

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Garbóczy Vajk

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás	
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>
WRITTEN BY	Garbóczy, Vajk	2019. május 8.

REVISION HISTORY

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

Ajánlás

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

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

DRAFT

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	7
2.3. Változók értékének felcserélése	9
2.4. Labdapattogás	10
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	12
2.6. Helló, Google!	14
2.7. 100 éves a Brun téTEL	15
2.8. A Monty Hall probléma	15
3. Helló, Chomsky!	17
3.1. Decimálisból unárisba átváltó Turing gép	17
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	17
3.3. Hivatalos nyelv	18
3.4. Saját lexikális elemző	19
3.5. l33t.l	20
3.6. A források olvasása	21
3.7. Logikus	22
3.8. Deklaráció	23

4. Helló, Caesar!	25
4.1. double ** háromszögmátrix	25
4.2. C EXOR titkosító	26
4.3. Java EXOR titkosító	28
4.4. C EXOR törő	30
4.5. Neurális OR, AND és EXOR kapu	33
4.6. Hiba-visszaterjesztéses perceptron	34
4.7. Gutenberg olvasónaplók	35
4.8. Juhász István - MAGAS SZINTŐ PROGRAMOZÁSI NYELVEK 1	35
4.9. Kernigen and Ritchie: A C programozási nyelv	35
4.10. Benedek Zoltán - Levendovszky Tihamér: Szoftverfejlesztés C++ nyelven	36
5. Helló, Mandelbrot!	37
5.1. A Mandelbrot halmaz	37
5.2. A Mandelbrot halmaz a std::complex osztálytal	39
5.3. Biomorfok	40
5.4. A Mandelbrot halmaz CUDA megvalósítása	41
5.5. Mandelbrot nagyító és utazó C++ nyelven	41
5.6. Mandelbrot nagyító és utazó Java nyelven	41
6. Helló, Welch!	42
6.1. Első osztályom	42
6.2. LZW	45
6.3. Fabejárás	48
6.4. Tag a gyökér	49
6.5. Mutató a gyökér	53
6.6. Mozgató szemantika	53
7. Helló, Conway!	54
7.1. Hangyaszimulációk	54
7.2. Java életjáték	58
7.3. Qt C++ életjáték	61
7.4. BrainB Benchmark	64

8. Helló, Schwarzenegger!	65
8.1. Szoftmax Py MNIST	65
8.2. Mély MNIST	66
8.3. Minecraft-Malmö	66
9. Helló, Chaitin!	67
9.1. Iteratív és rekurzív faktoriális Lisp-ben	67
9.2. Gimp Scheme Script-fu: króm effekt	68
9.3. Gimp Scheme Script-fu: név mandala	68
III. Második felvonás	73
10. Helló, Arroway!	75
10.1. A BPP algoritmus Java megvalósítása	75
10.2. Java osztályok a Pi-ben	75
IV. Irodalomjegyzék	76
10.3. Általános	77
10.4. C	77
10.5. C++	77
10.6. Lisp	77

Ábrák jegyzéke

2.1. 4 mag 100%-on	6
2.2. A programunk fut, de alszik	7
5.1. Mandelbrot halmaz	37
5.2. Fraktál a természetben	38
5.3. Komplex mandelbrot halmaz	40
5.4. Biomorf	41
7.1. Hangyák ételt keresnek	54
7.2. 'A' hangya ételt talál	55
7.3. Többek is csatlakoznak az erős feromon útra	55
7.4. A hangyák útnak indulnak	56
7.5. Erősebb és gyengébb útvonalak alakulnak ki	57
7.6. Program összetevők	58
7.7. Sejtautomata	59
7.8. Sejtautomata 2	60
7.9. Sikló	61
7.10. Siklókilövés	62
7.11. Siklókilövés 2	63
7.12. Életjáték a google keresőben	63
8.1. A betanítás folyamata	66
9.1. Eljárásbongésző	69
9.2. Saját mandalám	72

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 mászt 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

DRAFT

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

DRAFT

**Bátf41 Haxor Stream**

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

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás forrása

100%-os mag használatot könnyen elérhetünk bármely egyszerű végtelen ciklussal, melynek végtelenségét az adja, hogy a ciklus futási feltételét úgy adjuk meg hogy az mindig igaz legyen. Erre számos megoldás létezik, pl:

```
int main() {  
  
    while(1<2) {}  
  
    return 0;  
}
```

Ezzel az egyszerű programmal elérhetjük, hogy a programunk egy magot 100%-ban dolgoztasson meg. De mi van ha mi azt szeretnénk, hogy az összes magot 100%-ban használja ki? Nos, természetesen futathatók a programot annyiszor ahány magunk van, vagy pedig segítségül hivhatjuk az OpenMP könyvtár szolgáltatásait.

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

Ezt a kódot a gcc fordító -fopenmp kiegészítéssel ellátva fordítjuk, majd futtatjuk. Ekkor láthatjuk, hogy az összes magunk (közel vagy ténylegesen) 100%-os terhelésen fut.

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+
2015	vajk	20	0	4376	756	692	R	85,1	0,0	0:51.20
2012	vajk	20	0	4376	756	692	R	83,4	0,0	0:54.85
2013	vajk	20	0	4376	768	704	R	83,1	0,0	0:54.33
2014	vajk	20	0	4376	748	684	R	77,2	0,0	0:52.56
1047	vajk	20	0	4231948	360628	111740	S	65,9	4,4	0:43.41

2.1. ábra. 4 mag 100%-on

Nézzük hát, hogy hogyan érhetnénk el azt, hogy egy végtelen ciklust futtatunk ami 0%-ban használja a CPU-nkat. Nos, ezt a sleep() függvény segítségével fogjuk elérni. A sleep függvény ahogy neve is mutatja, altatja az adott programot paraméterként megadott másodpercnyi időre. Tehát, ha egy végtelen ciklusban sleep(1)-et alkalmazunk, a programunk folyamatosan aludni fog (ha csak ennyit tartalmaz a ciklus).

```
#include <unistd.h>
int
main () {

    for (;;)
        sleep (1);
    return 0;
}
```

```

File Edit View Search Terminal Help
top - 21:35:08 up 19 min, 1 user, load average: 1,07, 2,00, 1,31
Tasks: 198 total, 2 running, 149 sleeping, 0 stopped, 0 zombie
%Cpu0 : 4,6 us, 1,3 sy, 0,0 ni, 94,1 id, 0,0 wa, 0,0 hi, 0,0 si
%Cpu1 : 3,0 us, 1,0 sy, 0,0 ni, 96,0 id, 0,0 wa, 0,0 hi, 0,0 si
%Cpu2 : 7,4 us, 0,3 sy, 0,0 ni, 92,3 id, 0,0 wa, 0,0 hi, 0,0 si
%Cpu3 : 0,3 us, 0,0 sy, 0,0 ni, 99,7 id, 0,0 wa, 0,0 hi, 0,0 si
KiB Mem : 8167936 total, 6068048 free, 957944 used, 1141944 buff/
KiB Swap: 901484 total, 901484 free, 0 used. 6882948 avail

          PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+
1047 vajk      20   0 4235356 363264 115164 R  15,5  4,4  0:59.78
  892 vajk      20   0 758660 122432  81252 S   1,7  1,5  0:10.00
  1573 vajk     20   0 804724  38708 28448 S   1,0  0,5  0:04.57
  1629 vajk     20   0  52892   4340   3676 R   0,7  0,1  0:05.57

```

2.2. ábra. A programunk fut, de alszik

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

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

Megoldás videó:

Megoldás forrása: tegyük fel, hogy akkora haxorok vagyunk, hogy meg tudjuk írni a Lefagy függvényt, amely tetszőleges programról el tudja döntení, 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.

Tanulságok, tapasztalatok, magyarázat...

Ha a program olyan programot kap bemenetként melyben van végtelen ciklus akkor akkor visszaad egy igaz értéket a T1000es programnak, ami ekkor boldogan leáll és kiírja, hogy igen itt bizony végtelen ciklus

van. Ha viszont a vizsgált programban nincs végtelen ciklus akkor Ő maga fog végtelen ciklusba kerülni. Ebből már rögtön érzékelhetjük, hogy mi is lesz a gond ha saját magát kell ilyen módon elemeznie.

True: megáll, azaz végtelen ciklust talált, de ezek szerint nem áll le, de hogya meg nem áll le, akkor pedig újabban ismét végtelen ciklust talál, tehát leáll - de ha leáll akkor mégsem végtelen? És így tovább... Elsőre az embernek kicsit nehéz lehet ezt értelmezni, de a paradoxonokkal gyakran van ez így. A lényeg, hogy ilyen programot lehetetlen irni.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: <https://github.com/vajkone/prog1/tree/master/feladatok/batch1/swap>

Tanulságok, tapasztalatok, magyarázat...

A legegyszerűbb megoldás erre a problémára egy segédváltozó használata. Ezt a módszert szemléltetni úgy lehet a legkönyebb ha a változókra poharakként, a bennük tárolt értékekre pedig int viz gondolunk. Hogyan tudjuk hát 2 teli pohár tartalmát egyikből a másikba önteni? Természetesen egy harmadik, üres pohár segítségével:

```
int a=10;
int b=7;
int c;

c=a; //c=10
a=b; //a=7
b=c; //b=10
```

A dolgot természetesen matematikai szempontból is meg lehet közelíteni. Ha összeadjuk a két változót majd a kapott összegből kivonjuk a régi értéküket akkor megkapjuk hogy mi volt a másik változó értéke amit hozzáadtunk, azaz amivel cserélni akarjuk. Ez a gyakorlatban így néz ki:

```
int a = 9;
int b = 7;
a = a+b; //a=16
b = a-b; //b=9
a = a-b; //a=7
```

Létezik ugyancsak egy másik megoldás amihez szintén kell matematikai logika viszont már közelebb áll az informatika szakterületéhez, ez pedig nem más mint a XOR azaz a kizáró vagy logikai művelet és az ezzel való csere. A számítógép az általunk deklarált változók (meg úgy minden szám) értékét 2-es számrendszerű számként értelmezi és tárolja. Igy például az előző példában használt 7-es és 9-es így

néznének ki: 7: 0111, 9: 1001. Most hogy ezt tisztáztuk érhetőbb lesz, ha azt mondjuk hogy a XOR művelet 1-est ad vissza mindenhol az vagy az egyik vagy a másik szám bináris értékében 1-es szerepel, de nem mindkettőben. Ez a gyakorlatban így néz ki:

```
int a = 7; //binárisan: 0111
int b = 9; //binárisan: 1001
a = a^b; //a= 0111 ^ 1001 = 1110
b = a^b; //b= 1110 ^ 1001 = 0111
a = a^b; //a= 1110 ^ 0111 = 1001
```

2.4. Labdapattogás

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

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

Megoldás forrása: <https://github.com/vajkone/prog1/tree/master/feladatok/batch1/labdapattog>

Tanulságok, tapasztalatok, magyarázat...

A labdapattogás nem több mint a terminálon belüli, valamilyen karakter (ha már labda akkor legyen O), X Y tengelyen való eltolását jelenti. Ha a labda falhoz ér, azaz x vagy y értékünk 0 lesz, akkor pedig visszapattan. De honnan tudja a labda, hogy a terminálunk fala hol is van? Ebben segít nekünk ez a kód részlet:

```
WINDOW *ablak;
ablak = initscr();
```

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

int deltax = 1;
int deltay = 1;

int mx;
int my;
```

X és y a labdánk kiinduló pozíciója lesz, deltax és deltay pedig a lépésszámot fogja meghatározni az adott tengelyeken. Mx és my változók pedig az ablakunk szélességét és hosszát fogják reprezentálni (oszlopok és sorok száma az ablakunkban).

```
for ( ;; ) {
    getmaxyx ( ablak, my , mx );
    mvprintw ( y, x, "O" );
    refresh ();
    usleep ( 100000 );
    clear()
```

Kezdünk egy, a már korábbi programokból is ismert for ciklussal ami jól látszik hogy a végtelenségik fog futni. A getmaxyx(ablak, my , mx) függvénynek átadjuk az ablakunk hosszát és szélességét, a mvprintw (y, x, "O") függvény pedig y és x koordinátekra fogja rajzolni az 'O' karaktert, azaz a labdánkat. Az usleep() függvény a sleep()-el ellentétben nem másodpercekre, hanem mikroszekundumnyi időre altatja a programunkat, ezzel befolyásolva a labda mozgásának sebességét. A clear() függvény opcionális, ha kikommenteljük akkor a labdánk csíkot húz, azaz az előző kirazjolt labdánk nem fog eltűnni.

```
x = x + deltax;
y = y + deltay;

if ( x>=mx-1 ) {
    deltax = deltax * -1;
}
if ( x<=0 ) {
    deltax = deltax * -1;
}
if ( y<=0 ) {
    deltay = deltay * -1;
}
if ( y>=my-1 ) {
    deltay = deltay * -1;
}

}

return 0;
}
```

Itt történik meg a pozícióváltás, deltax és deltay-nal növeljük, változtatjuk a labda következő kirajzolandó helyzetét. Ezután pedig megvizsgáljuk, hogy a labda így elérte-e az ablakunk széleit (mx és my), ha igen,

akkor deltax és/vagy deltay előjelét megváltoztatjuk, hogy ezután, hogyha hozzádjuk őket a jelenlegi x és y pozícióhoz, akkor ellentétes irányba menjenek mint eddig, azaz a labdánk pattanjon vissza a falról.

Az if nélküli megoldáshoz ismét a matematika csodálatos tudományát hivjuk segítségül, mégpedig nem más mint a maradékos osztást. Ebben a megoldásunkban a for ciklusunkon belül eltávolítja a logikai vizsgálatokat és helyette matekozunk. A kód valahogy így fog kinézni:

```
for ( ; ; ) {  
  
    getmaxyx(ablak, my, mx);  
    x = (x - 1) % mx;  
    deltax = (deltax + 1) % mx;  
  
    y = (y - 1) % my;  
    deltay = (deltay + 1) % my;  
  
    clear ();  
  
    mvprintw (abs (y + (my - deltay)),  
              abs (x + (mx - deltax)), "O");  
  
    refresh ();  
  
    usleep (200000);  
  
}
```

Itt folyamatosan maradékos osztást végzünk az ablak szélességével és hosszával, melyek ügye a maximum értékek, tehát egészen addig amíg el nem érjük az ablak határait (tehát azokat az értékeket) addig ugyanazt kapjuk amit elosztottunk, de közben okosan léptetjük is ha már ott vagyunk. Ha elérjük az ablak szélét akkor az osztásunk 1-et ad vissza (mert ügye már inkrementáljuk is egyből) és ilyenkor a labdánk visszapattan.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/tree/master/feladatok/batch1/szohossz>

Tanulságok, tapasztalatok, magyarázat...

Először is: mi is az a szóhossz az informatikai zsargonban? Ez nem más mint a számítógépünk által, adatok ábrázolása és átvitele közben alkalmazott elemi egység. A BogoMips pedig egy Linus Torvalds által

irt program mely a CPU-nk gyorsaságát hivatott mérni. Persze ez nem ér fel egy CPU stretch test-tel, de diákok számára kiváló versenyt biztosíthat, hogy kinek fut le gyorsabban. A BogoMips-ből átemeljük a while ciklust, melyben a másodpercenkénti loop-ot egy másik változóval helyettesítjuk. Ez a változó fogja mérni a bitek számát. A tényleges szóhosszt bitshifteléssel fogjuk kideríteni. Mindez a programunkban így fog kinézni:

```
int h = 0;
int n = 0x01;
do
    ++h;
while (n <= 1);
```

A while ciklusból kilépve, tehát amikor az n csupa nullából áll, pedig kiirjuk a h-t, azaz hogy hányszor kellett az egyest balra léptetni a bináris formában, ami ha minden jól ment 32 lesz.

Ezek után vessünk egy pillantást a Linus Torvalds féle BogoMIPS-re.

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

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

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

    while (loops_per_sec <= 1 )
    {
        ticks = clock();
        delay (loops_per_sec);
        ticks = clock() - ticks;

        printf ("%llu %llu\n", ticks, loops_per_sec);

        if (ticks >= CLOCKS_PER_SEC)
        {
```

```
loops_per_sec = (loops_per_sec / ticks) * ←
    CLOCKS_PER_SEC;

printf ("ok - %llu.%02llu BogoMIPS\n", ←
    loops_per_sec/500000,
    (loops_per_sec/5000) % 100);
printf("ok - %lld %f BogoMIPS\n", loops_per_sec, ←
    log(loops_per_sec));
return 0;
}

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

2.6. Helló, Google!

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

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/tree/master/feladatok/batch1/pagerank>

Tanulságok, tapasztalatok, magyarázat...

A Google keresője mára vitathatatlanul a legnépszerűbb kereső motorrá nőtte ki magát. De minek köszönheti népszerűségét? Néha az az érzésünk támadhat, hogy a Google kereső jobban tudja, hogy mit is akarunk mint mi magunk. (Néha mikor még a fejemben se állt össze, hogy mire is szeretnénk rákeresni, a google már ki is hozta mint első javaslat. Hát már ilyen szintent megfigyelnek minket!?) Hogyan lett hát a google keresője ilyen kifinomult? Nos, a kereső algoritmus egyik, ha nem a legfontosabb része / funkciója a PageRank rendszer. Ez az az algoritmus mely eldönteni, hogy egyes lapok mennyire elől szerepeljenek a találatok közötti listában. Személy szerint nem emlékszem, hogy a közelmúltban bármikor is a találatok 2. oldalára kellett volna kattintanom, hogy megtaláljam amit keressek, szóval úgy vélem nagy biztonsággal kijelenthetjük, hogy a google PageRank algoritmusa jól végzi a dolgát. De hogyan is teszi mindezt?

A PageRank bizonyos pontrendszerrel dolgozik. Ezeket a pontokat részben azért kaphatja egy oldal, hogy ha más oldaláról rá mutatnak (linkek). Logikus, hiszen jobb minőségű oldalakra nyilván több link fog mutatni, mint egy gagyi oldalra. Ezek a linkek ún. szavazatoknak számítanak és nem egyenlő erősséggűek. Magasabb pontszámú oldalak szavazatai értéksebbek, mint gyengébb pontszámú társaiké. Egy oldal "szavazó-ereje" annál gyengébb minél több külső helyre mutat. Ez részben azt eredményezi, hogy az oldalak tényleg csak olyan oldalra fognak mutatni, akit arra érdemesnek tartanak.

2.7. 100 éves a Brun téTEL

Írj R szimulációt a Brun téTEL demonstrálására!

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

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

A Brun téTEL és a Brun szám Viggo Brun, norvég fizikus nevéhez kötődik. A téTEL azt mondja ki hogy az ikerprímek reciprokösszege egy bizonyos összeghez konvergál, azaz azt tetszőlegesen megközelíti, de el nem éri. Ezt a számot nevezik Brun konstansnak. Egyelőre csak becslések léteznek arról, hogy ez a szám mekkora is, de a szakemberek valahova 1,9 és 2,1754 közé tippelik.

Az R-ben megírt program így fog kinézni:

```
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))  
}
```

Az x paraméterként átadott számig a program eltárolja az összes primszámot a primes vektorban, ezután megkeressük ikerprímeket, majd vesszük ezen primek reciprokát és ezeket összeadjuk.

2.8. A Monty Hall probléMA

Írj R szimulációt a Monty Hall problémára!

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

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

Tanulságok, tapasztalatok, magyarázat...

Kezdjük azzal, hogy mi is az a Monty Hall probléMA?

Tegyük fel, hogy játékosok vagyunk egy televíziós műsorban, ahol egy autót nyerhetünk, ha jól választunk. HárOM ajtó közül kell választanunk, az egyik mögött ott rejlik a nyeremény, a másik kettő mögött semmi. Választunk egy ajtót, a műsorvezető kinyit egy másik ajtót amiről ő tudja hogy üres, majd felajánja, hogy ha szeretnénk, akkor változtathatunk eredeti választásunkon. Mit érdemes ilyenkor tenni? Nos tömören ez volna a Monty Hall probléMA.

A válasz nem feltétlen magától értetődő, de fáradhatatlan matematikusok izzadtságos munkájának és szimulációk milliárdjainak hála már tudjuk, hogy igen, megéri változtatni a kezdeti tippünkön ebben az

esetben. A magyarázat pedig a következő: kezdetben, ügye egyértelmű hogy 1/3-ad esélyünk van eltalálni a nyereményt rejtvő ajtót. A játékvezető ezután kinyit egy ajtót mely mögött semmi nem rejlik. Ha eltaláltuk elsőre (1/3 esély) és váltunk, akkor nem nyerünk, viszont ha maradunk eredeti döntésünknél akkor miénk a főnyeremény. Viszont ha elsőre nem jól tippeltünk (aminek 2/3 az esélye) és váltunk miután a másik üres ajtót kinyították, akkor nyerünk. Magyarul azért éri meg változtatni, mert az elején 2/3-ad (66,6%) az esélye, hogy rosszat választunk.

DRAFT

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/chomsky/turinggep.c>

Tanulságok, tapasztalatok, magyarázat...

Nagyon részletesen az unáris számrendszeret és a Turing gépet tárgyalni nem igazán lehet, ezért röviden: Az unáris (egyes) számrendszerben az N decimális számot N darab 1-essel vagy szimpla egyenes vonással (|) lehet ábrázolni. A Turing gép ezt úgy teszi meg, hogy az átváltandó számot beolvassa és addig vonogat ki belőle 1-et, amíg az eredeti szám értéke 0 nem lesz. A kivont egyeseket pedig sorban a kimenetre (vagy tárolóba) irja.

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

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

Megoldás videó:

Tutorom a megoldásban: Petrus Tamás József

Szabályok:

S → aBSc

S → abc

Ba → aB

Bb → bb

Levezetés:

S → aBSc → aBaBScc → aBaBabccc → aaBBbabccc → aaBaBbcc → ↪
aaaBBbccc → aaaBbbccc → aaabbccc

Szabályok:

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

Levezetés:

$S \rightarrow aXbc \rightarrow abXc \rightarrow abYbcc \rightarrow aYbbcc \rightarrow aaXbbcc \rightarrow aabXbcc \rightarrow \leftarrow aabbXcc \rightarrow aabbYbcc \rightarrow aabYbbccc \rightarrow aaYbbbccc \rightarrow aaabbccc$

A generálás során a meghatározott szabályok alapján a vizsgálandó szót lecseréljük a megadott részszóra, majd a kapott részszóból a szabályok alapján újabbat generálunk, ezt addig folytatjuk míg a nyil jobb oldalán csupa terminális változók nem állnak.

3.3. Hivatkozási nyelv

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

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/chomsky/statements.txt>

A Backus-Naur-Form egyfajta metaszintaxis melynek segítségével különböző nyelveket, azok nyalvtanát írhatjuk le. Nevét John Backusról és Peter Naurról kapta. A módszert magát Backus hozta létre és Peter Naur egyszerűsítette le. Eredetileg az ALGOL nyelv szintaxisának leírására szolgált, de azóta a legtöbb programozási nyelv szintaxisának leírására ezt a módszert használják.

```
for (int i=0; i<10; i++)
```

Az fenti kódcsipet például gcc -std=c89 fordítással nem fordul le, mert a for cikluson belül deklaráltuk az i változót. A konzolra kapunk is egy hiba üzenetet, hogy ezt csak C99 vagy C11 módban tehetjük meg.

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/chomsky/szamokszama.l>

Tanulságok, tapasztalatok, magyarázat...

A lexer program segítségével könnyedén tudunk olyan input elemző programot készíteni, amely megszámolja az inputban található számokat. A lexer egy szövegfjlból beolvassa a lexikális szabályokat, majd kimenetként C nyelvű kódot állít elő.

```
%{  
    #include <string.h>  
    int szamok_szama = 0;  
}  
%%
```

A kódon belül az egyes részeket `%%` (dupla százalék jel) választja el egymástól. Ebben a kódrészletben C nyelven inicializálunk egy változót ami majd a számokat számolja és tárolja és include-oljuk a string.h header file-t.

```
[0-9]+    ++szamok_szama;  
[a-zA-Z] [a-zA-Z0-9]* ;  
%%
```

Ezt követi a lexer rész, ahol a lexikális szabályokat adjuk meg, jelen esetben, hogy növelje a szamok_szama-t ha a bemenet szám, és hagyja figyelmen kívül ha a bemenet bármilyen betű, de még akkor is ha a betű vagy betűk sorozata után áll egy szám, azaz csak akkor számol ha a bemenet számmal kezdődik.

```
int main()  
{  
    yylex();  
    printf("%d szam\n", szamok_szama);  
    return 0;  
}
```

Végül, ismét C nyelven, meghívjuk a main függvényt, amin belül pedig a yylex() függvényt is, ami a lexikális elemzőnk, amely értelmezi az előző blokkban definiált szabályokat.

3.5. I33t.I

Lexelj össze egy l33t cipher!

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/chomsky/leet.l>

Tanulságok, tapasztalatok, magyarázat...

Ahogy azt az előző lexeres programnál is láttuk a kód itt is 3 részre oszlik. Előre kerülnek a header file-ok és egyéb definíciók, 2. blokkban jönnek a szabályok. Végül pedig a harmadik blokkban parse-oljuk yylex()-el a szabályokat.

```
struct cipher {
    char c;
    char *pool[3];
} l337d1c7 [] = {
```

Készítünk egy cipher struktúrát mely tartalmaz egy karaktert, ami majd az egyes betűket fogja reprezentálni és egy 3 méretű tömböt azoknak a karaktereknek amikre majd lecseréljük az betűket. Ezután végigmegyünk az abc betűin és megadunk minden betűhoz három leet alternatívát, ezzel létrehozva egy könyvtárat.

```
int found = 0;
for(int i=0; i<L337SIZE; ++i)
{
    if(l337d1c7[i].c == tolower(*yytext))
    {
        int r = 1+(int)(100.0*rand()/(RAND_MAX+1.0));
        if(r<34)
            printf("%s", l337d1c7[i].pool[0]);
        else if(r<67)
            printf("%s", l337d1c7[i].pool[1]);
        else
            printf("%s", l337d1c7[i].pool[2]);

        found = 1;
        break;
    }
}
if(!found)
```

```
printf("%c", *yytext);
```

A második blokkban jöhetnek a szabályok, kezdve egy for ciklussal ami végigmegy a leetcode-on, amit korábban definiáltunk (a leet könyvtárunk mérete osztva egy darab cipher tétel méretével). Majd azután egy random szám generátor segítségével döntjük el, hogy a három megadott leet alternatívából melyiket is válassza, mindegyikre nagyjából 33% esély van. Ha pedig olyan karakterrel találkozik amit nem adtunk meg a könyvtárunkban akkor szimplán visszaadja az adott karaktert.

3.6. A források olvasása

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

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

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



Bugok

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

- i.

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

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

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

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

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

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

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

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

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

Amennyiben a SIGINT nem kerül ignorálásra (SIG_IGN) akkor a jelkezelő kezelje a signalt

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

Két for ciklus amelyek 5-ször futnak le, különbség közöttük csak annyi, hogy az elsőben előbb növeljük az i-t és csak azután használnánk azt (magát az értéket), de mivel itt ilyenre nem kerül sor, ezért igazából jelentéktelen az eltérés.

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

Feltételezzük hogy az i változó már korábban deklarálva lett, ekkor ez a ciklus 5-ször fug lefutni és a tomb nevű tömb első ötelelemét fogja 0 1 2 3 4-re állítani. Gond csak akkor van ha a tömb nagysága kisebb mint 5.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

Minden x-hez létezik olyan y, hogy x kisebb, mint y, és y prím. Azaz: " minden számnál létezik nagyobb primszám. Magyarul: a primszámok száma végtelen.

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

Minden x-hez létezik olyan y, hogy x kisebb, mint y, y prim és y rákövetkezőjének rákövetkezője is prím. Azaz: minden számnál léteznek nagyobb ikerprímek, magyarul: az ikerprímek száma végtelen

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

Létezik olyan y, hogy minden x-re igaz, hogy ha x prim, akkor x kisebb, mint y. Számomra ez kétértelmű, egyszer jelentheti azt hogy létezik, egy bizonyos y amitől nincs nagyobb x prim, azaz a primek száma végtelen, vagy pedig hogy minden x prim-hez létezik egy y ami nagyobb tőle, ami csak annyit jelent, hogy a természetes és primszámok száma is végtelen.

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

Létezik olyan y hogy minden x-re igaz, hogy ha y kisebb mint x akkor nem primszám. Őszintén: ezt még el kell olvasnom párszor hogy megértem

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

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

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

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

Mit vezetnek be a programba a következő nevek?

- `int a; //egész tipusú változó`
- `int *b = &a; //egész változóra mutató mutató`
- `int &r = a; //egész változó referenciajára`
- `int c[5]; //5 méretű tömb egészek számára`
- `int (&tr)[5] = c; //egészek tömbjének referenciajára`
- `int *d[5]; //egészre mutató mutatók tömbje`
- `int *h (); //függvény ami egészekre mutatót ad vissza`
- `int *(*l) (); //mutató amely egészre mutató függvényre mutató mutatóra → mutat`
- `int (*v (int c)) (int a, int b); //egészet visszaadó és két egészet kapó ← függvényre mutatót visszaadó, egészet kapó függvény`

- ```
int (*(*z) (int)) (int, int); //függvénymutató egy egészet visszaadó és ↫
 két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó ↫
 függvényre
```

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/chomsky/bevezet.c>

DRAFT

## 4. fejezet

# Helló, Caesar!

### 4.1. double \*\* háromszögmátrix

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/caesar/matrix.c>

Tanulságok, tapasztalatok, magyarázat...

A háromszögmátrixok olyan négyzetes mátrixok (mátrixok melyeknek sorainak és oszlopainak száma meg-egyeznek), melyeknek a főátló alatti vagy feletti értékei csupa nullák. Egy ilyen mátrixban a determináns kiszámítása is pofon egyszerű, hiszen csak a főátlóban lévő számokat kell összeszorozni.

Mi most egy alsó háromszögmátrixot fogunk létrehozni, ami azt jelenti, hogy a főátló felett lesznek a nullák.

```
int nr = 5;
double **tm;

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

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;
 }
}
```

Az nr-ben tároljuk, hogy mekkora mátrixot szeretnénk, ez a mostani egy 5x5-ös lesz. Ezután ellenőrizzük, hogy a malloc által lefoglalt memória területünk NULL-ra mutató pointert ad-e vissza, (pl nincs memória) ha igen akkor hibába ütközünk és kilép a programból.

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

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

Itt értek adunk a mátrix elemeinek, 1-től kezdve iterálva, majd ki is irjuk a mátrixot.

```
tm[3][0] = 42.0;
(*tm + 3)[1] = 43.0;
*(tm[3] + 2) = 44.0;
*(*tm + 3) + 3) = 45.0;

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

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

free (tm);
```

Itt különböző módokon adunk értéket a 4. sor elemeinek. A tm[s][o] mátrixban az s a sor számát az o pedig az oszlopét jelöli. A program végén, mikor már nincs rá szükségünk, a free() függvényel felszabadítjuk a mátrix számára lefoglalt memóriát.

## 4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/caesar/exortitkosit.c>

Tanulságok, tapasztalatok, magyarázat...

A titkositáshoz szükségünk lesz egy titkositando szövegre és egy kulcsra, valamint az elvre ami alapján a szöveget titkositjuk. Jelen esetben ez az elv a XOR logikai művelet által biztosított lehetőségeken fog alapulni. Korábban már használtuk a kizáró vagyot 2 érték felcserélésekor, most megnézzük, hogyan tudjuk titkositáshoz használni.

```
#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;
```

Konstansként definiáljuk a max\_kulcsot és a buffer\_meret-et. Ezek értéke a későbbiekben már nem változható. A main függvényünk most a szokásostól eltérően kap 2 argumentumot, ezekre azért van szükség mert a kódolandó szöveget és kulcsot parancssorból fogja megkapni. Ezek után létrehozzuk a kulcs tömbünket (melynek mérete definíció szerint 100) és a buffer tömbünket melybe a beolvasott karaktereket fogjuk tárolni (egyszerre ügye maximum 256öt). Inicializálunk két változót, az egyik a kulcs tömb bejárására a másik pedig a beolvasott bajtok számlálására szolgál.

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

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

 buffer[i] = buffer[i] ^ kulcs[kulcs_index];
 kulcs_index = (kulcs_index + 1) % kulcs_meret;

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

A strlen() függvény visszaadja az argv[1] hosszát, azaz a második argumentumét ([1] = második) tehát az általunk megadott kulcsét, amelyet el is mentünk a kulcs\_meret változóba. Ezután a strncpy() függvénytellyel a kulcs tömbükbe másoljuk a parancssorban megadott kulcsunkat karakterenként (a kulcs ügye egy char tömb).

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

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

A while ciklusunk addig fog tartani, amíg a read() függvénynek van mit beolvasnia. Jelen esetben a buffer tömbbe fog beolvasni standard inputról (ez a 0) BUFFER\_MERET-nyi, azaz 256 bájtot. Ezután egy belső ciklusban végigmegyünk a jelenlegi beolvasott bájtokon vagy ha úgy tetszik akkor a karakterek számán és a bufferben tárolt beolvasott szövegrészről összexorozzuk a kulcs adott (a kulcs index adja meg hogy melyik) karakterével. Ezután növeljük a kulcs indexét eggyel. A %kulcs\_meret arra szolgál, hogy ha elérjük a kulcs méretét, azaz végét akkor a következő iterációban előről, 0-ról induljon a kulcs index. Végül pedig kiirjuk a buffer tartalmát (ami már ügye a titkos szöveg) a standard outputra.

### 4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/caesar/exortitkosito.java>

Tanulságok, tapasztalatok, magyarázat...

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
```

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

public class Main {

 public static void main(String[] args) throws IOException {
 FileInputStream fin;
 FileOutputStream fout;

 fin=new FileInputStream("tiszta.txt");
 int c;
 ArrayList<Integer> input = new ArrayList<Integer>();
 ArrayList<Integer> output = new ArrayList<Integer>();
 Scanner in = new Scanner(System.in);
 System.out.println("Adja meg a kulcsot: ");
 String kulcs = in.nextLine();
 int kulcs_hossz = kulcs.length();

 while((c=fin.read())!=-1)
 {
 input.add(c);
 }
 fout = new FileOutputStream("titkos.txt");
 for(int i=0;i<input.size();i++)
 {
 output.add(input.get(i)^kulcs.charAt(i%kulcs_hossz));
 fout.write(output.get(i));
 }

 for(int i=0;i<output.size();i++)
 {
 System.out.printf("%c",output.get(i));
 }

 fin.close();
 fout.close();
 }
}
```

Mivel a Java is C alapú nyelv ezért a program logikai felépítése szinte megegyezik az előzőjével. Érdemi változtatás ott történik részéről, hogy a titkositandó szöveget és kulcsot a main függvényünk most nem parancssori argumentumként kapja meg (ennél a feladatnál windowsban dolgoztam és így egyszerűbbnek éreztem), hanem a kódba ágyazva adjuk át neki a tiszta szöveget (vagyis az elérési útját) és a user a programot futtatva kap egy promptot ahol megadhatja a kulcsot. Ezután ugyancsak a kódba ágyazva megadjuk

az output file nevét és elérési útját (ha csak a nevét adjuk az is elég, ekkor a programunk src mappájába kerül automatikusan) és egy for loopban bájtonként beleirjuk a titkositott szöveget. Ezek után még fontos, hogy lezárjuk az input és output stameeket (amikkel file-okból olvastunk és file-okba írunk) különben előfordulhat, hogy például az output txt-nk üresen marad a program futása után.

## 4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/caesar/exortoro.c>

Tanulságok, tapasztalatok, magyarázat...

Próbáljuk hát meg feltörni, a korábban elkészített titkos szövegünket. Feltörés alatt a kulcs-ként megadott szó megfejtését kell érteni.

```
#define MAX_TITKOS 4096
#define OLVASAS_BUFFER 256
#define KULCS_MERET 8

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

Konstansok definiálása és könytárak hozzáadása a programunkhoz. Felmerülhet a kérdés hogy a KULCS\_MERET et miért végelegesítjük 8-asnak? Nos igen, ez azt fogja feltételezni, hogy a titkositásnál 8 karakternyi hosszú stringet adtunk meg kulcsként. Szóval a titkositónak muszáj kicsit segítenie minket azzal, hogy a "mi szabályaink" szerint játszik, ha azt szeretné, hogy a kódját feltörjük.

```
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;
}
```

Ahogy neve is mutatja ez a függvény a titkositott bemenet szavainak átlagos hosszát fogja számolni. Ezt úgy teszi, hogy végigmegy a bemeneten és ha szóközhösz ér akkor, a szavak számlálására használt változót növeli eggyel. Végül az argumentumként átadott titkositott szöveg méretét elosztja a szavak számával.

```
int tiszta_lehet (const char *titkos, int titkos_meret)
{
 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");

}
```

Ez a függvény azt vizsgálja, hogy a dekódolt szöveg megfelel-e az alábbi követelményeknek: Az átlagos szóhossz 6 és 9 között van, valamint tartalmazza-e a "hogy" "nem" "az" "ha" szavakat.

```
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;
 }
}
```

Ez a függvény fogja megpróbálni a kód visszafejtését, melyekhez a program által generált lehetséges kulcsokat fogja használni. Ha megfigyeljük, láthatjuk, hogy ez ugyanazt csinálja mint a titkositásnál használt pár sor, ez azért van mert ha valamit kétszer exorozunk akkor az eredeti szöveget kapjuk vissza, tehát egyszer exor=titkositott, kétszer exor=tiszta szöveg.

```
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);
}
```

Itt hivjuk meg a korábban bemutatott másik 2 függvényt. A return-ünk akkor lesz igaz ha a tiszta\_lehet függvénynek sikerült a visszafejtés.

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

for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
titkos[p - titkos + i] = '\0';

for (int ii = '0'; ii <= '9'; ++ii)
 for (int ji = '0'; ji <= '9'; ++ji)
 for (int ki = '0'; ki <= '9'; ++ki)
 for (int li = '0'; li <= '9'; ++li)
 for (int mi = '0'; mi <= '9'; ++mi)
 for (int ni = '0'; ni <= '9'; ++ni)
 for (int oi = '0'; oi <= '9'; ++oi)
 for (int pi = '0'; pi <= '9'; ++pi)
 {
 kulcs[0] = ii;
 kulcs[1] = ji;
 kulcs[2] = ki;
 kulcs[3] = li;
 kulcs[4] = mi;
 kulcs[5] = ni;
 kulcs[6] = oi;
 kulcs[7] = pi;

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

 exor (kulcs, KULCS_MERET, titkos, p - titkos);
```

}

A main függvényünkön belül elkezdjük a bufferbe olvasni a dekódolandó szöveget, ha a szöveg végére érünk és a bufferben üres hely marad azokat töröljük. Ezután 8 egymásba ágyazott for ciklussal elkezdjük előállítani a kulcsot. A legelső for ciklus végén meghívjuk az exor\_tores-t és megvizsgáljuk, hogy a a kulcs amit találtunk jó-e, tehát megvizsgáljuk elsőnek a 00000000-t, majd 00000001-t és így tovább. Amint azt láthatjuk, a for ciklusok 0-tól 9-ig mennek, magyarul ez a program csak olyan kulcsot tud visszafejteni ami 8 darab számból áll.

## 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](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/NN_R)

Tutorom a megoldásban: Lovász Botond

Tanulságok, tapasztalatok, magyarázat...

A neuronok agyunk olyan idegsejtjei melyek elekromos potenciállal rendelkeznek és más sejtekkel speciális kapcsolatok, úgynévezet szinapszisok révén kommunikálnak. Mesterséges neuronnak azt a matematikai modelt nevezzük, amely a biológiai neuron képletét hivatott megadni (McCulloch és Pitts, 1943). A mesterséges neuronok alapvető elemei egy (mesterséges) neurális hálónak.

A model megértéséhez először is fontos tudni, hogy egy neuron akkor "tüzel" ha a bemeneti értékek súlyozott összege meghalad egy bizonyos küszöböt.

Először is a programunkat meg kell tanítani az OR AND és EXOR logikai műveletek használatára, ez a programunkban így fog kinézni:

```
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)

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

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

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])
```

A compute függvénynek átadjuk a betantott neurális hálót és az elején megadott adatokat, ami ezután kiszámolja a logikai műveletek eredményét. Ha ezt így futtatjuk azt tapasztaljuk, hogy a program nagyjából 50%-os pontossággal fog dolgozni, ami nem jobb mint a véletlenszerű találhatóság. Ha viszont ún. rejtekt neuronokat adunk hozzá, azaz növeljük a számukat (mert ügyebár a fenti programban is látható hogy van csak 0-ra van állítva), a program tanulékonysága drasztikusan megnő, ezzel növelve a pontosságot is. Növeljük hát meg mind a rejtekt neuronok rétegét minden számukat:

```
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])
```

Ezekkel az értékekkel futtatva a plot függvény ugyan sokkal bonyolultabb ábrát fog kirajzolni, de a rejtekt neuronaknak hála mostmár az EXOR műveletet is nagy pontossággal végzi el a program.

## 4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:<https://www.youtube.com/watch?v=XpBnR31BRJY>

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

Tanulságok, tapasztalatok, magyarázat...

A gépi tanulás területén, a perceptron egy olyan algoritmus melyet a bináris osztályozás területén használnak. A perceptron éppen úgy működik mint a neurális hálózatok, ahhoz hogy működjön, előbb be kell tanitani, meg kell neki mondani, hogy bizonyos dolgokat miről ismer fel és hogy a jövőben ezeket mi alapján osztályozza. A perceptronnak 4 fő része van, ezek:

- 1: input értékek
- 2: súlyozás
- 3: össz súly
- 4: osztályozás funkció

Először is az inputokat meg kell szorozni a saját súlyukkal, ezután ezeket az értékeket összeadjuk amiből megkapjuk a súlyozott összeget, majd ezt az összeget betápláljuk a megfelelő funkcióba, ami ezután az inputokat osztályozza. Mint korábban említésre került, bináris osztályozásnál a funkciónak csak 2 osztály közül kell választania. Pl.: 0 és 1, sötét-világos, kutya-macska stb...

## 4.7. Gutenberg olvasónaplók

## 4.8. Juhász István - MAGAS SZINTŰ PROGRAMOZÁSI NYELVEK 1

A számítógépek megjelenése lehetővé tette az emberi gondolkodás bizonyos elemeinek automatizálását. Ehhez viszont az embernek szüksége volt arra hogy a számítógépet programozni tudja. Ennek a kivitel-zésre kailakult nyelveknek 3 szintjét különböztetjük meg. Az első a gépi nyelv ami valójában 0-k és 1-esek sorozata, azaz bináris kód. E fölött áll az assembly nyelv, ami félúton van gépi nyelv és emberi nyelv között. Ami pedig már "emberi fogyasztásra alkalmas" minősül, az pedig a magas szintű programozási nyelv azaz nyelvek merthogy mára már rengeteg van belőlük, viszont ez a könyv a korábbiak közül fog néhányra koncentrálni, azokon belül is leginkább az ADA és C nyelveken lesz a fókusz. Az ezeken a nyelveken íródott programokat forrásprogramnak vagy forrásszövegnek nevezzük (angolul source-code), melyekre vonatkoznak bizonyos nyelvtani és formai szabályok - ezeket szintaktikai szabályoknak nevezzük. Egy nyelvre vonatkoznak ugyanakkor szemantikai szabályok is, ezek az értelmezési, tartalmi és jelentésbeli szabályok.

## 4.9. Kernigen and Ritchie: A C programozási nyelv

A könyv azzal a céllal íródott, hogy az olvasó a lehető leggyorsabban eljusson arra a pontra, ahol már önállóan működő és használható programokat tud irni. A könyv irói feltételezik, hogy az olvasó nem teljesen nulláról indul, volt már dolga számítógépekkel esetleg programozással is próbálkozott. Ez persze nem jelenti azt, hogy bármely fontos részt átugranának, csak szimplán a szakzsargont nem részletezik.

A bevezető alapvető fogalmakra és tudnivalókra koncentrál: Változók és állandók, a C nyelv aritmetikája, input és output alapvető kezelése és az ezekhez kapcsolódó ismeretek, függvények, vezérlésátadás. A bevezetőbe nem kerültek bele a mutatók, struktúrák, a nyelv operátorkészletének java és valamennyi vezérlésátadó utasítás sem. Ezek tudatosan kerültek későbbi fejezetekbe, ugyanis ezek majd nagyobb és összetettebb programok irásánál lesznek elengedhetetlenek. Ez a megközelítés természetesen jár néhány hátránnal is, nevezetesen, hogy a nyelv összetevőinek bemutatása valamelyest szétszórtan lesz megtalálható a könyvben és nem tudunk majd szükség esetén egy bizonyos fejezetre visszalapozni, ahol minden egyben van.

## 4.10. Benedek Zoltán - Levendovszky Tihamér: Szoftverfejlesztés C++ nyelven

Ahogy a cím is mutatja C nyelvről váltunk C++-ra. Azért is fontos megjegyezni, hogy váltunk mert a könyv maga is gyakran a C-hez viszonyítva mutatja be a C++ nyelvet és azt, hogy az hol és hogyan változtat és javít rajta. Vitathatatlan, hogy a C++ előrelépés a C-hez képest, ahogy azt neve is viccesen mutatja, C csak inkrementálva. A C++ olyan változtatásokat vezet be mint az osztályok, ezért a C++ már egy objektum oriántált nyelv lesz. Megjelenik a boolean változó (true, false), melyet eddig ügye csak számok reprezentáltak. A könyv tárgyalja továbbá a referenciával történő paraméterátadást, ahol a függvényen belül változtathatunk egy külső változó értékén. De szó esik a "Rule of 3" és "Rule of 5" -ról is mely szabályok az erőforrásgazdálkodásra vonatkoznak. A Rule of 3 például azt mondja ki hogy ha a destructor, copy constructor vagy copy assignment operator valamelyikét tartalmazza egy osztály akkor explicit módon minden hármat érdemes definiálni.

DRAFT

## 5. fejezet

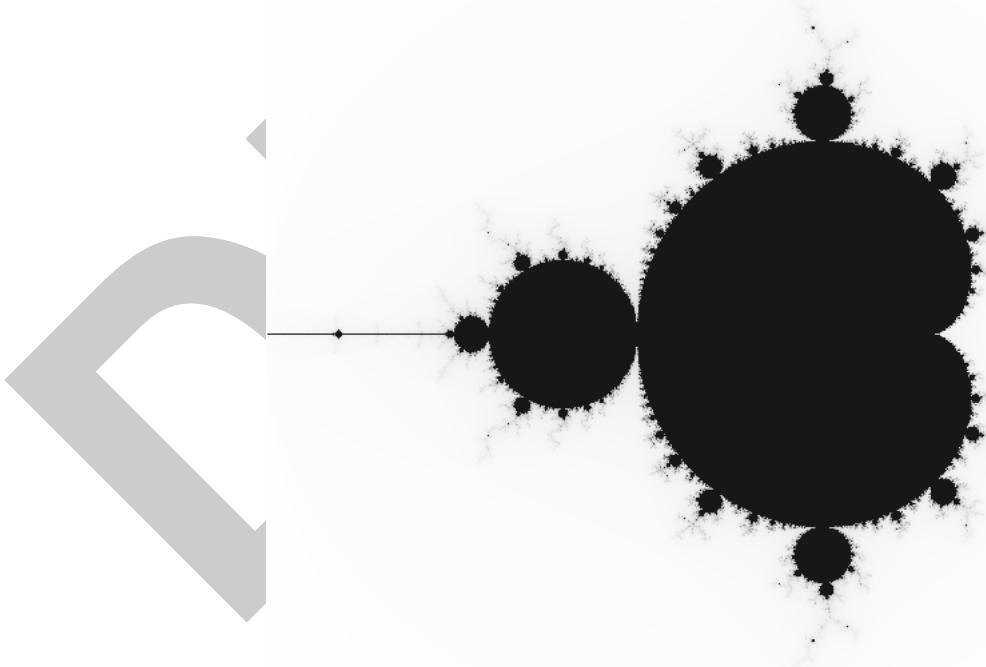
# Helló, Mandelbrot!

### 5.1. A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/mandelbrot/mandelbrot.cpp>

A Mandelbrot halmazt Benoît Mandelbrot fedezte fel 1979-ben, de csak később 1982-ben nevezték el ténylegesen róla. Ezt a halmazt a komplex számsík elemei adják. A halmazt ábrázolva egy fraktált kapunk.



5.1. ábra. Mandelbrot halmaz

A fraktálok olyan végtelenül komplex geometriai alakzatok melyeket két fő tulajdonságuk különböztet meg az átlagos geometriai alakzatoktól. Egyik, hogy a határoló vonalai "recések" avagy "érdesek", a másik pedig, hogy önhasonló tulajdonsággal rendelkeznek. Ez azt jelenti, hogy ha elkezdünk a korábban emlitett "recékre" közelíteni, akkor előbb-utóbb nagy valószínűséggel az visszakapjuk az eredeti alakzatot. Ilyen alakzatok nem csak a matematikában fordulnak elő, hanem a természetben is. Ilyen alakzatokat/mintákat produkál például a brokkoli is ha eléggé nagy nagyításban kezdjük elv izsgálni.



5.2. ábra. Fraktál a természetben

Lássuk hát, hogy hogyan lehet az első ábrán látott képet egy program segítségével generálni. Mivel mi ezt most c++ban fogjuk elkészíteni ezért először is szükségünk lesz a png++ header file-ra amivel ha még nem rendelkezünk, először is le kell töltenünk. Ehhez terminálban adjuk ki a következő parancsot: sudo apt-get install libpng++-dev.

Ha ez megvan nézzük, hogyan is épül fel a program. Először is include-oljuk a letöltött png++ header file-unkat:

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

}
```

Ezután megadjuk a függvény értékkészletét, a majd létrehozandó kép szélességét és magasságát és az iterációs határt amit majd a későbbi while ciklusban fogunk használni. Továbbá elkészítjük a png képet amibe majd pixelenként belerajzoljuk a mandelbrot halmazt.

```
double a = -2.0, b = .7, c = -1.35, d = 1.35;
int szelesseg = 600, magassag = 600, iteraciosHatar = 1000;

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

}
```

```
double dx = (b-a)/szelesseg;
double dy = (d-c)/magassag;
double reC, imC, reZ, imZ, ujreZ, ujimZ;

int iteracio = 0;
std::cout << "Szamitas";

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

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

 reC = a+k*dx;
 imC = d-j*dy;

 reZ = 0;
 imZ = 0;
 iteracio = 0;

 while (reZ*reZ + imZ*imZ < 4 && iteracio < iteraciosHatar) {

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

 ++iteracio;

 }

 }

}
```

Végül pedig az elején létrehozott kep png-be belrajzoljuk a generált mandelbrot halmazt, majd ezt a png-ténylegesen elhelyezzük a felhasználó által, argumentumként megadott png állományba.

```
kep.set_pixel(k, j, png::rgb_pixel(255-iteracio%256,
 255-iteracio%256,
 255-iteracio%256));
}
std::cout << "." << std::flush;
}

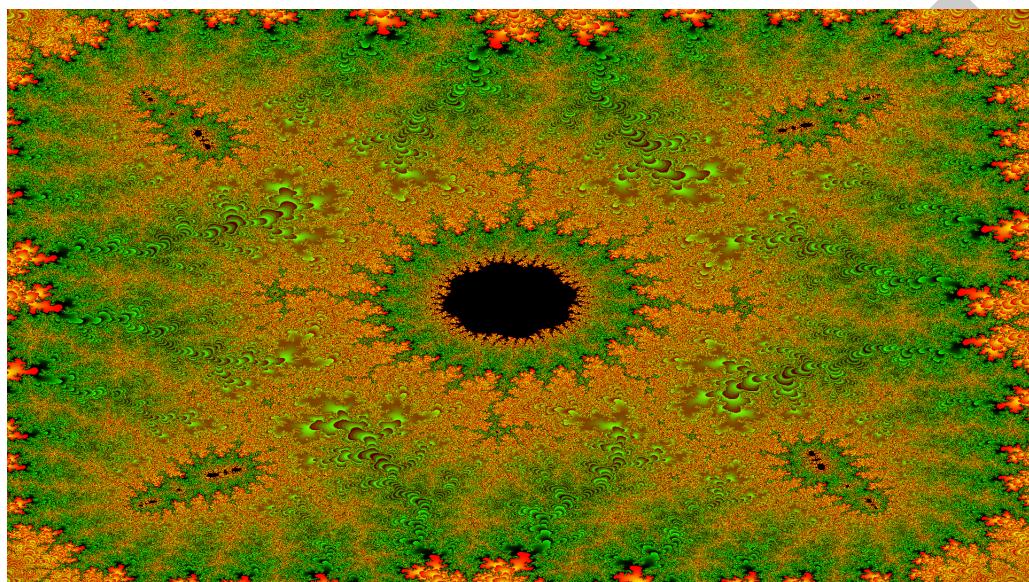
kep.write(argv[1]);
}
```

## 5.2. A Mandelbrot halmaz a `std::complex` osztálytal

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/blob/master/mandelbrot/3.1.2.cpp>

Az előző programhoz képest az eltérések a következők: A felhasználó parancssori argumentumként megadhat 8 paramétert, de nem kötelező, ekkor az alapértelmezett paraméterekkel fog lefutni a program, valamint most a c++ beépitett complex osztályát fogjuk használni a komplex számsíkon való lépegetéshez. A kép itt most színesebb lesz, valami ilyesmi:



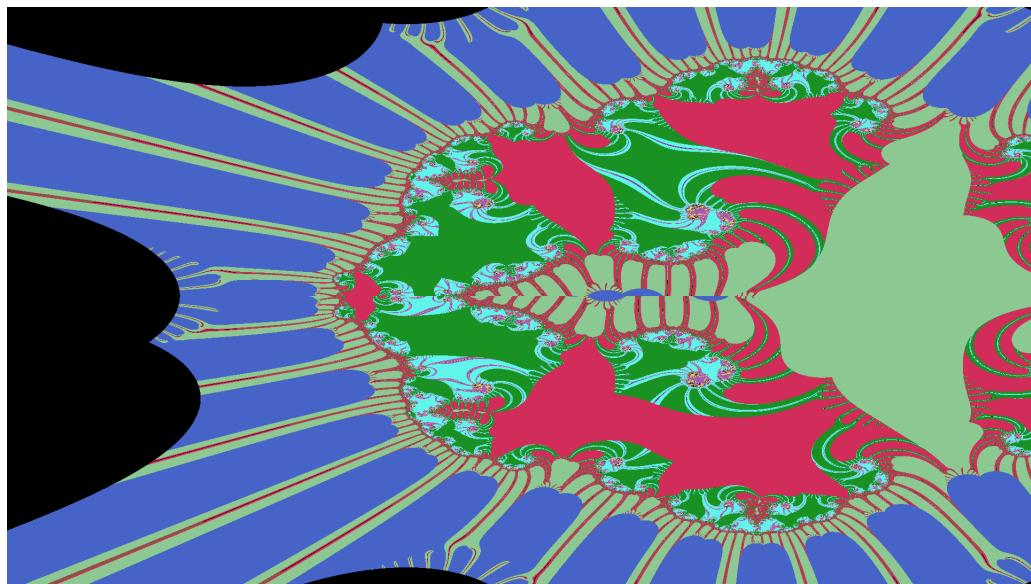
5.3. ábra. Komplex mandelbrot halmaz

### 5.3. Biomorfok

Megoldás videó: <https://youtu.be/IJMbgrzY76E>

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/Biomorf](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/Biomorf)

A biomorfok élő organizmusokra emlékeztető alakzatok, de nem feltétlenül biotikus eredetűek, hanem akár számítógéppel generáltak, mint például most a mi esetünkben. A paramétereiktől függően sokféle alakzatot felvehetnek, mint például ez:



5.4. ábra. Biomorf

#### 5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása:

#### 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 forrása:

Megoldás videó:

Megoldás forrása:

#### 5.6. Mandelbrot nagyító és utazó Java nyelven

## 6. fejezet

# Helló, Welch!

### 6.1. Első osztályom

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat... térd ki arra is, hogy a JDK forrásaiban a Sun programozói pont úgy csinálták meg ahogyan te is, azaz az OO nemhogy nem nehéz, hanem éppen természetes neked!

```
class Random {

public:
 Random();
 ~Random() {}
 double get();

private:
 bool exist;
 double value;

};
```

Létrehozzuk a Random nevű osztályt, melynek elemeit két részre osztjuk: publikus, azaz a class-on kívül is láthatóak/elérhetőek, valamint privát, mely elemeknek elérése a classon belülre korlátozódik. A public részben létrehozunk egy konstruktorot és egy destruktort. Nevükönél értelemszerűen adódik, hogy az egyik az objektumok létrehozását végzi, másik pedig az eltakarításukat. A get függvényünkkel pedig lekérjük,

hogy mit is generáltunk. Privát állományunkba kerül egy bool ami vizsgálja, hogy van-e már korábbi ilyen érték, amit pedig a value-ban tárolunk.

```
double Random::get() {
 if (!exist)
 {
 double u1, u2, v1, v2, w;

 do {

 u1 = std::rand () / (RAND_MAX + 10.0);
 u2 = std::rand () / (RAND_MAX + 1.0);
 v1 = 2 * u1 - 1;
 v2 = 2 * u2 - 1;
 w = v1 * v1 + v2 * v2;
 }
 while (w > 1);

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

 value = r * v2;
 exist = !exist;
 return r * v1;
 }

 else
 {
 return value;
 exist = !exist;
 }
};
```

Az algoritmus 2 random számot generál, az egyiket eltárolja a value-ban, a másikat pedig visszaadja a get() függvény. A value-ban lévőt akkor adja vissza, ha az exist igaz, magyarázatban már van korábbi random érték.

```
int main()
{
 Random rnd;

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

}
```

Végül generálunk 10 darab random számot.

Ugyanez Javaban:

```
public class PolarGen {

 public final static int RAND_MAX = 32767;
 private static boolean bExists;
 private double dValue;
 static Random cRandomGenerator = new Random();

 public PolarGen() {
 bExists = false;
 cRandomGenerator.setSeed(20);
 };

 public double PolarGet() {
 if (!bExists)
 {
 double u1, u2, v1, v2, w;

 do{
 u1 = cRandomGenerator.nextInt (RAND_MAX) / (RAND_MAX + 1.0);
 u2 = cRandomGenerator.nextInt (RAND_MAX) / (RAND_MAX + 1.0);
 v1 = 2 * u1 - 1;
 v2 = 2 * u2 - 1;
 w = v1 * v1 + v2 * v2;
 }
 while (w > 1);

 double r = Math.sqrt ((-2 * Math.log (w)) / w);

 dValue = r * v2;
 bExists = !bExists;

 return r * v1;
 }

 else
 {
 bExists = !bExists;
 return dValue;
 }
 };
```

```
public static void main(String args[])
{
 PolarGen cPolarGen = new PolarGen();
 double dEredmeny = cPolarGen.PolarGet();
 System.out.println(dEredmeny);
}

}
```

Annyi változás van (a nyelvspecifikus szintaktikát leszámitva), hogy itt minden a PolarGen osztályon belülre kerül, még a main függvényünk is.

## 6.2. LZW

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

Megoldás videó:

Megoldás forrása:

Kis történelmi kitekintés: Lempel–Ziv–Welch, azaz LZW egy univerzális adatvesztés nélkül járó tömörítési eljárás, melyet Abraham Lempel, Jacob Ziv és Terry Welch készített. Az algoritmus köz kedvelt egyszerű implementálhatóságának hála. A Unix rendszerek file-ok tömörítése esetén (compress) valamint a GIF képfájlok is ezt az algoritmust használják, de mára a PNG már képformátum már elterjedtebb.

```
typedef struct binfa
{
 int ertek;
 struct binfa *bal nulla;
 struct binfa *jobb_egy;

} BINFA, *BINFA_PTR;
```

Először is elkészítünk egy binfa struktúrát, amiben három dolgot definiálunk, az egyik egy int változó, az érték, a másik kettő pedig (a fa szerkezet miatt) a gyerekeire mutató mutatók (jobb és bal gyerek).

```
BINFA_PTR
uj_elem ()
{
 BINFA_PTR p;

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

```
 return p;
}
```

Ez a funkció fog tárhelyet foglalni az új elemek számára a memóriában, majd vissza is adja a lefoglalt területre mutató pointert. Hogyha elfogyna a memória akkor hibát dob a program és leáll.

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

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

A main függvényünk elején létrehozunk egy karakter változót amibe majd a beolvasott karaktereket fogjuk tárolni, továbbá léterhuzzuk a gyökér értékét, ami most a / jel lesz, majd a fa mutatót beállítjuk, hogy a gyoker-re mutasson.

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

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

Itt jön az input beolvasása (standard inputról (0), b bufferba, 1 byte-ot (azaz 1 karaktert egyszerre). Majd megvizsgáljuk, hogy a kapott karakter 1-es vagy 0-ás-e és annak megfelelően megvizsgáljuk megint, hogy ahova a fa mutató mutat (első futásnál ez ügye mindenkor a gyökér) elemnek az adott gyereke üres-e. Ha igen, akkor létrehozunk egy új elemet az uj\_elem() függvényel, értékét beállítjuk a megfelelőre(0-ás vagy 1-es) és megmondjuk hogy ennek az új elemnek még nincsenek gyerekei, tehát beállítjuk őket NULL-ra. Ezután a fa mutatót visszaállítjuk a gyökérre. Ha viszont a vizsgált gyerek nem üres, tehát már áll ott 1-es vagy 0-ás, akkor a fa mutatóval rálépünk arra a gyerekre amit éppen beolvastunk és várjuk a következő vizsgálandó karaktert, hogy legközelebb már ennek a gyereknek a gyerekeit vizsgálhassuk ugyanezzel a módszerrel.

Végül jöhet a kiiratás és a memória felszabadítás:

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

static int melyseg = 0;
int max_melyseg = 0;

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

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

A szabadit függvényt talán nem kell túltárgyalni, rekurzívan törli az elemeket a memóriából a gyökértől kezdve. A kiir függvény már kicsit trükkösebb: itt történik a fa-struktúra kialakítása és annak a standard outputra való kiírása. Először is rekurzívan bejárjuk a jobb oldali gyereket (meghívjuk rá is a kiir függvényt), majd feldolgozzuk a for cikluson belül a gyökeret, ezután ismét rekurzívan bejárjuk a bal oldali gyermeket. Ezt a bejárást hivjuk inorder bejárásnak.

### 6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása:

Az előző feladatban tárgyalt binfa tökéletesen megfelel számunkra ebben a feladatban is, az egyetlen rész amelyen módosítani fogunk az a kiir függvényben lesz. Eddig ügye a kiir függvény inorder bejárást végzett a fán, ami azt jelenti, hogy előbb a jobboldali gyermeket járjuk be, aztán a gyökeret dolgozza fel, majd a baloldali gyereket járja be ( mindez rekurzívan). A preordernél előbb feldolgozzuk a gyökeret, majd bejárjuk előbb a bal majd a jobb oldali gyereket. A postorderben pedig a gyökeret dolgozzuk fel utoljára. Nézzük is hogyan néz ki a postorder bejárás:

```
void kiir (BINFA_PTR elem)
{
 if (elem !=NULL)
 {
 ++melyseg
 if (melyseg>max_melyseg)
 max melyseg = melyseg;
 kiir(elem->jobb_egy);
 kiir(elem->bal nulla);
 for (int i=0; i<melyseg; i++)
 printf(" --- ");
 printf("%c(%d)\n", elem->ertek<2 ? '0' + elem->ertek : elem->erteke, melyseg-1);
```

```
--melyseg
}

}
```

Látható, hogy most a for ciklus amelyben a paraméterként kapott elemet (a gyökeret) feldolgozzuk a két rekurzív hívás utánra került, így előbb bejárjuk a jobb oldali gyermekeit a fának, majd a bal oldalit és csak aztán dolgozzuk fel a gyökeret. A preorder bejárás ennek az ellentéte, előbb feldolgozzuk a gyökeret és csak aztán a gyerekeit:

```
void kiir(BINFA_PTR elem)
{

 if (elem !=NULL)
 {
 ++melyseg
 if (melyseg>max_melyseg)
 max melyseg = melyseg;
 for (int i=0; i<melyseg;i++)
 printf("---");
 printf("%c(%d)\n",elem->ertek<2 ? '0' + elem->ertek : elem->
 ertek, melyseg-1);
 kiir(elem->jobb_egy);
 kiir(elem->bal nulla);

 --melyseg
 }

}
```

## 6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása: [https://progater.blog.hu/2011/03/31/imadni\\_fogjatok\\_a\\_c\\_t\\_egy\\_emberkent\\_tiszta\\_szivbol](https://progater.blog.hu/2011/03/31/imadni_fogjatok_a_c_t_egy_emberkent_tiszta_szivbol)

Kezdjük is azzal, hogy a C struktúránkat egy class-á, azaz osztállyá varázsoljuk.

```
class LZWBinFa
{
public:
 LZWBinFa (char b = '/') : betu (b), balNulla (NULL), jobbEgy (NULL) ←
 {};
 ~LZWBinFa () { };
}
```

Nézzük hogyan tovább.

```
void operator<<(char b)
{
 if (b == '0')
 {
 // van '0'-s gyermek az aktuális csomópontnak?
 if (!fa->nullasGyermek ()) // ha nincs, csinálunk
 {
 Csomopont *uj = new Csomopont ('0');
 fa->ujNullasGyermek (uj);
 fa = &gyoker;
 }
 else // ha van, arra lépünk
 {
 fa = fa->nullasGyermek ();
 }
 }
 else
 {
 if (!fa->egyesGyermek ())
 {
 Csomopont *uj = new Csomopont ('1');
 fa->ujEgyesGyermek (uj);
 fa = &gyoker;
 }
 else
 {
 fa = fa->egyesGyermek ();
 }
 }
}
```

Nézzük mi is történik itt. Ez a kódrészlet már ismerős a program C változatából, itt vizsgálta a programunk, hogy 0-ást vagy 1-est kapott-e, és cselekedett utána annak megfelelően. Itt ugyanez történik csak egy kis

módositással. Az operator<< segítségével ugynevezett shiftelést hajtunk végre. Ezzel a funkcióval a bemenetként kapott elemeket rögtön belerakjuk a fa struktúráinkba, de ez majd a végén a main függvénynél lesz látványos és fog értelmet nyerni. A csomopontból új osztályt készítettünk, melyből példányositani a new szócskával tudunk. Vessünk is egy pillantást a csomópont osztályra, melyet az LZWBInFa osztályon belül hoztunk létre:

```
private:

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

A Csonopont alapértelmezett kunstroktorában a csomópont értékét q '/'-re állítjuk, bal és jobb gyerekének pedig NULL NULL értéket adunk. A nullasGyermek és egyesGyermek függvények, melyek a bal és jobb gyerekre mutató mutatót fognak visszaadni. Az ujEgyesGyermek és ujNullasGyermek pedig a jobb és bal gyermek mutatóját állítja a paraméterként átadott csomópontokra. Mostmár értelmet nyer az egyel fentebbi kód részlet is, hogy a

```
if (b == '0')
{
```

```
if (fa->bal nulla == NULL)
{
 fa->bal nulla = uj elem ();
 fa->bal nulla->ertek = 0;
 fa->bal nulla->bal nulla = fa->bal nulla->jobb egy = NULL;
 fa = gyoker;
}
```

kódból, hogyan lett:

```
if (b == '0')
{
 if (!fa->nullasGyermek ())
 {
 Csomopont *uj = new Csomopont ('0');
 fa->ujNullasGyermek (uj);
 fa = &gyoker;
 }
}
```

Vessünk hát egy pillantást a main függvényünkre, ahol igazából már csak a beolvasás zajlik egy while ciklussal:

```
int main ()
{
 char b;
 LZWBinFa binFa;

 while (std::cin >> b)
 {
 binFa << b;
 }

 binFa.kiir ();
 binFa.szabadit ();

 return 0;
}
```

Tehát létrehozzuk egy LZWBinFa példányt melyet elnevezünk binFa-nak, majd amíg van bemenet addig azt szépen pakoljuk is bele egyenes a binFa-ba. Ezért volt szükségünk az operator<< túltöltésére, azaz valamelyest újradefiníálására, hogy ezt megtehessük. Ez természetesen nem a mi alkotásunk, a shiftelés

egy alapból létező dolog, mi csak megmondtuk, hogy számunkra hogyan is lenne megfelelő a működése jelen esetben.

## 6.5. Mutató a gyökér

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

Megoldás videó:

Megoldás forrása:

## 6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása:

## 7. fejezet

# Helló, Conway!

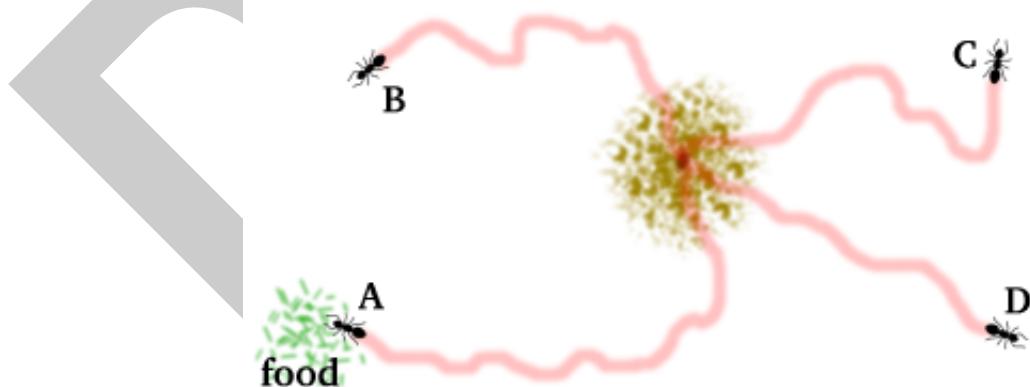
### 7.1. Hangyszimulációk

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

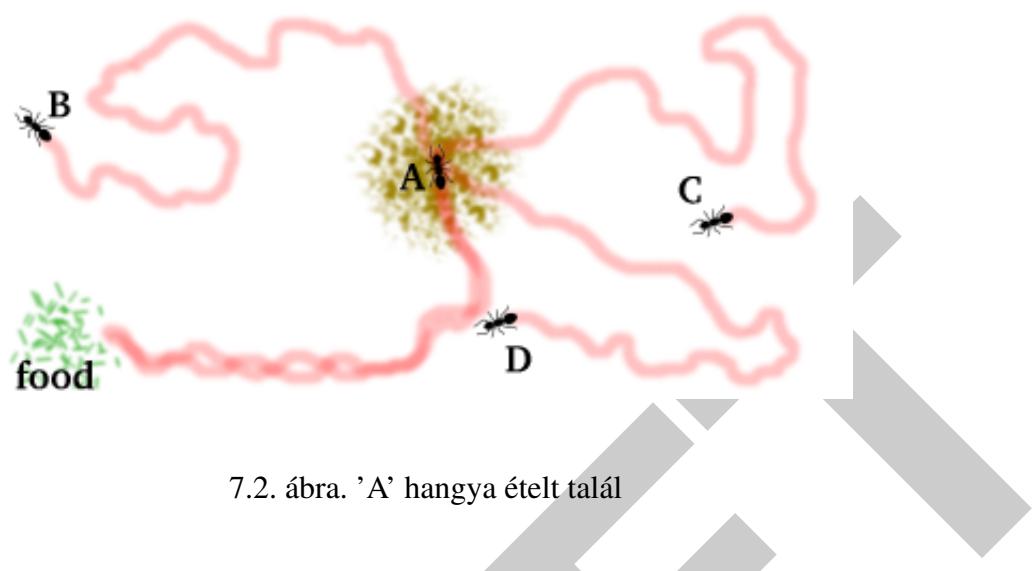
Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

Megoldás forrása: <https://github.com/vajkone/prog1/tree/master/conway>

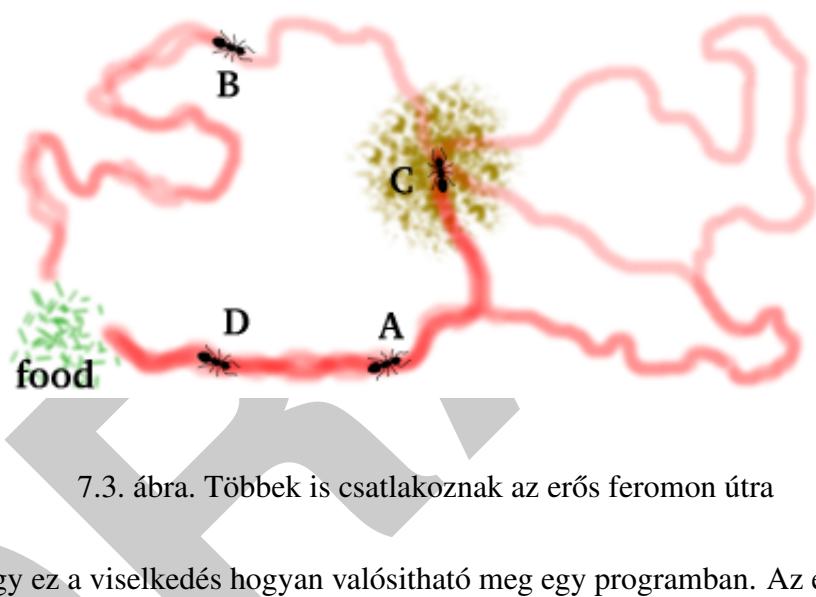
A hangyák mikor elhagyják a hangyabolty/fészküket, hogy élelmet vagy egyéb erőforrásokat találjanak akkor véletlenszerű irányokban indulnak kereszbe-kasul és amerre csak mennek egyfajta feromon-csikot húznak. Ez több szempontból is hasznos, először is, hogyha élelmet találnak, azt felveszik és már indulnak is vissza követve a saját feromon csikjukat ezáltal könnyen hazatalálva akár egy akadályokkal teli környezetben is. De nem csak ők egyedül használják a saját feromon-útjukat, hanem mikor visszaérnek a bolyba, akkor informálják a többieket, hogy amerre voltam, ott bizony élelmet találtam. De hogyan kommunikálják ezt le a hangyák? Nos természetesen ezt is feromon csikokkal. Ugyanis visszafele is húzza a csikot, megerősítve azt.



7.1. ábra. Hangyák ételt keresnek

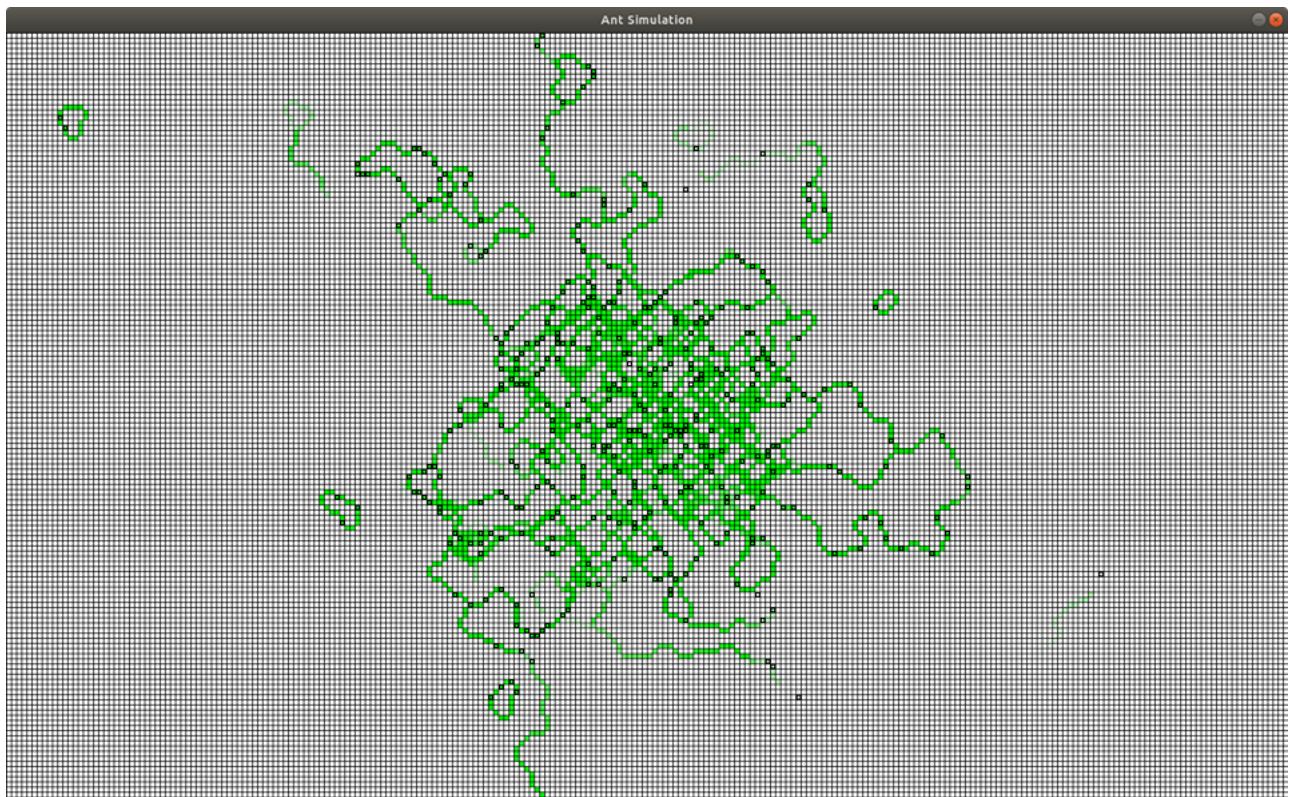


7.2. ábra. 'A' hangya ételt talál

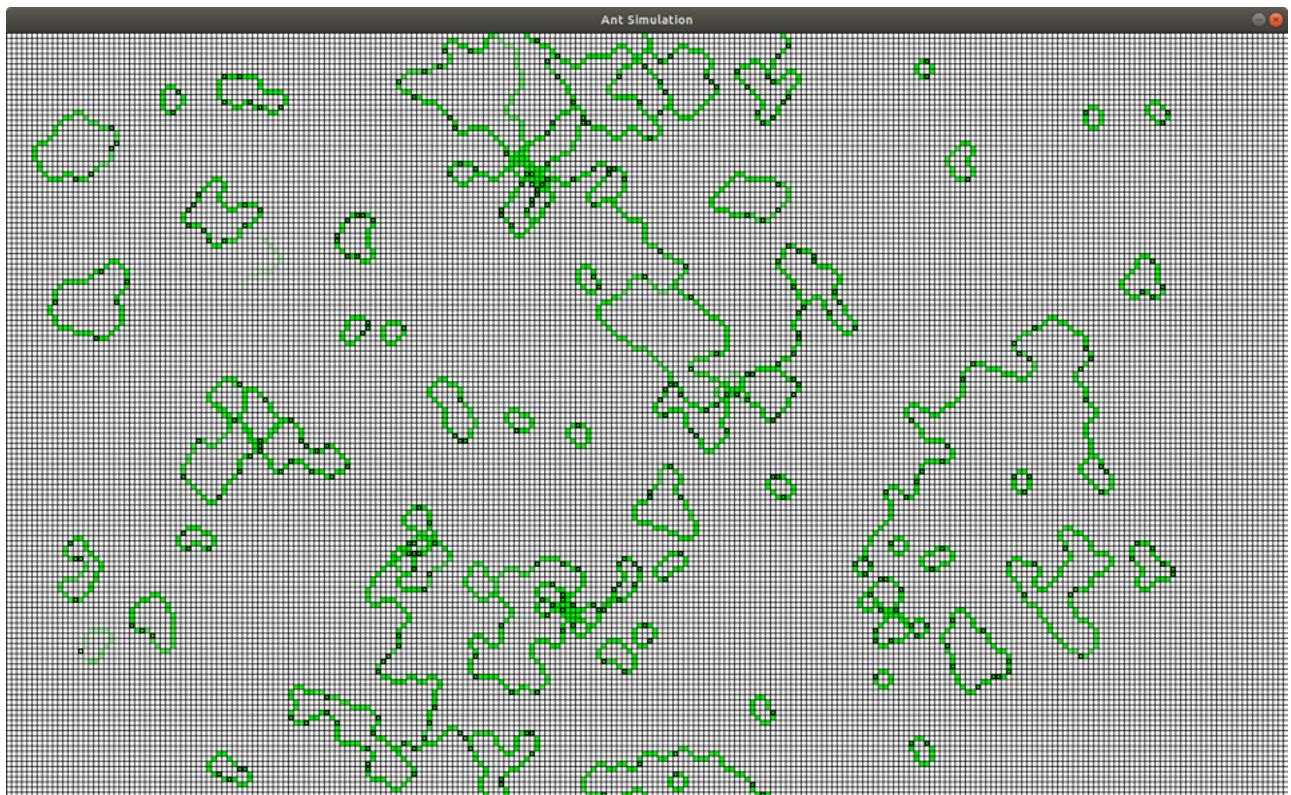


7.3. ábra. Többek is csatlakoznak az erős feromon útra

Nézzük hát meg hogy ez a viselkedés hogyan valósítható meg egy programban. Az elv nagyjából ugyanaz, a program ablakját cellákra osztjuk és hangyákat indítunk el középről, a bolyból. A hangyák egy idő után elkezdenek azon hangyák felé tendálni akiknek erősebb a feromon csikjuk. Elhagyva a sajátukat, amiknek erőssége pedig gyengül.

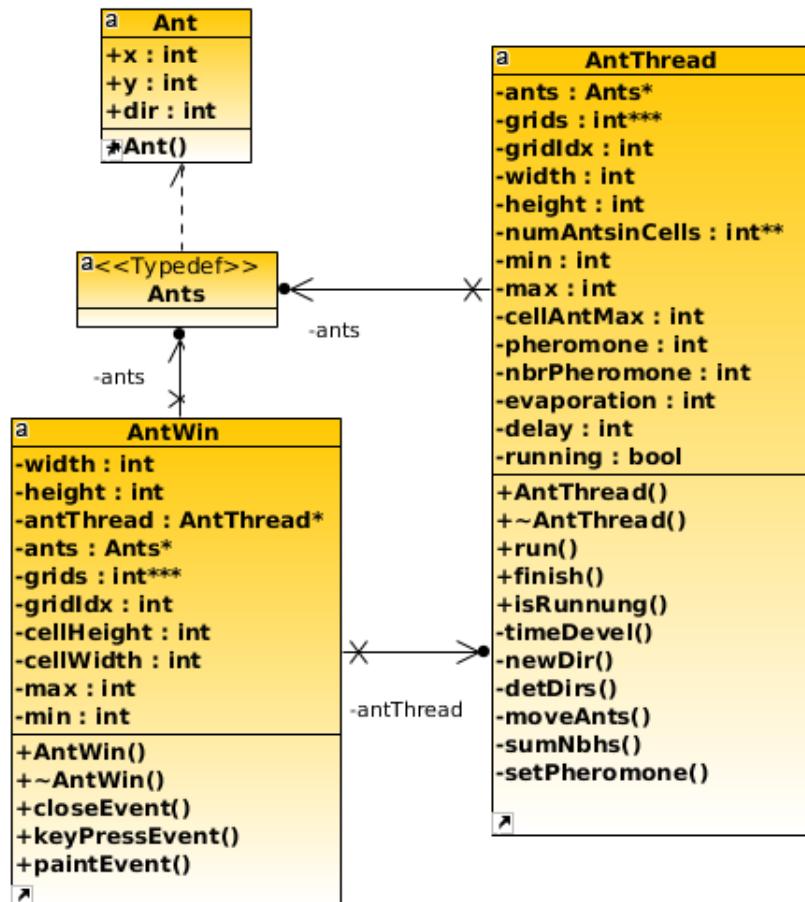


7.4. ábra. A hangyák útnak indulnak



7.5. ábra. Erősebb és gyengébb útvonalak alakulnak ki

A program több összetevőből is áll, az alábbi ábra jól mutatja, hogy mik is működtetik a programot:



7.6. ábra. Program összetevők

## 7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása: <https://github.com/vajkone/prog1/tree/master/conway/Sejtautomata>

Megoldás forrásának forrása: [Itt](#)

Az életjáték, eredeti nevén Game of life egy sejtautomata, melyet John Horton Conway dolgozott ki 1970-ben. A játék valójában nem igényel játékost, csak annyiban, hogy valaki elinditsa és/vagy kezdi paraméterekkel ellássa. Ezután figyelemmel lehet követni, hogyan hogyan fejlődnek a sejtek. De mi alapján történik a fejlődés? Nézzük meg a játék szabályait:

A játékban a sejteket négyzetek ábrázolják, melyek kitöltik az egész játékteret. Ezek a sejtek két állapotban létezhetnek: élő vagy halott. A fejlődés a sejtek és nyolc szomszédja (horizontális 3, vertikális 3, átlós 2) közti interakció eredménye, 4 előre meghatározott szabály alapján. Ezek a szabályok a következők:

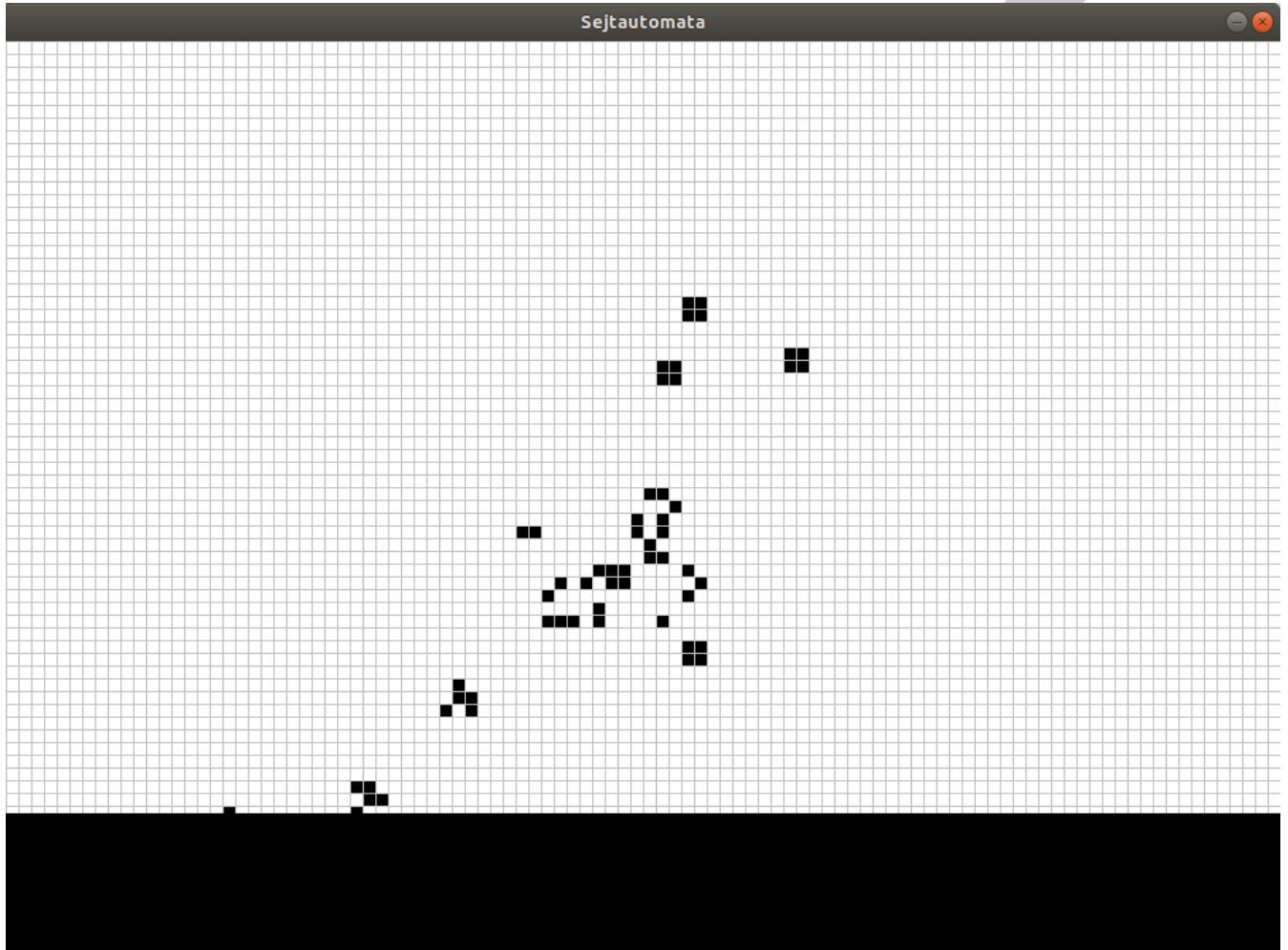
Bármely élő sejt, melynek kevesebb mint két élő szomszédja van, meghal.

Bármely élő sejt, melynek kettő vagy több élő szomszédja van, tovább él a következő generációba.

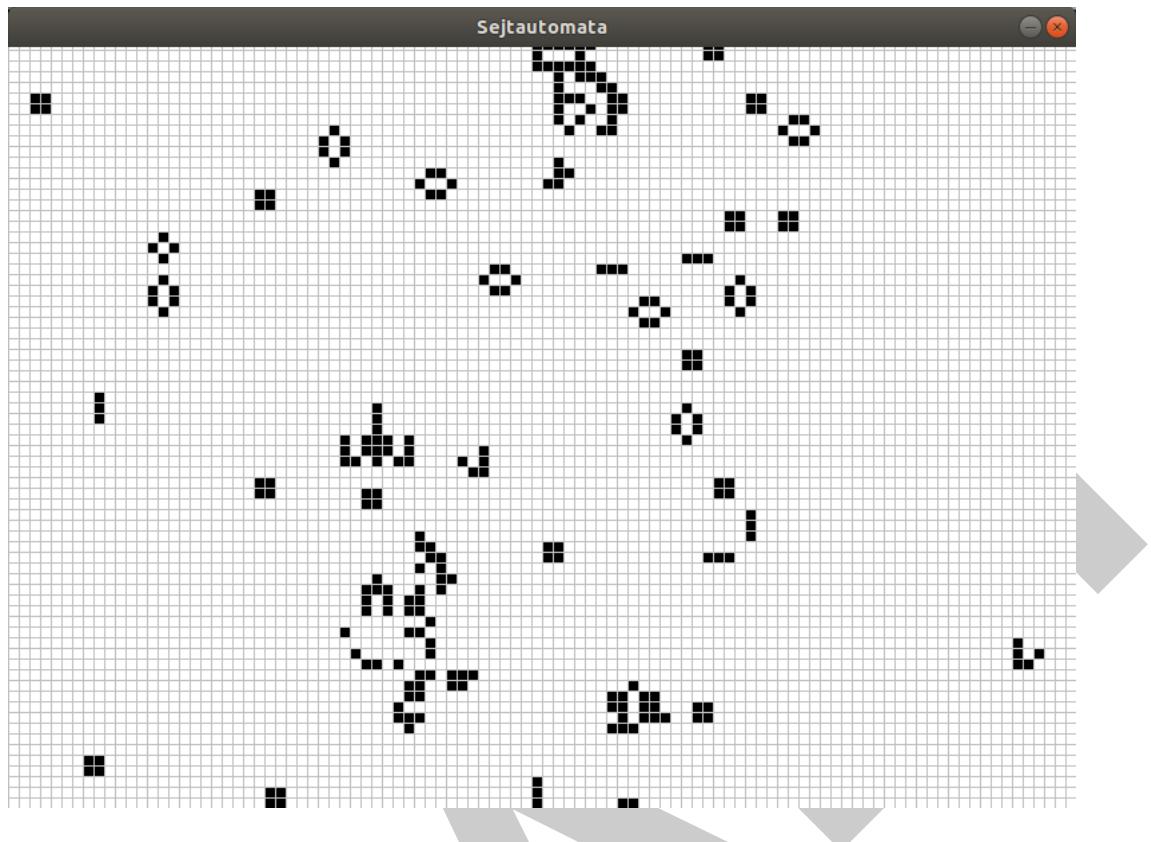
Bármely élő sejt, melynek háromtól több szomszédja van, meghal.

Bármely halott sejt, melynek pontosan három szomszédja van élővé válik.

A fent említett generáció jelentése nem más mint egy úgynévezett tick. Mind a sejtek halála és születése egyszerre, egy tick-nyi idő alatt történik. Nézzünk pár pillanatképet a programból:

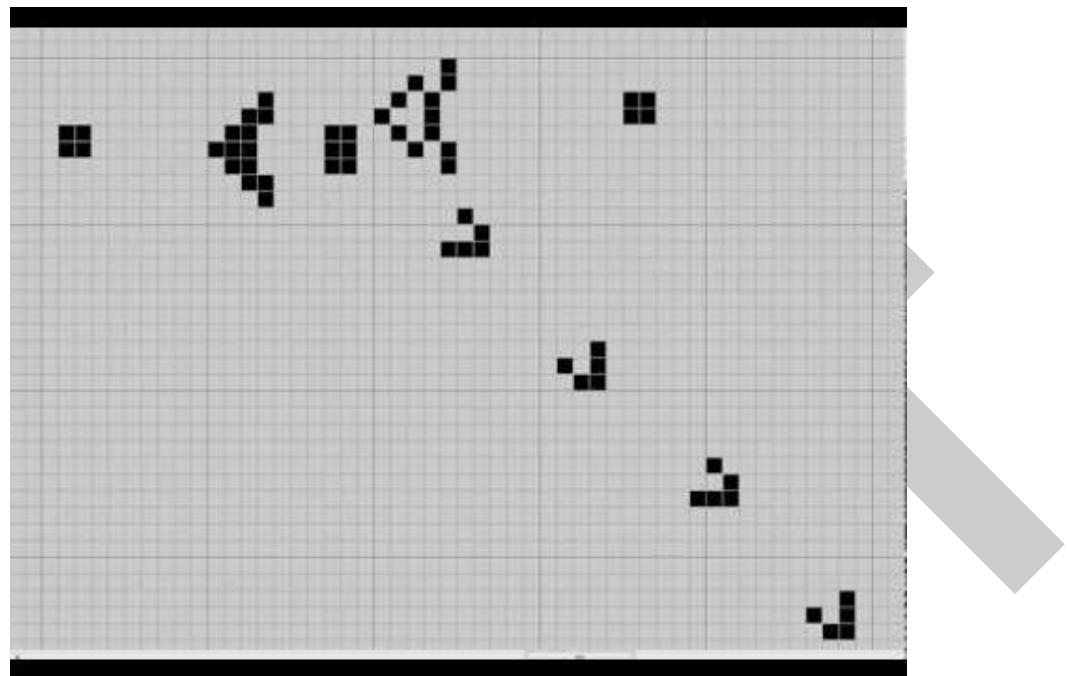


7.7. ábra. Sejtautomata



7.8. ábra. Sejtautomata 2

A sejtek bizonyos kezdeti pozícióit prezicen megadva igazán érdekes sejtmozgásokat érhetünk el. A leglátványosabbaknak nevei is vannak, ilyen például a sikló-kilövő is, amely folyamatosan "siklórepülőket" reptet. Ezt valahogy így kell elképzelni:



7.9. ábra. Sikló

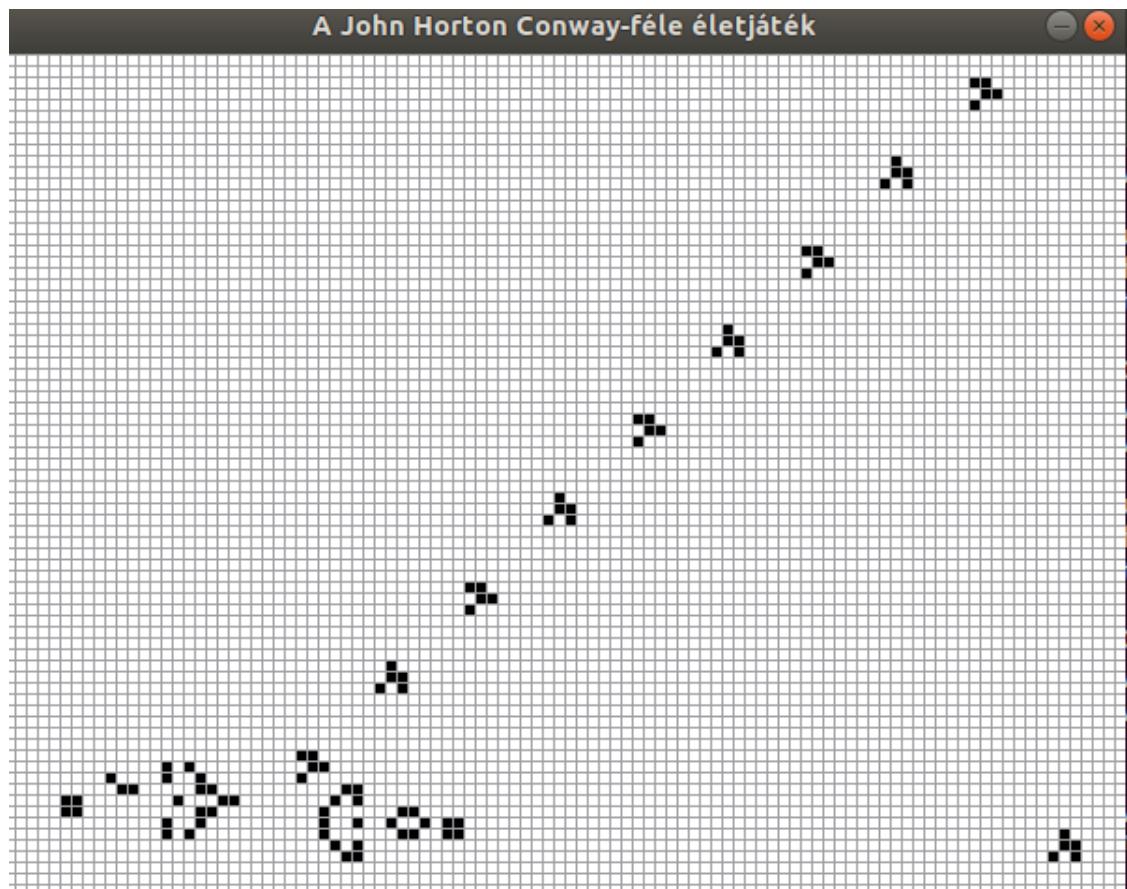
### 7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás video:

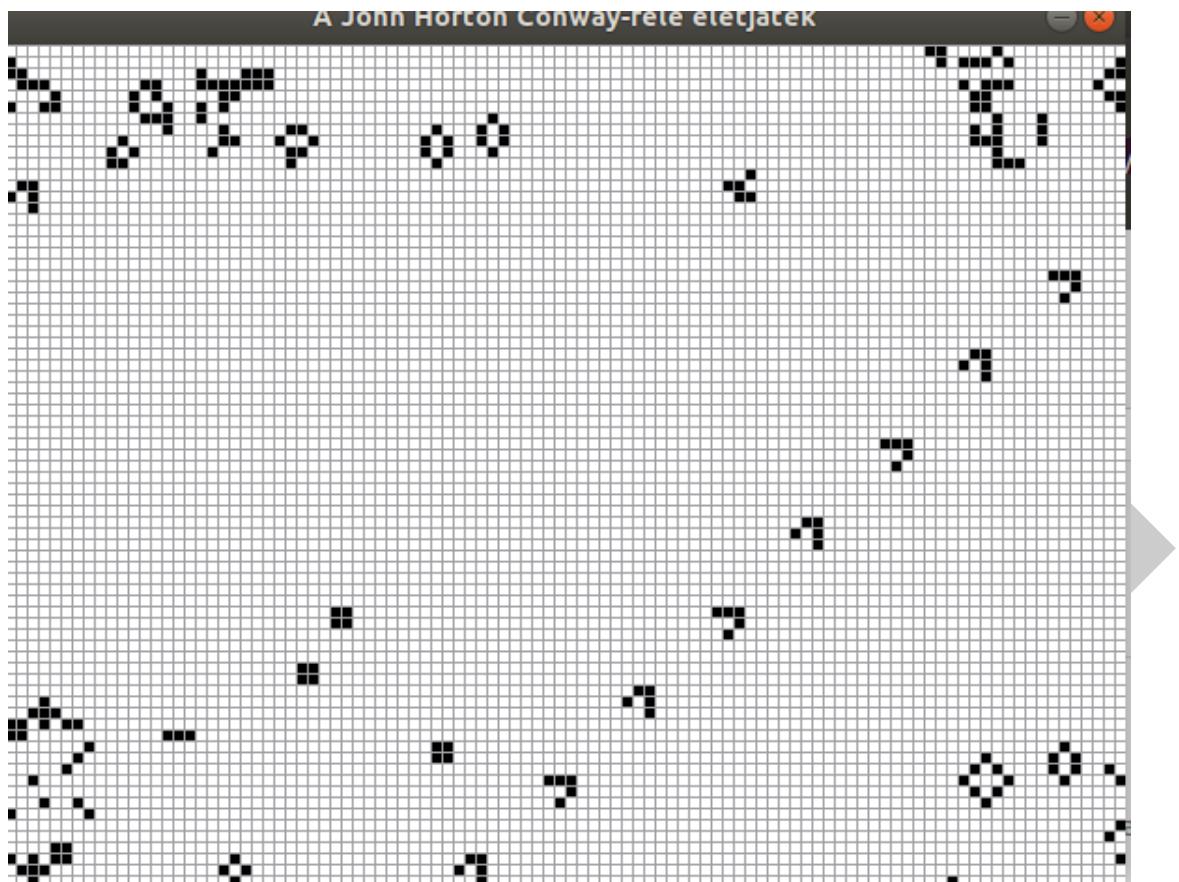
Megoldás forrása: <https://github.com/vajkone/prog1/tree/master/conway/sejtauto>

Az élet játékának szabályai ugyanazok, egyedül a környezet változott. Futtatáshoz: sudo apt-get install libqt4-dev, qmake Sejtauto.pro, make, ./Sejtauto Ime a siklókilövő megvalósítása:



7.10. ábra. Siklókilövés

Egy idő után viszont nálam a dolog felbomlott:



7.11. ábra. Siklókilövés 2

Egy kis érdekesség a végére: A Google-be bepötyögve, hogy 'Conway's Game of Life' az oldalon elindul a játék az oldal szélén :)



7.12. ábra. Életjáték a google keresőben

## 7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása: <https://github.com/nbatfai/esport-talent-search>

Felülnézetes játékoknál, ahol egy adott karaktert irányítunk (pl Mobák: Lol, Dota) gyakran megesik, hogy a játszma egy pontján, például egy teamfight-ban 10 ember karaktere sűrűsödik egy képernyőnyi területre és kezd el egyszerre mindenféle villogó, robbanó, repenő, suhanó stb képességet használni, hogy az ellenfél csapatot legyőzze. Ebben a forgatagban fontos, hogy tisztában legyen a játékos a saját karakterének poziciójánál, ugyanis a jó pozicionálás döntő fontosságú lehet egy-egy ütközet kimenetében.

Ez a program azt hivatott mérni, hogy egy ilyen szituációban meddig tudnánk a karakterünket figyelemmel kísérni és ha szem elől veszítjük, akkor milyen gyorsan lelünk rá a képernyőn ismét.



## 8. fejezet

# Helló, Schwarzenegger!

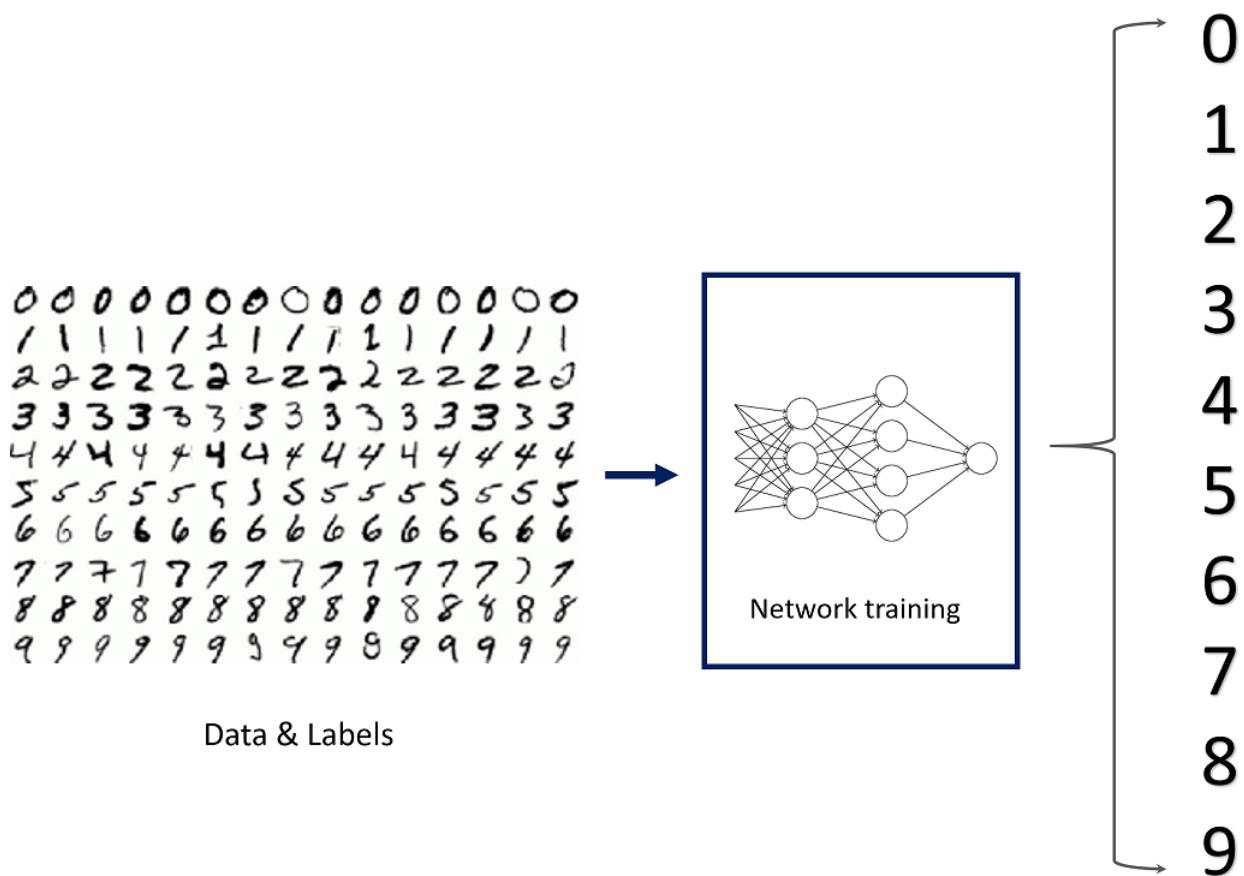
### 8.1. Szoftmax Py MNIST

aa Python

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

Megoldás forrása:

Az MNIST adatbázis egy hatalmas database melyben mely kézzel írott számok képeit tartalmazza. Ezt a több mint 60000 képet számláló adatbázist előszeretettel használják képfeldolgozó rendszereknél és gépi tanulásnál, mind a betanítás mind a tesztelés során.



8.1. ábra. A betanítás folyamata

## 8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

## 8.3. Minecraft-Malmö

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

## 9. fejezet

# Helló, Chaitin!

### 9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás video:

Megoldás forrása:

A Lisp a Fortran után a második legrégebbi magasszintű programozási nyelv. Magát a nyelvet John McCarthy alkotta meg 1958-ban az MIT-n. A név a LISt Processor szavak összevonásából ered és mutatja azt is, hogy a nyelv fő adatstruktúrája a lista. Ugyan a nyelv maga nagyon régi és gondolhatnánk hogy elavult is, de ez nem teljesen igaz. A mai napig vannak területek ahol használják. Mi most például a jól ismert GIMP képszerkesztő programban script írásához fogjuk felhasználni. Van egy-két dolog ami elsőre furcsának fog tűnni a Lisp használatánál a mai programnyelvekhez képest. Az egyik ilyen furcsaság, hogy a műveleteket prefix alakban kell megadni, azaz ha 3-t és 4-t összeszeretnénk adni azt így tennénk: (+ 3 4).

Az iteratív megoldáshoz egy ciklust fogunk használni. Ugyan nem for és nem és while a neve Lisp-ben, de attól a működési elv ugyanaz. Nézzük is hogyan néz ki:

```
(define (fakt x)
 (if (= x 0)
 (set! x 1)
)
 (let loop ((variable (- x 1)))
 (if (> variable 1)
 (begin
 (set! x (* x variable))
 (loop (- variable 1))
)
)
)
 n
)
```

Definiáljuk a fakt függvényünket, melynek átadunk majd egy számot. Ha ez a szám 0, akkor az eredményünk 1 lesz, ugyanis tudjuk hogy  $0!=1$ . Ha ez az ellenőrzés megtörtént, létrehozunk egy változót a let parancsal a paraméterként átadott szám alapján egyel kevesebb értékkel, majd ha az így kapott változónk nagyobb mint 1, akkor indítunk egy ciklust, hogy a változónkat és az eredeti számot összeszorozza és az eredeti szám értékét a szorzat értékére állítjuk a set! parancsal, majd a változónkat csökkentjük eggyel és az x-et ismét szorozzuk az új változóval, mindezt egészen addig amíg a változónk 1 nem lesz.

Ugynéz rekurzív megvalósítással:

```
(define (fakt n)
 (if (< n 1) 1
 (* n (fakt (- n 1)))))
```

Előre szintén ellenőrzés, hogy n kisebb-e mint 1, mert akkor az eredmény 1, amúgy meg n-t szorozzuk meg ugyanezzel a függvénnyel csak a paramétere most legyen n-1, azaz egyel kisebb.

## 9.2. Gimp Scheme Script-fu: króm effekt

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

Megoldás videó: [https://youtu.be/OKdAkI\\_c7Sc](https://youtu.be/OKdAkI_c7Sc)

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/GIMP\\_Lisp/Chrome](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Chrome)

Tanulságok, tapasztalatok, magyarázat...

## 9.3. Gimp Scheme Script-fu: név mandala

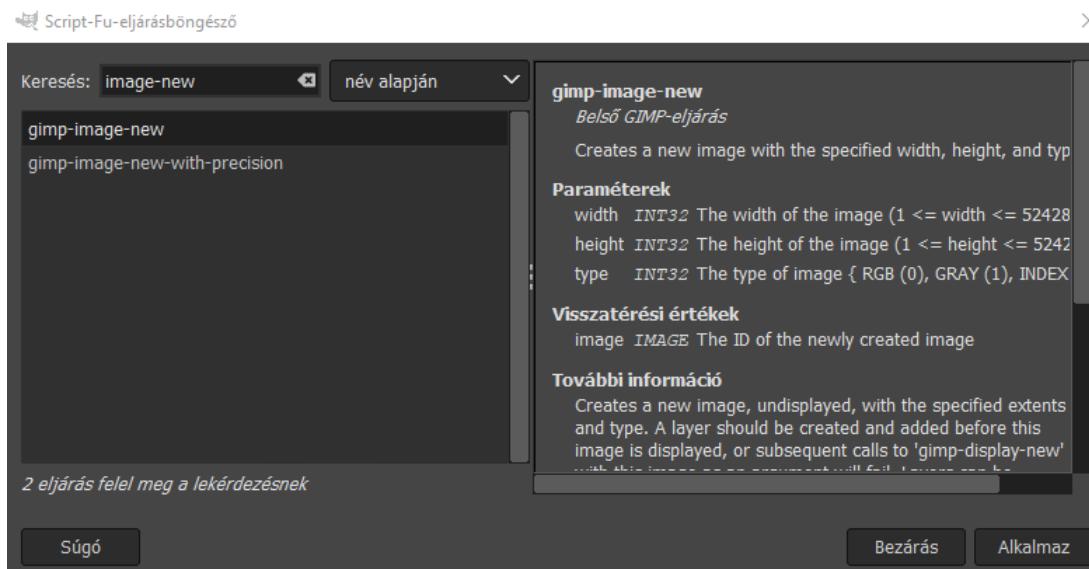
Írj olyan script-fu kiterjesztést a GIMP programhoz, amely név-mandalát készít a bemenő szövegből!

Megoldás videó: [https://bhaxor.blog.hu/2019/01/10/a\\_gimp\\_lisp\\_hackelese\\_a\\_scheme\\_programozasi\\_nyelv](https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelese_a_scheme_programozasi_nyelv)

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/GIMP\\_Lisp/Mandala](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Mandala)

A mandala egy spirituális szimbólum, elsősorban az indiai vallásokban; a hinduizmusban és buddhizmusban találkozhatunk vele és leggyakrabban az univerzumot reprezentálja. A szó eredeti szanszkrit jelentése kör, korong, körszelet. Ahogy neve is mutatja a mandala minden körkörösen ábrázol valamit, ami kötődik a valláshoz, meditációhoz, nyugalomhoz. Mi most kicsit elrugászkodunk ezektől a hagyományos eszméktől és a nevünket fogjuk egy mandalában kiirni.

GIMP-ben amit menüből kattintások révén elérhetünk, azokat gyakran meg tudjuk irni scriptként is, például, hogy képet hozunk létre, a képhez layeret adunk, színeket állítunk, szöveget készítünk majd a szöveget forgatjuk. Szerencsére ezeket a függvényeket nem kell nekünk elkészíteni, csak implementálnunk kell őket, hiszen a Script-fu eljárásból több mint 1200 függvény között válogathatunk.



9.1. ábra. Eljárásböngésző

Nézzük át a script egyes részeit:

```
(define (script-fu-bhax-mandala text text2 font fontsize width height color ←
 gradient)
(let*
 (
 (image (car (gimp-image-new width height 0)))
 (layer (car (gimp-layer-new image width height RGB-IMAGE "bg" 100 ←
 LAYER-MODE-NORMAL-LEGACY)))
 (textfs)
 (text-layer)
 (text-width (text-width text font fontsize))
 ;;
 (text2-width (car (text-wh text2 font fontsize)))
 (text2-height (elem 2 (text-wh text2 font fontsize)))
 ;;
 (textfs-width)
 (textfs-height)
 (gradient-layer)
)
)
```

Azt már tudjuk, hogy a let-tel tudunk lokális változókat létrehozni. Ilyen változó lesz példul az image, amiben magát a képet tároljuk majd vagy egyes layer-ek.

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

A háttérünknek valamelyen RGB szint adunk (jelenleg zöld), amivel feltöljtük a háttér layer-t.

```
(set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) ←
))
(gimp-image-insert-layer image textfs 0 -1)
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (/ ←
 height 2))
(gimp-layer-resize-to-image-size textfs)
```

A kívánt szöveget egy új layerként a képhez adjuk, offsetet adunk hozzá és méretezzük a kép méretei alapján.

```
(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate-simple text-layer ROTATE-180 TRUE 0 0)
(set! textfs (car (gimp-image-merge-down image text-layer ←
 CLIP-TO-BOTTOM-LAYER)))

(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 2) TRUE 0 0)
(set! textfs (car (gimp-image-merge-down image text-layer ←
 CLIP-TO-BOTTOM-LAYER)))

(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 4) TRUE 0 0)
(set! textfs (car (gimp-image-merge-down image text-layer ←
 CLIP-TO-BOTTOM-LAYER)))

(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 6) TRUE 0 0)
(set! textfs (car (gimp-image-merge-down image text-layer ←
 CLIP-TO-BOTTOM-LAYER)))
```

Itt forgatjuk a bevitt szöveget, de csak egy másolatát mert az eredeti megtartjuk és a forgatott szöveget ugyanarra a layerre rakjuk amin az eredeti szöveg van, azaz egybeolvasszuk őket. Ezt 4-szer tesszük meg.

```
(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width 2) ←
 (/ textfs-width 2)) 18)
 (- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) (+ ←
 textfs-height 36))
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

(gimp-context-set-brush-size 22)
(gimp-edit-stroke textfs)
```

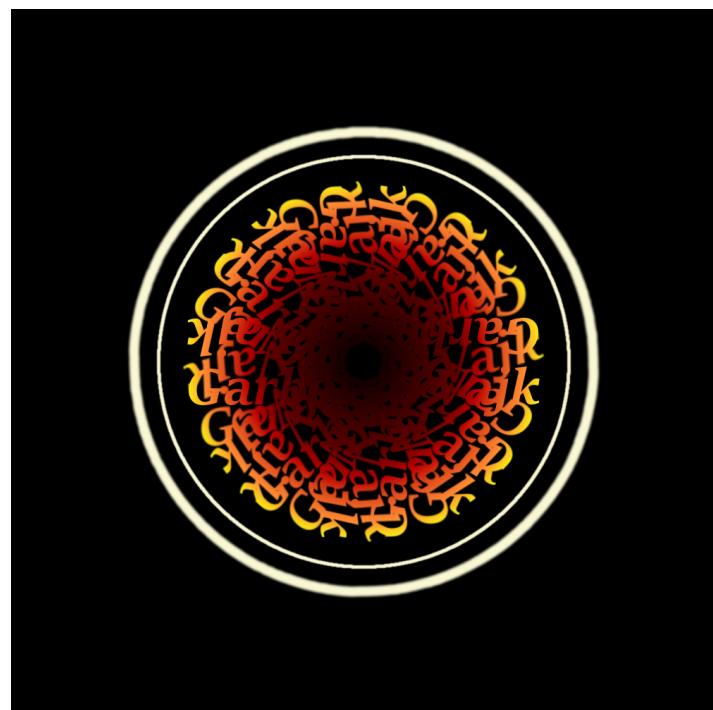
Ellipszis alakú kijelölést rakunk a képünk köre és a kijelölést a választott színnel körbefestjük.

```
(set! textfs (car (gimp-text-layer-new image text2 font fontsize PIXELS ←
)))
(gimp-image-insert-layer image textfs 0 -1)
(gimp-message (number->string text2-height))
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text2-width 2)) (- (/ ←
 height 2) (/ text2-height 2)))

;(gimp-selection-none image)
;(gimp-image-flatten image)

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

Végül megjelenítjük a mandalánkat.



9.2. ábra. Saját mandalám

DRAFT

**III. rész**

**Második felvonás**

**DRAFT**

**Bátf41 Haxor Stream**

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

DRAFT

## 10. fejezet

### Helló, Arroway!

#### 10.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

#### 10.2. 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**

DRAFT

## 10.3. Általános

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

## 10.4. C

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

## 10.5. C++

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

## 10.6. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, [http://arxiv.org/PS\\_cache/math/pdf/0404/0404335v7.pdf](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 érdeklődésekért és hasznos észrevételeikért.

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