

# **Univerzális programozás**

---

**Írd meg a saját programozás tankönyvedet!**

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

Copyright © 2019 Dankó Zsolt

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

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

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

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

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

**COLLABORATORS**

	<i>TITLE :</i>	
	Univerzális programozás	
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>
WRITTEN BY	Dankó, Zsolt	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 <a href="https://gitlab.com/nbatfai/bhax">https://gitlab.com/nbatfai/bhax</a> repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai
1.0.0	2019-05-09	Elkészült a könyv első inciális változata.	Dpara
2.0.0	2019-12-01	Elkészült a könyv második inciális változata.	Dpara

# 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

<b>I. Bevezetés</b>	<b>1</b>
1. Vízió	2
1.1. Mi a programozás? . . . . .	2
1.2. Milyen doksikat olvassak el? . . . . .	2
1.3. Milyen filmeket nézzek meg? . . . . .	2
<b>II. Tematikus feladatok</b>	<b>3</b>
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 . . . . .	9
2.6. Helló, Google! . . . . .	9
2.7. 100 éves a Brun téTEL . . . . .	9
2.8. A Monty Hall probléma . . . . .	11
3. Helló, Chomsky!	12
3.1. Decimálisból unárisba átváltó Turing gép . . . . .	12
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen . . . . .	12
3.3. Hivatalos nyelv . . . . .	13
3.4. Saját lexikális elemző . . . . .	14
3.5. l33t.l . . . . .	14
3.6. A források olvasása . . . . .	15
3.7. Logikus . . . . .	16
3.8. Deklaráció . . . . .	17

---

<b>4. Helló, Caesar!</b>	<b>18</b>
4.1. double ** háromszögmátrix . . . . .	18
4.2. C EXOR titkosító . . . . .	19
4.3. Java EXOR titkosító . . . . .	19
4.4. C EXOR törő . . . . .	20
4.5. Neurális OR, AND és EXOR kapu . . . . .	21
4.6. Hiba-visszaterjesztéses perceptron . . . . .	25
<b>5. Helló, Mandelbrot!</b>	<b>26</b>
5.1. A Mandelbrot halmaz . . . . .	26
5.2. A Mandelbrot halmaz a std::complex osztállyal . . . . .	27
5.3. Biomorfok . . . . .	27
5.4. A Mandelbrot halmaz CUDA megvalósítása . . . . .	27
5.5. Mandelbrot nagyító és utazó C++ nyelven . . . . .	28
5.6. Mandelbrot nagyító és utazó Java nyelven . . . . .	28
<b>6. Helló, Welch!</b>	<b>30</b>
6.1. Első osztályom . . . . .	30
6.2. LZW . . . . .	31
6.3. Fabejárás . . . . .	35
6.4. Tag a gyökér . . . . .	36
6.5. Mutató a gyökér . . . . .	38
6.6. Mozgató szemantika . . . . .	39
<b>7. Helló, Conway!</b>	<b>41</b>
7.1. Hangyszimulációk . . . . .	41
7.2. Java életjáték . . . . .	41
7.3. Qt C++ életjáték . . . . .	42
7.4. BrainB Benchmark . . . . .	42
<b>8. Helló, Schwarzenegger!</b>	<b>43</b>
8.1. Szoftmax Py MNIST . . . . .	43
8.2. Mély MNIST . . . . .	43
8.3. Minecraft-MALMÖ . . . . .	43

---

<b>9. Helló, Chaitin!</b>	<b>44</b>
9.1. Iteratív és rekurzív faktoriális Lisp-ben . . . . .	44
9.2. Gimp Scheme Script-fu: króm effekt . . . . .	44
9.3. Gimp Scheme Script-fu: név mandala . . . . .	45
<b>10. Helló, Gutenberg!</b>	<b>47</b>
10.1. Programozási alapfogalmak . . . . .	47
10.2. Programozás bevezetés . . . . .	48
10.3. Programozás . . . . .	49
<b>III. Második felvonás</b>	<b>51</b>
<b>11. Helló, Arroway!</b>	<b>53</b>
11.1. OO szemlélet . . . . .	53
11.2. „Gagyí” . . . . .	54
11.3. Yoda . . . . .	55
11.4. Kódolás from scratch . . . . .	56
<b>12. Helló, Liskov!</b>	<b>59</b>
12.1. Liskov helyettesítés sértése . . . . .	59
12.2. Anti OO . . . . .	60
12.3. Hello, Android! . . . . .	63
<b>13. Helló, Mandelbrot!</b>	<b>66</b>
13.1. Reverse engineering UML osztálydiagram . . . . .	66
13.2. Forward engineering UML osztálydiagram . . . . .	67
13.3. BPMN . . . . .	68
<b>14. Helló, Chomsky!</b>	<b>70</b>
14.1. Encoding . . . . .	70
14.2. l334d1c45 . . . . .	71
14.3. Full screen . . . . .	73
<b>15. Helló, Stroustrup!</b>	<b>76</b>
15.1. JDK osztályok . . . . .	76
15.2. Hibásan implementált RSA törése . . . . .	78
15.3. Változó argumentumszámú ctor . . . . .	80

---

---

<b>16. Helló, Gödel!</b>	<b>83</b>
16.1. STL map érték szerinti rendezése . . . . .	83
16.2. Alternatív Tabella rendezése . . . . .	84
16.3. GIMP Scheme hack . . . . .	85
<b>17. Helló, !</b>	<b>87</b>
17.1. OOCWC Boost ASIO hálózatkezelése . . . . .	87
17.2. SamuCam . . . . .	88
17.3. BrainB . . . . .	90
<b>18. Helló, Lauda!</b>	<b>93</b>
18.1. Port scan . . . . .	93
18.2. AOP . . . . .	94
18.3. Junit teszt . . . . .	96
<b>19. Helló, Calvin!</b>	<b>98</b>
19.1. MNIST . . . . .	98
19.2. CIFAR-10 . . . . .	101
19.3. Android telefonra a TF objektum detektálója . . . . .	102
<b>20. Helló, Berners-Lee!</b>	<b>105</b>
20.1. C++ és Java összehasonlítás . . . . .	105
20.2. Python . . . . .	106
<b>IV. Irodalomjegyzék</b>	<b>107</b>
20.3. Általános . . . . .	108
20.4. C . . . . .	108
20.5. C++ . . . . .	108
20.6. Lisp . . . . .	108

# Előszó

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

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

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

## Hogyan forgasd

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

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

## Milyen nyelven nyomjuk?

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

## Hogyan nyomjuk?

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

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

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

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



#### A DocBook XML 5.1 új neked?

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

## I. rész

### Bevezetés

# 1. fejezet

## Vízió

### 1.1. Mi a programozás?

### 1.2. Milyen doksikat olvassak el?

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

### 1.3. Milyen filmeket nézzek meg?

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

## **II. rész**

### **Tematikus feladatok**

**Bátf41 Haxor Stream**

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

## 2. fejezet

# Helló, Turing!

### 2.1. Végtelen ciklus

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

Megoldás forrása: [ciklus0.c](#) ; [ciklus1.c](#) ; [ciklusall.c](#)

A feladatba kicsit nehézkesen kezdtem bele, mivel végtelen ciklusokkal programozás során nem nagyon találkoztam, de egy kis kerestéssel megértettem, feldolgoztam az anyagot és megoldottam a feladatokat. Az alább használt `while(1) {}` ciklusokat azért választottam, mert szerintem ez az egyik legelterjedtebb és legegyszerűbb módszer. A `while(1) {}` azt jelenti, hogy addig fut a `while` ciklusban lévő kód, amíg teljesül az (1) feltétel, amely helyett boolean típusából kiindulva írhattam volna `while(true)-t`. Mivel itt nincs kód, ezért a ciklus a végtelenségig fut.

Elsőként az egymagos megoldást próbáltam ki, amely sikerült ezzel az egyszerű kóddal:

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

Következőnek azt próbáltam ki, hogyan lehetne megoldani, hogy ne foglalja le a CPU-t futás közben. Ehhez elég volt egy `sleep()`-et beraknom a végtelen ciklusba.

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

A végére hagytam azt a feladatot, hogy a CPU minden magját leterhelje a program, ez már kicsit bonyolultabb volt, mivel OpenMP-t kellett hozzá használnom, ezért `#include`-olnom kellett az "omp.h" könyvtárat.

```
#include <omp.h>

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

Ezt már fordítanom is máshogy kellett, a megszokott `gcc -fopenmp` helyett `gcc -fopenmp` parancsot kellett használnom.

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

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

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

    main(Input Q)
    {
        Lefagy2(Q)
    }
}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

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

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

Az eredményünk a következő: Ha a T1000 program megálló program, akkor mégsem áll meg, hanem végtelen ciklusba esik; és ha a T1000 végtelen ciklusba eső gép, akkor megáll a T1000 gép. Így egy szép kis paradoxont kapunk, amit Alan Turing fedezett fel. A felfedezés szerint a megállási probléma nem dönthető el, mivel az egyetlen megoldása ennek az ellentmondásnak az, hogy a T100 program nem létezik, tehát az alábbi algoritmusunk sem.

## 2.3. Változók értékének felcserélése

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

Megoldás videó: [https://bhaxor.blog.hu/2018/08/28/10\\_begin\\_goto\\_20\\_avagy\\_elindulunk](https://bhaxor.blog.hu/2018/08/28/10_begin_goto_20_avagy_elindulunk)

Megoldás forrása:[bhax/tmetic\\_tutorials/bhax-textbook/Turing/csere.c](https://bhax/tmetic_tutorials/bhax-textbook/Turing/csere.c)

Először is, deklarálnunk kell a két változót, értéket kell adni nekik, aztán kell felcserélnünk őket. Ezután hogy kimutassuk, a változók tényleg felcserélődnek-e, ki kell íratnunk a változók értékeit a printf() függvényvel. Ehhez #include-olunk kell a "stdio.h" könyvtárat, ebben van deklarálva a printf() függvényünk.

```
#include <stdio.h>

int main() {
    int a = 33;
    int b = 42;

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

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

    printf("Felcserélés után:\na = %d\nb = %d\n", a, b);
    return 0;
}
```

## 2.4. Labdapattogás

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

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

Megoldás forrása:[labda.c](https://bhax/labda.c); [labdaif.c](https://bhax/labdaif.c)

Az if-es megoldásban a labdának relatív "játszótere" van, ami attól függ, mekkora az ablak mérete; az if nélküli programban viszont a labdánk fix kereteken belül mozoghat, ezt indikálja a két szaggatott(kötőjeles) vonalunk. A kód sikeres fordításához meg kell hívnunk a "curses.h" könyvtárat, ami a libcurses package-ben van, így előbb le kellett töltenünk azt. A kód működésének szíve és lelke a for (; ; ) végtelen ciklus, itt "mozog" a labdánk. A lényege, hogy ha a labdánk eléri az egyik falat, akkor elkezdjen menni az ellenkező irányba, a gyorsaságot pedig az usleep(100000) függvény adja meg, ez 100ms-t jelent, vagyis a labda 0.1 másodpercenként mozog.

## 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: [szohossz.c](#)

A szóhossz megszámolásához szükségünk van a bitshift operátorra, ez található a while cikluson belül. Jelen ciklus azt csinálja, hogy az "a" változónkat addig shifteljük, vagyis addig léptetjük balra és adunk 0-t a jobb oldalára, amíg 0 nem lesz az értéke. Mindeközben minden shiftelésnél a "b" változó értékét is növeljük minden 1-el, amíg le nem zárul a ciklus, így a "b" fogja nekünk kimutatni, hány bites a szó a gépünkön.

## 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: [pagerank.c](#), BN FerSML projekt dia 55-59.

Egy weboldal fontossága teljesen szubjektív; attól függ, mit keres az olvasó. Viszont objektíven is meg tudjuk közelíteni a weboldalakat, mégpedig úgy, hogy a relatív fontosságot nézzük. Erre való a PageRank módszer, ami objektíven méri, az adott oldalt hányan keresik, vagyis más oldalak hányszor mutatnak rá.

A PageRankot linkmátrix megoldással fogjuk megvalósítani, ebben segítenek nekünk az alábbi ábrák:

$$L = (l_{i,j})$$

$$l_{i,j} = \begin{cases} \frac{1}{N(j)}, & j \rightarrow i \\ 0 & \text{otherwise} \end{cases}$$

$$L = \begin{pmatrix} l_{1,1} & \dots & l_{1,j} & \dots & l_{1,n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ l_{i,1} & \dots & l_{i,j} & \dots & l_{i,n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ l_{n,1} & \dots & l_{n,j} & \dots & l_{n,n} \end{pmatrix} \begin{pmatrix} PR(h_1) \\ \vdots \\ PR(h_n) \end{pmatrix}$$

$$\begin{aligned} PR(h_1) &= \sum_{j=1}^n l_{1,j} PR(h_j) \\ &\vdots \\ PR(h_i) &= \sum_{j=1}^n l_{i,j} PR(h_j) \\ &\vdots \\ PR(h_n) &= \sum_{j=1}^n l_{n,j} PR(h_j) \end{aligned}$$

Itt ha  $\mathbf{h}$  jelöli a PR vektort, akkor  $\mathbf{h} = \mathbf{Lh}$

## 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](https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/Primek_R)

Az ikerprímek olyan prímpárok, melyek különbsége 2.

Brun-tétel: Az ikerpríme reciprokösszege egy (Brun-konstans) véges értékhez konvergál. A prímek reciprokösszege divergens, ebből láthatjuk, hogy végtelen sok prímszám van.

Nézzük is ezt az R szimulációban:

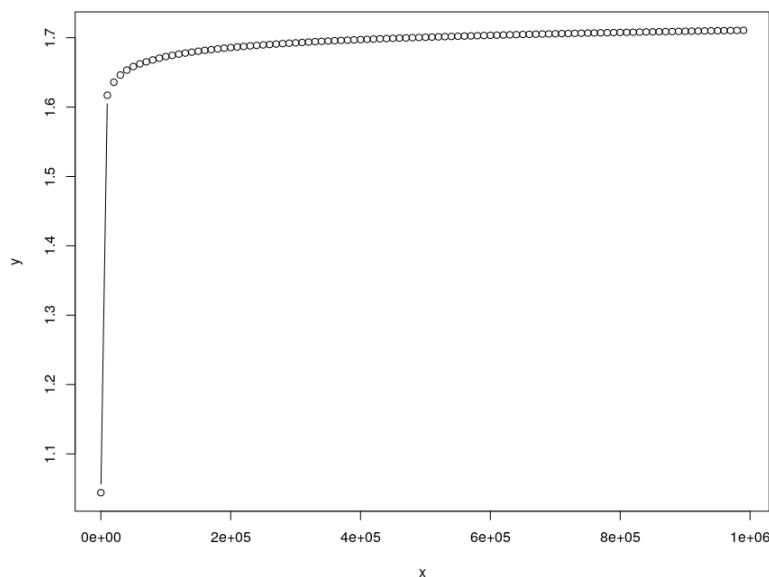
```
library(matlab)

stp <- function(x) {

  primes = primes(x)
  diff = primes[2:length(primes)] - primes[1:length(primes)-1]
  idx = which(diff==2)
  t1primes = primes[idx]
  t2primes = primes[idx]+2
  rt1plust2 = 1/t1primes+1/t2primes
  return(sum(rt1plust2))
}
```

Ha még nem tettük meg, töltük le a matlab könyvtárat, R-en belül `install.packages("matlab")`. Meghívjuk a matlab könyvtárat, majd megírjuk az `stp` függvényt. A függvény első sorában meghívunk egy `primes(x)` függvényt, ami az x prímeit adja vissza. Megírjuk a különbségfüggvényt, majd megnézzük, hogy hol 2 a különbség. A `t1primes` az ikerpár egyik tagja, az `idx` prímjeit mutatja; a `t2primes` a pár második tagja, szintén az `idx` prímjeit adja vissza, és hozzáad 2-t. Elosztjuk 1-el a két változót, majd összeadjuk őket, végül pedig visszakapjuk az előbbi függvény összegét. Rajzoljuk is ki ezt a függvényt:

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



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

A Monty Hall-paradoxon egy valószínűségi paradoxon, ami az USA-beli "Kössünk üzletet" című vetélkedő utolsó feladatán alapul. A paradoxon nevét Monty Hallról, a vetélkedő műsorvezetőjéről kapta. A lényege, hogy a műsorvezető bemutat három ajtót, amelyekből 2 ajtó túloldalán egy értéktelen nyeremény van, egy ajtó pedig egy értékes nyereményt rejti. A játékosnak ki kell választania egy ajtót, majd a műsorvezető kinyit egyet, amely egy értéktelen nyereményt tartalmaz. Ezután a játékosnak lehetősége van dönteni, hogy marad az eredetileg kiválasztott ajtajánál, vagy vált a másikra. Alapból a játékosnak  $\frac{1}{3}$  esélye van, hogy az értékes nyereményt találja el, és  $\frac{2}{3}$  esélye, hogy az értéktelent. Mivel kinyitották az egyik ajtót, így a játékos igazából 50% hogy az értékes nyeremény ajtaját nyitja ki, szóval a paradoxon nagy kérdése az, hogy érdemes lenne-e változtatni, vagy hogy minden számít-e egyáltalán.

## 3. fejezet

# Helló, Chomsky!

### 3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Tutor: Veress Máté

Megoldás forrása: <https://github.com/dankozsolti/prog1/blob/master/decimal.c>

Ezen Turing gép lényege, hogy a decimális számokat átváltja unáris számrendszerbe, és ezt a következőképp oldotta meg Turing: Feltételezzük, hogy a gép minden 0-ból indul, és a szalagunk a 0 fejtől balra üres. Egy szám végét szintén 0-val jelöljük, tehát ha input-ként beírjuk, hogy "6", az alábbit fogja kiírni: "01111110". A szalagot így fel lehet tölteni, például ha beírjuk, hogy "2, 3, 4", akkor a kimenet "0110111011110" lesz. Leprogramozni úgy tudjuk jelen gépünket, hogy bekérjük a decimális számot inputként, majd egy léptetéses ciklussal egyeseket csinálunk a számokból, és kiíratjuk azt.:

```
#include <stdio.h>

int main()
{
    int decimal;
    scanf ("%d", &decimal);
    printf("Unáris:\n0");

    for(int i=0; i < decimal; i++)
        printf("1");
    printf("0\n");
    return 0;
}
```

### 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:

Noam Chomsky amerikai nyelvész nevéhez többek közt a generatív nyelvtanok csoportosítása fűződik. Négy osztályba sorolta őket (Chomsky hierarchy):

- 0. típus(általános)
- 1. típus (környezetfüggő)
- 2. típus (környezetfüggetlen)
- 3. típus (reguláris)

Nekünk az első, azaz környezetfüggő típussal kell foglalkoznunk ebben a feladatban. Akkor környezetfüggő a grammatika, ha teljesül az adott feltétel:  $S \rightarrow \epsilon$ , és az S soha nem jelenik meg a jobb oldalon. Most pedig nézzük, hogy is néz ki ez gyakorlatban.

```
S, X, Y (változók)
a, b, c (konstansok)
S -> abc, S -> aXbc, Xb -> bX, Xc -> Ybcc, bY -> Yb, aY -> aaX, aY -> aa
```

Itt kiindulunk az S-ből, és a következőt kapjuk:

```
S (S -> aXbc)
aXbc (Xb -> bX)
abXc (Xc -> Ybcc)
abYbcc (bY -> Yb)
aYbcc (aY -> aa)
aabcc
```

Ezt a példát tovább fejthetjük, és így megkapjuk az alábbi levezetést:

```
S (S -> aXbc)
aXbc (Xb -> bX)
abXc (Xc -> Ybcc)
abYbcc (bY -> Yb)
aYbcc (aY -> aaX)
aaXbcc (Xb -> bX)
aabXbcc (Xb -> bX)
aabbcXcc (Xc -> Ybcc)
aabbcYbcc (bY -> Yb)
aabbcYbcc (bY -> Yb)
aaYbbbccc (aY -> aa)
aaabbccc
```

### 3.3. Hivatkozási nyelv

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

Megoldás videó:

Tutoráltam: Veress Máté

A legegyszerűbb módon egy `for()` ciklusban lehet bemutatni a két szabvány közötti különbséget. C99-ben, azaz az ISO szabvány szerint a `for` cikluson belül tudjuk egyszerre deklarálni, majd léptetni a változónkat, az alábbi módon:

```
for(int i, i<5, ++i)
```

C89-ben - vagyis az ANSI szabvány alkalmazásával - ha ugyanezt a kódot próbáljuk fordítani, hibát ír ki. Hogy működjön a kódcsipetünk, a `for()` cikluson kívül kell deklarálnunk az `i`-t.

```
int i;
for(i<5, ++i)
```

Hogy a gépünk felé jelezzük, melyik szabványt akarjuk alkalmazni, fordításkor a végére kell írnunk, hogy `-std=c99` vagy `-std=c89`

### 3.4. Saját lexikális elemző

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

Tutor: Veress Máté

Megoldás forrása:

Egy olyan lexert írunk, amely felismeri a bemenetünkbeli a valós számokat. A feladat megoldásához a Flex-et használjuk (így óriások vállán állunk, nem kispályázunk), ezért a `.l` kódot át kell konvertálnunk C-be, majd csak ezután tudjuk fordítani és futtatni, az alábbi módon:

```
lex -o valos.c valos.l
gcc valos.c -o valos
./valos
```

Mivel a lexerünk megírja magának a C programot, így csak annyit kell megadnunk, hogy miket tegyen a kódba. A kódunk `%` jelekkel van elválasztva; az első részben a `"stdio.h"` könyvtárat hívjuk meg és a `realnumbers` változónkat deklaráljuk, a második részben megadjuk, hogy számokkal dolgozunk. A következő részben a fordítási szabályokat adjuk meg a lexernek: Akármennyi számunk lehet (az első digit), majd legyen valamennyi a pont után, de legalább egy (így tizedes tört), de az utóbbi opcionális. Növeljük a `realnumbers` változót, majd kiíratjuk a `realnum`-ot. Az utolsó (main) rész elejében a `yylex()` függvényel hívjuk a lexikális elemzőt, majd kiírjuk a `realnumbers` számát.

### 3.5. l33t.I

Lexelj össze egy l33t cipher!

Tutoráltam: Szilágyi Csaba

Megoldás forrása: [1337.1](#)

Ehhez a feladathoz megoldásához szintén a Flex lexikális elemzőt használjuk, ezért az előző feladathoz hasonlóan fordítjuk a kódot:

```
lex -o 1337.c 1337.1
gcc 1337.c -o 1337
./1337
```

A kódunk fő eleme egy cipher, ennek a struktúráját építjük fel elsőnek. A <https://simple.wikipedia.org/wiki/Leet> oldalon található helyettesítő karaktereket használjuk ebben a programban. A struktúránkban deklarálunk egy karaktert(c) és azt, hogy az adott karaktert mire cserélhetjük. Hagyjuk a programnak, hogy kiválassza, melyik karakterre cserélje a négy megadott lehetőség közül az adott betűket vagy számokat. Hogy kezeljük a kis- és nagybetűket, a `tolower()` függvényt használjuk, ez átalakítja az összes nagybetűt lowercase-re.

## 3.6. A források olvasása

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

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

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



### Bugok

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

i.

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

ii.

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

iii.

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

iv.

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

v.

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

vi.

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

vii.

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

viii.

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

Megoldás forrása: [sig.c](#)

Megoldás videó:

Először is, mivel jeleket kezelünk, meg kell hívnunk a "signal.h" könyvtárat. Deklarálunk egy jelkezelő változót, ami kiírja, mikor elkapjuk a jelet. Ezután egy while(1) végtelen ciklusba megírjuk a kód azon részét, amely megfogja a SIGINT signalt, ami a Ctrl+C billentyűkombinációnak felel meg. Így hiába küldjük neki ezt a signalt, a programunk megfogja, így nem tudjuk azt bezárni, viszont a Ctrl+Z kombinációval továbbra is le lehet állítani, ráadásul ezt el sem tudjuk kapni.

## 3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/blob/master/attention\\_raising/MatLog\\_LaTeX](https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/MatLog_LaTeX)

Megoldás videó: <https://youtu.be/ZexiPy3ZxsA>, [https://youtu.be/AJSXOQFF\\_wk](https://youtu.be/AJSXOQFF_wk)

- $\$(\forall x \exists y ((x < y) \wedge (y \text{ prim}))) \$$  - minden x-re van olyan y, amely nagyobb x-nél és prímszám.
- $\$(\forall x \exists y ((x < y) \wedge (y \text{ prim}) \wedge (SSy \text{ prim}))) \$$  - minden x-re van olyan y, amely nagyobb x-nél, prímszám és SSy is prím.
- $\$ (\exists y \forall x (x \text{ prim}) \supset (x < y)) \$$  - létezik olyan y, amelynél minden x prímszám, és ezért x kisebb mint y.
- $\$ (\exists y \forall x (y < x) \supset \neg (x \text{ prim})) \$$  - létezik olyan y, amelynél minden x nagyobb mint y, ezért x nem prímszám.

### 3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

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

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:

**SKIP**

## 4. fejezet

# Helló, Caesar!

### 4.1. double \*\* háromszögmátrix

Megoldás videó:

Megoldás forrása: [tm.c](#)

A programhoz nem kell semmilyen előleges define vagy deklaráció a könyvtárhívásokon kívül, ezért egyből jöhet a main függvény. Deklarálunk egy int típusú változót, aminek értékül 5-öt adunk, ez a sorok számát fogja megadni később. Megadjuk, hogy ha bármilyen hibát észlel a program a malloc funkcionál, akkor a mutató a NULL-ba mutasson és kiugorjon a programból. A hibák észlelése között printf függvényekkel kiíratjuk a tárcímeket az output elejére. Nézzük meg a mutatókat három ugyanilyen függvénnel:

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

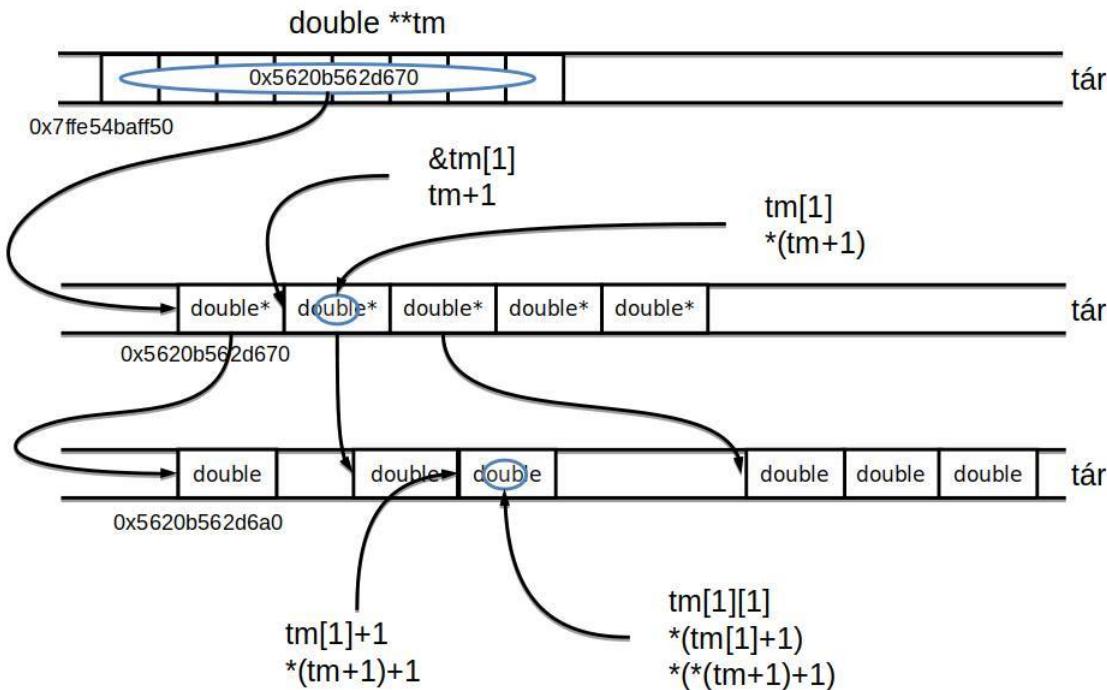
printf("%p\n", &tm);

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]);
```



A kép Bátfai Norbert előadás diájáról származik.

## 4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: [titkosit.c](#)

Mivel szöveget akarunk titkosítani, ezért azt bele kell rakkunk egy fájlba, a programunk pedig majd ezen fog dolgozni. Mivel kell egy visszafejtési kulcs, a könyvtárhívások után `#define`-oljuk a kulcsunk méretét, majd a buffer méretét is beállítjuk(256 byte). A define után nincs más dolgunk, mint a main függvényt megírni. A kulcsot és a buffert egy-egy `char` változóban tároljuk, majd megadjunk két `int` változót a kulcs indexének és az olvasott bajtoknak. A kulcs méretét is egy `int`-ben tároljuk, majd megadjuk, hogy a program futtatásakor be tudjuk írni a kívánt kulcsot argumentumként. Ezután jön egy `while` ciklus, ami a bufferből olvassa a bajtokat, majd a cikluson belül egy `for` ciklus, amely EXOR segítségével kicséri a szöveg karaktereit a megadott kulcsunk alapján. Mivel a program a standard outputra löki az átírt karaktereket, ezért egy outputot kell megadnunk, amelybe írhatja a titkosított szöveget.

## 4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Tutor: Győri Márk Patrik

Megoldás forrása: [titkositas.java](#)

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

class ExorTitkosito {
    public ExorTitkosito(String text, InputStream input, OutputStream output) {
        byte[] key = text.getBytes();
        byte[] buffer = new byte[256];

        int index = 0;
        int readed = 0;
        try {
            while ((readed = input.read(buffer)) != -1) {
                for(int i = 0; i < readed; i++) {
                    buffer[i] = (byte)(buffer[i] ^ key[index]);
                    index = (index + 1) % key.length;
                }
                output.write(buffer, 0, readed);
            }
        } catch (Exception e) {
            System.out.println("Hibas a ciklus *cry* Hiba: " + e.getMessage());
        }
    }
}

public class Main {
    public static void main(String[] args) {
        if(args.length > 0)
            new ExorTitkosito(args[0], System.in, System.out);
        else
            System.out.println("Nem adtal meg argumentet!");
    }
}
```

Egy külön osztályt hozunk létre a titkosításhoz (**class ExorTitkosito**), aminek input-ként szolgál a kulcs és egy olvasó-író argumentum. A **while** ciklus addig fut, amíg az olvasott bájtok el nem fogynak. Ezeket a bájtokat eltoljuk a megadott számú kules bájttal. minden bájton végigmegy a ciklus, majd visszaírjuk a kimeneti bufferbe. Ezt a művelet egy **try-catch** blokkba rakjuk, így elfogja a hibákat, ami java kódolásnál lényeges.

## 4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása: [toro.c](#)

A törő programunk lényege, hogy a titkosításban használt EXOR-ral fejtsük vissza a szöveget, bár ehhez még kellene fog pár plusz infó a programnak, amiből ki tud indulni. A program folyamatosan vizsgálja az átlagos szóhosszt, valamint feltételezi, hogy a szöveg tartalmazza a gyakori magyar szavakat(hogy, nem, az, ha). Ezekkel a vizsgálatokkal csökkentjük a potenciális töréseket. A másik dolog, amit tudnunk kell az az, hogy a kulcs hány és milyen típusú karakterből áll. A main függvényben a titkosításhoz hasonló deklarációkat használjuk, tehát a kulcs méretét és az olvasott bajtakat; majd egy while ciklusban olvassuk a titkosított bajtakat, ezután pedig a titkos bufferben lenullázzuk a maradék helyet. Előállítjuk az összes kulcsot for ciklusokkal, majd a lehetséges tiszta szövegeket kiírja a kulccsal együtt a standard outputra. Végül újra EXOR-ozunk, így nem kell egy második buffert alkalmaznunk.

## 4.5. Neurális OR, AND és EXOR kapu

R

A képlet és a vázlat forrása: [Tankönyvtár](#)

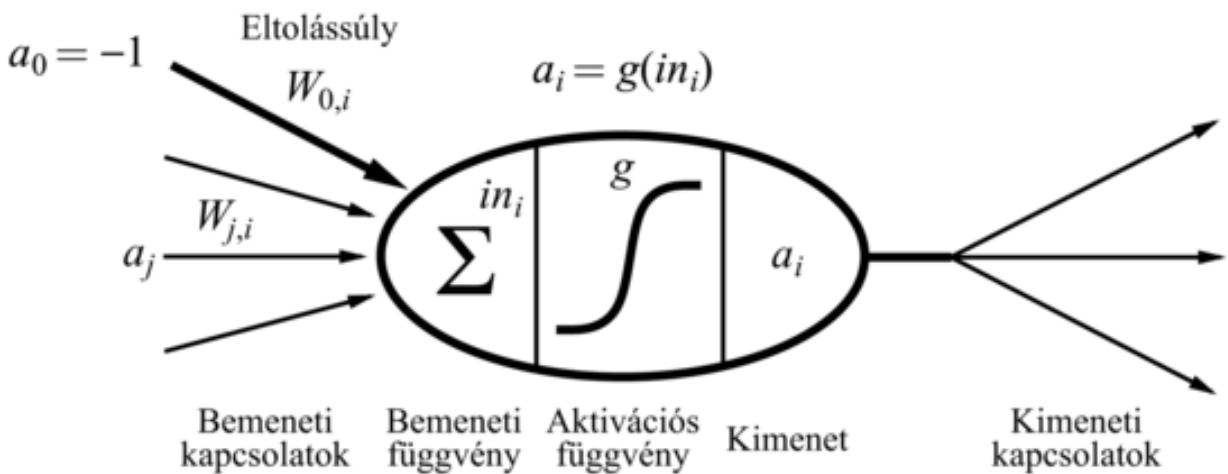
Megoldás videó: <https://youtu.be/Koyw6IH5ScQ>

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

Először is beszéljünk magáról a neuronról. A neuron egy olyan agysejt, melynek alapfeladata elektromos jelek összegyűjtése, feldolgozása és szétterjesztése. Elnagyolva az mondható, hogy a neuron akkor „tüzel”, amikor a bemeneti értékek súlyozott összege meghalad egy küszöböt. Az alábbi ábra a neuron egyszerű matematikai modelljét mutatja, ahogy McCulloch és Pitts megalkották. Az egység kimeneti aktivációja:

$$a_i = g\left(\sum_{j=0}^n W_{j,i} a_j\right)$$

ahol  $a_j$  a j-edik egység kimeneti aktivációja és  $W_{j,i}$  a j-től i-ig vezető összeköttetés súlya.



Mi főként a kapuval, vagyis az aktivációs függvényel foglalkozunk. Mivel ebben az R szimulációban neurális hálókkal fogunk foglalkozni ezért az első dolgunk az lesz, hogy meghívjuk a **library(neuralnet)** könyvtárat, majd a fenti ábra alapján megnézzük a kapukat. Kezdjük is az elsővel:

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

```

or.data <- data.frame(a1, a2, OR)

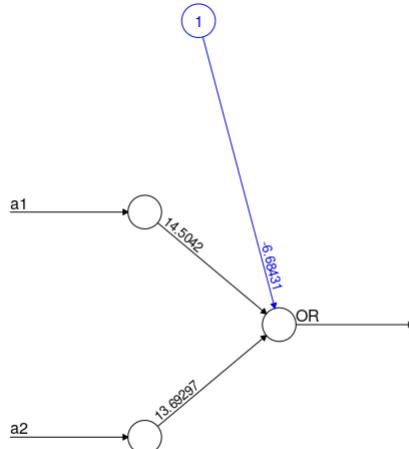
nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE, ←
  stepmax = 1e+07, threshold = 0.000001)

plot(nn.or)

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

```

Ha beírjuk a fenti kódot, a következőt fogja kidobni:



Error: 1e-06 Steps: 145

A fenti kód és ábra alapján a többi kaput is meg lehet oldani, nem kell szinte semmit átírnunk. Elkészítjük a második (ORAND) kaput.

```

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

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

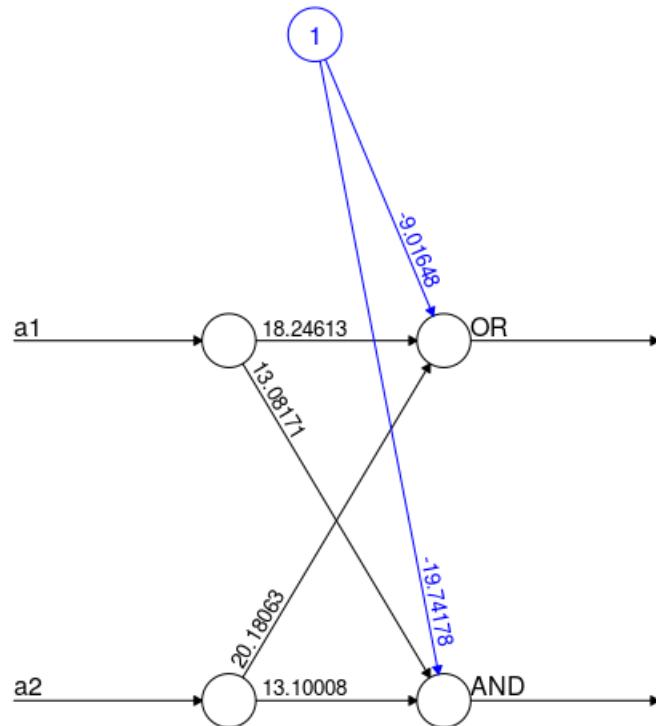
nn.orand <- neuralnet(OR+AND~a1+a2, orand.data, hidden=0, linear.output= ←
  FALSE, stepmax = 1e+07, threshold = 0.000001)

plot(nn.orand)

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

```

mivel az AND és az OR függetlenek egymástól, ezért ugyanazon hálózatban két kimenete lesz és így nyugodtan beírhatjuk mindkettőt.



Error: 3e-06 Steps: 190

Megnézzük az EXOR-os példát, itt sem kell átírnunk szinte semmit, csak az OR-t (vagy ORAND) EXOR-ra.

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

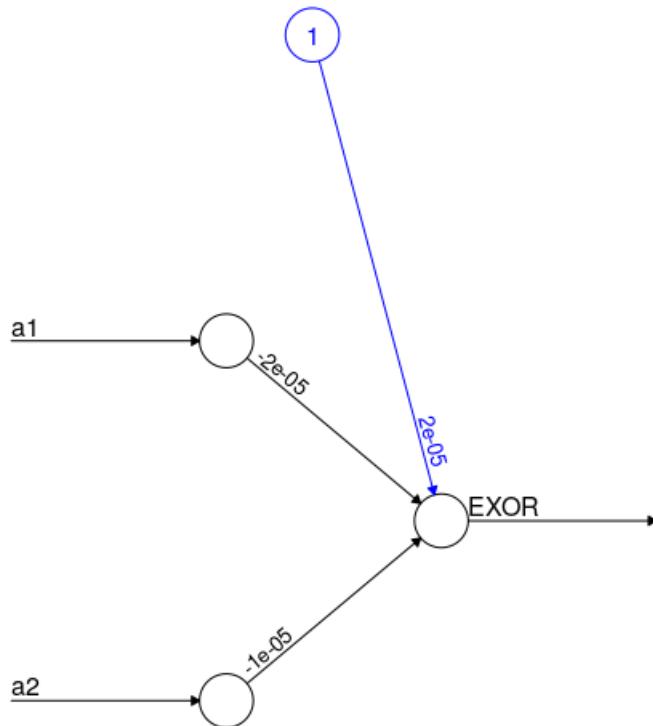
exor.data <- data.frame(a1, a2, EXOR)

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

plot(nn.exor)

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

A kizártó vagy példa az alábbi hálót fogja legenerálni:



Error: 0.5 Steps: 76

Végül játszunk egy kicsit a **hidden** részünkkel, mivel eddig csak a háló azon részeit láttuk, amiket lekérdeztünk, tehát nem láttunk rejtett neuronokat. Így viszont beleláthatunk kicsit jobban a rendszerbe, átadunk neki egy listát, hogyan is rajzolja ki a többrétegű hálónkat: **hidden=c(6, 4, 6)**. Nézzük is hogyan néz ki ez a kódon belül:

```

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

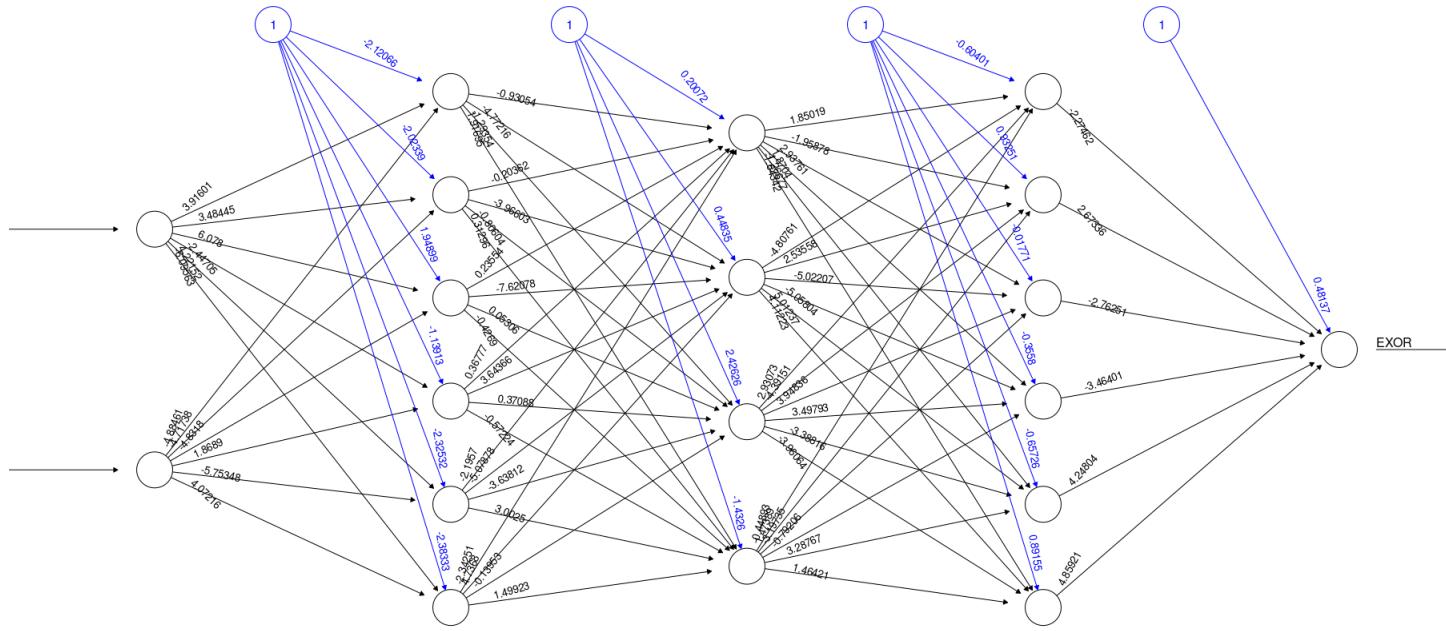
exor.data <- data.frame(a1, a2, EXOR)

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

plot(nn.exor)

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

Így már az ábránk is kicsit érdekesebb lesz:



## 4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

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

SKIP

## 5. fejezet

# Helló, Mandelbrot!

### 5.1. A Mandelbrot halmaz

Megoldás videó:

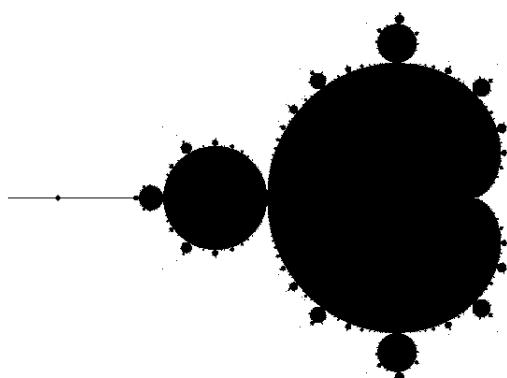
Megoldás forrása: [mandelpngt.c++](#)

Mielőtt kifejezetten a Mandelbrot-halmazról beszélünk előbb egy kicsit a fraktálokról, mint geometriai alakzatokról. A fraktálok végtelenül komplex geometriai alakzatok, melyek két gyakori, jellemző tulajdonsággal rendelkeznek: Az első, hogy a fraktálok határoló vonalai vagy felületei végtelenül "gyűröttek" vagy "érdesek". A második gyakori jellemzőjük a problematikus, de jól érzékelhető önhasonlóság. A Mandelbrot-halmaz azon c komplex számokból áll, melyekre az alábbi (komplex szám értékű)  $X_n$  rekurzív sorozat:

$$x_1 := c$$

$$x_{n+1} := (X_n)^2 + c$$

nem tart végtelenbe, azaz abszolút értékben korlátos. A halmazt Benoît Mandelbrot fedezte fel 1980-ban a komplex számsíkon. A Mandelbrot-halmazt úgy tudjuk megjeleníteni, hogy az origóra húzunk egy pl. 800x800-as rácsot, majd a rács minden pontján megnézzük, melyik komplex számnak felelnek meg. Ha jól csináltuk, az alábbi ábrát kell kapnunk:



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

Megoldás videó:

Megoldás forrása: [3.1.2.cpp](#)

Az előző feladat alapján megnézzük, hogyan is néz ki ez az algoritmus C++ nyelven megírva. Include-oljuk az alap "iostream" library-t; a "png++/png.hpp" könyvtárat, mivel képpel foglalkozunk; majd a lényeget, a "`complex`" könyvtárat. Bekérjük az argumentumokat a felhasználótól (mint például az ablakunk szélességét és magasságát, vagy az iterációs határt), majd készítünk egy üres képet, amin elkezdjük rajzolni a halmazunkat. Ez végigmegy a sorokon és oszlopokon egy léptetéses `for()` ciklussal, és a háló rácspontjaival megegyező komplex számokat kirajzolja.

## 5.3. Biomorfok

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

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

Mivel még mindig a Mandelbrot-halmazzal foglalkozunk, ezért a programunk forráskódja szerkezetileg hasonlítani fog az előző feladatban látott kódra. Mindezt az iterációk módosításával hozzuk létre. A biomorfokról bővebben [ezen](#) a linken olvashatunk, és ebből is fogunk dolgozni. Ugyanazokat a könyvtárakat használjuk, mint az előző feladatban, ugyanazokat az argumentumokat használjuk, de ebben a verzióban argumentumként kérjük be a komplex számsíkért felelő adatokat, nem pedig utólag deklaráljuk őket. Ezek után a szerkezet még ugyanaz marad, csinálunk egy üres képet, majd oszlopunként és soronként haladva kirajzoljuk a kért képünket; pont úgy, mint az előző feladatban. A képletet változtathatjuk; próbálkozhatunk, hogy kijön-e nekünk is ugyanaz az ábra, mint amit referenciajának használtunk a pdf-ből.

## 5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása: [mandelpngc\\_60x60\\_100.cu](#)

A CUDA megvalósítás C++ alapokon működik, ezért nem csoda, hogy hasonlít a C++ kódunkhoz. A változtatás benne az, hogy nem a CPU-t terheljük a kódunkkal, hanem az nVidia kártyánk CUDA magjait, amely gyorsabb végrehajtást nyújt, mint a processzorunk. Ezt viszont tesztelni is szeretnénk, ezért megírjuk a CUDA-kompatibilis kódot, amit majd nvcc-vel kell fordítanunk. A legfőbb változtatás az alábbi kódcsipet beírásával történik:

```
void
cudamandel (int kepadat [MERET] [MERET])
{
    int *device_kepadat;
    cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int));

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

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

cudaMemcpy (kepadat, device_kepadat,
            MERET * MERET * sizeof (int), cudaMemcpyDeviceToHost);
cudaFree (device_kepadat);

}
```

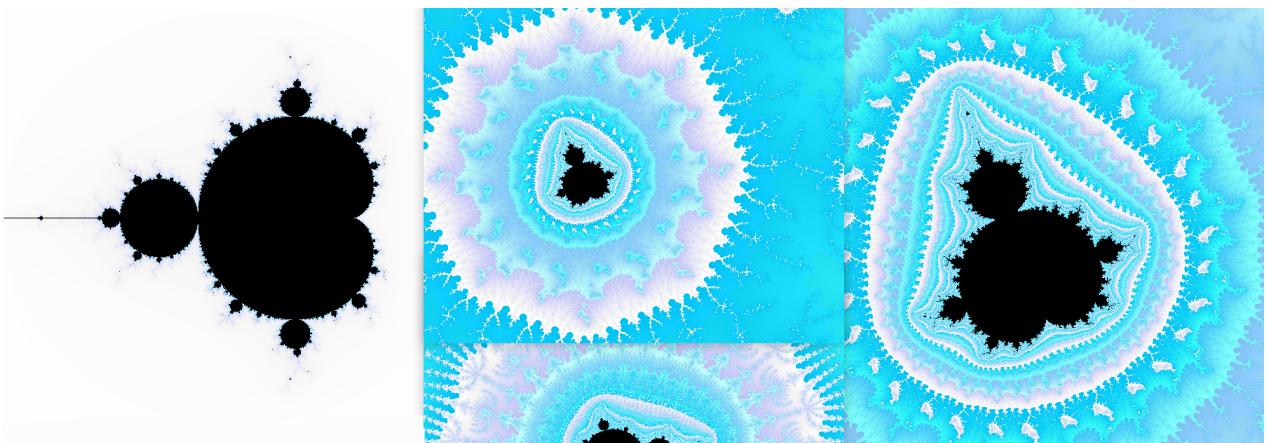
## 5.5. Mandelbrot nagyító és utazó C++ nyelven

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: [Frak](#)

Megoldás videó:

Hogy a forrásainkból képek legyenek, szükségünk lesz a libqt4-dev csomagra. Telepítés után a mappánkba lépve beírjuk a terminálba, hogy qmake, ez generálni fog nekünk egy Makefile-t, ezután egyszerűen egy make parancssal le is tudjuk generálni a dolgokat. Futtatás után négy képet dob ki a program, amelyekből az első az alap Mandelbrot-halmaz, a másik három pedig random nagyítások a halmazon belül.



## 5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás forrása: [MandelbrotHalmaz.java](#) [MandelbrotHalmazNagyító.java](#)

A kód hasonlóan működik a C++ nyelven írt Mandelbrot halmazhoz, de ezzel a programmal tudunk kicsit szórakozni vele, mert ezzel interaktívan tudunk belépni a halmazba. Fordítani a **javac MandelbrotHalmazNagyító.java** sorral tudjuk, majd **java MandelbrotHalmazNagyító.java**-val futtatjuk. Ahhoz, hogy a nagyító működjön, előbb a halmazt kell létrehoznunk, amit a **MandelbrotHalmaz.java** programban írtunk meg. A program interaktív része akkor mutatkozik meg, amikor kijelölünk az alap halmazon egy területet, és a program újraszámítja a halmazt, és csak az előzőleg kijelölt területet látjuk. Ezzel eljátszhatunk a végtelesig, mivel a fraktálok végtelenül komplex alakzatok. Elöször is elindítjuk az egér kattintásainak feldolgozását az alábbi kódcsipettel:

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

A terület kijelölése és feldolgozása úgy történik, hogy bekérjük a kijelölendő terület bal felső sarkát:

```
public void mousePressed(java.awt.event.MouseEvent m) {
    x = m.getX();
    y = m.getY();
    mx = 0;
    my = 0;
    repaint();
}
```

Majd húzással kijelöljük a nagyítandó területet, amit utána újraszámítunk:

```
public void mouseReleased(java.awt.event.MouseEvent m) {
    double dx = (MandelbrotHalmazNagyító.this.b
                 - MandelbrotHalmazNagyító.this.a)
                /MandelbrotHalmazNagyító.this.szélesség;
    double dy = (MandelbrotHalmazNagyító.this.d
                 - MandelbrotHalmazNagyító.this.c)
                /MandelbrotHalmazNagyító.this.magasság;
    new MandelbrotHalmazNagyító(MandelbrotHalmazNagyító.this.a+x*dx,
                                  MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,
                                  MandelbrotHalmazNagyító.this.d-y*dy-my*dy,
                                  MandelbrotHalmazNagyító.this.d-y*dy,
                                  600,
                                  MandelbrotHalmazNagyító.this.iterációsHatár);
}
```

});

## 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 video:

Megoldás forrása: [Polargen C++](#); [Polargen Java](#)

Remek OO bevezető példa lehet egy alább itt is bemutatásra kerülő polártranszformációs normális generátor megírása Javában. A java verzió megvalósításához szükségünk lesz az OpenJDK 6-os verziójára, mindenki meg is tudjuk, miért. A programunk megírása után a JDK src.zip állományában a java/util/Random.java forrásban láthatjuk, hogy magában a JDK-ban is hasonlóan oldották meg a feladatot a Sun programozói!

```
private double nextNextGaussian;
private boolean haveNextNextGaussian = false;

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

Elkészítjük a módosított polármódszeres algoritmust Javaban, majd C++ nyelven. Az algoritmusunk matematikai háttere most számunkra lényegtelen, fontos viszont az eljárás azon jellemzője, hogy egy számítási lépés két normális eloszlású számot állít elő, tehát minden páratlanadik meghíváskor nem kell számolnunk, csupán az előző lépés másik számát visszaadnunk. Ha a programot lefordítjuk és futtatjuk, akkor a következő kimenetet kaphatjuk:

```
-0.7302435745349951
0.3398973333782606
-0.1745186408410782
-0.6733291138289893
-0.7141255333702377
0.8105205642319349
-0.2166963741203095
-0.6100935726625737
-0.0061257158500475665
0.09213084665478943
```

## 6.2. LZW

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

Megoldás videó:

Megoldás forrása: [LZWbinFa](#)

Fa alatt egy olyan rekurzív adatszerkezetet értünk, amely belső és külső csúcsok hierarchikus (szülögyermek) elrendezéséből áll. A bináris fa legfőbb jellemzője az, hogy mindegyik csomópontnak csak két utóda lehet. A bináris fák utódjait megkülönböztetjük aszerint, hogy bal illetve jobb részfák. A programunk úgy működik, hogy megadunk neki egy szöveges fájlt, a program pedig lebontja a karaktereket binfa szerkezetre, ám hogy ezt megtekintsük, meg kell adnunk neki egy kimenetet is, hogy később megnézhessük azt.

```
typedef struct binfa
{
    int ertekek;
    struct binfa *bal_nulla;
    struct binfa *jobb_egy;
} BINFA, *BINFA_PTR;

BINFA_PTR
uj_elem ()
{
    BINFA_PTR p;

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

```
}
```

A fentebbi kódrészletben definiáltuk a binfa struktúrát, valamint a BINFA\_PTR-t, ami a felhasználó által definiált típus lesz. A következőkben megnézzük, hogyan is épül fel a binfa:

```
int
main (int argc, char **argv)
{
    char b;
    int egy_e;
    int i;
    unsigned char c;

    BINFA_PTR gyoker = uj_elem ();
    gyoker->ertek = '/';
    gyoker->bal nulla = gyoker->jobb_egy = NULL;
    BINFA_PTR fa = gyoker;
    long max=0;
    while (read (0, (void *) &b, sizeof(unsigned char)))
    {
        for(i=0;i<8; ++i)
        {
            egy_e= b& 0x80;
            if ((egy_e >>7)==0)
                c='1';
            else
                c='0';
        }
//        write (1, &b, 1);
        if (c == '0')
        {
            if (fa->bal nulla == NULL)
            {
                fa->bal nulla = uj_elem ();
                fa->bal nulla->ertek = 0;
                fa->bal nulla->bal nulla = fa->bal nulla->jobb_egy = NULL;
                fa = gyoker;
            }
            else
            {
                fa = fa->bal nulla;
            }
        }
        else
        {
            if (fa->jobb_egy == NULL)
            {
                fa->jobb_egy = uj_elem ();
                fa->jobb_egy->ertek = 1;
                fa->jobb_egy->bal nulla = fa->jobb_egy->jobb_egy = NULL;
                fa = gyoker;
            }
        }
    }
}
```

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

printf ("\n");
kiir (gyoker);
```

Fentebb láthatjuk, hogyan is működik a bináris fa. Létrehozzuk a gyökeret, majd balra és jobbra is elkezdjük létrehozni a gyerekeket. Hogy megtudjuk pontosan mennyi származtatásról is beszélünk, meg kell néznünk az átlagos ághosszat, vagyis a szórás összegét, az átlagot és a mélységet.

```
extern int max_melyseg, atlagosszeg, melyseg, atlagdb;
extern double szorasosszeg, atlag;

printf ("melyseg=%d\n", max_melyseg - 1);

/* Átlagos ághossz kiszámítása */
atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
ratlag (gyoker);
// atlag = atlagosszeg / atlagdb;
// (int) / (int) "elromlik", ezért casoljuk
// K&R tudatlansági védelem miatt a sok () :)
atlag = ((double) atlagosszeg) / atlagdb;

/* Ághosszak szórásának kiszámítása */
atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
szorasosszeg = 0.0;

rszoras (gyoker);

double szoras = 0.0;

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

printf ("atlag=%f\nszoras=%f\n", atlag, szoras);

szabadit (gyoker);
}
```

Az **atlag()** függvényt inicializálnunk kell, mert azt a rekurzív bejárás használja:

```
int atlagosszeg = 0, melyseg = 0, atlagdb = 0;
void
ratlag (BINFA_PTR fa)
{
    if (fa != NULL)
    {
        ++melyseg;
        ratlag (fa->jobb_egy);
        ratlag (fa->bal nulla);
        --melyseg;

        if (fa->jobb_egy == NULL && fa->bal nulla == NULL)
        {
            ++atlagdb;
            atlagosszeg += melyseg;
        }
    }
}
```

A szórással is eljátsszuk ugyanezt, tehát inicializálunk:

```
double szorasosszeg = 0.0, atlag = 0.0;

void
rszoras (BINFA_PTR fa)
{
    if (fa != NULL)
    {
        ++melyseg;
        rszoras (fa->jobb_egy);
        rszoras (fa->bal nulla);
        --melyseg;

        if (fa->jobb_egy == NULL && fa->bal nulla == NULL)
        {
            ++atlagdb;
            szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));
        }
    }
}
```

```
}
```

A következő kód részletben megírjuk a **kiir** függvényt, ami az inorder fabejárásért felelős, tehát az outputban a gyökér valahol a gyerekek között lesz.

```
int max_melyseg = 0;

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

Végül felszabadítjuk az elemeket:

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

## 6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása: [pre](#); [post](#)

Nem kell nagyon belenyúlni a binfa programba, mikor a fabejárást szeretnénk állítani. az inorder részt már az előző részben elmagyaráztuk, ennek alapján már nem lesz nehéz kitalálni, mire való a másik két bejárási módszer. a preorder azt jelenti, hogy a gyökeret az outputban elsőként írja fel, majd utána írja a többi

csoñópontot(vagy gyereket). A kódban egyszerűen annyit kell átírnunk, hogy felcseréljük a csoñópontok kiírásának sorrendjét. A preorder tehát így néz ki:

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

A postorder pedig így:

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

## 6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása: [z3a7.cpp](#)

Az LZW binfa C++ implementációja azzal kezdődik, hogy az egész fát egy **LZWBinFa** osztályba ágyazzuk. Az osztály definíciójába beágyazzuk a fa egy csomópontjának jellemzését, ez lesz a beágyazott Csomopont osztály. Azért ágyazzuk be, mert külön nem szánunk neki szerepet, csak a fa részeként számolunk vele.

```
class LZWBinFa
{
public:

    LZWBinFa () :fa (&gyoker)
    {
    }
    ~LZWBinFa ()
    {
        szabadit (gyoker.egyesGyermek ());
        szabadit (gyoker.nullasGyermek ());
    }
}
```

Ebben az osztályban a fa gyökere nem pointer, hanem a '/' betűt tartalmazó objektum; a fa viszont már pointer, mindenkor az épülő LZW-fa azon csomópontjára mutat, amit az input feldolgozásakor az LZW algoritmus diktál. Ez a konstruktor annyit csinál, hogy a fa mutatót ráállítja a gyökérre. Tagfüggvényként túlterheljük a << operátort, mert ekkor úgy nyomhatjuk a fába az inputot, hogy **binFa << b**; ahol a b egy '0' vagy '1'-es karakter. Mivel ezen operátor tagfüggvény, így van rá "értelmezve" az aktuális (this "rejtett paraméterként" kapott) példány, azaz annak a fának amibe éppen be akarjuk nyomni a b betűt a tagjai ("fa", "gyoker") használhatóak a függvényben.

```
void operator<< (char b)
{
    // Mit kell betenni éppen, '0'-t?
    if (b == '0')
    {
        /* Van '0'-s gyermeke az aktuális csomópontnak?
        megkérdezzük Tőle, a "fa" mutató éppen reá mutat */
        if (!fa->nullasGyermek ()) // ha nincs, hát akkor csinálunk
        {
            // elkészítjük, azaz páldányosítunk a '0' betű akt. ←
            // parammá
            Csomopont *uj = new Csomopont ('0');
            // az aktuális csomópontnak, ahol állunk azt üzenjük, hogy
            // jegyezze már be magának, hogy nullás gyereke mostantól ←
            // van
            // küldjük is Neki a gyerek címét:
            fa->ujNullasGyermek (uj);
            // és visszaállunk a gyökérre (mert ezt diktálja az alg.)
            fa = &gyoker;
        }
        else // ha van, arra rálépünk
    }
```

```
// azaz a "fa" pointer már majd a szóban forgó gyermekre ↪
// mutat:
fa = fa->nullasGyermek ();
}
}
// Mit kell betenni éppen, vagy '1'-et?
else
{
    if (!fa->egyesGyermek ())
    {
        Csomopont *uj = new Csomopont ('1');
        fa->ujEgyesGyermek (uj);
        fa = &gyoker;
    }
    else
    {
        fa = fa->egyesGyermek ();
    }
}
}
```

## 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: [z3a7\\_pointer](#)

Alapul vesszük az előző C++ kódot, ebben fogjuk átírni a szükséges dolgokat. Először átírjuk a **protected** osztályban a gyökér csomópontot pointerré:

```
Csomopont *gyoker;
```

Majd elkezdhetjük a többi kód részlet átírását, például a legelején a konstruktora Kezdjük már is a legelején, a konstruktort írjuk át erre:

```
LZWBinFa (gyoker)
{
    fa = gyoker;
}
```

Szintén a konstruktornban, mivel már mutató a gyökér, ezért a **szabadít()** függvényünkbe is bele kell nyúlnunk kicsit, mutatót állítunk a gyermekekre:

```
~LZWBinFa ()
{
    szabadít (gyoker -> egyesGyermek ());
    szabadít (gyoker -> nullasGyermek ());
}
```

Végül a **&gyoker** kifejezéseket kicséréljük simán **gyoker**-re, ugyanis nekünk nem a pointer címére lesz szükségünk, hanem arra a címre amire az mutat.

## 6.6. Mozgató szemantika

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

Tutor: Győri Márk Patrik

Megoldás forrása: [mozgato](#)

```
LZWBInFa & operator= (const LZWBInFa & cp) {
    if(&cp != this)
        rekurzioInditasa(cp.gyoker);
    return *this;
};
```

Létrehozunk egy operátort, ami ha nem a gyökeret tartalmazza, akkor lemásolja saját magát. A másolás rekurzióval végződik, ami a fa minden ágát újra létrehozza egy másik gyökérre.

```
void rekurzioInditasa(Csomopont csm) {
    if(csm.nullasGyermekek()) {
        fa = &gyoker;
        Csomopont *uj = new Csomopont ('0');
        fa->ujNullasGyermekek(uj);
        fa = fa->nullasGyermekek();
        std::cout << "GYOKER: nullas van" << std::endl;
        rekurzioAzAgakon(csm.nullasGyermekek());
    }
    if(csm.egyesGyermekek()) {
        fa = &gyoker;
        Csomopont *uj = new Csomopont ('1');
        fa->ujEgyesGyermekek(uj);
        fa = fa->egyesGyermekek();
        std::cout << "GYOKER: egyes van" << std::endl;
        rekurzioAzAgakon(csm.egyesGyermekek());
    }
}

void rekurzioAzAgakon(Csomopont * csm) {
    if (csm->nullasGyermekek()) {
        std::cout << "====van nullas" << std::endl;
        Csomopont *uj = new Csomopont ('0');
        fa->ujNullasGyermekek(uj);
    }
    if (csm->egyesGyermekek()) {
        std::cout << "====van egyes" << std::endl;
        Csomopont *uj = new Csomopont ('1');
        fa->ujEgyesGyermekek(uj);
```

```
    }
    Csomopont * nullas = fa->nullasGyermek();
    Csomopont * egyes = fa->egyesGyermek();
    if(nullas) {
        fa = nullas;
        rekurzioAzAgakon(csm->nullasGyermek());
    }
    if(egyes) {
        fa = egyes;
        rekurzioAzAgakon(csm->egyesGyermek());
    }
}
```

A **rekurzioInditasa** függvény indítja el a rekurziót, ha van nullás gyermeke akkor azon fut tovább, ha van egyes gyermeke, akkor arra is meghívásra kerül. A fő eljárást maga a **rekurzioAzAgakon** függvény végzi, ez fut át az összes ágon, és létrehozza az új node-okat.

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

A másolás már csak az "**=**" jel operátorral meghívva történik, így az alap **binFa** átmásolódik a **binFa2**-be.

## 7. fejezet

# Helló, Conway!

### 7.1. Hangyszimulációk

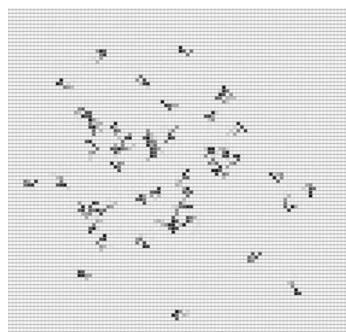
Írj Qt C++-ban egy hangyszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

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

Tutor: Győri Márk Patrik

Megoldás forrása: [ant](#)

A **main** függvény bekéri az argumentumokat, majd deklarálja az **AntWin** osztályt (w) a bemeneti argumentumokkal, amely felelős a megjelenítésért. Az **AntThread** osztály végzi a számolást és a "hangyák" lehelyezését. Itt az **AntThread::newDir** számolja ki, hogy melyik irányba menjen a hangya. Ennek a számolása a **AntThread::moveAnts** függvénytel történik. Az **AntThread** külön szálon fut, így megállítható, és újra indítható. Erre bindeltük a "P" betűt, amit a keyPressEvent-nél találunk.



### 7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása:

**SKIP**

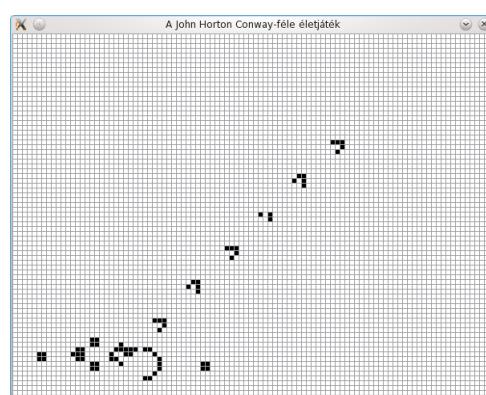
## 7.3. Qt C++ életjáték

Most Qt C++-ban!

Tutor: Győri Márk Patrik

Megoldás forrása:

Az elvet John Horton Conway dolgozta ki. A lényege, ha egy cella egyedül van, vagy nincs szomszédja, akkor eltűnik. Ha négy, vagy több szomszédja van akkor is eltűnik. Ha kettő vagy három szomszédja van, akkor megmarad. minden cella, aminek három szomszédja van az tovább osztódik. Ez a műveletsor ismétlődik végtelenszer. Ha például egy 2x2-es cella van, és nincsen szomszédja akkor nem történik semmi, mivel alkalmazzuk rá a fenti szabályokat.

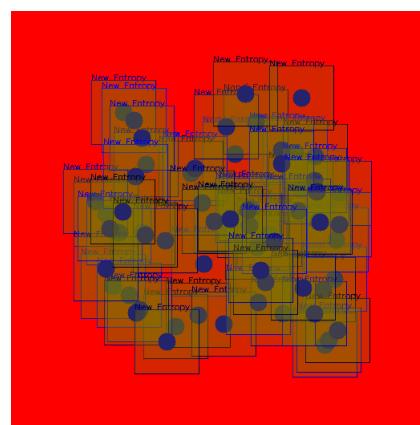


## 7.4. BrainB Benchmark

Tutor: Győri Márk Patrik

Megoldás forrása: [BrainB](#)

A játék lényege, hogy a **Samu Entropy** nevezetű négyzetet kell megkeresnünk a többi négyzet közül, és benne kell tartanunk a kurzort a kék körben. A konzol kiírja, ha véletlen hibáznánk, és méri, hogy mennyi bits/sec a reakció időnk. Egyre több hamis entropy jelenik meg a képernyőn, szóval egyre nehezebb lesz a dolgunk



## 8. fejezet

# Helló, Schwarzenegger!

### 8.1. Szoftmax Py MNIST

Python

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

Megoldás forrása: <https://github.com/tensorflow/tensorflow/releases/tag/v0.9.0> (/tensorflow-0.9.0/tensorflow/exa...  
[https://progpater.blog.hu/2016/11/13/hello\\_samu\\_a\\_tensorflow-bol](https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol)

Az SMNIST egy gépi tanulásra szóló program, ezért tensorflow-t használunk alapnak. Ezen program azt szolgálja, hogy megszámolja, hány fekete négyzetet lát egy adott képen. Ehhez le kell generálnunk pár képet, legyen ez most 60000 darab kép, hogy minél kevesebb legyen az ismétlődés.

### 8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása: [https://github.com/tensorflow/tensorflow/blob/r1.4/tensorflow/examples/tutorials/mnist/mnist\\_deep.py](https://github.com/tensorflow/tensorflow/blob/r1.4/tensorflow/examples/tutorials/mnist/mnist_deep.py)

Tanulságok, tapasztalatok, magyarázat...

### 8.3. Minecraft-MALMÖ

Megoldás videó: <https://youtu.be/bAPSu3Rndl8>

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

## 9. fejezet

# Helló, Chaitin!

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

Megoldás videó:

Megoldás forrása:

A Lisp-hez elegendő egy GIMP-et megnyitnunk, azon belül pedig a Script-fu konzolt kell elindítanunk, majd kezdődhet is a Lisp-es kalandunk. Még mielőtt a faktoriálist írnánk meg, kicsit ismerkedjünk a Lisp nyelvvel. A lényege az, hogy ha azt akarjuk kideríteni, mennyi  $2+2+2$ , akkor csak annyit írunk, hogy  $(+ 2 2)$ . Tudunk függvényt define-olni, például így: (**define (fakt n)**). A lényeg, hogy figyeljünk a zárójelekre, valamint a szóközökre, mert el lehet veszni bennük. Nézzük a várva várt faktoriális megoldást:

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

Ezután ha beírjuk például a (**fakt 5**) függvényt, akkor 120-at kapunk vissza, ami megfelel az **5!**-nak

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

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

Megoldás videó: [https://youtu.be/OKdAkl\\_c7Sc](https://youtu.be/OKdAkl_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)

Ahhoz, hogy futtatni tudjuk GIMP-en belül a létrehozott scriptünket, be kell másolni a kódot a GIMP script mappájába. Majd a programon belül: **File/Create/BHAX/bhax\_chrome3** és már szórakozhatunk is kedvünkre a kívánt szövegünk átszínezésével. Ha futtatjuk a scriptet, egy alap beállítást is kapunk, amely megadja, hogy 1000x1000 képpontos kép legyen, valamint beállítja a fontot és a font méretét. A színt és a színátmenetet is beállíthatjuk. Nézzük a kód fő lépésein, ami alapján felépül a króm effekt:

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

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

(set! layer (car (gimp-image-merge-down image textfs ←
    CLIP-TO-BOTTOM-LAYER)))
```

Az első lépésben megadunk egy fekete színnel kitöltött hátteret, majd jelezzük, hogy a szövegünk fehér színű legyen, a krómósítást pedig később fogjuk elvégezni. A második lépésben az ún. Gaussian módszerrel elmosssuk a szöveget:

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

Majd a harmadik lépésben lekanyarítjuk a szöveg éleit: (gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE) Végül ezt is elmosssuk: (plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE) Invertáljuk a képet, kitöröljük a fekete hátteret, így kapunk egy átlátszót, majd foglalkozunk a szöveggel. A lényeg a kilencedik lépés, ekkor egy Spline görbüvel meg tudjuk adni a gradient effektet: (gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))

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

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

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

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

A script felépítése hasonló az előző feladatban látott script felépítéséhez, ugyanúgy script-fu-t használunk, tehát GIMP-el foglalkozunk még mindig. A mandala lényege, hogy tükrözgetjük és forgatjuk a szövegünket, amíg egy szép kis képet nem kapunk belőle:

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

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

  (set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
  (gimp-image-insert-layer image text-layer 0 -1)
  (gimp-item-transform-rotate text-layer (/ *pi* 2) TRUE 0 0)
```

```
(set! textfs (car(gimp-image-merge-down image text-layer ←
CLIP-TO-BOTTOM-LAYER)))

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

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

(plug-in-autocrop-layer RUN-NONINTERACTIVE image textfs)
(set! textfs-width (+ (car(gimp-drawable-width textfs)) 100))
(set! textfs-height (+ (car(gimp-drawable-height textfs)) 100))
```

Majd ezután két kört rajzolunk, amelyeknek eltérő a vastagságuk (22 és 8):

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

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

(set! textfs-width (- textfs-width 70))
(set! textfs-height (- textfs-height 70))

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

(gimp-context-set-brush-size 8)
(gimp-edit-stroke textfs)
```

# 10. fejezet

## Helló, Gutenberg!

### 10.1. Programozási alapfogalmak

Kezdjük az alapfogalmakkal: A programozási nyelveket három szintre tudjuk bontani: gépi nyelv, assembly szintű nyelv és magas szintű nyelv. Mi az utolsó, magas szintű nyelvvel fogunk foglalkozni. Az ilyen szinten írt programokat forrásprogramnak nevezzük. A magas szintű nyelven írt forrásból gépi nyelvű programot kell alkotnunk; erre létezik interpretes és fordítóprogramos megoldás is. A **fordítóprogram** egy olyan program, amely a forrásunkból tárgyprogramot állít elő, amit tud kezelní a gép. Négy lépést hajt végre ez a folyamat, a lexikális elemzést, szintaktikai elemzést, szemantikai elemzést, majd legenerálja a kódot. Az interpretes megoldásnál is ugyanaz az első három lépés, de az interpreter nem csinál belőle tárgyprogramot. minden utasításnál sorra veszi a forrást, értelmezi majd végrehajtja azt. Itt egyből megkapjuk az eredményt egy gépi rutin szerint. A programnyelveket osztályozni tudjuk: Vannak imperatív nyelvek, deklaratív nyelvek, és másnyelvű nyelvek.

Beszéljünk a kifejezésekről. A kifejezések olyan szintaktikai eszközök, amik arra valók, hogy a program adott pontján ismert értékekből új értéket tudunk meghatározni. Ennek két komponense van, érték és típus komponens. Egy kifejezés három összetevőből áll: operandusok, operátorok és kerek zárójelek. A kifejezéseknek három alakja lehet, attól függ, hol áll az operátor az operandusokhoz képest: **prefix**, amikor az operátor az operandusok előtt van; **infix**, amikor az operátor az operandusok között helyezkedik el; valamint **postfix**, amikor az operátort az operandusok mögé tesszük. Ezek közül az infix alak nem minden egyértelmű. Az utasításokról: 9 csoportra bonthatjuk őket: Értékadó, üres, ugró, elágaztató, ciklusszervező, hívó, vezérlésátadó, I/O valamint egyéb utasítások. Az elágaztató utasításokat is tovább bonthatjuk: Kétirányú elágaztató utasítás, vagyis feltételes utasítás (IF ... THEN); Többirányú elágaztató utasítás, ennek szintaktikája és szemantikája nyelvenként különböző. A ciklusszervező utasításoknak köszönhetően egy tevékenységet bármennyiszer megismételhetünk. Ezek felépítése a fejből, magból és a végből épül fel. A feltételes ciklusnak is két fajtája van: Kezdőfeltételes ciklus, amikor a feltétel a fejben jelenik meg; valamint végfeltételes ciklus, amikor a feltétel a végen jelenik meg.

A programok szerkezetét nézegetve felmerül a kérdés, hogy egyben kell-e lefordítani a teljes szöveget, vagy fel lehet tördelni azt? Erre három válasz is létezik. Néhány nyelven a program önálló részeiből áll, ezek külön fordíthatóak, nem struktúráltak. Más nyelvekben az adott programot egy egységekkel kell lefordítanunk. Ez a program struktúrálható, az egységei nem függetlenek. Az utolsó esetként a kettő kombinációja is létezik. Ezek független programegységek, de tetszőleges struktúrával rendelkeznek.

Paraméterátadás. Ez a programegységek és az alprogramok közötti kommunikáció eszköze. Itt minden van egy hívó és egy hívott. Ezek a módok léteznek paraméterátadásra: érték, cím, érték-eredmény,

név és szöveg szerinti paraméterátadás. Az aktuális paraméterek lehetnek az alábbiak: érték szerinti: kifejezés -- cím szerinti: változó -- eredmény szerinti: változó -- érték-eredmény szerinti: változó. A következő témánk a blokk. Ez olyan programegység, amely csak és kizárolag más programegység belsejében helyezkedik el, kívül nem állhat. A blokk három részből épül fel; van kezdete, törzse és vége. A kezdetet jelzi egy szó vagy speciális karakterek; A törzsben végrehajtó és deklarációs utasítások is előfordulhatnak. A blokkoknak nincsenek paraméterei, bárhol elhelyezhető, ahol végrehajtó utasítás van. Egy blokkot többféleképp is tudunk aktivizálni.

Nézzük mit érdemes tudni az I/O-ról. Ez a programnyelvek azon területe, ahol a leginkább eltérnek a többi programnyelvtől. Az I/O olyan eszközrendszer, ami a perifériákkal kommunikál. Ennek középpontjában az állomány van. Az adott állomány funkció szerint tekintve lehet input állomány, output állomány valamint mindenkettő, tehát input-output állomány. Nézzünk pár példát, hogy egyes nyelvek hogyan kezelik ezeket az I/O eszközöket. Például a FORTRAN csak fix rekordformátumot képes kezelní, a COBOL-nak erős az I/O rendszere, mert minden konvertál. A PL/I a legjobb állománykezeléshez. Ezzel ellentétben a PASCAL állománykezelése nem túl jó. Ami számomra releváns, az a C, ennek a nyelvaz I/O eszközrendszer nem

## 10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

Elsőként a vezérlési szerkezetekkel foglalkozunk. A vezérlésátadó utasítások azt határozzák meg, milyen sorrendben hajtsa végre a számítógép a számításokat. Foglalkozzunk a különféle utasításokkal, annak típusaival. Az első ilyen típus az **egyszerű kifejezések**, az is lehet utasítás ha egy kifejezést, például egy **x = 1**-et pontosvesszővel zárunk. A kapcsos zárójelekkel behatárolt deklarációkat egy blokknak, összetett utasításnak veszünk. A következő típus az **if-else** és az **else if** utasítások. Az if-else utasítással választást, döntést írunk le, míg az else if megoldással többszörös elágazást készíthetünk. Ha több irányba el akarjuk ágaztatni a programot, ahhoz a **switch** utasítás remek eszköz. A switch kiértékeli azt a kifejezést, amit a zárójelek közé rakunk, majd megnézi, hogy az adott érték megegyezik-e a case-ben tartott értékkal; és ha talál ilyet, végrehajtja a case-en belüli utasításokat. Ezzel megspórolhatunk jónapot if-else utasítást. minden case-en belüli utasítás végére kell egy **break**; függvény, ezzel tudatjuk a programmal, hogy itt ér véget az adott utasításunk. Ha megírtuk az összes case-t amit szerettünk volna, meg kell adnunk egy **default** értéket, amit akkor ír ki a program, ha egyik case-nek sem felel meg az értékünk.

A következő, amivel foglalkozunk, az a **for** és a **while** utasítások. A **while()** ciklusban a gépünk kiértékeli a zárójelben lévő kifejezést, és ha az nem nulla, akkor végrehajtja a kiadott utasítást, majd újra kiértékeli a kifejezést. Ez a ciklus addig megy, amíg az érték 0 nem lesz. A **while** ciklus általános szerkezete:

```
while (valami)
    utasítások
```

A **for** szerkezetre példa:

```
for (x1, x2, x3)
    utasítások
```

ezt át tudnánk alakítani **while** szerkezetre is, az alábbi módon:

```
x1;
while (x2) {
    utasítások
    x3;
}
```

A **for** utasítás minden összetevője ( $x_1, x_2, x_3$ ) kifejezés. Általában az  $x_1$ ,  $x_3$  függvényhívás vagy értékkadás, az  $x_2$  pedig relációs kifejezés. El lehet hagyni a kifejezések bármelyikét, de a lényeg, hogy a pontosvessző maradjon. Ha a függvényhívás/értékkadás elmarad, akkor a pontosvessző elmarad a kifejtésből; ha viszont az  $x_2$  marad el, akkor igaznak tekinthetjük az alábbi kódot:

```
for(;;) {
    kódcsipet
}
```

ami egy végtelen ciklus. Ki tudunk belőle ugrani egy **break** vagy **return** függvénnel.

A következő amit megnézünk az a **do-while** utasítás lesz. Ez abban különbözik a **while** és a **for** utasításuktól, hogy míg ez a kettő a kiugrási feltétel teljesülését a ciklus elején vizsgálja, addig a **do-while** mindezt a ciklus végén teszi meg. nézzük is hogyan épül ez fel:

```
do
    valami
    while(kif);
```

Először az utasítást(valami) hajtja végre, majd a kifejezést értékeli ki(kif). Ha igaz, akkor végrehajtja az utasítást. Ha hamis, akkor a ciklus véget ér. Ezt a megoldást ritkábban szokás használni.

A soron következő utasítás a **break** utasítás. Ezt azért használjuk, mert kényelmesebb néha úgy kilépni a ciklusból, hogy nem függvényvizsgálattal tesszük azt. A **break** utasítástól a vezérlés egyből kilép a legelső ciklusból vagy switchből.

Az utolsó előtti utasításunk a **continue** utasítás. Ez az utasítás a **break**-hez kapcsolódik, de ritkábban használjuk. A while és a do esetében azonnal vizsgálja a feltételt, a for esetében pedig átlép az újrainicializálás lépére. A **continue** csak a for, while, do ciklusokra alkalmazható, switch-re nem.

A legutolsó utasításunk a **goto** utasítás. Ezzel címkékre tudunk ugratni. Eredetileg erre soha nincs szükségünk, ezért elég is ennyi róla.

## 10.3. Programozás

[BMECPP]

Kezdjük a kivételkezelés alapjaival. A kivételkezelés, angolul exception handling, olyan mechanizmus, ami ha hibát talál valahol akkor a program futása egyből a hibakezelő részen folytatódik. Ne hagyjük azt, hogy csak hibákat tudunk ezzel kiszűrni, mert **kivételkezelésről** beszélünk, és nem **hibakezelésről**. A kivételkezelő bármilyen kivételes helyzetben használható. Próbáljuk ki a kivételkezelést kódban is megnézni. Tegyük fel, hogy osztani akarunk egy olyan számmal, amit a felhasználó ad meg. De mi van, ha a felhasználó 0-t ír? Azt szeretnénk, ha ezt jelezni a felhasználó felé, ezért még mielőtt eljut a felhasználóhoz,

kipróbáljuk magunknak egy **try-catch blokk** alkalmazásával. A lényeg az, hogy a **try** védett blokkba beírjuk úgy a kódot, ahogy szeretnénk, hogy működjön. Ugyanebben a blokkban megpróbálunk dobni neki egy kivételt a **throw** függvényvel, így szeretnénk egy exception-t dobni a kódnak. Az életben is általában úgy van, ha eldobnak egy labdát, nem árt azt elkapni. Ugyanezt csináljuk a kódunkkal, a dobott kivételt elkapjuk egy **catch** blokkban. Dobáskor a futás egyből arra a catch-re ugrik, amely paraméterei megfelelnek a dobott kivétel paramétereivel. A kivétel elkapásakor az be fog íródni a catch paramétere közé, majd lefut a catch törzse. Ha szeretnénk, meg is tekinthetjük, mit kapott el a kivételkezelő, ezt úgy tudjuk megnézni, ha a catch zárójelei közé létrehozunk egy **const char\*** típusú változót. Gyakorlatban így nézne ki maga az elkapás:

```
catch (const char* kivetel)
{
    std::cout << "Hiba: " << kivetel << std::endl;
}
```

A kivételkezelés egyik legfontosabb jellemzője az, hogy ha egy try blokkban függvényhívásokra kerül sor, akármilyen mélyen is van a **throw**, a futás egyből a **catch** ágon megy tovább.

Tudunk olyat is készíteni, hogy egy try-catch blokkhoz **két catch** ágat rendelünk. A throw-nak megadunk egy paramétert, ezt a paramétert **kivételobjektumnak** nevezni. Ha egy kivételt nem sikerül elkapni, kezeletlen kivételnek tekintjük. Ebben az esetben egy függvény kilép az alkalmazásból. Olyat is tudunk csinálni hogy egymásba ágyazunk több **try-catch** blokkot. Ez azért hasznos, mert így néhány kivételt a dobott kivételünkhez közelebb tudjuk kezelni, egy fokkal alacsonyabb szintről.

Foglalkozzunk egy kicsit az operátorokkal. Ezek C nyelven az argumentumaikon végezik el a műveleteket. A C++ nyelvbe vezetették pár új operátort a C-hez képest. Az egyik ilyen a hatókör operátor (::). Ezt már láttuk sokszor C++ programozás során, csak hogy sorolják párat példának: std::cout, mozgató szemantikánál std::move kipróbálása. Újítás még a pointer-tag operátorok: .\* és -\*. A következő a túlterhelésről és a függvény szintaxis. C++ nyelven már egyszer használtuk az **operator**-t, ezzel jelezük a program felé, hogy speciális, saját függvényről van szó. Erre a mozgató szemantikás példát nézzük meg újra:

```
LZWBInFa & operator= (const LZWBInFa & cp) {
    if(&cp != this)
        rekurzioInditasa(cp.gyoker);
    return *this;
};
```

## **III. rész**

# **Második felvonás**

**Bátf41 Haxor Stream**

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

# 11. fejezet

## Helló, Arroway!

### 11.1. OO szemlélet

A módosított polártranszformációs normális generátor beprogramozása Java nyelven. Mutassunk rá, hogy a mi természetes saját megoldásunk (az algoritmus egyszerre két normálist állít elő, kell egy példánytag, amely a nem visszaadottat tárolja és egy logikai tag, hogy van-e tárolt vagy futtatni kell az algot.) és az OpenJDK, Oracle JDK-ban a Sun által adott OO szervezés ua.! <https://arato.inf.unideb.hu/batfai.norbert/UDPROG/> (16-22 fólia) Ugyanezt írjuk meg C++ nyelven is! (lásd még UDPORG repó: source/labor/polargen)

Megoldás forrása: [Polargen C++](#); [Polargen Java](#)

```
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());
    }
}
```

Fontos az eljárás azon jellemzője, hogy egy számítási lépés két normális eloszlású számot állít elő, tehát minden páratlanadik meghíváskor nem kell számolnunk, csupán az előző lépés másik számát visszaadunk. Ha a programot lefordítjuk és futtatjuk, akkor a következő kimenetet kaphatjuk:

```
-0.7302435745349951
0.3398973333782606
-0.1745186408410782
-0.6733291138289893
-0.7141255333702377
0.8105205642319349
-0.2166963741203095
-0.6100935726625737
-0.0061257158500475665
0.09213084665478943
```

## 11.2. „Gagyí”

Az ismert formális „while ( $x \leq t \&& x \geq t \&& t \neq x$ );” tesztkérdéstípusra adj a szokásosnál (miszerint  $x$ ,  $t$  az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciaja) „mélyebb” választ, írj Java példaprogramot mely egyszer végtelen ciklus, más  $x$ ,  $t$  értékekkel meg nem! A példát építsd a JDK Integer.java forrására<sup>3</sup>, hogy a 128-nál inkluzív objektum példányokat poolozza!

Megoldás forrása: [Gagyí.java](#)

```
package gagyi;

import java.lang.Thread;

public class Gagyí {

    public static void main(String[] args) throws InterruptedException {
        Integer x = 127;
        Integer t = 127;
```

```
System.out.println(x == t);

while (x <= t && x >= t && t != x) {
    System.out.println(1);
}

x = 128;
t = 128;

System.out.println(x == t);

while (x <= t && x >= t && t != x) {
    System.out.println("Végtelen ciklus");
    Thread.sleep(3000);
}

}
```

A kódunk elején a `java.lang.Thread`-re azért lesz szükségünk, hogy a végén ne spamelje szét a kimenetet, csak bizonyos időközönként írja ki; ezért kell nekünk a `Thread.sleep()`. Az első while ciklus true értékkel tér vissza, a következő viszont végtelen ciklust idéz elő. Az Integer metódus csak -128 és 127 között tárolja az értékeket, és ezért kapunk végtelen ciklust, amikor 127-nél nagyobb értéket írunk neki.

### 11.3. Yoda

Írunk olyan Java programot, ami `java.lang.NullPointerException`-el leáll, ha nem követjük a Yoda conditions-t! [https://en.wikipedia.org/wiki/Yoda\\_conditions](https://en.wikipedia.org/wiki/Yoda_conditions)

Megoldás forrása: [Yoda.java](#)

A Yoda condition egy olyan programozási stílus, amikor a tipikus sorrendet felcseréljük egy `if` statementben. Elnevezését Yoda sajátos beszédstílusa miatt kapta, pl. *"If blue is the sky"*. Ezt felhasználva az alábbi sorrendet fel tudjuk cserélni erről:

```
if(változó == érték) { /* ... */ }
```

erre:

```
if(érték == változó) { /* ... */ }
```

Nézzük magát a feladatot: A program első része hiba nélkül lefut, a másik pedig felcserélés esetén dob nekünk egy `java.lang.NullPointerException` kivételt.

```
public class Yoda {
    public static void main(String[] args) {
        String vmi = null;
        if ("Valami".equals(vmi))
```

```
System.out.println("Valami".equals(vmi));  
  
if (vmi.equals("Valami"))  
    System.out.println("Valami".equals(vmi));  
}  
}
```

Az ötödik sorig nincs semmi gond, de utána a 7. sorban már láthatjuk megjelenni a dobott kivételt, de nincs ezzel semmi baj, mert pont ezt akartuk tőle. Ennek az az oka, hogy **null** értékű a változónk, és ezt akartuk a szövegünkhez hasonlítani, nem pedig a szöveget a változóhoz.

```
Exception in thread "main" java.lang.NullPointerException  
at Yoda.main(Yoda.java:7)  
  
Process finished with exit code 1
```

## 11.4. Kódolás from scratch

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

Megoldás forrása: [PiBBP.java](#)

A BBP algoritmust az alábbi szerint számoljuk:

```
{16^d Pi} = {4 * {16^d S1} - 2 * {16^d S4} - {16^d S5} - {16^d S6}}  
(a {}) a törtrészt jelöli.)
```

a Pi hexa kifejtésében a d+1. hexa jegytől számoljuk a hexa jegyeket

```
package bpp;  
  
/*  
 * PiBBP.java  
 *  
 * DIGIT 2005, Javat tanítok  
 * Bátfai Norbert, nbatfai@inf.unideb.hu  
 *  
 */  
  
public class Main {  
    String d16PiHexaJegyek;  
  
    public Main(int d) {  
  
        double d16Pi = 0.0d;  
  
        double d16S1t = d16Sj(d, 1);
```

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

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

d16Pi = d16Pi - StrictMath.floor(d16Pi);

StringBuffer sb = new StringBuffer();

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

while (d16Pi != 0.0d) {

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

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

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

d16PiHexaJegyek = sb.toString();
}

public double d16Sj(int d, int j) {

    double d16Sj = 0.0d;

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

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

A továbbiakban bináris hatványozást hajtunk végre,  $16^n \bmod k$ -val, ahol n a kitevő, és k a modulus.

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

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

    long r = 1;

    while (true) {
```

```
if (n >= t) {  
    r = (16 * r) % k;  
    n = n - t;  
}  
  
t = t / 2;  
  
if (t < 1)  
    break;  
  
r = (r * r) % k;  
  
}  
  
return r;  
}
```

A következőben kiíratjuk a kiszámolt hexa jegyeket. Megnézzük az első 6 pontos peldául d=1000000 esetén. A stringünk lesz a kiszámolt hexa jegy. Végül pedig példányosítunk egy BBP algoritmust magába foglaló objektumot.

```
public String toString() {  
    return d16PiHexaJegyek;  
}  
  
public static void main(String args[]) {  
    System.out.print(new Main(1000000));  
}  
}
```

## 12. fejezet

# Helló, Liskov!

### 12.1. Liskov helyettesítés sértése

Írunk olyan OO, leforduló Java és C++ kódcsipetet, amely megsérti a Liskov elvet! Mutassunk rá a megoldásra: jobb OO tervezés. [https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2\\_1.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf) (93-99 fólia) (számos példa szerepel az elv megsértésére az UDPROG repóban, lásd pl. source/binom/Batfai-Barki/madarak/)

Tutor: Veress Máté

A liskov-féle helyettesítési elv (Liskov Substitution Principle) lényege az, hogy minden osztály legyen helyettesíthető a leszármazott osztályával úgy, hogy a program működése ne változzon meg. Ez az elv a S.O.L.I.D alapelvek egyik tagja. Adott egy P programunk. Itt ha S altípusa a T osztálynak, akkor elvileg a T helyére be tudnánk helyettesíteni gond nélkül az S-t is. Ezt az elvet viszont meg lehet sérteni.

A feladat forrását felhasználva elkészítünk egy ellenpéldát C++ nyelven, ami sérti a Liskov elvet, ez egy tipikus madár példa. Adott nekünk a Madarak osztály, amelynek van egy belső függvénye, a `repules()`.

```
#include <stdio.h>

class Madarak {
public:
    char repules() { printf("Repülök\n"); }
};
```

Ezután létrehozunk két alosztályt, (ezek lesznek a T és az S osztályok), amely a Madarakat bővíti ki. Legyen ez például a Sas, valamint a Pingvin:

```
class Sas : public Madarak {};
class Pingvin : public Madarak {};

int main() {
    Sas sas;
    Pingvin pingvin;
    sas.repules();
    pingvin.repules();
}
```

Ha a Sas a madaraktól örököl, akkor ezzel együtt a repules() "képességet" is képes használni, viszont az a gond, hogy a Pingvin is örökli azt. A pingvin nem tud repülni, így a programunk logikáltan alapon működik, erre kell megoldást találnunk. Ezt úgy oldjuk meg, hogy elvesszük a Madarak osztálytól a repülés funkciót, és átrakjuk azt egy **RepuloMadarak** osztályba, hiszen nem minden madár tud repülni, és ez már kezd hajazni a logikánakra. A Sas "megkapja" a repülés képességét, de a Pingvin nem. Na de nézzük is a "javított" kódot:

```
#include <stdio.h>

class Madarak {

//public:
//    char repules() { printf("Repülök\n"); };

};

class RepuloMadarak : public Madarak {

public:
    char repules() { printf("Repülök\n"); };
}

class Sas : public RepuloMadarak {};
class Pingvin : public Madarak {};

int main() {
    Sas sas;
    Pingvin pingvin;
    sas.repules();
    pingvin.repules();
}
```

Bár a logika már így helyes, a programunk sajnos nem fog lefordulni, mivel a Pingvin már nem kapja meg a repülést az osztályból.

## 12.2. Anti OO

A BBP algoritmussal a Pi hexadecimális kifejtésének a 0. pozíciótól számított  $10^6$ ,  $10^7$ ,  $10^8$  darab jegyét határozzuk meg C, C++, Java és C# nyelveken és vessük össze a futási időket! <https://www.tankonyvtar.hu/hu/tartanitok-javat/apas03.html#id561066>

Megoldás forrása:

Ebben a feladatban újra előkerül az előző heti BBP algoritmus feladat, most a Pi hexadecimális kifejtésének a 0. pozíciótól számított  $10^6$ ,  $10^7$ , valamint  $10^8$ . hexa jegyét meghatározzuk, valamint mérjük, melyik nyelv számolja ki a leggyorsabban a jegyeket, a "versenyzők" pedig: Java, C, C++, C#. Az összes kódot nem ömlesztettem a könyvbe, csak a Java verziót, mivel a másik három is lényegében ugyanaz, kisebb változtatásokkal.

```
public class PiBBPBench {  
    /**  
     * BBP algoritmus a Pi-hez, a [BBP ALGORITHMUS] David H. Bailey: The  
     * BBP Algorithm for Pi. alapján a {16^d Sj} részlet kiszámítása.  
     *  
     * @param d a d+1. hexa jegytől számoljuk a hexa jegyeket  
     * @param j Sj indexe  
     */  
    public static double d16Sj(int d, int j) {  
  
        double d16Sj = 0.0d;  
  
        for(int k=0; k<=d; ++k)  
            d16Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);  
  
        /* (bekapcsolva a sorozat elején az első utáni jegyekben növeli pl.  
         * a pontosságot.)  
        for(int k=d+1; k<=2*d; ++k)  
            d16Sj += Math.pow(16.0d, d-k) / (double)(8*k + j);  
        */  
  
        return d16Sj - Math.floor(d16Sj);  
    }  
    /**  
     * Bináris hatványozás mod k, a 16^n mod k kiszámítása.  
     *  
     * @param n kitevő  
     * @param k modulus  
     */  
    public static long n16modk(int n, int k) {  
  
        int t = 1;  
        while(t <= n)  
            t *= 2;  
  
        long r = 1;  
  
        while(true) {  
  
            if(n >= t) {  
                r = (16*r) % k;  
                n = n - t;  
            }  
  
            t = t/2;  
  
            if(t < 1)  
                break;  
        }  
    }  
}
```

```
r = (r*r) % k;

}

return r;
}
/***
 * A [BBP ALGORITHMUS] David H. Bailey: The
 * BBP Algorithm for Pi. alapján a
 * {16^d Pi} = {4*{16^d S1} - 2*{16^d S4} - {16^d S5} - {16^d S6}}
 * kiszámítása, a {} a törtrészt jelöli. A Pi hexa kifejtésében a
 * d+1. hexa jegytől
 */
public static void main(String args[]) {

    double d16Pi = 0.0d;

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

    int jegy = 0;

    long delta = System.currentTimeMillis();

    for(int d=1000000; d<1000001; ++d) {

        d16Pi = 0.0d;

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

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

        d16Pi = d16Pi - Math.floor(d16Pi);

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

    }

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

A jelenlegi beállítás szerint a program a 0. pozíciótól a  $10^6$ . jegyig számítja ki a hexa jegyeket, ezt úgy

tudjuk egyszerűen átírni a  $10^7$ , vagy esetleg a  $10^8$ . számra, hogy a for ciklusban az értékeket egy-egy (vagy két-két) nullával megemeljük.

```
for (int d=1000000; d<1000001; ++d) {
```

Többet is tudnánk írni, de szerintem senki nem szeretne órákat várni egy pi hexa jegy kiszámolására.

A C# programot Windows 10 alatt futtattam, A Java, C és C++ kódöt viszont Linuxon, egy VirtualBox segítségével. A teszthez egy Intel i5-7500 processzort használtam, a táblázatba foglaltakon kívül még jónárszor lefuttattam a tesztet, csak hogy megbizonyosodjak arról, hogy tényleg fair módon mérjem a futási időket. Mindegyik esetben csak egy CPU magon futott a program. Mint az alábbi táblázaton látható, a C és C++ kódok futási ideje között szinte semmi különbség nincs, a futtatási adataim szerint pedig a Java nyert, mindenkor jegynél körülbelül 15%-kal kevesebb futási idővel. Az időt milliszekundum pontosan mértük.

Próba / jegy	$10^6$	$10^7$	$10^8$
C 1.	1,733629	20,299903	229,782562
C 2.	1,735853	20,735417	238,939637
C 5.	1,811959	20,705411	237,521828
C átlag	1,760480	20,580244	235,414676
C++ 1.	1,711960	20,505400	236,263000
C++ 2.	1,778180	20,384400	233,652000
C++ 3.	1,769210	20,958800	230,752000
C++ átlag	1,753117	20,616200	233,555667
C# 1.	2,007485	22,066694	251,104549
C# 2.	2,088996	21,455329	253,568114
C# 3.	2,149383	23,064083	250,344271
C# átlag	2,081955	22,195369	251,672311
Java 1.	1,544000	18,426000	207,922000
Java 2.	1,594000	17,777000	206,136000
Java 3.	1,564000	18,298000	207,641000
Java átlag	1,567333	18,167000	207,233000

## 12.3. Hello, Android!

Élesszük fel az SMNIST for Humans projektet! <https://gitlab.com/nbatfai/smnist/tree/master/forHumans/SMNIST>

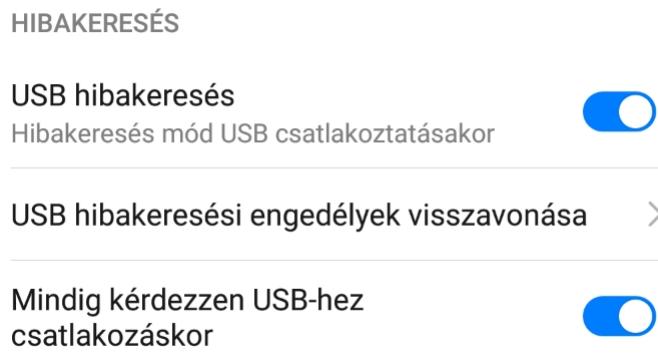
Apró módosításokat eszközölj benne, pl. színvilág.

Tutor: Győri Márk Patrik

Az SMNIST for Humans egy elég ambíciózus tervnek indult az esportolók képességeinek felmérése-re-fejlesztésére, jámagam is többször jártam sikерrel a "verseny" során, 0.3334988 végső értékkel. Néha a próbálkozások során eléggé zavart a színvilág, szóval ez a feladat pont kapóra jött, hogy én hogyan gondoltam ki az SMNIST színvilágát, megpróbáltam kicsit Mátrix-szerűre tervezni. Inkább csak a színeket írtam át, és először a gépen emulátorral, utána a saját telefonomon is kipróbáltam. A telefonos debugging-hoz először engedélyezni kellett a fejlesztő módot a beállításokban, a Build verzióra 7x kell koppintanunk hozzá.

Modell	FIG-LX1
Build-szám	9.1.0.123(C432E8R1P5)
EMUI verzió	9.1.0
Android verziója	9

Ezután pedig be kell állítanunk az USB debuggingot a fejlesztői eszközökön belül:



Miután ezeket beállítottuk, ki tudjuk próbálni a programot a telefonunkon keresztül is. a kódban a színek beállításáért a `setColor()` metódus felel, a metóduson belül a színeket pedig kétféleképp tudjuk megadni: **android.graphics.Color.BLUE**, ezen belül van pár szín ami alapból meg van határozva a funkciót belül; vagy pedig **android.graphics.Color.rgb()**, amelybe bármilyen RGB értéket beleírhatunk. Én inkább RGB színekkel dolgoztam, mert így sokkal több színkombinációt létre lehet hozni az alkalmazásunk számára. Így néz ki a két kódcsipet:

```
textPaint.setColor(android.graphics.Color.BLUE)
textPaint.setColor(android.graphics.Color.rgb(0, 0, 255))
```

A háttér váltakozó színeiért az alábbi kódcsipet felel, amit átállítottam sötét zöldre, hogy ne zavarjon be a színvilágba, és csak a pöttyökre és számokra tudjunk koncentrálni.

```
int[] bgColor =
{
    android.graphics.Color.rgb(18, 36, 17),
    android.graphics.Color.rgb(14, 23, 14)
};
```

A további színeket és azoknak kódját, valamint a program futását egy emulátoron keresztül az alábbi képen láthatjuk. Sajnos laptopon nem sikerült emulátorról indítani a programot, mivel az Android Studio csak a laptop processzorának integrált VGA-ját használja, nem pedig a dedikált kártyát.

The screenshot displays an Android application running on a virtual device. The app's title is "SMNIST". The main screen shows a 3x3 grid of numbers (0-8) arranged in a circle-like pattern around a central green dot. Below the grid, there is some text: "ms: (1000) 0 0 0 0 0 0 0 0 0 <0.0>" and "lvl/[...]: (3) 0/0 0/0 0/0 0/0 0/0 0/0 0/0 0/0 0/0". The code editor on the left shows the Java code for "SMNISTSurfaceView.java", which includes various paint settings and drawing logic for the grid.

```
305     }
306
307     @Override
308     protected void onSizeChanged(int newx, int newy, int x, int y) {
309
310         super.onSizeChanged(newx, newy, x, y);
311
312         width = newx;
313         height = newy;
314
315     }
316
317     private void cinit(android.content.Context context) {
318
319         textPaint.setColor(android.graphics.Color.rgb( red: 19, green: 173, blue: 31));
320         textPaint.setStyle(android.graphics.Paint.Style.FILL_AND_STROKE);
321         textPaint.setAntiAlias(true);
322         textPaint.setTextAlign(android.graphics.Paint.Align.CENTER);
323         textPaint.setTextSize(50);
324
325         msgPaint.setColor(android.graphics.Color.rgb( red: 78, green: 153, blue: 84));
326         msgPaint.setStyle(android.graphics.Paint.Style.FILL_AND_STROKE);
327         msgPaint.setAntiAlias(true);
328         msgPaint.setTextAlign(android.graphics.Paint.Align.LEFT);
329         msgPaint.setTextSize(40);
330
331         dotPaint.setColor(android.graphics.Color.rgb( red: 0, green: 255, blue: 21));
332         dotPaint.setStyle(android.graphics.Paint.Style.FILL_AND_STROKE);
333         dotPaint.setAntiAlias(true);
334         dotPaint.setTextAlign(android.graphics.Paint.Align.CENTER);
335         dotPaint.setTextSize(50);
336
337         borderPaint.setStrokeWidth(2);
338         borderPaint.setColor(android.graphics.Color.rgb( red: 0, green: 0, blue: 0));
339         fillPaint.setStyle(android.graphics.Paint.Style.FILL);
340         fillPaint.setColor(android.graphics.Color.rgb( red: 20, green: 20, blue: 20));
341
342         surfaceHolder = getHolder();
343
344         surfaceHolder.addCallback(new SurfaceEvents( SurfaceView: this));
345
346         scaleGestureDetector = new android.view.ScaleGestureDetector(context, new ScaleAda
347
348         for (int i = 0; i < digitsCoords.length / 2; ++i) {
349
SMNISTSurfaceView
```

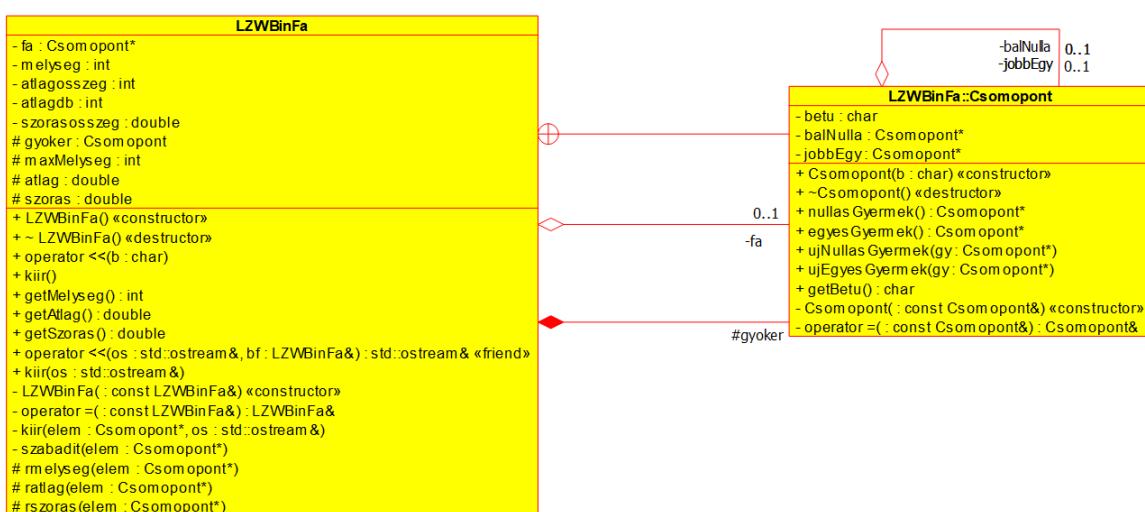
# 13. fejezet

## Helló, Mandelbrot!

### 13.1. Reverse engineering UML osztálydiagram

UML osztálydiagram rajzolása az első védési C++ programhoz. Az osztálydiagramot a forrásokból generáljuk (pl. Argo UML, Umbrello, Eclipse UML) Mutassunk rá a kompozíció és aggregáció kapcsolatára a forráskódban és a diagramon, lásd még: [https://youtu.be/Td\\_nlERIEOs](https://youtu.be/Td_nlERIEOs). <https://arato.inf.unideb.hu/batfai.norbert/UD/> (28-32 fólia)

Az osztálydiagram kirajzolásához az Umbrello nevű programot használtam, amibe csak be kellett importálnom az LZW binfa kódját, és már lehetett is dolgozni a kirajzoláson. Ezen modellező program UML (Unified Modelling Language) nyelvet vesz alapul, amely sok más program alapja is, mint például az ArgoUML, Eclipse UML, vagy például a NetBeans is támogatja. A binfa importálása után az editor készít nekünk egy osztálydiagramot, amiben két osztály van ábrázolva: maga az LZWBInFa, valamint a binfa Csomópont osztálya (a gyökér), amely kompozícióban és aggregációban is áll a fával. Az adott osztályban láthatjuk kiírva az osztály attribútumait, a piros vízszintes választóvonal felett a változóneveket, a vonal alatt pedig a funkciókat láthatjuk:



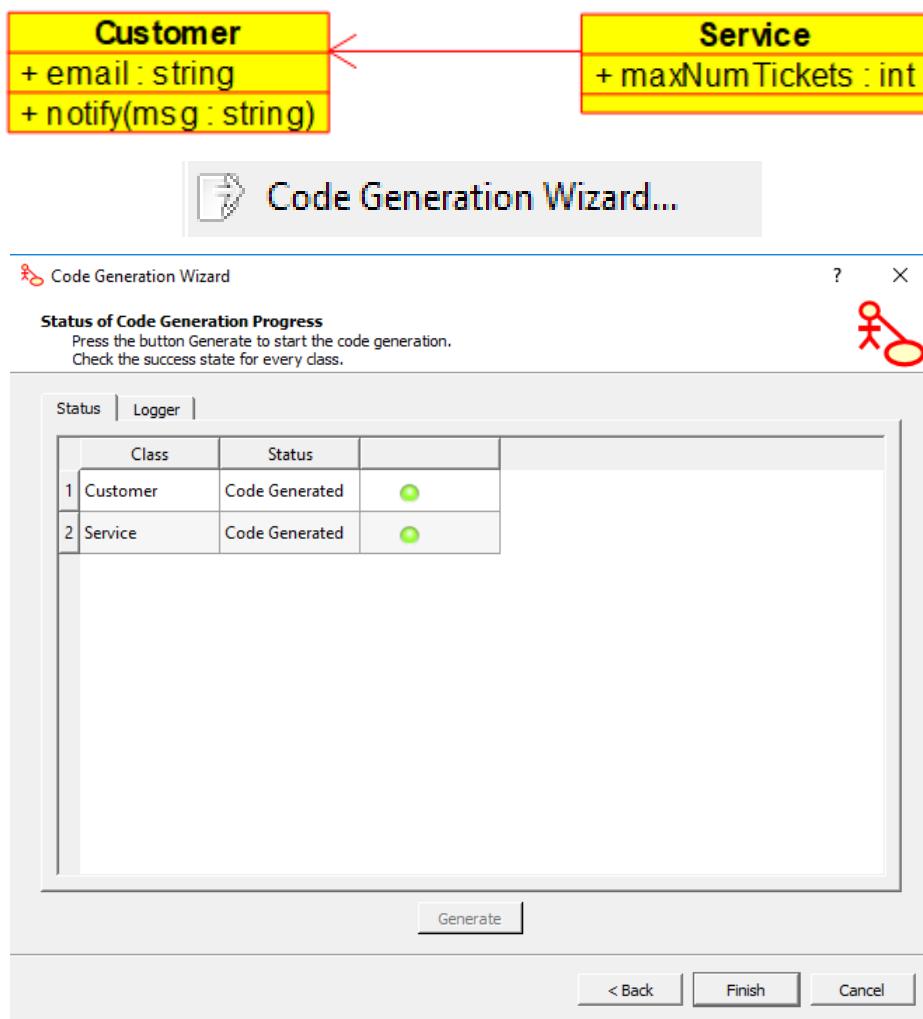
Fentebb említettem az aggregációt és a kompozíciót. Az aggregáció azt jelenti, hogy a Csomopont osztály a fa része, ennél erősebb pedig a kompozíció, miszerint jelen esetben a gyökér nem létezhet önmagában, csak a fa részeként. Kompozíciót kitöltött, aggregációt kitöltés nélküli rombusszal jelölünk.

## 13.2. Forward engineering UML osztálydiagram

UML-ben tervezünk osztályokat és generálunk belőle forrást!

Megoldás forrása:

Ebben a feladatban a "forward engineering"-gel, magyarul az előretervezéssel fogunk foglalkozni. Ez azt jelenti, hogy először a modellező szoftverben (jelen esetben Umbrello) lemodellezük például az osztályainkat és azok attribútumait, majd leképezés után implementálható kódot kapunk. Annyi gond van a forward engineering megoldással, hogy kevesebb infót kapunk, mivel az UML alapú modellek szemantikailag több információval látnak el minket bármelyik létező OO nyelvnél. Miután Umbrello-ban lemodellezük amit szerettünk volna, kóddá a "Code Generation Wizard..." segítségével tudjuk varázsolni. Én egy elektronikus levelezést próbáltam szemléltetni egy két osztályú modellel szerviz és vásárló között, amelyek asszociációban állnak. Kóddá varázslás után 4 fájl keletkezett: két .cpp fájl a két oszállynak, valamint két header fájl.



```

Customer.h
Customer.cpp
Service.h
Service.cpp

```

The screenshot shows four code editor tabs side-by-side:

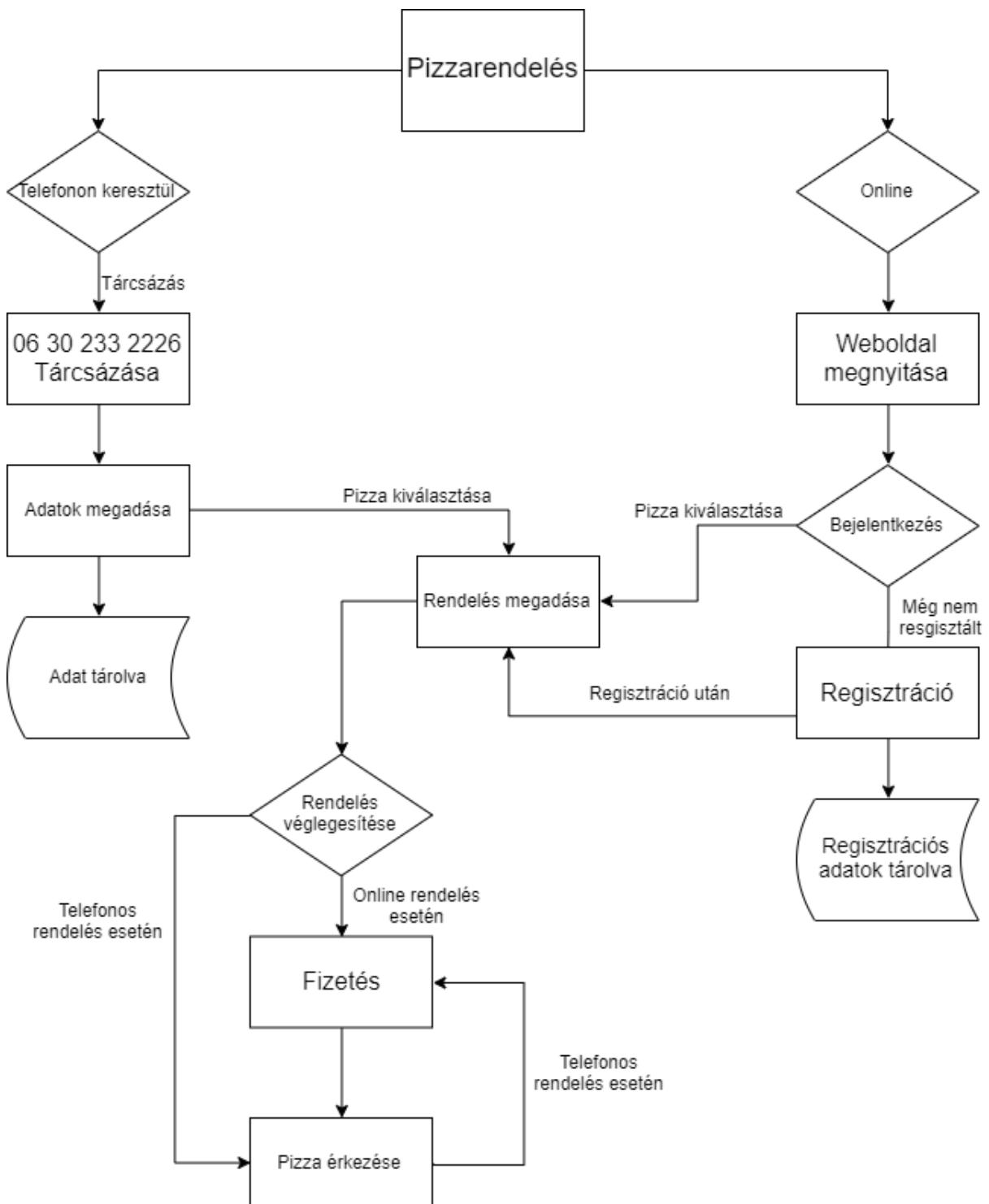
- Customer.h**: Header file for the Customer class. It includes "Customer.h", defines the Customer class with a constructor, destructor, and methods like initAttributes() and setEmail(). It also includes std::string.
- Customer.cpp**: Implementation file for the Customer class. It includes "Customer.h", defines the Customer class with a constructor, destructor, and methods like initAttributes() and setEmail(). It also includes std::string.
- Service.h**: Header file for the Service class. It includes "Service.h", defines the Service class with a constructor, destructor, and methods like initAttributes() and setMaxNumTickets(). It also includes std::string.
- Service.cpp**: Implementation file for the Service class. It includes "Service.h", defines the Service class with a constructor, destructor, and methods like initAttributes() and setMaxNumTickets(). It also includes std::string.

### 13.3. BPMN

Rajzoljunk le egy tevékenységet BPMN-ben! <https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog> (34-47 fólia)

Megoldás forrása: [Rendelés](#)

A BPMN (Business Process Model and Notation) egy BPML által tervezett, UML-hez hasonlító grafikus modellező nyelv, amely segítségével (nevéről értetődően) üzleti folyamatokat szokás modellezni egy egységes folyamatábrával. Két verzióval rendelkezik az OMG szabvány szerint: 1.1 és 2.0, ezekről bővebben itt olvashatunk: [1.1](#), [2.0](#). Én egy pizzarendelést modelleztem le két verzióban, egy draw.io nevű böngészős modellező oldalon. A rendelés két verziója a telefonos, illetve online rendelést mutatja be egy folyamatábrán keresztül. A folyamatábra draw.io modelljét [itt](#) találhatjuk, valamint itt a flowchart PNG-ben:



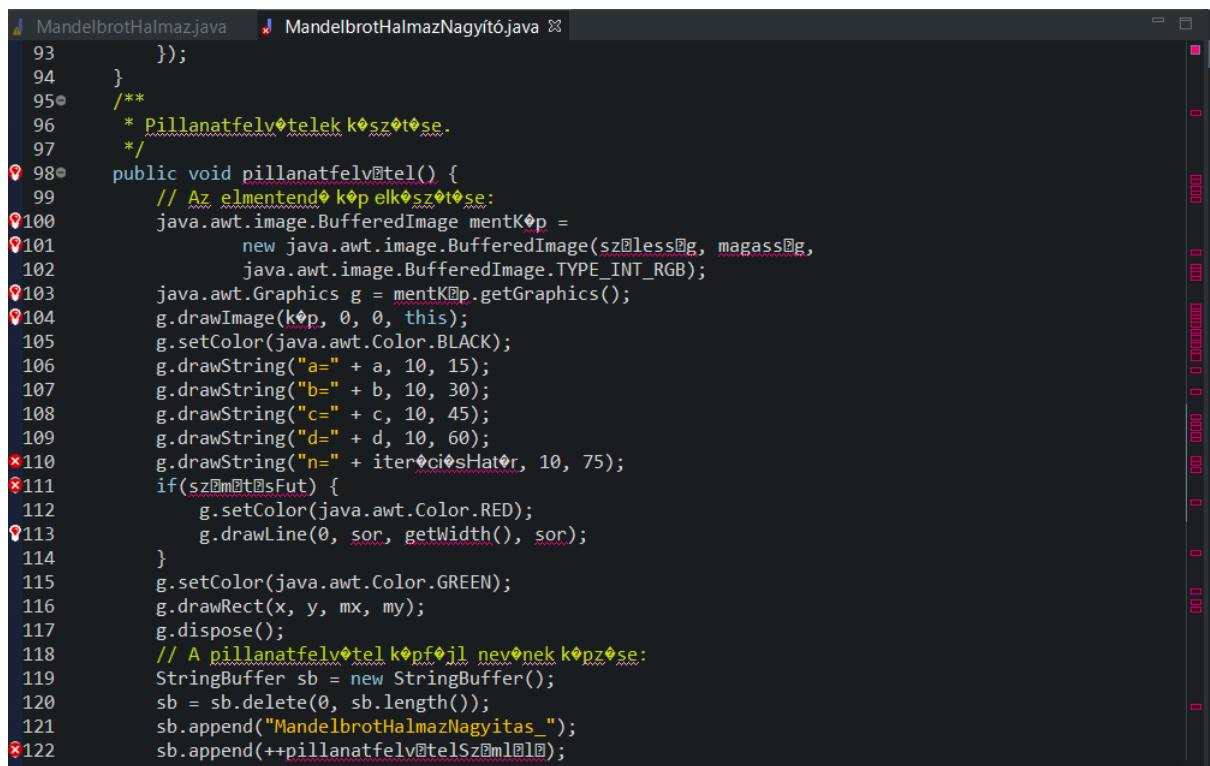
## 14. fejezet

# Helló, Chomsky!

### 14.1. Encoding

Fordítsuk le és futassuk a Javat tanítok könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékezes betűket! <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/adatok.html>

A feladatban megadott MandelbrotHalmazNagyító programunkat átírás nélkül kellett futtatnunk, ahol egy bökkenőbe ütközünk: figyelnünk kell a karakterkódolásra. Mivel a kódban ékezes betűk találhatóak, ezért megfelelő kódolást kell választanunk. Ha például az alapértelmezett UTF-8 kódolást alkalmaznánk, az alábbi (nem túl szép) látványt kapnánk:

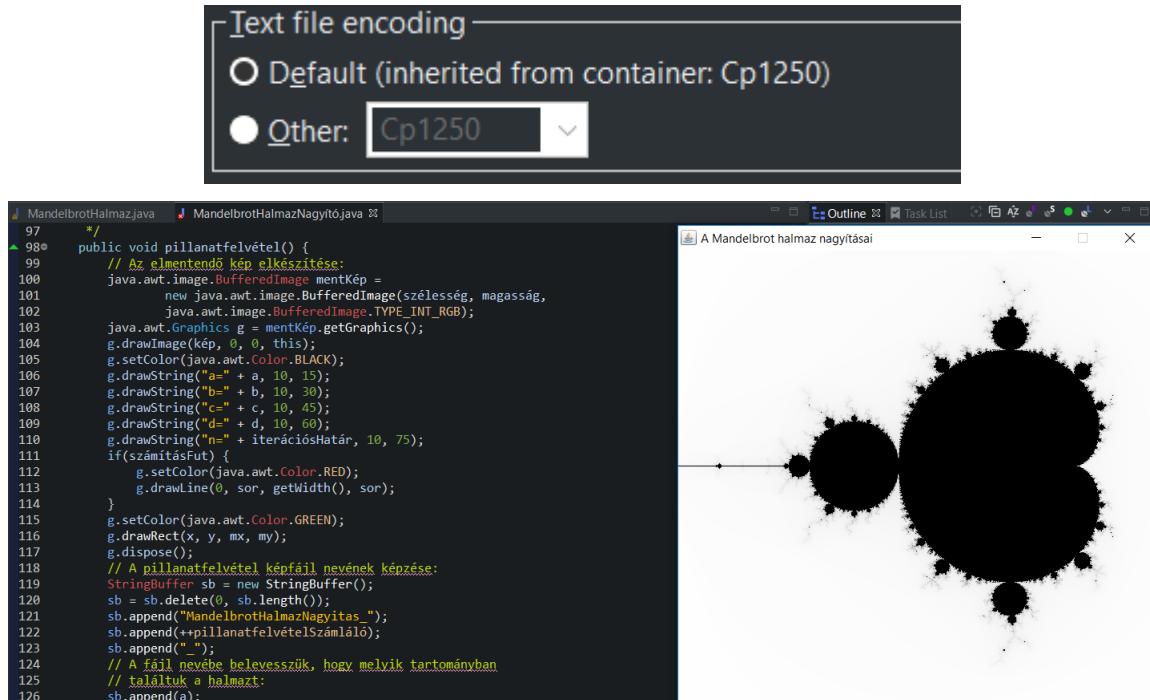


The screenshot shows a Visual Studio code editor with two tabs: "MandelbrotHalmaz.java" and "MandelbrotHalmazNagyító.java". The code in both tabs contains several encoding errors, specifically é, à, á, ô, ö, and ñ characters, which are displayed as question marks or other symbols. The code itself is a Java program that draws a fractal image and prints its parameters to the screen.

```
93     });
94 }
95 /**
96 * Píllanatfelv&telek k&sz&t&se.
97 */
98 public void pillanatfelv&tel() {
99     // Az e&lmentend& k&p el&k&sz&t&se:
100    java.awt.image.BufferedImage mentK&p =
101        new java.awt.image.BufferedImage(sz&less&g, magass&g,
102            java.awt.image.BufferedImage.TYPE_INT_RGB);
103    java.awt.Graphics g = mentK&p.getGraphics();
104    g.drawImage(k&p, 0, 0, this);
105    g.setColor(java.awt.Color.BLACK);
106    g.drawString("a=" + a, 10, 15);
107    g.drawString("b=" + b, 10, 30);
108    g.drawString("c=" + c, 10, 45);
109    g.drawString("d=" + d, 10, 60);
110    g.drawString("n=" + iter&ci&sHat&r, 10, 75);
111    if(sz&m&t&sFut) {
112        g.setColor(java.awt.Color.RED);
113        g.drawLine(0, sor, getWidth(), sor);
114    }
115    g.setColor(java.awt.Color.GREEN);
116    g.drawRect(x, y, mx, my);
117    g.dispose();
118    // A pillanatfelv&tel k&pfi&jl nev&nek k&pz&se:
119    StringBuffer sb = new StringBuffer();
120    sb = sb.delete(0, sb.length());
121    sb.append("MandelbrotHalmazNagyitas_");
122    sb.append(++pillanatfelv&telSz&ml&lk);
```

Ezt szerencsére Windows alatt a Visual Studio kiküszöbölte azzal, hogy automatikusan a CP1250 (vagy Windows 1250, ISO-8859-2, Latin 2) karakterkódolást választotta nekünk. Egyébként még megfelelt volna

nekünk a CP1252 kódolás is, amit ismerhetünk még ISO-8859-1, Windows 1252, Latin 1 néven is. Linux alatt egyszerűen egy kapcsolóra lenne szükségünk: "-encoding ISO-8859"



## 14.2. I334d1c45

Írj olyan OO Java vagy C++ osztályt, amely leet cipherként működik, azaz megvalósítja ezt a betű helyettesítést: <https://simple.wikipedia.org/wiki/Leet> (Ha ez első részben nem tettek meg, akkor írasd ki és magyarázd meg a használt struktúratömb memória foglalását!)

Tutor: Győri Márk Patrik

A leet cipher alapjának írnunk kell egy "ábécét", amiből majd a cipherünk válogatni tud. Ehhez használni tudjuk a fenti linket, ebben találunk pár példát a 1337 "ábécére".

```

import java.util.Arrays;
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Random;
import java.util.Scanner;

public class Leet {
    private String sb;
    private String word;
    private Random rand;

    @SuppressWarnings("serial")
    private HashMap<String, String[]> leet = new HashMap<String, String[]>()
    {
        put("a", new String[] { "4", "4", "@", "/-\\" });
        put("b", new String[] { "b", "8", "3", "}" });
    }
}

```

```

        put("c", new String[] { "c", "(", "@", "{" });
        put("d", new String[] { "d", ")", "]", "}" });
        put("e", new String[] { "3", "3", "3", "3" });
        put("f", new String[] { "f", "=", "ph", "#" });
        put("g", new String[] { "g", "6", "[", "[+" });
        put("h", new String[] { "h", "4", "-", "[ - ]" });
        put("i", new String[] { "1", "1", "|", "!" });
        put("j", new String[] { "j", "7", "_|", "/_" });
        put("k", new String[] { "k", "l<", "|{" });
        put("l", new String[] { "l", "1", "|_" });
        put("m", new String[] { "m", "44", "|V|" });
        put("n", new String[] { "n", "|\\|", "/V" });
        put("o", new String[] { "0", "0", "()", "[" });
        put("p", new String[] { "p", "o", "D", "o" });
        put("q", new String[] { "q", "9", "O_", "(,) });
        put("r", new String[] { "r", "12", "12", "2" });
        put("s", new String[] { "s", "5", "$", "$" });
        put("t", new String[] { "t", "7", "7", "''" });
        put("u", new String[] { "u", "_", "(_)", "[_]" });
        put("v", new String[] { "v" });
        put("w", new String[] { "w", "VV", "()" });
        put("x", new String[] { "x", "%", ")()", ")" });
        put("y", new String[] { "y" });
        put("z", new String[] { "z", "2", "7_", "_" });

        put("0", new String[] { "D", "0", "D", "0" });
        put("1", new String[] { "I", "I", "L", "L" });
        put("2", new String[] { "Z", "Z", "Z", "e" });
        put("3", new String[] { "E", "E", "E", "E" });
        put("4", new String[] { "h", "h", "A", "A" });
        put("5", new String[] { "S", "S", "S", "S" });
        put("6", new String[] { "b", "b", "G", "G" });
        put("7", new String[] { "T", "T", "j", "j" });
        put("8", new String[] { "X", "X", "X", "X" });
        put("9", new String[] { "g", "g", "j", "j" });
    } };
}

```

A **HashMap**-ban a kulcsok és az értékek párokban tárolódnak indexeléses megoldás helyett. A 1337 "ábécében" az angol ábécé betűi szerepelnek alternatívákkal, valamint az arab számok és azok megfelelői. Ezután példányosítunk, majd megírjuk a "titkosítót", amely kicséréli a karaktereket a fentebb eltárolt értékeinkre majd visszaadja azt, valamint ha esetleg nagybetű kerülne később az inputba, átírja kisbetűre.

```

public Leet(String word) {
    rand = new Random();
    this.word = word;
}

public Leet Encode() {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < word.length(); ++i) {

```

```
char c = word.charAt(i);
String loc = Character.toString(Character.toLowerCase(c));
if (leet.containsKey(loc))
    sb.append(leet.get(loc) [rand.nextInt(leet.get(loc).length)]);
else
    sb.append(c);
}
this.sb = sb.toString();

return this;

public String getLeet() {
    return this.sb;
}
}
```

Ezután már csak annyi hiányzik, hogy bekérjük az inputot, utána pedig kiíratjuk a már "leetelt" szöveget. Nézzük is:

```
public static void main(String[] args) {
    @SuppressWarnings("resource")
    Scanner console = new Scanner(System.in);
    String word;
    while (true) {
        word = console.nextLine();
        System.out.println(new Leet(word).Encode().getLeet());
    }
}
```

Teszteljük is a futtatott kódot:

```
alma
41|V|4
alma
4144/-\
szilva
s7_|1v/-\
```

### 14.3. Full screen

Készítsünk egy teljes képernyős Java programot! Tipp: [https://www.tankonyvtar.hu/en/tartalom/tkt/javatanitok-javat/ch03.html#labirintus\\_jatek](https://www.tankonyvtar.hu/en/tartalom/tkt/javatanitok-javat/ch03.html#labirintus_jatek)

A feladat megoldásához a Tanár Úr által mellékelt programot, vagyis a labirintus játékot élesztettem fel, amely alapból teljes képernyős móddal rendelkezik. A teljes képernyős mód kódját a LabirintusJáték.java fájlban tudjuk megnézni, a **teljesKépernyősMód** metóduson belül:

```
/**
 * Teljes képernyős módba (Full Screen Exclusive Mode) kapcsolás.
```

```
* Ha nem támogatott, akkor sima ablak fejléc és keret nélkül.  
*/  
public void teljesKépernyősMód(java.awt.GraphicsDevice graphicsDevice) ←  
{  
  
    int szélesség = 0;  
    int magasság = 0;  
    // Nincs ablak fejléc, keret.  
    setUndecorated(true);  
    // Mi magunk fogunk rajzolni.  
    setIgnoreRepaint(true);  
    // Nincs átméretezés  
    setResizable(false);
```

A fentiek után megnézzük továbbá, hogy át tudunk-e kapcsolni teljes képernyős módba:

```
boolean fullScreenTámogatott = graphicsDevice.isFullScreenSupported ←  
();
```

Ha át tudunk kapcsolni, akkor kapcsoljuk is át, és fullscreen "exkluzívba" váltunk.

```
if(fullScreenTámogatott) {  
    graphicsDevice.setFullScreenWindow(this);
```

Ezután bekérjük a képernyő jellemzőit: szélesség, magasság, színmélység, képfrissítés; majd kiírjuk azokat.

```
java.awt.DisplayMode displayMode  
        = graphicsDevice.getDisplayMode();  
szélesség = displayMode.getWidth();  
magasság = displayMode.getHeight();  
int színMélység = displayMode.getBitDepth();  
int frissítésiFrekvencia = displayMode.getRefreshRate();  
System.out.println(szélesség  
        + "x" + magasság  
        + ", " + színMélység  
        + ", " + frissítésiFrekvencia);
```

Elkérjük a képernyő lehetséges beállításait, majd megnézzük, hogy támogatja-e az 1024x768-as képernyő felbonntást, mivel a labirintus játéket ekkora felbontáshoz írtuk.

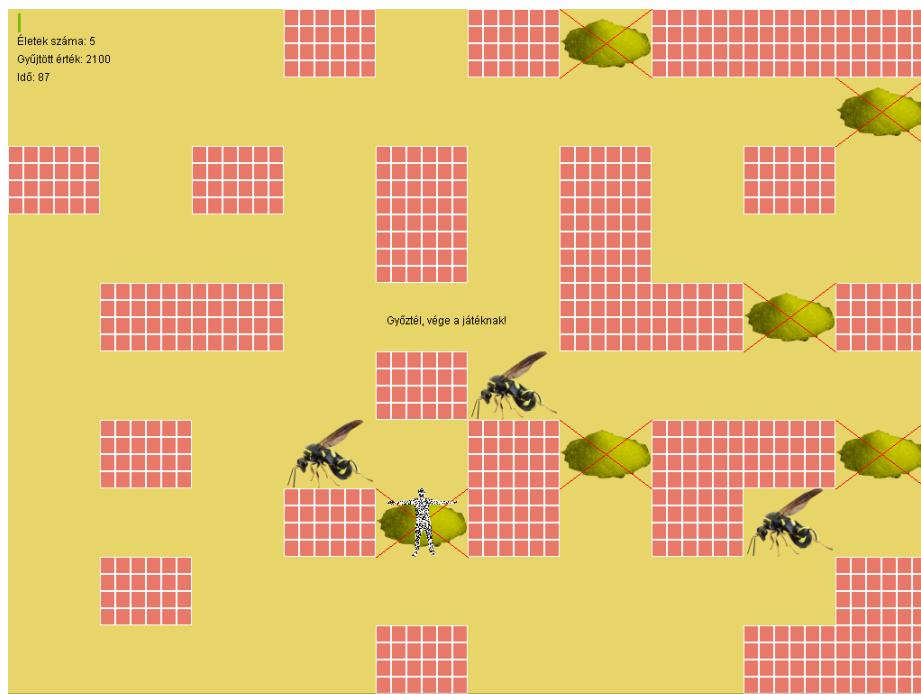
```
java.awt.DisplayMode[] displayModes  
        = graphicsDevice.getDisplayModes();  
  
boolean dm1024x768 = false;  
for(int i=0; i<displayModes.length; ++i) {  
    if(displayModes[i].getWidth() == 1024  
        && displayModes[i].getHeight() == 768  
        && displayModes[i].getBitDepth() == színMélység  
        && displayModes[i].getRefreshRate()  
        == frissítésiFrekvencia) {  
        graphicsDevice.setDisplayMode(displayModes[i]);  
        dm1024x768 = true;
```

```
        break;  
    }  
  
}
```

Ha nem elérhető az 1024x768 felbontás, tudassuk ezt a felhasználóval, egyébként folytathatjuk a kép felbontásának átállításával:

```
if (!dm1024x768)  
    System.out.println("Nem megy az 1024x768, de a példa képméretei ehhez a felbontáshoz vannak állítva.");  
  
} else {  
    setSize(szélesség, magasság);  
    validate();  
    setVisible(true);  
}  
  
createBufferStrategy(2);  
  
bufferStrategy = getBufferStrategy();  
  
}
```

És íme a kép a játék futásáról teljes képernyős módban:



# 15. fejezet

## Helló, Stroustrup!

### 15.1. JDK osztályok

Írunk olyan Boost C++ programot (indulj ki például a fénykardból) amely kilistázza a JDK összes osztályát (miután kicsomagoltuk az src.zip állományt, arra ráengedve)!

Tutor: Győri Márk Patrik

A feladatunk az volt, hogy írunk egy olyan programot, amely a JDK összes osztályát kilistázza. Először is letöltünk, aztán kibontunk egy tetszőleges jdk src.zip állományát. Én egy 1.8-as verziót használtam, ezzel lett tesztelve a program. A kapott zipet kicsomagoltam a "C:" meghajtón belüli "java src" mappába, hogy könnyebb legyen elérni, ne kelljen hosszú elérési utat megadni. A kódhoz használunk egy Boost nevű osztályt, amely számos funkcióval könnyíti meg az életünket. Ezekből implementáltunk párat a kódunkba, erről alább bővebben. Nézzük is a kódot:

```
#include <iostream>
#include <sstream>
#include "boost/filesystem.hpp"
#include <boost/algorithm/string.hpp>
#include <boost/algorithm/string/predicate.hpp>

using namespace std;
using namespace boost;
using namespace boost::filesystem;

string repeat(int n, string what) {
    ostringstream os;
    for (int i = 0; i < n; i++)
        os << what;
    return os.str();
}
```

a **repeat** függvényünk csak azért fog kelleni, hogy kicsit szebben nézzen ki a program futás közben, ezzel az osztály mélységét jelöljük. minden "+" jel +1 mélységet jelent.

```
int main()
{
```

```
string pa = "C:\\\\java src";
path apk_path(pa);
long long ennyimVan = 0;
recursive_directory_iterator end;
for (recursive_directory_iterator i(apk_path); i != end; ++i)
{
    const path cp = (*i);
    string name = cp.string();
    vector<string> strs;
    split(strs, name, is_any_of("\\\\"));
    string fName = strs[strs.size() - 1];

    if (ends_with(fName, "java")) {
        cout << " " << repeat(strs.size() - 7, "+") << " " << fName.substr(0, ←
            fName.size() - 5) << endl;
        ++ennyimVan;
    }
    else
    {
        cout << " | " << fName << " >" << endl;
    }
}

cout << "JDK -> Ennyi osztalyom van: " << ennyimVan << endl;
}
```

Eltároltuk a gyökér könyvtárat stringként, majd csinálunk belőle egy **path** típusú változót, mivel az a Boost funkció, amely rekurzívan járja be a mappákat (*recursive\_directory\_iterator*). A mappa mélységét a "per" jelekkel fogjuk vizsgálni, minden jel +1 mélység, ezt a szintén Boost-os **split** függvénytel tudjuk megoldani, amely szétbontja a szövegünköt. Szükségünk lesz egy escape karakterre, hogy ne legyen gond a "per" jelünk értelmezésével. Ezen túl már csak annyira lesz szükségünk, hogy kiírassuk a fájlokat és mélységeket. Mivel csak .java fájlokat keresünk, ezért vizsgáljuk egy szintén Boost könyvtárból hívott **ends\_with** függvénytel. A függőleges vonal az adott csomagot jelzi. A kód végén kiírjuk az osztályok számát, az "ennyimVan" segítségével.

```
HandlerBase
| helpers >
AttributeListImpl
AttributesImpl
DefaultHandler
LocatorImpl
NamespaceSupport
NewInstance
ParserAdapter
ParserFactory
SecuritySupport
XMLFilterImpl
XMLReaderAdapter
XMLReaderFactory
InputSource
Locator
Parser
SAXException
SAXNotRecognizedException
SAXNotSupportedException
SAXParseException
XMLFilter
XMLReader
JDK -> Ennyi osztalyom van: 7704
C:\Users\Para\source\repos\jdk\Debug\jdk.exe (process 1240) exited with code 0.
```

## 15.2. Hibásan implementált RSA törése

Készítünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló: <https://arato.inf.unideb.hu/batfai.mt.cs.szte.hu/~arato/inf/unideb/titkositasok/rsa/> (71-73 fólia) által készített titkos szövegen.

Tutor: Győri Márk Patrik

az RSA-algoritmus (Rivest, Shamir, Adleman) egy nyílt kulcsú titkosító algoritmus. Működése: Kiválasztunk két nagy prímet, majd összeszorozzuk azokat, és a kapott számunk lesz a nyilvános és a privát kulcsnak  $is(N)$ ; erre a számra kiszámoljuk rá az Euler-féle  $\varphi$  függvény értékét. Továbbá választunk egy  $e$  1-nél nagyobb egész számot, amely kisebb  $\varphi(N)$ -nél, valamint  $e$  és  $\varphi(N)$  relatív prímek, tehát legnagyobb közös osztójuk **1**. Az  $e$  lesz a nyilvános kulcs kitevője. Végül kiszámítjuk  $d$ -t, hogy teljesüljön **de**  $\equiv 1 \pmod{\varphi(N)}$ ; és  $d$  lesz a titkos kulcs kitevője. Az RSA titkosítást általában úgy alkalmazzuk, hogy az egész szövegünket titkosítjuk, de mivel a feladat hibásan implementált RSA-t kér, ezért nem az egész szöveget, hanem karakterenként titkosítunk.

```
public class rsa_chiper {
    public static void main(String[] args) {
        int bitlength = 2100;

        SecureRandom random = new SecureRandom();

        BigInteger p = BigInteger.probablePrime(bitlength/2, random);
        BigInteger q = BigInteger.probablePrime(bitlength/2, random);

        BigInteger publicKey = new BigInteger("65537");
```

```

BigInteger modulus = p.multiply(q);

String str = "this is a perfect string".toUpperCase();
System.out.println("Eredeti: " + str);

byte[] out = new byte[str.length()];
for (int i = 0; i < str.length(); i++) {
    char c = str.charAt(i);
    if (c == ' ')
        out[i] = (byte)c;
    else
        out[i] = new BigInteger(new byte[] {(byte)c}).modPow(publicKey, ←
            modulus).byteValue();
}
String encoded = new String(out);
System.out.println("Kódolt: " + encoded);

Decode de = new Decode(encoded);
System.out.println("Visszafejtett: " + de.getDecoded());
}
}

```

a fő osztályunk egy cipher, amely titkosítja a szövegünket karakterenként, majd megpróbálja azt dekódolni is a végén. A titkosítandó szövegünk a "*this is a perfect string*", ezt szeretnénk később visszafejteni. A szöveg visszafejtéséhez szükségünk lesz egy gyakorisági tábláta, amit egy txt-ben fogalmazunk meg, és a szövegünkben található karakterek gyakoriságát adjuk meg benne.

```

private void loadFreqList() {
    BufferedReader reader;
    try {
        reader = new BufferedReader(new FileReader("freq.txt"));
        String line;
        while((line = reader.readLine()) != null) {
            String[] args = line.split("\t");
            char c = args[0].charAt(0);
            int num = Integer.parseInt(args[1]);
            this.charRank.put(c, num);
        }
    } catch (Exception e) {
        System.out.println("Error when loading list -> " + e.getMessage());
    }
}

```

A fenti kódcsipettel beolvassuk a gyakorisági táblánkat, majd később eldönti, melyik karakterhez milyen betűt rendeljen hozzá. Az alábbi **for** ciklus karakterenként vizsgálja a szöveget, majd ha space-en kívüli karaktert talál, megnézi azt, hogy benne van-e a listában.

```

class Decode {
    private HashMap<Character, Integer> charRank;
    private String decoded;

```

```
public Decode(String str) {
    this.charRank = new HashMap<Character, Integer>();
    this.decoded = str;

    this.loadFreqList();

    HashMap<Character, Integer> frequency = new HashMap<Character, Integer <->
        >();
    for (int i = 0; i < str.length(); i++) {
        char c = str.charAt(i);
        if (c != ' ')
            if(frequency.containsKey(c))
                frequency.put(c, frequency.get(c) + 1);
            else
                frequency.put(c, 1);
    }

    while (frequency.size() > 0) {
        int mi = 0;
        char c = 0;
        for (Entry<Character, Integer> e : frequency.entrySet()) {
            if (mi < e.getValue()) {
                mi = e.getValue();
                c = e.getKey();
            }
        }
        this.decoded = this.decoded.replace(c, this.nextFreq());
        frequency.remove(c);
    }
}
```

Ha benne van a karakter a listában, növeli annak értékét 1-gyel, ha pedig nincs, 1-re állítja annak értékét. A futás végére elkészül egy lista, amely tartalmazza a karakterek gyakoriságát. Miután ez a lista elkészült, egy ciklus addig fog menni azon, amíg üres nem lesz; majd megnézzük a leggyakoribb elemet, kivesszük, majd átírjuk a leggyakoribb karakterre. Az említett karaktert a **nextFreq()** függvényünk fogja visszaadni.

Eredeti: THIS IS A PERFECT STRING
Kódolt:LmłĘ ŁĘ j ...\$iż\$ØL ŁLīŁ-=
Visszafejtett: TNIS IS F AERGEHT STRICP

Eredeti: THIS IS A PERFECT STRING
Kódolt:-"Í2 Í2 ĎB%#B#- 2-%ÍÉŠ
Visszafejtett: SGTI TI C NAREAES ISRTHF

### 15.3. Változó argumentumszámú ctor

Készítsünk olyan példát, amely egy képet tesz az alábbi projekt Perceptron osztályának bemenetére és a Perceptron ne egy értéket, hanem egy ugyanakkora méretű „képet” adjon vissza. (Lásd még a 4 hét/Perceptron osztály feladatot is.)

Tutor: Győri Márk Patrik

A feladat alapja a már jól ismert Perceptron osztály, viszont most egy érték helyett egy képet kell, hogy visszaadjon a kimenet. Nem akármilyet, hanem egy ugyanakkora méretű és felbontású képet.

```
double* operator() ( double image [] )  
{  
  
    units[0] = image;  
  
    for ( int i {1}; i < n_layers; ++i )  
    {  
  
        /*...*/  
  
        for(int i = 0; i < n_units[n_layers - 1]; i++)  
            image[i] = units[n_layers - 1][i];  
  
    }  
  
    return image;  
}
```

Ahhoz, hogy képet kapjunk vissza, a visszatérési érték típusát először át kell változtatni **double \*** típusúra, hogy ne egy értékkel térjen vissza a függvényünk. Ezután a **for** ciklust is átírjuk, hogy ne egy értéket adjon vissza, hanem végigjárva a rétegeket beállítja az *image* értékeit a kiszámolt értékekre, amelyeket az *units* táblában tároltunk.

```
#include <iostream>  
#include "png++/png.hpp"  
#include "mlp.hpp"  
  
using namespace std;  
using namespace png;  
  
int main(int argc, char* argv[]){  
    image<rgb_pixel> png_image(argv[1]);  
    image<rgb_pixel> new_png_image(png_image.get_width(), png_image.get_height());  
    int size = png_image.get_width() * png_image.get_height();  
  
    Perceptron* p = new Perceptron(3, size, 256, size);  
  
    double* image = new double[size];  
  
    for (int i{ 0 }; i < png_image.get_width(); i++)  
        for (int j{ 0 }; j < png_image.get_height(); j++) {  
            image[i * png_image.get_width() + j] = png_image[i][j].red;  
        }  
}
```

```
double* value = (*p)(image);
for (int i = 0; i < png_image.get_width(); i++)
    for (int j = 0; j < png_image.get_height(); j++) {
        new_png_image[i][j].red = png_image[i][j].red;
        new_png_image[i][j].green = png_image[i][j].green;
        new_png_image[i][j].blue = value[i * png_image.get_width() + j];
    }

new_png_image.write("uj_mandel.png");
delete p;
delete[] image;
}
```

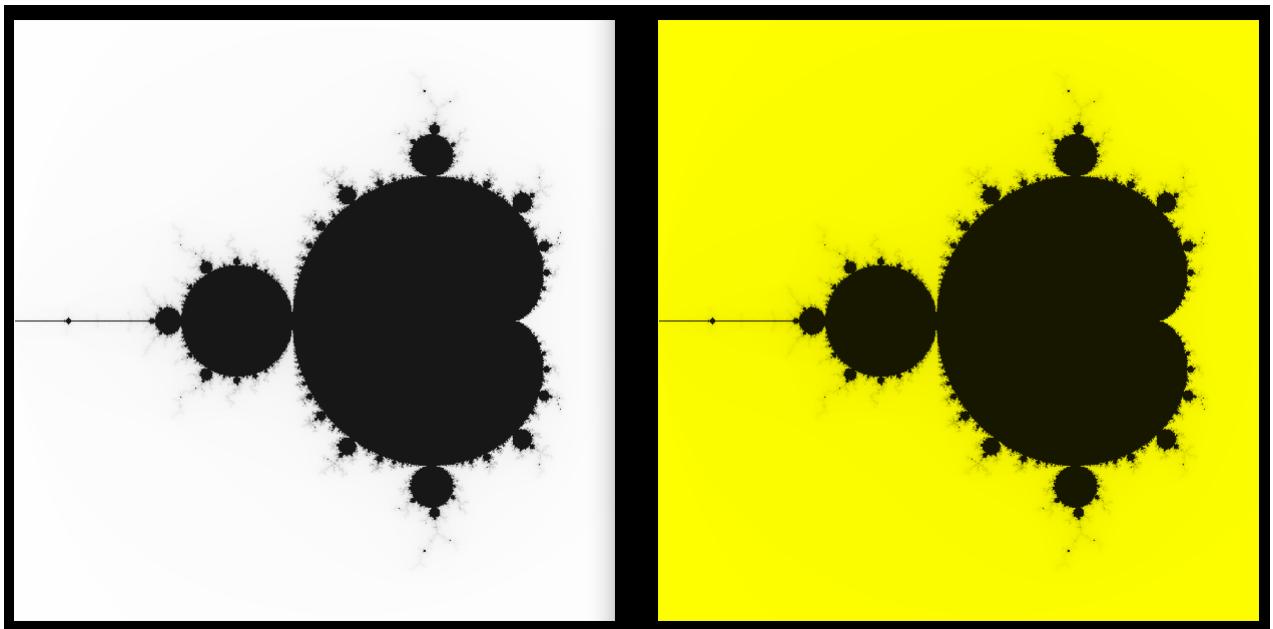
Arra kell ügyelnünk elsősorban, hogy a perceptron algoritmusunknak nem 1 értéket kell adnunk, hanem 256-ot.

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

Létrehozunk egy azonos felbontású képet, hogy szemmel látható legyen a változás. Mivel már nem egy visszatéríti érték van, a *double* típust átírjuk **double** \* típusúra. Egy dupla **for** ciklussal végigmegyünk a képen szélesség és hosszúság szerint, majd ugyanezzel a módszerrel beírjuk az új pixeleket. Az *uj\_mandel.png* piros és zöld tartományát nem bántjuk, a kék tartományt viszont a perceptron algoritmus által kiszámolt érték adja meg:

```
new_png_image[i][j].blue = value[i * png_image.get_width() + j];
```

Ezután a képünk kicsit sárgás háttérrel fog rendelkezni, de ezen kívül semmi nem fog változni:



# 16. fejezet

## Helló, Gödel!

### 16.1. STL map érték szerinti rendezése

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

Tutor: Győri Márk Patrik

Megoldás forrása:

A Standard Template Library (STL)-ben található **Mapok** olyan asszociatív tárolók, amelyek "mappolva" tárolnak elemeket, ezek az elemek kulcs-érték párral rendelkeznek. A feladatunk az volt, hogy a fénykardban található sort\_map függvényt úgy írjuk meg, hogy ha végigmegyünk rajta egy iterátorral, akkor érték szerint legyenek rendezve a sorok.

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>

using namespace std;

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

    for (auto& i : rank) {
        if (i.second) {
            pair<string, int> p{ i.first, i.second };
            ordered.push_back(p);
        }
    }
    sort(
        begin(ordered), end(ordered),
        [=] (auto&& p1, auto&& p2) {
            return p1.second > p2.second;
    })
}
```

```
    );
    return ordered;
}
int main() {
    map <string, int> rank;
    rank.insert(pair<string, int>("Alma", 23));
    rank.insert(pair<string, int>("Dinnye", 75));
    rank.insert(pair<string, int>("Szilva", 94));
    rank.insert(pair<string, int>("Barack", 15));
    rank.insert(pair<string, int>("Narancs", 65));
    rank.insert(pair<string, int>("Mandarin", 45));
    rank.insert(pair<string, int>("Citrom", 99));
    vector<pair<string, int>> res = sort_map(rank);
    for (auto& i : res)
        cout << i.first << " " << i.second << endl;
    return 0;
}
```

A függvény visszatérési értéke a vektor, ami egy szöveg-szám párból áll, ugyanúgy mint a main-ben található értékek. Maga a rendezés az **sort** függvénnyel történik. Bemenetnek kéri a vektor elejét, végét, valamint egy összehasonlító függvényt. Ugyanebben a feladatcsokorban találkozhatunk a Java-beli rokonára a **compareTo()** metódus személyében. A kettő között az a különbség, hogy a C++ függvény visszatérítési értéke egy logikai érték, Javában a **compareTo()** pedig egy számmal. A logikai értéket figyelembe véve a függvény rendezi az elemeket, így a végén kapunk egy rendezett listát. A rendezés a vektor értékei alapján történik. Végül egy **for** ciklussal végigmegyünk az elemeken, majd kiíratjuk azokat. Miután a ciklus végigment az elemeken, outputban megkapjuk azokat, immár érték szerint csökkenő sorrendben.

## 16.2. Alternatív Tabella rendezése

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 Interface Comparable T szerepét!

Megoldás videó:

Megoldás forrása:

Mielőtt a feladatba belekezdenénk, előbb frissítenünk kell az adatokat, mivel az évek műlásával változtak a csapatok, valamint az eredményeik is. A csapatok frissítése után a kereszttáblázatot is át kellett írnunk, ezen táblázat alapján. 0,1,2,3 értékekkel írjuk tele a táblázatunkat a színek alapján. A 0 üres(szürke) rublikát jelent, a zöld rublikákat 1-esekkel helyettesítjük, a sárgát 2-vel, a pirosat pedig 3-mal.

```
int [][] kereszt = {
    { 0, 1, 1, 3, 1, 1, 2, 1, 1, 1, 2, 3 },
    { 1, 0, 1, 1, 2, 1, 2, 2, 1, 1, 2, 3 },
    { 1, 1, 0, 3, 1, 2, 2, 1, 2, 2, 3, 3 },
    { 1, 2, 1, 0, 1, 1, 1, 2, 1, 1, 2 },
    { 3, 3, 2, 1, 0, 3, 3, 3, 1, 2, 3 },
    { 3, 1, 2, 3, 1, 0, 3, 1, 2, 1, 3, 2 },
    { 3, 2, 1, 3, 1, 2, 0, 3, 1, 1, 2, 1 },
```

```
{ 2, 3, 1, 3, 1, 3, 2, 0, 3, 1, 1, 3 },
{ 2, 1, 3, 3, 2, 1, 1, 1, 0, 1, 2, 3 },
{ 1, 3, 1, 1, 1, 2, 1, 3, 2, 0, 3, 1 },
{ 2, 1, 1, 2, 1, 1, 2, 3, 2, 1, 0, 1 },
{ 1, 2, 3, 1, 1, 1, 3, 2, 3, 1, 0 }
};
```

Az aktualizálás után végre törődhetünk magával a feladattal, hogy mi a szerepe a Comparable interface-nek a kódban: Először is nézzük, hogy mi is az a Comparable interface: Ezzel rendezhetjük a felhasználói osztályokat, és csak egyetlen metódusa van, a compareTo().

```
public int compareTo(Csapat csapat)
```

A metódus arra használatos, hogy összehasonlítsuk a jelenlegi objektumunkat a meghatározott objektummal. Háromféleképpen térhet vissza: Negatív integer, ha a jelenlegi objektum nagyobb mint a meghatározott; pozitív integer, ha a jelenlegi objektum kisebb, mint a meghatározott, valamint nulla, ha a két objektum egyenlő.

```
class Csapat implements Comparable<Csapat> {

    protected String nev;
    protected double ertek;

    public Csapat(String nev, double ertek) {
        this.nev = nev;
        this.ertek = ertek;
    }

    public int compareTo(Csapat csapat) {
        if (this.ertek < csapat.ertek) {
            return -1;
        } else if (this.ertek > csapat.ertek) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

Ezt a Comparable Interface-t megnézhetjük a JDK dokumentációjában az alábbi linken: [Comparable Interface](#)

### 16.3. GIMP Scheme hack

Ha az előző félévben nem dolgoztad fel a témat (például a mandalás vagy a króm szöveges dobozosat) akkor itt az alkalom!

Megoldás videó:

Megoldás forrása:

Ahhoz, hogy futtatni tudjuk GIMP-en belül a létrehozott scriptünket, be kell másolni a kódot a GIMP script mappájába. Majd a programon belül: **File/Create/BHAX/bhax\_chrome3** és már szórakozhatunk is kedvünkre a kívánt szövegünk átszínezésével. Ha futtatjuk a scriptet, egy alap beállítást is kapunk, amely megadja, hogy 1000x1000 képpontos kép legyen, valamint beállítja a fontot és a font méretét. A színt és a színátmenetet is beállíthatjuk. Nézzük a kód fő lépésein, ami alapján felépül a króm effekt:

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

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

(set! layer (car(gimp-image-merge-down image textfs   ←
  CLIP-TO-BOTTOM-LAYER)))
```

Az első lépésben megadunk egy fekete színnel kitöltött hátteret, majd jelezzük, hogy a szövegünk fehér színű legyen, a krómosítást pedig később fogjuk elvégezni. A második lépésben az ún. Gaussian módszerrel elmosssuk a szöveget:

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

Majd a harmadik lépésben lekanyarítjuk a szöveg éleit: (gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE) Végül ezt is elmosssuk: (plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE) Invertáljuk a képet, kitöröljük a fekete hátteret, így kapunk egy átlátszót, majd foglalkozunk a szöveggel. A lényeg a kilencedik lépés, ekkor egy Spline görbüvel meg tudjuk adni a gradient effektet: (gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))

# 17. fejezet

## Helló, !

### 17.1. OOCWC Boost ASIO hálózatkezelése

Mutassunk rá a scanf szerepére és használatára! <https://github.com/nbatfai/robocaremulator/blob/master/justine/rc.cpp>

Megoldás videó:

Megoldás forrása:

A **Boost ASIO** könyvtárat formázott beolvasásra használjuk, ami hálózati kommunikációra, valamint alacsony szintű I/O programozásra használatos, a neve is ezt takarja, "ASynchronous Input/Output". A neve is sugallja, hogy aszinkronban kell tevékenykednünk, ezt úgy tudjuk legjobban megfogalmazni röviden, hogy egy aszinkron művelet egy olyan függvény, amely azonnal visszatér érték nélkül, mivel az érték később adódik hozzá. Ez az aszinkron rendszer általában egy TCP kapcsolaton alapul, ugyanis arra érkeznek a csomagok a programok között. Sajnos a feladat címe valamint a feladat leírása elégé különbözik, a cím a Boost ASIO C++ könyvtárra tér ki, míg a feladat a carlexer.ll-ben található scanf függvények használatát kérdezi, ezért minden kettőről írok. Az **std::sscanf** segítségével be tudjuk olvasni az inputból az adatokat, majd azt eltárolhatjuk típus alapján.

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

A kapott inputból kiíratja az aktuális pozíciót, amiket a m\_cmd, from és a to referencia értékekből kapott meg. A to és from referenciakat a\_hpp fájlban találjuk meg (többek közt):

```
unsigned int get_from() const  
{  
    return from;  
}  
unsigned int get_to() const  
{  
    return to;  
}
```

A következő részlet pedig a kocsi útját hivatott olvasni; valamint itt láthatjuk az ss és sn-t is, az sn-t hozzáadjuk egy pufferértékhez, amely jelen esetben az ss; ezzel követni tudjuk majd a beolvasott bájtok mennyiségett.

```
{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;
}
```

## 17.2. SamuCam

Mutassunk rá a webcam (pl. Androidos mobilod) kezelésére ebben a projektben: <https://github.com/nbatfai/Samu>

Megoldás videó:

Megoldás forrása:

A SamuCam üzemeltetéséhez szükségünk lesz egy OpenCV-re, mivel képpel, kamerával foglalkozunk ebben a feladatban, az arcfelismeréshez az OpenCV CascadeClassifier osztályával fogunk dolgozni. A feladat az volt, hogy mutassunk rá a webkamera kezelésére a SamuCam projektben, ehhez szintén az OpenCV metodusait vesszük kölcsön. A metodus amit használni fogunk a videoCapture lesz, ez képes képek, videók, valamint a kamera képének befogadására is. Eloször is a **videoCapture.open** metodusra lesz szükségünk, ezzel megnyitjuk jelen esetben a kameránkat, majd beállítunk neki egy 176x144-es felbontást, valamint azt, hogy 10 képkocka/másodperces képfrissítési sebességgel rendelkezzen.

```
SamuCam::SamuCam ( std::string videoStream, int width = 176, int height = ←
    144 )
    : videoStream ( videoStream ), width ( width ), height ( height )
{
    openVideoStream();
}
SamuCam::~SamuCam ()
{
}
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 );
}
```

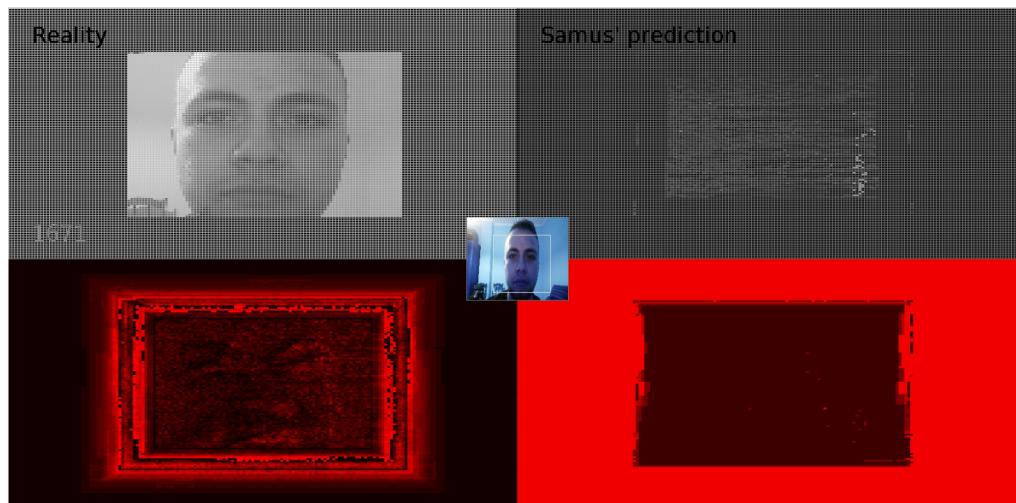
A **videoCapture.open()** metódusban megadhatunk 0 értéket, így az alapértelmezett kamerát fogja használni, nem kell nekünk külön megadni azt. A másik megoldás az, hogy egy IP kamerát használunk, futtatáskor csak annyit kell változtatnunk, hogy kapcsolónak írjuk a kamera ip-jét a **--ip** kapcsolóval. Ezután kell a programnak a faceXML fájlunk, ebben van benne hogy hogyan kell "felismerni egy arcot". A **videoCapture.read** függvényteljesítésekkel végezzük el a felismerést.

```
/*
void SamuCam::run()
{
    cv::CascadeClassifier faceClassifier;

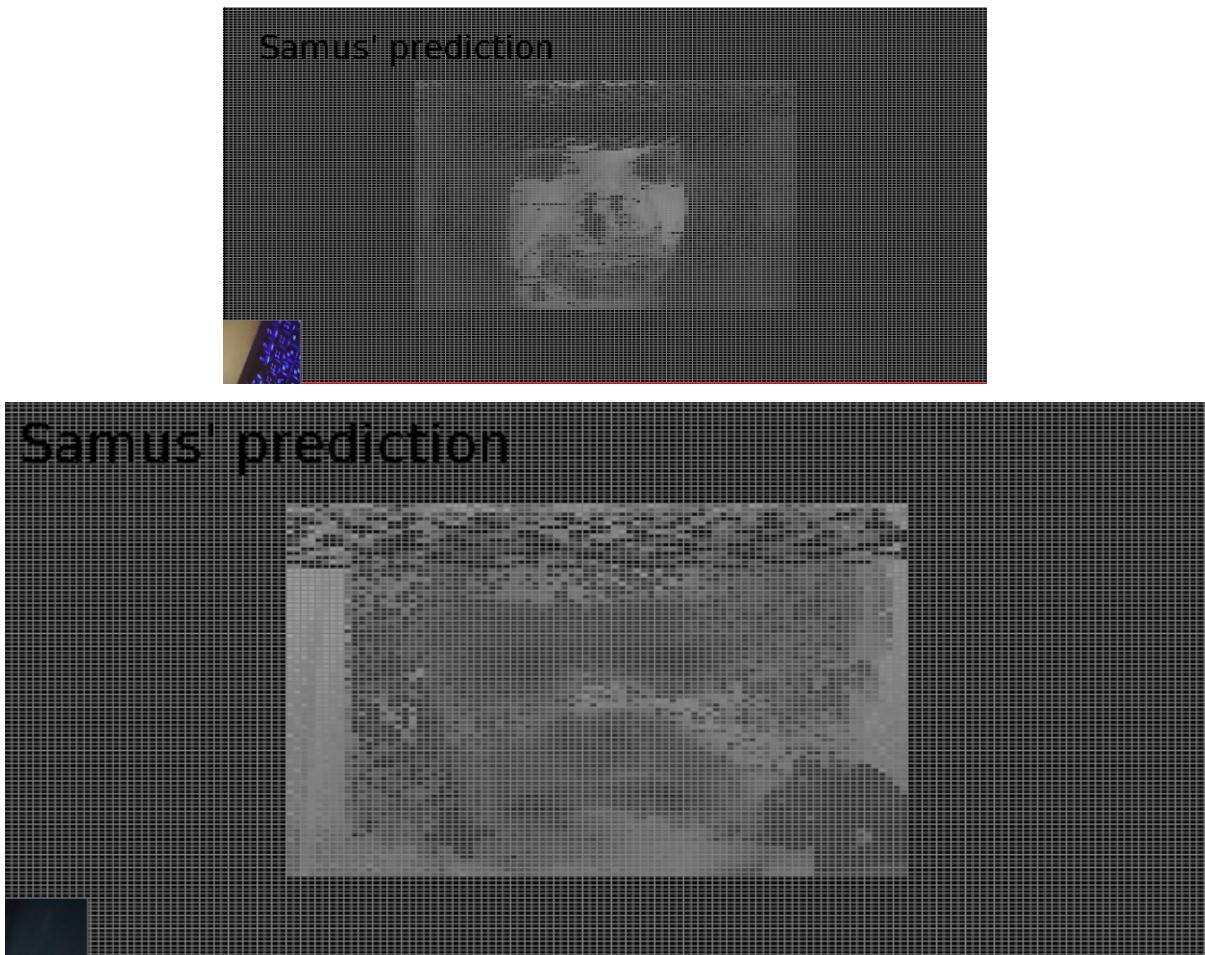
    std::string faceXML = "lbpcascade_frontalface.xml"; // https://github.com/Itseez/opencv/tree/master/data/lbpcascades

    if ( !faceClassifier.load ( faceXML ) )
    {
        qDebug() << "error: cannot found" << faceXML.c_str();
        return;
    }
    */
}
```

És itt van a kimenet:



Egyébként a program nagyon jó arra is, hogy megrémísszen minket a horrorosztikus predikcióival, íme két példa azoknak, akik nem szeretnének jobban aludni ezen fejezet olvasása után:



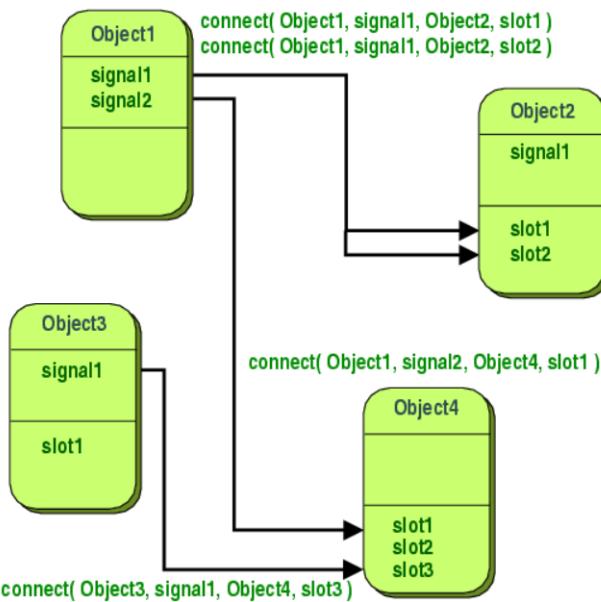
### 17.3. BrainB

Mutassuk be a Qt slot-signal mechanizmust ebben a projektben: <https://github.com/nbatfai/esporttalents-search>

Megoldás videó:

Megoldás forrása:

A BrainB nevezetű programmal sokan találkozhattunk már prog1, valamint e-sport órán is. Ez egy olyan program, amely a tehetségek kutatására lett kifejlesztve, és a "játék" lényege az, hogy a különféle entrópiák közül megtaláljuk és rajta is tartjuk a kurzort Samu Entropy-n. Egyre nehezebb feladatunk lesz, mivel egyre több új entrópia generálódik, és előbb-utóbb el fogjuk téveszteni. Futtatásához szükségünk lesz egy Qt-re, valamint megjelenítéshez egy OpenCV-re. Egy eléggé összetett progi, de a feladat jelenleg arra kér minket, hogy mutassuk be a játék slot-signal mechanizmusát. A slot-signal mechanizmus egy Qt-ben található feature, objektumok közötti kommunikációt tesz lehetővé. Ennek lényege az, hogy mikor megváltoztatunk valamit az egyik objektumban, arról a többi objektum is értesüljön. A könnyebb megértés érdekében mellékelem az alábbi képet, valamint a kép [forrását](#).



Ehhez kapcsolódik még az **emit** funkció, ami akkor teljesül, ha minden slot visszatér.

**BrainBThread.h:** itt találhatjuk a **signals** részt:

```
signals:
```

```
void heroesChanged ( const QImage &image, const int &x, const int &y );
void endAndStats ( const int &t );
```

**BrainBWin.h:** itt pedig a slot részt találjuk.

```
public slots :

void updateHeroes ( const QImage &image, const int &x, const int &y );
//void stats ( const int &t );
void endAndStats ( const int &t );
```

A **BrainBThread.cpp** fájlban is találunk egy emit függvényt:

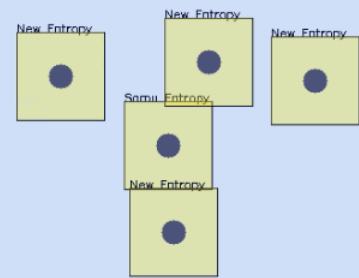
```
emit heroesChanged ( dest, heroes[0].x, heroes[0].y );
```

És most jön a legfontosabb rész, a **BrainBWin.cpp**-ben, Itt "áll össze a kép", itt kommunikál a BrainBThread és a BrainBWin. Az alábbi kódcsípetben makrókkal dolgozik a **connect()** függvény, a makrókat a nagybetűs **SLOT()** valamint **SIGNAL()** jelzik.

```
BrainBWin::BrainBWin ( int w, int h, QWidget *parent ) : QMainWindow ( ←
    parent )
{
    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 ) ) );
}
```

Press and hold the mouse button on the center of Samu Entropy  
1:1/10:0 18450 bps



## 18. fejezet

# Helló, Lauda!

### 18.1. Port scan

Mutassunk rá ebben a port szkennelő forrásban a kivételkezelés szerepére! <https://www.tankonyvtar.hu/hu/tartalom/tanitok-javat/ch01.html#id527287>

Megoldás videó:

Megoldás forrása:

Az alábbi program segítségével megnézhetjük, melyik portokat "figyeli" a gépünk.

```
public class KapusZkenner {

    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 + " figyeli");

                socket.close();

            } catch (Exception e) {

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

            }
    }
}
```

Maga a program elég egyszerű, kivételkezelés és socket programozás az alapja az egésznek. Egy **for** ciklusban zajlik le az egész, a **java.net.Socket()** osztály készít egy adott socketet, majd megpróbálja hozzárendelni annyi porthoz, amennyit megadtunk a ciklusban(ez jelen esetben 0-1023 portokat nézi meg). Ha nem

figyeli a gép az adott portot, kapunk egy kivételt, ha nem kapunk kivételt, akkor pedig kiírjuk, hogy az adott porton figyelik az adott IP címet, majd bezártuk a socketet a `socket.close()` metódussal. Futtatáskor argumentumnak megadjuk az IP címet, amit tesztelni szeretnénk, de figyeljünk arra, hogy csak saját IP-címen teszteljük, mert egy weboldal könnyen veszélyként kezeli, ha ráengedjük a programunkat.

## 18.2. AOP

Szőj bele egy átszövő vonatkozást az első védési programod Java átiratába! (Sztenderd védési feladat volt korábban.)

Megoldás videó:

Az Aspektus Orientált Programozás az OOP egy továbbfejlesztett változata, ahol például van egy programkódunk, és szeretnénk megnézni, hogy különböző aspektusokban hogyan viselkedik anélkül hogy belenyűlnünk a kódunkba, ez a módszer egy jó választás, bár már nem igazán használatos a gyakorlatban. Ahhoz, hogy dolgozni tudjunk az AspectJ-vel, szükségünk lesz egy AJDT-re, valamint egy Eclipse verzióra, amely támogatja azt. Ez jelen esetben egy 4.10-es verzió. Az AspectJ fájlunkat bele tudjuk szóni egy adott java kódba, és például tudunk egy adott metódus elő és utána is rakni kódcsipetet, a `before()` és `after()` meghívásával. Számunkra az LZWBInFa.java átiratába kell belefűzni egy AspectJ-s scriptet, amellyel az alap inorder kiíratás mellett preorderben illetve postorderben is kiírásra kerül a kimenet. Létrehozunk egy `pointcut`-ot, így a java fájlban a `kiir()` függvényünk lesz a pointcut, ezután ezek lesznek a joinpoint-jaink. A kiir függvényt a `call()` funkcióval hívjuk meg. Miután a futás elérte a kiir függvényig, életbe lép az AspectJ-s kódunk is, egy `after()` függvényel kiíratjuk a fát preorderben, majd postorderben is, végül bezártuk minden két fájlt. Kimenetként így három fájlt kapunk, a különböző rendezési módszerekkel.

```
package binfa;

import java.io.FileNotFoundException;
import java.io.IOException;

public aspect order {

    int melyseg = 0;

    public pointcut travel(LZWBInFa.Csomopont elem, java.io.PrintWriter os)
        : call(public void LZWBInFa.kiir(LZWBInFa.Csomopont, java.io.←
              PrintWriter)) && args(elem,os);

    after(LZWBInFa.Csomopont elem, java.io.PrintWriter os) throws IOException ←
        : travel(elem, os)
    {

        java.io.PrintWriter kiPre = new java.io.PrintWriter(
            new java.io.BufferedWriter(new java.io.FileWriter("preorder.txt")))
            ;
        java.io.PrintWriter kiPost = new java.io.PrintWriter(
            new java.io.BufferedWriter(new java.io.FileWriter("postorder.txt")))
            ;
    }
}
```

```
melyseg = 0;
preorder(elem, kiPre);
melyseg = 0;
postorder(elem, kiPost);

kiPre.close();
kiPost.close();
}

public void postorder(LZWBInFa.Csomopont elem, java.io.PrintWriter p) {
    if (elem != null) {
        ++melyseg;
        postorder(elem.egyesGyermek(), p);
        postorder(elem.nullasGyermek(), p);
        for (int i = 0; i < melyseg; ++i) {
            p.print("---");
        }
        p.print(elem.getBetu());
        p.print("(");
        p.print(melyseg - 1);
        p.println(")");
        --melyseg;
    }
}

public void preorder(LZWBInFa.Csomopont elem, java.io.PrintWriter p) {
    if (elem != null) {
        ++melyseg;
        for (int i = 0; i < melyseg; ++i) {
            p.print("---");
        }
        p.print(elem.getBetu());
        p.print("(");
        p.print(melyseg - 1);
        p.println(")");
        preorder(elem.egyesGyermek(), p);
        preorder(elem.nullasGyermek(), p);
        --melyseg;
    }
}
}
```

## 18.3. Junit teszt

A [https://propgater.blog.hu/2011/03/05/labormeres\\_othon\\_avagy\\_hogyan\\_dolgozok\\_fel\\_egy\\_pedat\\_poszt\\_kézzel\\_számított\\_mélységét és szórását dolgozd\\_be egy Junit tesztbe \(sztenderd védési feladat volt korábban\).](https://propgater.blog.hu/2011/03/05/labormeres_othon_avagy_hogyan_dolgozok_fel_egy_pedat_poszt_kézzel_számított_mélységét és szórását dolgozd_be egy Junit tesztbe (sztenderd védési feladat volt korábban).)

Megoldás video:

Megoldás forrása:

A Junit egy egyszerű framework Javához, amely segítségével teszteket tudunk elvégezni egy adott kódon, megvizsgálva azt, hogy egy olyan eredményt kapjunk vissza, amilyet szerettünk volna. Ilyen értékvizsgálatot tudunk végrehajtani például átlagok számolásakor, ez adhatta az ihletet a jelen feladathoz is. A feladatunk az, hogy egy ilyen Junit tesztet szőjünk bele az LZWBInFa programunkba, és vizsgáljuk meg az ott megkapott értékeket, amelyek az átlagot, szórást, valamint a mélységet reprezentálják. A tesztben előre kiszámolt értéket adunk meg egyik bemenetnek, másodiknak pedig azt az értéket, amit a fa programunk számol ki; majd ezt a két értéket összehasonlítjuk. A teszteléshez egy Maven projectet készítünk, és a *pom.xml* fájlba beírjuk dependency-ként a Junit-ot, az alábbi módon:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13-rc-1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Kezdetnek meghívjuk az LZWBInFát. A teszt kódunkban megadjuk a feladatban kért értéket (**0111100100100100**) amivel dolgozni fogunk a továbbiakban. A kódban a @Test értékkel adhatjuk meg azokat a részeket a kódban, amelynél tesztelni szeretnénk. létrehozunk három függvényt, egy az átlagot, egy a szórást, egy pedig a mélységet fogja kiszámolni. **for()** ciklusokkal minden hárrom teszben végigmegyünk a "value" minden jegyén, és összehasonlítjuk a kapott értéket a kiszámított értékünkkel az **assertEquals()** metódus segítségével. Ha megegyezik a két érték, helyes volt a tesztünk, viszont az assertEquals() metódus csak a hibás (nem megegyező) eredményt rögzíti.

```
LZWBInFA binfa = new LZWBInFa();
String value = "01111001001001000111";

@Test
public void atlagTest() {
    for(int i = 0; i < value.length(); i++)
        binfa.push_back(str.charAt(i));

    double atlag = binfa.getAtlag();
    assertEquals(2.75, atlag, 0.001);
}

@Test
public void szorasTest() {
    for(int i = 0; i < value.length(); i++)
        binfa.push_back(str.charAt(i));
```

```
        double szoras = binfa.getSzoras();
        assertEquals(0.9574271077563381, szoras, 0.001);
    }

@Test
public void melysegTest() {
    for(int i = 0; i < value.length(); i++)
        binfa.push_back(str.charAt(i));

    double melyseg = binfa.getMelyseg();
    assertEquals(1040, melyseg, 0.001);
}
```

# 19. fejezet

## Helló, Calvin!

### 19.1. MNIST

Az alap feladat megoldása, +saját kézzel rajzolt képet is ismerjen fel, [https://progpater.blog.hu/2016/11/13/hello\\_sbol](https://progpater.blog.hu/2016/11/13/hello_sbol) Háttérként ezt vetítsük le: <https://prezi.com/0u8ncvvoabcr/no-programming-programming/>

Megoldás videó:

Megoldás forrása:

Az MNIST (**M**odified **N**ational **I**nstitute of **S**tandards and **T**echnology) a NIST egy olyan alágazata, amely kézzel írott számokat hivatott felismerni, erre szolgál az MNIST adatbázis, amelyet 60000 kézzel írott számmal tanítottak, majd 10000-rel teszteltek, hogy minél pontosabban kiszűrje, milyen számról is van szó egyes esetekben. Ezek a számok 28x28 pixel felbontásúak, ezért a feladat során mi is egy ilyen felbontású képet fogunk rajzolni, amit a programnak fel kell ismernie. Másik kikötése az, hogy a képnek fekete-fehérnek kell lennie, mivel a program grayscale-ben dolgozik. Én egy 3-as számot rajzoltam GIMP-ben, ezt adtam a programnak felismerésre:



Mivel az MNIST gépi tanuláson alapul, értelemszerűen egy olyan platform kell, amely megfelel nekünk erre a célra. Jelen esetben ez a **TensorFlow**, amely példakódjai között megtaláljuk az MNIST példa forráskódját is. Így néz ki a kód egy kicsi átfárással:

```
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

import matplotlib.pyplot
```

```
FLAGS = None

def readimg():
    file = tf.io.read_file("harmas.png")
    img = tf.image.decode_png(file, channels=1)
    return img

def main(_):
    mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
```

Itt beimportáljuk többek között az MNIST dataset-jét, a TensorFlow gépi tanulós framework-jét, az argparse modult, valamint a matplotlibet, amellyel MATLAB-szerű plotokat tudunk megjeleníteni. Importálás után pedig inicializáljuk a FLAGS-et, amelyet az argparse-al társítva attribútumokkal látunk el, például számunkra a main-ben a data\_dir-rel. A readimg() definíálásával pedig később be tudjuk olvasni a kézzel rajzolt számunkat, majd dekódolni azt.

```
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 fenti kódrészletben elkészítjük a modellt, ami egyszerűen rámapolja az inputot az outputra, majd látjuk, hogy tényleg 28\*28-as képpel kell dolgoznunk, mivel meg van adva a 784-es méret, valamint mellette a 10-es érték azt jelenti, hogy ennyi féle kép közül válogathat a program; végül pedig összeszorozzuk az x és W mátrixokat majd hozzáadjuk a b-t. Most hogy kész a framework és a modell, elkezdjük tanítani. Először is megtanítjuk neki, hogy mi a jó és mi a rossz modell, ezt úgy tudjuk megtenni, hogy megnézzük a cross\_entropy-t, ami annyit csinál, hogy megmutatja mennyire is rossz az értékünk az igaz értékhez képest, majd egy Gradient Descent optimalizálót használunk, amely minimalizálja a rossz értékeket egy elfogadható értéig, ez jelen esetünkben 50%:

```
# Define loss and optimizer
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)
```

```
sess = tf.InteractiveSession()
# Train
tf.initialize_all_variables().run()
print("-- A halozat tanítása")
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("-----")
```

```
# 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. tesztképenek felismerése, mutatom a számot, a továbblételhez 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("-----")
```

Tanítás után tudjuk tesztelni a modellt, ezt mi a dataset 42. tesztképével próbáljuk meg. Amikor a modell ad egy predikciót, kapunk 10 számjegyet, amelyek 0-1 közötti értékkel rendelkeznek, a **correct\_prediction** pedig megnézi a legnagyobb értékűt, és azt rakja az outputra, majd ezt matplotlib-bel meg is jelenítjük, aztán pedig kiírjuk, melyik számjegyet is ismerte fel a képen.

Ezután odaadom neki a saját kézzel rajzolt hármasomat is, a program ezt fel is ismerte futáskor:

```
print("-- A saját kezi 3-asom felismerése, mutatom a számot, a továbblételhez csukd be az ablakat")

img = readimg()
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, 1), feed_dict={x: [image]})

print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
print("-----")
```

Végül pedig itt a kód, ami a futásért felelős:

```

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()

```

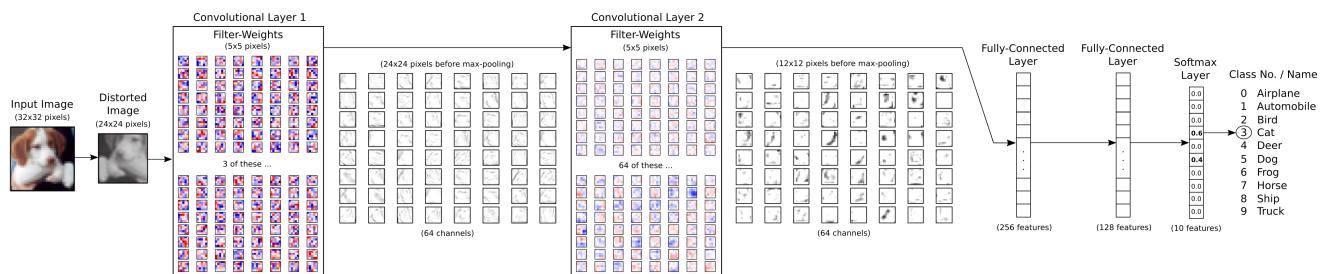
## 19.2. CIFAR-10

Az alap feladat megoldása, +saját fotót is ismerjen fel, [https://progpater.blog.hu/2016/12/10/hello\\_samu\\_a\\_cifar-10\\_tf\\_tutorial\\_peldabol](https://progpater.blog.hu/2016/12/10/hello_samu_a_cifar-10_tf_tutorial_peldabol)

Megoldás videó:

Megoldás forrása:

A CIFAR-10 egy másik TensorFlow-os példa, viszont itt már nem csak számokkal foglalkozunk, valamint nem csak grayscale-ben. Az előző feladatban 28x28px méretű képekkel dolgoztunk, ide viszont 32x32-es képek kellenek, hogy normálisan lefusson a gépi tanulós programunk. Az alap példában a CIFAR-10 dataset-et hívjuk segítségül, amely 60000 képet tartalmaz 10 osztályba rendezve, egyenként 6000 képpel: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. A program lényege az, hogy a 32x32-es képünkben csinál egy distorted 24x24-es képet, amelyet felbont, majd különböző szűrőkön átesve kimutatja, melyik osztályba tartozhat a 10 közül, valamint hány százalékban biztos benne a program. Ezt az alábbi folyamatábra remekül bemutatja:



Az egyik ilyen szűrő a konvolúciós réteg, ez egy olyan neurális hálózat, amely nem mátrixszorzást használ, hanem két függvény szorzatának integráltját veszi a számegyenesen, úgy hogy az egyiket eltolva és invertálva van. Ahhoz, hogy tanítani tudjuk a programunkat, egy bináris állományra lesz szükségünk, ezzel a kis programmal tudunk csinálni a képekből egy ilyen bináris állományt:

```

from PIL import Image
import numpy as np

im = Image.open('image.png')
im = (np.array(im))

r = im[:, :, 0].flatten()
g = im[:, :, 1].flatten()
b = im[:, :, 2].flatten()

```

```
label = [1]

out = np.array(list(label) + list(r) + list(g) + list(b),np.uint8)
out.tofile("out.bin")
```

Ezután jöhet a tanítás. Erre szükségünk lesz a **cifar10\_eval.py** progira, ez fogja valamelyik osztályba sorolni a képünket. A pontosság számunkra most nem szükséges, ezért ki lett kommentelve.

```
# while step < num_iter and not coord.should_stop():
# predictions = sess.run([top_k_op])
predictions = sess.run([top_k_op])
print(sess.run(logits[0]))
classification = sess.run(tf.argmax(logits[0], 0))
cifar10classes = ["airplane", "automobile", "bird", "cat", "deer", "dog" ←
", "frog", "horse", "ship", "truck"]
print(cifar10classes[classification])

true_count += np.sum(predictions)
step += 1

# Compute precision @ 1.
precision = true_count / total_sample_count
# print('%.3f' % (datetime.now(), precision))
```

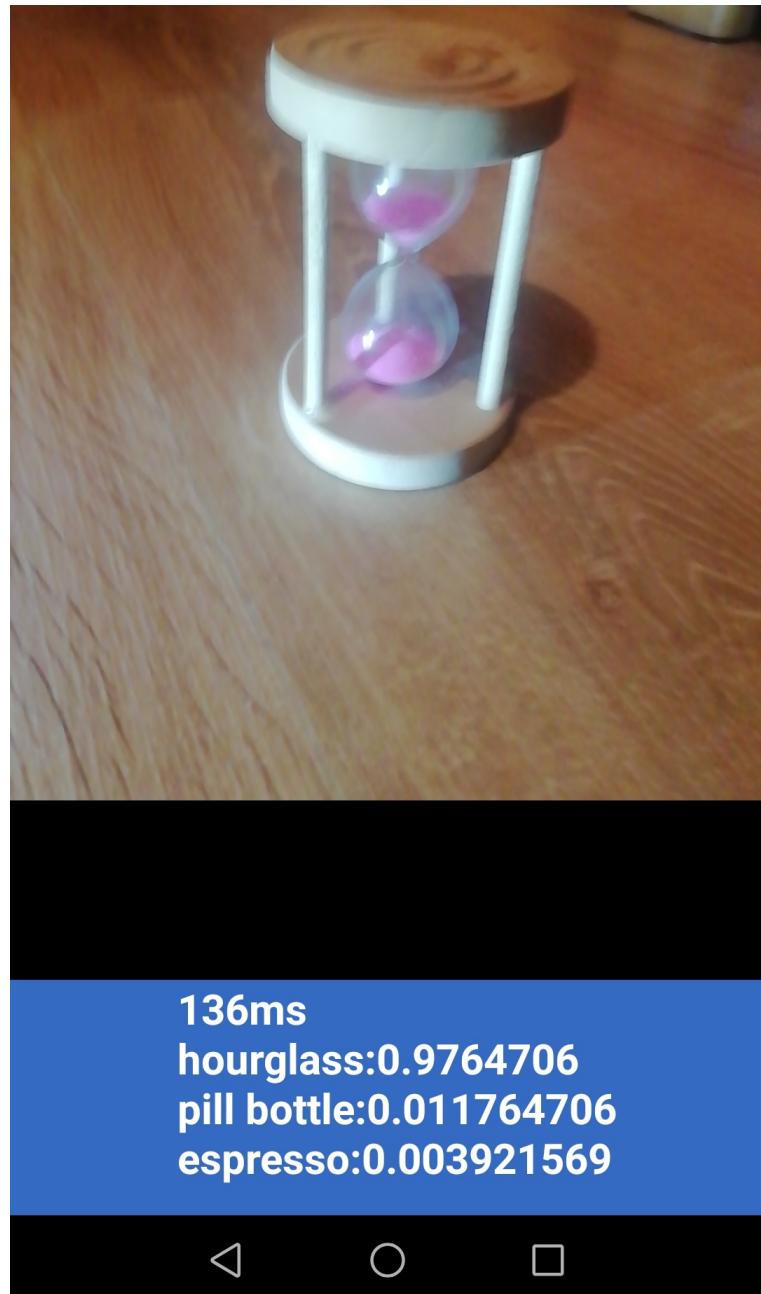
## 19.3. Android telefonra a TF objektum detektálója

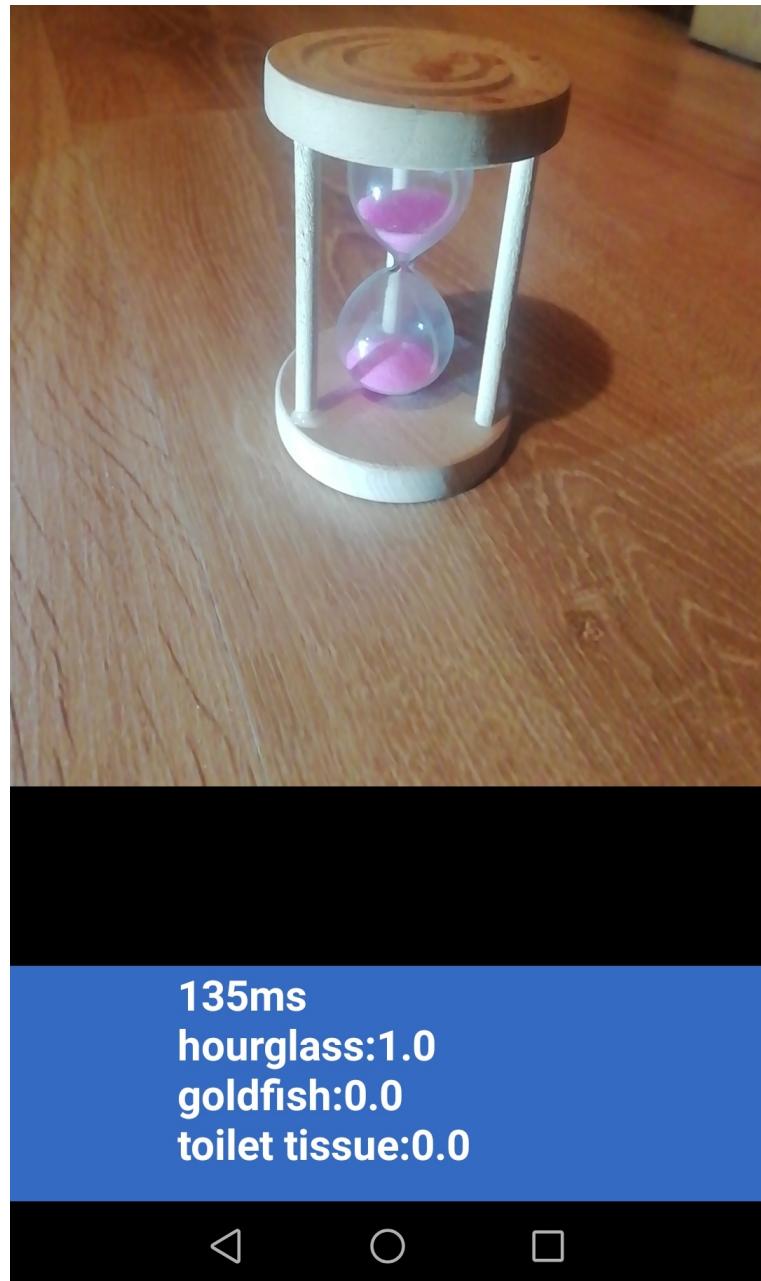
Telepítsük fel, próbáljuk ki!

Megoldás videó:

Megoldás forrása:

A **TensorFlow** Androidhoz írt objektum detektálója is a platform GitHub repójából származó példa, ezt egész egyszerűen le tudjuk tölteni androidos telefonunkra a [repó](#) dokumentációjában található linknek köszönhetően. A program kiírja a top 3 tippjét, hogy mire hasonlít a képünk a beletáplált adatok közül, és ezt százalékosan fejezi ki nekünk, hogy mekkora az egyezés mértéke. Elsőként egy cipőt próbált felismerni, ezt bár csak közel 37%-ban sikerült felismernie, de akkor is a döntő többség a futócipő tippre esett. Kicsit szétnéztem még, és kis keresgélés után fogtam egy homokórát, majd a kamera elé állítottam. Úgy tűnt, a homokórát nagyon szereti az appunk, mivel elsőre 95-98%-ban biztos volt, hogy amit lát, az egy homokóra, de sikerült screenshotolni egy olyan esetet is, amikor teljesen biztos (100%) volt magában. Véleményem szerint ez a kis program egy elég hasznos demó, a kezdők betekintést nyerhetnek, hogyan is néz ki egy objektum felismerésre használt gépi tanulásos program a gyakorlatban.





## 20. fejezet

# Helló, Berners-Lee!

### 20.1. C++ és Java összehasonlítás

C++: Benedek Zoltán, Levendovszky Tíhamér Szoftverfejlesztés C++ nyelven

Java: Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II

A Java és a C++ nyelvet összehasonlítva elég sok közös vonást találunk. Mindkettő általános célú objektumorientált programozási nyelv, melyek szintaxisban hasonlítanak egymásra. Nem véletlen, hiszen a Java A C és C++ nyelvektől örökölte a szintaxisát, viszont elég sokban különbözik: például sokkal egyszerűbb objektummodellel rendelkezik. Míg a Java nyelv platform független, ugyanezt nem tudjuk elmondani a C++ nyelvről, mivel utóbbit platform függő; részben azért, mert rendszer programozásra használatos, míg a Java inkább alkalmazások fejlesztésére alkalmas, ezen belül viszont nagyon széles körben használatos. A C++ nyelvet alapvetően ezért is készítették, hogy rendszereket és applikációkat tudjanak fejleszteni vele, a C nyelv bővítményeként tartották számon. A Java nyelvet viszont alapból egy interpreterként használták volna nyomtató rendszerek számára, de később kinőtte magát ebből a pozícióból. Arra terveztek, hogy könnyen használható legyen, valamint elérhető legyen széles körben.

Térjünk ki kicsit a fordítóra, valamint az interpreterre. A C++ nyelv csak compiler-t használ, ami lefordítja forráskódrról gépi kódra, utána futtatja; míg a Java nyelv compiler-t és interpretert is használ. A Java forráskódot fordításkor átkonvertáljuk bájtkóddá, majd az interpreterünk futtatja ezt a bájtkódot. Az interpreter miatt platform független a Java nyelv, és annak hiánya miatt platform függő a C++ nyelv. Az elején megemlítésre került, hogy minden nyelv objektumorientált. A két nyelv között ezen a téren csak annyi különbség van, hogy a Java (az alapvető típusokon kívül) minden objektumként kezel.

Pár szempont amivel még foglalkozhatunk: állítások, öröklés, thread támogatás. Pár állítással kapcsolatban eltér a két nyelv, például a **goto** állítás, amit a C++ támogat, a Java viszont nem. Az öröklés annyiban különbözik, hogy míg a C++ minden osztály öröklődési fát készít, a Java nyelv egyetlen fán belül intézi az öröklést, mivel minden osztály az objektum osztály gyermekje. Az objektum osztály az öröklődési fa gyökere. Többszörös öröklődésre csak a C++ képes natívan, Javában ezt csak interfészeken keresztül tudjuk megoldani, osztályon keresztül nem. Míg a Java-nak van beépített thread támogatása, a C++ nyelvnek nincs, utóbbi csak third-party könyvtárak segítségével képes erre.

## 20.2. Python

Python: Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba. Gyors prototípusfejlesztés Python és Java nyelven (35-51 oldal), a kijelölt oldalakból elmény-olvasónapló

A Python nyelvet Guido van Rossum holland programozó alkotta meg 1990-ben. Tulajdonképpen egy szkript nyelv, de magas szintű programozási nyelvként tekintünk rá. Ezen nyelv sok a fejlesztők számára sok pozitív tulajdonsággal rendelkezik. A Python magas szintű programozási nyelv, dinamikus, objektumorientált és ami a legjobb: platformfüggetlen. Leginkább tesztelésre szokás használni, de egyéb egyszerű alkalmazásokat is hatékonyan tudunk fejleszteni vele. A Python komponens más nyelveken megírt modulokkal is együtt tud működni. Rendkívül gyorsan tanulható, rövid idő alatt látványos eredményeket lehet vele elérni.

Alkalmazások fejlesztése során szükség van számunkra nem releváns és időigényes részek megírására is, mint például a fájlkezelő metódusok, a hálózatkezelés, a GUI kialakítása, stb. Mikor Pythonban fejlesztünk, nincs szükség fordításra, az interaktívan is használható interpreternek elég a Python-forrást megadni, amely automatikusan futtatja az alkalmazásunkat. Ezen nyelv segítségével könnyebben értelmezhető programokat tudunk készíteni, mint a Pythonnal ekvivalens nyelvekkel, mint például a C, C++ vagy Java.

A legfőbb jellemzője az, hogy behúzásalapú a szintaxisa. Nincs szükség kapcsos zárójelekre vagy explicit kulcsszavakra, mint a begin vagy az end. Másik sajátossága az, hogy nincs szükségünk a sor végi pontosvesszőre, mivel egy utasítás a sor végéig tart. Ha új sorban akarnánk folytatni az utasítást, akkor a sor elejére egy **backslash** jellel lehet ezt jelezni. Az interpreter minden egyes sort tokenekre bont, amelyek között tetszőleges mennyiségű whitespace lehet. Többféle token létezik: **azonosító**, **kulcsszó**, **delimiter**, **literál**.

Pythonban nem szükséges megadnunk a változó típusát, a rendszer automatikusan tudja azt, a változóhoz hozzárendelt érték alapján. Az adattípusok lehetnek: **számok**, **sztringek**, **ennesek(tuples)**, **listák**, **szótárak**. Pythonban változó alatt egy objektumra mutató referenciát értünk. Ezeknek nincsenek típusai, így bármely, akár különböző típusú objektumra is tudnak hivatkozni. Létezik Pythonban egy **garbage collector** funkció, amely automatikusan szabadítja fel az adott memóriaterületet, ha már egy adott objektumra az utolsó hivatkozást is töröltük. Kódunkban címkéket tudunk elhelyezni a **label** kulcsszóval, a címkéhez ugrani pedig a **goto** parancsal tudunk, valamint tudunk a label-ről a hozzá tartozó **comefrom** részhez ugrani.

Függvényeket a **def** kulcsszóval tudunk definiálni. Ezekre értékként is tekinthetünk, mivel ezek továbbadhatók más függvényeknek, konstruktorknak is. A Python támogatja az objektumorientált fejlesztési eljárásokat; tudunk definiálni osztályokat, valamint annak példányait, az objektumokat. Az osztályok kétféle attribútummal rendelkezhetnek: **objektumok**, valamint **függvények**. A függvényeket más néven **metódusoknak** vagy **tagfüggvények** hívjuk.

A Python is támogatja a kivételkezelést, itt csak egyszerűen a **try** kulcsszó után kell hogy szerepeljen az adott kódblokk, amit az **except** blokk követ (erre kerül a vezérlés hiba esetén.); valamint opcionálisan utána tudunk tenni egy **else** ágat is. Ezek helyett tehetünk egy **finally** blokkot is.

## **IV. rész**

### **Irodalomjegyzék**

## 20.3. Általános

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

## 20.4. C

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

## 20.5. C++

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

## 20.6. Lisp

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

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

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