

Univerzális programozás

Így neveld a programozód!

Ed. BHAX, DEBRECEN,
2020. március 2, v. 0.0.5

Copyright © 2019 Dr. Bátfai Norbert

Copyright (C) 2019, 2020, 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, és Tutor Tünde	2020. május 12.	

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
0.0.5	2020-03-02	Az Chomsky/ $a^n b^n c^n$ és Caesar/EXOR csokor feladatok kiírásának aktualizálása (a heti előadás és laborgyakorlatok támogatására).	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?	10
2.3. Változók értékének felcserélése	13
2.4. Labdapattogás	15
2.5. Szóhossz és Linus Tovald -féle BogoMIPS	18
2.6. Hello Google!	19
2.7. Monty Hall probléma	22
2.8. 100 éves a Brun Tétel	30
2.9. MALMÖ folytonos csiga feladat	31
3. Helló, Chomsky!	34
3.1. Decimálisból unárisba átváltó Turing gép	34
3.2. Hivatkozási nyelv	35
3.3. Saját lexikális elemző	37
3.4. Leetspeak	39
3.5. A források olvasása	42
3.6. Logikus	43
3.7. Vörös pipaxs pokol/csiga diszkrét mozgási parancsokkal	44

4. Helló, Caesar!	47
4.1. Double ** háromszög mátrix	47
4.2. C EXOR titkosító	49
4.3. 4.3 Java EXOR titkosító	52
4.4. EXOR törő	54
4.5. 4.5 Neurális OR, AND és EXOR kapu	57
4.6. Vörös pipacs pokol/Írd ki mit lát Steve!	64
5. Helló, Mandelbrot!	66
5.1. A Mandelbrot halmaz	66
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztálytalál	70
5.3. Biomorfok	73
5.4. A Mandelbrot halmaz CUDA megvalósítása	80
5.5. Mandelbrot nagyító és utazó C++ nyelven	85
5.6. Mandelbrot nagyító és utazó Java nyelven	91
5.7. Vörös pipacs pokol/fel a láváig és vissza	96
6. Helló, Welch!	98
6.1. LZW	98
6.2. Fabejárás	100
6.3. Mozgató szemantika	104
6.4. Vörös pipacs pokol/ 5x5x5, azaz Steve szemüvege	107
7. Helló, Conway!	109
7.1. Qt C++ életjáték	109
7.2. Vörös pipacs pokol/ 19 RF	119
8. Helló, Schwarzenegger!	123
8.1. Minecraft-MALMÖ	123
8.2. Vörös pipacs pokol/ Javíts a 19RF-en!	123
9. Helló, Chaitin!	125
9.1. Iteratív és rekurzív faktoriális Lisp-ben	125
9.2. Malmö kód továbbfejlesztése	128

10. Hello, Gutenberg!	135
10.1. 10.1. Programozási alapfogalmak	135
10.2. Keringhan.Ritchie: A C programozási nyelv	135
10.3. 10.4. Python bevezetés	136
10.4. 10.2. Szoftverfejlesztés C++ nyelven	137
III. Második felvonás	139
11. Helló, Arroway!	141
11.1. A BPP algoritmus Java megvalósítása	141
11.2. Java osztályok a Pi-ben	141
IV. Irodalomjegyzék	142
11.3. Általános	143
11.4. C	143
11.5. C++	143
11.6. Lisp	143

Ábrák jegyzéke

4.1.	A double ** háromszögmátrix a memóriában	48
4.2.	Az OR művelet	61
4.3.	Az AND és OR művelet	62
4.4.	Az EXOR rejtett rétegek nélkül művelet	63
4.5.	Az EXOR rejtett rétegekkel művelet	64
5.1.	A Júlia halmazokat ábrázoló szemléletes ábra a Fractalgasm Facebook csoportból	74
7.1.	Így néz ki az életjáték Arduinoval	115
7.2.	A leggyakoribb LCD bekötési mód amit én is követtem	116

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

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 kódcsipelek! 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.
- Kódjátszma, <https://www.imdb.com/title/tt2084970/>, benne a **kódtörő feladat** élménye.

- , , benne a bemutatása.

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

**tip**

íkon próba

**warning**

íkon próba

**caution**

íkon próba

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-nI>

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 péláinkban ilyen amikor a PageRank algoritmus rázza az 1 liter vizet az internetben, de az iteráció csak nem akar konvergálni...

A végtelen ciklusok léjtogosultsága eksőre kicsit meglepő lehet a kezdő programozóknak, miért is akarnánk, hogy egy program ne álljon le? Pedig egyébként sokszor nagyon hasznosak. Én egyébként szoktam gyerekekkel foglalkozni, és már velük is volt szó végtelen ciklusokról. A Microsoft Kodu-ban dolgozunk általában, átlag 6 éves korosztállyal és egy akadálypályás játékot kellett készítenünk. A cél az volt, hogy a karakterünkkel át kelljen kelni a szemben oda-vissza mozgó biciklisek között. A biciklisek kódja volt itt a kérdéses, megadott útvonalon kellett oda-vissza haladniuk addig amíg a játék tart. A kulcsó itt amíg a játék tart, persze nem olyan végtelen ciklusról volt szó amivel mi foglalkozunk, de szerintem ez a példa szemléletes. Tehát el kellett érünk, hogy a biciklisek ne hagyják abba mozgásukat. A gyerekekkel ezt a feladatot úgy oldottuk meg, hogy a feltétel megalkotásához 2 blokkot használtunk fel, az egyik a "lát" a másik pedig a "bármí". Tehát ameddig akármit lát a biciklis mozogni fog az útvonalán, és ugye valamit az adott környezetben minden lát tehát nem fog a cilusból kilépni. Egyébként a képen látszik, hogy itt "ha" feltételeket szabunk, de a labdapattogtatós feladatban majd láthatjuk milyen egyszerű ezekből ciklust készíteni.



Egy mag 100 százalékban:

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

    return 0;
}
```

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

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

    return 0;
}
```

```
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/bhx programok/Turing$ gcc infy-f.c -o if
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/bhx programok/Turing$ ./if
^C
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/bhx programok/Turing$ gcc infy-w.c -o if
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/bhx programok/Turing$ ./if
[...]
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
19688	tundetuto	20	0	4372	812	748	R	99.0	0.0	0:07.37	./if
19155	tundetuto	20	0	2917M	197M	110M	S	12.5	0.4	2:36.09	/usr/lib/firefox/
13996	tundetuto	20	0	3436M	327M	149M	S	3.9	0.7	3:45.38	/usr/lib/firefox/
19253	tundetuto	20	0	2917M	197M	110M	S	3.3	0.4	0:20.29	/usr/lib/firefox/
14080	tundetuto	20	0	3436M	327M	149M	S	2.6	0.7	0:16.76	/usr/lib/firefox/
19233	tundetuto	20	0	2917M	197M	110M	S	2.6	0.4	0:19.33	/usr/lib/firefox/
19234	tundetuto	20	0	2917M	197M	110M	S	1.3	0.4	0:10.31	/usr/lib/firefox/
19639	tundetuto	20	0	35332	664	3688	*	1.3	0.0	0:01.87	htop
19473	tundetuto	20	0	2917M	197M	110M	S	1.3	0.4	0:05.39	/usr/lib/firefox/
2873	tundetuto	-6	0	1137M	1328	9876	S	1.3	0.0	1:46.50	/usr/bin/pulseaudio
19252	tundetuto	20	0	2917M	197M	110M	S	0.7	0.4	0:10.28	/usr/lib/firefox/
19244	tundetuto	20	0	2917M	197M	110M	S	0.7	0.4	0:06.44	/usr/lib/firefox/
1232	root	20	0	442M	118M	83168	S	0.7	0.2	20:07.33	/usr/lib/xorg/Xorg
19255	tundetuto	20	0	2917M	197M	110M	S	0.7	0.4	0:06.27	/usr/lib/firefox/

Az első végtelen ciklus egy processzor magot használ 100%-ban. Ez úgy lehetséges hogy itt egy while ciklust használunk ami folyamatosan vizsgálja a megadott feltételt, ehhez 1 CPU magot használ viszont annak a teljesítményét muszáj kihasználnia hogy folyamatosan figyelje a ciklus feltételét. Ugyanez a helyzet igazából egy végtelen for ciklus esetén is, megjegyezném, hogy az assembly kódjuk megegyezik.

Azért érdemes a `for (; ;)` hagyományos formát használni, mert ez minden C szabvánnyal lefordul, másrészről 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 infy-f.S infy-f.c
$ gcc -S -o infy-w.S infy-w.c
$ diff infy-w.S infy-f.S
1c1
< .file "infy-w.c"
---
> .file "infy-f.c"
```

Egy mag 0 százalékban:

A második ciklus egy cpu magot közel 0%-ban használ. Ehhez használtunk egy for ciklust, alapvetőleg mindenketől 1 magot használ 100%-ban. Ahhoz hogy a cpu mag 0%-ot használjon használnunk kell a `sleep(1)` függvényt ami a számolást késlelteti ezért CPU rá van kényszerülve a késleltetésre, viszont így sem tudjuk elérni hogy pontosan 0%-on menjen, inkább közelít a 0-hoz, a gyakorlat azt mutatja hogy így kb 1%-on megy. A képen egyébként ha megnézzük, az 5. CPU magnál látunk egy 0.0 értéket, tehát hozzá került a kérdéses ciklus.

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

    return 0;
}
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
19155	tundetuto	20	0	2917M	199M	110M	S	14.4	0.4	2:50.60	/usr/lib/firefox/
19253	tundetuto	20	0	2917M	199M	110M	S	3.3	0.4	0:23.05	/usr/lib/firefox/
13996	tundetuto	20	0	3436M	326M	148M	S	2.6	0.7	3:50.12	/usr/lib/firefox/
19233	tundetuto	20	0	2917M	199M	110M	S	2.6	0.4	0:21.78	/usr/lib/firefox/
14212	tundetuto	20	0	3341M	517M	165M	S	2.6	1.1	5:16.57	/usr/lib/firefox/
19639	tundetuto	20	0	35332	4924	3748	R	2.0	0.0	0:03.40	htop
14080	tundetuto	20	0	3436M	326M	148M	S	1.3	0.7	0:19.12	/usr/lib/firefox/
19473	tundetuto	20	0	2917M	199M	110M	S	1.3	0.4	0:06.57	/usr/lib/firefox/
19240	tundetuto	20	0	2917M	199M	110M	S	1.3	0.4	0:11.53	/usr/lib/firefox/
19234	tundetuto	20	0	2917M	199M	110M	S	1.3	0.4	0:11.53	/usr/lib/firefox/
19252	tundetuto	20	0	2917M	199M	110M	S	1.3	0.4	0:11.50	/usr/lib/firefox/
1232	root	20	0	442M	118M	83168	S	1.3	0.2	28:08.60	/usr/lib/xorg/Xor
19255	tundetuto	20	0	2917M	199M	110M	S	1.3	0.4	0:06.99	/usr/lib/firefox/
19244	tundetuto	20	0	2917M	199M	110M	S	0.7	0.4	0:07.16	/usr/lib/firefox/

Minden mag 100 százalékban:

A harmadik ciklusunk minden cpu magot 100%-osan használ. Ehhez szükségünk van egy for ciklusra és az omp.h nevű fájlra. Ennek segítségével párhuzamosan engedi használni egy feladatra a processzunk minden magját. Alapvetőleg ha egy ilyen ciklust futtatunk egy magot használ ki 100%-ban, a tesztjeim során azt vettettem észre, hogy éppen melyik magot használja az váltakozhat. Fordítani és futtatni egyébként a következőképpen tudjuk ezt a programot:

```
gcc -fopenmp filenev.c -o fajlnev
```

Nos így néz ki ha lefutott. Nagyon kíváncsi voltam egyébként ennek a programnak a futására, mivel 2 CPU-t használok és érdekelte hogy így is működik-e dolog, viszont ahogy láthatjuk műköött.

```
#include <omp.h>
int
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
19922	tundetuto	20	0	98M	912	816	R	1189	0.8	1:06.67	/vo
19924	tundetuto	20	0	98M	912	816	R	99.7	0.0	0:05.59	/vo
19927	tundetuto	20	0	98M	912	816	R	99.7	0.0	0:05.59	/vo
19929	tundetuto	20	0	98M	912	816	R	99.7	0.0	0:05.59	/vo
19931	tundetuto	20	0	98M	912	816	R	99.7	0.0	0:05.59	/vo
19928	tundetuto	20	0	98M	912	816	R	99.7	0.0	0:05.58	/vo
19930	tundetuto	20	0	98M	912	816	R	99.7	0.0	0:05.58	/vo
19932	tundetuto	20	0	98M	912	816	R	99.7	0.0	0:05.56	/vo
19923	tundetuto	20	0	98M	912	816	R	97.8	0.0	0:05.50	/vo
19925	tundetuto	20	0	98M	912	816	R	97.8	0.0	0:05.50	/vo
19926	tundetuto	20	0	98M	912	816	R	97.1	0.0	0:05.44	/vo
19933	tundetuto	20	0	98M	912	816	R	96.4	0.0	0:05.48	/vo
19155	tundetuto	20	0	2917M	199M	110M	S	5.9	0.4	3:11.03	/usr/lib/firefox/
13996	tundetuto	20	0	3468M	348M	162M	S	2.0	0.7	4:01.43	/usr/lib/firefox/

```
main ()  
{  
#pragma omp parallel  
{  
    for (;;) ;  
}  
    return 0;  
}
```

A **gcc infty-f.c -o infty-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 infty-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 videó:

A példában emlitett „Lefagy” függvénynek bármely programról el kell tudnia dönteni hogy tartalmaz-e végtelen ciklust. Ez nem feltétlen valósulhat meg, pl.: ha tartalmaz végtelen ciklust akkor "true" értékkel kell visszatérnie viszont ha nem tartalmaz akkor elindít egy végtelen ciklust. Ekkor ha a T1000 program „nem lefagyó” akkor mindenképp lefagy mert ha a „Lefagy” függvény nem "true" értékkel tér vissza elindítja a végtelen ciklust. Tehát ilyen esetben saját magáról nem tudja eldönteni hogy van-e benne végtelen ciklus avagy sem. Azaz gyakorlatilag egy paradoxon jelenséggel állunk szemben. Megjegyzés: Néhány

fejlesztői környezet jelzi ha a programunkban van végtelen ciklus de ezek is többnyire csak a legegyszerűbbekre tudnak szűrni szintaktika alapján pl.: ha a programunk tartalmaz `while(true)`, `while(1)`, vagy `for(; ;)`, ciklusokat. Amikor először próbáltam az első feladathoz írni ilyen ciklust nekem a "Geany" fejlesztői környezetben való futtatáskor néhány másodperc után kilépett a ciklusból.

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 épőlő `Lefagy2` már nem tartalmaz feltételezett, csak csak konkrét kódot:

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

    boolean Lefagy2(Program P)
    {
```

```
if (Lefagy (P))
    return true;
else
    for (;;) ;
}

main (Input Q)
{
    Lefagy2 (Q)
}

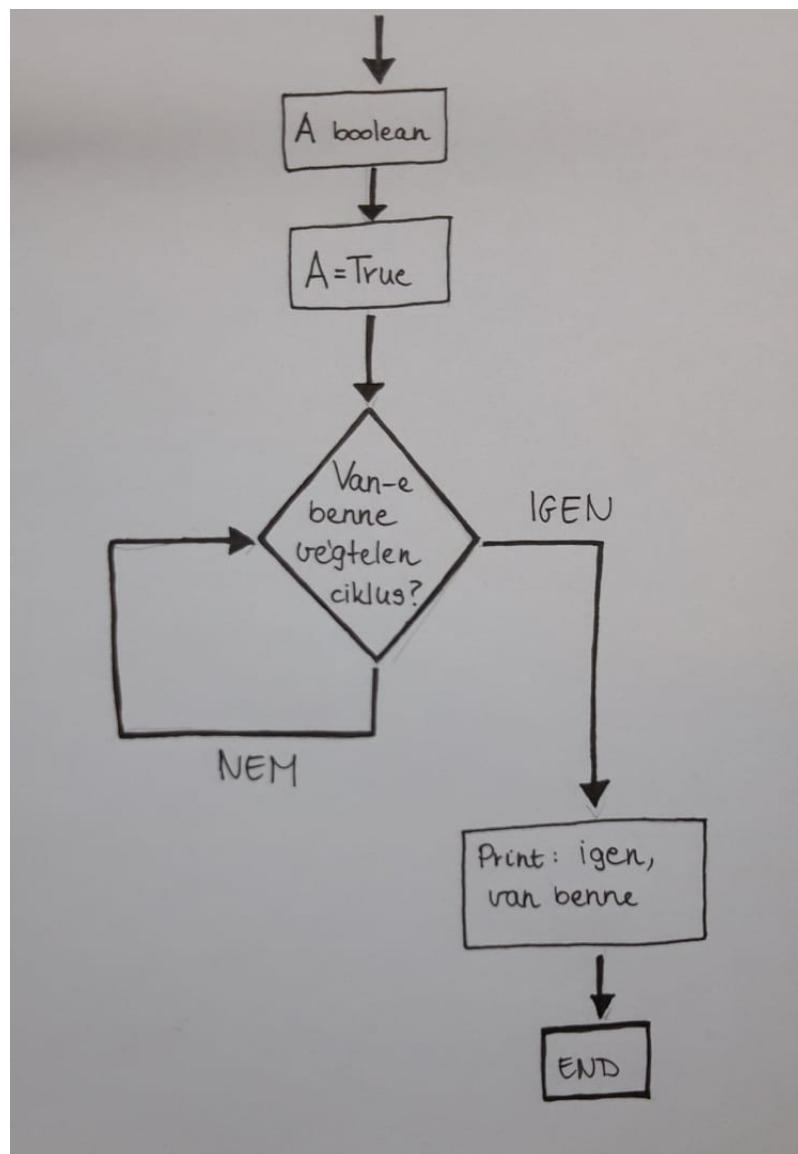
}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

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

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

A feladattal kapcsolatban készítettem egy folyamat ábrát. Ha nem igaz hogy tartalmaz végtelen ciklust, akkor visszatérünk újra ehhez az elágazáshoz, tehát lefagy a program. Amennyiben tartalmaz, kiírja tehát ellentmondásos.



2.3. Változók értékének felcserélése

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

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

Megoldás forrása: ugyanott.

Logikai utasítások nélkül a változók cseréjének legegyszerűbb módja segédváltozóval megcserélni az értékeket de egyéb megoldások is vannak pl.: az összeadásos vagy kivonásos. Logikai utasítással is meg tudjuk ezt tenni, a legjobb példa erre a bitenkénti kizáró vagy (XOR). Az én programom 3 fajta változócsereit tud végrehajtani, és minden futtatáskor véletlenszerűen választott módszerrel teszi ezt meg de nyílván az eredmény mindenkorban ugyanaz lesz.

```
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_tutorials/bhax_text
Fájl Szerkesztés Nézet Keresés Terminál Súgó
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_tutorials/bhax_text
book_IgyNeveldaProgramozod/Turing$ gcc valtozocsere.c -o vcs
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_tutorials/bhax_text
book_IgyNeveldaProgramozod/Turing$ ./vcs
1Minden futattasaskor egy random kivalasztott modeszert hasznalok.
Irj be 2 megcserelendo valtozot!
2 7
a = 2 es b = 7
A valtozokat az osszeadasos modszerrel cserelem meg.
Ekkor a valtozocsere utan: a = 7 es b = 2.
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_tutorials/bhax_text
book_IgyNeveldaProgramozod/Turing$ ./vcs
2Minden futattasaskor egy random kivalasztott modeszert hasznalok.
Irj be 2 megcserelendo valtozot!
6 8
a = 6 es b = 8
A valtozokat az kivonásos modszerrel cserelem meg.
Ekkor a valtozocsere utan: a = 8 es b = 6.
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_tutorials/bhax_text
book_IgyNeveldaProgramozod/Turing$ ./vcs
2Minden futattasaskor egy random kivalasztott modeszert hasznalok.
Irj be 2 megcserelendo valtozot!
18 3
a = 18 es b = 3
A valtozokat az kivonásos modszerrel cserelem meg.
Ekkor a valtozocsere utan: a = 3 es b = 18.
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_tutorialbhax_text
book_IgyNeveldaProgramozod/Turing$ ./vcs
0Minden futattasaskor egy random kivalasztott modeszert hasznalok.
Irj be 2 megcserelendo valtozot!
18 3
a = 18 es b = 3
A valtozokat segedvaltozo hasznalatalaval cserelem meg.
Ekkor a valtozocsere utan: a = 3 es b = 18.
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_tutorials/bhax_text
book_IgyNeveldaProgramozod/Turing$
```

Így néz ki a teljes program:

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

int main()
{
    srand(time(NULL));
    int r = rand() % 3;
    printf("%d", r);
    int a;
    int b;
    int seged;
    printf("Minden futattasaskor egy random kivalasztott modeszert ←
           hasznalok.\nIrj be 2 megcserelendo valtozot!\n");
    scanf("%d%d", &a, &b);
    if( r == 0)
    {
        printf("a = %d es b = % d\nA valtozokat segedvaltozo hasznalatalaval ←
               cserelem meg.", a, b);
        seged = a;
        a = b;
        b = seged;
        printf("\nEkkor a valtozocsere utan: a = %d es b = %d.\n", a, b);
    }
    else if( r == 1)
    {
```

```
printf("a = %d es b = % d\nA valtozokat az osszeadasos modszerrel ←
      cserelem meg.", a, b);
a = a + b;
b = a - b;
a = a - b;
printf("\nEkkor a valtozocsere utan: a = %d es b = %d.\n", a, b);
}
else
{
    printf("a = %d es b = % d\nA valtozokat az kivonasos modszerrel ←
          cserelem meg.", a, b);
    a = a - b;
    b = a + b;
    a = b - a;
    printf("\nEkkor a valtozocsere utan: a = %d es b = %d.\n", a, b);
}
return 0;
}
```

A segédváltozós módszer használata:

```
segéd = a;
a = b;
b = segéd;
```

A összeadásos módszer használata:

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

A kivonásos módszer használata:

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

2.4. Labdapattogás

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

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

Megoldás forrása: ugyanott.

A labdapattogató feladat lényege hogy a képernyőt úgy tudjuk használni mint egy koordináta rendszert (mivel az is). A kiválasztott karakterünket, akár az O betűt megadott pályán tudjuk, elindítani, itt fontos hogy kicsit elcsúsztatva kezdjük el a pattogtatást hogy be tudja járni az egész képernyőt. Ha pl x egyenlő

2, y egyenlő 2 értékeken indítjuk el akkor csak a képernyő egy átlóját tudja bejárni mivel csak az útvonalon fog oda-vissza pattogni. A képernyőnket mátrixként is lehet értelmezni.

Viszont nézzük meg a C programot! Itt az if-eket úgy kerüljük me, hogy for ciklusokat használunk az if helyett, mégpedig úgy, hogy nem adunk kezdőértéket a ciklusváltozónak és nem is növeljük azt, ez egy gyönyörű for ciklusnak álcázott if függvény. És itt van a teljes kód is:

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

//gcc bouncy0.c -o bouncy0 -lncurses

int main()
{
    WINDOW *ablak;
    ablak = initscr();

    int x = 0;
    int y = 0;

    int delX = 1;
    int delY = 1;

    int mx;
    int my;

    while(true)
    {
        printf("\033[0;32m");

        getmaxyx(ablak, my, mx);
        mvprintw(y, x, "O");

        refresh();
        usleep(100000);

        clear();

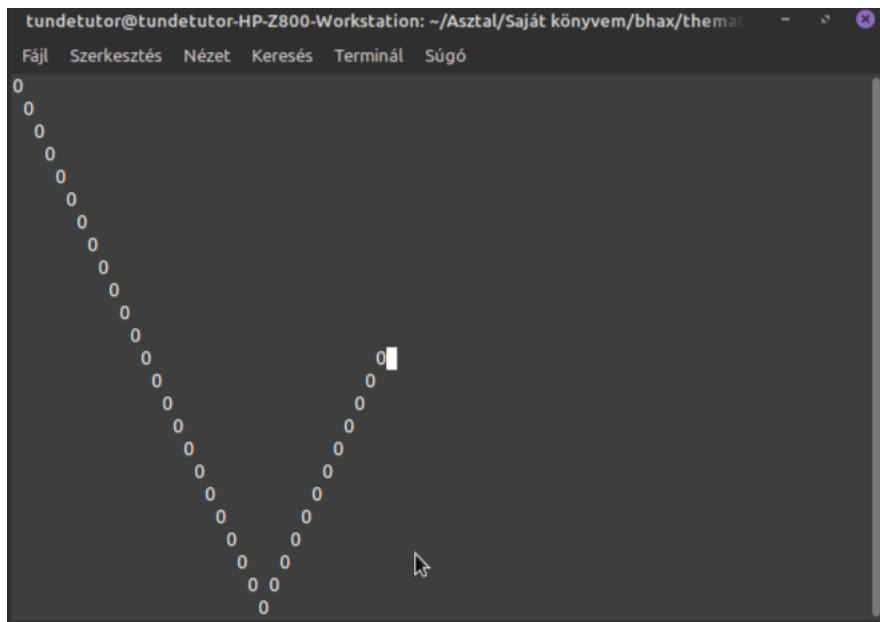
        x = x + delX;
        y = y + delY;

        for(;x <= 0;)
        {
            delX = delX * -1;
            break;
        }
        for(;x >= mx-1;)
        {
            delX = delX * -1;
        }
    }
}
```

```
        break;
    }
    for(;y <= 0;)
    {
        delY = delY * -1;
        break;
    }
    for(;y >= my-1;)
    {
        delY = delY * -1;
        break;
    }
}

return 0;
}
```

Képen annyira nem látszik mi történik, szóval a képhez kiszedtem a kódöt a clear-t így látszik szépen milyen utat jár be a labdánk.



Vizsont azt is tudjuk, hogy a for és while ciklusokkal ügyesen lehet variálgatni, szóval a for ciklusokat kicseréltem while ciklusokra és így is tökéletesen lefutott a program. Az alábbi kódcsipetben látható a while ciklusos megoldás is, illetve kommentben mellettük a for ciklus testvérei a while-jainknak.

```
while( !(x <=0) )           //for(;x <= 0;)
{
    delX = delX * -1;
    break;
}
while( !(x >= mx-1) )         //for(;x >= mx-1;)
{
    delX = delX * -1;
    break;
}
```

```
while( !(y <= 0) )           //for(;y <= 0;)
{
    delY = delY * -1;
    break;
}
while( !(y >= my-1) )        //for(;y >= my-1; )
{
    delY = delY * -1;
    break;
}
}
```

2.5. Szóhossz és Linus Tovald -félé BogoMIPS

A Linus Tovald féle BogoMIPS nevű program egy olyan program amivel a számítógépünk CPU-jának sebességét tudjuk "mérni". A "MIPS" jelentése kettős a névben, az első a mérés módszerére utal, a szám amit kiad a program ugyanis azt állapítja meg, hogy a processzorunk hány millió utasítást tud végrehajtani másodpercenként, ezért a MIPS-nek az első kibontása így szól: Millions of Instructions Per Seconds. Néhány Linux disztró esetén boot-olás során még mai napig kiírja ezt az értéket a gép, a hardware ellenőrzés során. Fontos megjegyeznünk, hogy ez egy "áltudományos mérés", a program névben erre a "Bogo" kifejezés utal. A "MIPS" másik jelentése: Meaningless Indication of Processor Speed. Na de nézzük meg az én gépemnek mennyi a BogoMIPS-e!

The screenshot shows a terminal window with the following text:

```
tundetutor@tundetutor-HP-Z800-Workstation: ~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ gcc bogomips.c -o bogomips
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ ./bogomips
Calibrating delay loop...ok - 816.00 BogoMIPS
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$
```

Láthatjuk, hogy ez egy 816-os érték. Megjegyezném, hogy itthon van mégégy hasonló workstation, abban jelenleg 2 db Xeon X5660-as processzort találunk, amikor még csak egy olyan volt benne akkor azon is végrehajtottam ezt a mérést és szintén 800 körül értéket kaptam. Az én gépem most szintén 2 db nagyon hasonló X5650-s processzorral van megáldva, szerintem nem ugyanaz a számítási kapacitás tehát találó a "Bogo" jelző. Viszont tértünk rá a szóhosszra!

Ennél a feladatnál a BogoMIPS program while ciklusának fejét kell felhasználnunk ami: **while(loops_per_sec <= 1)** formájú. Itt az történik a rövidke programon belül, hogy belül, hogy a bitshifttel, addig shifteljük

balra a biteket amíg el nem érünk a végéig. A **szo** változónk értéke most éppen 1, tehát áll valamennyi 0-ból a baloldalán, ha pedig jobb oldalról olvassuk akkor 1 db 1-essel kezdődik, ha megtudjuk mennyi 0 áll az 0-es előtt megtudjuk hány biten tárolódik, ugye ehhez kell a bitshift.

```
#include <stdio.h>

int main()
{
    int szohossz = 0, szo = 1;

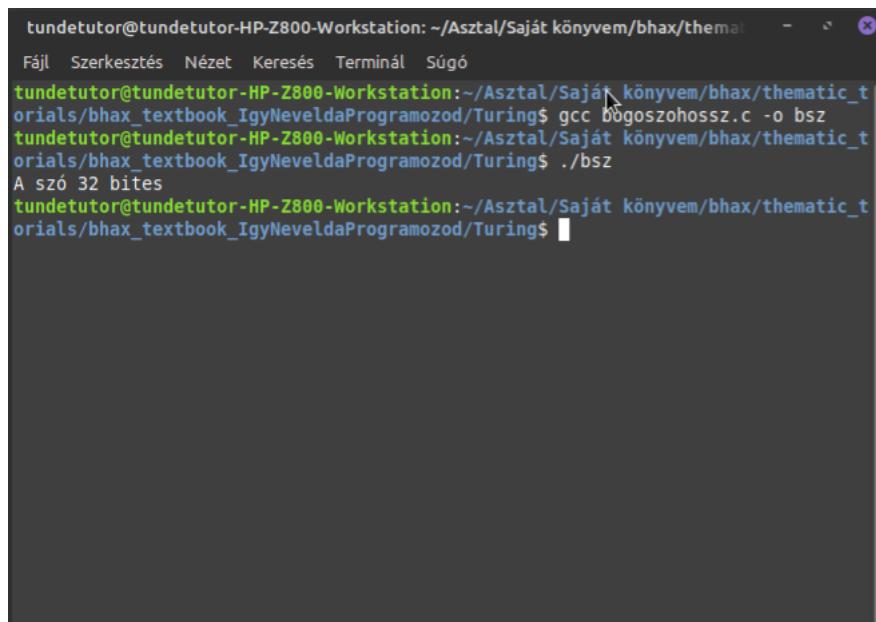
    do
    {

        szohossz++;

    } while(szo <= 1);

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

    return 0;
}
```



```
tundetutor@tundetutor-HP-Z800-Workstation: ~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ gcc b0goszohossz.c -o bsz
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ ./bsz
A szó 32 bites
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$
```

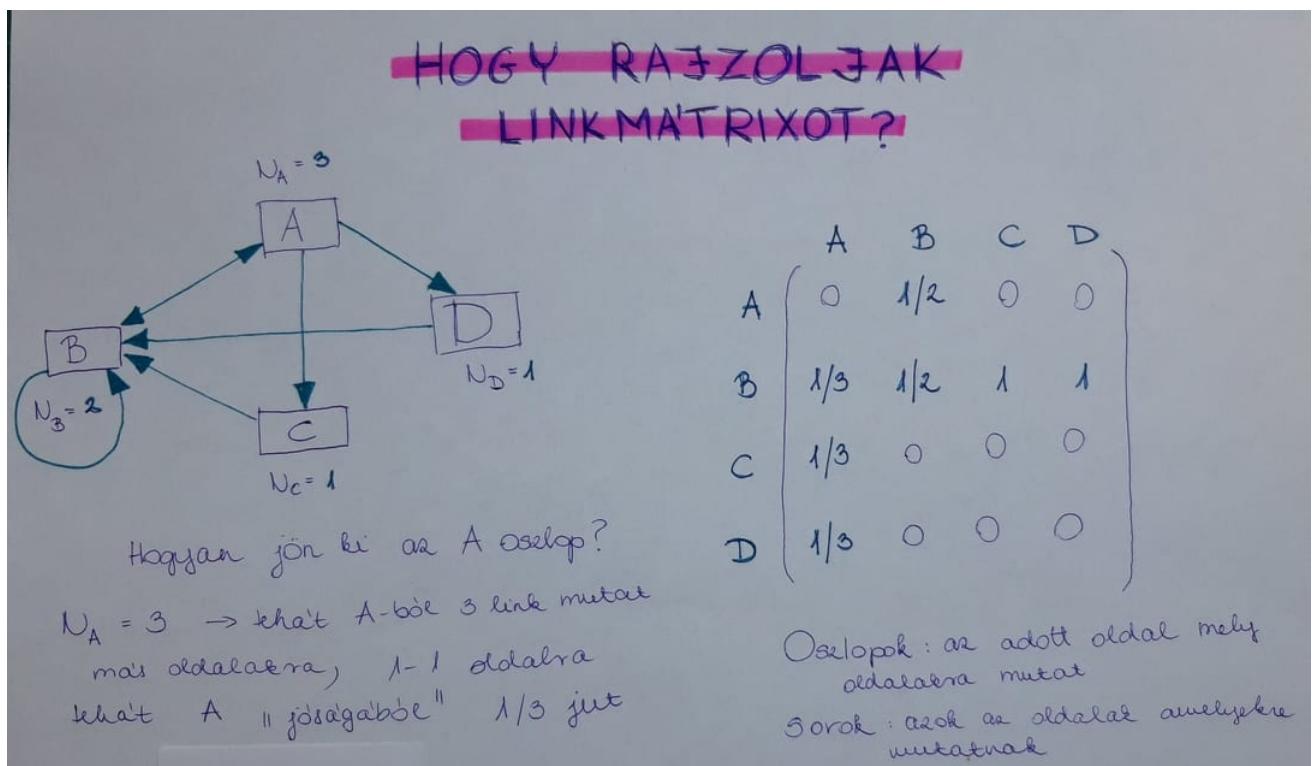
Fordítás és futtatás után megtudtuk, hogy 32 bites az 1-esünk. tehát áll 1 db 1-estől ami előtt tőle balra 31 db 0 szerepel.

2.6. Hello Google!

Ha csak nem más kereső motort használunk, akkor naponta nagyon sok alkalommal veszsük elő a jól ismert Google-t és keresünk rá valami kifejezésre. Amikor keresünk, reménykedünk benne, hogy a szükséges információ szerepelni fog az első pár találatban. De gondolkodtunk-e már rajta, hogy hogyan is talája meg a kereső motor a szémünkra leghasznosabb oldalakat?

Igazából ez is mint nagyon sok más dolog az emberi tényezőtől függ elsősorban, mégpedig ez most azt jelenti, hogy akik bizonyos weboldalakat szerkesztenek, mely más oldalakat linkelnek be a sajátukra. Az alapja az algoritmusnak, hogy sorba kell rendezni a találatokat, aszerint kell sorbarendezni őket, hogy az úgynevezett "PageRank" értékük mekkora, a PageRank pedig az adott weboldal minőségére utaló tulajdon-ság.

Ehhez a feladathoz én készítettem egy ábrát, az alapja a Bátfai Norbert által készített diasor, ezt be is linkelem a kép alá. Csak kicsit átírtam rajta a jelöléseket, picit könnyebben emészhetővé próbáltam alakítani.



Forrás: Bátfai Norbert diasora

Ez a rendszer gyakorlatilag egy 4 db honlapból álló gráf. A honlapokat nevezzük el A, B, C és D-nek. Mindegyikük rendelkezik egy "N" nevű tulajdonsággal, ami egy számmal írható le, ez a szám azt jelenti, hogy az adott oldalból hány db link fut ki. Az ábrán látható 4 honlap esetén $N(A)=3$, $N(B)=2$, $N(C)=1$ és $N(D)=1$. Belőük képeznünk kell egy linkmátrixot. A linkmátrix is fel van rajzolva, oszlopfoltonos módon a legegyszerűbb felrajzolnunk, most csak az A oszlopot írom le részletesen, utána a többi már magától értetődő lesz. Az A oszlop esetében nézzük az $N(A)$ értéket, ez ugye 3, tehát A PageRank értékét egyenlően el kell osztanunk azon 3 honlap között amelyekre mutat, most minden PageRank legyen először 1, ekkor a mátrixban A oszlopában A-hoz 0 kerül mivel magára nem mutat, a többi 3 (B, C és D) oldal esetében pedig 1/3-mmal kell számolnunk, tehát A a saját "jóságát" egyenlően osztja el.

A programban a main függvényen belül találjuk az elkészült linkmátrixot.

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

```
{0.0, 0.0, 1.0/3.0, 0.0}  
};  
  
double PR[4] = {0.0, 0.0, 0.0, 0.0};  
double PRv[4] = {1.0/4.0, 1.0/4.0, 1.0/4.0, 1.0/4.0};
```

Vegyük észre, hogy végtelen for ciklussal találkozunk, ugyanis ezeket a PageRank számításokat a szumma miatt a végtelenségig lehet számítani, minél többször fut le a ciklus annál pontosabb PageRank értéket kapunk a számítások végén. Ebből akkor tud kilépni, ha a **distance** kisebb a megadott 0.00001 értéknél, ami ugye egy nevezetes azonosság gyöke lesz, ekkor kapja meg a breaket a for ciklusunk.

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

Az itt található dupla for ciklusra azért van szükség, mert mátrix-szal dolgozunk.

```
for(int i = 0; i < 4; i++)  
{  
    double tmp = 0.0;  
    for(int j = 0; j < 4; j++)  
    {  
        tmp += L[i][j] * PR[j];  
    }  
    PRv[i] = tmp;  
}  
if(distance(PR, PRv, 4) < 0.00001)  
{  
    break;  
}  
}  
kiir (PR, 4);  
return 0;  
}
```

Itt láthatjuk a **distance** függvény működését. A végén gyökön kell vonnunk az összegből, nálam azért van ilyen furcsán megoldva mert elsőre nem jöttem rá hogy fordításhoz a **sqrt** függvény miatt kellene a -lm kapcsoló, aztán gondoltam 1/2-edik hatványra emelek így ez a kicsit matekos bonyolultabb megoldás maradt benne.

```
#include <stdlib.h>  
#include <math.h>  
  
double distance (double PR[], double PRv[], int db)  
{  
    float pwr = 0.5;  
    double osszeg = 0.0;  
    for(int i = 0; i < db; i++)
```

```
{  
    osszeg += (PRv[i] - PR[i]) * ( PRv[i] - PR[i] );  
}  
return pow(osszeg, pwr); //sqrt (osszeg);  
}
```

```
void kiir (double list[], int db)  
{  
    for(int i = 0; i < db; i++)  
    {  
        printf("PageRank [%d]: %lf\n", i, list[i]);  
    }  
}
```

Lefutattva a program a következő outputokat adja:

```
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/bhax programok/Turing  
Fájl Szerkesztés Nézet Keresés Terminál Súgó  
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/bhax programok/Turing$ gcc pagerank.c -o pr -lm  
pagerank.c: In function 'kiir':  
pagerank.c:18:9: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]  
    printf("PageRank [%d]: %lf\n", i, list[i]);  
          ^~~~~~  
pagerank.c:18:9: warning: incompatible implicit declaration of built-in function 'printf'  
pagerank.c:18:9: note: include '<stdio.h>' or provide a declaration of 'printf'  
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/bhax programok/Turing$ ./pr  
PageRank [0]: 0.090907  
PageRank [1]: 0.545460  
PageRank [2]: 0.272725  
PageRank [3]: 0.090907  
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/bhax programok/Turing$ gcc pagerank.c -o pr -lm  
pagerank.c: In function 'kiir':  
pagerank.c:18:9: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]  
    printf("PageRank [%d]: %lf\n", i, list[i]);  
          ^~~~~~  
pagerank.c:18:9: warning: incompatible implicit declaration of built-in function 'printf'  
pagerank.c:18:9: note: include '<stdio.h>' or provide a declaration of 'printf'  
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/bhax programok/Turing$ ./pr  
PageRank [0]: 0.090907  
PageRank [1]: 0.545460  
PageRank [2]: 0.272725  
PageRank [3]: 0.090907  
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/bhax programok/Turing$
```

2.7. Monty Hall probléma

A Monty Hall probléma egy amerikai TV műsorból ered, az alap szituáció az, hogy adott 3 db ajtó, melyek közül az egyik mögött nyereményt találunk. Választanunk kell ezek közül egyet, ez lesz a tippünk, 1/3, azaz 33,33% esélyünk van a megfelő ajtót kiválasztani igaz? Monty Hall, a műsorvezető viszont picit megkavarja a dolgokat. Felfedi mi van az egyik -szükségszerűen nem nyertes és nem általunk választott-ajtó mögött, majd ezután megkérdezi a játékost, hogy szeretné-e megváltoztatni a döntését. Itt van a csavar, a műsor nézői arra lettek figyelmesek, hogy a döntésüköt megváltoztató játékosok 2/3 része nyert. Ez akkor azt jelenti, hogy 1 db ajtóra, a 3 közül, 2/3 nyerési esély jut? Igen! Pontosan emiatt nevezhetjük a Monty Hall problémát matematikai paradoxonnak.

A feladat leírásában megtalálhatjuk Bátfai Norbert R nyelvű szimulációs programját. Ebben a programban hasonlóan az én 2. programomhoz, a gép sorsolja ki az eseteket, a nyertes és a választott ajtókat. Egyébként

mint aogy láthatjuk, alapból 1000000 esettel dolgozunk, de kisebb nagyobb esetszámokat is vizsgálhatunk a kódot átírva.

Mielőtt nekilátunk R programokat futtatni, ha még nem volt, akkor most telepítsük a szükséges dolgokat! A paracsok amiket használnunk kell:

```
sudo apt-get install r-base
```

Futtatni pedig így tudjuk az R nyelvű programokat:

```
Rscript filenec.r
```

A fő különbség az én kódомmal szemben, hogy az R nyelvű program előre generálja le az összes esetet, és ezeket vektorban tárolja majd később kiértékeli, míg az én programom legenerál egy esetet és egyből ki és értékeli.

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

for (i in 1:kiserletek_szama) {

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

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

  }else{

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

  }

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

}
```

Egy másik különbség, hogy ebben a programban kiszámoljuk hány esetben nyer a játékos ha változtat **length(valtoztatesnyer)** és ha nem változat a játékos **length(nemvaltoztatesnyer)**, míg az én programom csak azt mondja meg, hogy ha változtatunk a döntésen minden esetben akkor hány alkalommal lehet nyerni **X** esetből.

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

for (i in 1:kiserletek_szama) {

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

}
```

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

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

Mindkét program esetében megjelenik ez a 2/3 nyerési arány, tehát szerintem mind a két program pontosan dolgozik. Lássuk egy printsceent!

```
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/attention_raising/MontyH
Fájl Szerkesztés Nézet Keresés Terminál Súgó
[1] "Kiserletek szama: 1000"
[1] 328
[1] 672
[1] 0.4880952
[1] 1000
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/attention_raising/MontyH
[R]$ Rscript mh.r
[1] "Kiserletek szama: 1000"
[1] 355
[1] 645
[1] 0.5503876
[1] 1000
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/attention_raising/MontyH
[R]$ Rscript mh.r
[1] "Kiserletek szama: 1000"
[1] 326
[1] 674
[1] 0.4836795
[1] 1000
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/attention_raising/MontyH
[R]$ Rscript mh.r
[1] "Kiserletek szama: 1000000"
[1] 332982
[1] 667018
[1] 0.4992099
[1] 1000000
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/attention_raising/MontyH
[R]$
```

Mivel nekem ez a feladat különösen tetszett, ezért C nyelven is megvalósítottam, mégpedig 2 változatban. Azzal kezdem amelyikből kiindultam. Ebben a programban anyereményjátékot le lehet játszani, a felhasználó választhat ajtót, és "Monty Hall" pedig felfed egyet majd a felhasználó dönthet arról, hogy másikat választ-e. Nézzük meg a kódomat!

Mindössze egy main függvényből áll a kód, viszont így is elég hosszú, kb 280 sorról beszélünk, vázolom, hogy miért. Először is deklaráltam, a váltózóimat, kisorsolunk egy random számot, 0, 1 vagy 2 lehet az értéke, ez alapján van kisorsolva, hogy melyik a nyertes ajtó. A **valasztott** változóban pedig tároljuk melyik ajtót választja a felhasználónk. A továbbiakban a döntése alapján ágazik el a programunk először 3 fő ágra, ezek közül én az első esetet mutatom be, a többi is nagyon hasonlóan működik.

```
int main()
{
    int ajtol;
    int ajto2;
    int ajto3;
    int valasztott;
    int r;
    int valasz;
    int nyertesjatekok = 0;
```

```
srand(time(NULL));
r = rand() % 3;
//r = 0;

printf("Valassz egy ajtot!\n");
scanf("%d", &valasztott);
printf("\nMost felfedem az egyik ajtot: \n");
```

Tehát, az I. számú eset, amikor a random generálás során az 1-es ajtó lett a nyertes, ezt az első fő estet további 3 esetre kell bontanunk, a felhasználó ajtóbálasztásától függően. Daraboljuk a kódot ily módon!

A felhasználó az első nyertes ajtót választotta, felfedjük a 3. számú ajtót, amit ugye nem választott és nem is nyertes, majd rábízzuk, a döntést, hogy újat választ-e. ekkor még 2 esetre bomlik ez az eset is. Amikor eleve a nyertes ajtót választja a játékos az az eset amikor ha újat választ akkor veszít, az összes eset 1/3 része ugye.

```
if( r == 0) //I. eset, az első ajtó a nyertes
{
    ajto1 = 1;
    ajto2 = 0;
    ajto3 = 0;

    if( valasztott == 1) //I./1 eset, elve a nyertes ajtót ←
        valasztottuk
    {
        printf("A 3-mas szamu ajto nem nyertes.\nSzeretned a dontesed ←
            megvaltoztatni? (1 <-- Y/ 0 <-- N)\n");
        scanf("%d", &valasz);

        if(valasz == 1) //akkor az eleve nyertesként választott ajtó ←
            helyett egy vesztes ajtót választunk
        {
            valasztott = 2;
            printf("A valasztott ajtoban egy bena Porsce van. Porbald ←
                ujra!");
        }

        else if (valasz == 0) //maradunk a nyertes választásnál
        {
            printf("Nyertél! A nyeremenyed egy kecske!\n");
            nyertesjatekok = nyertesjatekok + 1;
        }
    }
    else
    {
        printf("Invalid bemenet");
    }
}
```

A második és 3 esetben nem nyertes ajtót választott a játékosunk, ha ilyenkor felfedjük a másik nem nyertes

ajtót akkor ha megváltoztatja a tippjét mindenképp nyerni fog, ez történik az **else if** és az **else** ágban is. Ugye így már látszik ez a 2/3-os arány? A II. és III. számú fő esetben is ugyanilyen szituáció alakul ki, tehát a nyerési esélyek csak ezzel a példával szemléltethetőek.

```
else if(valasztott == 2) //I./2 eset, elve a vesztes ajtót ←
    választottuk, az 1-es a nyertes
{
    printf("A 3-mas szamu ajto nem nyertes.\nSzeretned a dontesed ←
        megvaltoztatni? (1 <-- Y/ 0 <-- N)\n"); //felfedem a 3mas ←
        ajtót
    scanf("%d", &valasz);

    if(valasz == 1) //ekkor a korábban választott vesztes ajtó ←
        helyett egy nyertes ajtót választunk
    {
        valasztott = 1;
        printf("Nyertél! A nyeremenyed egy kecske!\n");
        nyertesjatekok = nyertesjatekok + 1;
    }

    else if (valasz == 0) //maradunk a nyertes választásnál
    {
        printf("A valasztott ajtoban egy bena Porsce van. Porbald ←
            ujra!");
    }
    else
    {
        printf("Invalid bemenet");
    }
}
else //I.3 vesztes ajót választottunk, a 3-masat
{
    printf("A 2-es szamu ajto nem nyertes.\nSzeretned a dontesed ←
        megvaltoztatni? (1 <-- Y/ 0 <-- N)\n");
    scanf("%d", &valasz);

    if(valasz == 1) //ekkor a korábban választott vesztes ajtó ←
        helyett egy nyertes ajtót választunk
    {
        valasztott = 1;
        printf("Nyertél! A nyeremenyed egy kecske!\n");
        nyertesjatekok = nyertesjatekok + 1;
    }

    else if (valasz == 0) //maradunk a nyertes választásnál
    {
        printf("A valasztott ajtoban egy bena Porsce van. Porbald ←
            ujra!");
    }
    else
```

```
{  
    printf("Invalid bemenet");  
}  
}  
}
```

Így néz ki a program ha futtatjuk, ez egy kis terminálos játék.

A terminal window titled "tundetutor@tundetutor-HP-Z800-Workstation: ~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing\$./mh". The session shows a game where the user is prompted to guess a number between 1 and 3. The user inputs 1, 0, and 1 respectively, and the program responds with "Nyertél! A nyereményed egy kecske!" (You won! Your prize is a dog!) for each correct guess. The user also sees the command "tundetutor@tundetutor-HP-Z800-Workstation: ~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing\$./mh" again at the end.

Viszont írtam egy olyan programot is ami úgy dolgozza fel az eseteket, mintha minden játék során a játékos új ajtót választana, az előző program tapasztalati és az R szimulációs program adta az ötletet ennek megvalósításához. Tehát elég minden ágban azt vizsgálni, hogy mi történik a döntés megváltoztatásakor. Nézzük a **main()** függvényt! Itt bekérünk egy vizsgálni kívánt esetszámot, és létrehozzuk a **nyertes** változót. A for cikluson belül meghívjuk a **mh(r, tipp)** függvényt, a kisorsolt nyerő ajtóval és a szintén random sorsolt választott ajtóval, ha a függvény visszatérési értéke 1, 1-gel növeljük a **nyertes** változót.

```
int main()  
{  
    int esetszam = 0;  
    int nyertes = 0;  
    int i;  
    int r;  
    int tipp;  
    srand(time(NULL));  
  
    printf("A program azt szimulalja, hogy minden esetben megvaltoztatjuk ←  
          az eredeti dontesunket.\n");  
    printf("Ird be az altalad vizsgalni kivánt esetszamot!\n");  
    scanf("%d", & esetszam);  
    printf("Esetszam: %d\n", esetszam);  
  
    for(i = 0; i < esetszam; i++)  
    {  
        //kisorsolom az ajtót  
    }  
}
```

```
r = rand() % 3;
//printf("r: %d\n", r);

//srand(time(NULL)); //kisorsolom a játékos tippjét
tipp = rand() % 3;
//printf("tipp: %d\n", tipp);

if( mh(r, tipp) == 1 )
{
    nyertes = nyertes + 1;
}
printf("Nyertesjatekok szama: %d", nyertes);
}
```

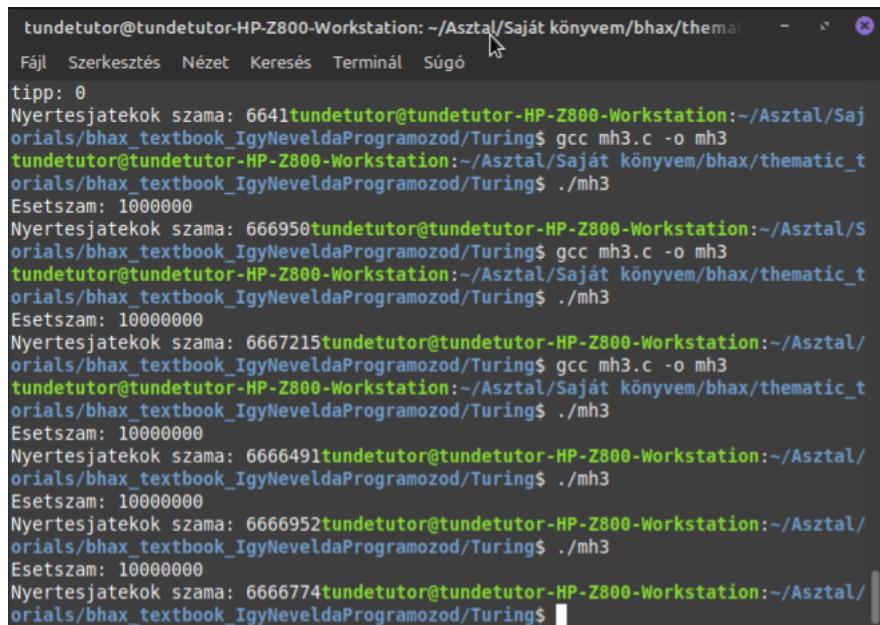
Itt pedig láthatjuk a **mh()** függvényt, az I., II. és III. fő esetekkel illetve annak aleseteivel dolgozik, tehát azzal, hogy melyik a nyertes ajtó és melyiket "választotta" a játékos. Ha az I/1.-es esetet vizsgáljuk, az 1. a nyertes ajtó és azt választotta a játékos, ha ekkor megváltoztatja a döntését -márpedig megváltoztatja- akkor nem nyert tehát a függvény 0 értékkel tér vissza.

```
int mh(int ajto, int valasztott)
{
    if( ajto == 0) //I. eset, az első ajtó a nyertes
    {
        if( valasztott == 0) //I./1 eset, elve a nyertes ajtót ←
            valasztottuk
        {
            return 0;
        }
        else if(valasztott == 1) //I./2 eset, elve a vesztes ajtót ←
            valasztottuk, az 1-es a nyertes
        {
            return 1;
        }
        else //I.3 vesztes ajót választottunk, a 3-masat
        {
            return 1;
        }
    }

    else if( ajto == 1)
    {
        if( valasztott == 0) //I./1 eset, elve vesztes ajtót valasztottuk
        {
            return 1;
        }
        else if(valasztott == 1) //I./2 eset, eleve a nyertes ajtót ←
            valasztottuk, az 1-es a nyertes
    }
}
```

```
{  
    return 0;  
}  
else //I.3 vesztes ajót választottunk, a 3-masat  
{  
    return 1;  
}  
}  
  
else  
{  
    if( valasztott == 0) //I./1 eset, elve a vesztes ajtót ←  
        választottuk  
{  
        return 1;  
}  
    else if(valasztott == 1) //I./2 eset, elve a vesztes ajtót ←  
        választottuk, az 1-es a nyertes  
{  
        return 1;  
}  
    else //I.3 nyertes ajót választottunk, a 3-masat  
{  
    return 0;  
}  
}  
}  
}
```

Az mh3.c nevű programom pedig így néz ki lefuttatva.



```
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ gcc mh3.c -o mh3  
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ ./mh3  
Esetszam: 1000000  
Nyertesjatekok szama: 66611tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ gcc mh3.c -o mh3  
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ ./mh3  
Esetszam: 10000000  
Nyertesjatekok szama: 666950tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ gcc mh3.c -o mh3  
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ ./mh3  
Esetszam: 10000000  
Nyertesjatekok szama: 6667215tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ gcc mh3.c -o mh3  
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ ./mh3  
Esetszam: 10000000  
Nyertesjatekok szama: 6666491tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ ./mh3  
Esetszam: 10000000  
Nyertesjatekok szama: 6666952tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ ./mh3  
Esetszam: 10000000  
Nyertesjatekok szama: 6666774tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Turing$ ./mh3  
Esetszam: 10000000
```

2.8. 100 éves a Brun Tétel

1919-ben Viggo Brun felfedezett egy speciális számot, egy konstanst, amit Brun-konstansnak emlegetünk. Mi teszi ezt a számot olyan különlegessé? Ahhoz, hogy ezt megértsük az alapjainál próbálon megragadni a témát.

A csak 1-gyel és önmagukkal osztható prímszámokról szerintem a könyv olvasója már biztosan hallott, általános iskolában is tanuljuk matematika órán, azt is tudjuk már, hogy a prímszámok halmazának számossága végtelen, viszont kevsebbet hallottak az ikerprímekről, vagy a négyes prímekről, de most az ikerprímeket vizsgáljuk meg átfogóbban. Az ikerprímek halmaza a prímszámok halmazának egy részhalmaza, olyan speciális számpárosokat tartalmaz, melyekben minden két szám prím és különbségük pontosan 2. Ilyen ikerprímek pl.: 3 és 5, 5 és 7 vagy a 17 és 19. Ezekkel kapcsolatban még nincs bizonyítva, hogy számuk véges vagy végtelen lenne, még ma is a matematika egyik nagy kérdése. Viszont ezekkel a különleges számokkal kapcsolatban amit már biztosan tudunk, hogy közük van a Brun-konstanshoz.

A Brun-konstans azért érdekes az ikerprímekkel kapcsolatban mivel ha sorra vesszük az ikerprímeink reciprokösszegeit észre kell vennünk, hogy egy számhoz közelítenek az értékek, a Brun konstanshoz, azaz ahhoz konvergálnak a reciprok összegek.

A feladathoz kaptunk egy R nyelven írt kódot, ez egy Bátfai Norbert által írt kód. A program egy grafikonon kirajzolja nekünk, hogy az ikerprímek reciprok összeg értéke, hogyan konvergál a Brun-Konstanshoz, ami megközelítőleg 1.90216, tehát valahol a 2 körül találunk meg a számegyenesen.

Mielőtt elemezzük és futattuk a kódot megfigyezzük, hogy bizonyos könyvtárak szükségesek a futtatásához. Én telepítettem már korábban egy R Studio-t ami felajánlotta a hiányzó könyvtárak telepítését, így nem kellett manuálisan megcsinálnom.

Itt látható a rövidke kód ami számol nekünk. Ha megfigyeljük, a program a **matlab** könyvtár hívásával kezdődik, erről írtam, hogy ennek a telepítése szükséges lesz a futtatáshoz. Ezt követően létrehozzuk az **stp** függvényt ami a számításokat fogja végezni.

Az **stp** függvényben először megkeressük a prímszámokat, majd a szomszédos prímek közti különbséget vizsgáljuk. Látható hogy az **idx** nevű változóba gyűjtjük azoknak a prímeknek az indexét, melyek között a különbség 2 volt, tehát ikerprímek. A megtalált ikerprímeknek ezután a reciprokát képezzük majd ezeket a számokat összeadjuk. Az **stp** függvény ezután az ikerprímek reciprokösszegények szummájával tér vissza. Ezzel megvizsgáltuk a program legfontosabb részét, már csak a grafikon kirajzoltatása van hátra.

```
library(matlab)

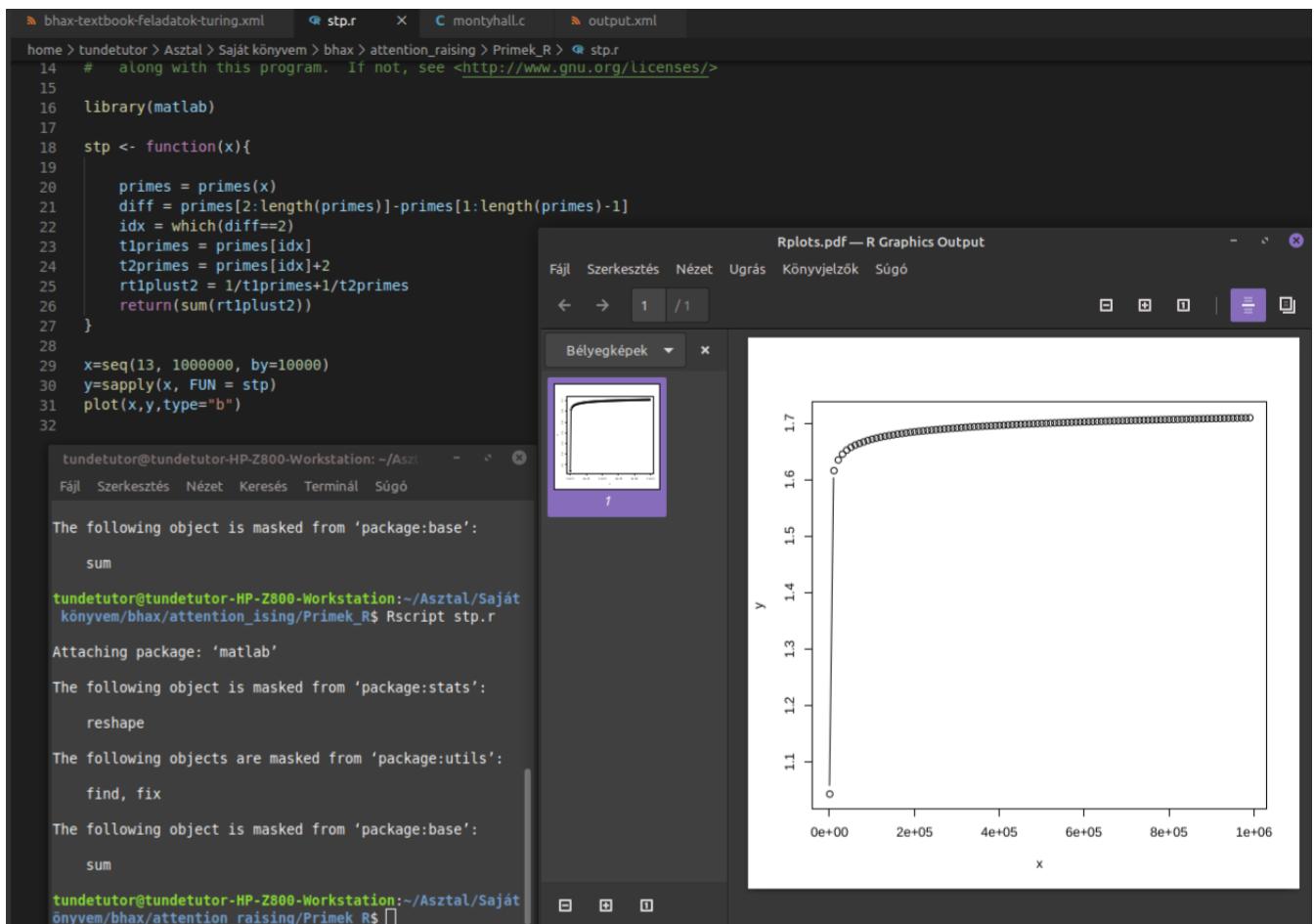
stp <- function(x) {

  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))
}
```

A kirajzoltatáshoz használunk egy **x**-et és egy **y**-t. Az **x**-ben egy sorozatot veszünk 13-tól egészen 1000000-ig, majd az **y** pedig megkapja az **stp** visszatérési értékét, ezzel megalkotva a koordináta rendszert. Aki használt már korábban Matlabot annak a **plot** ismerős lehet, matlabban ezzel a függvénytel tudjuk ábrázolni azokat amiket szeretnénk, és mivel meghívtuk a megfelő könyvtárat, most is ezt használhatjuk.

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

A futtatás után egy pdf-et kapunk amely ábrázolja hogy az ikerprímek reciprokösszegei hogyan konvergálnak a Brun-konstanshoz.



2.9. MALMÖ | folytonos csiga feladat

Ehhez a feladathoz 2 megoldásom társul. az egyik Python nyelven a másik pedig C++-ban. Viszont mind a kettő folytonos mozgási parancsokkal dolgozik.

Nézzük először a Python kódöt. Itt az történik, hogy Steve elkezd csiga vonalban egyre fentebb és fentebb menni az arénában, úgy hogy tőle a fal mindig balra van, tehát a sarkokban jobbra kell fordulnia. Én ebben a megoldásban 2 egymásba ágyazott for ciklust használtam, a külső 128-szor tudna maximum végrehajtóni,

ez cvsak azért maradt annyi mert egy nagy számot akartam írni, hogy ne hagyja abba a mozgását véletlenül sem. Ez a for ciklus azért felelős, hogy a szinteken fentebb tudjon menni.

A belső for ciklus minden alkalommal fut le, azért mert az arénának 4 oldala van és azt szeretnénk, hogy Steve minden szinten körbeérjen. Ebben a ciklusban minden **.87 * i** ideig végzi a mozgást, vagyis az előre haladását. Ez ennél a programnál egyfajta "varázsszám" volt, hogy a folytonos mozgási parancsokkal nagyjából megfelő hosszakat mozogjon szintenként. Ezután jobbra fordul egyet. Ha a belső for ciklus 4 alkalommal már lefutott, azaz Steve egy szintet bezárta, akkor ugrik egyet, hogy a következő szinten folytassa a mozgását.

```
def run(self):
    world_state = self.agent_host.getWorldState()
    i = 1
    j = 1
    # Loop until mission ends:
    while world_state.is_mission_running:
        print("--- nb4tf4i arena ----- \n ←")
        for i in range(128):
            for j in range(4):
                self.agent_host.sendCommand( "move 1" )
                time.sleep(.87 * i)
                self.agent_host.sendCommand( "turn 1" )
                time.sleep(.5)
                self.agent_host.sendCommand( "turn 0" )
                time.sleep(.5)
                self.agent_host.sendCommand( "jump 1" )
                time.sleep(.5)
                self.agent_host.sendCommand( "jump 0" )
                time.sleep(.5)

    world_state = self.agent_host.getWorldState()
```

Megoldás videó: Folytonos csiga python-ban

A videón látszik, hogy ugyan valóban csiga vonalban megy felfelé az arénában Steve, elég szerencsétlenül mozog. Ez a folytonos mozgásnak tudható be, ugyanis így nem tud olyan precízen mozogni mint diszkrét parancsokkal, ilyenkor a "turn 1" sem azt jelenti, hogy 90 fokkal forduljon jobbra hanem inkább "fordul jobbra valamerre".

A C++ verzió egy sokkal egyszerűbb kód, itt Steve csak forog és nem is az arénában van szemmel láthatóan. Itt annyit mondtunk Steve-nek, hogy 0.5 másodpercig menjen előre majd ugyanennyi ideig forduljon jobbra. Csak ezt a **do-while** ciklust vágtam ide, ugyanis itt a Steve-nek a mozgási parancsokat. Szerintem python-ban érdemesebb próbálkozni a Malmövel, C++-ban több hibalehetőségünk van.

```
do {
    agent_host.sendCommand("move 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds(500));

    agent_host.sendCommand("turn 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds(500));
```

```
world_state = agent_host.getWorldState();
for( boost::shared_ptr<TimestampedString> error : world_state.←
    errors )
    cout << "Error: " << error->text << endl;
} while (world_state.is_mission_running);
```

Megoldás videó: [Folytonos csiga C++-ban](#)

A videó 3 feladatmegoldást tartalmaz. ezek közül a kapcsolódó az első lesz. Megjegyzem, hogy a legoptimálisabb beállításokkal sem tudtam lerenderelni nem szemcsésre a videót, de szerintem a feladatok lényegi része így is látszik.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Először is mindenkorban be szeretném mutatni a decimális és az unáris számrendszeret. A decimális a jól ismert tízes számrendszer, melyben 10 számjegyből (0-9) tudunk leképezni a számokat. Általáos iskolából emlékezhetünk arra, hogy az ilyen tízes számrendszerbeli számokat oly módon írtuk fel hogy milyen számok vannak egyes helyiértékeken, pl.: egyes, tízes, százas, ezres. stb. Az adott helyiértéken álló számjegyet meg kell szoroznunk a helyiértékkel, majd ezekből összeget képzünk. Tehát pl.: a $321 = 3 \cdot 100 + 2 \cdot 10 + 1 \cdot 1$.

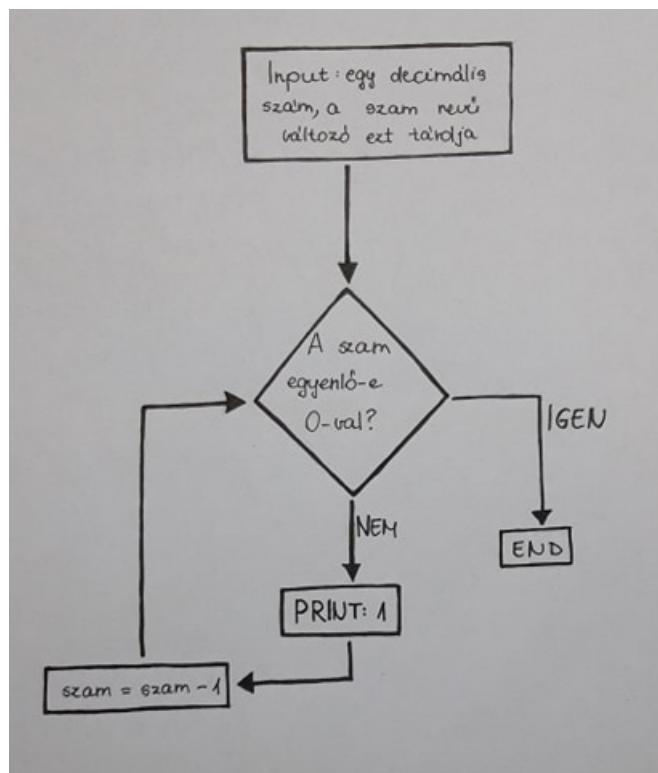
Az egyes számrendszer olyan számrendszer melyben a számokat 1 számjeggyel az 1-essel tudjuk leirni mégpedig oly módon, hogy amennyi maga a szám egymás után annyi egyest írunk pl.: $3 = 111$. A gép a két számrendszer között úgy vált át hogy a decimális számból addig von ki 1-eseket amíg az 0 nem lesz.

Ha ezt az átváltást program segítségével szeretnénk lemodellezni megoldás lehet ha mondjuk egy while ciklust használnunk amelynek kilépési feltétele hogy a decimális számuk 0 legyen. A ciklus törzsében minden lefutáskor levonunk a decimális számból egyet és egy 1-es számjegyet kiíratunk a kimenetre. Az én C nyelvű programom is iylen logika szerint működik így néz ki:

```
#include <stdio.h>

int main()
{
    int decszam;
    scanf("%d", &decszam);

    while( !(decszam == 0) )
    {
        printf("1");
        decszam--;
    }
}
```



Én az ábráimat kézzel szeretem rajzolni, szóval ide is egy ilyen ábra kerül. Ugye látható, hogy bekérünk egy tetszőleges decimális számot, ha az 0 a program leáll, amennyiben nem 0 kiír egy 1-est és csökkenti a számot 1-gyel majd ismét ellenőrzi a feltételt.

3.2. Hivatkozási nyelv

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

A BNF, azaz Backus-Naur-forma vagy Backus normálforma egy John backus által 1959-ben létrehozott környezetfüggetlen szintaxis, a leggyakoribb használata a programozási nyelvek leírása.

A C nyelvi utasítás fogalmának szintaktikai definíciója BNF szintaxisban:

```

<utasítás> ::= 
    <összetett_utasítás>
    <kifejezés>; (értékkedás pl, num=10)
    if(<kifejezés>) <utasítás>
    else if(<kifejezés>) <utasítás>
    else <utasítás>
    switch (<kifejezés>)
        <egész_konstans_kifejezés : <utasítás>
        goto <azonosító>;
        <azonosító> : <utasítás>
        break; continue; return<kifejezés>;
  
```

```
or(<kifejezés1><kifejezés2><kifejezés3>) <utasítás>
while(<kifejezés>) <utasítás>
do <utasítás> while<kifejezés>
; (üres utasítás, pl FORTRAN continue-ja)
```

A C nyelv 1989-ben jött létre és az első szabvány verziója a "ANSI C89" azaz amit mi a könyvben C89-nek hívunk. A C99-et 2000-ben mutatták be, azért 99 mert már 1999-ben gyakorlatilag létezett de hivatalosan 2000-ben vált az új C szabvánnyá. Olyan változtatások vannak a C99-es szabványban a C89-cel szemben mint a boolean típusú változók bevezetése és az egysoros kommenteket is C99-től tudunk írni "//"-rel. Egyébként a két verzió között voltak más C szabványok is de ezek azok amelyek a legismertebbek és a legtöbbek által használtak. Fontos megemlíteni hogy 2011-ben mutatták be a C11-et amiben a párhuzamos számításokat segítő funkciók jelentek meg az egyre jobban trjedő többmagos processzorok miatt.

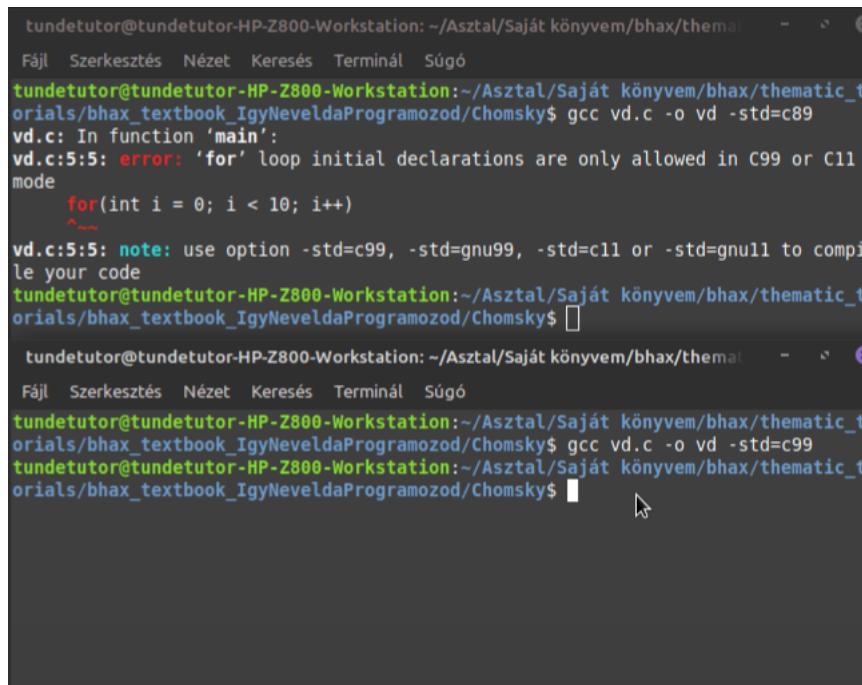
Példa olyan programra ami a C89-es szabvánnyal nem fordul de C99-cel igen:

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

Ebben a programban cikluson belül deklarálunk változót ami a C99-es szabványtól lehetséges. Ha C89-cel próbáljuk meg fordítani akkor a fordító javasolja is a C99 használatát. Egyébként nekem Arduino programozás során volt hasonló tapasztalatom, szintén cikluson belüli deklarációval próbálkoztam (sprólojunk egy sor kódot) viszont seholgy sem tudtam feltölteni a kódot az arduinora és a hibaüzenetben az szerepelt, hogy a ciklusváltozót nem dekláráltam. Az első gondolatom az volt mikor ránéztem a ciklusra, hogy lőtte fogom deklární a változót majd ezután sikeresen feltölteni a programot ami tökéletesen futott.

A képen 2 terminált tettem egymásra, hogy láthassunk, C89-cel nem fordul ellenben C99-cel igen. Ahhoz hogy különböző szabványokkal fordítsunk használjuk a következő kifejezést:

```
gcc vd.c -o vd -std=c89
```



```
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Chomsky$ gcc vd.c -o vd -std=c89
vd.c: In function 'main':
vd.c:5:5: error: 'for' loop initial declarations are only allowed in C99 or C11
mode
    for(int i = 0; i < 10; i++)
      ^
vd.c:5: note: use option -std=c99, -std=gnu99, -std=c11 or -std=gnull to compi
le your code
tundetutor@tundetutor-HP-Z800-Workstation:~/Asztal/Saját könyvem/bhax/thematic_torials/bhax_textbook_IgyNeveldaProgramozod/Chomsky$
```

3.3. Saját lexikális elemző

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

A lexer programok speciális szöveg elemző programok amivel szövegeket lehet tetszőleges szempontból elemezni illetve átalakítani. Ha még nem találkoztunk ilyenekkel akkor teleíptsük a függőségeit az alábbi módok valamelyikével:

```
sudo apt-get install flex-dev
```

```
sudo apt-get install flex-old
```

```
sudo apt install flex
```

A példaprogramunk lényege, hogy valós számokat keres a beadott szövegben, majd a szöveget átalakítva visszaadja. Ha az input pl.: "2teszt344lexer0.7" akkor azt kapjuk vissza hogy: "[realnum=2 2.000000]teszt[realnum=344.000000]lexer[realnum=0.7 0.700000]". Tehát elemzi és át is alakítja a szöveget.

A program maga három fő részre bontható amelyeket a "%%" kifejezések választanak el egymástól, eszerint fogom tagolni és elemezni is a kódöt. Nézzük az első részét.

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

Az első részben meghívjuk a szükséges header fájlt illetve deklaráljuk a **realnumbers** változót, ami számolja, hogy hány valós számot találtunk a szövegben.

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

Az második részben alkotjuk meg a szövegelemzés szabályait. Olyan számokat keresünk amelyek vagy tetszőleges számjegyből állnak vagy tetszőleges számjegyből 1 db pontból amit újabb tetszőleges számjegy követ. Ezután ha már megadtuk milyen mintára reagáljon a program definiáljuk, hogy hogyan reagáljon rá, azaz hogyan alakítsa át a szöveget, ezért a **printf()** függvény felel.

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

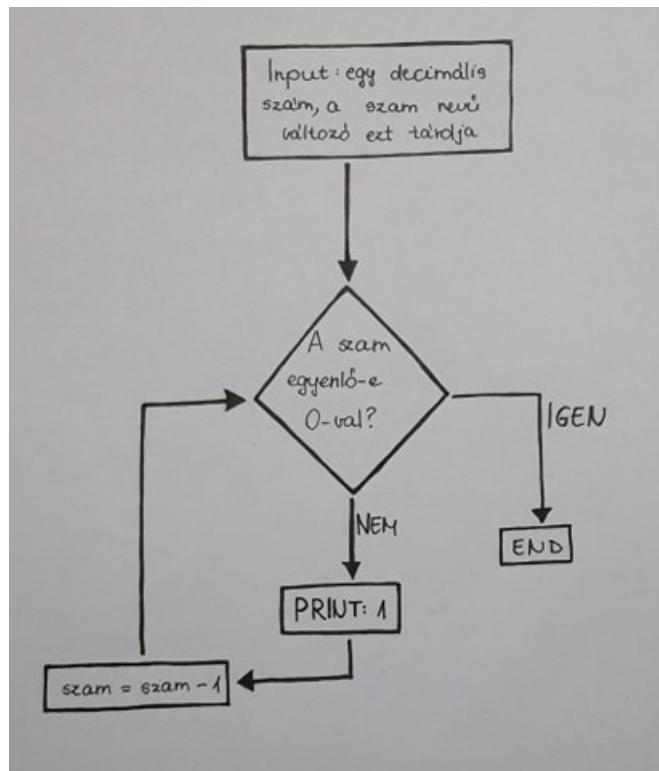
Az utolsó rész a fő programot tartalmazza, itt meghívjuk a lexert a **yylex ()** függvényel és egy kiíratást találunk ami a talált valós számok számát tartalmazó változó értékét írja ki. Viszont nézzük meg hogy néz ki amikor lefut a program!

A fordításhoz először használjuk a következő parancsokat:

```
lex -o realnumber.c realnumber.l
```

```
gcc realnumber.c -o realnumber -lfl
```

ekkora elkészült a futtatható állományunk.



Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/realnumber.1

3.4. Leetspeak

Lexelj össze egy l33t ciphert!

Az előző hasonlóan az ebben a feladatban tárgyalt pédagor program is egy lexer program, tehát szövegelemzésre és szövegek átalakítására szolgál. Ha az olvasó jártas az internet világában biztosan találkozott már azzal a jelenséggel, hogy betűket számokra cserélnek, ez gyakori például felhasználó nevek esetén, kapóra jön mivel lehet a kiválasztott név már foglalt viszont ha bizonyos karaktereket kinézetre hasonlóakra cserélünk nagyobb eséllyel szabad felhasználó nevet hozunk létre. Viszont korábban nem ilyen egyszerű dolgokra használták ezt a technikát ugyanis ez a titkosítás egyik kedvelt eszköze volt, ha így írunk le dolgokat a konkrét szavakra rákereső program a behelyettetsített számok miatt egy-egy kifejezést nem tud értelmezni, elszílik fölötté. Napjainkban szerveren való játék során is gyakran írnak így, hogy a sértő kifejezések miatt ne bannolják az illetőt.

A program maga úgy működik, hogy az angol ABC minden betűjéhez és minden számjegyhez tartozik 4 db ASCII karakter, tehát adott karakterhez társíthatunk betűt, számot vagy speciális karaktereket, a lényeg, hogy valamennyire hasonló legyen az eredetihez, annak érdekében, hogy az emberi szem számára még olvasható maradjon. Előfordul, hogy egy karakterhez nem lehet 4-et párosítani, hanem csak 2 hasonló van, ekkor kétszer párosítjuk az eredit egy másik karakterhez, mindenéppen egy karakternek 4 párja kell, hogy legyen. Ez a program is az előzőhöz hasonlóan 3 fő részre bontható, nézzük meg melyek ezek!

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

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

#define L337SIZE (sizeof l337d1c7 / sizeof (struct cipher))

struct cipher {
    char c;
    char *leet[4];
} l337d1c7 [] = {

{'a', {"4", "4", "@", "/-\\"}}, {'b', {"b", "8", "13", "|{}"}}, {'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", "|", "|_"}}, {'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
};
```

```
% }
```

Az első részben ismét meghívjuk a szükséges header fájlokat, illetve elkészül a chiper típusú tömb, ebben vannak összpárosítva az egyes karakterek a 4 db párjukkal. Ha figyelmesek vagyunk láthatjuk, hogy meghívjuk a time.h header-t ami majd random generáláshoz fog kelleni a kód második fő részében.

```
. {  
  
    int found = 0;  
    for(int i=0; i<L337SIZE; ++i)  
    {  
  
        if(1337d1c7[i].c == tolower(*yytext))  
        {  
  
            int r = 1+(int) (100.0*rand() / (RAND_MAX+1.0));  
  
            if(r<91)  
                printf("%s", 1337d1c7[i].leet[0]);  
            else if(r<95)  
                printf("%s", 1337d1c7[i].leet[1]);  
            else if(r<98)  
                printf("%s", 1337d1c7[i].leet[2]);  
            else  
                printf("%s", 1337d1c7[i].leet[3]);  
  
            found = 1;  
            break;  
        }  
  
    }  
  
    if(!found)  
        printf("%c", *yytext);  
  
}
```

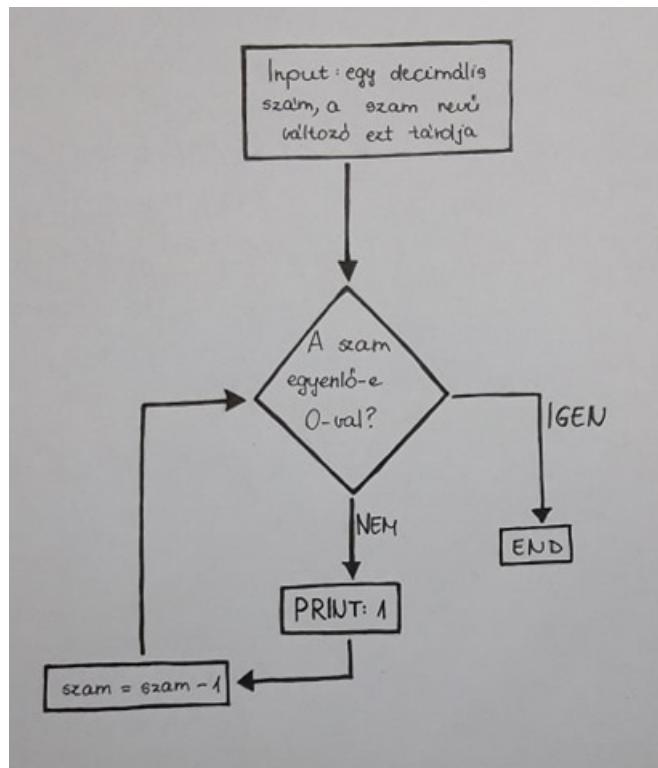
A program második része végzi a karakterek cseréjét. A csere egy random generáláson alapszik, ami a fő programban történik. Egy karakternek ugye 4 párra van, viszont el kell dönteni mikor helyettesítse a szövegbe, ehhez kell az 1 és 100 közötti random szám. Ha a random szám 91-nél kisebb akkor az első társított karaktert fogja behelyettesíteni, erre tehát jóval nagyobb esély van mint a többire, a másodikra 4%, a harmadika 3% az utolsó pedig 2%. Az utolsó dolog ebben a részben a kiíratás.

```
%%  
int  
main()  
{  
    srand(time(NULL)+getpid());  
    yylex();  
    return 0;
```

{}

Az utolsó rész a fő program, ismét mint az előző esetn meghívjuk a lexert a **yylex ()** függvénytel, illetve láthatjuk, hogy maga a random sorsolás itt történik.

Nézzük meg hogy hogyan néz ki a program lefuttatva! Ugyanúgy kell fordítani és futtatni mint az előző feladatban tárgyalt programot. Megjegyezném, hogy a program a kis és nagy betűket ugyanúgy felismeri.



Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/1337d1c7.1

3.5. A források olvasása

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?

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

Akkor és csak akkor kezelje a 'jelkezelo' függvény a SIGINT jelet, ha az eddig nem volt figyelmen kívül hagyva. Amennyiben figyelmen kívül van hagyva nem tudunk kilépni Crtl+C-vel a programból.

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

Egy for ciklus ami a négyeszer hajtja végre a hozzá rendelt utasításokat. Preorder módon először az i-t növeli és csak aztán végzi el az utasításokat.

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

Egy for ciklus ami a ötször hajtja végre a hozzá rendelt utasításokat. Postorder módon először elvégzi az utasításokat és csak azután növeli az i-t. Ugyanazt csinálja gyakorlatilag mint az előző ciklus.

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

Egy for ciklus ami a ötször hajtja végre a hozzá rendelt utasításokat és a tomb[] első öt értékét lecseréli az aktuális i értékre azaz a tömb első 5 eleme 0, 1, 2, 3, 4 lesznek.

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

Ebben az esetben egy for ciklusunk van aminek 2 feltétele lenne viszont a második igazából nem feltétel hanem utasítás, ezért hibás.

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

A standard outputra kiíratjuk az f() függvény visszatérési értékét, decimális számban.

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

A standard outputra kiíratjuk az f() függvény visszatérési értékét 'a'-ra és magát az 'a'-t is, decimális számban.

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

A standard outputra kiíratjuk az f() függvény visszatérési értékét 'a'-ra, aminek a memóriacímére mutat a mutató és magát az 'a'-t is, decimális számban.

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat...

3.6. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

Az előző félév során megismerkedhettem az ítéletlogika illetve az elsőrendű logikai nyelvekkel. Az elsőrendű logikai nyelvek között találhatunk nevezetes diszciplínákat, ilyen az Ar azaz az elemi aritmetikát leíró elsőrendű logikai nyelv vagy a Geom, vagyis az elemi geometriát leíró logikai nyelv.

Ahogy említettem az ar az elemi aritmetikát írja le minimális kifejezés használatával. A nyelv ABC-je csupán egy típusú változót, egy predikátumszimbólumot, három függvénszimbólumot és egy konsztnst használ. A változótípus természetes szám, a predikátum szimbólum kétváltozós, ez az egyenlőséget fejezi ki. A függvénszimbólumok között találunk egy egyváltozósat, ez a rákövetkező függvény, a másik kettő az összeadás és a szorzás pedig kétváltozósak. A nyelv egyetlen konstansa a 0. Mindenképp érdemes megemlíteni, hogy az Ar nyelv univerzuma a 0-val kiegészült természetes számok halmaza.

A nyelv csekély ABC-jét viszont mindig bővíthetjük, ugyan az előbb említett Ar nyelvi szavakkal leírható az elemi aritmetika minden fogalma érdemes a nagyon összetett kifejezésekhez új jelöléseket bevezetni.

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

- Bármely természetes számnál létezik nála nagyobb prímszám. A prímszámok száma végtelen.
- Bármely természetes szám esetén létezik nála nagyobb olyan prímszám melynek rákövetkezőjének rákövetkezője is prím. Tehát az ikerprímek száma végtelen.
- Létezik olyan természetes szám melynél bármely prímszám kisebb. A prímszámok száma véges.
- Létezik olyan természetes szám melynél bármely nála nagyobb szám nem prím. Azaz a prímszámok száma végtelen. A harmadik és a negyedik ekvivalens formulák votak.

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

3.7. Vörös pipaxs pokol/csiga diszkrét mozgási parancsokkal

A Turing feladatcsokorban hasonló feladattal találkoztunk de ott diszkrét helyett folytonos mozgást alkalmaztunk. A két mozgás között jelentős különbségek vannak, míg folytonos mozgásnál a "move 1" parancs mint kapcsoló működik ami miatt addig mozog amíg a "time.sleep()" engedi, addig diszkrét mozgásnál a "move 1" utasításként működik tehát 1 blokknyit mozog előre a "time.sleep()"-ben megadott idő alatt. Azt hogy csigában haladjon mi úgy oldottuk meg, hogy először az legalsó szinten lévő virágot kibányássza majd egy szinttel ugorjon fentebb és ezután kezdj el a csigát. A csigához 2 db for ciklust használtunk, a belső for ciklusban mindenkor az oldalnyi mozgáshoz elegendő lépésmennyiséget teszi a külső for ciklus pedig azt teszi lehetővé hogy adott szinten az adott oldalhosszot 4x tegye meg, ennek elején mindenkor fentebb ugrik egy szintet.

Aki először próbálkozik a Malmö-ben programozni, azt biztosan össze fogja zavarni, hogy 2 féle mozgást tud Steve végezni. Az előző feladatcsokor végén az abszolút mozgási parancsokkal programozott Steve-t mutattam be, most pedig a diszkrét parancsok használatát fogom.

Abszolút mozgás estén például a (**"move 1"**) parancs "kapcsolóként" viselkedik, tehát addig fog menni amíg azt nem mondjuk neki hogy álljon meg, azaz (**"move 0"**). Ez igazából a billentyűzetet próbálja szimulálni, így Steve úgy mozog mint amikor játszunk vele, addig megy amíg a billentyűzeten nyomjuk a gombot. Ezzel szemben diszkrét mozgás esetén a (**"move 1"**) parancs azt jelenti, hogy 1 egységet, a Minecraft világában egy "blokkot" haladjon előre.

Ennek megértésében Nagy Enikő segített nekem, ő magyarázta el mi a különbség a folytonos és a diszkrét mozgás között.

A feladathoz készített megoldás, egy összetettebb program, ezt a programot a könyvben más fejezetekben is megtalálható, viszont ott más szemszögből fogjuk vizsgálni. A program úgy működik, hogy Steve felszalad

a láváig, majd vissza, ezután az aréna aljából elindul csiga mozgással felfelá, a kód ezen részét mutatom be.

Python verzió

```
if poppy == 1:
    for i in range(4):
        self.agent_host.sendCommand( "turn 1" )
        time.sleep(.1)
    for j in range(move):

        if world_state. ←
            number_of_observations_since_last_state != 0:

                sensations = world_state.observations[-1].text
                print("    sensations: ", sensations)
                observations = json.loads(sensations)
                nbr3x3x3 = observations.get("nbr3x3", 0)
                print("    3x3x3 neighborhood of Steve: ", ←
                    nbr3x3x3)

                if "Yaw" in observations:
                    self.yaw = int(observations["Yaw"])
                if "Pitch" in observations:
                    self.pitch = int(observations["Pitch"])
                if "XPos" in observations:
                    self.x = int(observations["XPos"])
                if "ZPos" in observations:
                    self.z = int(observations["ZPos"])
                if "YPos" in observations:
                    self.y = int(observations["YPos"])

                print("    Steve's Coords: ", self.x, self.y, ←
                    self.z)
                print("    Steve's Yaw: ", self.yaw)
                print("    Steve's Pitch: ", self.pitch)

                if "LineOfSight" in observations:
                    LineOfSight = observations["LineOfSight"]
                    self.lookingat = LineOfSight["type"]
                    print("    Steve's <): ", self.lookingat)

if nbr3x3x3[12] == "red_flower" or nbr3x3x3[13] ←
    == "red_flower" or nbr3x3x3[14] == " ←
    red_flower":
    if nbr3x3x3[13] == "red_flower":
        self.agent_host.sendCommand( "move -1" ←
            )
```

```
        time.sleep(.1) #.5
        if nbr3x3x3[14] == "red_flower":
            self.agent_host.sendCommand( "move 0" )
            time.sleep(.5) #.5
            self.agent_host.sendCommand( "look 1" )
            #self.agent_host.sendCommand( "look .5" )
            self.agent_host.sendCommand( "move 0" )
            time.sleep(.5) #.5
            self.agent_host.sendCommand( "attack 1" )
            time.sleep(.5) #1
            self.agent_host.sendCommand( "look -.5" )
            #self.agent_host.sendCommand( "look -1" )
            self.agent_host.sendCommand( "jumpmove 1" )
            time.sleep(0.5)

world_state = self.agent_host.getWorldState()

if self.lookingat == "red_flower": #itt talalkozik ←
    a viraggal meg fel is veszi
    print("      VIRAAAG!")
    '''self.agent_host.sendCommand( "move 1" )
    time.sleep(.5)
    self.agent_host.sendCommand( "look 1" )
    self.agent_host.sendCommand( "move 0" )
    time.sleep(1.5)
    self.agent_host.sendCommand( "attack 1" )
    time.sleep(1.5)
    self.agent_host.sendCommand( "look -1" )
    self.agent_host.sendCommand( "jumpmove 1" )
    time.sleep(1)
    '''
#innen kezdodik a mozgasa
self.agent_host.sendCommand( "move 1" )
time.sleep(0.1)

world_state = self.agent_host.getWorldState()

self.agent_host.sendCommand( "jumpmove 1" )
time.sleep(0.5)
self.agent_host.sendCommand( "move 1" )
time.sleep(0.1)
move = move + 4
```

4. fejezet

Helló, Caesar!

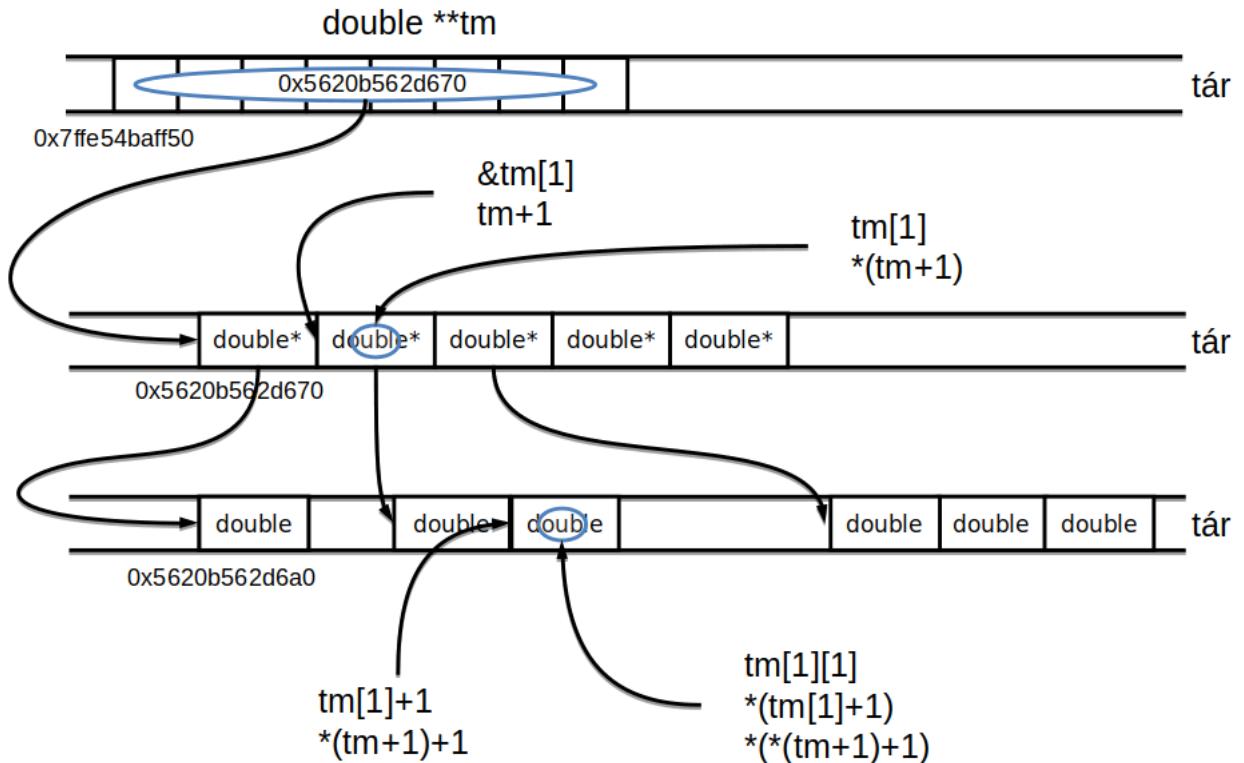
4.1. Double ** háromszögmátrix

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>.

A program működését a védés videómban is bemutatom sorról sorra, [itt megtekinthető](#).

A mátrixok körében találkozhatunk speciális mátrixokkal, például kvadratikus, azaz négyzetes mátrixokkal melyek sorai és oszlopai megegyező számúak. Viszont a négyzetes mátrixok között is találhatunk speciális mátrixokat, ezek a háromszögmátrixok. A háromszög mátrixok olyan mátrixok melyekben vagy a főátló alatt vagy a főátló fölött csupa 0 értékű elemeket találunk, az alsóháromszög mátrix esetében a főátló fölött szerepelnek ezek a 0 értékek. Fontos megemlíteni, hogy a programunk sorfolytonos mátrixot készít, tehát soronként balról jobbra haladunk! A programunk is ilyen mátrixot dolgoz fél, szóval nézzük is meg, hogy hogyan működik! Nézzük meg a Bátfai Norbert által készített ábrát!



4.1. ábra. A double ** háromszögmátrix a memóriában

Látjuk, hogy a 8 bájt méretű **double ***tm** mutató egy (ez esetben) 5 elemű mutatókból álló tömbre mutat amely 5 db **double *** mutatót tartalmaz, azaz ez 40 bájtot foglal a memóriából. Ezek a **double *** mutatók pedig már a mátrixunk egy-egy sorára mutatnak amelyek attól függően hányszorosan beszélünk tartalmaznak valamennyi **double**-t. Ez összesen 168 bájt, ha mindenöt össze számoljuk.

A forráskód elején meghívjuk a szükséges könyvtárakat, a **malloc()** használatához az **stdlib.h**-ra lesz szükségünk. Ezután a főprogramban deklarálunk egy **nr** nevű int típusú változót, ez adja meg a mátrix sorainak a számát, majd egy **double ***tm** mutatót.

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

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

Ezután lefoglaljuk a **double *** mutatók helyét egy **malloc()** függvényel, mégpedig úgy, hogy a sorok számával szorozzuk a **double *** méretét, hogy meglegyen amutatókból álló tömbünk helye. Majd egy for ciklussal lefoglaljuk a sorokat is. Itt arra kell figyelnünk, hogy minden sorban ezért minden i + 1 mennyiséggű double-nek kell helyet találni a szabad memóriában.

```
if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
{
```

```
    return -1;
}

printf("%p\n", tm);

for (int i = 0; i < nr; ++i)
{
    if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL -->
    )
    {
        return -1;
    }
}
```

Most hogy már megvan a megfelő mennyiségekű hely ideje feltölteni a mátrixunkat! Ezt kettő for ciklus segítségével oldjuk meg, a külső annyi alkalommal fut le ahány sora van a mátrixunknak a belső pedig minden annyiszor ahanyadik sorról van szó mivel az első sorba egy elem kerül a belső ciklus egyszer fut le, a második sorba kettő tehát kétszer fut le és így tovább. Az értékkedáshoz egy háromszögmátrixoknál használatos képletet adunk meg így az adott elem minden a mátrixban elfoglalt "sorszáma" lesz.

```
for (int i = 0; i < nr; ++i)
    for (int j = 0; j < i + 1; ++j)
        tm[i][j] = i * (i + 1) / 2 + j;
```

Most következik az elemek kiíratása. Ehhez az előbb használt for ciklus párosunkra lesz szükség, nemes egyszerűséggel kiíratjuk a megfelő elemeket, a külső for ciklus pedig a lefutása végén minden "kiír" egy sortörést, hogy szépen legyen ábrázolva a mátrix.

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

Az utolsó lépés a lefoglalt tárhely felszabadítása lesz. Most fordítva csináljuk mint a helyfoglalásnál, először a mutató tömböt szabadítjuk fel majd magát a **tm**-et, a **free()** függvény használatával.

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

free (tm);
```

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás forrása: egy részletes feldolgozása az [e.c](#) és [t.c](#) forrásoknak.

A titkosítás fogalma gyakran felbukkan. Az emberek évszázadok óta titkosítanak szövegeket különböző eljárásokkal, hogy csak azok tudják azt visszafekteni akik ismerik a titkosítás szabályát. Erre a legklasszikusabb példa amikor úgy írunk szövegeket, hogy egy betű helyett azt kell olvasnunk amely az ABC-ben előtte hárommal szerepel, ezt hívjuk Caesar titkosításnak. Az informatika világában pedig hatványozottan fontos a titkosítás, az ezzel foglalkozó területet kriptográfiának nevezzük. A példaprogramunk is szöveg titkosításra alkalmas program és itt is igaz, hogy az tud hozzáérni a titkosított információhoz aki ismeri a titkosítás szabályát. No de mi is az EXOR titkosító program szabálya?

A programunk egy kulccsal és egy titkosítandó szöveggel dolgozik, két részre fogom bontani a programot, hogy aszerint írjam le a működését. Az első részben meghívjuk a szükséges header fájlokat és definiáljuk a MAX_KULCS és a BUFFER_MERET konstansokat. A kulcsot és a titkosítandó szöveget karaktertömbként értelmezhetjük, illetve létrehozzuk a kulcs_index és olvasott_bajtok változókat.

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

#define MAX_KULCS 100
#define BUFFER_MERET 256

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

    char kulcs[MAX_KULCS];
    char buffer[BUFFER_MERET];

    int kulcs_index = 0;
    int olvasott_bajtok = 0;

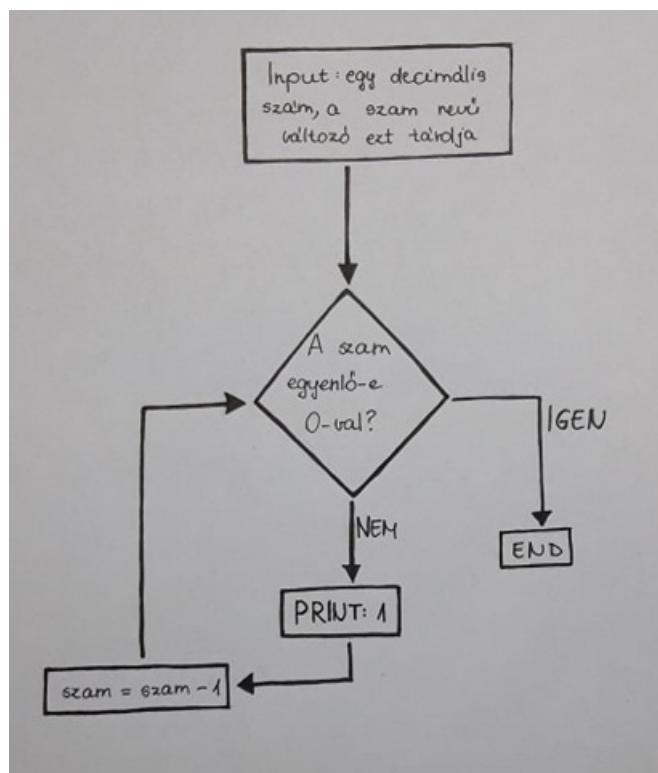
    int kulcs_meret = strlen (argv[1]);
    strncpy (kulcs, argv[1], MAX_KULCS);
```

A program második részében történik maga a titkosítás folyamata. Egy while ciklus végigmegy a bufferen és a while ciklusban lévő for ciklus pedig egyesével lépked a szöveg bájtjain. A buffer aktuális elemét, egy karaktert a szövegből ezután minden "összeexoroz" a kulcs aktuális karakterével. Ilyen az történik, hogy a kulcs adott bájtján és a szöveg adott bájtján elvégzi az EXOR azaz a kizáró vagy logikai műveletet, a C-ben ennek jele a ""). minden ilyen művelet elvégzése után léptet a program a kulcs aktuális karakterén. Ha a while ciklus lefutott készen van a titkosítás. Tehát az előbb említett titkosítás szabálya a kulcs megfelelő karakterével és a szöveg megfelelő karakterével történő kizáró vagy művelet elvégzése. Ha vissza szeretnék alakítani a szövegünket ugyanezt a programot kell futtatnunk azzal a kulccsal amivel titkosítottunk. Tehát itt a kulcs a titkosítás feltöréséhez szükséges indormáció.

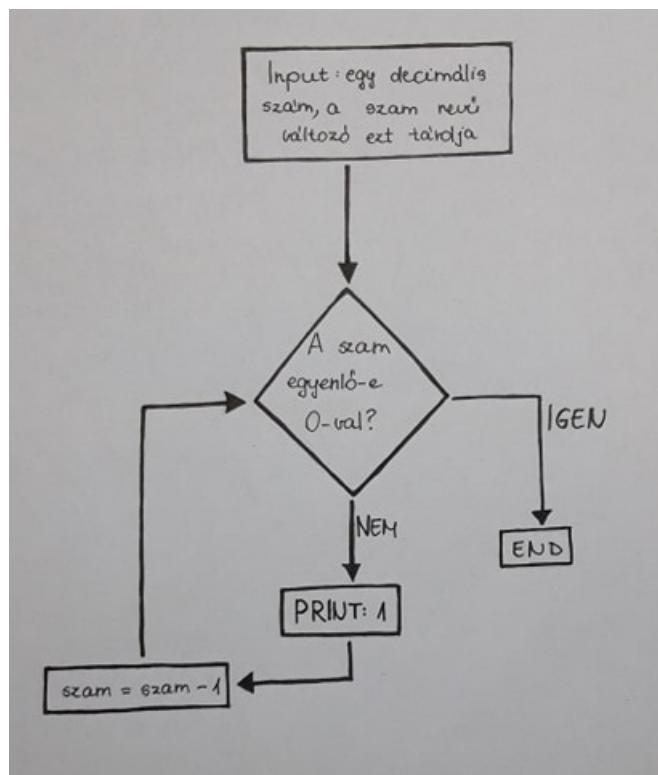
```
while ((olvasott_bajtok = read (0, (void *) buffer, BUFFER_MERET)))
{
    for (int i = 0; i < olvasott_bajtok; ++i)
```

```
{  
    buffer[i] = buffer[i] ^ kulcs[kulcs_index];  
    kulcs_index = (kulcs_index + 1) % kulcs_méret;  
  
}  
  
write (1, buffer, olvasott_bajtok);  
  
}  
}
```

Ha ez megtörtént a szövegünkben egy értelmezhetetlen bitsorozat keletkezik. Mivel a kizártó vagynak kiszámíthatatlan eredményei lehetnek ezért ha kiíratjuk a szöveget akkor nagyon ritka hogy ASCII karaktert találunk benne. A titkosított szöveg kinézete engem arra emlékeztett, hogy hogyan néz ki egy **cat** pranaccsal szövegként kiíratott **.blend**, azaz 3D-s fájl.



A könyvemből származó szöveg titkosítása a "maki" kulccsal.



Az előző szövegen lefuttatva a program ugyanazzal a kulccsal.

4.3. 4.3 Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

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

Mivel én még elég kezdőnek számítok a programozás világában nem igazán találkoztam ez előtt a Java nyelvvel ezért ehhez a feladathoz [Czanik András](#) segítségét kértem.

A Java objektum orientált nyelv, sajátossága, hogy mindenképp osztályokat kell használnunk a programok írásához ha ezt a nyelvet szeretnénk használni. Ennek vannak előnyei és hátrányai is. Az egyik hátrány szerintem akkor jelenik meg ha pici programokat szeretnénk írni, nézzünk egy példát a klasszikus Hello World program megvalósítására Javaban!

```
class Hello
{
public static void main (String args[])
{
System.out.println("Java Hello World");
}
```

Míg ugyanez a program Pythonban összesen egy sor, pedig minden esetben objektum orientált nyelvről beszélünk.

```
print("Hello, World!")
```

Ugyanakkor ha nagyobb, összetett programokat szretnénk írni nem véletlenül ajánlják a Javát, egyszerű fordítani és futtatni a kódokat, többek között emiatt használják sokan illetve pár éve a gazdasági szektorban is "industrial standard" programozási nyelv volt. Nézzük meg azonban a titkosító programot! Az első dolog amit megfigyelhetünk, hogy létrehozunk egy `main` nevű osztályt, szükségszerűen az osztály nevének egyeznie kell a program nevével. Az osztályban találunk egy `encode` függvényt, ez fogja végezni magát a titkosítást, ezt nem fogom bővebben részletezni, mivel teljesen ugyanúgy működik mint az előző feladatban tárgyalt program.

```
import java.io.InputStream;
import java.io.OutputStream;

public class main
{
    public static void encode (String key, InputStream in, OutputStream out ←
        ) throws java.io.IOException
    {
        byte[] kulcs = key.getBytes();
        byte[] buffer = new byte[256];
        int kulcsIndex = 0;
        int readBytes = 0;

        while((readBytes = in.read(buffer)) != -1)
        {
            for(int i=0; i<readBytes; i++)
            {
                buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]); //itt ←
                csinálja az összeEXOR-ozást
                kulcsIndex = (kulcsIndex + 1) % key.length();
            }

            out.write(inputBuffer, 0, readBytes);
        }
    }
}
```

A program második részében találjuk a `main` nevű osztálykonstruktort. Ebben a részben főleg hibakeresést, találunk, a `try` részben figyeli, hogy érkezik-e bemenet, azaz `kulcs`, mivel az az első argumentum, ha nem akkor a program leáll, és hibaüzenetet kapunk vissza.

```
public static main (String[] args)
{
    if(args[0] != "")
    {
        try
        {
            encode(args[0], System.in, System.out); //
        }
    }
}
```

```
        catch(java.io.IOException e)
        {
            e.printStackTrace();
        }
    }
else
{
    System.out.println("Please provide a key!");
    System.out.println("java main <key>");
}
}
```

4.4. EXOR törő

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

Megoldás videó:

Megoldás forrása:

Adott volt az előző feladatban a t.c program ami arra való hogy ugye XOR titkosított szövegeket törjünk vele. Labor feladat volt ezt kivitelezni is, amit sikerült is megoldanom pár esetben. Nagyon felbosszantott hogy nem tudtam mindenki szövegét törni majd utána jártam a dolognak hogy mi volt a baj...

Ha az előző feladatokban már sikerült egy titkosított szöveget előállítanunk akkor felmerülhet bennünk a kérdés, hogy a kulcs ismeretének hiányában hogyan kaphatjuk vissza az eredeti szöveget. Az ötlet az lenne, hogy próbáljuk ki minden lehetséges kulcsot, az ilyen törési technikát "brute force"nak nevezzük, azaz "nyers erő". Ez egy nagyon gyakori törési technika és például a különböző felhasználó fiókok ettől általában védeve is vannak a próbálkozási lehetőségek korlátozásával, vagy esetleg valamilyen más védelemmel. A jelszavainkat ezért is érdemes több típusú karakterből összeollózni mivel nagyobb védelmet biztosít, több karakter kell a programnak végigpróbálnia. Azokat a karaktereket amelyekkel próbálkozni akarunk egy tömbbe tesszük, ez lesz a karakterkészletünk. Én a Bátfai Norbert által írt programot az egyik laboron kiírt feladathoz módosítottam azaz a karakterkészletet lényegesen lecsökkentettem egészen pontosan 4 karakterre.

A program első részében definiáljuk a konstansainkat, azaz a kulcsnéretet és a buffer méretét, illetve meg-hívjuk a szükséges könyvtárakat.

```
#define MAX_TITKOS 4096
#define OLVASAS_BUFFER 256
#define KULCS_MERET 4
#define _GNU_SOURCE

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

```
}
```

A forráskódban találnunk egy `void exor` névvel ellátott eljárás, ez végzi az előző feladatokban tárgyalt titkosítás a bitenkénti kizárá vagy művelettel.

```
void  
exor (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;  
    }  
  
}
```

Viszont, nem elég végigzongoráznunk a kulcsokat, hiszen a program magától nem tudja, hogy egy exorral mikor sikerült megtörnie a szöveget, ezért a kulcsok próbálgatásának a leállítását valamihez hozzá kell kötnünk. Itt jön be a képbe a `atlagos_szohossz` és a `tiszta_lehet` függvénypáros.

Az `atlagos_szohossz` végigmegy a titkos szövegen és számolja benne a szóközök számát majd a `titkos_meret`-et azaz a bemenet összes karakterének számát osztjuk a szóközök számával, azaz a vélt szavak számával. Ebből kapjuk meg a függvény doble típusú visszatérési értékét.

```
double  
atlagos_szohossz (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;  
}
```

A `tiszta_lehet` függvény azt vizsgálja, hogy a szövegen megjelenik-e négy gyakori magyar szó amelyek a "hogy", "nem", "az" és a "ha". Amennyiben az átlagos szóhossz megfelel a kritériumnak, azaz 6 és 9 között van ami az átlag magyar szóhossz és tartalmazza a gyakori magyar szavakat akkor jó eséllyel sikeres a programnak megtalálnia a kulcsot.

```
int  
tiszta_lehet (const char *titkos, int titkos_meret)  
{  
    // A tiszta szöveg valszeg tartalmazza a gyakori magyar szavakat
```

```
// illetve az átlagos szóhossz vizsgálatával csökkentjük a
// potenciális töréseket

double szohossz = atlagos_szohossz (titkos, titkos_meret);

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

}
```

A `exor_tores` meghívja az `exor-t` és a tiszta lehet értékével tér vissza.

```
int
exor_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);
}
```

Már csak a fő programunk maradt ki a sorból. Az egyik legfontosabb teendőnk a megfelelő karakterek kiválasztása amelyekből kulcsot tudunk generálni. Én úgy módosítottam a programot, hogy ugye 4 karakteres kulcsot tudjon törni és az ABC-met az előző feladatokban használt kulcschoz igazítottam. Egy while ciklussal beolvassuk a titkosított szöveget ezután ha még maradt hely a bufferban akkor egy for ciklussal azt kinullázzuk. Ezután történik a kulcsok generálása, azaz ebben az esetben négy for ciklussal végigmegyünk a karakterkészlet minden elemén, esetünkben ez maximum 256 kombinációt jelent. Ha ellőállt egy kulcs akkor meghívjuk az `exor_tores-t` ami a `tiszta_lehet` értékével tér vissza. Ha igaz a visszatérési értéke a program kiírja a kulcsot és a tiszta szöveget, ha hamis akkor újra exorozunk, hogy az eredeti titkos szöveggel lehessen tovább dolgozni és ne kelljen másik buffert hazsnálni.

```
int
main (void)
{

    char kulcs[KULCS_MERET];
    char titkos[MAX_TITKOS];
    char betuk[5] = {'m', 'k', 'a', 'i', '\0'};
    char *p = titkos;
    int olvasott_bajtok;

    // titkos fajt berántasa
    while ((olvasott_bajtok =
        read (0, (void *) p,
        (p - titkos + OLVASAS_BUFFER <
        MAX_TITKOS) ? OLVASAS_BUFFER : titkos + MAX_TITKOS - p)))
        p += olvasott_bajtok;
```

```
// maradek hely nullazasa a titkos bufferben
for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
    titkos[p - titkos + i] = '\0';

// osszes kulcs eloallitasa
for (int ii = 0; ii <= 4; ++ii)
    for (int ji = 0; ji <= 4; ++ji)
        for (int ki = 0; ki <= 4; ++ki)
            for (int pi = 0; pi <= 4; ++pi)
            {
                kulcs[0] = betuk[ii];
                kulcs[1] = betuk[ji];
                kulcs[2] = betuk[ki];
                kulcs[3] = betuk[pi];

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

                // ujra EXOR-ozunk, ily nem kell egy masodik buffer
                exor (kulcs, KULCS_MERET, titkos, p - titkos);
            }

            return 0;
}
```

4.5. 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

Tanulságok, tapasztalatok, magyarázat...

Egy nagyon jó szemléltető videó

A neurális háló egy nagyon érdekes de nem könnyű téma. Itt az első bekezdésben röviden bemutatnám miről is van szó. Egy neurális háló 2 főbb dolgóból áll: neuronokból és rétegekből. A neuronok lényege hogy a kapott adatokat továbbítani vagy módosítani legyenek képesek. A neuronoknak van egy bemeneti oldala, itt a bejövő adatokat szummázza illetve van egy kimeneti oldala. A valódi idegsjekben a "bemeneti" adatok axonokon keresztül érkeznek, az axonok által összekötött idegsejtek pedig a neuron hálózatot alakítják ki, ehhez hasonlóan működnek a programozásbeli neurális halózatok is, ezeket gráffal tudjuk szemléltetni. A neuronok tehát adatokat kapnak és azokat továbbítják. Ez a kapcsolat alakítja ki a rétegeket. 3 fő rétegből szoktak állni a neurális hálózatok, ezek a bemeneti, a rejttett és a kimeneti réteg. Rejtett rétegből lehet több, vagy akár egy sem, minél több rejttett rétegből áll egy hálózat annál komplexebb feladatokat

képes végrehajtani. A rejttett réteget lehet műveleti rétegnek is nevezni, ahogy ezt sejteni lehet itt a kapott adatokkal a neuronok műveleteket végeznek, ezek leggyakrabban a legalapabb logikai műveletek, azaz OR, AND, XOR stb. Léteznek úgy nevezett deep learning neural network-ök, ezeknek a lényege az hogy a neurális hálózatunkhoz, egy "reward system"-et azaz jutalmazási rendszert adunk, ezek a hálózatok sok lefutás, generáción keresztül tanulnak, a folyamatosan érkező adatokat feldolgozzák. Minél pontosabb reward systemet alakítunk ki annál hatékonyabb lesz a tanulási folyamata.

Na de tértünk is rá a neurális OR, AND és XOR kapukra. Ezekhez 2 db input szükséges, az egyik lgyen A a másik pedig B. Az input módosítás nélül adja tovább az adatokat. Az OR és az And esetén nincs rejttett rétegünk, ezek elemi logikai műveletek. Ahhoz hogy a neurális hálónkat ezek elvégzésére megtanítsuk egy `data.frame` táblára van szükségünk, iletve a `neuralnet()` függvényre. Ez azt jelenti, hogy a `data.frame`-ben megadjuk neki az inputokat és a művelet elvégzése utáni eredményt, példát mutatunk be.

Az AND és az OR műveleteket, rejttett réteg bőlkül is el tudja végezni, akár a kettőt is egyszerre, de az XOR, azaz a kizáró vagy esetén más a helyzet. Ha ítéletlogikai formulaként felírjuk láthatjuk hogy egy sokkal összetettebb formuláról van szó, míg az AND egy elemi konjunkciónak felel meg az OR pedig ugye elemi diszjunkcióknak. Az XOR-hoz 3 rejttett rétegre lesz szükségünk.

A program kimenete:

```
$neurons
$neurons[[1]]
  a1 a2
[1,] 1  0  0
[2,] 1  1  0
[3,] 1  0  1
[4,] 1  1  1

$net.result
[,1]
[1,] 0.001417152
[2,] 0.999410346
[3,] 0.999027936
[4,] 0.999999999

dev.new(): using pdf(file="Rplots1.pdf")
$neurons
$neurons[[1]]
  a1 a2
[1,] 1  0  0
[2,] 1  1  0
[3,] 1  0  1
[4,] 1  1  1

$net.result
[,1]      [,2]
[1,] 1.055632e-07 2.947572e-09
[2,] 1.000000e+00 1.324911e-03
[3,] 9.999999e-01 1.366027e-03
```

```
[4,] 1.000000e+00 9.983784e-01

dev.new(): using pdf(file="Rplots2.pdf")
$neurons
$neurons[[1]]
    a1 a2
[1,] 1 0 0
[2,] 1 1 0
[3,] 1 0 1
[4,] 1 1 1

$net.result
[,1]
[1,] 0.4999967
[2,] 0.4999991
[3,] 0.4999993
[4,] 0.5000017

dev.new(): using pdf(file="Rplots3.pdf")
$neurons
$neurons[[1]]
    a1 a2
[1,] 1 0 0
[2,] 1 1 0
[3,] 1 0 1
[4,] 1 1 1

$neurons[[2]]
 [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      ←
 [,7]
[1,] 1 0.68636693 0.6092020 0.91035646 7.497751e-01 0.15431004 ←
 0.01945436
[2,] 1 0.99953772 0.6563290 0.90095274 2.772957e-03 0.63049301 ←
 0.52359387
[3,] 1 0.02439443 0.2006083 0.01702437 3.355866e-03 0.00140767 ←
 0.05089230
[4,] 1 0.96109643 0.2351460 0.01527601 3.124725e-06 0.01301067 ←
 0.74813172

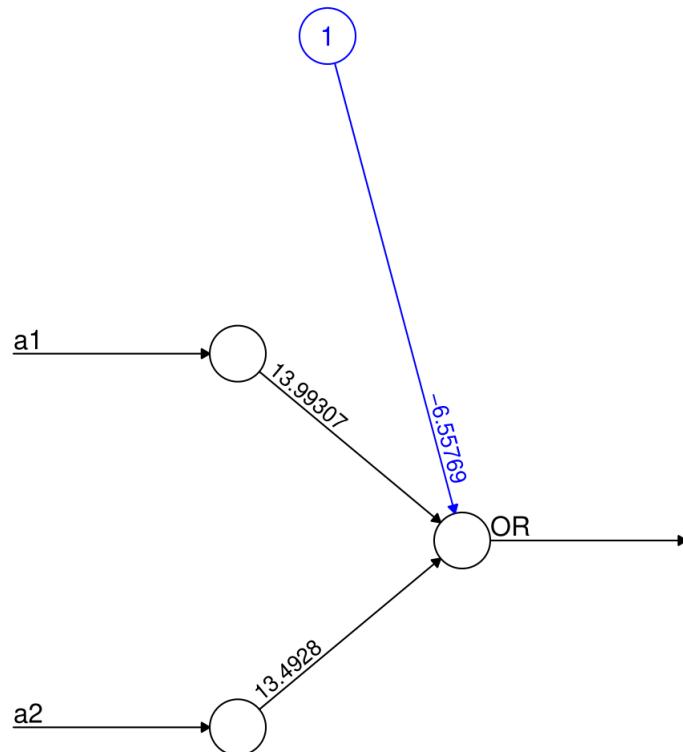
$neurons[[3]]
 [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 1 0.84563624 0.4130953 0.9529117 0.03482577
[2,] 1 0.22458384 0.1289220 0.1237234 0.99612886
[3,] 1 0.04217722 0.1161074 0.3859107 0.99282199
[4,] 1 0.92886172 0.8664505 0.9441624 0.99974822

$neurons[[4]]
 [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
[1,] 1 0.9927132 0.02167383 0.9761821 0.9940454 0.1324945 0.06255539
```

```
[2,] 1 0.1816136 0.81633663 0.2088652 0.2107308 0.5750897 0.78446193  
[3,] 1 0.1858673 0.75709052 0.3026211 0.1862775 0.5130758 0.74301901  
[4,] 1 0.9538813 0.06213487 0.8851530 0.9619660 0.1685140 0.08461656
```

```
$net.result  
[,1]  
[1,] 0.0003216646  
[2,] 0.9997287272  
[3,] 0.9994604261
```

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



Error: 2e-06 Steps: 136

4.2. ábra. Az OR művelet

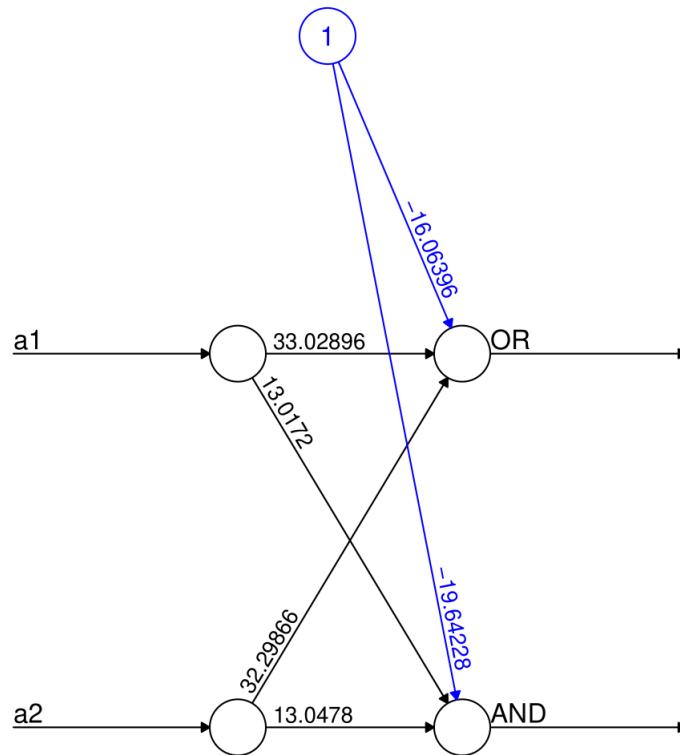
```
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
OR      <- c(0,1,1,1)
AND     <- c(0,0,0,1)

operand.data <- data.frame(a1, a2, OR, AND)

nn.operand <- neuralnet(OR+AND~a1+a2, operand.data, hidden=0, linear.output= FALSE,
                         stepmax = 1e+07, threshold = 0.000001)

plot(nn.operand)

compute(nn.operand, operand.data[,1:2])
```



Error: 3e-06 Steps: 315

4.3. ábra. Az AND és OR művelet

```

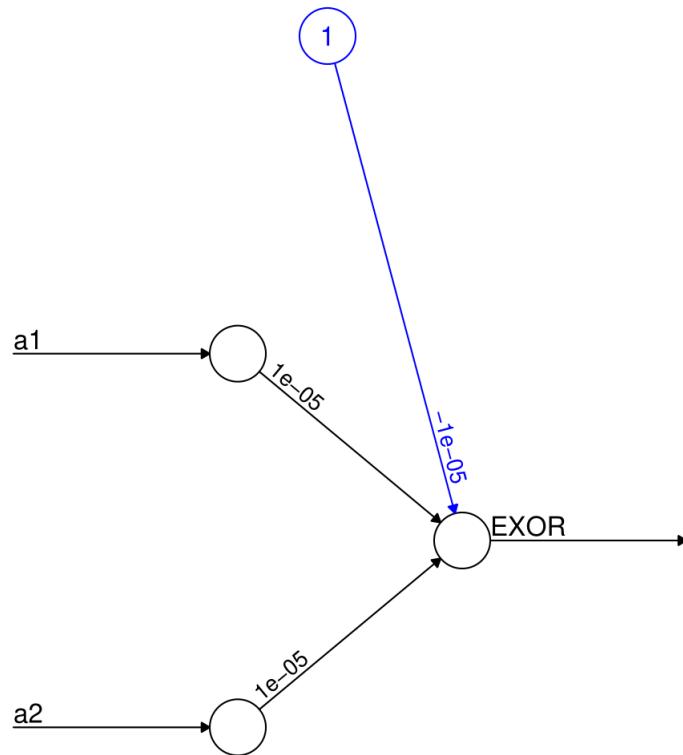
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
EXOR   <- c(0,1,1,0)

exor.data <- data.frame(a1, a2, EXOR)

nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=0, linear.output=FALSE, 
                      stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)

compute(nn.exor, exor.data[,1:2])
  
```



Error: 0.5 Steps: 123

4.4. ábra. Az EXOR rejtett rétegek nélkül művelet

```

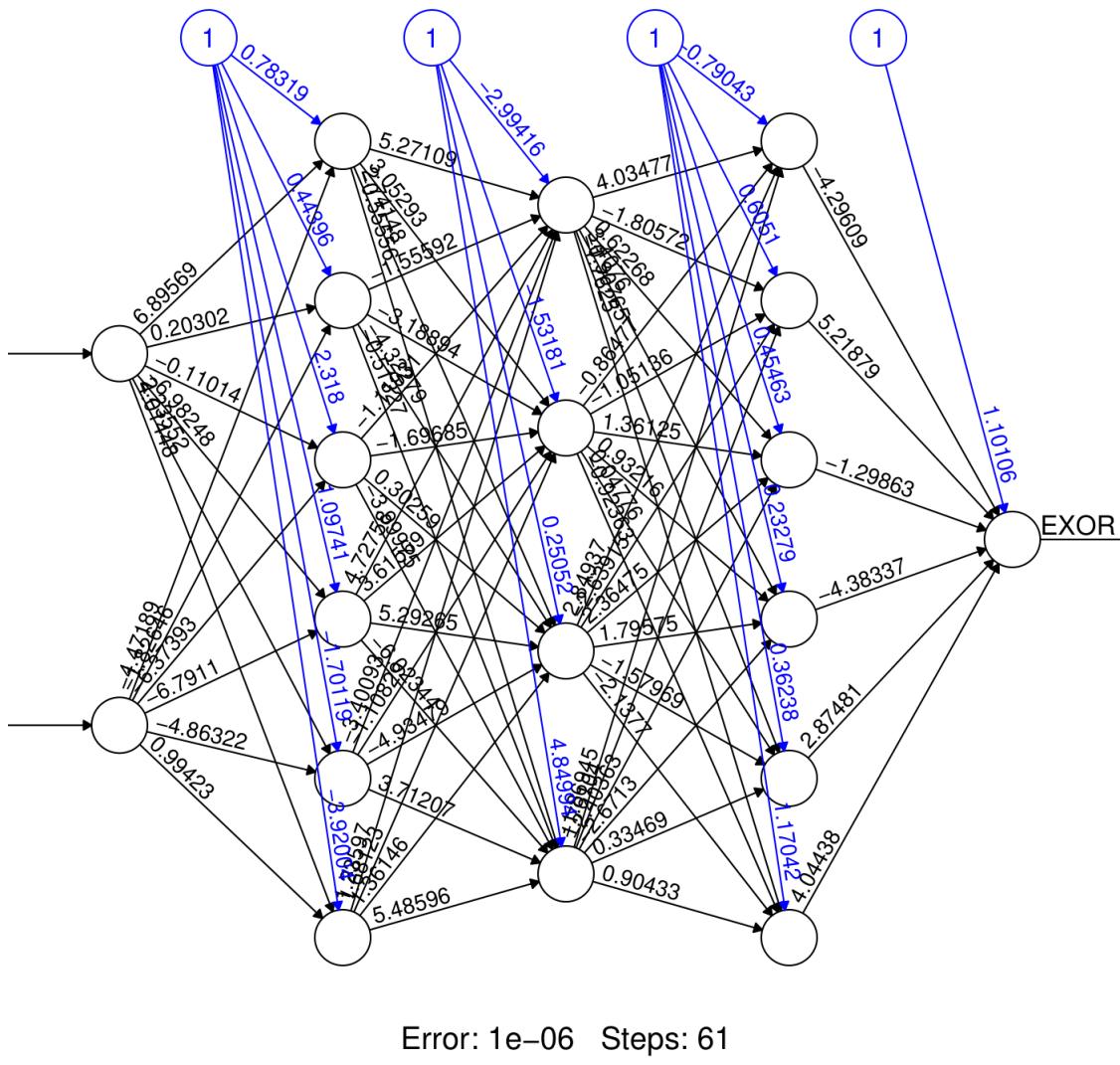
a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
EXOR   <- c(0,1,1,0)

exor.data <- data.frame(a1, a2, EXOR)

nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=c(6, 4, 6), linear. ←
  output=FALSE, stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)

compute(nn.exor, exor.data[,1:2])
  
```



4.5. ábra. Az EXOR rejtett rétegekkel művelet

4.6. Vörös pipacs pokol/Írd ki mit lát Steve!

Megoldás videó: <https://www.youtube.com/watch?v=DX8dI04rWtk>

Aki szokott a Minecraftnal játszani az biztosan tudja, hogy ha a karakterünkkel szeretnénk valamit csinálni, építeni, vagy blokkokat kiütni, bányászni, rá kell vinnünk az ablak közepén látható célkeresztet arra a blokkra amelyikkel dolgozni akarunk. Az F3+B billentyű kombinációval a játékon belül láthatjuk hogy minden egyes élőlénynek a játékban hova mutat a kis keresztre. Egy vonal jelenik meg ami azt mutatja meg, hogy egy játékos vagy élőlény ha egyenes vonalban néz akkor mire néz éppen. A Malmö programozás során ennek nagy jelentősége volt mivel mindenkor kellett érnünk, hogy Steve a virágra nézzen mivel akkor tudja kiütni. Ehhez változtatnunk kell esetleg a fej tartását, vagy azt hogy milyen szögben néz. Lekövetni pedig ezt a LineOfSight-tal tudjuk. A Line arra az egyenes vonalra utal amire mutat éppen Steve keresztre. Megadjuk hogy a self.lookingat legyen a LineOfSight és a kiíratásnál a self.lookingat-re

hivatkozunk.

```
if "LineOfSight" in observations:  
    LineOfSight = observations["LineOfSight"]  
    self.lookingat = LineOfSight["type"]  
    print("      Steve's <): ", self.lookingat)
```

A kiíratásnál egyszerűen a `self.lookingat` aktuális értékét kell figyelembe vennünk, ha ez `red_flower` akkor Steve reagáljon rá annyit, hogy kiírja `VIRAAAG!`, de egyébként az előző kódcsípetben látható kiíratás folyamatosan frissül, mindenkor azt mutatja amire Steve aktuálisan néz.

```
if self.lookingat == "red_flower": #itt talalkozik a viraggal meg ↵  
    fel is veszi  
        print("      VIRAAAG!")
```

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

Ebben a feladatban Czanik Andrást tutoráltam.

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

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

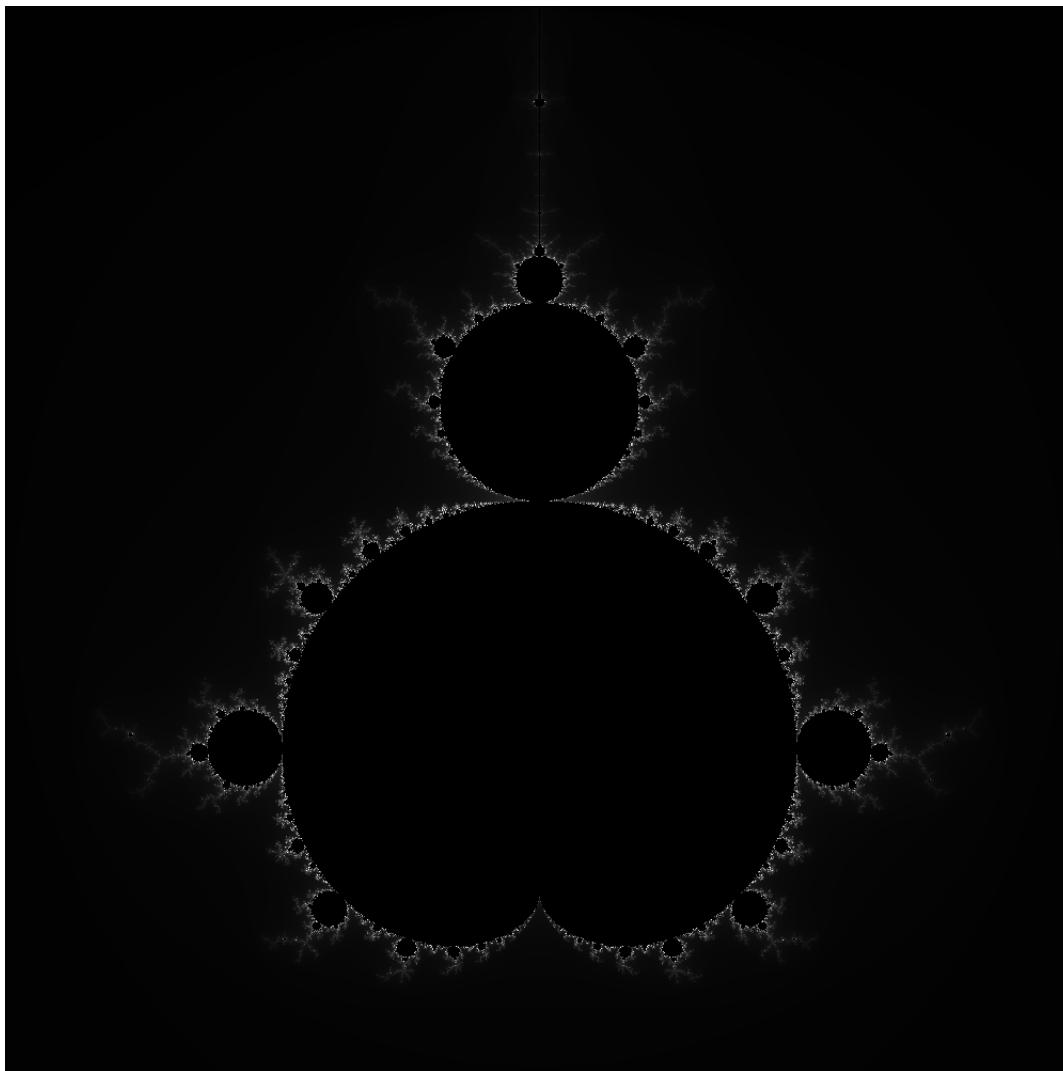
A Mandelbrot halmazt úgy láthatjuk meg, hogy a sík origója középpontú 4 oldalhosszúságú négyzetbe lefektetünk egy, mondjuk 800x800-as rácsot és kiszámoljuk, hogy a rács pontjai mely komplex számoknak felelnek meg. A rács minden pontját megvizsgáljuk a $z_{n+1} = z_n^2 + c$, ($0 \leq n$) képlet alapján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

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

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

A feladat leírását azzal szeretném kezdeni, hogy a feladatban tárgyalt kód Bátfai Norberté és ez egy C++ kód, mivel nem sikerült megfélő **libpng** könyvtárat telepítenem C nyelvhez. Mivel a kód nem használja a komplex osztályt ezért úgy kell gondolkodni mintha azt magunk akarnánk létrehozni.

A program maga a Mandelbrot halmazt rajzolja ki a komplex számsíkon. A Mandelbrot-halmaz volt az egyik első olyan fraktál alakzat amit felfedeztek. A fraktáloknak jelenleg még nincsen konkrétan megfogalmazott szigorú matematikai definíciója mivel a definíció megalkotásakor minden valamilyen ellentmondásba ütközünk. Ahogy jellemzni tudjuk őket, önhasonló alakzatok, azaz ha bizonyos részeikre ránagyítunk, mint a Mandelbrot-halmaz határterületei esetén is, az eredti alakzatot kapjuk vissza ezáltal egy végtelenül összetett struktúrát alkotva. A nevüket a latin "fractus" azaz töredzettség szóból származtatjuk ami a fraktálalakzatok határterületein megjelenő matematikailag nehezen leírható önméltódsre utal, mivel ezáltal nem jön létre tényleges határvonal például a Mandelbrot halmaz esetén sem. Viszont a természet nagyon sok fraktálszerű önhasonló struktúrában rendeződik, ilyen például a hegyek csipkézettsége, a növények növekedési logikája, vagy éppen a villámok is. Az általam egyik generált kép nekem pont a villámok által alkotott alakzatokat jutatta eszembe, ezért ezt az ábrát ide illesztem.



A program a **png++/png.hpp** könyvtár meghívásával kezdődik, ez szükséges ahhoz, hogy png-t tudjunk generálni. Ezután definiáljuk a szükséges konstansokat, ezek közül az N és az M határozzák meg, hogy mekkora méretű legyen az elkészült png. Én ezeket duplájukra, azaz 1000-re növeltelem hogy élesebb képet kaphassunk., ha ettől nagyobb számra növeljük, például 2000-re, szegmentálási hiba léphet fel, óvatosa kell

kiválasztani a számot. Ezen kívül szükségünk van a maximum illetve minimum x és y értékekre.

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

#define N 1000
#define M 1000
#define MAXX 0.7
#define MINX -2.0
#define MAXY 1.35
#define MINY -1.35
```

A következő fontos része a programunknak egy eljárás, viszont ez a program legvégén hívódik meg ezért ezt ott fogom kifejteni.

Ahhoz, hogy a komplex osztály nélkül tudjunk a komplex számsíkon dolgozni a programban létre kell hozni a komplex struktúrát. Ezzel tudjuk a komplex számainkat valós (re) és imaginárius (im) egységre bontani. Ezután a főprogramban létrehozunk egy **N*M** elemű kétdimenziós tömböt, ebbe fogjuk eltárolni, hogy a png mely léppontjai milyen színűek legyenek. A színek meghatározásához létrehozunk egy **iteracio** nevű változót. A program végigmegy a tömbön két for ciklussal amiben található egy while ciklus. A while ciklus maximum 256 alkalommal futhat le és közben az **iteracio** változóban számoljuk a futások számát. amennyiben az iteráció 256 alkalommal futott le az azt jelenti, hogy az adott érték nem tudott kilépni a Mandelbrot halmazból, annak része tehát a pixel fekete lesz az ábrán.

```
struct Komplex
{
    double re, im;
};

int main()
{
    int tomb[N][M];

    int i, j, k;

    double dx = (MAXX - MINX) / N;
    double dy = (MAXY - MINY) / M;

    struct Komplex C, Z, Zuj;

    int iteracio;

    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            C.re = MINX + j * dx;
            C.im = MAXY - i * dy;
```

```
    Z.re = 0;
    Z.im = 0;
    iteracio = 0;

    while(Z.re * Z.re + Z.im * Z.im < 4 && iteracio++ < 255)
    {
        Zuj.re = Z.re * Z.re - Z.im * Z.im + C.re;
        Zuj.im = 2 * Z.re * Z.im + C.im;
        Z.re = Zuj.re;
        Z.im = Zuj.im;
    }

    tomb[i][j] = iteracio - 1;//1 - iteracio; //256 - iteracio
}
}

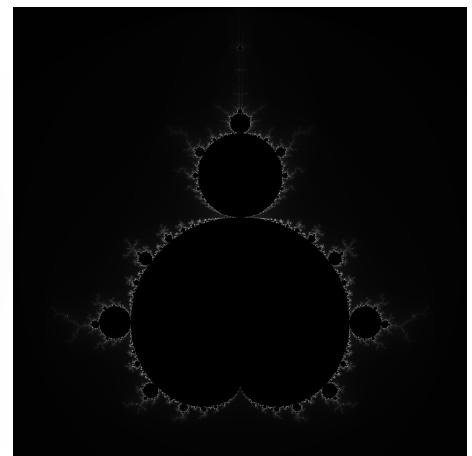
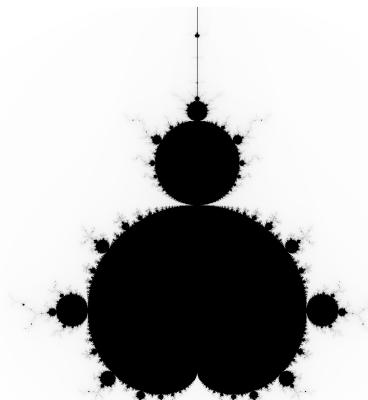
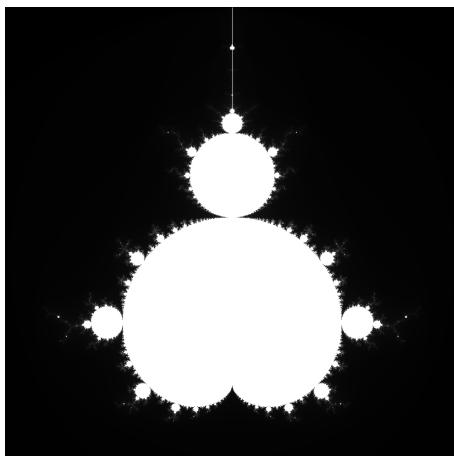
GeneratePNG(tomb);

return 0;
}
```

A pixelek színének meghatározása tehát az iterációtól függ. Én módosítottam egy kicsit a programon, az alap programban maga a halmaz fekete színű, ekkor a főprogram utolsó sorában a **tomb[i][j] = 256 - iteracio;** kifejezés szerepel. Viszont ha változtattunk ezen az értékadáson **tomb[i][j] = iteracio - 1;**-re akkor az eredи inverzét kapjuk. Ha az iterációk számából vonunk ki 256-ot akkor pedig a halmaz maga is fekete lesz és az eredи program által rajzolt ábra beli részek is feketék lesznek viszont a halmaz közvetlen környezete fehér vagy nagyon világos szürkés színű.

```
void GeneratePNG( int tomb[N][M])
{
    png::image< png::rgb_pixel > image(N, M);
    for (int x = 0; x < N; x++)
    {
        for (int y = 0; y < M; y++)
        {
            image[y][x] = png::rgb_pixel(tomb[x][y], tomb[x][y], tomb[x][y] ←
                ]); //x y
        }
    }
    image.write("kimenet.png");
}
```

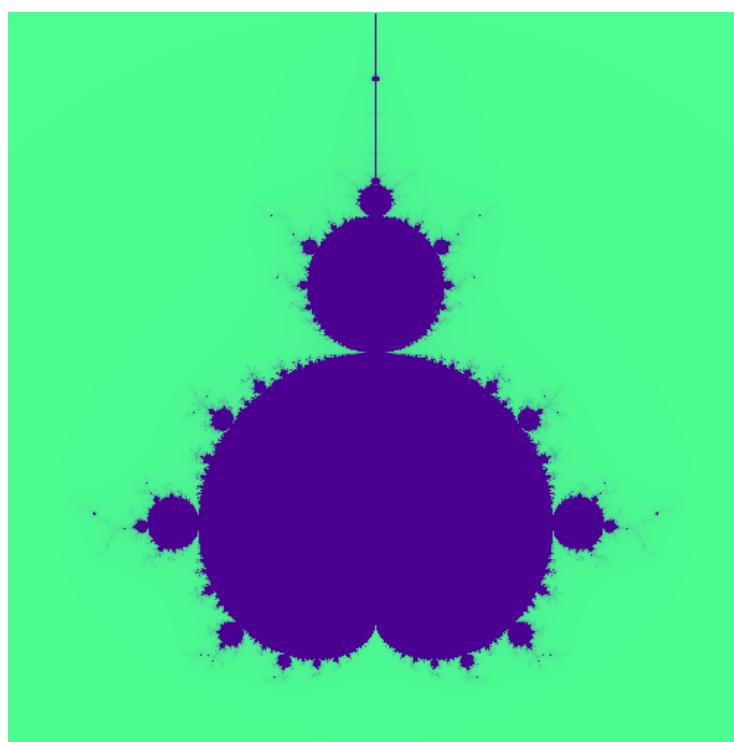
A program utolsó része a png generáló eljárás, itt is 2 for ciklus segítségével megyünk végig a kétdimenziós tömbön és megadjuk a pixelek értékét színbeli értékét. Én itt is módosítottam kicsit a képen ugyanis elforgattam, ehhez annyit kell tennünk hogy a pixelek értékadásánál az x és y-t felcseréljük.



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

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

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



A C++ sok változást hozott a C-vel szemben, több beépített osztályt és beépített függvényt találunk benne. Ezek között szerepel az `std::complex` osztály is. Az előző feladatban a programban magunk hoztuk létre a komplex struktúrát most viszont ki is használjuk a C++ előnyeit. A program maga szinte megegyezik az előző feladatban tárgyalattal, ez is Bátfai Norbert által írt program. Vizsgáljuk meg a különbségeket!

Az első fontos különbség, hogy a **png++/png.hpp** könyvtár meghívása után egyből meghívjuk a **complex** osztályt, ez a C++ beépített osztálya a komplex számokkal való kényelmesebb munkához. A **GeneratePNG** eljárás is megegyezik az előző feladatban tárgyalattal ezért annak leírását ott lehet megtekinteni.

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

```
#include <complex>

const int N = 500;
const int M = 500;
const double MAXX = 0.7;
const double MINX = -2.0;
const double MAXY = 1.35;
const double MINY = -1.35;

void GeneratePNG(const int tomb[N][M])
{
    png::image<png::rgb_pixel> image(N, M);
    for (int x = 0; x < N; x++)
    {
        for (int y = 0; y < M; y++)
        {
            image[x][y] = png::rgb_pixel(tomb[x][y], tomb[x][y], tomb[x][y] ←
                ]);
        }
    }
    image.write("kimenet.png");
}
```

A legnagyobb különbség persze a főprogramunkban van, azon belül is abban a részben, ahol a Mandelbrot-halmazt számoltatjuk a programmal. A matematikai háttere ugyanaz a két programnak csupán a komplex számokkal való folyamatok térnek el. Az előző programban ehhez egy struktúrát használtunk, mégpedig olyat, hogy két részre bontottuk a komplex számokat, valós és imaginárius egységre. Ennek a leegyszerűsítésére szolgál a komplex osztály, ugyanis egy értékadással tudunk értéket adni a C és Z számoknak, nem kell a két részüket külön kezelní mint az előző esetben ezáltal egy keveset rövidül is a program. Mivel a kép generálása és a pixelek színének meghatározásának folyamata egyezik az előző feladatban tárgyaltéval a részletesebb leírást ott lehet megtalálni.

```
int main()
{
    int tomb[N][M];

    double dx = ((MAXX - MINX) / N);
    double dy = ((MAXY - MINY) / M);

    std::complex<double> C, z, zuj;

    int iteracio;

    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
```

```
C = {MINX + j * dx , MAXY - i * dy};  
  
Z = 0;  
iteracio = 0;  
  
while (abs(Z) < 2 && iteracio++ < 255)  
{  
    Zuj = Z*Z+C;  
    Z = Zuj;  
}  
  
tomb[i][j] = 256 - iteracio;  
}  
}  
  
GeneratePNG(tomb);  
  
return 0;  
}
```

Nézzük meg hogyan néz ki a az értékadása egy komplex számnak az előző feladatban tárgyalt struktúrával:

```
C.re = MINX + j * dx;  
C.im = MAXY - i * dy;
```

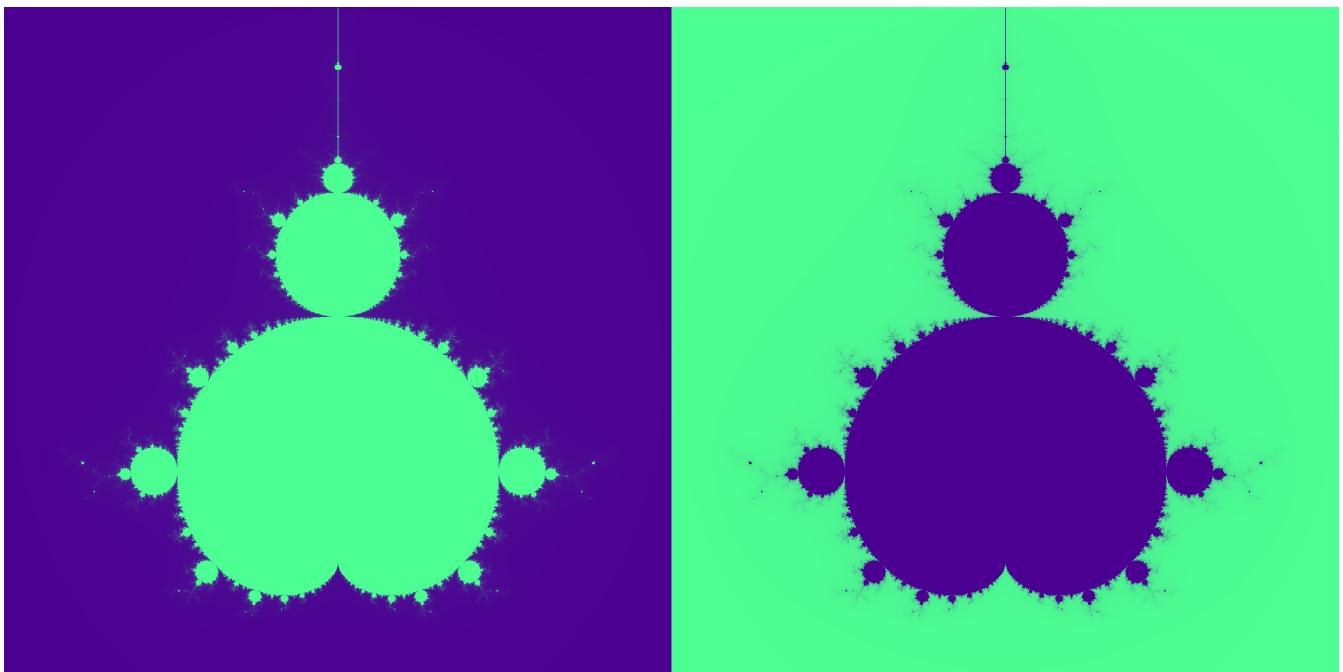
És nézzük meg ezt az `std::complex` osztályal is:

```
C = {MINX + j * dx , MAXY - i * dy};
```

A menta-lila színű halmazhoz használt színezéshez a **void GeneratePNG**-ben a következő sort így módosítottam:

```
image[y][x] = png::rgb_pixel(76, tomb[x][y], 146);
```

Illetve ismét megcseréltem az x és y tengelyt, hogy az y tengelyre legyen szimmetrikus a halmaz, szerintem így kellemesebb a szemnek a függőleges szimmetria tengely miatt.



5.3. Biomorfok

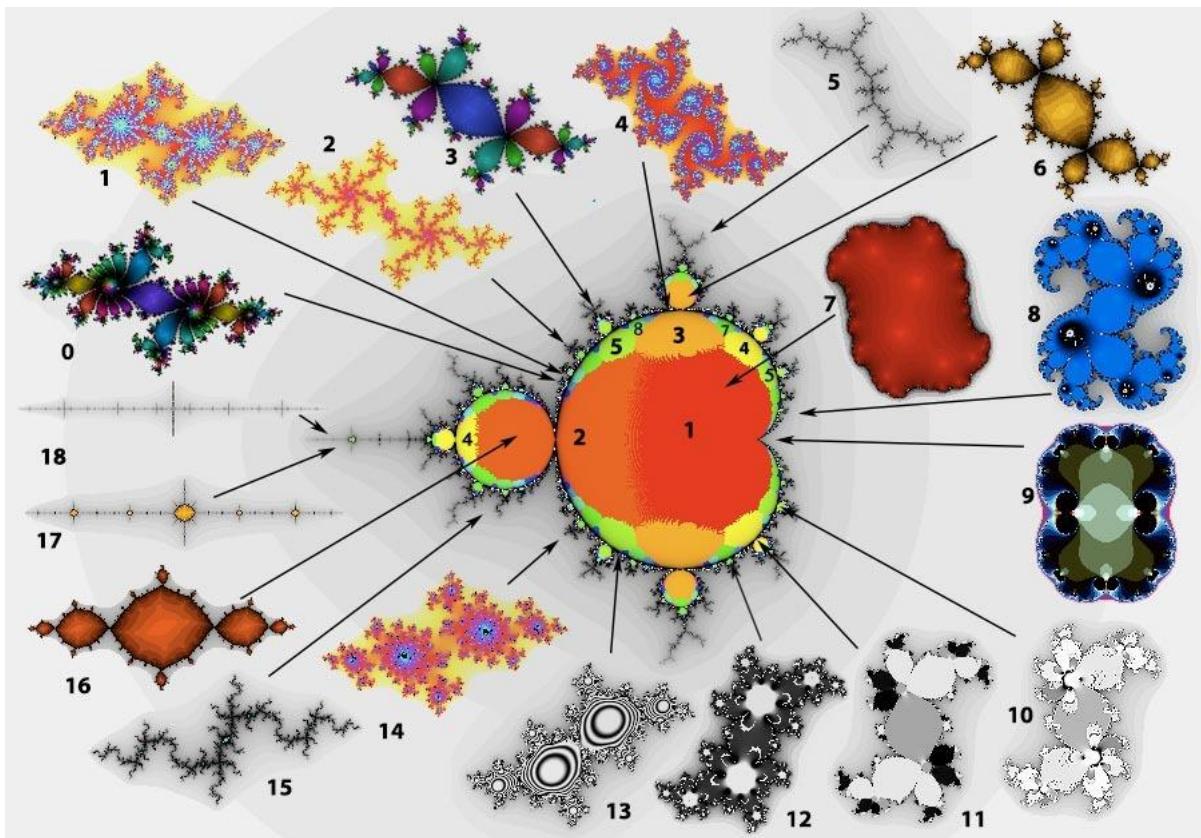
Megoldás videó: <https://youtu.be/IJMbgRzY76E>

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

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

A különbség a **Mandelbrot halmaz** és a Julia halmazok között az, hogy a komplex iterációban az előbbiben a c változó, utóbbiban pedig állandó. A következő Mandelbrot csipet azt mutatja, hogy a c befutja a vizsgált összes rácspontot.

Tehát ha a Júlia halmazokról beszélünk az azt jelenti, hogy a Mandelbrot-halmaz részhalmazait vesszük. Mivel most mi választjuk a C -t konstansnak végtelen mennyiségű ilyen halmazt találhatunk, mivel a konstansok száma végtelen. A program komplex iterációját számító részének logikája nem változik, azért kapunk más képet mert itt a C -t előre megadjuk.



5.1. ábra. A Júlia halmazokat ábrázoló szemléletes ábra a [Fractalgasm](#) Facebook csoportból

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

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

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

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

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

            ++iteracio;
        }
}
```

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

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

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

A biomorfok nevüket arról kapták, hogy amikor felfedézték őket rengeteg olyan alakzatot találtak amelyek természetben megtalálhatóak, mint például a sejtek is ezzel szemben az általam kirajzoltatott kép inkább hasonlít valamilyen szigorú meghatározott geometriai alakzatra. De a feladatmegoldás végén mindenkiőpp szemléltetem miről van szó.

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

```
// Verzio: 3.1.3.cpp
// Forditas:
// g++ 3.1.3.cpp -lpng -O3 -o 3.1.3
// Futtatas:
// ./3.1.3 bmorf.png 800 800 10 -2 2 -2 2 .285 0 10
// Nyomtatás:
// a2ps 3.1.3.cpp -o 3.1.3.cpp.pdf -l --line-numbers=1 --left-footer="↔
// BATF41 HAXOR STR34M" --right-footer="https://bhaxor.blog.hu/" --pro= ↔
// color
//
// BHAX Biomorphs
// 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/>.
//
// Version history
//
// https://youtu.be/IJMbqRzY76E
// See also https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9\_Iss5\_2305--2315\_Biomorphs\_via\_modified\_iterations.pdf
//

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

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

    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double xmin = -1.9;
    double xmax = 0.7;
    double ymin = -1.3;
    double ymax = 1.3;
    double reC = .285, imC = 0;
    double R = 10.0;

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

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

Ha a fenti kódcsipetet vizsgáljuk láthatjuk, hogy a program a futtatása során 11 argumentumot vár, ezek között a 9-es a C valós részét jelenti, a 10-s pedig az imaginárius egységes, azaz a programon belül bárhogy választhatunk konstanst, nem fogja befolyásolni a kirajzolt képet, viszont, ha a program makefile-jában ezeket az argumentumokat módosítjuk az már változtat rajta. Illetve a makefile módosításával befolyásolhatuk a kirajzolandó kép méretét is. Én az alábbi argumentumokat adtam meg a keletkezett képhez:

```
./biomorf bmorf.png 8000 8000 10 -3 3 -3 3 18.5 -15.45 10
```

melyekből a C-t befolyásoló értékek a következők:

```
18.5 -15.45
```

Illetve fontos megjegyezni, hogy én elég nagy felbontással dolgoztam, ezt a fentebbi sorból a 8000 8000 argumentumok takarják, tehát az én képem 8000*8000 pixeles felbontásban jelenik meg.

```
// Verzio: 3.1.3.cpp
png::image<png::rgb_pixel> kep( szelesseg, magassag );

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

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

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

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

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

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

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

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

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

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

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

Én a program színezését is módosítottam amit itt is hasonlóan lehet megtenni mint az előző programok esetében tehát a

```
png::rgb_pixel ( (iteracio*20)%255, (iteracio*40)%255, ( ←
    iteracio*60)%255 ) ;
```

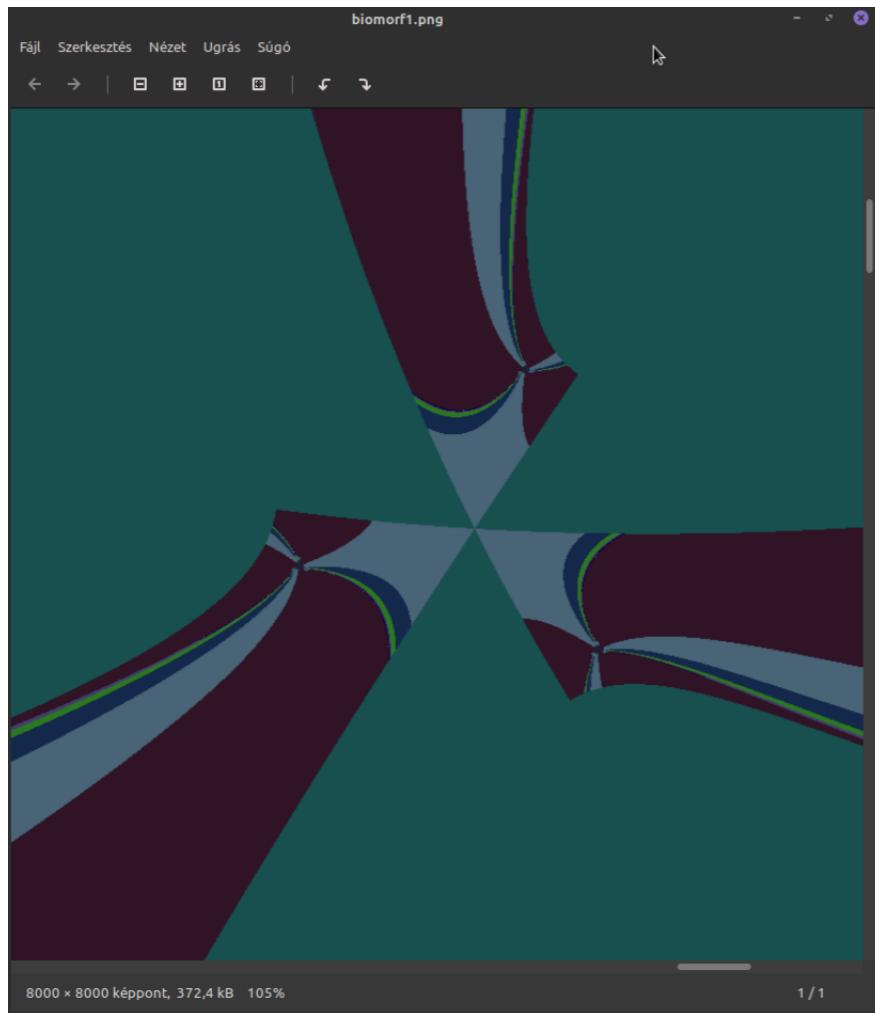
soron kell módsítanunk. Látható, hogy itt bonyolultabban számítjuk ki az iterációnól a színeket de ez majd hogynem mindegy hogyan tesszük, a program matematikai háttérét nem befolyásolja, csak a készült png-t. Én az alábbi módosításokat tettek meg a bordó-türkiz árnyalatú kép elkészítéséhez:

```
png::rgb_pixel ( (iteracio*100)%76, (iteracio*80)%140, ( ←
    iteracio*80)%122 ) ;
```

Az általam kirajzoltatott biomorf nekem azért tetszett nagyon mivel a középen található bordó háromszög szerű részek mindenkor pontosan 60 fokos szögeket zárnának be egymással, és az ábra kinyúló részeiben ugyanez az alakzat található meg csak kisebb méretben és más színekkel. És nézzük meg végre a kirajzolt png képet:



Mint ahogy említettem én nagy felbontással dolgoztam ezért megtehettem, hogy ránagyítok a kép egyes részeire, itt már felfedezhetjük benne a fraktálok önhasonló természtetét a háromszögszűrű alakzatok önis-métlődésében.



5.4. A Mandelbrot halmaz CUDA megvalósítása

Ebben a feladatban ugyanazzal a mandelbrot halmaz algoritmussal dolgozunk mint az eddigi feladatokban is viszont itt van egy nagy különbség az eddigiekhez képest. Itt a feladat lényege hogy a CPU helyett a videókártya számítási kapacitását használjuk ki. Ahhoz hogy ezt meg tudjuk csinálni mindenkorábban szükségünk van egy Nvidia kártyára mivel a CUDA az Nvidia saját fejlesztése.

A CUDA lényege, hogy az Nvidia kártyák rendelkeznek bizonyos mennyiségű CUDA magokkal, ezekkel párhuzamosan tudunk egyszerre több számítást is végezni. A CPU is rendelkezik magokkal amellyel párhuzamos számításokat végezhetünk viszont nem annyival mint egy Nvidia kártya és máshogyan is működnek, egy ilyen program esetében a CPU csak egy magot használ. Ahhoz hogy rávegyük a videókártyánkat a számításra mindenkorábban telepítenünk kell az Nvidia Cuda Toolkitet. CUDA illesztőprogramokból egyébként találhatunk többet is, ugyanis mostanában kezd egyre jobban elterjedni hogy a deep learning miatt, ami nagy számítási kapacitás igényű folyamat. Véleményem szerint a CUDA a következő évkben elégé elterjedté válhat ennek köszönhetően. Viszont most nekünk az alap Nvidia Cuda Toolkit tökéletesen megfelel.

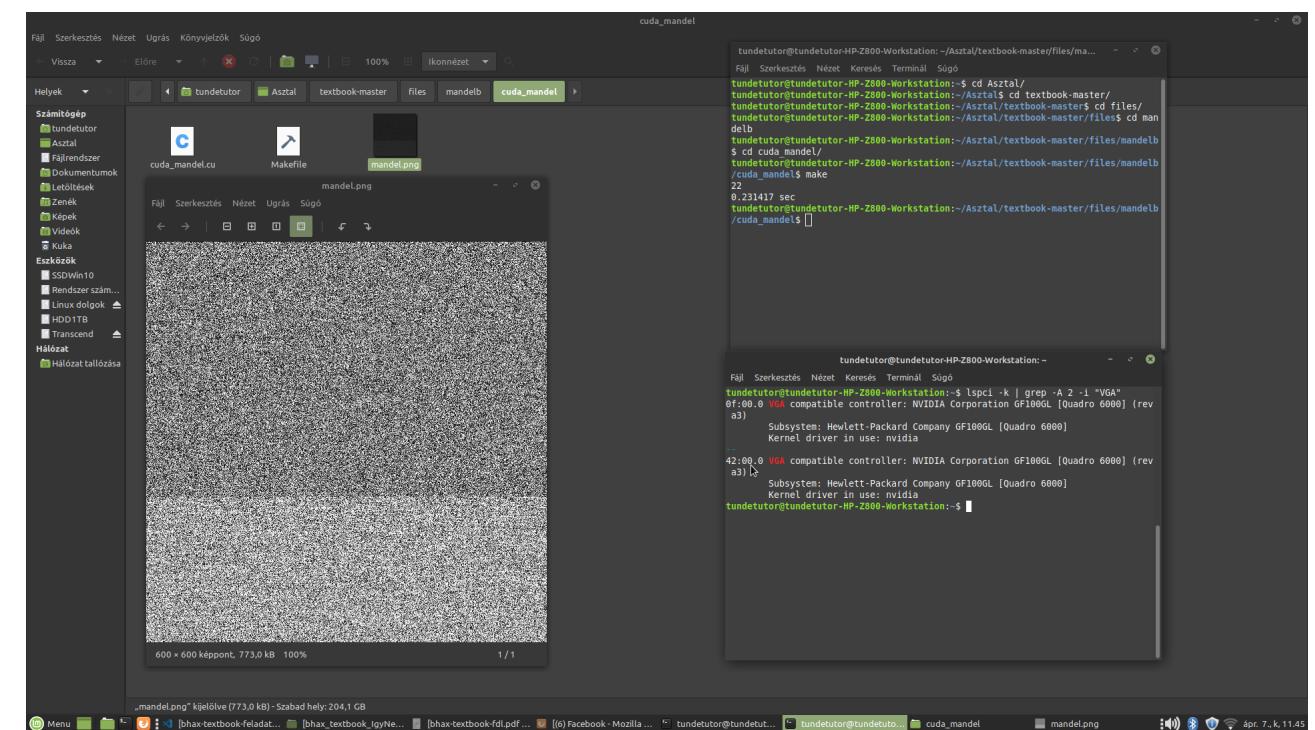
Az elképzelés az lenne hogy a png minden pixelét más szál, más CUDA mag tudja kiszámolni. Ha belegen dolunk ez hatalmas újítás ahhoz képest mintha ugyanezt a CPU-val szeretnénk megvalósítani. Nyílván ez a gyakorlatban nem teljesen így van hogy minden pixelre jut egy szál, de a CUDA-val ehhez már közelítünk, gondolunk bele hogy ha egy erősebb GPU-val rendelkezünk amiben közel 1200 cuda mag van valószínűleg

a számítás ideje jóval lecsökken a CPU-hoz képest. Azt is érdemes megemlítenünk, hogy hiába van sokmagos, sokszálas erős processzorunk ha egy folyamat csak 1 szálat fog használni, nem tudjuk kihasználni az egész teljesítményét míg a GPU esetén igen.

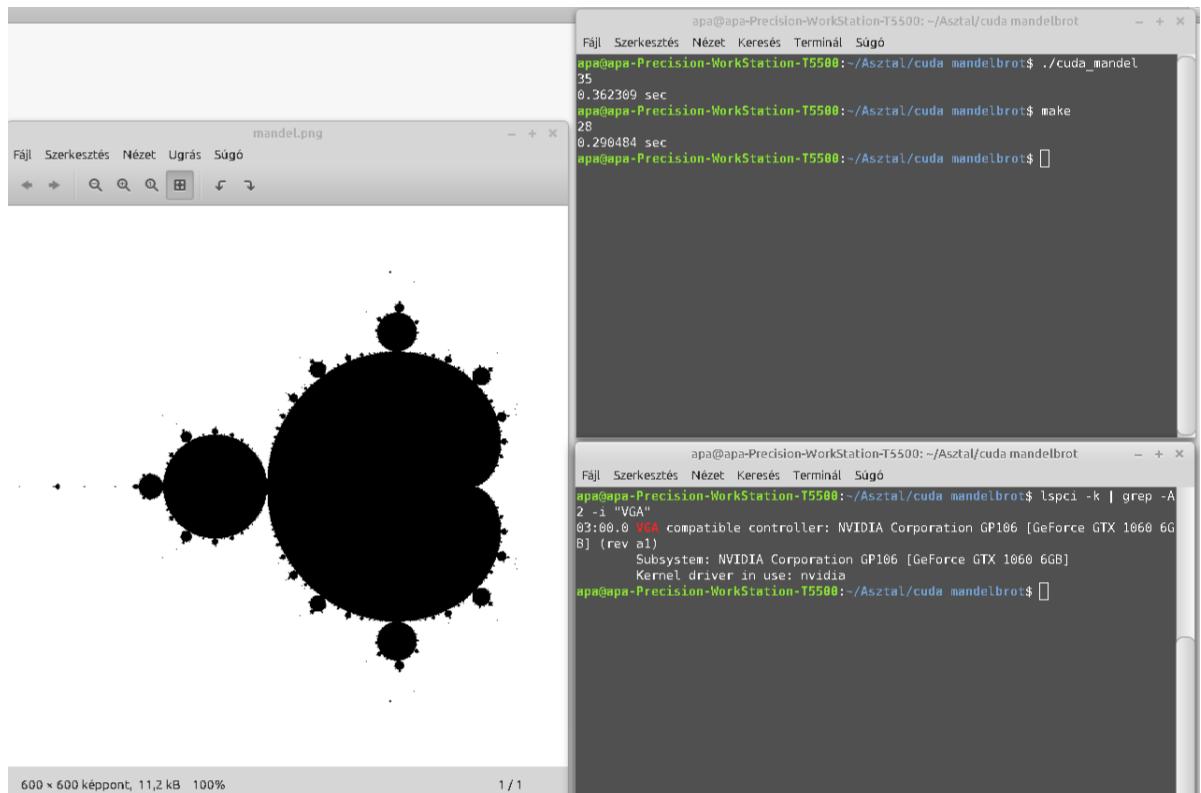
Én a CUDA-t a Blenderben való Cycles rendermotorral történő rendereléshez szoktam is használni. A Blender Cycles-e is ha nem rendelkezünk GPU-val processzorból fog számolni. A renderelés során egy ilyen technológia komoly napokat is megspórolhat nekünk, itt azért már érzi az ember, hogy valóban jelentősége van ennek. Egyébként 3D renderelés esetén hasonló dolog történik, mint a mandelbrot halmaznál, csak ott nem ezt az algoritmust használjuk, hanem a 3D felületet alkotó háromszögek normálvektoraiból lehet meghatározni egy-egy pixel minden színű legyen.

Nálam megcsinál egy png-t viszont az közel sem a mandelbrot hamlaz amit kiad. Az a sejtésem hogy a hardware konfigom a baj, én 2 db Nvidia Quadro kártyát használok, ezek kifejezettem 3D grafikai folyamatokhoz vannak fejlesztve, viszont még nem sikerült SLI hídba kötnöm őket és szerintem vagy ez okozza a gondot vagy az hogy ezek nem mai kártyák és driver probléma van. A program egyébként a következő értékeket írta ki:

```
13  
0.133559 sec
```



Akárhányszor futattom mindenkor csak hanygafocit rajzol a program. Szerencsére találtam itthon egy másik gépet is (ez egyébként apám) szóval arra is telepítettük a libpng-t majd futtatuk a programot. Abban a gépen egyébként egy Nvidia GTX 1060-as GPU-t találunk, megfelelő drivekkal és persze a CUDA Toolkit-et is már régen feltelepítettük rá. Csodák csodájára ez szépen kirajzolta a halmazt, természetesen ugyanazt a programot futtatva ami nekem ezeket a furcsa zajos képeket adta. Nyilvánvaló, hogy valami probléma van az iterációk számítása során. Amikor elhoztam a gépemet ezekkel a GPU-kkal tudtam, hogy kompromisszumokat kell kötnöm, 10 éves technológiáról beszélünk.



Megjegyzés: az általam használt videókártyák nem mai darabok, például a Blender a 2.80-as verziójától nem támogatja tehát ha a Blender rendermotorját szeretném használni akkor a 2.79-es verzióval kell dolgoznom. A 2.8 fölöttei verziók esetén a "User Preferences" fül alatt nálam nem jelenik meg a következő opció:



Viszont csatolom az általam használt kódot, ami Bátfai Norbert munkája, a matematikai háttere ennek a programnak is megegyezik az előzőleg használt programokkal.

```
// Copyright (C) 2019
// Norbert Batfai, batfai.norbert@inf.unideb.hu
// Released under GNU GPLv3

#include <png++/image.hpp>
#include <png++/rgb_pixel.hpp>
#include <sys/times.h>
#include <iostream>

#define SIZE 600
#define ITERATION_LIMIT 32000

// Vegigzongorazza a CUDA a szelesseg x magassag racsot:
```

```
__device__ int mandel(int k, int j)
{
    // most eppen a j. sor k. oszlopban vagyunk

    float a = -2.0, b = .7, c = -1.35, d = 1.35;
    int width = SIZE, height = SIZE, iterationLimit = ITERATION_LIMIT;

    float dx = (b - a) / width;
    float dy = (d - c) / height;
    float reC, imC, rez, imZ, ujrez, ujimZ;
    int iteration = 0;

    reC = a + k * dx;
    imC = d - j * dy;
    rez = 0.0;
    imZ = 0.0;
    iteration = 0;

    while (rez * rez + imZ * imZ < 4 && iteration < iterationLimit) {
        ujrez = rez * rez - imZ * imZ + reC;
        ujimZ = 2 * rez * imZ + imC;
        rez = ujrez;
        imZ = ujimZ;

        ++iteration;
    }
    return iteration;
}

__global__ void mandelkernel(int *buffer)
{
    int tj = threadIdx.x;
    int tk = threadIdx.y;

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

    buffer[j + k * SIZE] = mandel(j, k);
}

void cudamandel(int buffer[SIZE][SIZE])
{
    int *deviceImageBuffer;
    cudaMalloc((void **) &deviceImageBuffer, SIZE * SIZE * sizeof(int));

    dim3 grid(SIZE / 10, SIZE / 10);
```

```
dim3 tgrid(10, 10);
mandelkernel <<< grid, tgrid >>> (deviceImageBuffer);

cudaMemcpy(buffer, deviceImageBuffer, SIZE * SIZE * sizeof(int),
           cudaMemcpyDeviceToHost);
cudaFree(deviceImageBuffer);

}

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

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

    int buffer[SIZE][SIZE];

    cudamandel(buffer);

    png::image<png::rgb_pixel> image(SIZE, SIZE);

    for (int j = 0; j < SIZE; ++j) {
        //sor = j;
        for (int k = 0; k < SIZE; ++k) {
            image.set_pixel(k, j,
                            png::rgb_pixel(255 -
                                          (255 * buffer[j][k]) /
                                          ITERATION_LIMIT,
                                          255 -
                                          (255 * buffer[j][k]) /
                                          ITERATION_LIMIT,
                                          255 -
                                          (255 * buffer[j][k]) /
                                          ITERATION_LIMIT));
        }
    }

    image.write("mandel.png");

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

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

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

Megoldás forrása: bhax/attention_raising/CUDA/mandelpngc_60x60_100.cu nevű állománya.

5.5. Mandelbrot nagyító és utazó C++ nyelven

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

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

Megoldás forrása:

Ebben a feladatban olyan programot kellett készíteni amivel már valódi GUI-t (Graphical User Interface) is tudunk használni. Ahhoz, hogy egy ilyet lehessen készíteni szükségünk van Qt Application-re, ez fogja ugyanis a nagyító programunknak a programablakot biztosítani. A Qt-t egyébként nagyon széles körben alkalmazzák, népszerű választás ha GUI-ról van szó. Nézzük, meg hogy hogyan épül fel a program, a nagyítások miatt elégő összetett tehát nem fogom az egész forrást bemutatni, annak csak bizonyos részeit.

Kezdük a **main.cpp** nevű fájllal. Ez elég rövidke viszont jó kiindulási pont ha meg akarjuk érteni a programot. ebben láthatjuk, hogy amikor `w1` objektum létrejön meghívódik a `FrakAblak` konstruktur és `w1.show()` függvény megjeleníti a programablakot.

```
#include <QtWidgets/QApplication>
#include "frakablak.h"

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

    FrakAblak w1;
    w1.show();

    return a.exec();
}
```

Ezután tekintsük a **frakszal.cpp** programot. Ebben a programban történik a Mandelbrot halmaz kiszámítása a komplex síkon. Mivel a nagyítások során minden változik annak a négyzetről a pozíciója melyet a komplex síkon meg kell jelenítenie programunknak szüksége van az `a`, `b`, `c`, `d` értékekre amelyeket a `frakablak.cpp`-től kap meg. Itt láthatjuk a `FrakSzal` konstruktur:

```
#include "frakszal.h"

FrakSzal::FrakSzal(double a, double b, double c, double d,
```

```
        int szelessseg, int magassag, int iteraciosHatar, ←
            FrakAblak *frakAblak)
{
    this->a = a;
    this->b = b;
    this->c = c;
    this->d = d;
    this->szelessseg = szelessseg;
    this->iteraciosHatar = iteraciosHatar;
    this->frakAblak = frakAblak;
    this->magassag = magassag;

    egySor = new int[szelessseg];
}
```

Ez a 4 sor pedig a FrakSzal destruktora, látszik a "~" jelölésből.

```
FrakSzal::~FrakSzal()
{
    delete[] egySor;
}
```

A következő kódcsípet pedig az iteráció kiszámítását tartalmazza, terméshetesen a matematikai háttere itt is megegyezik az előző feladatokban megismert programokéval. Ebben a programban ami érdekes, hogy soronként számolja ki a kirajzolandó képet. Az adott sorra eső pixelek iteracio értékét, egy k elemű egySor nevű tömbbe tároljuk. Ha megnézzük a számítást, a két for ciklus közül a belső az ami k-szor, azaz szelesség számú hajtódiagram végre, tehát ezek a megjelenítendő "kép" sorai. Figyeljük meg az alábbi sort!

```
frakAblak->vissza(j, egySor, szelessseg);
```

Itt adja át ugyanis a frakSzal az egy sorban lévő iterációk értékét a frakAblak-nak.

```
void FrakSzal::run()
{
    // A [a,b]x[c,d] tartományon milyen sűrű a
    // megadott szélesség, magasság háló:
    double dx = (b-a)/szelessseg;
    double dy = (d-c)/magassag;
    double reC, imC, rez, imZ, ujrez, ujimZ;

    int iteracio = 0;

    for(int j=0; j<magassag; ++j) {
        for(int k=0; k<szelessseg; ++k) {
            reC = a+k*dx;
            imC = d-j*dy;
```

```
    rez = 0;
    imZ = 0;
    iteracio = 0;

    while(reZ*reZ + imZ*imZ < 4 && iteracio < iteraciosHatar) {
        // z_{n+1} = z_n * z_n + c

        ujreZ = reZ*reZ - imZ*imZ + reC;
        ujimZ = 2*reZ*imZ + imC;

        reZ = ujreZ;
        imZ = ujimZ;

        ++iteracio;

    }

    iteracio %= 256;

    egySor[k] = iteracio;
}

frakAblak->vissza(j, egySor, szelesseg);
}
frakAblak->vissza();

}
```

Most pedig térjünk át a megjelenítésért felelős **frakablak.cpp**-re! Először is láthatjuk a FrakAblak konstruktort. Ami 7 paramétert vár, ezek között találjuk az a, b, c, d értékeket, szelesseg-et, iteraciosHatar-t és a QWidget *parent-et. A program ezen része felelős azért, hogy elindítsa a FrakSzal-lal való komplex számításokat és újraszámításokat.

```
mandelbrot = new FrakSzal(a, b, c, d, szelesseg, magassag, iteraciosHatar, ←
    this);
    mandelbrot->start();
```



```
#include <iostream>
#include "frakablak.h"

FrakAblak::FrakAblak(double a, double b, double c, double d,
                      int szelesseg, int iteraciosHatar, QWidget *parent)
    : QMainWindow(parent)
{
    setWindowTitle("Mandelbrot halmaz");

    szamitasFut = true;
```

```
x = y = mx = my = 0;
this->a = a;
this->b = b;
this->c = c;
this->d = d;
this->szelesseg = szelesseg;
this->iteraciosHatar = iteraciosHatar;
magassag = (int)(szelesseg * ((d-c) / (b-a)));

setFixedSize(QSize(szelesseg, magassag));
fraktal= new QImage(szelesseg, magassag, QImage::Format_RGB32);

mandelbrot = new FrakSzal(a, b, c, d, szelesseg, magassag, ←
    iteraciosHatar, this);
mandelbrot->start();
}
```

Láthatjuk a FrakSzal-éhoz hasonló FrakAblak destrukturát.

```
FrakAblak::~FrakAblak()
{
    delete fraktal;
    delete mandelbrot;
}
```

Ezután a programunk javarészt osztályonkívül deklarált függvényeket tartalmaz, nézzük meg őket sorra! Mivel nagyon hosszú a program nem fogom mindegyik működését részletesen kifejteni, előfordulhat rövid leírást adok róluk.

Az első a sorban a `paintEvent()`, ez rajzolja ki a programablakunkba a képet. Ez a kirajzolás, mint ahogy említettem soronként történik.

```
void FrakAblak::paintEvent(QPaintEvent*) {
    QPainter qpainter(this);
    qpainter.drawImage(0, 0, *fraktal);
    if(!szamitasFut) {
        qpainter.setPen(QPen(Qt::white, 1));
        qpainter.drawRect(x, y, mx, my);
        if(!zX.empty()) //excuse me
        {
            for(int i=0; i<zX.size(); i++)
            {
                qpainter.drawLine(zX[i], zY[i], zX2[i], zY2[i]);
            }
        }
    }
    qpainter.end();
```

```
}
```

Az következő a mousePressEventEvent () , ez először megvizsgálja, hogy a bal egérgombot nyomtuk-e meg majd ha a feltétel igaz meghatározza az egér pozícióját a programablakban, ez szükséges, ahhoz, hogy amikor nagyítanunk megfelelően tudja kezelni a kattintástól mért kihúzás mértékét, vagyis melyik részére akarunk a komplex síknak nagyítani.

```
void FrakAblak::mousePressEvent (QMouseEvent* event) {  
  
    if (event->button() == Qt::LeftButton)  
    {  
        x = event->x();  
        y = event->y();  
        mx = 0;  
        my = 0;  
    }  
    else if (event->button() == Qt::RightButton)  
    {  
        double dx = (b-a)/szelessseg;  
        double dy = (d-c)/magassag;  
        double reC, imC, rez, imZ, ujrez, ujimZ;  
  
        int iteracio = 0;  
  
        reC = a+event->x()*dx;  
        imC = d-event->y()*dy;  
  
        rez = 0;  
        imZ = 0;  
        iteracio = 0;  
  
        while (rez*rez + imZ*imZ < 4 && iteracio < 255) {  
            // z_{n+1} = z_n * z_n + c  
            ujrez = rez*rez - imZ*imZ + reC;  
            ujimZ = 2*rez*imZ + imC;  
            zX.push_back((int)((rez - a)/dx));  
            zY.push_back((int)((d - imZ)/dy));  
            zX2.push_back((int)((ujrez - a)/dx));  
            zY2.push_back((int)((d - ujimZ)/dy));  
            rez = ujrez;  
            imZ = ujimZ;  
  
            ++iteracio;  
        }  
    }  
    update();  
}
```

Az egérgombot ha egyszer lenyomtuk azt el is foguk engedni, erre szolgál a mouseReleaseEventEvent (), ekkor történik meg az új számítás indítása.

```
void FrakAblak::mouseReleaseEvent(QMouseEvent* event) {  
  
    if(event->button() == Qt::LeftButton)  
    {  
        if(szamitasFut)  
            return;  
  
        szamitasFut = true;  
  
        double dx = (b-a)/szelessseg;  
        double dy = (d-c)/magassag;  
  
        double a = this->a+x*dx;  
        double b = this->a+x*dx+mx*dx;  
        double c = this->d-y*dy-my*dy;  
        double d = this->d-y*dy;  
  
        this->a = a;  
        this->b = b;  
        this->c = c;  
        this->d = d;  
  
        delete mandelbrot;  
        mandelbrot = new FrakSzal(a, b, c, d, szelessseg, magassag, ←  
            iteraciosHatar, this);  
        mandelbrot->start();  
    }  
    update();  
}
```

Ezután a mouseMoveEventEvent () függvénytel kiszámolunk az egér 2 lekért pozíciójából egy négyzetet ez lesz az új kirajzolandó kép a komplex síkon.

```
void FrakAblak::mouseMoveEvent(QMouseEvent* event) {  
  
    // A nagyítandó kijelölt terület szélessége és magassága:  
    mx = event->x() - x;  
    my = mx; // négyzet alakú  
  
    update();  
}
```

Az utolsó amiről írni szeretnék az alábbi függvény. A harmadik paramétere egy mutatót tartalmaz, ami a memóriában minden az aktuális kirajzolandó pixel iterációs értékére mutat, tehát innen kapja meg a

FrakAblak a FrakSzal-lal kiszámoltatott, tömbben tárol sorait a képnek. Ebben a részben én a színezés miatt módosítottam egy keveset illetve a program minidg kiírja az aktuláis pixel RGB komponenseit.

```

void FrakAblak::vissza(int magassag, int *sor, int meret)
{
    int piros = 0;
    int zold = 0;
    int kek = 0;

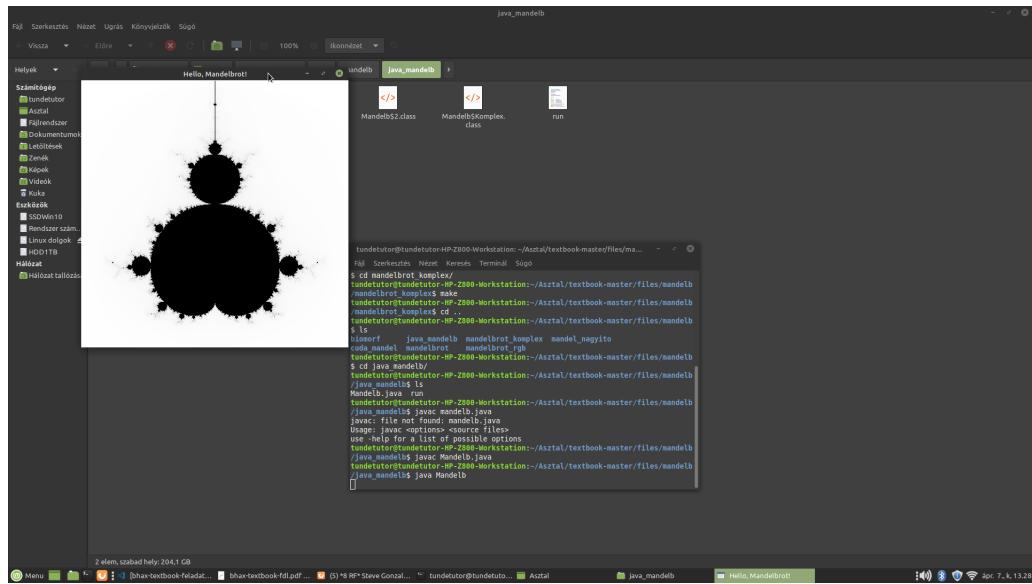
    for(int i=0; i<meret; ++i)
    {
        piros = 89 - sor[i];
        zold = 255 - sor[i];
        kek = 89 - sor[i];

        std::cout << piros << " " << zold << " " << kek << " " << std::endl;

        QRgb szin = qRgb(piros, zold, kek); //qRgb(135, 255-sor[i], 180); // ←
        //qRgb(piros, kek, zold); //qRgb(0, 255-sor[i], 0); qRgb(105, ←
        //255-sor[i], 180);
        fraktal->setPixel(i, magassag, szin);
    }
    update();
}

```

Így néz ki a kiszínezett Mandelbrot-halmaz, ha a különböző részeire ránagyítunk:



5.6. Mandelbrot nagyító és utazó Java nyelven

Ebben a feladatban egy Bátfai Norbert által készített és [Czanik András](#) általt módosított forrást dolgozok fel. Mivel az eredeti program még nem tudott nagyítani ezért András segítségét kértem. A forráskód logikája szinte ugyanaz mint a C++ verzióé.

Megoldás videó: <https://youtu.be/Ui3B6IJnssY>, 4:27-től. Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_apbs02.html#id570518

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

Itt látható a program matematikai számításokért felelős része a `run()` függvény, természetesen az iteráció kiszámítához még minden az előző képletet használtuk.

```
public void run() {  
  
    double dx = (b-a)/szélesség;  
    double dy = (d-c)/magasság;  
    double reC, imC, reZ, imZ, ujreZ, ujimZ;  
    int rgb;  
    int iteráció = 0;  
  
    for(int j=0; j<magasság; ++j) {  
        sor = j;  
        for(int k=0; k<szélesség; ++k) {  
            reC = a+k*dx;  
            imC = d-j*dy;  
            reZ = 0;  
            imZ = 0;  
            iteráció = 0;  
  
            while(reZ*reZ + imZ*imZ < 4 && iteráció < iterációsHatár) {  
                ujreZ = reZ*reZ - imZ*imZ + reC;  
                ujimZ = 2*reZ*imZ + imC;  
                reZ = ujreZ;  
                imZ = ujimZ;  
  
                ++iteráció;  
            }  
            iteráció %= 256;  
            rgb = (255-iteráció) |  
                  ((255-iteráció) << 8) |  
                  ((255-iteráció) << 16);  
            kép.setRGB(k, j, rgb);  
        }  
        repaint();  
    }  
    számításFut = false;  
}
```

A kód első része az alábbi, nem a számításokért felelős résszel kezdődik. Ebben a kódcsipetben látható a MandelbrotZoom osztály létrehozási illetve itt tröténik a változók deklarálása is. Mivel a változók neve és szerepe az előző feladatban tárgyalt kódossal szinte pontosan megegyezik ezért nem szeretném bővebben kifejteni.

```
public class MandelbrotZoom extends java.awt.Frame implements Runnable {  
  
    protected double a, b, c, d;  
    protected int szélesség, magasság;  
    protected java.awt.image.BufferedImage kép;  
    protected int iterációsHatár = 255;  
    protected boolean számításFut = false;  
    protected int sor = 0;  
    protected static int pillanatfelvételszámláló = 0;  
  
    public static int x;  
    public static int y;  
    public static int mx;  
    public static int my;
```

Az alábbi addMouseListener rész felel az egérgombok nyomásának vizsgálatáért, ez is ugyanúgy működik mint a C++ változatban, tehát ha megnyomjuk a bal egérgombot akkor a az aktuális pozíóját az egérnek az mx my változókba menti el, illetve ebben a kód részletben találjuk a mouseReleased függvényt is mivel ez is az egér gombnyomásával kapcsolatos műveletekhez tartozik.

```
    addMouseListener(new java.awt.event.MouseAdapter() {  
        @Override  
        public void mousePressed(java.awt.event.MouseEvent me) {  
            x = me.getX();  
            y = me.getY();  
        }  
  
        @Override  
        public void mouseReleased(java.awt.event.MouseEvent me) {  
            számításFut = true;  
  
            double dx = (b - a) / szélesség;  
            double dy = (d - c) / magasság;  
  
            double range = 60;  
  
            double a = MandelbrotZoom.this.a+x*dx;  
            double b = MandelbrotZoom.this.a+x*dx+range*dx;  
            double c = MandelbrotZoom.this.d-y*dy-range*dy;  
            double d = MandelbrotZoom.this.d-y*dy;  
  
            MandelbrotZoom.this.a = a;  
            MandelbrotZoom.this.b = b;  
            MandelbrotZoom.this.c = c;  
            MandelbrotZoom.this.d = d;  
  
            //new MandelbrotZoom(a, b, c, d, szélesség, magasság);  
            new Thread(MandelbrotZoom.this).start();  
        }  
    });
```

```
    //repaint();  
}  
});
```

Itt pedig a `mouseMoved` részt láthatjuk, a program itt számolja ki az egér mozgása által megtett utat amivel a nagyítás mértékét tudjuk meghatározni, azaz hogy a komplex síkon vett mely négyzetet szeretnénk a program által megjeleníteni.

```
addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {  
  
    @Override  
    public void mouseMoved(java.awt.event.MouseEvent me) {  
        int mx = me.getX() - x;  
        int my = mx;  
  
        repaint();  
    }  
});
```

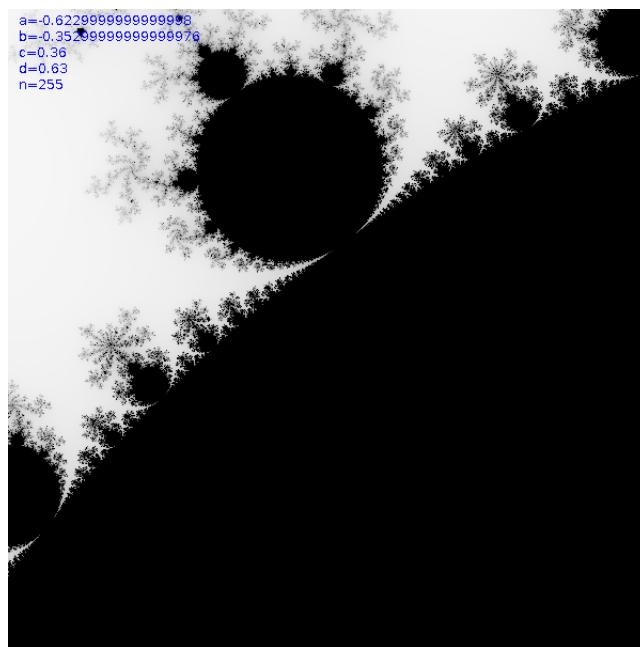
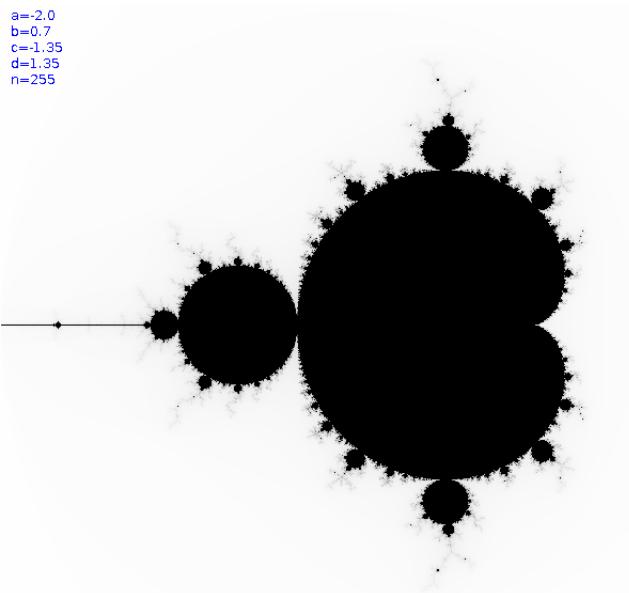
A Java és a C++ változatok közti legnagyobb különbség az alábbi függvényben rejlik. Ha megnyomjuk a billentyűzeten az "S" gombot a program egy pillanatképet készít a komplex sík azon részéről amelyiket éppen megjelenítette, mégpedig úgy hogy "rányomtatja" az aktuális `a`, `b`, `c`, `d` értékeket, és az aktuális iterációshatárt.

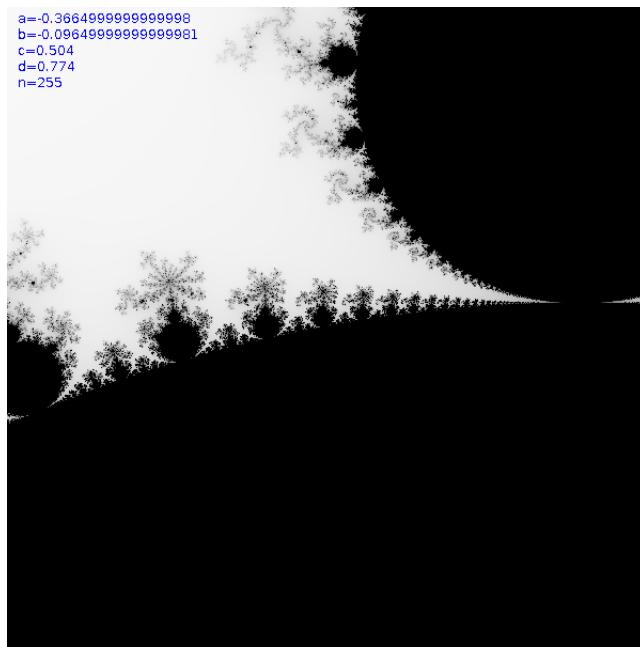
```
public void pillanatfelvétel() {  
    java.awt.image.BufferedImage mentKép =  
        new java.awt.image.BufferedImage(szélesség, magasság,  
            java.awt.image.BufferedImage.TYPE_INT_RGB);  
    java.awt.Graphics g = mentKép.getGraphics();  
    g.drawImage(kép, 0, 0, this);  
    g.setColor(java.awt.Color.BLUE);  
    g.drawString("a=" + a, 10, 15);  
    g.drawString("b=" + b, 10, 30);  
    g.drawString("c=" + c, 10, 45);  
    g.drawString("d=" + d, 10, 60);  
    g.drawString("n=" + iterációsHatár, 10, 75);  
    g.dispose();  
  
    StringBuffer sb = new StringBuffer();  
    sb = sb.delete(0, sb.length());  
    sb.append("MandelbrotZoom_");  
    sb.append(++pillanatfelvételszámláló);  
    sb.append("_");  
    sb.append(a);  
    sb.append("_");  
    sb.append(b);  
    sb.append("_");  
    sb.append(c);  
    sb.append("_");
```

```
sb.append(d);
sb.append(".png");

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

Én is készítettem néhány ilyen pillanatfelvételt a programmal, a mentett png-keket itt lehet megtekinteni:





5.7. Vörös pipacs pokol/fel a láváig és vissza

Ebben a feladatban tutoráltam volt Papp Csenge.

Megoldás videó

Mivel ahogy már korábban említettem egy összetettebb feladat megoldást szedt kisebb darabokra. A feladat lényegi része a következő. Stevnek Fel kell szaladnia az arénában ameddig csak tud, azaz amíg nem találkozik lávával. Ezt úgy oldottuk meg, hogy Steve alap mozága 2 move 1-ből és egy jumpmove 1-ből áll. Azaz egy adott szinten megtesz két lépést majd ugrik minidg a következőre egészen addig amíg lávát nem érzékel.

```
self.agent_host.sendCommand( "move 1" )
time.sleep(0.01)
self.agent_host.sendCommand( "move 1" )
time.sleep(0.01)
if escape == 0:
    self.agent_host.sendCommand( "jumpmove 1" )
    time.sleep(0.5)
```

A láva érzékelését egy 3x3x3-mas cuboid segítségével oldottuk meg. A cuboudot amit lát Steve egy 27 elemet tartalmazó tömbként értelmezhetjük, egy for ciklussal megvizsgáljuk annak minden elemét, hogy a tömb éppen tartalmaz-e "lava"-t vagy "flowing_lava"-t. Amennyiben erre a válasz igen Steve 180 fokos fordulatot vesz és leszalad az aréna aljába, innen fogja elkezdeni a csiga mozgását.

```
for i in range(27):
    if nbr3x3x3[i] == "flowing_lava" or nbr3x3x3[i] == "←
        lava":
        print("      Lavaaaaa! ")
```

```
if escape == 0:  
    self.agent_host.sendCommand( "turn -1" )  
    time.sleep(0.01)  
    self.agent_host.sendCommand( "turn -1" )  
    time.sleep(0.01)  
escape = 1 #ez az escape igaz hamis hogy kell-e ←  
menekülni, akkor már nem ugrik csak fut meg ne ←  
forogjon
```

6. fejezet

Helló, Welch!

6.1. LZW

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

A feladatcsokorban ez volt szerintem a legfontosabb feladat és a többi is főleg erre épített. Ezt a programot, Bátfai Norbert Tanár Úr videói alapján írtam meg. Most ezt a bináris fát a nulláról kezünk el írni, ez a "from scratch" fánk. Viszont felmerül a kérdés, mi is maga a bináris fa?

A bináris fák adatstrukturák. A félév során az Adatszerkezetek és algoritmusok nevű tantárgy is sokat foglalkozik ezzel a fajta adatstruktúrával, melyben különböző adattípusokat tudunk tárolni. Az egyik legegy-szerűbb talán amiben csak egész számokat tárolunk, ezen keresztül bemutatom a működését. Ha adott egy egész számokat tartalmazó rendezett tömbünk, annak elemeit bináris fában is tudjuk tárolni. Ezzel lényegesen lerövidíthetjük az adathalmazunkban történő keresés idejét. A bináris fáknak van 1 db gyökere, és 2 db gyermek a gyökérnek, egy jobb és egy baloldali. Ha az egész számos példánál maradunk, akkor a gyökér a tömbként ábrázolt adathalmazban a középső indexű elem, a bal oldali gyermek a gyökérnél kisebb értékek között a középső elem, a jobboldalinál pedig az annál nagyobbak közti középső elem és így tovább minden gyermekkel és azok gyermekivel. Tehát a bináris fánk szintenként duplázdik elemszámot tekintve, ez azt jelenti, hogy egy levélement (olyan csomópont amely nem rendelkezik gyermekkel) könnyedén elérhetővé válik. Ha van például kb 2500 adatunk, látható, hogy egy keresett elem maximum 11 lépésből elérhető, ez adja az algoritmus gyorsaságát.

A programunkban tárgyalta bináris fa első sorban 0 és 1 értékkal dolgozik, ezeket úgy rendezzük el a fában, hogy ha a vizsgált elem 0 akkor balra kerül, ha 1 akkor pedig jobbra. A bemenő adatokat vizsgáljuk és ha nem létezik még olyan csomópont mint a bemenő adat akkor létrehozunk egyet és a mutatót visszaállítjuk a gyökérre, ha már létezik akkor pedig ráállítjuk a mutatót. Ilyen módon tudjuk magát a fát felépíteni.



Megoldás videó:

Mielőtt megnézzük a forráskód részletet beszúrok egy videót amiben rajzon keresztül magyarázom el, hogy hogyan épül fel a bináris fa, [megtekintő itt](#).

Nézzük meg a "from scratch" bináris fa programunknak az LZW fa-építésért felelős részét (ebben a programban ZLW fának hívjuk)! Az első esetben azt vizsgáljuk, ha 0 érték érkezik. az első eset amit vizsgálunk

az az eset amikor annak az elemnek, amelyiken éppen van a mutató, nincs baloldali gyermeké akkor létre hozunk egy új csomópontot, ezután a `treep` ráállítjuk az újonnan létrejött csomópontra, majd átállítjuk a gyökérre. Az else ágban minden össze annyit találunk, hogy ha már létezik az adott elemnek baloldali gyermeké a mutatót arra állítjuk át.

```
template <typename ValueType>
ZLWTree<ValueType> & ZLWTree<ValueType>::operator<<(ValueType value ←
)
{
if (value == '0')
{
    if (!this -> treep -> leftChild())
    {
        typename BinTree<ValueType>::Node * node = new typename BinTree ←
            <ValueType>::Node(value);
        this -> treep -> leftChild(node);
        this -> treep = this -> root;
    }
    else
    {
        this -> treep = this -> treep -> leftChild();
    }
}
}
```

A fa szimmetrikus szerkezete miatt ha 0-tól különböző, azaz 1-es érték érkezik, azzal is ugyanolyan módon kell eljarnunk mint az előző esetben, viszont itt a csomópontot `rightChild(node)`-nak nevezzük. Az else ágban pedig itt is ráállítjuk a mutatót a már létező baloldali gyermekre.

```
else
{
    if (!this -> treep -> rightChild()) //a beérkző adat 1-es ezért ←
        jobboldali gyerek
    {
        typename BinTree<ValueType>::Node * node = new typename BinTree ←
            <ValueType>::Node(value);
        this -> treep -> rightChild(node);
        this -> treep = this -> root;
    }
    else
    {
        this -> treep = this -> treep -> rightChild();
    }
}
return *this;
}
```

A videóban ami alapján én is írtam a programot, Bátfai Norbert az alábbi bitsorral tesztelte a programját, így én is úgy döntöttem azzal tesztem, hogy biztos legyek annak működésében. Szerencsére jól lefutott a program és ezt a kimenetet kaptam:

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

6.2. Fabejárás

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

Ebben a feladatban segítséget kaptam [Kusmicki Balázstól](#).



Megoldás videó:

A feladattal kapcsolatosan készítettem egy [rajzos videót](#) a különböző fa bejárási algoritmusokról.

Ehhez a feladathoz is a "from sracth" programot használtam. Ez a kód alapvetőleg inorder bejárást használ, azaz először a baloldalt vizsgálja, aztán az adott csomópont gyökerét majd annak a jobb oldali gyermekeit, tehát ha inorder kiíratást végzünk magának a fának a gyökere középen fog megjelenni. Nézzünk is meg egy kódcsipetet és az ilyen módon bejárt fa kimetetét is.

```
template <typename ValueType>
void BinTree<ValueType>::print(Node *node, std::ostream & os)
{
    if (node)
    {
        ++depth;
        print(node -> leftChild(), os);

        for (int i = 0; i < depth; ++i)
        {
            os << "----";
        }

        os << node -> getValue() << ' ' << depth << ' ' << node -> getCount() << ' ' << std::endl;
    }
}
```

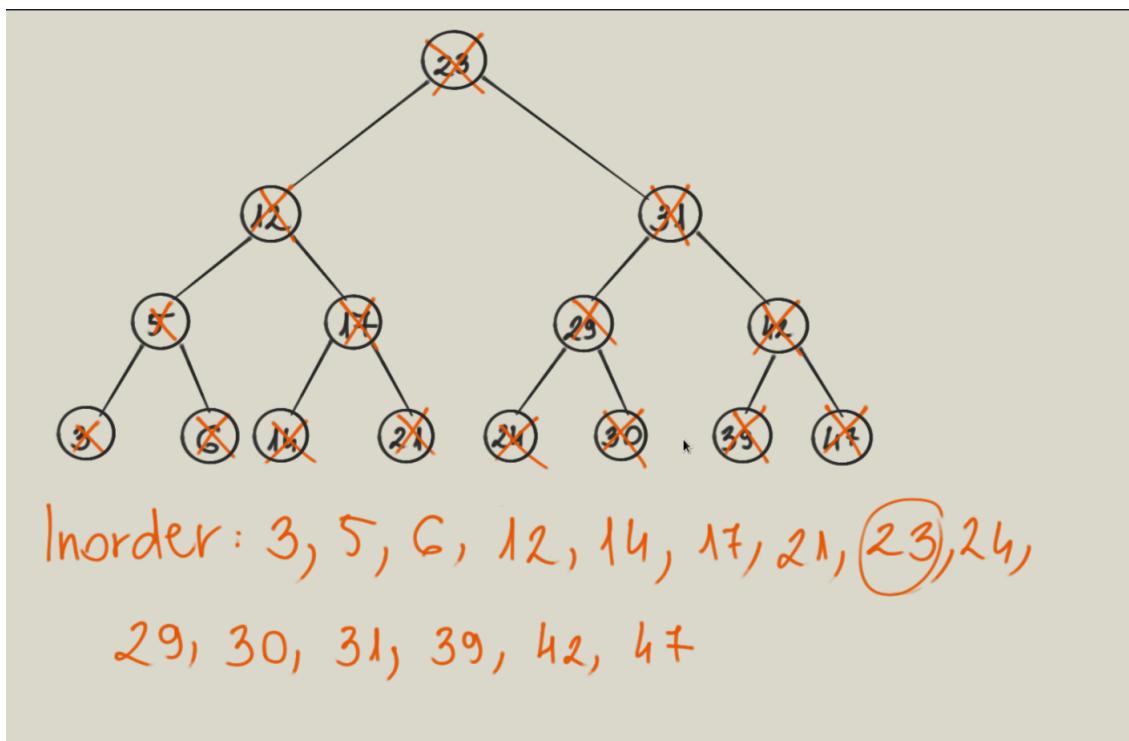
```
    print(node -> rightChild(), os);
    --depth;
}
}
```

```

-----0 3 0
-----0 2 0
-----1 3 0
---/ 1 0
-----0 5 0
-----0 4 0
-----0 3 0
-----1 2 0
-----1 3 0
-----1 4 0

```

Egyszerű péda inorder bejárásra:



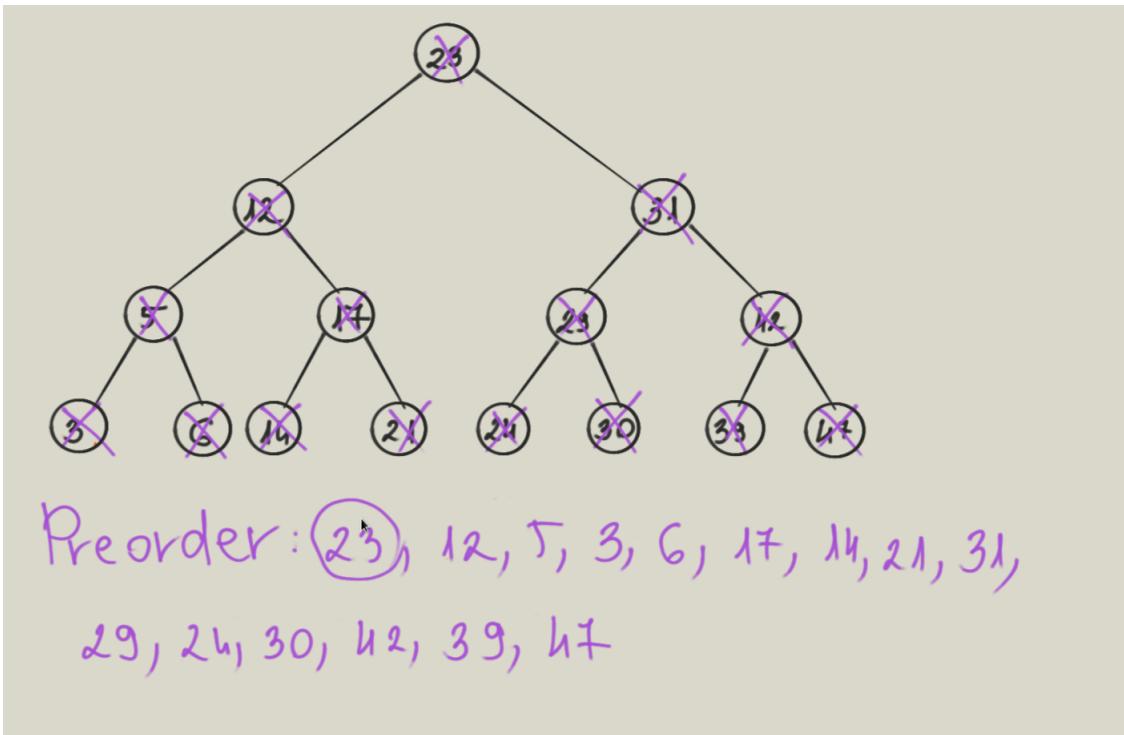
A következő bejárás a preorder bejárás lesz, ez azt jelenti, hogy először minden gyökeret vizsgáljuk, azt követően a előzőr a baloldalt, majd a jobb oldalt. Itt is megmutatom a kódcsipetet és a kimenetet is, ebben az esetben a teljes fa gyökere lesz az első kiíratott elem.

```
template <typename ValueType>
void BinTree<ValueType>::print(Node *node, std::ostream & os)
{
    if(node)
```

```
{  
    ++depth;  
  
    for(int i = 0; i < depth; ++i)  
    {  
        os << "---";  
    }  
  
    os << node -> getValue() << ' ' << depth << ' ' << node -> getCount() << endl;  
  
    print(node -> leftChild(), os);  
  
    print(node -> rightChild(), os);  
    --depth;  
}  
}  
}
```

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

Egyszerű péda preorder bejárásra:



Végül nézzük meg a postorder bejárást is. Ennél a bejárásnál az előzőhez képest annyi különbség van, hogy a yökeret vizsgáljuk utolára, de a balról-jobbra bejárás megmarad. Ekkor az utolsó kiíratott elem lesz magának a fának a gyökere.

```

        template <typename ValueType>
void BinTree<ValueType>::print(Node *node, std::ostream & os)
{
    if (node)
    {
        ++depth;

        print(node -> leftChild(), os);

        print(node -> rightChild(), os);

        for (int i = 0; i < depth; ++i)
        {
            os << "---";
        }

        os << node -> getValue() << ' ' << depth << ' ' << node -> getCount() << ' ' << std::endl;
    }
    --depth;
}
}

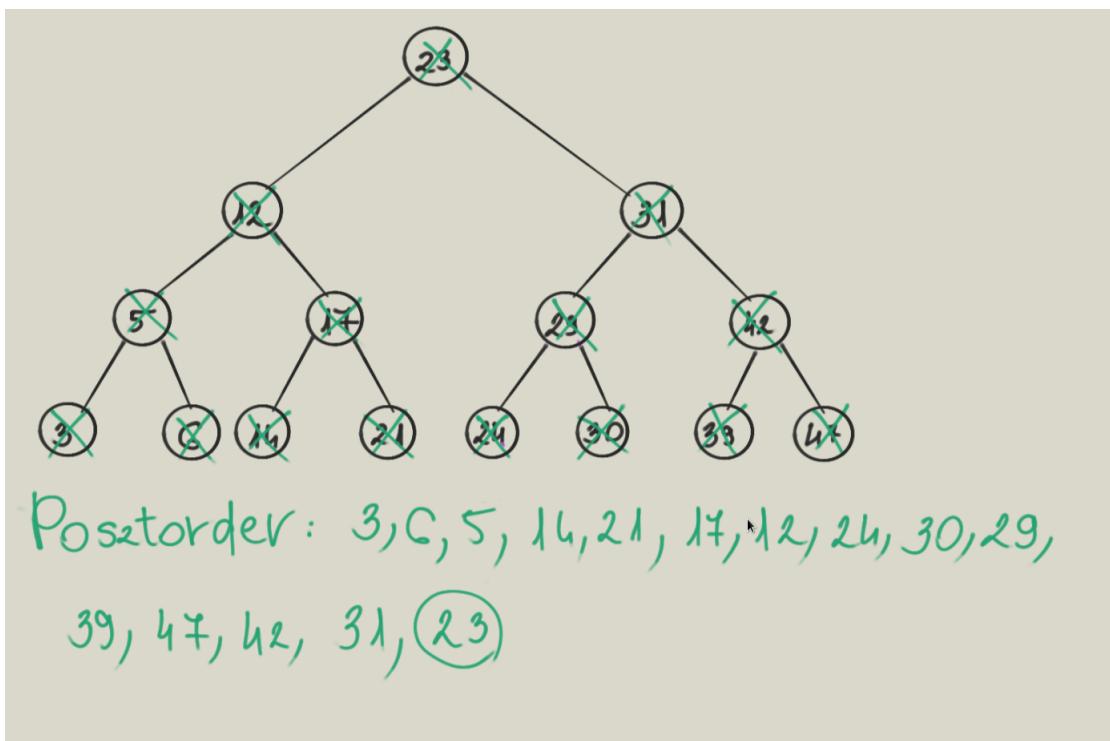
```

```

-----0 3 0
-----1 3 0
----0 2 0
-----0 5 0
-----0 4 0
----0 3 0
-----1 4 0
----1 3 0
-----1 2 0
---/ 1 0

```

Egyszerű péda posztorder bejárásra:



6.3. Mozgató szemantika

Ír az előző programhoz mozgató konstruktort és értékadást, a mozgató konstruktur legyen a mozgató értékadásra alapozva!

Megoldás video:

Ehhez a feladathoz azt a programot fejlesztük tovább amit a 2. feladathoz használtunk. Itt mindenki szóba kell hoznunk, hogy ha a fát szeretnénk mozgatni, akkor ezt csak úgy tehetjük meg a BinTree ősosztályba beágyazott Node osztálytal együtt rekurzívan hajtjuk végre. Egyébként készítetünk másoló és másoló értékadó konstruktort is de most azok kódcsipetét nem teszem bele a könyvbe, lehet a későbbiek során még belekerül, de most a mozgató szemantikán volt a hangsúly. Nézzünk egy kimenetet is, mivel látható a kódcsipetben hogy a mozgató konstruktorba tesztelés cljából tettünk kiíratást is.

Ehhez a feladathoz a második feladatban használt programom fejlesztettük tovább. A "rule of five" szerint fogunk eljárni, azaz ha valamelyik konstruktort szeretnénk használni, akkor hozzuk létre a másik négyet is (megjegyzem ezek közül az egyik a destruktur). Először a destruktort fogom megmutatni, a "~"-ról ismerhetjük meg egyszerűen, ez fogja a létrejött fa objektumunkat kitörölni a memóriából mégpedig a gyökértől kezdve.

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

Ezután tegyünk egy pillantást a másoló konstruktorunkra. Láthatjuk, hogy itt összetettebb folyamatokról van szó. Egy kiíratással kezdődik, hogy a program futása során lássuk mikor hívódik meg. A másoló konstruktur használatának célja, hogy a már létező fánkat szeretnénk valahova máshova elhelyezni a memóriában. Ahhot, hogy ezt meg tudjuk tenni először is valahol a memóriában létre kell hoznunk egy az előzővel megegyező struktúrát, ezért az eredit elnevezzük **old**-nak, ezután a **cp()** függvényel lemásoljuk az eredeti fa gyökerét és a fa mutatóját amit itt **treep**nek nevezünk. A **print()** rekurzív kiíratáshoz hasonlóan pedig kialakítjuk az új fánk struktúráját és a **new Node**-dal hozzuk létre az új f csomópontjait. Fontos, hogy ha éppen azt a csomópontot másoljuk amelyiken éppen volt az eredeti fa mutató az új fába is ugyanoda kell hogy kerüljön.

```
BinTree(const BinTree & old)
{
    std::cout << "BT copy ctor, másoló konstruktur" << 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;
}
```

A következő a sorban a másoló-értékkadó konstruktorkunk. Ahhoz, hogy ez működjön kell lennie már létező destruktornak és másoló konstruktornak ugyanis a másoló konstruktorkorra alapozzuk. amikor ezt a konstruktort szándékozzuk használni, akkor jó eséllyel egy már meglévő fát szeretnénk már szintén létező fába másolni. ahhoz, hogy ez megtörténjen az eredeti fának a struktúráát fel kell szabadítanunk. Léterhözünk egy ideiglenes **tmp** fát és ebbe fogjuk bele tenni az **old**-ról készült másolatot. A `swap()` függvénytel pedig megcserelem az ideiglenes fát az aktuálissal majd a destruktorknak az eredeti fának a tartalmát kiüríti.

```
BinTree & operator=(const BinTree & old)
{
    std::cout << "BT copy assign ctor, masolo ertekekadas" << std::endl;

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

Mivel a mozgató konstruktort a mozgató-értékadóra alapozzuk először azt fogom bemutatni. Ha a mozgató-értékadó konstruktort akarjuk használni szükségünk van a memóriában már két létező fára ugyanis ez a konstruktorknak őket hivatott megcsereálni. Nagyon egyszerűen működik, mivel már van két létező objektumunk annyi dolgunk van hogy megcsereljük őket mégpedig külön `swap()` függvénytel gyökereiket és külön `swap()` függvénytel a fa mutatókat. Ha ezzel készen vagyunk van mire alapozni a mozgató konstruktorunkat!

```
BinTree & operator=(BinTree && old) {
    std::cout << "BT move assign ctor, mozgato ertekekadas" << std::endl;

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

    return *this;
}
```

A mozgató-értékadás során új fát hozunk létre egy már megérvő fénak az memórián belüli áthelyezésével. Az új fának a gyökerét beállítjuk **nullptr** majd kicseréljük az eredével majd az eredeti fát kitörlijük

```
BinTree(BinTree && old) {
    std::cout << "BT move ctor, mozgato " << std::endl;

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

Mivel minden konstruktorkunkba tettünk kiíratást így a program kimenetéből látszik mikor éppen melyik konstruktort használtuk.

```
BT ctor
-----2 3 0
-----5 2 0
-----7 3 0
---8 1 0
-----9 2 0

BT ctor
-----0 3 0
-----0 2 0
---/ 1 0
BT copy ctor, masolo konstruktor
BT ctor
 ***
BT copy assign ctor, masolo ertekekadas
BT copy ctor, masolo konstruktor
BT move ctor, mozgato
BT move assign ctor, mozgato ertekekadas
BT move assign ctor, mozgato ertekekadas
BT move assign ctor, mozgato ertekekadas
BT dtor
BT dtor
 ***
BT move ctor, mozgato
BT move assign ctor, mozgato ertekekadas
BT dtor
BT dtor
BT dtor
BT dtor
BT dtor
```

6.4. Vörös pipacs pokol/ 5x5x5, azaz Steve szemüvege

Megoldás videó: <https://www.youtube.com/watch?v=DX8dI04rWtk>

Steve érzékelésének lényegi részét, azt a bizonyos cuboidot az előző fejezetben láthattuk viszont ttől többre is képes. Sokkal nagyobb teret képes belátni. Ahhoz hogy erre rákényszerítsük az xml fájlban kell módosításokat végeznünk. Látható, hogy az én xml-emben többféle, egészen pontosan 3 grid szerepel. Az első a 3x3x3-mas ami az alap volt, ekkor Steve környezetében minden csak a szomszédos blokkokat tudta tudta vizsgálni, ezért Steve-től számítva minden 1 és -1 értékekkel terjesztettük ki a gridet. Az 5x5x5-ös verzióban Steve szomszédságában két blokkot tudunk vizsgálni, a 7x7x7-esben pedig 3 blokk távolságon belül tud érzékelni. Azért, hogy lehessen válogatni éppen melyik gridet akarjuk használni külön hoztam létre minden a 3-mat, és a Python kódban külön tudunk ezekre hivatkozni.

```
<ObservationFromGrid>
    <Grid name="nbr7x7">
        <min x="-3" y="-3" z="-3"/>
        <max x="3" y="3" z="3"/>
    </Grid>
    <Grid name="nbr5x5">
        <min x="-2" y="-2" z="-2"/>
        <max x="2" y="2" z="2"/>
    </Grid>
    <Grid name="nbr3x3">
        <min x="-1" y="-1" z="-1"/>
        <max x="1" y="1" z="1"/>
    </Grid>
</ObservationFromGrid>
```

Itt láthatjuk, hogy Steve az 5x5x5-ös gridet hogyan tudja használni, ekkor a vizsgálható blokkok száma 125.

```
sensations = world_state.observations[-1].text
print("    sensations: ", sensations)
observations = json.loads(sensations)
nbr5x5x5 = observations.get("nbr5x5", 0)
print("    5x5x5 neighborhood of Steve: ", nbr5x5x5 ←
    )
```

És a 7x7x7-es verzióval pedig a tömbünk 343 eleműre bővül.

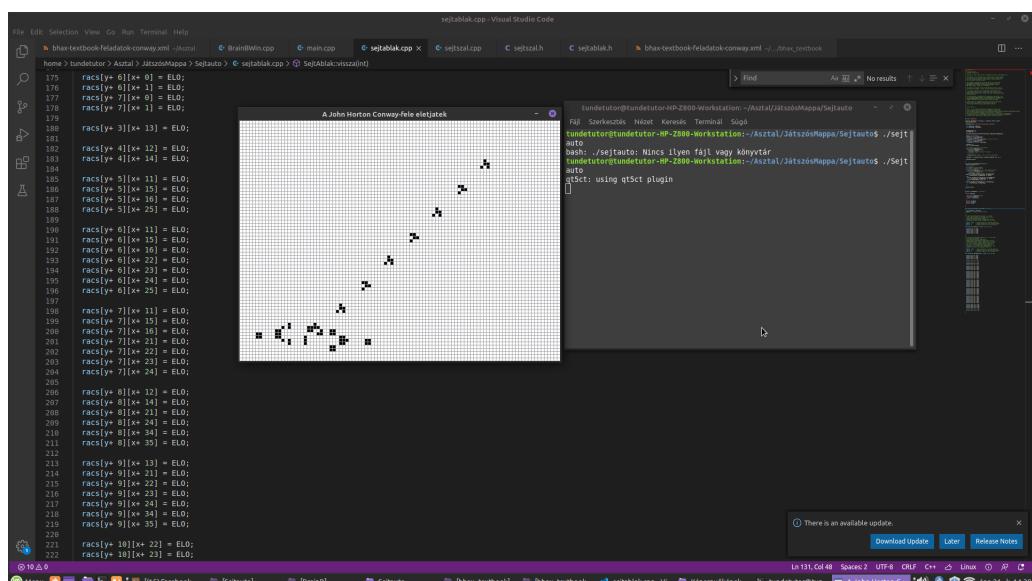
```
sensations = world_state.observations[-1].text
print("    sensations: ", sensations)
observations = json.loads(sensations)
nbr5x5x5 = observations.get("nbr5x5", 0)
print("    5x5x5 neighborhood of Steve: ", nbr5x5x5 ←
    )
```

7. fejezet

Helló, Conway!

7.1. Qt C++ életjáték

Most Qt C++-ban!



Megoldás forrása:

A játékot John Horton Conway, Cambridge-i matematikus alkotta meg. A játék egy sejtautomata. A lényege az hogy, szimulálja a sejtek életét, egyszerű szabályok meghatározásával. Maga a "játékmenet" egyébként elég passzív, a játkosnak annyi dolga van, hogy az első generációyi sejtet ő helyezi el a rácshálón majd figyeli mi történik velük. Nézzük meg a játék szabályait!

A sejtek a rácshálóban egy 8 másik rácspontot tudnak "érzékelni", ezek a szomszédjaik. Ez azért fontos mert egy sejt önmagában elpusztul, szüksége van más sejtekre is a környezetében, ezeknek a száma 2 és 3 lehet, a közvetlen szomszedságában. Tehát 1 sejt 1 szomszéddal elpusztul de ha 3-nál több szomszédja van, akkor már "túlszaporodtak" és megint elpusztul. Új sejt születik akkor ha egy sejtnek pontosan 3 szomszédja van. A játék körökre, generációkra bontható, és minden generáció változásait vizsgálhatjuk. A játék szabályai ilyen egyszerűek mégis nagyon összetett dolgokkal találkozhatunk.

A játék során kialakulhatnak úgynevezett stabil alakzatok, ilyen például egy egyszerű négyzet amely 4 db sejtből áll, ekkor minden sejtnek 3 szomszédja van tehát túlélik a kört. Az egyik leghíresebb azakzat az a

"síkló" amely átlósan tud "mozogni", illetve a vízsintesen mozgó ūrhajó. Ezek olyan alakzatok amelyek idővel "mozgásuk során" önmagukba alakulnak vissza. Csatolok is róluk ide egy képet, arról mikor futattam ezt a programt:

A program maga két header-ből és három .cpp fájlból áll. A header fájlok a **sejtszal.h** és a **sejtablak.h**, ezekben hozzuk létre a SejtSzal és SejtAblak osztályokat illetve deklaráljuk a szükséges változókat. Először a **sejtszal.cpp** programot szeretném kicsit részletesebben bemutatni.

A **sejtszal.cpp** elején találjuk a SejtSzal konstruktort és láthatjuk hogy elkéri a számításokhoz szükséges változókat.

```
SejtSzal::SejtSzal(bool ***racsok, int szelesseg, int magassag, int ←
    varakozas, SejtAblak *sejtAblak)
{
    this->racsok = racsok;
    this->szelesseg = szelesseg;
    this->magassag = magassag;
    this->varakozas = varakozas;
    this->sejtAblak = sejtAblak;

    racsIndex = 0;
}
```

Ezután az adott sejt szomszédjainak állapotát meghatározó függvényt láthatjuk, azaz a szomszedokSzama-t, ez a függvény egy adott sejt környezetét vizsgálja, és az "ELO" állapotú szomszédok számával tér vissza. Egy adott sejt szomszédjainak állapotát a rácshálóban úgy tudjuk megvizsgálni, hogy egy 3x3-mas mátrix környezetet nézünk amely ugye összesen 9 elem, ebből a középső lesz az éppen vizsgált sejt, az ő környezetében lévő 8 sejt állapotát szeretnénk minden meghatározni.

```
int SejtSzal::szomszedokSzama(bool **racs,
                                 int sor, int oszlop, bool allapot) {
    int allapotuSzomszed = 0;
    // A nyolcszomszedok vegigzongorázása:
    for(int i=-1; i<2; ++i)
        for(int j=-1; j<2; ++j)
            // A vizsgált sejetet magat kihagyva:
            if(!((i==0) && (j==0))) {
                // A sejttérbol szelenek szomszedai
                // a szembe oldalakon ("periodikus hatarfeltetel")
                int o = oszlop + j;
                if(o < 0)
                    o = szelesseg-1;
                else if(o >= szelesseg)
                    o = 0;

                int s = sor + i;
                if(s < 0)
                    s = magassag-1;
                else if(s >= magassag)
```

```
s = 0;

    if(racs[s][o] == allapot)
        ++allapotuSzomszed;
}

return allapotuSzomszed;
} 0;
}
```

A következő programrészlet egy eljárás lesz amiben összehasonlítjuk az aktuális mátrixot az egy generációval ót megelelőzővel. Igazából itt vannak definiálva az életjáték szabályai is, itt határozza meg a program hogy hogy alakul ki a következő generáció. Fontos hogy ugye két mátrixot használ a program mivel olyan helyen nem jöhet létre élő sejt ahol az adott körben halott sejt van.

```
void SejtSzal::idoFejlodes() {

    bool **racsElotte = racsok[racsIndex];
    bool **racsUtana = racsok[(racsIndex+1)%2];

    for(int i=0; i<magassag; ++i) { // sorok
        for(int j=0; j<szelesség; ++j) { // oszlopok

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

            if(racsElotte[i][j] == SejtAblak::ELO) {
                /* Elo elo marad, ha ketto vagy harom elo
                szomszedja van, kulonben halott lesz. */
                if(elok==2 || elok==3)
                    racsUtana[i][j] = SejtAblak::ELO;
                else
                    racsUtana[i][j] = SejtAblak::HALOTT;
            } else {
                /* Halott halott marad, ha harom elo
                szomszedja van, kulonben elo lesz. */
                if(elok==3)
                    racsUtana[i][j] = SejtAblak::ELO;
                else
                    racsUtana[i][j] = SejtAblak::HALOTT;
            }
        }
    }
    racsIndex = (racsIndex+1)%2;
}
```

Mivel nem akarunk véget vetni a játékmenetnek ezért azt egy végtelen while ciklusban futtatjuk. A **sejt-**

szal.cpp utolsó eleme pedig a SejtSzal destruktora.

```
void SejtSzal::run()
{
    while(true) {
        QThread::msleep(varakozas);
        idoFejlodes();
        sejtAblak->vissza(racsIndex);
    }
}

SejtSzal::~SejtSzal()
{}
```

A következő amivel foglalkozni szeretnék az a **sejtablak.cpp**, ebben készül el a programablakunk. Az első dolog amivel találkozunk az a SejtAblak konstruktor, azon pedig tartalmazza a programablak nevének kiírása. Ezután elkérjük a SejtAblak osztályban deklarált magassag és szelesseg változókat, ezután pedig értéket adunk a cellaMagassag és cellaSzelesseg változóknak, mindenkor 6 lesz mivel négyzeteket szeretnénk kialakítani. Magának a játék felületének a mérete ezek szorzatából fog kialakulni, ugyanis a magassag és szelesseg értékekre úgy kell tekintenünk hogy 1 cellányi egységben számoljuk őket. Ezután pedig helyet foglalunk a két rácsunknak amelyek a már korábban említett mátrixaink lesznek majd beállítjuk, hogy először minden sejt halott legyen a játék kezdetén és csak ezután hívjuk meg a siklókilövőt ami mindenkor ugye indul a játék. Ezek voltak a játék előkészületei, mostmár meghívhatjuk a start() függvényt ami el is indítja.

```
SejtAblak::SejtAblak(int szelesseg, int magassag, QWidget *parent)
: QMainWindow(parent)
{
    setWindowTitle("A John Horton Conway-féle eletjáték");

    this->magassag = magassag;
    this->szelesseg = szelesseg;

    cellaSzelesseg = 6;
    cellaMagassag = 6;

    setFixedSize(QSize(szelesseg*cellaSzelesseg, magassag*cellaMagassag));

    racsok = new bool**[2];
    racsok[0] = new bool*[magassag];
    for(int i=0; i<magassag; ++i)
        racsok[0][i] = new bool [szelesseg]; //mátrixot hoz létre
    racsok[1] = new bool*[magassag];
    for(int i=0; i<magassag; ++i)
        racsok[1][i] = new bool [szelesseg];
```

```
racsIndex = 0;
racs = racsok[racsIndex];

// A kiinduló racs minden cellaja HALOTT
for(int i=0; i<magassag; ++i)
    for(int j=0; j<szelesseg; ++j)
        racs[i][j] = HALOTT;
// A kiinduló racsra "ELOlenyeket" helyezünk
//siklo(racs, 2, 2);

sikloKilovo(racs, 5, 60); //itt hívja meg a siklókilövőt

eletjatek = new SejtSzal(racsok, szelesseg, magassag, 120, this);
eletjatek->start();
```

A paintEvent() függvényel történik a rács frissítése, újrarajzolása, illtve az "ELO" állapotú sejtek cellájának színezése is. Ide szúrnám be a vissza eljárást is mivel ő hívja meg minden újból a paintEvent()-et.

```
void SejtAblak::paintEvent(QPaintEvent*) {
QPainter qpainter(this);

// Az aktualis
bool **racs = racsok[racsIndex];
// racsot rajzoljuk ki:
for(int i=0; i<magassag; ++i) { // vegig leped a sorokon
    for(int j=0; j<szelesseg; ++j) { // s az oszlopok
        // Sejt cella kirajzolása
        if(racs[i][j] == ELO) //feltölти az adott színnel
            qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                cellaSzelesseg, cellaMagassag, Qt::black);
        else
            qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                cellaSzelesseg, cellaMagassag, Qt::white);
        qpainter.setPen(QPen(Qt::gray, 1));

        qpainter.drawRect(j*cellaSzelesseg, i*cellaMagassag,
            cellaSzelesseg, cellaMagassag);
    }
}

qpainter.end();
}

void SejtAblak::vissza(int racsIndex) //frissít
{
    this->racsIndex = racsIndex;
```

```
    update(); // Ő meghívja a paint eventet  
}
```

A programunk előre definiált élő sejtekkel indít, mégpedig egy siklókilövővel, ezt hívja meg a SejtAblak konstruktör. Viszont én most nem a siklókilövőt hanem a siklót akarom megmutatni, mivel ez picit talán rövidebb. Láthatjuk, hogy a megfelő rácsbeli cellánkra kell hivatkoznunk és azoknak egyszerűen "ELO" értéket állítunk be.

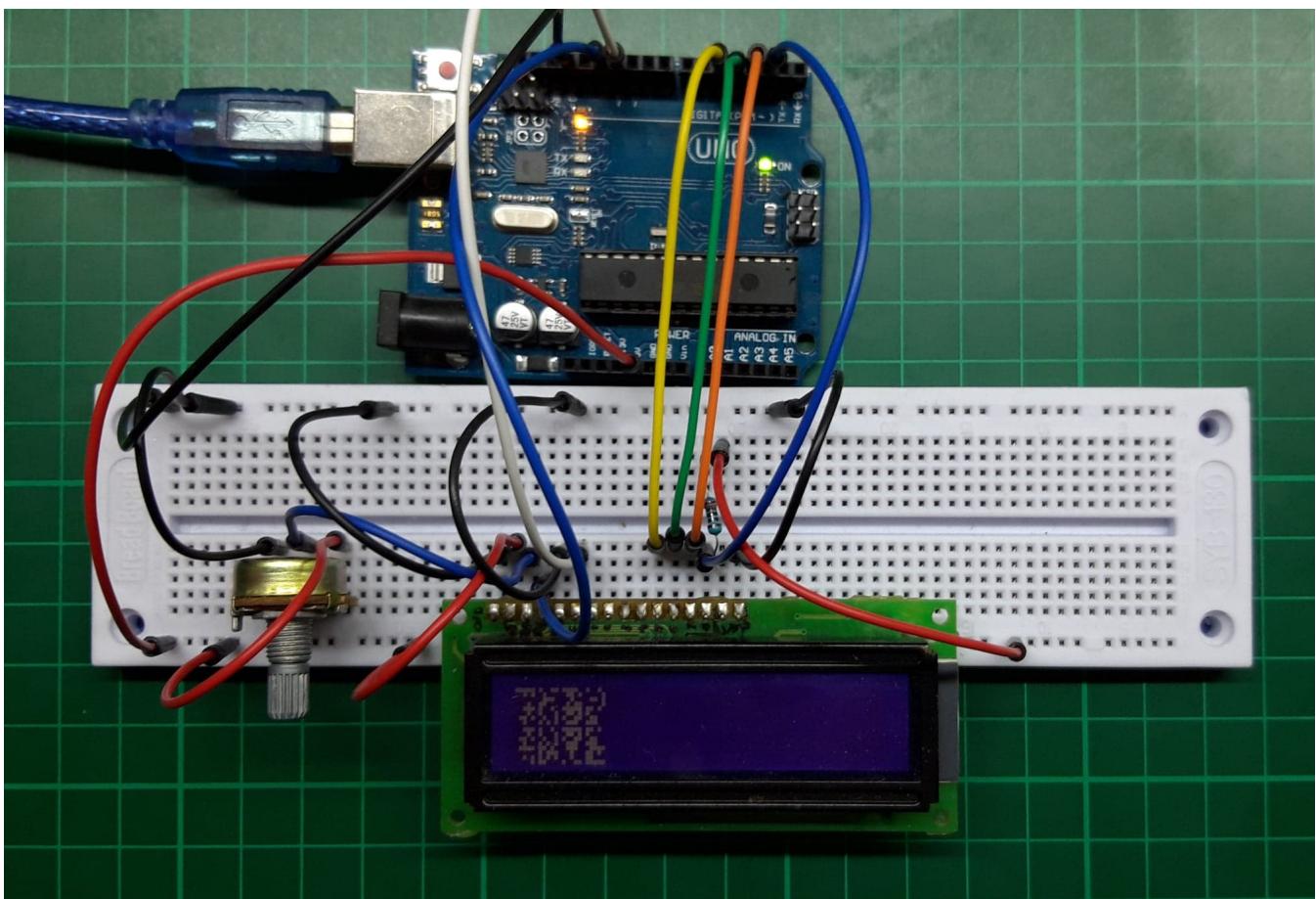
```
void SejtAblak::siklo(bool **racs, int x, int y) {  
  
    racs[y+ 0][x+ 2] = ELO;  
    racs[y+ 1][x+ 1] = ELO;  
    racs[y+ 2][x+ 1] = ELO;  
    racs[y+ 2][x+ 2] = ELO;  
    racs[y+ 2][x+ 3] = ELO;  
  
}
```



Alternatív megoldás:

Ehhez a feladathoz készítettem egy alternatív megoldást is, mégpedig a Conway-féle életjáték Arduino megvalósítását.

Az ötlet onnan jött, hogy még január környékén sikerült beszerezniem egy egész sok minden tartalmazó arduino kit-et. Tehát bőven van mindenféle mikrokontrollerem, szenzorom, kijelzőm, majd amikor elérkeztem ehhez a feladathoz és láttam mennyire népszerű a dolg tudtam, hogy lesz arduino verziója is. És igazam is lett. A YouTube-on méghozzá elég sokféle változata van a dolognak, attól függően milyen kijelzőt használnak. Nekem ez a 1602-es LCD megoldás tűnt a legegyszerűbbnek, de akad 8x8-as LEDmátrix verzió is.

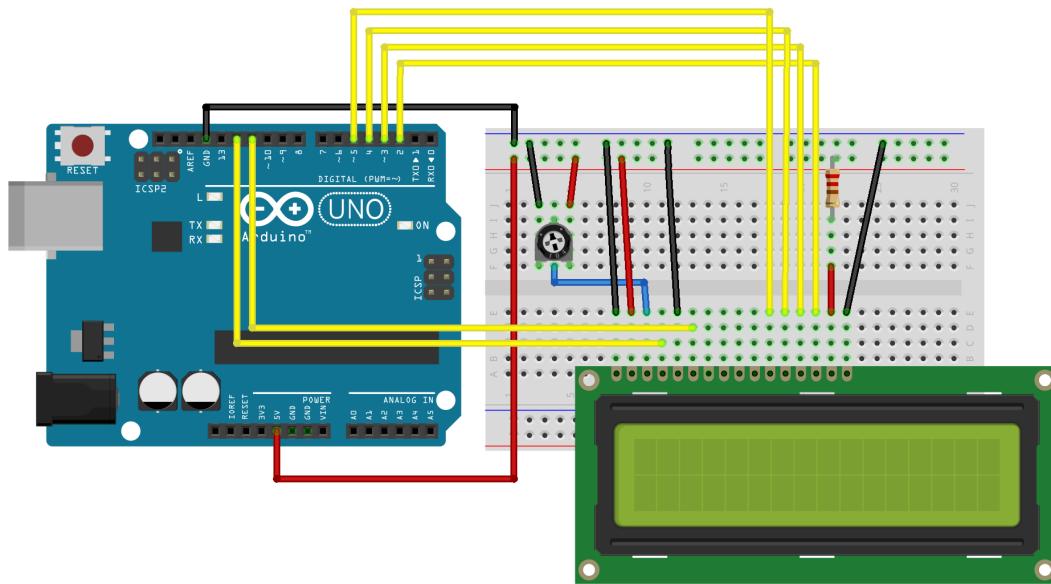


7.1. ábra. Így néz ki az életjáték Arduinoval



Az Arduino megoldás forrása:

A sejtautomata arduino megvalósításához [ezt a videót](#) használtam, megmutatták benne a kapcsolási rajzot illetve a kódot videóban élőben írták. Szerintem azért is nagyon jó videó mert nem csak az életjáték programját mutatja meg hanem az LCD kijelző használatát is.



7.2. ábra. A leggyakoribb LCD bekötési mód amit én is követtem

Amiket felhasználtam az építés során:

- Arduino UNO (lehet más típusú is)
- 1602-es LCD kijelző.
- egy potméter
- 1 db 220 ohm-os ellenállás
- kábelek
- breadboard



Megjegyzés:

A potméterrel tudjuk a kijelző fényerejét állítani.

Az Arduinos megoldás esetében szerintem a legérdekesebb dolog az LCD kijelző használata. Ebből létezik többféle is, én egy 1602-eset használtam. A kijelző neve beszédes ugyanis arra utal hogy 16 oszlopos 2 soros mátrixból épül föl, ezek a mátrixok tartalmazák a pixeleket mégpedig úgy, hogy egy-egy ilyen szekció 5 oszloból és 8 sorból áll. Ebből a felületből a program összesen 20*16, azaz 320 pixelt használ fel.

Az Arduino mikrokontrollekreknek saját programozási nyelve van **ino** kiterjesztésű fájlokat értünk az Arduino programok alatt. Ezek egy `void setup()` előkészítésből és egy `void loop()`-ból állnak. A `void setup()` lényege, hogy előkészítjük magát a mikrokontollert a program futtatására, a `void`

`loop()` pedig a főprogramunk lesz, ez ha belátható időn belül lefut akkor mindenig újrakezdődik, illetve ha a megnyomjuk a "reset" gombot, akkor ezt a fő programot tudjuk az elejtől futtatni. Az `ino` programok esetén mindenig az az első dolgunk, hogy felkészítjük az arduino-t arra, hogy milyen hardware-eket kell kezelnie, megmondjuk neki mely portokat kell használnia. A nyelv maga C alapú, nagyon közel áll hozzá viszont vannak eltérések és sajátosságai, mint az előbb említett fő program is.

Az ino kód forrása

A kódot a videó készítője nyílvánosan tette közzé és [ezen az oldalon](#) bárkinek elérhető. Én nem fogom az egész programot végigrészletezni, a számomra érdekes, fontosnak tartott részeit szeretnémet kiemelni. Mielőtt nekiállunk Arduinot programozni szerezzünk be egy **Arduino IDE** fejlesztői környzetet, ez azért fontos mert innen tudjuk feltölteni a kódunkat magára a mikrokontrollerre. Ha meg akarjuk nyitni a fejlesztői környzetet, akkor ezt rendszergazdaként tegyük meg egyébként nem tudjuk a kódot feltölteni az Arduinonkontra, eddig tapasztalataim alapján ez csak Linux rendszer használata esetén fordul elő.

Itt látható a `void setup()` és a `void loop()` főprogram. A loop-ban tudjuk a `delay()` értékét változtatni, azaz hogy mennyi időközönként frissüljön a rácsunk, millisecundum-ban megadva.

```
void setup() {  
Serial.begin(9600);  
lcd.begin(16, 2);  
GenerateRandomArray();  
GenerateGolArray();  
PrintToLCD();  
}  
  
void loop() {  
delay(500);  
UpdateBoard();  
GenerateGolArray();  
PrintToLCD();
```

A program elején meg kell hívunk a **LiquidCrystal.h** header-t, ami az LCD kezeléséhez szükséges, illetve meg kell adnunk, hogy mely portokra csatlakoztattuk az LCD-t, hogy a mikrokontroller megfelelően tudjon vele kommunikálni illetve deklaráljuk a konstansainkat amik a játéktér méretét befolyásolják.

```
#include <LiquidCrystal.h>  
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);  
  
// Size we will use on the LCD panel  
const int maxRows = 16;  
const int maxCols = 20;
```

A játéktér másolására szolgáló eljárás az alábbi kódcsipetben látható, itt is a C++ programhoz hasonlóan két mátrixot, játékteret használunk.

```
void CopyArray(bool gameBoard2[maxRows][maxCols]) {  
for(int a = 0; a < maxRows; a++)
```

```
{  
    for(int b = 0; b < maxCols; b++)  
    {  
        gameBoard2[a][b] = gameBoard[a][b];  
    }  
}  
}
```

Kiemelném a void UpdateBoard() -t, itt hívjuk meg a másolatkészítő függvényünket, a C++ programhoz hasonlóan itt is két "játékteret" használunk. És látható, hogy lehet végigvizsgálni azt pixelről pixelről pixelre, ami még fontosabb, ennek a kód részletnek a végén találjuk az életjáték szabályait is.

```
void UpdateBoard()  
{  
    bool gameBoard2[maxRows][maxCols] = {};  
  
    // Make a copy of the array  
    CopyArray(gameBoard2);  
  
    // Cycle through the columns of the array down  
    for(int a = 0; a < maxRows; a++)  
    {  
        // Cycle through the rows of the array across  
        for(int b = 0; b < maxCols; b++)  
        {  
            // We assume the cell is dead  
            int numTrue = 0;  
  
            // Used to check the value above and below the cell  
            for(int c = -1; c < 2; c++)  
            {  
                // Used to check the value to the left and right of the ← cell  
                for(int d = -1; d < 2; d++)  
                {  
                    // If we aren't checking the cell itself  
                    if(!(c == 0 && d == 0))  
                    {  
                        // Each time there is a true on the top, bottom,  
                        // right and left of the cell increment numTrue  
                        if(gameBoard2[a+c][b+d])  
                        {  
                            ++numTrue;  
                        }  
                    }  
                }  
            }  
        }  
  
        // If there are less than 2 trues around the cell mark it as ← dead  
        if(numTrue < 2) {gameBoard[a][b] = false;}  
    }  
}
```

```
// If their is exactly 3 live cells mark it as alive
else if(numOfTrues == 3){gameBoard[a][b] = true;}

// If there are more then 3 live cells mark it as dead
else if(numOfTrues > 3){gameBoard[a][b] = false;}
}
}
```

A void PrintToLCD() végzi a kiíratást, esetünkben a képernyő maga az LCD kijelző lesz.

```
void PrintToLCD(){

int startCol = 0;
for(int i = 0; i < (numOfChars/2); i++){

    // Print row 1 on LCD
    PrintChar(i, startCol, 0);

    // Print row 2 on LCD
    PrintChar(i + (numOfChars/2), startCol, 1);
    startCol++;
}
}
```

7.2. Vörös pipacs pokol/ 19 RF

A félév során volt egy feladat amiben, Bátfai Norbert Tanár Úr adott nekünk egy kódot amivel Steve 19 pipacs felszdeésére volt képes és ezt kellett úgy továbbfejlesztenünk, hogy több virágot tudjon szedni, erről fogom csatolni a videónkat. A kód legnagyobb újítása a calcNbrIndex fügvény. ennek az a lényege, hogy amikor Steve elfordul akkor a körülötte érzékelt cuboid nem fordul vele együtt ezért nehéz a tömbb beli elemeket megfelelően hivatkozni, viszont ennek a fügvénynek a segítségével yaw, azaz az elfordulás értékből meg lehet határozni hogy éppen mi van Steve előtt mögött, tőle balra stb.

```
def calcNbrIndex(self):
    if self.yaw >= 180-22.5 and self.yaw <= 180+22.5 :
        self.front_of_me_idx = 1
        self.front_of_me_idxr = 2
        self.front_of_me_idxl = 0
        self.right_of_me_idx = 5
        self.left_of_me_idx = 3
    elif self.yaw >= 180+22.5 and self.yaw <= 270-22.5 :
        self.front_of_me_idx = 2
        self.front_of_me_idxr = 5
        self.front_of_me_idxl =1
        self.right_of_me_idx = 8
```

```
    self.left_of_me_idx = 0
elif self.yaw >= 270-22.5 and self.yaw <= 270+22.5 :
    self.front_of_me_idx = 5
    self.front_of_me_idxr = 8
    self.front_of_me_idxl = 2
    self.right_of_me_idx = 7
    self.left_of_me_idx = 1
elif self.yaw >= 270+22.5 and self.yaw <= 360-22.5 :
    self.front_of_me_idx = 8
    self.front_of_me_idxr = 7
    self.front_of_me_idxl = 5
    self.right_of_me_idx = 6
    self.left_of_me_idx = 2
elif self.yaw >= 360-22.5 or self.yaw <= 0+22.5 :
    self.front_of_me_idx = 7
    self.front_of_me_idxr = 6
    self.front_of_me_idxl = 8
    self.right_of_me_idx = 3
    self.left_of_me_idx = 5
elif self.yaw >= 0+22.5 and self.yaw <= 90-22.5 :
    self.front_of_me_idx = 6
    self.front_of_me_idxr = 3
    self.front_of_me_idxl = 7
    self.right_of_me_idx = 0
    self.left_of_me_idx = 8
elif self.yaw >= 90-22.5 and self.yaw <= 90+22.5 :
    self.front_of_me_idx = 3
    self.front_of_me_idxr = 0
    self.front_of_me_idxl = 6
    self.right_of_me_idx = 1
    self.left_of_me_idx = 7
elif self.yaw >= 90+22.5 and self.yaw <= 180-22.5 :
    self.front_of_me_idx = 0
    self.front_of_me_idxr = 1
    self.front_of_me_idxl = 3
    self.right_of_me_idx = 2
    self.left_of_me_idx = 6
else:
    print("There is great disturbance in the Force...")
```

A program maga úgy működik, hogy Steve az aréna aljából elindul virágot szedni, amikor virágot érzékel a közelében akkor ránéz és kiüti azt. Gyakran előfordul ilyenkor, hogy "csapdába" esik mivel kiüti a pipacs alatt lévő földblokkot, ekkor ki kell ugrania a maga alatt ásott gödörből. Mivel az aréna úgy van megtervezve, hogy minden szinten pontosan 1 db virág található ezért ha egy szinten már megtalálta a virágot és azt felszedte Steve a lvlUp függvényel felmegy a következő szintre. A szinetket a self.y változóval tudjuk meghatározni.

```
self.calcNbrIndex()
```

```
if self.isInTrap(nbr) :
    self.agent_host.sendCommand( "jumpmove 1" )
    time.sleep(.1)
    self.turnFromWall(nbr)
    self.agent_host.sendCommand( "jumpmove 1" )
    time.sleep(.1)
    return True

if self.lookingat == "red_flower":
    print(" A RED FLOWER IS FOUND (lookingat)")
    self.pickUp()
    return True

for i in range(9):
    if nbr[i]=="red_flower" or nbr[i+9]=="red_flower" or nbr[i+18]=="red_flower":
        print("           I CAN SEE A RED FLOWER: ", i, " LEVEL ", self.y)
        if i == self.front_of_me_idx :
            print("F           A RED FLOWER IS RIGTH IN FRONT OF ME")
            self.agent_host.sendCommand( "move 1" )
            time.sleep(.2)
            self.agent_host.sendCommand( "look 1" )
            time.sleep(.2)
            print("Steve <) ", self.lookingat)
            return True
        elif i == self.front_of_me_idxr :
            print("R           A RED FLOWER IS RIGTH IN RIGHT OF ME")
            self.agent_host.sendCommand( "strafe 1" )
            time.sleep(.2)
            return True
        elif i == self.front_of_me_idxl :
            print("L           A RED FLOWER IS RIGTH IN LEFT OF ME")
            self.agent_host.sendCommand( "strafe -1" )
            time.sleep(.2)
            return True
        elif i == 4 :
            self.red_flower_is_mining = True
            print("           I AM STANDING ON A RED FLOWER!!!")

            if self.pitch != 90:
                self.agent_host.sendCommand( "look 1" )
                print("PITCH           I AM STANDING ON A RED FLOWER!!!")
                time.sleep(.3)
            else:
                print("ATTACK           I AM STANDING ON A RED FLOWER!!!")
```

```
        FLOWER!!! LEVEL ", self.y)
    self.pickUp()
    self.agent_host.sendCommand( "look -1" )
    time.sleep(.3)
    return True

else :
    print("           I AM TURNING TO A RED FLOWER")
    self.agent_host.sendCommand( "turn 1" )
    time.sleep(.2)
    return True

if self_LVLUp(nbr):
    print("           LVL UP")

if nbr[self.front_of_me_idx+9]!="air" and nbr[self.front_of_me_idx +9]!="red_flower":
    print("           THERE ARE OBSTACLES IN FRONT OF ME ", nbr[self. front_of_me_idx], end='')

self.turnFromWall(nbr)

else:
    print("           THERE IS NO OBSTACLE IN FRONT OF ME", end='')

    if nbr[self.front_of_me_idx]=="dirt":
        self.agent_host.sendCommand( "move 1" )
        time.sleep(.013)
    else:
        self.turnFromWall(nbr)

return True
```

8. fejezet

Helló, Schwarzenegger!

8.1. Minecraft-MALMÖ

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A könyvben eddig már sok Minecraft Malmö feladattal találkoztunk, a félév során én is már sokat foglalkoztam vele szóval már tudok róla írni pár szót. A Malmö Projekt egy nyílt forráskódú Minecraft Mod, amit azért alkottak meg, hogy bevezetésként szolgáljon az ágensprogramozásba illetve a mesterséges intelligencia alapjaiba is betekintést nyerhetünk. Ha sikerült leszednünk a Microsoft github repójából, nagyon sok dolgot találunk benne. Fontos megemlíteni, hogy több nyelven is programozhatjuk, ezek közül én a Python-t és C++-t használtam már.

Ha csak kipróbálni szeretnénk, akkor én a Python nyelvet ajánlom, ehhez a prebuldelt verziót elég letöltenünk majd a launcClient.sh állományt futtatva a python kódjainkat egy másik terminálból egyszerűen elindíthatjuk. Ha viszont mondjuk C++-t is szeretnénk használni én azt javaslom, hogy magunknak build-eljük forrásból ugyanis nekem másképp több hiba jelentkezett illetve több évfolyamtársam is ugyanúgy járt el mint én.

Fontos megjegyezni, hogy ha nincs túl erős gépünk, akkor nem biztos, hogy problémák nélkül fog futni a Malmö, tehát ha tudunk választani akkor eleve erősebb gépre ajánlom telepítni a biztos futás érdekében. Itt szeretném még azt is megemlíteni, hogy ilyen szempontból kicsit instabil a dolog, többen tapasztaltunk olyat, hogy egy kód a képernyő felvétel során rosszabb teljesítménnyel futott mint amikor nem próbáltuk felvenni, de nem is kell feltétlenül csinálnunk hozzá valami extrát hogy Steve kicsit megmakacsolja magát. Aki ki akarja próbálni ne lepődjön meg rajta, ha ugyanaz a kód ami egyszer hibátlanul lefutott utána más, rosszabb eredményt produkál.

Én egyébként mindenkinél ajánlom, hogy ha tehet és ideje engedi próbálja ki! Szerintem nagyon sok siekrélményt is ad a tanuló programozóknak, ha látják lépésről lépésr, hogyan fut a kódjuk. illetve teljesen más élményt ad mint amikor a lefutott kódunk visszaad egy számot, sokkal "kézzelfoghatóbb" az egész.

8.2. Vörös pipacs pokol/ Javíts a 19RF-en!

Megoldás videó: [Megoldás videó és kódmagyarázat](#)

Ebben a feladatban az előző feladatban tárgyalt kód javítását kellett megoldanunk azaz, több mint 19 virágot kellett összeszednie Steve-nek ugyanazon a pályán. Nekünk minimális változtatásokkal a virágszámot 20-ra sikerült növelnünk.

A legnagyobb módosítás az volt hogy a lvlUp függvényt módosítottuk, hogy jumpmove 1-ek helyett jumpstrafe 1-ekkel ment magassabb szintre. Ennek az volt a lényege hogy a jumpmove 1 esetén előfordult, hogy 2 szintet ugrott fel amivel egy virágot mindenkorábban el is vesztettünk. Ez a változtatás szinte önmagában elég volt a plusz 1 virág megszerzéséhez viszont még ami változtatás volt a programot illetően az a time.sleep () értékek változtaása volt. Ha ilyet csinálunk nagyon kell ügyelni rá, hogy ne menjünk túlzásba, akár századmásodpercek is befolyásolhatják az egész program működését.

```
def lvlUp(self, nbr):
    if self.collectedFlowers[self.y]:
        #self.turnToWall(nbr)
        if nbr[self.left_of_me_idx+9]=="dirt":
            self.agent_host.sendCommand( "jumpstrafe -1" )
            time.sleep(.10)#.2
        elif nbr[self.right_of_me_idx+9]=="dirt":
            self.agent_host.sendCommand( "jumpstrafe 1" )
            time.sleep(.10)#.2
        elif nbr[self.front_of_me_idx+9]=="dirt":
            self.agent_host.sendCommand( "jumpmove 1" )
            time.sleep(.10)#.2
        else:
            self.turnToWall(nbr)
            self.agent_host.sendCommand( "move 1" )
            time.sleep(.10)#.2
    return True
else:
    return False
```

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

A Lisp nyelvvel én ebben a feladatban találkoztam először. Az általam eddig megismert programnyelvektől lényeges különbözik. De nézzük is meg miért lóg ki annyira a sorból!

A Lisp-et 1958-ban alkotta meg John McCarthy, ekkor vált felkapott témává először a mesterséges intelligencia kutatás és a Lisp nyelv szinte egybe olvadt annak fogalmával. A mesterséges intelligencia és a Lisp nyelv kapcsolata megalkotójához köthető, ugyanis John McCarthy aktívan részt vett az ezzel kapcsolatos kutatásokban is. Annak ellenére, hogy nem mai nyelv, napjainkban is van haszna, a GNU Emacs szövegszerkesztő is használja a Lisp-et.

Ami a Lisp nyelvet illeti, fő adatstruktúrája a láncolt lista, innen kapta a nevét is, angolul "List Processing", innen ered a "Lisp" mint rövidítés. gymásba ágyazott láncolt listákkal dolgozik, primitív szintaktika és egyszerű függvények jellemzik, ugyanis minden kifejezést zárójelek közé kell tennünk. Ennek előnye, hogy a számítógép számára könnyen értelmezhető, interpreter, a Python-hoz hasonló módon nem szükséges a Lisp nyelven írt programokat fordítani, értelmező nyelvről beszélünk. Ez azt jeleti a Lisp esetében, hogy a láncolt listákat először beolvassa majd kiértékeli és kiírat. Elsőre nagyon furcsa lehet a szintaktikája, főleg a prefix jelölést értem ez alatt, ami azt jelenti, hogy egy művelet megfogalmazása esetén az operátor előre kerül majd az operandusok azt követik. Ilyennel én először az elsőrendű logikai nyelveknél találkoztam, azok közül is én a Lisp-et az Ar, azaz az elemi aritmetika nyelvéhez tudom hasonlítani. Az Ar nyelv is alapvetőleg prefix, viszont be tudunk vezetni infix jelöléseket is. Amit pedig még mindenki meg szeretnék említeni a Lisp nyelvvel kapcsolatban, hogy nagyon könnyen fejleszthető, ami alatt azt értem, hogy könnyen hozhatunk létre új, saját függvényeket és operátorokat emiatt "személyre szabható" és így napjainkban sem mondhatjuk rá, hogy elavult nelvről beszélünk. Az Ar nyelv, ahogy említettem, számonra nagyon hasonló, az Ar nyelv is alapvetőleg egy db predikátumszimbólummal dolgozik, ami az egyenlőség fogalma és 3 db függényszimbólummal, ami a rákövetkező függvény, a szorzás és az öszeadás, ezekből és 1 db konstans használatával az Ar nyelvben a teljes aritmetika felépíthető elemi lépésekre való lebontással, arról nem is beszélve, hogy a zárójelhasználat is nagyon hasonló.

Egy kis kitekintés az Ar nyelvre:

Lisp nyelv

$$\langle \{t\}, \{=\}, \{s, +, *\}, \{o\} \rangle$$

$x, y, z \dots \rightarrow$ termékesztés ná m változók

$$(x = y) \Leftrightarrow = (x, y)$$

$1 \Leftrightarrow s(o)$	$s(s(o)) * s(s(o)) = s(s(s(o)))$
$2 \Leftrightarrow s(1)$	
$3 \Leftrightarrow s(2)$	$(x \neq y) \Leftrightarrow \neg (x = y)$

kisebb vagy egyenlő mint y :

$$(x \leq y) \Leftrightarrow \exists z (x + z = y)$$

Térjünk rá a faktoriális számító programra! Először is tekintsük meg a faktoriális matematikai háttérét, két képletet alkalmaztunk a programhoz, az első amivel foglalkozni szeretnénk az a rekurzív képlet. Itt az az alapgondolat, hogy $n! = n * (n - 1)!$, tehát, ha írunk egy faktoriális kiszámítására szolgáló függvényt, melynek paramétere n rekurzívan ($n - 1$)-gyel fogjuk meghívni. Szerintem ez volt az egyszerűbb feladat, és megmutatom hozzá a Lisp nyelvű kódot is.

Megjegyzés: mielőtt futtatni szeretnénk a programot, győződjünk meg róla hogy telepítve van a szükséges csomag, amennyiben nincs a

```
sudo apt-get install clisp
```

parancssal tudjuk telepíteni, illetve, ha már megírtuk a kódot, futtatási jogot kell adnunk neki.

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

Láthatjuk, hogy a függvénybelül először azt vizsgáljuk, hogy az n egyenlő-e nullával, ennek az a jelentősége, hogy definíció szerint $0!$ az 1, célszerű ezzel kezdenünk. A függvény további részeiben a rekurzív hívást találjuk az előbb említett képlet szerint. Láttam az interneten olyan megoldást is, ahol az $n = 1$ esetére is külön feltételt vizsgáltak, de ebben a programban az teljesen feleslegessé válna a 0 miatt, ugyanis a rekurzív hívás során ha $n = 1$ akkor $(n - 1)!$ -sal szorzódik ami ugye 1 lesz, tehát $1 * 1$ -et kapunk.

A másik megoldás az iteratív képletet használja, ehhez létre kellett hoznunk egy **f** "lokális változót", majd a **dentimes** függvényvel n alkalommal megszorozza a szám rakkövetkezőjével, tehát az iteratív megoldás,

fordítva halad és minden f -et ad vissza.

A faktoriális **függvény** formális definíciója:

$$n! = \prod_{k=1}^n k \quad \text{ minden } n \geq 0 \text{ egész számra.}$$

Iteratív képlet a Wikipédiáról



Megoldás videó:

A program futásáról készült videóm itt található.

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

A programomat annyival még kiegészítettem, hogy az enyém bekéri az **n**-t, tehát nem a kódban kell meghatároznunk meddig szeretnénk a faktoriálist számolni.

Mivel a program rövid ezért ide beillesztem az egész forrást:

```
#!/usr/bin/clisp

(format t "Irj be egy szamot!~%")
(setq n (read))

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

(defun faktorialis_rekurziv (n)
  (if (= n 0)
      1
      (* n (faktorialis_rekurziv(- n 1)))) )

(format t "Rekurziv szamitas:~%")

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

```
(format t "Iterativ szamitas:~%")  
  
(loop for i from 0 to n  
      do (format t "~D! = ~D~%" i (faktorialis_iterativ i)))
```

A feladat megoldásához használtam [Besenczi Renátó videóját](#) illetve az alábbi oldalakról gyűjtöttem információkat: <http://ait.iit.uni-miskolc.hu/~dudas/MIEAok/MIea5.PDF> <http://nyelvek.inf.elte.hu/leirasok/Lisp/index.php?chapter=1>

9.2. Malmö kód továbbfejlesztése

Megoldás videó:



Ez a program, az előző kettő fejezetben tárgyalt programokon alapszik, viszont itt már sok fejlesztést tettünk bele. Ezzel a kóddal neveztem én és Czanik András csapatban a RFH III. versenyre és a tabellán 16. helyre kerültünk vele. A [kvalifikációs videón](#) itt található. Illetve készítettünk egy bemutatkozós, kódmagyarázós videót is [Ami](#) itt elérhető.

Steve átlagosan 27 virágot tud összegyűjteni, ez elég volt a kvalifikációhoz. A program lényege hogy Steve gyorsan felszalad a 30. szintre (ami virágszámban 28-at jelentne) és onnan kezdve csigavonalban halad lefelé úgy hogy az aréna fala tőle balra van. Amint virágot érzékel azt felveszi és jobbra ugrással az eggyel lentebbi szintre megy, ennek az a lényege, hogy olyan szinten amelyen már vett fel virágot ne időzzön tovább. Viszont nézzük meg a kódot!

Az előző kódokból átvett lényegi rész a **calcNbrIndex()** függvény. Ennek az a lényege, hogy ugye mint azt már tudjuk Steve egy cuboid-ban lát, viszont amikor elfordul egy sarokban, a cuboid nem fordul együtt vele ezért ugyanazokra a tömbbeli elemekre hivatkozva nem tudjuk elérni, hogy megfelelően reagáljon. Ez a függvény a **yaw** értékből, ami Steve elfordulását hivatott leírni, számolja ki, hogy a cuboid mely eleme lesz az amit keresünk adott elfordulás esetén.

```
def calcNbrIndex(self):  
    if self.yaw >= 180-22.5 and self.yaw <= 180+22.5 :  
        self.front_of_me_idx = 1  
        self.front_of_me_idxr = 2  
        self.front_of_me_idxl = 0  
        self.right_of_me_idx = 5  
        self.left_of_me_idx = 3  
    elif self.yaw >= 180+22.5 and self.yaw <= 270-22.5 :  
        self.front_of_me_idx = 2  
        self.front_of_me_idxr = 5  
        self.front_of_me_idxl = 1  
        self.right_of_me_idx = 8  
        self.left_of_me_idx = 0  
    elif self.yaw >= 270-22.5 and self.yaw <= 270+22.5 :  
        self.front_of_me_idx = 5
```

```
    self.front_of_me_idxr = 8
    self.front_of_me_idxl = 2
    self.right_of_me_idx = 7
    self.left_of_me_idx = 1
elif self.yaw >= 270+22.5 and self.yaw <= 360-22.5 :
    self.front_of_me_idx = 8
    self.front_of_me_idxr = 7
    self.front_of_me_idxl = 5
    self.right_of_me_idx = 6
    self.left_of_me_idx = 2
elif self.yaw >= 360-22.5 or self.yaw <= 0+22.5 :
    self.front_of_me_idx = 7
    self.front_of_me_idxr = 6
    self.front_of_me_idxl = 8
    self.right_of_me_idx = 3
    self.left_of_me_idx = 5
elif self.yaw >= 0+22.5 and self.yaw <= 90-22.5 :
    self.front_of_me_idx = 6
    self.front_of_me_idxr = 3
    self.front_of_me_idxl = 7
    self.right_of_me_idx = 0
    self.left_of_me_idx = 8
elif self.yaw >= 90-22.5 and self.yaw <= 90+22.5 :
    self.front_of_me_idx = 3
    self.front_of_me_idxr = 0
    self.front_of_me_idxl = 6
    self.right_of_me_idx = 1
    self.left_of_me_idx = 7
elif self.yaw >= 90+22.5 and self.yaw <= 180-22.5 :
    self.front_of_me_idx = 0
    self.front_of_me_idxr = 1
    self.front_of_me_idxl = 3
    self.right_of_me_idx = 2
    self.left_of_me_idx = 6
else:
    print("There is great disturbance in the Force...")
```

Az alábbi rész a felszaladásért felelős for ciklust tartalmazza, illetve előkészítjük Steve-t a csiga vonalban történő mozgás megkezdésére. Fontos volt hogy beállítsuk a program elején, hogy folyamatosan lefelé nézzen, így könnyebben üti ki a virágokat, nem kell arra várnunk, hogy rá is nézzen azokra.

```
i = 0
for i in range(30):
    self.agent_host.sendCommand( "jumpmove 1" )
    time.sleep(0.1)
    self.agent_host.sendCommand( "move 1" )
    time.sleep(0.1)

    self.agent_host.sendCommand( "turn 1" )
```

```
time.sleep(0.2)
self.agent_host.sendCommand( "look 1" )
self.agent_host.sendCommand( "look 1" )
```

A program többi része úgy működik, hogy különböző feltételek teljesülését vizsgáljuk, ezeket bemutatom. Az alábbi kódcsipetben Steve a sarkokat figyeli, tudnia kell, hogy mikor forduljon jobbra egy színen.

```
if nbr[self.front_of_me_idx+9] == "dirt" and nbr[self. ←
left_of_me_idx+9] == "dirt":
    self.agent_host.sendCommand( "turn 1" )
    time.sleep(0.2)

else:
    print("There is no corner")
```

Tettünk a programba egy lávakerülésre használt részt, amennyiben Steve lávát érzékel, jobbra ugrik kettőt, az aréna belseje felé. Ezt a program futása során ritkán használta és az is a potenciálisan megszerezhető virágok elvesztésével járt, viszont meg tudta úszni a lávával való találkozást.

```
if nbr[self.left_of_me_idx+18]== "flowing_lava" or nbr[self. ←
front_of_me_idx+9]== "flowing_lava" or nbr[self.front_of_me_idx ←
+18]== "flowing_lava":
    self.agent_host.sendCommand( "strafe 1" )
    time.sleep(0.1)
    self.agent_host.sendCommand( "strafe 1" )
    time.sleep(0.1)
```

Az következő részt azért kellet a programba tenni, mert Steve ha virágot talál gyakran aföldet is kiüti alólá és egyben maga alól, "csapdába kerül" ilyenkor ki kell ugrania belőle. Kiegészítettük jobbra ugrásokkal is mivel ilyenkor gyakran egy szinttel lentebb kell folytatnia a mozgását.

```
if nbr[self.front_of_me_idx+9]== "dirt" and nbr[self.right_of_me_idx ←
+9]== "dirt" and nbr[self.left_of_me_idx+9]== "dirt":
    print("      IT'S A TRAAAAAP")
    self.agent_host.sendCommand( "jumpmove 1" )
    time.sleep(0.1)

if nbr[self.front_of_me_idx+9]== "dirt" and nbr[self.left_of_me_idx ←
+9]== "air":
    self.agent_host.sendCommand( "turn 1" )
    time.sleep(0.2)
    self.agent_host.sendCommand( "jumpstrafe 1" )
    time.sleep(0.1)
    self.agent_host.sendCommand( "jumpstrafe 1" )
    time.sleep(0.1)
```

Elérkeztünk Steve virágszedéséhez, ha virágot talál meghívódik a **pickup()** függvény, azaz kiüti a virágot, itt találunk egy kevés késleltetést ami azért kell, hogy a kiütött virágot fel is tudja szedni, majd kettőt jobbra ugrik, tehát megy a következő szintre.

```
if nbr[self.front_of_me_idx+9]=="red_flower":  
    print("      VIRAGOT SZEDEK!!")  
    #self.agent_host.sendCommand( "move 1" )  
    #time.sleep(0.2)  
    self.pickUp()  
    time.sleep(0.2)  
    self.agent_host.sendCommand( "jumpstrafe 1" )  
    time.sleep(0.1)  
    self.agent_host.sendCommand( "jumpstrafe 1" )  
    time.sleep(0.1)
```

Ezekben a sorokban pedi az általános eseteket találjuk, amikor Steve-nek nincs más dolga mint menni és lépésenként ellenőrzni, hogy mi található a környezetében, hogy megfelően tudjon reagálni. Tehát az alap esetben egyet lép minden előre.

```
if nbr[self.front_of_me_idx+9] == "air" and nbr[self. ←  
front_of_me_idx] == "dirt":  
    self.agent_host.sendCommand( "move 1" )  
    time.sleep(0.05)  
  
if nbr[self.front_of_me_idx+9] == "air" and nbr[self. ←  
front_of_me_idx] == "air":  
    self.agent_host.sendCommand( "move 1" )  
    time.sleep(0.05)
```

Ebben a feladatban [Szoboszlai István](#) volt a tutorom, egész pontosan ami a C++ megvalósítást illeti. Mivel a C++ megoldás szinte teljesen ugyanaz mint a Python-os ezért ott nem fogom taglalni a kódöt csak beíllsztem ide annak lényegi részeit. És a C++ futásról készült videó pedig [itt található](#).

```
for (int i = 0; i < 30; i++)  
{  
    agent_host.sendCommand("jumpmove 1");  
    boost::this_thread::sleep(boost::posix_time::milliseconds(100));  
    agent_host.sendCommand("move 1");  
    boost::this_thread::sleep(boost::posix_time::milliseconds(100));  
}  
agent_host.sendCommand("turn 1");  
boost::this_thread::sleep(boost::posix_time::milliseconds(200));  
agent_host.sendCommand("look 1");  
agent_host.sendCommand("look 1");  
  
// main loop:  
do {  
  
    if(world_state.number_of_observations_since_last_state != 0)
```

```
{  
    std::stringstream ss;  
    ss << world_state.observations.at(0).get()->text;  
    boost::property_tree::ptree pt;  
    boost::property_tree::read_json(ss, pt);  
  
    vector<std::string> nbr3x3;  
  
    nbr3x3 = GetItems(world_state,ss,pt);  
    for(vector< boost::shared_ptr< TimestampedString>>::iterator it ←  
        = world_state.observations.begin();it !=world_state. ←  
        observations.end();++it)  
    {  
        boost::property_tree::ptree pt;  
        istringstream iss((*it)->text);  
        boost::property_tree::read_json(iss, pt);  
  
        //string x =pt.get<string>("LineOfSight.type");  
        //string LookingAt =pt.get<string>("XPos");  
        //y = pt.get<double>("YPos");  
        yaw = pt.get<double>("Yaw");  
        //cout<<" Steve's Coords: "<<y<<" "<<yaw<<" "<<"RF:"<<virag;  
    }  
  
    calcNbrIndex();  
  
    //checking corners  
  
    if (nbr3x3[front_of_me_idx+9] == "dirt" && nbr3x3[ ←  
        left_of_me_idx+9] == "dirt")  
    {  
        agent_host.sendCommand("turn 1");  
        boost::this_thread::sleep(boost::posix_time::milliseconds ←  
            (300));  
    }  
    else  
        cout << "\nThere is no corner";  
  
    //checking lava  
    if (nbr3x3[left_of_me_idx+18] == "flowing_lava" || nbr3x3[ ←  
        front_of_me_idx+9] == "flowing lava"  
        || nbr3x3[front_of_me_idx+18] == "flowing_lava")  
    {  
        agent_host.sendCommand("strafe 1");  
        boost::this_thread::sleep(boost::posix_time::milliseconds ←  
            (100));  
        agent_host.sendCommand("strafe 1");  
        boost::this_thread::sleep(boost::posix_time::milliseconds ←  
            (100));  
    }  
}
```

```
//checking traps
if (nbr3x3[front_of_me_idx+9] == "dirt" && nbr3x3[ ↵
    right_of_me_idx+9] == "dirt" ↵
    && nbr3x3[left_of_me_idx+9] == "dirt")
{
    cout << "\nIt's a TRAAAAP";
    agent_host.sendCommand("jumpmove 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds ↵
        (100));
}

if (nbr3x3[front_of_me_idx+9] == "dirt" && nbr3x3[ ↵
    left_of_me_idx+9] == "air")
{
    agent_host.sendCommand("turn 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds ↵
        (200));
    agent_host.sendCommand("jumpstrafe 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds ↵
        (100));
    agent_host.sendCommand("jumpstrafe 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds ↵
        (100));
}

//finding flower
if (nbr3x3[front_of_me_idx+9] == "red_flower")
{
    cout << "\nVIRÁGOT SZEDEK!!";
    agent_host.sendCommand("attack 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds ↵
        (230));
    boost::this_thread::sleep(boost::posix_time::milliseconds ↵
        (200));
    agent_host.sendCommand("jumpstrafe 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds ↵
        (100));
    agent_host.sendCommand("jumpstrafe 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds ↵
        (100));
}

if (nbr3x3[front_of_me_idx+9] == "air" && nbr3x3[ ↵
    front_of_me_idx] == "dirt")
{
    agent_host.sendCommand("move 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds ↵
        (50));
}
```

```
if (nbr3x3[front_of_me_idx+9] == "air" && nbr3x3[ ↵
    front_of_me_idx] == "air")
{
    agent_host.sendCommand("move 1");
    boost::this_thread::sleep(boost::posix_time::milliseconds ↵
        (50));
}
```

10. fejezet

Hello, Gutenberg!

10.1. 10.1. Programozási alapfogalmak

A programozási nyelveknek alapvetőleg 3 szintjét tudjuk megkülönböztetni: gépi nyelv, assemly szintű nyelv és a magas programozási nyelvek. A gépi nyelv jelentősége az hogy a CPU-k saját gépi nyelvvel rendelkeznek és csak azon a nyelven írt programokat tudják futtatni. Kezdő programozók tapasztalhatnak olyat ha pl .c kiterjesztésű fájlt próbálnak futtatni hogy szintaktikai hibát dob vissza mondjuk zárójel használatra, pontosan ezért van szükség fordítóprogramokra. Én linuxot használok így mondjuk gcc prog.c -o prog parancssal tudom előállítani a forráskódomból a futtatható tárgyprogramot. Amikor ezt tesszük a fordítóprogram először egy lexikális elemzést hajt végre majd ezt követően szintaktikai és szemantikai elemzést és az utolsó lépésekben kódot generál. Ezzel szemben az interpretens technika az utolsó lépést kihagyja és soronként elemzi és hajtja végre a megadott lépéseket, ilyen technikával tudunk futtatni Python nyelven írt programokat. Assemly szintű nyelv: ezekről a köny csak említés szintjén ír, ezek a nyelvek a gépi kód és a magas szintű programozási nyelvek közti átmenetet képviselik. Magas szintű programozási nyelvek:

10.2. Keringhan.Ritchie: A C programozási nyelv

Az első Alapismeretek c. fejezetet olvastam el. Mivel én már korábban is olvastam a könyvből, illetve főleg példaprogramokon keresztül mutatja be a programozás alapjait, ezért számomra viszonylag kevés új dolog volt benne főleg említés szintjén számolok be az olvasott témakról.

Először egy klasszikus Hello Word! (A könyvben "Figyelem emberek!") program bemutatásával foglalkozik a könyv és vázolja is mire van szükségünk ahhoz hogy ezt futattni tudjuk, itt megemlíti, hogy valamilyen környezetben írjuk meg a forráskódot, az elengedhetetlen fordítást is említi illetve hogy ki kell találnunk, hogy a szöveget milyen képernyőn szeretnénk kiirva látni. Megjegyzem Linuxban mindezt terminálból végrehajthatjuk, pl a nano szövegszerkesztő környezetben illetve a beépitett gcc-ven tudunk is fordítani. Bemutatásra kerül a jól ismert "main()" függvény, amibe a fő programunk kerül.

A következő pontban a változók kerülnek terítékre és azok deklarálása, ugye C-ben meg kell adnunk minden típusú változót szeretnénk létrehozni viszont ekkor még nem szükséges neki kezdőértéket adni mint Pythonban, ilyenkor változó tartalma gyakorlatilag memóriaszemét. A különböző típusú változók (a példában int és float) használatát és a while ciklus működését a köny egy Celsius-Farenheight változóprogramon keresztül mutatja be, illetve a változó típusokra való hivatkozás módját is szemlálteti (printf() függvény

használata). Ugyanezena példán keresztül bemutatja a for ciklus működését és szintaxisát is, hogy az olvasó láthassa ugyanúgy célra vezető lehet midnkét ciklus használata, viszont itt fontos megjegyezni, hogy problema specifikus mikor melyiket érdemes használni. Ezen kívül megemlíti a könyv a szimbolikus állandókat melyekkel bizonyos objektumokra hivatkozhatunk a "#define" kifejezés segítségével.

A következő pont a karakterek ki és bevitelével foglalkozik a "getchar()" és "putchar()" függvények bemutatásán keresztül. A könyv is kiemeli, hogy ezeket a müveleteket a "printf()" és "scanf()" függvényekkel is megtehetjük, az hogy melyeket használjuk megint probála specifikus. Ebben a bekezdésben szó esik az ASCII karakterkészletről is. A megoldandó példaprogram itta a különböző karakterek megszámlálásáról szól az előző téma körben tárgyalt ciklusok segítségével. A következő példaprogram bemenetre érkező szavak számlálását várja el, a szavakat szóköz választja el és az adott szavak első karakterét kell felismernie a kész programozóknak ez alapján annyi szót számol ahány szókezdő karaktert talál. Ekkor felmerül az elágazás kérdése is. A könyv bemutatja az if/else szintaxisát és az "andandhelye" azaz "és", illetve a "||" azaz "vagy" operátor működését, amik elengedhetetlenek a különböző feltételek pontos megfogalmazásához.

A karakterek téma köré utána könyv a tömböket taglalja, ezeket C-ben szerintem egyszerü kezelni, szerencsére a számosztuk is 0-ról indul mint általában a magasabb szintű programozási nyelveknél megszokott. Fontosnak tartom megjegyezni hogy C-ben a stringek leírására gyakran a karaktertömbök használata a megoldás ami furcsa lehet annak aki korábban mondjuk C++-ban programozott.

Az utolsó téma kör a függvények meghívásával, paraméterinek megadásáról és általános használatukról szól. Le van irva hogyan tudunk visszatérésé értékekkel pontosan dolgozni, egyik függvényét átadni másik függvénynek, általános használatukat a könyv jól szemlélteti.

Összefoglalás: Az Alapismeretek c. fejezet valóban a legelemebb szinten mutatja be a C nyelv használatát. Az olvasás során felmerülő témaik majdnem mindegyikével találkoztam már korábban. Kezdő programozóknak ajánlanám, én is mikor elkezdtettem C nyelven programozni ezt a könyvet kezdtettem el olvasgatni és valóban hasznos is volt.

10.3. 10.4. Python bevezetés

A Python programozási nyelv, amely 2018-ra a legnépszerűbb programozási nyelvvé vált, Guido van Rossum nevéhez fűződik aki azt 1990-ben alkotta meg. A kezdő programozóknak a Python ideális nyelv számos előnye maitt, viszonylag könnyen elsajátítható, objektum orientált és platformfüggetlen is. A Python olyan magas szintű programozási nyelv amely jobban hasonlit a természetes angol nyelvhez mint a többi magas szintű programozási nyelv, tömör, jól olvasható kódokat írhatunk és nincs szükség zárójelezésre sem. Fontos megjegyezni hogy alkalmazásfejlesztéshez is ideális, gyorsabban lehet fele dolgozni mint pl a C/C++ esetében mivel itt kimarad a fordítási fázis. A forráskódot az interpreternek köszönhetően azonnal futathatjuk és lényegesen gyorsabbá válik a programozási folyamat.

SZINTAXIS: Ha már a Python nyelvről irok, fontos megemlíteni a szintaxist. A nyelv szintaktikájának egyik legfontosabb tulajdonsága hogy behúzás alapú. Pl.: ha irunk egy for ciklust a ciklusban lévő állítások egy behúzással (tab = 4 szóköz) benteb vannak mint a for ciklus feje. Erre emiatt a nyelv nagyon érzékeny is, ha nem megfelelően vannak a behúzásaink elrendezve egyből kapunk hibákat. A C/C++ nyelvekhez képest, amelyeket én is használtam korábban, talán a legszembetűnőbb különbség a ";" hiánya, illetve a ciklusok és függvények után kapcsos zárójelek helyett használatos ":". Ha kommentelni szeretnénk akkor tudunk soronként a "#" -el vagy több sort 3 db aposztróffal melyeket a komment elejére és végére kell illesztenünk.

A könyvben olvashattam a tipusokról és változókról. Ebben a programozási nyelvben is a szokásos tipusokat tudjuk használni (számok, azon belül egyszek tizedes/lebegőpontos törtek, stringek stb.) viszont jelentős különbség, hogy a változóinkat nem kell deklarálnunk, futás közben az interpreter a egadott kezdőérték alapján felismeri milyen tipusu változóval van dolga. Változók alatt az egyes objektumokra mutató referenciákat értjük. Vannak lokális és globális változók. Ha az adott egy függvényben vesszük fel akkor lokális változóról beszélünk, ha globálissá szeretnén tenni a függvény elején a változó felvételekor használnunk kell a "global" kulcsszót. Különböző változótipusok között szabad a konverzió, tehát tudunk stringől számot képezni stb. Szerencsére a nyelv sok beépített függvényt tartalmaz így a változóinkat könnyen tudjuk kezelni. A tömbök számozása 0-tól kezdődik.

Fontos megemlíteni a nyelv eszközeit. Az első az elágazás, itt is a jól ismert if/elif/else kulccsszavakat használjuk, működését tekintve olyan mint bármely más programozási nyelvben, a sztaxisában pedig a kettőspont használata az ami eltérést mutat más nyelvekhez képest. A ciklusoknál hasonló a helyzet, a while ciklusnál is a feltétel mögötti kettőspontot használjuk a kapcsos zárójelek helyett. Eltéres a for ciklusban van leginkább ugyanis annak 2 fajtájával találkozhatunk. Az első a megszokott for ciklus, szntaxist illetően eltér de ezen kivül nincs különbség. A második fajta for cílus a "range()" függvényt használja, mely futtatása alatt egy listát ad vissza. A függvényeinkhez a "def" kulcsszót használhatjuk de itt is lányegbeli eltérést nem igazán tapasztalhatunk. Amikor az osztélyokról és objektumokról olvastam láttam magam előtt a "Class Steve:"-et és annak attribútumait. Mivel én korábban nem találkoztam objektum orientált programozással a gyakorlatban ezt még mindig tanulom és próbálok a gyakorlatba beépíteni de mindenkepp hasznos volt olvasnom erről is a könyvben.

10.4. 10.2. Szoftverfejlesztés C++ nyelven

A C++ egy magas szintű programozási nyelv általános célú frlhasználásra. Az első verziója 1983-ban jelent meg. jelentős különbség elődjéhez, a Chez klpest, hogy a C++ objektum orientált. C-hez nagyon közel áll, egy C++ fordító jó esélyel tudja a C-t is fordítani, hiába vannak benne pl C kulccsszavak vagy éppen C header fájlok. A 2-6. oldalon a C nyelvhez képesti továbbfejlesztésről olvashatunk, meg is említi a könyv hogy az itt bemutatott kódokat már csak C++ fordítóval fordulnak.

Először a függvények ekrülnek megemlítésre. A C nyelvben egy üres paraméterlistájú függvényt tetszőleges számú paraméterrel hivhatunk meg, mig C++-ban az üres paraméterlistára a "void" kifejezés utal. Ezen kívül jelentős különbség, hogy ha nem diniálunk visszatérési értéket C-ben alapértelmezettként int tipusú visszatérési értéket kapunk vissza mig C++-ban fordítási hibát. A "main()" függvény estén nem kötelező a return utasítást használni mivel ha a kód sikeresen a fordul annak visszatérési értéke mindenkepp 0 lesz.

A bool tipust szükségszerű kiemelnem. Fontos különbség a két nyelv között, hogy C++-ban van bool tipusú változó amely "true" azaz igaz vagy "false" azaz hamis logikai értékkel rendelkezhet. C nyelvben ezt a logikai értéket int tipussal tudjuk kifejezni ami lehet 0 vagy 1.

A C-ben és a C++-ban is lehet használni több bájtos stringeket, A C++-szal ellentétben ami rednelkezik ezek kezeléséhez szükséges függvényekkel, C-ben ahoz hogy a wchar_t típiust tudjuk használni meg kell hivnunk vagy az stdlib.h, a z stddef.h vagy a wchar.h header fájlokat.

C++-ban mindenhol lehet változókat deklarálni ahol utasítást tudunk megadni. Ez azért hasznos mert ott tudjuk deklarálni ahol használni szeretnénk így átláthatóbb lesz a kódunk és kisebb esélyel feledkezünk meg a változóinkról. Pl.: ha egy for ciklust irunk annak fejében tudjuk deklarálni a ciklusváltozót, ekkor annak csak a ciklus törzsére lesz hatása.

A fordítás, futattás, nyomonkövetésről: Ez a bekezdés a könyvben számonomra nem volt túl hasznos mivel itt a fejlesztői környezet fordításra és futtatásra való felhasználása volt kifejeve, ráadásul mint exe fájl Windows alatt. Én általában Linux rendszert használok és az elmentett .c vagy .cpp kiterjesztésű fájljaimat a rendszerbe beépítettem gcc és g++ fordítókkal tudom fordítani terminálon keresztül, tehát a hibaüzenetek is a terminál képernyőjén jelennek meg. Egy helyen tudom a sikeres fordítás után futtatni a kész futatható állományaimat. Szerintem ezzel a módszerrel fordítani mint ahogy az a könyvben van bemutatva de persze érdemes ismerni minden két módszert.

III. rész

Második felvonás

**Bátf41 Haxor Stream**

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

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

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

11.4. C

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

11.5. C++

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

11.6. Lisp

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

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

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