

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

Ed. BHAX, DEBRECEN,
2019. február 19, v. 0.0.6

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i>		
	Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert ÁCs Szűcs, Gergő	2019. október 2.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai
0.0.5	2019-04-24	Első draft kész.	SZG
0.0.6	2019-05-06	Minden fájl feltöltve github repoba: Könyv: https://gitlab.com/gergoszka/bhax_textbook Source fájlok: https://gitlab.com/gergoszka/konyv	SZG

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	7
2.4. Labdapattogás	8
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	10
2.6. Helló, Google!	10
2.7. 100 éves a Brun téTEL	12
2.8. A Monty Hall probléma	12
3. Helló, Chomsky!	14
3.1. Decimálisból unárisba átváltó Turing gép	14
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	14
3.3. Hivatalos nyelv	15
3.4. Saját lexikális elemző	16
3.5. Leetspeak	17
3.6. A források olvasása	19
3.7. Logikus	21
3.8. Deklaráció	21

4. Helló, Caesar!	25
4.1. double ** háromszögmátrix	25
4.2. C EXOR titkosító	26
4.3. Java EXOR titkosító	27
4.4. C EXOR törő	28
4.5. Neurális OR, AND és EXOR kapu	28
4.6. Hiba-visszaterjesztéses perceptron	29
5. Helló, Mandelbrot!	30
5.1. A Mandelbrot halmaz	30
5.2. A Mandelbrot halmaz a std::complex osztállyal	31
5.3. Biomorfok	33
5.4. A Mandelbrot halmaz CUDA megvalósítása	35
5.5. Mandelbrot nagyító és utazó C++ nyelven	35
5.6. Mandelbrot nagyító és utazó Java nyelven	36
6. Helló, Welch!	37
6.1. Első osztályom	37
6.2. LZW	38
6.3. Fabejárás	38
6.4. Tag a gyökér	39
6.5. Mutató a gyökér	40
6.6. Mozgató szemantika	40
7. Helló, Conway!	42
7.1. Hangyszimulációk	42
7.2. Java életjáték	43
7.3. Qt C++ életjáték	43
7.4. BrainB Benchmark	44
8. Helló, Schwarzenegger!	46
8.1. Szoftmax Py MNIST	46
8.2. Mély MNIST	47
8.3. Minecraft-MALMÖ	47

9. Helló, Chaitin!	48
9.1. Iteratív és rekurzív faktoriális Lisp-ben	48
9.2. Gimp Scheme Script-fu: króm effekt	49
9.3. Gimp Scheme Script-fu: név mandala	49
10. Helló, Gutenberg!	50
10.1. Programozási alapfogalmak	50
10.2. Programozás bevezetés	52
10.3. Programozás	53
III. Második felvonás	55
11. Helló, Berners-Lee!	57
11.1. Python	57
11.2. C++ és Java összehasonlítása	57
12. Helló, Arroway!	59
12.1. OO szemlélet	59
12.2. Homokatózó	61
12.3. „Gagyi”	61
12.4. Yoda	62
12.5. Kódolás from scratch	62
13. Helló, Liskov!	64
13.1. Liskov helyettesítés sértése	64
13.2. Szülő-gyerek	65
13.3. Anti OO	66
13.4. Ciklomatikus komplexitás	68
IV. Irodalomjegyzék	70
13.5. Általános	71
13.6. C	71
13.7. C++	71
13.8. Lisp	71

Ábrák jegyzéke

5.1. A Mandelbrot halmaz a komplex síkon	30
--	----

Táblázatok jegyzéke

13.1. Mérési eredmény	66
---------------------------------	----

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás videó:

Megoldás forrása:https://gitlab.com/gergoszka/konyv/tree/master/1_Turing

Tanulságok, tapasztalatok, magyarázat...

Végtelen ciklusok véletlen és direkt is létre jöhetnek. Direkt alkalmaznak végtelen ciklust például a programok futása közben, hiszen nincsen előre megadott kilépési feltétel, de a bezárás gomb megszakítja a ciklust.

Egy szál 100%-on: A for(;;) és a while(1) is olyan végtelen ciklust hoz létre amely egy magot 100%-on használ ki, gyakorlatban ha célunk egy végtelen ciklus a for(;;) használatos, ez ugyanis egyértelművé teszi más programozók számára mit akartunk vele elérni és nem figyelmetlenségből van ott.

```
int main ()
{
    while(int i=1)
        for(;;

    return 0;
}
```

Egy szál 0%-on: 0%-os CPU használtságot a sleep(0); parancs segítségevel tudunk elérni. A 0 végtelen ideig "altatja" a folyamatot, ezt másodpercben megadva használjuk.

Több szál 100%-on: Ehhez párhuzmosítanunk kell a programunkat, amit a **#pragma omp parallel** használatával érhetünk el. Igy a program minden szálat 100%ban ki tud majd használni.

Fordítás: **gcc infinite_p100 -o infinite_p100 -fopenmp**

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

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

Megoldás videó:

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

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

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

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

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

main(Input Q)
{
    Lefagy2(Q)
}

}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

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

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

Tanulságok, tapasztalatok, magyarázat...

T100nak ha saját magát adjuk meg azt írja ki hogy nincs benne végtelen ciklus pedig az argumentuma maga egy végtelen ciklus

A T1000nek pedig nem tudja eldönteni magáról hogy micsoda ugyanis ha nem fagy le bekerül egy végtelen ciklusba amitől lefagy.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/1_Turing

Tanulságok, tapasztalatok, magyarázat...

Csere segédváltozóval: Itt létrehozunk egy ideiglenes változót amiben eltároljuk az első változó értékét, majd az első változót egyenlővé tesszük a másodikat. Ezután a másodikat egyenlővé teszzük az ideiglenes változóban lévő értékkel

```
#include <stdio.h>
int main()
{
    int a=42, b=666;

    int c=a;
    a=b;
    b=c;

    return 0;
}
```

Ezen kívül lehetséges változó csere kivonás-összeadással és EXOr-ral is: //!!!

```
a = a - b; //Csere összeadás-kivonással
b = a + b;
a = b - a;

a = a ^ b; //Csere EXOR-al
b = a ^ b;
a = a ^ b;
```

2.4. Labdapattogás

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

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

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/1_Turing

Tanulságok, tapasztalatok, magyarázat...

If-el: x és y lesz a pálya mérete, lx és ly a labda helyzete, mx és my pedig a labda mozgásához kell majd. A labda pattogását egy végtelen ciklus és a kiir eljárás teszi lehetővé. Először növelem a labda jelenlegi helyzetét mx és my-al majd megnézem, hogy elérte e valamelyik falat és ha igen megfordítom a mozgatás előjelét. Késlelteni kell a kiirást amit a usleep-pel érek el.

```
int main(void)
{
    int x=90,y=10, //pálya mérete
        lx=1,ly=1, //labda helyzete
        mx=1,my=1; //labda mozgatása
```

```
char labda='o';

for(;;)
{
    lx+=mx;
    ly+=my;

    if (x-1<=lx) {
        mx*=-1;
    }

    if (y<ly) {
        my*=-1;
    }

    if (lx<0) {
        mx*=-1;
    }

    if (ly<0) {
        my*=-1;
    }

    kiir(lx,ly,labda);
    usleep (100000);

}
}
```

If nélkül: Itt a labda pattogását maradékos osztással és abszolút értékkel érjük el. Plus szükség van egy változóra amit mindenkorán többet növeljük majd szorozzuk a pálya méretének kétszeresével, ezt pedig kivonjuk a pálya magasságából/szélességből. Igy megkapjuk a labda kordinátáját és ha a változó nagyobb lesz a pályánál a kivonás miatt az abszolút értéke újra kicsi lesz.

```
x=abs(szelesseg-lepteto%(2*szelesseg));
y=abs(tmagassag-lepteto%(2*tmagassag));
rajzol(palya,x,y,labda);
lepteto++;
```

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/1_Turing

Tanulságok, tapasztalatok, magyarázat...

A gépi szóhossz méretét a bitshift operátorral tudjuk megnézni. Mi a leftshifet alkalmazzuk ami a megadott szám bináris értékében az egyeseket balra tolja és pótolja a nullákat. Igy az egyet szépen elkezdjük tologatni balra és számoljuk hogy hányszor mozdítottuk el amíg elérjük az int végét és a ciklus leáll. A végen a változó érteke 31 lesz, de a szóhossz mégis 32 mivel az elsőt nem számolta.

```
int szam = 1, a = 0;

while (szam <=1) {
    a++;
}

printf("%u bites a gépi szó \n", a+1);
```

2.6. Helló, Google!

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

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/1_Turing

Tanulságok, tapasztalatok, magyarázat...

A pagerankot a Google fejlesztette ki és arral szolgál, hogy megmutassa egy weboldal "értékét". Ezt az oldalra mutató linkek alapján számolják ki, minnél nagyobb Pagerankú oldal mutat a te oldaladra annan több "pontot" ad. Az alap elgondolás az volt, hogy emberek azért linkelnek be más oldalakat mert hasznosak.

Manapság már nem tölt be olyan fontos szerepet a google algoritmusában mert linkfarmokkal és fórumokon komment spameléssel manipulálható volt.

Létrehozzuk a kapcsoltati gráfot ami megmutatja hogy melyik link melyik oldalra mutat(halott linkek kezelésétől eltekintük)

```
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};
    4.0};
}
```

Ezután elindítunk egy végtelen ciklust ami akkor fejeződik be ha a pagerank kisebb lesz mint a csillapító tényező(damping factor, jelen esetben 0.00001). Ebben csinálunk egy for ciklust ami a kapcsolati gráfot tömb sorait megszorozza a pagerankkal, az értéket pedig betölti egy ideiglenes pagerankba, amíg ki nem lép a végtelen ciklusból.

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

for (i=0;i<4;i++) {
    double temp=0;
    for (j=0;j<4;j++)
        temp+=L[i][j]*PR[j];
    PRv[i]=temp;
}

if ( tavolsag(PR,PRv, 4) < 0.00001)
    break;

}
```

Ennek a feltétele pedig az hogy a tavolság függvény visszatérési értékenek kisebbnek kell lennie mint a csillapító tényező. Ez a függvény argumentumként megkapja a végleges és ideiglenes pagerankot és az oldalak számát, majd a két tömb i-edik elemének az értéket és ezek abszolút értékét összeadja.

```
double tavolsag(double pr[], double pr_temp[], int n)
{
    double osszeg= 0;
    for(int i=0; i<n; i++)
        osszeg += (pr_temp[i]-pr[i])*(pr_temp[i]-pr[i]);

    return osszeg;
}
```

2.7. 100 éves a Brun téTEL

Írj R szimulációt a Brun téTEL demonstrálására!

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

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

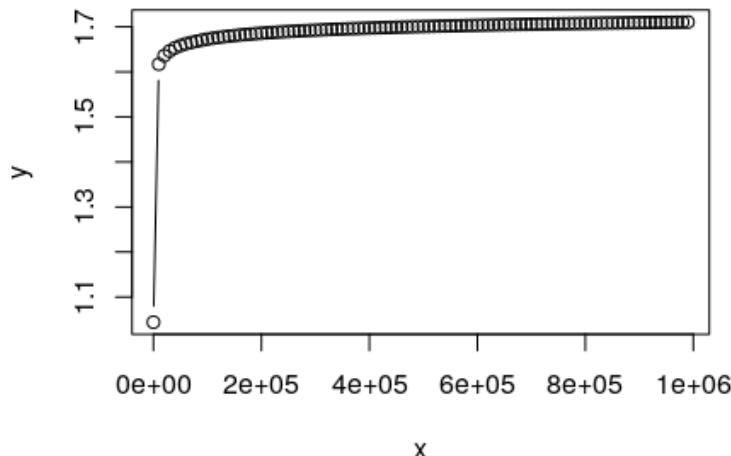
A prímek olyan számok amelyeknek csak két osztójuk van: 1 és saját maguk. Az ikerprímek pedig olyan prímek melyek különbsége kettő (pl:5 és 7). Így nézzünk egy programot ami kiszámol és ábrázol x prím között található ikerprímet.

```
primes = primes(x)
diff = primes[2:length(primes)]-primes[1:length(primes)-1]
idx = which(diff==2)
t1primes = primes[idx]
t2primes = primes[idx]+2
```

A program egy megadott x értékig kikeresi a prímekeket. Majd megnézi a köztük lévő differenciát (diff), ahol ez a differencia 2, annak az indexét egy tömbben(idx) tárolja (de csak az ikerprímpár első tagjának indexét, ezért kell a t2primes-nál a +2) tehát a prímek közül kiszűri, hogy melyek ikerprímek.

Az stp függvényben a megadott x-ig kiszedi az összes prímet és ha a különbségük 2 (diff==2), annak az indexét az idx-ben tárolja, de csak az első tagját így csinálunk két tömböt az első és második tagnak (t1 és t2primes). Az rt1plus2 összeadja ezek reciprokát, majd ennek az összegét adja vissza a függvény.

A Seq-el beosztjuk az x tengelyt (13-tól 1000000.ig, 10000-es lépésszámmal). A sapply pedig az y értékekhez rendeli a visszatérési értékeket. Pláttal pedig kirajzoljuk az ábrát.



2.8. A Monty Hall probléMA

Írj R szimulációt a Monty Hall problémára!

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

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

Tanulságok, tapasztalatok, magyarázat...

A Monty hall probléma egy statisztikai puzzle:

- Van 3 ajtó, 2 mögött egy-egy kecske, a harmadik mögött pedig egy autó.
- Te választassz egy ajtót (hívjuk A ajtónak). Természetesen te azt szeretnéd választani mi mögött az autó van.
- A játékmester megnézi a másik két ajtót (B és C) és kinyit egyet ami mögöt egy kecske van.

A játék a következő: Maradsz az eredeti ajtónál vagy átváltasz a másik zártra? Az ember először azt hinné, hogy minden esetben 50-50% a nyeremény esélye, valójában ha váltassz az esetek 2/3-ában nyersz! A kulcs az indoráció mennyiségeiben van: az elején semmit nem tudunk a az ajtókról, így az esélye hogy az auto az egyik ajtó mögött van 1/3, ha maradok az első választásomon ez ennyi marad, nem lehet sehogy a győzelmem esélye. E szerint a logika szerint a másik ajtónál kell legyen a maradék 2/3 esély. Sokkal jobban látszik a lényeg ha az ajtók mennyiségeit 3-ról 100-ra emeljük. Ugyanúgy kiválsztunk egy ajtót, a játékvezető pedig kinyit 98-at ami mögött biztos kecske van. Most maradsz-e az eredeti ajtódnál (1/100) vagy a legjobb ajtót választod a maradék 99 közül ?

Ha a statisztikai magyarázat nem győz meg, egy R szimulációban kiszámoljuk melyik a jobb választás:

A kísérletek_száma változóban adjuk meg a próbálkozások számát, azaz hogy hányszor fusson le a szimuláció. A kísérlet és a játékos tömböket feltöljük 1 és 3 közötti számokkal. A musorvezető egy vektor aminek a merete megegyezik a kísérletek számával

Egy for ciklussal bejárjuk a kísérletek_szamat és ha a játékos jól tippelt, akkor a mibol tömbbe a másik két ajtó kerül. Ezután Monty 'megtámaszt' egy ajtót, vagyis kiválaszt egyet a mibol tömbből. Aztán megnézzük hányszor nyerne a játékos ha az eredeti választásánál marad. Vagyis ide azok az elemek kerülnek amikor a játékos és a kísérlet azonos. Végül kiíratjuk a statisztikát, hogy mikor járnánk jobban ha minden változatnánk vagy ha maradnánk az első ajtónál.

3. fejezet

Helló, Chomsky!

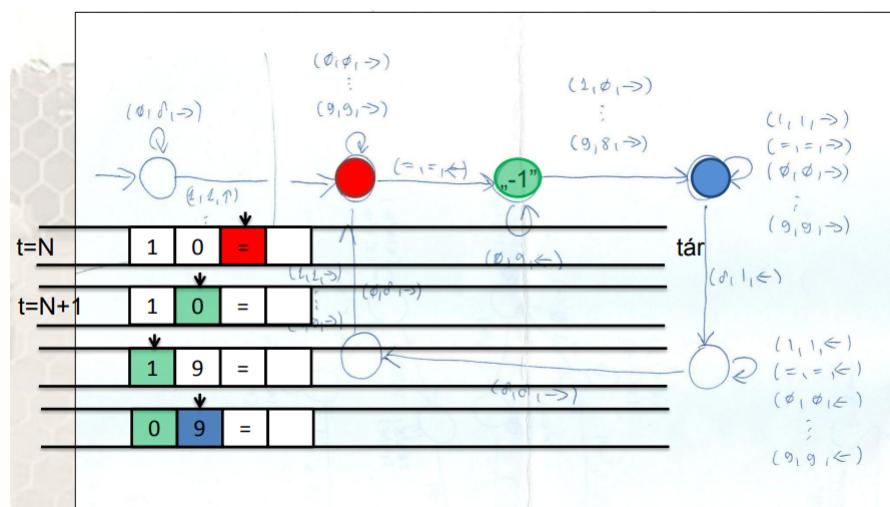
3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_1.pdf?fbclid=IwAR2AZ

Tanulságok, tapasztalatok, magyarázat...



Az unáris a legegyszerűbb számrendszer, mivel a számokat úgy jelöljük, hogy annyi egyest vagy vonalat írnunk amennyi a szám értéke (10 esetén 10 db vonalat, 50 esetén 50-et, stb...). Unárisban csak pozitív számokat tudunk ábrázolni illetve a nullát úgy jelöljük, hogy nem írnunk semmit.

Így decimálisból unárisba átváltani egy for ciklus segítségével a legegyszerűbb, amit 0-tól indítunk a kívánt szám értékéig és minden iterációban egy egyest szűrünk egy tömb végére majd kiíratjuk.

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

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Forrás: <https://gyires.inf.unideb.hu/KMITT/b24/ch03s02.html>

A környezetfüggő grammaтика olyan szabályok összessége, amely segítségével a nyelvben minden jelsorozatot képesek vagyunk előállítani.

I. $N = (S, X)$, $T = (a, b, c)$, ahol N nem terminális, T pedig terminális szimbólumok. A nyelvtan szabályai:

- 1. $S \rightarrow abc$
- 2. $S \rightarrow aXSc$
- 3. $Xb \rightarrow bb$
- 4. $Xa \rightarrow aX$

És ebből egy generált $a^2b^2c^2$ nyelv.

$S \rightarrow aXSc \rightarrow aXabcc \rightarrow aaXbcc \rightarrow aabbcc$

II. $N = (S, X, Y)$, $T = (a, b, c)$

- 1. $S \rightarrow abc$
- 2. $S \rightarrow aXbc$
- 3. $Xb \rightarrow bX$
- 4. $Xc \rightarrow Ybcc$
- 5. $bY \rightarrow Yb$
- 6. $aY \rightarrow aaX$
- 7. $aY \rightarrow aa$

És ebből egy generált $a^3b^3c^3$ nyelv.

$S \rightarrow aXbc \rightarrow abXc \rightarrow abYbcc \rightarrow aYbcc \rightarrow aaXbcc \rightarrow aabbXcc \rightarrow aabbYbcc \rightarrow aaYbbbccc \rightarrow aaabbccc$

3.3. Hivatkozási nyelv

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

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/2_Chomsky

Tanulságok, tapasztalatok, magyarázat...

C-ben az utasítások egymás után leforduló parancsok, a nyelv alapjai. Tartalmaz alap egysoros utasításokat, többsoros utasítás blokkokat, iterációkat(for,while,do-while), operátorokat(--,++,!=,stb...) és vezérlő szerkezeteket(if,switch) is.

Váltani a két változat között a következő módon lehet: **gcc -std="c99/89" hivatkozas.c-**

A következő kód részlet csak c99 alatt fordul le mivel c89-ben még nem lehetett for cikluson belül változót deklarálni.

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

int main()
{
    for(int i=0; i<5; i++) {
        printf("\n");
    };

    return 0;
}
```

3.4. Saját lexikális elemző

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

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

Megoldás forrása: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/realnumber.l](https://bhax.io/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/realnumber.l)

```
#include <stdio.h>
int realnumbers = 0;
%%
digit [0-9]
%%
{digit}*(\.{digit}+)? {++realnumbers;
    printf("[realnum=%s %f]", yytext, atof(yytext));}
%%
int
main ()
{
    yylex ();
```

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

Tanulságok, tapasztalatok, magyarázat...

LEXben az első rész tartalmazza a dekralációkat és a könyvtárakat és ennek a végén definiálunk és adjuk meg a karaktereket amiket keresni akarunk a szövegből. Mi itt megadjuk hogy számokat keresünk 0-9ig. A részeket a %% jelek választják el, ezután jön a második rész.

Itt megmondjuk mit csinálunk a karakterekkel ha megtalálja őket a lexer. Ha valós számot talál növeljük a számláló értéket és megmondjuk neki hogy írja ki a karaktersorozatot.

Ez után jön a main rész ahol meghívjuk az yylex() függvényt és kiírjuk hany számot találtunk.

Fordítás:

```
lex -o lexing.c lexing.l
```

```
gcc .o lexing lexing.c -lfl
```

Ezután elkezdhetünk számokat irni a terminálba, ha meguntuk pedig a CTRL+D-vel állíthatjuk meg a programot.

3.5. Leetspeak

Lexelj össze egy l33t cipher!

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

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/l337d1c7.1

```
% {
#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', {"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  
};  
  
%}  
%%  
. {  
  
    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", 1337d1c7[i].leet[1]);
    else if(r<98)
        printf("%s", 1337d1c7[i].leet[2]);
    else
        printf("%s", 1337d1c7[i].leet[3]);

    found = 1;
    break;
}

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

}
%%

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

Tanulságok, tapasztalatok, magyarázat...

A leet, vagy leetspeak egy régen gamerek és hackerek által használt online dialektus, aminek megvannak a saját szlengjei és írásmódja.

Programunkban először definoljuk a lexer struktúra méretét, a L337SIZE-t ,amivel számoljuk majd a sorok számát. Majd a chiper struktúrában létrehozunk egy char változót és egy négy elemű dinamikus tömböt.

A 1337d1c7 tömbben pedig tároljuk minden karakterre a lehetséges cseréket, amik közül később majd választunk. Ezután deklarálunk egy found változót amivel követjük, hogy megtaláltuk e a karakter a tömbben. Majd elidítünk egy for ciklust ami megkeresi a struktúrában a beolvastott karaktert. Ha megtalálta, a 1337d1c7 tömb négy lehetősége között random választunk egyet és azt adjuk vissza.

A main részben az srand növeli a számok randomitását és meghívjuk a yylex()-et ami elvégzi a leet szöveggé alakítást.

3.6. A források olvasása

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

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

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

Bugok

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

i.

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

ii.

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

For ciklus ami nullától ötig megy, a `++i` pre-incrementet jelent, vagyis a ciklus lefutása előtt növeli a változó értékét

iii.

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

For ciklus ami nullától ötig megy, a `i++` post-incrementet jelent, vagyis a ciklus lefutása után növeli a változó értékét

iv.

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

For ciklus ami nullától ötig megy, tömb i-edik eleme egyenlő lesz a léptetővel.

v.

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

For cílus ami addig megy mig i kisebb mint n, és a d tömbmutatót kicseréli arra amire az s mutat.

vi.

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

Kiirja az f függvény eredményét kétszer, csak más argumentumokkal és így vélhetőleg más eredménnyel.

vii.

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

Kiirja az 'a' változót az f fügvénnyel való módosítás után és változtatás nélkül.

viii.

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

Kiirná az 'a' pointert és az 'a' változót, de pointereket nem `%d`-vel hanem `%p`-vel kel kiiratni.

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/2_Chomsky

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat...

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

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

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

Tanulságok, tapasztalatok, magyarázat...

1. Bármely x-hez létezik olyan y, hogy az x kisebb mint az y és y prím.
2. Bármely x-hez létezik olyan y, hogy az x kisebb mint az y és y prím és y+2 is prím.
3. Létezik olyan y, hogy bármely x esetén, ha x prím akkor az x kisebb mint y.
4. Létezik olyan y, hogy bármely x esetén,, ha y kisebb mint x akkor x nem prím.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

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

Mit vezetnek be a programba a következő nevek?

- `int a;`
egész változó
- `int *b = &a;`
egész típusú változóra mutató mutató
- `int &r = a;`
az a változó memóriacímét r-ben tároljuk el
- `int c[5];`
5 elemű egészekből álló tömb
- `int (&tr)[5] = c;`
tr hivatkozik az egész c tömbre
- `int *d[5];`
d egy 5 elemű mutatókból álló tömb
- `int *h();`
h egy függvény aminek a visszatérési értéke egy egészre mutató mutató.
- `int *(*l)();`
l egy függvény aminek a visszatérési értéke egy egészre mutató mutatóra mutató
- `int (*v(int c))(int a, int b)`
Egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény
- `int (*(*z)(int))(int, int);`
Függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/2_Chomsky

Az utolsó két deklarációs példa demonstrálására két olyan kódot írtunk, amelyek összahasonlítása azt mutatja meg, hogy miért érdemes a **typedef** használata: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr.c](https://gitlab.com/gergoszka/konyv/tree/master/bhax_thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr.c), [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr2.c](https://gitlab.com/gergoszka/konyv/tree/master/bhax_thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/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;
}
```

Tanulságok, tapasztalatok, magyarázat...

4. fejezet

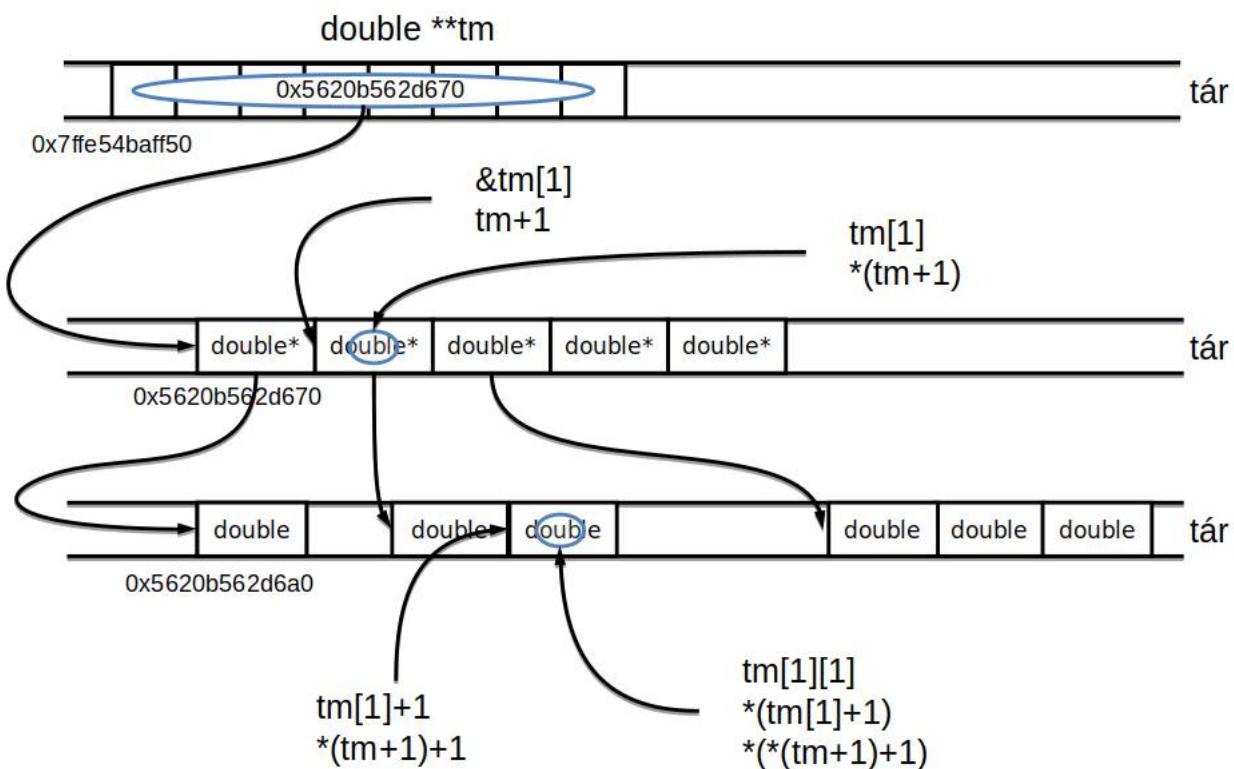
Helló, Caesar!

4.1. double ** háromszögmátrix

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/3_Caesar <https://bit.ly/2VaEcL>

Tanulságok, tapasztalatok, magyarázat...



Az alsó hároszög mátrix egy olyan kvadratikus (megegyező sor- és oszlopszámú) mátrix aminek főátlója fölött csak 0-ák szerepelnek. Az alsó háromszögmátrixokat sorfolyatonosan egy vektorban szoktuk tárolni. nr-ben megadjuk a mátrix sor és oszlop számát és egy double*ra mutatót. Az ifben double* nr-szeresát foglaljuk le, ha a malloc nem tud helyet foglalni akkor -1-es hibaüzenettel tér vissza. Majd a következő malloc-kal a sor elemeinek foglalunk helyet, mivel a double 8 bájtos az első sorban 1x8 majd következőben 2x8 és így tovább bájtot foglalunk le.

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

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

Majd egy for ciklus feltölti a tömböt 0-14ig számokkal, egy másik pedig a megadott alsó háromszög mátrix formában kiírja a terminálba.

```
greg@greg-X510UAR:~/Prog1$ gcc haromszog_matrix.c
greg@greg-X510UAR:~/Prog1$ ./a.out
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
6.000000, 7.000000, 8.000000, 9.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
```

Végül a free(tm)-el felszabadítjuk a malloc által lefoglalt helyet.

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/3_Caesar

Tanulságok, tapasztalatok, magyarázat...

Az exor titkosítás egy egyszerű, de hasznos titkosítási forma, aminek a biztonsága a megadott kulcs hosszával növekszik, mivel a feltörésnél annál több lehetőséget kell majd megnézni.

Mivel a szöveg minden karakteren xor-t hajtunk végre, az adott karaktert a kulcs adott karakterével xorozzunk, de mivel a kulcs rövidebb lesz mint a titkosítandó szöveg egy idő után elkezd ismétlődni.

A programban a while beolvassa a szövegfájlt, majd a for ciklus minden egyes karakter titkosít exorral, ha nagyob a szöveg mint a kulcs ismétlődésével titkosít.

```
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;
    }
}
```

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/3_Caesar

Tanulságok, tapasztalatok, magyarázat...

Mivel javabon programotunk kihaszáljuk a objektum orientált programozás előnyeit és készízünk is egy Exor osztályt. Ez egy kicsit overkill és lassabb is mint a c változat de mérföldekkel jobban olvasható a kódja. Egy stringbe fokjuk majd a kulcsot tárolni, ezen felül csinálunk kát csatornát az I/O kapcsolatoknak is. Hibakezelünk is, a throws elkapja ha valami hiba törtnéne a beolvasásnál vagy kiíratásnál. Majd csinálunk két byte tömböt a kulcsnak és a buffernek. A while ciklusban beolvassuk a szöveget és tároljuk a méretét az olvasottBájtok változóban. A for-ban ezután történik maga a titkosítás amit maradékos osztással végzünk. A titkosítás végeztével pedig a writeal kiírjuk a buffer tartalmát.

```
public class Exor {  
  
    public Exor(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 olvasottBájtok = 0;  
  
        while((olvasottBájtok =  
               bejövőCsatorna.read(buffer)) != -1) {  
  
            for(int i=0; i<olvasottBájtok; ++i) {  
  
                buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]);  
                kulcsIndex = (kulcsIndex+1) % kulcs.length;  
            }  
  
            kimenőCsatorna.write(buffer, 0, olvasottBájtok);  
        }  
    }  
}
```

4.4. C EXOR törő

Ebben a feladatban tutorált Nagy László Mihály!

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

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/3_Caesar

Tanulságok, tapasztalatok, magyarázat...

Az exor törő lényegében egy brute force variáns, ami végig próbálja az összes lehetséges kódolási módszert. Ez a kódoló tudatában könnyebb, de a valóságban sokkal több változót figyelembe kell venni, így sokáig tart a törés és e miatt nem is olyan elterjedt.

Itt annak eldöntésére, hogy a feltört szöveg megfelelő e, az átlagos szóhosszt (ami a szöveg hossza osztva a spacek számával) és a magyar mondatokban gyakran előforduló szavakat használjuk.

Meghívjuk az exor eljárást aztén a szöveget megvizsgáljuk ,ha passzol a akkor a for ciklusban az if teljesül és kiíratjuk standard outputra a feltört szöveget.

Majd a main részben a while ciklusban meghívjuk a titkos szöveget, majd az ezt követő for-ban nullázzuk a maradék buffert. Ezután jöhet a kulcs megtalálására szolgáló egymásba ágyazott for ciklusok amik a törést végzik és ha megtalláják kiköpik az eredményt.

Majd párhuzamosítjuk a for ciklust a gyorsabb feltörés érdekében:

```
#pragma omp parallel for
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)
                .
                .
                .
```

Így a program fordítása:

```
gcc exortör.c -fopenmp -O3 -o exortör
```

./exortör titkos szöveg.txt

4.5. Neurális OR, AND és EXOR kapu

R

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

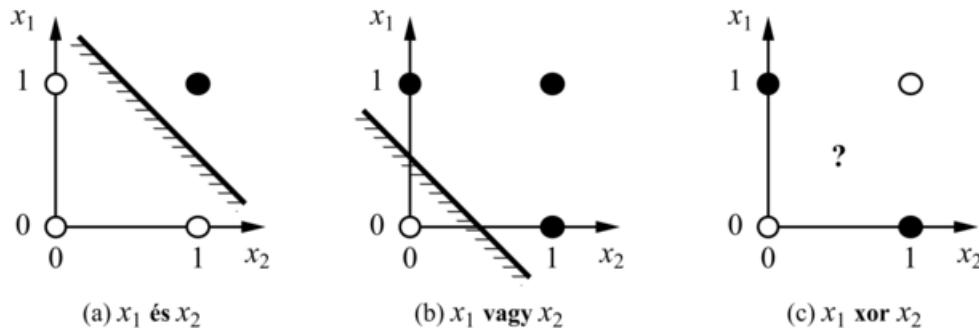
Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/NN_R http://project.mit.bme.hu/~mi_almanach/books/aima/ch20s05

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban tutoráltam Ranyhoczki Mariann-t!

Az AND és OR kapuk egyszerű perceptronként működnek. Ugyanis adatként megkapját az 1 vagy 0 bitjüket és, hogy milyen eredményt kell elérniük. Kb 100 próbálkozás után az algoritmus megtalálja a megfelelő súly értékeit és megközelítőleg jó eredményt ad, de az érdekességek az exornál kezdődnek.

Itt ugyanis változtatás nélkül nem tud választ adni a program, nem tud "vonalaat húzni" a két csoport között.



Ennek a megoldása rejtett neuronok bevezetése, amik lényegében egy új rétegnyi számolást tesz lehetővé, számuk növelése csökkenti a próbálkozások számát is.

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/4_Mandelbrot

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban tutoráltam Ranyhoczki Mariann-t és Nagy László Mihályt!

Egy perceptron a gépi tanulás egyik legegyszerűbb megvalósítása. Az elképzelés szerint egyetlen neuront ábrázolunk, ami véges sok input alapján egy algoritmus segítségével két csoportba sorolja. Ehhez nemcsak a bemenő adatokat adjuk meg, hanem a megoldást is, így az algoritmus az adatokhoz rendelt súlyok változtatásával megpróbálja megközelíteni az elvárt megoldást. Maga az algoritmus általában a szignum függvény ami az adatok és súlyaik szorzatának összegét egy -1 és 1 közötti tört számként ábrázolja. Logikusan egy elem akkor tartozik az egyik csoportba ha 0-nál nagyobb, a másikba ha kisebb. A program legelső próbálkozása alkalmával random súlyokat választ és az alapján, hogy a megoldása helyes vagy helytelen, csökkenti vagy növeli a súlyokat amíg a megfelelő megoldást nem kapjuk.

Jelen programunk annyiban tér el, hogy nem változtatja súlyait, hanem az első próbálkozásának eredményét adja vissza.

```
greg@greg-X510UAR:~/Prog1$ ./perc mandel.png
0.516501
greg@greg-X510UAR:~/Prog1$ ./perc mandel.png
0.500152
greg@greg-X510UAR:~/Prog1$ ./perc mandel.png
0.687392
greg@greg-X510UAR:~/Prog1$ ./perc mandel.png
0.730395
greg@greg-X510UAR:~/Prog1$ ./perc mandel.png
0.697868
```

5. fejezet

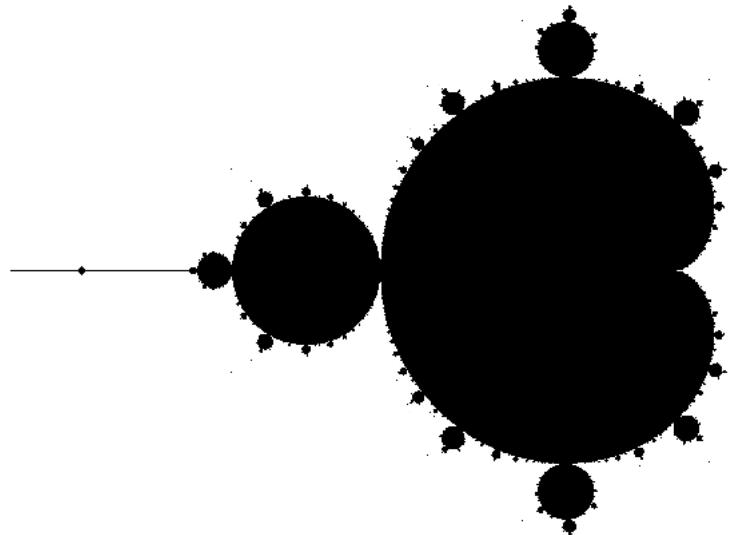
Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

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

Megoldás forrása: bhax/attention_raising/CUDA/mandelpngt.cpp nevű állománya.



5.1. ábra. A Mandelbrot halmaz a komplex síkon

A Mandelbrot halmazt 1980-ban találta meg Benoit Mandelbrot a komplex számsíkon. Komplex számok

azok a számok, amelyek körében válaszolni lehet az olyan egyébként értelmezhetetlen kérdésekre, hogy melyik az a két szám, amelyet összeszorozva -9-et kapunk, mert ez a szám például a 3i komplex szám.

A Mandelbrot halmazt úgy láthatjuk meg, hogy a sík origója középpontú 4 oldalhosszúságú négyzetbe lefektetünk egy, mondjuk 800x800-as rácsot és kiszámoljuk, hogy a rács pontjai mely komplex számoknak felelnek meg. A rács minden pontját megvizsgáljuk a $z_{n+1} = z_n^2 + c$, ($0 \leq n$) képlet alapján úgy, hogy a c az éppen vizsgált rácspont. A z_0 az origó.

Azaz kiindulunk az origóból (z_0) és elugrunk a rács első pontjába a $z_1 = c$ -be, aztán a c -től függően a további z -kbe. Ha ez az utazás kivezet a 2 sugarú körből, akkor azt mondjuk, hogy az a vizsgált rácspont nem a Mandelbrot halmaz eleme.

A program elején megadjuk a kép méretét és az iterációs határt (MERET és ITER_HAT) ameddig nézni fogjuk a rácspont helyzetét, hogy a halmazban van e vagy sem. A program main részében szükség van az argc és argc argumentumokra mivel így a program futtatásakor megadhatjuk milyen néven mentsük el a keletkezett képet. Ezután létrehozzuk az rgb képet, majd a két for ciklus végigmegy a rácson és a fenti algoritmus alapján feketére vagy fehérre színezi a rácspontot. Végül kiírjuk a kép elmentett nevét és hogy mennyi ideig tartott a számítás. Az így kapott kép a mandelbrot halmazról egy fraktál, vagyis ha ráközelítünk a képere, végtelen ideig fog ismétlődni valamilyen minta benne. Ezt egy későbbi feladatban meg is valósítjuk.

5.2. A Mandelbrot halmaz a std::complex osztálytal

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

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

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/4_Mandelbrot

A **Mandelbrot halmaz** pontban vázolt ismert algoritmust valósítja meg a repó [bhax/attention_raising/Mandelbrot/3.1.2.cpp](https://gitlab.com/bhax/attention_raising/Mandelbrot/3.1.2.cpp) nevű állománya.

```
// Verzio:
#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;
}

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";

// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
{
    // k megy az oszlopokon

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

        // c = (reC, imC) a halo racspontjainak
        // megfelelo komplex szam

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

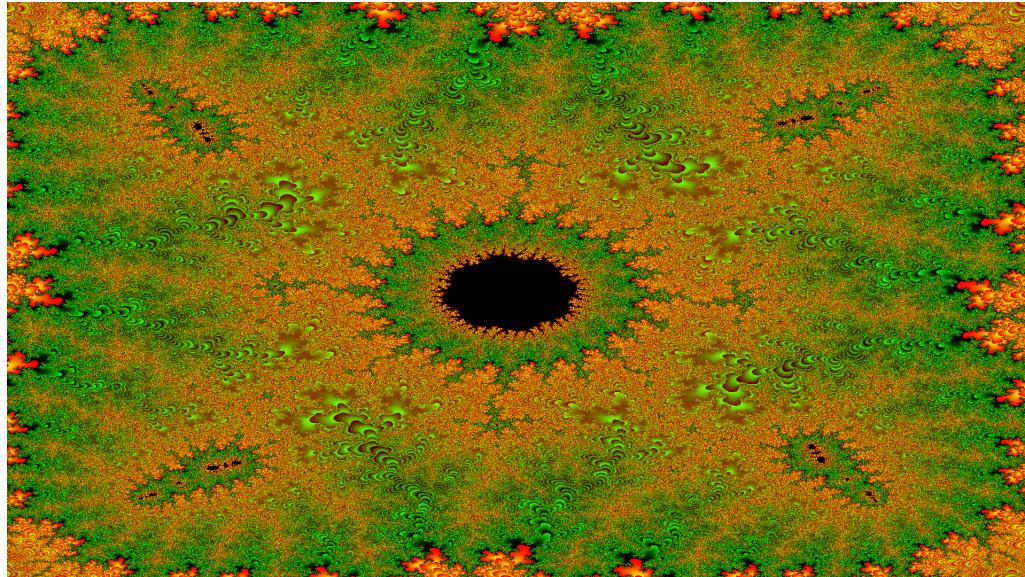
        std::complex<double> z_n ( 0, 0 );
        iteracio = 0;

        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 ) );
}
```



A 3.1.2.cpp nevű program a mandelpngt.cpp továbbfejlesztett változata. A főbb különbségek, hogy a program minden futtatásakor megadhatjuk a parancssorban a kép méretét, az iterációs határt és a b c d változókat amelyek a c complex szám valós és képzetes részét adják majd. Ellenőrizzük hogy megfelelő mennyiségű argumentum van-e és ha nem, hibát dobunk vissza a program használatával a usernek.

Ezen felül használjuk a complex könyvtárat amivel egyszerübb a complex számok képzése. A main részben pont mint az előbb, két for ciklus végig megy a kép pixelein és kiszámolja a C valós és képzetes részét, majd az alapján hogy benne van e az iterációs határban kiszinezzük;most színesre.

5.3. Biomorfok

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

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

A biomorfokra (a Julia halmazokat rajzoló bug-os programjával) rátaláló Clifford Pickover azt hitte természeti törvényre bukkant: https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9_Iss5_2305--2315_Biomorphs_via_modified_iterations.pdf (lásd a 2307. oldal aljától).

A különbség a **Mandelbrot halmaz** és a Julia halmazok között az, hogy a komplex iterációban az előbbiben a c változó, utóbbiban pedig állandó. Ezzel szemben a Julia halmazok csipetben a cc nem változik, hanem minden vizsgált z rácspontra ugyanaz.

```
// j megy a sorokon
for ( int y = 0; y < magassag; ++y )
{
```

```
// k megy az oszlopokon

for ( int x = 0; x < szelesség; ++x )
{

    double reZ = xmin + x * dx;
    double imZ = ymax - y * dy;
    std::complex<double> z_n ( reZ, imZ );

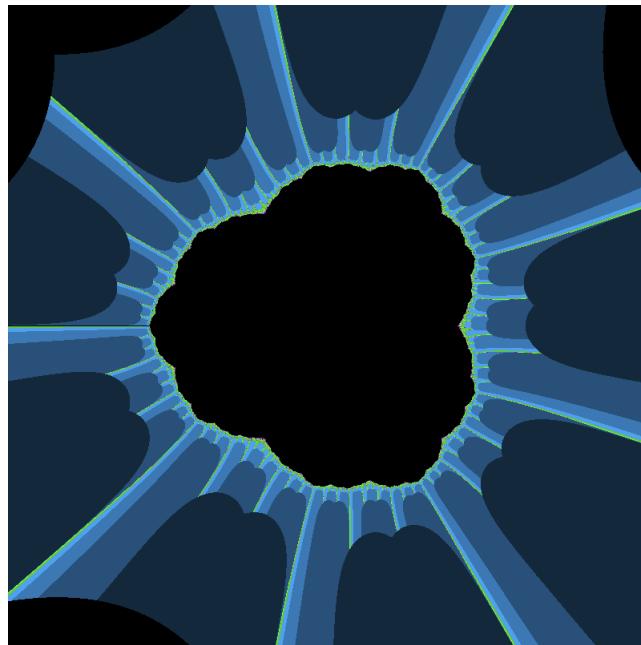
    int iteracio = 0;
    for (int i=0; i < iteraciosHatar; ++i)
    {

        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 ) );
}
```

A biomorf a mandelbort halmaz egy bug által létrejött kvázi mutációja. Clifford Pickover, aki felfedezte azt hitte a sejtek és az élet valamelyen törvényszerűségét sikerült vizualizálni, de ez sajnos nem így van. Bár nem lehet ezt felnöeni neki hiszen az elkészülő képek nagyon hasonlítanak a sejtek mikroszkópikus képeire.

Jelen programunk abban különbözik a 3.1.2.cpp nevű elődjétől, hogy most indításkot lehetőségünk van megadni a c komplex szám kezdőértékét és az iterációs határt is, ha a user nem adott meg elég argumentumot hibaüzenettel szólunk neki. Sőt még eddig csak az iterációval végeztünk maradékos osztást, hogy színessé tegyük a képet, most konstansokkal is szorzunk a színeket meghatározó képeletben.



5.4. A Mandelbrot halmaz CUDA megvalósítása

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

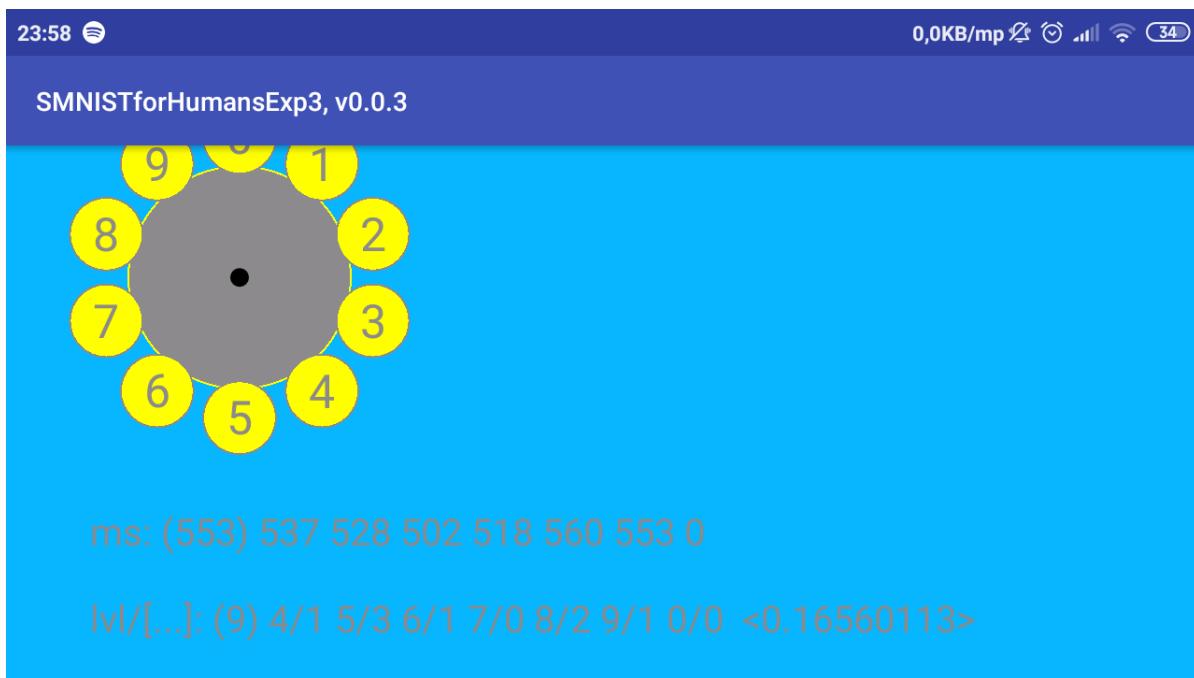
Megoldás forrása: bhax/attention_raising/CUDA/mandelpngc_60x60_100.cu nevű állomány.

5.5. Mandelbrot nagyító és utazó C++ nyelven

Építs GUI-t a Mandelbrot algoritmusra, lehessen egérrel nagyítani egy területet, illetve egy pontot egérrel kiválasztva vizualizálja onnan a komplex iteráció bejárta z_n komplex számokat!

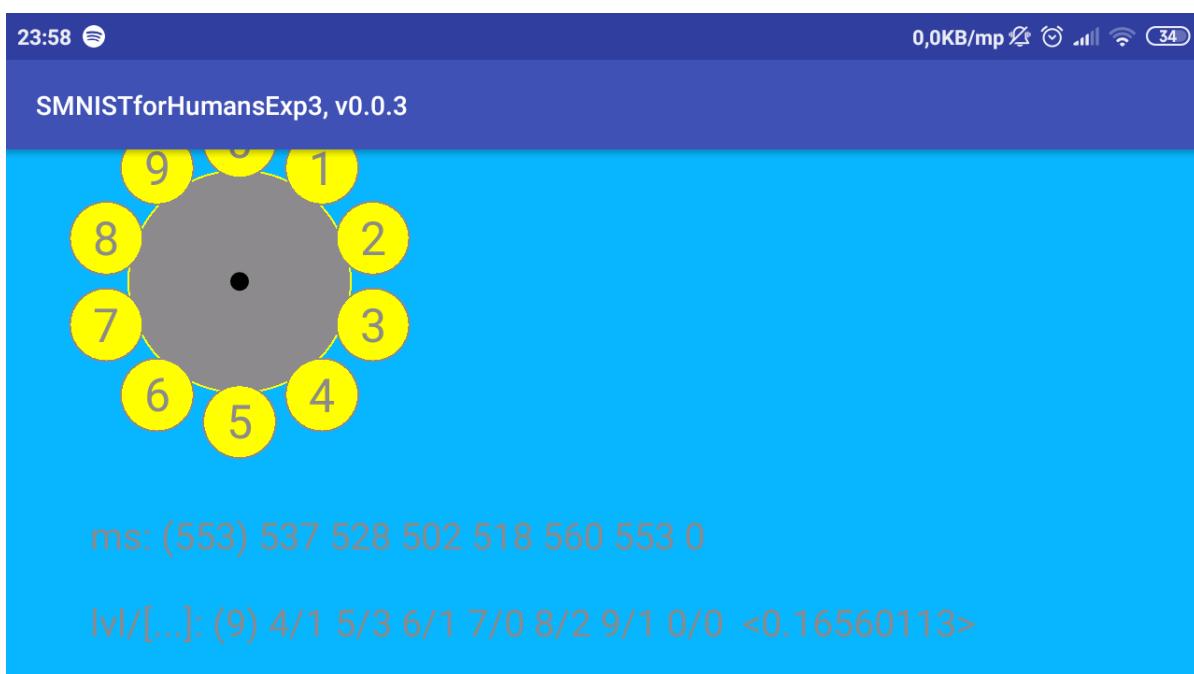
Megoldás videó: Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_a_mandelbrot_halmazzal.

Megoldás forrása:



5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás video: <https://youtu.be/Ui3B6IJnssY>, 4:27-től. Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_apbs02.html#id570518



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: https://gitlab.com/gergoszka/konyv/tree/master/5_Welch

Tanulságok, tapasztalatok, magyarázat...

A Polárgenerátor osztályban először deklarálunk egy nincsTárolt boolean típusú változót ami megmondja, hogy van e eltárolva random számunk, termászetesen alapesetben true értékű. Van egy tárolt változónk ami a nincsTárolt false értéke esetén visszaadja az eltárolt számot.

A következő() tagfüggvényben először megnézzük, hogy van e eltárolva számunk, ha nincs akkor kiszámítunk kettőt: egyiket visszaadjuk a usernek, a másikat elrakjuk a tárolt változóba és a nincstárolt értékét hamisra állítjuk; ha van eltárolt számunk akkor azt visszaadjuk és a nincsTárolt változót igazra állítjuk.

A mainben meghívjuk a PolárGenerátor osztályt g néven és forral csinálunk 10 példát a kimenetre.

Ez nagyban megegyezik ahhoz ahogy a JDK Random.javabán a Sun programozói csinálták:

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

public double nextGaussian() {
    if (haveNextNextGaussian) {
        haveNextNextGaussian = false;
        return nextNextGaussian;
    } else {
        double v1, v2, s;
        do {
            v1 = 2 * nextDouble() - 1;    // between -1.0 and 1.0
            v2 = 2 * nextDouble() - 1;    // between -1.0 and 1.0
```

```
s = v1 * v1 + v2 * v2;
} while (s >= 1 || s == 0);
double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
nextNextGaussian = v2 * multiplier;
haveNextNextGaussian = true;
return v1 * multiplier;
```

6.2. LZW

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

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/5_Welch

A binfa algoritmus egy vektorban tárolt bitsorozatból bináris fa típusú adatszerkezetet csinál.

Létrehozunk egy binfa nevű struktúrát amiben deklarálunk egy int érték és 2 struct típusú mutatót, amik a gyökér jobb és bal oldali elemeire mutatnak.

A BINFA_PTR függvény fogja visszaadni majd a p eredményét, eg sikeres memória foglalás után. A kiir és szabadít függvényeknél extern-nel jelezzük hogy majd később deklaráljuk őket. A mainben létrehozunk egy char változót amib mindenkor minden lementjük éppen milyen elemet olvasunk be. Aztán deklaráljuk a fát és a fa mutatót ráállítjuk a gyökérre és a while ciklusban a jobb vagy bal oldalra rajuk az elemet attól függően hogy 1 vagy 0

A kiirral végigmegyünk a fán és növeljük a mélyságet minden ugrásnál majd ábrázoljuk magát a bináris fa kinézetét a standard outputon. A szabadit felszabadítja a az erőforrásokat használat után.

```
0111001010110101011101001011110
-----1(3)
-----1(2)
-----1(6)
-----1(5)
-----1(4)
-----0(5)
-----0(6)
-----0(3)
---/(1)
-----0(2)
-----0(3)
melyseg=6
greg@greg-X510UAR:~/Prog1/Gitlab/konyv/5_Welch$
```

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása:https://gitlab.com/gergoszka/konyv/tree/master/5_Welch

Inorder: először a gyökér bal oldalát, majd magát a gyökeret és végül a jobb oldalt írjuk ki.

Preorder: először kiírjuk a gyökeret és csak ez után a bal és jobb oldalt.

Postorder: kiírjuk a bal és jobb oldalát a fának és végül a gyökeret.

Ez a kódban annyit jelent, hogy Inordernél a for ciklus előtt van az egyes gyerekek kiírása a nullásoké pedig utána. Preordernél minden a kettő mögötte, Postordernél minden a kettő előtte van.

```
-----1(3)
-----   1(6)
-----   1(5)
-----   0(6)
-----   0(5)
-----   1(4)
-----   0(3)
-----   1(2)
-----   0(3)
-----   0(2)
---/(1)
melyseg=6
```

Postorder

```
---/(1)
---   1(2)
-----   1(3)
-----   0(3)
-----   1(4)
-----   1(5)
-----   1(6)
-----   0(5)
-----   0(6)
-----   0(2)
-----   0(3)
melyseg=6
```

Preorder

6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása:https://gitlab.com/gergoszka/konyv/tree/master/5_Welch

Mivel ezt a programunkat most C++ ban írjuk lehetőségünk van az osztályok használatára: létre is hozzuk az LZWBInFa osztályt. Deklaráljuk az osztály konstruktőrét, ami az inicializációs listájában meg is kapja a fa jelenlegi gyökerét, ezzel tudjuk hogy jelenleg hol vagyunk a fában. E mellett létrehozunk destruktort is, ami a program végeztével felszabadítja a gyökérnek a heapen lefoglalt helyét.

```
class LZWBInFa
{
public:

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

A main részben új featuret adtunk hozzá, ugyanis most lehetőségünk van nem csak a bemenő, hanem a kimenő fájl nevét is megadni. A beírt argumentumok mennyiségét az argc , értékeit az argv tárolja,ha nem megfelelő az argumentumok mennyisége vagy értéke erről figyelmeztetjük a usert.

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

6.5. Mutató a gyökér

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

Megoldás videó:

Megoldás forrása:https://gitlab.com/gergoszka/konyv/tree/master/5_Welch

Mutatót csináltunk a gyökérből így elhagyhatjuk az összes címe operátort gyökér elől ahol eddig ott volt, mivel alapjáraton is egy memóriacímet fog visszadni. Át kell írnunk a metódusok referenciaját is `.' ról -> ra minden olyan helyen ahol használjuk. Eddig tagként volt jelen a csomópontban, de most hogy mutató lett már egyszerűen hivatkozhatunk rá a fában is.

```
LZWBinFa ()
{
    gyoker = new Csomopont ();
    fa=gyoker;
}
~LZWBinFa ()
{
    szabadit (gyoker->egyesGyernek ());
    szabadit (gyoker->>nullasGyernek ());
    delete gyoker;
}
```

Viszont a destruktörben fel kell szabadítani a neki lefoglalt helyet egy delete parancsal különben segfault-os lesz a program.

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása:https://gitlab.com/gergoszka/konyv/tree/master/5_Welch

Ehhez a példához az előző feladatokban is használt binfa programot fogjuk tovább hackelni. Ahhoz hogy mozgatni tudjunk egy osztályt létre kell hozni egy mozgató konstruktort és ennek egy =operátor túlterhéését. A konstruktorban a dupla címe operátorral jelöljük, hogy itt most mozgatni fogunk. A mostani fa gyökerét nullprt-re állítjuk majd a régi fát "átmozgtjuk" a std::move függvényvel. Ez igazából nem mozgat semmit ahogy a neve indikálná, hanem az argumentumát lvalue-ból xvalue-vá konvertálja, ami a compiler számára jelzi hogy törölhető a memóriacím mert az utasítás után ágy is mozgatva lesz. Majd túlterheljük a = operátort. Itt az újonnan létrejött gyökérbe belerakom a régi elemeit, majd visszaadjuk a *this-el. Végül töröljük a régi gyoker elemeket.

```
LZWBInFa ( LZWBInFa && old )
{
    gyoker = nullptr;
    *this = std::move(old);

}

LZWBInFa & operator= (LZWBInFa && old)
{
    std::swap(gyoker, old.gyoker);

    return *this;
    delete gyoker;
}
```

A main részben ezt úgy használjuk, hogy a move()-val átrakjuk a binfa-t az Binfa2-be. Eztután a művelet után az a régi Binfa üres lesz míg Binfa2 tartamazni fogja a régi elemeit.

```
LZWBInFa Binfa2 = std::move(Binfa);
```

7. fejezet

Helló, Conway!

7.1. Hangyszimulációk

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

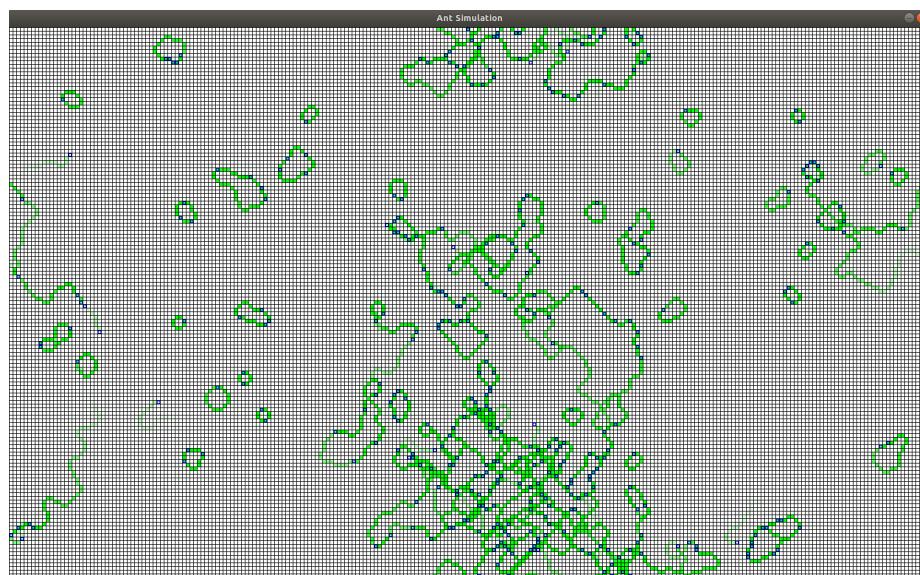
Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/6_Conway

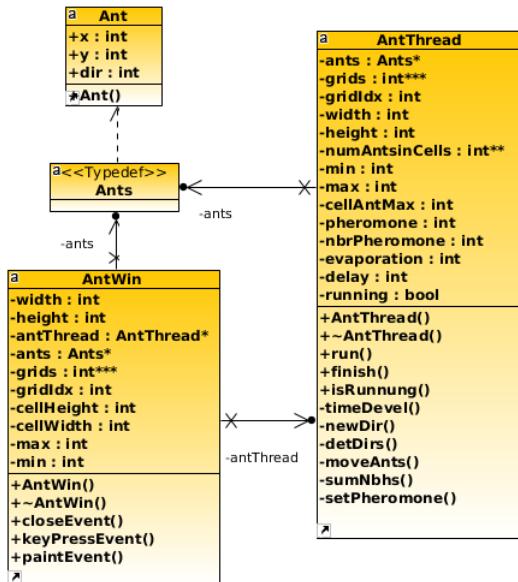
Forrás: <https://bhaxor.blog.hu/2018/10/10/myrmecologist?token=b90955bd5311e9986f0c3f99263ff386>

A hangyszimulációban egy hangy kolónia mozgását szimuláljuk. minden hangya feromon nyomot hagy, és ha feromon nyomra lép egy hangya, akkor elkezdi követni az előző hangya által hagyott nyomat, de ez egy idő után eltűnik ha nem lép rá senki. Minél erősebb a feromon annál nagyobb az esélye, hogy egy közeli hangya rálép. Először létrehozzuk a hangya osztályt, amelyben a hangyák koordinátáit és mozgásukat adjuk meg. A mozgásuk irányát maradékos osztással számoljuk.

A terminálban a program futtatásakor megadhatunk különböző flageket amelyek változtatják a tulajdon-ságait: -w a rács szélességét, -m a rács magasságát, -n a hangyák mennyiségét, -t a hangyák 2 lépés közötti sebességét, -t a párolgás értékét, -f a hanyott feromonok értékét, -a és -i a cellák maximum és minimum értékét , -c pedig azt adja meg hogy hány hangya lehet egy cellában egyszerre.



Az AntWin határozza meg az ablak tulajdonságait: hangyák kinézetét(ami az én változatomban kék négyzetekkel jelölünk), a rács méretét. Itt történik a key eventek implementálása is, P esetén a program megáll, Q vagy Esc lenyomásakor kilépünk. Ha nem adunk meg semmilyen kapcsolót futtatáskor az itt megadott értékeket használjuk.



7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.3. Qt C++ életjáték

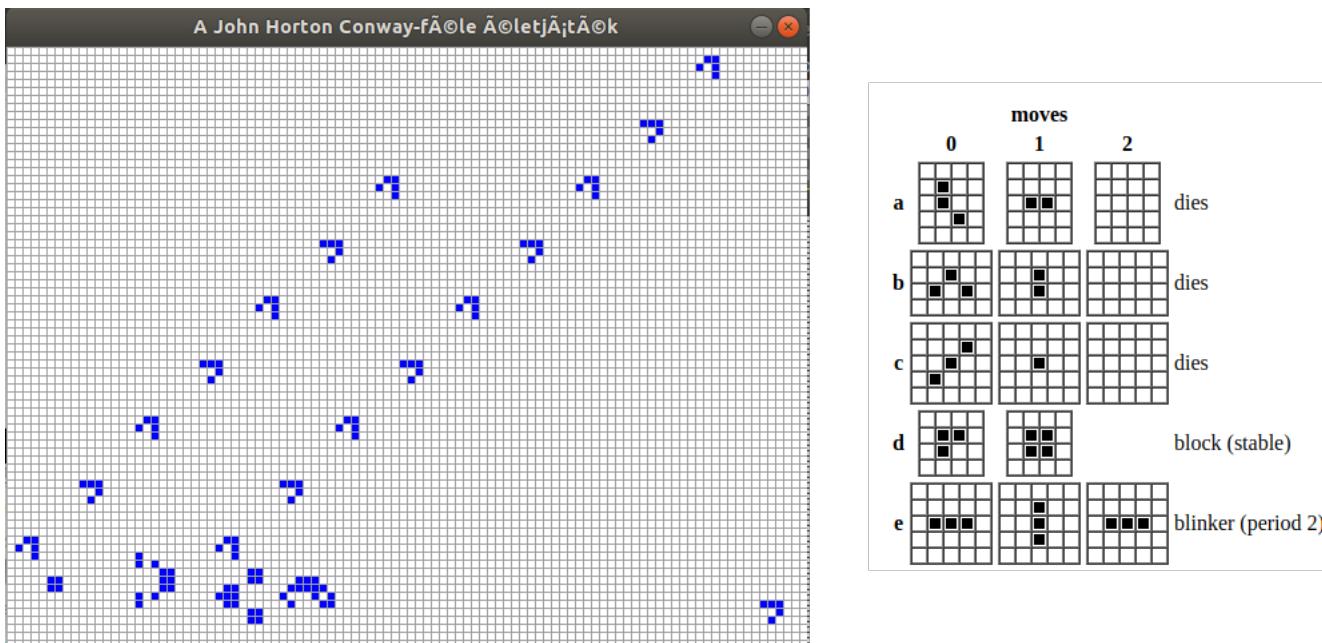
Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/6_Conway <https://bit.ly/2G2ojbN>

Tanulságok, tapasztalatok, magyarázat...

Az eredeti életjátékot (The Game of Life) John Horton Conway matematikus találta ki 1970-ben. Ez lényegében egy "nulla-fős" játék hiszen csak a kezdeti állapotot adhatjuk meg a többi a játék szabályai szerint kell kiszámolni. Ezt meg lehetne tenni kockás papíron is, de az unalmas és hosszadalmas ezért ezt a program megsinálja helyettünk.



A kísérletek száma változóban definiáljuk, hogy hányszor fusson le a kísérlet. Azaz a minták száma.

A kiserlet és a jatekos tömbök, amelyeket 1 és 3 közé eső számokkal tölt fel a sample. A műsorvezető egy vektor amelyet ugyan olyan méretűre deklarálunk mint a kísérletek száma.

Szabályok:

Egy x^*y méretű rácson játszunk, ahol a rácspontokat celláknak nevezzük. A játék kezdetén minden cella halott, mi dönjük el melyekre helyezünk sejteket, amik a játék elindítása után minden 'körben' a következő szabályok szerint élnek/halnak meg. (Egy sejt szomszédságán a körülötté lévő 8 cellát értjük)

- 1. Ha két vagy három szomszédja van, a sejt túléli a kört.
- 2. A sejt meghal, ha kettőnél kevesebb, vagy háromnál több szomszédja van.
- 3. Új sejt születik minden olyan cellában, melynek környezetében pontosan három sejt található.

Fontos megjegyezni, hogy a születések és elhalálozások egyszerre történnek, vagyis azok a sejtek amik meghalnak bele számítanak a születő sejtek meghatározásába. A szabályok szerinti sejtek felrakását és eltávolítását egy körnek vagy generációnak nevezzük.

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/6_Conway

Tanulságok, tapasztalatok, magyarázat...

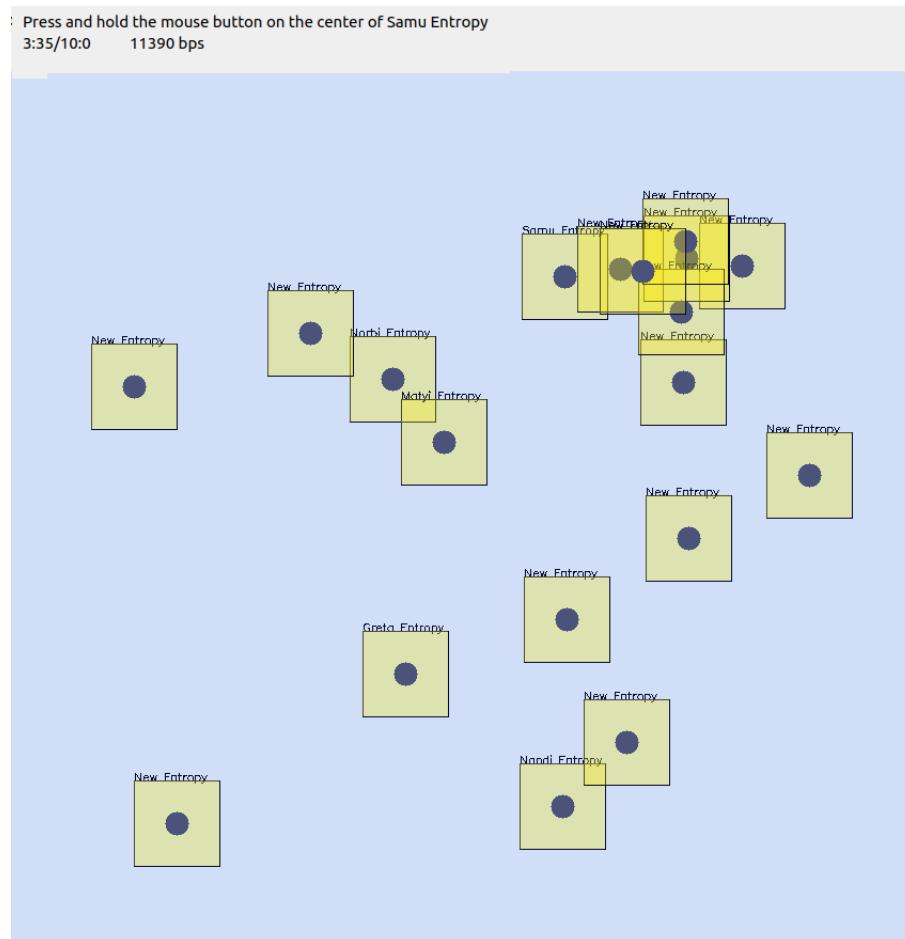
Forrás: http://real.mtak.hu/79216/1/it_2018_1_10_batfai_et_al.pdf

A BrainB benchmark program lényege a felhasználó koncentrációs képességének mérése. A teszt lényege, hogy az egér kurzort folyamatosan a 'Samu entropy' nevű nézet közepén, a kék körben kell tartani.

Samu és a többi objektum is random mozognak, a feladatot nehezíti, hogy minél tovább tartjuk rajta az egeret annál több új entropyt spawnolunk és gyorsaságuk is nő. Viszont ha elveszjük Samut csökken az entropyk száma és sebessége. A teszt 10 percig tart, amit kicsit soknak érzek mert elfárad az ember szeme és a keze is a folyamatos egérgomb lenyomástól a végére. Ha teljesítettük a tesztet a program egy külön fájlba elmenti eredményünket, amit a lost2found és found2lost változókból így számolunk:

```
int m1 = mean ( lost2found );
int m2 = mean ( found2lost );
double res = ( ( (double)m1+(double)m2 ) /2.0 ) /8.0 ) /1024.0;
textStream << "U R about " << res << " Kilobytes\n";
```

A BrainBThred konstruktorában deklaráljuk a hős osztályt, és állítjuk be a mozgásukat, amit random számolunk. Ezen felül itt írjuk meg az adatok méréséhez szükséges get_bps, get_w, lost, stb. függvényeket. A BrainWin-ben írjuk meg az eventeket amik érzékelik hogy az egér gombja le van-e nyomva vagy sem és ez alapján indítja vagy állítja meg a játékot. Ha Samun van ez egér új entropykat hozunk létre és növeljük az agilitijét, ha elveszítjük entropykat törlünk és lassítjuk. Ezen felül implementáljuk hogy a P gomb lenyomásával meglehessen állítani a játékot.



8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

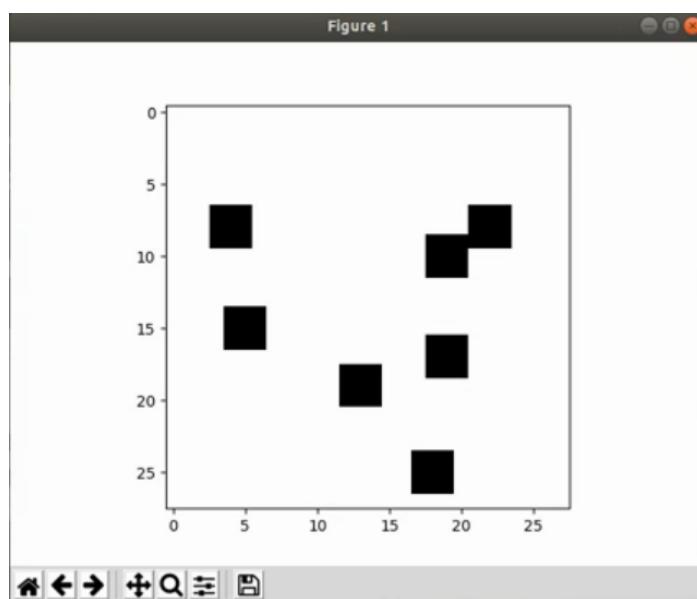
Python

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

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

Tanulságok, tapasztalatok, magyarázat...

A tensorflowt a google készítette a gépi tanulást segíti, a tervezésben a fejlesztésben és a tanulmányozásban használják főként mivel készít adatáramlási gráfot, amelyben a node-ok a matematikai műveletek és az élek az az áramló adatok.



A tensorflow-ot import tensorflow kent hívjuk meg. A readimg függvény beolvassa a kép file-t majd de-kódolja, erre a későbbiekben lesz szükség. A program lényege, hogy a megadott képen szereplő számot felismerje. Ehhez meg kell tanítanunk a programunkat. Tehát elsőnek készítünk egy modelt. Majd ezen gyakoroltatjuk a programunkat. Majd futtatunk egy teszt kört ahol a program kiirja a becsült pontosságát. Ezután a 42 es tesztkép felismerése következik. Majd végül a beolvastott képünkön teszteljük a program működését.

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Minecraft-MALMÖ

Megoldás videó: <https://youtu.be/bAPSu3Rndi8>

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása: https://gitlab.com/gergoszka/konyv/tree/master/8_Chaitin

Lispben két fő szintaktikai különbség van a híresebb programozási nyelvekhez képest. Az első az, hogy maga a program egymásba ágyazott S-kifejezések sorozata, a nyelv egyik nagy előnye és hátránya is egyben: egyszerű hozzá elemző programokat írni de az emberi szem hamar belekavarodik a sok zárójelbe. Második pedig hogy a nyelv prefix jelölést használ. Ez azt jelenti hogy egy például egy összeadásban először kell az összeadásjelet leírni és utána a művelet argumentumait(pl: + 2 2).

Rekurzív faktoriális

```
(defun fakt_r (n) (if(< n 1) 1 (* n (fakt_r(- n 1)))))
```

Definiáljuk a fügvényünket a defun kulcsszó után fakt_r néven, ahol n-re helyetessíjük be a kívánt faktoriálist. Ha n kiseb mint 1 akkor 1-et adunk vissza eredményül, egyébként n-et megszorozzuk a rekurzívan meghívott fakt_r függvényvel az n-1 elemre.

Iteratív faktoriális

```
(defun fakt_i (n) (loop for i from 1 to n for result = 1 then (* ←
    result i) finally return result))
```

Definiáljuk a fügvényünket fakt_i néven Majd egy loop for-al kezdünk, ami lényegében a lisp megfelelője az átlagos for-nak, ebben a ciklusváltozó i=1-től n-ig fut le a ciklus. minden iterációban megszorozzuk a result változót (amivel a végeredményt számoljuk) i-vel és így a vegére megkapjuk az adott faktoriálist amit returnnel visszaadunk.

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_hackelese_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

[?]

2.4 Adattípusok

Az adattípusok a programozási nyelvek egy absztrakt eszköze, ami minden valamilyen másik feladatunk részeként jelenik meg. Hárrom tulajdonság jellemzi őket: tartományuk, műveleteik és reprezentációjuk. A tartománya az spectrum melyen belül az adattípus elemi felvehetnek értéket. Azok a műveletek tartoznak hozzá, melyeket az elemein eltudunk végezni. Reprezentációja pedig az egyes elemek tárolását mutatja meg bitekben(pl: C-ben egy chart egy bájton reprezentálunk). minden típusos nyelvben vannak beépített típusok, néhányban pedig magunk is deklarálhatjuk őket a három tulajdonságuk megadásával. Ezen felül úgy is letrehozhatunk saját típust hogy egy már meglévő tartoményt csökkentjük. Az adattípusok két nagy csoportja az egyszerű és az összetett típusok. Az egyszerű közé tartoznak az egész, valós, karakter és a logikai típus míg az összetetthez tömb és a rekord. Ezenfelül vannak speciális típusok, például a mutató aminek adatrészében egy másik elem memóriacíme van, alapvető feladata a tárhelyen lévő adat elérése.

2.5 Nevesített konstans

A nevesített konstansnak van neve, típusa és értéke. Mindig deklarálni kell és egy olyan értéket jelöl beszédesebb nevekkel amelyek a programunk alatt nem változtatnak értéket. Ezen felül ha meg kell változtatni az értéket, akkor elég egy helyen és nem minden előfordulásában átírni.

2.6 Változó

A változó egy olyan programozási eszköz melynek neve, attribútumai, címe és értéke van. A nevével jelöljük programkódunkban, attribútumokat deklarációkor kap(pl típus), címe a memóriában elfoglalt helye, értéke pedig ezen a helyen lévő információ.

4. Utasítások

Az utasítások segítségével írjuk le az algoritmusunk feladatát lépésenként. Egy részük csak a fordítóprogramnak szólnak, ezek a deklarációs utasítások, melyek valamelyen szolgáltatást kérnek vagy információt szolgáltatni a fordítónak. Másik részük pedig a végrehajtó utasítások, melyekből majd a programkód épül fel. Csoportosításuk:

- 1. Értékadó utasítás: Ezzel adunk a változóinknak értékét.

- 2. Üres utasítás: Mint nevében is szerepel az ilyen utasítások törzse üres. Ilyenkor a processzor egy üres gépi utasítást csinál.
- 3. Ugró utasítás: A program jelenlegi pontjáról egy másik meghatározott pontra dobuk a vezérlést.
- 4. Elágaztató utasítások: Van kétirányú elágazó utasítás (if else) mely egy feltétel alapján két lehetséges tevékenység között választ. Ennek az összetettebb változata a többirányú elágazó utasítás(switch) ami egy feltétel alapján több lehetséges esetet próbál ráilelsteni, ha sikerül azt hajtja végre, ha nem akkor van egy default válasz hajtódiék végre.
- 5. Ciklusszervező utasítások: Lehetővé teszik , hogy a program valahány utasítását bármennyiszer megismételjük. Maga a ciklus magjában található a kód, ami megismétlésre kerül majd. Fontos része a ciklusoknak az ismétlődésre vonatkozó információ, ami lehet: egy feltétel igaz/hamis állapota, ezt a ciklus minden iteráció után(do..while)vagy előtt(while) ellenőrzi; lehet előírt lépésszámú ciklus is amely egy előre megadott értékszer hajtja végre a magját(for ciklus).
- 6. Hívó utasítás:
- 7. Vezérlésátadó utasítások: CONTINUE:Megnézi a ciklus ismétlődés feltételeit és vagy újrakezdi a cilust vagy befejezi azt; BREAK: A ciklust szabályosan befelyezi és kilép belőle; RETURN(érték): Befejezi a függvényt és visszadja az értéket a hívónak.
- 8. I/O utasítások
- 9. Egyéb utasítások

5.1 Alprogramok

Egy programnyelvknél minden kérdés, hogy hogyan lehet azt feldarabolni ls a tagokat hogyan kell fordítani. Lehetséges minden önálló részt külön, az egész programot egyben és ennek kombinációját is használni. Az eljárásorientált nyelvekben a programegységek között megkülönböztetünk alprogramot, blockot,csomagot és taszkot. Az alprogramok olyan kódrészletek, melyeket csak egyszer kell megírni, azután az egész programon belül megírhatjuk amikor szükségünk van rá. Egy alprogramot függvénynek nevezünk ha van visszatérési értéke, egyébként egy eljárásról beszélünk. Egy alprogram fejében megadhatunk argumentumokat, ezeket a megírásakor ki kell töltenünk a megfelelő típusú adatokkal, ez akkor hasznos ha valamilyen olyan adatot akarunk neki átadni ami a scope-ján kívül van.

5.2 Hívási lánc, rekurzió

Minden programegység hivatkozhat másik programegységekre, ennek a folyamatát hívási láncnak nevezzük. A hívási láncot minden a főprogram kezdi és szabályos esetben minden az akívvval kezdve zárul be Ha az alprogram egy akív alprogramot hív meg azt rekurziónak nevezzük. Ez lehet követlen, amikor az aktív alprogram saját magát hívja meg és közvetett, amikor egy másik programot hívunk meg a hívási láncból.

5.3 Másodlagos belépési pontok

Egy alprogramra nem csak a fejen keresztül tudunk hivatkozni. Egyes programnyelvek megengedik, hogy a program törzsében létrehozzunk egy másodlagos belépési pontot. így ha ezen keresztül érjük el az alprogramot csak a belépési pont utáni kód fut le a törzsben.

5.4 - 5.5 Paraméterek

Amikor egy alprogram meghívásánál az aktuális a paraméterek a formális paraméterekhez rendelődnek paraméterkiértékelésnek nevezzük. Egy programon belül a formális lista minden állandó, egy darab van

belőle, míg az aktuális paraméterek meghívásonként változhatnak. Ezeket az értékeket sok féleképpen lehet egymáshoz rendelni, leggyakoribb az érték és a referencia szerinti paraméterátadás. Érték szerinti paraméterátadásnál a megkapott változók eredeti értékei nem változnak, csak az alprogramon belül, míg referencia szerinti átadásnál mutatókat alkalmazunk a megkapott változó címére , így azt változtatjuk és nem egy másolt értéket.

5.6 Blokk

Egy blokkot különleges karakterekkel ({ },begin..end) kezdődő és végződő programegység. Egy blockba belépni és kilépni legegyszerűbben úgy lehet hogy a vezérlés rá kerül, de alkalmazhatunk GOTO utasítáast is erre a cérla. . Fő szerepe a program átláthatóvá tétele és a hatáskörük eltárolása.

5.7 Hatáskör

A hatáskörök olyan területek a programon belül, ahol egy adott név minden ugyanazt az értéket hivatkozza. Egy programegységen belül nevet lokális névnek nevezünk, az e feletti hatáskörökben deklarált neveket pedig globális neveknek. A nevek láthatósága kívülről befelé működik, vagyis a főprogram változóit mindenki látja , míg egy eljárásét csak ő. A hatáskörök lehetővé teszik hogy egy változót például kétszer deklarálunk két különböző alprogramban amik nem látják egymást, így nem akadnak össze az értékeik, bár ez is sem javasolt programozói szokás.

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

3.1 Utasítások és blockok

C-ben egy egy kifejezés akkor tekinthető utasításnak ha pontosvessző zárja, egyszerű elv mégis sok kezdő programozó fordítási hibáinak forrása. Egy blockot kapcsos zárójelek között hozhatunk létre, ezeknek majd a hatásköröknek lesz jelentőségük.

3.2 If-else utasítás

Az if-else szerkezet egy két végkimenettel rendelkező döntést hozó utasítás. A szyntaxa szerint az if után következik egy feltétel, ami ha igaz az if utáni block hajtódi végre, ha pedig hamis az else ág utáni. Az else ág nem mindenkor szükséges, ilyenkor a feltétel hamis léte esetén nem történik semmi. Az ifeket lehet egymásba fűzni, ezt úgy érjük el, hogy az else után egy if-et írunk, amit egy újabb kifejezésnek kell követnie. Ilyenkor a legutolsó else akkor fut le ha semelyik feltétel sem teljesült.

3.4 Switch

A switchet leginkább egy if-else ághoz lehetne hasonlítani azzal a különbséggel, hogy itt a kifejezést nem csak igaz-hamis állapotok szerint mérjük, hanem lehetőség van több lehetséges eredmény esetén különböző válaszokat adnunk. A különböző eseteket a case kulcsszó után kell írnunk, ebből bátmennyi lehet de mindeneknek egy break-kel kell zárónia. Ezen felül szükség van egy default válaszra, ami akkor aktiválódik ha egyik case-re sem illeszkedik a kifejezésünk.

3.5 While és for utasítás

A while után meg kell adni egy kifejezést zárójelekben. Ha ez az érték nem nulla akkor elkezdi a ciklust, ha végzett vele újra ellenőrzi a kifejezés értéket, ezt teszi egészen addig amíg az érték nulla nem lesz. A for

egy előre meghatározott lépésszámú ciklus, utána három argumentumra van szüksége a működéshez. Az első a kezdő érték, ahonnan kezdjük a számlálást, a második egy kifejezés vagy érték amit el kell érnie a kezdőértéknek hogy befejeződhessen a ciklus, harmadik pedig maga a lépésköz megadása amivel növeljük a kezdőértéket minden iterációban. Mind a kettőből kiléphetünk a 3.7-ben tárgyalt utasításokkal vagy az is megeshet hogy -direkt vagy véletlen- véglesen ciklust hoztunk létre és soha nem lépünk ki belőle.

3.6 Do while utasítás

Mind a for, mind a while a ciklus elején vizsgálja a kilépési feltétel teljesülését, ezt a do-while a ciklustörzs végrehajtása után teszi meg. így látható hogy a ciklus minimum egyszer biztosan le fog futni.

3.7 Break, continue, goto utasítások

A BREAK utasítással megszakítjuk az adott blokk folyamatát és azonnal kilépünk belőle, folytatva az adott block után mintha szabályosan végbement volna.

A CONTINUE hasonló mint a break bár kevesebbet használjuk. Ezt is egy ciklus törzsébe rakjuk és ha rákerül a vezérlés az adott ciklus újra ellenőrzi kilépési feltételét és annak értéke szerint vagy új iterációt kezd vagy kilép a ciklusból.

A GOTO-val egy megadott círe tudunk ugrani, használata nem ajánlott, de lehet haszna mélyen egymásba ágyazott ciklusokból való kilépés esetén.

10.3. Programozás

2.1.1 Függvényparaméterek és visszatérési érték

Ha C-ben megadunk egy függvényt paramétereik nélkül az bármennyi paraméterrel hívható, míg C++-ban azt jelenti, hogy nincs paramétere. Sőt ha visszatérési értéket nem adunk meg a C automatikusan intet fog visszaadni, a C++ pedig hibát mivel nincs alapértelmezett visszatérési értéke.

2.1.2 Main függvény

Az main függvény argumentumaként dekralálható argc a parancssorba beérkező argumentumok számát, míg az argv a paracssori argumentumokat adja meg. C++ nem muszáj returnt írni a main végére.

2.2 Függvények túlterhelése

Két függvényleg lehet ugyanaz a neve amíg argumentumlistájuk különbözik. A C++ a függvény nevet az argumentumokkal együtt tárolja, így a különböző változatoknak nem kell egyesével nevet adni.

2.3 Alapértelmezett függvényargumentumok

C++ nelvben lehetőség van a függvények argumentumainak alapértéket megadni. Ekkor ha a függvény hívásakot nem adjuk meg paraméterként akkor az alapértelmezett értéket használja.

2.4 Paraméteradás referenciatípussal

Ha azt szeretnénk C-ben, hogy egy függvény megváltoztassa az argumentum változó értékét pointerekkel kell hivatkoznunk rá, íg meg tudja változtatni az értékét. C++-ban ezt referenciatípus bevezetésével hidálják át. A C teljes pointerré alakítás helyett elég egy jelet írni a függvény deklarációjában a változónév elő.

3. fejezet

C-ben és assemlyben viszonylag egyszerű gyorsan futó kódot írni. Ez a egyszerűség viszont egy bonyolultabb programoknál problémákat okozhat mivel egy szint után rendezetlen, átláthatatlan lesz a kódunk. Erre

a problémára az 1990es években dolgoztak ki egy paradigmát : az objektum orientált nyelveket. Az OO nyelvekben az egy entitáshoz tartozó tulajdonságokat, függvényeket és változókat egy oszállyhoz rendeljük, ezt a folyamatot egysége zárásnak (encapsulation) nevezük. Az osztályoknak vannak olyan részei melyeket a program más része nem lát (data hiding), ez egyrész hasznos a kód egyszerűsítésében, másrészt az osztályon kívül lévő folyamatok így nem tudnak belenyúlni és így összekuszálni azt. Ezeket az osztályokat hogy használni tudjuk példányosítanunk kell. Ez azt jelenti hogy az általános entitást (auto) egy speciális egyedé tesszük (Ford focus 2004), így létrejön egy objectumunk. Az így létrejött objectum örökli az elődje tulajdonságait, de azoknak új értéket adhatunk, sőt növelhetjük tulajdonságainak listáját.

Egy struktúra adattagjait itt tagváltózóknak, függvényeit pedig metódusoknak nevezzük. Ahányszor egy tagváltózót hozunk létre az külön memóriaterületet foglal nekik, ez a metódusokra viszont nem igaz. C++ ban is a . ls -> operátorokkal tudunk hivatkozni az adattagokra, de vigyáznunk kell viszont ,hogy ezek nem minden egymás után tárolódnak a memoriában így ezt az előnyt már nem használhatjuk ki. A C++ függvények első argumentuma mindenig egy "láthatatlan" pointer. Ez a mutató mindenig a példányosított függvényre mutat és az osztályon belül a this kulcsszóval tudjuk elérni. Egy osztályban három rész van adatrejtés szempontjából: public, private, protected. A public részben deklarált függvényeket és változókat mindenki eléri, míg a privateben lévőket csak az osztályon belüli függvények használhatjuk. A protected sokban hasonlít a private-ra azzal a különbséggel, hogy az osztály gyerekei is láthatják.

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, Berners-Lee!

11.1. Python

A Python egy 1990-ben kifejlesztett programozási nyelv, ami népszerűségét könnyű tanulhatóságának és a más nyelvekkel való kompatibilitásának köszönheti. Lényegében egy szkriptnyelv, de rengeteg rengeteg csomagot tartalmaz így általában bonyolultabb feladatok prototípusainak elkészítésére használják. Ezt elősegíti, hogy a C++-al vagy a Java-val ellentétben nincs szükség külön fordításra és futtatásra, a python értelmezőnek elég a forrás és azt automatikusan fordítja és futtatja is. Rengeteg platformon elérhető és a magas szintű adattípusok miatt már akár tekinthető magas szintű programozási nyelvnek. A python egyik leginkább szenbetűnő jellemzője, hogy szintexisa behúzásalapú. Nincs szükség kapcsos zárójelekre, egy metódus törzsét egy tabulátor vagy néhány szóközös behúzás jelzi. Sőt a kifejezések végén sincs szükség pontosvesszöre, az interpreter a sor végét veszi záró karaktereket.

A pythonban minden adatot objektumok ábrázolnak. A változók deklarációjakor nincs szükség annak típusának megadásásra, azt automatikusan futási időben adja meg a hozzárendelt érték alapján. A python a következő típusokat ismeri: számok, sztringek, ennesek, listák, szótárak. A változók alatt objektumokra mutató referenciakat értünk. Egy objektumra több változó is mutathat, viszont ha minden referencia törlődik róla, akkor a garbage collector szabálytja fel az objektum memóriaterületét. A változók lehetnek lokálisak és globálisak is. Egy függvényen belül felvett változó alapvetően lokális, ha globálissá szeretnénk tenni a 'global' kulcszóval kell deklarálni a függvény elején.

A számok közötti konverzió támogatott, sőt sztringekből is képezhetünk számokat. A karakteres szekvenciáknak (sztring, lista, ennes) meg tudjuk mondani a hosszát, első és utolsó elemét és a konkatenáció is értelmezett rajtuk. A szekvenciák elemeit indexekkel érhetjük el. Az indexelés nullánál kezdődik, a negatív index azt jelentése, hogy hátulról visszafelé nézzük. Intervallumot is megadhatunk : karakterek között, ekkor az indexek közötti karaktereket kapjuk. A python alap kiírató methódusa a print. Ez után veszszővel elválasztva sorolhatjuk fel a sztringeket vagy más tetszőleges változókat.

11.2. C++ és Java összehasonlítása

1.Hét: Az objektumorientált paradigmával alapfoglalma. Osztály, objektum, példányosítás.

A C++ és a Java is objektum orientált programozási nyelv, így követik az ezzel járó szemléletmód általános tulajdonságait. Mindkét nyelvben a programozási hozzállás alapegységei az osztályok. Az osztályok valós

dolgok általános jellemzőit írják le. Ez a valami lehet egy fizikai doleg: autó, labda vagy bútor, és lehet valami megfoghatatlan pl: bankszámla. De egy osztály ezen entitásoknak csak egy tervrajza, még nem létezik az adott doleg csak megterveztük. Ahhoz, hogy létre is jöjjön példányosítani kell az osztályunkat, így egy objektumunk lesz amivel már lehet interaktálni. Mindkét nyelven hasonló módon fej és törzs részből épülnek fel az osztályok. A fej részt a class szó vezeti be majd az osztály neve követi, utána pedig egy zárójel jön, benne esetleges argumentumokkal. Az osztály törzse a kapcsos zárójelek között lévő terület, az itt deklarált változók és metódusok az osztály tagjai és tagfüggvényei. Egy osztály példányosításakor ezeknek a tagjainak foglalunk helyet a memóriában minden egyes új objektum létrejötékor(a tagfüggvényekkel ez nem így van, azok minden csak egy példányban vannak jelen). Mindkét nyelvben megadhatunk láthatósági kulcsszavakat az osztály deklarációja előtt. A public kulcsszóval nyilvánossá teszük az osztályunkat, így más osztályok is hozzáférhetnek a benne lévő változókhöz és függvényekhez. Ez akkor lehet káros ha másokkal dolgozunk egy projekten és az ő osztályaik véletlen vagy direkt is belekavarhatnak a miénkbe. Ennek kivédésére használhatunk privát előtagot. Ekkor a változókat és metódusokat csak az adott osztály használhatja. A kettő közötti félnyilvános osztályokat protectednek hívjuk, őket a szülő osztályból is el lehet érni. Talán a legnagyobb különbség a két nyelv között, hogy míg Javaban muszáj egy osztálynak léteznie aminek neve azonos a fájl nevével, C++-ban ilyenre nincs szükség, akár osztályok nélkül is írhatunk működő kódot, bár ez elveszi az egész OOP lényegét.

2.Hét: Öröklődés, osztályhierarchia. Polimorfizmus, metódustúlterhelés. Hatáskörkezelés. A bezárási eszközrendszer, láthatósági szintek. Absztrakt osztályok és interfések.

Az adatabsztraktió (elvonatkoztatás) szerint le kell egyszerűsíteni a valóságból átemelt objektumainkat, és csak azokkal az információval dolgozzunk amik relevánsak számunkra. Például egy program megírásához nem szükséges tudnunk minden függvény működését csak azt, hogy milyen adatokat vár el és mit ad vissza. Erre példa a metódusok és osztályok láthatósága.

Az objektum orientált programozás lényeges paradigmája az adatabsztraktió mellett az öröklődés és a hozzá kapcsolódó polimorfizmus. Ennek rendje és módja szerint mind a két nyelvben megtalálható, nagyon kis eltérésekkel. Javaban a származtatandó osztály neve után írt extends kulcsszóval jelezzük, hogy melyik osztályt szeretnénk bővíteni, ez C++-ban egy kettősponttal történik. A bővített osztályt gyereknek, összeségüket leszármazottaknak; az eredeti osztályt szülőnek, összeségüket pedig ősöknek nevezzük. Ha egy osztály származtatunk az eléri az ősei minden public és protected tagját, a privát adatok is inicializálódnak, őket viszint csak olyan metódusokon keresztül érjük el amik public vagy protected láthatóságúak. Az öröklődés egy is-a kapcsolat. Mindkét nyelv támogatja az egyszeres öröklést, ami szerint egy osztálynak legfeljebb egy őse és bármennyi leszármazottja lehet. Viszont a C++ engedi a többszörös öröklést is, amiben egy osztály több alaposztályból származtatunk. Ennek értelmezése és kezelése nehéz, ezért főleg az egyszeres öröklést alkalmazzuk, ami egy átlátható fa hierarchiát hoz létre.

A gyerek osztály minden látható változóval és metódussal bír mint a szülője, viszont azért bővítettük az osztályt mert ezeket változtatni vagy specifikálni szeretnénk. Szóval azt, hogy egy változó nem csak a deklarált, hanem a származtatott osztályokra is hivatkozhat polimorfizmusnak nevezzük. Felüldefiniálásnak nevezzük, amikor egy metódust az osztály örökölni egy definiciót az ősétől de ad rá saját definiciót. Osztálydefiníciók sose lehetnek felüldefiniálva. C++-ban csak akkor lehet felüldefiniálni, ha az el van látva a virtual kulcsszóval, Javaban az nem problémi mivel ott minden metódus virtuális.

Az interfések olyan osztályokon kívül értelmezett referenciák, amik már C-ben is léteztek protokol néven. Itt nincsenek meg sem a változók sem metódusok implementációi, csak a deklarációjuk. Az interfész egy olyan felület a Javaban, ahol el lehet vonatkoztatni az implementációtól és a tesztelésre koncentrálni. Az interfésekkel is jelen van az öröklődés és kiterjesztésnek nevezzük. Viszont különbség az osztályok öröklődésétől, hogy lehet több ősinterfész egy interfésznek és nincsen ősinterfész amiből az összes származik.

12. fejezet

Helló, Arroway!

12.1. OO szemlélet

Megoldás forrása:

A polárgenerátor C++ os változata 3 fájból áll össze: polargen.h.polargen.cpp és polargenteszt.cpp. A polatgen headerjében deklaráljuk a destruktort és a konstruktort, amely példányosításkor igazra állítja a nincsTarolt változónkat és a randomizáló függvényünket igazán véletlenszerűvé tesszük a rendszeridővel. Majd prototype-oljuk a kovetkezo() függvényt és deklaráljuk a privát nincsTarolt és tarolt változókat.

A polargen.cpp a fő részünk, itt írjuk meg az előző kovetkezo() függvény kódját. A kovetkezo() tagfüggvényben először megnézzük, hogy van-e eltárolva számunk, ha nincs akkor kiszá- mítunk kettőt: egyiket visszaadjuk a usernek, a másikat elrakjuk a tárolt változóba és a nincstárolt értékét hamisra állítjuk; ha van eltárolt számunk akkor azt visszaadjuk és a nincsTárolt változót igazra állítjuk.

A polarteszt.cpp-benpéldányosítjuk a polatgen osztályt pg néven és forral csinálunk 10 példát a kimenetre. A java megfelelője nagyon hasonló a programnak, bár ott egy fájlon belül tartjuk a forráskódot.

```
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;

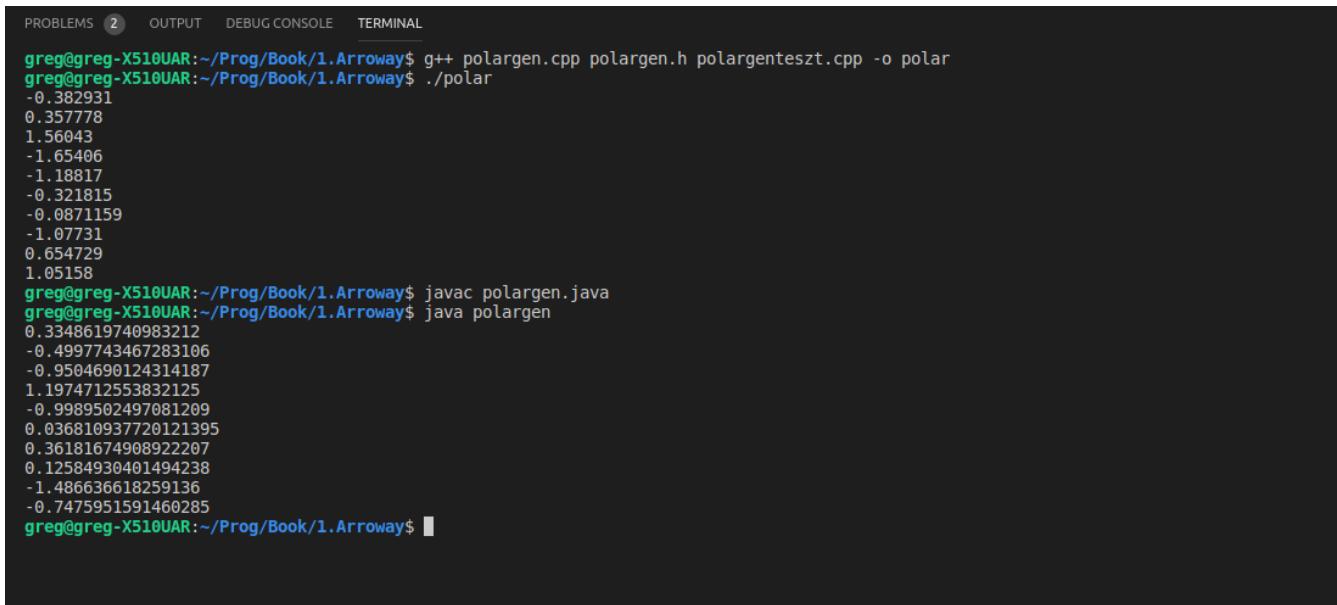
        return r * v1;
}
```

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

Ez nagyban megegyezik ahhoz ahogyan a JDK Random.javabán a Sun programozói csinálták:

```
public double nextGaussian() {
    if (haveNextNextGaussian) {
        haveNextNextGaussian = false;
        return nextNextGaussian;
    } else {
        double v1, v2, s;
        do {
            v1 = 2 * nextDouble() - 1;      // between -1.0 and 1.0
            v2 = 2 * nextDouble() - 1;      // between -1.0 and 1.0
            s = v1 * v1 + v2 * v2;
        } while (s >= 1 || s == 0);
        double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
        nextNextGaussian = v2 * multiplier;
        haveNextNextGaussian = true;
        return v1 * multiplier;
    }
}
```

C++ és Java verzió is lefut:



```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
greg@greg-X510UAR:~/Prog/Book/1.Arroway$ g++ polargen.cpp polargen.h polargenteszt.cpp -o polar
greg@greg-X510UAR:~/Prog/Book/1.Arroway$ ./polar
-0.382931
0.357778
1.56043
-1.65406
-1.18817
-0.321815
-0.0871159
-1.07731
0.654729
1.05158
greg@greg-X510UAR:~/Prog/Book/1.Arroway$ javac polargen.java
greg@greg-X510UAR:~/Prog/Book/1.Arroway$ java polargen
0.3348619740983212
-0.4997743467283106
-0.9504690124314187
1.1974712553832125
-0.9989502497081209
0.036810937720121395
0.36181674908922207
0.12584930401494238
-1.486636618259136
-0.7475951591460285
greg@greg-X510UAR:~/Prog/Book/1.Arroway$
```

12.2. Homokózó

Megoldás forrása:

12.3. „Gagyi”

Megoldás forrása:

A Gagyi.java programban lévő while ciklus csak akkor lesz igaz ha a két változó értéke megegyezik de maguk az objektumok más helyen vannak a memóriában. Ennek tesztelésére három példa kódcsipetet csinálunk az Integer típus felhasználásával.

Amikor egy Integer típusú változónak értéket adunk -128 és 127 között azt a Java egy kész poolból veszi ki és nem csinál új objekteket. Így, mint az első példából is látható, ha ezek között veszünk fel értéket az nem aktiválja a végtelen ciklust.

```
Integer x = 110;
Integer t = 110;

while (x <= t && x >= t && t != x);
    //false
```

Viszont egy olyan esetben, ha a poolon kívülről szeretnénk értéket adni két új objektum jön létre, amelyeknek bár ugyanaz az értékük más memóriacímen vannak eltárolva, így a végtelen ciklus lefut.

```
Integer x = 130;
Integer t = 130;

while (x <= t && x >= t && t != x);
    //true
```

Ez a folyamat bizonyítható a JDK Integer forrásából, amiben látható hogyha az értékünk a poolon kívül van (nem Cache.low és Cache.high között) akkor a return new Integer(i); fog lefutni amikor csináljuk a két Integert.

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

A harmadik példában pedig megkerüljük az egész poolozós témat és egyből használjuk az előző kódcsipet végén lévő new Integer() konstruktort.

```
Integer x = new Integer(110);
Integer t = new Integer(110);

while (x <= t && x >= t && t != x);
    //true
```

12.4. Yoda

Megoldás forrása:

A Yoda féle feltétel írás a Star Wars kis zöld jedi mesteréről kapta nevét, ami sok más mellett fura szintaxisú beszédével tűnik ki, amiben az alanyok és állítmányok helye felcserélődik. Ezt teszi esetünkben most is, ugyanis ha Yoda condition-ban írunk, akkor az összehasonlításokban a két szereplő tagok megcseréljük. Ennek előnye, hogy nem változik lesz elől, aminek null is lehet az értéke és így NullPointerException-t is dobhat. Ez akkor következik be, ha egy metódust null értékkel hívunk meg. Hátránya viszont, hogy a kód nehezebben olvasható, hiszen a megszokott balról jobbra olvasást megbontja.

```
String yoda = null ;
if(yoda.equals("Erő"))
    System.out.println("Működni én nem fogok");

if("Erő".equals(yoda))
    System.out.println("Működni én fogok");
```

```
(base) greg@greg-X510UAR:~/Prog/Book/1.Arroway$ javac Yoda.java
(base) greg@greg-X510UAR:~/Prog/Book/1.Arroway$ java Yoda
Exception in thread "main" java.lang.NullPointerException
        at Yoda.main(Yoda.java:6)
(base) greg@greg-X510UAR:~/Prog/Book/1.Arroway$ javac Yoda.java
(base) greg@greg-X510UAR:~/Prog/Book/1.Arroway$ java Yoda
(base) greg@greg-X510UAR:~/Prog/Book/1.Arroway$ █
```

12.5. Kódolás from scratch

Megoldás forrása:

A BBP algoritmus Segítségével kiszámolhatjuk a pi-nek a d+1-dik jegyének a hexadecimális értéket úgy, hogy a megelőző jegyek nem kell kiszámolni. Kezdetben létrehozunk egy osztályt a PiBBP algoritmusnak. A BBP algoritmus alapján: $\{16^d \text{Pi}\} = \{4 * \{16^d S1\} - 2 * \{16^d S4\} - \{16^d S5\} - \{16^d S6\}\}$ A PiBBP public osztály argumentumaként bekérünk egy d számot, ami d+ helyértéket adja meg. Inicializáljuk a számoláshoz szükséges változókat. A d16pi-nek a 0.0d kezdő értékét adunk, az S1,S4,S5,S6 számoknak pedig a d16Sj függvényel számoljuk ki az értékét. d16Pi-t kiszámoljuk a BBP algoritmusra szerint és

kivonjuk a lefelé kerekített értékét. Példányosítjuk a StringBuffert sb néven, ami egy üres stringet hoz létre és ezt bővítiük számonként. A hexaJegyek tömbben tároljuk 16-os számrendszer betű tagjait, mivel 10 fölött A-F számok is becsatlakoznak. Aztán indítunk egy while ciklust, ami addig megy amíg a d16Pi értéke nullával egyenlő nem lesz. Majd a d16Pi-t lefelé kerekítjük és belekényszerítjük egy int jegy változóba. Ha a szám 10-nél kisebb szimplán koncatenáljuk az sb-hez, viszont ha 10-nél nagyobb, akkor kivonunk belőle 10, így megkapjuk a hexaJegyek tömbből a neke megfelelő betűt. Például ha 16-ot kapunk eredményül, akkor abból 10-et kivonva a tömb 6. indexű elemét kapjuk meg, a mi F. Az osztály végén pedig toString-el stringgé konvertáljuk és betöljtük a d16PiHexaJegyek változóba.

```
double d16Pi = 0.0d;

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

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

d16Pi = d16Pi - StrictMath.floor(d16Pi);

StringBuffer sb = new StringBuffer();

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

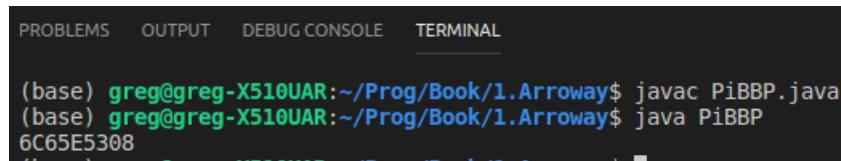
while(d16Pi != 0.0d) {

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

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

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

d16PiHexaJegyek = sb.toString();
```



The screenshot shows a terminal window with the following content:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) greg@greg-X510UAR:~/Prog/Book/1.Arroway$ javac PiBBP.java
(base) greg@greg-X510UAR:~/Prog/Book/1.Arroway$ java PiBBP
6C65E5308
(base) greg@greg-X510UAR:~/Prog/Book/1.Arroway$ █
```

13. fejezet

Helló, Liskov!

13.1. Liskov helyettesítés sértése

Megoldás forrása:

A liskov elv azt jelenti, hogy ha származtatunk egy osztályt annak behelyettesíthetőnek kell lennie minden olyan esetben ahol az őse is használható. Ezt jelölésekkel a következőképpen tudjuk leírni: legyen S T-nek egy leszármazottja, ekkor minden T típusú objektum legyen helyettesíthető S objektumokkal.

Ezt a gyakorlatban többféle módon is megelőzhetjük. Először is figyelnünk kell, hogy ne adjunk meg szorosabb megkötéseket a bemenetre a gyerek osztályban mint ősénél, mivel ha behelyettesítenénk ez kivételt okozhat. Ugyanúgy a kimeneti típus sem téphet el a kettő közötti ügynak azon okokból. Másodszor pedig figyelnünk kell az osztályok logikájának helyes megtervezésre, erre látunk is egy példát.

```
class Madar{
    void repul(){
        System.out.println("Minden madár repül");
    }
}

class Sas extends Madar{
    @Override
    void repul(){
        System.out.println("A sas repül");
    }
}

class Pingvin extends Madar{
}

class Liskov{
    public static void main(String[] args) {
        Madar madar=new Madar();
        madar.repul();
        madar=new Sas();
```

```
        madar.repul();
        madar=new Pingvin();
        madar.repul();
    }
}
```

ebben a programban T az a madár és az S-ek a pingvin és a sas osztályok. Sérül a liskov elv, ugyanis így szerintünk a pingvin képes repülni. A probléma az, hogy feltételeztük, hogy minden madár tud repülni. A megoldás az, ha csinálunk egy külön interface-t, amely a repülés tulajdonságot jelenti,bár ekkor a madár osztályt is interface-ve kell alakítanunk, mivel a többszörös öröklődés nem lehetséges javaban.

13.2. Szülő-gyerek

Megoldás forrása:

A polimorfizmus szabálya szerint egy gyermek osztály örököl minden olyan metódust ,ami deklarálva van a szülő osztályban. Ez viszont fordítva nem érvényes, még akkor sem ha maga a szülő referencia egy gyerek objektumra hivatkozik. Ezt a szülő-gyermek kapcsolatot fugjuk bebizonyítani Java és C++ nyelveken.

Példáinkban lesz egy Szülő és egy Gyerek osztály, mindegyik egy saját metódussal ami kiírat valamilyen szöveget a konzolra. A Gyerek osztályt a Szülőből származatjuk és adunk nekik egy-egy print metódust. A Mainben példányosítjuk a szülőn keresztül példányosítjunk egy Szülő és egy Gyerek osztályt. Így látjuk, hogy nem tudja meghívni a Gyerek print metódusát sem az sz1 (mondjuk neki tényleg nincs esélye) sem az sz2, ami a Gyerek osztályra hivatkozik.

```
class Szulo
{
    public void printParent()
    {
        System.out.println("Én vagyok a szülő metódusa!");
    }
}

class Gyerek extends Szulo
{
    public void printGyerek()
    {
        System.out.println("Én vagyok a gyerek metódusa!");
    }
}

public class SzuloGyerek
{
    public static void main(String[] args)
    {
        Szulo sz1 = new Szulo();
```

```

Szulo sz2 = new Gyerek();

Gyerek gy1 = new Gyerek();

sz1.printParent();

// A szülő osztály semmiképpen nem éri el a subclass metódusait
// sz1.printGyerek();
// sz2.printGyerek();

gy1.printParent();
gy1.printGyerek();
}

}

```

13.3. Anti OO

Megoldás forrása:

A BBP kódját java nyelven már kebeszéltük az előző fejezetben, ez írtuk át C, C++ és C# nyelvekre. Legegyszerűbb egyértelműen C#-ra volt, csak néhány függvény nevét kellett megváltoztatni és még futott is a program. C-nél az objektum orientáltság elhagyása miatt több időbe telt. C#-os kód linux futtatása .Net Core 3 és Visual Studio kominációjával volt futtatva, ez a windowsol programozási nyelvnek virtuális gépet generál ,ami közel nem olyan fejlett mint a java-é, ez meg is fog látszani a teszten.

Jegy	C	C++	C#	Java
10^6	00:01.732	00:01.724	00:01.825	00:01.554
10^7	00:20.109	00:19.993	00:21.463	00:18.473
10^8	04:56.586	04:38.099	04:63.636	03:41.419

13.1. táblázat. Mérési eredmény

Apropó, ami a tesztet illeti meglepő eredmények születtek. Én a C-t tiipeltem volna meg első heylezettnek a hardver közeli felépítése miatt de végül a Java győzedelmeskedett mindegyik fordulóban. Úgy tűnik a Java interpretere már annyira optimalizált, hogy az gyorsaság terén a többi magas szintű programozási nyelvet kenterbe veri. A lista másik végén nem meglepően pedig a C# szerepel, a linuxos környezet most nem neki kedvezett.

A C# kódja a Java-éhoz, a C++ kódja pedig a C-éhez nagyon hasonlít és mivel a Java programkódját már láttuk, a spamelést elkerülve most csak egy C kódcsipetet illesztek be.

```

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

```

```
long n16modk (int n, int k)
{
    long r = 1;

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

    for (;;)
    {
        if (n >= t)
        {
            r = (16 * r) % k;
            n = n - t;
        }

        t = t / 2;

        if (t < 1)
            break;

        r = (r * r) % k;
    }

    return r;
}

double d16Sj (int d, int j)
{
    double d16Sj = 0.0;
    int k;

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

    /*
        for(k=d+1; k<=2*d; ++k)
        d16Sj += pow(16.0, d-k) / (double) (8*k + j);
    */

    return d16Sj - floor (d16Sj);
}

int main ()
{
    double d16Pi = 0.0;
```

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

int jegy;
int d;

clock_t delta = clock ();

for (d = 100000000; d < 100000001; ++d)
{
    d16Pi = 0.0;

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

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

    d16Pi = d16Pi - floor (d16Pi);

    jegy = (int) floor (16.0 * d16Pi);

}

printf ("%d\n", jegy);
delta = clock () - delta;
printf ("%f\n", (double) delta / CLOCKS_PER_SEC);
}
```

13.4. Ciklomatikus komplexitás

Megoldás forrása:

A ciklomatikus komplexitás egy program bonyolultságát méri a programkód alapján egy számértékben ki-
fejezve. Egy lineáris program komplexitása e módon 1 lenne, ha bele írunk egy if szerkezetet azzal egy
elágazást hozunk létre és a komplexitás 2-re növekszik. A módszer ilyen szinten növeli a komplexitást
 minden elágazás, ciklus, try catch, logikai operátor, a ?: operátor és a ?. operátor esetén.

Manuálisan ez a $M = E - N + 2P$ képlettel számolható ki, ahol

A polárgen ciklomatikus komplexitásának kiszámításához a pom.xml-be beillesztjük a következő ncss plugin-t:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>javancss-maven-plugin</artifactId>
    <version>2.1</version>
</plugin>
```

Majd az mvn site és mvn javancss:report futtatásával a projekt könyvtárában kiadja a kiszámolt komplexitást egy html fájl formájában:

Methods

Methods

[package] [object] [method] [explanation]

TOP 30 Methods containing the most NCSS.

Methods	NCSS	CCN
com.mycompany.app.App.kovetkezo()	16	4
com.mycompany.app.App.main(String[])	4	2

Averages.

Program NCSS	NCSS average	CCN average	Javadocs average
27.00	10.00	3.00	0.00

IV. rész

Irodalomjegyzék

13.5. Általános

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

13.6. C

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

13.7. C++

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

13.8. Lisp

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

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

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