tömbjeként, amelyben a legészakibb sor az első sor, a legnyugatibb oszlop pedig az első oszlop. A fenti labirintusban a kezdő cella az első sor első oszlopa, a célcella pedig az első sor harmadik oszlopa.

A bemeneten egy vagy több feldolgozandó labirintus található. Minden egyes labirintus esetén először hat egész számot kell beolvasni. Az első kettő a labirintus magasságát (sorainak számát) és szélességét (oszlopainak számát) adja meg (cellában számolva). A következő kettő a kezdő cella pozíciója (sor- és oszlopszám), az utolsó kettő pedig a cél pozíciója. Egyik labirintusnak sem lesz 12-nél több sora, vagy 12-nél több oszlopa, és mindig létezik útvonal a kezdőponttól a célig.

Az első hat egész számot követően minden egyes cellához beolvasunk egy-egy egész számot sorfolytonosan. Az egész számok értéke azt jelzi, hogy van-e a cellának fala a keleti oldalán (1), és hogy van-e fala a déli oldalán (2). Például ha egy cellának nincs se keleti, se déli fala, akkor a hozzá tartozó érték 0. Ha a cellának csak déli fala van, akkor a hozzá tartozó érték 2. Ha a cellának keleti és déli fala is van, akkor a hozzá tartozó érték 3. A labirintus szélein található celláknak mindig van fala a megfelelő oldalon, hogy megakadályozzák, hogy a robot elhagyhassa a labirintust; ezek nincsenek megadva a bemeneti adatok között.

Az utolsó labirintust hat darab nulla követi.

Output

Minden egyes labirintus esetén rajzold ki a labirintust a fenti példában és a példa outputban látható módon, megfelelően felcímkézve és sorszámmal ellátva az egyes labirintusokat. A labirintusokat 1-től indulva folyamatosan sorszámozd.

Példa input

```
2 3 1 1 1 3
1 1 0
0 0 0

4 3 3 2 4 3
0 3 0
0 2 0
0 3 0
0 1 0

0 0 0 0 0 0
```

Példa output

```
Maze 1

+---+---+---+
| 1 |???| 5 |
+   +   +   +
| 2   3   4 |
+---+---+---+

Maze 2

+---+---+---+
|??? ???|???|
+   +---+   +
| 3   4   5 |
+   +---+   +
| 2   1 | 6 |
+   +---+   +
|       | 7 |
+---+---+---+
```

Megoldás

A feladat visszalépéses kereséssel (backtrack algoritmussal) oldható meg. Kétféle megoldást is adunk rá, egy iteratívat és egy rekurzívat. Mindkét megoldásban kihasználjuk, hogy a labirintus maximum 12×12 cellából áll, és az ehhez szükséges tömböket (`lab` és `falak`) már a programok elején deklaráljuk. Az iteratív megoldásban a visszalépéses keresés adminisztratív elemeinek a nyilvántartására egy külön struktúrát hozunk létre:

```
struct cella {
  int szam;
  int tovabb;
  int merrol;
};
```

Ebben a struktúrában tartjuk majd nyilván

• a cella bejárási (elérési) sorrend szerinti sorszámát (ami 0 lesz, ha még nem érintettük a cellát, pozitív, ha éppen a cél felé vezető úton szerepel, és negatív, ha csak bekukkantottunk ide a cél keresése közben, de ez a cella nem lesz rajta a keresett útvonalon),

• a cellából a következő kipróbálandó irány kódja (`NYUGAT`, `ESZAK`, `KELET` és `DEL`, attól függően, hogy hány irányba léptünk már tovább, és mennyi van még hátra), valamint

• annak az iránynak a kódja, ahonnan az adott cellába érkeztünk. Ezt azért tartjuk nyilván, hogy ha nem tudnánk továbbhaladni, akkor tudjuk, hogy merre kell visszamennünk.

Visszalépéskor a megelőző cella koordinátáinak a kiszámítását könnyíti meg az `irany` nevű kétdimenziós tömb, amely keletről történt érkezéskor nyugati, északról történt érkezéskor déli, nyugatról történt érkezéskor keleti, és délről történt érkezéskor északi irányba fogja növelni vagy csökkenteni az aktuális sor- és oszlopkoordinátáinkat (`x`-et és `y`-t).

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

#define MERET 12

enum { NYUGAT, ESZAK, KELET, DEL };

int mag, szel, s_sor, s_oszlop, c_sor, c_oszlop;
int falak[ MERET ][ MERET ];

struct cella
{
  int szam;
  int tovabb;
  int merrol;
} lab[ MERET ][ MERET ];

int irany[ 4 ][ 2 ] = { { 0, -1 }, { -1, 0 }, { 0, 1 }, { 1, 0 } };
int teszteset = 0, i, j, x, y, szamlalo;

int main()
{
  scanf( "%d %d %d %d %d %d",
         &mag, &szel, &s_sor, &s_oszlop, &c_sor, &c_oszlop );
  while ( mag || szel || s_sor || s_oszlop || c_sor || c_oszlop )
  {
    printf( "Maze %d\n\n", ++teszteset );

    for ( i = 0; i < mag; ++i )
      for ( j = 0; j < szel; ++j )
      {
        scanf( "%d", &falak[ i ][ j ] );
        lab[ i ][ j ].szam = 0;
        lab[ i ][ j ].tovabb = NYUGAT;
      }

    lab[ x = s_sor - 1 ][ y = s_oszlop - 1 ].szam = szamlalo = 1;
    while ( lab[ c_sor - 1 ][ c_oszlop - 1 ].szam == 0 )
    {
      switch ( lab[ x ][ y ].tovabb )
      {
        case NYUGAT: ++lab[ x ][ y ].tovabb;
                     if ( y > 0 && !( falak[ x ][ y-1 ] & 1 ) &&
                          !lab[ x ][ y-1 ].szam )
                     {
                       --y;
                       lab[ x ][ y ].szam = ++szamlalo;
                       lab[ x ][ y ].merrol = KELET;
                     }
                     break;
        case ESZAK:  ++lab[ x ][ y ].tovabb;
                     if ( x > 0 && !( falak[ x-1 ][ y ] & 2 ) &&
                          !lab[ x-1 ][ y ].szam )
                     {
                       --x;
                       lab[ x ][ y ].szam = ++szamlalo;
                       lab[ x ][ y ].merrol = DEL;
```

```
                    }
                    break;
        case KELET:  ++lab[ x ][ y ].tovabb;
                    if ( y < szel-1 && !( falak[ x ][ y ] & 1 ) &&
                         !lab[ x ][ y+1 ].szam )
                    {
                      ++y;
                      lab[ x ][ y ].szam = ++szamlalo;
                      lab[ x ][ y ].merrol = NYUGAT;
                    }
                    break;
        case DEL:    ++lab[ x ][ y ].tovabb;
                    if ( x < mag-1 && !( falak[ x ][ y ] & 2 ) &&
                         !lab[ x+1 ][ y ].szam )
                    {
                      ++x;
                      lab[ x ][ y ].szam = ++szamlalo;
                      lab[ x ][ y ].merrol = ESZAK;
                    }
                    break;
        default:     lab[ i = x ][ j = y ].szam *= -1;
                    --szamlalo;
                    x += irany[ lab[ i ][ j ].merrol ][ 0 ];
                    y += irany[ lab[ i ][ j ].merrol ][ 1 ];
                    break;
      }
    }

    putchar( '+' );
    for ( j = 0; j < szel; ++j )
      printf( "---+" );
    putchar( '\n' );
    for ( i = 0; i < mag; ++i )
    {
      putchar( '|' );
      for ( j = 0; j < szel; ++j )
      {
        if ( lab[ i ][ j ].szam > 0 )
          printf( "%3d", lab[ i ][ j ].szam );
        else
          printf( lab[ i ][ j ].szam ? "???" : "   " );
        putchar( j == szel-1 || ( falak[ i ][ j ] & 1 ) ? '|' : ' ' );
      }
      printf( "\n+" );
      for ( j = 0; j < szel; ++j )
        printf( i == mag-1 || ( falak[ i ][ j ] & 2 ) ?
                "---+" : "   +" );
      putchar( '\n' );
    }
    putchar( '\n' );

    scanf( "%d %d %d %d %d %d",
           &mag, &szel, &s_sor, &s_oszlop, &c_sor, &c_oszlop );
  }
  return EXIT_SUCCESS;
}
```

A rekurzív megoldás hasonlóképpen dolgozik: a `lab` tömbben pozitív érték jelzi egy cellában, hogy az adott cella része a célba vezető útvonalnak, 0, ha nem jártunk a cellában, és negatív, ha a cella nem része a keresett útnak.

Alapértelmezés szerint az érintett cellák negatív sorszámokat kapnak, hiszen azt, hogy melyik cella

lesz része a célba vezető útvonalnak, leghamarabb csak akkor tudjuk eldönteni, ha már elértük a célcellát. Ekkortól viszont -- a hívási láncon visszafelé haladva -- minden érintett cella sorszámát megszorozzuk -1-gyel, így kapjuk meg a keresett útvonalat. Persze amíg nem találjuk meg a célcellát, a robot algoritmusa szerint nyugati, északi, keleti és déli irányokban próbálkozunk a továbbhaladással.

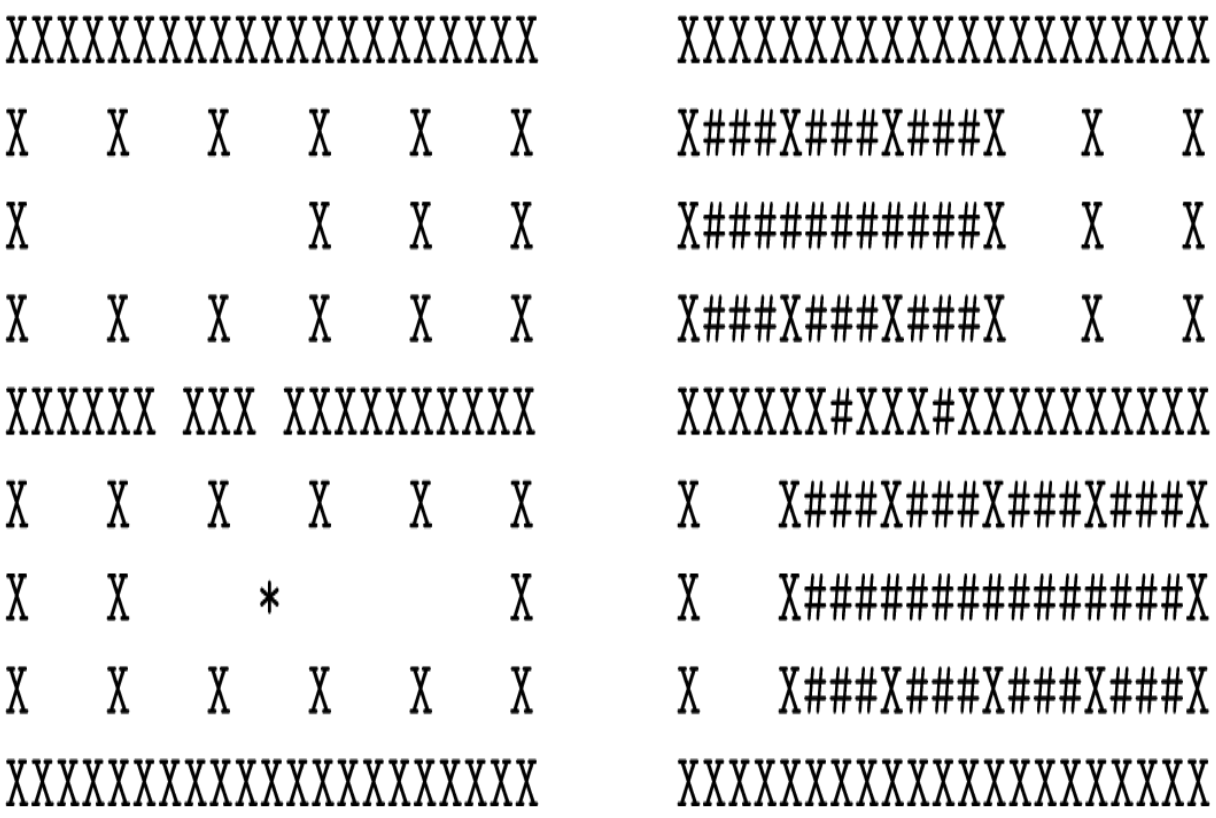
```
#include <stdio.h>
#include <stdlib.h>

#define MERET 12
#define HAMIS  0

int lab[ MERET ][ MERET ], falak[ MERET ][ MERET ];

int mag, szel, s_sor, s_oszlop, c_sor, c_oszlop;
int teszteset = 0, i, j, cel;

void halad( int x, int y, int c )
{
  if ( x < 0 || x >= mag || y < 0 || y >= szel )
    return;
  lab[ x ][ y ] = c;
  if ( x == c_sor - 1 && y == c_oszlop - 1 )
  {
    cel = !HAMIS;
    lab[ x ][ y ] *= -1;
    return;
  }
  if ( !cel && !( falak[ x ][ y - 1 ] & 1 ) && !lab[ x ][ y - 1 ] )
    halad( x, y - 1, c - 1 );
  if ( !cel && !( falak[ x - 1 ][ y ] & 2 ) && !lab[ x - 1 ][ y ] )
    halad( x - 1, y, c - 1 );
  if ( !cel && !( falak[ x ][ y ] & 1 ) && !lab[ x ][ y + 1 ] )
    halad( x, y + 1, c - 1 );
  if ( !cel && !( falak[ x ][ y ] & 2 ) && !lab[ x + 1 ][ y ] )
    halad( x + 1, y, c - 1 );
  if ( cel )
    lab[ x ][ y ] *= -1;
}

int main()
{
  scanf( "%d %d %d %d %d %d",
         &mag, &szel, &s_sor, &s_oszlop, &c_sor, &c_oszlop );
  while ( mag || szel || s_sor || s_oszlop || c_sor || c_oszlop )
  {
    for ( i = 0; i < mag; ++i )
      for ( j = 0; j < szel; ++j )
        lab[ i ][ j ] = 0;
    printf( "Maze %d\n\n", ++teszteset );
    for ( i = 0; i < mag; ++i )
      for ( j = 0; j < szel; ++j )
        scanf( "%d", &falak[ i ][ j ] );
    cel = HAMIS;
    halad( s_sor - 1, s_oszlop - 1, -1 );
    putchar( '+' );
    for ( j = 0; j < szel; ++j )
      printf( "---+" );
    putchar( '\n' );
    for ( i = 0; i < mag; ++i )
    {
```

```
      for ( j = 0; j < szel; ++j )
      {
        putchar( j == 0 || ( falak[ i ][ j - 1 ] & 1 ) ? '|' : ' ' );
        if ( lab[ i ][ j ] > 0 )
          printf( "%3d", lab[ i ][ j ] );
        else
          printf( lab[ i ][ j ] ? "???" :"   " );
      }
      printf( "|\n+" );
      for ( j = 0; j < szel; ++j )
        printf( "%s+", i == mag - 1 || ( falak[ i ][ j ] & 2 ) ?
                "---" : "   " );
      putchar( '\n' );
    }
    putchar( '\n' );
    scanf( "%d %d %d %d %d %d",
           &mag, &szel, &s_sor, &s_oszlop, &c_sor, &c_oszlop );
  }
  return EXIT_SUCCESS;
}
```

Labirintus

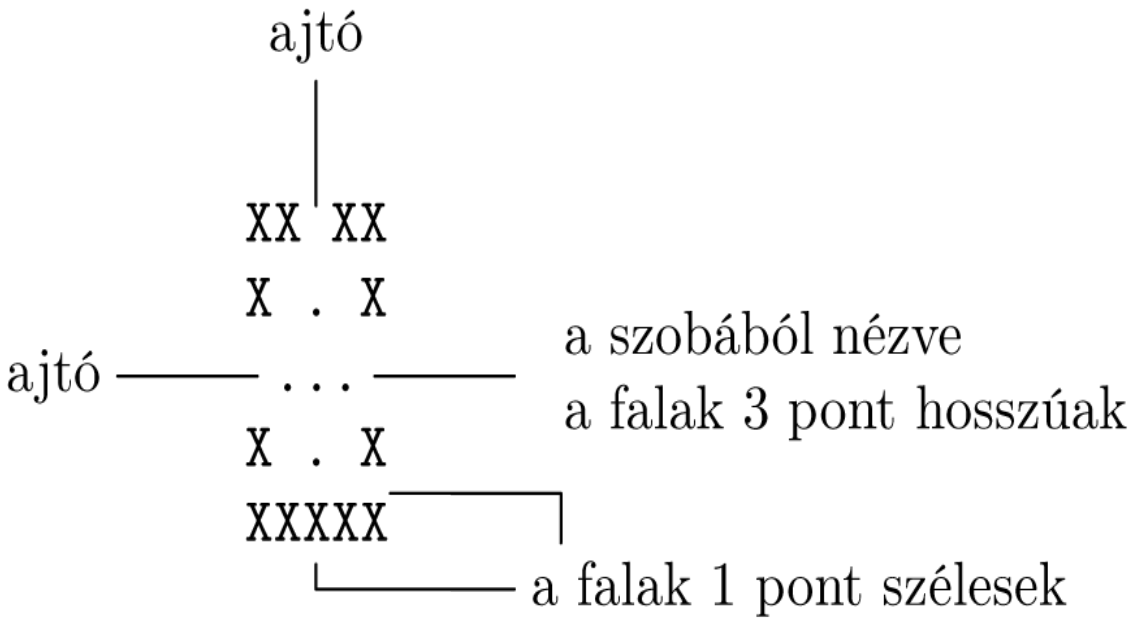
Egy négyzet alakú szobákból álló labirintust egy kétdimenziós ráccsal ábrázolhatunk, ahogy az a 10. ábra bal oldalán látható. A rács minden pontja egyetlen karakter. A szobák falainak a pontjai azonos karakterekkel vannak jelölve, amely bármely nyomtatható karakter lehet a '*', a '#' és a szóköz karaktereken kívül. A 10. ábrán ez a karakter az 'X'. A rács minden más pontja szóköz.

11.2. ábra -



A labirintus minden szobája azonos méretű, minden fal 3 pont hosszú és 1 pont széles, ahogy a 11.3. ábrán látható. Az egymás melletti szobák falai teljes hosszukban közösek. A szobákat ajtók köthetik össze, amelyek a falak közepén helyezkednek el. Nincs a szabadba kivezető ajtó.

11.3. ábra -



A feladatod az, hogy fesd be a labirintus minden olyan szobáját, amelyet be lehet járni egy adott szobából kiindulva, amelyet „kezdő szobának" nevezünk, és a szoba közepén elhelyezett csillaggal ('*') jelölünk (lásd a 11.2. ábra bal oldalát). Egy szobába átmehetünk egy másik szobából, ha a két szobát elválasztó falon van ajtó. Egy szoba ki van festve, ha a teljes felülete -- beleértve az ajtókat is -- a '#' karakterrel van jelölve, ahogy az a 11.2. ábra jobb oldalán látható.

Input

A bemenet a következőképpen néz ki:

1. Az első sor egy pozitív egész számot tartalmaz, a kifestendő labirintusok számát.

2. A bemenet hátralévő része a labirintusokat írja le.

A bemenet sorai különböző hosszúságúak lehetnek. A labirintust leíró szövegrészek egy-egy elválasztó sorral vannak lezárva, amelyek kizárólag aláhúzásjelet (_) tartalmaznak. Maximum 30 sor és soronként maximum 80 karakter tartozik egy labirintushoz.

A program olvassa be a labirintust a bemenetről, fesse ki, és írja ki a kifestett labirintust a kimenetre.

Output

A kifestett labirintusoknak ugyanaz a formája, mint a beolvasottaké, beleértve az elválasztó sorokat is. Az alábbi példában egy egyszerű bemenet látható, amely egyetlen labirintusból és a hozzá tartozó kimenetből áll.

Példa input

```
1
```

```
XXXXXXXXX
X   X   X
X *     X
X   X   X
XXXXXXXXX
X   X
X   X
X   X
XXXXX
```

Példa output

```
XXXXXXXXX
X###X###X
X#######X
X###X###X
XXXXXXXXX
X   X
X   X
X   X
XXXXX
```

Megoldás

A feladat legegyszerűbben rekurzívan oldható meg. A labirintust leíró sorok beolvasása után megkeressük a kiinduló szobát, kifestjük, majd az onnan elérhető szobákba lépünk tovább, mind a négy irányban próbálkozva. Azokkal a szobákkal, ahol már jártunk, nem foglalkozunk (hiszen már kifestettük őket).

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

#define SOROK 30
#define HOSSZ 80

char labirintus[ SOROK + 1 ][ HOSSZ + 1 ], *p;
int sor, i, j, teszteset, xpoz, ypoz;

void festes( int x, int y )
{
  if ( labirintus[ y ][ x ] == '#' )
    return;

  strncpy( &labirintus[ y - 1 ][ x - 1 ], "###", 3 );
  strncpy( &labirintus[ y ][ x - 1 ], "###", 3 );
  strncpy( &labirintus[ y + 1 ][ x - 1 ], "###", 3 );

  if ( labirintus[ y ][ x - 2 ] == ' ' )
  {
    labirintus[ y ][ x - 2 ] = '#';
    festes( x - 4, y );
  }
  if ( labirintus[ y - 2 ][ x ] == ' ' )
  {
    labirintus[ y - 2 ][ x ] = '#';
    festes( x, y - 4 );
  }
  if ( labirintus[ y ][ x + 2 ] == ' ' )
  {
```

```
    labirintus[ y ][ x + 2 ] = '#';
    festes( x + 4, y );
  }
  if ( labirintus[ y + 2 ][ x ] == ' ' )
  {
    labirintus[ y + 2 ][ x ] = '#';
    festes( x, y + 4 );
  }
}

int main()
{
  scanf( "%d\n", &teszteset );
  while ( teszteset-- )
  {
    gets( labirintus[ 0 ] );
    gets( labirintus[ 1 ] );
    gets( labirintus[ 2 ] );
    gets( labirintus[ 3 ] );
    gets( labirintus[ 4 ] );
    gets( labirintus[ sor = 5 ] );
    while ( labirintus[ sor ][ 0 ] != '_' )
    {
      gets( labirintus[ ++sor ] );
      gets( labirintus[ ++sor ] );
      gets( labirintus[ ++sor ] );
      gets( labirintus[ ++sor ] );
    }

    for ( i = 2; i < sor; i += 4 )
    {
      if ( p = strchr( labirintus[ i ], '*' ) )
      {
        xpoz = p - labirintus[ i ];
        ypoz = i;
        break;
      }
    }

    festes( xpoz, ypoz );

    for ( i = 0; i <= sor; ++i )
    {
      puts( labirintus[ i ] );
    }
  }

  return EXIT_SUCCESS;
}
```

12. fejezet - Formázott kimenet

Tartalom

Háromszöghullám

Ebben a feladatban egy háromszög alakú hullámformát kell előállítani egy megadott amplitúdó/frekvencia párnak megfelelően.

Input és output

A bemenet egy pozitív egész számmal kezdődik, amely a tesztesetek számát jelenti. Ezt követi egy üres sor, majd a tesztesetek, szintén egy-egy üres sorral elválasztva.

Minden teszteset két egész számból áll, amelyek külön sorban vannak megadva. Az első az amplitúdó, a második a frekvencia.

A programodnak hullámformákat kell kiírnia. Minden hullámforma után egy üres sor álljon (az utolsó után is). A hullámformák darabszáma egyenlő a frekvenciával, míg minden hullám vízszintes „magassága" egyenlő az amplitúdóval. Az amplitúdó sohasem lesz nagyobb kilencnél.

Magának a hullámformának minden egyes sorát azzal az egész számmal kell feltöltened, amely az adott sornak a „magasságát" jelzi.

Példa input

```
1

3
2
```

Példa output

```
1
22
333
22
1

1
22
333
22
1
```

Megoldás

Elsőként a triviálisnak tűnő, ciklusokat használó megoldást mutatjuk be:

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

int main()
{
  int n, i, j, amp, freq;

  scanf( "%d", &n );
  while ( n-- )
  {
    scanf( "%d %d", &amp, &freq );
    while ( freq-- )
    {
      for ( i = 1; i <= amp; ++i )
      {
        for ( j = 0; j < i; ++j )
          putchar( i + '0' );
        putchar( '\n' );
```

```
      }
      for ( i -= 2; i > 0; --i )
      {
        for ( j = 0; j < i; ++j )
          putchar( i + '0' );
        putchar( '\n' );
      }
      putchar( '\n' );
    }
  }

  return EXIT_SUCCESS;
}
```

Ez a megoldás a sok egymásba ágyazott ciklus miatt lassúnak tekinthető. Amennyiben a kiírandó hullámformákat sztring literálként összegyűjtjük egy tömbben, és mindig csak a megfelelő hullámformát íratjuk ki, megtakaríthatjuk a ciklusba ágyazott `putchar()` függvényhívásokat. Érdemes megfigyelni, hogy a hosszú sztring literálokat egymás után írt rövidebb sztring literálokkal adjuk meg.

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

char *szamok[ 10 ] = {
  "",
  "1\n",
  "1\n22\n1\n",
  "1\n22\n333\n22\n1\n",
  "1\n22\n333\n4444\n333\n22\n1\n",
  "1\n22\n333\n4444\n55555\n4444\n333\n22\n1\n",
  "1\n22\n333\n4444\n55555\n666666\n55555\n4444\n333\n22\n1\n",
  "1\n22\n333\n4444\n55555\n666666\n7777777\n"
  "666666\n55555\n4444\n333\n22\n1\n",
  "1\n22\n333\n4444\n55555\n666666\n7777777\n88888888\n"
  "7777777\n666666\n55555\n4444\n333\n22\n1\n",
  "1\n22\n333\n4444\n55555\n666666\n7777777\n88888888\n999999999\n"
  "88888888\n7777777\n666666\n55555\n4444\n333\n22\n1\n" };

int main()
{
  int n, amp, freq;

  scanf( "%d", &n );

  while ( n-- )
  {
    scanf( "%d %d", &amp, &freq );

    while ( freq-- )
      puts( szamok[ amp ] );
  }

  return EXIT_SUCCESS;
}
```

LCD-kijelző

Egy barátod épp most vásárol egy új számítógépet. Mostanáig a legnagyobb teljesítményű számítógép, amit használt, egy zsebszámológép volt. Látván az új számítógépét, egy kicsit

csalódott, mert annyira megszerette a számológépe LCD-kijelzőjét. Ezért elhatároztad, hogy írsz egy programot, ami a számítógépén LCD-szerűen jeleníti meg a számokat.

Input

A bemenet több sort tartalmaz, minden megjelenítendő számhoz egyet. Minden sor két egész számot tartalmaz, s-t és n-t ($1 \le s \le 10$, $0 \le n \le 99999999$), ahol n a megjelenítendő szám, és s az a méret, amelyben meg kell jeleníteni. A bemenet egy olyan sorral zárul, amely két 0-t tartalmaz. Ezt a sort nem kell feldolgoznod.

Output

Írd a kimenetre a bemeneten megadott számokat LCD-formátumban, s darab - jelet használva a vízszintes, és s darab | jelet a függőleges szakaszokhoz. Minden számjegy pontosan $s+2$ oszlopot és $2s+3$ sort foglal el. (Minden számjegyben a fehér karaktereket töltsd fel szóközökkel, még az utolsóban is!) Két számjegy között pontosan egy, szóközökből álló oszlopnak kell lennie.

Minden szám után írj ki egy üres sort. (Az összes számjegyre találsz példát a példa outputban.)

Példa input

```
2 12345
3 67890
0 0
```

Példa output

```
      --   --         --
   |    |    | |  | |
   |    |    | |  | |
      --   --   --   --
   | |       |    |    |
   | |       |    |    |
      --   --         --

 ---   ---   ---   ---   ---
|         | |   | |   | |   |
|         | |   | |   | |   |
|         | |   | |   | |   |
 ---         ---   ---
|   |     | |   |     | |   |
|   |     | |   |     | |   |
|   |     | |   |     | |   |
 ---         ---   ---   ---
```

Megoldás

A példa outputban meg lehet figyelni, hogy a számok öt fő részre bonthatók. Minden beolvasott szám esetén

• először a számok tetejét kell kiíratni (az 1-esnek és a 4-esnek nincs teteje, az összes többinek van),

• majd a felső részüket (az 5-ösnek és a 6-osnak csak bal oldala van, az 1-esnek, a 2-esnek, a 3-asnak és a 7-esnek csak jobb oldala, a többinek mindkettő),

• aztán a középső részüket (az 1-esnek, a 7-esnek és a 0-nak nincs közepe, az összes többinek van),

• ezt követően az alsó részüket (a 2-esnek csak bal oldala van, a 6-osnak, a 8-asnak és a 0-nak mindkettő, a többinek csak jobb oldala van),

• végül pedig az aljukat (az 1-esnek, a 4-esnek és a 7-esnek nincs alja, az összes többinek van).

Figyelni kell a nem látható, de kiírandó szóköz karakterekre. A könnyebb kezelhetőség érdekében a

kiírandó számokat a beolvasás után sztringgé konvertáljuk.

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

int main()
{
  unsigned int s;
  unsigned long int szam;

  scanf( "%u %lu", &s, &szam );
  while ( s || szam )
  {
    char lcszam[ 11 ];
    int hossz, sor, i;
    sprintf( lcszam, "%lu", szam );
    hossz = strlen( lcszam );
    for ( i = 0; i < hossz; ++i )  /* tető */
    {
      if ( i )
        putchar( ' ' );
      putchar( ' ' );
      switch( lcszam[ i ] )
      {
        int j;
        case '1':
        case '4':
          for ( j = 0; j < s; ++j )
            putchar( ' ' );
          break;
        case '0':
        case '2':
        case '3':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
          for ( j = 0; j < s; ++j )
            putchar( '-' );
          break;
      }
      putchar( ' ' );
    }
    putchar( '\n' );
    for ( sor = 0; sor < s; ++sor )  /* felső számrész */
    {
      for ( i = 0; i < hossz; ++i )
      {
        if ( i )
          putchar( ' ' );
        switch( lcszam[ i ] )
        {
          int j;
          case '1':
          case '2':
          case '3':
          case '7':
            putchar( ' ' );
            for ( j = 0; j < s; ++j )
              putchar( ' ' );
            putchar( '|' );
```

```
        break;
      case '5':
      case '6':
        putchar( '|' );
        for ( j = 0; j < s; ++j )
          putchar( ' ' );
        putchar( ' ' );
        break;
      case '0':
      case '4':
      case '8':
      case '9':
        putchar( '|' );
        for ( j = 0; j < s; ++j )
          putchar( ' ' );
        putchar( '|' );
        break;
    }
  }
  putchar( '\n' );
}
for ( i = 0; i < hossz; ++i )  /* középső sor */
{
  if ( i )
    putchar( ' ' );
  putchar( ' ' );
  switch( lcszam[ i ] )
  {
    int j;
    case '0':
    case '1':
    case '7':
      for ( j = 0; j < s; ++j )
        putchar( ' ' );
      break;
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '8':
    case '9':
      for ( j = 0; j < s; ++j )
        putchar( '-' );
      break;
  }
  putchar( ' ' );
}
putchar( '\n' );
for ( sor = 0; sor < s; ++sor )  /* alsó számrész */
{
  for ( i = 0; i < hossz; ++i )
  {
    if ( i )
      putchar( ' ' );
    switch( lcszam[ i ] )
    {
      int j;
      case '1':
      case '3':
      case '4':
      case '5':
```

```
          case '7':
          case '9':
            putchar( ' ' );
            for ( j = 0; j < s; ++j )
              putchar( ' ' );
            putchar( '|' );
            break;
          case '2':
            putchar( '|' );
            for ( j = 0; j < s; ++j )
              putchar( ' ' );
            putchar( ' ' );
            break;
          case '0':
          case '6':
          case '8':
            putchar( '|' );
            for ( j = 0; j < s; ++j )
              putchar( ' ' );
            putchar( '|' );
            break;
        }
      }
      putchar( '\n' );
    }
    for ( i = 0; i < hossz; ++i )  /* talprész */
    {
      if ( i )
        putchar( ' ' );
      putchar( ' ' );
      switch( lcszam[ i ] )
      {
        int j;
        case '1':
        case '4':
        case '7':
          for ( j = 0; j < s; ++j )
            putchar( ' ' );
          break;
        case '0':
        case '2':
        case '3':
        case '5':
        case '6':
        case '8':
        case '9':
          for ( j = 0; j < s; ++j )
            putchar( '-' );
          break;
      }
      putchar( ' ' );
    }
    putchar( '\n' );

    putchar( '\n' );
    scanf( "%u %lu", &s, &szam );
  }
  return EXIT_SUCCESS;
}
```

13. fejezet - Egyéb feladatok

Tartalom

Szelektív hulladékgyűjtés

Háttér

A hulladékgyűjtés, vagy adott súlyú objektumoknak különböző, adott követelményeknek megfelelő tárolókba történő elhelyezése egy nagy múlttal rendelkező, érdekes probléma. Bizonyos hulladékgyűjtési problémák NP-teljesek, de közelíthetők dinamikus programozási vagy közel optimális heurisztikus megoldásokkal.

A feladatban visszaváltható üvegek szelektív gyűjtésének a problémáját kell megoldanod.

A probléma

A visszaváltható üvegeket a színük szerint kell szétválasztanod a következő három kategóriába: barna üvegek, zöld üvegek és fehér (átlátszó) üvegek. A feladatban három rekesz van, amelyek adott számú barna, zöld és fehér üveget tartalmaznak. Az üvegeket úgy kell átrendezned, hogy minden rekesz csak egyféle színű üveget tartalmazzon.

A feladatod, hogy minimalizáld az egyik rekeszből a másik rekeszbe átpakolt üvegek számát. Feltételezheted, hogy az egyetlen probléma a rekeszek közötti mozgatások számának a minimalizálása.

Ezen probléma céljainak megfelelően minden rekesz végtelen kapacitású, és az egyetlen feladat, hogy úgy kell átpakolni az üvegeket, hogy minden rekesz csak egyféle színű üveget tartalmazzon. Az üvegek száma nem haladja meg a 2^{31}-t.

Input

A bemenet olyan sorokból áll, amelyek mindegyike 9 egész számot tartalmaz. Az első három szám rendre a barna, a zöld és a fehér üvegek számát jelenti az első rekeszben, a második három szám rendre a barna, a zöld és a fehér üvegek számát jelenti a második rekeszben, az utolsó három szám pedig rendre a barna, a zöld és a fehér üvegek számát jelenti a harmadik rekeszben. A

10 15 20 30 12 8 15 8 31

sor szerint például az első rekeszben 20 fehér üveg van, a másodikban 12 zöld, a harmadik rekeszben pedig 15 barna üveg található. Az egy sorban lévő számokat egy vagy több szóköz választja el egymástól. A programodnak a bemenet minden sorát fel kell dolgoznia.

Output

A bemenet minden egyes sorához ki kell írnod egy sort, amelyből kiderül, hogy milyen színű üvegek melyik rekeszbe kerülnek minimális számú pakolással. Ki kell még írnod továbbá az üvegmozgatások minimális számát is.

A kimenet minden sora egy olyan sztringgel kezdődjön, amely a `G`, `B`, `C` nagybetűkből áll (a zöld, a barna és a fehér színek jelölésére). A betűk az egyes rekeszekhez tartozó színeket jelentik.

A sztring első karaktere az első rekeszhez tartozó színt jelenti, a második karakter a második, a harmadik karakter pedig a harmadik rekeszhez tartozó színt jelenti.

Az üvegmozgatások minimális számát a sztring után, attól egy szóközzel elválasztva kell kiírnod. Ha egynél több sorrend létezik a minimális mozgatásszám mellett, akkor az alfabetikusan legkisebb (ábécé sorrend szerint legelőrébb szereplő) betűhármast kell kiírnod.

Példa input

```
1 2 3 4 5 6 7 8 9
5 10 5 20 10 5 10 20 10
```

Példa output

```
BCG 30
CBG 50
```

Megoldás

A feladat a megfogalmazás ellenére egyáltalán nem bonyolult. Hat esetet kell szisztematikusan végigvizsgálnunk a rövidítések angol ábécé szerinti sorrendjében. Mivel a minimális értéket keressük, ezért kezdetben az első esetet (amikor az első rekeszben gyűjtjük a barna üvegeket, a másodikban a fehéreket, a harmadikban pedig a zöldeket -- BCG) tekinthetjük minimumnak. Az összes többi eset pedig az összehasonlítások sorrendjében (BGC, CBG, CGB, GBC, GCB) változtathatja ezt a minimális értéket. Egyenlőség esetén a korábban megjegyzett elrendezést nem változtatjuk meg. Minden összehasonlítás-sorozat végén ki kell írni a minimumhoz tartozó elrendezést és a minimumértéket.

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

enum { BARNA, ZOLD, FEHER };

int main()
{
  int rekesz[ 3 ][ 3 ], minimum, index, ertek;
  char *elrendezes[] = { "BCG", "BGC", "CBG", "CGB", "GBC", "GCB" };

  while ( scanf( "%d %d %d %d %d %d %d %d %d",
    &rekesz[ 0 ][ BARNA ], &rekesz[ 0 ][ ZOLD ], &rekesz[ 0 ][ FEHER ],
    &rekesz[ 1 ][ BARNA ], &rekesz[ 1 ][ ZOLD ], &rekesz[ 1 ][ FEHER ],
    &rekesz[ 2 ][ BARNA ], &rekesz[ 2 ][ ZOLD ], &rekesz[ 2 ][ FEHER ] )
    != EOF )
  {
    minimum = rekesz[ 1 ][ BARNA ] + rekesz[ 2 ][ BARNA ] +
              rekesz[ 0 ][ ZOLD ]  + rekesz[ 1 ][ ZOLD ] +
              rekesz[ 0 ][ FEHER ] + rekesz[ 2 ][ FEHER ];
    index = 0;

    ertek = rekesz[ 1 ][ BARNA ] + rekesz[ 2 ][ BARNA ] +
            rekesz[ 0 ][ ZOLD ]  + rekesz[ 2 ][ ZOLD ] +
            rekesz[ 0 ][ FEHER ] + rekesz[ 1 ][ FEHER ];
    if ( ertek < minimum )
    {
      minimum = ertek;
      index = 1;
    }
    ertek = rekesz[ 0 ][ BARNA ] + rekesz[ 2 ][ BARNA ] +
            rekesz[ 0 ][ ZOLD ] +  rekesz[ 1 ][ ZOLD ] +
            rekesz[ 1 ][ FEHER ] + rekesz[ 2 ][ FEHER ];
    if ( ertek < minimum )
    {
      minimum = ertek;
      index = 2;
```

```
    }
    ertek = rekesz[ 0 ][ BARNA ] + rekesz[ 1 ][ BARNA ] +
            rekesz[ 0 ][ ZOLD ] +  rekesz[ 2 ][ ZOLD ] +
            rekesz[ 1 ][ FEHER ] + rekesz[ 2 ][ FEHER ];
    if ( ertek < minimum )
    {
      minimum = ertek;
      index = 3;
    }
    ertek = rekesz[ 0 ][ BARNA ] + rekesz[ 2 ][ BARNA ] +
            rekesz[ 1 ][ ZOLD ] +  rekesz[ 2 ][ ZOLD ] +
            rekesz[ 0 ][ FEHER ] + rekesz[ 1 ][ FEHER ];
    if ( ertek < minimum )
    {
      minimum = ertek;
      index = 4;
    }
    ertek = rekesz[ 0 ][ BARNA ] + rekesz[ 1 ][ BARNA ] +
            rekesz[ 1 ][ ZOLD ] +  rekesz[ 2 ][ ZOLD ] +
            rekesz[ 0 ][ FEHER ] + rekesz[ 2 ][ FEHER ];
    if ( ertek < minimum )
    {
      minimum = ertek;
      index = 5;
    }
    printf( "%s %d\n", elrendezes[ index ], minimum );
  }
  return EXIT_SUCCESS;
}
```

Szerelvényrendezés

Régi vasútállomásokon néha még mindig találkozhatunk „vagoncserélővel". A vagoncserélő az a vasúti alkalmazott, akinek az az egyetlen feladata, hogy rendezze a vasúti kocsikat. Ha már jól vannak elrendezve a vagonok, a masinisztának már csak le kell kapcsoltatnia a kocsikat egyesével azokon az állomásokon, ahová a rakományukat szánták.

A „vagoncserélő" elnevezés az első ilyen munkát végző emberről maradt fenn, aki egy vasúti híd közelében található állomáson dolgozott. Ez a híd nem felnyílt, hanem elfordult egy, a folyó közepén álló oszlopon. Ha a hidat 90 fokkal elfordították, át tudtak mellette haladni a hajók mindkét oldalon.

A vagoncserélő felfedezte, hogy a híd két vasúti kocsival a tetején is forgatható. Ha 180 fokkal forgatta el a hidat, a kocsik helyet cseréltek (mellékhatásként természetesen meg is fordultak, de mivel a vagonok mindkét irányba egyformán húzhatók, ez most nem fontos).

Most, hogy lassan minden vagoncserélő eltűnik az állomásokról, a vasúttársaságok szeretnék automatizálni ezt a tevékenységet. A feladatod, hogy írj egy olyan programot, amely feldolgozza az alább megadott formában érkező bemeneti adatokat, és eldönti, hogy mennyi az a legkevesebb vagonpárcsere, amivel egy adott vonat rendezhető.

Input

A bemenet első sora tartalmazza a tesztesetek számát (N). Minden teszteset két sorból áll. A teszteset első sora, egy L egész szám, a szerelvény hosszát határozza meg ($0 \leq L \leq 50$). A teszteset második sora az 1-től L-ig terjedő számok egy permutációja, amely a vagonok aktuális sorrendjét írja le. A vagonokat olyan sorrendbe kell rendezned, amelyben az 1-es vagon áll elöl, őt követi a 2-

es, stb., végül az L-es vagon zárja a sort.

Output

Minden egyes tesztszerelvényre a következő mondatot kell kiírnod:

`A vagoncserék optimális száma` S`.`

S helyén egy egész számnak, az adott tesztesethez tartozó vagoncserék optimális számának kell szerepelnie.

Példa input

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

Példa output

```
A vagoncserék optimális száma 1.
A vagoncserék optimális száma 6.
A vagoncserék optimális száma 1.
```

Megoldás

A feladat megoldásához a buborékrendezés algoritmusát lehet felhasználni.

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

#define MERET 50

int tomb[ MERET ], l;

void cserel( int i, int j )
{
  int seged = tomb[ i ];
  tomb[ i ] = tomb[ j ];
  tomb[ j ] = seged;
}

int buborek()
{
  int csere = 0;
  int korlat = l - 1, t;
  do
  {
    int j;
    t = -1;
    for ( j = 0; j < korlat; ++j )
      if ( tomb[ j ] > tomb[ j + 1 ] )
      {
        cserel( j, j + 1 );
        ++csere;
        t = j;
      }
    korlat = t;
  } while ( t != -1 );
  return csere;
}
```

```
int main()
{
  int n;
  scanf( "%d", &n );
  while ( n-- )
  {
    int i;
    scanf( "%d", &l );
    for ( i = 0; i < l; ++i )
      scanf( "%d", tomb + i );
    printf( "A vagoncserék optimális száma %d.\n", buborek( tomb, l ) );
  }

  return EXIT_SUCCESS;
}
```

Óramutatók

Egy hagyományos analóg óra perc- és óramutatója által bezárt szöget kell meghatároznod. Feltételezheted, hogy ha lenne másodpercmutató, az mindig a 12-esre mutatna. Minden szöget a legkisebb pozitív szögként kell megadnod. Például 9:00 esetén a szög 90 fok, és nem -90 vagy 270 fok.

Input

A bemenet időpontok listáját tartalmazza *H*`:`*M* formában, mindegyik külön sorban, ahol $1 \leq H \leq 12$ és $00 \leq M \leq 59$. A bemenet a `0:00` időponttal ér véget. Vigyázz, hogy H állhat egy és két számjegyből is (attól függően, hogy 1 és 9, vagy 10 és 12 közé esik-e), míg M mindig két számjegyű. (A bemeneti időpontok tehát egy tipikus digitális órán megjelenő időpontok.)

Output

A kimenetre a két mutató által bezárt legkisebb pozitív szöget kell fokokban mérve kiírnod minden bemeneti adat esetén. A szögek mindegyikének 0 és 180 fok közé kell esnie. Minden választ külön sorba írj, ugyanabban a sorrendben, ahogy a bemeneten szerepeltek. A kimenetet a legközelebbi $1/1000$-re kell kerekíteni, azaz három tizedesjegyet kell írnod a tizedespont után.

Példa input

```
12:00
9:00
8:10
0:00
```

Példa output

```
0.000
90.000
175.000
```

Megoldás

A feladat megoldásához a következő megfigyelésekre van szükség: az óra nagymutatója egy perc alatt $360^\circ / 60 = 6^\circ$-ot mozdul. A kismutató ugyanennyi idő alatt csak $360^\circ / 12 / 60 = 0,5^\circ$-ot halad. A kismutató helyzete persze függ az eltelt órák számától is: egy óra alatt 30°-ot fordul. Amennyiben a két mutató elfordulása közötti különbség nagyobb, mint 180°, akkor a különbséget ki kell vonni

360°-ból.

A számítások alapján látható, hogy a három tizedesjegyből valójában csak az első értékes, amely ráadásul csak 0 vagy 5 lehet.

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

int main()
{
  int ora, perc;

  scanf( "%d:%d", &ora, &perc );
  while ( ora ) {
    double orafok, percfok, kulonbseg;

    percfok = 6.0 * perc;
    orafok = ( ora % 12 ) * 30.0 + perc * 0.5;
    kulonbseg = fabs( orafok - percfok );
    if ( kulonbseg > 180.0 )
      kulonbseg = 360.0 - kulonbseg;
    printf( "%0.3lf\n", kulonbseg );
    scanf( "%d:%d", &ora, &perc );
  }

  return EXIT_SUCCESS;
}
```

Milyen nap van?

A ma használatban lévő naptár a rómaiaktól származik. Julius Caesar iktatta törvénybe azt a naptárrendszert, amelyet azóta Julián-naptárnak nevezünk. Ebben a rendszerben minden hónap 31 napos, kivéve áprilist, júniust, szeptembert és novembert, amelyek 30 naposak, valamint februárt, amely 28 napos nem szökőévekben és 29 napos szökőévekben. A szökőévek ebben a rendszerben négyévenként követték egymást. Ez azért van, mert az ősi Róma csillagászai az évet 365,25 nap hosszúnak számították, így minden negyedik évben egy extra napot kellett beszúrni, hogy a naptár együtt haladjon az évszakokkal. Ehhez minden néggyel osztható évben beiktattak egy extra napot (február 29-ét).

• **Julián-szabály:** Minden 4-gyel osztható év szökőév, azaz eggyel több napja van (február 29.).

1582-ben Gergely pápa csillagászai rájöttek, hogy az év nem 365,25 nap hosszú, hanem közelebb áll a 365,2425 naphoz. A szökőévszabályt ezért a következők szerint módosították:

• **Gergely-szabály:** Minden 4-gyel osztható év szökőév, kivéve ha osztható 100-zal, de nem osztható 400-zal.

Hogy kiegyenlítsék az évszakok addig felgyülemlett eltolódását a naptárhoz képest, a naptárt 10 nappal eltolták: az 1582. október 4-ét követő napot október 15-ének nyilvánították.

Anglia és birodalma (beleértve az Egyesült Államokat) nem váltottak a Gergely-féle naptárrendszerre 1752-ig, amikor a szeptember 2-át követő napot szeptember 14-ének nyilvánították. (A késést a VIII. Henrik és a pápa közötti rossz viszony okozta.)

Írj programot, amely egyesült államokbeli dátumokhoz tartozó napneveket határoz meg a megfelelő naptárt használva.

Input

A bemenet pozitív egész számok sorozatából áll, soronként háromból, ahol minden sor egy dátumot ír le. Az érvényes dátumok formátuma „*hónap nap év*", ahol a *hónap* egy 1 (január) és 12 (december) közé eső szám, a *nap* egy 1 és 31 közötti érték (a hónapnak megfelelően), az *év* pedig egy pozitív szám. A bemenet három darab 0-val zárul, amit már nem kell feldolgoznod.

Output

A kimenet a bemeneti dátumból és annak a napnak a nevéből álljon, amelyikre az adott dátum esik, a példában látható formátumban. Az érvénytelen vagy az adott időben az Egyesült Államokban használt naptárban nem létező dátumok esetén hibaüzenetet kell kiírnod.

Példa input

```
11 15 1997
1 1 2000
7 4 1998
2 11 1732
9 2 1752
9 14 1752
4 33 1997
0 0 0
```

Példa output

```
1997. november 15. szombat
2000. január 1. szombat
1998. július 4. szombat
1732. február 11. péntek
1752. szeptember 2. szerda
1752. szeptember 14. csütörtök
4/33/1997 érvénytelen dátum.
```

Megoldás

A számításainkhoz nyilvántartjuk azt, hogy milyen napra esik január 1-je az egyes években. Megfigyelhetjük, hogy 1752 előtt 28 éves, utána pedig 400 éves periódusokban ismétlődnek ezek a napok. Ezeket a periódusokat a szökőévek és a napok kombinációjából kaphatjuk meg. Ha tároljuk ezeket a napokat (például két tömbben), akkor könnyen meghatározhatjuk, hogy egy adott dátum milyen napra esik. Egyszerűsíthetjük a nap meghatározását, ha azt is tároljuk, hogy hány nap telt el az évből az egyes hónapok első napjáig.

Az 1752. szeptember 14. és december 31. közötti napok esetén 11 nappal korábbi értéket veszünk figyelembe a kieső napok miatt.

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

int szokoev( int ev )
{
  return ev <= 1752 && ev % 4 == 0 ||
         ev % 4 == 0 && ev % 100 != 0 || ev % 400 == 0;
}

const int elott[] =
  { 3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3, 4, 6,
    0, 1, 2, 4, 5, 6, 0, 2, 3, 4, 5, 0, 1, 2 };
const int utan[] =
  { 5, 0, 1, 2, 3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3, 4, 6, 0, 1,
    2, 4, 5, 6, 0, 2, 3, 4, 5, 0, 1, 2, 3, 5, 6, 0, 1, 3, 4, 5,
    6, 1, 2, 3, 4, 6, 0, 1, 2, 4, 5, 6, 0, 2, 3, 4, 5, 0, 1, 2,
    3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3, 4, 6, 0, 1, 2, 4, 5, 6,
    0, 2, 3, 4, 5, 0, 1, 2, 3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3,
```

```
    4, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3, 4, 6, 0, 1, 2, 4, 5, 6,
    0, 2, 3, 4, 5, 0, 1, 2, 3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3,
    4, 6, 0, 1, 2, 4, 5, 6, 0, 2, 3, 4, 5, 0, 1, 2, 3, 5, 6, 0,
    1, 3, 4, 5, 6, 1, 2, 3, 4, 6, 0, 1, 2, 4, 5, 6, 0, 2, 3, 4,
    5, 0, 1, 2, 3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3, 4, 6, 0, 1,
    2, 3, 4, 5, 6, 1, 2, 3, 4, 6, 0, 1, 2, 4, 5, 6, 0, 2, 3, 4,
    5, 0, 1, 2, 3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3, 4, 6, 0, 1,
    2, 4, 5, 6, 0, 2, 3, 4, 5, 0, 1, 2, 3, 5, 6, 0, 1, 3, 4, 5,
    6, 1, 2, 3, 4, 6, 0, 1, 2, 4, 5, 6, 0, 2, 3, 4, 5, 0, 1, 2,
    3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3, 4, 6, 0, 1, 2, 4, 5, 6,
    0, 1, 2, 3, 4, 6, 0, 1, 2, 4, 5, 6, 0, 2, 3, 4, 5, 0, 1, 2,
    3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3, 4, 6, 0, 1, 2, 4, 5, 6,
    0, 2, 3, 4, 5, 0, 1, 2, 3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2, 3,
    4, 6, 0, 1, 2, 4, 5, 6, 0, 2, 3, 4, 5, 0, 1, 2, 3, 5, 6, 0,
    1, 3, 4, 5, 6, 1, 2, 3, 4, 6, 0, 1, 2, 4, 5, 6, 0, 2, 3, 4 };
const char *honapnev[] =
  { "", "január",     "február", "március",  "április",
        "május",      "június",  "július",   "augusztus",
        "szeptember", "október", "november", "december" };
const char *napnev[] =
  { "hétfő", "kedd", "szerda", "csütörtök",
    "péntek", "szombat", "vasárnap" };
const int nem_szoko_ho[] =
  { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
const int szoko_ho[] =
  { 0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
const int ho_elott[] =
  { 0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334 };
const int szoko_ho_elott[] =
  { 0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335 };

int main()
{
  int honap, nap, ev, szam, szoko;
  scanf( "%d %d %d", &honap, &nap, &ev );
  while ( ev )
  {
    szoko = szokoev( ev );
    if ( honap > 12 ||
         !szoko && nap > nem_szoko_ho[ honap ] ||
         szoko && nap > szoko_ho[ honap ] ||
         ev == 1752 && honap == 9 && 2 < nap && nap < 14 )
      printf( "%d/%d/%d érvénytelen dátum.\n", honap, nap, ev );
    else
    {
      szam = ( szoko ? szoko_ho_elott[ honap ] : ho_elott[ honap ] ) +
             nap - 1;
      if ( ev == 1752 && ( honap > 9 || honap == 9 && nap > 2 ) )
        szam -= 11;
      szam += ev <= 1752 ? elott[ ev % 28 ] : utan[ ev % 400 ];
      printf( "%d. %s %d. %s\n",
              ev, honapnev[ honap ], nap, napnev[ szam % 7 ] );
    }
    scanf( "%d %d %d", &honap, &nap, &ev );
  }

  return EXIT_SUCCESS;
}
```

DNS-rendezés

Egy sorozat „rendezetlenségének" egy lehetséges mértéke azon elempárok száma, amelyek egymáshoz képest rossz sorrendben vannak. Például a `DAABEC` betűsorozatban ez a mérték 5, mivel a `D` a jobb oldalán álló betűk közül négynél is nagyobb, az `E` pedig nagyobb, mint a tőle jobbra lévő betű. Ezt a mértéket a sorozatbeli inverziók számának nevezzük. Az `AACEDGG` sorozatban csak 1 inverzió van (`E` és `D`), azaz majdnem rendezett, míg a `ZWQM` sorozatban 6 inverzió van (annyira rendezetlen, amennyire csak lehetséges -- pontosan fordított sorrendű).

Feladatod, hogy katalogizáld DNS-láncok (olyan sztringek, amelyek csak az `A`, `C`, `G` és `T` betűket tartalmazzák) egy sorozatát. Azonban nem ábécé szerint, hanem „rendezettségi" sorrend szerint kell őket sorba rakni, a „leginkább rendezettől" a „legkevésbé rendezettig" haladva. A rendezendő láncok azonos hosszúságúak.

Input

A bemenet első sora két egész számot tartalmaz: egy n pozitív egészet, amely a DNS-láncok hosszát jelenti, és egy m pozitív egészet, amely a rendezendő láncok számát adja meg ($0 < n \leq 50$, $0 < m \leq 100$). Ezeket m sor követi, amelyek mindegyike egy n hosszúságú sztringet tartalmaz.

Output

A kimenetre a beolvasott DNS-láncok listáját rendezve kell kiírnod, a „leginkább rendezettől" a „legkevésbé rendezettig" haladva. Ha két vagy több lánc egyformán rendezett, akkor ugyanabban a sorrendben írasd ki őket, mint ahogyan a bemeneten szerepeltek.

Példa input

```
10 6
AACATGAAGG
TTTTGGCCAA
TTTGGCCAAA
GATCAGATTT
CCCGGGGGGA
ATCGATGCAT
```

Példa output

```
CCCGGGGGGA
AACATGAAGG
GATCAGATTT
ATCGATGCAT
TTTTGGCCAA
TTTGGCCAAA
```

Megoldás

Első megoldásunkban egy 100 elemű, elemenként (a záró '\0' karakterrel együtt) 51 karaktert tartalmazó tömböt, és egy, az elemek sorszámát tartalmazó indexvektort használunk. Rendezéskor nem a sztringeket, hanem csak az indexeiket cserélgetjük, így megtakaríthatjuk a sztringek másolásával járó időveszteségeket. Rendező algoritmusunk a buborékrendezés lesz; ez egyike azon rendezéseknek, amelyek meg tudják tartani a rendezés szempontjából azonos tulajdonságú elemek eredeti (feldolgozás előtti) sorrendjét.

Megjegyzendő, hogy egyes C-implementációk az `stdlib.h` header állományban deklarált `qsort()` függvényt úgy implementálják, hogy az megtartja a rendezés szempontjából azonos tulajdonságúnak számító elemek eredeti sorrendjét (lásd a 5.7. feladatot).

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

```
#define MAXHOSSZ  50
#define MAXDARAB 100

char dnslancok[ MAXDARAB ][ MAXHOSSZ + 1 ];
int sorszam[ MAXDARAB ], n, m, i;

int inverzio( char *s )
{
  int i, j, szamlalo = 0;
  for ( i = 0; i + 1 < strlen( s ); ++i )
    for ( j = i + 1; j < strlen( s ); ++j )
      if ( s[ i ] > s[ j ] )
        ++szamlalo;
  return szamlalo;
}

void csere( int i, int j )
{
  int seged = sorszam[ i ];
  sorszam[ i ] = sorszam[ j ];
  sorszam[ j ] = seged;
}

void buborek( int meret )
{
  int i, j, utolsocsere;
  for ( i = meret - 1; i > 0; i = utolsocsere )
  {
    utolsocsere = 0;
    for ( j = 0; j < i; ++j )
      if ( inverzio( dnslancok[ sorszam[ j ] ] ) >
           inverzio( dnslancok[ sorszam[ j + 1 ] ] ) )
      {
        csere( j, j + 1 );
        utolsocsere = j;
      }
  }
}

int main()
{
  scanf( "%d %d\n", &n, &m );
  for ( i = 0; i < m; ++i )
    gets( dnslancok[ sorszam[ i ] = i ] );
  buborek( m );
  for ( i = 0; i < m; ++i )
    puts( dnslancok[ sorszam[ i ] ] );
  return EXIT_SUCCESS;
}
```

Az így megírt program jól működik ugyan, de lassú. Hogyan lehetne gyorsítani rajta? A következő lehetőségeink vannak:

• Gyorsabb algoritmust találunk ki a DNS-láncban lévő inverziók meghatározására.

• A DNS-láncok inverzióit minden beolvasott DNS-lánc esetén csak egyszer számítjuk ki, rögtön a beolvasáskor, és ezeket az értékeket tároljuk valahol (például egy 100 elemű, egészeket tartalmazó tömbben).

• Más, gyorsabb rendező algoritmust használunk a DNS-láncok rendezésére. Például az egyes DNS-láncokat beszúró rendezéssel tesszük a helyükre, rögtön a beolvasásuk után.

A következőkben egy viszonylag nagy tárigényű, de a korábbinál gyorsabb megoldást ismertetünk, amely kihasználja azt, hogy a maximum 50 karakter hosszúságú DNS-láncok nem tartalmazhatnak egy bizonyos mennyiségű inverziónál többet. Hogyan tudnánk megbecsülni ezt a mennyiséget? Nos, tegyük fel, hogy az 50 karakter hosszúságú DNS-lánc minden karaktere különböző. (Ez persze nem így van, ezért a becslésünk nagyon durva felső becslés lesz.) Ebben a DNS-láncban akkor lesz a legtöbb inverzió, ha az első karaktere az őt követő 49 karakter mindegyikével inverzióban áll (ez 49 darab inverzió), a második karaktere az őt követő 48 karakter mindegyikével inverzióban áll (ez újabb 48 darab inverzió), és így tovább. Ez összesen maximum

$$49+48+\ldots+2+1+0=50\cdot\frac{49+0}{2}=1225$$

darab inverziót jelent. Mivel maximum 100 DNS-láncot kell feldolgozni, ezért az 1225 inverziós érték mindegyikéhez hozzárendelünk egy-egy 100 elemű, mutatókat tartalmazó tömböt, amelynek elemei a beolvasott DNS-láncok tömbjének megfelelő elemeire hivatkozhatnak (ez lesz az inverziós táblázat), valamint egy indexvektort, amely azt mondja meg, hogy hány hivatkozás szerepel már az adott inverziójú bejegyzésnél. Ez utóbbi érték -- mivel 0 kezdőértékről indul -- a következő szabad bejegyzés indexét is jelzi egyúttal. A DNS-láncok beolvasásakor minden DNS-láncnál csak egyszer kell az inverziók számát meghatározni, ezt követően ugyanis a DNS-lánc sztringjének kezdőcíme bekerül az inverziós táblázatba, miután pedig az összes DNS-láncot beolvastuk, csak végig kell menni az inverziós táblázaton, és ki kell íratni a benne szereplő hivatkozásokat a bejárás sorrendjében.

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

#define MAXHOSSZ    50
#define MAXDARAB   100

#define INVERZIOK 1225

int i, j, k, n, m;
char dnslancok[ MAXDARAB ][ MAXHOSSZ + 1 ];
int inverziok[ MAXDARAB ];
char *t[ INVERZIOK + 1 ][ MAXDARAB ], inv[ INVERZIOK ];

int main()
{
  scanf( "%d %d\n", &n, &m );
  for ( i = 0; i < m; ++i )
  {
    gets( dnslancok[ i ] );
    for ( j = 0; j + 1 < n; ++j )
      for ( k = j + 1; k < n; ++k )
        if ( dnslancok[ i ][ j ] > dnslancok[ i ][ k ] )
          ++inverziok[ i ];
    t[ inverziok[i] ][ inv[ inverziok[i] ]++ ] = &dnslancok[ i ][ 0 ];
  }
  for ( i = 0; i < INVERZIOK; ++i )
    for ( j = 0; j < inv[ i ]; ++j )
      puts( t[ i ][ j ] );
  return EXIT_SUCCESS;
}
```

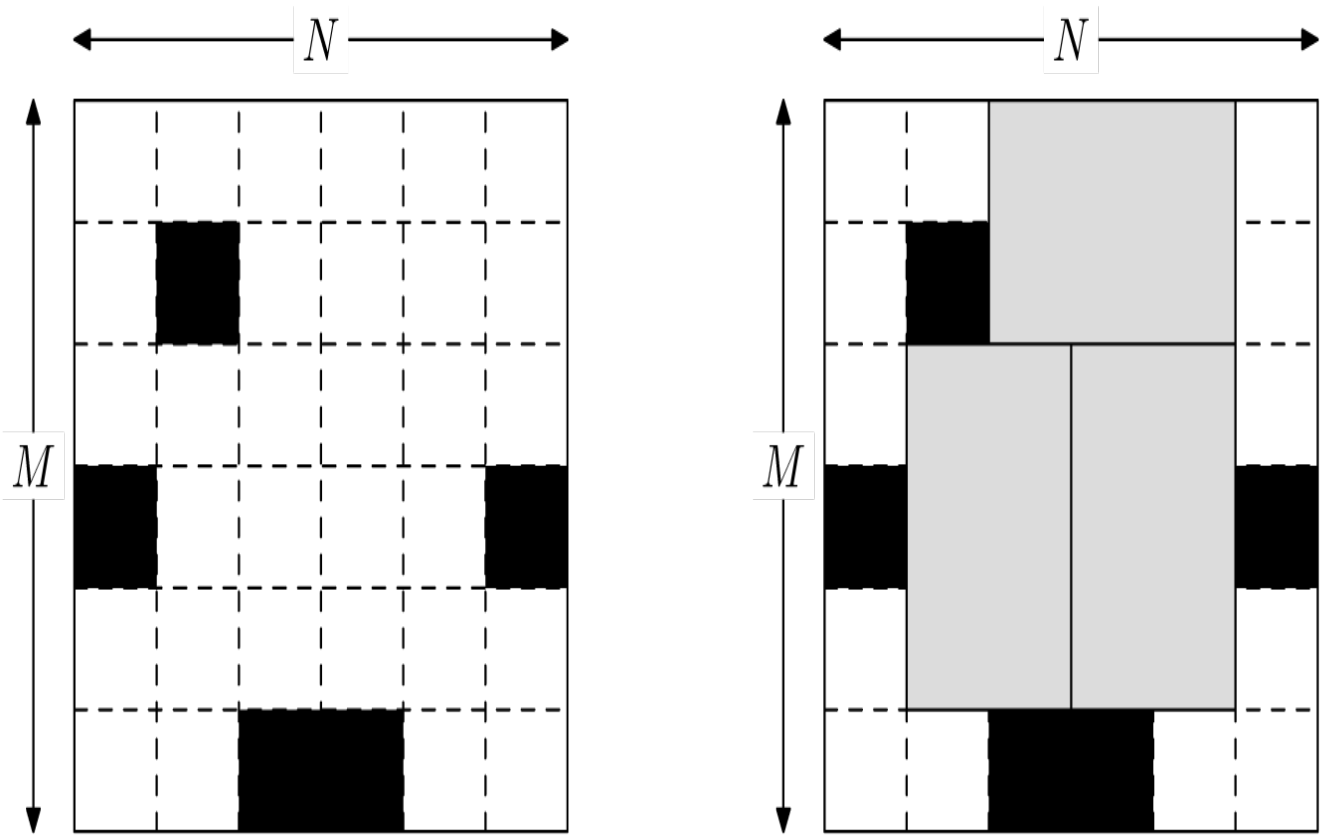
14. fejezet - Közép-európai Informatikai Diákolimpia, 2002, Kassa, Szlovákia

Tartalom

Bugs Integrated, Inc.

A Bugs Integrated, Inc. a fejlett memóriachipek egyik fő gyártója. Most kezdik meg egy új, 6 terabájtos Q-RAM chip gyártását. Minden chip 6 egységnégyzetből áll, 2×3-as téglalap formában. A Q-RAM chipeket a következőképpen készítik el: vesznek egy téglalap alakú szilikonlapkát, amely $N \times M$ egységnégyzetre van felosztva. Ezután minden négyzetet gondosan tesztelnek, és a rosszakat megjelölik fekete színnel (lásd a 14.1. ábrát).

14.1. ábra -



Végezetül a szilikonlapkát szétvágják memóriachipekre. Minden chip 2×3 (vagy 3×2) egységnégyzetből áll. Természetesen egyik chip sem tartalmazhat rossz (megjelölt) négyzeteket. Elképzelhető, hogy nem lehetséges úgy szétvágni a lapkát, hogy minden jó egységnégyzet valamely memóriachip része legyen. A vállalat a lehető legkevesebb jó négyzetet szeretné elpocsékolni, ezért tudni szeretnék, hogyan vágják szét a lapkát, hogy maximális számú chipet készíthessenek.

Feladat

Adva vannak szilikonlapkák dimenziói, és minden egyes lapka esetén a rossz egységnégyzetek listája. A feladatod egy olyan program írása, amely kiszámítja minden egyes lapka esetén a lapkából kivágható chipek maximális számát.

Input

A `bugs.in` bemeneti állomány első sora egyetlen D egész számból áll ($1 \leq D \leq 5$), amely a szilikonlapkák száma. Ezután D darab blokk következik, melyek mindegyike egy szilikonlapkát ír le. Minden blokk első sora három egész számot tartalmaz, N-et ($1 \leq N \leq 150$), M-et ($1 \leq M \leq 10$)

és K-t ($0 \leq K \leq MN$), egy-egy szóközzel elválasztva. N a lapka hossza, M a magassága, K pedig a lapkán található rossz négyzetek száma. A következő K sor a rossz négyzetek listáját tartalmazza. Minden sor két egész számból áll, x-ből és y-ból ($1 \leq x \leq N$, $1 \leq y \leq M$), amelyek egy-egy rossz négyzet koordinátái. (A bal felső négyzet koordinátája $[1,1]$, a jobb alsóé $[N,M]$.)

Output

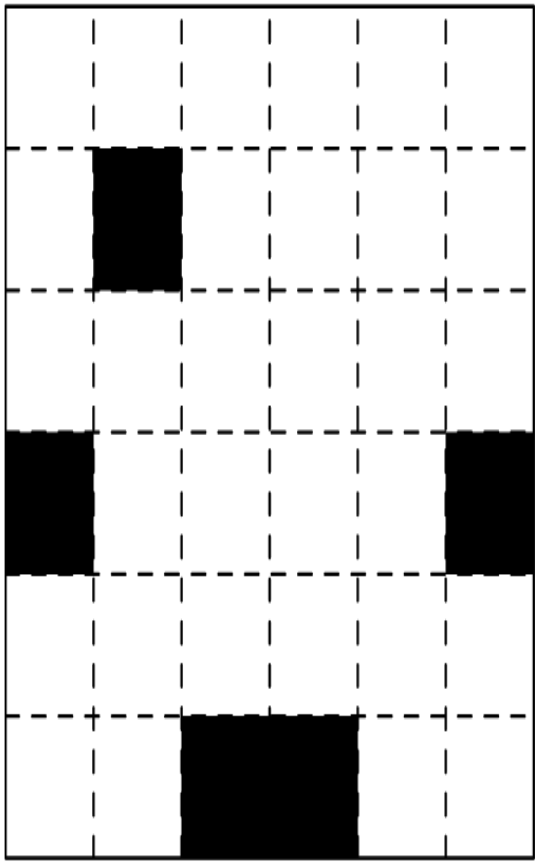
A bemeneti állomány minden egyes lapkájára írj ki egy sort a `bugs.out` állományba, amely az adott lapkából kivágható chipek maximális számát tartalmazza.

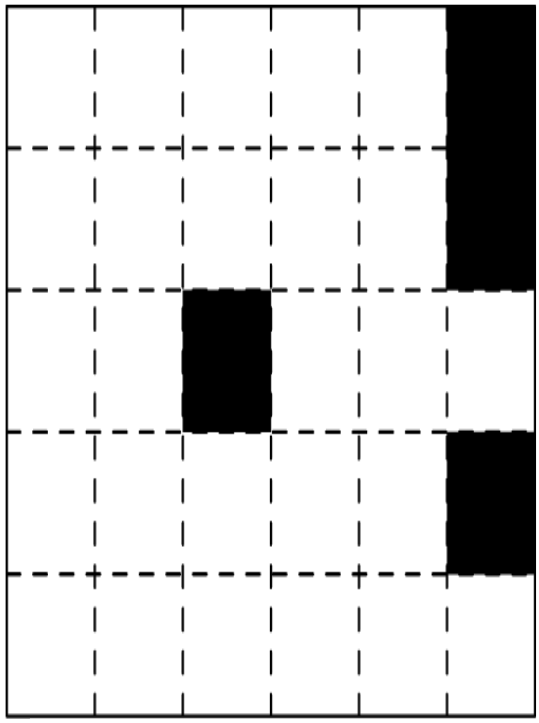
Példa

Input	Output
2	3
6 6 5	4
1 4	
4 6	
2 2	
3 6	
6 4	
6 5 4	
3 3	
6 1	
6 2	
6 4	

A 14.2. ábrán láthatók a példához tartozó szilikonlapkák.

14.2. ábra -





Idő- és memóriakorlát

Egy 600 MHz-es (vagy jobb) Celeron processzorral és 64 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 30 másodperc, a memóriakorlát 8 MB.

A Hódító zászlóalja

Az emberiség történelmében számos furcsa csatát találhatunk, mint például az alábbi, amely Franciaországban, 1747-ben zajlott.

Volt egy erőd a Dordogne folyó bal partján fekvő kis falu, Bassignac-le-Haut mellett, nem messze a Chastang gáttól. A gáttól felfelé az erődig egy széles vörösmárvány lépcső vezetett. Egyik reggel az őrség egy nagy zászlóaljat látott közeledni az erőd felé, egy rettegett vezetővel -- a Hódítóval.

Amikor a Hódító elérte az erődöt, az erőd parancsnoka már várta. Mivel a parancsnoknak kevés katona állt rendelkezésére, a következőt javasolta a Hódítónak: „*Látom, hogy mögötted a lépcsőkön sok katonád áll. Játszhatnánk egy kis <<játékot>>: Minden körben tetszőlegesen szétosztod a katonáidat két csoportba. Ezután eldöntöm, hogy melyik csoport marad, és melyik megy haza. Az ittmaradt katonák ezután egyet lépnek felfelé a lépcsőn. Ha legalább egy katonád eléri a legfelső lépcsőfokot, te leszel a győztes, különben te vesztesz. És ez esetben visszavonulsz a gát mögé*" -- tette hozzá a parancsnok, kezével a Chastang gát felé mutatva.

A Hódítónak azonnal megtetszett ez a játék, így beleegyezett, és elkezdett „hódítani".

Feladat

Te most a Hódító szerepét játszod. N lépcsőfok vezet az erődig ($2 \leq N \leq 2000$), és legfeljebb 1000000000 katonád van. Minden lépcsőfoknál adott a rajta álló katonák száma, 1 jelöli a legfelső lépcsőfokot, N a legalsót. Az 1-es lépcsőfokon kezdetben egyetlenegy katona sem áll.

A programodnak megadott minden olyan kezdőpozíció esetén nyernie kell, amelyik nyerő pozíció (azaz létezik olyan stratégia, amely lehetővé teszi számodra, hogy nyerj, függetlenül az ellenfeled lépéseitől), egyébként csak helyesen le kell játszani a játékot (és veszíteni).

Ez egy interaktív probléma; egy alább specifikált könyvtár ellen fogsz játszani. A programod minden körben megadja a könyvtárunknak a katonák egy csoportját. A könyvtár 1-et vagy 2-t ad vissza, amely megadja, hogy melyik csoport maradjon (1 jelenti az általad leírt csoportot, 2 a többi katonát). Ha a játék véget ér (vagy azért, mert te nyertél, vagy mert nem maradt több katona játékban), a könyvtár megszakítja a programodat. A programod másképpen nem állhat meg.

A könyvtár interfésze

A `libconq` könyvtár két alprogramot biztosít:

• `start` -- visszaadja az N számot, és feltölt egy `stairs` nevű tömböt, az egyes lépcsőfokokon álló katonák számaival (azaz az i-edik lépcsőn `stairs[i]` katona áll),

• `step` -- megkap egy `subset` nevű (legalább N elemű[2]) tömböt, amely az általad választott katonák csoportját írja le; a katonák csoportját az egyes lépcsőfokokon álló katonák számával kell megadni, ahogy a `start` függvényben.

Ha hibásan adsz meg egy csoportot, a játék véget ér, és a programod 0 pontot fog érni az adott tesztesetre. Vedd figyelembe, hogy C/C++-ban is 1-gyel kezdődik a lépcsőfokok számozása.

Az alábbiakban ezeknek az alprogramoknak a deklarációi láthatók FreePascal és C/C++ környezetben:

```
procedure start(var N: longint; var stairs: array of longint);
function step(subset: array of longint): longint;

void start(int *N, int *stairs);
int step(int *subset);
```

Lejjebb példákat találsz a könyvtár használatára FreePascal és C/C++ környezetben egyaránt; mindkét kódrészlet ugyanazt csinálja -- elindítják a játékot, majd lejátszanak egy kört úgy, hogy a kiválasztott csoport a véletlenszerűen választott lépcsőfokokon álló katonák összessége lesz. Az igazi programod valószínűleg végtelen ciklusban fogja lejátszani a köröket.

Erősen ajánlott, hogy a `stairs` és a `subset` tömböket ugyanúgy definiáld, ahogyan a példában láthatók. (Vedd figyelembe, hogy a FreePascal könyvtár a válaszát a tömb első N elemében adja vissza, függetlenül attól, hogyan definiáltad, a C/C++ könyvtár pedig az 1-től N-ig indexelt elemekben adja vissza a válaszát.)

A könyvtárat hozzá kell csatolnod a programodhoz -- FreePascalban a uses libconq; utasítással, C/C++-ban pedig az #include "libconq.h" direktívával, ahol úgy kell lefordítanod a programodat, hogy a libconq.c-t is hozzáadod a fordító argumentumaihoz.

FreePascal példa	C/C++ példa
uses libconq;	#include "libconq.h"

var stairs: array[1..2000] of longint;	int stairs[2001];
subset: array[1..2000] of longint;	int subset[2001];
i,N,result: longint;	int i,N,result;
...	...
start(N,stairs);	start(&N, stairs);
...	...
for i:=1 to N do	for (i=1;i<=N;i++)
if random(2)=0	if (rand()%2==0)
then subset[i]:=0	subset[i]=0;
else subset[i]:=stairs[i];	else subset[i]=stairs[i];
result:=step(subset);	result=step(subset);
...	...

Egy példajáték

Te	A könyvtár
start(N,stairs)	$N = 8$, $stairs = (0,1,1,0,3,3,4,0)$
step((0,1,0,0,1,0,1,0))	2-t ad vissza
step((0,1,0,0,0,1,0,0))	2-t ad vissza
step((0,0,0,3,2,0,0,0))	1-et ad vissza
step((0,0,2,0,0,0,0,0))	2-t ad vissza
step((0,1,0,0,0,0,0,0))	2-t ad vissza
step((0,1,0,0,0,0,0,0))	nem tér vissza, te nyertél

Erőforrások

A weboldalon példakönyvtárakat találhatsz C/C++-hoz és FreePascalhoz is. Ezek a könyvtárak különböznek azoktól, amelyeket a tesztelés során fogunk használni. Arra használhatod őket, hogy meggyőződj a könyvtári hívásaid helyességéről. A példakönyvtár a bemenetet a `libconq.dat` állományból olvassa, amely két sort tartalmaz. Az első sorban a lépcsőfokok N darabszáma, a második sorban pedig N darab egész szám van, az 1-től N -ig indexelt lépcsőfokokon álló katonák számai.

A fenti példához tartozó `libconq.dat` állomány a következőképpen néz ki:

```
8
0 1 1 0 3 3 4 0
```

Idő- és memóriakorlát

Egy 600 MHz-es (vagy jobb) Celeron processzorral és 64 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 1 másodperc, a memóriakorlát 16 MB.

A díszes kerítés

Richard épp befejezte új házának építését. Egyetlen dolog hiányzik még, egy szép kis fakerítés. Mivel Richardnak fogalma sincs, hogyan kell fakerítést készíteni, úgy döntött, hogy megrendeli azt. Valahogy a kezébe került a 2002-es ACME[3] Kerítés Katalógus, a szép kis fakerítések alapvető forrása. Miután elolvasta az előszót, már tudta, mitől lesz szép egy kis fakerítés.

Egy fakerítés N deszkából áll, amelyeket függőleges állásban egymás mellé helyezünk. Egy kerítés akkor és csak akkor szép, ha teljesülnek az alábbi feltételek:

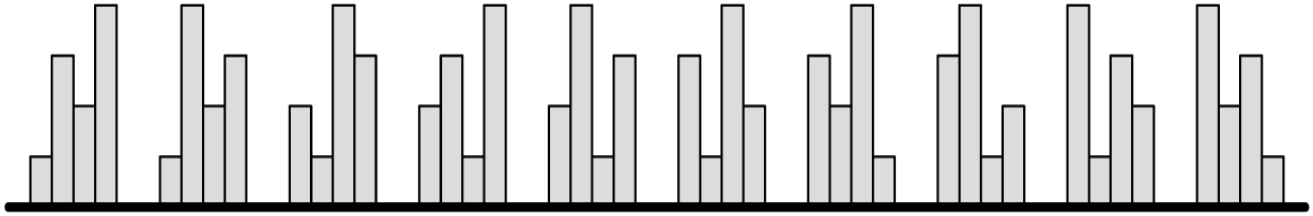
• A deszkák különböző hosszúságúak, mégpedig $1, 2, \ldots, N$ egységnyiek.

• Minden olyan deszka, amelynek két szomszédja van, vagy nagyobb mindkét szomszédjánál, vagy kisebb mindkettőnél. (Ez azt jelenti, hogy a kerítés teteje felváltva emelkedik, illetve süllyed.)

Ebből az következik, hogy minden N deszkából álló szép kerítést egyedileg megadhatunk az $1, \ldots, N$ számok egy $a_1, \ldots, a_N$ permutációjaként, amelyre $\forall i : 1 < i < N$ esetén $(a_i - a_{i-1}) \cdot (a_i - a_{i+1}) > 0$, és fordítva: minden ilyen permutáció egy szép kerítést ír le.

Nyilvánvaló, hogy sok különböző, N deszkából álló szép fakerítés létezik. Hogy sorrendet vigyenek a katalógusukba, az ACME kereskedelmi vezetője úgy döntött, hogy a következőképpen rendezi a kerítéseket: Az $a_1, \ldots, a_N$ permutáció által leírt A kerítés akkor és csak akkor lesz a katalógusban a $b_1, \ldots, b_N$ permutáció által leírt B kerítés előtt, ha létezik olyan i, amelyre $\forall j < i$ esetén $a_j = b_j$ és $a_i < b_i$. (Ahhoz tehát, hogy eldöntsük, a két kerítés közül melyik van előrébb a katalógusban, vegyük a megfelelő permutációkat, keressük meg az első eltérés helyét, és hasonlítsuk össze az ott szereplő két értéket.) Az N deszkából álló kerítéseket a katalógusban való előfordulásuk sorrendjében sorszámozzuk (1-től indulva). Ez a szám a kerítések *katalógusszáma*. A 14.3. ábrán látható az összes $N = 4$ deszkából álló szép kerítés, katalógusszámaik alapján rendezve.

14.3. ábra -



Miután gondosan átvizsgálta az összes szép kis fakerítést, Richard elhatározta, hogy néhányat megrendel közülük. Mindegyik esetén feljegyezte a deszkáik számát, valamint a katalógusszámaikat. Később, amikor a barátaival találkozott, meg akarta mutatni nekik a megrendelt kerítéseket, de elvesztette a katalógust. Csak a feljegyzései voltak nála. Segíts neki kitalálni, hogy fognak kinézni a kerítései.

Input

A `fence.in` bemeneti állomány első sora a bemeneti adathalmazok K számát tartalmazza ($1 \leq K \leq 100$). Ezután K sor következik, amelyek mindegyike egy bemeneti adathalmazt ír le.

Ezen sorok mindegyike az N és a C egész számokból áll ($1 \leq N \leq 20$), egy szóközzel elválasztva. N a kerítést alkotó deszkák száma, C pedig a kerítés katalógusszáma.

Felteheted, hogy a 20 deszkából álló szép kis fakerítések száma belefér egy 64 bites előjeles egész változóba (`long long` C/C++-ban, `int64` FreePascalban). Azt is felteheted, hogy a bemenet helyes, azaz C legalább 1, és nem haladja meg az N deszkából álló szép kerítések számát.

Output

Minden egyes bemeneti adathalmazra írj ki egy sort a `fence.out` állományba, amely leírja a katalógusban C-edikként szereplő, N deszkából álló kerítést. Pontosabban: ha a kerítés az $a_1,\ldots,a_N$ permutációval írható le, akkor a kimeneti állomány megfelelő sorának az a_i számokat kell tartalmaznia (megfelelő sorrendben), egy-egy szóközzel elválasztva.

Példa

Input	Output
2	1 2
2 1	2 3 1
3 3	

Idő- és memóriakorlát

Egy 600 MHz-es (vagy jobb) Celeron processzorral és 64 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 1 másodperc, a memóriakorlát 1 MB.

Az országút és a hét törpe

Egyszer volt, hol nem volt, volt egyszer egy ország, ahol törpecsaládok éltek. Ezt az országot Törpeföldnek hívták. Minden család egy házban élt. A törpék gyakran meglátogatták más családokban élő barátaikat. Mivel Törpeföldön nem volt ellenségeskedés, úgy esett, hogy minden törpe meglátogatta egymást egy rövid időszak alatt.

Egyszer a Törpeföld körül élő emberek elhatározták, hogy építenek egy csomó egyenes országutat. Mivel az emberek nem tudtak a törpékről, a tervezett országutak közül néhány keresztülhaladt Törpeföldön. A törpék ezt észrevették, és nagyon elszomorodtak. Mivel ők nagyon kicsik és nagyon lassúak is, ezért nem tudnak biztonságosan átmenni az országúton.

A törpéknek valahogy sikerült megszerezniük az országutak terveit, és most a te segítségedre van szükségük. Továbbra is szeretnék meglátogatni egymást, ezért nem örülnek azoknak az országutaknak, amelyek a házaikat két nem üres csoportra osztják. Miután megállapították, melyek ezek az országutak, egy varázslat segítségével megakadályozzák az embereket, hogy megépítsék őket.

A törpék nagyon kicsik, és nem érik fel a billentyűzetet, így tehát a te segítségedet kérik.

Feladat

Adott a síkban N számú pont (ezek a házak), valamint adottak egyenesek (országutak). Minden egyenes esetén meg kell határoznod, hogy az N pont mindegyike az egyenes ugyanazon oldalára esik-e, vagy sem. A programodnak az éppen feldolgozott egyeneshez tartozó választ **azelőtt** kell a kimenetre írnia, mielőtt beolvasná a következő leírását. Felteheted, hogy egyik országút sem megy keresztül egyik házon sem.

Input és output

A programodnak a bemenetet a standard inputról kell olvasnia (C/C++-ban *stdin*, FreePascalban *input*), a kimenetet pedig a standard outputra kell írnia (C/C++-ban *stdout*, FreePascalban *output*). A bemenet első sora az N egész számot tartalmazza ($0 \leq N \leq 100000$). Ezt N sor követi, amelyekből az i-edik az x_i, y_i valós számokból áll ($-10^9 \leq x_i, y_i \leq 10^9$), egy szóközzel elválasztva; ezek az i-edik ház koordinátái.

A következő sorok mindegyike négy valós számot tartalmaz: X_1-et, Y_1-et, X_2-t és Y_2-t ($-10^9 \leq X_1, Y_1, X_2, Y_2 \leq 10^9$), egy-egy szóközzel elválasztva. Ezek a számok az országút két különböző pontjának ($[X_1, Y_1]$ és $[X_2, Y_2]$) koordinátái. A programodnak a bemenet minden egyes sorára egy sort kell kiírnia, amely a „JÓ" sztringet tartalmazza, ha az adott pontok mindegyike az adott egyenes ugyanazon oldalára esik, vagy a „ROSSZ" sztringet, ha az adott egyenes kettéosztja a pontokat. Miután a kimenet egy sorát kiírtad, ki kell ürítened a kimeneti puffert. A következőkben találsz egy példát arra, hogyan kell ezt csinálni.

A programodat megállítjuk, miután megadta az utolsó országúthoz tartozó választ. **A programodnak nem szabad magától megállnia!** Felteheted, hogy nem lesz 100 000-nél több országút.

Input/output rutinok C/C++-ban

Egy sor beolvasása (figyeld meg, hogy az utolsó `%lf` után nincs szóköz):

```
double X_1, Y_1, X_2, Y_2;
scanf(" %lf %lf %lf %lf",&X_1,&Y_1,&X_2,&Y_2);
```

Egy egyeneshez tartozó kimenet kiírása:

```
printf("JÓ\n"); fflush(stdout);
```

Input/output rutinok FreePascalban

Egy sor beolvasása:

```
var X_1, Y_1, X_2, Y_2 : double;
read(X_1,Y_1,X_2,Y_2);
```

Egy egyeneshez tartozó kimenet kiírása:

```
writeln('JÓ'); flush(output);
```

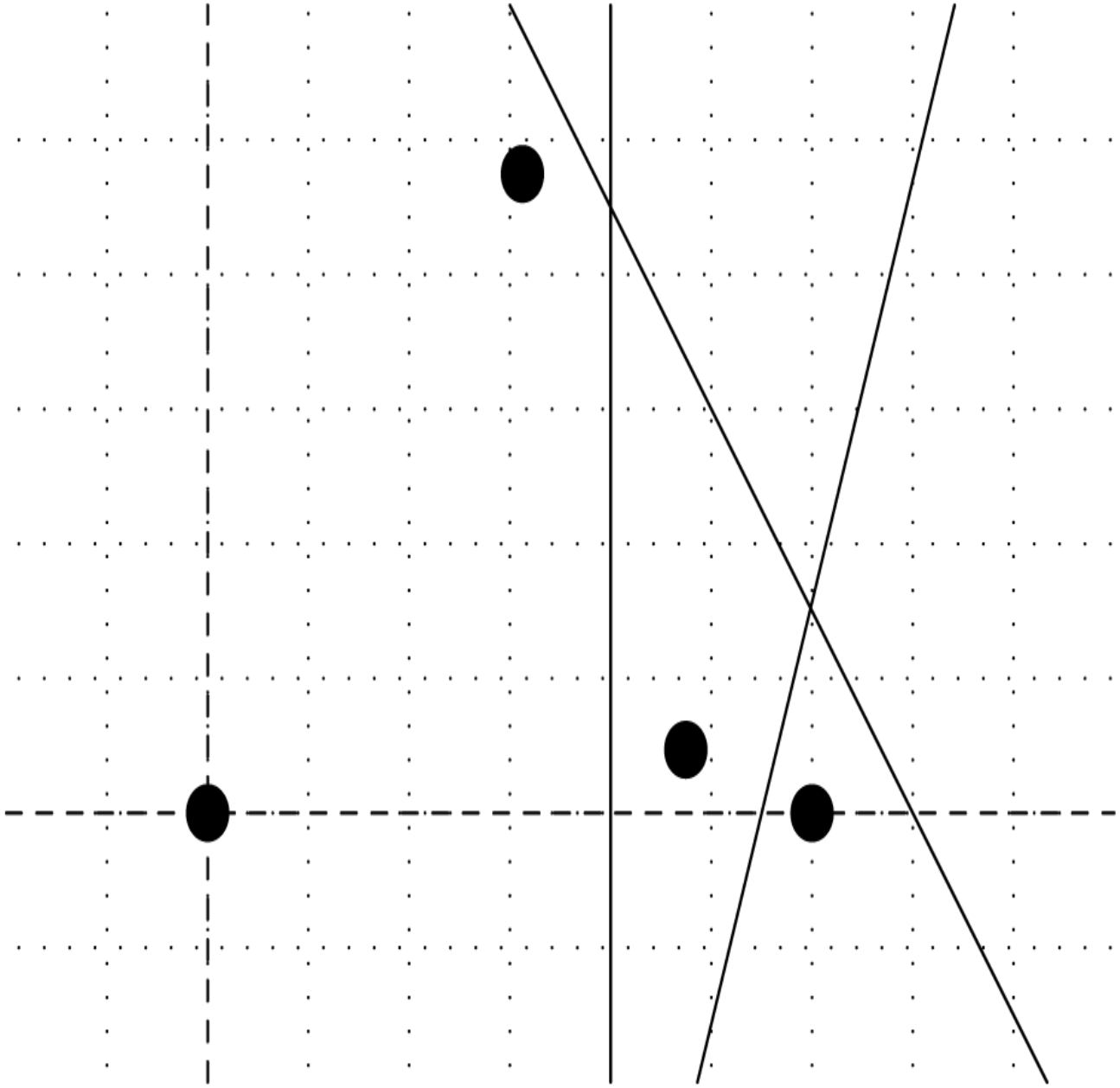
Figyelmeztetés

Tanácsos a `double` adattípust használni (C/C++-ban és FreePascalban egyaránt) a valós számok tárolására. Vedd figyelembe, hogy valós aritmetika használata esetén kerekítési hibák fordulhatnak elő. Ha el akarod dönteni, hogy két valós szám (x és y) egyenlő-e, ne az $x=y$ kifejezést használd, hanem az $|x-y|<\varepsilon$ vizsgálatot (ahol ε egy kicsi konstans, 10^{-4} megteszi).

Példa

Input	Output
4	JÓ
0.0 0	ROSSZ
6.00 -0.001	ROSSZ
3.125 4.747	
4.747 0.47	
5 3 7 0	
4 -4.7 7 4.7	
4 47 4 94	

14.4. ábra -



Idő- és memóriakorlát

Egy 600 MHz-es (vagy jobb) Celeron processzorral és 64 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 5 másodperc, a memóriakorlát 16 MB.

A király őrei

Egyszer volt, hol nem volt, volt egyszer egy királyság. Mindennel rendelkezett, amire egy királyságnak szüksége van, nevezetesen egy királlyal és a kastélyával. A kastély alaprajza egy

téglalap, amely $M \times N$ egységnégyzetre van felosztva. A négyzetek egy része fal, a többi szabad. A szabad négyzeteket *szobáknak* fogjuk hívni. A királyságunk királya rendkívül paranoiás volt, így egy nap úgy döntött, hogy néhány szobában rejtett csapdát készít (a fenekükön aligátorokkal).

Ez azonban nem volt elég. Egy héttel később úgy határozott, hogy annyi őrt helyez el a kastélyában, amennyit csak lehetséges. Ez viszont nem is olyan egyszerű. Az őröket úgy képezték ki, hogy amint meglátnak valakit, lelövik az illetőt. A királynak ezért elővigyázatosan kell elhelyeznie az őröket, mert ha két őr meglátja egymást, lelövik egymást! Az is nyilvánvaló, hogy a király nem helyezhet el őrt egy csapdával ellátott szobában.

Az egy szobában lévő őrök látják egymást, tehát minden szobában legfeljebb egy őr állhat. Két különböző szobában lévő őr akkor és csak akkor látja egymást, ha a szobáiknak megfelelő négyzetek a kastély alaprajzán ugyanazon sorban, vagy ugyanazon oszlopban vannak, **és** nincs köztük fal. (Az őrök csak négy irányban látnak, hasonlóan a bástyához a sakkban.)

Feladat

A feladatod annak meghatározása, hogy hány őrt tud a király elhelyezni a kastélyában (a fenti szabályoknak megfelelően), és ennyi őrnek egy lehetséges elrendezést találni a szobákban.

Input

A `guards.in` bemeneti állomány első sora az M és az N számokat tartalmazza ($1 \leq M, N \leq 200$), amelyek a kastély alaprajzának dimenziói. A következő M sor közül az i-edik N darab számból áll ($a_{i,1}, \ldots, a_{i,N}$), egy-egy szóközzel elválasztva, ahol

- $a_{i,j} = 0$ azt jelzi, hogy az $[i,j]$ négyzet szabad (csapda nélküli szoba);
- $a_{i,j} = 1$ azt jelzi, hogy az $[i,j]$ négyzet csapdát tartalmaz;
- $a_{i,j} = 2$ azt jelzi, hogy az $[i,j]$ négyzet fal.

Egy négyzet első koordinátája a sort, a második az oszlopot jelöli.

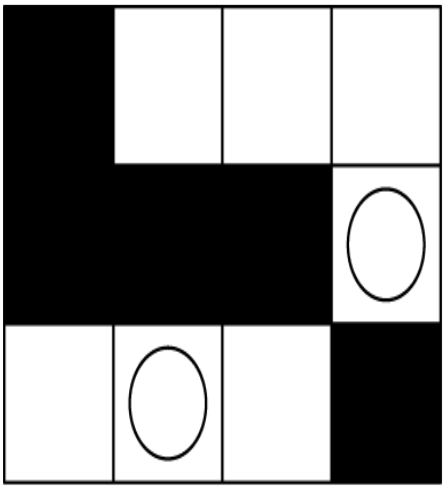
Output

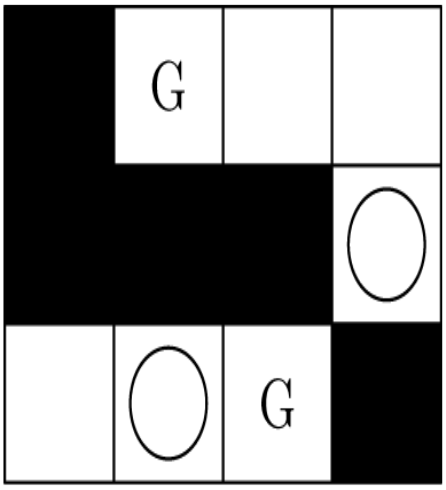
A `guards.out` kimeneti állomány első sorának a király által a kastélyába elhelyezhető őrök maximális számát (K) kell tartalmaznia. A következő K sorba K darab őrnek egy lehetséges elrendezését kell kiírni, ahogyan a kastély szabad szobáiban elhelyezkedhetnek úgy, hogy egyik őr sem látja a másikat.

Pontosabban: ezen sorok közül az i-ediknek az r_i és a c_i egész számokat kell tartalmaznia, egy szóközzel elválasztva. Az r_i és a c_i annak a szobának a koordinátái, ahol az i-edik őrt elhelyezzük (r_i a sor, c_i az oszlop).

A 14.5. ábrán a példa bemenetnek megfelelő kastély és egy lehetséges helyes kimenet látható.

14.5. ábra -





Példa

Input	Output
3 4	2
2 0 0 0	1 2
2 2 2 1	3 3
0 1 0 2	

Idő- és memóriakorlát

Egy 600 MHz-es (vagy jobb) Celeron processzorral és 64 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 15 másodperc, a memóriakorlát 8 MB.

Születésnapi parti

Lassan közeledik John születésnapja, és -- mint minden évben -- egy nagyszabású kerti partit rendez. Szeretné, ha minden barátja eljönne, de tudja, hogy ez (sajnos) szinte lehetetlen. A múlt héten például Susie szakított Steve-vel, így majdnem lehetetlen, hogy mindketten eljöjjenek. John az elmúlt hét nagy részét azzal töltötte, hogy meglátogatta és meghívta a barátait. Kapott néhány ígéretet, de még több kérést. („Ha engem meghívsz, akkor meg kell hívnod a barátomat is" -- mondta Veronica. „Ha meghívod a Burdiliak ikreket, akkor Joseph-re és rám ne számíts!" -- jelentette ki Peter.) John hirtelen rájött, hogy már az is elég nehéz lesz, hogy az összes kérésnek megfeleljen.

Feladat

Adva van azoknak a kéréseknek a leírása, amelyeket John a barátaitól kapott. A feladatod, hogy megtaláld az embereknek egy olyan csoportját, amelyből John ha minden embert meghívna (és

senki mást nem) a partijára, akkor az összes kérést teljesíteni tudná. A kérések a következőképpen vannak leírva:

• `név` egy kérés. Ez a kérés akkor és csak akkor teljesül, ha John meghívja a `név` nevű embert.

• `-név` egy kérés. Ez a kérés akkor és csak akkor teljesül, ha John nem hívja meg a `név` nevű embert.

A `név` mindkét esetben egy legfeljebb 20 kisbetűből álló sztring, amely nem tartalmaz szóközöket.

• Ha `R1`,,`Rk` kérések, akkor (R1 & ... & Rk) is egy kérés. Ez a kérés akkor és csak akkor teljesül, ha az `R1`,,`Rk` kérések teljesülnek.

• Ha `R1`,,`Rk` kérések, akkor (R1 | ... | Rk) is egy kérés. Ez a kérés akkor és csak akkor teljesül, ha az `R1`,,`Rk` kérések közül legalább egy teljesül.

• Ha `R1` és `R2` kérések, akkor (R1 => R2) is egy kérés. Ez a kérés akkor és csak akkor **nem** teljesül, ha `R1` teljesül, és `R2` nem teljesül.

Input

Tíz bemeneti állományt találsz a weblapon, melyek neve rendre `party1.in`, , `party10.in`. Minden bemeneti állomány 10 pontot ér.

A bemeneti állomány első sorában John barátainak F száma szerepel, a következő F sorban állnak a nevek, soronként egy. A következő sor a kérések N számát tartalmazza. A következő N sor mindegyikében egy kérés áll.

Output

Minden egyes `partyX.in` állomány esetén elő kell állítanod a megfelelő `partyX.out` kimeneti állományt, amely egy helyes megoldást tartalmaz. A kimeneti állomány első sora azoknak az embereknek a K száma legyen, akiket Johnnak meg kell hívnia. A következő K sorban álljanak a meghívottak nevei, soronként egy. Felteheted, hogy a bemeneti állományok mindegyikének van (nem szükségszerűen csak egy) megoldása. Ha több lehetséges megoldás is van, közülük bármelyiket kiírathatod.

A `party*.out` állományokat a webes felületet használva ugyanúgy kell beküldened, mint a többi feladathoz tartozó programokat.

Példa

Input	Output
3	2
veronica	steve
steve	dick
dick	
3	
(veronica => dick)	
(steve => -veronica)	

(steve & dick)

Idő- és memóriakorlát

Nincs.

---

[2] C/C++-ban $N+1$ elemű.

[3] A Company Making Everything -- A Mindent Gyártó Vállalat.

15. fejezet - Közép-európai Informatikai Diákolimpia, 2003, Münster, Németország

Tartalom

Hanoi tornyai

Már bizonyára találkoztál a *Hanoi tornyai* problémával: három rúdon különböző méretű fakorongok vannak egymáson, és kezdetben minden korong ugyanazon a rúdon van, méret szerint rendezve úgy, hogy a legnagyobb van legalul. A cél, hogy a teljes tornyot átrakjuk valamelyik másik rúdra úgy, hogy egyszerre csak egy korongot mozgatunk, és sosem rakunk egy nagyobb korongot egy kisebbre.

Egy régi mítosz szerint egy ősi tibeti kolostorban a szerzetesek 1000 évig próbálták megoldani ennek a problémának azt a különösen bonyolult változatát, amelyben 47 korong volt a rudakra pakolva. Mivel ez legalább $2^{47}-1$ mozgatást igényel, és a szerzetesek, követve ugyan a szabályokat, de mindenféle stratégia nélkül kezdtek hozzá a munkához, jól összekutyulták az egészet. Szeretnék viszont, hogy a korongok egy tetszőleges rúdra legyenek felpakolva minimális számú mozgatással. Tettek azonban egy olyan fogadalmat, amely megtiltja számukra, hogy a szabályokkal ellentétesen mozgassák a korongokat. Tudni szeretnék, hogy leginkább melyik rúdra kellene átrakniuk a korongokat, és hogy mennyi a mozgatások minimális száma.

Írj programot, amely megoldja a szerzeteseknek ezt a problémát. A programodnak tetszőleges számú (N) korongot kell tudnia kezelni ($0 < N \leq 100000$). A számítások során előálló számok igencsak nagyok lehetnek. Emiatt a szerzeteseket csak a mozgatások számának 1000000-val képzett maradéka érdekli.

Input

A `hanoi.in` bemeneti állomány első sora a korongok N számát tartalmazza. A második sor három egészből áll: s_1, s_2, s_3, ahol $0 \leq s_1, s_2, s_3 \leq N$, továbbá $s_1+s_2+s_3=N$. Ezek a rudakon lévő korongok darabszámai. A harmadiktól az ötödik sorig terjedő sorok az egyes rudakon lévő

korongok méreteit tartalmazzák. Pontosabban: a bemeneti állomány $(i+2)$-edik sora az $m_{i,1},\ldots,m_{i,s_i}$ egész számokból áll, ahol $1 \le m_{i,j} \le N$. Ezek az i-edik rúdon lévő korongok méretei. A korongok lentről felfelé vannak megadva, ezért $m_{i,1} > m_{i,2} > \ldots > m_{i,s_i}$. Az üres rúd üres sorral van megadva. Az N darab korong mindegyike különböző méretű. A számokat egy-egy szóköz választja el egymástól.

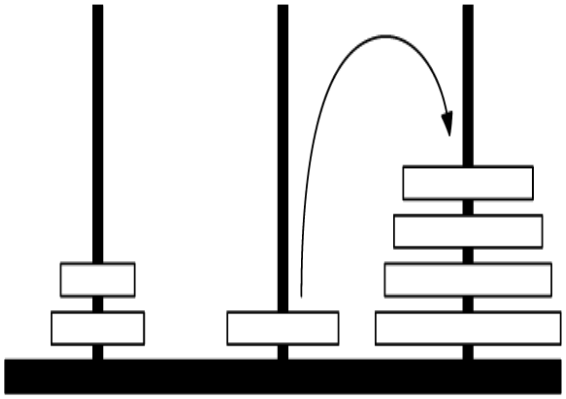
Output

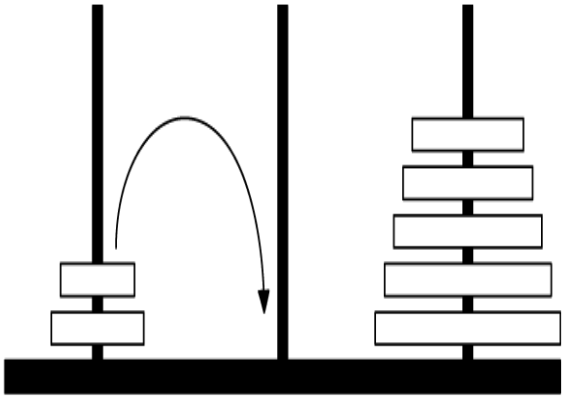
A `hanoi.out` kimeneti állomány első sora tartalmazza a $d \in \{1,2,3\}$ számot, annak a rúdnak a sorszámát, amelyre a korongokat minimális számú mozgatással lehet átrakni. A második sorban az az M szám szerepeljen, amely a szükséges mozgatások számának 1000000-val képzett maradéka.

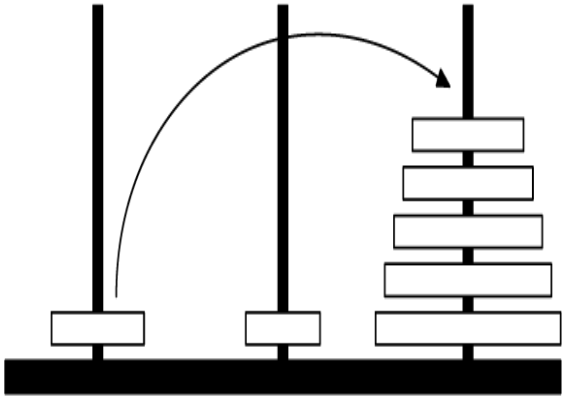
Példa

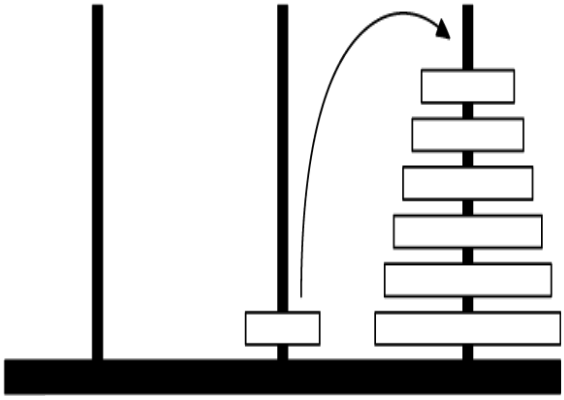
hanoi.in	hanoi.out
7	3
2 1 4	4
2 1	
3	
7 6 5 4	

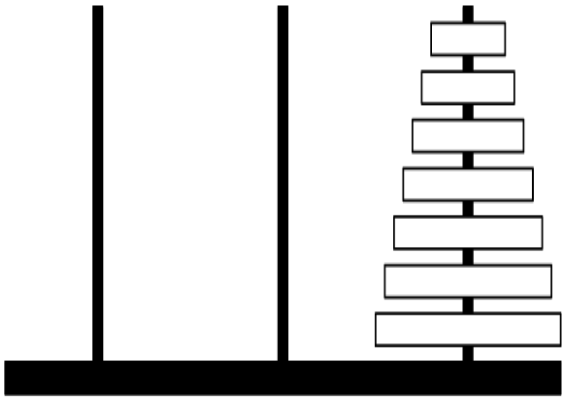
15.1. ábra -











Tesztadatok

Az értékelés során a programodat 20 különböző inputra teszteljük. A következő táblázat a bemeneti állományok első sorait tartalmazza, azaz a korongok N darabszámát az egyes bemeneti adathalmazokban.

Teszteset	1	2	3	4	5	6	7	8	9
N	5	10	15	20	50	100	150	200	1000

Teszteset	10	11	12	13	14	15
N	2000	3000	4000	30000	40000	50000

Teszteset	16	17	18	19	20
N	60000	70000	80000	90000	100000

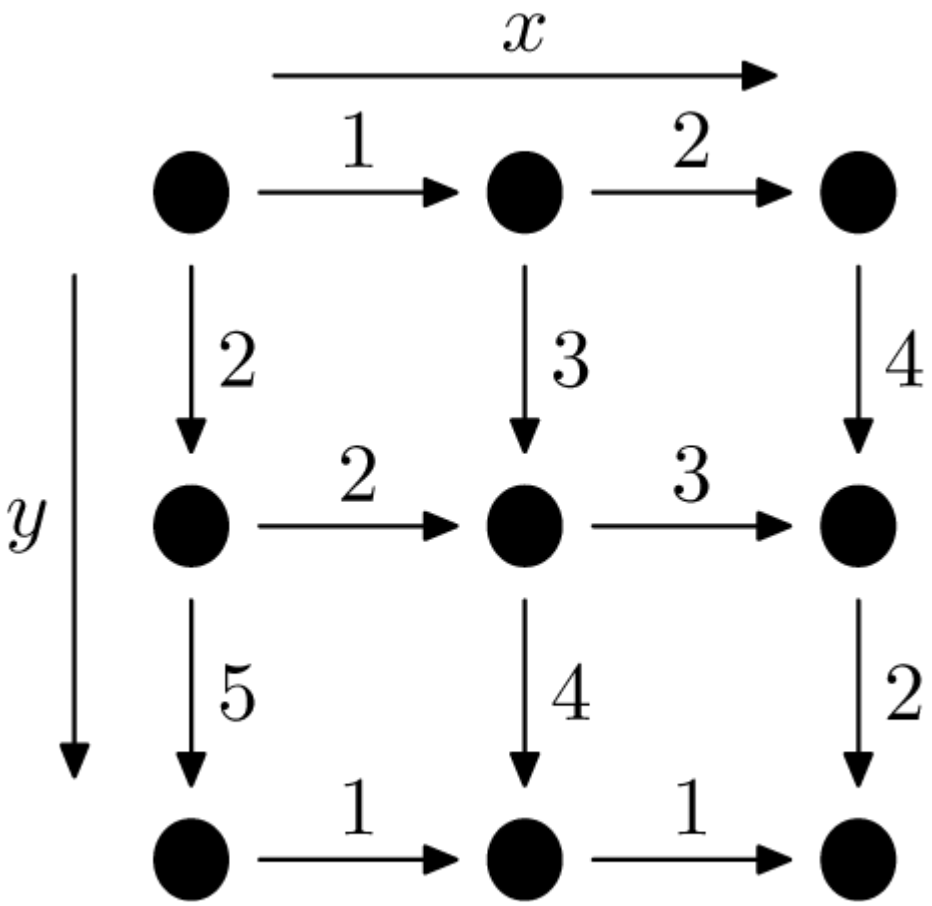
Idő- és memóriakorlát

Egy 650 MHz-es (vagy jobb) Pentium III processzorral és 128 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 3 másodperc, a memóriakorlát 16 MB.

Négyzet

Adott egy gráf, amelynek minden csúcsa egy $N \times N$ pontot ($1 \leq N \leq 2003$) tartalmazó rácson helyezkedik el. Minden csúcs pontosan egy rácsponttal esik egybe. Minden csúcsból irányított élek haladnak a jobb oldali és az alsó szomszéd csúcsba, ha azok léteznek. Ezek az élek súlyozva vannak egy w egész számmal ($1 \leq w \leq 500000$). Minden útnak, amely a bal felső csúcsból ($v_{1,1}$) egy $v_{x,y}$ csúcsba tart, azonos a súlya. Egy út súlya az út mentén található élek súlyainak az összege.

15.2. ábra -



A 15.2. ábrán látható gráfon, ahol $N = 3$, minden útnak, amely a $v_{1,1}$ csúcsból a $v_{3,2}$ csúcsba tart, a súlya 7. A $v_{1,1}$-ből a $v_{1,2}$-be vezető egyetlen út súlya 2.

Adott egy L egész szám ($1 \leq L \leq 2000000000$). A feladatod egy olyan $v_{x,y}$ csúcs megkeresése, amelyre a $v_{1,1}$-ből a $v_{x,y}$-ba tartó utak súlya pontosan L. A súlyokat nem közvetlenül adjuk meg a programodnak, ezért le kell kérdeznie azokat egy könyvtár segítségével. A program legfeljebb 6667-szer kérdezhet le súlyokat.

A könyvtár a következő függvényeket biztosítja: `getN()` visszaadja N értékét, `getL()` visszaadja L értékét, `getWeight(x, y, direction)` visszaadja a $v_{x,y}$-ból induló, a megadott irányba tartó él súlyát (ha `direction=0`, akkor a jobbra tartó él, ha `direction=1`, akkor a lefelé tartó él súlyát).

Ha megtaláltad a $v_{x,y}$ csúcsot, amelybe a $v_{1,1}$-ből vezető út pontosan L súlyú, meg kell hívnod a `solution(x, y)` eljárást. Ha nem létezik ilyen csúcs, akkor hívd meg a `solution(-1, -1)` eljárást. A programod automatikusan megáll, miután meghívtad a `solution(x, y)`-t. Ha több, mint 6667-szer hívod meg a `getWeight()` függvényt, vagy a megoldásod hibás, nem kapsz pontot az adott tesztesetre.

A feladathoz sem bemeneti, sem kimeneti állomány nem tartozik.

Könyvtári függvények

C/C++

int getN(void);
 	
int getL(void);
 	
int getWeight(int x, int y, int direction);
 	
void solution(int x, int y);

Pascal

function getN: Longint;
 	
function getL: Longint;
 	
function getWeight(x, y, direction: Longint): Longint;
 	
procedure solution(x, y: Longint);

A könyvtár egy példaimplementációját megtalálod a ~/ceoi vagy a c:\ceoi könyvtárban, bár a programod egy másik implementációval lesz kiértékelve.

A teszteléshez és nyomkövetéshez létrehozhatod a `square.in` állományt, amelyet a példakönyvtár olvashat majd. A `square.in` első sorában az N és L egész számoknak kell állniuk. A következő N sor mindegyike pontosan $N-1$ egész számot tartalmaz, a vízszintes élek súlyait. Az ezután következő $N-1$ sor mindegyike N egész számot tartalmaz, a függőleges élek súlyait.

A könyvtárt a következő módokon lehet csatolni a programhoz: #include "square_lib.h" vagy uses square_lib.

Megjegyzendő, hogy a példakönyvtár, amelyet a teszteléshez használhatsz, elég lassú és memóriaigényes. Ez azért van, mert be kell olvasnia az input állományodat a memóriába. Ettől függetlenül feltételezheted, hogy a kiértékelő könyvtárnak se memóriára, se futási időre nincs szüksége.

Példa input a könyvtárra

`square.in`	protokoll
3 4 1 2 2 3 1 1 2 3 4	getN() -> 3 getL() -> 4 getWeight(1,1,0) -> 1 getWeight(2,1,1) -> 3 solution(2,2)

5 4 2	

Idő- és memóriakorlát

Egy 650 MHz-es (vagy jobb) Pentium III processzorral és 128 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 1 másodperc, a memóriakorlát 16 MB.

A verseny

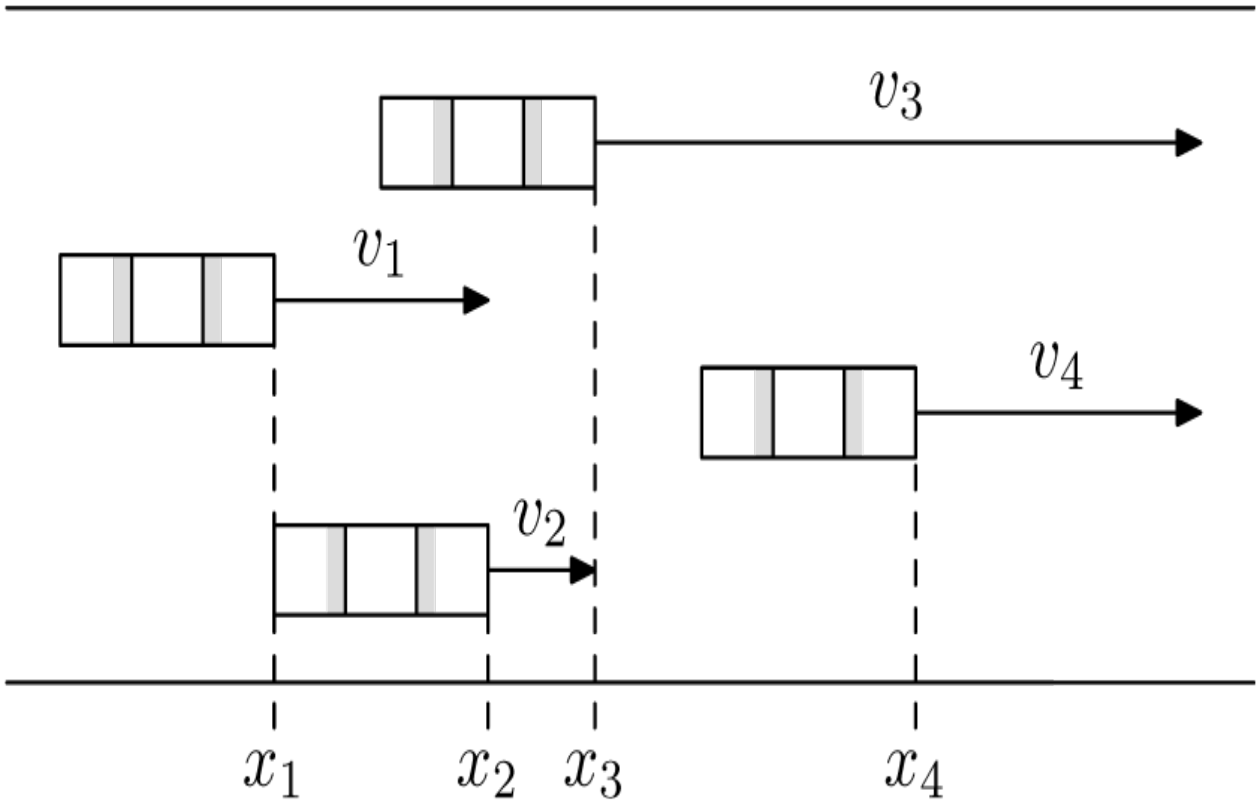
A Feltuningolt Űrhajók Éves Csillagközi Versenyén N űrhajó versenyez. Minden űrhajó úgy van tuningolva, hogy 0 idő alatt fel tud gyorsulni a maximális sebességére, és tartani tudja ezt a sebességet. Az i-edik űrhajó maximális sebessége v_i. Korábbi eredménye alapján az i-edik űrhajó az x_i kezdőpozícióból indul, amely a startvonaltól mért kilométerek számával van megadva.

A versenypálya végtelen hosszú. Az űrhajók nagy sebessége miatt a versenypálya végig egyenes. Ezen az egyenes pályán nagyon könnyen előzhetik egymást anélkül, hogy akadályoznák egymást.

A közönség soraiból sokan még nem jöttek rá, hogy a verseny kimenetele előre eldönthető. A te feladatod, hogy ezt megmutasd nekik azzal, hogy megmondod, hányszor fogják az űrhajók megelőzni egymást, és hogy megjósolod az első 10000 előzést időrendi sorrendben.

Feltehető, hogy minden űrhajó más-más pozícióból indul, továbbá sohasem lesz egyszerre kettőnél több űrhajó a pálya ugyanazon pontján.

15.3. ábra -



Input

A `therace.in` bemeneti állomány első sora a versenyző űrhajók számát (N) adja meg ($0 < N \leq 250000$). A következő N sor egy-egy űrhajó leírását tartalmazza. Az $(i+1)$-edik sor az i-edik hajót írja le az x_i és a v_i egész számokkal, amelyek az i-edik űrhajó kezdőpozícióját és sebességét jelentik ($0 \leq x_i \leq 1000000$, $0 < v_i < 100$). Az űrhajók kezdőpozícióik szerint vannak rendezve, azaz $x_1 < x_2 < \ldots < x_n$. A kezdőpozíció a startvonaltól mért kilométerek száma, ahonnan az űrhajó indul. A sebesség km/s-ban van megadva.

Output

A `therace.out` kimeneti állomány első sorának a versenyen előforduló előzések számának **1000000-val vett maradékát** kell tartalmaznia. Azzal, hogy az előzések számának csak az 1000000-val vett maradékát árulod el, egyrészt bizonyítod, hogy ismered ezt az értéket, másrészt nem rontod el a közönség soraiban lévő, kevésbé intelligens emberek örömét.

A következő sorok mindegyike egy-egy előzést adjon meg időrendi sorrendben. Ha 10000-nél több előzés lenne, csak az első 10000 előzést írd ki. Ha 10000-nél kevesebb előzés van, írd ki az összeset. Minden sornak két egész számot kell tartalmaznia (i-t és j-t), amely azt jelenti, hogy az i-edik űrhajó megelőzi a j-edik űrhajót. Ha egy időben több előzés történik, akkor azokat a versenyzők pozíciója szerint kell rendezni. Ez azt jelenti, hogy először a startvonalhoz közelebb eső előzést kell feltüntetni. Az előzés időpontja az az idő, amikor a két űrhajó ugyanazon pozícióban van.

Megjegyzések

Ha helyesen adod meg az előzések számát, tesztesetenként a pontok 40%-át kapod meg. Ha helyesen adod meg az első 10000 előzést, akkor megkapod a pontok maradék 60%-át. Minden részt külön értékelünk, feltéve, hogy a program az időhatáron belül megáll.

Példa

`therace.in`	`therace.out`
4 0 2 2 1 3 8 6 3	2 3 4 1 2

Idő- és memóriakorlát

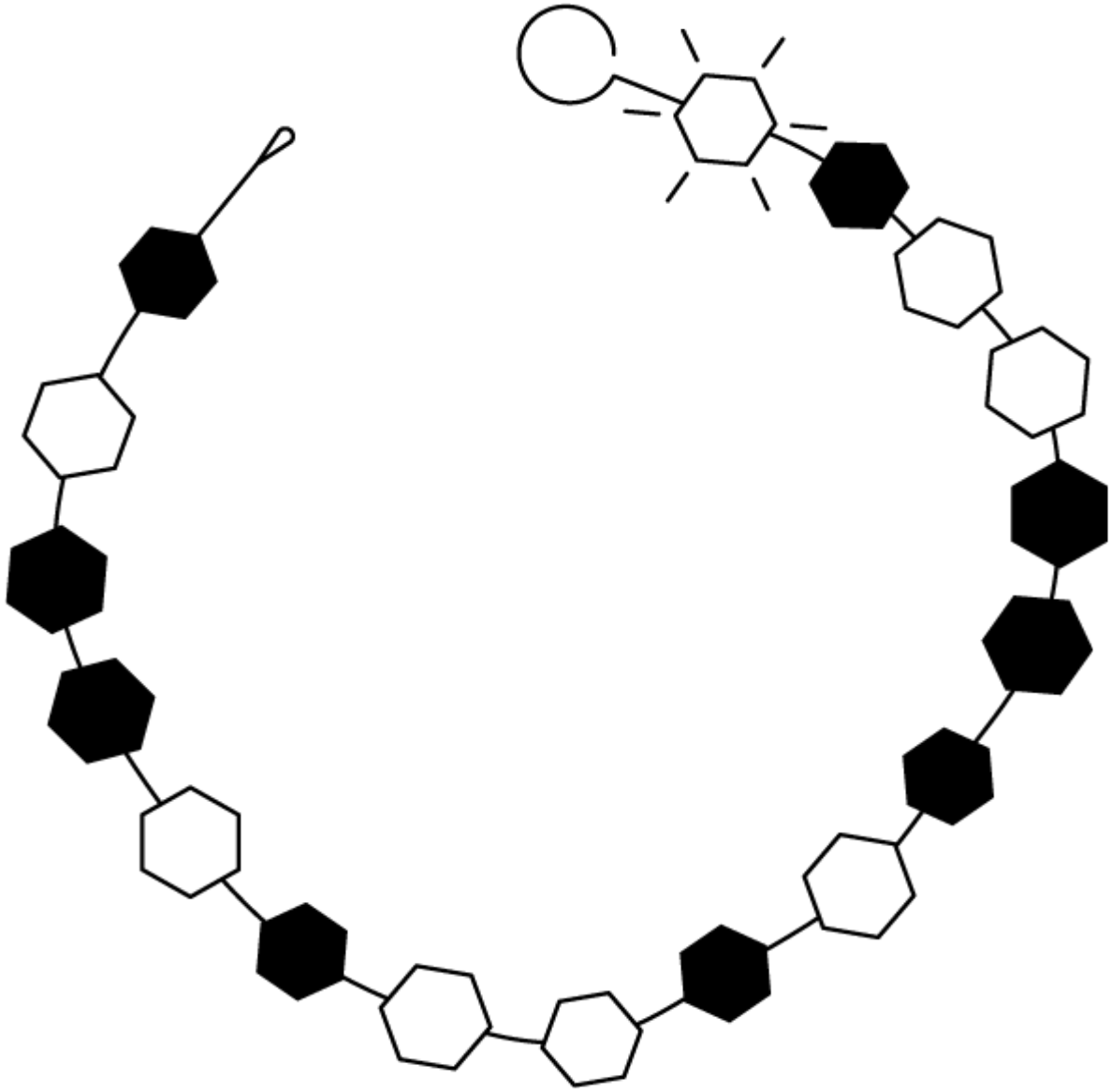
Egy 650 MHz-es (vagy jobb) Pentium III processzorral és 128 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 8 másodperc, a memóriakorlát 16 MB.

Gyöngy nyaklánc

Egy hegyen él két törpeklán: a vörös törpék és a zöld törpék. Egy közös expedíciójuk során zöld és

vörös törpék egy csoportja olyan nyakláncot talál, amely csak értéktelen fekete és fehér üveggyöngyökből áll, kivéve egy értékes gyémántot a lánc végén.

15.4. ábra -



Mindkét törpeklán szeretné megkaparintani a gyémántot. A törpék elhatározzák, hogy ezt a szituációt békés módon, a következő játékkal oldják meg:

Minden törpéhez hozzárendelünk egy egyedi azonosítót 1-től N-ig. Minden törpe kap két, mindenki által ismert listát, egy fehéret és egy feketét, amelyen törpeazonosítók vannak. (A törpék listái különbözhetnek egymástól.) Minden lista tartalmazhatja vörös és zöld törpék azonosítóit is. A játék alatt a nyakláncot egymásnak adogatják a következő szabályok szerint: Ha egy törpe megkapja a nyakláncot, leveszi az első gyöngyöt. Ha ez a gyöngy fehér, továbbadja a nyaklánc maradékát egy, a fehér listáján szereplő, általa kiválasztott törpének (aki saját maga is lehet). Ha a gyöngy fekete, akkor a fekete listáján szereplő egyik, általa választott törpének adja tovább a lánc maradékát (aki szintén lehet saját maga). A játék indításakor véletlenszerűen választják ki azt a törpét, aki először megkapja a nyakláncot.

A játék végén a nyaklánc már csak a gyémántot fogja tartalmazni. Az a törpe, aki ekkor kapja meg a nyakláncot, megtartja a saját klánjának, és a játék véget ér.

Írj programot, amely segít a zöld törpéknek megszerezni a gyémántot. Használd az alább leírt könyvtárat. Felteheted, hogy a vörös törpék optimálisan játszanak.

A könyvtár

Rendelkezésedre áll egy könyvtár, amely a következő függvényeket tartalmazza:

• `getNext()`: akkor kell meghívni, ha egy vörös törpe következik. Visszaadja annak a törpének az azonosítóját, akinek a vörös törpe átadja a nyakláncot.

• `setNext(d)`: akkor kell meghívni, ha egy zöld törpe következik. A d paraméter annak a törpének az azonosítója, akinek a zöld törpe át fogja adni a nyakláncot.

• `finish()`: akkor kell meghívni, amikor vége van a játéknak. Megállítja a programodat.

A programod tesztelésére rendelkezésedre állnak a könyvtár tesztváltozatai. A könyvtár pearls_lib.h és pearls_lib.pas forrásállományait a /home/ceoi/ vagy a c:\ceoi könyvtárban találod. A könyvtár ezen változatában a vörös törpék a nyakláncot mindig a megfelelő lista elején szereplő törpének adják tovább.

C/C++ specifikációk

Használd az #include "pearls_lib.h" direktívát a következő függvényeket biztosító könyvtár eléréséhez:

```
int getNext(void);
void setNext(int d);
void finish(void);
```

Pascal specifikációk

Használd az uses pearls_lib; utasítást a következő függvényeket biztosítókönyvtár eléréséhez:

```
function getNext: Integer;
procedure setNext(d: Integer);
procedure finish;
```

Input

A `pearls.in` bemeneti állomány első sora a nyaklánc L kezdeti hosszát, a törpék N számát, valamint annak a törpének az F azonosítóját tartalmazza, aki először megkapja a nyakláncot ($1 \le L \le 1000$, $1 \le N \le 1000$, $1 \le F \le N$). Bármely törpe i azonosítójára $1 \le i \le N$ teljesül.

A második sor L karaktert tartalmaz, amelyek a nyakláncot írják le. Az első $L-1$ karakter vagy a `B`, vagy a `W` betű. A `B` fekete gyöngyöt jelöl, a `W` pedig fehéret. Az utolsó karakter a `D` betű, amely a gyémántot jelöli.

A következő N sor a törpéket írja le. Ezek közül az i-edik sor az i azonosítójú törpe adatait tartalmazza. Ezek a sorok egy számmal kezdődnek, amely a törpe színét határozza meg: zöld törpe esetén 0, vörös törpe esetén 1. Ezután következik a törpe fekete listájának L_B hossza ($1 \le L_B \le 20$), majd a fekete listán szereplő törpék azonosítói. A listát a törpe fehér listájának L_W hossza követi ($1 \le L_W \le 20$), valamint a fehér listán szereplő L_W darab törpeazonosító.

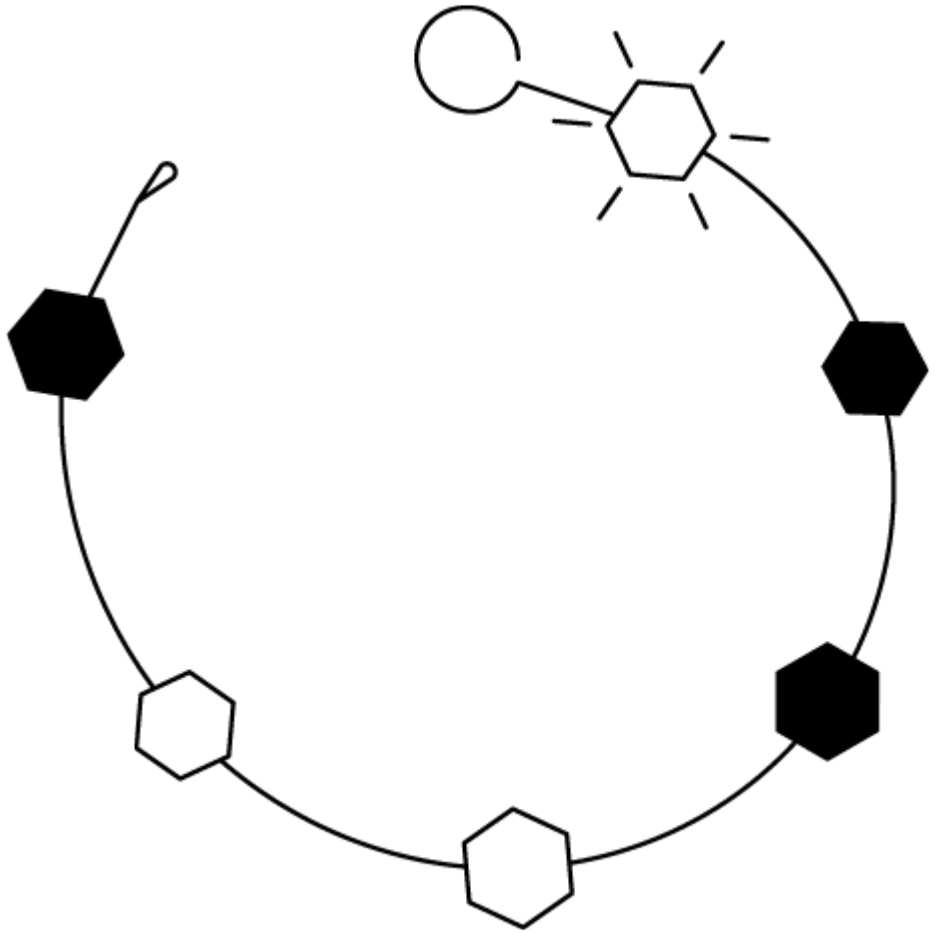
Output

Nincs.

Példa

pearls.in	Könyvtári hívás
6 4 2 BWWBBD 0 1 2 1 4 0 2 1 3 1 1 1 1 4 1 4 1 2 2 3 1 1	setNext(1) setNext(4) getNext() -> 1 setNext(2) setNext(1) finish()

15.5. ábra -



Értékelés

Ha nem hívod meg a `finish()`-t, vagy meghívod a `setNext(d)`-t, amikor nem zöld törpe következik, vagy a d törpe nincs a listán, vagy meghívod a `getNext()`-et, amikor nem vörös törpe következik, 0 pontot kapsz. Akkor is 0 pontot kapsz, ha a `finish()` hívásakor vörös törpénél van a gyémánt, vagy a nyaklánc a gyémánton kívül további gyöngyszemeket is tartalmaz. Csak akkor kapod meg a maximális pontszámot, ha a végén zöld törpénél van a gyémánt. Minden

tesztesetet úgy terveztünk meg, hogy ez elérhető.

Idő- és memóriakorlát

Egy 650 MHz-es (vagy jobb) Pentium III processzorral és 128 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 2 másodperc, a memóriakorlát 16 MB.

Shift regiszter

Egy számítógép egy regisztere N bitet tárol a számításokhoz. A shift regiszter egy speciális regiszter, amelynek a bitjei könnyen eltolhatók egy pozícióval.

Egy visszaható shift regiszter használatával bináris pszeudo-véletlenszámok generálhatók a következő módon: Egy N méretű shift regiszter kezdetben az $a_1, a_2, \ldots, a_N$ bitértékekkel van feltöltve. A regiszter minden órajelciklusban a kimenetére írja a jobb szélső bit (a_N) értékét. A többi bitérték egy pozícióval jobbra tolódik. Az első pozíció új értéket (a_1) kap a következőképpen:

A regiszter minden bitje egy XOR-kapura van rákötve egy kapcsolón keresztül (lásd a 15.6. ábrát). Az i-edik bithez az s_i kapcsoló tartozik (értéke 1 vagy 0 lehet), amely eldönti, hogy az a_i bitérték továbbítódik-e a XOR-kapuhoz, vagy sem. Legyen $k_i = s_i \cdot a_i$. Az a_1 új érték a XOR-kapu kimeneti értéke, $XOR(k_1, \ldots, k_N)$ lesz. (Megjegyzés: Ha a $k_1, \ldots, k_N$ értékek között szereplő egyesek száma páratlan, a $XOR(k_1, \ldots, k_N)$ értéke 1, különben 0.) Íme a formális definíciók:

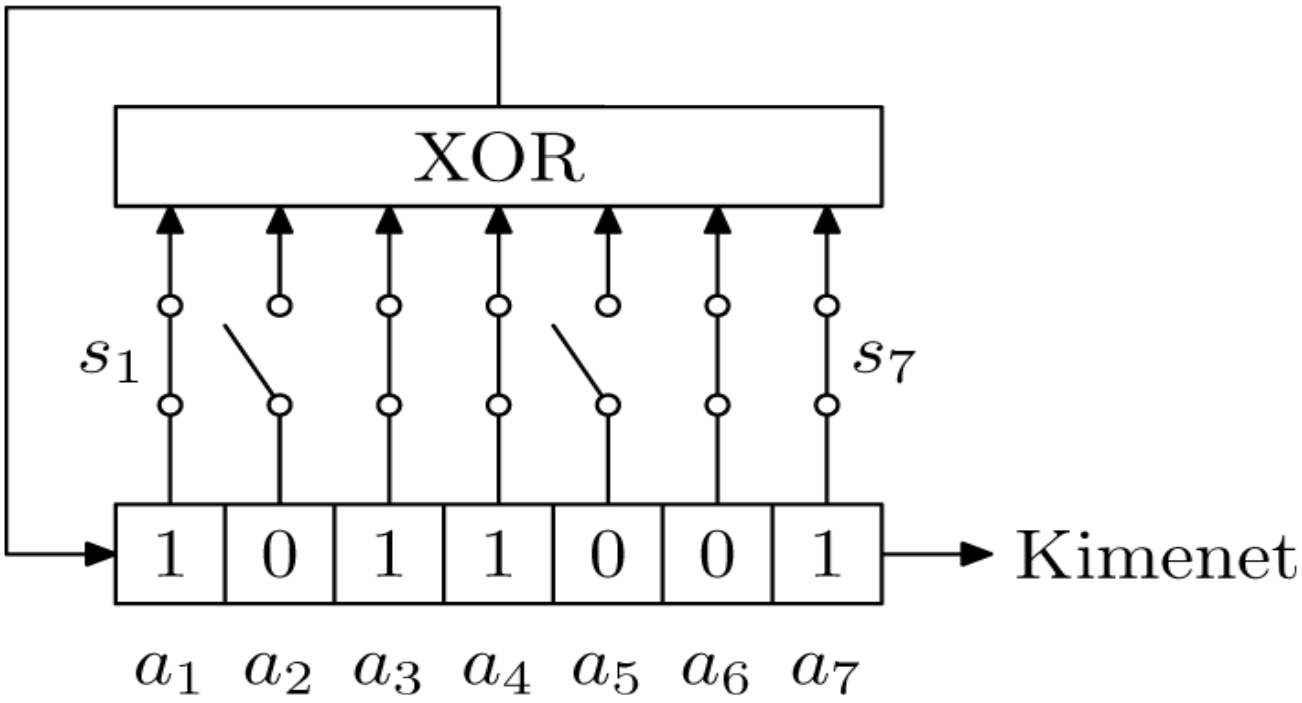
$a_1 = XOR(k_1, \ldots, k_N)$

$a_i = $ 2 i N $esetén kimenet = a_N$

órajel	a_1	a_2	a_3	a_4	a_5	a_6	a_7	kimenet
0	1	0	1	1	0	0	1	--
1	0	1	0	1	1	0	0	1
2	1	0	1	0	1	1	0	0
3	1	1	0	1	0	1	1	0
4	0	1	1	0	1	0	1	1
5	0	0	1	1	0	1	0	1
6	1	0	0	1	1	0	1	0
7	1	1	0	0	1	1	0	1
8	0	1	1	0	0	1	1	0
9	1	0	1	1	0	0	1	1

10	0	1	0	1	1	0	0	1
11	1	0	1	0	1	1	0	0
12	1	1	0	1	0	1	1	0
13	0	1	1	0	1	0	1	1
14	0	0	1	1	0	1	0	1

15.6. ábra -



A fenti példában a_1 értékét az első órajelciklusban a következőképpen számítjuk: XOR($1 \cdot 1, 0 \cdot 0, 1 \cdot 1, 1 \cdot 1, 0 \cdot 0, 1 \cdot 0, 1 \cdot 1) = 0$.

Adott egy ilyen visszaható shift regiszter első $2N$ kimeneti értéke. Ezekből az értékekből meg kell próbálnod meghatározni az s_i kapcsolóértékeket.

Input

A `register.in` bemeneti állomány első sora a shift regiszter N méretét tartalmazza (

$1 \leq N \leq 750$). A második sor $2N$ darab számból áll (0 vagy 1), amelyek a shift regiszter első $2N$ darab kimeneti bitértékét jelentik.

Output

A `register.out` kimeneti állomány pontosan egy sorból álljon. Ha a kapcsolóknak léteznek olyan állásai, amelyek a megadott kimeneti értékeket eredményezik, írasd ki bármelyik ilyen beállításhoz tartozó s_i kapcsolóértékeket, s_1-gyel kezdve. Ha nem létezik ilyen beállítás, csak a -1 számot írd ki.

Példák

`register.in`	`register.out`
7	1 0 1 1 0 1 1
1 0 0 1 1 0 1 0 1 1 0 0 1 1	
`register.in`	`register.out`
3	-1
0 0 0 1 1 1	

Idő- és memóriakorlát

Egy 650 MHz-es (vagy jobb) Pentium III processzorral és 128 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 1.5 másodperc, a memóriakorlát 16 MB.

Kirándulás

Alice és Bob szeretnének elmenni nyaralni. Mindketten megterveztek egy útvonalat, amely azon városok egy adott sorrendű listája, amelyeket meg szeretnének látogatni. Egy útvonalon egy város többször is szerepelhet.

Mivel együtt akarnak utazni, meg kell állapodniuk egy közös útvonalban. Egyikük sem akarja megváltoztatni a listájukon szereplő városok sorrendjét, sem új városokat felvenni. Ezért nincs más választásuk, mint levenni néhány várost az útvonalról. Természetesen a közös útvonalnak olyan hosszúnak kell lennie, amennyire csak lehetséges.

Pontosan 26 város van a környéken, ezért a listákon az angol ábécé kisbetűivel vannak kódolva, `a`-tól `z`-ig.

Input

A `trip.in` bemeneti állomány két sorból áll: az első sor Alice listája, a második Bob listája. Mindkét lista legalább 1, legfeljebb 80 darab kisbetűből áll, szóközök nélkül.

Output

A `trip.out` kimeneti állománynak az összes olyan útvonalat tartalmaznia kell, amely megfelel a fent leírt követelményeknek, de egyik útvonal sem szerepelhet többször. Minden útvonalat külön sorba kell kiírni. Legalább egy ilyen nem üres útvonal létezik, de nincs 1 000-nél több különböző útvonal. Az útvonalak sorrendje a kimeneti állományban nem számít.

Példa

trip.in	trip.out
abcabcaa acbacba	ababa abaca abcba acbca acaba acaca acbaa

Idő- és memóriakorlát

Egy 650 MHz-es (vagy jobb) Pentium III processzorral és 128 MB (vagy több) RAM-mal rendelkező számítógépen az időkorlát 7 másodperc, a memóriakorlát 16 MB.

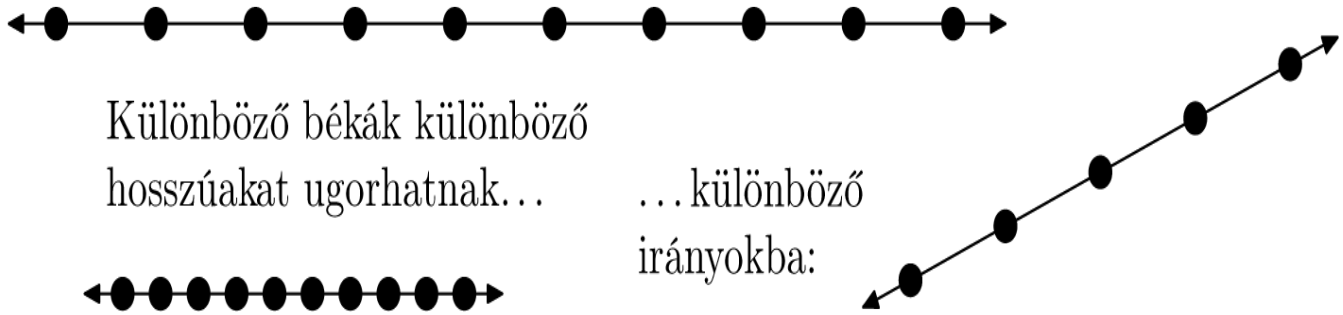
16. fejezet - Nemzetközi Informatikai Diákolimpia, 2002, Yong-In, Dél-Korea

Tartalom

A neveletlen béka

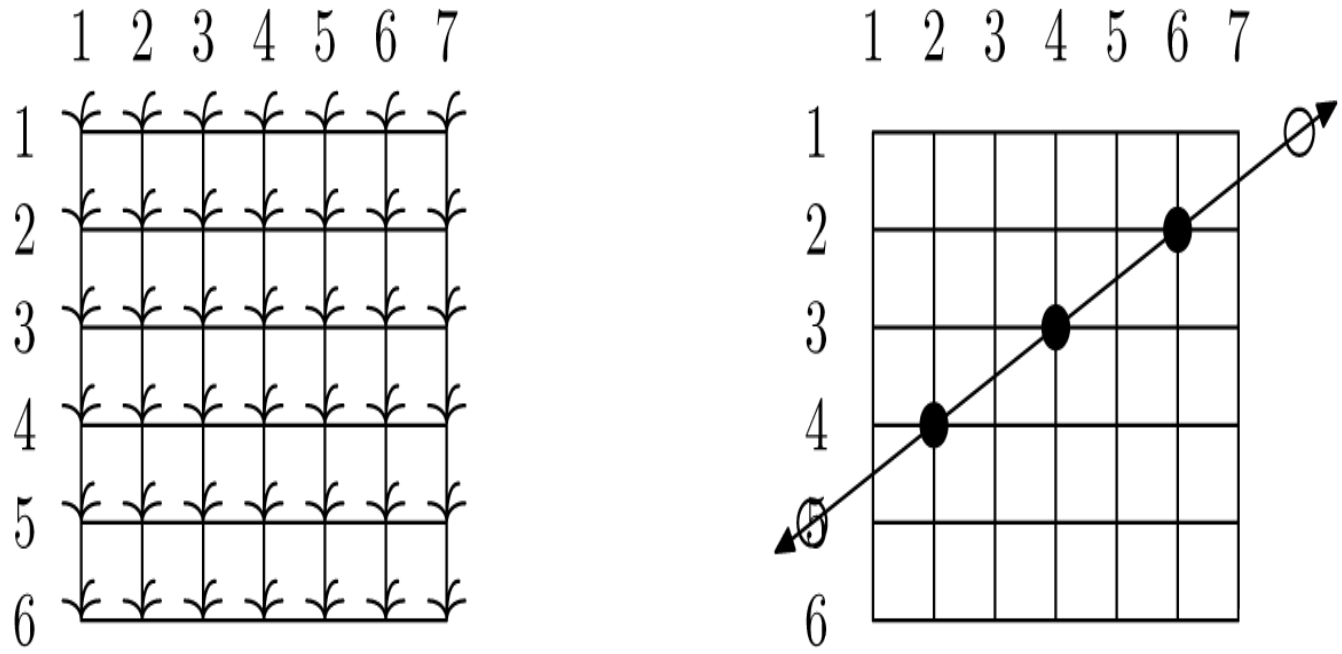
Koreában legendás a *cheonggaeguri* nevű béka szemtelensége. Ez jól kiérdemelt hírnév, mert a békák éjjelente keresztülugrálnak a rizsföldeken, lelapítva a rizsnövényeket. Reggel, miután megfigyeltük, mely növényeket lapították le, szeretnénk beazonosítani annak a békának az útvonalát, amelyik a legnagyobb kárt okozta. A békák mindig egyenes vonalban ugrálnak keresztül a rizsföldön, és minden ugrásuk azonos hosszúságú (lásd a 16.1. ábrát).

16.1. ábra -



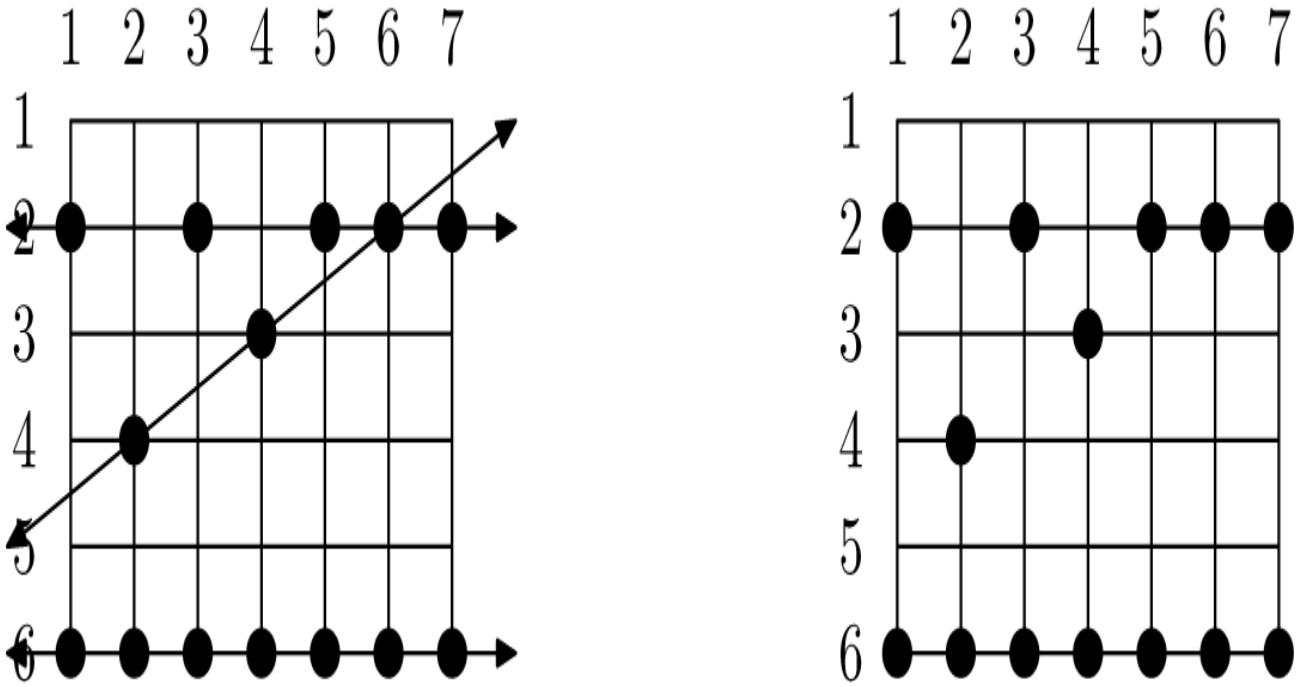
A rizsföldünkön a növények egy rács metszéspontjaiban helyezkednek el, ahogy a 16.2. ábra bal oldalán látható. A neveletlen békák teljesen keresztülugrálnak a rizsföldön, kívülről indulva az egyik oldalon, és kívülre érkezve a másik oldalon, az ábra jobb oldalán látható módon.

16.2. ábra -



Sok béka haladhat át az ültetvényen, növényről növényre ugrálva. Minden ugrás egy növényen végződik, amelyet lelapít, ahogy a 16.3. ábra bal oldalán látható. Egyes növényekre egynél több béka is ugorhat az éjszaka folyamán. Természetesen a békák útvonalát jelző egyenesek nem láthatók, mint ahogy az ültetvényen kívüli ugrásaik sem -- a 16.3. ábra bal oldalán szereplő példa esetén az ábra jobb oldalán berajzolt pontokat látjuk.

16.3. ábra -



A 16.3. ábra jobb oldalából rekonstruálhatjuk az összes lehetséges útvonalat, amelyeken a békák áthaladhattak a rizsföldön. Csak azok a békák érdekelnek, amelyek legalább 3 növényt lelapítottak útjuk során. Az ilyen útvonalat békaútvonalnak nevezzük. Ezek alapján a 16.3. ábra bal oldalán szereplő három útvonal békaútvonal (vannak további lehetséges békaútvonalak is). Az első oszlopban lévő függőleges útvonal egy 4 ugráshosszúságú békaútvonal lehetne, de csak két lelapított növény van, ezért számunkra nem érdekes. A 2. sor 3. oszlopában, a 3. sor 4. oszlopában és a 6. sor 7. oszlopában lévő növényeket tartalmazó átlós útvonal három lelapított növényből áll, de nincs olyan szabályos ugráshossz, amely alapján ezek az ugrások kialakulhatnának úgy, hogy továbbra is legalább 3 növényt érintsenek, ezért ez sem békaútvonal. Vegyük figyelembe, hogy egy békaútvonalhoz tartozó egyenesen további lelapított növények is lehetnek, amelyekre nem történik ugrás az adott útvonalon (ilyen például a 16.3. ábra jobb oldalán a $(2,6)$ koordinátákon található növény, amely a második soron végighaladó vízszintes útvonalon helyezkedik el). Valójában létezhetnek olyan lelapított növények is, amiket egyik békaútvonal sem magyaráz meg.

A feladatod egy olyan program írása, amely meghatározza az egy békaútvonalon előforduló növények maximális számát (a maximumot az összes lehetséges békaútvonalat figyelembe véve kell számítani). A 16.3. ábra jobb oldalához tartozó válasz 7, amelyet a 6. soron végighaladó békaútvonal alapján kapunk.

Input

A programodnak a bemenetet a standard inputról kell olvasnia. Az első sor az R és a C egész számokat tartalmazza, amelyek rendre a rizsföld sorainak és oszlopainak a száma ($1 \le R, C \le 5000$). A második sor egy N egész számból áll, amely a lelapított növények száma ($3 \le N \le 5000$). A maradék N sor mindegyike két egész számot tartalmaz, egy-egy lelapított növény sorát ($1 \le sor \le R$) és oszlopát ($1 \le oszlop \le C$), egy szóközzel elválasztva. Minden lelapított növény csak egyszer van felsorolva.

Output

A programodnak a standard kimenetre kell írnia. A kimenet egy sorból álljon, amelybe egy egész számot kell írni, a legnagyobb kárt okozó békaútvonal mentén elhelyezkedő lelapított növények számát, ha létezik legalább egy békaútvonal, különben 0-t.

Példák

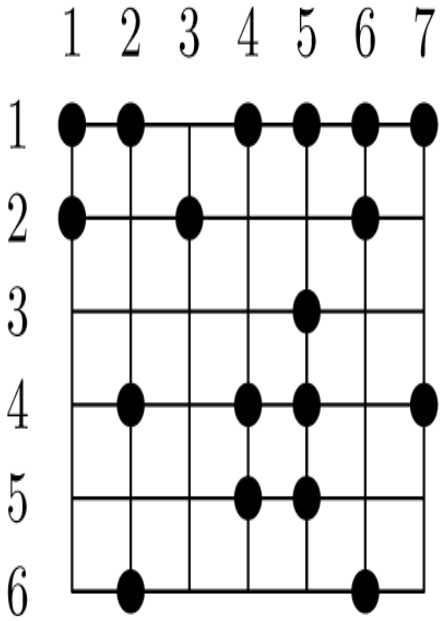
1. példa

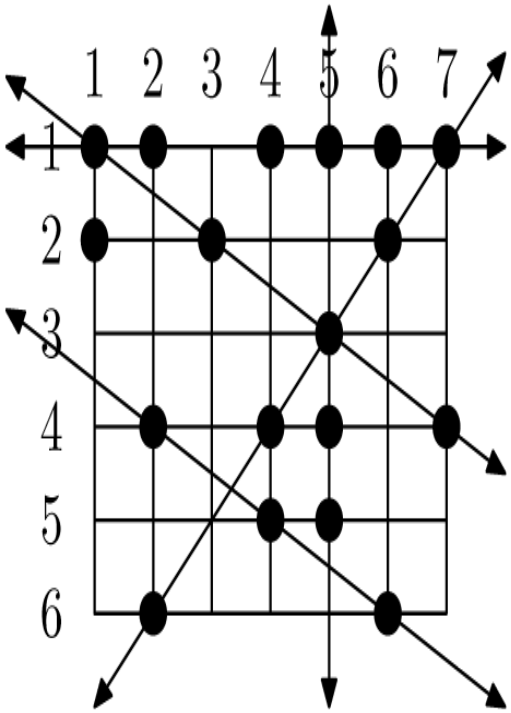
input	output
6 7 14 2 1 6 6 4 2 2 5 2 6 2 7 3 4 6 1 6 2 2 3 6 3 6 4 6 5 6 7	7

A példához tartozó békaútvonalakat és a lelapított növények helyeit a 16.3. ábrán láthatod.

2. példa

16.4. ábra -





input	output
6 7 18 1 1 6 2 3 5 1 5 4 7 1 2 1 4 1 6 1 7	4

2 1

2 3

2 6

4 2

4 4

4 5

5 4

5 5

6 6

A 16.4. ábrán a lelapított növények helyei és a belőlük meghatározható békaútvonalak láthatók.

Pontozás

Ha a programod egy tesztesetre kiírja a helyes választ az időkorláton belül, akkor a teljes pontszám jár az adott tesztesetre, különben 0 pontot kapsz.

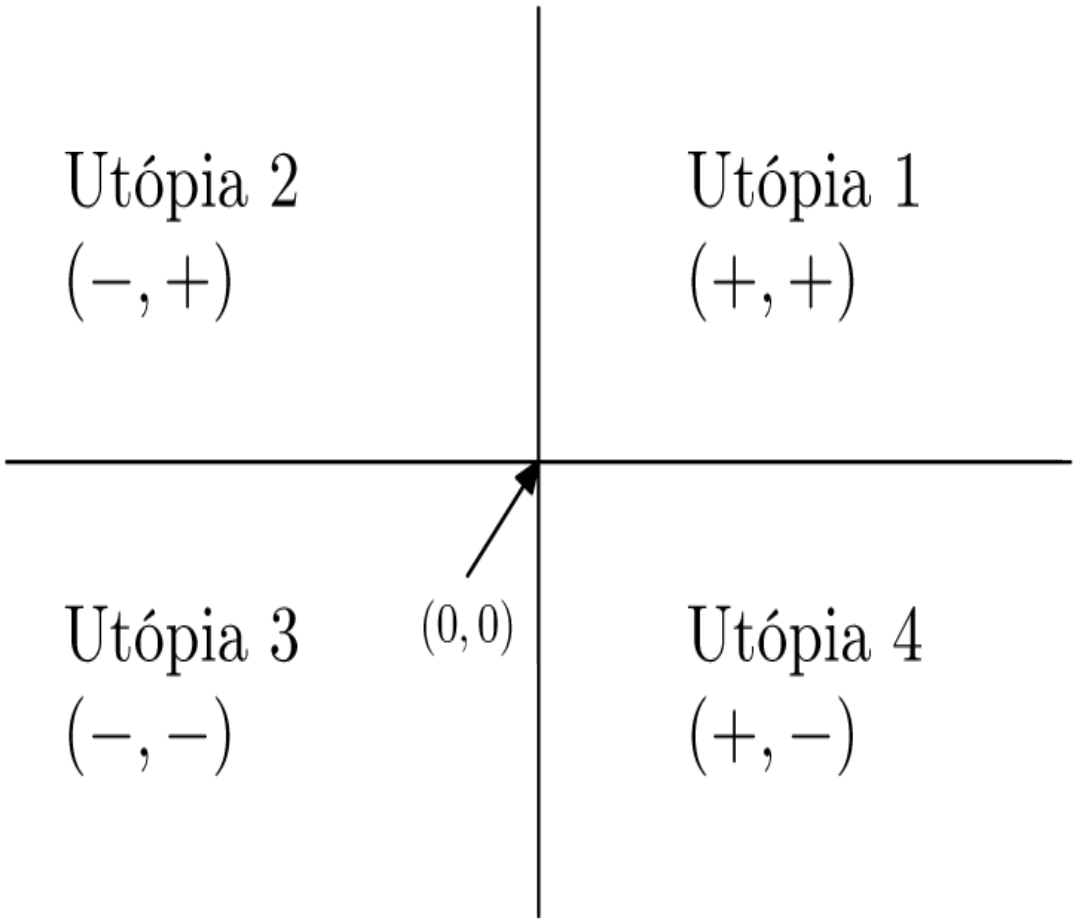
Idő- és memóriakorlát

Egy $1,7$ GHz-es Pentium 4 processzorral és 256 MB RAM-mal rendelkező számítógépen az időkorlát 2 másodperc, a memóriakorlát 64 MB.

A felosztott Utópia

Egyszer Utópia gyönyörű földjét lerombolta egy háború. Amikor a háborúskodás alábbhagyott, az országot négy régióra osztották egy földrajzi hosszúság (észak-déli vonal) és egy földrajzi szélesség (kelet-nyugati vonal) mentén. A két vonal metszete a $(0,0)$ pontként lett ismert. Mind a négy régió igényt tartott az Utópia névre, de az idő múltával Utópia 1 (északkelet), 2 (északnyugat), 3 (délnyugat) és 4 (délkelet) néven váltak ismertté. Az egyes régiók pontjait a $(0,0)$ ponttól mért keleti és északi irányú távolságukkal azonosították. Ezek a távolságok negatívak is lehettek; így egy Utópia 2-beli pontot egy (negatív, pozitív), egy Utópia 3-belit egy (negatív, negatív), egy Utópia 4-belit egy (pozitív, negatív) párral, egy Utópia 1-beli pontot pedig egy pozitív számpárral jelöltek.

16.5. ábra -



Az egyik fő probléma az volt, hogy az állampolgárok nem léphették át a határokat. Szerencsére néhány találékony utópiai IOI-versenyző kifejlesztett egy biztonságos teleportáló eszközt. A gép kódszámokat kér, amelyeket csak egyszer lehet felhasználni. Most az a kihívás vár rád, hogy eljuttasd a teleportálót a $(0,0)$ kezdeti pozíciójából Utópia régióiba a kívánt sorrendben. Az nem számít, hogy a régiókon belül hol landolsz, csak egy N darab régiószámból álló sorozat áll rendelkezésedre, amelyek megadják azokat a régiókat, ahová a teleportálót el kell juttani. Elképzelhető, hogy ugyanabban a régióban kettő vagy több egymást követő alkalommal is landolnod kell. Miután elhagyod a $(0,0)$ kezdőpontot, sosem landolhatsz egyetlen határon sem.

Bemenetként egy $2N$ darab kódszámból álló sorozatot fogsz kapni, amelyet egy N darab kódpárból álló sorozattá kell átírnod, minden egyes szám elé vagy egy plusz, vagy egy mínusz jelet írva. Ha jelenleg az (x,y) pontban vagy, és a $(+u,-v)$ kódpárt használod, akkor az $(x+u,y-v)$ pontba fogsz teleportálódni. A megkapott $2N$ számot tetszőleges sorrendben felhasználhatod, akár plusz, akár mínusz előjellel.

Tegyük fel, hogy a 7, 5, 6, 1, 3, 2, 4, 8 kódszámaid vannak, és a 4, 1, 2, 1 régiószám-sorozat alapján kell irányítanod a teleportálót. Ez a $(+7,-1)$, $(-5,+2)$, $(-4,+3)$, $(+8,+6)$ kódpársorozattal érhető el, mivel így a $(0,0)$ pontból a $(7,-1)$, a $(2,1)$, a $(-2,4)$ és a $(6,10)$ pontokba fogsz eljutni a megadott sorrendben. Ezek a pontok rendre az Utópia 4-ben, az Utópia 1-ben, az Utópia 2-ben és az Utópia 1-ben helyezkednek el.

Feladat

Adott $2N$ különböző kódszám, és egy N régiószámból álló sorozat, amely megadja, hogy hol kell a teleportálónak landolnia. A megadott számokból készíts egy kódpársorozatot, amely úgy irányítja a teleportálót, hogy az végighaladjon a megadott régiósorozaton.

Input

A programodnak a bemenetet a standard inputról kell olvasnia. Az első sor egy N pozitív egész számot tartalmaz ($1 \le N \le 10000$). A második sor $2N$ darab különböző egész kódszámból áll ($1 \le kódszám \le 100000$), egy-egy szóközzel elválasztva. Az utolsó sor egy N régiószámból álló sorozatot tartalmaz, ahol a régiószámok 1, 2, 3 vagy 4 lehetnek.

Output

A programodnak a kimenetet a standard outputra kell írnia. A kimenet N sorból álljon, és ezek mindegyike előjellel ellátott kódszámokból álló párokat tartalmazzon. Ezek azok a kódpárok, amelyek végigvezetik a teleportálót a megadott régiósorozaton. Az előjel után nem állhat szóköz, az első kódszám után viszont pontosan egy szóköznek kell állnia.

Ha több megoldás létezik, a programod bármelyiket kiírhatja. Ha nincs megoldás, akkor a 0 számot kell kiírnia.

Példák

1. példa

input	output
4	+7 -1
7 5 6 1 3 2 4 8	-5 +2
4 1 2 1	-4 +3
	+8 +6

2. példa

input	output
4	+3 -2
2 5 4 1 7 8 6 3	-4 +5
4 2 2 1	-6 +1
	+8 +7

Pontozás

Ha a programod egy tesztesetre kiírja a helyes választ az időkorláton belül, akkor a teljes pontszám jár az adott tesztesetre, különben 0 pontot kapsz.

Idő- és memóriakorlát

Egy $1,7$ GHz-es Pentium 4 processzorral és 256 MB RAM-mal rendelkező számítógépen az

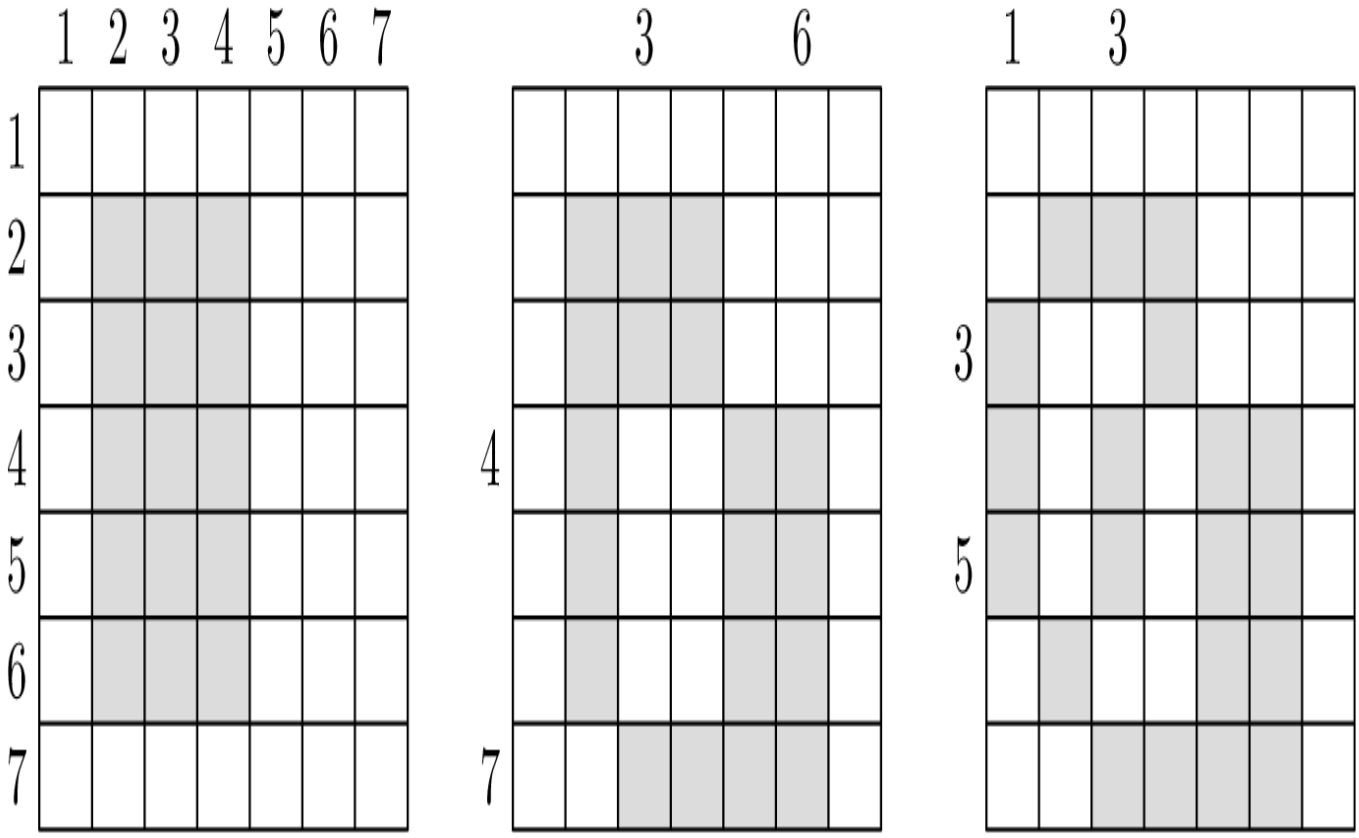
időkorlát 2 másodperc, a memóriakorlát 32 MB.

XOR

Egy fekete-fehér képernyővel rendelkező mobiltelefonra készítesz alkalmazást. A képernyő x-koordinátáit balról számozzuk, az y-koordinátákat pedig felülről, ahogyan a 16.6. ábrán látható. Az alkalmazáshoz különböző képekre van szükséged, amelyek nem mind egyforma méretűek. Ahelyett, hogy tárolnád a képeket, a telefon grafikus könyvtárát használva szeretnéd őket elkészíteni. Feltételezheted, hogy egy kép rajzolásának kezdetekor a képernyő minden pixele fehér. A telefon könyvtárának egyetlen grafikus művelete a `XOR(L,R,T,B)`, amely megfordítja a pixelértékeket az (`L`,`T`) bal felső és az (`R`,`B`) jobb alsó koordinátájú téglalapban, ahol `L` a bal, `T` a felső, `R` a jobb, `B` pedig az alsó koordináta. Más grafikus könyvtárakban az argumentumok sorrendje ettől eltérő lehet.

Vegyük például a 16.6. ábra jobb oldalán látható képet. Ha egy teljesen fehér képre alkalmazzuk a `XOR(2,4,2,6)` műveletet, akkor az 16.6. ábra bal oldalán látható képet kapjuk. Erre a képre alkalmazva a `XOR(3,6,4,7)` műveletet, a középső képet kapjuk, és végül erre alkalmazva a `XOR(1,3,3,5)` műveletet, megkapjuk a jobb oldalon látható képet.

16.6. ábra -



Adott fekete-fehér képek egy halmaza. A feladatod, hogy egy kezdetben fehér képernyőn előállítsd az egyes képeket a lehető legkevesebb `XOR`-hívást felhasználva. Adottak a képeket leíró bemeneti állományok. Azokat az állományokat kell beküldened, amelyek a szükséges `XOR`-hívások paramétereit tartalmazzák, nem pedig az ezen állományokat előállító programot.

Input

A `xor1.in`, , `xor10.in` nevű szöveges állományokban adott 10 problémaleírás. Minden bemeneti állomány a következőképpen néz ki. Az első sor az N egész számot tartalmazza ($5 \le N \le 2000$), amely a kép sorainak és oszlopainak a száma. A további sorok a kép sorait adják meg, felülről lefelé haladva. Minden sor N egész számból áll: a sorban lévő pixelek értékei balról jobbra. Ezen számok mindegyike vagy 0, vagy 1, ahol a 0 a fehér, az 1 pedig a fekete pixel.

Output

10 kimeneti állományt kell beküldened, amelyek a megadott bemeneti állományokhoz tartoznak.

Az első sor a

```
#FILE xor I
```

szöveget tartalmazza, ahol az `I` egész szám a megfelelő bemeneti állomány sorszáma. A második sorban egy K egész szám szerepeljen, amely az állományban leírt `XOR`-hívások száma. A következő K sor ezeket a hívásokat adja meg az elsőnek végrehajtandó hívástól az utolsóig. Ezen K sor mindegyike négy egész számot tartalmazzon: a `XOR`-hívás paramétereit `L`, `R`, `T`, `B` sorrendben.

Példa

`xor0.in`	`xor0.out`
7	#FILE xor 0
0 0 0 0 0 0 0	3
0 1 1 1 0 0 0	2 4 2 6
1 0 0 1 0 0 0	3 6 4 7
1 0 1 0 1 1 0	1 3 3 5
1 0 1 0 1 1 0	
0 1 0 0 1 1 0	
0 0 1 1 1 1 0	

Pontozás

Ha

• a kimeneti állományban leírt `XOR`-hívások nem a kívánt képet állítják elő, vagy

• a kimeneti állományban leírt `XOR`-hívások száma nem K, vagy

• a kimeneti állományban $K > 40000$, vagy

• a kimeneti állomány olyan `XOR`-hívást tartalmaz, amelyben $L > R$ vagy $T > B$, vagy

• a kimeneti állomány olyan `XOR`-hívást tartalmaz, amelyben nem pozitív koordináták szerepelnek, vagy

• a kimeneti állomány olyan XOR-hívást tartalmaz, amelyben egy N-nél nagyobb koordináta szerepel,

akkor 0 pontot kapsz. Különben a pontjaid száma

$$1+9\cdot\frac{\text{alegjobbmegoldásbanszereplõhívásokszáma}}{\text{ateválaszodbanszereplõhívásokszáma}}.$$

A pontszámot minden tesztesetnél egy tizedesjegyre, a végén az összpontszámot a legközelebbi egészre kerekítjük.

Tegyük fel, hogy a megoldásodban 121 XOR-hívás szerepel. Ha ez a legjobb megoldás az összes versenyző megoldásai közül, akkor 10 pontot kapsz. Ha az összes versenyző megoldásai közül a legjobb megoldás 98 XOR-hívást tartalmaz, akkor a pontjaid száma

$$1+9\cdot\frac{98}{121}\quad(=8{,}289\ldots),$$

amelyet $8{,}3$-re kerekítünk.

Idő- és memóriakorlát

Nincs.

Kötegütemezés

Adva van egy N feladatból álló sorozat, amelyet egyetlen gépen kell feldolgozni. A feladatokat 1-től N-ig sorszámozzuk, ezért a sorozat $1,2,\ldots,N$ lesz. A feladatsorozatot egy vagy több kötegre kell felosztani, ahol minden egyes köteg a sorozatban egymást követő feladatokból áll. A feldolgozás a 0 időpillanatban kezdődik. A kötegeket egyesével kell kezelni az első köteggel kezdve, a következőképpen. Ha egy b köteg tartalmaz kisebb sorszámú feladatokat, mint egy c köteg, akkor a b kötegeket a c köteg előtt kell feldolgozni. Az egy kötegben lévő feladatokat a gép egymás után dolgozza fel. Közvetlenül azután, hogy egy köteg összes feladata feldolgozásra került, a gép kiírja az adott köteg összes feladatának eredményét a kimenetre. Egy j feladat kimeneti idejének azt az időt nevezzük, amikor a j-t tartalmazó köteg befejeződik.

Minden egyes köteg indításakor szükséges egy S beállítási idő a gép beállításához. Minden egyes i feladatnak tudjuk az F_i költségtényezőjét, és a T_i feldolgozási idejét. Ha egy köteg az $x, x+1,\ldots,x+k$ feladatokat tartalmazza, és a t időpillanatban kezdődik, akkor az adott kötegben lévő minden egyes feladat kimeneti ideje $t+S+(T_x+T_{x+1}+\ldots+T_{x+k})$. A gép a kötegben lévő összes feladat eredményét is ebben az időpillanatban írja ki. Ha az i feladatnak O_i a kimeneti ideje, akkor a költsége $O_i\times F_i$. Tegyük fel például, hogy van öt feladat, a beállítási idő $S=1$, $(T_1,T_2,T_3,T_4,T_5)=(1,3,4,2,1)$ és $(F_1,F_2,F_3,F_4,F_5)=(3,2,3,3,4)$. Ha a feladatokat három kötegbe osztjuk ($\{1,2\}$, $\{3\}$, $\{4,5\}$), akkor a kimeneti idők $(O_1,O_2,O_3,O_4,O_5)=(5,5,10,14,14)$, míg a feladatok költségei rendre $(15,10,30,42,56)$. A felosztás összköltsége a feladatok költségeinek az összege. A fenti példában leírt felosztás összköltsége 153.

A feladatod egy olyan program írása, amely kiszámítja a lehetséges legkisebb összköltséget, ha adott egy beállítási idő és egy feladatsorozat feldolgozási időkkel és költségtényezőkkel együtt.

Input

A programod a bemenetet a standard inputról olvassa. Az első sor a feladatok N számát tartalmazza ($1 \le N \le 10000$). A második sorban az S beállítási idő szerepel egész számként megadva ($0 \le S \le 50$). A következő N sor rendre az $1, 2, \ldots, N$ feladatokról tartalmaz információkat az alábbiak szerint. Ezekben a sorokban először is egy T_i egész szám áll ($1 \le T_i \le 100$), amely a feladat feldolgozási ideje. Ezután egy F_i egész szám szerepel ($1 \le F_i \le 100$), amely a feladat költségtényezője.

Output

A programod a kimenetet a standard outputra írja. A kimenet egy sorból álljon, és egyetlen egész számot tartalmazzon, a lehetséges legkisebb összköltséget.

Példák

1. példa

input	output
2 50 100 100 100 100	45000

2. példa

input	output
5 1 1 3 3 2 4 3 2 3 1 4	153

Ez a szövegben szereplő példa.

Megjegyzés

Az összköltség egyik tesztesetben sem haladja meg a $2^{31} - 1$-et egy felosztás esetén sem.

Pontozás

Ha a programod egy tesztesetre kiírja a helyes választ az időkorláton belül, akkor a teljes pontszám jár az adott tesztesetre, különben 0 pontot kapsz.

Idő- és memóriakorlát

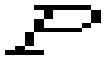
Egy $1,7$ GHz-es Pentium 4 processzorral és 256 MB RAM-mal rendelkező számítógépen az időkorlát $0,1$ másodperc, a memóriakorlát 32 MB.

Buszterminálok

Yong-In városa új buszhálózatot szeretne kiépíteni N buszmegállóval. Minden buszmegálló utcasarkon helyezkedik el. Yong-In egy modern város, ezért a térképe egyenlő méretű, négyzet alakú blokkok hálózata. A buszmegállók közül kettőt kiválasztunk csomópontnak (H_1 és H_2). A csomópontokat egy közvetlen buszjárat köti össze egymással, és a maradék $N-2$ buszmegálló közvetlenül csatlakozik vagy H_1-hez, vagy H_2-höz (de nem mindkettőhöz), de nem csatlakozik közvetlenül más megállókhoz.

Bármely két buszmegálló közötti távolság a lehető legrövidebb útvonal hossza az utcák mentén, azaz ha egy buszmegállót (x,y)-nal jelölünk (ahol x az x-koordináta, y pedig az y-koordináta), akkor az (x_1,y_1) és az (x_2,y_2) buszmegállók közötti távolság $|x_1-x_2|+|y_1-y_2|$. Ha az A és a B buszmegállók ugyanahhoz a H_1 csomóponthoz csatlakoznak, akkor az A-ból B-be vezető útvonal hossza az A-tól H_1-ig és a H_1-től B-ig mért távolságok összege. Ha az A és a B buszmegállók különböző csomópontokhoz csatlakoznak (például A a H_1-hez, B pedig a H_2-höz), akkor az A-ból B-be vezető útvonal hossza az A-tól H_1-ig, a H_1-től H_2-ig és a H_1-től B-ig mért távolságok összege.

Yong-In város városfejlesztési hatósága szeretne megbizonyosodni afelől, hogy minden polgár a város bármely pontját a lehető leggyorsabban el tudja érni. Ezért a várostervezők úgy akarják kiválasztani a két csomópontot, hogy a keletkező buszhálózatban bármely két buszmegálló közötti leghosszabb útvonal hossza a lehető legkisebb legyen.

A két csomópontnak és a hozzájuk kapcsolódó buszmegállóknak a P megválasztása jobb, mint egy másik  választás, ha a leghosszabb buszútvonal hossza P-ben kisebb, mint  -ban. A feladatod, hogy írj programot, amely kiszámítja a legjobb  választásban ennek a leghosszabb útvonalnak a hosszát.

Input

A programodnak a bemenetet a standard inputról kell olvasnia. Az első sor egy  pozitív egész számot tartalmaz ($2 \le N \le 500$), amely a buszmegállók száma. A maradék  sor az egyes buszmegállók x- és y-koordinátáit tartalmazza, amelyek értéke nem haladja meg az 5000-et. Minden buszmegálló más-más helyen található.

Output

A programodnak a kimenetet a standard outputra kell írnia. A kimenet egy olyan sort tartalmazzon, amelyben egyetlenegy pozitív egész szám szerepel, a leghosszabb buszútvonal minimális hossza az

adott bemenet esetén.

Példák

1. példa

input	output
6 1 7 16 6 12 4 4 4 1 1 11 1	20

2. példa

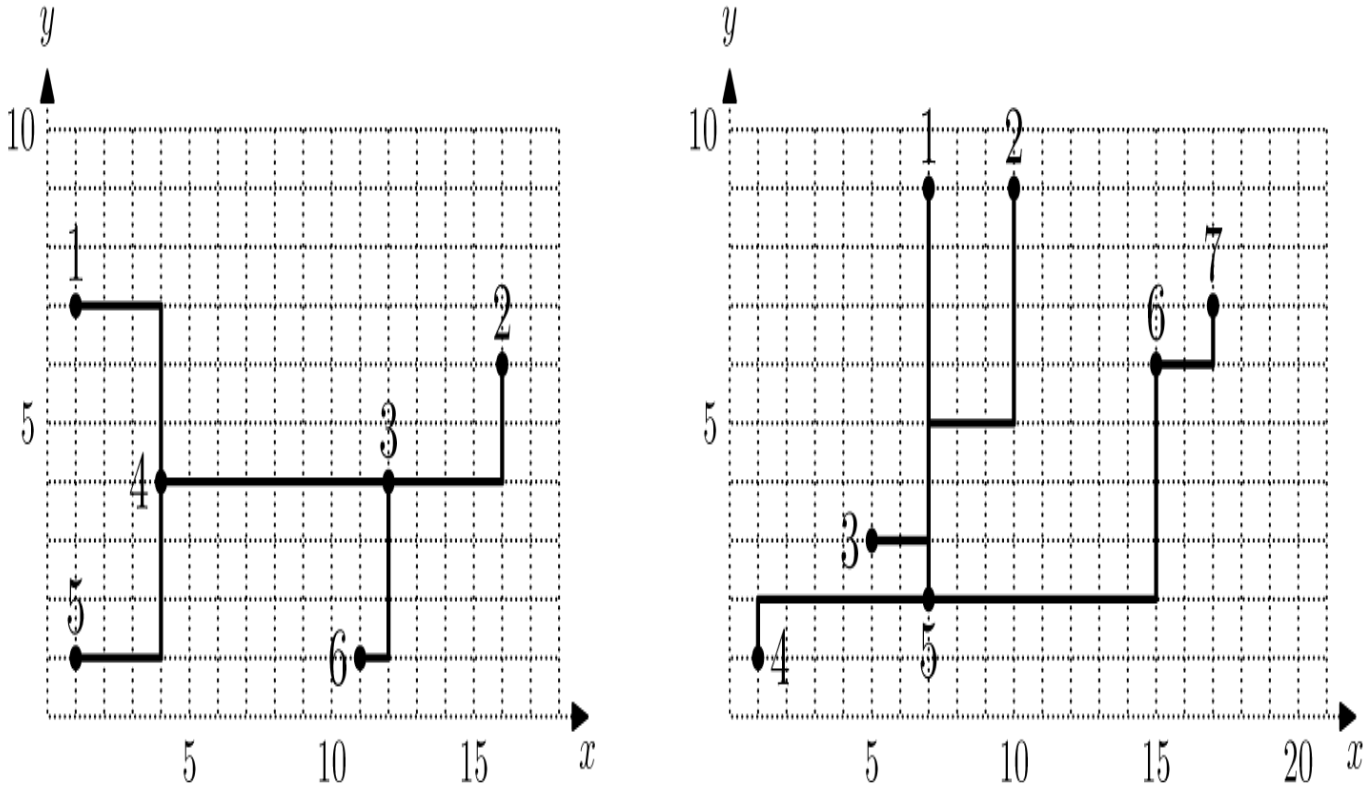
input	output
7 7 9 10 9 5 3 1 1 7 2 15 6 17 7	25

A 16.7. ábrán látható rajzok a fenti bemenetekhez tartozó buszhálózatokat mutatják (a bal oldali az 1. példa, a jobb oldali a 2. példa buszhálózatát). Ha az 1. példában a 3-as és a 4-es buszmegállót választjuk csomópontoknak, akkor a leghosszabb útvonal vagy a 2-es és az 5-ös megállók között, vagy a 2-es és az 1-es megállók között található. A csomópontokat nem tudjuk jobban megválasztani, a válasz pedig 20.

A 2. példában megadott buszhálózat esetén, ha az 5-ös és a 6-os buszmegállót választjuk csomópontoknak, akkor a leghosszabb útvonal a 2-es és a 7-es buszmegállók között lesz. A

csomópontokat nem tudjuk jobban megválasztani, a válasz 25.

16.7. ábra -



Pontozás

Ha a programod egy tesztesetre kiírja a helyes választ az időkorláton belül, akkor a teljes pontszám jár az adott tesztesetre, különben 0 pontot kapsz.

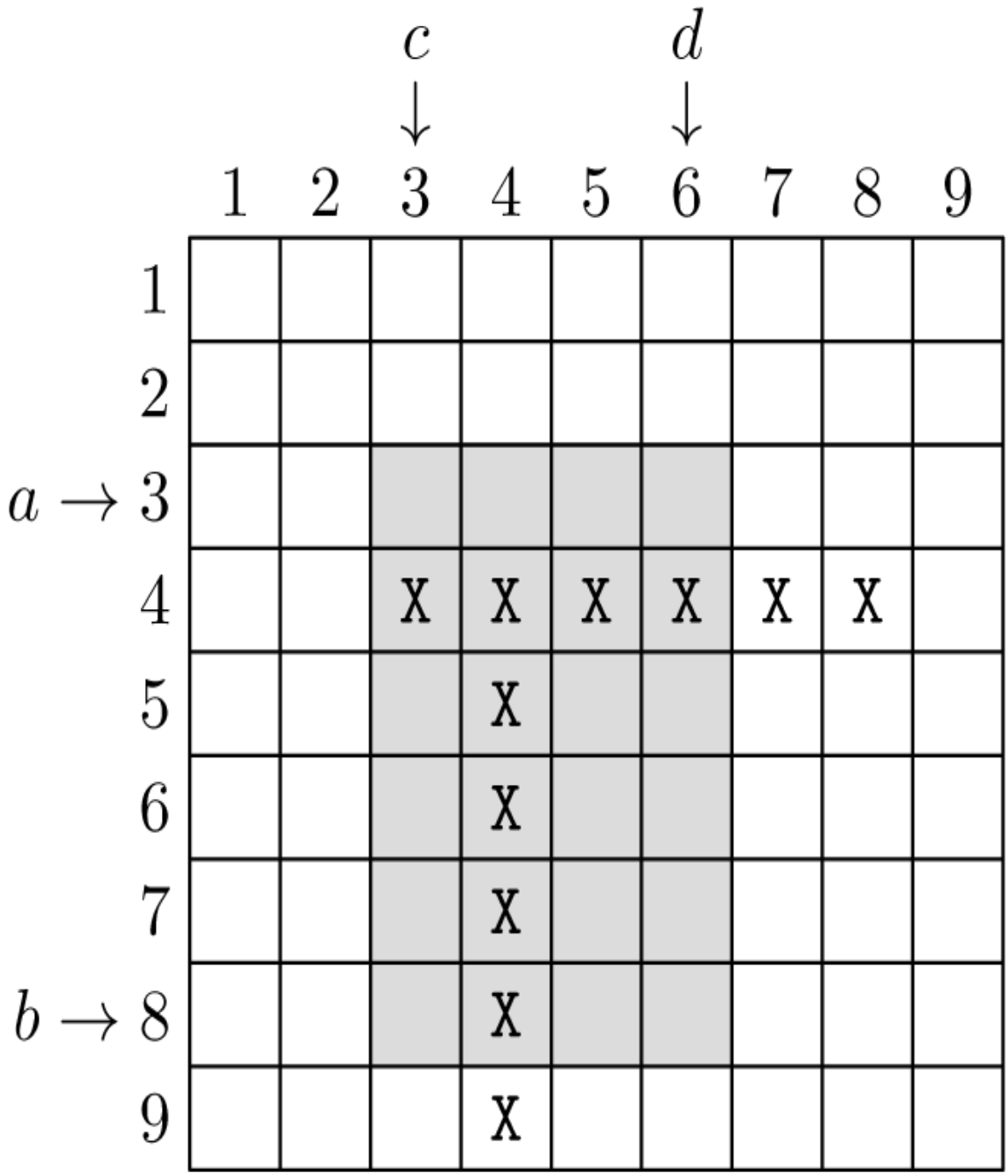
Idő- és memóriakorlát

Egy ^{1,7} GHz-es Pentium 4 processzorral és 256 MB RAM-mal rendelkező számítógépen az időkorlát 4 másodperc, a memóriakorlát 32 MB.

Két rúd

A rúd legalább 2 darab egymás melletti rácscella vízszintes vagy függőleges sorozata. Egy vízszintes és egy függőleges rudat elhelyeztünk egy $N \times N$ -es rácson. A 16.8. ábrán a két rudat X-ekkel jelöltük. A rudak lehet, hogy azonos hosszúságúak, de az is lehet, hogy nem; továbbá lehet közös cellájuk. Ha egy -- az alábbihoz hasonló -- ábrán egy cellát (például a $^{(4,4)}$ -et) úgy is lehet értelmezni, hogy csak az egyik rúdhoz tartozik, és úgy is, hogy mindkettőhöz, akkor úgy értelmezzük, hogy mindkettőhöz tartozik. Ezért a függőleges rúd legfelső cellája a $^{(4,4)}$, nem pedig az $^{(5,4)}$.

16.8. ábra -



Kezdetben nem tudjuk, hol vannak a rudak, a feladatod, hogy írj programot, amely meghatározza a helyzetüket. A vízszintes rudat RÚD1-nek, a függőleges rudat RÚD2-nek nevezzük. Minden rácscellát egy (r,c) sor/oszlop párral jelölünk, a rács bal felső sarkának az $(1,1)$ pontot tekintjük. A rudakat $\langle(r_1,c_1),(r_2,c_2)\rangle$ alakú cellapárokkal adjuk meg. A 16.8. ábrán a $\langle(4,3),(4,8)\rangle$ a RÚD1, a $\langle(4,4),(9,4)\rangle$ pedig a RÚD2.

Ebben a feladatban a bemenet beolvasására, a megoldás meghatározására és a kimenet kiírására könyvtári függvényeket kell használni. A négyzet alakú rács oldalhosszát a `gridsize` könyvtári függvény adja meg, amelyet a programodnak minden egyes teszteset elején meg kell hívnia. A rudak helyzetének meghatározásához csak a rect(*a*,*b*,*c*,*d*) könyvtári függvényt használhatod, amely

megvizsgálja az $[a,b]\times[c,d]$ téglalap alakú területet (az ábrán az árnyékolt részt), ahol $a \le b$ és $c \le d$. (Jól figyeld meg a paraméterek sorrendjét!) Ha bármelyik rúdnak legalább egy rácscellája beleesik a lekérdezett $[a,b]\times[c,d]$ téglalapba, a `rect` 1-et ad vissza, különben pedig 0-t. Így a példában a `rect(3,8,3,6)` 1-et ad vissza. Feladatod, hogy írj egy programot, amely kitalálja a rudak pontos helyzetét, korlátozott számú `rect`-hívást használva.

A kimenetet egy másik könyvtári függvény, a `report(`r_1`,`c_1`,`r_2`,`c_2`,`p_1`,`q_1`,`p_2`,`q_2`)` hívásával kell előállítanod, ahol a RÚD1 az $\langle(r_1,c_1),(r_2,c_2)\rangle$, a RÚD2 pedig a $\langle(p_1,q_1),(p_2,q_2)\rangle$. A `report` meghívása befejezteti a programodat. Ne felejtsd el, hogy a RÚD1 vízszintes, a RÚD2 pedig függőleges, és hogy az (r_1,c_1) a RÚD1 vízszintes rúd bal szélső cellája, míg a (p_1,q_1) a RÚD2 legfelső cellája. Ezért $r_1 = r_2$, $c_1 < c_2$, $p_1 < p_2$ és $q_1 = q_2$. Ha a `report` paraméterei nem felelnek meg ezeknek a követelményeknek, akkor hibaüzeneteket kapsz a standard outputon.

Követelmények

• A bemenethez csak a `gridsize` és a `rect` könyvtári függvények használatával férhetsz hozzá.

• A bemenet maximális sor- és oszlopméretére (N-re) teljesül, hogy $5 \le N \le 10000$.

• A `rect`-hívások száma tesztesetenként legfeljebb 400 lehet. Ha a programod a `rect`-et 400-nál többször hívja meg, a programod befejeződik.

• A programodnak a `rect`-et egynél többször, a `report`-ot pedig pontosan egyszer kell meghívnia.

• Ha a `rect`-hívás érvénytelen (például a lekérdezett terület kilóg a rács területéről), a programod befejeződik.

• A programod nem olvashat és nem írhat egyetlen állományt sem, és nem használhatja a standard be- és kimenetet.

A könyvtár

Adott egy könyvtár a következőképpen:

FreePascal könyvtár (`prectlib.ppu`, `prectlib.o`)

```
function gridsize: LongInt;
function rect(a, b, c, d: LongInt): LongInt;
procedure report(r1, c1, r2, c2, p1, q1, p2, q2: LongInt);
```

Instrukciók: A `rods.pas` lefordításához szúrd be a

```
uses prectlib;
```

utasítást a forráskódba, majd azt az

```
fpc -So -O2 -XS rods.pas
```

paranccsal fordítsd le! A `prodstool.pas` példaprogram bemutatja ennek a FreePascal-könyvtárnak a használatát.

GNU C/C++ könyvtár (`crectlib.h`, `crectlib.o`)

```
int gridsize();
int rect(int a, int b, int c, int d);
void report(int r1, int c1, int r2, int c2,
            int p1, int q1, int p2, int q2);
```

Instrukciók: A `rods.c` vagy `rods.cpp` lefordításához szúrd be az

```
#include "crectlib.h"
```

direktívát a forráskódba, majd azt a

```
gcc -O2 -static rods.c crectlib.o -lm
g++ -O2 -static rods.cpp crectlib.o -lm
```

paranccsal fordítsd le! A `crodstool.c` példaprogram bemutatja ennek a GNU C/C++-könyvtárnak a használatát.

C/C++ az RHIDE környezetben

Ne felejtsd el beállítani az `Option->Linker configuration` menüpontban a`crectlib.o`-t!

Kísérletezés

A könyvtár kipróbálásának céljából létre kell hoznod egy `rods.in` szöveges állományt, amelynek három sort kell tartalmaznia. Az első sor egy N egész számot tartalmaz, a rács méretét. A második sorban a RÚD1 koordinátái (r_1, c_1, r_2, c_2) szerepelnek egy-egy szóközzel elválasztva, ahol (r_1,c_1) a RÚD1 bal szélső cellája. A harmadik sorban a RÚD2 koordinátái (p_1, q_1, p_2, q_2) szerepelnek egy-egy szóközzel elválasztva, ahol (p_1,q_1) a RÚD2 legfelső cellája.

Miután lefuttatod a programodat, amely meghívja a `report`-ot, megkapod a `rods.out` kimeneti állományt. Ez az állomány tartalmazza a `rect`-függvényhívások számát és a rudak végpontjainak koordinátáit, amelyeket a `report`-hívásodban adtál meg. Ha a függvényhívások közben hiba lépett fel, vagy megsértetted a követelményeket, akkor a `rods.out` a megfelelő hibaüzeneteket fogja tartalmazni.

A programod és a könyvtár közötti párbeszédet a `rods.log` állományban rögzítjük. Ez a `rods.log` naplófájl

k : rect(*a*,*b*,*c*,*d*) = *válasz*

formában megmutatja a programod által végrehajtott függvényhívások sorozatát, amely azt jelenti, hogy a k -adik rect(*a*,*b*,*c*,*d*) függvényhívás visszatérési értéke *válasz* volt.

Példa

`rods.in`	`rods.out`
9	20
4 3 4 8	4 3 4 8
4 4 9 4	4 4 9 4

Pontozás

Ha a programod bármelyik követelményt megsérti (például 400-nál többször hívja meg a `rect`-et), vagy a programod kimenete (a rudak helyzete) hibás, a pontszám 0.

Ha a programod kimenete helyes, akkor a pontszámod a `rect`-hívások számától függ az egyes tesztadatok esetén. Minden olyan tesztesetre, ahol a `rect`-hívások száma legfeljebb 100, 5 pontot kapsz. Ha a programod 101 és 200 közötti `rect`-hívást végez, 3 pontot kapsz. Ha a `rect`-hívások

száma 201 és 400 között van, akkor 1 pont a jutalmad.

Idő- és memóriakorlát

Egy $1,7$ GHz-es Pentium 4 processzorral és 256 MB RAM-mal rendelkező számítógépen az időkorlát 1 másodperc, a memóriakorlát 32 MB.

17. fejezet - Nemzetközi Informatikai Diákolimpia, 2003, Kenosha, USA

Tartalom

Csapások fenntartása

John farmer tehenei szabadon akarnak mozogni a farmon lévő N ($1 \leq N \leq 200$) darab legelő között (amelyek 1-től N -ig vannak megszámozva), bár a legelők erdővel vannak elválasztva egymástól. A tehenek úgy szeretnének csapásokat fenntartani bizonyos legelők között, hogy bármelyik legelőről eljuthassanak bármelyik másikra a fenntartott csapások segítségével. A tehenek mindkét irányban mozoghatnak a fenntartott csapásokon.

A tehenek nem csinálnak új csapásokat, hanem a vadállatok által kitaposott ösvényeket használják. Minden héten eldönthetik, hogy az általuk ismert vadcsapások egy részét, vagy mindegyikét fenntartják.

Mivel a tehenek kíváncsiak, minden hét elején felfedeznek egy-egy új vadcsapást. Ezután el kell dönteniük, hogy az adott héten mely csapásokat tartják fenn, hogy bármelyik legelőről bármelyik másikra eljuthassanak. A tehenek csak azokat a csapásokat használhatják, amelyeket éppen fenntartanak.

A tehenek mindig minimalizálni szeretnék a fenntartandó csapások összhosszát. Az ismert vadcsapások bármely részhalmazát kiválaszthatják fenntartásra, függetlenül attól, hogy az előző héten mely csapásokat használták.

A vadcsapások sohasem egyenesek (a fenntartottak sem). Az ugyanazon legelőket összekötő két csapás különböző hosszúságú lehet. Bár a csapások keresztezhetik egymást, a tehenek nem váltanak csapást, hacsak nem épp egy legelőn vannak.

A tehenek minden hét elején megadják az éppen felfedezett vadcsapást. A programodnak ezután ki kell írnia az adott héten fenntartandó csapások minimális összhosszát, amelyeket úgy kell kiválasztani, hogy a tehenek bármelyik legelőről bármelyik másikra eljuthassanak, ha létezik a csapásoknak ilyen halmaza.

Input

A bemenetet a standard inputról kell olvasni.

• A bemenet első sora két, szóközzel elválasztott egész számot tartalmaz, N -et és W -t. W a program által lefedett hetek száma ($1 \leq W \leq 6000$).

• Minden hétre egy sort kell beolvasni, amely az adott héten felfedezett vadcsapást tartalmazza. Ez a sor három, szóközzel elválasztott egész számot tartalmaz: az új csapás végpontjait (legelőszámokkal megadva) és hosszát, amely egy 1 és 10000 közé eső egész szám. Egyik vadcsapásnak sem ugyanaz a legelő a két végpontja.

Output

A kimenetet a standard outputra kell írni. Amint a programod értesül az újonnan felfedezett vadcsapásról, ki kell írnia egy sort, amely az adott héten fenntartandó csapások minimális összhosszát tartalmazza, amelyeket úgy kell kiválasztani, hogy a tehenek bármelyik legelőről bármelyik másikra eljuthassanak. Ha a csapásoknak nem létezik olyan halmaza, amelynek segítségével a tehenek bármelyik legelőről bármelyik másikra eljuthatnának, -1-et kell kiírni.

A programodnak ki kell lépnie, miután kiírta az utolsó hétre vonatkozó választ.

Egy futási példa

Bemenet	*Kimenet*	*Magyarázat*
4 6		
1 2 10		
	-1	A 4-es mezőt egy csapás sem köti össze a többivel.
1 3 8		
	-1	A 4-es mezőt egy csapás sem köti össze a többivel.
3 2 3		
	-1	A 4-es mezőt egy csapás sem köti össze a többivel.
1 4 3		
	14	Tartsuk fenn az 1 4 3, az 1 3 8 és a 3 2 3 csapásokat.
1 3 6		
	12	Tartsuk fenn az 1 4 3, az 1 3 6 és a 3 2 3 csapásokat.
2 1 2		
	8	Tartsuk fenn az 1 4 3, az 2 1 2 és a 3 2

		3 csapásokat.
a program kilép		

Idő- és memóriakorlát

Egy $2,2$ GHz-es Pentium 4 processzorral és 256 MB RAM-mal rendelkező számítógépen az alábbi korlátok figyelembevételével kell a programot elkészíteni:

Futási idő	1 másodperc
Memória	64 MB

Pontozás

Teljes pontszám jár minden olyan tesztesetre, amelyre a programod helyes kimenetet állít elő. Részpontszám egyik tesztesetre sem adható.

Kódok összehasonlítása

A Racine Business Networks (RBN) beperelte a Heuristic Algorithm Languages (HAL) vállalatot, azzal vádolva őt, hogy forráskódot vett át az RBN UNIX™-ból, és felhasználta azt a HALnix nyílt forrású operációs rendszerben.

Az RBN és a HAL is egy olyan programozási nyelvet használnak, amelyben soronként egy utasítást lehet írni, `STOREA = STOREB + STOREC` formában, ahol `STOREA`, `STOREB` és `STOREC` változónevek. Pontosabban: az első változónév az első oszlopban kezdődik, ezt követi egy szóköz, egy egyenlőségjel, egy szóköz, a második változónév, egy szóköz, egy összeadásjel, egy szóköz és a harmadik változónév. Egy sorban ugyanaz a változónév többször is szerepelhet. A változónevek legalább 1, legfeljebb 8 darab angol nagybetűből (`A`, , `Z`) állnak.

Az RBN azt állítja, hogy a HAL közvetlenül az RBN forráskódjából lemásolt egymás utáni sorokat, és csak kisebb módosításokat hajtott végre:

• Az RBN szerint a HAL megváltoztatott néhány változónevet, hogy leplezze bűnét. Ez azt jelenti, hogy a HAL átvett egy kódrészletet az RBN programjából, és benne minden változó nevét kicserélte egy új névre, ami akár ugyanaz is lehetett, mint az eredeti. Természetesen nincs két olyan változó, amely ugyanazt az új nevet kapta volna.

• Az RBN azt is állítja, hogy a HAL megváltoztathatta néhány sorban a jobb oldal sorrendjét: a `STOREA = STOREB + STOREC` sort a `STOREA = STOREC + STOREB` sorra cserélhette.

• Az RBN állítása szerint a HAL nem változtatta meg az RBN forráskódjában a sorok sorrendjét.

Adottak az RBN-től és a HAL-tól származó programok forráskódjai. A feladatod, hogy megtaláld a HAL programjának azt a leghosszabb összefüggő részét, amely származhat az RBN programjának egy összefüggő részéből, felhasználva a fenti módosításokat. Vedd figyelembe, hogy a két programból származó kódrészletnek nem kell ugyanannál a sorszámnál kezdődnie mindkét állományban.

Input

A bemenetet a `code.in` állományból kell olvasni.

• A bemenet első sora két, szóközzel elválasztott egész számot, R-et és H-t tartalmazza (

$1 \le R \le 1000$, $1 \le H \le 1000$). R az RBN programjához tartozó forráskód sorainak a száma, H pedig a HAL programjához tartozó forráskód sorainak a száma.

• A következő N sor az RBN programját tartalmazza.

• A következő H sor a HAL programját tartalmazza.

Output

A kimenetet a `code.out` állományba kell írni. Az állománynak egyetlen sort kell tartalmaznia, amelyben egyetlen egész szám szerepel, annak a leghosszabb összefüggő kódrészletnek a hossza (sorokban), amelyet a HAL átmásolhatott az RBN-től, és átalakított.

Példa input

4 3
RA = RB + RC
RC = D + RE
RF = RF + RJ
RE = RF + RF
HD = HE + HF
HM = HN + D
HN = HA + HB

Példa output

2

Az RBN programjának első és második sora megegyezik a HAL programjának második és harmadik sorával, ha az RBN programjában elvégezzük a következő változónév-helyettesítéseket: `RA` → `HM`, `RB` → `D`, `RC` → `HN`, `D` → `HA` és `RE` → `HB`. Három vagy több sorból álló egyezés nincs.

Idő- és memóriakorlát

Egy $2,2$ GHz-es Pentium 4 processzorral és 256 MB RAM-mal rendelkező számítógépen az alábbi korlátok figyelembevételével kell a programot elkészíteni:

Futási idő	2 másodperc
Memória	64 MB

Pontozás

Teljes pontszám jár minden olyan tesztesetre, amelyre a programod helyes kimeneti állományt állít elő. Részpontszám egyik tesztesetre sem adható.

Csökkenő

Adott egy kétműveletes gép (Two-Operation Machine, röviden TOM) kilenc regiszterrel, amelyek 1-től 9-ig vannak megszámozva. Mindegyik regiszter egy-egy nemnegatív egész számot tárol 0-tól 1000 -ig. A gépnek két művelete van:

S i j	Az i -edik regiszter értékénél eggyel nagyobb érték tárolása a j -edik regiszterben. Az i és j egyenlő is lehet.
P i	Az i -edik regiszter értékének kiírása.

Egy TOM-program a regiszterek kezdőértékeinek halmazából és műveletek egy sorozatából áll. Ha adott egy N egész szám ($0 \leq N \leq 255$), készíts egy TOM-programot, amely kiírja az N, $N-1$, $N-2$, , 0 egész számok csökkenő sorozatát. Az egymást követő S -műveletek maximális számának a lehető legkisebbnek kell lennie.

Példa egy TOM-programra és futására $N = 2$ esetén:

Művelet	*Új regiszterértékek*									*Kiírt*
	1	2	3	4	5	6	7	8	9	*érték*
kezdőértékek	0	2	0	0	0	0	0	0	0	
P 2	0	2	0	0	0	0	0	0	0	2
S 1 3	0	2	1	0	0	0	0	0	0	
P 3	0	2	1	0	0	0	0	0	0	1
P 1	0	2	1	0	0	0	0	0	0	0

Az input esetek 1-től 16-ig vannak megszámozva, és a verseny szerverén keresztül érhetők el.

Input

• A bemeneti állomány első sora tartalmazza K -t, egy egész számot, amely az input eset sorszáma.

• A bemenet második sora az N -et tartalmazza.

Output

A kimenet első sorának a „`FILE reverse K`" sztringet kell tartalmaznia, ahol K az eset sorszáma.

A kimenet második sorának kilenc darab, szóközzel elválasztott értéket kell tartalmaznia, amelyek sorrendben a regiszterek kívánt kezdőértékeit jelentik (az első regiszterét, majd a második regiszterét stb.).

A kimenet hátralévő részének a végrehajtandó műveletek rendezett listáját kell tartalmaznia, soronként egy műveletet. Így a harmadik sor tartalmazza az elsőként végrehajtandó műveletet, és így tovább. Az állomány utolsó sorában egy olyan műveletnek kell szerepelnie, amely a 0 kiírására

szolgál. Minden egyes sorban egy érvényes műveletnek kell állnia. Az utasításokat a példa outputban megadott módon kell formázni.

Példa input

1
2

Példa output

1. példa (részpontos)

FILE reverse 1
0 2 0 0 0 0 0 0 0
P 2
S 1 3
P 3
P 1

2. példa (teljes pontos)

FILE reverse 1
0 2 1 0 0 0 0 0 0
P 2
P 3
P 1

Idő- és memóriakorlát

Nincs.

Pontozás

Minden teszteset pontozása az elkészített TOM-program helyességén és optimalitásán alapul.

Helyesség: a pontok 20%-a

Egy TOM-program helyes, ha nem hajt végre 131-nél több egymást követő S-műveletet, és ha az általa kiírt értéksorozat helyes (pontosan $N+1$ egész számot tartalmaz, N-nel kezdődik, és 0-val fejeződik be). Ha valamelyik S-művelet regisztertúlcsordulást okoz, a TOM-programot hibásnak tekintjük.

Optimalitás: a pontok 80%-a

Egy helyes TOM-program optimalitását a programban végrehajtott egymást követő S-műveletek maximális számával mérjük, amelynek a lehető legkisebbnek kell lennie. A pontozás során a te TOM-programod és a legjobb ismert TOM-program közötti különbséget vesszük figyelembe.

Melyik tehén?

A John farmer csordájában lévő N darab ($1 \leq N \leq 50$) tehén nagyon hasonlít egymásra. A tehenek 1-től N-ig vannak megszámozva. Amikor John farmer lefektet egy tehenet a rekeszében, tudnia kell, melyik tehenet fekteti le, hogy a megfelelő rekeszbe helyezhesse.

A teheneket P tulajdonság alapján lehet megkülönböztetni ($1 \leq P \leq 8$), amelyek mindegyike három lehetséges értékkel rendelkezik. Egy tehén fülének a vége például lehet sárga, zöld vagy piros. Az egyszerűség kedvéért minden tulajdonság értékeit az `X`, `Y` és `Z` betűkkel adjuk meg. John farmer tehenei közül bármelyik kettő legalább egy tulajdonságban eltér.

Írj programot, amely a John farmer csordájában lévő tehenek adott tulajdonságai alapján segít John farmernek eldönteni, hogy éppen melyik tehenet fekteti le. A programod John farmernek legfeljebb 100 kérdést tehet fel a következő formában: A tehén egy T tulajdonságának értéke egy S halmazban van? Próbáld meg a lehető legkevesebb kérdéssel eldönteni, melyik tehénről van szó.

Input

A bemenetet a `guess.in` állományból kell olvasni.

• A bemeneti állomány első sora két, szóközzel elválasztott egész számot tartalmaz (N-et és P-t).

• A következő N sor mindegyike egy tehén tulajdonságait írja le P darab, szóközzel elválasztott betűvel. Minden sor első betűje az 1-es tulajdonság értéke, és így tovább. A bemeneti állomány második sora az első tehenet írja le, a harmadik sor a másodikat stb.

Példa input

4 2
X Z
X Y
Y X
Y Y

Működés

A kérdés/válasz szakasz interaktívan, a standard inputon és a standard outputon keresztül zajlik.

A programod az éppen lefektetni kívánt tehénről feltesz egy kérdést úgy, hogy kiír a standard outputra egy sort, amelyben egy `Q` betű, egy szóköz, a tulajdonság száma, egy szóköz, és egy vagy több érték szóközzel elválasztott halmaza szerepel. A `Q 1 Z Y` például azt jelenti, hogy „Az 1-es tulajdonság értéke `Z` vagy `Y`-e a lefektetni kívánt tehén esetén?" A tulajdonságnak egy 1 és P közé eső egész számnak kell lennie. Az értékek csak `X`, `Y` vagy `Z` lehetnek, és egyik érték sem szerepelhet egynél többször egy kérdésben.

Minden egyes kérdésfeltevés után olvass be egy sort, amely egyetlenegy egész számot tartalmaz. Az 1-es szám azt jelenti, hogy a megadott tulajdonság értéke a lefektetni kívánt tehén esetén a megadott

értékhalmazban van, a 0 pedig azt jelenti, hogy nem.

A program kimenetének utolsó sorában egy `C` betűnek kell lennie, amelyet egy szóköz és egy egész szám követ, amely megadja, hogy melyik az a tehén, amelyet a programod szerint John farmer éppen lefektet.

Egy futási példa

A fenti példa inputot figyelembe véve egy lehetséges futási példa a következő:

Bemenet	*Kimenet*	*Magyarázat*
	`Q 1 X Z`	
`0`		lehet a 3-as vagy a 4-es tehén
	`Q 2 Y`	
`1`		ez csak a 4-es tehén lehet
	`C 4`	
a program kilép		

Idő- és memóriakorlát

Egy $2,2$ GHz-es Pentium 4 processzorral és 256 MB RAM-mal rendelkező számítógépen az alábbi korlátok figyelembevételével kell a programot elkészíteni:

Futási idő	1 másodperc
Memória	64 MB

Pontozás

Helyesség: a pontok 30%-a

A program helyességére csak akkor kapsz teljes pontszámot, ha a megadott tehén az egyetlen olyan tehén, amely a kapott válaszoknak megfelel. Ha a programod 100-nál több kérdést tesz fel egy tesztesetnél, az adott tesztesetre nem kapsz pontot.

Kérdésszám: a pontok 70%-a

A fennmaradó pontokat a tehén helyes meghatározásához szükséges kérdések száma alapján fogod megkapni. A teszteseteket úgy terveztük meg, hogy a legrosszabb eset kérdésszámának a minimalizálását jutalmazzuk. Közel optimális kérdésszám esetén részpontszám jár.

Bámulatos robotok

Büszke tulajdonosa vagy két robotnak, amelyek két különböző, téglalap alakú labirintusban helyezkednek el. Egy labirintusban az $(1,1)$ mező a bal felső sarokban lévő mező, amely az északnyugati sarok. Az i-edik labirintusban ($i=1,2$) G_i darab őr ($0 \le G_i \le 10$) próbálja meg elkapni a robotokat úgy, hogy oda-vissza járőröznek egy egyenes útvonal mentén. A feladatod egy

olyan parancssorozat meghatározása, amelynek segítségével a robotok kijutnak a labirintusokból anélkül, hogy bármelyiküket is elkapná egy őr.

Minden perc elején kiadod ugyanazt a parancsot mindkét robotnak. Minden parancs egy irány (észak, dél, kelet vagy nyugat). Egy robot egy mezőnyit mozdul a parancsban megadott irányba, hacsak nem ütközik falba, ez esetben mozdulatlan marad az adott percben. Egy robot úgy hagyja el a labirintust, hogy kisétál belőle. Miután elhagyta a labirintusát, figyelmen kívül hagyja a parancsokat.

Az őrök egy mezőnyit mozdulnak minden perc elején, ugyanakkor, amikor a robotok. Az egyes őrök adott mezőről, adott irányba nézve indulnak, és percenként egy mezőnyit lépnek előre, amíg eggyel kevesebbet nem lépnek, mint amennyi az őrjáratukat képező útvonalban szereplő mezők száma. Ekkor azonnal megfordulnak, és az ellenkező irányba sétálnak, vissza a kiinduló mezőjük felé, ahol újra megfordulnak és addig ismétlik az őrjáratukat, amíg mindkét robot el nem hagyta a labirintusát.

Az őröknek őrjáratuk során nem kell falakon átmenniük, vagy elhagyniuk a labirintust. Bár az őrök őrjáratai átfedhetik egymást, sosem fognak ütközni: sosem fogják ugyanazt a mezőt elfoglalni egy adott perc végén, és nem fognak mezőt cserélni egymással egy perc alatt. Egy labirintusban az őrök nem indulnak ugyanarról a mezőről, mint az adott labirintusban lévő robot.

Egy őr akkor kap el egy robotot, ha az őr ugyanazt a mezőt foglalja el egy adott perc végén, mint a robot, vagy ha az őr és a robot egy perc alatt mezőt cserélnek egymással.

Ha adott két labirintus (egyik sem nagyobb, mint 20×20), a két robot kiinduló mezőivel, és az egyes őrök őrjáratainak útvonalaival a két labirintusban, határozd meg azt a parancssorozatot, amelynek hatására elhagyják a labirintusaikat anélkül, hogy az őrök elkapnák őket. Minimalizáld azt az időt, amelyik a lassabb robotnak szükséges a labirintusa elhagyásához. Ha a robotok különböző időpontban hagyják el labirintusaikat, az az időpont, amikor a gyorsabb robot kilépett, nem érdekes.

Input

A bemenetet a `robots.in` állományból kell olvasni. Az első néhány sor az első labirintust és szereplőit adja meg. Az ezután következő sorok a második labirintust és szereplőit írják le.

• A bemenet első sora két, vesszővel elválasztott egész számot (R_1-et és C_1-et) tartalmaz, az első labirintus sorainak és oszlopainak számát.

• A következő R_1 sor mindegyike C_1 darab karaktert tartalmaz, amelyek a labirintus szerkezetét adják meg. A robot kiinduló mezője `X`-szel van jelölve. A `.` nyílt teret, a `#` falat jelöl. Minden labirintus pontosan egy robotot tartalmaz.

• A labirintus szerkezetét egy olyan sor követi, amely egyetlen egész számot tartalmaz (G_1-et), az első labirintusban lévő őrök számát ($0 \leq G_1 \leq 10$).

• A következő G_1 sor mindegyike egy-egy őr útvonalát írja le három egész számmal és egy karakterrel, amelyeket egy-egy szóköz választ el egymástól. Az első két egész szám az őr kiinduló mezőjének sor- és oszlopszámát adja meg. A harmadik egész szám az őr útvonalán szereplő mezők száma (2, 3 vagy 4). A karakter az őr mozgásának kezdeti irányát adja meg: `N`, `S`, `E` vagy `W` (észak, dél, kelet vagy nyugat).

A második labirintus leírása az elsőét követi, ugyanabban a formátumban, de potenciálisan különböző értékekkel.

Output

A kimenetet a `robots.out` állományba kell írni. A kimenet első sorának egyetlenegy K egész számot kell tartalmaznia ($K \leq 10000$), azon parancsok számát, amelyekkel mindkét robot kijut a labirintusából anélkül, hogy elkapnák őket. Ha létezik ilyen parancssorozat, a legrövidebb sorozatban nem lesz 10000-nél több parancs. A következő K sor maga a parancssorozat, mindegyik egy karaktert tartalmazzon az $\{N,S,E,W\}$ halmazból. Ha nem létezik ilyen sorozat, akkor egyetlen sort kell kiírni, amely a -1 számot tartalmazza.

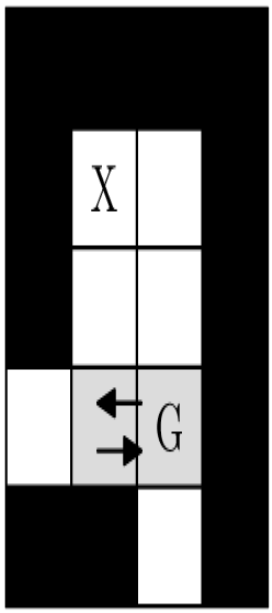
Mindkét robotnak el kell hagynia a labirintusát a parancssorozat végére. Az utolsó parancs hatására legalább az egyik robotnak ki kell lépnie a labirintusából. Ha több parancssorozat is létezik, amelynek hatására a robotok minimális idő alatt elhagyják labirintusaikat, bármelyiket elfogadjuk.

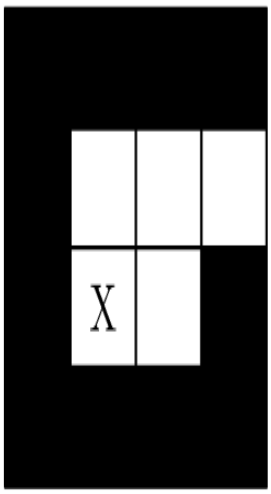
Példa input

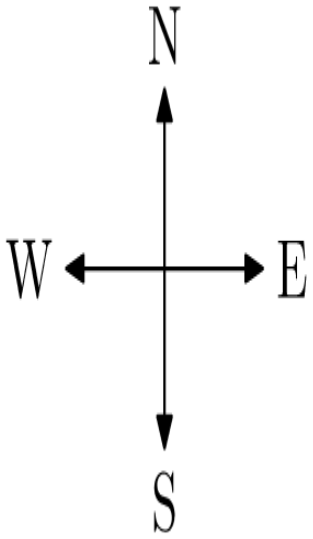
5 4
####
#X.#
#..#
...#
##.#
1
4 3 2 W
4 4
####
#...
#X.#
####
0

A 17.1. ábra a példa inputban leírt labirintusokat ábrázolja.

17.1. ábra -







Példa output

8
E
N
E
S
S
S
E
S

Idő- és memóriakorlát

Egy ^{2,2} GHz-es Pentium 4 processzorral és 256 MB RAM-mal rendelkező számítógépen az alábbi korlátok figyelembevételével kell a programot elkészíteni:

Futási idő	2 másodperc

Memória	64 MB

Pontozás

Nem jár részpontszám azokra a tesztesetekre, amelyekre nem létezik parancssorozat. Más tesztesetekre az alább ismertetettek szerint jár részpontszám.

Helyesség: a pontok 20%-a

Egy tesztesetre a kimeneti állomány akkor számít helyesnek, ha formailag helyes, nem tartalmaz 10000-nél több parancsot, és a parancssorozat határása a robotok kijutnak a labirintusokból úgy, hogy az utolsó parancs végrehajtásakor legalább egy robot elhagyja a labirintusát.

Minimalitás: a pontok 80%-a

Egy tesztesetre a kimeneti állomány minimális, ha helyes, és nincs rövidebb helyes parancssorozat. Az a program, amelynek a parancssorozata nem minimális, nem kap pontszámot a minimalitásra.

A látható határvonal

Don farmer nézi a kerítést, amely az N méterszer N méteres, négyzet alakú, sík legelőjét keríti körül ($2 \leq N \leq 500000$). A kerítés egyik sarka az origóban van, a $(0,0)$ pontban, az átellenes sarka pedig az (N,N) pontban. Don farmer kerítésének oldalai párhuzamosak az X és az Y tengelyekkel.

Kerítésoszlopok a kerítés mind a négy sarkában és az oldalai mentén minden egyes méternél előfordulnak, ez összesen $4N$ kerítésoszlopot jelent. A kerítésoszlopok függőlegesek, és úgy tekintjük, hogy nincs sugaruk. Don farmer tudni szeretné, hogy hány kerítésoszlopát látja a kerítésén belül egy adott helyen állva.

Don farmer legelője R óriási sziklát tartalmaz ($1 \leq R \leq 30000$), amelyek akadályozzák a rálátást bizonyos kerítésoszlopokra, mivel nem elég magas, hogy bármelyik fölött is átlásson. Az egyes sziklák alaprajza egy-egy nem nulla területű konvex sokszög, amelynek csúcsai egész koordinátákon helyezkednek el. A sziklák teljesen függőlegesek. A sziklák nem fedik át egymást, nem érintenek más sziklákat, és nem érintik sem Don farmert, sem a kerítést. Don farmer nem érinti a kerítést, nem áll sziklán belül, sem sziklán.

Ha adott Don farmer kerítésének a mérete, a kerítésen belül lévő sziklák helyzete és alakja, és az a hely, ahol Don farmer áll, számítsd ki a Don farmer által látott kerítésoszlopok számát.

Input

A bemenetet a `boundary.in` állományból kell olvasni.

• A bemenet első sora két, szóközzel elválasztott egész számot tartalmaz, N-et és R-et.

• A bemenet következő sora két, szóközzel elválasztott egész számot tartalmaz, amelyek Don farmer helyzetének X és Y koordinátáit adják meg a kerítésen belül.

• A bemeneti állomány maradék része az R sziklát írja le:

- Az i-edik szikla leírása egy p_i egész számot tartalmazó sorral kezdődik ($3 \leq p_i \leq 20$), amely a szikla alaprajzán lévő csúcspontok száma.

- A következő p_i sor mindegyike egy-egy szóközzel elválasztott egészszám-párt tartalmaz, amelyek az egyes csúcspontok X és Y koordinátái. A sziklák alaprajzainak csúcspontjai különbözőek, és az óramutató járásával ellentétes irányban vannak megadva.

Output

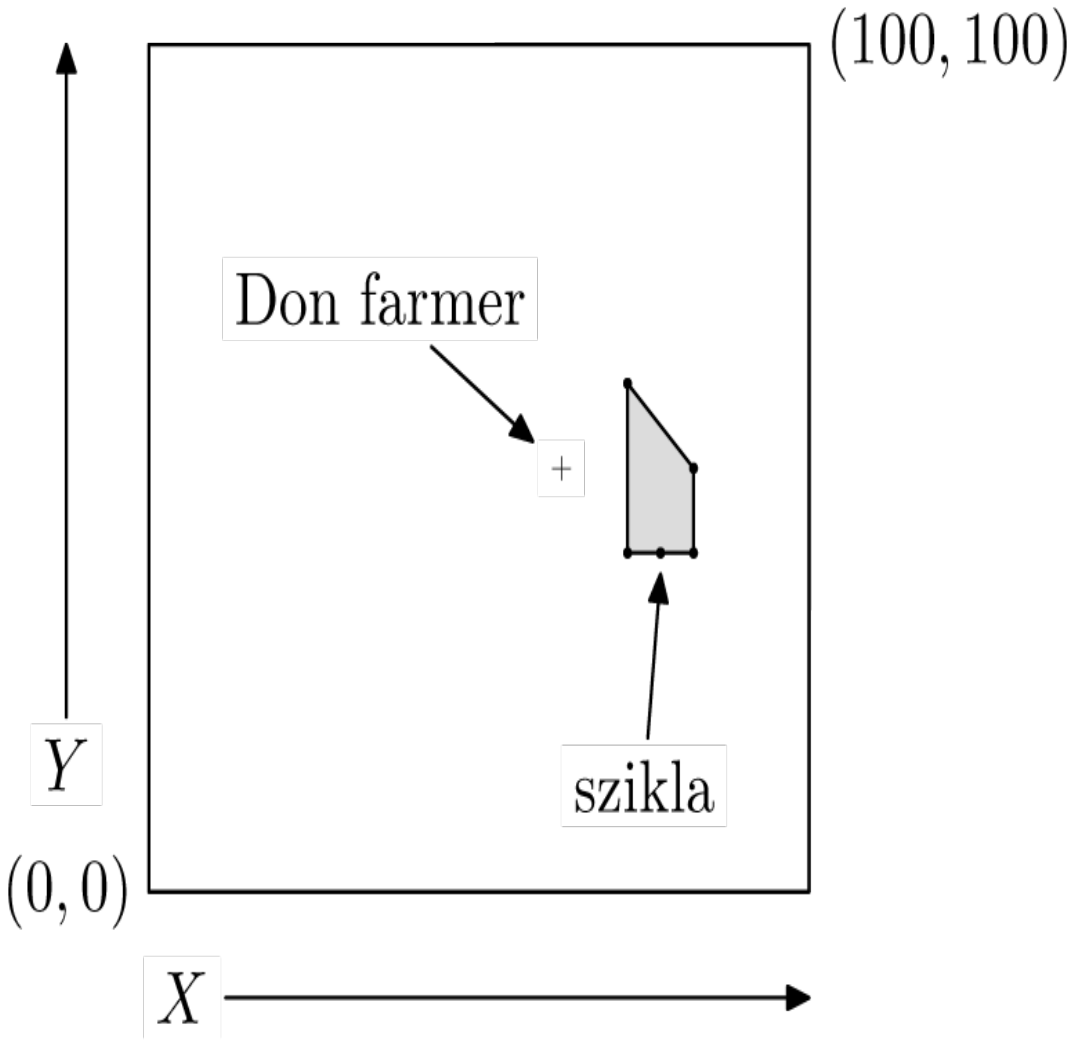
A kimenetet a `boundary.out` állományba kell írni. A kimeneti állománynak egyetlen sort kell tartalmaznia, amelyben egyetlen egész szám szerepel, a Don farmer által látható kerítésoszlopok száma.

Példa input

100 1
60 50
5
70 40
75 40
80 40
80 50
70 60

Vegyük észre, hogy a szikla alaprajzának van három, egy egyenesbe eső csúcspontja: $(70,40)$, $(75,40)$ és $(80,40)$, ahogy az a 17.2. ábrán is látható.

17.2. ábra -



Példa output

319

Idő- és memóriakorlát

Egy ^{2,2} GHz-es Pentium 4 processzorral és 256 MB RAM-mal rendelkező számítógépen az alábbi korlátok figyelembevételével kell a programot elkészíteni:

Futási idő	2 másodperc
Memória	64 MB

Pontozás

Teljes pontszám jár minden olyan tesztesetre, amelyre a programod helyes kimeneti állományt állít

elő. Részpontszám egyik tesztesetre sem adható.

18. fejezet - ACM közép-európai döntő, 2002, Varsó, Lengyelország

Tartalom

Család

Egy szörnycsalád tagjainak rokonsági fokát szeretnénk meghatározni. Mindegyik szörnynek ugyanannyi génje van, de maguk a gének szörnyről szörnyre különbözhetnek. Szeretnénk tudni, hogy két adott szörnynek hány közös génje van. Ez azonban lehetetlen, mivel a gének száma túl nagy. Szerencsére ismerjük a családfát (ami valójában nem is fa, de hát nem hibáztathatjuk őket, hiszen szörnyek, nem igaz?), és tudjuk, hogy a gének hogyan öröklődnek, ezért egész jól meg tudjuk becsülni a közös gének számát.

Az öröklési szabály nagyon egyszerű: ha a C szörny az A és B szörny gyermeke, akkor C minden génje azonos vagy A, vagy B megfelelő génjével, mindkettő 50% valószínűséggel. Minden szörny minden génje függetlenül öröklődik.

Az X és Y szörnyek rokonsági fokát definiáljuk a közös gének várható számával. Vegyünk például egy családot, amely az A és B két teljesen független szörnyből (azaz nincsenek közös génjeik), valamint két gyermekükből, C-ből és D-ből áll. Mennyi C és D rokonsági foka? C minden génje vagy A-tól, vagy B-től származik, mindkettő 50% valószínűséggel. Ugyanez érvényes D-re. Ezáltal annak a valószínűsége, hogy C egy adott génje megegyezik D megfelelő génjével, 50%. Ebből következik, hogy C és D rokonsági foka (azaz a közös génjeik várható száma) a gének számának 50%-a. Vegyük észre, hogy a válasz más lenne, ha A és B rokonságban lennének. Ha A-nak és B-nek vannak közös génjeik, akkor azokat C és D is örökli.

Feladat

Írj programot, amely

• a standard inputról beolvassa egy család leírását, valamint családtagpárok egy listáját,

• kiszámítja a rokonsági fokot (százalékban) a listában szereplő minden párra,

• kiírja az eredményt a standard outputra.

Input

A bemenet első sora két egész számot (n és k) tartalmaz egy szóközzel elválasztva. Az n ($2 \le n \le 300$) jelöli a család tagjainak a számát. A családtagok 1-től n-ig tetszőlegesen vannak megszámozva. A k ($0 \le k \le n-2$) azoknak a szörnyeknek a száma, amelyeknek vannak szüleik (az összes többi szörnyet az istenek teremtették, és nem állnak egymással rokonságban).

A következő k sor mindegyike három különböző egész számot (a, b és c) tartalmaz egy-egy szóközzel elválasztva. Az a, b, c hármas azt jelenti, hogy az a szörny a b és c szörnyek gyermeke.

A bemenet következő sora egy m egész számot tartalmaz ($1 \leq m \leq n^2$), amely a listában szereplő szörnypárok száma. A következő m sor mindegyike két egész számból áll egy szóközzel elválasztva, amelyek két szörny sorszámait jelentik.

Feltehetjük, hogy egyik szörny sem a saját őse. További feltételezéseket nem tehetünk az input adatokra vonatkozóan, nem tehetjük fel például, hogy léteznek nemek.

Output

A kimenet m sorból álljon. Az i-edik sor az input lista i-edik párjához tartozzon, és egyetlen számot kell tartalmaznia, amelyet egy százalékjel követ. A szám az i-edik párban szereplő szörnyek pontos rokonsági foka (százalékban). Értéktelen 0 számjegyek nem állhatnak a kimeneten (a tizedespont előtt azonban legalább egy számjegynek kell állnia, ezért például a 0.1 számban a 0 értékesnek számít, és nem írhatjuk ki $.1$ formában). A kimenet pontos formájához lásd a példa outputot.

Példa input

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

Példa output

```
0%
50%
81.25%
100%
```

Intervallumok

Adott n db zárt, egész intervallum ($[a_i, b_i]$) és n db egész szám ($c_1, \ldots, c_n$).

Feladat

Írj programot, amely

• a standard inputról beolvassa az intervallumok számát és végpontjait, valamint a $c_1, \ldots, c_n$ egészeket,

• kiszámítja a méretét annak a legkisebb, egészekből álló Z halmaznak, amelynek minden $i = 1, 2, \ldots, n$ esetén legalább c_i közös eleme van az $[a_i, b_i]$ intervallummal,

• kiírja az eredményt a standard outputra.

Input

A bemenet első sora az n egész számot tartalmazza ($1 \le n \le 50000$), amely az intervallumok száma. A következő n sor írja le az intervallumokat. Az input $i+1$-edik sora három egész számból (a_i, b_i és c_i) áll egy-egy szóközzel elválasztva, amelyekre $0 \le a_i \le b_i \le 50000$ és $1 \le c_i \le b_i - a_i + 1$.

Output

A kimenetnek pontosan egy egész számot kell tartalmaznia, amely annak a legkisebb, egészekből álló Z halmaznak a mérete, amelynek legalább c_i közös eleme van az $[a_i, b_i]$ intervallummal minden $i = 1, 2, \ldots, n$-re.

Példa input

```
5
3 7 3
8 10 3
6 8 1
1 3 1
10 11 1
```

Példa output

```
6
```

Egyirányú forgalom

Egy városban van n db kereszteződés, amelyeket két- és egyirányú utak kötnek össze. Ez egy nagyon modern város, ezért számos út alagutakon és viaduktokon fut keresztül. Természetesen bármely két kereszteződés között mindkét irányban utazhatunk, azaz eljuthatunk az a kereszteződésből a b kereszteződésbe, és b-ből a-ba is a közlekedési szabályok megszegése nélkül. Mivel az egyirányú utak biztonságosabbak, úgy döntöttek, hogy annyi egyirányú forgalmat hoznak létre, amennyit csak lehetséges. Hogy ne okozzanak túl nagy zavart, olyan döntést is hoztak, hogy a már létező egyirányú utakon a forgalom irányát nem szabad megváltoztatni.

A feladatod, hogy új forgalmi rendet alakíts ki a városban. Meg kell határoznod a forgalom irányát a lehető legtöbb kétirányú úton, biztosítva azt, hogy továbbra is mindkét irányban lehessen utazni bármely két kereszteződés között.

Feladat

Írj programot, amely

• a standard inputról beolvassa a város útrendszerének leírását,

• minden kétirányú útra meghatároz egy forgalmi irányt, vagy meghagyja az utat kétirányúnak,

• kiírja az eredményt a standard outputra.

Input

A bemenet első sora két egész számot tartalmaz (n és m), ahol $2 \le n \le 2000$ és $n - 1 \le m \le n(n-1)/2$. n a városban lévő kereszteződések száma, m pedig az utak száma.

A következő m sor mindegyike három egész számból (a, b és c) áll egy-egy szóközzel elválasztva, ahol $1 \le a \le n$, $1 \le b \le n$, $a \ne b$ és $c \in \{1, 2\}$. Ha $c = 1$, akkor az a és b kereszteződések egy egyirányú úttal vannak összekötve, az a-tól a b irányába. Ha $c = 2$, akkor az a és b kereszteződések egy kétirányú úttal vannak összekötve. Bármely két kereszteződés között legfeljebb egy út létezik.

Output

A kimenet pontosan annyi sorból álljon, ahány kétirányú út van a bemeneten. Minden ilyen útra (tetszőleges sorrendben) a programnak három egész számot kell kiírnia (a, b és c), amely az a-ból b-be vezető út új irányát jelenti (ha $c = 1$), vagy hogy az a-t és b-t összekötő út kétirányú marad (ha $c = 2$). Ha egynél több megoldás létezik maximális számú egyirányú utakkal, akkor a program bármelyiket kiírhatja, de csak egyet.

Példa input

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

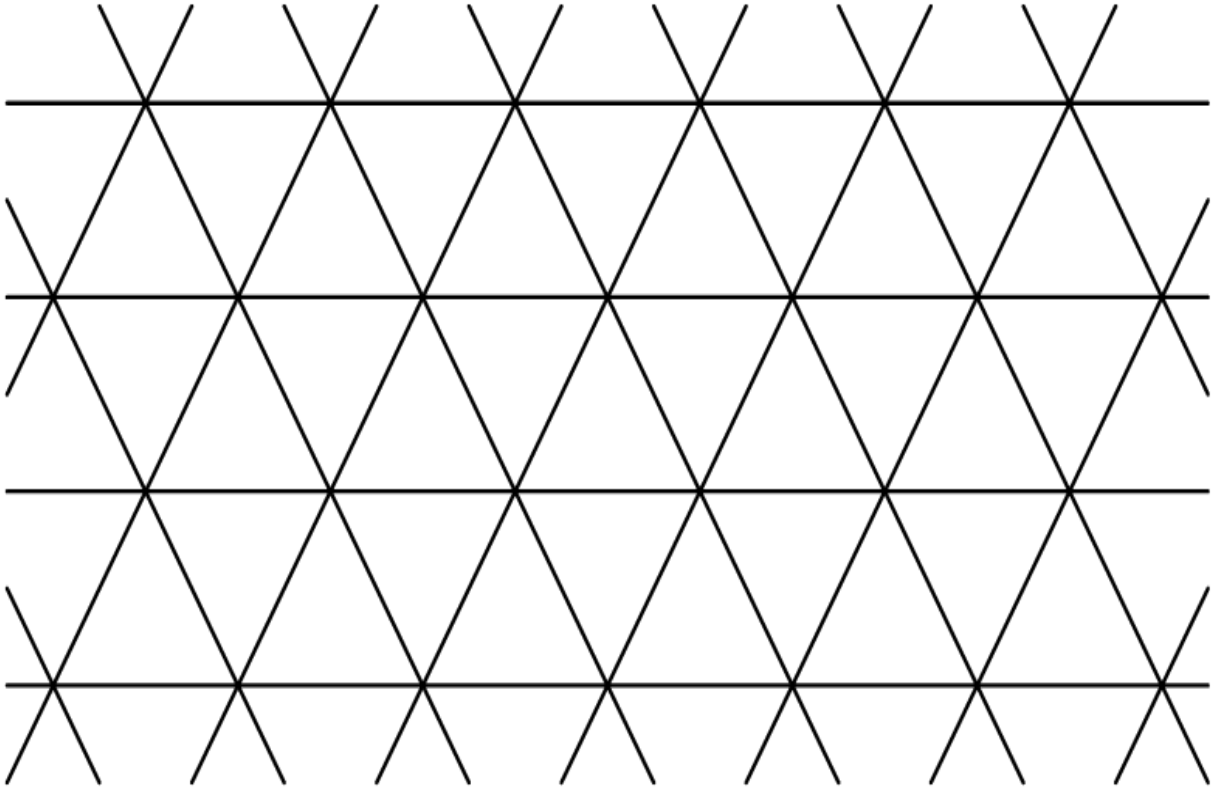
Példa output

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

Rombuszok

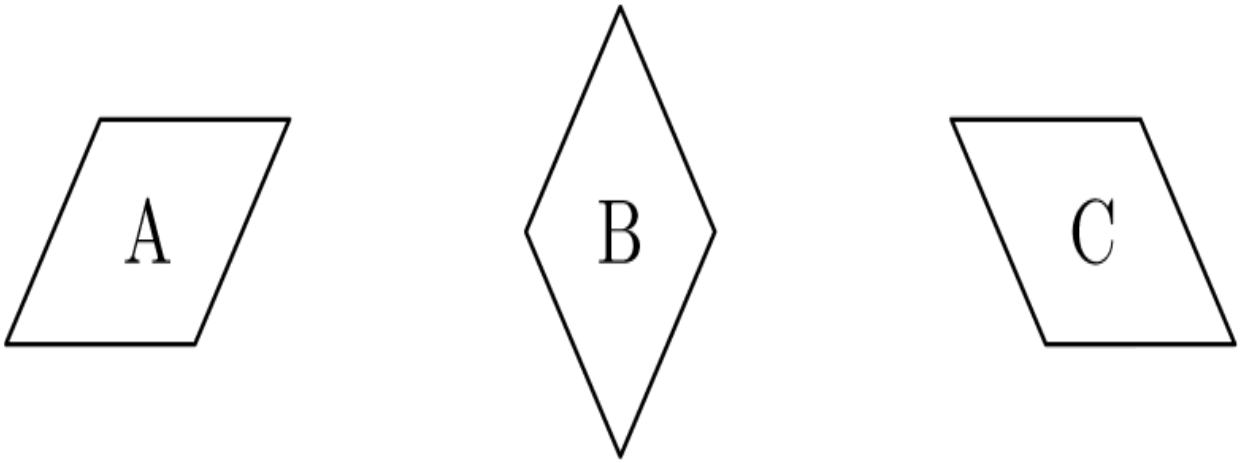
A végtelen háromszögrács egy egyenlőszárú háromszögekkel lefedett sík:

18.1. ábra -



A rácson két szomszédos háromszög egy rombuszt alkot. Ezeknek a rombuszoknak 3 fajtája létezik:

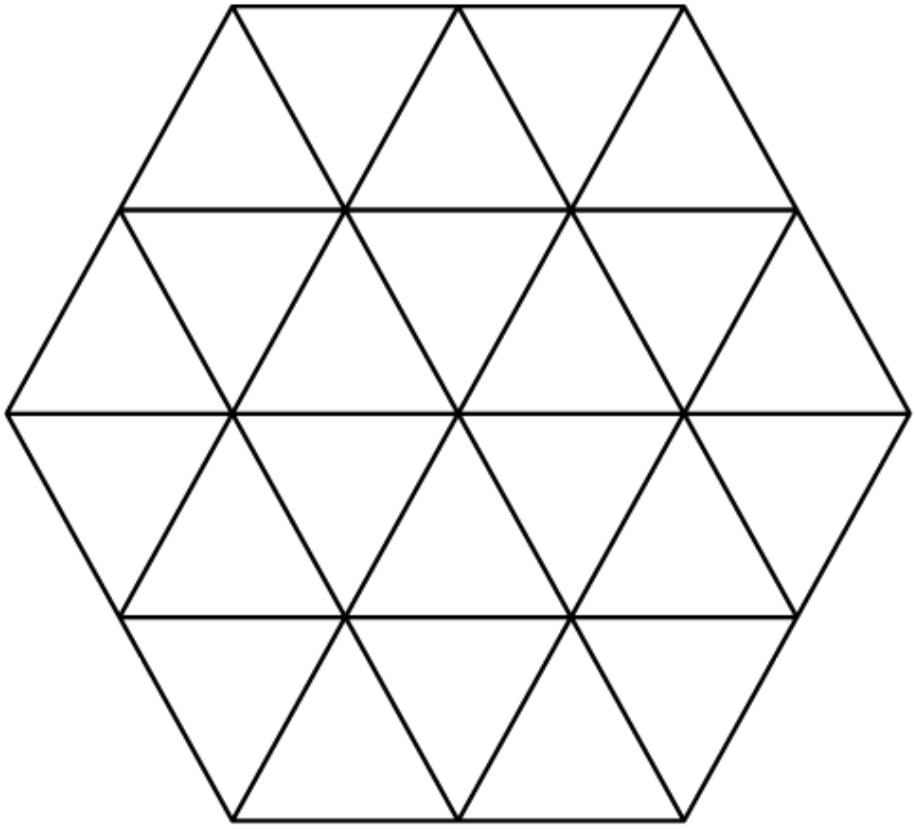
18.2. ábra -



A rácssokszög egy olyan egyszerű sokszög, amelynek oldalait teljes egészében a rács háromszögeinek az oldalai alkotják. Azt mondjuk, hogy egy rácssokszög rombikus, ha felosztható nem átfedő A, B és C típusú rombuszokra.

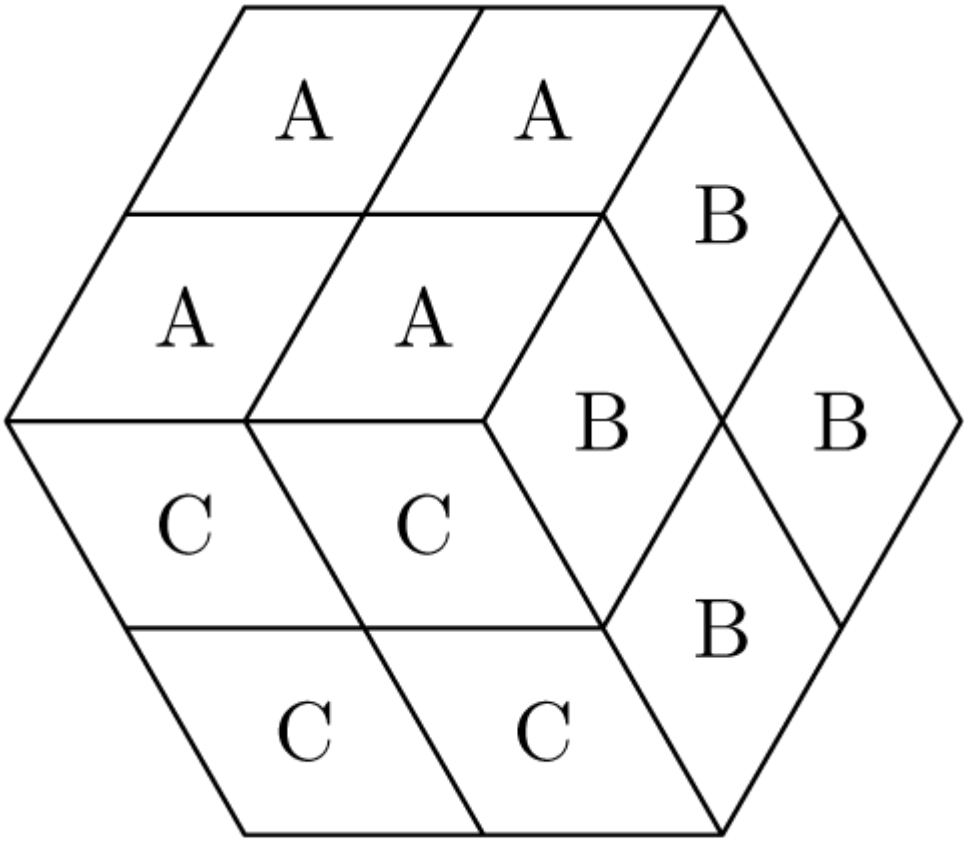
Tekintsük például a 18.3. ábrán látható rácshatszöget:

18.3. ábra -



Ez a hatszög 4 A típusú, 4 B típusú és 4 C típusú rombuszra osztható fel:

18.4. ábra -



Feladat

Írj programot, amely

• a standard inputról beolvassa egy rombikus rácssokszög leírását,

• kiszámítja az A, B és C típusú rombuszok számát a sokszög valamely helyes felosztásában,

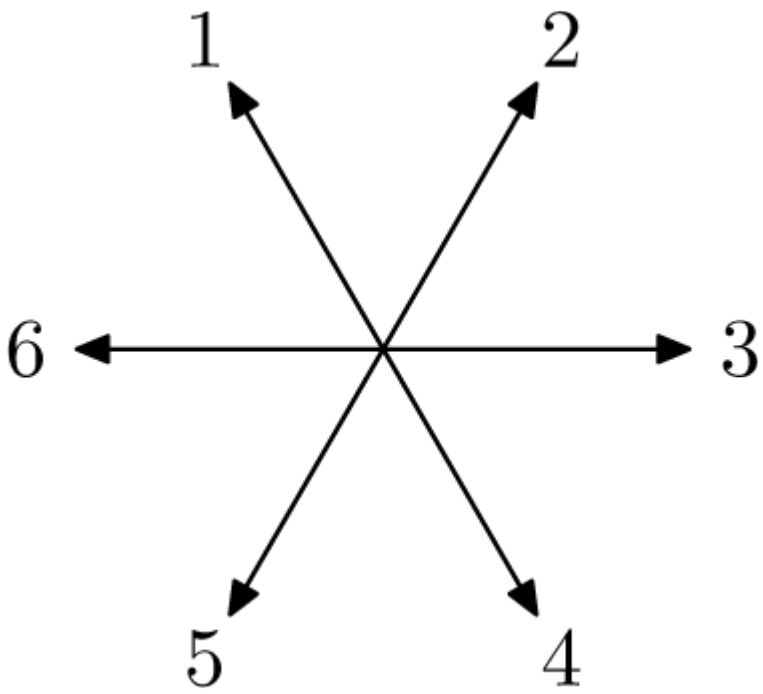
• kiírja az eredményt a standard outputra.

Input

A bemenet első sora egy n egész számot tartalmaz ($3 \leq n \leq 50000$), amely egy rombikus rácssokszög oldalainak a száma. A következő n sor a sokszög egy-egy oldalának a leírását tartalmazza. Az oldalak egyenként, az óramutató járásával megegyező irányban vannak megadva. A sokszög egyik szomszédos oldalpárja sem esik egy egyenesbe. Egy oldal leírása két egész számból áll (d és k) egy szóközzel elválasztva. A d az oldal irányát mondja meg a 18.5. ábra szerint.

A k a sokszög oldalhossza rácsháromszögoldalak darabszámában mérve. A k számok összege nem haladja meg a 100000-et.

18.5. ábra -



Output

A kimenet első és egyetlen sorának három egész számot kell tartalmaznia egy-egy szóközzel elválasztva, amelyek rendre az A, B és C típusú rombuszok számát jelentik a bemeneti sokszög valamely felosztásában.

Példa input

```
6
1 2
2 2
3 2
4 2
5 2
6 2
```

Példa output

```
4 4 4
```

Szerverek

A Bájtföldi Királyság egy nagy számítógépes hálózatot kíván kiépíteni, amely különböző szolgáltatásokat kínáló szerverekből áll.

A hálózat n szerverből áll, amelyek kétirányú kábelekkel vannak összekötve. Két szerver közvetlenül legfeljebb egy kábellel köthető össze. Minden szerver legfeljebb 10 másik szerverrel állhat közvetlen kapcsolatban, és bármely két szerver összeköttetésben áll egymással valamilyen útvonalon a hálózatban. Minden kábel egy fix pozitív adatátviteli idővel rendelkezik, amelyet milliszekundumban adunk meg. A V és a W szerverek közötti $\delta(V,W)$ távolság

(milliszekundumban) a V-t és W-t összekötő (az átviteli idő szerinti) legrövidebb út hossza a hálózatban. Az egyszerűség kedvéért legyen $\delta(V,V)=0$ minden V-re.

Egyes szerverek több szolgáltatást kínálnak, mint mások. Ezért minden V szerverhez hozzárendelünk egy $r(V)$ természetes számot, amelyet rangnak hívunk. Minél nagyobb egy szerver rangja, annál nagyobb a teljesítménye.

Minden szerver a közeli szerverekről is tárol adatokat. Azonban nem minden szerver érdekes. Alacsony rangú, távoli szerverekről nem szükséges adatokat tárolni. Pontosabban: egy W szerver akkor érdekes egy V szerver számára, ha minden olyan U szerverre, melyre $\delta(V,U) \le \delta(V,W)$, teljesül, hogy $r(U) \le r(W)$.

Például minden maximális rangú szerver érdekes az összes szerver számára. Ha a V szerver maximális rangú, akkor V számára pontosan a maximális rangú szerverek érdekesek. Jelölje $B(V)$ a V szerver számára érdekes szerverek halmazát.

Ki akarjuk számítani a szerverekről letárolandó adatok összmennyiségét, amely az összes $B(V)$ halmaz méretének az összege. A Bájtföldi Királyság azt akarta, hogy ez az érték nagyon kicsi legyen, ezért úgy építette meg a hálózatot, hogy az összeg ne legyen nagyobb, mint $30n$.

Feladat

Írj programot, amely

• a standard inpuról beolvassa egy szerverhálózat leírását,

• kiszámítja a szerverekről letárolandó adatok összmennyiségét,

• kiírja az eredményt a standard outputra.

Input

A bemenet első sora két természetes számot (n és m) tartalmaz egy szóközzel elválasztva, ahol n a szerverek száma a hálózatban ($1 \le n \le 30000$), m pedig a kábelek száma ($1 \le m \le 5n$).

A szerverek rangjai a következő n sorban vannak megadva. Az i-edik sor egy $r(i)$ egész számot tartalmaz ($1 \le r(i) \le 10$), amely az i-edik szerver rangja.

A kábeleket a következő m sor írja le. Minden kábel három számmal van megadva (a, b és t), amelyek között egy-egy szóköz áll ($1 \le t \le 1000$, $1 \le a,b \le n$, $a \ne b$). a és b a kábel által összekötött két szerver, t pedig a kábel átviteli ideje milliszekundumban mérve.

Output

A kimenet egyetlen egész számból álljon, amely a szerverekről letárolandó adatok összmennyisége.

Példa input

```
4 3
2
3
1
1
1 4 30
2 3 20
3 4 20
```

Példa output

(mert $B(1)=\{1,2\}$, $B(2)=\{2\}$, $B(3)=\{2,3\}$, $B(4)=\{1,2,3,4\}$)

Solitaire

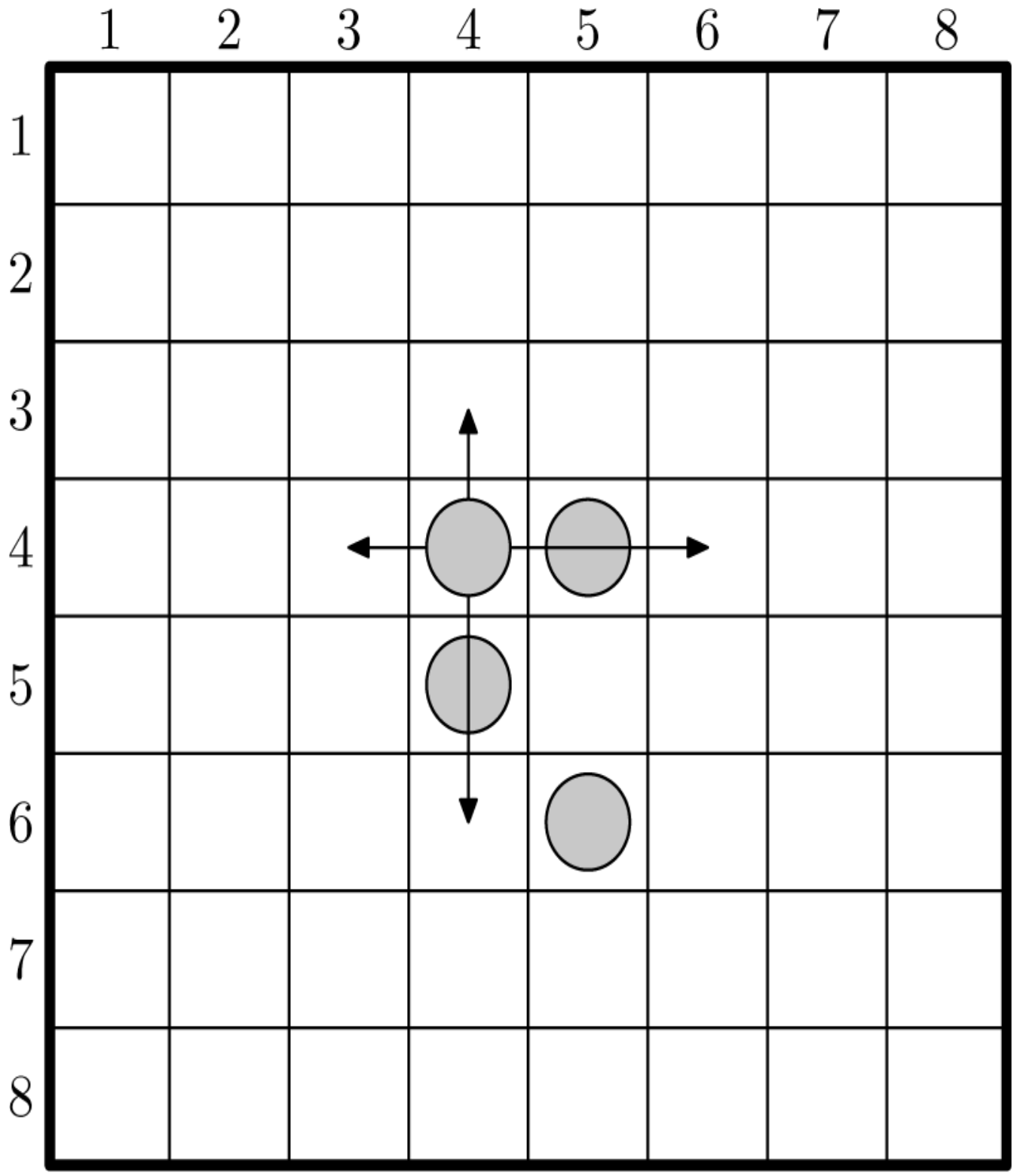
A Solitaire egy 8×8-as sakktáblán játszott játék. A sakktábla sorai és oszlopai 1-től 8-ig vannak számozva, fentről lefelé, illetve balról jobbra.

Van négy egyforma bábu a pályán. Kétféleképpen szabad lépni:

• egy bábuval egy szomszédos üres mezőre (fel, le, balra vagy jobbra),

• egy bábuval egy szomszédos bábut átugorva egy üres mezőre (fel, le, balra vagy jobbra két mezővel).

18.6. ábra -



A 18.6. ábrán minden bábu pontosan 4 lépést tehet. Vegyük például a 4. sor 4. oszlopában álló bábut. Ezt a bábut elmozgathatjuk egy sorral felfelé, két sorral lefelé, egy oszloppal balra vagy két oszloppal jobbra.

Feladat

Írj programot, amely

• a standard inputról beolvas két sakktáblaállást,

• leellenőrzi, hogy a második elérhető-e az elsőből 8 lépésen belül,

• kiírja az eredményt a standard outputra.

Input

A bemenet mindkét sora 8 egész számot tartalmaz ($a_1, a_2, \ldots, a_8$) egy-egy szóközzel elválasztva, és egy sakktáblaállást ír le. Az a_{2j-1} és az a_{2j} egész számok ($1 \leq j \leq 4$) egy-egy bábu pozícióját adják meg, rendre a sorszámmal és az oszlopszámmal.

Output

A kimenetnek egy szóból kell állnia: `IGEN`, ha a második sorban megadott állás elérhető az első sorban megadott állásból legfeljebb 8 lépésben, és `NEM` egyébként.

Példa input

```
4 4 4 5 5 4 6 5
2 4 3 3 3 6 4 6
```

Példa output

```
IGEN
```

Menetrend

Egy n várost összekötő vasúthálózat tulajdonosa vagy, ahol a városok 1-től n-ig vannak számozva. Minden vonat az induló állomástól a célállomásig megy egy speciális menetrend szerint (mindig pontosak), anélkül, hogy menet közben megállnának. Minden állomáson elérhető az indulási menetrend. Sajnos a menetrendek csak a közvetlen járatok adatait tartalmazzák. Egy utas, aki el akar jutni a p városból a q városba, nem csak a közvetlen járatokat használhatja, át is szállhat. Az átszállások nem vesznek igénybe időt, de az utas nem szállhat át az egyik vonatról a másikra, ha az utóbbi azelőtt indul, mielőtt az előbbi megérkezik. Szeretnénk, ha lenne egy, az összes optimális csatlakozást tartalmazó menetrendünk. Egy p városból A órakor induló, és q városba B órakor érkező csatlakozást optimálisnak nevezünk, ha nincs olyan csatlakozás, amely p-ből indul legkorábban A órakor, és q-ba érkezik legkésőbb B órakor. Csak azok a csatlakozások érdekesek a számunkra, amelyek egy napon indulnak és érkeznek.

Feladat

Írj programot, amely

• a standard inputról beolvassa a városok számát (n) és a menetrendeket,

• elkészíti az első városból az n-edik városba tartó optimális csatlakozások menetrendjét,

• kiírja az eredményt a standard outputra.

Input

A bemenet első sora egy n egész számot tartalmaz ($2 \leq n \leq 100000$). A következő sorok n darab menetrendet tartalmaznak, rendre az első, második, , n-edik városokra vonatkozóan.

A menetrendleírás első sora egyetlenegy egész számból áll (m). A következő m sor mindegyike

egy-egy menetrendi bejegyzésnek felel meg, és tartalmazza az A indulási időt, a B érkezési időt ($A < B$) és a célállomás sorszámát, t-t ($1 \leq t \leq n$), egy-egy szóközzel elválasztva. Az A indulási és a B érkezési idők *óó*:*pp* alakúak, ahol *óó* két számjegy, amely az órát jelöli ($00 \leq \text{óó} \leq 23$), *pp* pedig két számjegy, amely a percet jelöli ($00 \leq pp \leq 59$). A menetrendi bejegyzések indulási idők szerint nem csökkenő sorrendben vannak megadva. A bejegyzések száma egyik menetrendben sem haladja meg az 1000000-t.

Output

A kimenet első sorának egy r egész számot kell tartalmaznia, amely a megoldást jelentő menetrend bejegyzéseinek a száma. A következő r sor mindegyike egy A indulási időt és egy B érkezési időt tartalmazzon egy szóközzel elválasztva. Az idő formátumának ugyanolyannak kell lennie, mint az input adatoké volt, és a menetrendben a bejegyzéseket indulási idők szerint növekvően kell rendezni. Ha egynél több optimális csatlakozás létezik azonos indulási és érkezési időkkel, a programnak csak egyet kell kiírnia.

Példa input

```
3
3
09:00 15:00 3
10:00 12:00 2
11:00 20:00 3
2
11:30 13:00 3
12:30 14:00 3
0
```

Példa output

```
2
10:00 14:00
11:00 20:00
```

Falánk Steve

Steve és Digit vettek egy fánkokkal teli dobozt. Hogy felosszák maguk között a fánkokat, egy általuk kitalált speciális játékot játszanak. A játékosok felváltva vesznek ki legalább egy fánkot a dobozból, de nem többet, mint egy adott egész szám. A játékosok a fánkjaikat maguk előtt gyűjtik. Az a játékos, amelyik kiüríti a dobozt, megeszi az összegyűjtött fánkjait, míg a másik a sajátjait visszarakja a dobozba, és a játékot a ,,vesztes" játékos (aki visszarakta a fánkjait) folytatja. A játék addig tart, amíg az összes fánkot meg nem eszik. A játékosok célja, hogy minél több fánkot egyenek meg. Hány fánkra számíthat Steve, aki a játékot kezdi, feltéve, hogy mindkét játékos a legjobb stratégiája szerint játszik?

Feladat

Írj programot, amely

• beolvassa a standard inputról a játék paramétereit,

• kiszámolja, hogy hány fánkra számíthat Steve,

• kiírja az eredményt a standard outputra.

Input

A bemenet első és egyetlen sora pontosan két egész számot tartalmaz (n és m), amelyeket egyetlen

szóköz választ el egymástól, $1 \leq m \leq n \leq 100$. Ezek a játék paraméterei, ahol n a játék kezdetekor a dobozban lévő fánkok száma, m pedig az egy játékos által egy lépésben kivehető fánkok számának a felső korlátja.

Output

A kimenetnek pontosan egy egész számot kell tartalmaznia, amely azon fánkok száma, amennyire Steve számíthat.

Példa input

```
5 2
```

Példa output

```
3
```

19. fejezet - ACM közép-európai döntő, 2003, Varsó, Lengyelország

Tartalom

Könnyű feladat?

Egy adott feladat nehézségi fokát a következő statisztikai adatokkal jellemezhetjük: a feladat elfogadott megoldásainak a száma, a beküldött megoldások átlagos száma és a megoldáshoz felhasznált átlagos idő (ahogy a verseny általános szabályaiban le van írva: „egy feladat elfogadott megoldásához felhasznált idő az az idő, amely a verseny kezdetétől az elfogadott megoldás beküldéséig eltelt"). Az utóbbi két statisztikát csak azon csapatoknál vesszük figyelembe, akik megoldották a feladatot.

Feladat

Írj programot, amely

• beolvassa egy ACM-verseny beküldött megoldásainak listáját,

• minden feladatra kiszámítja a feladat elfogadott megoldásainak a számát, a beküldött megoldások átlagos számát és a megoldásához felhasznált átlagos időt,

• kiírja az eredményt.

Input

A bemenet első sora egy n egész számot tartalmaz ($1 \leq n \leq 2000$), amely a verseny folyamán beküldött összes megoldás száma. A következő n sor egy-egy beküldött megoldást ír le az alábbi adatokkal: beküldési idő (a verseny kezdetétől eltelt másodpercek száma), a csapatazonosító, a

feladatazonosító és a megoldás kiértékelésének az eredménye, mindezek egy-egy darab szóközzel elválasztva. A beküldési idő egy pozitív egész szám, amely nem nagyobb, mint 18 000. A csapatazonosító egy nem üres sztring, amely legfeljebb öt kisbetűt vagy számjegyet tartalmaz. A feladatazonosító az 'A', 'B', , 'I' nagybetűk valamelyike. Az eredmény szintén egy nagybetű, lehet 'A' (elfogadott) vagy 'R' (nem elfogadott).

A megoldások a beküldési idő szerint nem csökkenő sorrendben vannak megadva. A versenyben 60 csapat vesz részt.

Vegyük figyelembe, hogy ha egy feladatot egy csapat már sikeresen megoldott, ugyanezen feladatra ugyanez a csapat küldhet ugyan be további megoldásokat, de ezeket már nem szabad figyelembe venni.

Output

A kimenet kilenc sorból álljon. Az első sor az A feladathoz tartozzon, a második a B-hez, és így tovább. Minden sor tartalmazza a feladatazonosítót, a feladatra beküldött elfogadott megoldások számát, a megoldások átlagos számát azon csapatokat figyelembe véve, amelyek megoldották a feladatot, és a megoldáshoz felhasznált átlagos időt, mindezeket egy-egy szóközzel elválasztva. Az utolsó két statisztikát csak akkor kell kiírni, ha legalább egy elfogadott megoldás érkezett az adott feladatra, és két tizedesjegyre kell őket kerekíteni (például az $1,235$-et $1,24$-ra kell kerekíteni).

Példa input

```
12
10 wawu1 B R
100 chau1 A A
2000 uwr2 B A
2010 wawu1 A R
2020 wawu1 A A
2020 wawu1 B A
4000 wawu2 C R
6000 chau1 A R
7000 chau1 A A
8000 pp1 A A
8000 zil2 B R
9000 zil2 B A
```

Példa output

```
A 3 1.33 3373.33
B 3 1.67 4340.00
C 0
D 0
E 0
F 0
G 0
H 0
I 0
```

Kötegelés

Az Outel, egy híres félvezetőgyártó cég, nemrégiben kiadott egy új mikroprocesszor-modellt, amit Platiniumnak neveztek el. Mint sok modern processzor, a Platinium is több utasítást tud végrehajtani egy órajelciklus alatt, feltéve hogy nincs köztük függőség (az I_2 utasítás függ az I_1 utasítástól, ha például I_2 olyan regisztert olvas, amit I_1 ír). Némely processzor olyan okos, hogy menet közben számítja ki, hogy mely utasítások hajthatók végre biztonságosan párhuzamosan. A

Platinium azonban elvárja, hogy ezt az információt explicit módon megadjuk. Egy két utasítás közé beszúrt speciális jel -- amit egyszerűen *stopnak* hívunk -- jelzi, hogy egyes stop utáni utasítások függhetnek egyes stop előtti utasításoktól. Más szóval: két egymást követő stop közötti utasításokat párhuzamosan hajthatunk végre, azaz feltehető, hogy nincs közöttük függőség.

A Platinium egy másik érdekes jellemzője, hogy az utasítások sorozatát egy, kettő vagy három egymást követő utasításból álló csoportokba kell osztani. Minden csoportot egy tárolóban kell elhelyezni, amelyet *kötegnek* hívunk. Minden kötegnek 3 rekesze van, minden rekeszbe egy utasítás fér be, de néhány rekesz üresen is maradhat. Minden utasítás besorolható 10 utasítástípus valamelyikébe, amelyeket az A, B, , J nagybetűkkel jelölünk (az azonos típusú utasítások hasonlóan működnek, például A típusúak az egész aritmetikai utasítások, F típusúak a lebegőpontos utasítások). Csak bizonyos típusú utasításokat lehet egy kötegbe pakolni. Egy megengedett utasítástípus-kombinációt egy kötegen belül egy *minta* határoz meg. Egy minta meghatározhatja egy köteg közepén egy stop pozícióját is (legfeljebb egy ilyen stop megengedett). Ráadásul a stopok megengedettek bármely két szomszédos köteg között is. A minták egy halmazát *kötegelési profilnak* nevezzük. Amikor az utasításokat kötegekbe pakoljuk, csak a kötegelési profilban lévő mintákat használhatjuk.

Bár a Platinium el van látva utasítás cache-sel, úgy találták, hogy a maximális teljesítmény eléréséhez a legfontosabb, hogy az utasításokat olyan sűrűn pakoljuk, ahogy csak lehetséges. A második legfontosabb dolog, hogy kis számú stopot alkalmazzunk.

A feladatod egy olyan program írása, amely Platinium-utasításokat kötegel. Az egyszerűség kedvéért feltételezzük, hogy az utasításokat nem lehet átrendezni.

Feladat

Írj programot, amely

• beolvas egy kötegelési profilt, és utasításoknak egy sorozatát,

• kiszámítja azon kötegek minimális számát, amelyekbe az utasítássorozat bepakolható anélkül, hogy megtörnénk a függőségeket, valamint a minimális számú köteghez szükséges stopok minimális számát,

• kiírja az eredményt.

Input

A bemenet első sora két egész számot tartalmaz (t és n) egy szóközzel elválasztva. A t ($1 \le t \le 1500$) a kötegelési profilban található minták száma. Az n ($1 \le n \le 100000$) a kötegelendő utasítások száma.

A következő t sor mindegyike egy-egy mintát ír le, amely 3 nagybetűből, t_1-ből, t_2-ből és t_3-ból áll (közöttük nincsenek szóközök), ezeket egy szóköz után egy p egész szám követi. A t_i betű ($A \le t_i \le J$) az i-edik rekeszben megengedett utasítástípus. A p ($0 \le p \le 2$) annak a rekesznek az indexe, amelyik után a stop elhelyezkedik (a 0 azt jelenti, hogy nincs stop a kötegben).

A következő n sor mindegyike egy-egy utasítást ír le. Ennek az n sornak az i-edik sora egy c_i nagybetűt és egy d_i egész számot tartalmaz egy szóközzel elválasztva. A c_i ($A \le c_i \le J$) az i-edik utasítás típusa. A d_i ($0 \le d_i < i$) az utolsó olyan utasítás indexe (az előzőek közül), amelytől az i-edik utasítás függ (a 0 azt jelenti, hogy az utasítás nem függ egyik korábbi utasítástól sem).

Feltehetjük, hogy minden c utasítástípusra, amely az utasítássorozatban előfordul, legalább egy minta létezik, amely tartalmazza c-t.

Output

A kimenet első és egyetlen sorának két egész számot kell tartalmaznia (b-t és s-et) egy szóközzel elválasztva. A b a kötegek minimális száma egy érvényes elrendezésben. Az s a minimális számú köteghez szükséges stopok minimális száma.

Példa input

```
4 9
ABB 0
BAD 1
AAB 0
ABB 2
B 0
B 1
A 1
A 1
B 4
D 0
A 0
B 3
B 0
```

Példa output

```
4 3
```

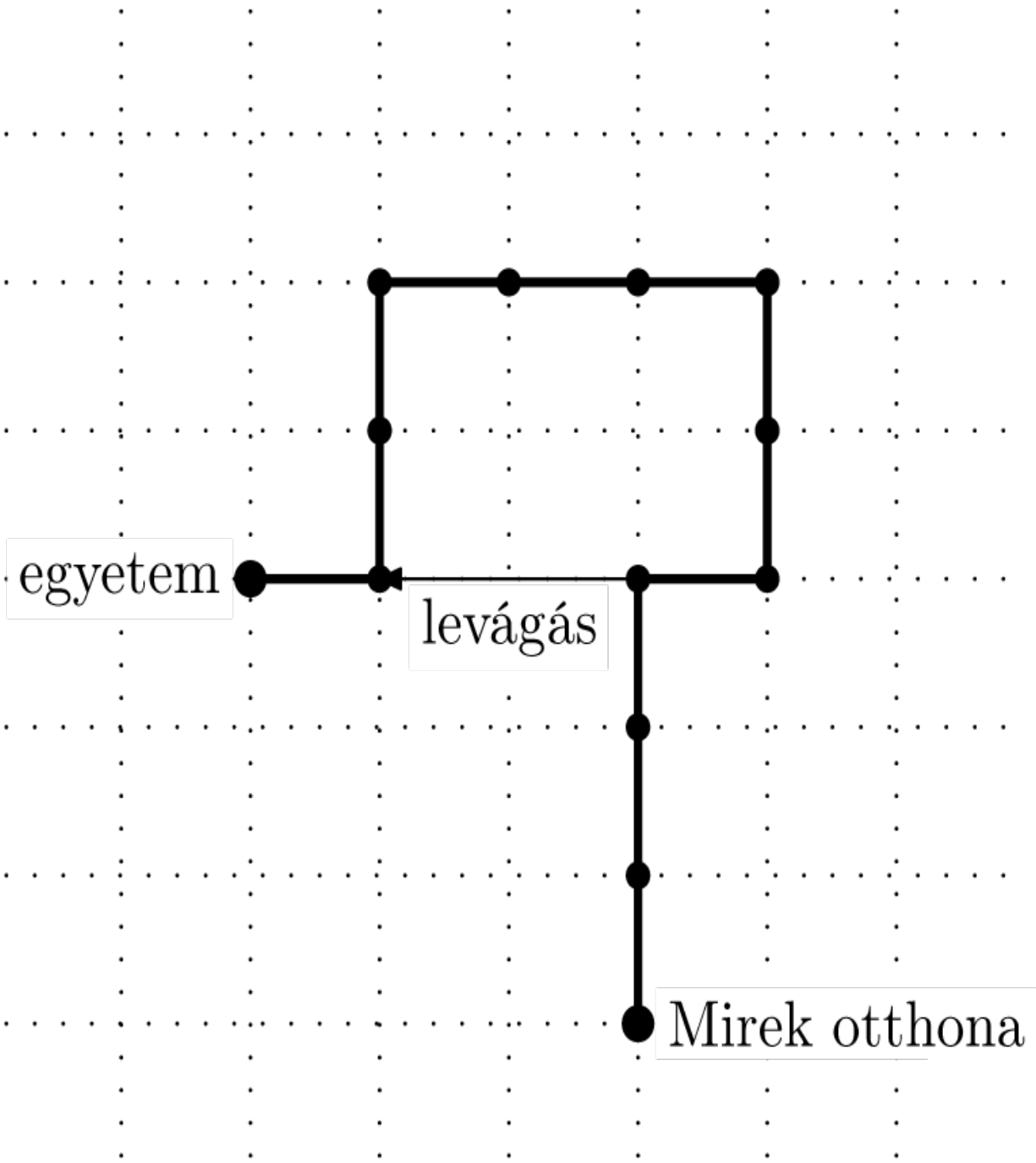
Levágás

Mireknek van egy kedvenc útvonala, amelyen munkanapokon az otthonától az egyetemig közlekedik. Az útvonal szakaszokból áll. Minden szakasz egy 10 méter hosszú egyenes. Minden szakasz vagy közvetlen meghosszabbítása az előző szakasznak, vagy merőleges arra. Miután Mirek megtesz egy szakaszt, egy rövid pihenőt tart, hogy megcsodálja a természet szépségeit. Sétája során soha nem érinti ugyanazt a helyet kétszer.

Tegnap Mirek sokáig bulizott, ezért ma későn kelt fel. Tudja, hogy lekési az első óráját, hacsak nem változtat a szokásos útvonalán. Úgy tervezi, hogy egy levágást csinál, de szeretné, ha ez a levágás a lehető legrövidebb lenne (titokban elárulhatjuk, hogy valójában nem akar pontos lenni, csak a lelkiismeretét akarja megnyugtatni). A levágásnak vagy egy függőleges, vagy egy vízszintes szakasznak kell lennie, amely összeköti Mirek útvonalának két töréspontját.

Segíts Mireknek megtalálni a legrövidebb levágást!

19.1. ábra -



Feladat

Írj programot, amely

• beolvassa Mirek útvonalát,

• kiszámítja az útvonalon lévő legrövidebb levágást,

• kiírja az eredményt.

Input

A bemenet első sora egy n egész számot tartalmaz ($3 \leq n \leq 250000$), ami az útvonalon lévő szakaszok száma. A bemenet második sora az N, E, S, W karakterek n hosszúságú sorozata, amely nem tartalmaz szóközöket. Minden karakter az útvonal egy szakaszának a leírása. Az N, E, S, W karakterek rendre azt jelentik, hogy Mirek 10 métert sétál északra, keletre, délre vagy nyugatra. Feltehető, hogy legalább egy levágás létezik a megadott útvonalon.

Output

A kimenet első és egyetlen sorának az l,b,e egészeket és a d karaktert kell tartalmaznia egymástól egy-egy szóközzel elválasztva. Az l a legrövidebb levágás hossza (10 méteres szakaszokban), a b és e rendre a levágás kezdő és záró töréspontjának a sorszámai (a töréspontokat egymást követő egész számokkal számozzuk 0-tól n-ig, azaz Mirek otthonától az egyetemig). A d karakter a levágás iránya. Ha egynél több minimális hosszúságú levágás létezik, azt kell kiírni, amelyik az útvonalon előbb kezdődik. Ha egynél több minimális hosszúságú levágás kezdődik ugyanannál a töréspontnál, azt kell kiírni, amelyik az útvonalon a legtávolabb végződik.

Példa input

```
12
NNNENNWWWSSW
```

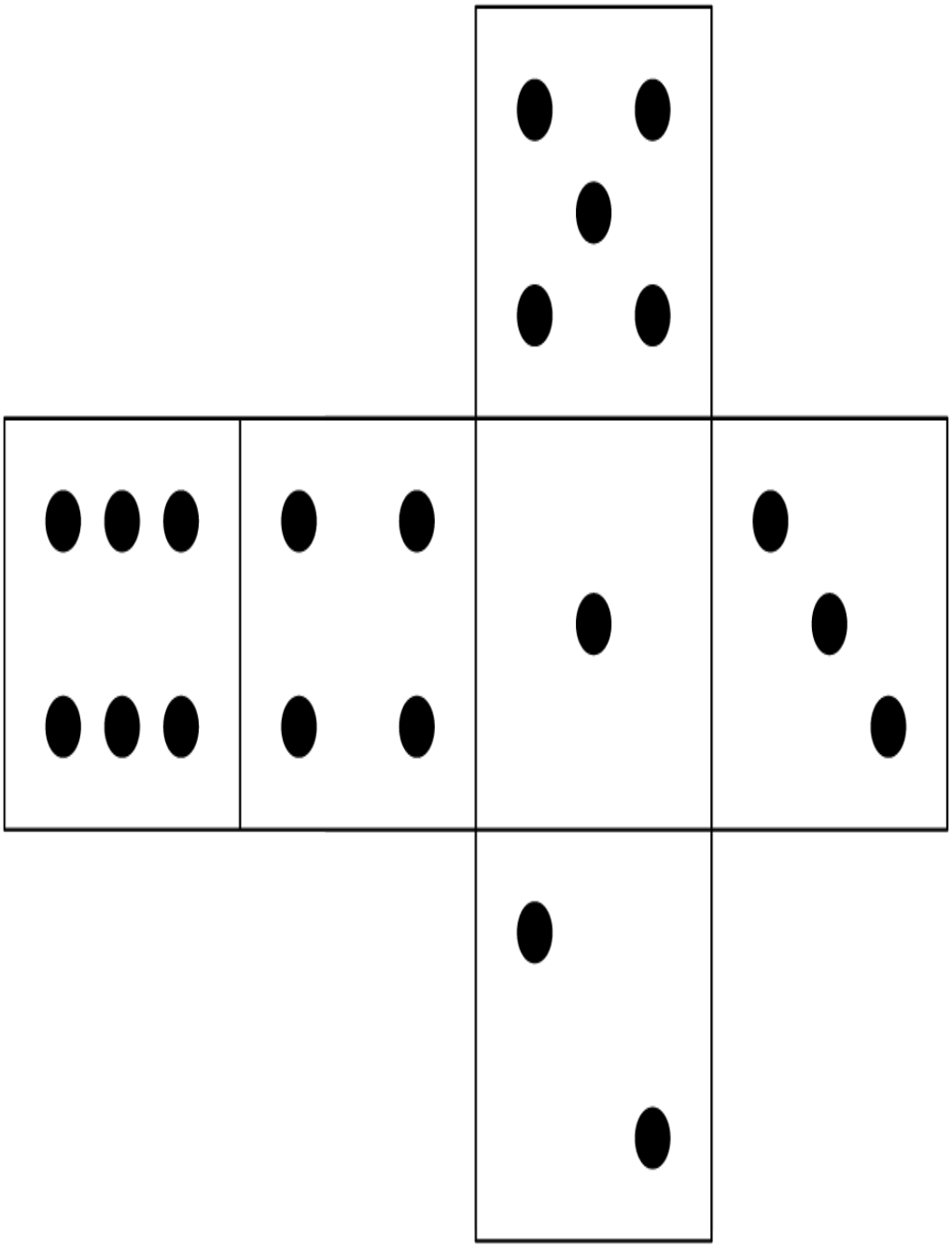
Példa output

```
2 3 11 W
```

Dobókockaverseny

Kockavárosban mindenki szeret játszani. Minden szombaton az egész közösség összegyűlik, hogy részt vegyenek egy kockaversenyben. Pár évvel ezelőtt kezdtek egy hagyományos hatoldalú dobókockával, amelynek az oldalain $1,2,\ldots,6$ darab pont van, és igen jól szórakoztak.

19.2. ábra -



Hamarosan azonban megunták, ezért manapság már rafináltabb dobókockákat használnak. Az oldalakra papírcetliket ragasztottak, amelyekre egy-egy pozitív egész számot írtak.

A versenyt egy táblán rendezik, amely mezőkre van felosztva úgy, mint a sakktábla. A tábla négy mező széles, és végtelen balra és jobbra (mondhatnánk, hogy ilyen nem létezik a valóságban, igaz?). A tábla sorai 1-től 4-ig vannak sorszámozva, alulról felfelé, az oszlopai pedig egymást követő egész számokkal balról jobbra. Minden mezőt egy (x,y) párral azonosítunk, ahol x az oszlopszám, y pedig a sorszám.

A játék úgy kezdődik, hogy a kockát a versenybizottság által választott mezőre helyezik, az 1 pontos felével felfelé, a 2 pontos felével a játékos felé. A kocka mozgatásához a játékosnak a kockát az egyik szélén át kell billenteni egy (vízszintesen vagy függőlegesen) szomszédos mezőre. A mozgatás költsége a görgetés után a kocka tetején látható szám. A játék célja, hogy átgörgessük a kockát a kezdő mezőből egy kiválasztott célmezőbe úgy, hogy a mozgatások összköltsége minimális legyen.

Feladat

Írj programot, amely

• beolvassa egy kocka leírását, a kezdő mezőt és a célmezőt,

• kiszámítja a kockának a kezdő mezőből a célmezőbe való görgetésének a minimális költségét,

• kiírja az eredményt.

Input

A bemenet első sora hat egész számot ($l_1, l_2, l_3, l_4, l_5, l_6$) tartalmaz egy-egy szóközzel elválasztva ($1 \le l_i \le 50$). Az l_i az a szám, amelyet arra az oldalra írunk, amelyen eredetileg i pont volt. A bemenet második sora négy egész számot (x_1, y_1, x_2, y_2) tartalmaz egy-egy szóközzel elválasztva ($-10^9 \le x_1, x_2 \le 10^9$, $1 \le y_1, y_2 \le 4$). Az x_1, y_1 a kezdő mező oszlop- és sorszáma. Az x_2, y_2 a célmező oszlop- és sorszáma.

Output

A kimenet első és egyetlen sora a kockának a kezdő mezőből a célmezőbe való görgetésének a minimális költségét tartalmazza.

Példa input

```
1 2 3 8 1 4
-1 1 0 2
```

Példa output

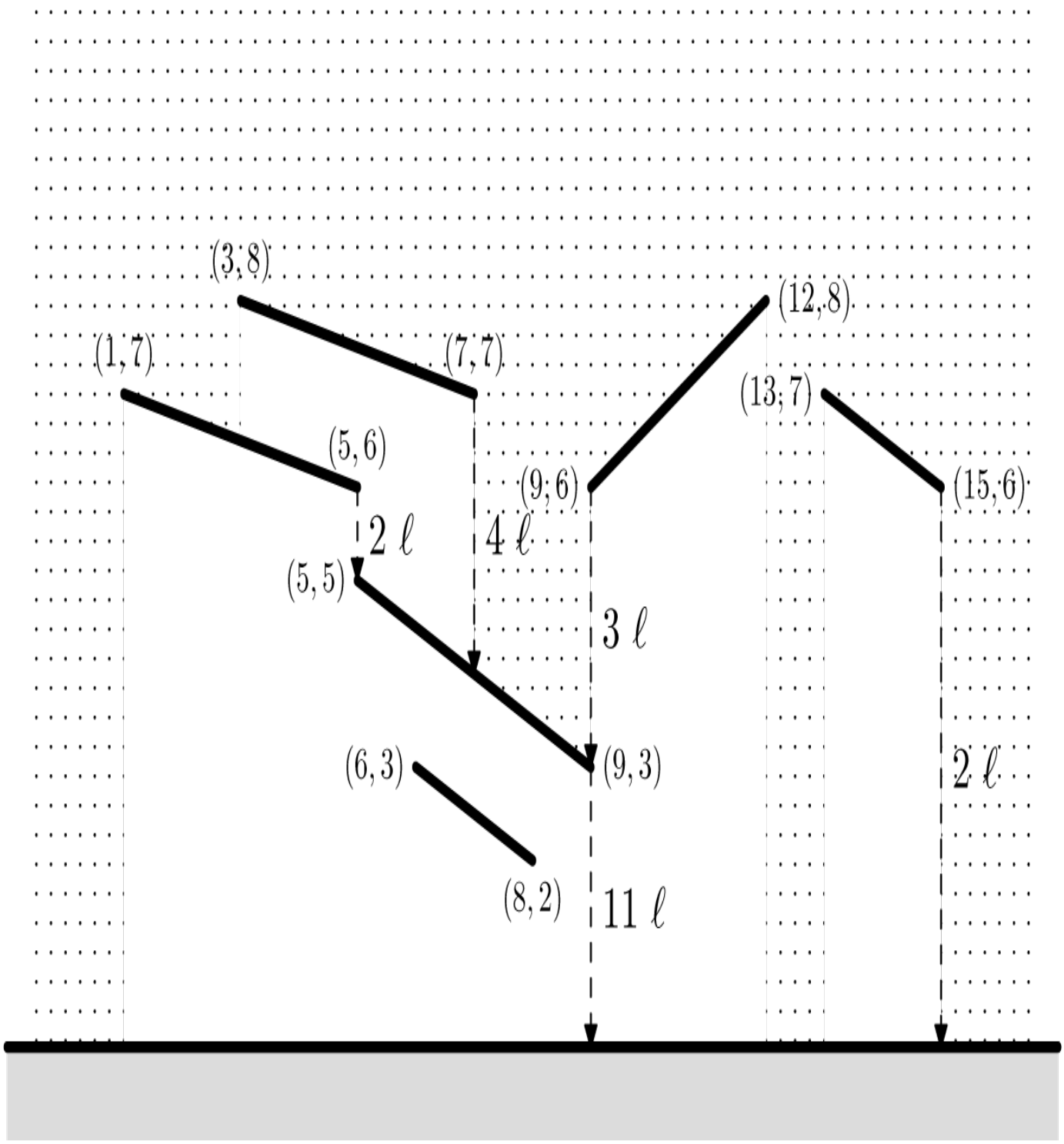
```
7
```

Novemberi eső

A mai épületeknek nagyon komplikált tetőzetük lehet. Ha egy ilyen tetőzetnek vesszük a függőleges metszetét, egy sor lejtős szegmenst kapunk. Ha esik, az esőcseppek függőlegesen esnek a tetőre az égből. Bizonyos szegmensek ki vannak téve az időjárás viszontagságainak, de lehetnek olyan szegmensek, amelyek részben vagy akár teljesen védve vannak más szegmensekkel. Egy szegmensre eső összes víz függőlegesen lefolyik a szegmens alsó végéről a földre, vagy esetleg valamelyik másik szegmensre. Konkrétan, ha egy vízfolyam egy szegmensről ráfolyik egy másikra, akkor ez a másik szegmens gyűjti össze azt.

Egy csatornarendszer tervezéséhez ki kell számítani, hogy mennyi víz folyik le a tetőzet egyes szegmenseiről. Hogy felkészüljünk a kemény novemberi esőkre,

19.3. ábra -



úgy számolunk, hogy egy méternyi vízszintes síkra másodpercenként egy liter esővíz esik.

Feladat

Írj programot, amely

• beolvassa a tetőzet leírását,

• kiszámítja hogy egy másodperc alatt mennyi víz folyik le a tetőzet egyes szegmenseiről,

• kiírja az eredményt.

Input

A bemenet első sora egy darab n egész számot tartalmaz ($1 \leq n \leq 40000$), amely a tetőzet szegmenseinek a száma. A következő n sor mindegyike a tetőzet egy-egy szegmensét írja le négy egész számmal (x_1, y_1, x_2, y_2), egy-egy szóközzel elválasztva őket ($0 \leq x_1, y_1, x_2, y_2 \leq 1000000$, $x_1 < x_2$, $y_1 \neq y_2$). Az x_1 a szegmens bal végének vízszintes pozíciója, az y_1 pedig ugyanennek a magassága. Az x_2 a szegmens jobb végének vízszintes pozíciója, az y_2 pedig ugyanennek a magassága. A szegmenseknek nincs közös pontjuk, és nincsenek vízszintes szegmensek. Azt is feltételezhetjük, hogy legfeljebb 25 szegmens van egymás felett bármely pontban.

Output

A kimenet n sorból álljon, az i-edik sor tartalmazza az i-edik szegmensről másodpercenként lefolyó víz mennyiségét (literben).

Példa input

```
6
13 7 15 6
3 8 7 7
1 7 5 6
5 5 9 3
6 3 8 2
9 6 12 8
```

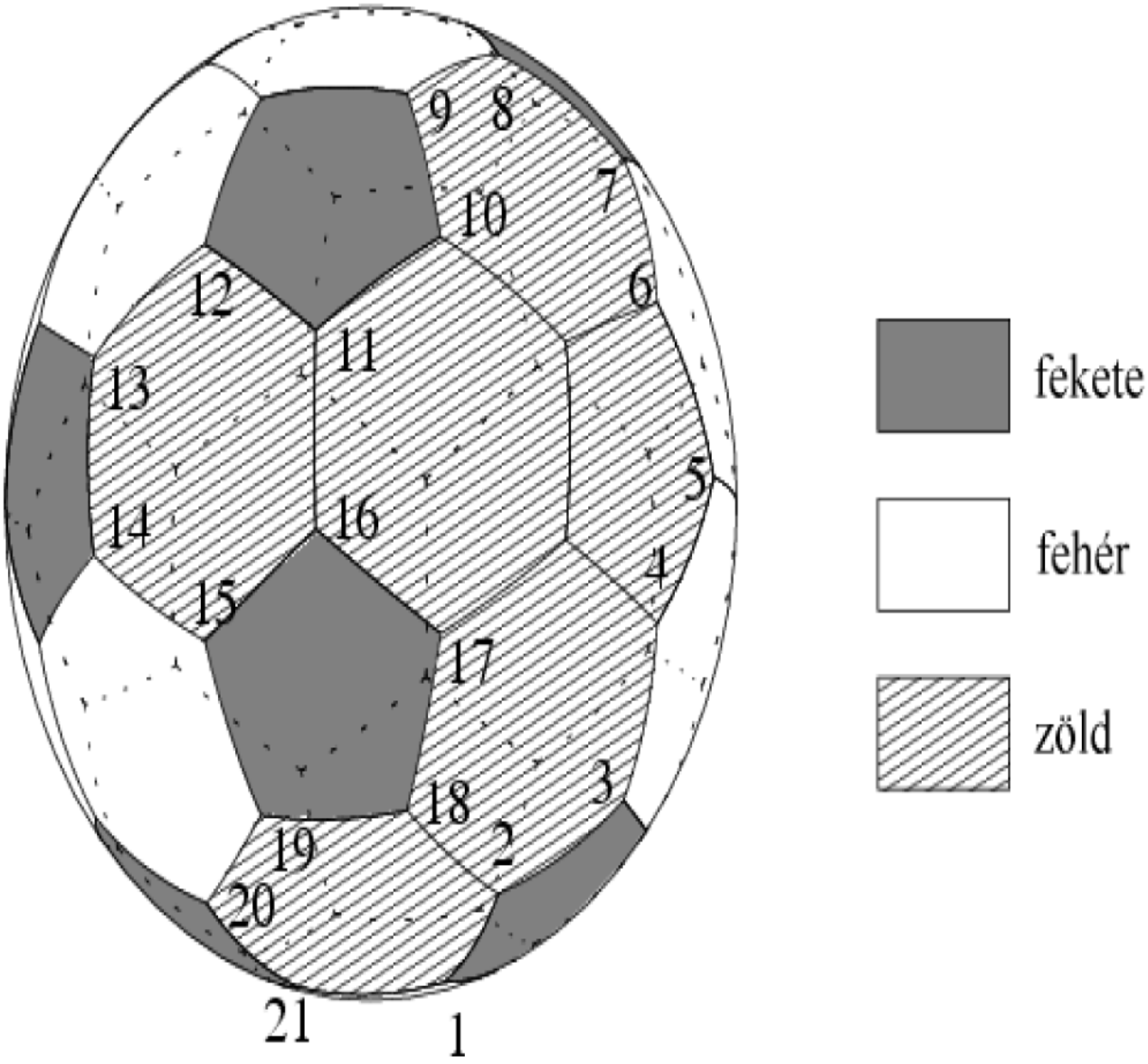
Példa output

```
2
4
2
11
0
3
```

Focilabda

Eric-nek van egy klasszikus focilabdája, amelyet 32 bőrdarabból varrtak: 12 fekete ötszögből és 20 fehér hatszögből. Minden ötszög 5 hatszöggel szomszédos, és minden hatszög 3 ötszöggel és 3 hatszöggel szomszédos. Eric rajzolt egy sokszöget (azaz egy metszéspontok nélküli zárt vonalat) a darabok szélei mentén. A sokszög a labdát két részre osztotta, melyből Eric az egyiket befestette zöldre.

19.4. ábra -



Feladat

Írj programot, amely

• beolvassa egy sokszög leírását,

• kiszámítja a fekete, a fehér és a zöld darabok számát,

• kiírja az eredményt.

Input

A bemenet első sora egy n egész számot tartalmaz, amely a sokszög csúcsainak a száma. A bemenet második sora n darab egész számot tartalmaz ($a_1, a_2, \ldots, a_n$) egy-egy szóközzel

elválasztva. Az a_i egész szám (ami 1 vagy 2 lehet) a sokszög i-edik csúcsával szomszédos zöld darabok száma. A sokszögnek az n-edik és az első csúcsot összekötő oldala mindig két hatszög között található.

Output

A kimenet első és egyetlen sorának három egész számot kell tartalmaznia (b-t, w-t és g-t), a fekete, a fehér és a zöld darabok számát.

Példa input

```
21
1 2 1 2 1 2 1 1 1 2 2 1 1 1 1 2 2 2 1 1 1
```

Példa output

```
11 15 6
```

Melyik a következő?

Minden informatikus hallgató ismeri a bináris fákat. Íme a bináris fák sok lehetséges definíciója közül az egyik. A bináris fákat induktívan definiáljuk. A t bináris fa vagy egy $\circ$ külső csúcs (levél), vagy egy $t=(t_1,t_2)$ rendezett páros, amely egy $\bullet$ belső csúcsot jelöl, amelyhez két részfa kapcsolódik: a t_1 bal oldali és t_2 jobb oldali részfa. Ezzel a definícióval a csúcsok száma bármely bináris fában páratlan. Ha adott egy n páratlan egész szám, jelölje $B(n)$ az n (belső és külső) csúccsal rendelkező bináris fák halmazát. Például $B(1)$ csak egy fából áll ($\circ$), $B(3)=\{(\circ,\circ)\}$ és $B(5)=\{(\circ,(\circ,\circ)),((\circ,\circ),\circ)\}$. A $B(5)$-höz tartozó fák a 19.5. ábrán láthatók:

19.5. ábra -

Jelölje $|t|$ a t fában lévő csúcsok számát. Egy adott t fa $N(t)$ egyedi egész azonosítóját a következőképpen definiáljuk:

- $N(\circ)=0$
- $N(t_1,t_2)=2^{|t_1|+|t_2|}+2^{|t_2|}\cdot N(t_1)+N(t_2)$

Például $N(\circ,\circ)=2^2+2^1\cdot 0+0=4$, $N(\circ,(\circ,\circ))=2^4+2^3\cdot 0+4=20$, $N((\circ,\circ),\circ)=2^4+2^1\cdot 4+0=24$.

Tekintsük a bináris fákon értelmezhető alábbi lineáris rendezést:

$\circ \preceq t$

$(t_1,t_2) \preceq$ t_1 u_1 , *vagy* t_1=u_1 *és* t_2 u_2

Ebben a rendezésben a $\circ$ levél a legkisebb fa, és két nem levél fa közül az a kisebb, amelyiknek kisebb a bal oldali részfája, ha a bal oldali részfák különbözőek, egyébként az a kisebb, amelyiknek kisebb a jobb oldali részfája.

Ezért például $(\circ,(\circ,\circ)) \prec ((\circ,\circ),\circ)$, mivel $\circ \prec (\circ,\circ)$. Tegyük fel, hogy a $B(n)$-beli fákat a $\preceq$ relációval rendeztük. Ekkor minden $B(n)$-beli t fára definiáljuk t rákövetkezőjét, amely az a fa, amely közvetlenül követi t-t $B(n)$-ben. Ha t a legnagyobb fa $B(n)$-ben, akkor t rákövetkezője a $B(n)$ halmazbeli legkisebb fa. Például $(\circ,\circ)$ rákövetkezője $B(3)$-ban ugyanaz a $(\circ,\circ)$ fa, $(\circ,(\circ,\circ))$ rákövetkezője $B(5)$-ben $((\circ,\circ),\circ)$.

Feladat

Írj programot, amely

• beolvassa valamely t bináris fa azonosítóját,

• kiszámítja t $B(|t|)$-beli rákövetkezőjének az azonosítóját,

• kiírja az eredményt.

Input

A bemenet első és egyetlen sora egy n egész számot tartalmaz ($0 \le n \le 2^{30}$), amely egy t bináris fa azonosítója.

Output

A kimenet első és egyetlen sorának egy s egész számot kell tartalmaznia, amely a t $B(|t|)$-beli rákövetkezőjének az azonosítója.

Példa input

```
20
```

Példa output

```
24
```

Megáll vagy nem áll meg?

A kis Tom programozni tanul. Épp most írt néhány programot, de fél lefuttatni őket, mert nem tudja, hogy megállnak-e. Írj egy programot, hogy segíts neki.

Ez a feladat nem is olyan könnyű, mint amilyennek látszik, mert Tom programjai nemdeterminisztikusan viselkedhetnek.

Ha adott egy Tom által írt program, a te programodnak meg kell mondania, hogy a programja megállhat-e, és ha igen, mi a legrövidebb lehetséges idő, amire megáll.

Tom számítógépe 32 darab egybites regisztert tartalmaz, és a program n utasításból áll. A regiszterek 0-tól 31-ig, az utasítások pedig 0-tól $(n-1)$-ig vannak megszámozva.

A továbbiakban MEM[a] az a -adik regiszter tartalmát jelenti, $0 \leq a,b < 32$, $0 \leq x < n$, $0 \leq c \leq 1$.

Az utasításkészlet a következő:

Utasítás	Szemantika
AND a b	MEM[a] := MEM[a] and MEM[b]
OR a b	MEM[a] := MEM[a] or MEM[b]
XOR a b	MEM[a] := MEM[a] xor MEM[b]
NOT a	MEM[a] := not MEM[a]
MOV a b	MEM[a] := MEM[b]
SET a c	MEM[a] := c
RANDOM a	MEM[a] := véletlen érték (0 vagy 1)
JMP x	ugrás az x -edik utasításra
JZ x a	ugrás az x -edik utasításra, ha $MEM[$ a $]=0$
STOP	a program megállítása

Minden program utolsó utasítása STOP (bár egynél több STOP utasítás is szerepelhet egy programban). Minden program a nulladik utasítással kezdődik. Indulás előtt a regiszterek tetszőleges értékeket tartalmazhatnak. Minden utasítás végrehajtása (a STOP utasításé is) egy ciklusidőt vesz igénybe.

Feladat

Írj programot, amely

• beolvassa Tom programját,

• kiszámítja a program lehetséges legrövidebb futási idejét,

• kiírja az eredményt.

Input

A bemenet első sora egy n egész számot tartalmaz ($1 \leq n \leq 16$), amely a program utasításainak a száma. A következő n sor mindegyike a program egy utasítását tartalmazza a fent megadott formában. Feltehető, hogy a programban az egyetlen fehér karakter az utasítások elemei közötti egyetlen szóköz.

Output

A kimenet első és egyetlen sorának a program lehetséges legrövidebb futási idejét kell tartalmaznia, ciklusidőben mérve. Ha a program nem állhat meg, a kimenetre a NEM ÁLL MEG szöveget kell írni.

Példa input

```
5
SET 0 1
JZ 4 0
RANDOM 0
JMP 1
STOP
```

Példa output

```
6
```

Példa input

```
5
MOV 3 5
NOT 3
AND 3 5
JZ 0 3
STOP
```

Példa output

```
NEM ÁLL MEG
```

A Maximalizáló minimalizálása

A Chris Kft. egy új, Maximalizáló nevű rendezőgépet készít. A Maximalizálónak n bemenete van, amelyek 1-től n-ig vannak megszámozva. Minden bemenet egy egész számot takar. A Maximalizálónak egy kimenete van, amely a bemenetein megjelenő értékek maximuma.

A Maximalizálót a $Sorter(i_1, j_1), \ldots, Sorter(i_k, j_k)$ rendezők sorozatával valósítják meg. Minden rendezőnek n bemenete és n kimenete van. A $Sorter(i, j)$ az $i, i+1, \ldots, j$ bemeneteken található értékeket rendezi nemcsökkenő sorrendbe, a többi bemenetet változatlanul továbbadja. A Maximalizáló kimenete az utolsó rendező n-edik kimenete.

Egy belső ember (egy korábbi ACM versenyző) megfigyelte, hogy egyes rendezőket ki lehet hagyni a sorozatból, és a Maximalizáló így is helyes eredményt szolgáltat. Mi lesz a hossza a rendezők azon legrövidebb részsorozatának, amely a bemeneti érték minden lehetséges kombinációjára helyes eredményt szolgáltat?

Feladat

Írj programot, amely

• beolvassa a Maximalizáló leírását, azaz rendezők egy kezdeti sorozatát,

• kiszámítja a rendezők kezdeti sorozata azon legrövidebb részsorozatának hosszát, amely minden lehetséges bemeneti adatra helyes eredményt szolgáltat,

• kiírja az eredményt.

Input

A bemenet első sora két egész számot tartalmaz (n és m) egy szóközzel elválasztva ($2 \le n \le 50000$, $1 \le m \le 500000$). Az n a bemenetek száma, m pedig a sorozatban lévő rendezők száma. A rendezők kezdeti sorozatát a következő m sor tartalmazza. Ezen sorok közül a k-adik a k-adik rendező paramétereit írja le: i_k és j_k két egész szám ($1 \le i_k < j_k \le n$) egy szóközzel elválasztva.

Output

A kimenet egyetlen sorból álljon, amely egy egész számot tartalmaz, a rendezők kezdeti sorozata azon legrövidebb részsorozatának hosszát, amely minden lehetséges bemeneti adatra helyes eredményt szolgáltat.

Példa input

```
40 6
20 30
1 10
10 20
20 30
15 25
30 40
```

Példa output

```
4
```

Irodalomjegyzék

Brian, W. Kernighan-Dennis M. Ritchie. *A C programozási nyelv*. Műszaki Könyvkiadó. Budapest. 2001.

Donald, E. Knuth. *A számítógépprogramozás művészete 1. (Alapvető algoritmusok)*. Műszaki Könyvkiadó. Budapest. 1994.

Donald, E. Knuth. *A számítógépprogramozás művészete 3. (Keresés és rendezés)*. Műszaki Könyvkiadó. Budapest. 1994.

Robert, Sedgewick. *Algorithms in C*. Addison--Wesley. 1990.

Seymour, Lipschutz. *Adatszerkezetek*. Panem--McGraw-Hill. Budapest. 1993.

Zsakó, László. *Programozási feladatok I--II*. Kossuth Kiadó. Budapest. 1997.

ACM, Problem Set Archive. *http://acm.uva.es/problemset*. Universidad de Valladolid. http://acm.uva.es/problemset.

C, Language Reference Manual. *http://techpubs.sgi.com/library*. Silicon Graphics, Inc.. http://techpubs.sgi.com/library.