

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

Ed. BHAX, DEBRECEN,
2019. február 19, v. 0.0.4

Copyright © 2019 Dr. Bátfai Norbert

Copyright (C) 2019, Norbert Bátfai Ph.D., batfai.norbert@inf.unideb.hu, nbatfai@gmail.com,

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

<https://www.gnu.org/licenses/fdl.html>

Engedélyt adunk Önnek a jelen dokumentum sokszorosítására, terjesztésére és/vagy módosítására a Free Software Foundation által kiadott GNU FDL 1.3-as, vagy bármely azt követő verziójának feltételei alapján. Nincs Nem Változtatható szakasz, nincs Címlapszöveg, nincs Hátlapszöveg.

<http://gnu.hu/fdl.html>

COLLABORATORS

	<i>TITLE :</i>	
	Univerzális programozás	
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>
WRITTEN BY	Bátfai, Norbert	2019. április 13.

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai

Ajánlás

„To me, you understand something only if you can program it. (You, not someone else!) Otherwise you don't really understand it, you only think you understand it.”

—Gregory Chaitin, *META MATH! The Quest for Omega*, [METAMATH]

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	7
2.3. Változók értékének felcserélése	9
2.4. Labdapattogás	10
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	11
2.6. Helló, Google!	14
2.7. 100 éves a Brun téTEL	15
2.8. A Monty Hall probléma	17
3. Helló, Chomsky!	20
3.1. Decimálisból unárisba átváltó Turing gép	20
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	20
3.3. Hivatalos nyelv	22
3.4. Saját lexikális elemző	23
3.5. l33t.l	24
3.6. A források olvasása	27
3.7. Logikus	30
3.8. Deklaráció	32

4. Helló, Caesar!	35
4.1. double ** háromszögmátrix	35
4.2. C EXOR titkosító	37
4.3. Java EXOR titkosító	39
4.4. C EXOR törő	40
4.5. Neurális OR, AND és EXOR kapu	42
4.6. Hiba-visszaterjesztéses perceptron	46
5. Helló, Mandelbrot!	48
5.1. A Mandelbrot halmaz	48
5.2. A Mandelbrot halmaz a std::complex osztállyal	51
5.3. Biomorfok	55
5.4. A Mandelbrot halmaz CUDA megvalósítása	60
5.5. Mandelbrot nagyító és utazó C++ nyelven	64
5.6. Mandelbrot nagyító és utazó Java nyelven	66
6. Helló, Welch!	69
6.1. Első osztályom	69
6.2. LZW	73
6.3. Fabejárás	76
6.4. Tag a gyökér	79
6.5. Mutató a gyökér	83
6.6. Mozgató szemantika	84
7. Helló, Conway!	86
7.1. Hangyaszimulációk	86
7.2. Java életjáték	88
7.3. Qt C++ életjáték	91
7.4. BrainB Benchmark	94
8. Helló, Schwarzenegger!	97
8.1. Szoftmax Py MNIST	97
8.2. Szoftmax R MNIST	97
8.3. Mély MNIST	97
8.4. Deep dream	97
8.5. Robotpszichológia	98

9. Helló, Chaitin!	99
9.1. Iteratív és rekurzív faktoriális Lisp-ben	99
9.2. Weizenbaum Eliza programja	99
9.3. Gimp Scheme Script-fu: króm effekt	99
9.4. Gimp Scheme Script-fu: név mandala	99
9.5. Lambda	100
9.6. Omega	100
10. Helló, Gutenberg!	101
10.1. Programozási alapfogalmak	101
10.2. Programozás bevezetés	105
10.3. Programozás	106
III. Második felvonás	107
11. Helló, Arroway!	109
11.1. A BPP algoritmus Java megvalósítása	109
11.2. Java osztályok a Pi-ben	109
IV. Irodalomjegyzék	110
11.3. Általános	111
11.4. C	111
11.5. C++	111
11.6. Lisp	111

Ábrák jegyzéke

2.1. A B_2 konstans közelítése	17
4.1. A double ** háromszögmátrix a memóriában	35

Előszó

Amikor programozónak terveztem állni, ellenezték a környezetemben, mondván, hogy kell szövegszerkesztő meg táblázatkezelő, de az már van... nem lesz programozói munka.

Tévedtek. Hogy egy generáció múlva kell-e még tömegesen hús-vér programozó vagy olcsóbb lesz alkalmi igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Minden esetre a programozás a teoretikus kultúra csúcsa. A GNU mozgalomban látom annak garanciáját, hogy ebben a szellemi kalandban a gyerekeim is részt vehessenek majd. Ezért programozunk.

Hogyan forgasd

A könyv célja egy stabil programozási szemlélet kialakítása az olvasóban. Módszere, hogy hetekre bontva ad egy tematikus feladatcsokrot. minden feladathoz megadja a megoldás forráskódját és forrásokat feldolgozó videókat. Az olvasó feladata, hogy ezek tanulmányozása után maga adja meg a feladat megoldásának lényegi magyarázatát, avagy írja meg a könyvet.

Miért univerzális? Mert az olvasótól (kvázi az írótól) függ, hogy kinek szól a könyv. Alapértelmezésben gyereknek, mert velük készítem az iniciális változatot. Ám tervezem felhasználását az egyetemi programozás oktatásban is. Ahogy szélesedni tudna a felhasználók köre, akkor lehetne kiadása különböző korosztályú gyereknek, családoknak, szakköröknek, programozás kurzusoknak, felnőtt és továbbképzési műhelyeknek és sorolhatnánk...

Milyen nyelven nyomjuk?

C (mutatók), C++ (másoló és mozgató szemantika) és Java (lebutított C++) nyelvekből kell egy jó alap, ezt kell kiegészíteni pár R (vektoros szemlélet), Python (gépi tanulás bevezető), Lisp és Prolog (hogy lássuk más is) példával.

Hogyan nyomjuk?

Rántsd le a <https://gitlab.com/nbatfai/bhax> git repót, vagy méginkább forkolj belőle magadnak egy sajátot a GitLabon, ha már saját könyvön dolgozol!

Ha megvannak a könyv DocBook XML forrásai, akkor az alább látható **make** parancs ellenőrzi, hogy „jól formázottak” és „érvényesek-e” ezek az XML források, majd elkészíti a dblatex programmal a könyved pdf változatát, íme:

```
batfai@entropy:~$ cd glrepos/bhax/thematic_tutorials/bhax_textbook/
batfai@entropy:~/glrepos/bhax/thematic_tutorials/bhax_textbook$ make
rm -f bhax-textbook-fdl.pdf
xmllint --xinclude bhax-textbook-fdl.xml --output output.xml
xmllint --relaxng http://docbook.org/xml/5.0/rng/docbookxi.rng output.xml ←
    --noout
output.xml validates
rm -f output.xml
dblatex bhax-textbook-fdl.xml -p bhax-textbook.xls
Build the book set list...
Build the listings...
XSLT stylesheets DocBook - LaTeX 2e (0.3.10)
=====
Stripping NS from DocBook 5/NG document.
Processing stripped document.
Image 'dblatex' not found
Build bhax-textbook-fdl.pdf
'bhax-textbook-fdl.pdf' successfully built
```

Ha minden igaz, akkor most éppen ezt a legenerált **bhax-textbook-fdl.pdf** fájlt olvasod.



A DocBook XML 5.1 új neked?

Ez esetben forgasd a <https://tdg.docbook.org/tdg/5.1/> könyvet, a végén találod az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

- Olvasgasd a kézikönyv lapjait, kezd a **man man** parancs kiadásával. A C programozásban a 3-as szintű lapokat fogod nézegetni, például az első feladat kapcsán ezt a **man 3 sleep** lapot
- [KERNIGHANRITCHIE]
- [BMECPP]
- Az igazi kockák persze csemegéznek a C nyelvi szabvány [ISO/IEC 9899:2017](#) kódcsipeteiből is.

1.3. Milyen filmeket nézzek meg?

- 21 - Las Vegas ostroma, <https://www.imdb.com/title/tt0478087/>, benne a **Monty Hall probléma** bemutatása.

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

A feladatokkal kapcsolatos élő adásokat sugároz a <https://www.twitch.tv/nbatfai> csatorna, melynek permanens archívuma a <https://www.youtube.com/c/nbatfai> csatornán található.

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

Írj olyan C végtelen ciklusokat, amelyek 0 illetve 100 százalékban dolgoztatnak egy magot és egy olyat, amely 100 százalékban minden magot!

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/tree/master/textbook_programs/Turing/vegtelen_ciklus

Ebben az esetben a következő végtelen ciklus 100%-ban dolgoztat egy magot:

```
int main()
{
    while(1) {}
    return 0;
}
```

A program fordítása és indítása után a **top -p `pgrep -u adrian loop1`** parancs beírásakor láthatjuk a várt eredményt.

```
Tasks: 1 total, 1 running, 0 sleeping, 0 stopped, 0 zombie
%Cpu0: 1,3 us, 0,3 sy, 0,0 ni, 98,0 id, 0,0 wa, 0,0 hi, 0,3 si, 0,0 st
%Cpu1: 0,3 us, 0,3 sy, 0,0 ni, 99,3 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu2: 100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu3: 1,4 us, 0,0 sy, 0,0 ni, 98,6 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem: 8025272 total, 4308952 free, 1653004 used, 2063316 buff/cache
KiB Swap: 3897340 total, 3897340 free, 0 used. 5653664 avail Mem

PID USER      PR  NI      VIRT      RES      SHR S  %CPU %MEM      TIME+ COMMAND
4134 adrian    20    0      4376      756      692 R 100,0  0,0      1:14.23 loop1
```

0% terhelés eléréséhez meg kell hívnunk a `sleep(n)` metódust ami lelassítja az iterálás sebességét.

```
int main()
{
    while(1)
    {
        sleep(1);
    }
    return 0;
}
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
4515	adrian	20	0	4376	816	752	S	0,0	0,0	0:00.00	loop0

Minden mag 100% terhelését szálasítással lehet megoldani az OpenMP segítségével.

```
#include <omp.h>

int main()
{
    #pragma omp parallel
    {
        while(1) {}
    }
    return 0;
}
```

gcc loop_all_cores.c -o loop -fopenmp parancsal fordítjuk le ebben az esetben.

```
Tasks: 1 total, 1 running, 0 sleeping, 0 stopped, 0 zombie
%Cpu0:100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu1:100,0 us, 0,3 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu2:100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu3:100,0 us, 0,3 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
4914	adrian	20	0	35576	1024	928	R	388,7	0,0	8:00.85	loop

2.2. Lefagyott, nem fagyott, akkor most mi van?

Mutasd meg, hogy nem lehet olyan programot írni, amely bármely más programról eldönti, hogy le fog-e fagyni vagy sem!

Megoldás videó:

Megoldás forrása: tegyük fel, hogy akkora haxorok vagyunk, hogy meg tudjuk írni a Lefagy függvényt, amely tetszőleges programról el tudja dönteni, hogy van-e benne végtelen ciklus:

```
Program T100
{
    boolean Lefagy(Program P)
    {
        if (P-ben van végtelen ciklus)
            return true;
        else
            return false;
    }

    main(Input Q)
    {
        Lefagy(Q)
    }
}
```

A program futtatása, például akár az előző v.c ilyen pszeudódjára:

```
T100(t.c.pseudo)
true
```

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

A T100-as programot felhasználva készítsük most el az alábbi T1000-set, amelyben a Lefagy-ra épőlő Lefagy2 már nem tartalmaz feltételezett, csak csak konkrét kódot:

```
Program T1000
{
    boolean Lefagy(Program P)
    {
        if (P-ben van végtelen ciklus)
            return true;
        else
            return false;
    }
```

```
boolean Lefagy2(Program P)
{
    if(Lefagy(P))
        return true;
    else
        for(; ; );
}

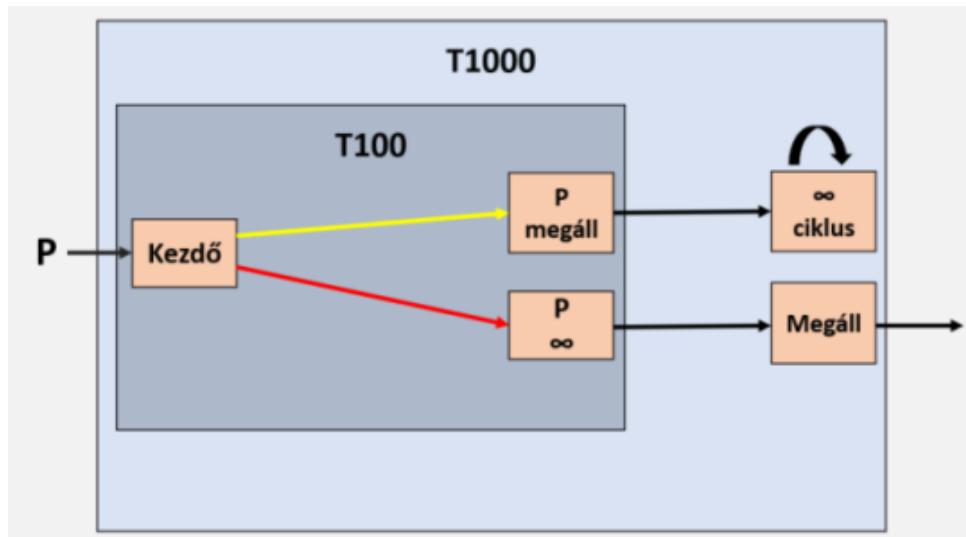
main(Input Q)
{
    Lefagy2(Q)
}

}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

- Ha T1000 lefagyó, akkor nem fog lefagyni, kiírja, hogy true
- Ha T1000 nem fagyó, akkor pedig le fog fagyni...

akkor most hogy fog működni? Sehogy, mert ilyen Lefagy függvényt, azaz a T100 program nem is létezik.



Tegyük fel, hogy létezik olyan program (T100), ami el tudja dönteni egy másik programról, hogy van-e benne végtelen ciklus. Ha van, akkor megáll a program, ha nincs, akkor pedig végtelen ciklusba kezd. Létrehozunk egy új programot (T1000) az előzőt (T100) felhasználva és ha a T100 megállt, akkor végtelen ciklusba kezd, ha pedig a T100 kezdett végtelen ciklusba, akkor megáll.

Mi történik, ha magát etetjük meg a programunkkal?

A megállás csak akkor lehetséges, ha a T100 nem áll meg, de ez pedig csak akkor lehet, ha a második argumentumként kapott saját programunk megáll.

Ebből ellentmondásra jutottunk, tehát nem lehet ilyen programot írni.

2.3. Változók értékének felcserélése

Írj olyan C programot, amely felcseréli két változó értékét, bármiféle logikai utasítás vagy kifejezés naszánálata nélkül!

Megoldás videó: https://bhaxor.blog.hu/2018/08/28/10_begin_goto_20_avagy_elindulunk

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Turing/val_csere.c

Két változó értékének felcserélésének legegyszerűbb módja, amikor segédváltozót használunk, amiben ideiglenesen eltároljuk az első változó értékét, majd az elsőt egyenlővé tesszük a másodikkal és a másodikat a segédváltozóval. Azonban léteznek módszerek segédváltozó nélküli cserére is.

Ebben az esetben segédváltozó nélkül cseréljük meg két változó értékét a különbségük kihasználásával.

```
#include<stdio.h>

int main()
{
    int a = 3;
    int b = 8;

    //segedvaltozo nélkül
    b = b-a;
    a = a+b;
    b = a-b;
    printf("a=%d b=%d\n", a, b);
```

A következő módszer pedig az EXOR-os csere.

```
#include<stdio.h>

int main()
{
    int a = 3;
    int b = 8;

    //exorra
    //kettes számrendszerben:
    //a = 8-> 0001
    //b = 3-> 1100

    a = a^b; //1101
    b = a^b; //0001
    a = a^b; //1100

    printf("a=%d b=%d\n", a, b);
```

2.4. Labdapattogás

Először if-ekkel, majd bármiféle logikai utasítás vagy kifejezés nasználata nélkül írj egy olyan programot, ami egy labdát pattogtat a karakteres konzolon! (Hogy mit értek pattogtatás alatt, alább láthatod a videókon.)

Megoldás videó: <https://bhaxor.blog.hu/2018/08/28/labdapattogas>

Megoldás forrása: https://github.com/p-adrian05/BHAX/tree/master/textbook_programs/Turing/labda

A labdapattogás nevű feladat klasszikus megoldása feltételvizsgálatokkal történik, ahol nyilvántartjuk a terminál méretét (fordításkor -Incursest könyvtár használatával), a labda koordinátáinak kezdőértékét és lépések mértékét. Ezután if feltételvizsgálattal nézzük, hogy ha elérte az egyik oldalt a labda, akkor változtatjuk a lépések irányát. Használjuk még a usleep(n) metódust aminek segítségével lassítjuk a labda mozgását és a clear() metódussal pedig csak 1 labdát fogunk látni, mivel minden egyes iterációnál törli a ablakot. Azonban if nélkül is megoldható a feladat:

```
#include <iostream>
#include <vector>
#include <unistd.h>
using namespace std;

const int szelesseg = 40;//ablak szelesseg
const int magassag = 20;//ablak magassag
int x=1, y=1; //labda kezdoertek
int deltax=1,deltay=1; // hanyasaval lepkedjen
vector<int> ablakx;
vector<int> ablaky;

void kirajzol();
void mozgatas(){
    x=x+deltax;
    y=y+deltay;
    deltax=deltax*ablakx[x];
    deltay=deltay*ablaky[y];
}

int main(){

    for (int i=0; i<szelesseg; i++)
    {
        ablakx.push_back(1);
    }

    for (int i=0; i<magassag; i++)
    {
        ablaky.push_back(1);
    }

    ablakx[0]=-1;//bal oldal
```

```
ablakx[szelesség-1]=-1; // jobb oldal
ablaky[0]=-1; //teteje
ablaky[magasság]=-1; // alja

for(;;)
{
    kirajzol();
    mozgatas();
    usleep(100000);
}

}
```

Hasonlóan az if-es megoldáshoz itt is rögzítjük a kezdőértékeket. Ebben az esetben nem saját ablakot rajzolunk ki, hogy lássunk ilyet is ,aminek előre meghatározzuk a magasságát és szélességét. A **kirajzol()** metódus keretet rajzol ki, amiben majd pattog a labda, más dolga nincs. A **main()** metódusban az ablakunk x és y szélességével töltjük fel a két vektort. Ezután beállítjuk -1 értékre a vektorok két elemét, hogy a **mozgatas()** metódusban megszorozzuk a labda lépéseinak nagyságát (ami egyben az irányért is felelős) az oldalak értékeivel. Ha -1-el szorzunk, akkor ez az jelenti, hogy elérte az egyik oldalt és változtatjuk az irányt, hasonlóan az if-es megoldásnál, ahol ugyanez ifekkel történt.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

Írj egy programot, ami megnézi, hogy hány bites a szó a gépeden, azaz mekkora az int mérete. Használd ugyanazt a while ciklus fejet, amit Linus Torvalds a BogoMIPS rutinjában!

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/tree/master/textbook_programs/Turing/bogomips

```
#include <stdio.h>

int main()
{
    int word = 1;
    int length = 0;

    do
    {
        length++;
    }
    while (word<<=1);

    printf("A szó %d bites\n", length);

    return 0;
}
```

```
}
```

A típus méretét, illetve hogy hány bitet foglal, bitshifteléssel könnyen meghatározhatjuk. A while ciklusunkban minden shiftelünk egyet balra a biteken és addig növeljük a length változót, amíg csupa 0 bitet nem fog tartalmazni a word változónk. Mivel az int típusú változók 4 byte-on tárolódnak, ezért 32 bites lesz a szavunk.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs$ gcc wordLength.c -o ←
w
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs$ ./w
A szó 32 bites
```

BogoMips:

Ez az algoritmus a Linux kernelben a CPU sebességét méri fel induláskor a busy-loop beállításához. A busy-loop azt jelenti, hogy egy folyamat folyamatosan vizsgál egy feltételt, amíg az igaz nem lesz és a BogoMips mutatja meg, hogy hány másodpercig nem csinál semmit a CPU.

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

void delay (unsigned long long loops)
{
    for (unsigned long long i = 0; i < loops; i++);
}

int main (void)
{
    unsigned long long loops_per_sec = 1;
    unsigned long long ticks;

    printf ("Calibrating delay loop..");
    fflush (stdout);

    while ((loops_per_sec <= 1))
    {
        ticks = clock ();
        delay (loops_per_sec);
        ticks = clock () - ticks;
        printf ("%llu %llu\n", ticks, loops_per_sec);

        if (ticks >= CLOCKS_PER_SEC)
        {
            loops_per_sec = (loops_per_sec / ticks) * CLOCKS_PER_SEC;
            printf ("ok - %llu.%02llu BogoMIPS\n", loops_per_sec / 500000,
```

```
    (loops_per_sec / 5000) % 100);

    return 0;
}
}

printf ("failed\n");
return -1;
}
```

A while ciklusban bitshifteléssel megyünk végig a 2 hatványain. A ticks-ben tároljuk mennyi processzoridőt használt a CPU eddig, majd a delay() függvénynek átadjuk loops_per_sec változót (aminek a bitjei mindenkorodók kivonva az előző ticks-ben tárolt CPU időt, így megkapjuk, mennyi ideig tartott a elszámolni a loops_per_sec változó végéig. Majd megnézzük if-el, hogy nagyobb vagy egyenlő a kapott ticks, mint a CLOCKS_PER_SEC aminek az értéke 1 millió és ha ez igaz, akkor kiszámoljuk, hogy milyen érték kell ahhoz, hogy a ciklusértéket megkapjuk, ezzel meghatározva a CPU sebességét.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Turing/bogomips$ ./ ←
    bogo
Calibrating delay loop...2 2
1 4
1 8
1 16
1 32
1 64
2 128
3 256
5 512
9 1024
16 2048
31 4096
58 8192
115 16384
266 32768
489 65536
919 131072
1844 262144
3678 524288
3216 1048576
4878 2097152
9305 4194304
19356 8388608
37773 16777216
76300 33554432
148557 67108864
292380 134217728
583864 268435456
1167209 536870912
```

```
ok - 918.00 BogoMIPS
```

2.6. Helló, Google!

Írj olyan C programot, amely egy 4 honlapból álló hálózatra kiszámolja a négy lap Page-Rank értékét!

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Turing/pagerank.c

A Page Rank az interneten található oldalakat rangsorolja. Kezdetben minden oldalnak van egy szavazati pontja és ha az egyik linkeli a másikat, akkor a linkelt oldal megkapja a linkelő pontját. Tehát egy oldal akkor lesz előkelőbb helyen egy google kereséskor, ha minél több másik oldal linkel rá, illetve ezen oldalakra is minél többen linkelnek, annál jobb minőségűnek fog számítani egy linkelése vagy szavazata. Az alábbi algoritmusunk 4 honlapot rangsorol:

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

void kiir (double tomb[], int db)
{
    int i;

    for (i = 0; i < db; ++i)
        printf ("%f\n", tomb[i]);
}

double tavolsag (double PR[], double PRv[], int n)
{
    double osszeg = 0.0;
    int i;

    for (i = 0; i < n; ++i)
        osszeg += (PRv[i] - PR[i]) * (PRv[i] - PR[i]);

    return sqrt(osszeg);
}

int main (void)
{

    double L[4][4] = {
        {0.0, 0.0, 1.0 / 3.0, 0.0},
        {1.0, 1.0 / 2.0, 1.0 / 3.0, 1.0},
        {0.0, 1.0 / 2.0, 0.0, 0.0},
        {0.0, 0.0, 1.0 / 3.0, 0.0}
    };
}
```

```
double PR[4] = { 0.0, 0.0, 0.0, 0.0 };
double PRv[4] = { 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0 };

int i, j;

for (;;)
{
    for (i = 0; i < 4; ++i)
    {
        PR[i] = 0.0;
        for (j = 0; j < 4; ++j)
            PR[i] += (L[i][j] * PRv[j]);
    }
    if (tavolsag (PR, PRv, 4) < 0.00000001)
        break;

    for (i = 0; i < 4; ++i)
        PRv[i] = PR[i];
}

kiir (PR, 4);

return 0;
}
```

Az L nevű többdimenziós tömbben vannak rögzítve mátrix formájában az adatok a linkelésekről, melyik oldalra melyik oldal linkel és mennyit. A végtelen ciklusban nullázzuk PR összes elemét, majd rögtön hozzáadjuk az L mátrix és PRv vektor szorzatainak értékét. Ezután a távolság metódusunkban végigmegyünk a PR és PRv vektorokon és egy változóban eltároljuk ezek különbségének a négyzetét (hogy ne legyen negatív) és gyököt vonva visszaadjuk az értéket, amely ha kisebb mint 0.00000001, akkor kilépünk a végtelen ciklusból, ellenkező esetben pedig PRv tömböt feltöljük PR elemeivel. Végül kiiratjuk az értékeket.

```
0.090909
0.545455
0.272727
0.090909
```

2.7. 100 éves a Brun téTEL

Írj R szimulációt a Brun téTEL demonstrálására!

Megoldás videó: <https://youtu.be/xbYhp9G6VqQ>

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/attention_raising/Primek_R/stp.r

A Brun téTEL azt mondja, hogy az ikerprímszámok reciprokából képzett összege konvergál egy számhoz. Ezt határt Brun konstansnak nevezzük. Ezzel ellentétben a prímszámok a végtelen felé tartanak.

Mik azok a prímszámok?

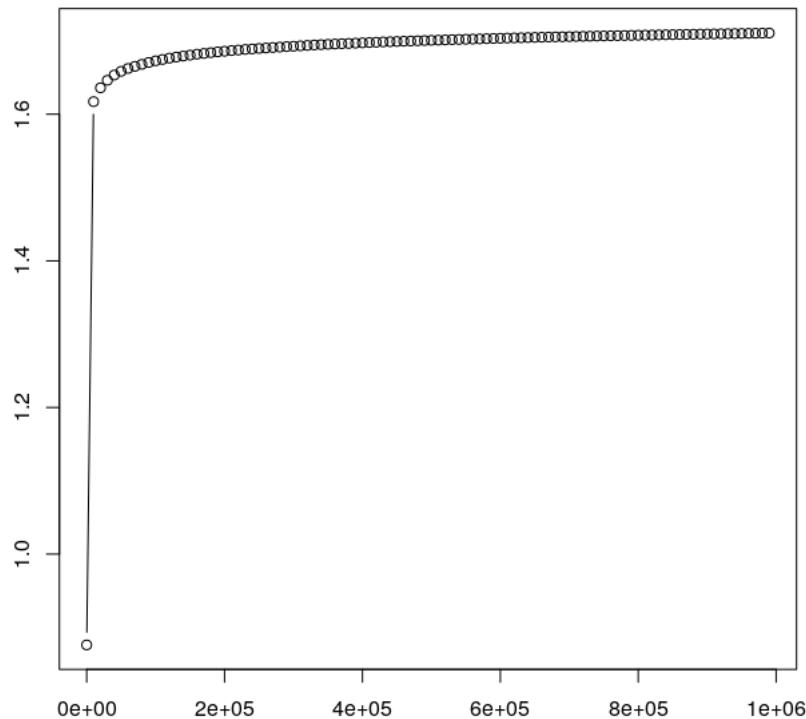
A prímszám olyan természetes szám, ami csak önmagával és eggyel osztható és minden természetes szám előállítható prímszámok szorzataként. Ikerprímek pedig azok a prímszámok, amelyek egymás után következnek és különbségük 2.

A Brun téTEL szimulációja a matematikai R nyelvben lesz megírva, használva a matlab csomagot a prímszámok számítására.

```
sumTwinPrimes <- function(x) {  
  
  primes = primes(x)  
  diff = primes[2:length(primes)] - primes[1:length(primes)-1]  
  idx = which(diff==2)  
  t1primes = primes[idx]  
  t2primes = primes[idx]+2  
  rt1plust2 = 1/t1primes + 1/t2primes  
  return(sum(rt1plust2))  
}  
  
x=seq(13, 1000000, by=10000)  
y=sapply(x, FUN = sumTwinPrimes)  
plot(x,y,type="b")
```

Létrehozunk egy függvényt aminek paraméterként átadjuk, hogy meddig számolja függvényünk az ikerprímszámokat. A primes vektorba kiszámoljuk a **primes(x)** fügvénnyel a prímszámokat. A diff vektorban eltároljuk az egymásután következő prímszámok különbségét. Az idx vektorban, pedig azokat a helyeket tároljuk, el ahol a különség 2, tehát ikerprímek találhatóak ott. A t1primes vektorban elátároljuk az ikerprímek első pájrát, a t2primes-ban pedig hozzáadva minden első párhoz kettőt, az ikerprímek második pájrát is. Az rt1plust2 vektorban tároljuk minden párnak a reciprokainak az összegét, majd ezeket az összegeket **sum()** fügvénnyel összeadjuk és visszaadjuk ezt az értéket.

Ezután kirajzoltatjuk és láthatjuk, hogy valóban egy felső határhoz konvergálnak az összegek.

2.1. ábra. A B_2 konstans közelítése

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

Megoldás videó: https://bhaxor.blog.hu/2019/01/03/erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall-paradoxon_kapcsan

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/attention_raising/MontyHall_R/mh.r

A Monty Hall probléma egy amerikai televíziós vetélezőben jelent meg, ahol a műsor végén a játékosnak 3 ajtó közül kellett választania. A nyeremény csak az egyik ajtó mögött volt. A játékos választása után a műsorvezető kinyitott egy üres ajtót és feltette a kérdést, hogy fenntartja-e a választását a játékos vagy egy másik ajtót választ. A Monty Hall probléma kérdése, hogy számít-e, hogy a játékos megváltoztatja-e a választását. Józan ésszel gondolkodva nem számít, mivel a maradék két ajtó közül az egyik mögött van a nyeremény, így 50-50% az esélye annak, hogy nyerünk. A feladvány bizonyítása több matematikai professzoron is kifogott, köztük a világhírű Erdős Pálon is, akit csak a számítógépes szimuláció győzött meg, ami alapján számít, hogy másik ajtót választunk, ugyanis ekkor megduplázódik az esélyünk a nyerésre.

Amikor először választunk ajtót, akkor $1/3$ az esélye annak, hogy eltaláljuk a nyertes ajtót és $2/3$, hogy nem. Ezután a játékvezető kinyit egy ajtót, amelyik üres és ha nem változtatunk a döntésünkön, továbbra is $1/3$ lesz annak az esélye, hogy nyerünk. Viszont mivel már csak 2 ajtó van a játékban ezért ha változtatunk, akkor $2/3$ lesz az esélyünk a nyerésre.

Ennek bizonyítását láthatjuk R nyelven megfogalmazva:

```
kiserletek_szama=1000000
kiserlet = sample(1:3, kiserletek_szama, replace=T)
jatekos = sample(1:3, kiserletek_szama, replace=T)
musorvezeto=vector(length = kiserletek_szama)

for (i in 1:kiserletek_szama) {

  if(kiserlet[i]==jatekos[i]){

    mibol=setdiff(c(1,2,3), kiserlet[i])

  }else{

    mibol=setdiff(c(1,2,3), c(kiserlet[i], jatekos[i]))


  }

  musorvezeto[i] = mibol[sample(1:length(mibol),1)]


}

nemvaltoztatesnyer= which(kiserlet==jatekos)
valtoztat=vector(length = kiserletek_szama)

for (i in 1:kiserletek_szama) {

  holvált = setdiff(c(1,2,3), c(musorvezeto[i], jatekos[i]))
  valtoztat[i] = holvált[sample(1:length(holvált),1)]


}

valtoztatesnyer = which(kiserlet==valtoztat)

sprintf("Kiserletek szama: %i", kiserletek_szama)
length(nemvaltoztatesnyer)
length(valtoztatesnyer)
length(nemvaltoztatesnyer)/length(valtoztatesnyer)
length(nemvaltoztatesnyer)+length(valtoztatesnyer)
```

Először eltároljuk, hogy hány kísérletet végezzünk el, majd a kiserlet és jatekos vektorokban kisorsolunk 1 és 3 között véletlenszerűen számokat. A műsorvezető vektorát beállítjuk a kísérletek számával. Ezután egy for ciklussal végigmegyünk minden kísérleten és ha kiserlet i-edig értéke megyegyezik a a játékos i-edig találatával, az jelenti, hogy eltalálta a játékos a nyereményt és a mibol vektorba az a két érték kerül be amelyeket a játékos nem választott. (Ez a két érték az üres ajtókat jelenti ebben az esetben.)

Ha a játékos nem találta el elsőre a kiserlet vektorban található számot, akkor a mibol vektorba már csak 1 egy érték kerülhet, az amelyik nem a nyeremény és nem a játékos által kiválasztott érték. Ezután a musorvezeto vektorba berakjuk a mibol vektorban található számot, illetve ha két érték van benne akkor a

kettőből az egyiket véletlenszerűen.

Ezután értkezik a kiértékelés. A nemvaltoztatesnyer vektorba kerülnek azok az esetek, amikor elsőre eltalálja a játékos a megfelelő ajtót. Megint végigmegyünk a kísérleteken és a holvált vektorba azok vagy az az érték kerül az 1, 2 és 3 közül amely nem egyenlő a műsorvezető és a játékos által választott vagyis ekkor ha váltana a játékos akkor nyerne. A változtat vektorba pedig a holvált vektor elemei közül az egyiket rakjuk át.

A valtoztatesnyer vektorba pedig azok az értékek kerülnek, amelyek a kiserlet vektorba és a változtat vektorba találhatóak, vagyis ekkor az az ajtó a nyertes ,amelyiket másodjára választanánk. Ezután pedig kiiratjuk az esetek számait:

```
[1] "Kiserletek szama: 1000000"
> length(nemvaltoztatesnyer)
[1] 333590
> length(valtoztatesnyer)
[1] 666410
> length(nemvaltoztatesnyer) /length(valtoztatesnyer)
[1] 0.5005777
> length(nemvaltoztatesnyer)+length(valtoztatesnyer)
[1] 1000000
```

3. fejezet

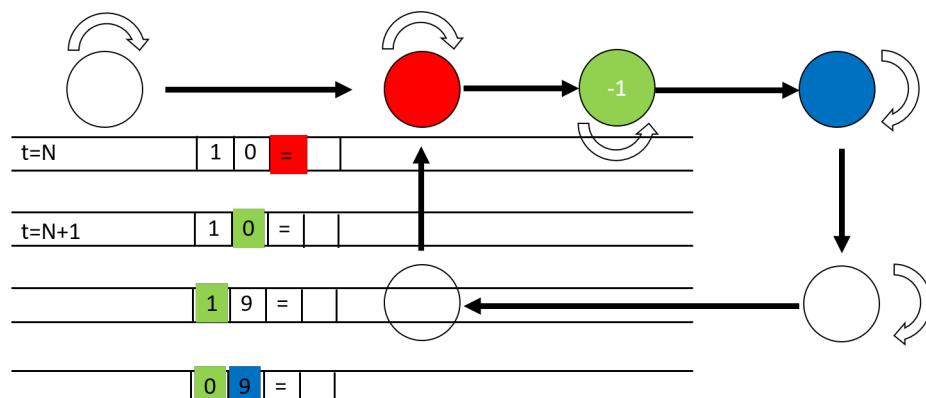
Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása:



A decimális számok N darab egyssel ábrázolva unáris számrendszerben, nincs más számjegy. A turing gép az átváltást úgy hajtja végre, hogy addig von ki a megadott decimális számból egyet, amíg az nulla nem lesz és a tárba pakolja az egyeseket. A feldolgozás mindenkor a szám utolsó számjegyével kezdődik. Ha ez 0, akkor 9-vel a kék állapotba kerül és addig ismétlődik, amíg 0 nem lesz. Ezután ez folytatódik a többi számjeggyel ugyanez.

3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen

Mutass be legalább két környezetfüggő generatív grammatikát, amely ezt a nyelvet generálja!

Megoldás videó:

Megoldás forrása:

A generatív grammatika Noam Chomsky nevéhez fűződik. A generatív nyelvelmélet célja a beszédtevékenység mechanizmusának modellálása, vagyis olyan modell létrehozása, amelynek a segítségével végtelen számú mondatot alkothatunk, generálhatunk.

A generatív grammaтика négy alkotóeleme:

Terminális szimbólumok: a definiálandó nyelv ábécéje.

Nem terminális jelek: egy másik ábécé, melynek jeleit csak segédeszközökkel használjuk a generálás során.

Kezdőszimbólum: egy kitüntetett szimbólum.

Helyettesítési szabályok, amelyekkel a szavak származtathatóak.

A generálás során adott egy szó, amit meghatározott helyettesítési szabály alapján cserélünk egy másik szóra, mindaddig, míg szavunk csak terminális jelekből fog állni.

S, X, Y „változók”

a, b, c „konstansok”

$S \rightarrow abc, S \rightarrow aXbc, Xb \rightarrow bX, Xc \rightarrow Ybcc, bY \rightarrow Yb, aY \rightarrow aaX, aY \rightarrow aa$

S (S → aXbc)

aXbc (Xb → bX)

abXc (Xc → Ybcc)

abYbcc (bY → Yb)

aYbbcc (aY → aa)

aabbcc

S (S → aXbc)

aXbc (Xb → bX)

abXc (Xc → Ybcc)

abYbcc (bY → Yb)

aYbbcc (aY → aaX)

aaXbbcc (Xb → bX)

aabXbcc (Xb → bX)

aabbXcc (Xc → Ybcc)

aabbYbcc (bY → Yb)

aabYbbccc (bY → Yb)

aaYbbbccc (aY → aa)

aaabbccc

A, B, C „változók”

a, b, c „konstansok”

$A \rightarrow aAB, A \rightarrow aC, CB \rightarrow bCc, cB \rightarrow Bc, C \rightarrow bc$

A (A → aAB)

aAB (A → aC)

aaCB (CB → bCc)

aabCc (C → bc)

aabbcc

A (A → aAB)

aAB (A → aAB)

aaABB (A → aAB)

aaaABBB (A → aC)

aaaaACBBB (CB → bCc)

```
aaaaabCcBB (cB → Bc)
aaaaabCBcB (cB → Bc)
aaaabCBBc (CB → bCc)
aaaabbCcBc (cB → Bc)
aaaabbCBcc (CB → bCc)
aaaabbbCccc (C → bc)
aaaabbbbcccc
```

3.3. Hivatkozási nyelv

A [KERNIGHANRITCHIE] könyv C referencia-kézikönyv/Utasítások melléklete alapján definiál BNF-ben a C utasítás fogalmát! Majd mutass be olyan kódcsipeteket, amelyek adott szabvánnyal nem fordulnak (például C89), mással (például C99) igen.

Megoldás videó:

Megoldás forrása:

```
<utasítás> ::= 
    <összetett_utasítás> | <feltételes_utasítás> | <iterációs_utasítás> | 
    <vezérlés_átadó_utasítás> | <címkézett_utasítás> | <kifejezés_utasítás ↔ 
    > | <>nulla_utasítás>
<összetett_utasítás> ::= <deklaráció_lista> | <utasításlista>
<deklaráció_lista> ::= <deklaráció>
<utasításlista> ::= <utasítás>
<feltételes_utasítás> ::= if | if else | switch
<iterációs_utasítás> ::= while | do while | for
<vezérlés_átadó_utasítás> ::= break | return | goto | continue
<címkézett_utasítás> ::= <azonosító>
<kifejezés_utasítás> ::= <kifejezés>
<nulla_utasítás> ::= ;
```

C99-el lefordul:

```
#include <stdio.h>

int main ()
{
    // Printing to screen.
    printf ("Hello World\n");
}
```

```
adrian@adrian-MS-7817:~/Desktop/programs$ gcc -o a -std=c99 a.c
adrian@adrian-MS-7817:~/Desktop/programs$
```

Azonban C89-el nem fordul le:

```
adrian@adrian-MS-7817:~/Desktop/programs$ gcc -o a -std=c89 a.c
a.c: In function 'main':
a.c:5:7: error: C++ style comments are not allowed in ISO C90
    // Printing to screen.
    ^
a.c:5:7: error: (this will be reported only once per input file)
```

3.4. Saját lexikális elemző

Írj olyan programot, ami számolja a bemenetben megjelenő valós számokat! Nem elfogadható olyan megoldás, amely maga olvassa betűnként a bemenetet, a feladat lényege, hogy lexert használunk, azaz óriások vállán állunk és ne kispályázzunk!

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Chomsky/realnumber.l

A lexer minta illesztésen alapulva legenerálja a lexikális elemzőket, nekünk csak a mintát kell megadnunk, ami alapján figyeli a begépelt szöveget. Ebben a feladatban olyan programot írunk, ami a begépelt szövegből felismeri és megszámolja a valós számokat.

```
% {
#include <stdio.h>
int realnumbers = 0;
%
digit [0-9]
%%
{digit}*(\.{digit}+) ? {++realnumbers;
    printf("[realnum=%s %f]", yytext, atof(yytext));}
%%
int
main ()
{
    yylex ();
    printf("The number of real numbers is %d\n", realnumbers);
    return 0;
}
```

Az első részben deklarálunk egy int változót, amit növelve kapjuk meg a számok számát és végén definiáljuk digit néven 0-9-ig a számjegyeket.

```
% {  
#include <stdio.h>  
int realnumbers = 0;  
%}  
digit [0-9]
```

A második részben adjuk a mintát. A * előtt a digit, azt jelenti, hogy digitból(számból) bármennyi lehet. A . önmagában azt jelenti, hogy bármilyen karakterre rá lehet illeszteni, azonban nekünk le kell védeni \jellel és így a valós számoknál lévő pontot fogja értelmezni. Utána digit, vagyis megint jönnek a számjegyek, a + pedig azt jelenti, hogy legalább 1 számnak kell lennie a pont után. És ha van találat a mintára, akkor növeljük a realnumbers változót. Ezután kiiratjuk a felismert számot, illetve az atof-al átkonvertált double verzióját is.

```
%%  
{digit}*(\.{digit}+)? {++realnumbers;  
printf("[realnum=%s %f]", yytext, atof(yytext));}  
%%
```

A harmadik rész maga a program, ahol meghívjuk a lexert, ami összerakja a fenti minta alapján a programkat és a program befejezéte után kiírja a talált számokat.

Fordításkor először a lexnek megmondjuk milyen kimenetet készítsen a fent megírt realnumber.l forrásból. Ezután szokott módon fordítjuk a kapott c forráskódot hozzálinkelve a flex könyvtárat.

```
adrian@adrian-MS-7817:~/Desktop/programs$ clear  
adrian@adrian-MS-7817:~/Desktop/programs$ lex -o realnumber.c realnumber.l  
adrian@adrian-MS-7817:~/Desktop/programs$ gcc realnumber.c -o realnumber -lfl
```

3.5. I33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Chomsky/I33t

Az l377dlc7 struktúrában fogalmazzuk meg a saját szótárunkat. Egy karakterhez négy féle őt ábrázolni próbáló karektort lehet helyettesíteni. Az inputként megadott karaktereken megyünk végig és ha a karakter megtalálható a szótárban, akkor egy random egész számot kap 1 és 100 között, ami alapján eldől, hogy a 4 karakter közül melyikkel lesz helyettesítve. 90%-ban a szótárunkban megfogalmazott első karakterrel lesz helyettesítve. 4%, hogy a második és 3-3%, hogy a harmadik vagy negyedik karakterrel. A found változóba tároljuk, hogy volt-e speciális karakterrel kiirva. A 0 hamis jelent, az 1 pedig igazat. Ha 0 marad, akkor a bevitt karektort írjuk ki.

```
% {  
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
#include <ctype.h>  
  
#define L337SIZE (sizeof l337d1c7 / sizeof (struct cipher))  
  
struct cipher {  
    char c;  
    char *leet[4];  
} l337d1c7 [] = {  
  
{'a', {"4", "4", "@", "/-\\"}},  
{'b', {"b", "8", "l3", "|{}"}},  
{'c', {"c", "(", "<", "{}"}},  
{'d', {"d", "l)", "[", "|{}"}},  
{'e', {"3", "3", "3", "3"}},  
{'f', {"f", "|=", "ph", "|#"}}},  
{'g', {"g", "6", "[", "[+"}}},  
{'h', {"h", "4", "|-", "[-]}},  
{'i', {"1", "1", "|", "!"}}},  
{'j', {"j", "7", "_|", "_/"}},  
{'k', {"k", "|<", "1<", "|{"}}},  
{'l', {"l", "1", "|", "|_"}},  
{'m', {"m", "44", "(V)", "|\\/|"}},  
{'n', {"n", "|\\|", "/\\/", "/v"}},  
{'o', {"0", "0", "()", "[]"}},  
{'p', {"p", "/o", "|D", "|o"}},  
{'q', {"q", "9", "O_", "(,)"}},  
{'r', {"r", "12", "12", "|2"}},  
{'s', {"s", "5", "$", "$"}},  
{'t', {"t", "7", "7", "'|'"}}},  
{'u', {"u", "|_|", "(_)", "[_]"}},  
{'v', {"v", "\\\/", "\\\/", "\\\/"}}},  
{'w', {"w", "VV", "\\\/\\\/", "(/\\\/)"}},  
{'x', {"x", "%", ")(")}},  
{'y', {"y", "", "", ""}}},  
{'z', {"z", "2", "7_", ">_"}},  
  
{'0', {"D", "0", "D", "0"}},  
{'1', {"I", "I", "L", "L"}},  
{'2', {"Z", "Z", "Z", "e"}},  
{'3', {"E", "E", "E", "E"}},  
{'4', {"h", "h", "A", "A"}},  
{'5', {"S", "S", "S", "S"}},  
{'6', {"b", "b", "G", "G"}},  
{'7', {"T", "T", "j", "j"}},
```

```
{'8', {"X", "X", "X", "X"}},  
'9', {"g", "g", "j", "j"}}  
};  
  
%}  
%%  
. {  
  
    int found = 0;  
    for(int i=0; i<L337SIZE; ++i)  
    {  
  
        if(l337d1c7[i].c == tolower(*yytext))  
        {  
  
            int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0));  
  
            if(r<91)  
                printf("%s", l337d1c7[i].leet[0]);  
            else if(r<95)  
                printf("%s", l337d1c7[i].leet[1]);  
            else if(r<98)  
                printf("%s", l337d1c7[i].leet[2]);  
            else  
                printf("%s", l337d1c7[i].leet[3]);  
  
            found = 1;  
            break;  
        }  
  
    }  
  
    if(!found)  
        printf("%c", *yytext);  
  
}  
%%  
int  
main()  
{  
    srand(time(NULL)+getpid());  
    yylex();  
    return 0;  
}
```

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Chomsky$ lex -o 133t ←  
.c 133t.l
```

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Chomsky$ gcc 133t.c ←
-o 133t -lfl
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Chomsky$ ./133t
Az 1377dlc7 struktúrában fogalmazzuk meg a saját szótárunkat. Egy ←
karakterhez négy féle őt ábrázolni
próbáló karektert lehet helyettesíteni. Az inputként megadott karaktereken ←
megyünk végig és ha a karakter
megtalálható a a szótárban, akkor egy random egész számot kap 1 és 100 ←
között, ami alapján eldől, hogy a 4
karakter közül melyikkel lesz helyettesítve.4z 1ETT||1cj struktúráb4n ←
f0g4lm4zzuk 4436 4 54ját szótárunk4t. 3gy k4|24kt3r43z négy fél_3 őt ←
ábrázoln1
próbáló |{4|23|{t3r' |' 13h3t h3ly3tt3sít3/V1. 4z 1nputként |\|3g4d()tt ←
k@r4kt3r3k3n m3gyünk vég1g és h4 4 |{4r3kt3r
m3gt4lálh4tó 4 4 s>_ótárb4n, 4kk0r 3gy r4nd0m 3gész szám0t k4p I és IDD ←
között, 4m1 414pján 3|_dől, h0g 4 h
```

3.6. A források olvasása

Hogyan olvasod, hogyan értelmezed természetes nyelven az alábbi kódcsipeteket? Például

```
if(signal(SIGINT, jelkezelo)==SIG_IGN)
    signal(SIGINT, SIG_IGN);
```

Ha a SIGINT jel kezelése figyelmen kívül volt hagyva, akkor ezen túl is legyen figyelmen kívül hagyva, ha nem volt figyelmen kívül hagyva, akkor a jelkezelo függvény kezelje. (Miután a **man 7 signal** lapon megismertem a SIGINT jelet, a **man 2 signal** lapon pedig a használt rendszerhívást.)



Bugok

Vigyázz, sok csipet kerülendő, mert bugokat visz a kódba! Melyek ezek és miért? Ha nem megy ránézésre, elkapja valamelyiket esetleg a splint vagy a frama?

i.

```
if(signal(SIGINT, SIG_IGN) !=SIG_IGN)
    signal(SIGINT, jelkezelo);
```

Ha a SIGINT jel kezelése nem volt figyelmen kívül hagyva, akkor a jelkezelo függvény kezelje, máskülönben figyelmen kívül legyen hagyva.

ii.

```
for(i=0; i<5; ++i)
```

for ciklussal 0-tól 4-ig megyünk, az i-t növelve. ++i azt jelenti, hogy megnöveli az i értékét majd visszaadja az 1-el megnövelt értéket.

iii.

```
for(i=0; i<5; i++)
```

for ciklussal 0-tól 4-ig megyünk, az i-t növelte. i++ azt jelenti, hogy megnöveli az i értékét, de először visszaadja az eredeti értéket és azután növeli. For ciklusban nincs jelentősége, hogy ++i vagy i++;

iv.

```
for(i=0; i<5; tomb[i] = i++)
```

Indulunk for ciklussal 0-tól 4-ig, majd tomb elemeit megváltoztatjuk arra az i-re ami megnövelés előtt volt.

```
adrian@adrian-MS-7817:~/Desktop/programs$ splint a.c
Splint 3.1.2 --- 20 Feb 2018

a.c: (in function main)
a.c:9:26: Expression has undefined behavior (left operand uses i, ←
modified by
    right operand): tomb[i] = i++
Code has unspecified behavior. Order of evaluation of function ←
parameters or
subexpressions is not defined, so if a value is used and modified in
different places not separated by a sequence point constraining ←
evaluation
order, then the result of the expression is unspecified. (Use - ←
    evalorder to
    inhibit warning)
a.c:15:2: Path with no return in function declared to return int
There is a path through a function declared to return a value on ←
    which there
is no return statement. This means the execution may fall through ←
    without
returning a meaningful result to the caller. (Use -noret to inhibit ←
warning)

Finished checking --- 2 code warnings
```

v.

```
for(i=0; i<n && (*d++ = *s++); ++i)
```

0-tól megyünk for ciklussal, addig amíg i kisebb mint n és ha d tömbre mutató mutató következő eleme egyenlő az s mutató által mutatott tömb következő elemével. Hiba: összehasonlításkor 2db egyenlőségjelet használunk.

vi.

```
printf("%d %d", f(a, ++a), f(++a, a));
```

Kiiratunk két egész számot. Mindkettőt az f függvény adja vissza. Első esetben az f függvénynek átadjuk az a változót és az a változó 1-el megnövelt értékét. Másodszor pedig az a változó 1-el megnövelt értékét és a-t adjuk átadjuk az f függvénynek.

```
adrian@adrian-MS-7817:~/Desktop/programs$ splint a.c
Splint 3.1.2 --- 20 Feb 2018

a.c: (in function main)
a.c:16:22: Argument 2 modifies a, used by argument 1 (order of ←
    evaluation of
        actual parameters is undefined): f(a, ++a)
Code has unspecified behavior. Order of evaluation of function ←
    parameters or
subexpressions is not defined, so if a value is used and modified in
different places not separated by a sequence point constraining ←
    evaluation
order, then the result of the expression is unspecified. (Use - ←
    evalorder to
    inhibit warning)
a.c:16:30: Argument 1 modifies a, used by argument 2 (order of ←
    evaluation of
        actual parameters is undefined): f(++a, a)
a.c:16:17: Argument 2 modifies a, used by argument 3 (order of ←
    evaluation of
        actual parameters is undefined): printf("%d %d", f(a, ++a), f(++a, ←
            a))
a.c:16:28: Argument 3 modifies a, used by argument 2 (order of ←
    evaluation of
        actual parameters is undefined): printf("%d %d", f(a, ++a), f(++a, ←
            a))
a.c:2:5: Function exported but not used outside a: f
A declaration is exported, but not used outside this module. ←
    Declaration can
use static qualifier. (Use -exportlocal to inhibit warning)
a.c:6:1: Definition of f

Finished checking --- 5 code warnings
```

vii.

```
printf("%d %d", f(a), a);
```

Két egész számot iratunk ki, az egyik az f függvény által visszaadott szám, az a változót átadjuk az f-nek, a másik pedig az a változó.

```
adrian@adrian-MS-7817:~/Desktop/programs$ splint a.c
Splint 3.1.2 --- 20 Feb 2018

a.c:2:5: Function exported but not used outside a: f
A declaration is exported, but not used outside this module. ←
    Declaration can
use static qualifier. (Use -exportlocal to inhibit warning)
```

```
a.c:6:1: Definition of f  
Finished checking --- 1 code warning
```

viii.

```
printf("%d %d", f(&a), a);
```

Két egész számot íratunk ki, az egyik az f függvény által visszaadott szám, az a változó memóriacímét átadjuk az f-nek, a másik pedig az a változó.

```
adrian@adrian-MS-7817:~/Desktop/programs$ splint a.c  
Splint 3.1.2 --- 20 Feb 2018  
  
a.c: (in function main)  
a.c:16:19: Function f expects arg 1 to be int gets int *: &a  
    Types are incompatible. (Use -type to inhibit warning)  
a.c:2:5: Function exported but not used outside a: f  
    A declaration is exported, but not used outside this module. ←  
        Declaration can  
    use static qualifier. (Use -exportlocal to inhibit warning)  
a.c:6:1: Definition of f
```

Finished checking --- 2 code warnings

Megoldás forrása:

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat...

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

```
$ (\forall x \exists y ((x < y) \wedge (y \text{ prim}))) $  
$ (\forall x \exists y ((x < y) \wedge (y \text{ prim})) \wedge (\forall y \text{ prim})) $  
$ (\exists y \forall x (x \text{ prim}) \supset (x < y)) $  
$ (\exists y \forall x (y < x) \supset \neg (x \text{ prim})) $
```

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Chomsky/Matlog/-matlog.tex

Megoldás videó: <https://youtu.be/ZexiPy3ZxsA>, https://youtu.be/AJSXOQFF_wk

Ahhoz, hogy a fenti mondatokat megadjuk, definiálni kell következőket:

```
$ (x \text{ páros}) \leftrightharpoons \exists y (y+y=x) $
$ (x < y) \leftrightharpoons \exists z (z+x=y) \wedge \neg (x=y) $
$ (x \text{ prím}) \leftrightharpoons (\forall z (z \text{ vert } x) \supset (z = x \vee z=S0)) \wedge (x \neq 0) \wedge (x \neq S0) $
```

Majd ezt lefordítva pdf formátumba ezt kapjuk:

- $(x \text{ páros}) \Leftrightarrow \exists y (y+y=x)$
- $(x < y) \Leftrightarrow \exists z (z+x=y) \wedge \neg (x=y)$
- $(x|y) \Leftrightarrow \exists z (z \cdot x = y) \wedge (x \neq 0)$
- $(x \text{ prím}) \Leftrightarrow (\forall z (z|x) \supset (z = x \vee z = S0)) \wedge (x \neq 0) \wedge (x \neq S0)$
- $(\forall x \exists y ((x < y) \wedge (y \text{ prím}))) \Leftrightarrow (\infty \text{ sok prímszám van})$
- $(\forall x \exists y ((x < y) \wedge (y \text{ prím} \wedge (SSy \text{ prím}))) \Leftrightarrow (\infty \text{ sok iker-prímszám van})$
- $(\exists y \forall x (x \text{ prím} \supset (x < y)) \Leftrightarrow (\text{véges sok prímszám van})$
- $(\exists y \forall x (y < x) \supset \neg (x \text{ prím})) \Leftrightarrow (\text{véges sok prímszám van})$

x akkor páros, ha van olyan y változó, amihez önmagát hozzáadva x -et kapjuk.

x kisebb mint y , ha van olyan z változó, amihez x -et hozzáadva y -t kapjuk és x nem egyenlő y -nal.

x osztója y -nak, ha létezik olyan z változó, hogy ha z -t megszorozzuk x -el, y -t kapjuk és x nem egyenlő 0-val.

x prímszám, ha minden olyan z változó osztója x -nek ami egyenlő x -el vagy 1-gyel és x nem egyenlő 0-val és 1-gyel.

Végtelen sok prímszám van: minden x változó esetén van olyan y változó, amely nagyobb mint x és y prímszám.

Végtelen sok iker-prímszám van: minden x változó esetén van olyan y változó, amely nagyobb mint x , prímszám és hozzáadva kettőt is prímszám.

Véges sok prímszám van: Van olyan y változó, amely minden x változó esetén, ha x prím, akkor nagyobb mint x .

(2. megfogalmazás) Véges sok prímszám van: Van olyan y változó, amely ha minden x változó esetén, kisebb mint x , akkor x nem prímszám.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciajára
- egészek tömbje
- egészek tömbjének referenciajára (nem az első elemé)
- egészre mutató mutatók tömbje
- egészre mutató mutatót visszaadó függvény
- egészre mutató mutatót visszaadó függvényre mutató mutató
- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény
- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

```
int *pointerFg(int *c)
{
    return c;
}

int fuggveny(int a, int b)
{
    return a;
}

typedef int (*A) (int,int);
A fuggveny2(int);

int main()
{
    //egesz
    int a = 1;

    //egeszre mutato mutato
    int *b;

    //egesz referenciajára
```

```
b = &a;

//egeszek tombje
int tomb[5] = {1,2,3,4,5};

//egeszek tombjenek referenciaja
int (&tombref)[5] = tomb;

//egeszre mutato mutatok tombje
int *tombMutato[5];

//egeszre mutato mutatot visszaado fuggveny
int *c = pointerFg(&a);

//egeszre mutato mutatot visszaado fuggvenyre mutato mutato
int *(*mutato)(int*) = pointerFg;

//egeszet visszaade es ket egeszet kapo fuggvenyre mutato mutatot ←
//visszaado, egeszet kapo fuggveny
A st = fuggveny2(10);

}
```

Mit vezetnek be a programba a következő nevek?

- `int a;`

Egy int típusú a nevű változó.

- `int *b = &a;`

b egészre mutató mutató a-ra mutat.

- `int &r = a;`

r egészre mutató mutató, ami a címet tartalmazza.

- `int c[5];`

Öt elemű egészkből álló tömb.

- `int (&tr)[5] = c;`

Öt elemű egészkből álló tombre mutató mutató, ami a c tömbre mutat.

- `int *d[5];`

5 elemű egészre mutató mutatókból álló tömb.

- ```
int *h ();
```

Egy egéssel visszatérő paraméter nélküli függvényre mutató mutató.

- ```
int *(*l) ();
```

Egy egészre mutató mutatóval visszatérő, paraméter nélküli függvényre mutató mutató.

- ```
int (*v (int c)) (int a, int b)
```

Egy egéssel visszatérő 2 egészet váró függvényre mutató mutatóval visszatérő 2 egészet váró függvény.

- ```
int (*(*z) (int)) (int, int);
```

Függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre mutató mutató.

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Chomsky/deklaracio.c

Tanulságok, tapasztalatok, magyarázat...

4. fejezet

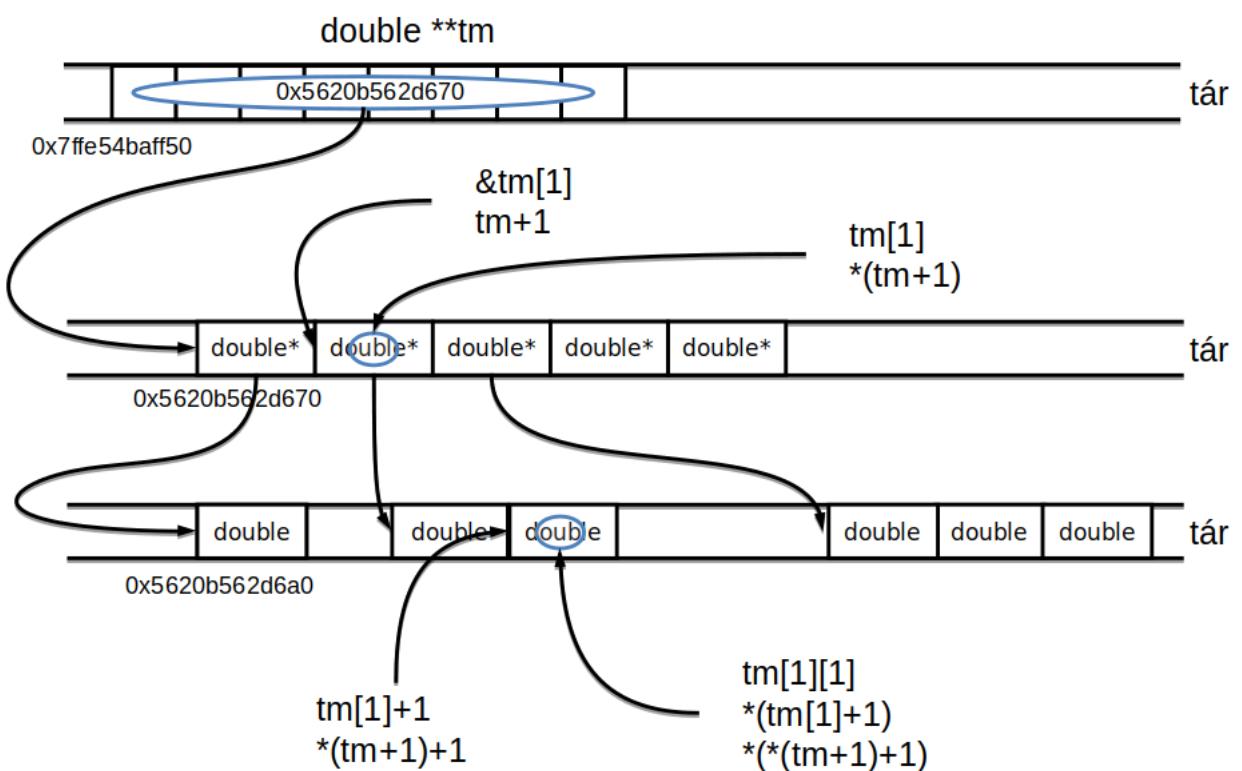
Helló, Caesar!

4.1. double ** háromszögmátrix

Írj egy olyan malloc és free párost használó C programot, amely helyet foglal egy alsó háromszög mátrixnak a szabad tárban!

Megoldás videó: <https://youtu.be/1MRTuKwRsB0>, <https://youtu.be/RKbX5-EWpzA>.

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Caesar/tm.c



4.1. ábra. A double ** háromszögmátrix a memóriában

```
#include <stdio.h>

#include <stdlib.h>
int
main ()
{
    int nr = 5;
    double **tm;
    if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
    {
        return -1;
    }
    for (int i = 0; i < nr; ++i)
    {
        if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL)
        {
            return -1;
        }
    }
    for (int i = 0; i < nr; ++i)
        for (int j = 0; j < i + 1; ++j)
            tm[i][j] = i * (i + 1) / 2 + j;
    for (int i = 0; i < nr; ++i)
    {
        for (int j = 0; j < i + 1; ++j)
            printf ("%f, ", tm[i][j]);
        printf ("\n");
    }
    for (int i = 0; i < nr; ++i)
    {
        for (int j = 0; j < i + 1; ++j)
            printf ("%f, ", tm[i][j]);
        printf ("\n");
    }
    for (int i = 0; i < nr; ++i)
        free (tm[i]);
    free (tm);
    return 0;
}
```

A képen a double ** háromszögmátrix memóriafoglalását bemutató ábrát láthatjuk. Az **nr** változóval megadjuk a háromszögmátrix magasságát. Jelen esetben ez 5 lesz. A **double ** tm** egy mutatóra mutató mutató. A **malloc**-al foglalunk memóriát ami egy mutatót ad vissza a lefoglalt területre (double * mutatót), amit double ** mutatóvá típuskényszerítjük, így a tm mutatónkat rá tudjuk állítani erre a tárterületre. A **malloc**-nak átadjuk mekkora területet foglaljon le. Ebben az esetben a double * mutató mérete, ami 8, szorozva az nr változóval, így 40 -et kapva ennyi bájtot foglalunk a memóriába.

Ezután egy for ciklussal végigmegyünk ezen a területen 1-5-ig és minden double * mutatót is ráállítunk

egy double típusú területre, aminek a darabszámát minden egyel növeljük, így az első mutató 1db double-re mutat, a második 2-re és így tovább haladva 5-ig, így kapjuk meg a háromszögmátrixot. Ezután értékeket adunk a változóinknak és kiiratjuk őket, majd felszabadítjuk a lefoglalt területeket.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Caesar$ ./tm
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
6.000000, 7.000000, 8.000000, 9.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
6.000000, 7.000000, 8.000000, 9.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
```

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Caesar/exor/exor.c

Ebben a feladatban a kizáró vagyos (XOR) titkosítás van bemutatva. A kizáró vagyos titkosításkor a titkosítandó szöveg bájtjait lefedjük a titkosító kulcs bájtjaival és egy kizáró vagy műveletet végzünk el rajtuk. A kizáró vagy művelet 1 értéket ad, ha a két bit különböző és 0-t ha megegyező. Például:

Kódolás:

A tiszta szöveg bájtja:	10011
A kulcs bájtja:	00110
A titkosított szöveg bájtjai XOR művelet után:	10101

Dekódolás:

A kulcs bájtja:	00110
A titkosított szöveg bájtjai:	10101
XOR művelet után visszakapjuk az eredeti szöveget:	10011

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#define MAX_KULCS 100
#define BUFFER_MERET 256
int
main (int argc, char **argv)
{
```

```
char kulcs[MAX_KULCS];
char buffer[BUFFER_MERET];
int kulcs_index = 0;
int olvasott_bajtok = 0;
int kulcs_meret = strlen(argv[1]);
strncpy(kulcs, argv[1], MAX_KULCS);
while ((olvasott_bajtok = read(0, (void *) buffer, BUFFER_MERET)))
{
    for (int i = 0; i < olvasott_bajtok; ++i)
    {
        buffer[i] = buffer[i] ^ kulcs[kulcs_index];
        kulcs_index = (kulcs_index + 1) % kulcs_meret;
    }
    write(1, buffer, olvasott_bajtok);
}
}
```

A program elején definiáljuk a kulcs maximum méretét és a beolvasáshoz szükséges buffer méretét. Ezután a main függvényben létrehozzuk az ezekkel a méretekkel rendelkező char típusú tömböket. A kulcs_index változó mutatja az aktuális elemét a kulcsnak, amivel majd végrehajtjuk a műveletet, az olvasott_bajtok pedig a beolvasott bájtok számát fogja tárolni. Az **strlen()** függvény segítségével rögzítjük a kulcs méretét amit a parancssorban adunk meg argumentumként. A while ciklus addig fut, amíg tudunk olvasni a bemenetről és azt tároljuk a bufferben, ha beolvasott szöveg végéhez értünk, akkor a **read()** függvény 0 értéket ad vissza és a ciklus véget ér. A while ciklusban végigmegyünk a beolvasott bájtokon és végrehajtjuk a titkosítást alkalmazva a kizáró vagy műveletet, majd az eredményt kiírjuk egy dokumentumba.

Fordítákor a c99 szabványt kell használnunk:

```
gcc exor.c -o e -std=c99
```

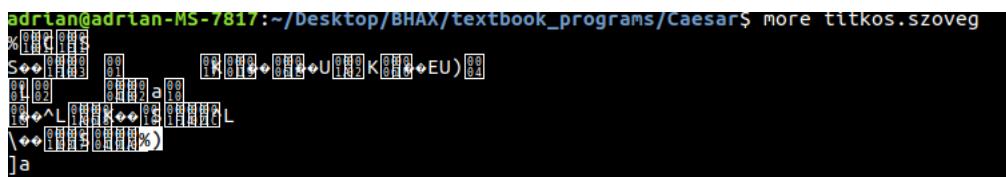
Eztúán létrehozunk egy szövegfájlt, amiben tároljuk a titkosítandó szövegünket

```
adrian@adrian-MS-7817:~/Desktop$ more tiszta.szoveg
Nem tudok kimerítő leírást adni arról, hogy hogyan tudsz megtanulni
programozni -- nagyon összetett tudásról van szó. Egyet azonban
elárulhatok: a könyvek és tanfolyamok nem érnak túl sokat (sok,
valószínűleg a legtöbb hacker autodidakta). Aminek van értelme:
(a) kódot olvasni és kódot írni.
```

Majd futtatjuk a programot megadva paraméterként a kulcsot és a tiszta.szoveg fájlunk tartalmát a programra irányítjuk (ezt olvassa be). A kimenő adatokat pedig a titkos.szoveg nevű fájlba irányítjuk. Ez lesz a titkosított szövegünk.

```
adrian@adrian-MS-7817:~/Desktop$ ./e kulcs <tiszta.szoveg >titkos.szoveg
```

Kiírva a titkos.szöveg tartalmát láthatjuk a kódolt szövegünket:



A dekódolás hasonlóan zajlik. Megadjuk azt a kulcsot amivel a kódolás történt és beolvassuk a titkos.szöveg tartalmat.

```
adrian@adrian-MS-7817:~/Desktop$ ./e kulcs <titkos.szöveg
Nem tudok kimerítő leírást adni arról, hogy hogyan tudsz megtanulni
programozni -- nagyon összetett tudásról van szó. Egyet azonban
elárulhatok: a könyvek és tanfolyamok nem érnek túl sokat (sok,
valószínűleg a legtöbb hacker autodidakta). Aminiek van értelme:
(a) kódot olvasni és kódot írni.
```

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása:https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Caesar/exor/ExorTitkos.java

```
public class ExorTitkosító {

    public ExorTitkosító(String kulcsSzöveg,
                          java.io.InputStream bejövőCsatorna,
                          java.io.OutputStream kimenőCsatorna)
        throws java.io.IOException {

        byte [] kulcs = kulcsSzöveg.getBytes();
        byte [] buffer = new byte[256];
        int kulcsIndex = 0;
        int olvasottBájtok = 0;
        while((olvasottBájtok =
               bejövőCsatorna.read(buffer)) != -1) {

            for(int i=0; i<olvasottBájtok; ++i) {

                buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]);
                kulcsIndex = (kulcsIndex+1) % kulcs.length;

            }

            kimenőCsatorna.write(buffer, 0, olvasottBájtok);
        }
    }
}
```

```
        }

    }

public static void main(String[] args) {
    try {
        new ExorTitkosító(args[0], System.in, System.out);
    } catch(java.io.IOException e) {
        e.printStackTrace();
    }
}
```

Javaban egy ExorTitkosító publikus osztályt vagy objektumot hozunk létre amiben a munkát végezzük. Először létrehozzuk a konstruktor, amivel majd átadjuk az objektumnak a kapott értékeket. Jelen esetben a paraméterként átadott kulcsot tároljuk egy stringbe, majd a bejövő és kijövő csatornát hozzuk létre. Egy byte tömben tároljuk a megadott kulcs bájtjait, illetve a buffer méretet. A while ciklusba addig olvasunk az int típusú beolvasottBájtok nevű változónkba, amíg -1-et nem kapunk, és egy for ciklussal iterálunk az olvasott bájtokon egyesével végezve a kizáró vagy műveletet a kulcsunk bájtjaival, amit tárolunk buffer tömbben. Végül a kimenő csatornán kiiratjuk a titkosított szöveget. A main() metódusban példányosítjuk az ExorTitkosító objektumunkat és átadjuk neki a paraméterként kapott kulcsot és a be és kimenő csatornát. Mindezt try és catch ágba téve, az esetleges hibák elkapásához.

4.4. C EXOR törő

Írj egy olyan C programot, amely megtöri az első feladatban előállított titkos szövegeket!

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Caesar/exor/t.c

```
int
main (void)
{
    char kulcs[KULCS_MERET];
    char titkos[MAX_TITKOS];
    char *p = titkos;
    int olvasott_bajtok;
```

```
while ((olvasott_bajtok =
    read (0, (void *) p,
    (p - titkos + OLVASAS_BUFFER <
     MAX_TITKOS) ? OLVASAS_BUFFER : titkos + MAX_TITKOS - p)))
    p += olvasott_bajtok;
// maradek hely nullazasa a titkos bufferben
for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
    titkos[p - titkos + i] = '\0';
// osszes kulcs eloallitasa
for (int ii = '0'; ii <= '9'; ++ii)
    for (int ji = '0'; ji <= '9'; ++ji)
        for (int ki = '0'; ki <= '9'; ++ki)
for (int li = '0'; li <= '9'; ++li)
    for (int mi = '0'; mi <= '9'; ++mi)
        for (int ni = '0'; ni <= '9'; ++ni)
            for (int oi = '0'; oi <= '9'; ++oi)
for (int pi = '0'; pi <= '9'; ++pi)
{
    kulcs[0] = ii;
    kulcs[1] = ji;
    kulcs[2] = ki;
    kulcs[3] = li;
    kulcs[4] = mi;
    kulcs[5] = ni;
    kulcs[6] = oi;
   kulcs[7] = pi;
if (exor_tores (kulcs, KULCS_MERET, titkos, p - titkos))
    printf
    ("Kulcs: [%c%c%c%c%c%c%c]\nTiszta szoveg: [%s]\n",
    ii, ji, ki, li, mi, ni, oi, pi, titkos);
    // ujra EXOR-ozunk, ily nem kell egy masodik buffer
    exor (kulcs, KULCS_MERET, titkos, p - titkos);
return 0;
}
```

Ez az exor törő program 0-9-ig számokat tartalmazó 8 számjegyű számmal kódolt szöveget töri fel. A program teljes egészében a forrásban látható. Hosszúsága miatt csak a main() függvényt szűrtam be. Hasonlóan az exor titkosítónál, itt is meg kell adni a max kulcs méretet, jelen esetben ez 8, illetve a titkos szövegünk max méretét. A while ciklusban beolvassuk a titkos szöveget, majd a titkos bufferben nullázzuk a maradék helyet. Ezután az összes lehetséges kulcsot előállítjuk for ciklusokkal 0-9-ig, közben alkalmazva a kulcsokon a kizáró vagy műveletét a titkos szöveggel és ha van találat akkor kiiratjuk a kulcsot és a már feltört szöveget, majd újra exorozunk, így nem kell új buffer.

Az exor titkosító programunkkal titkosítottunk egy szöveget a megfelelő kulcsot megadva, majd az exor törővel sikerült ezt feltörni, második találatra. Minél több különböző számot tartalmaz a kucsunk, annál tovább fog tartani a törés. Jelen esetben egy egyszerűbb kulccsal lett titkosítva a sz9veg, a törés gyorsabbá tétele miatt.

```

adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Caesar$ ./e 10000001 ←
    <tiszta.szoveg > titkos.szoveg
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Caesar$ ./t <titkos. ←
    szoveg
Kulcs: [10000000]
Tiszta szoveg: [Nem tudnk kimertő lerást 'dni arr$^3$1, hogx hogyan!tudsz ←
    mdgtanulnh
programozni --!nagyon lisszeteut tudárról vao szó. Dgyet aznnban
el;rulhatnk: a k$\div$nyvek és tanfomyamok ndm érnej túl snkat (soj,
valórzínúldg a leguöbb habker autndidakta(. Aminej van éstelme:
)a) kódnt olvasoi és k$^3$dot íjni.
]
Kulcs: [10000001]
Tiszta szoveg: [Nem tudok kimerítő leírást adni arról, hogy hogyan tudsz ←
    megtanulni
programozni -- nagyon összetett tudásról van szó. Egyet azonban
elárulhatok: a könyvek és tanfolyamok nem érnek túl sokat (sok,
valószínúleg a legtöbb hacker autodidakta). Aminek van értelme:
(a) kódot olvasni és kódot írni.

```

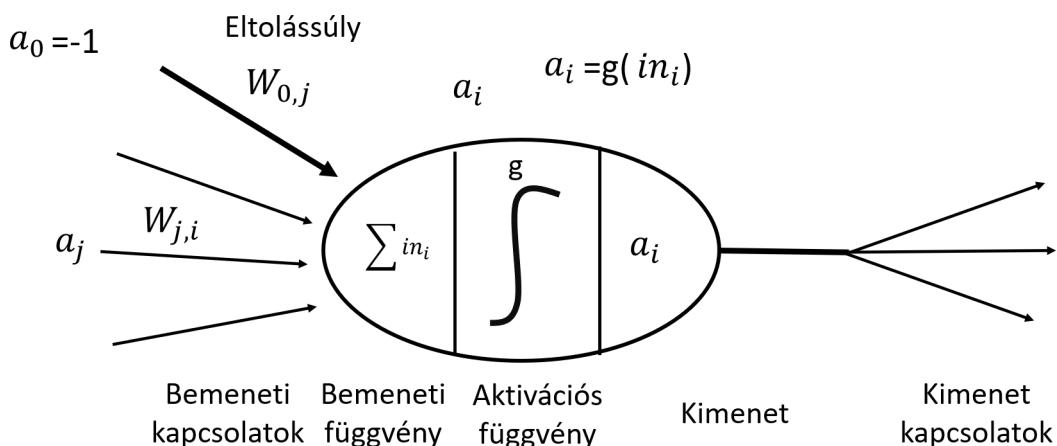
4.5. Neurális OR, AND és EXOR kapu

R

Megoldás videó: <https://youtu.be/Koyw6IH5ScQ>

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/attention_raising/NN_R/nn.r

A neurális háló vagy számítógépes idegháló az emberi agy által fogadott elektromos jelek feldolgozásán, összegyűjtésén és szétterjesztésén alapszik. Az ábrán látható az idegejt matematikai modellje:



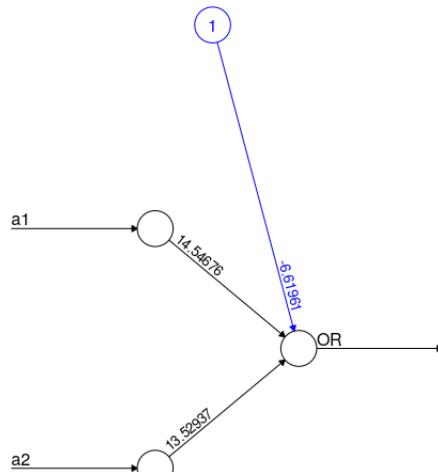
A bemeneten számok érkeznek amelyeknek van egy súlya, amit összeadunk és a neuron akkor fog "tüzelni", ha a bemeneti értékek súlyozott összege meghalad egy bizonyos köszöböt, ami a W_0 lesz -1 értékkel, ez az eltolássúly. Ezután egy g aktivációs függvényt alkalmaz a kapott értékre, majd tovább adja a függvény értékét.

```
library(neuralnet)
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
OR      <- c(0,1,1,1)
or.data <- data.frame(a1, a2, OR)
nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE, ←
    stepmax = 1e+07, threshold = 0.000001)
plot(nn.or)
compute(nn.or, or.data[,1:2])
```

R nyelvben dolgozunk, amihez telepíteni kell először a neuralnet csomagot. A a1 és a2-be berakjuk az értékeket és az OR-ban megadjuk mit kellene kapnia ha logikai vagy művelettel laklamzaunk a1 és a2-re, tehát megtanítjuk a szabályokat. Ez alapján elkezni magát tanítani a program, a súlyokat beállítja, majd a compute parancssal leelenőrizzük, hogy valóban a megadott értékeinkre helyes eredményt ad a program, amit megtanult, hogyan állítsan elő. Láthatjuk, hogy jó eredményeket kaptunk, illetve a jó eredményhez nagyon közelítő értékeket.

```
compute(nn.or, or.data[,1:2])
$neurons
$neurons[[1]]
  a1 a2
[1,] 1 0
[2,] 1 1
[3,] 1 0
[4,] 1 1
$net.result
  [,1]
[1,] 0.001332171
[2,] 0.999639318
[3,] 0.999002995
[4,] 1.000000000
```

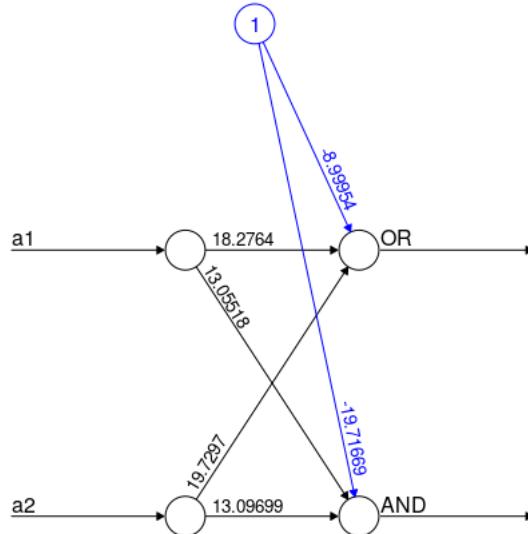
Illetve vizuálisan is láthatjuk a hálónkat a súlyokkal:



Error: 1e-06 Steps: 145

```
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
OR      <- c(0,1,1,1)
AND     <- c(0,0,0,1)
orand.data <- data.frame(a1, a2, OR, AND)
nn.orand <- neuralnet(OR+AND~a1+a2, orand.data, hidden=0, linear.output= FALSE,
                      stepmax = 1e+07, threshold = 0.000001)
plot(nn.orand)
compute(nn.orand, orand.data[,1:2])
```

Ebben az esetben már az és logikai műveletet is megtanítjuk a programnak, az előzővel megegyező módon.



Error: 3e-06 Steps: 188

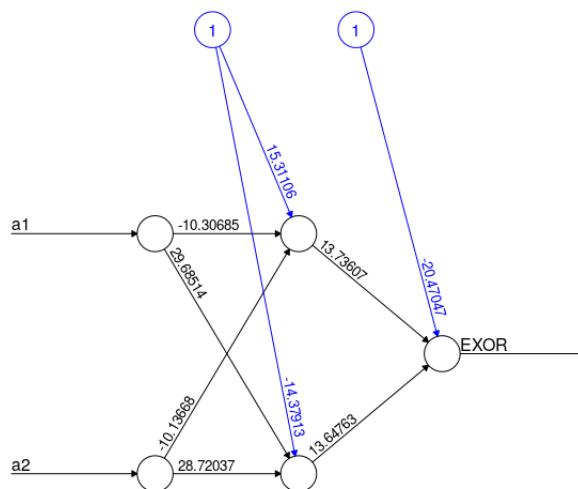
Az exor művelet megtanítása ugyanezzel az algoritmussal azonban már nem működik. Nem kapunk helyes eredményt, ezért ide egy más algoritmus fog kelleni.

```
compute(nn.or, or.data[,1:2])
$neurons[[1]]
  a1 a2
[1,] 1 0 0
[2,] 1 1 0
[3,] 1 0 1
[4,] 1 1 1
$net.result
  [,1]
[1,] 0.4999969
[2,] 0.5000000
[3,] 0.4999995
[4,] 0.5000025
```

```
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
EXOR   <- c(0,1,1,0)
exor.data <- data.frame(a1, a2, EXOR)
nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=2, linear.output=FALSE, ←
  stepmax = 1e+07, threshold = 0.000001)
plot(nn.exor)
compute(nn.exor, exor.data[,1:2])
```

A különbség ez előző algoritmusokhoz képest itt annyi, hogy több rejtett neuronnal dolgozunk, jelen esetben 2-vel, tehát több rétegű neuronokkal lehetséges a tanítás. A hidden értékét kell 2-re állítani. Láthatjuk az eredmény így már helyes lesz:

```
$neurons[ [1] ]
    a1 a2
[1,] 1 0 0
[2,] 1 1 0
[3,] 1 0 1
[4,] 1 1 1
$neurons[ [2] ]
[,1]      [,2]      [,3]
[1,] 1 0.99999978 5.691445e-07
[2,] 1 0.99333503 9.999998e-01
[3,] 1 0.99437209 9.999994e-01
[4,] 1 0.00586727 1.000000e+00
$net.result
[,1]
[1,] 0.001187882
[2,] 0.998911294
[3,] 0.998926671
[4,] 0.001178602
```



Error: 3e-06 Steps: 302

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:<https://youtu.be/XpBnR31BRJY>

Megoldás forrása: https://github.com/p-adrian05/BHAX/tree/master/textbook_programs/Caesar/perceptron

Ebben a feladatban a mandelbrot halmaz által generált kép rgb kódjait át tesszük a neurális háló inputjába, egy három rétegű hálót csinálunk és végül különböző számítások alapján kapunk a 3. rétegben egy számot.

```
#include <iostream>
#include "mlp.hpp"
#include "png++/png.hpp"
int main (int argc, char **argv)
{
    png::image<png::rgb_pixel> png_image (argv[1]);
    int size = png_image.get_width()*png_image.get_height();

    Perceptron* p = new Perceptron(3, size, 256, 1);
    double* image = new double[size];

    for(int i {0}; i<png_image.get_width(); ++i)
        for(int j {0}; j<png_image.get_height(); ++j)
            image[i*png_image.get_width()+j] = png_image[i][j].red;
    double value = (*p) (image);
    std::cout << value << std::endl;
    delete p;
    delete [] image;
}
```

Használjuk az mlp.hpp-t amiben a perceptron osztály van. Beimportáljuk az előállított képünket. A perceptron objektumnak foglalunk szabad területet. Megadjuk, hogy hány réteget használunk, a kép méretét, és hogy 1 szám legyen az eredmény. Lefoglalunk a képnek helyet, majd feltöljük for ciklusossal a paraméterben megadott képpel, végül meghívjuk a függvény operátort aminek átadjuk a képet. Végül kiiírjuk a függvény által visszaadott értéket.

Az alábbi módon fordítjuk:

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Caesar$ g++ mlp.hpp ←
main.cpp -o perc -lpng -std=c++11
```

Majd lefuttatva, átadva a képet megkapjuk az értéket:

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Caesar$ ./perc ←
mandel.png
0.585456
```

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

Megoldás videó: <https://youtu.be/gvaqijHIRUs>

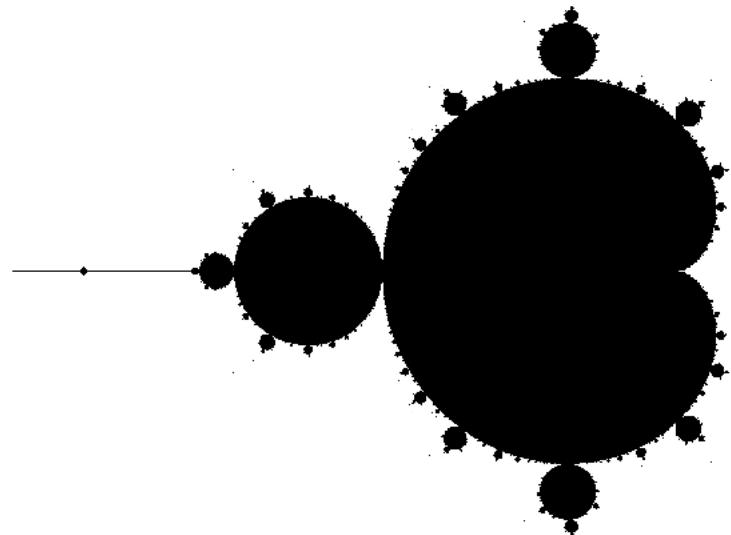
Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/attention_raising/CUDA/mandelpngt.c%2B

A Mandelbrot halmazt 1980-ban találta meg Benoit Mandelbrot a komplex számsíkon. Komplex számok azok a számok, amelyek körében válaszolni lehet az olyan egyébként értelmezhetetlen kérdésekre, hogy melyik az a két szám, amelyet összeszorozva -9-öt kapunk, mert ez a szám például a 3i komplex szám.

A Mandelbrot halmazt úgy láthatjuk meg, hogy a sík origója középpontú 4 oldalhosszúságú négyzetbe lefektetünk egy, mondjuk 800x800-as rácsot és kiszámoljuk, hogy a rács pontjai mely komplex számoknak felelnek meg. A rács minden pontját megvizsgáljuk a $z_{n+1} = z_n^2 + c$, ($0 \leq n$) képlet alapján úgy, hogy a c az éppen vizsgált rácpont. A z_0 az origó. Alkalmazva a képletet a

- $z_0 = 0$
- $z_1 = 0^2 + c = c$
- $z_2 = c^2 + c$
- $z_3 = (c^2 + c)^2 + c$
- $z_4 = ((c^2 + c)^2 + c)^2 + c$
- ... s így tovább.

Azaz kiindulunk az origóból (z_0) és elugrunk a rács első pontjába a $z_1 = c$ -be, aztán a c -től függően a további z -kbe. Ha ez az utazás kivezet a 2 sugarú körből, akkor azt mondjuk, hogy az a vizsgált rácpont nem a Mandelbrot halmaz eleme. Nyilván nem tudunk végtelen sok z -t megvizsgálni, ezért csak véges sok z elemet nézünk meg minden rácponthoz. Ha eközben nem lép ki a körből, akkor feketére színezzük, hogy az a c rácpont a halmaz része. (Színes meg úgy lesz a kép, hogy változatosan színezzük, például minél későbbi z -nél lép ki a körből, annál sötétebbre).



```
#include <stdio.h>
#include <iostream>
#include "png++/png.hpp"
#include <sys/times.h>

#define MERET 600
#define ITER_HAT 32000
void
mandel (int kepadat[MERET] [MERET]) {

    clock_t delta = clock ();
    struct tms tmsbuf1, tmsbuf2;
    times (&tmsbuf1);

    float a = -2.0, b = .7, c = -1.35, d = 1.35;
    int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;

    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, rez, imZ, ujrez, ujimZ;

    int iteracio = 0;

    for (int j = 0; j < magassag; ++j)
    {
```

```
for (int k = 0; k < szelesseg; ++k)
{
    reC = a + k * dx;
    imC = d - j * dy;
    // z_0 = 0 = (reZ, imZ)
    reZ = 0;
    imZ = 0;
    iteracio = 0;

    while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
    {
        // z_{n+1} = z_n * z_n + c
        ujreZ = reZ * reZ - imZ * imZ + reC;
        ujimZ = 2 * reZ * imZ + imC;
        reZ = ujreZ;
        imZ = ujimZ;

        ++iteracio;
    }
    kepadat[j][k] = iteracio;
}

times (&tmsbuf2);
std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
      + tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;

delta = clock () - delta;
std::cout << (float) delta / CLOCKS_PER_SEC << " sec" << std::endl;

}

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

    if (argc != 2)
    {
        std::cout << "Hasznalat: ./mandelpng fajlnev";
        return -1;
    }

    int kepadat[MERET][MERET];

    mandel(kepadat);

    png::image<png::rgb_pixel> kep (MERET, MERET);

    for (int j = 0; j < MERET; ++j)
    {
```

```
for (int k = 0; k < MERET; ++k)
{
    kep.set_pixel (k, j,
        png::rgb_pixel (255 -
        (255 * kepadat[j][k]) / ITER_HAT,
        255 -
        (255 * kepadat[j][k]) / ITER_HAT,
        255 -
        (255 * kepadat[j][k]) / ITER_HAT));
}
}

kep.write (argv[1]);
std::cout << argv[1] << " mentve" << std::endl;
}

}
```

A void metódusban számoltatjuk ki az adatokat a mandelbrot halmaz képi megjelenítéséhez. Előtte definiáljuk a kép méretét és az iterációs határértéket. Ezután megmérjük mennyi idő alatt történik a számolás, majd megadjuk a számításhoz szükséges adatokat. Két for ciklussal végigmegyünk a magasságon és szélességen és kiszámítjuk minden rács csomóponthoz tartozó komplex számot. A while ciklusban a $z_{n+1} = z_n^2 + c$ iterációkat számoljuk, míg a $|z_n|$ kisebb mint 2 vagy az iterációk száma nem érte el a 255-t, ha viszont elértek, akkor ez az jelenti, hogy a kiindulási c komplex számra az iteráció konvergens, tehát a c a mandelbrot halmaz eleme. Ezután kiiratjuk az eltelt időt, majd a main-ben összeállítjuk a képet.

Fordításkor használjuk a png könyvtárat a kép készítéséhez, majd futtatva megkapjuk a képet és az ehhez szükséges időtartamot.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/CUDA$ g++ mandelpngt ←
.c++ -lpng16 -o mandelpngt
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/CUDA$ ./mandelpngt t ←
.png
1894
18.9422 sec
t.png mentve
```

5.2. A Mandelbrot halmaz a `std::complex` osztállyal

Írj olyan C++ programot, amely kiszámolja a Mandelbrot halmazt!

Megoldás videó: <https://youtu.be/gvaqijHIRUs>

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/attention_raising/Mandelbrot/3.1.2.cpp

Tutorált: Országh Levente

```
#include <iostream>
#include "png++/png.hpp"
#include <complex>

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

    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double a = -1.9;
    double b = 0.7;
    double c = -1.3;
    double d = 1.3;

    if ( argc == 9 )
    {
        szelesseg = atoi ( argv[2] );
        magassag = atoi ( argv[3] );
        iteraciosHatar = atoi ( argv[4] );
        a = atof ( argv[5] );
        b = atof ( argv[6] );
        c = atof ( argv[7] );
        d = atof ( argv[8] );
    }
    else
    {
        std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c d ←
                     " << std::endl;
        return -1;
    }

    png::image< png::rgb_pixel > kep ( szelesseg, magassag );

    double dx = ( b - a ) / szelesseg;
    double dy = ( d - c ) / magassag;
    double reC, imC, reZ, imZ;
    int iteracio = 0;

    std::cout << "Szamitas\n";

    // j megy a sorokon
    for ( int j = 0; j < magassag; ++j )
    {
        // k megy az oszlopokon

        for ( int k = 0; k < szelesseg; ++k )
        {
```

```
// c = (reC, imC) a halo racspontjainak
// megfelelo komplex szam

reC = a + k * dx;
imC = d - j * dy;
std::complex<double> c ( reC, imC );

std::complex<double> z_n ( 0, 0 );
iteracio = 0;

while ( std::abs ( z_n ) < 4 && iteracio < iteraciosHatar )
{
    z_n = z_n * z_n + c;

    ++iteracio;
}

kep.set_pixel ( k, j,
                png::rgb_pixel ( iteracio%255, (iteracio*iteracio
                )%255, 0 ) );
}

int szazalek = ( double ) j / ( double ) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;
}

kep.write ( argv[1] );
std::cout << "\r" << argv[1] << " mentve." << std::endl;
}
```

C++-ban a komplex számokkal való számoláshoz importálni kell a komplex osztályt. Az előző programhoz hasonlóan itt is megadjuk a kép szélességét, magasságát, iterációs határt, számítás adatait, majd beolvassuk az argumentumként kapott adatokat és azokkal dolgozunk. A for ciklusokkal megyünk végig a háló sorain és oszlopain. A reC és imC a háló rácspontjait számoljuk és a rácspontokhoz tartozó komplex számokat a complex osztály segítségével. A while ciklus során, azt figyeljük, mennyire távolodik el a z_n a z_0 ponttól vagyis a koordinátarendszer középpontjától. Ha ez a távolság nagyobb lesz 2-nél, akkor a vizsgált c pontban az iteráció nem sűrűsödik be az origóhoz közel valamelyik pontban. Ha az iterációs harát elérése miatt lépünk ki a while ciklusból, akkor ebben a pontban az iteráció konvergens és a Mandelbrot halmaz elemének tekintjük, és feketére színezzük. Végül kimentjük a képet az argumentumban megadott névvel.

Fordításkor használjuk a png könyvtárat a kép készítéséhez. Futtatáskor pedig megadjuk az elkészült kép milyen nével jöjjön létre és a számítás adatait.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/Mandelbrot$ g++ ←
3.1.2.cpp -lpng -O3 -o 3.1.2
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/Mandelbrot$ ./3.1.2 ←
mandel.png 1920 1080 1020 0.4127655418209589255340574709407519549131 ←
0.4127655418245818053080142817634623497725 ←
```

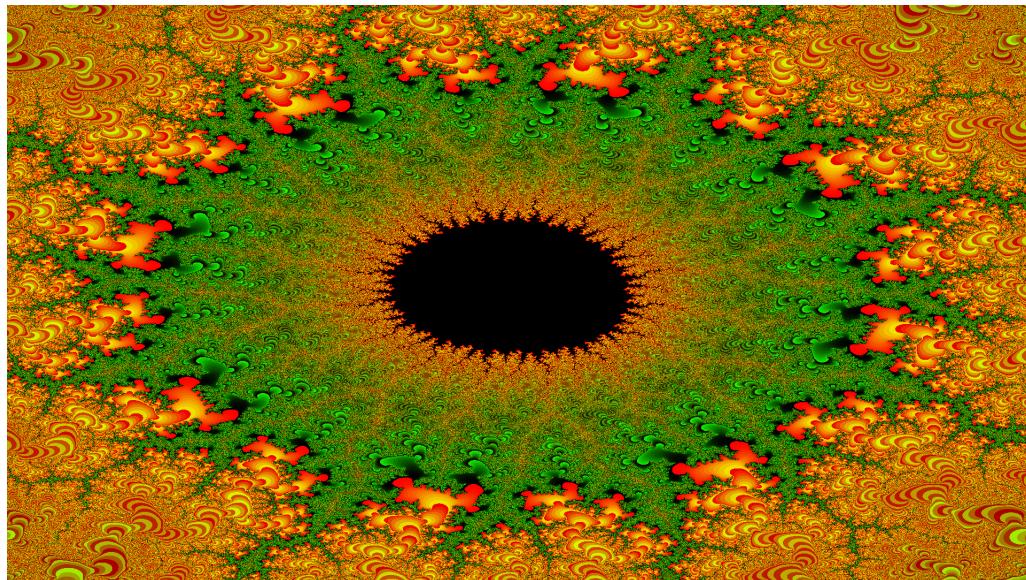
```
0.2135387051768746491386963270997512154281 ←  
0.2135387051804975289126531379224616102874
```

Számítás

mandel.png mentve.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/Mandelbrot$
```

Ezekkel az adatokkal ezt az eredményt kapjuk:

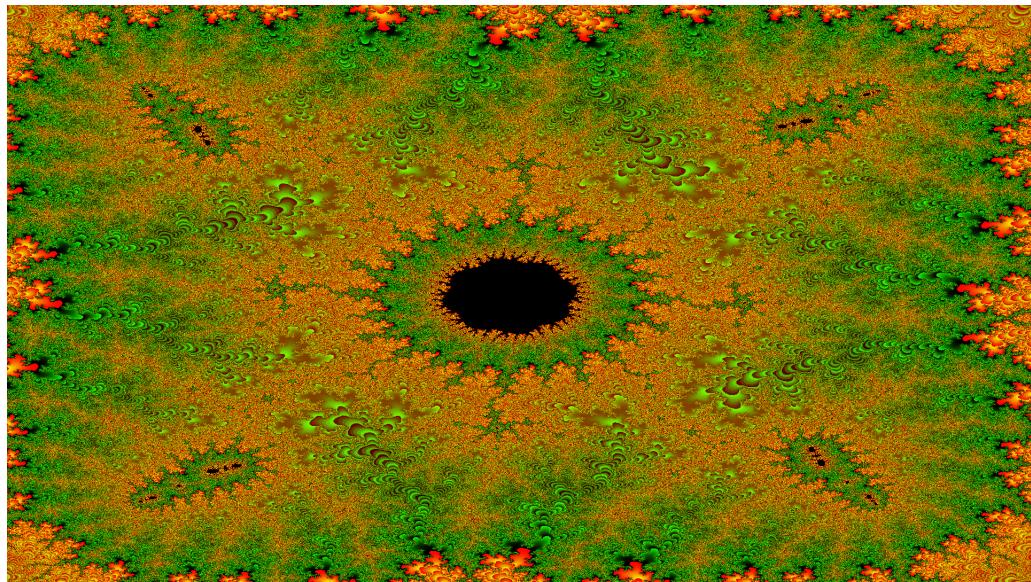


A következő adatokkal pedig ezt a képet kapjuk:

```
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/Mandelbrot$ ./3.1.2 ←  
mandel.png 1920 1080 2040 -0.01947381057309366392260585598705802112818 ←  
-0.0194738105725413418456426484226540196687 ←  
0.7985057569338268601555341774655971676111 ←  
0.79850575693437919611028519284445792436
```

Számítás

mandel.png mentve.



5.3. Biomorfok

Megoldás videó: <https://youtu.be/IJMbqRzY76E>

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/attention_raising/Biomorf/3.1.3.cpp

A biomorfokra (a Julia halmazokat rajzoló bug-os programjával) rátaláló Clifford Pickover azt hitte természeti törvényre bukkant: https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9_Iss5_2305--2315_Biomorphs_via_modified_iterations.pdf (lásd a 2307. oldal aljától).

A különbség a Mandelbrot halmaz és a Julia halmazok között az, hogy a komplex iterációban az előbbiben a c változó, utóbbiban viszont állandó. A következő Mandelbrot csipet azt mutatja, hogy a c befutja a vizsgált összes rácspontot, minden rácsponthoz más értéket rendelve.

```
// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
{
    for ( int k = 0; k < szelesseg; ++k )

        // c = (reC, imC) a halo racspontjainak
        // megfelelo komplex szam

        reC = a + k * dx;
        imC = d - j * dy;
        std::complex<double> c ( reC, imC );

        std::complex<double> z_n ( 0, 0 );
        iteracio = 0;

        while ( std::abs ( z_n ) < 4 && iteracio < iteraciosHatar )
        {
            z_n = z_n * z_n + c;
```

```
    ++iteracio;
}
```

Ezzel szemben a Julia halmazos csipetben a c nem változik, hanem minden vizsgált z rácpontra ugyanaz.

```
std::complex<double> c ( reC, imC );
// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
{
    // k megy az oszlopokon
    for ( int k = 0; k < szelesseg; ++k )
    {
        double rez = a + k * dx;
        double imZ = d - j * dy;
        std::complex<double> z_n ( rez, imZ );

        int iteracio = 0;
        for (int i=0; i < iteraciosHatar; ++i)
        {
            z_n = std::pow(z_n, 3) + cc;
            if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
            {
                iteracio = i;
                break;
            }
        }
    }
}
```

A bimorfos algoritmus pontos megismeréséhez ezt a cikket javasoljuk: https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9_Iss5_2305-2315_Biomorphs_via_modified_iterations.pdf. Az is jó gyakorlat, ha magából ebből a cikkből from scratch kódoljuk be a sajátunkat, de mi a királyi úton járva a korábbi Mandelbrot halmazt kiszámoló forrásunkat módosítjuk. Viszont a program változóinak elnevezését összhangba hozzuk a közlemény jelöléseivel:

```
#include <iostream>
#include "png++/png.hpp"
#include <complex>

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

    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double xmin = -1.9;
    double xmax = 0.7;
    double ymin = -1.3;
```

```
double ymax = 1.3;
double reC = .285, imC = 0;
double R = 10.0;

if ( argc == 12 )
{
    szelesseg = atoi ( argv[2] );
    magassag = atoi ( argv[3] );
    iteraciosHatar = atoi ( argv[4] );
    xmin = atof ( argv[5] );
    xmax = atof ( argv[6] );
    ymin = atof ( argv[7] );
    ymax = atof ( argv[8] );
    reC = atof ( argv[9] );
    imC = atof ( argv[10] );
    R = atof ( argv[11] );

}
else
{
    std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c ←
                  d reC imC R" << std::endl;
    return -1;
}

png::image< png::rgb_pixel > kep ( szelesseg, magassag );

double dx = ( xmax - xmin ) / szelesseg;
double dy = ( ymax - ymin ) / magassag;

std::complex<double> c ( reC, imC );

std::cout << "Szamitas\n";

// j megy a sorokon
for ( int y = 0; y < magassag; ++y )
{
    // k megy az oszlopokon

    for ( int x = 0; x < szelesseg; ++x )
    {

        double reZ = xmin + x * dx;
        double imZ = ymax - y * dy;
        std::complex<double> z_n ( reZ, imZ );

        int iteracio = 0;
        for (int i=0; i < iteraciosHatar; ++i)
        {
```

```
    z_n = std::pow(z_n, 3) + c;
    //z_n = std::pow(z_n, 2) + std::sin(z_n) + c;
    if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
    {
        iteracio = i;
        break;
    }
}

kep.set_pixel ( x, y,
                png::rgb_pixel ( (iteracio*20)%255, (iteracio ←
                    *40)%255, (iteracio*60)%255 ) );
}

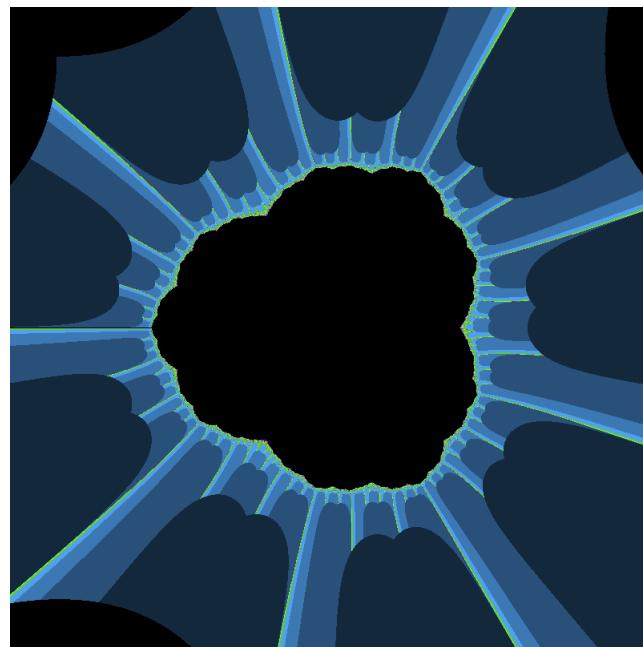
int szazalek = ( double ) y / ( double ) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;
}

kep.write ( argv[1] );
std::cout << "\r" << argv[1] << " mentve." << std::endl;
}
```

A c-t globális konstansként adjuk meg, ezért a rácspontokon való futás szerepét a z_n veszi át. A c értéket parancsorban adjuk át. Kiszámoljuk a most a z komplex számnak a valós részét és az imaginárius részét, majd ezekből előállítjuk az aktuális rácsponthoz tartozó komplex számot. A c valós része 0.285, amit megadunk állandónak, ahogy a cikkben szerepel az imaginárius rész pedig 0. A while ciklust for-ra cseréljük és ha ki kell lépni a rácspontból, miután kiugrik a kettő sugarú körből a z akkor kilépünk a ciklusból, de előtte elmentjük ezt a helyet. A feltételben, ahogy Pickover "hibázott" vagy művelettel nézzük z-nek a valós és képzetes részét, ha az egyik rész nagyobb mint az R, ami 10 lesz a cikk szerint. Ezután jön a színezés az iteráció változónak a felhasználásával.

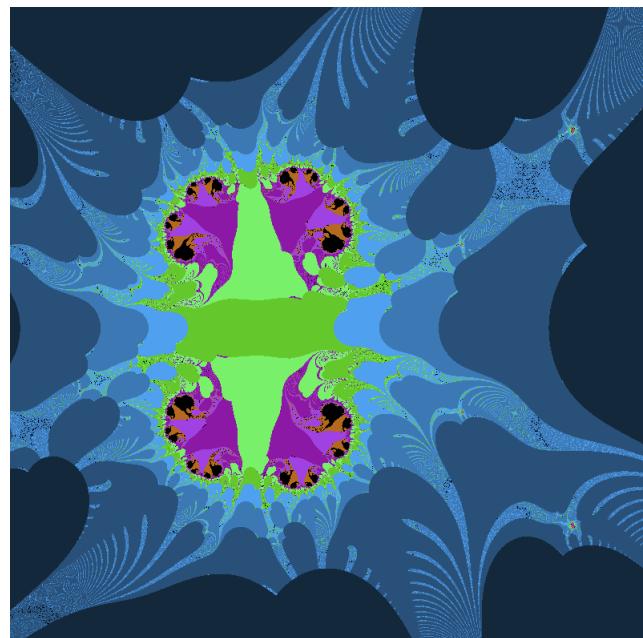
Fordításkor a használjuk a png könyvtárat és optimalizájuk 3. szinten, hogy gyors legyen. Futtatáskor megadjuk a globális változókat argumentumként.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/Biomorf$ g++ 3.1.3. ←
    cpp -lpng -O3 -o 3.1.3
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/Biomorf$ ./3.1.3 ←
    bmorf.png 800 800 10 -2 2 -2 2 .285 0 10
Szamitas
bmorf.png mentve..
```

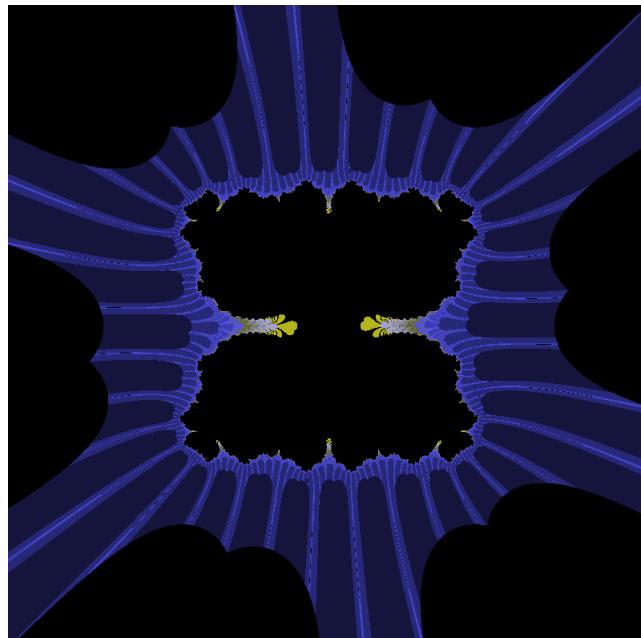


Most változtatjuk a függvényt és ugyanúgy fordítjuk, futtatjuk.

```
z_n = std::pow(z_n, 2) + std::sin(z_n) + c;
```

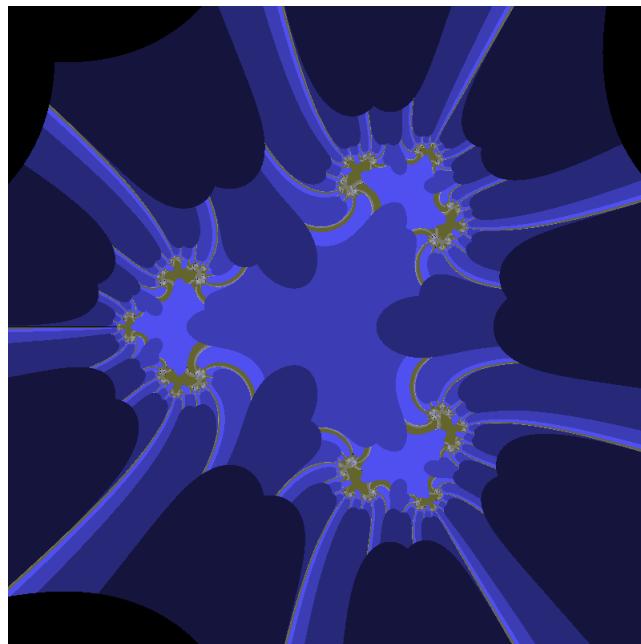


```
z_n = std::pow(z_n, 2) + std::pow(z_n, 6) + c;
```



```
z_n = std::pow(z_n, 3) + c;
```

És a c -t 1-re állítva:



5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó: <https://youtu.be/gvaqijHIRUs>

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/attention_raising/CUDA/mandelpngc_60x60.cu

A CUDA egy többszálas, párhuzamos feldolgozási platform és API, amit nvidia készített, ezért csak nvidia grafikus kártyákkal használható. Ennek segítségével tudunk számításokat végezni a GPU-val CPU helyett.

Az első feladatban a mandelbrot halmaz kiszámításához csak 1 szálon a CPU-t vettük igénybe ami közel 19 másodperct vett igénybe.

A CUDA-ban van egy rácsunk, amibe minden oszlopba és sorba berakunk 60db blokkot és blokkonként berakunk 100 szálat, így egyszerre fog számolni 60x60x100-at, 60db blokkban a 100 úgy jön ki, hogy 10x10. Mi esetünkben ki akarunk számolni egy 600x600 pixeles képet és ezt felhasználva minden pixelnek megfelel egy szál, így egyszerre számolódnak a pixelek ellentétben a CPU-s verziójánál, ahol 1 szálunk volt.

```
#include <png++/image.hpp>
#include <png++/rgb_pixel.hpp>

#include <sys/times.h>
#include <iostream>

#define MERET 600
#define ITER_HAT 32000

__device__ int
mandel (int k, int j)
{
    // Végigzongorázza a CUDA a szélesség x magasság rácsot:
    // most eppen a j. sor k. oszlopaban vagyunk

    // számítás adatai
    float a = -2.0, b = .7, c = -1.35, d = 1.35;
    int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;

    // a számítás
    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, rez, imZ, ujrez, ujimZ;
    // Hány iterációt csináltunk?
    int iteracio = 0;

    // c = (reC, imC) a rács csomópontjainak
    // megfelelő komplex szám
    reC = a + k * dx;
    imC = d - j * dy;
    // z_0 = 0 = (rez, imZ)
    rez = 0.0;
    imZ = 0.0;
    iteracio = 0;

    while (rez * rez + imZ * imZ < 4 && iteracio < iteraciosHatar)
    {
        // z_{n+1} = z_n * z_n + c
        ujrez = rez * rez - imZ * imZ + reC;
        ujimZ = 2 * rez * imZ + imC;
        rez = ujrez;
        imZ = ujimZ;
```

```
    ++iteracio;

}

return iteracio;
}

/*
__global__ void
mandelkernel (int *kepadat)
{

    int j = blockIdx.x;
    int k = blockIdx.y;

    kepadat[j + k * MERET] = mandel (j, k);

}
*/



__global__ void
mandelkernel (int *kepadat)
{

    int tj = threadIdx.x;
    int tk = threadIdx.y;

    int j = blockIdx.x * 10 + tj;
    int k = blockIdx.y * 10 + tk;

    kepadat[j + k * MERET] = mandel (j, k);

}

void
cudamandel (int kepadat[MERET] [MERET])
{

    int *device_kepadat;
    cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int));

    // dim3 grid (MERET, MERET);
    // mandelkernel <<< grid, 1 >>> (device_kepadat);

    dim3 grid (MERET / 10, MERET / 10);
    dim3 tgrid (10, 10);
    mandelkernel <<< grid, tgrid >>> (device_kepadat);

    cudaMemcpy (kepadat, device_kepadat,
```

```
        MERET * MERET * sizeof (int), cudaMemcpyDeviceToHost);
cudaFree (device_kepadat);

}

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

// Mérünk időt (PP 64)
clock_t delta = clock ();
// Mérünk időt (PP 66)
struct tms tmsbuf1, tmsbuf2;
times (&tmsbuf1);

if (argc != 2)
{
    std::cout << "Hasznalat: ./mandelpngc fajlnev";
    return -1;
}

int kepadat [MERET] [MERET];

cudamandel (kepadat);

png::image < png::rgb_pixel > kep (MERET, MERET);

for (int j = 0; j < MERET; ++j)
{
    //sor = j;
    for (int k = 0; k < MERET; ++k)
    {
        kep.set_pixel (k, j,
                      png::rgb_pixel (255 -
                                      (255 * kepadat [j] [k]) / ITER_HAT,
                                      255 -
                                      (255 * kepadat [j] [k]) / ITER_HAT,
                                      255 -
                                      (255 * kepadat [j] [k]) / ITER_HAT));
    }
}
kep.write (argv[1]);

std::cout << argv[1] << " mentve" << std::endl;

times (&tmsbuf2);
std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
+ tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;

delta = clock () - delta;
```

```
    std::cout << (float) delta / CLOCKS_PER_SEC << " sec" << std::endl;
}
```

A fordításhoz a következő, parancsot kell használni, nem a megszokott g++ fordítót:

```
nvcc mandelpngc_60x60_100.cu -lpng16 -O3 -o mandelpngc
```

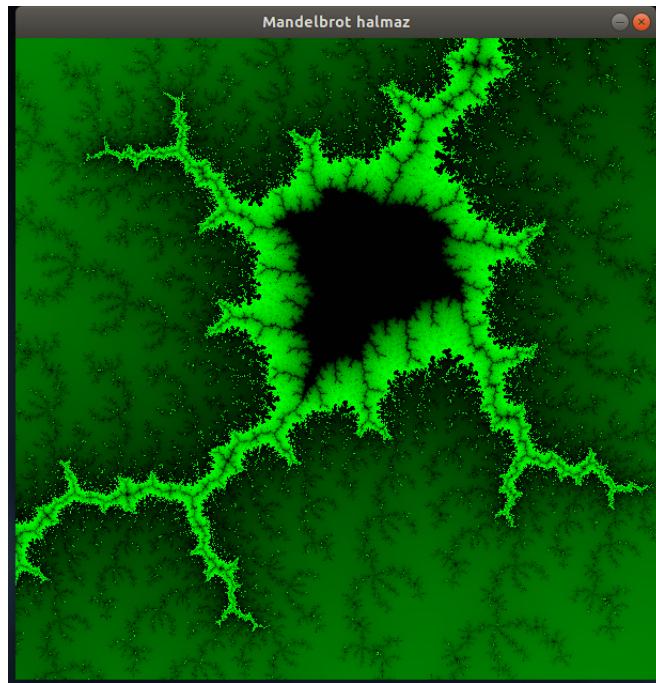
A program futtatásához nvidia kártyára van szükség, amivel nem rendelkezem, így nem tudjuk meg mennyivel lenne gyorsabb a CPU-hoz képest, de valahol 50-70 szeres gyorsulást lehet tapasztalni.

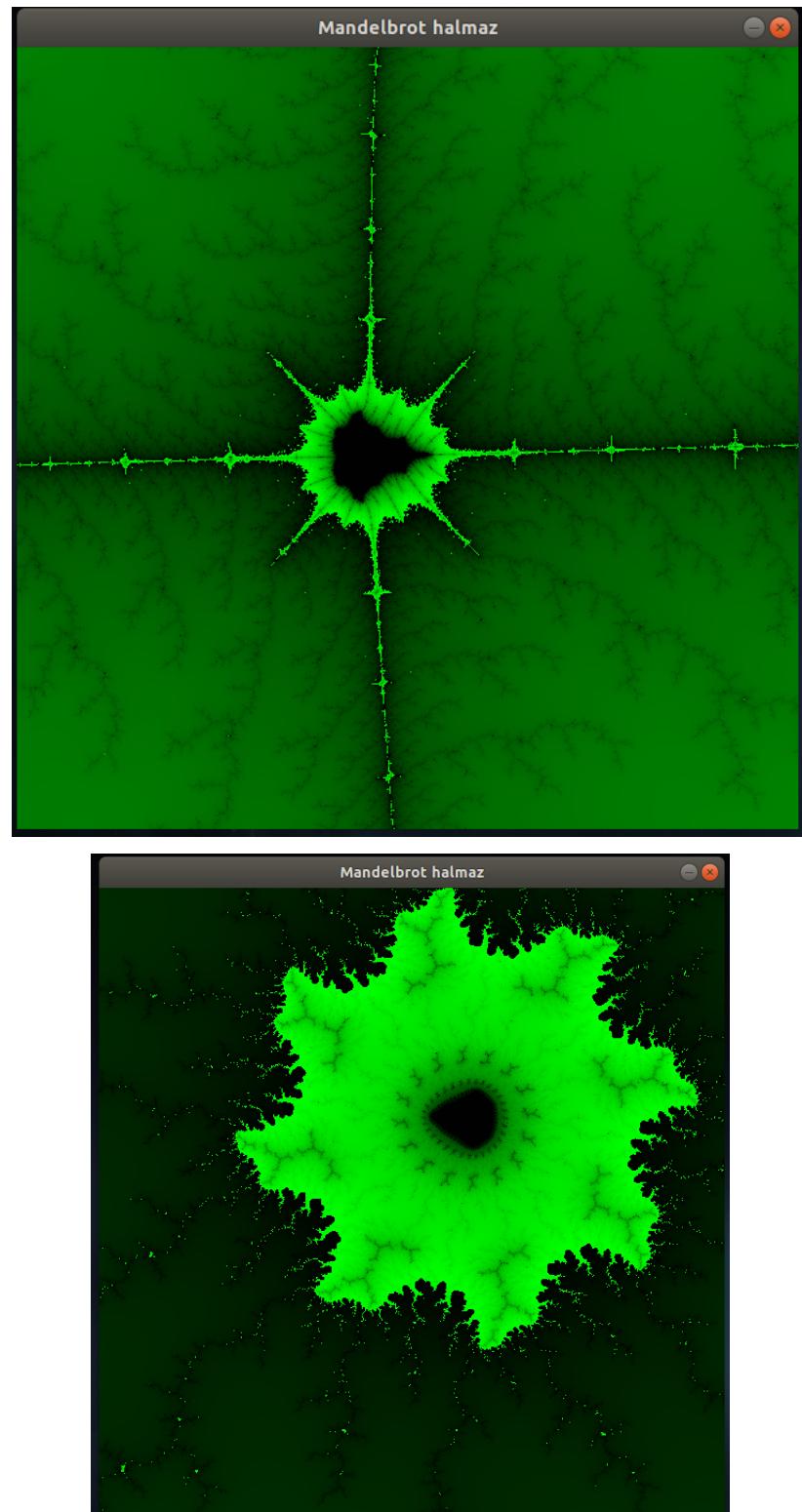
5.5. Mandelbrot nagyító és utazó C++ nyelven

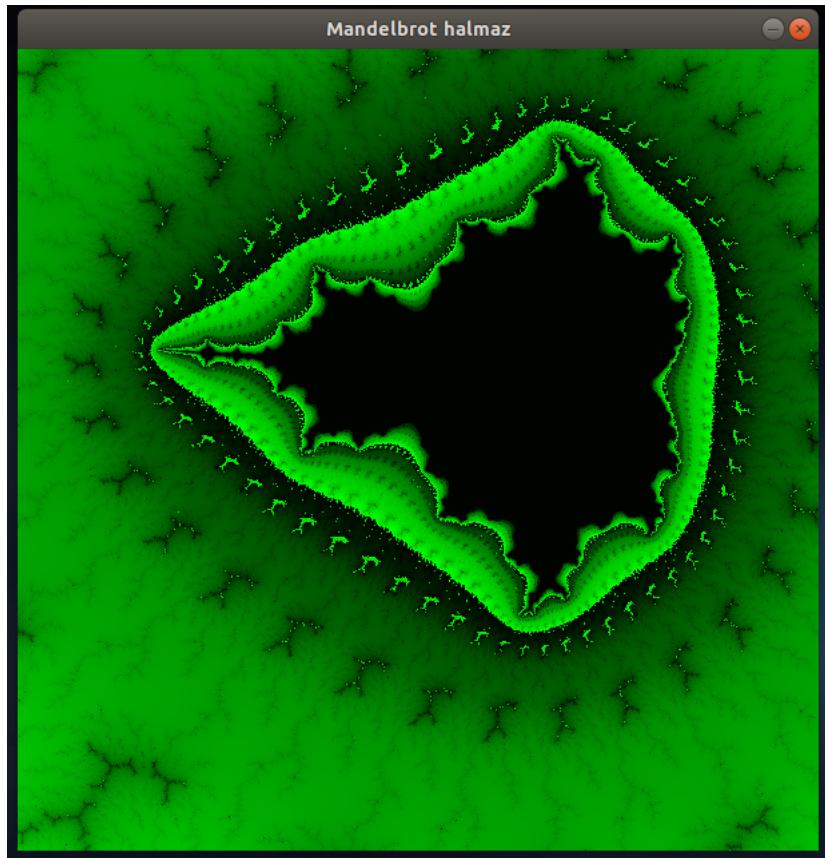
Építs GUI-t a Mandelbrot algoritmusra, lehessen egérrel nagyítani egy területet, illetve egy pontot egérrel kiválasztva vizualizálja onnan a komplex iteráció bejárta z_n komplex számokat!

Megoldás videó: Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_a_mandelbrot_halmazzal.

Megoldás forrása: https://github.com/p-adrian05/BHAX/tree/master/textbook_programs/Mandelbrot/c%2B%2B







Fordítás, futtatáshoz használjuk a Qt keretrendszer, azon belül a qmake-kel készítjük el a make fájlt.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Mandelbrot/c++ ←
_nagyito$ /home/adrian/Qt/5.12.2/gcc_64/bin/qmake frak.pro
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Mandelbrot/c++ ←
_nagyito$ make
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Mandelbrot/c++ ←
_nagyito$ ./frak
```

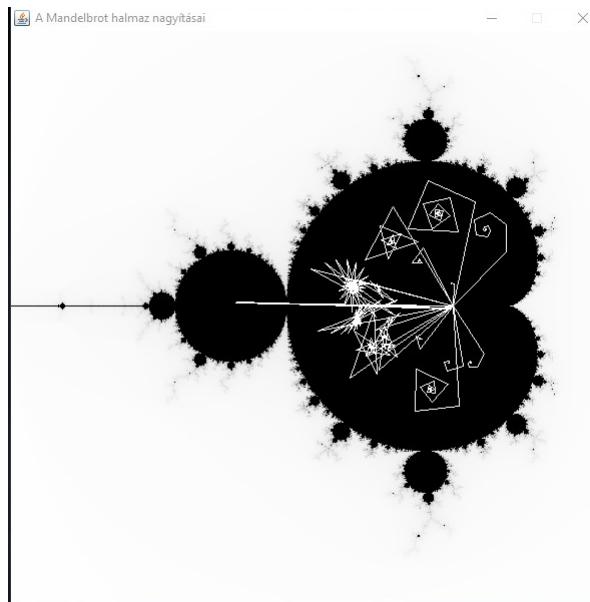
Ha a nagyítások során leromlott a pontosság, akkor az n gomb nyomásával növelni tudjuk a számítások iterációs határát.

5.6. Mandelbrot nagyító és utazó Java nyelven

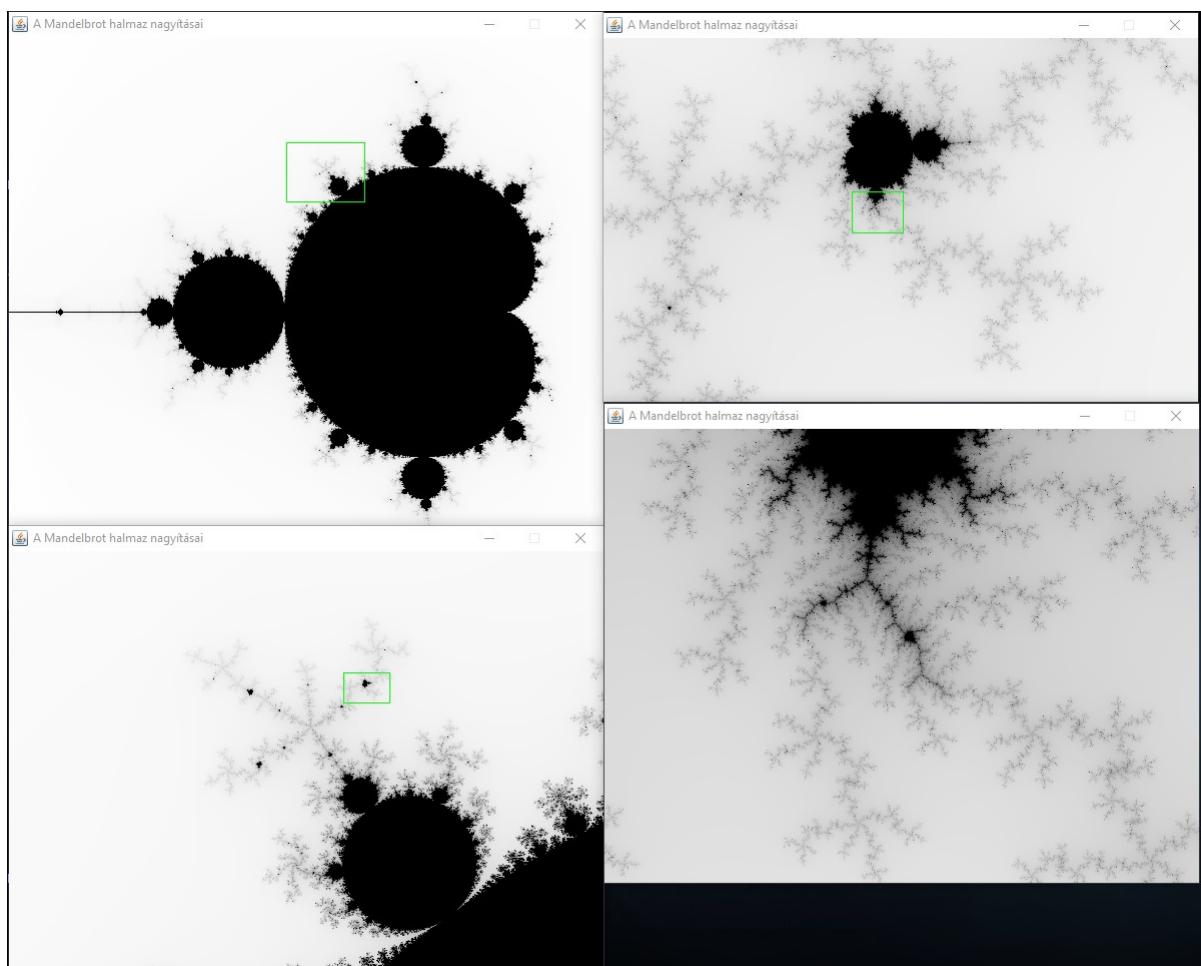
Megoldás videó: <https://youtu.be/Ui3B6IJnssY>, 4:27-től. Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_a_mandelbrot_halmaz_nagyitjanak_javaval/

Megoldás forrása: https://github.com/p-adrian05/BHAX/tree/master/textbook_programs/Mandelbrot/java_nagyitjanak

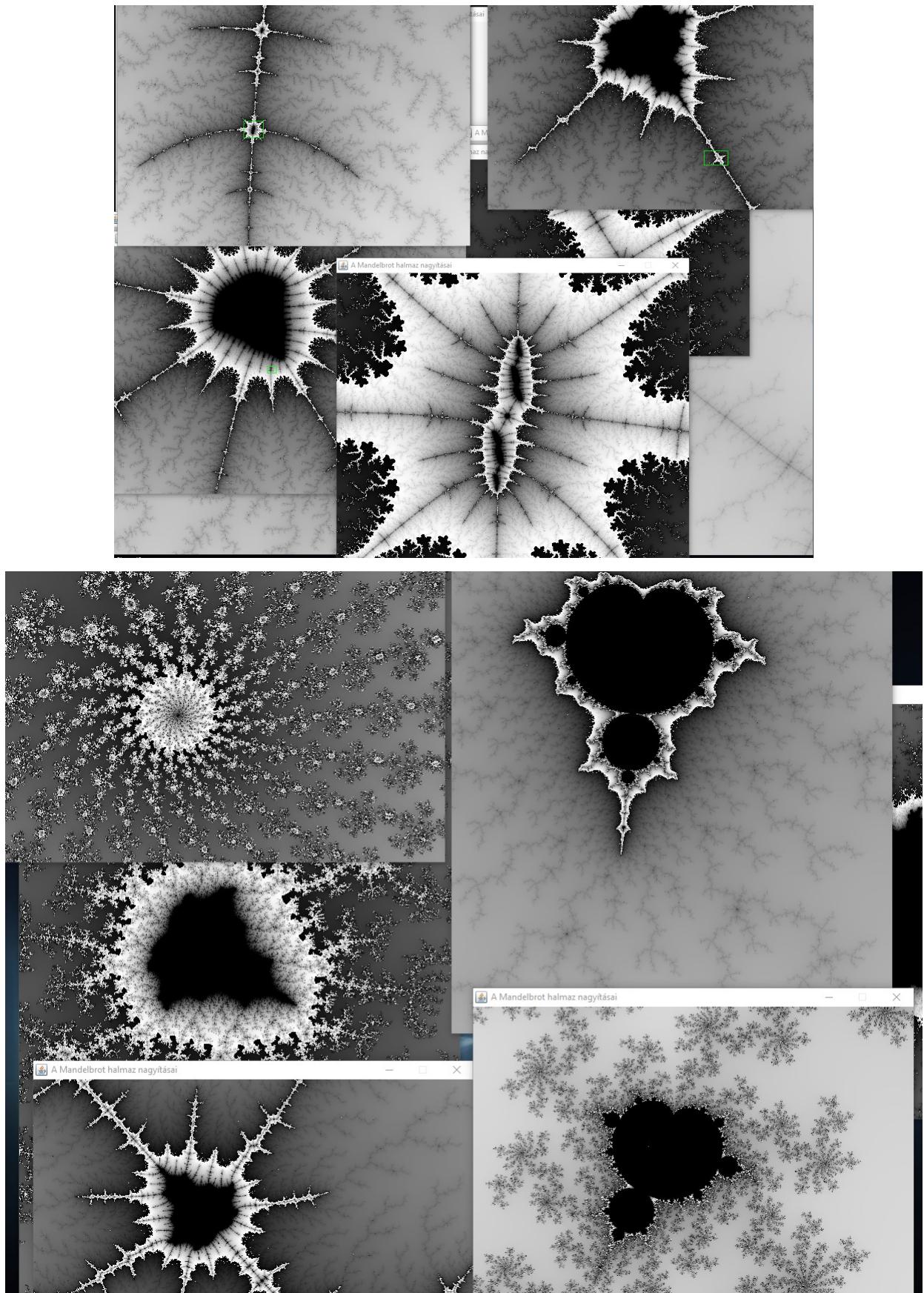
A következő képen a Mandelbrot halmaz utazó van bemutatva. A Mandelbrot halmazon belül egy pontra kattintva vizualizálja onnan a komplex iteráció bejárta z_n komplex számokat. Ezek azok a számok amelyek nem mennek ki a végtelenbe. A MandelbrotHalmazNagyító.java programot kell fordítani javac parancssal és futtatni a java parancssal, majd jobb klikkel a kiválasztva a pontot a mandelbrot halmazon belül.



Nagyításkor a program által kiszámolt Mandelbrot halmaz valamennyik részét nagyítjuk és annak is valamennyik részét és így tovább. Az egeren bal kíkérelt kiválasztani a nagyítás helyét.



Ha a nagyítások során leromlott a pontosság, akkor az n gomb nyomásával növelni tudjuk a számítások iterációs határát. A program futása alatt az s billentyűt lenyomva a kiszámolt halmazról egy felvételt készít a program, amit ugyanabban a könyvtárba ment ahol a program van,



6. fejezet

Helló, Welch!

6.1. Első osztályom

Valósítsd meg C++-ban és Java-ban az módosított polártranszformációs algoritmust! A matek háttér teljesen irreleváns, csak annyiban érdekes, hogy az algoritmus egy számítása során két normálist számol ki, az egyiket elspájzolod és egy további logikai taggal az osztályban jelzed, hogy van vagy nincs eltéve kiszámolt szám.

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/tree/master/textbook_programs/Welch/polargen

A Java és a C++ objektumorientált nyelvek. Objektumok használata egyszerűsíti a programozást, ugyanis a az objektum a valódi világ elemének a rá jellemző tulajdonságai és viselkedések által modelezett eleme. Erre jó példa a polártranszformációs algoritmus. Az algoritmus matematikai háttere most számunkra lényegtelen, fontos viszont az eljárás azon jellemzője, hogy egy számítási lépés két normális eloszlású számot állít elő, tehát minden páratlanadik meghíváskor nem kell számolnunk, csupán az előző lépés másik számát visszaadnunk. Hogy páros vagy páratlan lépéssben hívtuk-e meg a megfelelő számítást elvégző következő() függvényt, a nincsTárolt logikai változóval jelöljük. Igaz értéke azt jelenti, hogy tárolt lebegőpontos változóban el van tárolva a visszaadandó szám.

Java-ban:

```
public class PolárGenerátor {  
  
    boolean nincsTárolt = true;  
    double tárolt;  
  
    public PolárGenerátor() {  
  
        nincsTárolt = true;  
  
    }  
  
    public double következő() {
```

```
if(nincsTárolt) {  
  
    double u1, u2, v1, v2, w;  
    do {  
        u1 = Math.random();  
        u2 = Math.random();  
  
        v1 = 2*u1 - 1;  
        v2 = 2*u2 - 1;  
  
        w = v1*v1 + v2*v2;  
  
    } while(w > 1);  
  
    double r = Math.sqrt((-2*Math.log(w))/w);  
  
    tárolt = r*v2;  
    nincsTárolt = !nincsTárolt;  
  
    return r*v1;  
  
} else {  
    nincsTárolt = !nincsTárolt;  
    return tárolt;  
}  
}  
  
public static void main(String[] args) {  
  
    PolárGenerátor g = new PolárGenerátor();  
  
    for(int i=0; i<10; ++i)  
        System.out.println(g.következő());  
  
}
```

```
adrian@adrian-MS-7817:~/Desktop$ javac PolárGenerátor.java  
adrian@adrian-MS-7817:~/Desktop$ java PolárGenerátor  
-0.9977909475608935  
1.2436431306621076  
-1.0241389099503682  
0.9924334466411862  
0.36416475700814943  
0.9602245062934003
```

```
-0.5074463598559636
0.7797384696972893
-0.11174392973220601
0.8032167627641318
```

A JDK(Java Development Kit) Random.java osztályban a Sun programozói is lett megoldva a random szám generálás.

```
    ...
    */// generate a pair of sequential
public synchronized double nextGaussian() {
    // See Knuth, ACP, Section 3.4.1 Algorithm C.
    if (haveNextNextGaussian) {
        haveNextNextGaussian = false;
        return nextNextGaussian;
    } else {
        double v1, v2, s;
        do {
            v1 = 2 * nextDouble() - 1; // between -1 and 1
            v2 = 2 * nextDouble() - 1; // between -1 and 1
            s = v1 * v1 + v2 * v2;
        } while (s >= 1 || s == 0);
        double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
        nextNextGaussian = v2 * multiplier;
        haveNextNextGaussian = true;
        return v1 * multiplier;
    }
}

// stream methods, coded in a way intended to better isolate for
// maintenance purposes the small differences across forms.

/**

```

C++-ban:

```
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <iostream>
class PolarGen
{
public:
    PolarGen ()
    {
        nincsTarolt = true;
        std::srand (std::time (NULL));
    }
    ~PolarGen ()
    {

    }
double kovetkezo ()
{
    if (nincsTarolt)
    {
        double u1, u2, v1, v2, w;
```

```
    do
    {
        u1 = std::rand () / (RAND_MAX + 1.0);
        u2 = std::rand () / (RAND_MAX + 1.0);
        v1 = 2 * u1 - 1;
        v2 = 2 * u2 - 1;
        w = v1 * v1 + v2 * v2;
    }
    while (w > 1);

    double r = std::sqrt ((-2 * std::log (w)) / w);

    tarolt = r * v2;
    nincsTarolt = !nincsTarolt;

    return r * v1;
}
else
{
    nincsTarolt = !nincsTarolt;
    return tarolt;
}
}

private:
    bool nincsTarolt;
    double tarolt;
};

int main (int argc, char **argv)
{
    PolarGen pg;

    for (int i = 0; i < 10; ++i)
        std::cout << pg.kovetkezo () << std::endl;

    return 0;
}
```

```
adrian@adrian-MS-7817:~/Desktop$ g++ polargen.cpp -o p
adrian@adrian-MS-7817:~/Desktop$ ./p
0.33827
0.159026
1.07617
-0.573861
0.407831
0.337813
1.1272
1.33273
```

```
-1.67664  
0.401283
```

6.2. LZW

Valósítsd meg C-ben az LZW algoritmus fa-építését!

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Welch/z.c

Az LZW algoritmus fa struktúrában ábrázolja a beérkező bináris adatokat. Az input adatok feldolgozása során a gyökértől indulva, addig követjük a fa ágait, amíg egy olyan részsstringhez nem érünk, amely már nincs benne a fában. Ekkor a részsstring utolsó karakterével, ami éppen feldolgozás alatt van, bővítiük a fát. A következő inputtal egy új részsstringet indítunk és újra a gyökértől próbáljuk meg illeszteni az inputot. Az algoritmus teljes neve Lempel-Ziv-Welch , amit Abraham Lempel, Jacob Ziv, és Terry Welch dolgozott ki, ami egy veszteségmentes tömörítő algoritmus. Ezt az algoritmust használja a gif formátum, illetve sok tömörítő program is, pl compress, gzip, zip.

A program legelején először is létrehozunk egy binfa struktúrát. Ezzel létrehozunk egy binfa típust, amiben deklarálunk egy egész típusú változót, ebben tároljuk majd a beérkező adatot. Ezután a fa bal és jobb oldali mutatóját adjuk meg, ami a gyökér jobb illetve bal gyerekére fog mutatni, majd létrehozzunk a binfa típusra mutató mutatót. Az uj_elem() függvényben memória területet foglalunk le a mutatónak. A main függvényben dolgozzuk fel a beérkező adatokat. Létrehozzuk a gyoker mutatót, amit ráállítunk a neki lefoglalt memória területre, majd ennek megadjuk az értékét, ami '/' lesz mivel ez lesz a fő, első gyökér. Ezután létrehozunk egy fa mutatót. Következik az adatok beolvasása a inputból és megnézzük a beolvasott értéket, hogy ha 0, akkor megvizsgáljuk,hogy a fa mutató bal gyereke null értékű-e , tehát nincs gyereke. Ebben az esetben a fa mutató bal gyerekének lefoglalunk helyet és az értékét 0-ra állítjuk. Ezután ennek az új gyermeknek a jobb és bal mutatóját állítjuk null-ra és a fa mutatóját ráállítjuk a az új gyökérre. Ha pedig a fa bal gyermeke nem null-ra mutat, akkor a fát ráállítjuk a bal gyermekére, vagyis az lesz az aktuális gyökér. 1 érték esetén ugyanez játszódik le, csak a jobb oldali elemeket vizsgáljuk. Végül kiiratjuk az elkészült fánkat, végigjárva inorder bejárással, közben számolva a fa magasságát, majd felszabadítva a memóriát.

```
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
typedef struct binfa  
{  
    int ertek;  
    struct binfa *bal_nulla;  
    struct binfa *jobb_egy;  
} BINFA, *BINFA_PTR;  
  
BINFA_PTR
```

```
uj_elem ()
{
    BINFA_PTR p;

    if ((p = (BINFA_PTR) malloc (sizeof (BINFA))) == NULL)
    {
        perror ("memoria");
        exit (EXIT_FAILURE);
    }
    return p;
}

extern void kiir (BINFA_PTR elem);
extern void szabadit (BINFA_PTR elem);

int
main (int argc, char **argv)
{
    char b;

    BINFA_PTR gyoker = uj_elem ();
    gyoker->ertek = '/';
    BINFA_PTR fa = gyoker;

    while (read (0, (void *) &b, 1))
    {
        write (1, &b, 1);
        if (b == '0')
        {
            if (fa->bal nulla == NULL)
            {
                fa->bal nulla = uj_elem ();
                fa->bal nulla->ertek = 0;
                fa->bal nulla->bal nulla = fa->bal nulla->jobb_egy = NULL;
                fa = gyoker;
            }
            else
            {
                fa = fa->bal nulla;
            }
        }
        else
        {
            if (fa->jobb_egy == NULL)
            {
                fa->jobb_egy = uj_elem ();
                fa->jobb_egy->ertek = 1;
                fa->jobb_egy->bal nulla = fa->jobb_egy->jobb_egy = NULL;
                fa = gyoker;
            }
        }
    }
}
```

```
else
{
    fa = fa->jobb_egy;
}
}

printf ("\n");
kiir (gyoker);
extern int max_melyseg;
printf ("melyseg=%d", max_melyseg);
szabadit (gyoker);
}

static int melyseg = 0;
int max_melyseg = 0;

void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;
        kiir (elem->jobb_egy);
        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ↔
               ,
               melyseg);
        kiir (elem->bal nulla);
        --melyseg;
    }
}

void
szabadit (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        szabadit (elem->jobb_egy);
        szabadit (elem->bal nulla);
        free (elem);
    }
}
```

Fordításkor használjuk az c99 szabványt, majd futtatáskor az adat.txt-ben lévő adatokat irányítjuk be a programba.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Welch$ gcc z.c -o z ←
    -std=c99
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Welch$ ./z <adat.txt
000111011101101100001110

-----1 (4)
-----1 (3)
-----1 (5)
-----0 (4)
----1 (2)
----0 (3)
---/(1)
----1 (3)
----0 (2)
----0 (3)
----0 (4)
```

6.3. Fabejárás

Járd be az előző (inorder bejárású) fát pre- és posztorder is!

Megoldás videó:

Megoldás forrása:

A fa adatstruktúrákat három féleképpen lehet bejárni:inorder,postorder és preorder módon.

A bejárások előtt először megnézzük, hogy a bejárandó fa nem üres. Inorder esetben először végigjárjuk a gyökérelem jobb oldali részfáját, feldolgozzuk a gyökérelemet majd végigmegyünk a bal oldali részfán.

```
void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;

        kiir (elem->jobb_egy);

        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ←
               ,
```

```
    melyseg);  
  
    kiir (elem->bal_nulla);  
  
    --melyseg;  
}  
}
```

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Welch$ gcc z.c -o z ←  
-std=c99  
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Welch$ ./z <adat.txt  
000111011101101100001110  
  
-----1(4)  
----1(3)  
-----1(5)  
-----0(4)  
----1(2)  
----0(3)  
---/(1)  
-----1(3)  
----0(2)  
----0(3)  
-----0(4)
```

Preorder bejárásnál először dolgozzuk fel a gyökérelemet, majd bejárjuk a gyökérelem bal oldali részfáját, majd ezután a jobb oldali részfáját.

```
void  
kiir (BINFA_PTR elem)  
{  
    if (elem != NULL)  
    {  
        ++melyseg;  
        if (melyseg > max_melyseg)  
            max_melyseg = melyseg;  
        for (int i = 0; i < melyseg; ++i)  
            printf ("---");  
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ←  
               '  
               melyseg);  
  
        kiir (elem->bal_nulla);
```

```
    kiir (elem->jobb_egy);

    --melyseg;
}
}
```

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Welch/test$ ./z < ↵
adat.txt
000111011101101100001110

---/ (1)
-----0 (2)
-----0 (3)
-----0 (4)
-----1 (3)
-----1 (2)
-----0 (3)
-----1 (3)
-----0 (4)
-----1 (5)
-----1 (4)
```

Postorder bejárás pedig a preorder ellentetje, tehát ekkor a gyökérelem bal oldali részfáját járjuk be először, utána a jobb oldalit és ezután dolgozzuk fel a gyökérelemet.

```
void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;

        kiir (elem->bal nulla);
        kiir (elem->jobb_egy);

        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ←
                ,
                melyseg);
```

```
--melyseg;
}
}
```

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Welch/test$ ./z < ←
adat.txt000111011101101100001110

-----0 (4)
-----0 (3)
-----1 (3)
-----0 (2)
-----0 (3)
-----1 (5)
-----0 (4)
-----1 (4)
-----1 (3)
-----1 (2)
---/(1)
```

6.4. Tag a gyökér

Az LZW algoritmust ültessd át egy C++ osztályba, legyen egy Tree és egy beágyazott Node osztálya. A gyökér csomópont legyen kompozícióban a fával!

Megoldás videó:

Megoldás forrása:https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Welch/z3a7.cpp

```
class LZWBInFa
{
public:
    LZWBInFa () :fa (&gyoker)
    {
    }
    ~LZWBInFa ()
    {
        szabadit (gyoker.egyesGyermek ());
        szabadit (gyoker.nullasGyermek ());
    }

    void operator<< (char b)
    {
```

```
if (b == '0')
{
    if (!fa->nullasGyermek ())
    {

        Csomopont *uj = new Csomopont ('0');
        fa->ujNullasGyermek (uj);
        fa = &gyoker;
    }
    else
    {

        fa = fa->nullasGyermek ();
    }
}

else
{
    if (!fa->egyesGyermek ())
    {
        Csomopont *uj = new Csomopont ('1');
        fa->ujEgyesGyermek (uj);
        fa = &gyoker;
    }
    else
    {
        fa = fa->egyesGyermek ();
    }
}
}
```

Ebben az osztályban a fa gyökere nem mutató, hanem egy objektum, a fa a mutató, ami mindenkor az épülő fa aktuális csomópontjára mutat. Az LZWBinFa konstruktor ráállítja a fa mutatót a gyökérre, a dekonstruktőrben pedig felszabadítjuk a gyoker gyerekeit. Tagfüggvényként túlterheljük a

<<

operátort és az inputot eszerint tesszük a fába. Ha a bemenő karakter 0, akkor megnezzük, hogy az aktuális csomópontnak van-e 0-ás gyermekje, tehát a fa mutató éppen rá mutat-e. Ha nincs, akkor példányosítjuk a '0' betűt, új csomópontot hozunk étre. Ezután az aktuális csomópont nullás gyermekét ráállítjuk az új csomópontjára és a fával visszaállunk a gyökérre. Ha van nullás gyermek, akkor a fa mutatót ráállítjuk. '1' es karakter esetén ugyanez játszódik le.

```
private:
class Csomopont
{
```

```
public:  
    Csomopont (char b = '/') : betu (b), balNulla (0), jobbEgy (0)  
    {  
    };  
    ~Csonopont ()  
    {  
    };  
  
    Csonopont * nullasGyermek () const  
    {  
        return balNulla;  
    }  
  
    Csonopont * egyesGyermek () const  
    {  
        return jobbEgy;  
    }  
  
    void ujNullasGyermek (Csonopont * gy)  
    {  
        balNulla = gy;  
    }  
  
    void ujEyesGyermek (Csonopont * gy)  
    {  
        jobbEgy = gy;  
    }  
  
    char getBetu () const  
    {  
        return betu;  
    }  
  
private:  
    char betu;  
  
    Csonopont * balNulla;  
    Csonopont * jobbEgy;  
  
    Csonopont (const Csonopont &); // másoló konstruktor  
    Csonopont & operator= (const Csonopont &);  
};
```

A beágyazott Csonopont osztály privát, hogy csak az LZWBinFa osztályon belül tudjuk elérni. A paraméter nélküli konstruktor az alapértelmezett '/' betűvel hozza létre a csomópontot. Ilyet hívunk a fából, aki tagként tartalmazza a gyökeret, mert ha más betűvel hívjuk meg, akkor azt teszi a betű tagba, a két gyermekre mutató mutatót pedig nullra állítjuk. Ezután függvények következnek, amik visszaadják az aktuális csomópont jobb és bal gyermekét. A metódusok pedig arra szolgálnak, hogy az aktuális csomópont

akutális gyerekeit ráállítsák az új csomópontokra. Végül létrehozzuk a privát változókat, mutatókat és a másoló konstruktort, amit letiltunk.

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

    if (argc != 4)
    {
        usage ();
        return -1;
    }

    char *inFile = *++argv;
    if ((*((++argv) + 1) != 'o'))
    {
        usage ();
        return -2;
    }
    std::fstream beFile (inFile, std::ios_base::in);
    if (!beFile)
    {
        std::cout << inFile << " nem létezik..." << std::endl;
        usage ();
        return -3;
    }

    std::fstream kiFile (*++argv, std::ios_base::out);

    unsigned char b;
    LZWBinFa binFa;
    while (beFile.read ((char *) &b, sizeof (unsigned char)))
    {
        for (int i = 0; i < 8; ++i)
        {
            if (b & 0x80)
                binFa << '1';
            else
                binFa << '0';
            b <<= 1;
        }

        kiFile << binFa;
        kiFile << "depth = " << binFa.getMelyseg () << std::endl;
        kiFile << "mean = " << binFa.getAtlag () << std::endl;
        kiFile << "var = " << binFa.getSzoras () << std::endl;

        kiFile.close ();
    }
}
```

```
beFile.close ();
return 0;
}
```

A main függvényben először ellenőrizzük, hogy helyen bemenetet kaptunk parancssorból, majd létrehozzuk a binFa objektumot és a b változóba olvassuk be bejövő fájl bájtjait. Ezután következik az adatok beolvasása fájlból, amit binárisan olvasunk be, for ciklussal megnézzük egyenként a b-ben lévő bájt bitjeit. Ha a végigmegyünk a biteken és végig 0 lesz, kivéve a vége az 0 vagy 1, akkor aszerint tesszük a fába az '1' vagy '0' betűt. Végül a kimeneti csatornába egy fájlba kiirjuk a fát, illetve párt adatot a megépült fáról: magasság, szórás, átlag.

Futtatáskor meg kell adni egy fájlt amiből olvas és a kimenő fájl nevét, amiben a fát írja ki.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Welch$ g++ z3a7.cpp ←
-o fa
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Welch$ ./fa adat.txt ←
-o test.txt
```

6.5. Mutató a gyökér

Írd át az előző forrást, hogy a gyökér csomópont ne kompozícióban, csak aggregációban legyen a fával!

Megoldás videó:

Megoldás forrása:https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Welch/z3a8.cpp

Tutoriált: Loós Tamás

Az előző forrásban a gyökéret mint objektumot kezeltük. A fa mutatót egyszerűen ráállítjuk a gyökérre a konstruktőrben, amit alul hozunk létre.

Azonban most a gyökér is mutató lesz, ezért az előző forrásban mindenhol ahol referenciaként adtuk át a gyökeret a fa mutatónak, ott csak simán referencia nélkül tesszük. Továbbá a konstruktőrben a gyökeret példányosítjuk, helyet foglalul a memóriában és erre állítjuk a fát rá. Valamint a szabadításkor pont helyet nyilat használunk ha mutató mutatóit kell elérni.

```
public:
LZWBinFa ()
{
gyoker = new Csomopont();
fa = gyoker;
}
~LZWBinFa ()
{
szabadit (gyoker->egyesGyermekek ());
}
```

```
    szabadit (gyoker->nullasGyermekek ());
    delete gyoker;
}
```

```
protected:
    Csomopont *gyoker;
};
```

6.6. Mozgató szemantika

Írj az előző programhoz mozgató konstruktort és értékkadást, a mozgató konstruktor legyen a mozgató értékkadásra alapozva!

Megoldás videó:

Megoldás forrása:https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Welch/z3a11.cpp

Az előző feladatot továbbfejlesztve, megalkotjuk a mozgató konstruktort. Paraméterként átadjuk a régi fát, az új fa gyökerét null-ra állítjuk, majd a move függvényt meghívjuk, amivel mozgatjuk az elemeket. A mozgató értékkadásban pedig megcseréljük az új és a régi fa gyökerét swap függvény segítségével, ezzel kinullázva a régit.

```
LZWBinFa ( LZWBinFa && regi ) {
    std::cout << "LZWBinFa move ctor" << std::endl;

    gyoker = nullptr;
    *this = std::move (regi);

}

LZWBinFa & operator= ( LZWBinFa && regi)
{
    std::swap (gyoker, regi.gyoker);
    return *this;
}
```

Alább láthatunk három példát, hogy ahol a mozgató konstruktort használjuk. Másoló konstruktorral lemosjuk az eredeti fát kétszer és azokat megcseréljük. Egy vektorba rakjuk rakjuk a fát, illetve egy új fát hozunk létre mozgató konstruktor segítségével.

```
LZWBinFa binFa2 = binFa;
LZWBinFa binFa4= binFa;
```

```
std::cout << "Swap" << std::endl;
std::swap(binFa2, binFa4);

std::vector<LZWBinFa> v;
    std::cout << "Vector" << std::endl;
v.push_back(std::move (binFa ));

std::cout << "Új fa" << std::endl;
LZWBinFa binFa3 = std::move (binFa2);
```

Futtatáskor láthatjuk, mikor melyik konstruktor lép működésbe és hányszor.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Welch$ g++ z3a11.cpp ←
-o fa
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Welch$ ./fa adat.txt ←
-o output
LZWBinFa copy ctor
LZWBinFa copy ctor
Swap
LZWBinFa move ctor
Vector
LZWBinFa move ctor
Új fa
LZWBinFa move ctor
```

7. fejezet

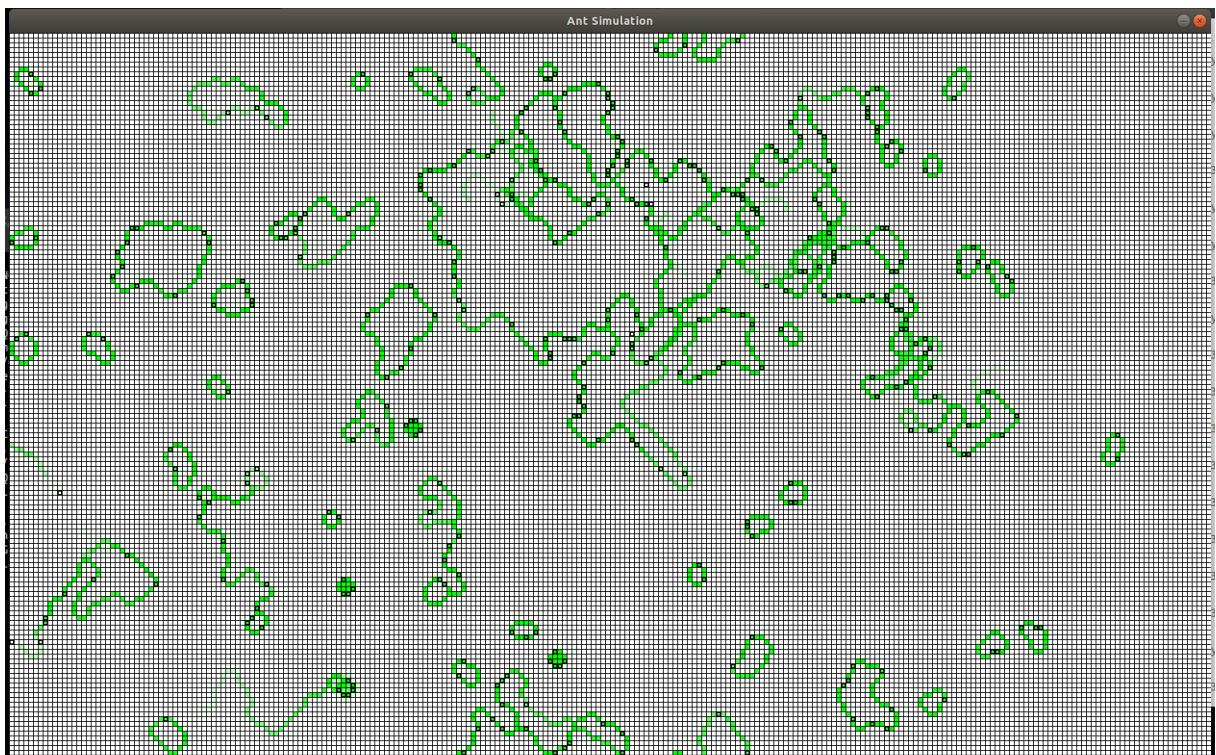
Helló, Conway!

7.1. Hangyszimulációk

Írj Qt C++-ban egy hangyszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

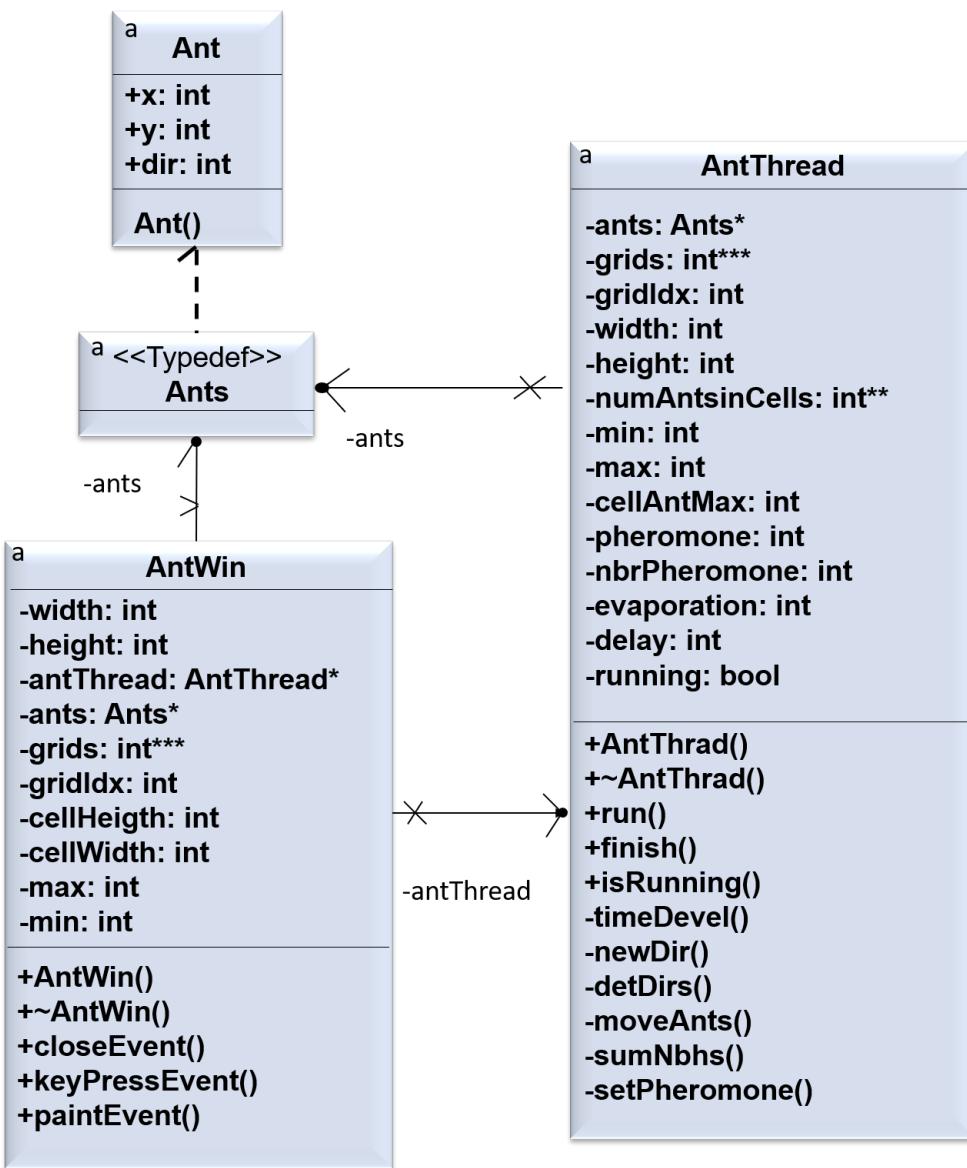
Megoldás forrása: https://github.com/p-adrian05/BHAX/tree/master/attention_raising/Myrmecologist



Fordításkor a letöltött Qt keretrendszerben található qmake-et használjuk. P billentyűvel lehet megállítani, szüneteltetni a hangyák mozgását q-val pedig kilépni a programból.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/Myrmecologist$ /home ←
    /adrian/Qt/5.12.2/gcc_64/bin/qmake myrmecologist.pro
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/Myrmecologist$ make
```

```
adrian@adrian-MS-7817:~/Desktop/BHAX/attention_raising/Myrmecologist$ ./ ←
myrmecologist -w 250 -m 150 -n 400 -t 10 -p 5 -f 80 -d 0 -a 255 -i 3 -s ←
3 -c 22
```



Az **Ant** osztályba a hangya tulajdonságai meghatározva vannak. Az `x` és `y` a koordinátái, a `dir` pedig az irány amibe tart. A `+` jel azt jelenti, hogy public elérésű, tehát az osztályon kívül is látszódik, elérhető. A `-` pedig private, tehát Az `ants` egy vektor, ami a hangyákat tárolja.

Az **AntWin** osztályban a `width` és `height` tulajdonság az ablak szélessége, magassága pixelben, a `cellWidth` és `cellHeighth` pedig a pixelk vagy cellák szélessége, magassága amiben a hangyák vannak ábrázolva, megjelenítve. Az **antThread** az ablaknak egy szála amivel a hangyszámításokat végzi. A `grids` a két rácsot jelenti, a `gridIdx` pedig a két rácspont közül egyet tárol. A `max` és `min` pedig a `min 1` és `max 255` fero-mon lehetséges értékét jelenti. A függvények közül ott van a konstruktur és a destruktur. A `closeEvent()` a kikapcsolási esemény ami meghívja az **AntThread** osztályból a `finish()` metódust ami leállítja az ablakot a running hamis értékre állításával, `keyPressEvent()` a gombnyomásokat dolgozza fel, a `paintEvent()` pedig a hangyák színezését alakítja.

Az **AntThread** osztályban a tulajdonságok hasonlóak mint az **AntWin**ben, kiegészítve párral. Az `evapora-`

tion párolgás mértékét tárolja, feromonok száma, hangyák száma egy cellában. A funkcióknál konstruktor és a destruktur, publikus funkciók mint a run() és finish(), illetve az isRunning függvény, ami visszaad egy igaz vagy hamis. Nem publikus, private függvények pl feromonok beállítására a setPheromone(), hangya mozgatáshoz MoveAnts(), új irány megadása newDir()-el.

7.2. Java életjáték

Írd meg Java-ban a John Horton Conway-féle életjátékot, valósítsa meg a sikló-kilövőt!

Megoldás video:

Megoldás forrása: https://github.com/p-adrian05/BHAX/blob/master/textbook_programs/Conway/Sejtautomata.java

A John Horton Conway féle életjáték szabályai egyszerűek. Van egy élettér, ami cellákból áll és cellánként egy sejt él. Egy élő sejt akkor marad élő, ha kettő vagy három élő szomszédja van, különben halott lesz. A halott sejt pedig halott marad, ha három élő szomszédja van, különben élő lesz. Ezek a szabályok vannak megfogalmazva a következő metódusban.

```
public void időFejlődés() {  
  
    boolean [][] rácsElőtte = rácsok[rácsIndex];  
    boolean [][] rácsUtána = rácsok[(rácsIndex+1)%2];  
  
    for(int i=0; i<rácsElőtte.length; ++i) { // sorok  
        for(int j=0; j<rácsElőtte[0].length; ++j) { // oszlopok  
  
            int élők = szomszédokSzáma(rácsElőtte, i, j, ÉLŐ);  
  
            if(rácsElőtte[i][j] == ÉLŐ) {  
                /* Élő élő marad, ha kettő vagy három élő  
                szomszedja van, különben halott lesz. */  
                if(élők==2 || élők==3)  
                    rácsUtána[i][j] = ÉLŐ;  
                else  
                    rácsUtána[i][j] = HALOTT;  
            } else {  
                /* Halott halott marad, ha három élő  
                szomszedja van, különben élő lesz. */  
                if(élők==3)  
                    rácsUtána[i][j] = ÉLŐ;  
                else  
                    rácsUtána[i][j] = HALOTT;  
            }  
        }  
    }  
    rácsIndex = (rácsIndex+1)%2;  
}
```

A sejttérben élőlényeket helyezünk el, ez a sikló. Adott irányba halad, másolja magát a sejttérben. A rács kettő dimenziós tömbbe helyezzük ez az állatkát. Az x érték a befoglaló téglalal bal felső sarkának oszlopára, az y pedig a bal felső sarkának sora.

```
public void sikló(boolean [][] rács, int x, int y) {  
  
    rács[y+ 0][x+ 2] = ÉLŐ;  
    rács[y+ 1][x+ 1] = ÉLŐ;  
    rács[y+ 2][x+ 1] = ÉLŐ;  
    rács[y+ 2][x+ 2] = ÉLŐ;  
    rács[y+ 2][x+ 3] = ÉLŐ;  
  
}
```

A sikló ágyú adott irányba lövi ki a fenti siklókat, a rács lesz ahova helyezzük a siklót, az x és y pedig a téglalal bal felső sarkának oszlopára, illetve sora.

```
public void siklóKilövő(boolean [][] rács, int x, int y) {  
  
    rács[y+ 6][x+ 0] = ÉLŐ;  
    rács[y+ 6][x+ 1] = ÉLŐ;  
    rács[y+ 7][x+ 0] = ÉLŐ;  
    rács[y+ 7][x+ 1] = ÉLŐ;  
  
    rács[y+ 3][x+ 13] = ÉLŐ;  
  
    rács[y+ 4][x+ 12] = ÉLŐ;  
    rács[y+ 4][x+ 14] = ÉLŐ;  
  
    rács[y+ 5][x+ 11] = ÉLŐ;  
    rács[y+ 5][x+ 15] = ÉLŐ;  
    rács[y+ 5][x+ 16] = ÉLŐ;  
    rács[y+ 5][x+ 25] = ÉLŐ;  
  
    rács[y+ 6][x+ 11] = ÉLŐ;  
    rács[y+ 6][x+ 15] = ÉLŐ;  
    rács[y+ 6][x+ 16] = ÉLŐ;  
    rács[y+ 6][x+ 22] = ÉLŐ;  
    rács[y+ 6][x+ 23] = ÉLŐ;  
    rács[y+ 6][x+ 24] = ÉLŐ;  
    rács[y+ 6][x+ 25] = ÉLŐ;  
  
    rács[y+ 7][x+ 11] = ÉLŐ;  
    rács[y+ 7][x+ 15] = ÉLŐ;  
    rács[y+ 7][x+ 16] = ÉLŐ;  
    rács[y+ 7][x+ 21] = ÉLŐ;  
    rács[y+ 7][x+ 22] = ÉLŐ;  
    rács[y+ 7][x+ 23] = ÉLŐ;  
    rács[y+ 7][x+ 24] = ÉLŐ;
```

```
rács[y+ 8][x+ 12] = ÉLŐ;
rács[y+ 8][x+ 14] = ÉLŐ;
rács[y+ 8][x+ 21] = ÉLŐ;
rács[y+ 8][x+ 24] = ÉLŐ;
rács[y+ 8][x+ 34] = ÉLŐ;
rács[y+ 8][x+ 35] = ÉLŐ;

rács[y+ 9][x+ 13] = ÉLŐ;
rács[y+ 9][x+ 21] = ÉLŐ;
rács[y+ 9][x+ 22] = ÉLŐ;
rács[y+ 9][x+ 23] = ÉLŐ;
rács[y+ 9][x+ 24] = ÉLŐ;
rács[y+ 9][x+ 34] = ÉLŐ;
rács[y+ 9][x+ 35] = ÉLŐ;

rács[y+ 10][x+ 22] = ÉLŐ;
rács[y+ 10][x+ 23] = ÉLŐ;
rács[y+ 10][x+ 24] = ÉLŐ;
rács[y+ 10][x+ 25] = ÉLŐ;

rács[y+ 11][x+ 25] = ÉLŐ;

}
```

Az egész programot, minden funkciójával együtt egy objektumként írjuk meg, majd a main függvényben példányosítjuk, paraméterként átadva az objektum konstruktorának az oszlop és sor méretet.

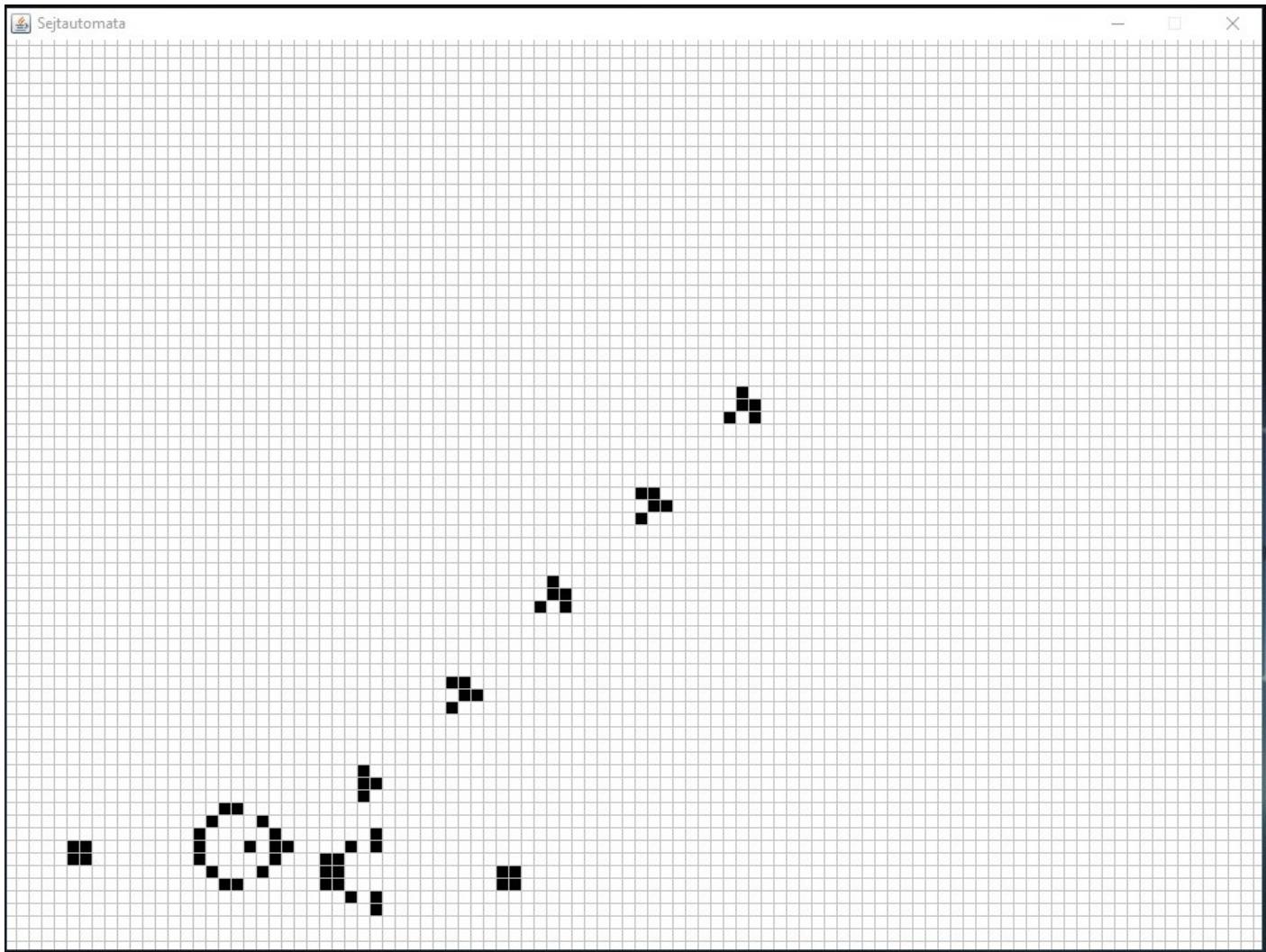
```
public static void main(String[] args) {

    new Sejtautomata(100, 75);
}

}
```

Fordításkor, futtatáskor java fordítóját használjuk illetve a szövegkódolást UTF 8-ra állítva:

```
javac -encoding UTF-8 Sejtautomata.java
java Sejtautomata
```



S billentyű lenyomásával felvétel készül a sejttér állapotáról. N-el lehet a sejtek méretét növelni, k-val csökkeneni, g-vel lehet gyorsítani a szimulációt és i-vel lassítani. Az egérmutató jobb vagy bal gombjával egy sejt állapotát az ellenkezőjére változtatjuk. Az egérmutató vonszolásával az érintett sejteket élő állapotba kapcsoljuk.

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: https://github.com/p-adrian05/BHAX/tree/master/textbook_programs/Conway/sejtautomata_c

A szabály implementálása ugyanúgy mint javaban:

```
void SejtSzal::idoFejlodes() {  
  
    bool **racsElotte = racsok[racsIndex];  
    bool **racsUtana = racsok[(racsIndex+1)%2];  
  
    for(int i=0; i<magassag; ++i) { // sorok
```

```
for(int j=0; jszelesség; ++j) { // oszlopok

    int elok = szomszedokSzama(racsElotte, i, j, SejtAblak::ELO);

    if(racsElotte[i][j] == SejtAblak::ELO) {

        if(elok==2 || elok==3)
            racsUtana[i][j] = SejtAblak::ELO;
        else
            racsUtana[i][j] = SejtAblak::HALOTT;
    } else {

        if(elok==3)
            racsUtana[i][j] = SejtAblak::ELO;
        else
            racsUtana[i][j] = SejtAblak::HALOTT;
    }
}
racsIndex = (racsIndex+1)%2;
}
```

Sikló:

```
void SejtAblak::siklo(bool **racs, int x, int y) {

    racs[y+ 0][x+ 2] = ELO;
    racs[y+ 1][x+ 1] = ELO;
    racs[y+ 2][x+ 1] = ELO;
    racs[y+ 2][x+ 2] = ELO;
    racs[y+ 2][x+ 3] = ELO;

}
```

Siklókilövő

```
void SejtAblak::sikloKilovo(bool **racs, int x, int y) {

    racs[y+ 6][x+ 0] = ELO;
    racs[y+ 6][x+ 1] = ELO;
    racs[y+ 7][x+ 0] = ELO;
    racs[y+ 7][x+ 1] = ELO;

    racs[y+ 3][x+ 13] = ELO;

    racs[y+ 4][x+ 12] = ELO;
    racs[y+ 4][x+ 14] = ELO;

    racs[y+ 5][x+ 11] = ELO;
    racs[y+ 5][x+ 15] = ELO;
    racs[y+ 5][x+ 16] = ELO;
```

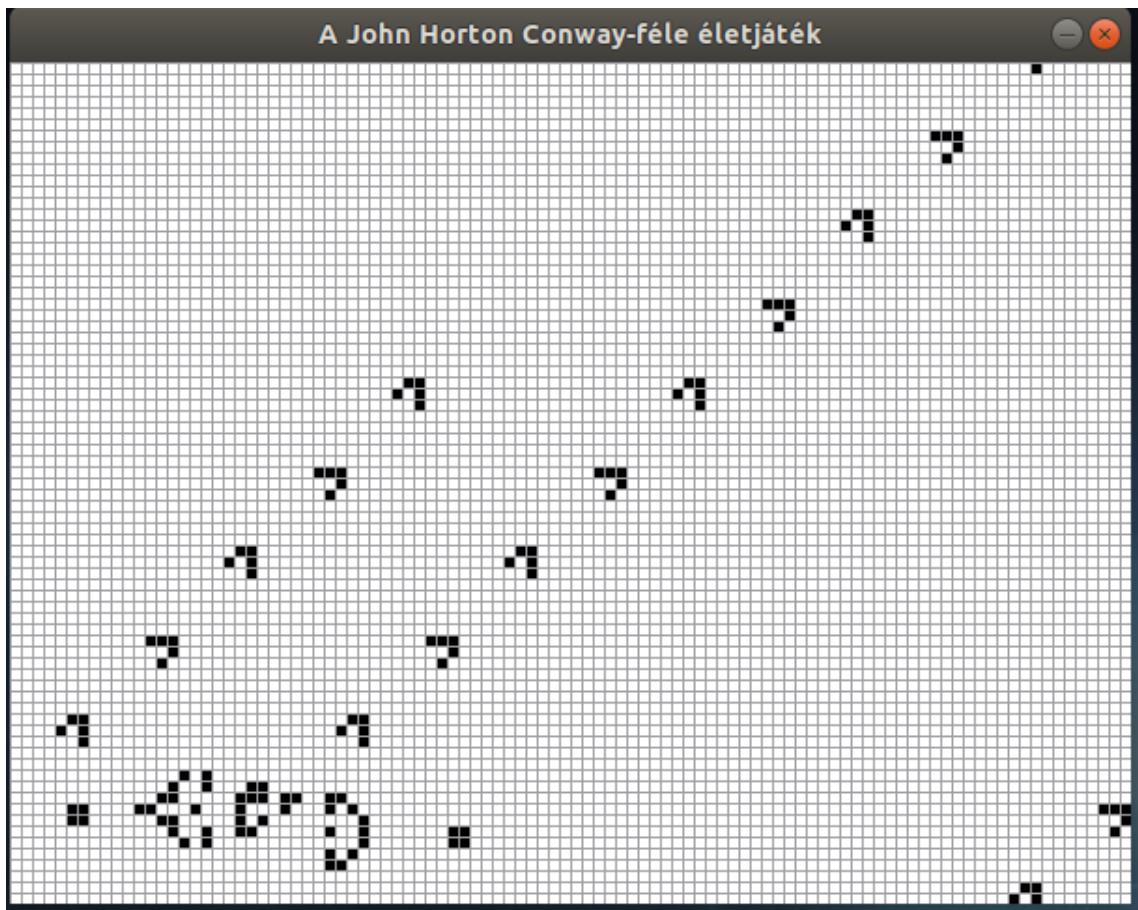
```
racs[y+ 5][x+ 25] = ELO;  
  
racs[y+ 6][x+ 11] = ELO;  
racs[y+ 6][x+ 15] = ELO;  
racs[y+ 6][x+ 16] = ELO;  
racs[y+ 6][x+ 22] = ELO;  
racs[y+ 6][x+ 23] = ELO;  
racs[y+ 6][x+ 24] = ELO;  
racs[y+ 6][x+ 25] = ELO;  
  
racs[y+ 7][x+ 11] = ELO;  
racs[y+ 7][x+ 15] = ELO;  
racs[y+ 7][x+ 16] = ELO;  
racs[y+ 7][x+ 21] = ELO;  
racs[y+ 7][x+ 22] = ELO;  
racs[y+ 7][x+ 23] = ELO;  
racs[y+ 7][x+ 24] = ELO;  
  
racs[y+ 8][x+ 12] = ELO;  
racs[y+ 8][x+ 14] = ELO;  
racs[y+ 8][x+ 21] = ELO;  
racs[y+ 8][x+ 24] = ELO;  
racs[y+ 8][x+ 34] = ELO;  
racs[y+ 8][x+ 35] = ELO;  
  
racs[y+ 9][x+ 13] = ELO;  
racs[y+ 9][x+ 21] = ELO;  
racs[y+ 9][x+ 22] = ELO;  
racs[y+ 9][x+ 23] = ELO;  
racs[y+ 9][x+ 24] = ELO;  
racs[y+ 9][x+ 34] = ELO;  
racs[y+ 9][x+ 35] = ELO;  
  
racs[y+ 10][x+ 22] = ELO;  
racs[y+ 10][x+ 23] = ELO;  
racs[y+ 10][x+ 24] = ELO;  
racs[y+ 10][x+ 25] = ELO;  
  
racs[y+ 11][x+ 25] = ELO;  
  
}
```

```
#include <QApplication>  
#include "sejtablak.h"  
#include <QDesktopWidget>  
  
int main(int argc, char *argv[])  
{  
    QApplication a(argc, argv);  
    SejtAblak w(100, 75);
```

```
w.show();  
  
    return a.exec();  
}
```

C++-nál használjuk a QT keretrendszerét, ezért fordításkor másképp kell eljárni. A **qmake -project** parancssal a .pro fájlt generálja le amibe a programot foglalja be, majd ezt futtatjuk, ami a makefile-t generálja le a fordításhoz és a **make** parancs fordítja le a programot. A futtatás ugyanúgy zajlik.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Conway/ ←  
    sejtautomata_c++$ qmake -project  
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Conway/ ←  
    sejtautomata_c++$ qmake sejtautomata_c++.pro  
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Conway/ ←  
    sejtautomata_c++$ make  
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Conway/ ←  
    sejtautomata_c++$ ./sejtautomata_c++
```



7.4. BrainB Benchmark

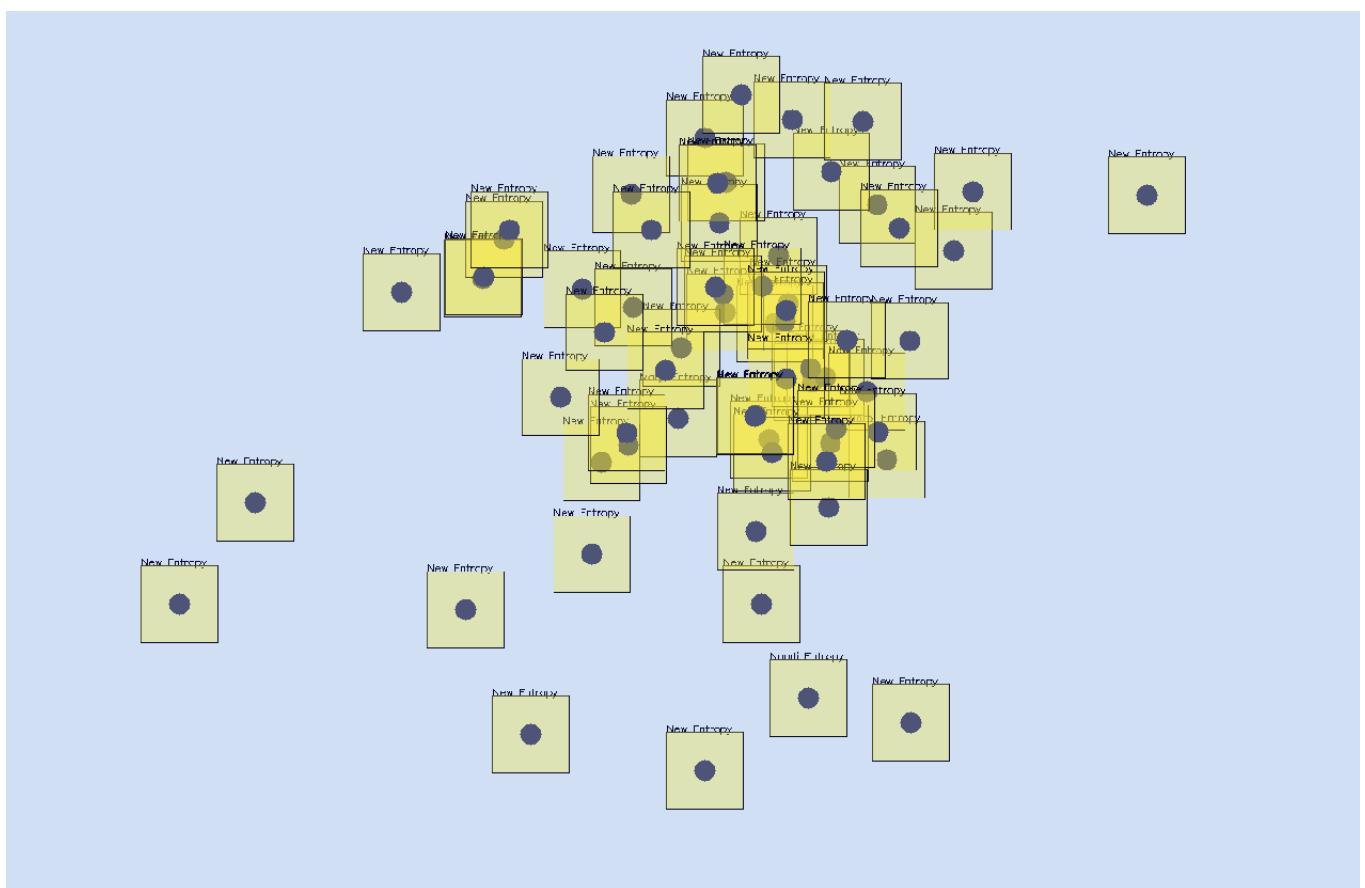
Megoldás video:

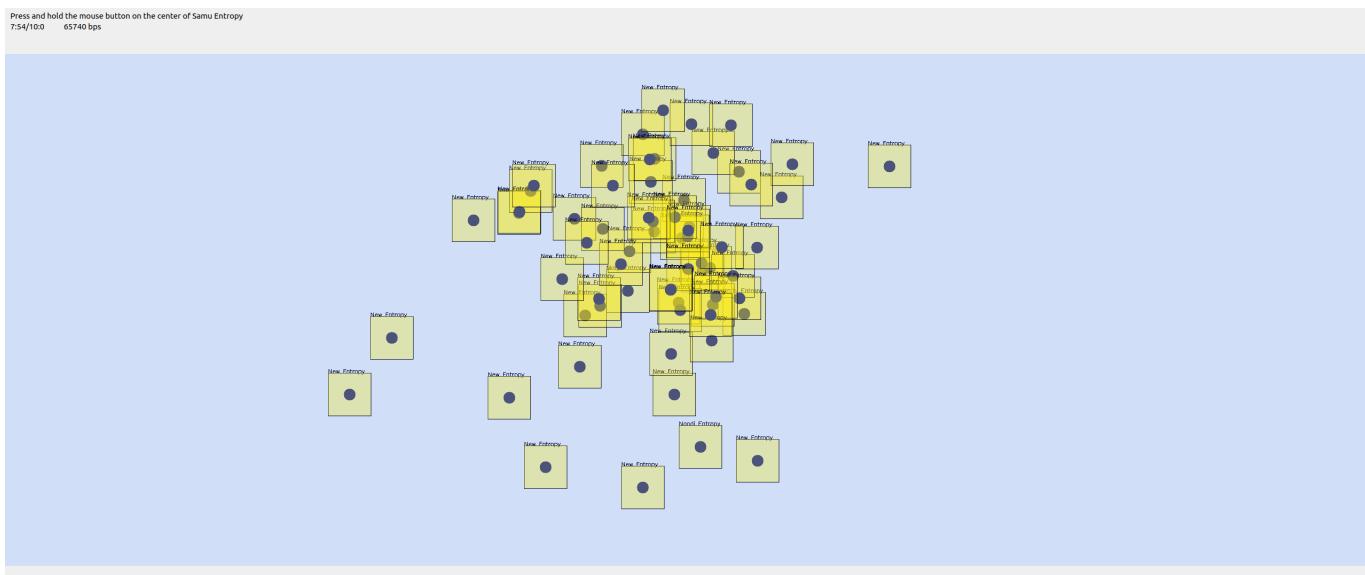
Megoldás forrása: https://github.com/p-adrian05/BHAX/tree/master/textbook_programs/Conway/brainB

A program célja az esportolók tesztelése, mennyire tudják követni a karakter mozgását. A Samu Entropy dobozt kell figyelni, a bal egérgombot lenyomva tartva kell a mozgó négyzetben lévő körben tartani az egérmutatót, miközben újabb dobozok jelennek meg. A folyamat során egyre több doboz jelenik meg a képernyőn, nehezítve a pötty követését. Ha 1 másodpercnél több ideig nincs találat, akkor a játékos elvesztette a dobozat, ebben az esetben lassul a mozgása a dobozoknak.

Fordításkor az OpenCv könyvtárat, illetve a Qt keretrendszer használjuk.

```
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Conway/brainB$ cd ~/home ↵
/adrian/Qt/5.12.2/gcc_64/bin/qmake brainB.pro
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Conway/brainB$ make
adrian@adrian-MS-7817:~/Desktop/BHAX/textbook_programs/Conway/brainB$ ./ ↵
BrainB
```





A mérés 10 percig tart és statisztika is készül a teljesítményről, amit a program könyvtárában egy txt fájlban találunk.

Statisztika:

```
NEMESPOR BrainB Test 6.0.3
time      : 4757
bps       : 69950
noc       : 66
nop       : 0
lost      :
88430 49890 38490 47690 22430 37320 55030 80340 23380 35690 31630
57480 51640 52430 54410 77910 65060 56100 38440 27990 73130 58950
91760 54150 64600 76010 57690 42230 88230 61460 87340 86600 78000
66020 55940 42510 29380 24860 18740 27180 20880 30210 98800 77670
mean     : 54639
var      : 22265
found    : 12260 27850 32540 35900 37480 38720 35430 48380 63090
53240 51950 31620 29260 38030 38100 45430 34040 40400 38150 13600
22650 10550 15040 41430 22450 39400 34580 42030 40890 28850 23090
26380 32400 41120 47030 58650 28890 29860 23800 33290 33890 39580
50580 34480 48240 35910 62090 39560 53180 27260 40480 56760 55280
52680 40060 18040 33110 29950 34880 37220 43520 47130 43310 36780
46640 56910 57250 74210 57090 61080 57030 61700 61160 63160 55330
52750 34830 30670 43520 47470 52590 57700 62180 49140 58600 70210
55620 53690 70350 73760 87230 56430 70120 63860 76840 84790 23620
39110 42570 53710 45910 68270 72020 77040 76780
mean     : 45569
var      : 16385.5
lost2found: 35430 31620 13600 10550 39400 42030 23090 26380 28890
33290 40480 55280 40060 18040 36780 57090 55330 34830 53690 56430
84790 23620
mean     : 38213
var      : 17239.9
found2lost: 47690 22430 37320 55030 80340 23380 35690 31630 57480
54410 77910 65060 38440 73130 91760 64600 76010 88230 61460 87340
86600 98800
mean     : 61579
var      : 23160.7
mean(lost2found) < mean(found2lost)
time      : 7:55
U R about 6.09082 Kilobytes
```

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

aa Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.2. Szoftmax R MNIST

R

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.4. Deep dream

Keras

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.5. Robotpszichológia

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó:

Megoldás forrása:

9.2. Weizenbaum Eliza programja

Éleszd fel Weizenbaum Eliza programját!

Megoldás videó:

Megoldás forrása:

9.3. Gimp Scheme Script-fu: króm effekt

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely megvalósítja a króm effektet egy bemenő szövegre!

Megoldás videó: https://youtu.be/OKdAkI_c7Sc

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Chrome

Tanulságok, tapasztalatok, magyarázat...

9.4. Gimp Scheme Script-fu: név mandala

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely név-mandalát készít a bemenő szövegből!

Megoldás videó: https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelese_a_scheme_programozasi_nyelv

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Mandala

Tanulságok, tapasztalatok, magyarázat...

9.5. Lambda

Hasonlítsd össze a következő programokat!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9.6. Omega

Megoldás videó:

Megoldás forrása:

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

Juhász István - Magas szintű programozási nyelvek 1

A programnyelvek három szintjét különböztetjük meg: gépi nyelv, assembly szintű nyelv, magas szintű nyelv. A magas szintű nyelven megírt programot forrásprogramnak nevezzük. Összeállítását szintaktikai és szemantikai szabályok alapján vegzi a fordító program ami gépi nyelvre fordítja a programot, hogy a processzor végre tudja hajtani a programot. A fordítóprogram a következő lépésekkel hajtja végre: lexikális elemzés, szintaktikai elemzés, szemantikai elemzés, kódgenerálás. A másik technika egy forrásprogram végrehajtására az interpreteres technika, ami nem készít tárgyprogramot, hanem az utasításokat értelmezi és azonnal végre is hajtja. minden programnyelvnek megvan a saját szabványa, amit hivatkozási nyelvnek hívunk.

Programnyelvek osztályozása

Imperatív nyelvek: a programozó egy programszöveget ír, algoritmust kódol ami működteti a processzort.
Alcsoporthajtás: eljárásorientált és objektumorientált nyelvek.

Deklaratív nyelvek: Nem algoritmikus nyelvek, a proramozó csak a problémát oldja meg, a nyelvi implementációkban van beépítve a megoldás megkeresésének módja, a programozónak nincs lehetősége memóriaműveletekre. Alcsoporthajtás: Funkcionális és logikai nyelvek.

Karakterkészlet

Minden program forrászövegének legkisebb alkotórészei a karakterek, amit minden nyelv definiál 3 kategóriába: betűk, számjegyek egyéb karakterek. A lexikális egységek a program azon elemei, melyeket a fordító a lexikális elemzés során felsímer és tokenizál. Fajtái: többkarateres szimbólum, szimbolikus név, címke, megjegyzés, literál. A többkarateres szimbolumok olyan karaktersorozatok, amelyeknek csak a nyelv tulajdonít jelentést. Pl.: ++, --, /*, */. Szimbolikus nevek közül az azonosító olyan karaktersorozat, ami betűvel kezdődik és betűvel vagy számjeggyel folytatódhat. A kulesszavak vagy védett szavak olyan szavak amelyeknek a nyelv jelentést tulajdonít. Pl.: if, for, case, break. A standart azonosítónak a nyelv tulajdonít jelentést, de a programozó által megváltoztatjátó, például a NULL. A megjegyzés olyan programozási eszköz, melynek segítségével a programban olyan karaktersorozatok írása megengedett, amelyek nem a fordítónak szól, hanem a programot olvasónak. Általában a ezek a program működésével kapcsolatos magyarázó szövegek. A literál pedig olyan eszköz aminek segítségével fix értékek építhetők be a program szövegébe.

Adattípusok

Az adattípusnak van egy neve, ami egy azonosító. minden adattípus mögött van egy belső ábrázolási mód. A reprezentáció az egyes típusok tartományába tartozó értékek tában való megjelenését határozza meg, tehát, hogy az egyes elemek hány bájtra képződnek le. Saját típust úgy tudunk létrehozni, hogy megadjuk a tartományát, a műveleteit és a reprezentációját. Két nagy csoportjuk van: az egyszerű adattípus, tartománya atomi értékeket tartalmaz és összetett adattípus, aminek tartományának elemei is valamilyen típussal rendelkeznek. Az egyszerű adattípusba tartozik az egész típus, belső ábrázolásuk fix pontos. A valós típusok belső ábrázoláluk lebegőpontos. A karakteres típus elemei karakterek, karakterlánc típusé pedig karakterszorozatok. A logikai típus, igaz vagy hamis értéket tárol. Az összetett típusok közül a két legfontosabb a tömb és a rekord. A tömb statikus és homogén összetett típus, amelyben az elemek azonos típusúak. A tömböt mint típust meghatározza a dimenzióinak száma, hány sor, hány oszloból áll, az elemek indexei, elemek sorszáma, amely egész típusú és az elemek típusa. A mutató típus egyszerű típus, amely tárccímeket tárolhat. Egyik legfontosabb művelete a megcímzett tárterületen elhelyezkedő érték elérése.

A nevesített konstansnak három része van: név, típus, érték és minden deklarálni kell, ennek értéke ekkor elől és nem változtatható meg. A változónak négy része van? név, attribútumok, cím, érték. A név egy azonosító. Az attribútum a típusa és deklarációval kap értéket, amely változtatható a program futása során. Mindaddig amíg nincs értéke, addig határozatlan, tehát nem használható fel. Explicit vagy automatikus deklaráció lehetséges. Előbbi esetén a programozó végzi a deklarációt, utóbbi esetben pedig a fordítóprogram rendel attribútumot azokhoz a változókhöz amelyek nincsenek explicit módon megadva, deklárvá. A változóhoz cím rendelhető két féle módon: dinamikus tárkiosztás: a futás előtt elől a változó címe és futás alatt ez nem változik. Dinamikus tárkiosztás esetén a rendszer végzi a cím hozzárendelést.

A C nyelvnek vannak aritmetikai és származtatott típusai. Az aritmetikai típusokhoz tartoznak az integrális típusok: egész(int,short,long), karakter(char), felsorolásos és valós(float, double, long double). A származtatott típusokhoz tartozik a tömb, függény, mutató, struktúra, union és vannak a void típusok. Az aritmetikai típusok az egyszerű, a származtatottak pedig az összetett típusok. Nincs logikai típus. A hamis az int 0, minden más inthez rendelt érték igaznak minősül. Az unsigned típusminősítő nem előjeles ábrázolást, a signed pedig előjeles ábrázolást jelöl. A struktúra egy fix szerkezetű rekord. A void típus tartománya üres, nincsenek műveletei. A const megagásával nevesített konstanst deklárlunk. Saját típus definiáláshoz TYPEDEF-el lehetséges, de ez csak a típus nevét adja meg nem hoz létre új típust. Struktúra deklarációja STRUCT-al lehetésges, union pedig UNION-nal. A C csak egydimenziós tömböket kezel. Deklaráláshoz az indexek darabszámát kell megadni, ami 0-tól darabszám-1-ig fut. A C a tömböt mutató típusként kezeli. A tömb neve a tömb első elemét címzi. Van automatikus deklaráció, int egész típus lesz ha egy névhez nem adunk visszatérési típust.

Kifejezések

A Kifejezések szintaktikai eszközök. A kövekező összetevőkből áll: operandusok, operátorok, kerek zárójelek. Az operandus literál, nevesített konstans, változó vagy függvényhívás lehet. Az értéket képviseli. Az operátorok műveleti jelek. A kerek zárójelek pedig a műveletek végrehajtási sorrendjét befolyásolják. Redundánsan alkalmazható. Alakjuk lehet prefix, az operátor az operandusok előtt állnak(* 3 5), infix(3 * 5) és postfix(3 5 *). A kifejezés kiértékelése történhet balról-jobbra, jobbról-balra vagy balról-jobbra a predencia táblázat figyelembevételével, az operátorok erősségeinek figyelembevételével. Zárójelek használatával a predenciátáblázat szerinti végrehajtási sorrendet felül lehet írni. Logikai operátorok esetén, pl és művelet esetén, úgy is előlhet a kifejezés értéke, hogy nem végezzük el az összes műveletet, ez a rövidzár kiértékelés. A kifejezés típusának meghatározásánál kétféle elvet követnek a nyelvek: típusügyenértékűséget és típuskényszerítést. Az előbbi esetén egy kifejezésben egy kétoperandusú operátornak csak azonos típusú operandusai lehetnek. Az utóbbi esetben azonban két különböző típusú operandusai lehetnek egy kétoper-

randusú operátornak. A műveletek viszont csak az azonos belső ábrázolású operandusok között végezhetők el különböző típusú operandusok esetén konverzió van. A c kifejezésorientált nyelv és a típuskényszerítést elvét vallja. A mutató típusú tartományának elemeivel összeadás és kivonás végezhető.

Utasítások

Az utasítások segítségével generálja a fordítóprogram a tárgyprogramot. Két nagy csoportja van: deklarációs és végrehajtható utasítások. A deklarációs utasítás a fordítóprogramnak szólnak, valamelyen szolgáltatást kérnek, üzemmódot állítanak be, információt szolgáltatnak a tárgykód generáláshoz. A végrehajtható utasításokból készül a tárgykód, ezeket az alábbiak szerint csoportosítjuk: Értékadó utasítás, üres utasítás, ugró, elágaztató, ciklusszervező, hívó, vezérlésátadó, I/O utasítások és egyéb utasítások. Az értékadó utasítás feladata beállítani vagy módosítani egy változó értékét. Az üres utasítás hatására a CPU egy üres gépi utasítást hajt végre. Pl continue, null. Az ugró utasítás átadjuk a vezérlést egy adott pontról egy adott címkelvel elátott utasításra. Ez a GOTO. Elágaztató utasítások közé tartotik a feltéletes utasítás ami kétriányú, ami arra szolgál, hogy a program két tevékenység közül válasszon. If, else. A többirányú elágaztató utasítés C-ben a switch(kifejezés) case (feltétel) : (tevékenység). Ez arra szolgál, hogy a program kölcsönösen kizáro akárhány tevékenység közül egyet végrehajtsunk. A ciklusszervező utasítások lehetővé teszik, hogy a program egy bizonyos tevékenységet akárhányszor megismételjen. Két szélsőséges eset amikor egyszer sem fut le, ami az üres ciklus és a másik, hogy soha nem áll le, ez a végtelen ciklus. A feltétes ciklus egy feltétel teljesülése szerint ismétlődik. A ciklusok fajtái: Kezdőfeltételes(while). Végfeltételes(do utasítás while(feltétel)), előírt lépésszámú(for). A végtelen ciklusból szabályosan a break utasítással tudnunk kilépni. A continue vezérlő utasítás a ciklus magjában alkalmazható. A ciklus hátralevő utasításai nem hajta végre, hanem az ismétlődés feltételeit vizsgálja meg és vagy újabb cikluslépésbe kezd, vagy befejezi a ciklust. A return[kifejezés] pedig szabályosan befejezteti a függvényt és visszaadja a vezérlést a hívónak.

A programok szerkezete

A program szövege programegységekre tagolható. Az alábbi porgramegységek léteznek: alprogram, blokk, csomag, taszk. Az alprogram az eljárásorientált nyelvekben a procedurális absztrakció első megjelenései formája. Az alprogram az újrafelhasználás eszköze. Csak egyszer kell megírni és a programrész azon pontjain, ahol szerepelne, csak hivatkozni kell rá. Ezt formális paraméterekkel látjuk el, általánosabban írjuk meg az újrafelhasználás érdekében. Az alprogram a felépítése: név, formális paraméter lista, törzs, környezet. A név egy azonosító, a formális paraméter is a fej része, amiven a azonosítók szerepelnek és ezek a törzsben saját programozási eszközök nevei lehetnek és egy általános szerepkört írnak le, amit a hívás helyén kell konkretizálni az aktuális paraméterek megadásával. A törzsben deklarációs és végrehajtható utasítások szereplnek. Az alprogramban lokális eszközök vannak, ami kívülről nem látható, érhető el, viszont a törzsben hivatkozhatunk globális nevekre. A környezet globális változók együttese. Az alprogramnak két fajtája van: eljárás és függvény. Az eljárás egy tevékenységet hajt végre, és ennek eredményét használjuk fel. A függvény pedig egyetlen értéket határoz meg, rendelkeznie kell visszatérési értékkel. A függvény mellékhatásának nevezük, azt ha megváltoztatja környezetét, paramétereit. Az eljárás szabályosan befejeződök, ha elértük a végét vagy külön utasítással, pl goto utasítással ki lehet lépni a megadott cimkén folytatva a programot. A visszatérési értéket külön utasítás adja vissza, amely egyben be is fejezeti a függvényt. A függvény, csak akkor fejeződök be szabályosan ha ad visszatérési értéket. A goto utasítás függvény esetében szabálytanak befejeződést jelent. Egy programegységek meghívhat egy másikat és az egy újabbat és így tovább, ez nevezük hívási láncnak. A hívási lánc első tagja minden a főprogram. A legutoljára meghívott programegység fejezi be legelőször a működés és a vezérlés visszatér az őt megelőző programegységbe. Azt amikor egy aktív alprogramot hívunk meg rekurzióban nevezünk. Ez lehet közvetlen, amikor egy alprogram önmagát hívja meg és közvetett amikor a hívási láncban már a korábban szereplő alprogramot hívjuk meg. Egyes nyelvek esetén egy alprogramnak meg lehet adni mádsodlagos belépési pontot, tehát nem csak a fejen keresztül lehet meghívni. Paraméterkiértékelésnek nevezük azt a folyama-

tot, amikor amikor egy alprogram hívásánál egymáshoz rendelődnek a formális és aktuális paraméterek, és meghatározódnak azok az információk, amelyek a paraméterátadásnál a kommunikációt szolgáltatják. A blokk olyan programegység, amely csak a másik programegység belsejében helyezkedhet el, külső szinten nem állhat. A blokknak van kezdete, törzse és vége. A kezdetet és véget egy speciális karakterSOROZAT vagy alapszó jelzi, a törzsben lehetnek deklarációs és végrehajtható utasítások. A blokknak nincs paramétere és bárhol elhelyezhető. Aktivizálni úgy lehet a blokkot, hogy rákerül a vezérlés vagy GOTO utasítással ráugrunk a kezdetére.

Paraméterkiértékelés

Paraméterkiértékelés az a folyamat, amikor egy alprogram hívásánál egymáshoz rendelődnek a formális és aktuális paraméterek. A formális paraméterlista az elsődleges, az alprogram specifikációját tartalmazza, csak egy darad van belőle. Az aktuális paraméterlistából annyi lehet ahányszor meghívjuk az alprogramot. Tehát minden az aktuális paramétereket rendeljük a formálisakhoz. Ennek két típusa van: sorrendi és név szerinti kötés. A sorrendi kötés esetén a formális paraméterekhez a felsorolás sorrendjében rendelődnek hozzá az aktuális paraméterek. A név szerinti kötés esetén pedig a paraméterlistában határozzák meg az egymáshoz rendelést. Ezek kombinációja is alkalmazható. Abban az esetben, amikor a formális paraméterek száma fix, ekkor az aktuális paraméterek számának meg kell egyeznie a formális paraméterek számával vagy kevesebb lehet. A típusokról néhány nyelv az típusügyenértékűséget vallja, ekkor az aktuális paraméter típusának azonosnak kell lennie a formális paraméter típusával vagy létezik a típuskényszerítés elve, ami alapján a paramétertípusok konvertálhatóak.

Paraméterátadás

A paraméterátadásnál van egy hívó, ami tetszőleges programegység és egy hívott, amely egy alprogram. Paraméterátadási módok: érték, cím, eredmény, érték-eredmény, név és szöveg szerinti. Érték szerinti átadás esetén, a formális paraméterek van címkomponensük a hívott alprogram területén és az aktuális paraméterek rendelkeznie kell értékkomponenssel a hívó oldalán. A hívott program semmit sem tud a hívóról, a saját területén dolgozik. Az információáramlás egyirányú. Cím szerinti paraméterátadáskor a formális paramétereknek nincs címkomponensük a hívott alprogram területén, aktuális paraméterek viszont rendelkeznie kell. Kiértékeléskor meghatározódik az aktuális paraméter címe és átadódik a hívott programnak. Az információáramlás kétrányú. Az információátadás kétirányú, az alprogram a hívó területéről átvehet értéket, és írhat is oda. Eredmény szerinti átadáskor a kommunikáció egyirányú, a hívottól a hívó felé irányulé és van értékmásolás. Érték-eredmény esetén van címkomponens a hívott területén és az aktuális paraméterek rendelkeznie kell érték és címkomponenssel. Kétirányú a kommunikációs, kétszer van értékmásolás. Név szerinti paraméterátadásnál az aktuális paraméter egy szimbólumsorozat lehet. Az információáramlás iránya az aktuális paraméter adott szövegkörnyezetbeli értelmezésétől függ. A szöveg szerinti paraméterátadás a név szerintinek egy változat. Alprogramok esetén típust paraméterként átadni nem lehet. Az alprogramok formális paramétereinek három csoportja van: input paraméterek, output és input-output paraméterek.

A blokk és hatáskör

A blokk olyan programegység, amely csak másik programegység belsejében helyezkedhet el, külső szinten nem állhat. A blokknak van kezdete, törzse és vége. A kezdetet és a véget egy-egy speciális karakterSOROZAT vagy alapszó jelzi és nincs paramétere. Bárhol elhelyezhető ahol végrehajtható utasítás állhat. A blokkot aktivizálni úgy lehet, hogy rákerül a sor, vagy GOTO utasítással. Egy név hatásköre alatt értjük a program szövegének azon részét, ahol az adott név ugyanazt a programozási eszközt hivatkozza. A név hatásköre a programegységekhez kapcsolódik. Egy programegységen belül nevet a programegység lokális nevének nevezünk. Azt a nevet amely ezen kívül van deklarálva, de ott hivatkozunk rá, szabad névnek hívjuk. Kétféle hatáskörkezelés van: statikus és dinamikus. A statikus fordítási időben történik.

Statikus hatáskörkezelés esetén egy lokális név hatásköre az a programegység, amelyben deklaráltuk és minden olyan programegység, amelyet ez az adott programegység tartalmaz. A hatáskör csak befelé terjed. Az a név amely nem lokális név, de az adott programegységen látható, globális névnek hívjuk. Dinamikus hatáskörkezelésnél egy név hatásköre az a programegység, amelyben deklaráltuk és minden olyan programegység, amely ezen programegységből induló hívási láncban helyezkedik el. Statikus hatáskörkezelés esetén a programban szereplő összes név hatásköre a forrásszöveg alapján egyértelműen megállapítható. Dinamikus hatáskörkezelésnél viszont a hatáskör futási időben változhat.

Input/Output

Az I/O platform, operációs rendszer és implementációfüggő. A perifériákkal való kommunikációért felelős, amely az operatív táróból odaküld vagy onnan vár adatot. Középponttában az állomány áll. Van fizikai állomány és logikai állomány. Funkciói szerint input állomány, amelyből csak olvasni lehet. Output állomány amelybe csak írni lehet. Input-output állomány amelybe írni és olvasni is lehet, ezért tartalma változik. Létezik formátumos modú, szerkesztett és listázott módú adatátvitel. Bináris adatátvitelkor az adatok a tárban és a periférián ugyanúgy jelennek meg. Programban állománnyal való munkához a következőket kell végrejteni: Deklaráció, összerendelés, állomány megnyitás, feldolgozás és lezárás. Összerendeléskor a logika állományt egy fizikai állománnyal feleltetjük meg. Lezáráskor pedig ez kapcsolat megszűnik. Implicit állománynak hívjuk azt, amikor a programozó az írás olvasást úgy kezeli, hogy az közvetlenül valamelyik perifériával történik. Ebben az esetben az állományt nem kell deklarálni, összerendelni, megnyitni és lezárni, mindezt futtató rendszer automatikusan kezeli. C nyelvben az I/O eszközrendszer nem része a nyelvnek, hanem könyvtári függvények általnak rendelkezésre. Ezek a függvények minimum egy karakter illetve egy bájt írását, olvasását teszik lehetővé.

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

Megoldás videó: <https://youtu.be/zmfT9miB-jY>

Vezérlési szerkezetek

A kifejezések utasítássá válnak, ha pontosvessző követi őket. C-ben a pontosvessző utasítás lezáró jel és nem elválasztó szimbólum. A kapcsos zárójelek közé pedig deklarációkat és utasításokat egyetlen blokkba foghatunk össze. Ez egyetlen utasítással lesz egenyértékű. Az if-else utasítással döntést, választást írunk le. A switch utasítás a többirányú programelágaztatás egyik eszköze. A switch kiértékeli a zárójelek közötti ki-fejezést és összehasonlítja az összes esettel(case). Ha valamelyik case azonos a kifejezés értékével, akkor a végrehajtás ennél kezdődik. Ha nincs egyező eset akkor a default címkelével elátott case kerül végrehajtásra. Mivel case-ek címkeként működnek, miután végrehajtódott, a következő case-re ugrik ezért gondoskodni kell a kilépésről break utasítással. A while utasítás fejével található utasítás kiértékelésre kerül és ha az értéke nem 0, akkor végrehajtja az utasítást, majd újra kiértékeli a kifejezést, addig amíg 0 nem lesz és véget és a végrehajtás. A for utasítás ugyanígy működik, csak 1 helyett 3 kifejezést kell megadni. Az első és 3. kifejetés értékadás vagy függvényhívás. A 2. pedig relációs kifejezés. A do-while utasítás ciklusban a kiugrás feltétel teljesülését nem a ciklus elején, hanem a végén vizsgálja a ciklustörzs végrehajtása után. A törzs tehát legalább egyszer végrehajtódnak. A break utasítással a vizsgálat előtt is ki lehet ugrani a ciklusból hasonlóan a switch-ből. A continue utasítás a break-hez kapcsolódik és a ciklus következő iterációjának megkezdését idézi elő. A while és a do esetében ez azt jelenti, hogy azonnal végrehajtódnak a feltételezésről, a for esetében pedig a vezérlés azonnal az újrainicializálási lépéstre kerül. A goto utasítással a vezérlés feltétel nélkül átadható egy címkét megadva. A címkézett utasítás azonosító: alakú előtagok,

amelyek az azonosítót címkeként deklarálják. A címke a goto célpontjaként szolgál. A nulla utasítás alakja pontosvessző, ami hordozhat címkét közvetlenül valamely összetett utasítás előtt, vagy a while-hoz hasonló valamelyik ciklusutasítás számára üres ciklustörzset képezhet.

10.3. Programozás

[BMECPP]

A C++ nem objektumorientált tulajdonságai

A C++ nyelv a C továbbfejlesztett változata. A C nyelv veszélyesebb elemeit cseréli le biztonságosabb megoldásokra és átláthatóbb, kényelmesebb szolgáltatásokat tesz lehetővé. C++-ban egy függvény üres paraméterlistával void függvénynek minősél, C-ben pedig azt hogy tetszőleges számú paraméterrel hívható. A main függvényben nem kötelező a return használata, mivel a fordító automatikusan return 0 jelzőt fordít a kódba. Bevezették a bool típust ami true vagy false értéket vehet fel. Beépített típus lett a wchar_t a több bájtos sztringliterálok definiálására. minden olyan helyen állhat változódeklaráció, ahol utasítás is állhat. A függvénynevek túlterhelése is megjelent, így lehetőség van azonos függvények létrehozására, amennyiben az argumentumlistájuk különöző. Lehetőség lett arra, hogy a függvények argumentumainak alapértelmezett értéket adjunk meg, amennyiben a függvényhíváskor nem adunk meg értéket az argumentumoknak. A C-ben pointerek segítségével kell megoldanunk a függvényparaméterek cí szerinti átadását. Ezt a problémát oldja meg a C++ referenciatípus bevezetése, ami feleslegessé teszi a pointereknek a cím szerinti paraméterátadásban betöltött szerepét. Ekkor csak egy és jelet kell írnunk a paraméter deklarációjában a név elő. Továbbá kihasználva ekkor cím szerint adjuk át az argumentumot, nagyméretű argumentumok esetén teljesítménynövekedés érhető el, ha csak az argumentumok címét adjuk át, és nem másoljuk le őket.

Operátorok és túlterhelésük

A C++ operátoros kifejezései az összeadás, szorzás, kivonás, osztás és egyéb operátorok végzik az adott műveletet. Az operátorok kiértékelési sorrendjét szigorú szabályrendszer rögzíti, ezt zárójelekkel befolyásolhatjuk. A szabályrendszer a precedenciáblázat tartalmazza. Az operátorok olyan speciális nevű függvények, amelyek kiértékelése egy speciális szabályrendszer alapján működik. Az operator kulcsszóval adjuk meg, hogy egy speciális függvényről van szó. Mivel a függvénynevek különböző argumentumok esetén túlterhelhetőek, ezért az operátorok neveit is túlterhelhetjük. Ezzel nem az a cél, hogy már bevált operátorok működését megváltoztassuk, hanem hogy az általunk definált típusokra is megadhassuk.

III. rész

Második felvonás

**Bátf41 Haxor Stream**

A feladatokkal kapcsolatos élő adásokat sugároz a <https://www.twitch.tv/nbatfai> csatorna, melynek permanens archívuma a <https://www.youtube.com/c/nbatfai> csatornán található.

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

Az előző feladat kódját fejleszd tovább: vizsgáld, hogy Vannak-e Java osztályok a Pi hexadecimális kifejtésében!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

[MARX] Marx, György, *Gyorsuló idő*, Typotex , 2005.

11.4. C

[KERNIGHANRITCHIE] Kernighan, Brian W. & Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

11.5. C++

[BMECPP] Benedek, Zoltán & Levendovszky, Tihamér, *Szoftverfejlesztés C++ nyelven*, Bp., Szak Kiadó, 2013.

11.6. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf , 2004.

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPORG tanulószoba, <https://www.facebook.com/groups/udprog>, a DEAC-Hackers előszoba, <https://www.facebook.com/groups/DEAHCackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésekért és hasznos észrevételeikért.

Ezen túl kiemelt köszönet illeti az említett UDPORG közösséget, mely a Debreceni Egyetem reguláris programozás oktatása tartalmi szervezését támogatja. Sok példa eleve ebben a közösségen született, vagy itt került említésre és adott esetekben szerepet kapott, mint oktatási példa.