Első C programok

Kormányos Andor

Komplex Rendszerek Fizikája Tanszék

2019. szeptember 9.

A minimális C program

A minimális C program a `main` függvény megvalósítását jelenti:

```
1 int main(int argc, char* argv[]) {
2   return 0;
3 }
```

- `main`: ez a függvény neve
  - a program végrehajtása mindig a `main`-nel kezdődik
  - a függvénynév általában tetszőleges karaktersor, a `main` speciális
- `int`: ez a függvény visszatérési értékének típusa
  - az `int` jelentése két bájtos egész szám
  - a `main` esetében mindig `int` a visszatérési érték típusa
  - később majd látjuk, hogy általában mire való a visszatérési érték
- `return 0;` - a függvény visszatérési értéke
  - a `main` esetében ez magának a programnak a visszatérési kódja
  - általában 0, ha nem 0, akkor valami hibakódot jelent
- `(int argc, char* argv[])` - parancssori argumentumok
  - ezek a `main` függvény paraméterei
  - ezekre később visszatérünk

Egy másik egyszerű példa

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

int main()
{
    printf("Hello world!\n");
    return 0;
}
```

Egy másik egyszerű példa

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("Hello world!\n");
7      return 0;
8  }
```

- A C-ben nincsen *beépített* függvény
  - de vannak standard könyvtárakat
  - ezeket az `#include` direktívával kell meghivatkozni

Egy másik egyszerű példa

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("Hello world!\n");
7      return 0;
8  }
```

- A C-ben nincsen *beépített* függvény
  - de vannak standard könyvtárakat
  - ezeket az `#include` direktívával kell meghivatkozni
- Ha nem akarunk a parancssorról beolvasni a `main` argumentumlistája üres

Egy másik egyszerű példa

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("Hello world!\n");
7      return 0;
8  }
```

- A C-ben nincsen *beépített* függvény
  - de vannak standard könyvtárakat
  - ezeket az `#include` direktívával kell meghivatkozni
- Ha nem akarunk a parancssorról beolvasni a `main` argumentumlistája üres
- `printf` függvény
  - szöveg kiírása a konzolablakba
  - a függvényhívás a függvény nevéből áll, melyet egy zárójeles rész követ
  - a zárójelek között a paramétereket kell felsorolni
  - paraméter lehet egyetlen változó, de összetett kifejezés is

Egy egyszerű adatmodell és algoritmus

Másodfokú egyenlet megoldóképlete

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Adatmodell

- három valós szám a memóriában az együtthatóknak
- egy valós szám a diszkriminánsnak
- két valós szám a gyököknek

Algoritmus:

- olvasd be a három számot
- számítsd ki a diszkriminánst
- ha a diszkrimináns negatív, írj ki egy hibát, és állj meg
- ha a diszkrimináns nulla, írd ki a gyököt és állj meg
- ha a diszkrimináns pozitív, írd ki a két gyököt és állj meg

Az első valódi program

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

int main() {
  double a, b, c;
  double d, r1, r2;
  a = 1.0;
  b = 3.0;
  c = 2.0;
  d = b * b - 4 * a * c;
  d = sqrt(d);
  r1 = (-b + d) / (2 * a);
  r2 = (-b - d) / (2 * a);
  printf("r1 = %f, r2 = %f\n", r1, r2);
  return 0;
}
```

Az első valódi program

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main() {
5     double a, b, c;
6     double d, r1, r2;
7     a = 1.0;
8     b = 3.0;
9     c = 2.0;
10    d = b * b - 4 * a * c;
11    d = sqrt(d);
12    r1 = (-b + d) / (2 * a);
13    r2 = (-b - d) / (2 * a);
14    printf("r1 = %f, r2 = %f\n", r1, r2);
15    return 0;
16  }
```

- standard könyvtárak

Az első valódi program

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main() {
5     double a, b, c;
6     double d, r1, r2;
7     a = 1.0;
8     b = 3.0;
9     c = 2.0;
10    d = b * b - 4 * a * c;
11    d = sqrt(d);
12    r1 = (-b + d) / (2 * a);
13    r2 = (-b - d) / (2 * a);
14    printf("r1 = %f, r2 = %f\n", r1, r2);
15    return 0;
16  }
```

- standard könyvtárak
- változók **deklarálása** és értékadás
  - itt valójában ennyi az adatmodell: öt szám
  - `double` - dupla pontosságú tört szám

Az első valódi program

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main() {
5     double a, b, c;
6     double d, r1, r2;
7     a = 1.0;
8     b = 3.0;
9     c = 2.0;
10    d = b * b - 4 * a * c;
11    d = sqrt(d);
12    r1 = (-b + d) / (2 * a);
13    r2 = (-b - d) / (2 * a);
14    printf("r1 = %f, r2 = %f\n", r1, r2);
15    return 0;
16  }
```

- standard könyvtárak
- változók **deklarálása** és értékadás
  - itt valójában ennyi az adatmodell: öt szám
  - `double` - dupla pontosságú tört szám
- változók értékeinek inicializálása
  - inicializálás nélkül a változóban szemét van! nem nulla!
  - később megnézzük, hogyan lehet a paramétereket kívülről beolvasni

Az első valódi program

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main() {
5     double a, b, c;
6     double d, r1, r2;
7     a = 1.0;
8     b = 3.0;
9     c = 2.0;
10    d = b * b - 4 * a * c;
11    d = sqrt(d);
12    r1 = (-b + d) / (2 * a);
13    r2 = (-b - d) / (2 * a);
14    printf("r1 = %f, r2 = %f\n", r1, r2);
15    return 0;
16  }
```

- standard könyvtárak
- változók **deklarálása** és értékadás
  - itt valójában ennyi az adatmodell: öt szám
  - `double` - dupla pontosságú tört szám
- változók értékeinek inicializálása
  - inicializálás nélkül a változóban szemét van! nem nulla!
  - később megnézzük, hogyan lehet a paramétereket kívülről beolvasni
- matematikai műveletek elvégzése

Az első valódi program

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

int main() {
  double a, b, c;
  double d, r1, r2;
  a = 1.0;
  b = 3.0;
  c = 2.0;
  d = b * b - 4 * a * c;
  d = sqrt(d);
  r1 = (-b + d) / (2 * a);
  r2 = (-b - d) / (2 * a);
  printf("r1 = %f, r2 = %f\n", r1, r2);
  return 0;
}
```

- standard könyvtárak
- változók **deklarálása** és értékadás
  - itt valójában ennyi az adatmodell: öt szám
  - `double` - dupla pontosságú tört szám
- változók értékeinek inicializálása
  - inicializálás nélkül a változóban szemét van! nem nulla!
  - később megnézzük, hogyan lehet a paramétereket kívülről beolvasni
- matematikai műveletek elvégzése
- eredmények **formázott** kiírása a konzolablakba (“%f”)

Az első valódi program

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main() {
5     double a, b, c;
6     double d, r1, r2;
7     a = 1.0;
8     b = 3.0;
9     c = 2.0;
10    d = b * b - 4 * a * c;
11    d = sqrt(d);
12    r1 = (-b + d) / (2 * a);
13    r2 = (-b - d) / (2 * a);
14    printf("r1 = %f, r2 = %f\n", r1, r2);
15    return 0;
16  }
```

- standard könyvtárak
- változók **deklarálása** és értékadás
  - itt valójában ennyi az adatmodell: öt szám
  - `double` - dupla pontosságú tört szám
- változók értékeinek inicializálása
  - inicializálás nélkül a változóban szemét van! nem nulla!
  - később megnézzük, hogyan lehet a paramétereket kívülről beolvasni
- matematikai műveletek elvégzése
- eredmények **formázott** kiírása a konzolablakba (“%f”)
- a program visszatérési értéke 0

A program lefordítása és futtatása

Ha a program a `1.c` fájlban van, akkor a fordítás menete:

- `gcc 1.c -o firstprog -lm`

```
1  $ ls -lt
2  -rwxrwx--- 1 kor kor  286 szept  3 10:53   1.c
3  -rw-r--r-- 1 kor kor   94 szept  3 10:38   1.py
4  $ gcc 1.c -o firstprog -lm
```

Ez a parancs létrehoz egy `firstprog` nevű futtatható fájlt:

```
1  $ ls -lt
2  -rwxrwxr-x 1 kor kor 8344 szept  3 11:00   firstprog
3  -rwxrwx--- 1 kor kor  286 szept  3 10:53   1.c
4  -rw-r--r-- 1 kor kor   94 szept  3 10:38   1.py
```

Ezek után jön a futtatás:

```
1  $ ./firstprog
2  r1 = -1.000000, r2 = -2.000000
```

Az első program Python-ban

```
1  a = 1.0
2  b = 3.0
3  c = 2.0
4  d = sqrt(b**2-4*a*c)
5  r1 = (-b+d)/(2*a)
6  r2 = (-b-d)/(2*a)
7  print(r1,r2)
```

- A Python gyakorlattal szemben C-ben az egész program egy függvény.
- Pythonban nem kellett deklarálni, C-ben kötelező
- C-ben az utasítások végére ";" kell, Pythonban nem kell
- Pythonban nem kell format string a kiíráshoz

Az előző program egy kicsit tömörebben

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main() {
5     double a = 1.0;
6     double b = 3.0;
7     double c = 2.0;
8     double d = sqrt(b * b - 4 * a * c);
9     double r1 = (-b + d) / (2 * a);
10    double r2 = (-b - d) / (2 * a);
11    printf("r1 = %f, r2 = %f\n", r1, r2);
12    return 0;
13  }
```

- a változók deklarálása összevonható az értékadással

Az előző program egy kicsit tömörebben

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main() {
5     double a = 1.0;
6     double b = 3.0;
7     double c = 2.0;
8     double d = sqrt(b * b - 4 * a * c);
9     double r1 = (-b + d) / (2 * a);
10    double r2 = (-b - d) / (2 * a);
11    printf("r1 = %f, r2 = %f\n", r1, r2);
12    return 0;
13  }
```

- a változók deklarálása összevonható az értékadással
- a függvények kifejezéseket is kaphatnak paraméterként
  - ilyenkor a kifejezés előbb kiértékelődik
  - a függvény a kapott eredményt kapja meg paraméterként

Kifejezések kiértékelési sorrendje, az operátorok

A megoldóképlet így néz ki

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

A képlet esetében ismerjük a zárójelezés szabályait, de hogyan kell programmá írni?

Egy kifejezésben szerepelhet

- függvényhívás (minden argumentum egy további kifejezés!)
- zárójelek
- előjelek (`+ -`)
- aritmetikai operátorok (`+ - * / %`)
- logikai operátorok (`== != < > <= >=`)

és még egy sor további operátor, amikről később tanulunk.

Kifejezések kiértékelési sorrendje, az operátorok

A megoldóképlet így néz ki

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

A képlet esetében ismerjük a zárójelezés szabályait, de hogyan kell programmá írni?

Egy kifejezésben szerepelhet

- függvényhívás (minden argumentum egy további kifejezés!)
- zárójelek
- előjelek (`+ -`)
- aritmetikai operátorok (`+ - * / %`)
- logikai operátorok (`== != < > <= >=`)

és még egy sor további operátor, amikről később tanulunk.

A végrehajtás sorrendjét az ún. **precedencia** szabja meg:

1. függvényparaméterek kiértékelése
2. függvények meghívása
3. zárójelek
4. előjelek
5. szorzás és osztás műveletek
6. összeadás és kivonás műveletek

Házi feladat: az alábbiak közül melyik helyes és melyik nem?

A megoldóképlet egyik fele így néz ki

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

1. `(-b + sqrt(b * b - 4 * a * c)) / 2 * a`
2. `(-b + sqrt(b * b - 4 * a * c)) / 2 / a`
3. `-b + sqrt(b * b - 4 * a * c) / (2 * a)`
4. `-(b + sqrt(b * b - 4 * a * c)) / (2 * a)`
5. `-(b - sqrt(b * b - 4 * a * c)) / (2 * a)`
6. `(-b + sqrt((b * b) - (4 * a * c))) / (2 * a)`
7. `-b / 2 / a + sqrt(b * b - 4 * a * c) / 2 / a`

Az első program fő problémái

1. Az egyenlet együtthatói nem paraméterek, hanem konstansok
   - ez úgy mondjuk, hogy az értékek “hard code”-olva vannak
   - az ilyet mindig kerülni kell, kivéve ha valóban konstansokról van szó

Az első program fő problémái

1. Az egyenlet együtthatói nem paraméterek, hanem konstansok
   - ez úgy mondjuk, hogy az értékek “hard code”-olva vannak
   - az ilyet mindig kerülni kell, kivéve ha valóban konstansokról van szó

2. Hogyan lehetne az együtthatókat paraméterként beadni?
   - parancssori argumentumként
   - billentyűzetről beolvasva
   - fájlból beolvasva

Az első program fő problémái

1. Az egyenlet együtthatói nem paraméterek, hanem konstansok
   - ez úgy mondjuk, hogy az értékek "hard code"-olva vannak
   - az ilyet mindig kerülni kell, kivéve ha valóban konstansokról van szó
2. Hogyan lehetne az együtthatókat paraméterként beadni?
   - parancssori argumentumként
   - billentyűzetről beolvasva
   - fájlból beolvasva
3. Mi történik akkor, ha olyan együtthatókat nézünk, ahol a diszkrimináns negatív?
   - házi feladat: kipróbálni

Parancssori paraméterek beolvasása

A parancssori argumentumokat a konzolablakban szeretnénk megadni futtatáskor, pl.:

```
1   $ ./roots 1 3 2
```

Parancssori paraméterek beolvasása

A parancssori argumentumokat a konzolablakban szeretnénk megadni futtatáskor, pl.:

```
$ ./roots 1 3 2
```

Ezeket az argumentumokat a `main` függvény paramétereiként kapjuk meg

- vigyázat: minden paraméter szövegként van kezelve
- át kell alakítani a szöveget számmá
- egyelőre itt a kész megoldás: `atof` függvény

Parancssori paraméterek beolvasása

A parancssori argumentumokat a konzolablakban szeretnénk megadni futtatáskor, pl.:

```
$ ./roots 1 3 2
```

Ezeket az argumentumokat a `main` függvény paramétereiként kapjuk meg

- vigyázat: minden paraméter szövegként van kezelve
- át kell alakítani a szöveget számmá
- egyelőre itt a kész megoldás: `atof` függvény

Parancssori paraméterek beolvasása

A parancssori argumentumokat a konzolablakban szeretnénk megadni futtatáskor, pl.:

```
$ ./roots 1 3 2
```

Ezeket az argumentumokat a `main` függvény paramétereiként kapjuk meg

- vigyázat: minden paraméter szövegként van kezelve
- át kell alakítani a szöveget számmá
- egyelőre itt a kész megoldás: `atof` függvény

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

int main(int argc, char* argv[]) {
  double a = atof(argv[1]);
  double b = atof(argv[2]);
  double c = atof(argv[3]);
  double d = sqrt(b * b - 4 * a * c);
  double r1 = (-b + d) / (2 * a);
  double r2 = (-b - d) / (2 * a);
  printf("r1 = %f, r2 = %f\n", r1, r2);
  return 0;
}
```

- Az `argc` paraméter a parancssori paraméterek számát tartalmazza
  - az első (0. indexű) paraméter mindig a futó program neve
- Az `argv` paraméterben szövegként kapjuk meg a paramétereket

Feltételes elágazások, az if utasítás

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   int main(int argc, char* argv[]) {
6     if (argc < 4) {
7       printf("Not enough arguments.\n");
8       exit(-1);
9     }
10
11    double a = atof(argv[1]);
12    double b = atof(argv[2]);
13    double c = atof(argv[3]);
14    double d = b * b - 4 * a * c;
15
16    if (d < 0) {
17      printf("No real solution.\n");
18      exit(-1);
19    } else {
20      d = sqrt(d);
21      double r1 = (-b - d) / 2 / a;
22      double r2 = (-b + d) / 2 / a;
23      printf("r1 = %f, r2 = %f\n", r1, r2);
24    }
25
26    return 0;
27  }
```

- Az if (...) utasítás zárójelében egy feltétel szerepel
  - logikai vagy aritmetikai kifejezés

Feltételes elágazások, az `if` utasítás

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  int main(int argc, char* argv[]) {
6    if (argc < 4) {
7      printf("Not enough arguments.\n");
8      exit(-1);
9    }
10
11   double a = atof(argv[1]);
12   double b = atof(argv[2]);
13   double c = atof(argv[3]);
14   double d = b * b - 4 * a * c;
15
16   if (d < 0) {
17     printf("No real solution.\n");
18     exit(-1);
19   } else {
20     d = sqrt(d);
21     double r1 = (-b - d) / 2 / a;
22     double r2 = (-b + d) / 2 / a;
23     printf("r1 = %f, r2 = %f\n", r1, r2);
24   }
25
26   return 0;
27 }
```

- Az `if (...)` utasítás zárójelében egy feltétel szerepel
  - logikai vagy aritmetikai kifejezés
- Ha feltétel teljesül (értéke nem nulla), akkor az `if` utáni `{ ... }` block fut le

Feltételes elágazások, az `if` utasítás

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  int main(int argc, char* argv[]) {
6    if (argc < 4) {
7      printf("Not enough arguments.\n");
8      exit(-1);
9    }
10
11   double a = atof(argv[1]);
12   double b = atof(argv[2]);
13   double c = atof(argv[3]);
14   double d = b * b - 4 * a * c;
15
16   if (d < 0) {
17     printf("No real solution.\n");
18     exit(-1);
19   } else {
20     d = sqrt(d);
21     double r1 = (-b - d) / 2 / a;
22     double r2 = (-b + d) / 2 / a;
23     printf("r1 = %f, r2 = %f\n", r1, r2);
24   }
25
26   return 0;
27 }
```

- Az `if (...)` utasítás zárójelében egy feltétel szerepel
  - logikai vagy aritmetikai kifejezés
- Ha feltétel teljesül (értéke nem nulla), akkor az `if` utáni `{ ... }` block fut le
- Ha a feltétel nem teljesült, az `else` ág fut le
  - az `else` ág nem kötelező

Milyen sorrendben hajtódik végre a program?

1. Az utasítások felülről lefelé hajtódnak végre
2. A program futása a `main` függvény első sorával indul
   - ekkor a parancssori argumentumok már fel lettek dolgozva, az `argv` értéke fel van töltve
3. Ha a program `if`-hez ér, akkor három dolog történhet
   - ha a feltétel teljesül, akkor a főág fut le
   - ha a feltétel nem teljesül, de van `else` ág, akkor az fut le
   - ha a feltétel nem teljesül, és nincsen `else` ág, akkor a program az `if` főágát átugorja, és az `if` utáni első utasításnál folytatódik
4. Ha a program egy `exit(0);` függvényhívással találkozik
   - ez egy speciális függvény, ami azonnal kilép a programból
   - a nem nulla argumentummal hibát lehet jelezni az operációs rendszer felé

Többágú elágazás egymásba ágyazott `if`-ekkel

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

int main(int argc, char* argv[]) {
  if (argc < 4) {
    printf("Not enough arguments.\n");
    return -1;
  }

  double a = atof(argv[1]);
  double b = atof(argv[2]);
  double c = atof(argv[3]);
  double d = b * b - 4 * a * c;

  if (d < 0) {
    printf("No real solution.\n");
    return -1;
  } else if (d == 0) {
    double r = -b / 2 / a;
    printf("r = %f\n", r);
  } else {
    d = sqrt(d);
    double r1 = (-b - d) / 2 / a;
    double r2 = (-b + d) / 2 / a;
    printf("r1 = %f, r2 = %f\n", r1, r2);
  }

  return 0;
}
```

- Az `if(...)` és az `else` után egy-egy utasításblokk jön

Többágú elágazás egymásba ágyazott `if`-ekkel

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

int main(int argc, char* argv[]) {
  if (argc < 4) {
    printf("Not enough arguments.\n");
    return -1;
  }

  double a = atof(argv[1]);
  double b = atof(argv[2]);
  double c = atof(argv[3]);
  double d = b * b - 4 * a * c;

  if (d < 0) {
    printf("No real solution.\n");
    return -1;
  } else if (d == 0) {
    double r = -b / 2 / a;
    printf("r = %f\n", r);
  } else {
    d = sqrt(d);
    double r1 = (-b - d) / 2 / a;
    double r2 = (-b + d) / 2 / a;
    printf("r1 = %f, r2 = %f\n", r1, r2);
  }

  return 0;
}
```

- Az `if(...)` és az `else` után egy-egy utasításblokk jön
  - de lehet egy másik `if` utasítás is

Saját függvények definiálása

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

double discr(double a, double b, double c) {
  return b * b - 4 * a * c;
}

void roots(double a, double b, double c) {
  double d = discr(a, b, c);
  if (d < 0) {
    printf("No real solution.\n");
    exit(-1);
  } else if (d == 0) {
    double r = -b / 2 / a;
    printf("r = %f\n", r);
  } else {
    d = sqrt(d);
    double r1 = (-b - d) / 2 / a;
    double r2 = (-b + d) / 2 / a;
    printf("r1 = %f, r2 = %f\n", r1, r2);
  }
}

int main(int argc, char* argv[]) {
  if (argc < 4) {
    printf("Not enough arguments.\n");
    exit(-1);
  }
  double a = atof(argv[1]);
  double b = atof(argv[2]);
  double c = atof(argv[3]);
  roots(a, b, c);
  return 0;
}
```

- Bizonyos feladatok többször is ismétlődhetnek a program futása során

Saját függvények definiálása

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

double discr(double a, double b, double c) {
  return b * b - 4 * a * c;
}

void roots(double a, double b, double c) {
  double d = discr(a, b, c);
  if (d < 0) {
    printf("No real solution.\n");
    exit(-1);
  } else if (d == 0) {
    double r = -b / 2 / a;
    printf("r = %f\n", r);
  } else {
    d = sqrt(d);
    double r1 = (-b - d) / 2 / a;
    double r2 = (-b + d) / 2 / a;
    printf("r1 = %f, r2 = %f\n", r1, r2);
  }
}

int main(int argc, char* argv[]) {
  if (argc < 4) {
    printf("Not enough arguments.\n");
    exit(-1);
  }
  double a = atof(argv[1]);
  double b = atof(argv[2]);
  double c = atof(argv[3]);
  roots(a, b, c);
  return 0;
}
```

- Bizonyos feladatok többször is ismétlődhetnek a program futása során
- Vagy csak érdemes logikailag leválasztani a program egy részét

Saját függvények definiálása

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

double discr(double a, double b, double c) {
  return b * b - 4 * a * c;
}

void roots(double a, double b, double c) {
  double d = discr(a, b, c);
  if (d < 0) {
    printf("No real solution.\n");
    exit(-1);
  } else if (d == 0) {
    double r = -b / 2 / a;
    printf("r = %f\n", r);
  } else {
    d = sqrt(d);
    double r1 = (-b - d) / 2 / a;
    double r2 = (-b + d) / 2 / a;
    printf("r1 = %f, r2 = %f\n", r1, r2);
  }
}

int main(int argc, char* argv[]) {
  if (argc < 4) {
    printf("Not enough arguments.\n");
    exit(-1);
  }
  double a = atof(argv[1]);
  double b = atof(argv[2]);
  double c = atof(argv[3]);
  roots(a, b, c);
  return 0;
}
```

- Bizonyos feladatok többször is ismétlődhetnek a program futása során
- Vagy csak érdemes logikailag leválasztani a program egy részét
- Ilyenkor saját függvényt definiálunk
  - már a `main` is eleve egy függvény volt

Saját függvények definiálása

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

double discr(double a, double b, double c) {
  return b * b - 4 * a * c;
}

void roots(double a, double b, double c) {
  double d = discr(a, b, c);
  if (d < 0) {
    printf("No real solution.\n");
    exit(-1);
  } else if (d == 0) {
    double r = -b / 2 / a;
    printf("r = %f\n", r);
  } else {
    d = sqrt(d);
    double r1 = (-b - d) / 2 / a;
    double r2 = (-b + d) / 2 / a;
    printf("r1 = %f, r2 = %f\n", r1, r2);
  }
}

int main(int argc, char* argv[]) {
  if (argc < 4) {
    printf("Not enough arguments.\n");
    exit(-1);
  }
  double a = atof(argv[1]);
  double b = atof(argv[2]);
  double c = atof(argv[3]);
  roots(a, b, c);
  return 0;
}
```

- Bizonyos feladatok többször is ismétlődhetnek a program futása során
- Vagy csak érdemes logikailag leválasztani a program egy részét
- Ilyenkor saját függvényt definiálunk
  - már a `main` is eleve egy függvény volt
- A függvényt a visszatérési értéke, a neve és a paramétereinek típusa azonosítja
- A függvény definíciója a `{ ... }` részbe kerül
  - `return`: ha csak vissza kell térni
  - `exit`: csak hibakezeléskor!

Saját függvények definiálása

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

double discr(double a, double b, double c) {
  return b * b - 4 * a * c;
}

void roots(double a, double b, double c) {
  double d = discr(a, b, c);
  if (d < 0) {
    printf("No real solution.\n");
    exit(-1);
  } else if (d == 0) {
    double r = -b / 2 / a;
    printf("r = %f\n", r);
  } else {
    d = sqrt(d);
    double r1 = (-b - d) / 2 / a;
    double r2 = (-b + d) / 2 / a;
    printf("r1 = %f, r2 = %f\n", r1, r2);
  }
}

int main(int argc, char* argv[]) {
  if (argc < 4) {
    printf("Not enough arguments.\n");
    exit(-1);
  }
  double a = atof(argv[1]);
  double b = atof(argv[2]);
  double c = atof(argv[3]);
  roots(a, b, c);
  return 0;
}
```

- Bizonyos feladatok többször is ismétlődhetnek a program futása során
- Vagy csak érdemes logikailag leválasztani a program egy részét
- Ilyenkor saját függvényt definiálunk
  - már a `main` is eleve egy függvény volt
- A függvényt a visszatérési értéke, a neve és a paramétereinek típusa azonosítja
- A függvény definíciója a `{ ... }` részbe kerül
  - `return`: ha csak vissza kell térni
  - `exit`: csak hibakezeléskor!
- Sok-sok függvényhívás

Függvény definiálása vs. függvényhívás

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  double discr(double a, double b, double c) {
6    return b * b - 4 * a * c;
7  }
8
9  void roots(double a, double b, double c) {
10   double d = discr(a, b, c);
11   if (d < 0) {
12     printf("No real solution.\n");
13     exit(-1);
14   } else if (d == 0) {
15     double r = -b / 2 / a;
16     printf("r = %f\n", r);
17   } else {
18     d = sqrt(d);
19     double r1 = (-b - d) / 2 / a;
20     double r2 = (-b + d) / 2 / a;
21     printf("r1 = %f, r2 = %f\n", r1, r2);
22   }
23 }
24
25 int main(int argc, char* argv[]) {
26   if (argc < 4) {
27     printf("Not enough arguments.\n");
28     exit(-1);
29   }
30   double a = atof(argv[1]);
31   double b = atof(argv[2]);
32   double c = atof(argv[3]);
33   roots(a, b, c);
34   return 0;
35 }
```

Mindig `függvénynév(...)` alak

- Függvény definiálása
  - visszatérési érték típusa a név előtt
  - paraméterek típusa a nevek előtt

Függvény definiálása vs. függvényhívás

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   double discr(double a, double b, double c) {
6     return b * b - 4 * a * c;
7   }
8
9   void roots(double a, double b, double c) {
10    double d = discr(a, b, c);
11    if (d < 0) {
12      printf("No real solution.\n");
13      exit(-1);
14    } else if (d == 0) {
15      double r = -b / 2 / a;
16      printf("r = %f\n", r);
17    } else {
18      d = sqrt(d);
19      double r1 = (-b - d) / 2 / a;
20      double r2 = (-b + d) / 2 / a;
21      printf("r1 = %f, r2 = %f\n", r1, r2);
22    }
23  }
24
25  int main(int argc, char* argv[]) {
26    if (argc < 4) {
27      printf("Not enough arguments.\n");
28      exit(-1);
29    }
30    double a = atof(argv[1]);
31    double b = atof(argv[2]);
32    double c = atof(argv[3]);
33    roots(a, b, c);
34    return 0;
35  }
```

Mindig `függvénynév(...)` alak

- Függvény definiálása
  - visszatérési érték típusa a név előtt
  - paraméterek típusa a nevek előtt
- Függvény hívása
  - sehol sincs kiírva a típus
  - a paraméter kifejezés is lehet

Összehasonlítás: Python

```
1   def discr(a,b,c):
2       return b * b - 4 * a * c
3
4   def roots(a,b,c):
5       d = discr(a, b, c)
6       if d < 0:
7           print("No real solution.")
8       elif d == 0:
9           r = -b / 2 / a
10          print("r = ", r)
11      else:
12          d = sqrt(d)
13          r1 = (-b - d) / 2 / a
14          r2 = (-b + d) / 2 / a
15          print("r1 = ", r1, ",   r2 = ",r2);
16
17  roots(1.1,3.3,2.2)
```

- Függvény definiálása
  - def utasítással
  - visszatérési érték típusát nem kell megadni

Összehasonlítás: Python

```
1   def discr(a,b,c):
2       return b * b - 4 * a * c
3
4   def roots(a,b,c):
5       d = discr(a, b, c)
6       if d < 0:
7           print("No real solution.")
8       elif d == 0:
9           r = -b / 2 / a
10          print("r = ", r)
11      else:
12          d = sqrt(d)
13          r1 = (-b - d) / 2 / a
14          r2 = (-b + d) / 2 / a
15          print("r1 = ", r1, ",   r2 = ",r2);
16
17  roots(1.1,3.3,2.2)
```

- Függvény definiálása
  - `def` utasítással
  - visszatérési érték típusát nem kell megadni
- Programblokkok nincsennek `{ ... }` -ben

Összehasonlítás: Python

```
1  def discr(a,b,c):
2      return b * b - 4 * a * c
3
4  def roots(a,b,c):
5      d = discr(a, b, c)
6      if d < 0:
7          print("No real solution.")
8      elif d == 0:
9          r = -b / 2 / a
10         print("r = ", r)
11     else:
12         d = sqrt(d)
13         r1 = (-b - d) / 2 / a
14         r2 = (-b + d) / 2 / a
15         print("r1 = ", r1, ",   r2 = ",r2);
16
17 roots(1.1,3.3,2.2)
```

- Függvény definiálása
  - `def` utasítással
  - visszatérési érték típusát nem kell megadni
- Programblokkok nincsennek `{ ... }` -ben
- a kiértékelendő feltételt nem kell `(...)` -be tenni

Összehasonlítás: Python

```
1   def discr(a,b,c):
2       return b * b - 4 * a * c
3
4   def roots(a,b,c):
5       d = discr(a, b, c)
6       if d < 0:
7           print("No real solution.")
8       elif d == 0:
9           r = -b / 2 / a
10          print("r = ", r)
11      else:
12          d = sqrt(d)
13          r1 = (-b - d) / 2 / a
14          r2 = (-b + d) / 2 / a
15          print("r1 = ", r1, ",   r2 = ",r2);
16
17  roots(1.1,3.3,2.2)
```

- Függvény definiálása
  - `def` utasítással
  - visszatérési érték típusát nem kell megadni
- Programblokkok nincsennek `{ ... }` -ben
- a kiértékelendő feltételt nem kell `(...)` -be tenni
- `else if` lerövidítve `elif`

A végrehajtási sorrend függvényhíváskor

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

double discr(double a, double b, double c) {
  return b * b - 4 * a * c;
}

void roots(double a, double b, double c) {
  double d = discr(a, b, c);
  if (d < 0) {
    printf("No real solution.\n");
    exit(-1);
  } else if (d == 0) {
    double r = -b / 2 / a;
    printf("r = %f\n", r);
  } else {
    d = sqrt(d);
    double r1 = (-b - d) / 2 / a;
    double r2 = (-b + d) / 2 / a;
    printf("r1 = %f, r2 = %f\n", r1, r2);
  }
}

int main(int argc, char* argv[]) {
  if (argc < 4) {
    printf("Not enough arguments.\n");
    exit(-1);
  }
  double a = atof(argv[1]);
  double b = atof(argv[2]);
  double c = atof(argv[3]);
  roots(a, b, c);
  return 0;
}
```

1. A program egy **függvényhíváshoz** ér
   - ekkor sorban kiértékelődik az összes paraméter értéke (ha kifejezések)
   - majd a program futása a meghívott függvény első során folytatódik
2. Ha a program egy `return` utasításhoz ér, akkor a függvény **visszatér**
   - ha van visszatérési érték, akkor azt a `return` utasításnak kell beállítania
   - a visszatérés utána a program a **függvényhívást követő** utasítástól folytatódik
3. Ha a program nem talál `return` utasítást, akkor a függvény az utolsó utasítás után tér vissza
4. A függvényhívások *szinte* tetszőleges mértékben egymásba ágyazhatók
   - két függvény hívhatja egymást kölcsönösen
   - sőt, egy függvény akár saját magát is hívhatja

Függvények előre deklarálása

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

double discr(double a, double b, double c);
void roots(double a, double b, double c);

int main(int argc, char* argv[]) {
  if (argc < 4) {
    printf("Not enough arguments.\n");
    exit(-1);
  }
  double a = atof(argv[1]);
  double b = atof(argv[2]);
  double c = atof(argv[3]);
  roots(a, b, c);
  return 0;
}

void roots(double a, double b, double c) {
  double d = discr(a, b, c);
  if (d < 0) {
    printf("No real solution.\n");
    exit(-1);
  } else if (d == 0) {
    double r = -b / 2 / a;
    printf("r = %f\n", r);
  } else {
    d = sqrt(d);
    double r1 = (-b - d) / 2 / a;
    double r2 = (-b + d) / 2 / a;
    printf("r1 = %f, r2 = %f\n", r1, r2);
  }
}

double discr(double a, double b, double c) {
  return b * b - 4 * a * c;
```

A függvényt korábban kell deklarálni, mint ahogyan hivatkozunk rá.

- ilyenkor csak a **szignatúrát** írjuk ki
- a sort ; zárja

Függvények előre deklarálása

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   double discr(double a, double b, double c);
6   void roots(double a, double b, double c);
7
8   int main(int argc, char* argv[]) {
9     if (argc < 4) {
10      printf("Not enough arguments.\n");
11      exit(-1);
12    }
13    double a = atof(argv[1]);
14    double b = atof(argv[2]);
15    double c = atof(argv[3]);
16    roots(a, b, c);
17    return 0;
18  }
19
20  void roots(double a, double b, double c) {
21    double d = discr(a, b, c);
22    if (d < 0) {
23      printf("No real solution.\n");
24      exit(-1);
25    } else if (d == 0) {
26      double r = -b / 2 / a;
27      printf("r = %f\n", r);
28    } else {
29      d = sqrt(d);
30      double r1 = (-b - d) / 2 / a;
31      double r2 = (-b + d) / 2 / a;
32      printf("r1 = %f, r2 = %f\n", r1, r2);
33    }
34  }
35
36  double discr(double a, double b, double c) {
37    return b * b - 4 * a * c;
```

A függvényt korábban kell deklarálni, mint ahogyan hivatkozunk rá.

- ilyenkor csak a **szignatúrát** írjuk ki
- a sort ; zárja

A függvény definíciója már lehet később, mint az első hívás

A második program: Fibonacci-számsorozat

A sorozatot egy **iterációs formulával** adjuk meg

$$\begin{aligned} f_1 &= 1 \\ f_2 &= 1 \\ f_n &= f_{n-2} + f_{n-1} \end{aligned}$$

Adatmodell:

- a sorozatnak mindig csak az előző két elemét kell tárolni $\Rightarrow$ elegendő két `int` típusú változó

Iteratív algoritmus:

- egy paraméter, hogy a sorozat hány elemét kell előállítani
- egy iterációs lépés, ami a sorozat előző két eleméből előállítja a következőt
- egy ciklus, ami az iterációs lépést ismétli

A második program: Fibonacci-számsorozat

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void fibo(int n) {
5      int a = 0, b = 1;
6      int i = 0;
7      while (i < n) {
8          int c = a + b;
9          a = b;
10         b = c;
11         printf("%d\n", a);
12         i++;
13     }
14 }
15
16 int main(int argc, char* argv[])
17 {
18     // Process command-line arguments
19     if (argc < 2) {
20         printf("Missing argument.\n");
21         exit(-1);
22     }
23     int n = atof(argv[1]);
24     fibo(n);
25     return 0;
26 }
```

- a `main` függvény csak a parancssort dolgozza fel, majd meghívja az algoritmust

A második program: Fibonacci-számsorozat

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void fibo(int n) {
5      int a = 0, b = 1;
6      int i = 0;
7      while (i < n) {
8          int c = a + b;
9          a = b;
10         b = c;
11         printf("%d\n", a);
12         i++;
13     }
14 }
15
16 int main(int argc, char* argv[])
17 {
18     // Process command-line arguments
19     if (argc < 2) {
20         printf("Missing argument.\n");
21         exit(-1);
22     }
23     int n = atof(argv[1]);
24     fibo(n);
25     return 0;
26 }
```

- a `main` függvény csak a parancssort dolgozza fel, majd meghívja az algoritmust
- használhatjuk az `atoi` vagy az `atof` függvényt, az eredmény típusát a paraméter deklaráció adja meg

A második program: Fibonacci-számsorozat

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void fibo(int n) {
5      int a = 0, b = 1;
6      int i = 0;
7      while (i < n) {
8          int c = a + b;
9          a = b;
10         b = c;
11         printf("%d\n", a);
12         i++;
13     }
14 }
15
16 int main(int argc, char* argv[])
17 {
18     // Process command-line arguments
19     if (argc < 2) {
20         printf("Missing argument.\n");
21         exit(-1);
22     }
23     int n = atof(argv[1]);
24     fibo(n);
25     return 0;
26 }
```

- a `main` függvény csak a parancssort dolgozza fel, majd meghívja az algoritmust
- használhatjuk az `atoi` vagy az `atof` függvényt, az eredmény típusát a paraméter deklaráció adja meg
- a függvényben lokális változókat deklarálunk
  - ezek csak a `fibo` függvény számára láthatók
  - `a` és `b` a sorozat két elemét tárolja
  - `i` egy **ciklusváltozó**

A második program: Fibonacci-számsorozat

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void fibo(int n) {
5      int a = 0, b = 1;
6      int i = 0;
7      while (i < n) {
8          int c = a + b;
9          a = b;
10         b = c;
11         printf("%d\n", a);
12         i++;
13     }
14 }
15
16 int main(int argc, char* argv[])
17 {
18     // Process command-line arguments
19     if (argc < 2) {
20         printf("Missing argument.\n");
21         exit(-1);
22     }
23     int n = atof(argv[1]);
24     fibo(n);
25     return 0;
26 }
```

- a `main` függvény csak a parancssort dolgozza fel, majd meghívja az algoritmust
- használhatjuk az `atoi` vagy az `atof` függvényt, az eredmény típusát a paraméter deklaráció adja meg
- a függvényben lokális változókat deklarálunk
  - ezek csak a `fibo` függvény számára láthatók
  - `a` és `b` a sorozat két elemét tárolja
  - `i` egy **ciklusváltozó**
- az iterációs szabályhoz be kell vezetni egy segédváltozót, különben felülírnánk a sorozat valamelyik elemét

A második program: Fibonacci-számsorozat

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void fibo(int n) {
5      int a = 0, b = 1;
6      int i = 0;
7      while (i < n) {
8          int c = a + b;
9          a = b;
10         b = c;
11         printf("%d\n", a);
12         i++;
13     }
14 }
15
16 int main(int argc, char* argv[])
17 {
18     // Process command-line arguments
19     if (argc < 2) {
20         printf("Missing argument.\n");
21         exit(-1);
22     }
23     int n = atof(argv[1]);
24     fibo(n);
25     return 0;
26 }
```

- a `main` függvény csak a parancssort dolgozza fel, majd meghívja az algoritmust
- használhatjuk az `atoi` vagy az `atof` függvényt, az eredmény típusát a paraméter deklaráció adja meg
- a függvényben lokális változókat deklarálunk
  - ezek csak a `fibo` függvény számára láthatók
  - `a` és `b` a sorozat két elemét tárolja
  - `i` egy **ciklusváltozó**
- az iterációs szabályhoz be kell vezetni egy segédváltozót, különben felülírnánk a sorozat valamelyik elemét
- az `i=i+1` léptetés tömören `i++`

A második program: Fibonacci-számsorozat

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void fibo(int n) {
5      int a = 0, b = 1;
6      int i = 0;
7      while (i < n) {
8          int c = a + b;
9          a = b;
10         b = c;
11         printf("%d\n", a);
12         i++;
13     }
14 }
15
16 int main(int argc, char* argv[])
17 {
18     // Process command-line arguments
19     if (argc < 2) {
20         printf("Missing argument.\n");
21         exit(-1);
22     }
23     int n = atof(argv[1]);
24     fibo(n);
25     return 0;
26 }
```

- a `main` függvény csak a parancssort dolgozza fel, majd meghívja az algoritmust
- használhatjuk az `atoi` vagy az `atof` függvényt, az eredmény típusát a paraméter deklaráció adja meg
- a függvényben lokális változókat deklarálunk
  - ezek csak a `fibo` függvény számára láthatók
  - `a` és `b` a sorozat két elemét tárolja
  - `i` egy **ciklusváltozó**
- az iterációs szabályhoz be kell vezetni egy segédváltozót, különben felülírnánk a sorozat valamelyik elemét
- az `i=i+1` léptetés tömören `i++`
- a `while(...)` addig ismétli az őt követő utasítást, amíg a feltétel igaznak értékelődik ki

Ciklusok

Ha a programnak többször meg kell ismételnie néhány utasítást, akkor ciklusokat írunk.

A C nyelvben összesen háromféle ciklus van

- `while(i < n) { }` – **elöltesztelős ciklus**, zárójelben az ismétlés feltétele
  - a feltétel egy kifejezés, ugyanazok igazak rá, mint az if esetében
  - a feltétel a **ciklusmag** lefutása *előtt* értékelődik ki
- `do { } while (i < n);` – **hátul tesztelős ciklus**
  - a feltétel a **ciklusmag** lefutása *után* értékelődik ki
  - a ciklusmag egyszer mindenképpen lefut!
  - figyelem! a végére kell `;`
- `for (...) { }` – iteratív ciklus

for ciklus

Nagyon gyakori, hogy egy ciklusváltozót lépésenként kell növelni (csökkenteni)

- while ciklus esetén túl sokat kell gépelni
- a fordító nehezen tudja kitalálni, hogy mi a ciklusváltozó

```
1 int i = 0;
2 while (i < 100) {
3   // do something
4   i++;
5 }
```

Az ennek megfelelő **for** ciklus (csak C99 szabvány!)

- két sorral tömörebb
- a ciklusváltozót nem tudjuk véletlenül a cikluson kívül használni

```
1 for (int i = 0; i < 100; i++) {
2   // do something
3 }
```

A `for` ciklus részletes szintaktikája

```
1  for ( [init] ; [condition] ; [increment] ) {
2    // do something
3  }
```

A két zárójelen belüli pontosvessző kötelező

Zárójelen belüli tagok:

1. **init**: utasítás, ami az első iteráció előtt hajtódik végre
   - jellemzően a ciklusváltozó inicializálása
   - de tetszőleges utasítás lehet

A `for` ciklus részletes szintaktikája

```
1  for ( [init] ; [condition] ; [increment] ) {
2    // do something
3  }
```

A két zárójelen belüli pontosvessző kötelező

Zárójelen belüli tagok:

1. **init**: utasítás, ami az első iteráció előtt hajtódik végre
   - jellemzően a ciklusváltozó inicializálása
   - de tetszőleges utasítás lehet
2. **condition**: kifejezés, minden iteráció előtt kiértékelődik
   - ha értéke != 0, akkor a ciklusmag lefut
   - elöltesztelős ciklus, mint a `while`

A `for` ciklus részletes szintaktikája

```
1  for ( [init] ; [condition] ; [increment] ) {
2    // do something
3  }
```

A két zárójelen belüli pontosvessző kötelező

Zárójelen belüli tagok:

1. **init**: utasítás, ami az első iteráció előtt hajtódik végre
   - jellemzően a ciklusváltozó inicializálása
   - de tetszőleges utasítás lehet
2. **condition**: kifejezés, minden iteráció előtt kiértékelődik
   - ha értéke != 0, akkor a ciklusmag lefut
   - elöltesztelős ciklus, mint a `while`
3. **increment**: utasítás, ami a ciklusmag legvégén hajtódik végre
   - tipikusan a ciklusváltozó növelésére, csökkentésére
   - csak akkor fut le, ha a feltétel teljesült, és a ciklusmag is lefutott

Néhány példa for ciklusra

```
1  for (int i = 0; i < 100; i++) {
2    for (int j = 0; j < 20; j++) {
3      // do something
4    }
5  }
```

```
1  for (int i = 0; i < 100; i++) {
2    for (int j = 0; j < i; j ++) {
3      // do something
4    }
5  }
```

```
1  for (int i = 99; i >= 0; i--) {
2    // do something
3  }
```

Pár tipp

Egy nagyon fontos szabály

- C nyelvben a ciklusokat mindig 0-tól futtatjuk $n-1$-ig
- ez a memóriacímzés miatt van így

Gyakorlófeladat

- Írjuk át a Fibonacci-sorozatot előállító programot `for` ciklusra és hátultesztelős `do – while` ciklusra!

A `break` és a `continue` utasítás

A `break` utasítás kilép a (legbelső) ciklusból

- általában valamilyen feltétel teljesülése esetén
- nem iteráció végén, hanem speciális esetekben használjuk
- `for` ciklusnál is működik, de többnyire `while`-nál

A `break` és a `continue` utasítás

A `break` utasítás kilép a (legbelső) ciklusból

- általában valamilyen feltétel teljesülése esetén
- nem iteráció végén, hanem speciális esetekben használjuk
- `for` ciklusnál is működik, de többnyire `while`-nál

A `continue` utasítás átugorja a ciklusmag hátralevő részét

- ezt is feltétellel együtt használjuk
- a ciklus maga folytatódik, csak a ciklusmag nem fut tovább
- nem az iteráció végén, hanem speciális esetekben használjuk

Figyelem!

- `for` ciklusnál ilyenkor lefut az inkrementáló rész (`i++`)
- `while` ciklusnál ügyelni kell, hogy ne legyen végtelen ciklus

Első házi feladat

Készítsünk programot, ami kilistázza a püthagoraszi számhármasokat, tehát az $a^2 + b^2 = c^2$ feltételnek eleget tevő a, b, c pozitív egész számhármasokat az alábbiak szerint!

1. Egy megadott n számig kilistáz minden $a, b < c$, $c \leq n$ feltétel szerinti számhármast, mindegyiknél ellenőrzi a fenti feltételt, és megjelöli azokat, amiknél az teljesül (A)
2. Az első n darab püthagoraszi számhármast írja ki, úgy értve, hogy az a, b, c számhármasok egymás közt felcserélt a, b értékekkel is előfordulhatnak. (A)
3. Az első n darab püthagoraszi hármast írja ki úgy, hogy a számhármasok b) pont szerinti ismétlődése nem megengedett. (T)
4. Az első n darab püthagoraszi hármast írja ki úgy, hogy a számhármasok nem lehetnek egymás számszorosai sem, azaz pl. (3, 4, 5) után (8, 6, 10) nem megengedett. (T)

A program paraméterét parancssori argumentumként adjuk meg: ha az első paraméter '1', az 1. feladatrész, ha '2' a 2. feladatrész megoldását adja vissza, stb. A második paraméter pedig legyen n értéke.
Beadási határidő: 2019. szeptember 22, 23.59