

RÁCZ ANDRÁS BHAX KÖNYVE

Egy egyetemre járó programozást tanuló hallgató könyve.

Ed. BHAX, DEBRECEN,
2019. május 9, v. 1.0.1

Copyright © 2019 RÁCZ ANDRÁS

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

Copyright (C) 2019, András RÁCZ, raczandras0204@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". The owner of the license is Dr. Norbert Bátfaí, whose book sample inspired my book.

<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> Rácz András BHAX könyve		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Rácz, András	2019. szeptember 15.	

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
0.1.0	2019-02-22	Saját fork létrehozása, alapvető beállítások, draft kivétele, Turing fejezet szemrevételezése.	raczandras
0.1.1	2019-02-23	Turing csokor elkezdése, a források elkészítése.	raczandras
0.2.0	2019-03-01	A Turing csokor teljesen elkészült.	raczandras

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.2.1	2019-03-02	A Chomsky csokor feladatainak tanulmányozása, a feladatok megoldásának elkezdése.	raczandras
0.3.0	2019-03-8	A Chomsky csokor feladatai elkészültek.	raczandras
0.3.1	2019-03-09	A Caesar csokor feladatainak tanulmányozása.	raczandras
0.4.0	2019-03-13	A Caesar csokor feladatai elkészültek.	raczandras
0.4.1	2019-03-15	Elkezdődik a Mandelbrot forradalom.	raczandras
0.5.0	2019-03-22	A Mandelbrot forradalom sikerrel zárul, a feladatok elkészültek.	raczandras
0.5.1	2019-03-23	A Welch csokor elkezdése.	raczandras
0.6.0	2019-03-31	A Welch csokor feladatai elkészültek.	raczandras
0.6.1	2019-04-04	A Conway csokor feladatainak a tanulmányozása, munka elkezdése.	raczandras
0.6.9	2019-04-09	A Conway csokor labor feladatai elkészültek.	raczandras
0.7.0	2019-04-19	A Conway csokor előadás feladata elkészült.	raczandras
0.8.0	2019-04-21	A Schwarzenegger csokor első feladata kész, a másik két feladat passzolva az SMNIST kutatásra hivatkozva.	raczandras
0.8.1	2019-04-22	A Gutenberg csokor hozzáadása, az olvasónapló elkezdése.	raczandras

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.8.2	2019-04-23	A Chaitin csokor tanulmányozása.	raczandras
0.8.3	2019-04-29	Kész az olvasónapló.	raczandras
0.9.0	2019-04-30	A Chaitin csokor elkészül.	raczandras
0.9.1	2019-05-04	Nyelvtani hibák javítása, a könyv áttekintése	raczandras
1.0.0	2019-05-04	A könyv elkészült	raczandras
1.0.1	2019-05-09	Utólagos módosítások	raczandras

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?	5
2.3. Változók értékének felcserélése	6
2.4. Labdapattogás	7
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	7
2.6. Helló, Google!	7
2.7. 100 éves a Brun tétel	8
2.8. A Monty Hall probléma	8
3. Helló, Chomsky!	10
3.1. Decimálisból unárisba átváltó Turing gép	10
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	11
3.3. Hivatkozási nyelv	11
3.4. Saját lexikális elemző	11
3.5. l33t.1	12
3.6. A források olvasása	12
3.7. Logikus	14
3.8. Deklaráció	14

4. Helló, Caesar!	17
4.1. double** háromszögmátrix	17
4.2. C EXOR titkosító	19
4.3. Java EXOR titkosító	19
4.4. C EXOR törő	20
4.5. Neurális OR, AND és EXOR kapu	22
4.6. Hiba-visszaterjesztéses perceptron	22
5. Helló, Mandelbrot!	24
5.1. A Mandelbrot halmaz	24
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	25
5.3. Biomorfok	26
5.4. A Mandelbrot halmaz CUDA megvalósítása	27
5.5. Mandelbrot nagyító és utazó C++ nyelven	28
5.6. Mandelbrot nagyító és utazó Java nyelven	28
6. Helló, Welch!	30
6.1. Első osztályom	30
6.2. LZW	30
6.3. Fabejárás	31
6.4. Tag a gyökér	32
6.5. Mutató a gyökér	32
6.6. Mozgató szemantika	33
7. Helló, Conway!	35
7.1. Hangyaszimulációk	35
7.2. Java életjáték	37
7.3. Qt C++ életjáték	38
7.4. BrainB Benchmark	39
8. Helló, Schwarzenegger!	40
8.1. Szoftmax Py MNIST	40
8.2. Mély MNIST	41
8.3. Minecraft-MALMÖ	42

9. Helló, Chaitin!	44
9.1. Iteratív és rekurzív faktoriális Lisp-ben	44
9.2. Gimp Scheme Script-fu: króm effekt	45
9.3. Gimp Scheme Script-fu: név mandala	46
10. Helló, Gutenberg!	48
10.1. Juhász István - Magas szintű programozási nyelvek 1 olvasónaplója	48
10.2. Kernighan és Richie olvasónaplója	49
10.3. Benedek Zoltán, Levendovszky Tihamér - Szoftverfejlesztés C++ nyelven olvasónaplója	50
III. Második felvonás	52
11. Helló, Berners-Lee!	54
11.1. Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven	54
11.2. Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II	54
11.3. Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba	54
12. Helló, Arroway!	55
12.1. OO szemlélet	55
12.2. Homokozó	55
12.3. Gagyí	55
12.4. Yoda	56
12.5. Kódolás from scratch	56
13. Helló, Liskov!	57
13.1. Liskov helyettesítés sértése	57
13.2. Szülő-gyerek	57
13.3. Anti OO	57
13.4. deprecated - Hello, Android!	58
13.5. Hello, Android!	58
13.6. Hello, SMNIST for Humans!	58
13.7. Ciklomatikus komplexitás	58

14. Helló, Mandelbrot!	59
14.1. Reverse engineering UML osztálydiagram	59
14.2. Forward engineering UML osztálydiagram	59
14.3. Egy esettan	59
14.4. BPMN	60
14.5. BPEL Helló, Világ! - egy visszhang folyamat	60
14.6. TeX UML	60
15. Helló, Chomsky!	61
15.1. Encoding	61
15.2. OOCWC lexer	61
15.3. l334d1c4	61
15.4. Full screen	62
15.5. Paszigráfia Rapszódia OpenGL full screen vizualizáció	62
15.6. Paszigráfia Rapszódia LuaLaTeX vizualizáció	62
15.7. Perceptron osztály	62
16. Helló, Stroustrup!	63
16.1. JDK osztályok	63
16.2. Másoló-mozgató szemantika	63
16.3. Hibásan implementált RSA törése	63
16.4. Változó argumentumszámú ctor	64
16.5. Összefoglaló	64
17. Helló, Gödel!	65
17.1. Gengszterek	65
17.2. C++11 Custom Allocator	65
17.3. STL map érték szerinti rendezése	65
17.4. Alternatív Tabella rendezése	66
17.5. Prolog családfa	66
17.6. GIMP Scheme hack	66
18. Helló, !	67
18.1. FUTURE tevékenység editor	67
18.2. OOCWC Boost ASIO hálózatkézelése	67
18.3. SamuCam	67
18.4. BrainB	68
18.5. OSM térképre rajzolása	68

19. Helló, Schwarzenegger!	69
19.1. Port scan	69
19.2. AOP	69
19.3. Android Játék	69
19.4. Junit teszt	70
19.5. OSCI	70
20. Helló, Calvin!	71
20.1. MNIST	71
20.2. Deep MNIST	71
20.3. CIFAR-10	71
20.4. Android telefonra a TF objektum detektálója	72
20.5. SMNIST for Machines	72
20.6. Minecraft MALMO-s példa	72
IV. Irodalomjegyzék	73
20.7. Általános	74
20.8. C	74
20.9. C++	74
20.10Lisp	74

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

Számomra a programozás az önkifejezés egy formája.

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Turing/ciklus.c>

Egy magot 100%-on dolgoztatni nem egy nagy kihívás, hiszen ha egy szimpla while ciklust megírunk, az alapvetően így működik. Egy magot 0%-on dolgoztatni sem egy egetrengető kihívás, viszont itt már kell minimálisan gondolkodni. De hamar rájövünk, hogy a sleep(x) parancs kiadásával x másodpercig nem használja a processzort a program. Kicsit érdekes, hogy ha nincs parancs a cikluson belül, vagyis nincs mit tenni, akkor 100%-on dolgozik a processzor. Ez azért történik, mert az operációs rendszer azt hiszi, hogy van elvégzendő feladat, ezért a programnak adja közel az összes processzoridőt. Viszont az összes magot 100%-on dolgoztatni már feladta a leckét. Először megpróbáltam a thread paranccsal kezdeni valamit, de az túl bonyolultnak tűnt egy ilyen feladathoz. Majd Besenci Renátó adott egy tippet, miszerint az OpenMp-t kellene tanulmányoznunk a feladat megoldásához. Innen pedig már pár fórumon és StackOverflow lapon keresztül egyenes út vezetett a győzelemhez.

A programot roppant egyszerű használni. Ha egy magot szeretnénk 100%-ban dolgoztatni, akkor semmit nem kell módosítani, szimplán csak le kell fordítani és futtatni.

Ha egy magot szeretnénk 100%-ban dolgoztatni, akkor vegyük ki a // -t a

```
//sleep(1)
```

függvényhívásból.

Ha pedig az összes magot szeretnénk 100%-ban dolgoztatni, akkor ugyanúgy a // -t kell kitörölni a következő helyről:

```
#pragma omp parallel while
```

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:

Nem tudunk olyan programot írni, ami minden más programról eldönti, hogy van-e benne végtelen ciklus. Mivel, ha tudnánk, akkor már valószínűleg lett volna olyan ember, aki ezt a programot megírja.

De tegyük fel, hogy megírjuk ezt a programot, aminek a neve legyen eldöntő. Annak a programnak a neve, amelyről el kell dönteni, hogy van-e benne végtelen ciklus, legyen eldöntendő. Nyilván az eldöntő bemeneti argumentuma lesz az eldöntendő. Ahhoz, hogy eldöntő megállapítsa, hogy van-e eldöntendőben végtelen ciklus, futtatnia kell az eldöntendő kérdéses részleteit. Ekkor ha az eldöntendő programban nincs végtelen ciklus, eldöntő hamissal tér vissza, ami azt jelenti, hogy nincs eldöntendőben végtelen ciklus.

Azonban ha az eldöntendő programban tényleg van egy végtelen ciklus, és azt eldöntő futtatja, hogy megbizonyosodjon róla, akkor eldöntő maga is egy végtelen ciklussá válik. Éppen ezért eldöntő sose fog igazsággal visszatérni, mert minden ilyen esetben ő is le fog fagyni.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Turing/csere.c>

Ez egy egyszerű matematikai/logikai feladat, amit ha egyszer megmutatnak az embernek, akkor örökké emlékezni fog rá. Olyan mint a biciklizés, nem lehet elfelejteni.

Még úgy is, hogy logikai utasítások, kifejezések nélkül kell megoldani ezt a problémát, rengeteg lehetőség közül választhatunk. Én itt most kettőt fogok bemutatni.

Az egyik az, hogy összeggel és különbséggel cseréljük fel a két változót a következőképpen:

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

Ha ezt végigvezetjük például az $a=5$ és $b=6$ értékekkel akkor az első lépés után $a=11$ és $b=6$. A második lépés után $a=11$ és $b=6$ a harmadik lépés után pedig $a=6$ és $b=5$

Egy másik lehetőség pedig az, hogy szorzattal cseréljük meg a két változó értékét aminek az alapja hasonló az előző megoldáshoz egy kis módosítással:

```
a = a*b;  
b = a/b;  
a = a/b;
```

Ezekon kívül még vannak módszerek amik megfelelnek a feladat leírásának. Ezek a forrásban megtalálhatóak és a működésük alapja ugyan az mint az előző két megoldásnak.

2.4. Labdapattogás

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

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

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Turing/labda.cpp>

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Egy egyszerű "grafikus" program, ami egy labdának álcázott o betűt mozgat a képernyőn egy egyszerű while ciklus segítségével. Habár maga a program nem hosszú, és nem is túl bonyolult, mégis elég nagy hatással van a kezdő mezei programozóra, hiszen nagyon sok programozást tanulónak (köztük nekem is) az egyik álma egy valódi grafikus felülettel működő program írása, és ez egy nagyon jó kezdet eme cél megvalósításához.

Maga a program két fő részből áll. Az egyik egy függvény, ami a labdát rajzolja ki a konzolra, A másik pedig maga a main.

A main-ben először létrehozunk egy maxX és egy maxY változót, amiket át is adunk a tx és a ty tömbök méretének.

Ezután két for ciklus végigmegy a két tömbön, a második, és az utolsó elemek értéke -1 lesz, a többi elem pedig 1

végül pedig egy while ciklus és a függvény segítségével kiírja a konzolra a labdát.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

Írd 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://github.com/raczandras/progbook/blob/master/src/Turing/bogo2.cpp>

Alapvetően a BogoMips a processzorunk sebességének meghatározásához használatos mértékegység. Azt mondja meg, hogy a számítógép processzora mekkora szóhosszal dolgozik

Ezt a XOR ^ művelet segítségével számolja ki a program, ami a kizáró vagy művelete. Az int értékének 1-et adunk, és addig shifteljük balra, ameddig lehet, vagyis amíg az int értéke 0 nem lesz.

Közben egy másik változóval számoljuk, hogy hányszor shiftelt balra az int, ezzel meghatározva a szóhosszt. Az én esetemben az eredmény 32 lett, ami azt jelenti, hogy az én processzorom szóhossza 32 bit, azaz 4 bájt.

2.6. Helló, Google!

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

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Turing/pagerank.c>

Én a Bevezetés a Programozásba nevű tárgyon már átnézett [PageRank](#) programot vettem alapnak, és azt alakítottam át c++-ból c-re. Ez azonban a vártnál több gondot okozott. De legalább mostmár tudom, hogy amennyiben c++ kódot akarok c-re átírni akkor nem érdemes az `abs()` függvényt használni, mert nem ugyanúgy működik a két nyelven belül ez a függvény. Ez fel is keltette, az érdeklődésemet, hogy miért nem? Kis utánajárás utána a [GeeksforGeeks](#) oldalon meg is találtam a választ, ami szerint C++ nyelven ennek a függvénynek a visszatérési típusa ugyan az, mint a bemeneti típus. Éppen ezért nyugodtan számolhattuk vele a double típusú távolságot. Ezzel szemben C nyelven a visszatérési típus minden esetben int lesz. Éppen ezért lett a végeredmény mind a négy lap esetén 0.25

Ezt a problémát én egy egyszerű if-else szerkezettel oldottam meg. Az eredeti c++ verzióban kiszámolta a függvény a távolságot, és annak az abszolút értékét adta vissza.

Ezzel szemben az én megoldásom megvizsgálja, hogy a távolság negatív-e. Ha nem, akkor szimplán visszaadja az értéket, ha viszont negatív, akkor az eredményt megszorozza -1 el ezáltal pozitív eredményt kapva. És ezt a pozitív értéket adja vissza a függvény.

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 megoldás forrása Bátfai Norbert tulajdona.

Mint tudjuk, léteznek a prímszámok. Ezek olyan számok, amik csak 1-el és önmagukkal oszthatóak. Valamint léteznek az ikerprímek. Ezek pedig olyan prímszámpárok, amiknek a különbsége pontosan 2. Ha minden ikerprím reciprokának az összegének vesszük a sorozatát, akkor ez a sorozat egy számhoz konvergál. Ez a szám a Brun-konstans. Nem tudjuk azt, hogy az ikerprímek száma véges vagy végtelen e, de ez nem okoz gondot, hiszen elvileg ha végtelen se lépi túl az összegük a Brun-konstanst. Na most be kell vallanom, hogy számtalan olyan ember létezik a földön, aki nálam jobban ért a matematikához. Viszont nekem erről egy elég érdekes dolog jutott eszembe, ami nem más, mint Zeno paradoxona. E szerint x utat teszünk meg, hogy elérjük a célunkat. Ezek alapján megteszünk $1/2x$ utat + $1/4x$ utat + $1/8x$ utat + $1/16x$ utat... Ha ezekből képzünk egy sorozatot, az a sorozat 1-hez fog konvergálni, Éppen ezért soha nem érünk el oda, ahova megyünk. Maga a tétel matematikailag helyes, azonban a való életben tudjuk, hogy ez nem így működik.

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

A megoldás forrása Bátfai Norbert tulajdona.

A Monty Hall problémát még középiskolában ismertem meg, sok különböző változata van. Az általam ismert történetben Monty Hall egy műsorvezető volt, akinél a nyertes játékosok választhattak három darab ajtó közül. A háromból két ajtó mögött 1-1 darab kecske, míg a harmadik ajtó mögött egy sportautó volt. A játékos választott egy ajtót, majd Monty Hall, aki tudta, hogy melyik ajtó mögött van az autó, kinyitott egy másik ajtót, ami mögött egy kecske lapult. Ezek után a játékosnak lehetősége volt változtatni a döntésén, vagy maradhatott az eredetileg kiválasztott ajtónál. A kérdés az, hogy mely esetben van több esélye megnyerni az autót? A legtöbb ember azt mondaná, hogy 50-50% esélye van megnyerni az autót, hiszen vagy az egyik ajtó mögött van az autó, vagy a másik mögött. Ekkor persze hiába magyarázzuk, hogy $1/3$ esélye van megnyerni az autót, ha nem vált, és $2/3$ ha vált, a legtöbb embert elég nehéz meggyőzni erről. Ekkor kell kicsit átalakítani a kérdést. Ha van 1 millió ajtó, ebből kiválaszt a játékos 1-et, majd kinyitnak 999,998 ajtót, amik mögött kecske van, akkor melyik esetben van több esélye a játékosnak megnyerni az autót? ilyenkor már a legtöbb ember egyértelműnek tartja, hogy vált, de van olyan ismerősöm, aki még ekkor is azt mondta, hogy 50-50% esélye van megnyerni az autót, ha vált ha nem. Ez a program ennek a játéknak a nyerési eseteit szimulálja. Tízmillió esetből hányszor nyer az, aki mindig vált, és az aki egyáltalán nem vált.

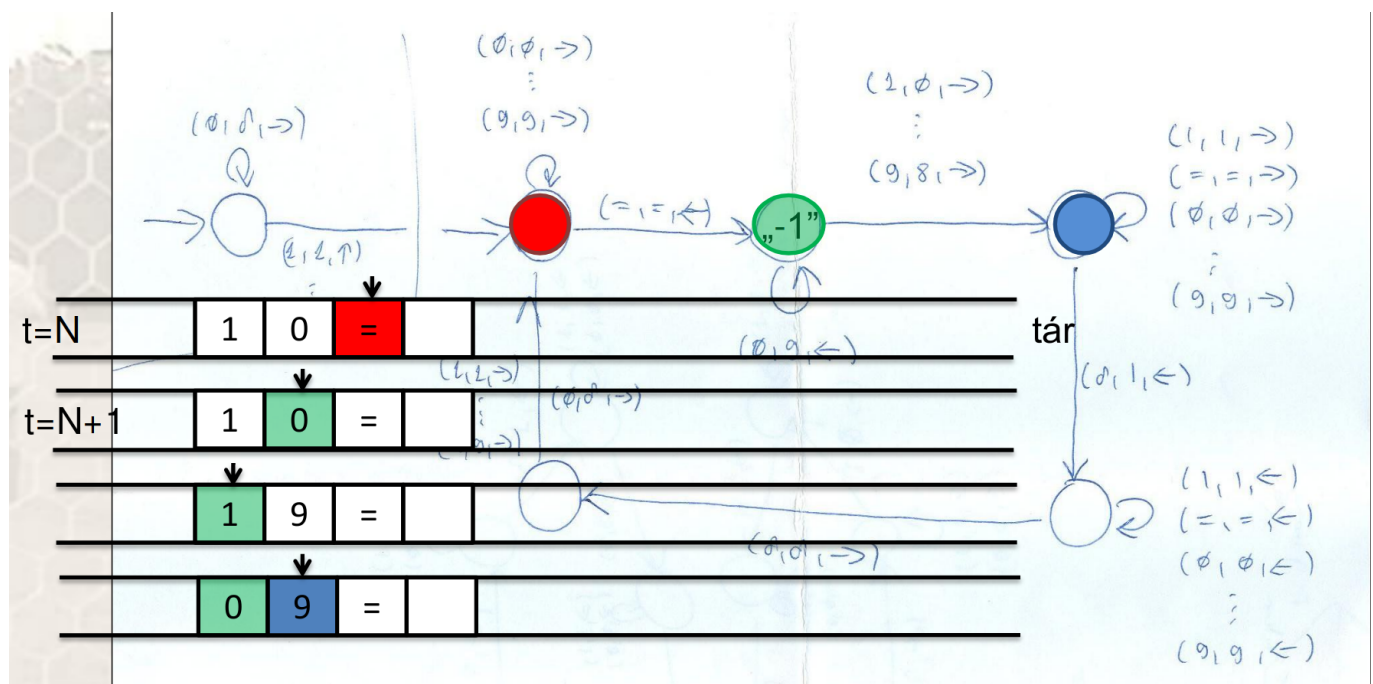
3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:



A kép nem az én tulajdonom, hanem a Magas szintű programozási nyelvek 1 nevű tárgyon kivetített előadás fóliáiról másoltam.

Maga az unáris számrendszer csupa 1-esekből vagy vonalakból áll. Ilyen például, amikor a kezünkön számolunk, vagy amikor az óvodákban a gyerekek a pálcikákkal rakják ki a dolgokat. Pontosan annyi 1-es vagy pálcika, vagy akármilyen jel kell, amennyi maga a szám értéke. például ha az 50-es számot szeretnénk felírni unárisban, akkor 50 darab 1-est kellene leírni egymás után. Éppen ezért, ebben a számrendszerben csak a természetes számokat tudjuk ábrázolni.

Egy ilyen decimálisból unárisba átváltó Turing gépet mutat a fenti ábra is. Ez a gép a kapott szám utolsó számjegyéből von le egyeseket. ha a számjegy 0 akkor 9db-ot von le, ha 5 akkor 4-et. Ezzel együtt a levont

egyesekeket a tába helyezi. Ezt minden számjeggyel megismétli.

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:

Az $a^n b^n c^n$ nyelv nem környezetfüggetlen. Na de először értsük meg, hogy mit is jelent ez a nyelv.

Az $a^n b^n c^n$ tulajdonképpen annyit jelent, hogy n darab a , majd n darab b , majd végül n darab c áll egymás után. Ezek a terminális szimbólumok. A szabály alapján a környezetfüggő nyelveknél bal oldalt csak egy önmagában álló nem-terminális szimbólum állhat. Azonban nem létezik olyan képzési szabály ami alapján ez a szabály teljesíthető, éppen ezért ez a nyelv nem környezetfüggetlen.

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Chomsky/nyelv.c>

A BNF (Backus-Naur-forma) használatával környezetfüggetlen nyelveket lehet leírni. Nagyon sok programozási nyelvek szintaxisai is BNF-ben vannak leírva.

A programozási nyelveknek is van nyelvtana, illetve nyelvtani szabályaik. Az egyik ilyen szabály C89-ben az, hogy a for ciklus fejrésében nem lehetett változót deklarálni, éppen ezért ha a következőképpen szeretnénk lefordítani a fenti programot:

```
gcc -o nyelv nyelv.c -std=c89
```

Akkor a következő hibaüzenetet kapjuk:

```
nyelv.c:3:2: error: 'for' loop initial declarations are only allowed in C99 or C11 mode
```

Ez pontosan leírja nekünk, hogy a for ciklusban deklarálni csak c99 vagy c11 módban lehet.

Éppen ezért ha `-std=c89` helyett nem írunk semmit, vagy `-std=c99`-et írunk, akkor a program gond nélkül lefordul.

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Chomsky/lex.l>

A megoldás forrása [Bátfai Norbert](#) tulajdona.

A lexer-rel szövegelemző programokat lehet generálni az általunk megadott szabályok alapján. A program különböző részeit % jelekkel kell elválasztani egymástól. Itt a numbers változóban fogjuk számolni a valós számok darabszámát. Majd megmondjuk, hogy a digit egy 0 és 9 között lévő számot jelöl. Ezek után jön az a kódrészlet, ami megmondja a lexernek, hogy a valós számokat számolja meg. Végül pedig kiíratjuk a valós számok darabszámát. A futtatáshoz először is telepítenünk kell a lex-et majd a forrásban található programot kell megírni.

Majd azt a következőképp kell lefordítanunk:

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

```
gcc -o lex lex.c -lfl
```

Ezzel magkapjuk a <https://github.com/raczandras/progbook/blob/master/src/Chomsky/lex.c> oldalon található programot, ami a feladat megoldása.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Chomsky/leet.l>

A megoldás forrása [Bátfai Norbert](#) tulajdona.

A leet vagy a saját formájában leírva l337 egy, az internettel együtt elterjedt szleng nyelv, amiben a betűket különböző számokként, és egyéb ASCII karaktereként, a számokat pedig különböző betűkként ábrázoljuk.

Itt is érvényes az a szabály, hogy a program egyes részeit % jelekkel kell elválasztani egymástól. Itt a legelső részben a cipher struktúrában meg vannak adva a karakterek leet formái

A második részben történik az érdemi munka, először a szöveget kisbetűssé alakítja a program, majd pedig végigmegy a szövegen, és minden karaktert a neki megfelelő leet formájú karakterré alakítja át.

A harmadik és egyben utolsó részben található a main amiben a lex meghívása történik.

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 megránézésre, elkapja valamelyiket esetleg a splint vagy a frama?

i.

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

ii.

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

iii.

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

iv.

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

v.

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

vi.

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

vii.

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

viii.

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

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Chomsky/signal.c>

Megoldás videó:

i: Ha kapunk egy INTERACT szignált, akkor a jelkezelő függvénnyel eldöntjük, hogy mihez kezdünk azzal a szignállal, mit reagáljon rá a program.

ii: Egy forciklus ami nullától négyig megy, és a ciklus törzsében lévő művelet elvégzése előtt nő az értéke eggyel.

iii: Szintén egy forciklus, ami szintén nullától négyig megy, viszont itt már a ciklus törzsében lévő műveletek elvégzése után növekszik az értéke.

iv: Egy for ciklus, ami berakja a tomb[i]-edik helyére az i értékénél eggyel nagyobb értéket, és közben i értékét is növeli.

v: Egy for ciklus, ami addig megy, amíg i kisebb mint n, illetve amíg a d és s pointerek értékei megegyeznek.

vi: Kiirunk két, az f nevű függvény által generált számot. az egyik szám az a majd a eggyel megnövelt értékének a feldolgozásából jön létre, míg a másik szám a+1 és a feldolgozásából. Fontos a sorrend.

vii: Szintén két számot írunk ki, az egyik szám az f nevű függvény által feldolgozott a nevű számból előállt érték, a másik pedig a értéke.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

Első értelmezése: Minden számra igaz, hogy létezik tőle nagyobb y prímszám.

Második értelmezése: Minden számra igaz, hogy létezik egy olyan tőle nagyobb y prímszám, hogy $y+2$ is prím.

Harmadik értelmezése: Létezik olyan szám, amitől minden prímszám kisebb.

Negyedik értelmezése: Létezik olyan szám, amitől egyik kisebb szám se prím.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

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

Mit vezetnek be a programba a következő nevek?

- ```
int a; //létrehoz egy egész típusú változót
```
- ```
int *b = &a; //egy pointer, ami a memóriacímére hivatkozik
```
- ```
int &r = a; //egy referencia a-ra
```
- ```
int c[5]; //5 elemű tömb aminek c a neve
```
- ```
int (&tr)[5] = c; //egy tr nevű referencia c-re
```
- ```
int *d[5]; //egy 5 elemű pointerekből álló tömb
```
- ```
int *h (); //Egy egészre mutató mutatót visszaadó függvény
```
- ```
int *(*l) (); //Egy egészre mutató mutatóra mutató mutatót visszaadó ↔  
függvény
```
- ```
int (*v (int c)) (int a, int b) //Függvénytmutató, ami egy egészet ↔
visszaadó függvényre mutató mutatóval visszatérő függvény
```
- ```
int ((*z) (int)) (int, int); //Függvénytmutató, ami egy egészet visszaadó ↔  
függvényre mutató mutatót visszaadó függvényre mutat
```

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Chomsky/dekla.cpp>

Egész:

```
int a;
```

Egészre mutató mutató:

```
int *b = &a;
```

Egész referenciája:

```
int &r = a;
```

Itt fontos megjegyezni, hogy c-ben nincs referencia, ezért ezt a kódcsipetet érdemes g++-al fordítani gcc helyett.

Egészek tömbje:

```
int c[5];
```

Egészek tömbjének referenciája (nem az első elemé):

```
int (&tr)[5] = c;
```

Egészre mutató mutatók tömbje:

```
int *d[5];
```

Egészre mutató mutatót visszaadó függvény:

```
int *h ();
```

Egészre mutató mutatót visszaadó függvényre mutató mutató:

```
int *(*l) ();
```

Egészre visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény:

```
int (*v (int c)) (int a, int b)
```

Függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre:

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

4. fejezet

Helló, Caesar!

4.1. double** háromszögmátrix

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

Tutoráltam: [Duszka Ákos Attila](#)

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Caesar/tm.c>

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Először is egy alap fogalom. Az alsó háromszög mátrixnak ugyanannyi sora van, mint oszlopa. Ezen kívül még egy nagyon fontos tényezője az is, hogy a főátlója felett csak 0 szerepel.

Általában az ilyen mátrixokat, ha tömbökben tároljuk, akkor nincs értelme a nullákat is tárolni a többi, számunkra érdekes elemmel együtt, éppen ezért ezeket nem is tároljuk. Amikor egy ilyen tömböt vissza szeretnénk alakítani az eredeti alakjára, akkor sorfolytonosan írjuk fel az elemeit. ez mindössze annyit jelent, hogy a mátrix első sorába az első elemet írjuk fel, a második sorába a 2. és 3. elemet, és így tovább minden sorban eggyel több elemet írunk fel mint az előző sorban.

Ebben a programban egy ilyen alsó háromszög mátrixot hozunk létre egy

```
double **
```

segítségével. Ez egy pointerre mutató pointer, ami tökéletes a többdimenziós tömbök használatához.

Ezek után a következő kis programrészlet:

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

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

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

```

    if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL ↔
    )
    {
        return -1;
    }
}

```

Ellenőrzi, hogy történt-e valamilyen memóriahiba, (pl. nincs-e tele a memória) és ha történt, akkor -1-el tér vissza.

Ellenkező esetben a program a

```
tm[i][j] = i * (i + 1) / 2 + j;
```

képletet használva feltölti a tömböt. Ezután két egymásba ágyazott for ciklus segítségével kiírja azt.

Ezek után módosítunk a tömb egyes elemein, majd megint kiírjuk őket.

Legvégül pedig a

```

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

free (tm);

```

függvény használatával felszabadítjuk a tömbnek lefoglalt helyet.

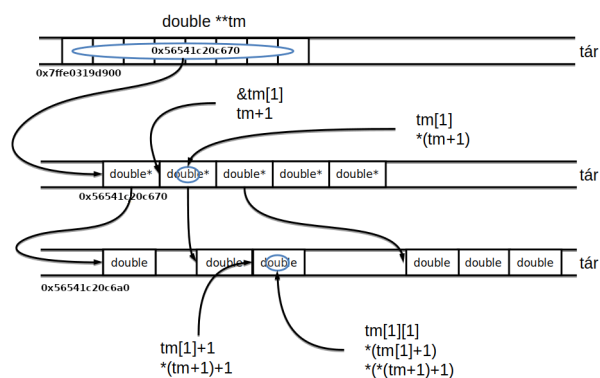
A program futtatásnál a következő memóriacímeket írta ki:

```

./tm
0x7ffe0319d900
0x56541c20c670
0x56541c20c6a0

```

Aminék a jelentése:



A képnek az alapját Bátfai Norbert Biztosította, én azt módosítottam.

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Caesar/exor.c>

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Ez a fajta titkosítás a kizáró vagy műveleten alapul. A megadott kulcs, és a forrásfájl karaktereit kizáró vaggal titkosítva egy szöveget úgy tudunk titkosítani, hogy egy olvashatatlan karaktermasszát kapunk végeredményül. Viszont aki ismeri a kulcsot az ugyan olyan egyszerűen vissza tudja alakítani a szöveget az eredeti alakjára úgy, hogy még egyszer lefuttatja a programot, de a titkosított forrást adja meg titkosítandóként, ezzel visszakapva az eredeti szöveget. Így más nem tudja elolvasni a titkainkat, csak az, aki ismeri hozzá a kulcsot. (legalábbis egyelőre. Két feladattal később már más lesz a helyzet.)

Először is a

```
#define MAX_KULCS 100
#define BUFFER_MERET 256
```

használatával megadjuk a maximális kulcs és buffer méretet. a main osztály első argumentuma a kulcs lesz, míg a második az maga a szöveg, amit titkosítani szeretnénk.

A következő ciklusok használatával:

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

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

program végigmegy a bemeneti adatok (titkosítandó fájl) karakterein, és mindegyiket titkosítja a kulcs használatával, és kiírja a végeredményt.

A program használata: `./exor kulcs <titkosítandó fájl> titkosított fájl`

Erre egy példa: `./exor 12345678 <lista> titkoslista`

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Caesar/exort.java>

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Itt az előző feladatban megírt EXOR titkosítót írjuk át java programozási nyelvre. Ehhez importálnunk kell az input/output streamet, ez ahhoz kell, hogy olvasni tudjuk a bemeneti fájlt, illetve, hogy írni tudjuk a kimeneti fájlt.

A main-ben megpróbáljuk a try-al beolvasni az args (argumentumok) tömbbe azt a fájlt, amit titkosítani szeretnénk, és ha ez nem sikerült, akkor "elkapjuk" a hibát a catch szerkezettel, és kiíratjuk, hogy mi a hiba:

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

        try {

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

        } catch (java.io.IOException e) {

            e.printStackTrace();

        }

    }
}
```

Ha viszont sikerült beolvasni a fájlt, akkor az ExorTitkosító nevű függvényt meghívva előállítjuk a titkosított szöveget. a System.in illetve System.out a bemenő és a kimenő fájlra utalnak.

Először is a függvény átadja a program a kulcs nevű tömbnek a bemenő szöveget, és létrehoz egy buffer nevű tömböt is 256-os mérettel. Erre az EXOR művelethez lesz szükség.

Végül a program egy while-ba épített for ciklus segítségével végigmegy a szövegen, és minden egyes karakternek meghatározza a titkosított verzióját, és kiírja azt a kimeneti fájlba.

4.4. C EXOR törő

Tutoráltam: [George Butcovan](#)

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

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Caesar/tores.c>

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Önnel is előfordult már az, hogy elfelejtette egy EXOR-ral titkosított fájl kulcsát? Ön is akart már kutakodni mások fájljai között, de nem tudott, mert EXOR-ra voltak titkosítva a fájlok?

Ne szenvedjen tovább. Az EXOR törő biztos megoldást nyújt önnek! Csupán annyit kell tudnia, hogy hány karakterből áll a kulcs, és máris használhatja ezt a fenomenális programot. A felhasználó ostobaságaiért és azok jogi következményeiért felelősséget nem vállalunk.

A működése roppant egyszerű. Mivel nem ismerjük a kulcsot, ezért a program az összes lehetséges kombinációt végigpróbálja. A következőkben bemutatott példában a kulcs 8 darab karakterből áll.

Legelőször a program a következő while ciklus:

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

Használatával beolvassa a feltörni kívánt fájlt, majd a maradék helyet a bufferben egy for ciklust használva

```
for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)  
titkos[p - titkos + i] = '\\0';
```

feltölti 0 értékekkel.

Ezek után egy halom (ami jelen esetben 8) for ciklussal:

```
#pragma omp parallel for private(kulcs)  
for (int ii = '0'; ii <= '9'; ++ii)  
for (int ji = '0'; ji <= '9'; ++ji)  
for (int ki = '0'; ki <= '9'; ++ki)  
for (int li = '0'; li <= '9'; ++li)  
for (int mi = '0'; mi <= '9'; ++mi)  
for (int ni = '0'; ni <= '9'; ++ni)  
for (int oi = '0'; oi <= '9'; ++oi)  
for (int pi = '0'; pi <= '9'; ++pi)  
{  
    kulcs[0] = ii;  
    kulcs[1] = ji;  
    kulcs[2] = ki;  
    kulcs[3] = li;  
    kulcs[4] = mi;  
    kulcs[5] = ni;  
    kulcs[6] = oi;  
    kulcs[7] = pi;  
  
    exor_tores (kulcs, KULCS_MERET, titkos, ←  
                p - titkos);  
}
```

Megpróbálja a program előállítani az eredeti szöveget. Azonban több kombináció is ad eredményt, éppen ezért nekünk kell kitalálni, hogy a kapott eredmények közül melyik a helyes. Kis érdekesség, hogy ezek a for ciklusok az összes magot dolgoztatni fogják, ezzel jelentősen lecsökkentve a töréshez szükséges időt.

Ha a kulcs nem 8 karakterből áll, akkor se essünk pánikba! Csupán néhány (pontosan 3) szekcióban kell módosítani a program kódját. Ezek a következők:

Először is a program fejében a

```
[#define KULCS_MERET 8
```

sorban a 8-at át kell írni arra a számra, amennyi karakterből áll a kulcs.

Majd a 70. és 71. sorokban lévő

```
[ printf("Kulcs: [%c%c%c%c%c%c%c%c]\nTiszta szoveg: [%s]\n",  
kulcs[0],kulcs[1],kulcs[2],kulcs[3],kulcs[4],kulcs[5],kulcs[6],kulcs[7],  
    buffer);
```

utasításokban annyi `%c` és `kulcs[n]` legyen, amennyi karakterből áll a kulcs.

Végül pedig az előzőekben már látott `for` ciklus halmon kell módosítanunk úgy, hogy pontosan annyi `for` ciklus, és pontosan annyi `kulcs[n] = xi`; legyen a programban, amennyi karakterből áll a kulcs.

Most hogy ezt mind tudjuk, a programot a következőképpen kell fordítani: **gcc tores.c -fopenmp -o tores -std=c99**

És futtatni: **./tores <titkosfajl**

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

A megoldás forrása Bátfai Norbert tulajdona.

A neurális hálózatokban például a machine learning esetében, A neuronok egy gráfban elhelyezkedve egymással kommunikálnak úgynevezett "Activation function", magyarul Aktivációs függvény segítségével.

Léteznek bemeneti, kimeneti, és rejtett neuronok is.

A bemeneti neuronok kapják meg a bemenetet. Itt több különböző fajta neuront is meg lehet különböztetni. Vannak egybemenetű és több bemenetű neuronok is. Ezeknek a neuronoknak nincs különösebb feldolgozó feladatuk, továbbítják a bemenetet a többi neuronnak.

A kimeneti neuronok amik a környezetnek adják tovább a kapott információt.

A rejtett neuronoknak pedig a bemenete és a kimenete is csakis más neuronokhoz kapcsolódik.

Ezek alapján egy neurális hálónak legalább két rétegből kell állnia. Egy bemenetiből, és egy kimenetiből. Felső határ, azaz hogy a bemeneti és a kimeneti neuronok között hány darab további réteg helyezkedik el, elviekben nincs.

Először minden neuron megkapja a saját bemeneteit, és minden neuron ebből a bemenetből előállít egy úgynevezett súlyozott összeget, és ezt az értéket vezeti végig az aktivációs függvényen. Egy példa lehet az, hogy ha a súlyozott összeg pozitív lesz, akkor az érték 1, míg ha a súlyozott összeg negatív, akkor az érték -1 lesz.

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

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

A megoldás forrása Bátfai Norbert tulajdona.

Perceptronról a Mesterséges Intelligenciák, és a neurális hálók témakörében lehet szó. Ellenőrzi a bemenetet, és egy feltétel alapján eldönti, hogy mi legyen a kimenet, Egy példa:

Van három bemeneti adatunk amikhez pozitív egész számokat várunk. Ha a három bemeneti számból kettő kisebb mint nulla, akkor a kimeneti adat -1 lesz, ha viszont a háromból legalább kettő pozitív szám, akkor a számok összege lesz a kimeneti adat.

Ekkor kimondhatjuk, hogy 1 a hibahatár, mert ekkor még megkapjuk az általunk kért dolgot, viszont ha már kettőt hibázunk akkor már -1 lesz a válasz.

Ezt a hibahatárt szokták finomhangolni. Nagyon magas hibahatárnál kezdenek, és egyre kisebbé teszik egészen addig amig elfogadható a hibák mennyisége.

Persze a mi három bemeneti adatok példánknál nem sokat lehet finomhangolni, de ha több millió bemeneti adatról beszélünk, ott ez egy elég fontos dolog.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Tutorálóm: [Duszka Ákos Attila](#)

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Mandelbrot/mandelbrot.cpp>

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Mielőtt bármihez hozzáfazdenénk egy nagyon fontos információ. Ahhoz, hogy leforduljon a programunk, szükséges a png++. Ezt a legegyszerűbben a **sudo apt install png++** paranccsal lehet megtenni. Most hogy ezt letudtuk, jöhet pár alapvető információ.

A Mandelbrot halmaz lényege (legalábbis számomra) az, hogy komplex számokkal, és egy egyenlettel dolgozik. Azok a számok amelyek kielégítik ezt az egyenletet egy nagyon szép képet alkotnak, ha levetítjük őket egy kétdimenziós síkra. Akit ez bővebben vagy részletesebben érdekel azoknak ajánlom a különböző weboldalakat, én nem fogom tovább boncolgatni, mert én magam sem értem.

A program legelején includeoljuk a png++-t, hiszen nagyrészt ezt fogja használni a program.

```
#include <png++-0.2.9/png.hpp>
```

Ezek után létrehozunk végleges értékeket N-nek és M-nek, valamint megadjuk X és Y lehetséges minimum és maximum értékét is.

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

két egymásba ágyazott for ciklus használatával megadjuk a C, a Z, és a Zuj nevű Komplex számok valós és imaginárius értékét. Ezek korábban lettek létrehozva a mainen belül, és a Komplex nevű struktúrához tartoznak.

```
struct Komplex
{
    double re, im;
};
```

```
struct Komplex C, Z, Zuj;
```

Végül pedig a `GeneratePNG(tomb)` nevű függvény használatával a program legenerálja a PNG fájlt. pixelről pixelre.

A programot a következőképpen tudjuk fordítani: **g++ mandelbrot.cpp -lpng16 -o mandelbrot** Futtatni pedig a szokásos módon **./mandelbrot** paranccsal tudjuk.

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

Tutorálóm: [Duszka Ákos Attila](#)

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Mandelbrot/mandelbrotkomplex.cpp>

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az **UDPROG** repóban.

Ebben a feladatban a végeredmény ugyan az kellene hogy legyen, mint az előző feladatban. Illetve azóta még a mandelbrot halmaz lényege sem változott, ezért azt nem írnám le újra.

A png++ ebben az esetben is kelleni fog, így ha nincs leszedve, akkor pillants az előző feladat magyarázatára, ahol megtalálod a szükséges dolgokat ahoz, hogy le tudjon fordulni a program.

Ebben az esetben az `std::complex` osztályt fogjuk használni a program megvalósításához. Ez az osztály, ahogy a neve is utal rá, a komplex számok kezelése miatt jött létre.

A program által használt függvényei a következők:

A `real(C)` a komplex szám valós részét határozza meg.

A `imag(C)` a komplex szám képzetes részét határozza meg.

Legelőször a program

```
#include <png++-0.2.9/png.hpp>
#include <complex>
```

beincludeolja a png++-t és a komplex osztályt

Ezek után az előző feladathoz hasonlóan itt is megadjuk a végleges értékeket az N, M valamint X és Y maximum és minimum értékeinek.

Legnagyobb részben ennek a feladatnak a megoldása megegyezik az előző feladat megoldásával, ezért azt nem írnám le újra, inkább arra koncentrálnék, hogy miben más ez a forrás mint az előző.

Az érdemi különbség a két forrás között az az, hogy itt az `std::complex` osztályt használva, már nem kell létrehoznunk egy saját struktúrát a komplex számoknak.

E helyett szimplán létrehozzuk a double típusú komplex számokat a következőképpen:

```
std::complex<double> C, Z, Zuj;
```

Illetve a for cikluson belül sem a struktúrán belüli elemek imaginárius és valós részére hivatkozunk, hanem a `real()` és `imag()` nevű függvényeket meghívva mondjuk meg a komplex szám részeinek értékét.

```
real(C) = MINX + j * dx;  
imag(C) = MAXY - i * dy;
```

A programot fordítani és futtatni ugyan úgy kell, mint az előző feladatot.

5.3. Biomorfok

Megoldás videó: <https://youtu.be/IJMbgRzY76E>

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

A megoldás forrása [Bátfai Norbert](#) tulajdona.

A két előző feladathoz hasonlóan itt is szükségünk van a `png++` ra, ezért ha még nem szedted le akkor pillants rá a az első feladat magyarázatára, ahol részletesen le vannak írva az ehez szükséges parancsok.

Ez egy olyan mandelbrot program, ahol maga a user adja meg a határokat. Előnye hogy az eredetihez képest teljesen más képeket kapunk, hátránya viszont hogy ha a user nem tudja, hogy mit csinál akkor az egész kép egy nagy fekete semmi lesz.

Először is

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

includeoljuk az `iostream`et a `png++`t és a `komplex` osztályt.

A main argumentumai a bemeneti adatok, amikből előállítjuk magát a képet.

Ellenőrzi a program, hogy megfelelő mennyiségű bemeneti értéket adott e meg a felhasználó, és ha nem, akkor felvilágosítja, hogy hogyan kell használni a programot.

```
if ( argc == 12 )  
{  
    szelesseg = atoi ( argv[2] );  
    magassag =  atoi ( argv[3] );  
    iteraciosHatar =  atoi ( argv[4] );  
    xmin = atof ( argv[5] );  
    xmax = atof ( argv[6] );  
    ymin = atof ( argv[7] );  
    ymax = atof ( argv[8] );  
    reC = atof ( argv[9] );  
    imC = atof ( argv[10] );  
    R = atof ( argv[11] );  
}
```

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

Ha viszont megfelelő mennyiségű argumentumot adott meg a felhasználó, akkor létrehozza a képet aminek a szélessége és a magassága a felhasználó által megadott szélesség és magasság lesz.

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

Ezek után a program két egymásba ágyazott for ciklus segítségével kiszámolja, és létrehozza a képet, és el menti a felhasználó által megadott néven.

A fordítása az előző két programhoz hasonlóan működik, a futtatása azonban már így néz ki:

./3.1.3 fajlnev szelesseg magassag n a b c d reC imC R Erre egy példa:

./3.1.3 biomorf.png 800 800 10 -2 2 -2 2 .285 0 10

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

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

A megoldás forrása Bátvai Norbert tulajdona.

Először is közérdekű közlemény, hogy ennek a programnak a sikeres fordításához szükségünk lesz egy CUDA magokat használó NVIDIA kártyára, illetve az nvidia-cuda-toolkit re amit a következő paranccsal tudunk feltenni:

sudo apt install nvidia-cuda-toolkit

Ez a program ugyanúgy a mandelbrot halmazt rajzolja ki, mint az előzőek, azonban itt egy nagyon fontos különbség az, hogy míg az előző feladatoknál a képet a CPU számolta ki és készítette el, addig itt, az NVIDIA kártyák CUDA magjait használjuk a kép kiszámításához.

Ez azért fontos, mert az előző feladatoknál egyetlen egy mag dolgozott és számolt ki mindent, addig itt az én GTX 1050TI videokártyám esetében 768 darab cuda mag számolja és rajzolja ki a képet.

Ez nyilvánvalóan egy sokszor gyorsabb futási időt eredményez. Az én esetemben például amikor CPU-val futtattam a programot akkor a következő eredmények jöttek ki:

```
andras@andrasubuntu:~/cuda_mandel$ ./mandelp t.png  
2573  
25.7395 sec  
t.png mentve
```


Ez azt jelenti, hogy egy AMD FX8350 processzornál majdnem 26 másodpercbe került, hogy lefusson a program, és elkészüljön a kép.

Azonban ha már a fentebb említett GTX 1050TI kártyát használva futtatom a programot, akkor már egy kicsit hamarabb lefut a program.

```
andras@andrasubuntu:~/cuda mandel$ ./mandelcuda c.png  
c.png mentve  
4  
0.047982 sec
```

Ezek alapján így már mindössze 0.05 másodpercbe került futtatni a programot ami egy jelentős csökkenés. Pontosabban körülbelül 514-szer gyorsabban futott le ezzel a módszerrel a programunk.

A programot a gcc helyett az nvcc nevű paranccsal kell fordítani. Futtatni pedig a szokásos módon.

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!

A feladat nem lett elkezdve időben ezért passzolásra került.

Megoldás videó:

Megoldás forrása:

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás Forrása: <https://github.com/raczandras/progbook/tree/master/src/Mandelbrot/MandelJava>

A megoldás forrása [Bátfa Norbert](#) tulajdona.

A feladat az ezt megelőző (passzolt) feladat átírása Java nyelvre. A GUI megírásához szükség van egy keretrendszerre, ami jelen esetben az Abstract Window Toolkit lesz.

Először nézzük a *Mandelbrothalmaz.java* fájlt.

A main-ben a MandelbrotHalmaz() meghívásával létrehozunk egy új halmazt a megadott paraméterekkel. Ezek a paraméterek a tartományok koordinátái, a halmazt tartalmazó tömb szélessége, és a számítás pontossága.

Utána a felhasználó tevékenységeit figyeli a program, és megfelelően reagál rájuk, valamint a GUI ablak tulajdonságait adja meg, illetve kirajzolja magának a halmaznak a képét.

A következő fájlnak a *MandelbrotHalmazNagyító.java*

A nevéből adódóan ez végzi a halmazon a nagyítás folyamatát, illetve magának a halmaznak a kirajzolását is. Maga a MandelbrotHalmazNagyító osztály figyeli a felhasználó egér tevékenységeit, azzal kapcsolatban, hogy hol szeretné nagyítani a képet, illetve kirajzolja az új, nagyított képet. Ezen kívül ez végzi az elmentendő képek készítését, és elmentését is.

Végül pedig a *MandelbrotIterációk.java* fájl szerepe.

Ez a programrészlet a nagyított mandelbrot halmazok pontjait tartja nyilván. Ez egy számításra létrehozott osztály, ami a kiválasztott ponthoz tartó utat mutatja meg.

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:

C++ forrás: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/polargen/>

java forrás: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/kezd0/elsojava/PolarGen.java#110>

Ehhez a programhoz java-ban szükségünk lesz az `util.random`, az `io.*` illetve a `lang.math` java könyvtárakra. Először is a `bExist` változót hamisra állítjuk a konstruktoron belül, majd pedig inicializálunk egy randomot, ennyi a konstruktor.

Ezek után a `PolarGet` függvény ami az érdemi munkát végzi. Először is ellenőrzi, hogy volt e már generálás. Ha volt akkor azt adja vissza, de ha nem, akkor a matekos algoritmus segítségével legenerálja a két random normált és `bExists`-et átállítja az ellentétére.

Érdekes, hogy a JDK-n belül is ez a megoldás van alkalmazva, ami annyit jelent, hogy azok akik a random java könyvtárat megírták, azok ugyan úgy gondolkoztak mint egy egyetemi hallgató.

6.2. LZW

Valósítsd meg C++-ban az LZW algoritmus fa-építését!

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Welch/lzw.cpp>

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az **UDPROG** repóban.

Mindkét esetben a bináris fa felépítésének a lépései a következők:

Ha 1-est szeretnénk betenni a fába, akkor először megnézzük, hogy az aktuális csomópontnak van e már ilyen eleme. Ha még nincs, akkor egyszerűen betesszük neki az 1-es gyermekének az 1-et. Azonban ha már van ilyen gyermeke, akkor létre kell hozni egy új csomópontot és az ő gyermekének adjuk át az 1-et.

Ez hasonlóan működik akkor is ha nullást szeretnénk betenni, annyi különbséggel, hogy nem az 1-eseket vizsgáljuk, hanem a nullásokat. Ezt a lépést a programban a következő részlet oldja meg:

```
void operator<<(char b) {
    if (b == '0') {
        if (!fa->nullasGyermek()) {
            Csomopont *uj = new Csomopont('0');
            fa->ujNullasGyermek(uj);
            fa = &gyoker;
        } else {
            fa = fa->nullasGyermek();
        }
    }
    else {
        if (!fa->egyenesGyermek()) {
            Csomopont *uj = new Csomopont('1');
            fa->ujEgyenesGyermek(uj);
            fa = &gyoker;
        } else {
            fa = fa->egyenesGyermek();
        }
    }
}
```

A megadott fájl tartalma alapján felépíti az LZWBinfá csomópontjait. Jelen esetben ezt a Bináris Fát in order bejárással dolgozzuk fel, ami annyit jelent, hogy először a fa bal oldalát dolgozzuk fel, majd a fának a gyökerét, és legvégül pedig a jobb oldalt. A következő feladatban ezen viszont már változtatunk.

Fordítása a szokásos módon történik a futtatása pedig a következőképpen:

./lzw bemenet -o kimenet

6.3. Fabejárás

Tutorálóm: [George Butcovan](#)

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

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Welch/fabe.cpp>

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az [UDPROG](#) repóban.

Az előző feladatban tárgyalt fát In Order módszerrel járta be a program. Ez azt jelenti, hogy először a részfa bal oldalát dolgozzuk fel, majd a részfa gyökerét, és legvégül pedig a részfa jobb oldalát.

Erre ugyanúgy megmaradt a lehetőségünk, csupán a következőképp kell futtatni a programot:

./lzw bemenet -o kimenet i

Ezzel szemben itt két másik fajta bejárési módszerrel dolgozzuk fel a fát. Az egyik a Pre Order bejárési mód, a másik pedig a Post Order.

A Pre Order bejárési módnál először a részfa gyökerét dolgozzuk fel, másodjára a részfa bal oldalát, és utoljára pedig a részfa jobb oldalát. A pre order bejárési mód használatához a következőképpen kell futtatni a programot:

./lzw bemenet -o kimenet r

A Post Order bejárési módnál pedig legelőször a részfa bal oldalát dolgozza fel a program, majd a jobb oldalát, és legvégül pedig a részfa gyökerét. A Post Order bejáráshoz a következő parancs használatával kell futtatni a programot:

./lzw bemenet -o kimenet r

6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Welch/tag.cpp>

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az **UDPROG** repóban.

Ez a program az eredeti Bevezetés a Programozásba tárgyon már tanult *z3a7.cpp* nevű program szerint működik, hiszem itt a csomópont már kompozícióban van a fával. Az egész az LZWBInfa osztállyal kezdődik, aminek van privát, és publikus része is. A publikus részen belül található a konstruktor, és a destruktor deklarációja. Itt kerülnek vizsgálatra a bemenő elemek, és jönnek létre a 0-s illetve 1-es elemek is. Túlterhelődik az operátor, és megvizsgálja a program, hogy létezik e már nullás gyermek. Ha nincs, akkor létrejön. Egyes gyermeknél ugyan ez a helyzet.

A kiír függvény pedig kiírja a csomópontokat.

Majd jön az LZWBInfa privát része. Itt található meg a Csomópont osztály amin belül a konstruktor megkapja a gyökeret. Még a Csomópont osztályon belül találhatóak azok a függvények, amivel le tudjuk kérdezni, hogy ki az aktuális csomópont nullás illetve egyes gyermeke, valamint az `ujNullasGyermek()` illetve `ujEgyesGyermek()` függvények, amik létrehozzák az új nullás és egyes gyermekeket

6.5. Mutató a gyökér

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

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Welch/gyoker.cpp>

Ehhez a feladathoz az **UDPROG** repóban megtalálható BinFa programot vettem alapul.

Ebben a megoldásban az előző feladathoz képest kicsit másképp a megoldás. A következő dolgokat kell átírni a már meglévő programban:

Először is a 315. sorban a csomópont után tegyünk egy `*`-ot ezzel mutatóvá téve a gyökeret. Ha így megpróbáljuk lefordítani a programot akkor nagyon sok szintaktikai hibát fogunk kapni a fordítótól válaszként.

Nem kell pánikolni. Az a dolgunk, hogy ezeket a hibákat egyesével kijavítsuk. Az első két hiba kijavításához a következő részletet kell átírni.

A 92. és a 93. sorban a

```
szabadit (gyoker.egyenesGyermek ());  
szabadit (gyoker.nullasGyermek ());
```

utasítások helyett a

```
szabadit (gyoker->egyenesGyermek ());  
szabadit (gyoker->nullasGyermek ());
```

utasításokat kell használni.

Ez után már kettővel kevesebb hibát kapunk. Az összes többi hibát a referenciák okozzák. Ahoz hogy ezeket a hibákat megoldjuk a következő sorokban kell tevékenykednünk: 92, 132, 147, 170, 210, 336, 344, és 356. Azonban a hibát minden sorban ugyan azzal a módszerrel kell javítani, ami nem más mint hogy a

```
&gyoker
```

helyett azt kell írni hogy

```
gyoker
```

Vagyis kiszedjük a referenciákat, mivel alaphoz a memóriacímek lesznek átadva.

Ezek után a programunk ugyan lefordul, de amikor megpróbáljuk futtatni, akkor szegmentálási hibát kapunk. Ennek a javításához a konstruktort kell átírni a következőképpen:

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

Ezzel foglalunk helyet a memóriában a gyökérnek. Viszont amit lefoglalunk, azt fel is kell szabadítani, éppen ezért a destruktort is módosítani kell a következőképpen:

```
~LZWBInFa ()  
{  
    szabadit (gyoker->egyenesGyermek ());  
    szabadit (gyoker->nullasGyermek ());  
    delete gyoker;  
}
```

Mostmár fel is szabadul, amit lefoglaltunk.

6.6. Mozgató szemantika

Tutorálóm: [Molnár Antal](#)

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

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Welch/mozgato.cpp>

A megoldás forrásának az alapja megtalálható az **UDPROG** repóban. Én ezt módosítottam.

Maga az LZWBinFa osztály felépítése úgy néz ki, hogy az osztályon belül léteznek a beágyazott csomópont objektumok amik a fát alkotják. Ezek alapján a fa másolása nem más, mint ezeknek a csomópontoknak a másolása. Ehhez létre kell hoznunk a mozgató illetve mozgató értékadás konstruktorokat.

```
LZWBinFa (LZWBinFa&& masik){
    gyoker=nullptr;
    *this= std::move(masik);

}

LZWBinFa& operator= (LZWBinFa&& masik){
    std::swap(gyoker,masik.gyoker);
    return *this;

}
```

Először a mozgató értékadásról (alsó) szólnék pár szót, ami csupán annyit jelent, hogy ha egyenlőségjel operátort használunk, akkor az `std::swap()` függvénnyel megcserélődik a két gyökér mutatója.

Másodszor pedig a mozgató konstruktor. Itt először is `nullptr` (nullpointer) értéket adunk abban a binfában lévő gyökérnek, amelyik fába akarjuk mozgatni a ("masik") fát. Majd a "masik" nevű fát átmozgatjuk az `std::move()` függvénnyel, ami annyit jelent, hogy a gyökér mutató mostmár a paraméterként kapott "masik" fára mutat, ami azért történhetett meg, mert az `std::move()` függvény tulajdonképpen nem is mozgat semmit, hanem a paraméterül kapott értéket jobbérték referenciává alakítja.

7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk

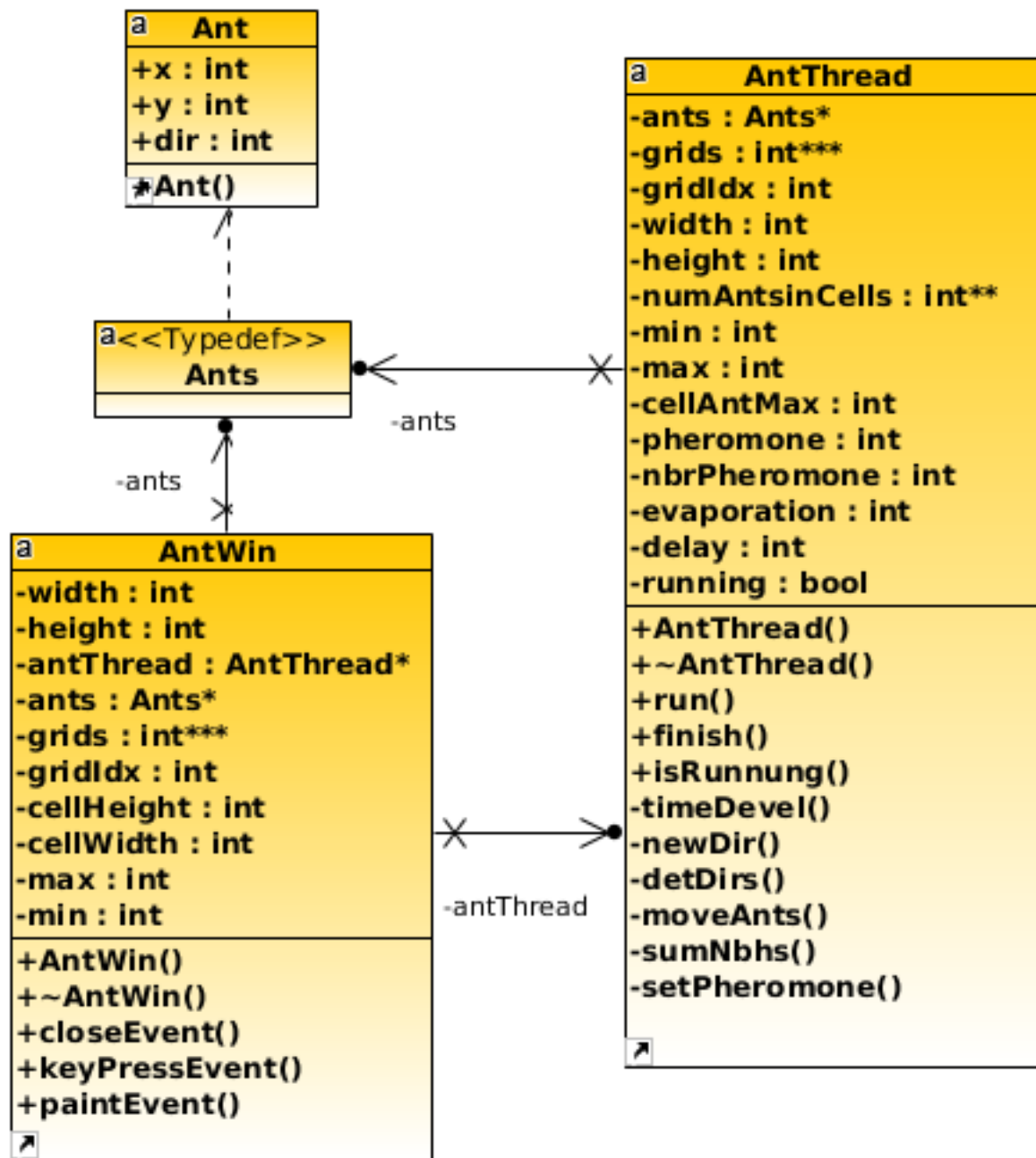
Írj Qt C++-ban egy hangyaszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

Tutoráltam: [Molnár Antal](#)

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

Megoldás forrása: <https://github.com/raczandras/progbook/tree/master/src/Conway/Ant>

Az osztálydiagram:



A megoldás forrása, illetve az UML osztálydiagram [Bátfai Norbert](#) tulajdona.

Ebben a feladatban hangyákat kell szimulálni. Maga a megoldás azt a biológiai tényt alkalmazza, hogy a hangyák a való életben szagokkal, úgynevezett feromonokkal kommunikálnak egymással. Ha például egy hangya valamilyen érdekes dolgot talált, akkor ott hagyja a nyomát, illetve megjelöli az útvonalat. Az éppen arra járó többi hangya ezt megérzi, és a legfrissebb feromon nyomát követve ők is el fognak jutni a célba. Ezeket észben tartva készítették el ezt a hangyaszimulációt.

Az osztálydiagrammon belül négy egységet találhatunk, ezek a következők: **Ant**; **Ants**; **AntWin**; és **AntThread**

Ezek a programunk osztályai, ezeken az egységeken belül vannak megadva az adott osztály változóit és függvényeit. Ilyen például az **AntWin** egységen belül található *width* és *height* változók, amik a képernyő

hosszúságát, és szélességét adják meg. Vagy például a `closeEvent()` és a `keyPressEvent()` függvények, amik szintén az `AntWin` osztály részei. Ezek alapján meghatározhatjuk, hogy az `AntWin` osztály a szimuláción belül a világot kezeli.

Az `AntThread` osztály kezeli a hangyákat, illetve azok mozgását, illetve a virtuális feromonok terjedéséről is ez az osztály gondoskodik.

7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/blob/master/src/Conway/sejt.java>

A Megoldás forrása [Bátfai Norbert](#) tulajdona.

Arról, hogy mi az az életjáték, illetve, hogy mik a szabályai, a következő feladat leírásában részletesebben írok. Most legyen elég ennyi:

Ha egy négyzetnek pontosan három darab élő szomszédja van, akkor abban a négyzetben egy új sejt jön létre.

Ha egy már élő sejtnek pontosan kettő vagy három darab szomszédja van, akkor az a sejt továbbra is életben marad.

Ha viszont egy már meglévő élő sejtnek háromnál több élő szomszédja van (túlnépesedés), vagy kettőnél kevesebb, akkor az a sejt meghal. Ezt szimulálja az életjáték.

Ezek a szabályok az `időFejlődés()` függvényben vannak lefektetve.

```
if(rácsElőtte[i][j] == ÉLŐ) {
    /* Élő élő marad, ha kettő vagy három élő
       szomszédja van, különben halott lesz. */
    if(élők==2 || élők==3)
        rácsUtána[i][j] = ÉLŐ;
    else
        rácsUtána[i][j] = HALOTT;
} else {
    /* Halott halott marad, ha három élő
       szomszédja van, különben élő lesz. */
    if(élők==3)
        rácsUtána[i][j] = ÉLŐ;
    else
        rácsUtána[i][j] = HALOTT;
```

Igaz ugyan, hogy az életjáték egy úgynevezett nullszemélyes játék, de ebben a példában a játékos mégis tudja irányítani kicsit a dolgokat. Ugyanis a program figyeli a billentyűzet bizonyos gombjait (`k`, `n`, `l`, `g`, `s`), illetve az egér mozgását, és kattintásait is. Ezt három függvénnyel teszi. Az `addKeyListener(new java.awt.event.KeyAdapter())` függvénnyel figyeli a billentyűzetet. Ezen a függvényen belül egy `if-else` szerkezet állapítja meg, hogy éppen melyik gombot nyomta le a felhasználó a

```
if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K){}
```

feltétel a K betű lenyomását azonosítja, és csökkenti a sejtek méretét.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N){}
```

feltétel az N betű lenyomását azonosítja, és növeli a sejtek méretét.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S){}
```

feltétel az S betű lenyomását azonosítja, és készít egy képet a sejtterről.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G)
```

feltétel a G betű lenyomását azonosítja, gyorsítja a szimuláció sebességét.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L
```

feltétel az L betű lenyomását azonosítja, és lassítja a szimuláció sebességét.

Az egér mozgását, illetve kattintásait pedig a `addMouseListener(new java.awt.event.MouseAdapter()` illetve a `addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {}` függvények figyelik. Az egér kattintásaival egy sejt állapotát tudjuk megváltoztatni. Az egér mozgásával pedig az összes érintett sejt élő állapotba kerül.

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/tree/master/src/Conway/Qt>

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Az életjátékot John Conway találta ki, és nem teljesen hiteles rá a játék kifejezés, mert ez egy úgynevezett nullszemélyes játék. Magának a játékosnak annyi a dolga, hogy megadja a kezdőalakzatot, majd pedig megfigyelheti, hogy mi lesz az eredmény.

Alapja egy négyzetrácsos tér, amikben élhetnek sejtek, de minden egyes négyzetben csak egy darab sejt élhet. Magának a "játéknak" a szabályai a következők:

Ha egy sejtnak kettő vagy három élő szomszédos sejtje van, akkor a sejt meg fogja élni a következő generációt. Az összes többi esetben viszont kihal a sejt, akár azért mert túl sok, akár azért mert túl kevés szomszédja van.

Ahol azonban egy üres négyzetrácsnak pontosan három élő sejt a szomszédja, akkor ott új sejt jön létre.

Ez mellesleg két részre osztotta az embereket. Voltak akik minen napi rutinjukká tették azt, hogy az életjátékkal "játszanak", egyfajta függők lettek, és voltak azok, akik nem értették hogy mi a jó benne.

Maga a program ugyanúgy működik, mint a java verzió. Mind a két program két darab mátrixsal dolgozik, viszont itt a teljes kód megírása helyett a Q-t is segítségül hívjuk.

A programot a következőképpen tudjuk fordítani és futtatni: **qmake Sejtauto.pro make ./Sejtauto**

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása: <https://github.com/raczandras/progbook/tree/master/src/Conway/BrainB>

A megoldás forrása [Bátfai Norbert](#) tulajdona.

A programhoz szükségünk lesz az OpenCV-re, aminek a feltelepitéséhez a lépéseket [ezen a linken](#) elérhető weblapon találjuk.

Ez egy miniatűr játék, ami a felhasználó szem-kéz koordinációjáról, illetve a megfigyelőképességéről gyűjt össze információkat.

Amikor elindítjuk a játékot akkor egy ablak fogad minket, és a lényege az, hogy a **Samu entropy** nevű négyzetben lévő fekete pöttyön belül tartjuk az egerünk mutatóját.

A játék a teljesítményünk alapján lesz könnyebb, vagy nehezebb. Minél jobban játszunk, annál több Entropy lesz a képernyőn, ezáltal megnehezítve a Samu entropy követését. Viszont ha már nem tudjuk nyomon követni a Samu entropy-t akkor folyamatosan eltüntet a hozzáadott entropy-kat, ezáltal megkönnyítve a játékot.

Én közel két perc játék után a következő eredményeket produkáltam:

```
NEMESPOR BrainB Test 6.0.3
time      : 1164
bps       : 52830
noc       : 17
nop       : 0
lost      :
4700 9800
mean      : 7250
var       : 3606.24
found     : 11740 19150 12360 36950 49930 33470 38680 20860 4790 2230 16870 ↔
          6500 25000 50340
mean      : 23490
var       : 16066.3
lost2found: 2230
mean      : 2230
var       : 0
found2lost:
mean      : 0
var       : 0
time      : 1:56
U R about 0.136108 Kilobytes
```

A programot futtatni az előző feladathoz hasonlóan szintén a **qmake** és **make** parancsokkal lehet fordítani.

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

aa Python

Megoldás videó:

Megoldás forrása: https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol

A megoldás forrása [Bátfai Norbert](#) tulajdona.

Ennél a feladatnál TensorFlow-t fogunk használni, ami egy nyílt forráskódú szoftver, amit főleg Machine Learning-nél (Gépi tanulás) használnak. A nagy cégek, pl Google is ezt használják. Jelentősége, hogy egyszerre több CPU-n és GPU-n is képes futni. Azonban ahhoz, hogy ezt használhassuk, fel is kell telepítenünk.

Ez a program a TensorFlow Hello World-je. Két számot szoroz össze neurális hálókat használva.

```
#!/usr/bin/env python2
# TensorFlow Hello World 1!
# twicetwo.py
#
import tensorflow as tf

node1 = tf.constant(2)
node2 = tf.constant(2)

node_twicetwo = tf.math.multiply(node1, node2, name="twicetwo")

sess = tf.Session()
print sess.run(node_twicetwo)

writer = tf.summary.FileWriter("/tmp/twicetwo", sess.graph)
# nbatfai@robopsy:~/Robopsychology/repos/tf/tf/tensorboard$ python ↵
# tensorboard.py --logdir=/tmp/twicetwo

tf.train.write_graph(sess.graph_def, "models/", "twicetwo.pb", as_text= ↵
False)
```

A program importálja a tensorflow-t `tf` néven. Majd a `node1`-nek illetve a `node2`-nek értékül adja a 2 értéket a

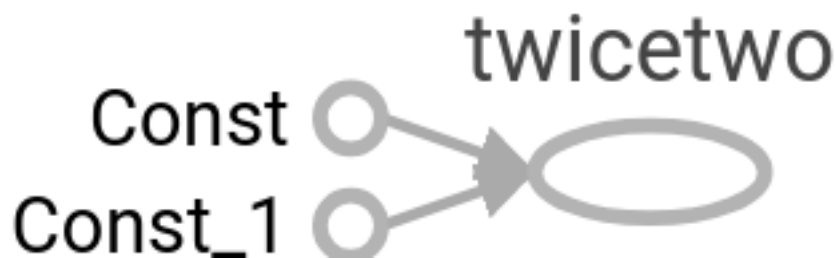
`tf.constant(2)` függvény segítségével.

Ezek után a `tf.math.multiply()` függvénnyel kiszámolja a két szám szorzatát, és azt értékül adja a `node_twicetwo`-nak.

Végül kiírja a szorzatot a `sess = tf.Session()` értékadással és függvénnyel, illetve a `print sess.run(node_twicetwo)` paranccsal.

Es a számítási gráfot a `writer = tf.summary.FileWriter("/tmp/twicetwo", sess.graph)` értékadással

illetve a `tf.train.write_graph(sess.graph_def, "models/", "twicetwo.pb", as_text=False)` függvénnyel.



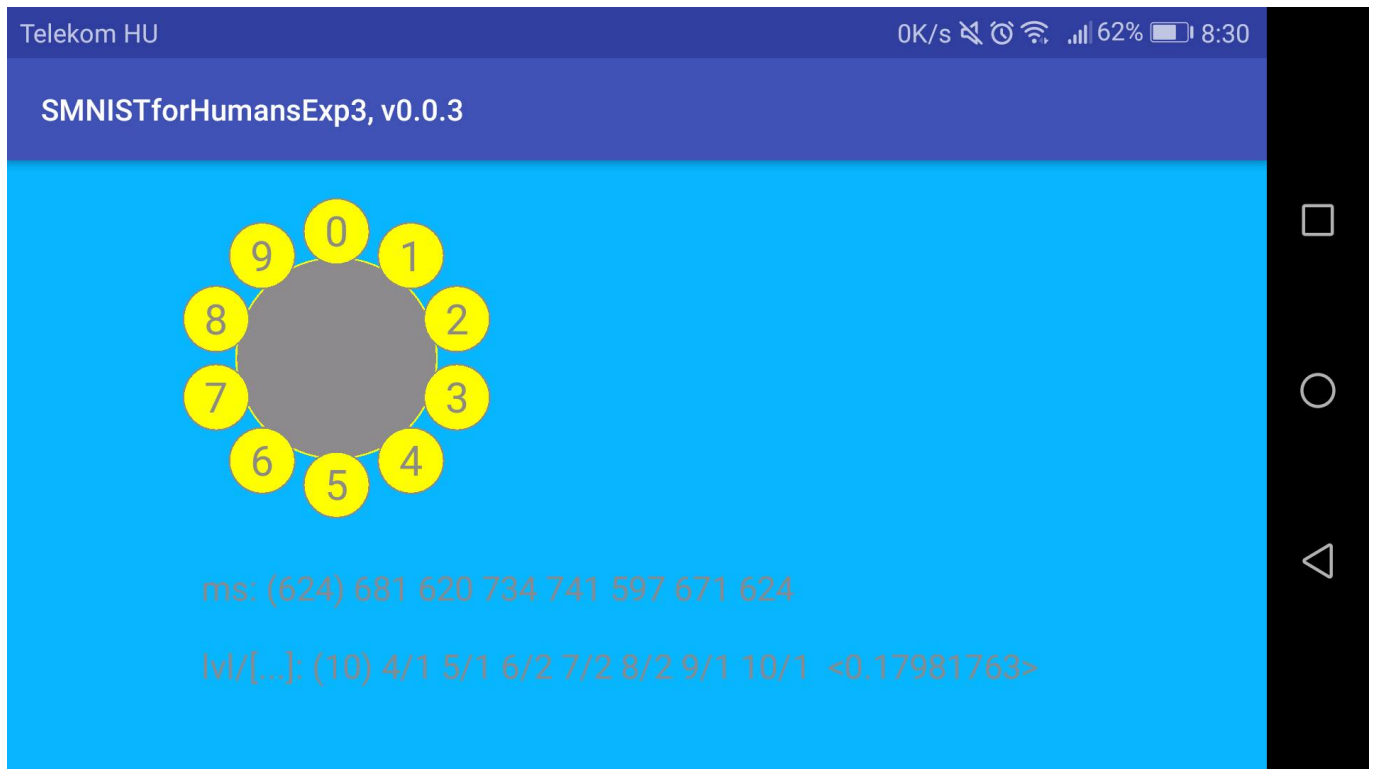
8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

A feladatot az SMNIST-ben elért eredményem alapján passzoltam.



8.3. Minecraft-MALMÖ

Megoldás videó:

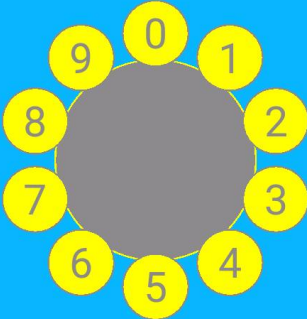
Megoldás forrása:

A feladatot az SMNIST-ben elért eredményem alapján passzoltam.

Telekom HU

0K/s 62% 8:30

SMNISTforHumansExp3, v0.0.3



ms: (624) 681 620 734 741 597 671 624

lv/[.]: (10) 4/1 5/1 6/2 7/2 8/2 9/1 10/1 <0.17981763>



9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó:

Iteratív megoldás: <https://github.com/raczandras/progbook/blob/master/src/Chaitin/iter.lisp>

Rekurzív megoldás: <https://github.com/raczandras/progbook/blob/master/src/Chaitin/rek.lisp>

A forrást a [codeforsharing](#) inspirálta.

Ebben a feladatban a program faktoriális számol iteratív illetve rekurzív módon. A lisp a második magas szintű programozási nyelv. Egyedül a fortran előzte meg. Először nézzük az iteratív módszert.

```
(defun fact(n)
  (setf f 1)
  (do ((i n (- i 1))) ((= i 1))
    (setf f (* f i)))
  )
fact(4))
```

A program első sorában definiáljuk magát a függvényt `fact` néven. Majd egy `f` nevű változót, aminek az értéket 1-re állítjuk. Majd jön egy ciklus, ami `i`-nek átadja a számot aminek a faktoriálisát ki kell számolni. A program `i`-ből folyamatosan kivon 1-et egészen addig, amíg `i` értéke 1 nem lesz. A ciklus törzsében pedig `f` értéke `f*i` lesz. Végül a program meghívja magát a `fact()` függvényt.

Ezzel szemben a rekurzív módszert valamennyivel könnyebb olvasni.

```
(defun fact(n)
  (if (= n 0) 1
      (* n (fact(- n 1)))
  )
)
fact(4)
```

Először ebben a példában is a `fact()` függvény kerül definiálásra. Azonban ezek után itt egy `if` szerepel, ami azt ellenőrzi, hogy `n` egyenlő-e 0-val. Ha nem, akkor szimplán meghívja a függvény saját magát, azonban itt már `n-1` amivel számol, így számolva ki a faktoriális értékét.

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

A megoldás forrása Bátfai Norbert tulajdona

Ez egy program a gimp-hez, ami a megadott szöveghez króm effektet ad. Maga a gimp egy ingyenes képszerkesztő program.

Maga a forrás egy tömbbel indul, ami a króm effekt megvalósításához szükséges információkat tartalmazza.

```
(define (color-curve)
  (let* (
    (tomb (cons-array 8 'byte))
  )
    (aset tomb 0 0)
    (aset tomb 1 0)
    (aset tomb 2 50)
    (aset tomb 3 190)
    (aset tomb 4 110)
    (aset tomb 5 20)
    (aset tomb 6 200)
    (aset tomb 7 190)
  tomb)
)
```

A következő függvény a betűk méretét határozza meg. A szükséges méretek a a GIMP beépített függvényeivel határozza meg a következőképpen:

```
(set! text-width (car (gimp-text-get-extents-fontname text fontsize PIXELS ↵
  font)))
(set! text-height (elem 2 (gimp-text-get-extents-fontname text ↵
  fontsize PIXELS font)))
```

Ez annyit jelent, hogy a `gimp-text-get-extents-fontname` első értékét (ami maga a méret) állítja be a `text-width` illetve a `text-height` változóknak a `set` utasítás használatával.

Majd a `script-fu-bhax-chrome-border` függvény hozza létre a tényleges króm effektet szöveget. Ezt egy új rétegen (layer) teszi. Ennek az új rétegnek a háttere fekete, a rá kerülő szöveg pedig fehér színű lesz.

```
(gimp-image-insert-layer image layer 0 0)

(gimp-image-select-rectangle image CHANNEL-OP-ADD 0 (/ text-height 2) ↵
  width height)
(gimp-context-set-foreground '(255 255 255))
(gimp-drawable-edit-fill layer FILL-FOREGROUND )
```

Végül a program regisztrálásra kerül magába a gimp-be, hogy el tudjuk érni

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

A megoldás forrása Bátfai Norbert tulajdona.

Az előző feladathoz hasonlóan itt is egy gimp kiegészítőről van szó. Itt azonban a bemenő szövegből egy név-mandala fog készülni. A mandala egy szimmetrikus kör alakú kép, ami a Hindu vallásban nagy szerepet játszik a Hindu istenek ábrázolásában.

Először a program meghatározza a szöveg hosszát, a `gimp-text-get-extents-fontname` függvény használatával. A kapott értéket a `set!` utasítással a `text-width` változó értékének adja.

```
(set! text-width (car (gimp-text-get-extents-fontname text fontsize PIXELS ↵
font)))
```

Ebben a feladatban, ugyanúgy határozzuk meg a szöveg méretét, mint az előzőben:

```
(set! text-width (car (gimp-text-get-extents-fontname text fontsize PIXELS ↵
font)))
(set! text-height (elem 2 (gimp-text-get-extents-fontname text ↵
fontsize PIXELS font)))
```

A GIMP beépített `gimp-text-get-extents-fontname` függvényét, és a `set!` utasítást felhasználva a `text-width` és a `text-height` változók értékei lesznek a szükséges méretek.

Ezek után jön maga a mandala. Először létrejön egy réteg (layer)

```
(gimp-image-insert-layer image layer 0 0)

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

Amit feltöltünk a felhasználó által megadott adatokkal. Ezek a a szöveg, a szöveg betűtípusa. Ezután a réteget tükrözi a program, ezzel elérve a szimmetriát, majd a program elforgatja a képet, és megismétli a tükrözést. Majd a réteget felnagyítja a kép teljes méretére:

```
(gimp-layer-resize-to-image-size textfs)
```

Ezután két körnek álcázott ellipszist illeszt a program a képre ezzel létrehozva a mandalát. Az egyik kör vastagsága 22, a másiké pedig 8 lesz.

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

```
(gimp-context-set-brush-size 22)
(gimp-context-set-brush-size 8)
```

Végül pedig megjeleníti a képet:

```
(gimp-display-new image)
```

Ezek után ismét már csak a GIMP-be regisztrálás van hátra.

10. fejezet

Helló, Gutenberg!

10.1. Juhász István - Magas szintű programozási nyelvek 1 olvasónaplója

Ez a Rácz András nevű Debreceni Egyetemi hallgató által készített olvasónapló a Juhász István által írt Magas szintű Programozási Nyelvek 1 (pici-könyv) című könyvből.

A számítógépet programozó nyelveknek három szintje van. Ezek a gépi nyelv, az assembly nyelv és a magas szintű nyelv. Mi a magas szintű programozási nyelvekkel foglalkozunk, ami az emberek által a legjobban érthető. Az ilyen nyelven megírt programot nevezzük forrásprogramnak. Azonban a processzorok csak az adott gépi nyelven írt programokat tudják végrehajtani. Ezért forrásprogramot át kell írni gépi kódra. Ezt a munkát végzik el a fordítók.

Minden programnyelvnek van saját szabványa, ez a hivatkozási nyelv. Pontosan meg vannak adva a nyelvtani szabályok, amiket be kell tartani, különben vagy szintaktikai, vagy szemantikai hibát fogunk kapni. A szintaktikai hiba az, amit a fordító észrevesz, és jelez nekünk, hogy gond van. Míg a szemantikai hiba esetén a fordító nem kapja el a hibát, de a program nem megfelelően fog működni.

Ezen kívül minden nyelvnek vannak Adattípusai is. Ezek lehetnek beépített, vagy a programozó által létrehozottak is. ilyen típusok például az egész számok, a karakterek, a karakterláncok, a tömbök, a listák, a mutatók.

Léteznek nevesített konstansok is. Ezek is lehetnek beépítettek, vagy létrehozottak is. Ilyen konstans például a π . Létrehozni pedig `c++` nyelvben a `#define` al míg `java`-ban a `final` utasítással tudunk.

A legalapvetőbb dolgok azonban a változók. Ezekben tároljuk a számunkra szükséges dolgokat. Egy változónak van típusa és értéke. A típusa lehet szám, karakter, karakterlánc, illetve logikai. Az értéke pedig a lehetséges típusok alapján lehet szám, karakter, karakterek sorozata, illetve igaz/hamis.

A programozási nyelvekben használunk még Kifejezéseket is. Ezek egyfajta műveletek. Egy kifejezésnek három része van. A művelet bal szélén valamilyen változó áll, aminek szeretnénk egy értéket adni, középen egy műveleti jel, és jobb oldalt pedig vagy egy másik változó, vagy egy konkrét érték, ami a bal oldalt álló változónak az új értéke lesz.

A gépi kódot a fordító az utasítások alapján generálja. Ezek az utasítások a következők lehetnek : Értékadó utasítás; Üres utasítás; Ugró utasítás; Elágaztató utasítások; Ciklusszervező utasítások; Hívó utasítás; Vezérlésátadó utasítások; I/O utasítások; Egyéb utasítások.

A ciklusokat is nagyon sokszor használják a programozók. Ezek segítségével a megadott parancsokat egymás után többször is elvégzi a program. A ciklusokhoz tartoznak a Vezérlő utasítások, amik a következők: A Continue parancs esetén a ciklus jelenlegi lépésében a hátra lévő utasításokat nem hajtja végre, hanem a következő cikluslépésre ugrik. A Break parancs esetén a ciklus megáll, és nem fut tovább. A return parancs esetén leáll a ciklus és visszaadja az eredményt.

Az alprogramok, vagy másnéven függvények olyan programrészletek, amiket megírva később meg lehet hívni őket, és a megadott értékekből előállítanak egy eredményt. Az alprogramoknak van neve és argumentumai. A névvel hívjuk meg őket, az argumentumok pedig azok az értékek amikből a végeredmény áll elő.

A programokban a blokkok olyan programrészletek amik programrészletekben helyezkednek el. Ilyen például az if elágazás után a potenciálisan végrehajtandó utasítások.

10.2. Kernighan és Richie olvasónaplója

Ez a Rácz András nevű Debreceni Egyetemi hallgató által készített olvasónapló a BRIAN W. KERNIGHAN – DENNIS M. RITCHIE által írt A C programozási nyelv című könyvből.

Először is a vezérlési Szerkezetek a ciklusok, az elágazások

Az elágazásokba beletartozik az if, if-else, else-if, else, és a switch feltételvizsgálatok. Ezekkel értékeket tudunk vizsgálni, és ezek végrehajtani a megfelelő utasítást, utasításokat végrehajtani. Az if, if-else, else-if, else kifejezéseknél az if után vizsgáljuk meg az értéket, majd jönnek az utasítások, és végül opcionálisan else-if vagy else. Bármennyi else-if lehet egymás után, azonban else csak egy vagy nulla. Viszont célszerű else-t is használni, mert általában kevés esély van arra, hogy minden esetet lefedünk szimplán else-if használatával.

Ezzel szemben a switch esetében megadjuk az értéket, majd tetszőleges darabszámú case használatával megnézzük, hogy az e az érték, ami nekünk kell, és ha igen akkor az aktuális case utasításait hajtja végre. Célszerű megjegyezni, switch-case használatánál a program minden esetben végigellenőrzi az összes case-t, ezért ha nem szeretnénk, hogy az összeset ellenőrizze, ha már talált egy egyezést, akkor használjuk a break utasítást.

A Ciklusok esetében beszélhetünk for, while, és do while ciklusokról.

A for esetében a programozó adja meg, hogy hányszor fusson le a ciklus. A while és a do while esetében pedig addig fut a ciklus, amíg egy feltétel nem teljesül. Éppen ezért vigyázni kell, nehogy véletlenül egy végtelen ciklus alakuljon ki. Fontos különbség még a while és a do while ciklusok között, hogy míg a while ciklus először ellenőrzi, hogy teljesült-e a feltétel, majd pedig lefuttatja az utasításokat, addig a do while ciklus először lefuttatja az utasításokat, majd pedig ellenőrzi, hogy teljesült-e már a feltétel.

A C nyelv alapvető adattípusai az int, a float, a double, a char, és a bool.

Az intek (integerek) egész számok amik lehetnek pozitívak és negatívak is. Az int mérete 4bájtt, azaz 32bit

A float és a double típusú változókban valós, úgynevezett lebegőpontos számokat lehet tárolni. Ilyen például a 0.5. A különbség a két változó között azonban az, hogy, hogy míg a float mérete csak 4bájtt, addig a double mérete 8bájtt.

A char (character) típusú változóban meglepő módon egy karaktert lehet eltárolni. A char mérete 1bájtt.

Az alapvető adattípusokon túl a C nyelvnek vannak Állandói is. Ilyen például a `#define`, amivel meg tudunk adni meg nem változtatható értékeket. Ezekre később hivatkozni tudunk. De ilyen állandók még az escape sorozatok, amiket az adatok kiíratásánál tudunk alkalmazni. Ilyen például a `\n` amivel egy új sort kezdünk.

10.3. Benedek Zoltán, Levendovszky Tihamér - Szoftverfejlesztés C++ nyelven olvasónaplója

A C++ egy objektum orientált programozási nyelv, ami egyben alacsonyabb szintű elemeket is támogat.

A C++-ban ha egy függvényt paraméterek nélkül hívunk meg, akkor az egyenértékű egy void paraméterrel. Aminek pont az a jelentése, hogy a függvénynek nincs paramétere. További különbség, hogy míg a C nyelvben egy függvényt csak a neve alapján azonosítunk, addig C++-ban egy függvényt a neve és az argumentumai határoznak meg. Ezáltal C++-ban előfordulhat két ugyan olyan nevű függvény különböző argumentumokkal. További változás, hogy C++-ba be lettek vezetve a referenciák, valamint egy új típus is bevezetésre került, ami nem más mint a bool. A bool egy logikai változó ami lehet igaz vagy hamis értékű.

A C++ bevezette az osztályokat, amik az adatok, és metódusok együttese. Innen ered az objektum orientáltság, mivel az objektum a egy darab osztály egy darab előfordulása. A metódus pedig az osztálynak egy olyan eleme, egy olyan függvény, ami az osztályba tartozó adatokat manipulálja.

A konstruktorok és destruktorok előredefiniált függvénymezők, amelyek kulcsszerepet játszanak a C++ nyelvben. Alapvető probléma a programozásban az inicializálás. Mielőtt egy adatstruktúrát elkezdénénk használni, meg kell bizonyosodnunk arról, hogy megfelelő méretű tárterületet biztosítsunk a számára, és legyen kezdeti értéke. Ezt a problémát orvosolják a konstruktorok.

A destruktorok pedig egy konstruktor által már létrehozott objektum törlésében segítenek. Törlik a tartalmát, és felszabadítják az objektum által elfoglalt helyet. Ha mi nem hozunk létre destruktort, akkor a C++ a saját alapértelmezett változatát fogja használni.

Létezik még másoló konstruktor is, ami egy már meglévő objektumból hoz létre egy újat. Lefoglal a memóriában egy részletet, és annak az értékét felülírja a már létező objektum értékeivel.

A C++ nyelven az osztályok adattagjai előtt szerepelhet a static szó. Ez azt jelenti, hogy ezeket a tagokat az osztály objektumai megosztva használják.

Gyakran kerülünk olyan helyzetbe, hogy egy adott típusnak úgy kellene viselkednie, mint egy másiknak. Ekkor kell típuskonverziót alkalmazni. Ezt meg lehet tenni implicit és explicit módon is.

Implicit konverziót általában hasonló típusokon lehet elvégezni. Ilyen például ha egy integer változó értékét szeretnénk átadni egy long típusú változónak.

```
int x = 5;
long y = x;
```

Mind a ketten egész szám típusok, viszont a long nagyobb méretű, ezért a konverzió gond nélkül megtörténik.

Ez a módszer explicit konverzió esetén nem biztos, hogy működni fog, és még adatvesztéssel is járhat. Ilyen például ha egy integer változó értékét szeretnénk átadni egy byte értékű változónak. A byte mérete kisebb mint az int, ezért a változó előtt kell lennie egy zárójelnek benne a típussal.

```
int x = 300;  
byte y = (byte)x;
```

Itt például az `y` értéke 44 lesz, mert a 300-at kilenc biten kell felírni, azonban a `byte` csak 8 bitet tárol, ezért az `x`-nek csak az első 8 bitjét fogja eltárolni.

C++-ban lehetőségünk van függvénysablonok és osztállysablonok létrehozására is, ezek a `template`-ek. A `template` argumentumai eltérnek a hagyományos argumentumoktól. Egyrészt már a fordítás közben kiértékelődnek, ezért a futás közben már konstansok. Éppen e miatt az argumentumok típusok is lehetnek, nem csak értékek.

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. Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven

Tanulságok, tapasztalatok, magyarázat...

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

Tanulságok, tapasztalatok, magyarázat...

11.3. Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba

Tanulságok, tapasztalatok, magyarázat...

12. fejezet

Helló, Arroway!

12.1. OO szemlélet

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.2. Homokozó

Írjuk át az első védési programot (LZW binfa) C++ nyelvről Java nyelvre, ugyanúgy működjön! Mutassunk rá, hogy gyakorlatilag a pointereket és referenciákat kell kiirtani és minden máris működik (erre utal a feladat neve, hogy Java-ban minden referencia, nincs választás, hogy mondjuk egy attribútum pointer, referencia vagy tagként tartalmazott legyen). Miután már áttettük Java nyelvre, tegyük be egy Java Servletbe és a böngészőből GET-es kéréssel (például a böngésző címsorából) kapja meg azt a mintát, amelynek kiszámolja az LZW binfáját!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.3. Gagyí

Az ismert formális

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

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.4. Yoda

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.5. Kódolás from scratch

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13. fejezet

Helló, Liskov!

13.1. Liskov helyettesítés sértése

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.2. Szülő-gyerek

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.3. Anti OO

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.4. deprecated - Hello, Android!

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.5. Hello, Android!

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.6. Hello, SMNIST for Humans!

Fejleszd tovább az SMNIST for Humans projektet SMNIST for Anyone emberre szánt appá! Lásd az [smnist2_kutatasi_jegyzokonyv.pdf](#)-ben a részletesebb hátteret!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.7. Ciklomatikus komplexitás

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14. fejezet

Helló, Mandelbrot!

14.1. Reverse engineering UML osztálydiagram

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.2. Forward engineering UML osztálydiagram

UML-ben tervezzünk osztályokat és generáljunk belőle forrást!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.3. Egy esettan

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.4. BPMN

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.5. BPEL Helló, Világ! - egy visszhang folyamat

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.6. TeX UML

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15. fejezet

Helló, Chomsky!

15.1. Encoding

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.2. OOCWC lexer

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.3. l334d1c4

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.4. Full screen

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.5. Paszigráfia Rapszódia OpenGL full screen vizualizáció

Lásd vis_prel_para.pdf! Apró módosításokat eszközölj benne, pl. színvilág, textúrázás, a szintek jobb elkülönítése, kézreállóbb irányítás.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.6. Paszigráfia Rapszódia LuaLaTeX vizualizáció

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.7. Perceptron osztály

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16. fejezet

Helló, Stroustrup!

16.1. JDK osztályok

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.2. Másoló-mozgató szemantika

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.3. Hibásan implementált RSA törése

Készítsünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló: <https://arato.inf.unideb.hu/batfai.r> (71-73 fólia) által készített titkos szövegen.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.4. Változó argumentumszámú ctor

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.5. Összefoglaló

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17. fejezet

Helló, Gödel!

17.1. Gengszterek

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.2. C++11 Custom Allocator

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.3. STL map érték szerinti rendezése

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.4. Alternatív Tabella rendezése

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

```
<T>
```

szerepét!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.5. Prolog családfa

Ágyazd be a Prolog családfa programot C++ vagy Java programba! Lásd para_prog_guide.pdf!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.6. GIMP Scheme hack

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18. fejezet

Helló, !

18.1. FUTURE tevékenység editor

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.2. OOCWC Boost ASIO hálózatkezelése

Mutassunk rá a scanf szerepére és használatára! <https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/carlexer.ll>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.3. SamuCam

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.4. BrainB

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.5. OSM térképre rajzolása

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19. fejezet

Helló, Schwarzenegger!

19.1. Port scan

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.2. AOP

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.3. Android Játék

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.4. Junit teszt

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.5. OSCI

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

20. fejezet

Helló, Calvin!

20.1. MNIST

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

20.2. Deep MNIST

Mint az előző, de a mély változattal. Segítő ábra, vedd össze a forráskóddal a https://arato.inf.unideb.hu/batfai.norbert/2016/11/13/hello_samu_a_deep_mnist/ 8. fóliáját!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

20.3. CIFAR-10

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

20.4. Android telefonra a TF objektum detektálója

Telepítsük fel, próbáljuk ki!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

20.5. SMNIST for Machines

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

20.6. Minecraft MALMO-s példa

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

20.7. Általános

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

20.8. C

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

20.9. C++

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

20.10. Lisp

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

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

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