

Univerzális programozás

Így neveld a programozód!

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>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert, Bátfai, Mátyás, Bátfai, Nándor, Bátfai, Margaréta, Ács Semendi, Ádám-István	2020. október 15.	

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	A Brun tételes feladat kidolgozása.	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	4
2. Helló, Turing!	6
2.1. Végtelen ciklus	6
2.2. Lefagyott, nem fagyott, akkor most mi van?	8
2.3. Változók értékének felcserélése	10
2.4. Labdapattogás	11
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	12
2.6. Helló, Google!	13
2.7. A Monty Hall probléma	15
2.8. 100 éves a Brun tétel	16
2.9. Minecraft MALMÖ -	20
3. Helló, Chomsky!	21
3.1. Decimálisból unárisba átváltó Turing gép	21
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	22
3.3. Hivatkozási nyelv	23
3.4. Saját lexikális elemző	24
3.5. Leetspeak	25

3.6. A források olvasása	27
3.7. Logikus	28
3.8. Deklaráció	29
3.9. Minecraft MALMÖ - Csiga diszkrét	32
4. Helló, Caesar!	33
4.1. double ** háromszögmátrix	33
4.2. C EXOR titkosító	35
4.3. Java EXOR titkosító	37
4.4. C EXOR törő	38
4.5. Neurális OR, AND és EXOR kapu	41
4.6. Hiba-visszaterjesztéssel perceptron	41
4.7. Minecraft MALMÖ - Mit lát Steve?	42
5. Helló, Mandelbrot!	43
5.1. A Mandelbrot halmaz	43
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	44
5.3. Biomorfok	47
5.4. A Mandelbrot halmaz CUDA megvalósítása	47
5.5. Mandelbrot nagyító és utazó C++ nyelven	47
5.6. Mandelbrot nagyító és utazó Java nyelven	48
5.7. Minecraft MALMÖ -Lávaig	48
6. Helló, Welch!	50
6.1. Első osztályom	50
6.2. LZW	52
6.3. Fabejárás	52
6.4. Tag a gyökér	52
6.5. Mutató a gyökér	52
6.6. Mozgató szemantika	53
6.7. Minecraft MALMÖ -5x5x5	58
7. Helló, Conway!	59
7.1. Hangyaszimulációk	59
7.2. Java életjáték	71
7.3. Qt C++ életjáték	79
7.4. BrainB Benchmark	79
7.5. Minecraft MALMÖ -Virág összeszedés	82

8. Helló, Schwarzenegger!	84
8.1. Szoftmax Py MNIST	84
8.2. Mély MNIST	86
8.3. Minecraft-MALMÖ és 8.4 Vörös Pipacs pokol/javíts a 19 RF-en	86
9. Helló, Chaitin!	87
9.1. Iteratív és rekurzív faktoriális Lisp-ben	87
9.2. Gimp Scheme Script-fu: króm effekt	88
9.3. Gimp Scheme Script-fu: név mandala	91
10. Helló, Gutenberg!	93
10.1. Programozási alapfogalmak	93
10.2. Programozás bevezetés	93
10.3. Programozás	93
10.4. Programozás, Python	94
III. Második felvonás	95
11. Helló, Arroway!	97
11.1. Olvasónapló	97
11.2. OO szemlélet	97
11.3. Gagyi	99
11.4. Yoda	99
11.5. Kódolás from scratch	100
12. Helló, Liskov!	103
12.1. Liskov helyettesítés sértése	103
12.2. Szülő-gyerek	106
12.3. Anti OO	107
12.4. Ciklomatikus komplexitás	113
13. Helló, Mandelbrot!	115
13.1. Reverse engineering UML osztálydiagram	115
13.2. Forward engineering UML osztálydiagram	116
13.3. BPMN	120
13.4. EPAM: OOP OO modellezés	120
13.5. .	121

IV. Irodalomjegyzék	122
13.6. Általános	123
13.7. C	123
13.8. C++	123
13.9. Lisp	123

DRAFT

Ábrák jegyzéke

2.1. 100 százalékos magok	8
2.2. A B_2 konstans közelítése	20
3.1. A Turing gép működése	21
4.1. A double ** háromszögmátrix a memóriában	35
5.1. A Mandelbrot halmaz a komplex síkon	43
7.1. BraiB működés közben	79
9.1. Krómozás	91
12.1. Ciklomatikus Komplexitása a z3a18qa5_from_scratch.cpp programnak	114
13.1.	115
13.2.	116
13.3. BPMN tevékenység	120

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 allokálni igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Mindenesetre 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 gyerekeknek, 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 gyerekeknek, 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 (hogyan lássuk mást is) példával.

Hogyan nyomjuk?

Ránts 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.xsl
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 találsz az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

Ne cifrázzuk: programok írása. Mik akkor a programok? Mit jelent az írásuk?

1.2. Milyen doksikat olvassak el?

- Kezd ezzel: <http://esr.fsf.hu/hacker-howto.html>!
- Olvasgasd aztán 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
- C kapcsán a [**KERNIGHANRITCHIE**] könyv adott részei.
- C++ kapcsán a [**BMECPP**] könyv adott részei.
- Az igazi kockák persze csemegéznek a C nyelvi szabvány [ISO/IEC 9899:2017](#) kódcsipeteiből is.
- Amiből viszont a legeslegjobban lehet tanulni, az a [The GNU C Reference Manual](#), mert gcc specifikus és programozókra van hangolva: szinte csak 1-2 lényegi mondat és apró, lényegi kódcsipetek! Aki pdf-ben jobban szereti olvasni: <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf>
- Az R kódok olvasása kis általános tapasztalat után automatikusan, erőfeszítés nélkül menni fog. A Python nincs ennyire a spektrum magától értetődő végén, ezért ahhoz olvasd el a [**BMECPP**] könyv - 20 oldalas gyorstalpaló részét.

1.3. Milyen filmeket nézzek meg?

- 21 - Las Vegas ostroma, <https://www.imdb.com/title/tt0478087/>, benne a **Monty Hall probléma** bemutatása.
 - Kódjátzsma, <https://www.imdb.com/title/tt2084970>, benne a **kódtörő feladat** élménye.
-

- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.

DRAFT

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.

DRAFT

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: <https://youtu.be/lvmi6tyz-nl>

Videó: <https://www.youtube.com/watch?v=bI-yjJ58zLc>

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/infty-f.c, bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/infty-w.c.

Számos módon hozhatunk és hozunk létre végtelen ciklusokat. Vannak esetek, amikor ez a célunk, például egy szerverfolyamat fusson folyamatosan és van amikor egy bug, mert ott lesz végtelen ciklus, ahol nem akartunk. Saját példánkban ilyen amikor a PageRank algoritmus rázza az 1 liter vizet az internetben, de az iteráció csak nem akar konvergálni...

Egy mag 100 százalékban:

```
int
main ()
{
    for (;;) ;

    return 0;
}
```

vagy az olvashatóbb, de a programozók és fordítók (szabványok) között kevésbé hordozható

```
int
#include <stdbool.h>
main ()
{
    while(true);
}
```

```
    return 0;
}
```

Azért érdemes a `for(;;)` hagyományos formát használni, mert ez minden C szabvánnyal lefordul, másrészt a többi programozó azonnal látja, hogy az a végtelen ciklus szándékunk szerint végtelen és nem szoftverhiba. Mert ugye, ha a `while`-al trükközünk egy nem triviális `1` vagy `true` feltétellel, akkor ott egy másik, a forrást olvasó programozó nem látja azonnal a szándékunkat.

Egyébként a fordító a `for`-os és `while`-os ciklusból ugyanazt az assembly kódot fordítja:

```
$ gcc -S -o infty-f.S infty-f.c
$ gcc -S -o infty-w.S infty-w.c
$ diff infty-w.S infty-f.S
1c1
<  .file "infty-w.c"
---
>  .file "infty-f.c"
```

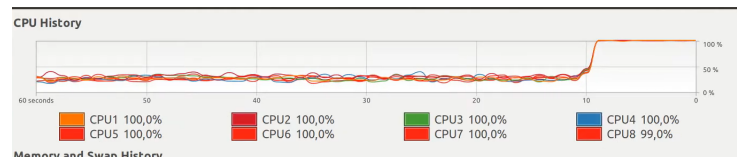
Egy mag 0 százalékban:

```
#include <unistd.h>
int
main ()
{
    for (;;)
        sleep(1);

    return 0;
}
```

Minden mag 100 százalékban:

```
#include <omp.h>
int
main ()
{
    #pragma omp parallel
    {
        for (;;)
        }
    return 0;
}
```



2.1. ábra. 100 százalékos magok

A **gcc infity-f.c -o infity-f -fopenmp** parancssorral készítve a futtathatót, majd futtatva, közben egy másik terminálban a **top** parancsot kiadva tanulmányozzuk, mennyi CPU-t használunk:

```
top - 20:09:06 up 3:35, 1 user, load average: 5,68, 2,91, 1,38
Tasks: 329 total, 2 running, 256 sleeping, 0 stopped, 1 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 : 99,7 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 : 99,7 us, 0,3 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu4 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu5 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu6 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu7 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem :16373532 total,11701240 free, 2254256 used, 2418036 buff/cache
KiB Swap:16724988 total,16724988 free, 0 used. 13751608 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
5850	batfai	20	0	68360	932	836	R	798,3	0,0	8:14.23	infity-f



Werkfilm

- <https://youtu.be/lvmi6tyz-nl>

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 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 pszeudókódjára:

```
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 épülő 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.

Alan Turing, angol matematikus, a Turing gép feltalálója is vizsgálta már ezt a problémát. Neki sikerült bebizonyítania, hogy nem létezik olyan algoritmus, aminek segítségével egy gép meg tudja állapítani egy másik gépről, hogy az le fog e fagyni, végtelen ciklusba fog e lépni.

2.3. Változók értékének felcserélése

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

Megoldás videó: <https://www.youtube.com/watch?v=E-A9iyUPRrA>

Megoldás forrása: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/csere.cpp](https://github.com/bhax/thematic_tutorials/blob/master/bhax_textbook_IgyNeveldaProgramozod/Turing/csere.cpp)

```
int a, b;
```

Deklarálom a szükséges változókat, a-t és b-t.

```
a=2;  
b=3;
```

A két változónak értéket adok.

```
a=a+b;  
b=a-b;  
a=a-b;
```

Ebben a részben cserélem fel a változók értékeit a következő képpen:

1. sor: Az a értéke $2+3=5$ lesz.
2. sor: A b értéke $5-3=2$ lesz.
3. sor: Az a értéke $5-2=3$ lesz.

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

Itt pedig kiírom a felcserélt értékeket.

Több megoldás is létezik 2 változó felcserélésére, ezért bemutatok egy másik megoldást is, ezt már C++ nyelven megírva.

```
int a, b;  
cin>>a;  
cin>>b;
```

Ebben a kódrészletben deklarálom a két változót, a-t és b-t, majd bekérek a felhasználótól két értéket.

```
a=a*b;  
b=a/b;  
a=a/b;
```

Szorzást használva felcserélem az értékeket. Például ha $a=2$ és $b=3$, akkor:

1.sor: Az a értéke $2*3=6$ lesz.

2.sor: A b értéke $6/3=2$ lesz.

3.sor: Az a értéke $6/2=3$ lesz.

Tehát az értékek felcserélődtek.

2.4. Labdapattogás

Először if-ekkel, majd bármiféle logikai utasítás vagy kifejezés használata nélkül írné 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 if-ekkel: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozasTuring/pattog.cpp](#)

```
int x = 0;  
int y = 0;
```

Megadom az x és y koordinátákat, innen fog indulni a labda

```
int irányx = 1;  
int irányy = 1;
```

Megadom az irányokat, ezek fogják befolyásolni a labda haladásának irányát és sebességét. Ha a szám pozitív, a labda koordinátái nőnek, ha negatív akkor pedig csökkennek. Minél nagyobb a szám, annál gyorsabban fogja elérni a terminál oldalát.

```
int sor;  
int oszlop;
```

Ezzel a két változóval fogom biztosítani, hogy a labda ne hagyja el a terminál területét

```
while (true) {  
  
    getmaxyx ( ablak, oszlop , sor );  
  
    mvprintw ( y, x, "X" );  
  
    refresh ();  
    usleep ( 1000000 );  
}
```

```
x = x + iranyx;
y = y + iranyy;

if ( x>=sor)
    iranyx *=-1;

if ( !(x>0) )
    iranyx *=-1;

if ( !(y>0) )
    iranyy *=-1;

if ( y>=oszlop )
    iranyy *=-1;

}
```

A getmaxyx függvény segítségével meg tudom állapítani a terminál méretét, míg a mvprintw függvény a labda útvonalát fogja jelezni.

A refres() usleep() parancsokkal beállítom, milyen sűrűn szeretném frissíteni a képernyőt, ami befolyásolja a labda sebességét.

A következő két sorban a koordinátákat fogom megváltoztatni, úgy, hogy hozzáadom az irány változót.

Az if-ek segítségével követem nyomon, hogy a labda elérte-e a terminál szélét, és ha igen, az irány változó értékét megszorozom (-1)el, hogy visszafelé haladjon a labda.

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: https://youtu.be/9KnMqrkj_kU, <https://youtu.be/KRZlt1ZJ3qk>, [Futás közben](#).

Megoldás forrása Bogomips: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozas/Bogomips.c](https://github.com/bhax/thematic_tutorials/blob/master/bhax_textbook_IgyNeveldaProgramozas/Bogomips.c)

Megoldás forrása Szóhossz: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozas/Eltolas.cpp](https://github.com/bhax/thematic_tutorials/blob/master/bhax_textbook_IgyNeveldaProgramozas/Eltolas.cpp)

```
int szo=1, db=1;
```

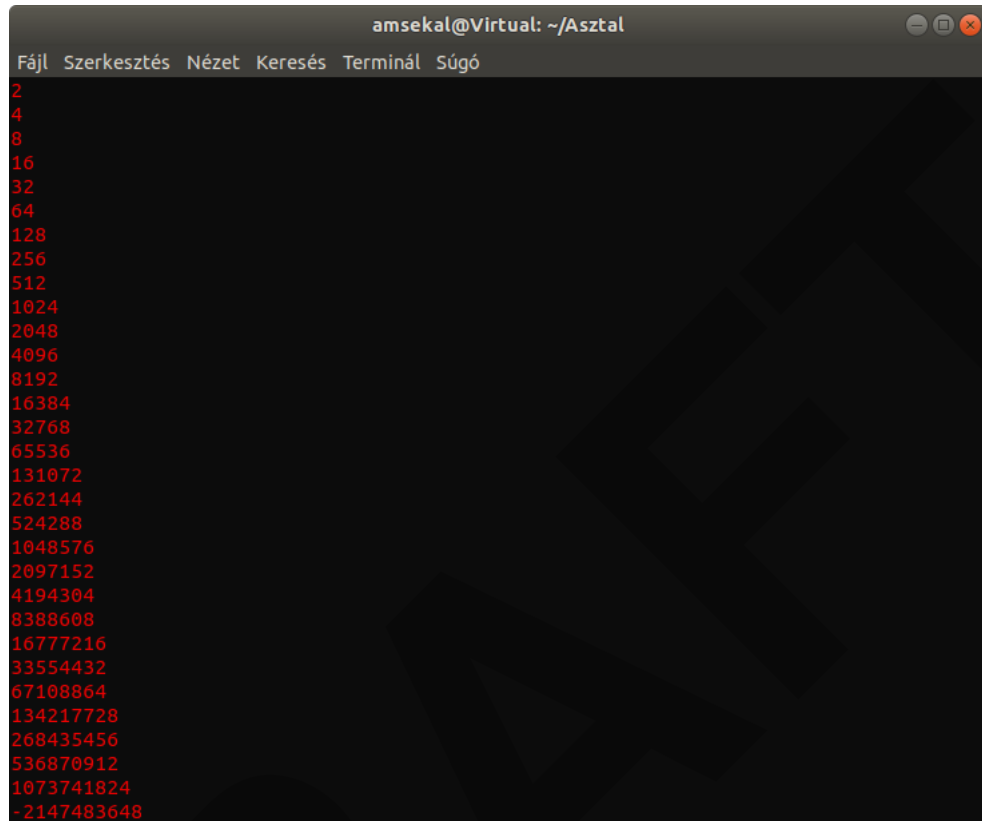
Két változót fogok használni: a szo változóban lesz az a szám, amit binárisan el fogok tolni, míg a db-ben fogom tárolni az eltolások számát.

Ezek után egy while ciklusban bináris shiftelés segítségével megnétem a szóhosszt. A db változót azért indítottam 1-től, mert a ciklusba nem a szo változó értékével kezdünk, hanem annak az eggyel eltolat változatával. Ezt az eltolást addig csinálom, amíg pozitív számokat kapok.

Példa bináris eltolásra:

Az 1 bináris értéke 1, ha ezt binárisa eggyel eltoljuk, az eredmény 10, ami a 2-nek felel meg. Ha még egy eltolást végzünk, a kapott szám 100 lesz, ami a 3-nak felel meg.

A program által elvégzett eltolások eredményei:



```
amsekal@Virtual: ~/Asztal
Fájl Szerkesztés Nézet Keresés Terminál Súgó
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
16777216
33554432
67108864
134217728
268435456
536870912
1073741824
~2147483648
```

A képen nem látható a db eredménye, ami az én esetemben 64 volt. Ez a szám jelzi, hogy hány bites operációs rendszert használunk.

2.6. Helló, Google!

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

Megoldás videó: https://youtu.be/9C_KZFyODLY

Megoldás forrása : [bhex/thematic-tutorials/bhex-textbook_IgyNeveldaProgramozod/Turing/pag.c](https://bhex.thematic-tutorials.com/bhex-textbook/IgyNeveldaProgramozod/Turing/pag.c)

A PageRank az internetes weboldalakat hivatott "rangsorolni", ezt a rendszert használja például a Google is. Az algoritmus lényege az, hogy végignézze egy oldalra hány hyperlink mutat, ugyanis az a koncepció, hogy egy 'A' oldalon csak akkor van egy másik 'B' oldalra vezető hyperlink, ha a 'B' oldalt jónak tartja az 'A' oldal szerkesztője. Tehát a hyperlink felfogható egyfajta szavazatként is.

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

void
PageRankKiiras (double tomb[], int db)
{
```



```
int i;
for (i=0; i<db; i++)
printf("PageRank [%d]: %lf\n", i+1, tomb[i]);
}

double tavolsag(double pr[],double pr_temp[],int db)
{
double tav = 0.0;
int i;
for(i=0;i<db;i++)
tav +=(pr[i] - pr_temp[i])*(pr[i] - pr_temp[i]);

return sqrt (tav);
}

int main(void)
{
double Lapok[4][4] = {
{1.0/2.0, 0.0, 1.0/3.0, 0.0},
{0.0, 1.0/3.0, 1.0/3.0, 1.0},
{0.0, 1.0/3.0, 0.0, 0.0},
{1.0/2.0, 1.0/3.0, 1.0/3.0, 0.0}
};

double PageRank[4] = {0.0, 0.0, 0.0, 0.0};
double Temp_PageRank[4] = {1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0};

long int i,j;
i=0; j=0;

for (;;)
{
for(i=0;i<4;i++)
PageRank[i] = Temp_PageRank[i];
for (i=0;i<4;i++)
{
double temp=0;
for (j=0;j<4;j++)
temp+=Lapok[i][j]*PageRank[j];
Temp_PageRank[i]=temp;
}

if ( tavolsag(PageRank,Temp_PageRank, 4) < 0.000001)
break;
}
PageRankKiiras (PageRank,4);
return 0;
}
```

A programban található mátrix mutatja meg nekünk, hogy melyik oldal melyik oldalra mutat:

Az első oszlop megmutatja, hogy az 'A' oldal önmagára és a 'D' weboldalra mutat, a 'B' oldal önmagára, a 'C' és a 'D' oldalra mutat, stb.

A PageRank, kezdetben üres tömb, szerepe az eredmények eltárolása lesz, a Temp_PageRank pedig, nevéből is ítélhetően ideiglenes eredményeket fog eltárolni, amiket majd később felhasználunk. Minden lap ugyanakkora értékkel indul, ezek fognak egy ciklusban egészen addig változni, amíg nem teljesül a ciklus megtörésének feltétele. Ha a ciklusnak vége, akkor megvannak az eredmények, amit a külön erre a feladatra szánt alprogrammal ki is íratunk.

2.7. 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://gitlab.com/nbatfai/bhax/tree/master/attention_raising/MontyHall_R

A Monty Hall paradoxon először az amerikai Let's Make a Deal nevű műsorban került bemutatásra, nevét is a műsor vezetőjéről kapta. Maga a paradoxon alaphelyzete nagyon egyszerű:

A játékos három ajtó közül választhatott: 2 ajtó mögött egy-egy kecske rejtőzött, míg a harmadik ajtó mögött egy autó.

Miután a játékos választott, a műsorvezető kinyitott egy ajtót, ami mögött egy kecske rejtőzött.

Ezek után a játékosnak lehetősége nyílt válztatni a döntésén.

A józan ész azt diktálja, hogy teljesen mindegy, hogy változtat-e a döntésén a játékos, 50 százaléka lesz nyerni. Azonban ez a gondolatmenet nem helyes, és ez a tény egy program segítségével is kimutatható. Míg a választás előtt minden ajtónak 1/3 esélye van, hogy ő rejtse az autót, miután a műsorvezető kinyitotta az egyik vesztes ajtót, a játékos által választottnak továbbra is 1/3 esélye van, de a másik zárt ajtónak már 2/3.

```
kiserletek_szama=10000
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) {

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

}

valtoztatesnyer = which(kiserlet==valtoztat)
```

A fenti kódban is pontosan ezt vizsgáljuk. Abban az esetben, ha a játékos jól választ, a nemvaltozasnyer lesz a helyes, míg ha a változtatás után fog a helyes ajtóra mutatni, akkor a változtatásnyer.

2.8. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/Primek_R

A természetes számok építőelemei a prímszámok. Abban az értelemben, hogy minden természetes szám előállítható prímszámok szorzataként. Például $12=2*2*3$, vagy például $33=3*11$.

Prímszám az a természetes szám, amely csak önmagával és eggyel osztható. Eukleidész görög matematikus már Krisztus előtt tudta, hogy végtelen sok prímszám van, de ma sem tudja senki, hogy végtelen sok ikerprím van-e. Két prím ikerprím, ha különbségük 2.

Két egymást követő páratlan prím között a legkisebb távolság a 2, a legnagyobb távolság viszont bármilyen nagy lehet! Ez utóbbit könnyű bebizonyítani. Legyen n egy tetszőlegesen nagy szám. Akkor szorozzuk össze $n+1$ -ig a számokat, azaz számoljuk ki az $1*2*3*\dots*(n-1)*n*(n+1)$ szorzatot, aminek a neve $(n+1)$ faktoriális, jele $(n+1)!$.

Majd vizsgáljuk meg az a sorozatot:

$(n+1)!+2, (n+1)!+3, \dots, (n+1)!+n, (n+1)!+(n+1)$ ez n db egymást követő szám, ezekre (a jól ismert bizonyítás szerint) rendre igaz, hogy

- $(n+1)!+2=1*2*3*\dots*(n-1)*n*(n+1)+2$, azaz $2*$ valamennyi $+2$, 2 többszöröse, így ami osztható kettővel

- $(n+1)!+3=1*2*3*\dots*(n-1)*n*(n+1)+3$, azaz $3*$ valamennyi $+3$, ami osztható hárommal
- ...
- $(n+1)!+(n-1)=1*2*3*\dots*(n-1)*n*(n+1)+(n-1)$, azaz $(n-1)*$ valamennyi $+(n-1)$, ami osztható $(n-1)$ -el
- $(n+1)!+n=1*2*3*\dots*(n-1)*n*(n+1)+n$, azaz $n*$ valamennyi $+n$, ami osztható n -el
- $(n+1)!+(n+1)=1*2*3*\dots*(n-1)*n*(n+1)+(n+1)$, azaz $(n+1)*$ valamennyi $+(n+1)$, ami osztható $(n+1)$ -el

tehát ebben a sorozatban egy prim nincs, akkor a $(n+1)!+2$ -nél kisebb első prim és a $(n+1)!+(n+1)$ -nél nagyobb első prim között a távolság legalább n .

Az ikerprímszám sejtés azzal foglalkozik, amikor a prímek közötti távolság 2. Azt mondja, hogy az egymástól 2 távolságra lévő prímek végtelen sokan vannak.

A Brun tétel azt mondja, hogy az ikerprímszámok reciprokaiból képzett sor összege, azaz a $(1/3+1/5)+(1/5+1/7)+(1/11+1/13)+\dots$ véges vagy végtelen sor konvergencia, ami azt jelenti, hogy ezek a törtek összeadva egy határt adnak ki pontosan vagy azt át nem lépve növekednek, ami határ számot B_2 Brun konstansnak neveznek. Tehát ez nem dönti el a több ezer éve nyitott kérdést, hogy az ikerprímszámok halmaza végtelen-e? Hiszen ha véges sok van és ezek reciprokait összeadjuk, akkor ugyanúgy nem lépjük át a B_2 Brun konstans értékét, mintha végtelen sok lenne, de ezek már csak olyan csökkenő mértékben járulnának hozzá a végtelen sor összegéhez, hogy így sem lépnék át a Brun konstans értékét.

Ebben a példában egy olyan programot készítettünk, amely közelíteni próbálja a Brun konstans értékét. A repó [bhax/attention_raising/Primek_R/stp.r](https://github.com/bhax/attention_raising/Primek_R/stp.r) nevű állománya kiszámolja az ikerprímeket, összegzi a reciprokaikat és vizualizálja a kapott részeredményt.

```
# Copyright (C) 2019 Dr. Norbert Bاتفai, nbatfai@gmail.com
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>

library(matlab)

stp <- 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 = stp)
plot(x,y,type="b")
```

Soronként értelmezzük ezt a programot:

```
primes = primes(13)
```

Kiszámolja a megadott számig a prímeket.

```
> primes=primes(13)
> primes
[1]  2  3  5  7 11 13
```

```
diff = primes[2:length(primes)]-primes[1:length(primes)-1]
```

```
> diff = primes[2:length(primes)]-primes[1:length(primes)-1]
> diff
[1] 1 2 2 4 2
```

Az egymást követő prímek különbségét képzi, tehát 3-2, 5-3, 7-5, 11-7, 13-11.

```
idx = which(diff==2)
```

```
> idx = which(diff==2)
> idx
[1] 2 3 5
```

Megnézi a diff-ben, hogy melyiknél lett kettő az eredmény, mert azok az ikerprím párok, ahol ez igaz. Ez a diff-ben lévő 3-2, 5-3, 7-5, 11-7, 13-11 különbségek közül ez a 2., 3. és 5. indexűre teljesül.

```
t1primes = primes[idx]
```

Kivette a primes-ből a párok első tagját.

```
t2primes = primes[idx]+2
```

A párok második tagját az első tagok kettő hozzáadásával képezzük.

```
rt1plust2 = 1/t1primes+1/t2primes
```

Az $1/t1primes$ a $t1primes$ 3,5,11 értékéből az alábbi reciprokokat képzi:

```
> 1/t1primes  
[1] 0.33333333 0.20000000 0.09090909
```

Az $1/t2primes$ a $t2primes$ 5,7,13 értékéből az alábbi reciprokokat képz:

```
> 1/t2primes  
[1] 0.20000000 0.14285714 0.07692308
```

Az $1/t1primes + 1/t2primes$ pedig ezeket a törteket rendre összeadja.

```
> 1/t1primes+1/t2primes  
[1] 0.53333333 0.3428571 0.1678322
```

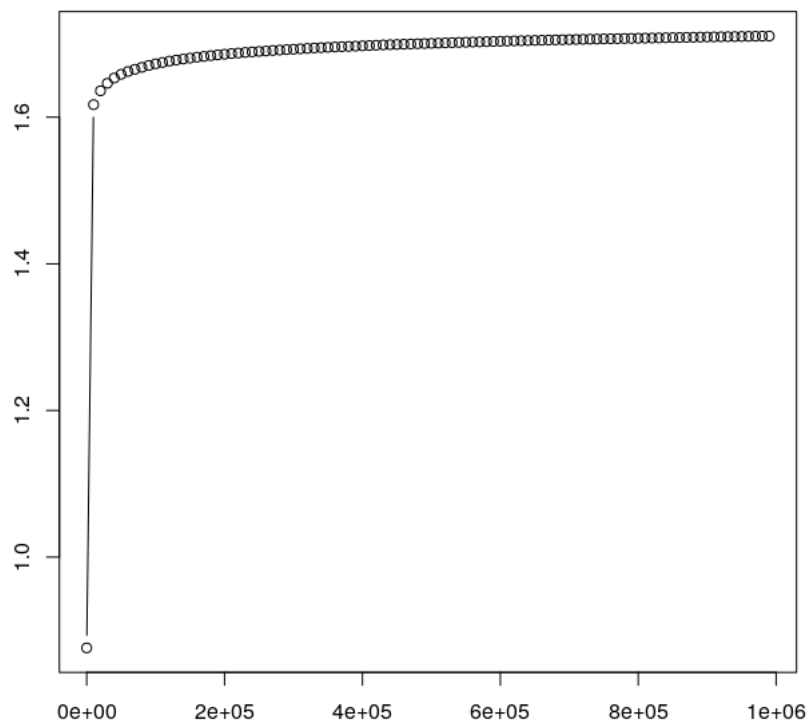
Nincs más dolgunk, mint ezeket a törteket összeadni a `sum` függvénnyel.

```
sum(rt1plust2)
```

```
> sum(rt1plust2)  
[1] 1.044023
```

A következő ábra azt mutatja, hogy a szumma értéke, hogyan nő, egy határértékhez tart, a B_2 Brun konstanshoz. Ezt ezzel a csipettel rajzoltuk ki, ahol először a fenti számítást 13-ig végezzük, majd 10013, majd 20013-ig, egészen 990013-ig, azaz közel 1 millióig. Vegyük észre, hogy az ábra első köre, a 13 értékhez tartozó 1.044023.

```
x=seq(13, 1000000, by=10000)  
y=sapply(x, FUN = stp)  
plot(x,y,type="b")
```

2.2. ábra. A B_2 konstans közelítése**Werkfilm**

- <https://youtu.be/VkMFrgBhN1g>
- <https://youtu.be/aF4YK6mBwf4>

2.9. Minecraft MALMÖ -

Forrás link: [Github link](#)

A feladatban az a célunk, hogy megoldjuk, hogy a minecraft karakterünk csigaszerű mozgással haladjon egyre feljebb az arénában, addig amíg nem találkozik a lávával, ami után értelem szerűen meghal. Ha fallal találkozik, akkor felugrik rá és elfordul, így halad az arénában.

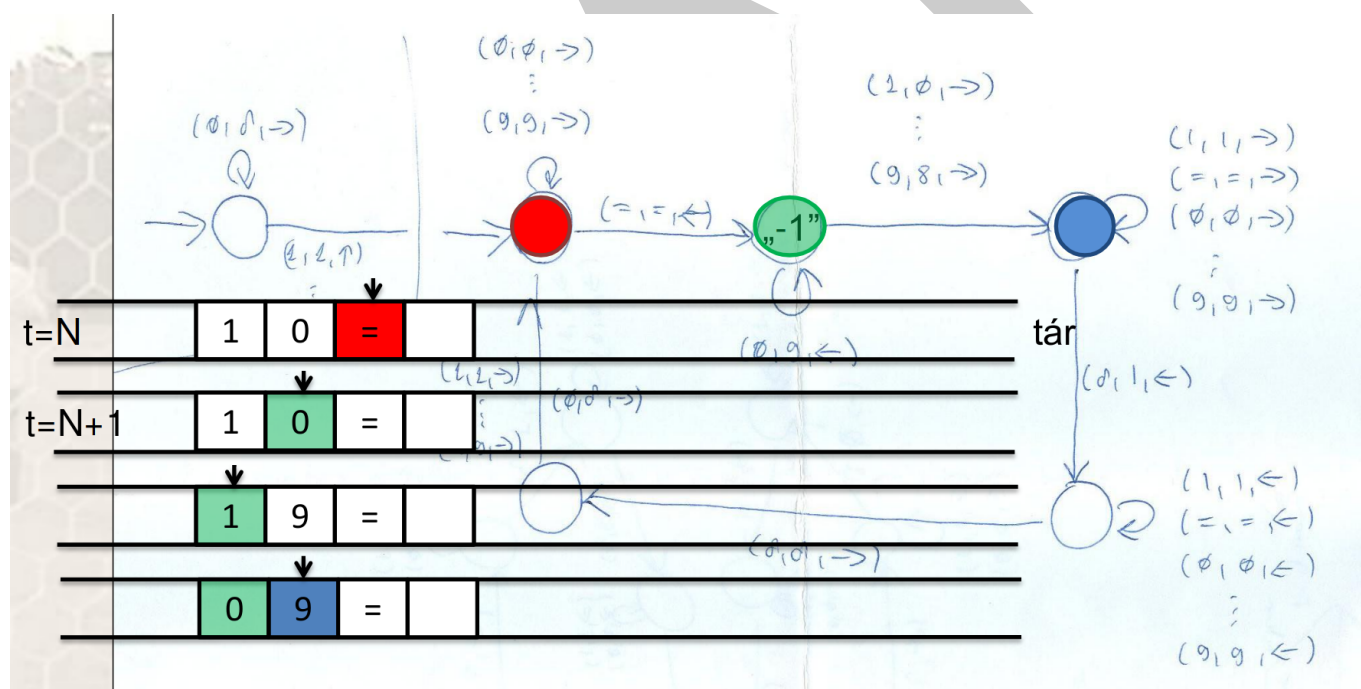
3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet grájával megadva írd meg ezt a gépet!

Állapotátmenet gráf: [27.dia](#)



3.1. ábra. A Turing gép működése

A Turing gép Alan Turing találmánya, ezt az automatikus számolás céljával hozta létre, akkoriban ugyanis még nem voltak számológépek.

Működése rétegelt: Minden időben van egy helyzete a memóriaszalagon, van aktuális állapota amit programozással adunk meg. A gép lépésenként olvas egy szimbólumot, majd adott feltételek segítségével választ

3 lehetőség közül: Beírja a szimbólumot, Pozíciót vált vagy Stop állapotba lép. Így működik a decimálisból unárisba váltó gép is.

Egy unáris számot úgy építünk fel, hogy annyi vonalat (1-es számjegy) húzunk, ahányszor a számból levonható 1. Ha vissza szeretnénk kapni a decimális számot, ezt az algoritmus kell elvégeznünk visszafelé.

A Decimális számrendszer a már mindenki által jól ismert és használt számrendszer, ahol 10 számjegy van, ezekből épülnek fel a számok. Ezek a számjegyek: 0,1,2,3,4,5,6,7,8,9.

Az Unáris számrendszer is egy ismert és használt számrendszer, csak név szerint sokan nem ismerik. Ez egy olyan számrendszer, ahol csak egy számjegyünk van: az egyes. Ezt szokták vonalakkal is jelölni. A vonalak száma jelzi a szám értékét. A gyerekek amikor az ujjukat használják számolásra, akkor unáris számrendszert használnak.

Például, ha a képen látható 10-es számot szeretnénk unárisba átalakítani, ez lenne a menete: a szám beolvasása után a program elér az egyenlőségjelhez, ami azt jelzi, hogy állapotváltás fog következni, ezért visszalép. A tizedesek helyén szereplő 1-est 0-ra cseréli, míg az egyesek helyén elhelyezkedő 0-ást egy kilencsere, így a decimális számunk jelenleg 09, ami 9. A lépés végetért visszatérünk az egyenlőségjel elé, és mivel elvégeztünk egy mínusz egyes állapotváltást, az unáris számunk értéke is növekszik. Mivel eddig egyetlen darab 1-es számjegy sem szerepelt az unáris számban, ez lesz az első, az unáris számunk jelen pillanatban: 1 Ezek után a fentebb leírtak alapján működik a program.

Következő lépés: Decimális szám: 8, Unáris szám: 11

Következő lépés: Decimális szám: 7, Unáris szám: 111

Következő lépés: Decimális szám: 6, Unáris szám: 1111

Következő lépés: Decimális szám: 5, Unáris szám: 11111

Következő lépés: Decimális szám: 4, Unáris szám: 111111

Következő lépés: Decimális szám: 3, Unáris szám: 1111111

Következő lépés: Decimális szám: 2, Unáris szám: 11111111

Következő lépés: Decimális szám: 1, Unáris szám: 111111111

Következő lépés: Decimális szám: 0, Unáris szám: 1111111111

Abban a pillanatban, ahogy lenulláztuk a megadott decimális számot, megkapjuk unárisban.

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

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

Forrás: [30.-32. dia](#)

Ez egy generatív nyelv. Ennek a nyelvnek alapja a terminális jel/szimbólum. A környezetfüggetlen nyelvten, olyan nyelvten, amelyben változók helyettési szabályok sorozatával konstansokból álló mondatokat kapunk. A fent említett nyelv környezetfüggő

1. környezetfüggő generatív gramatika

Első sorban szükségünk lesz a nyelv megalkotásához, nyelvi konstansokra és változókra.

A, B, C változók
a, b, c konstansok

Szükségünk lesz még levezetési szabályokra is, amelyeknek köszönhetően eljutunk a kívánt alakhoz.

```
A - abc, A - aBbc, Bb - bB, Bc - Cbcc, bC - Cb, aC - aaB, aC - aa
```

Vegyük az A változót, célunk, hogy úgy alakítsuk, hogy a kívánt alakhoz hasonló alakot vegyen fel.

```
A (A - aBbc) // A helyett aBbc-t fogunk írni
aBbc (Bb - bB)
abBc (Bc - Cbcc)
abCbcc (bC - Cb)
aCbbcc (aC - aa)
aabbcc
```

Az utolsó sorban megtekinthető a végleges alak, ami $a^n b^n c^n$ formában van.

2. környezetfüggő generatív gramatika

Első sorban szükségünk lesz a nyelv megalkotásához, nyelvi konstansokra és változókra.

```
A, B, C változók
a, b, c konstansok
```

Szükségünk lesz még levezetési szabályokra is, amelyeknek köszönhetően eljutunk a kívánt alakhoz.

```
A - aAB, A - aC, CB - bCc, cB - Bc, C - bc
```

Vegyük az A változót, célunk, hogy úgy alakítsuk, hogy a kívánt alakhoz hasonló alakot vegyen fel.

```
A (A - aAB)
aAB (A - aC)
aaCB (CB - bCc)
aabCc (C - bc)
aabbcc
```

Az utolsó sorban megtekinthető a végleges alak, ami $a^n b^n c^n$ formában van.

A két nyelv leírása és a szabályok megtalálhatóak [itt](#)

3.3. Hivatkozási nyelv

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

Megoldás videó: <https://www.youtube.com/watch?v=KAhxMUW1rVc>

Megoldás forrása: bhas.com/thematic_tutorials/bhas_textbook_IgyNeveldaProgramozod/Chomsky/hivatkozas.c

Tutoráltam: Deák Rúben

Először írok egy for ciklust.

```
%{  
#include <stdio.h>  
  
int main()  
{int s=0;  
for ( int i=0;i<5;i++)  
s++;  
printf("Vege\n");  
return 0;  
}
```

A leírt program nem fut le C98-as szabványt használva, mivel ebben a szabványban még nem volt lehetséges ciklusváltozó deklarálása. Ha ezt a kódot szeretnénk C89-es szabvánnyal futtatni, az alábbi módon lehetne:

```
%{ #include <stdio.h>  
  
int main()  
{int i,s=0;  
for ( i=0;i<5;i++)  
s++;  
printf("Vege\n");  
return 0;  
}
```

3.4. Saját lexikális elemző

Írj olyan programot, ami számolja a bemenetén megjelenő valós számokat! Nem elfogadható olyan megoldás, amely maga olvassa betűként a bemenetet, a feladat lényege, hogy lexert használjunk, azaz óriások vállán álljunk és ne kispályázzunk!

Megoldás videó: https://youtu.be/9KnMqrkj_kU (15:01-től). Futás közben: https://www.youtube.com/watch?v=_

Megoldás forrása: [bham/academic/tutorials/bham_textbook/IgyNeveldeProgramozod/Chomsky/realnumber.1](https://bham.ac.uk/~bham/academic/tutorials/bham_textbook/IgyNeveldeProgramozod/Chomsky/realnumber.1)

A használt program 3 részből áll:

```
%{  
#include <stdio.h>  
int realnumbers = 0;  
%}
```

Az első rész sima C kód, amelyben deklarálunk egy `realnumbers` nevű változót, ami a valós számok előfordulását fogja számolni. Ezt a változót 0-tól indítjuk.

```
%{  
digit [0-9]  
%%  
{digit}* (\.{digit}+)? {++realnumbers;
```

```
printf("[realnum=%s %f]", yytext, atof(yytext));}  
%%
```

A második részben már nem csak C kód található. Itt megadjuk a keresési feltételeket: egy olyan szövegrészletet keresünk, ahol 0 vagy több számjegy van (ezt jelzi a digit *), amit egy pont követ, ami után 1 vagy annál több számjegyet találunk. A pont előtti \ azt a célt szolgálja, hogy a program felismerje, hogy mi a pontra, mint karakterre gondolunk, nem egy bármilyen karakterre. Ezek után ismét C kód következik, ezzel kiírjuk a szövegbe, ha találunk egy valós számot, és növeljük a realnumbers változó értékét eggyel.

```
%{  
int  
main ()  
{  
    yylex ();  
    printf("The number of real numbers is %d\n", realnumbers);  
    return 0;  
}
```

A harmadik kódrészletben hívjuk meg a lexikális elemzőt, ami az általunk megadott feltételek alapján fog keresni a szövegben. Egy lexikális elemző nagyon hasznos tud lenni, ugyanis a segítségével a programunknak nem betűnként kell beolvasnia a forrást, így sok időt és erőforrást tudunk megspórolni.

3.5. Leetspeak

Lexelj össze egy l33t ciphert!

Megoldás videó: https://youtu.be/06C_PqDpD_k

Futás közben: <https://www.youtube.com/watch?v=-OrREI6GZKI>

Megoldás forrása: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/1337d1c7.1](https://github.com/bhax/thematic_tutorials/blob/master/bhax_textbook_IgyNeveldaProgramozod/Chomsky/1337d1c7.1)

```
%{  
    #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', {"8", "8", "|3", "|"}},  
        {'c', {"c", "(", "<", "{"}},  
        {'d', {"d", "|)", "|", "|"}},
```

```

{'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", "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"}}

```

```

// https://simple.wikipedia.org/wiki/Leet
};

```

```

%}

```

```

%%

```

```

. {

```

```

    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;
}
```

A leet nyelv lényege, hogy a betűket rájuk hasonló számmal vagy más karakterekkel helyettesítsünk, például az a helyett használhadjuk: 4, @, /-\\.. A program hasonlít az előző feladathoz, azzal a különbséggel, hogy most nem számokat keresünk, hanem betűket. Létrehozunk egy tömböt, ahol eltároljuk a betűket és leet megfelelőiket. Ezután egy for ciklus segítségével megkeressük a betűt ebben a tömbben, és egy randomizáló rész segítségével véletlenszerűen választunk egy leet megfelelőt a megadottakból. Ha egy betűnek nincs leet megfelelője, változtatás nélkül kiíratásra kerül.

3.6. A források olvasása

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

```
if(signal(SIGINT, jelkezelő)==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 jelkezelő 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, jelkezelő);
//INTERACT signal esetén a jelkezelő eldönti, hogyan reagáljon a ←
program
```

ii.

```
for(i=0; i<5; ++i)
//Indítunk egy for ciklust, ahol i 0-tól indul, és a ciklusban lévő ←
utasítások elvégzése előtt nő az értéke.
```

iii.

```
for(i=0; i<5; i++)
//For ciklus, ahol az i értéke a ciklusban lévő utasítások elégezése ←
után nő
```

iv.

```
for(i=0; i<5; tomb[i] = i++)
// Egy for ciklus, ahol egy tömbi i-edik eleme felveszi az i+1 értéket ←
, és az i nő a ciklus végén
```

v.

```
for(i=0; i<n && (*d++ = *s++); ++i)
//Egy for ciklus, ami addig megy amíg i kisebb mint n és a d pointer ←
egyenlő az s pointerrel.
```

vi.

```
printf("%d %d", f(a, ++a), f(++a, a));
//Kiirunk két számot, amit az f függvény állít elő.
```

vii.

```
printf("%d %d", f(a), a);
//Kiirunk két számot, az egyiket az f függvény állítja elő az a ←
változó segítségével, a másik szám pedig az a változó.
```

viii.

```
printf("%d %d", f(&a), a);
//Kiir két számot, az egyiket az f függvény segítségével, a másik ←
pedig az a értéke lesz, ami ha a függvényben megváltozott, a ←
változás megmarad.
```

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

```

$$\begin{aligned} & \$(\backslash\text{forall } x \backslash\text{exists } y ((x < y) \wedge (y \text{ \texttt{prím}}))) \$ \\ & \$(\backslash\text{forall } x \backslash\text{exists } y ((x < y) \wedge (y \text{ \texttt{prím}})) \wedge (S y \text{ \texttt{prím}})) \leftrightarrow \\ & \quad ) \$ \\ & \$(\backslash\text{exists } y \backslash\text{forall } x (x \text{ \texttt{prím}}) \supset (x < y)) \$ \\ & \$(\backslash\text{exists } y \backslash\text{forall } x (y < x) \supset \neg (x \text{ \texttt{prím}})) \$ \end{aligned}$$

```

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/MatLog_LaTeX

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

1. Minden x esetén igaz, hogy van olyan y , amely nagyobb mint x és primszám.
2. Minden x esetén igaz, hogy van olyan y , amely nagyobb mint x , primszám és az $y+2$ is primszám.
3. Létezik olyan y , amely minden x primnél nagyobb
4. Létezik olyan y , amelynél egyik x sem kisebb és prim.

3.8. Deklaráció

Megoldás videó: <https://www.youtube.com/watch?v=rjaIqjzDVC4>

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/deklaracio.cpp

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciája
- egészek tömbje
- egészek tömbjének referenciája (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

Mit vezetnek be a programba a következő nevek?

- `int a; // integer típusú változó`
- `int *b = &a; // b pointer, ami a memóriacímére mutat`
- `int &r = a; // r hivatkozás a-ra`
- `int c[5]; // c tömb`
- `int (&tr)[5] = c; // tr tömb referencia c tömbre`
- `int *d[5]; // d pointerekből álló tömb`
- `int *h (); // egészre mutató visszaadó függvény`
- `int *(*l) (); // mutatóra mutató függvény`
- `int (*v (int c)) (int a, int b) // Függvénytmutató, ami egy egészet ↵
visszaadó függvényre mutató mutatóval visszatérő függvény`
- `int ((*z) (int)) (int, int); // Függvénytmutató, ami egy egészet visszaadó ↵
függvényre mutató mutatót visszaadó függvényre mutat`

Az utolsó két deklarációs példa demonstrálására két olyan kódot írtunk, amelyek összehasonlítása azt mutatja meg, hogy miért érdemes a **typedef** használata: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr.c](https://github.com/bhax/thematic_tutorials/blob/master/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr.c), [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr2.c](https://github.com/bhax/thematic_tutorials/blob/master/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr2.c).

```
#include <stdio.h>

int
sum (int a, int b)
{
    return a + b;
}

int
mul (int a, int b)
{
    return a * b;
}

int (*sumormul (int c)) (int a, int b)
{
    if (c)
```

```
        return mul;
    else
        return sum;
}

int
main ()
{
    int (*f) (int, int);

    f = sum;

    printf ("%d\n", f (2, 3));

    int (*(g) (int)) (int, int);

    g = sumormul;

    f = *g (42);

    printf ("%d\n", f (2, 3));

    return 0;
}
```

```
#include <stdio.h>

typedef int (*F) (int, int);
typedef int (*(G) (int)) (int, int);

int
sum (int a, int b)
{
    return a + b;
}

int
mul (int a, int b)
{
    return a * b;
}

F sumormul (int c)
{
    if (c)
        return mul;
    else
        return sum;
}
```

```
}  
  
int  
main ()  
{  
  
    F f = sum;  
  
    printf ("%d\n", f (2, 3));  
  
    G g = sumormul;  
  
    f = *g (42);  
  
    printf ("%d\n", f (2, 3));  
  
    return 0;  
}
```

A feladat megoldása alatt rengeteg dolgot tudtam meg és tanultam. Ilyen például a mutatók használata.

Mint kiderült a mutatókat rengeteg féle képpen lehet használni, a lehetőségek határtalanok. Az embernek viszont rengetek időbe kerül megfigyeltetni hogy mit is akar a kódrészlet, ha több mutató is szerepel benne, legalábbis magamon azt vettem észre, hogy minél több mutató van és minél összetettebb dolgokra mutatnak, annál nehezebb rájönni, pontosan mit is lehet vele elérni.

A mutatók használata rengeteg előnnyel jár, viszont ha túl bonyolult helyeken akarjuk őket használni, nem biztos, hogy átlátható és megérthető lesz a programunk.

3.9. Minecraft MALMÖ - Csiga diszkrét

Forrás link: [Github link](#)

A feladatban az a célunk, hogy megoldjuk, hogy a minecraft karakterünk csigaszerű mozgással haladjon egyre feljebb az arénában, addig amíg nem találkozik a lávával, ami után értelem szerűen meghal, hasonlóan az első MALMÖs feladathoz. Itt azonban diszkrét mozgási parancsokat használunk, számoljuk a karakterünk lépéseit, és az alapján döntünk a következő mozdulatról.

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> Illetve <https://youtu.be/-mderKzd9gx8>.

Megoldás forrása: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/tm.c](https://github.com/bhax/thematic_tutorials/blob/master/bhax_textbook_IgyNeveldaProgramozod/Caesar/tm.c)

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

int
main ()
{
    int nr = 5;
    double **tm;
}
```

Meghatározom, hogy a háromszögmátrix, amit használni fogok, 5 soros legyen és deklarállok egy tm pointer-t, ami pointerre fog mutatni

```
#include <stdio.h>
if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
{
    return -1;
}
}
```

Ezzel az íffel megpróbálok helyet foglalni a memóriában a tm-nek, ha ez nem sikerül, -1-et fogunk visszaadni, ez fogja számunkra jelezni, hogy a művelet nem sikerült

```
#include <stdio.h>
for (int i = 0; i < nr; ++i)
```

```
{
    if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL ↵
    )
    {
        return -1;
    }
}
}
```

Hasonlóan az előző ifhez, helyet foglalunk a tm-ben a pointereknek, ha ez nem sikerül, a visszatérési érték -1 lesz.

```
#include <stdio.h>
for (int i = 0; i < nr; ++i)
    for (int j = 0; j < i + 1; ++j)
        tm[i][j] = i * (i + 1) / 2 + j;
}
```

Értéket adunk az alsó háromszögmátrixunknak

```
#include <stdio.h>
for (int i = 0; i < nr; ++i)
{
    for (int j = 0; j < i + 1; ++j)
        printf ("%f, ", tm[i][j]);
    printf ("\n");
}
}
```

Kiiratjuk az alsó héromszögmátrixunkat

```
#include <stdio.h>
tm[3][0] = 42.0;
(*(tm + 3))[1] = 43.0;
*(tm[3] + 2) = 44.0;
*(*(tm + 3) + 3) = 45.0;
}
```

Értéket adunk a negyedik sor első (42), második (43), harmadik(44) és ötödik(45) elemének. A különbség, hogy az első esetben konkrétan megmondjuk melyik elem értékét változtatjuk, a többi esetben a pointerek mozgatásának segítségével változtatunk az értéken.

```
#include <stdio.h>
for (int i = 0; i < nr; ++i)
{
    for (int j = 0; j < i + 1; ++j)
        printf ("%f, ", tm[i][j]);
    printf ("\n");
}
}
```

```

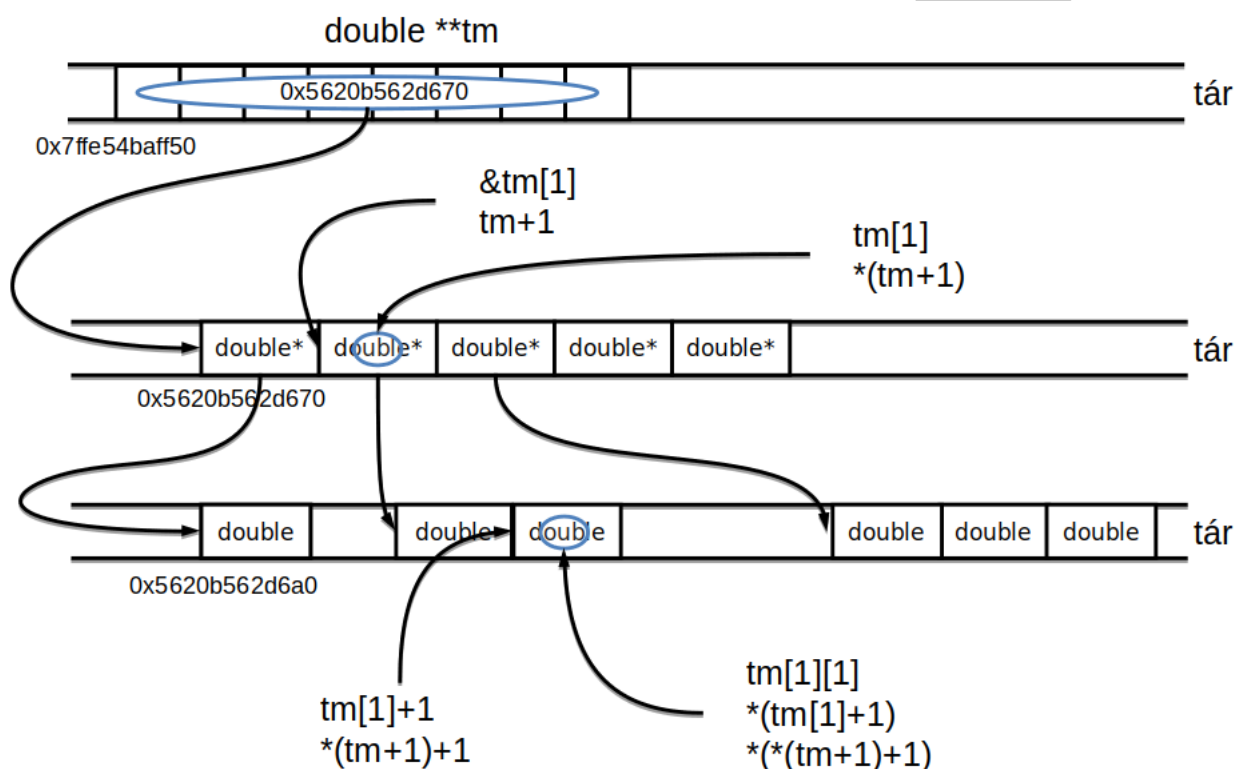
for (int i = 0; i < nr; ++i)
    free (tm[i]);

free (tm);

return 0;
}

```

Kiiratjuk a mátrixot az új értékekkel, majd felszabadítjuk a tárhelyet.



4.1. ábra. A `double **` háromszögmátrix a memóriában

A `tm` különlegessége, hogy egy olyan vektorra mutat, amiben szintén pointerek vannak. Ahogy a felső ábrán is látható a `tm[1]` egy `double` pointert adna vissza. A mások pointer a mátrix egyik sorára mutat, tehát a `tm[1][1]` a második sor második elemére mutat, mivel a számozás 0-tól kezdődik. A mátrix elemeire többféleképpen is hivatkozhatunk a pointerek segítségével, ez a feladat ezt hivatott bemutatni. Ámbár rengeteg lehetőségünk van megnevezni a mátrix egy elemét, szerintem a legjobb döntés a legrövidebb és legkönnyebben megérthető leírási formát alkalmazni, ami a `tm[x][y]`

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó: https://www.youtube.com/watch?v=RribkJ_s-7w.

Megoldás forrása: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/e.c](https://bham.ac.uk/~bham/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/e.c).

```
#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];
}
```

Definiálok 2 állandó értéket, a MAX_KULCS és a BUFFER_MERET néven. A MAX_KULCS fogja jelenteni, hogy maximum hány karakterből állhat a kulcs, a BUFFER_MERET pedig, hogy hány karaktert tárolhatunk a bufferben.

```
int kulcs_index = 0;
int olvasott_bajtok = 0;

int kulcs_meret = strlen (argv[1]);
strncpy (kulcs, argv[1], MAX_KULCS);
}
```

Mivel a program meghívásakor adom meg a használni kívánt kulcsot, ezért használnom kell az argv változót, ami azt fogja mutatni hanyadik szónál tartunk. Az első szó, azaz a 0. elem a program indítása lesz, a második szó maga a kulcs, a harmadik a titkosítani kívánt file neve, míg a negyedik a kimenet neve.

```
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;
    }
}
```

Ameddig tart a szöveg, a kulcs és a bitenkénti eltolás segítségével megváltoztatjuk a szöveget, mondhatni beleolvasztjuk a kulcsot a szövegbe. Ha ezt a műveletet mégegyszer használnánk a titkosított szövegen, az eredetit kapnánk vissza.

```
write (1, buffer, olvasott_bajtok);
```

```
}  
}  
}
```

Végül kiiratjuk a titkosított szöveget.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó: <https://www.youtube.com/watch?v=6cJVzgdVa5U>.

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html#exor_titkosito

A java EXOR törő is hasonlóan működik a C-s változathoz, lehetséges például egy Java exorral titkosított fület a C titkosítóban feltörni, természetesen csak akkor, ha ismert a kulcs.

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

Létrehozzuk az ExorTitkosító osztályt, amibe a program elindításakor elmetjük a kulcsot (kulcsSzöveg), azt a szöveget amit titkosítani akarunk (bejövőCsatorna) és létrehozzuk a kimeneti file-t (kimenőCsatorna). Az utolsó sor a hibakezelést végzi.

```
        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);  
  
        }  
  
    }  
}
```


Hasonlóan a C EXOR titkosítóhoz, itt is használni fogunk egy tömböt amiben eltároljuk a kulcsot, és egy tömböt, amibe beolvassuk a bejövő szöveget, ez lesz a buffer. Egy while ciklusban végezzük ez a logikai vagy műveletet, addig, amíg van szöveg a bufferben. Ha befejeztük a szöveg titkosítását, a kimenetet kiírjuk a megfelelő helyre.

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

Már csak a főfüggvény megírása maradt. A main megpróbálja elindítani a programot a try utasítással. A próba során meghívja a titkosítót. Ha valami hiba van, a catch fogja elkapni az üzenetet.

4.4. C EXOR törő

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

Megoldás videó:<https://www.youtube.com/watch?v=sRDQ0CX1hko> .

Megoldás forrása: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/t.c](https://github.com/bhax/thematic_tutorials/blob/master/bhax_textbook_IgyNeveldaProgramozod/Caesar/t.c)

Az egyszerűség kedvéért, és hogy a program biztosan lefusson bármilyen gépen, anélkül hogy túlságosan megterhelje a rendszert egy kicsit egyszerűsítet a problémán. A kódban maximum 4 betűs kulcsokat keresek, úgy, hogy magukat a betűket ismerem a kulcsból, csak a sorrendet nem. Ezzel megkimélem a gyengébb gépeket, de emiatt a program mechanizmusa nem változik.

```
#define MAX_TITKOS 4096  
#define OLVASAS_BUFFER 256  
#define KULCS_MERET 4  
#define _GNU_SOURCE  
  
#include <stdio.h>  
#include <unistd.h>  
#include <string.h>  
  
double
```

```
atlagos_szo_hossz (const char *titkos, int titkos_meret)
{
    int sz = 0;
    for (int i = 0; i < titkos_meret; ++i)
        if (titkos[i] == ' ')
            ++sz;

    return (double) titkos_meret / sz;
}

int
tiszta_lehet (const char *titkos, int titkos_meret)
{
    double szohossz = atlagos_szo_hossz (titkos, titkos_meret);

    return szohossz > 6.0 && szohossz < 9.0
        && strcasestr (titkos, "hogy") && strcasestr (titkos, "nem")
        && strcasestr (titkos, "az") && strcasestr (titkos, "ha");
}
}
```

Ismét szükségem lesz egy alapról definiált számra, ami meghatározza a kulcs méretét és a buffert. Két alprogramot fogok használni, hogy megállapítsam, a kigenerált kulcs jó e. Ezt úgy teszem meg, hogy meg-nézem szerepel-e a szavak között a "hogy", "nem", "az" és a "ha" szó. Ezek a szavak szoktak a legtöbbször szerepelni a mondatokban, így ha a szövegünk tartalmazza ezeket a szavakat, majdnem biztosak lehetünk benne, hogy jó kulcsot találtunk.

```
void
xor (const char kulcs[], int kulcs_meret, char titkos[], int titkos_meret)
{
    int kulcs_index = 0;

    for (int i = 0; i < titkos_meret; ++i)
    {
        titkos[i] = titkos[i] ^ kulcs[kulcs_index];
        kulcs_index = (kulcs_index + 1) % kulcs_meret;
    }
}

int
xor_tores (const char kulcs[], int kulcs_meret, char titkos[],
            int titkos_meret)
{
}
```

```
    exor (kulcs, kulcs_meret, titkos, titkos_meret);

    return tiszta_lehet (titkos, titkos_meret);
}
}
```

Feltörés esetén is a bitenkénti vagyot, ezzel kapjuk meg a titkosított szöveg formáját, ha használjuk rajta a kulcsot.

```
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;

    for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
        titkos[p - titkos + i] = '\0';
    char str[4] = {'t', 'i', 'k', 'a'};

    for (int ii = 0; ii <= 4; ++ii)
    for (int li = 0; li <= 4; ++li)
        for (int ki = 0; ki <= 4; ++ki)
            for (int ji = 0; ji <= 4; ++ji)
                {
                    kulcs[0] = str[ii];
                    kulcs[1] = str[li];
                    kulcs[2] = str[ki];
                    kulcs[3] = str[ji];

                    if (exor_tores (kulcs, KULCS_MERET,
                                     titkos, p - titkos))
                        printf
                            ("Kulcs: [%c%c%c%c]\nTiszta szoveg: [%s]\n",
                             kulcs[ii], kulcs[li], kulcs[ki],
                             kulcs[ji], titkos);
                }
}
```

```
        exor (kulcs, KULCS_MERET, titkos, p - ←  
            titkos);  
    }  
  
    return 0;  
}
```

Mostmár csak meg kell hívni az elkészített alprogramokat a főfüggvényben. Itt adom meg azt is, hogy melyik az a 4 karakter aminek szeretném ha kipróbálná az összes variációját, jelen esetben ezek a t,i,k,a . Négy ciklust használok, ezek segítségével írom át a szöveget. Ahogy megkapom a fordítást és tiszta is, megvan a végeredményem, a program feltörte a kódot. Ha nem négy megadott betűre szeretném lefuttatni, annyi dolgom van, hogy az összes lehetséges betű kombinációját kell vennem, ez megoldható ciklusok használatával is. A kód feltörése sokkal bonyolultabb és hosszabb folyamat, mint a titkosítás, de ha az ember egy jó géppel rendelkezik, akkor egy 16 betűs kulcs feltörése sem jelent nagy kihívást, pontosan ezért nem használják ezt a fajta titkosítást a modern időkben.

4.5. Neurális OR, AND és EXOR kapu

R

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

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

Neurális hálók alatt két dolgot érthetünk: biológiai értelemben véve, a neurális hálózatot neuronok alkotják, amik összeköttetésben vannak egymással - ezek a hálók megtalálhatóak az idegrendszerekben - informatikai értelemben véve pedig mesterséges neurális hálózatra gondolunk, ez lehet akár gép vagy program, ami egy biológiai neurális hálózatot mintáz. A mesterséges neurális hálókat általában a technikában használják.

A programban mi egy mesterséges neurális hálózattal foglalkozunk, neki próbálunk "megtanítani" különböző műveleteket, név szerint az OR-t, az AND-et és az EXOR-t. Azért, hogy ezeket a műveleteket használni is tudja, adatokra is szükség van, ezt a programban ax-el jelöljük, ahol x egy természetes szám. A plot függvénnyel ki is tudjuk rajzolni a hálót, és az is megfigyelhető, hogy egészen jól tudja használni az OR-t, az értékek megközelítőleg pontosak.

Az AND is, hasonlóan az OR-hoz, egészen jó megoldásokat ad vissza, így az ember azt gondolná, hogy az EXOR-al sem lehet semmi baj. Ez egy hatalmas tévedés, ugyanis itt a hibahatár már nem egy rendkívül kis szám lesz, a program egész nagyokat tud tévedni. Rengeteg gondolkodás után, végül rájöttek a programozók, hogyan is hidalható át ez a probléma, rejtett- vagy hidden neuronokkal. Ezt a megoldást használva, a hálónak már az EXOR sem jelent igazi nehézséget, az eredmények ismét nagyon pontosak lesznek.

4.6. Hiba-visszaterjesztéses perceptron

C++

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

Megoldás forrása: <https://github.com/nbatfai/nahshon/blob/master/ql.hpp#L64>

Perceptronokról informatikában leginkább a mesterséges intelligencia, azon belül a neurális hálók témakörében találkozhatunk. A perceptronnak fontos szerepe van, leellenőrzi a bemenetet, majd egy feltétel alapján dönt, mi legyen a kimenet.

Tehát a perceptronok hasznosak, segítségükkel létrehozhatunk egy hibahatárt, amit a program be fog tartani. Feltételezzük, hogy 100 emberből legalább 80-nak magánhangzóval kezdődik a neve. Egyértelmű, hogy azt várjuk el, hogy legyen 80 olyan bemeneti adatunk, ahol a személy neve magánhangzóval kezdődik. Azonban ha a Perceptron megtalálja a 21. személyt akinek mássalhangzóval kezdődik a neve visszadob egy olyan értéket, ami jelzi nekünk, hogy a feltétel nem teljesült, például a (-1)-et. Ha ez történt meg, akkor a hibahatárt át kell állítanunk, finomhangolást kell végeznünk rajta.

Általában a finomhangolást nagyon magas határnál kezdi, majd addig csökkenti, amíg elfogadható a hibák mennyisége.

4.7. Minecraft MALMÖ - Mit lát Steve?

Forrás link: [Github link](#)

A feladat célja, hogy feldolgozzuk azt az információt, hogy mi van a karakterünk előtt, legyen az föld, levegő, láva, vagy virág, és ezt kiírjuk a képernyőre. Egy 3x3x3-as környezetben vizsgáljuk mindezt.

5. fejezet

Helló, Mandelbrot!

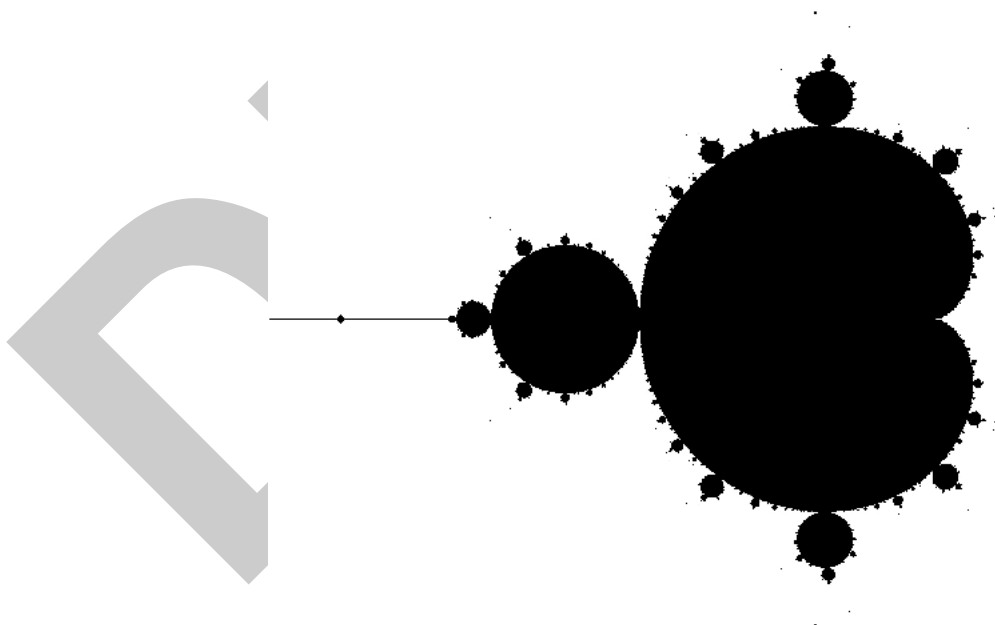
5.1. A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

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

Futtatás: <https://www.youtube.com/watch?v=dJKAj5ebrGg>

Megoldás forrása: https://github.com/Amsekal/Bhax/blob/master/bhax-master/attention_raising/CUDA/mandelp



5.1. ábra. A Mandelbrot halmaz a komplex síkon

A Mandelbrot halmaz egy olyan komplex számok halmaza, amelyet ha kivetítünk a síkra egy látványos és egyben érdekes alakzatot kapunk. Ezt a halmazt olyan számok alkotják, amelyeknek van valós és imaginárius részük is, így lehetséges az is, hogy gyök alatt negatív számunk legyen.

A kód segítségével megállapítjuk, hogy egy pont mikor eleme a halmaznak. Ha eleme, a pontot feketére színezzük.

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

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

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

Forrás: https://github.com/Amsekal/Bhax/blob/master/bhax-master/attention_raising/Mandelbrot/3.1.2.cpp

Hasonlóan az előző feladathoz, itt is az a célunk, hogy a Mandelbrot halmazt kirajzoljuk, tehát a két végeredménynek azonosnak kell lennie. A különbség csak annyi, hogy ebben a programban használni fogunk egy osztályt, ami segít a komplex számok kezelésében.

```
// Verzio: 3.1.2.cpp
// Forditas:
// g++ 3.1.2.cpp -lpng -O3 -o 3.1.2
// Futtatas:
// ./3.1.2 mandel.png 1920 1080 2040 ↵
-0.01947381057309366392260585598705802112818 ↵
-0.0194738105725413418456426484226540196687 ↵
0.7985057569338268601555341774655971676111 ↵
0.798505756934379196110285192844457924366
// ./3.1.2 mandel.png 1920 1080 1020 ↵
0.4127655418209589255340574709407519549131 ↵
0.4127655418245818053080142817634623497725 ↵
0.2135387051768746491386963270997512154281 ↵
0.2135387051804975289126531379224616102874
// Nyomtatas:
// a2ps 3.1.2.cpp -o 3.1.2.cpp.pdf -1 --line-numbers=1 --left-footer=" ↵
BATF41 HAXOR STR34M" --right-footer="https://bhaxor.blog.hu/" --pro= ↵
color
// ps2pdf 3.1.2.cpp.pdf 3.1.2.cpp.pdf.pdf
//
//
// Copyright (C) 2019
// Norbert Batfai, batfai.norbert@inf.unideb.hu
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <https://www.gnu.org/licenses/>.
```

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

Látható, hogy továbbra is szükség van a png++ -ra, ennek köszönhetően vagyunk képesek kirajzolni a halmazt a sikra ebben és az előző programban is.

```
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";

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

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

        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;
}
```

A különbség tehát a complex osztály használatában van. Ez egyszerűbbé is teszi a kódot, ugyanis nem kell azzal bajlódni a programozónak, hogy külön változót használjon egész és imaginárius résznek. Az osztály segítségével elég egy változót használni, és egy utána irt zárójelbe megadni a két részt.

Fontosnak tartom még megjegyezni, hogy a programot ugyanúgy kell futtatni, mint az előző esetben.

5.3. Biomorfok

Megoldás videó: <https://youtu.be/IJMbgRzY76E>

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

A Mandelbrot és Júlia halmazok igen hasonlóak, a legszembetűnőbb különbség az, hogy míg a c változó valós és imaginárius része változott a program továbbhaladtával a Mandelbrot halmaz esetén, addig a Júlia halmaznál ezeket az értékeket rögzítjük. Ezt a kódokban a két for ciklusban figyelhetjük meg.

Most következi egy igazán fontos kérdés: Ez nekünk miért jó? Erre a válasz egészen egyszerű: amíg a Mandelbrot halmaz ugyanúgy néz ki, addig Júlia halmazból jóval több van, és ezzel a megoldással ki is tudjuk őket rajzolni. Nincs más dolgunk csak a C változó értékét módosítsuk. Azonban nem értő kezek között nagy annak a valószínűsége, hogy a kapott halmaz nem lesz látványos, szerencsére az internet bőven szolgál számértékekkel, amivel csodás látványt érhetünk el.

Ha célunk a biomorfok pontos megértése akkor a következő cikk a segítségünkre lehet. https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9_Iss5_2305--2315_Biomorphs_via_modified_iterations.pdf. A kód megírása nehézséget is jelenthet, szóval nyugodtan használjuk fel az előző feladatokban írt kódot, és a fentebb említetteket írjuk át.

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó: <https://www.youtube.com/watch?v=ihZBrMlpVbM>

Megoldás forrása: https://github.com/Amsekal/Bhax/blob/master/bhax-master/attention_raising/CUDA/mandelpnge_60x60_100.cu

Ez a feladat egy kicsivel különlegesebb, mint az előzőek, ugyanis CUDA magokat használó kártyára van szükségünk a futtatásához. Koncepcióban a program megegyezik az előző Mandelbrot feladatokkal, a különbség abban feezhető fel, hogy itt a CUDA magok segítségével futtatjuk le a programot.

Mint észrevehető a videókban, a különbség nem triviális, sokkal gyorsabban képes elkészíteni a képet a program, ha CUDA magokat használ. Minél jobb a kártya, annál gyorsabb képgenerálás figyelhető meg.

Mivel a programot a videokártyának írtuk, ezért megfigyelhető, hogy a megszokott gcc vagy g++ helyett most nvcc-t használunk. Hogy használni tudjuk ezt a parancsot szükségünk lesz az nvidia-cuda-toolkitre.

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: Folyamatban Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_a_mandelbrot_halmazszal.

Megoldás forrása: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/binom/Batfai-Barki/frak/>

Visszatérünk a Mandelbrot halmazokhoz, de most nem csak egy képet szeretnénk végeredményként kapni, hanem egy olyan ablakot, ahol képesek vagyunk ránagyítani a halmazra. Ez lehetséges png file esetén is, de ott egy idő után a kép pixeles lesz, és nem lehet kivenni belőle semmit.

A megoldás amit használni fogunk ennél egy kicsit bonyolultabb, emiatt talán ez a fejezet legösszetettebb feladata. Az ötlet az, hogy az egérrel kijelölt részre az ablak nem ránagyít, hanem újragenerálja azt a részt. Így kiküszöbölhető az a probléma, hogy nagyon pixelesse válik a kép, ugyanis a pixelszám azonos lesz.

Az első probléma még a kódolás előtt megjelenik, hogyan készítsük el ezt az ablakot. Egy nagyszerű megoldás a Qt eszköztár használata, amelyben elkészíthető egy gui(grafikus interfész). Ennek az interfésznek a futtatásához 5 kódrészre van szükségünk, a következőekben ezek szerepeit mutatom be:

frakablak.h

Ebben a kódrészben egy osztályt hozunk létre, aminek van publikus, védett és privát tagja is. A publikus részben a tartományokat/határokat szabjuk meg, amik között mozoghatunk. A védett részben egy függvényt deklarálunk, amely figyelemmel követi, mit csinál a felhasználó. Ennek fontos szerepe lesz, hogy a program tudja, mikor kell nagyítania. Végül a privát tagban a nagyítandó terület van meghatározva.

frakszal.h

Ismét egy osztályt láthatunk és még pár változót. Ezek segítenek majd a programnak a számolásban és rajzolásban

frakszal.cpp

Ebben a kódrészben történik a legfontosabb dolog: a Mandelbrot halmaz elemeinek kiszámolása, ehhez hasonló kódot írtunk az első feladatokban is.

main.cpp

Itt kerül meghívásra qt konstruktora, és itt tesszük a kódhoz a Qt könyvtárat.

frak.pro

A frak.pro lesz az úgynevezett király, alatta fognak futni az előbb bemutatott kódok, az ő feladata lesz a teljes folyamat kezelése.

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

Megoldás forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#id570518>

A feladat itt is ugyan az, mint amit az előbb láthattunk, annyi különbséggel, hogy itt JAVA nyelvben kell megírunk a programot. A kód szerkezete rendkívül hasonló az előző feladatban megírtakéhoz, itt is az egérrel szeretnénk kiválasztani egy részt a nagy egészről, amire ráközelítenénk/ahová utaznánk.

Azonban ez a program többre is képes, mint amit az előbb létrehoztunk. Nem csak kirajzolja azt amire ránagyítunk, de képes pillanatképet is készíteni arról, ezt a kódrészt a public void pillanatfelvétel() alatt találhatjuk meg. A programrész nem olyan bonyolult, lényegében kiment a jelenleg aktív ablakot, és elnevezi. Ezt az elnevezést segíti egy számláló is, így ha több képet akarunk csinálni, az egyik nem fog rámentődni a másikra a néveggyezés miatt.

5.7. Minecraft MALMÖ -Lávaig

Forrás link: [Github link](#)

A feladat célja, hogy Steve felfele haladjon az arénában addig, amíg nem lát látót. Ha ez megtörténik akkor pedig fusson vissza. Ez egyszerűen megoldható egy while ciklussal, amit akkor szakítunk meg, amikor a látókörünkbe látó kerül.

DRAFT

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 eltérve kiszámolt szám.

Megoldás videó C++:<https://www.youtube.com/watch?v=-0uMOANtCPA>

Megoldás videó Java:<https://www.youtube.com/watch?v=TGpmeDn7KBY>

Megoldás forrása: https://github.com/Amsekal/Bhax/tree/master/bhax_textbook_IgyNeveldaProgramozod/Welch

Illetve <https://sourceforge.net/p/udprog/code/ci/master/tree/source/kezdo/elsojava/PolarGen.java#l10>

A kód maga igen egyszerű. Először is, szükség lesz egy osztályra:

```
class PolarGen
{
public:
    PolarGen ()
    {
        nincsTarolt = true;
        std::srand (std::time (NULL));
    }
    ~PolarGen ()
    {
    }
    double kovetkezo ();

private:
    bool nincsTarolt;
    double tarolt;
};
```

Az osztályt PolarGennek nevezzük el. Ebben az osztályban 2 privát tag lesz, a boolean típusu változó, a nincsTarolt, ami azt fogja jelezni számunkra, hogy van e eltárolt érték, és egy double típusu tarolt nevű változó. A konstruktorban megváltoztatjuk a nincsTarolt értékét igazra, majd létrehozunk egy random számot a srand() függvény segítségével. A destruktort üresen hadjuk. Ezen kívül a következő alprogram fejlécét találjuk az osztály definíciójában.

```
double
PolarGen::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;
    }
}
```

Hogy a program működjön is, szükség van a következő alprogramra. Itt, ha a nincsTarolt értéke igaz, akkor egy matematikai képletet fog alkalmazni, hogy végbemenjen a polártranszformáció. A tarolt kap egy értéket a képlet alapján, majd a boolean változót hamisra állítjuk. Ha a nincsTarolt értéke hamis, igazra állítjuk, és visszaadjuk a tarolt értékét.

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

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

```
return 0;  
}  
}
```

Ezek után már csak egy dolgunk maradt, megírni a főfüggvényt. Deklarálunk egy változót ami a PolarGen osztályhoz tartozik, a pg-t, majd egy for ciklusban többször is meghívjuk a következő függvényt, ennek következtében több eredményt is kapunk. A for ciklus után véget ér a program.

A java kód is pontosan ugyanezen alapok mellett íródott, egyetlen különbség, hogy ott csak egyszer lesz meghívva az alprogram, így csak egy eredményt kapunk.

Érdekesség, hogy létezik a Sun programozói által készített Random.java file, ami nagyban hasonlít az általunk elkészített megoldáshoz.

6.2. LZW

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

Megoldás videó:

Megoldás forrása: 6.6-os feladatban

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása: 6.6-os feladatban

6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása: 6.6-os feladatban

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: 6.6-os feladatban

6.6. Mozgató szemantika

Írj az előző programhoz mozgató konstruktort és értékadást, a mozgató konstruktor legyen a mozgató értékadásra alapozva!

Megoldás videó: https://www.youtube.com/watch?v=a_K2EWM53tg

Megoldás forrása: https://gitlab.com/nbatfai/bhax/-/blob/master/distance_learning/ziv_lempel_welch/z3a18qa_fr

Ahogy látható, az előző feladatokban el kellett készítenünk egy binfát, amit folyamatosan fejlesztünk. Az egyszerűség kedvéért, én az összes alpontot ebben a feladatban fogom bemutatni.

A Bináris fa egy különleges szerkezet, mivel minden gyökér maximum 2 utóddal rendelkezhet, egy jobboldalival és egy baloldalival. Előfordulhat olyan eset is, amikor csak az egyik oldalon rendelkezik utóddal, vagy nem rendelkezik egyel sem. A mi binfánk esetében a csomópontokban tárol értékek egyesek és nullák lesznek.

Amikor megkapjuk azt az értéket, legyen most ez 1, amit be szeretnénk tenni a fába, a következő folyamat fog lezajlani: megnézzük, hogy az aktuális csomópontnak/ helyi gyökérnek (legelső gyökér egy '/'-t tárol) van-e 1-es utódja. Ha van, a dolgunk egyszerű, az 1-es lesz a következő helyi gyökér. Ha nincs, akkor létrehozunk egy új csomópontot, és az ő utódja lesz az 1. Azt, hogy melyik elemnél tartunk egy pointer mutatja, mikor új csomópontot hozunk létre, ez a pointer visszatér az eredeti gyökérre, a /-re. Ugyanez az algoritmus akkor is, ha az érték 0.

A feladat teljes megétetéséhez pár fogalom tisztázásra szorul:

Rekurzió: Rekurzió a programozásban nem jelent mást, minthogy egy alprogram úgy végzi el a feladatot, hogy segítségül saját magát hívja meg. Fontos megjegyezni, hogy a rekurzió egy úgynevezett verem memóriát használ, ami képes megtelni, esetenként túlcsordulni. Rekurzívan lehet megoldani különböző kereséseket, de akár a backtracking megírásában is segíthet.

Konstruktor: Egy osztály metódusa, úgynevezett tagfüggvény, amely akkor hívódik meg, ha létrejön az objektum. Fontos, hogy a neve azonos kell legyen az osztályéval, nem hívható meg közvetlenül, és nem tartalmazhat visszatérési értéket.

Destruktor: A konstruktorhoz hasonlóan ez is egy tagfüggvény, akkor hívódik meg, ha az objektum megszűnik. A neve azonos az osztályéval, de előtte egy ~-nek kell szerepelnie. Használati szabályai azonosak a konstruktorral.

Hogy ezt meg tudjuk valósítani, szükségün lesz egy BinFa osztályra, amit jelen esetben BinTree-nek fogunk hívni. Ebben az osztályban használni fogunk még egy osztályt, ami a csomópontok használatában fog segíteni, ezt Node-nak nevezzük el.

```
#include <iostream>

template <typename ValueType>
class BinTree {

protected:
    class Node {

    private:
        ValueType value;
        Node *left;
```



```

    Node *right;
    int count{0};

    // TODO rule of five
    Node(const Node &);
    Node & operator=(const Node &);
    Node(Node &&);
    Node & operator=(Node &&);

public:
    Node(ValueType value): value(value), left(nullptr), right(nullptr) ↔
    {}
    ValueType getValue(){return value;}
    Node * leftChild(){return left;}
    Node * rightChild(){return right;}
    void leftChild(Node * node){left = node;}
    void rightChild(Node * node){right = node;}
    int getCount(){return count;}
    void incCount(){++count;}
};
}

```

A Node osztályban a value értékben lesz eltárolva az érték amivel dolgozni szeretnénk, utána pedig 2 mutatót is deklarálunk, ami a két lehetséges utódra mutat, végül pedig egy integer változót, amivel számolni fogunk. A következő sorokban tiltjuk a másoló/mozgató konstruktor és másoló értékadást.

Publicban található meg a konstruktor, ahol értéket adunk az előbb bevezetett változóknak, a value változó megkapja az értéket, a pointererek, mivel kezdetben nincs utódjuk, nem mutatnak semmire. Ezek után létrehozunk alprogramokat, az első 3 egyszerű érték visszaadás, az utánnuk következő kettővel a pointert tudjuk majd állítani, a getCount visszaadja a count értékét, az incCount pedig a count értékét növeli. Most, hogy készen vagyunk a Node osztállyal, következhet a BinTree osztály befejezése.

```

Node *root;
Node *treep;
int depth{0};

private:
    // TODO rule of five

public:
    BinTree(Node *root = nullptr, Node *treep = nullptr): root(root), treep ↔
    (treep) {
        std::cout << "BT konstruktor" << std::endl;
    }

    BinTree(const BinTree & old) {
        std::cout << "BT masolo konstruktor" << std::endl;

        root = cp(old.root, old.treep);
    }

```

```
}

Node * cp(Node *node, Node *treep)
{
    Node * newNode = nullptr;

    if(node)
    {
        newNode = new Node(node->getValue());

        newNode->leftChild(cp(node->leftChild(), treep));
        newNode->rightChild(cp(node->rightChild(), treep));

        if(node == treep)
            this->treep = newNode;
    }

    return newNode;
}

BinTree & operator=(const BinTree & old) {
    std::cout << "BT masolo ertekadas" << std::endl;

    BinTree tmp{old};
    std::swap(*this, tmp);
    return *this;
}

BinTree(BinTree && old) {
    std::cout << "BT mozgato konstruktor" << std::endl;

    root = nullptr;
    *this = std::move(old);
}

BinTree & operator=(BinTree && old) {
    std::cout << "BT mozgato ertekadas" << std::endl;

    std::swap(old.root, root);
    std::swap(old.treep, treep);

    return *this;
}

~BinTree(){
    std::cout << "BT destruktor" << std::endl;
    deltree(root);
}
```

```
    BinTree & operator<<(ValueType value);
    void print(){print(root, std::cout);}
    void print(Node *node, std::ostream & os);
    void deltree(Node *node);

};

template <typename ValueType, ValueType vr, ValueType v0>
class ZLWTree : public BinTree<ValueType> {

public:
    ZLWTree(): BinTree<ValueType>(new typename BinTree<ValueType>::Node(vr) ←
        ) {
        this->treep = this->root;
    }
    ZLWTree & operator<<(ValueType value);

};
}
```

Kezdeként készítünk még 2 pointert, egyet a gyökérnek, egyet a fának, és egy integer változóra is szükség lesz, hogy a mélységet is nyomon tudjuk követni. Itt is szükség lesz egy konstruktorra, és hogy nyomon tudjuk követni a dolgokat, egy kiíratás is szerepet kap. Létrehozunk még egy másoló- és mozgató értékadás meg mozgatás konstruktort is. A mozgatást egy swap függvénnyel oldjuk meg, itt meg kell adni a két csomópontot amit cserélnénk. Fontos észrevenni, hogy nem csak a csomópontok cserélődnek, hanem a mutatók is.

A másolás egy kicsit bonyolultabb, itt használjuk majd a `Node *` cp-t is. Ezzel a kóddal rekurzió segítségével tudjuk létrehozni a fa másolatát. Végezetül a shiftelést definiáljuk, ennek következtében tudunk a main függvényben értéket adni majd a programnak, ami alapján felépül a fa. Azonban ezt egy template-ben csináljuk meg.

```
template <typename ValueType>
BinTree<ValueType> & BinTree<ValueType>::operator<<(ValueType value)
{
    if(!treep) {

        root = treep = new Node(value);

    } else if (treep->getValue() == value) {

        treep->incCount();

    } else if (treep->getValue() > value) {

        if(!treep->leftChild()) {

            treep->leftChild(new Node(value));

        }

    }

}
```

```
    } else {

        treep = treep->leftChild();
        *this << value;
    }

} else if (treep->getValue() < value) {

    if(!treep->rightChild()) {

        treep->rightChild(new Node(value));

    } else {

        treep = treep->rightChild();
        *this << value;
    }

}

treep = root;

return *this;
}
}
```

A fenti kódban látható a fa felépítése, az alapján amit a fentiekben leírtam. A programban található még egy template ami a fa mélységét számolja ki és kiírja (bejárja) a fát, de mivel ez csak számolás és kiírás, erre nem térnék ki külön.

```
template <typename ValueType>
void BinTree<ValueType>::deltree(Node *node)
{
    if (node)
    {
        deltree(node->leftChild());
        deltree(node->rightChild());

        delete node;
    }
}
}
```

Ez a template felelős a fa törléséért, először az utódokat törli, majd végül a helyi gyökeret.

```
int main(int argc, char** argv, char ** env)
{
    bts << "alma" << "korte" << "banan" << "korte";

    bts.print();
}
```

```
}  
}
```

A main függvényben már semmi dolgunk nincs, csak felhasználni az eddig készített kódokat. A `bts`-be shiftelünk szavakat, de mivel a shifteléshez irtunk egy programot, így a szavakból fát fog készíteni, amit a `print` utasítással irtunk ki.

6.7. Minecraft MALMÖ -5x5x5

Forrás link: [Github link](#)

Hasonlón egy előző feladathoz, itt is Steve látásával fogunk foglalkozni. A feladat egyszerű része megnövelni a látótávolságot, itt csak a számokat kell átírni. A nehézségek akkor kezdődnek, amikor ezeket az információkat kell feldolgozni, de a MALMÖ oldalán találhatunk segítséget.

7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk

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

Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist> <https://www.youtube.com/watch?v=H2wnsA>
Ns - Működés közben

Megoldás forrása: [Link](#)

Ebben a feladatban egy hangyaszimulációt fogunk készíteni, de még mielőtt nekiesnénk a feladatnak, jó volna tisztázni, mit is szeretnénk szimulálni.

Mindenki látott már életében hangyákat, és akár meg is figyelhette, hogy gyakran egymást követik. A hangyák egy különleges feromont bocsátanak ki a testükből, ezzel jelezve, hogy merre jártak, hol találtak valami érdekeset/fontosat a kolónia számára. Minél több hangya követi ezt a feromon "utat", annál erősebb lesz a vonzó hatása a hangyák számára, ugyanis mindenki aki követi a feromont, hátrahadja a sajátját is.

Ezt leginkább úgy lehet elképzelni, mint amikor túrázás közben a mások által hátrahagyott táblákat követjük, hogy eljussunk a kijelölt helyre.

A programot több kis részből építjük fel, ezeket fogom részletesen bemutatni és elmagyarázni.

antthread.h

```
#ifndef ANTTHREAD_H
#define ANTTHREAD_H

#include <QThread>
#include "ant.h"

class AntThread : public QThread
{
    Q_OBJECT

public:
    AntThread(Ants * ants, int ***grids, int width, int height,
              int delay, int numAnts, int pheromone, int nbrPheromone,
```

```
        int evaporation, int min, int max, int cellAntMax);

~AntThread();

void run();
void finish()
{
    running = false;
}

void pause()
{
    paused = !paused;
}

bool isRunnung()
{
    return running;
}

private:
    bool running {true};
    bool paused {false};
    Ants* ants;
    int** numAntsinCells;
    int min, max;
    int cellAntMax;
    int pheromone;
    int evaporation;
    int nbrPheromone;
    int ***grids;
    int width;
    int height;
    int gridIdx;
    int delay;

    void timeDevel();

    int newDir(int sor, int oszlop, int vsor, int voszlop);
    void detDirs(int irány, int& ifrom, int& ito, int& jfrom, int& jto );
    int moveAnts(int **grid, int row, int col, int& retrow, int& retcol, ←
        int);
    double sumNbhs(int **grid, int row, int col, int);
    void setPheromone(int **grid, int row, int col);

signals:
    void step ( const int &);

};
```

```
#endif  
  
}
```

A fent látható kódban hozzuk létre az ablakot, ahol nyomon tudjuk majd követni a szimulációt. Működése hasonló lesz az előző qt-s programhoz (Mandelbrot fejezet), annyi különbséggel, hogy itt megjelenik még pár függvény deklarációja, ezeknek a hangyák és a feromon viselkedésében lesz szerepük a későbbiekben.

ant.h

```
#ifndef ANT_H  
#define ANT_H  
  
class Ant  
{  
  
public:  
    int x;  
    int y;  
    int dir;  
  
    Ant(int x, int y): x(x), y(y) {  
  
        dir = grand() % 8;  
  
    }  
  
};  
  
typedef std::vector<Ant> Ants;  
  
#endif  
  
}
```

A következő kódrészünk az ant.h. Láthatóan egyszerűbb és rövidebb mint a kódok, amiket általában használunk, de ennek köszönhetően kevesebbet is érünk el vele. A kód létrehozza a hangya (ant) osztályt, aminek köszönhetően nyomon tudjuk majd követni a hangya koordinátáit (x és y koordináta), és az irányát/direkcióját. Mint látható, egy konstruktort is használni fogunk (konstruktor definíciója az előző fejezetben), ennek segítségével határozzuk meg az irányt. A grand() függvény egy véletlenszerű értéket fog adni a dir változónak.

Végül a typedef utasítás segítségével létrehozunk egy vektort, ami Ant típusu, és elnevezzük Ants-nek. A névből is ki lehet találni, hogy ebben a vektorban lesznek eltárolva a hangyák, akik a képrenyőnkön fognak tevékenykedni.

antwin.h

```
#ifndef ANTWIN_H  
#define ANTWIN_H  
  
#include <QMainWindow>
```



```
#include <QPainter>
#include <QString>
#include <QCloseEvent>
#include "antthread.h"
#include "ant.h"

class AntWin : public QMainWindow
{
    Q_OBJECT

public:
    AntWin(int width = 100, int height = 75,
           int delay = 120, int numAnts = 100,
           int pheromone = 10, int nbhPheromon = 3,
           int evaporation = 2, int cellDef = 1,
           int min = 2, int max = 50,
           int cellAntMax = 4, QWidget *parent = 0);

    AntThread* antThread;

    void closeEvent ( QCloseEvent *event ) {

        antThread->finish();
        antThread->wait();
        event->accept();
    }

    void keyPressEvent ( QKeyEvent *event )
    {

        if ( event->key() == Qt::Key_P ) {
            antThread->pause();
        } else if ( event->key() == Qt::Key_Q
                    || event->key() == Qt::Key_Escape ) {
            close();
        }

    }

    virtual ~AntWin();
    void paintEvent(QPaintEvent*);

private:

    int ***grids;
    int **grid;
    int gridIdx;
    int cellWidth;
    int cellHeight;
    int width;
```

```
int height;
int max;
int min;
Ants* ants;

public slots :
    void step ( const int &);

};

#endif

}
```

Ebben a header fileban fogunk a billentyűzetek lenyomásával foglalkozni, ezen kívül megtalálhatóak az előbbieken bemutatott kódok is. Itt fogjuk megoldani a program kezelését, mi történjen ha elindul vagy ka kikapcsol a program. Ezen kívül programozunk még egy eventet arra az esetre, ha szeretnénk ideiglenesen megállítani a hangyákat.

Ezek voltak a .h kiterjesztésű fileok, most következzenek a .cpp kiterjesztésűek.

antthread.cpp

```
#include "antthread.h"
#include <QDebug>
#include <cmath>
#include <QDateTime>

AntThread::AntThread ( Ants* ants, int*** grids,
                      int width, int height,
                      int delay, int numAnts,
                      int pheromone, int nbrPheromone,
                      int evaporation,
                      int min, int max, int cellAntMax)
{
    this->ants = ants;
    this->grids = grids;
    this->width = width;
    this->height = height;
    this->delay = delay;
    this->pheromone = pheromone;
    this->evaporation = evaporation;
    this->min = min;
    this->max = max;
    this->cellAntMax = cellAntMax;
    this->nbrPheromone = nbrPheromone;

    numAntsinCells = new int*[height];
    for ( int i=0; i<height; ++i ) {
        numAntsinCells[i] = new int [width];
    }
}
```

```
for ( int i=0; i<height; ++i )
    for ( int j=0; j<width; ++j ) {
        numAntsinCells[i][j] = 0;
    }

qsrand ( QDateTime::currentMSecsSinceEpoch() );

Ant h {0, 0};
for ( int i {0}; i<numAnts; ++i ) {

    h.y = height/2 + qrand() % 40-20;
    h.x = width/2 + qrand() % 40-20;

    ++numAntsinCells[h.y][h.x];

    ants->push_back ( h );

}

gridIdx = 0;
}

double AntThread::sumNbhs ( int **grid, int row, int col, int dir )
{
    double sum = 0.0;

    int ifrom, ito;
    int jfrom, jto;

    detDirs ( dir, ifrom, ito, jfrom, jto );

    for ( int i=ifrom; i<ito; ++i )
        for ( int j=jfrom; j<jto; ++j )

            if ( ! ( ( i==0 ) && ( j==0 ) ) ) {
                int o = col + j;
                if ( o < 0 ) {
                    o = width-1;
                } else if ( o >= width ) {
                    o = 0;
                }

                int s = row + i;
                if ( s < 0 ) {
                    s = height-1;
                } else if ( s >= height ) {
                    s = 0;
                }
            }
        }
```

```
        sum += (grid[s][o]+1)*(grid[s][o]+1)*(grid[s][o]+1);

    }

    return sum;
}

int AntThread::newDir ( int sor, int oszlop, int vsor, int voszlop )
{

    if ( vsor == 0 && sor == height -1 ) {
        if ( voszlop < oszlop ) {
            return 5;
        } else if ( voszlop > oszlop ) {
            return 3;
        } else {
            return 4;
        }
    } else if ( vsor == height - 1 && sor == 0 ) {
        if ( voszlop < oszlop ) {
            return 7;
        } else if ( voszlop > oszlop ) {
            return 1;
        } else {
            return 0;
        }
    } else if ( voszlop == 0 && oszlop == width - 1 ) {
        if ( vsor < sor ) {
            return 1;
        } else if ( vsor > sor ) {
            return 3;
        } else {
            return 2;
        }
    } else if ( voszlop == width && oszlop == 0 ) {
        if ( vsor < sor ) {
            return 7;
        } else if ( vsor > sor ) {
            return 5;
        } else {
            return 6;
        }
    } else if ( vsor < sor && voszlop < oszlop ) {
        return 7;
    } else if ( vsor < sor && voszlop == oszlop ) {
        return 0;
    } else if ( vsor < sor && voszlop > oszlop ) {
        return 1;
    }
}
```

```
    else if ( vsor > sor && voszlop < oszlop ) {
        return 5;
    } else if ( vsor > sor && voszlop == oszlop ) {
        return 4;
    } else if ( vsor > sor && voszlop > oszlop ) {
        return 3;
    }

    else if ( vsor == sor && voszlop < oszlop ) {
        return 6;
    } else if ( vsor == sor && voszlop > oszlop ) {
        return 2;
    }

    else { //(vsor == sor && voszlop == oszlop)
        qDebug() << "ZAVAR AZ EROBEN az iranynal";

        return -1;
    }
}

void AntThread::detDirs ( int dir, int& ifrom, int& ito, int& jfrom, int& ←
    jto )
{

    switch ( dir ) {
    case 0:
        ifrom = -1;
        ito = 0;
        jfrom = -1;
        jto = 2;
        break;
    case 1:
        ifrom = -1;
        ito = 1;
        jfrom = 0;
        jto = 2;
        break;
    case 2:
        ifrom = -1;
        ito = 2;
        jfrom = 1;
        jto = 2;
        break;
    case 3:
        ifrom =
            0;
        ito = 2;
        jfrom = 0;
```

```
        jto = 2;
        break;
    case 4:
        ifrom = 1;
        ito = 2;
        jfrom = -1;
        jto = 2;
        break;
    case 5:
        ifrom = 0;
        ito = 2;
        jfrom = -1;
        jto = 1;
        break;
    case 6:
        ifrom = -1;
        ito = 2;
        jfrom = -1;
        jto = 0;
        break;
    case 7:
        ifrom = -1;
        ito = 1;
        jfrom = -1;
        jto = 1;
        break;
    }
}

int AntThread::moveAnts ( int **racs,
                          int sor, int oszlop,
                          int& vsor, int& voszlop, int dir )
{
    int y = sor;
    int x = oszlop;

    int ifrom, ito;
    int jfrom, jto;

    detDirs ( dir, ifrom, ito, jfrom, jto );

    double osszes = sumNbhs ( racs, sor, oszlop, dir );
    double random = ( double ) ( grand() %1000000 ) / ( double ) 1000000.0;
    double gvalseg = 0.0;

    for ( int i=ifrom; i<ito; ++i )
```

```
for ( int j=jfrom; j<jto; ++j )
    if ( ! ( ( i==0 ) && ( j==0 ) ) )
    {
        int o = oszlop + j;
        if ( o < 0 ) {
            o = width-1;
        } else if ( o >= width ) {
            o = 0;
        }

        int s = sor + i;
        if ( s < 0 ) {
            s = height-1;
        } else if ( s >= height ) {
            s = 0;
        }

        //double kedvezo = std::sqrt((double) (racs[s][o]+2)); //( ←
        racs[s][o]+2)*(racs[s][o]+2);
        //double kedvezo = (racs[s][o]+b)*(racs[s][o]+b);
        //double kedvezo = ( racs[s][o]+1 );
        double kedvezo = (racs[s][o]+1)*(racs[s][o]+1)*(racs[s][o] ←
            ]+1);

        double valseg = kedvezo/osszes;
        gvalseg += valseg;

        if ( gvalseg >= random ) {

            vsor = s;
            voszlop = o;

            return newDir ( sor, oszlop, vsor, voszlop );

        }

    }

    qDebug() << "ZAVAR AZ EROBEN a lepesnel";
    vsor = y;
    voszlop = x;

    return dir;
}
}
```

Ebben a részben rengeteg kód van, de legnagyobb részt csak egyszerű adminisztráció található itt. Először meghívjuk a már elkészített .h-kat, amiket fel fogunk használni. Ezek után találhatóak alprogramok, ezek szerepe, hogy az irányt helyes értékre állítsák, hogy a szimuláció ténylegesen egy szimuláció legyen, és ne csak kis négyzetek legyenek, amik össze-vissza mozognak. Ezután látható a moveAnts függvény ami

már egy kicsivel komplexebb. Itt fogjuk felhasználni az irányváltáshoz írt alprogramokat, és itt oldjuk meg, hogy a hangyák a négyzethálóban haladjanak. Minden alprogramhoz van egy debug is, ami leellenőrzi, hogy minden fut e, és ha nem egy üzenetet ír ki, ami jelen esetben egy "Csillagok háborúja" utalás.

```
void AntThread::timeDevel()
{

    int **racsElotte = grids[gridIdx];
    int **racsUtana = grids[ ( gridIdx+1 ) %2];

    for ( int i=0; i<height; ++i )
        for ( int j=0; j<width; ++j )
        {
            racsUtana[i][j] = racsElotte[i][j];

            if ( racsUtana[i][j] - evaporation >= 0 ) {
                racsUtana[i][j] -= evaporation;
            } else {
                racsUtana[i][j] = 0;
            }

        }

    for ( Ant &h: *ants )
    {

        int sor {-1}, oszlop {-1};
        int ujirany = moveAnts( racsElotte, h.y, h.x, sor, oszlop, h.dir );

        setPheromone ( racsUtana, h.y, h.x );

        if ( numAntsinCells[sor][oszlop] <cellAntMax ) {

            --numAntsinCells[h.y][h.x];
            ++numAntsinCells[sor][oszlop];

            h.x = oszlop;
            h.y = sor;
            h.dir = ujirany;

        }

    }

    gridIdx = ( gridIdx+1 ) %2;
}
}
```

A fenti alprogrammal érjük el, hogy a hangyák képesek legyenek maguk után egy feromoncsikot húzni. Ez az egyik legfontosabb része a szimulációnak, ugyanis ha ez nem megy, lehetetlen lesz az egymást követő

hangyák képrenyőn való ábrázolása. A megoldás igen egyszerű, a négyzetrács, amin a hangya áthalad kapni fog egy feromon értéket, amit ha egy másik hangya "megérez", követni fogja.

```
void AntThread::setPheromone ( int **racs,
                               int sor, int oszlop )
{
    for ( int i=-1; i<2; ++i )
        for ( int j=-1; j<2; ++j )
            if ( ! ( ( i==0 ) && ( j==0 ) ) )
            {
                int o = oszlop + j;
                {
                    if ( o < 0 ) {
                        o = width-1;
                    } else if ( o >= width ) {
                        o = 0;
                    }
                }
                int s = sor + i;
                {
                    if ( s < 0 ) {
                        s = height-1;
                    } else if ( s >= height ) {
                        s = 0;
                    }
                }

                if ( racs[s][o] + nbrPheromone <= max ) {
                    racs[s][o] += nbrPheromone;
                } else {
                    racs[s][o] = max;
                }
            }

    if ( racs[sor][oszlop] + pheromone <= max ) {
        racs[sor][oszlop] += pheromone;
    } else {
        racs[sor][oszlop] = max;
    }
}

void AntThread::run()
{
    running = true;
    while ( running ) {
```

```
        QThread::msleep ( delay );

        if ( !paused ) {
            timeDevel();
        }

        emit step ( gridIdx );

    }

}

AntThread::~~AntThread()
{
    for ( int i=0; i<height; ++i ) {
        delete [] numAntsinCells[i];
    }

    delete [] numAntsinCells;
}

}
```

Végül megírjuk, hogy pontosan hogyan fogja a feromont hátrahagyni.

antwin.cpp és main.cpp

Az antwin.cpp-vel nem fogok részletekbe menően foglalkozni, ugyanis itt csak a megjelenítéssel foglalkozunk, mint például a feromon színe, intenzivitása, és itt frissítjük a lépéseket is.

A main.cpp a főfüggvény, itt csak meghívjuk az elkészített kódokat, és megadunk pár paramétert ami a futtatáshoz szükséges, mint például az ablak szélessége.

7.2. Java életjáték

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

Megoldás videó: https://www.youtube.com/watch?v=Oo_TCojFROo

Megoldás forrása: [Link](#)

John Horton Conway egy angol matematikus volt, aki a Neumann János által felvetett problémára próbált megoldást találni. Ez a megoldás lett, a következőekben olvasható program. Az életjáték valójában egy sejt-automata, ami a sejtek viselkedését szeretné bemutatni. Fontos megjegyezni, hogy ez egy nulla személyes játék, a "játékos" csak az alakzatokat tudja meghatározni, minen más a gép végez.

A játéknak szabályai vannak, ahol a cella sejteket jelképez:

1. Minden cella környezete a körülötte lévő 8 cella.

2. A cella akkor él, ha 2 vagy 3 élő cella van körülötte, különben halott.

3. Egy halott cella életre kelhet, ha 3 élő cella van a környezetében.

A feladat megvalósításához szükséges még említést tenni a siklóról. A sikló egy különleges alakzat, amely periodikusan változik, és átlósan halad felfelé.

```
/*
 * Sejtautomata.java
 *
 * DIGIT 2005, Javat tanítók
 * Bátfai Norbert, nbatfai@inf.unideb.hu
 *
 */
/**
 * Sejtautomata osztály.
 *
 * @author Bátfai Norbert, nbatfai@inf.unideb.hu
 * @version 0.0.1
 */
public class Sejtautomata extends java.awt.Frame implements Runnable {

    public static final boolean ÉLŐ = true;

    public static final boolean HALOTT = false;

    protected boolean [][][] rácsek = new boolean [2][][];

    protected boolean [][] rácsh;

    protected int rácshIndex = 0;

    protected int cellaSzélesség = 20;
    protected int cellaMagasság = 20;

    protected int szélesség = 20;
    protected int magasság = 10;

    protected int várakozás = 1000;

    private java.awt.Robot robot;

    private boolean pillanatfelvétel = false;

    private static int pillanatfelvételSzámológ = 0;}
```

A fent látható kódrészben, létrehozunk egy kiterjesztett osztályt, ezt az osztály fejlécében látható extends java.awt.Frame rész jelzi. Ebben az osztályban fogunk figyelni a sejtek életére ,adataira és a rácsek állapota, ezen kívül még a pillanatképekre. A rácshIndex az aktuális rácshra hivatkozik.

```
public Sejtautomata(int szélesség, int magasság) {
```

```
this.szélesség = szélesség;
this.magasság = magasság;

rácsok[0] = new boolean[magasság][szélesség];
rácsok[1] = new boolean[magasság][szélesség];
rácsIndex = 0;
rács = rácsok[rácsIndex];

for(int i=0; i<rács.length; ++i)
    for(int j=0; j<rács[0].length; ++j)
        rács[i][j] = HALOTT;

siklóKilövő(rács, 5, 60);

addWindowListener(new java.awt.event.WindowAdapter() {
    public void windowClosing(java.awt.event.WindowEvent e) {
        setVisible(false);
        System.exit(0);
    }
});

addKeyListener(new java.awt.event.KeyAdapter() {

    public void keyPressed(java.awt.event.KeyEvent e) {
        if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K) {

            cellaSzélesség /= 2;
            cellaMagasság /= 2;
            setSize(Sejtautomata.this.szélesség*cellaSzélesség,
                    Sejtautomata.this.magasság*cellaMagasság);
            validate();
        } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {

            cellaSzélesség *= 2;
            cellaMagasság *= 2;
            setSize(Sejtautomata.this.szélesség*cellaSzélesség,
                    Sejtautomata.this.magasság*cellaMagasság);
            validate();
        } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)
            pillanatfelvétel = !pillanatfelvétel;
        else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G)
            várakozás /= 2;
        else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L)
            várakozás *= 2;
        repaint();
    }
});

addMouseListener(new java.awt.event.MouseAdapter() {
```

```
        public void mousePressed(java.awt.event.MouseEvent m) {

            int x = m.getX()/cellaSzélesség;
            int y = m.getY()/cellaMagasság;
            rácsok[rácsIndex][y][x] = !rácsok[rácsIndex][y][x];
            repaint();
        }
    });

    addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {

        public void mouseDragged(java.awt.event.MouseEvent m) {
            int x = m.getX()/cellaSzélesség;
            int y = m.getY()/cellaMagasság;
            rácsok[rácsIndex][y][x] = ÉLŐ;
            repaint();
        }
    });

    cellaSzélesség = 10;
    cellaMagasság = 10;

    try {
        robot = new java.awt.Robot(
            java.awt.GraphicsEnvironment.
                getLocalGraphicsEnvironment().
                getDefaultScreenDevice());
    } catch (java.awt.AWTException e) {
        e.printStackTrace();
    }

    setTitle("Sejtautomata");
    setResizable(false);
    setSize(szélesség*cellaSzélesség,
            magasság*cellaMagasság);
    setVisible(true);

    new Thread(this).start();
}
}
```

Ebben a kódrészben a konstruktort találjuk. Itt készítjük el a rácsokból álló képet, és minden cellát alapértelmezetten halottra állítunk. Ezen felül láthatunk még eventeket is, amelyeket egy gomb lenyomása aktivál. Az N (Nagyít) és K (Kicsinyít) gombbal az ablak méretét tudjuk befolyásolni, a G (gyorsít) és L (Lassít) gombokkal pedig a sebességet irányíthatjuk. Az ablak nagyításakor az egész képet újrarajzoltatjuk. Az egeret is szeretnénk használni, így neki is írunk eventet: kattintáskor és a kurzor mozgásakor élő cellákat tudunk létrehozni, amelyek beleépülnek a játékba, bár ez nem javasolt gyors sebességen, ugyanis mire az alakzatunkat megrajzolnánk, az már rég megváltozott.

```
public void paint(java.awt.Graphics g) {
```

```
boolean [][] rács = rácsok[rácsIndex];

for(int i=0; i<rács.length; ++i) {
    for(int j=0; j<rács[0].length; ++j) {

        if(rács[i][j] == ÉLŐ)
            g.setColor(java.awt.Color.BLACK);
        else
            g.setColor(java.awt.Color.WHITE);
        g.fillRect(j*cellaSzélesség, i*cellaMagasság,
                    cellaSzélesség, cellaMagasság);

        g.setColor(java.awt.Color.LIGHT_GRAY);
        g.drawRect(j*cellaSzélesség, i*cellaMagasság,
                    cellaSzélesség, cellaMagasság);
    }
    if(pillanatfelvétel) {

        pillanatfelvétel = false;
        pillanatfelvétel(robot.createScreenCapture
                        (new java.awt.Rectangle
                        (getLocation().x, getLocation().y,
                         szélesség*cellaSzélesség,
                         magasság*cellaMagasság)));
    }
}
```

Itt a kirajzolás és a képernyőkép kimentése látható, ez hasonló a régebben tárgyalt programokéhoz, így különösebb magyarázatot nem igényel.

```
public int szomszédokSzáma(boolean [][] rács,
                           int sor, int oszlop, boolean állapot) {
    int állapotúSzomszéd = 0;

    for(int i=-1; i<2; ++i)
        for(int j=-1; j<2; ++j)

            if(!((i==0) && (j==0))) {

                int o = oszlop + j;
                if(o < 0)
                    o = szélesség-1;
                else if(o >= szélesség)
                    o = 0;

                int s = sor + i;
                if(s < 0)
                    s = magasság-1;
```

```
        else if(s >= magasság)
            s = 0;

        if(rács[s][o] == állapot)
            ++állapotúSzomszéd;
        }

        return állapotúSzomszéd;
    }

    public void időFejlődés() {

        boolean [][] rácsElőtte = rácsok[rácsIndex];
        boolean [][] rácsUtána = rácsok[(rácsIndex+1)%2];

        for(int i=0; i<rácsElőtte.length; ++i) {
            for(int j=0; j<rácsElőtte[0].length; ++j) {

                int élők = szomszédokSzáma(rácsElőtte, i, j, ÉLŐ);

                if(rácsElőtte[i][j] == ÉLŐ) {

                    if(élők==2 || élők==3)
                        rácsUtána[i][j] = ÉLŐ;
                    else
                        rácsUtána[i][j] = HALOTT;
                } else {

                    if(élők==3)
                        rácsUtána[i][j] = ÉLŐ;
                    else
                        rácsUtána[i][j] = HALOTT;
                }
            }
        }
        rácsIndex = (rácsIndex+1)%2;
    }

    public void run() {

        while(true) {
            try {
                Thread.sleep(várakozás);
            } catch (InterruptedException e) {}

            időFejlődés();
            repaint();
        }
    }
}
```

```
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 szomszédokszáma függvény, a nevéből is láthatóan egy sejt szomszédait fogja számolni, és ezt a számot elmenteni. Ez a szám dönti ugyanis el, hogy a sejt él e, vagy hal. Az időfejlődés fog végigmenni a cellákon, és az állapotát a helyzetéhez igazítja, a szabály szerint. A run függvényben egy try catch-et láthatunk, és 2 alprogram meghívását. Ez felelős a folyamatosságért.

```
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;

}

public void pillanatfelvétel(java.awt.image.BufferedImage felvetel) {

    StringBuffer sb = new StringBuffer();
    sb = sb.delete(0, sb.length());
    sb.append("sejtautomata");
    sb.append(++pillanatfelvételSzámláló);
    sb.append(".png");

    try {
        javax.imageio.ImageIO.write(felvetel, "png",
            new java.io.File(sb.toString()));
    } catch (java.io.IOException e) {
        e.printStackTrace();
    }
}

public void update(java.awt.Graphics g) {
    paint(g);
}

public static void main(String[] args) {

    new Sejtautomata(100, 75);
}
}
```

Végezetül láthatjuk a siklókilövő felépítését, ami a bal alsó sarokba fog kerülni. A pillanatfelvétel függvényben pedig a pillanatkép elmentése található, png formátumba.

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás forrása: [Link](#)

Ez a feladat az előzőekben definiált életjáték c++ kódját tartalmazza. A feladat menete, algoritmusai megegyeznek a javás társával, csak a programozási nyelv különbözik. Itt qt segítségével fogjuk futtatni a programot, itt is szükség lesz egy ablakra, amit négyzetrácsokra bontunk, itt is körbejárjuk a sejt környezetét, hogy megállapítsuk élő e vagy halott és itt is létrehozuk a kilövőt.

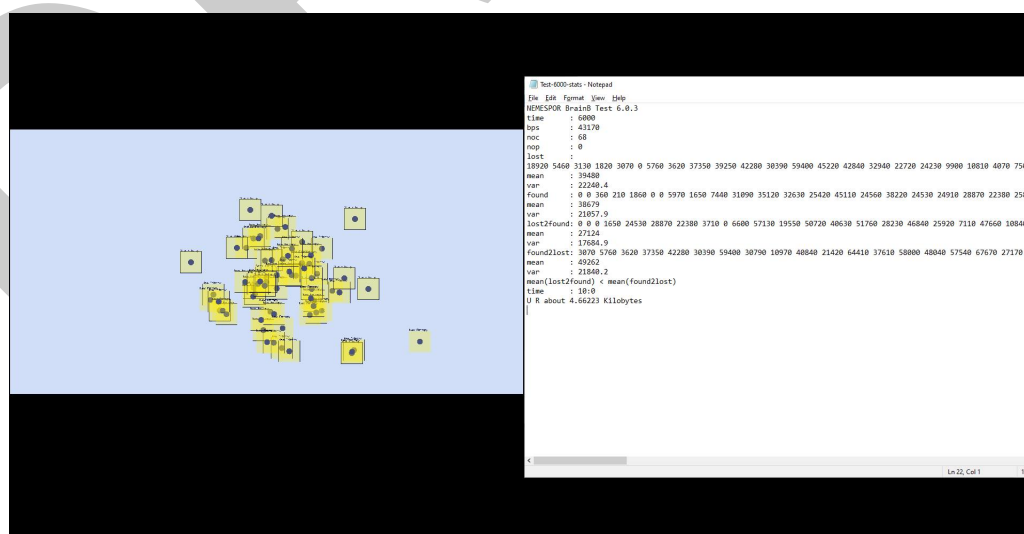
7.4. BrainB Benchmark

Megoldás videó: <https://www.youtube.com/watch?v=QntHsqSe3S4>

Megoldás forrása: [Link](#)

A BrainB-t úgy is fel lehet fogni, mint egy játékot, aminek az a célja, hogy a lehető legtöbb ideig tartsuk a kurzort a Samu entropy nevű fekete pöttyön, miközben a játék folyamatosan nehezedik, egyre több másnevű fekete pötty jelenik meg.

A valóságban ezért ennél többről van szó, ez a program a "játékos" kéz és szem koordinációját hivatott megfigyelni. Ezt a végén egy pontszámban ki is mutatja, egy teljes elemzés mellett. A program segít felmérni a kéz-szem koordinációt, így meg lehet állapítani, például azt is, hogy mennyire lenne az ember jó e-sportoló. Természetesen ez csak egy példa, és ha valaki nagy eredményt ér is el, nem lesz automatikusan profi, pont úgy, mint ha valaki megtanul egy receptet, még nem biztos hogy el is tudja készíteni az adott ételt.



7.1. ábra. BrainB működés közben

Nézzünk bele a kódba: BrainBThread.h

```
class Hero
{
public:
    int x;
    int y;
    int color;
    int agility;
    int conds {0};
    std::string name;

    Hero ( int x=0, int y=0, int color=0, int agility=1, std::string name ←
        ="Samu Entropy" ) :
        x ( x ), y ( y ), color ( color ), agility ( agility ), name ( name ←
        )
    {}
    ~Hero() {}

    void move ( int maxx, int maxy, int env ) {

        int newx = x+ ( ( ( double ) agility*1.0 ) * ( double ) ( std::rand ←
            () / ( RAND_MAX+1.0 ) )-agility/2 ) ;
        if ( newx-env > 0 && newx+env < maxx ) {
            x = newx;
        }
        int newy = y+ ( ( ( double ) agility*1.0 ) * ( double ) ( std::rand ←
            () / ( RAND_MAX+1.0 ) )-agility/2 ) ;
        if ( newy-env > 0 && newy+env < maxy ) {
            y = newy;
        }

    }

};
```

Ez ennek a header filenak a legfontosabb része. Itt egy osztályt készítünk Hero néven, ő lesz a "hősünk", az az entropy, amin a kurzort kell tartanunk. A konstruktorban megadjuk az alapvető információkat, mint a nevét, a helyzetét, a színét stb. , majd láthatjuk a destruktort is. Ezek alatt található az entropy mozgatása, ami fontos a program szempontjából. A teljes header fileban megtalálható még egy rendszer, amivel a pontokat tudjuk számolni, és ez alapján nehezedik a játék, értsd nő a hős agilityje.

BrainBThread.cpp

```
#include "BrainBThread.h"

BrainBThread::BrainBThread ( int w, int h )
{

    dispShift = heroRectSize+heroRectSize/2;
```

```
this->w = w - 3 * heroRectSize;
this->h = h - 3 * heroRectSize;

std::srand ( std::time ( 0 ) );

Hero me ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - ←
    100,
        this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - ←
        100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), 9 );

Hero other1 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←
    ) - 100,
        this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←
        ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), ←
        5, "Norbi Entropy" );
Hero other2 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←
    ) - 100,
        this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←
        ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), ←
        3, "Greta Entropy" );
Hero other4 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←
    ) - 100,
        this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←
        ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), ←
        5, "Nandi Entropy" );
Hero other5 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←
    ) - 100,
        this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←
        ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), ←
        7, "Matyi Entropy" );

heroes.push_back ( me );
heroes.push_back ( other1 );
heroes.push_back ( other2 );
heroes.push_back ( other4 );
heroes.push_back ( other5 );

}

BrainBThread::~BrainBThread()
{

}

void BrainBThread::run()
{
    while ( time < endTime ) {

        QThread::msleep ( delay );
```

```
        if ( !paused ) {

            ++time;

            devel();

        }

        draw();

    }

    emit endAndStats ( endTime );

}

void BrainBThread::pause()
{

    paused = !paused;
    if ( paused ) {
        ++nofPaused;
    }

}

void BrainBThread::set_paused ( bool p )
{

    if ( !paused && p ) {
        ++nofPaused;
    }

    paused = p;

}
```

Ebben a cpp fileban találhatjuk meg a header fileok függvényeinek kidolgozását, és a program kezdetekor letermő entropyk kezelését is. Ezen felül a kód végén láthatunk egy pause funkció implemetálást is.

7.5. Minecraft MALMÖ -Virág összeszedés

Videó: [20 pipacs](#)

Forrás link: [Github link](#) (ez már a továbbfejlesztett változat)

Ebben a feladatban össze kell állítanunk egy programot az eddig megírtakból, ami a lehetőleg 19 virágot

szed össze. Itt Bátfai Norbert forrását használtam, amit átírtam, hogy sikerüljön összeszedni a kitűzött számú virágokat

DRAFT

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

Megoldás videó: <https://youtu.be/j7f9SkJR3oc>

Megoldás forrása: <https://github.com/tensorflow/tensorflow/releases/tag/v0.9.0> (/tensorflow-0.9.0/tensorflow/exa
https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol

A feladat megoldásához használom a tensorflow nevű alkalmazást. Ennek az aplikációnak a használatát megtalálhatjuk az alkalmazás weboldalán. Amit tudni kell a tensorflowról:

Ez egy open source platform, amely a gépi tanulás világába kalauzolja el a programozót.

Ezen a platformon rengeteg eszköz áll rendelkezésünkre, hogy megértsük a gépi tanulást, és érdekes projekteket tudjunk létrehozni.

Felhő alapú

Most, hogy ezeket tisztáztuk, térjünk rá, hogy mit is csinál a programunk

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])
```

A megfelelő könyvtárak importálása után az adatokat teszt adaszerkezetekbe töltjük fel. Külön töltjük be a traint és külön a testet. Ezek után jöhet is a neurális háló elkészítése, ennek segítségével tanítjuk fel az adatokat. A tensorflow itt is sok segítséget nyújt, jelen esetben a beépített modelleket használjuk fel. Itt adjuk meg az input méretét, ami jelen esetben 28x28. Látható még a köztes réteg megadása is, amit pár

neuron eltávolítása követ, ezzel elkerülve a túlillesztést. Végül megadjuk a kimenetet is, és hogy ez hány elemű lesz.

```
predictions = model(x_train[:1]).numpy()
predictions

tf.nn.softmax(predictions).numpy()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

loss_fn(y_train[:1], predictions).numpy()
```

Predikciókra is szükségünk lesz, ezeket a train adatbázison kell használnunk. Ezek a predikciók fognak a neurális háló kimeneteként szolgálni. A softmax ezek átkonvertálására szolgál, így lesznek osztályok. A loss függvényt is létrehozuk, ebben is segít a tensorflow. Ez a függvény a neurális háló kimenete alapján fog működni.

```
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

model.evaluate(x_test, y_test, verbose=2)

probability_model = tf.keras.Sequential([
    model,
    tf.keras.layers.Softmax()
])

probability_model(x_test[:5])
```

Végül létrehozuk a modelleket, a compile segítségével. Ezt a loss és az optimizer paraméterezi, és egy metrikát is alkalmazunk. A tanítás elindítását a model.fit-el tudjuk elindítani, itt meg kell adnunk hogy hány epochost fusson a tanítás, ezek utánmár futtathatjuk is a programot. Minél nagyobb az epochost annál több ideig tart a tanulás, de annál pontosabb is lesz a végeredmény.

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása: https://github.com/Amsekal/Bhax/blob/master/bhax-master/thematic_tutorials/bhax_textbook

Az előzőhöz hasonlóan itt is egy neurális háló létrehozása a cél. Itt azonban futtatáskor eredményt is fogunk kapni. Bemeneti paraméterként megadhatunk egy képet amin egy szám szerepel, és a ő képes lesz eldönteni, hogy milyen számjegy található a bemeneti képen. Továbbra is igaz, hogy minél több időt adunk tanulni a gépnek, annál pontosabb lesz.

8.3. Minecraft-MALMÖ és 8.4 Vörös Pipacs pokol/javíts a 19 RF-en

Megoldás videó: <https://youtu.be/bAPSu3Rndi8> Megoldás videó 8.4: <https://youtu.be/BxKswdRrUns>

Megoldás forrása: <https://github.com/Amsekal/Bhax/blob/master/smartsteve.py>

8.4-ben tutorált: Hosszú Szilárd

Minden fejezethez tartozott egy malmős feladat, ezek forrását a <https://github.com/Amsekal/Bhax> repóban lehet megtalálni. A Malmö projekt segítségével bárki betekintést nyerhet a mesterséges intelligencia világába, ezen belül is annak programozásába. A projekt kretein belül a Minecraft segítségével készíthetünk egy mesterséges intelligenciát. A Magas szitnű programozási nyelvek 1 tantárgy segítségével mi is kipróbálhatuk magunkat, és egy versenysorozaton összemérhettük tudásunkat és szerencsénket is. A verseny célja egy olyan MI megírása volt, amely képes a tölcészerű pályán Pipacsokat összeszedni, még mielőtt elérné a fentről folyó láva.

Ennek a projektnek köszönhetően rengeteget tanultam az ágensek írásáról és működéséről. Érdekes és sok gondolkodást igénylő feladatok elé állított, amiket többnyire meg is tudtam oldani, még ha néha csak segítséggel is. Biztos vagyok benne hogy a jövőben is hasznát fogom venni annak a tudásnak amire itt tettem szert, habár a mesterséges intelligencia jelenleg még csak szárnyait bontogatja. A projekt nagy tömegeket tud megszólítani, mert az emberek nagy részét érdekli az MI de egy még nagyobb része szereti a Minecraftot, és örül, ha valami érdekeset kreálhat benne. Nagyon sokan állítják, hogy a Minecraft mint játék fejleszti a kreativitást, ha ehhez hozzávesszük a Malmöt, akkor ez a kijelentés biztosan igaz. Örülök, hogy a programozás, és az ágensek írása már nem csak a nagyok kiváltsága, hanem próbálják elérhetővé tenni minden érdeklődő számára. Én jól szórakoztam az ágensek írásával, és szerintem ez egy olyan lehetőség, amit nem szabad kihagyni.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó: <https://youtu.be/z6NJE2a1zIA>, Futtatás: https://youtu.be/E_3xkGiiOJ4 https://youtu.be/E_3xkGiiOJ4

Megoldás forrása: [fact.lisp](#)

Ebben a programban egy szám faktoriálisát számoljuk ki lispen. Mielőtt azonban rátérnénk a programra ejtsünk pár szót a lispről.

A Lisp egy programozási nyelvcsalád, amit manapság már csak kevesen használnak. Ez egy általános programozási nyelv, de használható mesterséges intelligencia programozásában is. Most azonban egy általános program megírására használjuk.

A nyelv listák egymásba ágyazásával működik. Szintaktikája egyszerű, a műveleti jelek a számok/változók előtt szerepelnek. Most, hogy megismertük a nyelvet, nézzük a programot.

```
#!/usr/bin/clisp

(defun factorial_iterative (n)
  (let ((f 1))
    (dotimes (i n)
      (setf f (* f (+ i 1)))))
    f
  )
)
```

Az első sor arra utal, hogy a clisp-et akarjuk használni, tehát először ezt szükséges is telepíteni. A defun létrehoz számunkra egy függvényt, amit factorial_iterative-nak nevezünk el és adunk neki egy n paramétert. A következő sorban létrehozunk egy f változót, aminek a kezdő értéke 1 lesz. Ezek után láthatjuk a dotimes-t ami a lisp-ben egy for ciklushoz hasonló utasítás. A dotimes-ban végezzük el a szorzásokat, számoljuk a faktoriális.

```
(defun factorial_recursive (n)
  (if (= n 0)
      1
      (* n (factorial_recursive (- n 1)) ) )
)
```

Hasonlóan az előző sorokhoz, itt is létrehozunk egy függvényt, azonban itt egy rekurzív megadással érjük el, hogy a programunk számoljon.

```
(format t "Recursive:~%")

(loop for i from 0 to 20
  do (format t "~D! = ~D~%" i (factorial_recursive i)) )
```

Itt található a `factorial_recursive` függvényt meghívjuk 20-ra, tehát kiszámoljuk a 20!-t. a `format` utasítással pedig kiíratunk egy üzenetet.

```
(format t "Iterative:~%")

(loop for i from 0 to 20
  do (format t "~D! = ~D~%" i (factorial_iterative i)) )
```

Hasonlóan az előző meghíváshoz, itt meghívjuk a `factorial_iterative` függvényt, szintúgy 20-ra.

A program nem bonyolult, a nehézséget a programozási nyelv megismerése jelenti, azonban az egyszerű szintaktikája miatt ez sem teljesíthetetlen.

9.2. 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

Kezdeként tisztázzuk mi is az a Gimp. A Gimp egy kép szerkesztő/manipuláló applikáció, melyben rétegeket is létrehozhatunk, hogy a kép jobban szerkeszthető legyen.

A Gimp egy másik különlegessége, hogy feladatok elvégzésére írhatunk szkripteket. Ezek a szkriptek a Script-fu nyelven íródnak, ami a Lisp nyelvcsaládjába tartozik, így szintaktikája is nagyban hasonlít az első feladatban megírt programéhoz.

A feladat lényege, hogy a beírt szöveget krómozott effektel lássuk el, és ezt egy szkriptel érjük el.

```
(define (color-curve)
  (let* (
    (tomb (cons-array 8 'byte))
  )
    (aset tomb 0 0)
    (aset tomb 1 0)
    (aset tomb 2 50)
    (aset tomb 3 190)
    (aset tomb 4 110)
    (aset tomb 5 20)
    (aset tomb 6 200)
    (aset tomb 7 190)
    tomb)
  )
```

Szükségünk lesz egy color-curve függvényre, ahol egy tömbbe beírjuk azokat a számokat, amelyek szükségesek lesznek a krómozáshoz.

```
(define (elem x lista)

  (if (= x 1) (car lista) (elem (- x 1) (cdr lista) ) )

)

(define (text-wh text font fontsize)
(let*
  (
    (text-width 1)
    (text-height 1)
  )

  (set! text-width (car (gimp-text-get-extents-fontname text fontsize ↵
    PIXELS font)))
  (set! text-height (elem 2 (gimp-text-get-extents-fontname text ↵
    fontsize PIXELS font)))

  (list text-width text-height)
)
)
```

Itt két függvényt is létrehozunk, az elsővel a betű helyzetét nézzük a szövegben, míg a másodikkal a szöveg dimenzióit alakítjuk.

```
(define (script-fu-bhax-chrome text font fontsize width height color ↵
  gradient)
(let*
  (
    (image (car (gimp-image-new width height 0)))
    (layer (car (gimp-layer-new image width height RGB-IMAGE "bg" 100 ↵
      LAYER-MODE-NORMAL-LEGACY)))
    (textfs)
    (text-width (car (text-wh text font fontsize)))
    (text-height (elem 2 (text-wh text font fontsize)))
    (layer2)
  )
)
```

Létrehozzuk a script-fu szkriptünket is, e nélkül ugyanis nem tudnánk elérni célunkat. Itt hozzuk létre az összes dolgot, amire a szkriptnek szüksége lehet, szöveg magasság, szélesség, layerek stb. Ezek után következik pár lépés:

```
(gimp-image-insert-layer image layer 0 0)
(gimp-context-set-foreground '(0 0 0))
(gimp-drawable-fill layer FILL-FOREGROUND )
(gimp-context-set-foreground '(255 255 255))
```

```
(set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) ←  
))  
(gimp-image-insert-layer image textfs 0 0)  
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (- (/ ←  
height 2) (/ text-height 2)))  
  
(set! layer (car (gimp-image-merge-down image textfs CLIP-TO-BOTTOM- ←  
LAYER)))
```

Először adunk egy réteget a képünkhöz, a betű színeit fehérre, míg a hátteret feketére állítjuk.

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 15 TRUE TRUE)
```

Itt egy már megírt plugint használunk, ami elmosódás látszatát kelti majda képen.

```
(gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE)
```

Itt a kép frekvenciáját állítjuk be.

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE)
```

Újra használunk elmosódást, most kisebb értékkel.

```
(gimp-image-select-color image CHANNEL-OP-REPLACE layer '(0 0 0))  
(gimp-selection-invert image)
```

A fekete szint elválasztjuk, majd invertáljuk a szelektálást.

```
(set! layer2 (car (gimp-layer-new image width height RGB-IMAGE "2" 100 ←  
LAYER-MODE-NORMAL-LEGACY)))  
(gimp-image-insert-layer image layer2 0 0)
```

```
(gimp-context-set-gradient gradient)  
(gimp-edit-blend layer2 BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY GRADIENT- ←  
LINEAR 100 0 REPEAT-NONE  
FALSE TRUE 5 .1 TRUE width (/ height 3) width (- height (/ height ←  
3)))
```

```
(plug-in-bump-map RUN-NONINTERACTIVE image layer2 layer 120 25 7 5 5 0 0 ←  
TRUE FALSE 2)
```

```
(gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))
```

```
(gimp-display-new image)  
(gimp-image-clean-all image)  
)  
)
```

A többi lépés is azt a célt szolgálja, hogy a képen elérjük a választott effektet

```
(script-fu-register "script-fu-bhax-chrome"
  "Chrome3"
  "Creates a chrome effect on a given text."
  "Norbert Bátfai"
  "Copyright 2019, Norbert Bátfai"
  "January 19, 2019"
  ""
  SF-STRING      "Text"      "Bátf41 Haxor"
  SF-FONT         "Font"      "Sans"
  SF-ADJUSTMENT   "Font size" '(100 1 1000 1 10 0 1)
  SF-VALUE        "Width"     "1000"
  SF-VALUE        "Height"    "1000"
  SF-COLOR        "Color"     '(255 0 0)
  SF-GRADIENT     "Gradient"  "Crown molding"
)
(script-fu-menu-register "script-fu-bhax-chrome"
  "<Image>/File/Create/BHAX"
)
```

Végül regisztrálnunk kell a szkriptet, hogy használni tudjuk.



Amsekal

9.1. ábra. Krómozás

9.3. 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_hackese_a_scheme_programozasi_nyelv

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

I RFH verseny-ből nyert passzomat felhasználom.

DRAFT

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

[?]

A programozási nyelveket 3 részre oszthatjuk: assembly, gépi és magas szintű nyelv, amivel forrásszöveget írhatunk. A forrásszövegből a gép által is értelmezhető kódot fordítóprogramok segítségével érhetünk el. A fordítóprogram lépései: lexikális, szintaktikai, szemantikai elemzés, kódgenerálás. Minden programnyelv rendelkezik jól meghatározott szabvánnyal melyben megtalálhatóak a nyelv szabályai. Az implementációk inkompatibilitását a hordozhatóság problémájának nevezzük, ez még a mai napig vár egy teljes megoldásra. A programnyelvek lehetnek: imperatív, deklaratív, egyéb nyelvek. Imperatív nyelvek: algoritmusokat használnak. Deklaratív nyelvek: nem használnak algoritmusokat, pl logikai nyelv. Más nyelvek: nincs jellemzőjük.

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

Alapismeretek

A könyv első fejezetében a C nyelv alapjaival ismerkedhetünk meg. Egyszerű feladatok segítségével ismerjük meg a nyelv szematikáját, szabványát, a változók használatát és bemutatásra kerülnek az ismétlődő ciklusok is. Ezen felül foglalkozunk még alapvető függvényekkel és pár napi szinten használt eljárással.

Tipusok, operátorok, kifejezések

Ahogy a fejezet címéből is kiderül, a változók típusaival, elnevezési szabályaival és felhasználásával találkozunk. Megtudjuk a pi és az Euler féle szám értékét és megtanulunk mi is ezekhez hasonló állandót létrehozni. Találkozunk a logikai és matematikai operátorokkal, és ezek felhasználásával, de a bitenkénti logikai operátorok sem maradnak ki.

10.3. Programozás

[BMECPP]

Az első fejezet betekintést enged nyerni a c++ történelmébe.

A második fejezet már több információval szolgál programozás terén, bemutatja a c++ nyelvet és az alapvető különbségeket közte és a C között. Megimserkedünk a main függvénnel, és olyan dolgokkal, amik C-ben nincsenek: bool adattípus, stb. Kitér az új nyelvi szabályokra is, ami c után egy kicsit furcsa lehet, például struct helyett class-t használunk, de rengeteg módosítás történt.

10.4. Programozás, Python

Python nyelv bemutatása

A Python egy, a többi nyelvhez képest gyors fejlesztési sebességgel rendelkező nyelv, rengeteg támogatott eszközzel rendelkezik, ezen felül egy általános célú programozási nyelv

Python

Pythonban nem található meg a jól ismert begi, end, de még a pontos vessző sem. Nem szükséges változókat deklarálni, azok bármikor használhatóak, így elmondható, hogy az egyik leg kezdőbarát prog. nyelv. A nyelv rendkívül különleges szemantikával és szabályrendszerrel rendelkezik, főleg ha az ember C vagy C++-ról tér át.

III. rész

Második felvonás

DRAFT

**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.

DRAFT

11. fejezet

Helló, Arroway!

11.1. Olvasónapló

Forrás: [Olvasónapló](#)

11.2. OO szemlélet

A módosított polártranszformációs normális generátor beprogramozása Java nyelven. Mutassunk rá, hogy a mi természetes saját megoldásunk (az algoritmus egyszerre két normálist állít elő, kell egy példánytag, amely a nem visszaadottat tárolja és egy logikai tag, hogy van-e tárolt vagy futtatni kell az algot.) és az OpenJDK, Oracle JDK-ban a Sun által adott OO szervezés ua.!

A módosított polártranszformációs normális generátor segítségével képesek vagyunk legenerálni 2 normálist, az egyiket a program elmenti és egy logikai változó használatával jelzi, hogy rendelkezik e elmentett számmal.

A Java kód:

```
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;
    }
}
```

Ahogy a kódban is látható, egy osztályt hozunk létre a generátor számára, ezt PolárGenerátornak nevezzük el. Deklaráljuk a fentebb említett logikai változót is, `nincsTárolt` néven. A `tárolt` változóban lesz eltárolva a leggenerált normális. A következő függvény, aminek neve egyezik az osztály nevével, egy konstruktor lesz, amikor egy egyedet előállítunk, a logikai változót igazra állítjuk, mert `mlg` nincs semmi eltárolva. Ezek után következik a számolás: ha nincs tárolt szám, akkor végrehajtódnak a matematikai műveletek, eltároljuk a polártranszformáció eredményét. Ha már volt tárolt szám, akkor annak az értékét adjuk vissza.

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

    PolárGenerátor g = new PolárGenerátor();

    for(int i=0; i<10; ++i)
        System.out.println(g.következő());

}
```

Ezek után már csak meg kell hívnunk a függvényeket, és működik is a programunk.

A C++ kód a 6. fejezet 1. feladatában tekinthető meg.

11.3. Gagy

Az ismert formális³ `while (x <= t && x >= t && t != x);` tesztkérdéstípusra adj a szokásosnál (miszerint `x`, `t` az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciája) „mélyebb” választ, írd Java példaprogramot mely egyszer végtelen ciklus, más `x`, `t` értékekkel meg nem! A példát építsd a JDK `Integer.java` forrására⁴, hogy a 128-nál inkluzív objektum példányokat poolozza!

Hogy pontosan megértsük, miért is fordul elő ez a jelenség, először szemügyre kell vennünk, hogy a Java hogyan is dolgozik az integerekkel. Ezt a dokumentáció alapján tehetjük meg.

```
public static Integer valueOf(int i)
```

Returns an Integer instance representing the specified int value. ←
If a new Integer instance is not required, this method should ←
generally be used in preference to the constructor `Integer(int)`, ←
as this method is likely to yield significantly better space ←
and time performance by caching frequently requested values. ←
This method will always cache values in the range -128 to 127, ←
inclusive, and may cache other values outside of this range.

Parameters:

`i` – an int value.

Returns:

an Integer instance representing `i`.

Since:

1.5

Ahogy a fenti dokumentációban láthatjuk a Java ebben az esetben a gyorsaságot fontosnak találja, így egy gyorsítótárba elmentni a -128,127 zárt intervallumban található számokat. Ezt azért teszi, mert azt feltételezi, hogy ezek a számok gyakran fognak előfordulni a legtöbb kódban. Tehát, ha ebből az intervallumból választunk egy számot, akkor az értéket a gyorsítótárból választja ki a Java.

```
[while (x <= t && x >= t && t != x)  
System.out.println("Belepett");
```

Vegyünk példának a fenti kódrészletet. Ha a fentiekben említett intervallumból veszünk azonosértéket `x`-nek és `t`-nek, nem lép be a `while` ciklusba, mivel a harmadik feltétel nem teljesül. A két változó ugyanarra az objektumra hivatkozik, de ha nem az intervallumból vesszük a számokat, akkor ez nem történik meg.

11.4. Yoda

Írjunk olyan Java programot, ami `java.lang.NullPointerException`-el leáll, ha nem követjük a Yoda conditions-t!
https://en.wikipedia.org/wiki/Yoda_conditions

Yodát szerintem senkinem sem szükséges bemutatni. A kis zöld lény ikonikussá vált a "Csillagok Háborúja" filmeknek köszönhetően. Ami viszont teljesen egyedivé teszi, az a fura beszédmódja, ugyanis az igéket a szavak végére teszi. Ezt használták alapnak a programozók, amikor létrehozták a Yoda conditions programozási sémát: a feltételeket fordítva írjuk fel, például `if(osszeg == 10)` helyett, `if(10==osszeg)` fog szerepelni a kódunkban. Ez egy haszontalan ötletnek tűnik, de néha képes csodákra.

```
String myString = null;
if (myString.equals("foobar")) {
    System.out.println("Yoda");
}
```

Ebben az esetben egy `NullPointerException` fog dobni a Java, ugyanis `null`-hoz nem tud összehasonlítani valamivel.

```
String myString = null;
if ("foobar".equals(myString))
{ System.out.println("Yoda"); }
```

Itt egy létező stringet akarunk összehasonlítani egy `null` értékkel, ami Javában már lehetséges, hiba helyett egy `false` értéket fogunk kapni az ifben.

11.5. Kódolás from scratch

Induljunk ki ebből a tudományos közleményből: <http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/bbpalg.pdf> és csak ezt tanulmányozva írjuk meg Java nyelven a BBP algoritmus megvalósítását! Ha megakadsz, de csak végső esetben: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok/javat/apbs02.html#pi_jegyei (mert ha csak lemásolod, akkor pont az a fejlesztői élmény marad ki, melyet szeretném, ha átélnél).

A feladat megoldásához Bátfai Norbert Tanár úr kódját fogom felhasználni.

A megemlített tudományos szövegben a BBP formulát olvasatunk, ennek segítségével ki tudjuk számolni a Pi értékét.

```
public class PiBBP {

    String d16PiHexaJegyek;

    public PiBBP(int d) {

        double d16Pi = 0.0d;

        double d16S1t = d16Sj(d, 1);
        double d16S4t = d16Sj(d, 4);
        double d16S5t = d16Sj(d, 5);
        double d16S6t = d16Sj(d, 6);
```

```
d16Pi = 4.0d*d16S1t - 2.0d*d16S4t - d16S5t - d16S6t;

d16Pi = d16Pi - StrictMath.floor(d16Pi);

StringBuffer sb = new StringBuffer();

Character hexaJegyek[] = {'A', 'B', 'C', 'D', 'E', 'F'};

while(d16Pi != 0.0d) {

    int jegy = (int)StrictMath.floor(16.0d*d16Pi);

    if(jegy<10)
        sb.append(jegy);
    else
        sb.append(hexaJegyek[jegy-10]);

    d16Pi = (16.0d*d16Pi) - StrictMath.floor(16.0d*d16Pi);
}

d16PiHexaJegyek = sb.toString();
}

public double d16Sj(int d, int j) {

    double d16Sj = 0.0d;

    for(int k=0; k<=d; ++k)
        d16Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);

    return d16Sj - StrictMath.floor(d16Sj);
}
```

A fenti kódrészben létrehozzuk a fő osztályunkat, majd ebben egy stringet készítünk, ebben lesz az eredmény eltárolva. Ezek után következik a formula implementálása. Itt a lényeges érzés az, hogy a PiBBP d paramétere tárolja el a pozíciót, ahonnan érdekel majd minket a Pi számjegyei.

```
public long n16modk(int n, int k) {

    int t = 1;
    while(t <= n)
        t *= 2;

    long r = 1;

    while(true) {

        if(n >= t) {
            r = (16*r) % k;
        }
    }
}
```



```
        n = n - t;
    }

    t = t/2;

    if(t < 1)
        break;

    r = (r*r) % k;

}

    return r;
}
public String toString() {

    return d16PiHexaJegyek;
}
public static void main(String args[]) {
    System.out.print(new PiBBP(1000000));
    System.out.println("\n");
}
}
```

Ezek után már csak annyi dolgunk maradt, hogy átalakítsuk hexadecimális alakba, majd kiírjuk toString segítségével a számokat. A kód végén egy példányosítás látható, itt kapjuk majd meg az eredményt is.

12. fejezet

Helló, Liskov!

12.1. Liskov helyettesítés sértése

Írjunk olyan OO, leforduló Java és C++ kódcsipetet, amely megsérti a Liskov elvet! Mutassunk rá a megoldásra: jobb OO tervezés.

A Liskov Helyettesítési Elv kimondja, hogy ha A alosztálya B, akkor minden A típusú objektum helyettesíthető B típusú objektummal, úgy, hogy a program működése és tulajdonságai azonosak maradnak.

Ahhoz, hogy ezt az elvet megsértsük, egy olyan alosztályra van szükségünk, ami nem rendelkezik a főosztály egyik, vagy akár több funkciójával. Ilyen példa lehet a kacsacsőrű emlős és a többi emlős szaporodása, mert míg a kacsacsőrű emlős tojást rak, a többi emlős állat eleven utódokat hoz létre, vagy egy egyszerűbb példa, a Pingvin, a röpképtelen madár és a többi madár akik képesek repülni. Én az utóbbi példát fogom felhasználni. A kódhoz Bátfai Norbert tanárúr kódját fogom átírni Java nyelvre.

Java kód:

```
class Madar {  
  
    public void repul() {  
  
    };  
};  
  
class Program {  
  
    public void fgv ( Madar madar ) {  
        madar.repul();  
    }  
};  
  
class Kacsa extends Madar  
{  
  
};
```

```
class Pingvin extends Madar
{
};

class LSP {
    public static void main (String[] args)
    {
        Program program = new Program();
        program.fgv (new Madar() );

        program.fgv (new Kacsa() );

        program.fgv (new Pingvin() );
    }
}
```

A fenti kódcsipetben létrehoztunk egy Madár osztályt, amiben a repul függvény is szerepel. Ha ennek az osztálynak készítünk egy alosztályt, akkor abban is szerepelni fog a repul függvény. A Program osztályban az fgv függvény alkalmazza majd a Madar típusú objektumokra a repul függvényt. Ezek után létrehozzuk az alosztályokat, majd a Main-ben létrehozzuk az objektumokat, egy Madar egy Kacsa és egy Pingvin típusú objektumot, amik renfelkezni fognak a repul függvénnel. Nézzük meg ezt C++ban is.

```
class Madar {
public:
    virtual void repul() {};
};

class Program {
public:
    void fgv ( Madar &madar ) {
        madar.repul();
    }
};

class Kacsa : public Madar
{
};

class Pingvin : public Madar
{
};

int main ( int argc, char **argv )
{
    Program program;
    Madar madar;
    program.fgv ( madar );
}
```

```
Kacsa kacsa;  
program.fgv ( kacsa );  
  
Pingvin pingvin;  
program.fgv ( pingvin );  
  
}
```

Ezek után, nézzük meg, hogyan lehetne elkerülni az elv megsértését:

```
class Madar {  
  
};  
  
class Program {  
public:  
    void fgv ( Madar &madar ) {  
  
    }  
};  
  
class RepuloMadar : public Madar {  
public:  
    virtual void repul() {};  
};  
  
class Kacsa : public RepuloMadar  
{};  
  
class Pingvin : public Madar  
{};  
  
int main ( int argc, char **argv )  
{  
    Program program;  
    Madar madar;  
    program.fgv ( madar );  
  
    Kacsa kacsa;  
    program.fgv ( kacsa );  
  
    Pingvin pingvin;  
    program.fgv ( pingvin );  
  
}
```

Úgy orvosoltuk a problémát, hogy a Madar osztályból eltűnt a repul függvény, egy alosztályába került, a RepuloMadar alosztályba. A röpképtelen madarak a Madar alosztályai lesznek, a röpképesek pedig a RepuloMadaré.

12.2. Szülő-gyerek

Írjunk Szülő-gyerek Java és C++ osztálydefiníciót, amelyben demonstrálni tudjuk, hogy az ősön keresztül csak az ős üzenetei küldhetők!

A feladat lényege igazán egyszerű, meg kell próbálnunk a szülőben létrehozott üzenetet egy gyerekben próbáljuk meg kiírni. Kezdjük a Java kóddal:

```
class szulo
{
    public void uzenet ()
    {
        System.out.println("Szulo");
    }
};

class gyerek extends szulo
{
    public void uzenetgyerek()
    {
        System.out.println("Gyerek");
    }

    public static void main(String[] args)
    {
        szulo sz1 = new szulo();
        szulo sz2 = new gyerek();
        sz1.uzenet();
        sz2.uzenetgyerek();
    }
};
```

Ahogy megpróbálnánk ezt futtatni, egyből hibát fogunk kapni, ami bizonyítja, hogy tényleg ősön keresztül csak az ős üzeneteit küldhetjük. C++ kód:

```
#include <iostream>

using namespace std;
```

```
class szulo {
    public:
        void uzenet()
        {
            std::cout<<"Szulo uzenete";
        }
};

class gyerek : public szulo {
    public:
        void gyerekuzenete()
        {
            std::cout<<"Gyerek uzenete";
        }
};

int main()
{
    szulo* sz1 = new szulo();
    szulo* sz2 = new gyerek();
    sz1->uzenet();
    sz2->gyerekuzenete();
    return 0;
}
```

Hasonló végeredményre jutunk, tehát ez a művelet nem csak Javában de C++-ban sem kivitelezhető

12.3. Anti OO

A BBP algoritmussal a Pi hexadecimális kifejtésének a 0. pozíciótól számított 10^6 , 10^7 , 10^8 darab jegyét határozzuk meg C, C++, Java és C# nyelveken és vessük össze a futási időket!

A feladatban előre megírt forráskódokat kell lefuttatni, és a futási időket összehasonlítani.

Java kód:

```
*/
/**
 * A PiBBP.java-ból kivettük az "objektumorientáltságot", így ↵
 * kaptuk
 * ezt az osztályt.
 *
 * (A PiBBP osztály a BBP (Bailey-Borwein-Plouffe) algoritmust a Pi ↵
 * hexa
 * jegyeinek számolását végző osztály. A könnyebb olvashatóság
 * kedvéért a változó és metódus neveket megpróbáltuk az ↵
 * algoritmust
```

```
* bemutató [BBP ALGORITMUS] David H. Bailey: The BBP Algorithm for ↵
  Pi.
* cikk jelöléseihhez.)
*
* @author Bátfai Norbert, nbatfai@inf.unideb.hu
* @version 0.0.1
*/
public class PiBBPBench {

    public static double d16Sj(int d, int j) {

        double d16Sj = 0.0d;

        for(int k=0; k<=d; ++k)
            d16Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + ↵
                j);

        return d16Sj - Math.floor(d16Sj);
    }

    public static long n16modk(int n, int k) {

        int t = 1;
        while(t <= n)
            t *= 2;

        long r = 1;

        while(true) {

            if(n >= t) {
                r = (16*r) % k;
                n = n - t;
            }

            t = t/2;

            if(t < 1)
                break;

            r = (r*r) % k;

        }

        return r;
    }

    public static void main(String args[]) {

        double d16Pi = 0.0d;
```

```
double d16S1t = 0.0d;
double d16S4t = 0.0d;
double d16S5t = 0.0d;
double d16S6t = 0.0d;

int jegy = 0;

long delta = System.currentTimeMillis();

for(int d=100000000; d<1000000001; ++d) {

    d16Pi = 0.0d;

    d16S1t = d16Sj(d, 1);
    d16S4t = d16Sj(d, 4);
    d16S5t = d16Sj(d, 5);
    d16S6t = d16Sj(d, 6);

    d16Pi = 4.0d*d16S1t - 2.0d*d16S4t - d16S5t - d16S6t;

    d16Pi = d16Pi - Math.floor(d16Pi);

    jegy = (int)Math.floor(16.0d*d16Pi);

}

System.out.println(jegy);
delta = System.currentTimeMillis() - delta;
System.out.println(delta/1000.0);
}
```

Ez a Java megoldása a feladatnak. A kapott eredmények: 10^6 : 1.35, 10^7 :15.80, 10^8 : 180.29

A megoldás C sharpban:

```
public class PiBBPBench {

    public static double d16Sj(int d, int j) {

        double d16Sj = 0.0d;

        for(int k=0; k<=d; ++k)
            d16Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);

        return d16Sj - System.Math.Floor(d16Sj);
    }
}
```



```
public static long n16modk(int n, int k) {  
  
    int t = 1;  
    while(t <= n)  
        t *= 2;  
  
    long r = 1;  
  
    while(true) {  
  
        if(n >= t) {  
            r = (16*r) % k;  
            n = n - t;  
        }  
  
        t = t/2;  
  
        if(t < 1)  
            break;  
  
        r = (r*r) % k;  
  
    }  
  
    return r;  
}  
  
public static void Main(System.String[]args) {  
  
    double d16Pi = 0.0d;  
  
    double d16S1t = 0.0d;  
    double d16S4t = 0.0d;  
    double d16S5t = 0.0d;  
    double d16S6t = 0.0d;  
  
    int jegy = 0;  
  
    System.DateTime kezd = System.DateTime.Now;  
  
    for(int d=1000000; d<1000001; ++d) {  
  
        d16Pi = 0.0d;  
  
        d16S1t = d16Sj(d, 1);  
        d16S4t = d16Sj(d, 4);  
        d16S5t = d16Sj(d, 5);  
        d16S6t = d16Sj(d, 6);  
    }  
}
```

```
        d16Pi = 4.0d*d16S1t - 2.0d*d16S4t - d16S5t - d16S6t;

        d16Pi = d16Pi - System.Math.Floor(d16Pi);

        jegy = (int)System.Math.Floor(16.0d*d16Pi);

    }

    System.Console.WriteLine(jegy);
    System.TimeSpan delta = System.DateTime.Now.Subtract(kezd);
    System.Console.WriteLine(delta.TotalMilliseconds/1000.0);
}
}
```

A futtatás eredményei a C sharp verzióra: 10^6 : 1.69, 10^7 : 19.5, 10^8 : 216.18

C/C++ forrás:

```
#include <stdio.h>
#include <math.h>
#include <time.h>
/*
 * pi_bbp_bench.c
 *
 * DIGIT 2005, Javat tanítok
 * Bátfai Norbert, nbatfai@inf.unideb.hu
 *
 * A PiBBP.java-ból kivettük az "objektumorientáltságot", így ↵
 *   kaptuk
 * a PiBBPBench osztályt, amit pedig átírtuk C nyelvre.
 *
 */

long
n16modk (int n, int k)
{
    long r = 1;

    int t = 1;
    while (t <= n)
        t *= 2;

    for (;;)
    {
        if (n >= t)
        {
            r = (16 * r) % k;
            n = n - t;
        }
    }
}
```

```
    }

    t = t / 2;

    if (t < 1)
        break;

    r = (r * r) % k;

}

return r;
}

double
d16Sj (int d, int j)
{

    double d16Sj = 0.0;
    int k;

    for (k = 0; k <= d; ++k)
        d16Sj += (double) n16modk (d - k, 8 * k + j) / (double) (8 * k + j);

    return d16Sj - floor (d16Sj);
}

main ()
{

    double d16Pi = 0.0;

    double d16S1t = 0.0;
    double d16S4t = 0.0;
    double d16S5t = 0.0;
    double d16S6t = 0.0;

    int jegy;
    int d;

    clock_t delta = clock ();

    for (d = 100000000; d < 1000000001; ++d)
    {

        d16Pi = 0.0;

        d16S1t = d16Sj (d, 1);
        d16S4t = d16Sj (d, 4);
```

```
d16S5t = d16Sj (d, 5);  
d16S6t = d16Sj (d, 6);  
  
d16Pi = 4.0 * d16S1t - 2.0 * d16S4t - d16S5t - d16S6t;  
  
d16Pi = d16Pi - floor (d16Pi);  
  
jegy = (int) floor (16.0 * d16Pi);  
  
}  
  
printf ("%d\n", jegy);  
delta = clock () - delta;  
printf ("%f\n", (double) delta / CLOCKS_PER_SEC);  
}
```

Eredmények: 10^6 : 1.48, 10^7 : 17.68, 10^8 : 202.97

Ha elvégezzük az összehasonlítást, egyértelműen látszik, hogy a Java teljesített a legjobban, utána következik a C, majd utolsóként a C sharp.

12.4. Ciklomatikus komplexitás

Számoljuk ki valamelyik programunk függvényeinek ciklomatikus komplexitását! Lásd a fogalom tekintetében a https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_2.pdf (77-79 fóliát)!

A Ciklomatikus komplexitás egy szoftvermetrika, aminek a segítségével, a forráskód elemzése után megtudja állapítani egy szoftver komplexitását, és ezt egy számmal kifejezi.

A komplexitás képlete: $M=E-N+2*P$

A képletben az M a komplexitást jelzi, az E a gráf éleinek számát, az N a gráfban levő csúcsok számát, a P pedig az összefüggő komponensek számát.

File Type .cpp Token Count 1320 NLOC 174				
Function Name	NLOC	Complexity	Token #	Parameter #
BinTree::Node::Node	1	1	22	
BinTree::Node::getValue	1	1	8	
BinTree::Node::leftChild	1	1	8	
BinTree::Node::rightChild	1	1	8	
BinTree::Node::leftChild	1	1	12	
BinTree::Node::rightChild	1	1	12	
BinTree::Node::getCount	1	1	8	
BinTree::Node::IncCount	1	1	8	
BinTree::BinTree	3	1	38	
BinTree::BinTree	4	1	34	
BinTree::cp	13	3	83	
BinTree::operator =	6	1	42	
BinTree::BinTree	5	1	34	
BinTree::operator =	6	1	47	
BinTree::~BinTree	4	1	22	
BinTree::print	1	1	14	
ZLWTree::ZLWTree	3	1	31	
BinTree<ValueType>::operator < <	24	7	162	
ZLWTree<ValueType,vr,v0>::operator < <	21	4	167	
BinTree<ValueType>::print	13	3	103	
BinTree<ValueType>::deltree	9	2	40	
main	20	1	202	

12.1. ábra. Ciklomatikus Komplexitása a z3a18qa5_from_scratch.cpp programnak

A táblázat a [Lizard](#) oldalon található ciklomatikus komplexitást számoló programot használtam fel.

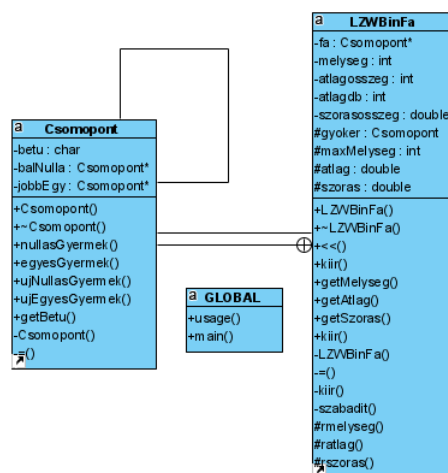
13. fejezet

Helló, Mandelbrot!

13.1. Reverse engineering UML osztálydiagram

UML osztálydiagram rajzolása az első védési C++ programhoz. Az osztálydiagramot a forrásokból generáljuk (pl. Argo UML, Umbrello, Eclipse UML) Mutassunk rá a kompozíció és aggregáció kapcsolatára a forráskódban és a diagramon, lásd még: https://youtu.be/Td_nLERIEOs. Lásd fíliák!

Ebben a feladatban a legelső C++ védési programunk osztálydiagrammját kell elkészítenünk egy erre alkalmas szoftverben. Az én választásom az Visual Paradigm nevű programra esett.



13.1. ábra. .

A fent látható osztálydiagramm a könyv első fejezetében bemutatott Binfá kódjához tartozik. Ezzel a diagrammal ábrázolhatjuk a program felépítését, átláthatóvá teszi, hogy milyen módon kapcsolódnak egymáshoz az osztályok.

Felfedezhető a képen pár érdekes szimbólum, mint például a +,-,#. Ezek jelentései:

A + szimbólum a publikus attribútumoknál figyelhető meg.

A - szimbólum a privát attribútumoknál figyelhető meg.

A # szimbólum a védett attribútumoknál figyelhető meg.

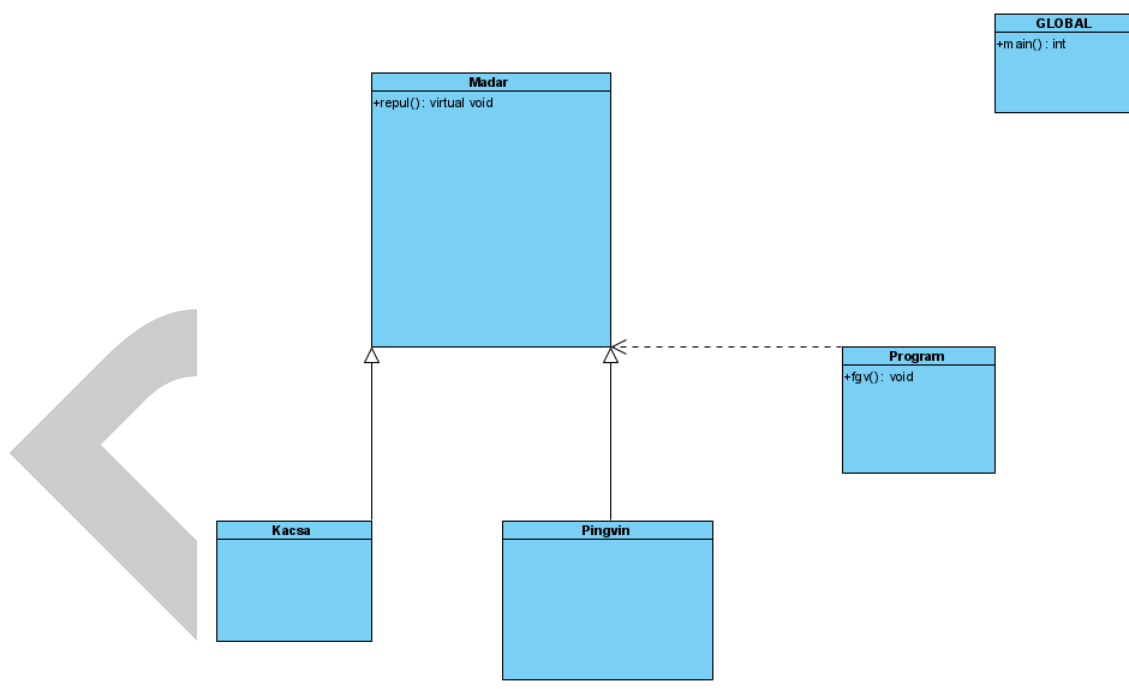
A fekete rombuszt tartalmazó nyíl a kompozíció jele, az üres rombusz az agregáció jele, a szaggatott nyíl a függést szimbolizálja, a sima az asszociációt.

A feladat még elvárja, hogy nézzük meg a kompozíció és aggregáció között lévő kapcsolatot. Az aggregáció két osztály közti kapcsolatra utal, úgy hogy az egyik osztály a másiknak egy része. A kompozíció hasonlóan 2 osztály közti kapcsolatát jelenti, viszont itt az osztályok teljes szinkronban működnek, az objektumok hasonlóan viselkednek majd.

13.2. Forward engineering UML osztálydiagram

UML-ben tervezzünk osztályokat és generáljunk belőle forrást!

A feladat megoldásához felhasználok az előző feladatban leírt madaras példát, ennek készítem el az osztálydiagramját szintén a Visual Paradigm nevű programban



13.2. ábra. .

Látható, hogy van egy madar osztályunk, amely rendelkezik egy függvénnyel, nevezetesen ez a repul függvény. 2 leszármazottja is van, a Kacsa és a Pingvin alosztály képében. A Program osztály függ a Madar osztálytól, ezt mutatja a szaggatott nyíl is.

Most pedig következzen a legenerált forrás:

-GLOBAL.h-

```
#include <exception>
using namespace std;

#ifndef __GLOBAL_h__
#define __GLOBAL_h__

class GLOBAL;

class GLOBAL
{
    public: int main();
};

#endif
```

-GLOBAL.cpp-

```
#include <exception>
using namespace std;

#include "GLOBAL.h"
int GLOBAL::main() {
    throw "Not yet implemented";
}
```

-Madar.h-

```
#include <exception>
using namespace std;

#ifndef __Madar_h__
#define __Madar_h__

class Madar;

class Madar
{
    public: virtual void repul();
};
```



```
};  
  
#endif
```

-Madar.cpp-

```
#include <exception>  
using namespace std;  
  
#include "Madar.h"  
virtual void Madar::repul() {  
    throw "Not yet implemented";  
}
```

-Pingvin.h-

```
#ifndef __Pingvin_h__  
#define __Pingvin_h__  
  
#include "Madar.h"  
  
// class Madar;  
class Pingvin;  
  
class Pingvin: public Madar  
{  
};  
  
#endif
```

-Pingvin.cpp-

```
#include "Pingvin.h"  
#include "Madar.h"
```

-Program.h-

```
#include <exception>  
using namespace std;
```

```
#ifndef __Program_h__
#define __Program_h__

class Program;

class Program
{
    public: void fgv();
};

#endif
```

-Program.cpp-

```
#include <exception>
using namespace std;

#include "Program.h"
void Program::fgv() {
    throw "Not yet implemented";
}
```

-Kacsa.h-

```
#ifndef __Kacsa_h__
#define __Kacsa_h__

#include "Madar.h"

// class Madar;
class Kacsa;

class Kacsa: public Madar
{
};

#endif
```

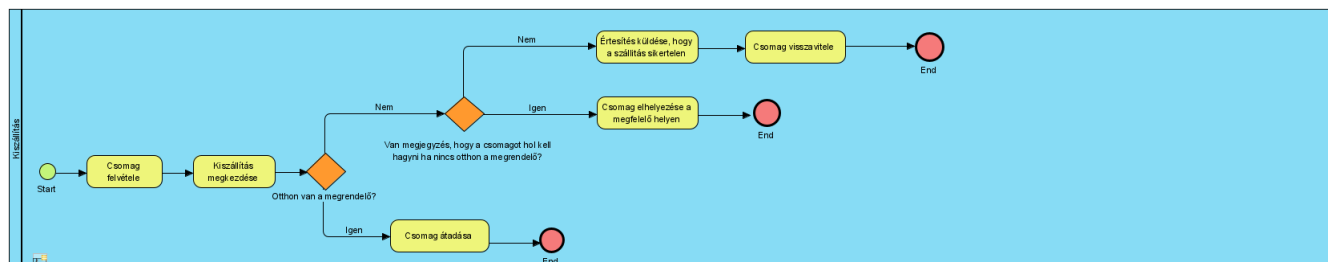
-Kacsa.cpp-

```
#include "Kacsa.h"
#include "Madar.h"
```

13.3. BPMN

Rajzoljunk le egy tevékenységet BPMN-ben!

A feladatban az volt a dolgunk, hogy egy tevékenységet rajzoljunk/modellezzünk le BPMN-ben. Ehhez a már megismert Visual Paradigm-ot fogom felhasználni.



13.3. ábra. BPMN tevékenység

A fentebb látható ábrán egy csomag kiszállítása figyelhető meg. A futár azzal kezdi a kiszállítást, hogy felveszi a csomagot, majd elindul a megfelelő címre. Ha a csomag megrendelője otthon tartózkodik, akkor a csomagot átadja, és befejezte a kiszállítást. Ha nincs otthon akkor egy másik elágazáshoz ér, megnézi, hogy kapott-e a csomag rendelőjétől megjegyzést, hogy mi a teendő ha nem tudja személyesen átadni a csomagot. Ha van, akkor az utasítások alapján cselekszik, ha nincs akkor viszont küld egy értesítést a megrendelőnek, hogy sikertelen a kiszállítás, és a csomagot visszaviszi.

13.4. EPAM: OOP OO modellezés

Írj egy 1 oldalas esszét arról, hogy OO modellezés során milyen elveket tudsz követni (pl.: SOLID, KISS, DRY, YAGNI).

A **SOLID** elv a következő öt elv kezdőbetűiből áll össze:

1. **Single Responsibility principle.** Már a névből is következtethetünk a jeletésére, egy modul egy problémát akarjon csak orvosolni, segítve így az átláthatóságot és a javítások kivitelezését. Sokkal egyszerűbb átlátni a programot, ha nem egy osztályban szeretnénk mindent megoldani, hanem minden feladatra jutna egy.
2. **Open/Closed principle.** Ha már van egy létező, jól működő, a feladatát teljesítő osztályunk, akkor azt ne módosítsuk egy új funkció beépítésekor. Ilyen esetben az osztály kiterjesztése a helyes megoldás, különben a programban hibák jelentkezhetnek, ami elkerülendő.
3. **Liskov Substitution principle.** Ez már az előző fejezetben is szerepelt. Ez a rész azt mondja ki, hogy egy objektum a programunkban helyettesíthető legyen az altípusai példányaival is, úgy, hogy a program működése ne változzon meg. A korábbi példában is láthattuk, hogy ha ez az elv sérül, nem kell mást tennünk csak egy jobban átgondolt implementációt írunk.
4. **Interface Segregation principle.** Ez az elv azt mondja ki, hogy nem jó túl nagy interfészeket használni a programunkban, ezeket inkább szegregálva/széttörve alkalmazzuk. Hasonlóan az első elvhez itt is az a cél, hogy az interfészek specifikusak legyenek, ne egy próbáljon mindent megoldani magában.

5. Dependency Inversion principle. Az elv szerint a felsőbbrendű osztály nem kell ismerje az alacsonyabb rendűek implementációját.

A **KISS**, avagy **Keep It Simple, Stupid** elv kimondja hogy a kódunk legyen egyszerű, bárki számára megérthető, legyen tiszta a kód. A cél, hogy ne csak mi értsük meg, mire hivatott a kódunk, ugyanis lehet, hogy más is dolgozni fog vele. A kis metódusok használata ajánlott, hasonló szabályok vonatkoznak itt is mint a **Single Responsibility Principle**-ben. A változók neve legyen érthető. Minél több esetünk van, annál egyszerűbb észrevenni, hogy pontosan melyik nem működik, vagy nem csinálja azt amit mi szeretnénk. Ha előfordulna, hogy nem teljesül az elv, akkor megoldhatjuk a problémát a kód tördelésével, a refactor használatával.

A **DRY**, azaz a **Don't Repeat Yourself** elv kimondja, hogy a kódunkban minden sor egyszer szerepeljen, ne ismételtessük önmagunkat, ha valamire többször szükségünk van, írjunk rá egy függvényt. Rengeteg pozitív hatása van ennek az elvnek, például, ha a függvényt használjuk, akkor egy esetleges hibát elég egyszer kijavítunk, és mindenhol ahol a függvény meg lett hívva, a probléma elhárul. Ezt az elvet e legkönnyebb megsérteni, ugyanis elég annyi, hogy ne vegyük észre, hogy egy adott részt már használtunk a kód megírása során, és újra leírjuk.

A **YAGNA** elv a következő, jelentése: **You Aren't Gonna Need It**. Az elv kimondja, hogy ne dolgozzunk előre, ne írjunk olyan dolgokat, aminek a működéséhez szükséges dolgokat még nem implementáltuk. Ez azért fontos, mert lehet, hogy végül teljesen más megoldást választunk, és akkor fölöslegesen dolgoztunk, vagy lehet, hogy végül más adatokkal kell majd dolgoznia az előre megírt résznek, mint az eredetileg terveztük.

13.5. .

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

13.6. Általános

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

13.7. C

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

13.8. C++

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

13.9. 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 UDPROG tanulószoba, <https://www.facebook.com/groups/udprog>, a DEAC-Hackers előszoba, <https://www.facebook.com/groups/DEACHackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésükért és hasznos észrevételeikért.

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