

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Kosztya Zoltán

Copyright (C) 2019, Zoltán Kosztya, kzotya99@gmail.com,

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Kosztya, Zoltán	2019. október 23.	

REVISION HISTORY

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

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	7
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	10
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. Hivatkozási nyelv	12
3.4. Saját lexikális elemző	13
3.5. l33t.1	13
3.6. A források olvasása	14
3.7. Logikus	15
3.8. Deklaráció	15

4. Helló, Caesar!	17
4.1. int *** háromszögmátrix	17
4.2. C EXOR titkosító	18
4.3. Java EXOR titkosító	19
4.4. C EXOR törő	19
4.5. Neurális OR, AND és EXOR kapu	19
4.6. Hiba-visszaterjesztéses perceptron	20
5. Helló, Mandelbrot!	21
5.1. A Mandelbrot halmaz	21
5.2. A Mandelbrot halmaz a std::complex osztállyal	23
5.3. Biomorfok	24
5.4. A Mandelbrot halmaz CUDA megvalósítása	26
5.5. Mandelbrot nagyító és utazó C++ nyelven	26
5.6. Mandelbrot nagyító és utazó Java nyelven	28
6. Helló, Welch!	29
6.1. Első osztályom	29
6.2. LZW	30
6.3. Fabejárás	35
6.4. Tag a gyökér	36
6.5. Mutató a gyökér	43
6.6. Mozgató szemantika	44
7. Helló, Conway!	45
7.1. Hangyaszimulációk	45
7.2. Java életjáték	45
7.3. Qt C++ életjáték	45
7.4. BrainB Benchmark	49
8. Helló, Schwarzenegger!	51
8.1. Szoftmax Py MNIST	51
8.2. Szoftmax R MNIST	51
8.3. Mély MNIST	51
8.4. Deep dream	51
8.5. Robotpszichológia	52

9. Helló, Chaitin!	53
9.1. Iteratív és rekurzív faktoriális Lisp-ben	53
9.2. Weizenbaum Eliza programja	53
9.3. Gimp Scheme Script-fu: króm effekt	53
9.4. Gimp Scheme Script-fu: név mandala	53
9.5. Lambda	54
9.6. Omega	54
 III. Második felvonás	 55
10. Helló, Arroway!	57
10.1. OO szemlélet	57
10.2. Homokozó	58
10.3. „Gagyí”	59
10.4. Yoda	60
10.5. Kódolás from scratch	61
 11. Helló, Liskov!	 62
11.1. Liskov helyettesítés sértése	62
11.2. Szülő-gyerek	63
11.3. Anti OO	64
11.4. deprecated - Hello, Android!	64
11.5. Hello, Android!	65
11.6. Hello, SMNIST for Humans!	65
11.7. Ciklomatikus komplexitás	65
 12. Helló, Mandelbrot!	 67
12.1. Reverse engineering UML osztálydiagram	67
12.2. Forward engineering UML osztálydiagram	69
12.3. Egy esettan	72
12.4. BPMN	72
12.5. BPEL Helló, Világ! - egy visszhang folyamat	73
12.6. TeX UML	73

13. Helló, Chomsky!	74
13.1. Encoding	74
13.2. OOCWC lexer	74
13.3. l334d1c45	75
13.4. Full screen	75
13.5. Paszigráfia Rapszódia OpenGL full screen vizualizáció	75
13.6. Paszigráfia Rapszódia OpenGL full screen vizualizáció	75
13.7. Paszigráfia Rapszódia LuaLaTeX vizualizáció	76
13.8. Perceptron osztály	76
14. Helló, Stroustrup!	78
14.1. JDK osztályok	78
14.2. Másoló-mozgató szemantika	79
14.3. Hibásan implementált RSA törése	79
14.4. Változó argumentumszámú ctor	82
14.5. Összefoglaló	83
15. Helló, Gödel!	84
15.1. Gengszterek	84
15.2. C++11 Custom Allocator	84
15.3. STL map érték szerinti rendezése	84
15.4. Alternatív Tabella rendezése	85
15.5. Prolog családja	85
15.6. GIMP Scheme hack	85
16. Helló, !	86
16.1. FUTURE tevékenység editor	86
16.2. OOCWC Boost ASIO hálózatkézelése	86
16.3. SamuCam	86
16.4. BrainB	87
16.5. OSM térképre rajzolása	87
17. Helló, Schwarzenegger!	88
17.1. Port scan	88
17.2. AOP	88
17.3. Android Játék	88
17.4. Junit teszt	89
17.5. OSCI	89

18. Helló, Calvin!	90
18.1. MNIST	90
18.2. Deep MNIST	90
18.3. CIFAR-10	90
18.4. Android telefonra a TF objektum detektálója	91
18.5. SMNIST for Machines	91
18.6. Minecraft MALMO-s példa	91
19. Helló, Arroway!	92
19.1. C++ és Java összehasonlítás	92
19.2. Python	93
IV. Irodalomjegyzék	94
19.3. Általános	95
19.4. C	95
19.5. C++	95
19.6. Lisp	95

Ábrák jegyzéke

11.1. Példa a komplexitásra	66
12.1. 1.	67
12.2. 2.	68
12.3. 3.	68
12.4. 4.	69
12.5. 5.	70
12.6. 6.	70
12.7. 7.	71
12.8. 8.	71
12.9. 9.	72
12.10.10.	73
13.1. 1.	74
14.1. RSA	82

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás forrása: <https://github.com/kzoltan99/v-gtelen-ciklus> Program ami végtelen és 0 %-ban dolgoztatja a magokat:a while ciklus különböző iterációi

```
#include <stdio.h>
main()
{
    while(1) {}
}
```

```
#include <stdio.h>
#include <omp.h>
main()
{
    #pragma omp parallel
    while(1) {}
}
```

```
#include <stdio.h>
main()
{
    while(1) {}
    sleep(1);
}
```

Tanulságok, tapasztalatok, magyarázat... A végtelen ciklus a c nyelv egyik legalapvetőbb és legegyszerűbb kis programja. Nem jelentett számomra nehézséget megírni és alkalmazni. Működése egyszerű: a sima while ciklus a végtelenségig fut amíg igaz, ezzel alapvetően egy processzor szálát terhel 100 %-ig. Ha altatjuk a ciklust sleep-el akkor nem terheli a cpu-t. Ha pedig alkalmazzuk a párhuzamos futtatást akkor a cpu összes szálát 100 %-ig terheli le.

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

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

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

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

Mit for kiírni erre a T1000 (T1000) futtatásra?

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

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

Tanulságok, tapasztalatok, magyarázat... Véleményem szerint ez a program azért nem működőképes mert a lefuttatása után mindig ellentétes eredményt kapunk ha jól látom. Ha a T100-as gépnek olyan függvényt adunk akkor lefagy. Ha pedig olyat amiben nincs akkor végtelen ciklusként futtatja a gép. Szerintem ezért nem lehetséges egy ilyen program létrehozása bár számomra nem teljesen tiszta még ennek a gépnek a működése (vagy helyesebben a nem működése).

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/valtozo.c>

Tanulságok, tapasztalatok, magyarázat... A feladat megoldásához elég néhány egyszerű matematikai műveletet ismerni és megfelelően alkalmazni. A feladat többféleképpen is megoldható én most a szorzás és osztás alkalmazása mellett döntöttem. Először is adunk 2 random intet amelyeket előbb összeszorozunk. Aztán egyikkel és a masikkal is elosztjuk az eredményt és így az a és b int értéke megcserélődik.

2.4. Labdapattogás

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

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

Megoldás forrása: <https://github.com/kzoltan99/labdapattogtatas>

Tanulságok, tapasztalatok, magyarázat... Ez a feladat már kicsit nagyobb nehézséget okozott számomra de kezdem érteni a dolgot. Az if-es megoldás már megvan. A program lényege az h meg kell adnunk neki azt h az x és y tengelyen mekkorákat lépkedjen illetve a clear használatával akár azt is megoldhatjuk h egyszerűen csak egy kis o betű jelenjen meg a képernyőnkön. Majd ezek után if-ek használatával tudjuk úgy mond "visszafordítani" az o betű lépkedését hiszen így tudjuk meg hogy már a képernyő szélén járunk.

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

Az nélküli megoldás annyiban tér el, hogy az if-ek helyet maradékos osztást alkalmazunk. Egészen addig az osztandó szám értékét kapjuk vissza amíg nem érjük el a képernyő szélét. Ilyenkor az értékünk visszaáll -1-re (vagy 1-re) és a labda pattogási iránya megfordul.

```
getmaxyx(ablak, my, mx);
xj = (xj - 1) % mx;
xk = (xk + 1) % mx;

yj = (yj - 1) % my;
yk = (yk + 1) % my;

//clear ();

mvprintw (abs (yj + (my - yk)),
          abs (xj + (mx - xk)), "X");

refresh ();
usleep (150000);
```

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 forrása: <https://github.com/kzoltan99/feladatok/blob/master/szohossz.c>

Tanulságok, tapasztalatok, magyarázat... Ez a program a már előző félévben használt bitshifteléses módszert alkalmazza, tehát a biteket egyenként addig lépteti amíg a legelső nem 0 lesz. Ezt a módszert lehet akár szavak bithosszának mérésére is ahogyan ez ebben a feladatban látható is. A BogoMIPS maga egy a linux kernele által létrehozott processzor sebességmérő melyet maga a linux kitalálója Linus Torvalds alkotott meg.

2.6. Helló, Google!

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/pg.c>

Tanulságok, tapasztalatok, magyarázat... A PageRank algoritmust a Stanford egyetemen fejlesztette ki Larry Page és Sergey Brin 1998-ban. Ők a Google megalkotói és a mai napig ez az algoritmus a keresőmotorjuk szíve. Maga az algoritmus hyperlinkekkel foglalkozik. Ezeket megszámozott dokumentumokkal köti össze és rangsorolja a hálózatban betöltött szerep alapján. Lényegében arra a feltételezésre épít, hogy ha valaki egy oldalt belinkel a sajátjáról akkor azt fontosnak, jónak, hasznosnak tartja. Egy oldalt minél több helyről linkelnek be annál előrébb veszi a sorrendben a keresőmotor, persze az is számít, hogy az oldal ahonnan belinkelték a másikat mennyire fontos. Tehát a rekurzió elve is nagy szerepet játszik. Az én forrásomban 4 oldalról van szó, ezek kapcsolatát egy 4x4-es mátrixban tárolom el:

```
double L[4][4] = {
    {0.0, 0.0, 1.0/3.0, 0.0},
    {1.0, 1.0/2.0, 1.0/3.0, 1.0},
    {0.0, 1.0/2.0, 0.0, 0.0},
    {0.0, 0.0, 1.0/3.0, 0.0}
};
```

Ezt a mátrixot adjuk be a pagerank függvényünknek. Eredményként egy 4x1-es vektort fogunk kapni, ami az oldalak pagerankját tartalmazza. A PRv nevű tömbben van az oldalak eredeti értéke tárolva, a PR-ben pedig már a mátrixszorzás eredménye, ugyanis ezzel a művelettel tudunk pageranket számítani. Azonban nagyon oda kell figyelni a sorrendre mivel kihatással lehet az eredményre. (L és PRv tömb között kell elvégezni. Végül pedig a kiír függvény kiírja a számítás eredményeit.

Maga a számítás:

```
void
pagerank(double T[4][4]) {
    double PR[4] = { 0.0, 0.0, 0.0, 0.0 };
    double PRv[4] = { 1.0/4.0, 1.0/4.0, 1.0/4.0, 1.0/4.0};
```

```
int i, j;

for(;;){

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

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

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

kiir (PR, 4);
}
```

Illetve a kiírást végző függvény:

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

    int i;

    for (i=0; i<db; ++i){
        printf("%f\n",tomb[i]);
    }
}
```

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

Tanulságok, tapasztalat, magyarázat... Brun tétele azt mondja ki, hogy azok a számok melyeket ikerprímeknek, azaz olyan prímeknek melyeknek különbsége kettő, reciprokának összege egy véges értékhez konvergál melynek neve Brun-konstans. Szitaeljárások vizsgálata során bír nagy jelentőséggel ez a tétel, melyet Viggo Brun bizonyított be 1919-ben.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

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

Tanulságok, tapasztalatok, magyarázat... A Monty Hall probléma vagy paradoxon egy régi amerikai show legutolsó feladatán alapszik és annak műsorvezetőjéről nevezték el. A játék nagyon egyszerű: a játékosnak 3 ajtó közül kell választania és azt nyeri meg ami az ajtó mögött van. 2 ajtó mögött egy kecske egy mögött pedig egy autó van. A játékos először választ egyet de még nem tudjuk meg mi van mögötte. Először a műsorvezető kinyitja az egyiket (tudván azt a tényt, hogy mi lapul mögötte). Természetesen egy kecske lesz az. Ezekután a játékos döntés elé állítják: változtat a döntésén vagy sem. A probléma fő kérdése a változtatás fontossága és az h mennyire éri meg. A valószínűségszámítás szerint legtöbb esetben érdemes változtatni az eredeti döntésen. Viszont ez józan paraszti ésszel gondolkodva eléggé ellentmondásos innen a paradoxon elnevezés. A programunk is ezt a számítást végzi el. Megadjuk neki a kísérletek számát, a játékos és a műsorvezető helyzetét. For ciklusban történik a műsorvezető választása, az if és az else segítségével mindig azt az ajtót választja ami mögött nincs autó és a játékos sem választotta. A forrásunk vége egyértelmű: nemváltoztatásnyer akkor kap értéket ha a játékos elsőre jó ajtót választott és nem másította meg döntését. A változtatásnyer pedig akkor ha változtatott és így nyert. Legvégül pedig kiírjuk a kísérletek számát és hogy a két lehetőség hányszor fordult elő így összehasonlítva azt, hogy érdemes-e váltani vagy sem.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfiával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_1.pdf?fbclid=IwAR3W

Tanulságok, tapasztalatok, magyarázat... Az unáris (egyes) számrendszer a legegyszerűbb számrendszer és természetes számoknál használjuk. Lényege, hogy van egy N szimbólum melynek értéke egy, ezeket az N-eket egymás mellé rakosgatva tudunk számolni. (Ilyen például az is ha a kezünkön számolunk.) A decimálisba való átváltás elég egyszerű: fogjuk az N-eket és kivonogatjuk a decimális számból majd egyenként felírjuk őket.

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:

Tanulságok, tapasztalatok, magyarázat...

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/c89.c>

Tanulságok, tapasztalatok, magyarázat... A C szabványok az évek során egyre nagyobb fejlődésen mentek keresztül. Ezen fejlesztések által a szabványok egyre több funkciót tartalmaznak és több mindenre képesek. Az általam bemutatott kódcsipetben látható a fejlődés hiszen az a C89-es szabványban nem, de a C99-esben lefordul.

```
#include <stdio.h>
int main()
{
    for(int a=5; a>10; a++);
    return 0;
}
```

Ez azért van mert c89-ben a statement scope változó deklarálása nem lehetséges. De az újabb szabvány segítségével már ezt is megtehetjük és ezzel egyszerűbbé téve a munkánkat. Mindenképpen ennél sokkal számottevőbb különbség van a két szabvány között, de számomra ez volt a legszembetűnőbb, ezért választottam. A

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/realnumber.l>

Tanulságok, tapasztalatok, magyarázat... Először is a definíciók részben deklaráljuk a valós számokat (realnumbers) megadjuk a fajtáját és a kezdőértékét. Az első rész végén pedig megadjuk a digit (számok 0-9) definíciót. Aztán a program felismeri a számot, ami lehet akár 0 is. Ha talál számot elkezd növelni a 0 kiindulási értéktől. Aztán kiírja a a realnum-ot előbb stringel majd számként az atof függvény segítségével. A végén pedig elindítjuk a lexikális elemzést. Ez a feladat a lexelés egy egyszerűbb változata pár sorból áll és számomra segített magának a lex-nek a megértésében.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/l337d1c7.c>

Tanulságok, tapasztalatok, magyarázat... Ebben a feladatban a c forráskódot nem nekünk kell megírni hanem a lexer segítségével készítjük el. Először létrehozunk egy cipher nevű struktúrát amely egy betűt és az ahhoz társított négy titkosítási változatot fogja tartalmazni. Aztán létrehozuk a l337d1c nevű tömböt aminek nem monjuk meg hány eleme lehet, de ezt nem is kell hiszen a fordító majd megszámlálja. Ezt egy %% -lel zárjuk. Ez a rész a lex definíciók része. A "." -al kezdődik a szabályok rész. Ebben meghatározzuk, hogy mely karakter olvasása után mit csináljon a program. Például ha entert ütünk ebben a programban

akkor semmi sem történik. Az utolsó rész pedig maga a c kód. Itt indítja el a lexer futtatását az előbbieken alapján. Ennek a feladatnak a megoldása és megértése kisebb nehézséget okozott számomra mivel még nem használtam lexet, de Bátfai Tanár Úr videója alapján sikerült viszonylag hamar megértenem a dolgot.

3.6. A források olvasása

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

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

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



Bugok

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

i.

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

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: <https://github.com/kzoltan99/sigint>

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat... A feladat megoldása folyamatban van...

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

Tanulságok, tapasztalatok, magyarázat... 1. Minden x-re van olyan y, hogy x kisebb mint y és y prím. 2. Minden x-re van olyan y, hogy x kisebb mint y, y prím és y rákövetkezőjének a rákövetkezője is prím. 3. Van olyan y, hogy minden x-re igaz, hogy ha x prím akkor x kisebb y-nál. 4. Van olyan y, hogy minden x-re igaz, hogy ha y kisebb mint x akkor x nem prím.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

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

Mit vezetnek be a programba a következő nevek?

- `int a;` egész típusú változó
- `int *b = &a;` egész típusú változó
- `int &r = a;` egész referenciája
- `int c[5];` egészek 5 elemű tömbje
- `int (&tr)[5] = c;` egészek 5 elemű tömbjének referenciája, ami c-re mutat
- `int *d[5];` 5 elemű egészre mutató mutatók tömbje
- `int *h ();` egészre mutató mutatót visszaadó függvény
- `int *(*l) ();` egészre mutató mutatót visszaadó függvényre mutató mutató
- `int (*v (int c)) (int a, int b);` egészet visszaadó és két egészet kapó ←
függvényre mutató mutatót visszaadó, egészet kapó függvény
- `int ((*z) (int)) (int, int);` függvényt mutató egy egészet visszaadó és két ←
egészet kapó függvényre mutató mutatót visszaadó, egészet
kapó függvényre

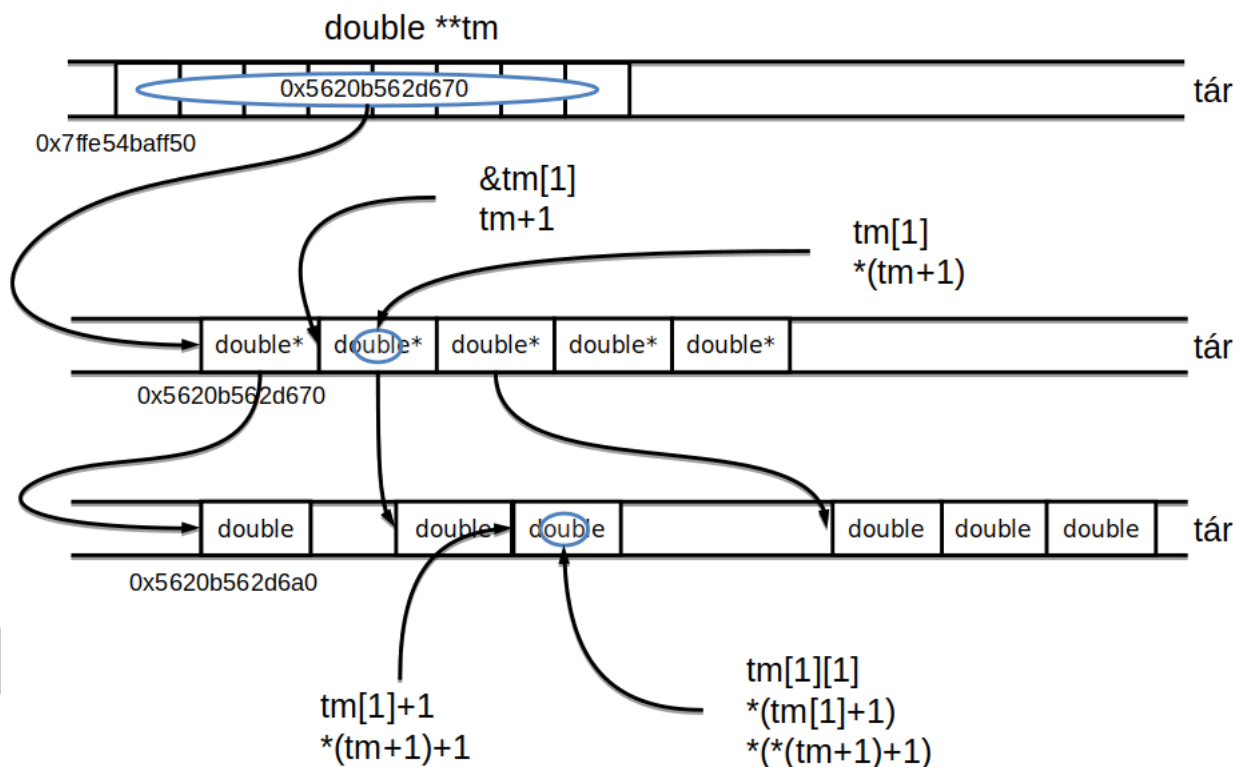
Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/deklaracio.c>

4. fejezet

Helló, Caesar!

4.1. int *** háromszögmátrix



Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/tm.c>

Tanulságok, tapasztalatok, magyarázat... Az `int ***` háromszögmátrixban a főátló alatt és fölött csak 0 szerepelhet. A mátrix neve megtévesztő mivel sorainak és oszlopainak száma megegyezik. Deklarálunk egy `int`-et (aminek most a kezdőértéke legyen 5) illetve a `**tm` mutatót. Aztán malloc segítségével megnézzük, hogy van-e a memóriában hely a `**tm` számára. Ha nincs akkor a progit leáll. Aztán egy `for` ciklusban megnézzük, hogy van-e hely a sorok számára. Mint az előző esetben, ha nincs akkor kilépés történik. Végül pedig kiíratjuk a sorokat ("i") és a pointereket ("j") amelyeket a mátrixunk tartalmaz.

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/e.c>

Tanulságok, tapasztalatok, magyarázat... Az EXOR titkosító programunk egy logikai műveletre épül (vagy). Szépen bitenként hasonlítjuk össze a titkosítandó szöveget és a kulcsunkat amivel titkosítunk. Ha a bitek egyeznek akkor 0-át ha nem akkor egyet kapunk vissza. Minél rövidebb a kulcsunk annál könnyebb a feltörés. Viszont egy erős titkosítás nagyon időigényes feladat, ezért gondolkozzunk csak kicsiben:

```
#define MAX_KULCS 100
#define BUFFER_MERET 256

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

Ebben a csipetben látható, hogy a kulcs és buffer méret konstansként szerepel, így értéke nem változtatható. Konstansokat csupa nagy betűvel definiálunk, így megkülönböztetve egy sima változótól. Aztán maga a main sem szokványos, mivel terminálon keresztül szeretnénk majd argumentumot adni neki. Ezeket az arg.-ra mutató mutatókat az argv-ben tároljuk, számukat pedig az argc-vel számoltatjuk meg.

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

int kulcs_index = 0;
int olvasott_bajtok = 0;

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

Itt tároljuk két tömbben a kulcs méretét illetve az olvasott bajtokat. (Ezek számát "maximalizáltuk" a kód elején). A kulcs_index segítségével járjuk be a tömböt, valamint az olvasott_bajtok-kal számoljuk a bajtokat. Az strlen-el kapjuk meg a kulcs méretet, ami az argv[1] hosszát adja vissza nekünk egy string formájában. az strncpy pedig szépen karakterenként másolja az argv[1]-ben lévő stringeket. A MAX_KULCS pedig határt szab a kulcs méretének.

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

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

```
}  
}
```

Ameddig a read beolvassa a bufferből a megadott mennyiségű bájtot addig a while ciklusunk igaz lesz. Ez a mennyiség a BUFFER_MERET-ben van definiálva (jelen esetben ez 256). Visszkapjuk a beolvasott bájtok számát. A bufferben tárolt karaktereket egyenként EXORozzuk a kulcs tömb megfelelő elemével. A kulcs_indexet egyel növeljük mindaddig amíg akkora nem lesz mint a megadott kulcs_meret. Ezután pedig végső lépésként kiírjuk a bufferben tárolt tartalmat.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/ExorTitkos%C3%ADt%C3%B3.java>

Tanulságok, tapasztalatok, magyarázat... A program ugyanazt a feladatot végzi el mint a c-s társa csak java nyelvben leprogramozva. A java mostanában talán az egyik legfelkapottabb programnyelv mivel sok hasznos C és C++ funkciót tartalmaz illetve talán legnagyobb előnye az objektum-orientáltság ami nekünk embereknek szinte természetes, mivel mindent tárgyként kezelünk.

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/t.c>

Tanulságok, tapasztalatok, magyarázat... A C Exortörő programunk az előzőekben exorosan letitkosított szöveget képes feltörni. A programunk elején nagyon fontos az olvasásbuffer és a kulcsméret pontos definiálása, hiszen ezeknek nagy szerepe van a helyes működésben. Az alapkoncepció az, hogy egy szöveg már fel van törve tehát tiszta szöveg lesz ha tartalmazza az átlagosan előforduló magyar szavakat (Ezek a szavak a: hogy, nem, az, ha). Ha ez így van akkor a program kiírja a visszafejtés eredményét. Egyébként még a titkos szöveg kategóriába tartozik alias nincs még feltörve és null pointert kapunk vissza. A feltörés maga a titkosításra alkalmazott kulcs kiszámításával történik. Az exorozás és maga az exortörés ugyanazon argumentumokkal dolgozik és a titkosított szöveg mindenegybes karakterén végrehajtódik. Egy egyszerű while ciklus segítségével beolvastatjuk magát a titkosított szöveget. (Itt köszön vissza az olvasásbuffer pontos vagy pontatlan definiálása.) Aztán jönnek az egymásba ágyazott for ciklusok melyek számát az határozza meg, hogy mekkora a kulcsméret. Itt jönnek létre a lehetséges kulcsok amelyeket szépen rápróbálunk a titkosított szövegre. Majd végül a sikeres törés után megkapjuk a kulcsot illetve magát a tiszta szöveget.

4.5. Neurális OR, AND és EXOR kapu

R

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

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

Tanulságok, tapasztalatok, magyarázat...

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/ql.hpp>

Tanulságok, tapasztalatok, magyarázat... A perceptron a gépi tanulásban és az adatbányászatban játszik fontos szerepet. Egy osztályba soroló algoritmus ami bináris inputot tud különböző osztályokba besorolni előre meghatározott próbálkozás alapján melyek száma véges. A mi programunk inputja nem más mint a mandelbrot halmaz által létrehozott kép.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/mandelpngt.c%2B%2B>

Tanulságok, tapasztalat, magyarázat... A Mandelbrot halmazt Benoit Mandelbrot találta meg a komplex számsíkon. Komplex számok azok a számok, amelyekkel válaszolni tudunk egyébként értelmezhetetlen kérdésekre. A Mandelbrot halmaz egy sík origójú, 4 oldalúhosszú négyzetbe helyezett rács. A rács pontjait pedig komplex számoknak feleltetjük meg. A rács minden pontját megvizsgáljuk a $z_{n+1} = z_n^2 + c$, (0 kisebb egyenlő n) képlettel. C a vizsgált rácspon. A z_0 pedig az origó. • $z_0 = 0$ • $z_1 = 0^2 + c = c$ • $z_2 = c^2 + c$ • $z_3 = (c^2 + c)^2 + c$ • $z_4 = ((c^2 + c)^2 + c)^2 + c$ • ... Az origóból kiindulva (z_0) átlépünk a rács első pontjába a $z_1 = c$ -be, aztán a c -től függően a további z -kbe. Ha így kijutunk a 2 sugarú körből, akkor a vizsgált rácspon nem a Mandelbrot halmaz eleme. Csak véges sok z -t nézünk meg minden rácsponthoz. Ha nem lép ki akkor feketére színezzük, ezzel jelezve ,hogy az a c rácspon a halmaz része.

Először is definiáljuk egy konstansban a létrehozandó képünk méretet, valamint az iterációs határunkat:

```
#define MERET 600
#define ITER_HAT 32000
```

Majd jöhet az értékkészlet valamint az értelmezési tartomány, illetve az előbb definiált adatok hívása:

```
float a = -2.0, b = .7, c = -1.35, d = 1.35;
int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;
```

Aztán a számítások a következő részben történnek két for ciklus segítségével de előbb deklarálnunk kell a lépésközünket valamint változókat amiben tárolni fogjuk a "c" illetve a "z" komplex számok imaginárius és valós részeit.

Változók és lépésköz deklarációja:

```
float dx = (b - a) / szelesseg;
float dy = (d - c) / magassag;
float reC, imC, reZ, imZ, ujureZ, ujimZ;
int iteracio = 0;
```

A for ciklusok:

```
for (int j = 0; j < magassag; ++j)
{
    //sor = j;
    for (int k = 0; k < szelesseg; ++k)
    {
        // c = (reC, imC) a rács csomópontjainak
        // megfelelő komplex szám
        reC = a + k * dx;
        imC = d - j * dy;
        // z_0 = 0 = (reZ, imZ)
        reZ = 0;
        imZ = 0;
        iteracio = 0;
        // z_{n+1} = z_n * z_n + c iterációk
        // számítása, amíg |z_n| < 2 vagy még
        // nem értük el a 255 iterációt, ha
        // viszont elértük, akkor úgy vesszük,
        // hogy a kiindulási c komplex számra
        // az iteráció konvergens, azaz a c a
        // Mandelbrot halmaz eleme
        while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
        {
            // z_{n+1} = z_n * z_n + c
            ujreZ = reZ * reZ - imZ * imZ + reC;
            ujimZ = 2 * reZ * imZ + imC;
            reZ = ujreZ;
            imZ = ujimZ;

            ++iteracio;
        }

        kepadat[j][k] = iteracio;
    }
}
```

Az értelmezési tartományon kezdjük a lépegetést, és kiszámolunk egy z értéket a c számhoz minden iterációban. Itt jön a képbe a `while` ciklus, ez ugyanis addig igaz (tehát fut a számítás) ameddig a z négyzete kisebb 4-nél, illetve az iterációs határt még nem értük el. C -t akkor mondhatjuk a Mandelbrot halmaz elemének ha elértük az iterációs határt. (a kiindulási c komplex számra az iteráció konvergens)

Ha a felhasználó esetleg rosszul futtatná a programot akkor ezt a hibát iratjuk ki neki. Ebben a pontos futtatási parancs van:

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

    if (argc != 2)
```

```
{
    std::cout << "Hasznalat: ./mandelpng fajlnev";
    return -1;
}
```

Itt pedig létrehozunk egy üres png-t. Ebbe rakjuk majd a képünket a halmazról.

```
png::image < png::rgb_pixel > kep (MERET, MERET);
```

Szint adunk a pixeleknek, ezzel befolyásolva a kirajzolt halmazunk színét

```
kep.set_pixel (k, j,
               png::rgb_pixel (255 -
                               (255 * kepadat[j][k]) / ITER_HAT ←
                               '
                               255 -
                               (255 * kepadat[j][k]) / ITER_HAT ←
                               '
                               255 -
                               (255 * kepadat[j][k]) / ITER_HAT ←
                               ));
```

Legvégül pedig abba a fájlba amit felhasználó megadott argumentumként belenyomjuk a kész képet és a sikeres művelet végét egy "mentve" kiírással nyugtázzuk.

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

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/3.1.2.cpp>

Tanulságok, tapasztalat, magyarázat... Ez az előző feladat egy továbbfejlesztett változata. Tartalmazza az `std::complex` osztályt amelyben a nevéből is adódóan komplex számok vannak. Segítségével a felhasznált változók számát tudjuk egyre csökkenteni ami egyszerűsíti így könnyebben értelmezhetőbbé teszi kódunkat.

Itt láthatjuk magát az `std::complex` osztályt a forrásunkba implementálva:

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

std::complex<double> z_n ( 0, 0 );
iteracio = 0;
while ( std::abs ( z_n ) < 4 && iteracio < iteraciosHatar )
{
    z_n = z_n * z_n + c;

    ++iteracio;
}
```

Maga a komplex osztály double-öket tartalmaz (két rész: valós és imaginárius) Magát a képzési szabályt is simán beírhatjuk.

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

Valamint még egy kicsi de szerintem érdekes változtatást mutat be a fent idézett kódcsipet: már futás közben láthatjuk hány százaléknál jár a számítás.

5.3. Biomorfok

Megoldás videó: <https://youtu.be/IJMbGRzY76E>

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

Tanulságok, tapasztalatok, magyarázat... Az ún. biomorfokra Clifford Pickover talált rá 1986-ban. Legjobb tudomása szerint ezek valamiféle természeti törvények voltak. Írt is rá egy a Julia halmazokat rajzoló programot. A különbség a Mandelbrot halmaz és a Julia halmazok között az hogy a komplex számsíkban elhelyezkedő c az előbbiben változó, utóbbiban pedig állandó. Ahogy a kódban láthatjuk a Mandelbrot halmazokat alkalmazó programban a c befutja a vizsgált összes rácspontot amíg a Julia halmazt alkalmazóban végig egy értéke van. Ennek a feladatnak a megoldásához elég módosítani a Mandelbrot halmazt kiszámoló forrást és átnevezni a változókat.

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

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

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

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

```
reC = atof ( argv[9] );  
imC = atof ( argv[10] );  
R = atof ( argv[11] );
```

A hasonlóság szemmel látható, habár van egy pár különbség is. A küszöbszámot illetve a cc konstans-t most a felhasználótól kérjük, illetve a parancssori argumentumok száma megugrott 10-re. Ha a felhasználó ezeket mégsem használná ki (ami persze nem kötelező) akkor az alapértelmezett érték lép életbe.

A felhasználó segítségére újra implementálunk egy hibaüzenetet a hibás futtatás esetén:

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

Itt is létrehozuk a png-t ami majd tartalmazni fogja a halmazunkat:

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

A rácspontokon ismét két egymásba épített for ciklus vezet át minket. Valamint itt használunk egy harmadikat is, amivel amíg nem érjük el az iterációs határt vagy nem teljesítődik a feltétel (maga az eredeti program által tartalmazott bug)

```
if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
```

számítjuk a függvényértékeket.

```
double dx = ( xmax - xmin ) / szelesseg;  
double dy = ( ymax - ymin ) / magassag;  
  
std::complex<double> cc ( reC, imC );  
  
std::cout << "Szamitas\n";  
  
// j megy a sorokon  
for ( int y = 0; y < magassag; ++y )  
{  
    // k megy az oszlopokon  
  
    for ( int x = 0; x < szelesseg; ++x )  
    {  
  
        double reZ = xmin + x * dx;  
        double imZ = ymax - y * dy;  
        std::complex<double> z_n ( reZ, imZ );  
  
        int iteracio = 0;  
        for (int i=0; i < iteraciosHatar; ++i)  
        {
```

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

A kód vége pedig megegyezik a mandelbrotos kódunkkal. Előbb a `kep.set_pixel` segítségével színt adunk a pixeleknek, majd kiíratjuk a létrehozott png-be az eredményt. (Illetve itt is alkalmazzuk a számítás előre-haladtát mutató kis fejlesztésünket).

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

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

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

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/CUDA/mandelpngc_60x60_100.

Tanulságok, tapasztalat, magyarázat... Az Nvidia CUDA technológiája segítségével jelentősen fel tudjuk gyorsítani a Mandelbrot halmazos képünk kirajzolódását. Egy 600x600 darab blokkból álló rács áll rendelkezésünkre amiben mindenegyik blokkhoz tartozik egy szál. Ezzel a program párhuzamos futású lesz. A CUDA használatához nvidia GPU-ra van szükség, ami sajnos nekem nem áll a rendelkezésemre ezért a program futását nem tudtam tesztelni.

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: <https://github.com/kzoltan99/mandel-c->

Megoldás videó:

Tanulságok, tapasztalat, magyarázat... Ez a program egy több összefüggő kódból hozható létre. Lényege az, hogy a már az előbbieken megírt Mandelbrot halmazt kirajzoló progink képes legyen az egérrel történő nagyításra. Ehhez a qt-t hívjuk segítségül. Ez alapjáraton nem része linux rendszerünknek (legalábbis nekem

nem volt) ezért telepítenünk kell. Ezt a "sudo apt-get install build-essential libgl1-mesa-dev" paranccsal érhetjük el terminálunkban. Qmake -projects paranccsal kapunk egy .pro fájlt. Ebbe be kell implementálni a qt +=widgets-et illetve a targetnek egy tetszőleges nevet adni. Aztán újra qmake jön csak kapcsoló nélkül (qmake *.pro), ezután egy make parancs majd végül szokás szerint ./-el futtatjuk az általunk elnevezett progit.

Maga a nagyítási művelet a FrakAblak.cpp fájlban történik. Itt hívjuk a FrakAblak.h headerben lévő mousePressEvent(et (egérgomb lenyomása), mouseMoveEvent(et (egér "vonszolása")-et, mouseReleaseEvent(et (egérgomb felengedése)-et és a keyPressEvent(et (billentyű lenyomása)-et. Ezek segítségével tudjuk implementálni kódunk az egérrel való kattintást, és a nagyítandó terület egér általi kijelölését. Az "n" betű lenyomásával tudjuk a nagyított területet részletesebben kirajzolni. (a kijelölt területen újraszámolja a z-ket)

```
void FrakAblak::mousePressEvent(QMouseEvent* event) {

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

    update();
}

void FrakAblak::mouseMoveEvent(QMouseEvent* event) {

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

    update();
}

void FrakAblak::mouseReleaseEvent(QMouseEvent* event) {

    if(szamitasFut)
        return;

    szamitasFut = true;

    double dx = (b-a)/szelesseg;
    double dy = (d-c)/magassag;

    double a = this->a+x*dx;
    double b = this->a+x*dx+mx*dx;
    double c = this->d-y*dy-my*dy;
    double d = this->d-y*dy;

    this->a = a;
    this->b = b;
    this->c = c;
    this->d = d;
```

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

        update();
    }

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

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

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

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás forrása: <https://github.com/kzoltan99/mandel.java>

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 eltérő kiszámolt szám.

Megoldás videó:

Megoldás forrása:

Java: <https://github.com/kzoltan99/feladatok/blob/master/polargen.java>

C++: <https://github.com/kzoltan99/feladatok/blob/master/polargen.cpp>

Tanulságok, tapasztalatok, magyarázat... térj ki arra is, hogy a JDK forrásaiban a Sun programozói pont úgy csinálták meg ahogyan te is, azaz az OO nemhogy nem nehéz, hanem éppen természetes neked! A megoldásban létrehoztunk egy Polargen nevű osztályt. A konstruktorban megadjuk, hogy még nincs tárolt szám és létrehozunk egy random-szám generátort. Az objektum orientált programozás nem nehéz, hanem természetes. A Java SDK-ban is hasonlóan megírt programrészleteket találhatunk. Számunkra talán azért tűnhet természetesnek mert mi minent objektumként kezelünk. Az algoritmus 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. Hogy páros vagy páratlan lépésben hívtuk-e meg a megfelelő számítást elvégző következő() függvényt, a nincsTárolt logikai változóval jelöljük. Igaz értéke azt jelenti, hogy tárolt lebegőpontos változóban el van tárolva a visszaadandó szám.

Itt történik maga a generálás:

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

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

```
return 0;
}
```

Legenerálunk 10 db random számot, és ezt adjuk magának az algoritmusnak.

6.2. LZW

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/z.c>

A függvény először megvizsgálja, hogy a kapott érték 0-e. Ha 0, akkor megnézi, mutató által címzett csomópontok van-e 0-s gyermeke. Ha van, akkor a fa mutató az aktuális csomópont bal gyermekére lép. Ha nincs, akkor létrehozza azt, a mutatót a gyökerre állítja. Ha a kapott érték nem 0, a függvény végrehajtja az utasításokat, az aktuális csomópont jobb gyermekére. Az eljárás inorder módon rekurzívan bejárja a bináris fát. Az inorder bejárásnál először a fa bal oldalát járjuk be, majd a gyökerét, végül pedig a jobb oldalát dolgozzuk fel. A eljárás rekurzívan postorder módon bejárja a fát és minden rekurzió végén felszabadítja a részfa gyökerelemét. A felszabadítás előtt meg kell vizsgálni, hogy a részfa gyökere egyenlő-e a teljes fa gyökerével, ha gyökere nem dinamikusán foglalt. Inorder módon bejárjuk a fát, majd felszabadjuk a mutatóit.

Először létrehozzuk magát a binfa struktúrát ahol definiáljuk a binfa típust. A typedef használata elég előnyös, mivel így (itt megadott) más névvel is tudunk hivatkozni az osztályra. Aztán az `uj_elem` függvényben `malloc` segítségével helyet foglalunk a `BINFA` típusú változóknak és visszkapunk egy pointert ami a foglalt területre mutat. Majd deklaráljuk a `kiir`, `ratlag`, `rszoras` és `szabadit` függvényeket melyeket később fogunk használni.

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

typedef struct binfa
{
    int ertek;
    struct binfa *bal_nulla;
    struct binfa *jobb_egy;
    //>itt definiáljuk a binfa típust
} 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;
}

extern void kiir (BINFA_PTR elem);
extern void ratlag (BINFA_PTR elem);
extern void rszoras (BINFA_PTR elem);
extern void szabadit (BINFA_PTR elem);
```

A main hozzuk létre a gyökeret. Mivel itt még se jobboldali se baloldali gyermek sincs még ezért NULL értéket adunk a pointernek, a fát gyökerre állítjuk. Illetve "/" lesz most a gyökerünk értéke. A fát a while ciklusunkban csináljuk meg a már fentebbi (megoldás forrása utáni) részben említett módon.

```
int
main (int argc, char **argv)
{
    char b;
    int egy_e;
    int i;
    unsigned char c;
    //>BinfaPTR== user által definiált típus
    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;  
    }  
}  
}
```

Itt történik a fa kiírása:

```
printf ("\n");  
kiir (gyoker);  
  
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 `ratlag`-nak beadjuk a `fa` pointert, ha ez nem `NULL` akkor a `melyseg`-et növeljük illetve ráengedjük az `ratlag`-ot a jobb és bal gyermekekre is. Ha a jobb és bal gyermek is a legutolsó az összes közül akkor az `atlagdb`-t növeljük egyel. Ha ezzel megvagyunk akkor még a mélységhez hozzáadjuk az `atlagosszeget`. Az `rszoras` ugyanezen az elven működik, annyi a különbség, hogy a végén más műveletet végzünk el: most már az előbb kiszámolt átlagot is használjuk. Kivonjuk a mélységből és a kapott különbséget négyzetre emeljük, és ezt adjuk hozzá a szórásösszeghez.

```
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;
        }
    }
}

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));

    }

}

}

//static int melyseg = 0;
int max_melyseg = 0;

void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        _melyseif (melyseg > maxg);
        max_melyseg = melyseg;
        kiir (elem->jobb_egy);
        // ez a postorder bejáráshoz képest
        // 1-el nagyobb mélység, ezért -1
        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;
    }
}
```

Legvégül pedig a szabadít függvény. Ennek segítségével szabadítjuk fel a területet melyet lefoglaltunk eddig. Ha van gyermeke az elemnek akkor azokra is hívjuk a függvényt. A szabadításban a free függvény áll rendelkezésünkre.

```
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: A megoldás forrása az előző feladat forrásának egy iterációja. csupán a kiir függvényt kell átírnunk.

A preorder fabejárás abban különbözik az inorder eljárástól, hogy ebben először a részfa gyökerét dolgozzuk fel majd a részfa bal oldalát és jobb oldalát járjuk be:

```
void  
kiir (BINFA_PTR elem)  
{  
    if (elem != NULL)  
    {  
        ++melyseg;  
        if (melyseg > max_melyseg)  
            max_melyseg = melyseg;  
        for (int i = 0; i < melyseg; ++i)  
            printf ("---");  
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem-> ←  
            ertek,  
            melyseg-1);  
        kiir (elem->jobb_egy);  
        // ez a postorder bejáráshoz képest  
        // 1-el nagyobb mélység, ezért -1  
        kiir (elem->bal_nulla);  
        --melyseg;  
    }  
}
```

A postorder fabejárás abban különbözik az inorder eljárástól, hogy ebben először a részfa bal oldalát és jobb oldalát járjuk be. Végül pedig részfa gyökerét dolgozzuk fel:

```
void  
kiir (BINFA_PTR elem)  
{  
    if (elem != NULL)  
    {  
        ++melyseg;  
        if (melyseg > max_melyseg)  
            max_melyseg = melyseg;  
        kiir (elem->jobb_egy);  
        // ez a postorder bejáráshoz képest  
        // 1-el nagyobb mélység, ezért -1
```

```
        kiir (elem->bal_nulla);
        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem-> ←
                ertek,
                melyseg-1);
        --melyseg;
    }
}
```

6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/z3a7.cpp>

Ez a megoldás az előző feladat megoldására épül. A különbség, hogy a fát kezelő függvényeket a Binfa osztályba rendezzük, illetve a Binfa osztály privát részébe helyezzük a Node-ot. A binfa osztályban túlterheljük a balra bitshift operátort, amely a fa építését látja el. Ezt a c++ használata miatt tudjuk megvalósítani, hiszen c-ben még nem lehetett operátort túlterhelni. Bármint így sem tudunk az operátorral megcsináltatni de tudjuk alakítani bizonyos szinten annak függvényében, hogy mire szeretnénk használni.

C++-ban újdonság még a classok (osztályok) bevezetése. Tehát már mi is így vezetjük be forrásunkba a binfát nem pedig struktúraként:

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

Itt történik az operátor túlterhelése valamint a feladat elején leírt bitshiftelése. Az történik, hogy megnézzük, hogy van-e egyes illetve nullás gyermeke a fának. Ha van akkor arra a gyermekre állítjuk a fát. Ha nincs akkor létrehozunk egyet ujNullasGyermek (vagy ujEgyesGyermek) néven. A new függvény segítségével foglalunk helyet. Mindkét esetben a fa pointerünket az adott gyermekre állítjuk.

```
void operator<< (char b)
{
    if (b == '0')
    {
        if (!fa->nullasGyermek ())
```



```

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

```

Az új c++-os egyszerűsített kiir függvényünk pedig 0-ra állítja a mélységet valamint hív egy másik kiir függvényt (c++-ban már akár 2 ugyanolyan függvény is lehet egymásba ágyazva, a fordító képes különbséget tenni köztük).

```

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

```

A mélységet, átlagot és a szórást get-tel meghívjuk a privát részből. Megint túlterheljük az operátort, kimenetet kapunk és átadunk neki egy kimenetet és egy objektum referenciát. Végül pedig előhívjuk a kiir-t.

```

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

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

```

```
        kiir (&gyoker, os);  
    }
```

Az lzwbinfa osztályunk privát részébe kerül a csomópont osztály. Ugyanúgy mint a c-s változatban a csomópont eredeti, alapértelmezett értéke "/" lesz. A balnullát és a jobbegyét 0-ra állítjuk. A nullás és egyesgyermek a megfelelő gyermekre mutatót ad vissza. Az átadott csomópontra állítják a gyermekek mutatóját az ujNullasGyermek és az ujEgyesGyermek. A getBetu függvény segítségével olvassuk a bemenetet.

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

Aztán jöhet a fa mutató deklarációja:

```
Csomopont *fa;
```

```
int melyseg, atlagosszeg, atlagdb;
double szorasosszeg;

//nocopy
LZWBinFa (const LZWBinFa &);
LZWBinFa & operator= (const LZWBinFa &);
```

A c-s verzióban végbement számítások így néznek ki a fejlesztett c++-os forrásunkban. Itt történik az osztályon belüli függvények definiálása is.

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

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

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

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

};

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

```
}

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

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

    rszoras (&gyoker);

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

    return szoras;
}

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

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

```
--melyseg;
if (elem->egyenesGyermek () == NULL && elem->nullasGyermek () == NULL <=
)
{
    ++atlagdb;
    atlagosszeg += melyseg;
}
}

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

Aztán segítünk a felhasználónak a helyes használatban. Ez tényleg nagy segítség mert őszintén szólva elsőre én magam sem tudtam hogyan futtassam a programot. Egyszerűen meghatározzuk hány db (esetünkben 4) parancssori argumentum szükséges a helyes futáshoz. A usage függvény segítségével írjuk ki a helyes módszert. Fontos még az is, hogy a harmadik argumentum második elem o legyen. Ezt is ellenőrizzük és ha nem az akkor újra hibát küldünk és ismét leírjuk a helyes futtatást. Ha nem létező bemeneti fájlt ad meg a felhasználó akkor az fstream-et hívjuk segítségül és nemes egyszerűséggel tudatára adjuk, hogy a fájl nem létezik. Ha minden jól működik és helyesen van a program elindítva akkor újfent az fstream segítségével létrehozuk a kiFile-t melynek a negyedik argumentumot adjuk meg első paraméterként ugyanis ebbe a fájlba fog történni a fa írása. A b változóba pedig bitenként fogjuk olvasni a bemenetet.

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

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

```
        return -1;
    }

    char *inFile = *++argv;

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

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

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

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

    unsigned char b;
    LZWBinFa binFa;
```

A binfát itt is while ciklusok segítségével rajzoljuk meg ugyanúgy mint a c-s változatban, de persze ebben is vannak eltérések. Kezdődik a b változó működése, ha valami hibát észlel a program olvasás közben tehát törésre ütközik akkor megállítja a ciklust. A fa építése a második ciklusban történik. Karakter észlelése során igaz, törés esetén hamis értéket adunk a változónak a kommentben. Végül pedig egy for ciklust is használunk. A ciklus szépen bitenként végigfut a beolvasott karakteren és éseli a b-vel. Ha egyet kapunk akkor egyest ír a fába egyébként pedig nullát.

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

bool kommentben = false;

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

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

    if (b == 0x0a)
    {
        // újsor
```

```
        kommentben = false;
        continue;
    }

    if (kommentben)
        continue;

    if (b == 0x4e)    // N betű
        continue;

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

A forrásunk legvégén pedig szépen bele bitshifteljük a fát a kimeneti fájlba. Plusz ugyanezt tesszük a korábban számolt mélység, átlag és szórás értékekkel.

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

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

return 0;
```

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: <https://github.com/kzoltan99/feladatok/blob/master/z3a73.cpp>

A különbség az előző feladathoz képest, hogy ebben a gyökérelemre is már egy mutató mutat, azért a Binfa konstruktorában létre kell hozni a gyökérobjektumot. A program mostmár a gyökérelemet adja át nem a referenciáját.

A gyökér mutatóvá alakítása:

```
Csomopont* gyoker;
```

A fának és a gyöker mutatónak új területet foglalunk le, de ezt fel is kell szabadítani, ezért változtatni kell a destruktoron is a következőképpen:

```
LZWBinFa () {
    gyoker = new Csomopont ('/');
    fa = gyoker;
}

~LZWBinFa ()
{
    szabadit (gyoker->egyenesGyermekek ());
    szabadit (gyoker->nullasGyermekek ());
    delete(gyoker);
}
```

A forrásunk többi része megegyezik az eredetivel.

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/feladatok/blob/master/z3a9.cpp>

A másoló szemantika lényege, hogy az értékül kapott bináris fát értékül adja az eredeti fának, minden érték másolásával. A mozgató szemantika pedig úgy működik, hogy az eredeti bináris fa gyökerét "kicseréli" az értékül kapott fa gyökerével, az értékül kapott fa gyökerének gyermekeit nullpointerre állítja, hogy az eredetileg létező fa ne törlődjön le.

Maga a mozgató konstruktor a forrásba implementálva:

```
LZWBinFa (LZWBinFa && regi) {
    std::cout << "LZWBinFa move ctor" << std::endl;
    gyoker = nullptr;
    *this = std::move(regi);
    // gyoker = regi.gyoker;
    // regi.gyoker = nullptr;
}

LZWBinFa& operator=(LZWBinFa && regi){
    std::cout << "LZWBinFa move assign" << std::endl;
    std::swap(gyoker, regi.gyoker);
    return *this;
}
```


7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk

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

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

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

Tanulságok, tapasztalatok, magyarázat... A program a hangyák kommunikációját szimulálja. A cellákra osztott képernyőn a hangyák megkeresik legerősebb feromonnal rendelkező társukat és feléjük mennek. Az utakat a halvány kékeszöld négyzetek jelentik. A feromonon szint folyamatosan csökken, de ha valahová belép a hangya ott megnő a szint. Az értékeket parancssori argumentumokkal adjuk meg.

7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/etjatek-c->

Tanulságok, tapasztalatok, magyarázat... Az életjátékot John Conway találta ki. A játékos szerepe annyi, hogy megad egy kezdőalakzatot, és azután csak figyeli az eredményt. Matematikai szempontból a sejtautomaták közé tartozik. A játék egyes lépéseinek eredményét számítógép számítja ki. A játék, a felfedezése

után nagyon népszerű lett amerikában. Komoly matematikai és filozófiai vonatkozásai vannak. Szabályok:

- A sejt túléli a kört, ha két vagy három szomszédja van.
- A sejt elpusztul, ha kettőnél kevesebb (elszigetelődés), vagy háromnál több (túlnépesedés) szomszédja van.
- Új sejt születik minden olyan cellában, melynek környezetében pontosan három sejt található.

A játék lépéseinek sorrendje:

- Az elhaló sejtek megjelölése
- A születő sejtek elhelyezése
- A megjelölt sejtek eltávolítása

A sejtablak.h és a sejtablak.cpp tartalmazza azt az osztályt (SejtAblak) ami felelős a kirajzolásért.

```
#ifndef SEJTABLAK_H
#define SEJTABLAK_H

#include <QMainWindow>
#include <QPainter>
#include "sejtszal.h"

class SejtSzal;

class SejtAblak : public QMainWindow
{
    Q_OBJECT

public:
    SejtAblak(int szelesseg = 100, int magassag = 75, QWidget *parent = 0);
    ~SejtAblak();
    // Egy sejt lehet élő
    static const bool ELO = true;
    // vagy halott
    static const bool HALOTT = false;
    void vissza(int racsIndex);

protected:
    // Két rácsot használunk majd, az egyik a sejttér állapotát
    // a t_n, a másik a t_n+1 időpillanatban jellemzi.
    bool ***racsok;
    // Valamelyik rácsra mutat, technikai jellegű, hogy ne kelljen a
    // [2][][]-ból az első dimenziót használni, mert vagy az egyikre
    // állítjuk, vagy a másikra.
    bool **racs;
    // Megmutatja melyik rács az aktuális: [racsIndex][][]
    int racsIndex;
    // Pixelben egy cella adatai.
    int cellaSzelesseg;
    int cellaMagassag;
    // A sejttér nagysága, azaz hányszor hány cella van?
    int szelesseg;
    int magassag;
    void paintEvent(QPaintEvent*);
    void siklo(bool **racs, int x, int y);
    void sikloKilovo(bool **racs, int x, int y);

private:
```

```
SejtSzal* életjatek;

};

#endif // SEJTABLAK_H
```

A siklókilövőhöz most egy külön eljárást kell írunk. A koordinátákat egyenként rajzoljuk ki.

```
void SejtAblak::siklo(bool **racs, int x, int y) {

    racs[y+ 0][x+ 2] = ELO;
    racs[y+ 1][x+ 1] = ELO;
    racs[y+ 2][x+ 1] = ELO;
    racs[y+ 2][x+ 2] = ELO;
    racs[y+ 2][x+ 3] = ELO;

}

/**
 * A sejttérbe "ELOlényeket" helyezünk, ez a "sikló ágyú".
 * Adott irányban siklókat lő ki.
 * Az ELOlény ismertetését lásd például a
 * [MATEK JÁTÉK] hivatkozásban /Csákány Béla: Diszkrét
 * matematikai játékok. Polygon, Szeged 1998. 173. oldal./,
 * de itt az ábra hibás, egy oszloppal told még balra a
 * bal oldali 4 sejtes négyzetet. A helyes ágyú rajzát
 * lásd pl. az [ÉLET CIKK] hivatkozásban /Robert T.
 * Wainwright: Life is Universal./ (Megemlíthetjük, hogy
 * mindkettő tartalmaz két felesleges sejtet is.)
 *
 * @param racs a sejttér ahová ezt az állatkát helyezzük
 * @param x a befoglaló téglal bal felső sarkának oszlopa
 * @param y a befoglaló téglal bal felső sarkának sora
 */
void SejtAblak::sikloKilovo(bool **racs, int x, int y) {

    racs[y+ 6][x+ 0] = ELO;
    racs[y+ 6][x+ 1] = ELO;
    racs[y+ 7][x+ 0] = ELO;
    racs[y+ 7][x+ 1] = ELO;

    racs[y+ 3][x+ 13] = ELO;

    racs[y+ 4][x+ 12] = ELO;
    racs[y+ 4][x+ 14] = ELO;

    racs[y+ 5][x+ 11] = ELO;
    racs[y+ 5][x+ 15] = ELO;
    racs[y+ 5][x+ 16] = ELO;
    racs[y+ 5][x+ 25] = ELO;

    racs[y+ 6][x+ 11] = ELO;
```

```
racs[y+ 6][x+ 15] = ELO;
racs[y+ 6][x+ 16] = ELO;
racs[y+ 6][x+ 22] = ELO;
racs[y+ 6][x+ 23] = ELO;
racs[y+ 6][x+ 24] = ELO;
racs[y+ 6][x+ 25] = ELO;

racs[y+ 7][x+ 11] = ELO;
racs[y+ 7][x+ 15] = ELO;
racs[y+ 7][x+ 16] = ELO;
racs[y+ 7][x+ 21] = ELO;
racs[y+ 7][x+ 22] = ELO;
racs[y+ 7][x+ 23] = ELO;
racs[y+ 7][x+ 24] = ELO;

racs[y+ 8][x+ 12] = ELO;
racs[y+ 8][x+ 14] = ELO;
racs[y+ 8][x+ 21] = ELO;
racs[y+ 8][x+ 24] = ELO;
racs[y+ 8][x+ 34] = ELO;
racs[y+ 8][x+ 35] = ELO;

racs[y+ 9][x+ 13] = ELO;
racs[y+ 9][x+ 21] = ELO;
racs[y+ 9][x+ 22] = ELO;
racs[y+ 9][x+ 23] = ELO;
racs[y+ 9][x+ 24] = ELO;
racs[y+ 9][x+ 34] = ELO;
racs[y+ 9][x+ 35] = ELO;

racs[y+ 10][x+ 22] = ELO;
racs[y+ 10][x+ 23] = ELO;
racs[y+ 10][x+ 24] = ELO;
racs[y+ 10][x+ 25] = ELO;

racs[y+ 11][x+ 25] = ELO;

}
```

A szabályok pedig megint más fájlokban kaptak helyet, a kód "felbontása" több részre az átláthatóságot segíti. A szabályok a `sejtszal.cpp`-ben illetve a `sejtszal.h`-ban kapnak helyet.

```
void SejtSzal::idoFejlodes() {

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

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

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

```
if(racsElotte[i][j] == SejtAblak::ELO) {
    /* Élő élő marad, ha kettő vagy három élő
    szomszédja van, különben halott lesz. */
    if(elok==2 || elok==3)
        racsUtana[i][j] = SejtAblak::ELO;
    else
        racsUtana[i][j] = SejtAblak::HALOTT;
} else {
    /* Halott halott marad, ha három élő
    szomszédja van, különben élő lesz. */
    if(elok==3)
        racsUtana[i][j] = SejtAblak::ELO;
    else
        racsUtana[i][j] = SejtAblak::HALOTT;
}
}
}
racsIndex = (racsIndex+1)%2;
}

/** A sejttér időbeli fejlődése. */
void SejtSzal::run()
{
    while(true) {
        QThread::msleep(varakozas);
        idoFejlodes();
        sejtAblak->vissza(racsIndex);
    }
}

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

7.4. BrainB Benchmark

Megoldás videó:

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

Tanulságok, tapasztalatok, magyarázat... A BrainB Benchmark feladata az esport tehetségek felkutatása. Maga a benchmark: a játékban elveszítjük a karakterünket, mennyi ideig tart megtalálnunk, és ha megtaláltuk, mennyi ideig tart elveszítenünk. bit/sec képernyőváltásokkal méri a sebességünket. A feladat annyi, hogy a kurzort rajta kell tartani a Samu Entropyn, minnél tovább tartod rajta, annál több jelenik meg, ezzel nehezítve a dolgot. Ha elveszted, akkor lassabban kezdenek el megjelenni a karakterek azért, hogy kicsit

könnyítsenek a dolgodon. Az egész játék 10 percig tart és a végén megkapott kép minnél bonyolultabb annál jobb a teljesítmény amit elértél. AZ eredményt egy fájlban kapod meg a teljesítés után.

DRAFT

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

aa Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.2. Szoftmax R MNIST

R

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.4. Deep dream

Keras

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.5. Robotpszichológia

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó:

Megoldás forrása:

9.2. Weizenbaum Eliza programja

Éleszd fel Weizenbaum Eliza programját!

Megoldás videó:

Megoldás forrása:

9.3. Gimp Scheme Script-fu: króm effekt

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

Megoldás videó: https://youtu.be/OKdAkI_c7Sc

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

Tanulságok, tapasztalatok, magyarázat...

9.4. Gimp Scheme Script-fu: név mandala

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

Megoldás videó: https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelese_a_scheme_programozasi_nyelv

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

Tanulságok, tapasztalatok, magyarázat...

9.5. Lambda

Hasonlítsd össze a következő programokat!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9.6. Omega

Megoldás videó:

Megoldás forrása:

DRAFT

III. rész

Második felvonás

**Bátf41 Haxor Stream**

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

DRAFT

10. fejezet

Helló, Arroway!

10.1. 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 UDPROG repó: source/labor/polargen)

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/prog2-feladatok>

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban a polártranszformációs generátor megírása volt a feladat. A szemléltetés a Bátfai Tanrár úr által megírt kódon keresztül történik, hiszen ez a Sun által írt program egy egyszerűbb, könnyebben megérthető változata. A működési elv egyszerű:

```
public class PolárGenerátor {  
  
    boolean nincsTárolt = true;  
    double tárolt;  
  
    public PolárGenerátor() {  
        nincsTárolt = true;  
    }  
}
```

Tehát ebben a részben annyit csinál a program, hogy boolean típusú változó segítségével megnézi, hogy van-e eltárolt normális. Ha nincs akkor a következő módon állítunk elő kettőt:

```
public double következő() {  
    if (nincsTárolt) {  
        double u1, u2, v1, v2, w;  
        do {  
            u1 = Math.random();  
            u2 = Math.random();  
            v1 = 2 * u1 - 1;  
            v2 = 2 * u2 - 1;  
            w = u1 * u2 + v1 * v1;  
            if (w > 1) continue;  
            tárolt = u1 + v1 * i;  
            nincsTárolt = false;  
        } while (true);  
    }  
    return tárolt;  
}
```

```
v2 = 2 * u2 - 1;
w = v1 * v1 + v2 * v2;
} while (w>1);
double r = Math.sqrt((-2 * Math.log(w)) / w);
tárolt = r * v2;
nincsTárolt = !nincsTárolt;
return r * v1;
} else {
    nincsTárolt = !nincsTárolt;
    return tárolt;
```

Ebből a kettőből egyet eltárolunk és a másikat fogjuk a végrehajtásra felhasználni, vagyis ezzel fog a következő() függvény visszatérni. Végül pedig a mainben használjuk fel a következő függvény-t és íratjuk ki a standard outputra az eredményt.

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

De a JDK-ban a Sunos megoldás mégiscsak különbözik a miénktől, mégpedig a synchronized public double használatában. Ez annyit tesz csak, hogy a program futását korlátozza egyetlen egy szála.

```
synchronized public double nextGaussian() {
    // See Knuth, ACP, Section 3.4.1 Algorithm C.
    if (haveNextNextGaussian) {
        haveNextNextGaussian = 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;
    }
}
```

10.2. Homokozó

Írjuk át az első védelési programot (LZW binfa) C++ nyelvről Java nyelvre, ugyanúgy működjön! Mutassunk rá, hogy gyakorlatilag a pointereket és referenciákat kell kiírtani és minden máris működik (erre utal a feladat neve, hogy Java-ban minden referencia, nincs választás, hogy mondjuk egy attribútum pointer,

referencia vagy tagként tartalmazott legyen). Miután már áttettük Java nyelvre, tegyük be egy Java Servletbe és a böngészőből GET-es kéréssel (például a böngésző címsorából) kapja meg azt a mintát, amelynek kiszámolja az LZW binfáját!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

10.3. „Gagyi”

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/prog2-feladatok/tree/master>

Tanulságok, tapasztalatok, magyarázat...

Ha az integer értékét úgy adjuk meg, hogy az -128 és 127 közé essen akkor nem lesz végtelen ciklusunk. Ez azért van így mert ebben a tartományban az Integer ugyanazt az objektumot fogja felhasználni, csupán más értékek fognak hozzárendelődni. Ezzel azt érjük el, hogy a t!=x -re minden egyes lefutás során hamis értéket fogunk kapni. Ezt például a következő képpen érhetjük el:

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

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

Ha viszont egy nagyon csekély változtatást alkalmazunk a kis kódunkban, máris végtelen ciklust kapunk:

```
public class Gagyi_2 {

    public static void main(String[] args) {

        Integer x = 128;
        Integer t = 128;

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

```
}  
}
```

10.4. Yoda

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A Yoda egy olyan programozási stílus, ahol a kifejezések sorrendét felcseréljük, mégpedig olyan módon, hogy a konstans kifejezés a bal oldalra kerül. Ez nem változtatja meg a program működését. Azért nevezték el Yodaról ezt a módszert mert a Star Wars filmekben ő is így beszélt, tehát nem helyes nyelvtani sorrendben mondta a mondatokat. Ez a módszer nagyon hatásos akkor ha el akarjuk kerülni a nullpointeres hibákat. Ezt a módszert a null pointeres hibák ellen használjuk. Először is nézzünk egy olyan példát ahol nem használjuk a Yoda conditions és ez problémát okoz:

```
class yoda  
{  
    public static void main(String[] args)  
    {  
        String myString = null;  
        if (myString.equals("valami"))  
        {  
            System.out.println("semmi");  
        }  
    }  
}
```

Ha ezt a kis egyszerű csipetet így használnánk akkor `NullPointerException`-et kapunk és leáll. Ha viszont használjuk Yoda mester módszerét akkor a program lefut:

```
class yoda  
{  
    public static void main(String[] args)  
    {  
        String myString = null;  
        if ("valami".equals(myString))  
        {  
            System.out.println("semmi");  
        }  
    }  
}
```


10.5. Kódolás from scratch

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/prog2-feladatok/blob/master/PiBBP.java>

Tanulságok, tapasztalatok, magyarázat...

A BBP (Bailey-Borwein-Plouffe) algoritmus a Pi hexa jegyeit kiszámoló osztály. Az alapja a BPP formula amit 1995-ben találtak fel. Ez alapján a matematikai képlet alapján működik. Pi tizediktől a tizenötödik hatványáig számolja ki a számokat a program.

11. fejezet

Helló, Liskov!

11.1. Liskov helyettesítés sértése

Írjunk 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 (93-99 fólia) (számos példa szerepel az elv megsértésére az UDPROG repóban, lásd pl. [source/binom/Batfai-Barki/madarak/](#))

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/prog2-feladatok/blob/master/liskovsert.cpp>

Tanulságok, tapasztalatok, magyarázat...

A Liskov féle helyettesítési elv lényege: Például ha S altípusa T-nek, akkor minden olyan helyen ahol T-t felhasználjuk S-t is be tudjuk helyettesíteni de a programrész tulajdonságai így sem változnak. Ha kicsit egyszerűbben szeretnénk kifejezni magunkat akkor mondhatjuk azt is, hogy ha S a T osztály leszármazottja akkor be tudjuk helyettesíteni minden olyan helyre ahol T típust várunk. Nézzünk most egy egyszerű példát:

```
class Shape { }

class Ellipse : Shape
{
    public double MajorAxis { get; set; }
    public double MinorAxis { get; set; }
}

class Circle : Ellipse
{
}
```

Az osztályhierarchiában a Kör az Ellipszisből származik, mivel az annak egy különleges esete. Ha ezt vesszük figyelembe akkor egy szerintem érdekes problémába fogunk ütközni. készítsünk egy olyan függvényt ami az ellipszis tengelyét is módosítja. Maga a Liskov elv alkalmazásából adódóan a ezt a függvényt akármelyik Kör objektumra is meg tudjuk hívni. Mi történik akkor, ha egy Kör objektum tengelyei megváltoznak? A válasz egyértelmű: a kör megszűnik kör lenni, de maga az objektum, hibásan ugye, de

megmarad. Végülis az a probléma, hogy azt gondoljuk, hogy a Kör egy Ellipszis. Ennek a matematikai része igaz is, viszont nekünk most a programozási részről kellene megközelíteni a dolgot. Az a baj, hogy hiába vannak a Körnek Ellipszishez hasonló tulajdonságai, nem feltétlenül tudjuk vele betartani az arra vonatkozó szabályokat. Ezt a problémát kiküszöbölni úgy tudjuk ha ún. immutábilis objektumokat használunk. Ez azt jelenti, hogy minden olyan esetben amikor megváltoztatnánk az objektumot akkor egy új jön létre. Vagy éppenséggel azt is megtehetjük, hogy a Kört nem is vesszük bele a hierarchiába hanem helyette mindig Ellipszist használunk.

11.2. Szülő-gyerek

Írjunk Szülő-gyerek Java és C++ osztálydefiníciót, amelyben demonstrálni tudjuk, hogy az ősön keresztül csak az ős üzenetei küldhetőek! https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf (98. fólia)⁴

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Ennek a feladatnak a lényege az, hogy szemléltetni tudjuk azt a jelenséget, hogy a Szülő-Gyerek kapcsolatban a Szülő mindig érti a gyerek üzenetét viszont fordítva ez nincs így. Hiszen ha valamit a gyerek classban definiálunk akkor azt a szülő nem fogja érteni és elszáll a programunk. Mivel vannak olyan osztályok, metódusok amikkel a szülő rendelkezik és ezeket a gyerek örökölni tudja, viszont ez visszafelé már nem működik.

```
class Osztaly{
    public static void main(String[] args){
        Szulo sz = new Gyerek();
        sz.kiir1();
        sz.kiir2();
    }
}

class Szulo{
    public void kiir1(){
        System.out.println("Szulo");
    }
}

class Gyerek extends Szulo{
    public void kiir2(){
        System.out.println("Gyerek");
    }
}
```

Ennek szemléltetésére szerettem volna ezt a kis csipetet bemutatni. a szülő rendelkezik a kiir1 metódussal amit tud tőle örökölni a gyerek is, de ahogy láthatjuk a kódban a gyereknek van egy kiir 2 metódusa is amit viszont a szülő nem ismer és ez okozza nekünk a galibát. A kiir 2-t nem tudjuk meghívni viszont a kiir 1-et simán tudnánk, fordulna, futna a progi probléma nélkül. Javában minden objektum referencia, és a kötés dinamikus de ezzel nem küldhetjük a gyerek által hozott új üzeneteket, tehát az ősön keresztül csak az ős üzeneteit tudjuk küldeni.

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

class Szulo{
public:
    void kiir1(){
        cout << "Szulo" << endl;
    }
};
class Gyerek : public Szulo{
public:
    void kiir2(){
        cout << "Gyerek" << endl;
    }
};
int main(){

    Szulo* sz = new Gyerek();
    sz->kiir1();
    sz->kiir2();
}
```

Itt pedig a c++-os verzió látható. Az eredmény ugyanaz lesz mint Javaban. A példányosításnál hiába hívjuk a kiir2-t hibát fogunk kapni mint ahogy az Javaban is történt.

11.3. Anti OO

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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/pi-bbp-bench>

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban négy különböző programnyelven(c, c++, c# és java) ugyanazt a programot fogjuk futtatni és összevetjük a futási időket. Ez a program az előző fejezetben már taglalt BBP algoritmus ami a a Pi hexadecimális alakjának a 0. pozíciótól számított 10 a hatodikon, hetedikon és nyolcadikon db jegyét határozza meg.

11.4. deprecated - Hello, Android!

Élesszük fel a <https://github.com/nbatfai/SamuEntropy/tree/master/cs> projektjeit és vessünk össze néhány egymásra következőt, hogy hogyan változtak a források!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.5. Hello, Android!

Élesszük fel az SMNIST for Humans projektet! <https://gitlab.com/nbatfai/smnist/tree/master/forHumans/SMNIST>
Apró módosításokat eszközölj benne, pl. színvilág.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.6. Hello, SMNIST for Humans!

Fejleszd tovább az SMNIST for Humans projektet SMNIST for Anyone emberre szánt appá! Lásd az `smnist2_kutatasi_jegyzokonyv.pdf`-ben a részletesebb háttérrel!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.7. Ciklomatikus komplexitás

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A ciklomatikus komplexitás nem más mint egy ún. szoftvermetrika amit 1976-ban talált fel Thomas J. McCabe. Hallhattunk már róla akár McCabe-komplexitás néven is. Képes egy adott forráskód alapján a szoftver komplexitását kiszámolni és ezt konkrétan számunkra értelmezhető számokban ki is tudja fejezni. Az eredményül kapott szám a kiszámítási művelete a gráfelméletre épül. A gráf pontjai és a köztük lévő élek a forrásban lévő elágazások alapján épülnek ki és ezek alapján történik a számítás.

A komplexitás értéke: $M = E - N + 2P$ E: a gráf éleinek száma. N: a gráfban lévő csúcsok száma. P: Az összefüggő komponensek száma A ciklomatikus szám: $M = E - N + P$

A ciklomatikus komplexitás a gráf ciklomatikus száma ami a lehetséges kimeneteket köti össze a bemenetekkel. Tehát a gráf alkotói függvényben lévő utasítások. Ha egyik utasítás után rögtön végre lehet hajtani a másikat akkor van él közöttük ami abból következik, hogy a lineárisan független útvonalakat a metrika közvetlenül számolja a forrásból.

A komplexitás számítására találtam egy nagyon jó kis oldalt amit segítségül hívtam: <http://www.lizard.ws/>

Try Lizard in Your Browser

.java

Analyse

```
public class PolárGenerátor {  
  
    boolean nincsTárolt = true;  
    double tárolt;  
  
    public PolárGenerátor() {  
        nincsTárolt = true;  
    }  
  
    public double következő() {
```

Code analyzed successfully.

File Type

.java

Token Count

239

NLOC

32

Function Name	NLOC	Complexity	Token #	Parameter #
PolárGenerátor::tor	3	1	11	
PolárGenerátor::ö	19	3	137	
PolárGenerátor::main	6	2	57	

11.1. ábra. Példa a komplexitásra

12. fejezet

Helló, Mandelbrot!

12.1. Reverse engineering UML osztálydiagram

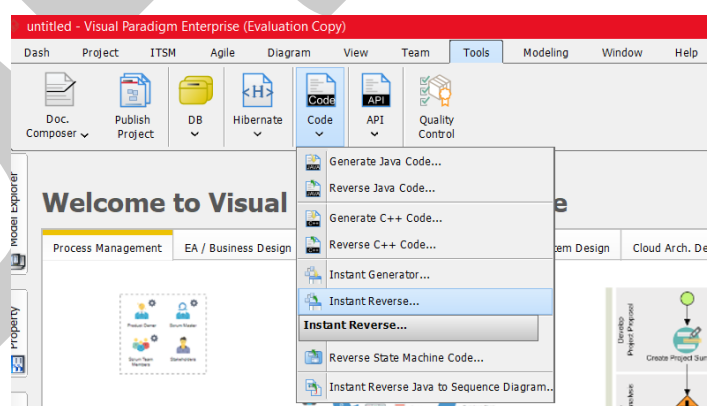
UML osztálydiagram rajzolása az első védési C++ programhoz. Az osztálydiagramot a forrásokból generáljuk (pl. Argo UML, Umbrello, Eclipse UML) Mutassunk rá a kompozíció és aggregáció kapcsolatra a forráskódban és a diagramon, lásd még: https://youtu.be/Td_nIERIEOs. <https://arato.inf.unideb.hu/batfai.norbert/UD>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

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/prog2-feladatok/blob/master/Binfa.cpp>

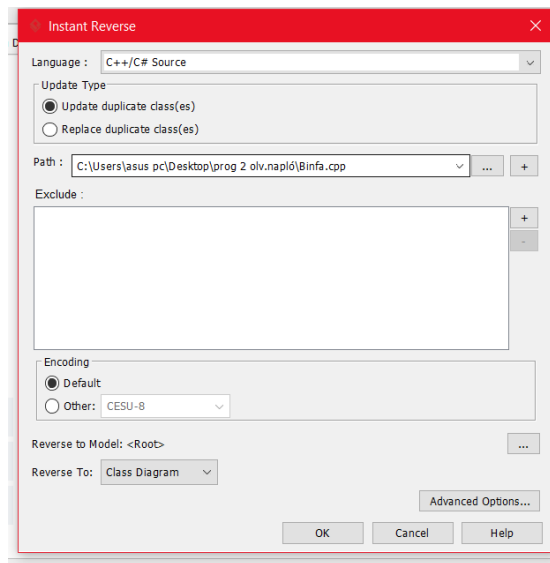
Tanulságok, tapasztalatok, magyarázat...

Itt a C++-os binfát kellett legenerálni egy uml osztálydiagrammba. Ezt a feladatot a Visual Paradigm segítségével oldottam meg. A feladat viszonylag egyszerű volt egy pár kattintással meg lehetett oldani. Először is a visual paradigm-ban kiválasztjuk a tools fület azon belül pedig lenyitjuk a code-ot. A code opció belül pedig kiválasztjuk az instant reverse-t.



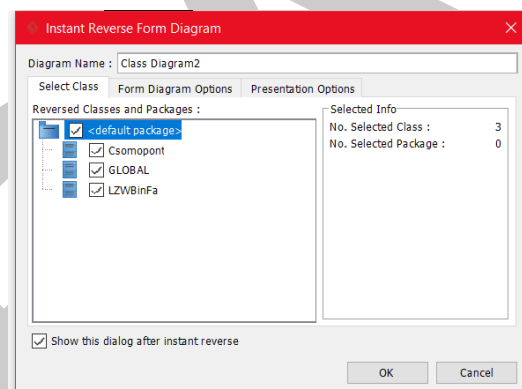
12.1. ábra. 1.

A language-nél megadjuk a c++ forrást, majd megadjuk a kódunk elérési útját és az ok-ra kattintunk, ezzel el is készült az osztálydiagram.



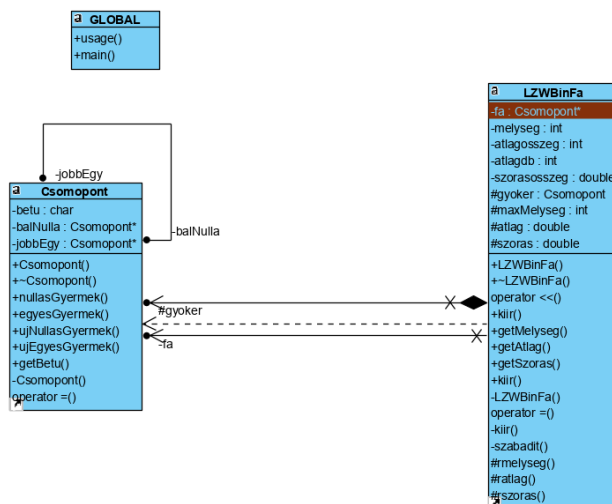
12.2. ábra. 2.

Már csak annyi a teendőnk, hogy a reversed classes and packages ablakban mindent bepipálunk így láthatóvá válik a diagrammunk.



12.3. ábra. 3.

Az importált elemek automatikusan behúzásra kerülnek megfelelő módon.(tehát minden kis vonal oda mutat ahová kell)



12.4. ábra. 4.

12.2. Forward engineering UML osztálydiagram

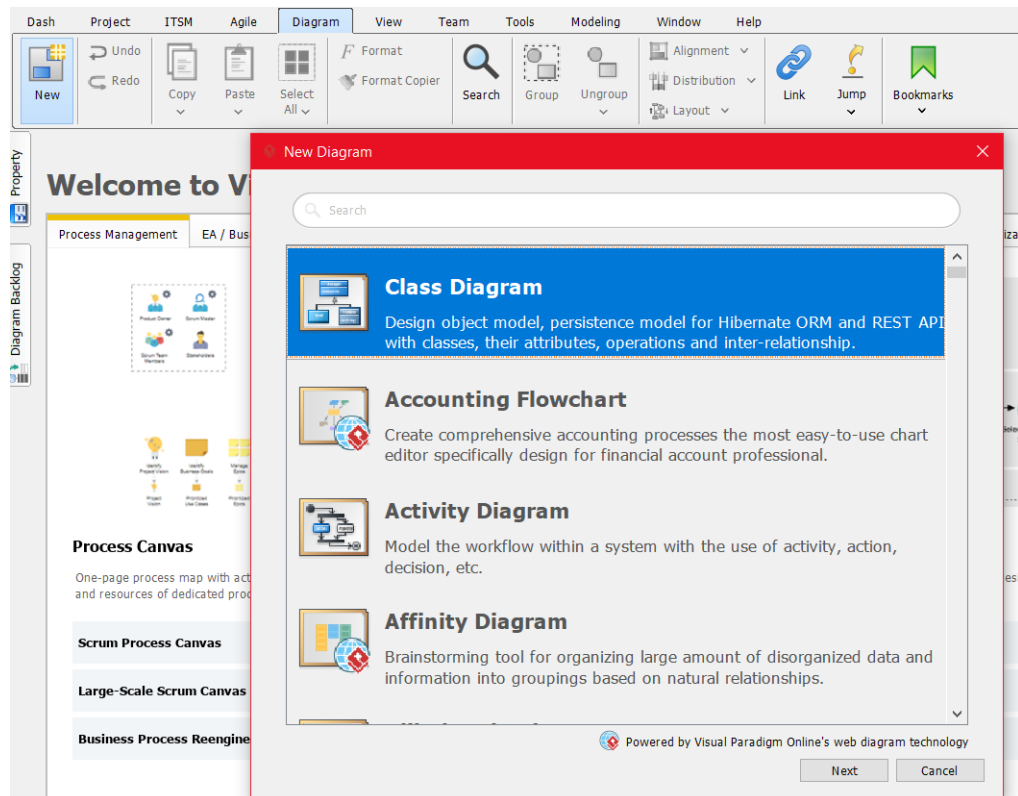
UML-ben tervezzük osztályokat és generáljuk belőle forrást!

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/forward-UML>

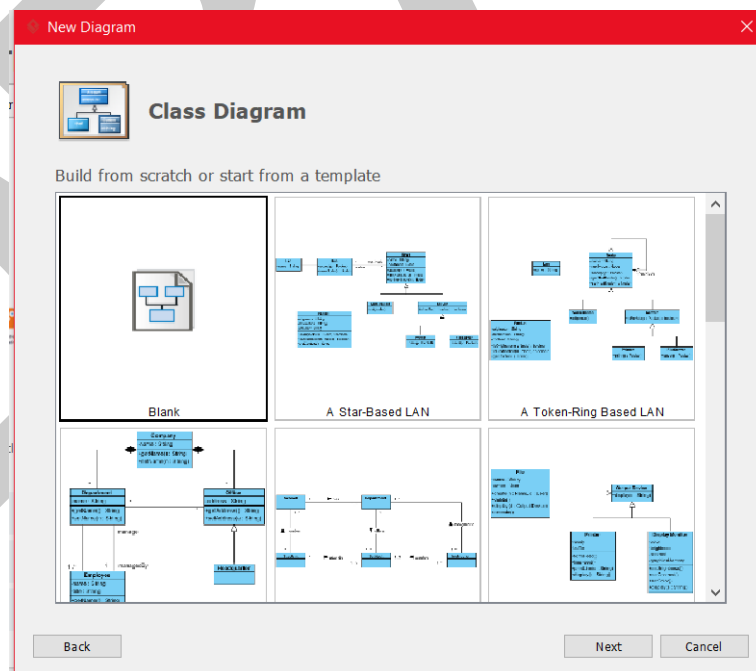
Tanulságok, tapasztalatok, magyarázat...

Ez a feladat ha úgy vesszük akkor az előző fordítottja lenne. Tehát most nekünk kell egy osztálydiagramot készíteni és ebből fogunk majd kódot generálni. Megint a Visual Paradigm-ot használtam a megoldáshoz. Ez a feladat már számomra nehezebb volt mivel még nem csináltam soha osztálydiagramot, de ettől függetlenül összeállítottam egy egyszerűt. Először is a a diagram fülön a new opciót választottam majd kikerestem a class diagramot.



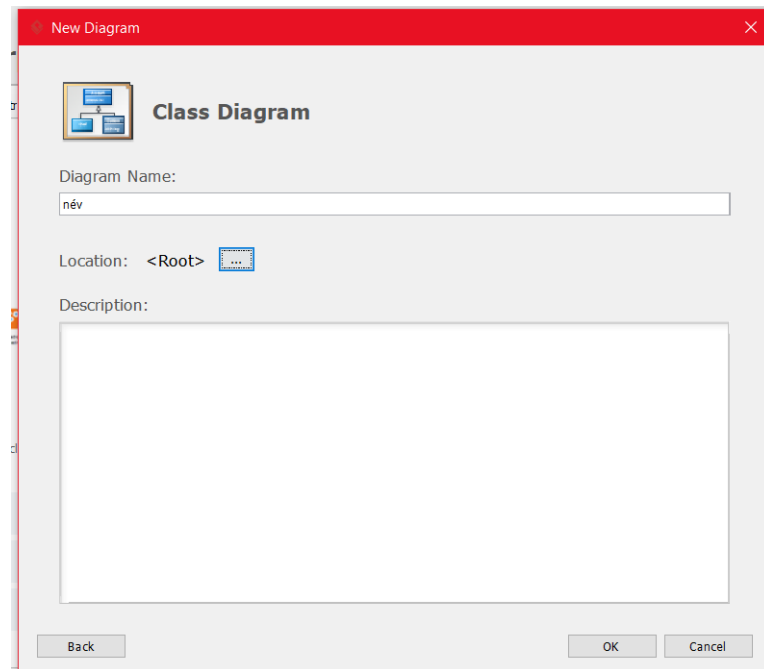
12.5. ábra. 5.

Azon belül pedig a blank opciót



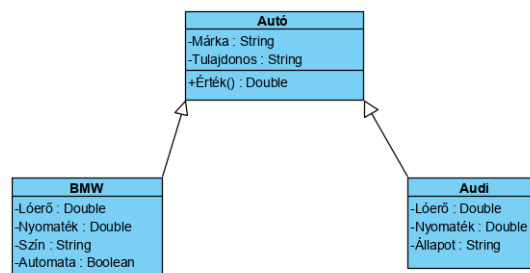
12.6. ábra. 6.

Aztán megadjuk a diagramunk nevét és okéra kattintunk.



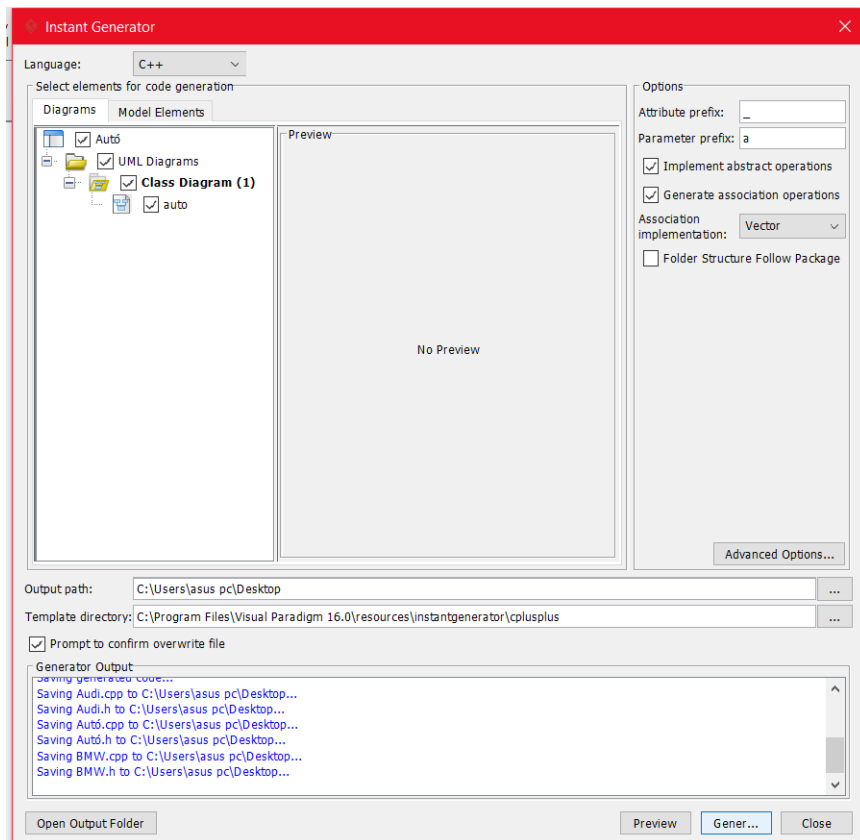
12.7. ábra. 7.

Ezek után kapunk egy üres oldalt ahol kedvünkre hozhatunk létre akármilyen osztálydiagramot. Magunktól kell, hogy létrehozzuk az osztályokat, azok metódusait, változóit, függvényeit, szóval tényleg mindent ami hozzá tartozik. Az enyém így néz ki:



12.8. ábra. 8.

Mostmár pedig már csak annyi dolgunk van, hogy ebből legeneráljuk magát a kódot amit a következőképpen csinálunk: Ugyanúgy mint az előző feladatban kiválasztjuk a tool fülön belüli code opciót de most azon belül az instant generator-re kattintunk. Itt megint bepipáljuk amiot le szeretnénk generálni, az output path opciónál megadjuk neki hova rakja a forrást, generálunk és már készen is vagyunk.



12.9. ábra. 9.

12.3. Egy esettan

A BME-s C++ tankönyv 14. fejezetét (427-444 elmélet, 445-469 az esettan) dolgozzuk fel!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.4. BPMN

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

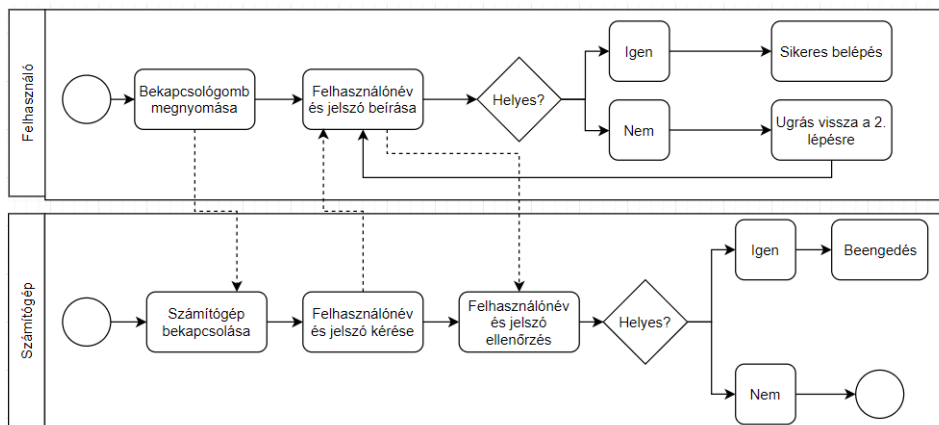
Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A BPMN (Business Process Model and Notation) legfőképpen az üzleti szférában alkalmazott folyamatleíró. Segítségével könnyen és egyszerűen tudunk grafikusán ábrázolni folyamatokat. Ebben a feladatban

ennek a segítségével kellett valami egyszerű tevékenységet ábrázolni. Én úgy döntöttem, hogy ez a számítógép bekapcsolásának és a bejelentkezésnek a folyamata lesz. A draw.io (Google) segítségével oldottam meg a feladatot. Használata elég egyszerű és szinte adja magát, hogy mit hogyan kell csinálni benne. Itt is mindent tudunk ábrázolni szinte amit szeretnénk (pl: event, activity, gateway) Az én verzióm így néz ki:



12.10. ábra. 10.

Véleményem szerint a az ábra értelmezése is meglehetősen egyszerű, csupán ránézésre meg lehet érteni milyen folyamatot ír le és mi történik abban.

12.5. BPEL Helló, Világ! - egy visszhang folyamat

Egy visszhang folyamat megvalósítása az alábbi teljes „videó tutorial” alapján: https://youtu.be/0OnlYWX2v_I

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.6. TeX UML

Valamilyen TeX-es csomag felhasználásával készíts szép diagramokat az OOCWC projektről (pl. use case és class diagramokat).

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13. fejezet

Helló, Chomsky!

13.1. Encoding

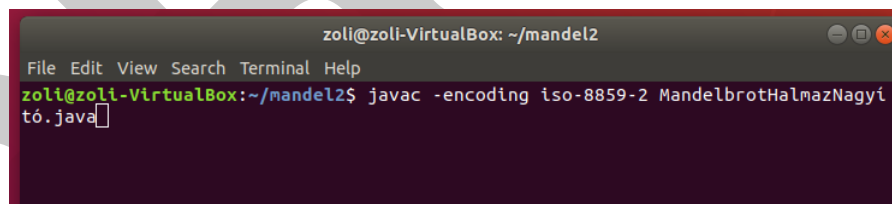
Fordítsuk le és futtassuk a Javat tanítók könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékezetes betűket! <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/adatok.html>

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/mandel.java/blob/master/MandelbrotHalmazNagy%C3%ADt%C3%B3.java>

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban az encoding kapcsolót kell használatba vennünk annak érdekében, hogy a magyar ábécében szereplő ékezetes betűket is használhassuk a kódban illetve annak nevében. Mivel alapjáraton a fordítónk UTF-8-as karakterkódolást használ és ezt kell egy számunkra megfelelőre átállítani. Erre szerintem megfelelő lenne a Latin-2-es kódolás (kódja: iso-8859-2) Így már fordul a program gond nélkül.



```
zoli@zoli-VirtualBox: ~/mandel2
File Edit View Search Terminal Help
zoli@zoli-VirtualBox:~/mandel2$ javac -encoding iso-8859-2 MandelbrotHalmazNagyító.java
```

13.1. ábra. 1.

13.2. OOCWC lexer

Izzítsuk be az OOCWC-t és vázoljuk a <https://github.com/nbatfai/robocaremulator/blob/master/justine/rcemu/src/lexer> és kapcsolását a programunk OO struktúrájába!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.3. l334d1c45

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 tette meg, akkor írasd ki és magyarázd meg a használt struktúrák memóriafoglalását!)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.4. Full screen

Készítsünk egy teljes képernyős Java programot! Tipp: https://www.tankonyvtar.hu/en/tartalom/tkt/javatanitok-javat/ch03.html#labirintus_jatek

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.5. Paszigráfia Rapszódia OpenGL full screen vizualizáció

Megoldás videó:

Megoldás forrása: <https://github.com/kzoltan99/prog2-feladatok/blob/master/para6.cpp>

Tanulságok, tapasztalatok, magyarázat...

13.6. Paszigráfia Rapszódia OpenGL full screen vizualizáció

A program sikeres futtatásához szükséges egy pár könyvtár telepítése ha ezek még nincsenek feltelepítve persze. Ezek a libm, a freeglut és a boost. A sikeres fordításhoz a `-lboost_system -lGL -lGLU -lglut` kapcsolót kell használatba vennünk. A feladat megoldásához Bátfai Tanár Úr `para6.cpp` nevű forrását használtam fel.

A színeken kellett változtatni illetve meg kellett oldani, hogy a program teljes képernyős módban is futtasson. A `glColor3f` eljárás segítségével tudtam változtatni a színek kódokon. 3 értékre van szüksége amely egy rgb színekódot ad ki. $(x * 255)$ Ez fogja a rajzolt elemeket kiszínezni. A vonalak színét a `glBegin (GL_LINES)` meghívása után tudhatjuk, hogy itt a vonalak színezése fog történni.

A feladat másik része pedig az volt, hogy teljes képernyőre kellett rakni a programot. Itt a már létező `KeyPress` eventel kellett kicsit variálni. Pontosabban ki kellett egészíteni még két új blokkal. Ezek `f` és `m` névre hallgatnak ebben az iterációban. A működése egyszerű: Ha `f` betűt nyomunk akkor a program teljesképernyős módra vált át, ha pedig ezután megnyomjuk az `m`-et akkor visszatérünk az eredeti ablakmérethez.

```
} else if (key == 'f') {  
    glutFullScreen();  
} else if (key == 'm') {  
    glutReshapeWindow(640, 480);  
}
```

13.7. Paszigráfia Rapszódia LuaLaTeX vizualizáció

Lásd vis_prel_para.pdf! Apró módosításokat eszközölj benne, pl. színvilág, még erősebb 3D-s hatás.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.8. Perceptron osztály

Megoldás forrása: <https://github.com/kzoltan99/perceptron>

Dolgozzuk be egy külön projektbe a projekt Perceptron osztályát! Lásd <https://youtu.be/XpBnR31BRJY>

A gépi tanulásban nagyon fontos algoritmus a perceptron. Léteznek úgynevezett bináris osztályozók is melyeknek feladata az, hogy eldöntsék, hogy a bemenet egy különleges(specifikus) osztály része-e. Magát a perceptront sokkal jobban nem ecsetelném mivel nagyon összetett dolog és szinte minden információ megtalálható róla pl a Wikipédián is.

Három részből áll a perceptron: van egy olyan rész melyet "retinának" nevezünk, ez az első elem, feladata azon cellák tárolása melyek fogadják az inputot. Aztán maguk a cellák. ezek összegzik a jeleket amik beérkeznek. Végül pedig vannak olyan cellák melyek már a perceptron kimeneti részében vannak, ezek a döntési cellák. A működésük módja hasonló a többi cellához.

Ez a feladat egy az előző félévi perceptronos feladat egyik iterációja. Most ugye azt is meg kell oldani, hogy bemenetként vegyünk egy képet és az lesz a többbrétegű perceptron bemenete.

Mivel többbrétegű perceptront fogunk alkalmazni ezért meg kell tennünk a megfelelő include-okat. Ezek névszerint az mlp és a png kép használata végett a png könyvtárak.

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

Maga a forrás nem túl hosszú. Először is beolvassuk a képet (get_width, get_height) és a new operátorral létrehozuk a perceptront.

```
int main(int argc, char **argv){  
    png::image <png::rgb_pixel> png_image(argv[1]);  
    int size = png_image.get_width()*png_image.get_height();  
    Perceptron *p = new Perceptron(3, size, 256, 1);
```


Aztán létre kell hoznunk egy double változót. Utána szépen végigmegyünk a kép magasság/szélesség pontjain for ciklusokkal. Az image fogja tárolni azt a színkomponenst amit a kódban megadtunk. A value pedig majd azt a double-t fogja tárolni amit a végén kiíratunk.

```
double* image = new double[size];
for(int i = 0; i<png_image.get_width(); ++i)
    for(int j = 0; i<png_image.get_height(); ++j)
        image[i*png_image.get_width()+j] = png_image[i][j].red;
double value = (*p) (image);
cout << value << endl;
```

A végén pedig a delete-el felszabadítjuk a számunkra szükséges helyet a memóriában.

```
delete p;
delete [] image;
```

14. fejezet

Helló, Stroustrup!

14.1. JDK osztályok

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Ezt a feladatot a leírásban ajánlott Bátfa Tanár Úr által készített fénykard.cpp alapján készítettem el.

Azért jó kiindulási kód a fénykard mert ebben is állományok kilistázása valamint rekurzív bejárás megvalósítása történik. Ezt a feladatot itt a forrás 72. sorában található read_acts függvény végzi el, viszont nekünk ezen kell változtatnunk, mivel mi a JDK osztályait akarjuk majd kilistázni.

```
void read_acts(boost::filesystem::path path, std::map <std::string, int> & acts)
{
    if (is_regular_file(path)) {
        std::string ext(".props");
        if (!ext.compare(boost::filesystem::extension(path))) {
            std::string actproppath = path.string();
            std::size_t end = actproppath.find_last_of("/");
            std::string act = actproppath.substr(0, end);
            acts[act] = get_points(path);
            std::cout << std::setw(4) << acts[act] << "    " << act << std::endl;
        }
    } else if (is_directory(path))
        for (boost::filesystem::directory_entry & entry : boost::filesystem::directory_iterator(path))
            read_acts(entry.path(), acts);
}
```

Ahogy itt láthatjuk a kódcsipetben ez a függvény a .props állományokat listázza ki nekünk. Ezt módosítjuk úgy, hogy a .java állományokkal tegye ezt. Ezt úgy tudjuk megvalósítani, hogy egy kicsit fargunk ebből a függvényből, vagyis az act-okat kitöröljük mivel számunkra azok lényegtelenek. Egy egyszerű push_back-el helyettesítjük, és a .props valamint acts előfordulásokat rendre .java és Classes előfordulásra cseréljük.

Így néz ki a számunkra megfelelő verzió:

```
void readClasses(boost::filesystem::path path, vector<string>& classes){
    if (is_regular_file(path)){
        std::string ext(".java");
        if (!ext.compare(boost::filesystem::extension(path))){
            classes.push_back(path.string());
        }
    }
    else if (is_directory(path))
        for (boost::filesystem::directory_entry & entry : boost::filesystem::↔
            ↳ directory_iterator(path))
            readClasses(entry.path(), classes);
}
```

És ezekkel a változtatásokkal sikerült elérni a célunkat vagyis, hogy a programunk rekurzívan bejárja a kapott állományt és abból az összes .java-t kilistázza nekünk.

14.2. Másoló-mozgató szemantika

Kódcsipeteken (copy és move ctor és assign) keresztül vesd össze a C++11 másoló és a mozgató szemantikáját, a mozgató konstruktort alapozd a mozgató értékadásra!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.3. Hibásan implementált RSA törése

Készítsünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló: https://arato.inf.unideb.hu/-batfai.norbert/UDPROG/deprecated/Prog2_3.pdf (71-73 fólia) által készített titkos szövegen.

Az RSA egy viszonylag régen, 1976-ban kifejlesztett titkosító algoritmus, de ettől függetlenül a mai napon is az egyik leggyakrabban használt. Az egész algoritmus nagyon összetett és bonyolult matematikai háttérrel rendelkezik, és véleményem szerint a feladatunk megoldásához most ez nem is olyan lényeges. Működési elve viszont annyira nem nehezen érthető szerintem. Van egy titkos és egy nyilvános kulcsunk, a nyilvánossal tudunk titkosítani, és ez publikus valamint a titkos kulccsal tudjuk feloldani a titkosítást. Nekünk most azt kell valahogy megvalósítani, hogy ne működjön ez az algoritmus, tehát ne lehessen a kódolást visszafejteni. Az RSA algoritmust sajnos még nem volt lehetőségem behatóbban megismerni ezért Szilágyi Csaba barátom segítségét vettem igénybe a megoldáshoz. Szokásos módon a fontos könyvtárak importálásával kezdünk. (pl:filereader és a BigInteger)

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.HashMap;
import java.util.Map.Entry;
```

Aztán itt láthatjuk magát a main függvényt.

```
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("Kodolt:" + encoded);

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

Mainben fog történni a szöveg letitkosítása valamint itt jön létre a kulcs is. A fő osztályunk itt az `rsa_chiper` lesz. Itt kell majd megadni azt a szöveget is amit szeretnénk titkossá tenni, ezt fogja majd megpróbálni visszafejteni a programunk. És ezután látható majd a feladat megvalósításának egyik fontos lépése: a betűről betűre való titkosítás megvalósítása.

```
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();
    }
```

A titkosított szöveget úgy állítja elő a program, hogy a byte-okból emberek számára "fogyaszthatatlan" humbug szöveget csinál.

Az algoritmus a betűk gyakoriságát figyeli és aszerint helyettesíti be a karaktereket ezért kell egy lista amelyben ez az információ szerepel. Érdemes a listát a használt emberi nyelven belüli betűgyakoriság alapján beállítani.

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

Aztán szükségünk van egy olyan függvényre ami ezt a listát beolvassa és ha előfordulást észlel akkor belerakja a listába 1-es kezdőértékkel, ha már benne van akkor pedig növelni ezt a számot egyel. Csak és kizárólag a betűket számolja, tehát mondjuk a space-t nem veszi figyelembe. Maga a vizsgálat azért fontos, hogy megtudjuk melyik betűnek van a legnagyobb értéke bagyis melyik szerepel a legtöbbet, és ez lesz a prioritásunk.

Végül a programunk meghívja a nextFreq függvényt, ez a listából behelyettesíti a karaktereket,(ún. max. kiválasztásos módszer alkalmazásával.) és ezeket ki is veszi a listából így az végül üres lesz. A visszafejtés pontossága nem mindig kielégítő de legtöbb esetben az eredeti szöveghez hasonlót kapunk vissza. A sikeresség nagyban, vagyis szinte teljesen az elkészített listától függ.

```
private char nextFreq() {
    char c = 0;
    int nowFreq = 0;
    for(Entry<Character, Integer> e : this.charRank.entrySet()) {
        if (e.getValue() > nowFreq) {
            nowFreq = e.getValue();
            c = e.getKey();
        }
    }
    if (this.charRank.containsKey(c))
        this.charRank.remove(c);
}
```

```
    return c;
}
```

Legvégül pedig a `getDecode` visszaadja nekünk a szöveget amit a program visszafejtett.

14.1. ábra. RSA

14.4. 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 feladatát is.)

A feladatot az előző csokorban szereplő perceptronos feladat alapján csináltam meg. A forrás nagy része így megegyezik.

Mivel többretegű perceptront fogunk alkalmazni ezért meg kell tennünk a megfelelő `include`-okat. Ezek névszerint az `mlp` és a `png` kép használata végett a `png` könyvtárak.

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

Maga a forrás nem túl hosszú. Először is beolvassuk a képet (`get_width`, `get_height`) és a `new` operátorral létrehozuk a perceptront.

```
int main(int argc, char **argv){
    png::image <png::rgb_pixel> png_image(argv[1]);
    int size = png_image.get_width()*png_image.get_height();
    Perceptron *p = new Perceptron(3, size, 256, 1);
```

Aztán létre kell hoznunk egy `double` változót. Utána szépen végigmegyünk a kép magasság/szélesség pontjain for ciklusokkal. Az `image` fogja tárolni azt a színkomponenst amit a kódban megadtunk. A `value` pedig majd azt a `double`-t fogja tárolni amit a végén kiíratunk.

```
    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);
    cout << value << endl;
```

Ezek után jöhetnek a változtatások az eredeti perceptronos feladathoz képest. Most azt szeretnénk, hogy egy képet generáljon nekünk a program nem pedig értékeket írjon. Ezt végrehajtandó implementálunk két új for ciklust az előzőekkel megfelelő céllal valamint `double*`-ot fogunk használni a `double` helyett. Ezek azért szükségesek, hogy a megfelelő végeredményt kapjuk, vagyis a képünk megfelelő adatokat kapjon a sikeres generáláshoz és a `write png` kiterjesztésű képet alkosson nekünk.

```
double* newPicture = (*p) (image);  
for(int i=0; i<png_image.get_width(); ++i)  
for(int j=0; j<png_image.get_height(); ++j)  
    png_image[i][j].red = newPicture[i*png_image.get_width()+j];  
png_image.write("output.png");
```

A kódunk végén pedig elvégezzük a szükséges hely felszabadítást a memóriában, amit a következő képpen teszünk:

```
delete p;  
delete [] image;
```

Ahhoz, hogy minden probléma nélkül fusson már csak annyit kell tennünk, hogy az mlp.hpp header fájlban is átírjuk a double-t double*-ra, mivel azt szeretnénk, hogy visszaadja.

```
double* operator() ( double image [] )
```

14.5. Összefoglaló

Az előző 4 feladat egyikéről írd egy 1 oldalas bemutató „esszé szöveget”!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15. fejezet

Helló, Gödel!

15.1. Gengszterek

Gengszterek rendezése lambdával a Robotautó Világbajnokságban <https://youtu.be/DL6iQwPx1Yw> (8:05-től)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.2. C++11 Custom Allocator

<https://prezi.com/jvvbytkwgsxj/high-level-programming-languages-2-c11-allocators/> a CustomAlloc-os példa, lásd C forrást az UDPROG repóban!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.3. STL map érték szerinti rendezése

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.4. Alternatív Tabella rendezése

Mutassuk be a https://progater.blog.hu/2011/03/11/alternativ_tabella a programban a java.lang Interface Comparable T szerepét!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.5. Prolog családfa

Ágyazd be a Prolog családfa programot C++ vagy Java programba! Lásd [para_prog_guide.pdf](#)!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.6. GIMP Scheme hack

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16. fejezet

Helló, !

16.1. FUTURE tevékenység editor

Javítsunk valamit a ActivityEditor.java JavaFX programon! <https://github.com/nbatfai/future/tree/master/cs/F6>
Itt láthatjuk működésben az alapot: <https://www.twitch.tv/videos/222879467>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.2. OOCWC Boost ASIO hálózatkezelése

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.3. SamuCam

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.4. BrainB

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.5. OSM térképre rajzolása

Debrecen térképre dobjunk rá cuccokat, ennek mintájára, ahol én az országba helyeztem el a DEAC hekkereket: <https://www.twitch.tv/videos/182262537> (de az OOCWC Java Swinges megjelenítőjéből: <https://github.com/emulator/tree/master/justine/rcwin> is kiindulhatsz, mondjuk az komplexebb, mert ott időfejlődés is van...)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17. fejezet

Helló, Schwarzenegger!

17.1. 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:

Tanulságok, tapasztalatok, magyarázat...

17.2. AOP

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.3. Android Játék

Írjunk egy egyszerű Androidos „játékot”! Építkezzünk például a 2. hét „Helló, Android!” feladatára!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.4. Junit teszt

A https://progater.blog.hu/2011/03/05/labormeres_otthon_avagy_hogyan_dolgozok_fel_egy_pedat poszt kézzel számított mélységét és szórását dolgozd be egy Junit tesztbe (sztenderd védési feladat volt korábban).

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.5. OSCI

Készíts egyszerű C++/OpenGL-es megjelenítőt, amiben egy kocsit irányítasz az úton. A kocsí állapotát minden pillanatban mentsd le. Ezeket add át egy Prolog programnak, ami egyszerű reflex ágensként adjon vezérlést a kocsinak, hasonlítsd össze a kézi és a Prolog-os vezérlést. Módosítsd úgy a programodat, hogy ne csak kézzel lehessen vezérelni a kocsit, hanem a Prolog reflex ágens vezérelje!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18. fejezet

Helló, Calvin!

18.1. MNIST

Az alap feladat megoldása, +saját kézzel rajzolt képet is ismerjen fel, https://progater.blog.hu/2016/11/13/hello_samu_a_mnist
bol Hátterként ezt vetítsük le: <https://prezi.com/0u8ncvvoabcr/no-programming-programming/>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.2. Deep MNIST

Mint az előző, de a mély változattal. Segítő ábra, vedd össze a forráskóddal a <https://arato.inf.unideb.hu/batfai.norbert>
8. fóliáját!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.3. CIFAR-10

Az alap feladat megoldása, +saját fotót is ismerjen fel, https://progater.blog.hu/2016/12/10/hello_samu_a_cifar-10_tf_tutorial_peldabol

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.4. Android telefonra a TF objektum detektálója

Telepítsük fel, próbáljuk ki!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.5. SMNIST for Machines

Készíts saját modellt, vagy használj meglévőt, lásd: <https://arxiv.org/abs/1906.12213>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.6. Minecraft MALMO-s példa

A <https://github.com/Microsoft/malmo> felhasználásával egy ágens példa, lásd pl.: <https://youtu.be/bAPSu3Rndi8>, https://bhaxor.blog.hu/2018/11/29/eddig_csaltunk_de_innentol_mi, <https://bhaxor.blog.hu/2018/10/28/minecraft>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19. fejezet

Helló, Arroway!

19.1. C++ és Java összehasonlítás

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

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Osztályozásnak nevezzük azt a folyamatot, amelynek során a hasonló objektumokat közös csoportokba azaz osztályokba soroljuk. Az objektumorientált programokban közös tervezésre ad lehetőséget, hogy sok objektum hasonló jellemzőkkel rendelkezik. Nagy előnyt jelent az, ha sok hasonló objektumot közös "tervrajz" alapján tudunk elkészíteni. Ezeket az ún. tervdrajzokat hívjuk osztályoknak. Az osztály bizonyos fajta objektumok közös változóit és metódusait írja le. Az osztályok definiálhatnak példányváltozókat, osztályváltozókat és osztálymetódusokat is. Az osztályváltozók az összes objektumpéldány számára megosztott információkat tartalmaznak. Ha osztályváltozókat alkalmazunk akkor feleslegesek lesznek a példányváltozók. Az osztályok legnagyobb előnye az újrafelhasználhatóság. Az objektum változókból és kapcsolódó metódusokból felépített egység. Az objektum tulajdonságait célszerű elrejtetni tehát nem publikusként kezelni és csak a metódusokon keresztül befolyásolni. Az objektumok használatának legnagyobb előnye a modularitás és az információelrejtés. Modularitás: Az objektum forráskódja független marad más objektumok forráskódjától és ennek köszönhetően könnyen tud illeszkedni a rendszer különböző részeihez. Információ elrejtés: Az objektum a publikus interfészen kommunikál a többi objektum felé. Illetve gondoskodik a saját adatairól és csak a metódusain keresztül ad változtatási lehetőséget a külső objektumoknak. A külső objektumoknak igazából nem is kell tudnia arról, hogy az objektum állapota hogyan van reprezentálva, csak a kívánt viselkedést kell kérnie a metódusokon keresztül. A példányosításhoz a new operátort használjuk a Javában. Ez egy új példányt hoz létre az osztályból és foglal helyet a memóriában. Szükségünk van egy konstruktorra is ami a hívást írja elő, ennek a neve adja majd meg, hogy melyik osztályból kell létrehozni az új példányt és inicializálja az új objektumot. A new operátor egy hivatkozást ad vissza a létrehozott objektumra. Gyakran ezt a hivatkozást hozzárendeljük egy változóhoz. Ha a hivatkozás nincs hozzárendelve változóhoz, az objektumot nem lehet majd elérni, miután a new operátort tartalmazó utasítás végrehajtódott. Az ilyen objektumot névtelen objektumnak is szoktuk nevezni. A java fordító egy bájtódnak nevezett formátumra fordítja le a forráskódot amit a jvm önálló interpreterként fog érzékelni. Előnyös

biztonsági szempontból de lassú, ezért minden jó jvm próbálja növelni a sebességet. A kódot fordítás előtt platformfüggő gépi kódra alakítja át. A nyelv szintaxisa c, c++ ból fejlődött ki, ez a szerkezetben jelenik meg de a java el is tér tőlük vagyis a hasonlóság nem egyenlő az azonossággal. C és c++-al szemben nincs alapértelmezett visszatérési érték, mindig meg kell azt adni, változókhöz "="-el lehet értéket rendelni kezdeti értékadás után nincs definiálva az érték. Még egy különbség, hogy név túlterhelés ugyan van a Javaban is viszont operátor túlterhelés nincs benne. Illetve számos c++ kifejezés, utasítás szintaktikailag helyes Javaban is sőt sokszor a jelenetése is megegyezik. A java mint nyelv szűkebb a c++-nál, de az osztálykönyvtárai miatt szélesebb az alkalmazhatósági területe. Támogatja például a GUI programozást, network programozást vagy éppen a perzisztenciát is. C++-ban is lehet ezeket csinálni de van amelyekhez külső könyvtárak segítségét kell igénybe vennünk. Valamint lehet benne forrás szinten hordozható programokat írni de a szerkesztett bináris kód már nem hordozható, mert a lefordított kód tartalmazza a helyi oprendszerre és hardverre vonatkozó feltételezéseket. A Java egyik fő célja és egyben egyik legnagyobb különbsége a c++-hoz képest az, hogy a kód teljes mértékben hordozható. Emiatt a Java szigorúbb előírásokat szab a típusok méretére, belső szerkezetére, a kifejezések kiértékelésére és a kivételek kiváltásának ellenőrzésére. A statikus változók inicializálása is futási időben történik Javaban valamint sokkal kevesebb dolgot bíz az implementációra mint a c vagy a c++. Ezt azért teszi, hogy maga a kód amely hordozható ne függjön annyira a platformtól és az implementációtól. A Java nyelv nagyon odafigyel arra, hogy a kód a lehető legpontosabban forduljon le és működjön. Ezért az ellenőrzés során kitér olyan dolgokra is amire a c++ és a c nem. Ilyen például a lokális változók ellenőrzése, pontosabban annak ellenőrzése, hogy kapnak-e értéket.

19.2. Python

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A python programozási nyelv a C++-tól, c-től és a Javatól eltérő módon arra lett inkább tervezve hogy ne a futási sebességet tegye előtérbe, hanem inkább a programozót segítse azzal, hogy könnyebben olvasható. Maga a Python egy nagyon magas szintű programozási nyelv melyet 1989 és 1991 között alkottak meg. Egy objektumorientált interpreteres nyelv (tehát rögtön futtatható, nincs különbség a forrás és a tárgykód között). Ahogy a könyv címe is sejteti az olvasóval a Python legelterjedtebb felhasználási területe a mobilprogramozás. A nyelv legfőbb jellemzője ami megkülönbözteti az általunk már jobban ismert nyelvektől és újdonság lehet a számunkra, hogy a szintaxisa behúzás alapú. Ez azt jelenti h az állításokat azonos szintű behúzásokkal tudjuk csoportosítani. Nem kell kapcsos zárójeleket és kulcsszavakat mint például a begin és az end használatba vennünk ehhez a feladathoz. Nagyon fontos, hogy az első utasítás a szkriptben nem lehet behúzás illetve ezeket egységesen kell kezelniük. Továbbá az utasítások csak a sor végéig tartanak nem kell ezeket lezárni. Ha mégsem férne egy utasítás egy sorba akkor '/'-el tudjuk ezt folytatni a következő sorba.

IV. rész

Irodalomjegyzék

19.3. Általános

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

19.4. C

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

19.5. C++

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

19.6. Lisp

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

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

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