

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Mester, Ákos	2019. december 1.	

REVISION HISTORY

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

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	8
2.4. Labdapattogás	8
2.5. Szóhossz és a Linus Torvalds féle BogomIPS	10
2.6. Helló, Google!	10
2.7. 100 éves a Brun tétel	14
2.8. A Monty Hall probléma	14
3. Helló, Chomsky!	15
3.1. Decimálisból unárisba átváltó Turing gép	15
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	16
3.3. Hivatkozási nyelv	17
3.4. Saját lexikális elemző	17
3.5. l33t.1	18
3.6. A források olvasása	19
3.7. Logikus	23
3.8. Deklaráció	24

4. Helló, Caesar!	26
4.1. double *** háromszögmátrix	26
4.2. C EXOR titkosító	28
4.3. Java EXOR titkosító	29
4.4. C EXOR törő	30
4.5. Neurális OR, AND és EXOR kapu	33
4.6. Hiba-visszaterjesztékes perceptron	34
5. Helló, Mandelbrot!	35
5.1. A Mandelbrot halmaz	35
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	37
5.3. Biomorfok	38
5.4. A Mandelbrot halmaz CUDA megvalósítása	39
5.5. Mandelbrot nagyító és utazó C++ nyelven	39
5.6. Mandelbrot nagyító és utazó Java nyelven	42
6. Helló, Welch!	43
6.1. Első osztályom	43
6.2. LZW	46
6.3. Fabejárás	46
6.4. Tag a gyökér	47
6.5. Mutató a gyökér	55
6.6. Mozgató szemantika	61
7. Helló, Conway!	64
7.1. Hangyaszimulációk	64
7.2. Java életjáték	64
7.3. Qt C++ életjáték	65
7.4. BrainB Benchmark	69
8. Helló, Schwarzenegger!	70
8.1. Szoftmax Py MNIST	70
8.2. Szoftmax R MNIST	70
8.3. Mély MNIST	70
8.4. Deep dream	71
8.5. Robotpszichológia	71

9. Helló, Chaitin!	72
9.1. Iteratív és rekurzív faktoriális Lisp-ben	72
9.2. Weizenbaum Eliza programja	72
9.3. Gimp Scheme Script-fu: króm effekt	72
9.4. Gimp Scheme Script-fu: név mandala	72
9.5. Lambda	73
9.6. Omega	73
 III. Második felvonás	 74
10. Helló, Berner-Lee!	76
10.1. Java és C++ összehasonlítás	76
10.2. Bevezetés a mobilprogramozásba	80
11. Helló, Liskov!	82
11.1. Liskov helyettesítés sértése	82
11.2. Szülő-gyerek	82
11.3. Ciklomatus komplexitás	83
11.4. Anti OO	84
12. Helló, Mandelbrot!	87
12.1. A Mandelbrot halmaz	87
12.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	89
12.3. Biomorfok	90
12.4. A Mandelbrot halmaz CUDA megvalósítása	91
12.5. Mandelbrot nagyító és utazó C++ nyelven	91
12.6. Mandelbrot nagyító és utazó Java nyelven	94
13. Helló, Chomsky!	95
13.1. Encoding	95
13.2. OOCWC lexer	97
13.3. l334d1c4	100
13.4. Perceptron osztály	101

14. Helló, Stroustrup!	103
14.1. JDK osztályok	103
14.2. Hibásan működő	104
14.3. String osztály és másoló-mozgató szemantika	107
14.4. ESSZÉ: RSA titkosító eljárás	110
15. Helló, Gödel!	112
15.1. Gengszterek	112
15.2. Custom allokátor	112
15.3. STL map érték szerinti rendezése	115
15.4. Alternatív Tabella rendezése	116
16. Helló, !	118
16.1. OOCWC Boost ASIO hálózatkézelése	118
16.2. SamuCam	119
16.3. BrainB	121
16.4. FUTURE tevékenység editor	123
17. Helló, Lauda!	125
17.1. Port scan	125
17.2. AOP	126
17.3. Junit teszt	128
17.4. Android játék	129
18. Helló, Calvin!	133
18.1. MNIST	133
18.2. Deep MNIST	137
18.3. Androig telefonra a TF objektum detektálója	140
18.4. CIFAR-10	143
IV. Irodalomjegyzék	145
18.5. Általános	146
18.6. C	146
18.7. C++	146
18.8. Lisp	146

Táblázatok jegyzéke

11.1.	85
------------	----

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

```
batfai@entropy:~$ cd glrepos/bhax/thematic_tutorials/bhax_textbook/
batfai@entropy:~/glrepos/bhax/thematic_tutorials/bhax_textbook$ make
rm -f bhax-textbook-fdl.pdf
xmllint --xinclude bhax-textbook-fdl.xml --output output.xml
xmllint --relaxng http://docbook.org/xml/5.0/rng/docbookxi.rng output.xml  ←
--noout
output.xml validates
rm -f output.xml
dblatex bhax-textbook-fdl.xml -p bhax-textbook.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 találsz az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

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:

Egy processzorszál 0%-os használata:

Ahhoz, hogy egy program futtatása közben megközelítőleg 0%-os processzorhasználatot kapjunk `sleep(1)` rendszerhívást kell használnjuk. Ezzel a paranccsal az argumentumként megadott számú miliszekundumig jelezzük az operációsrendszernek, hogy nem szeretnénk a processzort használni.

```
#include <stdio.h>

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

Egy processzorszál 100%-os használata:

```
#include <stdio.h>

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



```
    return 0;
}
```

Ehhez a feladathoz egy egyszerű üres for ciklusra van szükségünk, mivel ilyenkor az OS úgy érzékeli, hogy rengeteg számolnivalója van, ezért sok processzoridőt ad a programnak. Ha használjuk a `sleep(1)` parancsot, akkor az OS tudja, hogy 1 milliszekundumig a processzornak nincs dolga. Tehát hiában van 1 paranccsal több a kódban, mégsem használja a processzort.

Minden processzorszál 100%-os használata:

```
#include <stdio.h>

int main(void)
{
    int x=0;
    #pragma omp parallel
    while (x<1) {
    }
}
```

Ahhoz, hogy a számítógépünk processzorát 100%-osan terheljük egy végtelen ciklussal csatolni kell az "omp.h" könyvtárat a programkódban, majd fordításkor `$gcc -fopenmp ciklus.c -o ciklus` parancsot kell használni.

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

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

Megoldás videó:

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

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

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

A program futtatása, például akár az előző v. c ilyen 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 építő Lefagy2 már nem tartalmaz feltételezett, 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.

Ilyen program nem létezik, ahogy azt Alan Turing is bebizonyította

2.3. Változók értékének felcserélése

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

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

Megoldás forrása:

Két változó értékét logikai utasítás, kifejezés vagy segédváltozó nélkül egy egyszerű matematikai példával lehet a következőképpen

```
#include <stdio.h>

int main()
{
    int a = 5;
    int b = 10;
    a = a + b;
    b = a - b;
    a = a - b;
    return 0;
}
```

2.4. Labdapattogás

Először if-ekkel, majd bármiféle logikai utasítás vagy kifejezés használata nélkül írd 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:

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

int
```

```
main ( void )
{
    WINDOW *ablak;
    ablak = initscr ();

    int x = 0;
    int y = 0;

    int xnov = 1;
    int ynov = 1;

    int mx;
    int my;

    for ( ;; ) {

        getmaxyx ( ablak, my , mx );

        mvprintw ( y, x, "O" );

        refresh ();
        usleep ( 100000 );

        x = x + xnov;
        y = y + ynov;

        if ( x>=mx-1 ) { // elerte-e a jobb oldalt?
            xnov = xnov * -1;
        }
        if ( x<=0 ) { // elerte-e a bal oldalt?
            xnov = xnov * -1;
        }
        if ( y<=0 ) { // elerte-e a tetejet?
            ynov = ynov * -1;
        }
        if ( y>=my-1 ) { // elerte-e a aljat?
            ynov = ynov * -1;
        }

    }

    return 0;
}
```

A feladat célja, hogy olyan függvényt írjunk, ami egy labda pattogását imitálja úgy, hogy az ne hagyja el a terminált. A "labda" lépéseinek sűrűségét az `usleep(x)` paranccsal változthatjuk, ami a függvény futását késlelteti az argumentumként megadott mikroszekundumnyi értékkel. Ez segít abban, hogy a pattogás az emberi szemnek is látható legyen. A működéshez szükség van az `ncurses` könyvtár csatolására, majd fordításkor is meg kell adnunk kapcsolónak (`-lncurses`). A `clear()` függvény opcionális, meghí-

vásával a képernyőn csak egy labda lesz látható, nem fog rajzolni. A kiíratáshoz a `mvprintw` függvényt használjuk, ami a megadott koordinátáknak megfelelő helyen iratja ki a karatkert a képernyőn.

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:

```
#include <stdio.h>

int main()
{
    unsigned long int a = 1, count = 0;

    do
        count++;
    while (a <= 1);

    printf("Szóhossz: %d.\n", count);
    return 0;
}
```

A fenti kódrészlet az adott szó hosszát bitenkénti eltolással számolja meg. Egy változó értékét beállítjuk egyre, majd egy ciklusban addig shifteljük a szót balra, ameddig lehet, majd megszámláljuk, hogy hányszor lett eltolva.

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:

A Google internetes keresőmotor legfontosabb eleme a PageRank algoritmus, amit Larry Page és Sergey Brin fejlesztettek 1998-ban. Alapelve az, hogy egy adott oldal annál jobb, minél több oldal mutat rá és a rá mutató oldalra mennyi mutatóval rendelkező oldal mutat.

Kezdetben minden oldal egy egységnyi ponttal ("szavazattal") rendelkezik, amit eloszt azok között az oldalak között, amikre hivatkozik és a más oldalaktól kapott szavazatot is továbbosztja. Tehát minél több szavazatot kap egy oldal, annál fontosabb, de figyelembe kell venni, hogy a szavazatot leadó oldal mennyire fontos.

Forráskód 4 honlapra:

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

void kiir(double tomb[], int db);
double tavolsag(double pagerank[], double pagerank_temp[], int db);

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

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

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

        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 (tavolsag(PR, PRv, 4) < 0.000001)
            break;
    }

    kiir(PR, 4);

    return 0;
}

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

```
}  
  
double tavolsag(double pagerank[], double pagerank_temp[], int db)  
{  
    double tav = 0.0;  
  
    for (int i = 0; i < db; i++) {  
        tav +=  
            (pagerank[i] - pagerank_temp[i]) * (pagerank[i] -  
            pagerank_temp  
            [i]);  
    }  
  
    return sqrt(tav);  
}
```

A programban található egy végtelen ciklus, ami csak akkor fejeződik be, ha a távolság nagyobb, mint 0,00000001.

```
if (tavolsag(PR, PRv , 4) < 0.00000001)  
break;
```

A mátrixunk egy 4x4-es tömb lesz, ami tárolja, hogy ki mutat kire. Például ha a mátrix első sorában található szám 1-es, az azt jelenti, hogy az első sorelem mutat az első oszlopelemre.

A matematikai számítások elvégzéséhez csatolnunk kell a `stdio` és a `math` könyvtárakat.

A `kiir` eljárás paraméterei `double` és `int` típusú tömbök. Feladata, hogy for `i=0`-tól `db`-ig kiírja a tömb `i`-edik elemét.

```
language="C">  
double tavolsag( double PR[], double PRv[], int n)  
{  
    int i;  
    double osszeg=0.0;  
  
    for(i =0; i<n; ++i)  
        osszeg+= (PRv[i] - PR[i]) * (PRv[i]-PR[i]);  
    return sqrt (osszeg);  
}
```

A fenti kódcsipet kiszámolja az `osszeg` változó értékének, a `Prv` és a `PR` tömbök `i`-edik elemeinek különbségének szorzatát. A `PR` mátrix az aktuális oldal `pagerank`-jét a `PRv` mátrix pedig az egy körrel korábbi értékeket tartalmazza. A függvény végén megkapjuk az `osszeg` változó négyzetgyökét.

Ami a fő matematikai számítást végzi, az a `pagerank` függvény.

```
void pagerank(double T[4][4]){  
    double PR[4] = { 0.0, 0.0, 0.0, 0.0 };
```

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

int i, j;

for(;;){

    for (i=0; i<4; i++){
        PR[i]=0.0;
        for (j=0; j<4; j++){
            PR[i] = PR[i] + T[i][j]*PRv[j];
        }
    }

    if (tavolsag(PR,PRv,4) < 0.0000000001)
        break;

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

kiir (PR, 4);
}
```

Kezdetben a PR tömb minden értéke 0, a folyamat után ez a tömb fogja tartalmazni az eredményeket.

A main() függvényben hozzuk létre az oldalakat szimbolizáló 4x4-es mátrixainkat (tömbeinket). A tömbök mind különféle eseteket szimbolizálnak. A pagerank függvény meghívásra kerül mindhárom alkalommal más és más mátrixokkal a lehetséges eseteket reprezentálva.

A kimenet, ha az egyik oldal nem mutat semmire sem:

Amikor az egyik oldal semmire sem mutat:

```
PageRank [0]: 0.000000
PageRank [1]: 0.000000
PageRank [2]: 0.000000
PageRank [3]: 0.000000
```

A kimenet, ha az egyik oldal önmagára mutat:

Amikor az egyik oldal csak magára mutat:

```
PageRank [0]: 0.000000
PageRank [1]: 0.000000
PageRank [2]: 0.000000
PageRank [3]: 1.000000
```

Végül, eredeti értékekkel

```
PageRank [0]: 0.090909
PageRank [1]: 0.545454
PageRank [2]: 0.272728
PageRank [3]: 0.090909
```


2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

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

Ikerprímeknek azokat a prímszámokat (csak önmagával és 1-gyel osztható) nevezzük, melyek különbsége 2.

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...

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet grájával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása:

```
#include <stdio.h>

int main(void)
{
    printf
        ("Adj meg egy számot decimális formában: ");

    unsigned int in = 0;
    scanf("%d", &in);

    for (int i = 0; i < in; ++i)
        printf((i % 5) ? "|" : " |");

    printf("\n");

    return 0;
}
```

Unáris számrendszer olyan számrendszer, ahol a számokat vonalakkal ('|') jelöljük, tehát például a 3-at unárisan III-nak írjuk fel. Ez a program mindössze annyit csinál, hogy bekér egy számot decimális alakban és átalakítja őket unáris számrendszerbeli számokká. A vonalakat a program ötös csoportokba rendezi.

A kódon túl sok magyaráznivaló nincs. A lényegi része az alábbi for ciklus:

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

```
printf((i % 5) ? "|" : " ");
```

Láthatjuk, hogy a ciklus 0-tól, in(bekért szám)-ig ismétlődik. Minden ismétlődéskor rajzol egy 'l' szimbólumot. Ha a szám 5-tel osztva maradékosan egyenlő 0-val akkor a következő 'l' szimbólum előtt ír egy szóközt, így csoportosítja 5-ös csoportkora, a kimenet így nem ömlesztett, tisztán olvasható.

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

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

Megoldás videó:

Megoldás forrása:

I. Legyenek S, X, Y változók. Legyen a, b, c konstansok.

$S \rightarrow abc, S \rightarrow aXbc, Xb \rightarrow bX, Xc \rightarrow Ybcc, bY \rightarrow Yb, Ay \rightarrow aax, Ay \rightarrow aa$

```
S
aXbc
abXc (Xc -> Ybcc)
abYbcc
```

- $S (S \rightarrow aXbc)$
- $aXbc (Xb \rightarrow bX)$
- $abXc (Xc \rightarrow Ybcc)$
- $abYbcc (bY \rightarrow Yb)$
- $aYbcc (aY \rightarrow aa)$
- $aabbcc$

II. Legyenek S, X, Y változók. Legyen a, b, c konstansok.

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

- $A (A \rightarrow aAB)$
- $aAB (A \rightarrow aC)$
- $aaCB (CB \rightarrow bCc)$
- $aaabCc (C \rightarrow bc)$
- $aabbcc$

A Chomsky-féle nyelvosztályok kitalálója az 50-es években tevékenykedő nyelvész, Noam Chomsky. A fenti példa bizonyítja, hogy $a^n b^n c^n$ nem környezetfüggetlen, hiszen kétféleképpen is meg tudtuk adni.

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása:

```
#include <stdio.h>

int main()
{
    for(int i = 0; i < 100; i++)
    {
        printf("IDK\n");
    }
    return 0;
}
```

A fenti kód C99-es fordítóval hibátlanul lefordul és működik, C89-cel azonban már nem. A hiba a for ciklusban deklarált változóval van, ami csak a későbbi (C99) verzióval kompatibilis. Ahhoz, hogy C89-cel is működjön, a számlálót a ciklus előtt kell deklarálni.

3.4. 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 használjunk, azaz óriások vállán álljunk és ne kispályázzunk!

Megoldás videó:

Megoldás forrása:

```
#include <stdio.h>
%}

%option noyywrap

%%
[+-]?([0-9]*\.[0-9]+|[0-9]+) {
    printf("Valos szam: %s\n", yytext);
}
.\n {}
%%
```

```
int main(void)
{
    yylex();
    return 0;
}
```

A fenti kód egy lexikális elemző, ami megszámolja a bemenetre érkező valós számokat. Ezt "óriások vállán állva", lexert használva érjük el - csak megadjuk a definíciót és hagyjuk, hogy a program tegye a dolgát. $[-+]?([0-9]*\.[0-9]+|[0-9]+)$ jelentése, hogy bármely szám nullától kilencig, bárhányszoros előfordulása érvényes. A következő sorban az `.\n {}` utasítással minden más bemenetet ignorálunk.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó:

Megoldás forrása:

```
...
%%
"a"  { printf("4");      }
"c"  { printf("k");      }
"e"  { printf("3");      }
"o"  { printf("0");      }
"i"  { printf("1");      }
"t"  { printf("7");      }
"q"  { printf("kw");     }
"A"  { printf("/-\\");   }
"B"  { printf("13");     }
"C"  { printf("K");      }
"E"  { printf("3");      }
"I"  { printf("1");      }
"V"  { printf("\\\\/");   }
"bye"      { printf("bai");      }
"and"      { printf("nd");       }
"dude"     { printf("d00d");     }
[...]
.|\n { printf("%s", yytext); }
%%
...
```

Óriások vállán állni sokkal egyszerűbb, mint magunknak megírni mindent. Ez a kód kiválóan szemlélteti ezt. Egyszerű szabályokkal megadjuk, hogy az adott bemenetre mit adjon vissza a program, minden másat pedig írjon le változatlanul. Ha ezt önállóan szeretnénk megoldani, akkor magunknak kellene olvasni a bemenetet és keresni a felcserélendő karakterláncokat.

A %% jelek között definiáljuk a szabályokat, hogy adott bemenetre milyen kimenetet adjon. Ha "nem állnánk óriások vállán", akkor ezt rengeteg elágazással tehetnénk csak meg. Bal oldalon " " jelek között találjuk a várt bemenetet, amit ezután { printf("*a_kívánt_kimenet* "); } a standard kimenetre kiíró utasításban található " " közötti karakterre vagy karakterláncra cserélünk.

Kimenet:

```
hello, world!
h3ll0, w0rld!
what's up?
wut's up?
this is so cool!
thls ls s0 kewl!
what's up mate?
wut's up m8?
hey dude!
h3y d00d!
loveu
10/3u
Using a lexer is fun!
Us1ng 4 l3x3r ls fvn!
```

3.6. A források olvasása

Amennyiben nem jól értelmeztem a feladatot és a megoldás helytelen, akkor elhasználnám rá 1 passzolási lehetőségemet, viszont, ha jó, akkor kiváltanék vele egy mulasztást (kiszabott határidőben nem elkészült feladatot).

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

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

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



Bugok

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

i.

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

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);
```

Megoldás forrása:

Megoldás videó:

I.

A SIGINT jelkezelő figyelembe van véve, tehát a jelkezelő függvény végzi a jelkezelést. A jelkezelő függvény egy void típusú függvény, amiben egy egyszerű kiiratás van (`printf("")`). A programot CTR+C paranccsal hiába próbáljuk megszakítani, nem működik. A leállításhoz egy új terminált kell indítani, abban megkeresni a program PID-jét (Process ID) és a `sudo kill -s PID` paranccsal leállítani.

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

void jelkezelolo()
{
    printf("____SZÖVEG____\n");
}

int main()
{
    for(;;)
    {
        if(signal(SIGINT, jelkezelolo)==SIG_IGN)
            signal (SIGINT, SIG_IGN);
    }
    return 0;
}
```

II.

Elvégzünk néhány változtatást, feltöltjük a ciklusmagot, így az már nem végtelen lesz. A ciklusváltozó értékét prefix növeljük, így az általa utolsónak felvett érték a 4 lesz.

```
#include <stdio.h>
#include <signal.h>
void jelkezeselo()
{
    printf("____SZÖVEG____\n");
}

int main()
{
    int i;
    for(i=0; i<5; ++i)
    {
        if(signal(SIGINT, jelkezeselo)==SIG_IGN)
            signal (SIGINT, SIG_IGN);
    }

    printf("\n");

    return 0;
}
```

III.

Az előző kódhoz képest a változás csupán a ciklusmagban van, a változó értékét postfix növeljük, így az általa utolsónak felvett érték 5 lesz.

```
#include <stdio.h>
#include <signal.h>
void jelkezeselo()
{
    printf("____SZÖVEG____\n");
}

int main()
{
    int i;
    for(i=0; i<5; i++)
    {
        if(signal(SIGINT, jelkezeselo)==SIG_IGN)
            signal (SIGINT, SIG_IGN);
    }

    printf("\n");

    return 0;
}
```

IV.

Itt a tomb tömb első 5 elemét 1-gyel növeljük, azonban a `tomb[i]=i++` hibás lehet, más gépeken vagy más fordítókkal futtatva más értékeket adhat vissza, mivel rossz a végrehajtási sorrendje.


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

void jelkezeselo()
{
    printf("____SZÖVEG____\n");
}

int main()
{
    for(i=0; i<5; tomb[i] = i++)
    {
        if(signal(SIGINT, jelkezeselo)==SIG_IGN)
            signal (SIGINT, SIG_IGN);
    }
    return 0;
}
```

V.

Ez a ciklus 0-tól n-ig tart. n-re az 's' pointer mutat, aminek az értéke a 'd' pointerre mutat. A pointereket 1-gyel léptetjük és minden iteráció végén növeljük 'i' értékét 1-gyel

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

void jelkezeselo()
{
    printf("____SZÖVEG____\n");
}

int main()
{
    for(i=0; i<n && (*d++ = *s++); ++i)
    {
        if(signal(SIGINT, jelkezeselo)==SIG_IGN)
            signal (SIGINT, SIG_IGN);
    }
    return 0;
}
```

VI.

A program hibája, hogy a kiértékelési sorrendet előre határozza meg. Kiíratunk 2 egészet, aminek $f(a, ++a)$, $f(++a, a)$ a meghatározója.

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

VII.

`f` és `'a'` változó által kiírt egészek.

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

VIII.

A kiértékelési sorrend ismét felborul, mivel az `f` közvetlenül tudja változtatni az `'a'` változó értékét.

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

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

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

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

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

A LaTeX egy szövegformáló szoftver, amit matematikusok előszeretettel használnak. A fent látható sorokat beillesztjük egy szöveges állományba, majd segédprogram segítségével az alábbi módon futtatjuk.

```
$ pandoc -t latex logikus.tex -o logikus.pdf
```

Ez létrehoz kimenetként egy képet, amit a fent leírtak szerint formáz meg.

Az Ar nyelv egy elsőrendű logikai nyelv.

A fent leírtak a következőket jelentik:

- a prímszámok száma végtelen;
- az ikerprímek száma végtelen;
- a prímszámok száma véges;
- a prímszámok száma véges.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

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

Mit vezetnek be a programba a következő nevek?

- `int a;`
- `int *b = &a;`
- `int &r = a;`
- `int c[5];`
- `int (&tr)[5] = c;`
- `int *d[5];`
- `int *h ();`
- `int *(*l) ();`
- `int (*v (int c)) (int a, int b)`

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

Megoldás videó:

Megoldás forrása:

```
#include "stdio.h"

int main()
{
 int a; // egész
 int *b = &a; // egészre mutató mutató
 int &r = a; // egész referenciája
 int c[5]; // egészek tömbje
 int (&tr)[5] = c; // egészek tömbjének referenciája (nem az első elemé)
 int *d[5]; // egészre mutató mutatók tömbje
 int *h (); // egészre mutató mutatót visszaadó függvény
 int *(*l) (); // egészre mutató mutatót visszaadó függvényre mutató mutató
 int (*v (int c)) (int a, int b); // egészet visszaadó és két egészet kapó ←
 függvényre mutató mutatót visszaadó, egészet kapó függvény
 int ((*z) (int)) (int, int); // függvényt mutató egy egészet visszaadó és ←
 két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó ←
 függvényre

 return 0;
}
```

Tanulságok, tapasztalatok, magyarázat...

## 4. fejezet

# Helló, Caesar!

### 4.1. double \*\*\* háromszögmátrix

Megoldás videó:

Megoldás forrása:

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

int
main ()
{
 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;
 }
 }
}
```

```
printf("%p\n", tm[0]);

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

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

tm[3][0] = 42.0;
(*(tm + 3))[1] = 43.0; // mi van, ha itt hiányzik a külső ()
*(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);

return 0;
}
```

Ez a feladat a C egyik funkciójára világít rá, ami a dinamikus memóriakezelés. Létrehozunk egy háromszögmátrixot `matrix **`, amit a fordítónak hála más módon is leírhatunk, így nem kell foglalkozni a mutató- és címaritmetikával foglalkozni.

A kód elején megadjuk a kétdimenziós tömb méretét, majd lefoglaljuk a memóriában a helyet a háromszögmátrixhoz (`matrix`), majd lefoglalunk a sorok számával megegyező számú tömböt, amik szintén tartalmazni fognak az oszlopok számával megegyező számú tömböt.

<https://github.com/mesterakos963/Prog1/blob/master/doublecscs.png>

Kimenetként a következőt kapjuk:

```
$
Mutato cime: 0x7ffffbc435de0
Sorok tombjenek cime: 0x5567bca69670
Elso sor cime: 0x5567bca696a0
|1.00 0.00 0.00 0.00 0.00 |
|2.00 1.00 0.00 0.00 0.00 |
```

|      |      |      |      |      |  |
|------|------|------|------|------|--|
| 3.00 | 1.00 | 1.00 | 0.00 | 0.00 |  |
| 4.00 | 2.00 | 1.00 | 1.00 | 0.00 |  |
| 5.00 | 2.00 | 1.00 | 1.00 | 1.00 |  |

Láthatjuk, hogy a kapott mátrixunk felső háromszög alakú, ami azt jelenti, hogy a főátló fölötti értékek mindegyike 0-val egyenlő.

## 4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása:

```
#define MAX_KEY_SIZE 100
#define BUFFER_SIZE 256

char key[MAX_KEY_SIZE];
char buffer[BUFFER_SIZE];

int key_index = 0, read_bytes = 0;
int key_size = strlen(argv[1]);

strncpy(key, argv[1], MAX_KEY_SIZE);

while ((read_bytes = read(0, (void *)buffer, BUFFER_SIZE))) {
 for (int i = 0; i < read_bytes; i++) {
 buffer[i] ^= key[key_index];
 key_index = (key_index + 1) % key_size;
 }

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

Ez egy elég egyszerű titkosító program. A bemeneten érkező szövegen karakterenként végi el a XOR utasítást, majd a kapott titkosított szöveget kiírja fájlba vagy a standard kimenetre, de a működéshez szükség van a kulcsra.

A kód elején definiáljuk a kulcs maximum hosszát, valamint a buffer méretét, aztán ezeket az értékeket használjuk fel a key valamint a buffer karakterláncok méretének megadásához.

A két egymásba ágyazott ciklus hivatott arra, hogy a bemenetet addig olvassuk, ameddig az tart, valamint rögtön törjük  $\wedge$ (XOR) operátorral.

Mivel a bemenetet a key tömb `key_index`-edik eleme szerint törjük, csak a megfelelő kulcs segítségével kapunk jó kimenetet, egyébként a kapott kódolt szöveg dekódolhatatlan lesz.

## 4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása:

```
import java.util.*;

class XorEncode {
 public static void main(String[] args) {
 String kulcs = "";

 if(args.length > 0) {
 kulcs = args[0];
 } else {
 System.out.println("Kulcs nélkül nem titkosítok!");
 System.out.println("Hasznalat: java XorEncode.java [kulcs]");
 System.exit(-1);
 }

 Scanner sc = new Scanner(System.in);
 String str = "";

 while(sc.hasNext()) {
 str = sc.next();
 System.out.println(xor(kulcs, str));
 }

 public static String xor(String kulcs, String s) {
 StringBuilder sb = new StringBuilder();

 for(int i = 0; i < s.length(); i++) {
 sb.append((char) (s.charAt(i) ^ kulcs.charAt(i % kulcs.length())));
 }

 return sb.toString();
 }
 }
}
```

A program úgygy működik, mint a C-s változata. Az argumentumok közül kiolvassa a kulcsot majd a bemeneten érkező szövegen végrehajtjuk a XOR utasítást, a kimenetet pedig egy szöveges fájlba tesszük. A Java a C-vel szemben egy objektum orientált programozási nyelv, nincs benne memóriakezelés, ami miatt jóval egyszerűbb, viszont szintaxisban vannak hasonlóságok a C-vel.



## 4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása:

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

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

double atlagos_szohossz(const char *titkos, int titkos_meret);
int tiszta_lehet(const char *titkos, int titkos_meret);
void exor(const char kulcs[], int kulcs_meret, char titkos[],
 int titkos_meret, char *buffer);
void exor_tores(const char kulcs[], int kulcs_meret, char titkos[],
 int titkos_meret);

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

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

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

 int ii, ki, ji, li, mi, ni, oi, pi;

#pragma omp parallel for private(kulcs, ii, ki, ji, li, mi, ni, oi, pi) ←
 shared(p, titkos)
 // osszes kulcs eloallitasa
 for (ii = '\\0'; ii <= '9'; ++ii)
```

```
for (ji = '0'; ji <= '9'; ++ji)
 for (ki = '0'; ki <= '9'; ++ki)
 for (li = '0'; li <= '9'; ++li)
 for (mi = '0'; mi <= '9';
 ++mi)
 for (ni = '0';
 ni <= '9'; ++ni)
 for (oi = '0';
 oi <=
 '9';
 ++oi)
 for (pi = '0'; pi <= '9'; ++pi) {
 if ((kulcs = (char *)malloc(sizeof(char) * KULCS_MERET)) ←↔
 == NULL) {
 printf
 ("Memoria (kulcs) falióra\n");
 exit(-1);
 }

 kulcs
 [0]
 =
 ii;
 kulcs
 [1]
 =
 ji;
 kulcs
 [2]
 =
 ki;
 kulcs
 [3]
 =
 li;
 kulcs
 [4]
 =
 mi;
 kulcs
 [5]
 =
 ni;
 kulcs
 [6]
 =
 oi;
 kulcs
 [7]
 =
```

```
 pi;

 exor_tores
 (kulcs,
 KULCS_MERET,
 titkos,
 p
 -
 titkos);
 }

 return 0;
}

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;
}

int tiszta_lehet(const char *titkos, int titkos_meret)
{
 // a tiszta szoveg valszeg tartalmazza a gyakori magyar szavakat
 // illetve az átlagos szóhossz vizsgálatával csökkentjük a
 // potenciális töréseket

 double szohossz = atlagos_szohossz(titkos, titkos_meret);

 return szohossz > 3.0 && szohossz < 9.0
 && strcasestr(titkos, "hogy") && strcasestr(titkos, "nem")
 && strcasestr(titkos, "ne")
 && strcasestr(titkos, "az") && strcasestr(titkos, "ha");
}

void exor(const char kulcs[], int kulcs_meret, char titkos[],
 int titkos_meret, char *buffer)
{
 int kulcs_index = 0;
 for (int i = 0; i < titkos_meret; ++i) {
 buffer[i] = titkos[i] ^ kulcs[kulcs_index];
 kulcs_index = (kulcs_index + 1) % kulcs_meret;
 }
}
```

```
void exor_tores(const char kulcs[], int kulcs_meret, char titkos[],
 int titkos_meret)
{
 char *buffer;

 if ((buffer =
 (char *)malloc(sizeof(char) * titkos_meret)) == NULL) {
 printf("Memoria (buffer) falióra\n");
 exit(-1);
 }

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

 if (tisztalehet(buffer, titkos_meret)) {
 printf
 ("Kulcs: [%c%c%c%c%c%c%c%c]\nTiszta szöveg: [%s]\n",
 kulcs[0], kulcs[1], kulcs[2], kulcs[3], kulcs[4],
 kulcs[5], kulcs[6], kulcs[7], buffer);
 }

 free(buffer);
}
```

Ez a program egy bruteforce algoritmus segítségével fejt vissza a 4.2-es feladat után kapott kódolt szöveget. Mivel ez egy hosszú folyamat, ezért a minél gyorsabb futás érdekében használjuk az OpenMP-t. A bemenetről olvasott szavak hosszának vizsgálása után a keresi a leggyakoribb magyar szavakat és kiírja őket kimenetre.

A bemenetet olvasva a szavak hosszának vizsgálata után keresi a leggyakoribb magyar szavakat, amennyiben a szó hossza megfelel egy bizonyos értékhatárnak. Ha talál ilyen szavakat, azt kiírja kimenetre. (meg, van, vagy, és, volt, már, hogy, stb).

## 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)

Ebben a feladatban egy neurális hálót építünk fel, ami úgy működik, hogy megadjuk neki, hogy milyen bemenetre milyen kimenetet adjon vissza majd ő ezt megpróbálja mesterségesen utánozni. A program futtatásakor a kimenetről a kimenetet vizualizálva kapjuk meg. A programnak 149 lépés alatt sikerült megtanulnia a neurális hálónkat az OR utasításra. A matematikai számításokat a neuralnet nevű könyvtár végezte.

## 4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

Megoldás forrása:

```
#include "ml.hpp"

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

int main(int argc, char **argv) {
 png::image<png::rgb_pixel> image(argv[1]);

 int size = image.get_width() * image.get_height();

 Perceptron *p = new Perceptron(3, size, 256, 1);

 double *image_d = new double[size];

 for(int i = 0; i < image.get_width(); i++)
 for(int j = 0; j < image.get_height(); j++)
 image_d[i*image.get_width() + j] = image[i][j].red;

 double value = (*p)(image_d);

 std::cout << value << std::endl;

 delete p;
 delete[] image_d;

 return 0;
}
```

A program kódja viszonylag egyszerű, nem tartalmaz semmiféle trükköt. A mandelbrot feladat által generált png-t vizsgálja pixelről pixelre és számolja a piros színű pixeleket, kimenetként pedig megkapjuk, hogy az adott képnek hány százalékát alkotják piros pixelek.

## 5. fejezet

# Helló, Mandelbrot!

### 5.1. A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása:

Program forráskódja:

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

#define N 500
#define M 500
#define MAXX 0.7
#define MINX -2.0
#define MAXY 1.35
#define MINY -1.35

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[x][y] = png::rgb_pixel(tomb[x][y], tomb[x][y], tomb[x][y] ←
);
 }
 }
 image.write("kimenet.png");
}

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] = 256 - iteracio;
 }
 }

 GeneratePNG(tomb);

 return 0;
}
```

### Make file

```
all: mandelbrot clean

mandelbrot.o: mandelbrot.cpp
 @g++ -c mandelbrot.cpp `libpng-config --cflags`
```

```
mandelbrot: mandelbrot.o
@g++ -o mandelbrot mandelbrot.o `libpng-config --ldflags`

clean:
@rm -rf *.o
@./mandelbrot
@rm -rf mandelbrot
```

A Mandelbrot halmaz egy olyan komplex halmaz, amely illeszkedik a  $f_c(z) = z^2 + c$  függvény képére, s nullától iterálva nem divergál, tehát a  $f_c(0), f_c(f_c(0)), \dots$  abszolútértékben korlátos. A program alapjául a png++ könyvtár szolgál. Ebben a feladatban nem használjuk az `std::complex` osztályt, hanem magunknak hozunk létre egy struktúrát, ami tartalmazza az imaginárius és képzetes egységet, amiket `double` változóban tárolunk. Ezután megadunk egy halmazt, amire az egyenletünk illeszkedik és az ennek megfelelő pixeleket kiszínezzük.

GeneratePNG függvény felelős a kimeneti png létrehozásáért.

A kimenet ezen a linken található: <https://github.com/mesterakos963/Prog1/blob/master/mandelbrot.png>

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

```
#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");
}

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;
}
```

A program az előző feladatnak egy elegánsabb megoldása. Itt használjuk az `std::complex` osztályt, ami ad némi előnyt, de a végeredmény ugyanaz a kimenet lesz. Működési elve szintén annyi, hogy a Mandelbrot halmaznak megfelelő pixeleket kiszínezi.

Az `std::complex` osztállyal könnyedén reprezentálhatunk számokat a komplex számok halmazáról. Az osztályon belül sok előre deklarált változónak adhatunk értéket. `PL`: `real` - valós rész, `imag` - imaginárius rész, `abs` - komplex szám abszolút értéke, `conj` komplex szám konjugáltja. Az osztály használata a végeredményen nem változtat, csupán egy eszköz, ami megkönnyíti a programírást.

## 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 szoros kapcsolatban állnak a Mandelbrot halmazzal. Itt is létrehozunk egy make file-t és futtatjuk a programot, ami létrehoz egy bmorf.png képet az előzőhöz hasonló módon, de mivel az egyenletünk más ebből egyértelműen következik, hogy a kapott ábra is másog fog kinézni. A formán kívül azonban az is szembeötlő, hogy ez az ábra már színes. Ezt úgy kapjuk meg, hogy az iteráció számát elosztjuk 255-tel, így RGB színezést kapunk.

Kimenet: <https://github.com/mesterakos963/Prog1/blob/master/bmorf.png>

## 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:

```
// frakablak.cpp
//
// Mandelbrot halmaz nagyító

#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();
}

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

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);
 }
 qpainter.end();
}

void FrakAblak::mousePressEvent(QMouseEvent* event) {

 // A nagyítandó kijelölt területet bal felső sarka:
 x = event->x();
 y = event->y();
 mx = 0;
 my = 0;

 update();
}

void FrakAblak::mouseMoveEvent(QMouseEvent* event) {

 // A nagyítandó kijelölt terület szélessége és magassága:
 mx = event->x() - x;
 my = mx; // négyzet alakú

 update();
}

void FrakAblak::mouseReleaseEvent(QMouseEvent* event) {

 if(szamitasFut)
 return;

 szamitasFut = true;

 double dx = (b-a)/szelesseg;
```

```
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, szelesseg, magassag, ←
 iteraciosHatar, this);
mandelbrot->start();

update();
}

void FrakAblak::keyPressEvent(QKeyEvent *event)
{
 if(szamitasFut)
 return;

 if (event->key() == Qt::Key_N)
 iteraciosHatar *= 2;
 szamitasFut = true;

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

void FrakAblak::vissza(int magassag, int *sor, int meret)
{
 for(int i=0; i<meret; ++i) {
 QRgb szin = qRgb(0, 255-sor[i], 0);
 fraktal->setPixel(i, magassag, szin);
 }
 update();
}

void FrakAblak::vissza(void)
{
 szamitasFut = false;
```

```
x = y = mx = my = 0;
}
```

Ehhez a feladathoz a QT Creator nevű programot használjuk, mely egy multiplatform GUI építő szoftver.

A már meglévő Mandelbrot-halmaz kódhoz építünk felhasználói felületet. Használatához kijelöljük az újragenerálandó területet, ezzel érjük el a nagyítást.

A szükséges kiegészítő kódok itt találhatóak: <https://sourceforge.net/p/udprog/code/ci/master/tree/source-labor/Qt/Frak/>

## 5.6. Mandelbrot nagyító és utazó Java nyelven

Ehhez a feladathoz az OpenJFX 11 nevű szoftvert használni, ami a QT-hoz hasonló GUI építő szoftver.

A Mandelbrot nagyító és utazó JAVA nyelvű megvalósítása itt található: <https://sourceforge.net/p/udprog/code/ci/master/tree/source-labor/Qt/Frak/>

## 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:

A kód JAVA nyelven

```
#include <stdio.h>
public class PolarGenerator {
 boolean nincsTarolt = true;
 double tarolt;
 public PolarGenerator() {

 nincsTarolt = true;

 }
 public double kovetkezo() {
 if(nincsTarolt) {
 double u1, u2, v1, v2, w;
 do {
 u1 = Math.random();
 u2 = Math.random();

 v1 = 2*u1 - 1;
 v2 = 2*u2 - 1;

 w = v1*v1 + v2*v2;

 } while(w > 1);
```

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

 tarolt = r*v2;
 nincsTarolt = !nincsTarolt;

 return r*v1;

 } else {
 nincsTarolt = !nincsTarolt;
 return tarolt;
 }
}

public static void main(String[] args) {
 PolarGenerator g = new PolarGenerator();
 for(int i=0; i<10; ++i)
 System.out.println(g.kovetkezo());
}
```

### A kód C++-ban

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <ctime>
class PolarGen
{
public:
 PolarGen ()
 {
 nincsTarolt = true;
 std::srand (std::time (NULL));
 }
 ~PolarGen ()
 {
 }
 double kovetkezo ()
 {
 if (nincsTarolt)
 {
 double u1, u2, v1, v2, w;
 do
 {
 u1 = std::rand () / (RAND_MAX + 1.0);
 u2 = std::rand () / (RAND_MAX + 1.0);
 v1 = 2 * u1 - 1;
 v2 = 2 * u2 - 1;
 w = v1 * v1 + v2 * v2;
 }
```

```
 }
 while (w > 1);
 double r = std::sqrt ((-2 * std::log (w)) / w);
 tarolt = r * v2;
 nincsTarolt = !nincsTarolt;
 return r * v1;
}
else
{
 nincsTarolt = !nincsTarolt;
 return tarolt;
}
}
private:
 bool nincsTarolt;
 double tarolt;
};
int
main (int argc, char **argv)
{
 PolarGen pg;
 for (int i = 0; i < 10; ++i)
 std::cout << pg.kovetkezo () << std::endl;
 return 0;
}
```

Kezdetben deklarálunk a Polargen osztályt. A random szám generátort beállítjuk, hogy a legenerált számok ne legyenek nagyobbak 100-nál, a konstruktorban pedig a nincsTarolt boolean típusú változóval megadjuk, hogy még nincs eltárolt szám. A kovetkezo függvény vizsgálja meg, hogy van-e már eltárolt szám a konstruktorban, amennyiben ez hamis, legenerál kettőt, ebből ez egyiket eltárolja, és a nincsTarolt változót hamisra állítja, a másikkal pedig visszalép. Páratlanadik számú meghívás esetén az előző lépés másik számát adjuk vissza. Azt, hogy éppen milyen lépésnél járunk (páros/páratlan), a nincsTarolt logikai függvénnyel tartjuk számon.

A program futása után láthatjuk, hogy a program 10 véletlenszerű számot ad vissza normalizálva. A matematikai háttér el van rejtve, de ez számunkra teljesen lényegtelen.

```
$ /java/bin/java polargen.java
-0.7353431820414118
-0.33784190028284766
0.7750031835316805
0.5524713543467192
-0.5380423283211784
1.512849268596637
2.7148874695500966
-0.23688836801277952
-0.3238588036816322
-0.7963150809415576
$ /java/bin/java polargen.java
-0.6566325405553158
```



```
0.40465899229436114
0.08634239512228409
-0.9470321445590416
0.1926238606249351
0.7705517022243931
0.9084531239664848
-1.4472688950554047
-1.6250659297425345
-0.7791586500972545
```

## 6.2. LZW

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

Megoldás videó:

Megoldás forrása: <https://github.com/mesterakos963/Prog1/blob/master/binfa.c>

A LZWBInfa algoritmus a megadott bemenetifájlt alakítja át bináris formára, majd ebből egy fát épít. Ez egy igencsak elterjedt tömörítési eljárás. A működési elve a következő:

- ha 0-át kell beírni a fába, akkor megvizsgáljuk, hogy a csomópont tartalmaz-e 0 elemet \* ha igen, akkor készítünk egy új csomópontot és az ő gyermekeként illesztjük be a nullát; \* ha nem akkor betesszük a 0 gyermekének a nullát;
- 1-es esetén a lépések megegyeznek;

Használata a következőképpen néz ki, de hibás futtatás esetén ezt a programunk is kiírja: `./(fordított program neve) -bemenetifájl.kiterjesztés -o -kimenetifájl.kiterjesztés`

A programot úgy kezdjük, mint bármelyik másikat, csatoljuk a használni kívánt függvénykönyvtárakat. Ezután létrehozuk a `binfa` struktúrát. A struktúra tagjait a `bal_nulla` és `jobb_egy` mutatók alkotják. Ez követően jön létre az `uj_elem`, ami hibával tér vissza, ha nem foglal le elegendő helyet a binfánk számára. Itt deklaráljuk a függvényeket, amiket majd a `main` után definiálunk. A beolvasott bite-ok a `b` változóba kerülnek, majd létrejön `gyoker`, mint új elem, aminek rögtön értéket is adunk. Az építés rész, a fentebb említett brute force algoritmus alapján ezután következik. Ha a fenti szabályok értelmében új elemet kell létrehozni, akkor az `uj_elem` függvény hívódik meg és a létrehozott elemnek rögtön értéket is adunk, szabálynak megfelelően 0-át vagy 1-et. Ezt követően matematikai számítások történnek, a program itt számolja ki a fa szórását, átlagát és írja ki ezeket.

Mivel C nyelven írt programról van szó, nem hagyhatjuk ki a memóriakezelést. Ehhez tulajdonképpen 2 parancsot kell megfelelően használni: `malloc` - lefoglalja a helyet a binári fánknak és hogy elkerüljük a memóriaszivárgást, használnunk kell a `free` parancsot. Ezt a szabadit függvényben tesszük meg, itt szabadítjuk fel az `elem` változót.

## 6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása:

Inorder bejárás

```
if (bejar == 0) {
 if (elem != NULL) {
 ++melyseg;
 kiir(elem->nullasGyermek(), os);
 for (int i = 0; i < melyseg; ++i)
 os << "---";
 os << elem->getBetu() << "(" << melyseg << ")" << std::endl;
 kiir(elem->egyenesGyermek(), os);
 --melyseg;
 }
}
```

Postorder bejárás

```
else if (bejar == 1) {
 if (elem != NULL) {
 ++melyseg;
 kiir(elem->egyenesGyermek(), os);
 for (int i = 0; i < melyseg; ++i)
 os << "---";
 os << elem->getBetu() << "(" << melyseg << ")" << std::endl;
 kiir(elem->nullasGyermek(), os);
 --melyseg;
 }
}
```

A z3a7.cpp alpból inorder járja be a bináris fákat, ami azt jelenti, hogy először a bal oldali részt, aztán a gyökeret, majd a jobb oldali részt vizsgálja meg. A preorder bejárás során a gyökérrel kezdünk, majd a bal és a jobb oldali részt járjuk be. Postorder bejárásnál a bal-jobb-gyökér sorrendet követi a program. Amint látható a különböző bejárások során csak a gyökér bejárását toljuk jobbra vagy balra, a bal-jobb sorrend mindig megmarad.

Ahhoz, hogy a bejárás módját mi adjuk meg növelni kell az argumentumok számát, majd a futtatáskor megadott argumentumok alapján bejárni a fát. (1 = Inorder; 0 = Postorder).

A futtatás már a következő képen néz ki: `./binfa bemeneti_file -o kimeneti_file [0/1]`

## 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:

```
// z3a7.cpp

#include <iostream> // mert olvassuk a std::cin, írjuk a std::cout ←
 csatornákat
#include <cmath> // mert vonunk gyököt a szóráshoz: std::sqrt
#include <fstream> // fájlból olvasunk, írunk majd

class LZWBinFa
{
public:
 LZWBinFa () : fa (&gyoker)
 {
 }
 ~LZWBinFa ()
 {
 szabadit (gyoker.egyenesGyermek ());
 szabadit (gyoker.nullasGyermek ());
 }

 void operator<< (char b)
 {
 if (b == '0')
 {
 if (!fa->nullasGyermek ())
 {
 Csomopont *uj = new Csomopont ('0');
 fa->ujNullasGyermek (uj);
 fa = &gyoker;
 }
 else
 {
 fa = fa->nullasGyermek ();
 }
 }
 // Mit kell betenni éppen, vagy '1'-et?
 else
 {
 if (!fa->egyenesGyermek ())
 {
 Csomopont *uj = new Csomopont ('1');
 fa->ujEgyenesGyermek (uj);
 fa = &gyoker;
 }
 else
 {
 fa = fa->egyenesGyermek ();
 }
 }
 }
};
```

```
 }
}

void kiir (void)
{
 melyseg = 0;
 kiir (&gyoker, std::cout);
}

int getMelyseg (void);
double getAtlag (void);
double getSzoras (void);

friend std::ostream & operator<< (std::ostream & os, LZWBinFa & bf)
{
 bf.kiir (os);
 return os;
}
void kiir (std::ostream & os)
{
 melyseg = 0;
 kiir (&gyoker, os);
}

private:
class Csomopont
{
public:
 Csomopont (char b = '/'):betu (b), balNulla (0), jobbEgy (0)
 {
 };
 ~Csomopont ()
 {
 };
 Csomopont *nullasGyermekek () const
 {
 return balNulla;
 }
 Csomopont *egyenesGyermekek () const
 {
 return jobbEgy;
 }
 void ujNullasGyermekek (Csomopont * gy)
 {
 balNulla = gy;
 }
 void ujEgyenesGyermekek (Csomopont * gy)
 {
 jobbEgy = gy;
 }
}
```

```
 }
 char getBetu () const
 {
 return betu;
 }

private:
 char betu;
 Csomopont *balNulla;
 Csomopont *jobbEgy;
 Csomopont (const Csomopont &); //másoló konstruktor
 Csomopont & operator= (const Csomopont &);
};

Csomopont *fa;
int melyseg, atlagosszeg, atlagdb;
double szorasosszeg;

//nocopy
LZWBInFa (const LZWBInFa &);
LZWBInFa & operator= (const LZWBInFa &);

void kiir (Csomopont * elem, std::ostream & os)
{
 if (elem != NULL)
 {
 ++melyseg;
 kiir (elem->egyenesGyermekek (), os);
 for (int i = 0; i < melyseg; ++i)
 os << "----";
 os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
 kiir (elem->nullasGyermekek (), os);
 --melyseg;
 }
}

void szabadit (Csomopont * elem)
{
 if (elem != NULL)
 {
 szabadit (elem->egyenesGyermekek ());
 szabadit (elem->nullasGyermekek ());
 delete elem;
 }
}

protected:
 Csomopont gyoker;
 int maxMelyseg;
 double atlag, szoras;
```

```
void rmelyseg (Csomopont * elem);
void ratlag (Csomopont * elem);
void rszoras (Csomopont * elem);

};

int
LZWBinFa::getMelyseg (void)
{
 melyseg = maxMelyseg = 0;
 rmelyseg (&gyoker);
 return maxMelyseg - 1;
}

double
LZWBinFa::getAtlag (void)
{
 melyseg = atlagosszeg = atlagdb = 0;
 ratlag (&gyoker);
 atlag = ((double) atlagosszeg) / atlagdb;
 return atlag;
}

double
LZWBinFa::getSzoras (void)
{
 atlag = getAtlag ();
 szorasosszeg = 0.0;
 melyseg = atlagdb = 0;

 rszoras (&gyoker);

 if (atlagdb - 1 > 0)
 szoras = std::sqrt (szorasosszeg / (atlagdb - 1));
 else
 szoras = std::sqrt (szorasosszeg);

 return szoras;
}

void
LZWBinFa::rmelyseg (Csomopont * elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 if (melyseg > maxMelyseg)
 maxMelyseg = melyseg;
 rmelyseg (elem->egyGyermek ());
 }
}
```

```
 melyseg (elem->nullasGyermek ());
 --melyseg;
 }
}

void
LZWBinFa::ratlag (Csomopont * elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 ratlag (elem->egyesGyermek ());
 ratlag (elem->nullasGyermek ());
 --melyseg;
 if (elem->egyesGyermek () == NULL && elem->nullasGyermek () == NULL ↔
)
 {
 ++atlagdb;
 atlagosszeg += melyseg;
 }
 }
}

void
LZWBinFa::rszoras (Csomopont * elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 rszoras (elem->egyesGyermek ());
 rszoras (elem->nullasGyermek ());
 --melyseg;
 if (elem->egyesGyermek () == NULL && elem->nullasGyermek () == NULL ↔
)
 {
 ++atlagdb;
 szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));
 }
 }
}

void
usage (void)
{
 std::cout << "Usage: lzwtree in_file -o out_file" << std::endl;
}

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

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

char *inFile = *++argv;

if ((*((++argv) + 1) != 'o')
{
 usage ();
 return -2;
}

std::fstream beFile (inFile, std::ios_base::in);

if (!beFile)
{
 std::cout << inFile << " nem letezik..." << std::endl;
 usage ();
 return -3;
}

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

unsigned char b;
LZWBinFa binFa;

while (beFile.read ((char *) &b, sizeof (unsigned char)))
 if (b == 0x0a)
 break;

bool kommentben = false;

while (beFile.read ((char *) &b, sizeof (unsigned char)))
{
 if (b == 0x3e)
 {
 // > karakter
 kommentben = true;
 continue;
 }

 if (b == 0x0a)
 {
 // újsor
 kommentben = false;
 continue;
 }

 if (kommentben)
```



```
 continue;

 if (b == 0x4e) // N betű
 continue;

 for (int i = 0; i < 8; ++i)
 {
 if (b & 0x80)
 binFa << '1';
 else
 binFa << '0';
 b <<= 1;
 }

}

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

kiFile.close ();
beFile.close ();

return 0;
}
```

Ehhez a feladathoz az alap `z3a7.cpp` programot használjuk, mely alpból úgy van megírva, hogy a tag a gyökér. Ez az imént használt C programunk továbbfejlesztett, C++-ra átírt változata. A két nyelv közötti legnagyobb eltérés, hogy míg a C egy eljárásorientált programozási nyelv, addig a C++ objektumorientált. Hogy ez mit is jelent?

Egyszerű struktúra helyett itt élünk a C++ nyújtotta lehetőségekkel és a bináris fánk kialakításához `LZWBinFa` osztályt használunk, aminek a konstruktor, valamint a destruktorként amiben a `szabadit` függvényünket hívjuk majd meg több ízben. Itt történik meg az imént említett függvény definiálása, `<<` operátor túlterhelése is, amivel egyszerűbben tudjuk kiírni a binfa adatait, valamint itt írjuk meg még meg a binfa építő algoritmust, ami csakúgy, mint a C-s verzió esetében itt is a bemenettől, illetve az eltárolt tagoktól függően épít majd bináris fát.

A kiírásról számszerint 2 függvény is gondoskodik, ezek neve változatlanul `kiir`, attól függetlenül, hogy a 2 függvény nem egyezik meg, egyikük jóval összetettebb.

Az `LZWBinfa` osztály privát részében található a beágyazott csomópont osztály, aminek a privát része tartalmazza a konstruktort, mellyel beállítjuk a gyökeret, destruktort, a binfa pillanatnyi állapotait kezelő, illetve a bemenő karaktereket olvasó függvényeket. A privát részben található a `balNulla` és `jobbEgy` mutatók.

A kódban megtaláljuk az ebben a verzióban még letiltásra kerülő másoló konstruktorunkat, a `szabadit` függvényt, melynek működése megegyezik a C-s verzióban használt `szabadit` függvénnyel. Valamint ez a program is számolja a fa szórását és átlagát, csakúgy, mint a C-s verzió.

A programban használt függvényeink bár már az LZWBinfa osztályban deklarálva lettek, definiálásukra csak a kód végén kerül sor.

## 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:

```
#include <iostream>
#include <cmath>
#include <fstream>

class LZWBinFa
{
public:
 LZWBinFa ():fa (gyoker = new Csomopont ('/'))
 {
 }

 ~LZWBinFa ()
 {
 szabadit (gyoker->egyesGyermeke());
 szabadit (gyoker->nullasGyermeke());
 delete gyoker;
 }

 void operator<< (char b)
 {
 if (b == '0')
 {
 if (!fa->nullasGyermeke())
 {
 Csomopont *uj = new Csomopont ('0');
 fa->ujNullasGyermeke (uj);
 fa = gyoker;
 }
 else
 {
 fa = fa->nullasGyermeke ();
 }
 }

 else
 {
 if (!fa->egyesGyermeke ())
 {

```

```
 Csomopont *uj = new Csomopont ('1');
 fa->ujEgyesGyermekek (uj);
 fa = gyoker;
 }
 else
 {
 fa = fa->egyesGyermekek ();
 }
}

void kiir (void)
{
 melyseg = 0;
 kiir (gyoker, std::cout);
}

int getMelyseg (void);
double getAtlag (void);
double getSzoras (void);

friend std::ostream & operator<< (std::ostream & os, LZWBinFa & bf)
{
 bf.kiir (os);
 return os;
}

void kiir (std::ostream & os)
{
 melyseg = 0;
 kiir (gyoker, os);
}

private:
class Csomopont
{
public:
 Csomopont (char b = '/') : betu (b), balNulla (0), jobbEgy (0)
 {
 };

 ~Csomopont ()
 {
 };

 Csomopont *nullasGyermekek () const
 {
 return balNulla;
 }

 Csomopont *egyesGyermekek () const
```

```
{
 return jobbEgy;
}
void ujNullasGyermekek (Csomopont * gy)
{
 balNulla = gy;
}
void ujEgyesGyermekek (Csomopont * gy)
{
 jobbEgy = gy;
}
char getBetu () const
{
 return betu;
}

private:
 char betu;
 Csomopont *balNulla;
 Csomopont *jobbEgy;
 Csomopont (const Csomopont &);
 Csomopont & operator= (const Csomopont &);
};

Csomopont *fa;
int melyseg, atlagosszeg, atlagdb;
double szorasosszeg;
LZWBinFa (const LZWBinFa &);
LZWBinFa & operator= (const LZWBinFa &);

void kiir (Csomopont * elem, std::ostream & os)
{
 if (elem != nullptr)
 {
 ++melyseg;
 kiir (elem->egyenesGyermekek (), os);
 for (int i = 0; i < melyseg; ++i)
 os << "----";
 os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
 kiir (elem->nullasGyermekek (), os);
 --melyseg;
 }
}

void szabadit (Csomopont * elem)
{
 if (elem != nullptr)
 {
 szabadit (elem->egyenesGyermekek ());
 szabadit (elem->nullasGyermekek ());
 }
}
```

```
 delete elem;
 }
}

protected:
 // A fában tagként benne van egy csomópont, ez erősen ki van tüntetve, ←
 // Ő a gyökér:
 Csomopont * gyoker = new Csomopont('/');
 int maxMelyseg;
 double atlag, szoras;

 void rmelyseg (Csomopont * elem);
 void ratlag (Csomopont * elem);
 void rszoras (Csomopont * elem);

};

int
LZWBinFa::getMelyseg (void)
{
 melyseg = maxMelyseg = 0;
 rmelyseg (gyoker);
 return maxMelyseg - 1;
}

double
LZWBinFa::getAtlag (void)
{
 melyseg = atlagosszeg = atlagdb = 0;
 ratlag (gyoker);
 atlag = ((double) atlagosszeg) / atlagdb;
 return atlag;
}

double
LZWBinFa::getSzoras (void)
{
 atlag = getAtlag ();
 szorasosszeg = 0.0;
 melyseg = atlagdb = 0;

 rszoras (gyoker);

 if (atlagdb - 1 > 0)
 szoras = std::sqrt (szorasosszeg / (atlagdb - 1));
 else
 szoras = std::sqrt (szorasosszeg);

 return szoras;
}
```

```
void
LZWBinFa::rmelyseg (Csomopont * elem)
{
 if (elem != nullptr)
 {
 ++melyseg;
 if (melyseg > maxMelyseg)
 maxMelyseg = melyseg;
 rmelyseg (elem->egyenesGyermekek ());
 rmelyseg (elem->nullasGyermekek ());
 --melyseg;
 }
}

void
LZWBinFa::ratlag (Csomopont * elem)
{
 if (elem != nullptr)
 {
 ++melyseg;
 ratlag (elem->egyenesGyermekek ());
 ratlag (elem->nullasGyermekek ());
 --melyseg;
 if (elem->egyenesGyermekek () == nullptr && elem->nullasGyermekek () == nullptr)
 {
 ++atlagdb;
 atlagosszeg += melyseg;
 }
 }
}

void
LZWBinFa::rszoras (Csomopont * elem)
{
 if (elem != nullptr)
 {
 ++melyseg;
 rszoras (elem->egyenesGyermekek ());
 rszoras (elem->nullasGyermekek ());
 --melyseg;
 if (elem->egyenesGyermekek () == nullptr && elem->nullasGyermekek () == nullptr)
 {
 ++atlagdb;
 szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));
 }
 }
}
```

```
void
usage (void)
{
 std::cout << "Usage: lzwtree in_file -o out_file" << std::endl;
}

int
main (int argc, char *argv[])
{
 if (argc != 4)
 {
 usage ();
 return -1;
 }

 char *inFile = *++argv;

 if ((*++argv + 1) != 'o')
 {
 usage ();
 return -2;
 }

 std::fstream beFile (inFile, std::ios_base::in);

 if (!beFile)
 {
 std::cout << inFile << " nem letezik..." << std::endl;
 usage ();
 return -3;
 }

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

 unsigned char b;
 LZWBinFa binFa;

 while (beFile.read ((char *) &b, sizeof (unsigned char)))
 if (b == 0x0a)
 break;

 bool kommentben = false;

 while (beFile.read ((char *) &b, sizeof (unsigned char)))
 {

 if (b == 0x3e)
 {
 // > karakter
 kommentben = true;
 }
 }
}
```

```
 continue;
 }

 if (b == 0x0a)
 {
 // újsor
 kommentben = false;
 continue;
 }

 if (kommentben)
 continue;

 if (b == 0x4e) // N betű
 continue;

 for (int i = 0; i < 8; ++i)
 {
 if (b & 0x80)
 binFa << '1';
 else
 binFa << '0';
 b <<= 1;
 }
}

kiFile << binFa;

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

kiFile.close ();
beFile.close ();

return 0;
}
```

Az eltérés az előző programhoz képest az, hogy a Binfánk gyökere mutató. Azaz a Csomopont gyoker helyett Csomopont gyoker\* szerepel, viszont ezután már hivatkozáskor át kell írunk a gyoker-et gyoker\*-ra, ezt követően pedig fel kell szabadítanunk a memóriát a ~LZWBinFa destruktorkal.

## 6.6. Mozgató szemantika

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



Megoldás videó:

Megoldás forrása: <https://github.com/mesterakos963/Prog1/blob/master/Mozgat%C3%B3>

A kód szinte teljes mértékben megegyezik az előzővel, a lényegi része a rekúzív másoló és mozgató függvények.

A lényegi rész itt található:

```
kiFile << binFa;

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

LZWBinFa binFa3 =std::move(binFa);

kiFile << "depth = " << binFa3.getMelyseg () << std::endl;
kiFile << "mean = " << binFa3.getAtlag () << std::endl;
kiFile << "var = " << binFa3.getSzoras () << std::endl;

kiFile.close ();
beFile.close ();
```

A mozgató konstruktorban nullpointerre állítjuk annak a binfának a gyökerét amibe mozgatni szeretnénk a kiindulási binfánkat. Ezután használjuk a már előző feladatban is definiált, azonban ott még nem használt = operátort. (hiszen pont ennek az operátornak a túlterhelése történik lentebb) Meghívjuk move függvényt ami egy neki átadott bal értékből jobb értéket csinál.

Itt található a mozgató értékadás, melyre a mozgató konstruktor megvalósítása épül. Túlterheljükkerül az egyenlőségjel operátor amit a konstruktorban fel is tudunk használni. Az std függvénykönyvtár swap függvényét felhasználva felcseréljük a kiindulási és az új binfa gyökerét. A \*this egy úgynevezett visszahivatkozási mutató, a sima this-szel ellentétben nem egy mutatót, hanem egy "klónt" ad vissza az objektumról.

```
kiFile << binFa;

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

LZWBinFa binFa3 =std::move(binFa);

kiFile << "depth = " << binFa3.getMelyseg () << std::endl;
kiFile << "mean = " << binFa3.getAtlag () << std::endl;
kiFile << "var = " << binFa3.getSzoras () << std::endl;

kiFile.close ();
beFile.close ();
```

A main-ben meghívjuk az std függvénykönyvtár `move` függvényét. A kimeneti fájlba bekerül az eredeti és az új Binfá példányunk is. Ezután bezárjuk a ki- és bemeneti fájlokat.

Fordítás és futtatás után a lentebb található eredményt kapjuk: <https://github.com/mesterakos963/Prog1/blob/master/mozgat%C3%B31.png> <https://github.com/mesterakos963/Prog1/blob/master/mozgat%C3%B32.png>

DRAFT

## 7. fejezet

# Helló, Conway!

### 7.1. Hangyaszimulációk

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

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

Megoldás forrása: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/kozepes/Qt/sejtautomata/>

Ez a program a hangyák viselkedését próbálja meg szimulálni, ahogyan nyomot hagynak maguk után a társaik számára.

A hangya tulajdonságát az Ant osztály tartalmazza:

- x és y változók - a hangya oszlópa és sora;
- dir(direction) - a hangya mozgásiránya

Ezek alapján egy hangyáról egyértelműen meghatározható, hogy hol van és merre halad.

Az AntWin osztály a cellák méretét(pixelekben), rács számát (int\*\*) és a hangyák tulajdonságait tartalmazza. A hangyák tulajdonságai publikusak, tehát más osztályokból is elérhetők. A keyPress() függvény célja, hogy a 'P' lenyomásakor a program, így a hangyák is megálljanak. Ezt egy a running() függvényben található boolean típusú változó értékének változtatásával éri el.

Osztálydiagramm: <https://github.com/mesterakos963/Prog1/blob/master/diagramm.png>

### 7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

## 7.3. Qt C++ életjáték

Erre a feladatra elhasználnám egy, az SMNIST-ért kapott passzolási lehetőséget.

Most Qt C++-ban!

Megoldás videó:

```
#include <iostream>
#include <cstdlib>
#include <unistd.h>
#include <fstream>
#include <string>
#include <sstream>

using namespace std;

const int gridSize = 25;
void printGrid(bool gridOne[gridSize+1][gridSize+1]);
void determineState(bool gridOne[gridSize+1][gridSize+1]);
void clearScreen(void);

int main(){

 system("color A");
 bool gridOne[gridSize+1][gridSize+1] = {};
 int x,y,n;
 string nc;
 string start;
 string filename;
 cout << " THE GAME OF life - Implementation in ↵
 C++" << endl;
 cout << endl;
 cout << endl;
 cout << endl;
 cout << "Also known simply as life, " << endl;
 cout << "is a cellular automaton devised by the British mathematician ↵
 John Horton Conway in 1970." << endl;
 cout << endl;
 cout << "Rules" << endl;
 cout << "The universe of the Game of life is an infinite two- ↵
 dimensional orthogonal grid of square cells," << endl;
 cout << "each of which is in one of two possible states, life or dead. ↵
 Every" << endl;
 cout << "cell interacts with its eight neighbours, which are the cells ↵
 that are horizontally, vertically, or diagonally adjacent." << endl;
 cout << "At each step in time, the following transitions occur:" << ↵
 endl;
 cout << "1. Any live cell with fewer than two live neighbours dies, as ↵
 if caused by under-population." << endl;
```

```
cout << "2. Any live cell with two or three live neighbours lives on to ↵
the next generation." << endl;
cout << "3. Any live cell with more than three live neighbours dies, as ↵
if by over-population." << endl;
cout << "4. Any dead cell with exactly three live neighbours becomes a ↵
live cell, as if by reproduction." << endl;
cout << endl;
cout << "0 - living cell" << endl;
cout << "." - dead cell" << endl;
cout << endl;
cout << "Enter the number of cells, or 'r' to read cells from file: ";
cin >> nc;
cout << endl;

if (nc == "r")
{
while (true)
{

 cout << "Enter name of file to read from: "<<endl;
 cin >> filename;

 ifstream readfile(filename);
 if (readfile.is_open())
 {
string fileline,xx,yy;

while (getline(readfile,fileline))
{
 stringstream ss(fileline);

 getline(ss,xx,' ');
 getline(ss,yy,' ');

 x = stoi(xx);
 y = stoi(yy);

 gridOne[x][y] = true;
 }
break;
 } else {
 cout << "No such file, try again." << endl;
 }
 }
 }
else
{

for(int i=0;i<stoi(nc);i++)
{
```

```
 cout <<stoi(nc)<< "Enter the coordinates of cell " << i+1 << " : ";
 cin >> x >> y;
 gridOne[x][y] = true;
 printGrid(gridOne);
 }
}
cout << "Grid setup is done. Start the game ? (y/n)" << endl;
printGrid(gridOne);
cin >> start;
if(start == "y" || start == "Y")
{
 while (true)
 {
 printGrid(gridOne);
 determineState(gridOne);
 usleep(200000);
 clearScreen();
 }
}
else
{
 return 0;
}
}

void clearScreen(void) {
 // Tested and working on Ubuntu and Cygwin
 #if defined(_WIN32) || defined(WIN32) || defined(__MINGW32__) || ↵
 defined(__BORLANDC__)
 #define OS_WIN
 #endif

 #ifdef OS_WIN
 system("CLS");
 #endif

 #if defined(linux) || defined(__CYGWIN__)
 system("clear");
 #endif
}

void printGrid(bool gridOne[gridSize+1][gridSize+1]){
 for(int a = 1; a < gridSize; a++)
 {
 for(int b = 1; b < gridSize; b++)
 {
 if(gridOne[a][b] == true)
 {
 cout << " O ";
 }
 }
 }
}
```

```
 else
 {
 cout << " . ";
 }
 if(b == gridSize-1)
 {
 cout << endl;
 }
 }
}

void compareGrid (bool gridOne[gridSize+1][gridSize+1], bool gridTwo[←
gridSize+1][gridSize+1]){
 for(int a =0; a < gridSize; a++)
 {
 for(int b = 0; b < gridSize; b++)
 {
 gridTwo[a][b] = gridOne[a][b];
 }
 }
}

void determineState(bool gridOne[gridSize+1][gridSize+1]){
 bool gridTwo[gridSize+1][gridSize+1] = {};
 compareGrid(gridOne, gridTwo);

 for(int a = 1; a < gridSize; a++)
 {
 for(int b = 1; b < gridSize; b++)
 {
 int alive = 0;
 for(int c = -1; c < 2; c++)
 {
 for(int d = -1; d < 2; d++)
 {
 if(!(c == 0 && d == 0))
 {
 if(gridTwo[a+c][b+d])
 {
 ++alive;
 }
 }
 }
 }
 if(alive < 2)
 {
 gridOne[a][b] = false;
 }
 else if(alive == 3)
```

```
 {
 gridOne[a][b] = true;
 }
 else if(alive > 3)
 {
 gridOne[a][b] = false;
 }
 }
}
```

Conway életjátékának szabályai:

- minden cella, amelyiknek nincs szomszédja vagy csak 1 szomszédja van meghal (magányában)
- minden 4 vagy több szomszéddal rendelkező cella meghal (túlnépesedésben)
- a 2 vagy 3 szomszéddal rendelkező cellák életben maradnak

Ez a játék, nem egy tipikus számítógépes játék. Kitalálójá a Cambridge-i egyetem matematikusa, John Conway. 1970-ben vált ismertté, amikor a Scientific America című újság publikált egy cikket róla. Matematikai szempontból a selytautómaták közé tartozik. A játékos feladata csupán annyi, hogy megad egy kezdőalakzatot, a végkimenetel ettől az alakzattól függ.

<https://github.com/mesterakos963/Prog1/blob/master/lifegame.png>

## 7.4. BrainB Benchmark

Ezért a feladatért kaptam egy behúzást, nem volt kész határidőre.

Megoldás videó:

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

A program célja, hogy a felhasználóról megmondja, hogy mennyire lehet sikeres Esportoló. A futáshoz szükséges OpenCV szükséges.



## 8. fejezet

# Helló, Schwarzenegger!

### 8.1. Szoftmax Py MNIST

aa Python

Megoldás videó:

Megoldás forrása: [https://github.com/tensorflow/tensorflow/blob/r1.4/tensorflow/examples/tutorials/mnist/mnist\\_deep.py?fbclid=IwAR3vlodRRcrLj5qK\\_LdJQfHijVNyW8HYOgru-NLEajGSONkPBRcP5JCL6-E](https://github.com/tensorflow/tensorflow/blob/r1.4/tensorflow/examples/tutorials/mnist/mnist_deep.py?fbclid=IwAR3vlodRRcrLj5qK_LdJQfHijVNyW8HYOgru-NLEajGSONkPBRcP5JCL6-E)

A TensorFlow egy nyílt szoftver könyvtár, amit gépi tanuláshoz és neurális hálókhoz használnak. A Google Brain fejleszti, 2015. november 9-én jelent meg.

### 8.2. Szoftmax R MNIST

R

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

### 8.3. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

## 8.4. Deep dream

Keras

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

## 8.5. Robotpszichológia

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

## 9. fejezet

# Helló, Chaitin!

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

Megoldás videó:

Megoldás forrása:

### 9.2. Weizenbaum Eliza programja

Éleszd fel Weizenbaum Eliza programját!

Megoldás videó:

Megoldás forrása:

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

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

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

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

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

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

Tanulságok, tapasztalatok, magyarázat...

---

## 9.5. Lambda

Hasonlítsd össze a következő programokat!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

## 9.6. Omega

Megoldás videó:

Megoldás forrása:

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, Berner-Lee!

### 10.1. Java és C++ összehasonlítás

A Java nyelv teljesen objektumorientált. Egy osztály két dologból tevődik össze: mezők(adatok tárolása), metódusok(adatokon végezhető műveletek kódja).

Futtatáshoz szükségünk van egy Java nevű fordítóprogramra. Ez egy bájtódknak nevezett formátumra fordítja le a forráskódot, amelyet majd a Java Virtuális Gép önálló interpreterként fog értelmezni. Ez lassabb futást eredményez. A legnagyobb különbség a Java illetve a C++ között itt mutatkozik. A C++ kézenfekvőbb, ha olyan programokat szeretnénk írni, amelyeknél a futás sebessége elsődleges szempont. A Java inkább a web-es téren elterjedt.

A Java nyelvnek vannak egyszerű típusai is, melyeket az adatok egyszerű reprezentálására lehet használni. Ezeknek értéket adni az '=' operátorral lehet. Ilyenkor ez valódi értékadást jelent, összetett típusok esetében csak egy referencia átmásolását jelenti.

Eltérés még a C++ és a Java között, hogy Java-ban már 16 bites Unicode karaktereket is lehet használni változók vagy konstansok deklarálásához, tehát használhatunk ékezetes karaktereket, görög ábécé betűit, stb.

Megjegyzéseket ugyanúgy adhatunk hozzá a kódhoz, mint a C++ esetében.

A Java nyelvben semmilyen explicit eszköz nincs egy objektum megszüntetésére, egyszerűen nem kell hivatkozni rá és magától meg fog szűnni. Hivatkozás megszüntetéséhez az eddigi referencia helyett a null referenciát adjuk neki értékül.

A Java nyelvben tömböket a [] jelöléssel lehet megadni. A C++-tól eltérően ez egy igazi típus lesz és nem csak a mutató típus egy másik megjelenítési formája. A tömb típusok nem primitív típusok, a tömb típusú változók objektumhivatkozást tartalmaznak. Eltérés még a C++ és a Java között, hogy a Java-ban nincsenek többdimenziós tömbök, erre a tömb a tömbben megoldást használja.

További eltérés még a Goto utasítás teljes hiánya, ez a Java nyelvből kimaradt.

Java-ban egy objektumot a következő képen példányosítunk.

```
#Adott egy objektum
public class Alkalmazott {
String nev; int fiztes;
void fizetesemeles (int novekmenny) {
```

```
fizetes += novekmény
 }
}
#Példányosítás
Alkalmazott a = new Alkalmazott();
```

A new operátor mögött adjuk meg, h melyik osztályt példányosítjuk. A zárójelek közé közé az adott esetben a konstruktornak szánt paramétereket írjuk. Csakúgy, mint a C++-ban, itt is vannak nyilvános (public) és privát (private) tagok. A private tagokhoz az adott osztályon kívül nem férhet hozzá semmi.

Java megkülönbözteti a referenciát és az imperatív programozási nyelvekben használt mutatót aképpen, hogy a kifejezésekben automatikusan a mutatott objektumot jelenti, nem pedig a címet.

Az öröklődés legegyszerűbb esete, amikor egy létező osztályt egy másik osztállyal bővítünk ki. Ez jelentheti új műveletek vagy változók bevezetését. A már létező osztály lesz a szülő osztály, amivel kibővítettük pedig a gyermek. A gyermek a benne megadott tagok kívül a szülő tagjaival is rendelkezni fog. Ezt nevezzük öröklődésnek. Azonban a gyermek csak azokhoz a tagokhoz fér hozzá, amelyikhez a szülő engedi, viszont a gyermek példányai rendelkeznek a szülő összes tagjával.

Mivel a gyermek őseinek minden változójával és metódusával a bír, minden olyan környezetben használható, ahol ősei használhatók. Ezt nevezzük polimorfizmusnak.

Az osztályok rokonsági viszonyainak összességét osztályhierarchiának hívjuk. Az Object osztályból származtatva fentről lefelé növe fa ként lehet elképzelni.

Absztrakt osztályok - megvalósítás(törzs) nélküli metódusokat megvalósító osztályok. A Java nyelvben az abstract módosító jelöli az absztrakt osztályokat valamint metódusokat. Ezek az osztályok nem példányosíthatók, nem lehet private, final, static vagy native módosítóval ellátottak.

Final módosító - megakadályozza az előtaggal ellátott osztály felüldefiniálását.

Interfészeknek az olyan osztályokat nevezzük, amelyek tartalmazhatnak absztrakt (törzs nélküli) függvényeket. Az interfészek, mint egy külső felület engedik, hogy belőlük származtatott osztályokból példányosított objektumokat egységesen kezeljük. Szintén abstract kulcsszóval adjuk meg az absztrakt metódusokat. Az absztrakt osztályokat példányosítani nem lehet. Helyette azt mondjuk, hogy megvalósít vagy implementál egy absztrakt metódust, ha szülőjének egy absztrakt metódusát felüldefiniálja, azaz más implementációt ad rá.

Az UML az egyetlen szabványos modellező nyelve az általános szoftverfejlesztésnek. Leggyakoribb felhasználási területe az UML osztálydiagram. Nagyon jól szemlélteti a programban található osztályokat, interfészeket és egyéb típusokat és köztük fennálló kapcsolatot.

A java programok forráskódjában tetszőleges unicode karakterek használhatóak, mivel ez több biten tárolja a karaktereket, mint például az ASCII és az EBCDIC, amelyek 8 biten tárolják őket. Ezeket használva nem lehetne alkalmazni a magyar ű és ő betűket. Megfelelő alternatíva lehetne még a Latin-2 karakterkészlet, azonban az unicode sokkal célszerűbb.

Tokenek - lexikális egységek, amelyekből utasítások és kifejezések állíthatók össze.

Tokenek lehetnek:

- fehér-karakterek
- azonosítók - betűvel kezdődnek és betűvel vagy számmal folytatódnak;



- operátorok - szimbólumok, amelyek értékeken végeznek műveletet;
- literálok - egyszerű típusok inicializálásához használatosak;
- kulcsszavak - a rendszer által lefoglalt szavak, azonosítók nem lehetnek;
- punctuatorok (C++ esetén).

Utasítás lehet:

- deklaráció-utasítás - lokális változó létrehozását és tetszés szerinti inicializálást jelent;
- kifejezés-utasítás - ez lehet értékadás, metódushívás, ++ vagy -- operátorokkal képzett kifejezések, példányosítás.

Közös bennük, hogy pontosvessző zárja le őket. Tetszés szerint válthatják egymást.

Java-ban jelentős az eltérés primitív típusú változók és egyéb változók között. A nem primitív változók objektumhivatkozásokat tartalmaznak, osztályként kell rájuk tekinteni.

Java nyelv primitív típusai:

- byte
- short
- int
- boolean
- char
- long
- float
- double

Konstans változó - bármilyen típus lehet, létrehozáskor kezdőértékkel kell ellátni. Egy fordítási egységen belül szabadon hivatkozhatunk más változókra. Ha nincs körkörös hivatkozás, akkor a fordító biztosítja a helyes sorrendet.

Minden osztálynak van konstruktora, amik példányosítás során hajtódnak végre automatikusan. Neve megegyezik az osztály nevével. Ez a metódus adja meg az objektum kezdeti állapotát. A destruktort a konstruktor ellentéte. Az erőforrások felszabadításához van rá szükség.

Operátor túlterhelés - az operátorok segítségével műveleteket lehet végezni az egyszerű változókon. Ezeket az osztályok túl tudják tölteni, így saját működést társítanak hozzá.

Indexer (indexelő) - ezek segítségével úgy indexelhetünk osztályokat, mint tömböket.

Beágyazott osztályok - két osztály között lévő szoros kapcsolat kifejezésére használjuk őket. Egy osztályt megadhatunk egy másik osztályon belül (beágyazás). A beágyazott osztály a tartalmazó osztályon kívül nem létezhet.

Gyűjtemények - olyan eszközök, amelyek egy vagy több típusba tartozó objektumok példányait a memóriában tárolja, manipulálja vagy lekérdezi. Ezek minden programnyelvben megtalálhatóak (van néhány kivétel). Legjellemzőbb fajtája a tömb és az objektum referenciák vagy mutatók segítségével történő összekapcsolás. Java 2-vel került be a nyelvbe.

Funkcionális nyelvek - deklaratív nyelvek, a program végrehajtását egy függvény kiértékelésének tekinti. Egy kezdetleges funkcionális nyelv esetén nem lehet változóról sem értékadásról beszélni. A függvénye csupán a bemeneti értékek alapján számítják ki a megfelelő kimenetet.

Lambda kifejezés - az imperatív nyelvekbe átemelt legnépszerűbb funkcionális elem. Lehetőséget ad inline függvények létrehozására. Rövid kódrészletek, általában "egyszerhasználatosak", tehát nincs értelme elnevezni őket. Ott írjuk meg őket, ahol használni szeretnénk.

Java-ban minden adatfolyam típusnak egy osztály fog megfelelni.

Három féle képen lehet osztályozni őket:

- adatfolyam típusa alapján - bájtfolyam, karakterfolyam;
- adatfolyam iránya alapján - bemeneti és kimeneti;
- feladatuk alapján - adatfolyam forrása valamint meglévő adatot kiegészítő.

A fájlok manipulálásához Java-ban nem az adatfolyamok alkalmatlanok sok esetben, mivel nincs biztosítva egyidőben a fájlok olvasása és írása. A probléma megoldásához egy közvetlen elérésű fájl (random access file) segít. Legkönnyebben egy fájlrendszerben tárolt bájttömbként képzelhető el, amelynek bármelyik pozíciójához olvashatunk és írhatunk adatot. Minden ilyen objektumhoz tartozik egy fájlmutató. Ez egy pozíciót jelöl a fájlban. Ettől a mutatótól kezdődnek a kiíró és beolvasó műveletek.

Serializáció - az objektumok byte-sorozattá alakítását jelenti. Az adatok külső tárolásában játszik fontos szerepet. Csak az az osztály szerilizálható, amelyik megvalósítja a serializable interfészt. Ha egy serializálható, akkor a leszármazottai is.

Kivételkezelés - mechanizmus, amely biztosítja, hogy hiba esetén a futás a hibakezelő ágon folytatódjon. Bármilyen kivételes helyzet esetén használható.

Java exception - olyan kivételes esemény, amely hatására az utasítások normál végrehajtási sorrendje megszakad. A Java kivételkezelés - ahogy minden más is - osztályokon alapszik. Amikor egy kivétel keletkezik egy metódus futása közben, akkor egy hiba objektum jön létre, ami tartalmazza a hiba tulajdonságait. A hiba kezelése után a vezérlés nem tér vissza a hibát kiváltó részhez.

Annotációk és metaadatok - információt biztosítanak a programról, de nem részei annak, nem befolyásolják a kód működését, amelyet annotálnak. Annotációval adhatunk információt a fordítóprogram számára, amit például hibakeresésnél hasznosíthat. A Java 5-tel kerültek be a nyelvbe. Egy annotációs típus lényegében egy speciális interfésztípus. Nevét egy '@' jel előzi meg.

Egy paradigma jellemzi egy nyelv stílust, fogalmait, eszközeit, képességeit. A paradigmák általában egy adott területen használhatók, viszont kombinált feladatot megoldására ezeket kombinálni kell. Az ilyen feladatokhoz használjuk a multiparadigmás nyelveket.

Multiparadigmás programozási nyelv - olyan nyelv, amely kettő vagy több programozási paradigmát támogat. Biztosítja, hogy a programozó különböző stílusokban tudjon dolgozni, a paradigma elemeit szabadon kombinálja valamint, hogy lehetővé tegye a programozó számára, hogy a legjobb eszközt használja egy probléma megoldására. Procedurális paradigmát követnek, tehát lépésről lépésre írják le a probléma megoldását.

## 10.2. Bevezetés a mobilprogramozásba

A Python egy általános célú programozási nyelv. Alkotója Guido van Rossum. Leginkább prototípus készítésre és tesztelésre szokás alkalmazni. Képes együttműködni más nyelveken íródott modulokkal. Elterjedtségét főként az imént említett tulajdonságának, valamint a rövid tanulási ciklusnak köszönheti. A Python-ban megírt programok terjedelmükben jóval rövidebbek, mint a C, C-ben, C++-ban vagy Java-ban készült, velük ekvivalens társaik. A kódok tömörek és könnyen olvashatóak. A kódcsoportosításhoz új sort és tabulátort használ (behúzásalapú), nincs szükség nyitó és zárójelekre, nincs szükség változó vagy argumentumdefinícióra. Az utasítások a sorok végéig tartanak, nincs szükség ';' -re, ha egy adott utasítást a következő sorban szeretnénk folytatni, akkor ezt a '\n' jellel tehetjük meg.

Példa:

```
if feltétel1 és feltétel2
 alapfeladat()
 egyébfeladat()
```

Az értelmező a sorokat tokenekre bontja, amelyek közt tetszőleges üres (whitespace) karakter lehet. A tokenek lehetnek: azonosító(változó, osztály, függvénymodul), kulcsszó, operátor, delimiter, literál.

Lefoglalt kulcsszavak: and, del, for, is, raise, assert, elif, from, lambda, return, break, else, global, not, try, class, except, if, or, while, continue, exec, import, pass, yield, def, finally, in, print.

Típusok

A Pythonban minden adatot típusok reprezentálna. Az adatokon végezhető műveleteket az objektum típusa határozza meg. A változók típusait a rendszer futási időben "kitalálja".

Számok lehetnek: egészek(decimális, oktális, hexadecimális), lebegőpontosak és komplex számok.

Sztringeket aposztrófok közé írva adhatunk meg, illetve a 'u' betű használatával Unicode szövegeket is felvehetünk.

Példa:

```
print u"Hello %s, kedves %s!"
```

Az ennesek (tuples) objektumok gyűjteményei vesszővel elválasztva.

Példa:

```
('a','b','c') #három elemű ennes
tuple('abc') #tuple generálás kulcsszóval (ugyanaz, mint az előző)
() #üres ennes
(1, "szia", 3,) #három elemű ennes, az utolsó vessző elhagyható
```

A lista különböző típusú elemek rendezett szekvenciája, elemeit szögletes zárójelek közé írjuk, dinamikusan nyújtózkodik. Az elemeket az indexükkel azonosítjuk.

Példa:

```
[a','b','c'] #három elemű lista
list('abc') #lista generálás kulcsszóval (ugyanaz, mint az előző)
[] #üres ennes
[1, "szia", 3,] #három elemű lista, az utolsó vessző elhagyható
```

A szótár kulcsokkal azonosított elemek rendezetlen halmaza. Kulcs lehet: szám, sztring stb.

Példa:

```
{'a':1, 'b':5, 'e':1982}
{1:1, 2:1, } #az utolsó vessző elhagyható
{ } #üres szótár
```

Változók Pythonban - egyes objektumokra mutató referenciák. Típusaik nincsenek. A hozzárendelést az '=' karakterrel végezzük. A del kulcsszóval törölhető a hozzárendelés, a mögöttes objektum törlését a garbage collector fogja elvégezni. Léteznek globális és lokális változók (akárcsak a C++-ban). A változók közötti típuskonverzió támogatott.

A szekvenciákon (sztringek, listák, ennesek) műveletek végezhetőek el (összefűzés, szélsőérték helyek meghatározása stb).

Az elágazások és ciklusok működése megegyezik a többi programnyelvével.

Függvényeket a def kulccszóval definiálunk, tekinthetünk rájuk úgy, mint értékekre, mivel továbbadhatóak más függvényeknek illetve objektumkonstruktoroknak.

```
def hello();
 print "Hello world!"
 return
```

A Python támogatja a klasszikus, objektumorientált fejlesztési eljárásokat (osztályokat definiálunk, amelyek példányai az objektumok. Osztályoknak lehetnek attribútumaik(objektumok, függvények) illetve örökölhetnek más osztályokból.

A kivételkezelés:

Példa

```
try:
 utasítások
except [kifejezés]: #akkor fut le, ha az történik, amit a try után leírtunk
[else:
 utasítások]
```

## 11. fejezet

# Helló, Liskov!

### 11.1. Liskov helyettesítés sértése

Megoldás forrása:

Java: <https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloLiskov/Liskov.java>

C++: <https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloLiskov/Liskov.cpp>

A feladat azt kérte tőlünk, hogy írjunk olyan leforduló Java ill. C++ kódot, ami sérti a Liskov elvet, ami a következőt jelenti: ha A altípusa B-nek, akkor minden olyan helyen ahol A-t felhasználjuk B-t is minden gond nélkül behelyettesíthetjük anélkül, hogy a programrész tulajdonságai megváltoznának.

A mi esetünkben A a Teglalap, míg B a Negyzet osztállyal egyezik meg, a Negyzet gyermeke a Teglalap osztálynak. A Teglalap osztály tartalmazza a magasság és szélesség értékeket és ez alapján könnyen kiszámolhatjuk a területet, ami adott esetben  $5 \cdot 10 = 50$  lenne. A Negyzet osztály is tartalmazza a saját értékeit, de a négyzet tulajdonságát - miszerint minden oldala egyforma - szemelőtt tartva, a magasságot a szélességgel tesszük egyenlővé. A Liskov elv szerint ha A-t behelyettesíthetjük, akkor B-t is, mivel B gyermeke A-nak. Ebből kiindulva példányosításnál nem egy téglalapot példányosítunk, hanem egy négyzetet.

A programot futtatva nem a várt  $5 \cdot 10 = 50$  értéket kapjuk, hanem 100-at. Ennek oka, hogy Negyzet objektumban a magasságot egyenlőítettük a szélességgel, ami egy téglalap esetében nem igaz. Ez a hiba könnyen kiküszöbölhető lenne, ha szem előtt tartanánk a Liskov elvet.

### 11.2. Szülő-gyerek

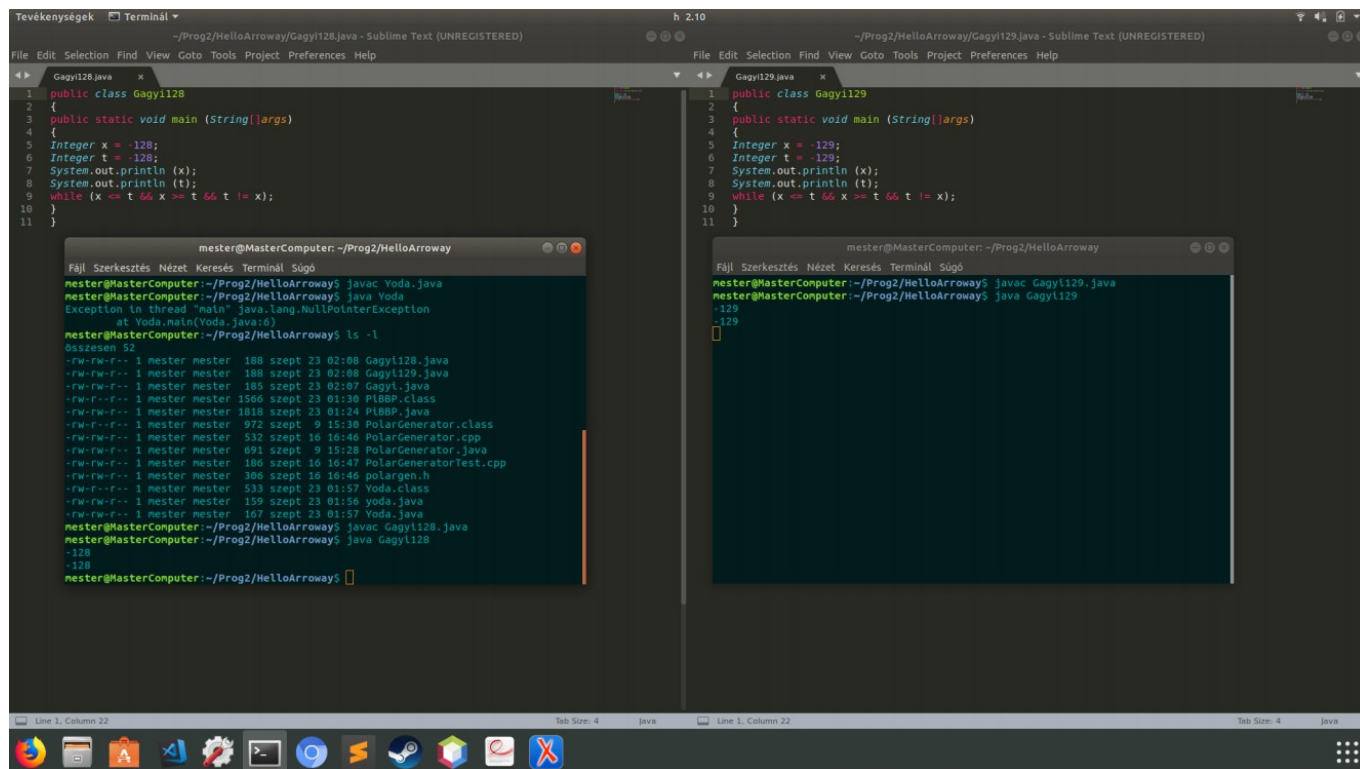
Megoldás forrása:

Java: <https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloLiskov/szulo.java>

C++: <https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloLiskov/szulogyerek.cpp>

Ebben a feladatban az volt a cél, hogy írjunk olyan Java ill. C++ kódot, ami rávilágít arra, hogy az ősön keresztül csak az ős üzenetei érhetők el. Tehát ha definiálunk egy függvényt az altípusban és az altípust szupertípusként használjuk, akkor a saját függvényei nem lesznek elérhetőek.

Futtatás után az alábbi hibüzenetet kapjuk Java-ban:



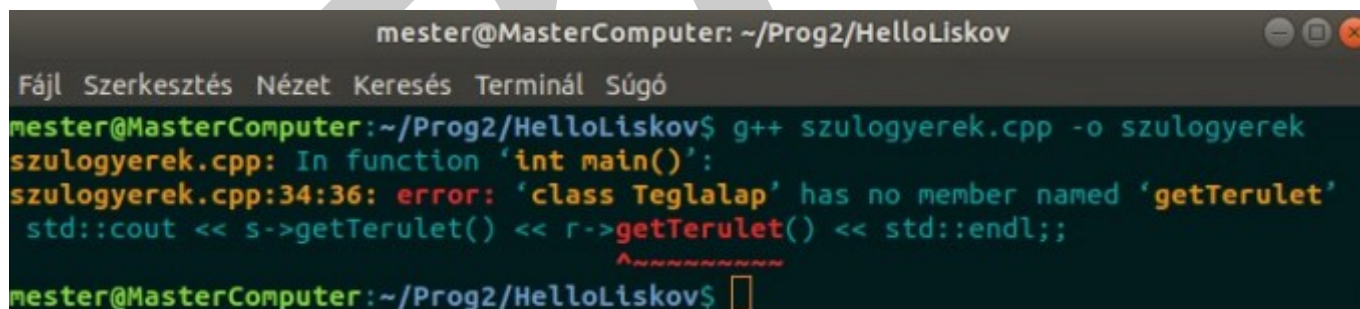
```
1 public class Gagyi128
2 {
3 public static void main (String[] args)
4 {
5 Integer x = -128;
6 Integer t = -128;
7 System.out.println (x);
8 System.out.println (t);
9 while (x == t && x == t && t != x);
10 }
11 }
```

```
mester@MasterComputer: ~/Prog2/HelloArroway
mester@MasterComputer:~/Prog2/HelloArroway$ javac Yoda.java
mester@MasterComputer:~/Prog2/HelloArroway$ java Yoda
Exception in thread "main" java.lang.NullPointerException
 at Yoda.main(Yoda.java:6)
mester@MasterComputer:~/Prog2/HelloArroway$ ls -l
összesen 52
-rw-rw-r-- 1 mester mester 188 szept 23 02:00 Gagyi128.java
-rw-rw-r-- 1 mester mester 188 szept 23 02:00 Gagyi129.java
-rw-rw-r-- 1 mester mester 185 szept 23 02:07 Gagyi.java
-rw-rw-r-- 1 mester mester 1506 szept 23 01:30 PIBBP.class
-rw-rw-r-- 1 mester mester 1818 szept 23 01:24 PIBBP.java
-rw-rw-r-- 1 mester mester 972 szept 9 15:30 PolarGenerator.class
-rw-rw-r-- 1 mester mester 532 szept 16 16:46 PolarGenerator.cpp
-rw-rw-r-- 1 mester mester 691 szept 9 15:28 PolarGenerator.java
-rw-rw-r-- 1 mester mester 186 szept 16 16:47 PolarGeneratorTest.cpp
-rw-rw-r-- 1 mester mester 306 szept 16 16:46 polargen.h
-rw-rw-r-- 1 mester mester 533 szept 23 01:57 Yoda.class
-rw-rw-r-- 1 mester mester 159 szept 23 01:56 yoda.java
-rw-rw-r-- 1 mester mester 167 szept 23 01:57 Yoda.java
mester@MasterComputer:~/Prog2/HelloArroway$ javac Gagyi128.java
mester@MasterComputer:~/Prog2/HelloArroway$ java Gagyi128
-128
-128
mester@MasterComputer:~/Prog2/HelloArroway$
```

```
1 public class Gagyi129
2 {
3 public static void main (String[] args)
4 {
5 Integer x = -129;
6 Integer t = -129;
7 System.out.println (x);
8 System.out.println (t);
9 while (x == t && x == t && t != x);
10 }
11 }
```

```
mester@MasterComputer: ~/Prog2/HelloArroway
mester@MasterComputer:~/Prog2/HelloArroway$ javac Gagyi129.java
mester@MasterComputer:~/Prog2/HelloArroway$ java Gagyi129
-129
-129
mester@MasterComputer:~/Prog2/HelloArroway$
```

És C++-ban:



```
mester@MasterComputer: ~/Prog2/HelloLiskov
Fájl Szerkesztés Nézet Keresés Terminál Sűgó
mester@MasterComputer:~/Prog2/HelloLiskov$ g++ szulogyerek.cpp -o szulogyerek
szulogyerek.cpp: In function 'int main()':
szulogyerek.cpp:34:36: error: 'class Teglalap' has no member named 'getTerulet'
 std::cout << s->getTerulet() << r->getTerulet() << std::endl;;
 ^
mester@MasterComputer:~/Prog2/HelloLiskov$
```

Szülőn keresztül nem hívhatunk meg olyan metódust, amit a gyermek igen, a szülő viszont nem definiált. Ez a dinamikus kötésnek tudható be, ami azt jelenti, hogy futási időben rendeli hozzá a metódust az objektumhoz.

### 11.3. Ciklomatikus komplexitás

Feladat azt kérte tőlünk, hogy számoljuk ki egy tetszőleges program ciklomatikus komplexitását. Ezt megtehettük képlet alapján vagy valamilyen program segítségével.

De kezdjük ott, hogy mit is jelent a ciklomatikus komplexitás.

Egy szoftver metrika, másnéven McCabe-komplexitásnak is nevezik. A gráfelméletre alapul, a forráskódban az elágazó gráfok pontjai és a köztük lévő élek száma alapján számítható ki. A végeredmény egy pozitív egész szám lesz.

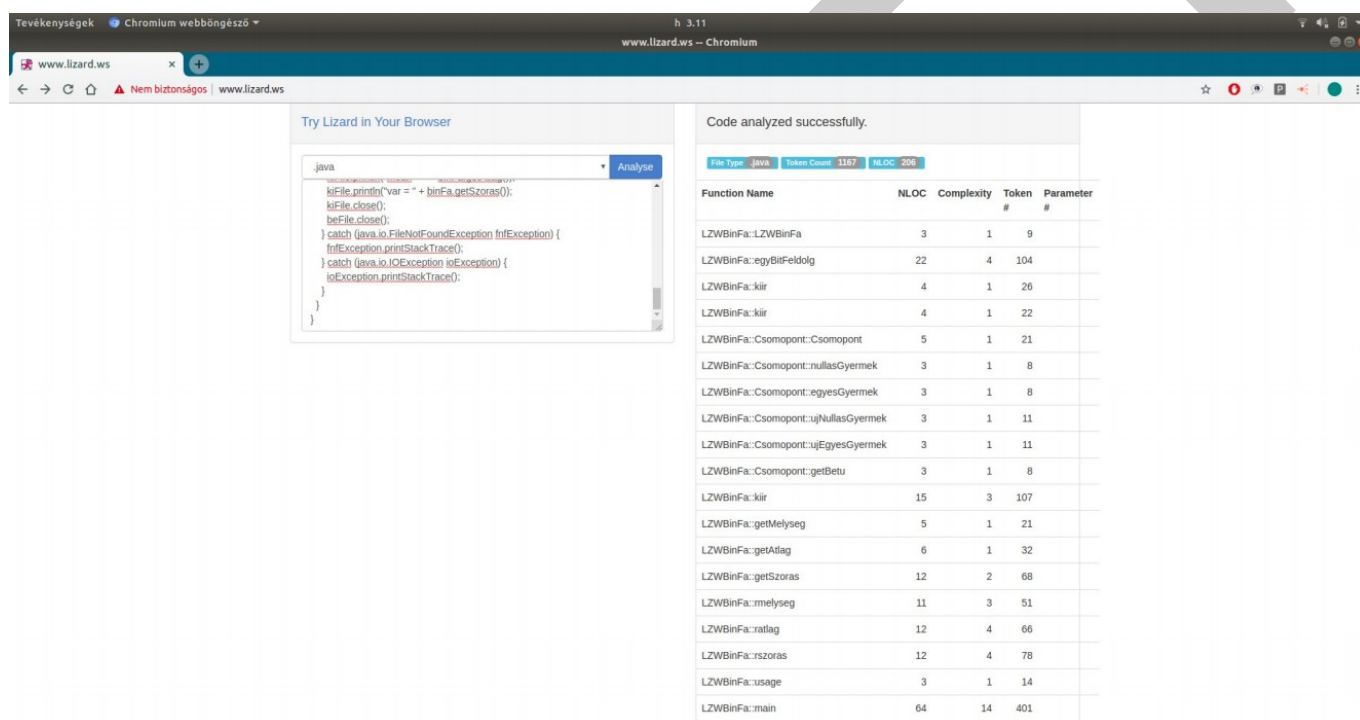
A képlet:

$M = E - N + 2P$ , ahol: M - ciklomatikus komplexitás, E-gráf éleinek száma, N-gráfban lévő csúcsok száma, P-összefüggő komponensek száma.

Az általam választott program, a már elég jól ismert binfa program, mivel ez egy elég összetett algoritmus, így a végeredménynek is egy elég nagy szám várható. Viszont egy ilyen kód esetében ezt képlettel kiszámolni elég hosszasan lenne, valamint nagyon nagy a hibalehetőség, ezért én egy online kalkulátor segítségével számoltam ezt ki.

<http://www.lizard.ws/>

Az eredmény a következő lett:



The screenshot shows the Lizard website interface. On the left, a code editor displays a Java snippet. On the right, a table titled 'Code analyzed successfully.' shows the analysis results. The table has columns for Function Name, NLOC, Complexity, Token #, and Parameter #. The data is as follows:

| Function Name                          | NLOC | Complexity | Token # | Parameter # |
|----------------------------------------|------|------------|---------|-------------|
| LZWBInFa::LZWBInFa                     | 3    | 1          | 9       |             |
| LZWBInFa::egyBisFeldolgo               | 22   | 4          | 104     |             |
| LZWBInFa::kilor                        | 4    | 1          | 26      |             |
| LZWBInFa::kilor                        | 4    | 1          | 22      |             |
| LZWBInFa::Csomopont::Csomopont         | 5    | 1          | 21      |             |
| LZWBInFa::Csomopont::nullasGyermekek   | 3    | 1          | 8       |             |
| LZWBInFa::Csomopont::egyesGyermekek    | 3    | 1          | 8       |             |
| LZWBInFa::Csomopont::ujNullasGyermekek | 3    | 1          | 11      |             |
| LZWBInFa::Csomopont::ujEgyesGyermekek  | 3    | 1          | 11      |             |
| LZWBInFa::Csomopont::getBetu           | 3    | 1          | 8       |             |
| LZWBInFa::kilor                        | 15   | 3          | 107     |             |
| LZWBInFa::getMelyseg                   | 5    | 1          | 21      |             |
| LZWBInFa::getAllag                     | 6    | 1          | 32      |             |
| LZWBInFa::getSzoras                    | 12   | 2          | 68      |             |
| LZWBInFa::rmelyseg                     | 11   | 3          | 51      |             |
| LZWBInFa::ratlag                       | 12   | 4          | 66      |             |
| LZWBInFa::rszoras                      | 12   | 4          | 78      |             |
| LZWBInFa::usage                        | 3    | 1          | 14      |             |
| LZWBInFa::main                         | 64   | 14         | 401     |             |

Ahogy látszik ez az kalkulátor egy elég részletes eredményt ad vissza.

## 11.4. Anti OO

A feladat azt kérte tőlünk, hogy BBP algoritmussal 7 a Pi hexadecimális kifejtésének a 0. pozíciótól számított  $10^6$ ,  $10^7$ ,  $10^8$  darab jegyét határozzuk meg C, C++, Java és C# nyelveken és hasonlítsuk össze a futási idejüket.

Mivel az algoritmus a "Kódolás from scratch" nevű feladatunkból származik, ezért most nem ejtek szót a forrásról. Ebben a feladatban inkább a végeredmény a fontos, amit a lenti táblázatba foglaltam.

Forrásfájlok:

Java: <https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloLiskov/PiBBPBench.java>



C:[https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloLiskov/pi\\_bbp\\_bench.c](https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloLiskov/pi_bbp_bench.c)

C++:<https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloLiskov/pibbpbench.cpp>

C#:<https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloLiskov/PiBBPBench.cs>

Az eredményeket tartalmazó táblázat:

|        | Java    | C          | C++     | C#        |
|--------|---------|------------|---------|-----------|
| $10^6$ | 2,001   | 2.218048   | 2,54988 | 2,040878  |
| $10^7$ | 23,5    | 26.349407  | 29,682  | 23,749305 |
| $10^8$ | 268,231 | 303.937076 | 336,813 | 273,46592 |

11.1. táblázat.

Java:

```
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ java PiBBPBench
6
1.986
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ javac PiBBPBench.java
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ java PiBBPBench
7
23.276
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ javac PiBBPBench.java
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ java PiBBPBench
8
268.231
```

C++:

```
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$./pibbp.cpp
6
2.54988
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ g++ pibbpbench.cpp -o pibbp.cpp
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$./pibbp.cpp
7
29.682
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ g++ pibbpbench.cpp -o pibbp.cpp
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$./pibbp.cpp
8
336.813
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$
```

C#:



```
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ mcs PiBBPBench.cs
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$./PiBBPBench.exe
6
2,040878
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ mcs PiBBPBench.cs
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$./PiBBPBench.exe
7
23,749305
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ mcs PiBBPBench.cs
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$./PiBBPBench.exe
8
273,46592
```

C:

```
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ mcs PiBBPBench.cs
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$./PiBBPBench.exe
6
2,040878
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ mcs PiBBPBench.cs
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$./PiBBPBench.exe
7
23,749305
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$ mcs PiBBPBench.cs
mester@MasterComputer:~/Prog2/HelloLiskov/Becnhmark$./PiBBPBench.exe
8
273,46592
```

## 12. fejezet

# Helló, Mandelbrot!

### 12.1. A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása:

Program forráskódja:

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

#define N 500
#define M 500
#define MAXX 0.7
#define MINX -2.0
#define MAXY 1.35
#define MINY -1.35

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[x][y] = png::rgb_pixel(tomb[x][y], tomb[x][y], tomb[x][y] ←
);
 }
 }
 image.write("kimenet.png");
}

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] = 256 - iteracio;
 }
 }

 GeneratePNG(tomb);

 return 0;
}
```

### Make file

```
all: mandelbrot clean

mandelbrot.o: mandelbrot.cpp
 @g++ -c mandelbrot.cpp `libpng-config --cflags`
```

```
mandelbrot: mandelbrot.o
@g++ -o mandelbrot mandelbrot.o `libpng-config --ldflags`

clean:
@rm -rf *.o
@./mandelbrot
@rm -rf mandelbrot
```

A Mandelbrot halmaz egy olyan komplex halmaz, amely illeszkedik a  $f_c(z) = z^2 + c$  függvény képére, s nullától iterálva nem divergál, tehát a  $f_c(0), f_c(f_c(0)), \dots$  abszolútértékben korlátos. A program alapjául a png++ könyvtár szolgál. Ebben a feladatban nem használjuk az `std::complex` osztályt, hanem magunknak hozunk létre egy struktúrát, ami tartalmazza az imaginárius és képzetes egységet, amiket `double` változóban tárolunk. Ezután megadunk egy halmazt, amire az egyenletünk illeszkedik és az ennek megfelelő pixeleket kiszínezzük.

GeneratePNG függvény felelős a kimeneti png létrehozásáért.

A kimenet ezen a linken található: <https://github.com/mesterakos963/Prog1/blob/master/mandelbrot.png>

## 12.2. A Mandelbrot halmaz a `std::complex` osztállyal

```
#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");
}

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;
}
```

A program az előző feladatnak egy elegánsabb megoldása. Itt használjuk az `std::complex` osztályt, ami ad némi előnyt, de a végeredmény ugyanaz a kimenet lesz. Működési elve szintén annyi, hogy a Mandelbrot halmaznak megfelelő pixeleket kiszínezi.

Az `std::complex` osztállyal könnyedén reprezentálhatunk számokat a komplex számok halmazáról. Az osztályon belül sok előre deklarált változónak adhatunk értéket. `PL`: `real` - valós rész, `imag` - imaginárius rész, `abs` - komplex szám abszolút értéke, `conj` komplex szám konjugáltja. Az osztály használata a végeredményen nem változtat, csupán egy eszköz, ami megkönnyíti a programírást.

## 12.3. 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 szoros kapcsolatban állnak a Mandelbrot halmazzal. Itt is létrehozunk egy make file-t és futtatjuk a programot, ami létrehoz egy bmorf.png képet az előzőhöz hasonló módon, de mivel az egyenletünk más ebből egyértelműen következik, hogy a kapott ábra is másog fog kinézni. A formán kívül azonban az is szembeötlő, hogy ez az ábra már színes. Ezt úgy kapjuk meg, hogy az iteráció számát elosztjuk 255-tel, így RGB színezést kapunk.

Kimenet: <https://github.com/mesterakos963/Prog1/blob/master/bmorf.png>

## 12.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása:

## 12.5. 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:

```
// frakablak.cpp
//
// Mandelbrot halmaz nagyító

#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();
}

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

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);
 }
 qpainter.end();
}

void FrakAblak::mousePressEvent(QMouseEvent* event) {

 // A nagyítandó kijelölt területet bal felső sarka:
 x = event->x();
 y = event->y();
 mx = 0;
 my = 0;

 update();
}

void FrakAblak::mouseMoveEvent(QMouseEvent* event) {

 // A nagyítandó kijelölt terület szélessége és magassága:
 mx = event->x() - x;
 my = mx; // négyzet alakú

 update();
}

void FrakAblak::mouseReleaseEvent(QMouseEvent* event) {

 if(szamitasFut)
 return;

 szamitasFut = true;

 double dx = (b-a)/szelesseg;
```

```
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, szelesseg, magassag, ←
 iteraciosHatar, this);
mandelbrot->start();

update();
}

void FrakAblak::keyPressEvent(QKeyEvent *event)
{
 if(szamitasFut)
 return;

 if (event->key() == Qt::Key_N)
 iteraciosHatar *= 2;
 szamitasFut = true;

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

void FrakAblak::vissza(int magassag, int *sor, int meret)
{
 for(int i=0; i<meret; ++i) {
 QRgb szin = qRgb(0, 255-sor[i], 0);
 fraktal->setPixel(i, magassag, szin);
 }
 update();
}

void FrakAblak::vissza(void)
{
 szamitasFut = false;
```



```
x = y = mx = my = 0;
}
```

Ehhez a feladathoz a QT Creator nevű programot használjuk, mely egy multiplatform GUI építő szoftver.

A már meglévő Mandelbrot-halmaz kódhoz építünk felhasználói felületet. Használatához kijelöljük az újragenerálandó területet, ezzel érjük el a nagyítást.

A szükséges kiegészítő kódok itt találhatóak: <https://sourceforge.net/p/udprog/code/ci/master/tree/source-labor/Qt/Frak/>

## 12.6. Mandelbrot nagyító és utazó Java nyelven

Ehhez a feladathoz az OpenJFX 11 nevű szoftvert használni, ami a QT-hoz hasonló GUI építő szoftver.

A Mandelbrot nagyító és utazó JAVA nyelvű megvalósítása itt található: <https://sourceforge.net/p/udprog/code/ci/master/tree/source-labor/Qt/Frak/>

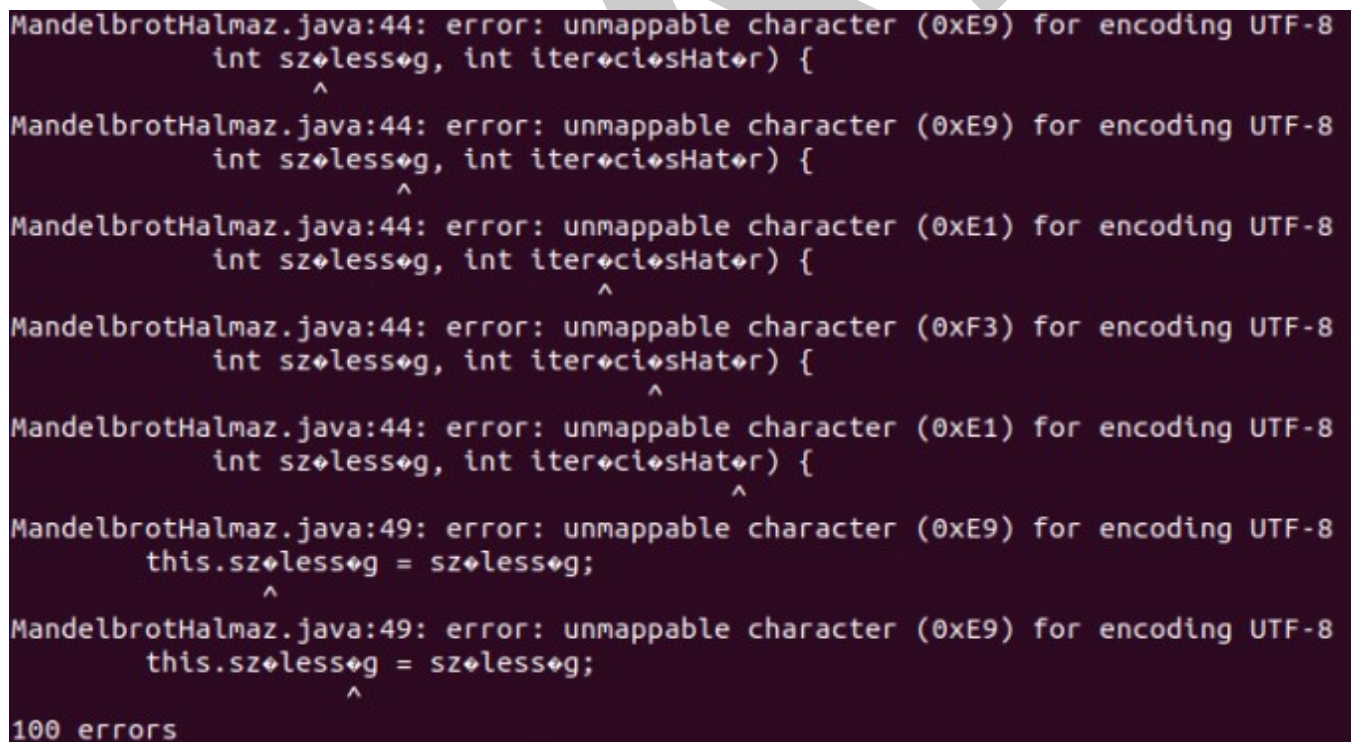
## 13. fejezet

# Helló, Chomsky!

### 13.1. Encoding

A feladatunk az volt, hogy fordítsuk le és futtassuk a Javát tanítók könyv MandelbrotHalmazNagyító.java forrását az ékezetes betűk meghagyásával a fájlnevekben és a forrásban. Ez a feladat a karakterkészlet gyakorlatbeli fontosságára világít rá.

Fordításkor elég sok error-ba ütköztünk.

A screenshot of a Java compilation error log. It shows multiple instances of the error: "MandelbrotHalmaz.java:44: error: unappable character (0xE9) for encoding UTF-8" and "MandelbrotHalmaz.java:49: error: unappable character (0xE9) for encoding UTF-8". The errors point to specific lines of code in the file. At the bottom, it says "100 errors".

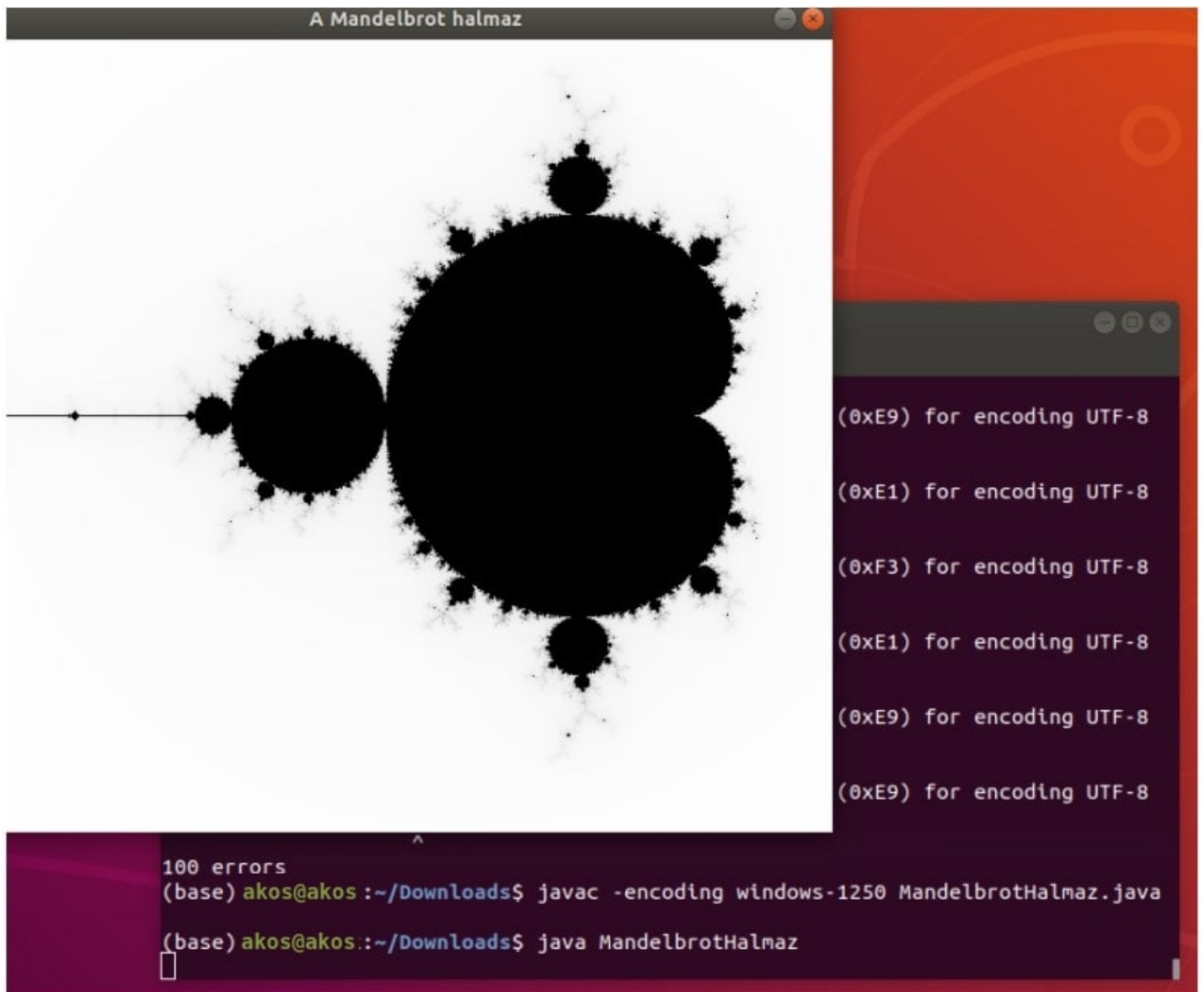
```
MandelbrotHalmaz.java:44: error: unappable character (0xE9) for encoding UTF-8
 int szélesség, int iterációsHatór) {
 ^
MandelbrotHalmaz.java:44: error: unappable character (0xE9) for encoding UTF-8
 int szélesség, int iterációsHatór) {
 ^
MandelbrotHalmaz.java:44: error: unappable character (0xE1) for encoding UTF-8
 int szélesség, int iterációsHatór) {
 ^
MandelbrotHalmaz.java:44: error: unappable character (0xF3) for encoding UTF-8
 int szélesség, int iterációsHatór) {
 ^
MandelbrotHalmaz.java:44: error: unappable character (0xE1) for encoding UTF-8
 int szélesség, int iterációsHatór) {
 ^
MandelbrotHalmaz.java:49: error: unappable character (0xE9) for encoding UTF-8
 this.szélesség = szélesség;
 ^
MandelbrotHalmaz.java:49: error: unappable character (0xE9) for encoding UTF-8
 this.szélesség = szélesség;
 ^
100 errors
```

A hibaüzenetek elég világossá teszik számunkra, hogy a kódolással van gond. A fordító azt mondja, hogy nem található karakter az UTF 8 kódolás számára. Ékezetes karaktereket a Latin1 és a Latin2 (ISO 8859-1 és ISO 8859-2) szabványok tartalmazznak. Viszont a fordítás ezeket használva sem ment.

Kis keresgélés után az alábbi oldalra akadtam:

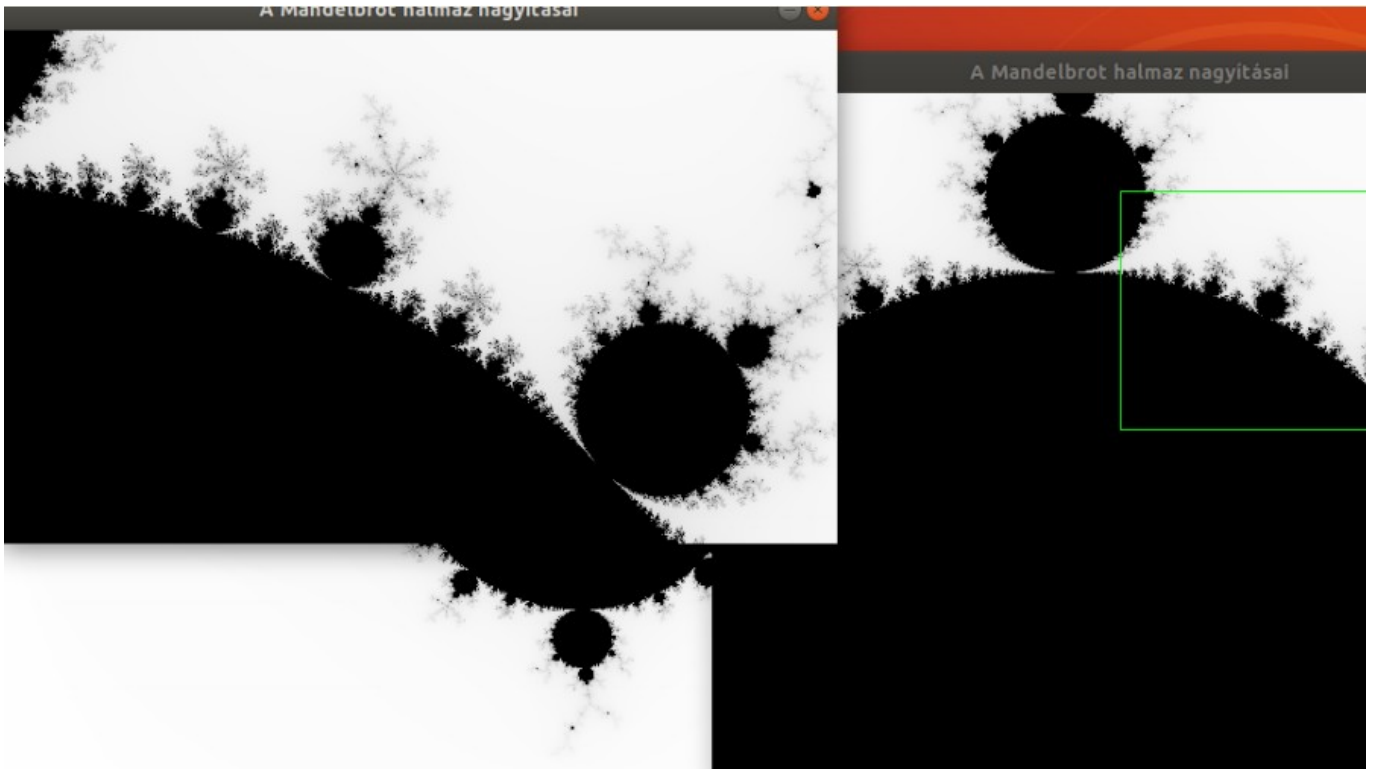
<https://docs.oracle.com/javase/9/intl/supported-encodings.htm#JSINT-GUID-18>

Megoldást az windows-1250\u0000 jelentett, ami az ISO-8859-1 windowsbeli megfelelője. Ezzel a kódolással a program már gond nélkül lefordult. Ehhez a -encoding flaget kellett használnunk.



Most, hogy van egy futó programunk haladunk tovább a feladattal, azaz először a Mandelbrot iterációkat fordítjuk le.

Nagyítás:



## 13.2. OOCWC lexer

A feladat azt kérte tőlünk, hogy izzítsuk be az OOCWC-t és vázoljuk a <https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/carlexer.ll> lexert és kapcsolását a programunk OO struktúrájába! Ez a feladat a lexerekre világít rá, de jelen esetben nem a programozási nyelv lexikális egységeiről lesz szó, hanem tokenekről.

A lexert bemeneti sztringet feldolgozására szokás használni, ahogy jelen esetben is. Van néhány előre meghatározott és nevesített reguláris kifejezés.

```
INIT "<init"
INITG "<init guided"
WS [\t]*
WORD [^-\:\n \t()]{2,}
INT [0123456789]+
FLOAT [-.0123456789]+
ROUTE "<route"
CAR "<car"
POS "<pos"
GANGSTERS "<gangsters"
STAT "<stat"
DISP "<disp>"
```

A `WS [ \t]*` a whitespace karakter előfordulását jelzi, a `'*'` azt jelenti, hogy bármennyiszer előfordulhat.

Az `INT [0123456789]+` a természetes számokat jelenti, a `'+'` karakter jelentése, hogy a felsorolt számok bármennyiszer, de minimum egyszer szerepelnek. Float esetén ugyanígy működik, azonban ott ki kell egészítenünk egy tizedes vesszővel.

Ha ezek közül a karakterláncok közül valamelyik szerepel a bemenetben, akkor az annak megfelelő kód fog ledutni.

```
{DISP} {
m_cmd = 0;
}

{POS}{WS}{INT}{WS}{INT}{WS}{INT} {

std::sscanf(yytext, "<pos %d %u %u", &m_id, &from, &to);
m_cmd = 10001;
}

{CAR}{WS}{INT} {

std::sscanf(yytext, "<car %d", &m_id);
m_cmd = 1001;
}

{STAT}{WS}{INT} {

std::sscanf(yytext, "<stat %d", &m_id);
m_cmd = 1003;
}

{GANGSTERS}{WS}{INT} {

std::sscanf(yytext, "<gangsters %d", &m_id);
m_cmd = 1002;
}

{ROUTE}{WS}{INT}{WS}{INT}({WS}{INT})* {

int size{0};
int ss{0};
int sn{0};
std::sscanf(yytext, "<route %d %d%n", &size, &m_id, &sn);
ss += sn;
for(int i{0}; i<size; ++i)
{
unsigned int u{0u};
std::sscanf(yytext+ss, "%u%n", &u, &sn);
route.push_back(u);
ss += sn;
}
m_cmd = 101;
}
{INIT}{WS}{WORD}{WS}("c"|"g") {

std::sscanf(yytext, "<init %s %c>", name, &role);
```

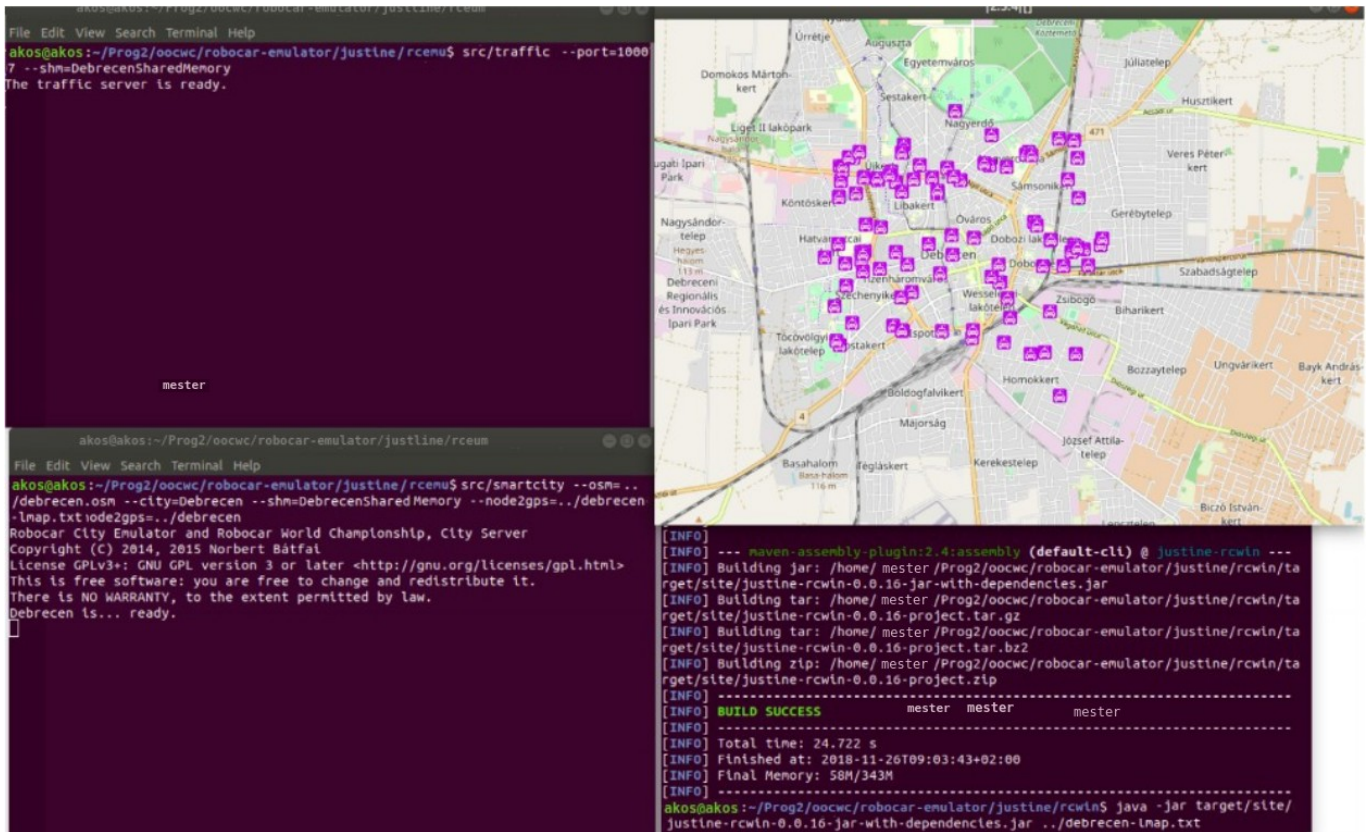
```
num = 1;
m_cmd = 0;
}
{INIT}{WS}{WORD}{WS}{INT}{WS}("c"|"g") {

std::sscanf(yytext, "<init %s %d %c>", name, &num, &role);
if(num >200)
{
m_errnumber = 1;
num = 200;
}
m_cmd = 1;
}
{INITG}{WS}{WORD}{WS}("c"|"g") {

std::sscanf(yytext, "<init guided %s %c>", name, &role);
num = 1;
m_guided = true;
m_cmd = 3;
}
{INITG}{WS}{WORD}{WS}{INT}{WS}("c"|"g") {

std::sscanf(yytext, "<init guided %s %d %c>", name, &num, &role);
if(num >200)
{
m_errnumber = 1;
num = 200;
}
m_guided = true;
m_cmd = 2;
}
. {;}
```

Futás:



### 13.3. I334d1c4

A feladat az volt, hogy készítsünk olyan Java vagy C++ programot, ami leet chiperként működik. A "leet speak" kommunikáció során a betűket a rá hasonló számokkal helyettesítjük.

Például a leet-et ilyen formában a következő képen néz ki: 1337.

Megoldás forrása: <https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloC/L33tSpeak.java>

A program elején csatoljuk a szükséges könyvtárakat.

```
import java.io.*;
import java.util.*;
```

A java.io rendszerbevitelt és -kimenetet biztosít adatfolyamokon, sorosításon és a fájlrendszeren keresztül. Ha a null argumentumot adunk neki, a csomag bármelyik osztályában vagy interfészében egy konstruktor-nak vagy metódusnak, akkor NullPointerException lép ki a program.

A java.util strig tokenizert, random szám generátort és bit tömböt.

Következő lépésben létrehozuk a L33tSpeak osztályt.

```
public class L33tSpeak {
 public static void main(String[] args) throws FileNotFoundException {
 Scanner input = new Scanner(new File("bemenet.txt"));
 PrintStream out = new PrintStream(new File("kimenet.txt"));
```



```
 leetSpeak(input, out);
}
```

Itt adjuk meg a be- és kimeneti állományt, valamint egy hibaüzenetet is létrehozunk arra az esetre, ha a bemeneti állomány nem létezik.

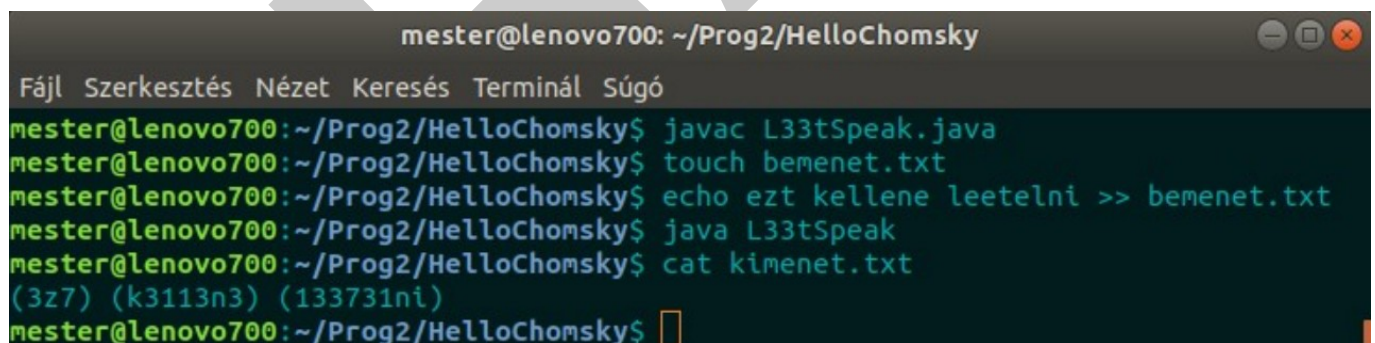
A következő részben vannak megadva a szabályok a leeteléshez, amiket bővíthetünk, módosíthatunk kedvünk szerint. Itt meggy végbe a leetelés és a leetelt szöveg kimenet.txt állományba való kiírása.

```
public static void leetSpeak(Scanner input, PrintStream output) {
 while (input.hasNextLine()) {
 String sor = input.nextLine();
 Scanner tokens = new Scanner(sor);

 while (tokens.hasNext()) {
 String token = tokens.next();
 token = token.replace("a", "4");
 token = token.replace("o", "0");
 token = token.replace("l", "1");
 token = token.replace("e", "3");
 token = token.replace("t", "7");
 if (token.endsWith("s")) {
 token = token.substring(0, token.length() - 1) + "Z";
 }
 output.print("(" + token + ") ");
 }

 output.println();
 }
}
```

Futtatáskor fontos, hogy legyen létrehozva a bemenet.txt fájl a leetelni kívánt szöveggel.



```
mester@lenovo700: ~/Prog2/HelloChomsky
Fájl Szerkesztés Nézet Keresés Terminál Súgó
mester@lenovo700:~/Prog2/HelloChomsky$ javac L33tSpeak.java
mester@lenovo700:~/Prog2/HelloChomsky$ touch bemenet.txt
mester@lenovo700:~/Prog2/HelloChomsky$ echo ezt kellene leetelni >> bemenet.txt
mester@lenovo700:~/Prog2/HelloChomsky$ java L33tSpeak
(3z7) (k3113n3) (133731ni)
mester@lenovo700:~/Prog2/HelloChomsky$ cat kimenet.txt
```

## 13.4. Perceptron osztály

A feladat az volt, hogy dolgozzuk be egy külön projektbe a projekt Perceptron osztályát!

Megoldás forrása: <https://github.com/mesterakos963/Prog2/blob/master/Prog2/HelloC/perceptron.cpp>

A perceptron a mesterséges intelligenciában a neutron megfelelője. Tanulásra képes, a bemeneti 0-ák és 1-ek sorozatából mintákat tanul meg.

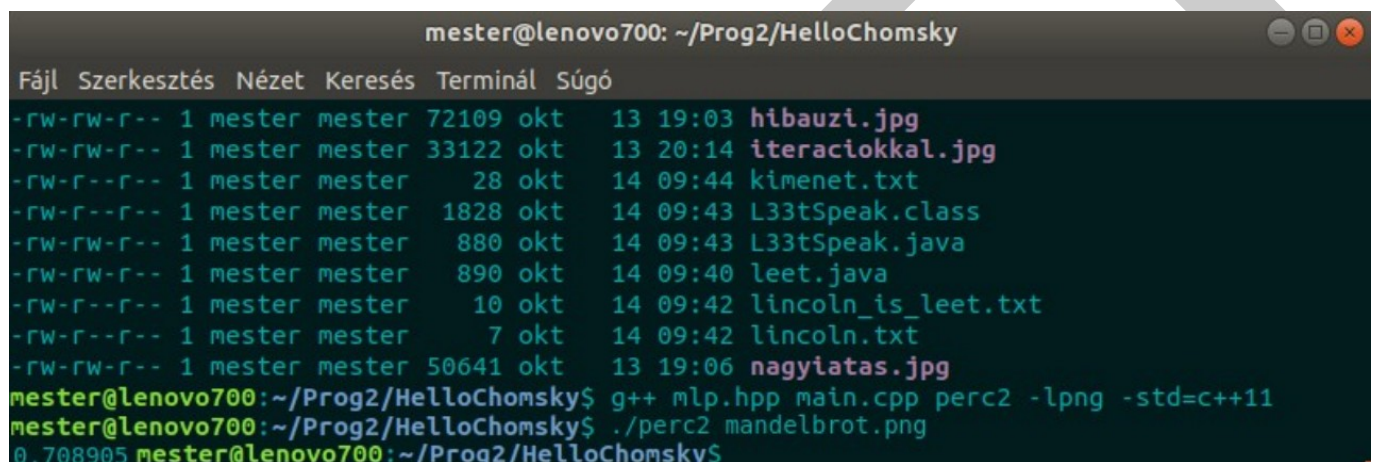


Ebben a feladatban a perceptron kap egy képet bemenetnek (a mandelbrot halmazt ábrázoló képet). Ez lesz a többrétegű neurális háló bemenete.

Mivel többrétegű perceptront akarunk létrehozni, ezért csatolnunk kell az mlp és a png++/png könyvtárakat, valamint utóbbi könyvtár a png-kel való munkát teszi lehetővé.

A main függvényben adjuk meg, hogy az első argumentumban szereplő állományt szeretnénk olvasni. Meghatározzuk a kép méretét valamint ezeket az értékeket el is tároljuk. Létrehozunk egy példányt a perceptron osztályból, melynek argumentumai a rétegek száma, az első és a második réteg neuronjai az inputrétegben. Ezután két for ciklussal végigmegyünk a képen vízszintesen és függőlegesen. A vörös színkomponenst az image-ben tároljuk. A value egy double típusú értéket tárol. Ezt az értéket a perceptron image-re történő meghívása adja majd meg. Kiírjuk az értéket, majd a memória felszabadítása végett töröljük a már használni nem kívánt értékeket.

Fordítás, futtatás:



```
mester@lenovo700: ~/Prog2/HelloChomsky
Fájl Szerkesztés Nézet Keresés Terminál Súgó
-rw-rw-r-- 1 mester mester 72109 okt 13 19:03 hibauzi.jpg
-rw-rw-r-- 1 mester mester 33122 okt 13 20:14 iteraciokkal.jpg
-rw-r--r-- 1 mester mester 28 okt 14 09:44 kimenet.txt
-rw-r--r-- 1 mester mester 1828 okt 14 09:43 L33tSpeak.class
-rw-rw-r-- 1 mester mester 880 okt 14 09:43 L33tSpeak.java
-rw-rw-r-- 1 mester mester 890 okt 14 09:40 leet.java
-rw-r--r-- 1 mester mester 10 okt 14 09:42 lincoln_is_leet.txt
-rw-r--r-- 1 mester mester 7 okt 14 09:42 lincoln.txt
-rw-rw-r-- 1 mester mester 50641 okt 13 19:06 nagyiatas.jpg
mester@lenovo700:~/Prog2/HelloChomsky$ g++ mlp.hpp main.cpp perc2 -lpng -std=c++11
mester@lenovo700:~/Prog2/HelloChomsky$./perc2 mandelbrot.png
0.708905 mester@lenovo700:~/Prog2/HelloChomsky$
```

## 14. fejezet

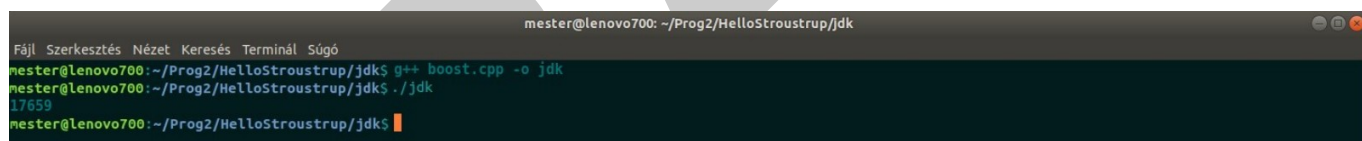
# Helló, Stroustrup!

### 14.1. JDK osztályok

A feladatunk az volt, hogy írjunk olyan Boost C++ programot a fénykart példából kiindulva, amely kilis-tázza a JDK összes osztályát (miután kicsomagoltuk az src.zip állományt, arra ráengedve)!

Ehhez először a `sudo apt-get install libboost-all-dev` paranccsal telepítenünk kell a boost könyvtárat. Ez a könyvtár több dologban is a segítségünkre lesz: ezzel határozzuk meg az src elérési útját és megyünk végig az összes fájlön és almappán, továbbá ezzel határozzuk meg a fájlok kiterjesztését és döntjük el, hogy a vizsgált elem almappa vagy fájl.

Fordítás és futtatás után a következőt kapjuk:



```
mester@lenovo700: ~/Prog2/HelloStroustrup/jdk
mester@lenovo700:~/Prog2/HelloStroustrup/jdk$ g++ boost.cpp -o jdk
mester@lenovo700:~/Prog2/HelloStroustrup/jdk$./jdk
17659
mester@lenovo700:~/Prog2/HelloStroustrup/jdk$
```

A kapott szám az osztályok számát mutatja.

A kód elején csatoljuk a könyvtárakat.

```
#include <boost/filesystem.hpp>
Filesystem declarations
#include <iostream>
using namespace std;
```

Ami számunkra most érdekes, az a `<boost/filesystem.hpp>` könyvtár. Ez az a könyvtár, amit a feladat elején letöltöttünk. Az ott említett feladatokat látja el.

```
int main(int ac, char** av)
{
 string extension;
 int count = 0;
 boost::filesystem::recursive_directory_iterator iterator(string ←- ("/ ←-
 Prog2/HelloStroustrup/src"));
 while(iterator != boost::filesystem:: ←- recursive_directory_iterator())
 {
```

```
string extension = boost::filesystem::extension(iterator-> path().filename());
if(boost::filesystem::is_regular_file(iterator->path()) && extension != ".java")
{
 count++;
}
++iterator;
}
cout << count << endl;
return 0;
}
```

Ezt a main függvény követi, ahol egy egyszerű while ciklussal addig vizsgáljuk át az elemeket, amíg a végére nem érünk, ezzel együtt növelünk egy számlálót, amit a végén kiíratunk std::cout-tal.

## 14.2. Hibásan működő

A feladatunk az volt, hogy készítsünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló: [https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2\\_3.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_3.pdf) (71-73 fólia) által készített titkos szövegen.

Az elrontott RSA nem egyben kódolja a szöveget, hanem karakterenként. Ha betű gyakoriság alapján törünk egy szöveget, az azt jelenti, hogy a miután a törő program megvizsgálta a kódolt szöveget egy, a betűk gyakoriságát tartalmazó táblázat segítségével visszafejti azt.

RSA kódolás - nyílt kulcsú, azaz asszimetrikus titkosító algoritmus. Kitalálója Ron Rivest, Adi Shamir és Len Adleman (1976) (innen kapta a nevét). Napjaink leggyakrabban használt titkosítási eljárása. Kettő kulcs van: egy titkos és egy nyilvános. A nyilvános kulcsra a szöveg titkosításához van szükség, a titkos(privát) kulcsra pedig a dekódoláshoz. Helytelen kulcs használata esetén a kapott eredmény a dekódolás után hibás lesz.

A titkosító kód:

```
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.*;
```

A kód elején importáljuk a használni kívánt osztályokat: BigInteger - nagy számokkal való munkához szükséges (kulcsok); SecureRandom - előállít egy kódoláshoz használt biztonságos számot.

```
public class RSA {
 private final static BigInteger one = new BigInteger("1");
 private final static SecureRandom random = new SecureRandom();

 private BigInteger privateKey;
 private BigInteger publicKey;
 private BigInteger modulus;

 RSA(int N) {
```

```
BigInteger p = BigInteger.probablePrime(N/2, random);
BigInteger q = BigInteger.probablePrime(N/2, random);
BigInteger phi = (p.subtract(one)).multiply(q.subtract(one));
modulus = p.multiply(q);
publicKey = new BigInteger("65537");
privateKey = publicKey.modInverse(phi);
}
```

Az RSA eljárásban létrehozunk 2 probablePrime (valószínűleg prím számot), amit a 'p' és a 'q' változók fognak tárolni. Ezt követően kiszámoljuk a phi moduló értékét.

```
BigInteger encrypt(byte[] bytes) {
 BigInteger swap = new BigInteger(bytes);
 return swap.modPow(publicKey, modulus);
}
```

A fenti encrypt függvény végzi a titkosítást. A byte-ba átváltott karaktert kapja meg, amit majd swap-ban fog eltárolni, mivel a modPow csak BigInteger-en alkalmazható. A titkosítás a visszatérésnél megy végbe.

```
public static void main(String[] args) {
 int N = Integer.parseInt(args[0]);
 RSA key = new RSA(N);
 System.out.println("key = " + key);
 String s = "test";
 byte[] bytes = s.getBytes();
 List<BigInteger> result = new ArrayList<BigInteger>();
 byte[] atmenet = new byte[1];
 for(int i = 0; i < bytes.length; i++)
 {
 atmenet[0] = bytes[i];
 result.add(key.encrypt(atmenet));
 }
 System.out.println("message = " + s);
 System.out.println("encrypted = " + result);
}
```

A main függvényben parancssorból bekérjük az N-t, amivel majd legeneráljuk a kulcsot. Az 's'-be adjuk meg a titkosítani kívánt szöveget byte típusban. Az eredményt eltároljuk a result-ban. A for ciklus megy végig a byte tömbön és a kódolt szöveget folyamatosan fűzzük hozzá a result-hoz, amit majd a program végén kiíratunk.

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
mester@lenovo700:~/Prog2/HelloStroustrup/rsa$ javac RSA.java
mester@lenovo700:~/Prog2/HelloStroustrup/rsa$ java RSA 19
key = RSA@27973e9b
message = ezt kodoltuk
encrypted = [42743, 8695, 6739, 77261, 63157, 36757, 68977, 36757, 65086, 6739, 67840, 63157]
```

## Dekódolás

Ehhez két kódot kell átnéznünk: Kulcspar.java és RsaTores.java

A Kulcspar.java 3 privát változót tartalmaz, valamint metódusokat, hogy ezeket a számokat kívülről is meg tudjuk nézni és felül tudjuk írni.

```
class KulcsPar{
 private String values;
 private char key = '_';
 private int freq = 0;
 public KulcsPar(String str){
 this.values = str;
 }
 public void setValue(String str){
 this.values = str;
 }
 public void setKey(char k){
 this.key = k;
 }
 public String getValue(){
 return this.values;
 }
 public char getKey(){
 return this.key;
 }
 public void incFreq(){
 freq += 1;
 }
 public int getFreq(){
 return freq;
 }
}
```

Ami számunkra érdekesebb, az az RsaTores.java. Elsőként beolvassuk a dekódolandó szöveget a bemenet.txt állományból soronként, közben léptetünk egy számlálót.

```
BufferedReader inputStream = new BufferedReader(new FileReader("bemenet ←
 ←-.txt"));
int lines = 0;
String line[] = new String[10000];
while((line[lines] = inputStream.readLine()) != null) {
 lines++;
}
inputStream.close();
```

Ezt követően létrehozunk egy 'kp' tömböt, amibe az első beolvasott sort fogjuk tárolni, aztán létrehozunk egy 'db' változót 1 kezdőértékkel. Megyünk végig a sorokon és megszámláljuk, hogy melyik betűből hány van.

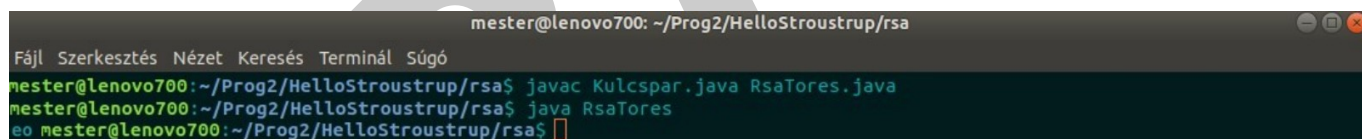
```
kp[0] = new KulcsPar(line[0]);
int db = 1;
for(int i = 1; i < lines; i++) {
 volt = false;
 for(int j = 0; j < db; j++) {
 if(kp[j].getValue().equals(line[i])) {
 kp[j].incFreq();
 }
 }
}
```

```
 volt = true;
 break;
 }
}
if(volt == false) {
 kp[db] = new KulcsPar(line[i]);
 db++;
}
}
```

Miután megszámoztuk a karaktereket, ezeket csökkenő sorrendbe rendezzük. Rendezés után beolvassuk az "angol.txt" állományt és tároljuk a "key" tömbben. Ez az állomány tartalmazza, hogy a karakterek gyakoriságát a szövegben.

```
for(int i = 0; i < kdb && kp[i] != null; i++) {
 kp[i].setKey(key[i]);
}
for(int i = 0; i < lines; i++) {
 for(int j = 0; j < db; j++) {
 if(line[i].equals(kp[j].getValue())) {
 System.out.print(kp[j].getKey());
 }
 }
}
```

A program végén párosítjuk a kódolt szöveget az angol.txt állománnyal. Itt történik a dekódolás, majd az eredményt kiíratjuk.



```
mester@lenovo700: ~/Prog2/HelloStroustrup/rsa
Fájl Szerkesztés Nézet Keresés Terminál Súgó
mester@lenovo700:~/Prog2/HelloStroustrup/rsa$ javac Kulcspar.java RsaTores.java
mester@lenovo700:~/Prog2/HelloStroustrup/rsa$ java RsaTores
eo mester@lenovo700:~/Prog2/HelloStroustrup/rsa$
```

### 14.3. String osztály és másoló-mozgató szemantika

A feladat egy egyszerű string osztály elkészítése volt, hogy ne a verembe tegyük a kockászárójel megvalósítását, mert az nyilván odacsap az osztály értelmezésének.

```
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;
```

A kód elején csatoljuk a könyvtárakat.

<iostream> - standard in- és output használatához szükséges.

<cstring> - karakterláncok kezeléséhez szükséges.

<algorithm> - olyan függvényeket tartalmaz, mint például: keresés, rendezés, számolás stb.

Ez egy string osztály definiálása, ami egy heap-en foglalt memóriaterületre mutató mutatót tartalmaz. A heap allokáció jelentése, hogy futás közben foglaljuk le a tárhelyet, ellentétben a stack allokációval.

```
class mystring
{
 char* data;
public:
 char* Data() const {return data;}
 mystring(const char* p)
 {
 cout<<"konstruktor"<<endl;
 size_t size = strlen(p) + 1;
 data = new char[size];
 memcpy(data, p, size);
 }
}
```

A string számára lefoglalt memóriaterület első karakterére mutató pointer.

```
class mystring
{
 char* data;
public:
 char* Data() const {return data;}
}
```

A char típusú privát tag elérésére szolgáló függvény.

```
mystring(const char* p)
{
 cout<<"konstruktor"<<endl;
 size_t size = strlen(p) + 1;
 data = new char[size];
 memcpy(data, p, size);
}
```

Mi foglaljuk le a memóriát, ezért nekünk kell kiüríteni azt, hogy elkerüljük a memóriefolyást. A delete[]-tel jelezzük, hogy a tömböt szeretnénk felszabadítani.

```
~mystring()
{
 delete[] data;
}
```

A másoló konstruktor adja meg, hogy mit jelent a mystring típusú objektum. C++11-ben lehetőségünk van detektálni az rvalue(jobbérték) argumentumokat. Ez egy olyan ideiglenes objektum, ami az őket tartalmazó kifejezés végén megsemmisül.

Konstruktor csinálunk ilyen rvalue paraméterrel. Ha ilyen jobbértékről van szó a futás közben, akkor a konstruktor fog meghívódni.

```
mystring(const mystring& regi)
{
 cout<<"másoló konstruktor"<<endl;
}
```

```
size_t size = strlen(regi.data) + 1;
data = new char[size];
memcpy(data, regi.data, size);
}
```

A másoló értékadásnál túlterheljük az '=' operátort és a swap funkcióval felcseréljük a két pointert.

```
mystring& operator=(const mystring& regi)
{
 cout<<"másoló értékadás"<<endl;
 delete[] data;
 size_t size = strlen(regi.data) + 1;
 data = new char[size];
 memcpy(data, regi.data, size);
 return *this;
}
```

Az itt következő részben nem végzünk másolást, csupán mozgatót amódon, hogy a heap-en lévő adat másolása helyett csupán a pointert másoljuk és állítjuk az értékét null-ra. Így nem fog lefutni a destruktork.

```
mystring (mystring&& regi)
{
 cout<<"mozgató konstruktor"<<endl;
 data = regi.data;
 regi.data = nullptr;
}
```

Operátor túlterhelés segítségével érjük el, hogy ki tudjuk iratni az objektumunk tartalmát.

A "\*this" az aktuális objektumra mutat rá. A move funkció kikényszerítve a mozgató szemantikát a bal értékből jobb értéket készít (lvalue --> rvalue)

```
mystring& operator=(mystring&& regi)
{
 cout<<"mozgató értékadás"<<endl;
 swap(data, regi.data);
 return *this;
}
```

Túlterheljük a [] operátort a const kulcsszó használata nélkül, így a funkció csak konstans objektumokra fog meghívódni.

```
char operator[](int i)
{
 size_t size = strlen(data) - 1;
 if(i > size)
 {
 cout << "Out of range" <<endl;
 }
 return data [i];
}
};
```



A teljes forrás elérhető itt: <https://github.com/mesterakos963/Prog2/blob/master/Prog2/String%20class/string.cpp>

## 14.4. ESSZÉ: RSA titkosító eljárás

Az RSA-eljárás, mint ahogy azt már egy korábbi feladatban is leírtam, egy nyílt kulcsú titkosító algoritmus, napjaink leggyakrabban használt titkosítási eljárása. Kitalálásának éve: 1976. Alkotói (egyben a nevét is innen kapta): Ron Rivest, Adi Shamir és Len Adleman. Alapját a moduláris és prímszámelmélet bizonyos tételei jelentik.

### MATEMATIKAI HÁTTÉR:

Legyen:

- $p, q \rightarrow$  nagy prímek;
- $N = p * q$ ;
- $(t, \phi(N)) = 1$ ;
- $T, M \rightarrow$  kulcspár;
- $T(r) = r^t$  legkisebb pozitív maradéka;
- $M(s) = s^m$  legkisebb pozitív maradéka;
- $r, s = 1, 2, \dots, N$ ;
- $m * t = 1 + k * \phi(N)$ ;
- $N, t, T \rightarrow$  nyilvános;
- $p, q, m, M \rightarrow$  titkos;
- $M = T^{-1}$ ,  $M$  a  $T$  ismeretében sem határozható meg.

A ' $p$ ' és ' $q$ ' számokat a tulajdonos kapja meg prímteszteléssel. Ezek segítségével határozza meg ' $m$ '-et. Az " $M(s)$ " függvényt csak a kulcsok tulajdonosa számolhatja ki, szemben a " $T(r)$ " függvénnyel, amit bárki.

### MŰKÖDÉSE:

A titkosításhoz szükség van egy nyílt és egy titkos kulcsra. A nyílt kulcs mindenki számára ismert, így bárki tud kódolni, azonban a dekódoláshoz már szükség van a titkos kulcsra, amit már csak az birtokol, aki majd dekódolni fogja a az "üzenetet".

Ahhoz, hogy biztosra menjünk a kódolás biztonságát illetően, biztossá kell tennünk, hogy tényleg a feladattól kaptuk az üzenetet és nem valaki más küldte az ő nevében.

Ehhez nézzük meg az alábbi példát:

Buda Pest egy nyilvános kulcsát használja, hogy eljuttasson egy titkosított üzenetet, amiben bizonyítani próbálja, hogy ő valójában Buda, nem pedig valaki más. Mivel a nyilvános kulcsot bárki használhatja,

ezért Pest egyáltalán nem lehet biztos abban, hogy az üzenet Budától jött. Szerzői kiletünket RSA szintű biztonsággal tanúsíthatjuk, így az RSA már valódi nyilvános kulcsú titkosító eljárássá nővi ki magát.

Tehát Buda titkosított üzenetet szeretne küldeni Pestnek. Az üzenetből Buda kivág egy részletet, aminek meghatározza az ASCII értékét, majd ez az értéket 'd'-edik hatványra emeli, ezután a kapott számot modulo 'N'. A végeredményt, mint egy aláírást csatolja az üzenethez. Így Pest számára már biztossá vált, hogy valóban Buda küldte az üzenetet.

#### DE HOGYAN DEKÓDOLJA PEST AZ ALÁÍRÁST?

Ehhez Pestnek az aláírást felemeli 'e' kitevőjére és veszi modulo 'N' értékét. Mivel ez a két művelet, amit Buda elvégzett az aláírás kódolásakor és amit Pest végez dekódoláskor egymás inverzei, ezért az eredményként megkapott üzenetet összeveti az üzenetben szereplő szövegrészlettel és ha egyezést talál, akkor Pest már 100%-ban biztos lehet abban, hogy az üzenet Budától származik, mivel a titkos kucsl az ő birtokában volt.

#### MÉGIS MENNYIRA BIZTONSÁGOS EZ AZ ELJÁRÁS?

A jelenlegi matematikai ismeretek szerint egy alaposan RSA kódolt üzenet nem fejthető vissza számelméleti okok miatt belátható időn belül, azonban nincs bizonyítva az, hogy lehetetlen olyan algoritmust kifejleszteni, amivel már kellőképpen gyorsan lehetne dekódolni az üzenetet, hogy érdemes legyen megpróbálni.

Az eljárás alapját az adja, hogy egy kellően nagy prímszámról nehéz megállapítani a prímtényezőit, főleg ha ez a szám két nagy prímszám szorzásának eredménye.

1994-ben jelent meg Peter Shor publikációja, mely rámutat arra, hogy egy kvantumszámítógép elméletileg végre tudja hajtani a faktORIZÁCIÓT polinom időn belül. Ha ez tényleg megtörténne a gyakorlatban is, az az RSA kódolás teljes elavulását jelentené.

## 15. fejezet

# Helló, Gödel!

### 15.1. Gengszterek

Feladatunk a gengszterek rendezése lambdával a Robotautó világbajnokságban.

Ez a feladat a C++11-ben bevezetett `std::sort()` függvényt helyezi középpontba.

Lássuk a kódrészletet:

```
std::sort (gangsters.begin(), gangsters.end(), [this, cop] (←
 Gangster x, Gangster y)
 {
 return dst (cop, x.to) < dst (cop, y.to);
 });
```

Ez az `std::sort()` függvénynek több verziója is van: 2 és 3 paraméteres. Az első kettőben a tartományt adjuk meg, amelyet rendezni szeretnénk. A mi esetünkben az összes gengsztert szeretnénk, tehát ez a `gangsters.begin()` és a `gangsters.end()` függvénnyel adjuk meg, ami megadja az első és utolsó elem indexét. A harmadik paraméter opcionális, egy rendezés alapjául szolgáló függvény. Ennek hiánya esetén a beépített függvény dönti el, hogy mi alapján megy majd végbe a rendezés.

A C++ 11 óta támogatott szintaxis a következő:

```
[](paraméter1, paraméter2, paraméter3)-> visszatérési ←
 típusa {utasítások}
```

A `[]` jelek közé írt változókat a függvényen kívülről is el tudjuk érni. A mi esetünkben a `cop` objektumra, illetve az aktuális objektum mutatójára lesz szükség. Paraméterként megadunk két `Gangster` osztályú objektumot, melyek a `sort` által összehasonlított elemek lesznek. Azt vizsgáljuk, hogy melyik gengszter van közelebb a rendőrhöz és aszerint rendezzük növekvő sorrendbe őket, majd visszatérítünk egy `bool` értéket.

### 15.2. Custom allokátor

Megoldás forrása: <https://github.com/mesterakos963/Prog2/blob/master/HelloG%C3%B6del/customalloc.cpp>

Egy objektum példányosításakor a memóriában lefoglalódik a szükséges terület. C++-ban ezt a new operátor segítségével tehetjük meg. Ez meghívja az alapértelmezett allokatort. Tökéletesen ellát minden feladatot, azonban ha kicsit jobban bele szeretnénk avatkozni a memóriakezelésbe, akkor saját allokatort kell készítenünk.

Létrehozunk egy CustomAlloc osztályt. Ez egy template osztály lesz, tehát bármilyen típussal használható.

```
template<typename T>
struct CustomAlloc {

}
```

A using kulcsszóval megadjuk, hogy mit milyen néven szeretnénk használni ezután. Láthatjuk, hogy a T\* pointert fogja jelenteni, a T& a referenciát stb.

```
using size_type = size_t;
using value_type = T;
using pointer = T*;
using const_pointer = const T*;
using reference = T&;
using const_reference = const T&;
using difference_type = ptrdiff_t;
```

Az allocate nevű funkció egy pointert ad vissza. Paramétere size\_type típusú. Ez fogja a szükséges memóriaterületet lefoglalni.

```
pointer allocate (size_type n) {
 int s;
 char* p=abi::__cxa_demangle (typeid (T).name(),0,0, &s);
 std::cout << "Allocating "
 << n << " objects of "
 << n*sizeof (T)
 << " bytes. "
 << typeid (T).name() << "=" << p
 << std::endl;
 free (p);
 return reinterpret_cast<T*> (
 new char[n*sizeof (T)]);
}
void deallocate (pointer p, size_type n) {
 delete[] reinterpret_cast<char *> (p);
 std::cout << "Deallocating "
 << n << " objects of "
 << n*sizeof (T)
 << " bytes. "
 << typeid (T).name() << "=" << p
 << std::endl;
}
```

Az újdonságnak számí a \_\_cxa\_demangle funkció. Ez a fordító által kódolt funkció és változónevek visszafejtéséhez szükséges. Első paramétere a visszafejtendő név, második az output buffer, aztán a hossz, majd a státusz. Ezt eltávolítjuk az integer típusú 's' változónkba.

Ezután kiiratjuk, hogy hány objektumnak foglalunk helyet és ezt hány bájton tesszük meg. A 'p'-ben tároljuk el a typeid(T).name() által visszafejtett típus típusazonosítóját. Ezután felszabadítjuk a 'p'-t és visszatérünk a lefoglalt területre mutató T típusú mutatóval.

A deallocate szabadítja fel a lefoglalt memória területet.

Most jön a main() függvény:

```
int main(){
std::vector<int, CustomAlloc<int>> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
v.push_back(5);
v.push_back(6);
v.push_back(7);
v.push_back(8);
v.push_back(9);
v.push_back(10);
v.push_back(11);
v.push_back(12);
v.push_back(13);
v.push_back(14);
v.push_back(15);
for(int x : v){
 std::cout << x << std::endl;
}
return 0;
}
```

Létrehozunk egy integer elemeket tartalmazó vektort és megadjuk, hogy memóriefoglalásnál a mi allokátorunkat szeretnénk használni. A v.push\_back() függvénnyel hozzáfűzünk 15 db integert és végigmegyünk a vektoron egy for ciklussal.

Fordítás, futtatás:

```
mester@lenovo700: ~/Prog2/HelloGödel
Fájl Szerkesztés Nézet Keresés Terminál Súgó
mester@lenovo700:~/Prog2/HelloGödel$ g++ customalloc.cpp -o ca
mester@lenovo700:~/Prog2/HelloGödel$./ca
Allocating 1 objects of 5 bytes. i=int
Allocating 2 objects of 6 bytes. i=int
Deallocating 1 objects of 5 bytes. i=0x5605caa0ee70
Allocating 4 objects of 8 bytes. i=int
Deallocating 2 objects of 6 bytes. i=0x5605caa0f2a0
Allocating 8 objects of 12 bytes. i=int
Deallocating 4 objects of 8 bytes. i=0x5605caa0ee70
Allocating 16 objects of 20 bytes. i=int
Deallocating 8 objects of 12 bytes. i=0x5605caa0f2c0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Deallocating 16 objects of 20 bytes. i=0x5605caa0f2f0
mester@lenovo700:~/Prog2/HelloGödel$
```

### 15.3. STL map érték szerinti rendezése

A Standard Template Library (STL) egy tárolókat és algoritmusokat tartalmazó könyvtár. Ebben található meg például a vector, list, stack és a map, amiről ebben a feladatban szó lesz.

A map asszociatív tároló, a benne tárolt értéken kívül rendelkezik egy kulcs értékkel is, ami szerint a benne tárolt adatok növekvő sorrendben vannak rendezve.

Nézzük meg ennek a felhasználását a Tanár úr által csatolt példában, ami az alábbi linken érhető el:

<https://github.com/nbatfai/future/blob/master/cs/F9F2/fenykard.cpp#L180>

A kódcsipet:

```
std::vector<std::pair<std::string, int>> sort_map (std::map <std::string, ↵
 int> &rank)
{
 std::vector<std::pair<std::string, int>> ordered;

 for (auto & i : rank) {
 if (i.second) {
```

```
 std::pair<std::string, int> p {i.first, i.second};
 ordered.push_back (p);
 }

 std::sort (
 std::begin (ordered), std::end (ordered),
 [=] (auto && p1, auto && p2) {
 return p1.second > p2.second;
 }
);

 return ordered;
}
```

A függvény neve `sort_map`, visszatérési értéke egy `string-int` párokat tartalmazó vektor. Ehhez az `std::pair` fogjuk használni, amiben különböző elemekből alkotott párokat lehet tárolni. Paraméterként a függvény megkapja az `std::map` referenciát.

Létrehozunk egy üres vektort, ami `string` párokat fog tartalmazni, ez lesz később a visszatérési érték. Ezután egy `for` ciklussal végigjárjuk a `rank` nevű `map`-et és ellenőrizzük, hogy van-e már második értékpár. Ha van, akkor létrehozunk egy `std::pair` adatstruktúrát és a létrehozott párokat tároljuk az `ordered` vektorba.

Meghívjuk az `std::sort` függvényt az `ordered` vektorra és rendezzük az elejétől a végéig. Az első feladatban használt `lambda` kifejezés segítségével megadjuk, hogy miképpen szeretnénk összehasonlítani az elemeket. Paraméterei kettő `auto` típusú jobbérték, visszatérési értéke `bool`, aminek értékét a `p1.second` és `p2.second` összehasonlítása adja meg.

A `[ = ]` jelentése, hogy a `lambda` kifejezés másolás útján fogja átvenni a változókat, nem referenciaként.

A végén visszaadjuk a rendezett vektort. (`return ordered`).

## 15.4. Alternatív Tabella rendezése

Feladatunk hogy mutassuk be a [https://progpater.blog.hu/2011/03/11/alternativ\\_tabella](https://progpater.blog.hu/2011/03/11/alternativ_tabella) a programban a `java.lang` `Comparable<T>` szerepét!

Kezdjük ott, hogy mi az az alternatív tabella:

Az alternatív tabella a futball bajnokságoknál használt tabellával (tehát ahol a győzelemért 3, döntetlenért 1, vereségért pedig 0 pont jár) szemben azt veszi figyelembe sorbarendezeznél, hogy egy adott csapat melyik másik csapattal szemben érte el az adott eredményt. Normál tabellánál a leggyengébb csapat elleni győzelem ugyanúgy 3 pontot ér, mint a legerősebb elleni. Az alternatív tabella erre kínál alternatív megoldást.

A `Java Comparable` interfészt a felhasználó által megadott típusok rendezésére használjuk.

`T -->` az objektum típusa, amihez hasonlítani szeretnénk az interfészt implementáló objektumot. Ez a `java.lang` csomagban található meg és csupán a `compareTo(Object)` metódust tartalmazza.

Hozzunk létre egy osztályt egy csapatra és implementáljuk a `Comparable` interfészt rá.

```
class Csapat implements Comparable<Csapat> {
 protected String nev;
 protected double ertekek;
 public Csapat(String nev, double ertekek) {
 this.nev = nev;
 this.ertekek = ertekek;
 }
 public int compareTo(Csapat csapat) {
 if (this.ertekek < csapat.ertekek) {
 return -1;
 } else if (this.ertekek > csapat.ertekek) {
 return 1;
 } else {
 return 0;
 }
 }
}
```

Az osztály konstruktora adja meg a név és érték változókat. Definiáljuk az interfész fentebb is említett `compareTo()` metódusát. Ha a paraméterként megadott objektumban szereplő érték változó kisebb, mint a hasonlítandó objektum érték változója, akkor visszatérési értéknek -1-et kapunk, ha nagyobb 1-et. Ha az értékek egyenlők, akkor a visszatérési érték 0 lesz. Így határozzuk meg az osztályból példányosított objektumok sorbarendezési módját.



## 16. fejezet

# Helló, !

### 16.1. OOCWC Boost ASIO hálózatkezelése

A feladatunk az volt, hogy mutassunk rá a scanf szerepére és használatára! <https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/carlexer.ll>

Ez a feladat egy input stream adatainak feldolgozását mutatja be, a mintaprogramunk pedig a már ismert OOCWC projekt, annak is a már korábban boncolgatott lexer része.

Megoldás forrása: <https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/carlexer.ll>

Az sscanf formázott karaktersorozatból olvas be. Beolvasáshoz szükség van egy feltételre a while ciklusban, mivel nem tudjuk, hogy hány gangster adatot olvasunk majd be, így ellenőrizni kell az adatok validságát.

A várt formula:

```
"<OK %d %u %u %u>%n"
```

Az validság ellenőrzésére az sscanf visszatérési értékét használjuk, ami a beolvasott paraméterek számát jelöli. Jelen esetben ez 4. A ciklus a következő képen néz ki:

```
while (std::sscanf (data+nn, "<OK %d %u %u %u>%n", &idd, &f, &t, &s, &n) ←
 == 4)
{
 nn += n;
 gangsters.push_back (Gangster {idd, f, t, s});
}
```

Ha a visszatérési értékünk 4, az azt jelenti, hogy egy gengszter minden adatát beolvastuk, az adatok validak voltak és létrehoztunk egy új gangster objektumot, amit hozzáfűzünk a "gangsters" vektorhoz. Az 'nn' változó a beolvasott adatok számát tárolja. Erre azért van szükség, hogy a "data" karakter tömbből ne olvassuk be kétszer ugyanazt az adatot.

## 16.2. SamuCam

A feladat az volt, hogy mutassunk rá a webcam (pl. Androidos mobilod) kezelésére ebben a projektben: <https://github.com/nbatfai/SamuCam>.

A SamuCam webkameráról olvas le arcokat.

Letöltéshez a következő parancsra lesz szükség:

```
git clone https://github.com/nbatfai/SamuCam.git
```

A letöltött mappába a belépve:

```
wget https://github.com/Itseez/opencv/raw/master/data/lbpcascades/ ←-
lbpcascade_frontalface.xml
```

Erre azért van szükség, mivel a program alapja az arcfelismerés, amit a `cv::CascadeClassifier` osztály tesz lehetővé. Ezen a fájlon keresztül adjuk meg neki, hogy milyen felismerést szeretnénk használni. Az emberi arcot a "faceClassifier" objektum segítségével ismeri fel a kamera képén, amit a képernyőn meg is jelenít.

Qt-s mappából:

```
~/Qt/5.9.8/gcc_64/bin/qmake SamuLife.pro
```

Ezután már csak a "make" parancsot kell használnunk és indítható is a program.

Ahhoz, hogy hozzá tudjunk férni a kameránk által biztosított adatokhoz, a `cv::VideoCapture` osztály van segítségünkre. Ahhoz, hogy megnyissuk a webkamera servere által biztosított felvételt, a `SamuCam::openVideoStream()` függvényt használjuk.

A kameránk által biztosított adatok hozzáféréséhez a `cv::VideoCapture` osztályra lesz szükség. A `SamuCam::openVideoStream()` függvénnyel pedig a webkamera servere által biztosított felvétel megnyitásához lesz szükség.

```
void SamuCam::openVideoStream()
{
 videoCapture.open (videoStream);
 videoCapture.set (CV_CAP_PROP_FRAME_WIDTH, width);
 videoCapture.set (CV_CAP_PROP_FRAME_HEIGHT, height);
 videoCapture.set (CV_CAP_PROP_FPS, 10);
}
```

Az `open` függvény paraméterként megkaphatja a kameránk számát (alapértelmezett: 0), videó fájlt vagy (ahogy a mi példánkban is) IP címet is. A "videoCapture" objektum tulajdonságait a "set" függvénnyel adjuk meg. A mi példánkban a méret és az FPS szám van megadva.

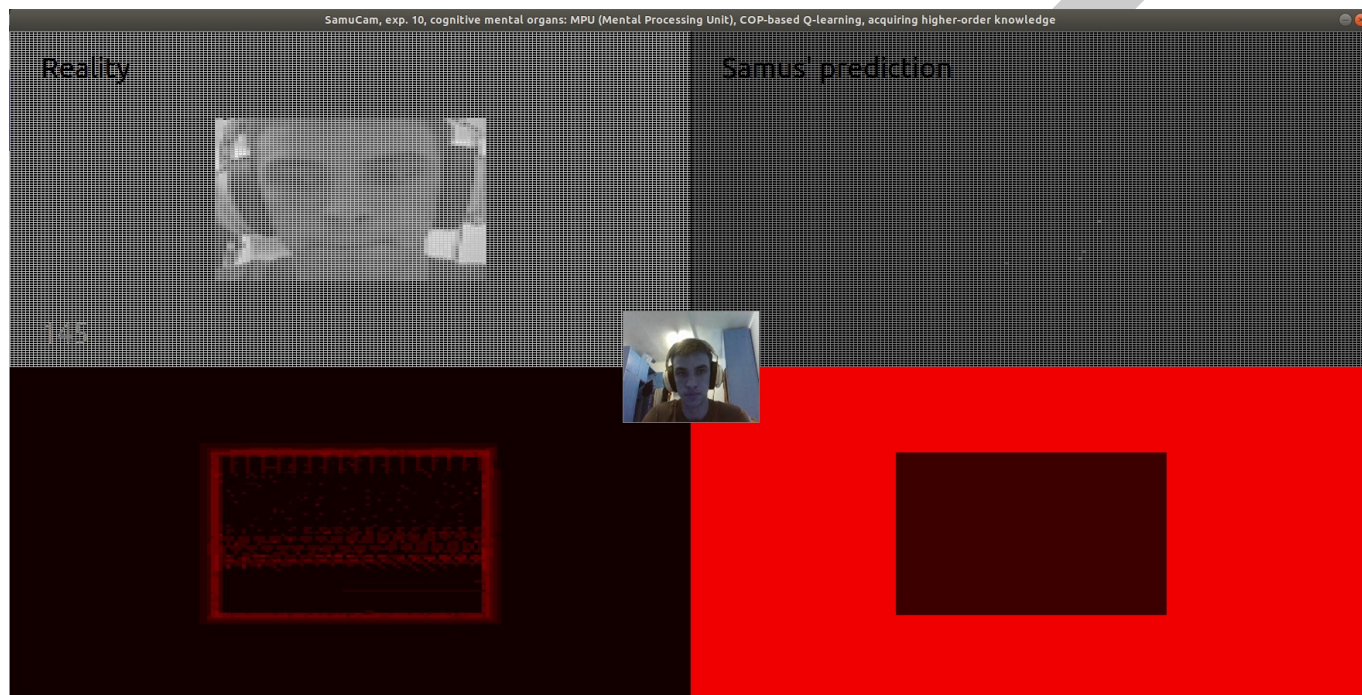
Az arcfelismerés kezelését a "SamuCam" osztály "run" függvénye végzi.

```
void SamuCam::run()
{
 cv::CascadeClassifier faceClassifier;
 std::string faceXML = "lbpcascade_frontalface.xml"; // https://github. ←
 com/Itseez/opencv/tree/master/data/lbpcascades
 if (!faceClassifier.load (faceXML))
```

```
{
 qDebug() << "error: cannot found" << faceXML.c_str();
 return;
}
cv::Mat frame;
while (videoCapture.isOpened())
{
 QThread::msleep (50);
 while (videoCapture.read (frame))
 {
 if (!frame.empty())
 {
 cv::resize (frame, frame, cv::Size (176, 144), 0, 0, cv::↵
 ::INTER_CUBIC);
 std::vector<cv::Rect> faces;
 cv::Mat grayFrame;
 cv::cvtColor (frame, grayFrame, cv::COLOR_BGR2GRAY);
 cv::equalizeHist (grayFrame, grayFrame);
 faceClassifier.detectMultiScale (grayFrame, faces, 1.1, 4, ↵
 0, cv::Size (60, 60));
 if (faces.size() > 0)
 {
 cv::Mat onlyFace = frame (faces[0]).clone();
 QImage* face = new QImage (onlyFace.data,
 onlyFace.cols,
 onlyFace.rows,
 onlyFace.step,
 QImage::Format_RGB888);
 cv::Point x (faces[0].x-1, faces[0].y-1);
 cv::Point y (faces[0].x + faces[0].width+2, faces[0].y ↵
 + faces[0].height+2);
 cv::rectangle (frame, x, y, cv::Scalar (240, 230, 200 ↵
));
 emit faceChanged (face);
 }
 QImage* webcam = new QImage (frame.data,
 frame.cols,
 frame.rows,
 frame.step,
 QImage::Format_RGB888);
 emit webcamChanged (webcam);
 }
 QThread::msleep (80);
 }
 if (! videoCapture.isOpened())
 {
 openVideoStream();
 }
}
}
```

Számunkra a "while" ciklus a lényeg a fenti kódcsipetből, ami addig ismétlődik, amíg a kamera nyitva van. A cv::Mat osztályú objektum segítségével a képkockákat eltároljuk többdimenziós tömbben. Beolvassuk a "frame" tartalmát. Ha ez üres, akkor beolvassuk a kamera képéről egy arcot. A "detectMultiScale" függvénnyel különböző méretű objektumokat tudunk felismerni, amik a mi esetünkben arcok lesznek. Az eredményt egy vektorban tárolja négyzetek formájában. Az első felismert objektumot kiolvassuk, képpé alakítjuk, majd jelezzük, hogy megváltozott a kamera képe.

Futás közben:



## 16.3. BrainB

A feladatunk, hogy mutassuk be a Qt slot-signal mechanizmust ebben a projektben: <https://github.com/nbatfai/esport-talent-search>

A BrainB egy már jól ismert projekt, aminek a célja a tehetségkutatás az esportban. Az játékokban előforduló karakter elvesztést szimulálja. A program 10 percig fut, ezidő alatt a feladatunk, hogy a bal egérgombot lenyomva Samu Entropyn tartssuk az egeret. A futás végeztével megtekinthetjük a statisztikákat.

A használathoz a következő parancsok szükségesek:

```
sudo apt-get install libqt4-dev
sudo apt-get install opencv-data
sudo apt-get install libopencv-dev
```

A forráskódot tartalmazó mappában kiadjuk a következő parancsot:

```
~/Qt/5.12.2/gcc_64/bin/qmake -project
```

Ez létrehozza a project fájlt, amire kiadjuk a következő parancsot:

```
~/Qt/5.12.2/gcc_64/bin/qmake BrainB.pro
```

Ezután ha kiadunk egy "make" parancsot, már csak a futtatni kell a programot.

```
./BrainB
```

Nézzük a slot-signal mechanizmust:

Qt-ben a slotokat és signalokat az objektumok közötti kommunikációra használjuk. GUI programozásnál ha egy elem megváltozik, akkor erről a többi elemnek értesülnie kell. Például ha megnyomjuk a bezár gombot, akkor hívódjon meg az ablak close() funkciója. A Qt ehhez használja a slot-signal mechanizmust. Egy adott esemény bekövetkeztekor egy signal hajtódik végre. Számos előre definiált signal van. A slot egy funkció, ami a megfelelő signal jelzésére hajtódik végre.

Signal - olyan függvény, ami csak deklarálva van, nincs definiálva, nincs visszatérési értéke, azonban vannak paraméterei.

Slot - függvényre hasonlítanak, definiáltak, vannak paraméterei, meghívhatóak, viszont nincs visszatérési értékük.

A kompatibilis signal-slot párok összekapcsolhatóak. Ahhoz, hogy kompatibilisek legyenek a paraméterei típusának kell megegyeznie vagy a slotnak kell paraméter nélkülinek lennie (ilyenkor a signal paraméterei átadhatóak).

A slot-signal mechanizmus a BrainB programban:

```
BrainBWin::BrainBWin (int w, int h, QWidget *parent) : QMainWindow (←←
 parent)
{
// setWindowTitle(appName + " " + appVersion);
// setFixedSize(QSize(w, h));
 statDir = appName + " " + appVersion + " - " + QDate::currentDate() ←←
 .toString() + QString::number (QDateTime:: ←←
 currentMSecsSinceEpoch());
 brainBThread = new BrainBThread (w, h - yshift);
 brainBThread->start();
 connect (brainBThread, SIGNAL (heroesChanged (QImage, int, int) ←←
),
 this, SLOT (updateHeroes (QImage, int, int)));
 connect (brainBThread, SIGNAL (endAndStats (int)),
 this, SLOT (endAndStats (int)));
}
```

Ebből a példából számunkra a connect funkció lényeges. Ennek a segítségével kötjük össze a signalokat a slotokkal. A paraméterek sorrendben: signal-t küldő objektum; signal, amit kezelni szeretnénk; signal-t kezelő objektumra mutató pointer; slot, ami a kiváltott signal-t fogja kezelni.

A lenti példában láthatjuk, hogyha a "brainBThread" objektum "heroesChanged" signal-ja kiváltódik, akkor a "BrainBWin" objektum kezelje "updateHeroes" slottal.

```
connect (brainBThread, SIGNAL (heroesChanged (QImage, int, int) ←←
),
 this, SLOT (updateHeroes (QImage, int, int)));
```

Futás közben:



## 16.4. FUTURE tevékenység editor

A feladatunk az volt, hogy javítsunk valamit az ActivityEditor.java JavaFX programon.

Program forrása: <https://github.com/nbatfai/future/tree/master/cs/F6>

A programban alaphoz a jobb klikk lenyomásával új tevékenység vagy props fájl jön létre a könyvtárszerkezetben. Ennek elkerülése érdekében erre a folyamatra egy új billentyűkombinációt implementáltam.

```
javafx.scene.Scene scene = FileTree.this.getScene();

KeyCombination kcl = new KeyCodeCombination(KeyCode.N, KeyCombination. ←
 ←-
 CONTROL_DOWN);
Runnable rn1 = ()-> {
 System.out.println("Ctrl N");
 javafx.scene.control.TreeItem<java.io.File> item = FileTree.this. ←
 ←-
 getSelectionModel().getSelectedItem();
 System.out.println(item.getValue());
 java.io.File file =item.getValue();
 java.io.File f = new java.io.File(file.getPath() + System. ←
 getProperty("file ←-
 .separator") + "Új altevékenység");
 if (f.mkdir())
 {
 javafx.scene.control.TreeItem<java.io.File> newAct
```

```
 = new FileTreeItem(f, new javafx.scene.image.ImageView(↵
 actIcon));
 item.getChildren().add(newAct);
 } else {
 System.err.println("Cannot create " + f.getPath());
 }
};
scene.getAccelerators().put(kc1, rn1);
```

Új tevékenységet hozunk létre a CTR+N kombinációra. Ez egy új mappát hoz létre a könyvtárszerkezetben. Ezzel gyakorlatilag megkapta a jobb klikk egyik funkcióját, amit a feladat elején is említettem, viszont a jobb klikkel új props fájlt is lehetett létrehozni. Ehhez szintén implementálunk egy billentyűkombinációt.

```
KeyCombination kc2 = new KeyCodeCombination(KeyCode.M, KeyCombination. ↵
 ↵-
 CONTROL_DOWN);
Runnable rn2 = ()-> {
 System.out.println("Ctrl M");
 javafx.scene.control.TreeItem<java.io.File> item = FileTree.this. ↵
 ↵-
 getSelectionModel().getSelectedItem();
 System.out.println(item.getValue());
 java.io.File file = getTreeItem().getValue();
 java.io.File f = new java.io.File(file.getPath() + System. ↵
 getProperty("file ↵-
 separator") + "Új altevékenység tulajdonságok");
 try {
 f.createNewFile();
 } catch (java.io.IOException e) {
 System.err.println(e.getMessage());
 }
 javafx.scene.control.TreeItem<java.io.File> newProps
 = new FileTreeItem(f, new javafx.scene.image.ImageView(↵
 actpropsIcon));
 item.getChildren().add(newProps);
};
scene.getAccelerators().put(kc2, rn2);
```

Ez a hot key a CTR+M lesz. Az eltérés csupán annyi, hogy könyvtár helyett egy új props fájlt adunk hozzá a faszerkezethez.

## 17. fejezet

# Helló, Lauda!

### 17.1. Port scan

A feladat az volt, hogy mutassunk rá ebben a port szkennelő forrásban a kivételkezelés szerepére!

Forrás: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html#id527287>

Az alábbi program a parancssori argumentumként kapott gép 1024-nél kisebb portjával próbál TCP/IP kapcsolatot létrehozni. Sikeres futtatás esetén az adott gépen egy szerverfolyamat figyeli az adott portot.

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

 for(int i=0; i<1024; ++i)

 try {

 java.net.Socket socket = new java.net.Socket(args[0], i);

 System.out.println(i + " figyel");

 socket.close();

 } catch (Exception e) {

 System.out.println(i + " nem figyel");

 }

 }
}
```

A kapcsolódáshoz a "Socket" típusú objektumra van szükség, melynek konstruktorába kell megadni az IP-címet és a portot. Mint azt már említettem, sikeres kapcsolódás esetén egy szerver fogja figyelni az adott portot, de mi történik akkor, ha a kapcsolódás sikertelen?



Itt jön a képbe a kivételkezelés. Ebben az esetben a "Socket" konstruktora "IOException" kivételt dob. Ha a programunk hibát ad vissza, akkor a terminálba kiírjuk, hogy az adott port nem figyelt. Kivételkezelés nélkül is tudjuk lefordítani a programot.

## 17.2. AOP

A feladat az, hogy szőjünk bele egy átszövő vonatkozást az első védési programod Java átíratába!

Ebben a feladatban az átszövés orientált programozással fogunk ismerkedni meg, amihez az AspectJ nyelvet fogjuk használni. Az átszövés lényege, hogy anélkül tudunk változtatni egy korábban írt program működésén, hogy a forráskódhoz hozzányúlunk. Ehhez egy szöveget kell írni, amivel pl egy függvény működésén változtatunk.

Először 3 fogalommal kell megismerkednünk:

Kapcsolódási pont - az eredeti program egyik függvénye.

Vágási pont - a csatlakozási pontokat jelöljük vele.

Tanács - azt tartalmazza, hogy hogyan szeretnénk módosítani az eredeti program működését.

A fát preorder szeretnénk bejárni. Ezt már korábban megcsináltuk, azonban akkor a forráskódot közvetlenül módosítottuk. Nézzük az "Aspect.aj" tartalmát.

```
privileged aspect Aspect{
 void around(LZWBInFa fa, LZWBInFa.Csomopont elem, java.io. ←
 BufferedWriter os):
 call(public void LZWBInFa.kiir(LZWBInFa.Csomopont, java.io. ←
 BufferedWriter))
 && target(fa) && args(elem, os){
 if (elem != null)
 {
 try{
 ++fa.melyseg;
 for (int i = 0; i < fa.melyseg; ++i)
 os.write("---");
 os.write(elem.getBetu () + "(" + (fa.melyseg - 1) + ")\n");
 fa.kiir(elem.egyGyermek (), os);
 fa.kiir(elem.nullasGyermek (), os);
 --fa.melyseg;
 }
 catch (java.io.IOException e){
 System.out.println("Csomópont írása nem sikerült.");
 }
 }
 }
}
```

Az eredeti Java forráshoz képest nem sok eltérés van. Ami újdonságként feltűnhet, az a "privileged". Ez ahhoz szükséges, hogy az aspektus hozzá tudjon férni az osztályok privát tagjához. Azt, hogy éppen aspektust írunk, nem osztályt, a "aspect" kulcsszó használatával tudjuk megadni.

```
void around(LZWBinFa fa, LZWBinFa.Csomopont elem, java.io.BufferedWriter os ↵
):
 call(public void LZWBinFa.kiir(LZWBinFa.Csomopont, java.io. ↵
 BufferedWriter))
```

Az "around" függvény paraméterének megadjuk az LZWBinFa objektumot, melyen keresztül elérjük a nem statikus tagokat. Ezután megadjuk, hogy melyik függvény helyett hívódjon meg az "around" függvény. Ezt követően az "args" kulcsszóval megadjuk, hogy az "around" mely paramétereit szeretnénk átadni a "kiir" függvénynek, majd '&&' elválasztva azt, "target" kulcsszóval hogy melyik objektumra szeretnénk végrehajtani. Az "around" törzse a már ismert kódrészletet tartalmazza.

Futtatás:

```
mester@lenovo700: ~
Fájl Szerkesztés Nézet Keresés Terminál Súgó
mester@lenovo700:~$ ajc LZWBinFa.java Aspect.aj
mester@lenovo700:~$ java -cp ./aspectjrt.jar:. LZWBinFa bemenet.txt -o kimenet.t
```

Alapprogram kimenet:

```
-----1 (2)
-----0 (3)
-----1 (5)
-----0 (4)
-----1 (1)
-----0 (2)
-----0 (3)
-----0 (4)
---/ (0)
-----1 (2)
-----0 (1)
-----0 (2)
-----0 (3)
depth = 5
mean = 3.5
var = 1.2909944487358056
```

Aspektus használatával:

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

```
-----0(3)
depth = 5
mean = 3.5
var = 1.2909944487358056
```

## 17.3. Junit teszt

A [https://progater.blog.hu/2011/03/05/labormeres\\_otthon\\_avagy\\_hogyan\\_dolgozok\\_fel\\_egy\\_pedat](https://progater.blog.hu/2011/03/05/labormeres_otthon_avagy_hogyan_dolgozok_fel_egy_pedat) poszt kézzel számított mélységét és szórását dolgozd be egy Junit tesztbe.

A Junit egy a Java nyelvhez kifejlesztett egységtesztelő keretrendszer. Segítségével automatizált módon tudjuk tesztelni, hogy a programunk a várt módon működik-e. Mi a mélység, szórás és átlag értékét fogjuk ellenőrizni, hogy megfelelnek-e a posztban látottaknak.

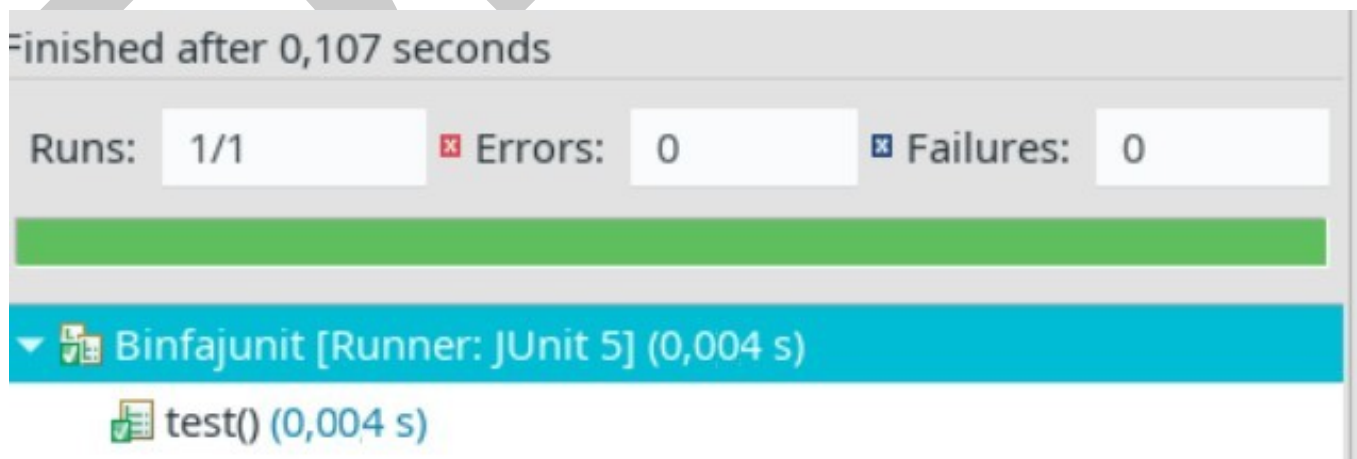
A posztban felhasznált bitsorozatot megadjuk az LZWBinFa-nak:

```
public class BinfaTest {
 LZWBinFa binfa = new LZWBinFa();
 @org.junit.Test
 public void tesBitFeldolg() {
 for (char c : "01111001001001000111".toCharArray())
 {
 binfa.egyBitFeldolg(c);
 }
 }
}
```

Az egyBitFeldolg metódussal feldolgozzuk az inputot karakterenént. Ezt követően jön a feladat lényegi része, a Junit teszt.

```
org.junit.Assert.assertEquals(4, binfa.getMelyseg(), 0.0);
org.junit.Assert.assertEquals(2.75, binfa.getAtlag(), 0.001);
org.junit.Assert.assertEquals(0.957427, binfa.getSzoras(), 0.0001);
```

A bitsorozat feldolgozása után ellenőrizzük le, hogy a BinFa objektum és posztban szereplő értékek megegyeznek-e. Ha igen, akkor a program jól működött, a teszt sikeres volt. Ha nem egyeznek az értékek, akkor "AssertionError"-t kapunk és a teszt sikertelen.



Láthatjuk, hogy a teszt sikeres volt.

## 17.4. Android játék

A feladat az volt, hogy készítsünk egy tetszőleges, egyszerű android játékot. Az általam készített játék egyben az is, amit a legegyszerűbbnek vélek: Találd ki a számot (Higher Lower) játék. Segítségül és alapul az alábbi videó szolgált:

<https://www.youtube.com/watch?v=I9YU0bfxeV0>

Először megtervezzük a felhasználói felületet. A játékunk UI-ja nagyon egyszerű lesz a következő nézetekkel:

- TextView, amely lehetővé teszi számunkra, hogy minden kísérlet után üzenetet jelenítsünk meg a felhasználó számára. A megjelenő üzenet vagy „túl magas”, vagy „túl alacsony” lesz.
- Az EditText lehetővé teszi a játékos számára annak a számnak a beírását, ami szerinte a helyes szám lesz
- Egy gomb, ami lehetővé teszi a játékos számára, hogy érvényesítse ("adja le") a tippjét.

Az *activity\_main.xml* fájl a következő képen néz ki:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 xmlns:tools="http://schemas.android.com/tools"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 tools:context="com.ssaurel.higherlower.MainActivity">

 <TextView
 android:id="@+id/msg"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_centerHorizontal="true"
 android:layout_marginTop="50dp"
 android:text="@string/start_msg"
 android:textSize="22sp"
 android:textStyle="bold"/>

 <EditText
 android:id="@+id/numberEnteredEt"
 android:layout_width="200dp"
 android:layout_height="wrap_content"
 android:layout_below="@id/msg"
 android:layout_centerHorizontal="true"
 android:layout_marginTop="50dp"
 android:inputType="number"/>

 <Button
 android:id="@+id/validate"
 android:layout_width="200dp"
 android:layout_height="wrap_content"
```

```
android:layout_below="@id/numberEnteredEt"
android:layout_centerHorizontal="true"
android:layout_marginTop="60dp"
android:text="Validate"/>
```

```
</RelativeLayout>
```

Ezután megírjuk a játék "logikáját" Java nyelven, tehát azt az algoritmust, ami a háttérben dolgozni fog.

A játék elindításához egy `newGame()` metódust írunk, amelyben a `Random` Java standard objektum `nextInt` statikus módszerét használjuk az 1 és 1000 közötti szám kiválasztására. Ezután egy kezdő üzenetet jelenítünk meg a `TextView`-ban. Töröljük az `EditText` szöveget, amelyet a játékos fog használni a kitalálni kívánt szám megadására. Végül nullára állítottuk a `numberTries` tulajdonságot.

```
private void newGame() {
 numberToFind = RANDOM.nextInt(MAX_NUMBER) + 1;
 msgTv.setText(R.string.start_msg);
 numberEnteredEt.setText("");
 numberTries = 0;
}
```

Ezután szükségünk van egy metódusra a játékos által megadott szám érvényesítésére. A szám érvényesítése azt jelenti, hogy ellenőrizzük, hogy a szám megegyezik-e a `numberToFind` egész számmal. Ha igen, akkor egy üzenetet jelenítünk meg, amely jelzi a játékosnak, hogy megtalálta a számot és a kísérletek számát.

Ha nem, akkor ellenőrizzük, hogy a megadott szám alacsonyabb vagy nagyobb-e, mint a `numberToFind` egész szám. Ez lehetővé teszi, hogy megjelenítsük a felhasználó számára a „Túl magas” vagy a „Túl alacsony” üzenetet a felhasználói felületünk `TextView`-ban. Nyilvánvaló, hogy nem felejtjük el, hogy minden alkalommal megnöveljük a kísérletek számát, amikor az `validate()` metódust meghívódik.

A `validate()` metódus:

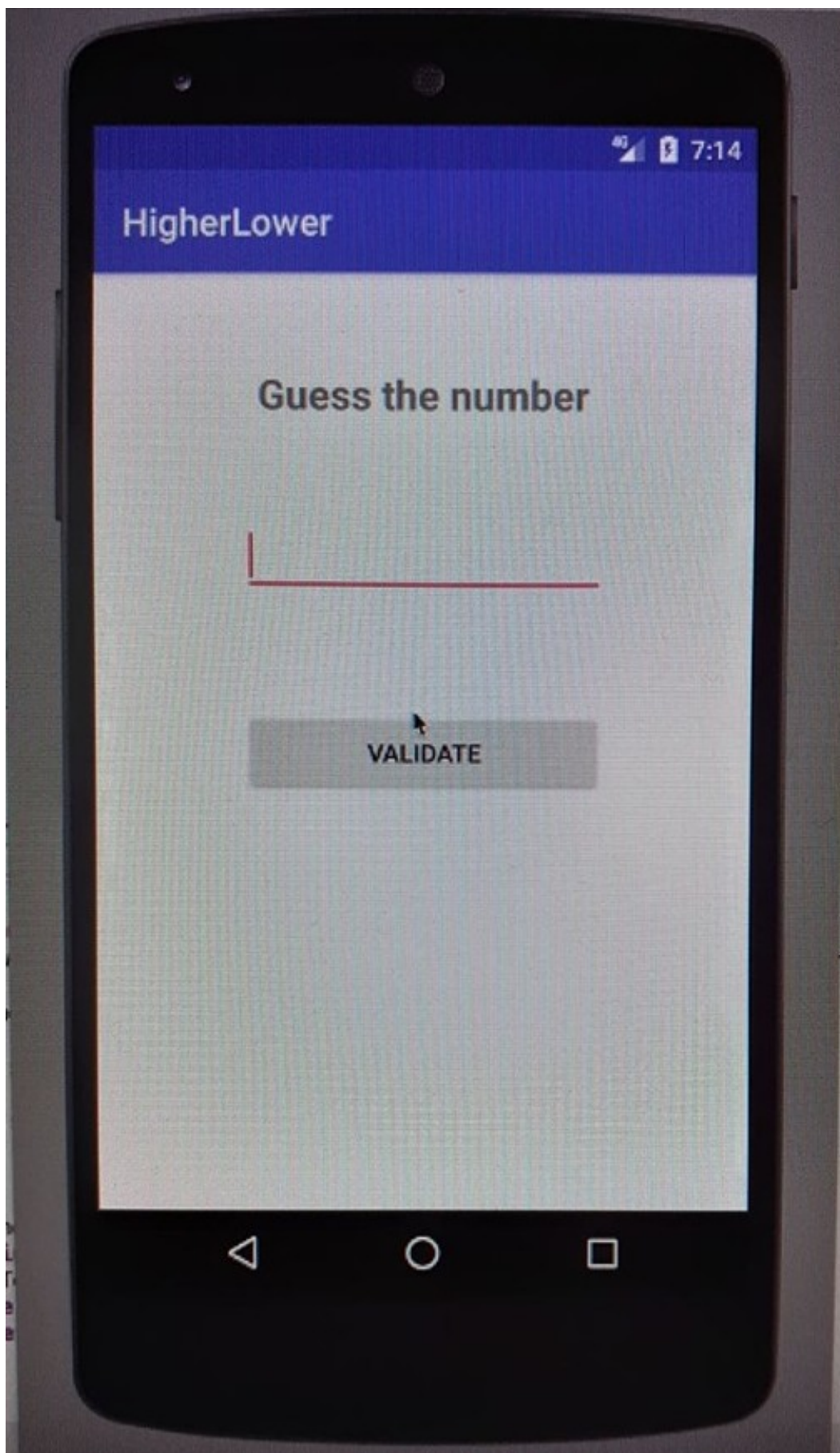
```
private void validate() {
 int n = Integer.parseInt(numberEnteredEt.getText().toString());
 numberTries++;

 if (n == numberToFind) {
 Toast.makeText(this, "Congratulations ! You found the number " +
 numberToFind +
 " in " + numberTries + " tries", Toast.LENGTH_SHORT).show();
 newGame();
 } else if (n > numberToFind) {
 msgTv.setText(R.string.too_high);
 } else if (n < numberToFind) {
 msgTv.setText(R.string.too_low);
 }
}
```

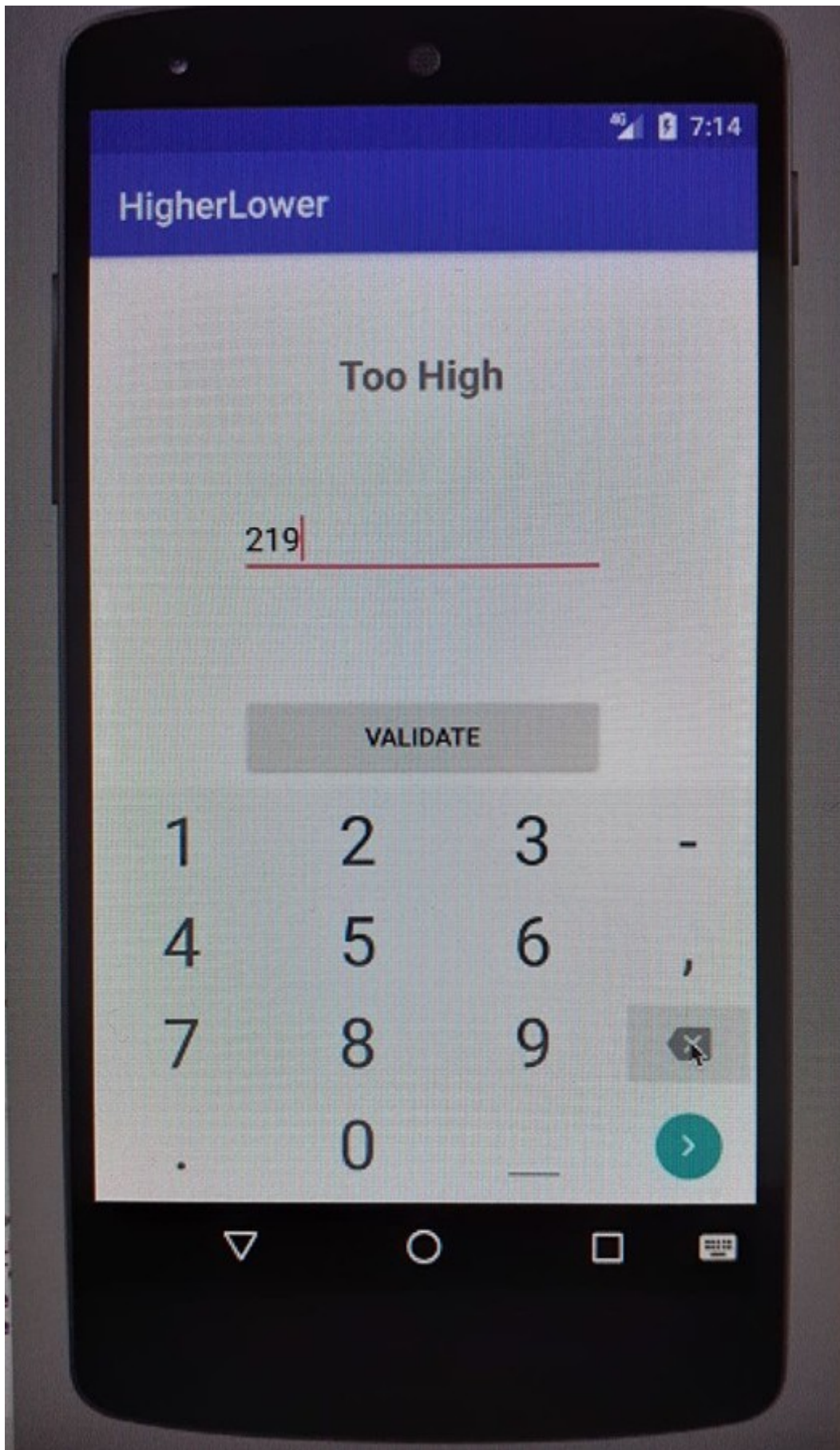
A teljes forráskód megtalálható az alábbi linken:

<https://github.com/mesterakos963/Prog2/tree/master/Prog2/Android%20game>

Futás közben:







## 18. fejezet

# Helló, Calvin!

### 18.1. MNIST

Az alap feladat megoldása, +saját kézzel rajzolt képet is ismerjen fel,

[https://progpater.blog.hu/2016/11/13/hello\\_samu\\_a\\_tensorflow-bol](https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol) Háttérként ezt vetítsük le:

<https://prezi.com/0u8ncvvoabcr/no-programming-programming/>

A program futtatásakor rengeteg error-ba ütközünk. Ezek javítása után a program gond nélkül fog futni. Tanítás után először a `W_0` súlyokat ábrázolja. Ez adja meg 0 valószínűségét. A kapott ábra fogja mutatni, hogy a program milyenek "képzeli el" vagy milyenek "tanulta meg" a 0 számjegyet.

Az MNIST olyan adatbázissal dolgozik, ami 60 000 képet tartalmaz. A pontosság meghatározásához 10 000 képet használunk.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse

Import data
from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf
old_v = tf.logging.get_verbosity()
tf.logging.set_verbosity(tf.logging.ERROR)

import matplotlib.pyplot

FLAGS = None

def main(_):
 mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
 # Create the model
 x = tf.placeholder(tf.float32, [None, 784])
 W = tf.Variable(tf.zeros([784, 10]))
```



```
b = tf.Variable(tf.zeros([10]))
y = tf.matmul(x, W) + b
```

Az MNIST adatbázis behívását és eltárolását követően kezdjük el inicializálni a Tensorflow könyvtárat.

Ahogy a Perceptron feladatban is, itt is egy neurális hálózatot használunk. Ennek rétegeit a felhasználás szerint kell beállítani. A rétegek az inputból információt nyernek ki.

Beállítunk néhány változót:

- `x` - egy tenzor, ebbe fogjuk küldeni az értékeket, megadjuk neki, hogy 784 pixelt tartalmazó képeket fog kapni;
- `w` - súly;
- `b` - bias;
- `y` - egy függvény (`tf.placeholder`) kiértékelése utáni értéket tárolunk benne, szükségese hozzá a '`w`' és '`y`' változók.

```
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(←
 ←
 labels = y_, logits = y))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(←
 cross_entropy)
```

`cross_entropy` - függvény, ami az eredeti és a becsült érték közötti különbséget adja meg, a mi célunk ennek az értéknek a minimalizálása.

```
sess = tf.InteractiveSession()
Train
tf.initialize_all_variables().run(session=sess)
print("-- A halozat tanitasa")
for i in range(1000):
 batch_xs, batch_ys = mnist.train.next_batch(100)
 sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
 if i % 100 == 0:
 print(i/10, "%")
print("-----")
```

A tanítás folyamata lépésekben történik. Ehhez egy beépített "`GradientDescentOptimizer`" nevű függvényt használunk. Ez egy kezdeti értéket frissít addig, amíg a "`cost`" függvény nem éri el a minimumot.

Az alábbi programrészben elindítjuk a folyamatot, az adatot feldaraboljuk. A pontosságot kiírjuk a standard kimenetre.

```
Test trained model
print("-- A halozat tesztelese")
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

```
print("-- Pontosság: ", sess.run(accuracy, feed_dict={x: mnist.test. ←
 ←-
 images,
 y_: mnist.test. ←
 labels}))
print("-----")
print("-- A MNIST 42. tesztkepenek felismerese, mutatom a számot, a ←-
 továbblepeshhez csukd be az ablakat")
img = mnist.test.images[42]
image = img

matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot. ←
 cm ←-
 .binary)
matplotlib.pyplot.savefig("4.png")
matplotlib.pyplot.show()

classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
print("-----")

print("-- A MNIST 11. tesztkepenek felismerese, mutatom a számot, a ←-
 továbblepeshhez csukd be az ablakat")

img = mnist.test.images[11]
image = img
matplotlib.pyplot.imshow(image.reshape(28,28), cmap=matplotlib.pyplot. ←
 cm. ←-
 binary)
matplotlib.pyplot.savefig("8.png")
matplotlib.pyplot.show()

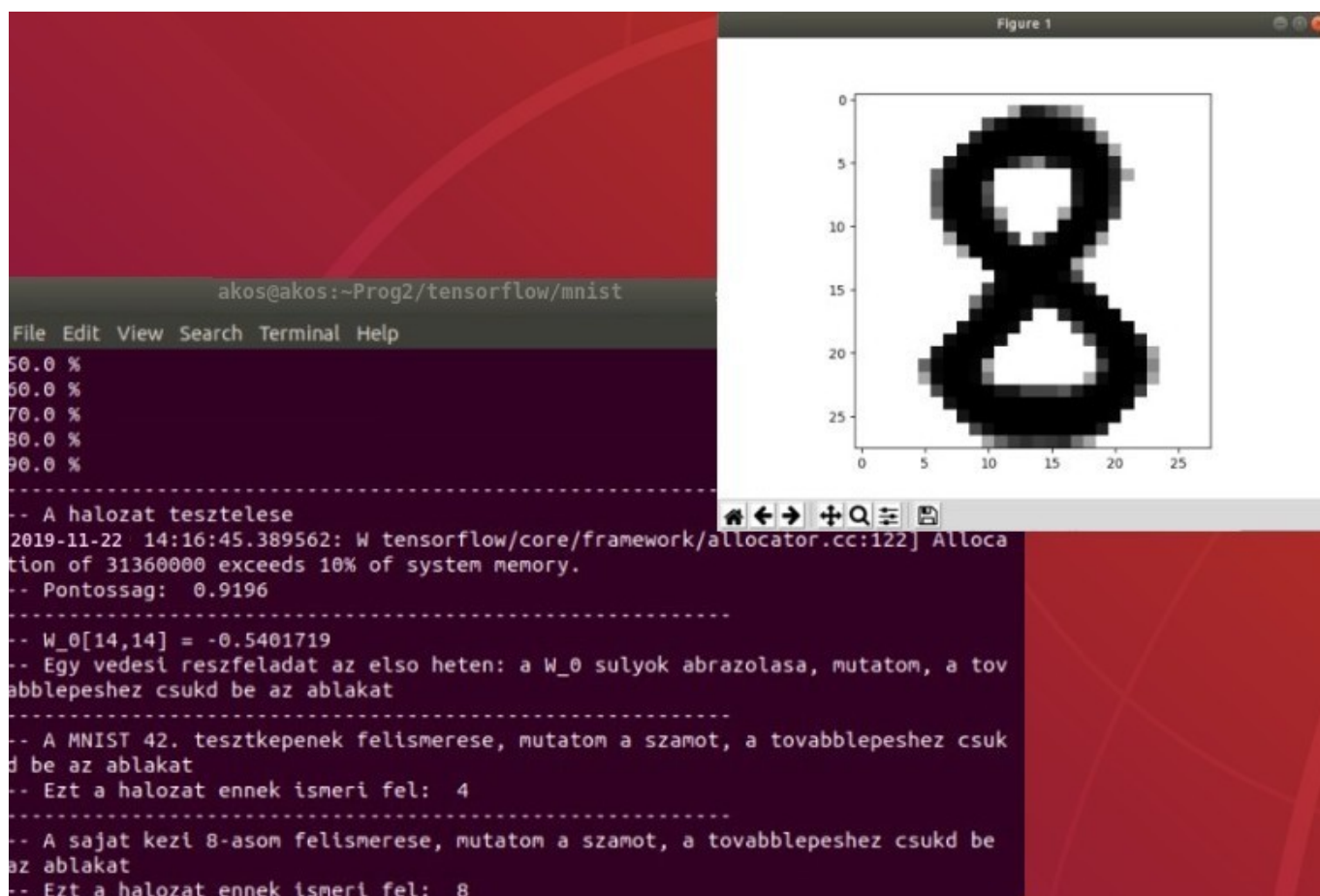
classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
print("-----")

if __name__ == '__main__':
 parser = argparse.ArgumentParser()
 parser.add_argument('--data_dir', type=str, default='/tmp/ ←
 tensorflow/ ←-
 mnist/input_data',
 help='Directory for storing input data')
 FLAGS = parser.parse_args()
 tf.app.run()
```

Elkészült a hálózat. Lehet tesztelni az MNIST adatbázis képeivel, aztán ahogy a feladat is kérte egy általunk rajzolt számmal.





Amint azt láthatjuk a program a helyesen ismerte fel a nyolcast.

## 18.2. Deep MNIST

A feladatunk ugyanaz, mint az előbb, csupán a mély változatot kell megcsinálnunk. Ez annyit tesz, hogy több köztes réteg, van, amivel megszűrjük az adatot. Lényegesen megnő a tanulási idő. Akár 8-10 órát is igénybevehet.

Csakúgy, mint az előző feladatban, itt is először meg kell nyitni a vizsgálandó képet, majd dekódolni. A kimeneti kép color channel-je grayscale lesz. Ezt a `decode_png()` függvényénél tudjuk beállítani az 1-es paraméter megadásával.

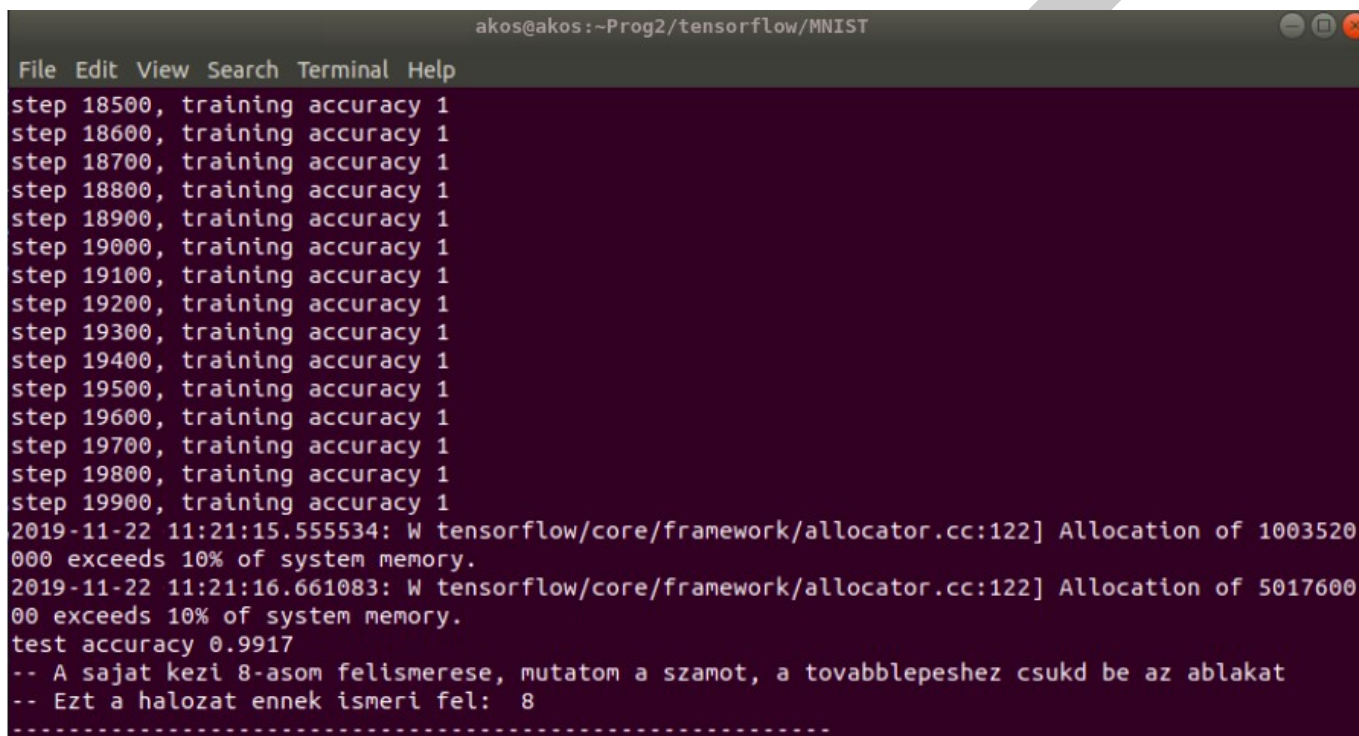
```
def reading():
 file = tf.read_file("sajat8a.png")
 img = tf.image.decode_png(file, 1)
 return img
```

A hosszú tanítási folyamat után leteszteljük a képünket az előző feladatban látottak alapján.

```
img = reading()
image = img.eval()
image = image.reshape(28*28)
matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot.cm.binary)
```

```
matplotlib.pyplot.savefig("8.png")
matplotlib.pyplot.show()
classification = sess.run(tf.argmax(y_conv, 1), feed_dict={x: [image], ←
 keep_prob: 1.0})
```

Az előző példában használt 8-as képet használtam fel. Itt annyi történik, hogy az `y_conv`-ből kiválasztjuk a legnagyobb értékkel rendelkező indexet, tehát azt, amelyik a legnagyobb valószínűséggel illeszkedik a karakterünkre.

A screenshot of a terminal window titled 'akos@akos:~Prog2/tensorflow/MNIST'. The terminal shows the output of a training process. It lists steps from 18500 to 19900, each with 'training accuracy 1'. Below this, there are two memory warnings from TensorFlow: 'Allocation of 100352000 exceeds 10% of system memory.' and 'Allocation of 50176000 exceeds 10% of system memory.' followed by 'test accuracy 0.9917'. At the bottom, there are two lines of Hungarian text: '-- A saját kezi 8-asom felismerese, mutatom a szamot, a tovabblepeshez csukd be az ablakat' and '-- Ezt a halozat ennek ismeri fel: 8'.

```
akos@akos:~Prog2/tensorflow/MNIST
File Edit View Search Terminal Help
step 18500, training accuracy 1
step 18600, training accuracy 1
step 18700, training accuracy 1
step 18800, training accuracy 1
step 18900, training accuracy 1
step 19000, training accuracy 1
step 19100, training accuracy 1
step 19200, training accuracy 1
step 19300, training accuracy 1
step 19400, training accuracy 1
step 19500, training accuracy 1
step 19600, training accuracy 1
step 19700, training accuracy 1
step 19800, training accuracy 1
step 19900, training accuracy 1
2019-11-22 11:21:15.555534: W tensorflow/core/framework/allocator.cc:122] Allocation of 1003520
000 exceeds 10% of system memory.
2019-11-22 11:21:16.661083: W tensorflow/core/framework/allocator.cc:122] Allocation of 5017600
00 exceeds 10% of system memory.
test accuracy 0.9917
-- A saját kezi 8-asom felismerese, mutatom a szamot, a tovabblepeshez csukd be az ablakat
-- Ezt a halozat ennek ismeri fel: 8
.....
```

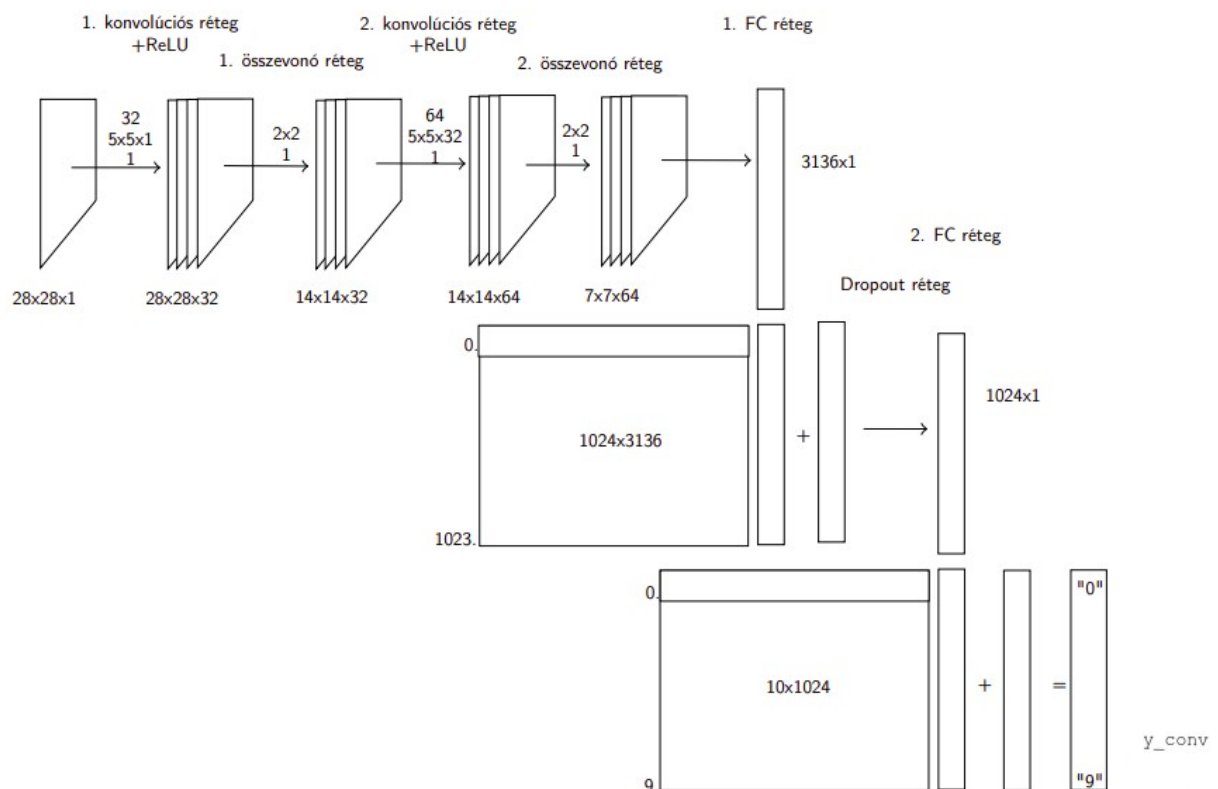
A hosszú futásidő oka az iterációk számában keresendő. 1000 helyett most 20.000 iterációt használunk illetve bonyolultabb modellt. Hogy ne kelljen a tanítási időre várni a tanítási állapot kimenthető.

```
for i in range(20000):
 batch = mnist.train.next_batch(50)
 if i % 100 == 0:
 train_accuracy = accuracy.eval(feed_dict={
 x: batch[0], y_: batch[1], keep_prob: 1.0})
 print('step %d, training accuracy %g' % (i, train_accuracy))
 train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: ←
 0.5})
```

Minden változó értékét kimentjük.. Ezt a `save` függvénnyel érjük el, ami minden változót kiment egy adott fájlba (`model.ckpt`).

```
saver = tf.train.Saver()
saver.save(sess, "./model/model.ckpt")
```

Tanár úr diasorából az ábra:



Jól szemlélteti a modell bonyolultságát.

Egy pár szót a forrásról:

Az első réteg a 28x28-as kép. Az első konvolúciós réteget így kapjuk meg:

```
with tf.name_scope('conv1'):
 W_conv1 = weight_variable([5, 5, 1, 32])
 b_conv1 = bias_variable([32])
 h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
```

`weight_variable` - megadott alakú változót ad vissza.

`bias_variable` - 0 vagy 1 értéket ad vissza.

`conv2d` - kettő 4 dimenziós paraméternek számolja ki a 2 dimenziós konvolúcióját (paraméterek: input, szűrő).

A `max_pool` függvény maximum poolozást hajt végre. Az input tensor 2. és 3. dimenziójához képest a kimenet mérete a felére csökken. Ezután jön még egy konvolúciós réteg és egy poolozás.

```
with tf.name_scope('conv2'):
 W_conv2 = weight_variable([5, 5, 32, 64])
 b_conv2 = bias_variable([64])
 h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)

with tf.name_scope('pool2'):
 h_pool2 = max_pool_2x2(h_conv2)
```

Ezt követően megcsináljuk az első fully connected réteget:

```
with tf.name_scope('fc1'):
 W_fc1 = weight_variable([7 * 7 * 64, 1024])
 b_fc1 = bias_variable([1024])

 h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
 h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

Átalakítjuk a pool2-t, új súlyokat és biasokat vezetünk be. Az átadott keep\_prob érték ezután jelenik meg.

```
with tf.name_scope('dropout'):
 keep_prob = tf.placeholder(tf.float32)
 h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Ezután 10 értékre leképezzük a súlyokat és biasokat:

```
with tf.name_scope('fc2'):
 W_fc2 = weight_variable([1024, 10])
 b_fc2 = bias_variable([10])

 y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
 return y_conv, keep_prob
```

A súlyok és biasok értéke tanulás közben állítódik be.

### 18.3. Androig telefonra a TF objektum detektálója

Talán ez volt a legkönnyebb feladat. Csupán annyi volt a dolgunk, hogy klónozzuk le az alábbi repót:

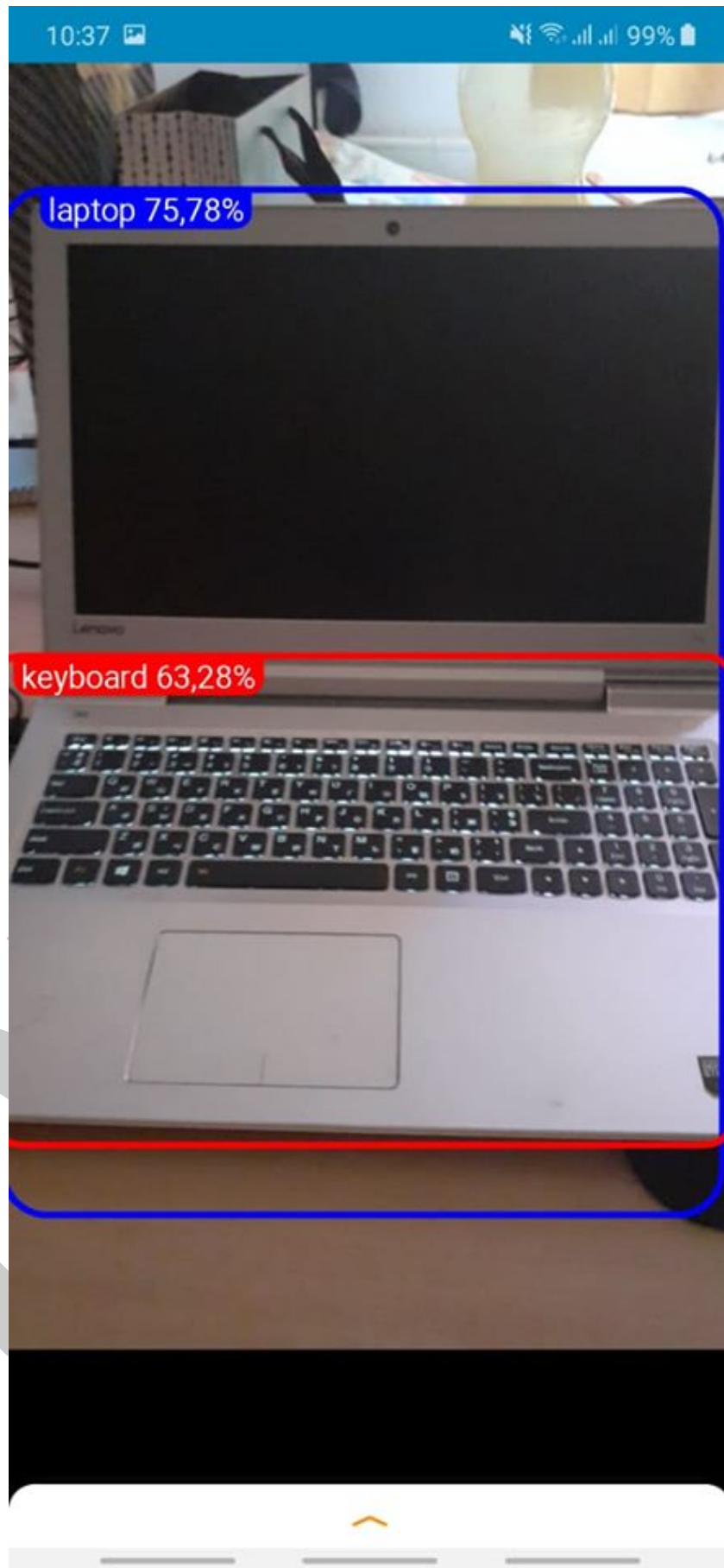
<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android>

Ebből Android studióban csinálunk egy apk-t és telepítjük telefonra.

Itt van 2 kép használat közben:







Láthatjuk, hogy a technológia egészen jól működik. Felismeri a tárgyat és megmondja, hogy hány százalékban biztos abban, amit lát.

## 18.4. CIFAR-10

A feladatunk szinte megegyezik az első kettővel, csupán annyi a különbség, hogy számok helyett most tárgyakat, élőlényeket kell felismertetni a programunkkal. A képek mostmár RGB-sek illetve a felbontásuk 28x28-ról 32x32-re nő. Az adatbázisban alapból 10 dologról találhatók képek.

Ahhoz, hogy saját képet is fel tudjunk ismertetni, először szükségünk van egy programra, ami kompatibilissé fogja tenni és átalakítja azt bináris formára.

```
from PIL import Image
import numpy as np
import sys

im = Image.open(sys.argv[1])
im = (np.array(im))

r = im[:, :, 0].flatten()
g = im[:, :, 1].flatten()
b = im[:, :, 2].flatten()
label = [0]

out = np.array(list(label) + list(r) + list(g) + list(b), np.uint8)
out.tofile("./cifar10_data/cifar-10-batches-bin/" + sys.argv[2])
```

Paraméternek megadjuk neki a kép nevét. A kép RGB komponenseit eltárolja egy tömbbe.

```
train_X = train_X.reshape(-1, 32, 32, 3)
test_X = test_X.reshape(-1, 32, 32, 3)
```

Amint láthatjuk az előző feladathoz képest a paraméterek megváltoztak: a képek mérete, amint azt az előbb is írtam, az utolsó paraméter (3) pedig azt adja meg, hogy most RGB képekkel fogunk dolgozni.

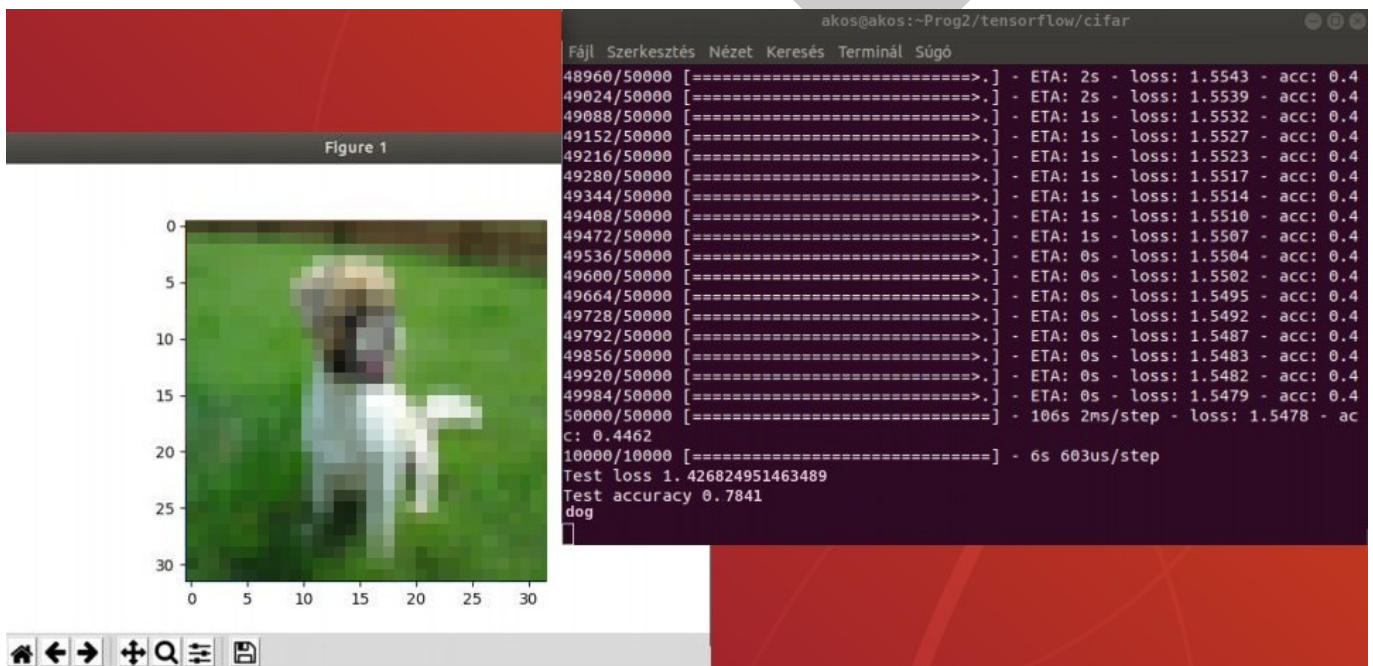
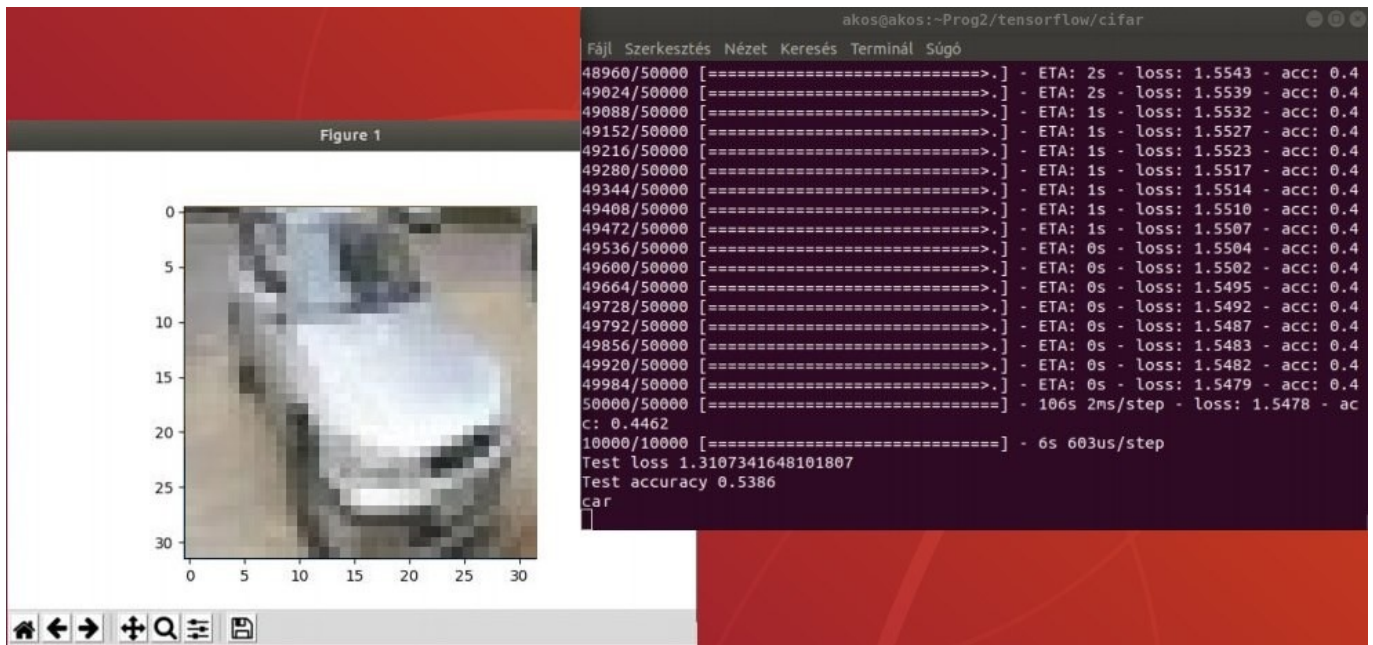
Néhány módosítást kell végrehajtani a kódon. Például az eredeti "test\_batch.bin" helyett "input.bin"-t kell használnunk. A cifar10.py-ban a "batch\_size" értékét 1-re kell állítani, mivel 1 db képet kell felismertetnünk vele.

Több neuront használunk fel és változnak az input\_shape függvény paraméterei is.

```
cifar_classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', ' ← dog', 'frog', 'horse', 'ship', 'truck']
```

Itt található az a 10 dolog, amit az adatbázis alapból tartalmaz (feladat elején említettem).

Nézzük meg futás közben:



Látható, hogy nem túl pontos a program, azonban képes volt minden általam kipróbált képet felismerni még így is.

## **IV. rész**

### **Irodalomjegyzék**

DRAFT

## 18.5. Általános

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

## 18.6. C

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

## 18.7. C++

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

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

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