2. gyakorlat

Egymásba ágyazott ciklusok, tömbök, beolvasás fájlból

avagy a szorgalmi feladatok megoldása és ami az 1. beadandó feladathoz kell

Az előadások, beleértve az ehetit, videóként is elérhetők az elméleti oldalon.

Fibonacci számok

Készítsünk programot, ami

- adott n-hez kiszámítja az első n Fibonacci számot!
- avagy a 0-diktól az n-dikig (0, 1, 1, 2, 3, 5, ...)

In [ ]:
```c
#include <stdio.h>
#include <stdlib.h>     // Így lehet egysoros kommentet csinálni

void fibo(int n)        // Mivel kiíratjuk a számokat, a függvénynek nincs
                        // visszatérési értéke. Ezt jelzi a void,
                        // összhangban az üres return; utasításokkal.
{
    if (n<0) return;          // Ha nem kértünk egyet sem visszatér.
    printf("%d %d\n", 0, 0);  // n legalább 0, így kiírjuk a 0-dikat.
    if (n==0) return;         // Ha n=0 nincs több teendőnk.
    int a = 0, b = 1;         // Előkészítés.
    int i = 1;                // i jelzi majd, hogy hanyadiknál tartunk
    printf("%d %d\n", 1, 1);  // az első írjuk ki
    while (i < n) {           // Ha nem értük el az n-diket ...
        int c = a + b;
        a = b;
        b = c;
        i++;
        printf("%d %d\n", i, c);  // ... kiírjuk az i-diket
    }
    return;
}

int main()
{
    fibo(5);

    return 0;
}
```

Pitagoraszi számhármasok

Készítsünk programot, ami kilistázza a pitagoraszi számhármasokat, azaz az $a^2 + b^2 = c^2$ feltételt teljesítő pozitív egész számokból álló hármasokat!

a) változat:

- Egy megadott n számig kilistáz minden a,b,c számhármast, amire a<c, b<c és c<=n teljesül,
- és megjelöli azokat, amiknél az $a^2 + b^2 = c^2$ teljesül.

In [ ]:
```
// először csak az a,b értékeket variáljuk

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

int main()
{
    int a,b,c;
    int n=5;

    for (a=1; a<=n; a++)
    {
         for (b=1; b<=n; b++)     // minden 'a' érték mellett b végigmegy
         {                        // a lehetséges értékeken
            printf("%d %d\n",a,b);
         }
    }

    return 0;
}
```

In [ ]:
```
// Ebbe az a<c és b<c feltétel nehezen építhető bele.
// Másrészt így furcsa lenne a sorrend, pl. (1,4,5) megelőzné a (2,1,3)-at.

// Jobb, ha a külső ciklus megy c-re:

    for (c=1; c<=n; c++)
    {
         for (a=1; a<c; a++)
         {
            for (b=1; b<c; b++)
            {
                printf("%d %d %d\n",a,b,c);
            }
         }
    }
```

In [ ]:
```
// Vizsgáljuk meg, hogy Pitagoraszi-e és jelöljük meg:
// legbelső blokkot bővítsük így:
                printf("%d %d %d\n",a,b,c);
                if (a*a+b*b==c*c) printf("Ez Pitagoraszi!\n",a,b,c);
// vagy
                printf("%d %d %d",a,b,c);
                if (a*a+b*b==c*c) printf("  Pitagoraszi!");
                printf("\n");
// vagy
                if (a*a+b*b==c*c) printf("%d %d %d Pitagoraszi!\n",a,b,c);
                else printf("%d %d %d\n",a,b,c);
```

In [ ]:

```
// Tegyük át függvénybe:

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

void pita_a(int n)
{
    int a,b,c;
    for (c=1; c<=n; c++)
    {
         for (a=1; a<c; a++)
         {
            for (b=1; b<c; b++)
            {
                if (a*a+b*b==c*c) printf("%d %d %d Pitagoraszi!\n",a,b,c);
                else printf("%d %d %d\n",a,b,c);
            }
         }
    }
}

int main()
{
    int n=5;

    pita_a(n);

    return 0;
}
```

b) feladatrész:

Az első n darab pitagoraszi számhármast írja ki, úgy értve, hogy az (a,b,c) számhármasok egymás közt felcserélt a,b értékekkel is előfordulhatnak.

In [ ]:
```
// Másoljuk le a függvényünk a b) feladatrészhez, átnevezve,
// írjuk ki csak a jókat.
// Mivel akár az a, akár b, c növelésekor elérhetjük a kívánt darabszámot,
// mindegyik ciklusnál figyelni kell azt.
// Bonyolultabb lesz a leállási feltétel, ezért írjuk át while ciklusra,
// egyelőre meghagyva az érték szerinti korlátozást n-nel:

void pita_b(int n)
{
    int a,b,c;
    c=1;
    while (c<=n)
    {
        a=1;         // Ne felejtsünk el kezdőértéket adni a c,a,b-nek!
        while (a<c)
        {
            b=1;
            while (b<c)
            {
                if (a*a+b*b==c*c) printf("%d %d %d  Pitagoraszi!\n",a,b,c);
                b++;
            }
            a++;  // A növelő utasítást se felejtsük ki a három változóhoz!
        }
        c++;
    }
}

// Próbáljuk ki n=20-szal!
// Figyeljük meg, hogy egyes soroknál a c nő, másoknál az a, vagy a b.
// Ezért a darabszám szerinti leállítás megírásához a,b és c növelése után
// is ellenőrizni kell majd, hogy elértük-e a kívánt darabszámot.
```

In [ ]:
```
// újabb változóval követjük a darabszámot és minden ciklusfeltételben
// ellenőrizzük, hogy elértük-e az n-et

void pita_b(int n)
{
    int a,b,c;
    c=1;
    int k=0;
    while (k<n)
    {
        a=1;
        while ((a<c) && (k<n))
        {
            b=1;
            while ((b<c) && (k<n))
            {
                if (a*a+b*b==c*c)
                {
                    printf("%d %d %d  Pitagoraszi!\n",a,b,c);
                    k++;
                }
                b++;
            }
            a++;
        }
        c++;
    }
}
```

c) feladatrész

Az első n darab pitagoraszi hármast írja ki úgy, hogy a számhármasok b) pont szerinti ismétlődése nem megengedett!

In [ ]:
```
// Újabb másolással megcsináljuk a c változatot.
// Hogy a felcserélt a,b értekkel ismétlődés nem megengedett,
// azt kiköthetjük úgy, hogy a<=b. (Meggondolva, hogy a és b Pitagoraszi
// számhármasban sosem egyenlő, a<b is használható.)
// Csak egy helyen kell módosítani:

void pita_c(int n)
{
    int a,b,c;
    c=1;
    int k=0;
    while (k<n)
    {
        a=1;
        while ((a<c) && (k<n))
        {
            b=a;                    // b kezdőértékét módosítottuk
                                    // lehet b=a+1 is az erősebb feltételhez
            while ((b<c) && (k<n))
            {
                if (a*a+b*b==c*c)
                {
                    printf("%d %d %d  Pitagoraszi!\n",a,b,c);
                    k++;
                }
                b++;
            }
            a++;
        }
        c++;
    }
}
```

d) részt legközelebb beszéljük meg, addig lehet tovább próbálgatni

Input

In [ ]:
```
// a terminálból, vagy esetleg '|' ill. '<' jelek segítségével átadott
// adatokból (azaz a standard inputról)
// így olvashatunk be egész típusú változóba:

    int n;
    printf("n ");
    scanf("%d",&n);    // format string kell és a változó címe
    printf("%d\n",n);  // ellenőrzésül mindjárt ki is írjuk

// Próbáljuk ki az eddigi programunk main részében!
// Miért a változó címe kell? C-ben a függvények argumentumként fv(n) módon
// megadott változóknak csak az értékét kaphatják meg. Nem kapják meg a
// címét, azt a pozíciót, ahol a memóriában tárolódik.
// (Írhatunk oda számot, kifejezést is, amikor nem lenne értelme az
// argumentum címének. A c egyszerűségével, következetességével összhangban
// változó átadásakor sem kaphatja meg annak a címét.) Így viszont egy
// függvény fv(n) módon hívva nem tud a változóba írni. Ezért a scanf-nek a
// változó címét kell átadnunk, és azt a scanf is címként kezeli. Az & jel
// szolgál arra, hogy az n változóról leolvassa a címét, ahol tárolódik.
```

In [ ]:
```
// Lebegőpontos számok beolvasása

// Mivel a változó címét kell megadnunk, annak az az érdekes következménye,
// hogy a scanf nem tudja, milyen típusú változó az. Így valós szám esetén
// azt is meg kell adnunk, hogy float v. double típusú-e. Nem elég a %f
// típus jelzés, mint a prinf-nél, hanem float típus esetén kell %f, double
// esetén pedig %lf használandó.
// Vigyázzunk, ez a printf-nél nincs így, ott mindkét esetben %f kell.

// Próbáljuk ezt ki egy új projektben:

// pl. "vektorok" néven hozzuk létre, mert tömböket is fogunk létrehozni!

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

int main()
{
    float s;
    double x;

    printf("s ");
    scanf("%f",&s);
    printf("%f\n",s);  // Beolvasunk egy float változóba, és kiírjuk.

    printf("x ");
    scanf("%lf",&x);
    printf("%f\n",x);  // Beolvasunk egy double változóba, és kiírjuk.

    printf("%f\n",(x-s)*1000);  // Így látjuk is, hogy a float pontatlanabb.

    return 0;
}
```

Vektorok, tömbök létrehozása és használata

In [ ]:
```
// - pl. a Fibonacci számok v. más adatsorok (pythonban: list és array)
// - tömbként viselkedik a string is, erről később lesz szó

// Kétféleképp hozhatunk létre egyindexes tömböt:

// A veremben:

    double xx[4];

  // Értéket adni és kiolvasni egyszerű:
    xx[0]=1.1;              // index: 0...3
    printf("%f\n",xx[1]);

// A halom területen pedig:

    int n=100;
    double *v;  // pointer változó létrehozása, ami memóriacímet tárolhat
    v = (double*)malloc(n*sizeof(double));  // Külön kell gondoskodnunk a
          // helyfoglalásról. Zárójelben a lefoglalandó byteok száma.
          // A malloc visszaadja a lefoglalt terület kezdőpontjának a címét.

  // Ezt felhasználva ugyanúgy tudunk adatokat tárolni a lefoglalt terület
  // egyes celláiban, mint a veremben tárolt esetben:
    v[2]=2.2;               // index: 0...99
    printf("%f\n",v[2]);
    free(v);                // ha már nem használjuk a tömb területét,
                            // meg kell szüntetni a foglalást

// Mi a különbség a kettő között? Ennek megértéséhez ajánlott az elméleti
// anyagokat átolvasni. Tömören megfogalmazhatjuk az alábbi módon.
// A programok által használható memóriaterület két részre bomlik:

// |------------ halom ------------|- verem -|

// Az egyszerű változók a verem területen foglalódnak. Az xx[4] módon itt
// foglalni is egyszerűbb, mint a halom területen foglalni.
// Viszont egy függvény befejezésekor az általa használt része a veremnek
// felszabadul a foglalás alól, így az ő egyszerű változói már nem
// használhatók.
// Tehát ha függvényben akarunk úgy létrehozni tömböt, hogy a függvényen
// kívülről is elérjük, akkor a második módszert kell használnunk.
// ezen kívül még ez a praktikus, ha a tömb mérete nagy, vagy csak a program
// futásakor derül ki, hogy mekkora (pl. ha a program egyik paramétere az).

// Még egy fontos dolog tömbökkel kapcsolatban:
// A fenti program futásakor egyes fordítók figyelmeztetést adnak, hogy
// xx[1] értékét előzetes értékadás nélkül akarjuk használni. Igazuk van,
// hiszen itt egy elírás van, az xx[0]-nak adtunk csak értéket, tehát
// csak azt olvashatjuk ki. Javítsuk is ki!
// Ez viszont felhívja rá a figyelmet, hogy az indexekkel vigyázni kell,
// mert ha indexként változó értékét írjuk be, vagy egy kifejezést, akkor
// már nem veszi észre a fordító, hogyha helytelen index értékkel akarjuk a
// tömböt címezni. Még akkor sem mindig figyelmeztet, ha a megengedett
// tartományból kicímzünk.
```

In [ ]:
```
// Beolvasás vektorba:

    printf("v[3] ");
    scanf("%lf",&v[3]);  // A korábbihoz hasonlóan a v[3] címét adjuk át.
    printf("%f\n",v[3]);
```

Beolvasás fájlból

In [ ]:
```
// Eddigi programunkat /* és */ jelekkel körítve kommentté tudjuk
// alakítani, így az megmarad mintának, de hatástalan lesz.
/* Tehát e módon lehet könnyen többsoros kommentet csinálni, ahogy e hét
sor is mutatja.
Alatta pedig kipróbálhatjuk az újabb példaprogramot. Ahhoz persze létre
kell hoznunk egy adatok.dat nevű fájlt abban a direktoriban, ahol a c
program van, és írni bele egy számot (a későbbiek kedvéért inkább ötöt).

Beolvasás fájlból így lehetséges:*/

/*    float s;             // az eddigi sorokat kikommenteltük
    ...
    printf("%f\n",v[3]);
*/

    double x;
    printf("adatok.dat \n");
    FILE* f;                        // fájlpointer változó létrehozása
    f = fopen("adatok.dat", "r");  // az értékadással adjuk meg, hogy
                // melyik fájlt akarjuk megnyitni,
                // és hogy olvasni ("r"), vagy írni ("w") akarunk e bele
    fscanf(f,"%lf", &x);    // az fscanf a scanf-hez hasonlóan használható
    fclose(f);              // ha már nem használjuk, így zárjuk be a fájl!
    printf("%f\n",x);
```

In [ ]:
```
/* Kombináljuk az utóbb tanult két dolgot: vektort olvassunk be fájlból!
Megint kommenteljük ki az előző programrészt! */

    int n,i;
    n=4;
    double *v;
    v = (double*)malloc(n*sizeof(double));
    FILE *f;
    f = fopen("adatok.dat", "r");
    for (i = 0; i < n; i++) {
        fscanf(f,"%lf", &v[i]);
    }
    fclose(f);

    printf("%lf\n",v[0]);
    printf("%lf\n",v[1]);
    printf("%lf\n",v[2]);
    printf("%lf\n",v[3]);
    free(v);
```

In [ ]:
```
/* Praktikus, ha függvénybe szervezzük ki a beolvasást. Így könyebben
tudjuk felhasználni egy másik programban, ill. egy programban két vagy több
tömböt beolvasni. Ehhez szebb megoldás, ha a fájlnevet változóként adjuk
át, akkor tudjuk a függvényt különböző fájlok beolvasására is használni.
Ehhez meg kell tanulni, hogyan lehet stringet lérehozni:
(legközelebb bővebben lesz szó stringekről) */

    char *fn;
    fn = "adatok.dat";
    FILE* f = fopen(fn, "r");  // a stringet pedig átadhatjuk az fopen-nek
```

In [ ]:
```
// Mostmár tényleg tegyük át függvénybe a beolvasást:

double *readvec(char *fn, int n) {  // így jelezzük, hogy a függvény
        // stringként kapja a fájlnevet, egészként a beolvasandó darabszámot
    double *v;
    int i;
    FILE *f;

    v = (double*)malloc(n*sizeof(double));
    f = fopen (fn, "r");
    for (i = 0; i < n; i++) {
        fscanf(f,"%lf", &v[i]);
    }
    fclose(f);
    return v;    // visszaadjuk a lefoglalt területre mutató pointert
}

// main-ben ez marad:

    int n;
    double *v;  // ilyenkor kívül is kell egy pointer változó

    n=4;
    char *fn;
    fn="adatok.dat";
    v = readvec(fn,n);  // eltároljuk a pointer értékét,
                // innentől ugyanúgy tudjuk használni, mint az előzőekben:
    printf("%lf\n",v[0]);
    printf("%lf\n",v[1]);
    printf("%lf\n",v[2]);
    printf("%lf\n",v[3]);
    free(v);
```

Tudnivaló az 1. beadandó feladathoz:

Egyelőre a programba lehet beírni az adatokat, ahogy a fenti példákban. A parancsargumentumok használatát legközelebb tanuljuk meg.