

Univerzális programozás

Így neveld a programozód!

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

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

KÖZREMÜKÖDTEK

	<i>CÍM :</i> Univerzális programozás		
<i>HOZZÁJÁRULÁS</i>	<i>NÉV</i>	<i>DÁTUM</i>	<i>ALÁÍRÁS</i>
ÍRTA	Balla, Csaba	2020. május 3.	

VERZIÓTÖRTÉNET

VERZIÓ	DÁTUM	LEÍRÁS	NÉV
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	A Brun tételes feladat kidolgozása.	nbatfai
0.1.0	2020-02-22	A feladatok kidolgozásának kidolgozása. A könyv kidolgozásának elkezdése.	Csabatron99

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?	3
II. Tematikus feladatok	4
2. Helló, Turing!	6
2.1. A	6
2.2. Végtelen ciklus	6
2.3. Lefagyott, nem fagyott, akkor most mi van?	8
2.4. Változók értékének felcserélése	9
2.5. Labdapattogás	10
2.6. Szóhossz és a Linus Torvalds féle BogoMIPS	12
2.7. Helló, Google!	12
2.8. A Monty Hall probléma	13
2.9. 100 éves a Brun tételel	14
3. Helló, Chomsky!	18
3.1. Decimálisból unárisba átváltó Turing gép	18
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	20
3.3. Hivatalos nyelv	21
3.4. Saját lexikális elemző	22
3.5. Leetspeak	23
3.6. A források olvasása	25
3.7. Logikus	27
3.8. Deklaráció	27

4. Helló, Caesar!	31
4.1. double ** háromszögmátrix	31
4.2. C EXOR titkosító	34
4.3. Java EXOR titkosító	35
4.4. C EXOR törő	36
4.5. Neurális OR, AND és EXOR kapu	39
4.6. Hiba-visszaterjesztéses perceptron	39
5. Helló, Mandelbrot! Teszt	41
5.1. A Mandelbrot halmaz	41
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztálytal	44
5.3. Biomorfok	47
5.4. A Mandelbrot halmaz CUDA megvalósítása	50
5.5. Mandelbrot nagyító és utazó C++ nyelven	53
5.6. Mandelbrot nagyító és utazó Java nyelven	55
6. Helló, Welch!	59
6.1. Első osztályom	59
6.2. LZW	60
6.3. Fabejárás	61
6.4. Tag a gyökér	63
6.5. Mutató a gyökér	63
6.6. Mozgató szemantika	63
7. Helló, Conway!	66
7.1. Hangasztimulációk	66
7.2. Java életjáték	67
7.3. Qt C++ életjáték	68
7.4. BrainB Benchmark	70
8. Helló, Schwarzenegger!	73
8.1. Szoftmax Py MNIST	73
8.2. Mély MNIST	74
8.3. Minecraft-MALMÖ	75
9. Helló, Chaitin!	76
9.1. Iteratív és rekurzív faktoriális Lisp-ben	76
9.2. Gimp Scheme Script-fu: króm effekt	76
9.3. Gimp Scheme Script-fu: név mandala	76

10. Helló, Gutenberg!	77
10.1. Programozási alapfogalmak	77
10.2. Programozás bevezetés	77
10.3. Programozás	78
10.4. Mobilprogramozás	78
III. Második felvonás	79
11. Helló, Arroway!	81
11.1. A BPP algoritmus Java megvalósítása	81
11.2. Java osztályok a Pi-ben	81
IV. Irodalomjegyzék	82
11.3. Általános	83
11.4. C	83
11.5. C++	83
11.6. Lisp	83

Ábrák jegyzéke

2.1. A B_2 konstans közelítése	17
4.1. A double ** háromszögmátrix a memóriában	34
5.1. A kiadott kép	44
5.2. A program áltak megcsinált kép	47
5.3. Biomorf kép	50
5.4. C++ mandelbrot nagyító és utazó	55
5.5. Java mandelbrot nagyító és utazó	58
6.1. Inorder bejárás	61
6.2. Preorder bejárás	62
6.3. Postorder bejárás	63
7.1. A hangyaszimulációs program UML diagramja	66
7.2. Hangyaszimuláció	67
7.3. Életjáték	68
7.4. Életjáték	70
7.5. BrainB program és kimenete	72

Előszó

Amikor programozónak terveztem állni, ellenezték a környezetben, 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? Kí tudhatná ma.

Minden esetben 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 készítettem az iniciális változatot. Ám tervezem felhasználását az egyetemi programozás oktatásban is. Ahogy szélesedné 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 (hogy lássuk mászt is) példával.

Hogyan nyomjuk?

Rántsd le a <https://gitlab.com/nbatfai/bhax> git repót, vagy méginkább forkolj belőle magadnak egy sajátot a GitLabon, ha már saját könyvon 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 találd az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

Ne cifrázzuk: programok írása. Mik akkor a programok? Mit jelent az írásuk? Számítógép-programozás (vagy egyszerűen programozás) egy vagy több absztrakt algoritmus megvalósítását jelenti egy bizonyos programozási nyelven. A programozásban megtaláljuk a művészet, a tudomány, a matematika és a mérnöki tudomány elemeit. A programok maga az az utasítás vagy utasítás sorozat ami vezérel egy számítógépet. Valamint segíti a felhasználót(Szoftverek) a számítógép netalántán más gépek használatában, vagy elszörakoztatja(Játékok) azt különböző szórakoztató programokkal. Ennek a skálája végtelennek tekinthető, hogy miket lehet létrehozni ebben a szakmában. Az írásuknak van egy úgymond részei:

- 1. A megoldandó probléma meghatározása, felmérése a majdani felhasználók igényei alapján, specifikáció készítése
- 2. Valamely programtervezési módszerrel a programszerkezet megalkotása és a használandó eszközök kiválasztása. (Hardver platform, nyelvek, adatok stb...)
- 3. Forrásprogram elkészítése (kódolás)
- 4. A kész program tesztelése
- 5. Dokumentáció készítése, mely tartalmazza a szoftvertervezés fázisaiban keletkezett adatokat (felhasználói leírás, igény-felmérés, programtervek, algoritmusok, forráskód, tesztelési jegyzőkönyvek stb.), fő célja a szoftver későbbi fejlesztésének elősegítése.

1.2. Milyen doksikat olvassak el?

- Kezd ezzel: [http://esr.fsf.hu/hacker-howto.html!](http://esr.fsf.hu/hacker-howto.html)
- Olvasgasd aztán a kézikönyv lapjait, kezd a **man man** parancs kiadásával. A C programozásban a 3-as szintű lapokat fogod nézegetni, például az első feladat kapcsán ezt a **man 3 sleep** lapot
- C kapcsán a [[KERNIGHANRITCHIE](#)] könyv adott részei.
- C++ kapcsán a [[BMECPP](#)] könyv adott részei.
- Az igazi kockák persze csemegéznek a C nyelvi szabvány [ISO/IEC 9899:2017](#) kódcsipeteiből is.
- Amiből viszont a legeslegjobban lehet tanulni, az a [The GNU C Reference Manual](#), mert gcc specifikus és programozókra van hangolva: szinte csak 1-2 lényegi mondat és apró, lényegi kódcsipetek! Aki pdf-ben jobban szereti olvasni: <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf>
- Az R kódok olvasása kis általános tapasztalat után automatikusan, erőfeszítés nélkül menni fog. A Python nincs ennyire a spektrum magától értetődő végén, ezért ahhoz olvasd el a [[BMECPP](#)] könyv - 20 oldalas gyorstalpaló részét.

1.3. Milyen filmeket nézzek meg?

II. rész

Tematikus feladatok

Bátf41 Haxor Stream

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

2. fejezet

Helló, Turing!

2.1. A

kódok megtekíthetőek githubon. Link: <https://github.com/Csabatron99/BHAX-konyv-b-csaba/tree/master/Source%20Codes>

2.2. Végtelen ciklus

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

Megoldás videó(nbatfai): <https://youtu.be/lvmi6tyz-nI>

Megoldás forrása: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/infty-f.c](https://github.com/bhax/thematic_tutorials/blob/main/bhax_textbook_IgyNeveldaProgramozod/Turing/infty-f.c), [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/infty-w.c](https://github.com/bhax/thematic_tutorials/blob/main/bhax_textbook_IgyNeveldaProgramozod/Turing/infty-w.c).

Számos módon hozhatunk és hozunk létre végtelen ciklusokat. Vannak esetek, amikor ez a célunk, például egy szerverfolyamat fusson folyamatosan és van amikor egy bug, mert ott lesz végtelen ciklus, ahol nem akartunk. Saját péláinkban ilyen amikor a PageRank algoritmus rázza az 1 liter vizet az internetben, de az iteráció csak nem akar konvergálni...

Egy mag 100 százalékban:

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

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

```
#include <stdbool.h>
int
main ()
{
    while(true);
    return 0;
}
```

Azért érdemes a `for (;;)` hagyományos formát használni, mert ez minden C szabvánnyal lefordul, másrészt a többi programozó azonnal látja, hogy az a végtelen ciklus szándékunk szerint végtelen és nem szoftverhiba. Mert ugye, ha a `while`-al trükközünk egy nem triviális 1 vagy `true` feltétellel, akkor ott egy másik, a forrást olvasó programozó nem látja azonnal a szándékunkat.

Egyébként a fordító a `for`-os és `while`-os ciklusból ugyanazt az assembly kódot fordítja:

```
$ gcc -S -o infny-f.S infny-f.c
$ gcc -S -o infny-w.S infny-w.c
$ diff infny-w.S infny-f.S
1c1
<   .file "infny-w.c"
---
>   .file "infny-f.c"
```

Egy mag 0 százalékban:

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

    return 0;
}
```

Minden mag 100 százalékban:

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

A `gcc [filename].c -o [filename] -fopenmp` parancssorral készítve a futtathatót, majd futtatva, közben egy másik terminálban a `top` parancsot kiadva tanulmányozzuk, mennyi CPU-t használunk:

```
top - 20:09:06 up 3:35, 1 user, load average: 5,68, 2,91, 1,38
Tasks: 329 total, 2 running, 256 sleeping, 0 stopped, 1 zombie
%Cpu0 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu1 : 99,7 us, 0,3 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu2 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu3 : 99,7 us, 0,3 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st

      PID USER      PR  NI      VIRT      RES      SHR S  %CPU  %MEM     TIME+ COMMAND
  5850 batfai      20    0    68360      932      836 R 798,3  0,0   8:14.23 infny-f
```

Werkfilm

- <https://youtu.be/lvmi6tyz-nl>
-

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

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

Megoldás videó:

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

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

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

A program futtatása, például akár az előző v.c ilyen pszeudókódjára:

```
T100(t.c.pseudo)
true
```

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

A T100-as programot felhasználva készítsük most el az alábbi T1000-set, amelyben a `Lefagy`-ra épőlő `Lefagy2` már nem tartalmaz feltételezett, csak csak konkrét kódot:

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

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

    main(Input Q)
    {
```

```
Lefagy2 (Q)
}

}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

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

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

Ez a feladat a megállási probléma bemutatására szolgál. A megálláso probléma abból áll, hogy el lehet-e dönteni egy programról adott bemenet esetén, hogy végtelen ciklusba kerül-e. Alan Turin 1963-ban bizonyította be, hogy nem lehetséges olyan általános algoritmust írni, amely minden program-bemenet párról megmondja, hogy végtelen ciklusba kerül-e.

2.4. Változók értékének felcserélése

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

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

Megoldás forrása:

Változócserére több lehetőség is van, 4 féle alap módszer van. Ezek lehetnek:

- Segédváltozóval
- Kivonással / Összeadással
- XOR logikai módszer

Az első példára a segédváltozónak van jó és rossz oldala is. A jó oldala, hogy ez a legkönyebb minden közül logikailag kitalálni és programba illetve pszeudokódba írni. A rossz oldala pedig a segédváltozó miatt van. A plusz változó miatt több helyet kell lefoglalni. C beli leírása a következő:

```
#include <stdio.h>
int main()
{
    int tmp = 0, a = 1, b = 2;
    tmp = a;
    a = b;
    b = tmp;
}
```

A második példához a matematikailag ismert osszeadási és kivonási szabályokból ismeretes elemeket használjuk fel. Az egyik a következő: $a = a + b$ $b = a - b$ $a = a - b$ $b = a - a$ Lássuk végrehajtva. Az első lépésben $a = 20 + 10 = 30$ lesz. Utána $b = 30 - 10 = 20$, vagyis b-be már bekerült a eredeti értéke, a harmadik lépésben pedig $a = 30 - 20 = 10$, puff, helyet cserélt a kettő! A másik ilyen módszer a következő: $a = a - b$ $b = a + b$ $a = b - a$ Ezt is lássuk végrehajtva. Az első lépésben $a = 20 - 10 = 10$ lesz. Utána $b = 10 + 10 = 20$, vagyis b-be már bekerült a eredeti értéke, a harmadik lépésben pedig $a = 20 - 10 = 10$, és így helyet cserélt a kettő! C beli leírása a következő:

```
#include <stdio.h>
int main()
{
    int a = 1, b = 2;
    a = a + b;
    b = a - b;
    a = a - b;
}
```

```
#include <stdio.h>
int main()
{
    int a = 1, b = 2;
    a = a - b;
    b = a + b;
    a = b - a;
}
```

Az utoló módszerről pedig kicsit többet kell beszéljünk mivel az már egy kicsivel bonyolultabb, helytelen működésű és lassab. Először nézzük, hogy mi is az az XOR csere: Annak idején, amikor még mindenki Assembly nyelvű programokat írt, rengeteg apró optimalizálást kézzel végeztek el a programozók. Például egy regiszter nullázása helyett: movl \$0, eax (C-ben: eax = 0), inkább saját magával XOR kapcsolatba hozták azt: xorl eax, eax (eax ^= eax). Mert ez kevesebb bajtot foglalt, nem kellett a 0 számot eltárolni hozzá a memoriában. Akkoriban jöttek rá arra is, hogy két regiszter értékét szintén meg lehet cserélni az xor gépi utasítással. A módszer előnye, hogy nincsen hozzá szükség segédregiszterre. Eddig minden rendben is lenne. A baj csak az, hogy a '70-es években kitalált, az akkori szemléletet tükröző módszer annyira beszivárgott a köztudatba, hogy egyesek manapság is használják, magas szintű programozási nyelveken. A XXI. században, amikor a fordítóprogramok legtöbbje gyorsabb kódot tud generálni, mint amilyen Assembly kódot az emberek írnának. Magával a módszerrel az a baj, hogy sok programozó nem értené mit akarunk ezzel a programrésszel valamint sokkal nehezebb megérteni, ezért nem terjedt el és nem is szokták használni viszont úgy éreztem jólenne bemutatni mivel ha valamilyen oknál fogva talalkoznál vele akkor tudd, hogy mi is ez pontosan és egy érdekességnak is elkönyvelhető. C beli leírása pedig a következő:

```
#include <stdio.h>
int main()
{
    int a = 1, b = 2;
    a ^= b;
    b ^= a;
    a ^= b;
}
```

A tanulságnak levonhatjuk azt, hogy a segédváltozó segítségével a legkönyebb viszont a második módszerrel nem kell külön lefoglalni helyet a segédváltozónak. Az XOR módszer pedig nem ajánlott használni a fenti okokból kifolyólag. Így a második módszert ajánlom használatra változócsere esetén.

2.5. 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 (nbafai): <https://bhaxor.blog.hu/2018/08/28/labdapattogas>

Megoldás forrása:

Az if es megoldás C kódja:

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

int main(void)
{
    WINDOW *window;
    window = initscr();
    int x=0, y=0;
    int xnow=1, ynow=1;
    int mx,my;
    for(;;)
    {
        getmaxyx(window,my,mx);
        mvprintw(y,x, "O");
        refresh();
        usleep(100000);
        clear();
        x=x+xnow;
        y=y+ynow;
        if (x>=mx-1) //Elerte-e a jobb oldalt?
        {
            xnow = xnow*-1;
        }
        if (x<=0) //Elerte-e a bal oldalt?
        {
            xnow = xnow*-1;
        }
        if (y<=0) //Elerte-e a tetejet?
        {
            ynow=ynow*-1;
        }
        if (y>=my-1) //Elerte-e az aljat?
        {
            ynow = ynow*-1;
        }
    }
    return 0;
}
```

Az if nélküli megoldás C kódja:

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

int main(void)
{
    WINDOW *window;
    window = initscr();
    int xj=0, xk=0, yj=0, yk=0;
    int mx,my;
    nodelay(window, true);
    getmaxyx(window,my,mx);
    my=my*2;mx=mx*2;
    for(;;)
    {
        xj = (xj-1)%mx;
```

```

xk = (xk+1)%mx;
yj = (yj-1)%my;
yk = (yk+1)%my;
clear();
mvprintw(abs((yj-(my-yk)) / 2),abs((xj+(mx-xk)) / 2),"O");
refresh();
usleep(100000);
}
return 0;
}

```

A logikai változós megoldás könnyen megoldható viszont enélkül elégé megnehezítette a dolgomat. Végül sikerült megoldanom néhány érték atfésüléssel ahogy az a fenti példában is látható. Részletesebb magyarázatért megtékintheted a videót.

2.6. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás videó(nbatfai): https://youtu.be/9KnMqrkj_kU, <https://youtu.be/KRZlt1ZJ3qk>.

Megoldás forrása(nbatfai): [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/bogomips.c](https://bhax.io/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/bogomips.c)

A megoldás C++ kódja:

```

#include <iostream>
int main()
{
    int num = -1;
    int count = 0;
    unsigned int num_copy = (unsigned int)num;
    while(num_copy >=1)
    {
        count++;
    }
    printf("%d", (count+1));
    return 0;
}

```

Először elég sokáig gondolkoltam, hogy a feladatban pontosan mit is kell csinálni mi is az amit kér. Egy barátom segítségével aki felsőbbéves rájöttem, innestől már elég könnyen ment a feladat megoldása és a meglévő algoritmus átfarása.

2.7. Helló, Google!

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

Megoldás videó:

Megoldás forrása:

A megoldás C kódja:

```

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

void kiir (double tomb[], int db)

```

```

{
    int i;
    for (i=0; i<db; i++)
        printf("%lf\n", tomb[i]);
}
double tavolsag(double pagerank[], double pagerank_temp[], int db)
{
    double tav = 0.0;
    int i;
    for(i=0;i<db;i++)
        tav+=(pagerank[i] - pagerank_temp[i])*(pagerank[i] - pagerank_temp[i]);
    sqrt(tav);
    return tav;
}
int main(void)
{
    double L[4][4] = {
        {0.0, 0.0, 1.0 / 3.0, 0.0},
        {1.0, 1.0 / 2.0, 1.0 / 3.0, 1.0},
        {0.0, 1.0 / 2.0, 0.0, 0.0},
        {0.0, 0.0, 1.0 / 3.0, 0.0}
    };
    double PR[4] = {0.0, 0.0, 0.0, 0.0};
    double PRv[4] = {1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0};
    for (;;)
    {
        for(int i=0;i<4;i++)
        {
            PR[i]=0;
            for (int j=0; j<4; j++)
            {
                PR[i] = PR[i]+(PRv[j]*L[i][j]);
            }
        }
        if ( tavolsag(PR,PRv, 4) < 0.00000001)
            break;
        for(int i=0;i<4;i++)
            PRv[i]=PR[i];
    }
    kiir (PR,4);
    return 0;
}

```

Mivel pagerank ot már sikerült megírjam egyszer C++-ban így az átírás könyedén megoldható volt.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

Megoldás videó(nbatfai): https://bhaxor.blog.hu/2019/01/03/erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall-paradoxon_kapcs

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

```

kis_szam=10000000
kis=sample(1:3,kis_szam,replace=T)
jatekos=sample(1:3,kis_szam,replace=T)
vezeto=vector(length=kis_szam)
for(i in 1:kis_szam)
{

```

```

if(kis[i]==jatekos[i])
{
  mibol=setdiff(c(1,2,3),kis[i])
}
else
{
  mibol=setdiff(c(1,2,3),c(kis[i],jatekos[i]))
}
vezeto[i]=mibol[sample(1:length(mibol),1)]
}
nemvalnyer=which(kis==jatekos)
valt=vector(length=kis_szam)
for(i in 1:kis_szam)
{
  holvaltoztat=setdiff(c(1,2,3),c(vezeto[i],jatekos[i]))
  valt[i]=holvaltoztat[sample(1:length(holvaltoztat),1)]
}
valtnyer=which(kis==valt)
sprintf("Kiserletek szama: %i",kis_szam)
length(nemvalnyer)
length(valtnyer)
length(nemvalnyer)/length(valtnyer)
length(nemvalnyer)+length(valtnyer)

```

Őszintén megvallva életemben nem hallottam a Monty Hall problémáról eddig. Viszont informálódva Bátfai anyagaiból hamar megtudtam érteni a lényegét és összerakni a szimulációt. Igaz magáról az R nyelvről sem hallottam de ahogy az alapjait kezdtem nézni be kell valjam nem nehéz nyelv valamitn nem valami elterjedt a használata.

2.9. 100 éves a Brun téte

Írj R szimulációt a Brun téte demonstálására!

Megoldás videó(nbatfai): <https://youtu.be/xbYhp9G6VqQ>

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

A természetes számok építőelemei a prímszámok. Abban az értelemben, hogy minden természetes szám előállítható prímszámok szorzataként. Például $12=2 \cdot 2 \cdot 3$, vagy például $33=3 \cdot 11$.

Prímszám az a természetes szám, amely csak önmagával és egygyel osztható. Eukleidész görög matematikus már Krisztus előtt tudta, hogy végtelen sok prímszám van, de ma sem tudja senki, hogy végtelen sok ikerprím van-e. Két prím ikerprím, ha különbségük 2.

Két egymást követő páratlan prím között a legkisebb távolság a 2, a legnagyobb távolság viszont bármilyen nagy lehet! Ez utóbbit könnyű bebizonyítani. Legyen n egy tetszőlegesen nagy szám. Akkor szorozzuk össze $n+1$ -ig a számokat, azaz számoljuk ki az $1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n \cdot (n+1)$ szorzatot, aminek a neve $(n+1)$ faktoriális, jele $(n+1)!$.

Majd vizsgáljuk meg az a sorozatot:

$(n+1)!+2, (n+1)!+3, \dots, (n+1)!+n, (n+1)!+(n+1)$ ez n db egymást követő azám, ezekre (a jól ismert bizonyítás szerint) rendre igaz, hogy

- $(n+1)!+2=1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n \cdot (n+1)+2$, azaz $2^{\text{valamennyi}}+2$, 2 többszöröse, így ami osztható kettővel
- $(n+1)!+3=1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n \cdot (n+1)+3$, azaz $3^{\text{valamennyi}}+3$, ami osztható hárommal
- ...
- $(n+1)!+(n-1)=1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n \cdot (n+1)+(n-1)$, azaz $(n-1)^{\text{valamennyi}}+(n-1)$, ami osztható $(n-1)$ -el
- $(n+1)!+n=1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n \cdot (n+1)+n$, azaz $n^{\text{valamennyi}}+n$, ami osztható n-el

- $(n+1)!+(n+1)=1*2*3*\dots*(n-1)*n*(n+1)+(n+1)$, azaz $(n+1)*\text{valamennyi}+(n+1)$, ami osztható $(n+1)$ -el

tehát ebben a sorozatban egy prim nincs, akkor a $(n+1)!+2$ -nél kisebb első prim és a $(n+1)!+ (n+1)$ -nél nagyobb első prim között a távolság legalább n.

Az ikerprímszám sejtés azzal foglalkozik, amikor a prímek közötti távolság 2. Azt mondja, hogy az egymástól 2 távolságra lévő prímek végtelen sokan vannak.

A Brun téTEL azt mondja, hogy az ikerprímszámok reciprokaiból képzett sor összege, azaz $(1/3+1/5)+(1/5+1/7)+(1/11+1/13)+\dots$ véges vagy végtelen sor konvergens, ami azt jelenti, hogy ezek a törtek összeadva egy határt adnak ki pontosan vagy azt át nem lépve növekednek, ami határ számot B_2 Brun konstansnak neveznek. Tehát ez nem dönti el a több ezer éve nyitott kérdést, hogy az ikerprímszámok halmaza végtelen-e? Hiszen ha véges sok van és ezek reciprokait összeadjuk, akkor ugyanúgy nem lépjük át a B_2 Brun konstans értékét, mintha végtelen sok lenne, de ezek már csak olyan csökkenő mértékben járulnának hozzá a végtelen sor összegéhez, hogy így sem lépnék át a Brun konstans értékét.

Ebben a példában egy olyan programot készítetünk, amely közelíteni próbálja a Brun konstans értékét. A repó [bhax/attention_raising/Primek_R/stp.r](#) mevű állománya kiszámolja az ikerprímeket, összegzi a reciprokaikat és vizualizálja a kapott részeredményt.

```
# Copyright (C) 2019 Dr. Norbert Bátfai, nbatfai@gmail.com
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>
library(matlab)
stp <- function(x) {
  primek = primes(x)
  differencia = primek[2:length(primes)] - primek[1:length(primes)-1]
  index = which(differencia==2)
  v1primek = primek[idx]
  v2primek = primek[idx]+2
  v1pv2 = 1/v1primek + 1/v2primek
  return(sum(v1pv2))
}
x=seq(13, 1000000, by=10000)
y=sapply(x, FUN = stp)
plot(x,y,type="b")
```

Soronként értelemezzük ezt a programot:

```
primek = primes(13)
```

Kiszámolja a megadott számig a prímeket.

```
> primek=primes(13)
> primek
[1] 2 3 5 7 11 13
```

```
diff = primek[2:length(primes)]-primek[1:length(primek)-1]

> diff = primek[2:length(primes)]-primek[1:length(primek)-1]
> diferencia
[1] 1 2 2 4 2
```

Az egymást követő prímek különbségét képzi, tehát 3-2, 5-3, 7-5, 11-7, 13-11.

```
index = which(diff==2)

> index = which(diff==2)
> index
[1] 2 3 5
```

Megnézi a diferencia-ban, hogy melyiknél lett kettő az eredmény, mert azok az ikerprím párok, ahol ez igaz. Ez a diferencia-ban lévő 3-2, 5-3, 7-5, 11-7, 13-11 különbségek közül ez a 2., 3. és 5. indexűre teljesül.

```
v1primek = primek[index]
```

Kivette a primes-ból a párok első tagját.

```
v2primek = primek[index]+2
```

A párok második tagját az első tagok kettő hozzáadásával képezzük.

```
rt1plust2 = 1/t1primes+1/t2primes
```

Az 1/v1primek a v1primek 3,5,11 értékéből az alábbi reciprokokat képzi:

```
> 1/v1primek
[1] 0.33333333 0.20000000 0.09090909
```

Az 1/v2primek a v2primek 5,7,13 értékéből az alábbi reciprokokat képzi:

```
> 1/v2primek
[1] 0.20000000 0.14285714 0.07692308
```

Az 1/v1primek + 1/v2primek pedig ezeket a törteket rendre összeadja.

```
> 1/t1primes+1/t2primes
[1] 0.53333333 0.34285714 0.1678322
```

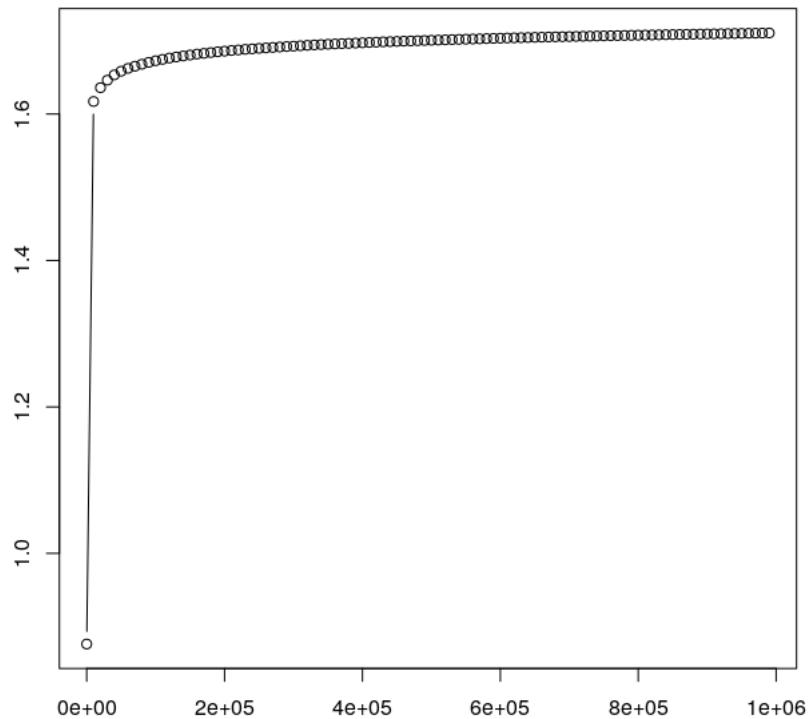
Nincs más dolgunk, mint ezeket a törteket összeadni a sum függvényvel.

```
sum(rt1plust2)

> sum(rt1plust2)
[1] 1.044023
```

A következő ábra azt mutatja, hogy a szumma értéke, hogyan nő, egy határértékhez tart, a B_2 Brun konstanshoz. Ezt ezzel a csipettel rajzoltuk ki, ahol először a fenti számítást 13-ig végezzük, majd 10013, majd 20013-ig, egészen 990013-ig, azaz közel 1 millióig. Vegyük észre, hogy az ábra első köre, a 13 értékhez tartozó 1.044023.

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



2.1. ábra. A B_2 konstans közelítése

Werkfilm

- <https://youtu.be/VkMFrgBhN1g>
 - <https://youtu.be/aF4YK6mBwf4>
-

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

A megoldáshoz tartozik egy kód, amit könnyen lehet vizualizálni és futtatni a következő linken: <https://turingmachine.io/> A kód pedig a következő:

```
%{
#https://turingmachine.io/
input: '12'
blank: ' '
start state: GoRight
table:
  # scan to the rightmost digit
  GoRight:
    [0,1,2,3,4,5,6,7,8,9]: R
    ' ': {L: Decrease}

  Decrease:
    ' ': {R: done}
    0: {L: Oism}
    1: {write: 0, R: ToUniStartP}
    2: {write: 1, R: ToUniStartP}
    3: {write: 2, R: ToUniStartP}
    4: {write: 3, R: ToUniStartP}
    5: {write: 4, R: ToUniStartP}
    6: {write: 5, R: ToUniStartP}
    7: {write: 6, R: ToUniStartP}
    8: {write: 7, R: ToUniStartP}
    9: {write: 8, R: ToUniStartP}

  ToUniStartP:
    [0,1,2,3,4,5,6,7,8,9]: R
    ' ': {R: ToUniEndP}

  ToUniEndP:
    1: R
    ' ': {write: 1, L: ToDecimalStartP}

  ToDecimalStartP:
    1: L
    ' ': {L: Decrease}

  Oism:
```

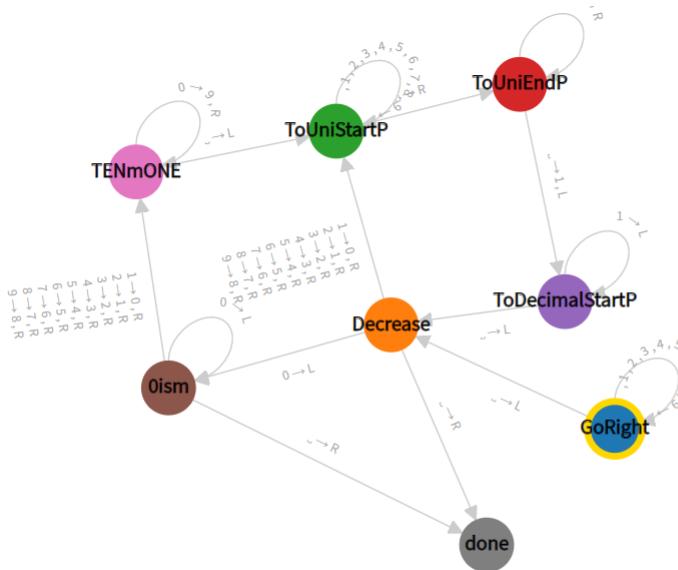
```
0: {L: 0ism}
' ': {R: done}
1: {write: 0, R: TENmONE}
2: {write: 1, R: TENmONE}
3: {write: 2, R: TENmONE}
4: {write: 3, R: TENmONE}
5: {write: 4, R: TENmONE}
6: {write: 5, R: TENmONE}
7: {write: 6, R: TENmONE}
8: {write: 7, R: TENmONE}
9: {write: 8, R: TENmONE}

TENmONE:
0: {write: 9, R: TENmONE}
' ': {L: ToUniStartP}
done:
}
```

A C beni átírása pedig a következő:

```
#include <stdio.h>
int main()
{
    int a, db=0;
    printf("Adj meg egy decimalis szamot!\n");
    scanf("%d", &a);
    printf("A megadott szam unarisba atváltva:\n");
    for (int i = 0; i < a; i++)
    {
        printf("|");
        db++;
        if (db % 5 == 0)
            {
                printf(" ");
            }
    }
    printf("\n");
    return 0;
}
```

A turing gép vizualizációja:



Megoldás videó: (Hamarosan)

Megoldás forrása: (hamarosan)

Ilyen tárgyunk ahol ezzel részletesebben foglalkoztunk volna még nem volt, de a Turing-gép elég egyszerűnek bizonyult ahhoz hogy előzetes tudás nélkül egy online eszköz használatával megírjam a programot.

Megéri belekezdeni egy olyan feladatba is amiről előzetesen semmit sem tudunk, mert lehet könnyen megérthető és egyszerűbb, mint gondolnánk. Előre elégé szkeptikus voltam de kicsi áskálás után már világossá és könnyuvé vállt. A tanulást sok online eszköz is elősegíti. Én ezt az állapotátmenet-gráfot a turingmachine.io segítségével írtam, ami egy egyszerű basic-szerű scriptnyelvből tud turing-gépeket generálni és szép grafikákat is csinálni hozzá.

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

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

A generatív grammatika Noam Chomsky nevéhez kötődik. Úgy vizsgálja a nyelvtant mint az ismeret alapját, hiszen ha nem lenne nyelvtanunk, akkor a tudást nem tudnánk se megörökíteni, se továbbadni. Nézete azt a vállotta, hogy a tudás és az ismeret többnyire öröklött (generációról generációra terjed), vagyis univerzális (gondolván a gyerekekre, akik könnyedén elsajátítják anyanyelüküket). A generatív grammatikának négy fő része van: nemterminális jelek, terminális jelek, helyettesítési/képzési szabályok és mondat/kezdő szimbólumok, illetve három nyelvtan fajtája: környezetfüggő, környezetfüggetlen és reguláris. Nézzünk meg két példát környezetfüggő leírásra (a nyilak jelölik majd a képzési szabályokat)

Megoldás videó:

Megoldás forrása:

```

S, X, Y: „változók” (a nemterminálisok)
a, b, c: „konstansok” (a terminálisok)
S → abc, S → aXbc, Xb → bX, Xc → Ybcc, bY → Yb, aY → aa (a helyettesítési ↔
szabályok)
S (a mondatszimbólum)

S (S → aXbc)
aXbc (Xb → bX)
abXc (Xc → Ybcc)
abYbcc (bY → Yb)
  
```

```
aabbcc

S (S → aXbc)
aXbc (Xb → bX)
abXc (Xc → Ybcc)
abYbcc (bY → Yb)
aYbbcc (aY → aaX)
aaXbbcc (Xb → bX)
aabXbcc (Xb → bX)
aabbXcc (Xc → Ybcc)
aabbYbcc (bY → Yb)
aabYbbccc (bY → Yb)
aaYbbbccc (aY → aa)
aaabbccc
```

A, B, C: „változók” (a nemterminálisok)
a, b, c: „konstansok” (a terminálisok)
A → aAB, A → aC, CB → bCc, cB → Bc, C → bc (a képzési szabályok)
S (A kezdőszimbólum)

```
A (A → aAB)
aAB ( A → aC)
aaCB (CB → bCc)
aabCc (C → bc)
aabbcc

A (A → aAB)
aAB ( A → aAB)
aaABB ( A → aAB)
aaaABBB ( A → aC)
aaaaCBBB (CB → bCc)
aaaabCcBB (cB → Bc)
aaaabCBcB (cB → Bc)
aaaabCBBc (CB → bCc)
aaaabbCcBc (cB → Bc)
aaaabbCBcc (CB → bCc)
aaaabbbCccc (C → bc)
aaaabbbbcccc
```

Tanulságok, tapasztalatok, magyarázat...

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása:

```
<utasítás> ::= <címkézett> | <kifejezés> | <összetett> | <kiválasztó> | <iterációs> | <↔
vezérlésátadó>

<címkézett> ::= azonosító : <utasítás>
                  case <állandó_kif> : <utasítás>
                  default : <utasítás>
```

```
<kifejezés> ::= <kifejezés_kif> ;
<összetett> ::= {déklarációs_lista utasítás_lista}

<déklációs_lista> ::= dékláció
                      déklációs_lista dékláció
<utasítás_lista> ::= utasítás
                      utasítás_lista utasítás

<kiválasztó> ::= if ( <kifejezés_kif> ) <utasítás>
                   if ( <kifejezés_kif> ) <utasítás> else <utasítás>
                   switch ( kifeje
                           zés_kif ) <utasítás>

<iterációs> ::= while ( <kifejezés_kif> ) <utasítás>
                  do <utasítás> while ( <kifejezés_kif> ) ;
                  for (<kifejezés_kif> ; <kifejezés_kif> ; <kifejezés_kif> ) <utasítás>

<vezérlésátadó> ::= goto <azonosító> ;
                      continue ;
                      break ;
                      return <kifejezés_kif> ;
```

Valamint a C kód:

```
#include "stdio.h"

int main()
{
    //c99+
    for (int i = 0; i<5; i++)
    {
        long long int a = 5;
    }

    char* a;
    //c11-
    gets(a);

    return 0;
}
```

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása:https://youtu.be/9KnMqrkj_kU (15:01-től).

A lexer egy olyan program amelyik megír, összeállít nekünk egy C programot. A lex programok úgymond több részből tevődnek össze, ezeket a részeket %%-al vannak elválasztva egymástól. A mi programunk most 3 részből áll. A programot ezen részletek alapján vizsgáljuk meg.

A forráskód megtalálható a következő linken is: [./Chomsky/realnumber.l](#)

```
% {
```

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

A kapcsoszárójelek közötti részt a lexer, úgy ahogy van egy az egyben berakja a C programba. Az első része ismerős a C kódokból, az **include**-dal deklaráljuk az stdio header fájlt. Ezután egy deklarálás és egyben egy értékkedés jön (a **realnumber** változó kezdetben a 0-ás értékkel kapja). A a **realnumber** változóval azt számoljuk majd, hogy hány számot olvas be a program. A kapcsoszárójelezett rész után jönnek a definíciók, a mi esetünkben is van egy: a **digit** nevű definícióval egy szám csoportot adunk meg (0-tól 9-ig tartalmazza a számokat, tehát a számjegyeket tárolja). Szöglletes zárójelek között egy karaktereket tudunk megadni, de mi most mivel számokat írtunk közé, ezért nem karaktereket, hanem számokat fog tárolni. Majd mindenek után, %%-al lezárjuk ezt a részt.

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

Ebben, a második részben, jelennie meg a fordítási szabályok, ahol már használjuk is a definícióknál megadottakat, tehát egy-fajta használati utasítást adunk meg hozzjuk. A **{digit}*{digit}+{realnumbers}** azt jelenti, hogy a digitből lehet nulla vagy bármennyi darab. Az informatikában a **".**" karakter magában azt jelenti, hogy amit szeretnénk azt bármilyen karakterre rá lehet illeszteni. De mivel mi most a lexel azt akarjuk elérni, hogy tokeneket, valós számokat ismerjen fel (tehát olyan formájú számokat, mint például a 3.5, vagyis szám.szám), ezért a **".**-ot le kell védeni, amit a **/** jel segítségével érjük el. A levédett pont után egy újabb kifejezés következik (**{digit}+**), ami azt takarja hogy a pont után is számjegyek jönnek, bármennyi de legalább egy mindenkiépp (+). Ha ez a leírás megegyezik az egyik intup adattal, tehát ha talál egy ilyet, akkor (jön egy utasítássorozat ami {}-ek közé van téve) növeli a **realnumber** értékét, és ezt kiírja a képernyőre először stringként (%s), melyet el is tárolunk az **yytext**-be, majd számként kiiratjuk (%f), az **atof** függvényel az **yytext**-ben lévő stringet átkonvertáljuk double típusú számmá.

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

Az utolsó rész pedig már úgymond maga a tényleges program, ahol meghívjuk az előbb definiált lexikális **yylex** függvényt, elemzést és miután ez véget ér, akkor kiíratom a valós számok számát, vagyis a **realnumber** értékét.

C forráskód létrehozása: **lex -o realnumber.c realnumber.l**

Fordítás: **gcc realnumber.c -o realnumber -lfl**

Futtatás: **./realnumber**

3.5. Leetspeak

Lexelj össze egy l33t cipher!

Megoldás videó: https://youtu.be/06C_PqDpD_k

Megoldás forrása: bhax.t thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/l337d1c7_1

A forráskód megtalálható a következő linken is: [..Chomsky/l33tsp3ak.l](https://Chomsky/l33tsp3ak.l)

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

```
#include <ctype.h>

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

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

{'a', {"4", "4", "@", "/-\\"}},
{'b', {"b", "8", "|3", "|{}"}},
{'c', {"c", "(", "<", "{}"}, {"d", ")", "|)", "|{}"}},
{'e', {"3", "3", "3", "3"}}, {"f", {"f", "|=", "ph", "|#"}}, {"g", {"g", "6", "[", "[+"]}}, {"h", {"h", "4", "|-", "|-", "|-|"}}, {"i", {"1", "1", "|", "!"}}, {"j", {"j", "7", "|-", "|/_"}}, {"k", {"k", "|<", "1<", "|{"}}, {"l", {"l", "1", "|", "|_"}}, {"m", {"m", "44", "(V)", "|\\|/|"}}, {"n", {"n", "|\\|", "|\\|/", "/V"}}, {"o", {"0", "0", "()", "[]"}}, {"p", {"p", "/o", "|D", "|o"}}, {"q", {"q", "9", "O_", "(,)"}}, {"r", {"r", "12", "12", "|2"}}, {"s", {"s", "5", "$", "$"}}, {"t", {"t", "7", "7", "'|'"}}, {"u", {"u", "|_|", "(_)", "[_]"}}, {"v", {"v", "\\\/", "\\\/", "\\\/"}}}, {"w", {"w", "VV", "\\\/\\\/", "(/\\)"}}, {"x", {"x", "%", ")("}}, {"y", {"y", "", "", ""}}, {"z", {"z", "2", "7", ">_"}}, {"0, {"D", "0", "D", "0"}}, {"1, {"I", "I", "L", "L"}}, {"2, {"Z", "Z", "Z", "e"}}, {"3, {"E", "E", "E", "E"}}, {"4, {"h", "h", "A", "A"}}, {"5, {"S", "S", "S", "S"}}, {"6, {"b", "b", "G", "G"}}, {"7, {"T", "T", "j", "j"}}, {"8, {"X", "X", "X", "X"}}, {"9, {"g", "g", "j", "j"}}

// https://simple.wikipedia.org/wiki/Leet
};

%}
%%
```

Pont úgy, mint az előző lexikális elemző feladatban, a program szerkezete, felépítése úgyanaz (a három fő rész). Először is megadjuk a header fájloka, majd szintén következnek a definíciók. A definícióknál deklarálunk egy `L337SIZE`-ot, ami segítségével meg tudjuk majd határozni az input hosszát. Egy struktúra segítségével létrehozzuk a cipher-t ami megkapja a karaktert (amit majd átfű), illetve a négy lehetőséget, stringet (amire majd cseréli). Majd ebből a struktúrából létrehozunk egy `l337d1c7` nevű tömböt (nem adjuk meg az elemeit, azt majd a program megszámol magának), aminek be is adagoljuk az elemeit a struktúra felépítettsége szerint (először a latin betű, majd pedig a négy lehetőség amire cserélheti a program). Ezek vége is az program első részének.

```

. {
    int found = 0;
    for(int i=0; i<L337SIZE; ++i)
    {

        if(l337d1c7[i].c == tolower(*yytext))
        {

            int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0));

            if(r<91)
                printf("%s", l337d1c7[i].leet[0]);
            else if(r<95)
                printf("%s", l337d1c7[i].leet[1]);
            else if(r<98)
                printf("%s", l337d1c7[i].leet[2]);
            else
                printf("%s", l337d1c7[i].leet[3]);

            found = 1;
            break;
        }
    }

    if(!found)
        printf("%c", *yytext);
}
%%

```

A második részben egy for ciklussal végig megyünk a kapott inputon. Az i-edig (tehár majd rendre mindegyik) latin karaktert átalakítja kisbetűvé, megkeresi az adott listában, random számot generál neki (körülbelül 0-100 között), majd az if résznél lévő vizsgálatok alapján (a random számot vizsgálva) dönti el hogy a latin betűnek melyik megfelelőjét írja ki a négy közül. Ha a random szám kisebb mint 91, akkor az első karaktert választja (az elsőnek a legnagyobb a valószínűsége), ha 91 és 94 közötti, akkor a második lehetőséget, ha 94 és 97 közötti, akkor harmadik lehetőséget, ha pedig 97-től nagyobb, akkor az utolsó karaktert választja és írja ki.

```

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

```

A program harmadik és egyben utolsó része egy c programrészlet, ahol a `yylex()` függvényhívással elindítjuk az input átváltoztatását.

C forráskód létrehozása: **lex -o l33tsp3ak.c l33tsp3ak.l**

Fordítás: **gcc l33tsp3ak.c -o l33tsp3ak -lfl**

Futtatás: **./l33tsp3ak**

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.

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

Először i-t 0-val tessük egyenlővé. Ezután megnézzük, hogy i kisebb-e mint 5. Ha igen, akkor lefuttatjuk a body-t majd inkrementáljuk i-t. Ha nem, akkor tovább haladunk a következő utasításhoz

ii.

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

Először i-t 0-val tessük egyenlővé. Ezután megnézzük, hogy i kisebb-e mint 5. Ha igen, akkor lefuttatjuk a body-t majd inkrementáljuk i-t. Ha nem, akkor tovább haladunk a következő utasításhoz

A post-vagy pre-inkrementálás különbsége itt teljesen mindegy, mivel nem használjuk a visszadott értéket és a tévhittel ellenben ugyanannyi instrukciót generál (ugyanolyan "gyors") mindenkorban.

iii.

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

Először i-t 0-val tessük egyenlővé. Ezután megnézzük, hogy i kisebb-e mint 5. Ha igen, akkor lefuttatjuk a body-t majd a tomb[i]-t egyenlővé tessük i-vel, ezután inkrementáljuk i-t. Ha nem, akkor tovább haladunk a következő utasításhoz

iv.

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

s-ből d-be n elemet másol.

v.

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

Unspecified behaviour. (Mert a függvényargumentumok kiértékelési sorrendje nincs meghatározva.) Természetes nyelven: Mindhárom függvényhívás kiértékelődik, de nem tudjuk milyen sorrendben értékelődnek ki az argumentumok.

vi.

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

Kiírja az f(a)-t integerként és az a-t integerként. Itt azért nincs baj mert a-t másoljuk mikor meghívjuk f-et, tehát a eredeti értéke nem változik.

vii.

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

Ha a-t módosítjuk f-ben akkor mivel nincs meghatározva a kiértékelési sorrend, ezért az a lehet a módosítatlan a is vagy a módosított is.

Megoldás forrása:

Megoldás videó:

Egy kis gyakorlás C ből. Valamint, hogy legyen fogalmunk mi mit csinál és hogyan. Megtudtam milyen jó dolgunk van manapság a c++ auto-val, ami kitalálja helyettünk a típust, valamint a template-ekkel amik segítségével nem is kell foglalkoznunk a funkciók pontos definíciójával.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

Minden valós szám estén létezik egy másik olyan valós szám, ami nagyobb tőle és prím (tehát minden számtól van nagyobb prím).

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

Minden valós szám esetén létezik egy másik olyan valós szám ami nagyobb tőle, prím és a tőle kettővel nagyobb valós szám is prímszám (tehát véges sok ikerprím van).

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

Létezik olyan y valós szám minden x valós szám esetén, hogy ha az x prímszám, akkor az x kisebb mint az y (tehát véges sok prímszám van).

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

Létezik olyan y minden x esetén, hogy ha y kisebb mint az x, akkor az x nem prímszám (tehát ugyanaz mint az előző esetben, vagyis hogy véges sok prímszám van).

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

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

First things first: Ragd be LAETX-be

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
int a
- egészre mutató mutató
std::add_pointer_t<std::add_pointer_t<decltype(3)>> a
- egész referenciája
std::add_lvalue_reference_t<decltype(4)> a
- egészek tömbje
int a[5];
- egészek tömbjének referenciája (nem az első elemé)
int (&a)[64];

- egészre mutató mutatók tömbje
int *a[64];
- egészre mutató mutatót visszaadó függvény
int* f();
- egészre mutató mutatót visszaadó függvényre mutató mutató
int* (*fn)();
- egészet visszaadó és két egészet kapó függvényre mutatót visszaadó, egészet kapó függvény
int (*fn(int))(int, int)
- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutatót visszaadó, egészet kapó függvényre
int (*(*fn)(int))(int, int)

Mit vezetnek be a programba a következő nevek?

- `int a;`
egész
- `int *b = &a;`
egész pointer
- `int &r = a;`
egész referencia
- `int c[5];`
egész tömb
- `int (&tr)[5] = c;`
egész tömb referencia
- `int *d[5];`
egész pointerek tömbje
- `int *h();`
funkció ami egész pointert ad vissza
- `int *(*l)();`
funkcióra mutató pointer ami egész pointert ad vissza
- `int (*v(int c))(int a, int b)`
egészet kérő funkció ami egy pointert ad vissza egy funkcióra ami két egészet kér és egy egészet ad vissza
- `int (*(*z)(int))(int, int);`
egészet kérő funkcióra mutató pointer ami egy pointert ad vissza egy funkcióra ami két egészet kér és egy egészet ad vissza

Megoldás videó:

Megoldás forrása:

Az utolsó két deklarációs példa demonstrálására két olyan kódot írtunk, amelyek összehasonlítása azt mutatja meg, hogy miért érdemes a **typedef** használata: [Chomsky/fptr.c](#), [Chomsky/fptr2.c](#).

```
#include <stdio.h>

int
sum (int a, int b)
{
    return a + b;
}

int
mul (int a, int b)
{
    return a * b;
}

int (*sumormul (int c)) (int a, int b)
{
    if (c)
        return mul;
    else
        return sum;
}

int
main ()
{
    int (*f) (int, int);

    f = sum;

    printf ("%d\n", f (2, 3));

    int (*g) (int) (int, int);
    g = sumormul;

    f = *g (42);

    printf ("%d\n", f (2, 3));

    return 0;
}
```

```
#include <stdio.h>

typedef int (*F) (int, int);
typedef int (*(*G) (int)) (int, int);

int
sum (int a, int b)
{
    return a + b;
}

int
```

```
mul (int a, int b)
{
    return a * b;
}

F sumormul (int c)
{
    if (c)
        return mul;
    else
        return sum;
}

int
main ()
{
    F f = sum;

    printf ("%d\n", f (2, 3));

    G g = sumormul;

    f = *g (42);

    printf ("%d\n", f (2, 3));

    return 0;
}
```

Nagyon nehéz a functio-pontier-functio.... es-atöbbi deklarálása amikor íylen sokszor kell alkalmazni. Az a jó, hogy sokszor viszont nem kell alkalmazni ezt a módszert mert vagy túlbonyolítaná a dolgokat, vagy mert teljesen felesleges. Olvasni meg megérteni pláne a legnehezebb ezért ez a módszer kerülendő.

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

Írj egy olyan `malloc` és `free` párost használó C programot, amely helyet foglal egy alsó háromszög mátrixnak a szabad tárban!

Megoldás videó: <https://youtu.be/1MRTuKwRsB0>, <https://youtu.be/RKbX5-EWpzA>.

Megoldás forrása: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/tm.c](https://bhax.thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/tm.c)

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

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

    if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
    {
        return -1;
    }

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

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

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

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

```

*(tm[3] + 2) = 44.0;
*(*(tm + 3) + 3) = 45.0;

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

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

free (tm);

return 0;
}

```

```

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

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

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

    if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
    {
        return -1;
    }
    printf ("%p\n", tm);
}

```

A program első soraiban létrehozunk egy `nr` változót, melynek értéket is adunk: 5. Ez az ötös szám arra utal, hogy a program futtatását követően kiírt alsó háromszögmátrix (az előző sorban eg elem lesz, a másodikban kettő, stb.) 5 soros lesz. A `double **tm` véghajtódásakor deklaráljuk a `tm` nevű változót, a *-al jelezvén hogy ez egy pointer, egy mutató, és a program le is foglal neki monjuk 8 bájtnyi tárhelyet. Ezek után ki is íratjuk a `tm` memóriacímét. A következőkben szerepel a `malloc` függvény, amelyik (mint a **man 3 malloc** paracs lekérése után is kiderül), helyet foglal a memóriában és egy `void *` pointert ad vissza, ami bármire mutathat, majd mi megadunk meg egy típust, hogy arra mutasson amire mi akarjuk. A `malloc` paraméterül megkapja hogy mekkora területet kell lefoglaljon, most egy `sizeof` paramétert kap, ami a `double *` típus helyigényét adja vissza (tehát `nr * sizeof (double *) = 5*8 = 40` bájtot kell lefoglaljon a `tm` számára). Az if-el attól ellenőrizzük, hogy a `malloc` sikeresen lefoglalta-e a helyet a memóriában és hogy sikeresen vissza adta-e a `void *` mutatót. Ha ez nem sikerült, akkor egyenlő a `NULL`-al, vagyis nem mutat sehol semmilyen területet, amit a `malloc` által visszaadott címét. Ezért nem mutat sehol semmilyen területet, amit a `malloc` által visszaadott címét.

```

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

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

```

A következőkben egy for ciklussal úgymond végigmegyünk az 5 soron és minden egyik sorban úrja lesz egy memória foglalás. Nézzük példának hogyan mit fog csinálni abban az esetben amikor az `i=3`: mint mondottam a `malloc` egy `void *` pointert ad

vissza, de mi kikötjük hogy ez nekünk most double * legyen. A malloc a tm[2]-nek most (2+1 * 8) bájtot foglal, vagyis a harmadik sorban három 8 bájtnyi helyet foglal le, a harmadik double * egy olyan sorra mutasson ahol három doublenek van lefoglalva hely (az ábrán ezt az utolsó sor urolsó három cellára jelöli). Ez az egész egy if feltételeként szerepel, melyben ismét ellenőrzi hogy sikeres volt-e a pointer létrehozása és a hely lefogalása, ha nem akkor egyenlő a NULL-lal és megáll a program. De tegyük fel újra, hogy sikkerrel jártunk, ezért kiíratjuk a tm[0], az első sor memóriacímét, majd megyünk tovább.

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

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

Egy következő for ciklussal megszerkesztjük az alsó háromszögmátrixot (ehhez két for ciklus szükséges, mivel így tudjuk meg-határozni az elem sorát és oszlopát), azáltal, hogy az egyes, sor- és oszlopindexekkel jelölt elemek értéket kapnak (az i nr-ig, a j pedig i-ig megy, így biztosítva az alsó háromszögmátrix alakot). Lássuk, hogy hogy működik az értékkedás, amikor az i=1 és j=0: ekkor a második sor első elemének adunk egy (1*2/2+0=) 1-es (double) értéket. Az értékkedások végeztével, újabb for ciklusok segítségével kiíratjuk a kapott alsó háromszögmátrixot.

```
tm[3][0] = 42.0;
(*tm + 3)[1] = 43.0; // mi van, ha itt hiányzik a külső ()
*(tm[3] + 2) = 44.0;
*(*(tm + 3) + 3) = 45.0;

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

Ebben a részben az első négy sor különböző leírások arra, hogy a negyedik sorban lévő négy értéket hogy változtassuk meg, tehát példa arra, hogy hogyan lehet hivatkozni a különböző elemekre ahhoz hogy valamit tudjunk velük csinálni (ezesetben az értékmódosítás művelet végrehajtását). Az értékek megváltoztatása után újra kiíratjuk az alsó háromszögmátrixunkat.

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

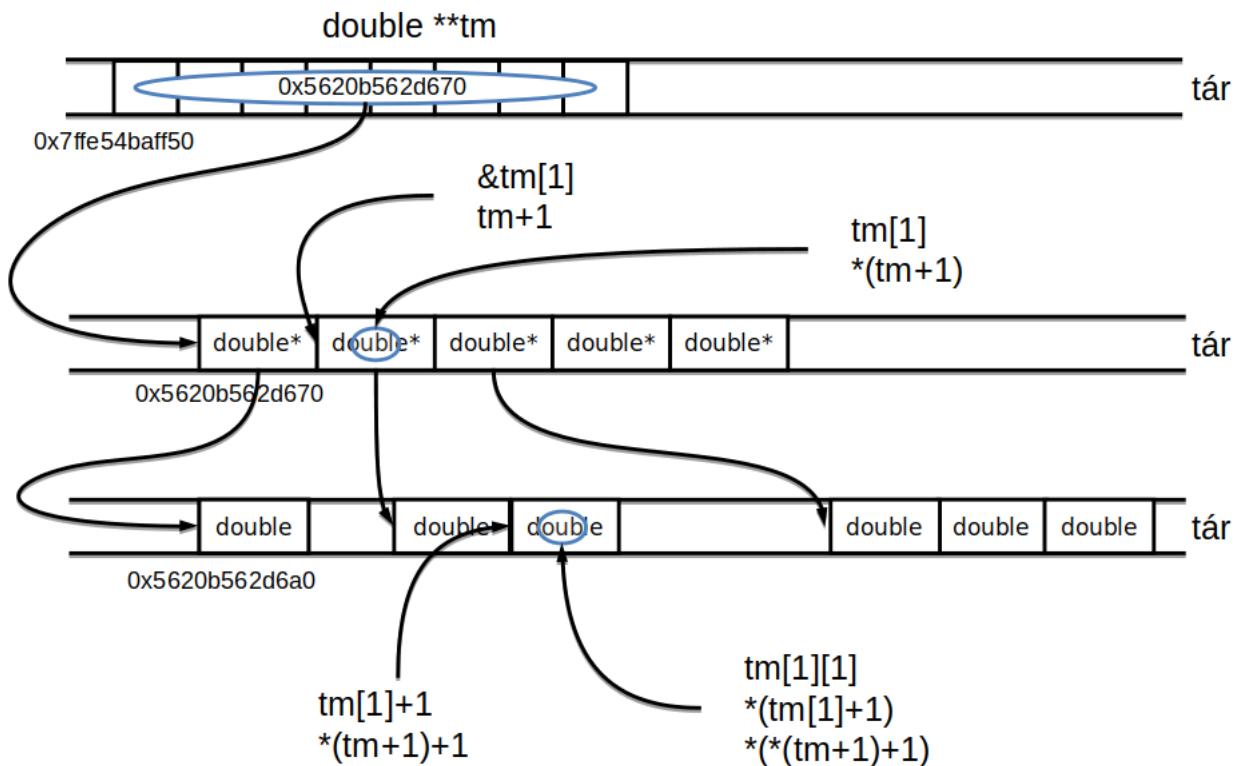
free (tm);

return 0;
}
```

A program utolsó részében a free függvény segítségével felszabadítjuk az egyes sorokban illetve az egész tm által foglalt memóriát.

Fordítás: **gcc tm.c -o tm**

Futtatás: **./tm**

4.1. ábra. A `double **` háromszögmátrix a memóriában

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása:

```

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

#define MAX_KULCS 100
#define BUFFER_MERET 256

int
main (int argc, char **argv)
{
    char kulcs[MAX_KULCS];
    char buffer[BUFFER_MERET];

    int kulcs_index = 0;
    int olvasott_bajtok = 0;

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

    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);  
  
}  
}
```

Egy **tiszta.szöveg** nevű fájl létrehozása, egy szöveg beletétele.

Fordítás: **gcc e.c -o e -std=c99**

Futtatás: **./e kulcs <tiszta.szöveg >titkos.szöveg** (a kulcs bármi lehet)

Dekódolás: **./e kulcs <titkos.szöveg** (a kulcs meg kell egyezzen a fent használt kulccsal)

Az exor titkosító, mint ahogy a neve is mondja, exor művelet segítségével szöveget titkosít a számunkra. A bevitt, titkosítandó szöveg emberi fogyasztásra alkalmas, de a titkosított viszont már annál kevésbé. A dekódolásnál lévő parancs segítségével képesek vagyunk a titkos szövegből visszaadni az eredeti szöveget. A program elején deklarálunk egy kulcsot, egy buffert (ami az olvasott bajtok egyes bajtjait veszi fel), majd bekérjük az olvasott bajtokat (a mi esetünkben a tiszta.szöveg nevű fájlból). A program egyessével végigmegy a beolvasott bajtokon, majd azokat össze xorozza az adott kulcs indexel ami a kulcsban következik, majd a kulcson belül is továbblép a következő karakterre (ha a kulcs 4 karakter akkor sorban minden azon a 4 karakteren megy végig). Ezért játszik fontos szerepet a kulcs, mivel anélkül nem tudjuk feltörni a titkosított szöveget.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

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

A következő java program egy titkosító és egyben egy törő program is, mivel ha újra végrehajtjuk a titkosítást akkor visszakapjuk az eredeti szöveget. Ezt az műveletet az exortitkosito osztályban végezzük el.

```
public class exortitkosito {  
  
    public exortitkosito(String kulcsSzöveg,  
                         java.io.InputStream bejövőCsatorna,  
                         java.io.OutputStream kimenőCsatorna)  
        throws java.io.IOException {  
  
        byte [] kulcs = kulcsSzöveg.getBytes();  
        byte [] buffer = new byte[256];  
        int kulcsIndex = 0;  
        int olvasottBajtok = 0;  
  
        while((olvasottBajtok =  
               bejövőCsatorna.read(buffer)) != -1) {  
  
            for(int i=0; i<olvasottBajtok; ++i) {  
  
                buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]);  
                kulcsIndex = (kulcsIndex+1) % kulcs.length;  
            }  
        }  
    }  
}
```

```
        }

        kimenőCsatorna.write(buffer, 0, olvasottBájtok);

    }

}

public static void main(String[] args) {

    try {

        new exortitkosito(args[0], System.in, System.out);

    } catch (java.io.IOException e) {

        e.printStackTrace();

    }

}

}
```

Mint mindegyik titkosító/törő programban van egy ciklus melyik apránként olvassa a bemenetet, ebben az esetben egy while ciklus, mely tömbönként olvas be. A titkosítás folyamata teljesen az mint az előző C exor titkosító programban. A kulcsIndex-et ráállítjuk a kulcs adott karakterére ami következik, majd ezen és a buffer által beolvasott karakteren elvégezzük a kizárá vagy műveletet. Az eredmény a buffer tömbbe kerül, a program végén ezt íratjuk majd ki.

A program futtatása (titkosítandó szöveg bekérése, titkosítása és visszaváltoztatása):

javac exortitkosito.java (létrejön az exortitkosito.class fájl)
java exortitkosito alma > titkositott.txt (a titkosítandó szöveg bevitelére)
more titkositott.txt (kiíratódik a titkosított szöveg)
java exortitkosito alma < titkositott.txt (dekódoljuk a titkosított szöveget, visszakapva az eredetit)

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása:

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

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

double
atlagos_szohossz (const char *titkos, int titkos_meret)
{
    int sz = 0;
    for (int i = 0; i < titkos_meret; ++i)
        if (titkos[i] == ' ')
```

```

    ++sz;

    return (double) titkos_meret / sz;
}

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

    double szohossz = atlag_szohossz (titkos, titkos_meret);

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

```

A program elején lesz három nagyon fontos és beszédes definícióink: A titkos szöveg maximum mérete, a beolvasott buffer mérete és a kulcs mérete, ami, az előző programmal szemben, itt már nem lehet akármi, hanem ezt a program végén fogjuk ezt iterálni, most annyit tudunk hogy 8 hosszúságú. A `atlag_szohossz` függvénynek is beszédes neve van, ezt a függvényt a következő függvényben már használjuk is ami nemmás mint a `tiszta_lehet`. Ez a függvény úgymond megtippeli, hogy a szöveg amit kapott az megfelel-e a magyar nyelv néhány iratlan szabályának, mint például az hogy tartalmazza-e a leggyakoribb magyar szavakat (hogy, nem, az, ha), mivel ritka az az értelmes magyar nyelven íródott szöveg amikben ezek egyike legalább nem fordul elő.

```

void
exor (const char kulcs[], int kulcs_meret, char titkos[], int titkos_meret)
{

    int kulcs_index = 0;

    for (int i = 0; i < titkos_meret; ++i)
    {

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

```

A követező függvény az ami a tiszta szövegből titkosat csinál az `exor` művelet segítségével, ugyanazzal a módszerrel mint az `Exor` titkosító programban. A program egyessével végigmegy a beolvasott bájtokon, majd azokat össze xorozza az adott kulcs indexel ami a kulcsban következik, majd a kulcson belül is továbblép a következő karakterre.

```

int
exor_tores (const char kulcs[], int kulcs_meret, char titkos[],
            int titkos_meret)
{

    exor (kulcs, kulcs_meret, titkos, titkos_meret);

    return tiszta_lehet (titkos, titkos_meret);
}

```

Nagy meglepődésekre a törő függvény nem rendelkezik külön erre a célra kifejlesztett algoritmussal, hanem felhasználja az előbb bemutatott `exor` függvényt, mivel ha valamit exorozunk egy adott kulccsal, majd ugyanazzal a kulccsal újra megesináljuk

az exor műveletet akkor visszakapjuk az eredeti szöveget. Az exor eme tulajdonságát kihasználva az `exor_toro` elvégzi a titkos szöveg visszaváltoztatását

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

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

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

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

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

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

    return 0;
}
```

Következik a main, a főprogram ahol az előbbieket felhasználjuk. Beolvassuk a titkos szöveget, és itt látszik egy nagy különbség az előző C exor titkosító programmal, ez nem foglal le külön területet a tiszta illetve a titkos szövegeknek, tehát a kódolt és a dekódolt szövegeknek. A titkos szöveg bekérése után, megnézi hogy a kisebb e a mérete mint amennyit az elején lefoglalt neki a `MAX_TITKOS`-ba, ha igen akkor a maradék helyet kinullázza. Ezután előállítja a kulcsokat majd jöhét az exorozás, tehát a törés és a titkosítás művelete.

Fordítás: `gcc t.c -o t`

Futtatás: `./t`

4.5. Neurális OR, AND és EXOR kapu

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

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

A neurális hálók irányított (bemeneti és kimeneti) kapcsolatokkal összekötött egységekből állnak. Az a(0-j) az inputról (más idegsejtekből) bejövő axonok, kapcsolatok (az "a"-k). Ezekből az "a"-kból rendre csinál egy eltolási súlyt, a W-t, ami meghatározza a kapcsolat erősséget és előjelét, majd ezt a megfelelő "a"-val összeszummázza, tehát i a bemenetek egy súlyozott összege lesz. A kapott összegre alkalmazunk egy aktivációs függvényt (g), ezáltal megkapva a kimeneti kapcsolatot.

A forráskód megtalálható a következő linken is: [../Caesar/neutoae.r](#)

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó: <https://youtu.be/XpBnR31BRJY>

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

Ez a program sokban kötődik az előző programhoz, mivel ebben is egyfajta tanítási módszert mutatunk be. A perceptronok, más néven egyrétegű előrecsatolt neurális hálók, olyan hálók amelyekben az összes bemenet közvetlenül a kimenetekre kapcsolódik (minden súly csak egy kimenetre van hatással). A perceptronok az alap logikai műveleteken túl képesek a bonyolultabbakat is bemutatni röviden. A hiba-visszaterjesztést többrétegű perceptron (MLP) esetén tudjuk csak alkalmazni. A többrétegű perceptron rétegekbe szervezett neuronokból áll. A rétegek mennyisége többnyire változó, minden esetben van egy bemeneti-, egy kimeneti- és a kettő között egy vagy több rejtett réteg A következő programok együttese olyan algoritmus, amelyik megtanítja a gépnek a bináris osztályozást.

A forráskód megtalálható a következő linken: [../Caesar/mandelpng.cpp](#)

Szükségünk van még a következő programra, amelyik megtalálható a következő linken: [../Caesar/perceptron.hpp](#)

A forráskód megtalálható a következő linken: [../Caesar/main.cpp](#)

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

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);

    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);

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

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

A fentebb említett perceptron.cpp programot a mainben meg is hívjuk header fájlként, átláthatóbbá téve a főprogramot. A fő számítások viszont a perceptron.hpp-ben vannak, a mainben az ott deklarált Perceptron osztályt hívjuk segítségül meg az eredmény kiszámolásának céljából. A mainben a png.hpp header fájl segítségével létrehozunk egy új png kiterjesztésű képet, úgyanolyan szélességgel és magasséggel mint a mandelbrotos kép volt. A két egymásbaágazódó for ciklus segítségével végigmegyünk a kép minden pixelén és az előzőekben lementett mandel_perceptron.png pixeleinek piros (red) komponenseit rámásoljuk a most létrehozott kép pixeleire. A program végén pedig kiíratjuk ezt a percceptron értéket a value változó segítségével.

Fordítás: **g++ perceptron.hpp main.cpp -o main -lpng -std=c++11**

Futtatás: **./main mandel_perceptron.png**

A kiadott eredmény: **0.731044**

5. fejezet

Helló, Mandelbrot! Teszt

5.1. A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

Megoldás forrása: [bhax/attention_raising/CUDA/mandelpngt.cpp](#) nevű állománya.

Az ebben a fejezetben szereplő programok, minden szervesen kötődnek egymáshoz. Ha sikerül egy képet megcsinálni akkor az összes programot át tudjuk rá ültetni, tehát egy másik programváltozattal tudunk benne majd nagyítani, vagy úgy is át tudjuk majd írni, hogy a GPU-t használja, ezáltal nagyon meggyorsítva a számítási folyamatokat.

A forráskód megtalálható a következő linken is: [..//Mandelbrot/mandelpngt.cpp](#)

```
// mandelpngt.cpp
// Copyright (C) 2019
// Norbert Bátfai, batfai.norbert@inf.unideb.hu
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <https://www.gnu.org/licenses/>.
//
// Version history
//
// Mandelbrot png
// Programozó Páternoszter/PARP
// https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063\_01\_parhuzamos\_prog\_linux
//
// https://youtu.be/gvaqijHlRUs
//
#include <iostream>
#include "png++/png.hpp"
#include <sys/times.h>

#define MERET 600
#define ITER_HAT 32000
```

A program elején, mint minden, megadjuk a header fájlokat. Ez esetben ami ismeretlen lehet az a png++/png.hpp és a sys/times.h fejlécek. Az elsőre azért van szükségünk, hogy létre tudjuk hozni a képet, hogy viziálisan is láthatunk a Mandelbrot halmaz eredményét, ami kép. A második header fájl pedig az idővel kapcsolatos számításokat, az idővel összefüggő függvények meghívását biztosítja számunkra. Ildletve efiníció megadaások is szerepelnek, melyek majd könnyítik a dolgunkat a program írása alatt.

```
void
mandel (int kepadat [MERET] [MERET]) {

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

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

    // a számítás
    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, rez, imZ, ujrez, ujimZ;
    // Hány iterációt csináltunk?
    int iteracio = 0;
    // Végigzongorázzuk a szélesség x magasság rácsot:
    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 elértek, akkor úgy vesszük,
            // hogy a kiinduláci 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;
        }
    }

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

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

A `mandel` függvény az ami majd kiszámolja nekünk a Mandelbrot halmazt. Egy időszámítás jön, ami segítségével majd a program lefutás után kiírja azt az időt amit igénybe vett a számolás (az én gépen és egy i3-3227U-s processzor esetén ez körülbelül egy 15 másodpercbe telt). Ezek és a változódeklárálások után jöhetnek a nagyobb számítások. Létrehozunk egy `dx` szélességű és `dy` magasságú rácsot a két egymásbaágazódó for ciklus segítségével, majd megvizsgáljuk, hogy a `c` komplex szám (mely megkapja a `reC` és a `imC` értékek által meghatározott pontot, pixelt) benne van-e a Mandelbrot halmazban, ha igen akkor azt a pontot beszínezi.

```
int
main (int argc, char *argv[])
{
    if (argc != 2)
    {
        std::cout << "Használat: ./mandelpng fajlnev";
        return -1;
    }

    int kepadat [MERET] [MERET];

    mandel (kepadat);

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

    for (int j = 0; j < MERET; ++j)
    {
        //sor = j;
        for (int k = 0; k < MERET; ++k)
        {
            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));
        }
    }

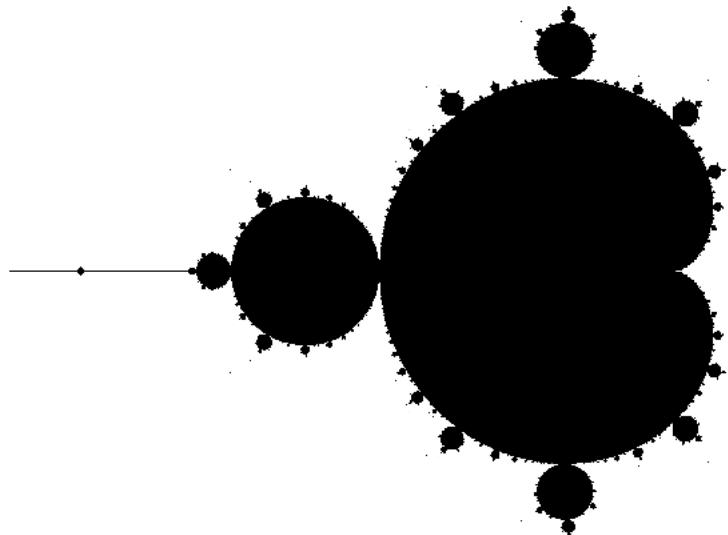
    kep.write (argv[1]);
    std::cout << argv[1] << " mentve" << std::endl;
}
```

A főprogram elején megvizsgáljuk, hogy a felhasználó jól használta-e a futtatási parancsot, tehát hogyha a parancs nem két argumentumból áll, akkor kiírunk egy hibaüzenetet a kimenetre. A `kepadat` változóba bekérjük a méreteket, tehát a kép szélességét és a magasságát majd ezeket átadjuk a `mandel` függvénynek, ami kiszámolja nekünk a mandelbrot halmazt, és ez alapján megkotjuk a `png` kiterjesztésű képet. Miutána program befejezte a számításokat és sikeresen megcsinálta a képet, kiírja a kimenetre a "mentve" üzenetet a felhasználó számára, tudatva a sikert.

Fordítás: **g++ mandelpngt.cpp -lpng -O3 -omandelpngt**

Futtatás: **./mandelpngt mt.png**

Kép megnyitása terminálból: **eog mt.png**



5.1. ábra. A kiadott kép

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

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

Megoldás forrása:

A [Mandelbrot halmaz](#) pontban vázolt ismert algoritmust valósítja meg a repó [bhax/attention_raising/Mandelbrot/3.1.2.cpp](#) nevű állománya.

Felvetődik egy kérdés: melyiket a szám amelyiket önmagával megszorozva 9-et kapunk (természetesen ez a 3). Ebből kiindulva a következő kérdés pedig az, hogy melyik az a szám amit ha megszorozunk önmagával -9-et kapunk? Ez pedig már nem lehetséges a valós számhalmazon, így jönnek képbe a komplex számok, melyek úgymond a valós számhalmaz továbbbővítése. A komplex számok alapja az "i" szám, melynek értéke a $\sqrt{-1}$, ennek a segítségével el lehet végezni a negatív számból való négyzetgyökvonást. Így már meg tudjuk válaszolni az előző feltett kérdést, az "i" szám segítségével már ki tudjuk hozni a -9-et (tehát $3i^2 = -9$).

Ez a program és az előző közötti legnagyobb kölönbség az hogy a c amit vizsgálunk, hogy benne van-e a Mandelbrot halmazban, az előző programban egy változó, ebben pedig egy állandó. Így itt a c a rács minden vizsgálandó pontját befutja.

A forráskód megtalálható a következő linken is: [../Mandelbrot/mandelbrotcomplex.cpp](#)

```
// Verzio: 3.1.2.cpp
// Forditas:
// g++ 3.1.2.cpp -lpng -O3 -o 3.1.2
// Futtatas:
// ./3.1.2 mandel.png 1920 1080 2040 -0.01947381057309366392260585598705802112818 ←
// -0.0194738105725413418456426484226540196687 0.7985057569338268601555341774655971676111 ←
// 0.798505756934379196110285192844457924366
// ./3.1.2 mandel.png 1920 1080 1020 0.4127655418209589255340574709407519549131 ←
// 0.4127655418245818053080142817634623497725 0.2135387051768746491386963270997512154281 ←
// 0.2135387051804975289126531379224616102874
```

```
// Nyomtatás:  
// a2ps 3.1.2.cpp -o 3.1.2.cpp.pdf -l --line-numbers=1 --left-footer="BATF41 HAXOR STR34M" ←  
// --right-footer="https://bhaxor.blog.hu/" --pro=color  
// ps2pdf 3.1.2.cpp.pdf 3.1.2.cpp.pdf  
//  
//  
// Copyright (C) 2019  
// Norbert Bátfai, batfai.norbert@inf.unideb.hu  
//  
// This program is free software: you can redistribute it and/or modify  
// it under the terms of the GNU General Public License as published by  
// the Free Software Foundation, either version 3 of the License, or  
// (at your option) any later version.  
//  
// This program is distributed in the hope that it will be useful,  
// but WITHOUT ANY WARRANTY; without even the implied warranty of  
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
// GNU General Public License for more details.  
//  
// You should have received a copy of the GNU General Public License  
// along with this program. If not, see <https://www.gnu.org/licenses/>.  
  
#include <iostream>  
#include "png++/png.hpp"  
#include <complex>  
  
int  
main ( int argc, char *argv[] )  
{  
  
    int szelesseg = 1920;  
    int magassag = 1080;  
    int iteraciosHatar = 255;  
    double a = -1.9;  
    double b = 0.7;  
    double c = -1.3;  
    double d = 1.3;  
  
    if ( argc == 9 )  
    {  
        szelesseg = atoi ( argv[2] );  
        magassag = atoi ( argv[3] );  
        iteraciosHatar = atoi ( argv[4] );  
        a = atof ( argv[5] );  
        b = atof ( argv[6] );  
        c = atof ( argv[7] );  
        d = atof ( argv[8] );  
    }  
    else  
    {  
        std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c d" << std::endl;  
        return -1;  
    }  
}
```

A program elején a header fájlok deklarálása után (iostream, a cin, cout miatt főleg; png++/png.hpp, a kép létrehozása miatt; complex, a komplex iterációk és ahogyan a címben is említve van a komplex osztályos megoldás miatt) jönnek a változók deklarációja, ezek a váltoozók (szelesseg, magassag, iteraciosHatar, a, b, c és d) a futtaási parancsban is fontos szerepet játszanak hiszen az első két argumentum után (ami a ./futtató fájl neve illetve egy png fájlneve, ami a mentett kép neve lesz) ahogyan deklárolva vannak, úgynilyen sorrenbe vannak megadva szóközzel elválasztva egymástól. Miután a változók deklárolva lettek, megvizsgáljuk hogy a felhasználó jól futtat-e a programot, tehát hogyha a parancs 9 argumentumból áll akkor megadjuk a prog-

ramban, hogy az egyes változók hányadik értéket kapják meg a parancsból, tehát hogy a program tudja hogy az egyes értékek mit jelentenek számára. Ha vizsont a parancs nem 9 argumentumból áll, akkor kiíratunk az outputra egy üzenetet a felhasználónak, hogy tudja a helyes futtatási használatot, **return -1**-el pedig az operációs rendszernek is jelezük a hibát.

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

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

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

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

        kep.set_pixel ( k, j,
                        png::rgb_pixel ( iteracio%255, (iteracio*iteracio)%255, 0 ) );
    }

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

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

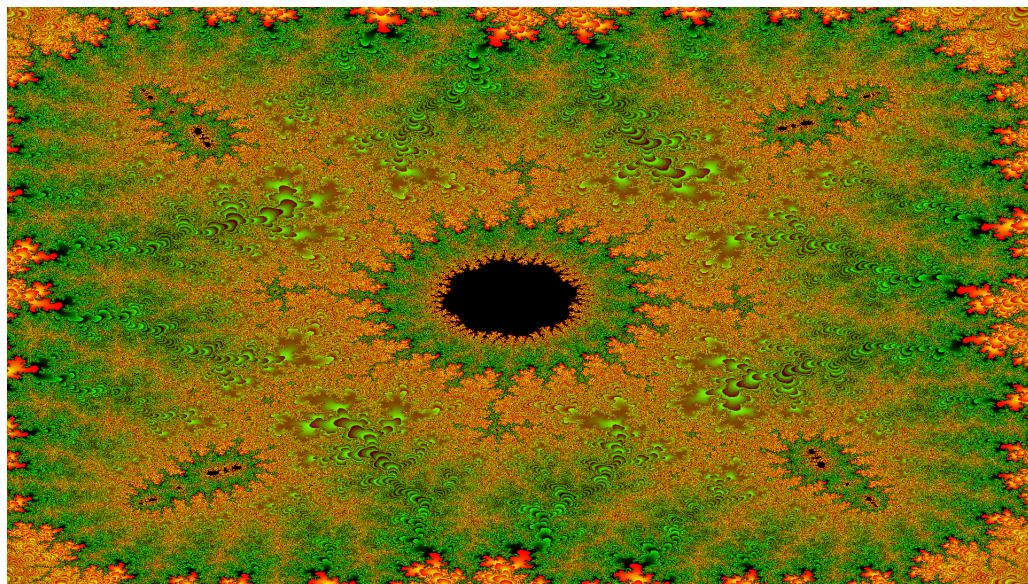
}

A számítások részben deklaráljuk a `reC`, `imC`, `reZ` és `imZ` (valós és imaginárius részű) változókat, melyekből már látszik is hogy a feladatban sokszerepet fognak játszani a komplex számok. A `c=(reC, imC)` a haló rácpontjainak megfelelő aktualis pont. Úgyanúgy mint az előző programban a két for ciklus segítségével megcsináljuk a rácsot, elindulunk az origóban majd egy `c` pontra ugrunk a c^2+c majd ez az egész a négyzetben $+c$ és így tovább. Ha sikeres volt minden művelet, akkor kiíratunk a standard kimenetre egy "mentve" üzenetet, melyből a felhasználó tudja is hogy a kép sikeresen el lett készítve, lehet megtekinteni.

Fordítás: **g++ mandelbrotcomplex.cpp -lpng -O3 -o mandelbrotcomplex**

Futtatás: **./mandelbrotcomplex mandelcomplex.png 1920 1080 1020 0.4127655418209589255340574709407519549131 0.4127655418245818053080142817634623497725 0.2135387051768746491386963270997512154281 0.21353870518049752891265**

Kép megnyitása terminálból: **eog mandelcomplex.png**



5.2. ábra. A program áltak megcsinált kép

5.3. Biomorfok

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

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

A cikkben arról olvashatunk, hogy Pickover amikor felfedezte a biomorfokat, teljesen meg volt győződve arról hogy felfedezte a természet törvényeit, tehát hogy hogyan néznek ki és alakulnak ki az élő orgazmusok. A cikkben látjuk azt is, hogy a különböző képeket különböző függvények segítségével hozunk létre, hasonlóan a Mandelbrot halmazhoz, elindul a rácson és azon végzi el a képhez tartozó függvényt. A program a komplex számsíkon dolgozik, tehát van "i" számunk, és a c egy állandó. Próbáljuk ki és nézzük meg hogy hogy is működik a program.

A forráskód megtalálható a következő linken is: [..../Mandelbrot/biomorfok.cpp](#)

```
// Verzio: 3.1.3.cpp
// Forditas:
// g++ 3.1.3.cpp -lpng -O3 -o 3.1.3
// Futtatas:
// ./3.1.3 bmorf.png 800 800 10 -2 2 -2 2 .285 0 10
// Nyomtatás:
// a2ps 3.1.3.cpp -o 3.1.3.cpp.pdf -1 --line-numbers=1 --left-footer="BATF41 HAXOR STR34M" ←
// --right-footer="https://bhaxor.blog.hu/" --pro=color
//
// BHAX Biomorphs
// Copyright (C) 2019
// Norbert Batfai, batfai.norbert@inf.unideb.hu
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <https://www.gnu.org/licenses/>.
//
// Version history
//
// https://youtu.be/IJMbqRzY76E
// See also https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9_Iss5_2305--2315 ↫
// _Biomorphs_via_modified_iterations.pdf
//

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

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

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

    if ( argc == 12 )
    {
        szelesseg = atoi ( argv[2] );
        magassag = atoi ( argv[3] );
        iteraciosHatar = atoi ( argv[4] );
        xmin = atof ( argv[5] );
        xmax = atof ( argv[6] );
        ymin = atof ( argv[7] );
        ymax = atof ( argv[8] );
        reC = atof ( argv[9] );
        imC = atof ( argv[10] );
        R = atof ( argv[11] );
    }
    else
    {
        std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c d reC imC R" << ↫
            std::endl;
        return -1;
    }
}
```

Header fájlként megadjuk a képkészítéshez a png++/png.hpp fejlécet, majd ami új az pedig a complex header fájl ami természetesen a komplex számokkal való számolás miatt kell. Ezek után már kezdjük is a főprogramot, melyben legelőször deklaráljuk a változókat, majd meghatározzuk a magasságot, szálességet és a rácsban való mozgáshoz szükségez koordinátákat. Ez a kép kirajzoltatásához a c állandó értékében most nincs "i", ezért a c-nek a valós részéhez (reC) adunk meg 0-tól különböző értéket, az imaginárius, képzett részéhez (imC) pedig 0-t.

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

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

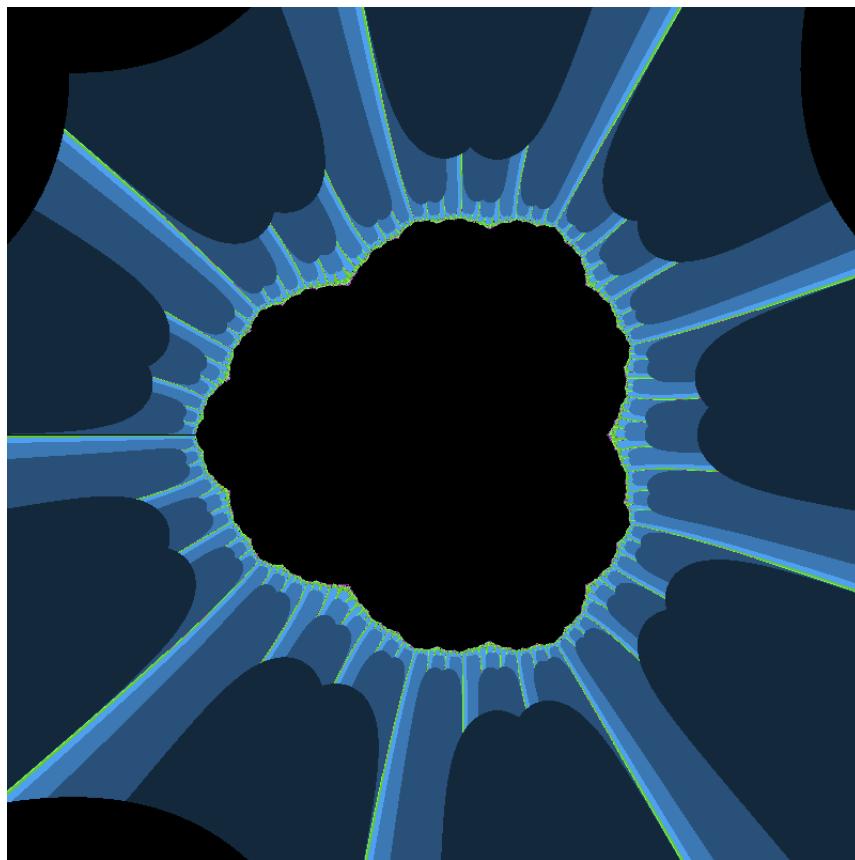
```
std::complex<double> cc ( reC, imC );  
  
std::cout << "Szamitas\n";  
  
// j megy a sorokon  
for ( int y = 0; y < magassag; ++y )  
{  
    // k megy az oszlopokon  
  
    for ( int x = 0; x < szelessseg; ++x )  
    {  
  
        double rez = xmin + x * dx;  
        double imZ = ymax - y * dy;  
        std::complex<double> z_n ( rez, imZ );  
  
        int iteracio = 0;  
        for (int i=0; i < iteraciosHatar; ++i)  
        {  
  
            z_n = std::pow(z_n, 3) + cc;  
            //z_n = std::pow(z_n, 2) + std::sin(z_n) + cc;  
            if (std::real ( z_n ) > R || std::imag ( z_n ) > R)  
            {  
                iteracio = i;  
                break;  
            }  
        }  
  
        kep.set_pixel ( x, y,  
                        png::rgb_pixel ( (iteracio*20)%255, (iteracio*40)%255, ( iteracio*60)%255 ));  
    }  
  
    int szazalek = ( double ) y / ( double ) magassag * 100.0;  
    std::cout << "\r" << szazalek << "%" << std::flush;  
}  
  
kep.write ( argv[1] );  
std::cout << "\r" << argv[1] << " mentve." << std::endl;  
}
```

A következő programrészben, az első pixelre állunk, majd elkezdjük a számításokat. A két egymásbaágynó for ciklussal rácsot teszünk a komplex számsíkra (ahol a j-vel a soron, a k-al pedig az oszlopon megyünk végig). Az előző complex osztállyal megvalósított Mandelbrot halmazzal szemben, itt a cc nem változik, hanem minden vizsgált z rácspontra ugyanaz, állandó lesz (a program eme változata a Júlia halmaz része). A mi programunkban a függvény a következőképpen néz ki: $z_n^3 + c$. A függvény változtatásával a készített kép is fog változni.

Fordítás: **g++ biomorfok.cpp -lpng -O3 -o biomorfok**

Futtatás: **./biomorfok biomorf.png 800 800 10 -2 2 -2 2 .285 0 10**

Kép megnyitása terminálból: **eog biomorf.png**



5.3. ábra. Biomorf kép

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás forrása: [bhax/attention_raising/CUDA/mandelpngc_60x60_100.cu](#) nevű állománya.

A forráskód megtalálható a következő linken is: [./Mandelbrot/mandelpngc_60x60_100.cu](#)

```
// mandelpngc_60x60_100.cu
// Copyright (C) 2019
// Norbert Bátfai, batfai.norbert@inf.unideb.hu
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <https://www.gnu.org/licenses/>.
//
// Version history
//
// Mandelbrot png
// Programozó Páternoszter/PARP
```

```
//  https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_01_parhuzamos_prog_linux
//  https://youtu.be/gvaqijHlRUs
//

#include <png++/image.hpp>
#include <png++/rgb_pixel.hpp>

#include <sys/times.h>
#include <iostream>

#define MERET 600
#define ITER_HAT 32000

__device__ int
mandel (int k, int j)
{
    // számítás adatai
    float a = -2.0, b = .7, c = -1.35, d = 1.35;
    int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;

    // a számítás
    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, reZ, imZ, ujreZ, ujimZ;
    // Hány iterációt csináltunk?
    int iteracio = 0;

    // c = (reC, imC) a rács csomópontjainak
    // megfelelő komplex szám
    reC = a + k * dx;
    imC = d - j * dy;
    // z_0 = 0 = (reZ, imZ)
    reZ = 0.0;
    imZ = 0.0;
    iteracio = 0;

    while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
    {
        // z_{n+1} = z_n * z_n + c
        ujreZ = reZ * reZ - imZ * imZ + reC;
        ujimZ = 2 * reZ * imZ + imC;
        reZ = ujreZ;
        imZ = ujimZ;

        ++iteracio;
    }
    return iteracio;
}

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

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

```
    kepadat[j + k * MERET] = mandel(j, k);
}

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

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

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

int
main (int argc, char *argv[])
{
    // Mérünk időt (PP 64)
    clock_t delta = clock ();
    // Mérünk időt (PP 66)
    struct tms tmsbuf1, tmsbuf2;
    times (&tmsbuf1);

    if (argc != 2)
    {
        std::cout << "Hasznalat: ./mandelpngc fajlnev";
        return -1;
    }

    int kepadat[MERET][MERET];
    cudamandel (kepadat);
    png::image<png::rgb_pixel> kep (MERET, MERET);

    for (int j = 0; j < MERET; ++j)
    {
        //sor = j;
        for (int k = 0; k < MERET; ++k)
        {
            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));
        }
    }
    kep.write (argv[1]);

    std::cout << argv[1] << " mentve" << std::endl;

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

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

```
}
```

Ugynúgy mint a legelső Mandelbrotos feladat, a mandelbrot halmazt számolja ki, hasonló műveletekkel, viszont van egy nagy különbség a kettő között: míg az első a CPU-t használja a számítások közben, addig ez a program a GPU-t használja. A program futtatása után a számok melyek az időt mutatják magukért beszélnek. Míg a CPU-t használó program 15 másodpercet vett igénybe, addig CUDA-s program mintegy 0.15 másodperc alatt kiszámolja nekünk ugyanazt az eredményt. A két program leírásban nagyon hasonlít az előző feladathoz, mint mondtam itt a számítási idők mások, ezt az időt a program során számoljuk és iratjuk ki, hogy legyen ű vizsgálati alapunk.

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!

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

Megoldás forrása: az ötödik előadás 26-33 fólia, illetve <https://sourceforge.net/p/udprog/code/ci/master/tree/source/binom/Batfai-Barki/frak/>.

A forráskód megtalálható a következő linken is: [..../Mandelbrot/mandelbrot_nagyito.cpp](#)

A Mandelbrot halmaz kiszámítása mellett ezzel a programmal a kiadott képen nagyítást is tudunk végrehaktani. A kép megnyitása után az egerünk görgőjének segítségével tudunk benne nagyítani, illetve touchpad-en a kétujjas mozdulattal.

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

int
main ( int argc, char *argv[] )
{
    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double a = -1.9;
    double b = 0.7;
    double c = -1.3;
    double d = 1.3;

    if ( argc == 9 )
    {
        szelesseg = atoi ( argv[2] );
        magassag = atoi ( argv[3] );
        iteraciosHatar = atoi ( argv[4] );
        a = atof ( argv[5] );
        b = atof ( argv[6] );
        c = atof ( argv[7] );
        d = atof ( argv[8] );
    }
    else
    {
        std::cout << "Hasznalat: ./3.1.1 fajlnev szelesseg magassag n a b c d" << std::endl;
        std::cout << "Most az alapbeallitasokkal futtatjuk " << szelesseg << " "
        << magassag << " "
        << iteraciosHatar << " "
        << a << " "
        << b << " "
        << c << " "
        << d << " " << std::endl;
        //return -1;
    }
}
```

A program eleje ismerős hiszen úgyanaz mint az előző programok esetében: header fájlok deklarálása, majd a változók beadolása (szelesseg, magassag, iteraciosHatar, a, b, c és d). Most is megvizsgáljuk hogy a parancs amivel a felhasználó futtatta a programot, az tényleg 9 argumentumból áll-e. Ha igen akkor az az egyes argumentumokat, értékeket átadjuk az egyes változóknak, ha viszont az argumentumok száma nem felel meg, akkor használati utasítást adunk a programot futtatónak hogy mi lenne a helyes mód, illetve még segítségképp megmutatjuk azt is hogy mik voltak az általunk adott alap beállítások.

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

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

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

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

        iteracio %= 256;

        kep.set_pixel ( k, j,
            png::rgb_pixel ( iteracio%255, 0, 0 ) );
    }
}

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

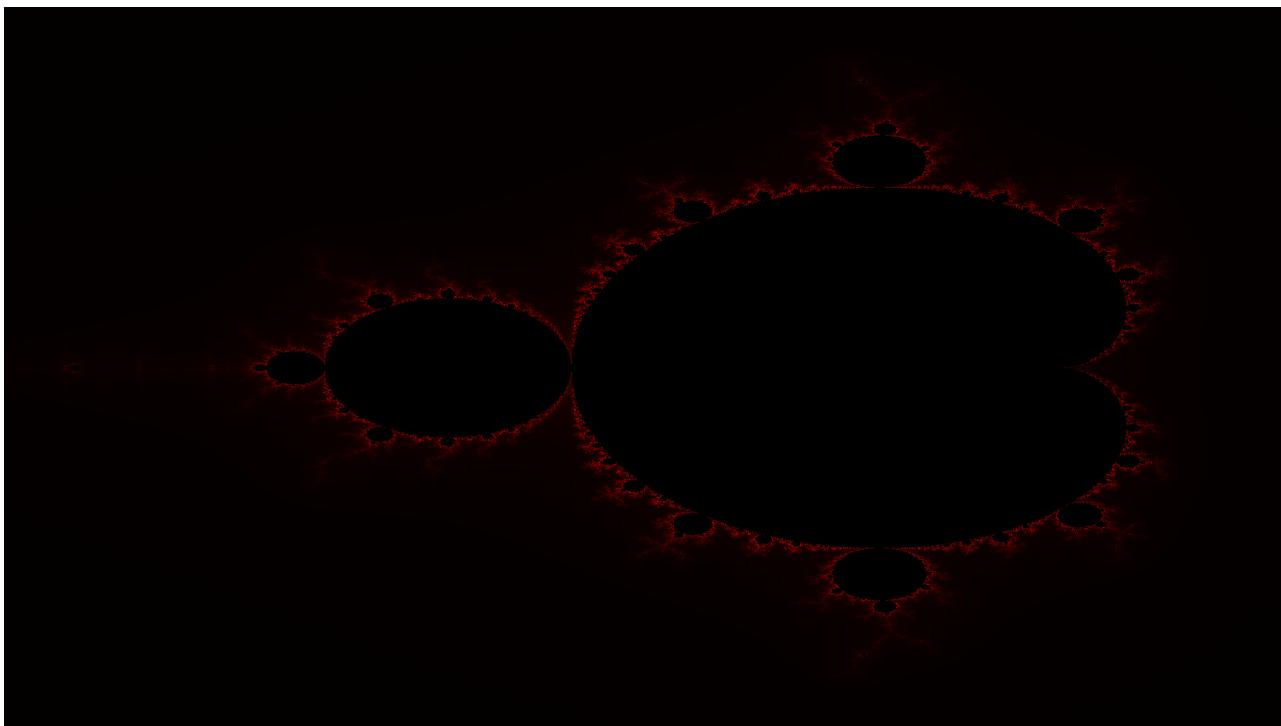
kep.write ( argv[1] );
std::cout << "\r" << argv[1] << " mentve." << std::endl;
}
```

A program utolsó része ugynőgy komplexszámos megoldással van csinálva. A két for ciklussal végigmegyünk a rácson j-val a soron, k-val az oszlopon. A a reC, imC, reZ és imZ (valós és imaginárius részű) változókat használva végigmegyünk minden rácsponton úgy, hogy a c=(reC, imC) a haló rácspontjainak megfelelő komplex szám. A megadott képletet használva megalkotjuk a várt képet, megkapjuk a "mentve" üzenetet, megnyitjuk a képet és tetszsünk szerint nagyítgathatunk a kapott képbén aszerint ahogy a feladat legelején bemutattam.

Fordítás: g++ mandelbrotnagyito.cpp -lpng16 -O3 -o mandelbrotnagy

Futtatás: ./mandelbrotnagy mandelnagy.png 1920 1080 1020 0.41276554182095892553405747094075195491310.412765541824581
0.2135387051768746491386963270997512154281 0.2135387051804975289126531379224616102874

Kép megnyitása terminálból: eog mandelnagy.png



5.4. ábra. C++ mandelbrot nagyító és utazó

5.6. Mandelbrot nagyító és utazó Java nyelven

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

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

A program célja ugyanaz mint az előző programé: nagyítunk a kapott képben. A java-s program úgy van megírva, hogy itt a bal egérgomb segítségével kell kijelöljük azt a részt amit pontosabban, "közelebbről" megszeretnénk tekinteni. Ezt a műveletet nagyon sokszor tudjuk megcsinálni egymás után. A programban commentként rengeteg magyarázatot találunk, melyek a különböző sorok, parancsok működését írják le.

A forráskód megtalálható a következő linken is: [..//Mandelbrot/MandelbrotHalmazNagyito.java](http://Mandelbrot/MandelbrotHalmazNagyito.java)

```
/*
 * MandelbrotHalmazNagyító.java
 *
 * DIGIT 2005, Javat tanítok
 * Bátfai Norbert, nbatfai@inf.unideb.hu
 *
 */
/**
 * A Mandelbrot halmazt nagyító és kirajzoló osztály.
 *
 * @author Bátfai Norbert, nbatfai@inf.unideb.hu
 * @version 0.0.1
 */
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {
    /** A nagyítandó kijelölt területet bal felső sarka. */
    private int x, y;
    /** A nagyítandó kijelölt terület szélessége és magassága. */
    private int mx, my;
    /**
```

```

* Létrehoz egy a Mandelbrot halmazt a komplex sík
* [a,b]x[c,d] tartománya felett kiszámoló és nyíltan tudó
* <code>MandelbrotHalmazNagyító</code> objektumot.
*
* @param      a          a [a,b]x[c,d] tartomány a koordinátája.
* @param      b          a [a,b]x[c,d] tartomány b koordinátája.
* @param      c          a [a,b]x[c,d] tartomány c koordinátája.
* @param      d          a [a,b]x[c,d] tartomány d koordinátája.
* @param      szélesség    a halmazt tartalmazó tömb szélessége.
* @param      iterációsHatár a számítás pontossága.
*/
public MandelbrotHalmazNagyító(double a, double b, double c, double d,
    int szélesség, int iterációsHatár) {
    super(a, b, c, d, szélesség, iterációsHatár);
    setTitle("A Mandelbrot halmaz nagyításai");
    addMouseListener(new java.awt.event.MouseAdapter() {
        public void mousePressed(java.awt.event.MouseEvent m) {
            x = m.getX();
            y = m.getY();
            mx = 0;
            my = 0;
            repaint();
        }
        public void mouseReleased(java.awt.event.MouseEvent m) {
            double dx = (MandelbrotHalmazNagyító.this.b
                - MandelbrotHalmazNagyító.this.a)
                /MandelbrotHalmazNagyító.this.szélesség;
            double dy = (MandelbrotHalmazNagyító.this.d
                - MandelbrotHalmazNagyító.this.c)
                /MandelbrotHalmazNagyító.this.magasság;
            new MandelbrotHalmazNagyító(MandelbrotHalmazNagyító.this.a+x*dx,
                MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,
                MandelbrotHalmazNagyító.this.d-y*dy-my*dy,
                MandelbrotHalmazNagyító.this.d-y*dy,
                600,
                MandelbrotHalmazNagyító.this.iterációsHatár);
        }
    });
}

```

A program elején úgyanúgy mint ahogy megszoktuk jönnek a változók deklarálásai, ahol az x és az y változókkal határoljuk be a nagytárdi terület bal felső sarkát, illetve az mx és my pedig a kijelölt nagytárdi terület magassága és szélessége. Vjuk az ōs osztály konstruktőrét, beállítjuk az ablak címét (amiben majd nagytárdi) és deklaráljuk az egér kattintás műveletét. Az azonnal következő mousePressed függvény segítségével deklaráljuk a nagytárdi terület bal felső sarkát, szélességét és magasságát. A következő mouseReleased függvénytel az a művelet aktivizálódik, amikor kattintjuk az egeret, vonszolással kijelöljük a nagytárdi kívánt területet és amikor elengedjük az egér gombot akkor a kijelölt terület újraszámítása kezdődik el, illetve megalkotódik a kinagyított terület, az lesz az aktuális kép amit az ablakban látunk és továbbnagytárdi majd.

```

addMouseMotionListener(new java.awt.event.MouseMotionAdapter());
public void mouseDragged(java.awt.event.MouseEvent m) {
    mx = m.getX() - x;
    my = m.getY() - y;
    repaint();
}
});
}

```

Itt deklaráljuk a fentebb már használt függvényt, ami arra szolgál hogy követni lehessen a egér mozgását, kattintásának feldolgozását, a kijelölés menetét mindenkorral hogy közben megadjuk a mouseDragged függvényt (ami nem más mint a vonszolás, a kijelölés menete, az új szélesség és a magasság bellítása és az újraméretezés által).

```
public void pillanatfelvétel() {
```

```
java.awt.image.BufferedImage mentKép =
    new java.awt.image.BufferedImage(szélesség, magasság,
        java.awt.image.BufferedImage.TYPE_INT_RGB);
java.awt.Graphics g = mentKép.getGraphics();
g.drawImage(kép, 0, 0, this);
g.setColor(java.awt.Color.BLUE);
g.drawString("a=" + a, 10, 15);
g.drawString("b=" + b, 10, 30);
g.drawString("c=" + c, 10, 45);
g.drawString("d=" + d, 10, 60);
g.drawString("n=" + iterációsHatár, 10, 75);
if(számításFut) {
    g.setColor(java.awt.Color.RED);
    g.drawLine(0, sor, getWidth(), sor);
}
g.setColor(java.awt.Color.GREEN);
g.drawRect(x, y, mx, my);
g.dispose();
StringBuffer sb = new StringBuffer();
sb = sb.delete(0, sb.length());
sb.append("MandelbrotHalmazNagyítás_");
sb.append(++pillanatfelvételSzámláló);
sb.append("_");
sb.append(a);
sb.append("_");
sb.append(b);
sb.append("_");
sb.append(c);
sb.append("_");
sb.append(d);
sb.append(".png");
// png formátumú képet mentünk
try {
    javax.imageio.ImageIO.write(mentKép, "png",
        new java.io.File(sb.toString()));
} catch(java.io.IOException e) {
    e.printStackTrace();
}
}
```

Mint a sima nagyítás nélküli programban, itt is lehetővé tesszük a felhasználó számára a pillanatfelvétel készítésének a lehetőségét. Ebben az esetben figyelme vesszük azt is, hogy az adott képernyőfotó hányadik volt, mivel hogy könnyebben tudjunk majd eligazodni a képek között, a kép nevében szerepelni fog hogy mikor csináltuk azt, hányadikként. A fájl nevébe belevesszük, hogy melyik tartományban találtuk a halmazt:

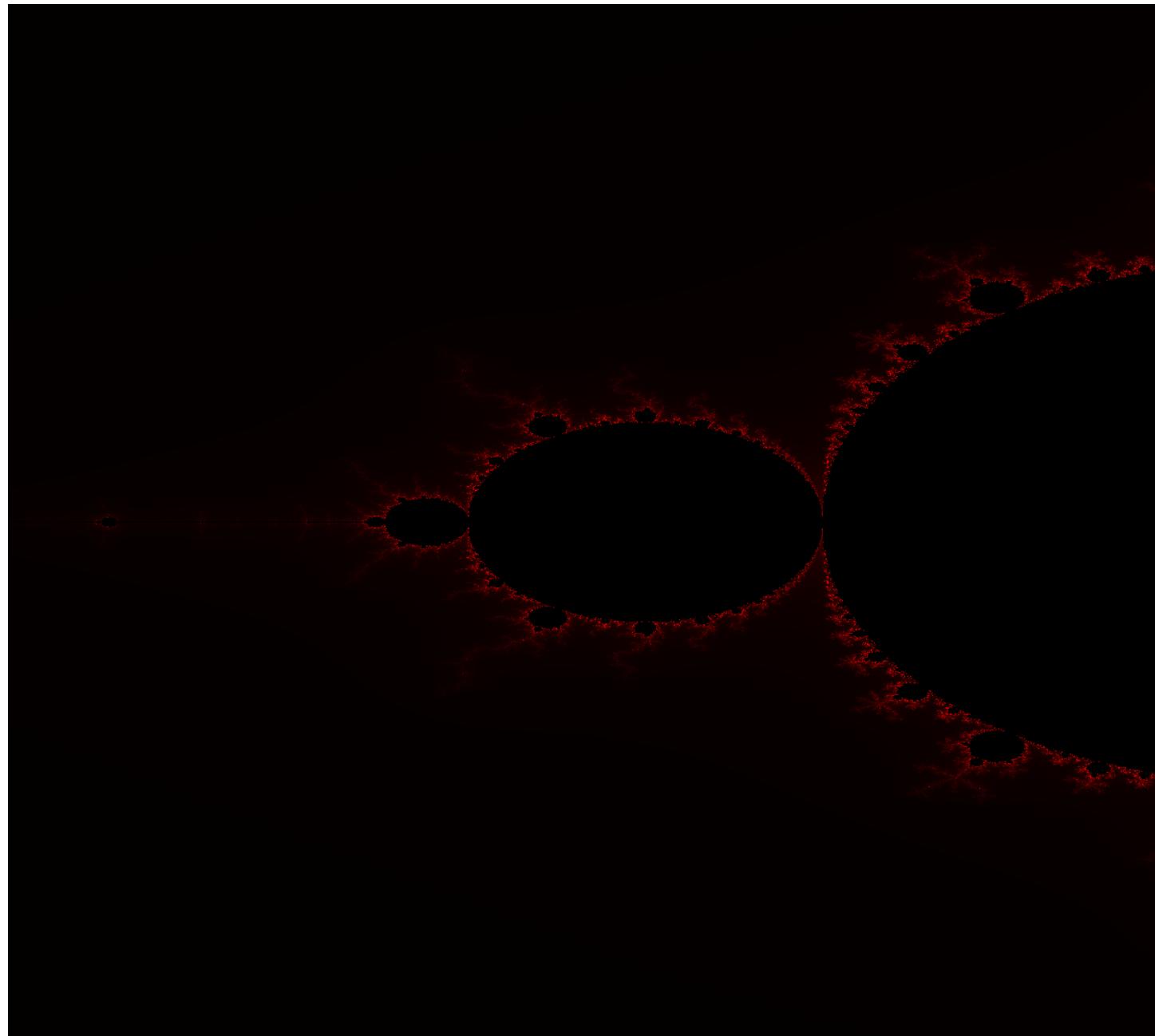
```
public void paint(java.awt.Graphics g) {
    g.drawImage(kép, 0, 0, this);
    if(számításFut) {
        g.setColor(java.awt.Color.RED);
        g.drawLine(0, sor, getWidth(), sor);
    }
    g.setColor(java.awt.Color.GREEN);
    g.drawRect(x, y, mx, my);
}

public static void main(String[] args) {
    new MandelbrotHalmazNagyító(-2.0, .7, -1.35, 1.35, 600, 255);
}
```

A paint segítségével legelőször is kirajzoljuk a Mandelbrot halmazt. Ha a számítások éppen futnak, akkor azt hogy melyik sorban tart egy vörös csíkkal jelöljük. Végül pedig kirajzoltatjuk a nagyítandó terület körponalát zöld színnel.

Fordítás: **javac MandelbrotHalmazNagyító.java**

Futtatás: **java MandelbrotHalmazNagyító**



5.5. ábra. Java mandelbrot nagyító és utazó

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás videó:

Megoldás forrása:

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

PolarGen osztály:

```
#ifndef POLARGEN_H
#define POLARGEN_H

#include <cstdlib>
#include <cmath>
#include <ctime>

class PolarGen
{
public:
    PolarGen ()
    {
        nincsTarolt = true;
        std::srand (std::time (NULL));
    }
    ~PolarGen () {
    }
    double kovetkezo();

private:
    bool nincsTarolt;
    double Tarolt;
};

#endif
```

A PolarGen osztály definiálása:

```
#include "polargen.h"
```

```
double
PolarGen::kovetkezo ()
{
    if(nincsTarolt){
        double u1, u2, v1, v2, w;
        do{
            u1 = std::rand() / (RAND_MAX + 1.0);
            u2 = std::rand() / (RAND_MAX + 1.0);
            v1 = 2*u1 -1;
            v2 = 2*u2 -1;
            w = v1*v1+v2*v2;
        }
        while( w > 1);
        double r = std::sqrt((-2*std::log(w)) / w);

        Tarolt = r*v2;
        nincsTarolt = !nincsTarolt;
        return r*v1;
    }

    else
    {
        nincsTarolt = !nincsTarolt;
        return Tarolt;
    }
}
```

Majd a mainfüggvényben az osztály példányosítása és használata:

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

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

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

    return 0;
}
```

A forrás megtalálható a Welch/Preorder mappában.

Az osztály az objektum-orientált programozásban használt fogalom. Egy programban különböző feladatokat ellátó részeket osztályokra szeparálhatjuk, melyekkel egy sokkal rendezettebb forráskódot írhatunk, amit így könnyebben is kezelhetünk és olvashatunk. Egy osztályban több mező is található, melyek metódusokat, deklarációkat tartalmaznak és így ezek lehetnek publikusak és privátak is, de örökölhetjük is őket. Az osztályokatból objektumokat példányosíthatunk le, melyek az osztály tulajdonságaival fognak rendelkezni.

6.2. LZW

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

Megoldás forrása: https://progpater.blog.hu/2011/03/05/labormeres_oththon_avagy_hogyan_dolgozok_fel_egy_pedat.

Megoldás a 6.6 feladatban

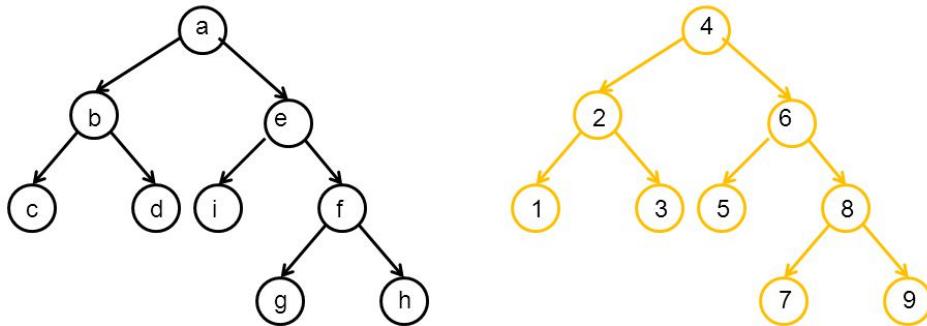
6.3. Fabejárás

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

Megoldás forrása: <https://slideplayer.hu/slide/2200487/>.

Egy bináris fát több féleképpen is bejárhatunk. Három féle bejárási módot ismerünk: Inorder,preorder, posztorder.

Bináris fa inorder bejárása



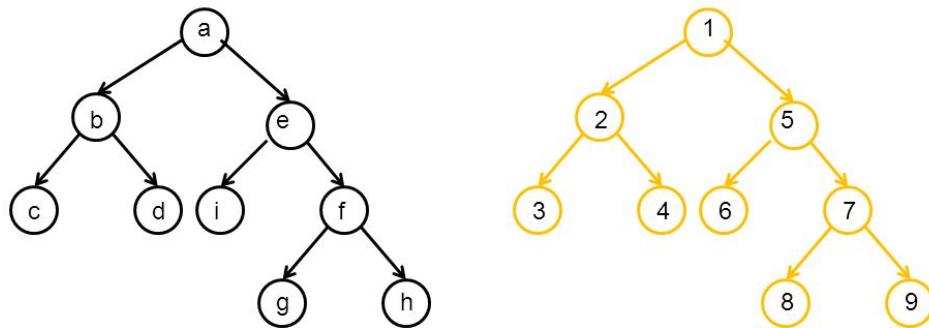
cbdaieghf

19:19:17

14

6.1. ábra. Inorder bejárás

Bináris fa preorder bejárása



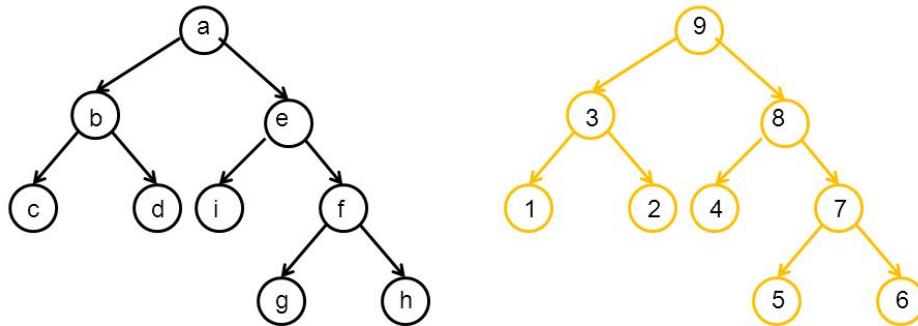
abcdeifgh

19:19:17

12

6.2. ábra. Preorder bejárás

Bináris fa postorder bejárása



cdbighfea

19:19:17

16

6.3. ábra. Postorder bejárás

6.4. Tag a gyökér

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

Megoldás videó: https://youtu.be/_mu54BDkqiQ

Megoldás a 6.6 feladatban

6.5. Mutató a gyökér

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

Megoldás videó: https://youtu.be/_mu54BDkqiQ

Megoldás a 6.6 feladatban

6.6. Mozgató szemantika

Írj az előző programhoz másoló/mozgató konstruktort és értékkadást, a mozgató konstruktur legyen a mozgató értékkedásra alapozva, a másoló értékkedás pedig a másoló konstruktorra!

Megoldás videó: <https://youtu.be/QBD3zh5OJ0Y>

Megoldás forrása:

A forráskód a Welch mappában található LZWBInFa.cpp néven

A bináris fánk felépítését a BinRandTree osztály létrehozásával kezdjük, ez, mint minden osztály a programunkban template osztály lesz, hogy minél szélesebb körben tudjuk használni a fánkat, azaz több típusra is használhatjuk majd.

A BinRandTree osztályunk protected részébe beágyazva található a Node osztály, ami a csomópontokat jelenti. Egy csomópont egy elemet jelent, egy elemnek maximum két gyermek lehet, egy baloldali és egy jobboldali, ezek új csomópontok lesznek.

```
template <typename ValueType>
class BinRandTree {

protected:
    class Node {

private:
    ValueType value;
    Node *left;
    Node *right;
    int count{0};

    Node(const Node &);
    Node & operator=(const Node &);

    Node(Node &&);
    Node & operator=(Node &&);

public:
    Node(ValueType value, int count=0) : value(value), count(count), left(nullptr), right(nullptr) {}
    ValueType getValue() const { return value; }
    Node * leftChild() const { return left; }
    Node * rightChild() const { return right; }
    void leftChild(Node * node){ left = node; }
    void rightChild(Node * node){ right = node; }
    int getCount() const { return count; }
    void incCount() { ++count; }
};

};
```

Szükségünk lesz még mutatókra is, ez lesz a *root (gyökérmutató, ami a fa kezdetét jelenti) és *treep (mutató mellyel bejárhatjuk a fát, ez jelzi hol is járunk éppen).

```
Node *root;
Node *treep;
int depth{0};
```

Ugyanakkor még szükségünk van mozgató és másoló konstruktorra és értékátadásra:

```
public:
    BinRandTree(Node *root = nullptr, Node *treep = nullptr) : root(root), treep(treep) {
        std::cout << "BT ctor" << std::endl;
    }

    BinRandTree(const BinRandTree & old) {      //Másoló konstruktor
        std::cout << "BT copy ctor" << std::endl;
```

```
root = cp(old.root, old.treep);      //A létrehozott fa *rootjának értékül adjuk a ←
    paraméterként kapott fáét

}

Node * cp(Node *node, Node *treep)      //A másoláshoz szükséges függvény, mely egy ←
    Node mutatóval tér vissza
{
    Node * newNode = nullptr;        // Először létrehozunk egy Node mutatót ami egy ←
        nullptr még

//A létrehozott mutatónak allokálunk memóriát a paraméterként kapott node adataival

if(node)
{
    newNode = new Node(node->getValue(), 42 /*node->getCount()*/);
    newNode->leftChild(cp(node->leftChild(), treep));
    newNode->rightChild(cp(node->rightChild(), treep));

    if(node == treep)
        this->treep = newNode;
}

return newNode;
}

BinRandTree & operator=(const BinRandTree & old) {      //Másoló értékkadás
    std::cout << "BT copy assign" << std::endl;

    BinRandTree tmp{old};      //Létrehozunk egy ideiglenes fát ami a paraméterként ←
        kapott fa másolata
    std::swap(*this, tmp);      //majd az egyenlőség bal oldalán lévő fával ←
        megcseréljük.
    return *this;
}

BinRandTree(BinRandTree && old) {      //Mozgató konstruktur
    std::cout << "BT move ctor" << std::endl;

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

BinRandTree & operator=(BinRandTree && old) { //Mozgató értékkadás
    std::cout << "BT move assign" << std::endl;

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

    return *this;
}

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

7. fejezet

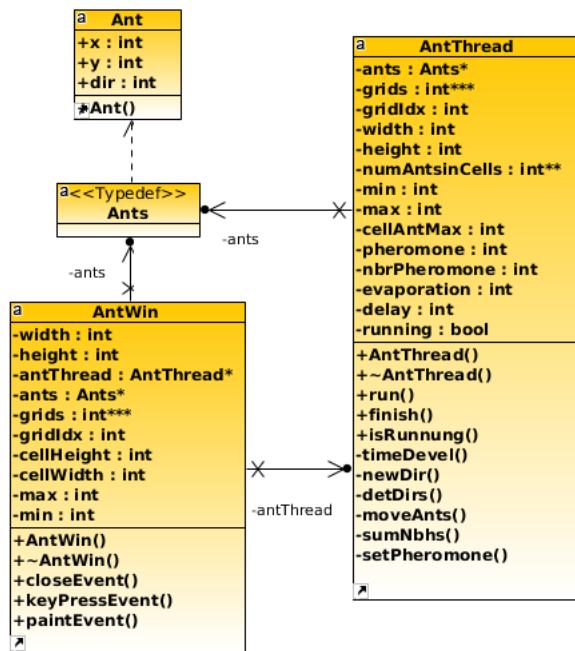
Helló, Conway!

7.1. Hangyszimulációk

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

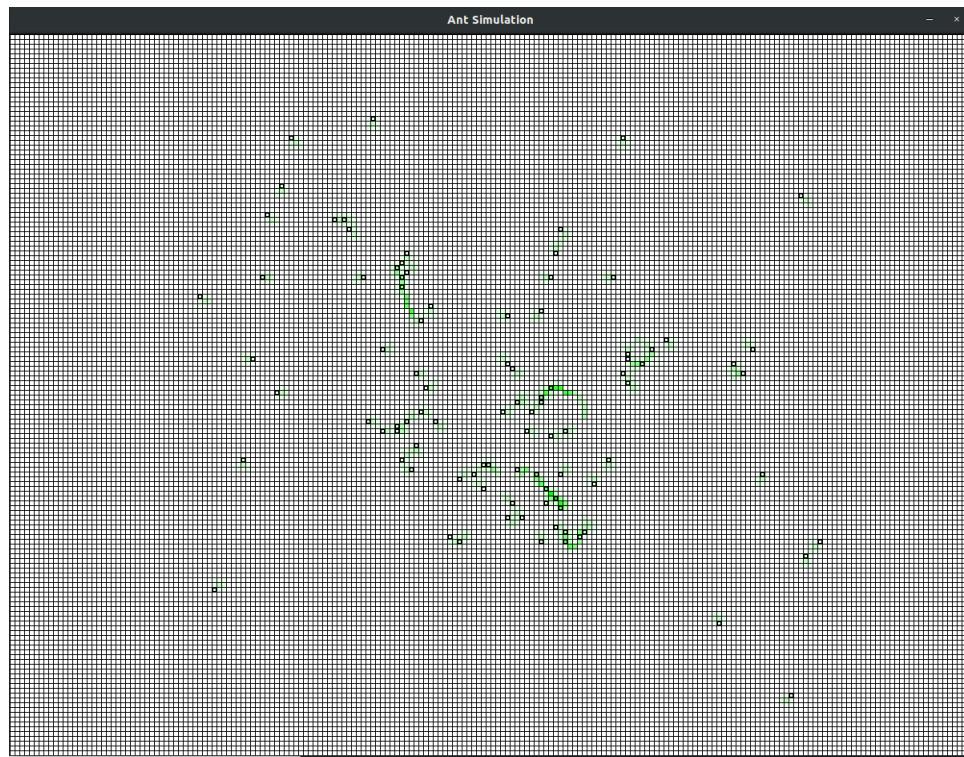
Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

Megoldás forrása: https://gitlab.com/nbatfai/bhax/-/tree/master/attention_raising%2FMyrmecologist



7.1. ábra. A hangyszimulációs program UML diagramja

A programunk egy hangyszimulációs program lesz, mely a hangyáknak egy véletlenszerű mozgását fogja szimulálni. A mozgások egy 2 dimenziós területen fognak történni. A program 3 osztályból épül fel, az **Ant**, **AntThread** és **AntWin** osztályokból, mint azt láthatjuk a fentebb lévő UML diagrammon is.



7.2. ábra. Hangyszimuláció

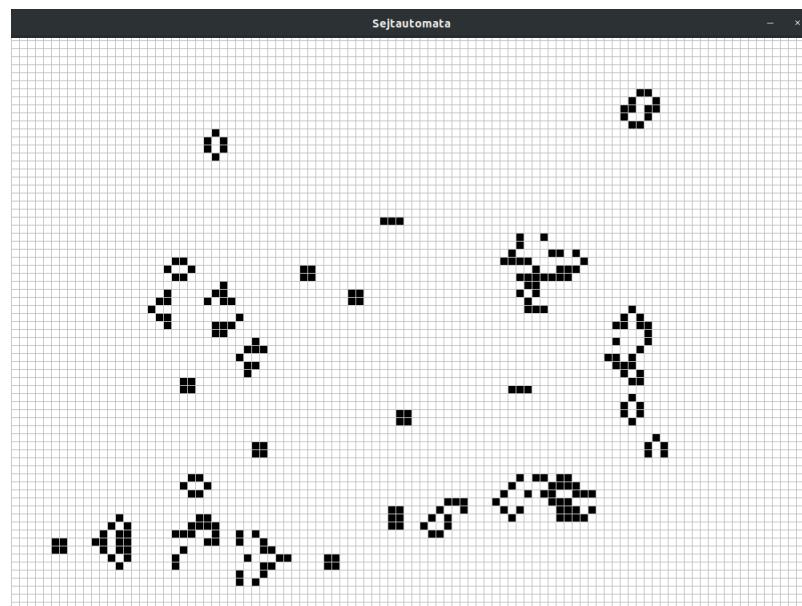
7.2. Java életjáték

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

Megoldás forrása: <https://regi.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apb.html#conway>

Megalátható a forráskód: [Conway/Sejtautomata.java](#)

A játékunkban egy sejt egy sejttér eleme, a sejt állapota lehet élő vagy halott. A adott időben működő sejttér adott sejtjének állapotát a következő időpillanatban a következő szabályok alapján számolhatjuk ki: -Elő sejt élő marad, ha kettő vagy három élő szomszédja van, különben halott lesz. -Halott sejt halott marad, ha három élő szomszédja van, különben élő lesz.



7.3. ábra. Életjáték

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás forrása: https://bhaxor.blog.hu/2018/09/09/ismerkedes_az_eletjatekkal

Megoldás videó: a hivatkozott blogba ágyazva.

Ez a programunk két osztályból fog állni. A SejtAblak osztályunkban állítjuk be az ablakunk méretét szélesség és magasság szerint, és ugyanígy állítjuk be a cellák méretét is. Ezek a SejtAblak konstruktőrában fognak megtörténni

Az ablak magasságát és szélességét inicializáljuk a paraméterként kapott számokkal

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

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

    cellaSzelesseg = 6;
    cellaMagassag = 6;
```

Láthatjuk, hogy a cellák méretétől függően az ablak mérete is változik. A for ciklusokkal a rácson végigmenve a cellákra beállítunk egy-egy boolt, amiből majd látjuk, hogy melyik cella lesz halott és melyik élő sejt.

```
    setFixedSize(QSize(szelesseg*cellaSzelesseg, magassag*cellaMagassag));
    racsok = new bool**[2];
    racsok[0] = new bool*[magassag];
    for(int i=0; i<magassag; ++i)
```

```

    racsok[0][i] = new bool [szelesseg];
    racsok[1] = new bool*[magassag];
    for(int i=0; i<magassag; ++i)
        racsok[1][i] = new bool [szelesseg];

    racsIndex = 0;
    racs = racsok[racsIndex];
    // A kiinduló racs minden cellája HALOTT
    for(int i=0; i<magassag; ++i)
        for(int j=0; j<szelesseg; ++j)
            racs[i][j] = HALOTT;
    // A kiinduló racsra "ELOLényeket" helyezünk
    //siklo(racs, 2, 2);
    sikloKilovo(racs, 5, 60);

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

}

```

Miután a SejtSzal konstruktőrét meghívjuk, végigvizsgáljuk a sejtek szomszédjait a SejtSzal osztályban lévő szomszedokSzama függvényteljesítőjében.

```

int SejtSzal::szomszedokSzama(bool **racs,
                                int sor, int oszlop, bool allapot) {
    int allapotuSzomszed = 0;
    // A nyolcszomszédok végigvizsgálása:
    for(int i=-1; i<2; ++i)
        for(int j=-1; j<2; ++j)
            // A vizsgált sejetet magát kihagyva:
            if(!((i==0) && (j==0))) {
                // A sejttérből szélénél szomszédai
                // a szembe oldalakon ("periódikus határfeltétel")
                int o = oszlop + j;
                if(o < 0)
                    o = szelesseg-1;
                else if(o >= szelesseg)
                    o = 0;

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

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

```

Majd ezt a függvényt felhasználjuk a következő metódusunkban, amiben az élő szomszédok számát ellenőrizzük és eszerint adunk új értéket a vizsgált cellának.

```
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
            szomszedja 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
            szomszedja van, különben élő lesz. */
            if(elok==3)
                racsUtana[i][j] = SejtAblak::ELO;
            else
                racsUtana[i][j] = SejtAblak::HALOTT;
        }
    }
    racsIndex = (racsIndex+1)%2;
}
```

Fentebb a lényegesebb részeket emeltem ki, a kód többi része megtalálható a Conway/cppeletjatek mappában: **SejtAblak** **SejtSzal**



7.4. ábra. Életjáték

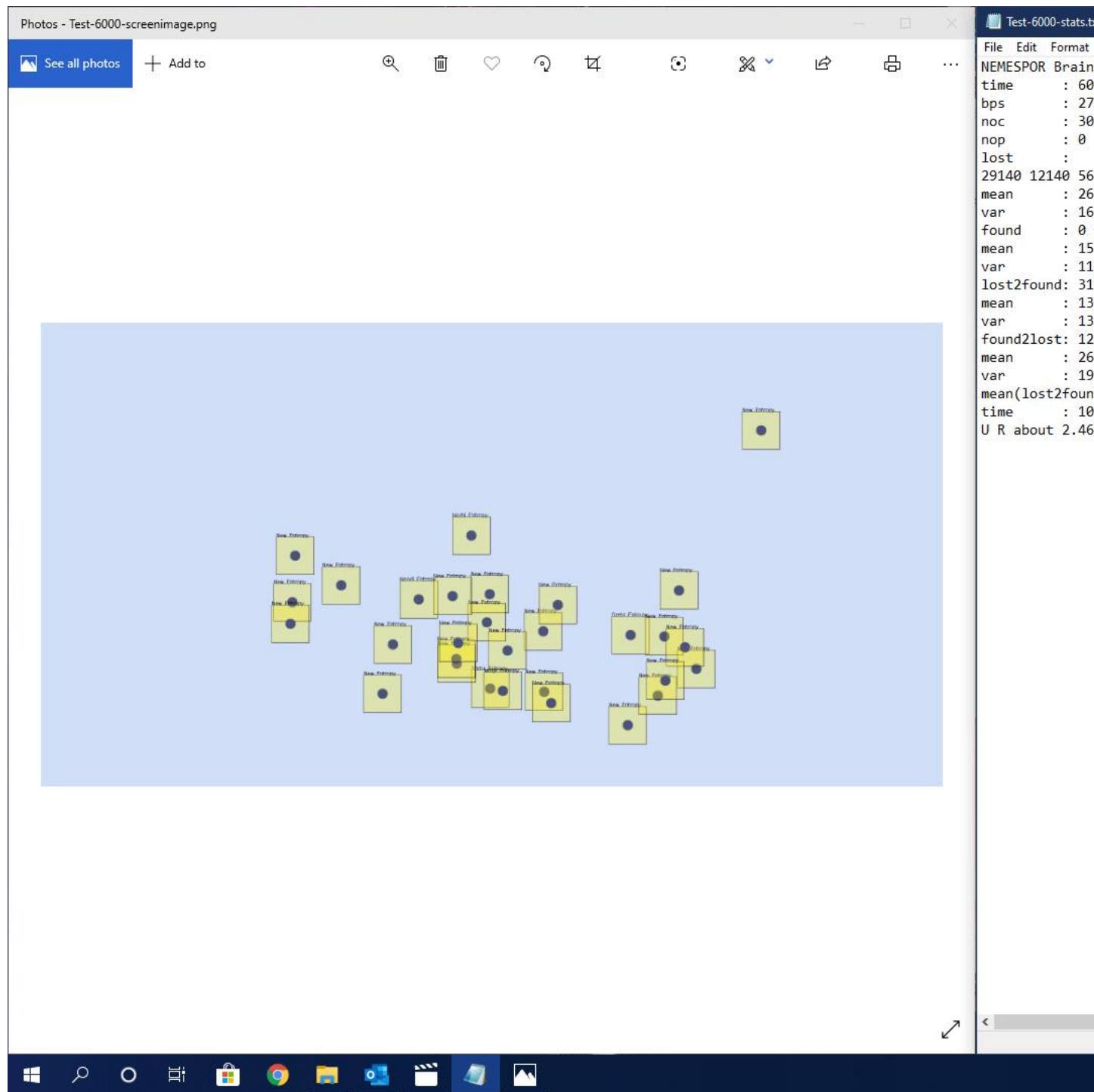
7.4. BrainB Benchmark

Megoldás videó: initial hack: <https://www.twitch.tv/videos/139186614>

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

A BrainB program tulajdonképpen az embert vizsgálja. Tapasztalhattuk már, hogy különböző játékokban, az irányított karakterünket szem elől veszítjük különböző tényezők hatására. Mint például ha még több karakter gyűlik a miénk köré, esetleg különböző effektek is takarják a láthatóságát közben. Ilyenkor nehéz követni, hogy mi hol is vagyunk, mit csinálunk éppen. A BrainB program pont ezt a képességünket méri. A programban, mint láthatjuk a lentebb lévő képünkön, sok kis négyzet között, melyeknek a közepén egy pont van, egyik a saját négyzetünk.

A mi feladatunk, hogy azt a négyzetet kövessük az egérmutatónkkel. Erre időlimit van, maximum 10 percig követhetjük a négyzetünket és közben lenyomva kell tartanunk a balegérgombunkat. Végül kapunk egy eredményt abból számítva, hogy hányszor vesztettük el a négyzetünket, mennyire tudtuk követni.



7.5. ábra. BrainB program és kimenete

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python (lehet az SMNIST [?] is a példa).

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

Megoldás forrása: <https://github.com/tensorflow/tensorflow/releases/tag/v0.9.0> (/tensorflow-0.9.0/tensorflow/examples/tutorials/mnist/),
https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol

A programunk célja, hogy a neurális hálónkat betanítsa és hogy az MNIST-ben lévő számokat fel tudja ismerni, majd ennek a hatékonyságát közli velünk. Az MNIST adatbázis egy kézzel írott számok képeiből álló adatbázis, amelyet gyakran használnak a gépi tanulás köreiben és képfeldolgozó rendszerek tanítására. A tensorflow egy gépi tanulásra készített library amelyet a google fejlesztett ki, a programunkban ezt is használni fogjuk.

```
#először is importáljuk a tensorflow könyvtárat és az mnist datesetjét betöljtük.

import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

#Felépítjük a modellünket, ez egy 28x28-as kép lesz

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])

predictions = model(x_train[:1]).numpy()
predictions

tf.nn.softmax(predictions).numpy()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

loss_fn(y_train[:1], predictions).numpy()

#Itt a program hatékonyságát számoljuk ki
```

```
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

model.evaluate(x_test, y_test, verbose=2)

probability_model = tf.keras.Sequential([
    model,
    tf.keras.layers.Softmax()
])

probability_model(x_test[:5])
```

8.2. Mély MNIST

Python (MNIST vagy SMNIST, bármelyik, amely nem a softmaxos, például lehet egy CNN-es).

Megoldás videó: https://bhaxor.blog.hu/2019/03/10/the_semantic_mnist

Megoldás forrása: SMNIST, [https://gitlab.com/nbatfai/smni](https://gitlab.com/nbatfai/smnist)st

A programunk hasonló az előző feladatban lévőhöz, itt viszont egy képet átadva a programnak, megpróbálja felismerni, hogy a képen milyen szám van.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

from PIL import Image
import numpy as np
import sys

#A következő három sort kommentbe raktam, így tudott csak lefutni a gépemen a program

#physical_devices = tf.config.experimental.list_physical_devices('GPU')
#assert len(physical_devices) > 0, "Not enough GPU hardware devices available"
#tf.config.experimental.set_memory_growth(physical_devices[0], True)

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

#A kapott képet itt újraméretezzük

x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

```
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation=tf.nn.relu))
model.add(Dropout(0.2))
model.add(Dense(10, activation=tf.nn.softmax))

tb_log_dir = "./cnn_tb"
file_writer = tf.summary.create_file_writer(tb_log_dir)
file_writer.set_as_default()
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=tb_log_dir, profile_batch=0)

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x=x_train,y=y_train, epochs=10, callbacks=[tensorboard_callback])

model.evaluate(x_test, y_test)

input_image = np.array(Image.open(sys.argv[1]).getdata(0).resize((28, 28), 0))

pred = model.predict(input_image.reshape(1, 28, 28, 1))

print (pred)

print("The number is = ", pred.argmax())
```

8.3. Minecraft-MALMÖ

Most, hogy már van némi ágensprogramozási gyakorlatod, adj egy rövid általános áttekintést a MALMÖ projektről!

Megoldás videó: initial hack: <https://youtu.be/bAPSu3Rndl8>. Red Flower Hell: <https://github.com/nbatfai/RedFlowerHell>.

Megoldás forrása:

A Malmö Projekt egy minecraft mod, amit a Microsoft készített. Itt célnunk kódokkal irányítani a karakterünket, ezzel a mesterséges intelligencia felé próbálunk közelíteni. A kódjainkban igyekszünk úgy irányítani a karakterünket, hogy különböző szituációkkor különféleképpen cselekedjen. Az egyetemen a mi feladatunk az volt, hogy egy legenerált pályán Steve (a minecraft karakterünk) járja körbe az arénát, érzékelje a környezetét és keressen, majd szedjen fel minél több virágot akadálymentesen mielőtt az arénánk oldalain lefolyó lávával találkozik, tehát ha például csapdába esik, akkor abból szabaduljon ki, ha virág van a közelében, akkor úgy lépjen, hogy azt fel tudja venni, stb.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó: <https://youtu.be/z6NJE2a1zIA>

Megoldás forrása:

9.2. Gimp Scheme Script-fu: króm effekt

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

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

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

Tanulságok, tapasztalatok, magyarázat...

9.3. Gimp Scheme Script-fu: név mandala

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

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

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

Tanulságok, tapasztalatok, magyarázat...

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

[?]

1.2 Alapfogalmak

A számítógépes programozás három nyelvét különböztethetjük meg: gépi nyelv, assembly szintű nyelv, vagy magas szintű nyelv. Egy magas szintű programnyelvet a szintaktikai és szemantikai szabályai határoznak meg, és jelen kell lennie mindeneknek. A szintaktika a formai szabályok, míg a szemtanika a tartalmi, jelentésbeli szabályok együttese. Ahhoz, hogy egy magas szintű programozási nyelven megírt programot le tudunk futtatni, azt előbb le kell fordítani a processzor saját gépi nyelvére. Erre kétféle lehetőségünk van, a fordítóprogramos és az interpreteres.

A fordítóprogram a magas szintű nyelvű forrásból egy gépi kódú tárgyprogramot kreál négy lépében. Lexikális elemzés, szintaktikai elemzés, szemantikai elemzés, majd kódgenerálás. Amennyiben az adott nyelv szabályai szerint szintaktikailag helyes, az elkészült tárgyprogramból a kapcsolatszerkesztő állít elő egy már ténylegesen futtatható programot.

Az interpreteres megoldás esetén is megvan az első három lépés, de itt nem készül tárgyprogram. Helyette utasításonként sorra veszi a forrásprogramot, értelmezi, majd végre is hajtja azt.

A programnyelvek lehetnek vagy fordítóprogramosak, vagy interpreteresek is, de lehetnek egyszerre mindkettő is.

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

Alapismeretek

Az első fejezet, mint bevezetés, írat velünk egy egyszerű helloworld programot, ezzel megtanítva, egyszersint begyakorolhatva velünk az alapvető szintaktikát, valamint a fordítás és futtatás mikéntjét. A továbbiakban új alapfogalmakat tanít, mint hogy a C-beli programok .c-re végeződnek, vagy épp hogy mi az a változó és milyen típusai vannak. Ezután találkozhatunk még olyan fontos elemekkel, mint a ciklusok, feltételes elágazások, tömbök és függvények. Az olvasót végig arra próbálják sarkallni, hogy minél többet gyakoroljon.

Típusok, operátorok és kifejezések

A második fejezetben olvashatunk bővebben a változókról, például a nevükhez tartozó konvenciókról, a típusokról és azok méréteiről. Bővebb szó esik az értékadásról, megjelennek a konstansok. Érdemben tárgyal az aritmetikai, relációs és logikai operátorokról. Típuskonverziók.

10.3. Programozás

[BMECPP]

Ebben a kis olvasmányban megismerhettük a C++ és elődje, a C közötti néhány különbséget. C-vel ellentétben C++-ban az üres paraméterlista void visszatérési értéket jelent. Szintén lényeges különbséget jelent, hogy már nem kötelező a return használata, azt a fordítóprogram automatikusan a végére teszi. C++-ban megjelent a bool típusú változó, ami logikai igaz/hamis értékekkel dolgozik. Ebből kifolyólag a nyelv kulcsszavai részévé váltak a bool, true és false kifejezések.

10.4. Mobilprogramozás

[?]

2.3. Python

A Python egy magas szintű, általános célú programozási nyelv melyet a szkriptnyelvek családjába szoktak sorolni. A Pythonban megírt programokat általában valamilyen fejlesztői környezetben tudjuk futtatni. Egyik nagy előnye, hogy rendelkezik egy rendkívül hasznos alapkönyvtárral, mely számos praktikus segítséget tartalmaz, ilyenek például a szabványos kifejezések. Másik nagy előnye a bővíthetőség, C-ben vagy C++-ban írt programrészekkel is kompatibilis tud lenni, így azokkal könnyen bővíteni lehet a kódot. Népszerűsége egyszerűségéből fakad. Fontos hátulütője viszont, hogy a legtöbb mobil alapvetően nem rendelkezik Python-futtató-környezettel, így hiába egyszerűbb megírni egy programot, nehezebb terjeszteni azt. Ha viszont egy mobil rendelkezik Python-futtató-környezettel, akkor lehetővé teszi azt is, hogy mobilon programozzunk és teszteljünk is, bár előbbi nem ajánlott hosszabb programkódok esetén.

III. rész

Második felvonás

Bátf41 Haxor Stream

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

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

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

11.4. C

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

11.5. C++

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

11.6. Lisp

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

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

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