

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i>		
	Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert ÁCs Ujházi, Balázs	2019. december 9.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai
0.0.5	2019-03-06	A könyv feladatainak kidolgozásának a kezdete.	bujhazi
0.0.6	2019-04-26	Feladat-csokrok kész.	bujhazi
0.0.7	2019-04-30	Feladat leírások bővítése. Íráshibák javítása.	bujhazi

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.8	2019-05-09	Végléges könyv elkészült.	bujhazi

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	7
2.4. Labdapattogás	7
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	7
2.6. Helló, Google!	8
2.7. 100 éves a Brun téTEL	8
2.8. A Monty Hall probléma	9
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. Hivatalos nyelv	11
3.4. Saját lexikális elemző	12
3.5. l33t.l	13
3.6. A források olvasása	15
3.7. Logikus	17
3.8. Deklaráció	17

4. Helló, Caesar!	20
4.1. double ** háromszögmátrix	20
4.2. C EXOR titkosító	22
4.3. Java EXOR titkosító	23
4.4. C EXOR törő	23
4.5. Neurális OR, AND és EXOR kapu	23
4.6. Hiba-visszaterjesztéses perceptron	24
5. Helló, Mandelbrot!	25
5.1. A Mandelbrot halmaz	25
5.2. A Mandelbrot halmaz a std::complex osztállyal	25
5.3. Biomorfok	25
5.4. A Mandelbrot halmaz CUDA megvalósítása	26
5.5. Mandelbrot nagyító és utazó C++ nyelven	26
5.6. Mandelbrot nagyító és utazó Java nyelven	26
6. Helló, Welch!	27
6.1. Első osztályom	27
6.2. LZW	27
6.3. Fabejárás	28
6.4. Tag a gyökér	28
6.5. Mutató a gyökér	29
6.6. Mozgató szemantika	29
7. Helló, Conway!	31
7.1. Hangyszimulációk	31
7.2. Java életjáték	31
7.3. Qt C++ életjáték	31
7.4. BrainB Benchmark	32
8. Helló, Schwarzenegger!	33
8.1. Szoftmax Py MNIST	33
8.2. Mély MNIST	33
8.3. Minecraft-MALMÖ	33

9. Helló, Chaitin!	36
9.1. Iteratív és rekurzív faktoriális Lisp-ben	36
9.2. Gimp Scheme Script-fu: króm effekt	36
9.3. Gimp Scheme Script-fu: név mandala	36
III. Második felvonás	37
10. Helló, Arroway!	39
10.1. A BPP algoritmus Java megvalósítása	39
10.2. Java osztályok a Pi-ben	39
11. Helló, Berners-Lee!	40
11.1. Java 2 Útikalauz programozóknak I.	40
11.2. Java 2 Útikalauz programozóknak II.	41
11.3. Szoftverfejlesztés C++ nyelven	41
11.4. Bevezetés a mobilprogramozásba. Gyors prototípus-fejlesztés Python és Java nyelven (35-51 oldal)	41
12. Helló, Arroway!	42
12.1. OO szemlélet	42
12.2. „Gagyi”	42
12.3. Yoda	43
12.4. Kódolás from scratch	44
13. Helló, Liskov!	45
13.1. Liskov helyettesítés sértése	45
13.2. Szülő-gyerek	47
13.3. Anti OO	48
13.4. Ciklomatikus komplexitás	50
14. Helló, Mandelbrot!	52
14.1. Reverse engineering UML osztálydiagram	52
14.2. Forward engineering UML osztálydiagram	53
14.3. BPMN	54
14.4. TeX UML	55

15. Helló, Chomsky!	57
15.1. Encoding	57
15.2. OOCWC lexer	59
15.3. l334d1c4	60
15.4. Full screen	61
16. Helló, Stroustrup!	63
16.1. JDK osztályok	63
16.2. Másoló-mozgató szemantika	64
16.3. Hibásan implementált RSA törése	66
16.4. Összefoglaló	70
17. Helló, Gödel!	72
17.1. Gengszterek	72
17.2. C++11 Custom Allocator	73
17.3. STL map érték szerinti rendezése	74
17.4. Alternatív Tabella rendezése	75
18. Helló, !	78
18.1. FUTURE tevékenység editor	78
18.2. OOCWC Boost ASIO hálózatkezelése	78
18.3. SamuCam	79
18.4. BrainB	80
19. Helló, Lauda!	82
19.1. Port scan	82
19.2. AOP	83
19.3. Android Játék	84
19.4. Junit teszt	86
20. Helló, Calvin!	88
20.1. MNIST	88
20.2. Deep MNIST	88
20.3. Android telefonra a TF objektum detektálója	89
20.4. SMNIST for Machine	90

IV. Irodalomjegyzék	92
20.5. Általános	93
20.6. C	93
20.7. C++	93
20.8. Lisp	93

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás forrása: [link](#)

Megoldás forrása: [link](#)

A feladat megoldásában segített, tutorált engem: Tóth Donát.

Végtelen ciklusokat könnyedén létre lehet hozni, bár súlyos hibának számít sok esetben. egy előltesztelelős (while) ciklusból végtelen ciklus lesz, ha feltételnek csak egy egyszerű "true"-t adunk meg. Ha for ciklust szeretnénk használni, akkor feltételnek csak 2 db ";"-t kell megadnunk. Ezek a ciklusok 100%-ban fogják dolgoztatni a processzor egy darab magját. Ha több magot szeretnénk egyszerre használni, akkor threadeket kell alkalmaznunk a programunkban. Annyi új threadet kell nyitnunk, ahány magos a processzorunk. Azt, hogy egy végtelen ciklus 0%-on dolgoztassa a processzort úgy érhetjük el, hogy sleep-et alkalmazunk.

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!

A feladat megoldásában segített, tutorált engem: Tóth Donát.

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

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

```
        return false;
    }
main(Input Q)
{
    Lefagy(Q)
}
}
```

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

```
Program T1000
{
boolean Lefagy(Program P)
{
    if(P-ben van végtelen ciklus)
        return true;
    else
        return false;
}
boolean Lefagy2(Program P)
{
    if(Lefagy(P))
        return true;
    else
        for(;;);
}
main(Input Q)
{
    Lefagy2(Q)
}
}
```

Mit fog kiírni erre a T1000 (T1000) futtatásra?

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

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

Olyan programot írni lehetetlen, ami felismeri, hogy egy másik program jól működik-e.

2.3. Változók értékének felcserélése

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

Megoldás forrása: [link](#)

Tutoráltam, a feladat megoldásában segítettem neki: Tóth Donát.

A feladat megoldható egy csere változó létrehozásával, vagy ha nem szeretnénk csere változót, akkor összeadás, kivonás, illetve exor segítségével. Python nyelvben ez a feladat még ennél is könnyebb: `a,b = b,a`

2.4. Labdapattogás

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

Megoldás forrása: [link](#)

Tutoráltam, a feladat megoldásában segítettem neki: Tóth Donát.

Ehhez a programhoz használnunk kell a `curses.h` és az `unistd.h` nevű header fileokat. A `curses` tartalmazza azokat a függvényeket, amelyek segítségével a terminálon belül változtathatjuk a kurzorunk pozícióját, illetve kiiratásra is tökéletes. Az `unistd`-ből pedig a `usleep` nevű függvény jött segítségül, mivel ez "altatja" a programunk, hogy lassabban fusson.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás forrása: [link](#)

A MIPS (Million Instructions Per Second) jelentése: Egy millió utasítás másodpercenként. A számítógép számításának sebességének ez a mértékegysége. A BogoMIPS Linus Torvalds nevéhez fűződik. A BogoMIPS lényege, hogy egy körülbelüli értéket ad vissza a processzorunk sebességére. Ez az érték nem lesz pontos, ebből ered a Bogo a MIPS előtt a névben (a bogus angol szóból).

A forrásunkban a BogoMIPS-ből vesszük kölcsön a ciklus feltételét, mert így vissza kapjuk, hogy milyen hosszú a gépünkön a szóhossz. Ez az érték bitben értendő. A bitshiftelés lényege az, hogy addig lépteti a program balra a bitet, ameddig nem áll az egész csupa 0-ból. Ez azért lehetséges, mert minden léptetésnél egy 0-val egészíti ki jobbról.

```
#include <stdio.h>
int main (void)
{
    int h = 0;
    int n = 0x01;
    do
        ++h;
    while (n <= 1);
    printf ("A szohossz ezen a gepen: %d bites\n", h);
    return 0;
}
$ gcc szohossz.c -o szohossz
$ ./szohossz
A szohossz ezen a gepon: 64 bites
```

Egy kis információ a BogoMIPS programról: A ciklus feltételéről beszéltünk, azt alkalmaztuk feljebb. A programban van egy ticks nevű változó, amiben a processzoridőt tároljuk. A loops_per_sec változót shifteljük egyel a delay fügvénnyel. Ez után a ticks változó értékét felülírjuk az új értékkal, ami úgy jön létre, hogy a jelenlegi processzoridőből kivonjuk a ticks-ben tárolt értéket. Ha a ticks értéke nagyobb, vagy egyenlő, mint a CLOCKS_PER_SEC, akkor visszakapjuk a processzorunk sebességét.

2.6. Helló, Google!

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

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/blob/master/attention_raising/Source/pagerank.c

A PageRank algoritmus azon az elven működik, hogy a jobb minőségű weboldalak más jobb minőségű weboldalakra vezetnek. A PageRank kitalálója és névadója: Larry Page. A Google ezt az algoritmust használja a háttérben. A PageRank képlete: $PR(A) = (1-d) + d \left(\frac{PR(T_1)}{C(T_1)} + \dots + \frac{PR(T_n)}{C(T_n)} \right)$. Egy kis eligazítás: -A: aktuális weboldal. -T: olyan weboldalak, amelyek át vezetnek az "A" oldalra. - $PR(A)$: "A" oldal PageRank értéke. - $PR(T_n)$: Azok a weboldalak PageRank értéke, amelyek át vezetnek az "A" oldalra. - $C(T_n)$: A "T" weboldalon található "A" weboldalra vezető linkek száma. -d: damping factor (0, vagy 1).

A weboldalak adatait rögzítjük az "L" nevű tömbben. Be kerülünk a végtelen ciklusba, a PR elemeket lenullázzuk, aztán a PRv értékeit összeszorozzuk a mátrix-al és hozzáadjuk a PR elemekhez. A "tavolsag" függvényben PRv és PR értékeit kivonjuk egymásból, majd négyzetre emeljük. A függvény visszatérési értéke lesz ennek az értéknek a gyöke. Ha ez az érték kisebb, mint 0.000000001, akkor kilépünk a ciklusból. Egyéb esetben PRv megkapja PR értékét.

2.7. 100 éves a Brun téTEL

Írj R szimulációt a Brun téTEL demonstrálására!

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/brun.cpp>

A prímszámok fogalma: Az a szám prím, amely csak egyvel és önmagával osztható. Egy számot elő tudunk állítani prím számok szorzatával. Ikerprímek azok a prím számok, amelyek 2-vel térnak el egymástól. Pl.: 5 és 7. Kettő darab egymás utáni páratlan prím szám közötti különbség minimum 2, maximum végtelen.

A Brun tétele lényege az, hogy ha vesszük az ikerprímek reciprokát és összeadjuk őket, akkor az összegük konvergens lesz. Ennek van egy határa, amit nem lépnek át. A határ számát Brun konstansnak hívják. A mellékelt programunk ezt fogja végre hajtani, a végén pedig megmutatja nekünk a részeredményt.

A program működése részletesebben: -primes: n számig megkapjuk a prím számokat. -diff: megnézi, hogy 2 egymás utáni prím között mekkora a különbség. -idx: vissza adja, hogy a diffben hol van 2 különbség, mert ezek lesznek az iker prím párok. -1/t1primes: eltárol egyet az iker prím párokból és veszi a reciprokát. -1/t2primes: hozzá ad 2-t a 1/t1primes-hez, így megkapja a második számot és veszi a reciprokát. -rt1plust2: össze adja az előző két számot.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

Megoldás forrása: https://github.com/ujhazibalazs/MagProg1/blob/master/attention_raising_MontyHall_R_mh.r

Ha van 3db ajtó és az egyik mögött álmaink autója, akkor van 1/3-ad esélyünk kinyitni pontosan azt az ajtót. Viszont ha választunk egy ajtót, és valaki kinyit egy másikat a 3-ból ami mögött nincsen semmi, akkor a választott ajtónk mögött 1/3-ad esély van rá, hogy az autó lesz, viszont a 3. ajtó mögött 2/3-ad eséllyel lesz az autó, így ha van esély rá érdemes ajtót cserélni mindig arra, ami kimarad. Ez sokkal látványosabb, ha 3 ajtó helyett 100-at használunk. Így mi választunk egy db ajtót, de mennyi az esélye, hogy pont amögött lesz az autó? Nem sok. 1 a 100-hoz. De ha jön valaki és kinyitja a maradék 98 ajtót, ami mögött nincsen semmi, akkor az a 98/100-ad esély minden egy ajtóba tömörül és szinte biztos, hogy az autót ott találjuk. Emiatt minden érdemes a másik ajtó választani.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/turinggep.cpp>

```
#include <iostream>
int main()
{
    int a;
    int szamlalo = 0;
    std::cout<<"Add meg egy decimalis szamot!\n";
    std::cin >> a;
    std::cout<<"Unárisban:\n";
    for (int i = 0; i < a; ++i)
    {
        std::cout<<"| ";
        ++szamlalo;
        if (szamlalo % 5 == 0) std::cout<<" ";
    }
    std::cout<<'\n';
    return 0;
}
```

Az egyes számrendszer (unáris számrendszer) a természetes számok halmazán lehet használni. Ez a legegyszerűbb a számrendszerök közül, mivel az ujjunkon való számolás is ugyan ezen az elven működik. A számot úgy lehet leolvasni, hogy megszámoljuk hány vonalból áll. Könnyítés érdekében gyakran ötösével csoportosítják, ha eljutunk a negyedik vonalhoz utána az ötödikkel áthúzzuk kereszben az előző négyet.

A turing gépnek tízes számrendszerből kell egyes számrendszerbe váltania. Ezt úgy oldottuk meg a programban, hogy a vonalakat 1-esekkel helyettesítettük. A turing gép beolvassa a szalag cellájából a számot, ha '=' jelet talál, akkor addig von ki belőle 1-et folyamatosan, ameddig nem lesz 0 a szám. Ezt addig ismétli, ameddig az előtte lévő szám nem lesz 0. A mi példánkban a szám 10, tehát az első szám amit a végéről néz

az a 0. Mivel a 0 előtt még áll egy 1-es, ezért itt a program nem érhet véget, a 0-ból 9 lesz és az előtte álló 1-esből levon 1-et, tehát az 0 lesz. A program annyi 1-est tárol le, ahányat kivont és a végén annyi egyest kapunk amennyi a megadott decimális érték. Már csak annyi dolga van a programnak, hogy ki írjon annyi "!" karaktert (vonalat), ahány 1-est eltárolt és kész is vagyunk.

A program futtatása után:

```
$ g++ unaris.cpp -o unaris
$ ./unaris
Adj meg egy számot decimálisan!
10
Unárisan:
||||| |||||
```

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

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

A generatív nyelvtanban olyan szabályok vannak, amelyek jelsorozatok átalakítására vonatkoznak. A nyelv szabályokból és egy kezdő értékből épül fel. Noam Chomsky egy nyelvész volt a 20. században, ő volt az első aki generatív nyelvtant formalizált. Szerinte a nyelvtan 5 elemből épül fel: nem termális szimbólumok, termális szimbólumok, produkciós szabályok, előállítási szabályok és egy kezdő érték. Chomsky továbbá csoportosította is a generatív nyelvtanokat a következőképpen: rekurzív felsorolható nyelvtanok, környezetfüggő nyelvtanok, környezetfüggetlen nyelvtanok, reguláris nyelvtanok. Ebben a feladatban a környezetfüggő nyelvtant fogjuk vizsgálni.

```
S, X, Y „változók” – nemterminálisok
a, b, c „konstansok” – terminálisok
S → abc, S → aXbc, Xb → bX, Xc → Ybcc, bY → Yb, aY → aa ← →
aaX, aY → aa -képzési szabályok
Jelen esetben a kezdő szimbólumunk az S lesz.
A képzési szabályokat alkalmazva a következőket kapjuk:
S (S → aXbc)
aXbc (Xb → bX)
abXc (Xc → Ybcc)
abYbcc (bY → Yb)
aYbbcc (aY → aa)
aabbbcc
```

Ez a nyelv környezetfüggő, mivel a nyíl bal oldalán is megjelenik terminális szimbólum. Egy környezetfüggetlen nyelvben ez nem fordulhatna elő.

3.3. Hivatkozási nyelv

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

Megoldás forrása:

```
<postai_cím> ::= <név_rész> ", " <irányítószám_rész> " " <↔
    cím_rész>
<név_rész> ::= <személyi_rész> <keresztnév> | <név_rész> <↔
    keresztnév>
<személyi_rész> ::= <titulus> ".\"" "<vezetéknév>|<↔
    vezetéknév>" "
<cím_rész> ::= <kerület> <elnevezés> <közterület_tipus> <↔
    szám> <EOL>
<közterület_tipus> ::= "utca"|"tér"|"körút"|"lépcső"|"u."|"↔
    krt."
<irányítószám_rész> ::= <ország_kód>"-<irányítószám_belső>
    | <irányítószám_belső>
<irányítószám_belső> ::= <irányítószám> " <városnév> ", "
```

A BNF segítségével lehet környezetfüggetlen nyelvtant írni. Ezt használják a programozási nyelvek helyeségének megadására. A BNF kitalálója John Buckus, később Peter Naur változtatott a jelöléseken, kevesebb karaktert hagyott benne, így egyszerűbbé téve azt.

Az alábbi kód BNF leírással készült:

```
<postai_cím> ::= <név_rész> ", " <irányítószám_rész> " " <↔
    cím_rész>
<név_rész> ::= <személyi_rész> <keresztnév> | <név_rész> <↔
    keresztnév>
<személyi_rész> ::= <titulus> ".\"" "<vezetéknév>|<↔
    vezetéknév>" "
<cím_rész> ::= <kerület> <elnevezés> <közterület_tipus> <↔
    szám> <EOL>
<közterület_tipus> ::= "utca"|"tér"|"körút"|"lépcső"|"u."|"↔
    krt."
<irányítószám_rész> ::= <ország_kód>"-<irányítószám_belső>
    | <irányítószám_belső>
<irányítószám_belső> ::= <irányítószám> " <városnév> ", "
```

A BNF leírásban az értékadást " ::= "-vel tehetjük meg egy sima "=" helyett. A részeket ","-vel, vagy " "-el válasszuk el egymástól. A leírás lehet rekurzív is, tehát hivatkozhat saját magára is egy-egy rész.

A C nyelv eleinte a C89-et használta, később a C99-et és a legújabb verziója C11-et.

3.4. Saját lexikális elemző

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

Megoldás forrása:

Ebben a feladatban a lex nevű programot fogjuk használni. A lex egy szöveges fájlból be olvassa a lexikális szabályokat, majd ebből készít egy forrást. Ezt a forrást, mint bármely másikat át tudjuk fordítani g++-al, vagy gcc-vel. A lex 3 részből áll: -definíciós rész -szabályok -C nyelvű kód

Ha a forráskódot nézzük, akkor a részek az alábbiak: -Az első rész a következő:

```
% {  
#include <stdio.h>  
int realnumbers = 0;  
% }
```

-A második rész:

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

-A harmadik rész:

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

A lexer kipróbálását úgy tehetjük meg, hogy használjuk a lex parancsot a -o kapcsolóval:

```
$ lex -o lexikalisc lexikalisl
```

Utána a gcc-vel lefordítjuk a szokásos módon:

```
$ gcc lexikalisc -o lexikalisc
```

Ez után a ./lexikalisc parancs le is futtatja nekünk a programot. Ekkor adhatunk meg számokat a programnak, annyit adunk meg amennyit szeretnénk. Ha végeztünk egy CTRL+D betűkombinációt kell nyomnunk és akkor megáll. Szöveges fájlt is adhatunk a programnak, így nem kell nekünk beirogatni a számokat:

```
./lexikalisc <fajl_nev
```

3.5. I33t.I

Lexelj össze egy I33t cipher!

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/tree/master/attention_raising/Source/Leet

A leet (l33t) nyelvnek az a lényege, hogy az ABC bizonyos karakterét kicseréljük, vagy helyettesítjük más speciális karakterekkel, vagy esetleg számokkal. A leet ABC itt elérhető: <https://qntm.org/l33t>.

Ebben a programban ahhoz, hogy a leet nyelvet elkészítsük szükségünk lesz egy lexerre. Itt az első része:

```
#define L337SIZE (sizeof l337d1c7 / sizeof (struct cipher))
struct cipher {
    char c;
    char *leet[4];
} l337d1c7 [] = {
{'a', {"4", "4", "@", "/-\\"}},
{'b', {"b", "8", "|3", "|}"}, 
{'c', {"c", "(", "<", "{"}}, 
    ...
};
```

A #define parancs lényege az, hogy a forráskódban ahol azt írjuk, hogy L337SIZE, ott minden kicséri azt a mellette megadott kódra. Létrehozunk egy struktúrát cipher néven, ebben egy karakter típusú c nevű változó van és egy 4 karaktert tartalmazó karaktertömbre mutató mutatót. Ez után létrehozzuk a l337d1c7 nevű tömböt. Ebben lesz megadva minden betű és az ahhoz tartozó l33t ABC változata.

A második része:

```
{
int found = 0;
for(int i=0; i<L337SIZE; ++i)
{
    if(l337d1c7[i].c == tolower(*yytext))
    {
        int r = 1+(int)(100.0*rand()/(RAND_MAX+1.0)); //random szám generálás
        if(r<91)
            printf("%s", l337d1c7[i].leet[0]);
        else if(r<95)
            printf("%s", l337d1c7[i].leet[1]);
        else if(r<98)
            printf("%s", l337d1c7[i].leet[2]);
        else
            printf("%s", l337d1c7[i].leet[3]);
        found = 1;
        break;
    }
}
if(!found)
    printf("%c", *yytext);
```

```
}
```

Ha a l337d1c7 i-edik elemének a c karakter típusú változója megegyezik a yytext-re mutató karakter kisbetűs változatának, akkor generálunk egy random számot 1 és 100 között. Megnézzük, hogy a szám melyik leet tömb elemének felel meg és ki iratjuk a megfelelő változatát. Ha nincsen ilyen akkor a program az eredeti karaktert adja vissza.

A harmadik részben csak szimplán meghívjuk a forrásban a függvényt amit készítettünk.

3.6. A források olvasása

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

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

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



Bugok

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

i.

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

ii.

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

iii.

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

iv.

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

v.

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

vi.

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

vii.

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

viii.

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

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/blob/master/attention_raising/Source/forras.c

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

Ha a SIGINT jel nem volt figyelmen kívül hagyva, akkor a jelkezelő kezelje.

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

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

Ez egy egyszerű for ciklus, ami 0-tól indul minden esetben mikor lefut akkor növeli 1-el az i-t és megnézi, hogy megfelel-e a megadott feltételnek az i értéke. Tehát ameddig az i értéke kisebb, mint 5, addig le fog futni újra.

```
int a = 5;
int b = a++; //itt a b értéke 5 lesz, majd növeljük az a ←
              étékét 1-el
int c = ++a; //c értéke már nem 6 lesz, hanem 7, mivel itt ←
              előbb növelünk
```

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

Ez a for ciklus hibás mert érték adás történik ott, ahol az i változót növeljük.

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

Ennek a for ciklusnak a feltétel része érdekes, mivel az hibásan lett meg adva. Az első feltétellel nincsen semmi gond, de utána egy érték adás szerepel, emiatt a program hibás lesz.

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

Az f függvénynek adjuk az a változót kétszer is és mégegyszer meghívjuk az f-et ugyan ezzel a változóval. Ez a kód is hibás, nem lehet tudni, hogy mi a kiértékelés sorrendje.

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

A printf-el ki iratjuk az f függvény értékét a változóval, majd ki írjuk az a változó értékét is.

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

Ez a kód hibás, mert az `f` függvény megkapja az a változót, amit módosít és utána megpróbálja a `printf` kiírni az a értékét, de nem lehet tudni, hogy a függvény módosítás előtti, vagy utáni értéket írja ki a program.

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 ( \leftrightarrow  
SSy \text{ prim})) $  
$ (\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/ujhazibalazs/bhax/tree/master/attention_raising/Source/Logikus

A `forall` az univerzális kvantort jelenti és az `exists` az egzisztenciális kvantort. A `wedge` az implikációt jelenti és a `supset` pedig a konjunkciót. Az AR nyelvet használjuk ebben a feladatban, ez a nyelv tartalmaz egy olyan függvényt, ami vissza adja a rakkövetkező értéket. Ennek a jele: `S`. A feladatnál megadott formulákat értelmezzük:

1. minden x -hez létezik olyan y , amelynél ha x kisebb, \leftrightarrow akkor y prím.
2. minden x -hez létezik olyan y , amelynél ha x kisebb, \leftrightarrow akkor y prím, és ha y prím, akkor annak második \leftrightarrow rakkövetkezője is prím.
3. létezik olyan y , amelyhez minden x esetén az x prím, és \leftrightarrow x kisebb, mint y .
4. létezik olyan y , amelyhez minden x esetén az x nagyobb, \leftrightarrow és x nem prím.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referencia

- 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 mutatót visszaadó, egészet kapó függvény
- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutatót visszaadó, egészet kapó függvényre

Mit vezetnek be a programba a következő nevek?

- `int a;`
- `int *b = &a;`
- `int &r = a;`
- `int c[5];`
- `int (&tr)[5] = c;`
- `int *d[5];`
- `int *h();`
- `int *(*l)();`
- `int (*v(int c))(int a, int b);`
- `int (*(*z)(int))(int, int);`

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/blob/master/attention_raising/Source/deklaracio.cpp

```
1.: int a;
2.: int *b = &a;
3.: int &r = a;
4.: int c[5];
5.: int (&tr)[5] = c;
6.: int *d[5];
7.: int *h ();
8.: int *(*l) ();
9.: int (*v(int c))(int a, int b)
10.: 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!

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/blob/master/attention_raising/Source/tm.c

Az alsó háromszögmátrix tulajdonságai:

-Négyzetes: A mátrixnak ugyan annyi sora van, mint oszlopa.

-Főátlója feletti összes elem értéke 0.

-Ha a mátrixot i-vel és j-vel indexeljük, ahol i a sorok számát és j az oszlopok számát jelöli, akkor biztosak lehetünk abban, hogy ha j nagyobb, mint i, a mátrix eleme 0.

A program lényege: a standard output-ra ír egy nr változóban megadott számnyi soros és oszlopos háromszögmátrixot. A program úgy működik, hogy lefoglal annyi helyet a memóriában, amennyire szüksége lesz a mátrixnak. Ez után ellenőrzi, hogy sikeres volt-e lefoglalni azt. Ha sikeres volt, akkor kiírja a kétdimenziós tömb elemeinek az indexeit.

A memória foglalás a malloc nevű függvényel történik. A függvény ("malloc()") a következőképpen van deklarálva: void *malloc(size_t size). Tehát a függvénynek paraméterként egy size_t típusú értéket kell adnunk. Ez az érték a lefoglalandó memória blokk bájtban megadva. A függvény visszatérési értéke egy mutató, ami a lefoglalt memóriára mutat, ha nem sikerült memóriát foglalni, akkor null értéket ad vissza.

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

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

    if ((tm = (double **) malloc (nr * sizeof (double *))) == ←
        NULL)
    {
```

```
        return -1;
    }

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

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

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

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

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

Feltöljük a tömböt adatokkal, ezt több módon is megtehetjük:

-A tömb egy elemének az értékét explicit módon adjuk meg úgy, hogy a tömb különálló elemeire hivatkozunk.

-A tömbre mutató mutató értékét növeljük, így egy tömbre tudunk hivatkozni.

-Mutatót állítunk a negyedik tomb harmadik elemére.

-Mutatót állítunk a tömbben lévő harmadik tömb harmadik elemére. Itt használjuk a double **-ot.

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

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

```

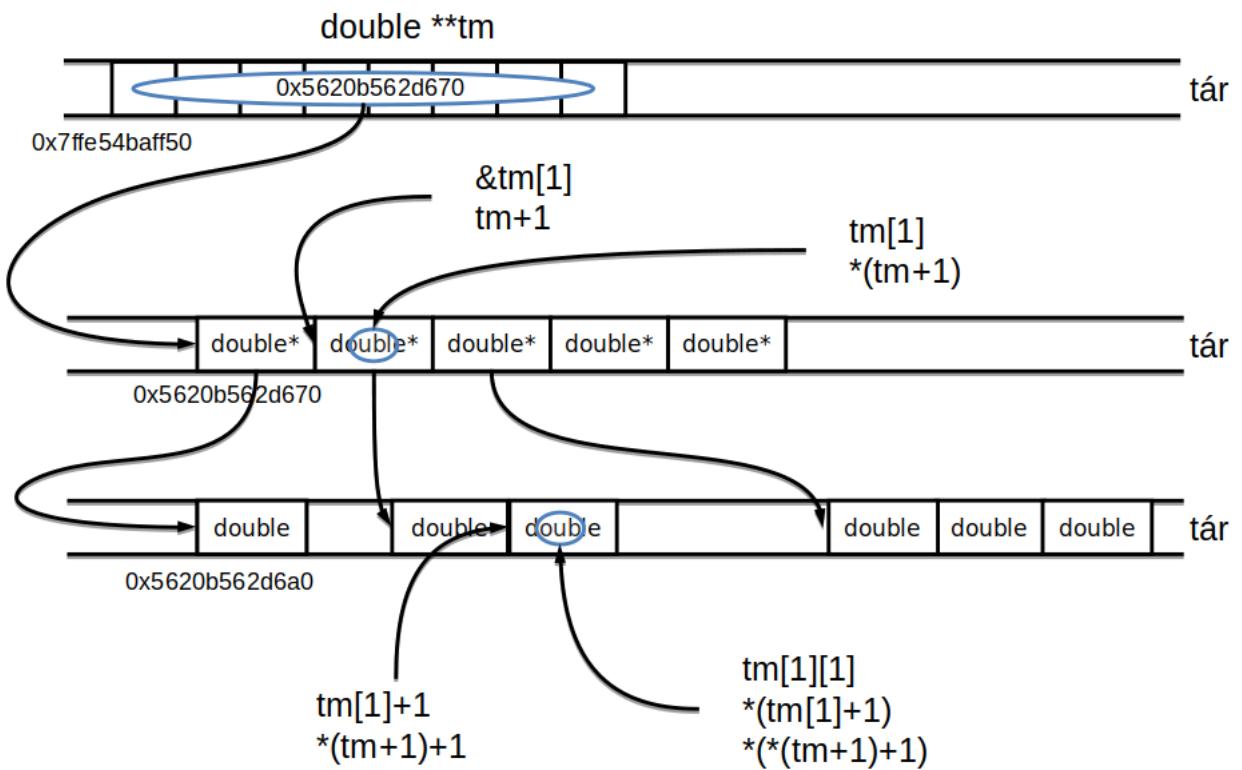
        for (int i = 0; i < nr; ++i)
            free (tm[i]);

        free (tm);

        return 0;
    }
}

```

Feltöltés után a program ki írja a standard output-ra a háromszögmátrix elemeit, majd felszabadítja a lefoglalt memóriát.



4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/Exor.c>

Ebben a programban a kizártó vaggyal(XOR) titkosítunk bármilyen szöveget, inputot. A program elején létrehozunk 2db karakter tömböt, az egyiket a kulcsnak, a másikat a buffernek. Ezután létrehozunk 2db egész(int) típusú változót, az egyiket azért, hogy tudjuk, a kulcs hanyadik karakterénél járunk titkosítás közben, a másikat pedig azért, hogy tudjuk hány bajtot olvastunk be a megadott szövegből, inputból. A 17. sorban egy egész típusú változóban eltároljuk az első paraméternek megadott kulcs hosszát, mivel ez egy karaktersorozat, ezért a strlen-el megszámoltatjuk hány db. Ezután egy while ciklusban karakterenként titkosítjuk a megadott szöveget exorral és minden karakter titkosítása után a kulcs következő karakterére

lépünk, ha a végére értünk, akkor kezdjük az elejéről. Ha while ciklus végzett a titkosítással már csak annyi a dolgunk, hogy ki irassuk a titkos szöveget ami létrejött.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/exorjava>

Ez a program ugyan arra a célra lett létrehozva, mint a felette lévő, annyi különbséggel, hogy másik nyelven íródott. A program Java nyelvben készült, ami annyiban tér el az eddig megszokott nyelvünktől, hogy objektum-orientált. Mint láthatjuk a programban egy "ExorTitkosító" nevű függvényünk van a "Main"-en kívül. Ez az "ExorTitkosító" ugyan azt csinálja lényegében, mint amit feljebb leírtam. A "Main"-ben viszont annyi különbség van, hogy egy try, catch-et használunk. A try-ban létrehozzuk az elején elkészített "ExorTitkosító" objektumunkat és átadjuk neki a megfelelő paramétereket: kulcs, bemeneti és kimeneti csatorna. Ha ez hiba nélkül lefut, akkor a program kilép, de ha bármi hiba történne a catch el fogja kapni azt és tudatja velünk mi volt a baj.

4.4. C EXOR törő

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

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/ExorTores.c>

Ez a program a fentebb készített exor titkosítóval titkosított szövegeket töri fel és árulja el nekünk, hogy mi volt a kulcs. Tulajdonképpen bruteforce-ot alkalmazunk, ami azt jelenti, hogy minden létező kombinációt megpróbálunk és majd idővel sikerül. Ez nem túl jó taktika, ha a kulcs bonyolult és hosszú, mert nagy kapacitású gép kell a töréséhez és nagyon sok idő. A programban tudnunk kell a kulcs hosszát, majd annyi for ciklust készítünk egymásban amennyi karakter hosszú a kulcs és így minden kombinációt kipróbálhatunk a szövegen. Ha a szöveg "tisztnak" tűnik, azaz előfordulnak benne gyakori szavak, pl: hogy, nem, az, ha, akkor nagy valószínűséggel sikerült feltörni (kivéve, ha ezek a szavak nem fordulnak elő a tiszta szövegen). A végén csak ki iratjuk for ciklusok indexei által megkapott kulcsot és a tiszta szöveget.

4.5. Neurális OR, AND és EXOR kapu

R

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/nn.r>

Az AND (és) csak abban az esetben tér vissza 1-es értékkel, ha A AND B esetében A és B is 1-es értékkel rendelkezik. minden egyéb esetben pl: $A = 0 B = 0$, $A = 1 B = 0$, $A = 0 B = 1$ 0 értékkel tér vissza. Az OR (megengedő vagy) minden esetben 1-es értékkel tér vissza, kivéve ha A OR B esetében mind a 2 0 értéket tartalmaz. pl: $A = 0 B = 0$ esetben A OR B = 0, minden más esetben A OR B = 1 pl: $A = 1 B = 1$, $A = 0 B = 1$, $A = 1 B = 0$. Az XOR (kizáró vagy) akkor tér vissza 1-es értékkel, ha A XOR B esetében minden két elem értéke eltérő. Ha a két elem értéke megegyezik A XOR B = 0 lesz. pl: $A = 1 B = 0$ és $A = 0 B = 1$ esetében lesz A XOR B értéke 1. $A = 1 B = 1$, $A = 0 B = 0$ esetében lesz A XOR B értéke 0.

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/mlp.hpp>

Első sorban tisztázzuk, hogy mi az a perceptron. A perceptron egy olyan algoritmus, ami felügyelten tanít bináris osztályokat a gépnek. A gépi tanulás (machine learning) része.

A kódot C++ nyelvben írták meg.

Először is fel kell telepítenünk a libpng++ nevű csomagot: **sudo apt-get install libpng++ -dev**. Erre a csomagra azért van szükségünk, mert fordításnál kellene fog a -lpng nevű kapcsoló. A forráskód a mandelpng.cpp-ben található. A kódot ezzel a parancssal fordítjuk le: **g++ mandelpng.cpp -o mandel -lpng** Ezután futtatjuk: **./mandel mandel.png** Futtatásnál a forráskódnak kell egy kép bemenetként. Ez a kép lesz a mandel.png. Használjuk az ml.hpp headerben található perceptron osztályt.

A program:

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

int main (int argc, char **argv)
{
    png::image<png::rgb_pixel> png_image (argv[1]);

    int size = png_image.get_width() * png_image.get_height();

    Perceptron* p = new Perceptron (3, size, 256, 1);

    double* image = new double[size];

    for (int i = 0; i<png_image.get_width(); ++i)
        for (int j = 0; j<png_image.get_height(); ++j)
            image[i*png_image.get_width() + j] = png_image[i][j].red;

    double value = (*p) (image);

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

    delete p;
    delete [] image;

}
```

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

A Mandelbrot-halmaz a komplex számsíkon ábrázolva, fraktálalakzatot vesz fel. A fraktálok: végtelenül komplex geometriai alakzatok, két gyakori tulajdonsággyal rendelkeznek: Az első: A fraktálok határoló vonalai vagy -felületei végtelenül „gyűröttek” vagy „érdesek”, illetve „szakadásosak”. A második: A rendkívül problematikus, de jól érzékelhető önhasonlóság. A Mandelbrot-halmazt Benoît Mandelbrot fedezte fel, és Adrien Douady és John Hamal Hubbard nevezte el róla 1982-ben.

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/3.1.2.cpp>

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

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/mandelpng.cpp>

Ez a feladat szinte teljesen ugyan olyan mint az előző. Az előző feladatban a komplex számokat két darab különböző változóban tároltuk el, az egyikben volt a valós rész, a másikban pedig a képzetesz rész. Ez a feladat viszont a complex könyvtárat használva megoldható 1 változóval is.

5.3. Biomorfok

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/biomorf.cpp>

Ez a feladat a Mandelbrot halmaz helyett a Julia halmazhoz kötődik. A Mandelbrot halmaz nagyobb és magába foglalja a Julia halmazt is. A Julia halmaznál a c konstans, viszont a Mandelbrotnál a c is változó. A Biomorfok megtalálója Clifford Pickover. Ő készített egy programot, ami a Julia halmazt rajzolta ki, és eközben fedezte fel őket. A programjában volt egy hiba és az okozta a felfedezést. A mi forrásunk is ezzel a módszerrel rajzolja ki a biomorfokat, annyi különbséggel, hogy a mi forrásunk a Mandelbrot programra épül.

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/cuda.cu>

A CUDA egy olyan API, amit az Nvidia cég készített. Ez az API sok időt tud megspórolni nekünk például a Mandelbrot halmaz képének generálásánál. A CUDA-nak a lényege, hogy készít egy 600x600-as táblázatot (gridet) és a táblázat minden egyes cellájához rendel egy szálat (threadet) így a videókártyában lévő kisebb processzorok képesek minden szálat egy időben számolni. Természetesen a CUDA-t csak az tudja igénybe venni, aki rendelkezik Nvidia videókártyával, ráadásul olyannal, ami támogatja is ezt a CUDA-t. Ha a hardware részét biztosítottuk, akkor az nvidia CUDA toolkitet kell már csak feltelepítenünk a gépünkre és használható lesz.

5.5. Mandelbrot nagyító és utazó C++ nyelven

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

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/frakablak.cpp>

Telepítési útmutató: sudo apt-get install libqt4-dev. A Mandelbrot-halmazhoz Qt nyelvben készült a GUI. Ahoz, hogy a program leforduljon 4 darab fájlra van szükségünk, ezeknek a fájloknak ugyan abban a mappában kell lenniük. Ha ez kész, akkor a mappán belül ki kell adni a qmake -project parancsot. Ez egy pro kiterjesztésű fájlt hoz majd létre. Ezt a fájlt kell kiegészítsük QT += widgets nevű sorral. Ha ez kész, akkor a make parancsot kell használjuk újból, csak most a pro kiterjesztésű fájlra: qmake valami.pro. Ha ez kész, akkor a mappánkban megjelenik egy Makefile, ezt egy make parancs kiadásával tudjuk használni. Végül ha ez is kész, akkor lesz egy futtatható fájlunk a mappánkban.

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/mandelbrotnagyito.java>

Ebben a feladatban a Qt nyelvben megírt nagyítót kell elkészítenünk Java nyelven. Ha lefuttatjuk a programot, akkor észre vehetjük, hogy nem úgy nagyít, hogy rá közelít a képre, hanem egy új lapon nyitja meg azt a képrészletet, ahova nagyítottunk. Azt is láthatjuk, hogy a forrásban egy másik Java programot importáltunk és ezt használjuk. Eddig #include parancsot használtunk C-ben és C++-ban, hogy egy másik fájlt használunk, Java-ban viszont importálni kell azokat a fájlokat amikkel dolgozni szeretnénk.

6. fejezet

Helló, Welch!

6.1. Első osztályom

Megoldás Forrása: https://gitlab.com/ujhazibalazs/bhax/tree/master/attention_raising/Source/Elsoosztaly

Az objektum-orientált programozás (object-oriented programming) megkönnyíti a programozók feladatát. Úgy képzelhetjük ezt el, hogy pl.: létrehozunk egy kutyák osztályt és ebből tudunk kutyákat objektetek készíteni. Ez a kutyák objekt minden kutyák osztályban lévő tulajdonságot megkap, így az összes lokális változóját kedvünkre változtathatjuk, még sem lesz semmi köze a többi kutyához és így nem ronthatja el azokat. Objektum-orientált nyelvek: Java, C#, C++.. stb. Ebben a feladatban egy polártranszformációs generátort készítünk. A polártranszformációs generátor algoritmusa nagyon nehéz matematikai egyenletekből épül fel, de szerencsére nekünk a programban nem kell semmilyen matematikával foglalkoznunk. A forrásban lényeges megjegyezni, hogy egy lépés 2 darab számot hoz létre, azaz minden páros lépésnél kell számolnunk és minden páratlan lépésnél csak az előző számításból elkészült második számot adjuk meg. Ezt a "nincsTarolt" bool típusú változó fogja nézni, ha igaz, akkor van benne szám, ha hamis, akkor számolnunk kell.

6.2. LZW

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

Megoldás forrása: <https://github.com/ujhazibalazs/MagProg1/blob/master/z3a7.cpp>

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.

Ahhoz, hogy egy bináris fát elkezdjük számolni, szükségünk lesz egy gyökérmutatóra. Ez a gyökérmutató fog a 2 darab gyermekére mutatni. Lesz egy bal oldali gyermek (nullás gyermek) és egy jobb oldali gyermek (egyes gyermek). Ha ezt objektum-orientált nyelvben írjuk, akkor a fent említett módszerrel elkezdjük az osztályt a fának és abból kezdjük az objektumokat. Ha C-ben szeretnénk ezt megírni, akkor struktúrákat kell alkalmaznunk. Viszont itt nagyon figyelnünk kell a memóriakezelésre, ha lefoglaltuk a memóriát a fa gyökerének és a gyermekéinek és azoknak a gyermekéknak is és stb, akkor a végén soha nem szabad elfelejteni, hogy fel kell ezt a memóriát szabadítanunk a "delete"-el.

A main-ben a következő történik: Az "uj_elem" függvény létrehozza a gyökeret, ez után nullpointerré teszi a gyökér két mutatóját. A "fa" pointer a gyökerünkre mutat. Ezek után beolvassunk: Ameddig el nem fogy a bemenet, addig 1 bájt nyit olvasunk és azt bitenként ÉS-eljük 128-al. Ez nyolcszor történik meg, mert 1 bájt 8 bit. Ez után shifteljük a biteket balra, így lesz a végén egy 0.

6.3. Fabejárás

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

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/tree/master/attention_raising/Source/Fabejaras

A bináris fákat 3 féle képpen lehet bejárni: -inorder: Ezzel a módszerrel járjuk be az előző feladatban alapértelmezetten. Ez a bejárási mód úgy működik, hogy először vesszük a bal oldali gyermeket, majd jön a gyökér és végül a jobb oldali gyermek. A nevéről is egyszerűen kideríthető ugyanis az "in" angol jelentése: valami-ben, valami között. A név mindenkor a gyökér helyzetére utal, a gyökér minden esetben a két gyermek között helyezkedik el. -preorder: A preorder fa bejárási módszer úgy zajlik le, hogy a gyökérelem kerül legelölre és utána következik a bal oldali gyermek, végül a jobb oldali gyermek. Ennek a bejárási módszernek is a nevéről könnyen megjegyezhető a működése. A "pre" szó angol jelentése: elő, előtti. A gyökér helyzete ebben az esetben elől van. -postorder: Az utolsó bejárásunk a postorder. Ebben az esetben az első helyen a gyökér bal oldali gyermek helyezkedik el, utána következik a gyökér jobb oldali gyermek és végül a gyökérelem. A bejárás nevéről kideríthető a gyökér helyzete itt is. A "post" szó angol jelentése: valami után. Tehát a gyökér helyzete a gyermeket után lesz. A programunkban nem bal és jobb oldali gyermekként vannak nyilvántartva a gyökér mutatói, hanem nullás és egyes gyermekként. Itt a nullás gyermek reprezentálja a bal oldali gyermeket és az egyes gyermek a jobb oldali gyermeket.

6.4. Tag a gyökér

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

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/blob/master/attention_raising/Source/Tag/z3a7.cpp

A legnagyobb különbség az lesz ez a feladat és a C-ben megírt párja között, hogy itt struktúrák helyett osztályokat használunk a nyelv előnyei miatt. Készítünk egy Binfa osztályt, ehhez kell egy konstruktur, ez beállítja, hogy a gyökér címére mutasson a fa mutató. Mivel hoztunk létre konstruktort, ezért a "rule of three" miatt biztos, hogy szükségünk lesz destruktora és másoló operátorra is. Operátor türterhelést fogunk használni a shifteléshez, ezt a mainben látjuk majd. Az operátor túlterhelésnél egy létező operátor működését tudjuk felül bírálni. Fontos, hogy új operátort nem lehet C++-ban létrehozni. Két darab "kiir" nevű függvényünk is van, ez mint már tudjuk, úgy lehetséges, hogy a paramétereik különbözőek. Ebben a feladatban már filekezelést is alkalmazunk. A konstruktur egyetlen feladata az, hogy gyökért ad a csomópontnak és nullpointerre állítja a mutatóit. A privát részben lett a fa mutató deklarálva, ez csomópontra mutat. Itt van még a másoló konstruktur és a másoló értékkadás is, de a programban nincsen használva. A gyökér elem csomópont típusú, tehát a fa tagja.

6.5. Mutató a gyökér

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

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/blob/master/attention_raising/Source/Mutato/z3a8.cpp

A jelenlegi feladatunk célja, hogy a gyökér ne legyen tagja a fának, csak egy mutató legyen. A gyökér típusát át állítjuk csomópontra mutató mutatóra. Miután ezt megtettük a konstruktörben is módosítanunk kell: a gyökér egy új csomópontra mutató mutató legyen, és mutasson a gyökér mutatóra. A destruktort is megfelelően át kell írjuk. Végül a programban az összes helyen előforduló gyökér referenciát törölünk kell.

6.6. Mozgató szemantika

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

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/blob/master/attention_raising/Source/z3a9.cpp

Ha egy elkészült fát szeretnénk másolni, akkor gondolhatunk arra, hogy a pointereket egyszerűen át másoljuk egy új fa struktúrájába. Ez azért nem lenne jó megoldás, mert amikor az eredeti fa törlésre kerül, akkor az új fának a mutatói nem mutatnának semmire. A jó megoldás az, ha készítünk egy teljesen új fát. Akkor foglalunk a memóriában helyet az új elemeknek, amikor elérjük a fának a levél elemét. Így ha törlődik az egyik fa, a másik ugyan úgy megmarad.

Az alábbi kódcsipet a mozgató konstruktör. Ez egy "jobbérteket" vár paraméterül. Értékadásnál például "A = B": A balérték az "A", tehát a balérték mindenek értékét adunk. A jobbértek a "B", tehát az amelyiket értékül adjuk a balértéknek. A "gyoker" pointert nullpointerre (nullptr) állítjuk, ezután meghívjuk az std::move() függvényt a paraméterül adott "regi"-vel. Az std::move() függvény jobbérteket készít a paraméterként megadott változóból. A *this lesz az az LZWBInFa ahova mozgatva lett.

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

Az alábbi kódcsipetben történik az operátor túlterhelés:

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

Fontos megemlíteni, hogy ez nem mozgat semmit sehol sem, csak jobbérték-referenciát csinál a neki átadott struktúrából.

7. fejezet

Helló, Conway!

7.1. Hangyszimulációk

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

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/tree/master/attention_raising/Source/Ant

A program a hangyák kommunikációját mutatja be. A képernyőt felbontjuk cellákra és minden cellában elhelyezünk egy hangyat. minden cellának van egy feromon szintje és a hangyáknak az a feladata, hogy az olyan cellák felé haladjanak, ahol magasabb a feromon szint. A cellák feromon értéke folyamatosan az idő teltével csökken, viszont ha egy hangya belép a cellába, akkor megnő ez az érték.

7.2. Java életjáték

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

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/tree/master/attention_raising/Source/Lifegame

A életjáték program lényege, hogy sejtek életét szimulálhatjuk úgy, hogy mi határozzuk meg, hogyan maradhatnak életben, hogyan születhetnek és, hogy mitől halhatnak meg. Ezt Melvin Conway készítette el 1970-ben. Conway szimulációjának 3 szabálya volt: -A sejt életben marad, ha 2, vagy 3 szomszédja van. -A sejt meghal, ha 3-nál több, vagy 2-nél kevesebb szomszédja van. -Egy új sejt jön létre, ha egy üres cellát 3 sejt vesz körül.

A feladatban a sikló-kilövőt kell megvalósítani. Ez úgy lehetséges, hogy fix cellákba helyezzük a sejteket és azok felfele indulnak el folyamatosan.

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/tree/master/attention_raising/Source/Lifegame

Ebben a feladatban ugyan azt készítjük el, mint az előzőben. A különbség az lesz, hogy Qt nyelven fogjuk most megírni a programot. Az életben maradás, születés és halálozás szabályai ugyan azok mint az előző feladatban. Itt is a sikló-kilövőt kell létrehozzuk.

7.4. BrainB Benchmark

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/tree/master/attention_raising/Source/BrainB

A BrainB egy olyan program, amely teszteli az agunk kognitív képességeit. Ezt arra használják, hogy a tehetségesebb e-sportolókat kiválasszák az emberek közül. A program egy olyan "játék", amelyben figyelnünk kell a karakterünket és az a lényeg, hogy ne veszítsük el, ha elveszítjük akkor minél hamarabb megtaláljuk. A program ezt pontozza és ez által kapunk egy eredményt, hogy mennyire jól teljesítettünk.

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

aa Python

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/tree/master/attention_raising/Source/Softmax_Passz.

8.2. Mély MNIST

Python

Megoldás forrása:

Passz.

8.3. Minecraft-MALMÖ

Megoldás forrása: https://gitlab.com/ujhazibalazs/bhax/tree/master/attention_raising/Source/Minecraft

A Project Malmo egy olyan kutatás, amely 2015-ben jött létre. A kutatás lényege, hogy a Minecraft nevű népszerű játékhoz próbálnak olyan mesterséges intelligenciát készíteni, amely meg tudja tanítani magát arra, hogy hogyan csináljon egyszerű dolgokat. Pl.: felmászni a világ legmagasabb pontjára. Az ágens úgy indul mint egy valódi játékos eleinte, nem tud semmit a környezetéről és arról sem, hogy mit kellene elérnie. Fel kell fognia és meg kell értenie mi van a környezetében és el kell dönteni, hogy mi az ami fontos (minél magasabbra menni), és hogy mi az ami nem fontos. Ez az ágens trial and error alapon tanul, tehát hibákat követ el és ettől lesz okosabb. Rewardok segítségével tudja azt, hogy közelebb jutott a céljához.

A Project Malmo hivatalos github repóval rendelkezik: <https://github.com/Microsoft/malmo>. Telepíteni több módon is lehet, elérhető a Pip-en is és pre-built verzióban is. Mi Windows Operációs rendszeren a pre-built verziót vesszük igénybe. Hasznos link lehet az alábbi, ahol részletezik a program használatát angol nyelven: <https://github.com/Microsoft/malmo#getting-started>. Ha letöltöttük a pre-built verziót, akkor ki kell tömörítsük. Ez után fel kell telepítenünk az OpenJDK Java 8-at, ha még nincsen fel telepítve a gépünkön. Ha ezzel is meg vagyunk, akkor a PowerShellben kell egy pár parancsot lefuttatnunk:

```
PS D:\> Set-ExecutionPolicy -Scope CurrentUser Unrestricted
PS D:\> cd .\Malmo-0.37.0-Windows-64bit_withBoost_Python3.6\
PS D:\Malmo-0.37.0-Windows-64bit_withBoost_Python3.6> cd scripts
PS D:\Malmo-0.37.0-Windows-64bit_withBoost_Python3.6\scripts> .\malmo_install.ps1
```

Ha ez sikerült, akkor indíthatjuk a játékot ugy PowerShellen keresztül:

```
PS D:\Malmo-0.37.0-Windows-64bit_withBoost_Python3.6\scripts> cd ..
PS D:\Malmo-0.37.0-Windows-64bit_withBoost_Python3.6> cd Minecraft
PS D:\Malmo-0.37.0-Windows-64bit_withBoost_Python3.6\Minecraft> .\launchClient.bat
```

Ha a játék elindult, akkor minden jól csináltunk. Ahhoz, hogy az ágensel tudjunk kommunikálni, egy újabb PowerShell-t kell indítanunk. Ebben a terminálban a lefuttatott parancsok alapján fog viselkedni az ágensünk. Az alábbi utasítások adhatóak az ágensnek:

```
agent_host.sendCommand("turn -1") //balra fordul, teljes sebességgel
agent_host.sendCommand("move 1") //előre halad, teljes sebességgel
agent_host.sendCommand("pitch 1") //felfele tekintés, teljes sebességgel
agent_host.sendCommand("strafe -1") //bal oldalra mozdul, teljes sebességgel
agent_host.sendCommand("jump 1") //folyamatos ugrálás
agent_host.sendCommand("crouch 1") //folyamatos guggolás
agent_host.sendCommand("attack 1") //folyamatos támadás
agent_host.sendCommand("use 1") //itemek folyamatos használata
```

Ahhoz, hogy küldetést adjunk a Minecraft hősünknek, egy újabb parancsra van szükségünk: my_mission = MalmoPython.MissionSpec(). Miután használjuk ezt a parancsot, létre fog jönni egy XML sort. A missionXML rész tartalmazza a küldetés rövid leírását és a játékmódot. A küldetés lehet akár időre is, ezt miliszekundban kell megadni. A küldetés biome-ját is meg adhatjuk a FlatWorldGenerator generatorString-el. A programban lévő változók értelmezése: -stevex, stevey, stevez: A hősünk jelenlegi pozíciójának a koordinátái. -steveyaw: Jelenleg milyen irányba halad a hősünk. (Észak: 180, dél: 0, nyugat: 90, kelet: -90) -stevepitch: Milyen nézetet használunk. Ahhoz, hogy a hősünk akadálymentesen tudjon közelekedni

tudnia kell arról, hogy milyen blockok veszik körül. Körbe fogja venni őt egy 3x3-as grid és az ezen belül előforduló blockokról pontosan tudni fogja, hogy mik azok. Ha a hősünk előtt nincs akadály, akkor halad tovább egyenese. Ha beleütközik valamibe, akkor megfordul. Ha vízbe esik, akkor abból ki kellene ugrania, viszont ez nem lehetséges ha 1 blocknál magasabb a fala annak a lyuknak, ahova be esett, hiszen a hősünk csak 1 blocknyi magasra tud ugrani.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás forrása:

Passz.

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 forrása:

Passz.

9.3. Gimp Scheme Script-fu: név mandala

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

Megoldás forrása:

Passz.

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.

10. fejezet

Helló, Arroway!

10.1. A BPP algoritmus Java megvalósítása

Megoldás forrása:

Passz

10.2. Java osztályok a Pi-ben

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

Megoldás forrása:

Passz

11. fejezet

Helló, Berners-Lee!

11.1. Java 2 Útikalauz programozóknak I.

1. fejezet - Nyolc nap alatt a nyelv körül

1.1: Megírjuk az első "Hello, World!" programunkat Java nyelvben. A forráskódban az osztály a HelloVilag nevet kapta, ezért a forráskódot tartalmazó fájl neve is HelloVilag.java lett. A forráskódunkat a "javac HelloVilag.java" parancssal fordítjuk és a "java HelloVilag" parancssal futtatjuk.

1.2: Készítünk egy olyan Java programot, amelyet HTML oldalba építve tudunk futtatni. A program neve Viszlat.java, ezt az 1.1-ben említett módon fordítjuk, de még nem lesz futtatható. Készítünk egy HTML oldalt, amibe az applet parancssal beágyazzuk az előbb elkészített Java programunkat. A végeredményt megnézhetjük a böngészőnkben, vagy a Java által biztosított appletviewer parancssal. Az oldalba beépített kód le fog töltődni minden látogató gépére és ott fog lefutni.

1.3: Néhány szám faktoriálisát számítja ki a következő kis programunk. Az alábbi változókkal ismerkedünk meg: boolean, char, byte, short, int, long, float, double. Az értékadás az egyenlőség jellel valósul meg. Addig, ameddig egy változónak nincsen értéke, addig nem szabad felhasználni, mert ez hibához vezet. Előltesztelős ciklus: while(feltétel), ahol a feltétel egy boolean kell legyen, azaz egy logikai érték (true, vagy false). A Java nyelvben a névtűlterhelés megengedett, viszont nincsen operátor túlterhelés.

1.4: Konstansokat a final parancssal adhatunk meg. A konstansokra egy tetszőleges névvel hivatkozhatunk, viszont nem változhat az értéük (Pl.: final static double PI = 3.14;). A Java nyelvben már nemcsak az ASCII karaktereket, hanem akár magyar karaktereket is használhatunk azonosítóként. Ez azért lehetséges, mert a Java 16 bites Unicode karaktereket használ.

1.5: Három féle módon írhatunk megjegyzéseket: egysoros megjegyzés: //, többsoros megjegyzés: /* */, dokumentációs megjegyzés: /** */. A programunkhoz a javadoc parancssal készíthetünk dokumentációt, ebben a dokumentációs megjegyzések látszódni fognak. Sok HTML oldalból áll, az index.html-el tekinthetjük meg.

1.6: Osztály létrehozása a class parancssal lehetséges. Egy osztály adattagokat(változókat) és metódusokat(függvényeket) tartalmazhat. Elemek láthatósága: public: külvilág számára látható, protected: leszármazottak számára látható, private: senki más számára nem látható. Ha nem jelölünk meg láthatóságot, akkor az adott elem csak az adott csomagban lesz látható, a külvilág számára nem. A Java nyelvben a karakterláncok kezelésére egy külön osztály van, ez a String osztály, ebben is különbözik a C++ nyelvtől. Objektumokat a new parancssal hozhatunk létre, ez lefoglalja a számukra szükséges memóriát. Az objektum egy elemére

úgy hivatkozhatunk, hogy "Objektumnév.elemnév". Ha a static kulcsszót használjuk, akkor az elem nem egy-egy objektumhoz fog tartozni, hanem az osztályhoz. A memória felszabadítást a Garbage Collector végzi. Nincsen módunk egy objektumot megszüntetni. Ha nem használunk egy objektumot, akkor idővel ez a Garbage Collector megszünteti azt.

1.7:

11.2. Java 2 Útikalauz programozóknak II.

11.3. Szoftverfejlesztés C++ nyelven

11.4. Bevezetés a mobilprogramozásba. Gyors prototípus-fejlesztés Python és Java nyelven (35-51 oldal)

A Python egy magas szintű, objektumorientált programozási nyelv. Akár script nyelvnek is hívhatjuk. Egyik nagy előnye, hogy platform független. Prototípusok készítésére és azok tesztelésére használhatjuk. Ezen kívül bármilyen alkalmazás elkészítésére is alkalmas. Egy komponense képes több modullal is együtt működni. A Python nyelv rengeteg csomagot és ezen kívül nagyon sok beépített eljárást foglal magába. Másik nagy előnye, hogy nincsen szükségünk fordításra, ha Python-t használunk.

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/UDPROG> (16-22 fólia) Ugyanezt írjuk meg C++ nyelven is! (lásd még UDPROG repó: source/labor/polargen)

Link: [polargen.cpp](#) Link: [polargen.h](#) Link: [PolarGenerator.java](#)

A programunk egy polargen.h nevű header fájlt vesz igénybe. A header fájlban a nincsTarolt nevű változónk alapértelmezetten igaz, ezért az if-ben lévő kódblokk fut le először.

Ebben a kódblokkban egy random számgenerátorral és matematika segítségével kapunk két számot.

Az egyik számot a tarolt nevű változóban tároljuk el, a másik számot a kovetkezo függvény visszatérési értékeként kapjuk meg. Miután a tarolt változóba kerül egy szám, a nincsTarolt hamis lesz. Így csak 2 számonként fogunk újakat generálni.

A main függvényben csak a kiiratás történik. Ha generálunk számokat, akkor a return-al visszakapott érték lesz mindenki először és utána (mivel a nincsTarolt hamis) a return az eltárolt értéket adja majd vissza és ezt írjuk ki.

12.2. „Gagyí”

Az ismert formális

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

tesztkérdéstípusra adj a szokásosnál (miszerint x, t az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciája) „mélyebb” választ, írj Java példaprogramot mely egyszer végtelen ciklus, más x, t értékekkel meg nem! A példát építsd a JDK Integer.java forrására3 , hogy a 128-nál inkluzív objektum példányokat poolozza!

Első eset: nem végtelen ciklus, azaz a while ciklus feltétele hamis.

```
Integer x = -128;  
Integer t = -128;
```

Ha két Integer osztálybeli változónak szeretnénk ugyan azt a számot adni értékül, akkor figyelnünk kell arra, hogy ha ez a szám -128 és 127 között van, akkor csak 1 objektum jön létre. Ezeket a számokat a Java nyelv cacheli. Mindkét változó referenciaja ugyan erre az objektumra fog mutatni. Ezt azért használja a Java nyelv, mert a -128 és 127 közötti számok használata elég gyakori. Emiatt ebben a példában a while ciklus feltételének azon része, hogy

```
x != t
```

hamis lesz.

Második eset: végtelen ciklus, azaz a while ciklus feltétele igaz.

```
Integer x = -129;  
Integer t = -129;
```

A két változó értéke nincs a -128 és 127 közötti tartományban, emiatt érték adáskor minden két változónak egy külön Integer osztálybeli objektum jön létre. Így a feltétel, hogy

```
x != t
```

igaz lesz.

Az alábbi módon tudunk létrehozni két külön objektumot, a tartományban lévő számokkal:

```
Integer x = new Integer(-128);  
Integer t = -128;
```

Ebben a példában a számokat a tartományból vettük, viszont mégis végtelen ciklus lesz, mivel két külön objektumként hoztuk őket létre. Ezt a jelenséget hívják autoboxingnak. Autoboxing: Egy primitív értéket (int, char, double, ... stb.) konvertál át, a hozzá tartozó csomagoló osztálybeli (Integer, Character, Double, ... stb.) objektummá.

12.3. Yoda

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

A Yoda Conditions egy programozási stílus, amit hibák elkerülésére használhatunk. A nevét az egyik Star Wars karakterről, Yodáról kapta, a különleges beszédstílusa miatt.

Gyakorlatban annyit jelent, hogy felcseréljük a változót és a konstanst / értéket. Pl.: if(color == red) helyett if(red == color) Ennek a programozási stílusnak az előnye, hogy ha véletlen "==" helyett "="-et írunk, akkor a fordító jelezni fog, hogy hibás, mert az, hogy "red = color" nem értelmezhető. Viszont, ha ez fordítva történik, akkor bajban vagyunk, mivel a "color == red" is értelmezhető és a "color = red" is. Így ha hibázunk, akkor is le fog fordulni a program, viszont nem úgy fog működni, ahogyan mi szeretnénk. Hiszen egy darab egyenlőség jel értékadást jelent, és annak, hogy "color = red" van értelme így is. Ha egy if elágazásban történik ez a hiba, akkor az értékadás megtörténik, az if "true" lesz, emiatt lefut az if-ben lévő kódblokk is. Tehát a Yoda Conditionst akkor érdemes használni, ha attól tartunk, hogy az összehasonlítás helyett véletlen értékadást írunk, és így a programunk hibásan fut le.

12.4. Kódolás from scratch

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

Link: [PiBBP.java](#)

A Bailey-Borwein-Plouffe formulát 1995-ben találta ki Simon Plouffe. A nevét annak a cikknek a szerzőiről kapta, amelyben megjelent ez a formula. A formula: A formulával a pi-nek az n-edik számjegyét lehet kiszámolni hexadecimálisan. Ezelőtt azt hitték az emberek, hogy a pi-nek az első n darab számjegyét ugyan olyan nehéz kiszámolni, mint a pi n-edik számjegyét. Ezzel a formulával viszont pár másodperc alatt kitudja számolni a gép. A BBP formulával létrejött egy "Spigot" algoritmus, amellyel úgy tudjuk kiszámolni a pi n-edik számjegyét hexadecimálisan, hogy az előtte lévő számjegyeket nem kell kiszámoljuk. A "Spigot" algoritmusok olyan algoritmusok, amelyek segítségével matematikai konstansok értékét számolhatjuk ki, mint például a pi, e.

A forráskódunkat fordítva és futtatva az alábbi kimenetet kapjuk: javac PiBBP.java java PiBBP 6C65E5308 Ha megnézzük a main függvényünk, akkor láthatjuk, hogy csak egy kiírás történik. Mégpedig a pi 1000001-edik számjegyét írjuk ki. public static void main(String args[]) { System.out.print(new PiBBP(1000000)); } A kapott érték természetesen hexadecimális. A programunkban ezt az értéket megváltoztatva számolhatjuk ki a pi bármely számjegyét. for (int i=0; i<100; i+=1) { PiBBP piBBP = new PiBBP(i); System.out.println(piBBP.toString().charAt(0)); } Ezzel az apró változtatással már a pi első 100 számjegyét kapjuk meg hexadecimálisan: 243F6A8885A308D313198A2E03707344A4093822299F31D0082EFA98EC4E60

13. fejezet

Helló, Liskov!

13.1. Liskov helyettesítés sértése

Írunk olyan OO, leforduló Java és C++ kódcsipetet, amely megséríti 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 Java forrása: [Java](#)

Megoldás C++ forrása: [C++](#)

A Liskov elv a S.O.L.I.D. - Objektum orientált tervezési elvek egyik része. A S.O.L.I.D.-ból az "L" betű azt jelenti, hogy LSP. Az LSP pedig a Liskov Substitution Principle rövidítése.

A Liskov elv szerint, ha T-nek altípusa S, akkor mindenhol ahol T-t felhasználtuk, ott S-t is felhasználhatjuk. Ez nem fog semmilyen változást okozni a programunk viselkedésében.

A Liskov elv megsértésére általában két féle példát szoktak felhözni: a téglalap és kocka esete és az ellipszis és kör esete. Az alábbi példában a téglalap és kocka esetét nézzük meg. Készítünk egy Rectangle (téglalap) osztályt. A téglalap osztályban két protected láthatóságú int változó van: a magasság és hossz.

```
protected int m_width;
protected int m_height;
```

Az, hogy "protected" azt jelenti, hogy az adott osztályon és annak alosztályain kívül semmi más nem fér hozzá ezekhez a változókhöz. Ezen kívül még két féle láthatóság létezik: public és private. A private esetében csak az adott osztályon belül férnek hozzá az elemhez, public esetében pedig bárki hozzáférhet.

Ennek a téglalap osztálynak van két metódusa: setWidth és setHeight. Ezekkel a metódusokkal adhatjuk meg a téglalap hosszát és magasságát.

```
public void setWidth(int width) {
    m_width = width;
}
```

```
public void setHeight(int height) {
    m_height = height;
}
```

Továbbá készítünk egy Square (kocka) osztályt is, ami a téglalap osztály alosztálya lesz. Ez azt jelenti, hogy minden ami a téglalap osztályban van, azt használhatja a kocka osztály is. A kockák viszont speciális téglalapok, azaz minden oldaluk egyenlő hosszúságú. Emiatt felülírjuk a setWidth és setHeight metódusokat ebben az osztályban.

```
class Square extends Rectangle{
    public void setWidth(int width) {
        m_width = width;
        m_height = width;
    }

    public void setHeight(int height) {
        m_width = height;
        m_height = height;
    }
}
```

A main függvényben készítünk egy téglalap típusú r nevű kocka objektumot. Ennek az objektumnak adunk meg hosszt és magasságot, utána ki iratjuk a területét. A terület viszont nem lesz helyes.

```
private static Rectangle getNewRectangle() {
    return new Square();
}

public static void main(String[] args) {
    Rectangle r = LspTest.getNewRectangle();
    r.setWidth(5);
    r.setHeight(10);
    System.out.println(r.getArea());
}
```

```
ujhazi@ubuntu:~$ javac LspTest.java
ujhazi@ubuntu:~$ java LspTest
100
ujhazi@ubuntu:~$ █
```

13.2. Szülő-gyerek

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

Megoldás Java forrása: [Java](#)

Megoldás C++ forrása: [C++](#)

A Java programunkban készítünk egy szülő osztályt. Ennek az alosztályaként jön létre a gyerek osztály. Ez a gyerek osztály tartalmazza a getName metódust.

```
public String getName() {
    return m_name;
}
```

A program main függvényében egy szülő típusú s nevű objektumot, és egy gyerek típusú gy nevű gyerek objektumot hozunk létre. Ezeknek adunk nevet és kort.

```
szulo s = new gyerek();
s.setName("Szülő");
s.setAge(60);

gyerek gy = new gyerek();
gy.setName("Gyerek");
gy.setAge(15);
```

Ez után megpróbáljuk minden objektum nevét kiiratni.

```
System.out.println(gy.getName() + " " + s.getName());
```

Ez fordításkor hibát fog okozni, mert a szülő típusú objektum nem fér hozzá a gyerek osztály metódusaihoz.

A Java program hiba üzenete fordításkor:

```
ujhazi@ubuntu:~$ javac szuloGyerek.java
szuloGyerek.java:36: error: cannot find symbol
        System.out.println(gy.getName() + " " + s.getName());
                           ^
      symbol:   method getName()
      location: variable s of type szulo
1 error
ujhazi@ubuntu:~$
```

A C++ program hiba üzenete fordításkor:

```
ujhazi@ubuntu:~$ gcc szuloGyerek.cpp -o szuloGyerek
szuloGyerek.cpp: In function ‘int main()’:
szuloGyerek.cpp:34:34: error: ‘class Rectangle’ has no member named ‘getArea’
    std::cout << s->getArea() << r->getArea() << std::endl;;
                                         ^
ujhazi@ubuntu:~$ 
```

13.3. Anti OO

A BBP algoritmussal a Pi hexadecimális kifejtésének a 0. pozíciótól számított 106, 107, 108 darab jegyét határozzuk meg C, C++, Java és C# nyelveken és vessük össze a futási időket! <https://www.tankonyvtar.hu/hu/tartanitok-javat/apas03.html#id561066>

Megoldás Java forrása: [Java](#)

Megoldás C forrása: [C](#)

Megoldás C# forrása: [C#](#)

A forráskódokat egy laptopon, Windows 10-re telepített VMware-ben lévő virtuális 64 bites Ubunturól futtattam le. Az Ubuntu verziója: Ubuntu 18.04.3 LTS. Ezt az lsb_release parancssal és a -a kapcsolóval tekinthetjük meg:

```
lsb_release -a
```

A parancs futtatása után ezt láthatjuk:

```
ujhazi@ubuntu:~/Desktop/prog2$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.3 LTS
Release:        18.04
Codename:       bionic
ujhazi@ubuntu:~/Desktop/prog2$ 
```

A laptopban lévő processzor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz. Ezt az információt a list hardware (lshw) parancssal deríthetjük ki:

```
sudo lshw -html > mySpecs.html
```

Ez készít egy mySpecs névű html fájlt, amit ha egy tetszőleges böngészővel megnyitunk, akkor minden fontos hardware-el kapcsolatos információt megtudhatunk.

A virtuális gép 4 Gb ramot használ. Ezt a proc/meminfoból olvashatjuk ki:

```
cat /proc/meminfo
```

```
ujhazi@ubuntu:~/Desktop/prog2$ cat /proc/meminfo
MemTotal:      4015780 kB
MemFree:       1219460 kB
MemAvailable:  2115620 kB
Buffers:        149020 kB
Cached:         1029152 kB
SwapCached:     0 kB
Active:         1559096 kB
Inactive:      1077521 kB
```

Mostmár minden tudunk a virtuális gépről. Nézzük meg, hogy mennyi idő alatt futnak le a forráskódok.

Fordítás és futtatás után kiderült, hogy a C forráskód 1.732643 másodperc alatt futott le.

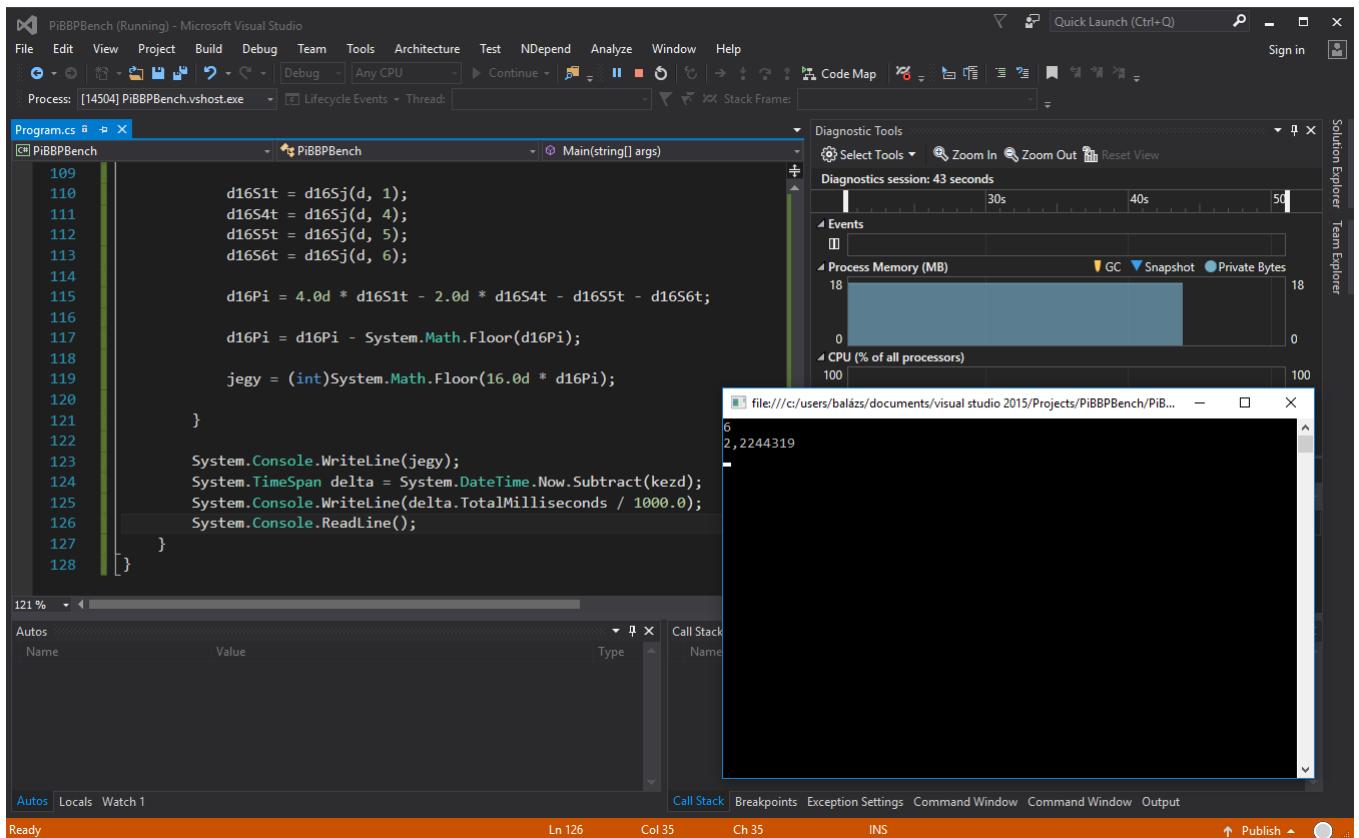
```
ujhazi@ubuntu:~/Desktop/prog2$ g++ pi_bb_p_bench.c  
ujhazi@ubuntu:~/Desktop/prog2$ ./pi_bb_p_bench.out  
6  
1.732643  
ujhazi@ubuntu:~/Desktop/prog2$ 
```

A Java forráskód pedig 1.583 másodperc alatt, tehát gyorsabban, mint a C forráskód.

```
ujhazi@ubuntu:~/Desktop/prog2$ java PiBBPBench  
6  
1.583  
ujhazi@ubuntu:~/Desktop/prog2$ █
```

Linux egy disztribúcióján való C# forráskód fordításához, illetve futtatásához használhatunk Mono-t, vagy .NET Core-t. Én az előbböt próbáltam meg használni, de problémába ütköztem. Követtem az utasításokat ezen a weboldalon: https://www_mono-project_com/download/stable/#download-lin, visszont sajnos nem működik.

Windows 10 alatt Visual Studioban a C# kód futtatása 2,2244319 másodperc volt. Viszont ez nem hiteles, mert a gazdagépnek és a virtuális gépnek nem egyezik meg a hardware része.



13.4. 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 forrása:

A Ciklomatikus komplexitást Thomas J. McCabe alkotta meg. Emiatt sokszor McCabe-komplexitásnak is hívják. A Ciklomatikus komplexitás lényege, hogy forráskód alapján megmondja annak komplexitását egy konkrét számmal. A számítás a gráfelméletre alapul, az elágazásokból felépülő gráf pontjai, és a köztük lévő élek alapján számítható a komplexitás.

A komplexitás értéke: 9

$$M = E - N + 2P$$

ahol E: a gráf éleinek száma, N: a gráfban lévő csúcsok száma és P: az összefüggő komponensek száma.

A PiBBPBench C# nyelven írt programnak néztem meg a komplexitását. A Visual Studioban ez egy beépített funkció. A Visual Studioban belül a fenti menüből kiválasztjuk az Analyze menüpontot. Ezen belül a Calculate Code Metrics-et választjuk és végül a For Solution-ra kattintunk. Az eredmény a program alján látható Code Metrics Results néven. Itt láthatjuk, hogy a PiBBPBench Ciklomatikus komplexitása 9.

PiBBPBench - Microsoft Visual Studio

File Edit View Project Build Debug Team Tools Architecture Test NDepend Analyze Window Help

Program.cs

```
public static void Main(System.String[] args)
{
    double d16Pi = 0.0d;

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

    int jegy = 0;

    System.DateTime kezd = System.DateTime.Now;

    for (int d = 1000000; d < 1000001; ++d)
    {
        d16Pi = 0.0d;

        d16S1t = d16Sj(d, 1);
        d16S4t = d16Sj(d, 4);
        d16S5t = d16Sj(d, 5);
    }
}
```

Solution Explorer

- Solution 'PiBBPBench' (1 project)
 - Properties
 - References
 - App.config
 - Program.cs

Code Metrics Results

Hierarchy	Maintainability Index	Cyclomatic Comple...	Depth of Inheritance	Class Coupling	Lines of Code
PIBBPBench (Debug)	62	9	1	4	39

Code Metrics Results | Error List... | Output | Task Runner Explorer

14. fejezet

Helló, Mandelbrot!

14.1. Reverse engineering UML osztálydiagram

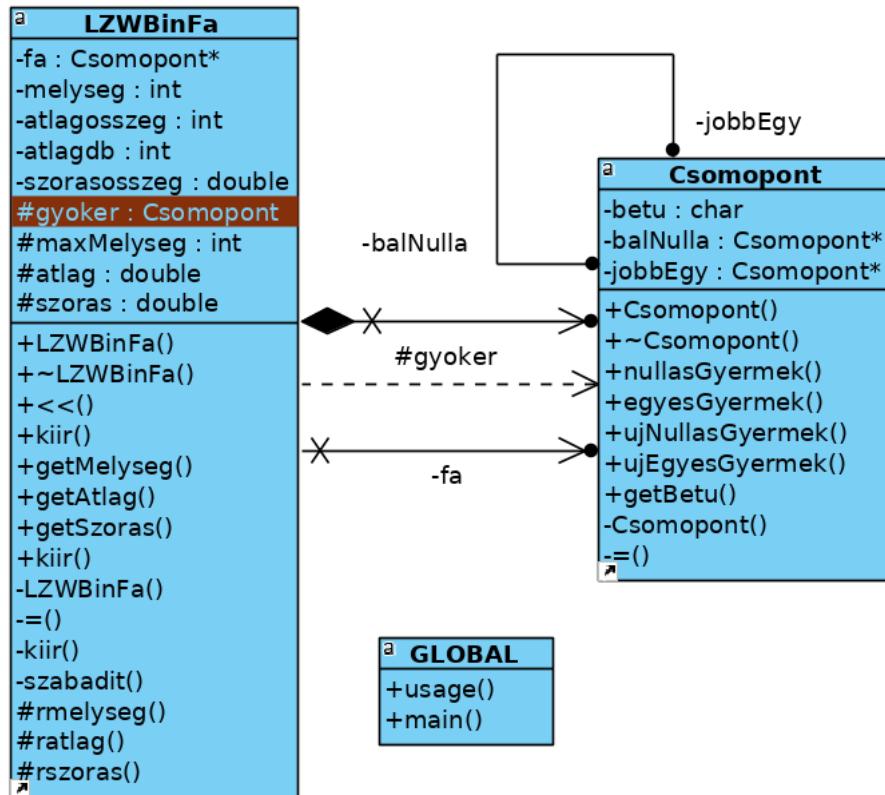
UML osztálydiagram rajzolása az első védési C++ programhoz. Az osztálydiagramot a forrásokból generáljuk (pl. Argo UML, Umbrello, Eclipse UML) Mutassunk rá a kompozíció és aggregáció kapcsolatára a forráskódban és a diagramon, lásd még: https://youtu.be/Td_nIERIEOs. <https://arato.inf.unideb.hu/batfai.norbert/UD/>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

Megoldás forrása: [Binfa](#)

Az első védési program a binfa volt. A forráskódját megtaláljuk az UDPORG Sourceforge repoban, vagy a fenti linken.

Az osztálydiagram generálásához (reverse engineering) a Visual Paradigm programot használtam. A program letölthető Windowsra és Linuxra is az oldalukról. Telepítés után csak annyi dolgunk van, hogy kiválasztjuk a fenti menüpontok közül, a Tools-t. A Tools-on belül pedig a Code-ra kattintunk, majd Instant Reverse.

A megnyíló kis ablak tetején beállítjuk a nyelvet "C++ / C# Source"-ra. Az elérési út kiválasztásánál a fájl típusát átállítjuk C++ Header File-ról C++ Source File-ra, majd kiválasztjuk a z3a7.cpp-t. Végül az OK gombra kattintva meg is történik a forráskódból való osztálydiagram generálás (reverse engineering).



Aggregáció: részkapcsolat, egy részosztály több befogadó osztályhoz tartozhat, van értelmük külön.

Kompozíció: részosztály, csak az egész osztállyal együtt értelmezhető. Ha törlésre kerül, az egész törlődik, mert önmagukban nincs értelmük.

14.2. Forward engineering UML osztálydiagram

UML-ben tervezünk osztályokat és generálunk belőle forrást!

Megoldás forrása: [Bifna](#)

Az első feladatban elkészült osztálydiagramból generáltam kódot. A program, amit használtam megint a Visual Paradigm.

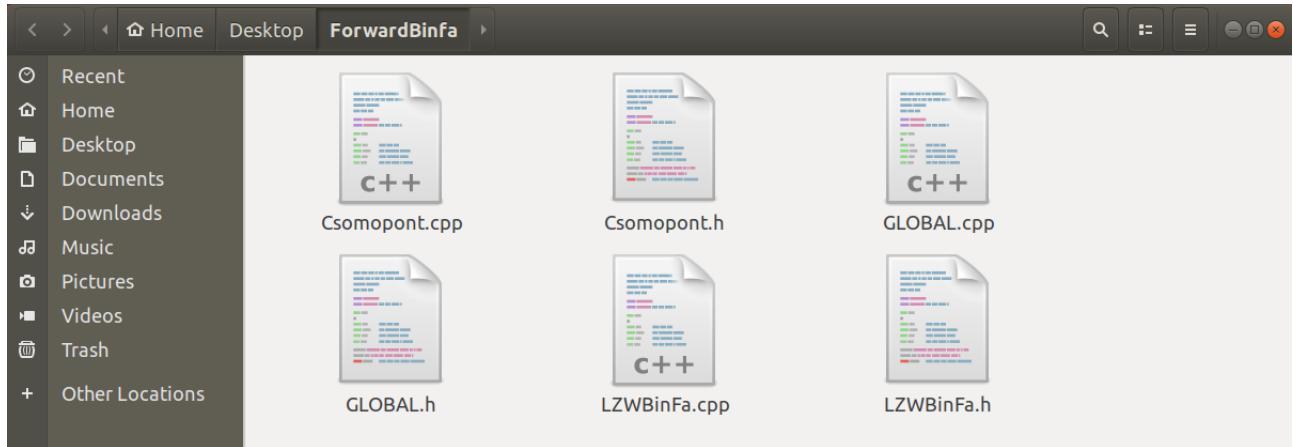
A programban a felső menüpontból kiválasztjuk a Tools-t. Itt a Code-ra kattintunk és ott kiválasztjuk azt, hogy Instant Generator. A megnyíló ablak tetején a nyelvet beállítjuk C++-ra és kijelöljük a diagramok közül azt, amelyikből kódot szeretnénk generálni. Az output path jelöli azt az elérési utat, ahova kerülnek a generált kódok. Ajánlott egy üres mappát készíteni és ezt beállítani. A template directory maradjon az alapértelmezett elérési út, az én esetben ez a következő:

```
/home/ujhazi/Visual_Paradigm_16.0/resources/ ←
instantgenerator/cplusplus
```

A Preview gombra kattintva lehet generálás előtt megnézni, hogy minden rendben van-e. Itt láthatjuk a fájlokat amiket létre fog hozni és ha rákattintunk egy-egy fájlra, akkor a tartalmát is megnézhetjük. Ha

mindent leellenőriztünk, akkor mehet a generálás a Generate gombbal. Az Open Output Folder gombbal pedig megnyithatjuk azt a mappát, ahova generálódni fognak a kódok.

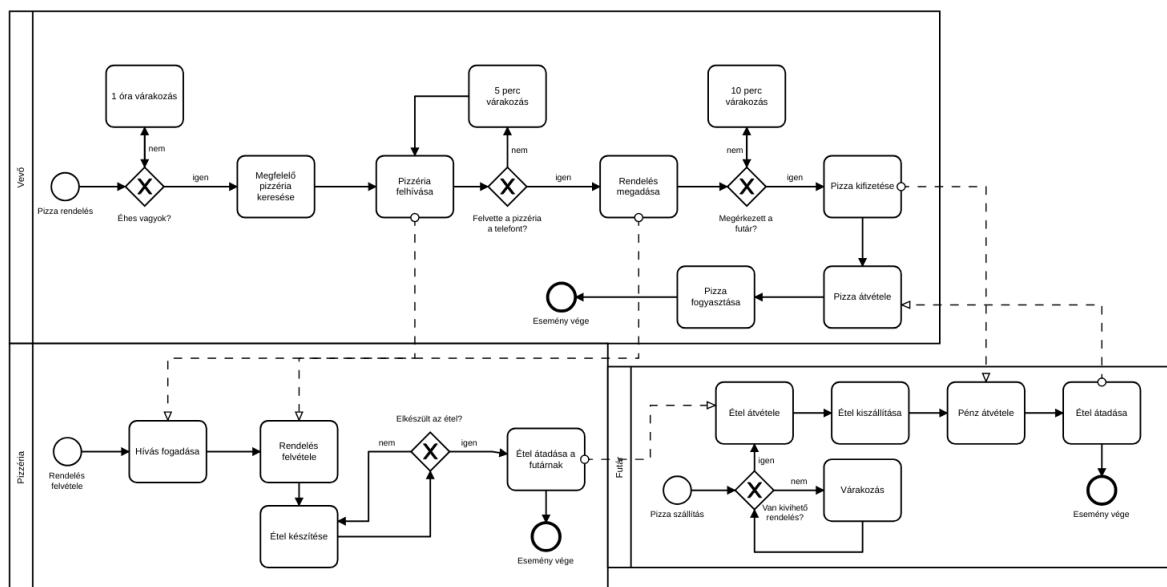
Generálás után így néz ki a mappa tartalma:



14.3. BPMN

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

A Camunda Modeler programot használtam a BPMN modell készítéséhez. A tevékenység, amit ábrázoltam: pizza rendelés.



A képen látható, hogy 3 esemény van összesen. Az első a pizza rendelésének eseménye, azaz a vevőé. Addig várunk ameddig nem leszünk éhesek, ha éhesek lettünk, akkor keresünk egy megfelelő pizzériát és felhívjuk. Addig próbálkozunk a hívással ameddig nem lesz sikeres. Ha felveszik a telefont, akkor megrendeljük az ételt, majd várunk arra, hogy megérkezzen. Ha megérkezik, akkor kifizetjük és átvesszük, végül elfogyasztjuk.

A második esemény a pizzéria szemszögéből történik. A rendelés felvétele úgy történik, hogy ha van bejövő hívás, akkor azt fogadja, majd felveszi a rendelést. Ez után elkészíti a kiszállítandó ételt, és ezt átadja a futárnak.

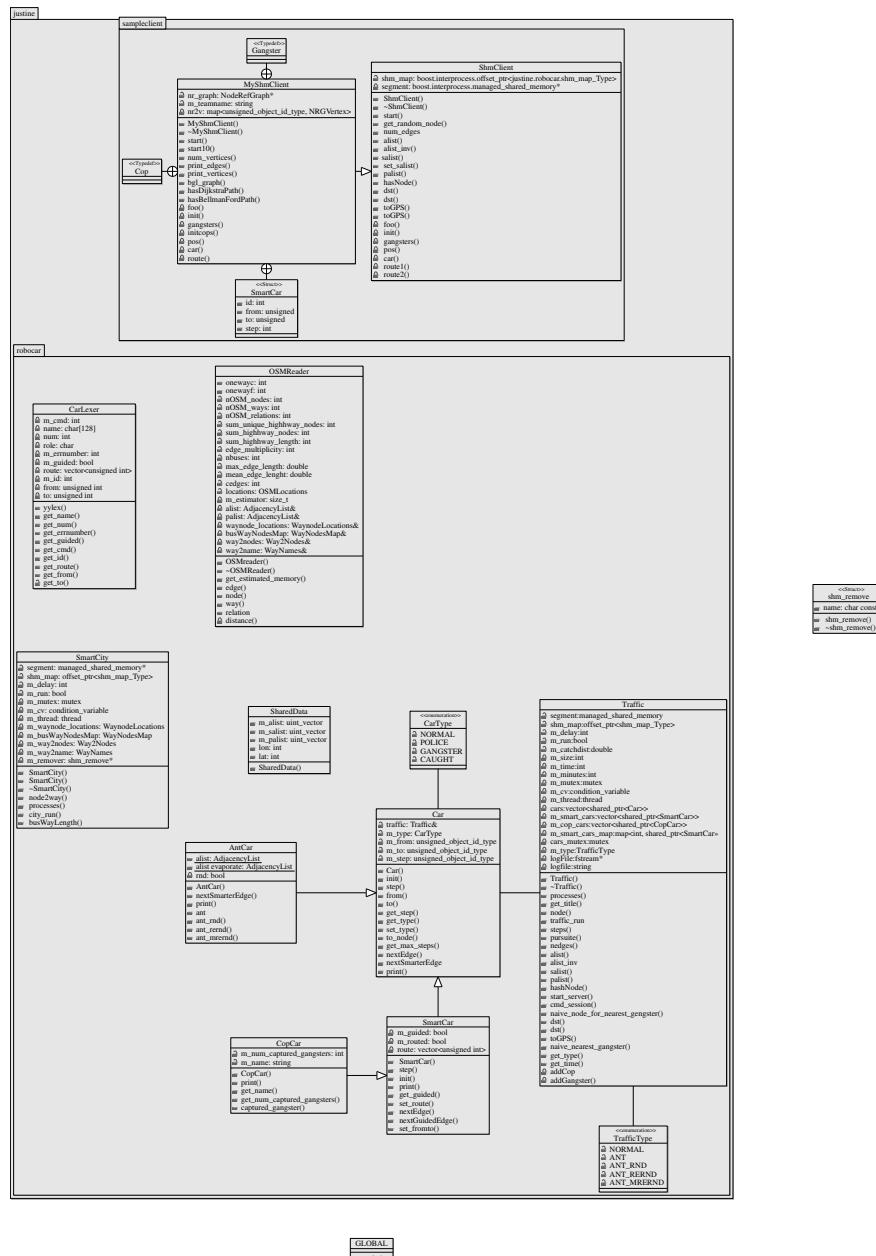
A harmadik és egyben utolsó esemény a futár szemszöge. Itt történik a pizza kiszállítása. Addig várunk, ameddig nem lesz kiszállítandó étel. Ha van étel, amit ki kell szállítani, akkor átvesszük és kiszállítjuk. Végül átvesszük a pénzt és átadjuk az ételt.

14.4. TeX UML

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

Megoldás forrása: [OOCWC](#) Megoldás forrása: [MetaUML](#)

Az OOCWC (Robocar World Championship) projektről a MetaUML segítségével készítettem osztálydiagrammot.



A MetaUML-ben írt fájlok kiterjesztése a .mp. Ahhoz, hogy ebből a forrásból diagram legyen, át kell fordítanunk pdf-re. Az "mptopdf" parancs fog nekünk ebben segíteni. Használata jelen esetben:

```
ujhazi@ubuntu:~/Downloads$ mptopdf oocwmp.mp
MPtoPDF 1.4.1 : running 'mpost --mem=mpost oocwmp.mp'
This is MetaPost, version 2.000 (TeX Live 2017/Debian) (kpathsea version 6.2.3)
(/usr/share/texlive/texmf-dist/metapost/base/mpost.mp
(/usr/share/texlive/texmf-dist/metapost/base/plain.mp
Preloading the plain mem file, version 1.005) ) (.oocwmp.mp
(/usr/share/texlive/texmf-dist/metapost/metauml/metauml.mp
(/usr/share/texlive/texmf-dist/metapost/metauml/_util_infrastructure.mp
(/usr/share/texlive/texmf-dist/metapost/metauml/_util_log.mp))
(/usr/share/texlive/texmf-dist/metapost/metauml/_util_object.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_util_commons.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_util_margins.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_util_picture.mp)
(/usr/share/texlive/texmf-dist/metapost/base/boxes.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_util_group.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_util_picture_stack.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_util_positioning.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_util_shade.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_metauml_base.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_metauml_links.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_metauml_paths.mp)
(/usr/share/texlive/texmf-dist/metapost/metauml/_metauml_note.mp)
```

Egy kis idő után a pdf-ünk el is készül, ez látható a fenti képen.

15. fejezet

Helló, Chomsky!

15.1. Encoding

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

Megoldás forrása: [Encoding](#)

Egy bizonyos fájlról kideríthetjük, hogy milyen karakterkészletet használ. Erre a célra használhatjuk például a Sublime Text programot. Megnyitjuk a fájlt Sublime Text-ben:

```
ujhazi@ubuntu:~/Downloads/nehany_egyeb_pelda$ sublime-text.subl MandelbrotHalmazNagyító.java
ujhazi@ubuntu:~/Downloads/nehany_egyeb_pelda$
```

Majd kiválasztjuk a "View"-t a fenti menüpontból, és ott megkeressük a "Show Console" opciót. A konzol megnyitásához gyorsbillentyű kombinációt is használhatunk, ez alapértelmezetten a "Ctrl + `". A konzol, a "view.encoding()" parancs megadása után ki is írja, hogy milyen karakterkészletet használ a fájl.

```
>>> view.encoding()
'Western (Windows 1252)'
```

A forrásunk fordításkor az alábbi hibát dobja ki:

```
ujhazi@ubuntu:~/Downloads$ javac MandelbrothHalmazNagyító.java
MandelbrothHalmazNagyító.java:2: error: unmappable character for encoding UTF8
 * MandelbrothHalmazNagyító.java
   ^
MandelbrothHalmazNagyító.java:2: error: unmappable character for encoding UTF8
 * MandelbrothHalmazNagyító.java
   ^
MandelbrothHalmazNagyító.java:4: error: unmappable character for encoding UTF8
 * DIGIT 2005, Javat tanított
   ^
MandelbrothHalmazNagyító.java:5: error: unmappable character for encoding UTF8
 * Bétfai Norbert, nbatfai@inf.unideb.hu
   ^
MandelbrothHalmazNagyító.java:9: error: unmappable character for encoding UTF8
 * A Mandelbrot halmazt nagyítás kirajzolás osztály.
   ^
MandelbrothHalmazNagyító.java:9: error: unmappable character for encoding UTF8
 * A Mandelbrot halmazt nagyítás kirajzolás osztály.
   ^
MandelbrothHalmazNagyító.java:9: error: unmappable character for encoding UTF8
 * A Mandelbrot halmazt nagyítás kirajzolás osztály.
   ^
MandelbrothHalmazNagyító.java:9: error: unmappable character for encoding UTF8
```

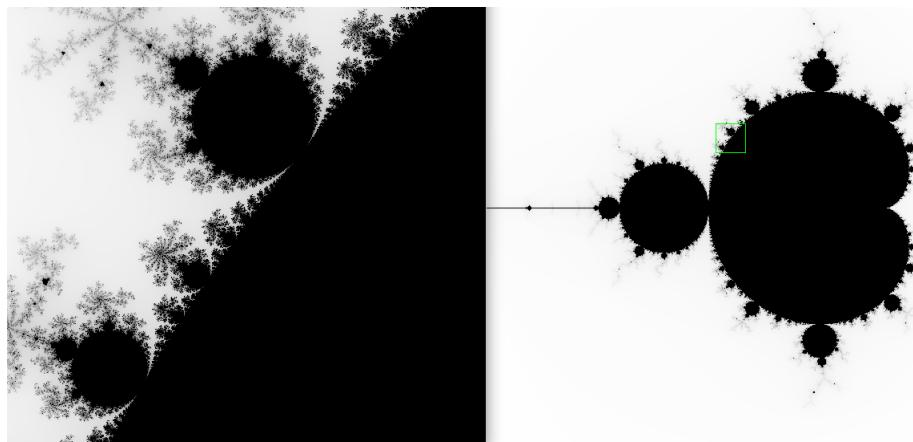
Miután kiderítettük ezt a fontos információt, mostmár le tudjuk fordítani a forrásunkat. A javac parancsot kell használnunk a -encoding kapcsolóval.

```
javac -encoding windows-1252 MandelbrotIterációk.java
javac -encoding windows-1252 MandelbrotHalmaz.java
javac -encoding windows-1252 MandelbrothHalmazNagyító.java
```

A windows-1252 karakterkészlet helyett fordíthatjuk iso-8859-1 karakterkészlettel, vagy iso-8859-15-el.

Végül a lefordított forrásunkat futtatjuk:

```
java MandelbrothHalmazNagyító
```



15.2. OOCWC lexer

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

Megoldás forrása:

Ha megnyitjuk a carlexer.ll file-t, akkor találhatunk néhány reguláris kifejezést:

```
INIT      "<init"
INITG     "<init guided"
WS        [ \t ]*
WORD      [^-:\n \t () ]{2, }
INT       [0123456789]+
FLOAT     [-.0123456789]+
ROUTE     "<route"
CAR       "<car"
POS       "<pos"
GANGSTERS  "<gangsters"
STAT      "<stat"
DISP      "<disp">
```

A file-t tovább nézve, a reguláris kifejezések alatt közvetlenül vannak a fenti kifejezésekhez tartozó kódok. Ha a bemeneti stringben valamelyik kifejezés előfordul, akkor az ahoz tartozó kód fog lefutni.

Erre egy példa:

```
{CAR} {WS} {INT}
{
    std::sscanf(yytext, "<car %d", &m_id);
    m_cmd = 1001;
}
```

Legtöbb esetben a sscanf függvényt használjuk, ez a filekezelő függvények egyike. A sscanf függvény bemenetet olvas egy stringből.

Szintaxisa az alábbi:

```
int sscanf(const char *str, const char *format, ...)
```

Az első paraméterként megadott stringből fog olvasni és a formatban pedig egy, vagy több whitespace karaktert (\b, \t, \n, \w, \f) adhatunk meg, illetve egyéb karaktereket is, például: %.

Az int miatt tudjuk, hogy a függvény visszatérési értéke egy szám lesz. Ez a szám a sikeresen beolvasott értékek darabszáma.

A lexert a TCP porton kapott információ feldolgozására használjuk. A lexer bemenetként a TCP socket-ben lévő stringet kapja meg.

Az OOCWC projekt futtatása nem sikerült, az alábbi hibába ütköztem az autoconf configure.ac parancs futtatásakor:

```
ujhazi@ubuntu: ~/Downloads/robocar-emulator-master/justine/rcemu
ujhazi@ubuntu: ~/Downloads/robocar-emulator-master/justine/rcemu 80x24
# Unfortunately, on DOS this fails, as config.log is still kept open
# by configure, so config.status won't be able to write to it; its
# output is simply discarded. So we exec the FD to /dev/null,
# effectively closing config.log, so it can be properly (re)opened and
# appended to by config.status. When coming back to configure, we
# need to make the FD available again.
if test "$no_create" != yes; then
    ac_cs_success=:
    ac_config_status_args=
    test "$silent" = yes &&
        ac_config_status_args="$ac_config_status_args --quiet"
    exec 5>/dev/null
    $SHELL $CONFIG_STATUS $ac_config_status_args || ac_cs_success=false
    exec 5>>config.log
    # Use ||, not &&, to avoid exiting from the if with $? = 1, which
    # would make configure fail if this is the last instruction.
    $ac_cs_success || as_fn_exit 1
fi
if test -n "$ac_unrecognized_opts" && test "$enable_option_checking" != no; then
    { $as_echo "$as_me:${as_lineno-$LINENO}: WARNING: unrecognized options: $ac_unrecognized_opts" >&5
$as_echo "$as_me: WARNING: unrecognized options: $ac_unrecognized_opts" >&2; }
fi
ujhazi@ubuntu:~/Downloads/robocar-emulator-master/justine/rcemu$
```

15.3. I334d1c4

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

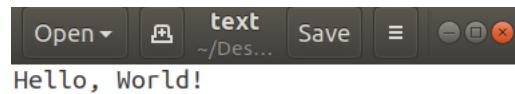
Megoldás forrása: [LeetCypher](#)

Készítünk egy leet_alphabet nevű vektor tömböt és ezt feltöljük a leet ábécé szimbólumaival. Egy másik vektort feltöltünk a kicsérélendő betűkkel. Ez minden konstruktorban történik meg. Az egész bemeneti szöveget nagybetűsre alakítjuk, majd karakterenként nézzük. Ha a karakter szerepel a kicsérélendő betűk között, akkor a karakternek megfelelő leet_alphabetben lévő szimbólumot hozzáfűzzük egy stringhez. Ha egy adott betűhöz több szimbólum is tartozik, akkor ezek közül véletlenszerűen választunk egyet. Ha a vizsgált karakter nincs a kicsérélendő betűk között, akkor csak egyszerűen hozzáfűzzük az eddig stringhez. A végén pedig az eredményül kapott szöveget egy kimeneti fájlba írjuk.

A forrás fordítása és futtatása:

```
ujhazi@ubuntu:~/Desktop$ g++ LeetCypher.cpp -o LeetCypher.out
ujhazi@ubuntu:~/Desktop$ ./LeetCypher.out text outputtext
ujhazi@ubuntu:~/Desktop$
```

A bemeneti szöveg:



A végeredményül kapott szöveg:



15.4. Full screen

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

Megoldás forrása: [BouncingBall](#)

A labda mozgatása if-ek nélkül történik. A keretet undecorateddé tesszük a konstruktorban, így nem lesz a felső menüsor megjelenítve. Ezáltal igazi full screen hatást kelt a program. A setFullScreenWindow() függvényt használva full screen módba vált a programunk. Ez a függvény egy frame, vagy egy window típusú objektumot vár paraméterként. Ez után kiválasztjuk a legjobb megjelenítési módot. Végül kirajzoljuk a háttérét és a labdát.

A forrás fordítása és futtatása:

```
ujhazi@ubuntu:~/Desktop$ javac BouncingBall.java
ujhazi@ubuntu:~/Desktop$ java BouncingBall
```

Kép a teljes képernyős labdapattogásról:



16. fejezet

Helló, Stroustrup!

16.1. JDK osztályok

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

Megoldás forrása: [jdkosztalyok](#)

Ebben a feladatban a fénykardnak azt a részét írtuk át, amely a .props fájlokat kereste. Egész egyszerűen a kiterjesztést átírhatjuk .java-ra.

```
string GetCurrentWorkingDir( void ) {
    char buff[FILENAME_MAX];
    GetCurrentDir( buff, FILENAME_MAX );
    string current_working_dir(buff);
    return current_working_dir;
}

vector<string> roots = {GetCurrentWorkingDir() + "/" + "src"};
```

A GetCurrentWorkingDir lekérdezi a jelenlegi helyzetünket a getcwd-vel és visszaadja azt stringben. Tehát a fenti sor azt csinálja, hogy a gyökérkönyvtárunk a jelenlegi helyzetünkben lévő src mappa lesz.

```
vector<string> searchRootFolders (vector<string>folders) {
vector <string> classes;
for ( const auto & path : folders) {
    boost::filesystem::path root ( path );
    readClasses ( root, classes);
}
return classes;
}
```

A searchRootFolders függvény a gyökérkönyvtárban lévő mappákat nézi végig, és minden mappára meg-hívja a readClassses függvényt.

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

A readClasses függvény megnézi a paraméterül kapott fájlok kiterjesztését. Ha a kiterjesztés .java, akkor hozzáadja a classes nevű vektorhoz. Ha a paraméter nem egy fájl, hanem egy mappa, akkor meghívja a mappára a readClasses függvényt, azaz saját magát. Így rekurzívan végigmegyünk minden fájlon és megkapjuk a megfelelő végeredményt.

Az src.zip fájlt letölthetjük az alábbi linken: <https://download.java.net/openjdk/jdk8/>

Kép a program futásáról:

```
ujhazi@ubuntu:~/Downloads$ g++ jdkosztalyok.cpp -o jdkosztalyok.out -lboost_system
/tmp/cc6Bw28y.o: In function `boost::filesystem::is_directory(boost::filesystem::path const&)':
jdkosztalyok.cpp:(.text._ZN5boost10filesystem12is_directoryERKNS0_4pathE[_ZN5boost10filesystem12is_directoryERKNS0_4pathE]+0x2f): undefined reference to `boost::filesystem::detail::status(boost::filesystem::path const&, boost::system::error_code*)'
/tmp/cc6Bw28y.o: In function `boost::filesystem::is_regular_file(boost::filesystem::path const&)':
jdkosztalyok.cpp:(.text._ZN5boost10filesystem15is_regular_fileERKNS0_4pathE[_ZN5boost10filesystem15is_regular_fileERKNS0_4pathE]+0x2f): undefined reference to `boost::filesystem::detail::status(boost::filesystem::path const&, boost::system::error_code*)'
/tmp/cc6Bw28y.o: In function `boost::filesystem::detail::dir_itr_imp::~dir_itr_i
mp()':
jdkosztalyok.cpp:(.text._ZN5boost10filesystem6detail11dir_itr_impD2Ev[_ZN5boost10filesystem6detail11dir_itr_impD2Ev]+0x24): undefined reference to `boost::filesystem::detail::dir_itr_close(void*&, void*&)'
/tmp/cc6Bw28y.o: In function `boost::filesystem::directory_iterator::directory_iterator(boost::filesystem::path const&)':
jdkosztalyok.cpp:(.text._ZN5boost10filesystem18directory_iteratorC2ERKNS0_4pathE[_ZN5boost10filesystem18directory_iteratorC5ERKNS0_4pathE]+0x4b): undefined reference to `boost::filesystem::detail::directory_iterator::construct(boost::filesystem::path const&)'
```

16.2. Másoló-mozgató szemantika

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

Másoló konstruktor és értékkopírozás:

```
    Int (const Int& n) : ertek (new int) {
        bogoc = counter++;
        *ertek = *(n.ertek);
        std::cout << "--Int masolo ctor " << bogoc << ". " << *ertek << " ←
                    "
        << this << " " << ertek << std::endl;
    }

    Int& operator= (const Int &n) {
        int *ujertek = new int();
        *ujertek = *(n.ertek);
        delete ertek;
        ertek = ujertek;
        std::cout << "-- Int masolo ertekeadas" << bogoc << ". "
        << *ertek << " " << this << " " << ertek << std::endl;
        return *this;
    }
```

A másoló konstruktur és az értékkadás fontos része az objektum orientált programozásnak. Előfordulhat, hogy egy objektumra, vagy egy objektum darabjára van szükségünk, ilyenkor jönnek jól ezek az eszközök. C++-ban ha nem készítünk magunknak egy másoló konstruktort, akkor alapértelmezetten készül nekünk egy. Ez viszont csak a nem statikus tagváltozókat másolja le, ami nem minden esetben lehet jó nekünk. Ha az osztály mutatókat, vagy referenciakat tartalmaz, akkor nem azt másolja le amire mutat, hanem magát a mutatót. Ezt a fajta másolást sekély másolásnak nevezzük.

Mozgató konstruktur és értékkadás:

```
    Int& operator= (Int && n) {
        std::swap (ertek, n.ertek);
        std::cout << "--Int mozgato ertekeadas" << bogoc << ". "
        << *ertek << " " << this << ertek << std::endl;
        return *this;
    }

    Int (Int && n) : ertek (nullptr) {
        bogoc = counter++;
        *this = std::move(n);
        std::cout << "--Int mozgato ctor mozgato ertekeadasra alapozva" ←
                    "
        << bogoc << ". " << *ertek << " " << this << " " << ertek << ←
                    "
        std::endl;
    }
```

A C++11-ben két új függvény van: a mozgató konstruktur és a mozgató értékkadás. A mozgató szemantika lényege az, hogy átadja egy objektum erőforrásait egy másik objektumnak. Ez kevesebb erőforrást igényel és gyorsabb, mint ha az egész objektumot lemosolnánk a másoló szemantikával. Miután a forrás objektum elveszíti az erőforrásait, használhatatlan lesz. De ez nem gond, mert általában az ilyen objektumot

ideiglenesek. A másoló szemantika esetén a paraméter egy konstans bal érték referencia, addig a mozgató szemantika esetén egy jobb érték referencia, ami nem konstans. A konstruktorban az ertek nevű pointerünk a paraméterként kapott objektum ertek mutatója által mutatott objektumra fog mutatni. Ezután a kapott objektum mutatóját null-ra állítjuk. Ez azért fontos, mert ez az objektum ideiglenes, és ha törlődne, akkor a destrukturörölné azokat az erőforrásokat is, amikre mutat. Tehát az is törlődne, amire az új objektumunk mutat. Mozgatás esetén az std::swap függvényt használjuk, ez felcseréli a két paraméternek az értékét.

16.3. Hibásan implementált RSA törése

Készítünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló: <https://arato.inf.unideb.hu/batfa1.n> (71-73 fólia) által készített titkos szövegen.

Megoldás forrása: [RSA](#)

Az RSA egy nyílt kulcsú, vagyis asszimetrikus titkosító algoritmus. A moduláris számelméletben és a prím-számelméletben alapszik. Az RSA titkosításnál van egy nyílt és egy titos kulcsunk. A nyílt kulcsunkat bárki ismerheti, ezt arra használják, hogy a nekünk szánt üzeneteket titkosításak ezzel a nyílt kulccsal. A titkosított üzeneteket mindenekkel csak a titkos kulccsal lehet visszafejteni. Titkos kulcs nélkül az üzenet feltörése lehetetlen, mert olyan sok időt venne igénybe.

A kulcsok készítésének a menete úgy történi, hogy választunk két különböző prím számot. Ezek lesznek a p és q. Fontos, hogy ez a két szám megfelelően nagy kell legyen, ahhoz, hogy ne lehessen könnyen feltörni a kulcsot. A két prímet összeszorozva megkapjuk n-t. Ez az n szám lesz a nyílt kulcsunk egyik része. Ennek az n-nek ki kell számoljuk az Euler-féle $\phi(n)$ -fűggvény értékét. Ezt könnyen ki tudjuk számolni, mert prím számok esetén egyszerű a képlet:

$$\phi(n) = (p-1) * (q-1)$$

Miután ezzel megvagyunk, választunk kell egy megfelelő e számot. Az e egy 1 és $\phi(n)$ közötti egész szám kell legyen. Valamint relatív príme kell legyen $\phi(n)$ -nek. Ez azt jelenti, hogy e és $\phi(n)$ legnagyobb közös osztója 1. Azért kell, hogy relatív prímek legyenek, mert ez biztosítja, hogy a d kiszámításánál használt diofantoszi egyenletnek lesz megoldása. Ezt könnyen ellenőrizhetjük az Euklideszi algoritmussal. Gyakran választják e-nek a 65537 számot, azaz $2^{16} + 1$ -et. Viszont a gyorsabb számolás érdekében a 3, 5, vagy 35 is gyakori választás. Viszont ilyen kis számoknál sokkal nagyobb a kockázat is. Ha ezzel megvagyunk, akkor sikeresen elkészítettük a nyílt kulcsunkat, ami n-ből és e-ből tevődik össze. Ezután ki kell számoljuk a d-t. Ennek a képlete a következő:

$$d * e \pmod{\phi(n)} = 1$$

azaz

$$d * e = 1 + k * \phi(n)$$

ahol k egy pozitív egész szám. A d-t a bővített Euklideszi algoritmussal (Extended Euclidean algorithm) számolhatjuk ki. Fontos, hogy d-re több jó megoldás is létezik. A d-ből és az n-ből áll a titkos kulcsunk. Ezt a d-t soha senkivel nem szabad megosztanunk, hiszen emiatt lesz a titkosított üzenet könnyen visszafejthető. p és q is nagyon fontos, hogy ne tudódjon ki, mivel velük tudjuk gyorsan kiszámolni az n-t és a d-t egy bizonyos e számmal. Az n visszafejtése elég nagy p és q esetén nagyon sok időbe kerülhet. A bináris alakban felírt n szám bitjeinek száma határozza meg a rejtelzőkód hosszúságát. Ez általában 512, 1024, 2048. Minél hosszabb, annál több idő feltörni, azaz annál biztonságosabb. De a kulcs generálása is annál több idő fog igénybe venni.

Ha egy üzenetet titkosítani szeretnénk, akkor először is meg kell kérni a címzettet, hogy küldje el a nyilvános kulcsát, ha azt még nem birtokoljuk. Ezután az M-el jelölt szöveges üzenetünket valamilyen karakterkódolással számokká kell alakítsuk (pl.: ASCII), és úgy daraboljuk fel ezt a kapott m üzenetet, hogy minden esetben m kisebb legyen, mint n. Végül a kódolt szöveget az alábbi képlettel kapjuk meg:

$$m^e \bmod n$$

Ezt a kódolt szöveget küldjük el a címzettnek és ezek után c-vel fogjuk jelölni.

A címzett ezt a kódolt szöveget vissza tudja fejteni a titkos kulcsával, azaz d-vel, az alábbi módon:

$$c^d \bmod n$$

Így megkapja az m-et, azaz az eredeti üzenetet, csak számokká alakítva. Végül a megfelelő karakterkódlással vissza tudja alakítani az m-et az eredeti üzenetre, azaz M-re.

Egy példa: Két különböző prímszám $p = 41$ és $q = 43$. $n = p * q$, azaz $n = 41 * 43$, $n = 1763$. A $\phi(n)$ -t ebből egyszerűen ki tudjuk számolni: $\phi(n) = (p - 1) * (q - 1)$, $\phi(1763) = 40 * 42$, $\phi(1763) = 1680$. Ebben az esetben $e = 11$ lesz. Ezt úgy kapjuk meg, hogy az Euklideszi algoritmussal elkezdjük kiszámolni a legnagyobb közös osztóját e-nek és $\phi(n)$ -nek, azaz 1680-nak, úgy, hogy az e 1-től indul. Ezt akár egy gcd calculatorral is megnézhetjük, ha nincs kedvünk számolgatni (pl.: www.alcula.com/calculators/math/gcd/). Addig számoljuk, ameddig egy olyan e-t kapunk, ahol a legnagyobb közös osztó 1 lesz. A nyilvános kulcsunkat így már meg is kaptuk: $e = 11$, $n = 1763$. A d kiszámítása a fent leírt módon történik, szépen visszafejtük az Euklideszi algoritmusból a Bővített Euklideszi algoritmussal a d-t:

$$d = 11^{-1} \pmod{1680}$$

$$\begin{array}{rcl} 1680 = 11 \cdot 152 + 8 & | & 1680 - 11 \cdot 152 = 8 \\ 11 = 8 \cdot 1 + 3 & & 11 - 8 \cdot 1 = 3 \checkmark \\ 8 = 3 \cdot 2 + 2 & & 8 - 3 \cdot 2 = 2 \checkmark \\ 3 = 2 \cdot 1 + \underline{\underline{1}} & & 3 - 2 \cdot 1 = 1 \checkmark \end{array}$$

$$\begin{array}{rcl} 3 - 2 = 1 \\ 3 - (8 - 3 \cdot 2) = 1 \\ 3 - 8 + 3 \cdot 2 = 1 \\ 3 \cdot 3 - 8 = 1 \end{array}$$

$$\begin{array}{rcl} 3 \cdot (11 - 8) - 8 & = 1 \\ 3 \cdot 11 - 3 \cdot 8 - 8 & = 1 \\ 3 \cdot 11 - 4 \cdot 8 & = 1 \\ 3 \cdot 11 - 4 \cdot (1680 - 11 \cdot 152) & = 1 \end{array}$$

$$\begin{aligned}
 & 3 \cdot 11 - 4 \cdot (1680 - 11 \cdot 152) = 1 \\
 & 3 \cdot 11 - 4 \cdot 1680 + 4 \cdot (11 \cdot 152) = 1 \\
 & 3 \cdot 11 - 4 \cdot 1680 + 608 \cdot 11 = 1 \\
 & 611 \cdot 11 - 4 \cdot 1680 = 1 \\
 \\
 & 1680s + 11t = \gcd(1680, 11) \\
 & 1680s + 11t = 1 \\
 & 1680 \cdot (-4) + \boxed{11} \cdot 11 = 1 \\
 \\
 & d = 611 \checkmark
 \end{aligned}$$

Megkaptuk, hogy a d értéke 611 lett. Ez lesz a titkos kulcsunk. Mostmár tudunk üzeneteket titkosítani. Az üzenetünket az alábbi ABC alapján számokká alakítjuk:

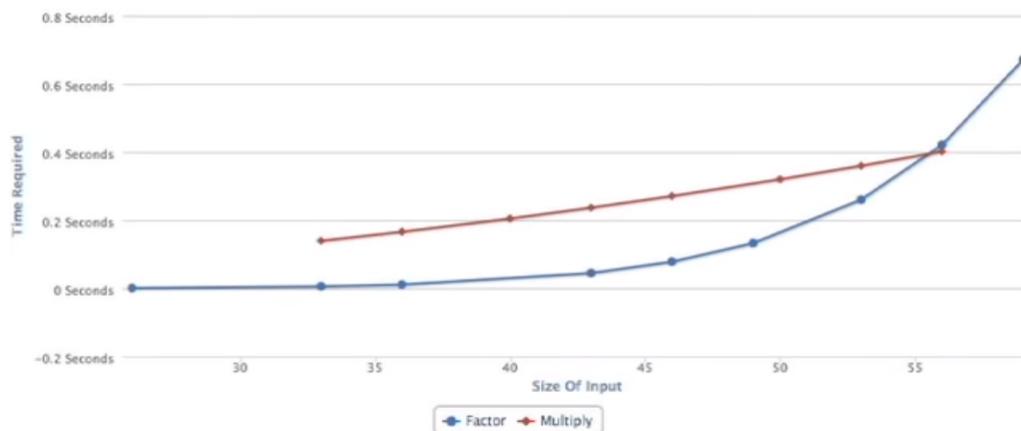
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
t	u	v	w	x	y	z												
20	21	22	23	24	25	26												
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
T	U	V	W	X	Y	Z	0	1	2	3	4	5	6	7	8	9		
46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62		

Az ABC-hez pár darab plussz karakter, ami kímaradt: $63 = ",$, $64 = \text{szóköz}$, $65 = "!"$. Az üzenet az alábbi lesz: M = Hello, World! Az ABC alapján az üzenet számokká alakítva és feldarabolva megfelelő méretre:

$m = 345\ 1212\ 1563\ 644\ 915\ 181\ 246\ 5$. A számok kódolás után: $c = 216\ 720\ 1020\ 343\ 1101\ 298\ 533\ 77$. Ezt dekódolva visszakapjuk az m -et, és ezt vissza tudjuk keresni az ABC-nkben, hogy meg legyen az eredeti üzenet (M). Ahhoz, hogy teljesen jól működjön, be vezethetnénk egy karakter elválasztó karaktert is, hogy tudjuk, mikor van vége egy karakternek és mikor kezdődik egy új.

Ha elrontjuk a titkosítót és a szöveget nem egyben, hanem betűnként titkosítjuk, akkor sokkal könnyebb lesz feltörni. Ilyen esetben ha bárki tudja, hogy az eredeti szöveg milyen nyelven íródott, akkor csak egyszerűen használhat egy betű gyakoriság alapú statisztikát. Így a szöveg nagyrészt visszafejthető értelmes szövegre, kivéve, ha az eredeti szöveget olyan régen titkosították, hogy azóta már megváltozott ez a betű gyakoriság alapú statisztika.

Egy érdekes kép arról, hogy mennyivel több idő kiszámítani a faktorizálást, mint a sima szorzást:



16.4. Összefoglaló

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

A hibásan implementált RSA törésről írtam az 1 oldalas "esszét".

Kép a nyers szövegről LibreOffice Writerben 12-es betűmérettel és 1-es sorközzel:

Az RSA egy nyílt kulcsú, vagyis asszimmetrikus titkosító algoritmus. A moduláris számelméletben és a primszámléletben alapszik. Az RSA titkosításnál van egy nyílt és egy titkos kulcsunk. A nyílt kulcsunkat bárki ismerheti, ezzel arra használjuk, hogy a nekünk szánt üzeneteket titkosításával a nyílt kulcsal. A titkosított üzeneteket mindenhol csak a titkos kulccsal lehet visszafeleíteni. Titkos kulcs nélkül az üzenetet felülni lehetetlen, mert olyan sok időt venne igénybe. A kulcsok létrehozásának a menete úgy történik, hogy valasztunk két különböző prím számot. Ezek lesznek a p és q. Fontos, hogy ez a két szám megfelelően nagy kell legyen, abhoz, hogy ne lehessen könnyen feltörni a kulcsot. A két prímet összeszorozva megkapunk n-t. Ez az n szám lesz a nyílt kulcsunk egyik része. Ennek az n-nek ki kell szamolni az Euler-féle $\phi(n)$ függvény értékét. Ezt könnyni ki tudjuk számolni, mert prímszámok esetén egyszerű a képlet: Miután ezzel megyünk vissza, valasztunk kell egy megfelelő számot. Az e egy 1 és $\phi(n)$ közötti egész szám kell legyen. Valamint relátiív prime kell legyen $\phi(n)$ -nek. Ez azt jelenti, hogy e és $\phi(n)$ legnagyobb közös osztója 1. Azért kell, hogy relátiív primek legyenek, mert ez biztosítja, hogy a kiszámításnál használt diofantoszi egyenletek lesznek megoldásai. Ez könnyen ellenőrizhető az Euklideszi algoritmussal. Gyakran választják e-nek a 65537 számot, azaz $2^{16} + 1$ -et. Viszont a gyorsabb számítás eredménye a 3, 5, vagy 35 is gyakori választás. Viszont ilyen kis számokkal sokkal nagyobb a kockázat is. Ha ezzel megyünk vissza, akkor sikeresen elkerülhetjük a nyílt kulcsunkat, ami n-ból és e-ból tevődik össze. Ezután ki kell számolniuk a d-t. Ennek a képlete a következőtől a két pozitív egész szám. A d-t a bővített Euklideszi algoritmussal (Extended Euclidean algorithm) számolhatuk ki. Fontos, hogy d-re több jó megoldás is létezik. A d-ból és az n-ból áll a titkos kulcsunk. Ezt a d-t soha senkivel nem szabad megosztanunk, hiszen emiatt lesz a titkosított üzenet könnyen visszafeleíthető, p és q is nagyon fontos, hogy ne tudjuk ki, mielőtt tudnánk gyorsan kiszámolni az n-t és a d-t egy bizonyos és számmal. Az n visszafeleíse elég nagy p és q esetén nagyon sok időbe kerülhet. A bináris alakban felírt n szám bitjeinek száma határozza meg a rejtelmeződ hosszúságát. Ez általában 512, 1024, 2048. Minél hosszabb, annál több idő felülni, azaz annál biztosan aggaszt. De a kulcs generálása is annál több időt fog igénybe venni. Ha egy üzenetet titkosítanunk, akkor először is meg kell kérni a címzettet, hogy küldje el a nyilvános kulcsát, ha azt meg nem bírjukoljuk. Ezután az M-ei előlt szöveges üzenetünk valamelyen karakterkódolással számokká kell alakítunk (pl.: ASCII), és úgy darabolunk fel ezt a kapott m üzenetet, hogy minden esetben minőségi legyen, mint n. Végül a kódolt szöveget az alábbi képpel kapjuk meg. Ezt a kódolt szöveget küldjük el a címzettek és ezek után c-vel fogjuk jelölni. A címzet ezt a kódolt szöveget vissza tudja fejteni a titkos kulcsaval, azaz d-val, az alábbi módon: Így megkapja az m-et, azaz az eredeti üzenetet, csak számokká alakítva. Végül a megfelelő karakterkódolással vissza tudja alakítani az m-et az eredeti üzenetre, azaz M-re. Egy példa: Két különböző prímszám p = 41 és q = 43, n = p * q, azaz n = 41 * 43, n = 1763. A $\phi(n)$ -ről egyrészt ki tudjuk számolni: $\phi(n) = (p - 1) * (q - 1)$, így $\phi(1763) = 40 * 42$, $\phi(1763) = 1680$. Ebben az esetben e = 11 lesz. Ez így kapjuk meg, hogy az Euklideszi algoritmussal elkezdjük kiszámolni a legnagyobb közös osztóját e-nek és $\phi(n)$ -nek, azaz 1680-nak, úgy, hogy az e-től indul. Ezt akár egy gcd calculatorral is megnehézíthetjük, ha nincs kedvünk számolni (pl.: www.alcula.com/calculators/math/gcd). Addig számoljuk, ameddig egy olyan e-t kapunk, ahol a legnagyobb közös osztó 1 lesz. A nyilvános kulcsunkat így már meg is kaptuk: e = 11, n = 1763. A d-kiszámítása a fent leírt módon történik, szépen vissza fejtjük az Euklideszi algoritmusból a Bővített Euklideszi algoritmussal a d-t. Megkapjuk, hogy a d értéke 611 lesz. Ez lesz a titkos kulcsunk. Most már tudunk üzeneteket titkosítani. Az üzenetünk az alábbi ABC alapján számokká alakítjuk: Az ABC-hez pár darab plüssz karakter, ami kímaradt: 63 = "", 64 = szóköz, 65 = "!". Az üzenet az alábbi lesz: M = Hello, World! Az ABC alapján az üzenet számokká alakítva a és feldarabolva megfelelő méretre: m = 345 1212 1563 644 915 181 246 5. A számok kódolás után: c = 216 720 1020 343 1101 298 533 77. Ezt dekódolva visszakapjuk az m-et, és ezt vissza tudjuk keresni az ABC-nkben, hogy meg legyen az eredeti üzenet (M). Abhoz, hogy teljesen jól működjön, bevezethetünk egy karakter elválasztó karaktert is, hogy tudjuk, mikor van vege egy karakternek és mikor kezdődik egy új. Ha elrontjuk a titkosítót és a szöveget nem egyben, hanem betűnként titkosítjuk, akkor sokkal könnyebb lesz feltörni.

Ilyen esetben ha bárki tudja, hogy az eredeti szöveg milyen nyelven íródott, akkor csak egyszerűen használhat egy betű gyakoriság alapú statisztilát. Így a szöveg nagyreszt visszafeleíthető értelmes szövegre, kivéve, ha az eredeti szöveget olyan régen regen titkosították, hogy azóta már meg változott ez a betű gyakoriság alapú statiszтика. Egy érdekes kép arról, hogy mennyivel több idő kiszámítani a faktorizálást, mint a sima szorzást.

17. fejezet

Helló, Gödel!

17.1. Gengszterek

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

Megoldás forrása: [OOCWC](#)

A lambda kifejezések a C++ 11-ben jöttek be. A lényegük, hogy olyanok, mint a függvények, viszont nincsen nevük. Ez amiatt van, mert olyan függvényeknél használjuk a lambda kifejezéseket, amelyek nem fognak megismétlődni a programunkban többször.

Lambda kifejezés C++-ban:

```
[ ] (paraméterek) -> visszatérési érték típusa
{
}
```

Egy lambda kifejezés visszatérési értékét nem muszáj megadni, mert ezt a fordító kikövetkezteti.

Az OOCWC-ben lévő kód részlete:

```
std::sort ( gangsters.begin(), gangsters.end(), [this, cop] <-
            ( Gangster x, Gangster y )
{
    return dst ( cop, x.to ) < dst ( cop, y.to );
} );
```

A fenti kód részletben a sort függvényel rendezünk, és ebben a függvényben fordul elő a lambda kifejezésünk paraméterként. A sort függvény három paramétert vár: first, last és compare. Itt a lambda kifejezés fogja megmondani, hogy mi alapján rendezzen a sort függvény, azaz a compare paramétert adjuk meg a lambdával. A lambda kifejezésünk egy x és egy y nevű Gangster objektumot vár. Akkor lesz a visszatérési érték igaz, ha az x közelebb van a rendőrhöz (cop), mint az y. Így miután a sort függvény lefut, a vectorunkban az első helyen lesz a rendőrhöz legközelebb lévő gangster.

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 UDPORG repóban!

Megoldás forrása: [customalloc.cpp](#)

Egy allokátor célja, hogy memóriát biztosítson egy típus számára. Majd később, ha már nem használjuk ezt a memóriát, akkor ide lehet vissza adni a lefoglalt memóriát. Írhatunk saját allokátort is, az allocator_traits biztosítja, hogy meglegyenek az alapvető függvények.

Probléma lehet, ha konténereket használunk saját allokátorral. Erre egy megoldás az, ha megmondjuk a konténernek, hogy milyen allokátort adjon az elemeinek. A scoped_allocator osztály segítségével egy külső és egy belső allokátort is számon tarthatunk. A külsővel a konténer helyet foglal az elemeinek, a belsőt pedig az elemek kapják meg.

A forrásunkban kiiratjuk azt, hogy mekkora memória területet foglalunk le, és hogy ez hol kezdődik. A memória terület megkétszereződik, ha nem elég a lefoglalt memória az elemek. 5 elemet próbálunk eltárolni a vektorban. minden elem esetén újabb memóriát fog lefoglalni, így végül az 5. elemnél már 8 objektum számára foglal memóriát.

A forrásunkban a deallokálás nem jár memória folyással. Egy gond az, hogy az egyszer lefoglalt memória, miután már nem használjuk, nem osztható ki újra, mert a pointert állandóan léptetjük tovább. Nem másolhatunk minden memcpay-vel arrébb annyival amennyi memóriánk felszabadult, mert a vectorban lévő értékünk nem egyezne meg vele. Megoldás lehet még az, ha figyeljük, hogy mennyi memória van már felszabadítva és ha van elég felszabadított memóriánk akkor ezt adjuk át, és nem a következő arena.q-t. Ebben az esetben viszont az lesz a gond, hogy mivel minden dupláztódik a lefoglalni kívánt memória, ezért sosem lesz annyi felszabadított memória, hogy az elég legyen. A forrásunkban ha folyton a legelső arena.q-t adjuk át, akkor ki tudjuk használni teljesen a kapacitását, és az elemek másolásával sem lesz probléma.

```
char *q = arena.q;
std::cout << "q " << static_cast<const void *>( q ) << std::endl;
//arena.q += n*sizeof(T);
return reinterpret_cast<T*>( q );
```

Ebben az esetben sem fog folyni a memória. Mivel minden a kezdeti arena.q-t adjuk át, ezért a memória cím minden ugyan az lesz memória foglalásnál.

A program futása:

```
ujhazi@ubuntu:~/Downloads$ sublime-text.subl customalloc.cpp
ujhazi@ubuntu:~/Downloads$ g++ customalloc.cpp -o customalloc.out
ujhazi@ubuntu:~/Downloads$ ./customalloc.out
Allocating 1 objects of 4 bytesi=int hivasok szama 1 hivasok szama 1
q 0x7fe875755064
Allocating 2 objects of 8 bytesi=int hivasok szama 2 hivasok szama 2
q 0x7fe875755068
Allocating 4 objects of 16 bytesi=int hivasok szama 3 hivasok szama 3
q 0x7fe875755070
v 0x7fe875755000
o 0x7fe875755000
0 0x7fe875755000
42
43
44
42
43
44
ujhazi@ubuntu:~/Downloads$ 
```

17.3. STL map érték szerinti rendezése

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

Megoldás forrása: [map.cpp](#)

A map kulcs, érték párokat tárol el. Két értékhez nem tartozhat ugyan az a kulcs. A kulcs és az érték is lehet akár string, vagy int, stb. A map létrehozása az alábbi módon történik:

```
map<int, int> szamok;
```

Az első int a kulcs típusát fogja jelenteni, a második pedig az érték típusát. A "szamok" pedig a map neve, ezzel fogunk rá hivatkozni a későbbiekben. Miután elkészítettük a mapet, töltük fel kulcs, érték párokkal:

```
szamok.insert(pair<int, int>(1, 40));
szamok.insert(pair<int, int>(2, 30));
...
```

Ezt az insert függvénytel tehetjük meg, a fenti sorokban látszik, hogy az 1-es kulcshoz a 40-es érték fog tartozni, a 2-es kulcshoz pedig a 30-as érték, stb. Miután feltöltöttük tetszőleges mennyiségű adattal, ki is irathatjuk ezeket:

```
map<int, int>::iterator itr;

for (itr = szamok.begin(); itr != szamok.end(); itr++) {
    cout << itr->first << '\t' << itr->second << '\n';
}
```

A begin() függvény a szamok map első kulcs, érték pájára fog mutatni, az end() függvény pedig az utolsóra. Így egy adott kulcs, érték pár kulcsát az itr->first-el kapjuk meg, még az értéket az itr->second-el.

```
typedef function<bool(pair<int, int>, pair<int, int>)> Comparator;

Comparator comp = [](pair<int, int> also, pair<int, int> masodik) {
    return also.second < masodik.second;
};
```

Egy lambda függvényben összehasonlítjuk két kulcs, érték pár értékeit. Visszatérési értékként igazat kapuk, ha az első érték kisebb, mint a második, egyébként hamisat. Végül sorbarendezzük érték szerint a párokat és kiiratjuk a végeredményt:

```
set<pair<int, int>, Comparator> osszehas(szamok.begin(), szamok.end(), comp);

for (pair<int, int> element : osszehas) {
    cout << element.first << '\t' << element.second << '\n';
}
```

A program futása:

```
ujhazi@ubuntu:~/Desktop$ g++ map.cpp -o map.out
ujhazi@ubuntu:~/Desktop$ ./map.out

Kulcs szerint rendezve:
Kulcs Érték
1      40
2      30
3      60
4      20
5      50
6      10

Érték szerint rendezve:
Kulcs Érték
6      10
4      20
2      30
1      40
5      50
3      60
ujhazi@ubuntu:~/Desktop$ □
```

17.4. Alternatív Tabella rendezése

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

Megoldás forrása: [Wiki2Matrix](#)

Megoldás forrása: [AlternativTabella](#)

A Java nyelvben a Comparable Interface-eket arra használjuk, hogy rendezzük az objektumokat. Ez az interfész a java.lang csomagban található, és csak egy függvényt tartalmaz, a compareTo(Object) függvényt.

```
public int compareTo(Object obj)
```

A fenti függvény a jelenlegi objektumot összehasonlítja egy másik objektummal. Visszatérési értékként egy számot kapunk. A szám pozitív, ha a jelenlegi objektum nagyobb, mint a másik objektum. Negatív, ha az objektum kisebb, mint a másik objektum, és végül a szám 0, ha egyenlő a két objektum.

String objektumokat, Csomagoló osztály objektumokat és Felhasználó által definiált osztály objektumait tudjuk rendezni.

A jelenlegi forrásunkban a Csapat osztály használja a Comparable interfészt.

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

A compareTo függvény kap egy csapatot paraméterként és így összehasonlítja két csapat értékeit. Ha a jelenlegi csapat értéke kisebb, mint a paraméterként kapott csapaté, akkor -1 számmal tér vissza (negatívvval), ellenkező esetben 1-el, és ha a két csapat értéke egyenlő, akkor 0-val.

```
ujhazi@ubuntu:~/Desktop$ javac AlternativTabella.java
ujhazi@ubuntu:~/Desktop$ java AlternativTabella
iteracio...
norma = 0.05918759643574294
φsszeg = 1.0
iteracio...
norma = 0.014871507967965858
φsszeg = 0.9999999999999999
iteracio...
norma = 0.006686056768932613
φsszeg = 1.0
iteracio...
norma = 0.007776749198562133
φsszeg = 1.0
iteracio...
norma = 0.007661483555477314
φsszeg = 0.9999999999999999
iteracio...
norma = 0.007501657116770109
φsszeg = 0.9999999999999998
iteracio...
norma = 0.007532790726590474
φsszeg = 0.9999999999999998
iteracio...
norma = 0.007538144256251714
φsszeg = 0.9999999999999998
iteracio...
```

```
ujhazi@ubuntu:~/Desktop$ javac Wiki2Matrix.java
ujhazi@ubuntu:~/Desktop$ java Wiki2Matrix
A x ontot-szerez y-tól matrix
```

```
0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 2, 1, 0,
0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1,
1, 2, 0, 0, 0, 2, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0,
0, 0, 1, 0, 1, 0, 1, 1, 2, 1, 2, 1, 0, 1, 1, 1,
0, 1, 1, 0, 0, 1, 1, 2, 1, 1, 1, 1, 0, 1, 0,
2, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 2, 1, 0,
1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 2,
1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0,
1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 2, 1, 1, 0, 1, 1,
1, 2, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0,
1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1,
0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 1, 0, 1, 0, 0, 0,
1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 2, 0, 0, 0, 2,
2, 1, 1, 1, 1, 1, 2, 0, 0, 0, 0, 1, 1, 0, 0, 0,
2, 0, 2, 0, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 0, 1,
1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 2, 1, 0, 0, 0,
```

Sor is oszlop φsszegekkel

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 forrása: [ActivityEditor.java](#)

Gyorsbillentyű kombináció implementálása. Az egeren lévő jobb gomb lenyomásával történő új tevékenység vagy props file létrehozása lesz lehetséges gyorsbillentyűkkel. Láthatjuk hogy Ctrl + E esetén egy új tevékenységet hozunk létre, ami lényegében, egy új mappa lesz a könyvtárszerkezetben. Megkeressük a ki-jelölt elemet fában, ennek a nevéhez hozzáfűzzük az "/Új altevékenység"-et és ha ilyen nevű könyvtár még nincs, akkor létrehozunk egyet, majd egy ennek megfelelő elemet hozzáadunk a fához (mint a kiválasztott elem gyermeke). A Ctrl+T gyorsbillentyű lenyomásakkor is hasonlóan járunk el, csak könyvtár helyett egy props-t adunk hozzá a faszerkezethez.

18.2. OOCWC Boost ASIO hálózatkezelése

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

Megoldás forrása: [carlexer.ll](#)

```
{ CAR } { WS } { INT }
{
    std::sscanf(yytext, "<car %d", &m_id);
    m_cmd = 1001;
}
```

A carlexer.ll fájlban legtöbb esetben a sscanf függvényt használjuk, ez a filekezelő függvények egyike. A sscanf függvény bemenetet olvas egy formázott stringből.

Szintaxisa az alábbi:

```
int sscanf(const char *str, const char *format, ...)
```

Az első paraméterként megadott stringből fog olvasni és a formatban pedig egy, vagy több whitespace karaktert (\b, \t, \n, \v, \f) adhatunk meg, illetve egyéb karaktereket is, például: %.

Az int miatt tudjuk, hogy a függvény visszatérési értéke egy szám lesz. Ez a szám a sikeresen beolvasott értékek darabszáma.

A lexert a TCP porton kapott információ feldolgozására használjuk. A lexer bemenetként a TCP socket-ben lévő stringet kapja meg.

18.3. SamuCam

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

Megoldás forrása: [SamuCam](#)

```
std::string videoStream = parser.value ( webcamipOption ) . ←  
    toStdString ();  
SamuLife samulife ( videoStream, 176, 144 );
```

A program futtatásánál paraméterként megadott IP címet az IP webkamerának fogjuk átadni paraméterként. A headerben két fontos tag van:

```
std::string videoStream;  
cv::VideoCapture videoCapture;
```

Megnyitjuk a video streamet. A videoCapture objektum open() függvénye megnyitja a paraméterként kapott videó fájlt. Utána beállítjuk a videó méretét és FPS-ét. Futás előtt betöltünk egy CascadeClassifier-t, ez fogja elemezni a képeket. A load függvény segítségével megnyit egy classifieret, ami egy emberi arcot ír majd le.

Eztán 80 milliszekundumonként beolvasunk egy képkockát a read függvény segítségével, ami a paraméterként kapott cv::Mat tömbbe menti a beolvasott képet. Utána átméretezzük a képet és interpoláljuk. Ez után átváltjuk a kép color space-t grayscale-re, majd kiegyenlítjük a hisztogramját ennek. Utána a detectMultiScale függvényel különböző méretű arcokat keresünk az bemeneti képben, a megtalált arcok egy rectangle listaként térítjük vissza. Az első talált arcból egy QImage-t készítünk, amit majd a SamuBrain osztály fog feldolgozni a faceChanged signal után. Ez után az arc köré rajzolunk egy keretet, amiből egy új QImage-et csinálunk, amit át adunk a SamuLife-nak, ez frissíteni fogja a megjelenítendő webkamera képét.

A program futása:

18.4. BrainB

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

Megoldás forrása: [BrainB](#)

Qt-ban a slot-signal objektumok közötti kommunikációra használjuk. Egy signal akkor fut le, ha egy esemény bekövetkezik. A slot egy függvény, ami akkor hívódik meg, ha egy signal bekövetkezik. A connect függvényel tudjuk a slotokat a signalokhoz kapcsolni. A signal signatureje meg kell egyezzen a signalt kapó slot signaturejével. A forrásunkban két connect van:

```
connect ( brainBThread, SIGNAL ( heroesChanged ( QImage, ←
    int, int ) ),
this, SLOT ( updateHeroes ( QImage, int, int ) ) );
connect ( brainBThread, SIGNAL ( endAndStats ( int ) ),
this, SLOT ( endAndStats ( int ) ) );
```

Ez azt jelenti, hogy ha a brainBThread-ben a heroesChanged signal bekövetkezik, akkor lefut az updateHeroes függvény. Az endAndStats signal akkor következik be ha lejár a futási idő. Ekkor kiíródik a debug üzenet, hogy vége a játéknak és kimenti az eredményt.

A Qt-ban van egy újabb slot-signal szintaxis. Az új szintaxissal használhatunk 2 QObjectet:

```
connect (
    sender, &Sender::valueChanged,
    receiver, &Receiver::updateValue
);
```

QObjectek helyett akár függvényeket is használhatunk.

```
connect (
    sender, &Sender::valueChanged,
    someFunction
);
```

Ez lehetővé teszi azt, hogy a bind függvényel együtt használjuk

```
connect (
    sender, &Sender::valueChanged,
    std::bind( &Receiver::updateValue, receiver, "←
        senderValue", std::placeholders::_1 )
);
```

A C++11-es lambda kifejezésekkel is használhatjuk.

```
connect (
    sender, &Sender::valueChanged,
    [=] ( const QString &newValue ) { receiver->updateValue( ←
        "senderValue", newValue ); }
```

19. fejezet

Helló, Lauda!

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 forrása: [KapuSzkenner](#)

A forrás TCP kapcsolatot próbál létesíteni az 1-es porttól az 1024-es portig. Ha egy porton sikeresen kiépül a kapcsolat, akkor ki írja a port számát és mellé azt, hogy figyeli. Ha nem sikerül kapcsolatot létesíteni, akkor valamilyen kivételt kapott el a catch blokkunk. A forrásban bármilyen kivételt kapunk, azt úgy kezeljük le, hogy ki írjuk, hogy az a bizonyos portot nem figyeli semmi. Ha a catch blokkba teszünk egy sort, ami ki írja a kivételt, akkor többet tudhatunk meg:

```
    } catch (Exception e) {
        System.out.println(e);
    }
```

Ha ezzel a sorral lefuttatjuk a forrásunkat úgy, hogy paraméterként megadjuk a localhost IP címét, azaz 127.0.0.1-et, akkor az alábbi kivételt dobja a program:

```
java.net.ConnectException: Connection refused (Connection refused)
```

Ez a kivétel jelenthet egy pár dolgot: a kliens és a szerver nincsenek egy hálózaton, a szerver nem fut, a szerver fut, de nem hallgatózik azon a porton, amin a kliens próbál csatlakozni, vagy esetleg a tűzfal blokkolja azt a portot, vagy IP címet, amire csatlakozni próbálunk. A mi esetünkben biztos, hogy egy hálózaton vannak, és a tűzfal sem blokkolja, tehát tudjuk, hogy egy szerver sem hallgatózik azon a porton, tehát azt a portot nem figyeli semmilyen program.

Egy másik kivételt is kikényszeríthetünk a forrásunkból, ha nem adunk meg paraméterként IP címet. Ilyenkor az alábbi kivételt dobja a forrás:

```
java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
```

Ez amiatt van, mert a lentebb lévő sor próbálja használni az args[0]-t, de mivel nem adtunk meg paraméterként semmit, IP címet sem, ezért nem fog működni.

```
java.net.Socket socket = new java.net.Socket(args[0], i);
```

A program futtatása:

```
ujhazi@ubuntu:~/Downloads$ javac KapuSzkenner.java
ujhazi@ubuntu:~/Downloads$ java KapuSzkenner 127.0.0.1 > output
ujhazi@ubuntu:~/Downloads$
```

Az output fájl tartalma:

```
0 nem figyeli
1 nem figyeli
2 nem figyeli
3 nem figyeli
4 nem figyeli
5 nem figyeli
6 nem figyeli
7 nem figyeli
8 nem figyeli
9 nem figyeli
10 nem figyeli
11 nem figyeli
12 nem figyeli
13 nem figyeli
14 nem figyeli
15 nem figyeli
16 nem figyeli
17 nem figyeli
18 nem figyeli
19 nem figyeli
20 nem figyeli
```

19.2. AOP

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

Megoldás forrása: AOP

Ez a feladat az aspektus orientáltságról szól. Ez azt jelenti, hogy van egy komponensnyelvünk és egy aspektusnyelvünk, amelyben le vannak írva az aspektusok. Ez a két nyelv akár azonos is lehet. A forráunk komponens nyelven van megírva, és erre tudunk aspektusokat alkalmazni. Egy aspektus módosítja a program futását, vagy a működését. Ezt hívjuk weaving-nek, azaz szövésnek. Szövéskor a megadott

forráskódhoz "szövünk" hozzá új kódcsipetet, bizonyos előre megadott helyekre. Azokat a helyeket, ahova új kódot "szőhetünk" join point-nak nevezzük.

Az LZWBinfa Java forrásába szövünk bele, hogy legyen inorder, preorder, és posztorder rendezés is. Az around() függvényben lévő kód a megszabott hívás helyett fut le. Meg kell adni utána, hogy milyen függvényhívást szeretnénk helyettesíteni. Erre használhatjuk a pointcut-ot. A target() függvényben paraméterként megadott típussal megegyező join pointokkal fog matchelni a target(). A call() függvény a paraméterként kapott hívás. Az args() függvényben megadott paraméterek típusával és számával megegyező metódusokat jelent.

19.3. Android Játék

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

Megoldás forrása: [RPS](#)

Ebben a feladatban egy kő-papír-olló játékot készítettem. A program 3 gombból és 2db TextView-ből áll. A TextView szerepe annyi, hogy ki írja a játékos és a számítógép pontjait, és ha ez változna, akkor frissül ez a szöveg is. A 3 gomb a kő, papír és az olló. Amikor rányomunk az egyik gombra, akkor egy random számgenerátor generál egy számot 0-tól 100-ig.

```
if (result < 34) {
    computerChoice = "rock";
}
else if (result < 67) {
    computerChoice = "paper";
}
else {
    computerChoice = "scissor";
}
```

Ha ez a szám kisebb, mint 34, akkor a számítógép követ fog választani. Ha a szám kisebb, mint 67, de nagyobb, mint 34, akkor papírt, egyéb esetben pedig ollót választ. Ilyenkor már tudjuk a számítógép és a játékos választását is, tehát megnézhetjük, hogy ki fog nyerni.

```
private void checkWin() {
    switch (userChoice) {
        case "rock":
            switch (computerChoice) {
                case "rock": draw(); break;
                case "paper": computerwins(); break;
                case "scissor": playerwins(); break;
            }
            break;
        case "paper":
            switch (computerChoice) {
                case "rock": playerwins(); break;
```

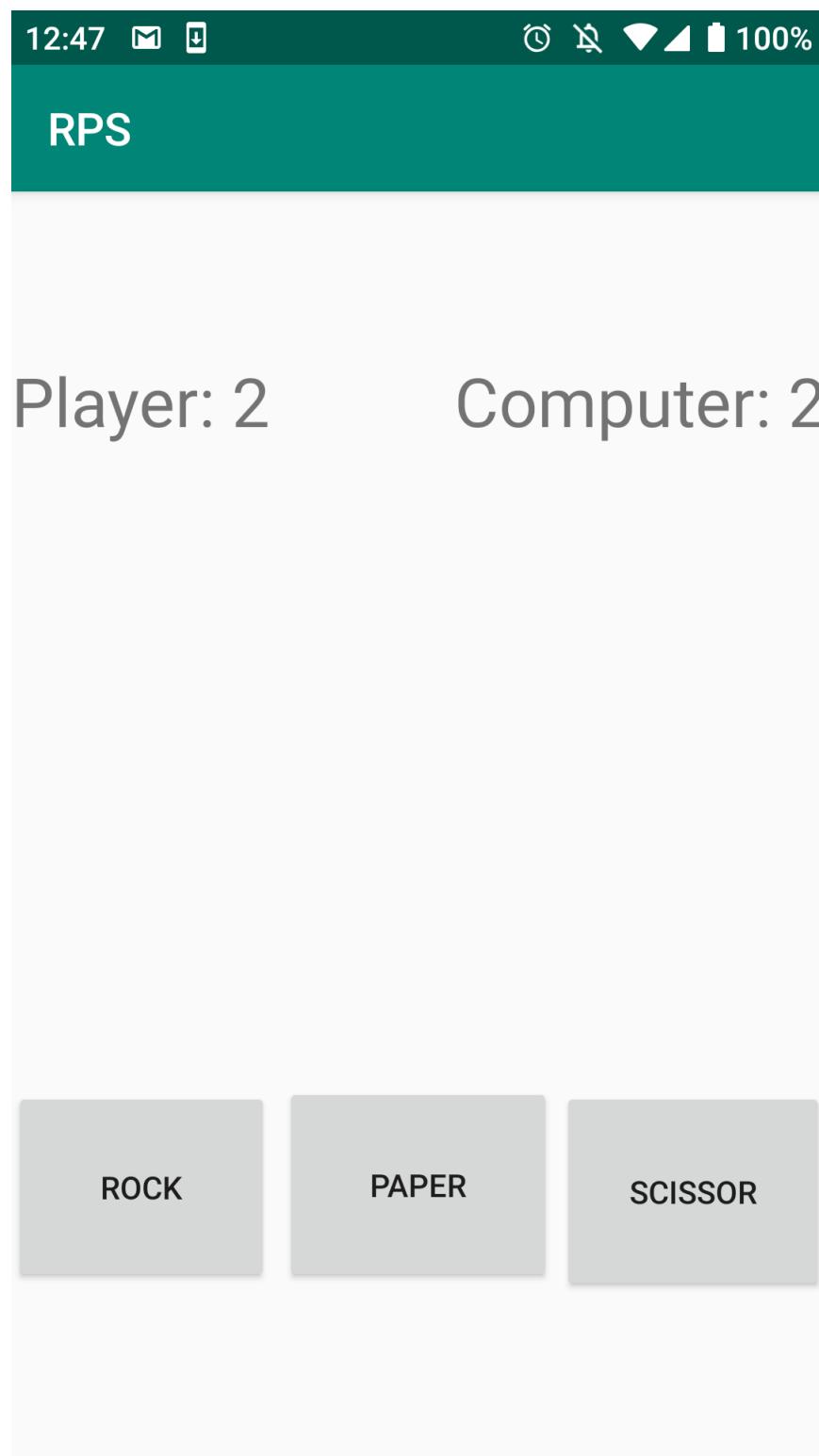
```
        case "paper": draw(); break;
        case "scissor": computerwins(); break;
    }
    break;
case "scissor":
    switch (computerChoice) {
        case "rock": computerwins(); break;
        case "paper": playerwins(); break;
        case "scissor": draw(); break;
    }
    break;
}
```

A checkWin függvény fogja megnézni, hogy ki nyerte az adott kört. Ehhez switcheket használtam.

```
private void playerwins() {
    playerpoints++;
    Toast.makeText(getApplicationContext(), "Player won!", Toast.LENGTH_SHORT).show();
    updatePointsText();
}
```

Ha a játékos nyer, akkor a playerwins függvény fut le, ilyenkor növeljük a játékos pontját 1-el, és ki iratjuk, hogy "Player won!", végül frissítjük a TextView-ban kiírt szöveget. Ha a számítógép nyer ugyan ez történik, csak a játékos helyett a számítógéppel. Ha megegyezik a két választás, azaz mindenketten követ választottak például, akkor a draw függvény fut le. Ebben az esetben nem változnak a pontok, csak ki írjuk, hogy döntetlen lett az adott kör.

Kép a játék futásáról:



19.4. Junit teszt

A [https://progpater.blog.hu/2011/03/05/labormeres_othton_avagy_hogyan_dolgozok_fel_egy_pedat_poszt_kézzel_számított_mélységét és szórását dolgozd be egy Junit tesztbe \(sztenderd védési feladat volt korábban\).](https://progpater.blog.hu/2011/03/05/labormeres_othton_avagy_hogyan_dolgozok_fel_egy_pedat_poszt_kézzel_számított_mélységét és szórását dolgozd be egy Junit tesztbe (sztenderd védési feladat volt korábban).)

Megoldás forrása: Junit

Ahhoz, hogy megértsük a mélységet és a szórást, először próbáljuk meg kiszámolni papíron. A feladatban megadott teszt sorozattal fogunk dolgozni: 01111001001001000111. Ezt felírva az alábbi fát kapjuk:

KÉP

A mélység 0-tól kezdődik, tehát a gyökérelem mélysége 0. A szórás kiszámításához kelleni fog az átlag. Az átlagot úgy tudjuk kiszámítani, hogy minden ág mélységét összeadjuk és ezt az összeget elosztjuk az ágak darabszámával. Ez a mi esetünkben 2.75 ($(2+2+4+3) / 4$). A szórást úgy számoljuk ki, hogy vesszük az N számunkat, ez jelen esetben az ágak darabszáma, azaz 4. $(1 / (N-1))$ -et összeszorozzuk azzal az összeggel, amit úgy kapunk meg, hogy az egyes ágak mélységéből kivonjuk az átlagot és megszorozzuk önmagával ezt a számot. Ezt az értéket minden ágra kiszámoljuk és összeadjuk őket. A végeredmény, azaz a szórás ennek a szorzatnak a gyöke lesz.

20. fejezet

Helló, Calvin!

20.1. MNIST

Az alap feladat megoldása, +saját kézzel rajzolt képet is ismerjen fel, [progpaterw](#) -bol Háttérként ezt vetítük le: [prezi](#)

Megoldás forrása: [MNIST](#)

[tensorflow](#) oldalon lévő beginner kódöt kimásoltam és beillesztettem egy tf.py nevű fájlba. A .py kiterjesztésű fájl futtatásához Python szükséges, ezért a Python3-at telepítettem. Ezt követően lefuttatjuk a kódot a Python3 tf.py parancsal. Ilyenkor láthatjuk, hogy 0-tól 9-ig terjedő számjegyeket tanul a program 60.000 képből. Addig tanul, amíg nem lesz a pontosság (accuracy) 97%. Miután betanulta, 10.000 képpel teszteli, láthatjuk, hogy 97% fölött van a képen lévő számjegyek felismerésének pontossága.

Kép a forrás futtatásáról:

```
49504/60000 [=====>.....] - ETA: 0s - loss: 0.0753 - acc: 0.9
50336/60000 [=====>.....] - ETA: 0s - loss: 0.0752 - acc: 0.9
51168/60000 [=====>.....] - ETA: 0s - loss: 0.0750 - acc: 0.9
51936/60000 [=====>.....] - ETA: 0s - loss: 0.0751 - acc: 0.9
52768/60000 [=====>.....] - ETA: 0s - loss: 0.0755 - acc: 0.9
53568/60000 [=====>.....] - ETA: 0s - loss: 0.0754 - acc: 0.9
54400/60000 [=====>.....] - ETA: 0s - loss: 0.0752 - acc: 0.9
55232/60000 [=====>.....] - ETA: 0s - loss: 0.0750 - acc: 0.9
56064/60000 [=====>..] - ETA: 0s - loss: 0.0751 - acc: 0.9
56864/60000 [=====>..] - ETA: 0s - loss: 0.0749 - acc: 0.9
57728/60000 [=====>..] - ETA: 0s - loss: 0.0752 - acc: 0.9
58496/60000 [=====>..] - ETA: 0s - loss: 0.0751 - acc: 0.9
59296/60000 [=====>..] - ETA: 0s - loss: 0.0748 - acc: 0.9
60000/60000 [=====] - 4s 63us/sample - loss: 0.0747 - a
cc: 0.9770
      32/10000 [.....] - ETA: 7s - loss: 0.0076 - acc: 1.0
1696/10000 [==>.....] - ETA: 0s - loss: 0.1083 - acc: 0.9
3264/10000 [=====>.....] - ETA: 0s - loss: 0.1106 - acc: 0.9
5120/10000 [=====>.....] - ETA: 0s - loss: 0.1085 - acc: 0.9
6944/10000 [=====>.....] - ETA: 0s - loss: 0.0923 - acc: 0.9
8896/10000 [=====>.....] - ETA: 0s - loss: 0.0778 - acc: 0.9
10000/10000 [=====] - 0s 31us/sample - loss: 0.0777 - a
cc: 0.9764
ujhazi@ubuntu:~/Downloads$
```

20.2. Deep MNIST

Mint az előző, de a mély változattal. Segítő ábra, vesd össze a forráskóddal a [fóliafóliáját!](#)

Megoldás forrása: DeepMNIST

A Deep MNIST és az MNIST között az a különbség, hogy a Deep változatban több rétegen át szűri meg az adatokat. A Tensorflow-nak a Deep MNIST példáját használjuk ehhez a feladathoz. Megnyitjuk a vizsgálandó képet és úgy dekódoljuk, hogy a kimeneti kép color channelje grayscale legyen (a decode_png() függvényben az 1-es paraméter). A modellt betanítjuk, majd teszteljük. Tesztelésnél átadjuk a képet, majd az y_conv-ból kiválasztjuk azt az indexet, amelynek a legnagyobb az értéke, ez a legnagyobb valószínűséggel rendelkező karakter lesz.

A program futtatásakor a tanítás és a tesztelés sokkal több időt vesz igénybe, mint az előző feladatban. Itt a ciklus 1000 helyett 20000-szer fut le és a modellünk is bonyolultabb. Ha nem akarjuk, hogy állandóan újra kelljen tanítani a modellt, akkor a súlyokat el is menthetjük. Így elég lesz csak egyszer tanítani. Ezt a save függvénytelhetjük meg. A save minden változónak az értékét ki írja egy fileba, így a legközelebbi futásnál ezeknek a változóknak az értékét a file-ból betudjuk olvasni.

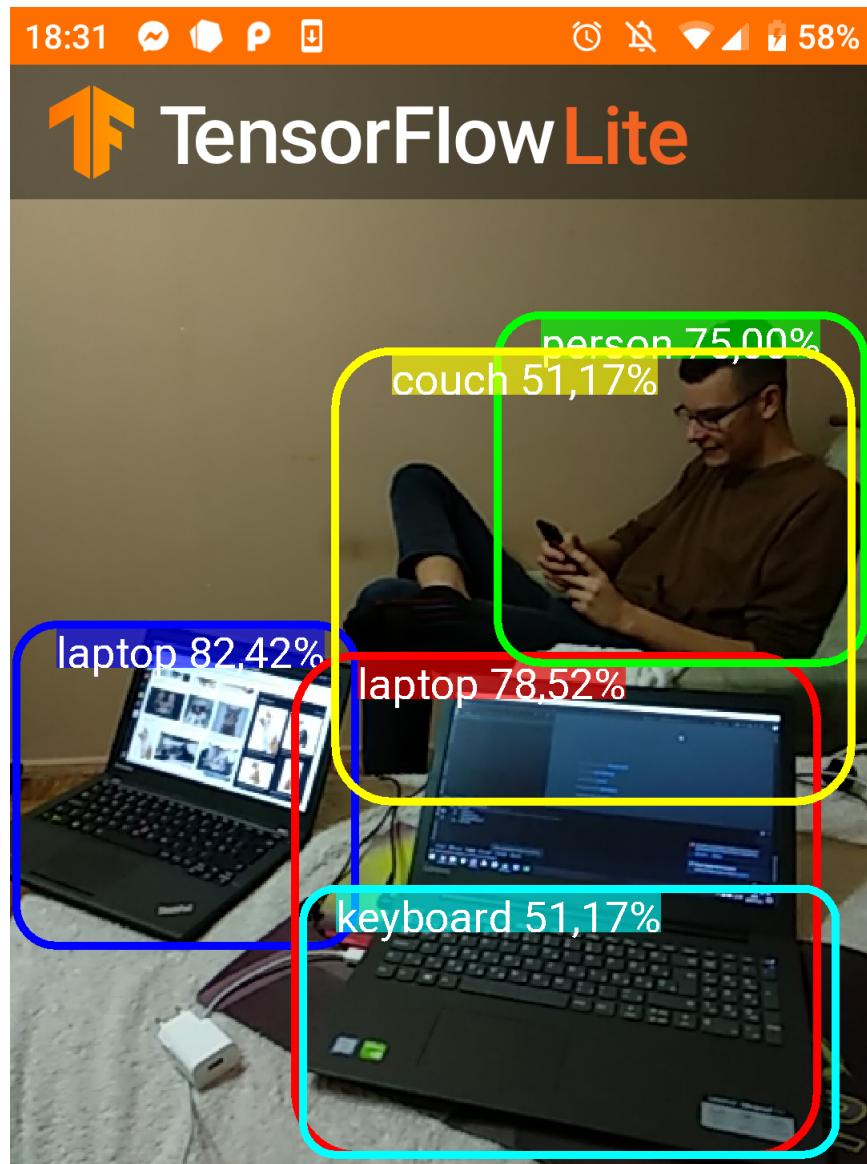
20.3. Android telefonra a TF objektum detektálója

Telepítsük fel, próbáljuk ki!

Megoldás forrása: [TFAndroid](#)

A forrásban megadott linken le klónozhatunk egy tensorflow github repót. Ebben az examples-master/lite/example mappában lévő android mappát tudjuk importálni Android Studioban.

Ha importáltuk, akkor az Android Studio minden szükséges fájlt letölt a projekthez. Ha ez kész, akkor a telefonunkat a géphez csatlakoztatva le futtatjuk a projektet és így fel települ a TFLiteObjectDetection program a telefonunkra. A telefonon automatikusan elindul a program, és már ki is tudjuk próbálni:



20.4. SMNIST for Machine

Készíts saját modellt, vagy használj meglévőt, lásd:[link](#)

Megoldás forrása: [SMNIST](#)

Az SMNIST a Semantic MNIST rövidítése. Az SMNIST-ben a képben elhelyezett objektumok számossága meg van határozva. Két féle kísérlet van: az SMNIST for Humans és az SMNIST for Machine. Az SMNIST for Humans az Object File System (OFS) kapacitását méri az emberekben. Az SMNIST for Machine pedig létező és jól ismert deep learning programokat vizsgál.

Az SMNIST nem olyan képeket kap, amelyeken számok vannak, hanem olyanokat, amelyen 0-tól 10-ig terjedő pontok vannak. A program feladata, hogy megpróbálja megmondani első ránézésre, hogy hány darab pontot lát. Emiatt a tanításnál sokkal kevesebb lesz a pontossága(kb. 60%) az MNIST-hez képest(kb. 97%).

Generálunk tanító képeket, ebből ugy 60000 darabot, mint az MNIST-nél. A generálást az smnistg.cpp-vel tehetjük meg. Ezt fordítva és futtatva az alábbi kapjuk:

```
g++ smnistg.cpp -o smnistg.out -lpng
./smnistg.out train-labels-idx1-ubyte train-images-idx3-←
      ubyte 60000
```

Tanítás után láthatjuk a pontosságokat:

```
ujhazi@ubuntu:~/Downloads/smnist-master/smnistg$ ./smnistg.out train-labels-idx1-ubyte train-images-idx3-ubyte 60000
smnist-4-59999.png saved
Stat:
0 6025
1 5977
2 5965
3 5928
4 6075
5 6067
6 6004
7 5930
8 6051
9 5978
ujhazi@ubuntu:~/Downloads/smnist-master/smnistg$ █
```

IV. rész

Irodalomjegyzék

20.5. Általános

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

20.6. C

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

20.7. C++

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

20.8. Lisp

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

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

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