

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	Halász, Dávid	2019. április 6.	

REVISION HISTORY

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

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	8
2.4. Labdapattogás	9
2.5. Szóhossz és a Linus Torvalds féle BogomIPS	11
2.6. Helló, Google!	13
2.7. 100 éves a Brun tétel	16
2.8. A Monty Hall probléma	17
3. Helló, Chomsky!	20
3.1. Decimálisból unárisba átváltó Turing gép	20
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	21
3.3. Hivatkozási nyelv	21
3.4. Saját lexikális elemző	22
3.5. l33t.1	23
3.6. A források olvasása	25
3.7. Logikus	27
3.8. Deklaráció	27

4. Helló, Caesar!	30
4.1. int *** háromszögmátrix	30
4.2. C EXOR titkosító	31
4.3. Java EXOR titkosító	33
4.4. C EXOR törő	35
4.5. Neurális OR, AND és EXOR kapu	38
4.6. Hiba-visszaterjesztéses perceptron	42
5. Helló, Mandelbrot!	43
5.1. A Mandelbrot halmaz	43
5.2. A Mandelbrot halmaz a std::complex osztállyal	46
5.3. Biomorfok	49
5.4. A Mandelbrot halmaz CUDA megvalósítása !!	51
5.5. Mandelbrot nagyító és utazó C++ nyelven	55
5.6. Mandelbrot nagyító és utazó Java nyelven !!	57
6. Helló, Welch!	60
6.1. Első osztályom	60
6.2. LZW	64
6.3. Fabejárás	69
6.4. Tag a gyökér	71
6.5. Mutató a gyökér	78
6.6. Mozgató szemantika	79
7. Helló, Conway!	81
7.1. Hangyaszimulációk	81
7.2. Java életjáték	81
7.3. Qt C++ életjáték	87
7.4. BrainB Benchmark	93
8. Helló, Schwarzenegger!	95
8.1. Szoftmax Py MNIST	95
8.2. Szoftmax R MNIST	95
8.3. Mély MNIST	95
8.4. Deep dream	95
8.5. Robotpszichológia	96

9. Helló, Chaitin!	97
9.1. Iteratív és rekurzív faktoriális Lisp-ben	97
9.2. Weizenbaum Eliza programja	97
9.3. Gimp Scheme Script-fu: króm effekt	97
9.4. Gimp Scheme Script-fu: név mandala	97
9.5. Lambda	98
9.6. Omega	98
10. Helló, Gutenberg!	99
10.1. Programozási alapfogalmak	99
10.2. Programozás bevezetés	103
10.3. Programozás	103
III. Második felvonás	105
11. Helló, Arroway!	107
11.1. A BPP algoritmus Java megvalósítása	107
11.2. Java osztályok a Pi-ben	107
IV. Irodalomjegyzék	108
11.3. Általános	109
11.4. C	109
11.5. C++	109
11.6. Lisp	109

Ábrák jegyzéke

2.1. EXOR működése	9
2.2. Monty Hall probléma	17

DRAFT

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/vegtelen

Végtelen ciklusnak nevezzük azt az utasítást, ami örökké futna, ha nem állítanánk le a programot. A végtelen ciklus nem minden esetben jelent rossz dolgot, csak akkor, ha olyan feladat során keletkezik, amelynek végrehajtása nem igényel ilyen ciklusos algoritmust. Ilyen ciklus lehet például az alábbi kódrészlet:

```
Program vegtelen.c {  
  
    int main() {  
        int i = 0;  
        while (i<=0) {  
            i--;  
            printf("\n vegtelen ciklus");  
        }  
    }  
}
```

A program a while utáni zárójelben lévő kifejezést értékeli ki. Mivel a az „i” változó értéke egyenlő vagy kisebb a nullával ezért ez a feltétel igaz lesz és a program lefut. Ennek eredményeképpen az i változó értéke eggyel kisebb lesz és újból indul az egész. Mivel az eredmény sose lesz nullánál nagyobb, ezért a feltétel mindig igaz lesz, azaz végtelen ciklusba kerül. Vannak esetek, amikor szükség van végtelen ciklusra, ekkor érdemes olyat programkódot használni, amiről lehet látni, hogy ezt direkt így akartuk és nem pedig szoftverhiba. Ilyen ciklus az alábbi példa, amikor nem töltjük ki a for ciklus fejlécét:

```
Program vegtelen_for {  
  
    int main() {  
        for(;;);  
    }  
}
```

```
}
```

Ez a végtelen ciklus 1 magot használ 100%-on. Ha több magot szeretnénk dolgoztatni, akkor hozzá kell adni a `#pragma omp parallel-t`, tehát a végleges kódrészlet így fog kinézni:

```
Program vegtelen_pragma.c {  
  
    int main() {  
        #pragma omp parallel  
        for(;;);  
    }  
  
}
```

Ezzel az elhelyezett OpenMP alapú parallel régiók segítségével olyan függvényt hívunk elő, amellyel könnyen kihasználhatjuk vele egy számítógép processzorainak teljes számítási kapacitását. Fontos megjegyezni, hogy fordítás során hozzá kell adni `-fopenmp` kódrészletet is, ezért végül így fog kinézni: `gcc vegtelen_pragma.c -o vegtelen_pragma -fopenmp` Ha olyan programra van szükség, ahol csak 0%-ban dolgoztatnak meg egy magot, akkor a `sleep(1)` függvényt kell használnunk:

```
Program vegtelen_sleep.c {  
  
    int main() {  
        for(;;) {  
            sleep(1);  
        }  
    }  
  
}
```

A `sleep()` függvény hatására a program a zárójelben megadott értékű másodpercre megáll, alszik. Ha a zárójelben nincs megadva argumentum, akkor meghatározatlan időre alszik. Ezzel érhető el, hogy a mag ne legyen 100%-ban kihasználva.

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

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

Megoldás videó:

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

```
Program T100  
{  
  
    boolean Lefagy(Program P)  
    {  
        if(P-ben van végtelen ciklus)
```



```
    return true;
  else
    return false;
}

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

```
Program T1000
{

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

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

  main(Input Q)
  {
    Lefagy2(Q)
  }

}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

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

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

Tanulságok, tapasztalatok, magyarázat...

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/csere.c

Két változó értékének felcserélése segédváltozó nélkül a legegyszerűbben XOR-ral oldható meg:

```
Program csere.c {  
  
    int main(){  
  
        int x = 3;  
        int y = 5;  
  
        x ^= y; // x = 6  
        y ^= x; // y = 3  
        x ^= y; //x = 5  
  
        printf("x:%d y:%d\n", x, y);  
  
        return 0;  
    }  
}
```

Az XOR lényege, hogy a számok bináris alakjainak különbségét vizsgálja meg. Nézzük meg sorról sorra a kódot:

3 bináris alakja: 0011

5 bináris alakja: 0101

$x \oplus y$ esetén x értéke 0110 lesz, mert a 2 szám binárisan összehasonlítva, ha az adott biten a két szám értéke megegyezik, akkor 0-át kapunk, ha különbözőek, akkor pedig 1-et. Az eredményt az alábbi táblázat mutatja, ahol az eltérést pirossal, az egyezést zölddel jelöltem:

	BINÁRIS ALAK			
3	0	0	1	1
5	0	1	0	1
Összehasonlítás	0	1	1	0

2.1. ábra. EXOR működése

Ezután $y \hat{=} x$ esetén (ahol $y = 0101$ és $x = 0110$) y értéke 0011 lesz, ami a 3-as szám bináris alakjának felel meg. Ezután már csak a x értékét kell helyesen megadni, azaz $x \hat{=} y$ esetén (ahol $x = 0110$ és $y = 0011$) x értéke binárisan 0101 lesz, ezzel pedig meg is kapjuk az 5-ös számot, azaz y és x értékét sikeresen felcseréltük segédváltozó nélkül.

2.4. Labdapattogás

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

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

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/labdapattogas

Először nézzük meg a labdapattogatást if-ekkel. Az `initscr()` curses módba kapcsolja a terminált. Ez törli a képernyőt és fekete háttérrel ad, továbbá tárolja az ablak adataival kapcsolódó információkat. Először fontos, hogy az ablakkal és a lépésekkel kapcsolatos változókat meghatározása. Az `int x`-el és az `int y`-al adjuk meg az aktuális pozíciót, mely az x és az y tengelyen fog elhelyezkedni. Az `int xnov` és az `int ynov` lesznek a lépésközök, azaz hogy mennyit lépjen majd előre a labda az x és y tengelyen. Az `int mx`-ben és az `int my`-ben pedig az ablak méreteit fogjuk majd eltárolni.

Program `labdaif.c`

```
{  
  
    WINDOW *ablak;  
    ablak = initscr ();  
  
    int x = 0;  
    int y = 0;  
  
    int xnov = 1;  
    int ynov = 1;  
  
    int mx;  
    int my;  
    ...  
}
```

Mivel azt szeretnénk, hogy a program folyamatosan, megállás nélkül fusson, ezért végtelen ciklusra lesz szükségünk, amit a `for(;;)` ciklussal érhetünk el. Ezen belül kell program számára megadni, hogy pontosan mekkora lesz a képernyő. Ezt a `getmaxyx()` függvény segítségével érhetjük el, ahol az ablak-ban tárolt értékeket elmenti az `mx` és `my` változóba.

Megadhatjuk továbbá, hogy a képernyő ciklusonként törölje az előző "labda" nyomát a `clear()` függvénnyel. Ha ezt kihagyjuk, akkor a képernyőn folyamatosan látni fogjuk a labda eddigi helyzetét is, azaz folyamatosan "csíkot húz" maga után. Ennek a függvénynek mindig az `mvprintw()` előtt kell szerepelnie, ellenkező esetben csak üres ablakot fogunk látni. Ez utóbbi függvény segítségével adhatjuk meg, hogy az "O" karaktert, hol jelenítse meg a képernyőn. Majd az a következő pár sorban az `x` és `y` értékeket 1-el növeljük.

```
Program labdaif.c
{
    ...
    getmaxyx ( ablak, my , mx );

    clear ();
    mvprintw ( y, x, "O" );

    x = x + xnov;
    y = y + ynov;
    ...
}
```

Ezután következik annak meghatározása, hogy a labda elérte-e a képernyő szélét. Ha igen, akkor az értékeket meg kell szorozni -1-el és így a labda "visszapattog". Itt kell használnunk az `if()` függvényeket. Ha itt megnézzük az első sort, akkor láthatjuk, hogy ha az `x` értéke nagyobb mint az ablak `mx-1` értéke, akkor a labda útját, azaz az `xnov` értékét meg kell szorozni mínusz 1-gyel. Értelemszerűen ugyanígy kell meghatározni a többit is a hozzá tartozó változókkal.

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

Végül az `usleep()` függvénnyel adjuk meg a labda sebességét. Jelenleg 1 másodperc van megadva, ami átszámolva 100000 mikroszekundum. Ha ez letelik, a ciklus újraindul.

Ha a **labdapattogást if nélkül** szeretnénk megoldani, akkor maradékos osztást kell használni, ami az alábbi kódrészletben látható:

```
Program labda.c
{
    ...

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

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

        clear ();

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

        refresh ();
        usleep (100000);

    }

    ...
}
```

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. Használj ugyanazt a `while` ciklus fejet, amit Linus Torvalds a BogoMIPS rutinjában!

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

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/bogomips

A szóhosszt az alábbi programmal tudjuk kiírni:

```
Program szohossz.c
{
    int h = 0;
    int n = 0x01;
    do
        ++h;
```

```
while (n <= 1);  
printf ("A szóhossz ezen a gépen: %d bites\n", h);  
return 0;  
  
}
```

A program elindítása után a terminálban olvashatjuk, hogy a szóhoz ezen a gépen 32 bites.

A `0x01` kifejezés hexadecimális (16-os számrendszer) számot jelent, mely `0x` prefixszel kezdődik. A `do` és `while` páros pedig azt jelenti, hogy "csináld ezt, amíg a feltétel teljesül". Van még egy fontos eleme a programnak, ami a `while` utáni zárójelben szerepel, ez pedig a bittologató (vagy más néven shiftelő) operátor. Ennek a jele: `<<` vagy `>>` lehet. Jelen példában a `<<` operátort használjuk, mert a biteket balra szeretnénk tolni egyesével, egészen addig, amíg csupa nullát nem kapunk. A bitek léptetését így kell elképzelni:

Bitek léptetése

```
00000001  
00000010  
00000100  
00001000  
00010000  
00100000  
01000000  
10000000  
00000000
```

A program végül azért ír 32 bitet végeredményül, mert ennyiszer fut le a ciklus, amíg csupa nulla nem lesz.

A **Bogomips** egy Linux operációs rendszeren nyújtott mérési program, amely relatív módon jelzi, hogy a számítógép processzor milyen gyorsan fut, azaz a processzor hányszor halad át egy adott cikluson egy meghatározott idő alatt. A programot Linus Torvalds írta. Most ennek a programnak a működését fogom bemutatni.

A program elején látható, hogy először 2 változót deklaráltunk: az egyik a `loops_per_sec`, melynek értéke 1, a másik pedig a `ticks`, ami a `while` cikluson belül a `clock()` függvényt rendeljük hozzá. Ez a `clock()` függvény méri, hogy eddig mennyi processzor idő telt el. A `while` ciklus zárójelében a `loops_per_sec` értéke a `<=` shiftelő operátor miatt 2 hatványaival fog számolni. Ezután meghívjuk a `delay()` függvényt, aminek mindig átadom a ciklus változó értékét. Nagyon gyorsan növekedik, mindig hatványodik és ez a `delay()` függvény azt csinálja, hogy 0-tól egyesével növelgetve elszámol a `loops` értékig. Ezt követően újra lekértem a `ticks` értékét, mégpedig úgy, hogy a processzoridőből kivonjuk az előző időt. Lényegében ezzel kapjuk meg, hogy mennyi idő telt el.

Program `bogomips.c`

```
void delay (unsigned long long loops)  
{  
    for (unsigned long long i = 0; i < loops; i++);  
}  
  
...
```

```
while (loops_per_sec <= 1)
{
    ticks = clock ();
    delay (loops_per_sec);
    ticks = clock () - ticks;

    ...
}
```

Az if függvénnyel megvizsgáljuk, hogy a `ticks` értéke nagyobb vagy egyenlő-e a `CLOCKS_PER_SEC` értékénél. Ez utóbbi értéke mindig 1.000.000. Ha ez a feltétel teljesül, akkor kiszámoljuk, `loops_per_sec` értékét. Itt arra vagyunk kíváncsiak, hogy milyen ciklusérték tartozna hozzá, ha nem 2 hatványaival mennénk. Tehát, hogy milyen hosszú ciklust képes végrehajtani a gép. Kiíratásnál még elosztjuk az adatokat, hogy jobban olvasható legyen a kapott eredmény.

Program `bogomips.c`

```
...
    if (ticks >= CLOCKS_PER_SEC)
    {
        loops_per_sec = (loops_per_sec / ticks) * CLOCKS_PER_SEC;

        printf ("ok - %llu.%02llu BogoMIPS\n", loops_per_sec / ↵
            500000,
            (loops_per_sec / 5000) % 100);

        return 0;
    }
}
...
```

2.6. Helló, Google!

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

Megoldás videó:

Megoldás forrása:

A PageRank a Google internetes keresőmotor alapja, amit a Google alapítói, Larry Page és Sergey Brin fejlesztettek ki 1998-ban a Stanford Egyetemen. A PageRank a Google bejegyzett védjegye, legfontosabb tulajdonsága, hogy képes elemezni az oldalak közötti kapcsolatokat és ezáltal relevánsabb találatokat tud visszaadni. A forrásban látható program egy 4 honlapból álló hálózatra számolja ki a négy lap PageRank-ét.

Először is szükségünk lesz egy 4x4-es mátrixra, mivel 4 honlapunk van. Ezt az `L` változóban deklaráljuk. Látható, hogy oszloponként vannak kiszámolva az eredmények. Minden oldalnak egy egységnyi szavazata van, amit szétszét azok között az oldalak között, amikre hivatkozik. Tehát ha az `A` oszlopban csak 1

honlapra van hivatkozás, ezért oda 1-es kerül. A második oszlopban lévő honlap 2 oldalra is hivatkozik ezért ezt egyenlő arányban osztja szét, tehát mindkét hivatkozott oldal fél-fél szavazatot kap és ez így megy tovább a többi oszlopnál is. Ezután meghívjuk a `pagerank()` függvényt, aminek átadjuk az `L` változóban tárolt mátrixot.

Program `pagerank.c`

```
...

int main (void){
    double L[4][4] = {
        {0.0, 0.0, 1.0/3.0, 0.0},
        {1.0, 1.0/2.0, 1.0/3.0, 1.0},
        {0.0, 1.0/2.0, 0.0, 0.0},
        {0.0, 0.0, 1.0/3.0, 0.0}
    };

    pagerank(L);

    printf("\n");

    return 0;
}
```

A `pagerank()` függvényben a `PRv[]` tömbben lesznek eltárolva az `L` mátrixból kapott értékek. A `PR[]` tömbben pedig majd a számítás során kapott eredményeket. A `for` függvény négyszer fog lefutni, mert összesen ennyi honlapunk van, ezeket össze fogja adni, míg végül egy oszlopból álló 4 soros mátrixot kapunk, aminek eredménye bekerül a már említett `PR[]` tömbbe.

Program `pagerank.c`

```
...

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

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

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

        for (i=0; i<4; i++){
```



```
        PRv[i]=PR[i];
    }
}
kiir (PR, 4);
}

...
```

Ezután if függvénnyel megvizsgáljuk, hogy a távolság() függvény által adott eredmény kisebb-e 0.0000000001-val. Az ezt a függvényt az alábbi kódrészletben lehet látni:

Program pagerank.c

```
...

double
tavolsag (double PR[], double PRv[], int n){

    int i;
    double osszeg=0;

    for (i = 0; i < n; ++i)
        osszeg += (PRv[i] - PR[i]) * (PRv[i] - PR[i]);
    return sqrt(osszeg);
}

...
```

Ezután már csak kiíratjuk a PR[] tömbben tárolt eredményeket egyesével, amit a kiir() függvény segítségével tesszük meg.

Program pagerank.c

```
...

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

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

...
```

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Primek_R/stp.r

A számelmélet egyik legfontosabb tétele a Brun-tétel, mely a prímszámokkal kapcsolatos elméleteket gondolja tovább. Prímszámoknak nevezzük azokat a számokat, amelyeknek trivális osztói vannak, azaz csak önmagával és 1-el oszthatók. A Brun-tétel szerint végtelen sok olyan p prím létezik, amire $p+2$ is prím (pl.: 3,5; 5,7; 7,9; stb). Az ilyen prímpárokat, amiknek a különbsége 2, ikerprímeknek nevezzük. Az ilyen ikerprímsejtések bizonyítása vagy cáfolata jelenleg meghaladja a matematika eszközeit. Az ikerprímek, ha végtelen sokan vannak is, nagyon ríktán fordulnak elő a prímek között. Ugyanis míg a prímek reciprokaiból álló sor divergens, addig az ikerprímek reciproka sora konvergens. Konvergensnek nevezzük azokat a sorokat, ha az elemek tartanak egy számhoz. Azt a számot pedig ahova eljut, a sor összegének nevezzük. A Brun tétel szerint ezt a sor összeget az ikerprímszámok reciprokainak összeadásával kapjuk meg $[(1/3+1/5)+(1/5+1/7)+(1/7+1/9)+...]$ és megkapunk egy határértéket. Ezt a határértéket nevezzük Brun konstansnak. Most nézzük meg az alábbi programot, ami megpróbálja közelíteni a Brun konstans értékét:

```
Program stp.r
library(matlab)

stp <- function(x){

  primes = primes(x)
  diff = primes[2:length(primes)]-primes[1:length(primes)-1]
  idx = which(diff==2)
  t1primes = primes[idx]
  t2primes = primes[idx]+2
  rt1plust2 = 1/t1primes+1/t2primes
  return(sum(rt1plust2))
}

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

A kód futtatásához Matlabra lesz szükség. A `primes()` függvény egy paramétert kér, az x helyére bármilyen számot írhatunk és az ott megadott számig fogja kiszámolni a prímeket.

A `diff` a `primes` vektorban lévő számok segítségével az egymást követő számok különbségét kapjuk meg egy vektorba rendezve.

Az `idx` változóban megkeressük azokat a számokat, ahol a `diff` egyenlő 2-vel. Ugyanis ezek lesznek majd az ikerprím párok.

A `t1primes` kiveszi az ikerprímpárok első tagját, majd a `t2primes` változóban az ikerprím pár első tagjához hozzáadunk 2-őt, és így megkapjuk az ikerprím pár második tagját is. Majd ezen ikerpárok reciprokösszegét a `rt1plust2` változóban tároljuk. Végül pedig a `sum()` függvénnyel ezeket a törteket összeadjuk.

Ezután már csak meg kell jelenítenünk a kapott eredményeket egy függvény segítségével. A jelen példában egy `seq()` függvényt fogjuk használni. Az `y` változóban az `sapply()` függvény használatával rendeljük hozzá az `x` érték adatait. Végül pedig a `plot()` függvénnyel rajzoljuk ki az eredményt.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

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

A paradoxon egy az Egyesült Államokban az 1960-as években nagy sikerrel futott televíziós vetélkedő utolsó játékán alapszik. Nevét a show házigazdájáról kapta.

A probléma alaphelyzete a következő: A játékosnak mutatnak három zárt ajtót, melyek közül kettő mögött egy-egy kecske van, a harmadik pedig egy vadonatúj autót rejt. A hangulat fokozása érdekében a választást, illetve a felnyitást egy kicsit megbonyolították. A játékos kiválaszt egy ajtót, de mielőtt ezt kinyitná, a műsorvezető a másik két ajtó közül kinyit egyet, mely mögött biztos nem az autó van (természetesen a showman a kezdettől tisztában van azzal, hogy melyik ajtó rejt az autót). Ezt követően megkérdezi a már amúgy is ideges vendéget, hogy jól meggondolta-e a választását, vagyis nem akar-e váltani. A játékos dönt, hogy változtat, vagy sem, végül feltárul az így kiválasztott ajtó, mögötte a nyereménnyel. A paradoxon kérdése az, hogy érdemes-e változtatni, illetve van-e ennek egyáltalán jelentősége.

Vizsgáljuk meg a Monty Hall paradoxont: kezdetben tehát 1:3 az esélye, hogy bármelyik ajtó mögött ott lehet a főnyeremény, az autó. Ezután a műsorvezető kinyit egy olyan ajtót, ami mögött nincs ott az autó. Ekkor még mindig 1:3 lesz az esélye annak, hogy az általunk választott ajtó mögöttrejtőzik a kocs, és 2:3 hogy mégse ott. Így tehát 2:3 lesz annak is az esélye, hogy a másik ajtó mögött találjuk meg a főnyereményt, ezért jó, ha váltunk. Az alábbi táblázat bemutatja a nyerési esélyeinket:

1. ajtó	2. ajtó	3. ajtó	Monty kinyitja	Ha váltunk
kecske	kecske	autó	a 2. ajtót	nyerünk
kecske	autó	kecske	a 3. ajtót	nyerünk
autó	kecske	kecske	a 2. vagy a 3. ajtót	vesztünk

2.2. ábra. Monty Hall probléma

Láthatjuk, hogy ha mindig az első ajtót választjuk, és változtatunk a döntésünkön, akkor a három esetből kétszer nyerünk.

Most pedig nézzünk meg egy ezzel kapcsolatos R szimulációt. Először is megadjuk, hogy mennyi legyen a kísérletek száma (1000000). A kísérlet változóban deklaráljuk, hogy `sample()` függvénnyel, hogy

1-től 3-ig, generáljon random számokat, mégpedig annyiszor, ahány a `kiserletek_szama` változó értéke. A `replace=T` engedélyezi, hogy legyen ismétlődés a számok között. Ide kerül tehát, majd az, hogy hol lesz a nyeremény. ezután megint ugyanezt a kódrészletet adjuk meg a `jatekos` változónak is, mely a játékos választásait fogja tárolni. Majd a `musorvezeto` értékéhez azt adjuk meg, hogy mennyi a `kiserletek_szama` mérete.

Program mh.r

```
kiserletek_szama=10000000
kiserlet = sample(1:3, kiserletek_szama, replace=T)
jatekos = sample(1:3, kiserletek_szama, replace=T)
musorvezeto=vector(length = kiserletek_szama)
...
```

A következő `for` ciklus annyiszor fog lefutni, ahány a `kiserletek_szama`. Azon belül, ha a `kiserlet` adott eleme megegyezik a `jatekos` adott elemével, azaz eltalálta a helyes ajtót, akkor a `kiserlet` tömbből kivesszük az egyezést és így lesz elmentve az adat a `mibol` változóban. Ezt a `setdiff()` függvénnyel érhetjük el. Ellenkező esetben (`else`) mind a `kiserlet[]` és a `jatekos[]` tömbből ki kell venni az adott értéket. Ezután állítjuk össze a `musorvezeto[]` értékeit a `mibol` és a `sample()` függvény segítségével.

Program mh.r

```
...

for (i in 1:kiserletek_szama) {

  if(kiserlet[i]==jatekos[i]){

    mibol=setdiff(c(1,2,3), kiserlet[i])

  }else{

    mibol=setdiff(c(1,2,3), c(kiserlet[i], jatekos[i]))

  }

  musorvezeto[i] = mibol[sample(1:length(mibol),1)]

}

...
```

A `nemvaltoztatasesnyer` tömbbe kerülnek azok az adatok, hogy a `kiserlet` és a `jatekos` értékeit összehasonlítva hol találhatók egyezések. Ehhez a `which()` függvényt használjuk. A `valtoztat` vektorban megadjuk, hogy a változó értéke azonos legyen a `kiserletek_szama`-val.

Program mh.r

```
...

nemvaltoztatasesnyer= which(kiserlet==jatekos)
valtoztat=vector(length = kiserletek_szama)
```

...

Ezután megint for ciklus jön, megint ugyanannyiszor, ahány a `kiserletek_szama`. Dekráljuk a `holvalt` tömböt, mégpedig úgy, hogy a `setdiff()` függvénnyel kivesszük azokat az értékeket, amik egyeznek az adott indexben a `musorvezeto` és a `jatekos` értékeivel.

Program `mh.r`

...

```
valtoztatesnyer = which(kiserlet==valtoztat)
```

```
sprintf("Kiserletek szama: %i", kiserletek_szama)
```

```
length(nemvaltoztatesnyer)
```

```
length(valtoztatesnyer)
```

```
length(nemvaltoztatesnyer)/length(valtoztatesnyer)
```

```
length(nemvaltoztatesnyer)+length(valtoztatesnyer)
```

Ha ezzel megvagyunk, akkor a `valtoztatesnyer` változót ugyanúgy adjuk meg, ahogy eddig, majd kiíratjuk az adatokat.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfiával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/Chomsky/convert.c

A decimális számrendszer, vagy más néven tízes számrendszer egy 10 elemből álló halmaz, melynek első tagja a 0, az utolsó tagja pedig a 9. Ha ezt át szeretnénk váltani unáris, azaz egyes számrendszerbe, akkor a végeredményként egy csupa 1-esekből álló értéket kapunk. Az 1 csupán szimbólum, lehet helyettesíteni bármilyen más szimbólummal, a lényeg, hogy az N számot az általunk választott szimbólum N-szeri ismétlésével ábrázoljuk.

Ha egy decimálisból unárisba átváltó programot szeretnénk írni, akkor egy olyan ciklust kell létrehozni, ami mindig annyiszor írja ki az egyest, ahány számot adtunk meg. Ha az adott szám nulla, akkor eredményként nem fogunk kapni semmit, mivel a nullát nem képes egyes számrendszerben ábrázolni.

```
#include <stdio.h>

int main()
{
    int a;
    int count = 0;

    printf("Adj meg egy természetes számot\n");
    scanf("%d", &a);
    printf("A beírt szám unáris alakban:");

    for (int i = 0; i < a; ++i) {
        printf("1");
        count++;
    }

    printf("\n");
    return 0;
}
```

```
}
```

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

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

Megoldás videó:

Megoldás forrása:

A Noam Chomsky nevéhez fűződő úgynevezett Chomsky-féle hierarchia 1-es típusa definiálja a környezetfüggő nyelvtanokat. Kétféle szabály létezik:

- $\alpha A \gamma \rightarrow \alpha \beta \gamma$ alakú,
ahol $A \in N$, $\alpha, \gamma \in (NUT)^*$, $\beta \in (NUT)^+$. Tehát itt az A nemterminális, α pedig egy nemterminálisokból és terminálisokból álló üres vagy nem üres szó. A jobb oldalon álló β és γ viszont már nemterminális és terminálisokból álló szavak.
- $S \rightarrow \varepsilon$ alakú,
ahol az S kezdőszimbólum nem szerepel egyetlen szabály jobb oldalán sem.

Egy példa: Legyen $G = (\{S, A, B, C\}, \{a, b, c\}, S, \{S \rightarrow \lambda, S \rightarrow abc, S \rightarrow aABC, A \rightarrow aABC, A \rightarrow aBC, CB \rightarrow BC, aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc\})$.

Ekkor bizonyítható, hogy G az $L(G) = \{a^n b^n c^n \mid n \geq 0\}$ nyelvet generálja.

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/Chomsky/szabvany.

A BNF (Backus-Naur Form) egy olyan metanyelv, melyek segítségével szabályok alkothatók meg. Legfőbb célja a programozási nyelvek szintaxisának leírása. Főbb részei:

- $\langle \text{név} \rangle$

Olyan fogalmak tartoznak ide, mint pl.: azonosító, betű, törtszám, prízszám, stb. Tehát bármi lehet. Nem terminális elem.

- $::=$

Ez fogja elválasztani a szabály bal- és jobb oldalát.

- A definiálandó nyelv karakterkészlete, azaz a terminálisok. Az elválasztó jel jobb oldalán helyezkedik el.

Ezek alapján néhány C utasítások BNF-ben:

```
//feltételes utasítás
if (<kifejezés>)
    <utasítás>
else if (<kifejezés>)
    <utasítás>
else (<kifejezés>)
    <utasítás>

//while utasítás
while (<kifejezés>)
    do <utasítás>
```

Most pedig jöjjön egy olyan kódcsipet, amely c89-cel nem, de c99-el működik:

```
int main()
{
    //comment
    return 0;
}
```

A fenti kódot ha c89-cel fordítjuk le, akkor az alábbi hibaüzenetet fogunk kapni:

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

szabvany.c: In function 'main':
szabvany.c:3:5: error: C++ style comments are not allowed in ISO C90  ↵
    //comment
    ^
szabvany.c:3:5: error: (this will be reported only once per input file)  ↵
```

Ha pedig c99-cel, akkor gond nélkül működik:

```
gcc -std=c99 szabvany.c -o szabvany
```

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/Chomsky/lexer.l

Az alábbi kódot .l végződésű fájlba kell menteni, mert a lex programmal fogunk dolgozni. A program segítségével lexikális szabályokból lexikális elemző programkódot fogunk generálni. Ehhez a következő parancsot kell megadnunk: `lex -o lexer.c lexer.l`, ami elkészíti számunka a c forráskódot. Ezután már csak a szokásos módon gcc-vel fordítjuk le, azzal a különbséggel, hogy a végéhez beírjuk az `-lfl` is: `gcc lexer.c -o lexer -lfl`.

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

int realnumbers = 0;
}%
digit [0-9]
%%
{digit}* (\.{digit}+)? {++realnumbers;
    printf("[realnum=%s %f]", yytext, atof(yytext));}
%%
int
main ()
{
    yylex ();
    printf("The number of real numbers is %d\n", realnumbers);
    return 0;
}
```

A fenti program 3 fő részből áll, amelyeket a `%%` szimbólum választja szét őket. Az első rész tartalmazza a headert és a változót. Itt bármilyen C kódot megadhatunk a `%{` és `%}` jelek között. Ezt a Flex módosítás nélkül fogja átmásolni a generálandó kódba.

A második rész 2 részből álló szabályokat tartalmaz: Először a reguláris kifejezésekkel kezdjük, utána jön a kapcsos zárójelbe zárva a C kód. A reguláris kifejezéseknél határozzuk meg a keresési szabályokat. Jelen esetben számmal kezdődöt keresünk (`digit`), ami akár többször is előfordulhat (`*`). Ezt követően van egy zárójeles rész is, ami azt jelenti, hogy `????`. Az ezt követő C kódban adjuk meg, hogy majd a terminálba bevitt szöveget vizsgálja meg. Az `atof()` függvény segítségével a `string`-ből `double` lesz. Tehát ezekből a szabályokból létrejön majd a `yylex` nevű függvény és `return` esetén újra ide fog visszakerülni.

A program harmadik részében látható egy `yylex()` függvény, ami a fenti szabályokat hívja elő, továbbá kiírjuk az eredményt.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/Chomsky/leet.l

A leet nyelv lényege, hogy bizonyos betűket számokkal vagy ASCII karakterekkel szokták helyettesíteni. Innen jön az, hogy leet helyett l33t írunk. A mostani példában olyan programot mutatok be, ami az adott szövegben ezeket a betűket kicseréli a megfelelő számokkal, vagy karakterekkel. A megadott forráskódot

is lex programmal kell C-be fordítani. Vizsgáljuk meg a programot. Ahogy az előző feladatnál, úgy itt is 3 részből áll.

Az első részben definiáljuk a L33SIZE konstansszerű elemet. Ezt a #define segítségével oldjuk meg és a nevet mindig csupa nagybetűvel kell megadni. Ha a programban valahol beírjuk a megadott konstans nevet (L33SIZE), akkor a helyére fogja majd behelyettesíteni a hozzá tartozó szöveget. Jelen esetben ehhez a változóhoz meg van adva egy char c és egy hozzá tartozó char *leet[4] tömb, ebből a kettőből fog összeállni a l337d1c7[] tömb, ahol az előbbi lesz majd a kicserélendő betű, utóbbi pedig az, hogy mire lehet majd kicserélni (4 választási lehetőség van).

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

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

    {'a', {"4", "4", "@", "/-\\\"}},
    ...
}
```

Ha az egyes betűkhöz hozzárendeltük a megfelelő számokat vagy karaktereket. Akkor a második részben fogjuk megadni C-ben, hogy hogyan történjen a karaktercsere:

```
...
. {

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

        if(l337d1c7[i].c == tolower(*yytext)) //Megegyezik-e a két karakter?
        {

            int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0)); //random szám ←
                generálása 1 és 100 között

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

            found = 1;
            break;
        }
    }
```

```
    }

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

    }
    ...
```

A kapcsos zárójel elején egy pont van, ami azt jelenti, hogy minden egyes karaktert vizsgáljon meg a szövegben. Ez lesz tehát a reguláris kifejezés. A C részben látható, hogy a `for` ciklusban előhívjuk a `L33SIZE` nevű konstanst, tehát ide fogja behellyettesíteni azt, amit már az első részben megadtunk.

Szükségünk lesz a `tolower()` függvényre is, mert ha nagybetű van a szövegben, akkor azt nem fogja számításba venni a program, ezért ezzel a függvénnyel minden betűt kisbetűvé alakítunk át. Ezután a kapott random szám alapján cseréljük ki az adott betűket. Tehát például az "a" betűt kell kicserélni, és `r` értéke 97, akkor ez azt jelenti, hogy az "a" betű "@"-ra lesz kicserélve, mivel az if-else utasításban itt fog teljesülni:

```
...
    else if(r<98)
        printf("%s", l337d1c7[i].leet[2]);
    ...
```

Ez azt jelenti, hogy `l337d1c7[i]` tömbön belül a `leet[2]` tömb 3. eleme legyen kiválasztva, ami a "@". Előfordul azonban, hogy a két karakter nem egyezik meg a vizsgálat során (`if(!found)`), ezért ilyenkor az eredeti karaktert írjuk ki.

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 meggyránézésre, elkapja valamelyiket esetleg a splint vagy a frama?

Mielőtt átnéznénk a kódcsipeteket, előbb fontosnak tartom megemlíteni, hogy a kódokat úgy vizsgáljuk meg, hogy feltételezzük, az adott változók, függvények értéke már előzetesen deklarálva vannak.

i.

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

Ez a kódcsipet az előző példa ellentettje. Azt jelenti, hogy ha a SIGINT jel kezelése nem volt figyelmen kívül hagyva, akkor ezután se legyen figyelmen kívül hagyva.

ii.

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

A for ciklus 5-ször hajtódik végre. Az *i* változó értéke 0-tól 4-ig fog változni. A *++i* azt jelenti, hogy *i* értékét előbb növeljük és ez lesz az új érték.

iii.

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

Ez a ciklus ugyanúgy fog végrehatjtódni, mint az előző. Itt is ugyanúgy 0-tól 4-ig fog számolni. Az *i++* pedig azt jelenti, hogy *i* értéke marad a régi, majd utána növeli 1-el.

iv.

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

A for ciklus kicseréli a tömb elemeit. A tömb első eleme marad a régi, az azt követő elemek értéke azonban már rendre 0, 1, 2, 3 lesz. Ha a tömb mérete 5-nél nagyobb, akkor a többi értéket figyelmen kívül hagyja, mivel a for ciklus csak 5-ször fut le. Ezért tehát a program bár gond nélkül le fog futni, de bugos.

v.

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

A for ciklusnak itt 2 felétele van, ugyanis látható, hogy van egy && (ÉS) logikai operátor is. A ciklus addig fog futni, amíg *i* értéke kisebb mint *n*. A logikai operátor jobb oldalán nincs relációs operátor megadva, ezért ez a program bugos. A változók előtti * jel azt jelenti, hogy egy adott tárterületre mutatnak a változók értékei.

vi.

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

Az printf() függvény segítségével íratjuk ki a zárójelben megadott paraméterek alapján. Jelen esetben a cél az lenne, hogy kiíratjuk az f() függvények visszatérési értékét, de ez a kód bugos, ugyanis az f() függvény két paramétert kér, de ezek kiértékelési sorrendje nincs meghatározva.

vii.

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

Ez a kód már helyes. Az f() függvény egy paramétert kér, ami jelen esetben az "a" lesz, így ennek az értéke kerül kiíratásra.

viii.

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

Ez a kód is le fog fordulni, csak itt az f() függvénynél &a argumentum van megadva, ami azt jelenti, hogy egy terület memóriacímére mutató értéket ad paraméterül.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

```

$$\$(\text{\texttt{\textbackslash forall}}\ x\ \text{\texttt{\textbackslash exists}}\ y\ ((x < y) \text{\texttt{\textbackslash wedge}} (y\ \text{\texttt{\textbackslash text}}\{ \text{prím}})))\$$$

```

```

$$\$(\text{\texttt{\textbackslash forall}}\ x\ \text{\texttt{\textbackslash exists}}\ y\ ((x < y) \text{\texttt{\textbackslash wedge}} (y\ \text{\texttt{\textbackslash text}}\{ \text{prím}})) \text{\texttt{\textbackslash wedge}} (S y\ \text{\texttt{\textbackslash text}}\{ \text{prím}})) \leftrightarrow )\$$$

```

```

$$\$(\text{\texttt{\textbackslash exists}}\ y\ \text{\texttt{\textbackslash forall}}\ x\ (x\ \text{\texttt{\textbackslash text}}\{ \text{prím}}))\ \text{\texttt{\textbackslash supset}}\ (x < y))\ \$$$

```

```

$$\$(\text{\texttt{\textbackslash exists}}\ y\ \text{\texttt{\textbackslash forall}}\ x\ (y < x)\ \text{\texttt{\textbackslash supset}}\ \text{\texttt{\textbackslash neg}}\ (x\ \text{\texttt{\textbackslash text}}\{ \text{prím}})))\ \$$$

```

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

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

Végtelen sok prím szám van.

```

$$\$(\text{\texttt{\textbackslash forall}}\ x\ \text{\texttt{\textbackslash exists}}\ y\ ((x < y) \text{\texttt{\textbackslash wedge}} (y\ \text{\texttt{\textbackslash text}}\{ \text{prím}})))\ \$$$

```

Végtelen sok ikerprím szám van.

```

$$\$(\text{\texttt{\textbackslash forall}}\ x\ \text{\texttt{\textbackslash exists}}\ y\ ((x < y) \text{\texttt{\textbackslash wedge}} (y\ \text{\texttt{\textbackslash text}}\{ \text{prím}})) \text{\texttt{\textbackslash wedge}} (S y\ \text{\texttt{\textbackslash text}}\{ \text{prím}})))\ \$$$

```

Véges sok prím szám van.

```

$$\$(\text{\texttt{\textbackslash exists}}\ y\ \text{\texttt{\textbackslash forall}}\ x\ (x\ \text{\texttt{\textbackslash text}}\{ \text{prím}}))\ \text{\texttt{\textbackslash supset}}\ (x < y))\ \$$$

```

Véges sok prím szám van.

```

$$\$(\text{\texttt{\textbackslash exists}}\ y\ \text{\texttt{\textbackslash forall}}\ x\ (y < x)\ \text{\texttt{\textbackslash supset}}\ \text{\texttt{\textbackslash neg}}\ (x\ \text{\texttt{\textbackslash text}}\{ \text{prím}})))\ \$$$

```

3.8. Deklaráció

Megoldás videó:

Megoldás forrása:

- egész

```
int egesz = 5;
```

- egészre mutató mutató

```
*mutato = &eges;
```

- egész referenciája

```
int &r_egesz = egesz;
```

- egészek tömbje

```
int tomb[3] = {1,2,3};
```

- egészek tömbjének referenciája (nem az első elemé)

```
int (&r_tomb)[3] = tomb;
```

- egészre mutató mutatók tömbje

```
int *m_tomb[3];
```

- egészre mutató mutatót visszaadó függvény

```
int* e_mutato = f(mutato);
```

- egészre mutató mutatót visszaadó függvényre mutató mutató

```
int* (*f_mutato)(int*) = f;
```

- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény

```
int (*getInt (int a)) (int b, int c);
```

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

Mit vezetnek be a programba a következő nevek?

- ```
int a;
```

Integer, azaz egész típusú változót, melynek változóneve: a

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

Az első sorban bevezetett a változóra mutató mutató lesz a *b.

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

Az a változó referenciája lesz az &r

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

Integer típusú, c változónevű tömb deklarálása, melynek mérete 5.

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

A c tömb referenciája.

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

Ez egy integer típusú tömbre mutató mutató.

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

Integer típusú h függvényre mutató mutató.

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

Egy egészre mutató mutatóval visszatérő függvényre mutató mutató.

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

Egy egészszel visszatérő, 2 egész paramétert váró függvényre mutató mutató.

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

függénymutató, mely egy egészet visszaadó, két egész paramétert váró függvényre mutató mutatót visszaadó, egészet kapó függvény.

4. fejezet

Helló, Caesar!

4.1. `int ***` háromszögmátrix

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

Megoldás videó:

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

Háromszögmátrixnak nevezünk egy négyzetes mátrixot, melynek főátlója alatt vagy felett minden elem nulla. Ez alapján tehát két típusa van: alsó háromszögmátrix és felső háromszögmátrix.

Az `int nr = 5` változóban deklaráljuk a sorok számát. A `print()` függvénnyel kiíratjuk a `tm`-nek lefoglalt memória címét. Itt a `&` szimbólum azt jelenti, hogy valaminek a memóriacímére mutat. Az `if()` függvényen belül a `malloc` visszaad nekünk egy pointert, ami a lefoglalt memóriára mutat. Pontosabban a `malloc` egy `void` típusú mutatót visszaadó függvény, ami csak egy paramétert vár, mégpedig a lefoglalandó tárterület mennyiségét bájtokban. A függvény használatához tudnunk kell az egyes adattípusok méretét, ezért kell használnunk a `sizeof()` operátort, amely az adott adattípus bájtokban megadott méretével tér vissza. Ennek az operátornak a paramétere lehet egy változó, egy tömb, egy kifejezés vagy egy adattípus. Mi most az utóbbit adjuk meg, azaz legyen `double` típusú, ami mondjuk legyen 8 bájt, amit megszorunk `nr` értékével, mivel nekünk 5 sorunk van. Ha az `if` függvényben lévő feltétel egyenlő `NULL`, akkor hiba van, ezért kilép és -1-et ad vissza. Ezután kiíratom, amit a `malloc()` függvény a `tm`-re visszaad.

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

printf("%p\n", &tm); //a tm memóriacím kiíratása. A & szimbólum azt jelenti ←
, hogy a memóriacímre mutat.

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

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



```
...
```

Ezután egy for ciklus következik, ami 5-ször fog lefutni. Az if() függvényen belül a malloc() visszaad egy pointert a lefoglalt memóriákra. Minden sorban egyre több 8 bájtnyi memóriát foglal le ((i + 1) * sizeof (double)). Itt is ugyanúgy ellenőrizzük, hogy teljesül-e a feltétel, mint a fenti kódrészletben.

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

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

Végül kiíratjuk a tm tömb adatait egyenként, úgy, hogy alsó háromszögmátrixot kapjunk. Ezt úgy érhetjük el, hogy a mátrix minden elemét ezzel a képlettel számoljuk ki: $tm[i][j] = i * (i + 1) / 2 + j$;

```
...
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");
}
...
```

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/caesar/EXOR/-exor.c

Ebben a feladatban XOR - magyarul: kizáró vagy - titkosítással foglalkozunk. Ennek az egyszerű titkosítási eljárásnak a lényege, hogy a szöveget egy kulcs segítségével titkosíthatjuk és visszafelé pedig ugyanezzel

a kulccsal tudjuk dekódolni a titkosított adatot. A maximális biztonság eléréséhez, amit egy külső támadó sem tud feltörni, az alábbi feltételeket kell figyelembe venni:

- A kulcs ugyanolyan hosszú legyen, mint a titkosítandó szöveg. Ellenkező esetben, ha a kulcs rövidebb, akkor az ismétlődni fog a kimeneti bitsorozatban, amiből már kinyirehető egy szövegrészlet.
- A kulcs véletlenszerűen generált legyen, ne használjuk ugyanazt a kulcsot minden titkosítás során.

A forrásban látható program is egy XOR titkosítási eljárást használ. Első lépésként a `#define` előfeldolgozó utasítás használatával definiálunk két változónevet, amiket csupa nagybetűvel adunk meg, majd ezekhez konstans értékeket rendelünk hozzá. Ezeket a változóneveket bárhol használhatjuk a programban, ahol az előfeldolgozó az ahhoz megadott értékeket helyezi majd el a fordítási folyamat előtt.

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

Ezután a szokásos `main()` függvényt kiegészítjük két argumentummal. Erre azért van szükség, mert amikor a futtatási rendszer meghívja a `main()` függvényt, akkor az alábbi két paramétert fogja átadni:

- `argc` (argument count): A parancssorban kapott egész számot adja meg.
- `**argv` (argument vector): Lényegében egy tömb, amely a karakterláncok mutatóit tartalmazza.

Ezt a kódrészletet úgy kell elképzelni a gyakorlatban, hogy a program kérni fog tőlünk adatokat, amiket be kell majd írunk. Jelen esetben a kulcsot és a szöveget.

Az `main()` függvényen belül két `char` típusú tömböket deklarálunk (`kulcs[]`, `buffer[]`), amelyek méretét a már fent említett `#define` konstans változónevek használatával adjuk meg. Ezt követően két `integer` típusú változót is megadunk (`kulcs_index`, `olvasott_bajtok`), melyek értéke mindkét esetben 0 lesz. Az előbbi a `kulcs[]` tömb indexét fogja majd megadni. Deklarálunk még egy `kulcs_meret` változót is, ami az `argv[]` tömb második paraméterének karakterlánc hosszát fogja kiszámolni az `strlen()` függvény segítségével. Látható még egy `strncpy()` függvény is, ami azt fogja csinálni, hogy egyik helyről a másikba másolja át az adatokat és megadhatjuk a a karaktermásolás számát is. Jelen esetben a kulcs tömbbe másoljuk az `argv[1]` adatait és a másolás mérete `MAX_KULCS` konstans értéke lesz.

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

    int kulcs_index = 0;
    int olvasott_bajtok = 0;

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

A változók deklarálása után szükségünk van egy `while` ciklusra, ami addig fog futni, amíg a byte-ok olvasása tart. A `read()` függvény 3 paramétert kér:

- 0: A 0 szám itt azt jelenti, hogy a standard inputról fogja beolvasni az adatot. Ez a rész mindig egy int típusú fájlleíró lesz. 3 típusa van: a már megadott 0, 1 (standard output), 2 (standard error).
- (void*) buffer: A beolvasott adatok a buffer tömbbe lesznek eltárolva.
- BUFFER_MERET: Az olvasandó byte-ok száma.

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

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

A while cikluson belül egy for ciklus lesz, ami addig fog futni, amíg el nem éri az olvasott byte-ok számát. Az adott buffer[i] karakter ki lesz cserélve az adott kulccsal exor segítségével (^). Ezután a kulcs_index értékét megnöveljük 1-el és ez addig fog futni, amíg tart a ciklus. A for cikluson kívül ezután előhívjuk a write() függvényt, mely hasonló a read() függvényhez, csak itt a standard outputra (1) írjuk ki a buffer-ben eltárolt adatokat és az írás mérete az olvasott_bajtok lesz.

A program működését az alábbi képen lehet látni:

```
dave@dave-K501UB:~/gyakorlas$ more tiszta.txt
A titkosítás vagy rejtjelezés a kriptográfiának az az eljárása, amellyel az információt (nyílt szöveg) egy algoritmus
(titkosító eljárás) segítségével olyan szöveggé alakítjuk, ami olvashatatlan olyan ember számára, aki nem rendelkezi
k az olvasáshoz szükséges speciális tudással, amit általában kulcsnak nevezünk. Az eredmény a titkosított információ
(titkosított szöveg). Sok titkosító eljárás egy az egyben (vagy egyszerű átalakítással) használható megfejtésre is, a
zaz, hogy a titkosított szöveget újra olvashatóvá alakítsa.
dave@dave-K501UB:~/gyakorlas$ gcc exor.c -o exor -std=c99
dave@dave-K501UB:~/gyakorlas$ ./exor 00012345 <tiszta.txt >titkos.txt
dave@dave-K501UB:~/gyakorlas$ more titkos.txt
qFbXFX[F00D000QWIV^AZU\TH0F000ZDA_WB00U]00QZBN00J00^Y0B00BS000U\]KVX00J00\U[G]00R[00A000H00XA00J00EQR000VK
00YW_BXF^AF000XFX[F0
0D0000YZ00C00G000UV000F00W00EQY00\HS]00J00GHTS000]SX00DZEZ000XY00]DRG]QDQE^RZ00\IP\00XRUB00I00]00CS000[Y00W^00U^TT^X
QOY[00H00YFQC00@Z
J00K00_F00WTA00EUSY00_]F00EU00GFQ\000^]A000]FRX00RQ_00AYSC^PY00PFUJ00_]000J00AQQ]00_K000BYDZ]@00D_DE00ZS_B]00P]0000[
G_ZC00E]G000J00DV
S0000^Y00\DI_B000000]X00G00C00TM00J00UJVP^00GSTM00WIBHV0000FRXT[00E00GFQ\000ZRG0^00]ZR0000TUUQ_D00B@V00C000HRN000_
VK000BYDZ]@00D_DE00N00F
UVWG000ZBP00XCQCXPF00C000^R_00DCP000
dave@dave-K501UB:~/gyakorlas$
```

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/caesar/EXOR-javaexor.java

Javában először mindig egy osztályt kell létrehozni, ami jelen esetben az ExorTitkosító lesz. Ezen belül a `public ExorTitkosító()` zárójelében deklaráljuk a `kulcsSzöveg`-et, ami egy `String` típusú változó lesz, továbbá java függvényeket hívunk elő, mint ahogy C-nél tettük `include` használatával, csak itt mindig `java.io.`-val kezdődik és a függvény nevével fejeződik be. Mivel ezek általában hosszúak, a függvény után szóközzel elválasztva adhatunk neki, egy számunkra sokkal olvashatóbb nevet, mint ahogy a példában a `bejövőCsatorna` és a `kimenőCsatorna` neveket adtuk meg.

```
public class ExorTitkosító {

    public ExorTitkosító(String kulcsSzöveg,
        java.io.InputStream bejövőCsatorna,
        java.io.OutputStream kimenőCsatorna)
        throws java.io.IOException {

        byte [] kulcs = kulcsSzöveg.getBytes();
        byte [] buffer = new byte[256];
        int kulcsIndex = 0;
        int olvasottBájtok = 0;

        while((olvasottBájtok =
            bejövőCsatorna.read(buffer)) != -1) {

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

                buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]);
                kulcsIndex = (kulcsIndex+1) % kulcs.length;

            }

            kimenőCsatorna.write(buffer, 0, olvasottBájtok);

        }

    }

    ...
}
```

A változók deklarálása után a `while()` ciklust hozunk létre, ami addig fog futni, amennyi a buffer mérete. A `for` ciklusban pedig megtörténik a már C változatban ismertetett EXOR-ozás. Itt is karakterenként cseréljük ki az adott kulccsal a szöveg tartalmát. Eddig ez a program önmagában nem fog csinálni semmit, mert ez csak egy segédfüggvény volt. Szükség van egy `public static void main()` függvényre is, ami a C program `int main()` függvényre hasonlít.

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

    try {
        new ExorTitkosító(args[0], System.in, System.out);
    }
}
```

```
    } catch (java.io.IOException e) {  
  
        e.printStackTrace();  
  
    }  
  
}
```

Itt a `try` és `catch` páros azt jelenti, hogy vizsgáljuk meg, hogy az `ExorTitkosító()` function-be bevitt adatok helyesen lettek-e megadva. Ha igen, akkor hajtsa végre a feladatot, ellenkező esetben jön a `catch()` függvény, hogy kiírassuk a hibaüzenetet.

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/caesar/EXOR-tores.c

A forrásban található programnak az a célja, hogy az első feladatban titkosított szöveget addig próbáljuk a program által generált kulcsokkal feltörni, amíg tiszta szöveget nem kapunk. Jelen programot leszűkítettük, 8 számjegyjű kulcsokra és minden egyes számjegy 0-tól 9-ig terjed. Erre azért van szükség, mert ha az összes lehetséges karaktert és méretet megvizsgáljuk akkor nagyon sokáig tartana, mire a program feltöri a titkos szöveget. A generált kulcsokat a `kulcs[]` tömbben tároljuk el.

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

```
        printf
        ("Kulcs: [%%%%%%%%%%%%%%%%%%]\nTiszta szoveg: ↵
         [%s]\n",
         ii, ji, ki, li, mi, ni, oi, pi, titkos);

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

    ...
```

Ha egy kulcsot legenerált a program, akkor `if()` függvénnyel megvizsgáljuk, hogy működik-e az adott kulcs. Látható, hogy itt az `exor_tores()` függvényt hívjuk elő, ami 4 paramétert kér. Ha a kulcs helyes, akkor kiíratjuk az eredményt, ellenkező esetben újra EXOR-ozunk. Az `exor_tores()`-en belül 2 függvény látható. Az egyik az `exor()` függvény, ahol ténylegesen történik az exor-ozás, a másik pedig a `tiszta_lehet()` ez fogja majd az értéket visszaadni a `return` függvénnyel.

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

    int kulcs_index = 0;

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

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

    }
}

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

    exor (kulcs, kulcs_meret, titkos, titkos_meret);

    return tiszta_lehet (titkos, titkos_meret);

}

...
```

A `tiszta_lehet()` függvényen belül van egy `double` típusú szohossz változó, ami a megszámlolt titkos szavak méretének átlagát tartalmazza. Ezalatt a `return` függvénnyel megvizsgáljuk, hogy a szohossz mérete

nagyobb-e, mint 6 és kisebb mint 9 továbbá, `strcasestr()` függvénnyel megvizsgáljuk, hogy tartalmazza-e az adott szavakat ("hogya", "nem", "az", "ha"). Azért pont ezeket a szavakat, mert ezek a leggyakoribb magyar szavak, amik előfordulhatnak. Ezeket akár bővíthatjuk további szavakkal is. Ha ezek teljesülnek, akkor nagy valószínűséggel a generált kulcs helyes, így a `return true` értéket ad vissza és megtörténik a `main()` függvényben a kiíratás.

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

    return (double) titkos_meret / sz;
}

int
tisztas_lehet (const char *titkos, int titkos_meret)
{
    // a tiszta szoveg valszeg tartalmazza a gyakori magyar szavakat
    // illetve az atlagos szohossz vizsgálatával csökkentjük a
    // potenciális töréseket

    double szohossz = atlagos_szohossz (titkos, titkos_meret);

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

Ha a program helyesen működik, akkor a terminálban az alábbi kellene kapnunk a program elindítása után:

```
sar/EXOR$ ./tores <titkos.txt
Kulcs: [00012345]
Tiszta szoveg: [A titkosítás vagy rejtjelezés a kriptográfiának az az eljárása,
amellyel az információt (nyílt szöveg) egy algoritmus (titkosító eljárás) segíts
égével olyan szöveggé alakítjuk, ami olvashatatlan olyan ember számára, aki nem
rendelkezik az olvasáshoz szükséges speciális tudással, amit általában kulcsnak
nevezünk. Az eredmény a titkosított információ (titkosított szöveg). Sok titkosí
tó eljárás egy az egyben (vagy egyszerű átalakítással) használható megfejtésre i
s, azaz, hogy a titkosított szöveget újra olvashatóvá alakítsa.
]
```

4.5. Neurális OR, AND és EXOR kapu

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

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/caesar/neuralis.c

Ebben a feladatban a gépet tanítjuk meg arra, hogy az általunk megadott adatokból végezzen számítást, úgy hogy a végeredmény ugyanannyi legyen. A program elindításához R szimulációra lesz szükségünk.

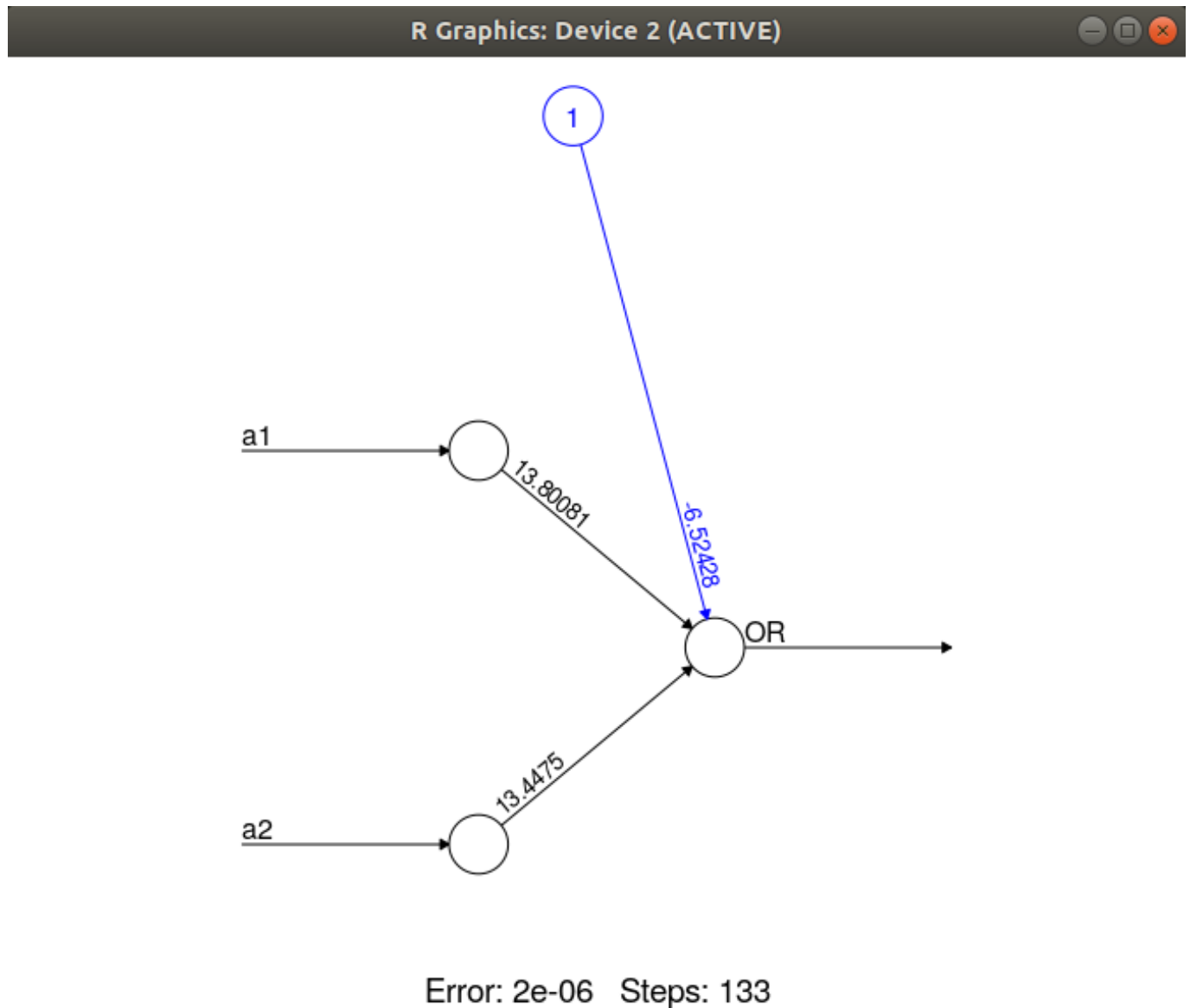
Az első példában 2 inputunk van (a1 és a2) és egy kimenet, amit OR-nak neveztünk el. Az a1 és az a2 tartalmazza az adatokat és az OR sorral pedig azt mondjuk neki, hogy mit kell kapnunk. Tehát ezzel fixáljuk le a szabályokat. Ezen adatok összeségét elnevezzük `or.data`-nak amit a `data.frame()` függvénnyel oldhatunk meg. Itt kell megadni paraméterként az adatokat. Ezután deklaráljuk az `nn.or` változót is, amihez a `neuralnet()` függvényt használjuk. Az `OR=a1+a2` azt jelenti, hogy az OR a kimenet, majd kötőjellel elválasztjuk a bemenettől és a bemeneteket vessző helyett + jellel adjuk hozzá. A `hidden=0` pedig azt jelenti, hogy itt most nem lesz rejtett réteg. Majd a `plot()` függvénnyel íratjuk ki az eredményt.

```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
OR  <- c(0,1,1,1)

or.data <- data.frame(a1, a2, OR)
nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE, ←
  stepmax = 1e+07, threshold = 0.000001)

plot(nn.or)
```

Lényegében ezeket kell majd bemásolni az R szimulációba és megjelenik az alábbi ábra a képernyőn. Itt látható, hogy az `ERROR: 2e-06`, ami azt jelenti, hogy a számítási hibahatár `0.0000002`, ami nagyon kicsi, így ez az érték elfogadható.



A következő példánál már egy új sort adunk hozzá (AND) és ugyanúgy futtatjuk a programot, ahogy a fenti kódcsipetnél csináltuk. Itt az OR és az AND két külön adat lesz, tehát a program nem veszi őket figyelembe számításnál, csak az a1 és az a2 adataival számol.

```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
OR <- c(0,1,1,1)
AND <- c(0,0,0,1)

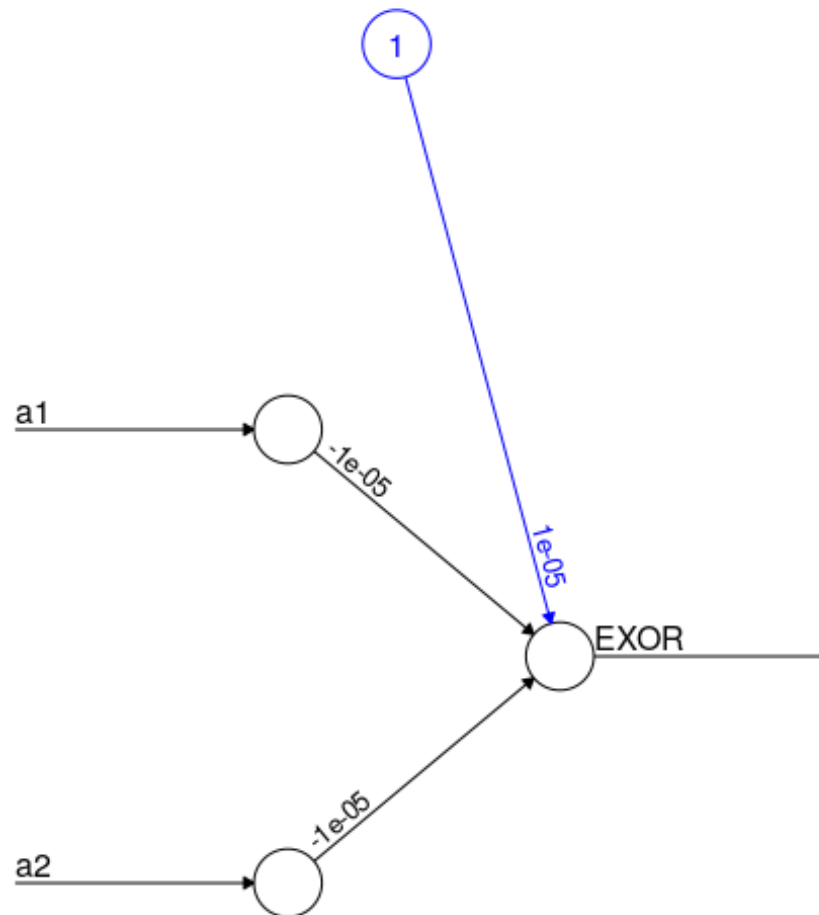
orand.data <- data.frame(a1, a2, OR, AND)
```

```
nn.orand <- neuralnet(OR+AND~a1+a2, orand.data, hidden=0, linear.output= FALSE, stepmax = 1e+07, threshold = 0.000001)

plot(nn.orand)

compute(nn.orand, orand.data[,1:2])
```

És most jön az EXOR. Ha a kódcipetet bemásoljuk az R szimulációba, akkor az alábbi kapjuk.



Error: 0.5 Steps: 79

Figyeljük meg a hibahatárt. A 0.5-es hiba nagyon nagy, a program nem tud egyértelmű választ adni, ezért szükségünk lesz többretegű neuronokra a probléma megoldására, amit az alábbi kódnál mutatom be:

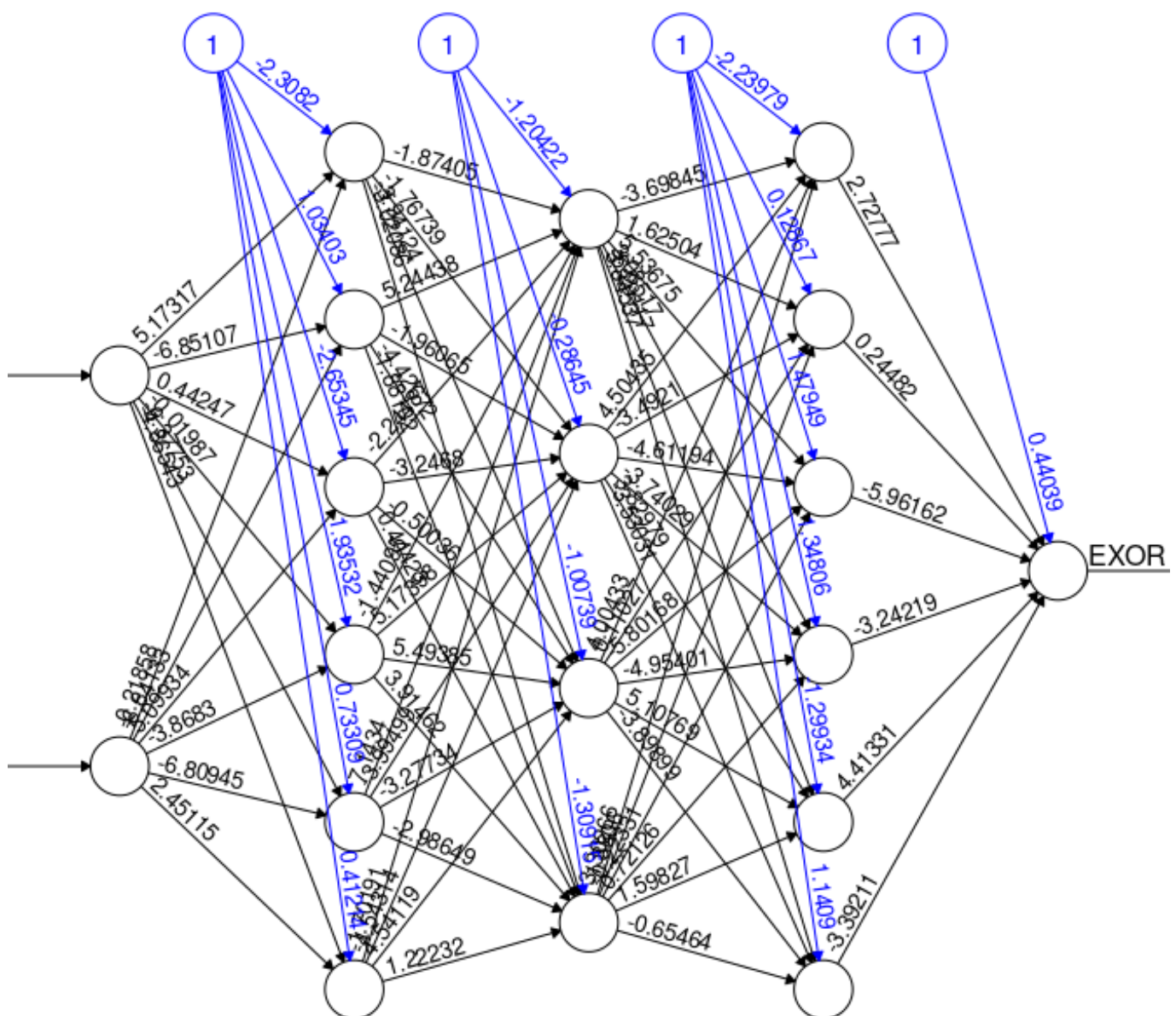
```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
EXOR <- c(0,1,1,0)

exor.data <- data.frame(a1, a2, EXOR)
```

```
nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=c(6, 4, 6), linear. <-
  output=FALSE, stepmax = 1e+07, threshold = 0.000001)
plot(nn.exor)

compute(nn.exor, exor.data[,1:2])
```

Az előző kódrészlethez képest annyi a változás, hogy a `hidden=c(6, 4, 6)`-nál hozzáírtunk 6 neuront, tartalmaz további 4 neuront és megint 6-ot. Ezzel a töbrétegű neuronok hozzáadásával a hibahatár 0.000001 lesz. További különbség az eddigi ábrához képest, hogy a 2 input és az output között megjelennek a neuronok, azaz 6, 4 és 6 kör lesz oszloponként.



Error: 1e-06 Steps: 67

4.6. Hiba-visszaterjesztéses perceptron

C++

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

Megoldás forrása: <https://github.com/nbatfai/nahshon/blob/master/ql.hpp#L64> https://gitlab.com/davidhalasz/-/blob/master/attention_raising/Source/caesar/perceptron

Ez a program több fájlból áll össze, de mi most csak a main.cpp-t vizsgáljuk meg. Onnan lehet tudni, hogy más programot is használunk, hogy headerben hozzáadjuk őket (`#include "ml.hpp"` és `#include <png++/png.hpp>`) A `main()` függvényen belül a `png::` névterű sorban mondjuk meg, hogy mi legyen az importált kép amit szeretnénk beadni a tanulo programnak. A `int size` változóban deklaráljuk, hogy mennyi a kép szélessége és hosszúsága pixelben megadva. Ezek elvégzéséhez szüksége lesz a programnak a headerben megadott `png++/png.hpp` fájlra.

```
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];
```

A `Perceptron *p` változóban újperceptront hozunk létre. Ez a függvény az `ml.hpp`-ben van felépítve. A függvény 4 paramétert vár: a 3-as szám a layerek száma lesz, a `size` azt jelenti, hogy az első rétegre `size` darab neuront akarunk, a második réteg 256 legyen, a harmadik pedig legyen 1. Alatta az `image` mutató "size" méretű tömb lesz, de egyelőre itt csak a hely lesz neki foglalva, amit majd a for ciklusban feltöltjük őket.

```
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;
```

Az első for ciklussal végigmegyek a kép szélességén, a második ciklussal pedig a magasságon. Az `image` tömböt feltöltjük a red componensekkel. Majd a `delete` függvénnyel töröljük a `Perceptron* p`-t mivel előzőleg foglaltunk neki memóriát, továbbá az `image` tömböt is a memóriából.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

A mandelbrot-halmaz

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/mandelpngt.c++

A forrásban megfigyelhető, hogy az eddigiekhez képest két új könyvtárat is használunk a headrünkben. Az egyik a `#include "png++/png.hpp"`, amivel png kép készíthető el, a másik könyvtár pedig a `#include < sys/times.h >`, amivel mérhetjük az időt, jelen esetben azt, hogy meddig tart a amíg a program elkészíti a képet. Ez utóbbit kihagyhatjuk, ez csak akkor érdekes, ha meg szeretnénk vizsgálni, hogy mennyi idő alatt készíti el a képet a processzor és mennyi ideig a videokártya.

A main függvény 2 argumentumot tartalmaz, amit az if függvénnyel ellenőrizünk le, azaz, ha a terminálban megadott paraméterek száma nem egyenlő a kettővel, akkor kiíratjuk a kép elkészítéséhez szükséges parancssor használat módját és egy hibaüzenetet. Az első paraméternek a program nevének, a másodiknak pedig az elmenteni kívánt png típusú fájl nevének kell lennie.

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

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

}
```

Következő lépésként deklaráljuk a kép számításához szükséges adatokat (a, b, c, d, szélesség és magasság). Amit a következő négy sorban használjuk fel. A `png::image < png::rgb_pixel > kep (szélesség, magasság);` kódsorral létrehozunk egy 600x600 pixel méretű üres képet, továbbá deklaráljuk a dx és a dy változókat, melyek értékei az x és y tengely lépésközeit adja meg.

```
double a = -2.0, b = .7, c = -1.35, d = 1.35;
int szélesség = 600, magasság = 600, iteraciosHatar = 32000;

png::image < png::rgb_pixel > kep (szélesség, magasság);

double dx = (b - a) / szélesség;
double dy = (d - c) / magasság;
double reC, imC, reZ, imZ, ujreZ, ujimZ;

}
```

Ezt követően két for ciklussal zongorázzuk végig az adatokat, melyek addig tartanak, amennyi a szélesség és a magasság mérete. Miközben egyenként végigmegyünk a komplex síkon, a while ciklussal ellenőrizzük, hogy a a reZ és a imZ négyzeteinek összege kisebb-e, mint 4 ÉS hogy az iteráció nem éri-e el az iterációs határt. Ha a feltétel teljesül, akkor az adotrácspont nem lesz a Mandelbrot-halmaz eleme és számítással új értékeket rendelünk a reZ és az imZ változókhoz. Ettől a while ciklustól függ a pixelek kiíratása, azaz, ha a feltétel teljesül, akkor nem lesz pixel, ellenkező esetben előhívjuk a `kep.set_pixel()` függvényt a fekete képpont rajzolásához, ahol a k lesz a sor, a j pedig az oszlop.

```
for (int j = 0; j < magasság; ++j)
{
    for (int k = 0; k < szélesség; ++k)
    {
        reC = a + k * dx;
        imC = d - j * dy;
        reZ = 0;
        imZ = 0;
        iteracio = 0;

        while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
        {
            ujreZ = reZ * reZ - imZ * imZ + reC;
            ujimZ = 2 * reZ * imZ + imC;
            reZ = ujreZ;
            imZ = ujimZ;

            ++iteracio;
        }

        kep.set_pixel (k, j,
            png::rgb_pixel (255 -
                (255 * iteracio) / iteraciosHatar,
                255 -
                (255 * iteracio) / iteraciosHatar,
                255 -
```

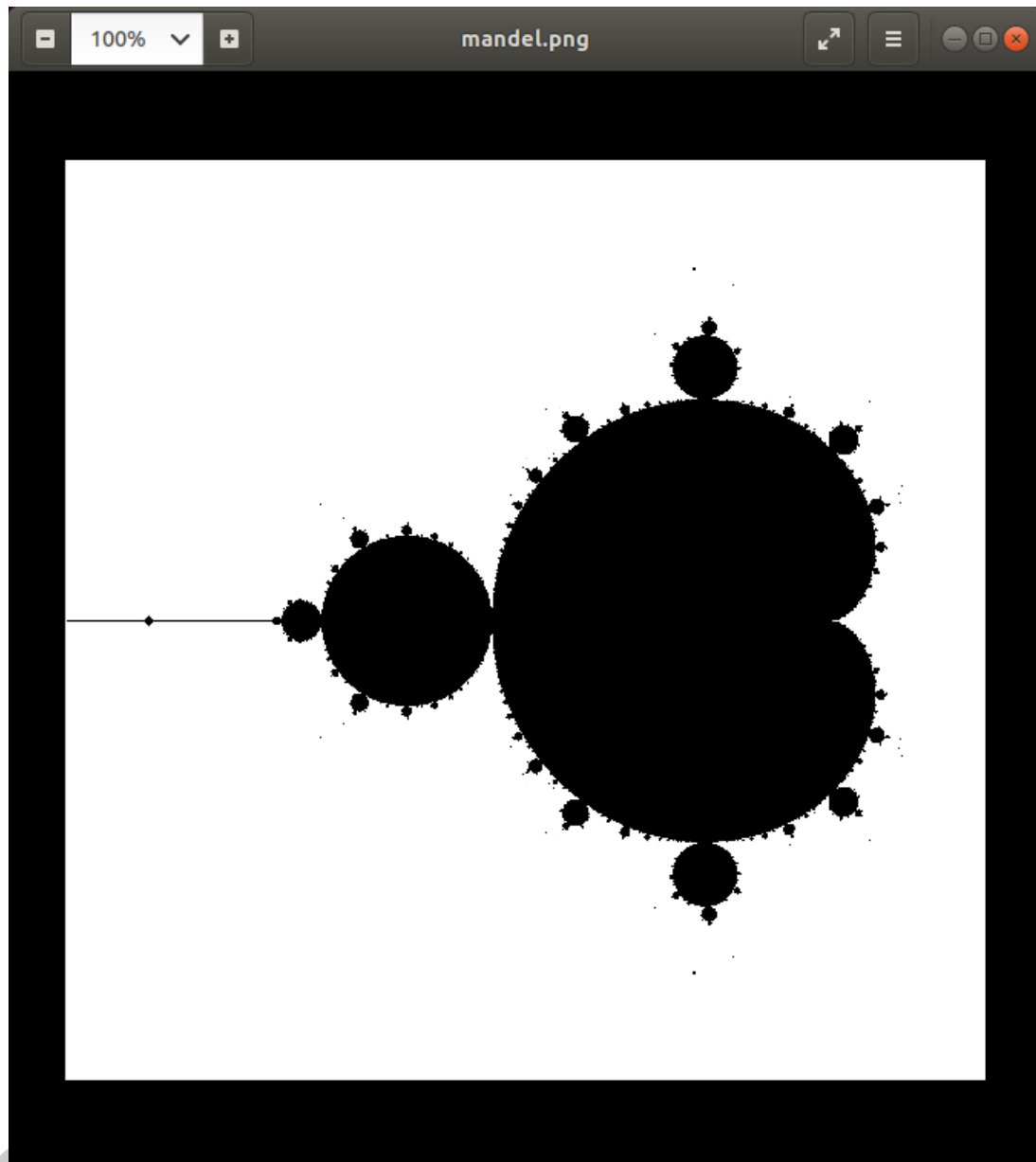
```
        (255 * iteracio) / iteraciosHatar));  
    }  
    std::cout << "." << std::flush;  
}  
}
```

A fenti képet végül lementjük abba a fájlba, amit még a program elindítása során adtuk meg és kiíratjuk, hogy a kép mentése elkészült. Továbbá a kép elkészítésének idejét is kiíratjuk, amihez előbb számításokat kell végeznünk.

```
kep.write (argv[1]);  
std::cout << argv[1] << " mentve" << std::endl;  
  
times (&tmsbuf2);  
std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime  
    + tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;  
  
delta = clock () - delta;  
std::cout << (double) delta / CLOCKS_PER_SEC << " sec" << std::endl;  
}
```

A program elindítása és a végeredmény az alábbi két képen látható:

```
dave@dave-K501UB:~/gyakorlas$ g++ mandelpngt.c++ -lpng16 -o3 -o mandelpngt  
dave@dave-K501UB:~/gyakorlas$ ./mandelpngt mandel.png  
2182  
21.8237 sec  
mandel.png mentve  
dave@dave-K501UB:~/gyakorlas$
```



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

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/mandelbrot/-3.1.2.cpp

Az `std::complex` standard könyvtár segítségével a komplex számokat (azaz a valós és imaginárius egységeket) a `complex()` függvénnyel egy helyen kezelhetjük, anélkül, hogy további változókat hoznánk létre. Most az előző Mandelbrot programot készítjük el a `complex` könyvtár használatával. Az időmérést ebből a programból most kihagyjuk.

A headerbe beillesztjük a `<complex>` könyvtárat. A `main` függvényen belül megadjuk a szélességet és a hosszúságot is, jelen esetben most egy fullhd értékeivel megegyező méretet adtunk meg. Az `if()` függvényben megvizsgáljuk, hogy a program elindításához szükséges parancssor paramétereinek száma egyenlő-e 9-cel, ha ige, akkor a változókhoz értékeket rendelünk hozzá. Az `atoi()` függvény azt csinálja, hogy a be-

vitt adatot integer típusú értékke alakítja, míg az atof() függvény pedig double lesz. Ellenkező esetben, ha a feltétel nem teljesül, akkor kiíratjuk a program helyes elindításához szükséges módot, továbbá egy hibaüzenetet.

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

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

    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double a = -1.9;
    double b = 0.7;
    double c = -1.3;
    double d = 1.3;

    if ( argc == 9 )
    {
        szelesseg = atoi ( argv[2] );
        magassag = atoi ( argv[3] );
        iteraciosHatar = atoi ( argv[4] );
        a = atof ( argv[5] );
        b = atof ( argv[6] );
        c = atof ( argv[7] );
        d = atof ( argv[8] );
    }
    else
    {
        std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c d ↵"
                  << std::endl;
        return -1;
    }
}
```

További különbség az előző programunkhoz képest, hogy a második for cikluson belül használjuk az std::complex osztályt. Jellemzője, hogy double típusú paraméterek lesznek és két részből áll: egy valós (reC) és egy imaginárius (imC) egységből. Ez lesz a c változó. Ugyanígy adjuk meg a z_n változót is, csak itt nullákat adunk meg, amik majd a while ciklusban fog változni, ha z_n értéke kisebb mint 4 és az iteráció kisebb mint az iterációs határ. Ha az adott érték eleme a Mandelbrot-Halmaznak, akkor pixeleket rajzolunk, továbbá kiíratjuk, hogy hány százaléknál jár a kép feldolgozása, majd az előző programból ismert módon mentünk.

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

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

```
int iteracio = 0;

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

for ( int j = 0; j < magassag; ++j )
{
    for ( int k = 0; k < szelesseg; ++k )
    {
        reC = a + k * dx;
        imC = d - j * dy;
        std::complex<double> c ( reC, imC );

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

        while ( std::abs ( z_n ) < 4 && iteracio < iteraciosHatar )
        {
            z_n = z_n * z_n + c;

            ++iteracio;
        }

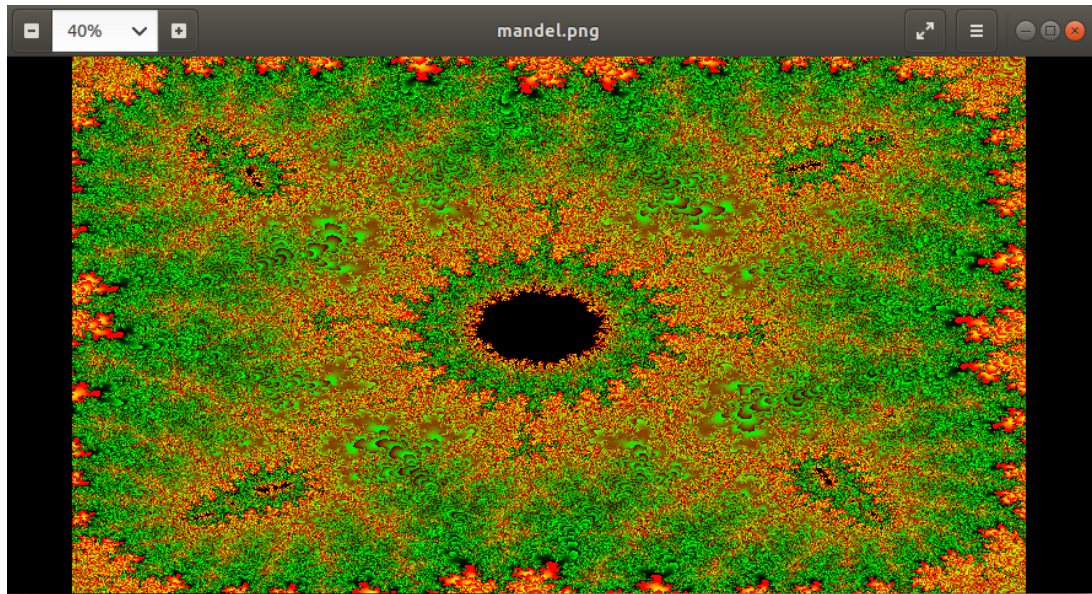
        kep.set_pixel ( k, j,
            png::rgb_pixel ( iteracio%255, (iteracio*iteracio)%255, 0 ) );
    }

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

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

A program elindítása után egy nem egy fekete-fehér képet, hanem sokkal színebb eredményt kapunk:

```
dave@dave-K501UB:~/gyakorlas$ g++ 3.1.2.cpp -lpng -O3 -o 3.1.2
dave@dave-K501UB:~/gyakorlas$ ./3.1.2 mandel.png 1920 1080 2040 -0.01947381057309366392260585598705802112818 -0.01947381057254134184564264842
26540196687 0.7985057569338268601555341774655971676111 0.798505756934379196110285192844457924366
Szamitas
mandel.png mentve.
```



5.3. Biomorfok

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

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Biomorf/3.1.3.cpp

Vmi a Julia halmazról. Az előző feladathoz képest az a különbség, hogy itt 9 helyett 12 paramétert kell megadni a program elindítása során. Ugyanis itt a valós és az imaginárius egységek továbbá az R is állandó érték lesz.

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

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

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

    if ( argc == 12 )
    {
        szelesseg = atoi ( argv[2] );
        magassag = atoi ( argv[3] );
        iteraciosHatar = atoi ( argv[4] );
```

```
xmin = atof ( argv[5] );
xmax = atof ( argv[6] );
ymin = atof ( argv[7] );
ymax = atof ( argv[8] );
reC = atof ( argv[9] );
imC = atof ( argv[10] );
R = atof ( argv[11] );

}
else
{
    std::cout << "Hasznalat: ./3.1.3 fajlnev szelesseg magassag n a b c ←
        d reC imC R" << std::endl;
    return -1;
}

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

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

std::complex<double> cc ( reC, imC );

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

A másik lényeges különbség a for ciklusban figyelhető meg. Itt nincs while ciklus, helyette for-t használunk. Az első két ciklussal végigmegyünk a rácson, a harmadik pedig addig fog futni, amíg el nem éri az iterációs határt. Az utóbbin belül a z_n értéke az aktuális z_n változó értékének 3. hatványa lesz, amit a `std::pow()` osztállyal számítjuk ki, majd ezt az értéket vizsgáljuk meg egy `if()` függvénnyel. Az `std::real()` és azt `std::imag()` értelemszerűen azt jelenti, hogy az adott z_n változó valós és imaginárius értékét vegye figyelembe. Tehát ha a valós szám vagy az imaginárius szám értéke nagyobb, mint R akkor a feltétel teljesül és az iteracio értéke az adott i változó értékével lesz megegyező. Ezután elkészítjük a Julia-Halmazos képet.

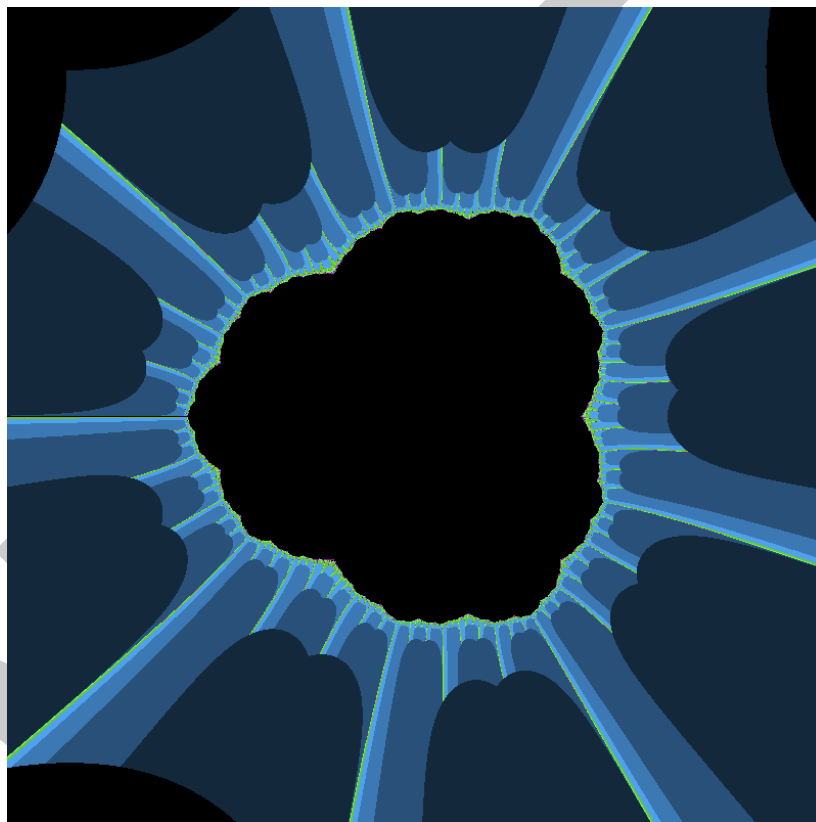
```
for ( int y = 0; y < magassag; ++y )
{
    for ( int x = 0; x < szelesseg; ++x )
    {
        double reZ = xmin + x * dx;
        double imZ = ymax - y * dy;
        std::complex<double> z_n ( reZ, imZ );

        int iteracio = 0;
        for (int i=0; i < iteraciosHatar; ++i)
        {
            z_n = std::pow(z_n, 3) + cc;
            if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
            {
                iteracio = i;
            }
        }
    }
}
```

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

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

Ezután elkészítjük a Julua-Halmazos képet:



5.4. A Mandelbrot halmaz CUDA megvalósítása !!

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/CUDA/mandelpngc_60x60

A CUDA az NVIDIA által fejlesztett programozási környezet, amely az NVIDIA grafikus processzorainak párhuzamos programozására használható. Legfontosabb célja, hogy az adott feladatokat a lehető legrövidebb idő alatt oldja meg. Amíg a CPU-k csak néhány szálat tudnak egymástól függetlenül futtatni, addig a GPU-k sokkal nagyobb számú feladatokat is képesek elvégezni, és a CUDA ezt az előnyt használja ki, nem csak képalkotási feladatoknál, hanem akár általános célú programok megvalósítása során is felhasználhatjuk.

Jelen példánkban a `.cu` fájl típusú programot fogjuk használni. Ehhez előzetesen szükségünk lesz az NVIDIA cuda toolkit programot telepítenünk, hogy tudjuk használni az `nvcc` parancsot a program fordítása során. A programunk az első feladatban megismert alkalmazás továbbfejlesztése. A legszembevetőbb változtatás a programunk elején láthatóak, amelyek mindig valamilyen függvény minősítővel kezdődnek. Ilyen függvény minősítők lehetnek például: `__global__`, `__device__` `__host__` és a `__noinline__`, `__forceinline__`. Jelen esetünkben van egy `__device__` függvényünk. Ez a GPU-n futó függvény kódja, ami csak a GPU kódból hívható. Ide rakjuk be az első feladatból megismert változókkal, számításokkal és a rács csomópontjaival kapcsolatos kódcsipetet. A magasság és a meret most 600x600 pixel lesz.

```
__device__ int
mandel (int k, int j)
{

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

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

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

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

        ++iteracio;
    }
    return iteracio;
}
```

Továbbá van még két `__global__` függvényminősítőnk, és mindkettő a `mandelkernel()` függvényhez

kapcsolódik. Ez egy kernel függvény lesz, ami azt jelenti, hogy a GPU-n futó kódból, azaz a gazda kódból hívható. A `blockIdx` a CUDA saját beépített változója, ami `uint3` típusú, a futó blokk számát adja meg. A `threadIdx` is egy `uint3` típusú változó, csak ez a futó szál számát fogja megadni.

```
/*
__global__ void
mandelkernel (int *kepadat)
{

    int j = blockIdx.x;
    int k = blockIdx.y;

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

}
*/

__global__ void
mandelkernel (int *kepadat)
{

    int tj = threadIdx.x;
    int tk = threadIdx.y;

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

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

}
```

Ez a `cudamandel()` függvény két paramétert vár: integer típusú `kepadat` tömb értékeit, ami értéke most egyenként 600 lesz. A `cudaMalloc` működése hasonló a C++ `malloc()` függvényéhez, amit szintén memó-riafoglaláshoz használunk. A `dim3` típusú `grid` blokk mérete 60x60 lesz, mivel a méretet elosztjuk 10-el és blokkonként 100 szállal fog dolgozni, ezt a `tgrid()` függvénnyel adtuk meg.

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

    int *device_kepadat;
    cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int));

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

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

}
```

```
}  
  
}
```

A programunk main() részét nem tartom szükségesnek bemutatni, ugyanis ha megvizsgáljuk, akkor nem találunk semmi újdonságot az eddigi programunkhoz képest. Van benne egy cudamandel() függvény, ami megfogja kapni a fenti adatokat, így fogjuk elkészíteni a képet.

```
int  
main (int argc, char *argv[])  
{  
  
    // MÉRÜNK IDŐT (PP 64)  
    clock_t delta = clock ();  
    // MÉRÜNK IDŐT (PP 66)  
    struct tms tmsbuf1, tmsbuf2;  
    times (&tmsbuf1);  
  
    if (argc != 2)  
    {  
        std::cout << "Hasznalat: ./mandelpngc fajlnev";  
        return -1;  
    }  
  
    int kepadat[MERET][MERET];  
  
    cudamandel (kepadat);  
  
    png::image < png::rgb_pixel > kep (MERET, MERET);  
  
    for (int j = 0; j < MERET; ++j)  
    {  
        //sor = j;  
        for (int k = 0; k < MERET; ++k)  
        {  
            kep.set_pixel (k, j,  
                png::rgb_pixel (255 -  
                    (255 * kepadat[j][k]) / ITER_HAT,  
                    255 -  
                    (255 * kepadat[j][k]) / ITER_HAT,  
                    255 -  
                    (255 * kepadat[j][k]) / ITER_HAT));  
        }  
    }  
    kep.write (argv[1]);  
  
    std::cout << argv[1] << " mentve" << std::endl;  
  
    times (&tmsbuf2);  
    std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
```



```
+ tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;

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

}
```

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!

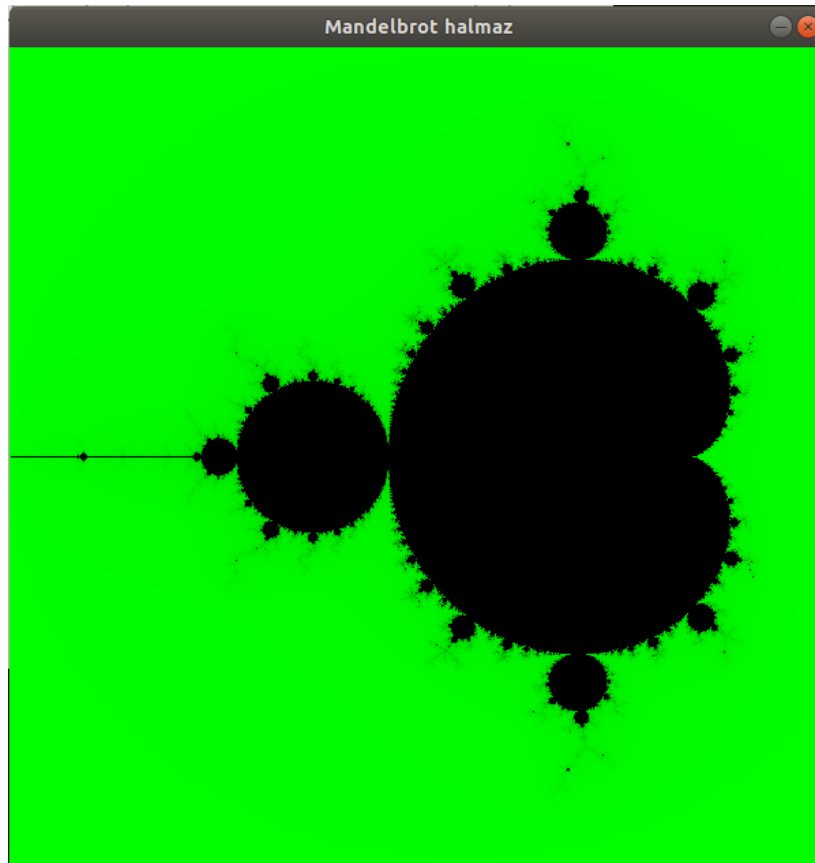
Megoldás forrása: https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/mandelbrot/nagyitas

Tutorial: Fürjes-Benke Péter

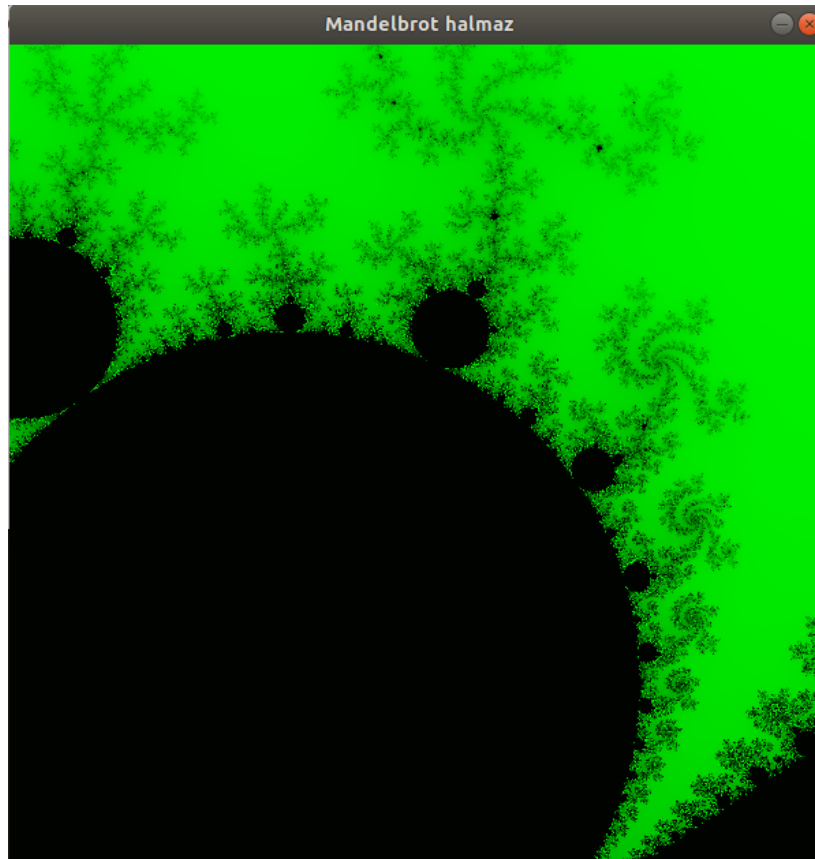
Mielőtt elkezdenénk a programot, először szükséges telepíteni a libqt4-dev csomagot, mert enélkül nem fog elindulni a program. A mappában egy helyen kell szerepelnie az alábbi négy fájlnek, ami a képen is látható, miután elindítottam az `ls -l` parancsot. Továbbá a `qmake -project` létrehoz egy `.pro` végződésű fájlt, a fájl neve megegyezik az aktuális mappa nevével. Miután ez megtörtént, ezt a `.pro` végződésű fájl-ba bele kell másolni ezt a sort: `QT += widgets`. Ezután lehet a parancssorba beírni, hogy `qmake *.pro`

```
dave@dave-K501UB:~/gyakorlas/nagyitas$ ls -l
összesen 24
-rw-rw-r-- 1 dave dave 2634 márc 23 22:22 frakablak.cpp
-rw-rw-r-- 1 dave dave 1348 márc 23 22:22 frakablak.h
-rw-rw-r-- 1 dave dave 2609 márc 23 22:22 frakszal.cpp
-rw-rw-r-- 1 dave dave 866 márc 23 22:22 frakszal.h
-rw-rw-r-- 1 dave dave 600 márc 23 22:22 main.cpp
-rw-r--r-- 1 dave dave 976 márc 25 20:25 nagyitas.pro
dave@dave-K501UB:~/gyakorlas/nagyitas$ qmake -project
dave@dave-K501UB:~/gyakorlas/nagyitas$ qmake *.pro
```

Ez a parancs létrehozza a Makefile-t, amit a `make` parancs használatával egy bináris fájlt hozunk létre, aminek a neve megegyezik az aktuális mappánk nevével. Ezt kell elindítanunk, ugyanúgy ahogy bármely más programot szoktunk. Haminden jól megy akkor az alábbi Mandelbrot-halmazt kell kapnunk:



Nagyítani úgy tudunk, hogy az egérrel kijelölünk egy részt, majd a program automatikusan oda zoomol. Ha a kép életlennek tűnik, akkor meg kell nyomnunk az "n" billentyűt, ugyanis ilyenkor a program számítást végez és élesíti a képet.



5.6. Mandelbrot nagyító és utazó Java nyelven !!

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/mandelbrot/java

Ebben a feladatban a az előzőt csináljuk meg ugyanazt, csak java-ban. A fordítás a `javac` paranccsal történik, és a program elindításához pedig a `java`-t használjuk. Ezt az alábbi kódsor szemlélteti:

```
javac MandelbrotHalmazNagyító.java
java MandelbrotHalmazNagyító
```

A nagyítást végző kódsorokat `MandelbrotHalmazNagyító.java` programon belül a `MandelbrotHalmazNagyító` osztályban hozzuk létre. A `MouseListener` implementálja a `MouseListener`-t. A `MousePressed` osztály a `MouseEvent` paramétereit várja, ami akkor lép működésbe, ha egy pontra történik az egér klikkelés. Ilyenkor az osztályon belül lekéri a nagyítandó kijelölt terület bal felső sarkának helyét. Alaphelyzetben a szélesség (`mx`) és a magasság (`my`) értékei 0, ami megváltozik ha az egérrel kijelölünk egy területet.

```
// Egér kattintó események feldolgozása:
addMouseListener(new java.awt.event.MouseAdapter() {
    // Egér kattintással jelöljük ki a nagyítandó területet
    // bal felső sarkát:
    public void mousePressed(java.awt.event.MouseEvent m) {
        // A nagyítandó kijelölt területet bal felső sarka:
        x = m.getX();
        y = m.getY();
    }
})
```

```
        mx = 0;
        my = 0;
        repaint();
    }
}
```

Ha kijelöltünk egy területet, akkor a `MouseReleased()` osztály elvégz egy számítást, és egy új nagyító objektumot hoz létre.

```
public void mouseReleased(java.awt.event.MouseEvent m) {
    double dx = (MandelbrotHalmazNagyító.this.b
        - MandelbrotHalmazNagyító.this.a)
        /MandelbrotHalmazNagyító.this.szélesség;
    double dy = (MandelbrotHalmazNagyító.this.d
        - MandelbrotHalmazNagyító.this.c)
        /MandelbrotHalmazNagyító.this.magasság;

    new MandelbrotHalmazNagyító(MandelbrotHalmazNagyító.this.a+x*dx,
        MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,
        MandelbrotHalmazNagyító.this.d-y*dy-my*dy,
        MandelbrotHalmazNagyító.this.d-y*dy,
        600,
        MandelbrotHalmazNagyító.this.iterációsHatár);
}
}
```

A `mouseDragged()` osztállyal figyeljük, hogy kijelölt területet. Ekkor kapjuk meg a szélességet és a magasságot értékül, amik rendre az `mx` és az `my` változóban lesznek eltárolva.

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

Itt is lehet élesíteni a képet, ha nagyítás után homályos lesz. Ehhez a `KeyAdapter()` osztályt hívjuk segítségül, ami a `MandelbrotHalmaz.java` programban található. A `KeyListener()` figyel a billentyűzetleütéseket. Jelen esetben az 's', 'n' és 'm' gombokat figyel. Ha az a kapott billentyűzet (`e.getKeyCode()`) egyenlő az 'n' billentyűzettel (`KeyEvent.VK_N`), akkor újraszámítjuk az adatokat, kivéve, ha már előtte ki volt számítva.

```
...
addKeyListener(new java.awt.event.KeyAdapter() {
    // Az 's', 'n' és 'm' gombok lenyomását figyeljük
    public void keyPressed(java.awt.event.KeyEvent e) {
        if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)
            pillanatfelvétel();
        // Az 'n' gomb benyomásával pontosabb számítást végzünk.
    }
});
```

```
        else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {
            if(számításFut == false) {
                MandelbrotHalmaz.this.iterációsHatár += 256;
                // A számítás újra indul:
                számításFut = true;
                new Thread(MandelbrotHalmaz.this).start();
            }
        } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_M) {
            if(számításFut == false) {
                MandelbrotHalmaz.this.iterációsHatár += 10*256;
                // A számítás újra indul:
                számításFut = true;
                new Thread(MandelbrotHalmaz.this).start();
            }
        }
    }
}
...
}
```

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás C++ forrása: https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/welch/polargen/cpp

Megoldás Java forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/-polargen/java/PolarGenerator.java

Ebben a feladatban a polártranszformációs algoritmust fogom bemutatni. Először vizsgáljuk meg a C++ programot. Ez a program 3 részből áll: `polargenteszt.cpp`, `polargen.cpp` és `polargen.h`. A `polargen.h` felépítése az alábbi:

```
#ifndef POLARGEN__H
#define POLARGEN__H

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

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

```
private:
    bool nincsTarolt;
    double tarolt;

};

#endif
```

Első lépésként deklaráljuk a `nincsTarolt` változót, aminek értéke igaz lesz. Ezután a véletlenszám-generátor függvényt hívjuk segítségül. Azért használjuk az `srand()` függvényt a `rand()` helyett, mert folyamatosan változó, megjósolhatatlan véletlen számra van szükségünk, ezért a függvénynek paraméterül a `std::time()` függvényt adjuk meg, azaz a számítógépünk órája által jelzett idő bitje lesz. Figyeljük meg, hogy a véletlenszám készítőnk egy `PolarGen` osztályban van felépítve. Így használhatóbb lesz a generátorunk és könnyebb lesz különböző értéktartományokhoz véletlen számokat előállítani. A `nincsTarolt` egy boolean típusú változó lesz, a `tarolt` pedig `double` típusú. Ezt a programban más-hol nem fogjuk tudni megváltoztatni, mivel ez egy `private` osztályban vannak elhelyezve. Van a programban egy `double` `kovetkezo()` függvényünk is, mely a `polargen.cpp`-ben van felépítve:

```
#include "polargen.h"

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

        double r = std::sqrt ((-2 * std::log (w)) / w);

        tarolt = r * v2;
        nincsTarolt = !nincsTarolt;

        return r * v1;
    }
    else
    {
        nincsTarolt = !nincsTarolt;
        return tarolt;
    }
}
```

```
}
```

Gyakorlatilag ez maga a polártranszformációs függvényünk. A számításokat itt végezzük el. Először is megvizsgáljuk hogy a `nincsTarolt` változó értéke hamis, vagy igaz. Ha igaz, akkor azt jelenti, hogy a `tarolt` változóban el van tárolva a visszaadandó lebegőpontos szám. A matematikai háttér most nem fontos, ezért a polártranszformációs eljárással kapcsolatos kódcsipetet átlépjük.

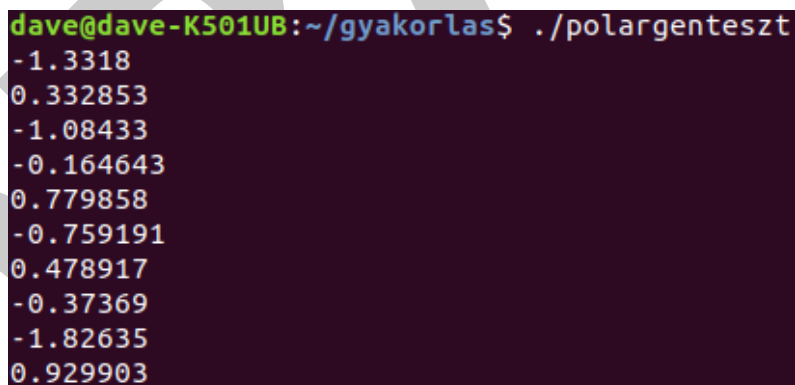
```
#include <iostream>
#include "polargen.h"
#include "polargen.cpp"

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

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

    return 0;
}
```

A fenti kódcsipet a `polargenteszt.cpp` fájlban található. Ez lesz a fő fájl, azaz majd ezt kell lefordítanunk és futtatni. For ciklussal megadjuk, hogy a program tízszer fusson le és írassa ki a kapott számításokat. Így tehát a program futtatása után megkapjuk a terminálban a következő kimenetet:



```
dave@dave-K501UB:~/gyakorlas$ ./polargenteszt
-1.3318
0.332853
-1.08433
-0.164643
0.779858
-0.759191
0.478917
-0.37369
-1.82635
0.929903
```

Most nézzük meg ugyanezt a programot Java-ban. Ha összehasonlítjuk az előző programmal, akkor láthatjuk, hogy szembetűnő változás nincs. Talán csak annyi, hogy a Java verzió sokkal letisztultabbnak tűnik. A Java alapról osztályokkal dolgozik, tehát itt programozni csak az objektum orientált paradigma mentén lehet.

```
public class PolarGenerator {
    boolean nincsTarolt = true;
    double tarolt;

    public PolarGenerator() {
        nincsTarolt = true;
    }
}
```



```
public double következő() {
    if(nincsTárolt) {
        double u1, u2, v1, v2, w;
        do {
            u1 = Math.random();
            u2 = Math.random();
            v1 = 2*u1 - 1;
            v2 = 2*u2 - 1;
            w = v1*v1 + v2*v2;
        } while(w > 1);
        double r = Math.sqrt((-2*Math.log(w))/w);
        tárolt = r*v2;
        nincsTárolt = !nincsTárolt;
        return r*v1;
    } else {
        nincsTárolt = !nincsTárolt;
        return tárolt;
    }
}

public static void main(String[] args) {

    PolarGenerator g = new PolarGenerator();

    for(int i=0; i<10; ++i)
        System.out.println(g.következő());
}
```

A programban megfigyelhető következő() függvény megtalálható a Sun programozói által készített Java JDK forrásaiban, pontosabban az src.zip állományában a java/util/Random.java forrásban. Ennek a kódcsipetnek a megoldása nagyon hasonlít a miénkhez, amit az alábbi képen is megfigyelhetjük:

```
public synchronized double nextGaussian() {
    // See Knuth, ACP, Section 3.4.1 Algorithm C.
    if (haveNextNextGaussian) {
        haveNextNextGaussian = false;
        return nextNextGaussian;
    } else {
        double v1, v2, s;
        do {
            v1 = 2 * nextDouble() - 1; // between -1 and 1
            v2 = 2 * nextDouble() - 1; // between -1 and 1
            s = v1 * v1 + v2 * v2;
        } while (s >= 1 || s == 0);
        double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
        nextNextGaussian = v2 * multiplier;
        haveNextNextGaussian = true;
        return v1 * multiplier;
    }
}
```

A JDK-ban a `haveNextNextGaussian` és a `nextNextGaussian` a fenti `nextGaussian()` függvény felett van definiálva, mégpedig úgy, hogy a változók `private` módban vannak megadva, azaz csak a saját osztálya számára lesz látható, más olyszály azonban nem fog tudni hozzáférni az adathoz:

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

6.2. LZW

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

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa/-lzw.c

Az LZW (Lempel-Ziv-Welch) az LZ77 tömörítőprogram továbbfejlesztése, amit Terry Welch publikált 1984-ben. Ez lényegében egy veszteségmentes tömörítési eljárás. Mivel könnyű implementálni, ezért nagyon elterjedt. Jelen programunk lényege az, hogy egy binárisan megadott karakter-sorozatot részekre tördeljük rekurzíven. Ha egy darabot letördeltünk, akkor utána addig vizsgáljuk a sorozat további részeit, ami még nincs a "szótárunkban", ugyanis ha ilyen találat van, akkor azt szintén letördeljük és felvesszük a szótárunkba. Ha a sorozat végéhez érünk, akkor ebből a részekből egy bináris fa állítható elő. Jelen esetünkben úgy kell elképzelni, hogy a gyöker elemnek van egy 1-es és egy nullás gyermeke, majd ezekhez is fog tartozni további nullás és egyes gyermekes és így tovább.

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

typedef struct binfa
{
    int ertek;
    struct binfa *bal_nulla;
```

```
struct binfa *jobb_egy;

} BINFA, *BINFA_PTR;

BINFA_PTR
uj_elem ()
{
    BINFA_PTR p;

    if ((p = (BINFA_PTR) malloc (sizeof (BINFA))) == NULL)
    {
        perror ("memoria");
        exit (EXIT_FAILURE);
    }
    return p;
}
```

Ennél a kódcsipetnél az `uj_elem()` függvényen belül építjük fel a memóriefoglalási alprogramunkat. A `malloc` függvényről már volt szó, ugye ez fogja számunkra lefoglalni a memóriát, paraméterként a `BINFA` változót adjuk meg. Ezután a `main()` függvényben a gyökér értéket `'/'` karakterre állítjuk be, tehát ez lesz majd a bináris fánk tetején és 0-val fog kezdődni. A While ciklusban megvizsgáljuk, hogy a fa bal oldala tartalmaz-e nullát, ha nem létrehozunk egyet, `NULL` értéket adunk az egyes gyermeknek, és visszakerülünk a gyökérbe. Ezután megint egy `if` függvény jön, ugye mivel `NULL` értéket adtunk az egyes gyermeknek ezért a feltétel igaz lesz, és létrehozuk ez egyes gyermeket is.

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

int
main (int argc, char **argv)
{
    char b;

    BINFA_PTR gyoker = uj_elem ();
    gyoker->ertek = '/';
    gyoker->bal_nulla = gyoker->jobb_egy = NULL;
    BINFA_PTR fa = gyoker;

    while (read (0, (void *) &b, 1))
    {
        // write (1, &b, 1);
        if (b == '0')
        {
            if (fa->bal_nulla == NULL)
            {
                fa->bal_nulla = uj_elem ();
                fa->bal_nulla->ertek = 0;
            }
        }
    }
}
```

```
        fa->bal_nulla->bal_nulla = fa->bal_nulla->jobb_egy = NULL;
        fa = gyoker;
    }
    else
    {
        fa = fa->bal_nulla;
    }
}

    else
    {
        if (fa->jobb_egy == NULL)
        {
            fa->jobb_egy = uj_elem ();
            fa->jobb_egy->ertek = 1;
            fa->jobb_egy->bal_nulla = fa->jobb_egy->jobb_egy = NULL;
            fa = gyoker;
        }
        else
        {
            fa = fa->jobb_egy;
        }
    }
}

printf ("\n");
kiir (gyoker);
```

Az alábbi kódrészletekben csak számításokat végzünk, arra vonatkozóan, hogy mennyi lesz majd az elkészült binfánk mélysége, átlaga stb. Majd a végén kiíratjuk a kiir (BINFA_PTR elem) függvénnyel a kapott eredményt.

```
extern int max_melyseg, atlagosszeg, melyseg, atlagdb;
extern double szorasosszeg, atlag;

printf ("melyseg=%d\n", max_melyseg-1);

atlag = ((double)atlagosszeg) / atlagdb;

/* Ághosszak szórásának kiszámítása */
atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
szorasosszeg = 0.0;

rszoras (gyoker);

double szoras = 0.0;
```

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

printf ("atlag=%f\nszoras=%f\n", atlag, szoras);

szabadit (gyoker);
}

int atlagosszeg = 0, melyseg = 0, atlagdb = 0;

void
ratlag (BINFA_PTR fa)
{
    if (fa != NULL)
    {
        ++melyseg;
        ratlag (fa->jobb_egy);
        ratlag (fa->bal_nulla);
        --melyseg;

        if (fa->jobb_egy == NULL && fa->bal_nulla == NULL)
        {
            ++atlagdb;
            atlagosszeg += melyseg;
        }
    }
}

double szorasosszeg = 0.0, atlag = 0.0;

void
rszoras (BINFA_PTR fa)
{
    if (fa != NULL)
    {
        ++melyseg;
        rszoras (fa->jobb_egy);
        rszoras (fa->bal_nulla);
        --melyseg;

        if (fa->jobb_egy == NULL && fa->bal_nulla == NULL)
        {
```

```
    ++atlagdb;
    szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));

}

}

}

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

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

void
szabadit (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        szabadit (elem->jobb_egy);
        szabadit (elem->bal_nulla);
        free (elem);
    }
}
```

A program futtatása és a kapott eredményt az alábbi képen láthatjuk. Eszerint a b.txt-ben található 11 karakterből álló bináris sorból végül 6-ot tároltunk el és a fánk mélysége 3 lett.

```
dave@dave-K501UB:~/gyakorlas$ more b.txt
00011101110
dave@dave-K501UB:~/gyakorlas$ gcc lzw.c -o lzw -lm
dave@dave-K501UB:~/gyakorlas$ ./lzw < b.txt > output.txt
dave@dave-K501UB:~/gyakorlas$ more output.txt

-----1(2)
-----0(3)
-----1(1)
---/(0)
-----1(2)
-----0(1)
-----0(2)
melyseg=3
altag=2.333333
szoras=0.577350
dave@dave-K501UB:~/gyakorlas$
```

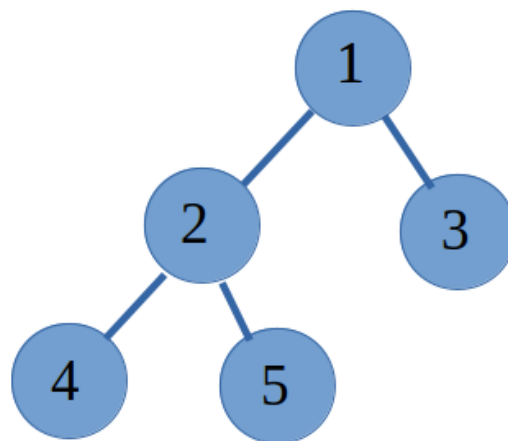
6.3. Fabejárás

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

Preorder Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa/binfa%20c/pre/lzwpre.c

Postorder Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa/binfa%20c/post/lzwpost.c

Ellentétben a lineáris adatstruktúrákkal, amelyeknek csak egy logikai módja van, a fabejárás történhet pre- és postorder módon. Az alábbi kis ábra és a hozzá tartozó szöveges magyarázat szemlélteti az in-, post és preorder közötti különbséget:



- Inorder esetén a fenti fa a következő módon lesz bejárva: 4 2 5 1 3. Azaz először alulról a bal oldali gyermeket, majd a gyökérelmet, utána annak jobb oldali gyermekét.
- Preorder: 1 2 4 5 3. Tehát először a legelső gyökérelmet, onnan a bal, majd a jobb oldali gyermeket.
- Postorder-nél pedig: 4 5 2 3 1. Alulról a bal oldali gyermeket, jobb oldali gyermeket, majd végül a gyökérelmet.

Most a forráskódból csak a megváltozott kódcsipeteket másolom be, mivel a többi teljesen megegyezik az előző feladat forráskódjával. Először vizsgáljuk meg a preorder esetet. Itt is az előző feladatban használt b.txt fájlban tárolt bináris kódsort használjuk tesztelésre.

```
...
void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;
        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem-> ←
            ertek,
            melyseg);
        kiir (elem->bal_nulla);
        kiir (elem->jobb_egy);
        --melyseg;
    }
}
...
```

A program futtatása után az alábbi lesz az eredmény:

```
dave@dave-K501UB:~/gyakorlas/pre$ more b.txt
00011101110
dave@dave-K501UB:~/gyakorlas/pre$ ./lzwpre < b.txt >output.txt
dave@dave-K501UB:~/gyakorlas/pre$ more output.txt

---/(1)
-----0(2)
-----0(3)
-----1(3)
-----1(2)
-----1(3)
-----0(4)
melyseg=3
altag=2.333333
szoras=0.577350
dave@dave-K501UB:~/gyakorlas/pre$
```

Végül pedig nézzük meg a postorder eredményét:

```
...
void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;
```



```

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

```

```

dave@dave-K501UB:~/gyakorlas/post$ gcc lzwpost.c -o lzwpost -lm
dave@dave-K501UB:~/gyakorlas/post$ ./lzwpost < b.txt >output.txt
dave@dave-K501UB:~/gyakorlas/post$ more output.txt
-----0(3)
-----1(3)
-----0(2)
-----0(4)
-----1(3)
-----1(2)
---/(1)
melyseg=3
altag=2.333333
szoras=0.577350
dave@dave-K501UB:~/gyakorlas/post$

```

6.4. Tag a gyökér

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

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa/binfa%20c++/z3a7.cpp

Ebben a feladatban a C binfás programunkat fogjuk átírni C++ verzióba. Jelen esetünkben van egy LZWBinFa osztályunk, azon belül egy Csomopont osztály továbbá egy void usage (void) és az elmaradhatatlan int main() függvényünk is. Az LZWBinFa fa egy pointer, amely mindig az épülő LZW fa azon csomópontjára mutat, amit az input feldolgozása során az LZW algoritmus logikája diktál.

```

#include <iostream>
#include <cmath>
#include <fstream>

class LZWBinFa
{
public:

    LZWBinFa () : fa (&gyoker)
    {
    }
    ~LZWBinFa ()

```

```
{
    szabadit (gyoker.egyenesGyermekek ());
    szabadit (gyoker.nullasGyermekek ());
}

void operator<< (char b)
{
    if (b == '0')
    {
        if (!fa->nullasGyermekek ())
        {
            Csomopont *uj = new Csomopont ('0');
            fa->ujNullasGyermekek (uj);
            fa = &gyoker;
        }
        else
        {
            fa = fa->nullasGyermekek ();
        }
    }
    else
    {
        if (!fa->egyenesGyermekek ())
        {
            Csomopont *uj = new Csomopont ('1');
            fa->ujEgyenesGyermekek (uj);
            fa = &gyoker;
        }
        else
        {
            fa = fa->egyenesGyermekek ();
        }
    }
}

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

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

friend std::ostream & operator<< (std::ostream & os, LZWBinFa & bf)
{
    bf.kiir (os);
    return os;
}
```

```
void kiir (std::ostream & os)
{
    melyseg = 0;
    kiir (&gyoker, os);
}
```

A Csomopont osztályba implementáljuk a Csomopont konstruktort, ami paraméter nélküli lesz és itt hozza létre a '/' karakterrel kezdődő gyökér csomópontját. Ez lesz tehát az alapértelmezett, de, ha valami betűvel hívjuk, akkor '/' helyett azt teszi be és a két gyermekre mutató mutatót pedig mindkét esetben nullára állítjuk. Ezután a nullasGyermek és egyesGyermek függvényekben kérdezzük le az aktuális csomópont bal és jobb oldali gyermekét, ujNullasGyermek és a ujEgyesGyermek függvényekben. Pontosan ugyanúgy ahogy már a C programunknál megismertük. Ami számunkra most újdonság, az, hogy a Csomopont osztályban van egy private típusú kódrészlet is. Ez azt jelenti, hogy az LZWBInfa ezekhez az adatokhoz nem közvetlenül fér hozzá, hanem csak lekérdező üzenetekkel érheti el őket.

```
private:
    class Csomopont
    {
    public:
        Csomopont (char b = '/'):betu (b), balNulla (0), jobbEgy (0)
        {
        };
        ~Csomopont ()
        {
        };

        Csomopont *nullasGyermek () const
        {
            return balNulla;
        }

        Csomopont *egyesGyermek () const
        {
            return jobbEgy;
        }

        void ujNullasGyermek (Csomopont * gy)
        {
            balNulla = gy;
        }

        void ujEgyesGyermek (Csomopont * gy)
        {
            jobbEgy = gy;
        }

        char getBetu () const
        {
            return betu;
        }
    };
};
```

```
    }

private:

    char betu;
    Csomopont *balNulla;
    Csomopont *jobbEgy;
    Csomopont (const Csomopont &);
    Csomopont & operator= (const Csomopont &);
};

Csomopont *fa;
int melyseg, atlagosszeg, atlagdb;
double szorasosszeg;
LZWBinFa (const LZWBinFa &);
LZWBinFa & operator= (const LZWBinFa &);

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

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

protected:

    Csomopont gyoker;
    int maxMelyseg;
    double atlag, szoras;

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

```
void rszoras (Csomopont * elem);  
  
};
```

Az osztály definálása után meghatározzuk a mélységet, átlagot, a szórást és a kiír függvényeket, amik megegyeznek a C programból ismertekkel. Ezeket a függvényeket rekurzívan valósítjuk meg.

```
int  
LZWBinFa::getMelyseg (void)  
{  
    melyseg = maxMelyseg = 0;  
    rmelyseg (&gyoker);  
    return maxMelyseg - 1;  
}  
  
double  
LZWBinFa::getAtlag (void)  
{  
    melyseg = atlagosszeg = atlagdb = 0;  
    ratlag (&gyoker);  
    atlag = ((double) atlagosszeg) / atlagdb;  
    return atlag;  
}  
  
double  
LZWBinFa::getSzoras (void)  
{  
    atlag = getAtlag ();  
    szorasosszeg = 0.0;  
    melyseg = atlagdb = 0;  
  
    rszoras (&gyoker);  
  
    if (atlagdb - 1 > 0)  
        szoras = std::sqrt (szorasosszeg / (atlagdb - 1));  
    else  
        szoras = std::sqrt (szorasosszeg);  
  
    return szoras;  
}  
  
void  
LZWBinFa::rmelyseg (Csomopont * elem)  
{  
    if (elem != NULL)  
    {  
        ++melyseg;  
        if (melyseg > maxMelyseg)
```

```
        maxMelyseg = melyseg;
        rmelyseg (elem->egyenesGyermekek ());
        rmelyseg (elem->nullasGyermekek ());
        --melyseg;
    }
}

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

void
LZWBinFa::rszoras (Csomopont * elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        rszoras (elem->egyenesGyermekek ());
        rszoras (elem->nullasGyermekek ());
        --melyseg;
        if (elem->egyenesGyermekek () == NULL && elem->nullasGyermekek () == NULL ↔
            )
        {
            ++atlagdb;
            szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));
        }
    }
}

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

int
```

```
main (int argc, char *argv[])
{
    if (argc != 4)
    {
        usage ();
        return -1;
    }

    char *inFile = *++argv;

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

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

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

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

    unsigned char b;
    LZWBinFa binFa;

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

    bool kommentben = false;

    while (beFile.read ((char *) &b, sizeof (unsigned char)))
    {
        if (b == 0x3e)
        {
            kommentben = true;
            continue;
        }

        if (b == 0x0a)
        {
            kommentben = false;
            continue;
        }
    }
```

```
    if (kommentben)
        continue;

    if (b == 0x4e)
        continue;

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

    }

    kiFile << binFa;

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

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

    return 0;
}
```

A program futtatásához bemenetként egy tömörített fájlra van szükség, ha sima txt-t adunk meg akkor nem lesz bináris fájl. Mivel a tömörített fájl eredménye nagyon hosszú lett (13-as mélységű), ezért ezt most nem teszem be.

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/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa/-binfa%20c++/z3a8.cpp

Ez a feladat az előző példánkkal teljesen megegyezik, azzal a különbséggel, hogy a destruktorba hozzáadtunk egy új sort: `delete gyoker`

```
~LZWBinFa ()
{
    szabadit (gyoker->egyGyermek ());
    szabadit (gyoker->nullasGyermek ());
}
```



```
delete gyoker;  
}
```

```
ch/binfa/binfa c++$ more output.txt  
-----1(2)  
-----1(1)  
-----0(2)  
-----1(5)  
-----1(4)  
-----0(3)  
-----0(4)  
---/(0)  
-----1(3)  
-----0(4)  
-----0(5)  
-----0(6)  
-----1(2)  
-----0(3)  
-----0(1)  
-----1(4)  
-----1(3)  
-----0(4)  
-----0(2)  
-----1(4)  
-----0(5)  
-----0(3)  
-----1(5)  
-----0(4)  
-----0(5)  
depth = 6  
mean = 4.3  
var = 1.1595
```

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa/binfa%20c++/z3a9.cpp

A Mozgató szemantikát elsősorban nagyobb méretű objektumok esetén szoktuk használni, segítségével kevesebb helyet foglalunk, mely gyorsabb programot eredményez. Lényege, hogy egy változó értékeit másolás helyett áthelyezzük egy másik vektorba, ekkor a régi objektum törlésre kerül, azaz elveszíti elemeit. A C++ nyelvben ezt a `move()` függvénnyel oldhatjuk meg. Az alap C++ Binfás programunkhoz képest annyi a változtatás, hogy a program végét kiegészítjük néhány sorral, az alábbiakkal:

```
LZWBinFa binFa3 = std::move ( binFa );  
  
kiFile << "depth = " << binFa3.getMelyseg () << std::endl;  
kiFile << "mean = " << binFa3.getAtlag () << std::endl;
```

```
kiFile << "var = " << binFa3.getSzoras () << std::endl;
```

Ha megfigyeljük a kódcipetet, akkor a forráskódban látható, hogy tulajdonképpen copy-paste a `kiFile` kiíró kódsorok, cska a változó nevét változtattuk meg `binFa3`-ra, ugyanis fentebb deklaráltunk egy ugyanilyen nevű változót, amihez hozzárendeltük a `move()` függvényt. Paraméterként a `binFa`-t adtuk meg.

```
LZWBinFa ( LZWBinFa && regi ) {  
    std::cout << "LZWBinFa move ctor" << std::endl;  
  
    gyoker.ujEgyesGyermek ( regi.gyoker.egyedGyermek() );  
    gyoker.ujNullasGyermek ( regi.gyoker.nullasGyermek() );  
  
    regi.gyoker.ujEgyesGyermek ( nullptr );  
    regi.gyoker.ujNullasGyermek ( nullptr );  
  
}
```

Programunkat kiegészítettük még a fenti kódcipettel is. Itt a `nullptr` kulcsszó a null pointert jelöli. Ez egy tetszőleges mutató típusra implicit konvertálódik. Erre azért van szükség, mert a 0 konstansnak két jelentése van: egyrészt 0 számot, másrészt pedig a null pointer.

7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk

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

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.2. Java életjáték

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

Megoldás videó:

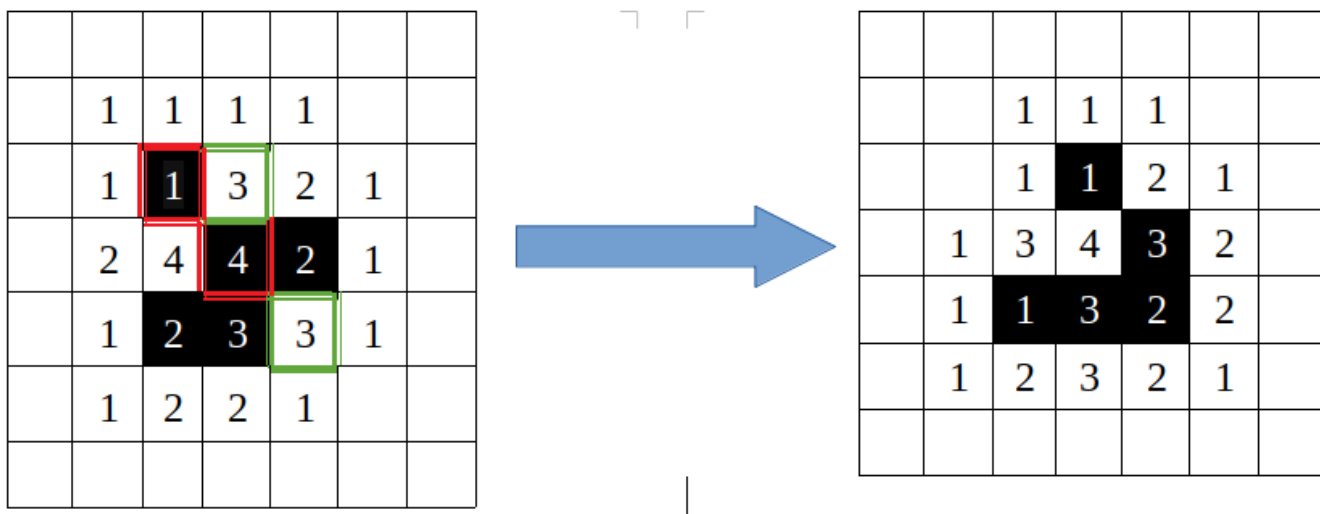
Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/conway/java-Sejtautomata.java

John Horton Conway a 70-es évek elején valósította meg az általa elnevezett Élejtájékot, melynek lényege, hogy van egy négyzethálós élettér, és minden cellában egy sejt élhet. négy keresztben, négy pedig átlósan. A szabályok pedig a következők:

- Ha egy sejtnak kettő vagy három szomszédja van: életben marad.
- Ha egy sejtnak négy vagy több szomszédja van: meghal. Ugyanez vonatkozik a 2-nél kisebb szomszédokkal rendelkezőkre.
- Ha egy üres cellának pontosan három élő sejt van a szomszédjában, akkor ott új sejt születik.

Innen az élet elnevezése, ugyanis a szimuláció generációról generációra mutatja meg a sejtek alakulását. Hogy könnyebb legyen elképzelni a fenti szabályokat, készítettem egy ábrát, ami egy lépést (azaz egy generáció születését) mutat be. Az alábbi ábrán a fekete háttérrel rendelkezők az élő sejtek, a piros szegéllyel

jelöltek jelentik a halálozást, mivel 2-nél kisebb és 3-nál több szomszédval rendelkeznek. Ezek a következő generációban tehát fehér háttérűek lesznek. Vannak még továbbá üres cellák, zöld szegéllyel, ezeknek pontosan 3 élő sejttel rendelkező szomszédjuk van, így a következő lépésben itt fognak új sejtek születni.



Ezt a szimulációt fogjuk megvalósítani java-ban, melynek neve `Sejtautomata.java`. Először is létrehozunk a `Sejtautomata` osztályt, majd azon belül deklaráljuk a változókat és értékeket rendelünk hozzá. Például megadunk két boolean típusú változót, mellyel azt fogjuk meghatározni, hogy egy sejt élő vagy halott lesz-e. Továbbá a rács elkészítéséhez fontos adatokat is itt deklaráljuk.

```
public class Sejtautomata extends java.awt.Frame implements Runnable {

    public static final boolean ÉLŐ = true;
    public static final boolean HALOTT = false;
    protected boolean [][][] rácscok = new boolean [2][][];
    protected boolean [][] rácsc;
    protected int rácscIndex = 0;
    protected int cellaSzélesség = 20;
    protected int cellaMagasság = 20;
    protected int szélesség = 20;
    protected int magasság = 10;
    protected int várakozás = 1000;
    private java.awt.Robot robot;
    private boolean pillanatfelvétel = false;
    private static int pillanatfelvételSzámláló = 0;
    /**
     * Létrehoz egy <code>Sejtautomata</code> objektumot.
     *
     * @param szélesség a sejttér szélessége.
     * @param magasság a sejttér magassága.
     */
    ...
}
```

Ezután létrehozunk a `Sejtautomata()` függvényt. Itt készítjük el a rácscokat a fenti adatok felhasználásával. Kezdetben a rács minden cellája halott lesz, ezért for ciklussal végigmegyünk először az egyes

cellákon és HALOTT azaz false értékeket rendelünk hozzá. Meghívjuk a siklóKilövőő() függvényt, melynek paraméterként a rácsot, a sor és az oszlop számát adjuk át. Lényege, hogy a sejtterbe ezzel helyezzük el a "sikló ágyút". Ez fog majd először megjelenni a programunk elindítása után, mely egy adott irányba fog elindulni. Az addWindowListener függvény figyel az ablakot, ha bezárjuk, akkor windowClosing függvény leállítja a programot is. Van még egy addKeyListener függvényünk is, amely a nevéből adódóan a billentyűzetleütéseket figyel. Itt megadjuk hogy, hogy ha a leütött billentyűzet megegyezik a programban megadottakkal, akkor végezze ez az utasításokat. Például 'K' leütése megfelel a sejtek méretét, 'N' billentyű lenyomásával pedig növeljük. Az 'S' billentyű pillanatfelvételt készít. Itt azért van felkiáltójel a pillanatfelvétel mögött, mert alapértelmezettként false van megadva, ezért ezzel true értéké alakítjuk át. A 'G' billentyűzettel megfelezzük az időt, azaz gyorsítjuk, végül pedig az 'L'-el fogjuk tudni lassítani, ha arra lenne szükségünk.

...

```
public Sejtautomata(int szélesség, int magasság) {
    this.szélesség = szélesség;
    this.magasság = magasság;

    rácsok[0] = new boolean[magasság][szélesség];
    rácsok[1] = new boolean[magasság][szélesség];
    rácsIndex = 0;
    rács = rácsok[rácsIndex];

    for(int i=0; i<rács.length; ++i)
        for(int j=0; j<rács[0].length; ++j)
            rács[i][j] = HALOTT;

    siklóKilövőő(rács, 5, 60);

    addWindowListener(new java.awt.event.WindowAdapter() {
        public void windowClosing(java.awt.event.WindowEvent e) {
            setVisible(false);
            System.exit(0);
        }
    });

    addKeyListener(new java.awt.event.KeyAdapter() {

        public void keyPressed(java.awt.event.KeyEvent e) {
            if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K) {

                cellaSzélesség /= 2;
                cellaMagasság /= 2;
                setSize(Sejtautomata.this.szélesség*cellaSzélesség,
                    Sejtautomata.this.magasság*cellaMagasság);
                validate();
            } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {

                cellaSzélesség *= 2;
                cellaMagasság *= 2;
```

```
        setSize(Sejtautomata.this.szélesség*cellaSzélesség,
                Sejtautomata.this.magasság*cellaMagasság);
        validate();
    } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)
        pillanatfelvétel = !pillanatfelvétel;
    else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G)
        várakozás /= 2;
    else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L)
        várakozás *= 2;
    repaint();
}
});

addMouseListener(new java.awt.event.MouseAdapter() {
    public void mousePressed(java.awt.event.MouseEvent m) {
        int x = m.getX()/cellaSzélesség;
        int y = m.getY()/cellaMagasság;
        rácsok[rácsIndex][y][x] = !rácsok[rácsIndex][y][x];
        repaint();
    }
});

addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
    public void mouseDragged(java.awt.event.MouseEvent m) {
        int x = m.getX()/cellaSzélesség;
        int y = m.getY()/cellaMagasság;
        rácsok[rácsIndex][y][x] = ÉLŐ;
        repaint();
    }
});

cellaSzélesség = 10;
cellaMagasság = 10;
try {
    robot = new java.awt.Robot(
        java.awt.GraphicsEnvironment.
            getLocalGraphicsEnvironment().
            getDefaultScreenDevice());
} catch(java.awt.AWTException e) {
    e.printStackTrace();
}
setTitle("Sejtautomata");
setResizable(false);
setSize(szélesség*cellaSzélesség,
        magasság*cellaMagasság);
setVisible(true);

new Thread(this).start();
}
...

```

Valahogy ki kellene rajzolni az élő sejteket is, ezért készítünk egy `paint()` függvényt. For ciklussal végigmenyünk a sorokon és az oszlopokon, ha az adott sor adott oszlopában van egy ÉLŐ sejt, akkor feketére színezzük, különben fehér lesz. Továbbá itt figyeljük azt is, hogy a pillanatfelvétel készítés aktiválva lett-e, ha igen, meghívjuk a pillanatfelvétel függvényt és készítünk egy képet továbbá visszaállítjuk `false` értékre, különben folyamatosan készítené a képeket, amit ugye nem szeretnénk, hogy megtörténjen.

```
...

public void paint(java.awt.Graphics g) {

    boolean [][] rács = rácsok[rácsIndex];

    for(int i=0; i<rács.length; ++i) {
        for(int j=0; j<rács[0].length; ++j) {

            if(rács[i][j] == ÉLŐ)
                g.setColor(java.awt.Color.BLACK);
            else
                g.setColor(java.awt.Color.WHITE);
            g.fillRect(j*cellaSzélesség, i*cellaMagasság,
                      cellaSzélesség, cellaMagasság);

            g.setColor(java.awt.Color.LIGHT_GRAY);
            g.drawRect(j*cellaSzélesség, i*cellaMagasság,
                      cellaSzélesség, cellaMagasság);

        }
    }

    if(pillanatfelvétel) {

        pillanatfelvétel = false;
        pillanatfelvétel(robot.createScreenCapture
            (new java.awt.Rectangle
              (getLocation().x, getLocation().y,
               szélesség*cellaSzélesség,
               magasság*cellaMagasság)));

    }

}

...
```

A `szomszédokSzama()` függvényben fogjuk megszámolni a szomszédokat az adott élő sejt körül. Itt is két for ciklus lesz, a sorok és az oszlopok miatt, azonban -1-től 1-ig fogjuk megvizsgálni, és csak akkor ha aza adott sejt nincs benne (`i==0` és `j==0`).

```
...

public int szomszédokSzama(boolean [][] rács,
    int sor, int oszlop, boolean állapot) {
    int állapotúSzomszéd = 0;
```

```

        for(int i=-1; i<2; ++i)
            for(int j=-1; j<2; ++j)
                if(!((i==0) && (j==0))) {

                    int o = oszlop + j;
                    if(o < 0)
                        o = szélesség-1;
                    else if(o >= szélesség)

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

                    if(rács[s][o] == állapot)
                        ++állapotúSzomszéd;
                }
        return állapotúSzomszéd;
    }
    ...

```

A bevezetőben ismertetett életjáték szabályokat az `időFejlődés()`-ben adjuk meg. If függvénnyel megvizsgáljuk, hogy az adott cella ÉLŐ-e? Ha igen, akkor a vizsgált élő cellának van-e 2 vagy 3 ÉLŐ szomszédja, ha ez teljesül, akkor az ezt követő rácsban is élő marad az adott pozícióban. Ellenkező esetben halott lesz. Ha a vizsgált cella nem élő akkor lépünk az else ágba. If függvénnyel itt azt vizsgáljuk meg, hogy van-e pontosan 3 szomszédja, ha igen, akkor ez is élő lesz, ellenkező esetben halott.

```

...

public void időFejlődés() {

    boolean [][] rácsElőtte = rácsok[rácsIndex];
    boolean [][] rácsUtána = rácsok[(rácsIndex+1)%2];

    for(int i=0; i<rácsElőtte.length; ++i) { // sorok
        for(int j=0; j<rácsElőtte[0].length; ++j) { // oszlopok

            int élők = szomszédokSzáma(rácsElőtte, i, j, ÉLŐ);

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

```



```
        /* Halott halott marad, ha három élő
           szomszédja van, különben élő lesz. */
        if(élők==3)
            rácsUtána[i][j] = ÉLŐ;
        else
            rácsUtána[i][j] = HALOTT;
    }
}
rácsIndex = (rácsIndex+1)%2;
}
...
```

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/conway/c++

A program felépítése nagyon hasonlít a Mandelbrot nagyítós fejezetnél megismert programhoz, és a futtatása is ugyanúgy történik (qmake és make parancsok használata). Itt is 5 különálló fájlból fog állni a programunk. A main.cpp-ben adjuk meg, hogy mekkora legyen az ablakunk mérete. Jelen esetünkben 100x75 pixel méretű lesz. `w.show()` függvénnyel hívjuk elő a az ablakot.

```
#include <QApplication>
#include "sejtablak.h"
#include <QDesktopWidget>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    SejtAblak w(100, 75);
    w.show();

    return a.exec();
}
```

A `sejtablak.h` fájlban megadjuk az ablakon belüli rácsokat, melynek mérete megegyezik a az ablakunk méretével. Továbbá az ELO és HALOTT boolean típusú változónk konstans változók lesznek, azaz ezt a későbbiekben a programon belül ezek értékeit nem fogjuk tudni megváltoztatni. A `protected`-en belül pedig további változókat deklarálunk.

```
#ifndef SEJTABLAK_H
#define SEJTABLAK_H

#include <QMainWindow>
```

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

class SejtSzal;

class SejtAblak : public QMainWindow
{
    Q_OBJECT

public:
    SejtAblak(int szelesseg = 100, int magassag = 75, QWidget * ←
        parent = 0);

    ~SejtAblak();
    static const bool ELO = true;
    static const bool HALOTT = false;
    void vissza(int racsIndex);

protected:
    bool ***racsok;
    bool **racs;
    int racsIndex;
    int cellaSzelesseg;
    int cellaMagassag;
    int szelesseg;
    int magassag;
    void paintEvent(QPaintEvent*);
    void siklo(bool **racs, int x, int y);
    void sikloKilovo(bool **racs, int x, int y);

private:
    SejtSzal* eletjatek;

};

#endif // SEJTABLAK_H
```

sejtszal.h

```
#ifndef SEJTSZAL_H
#define SEJTSZAL_H

#include <QThread>
#include "sejtablak.h"

class SejtAblak;

class SejtSzal : public QThread
{
    Q_OBJECT
```

```
public:
    SejtSzal(bool ***racso, int szelesseg, int magassag,
             int varakozas, SejtAblak *sejtAblak);
    ~SejtSzal();
    void run();

protected:
    bool ***racso;
    int szelesseg, magassag;
    int racsIndex;
    int varakozas;
    void idoFejlodes();
    int szomszedokSzama(bool **racs,
                       int sor, int oszlop, bool allapot);
    SejtAblak* sejtAblak;

};

#endif // SEJTSZAL_H
```

A sejtablak.cpp fájlban a SejtAblak() függvénnyel az ablakkal kapcsolatos adatokat adjuk meg. Ez 3 paramétert vár: szélesség, magasság és egy pointert, ami a QWidget "parent" paraméteréhez tartozik. Ennek értéke 0. A setWindowTitle() azt jelenti, hogy a zárójelben megadott szöveg lesz majd az ablakunk fejlécében látható cím. A setFixedSize() -ban a szélességet és a magasságot megszorozzuk 6-al, majd ez alapján for ciklusokkal létrehozunk két rácsot. A paintEvent-en belül végiglépkedünk a sorokon és az oszlopokon és ha az adott cellában van egy ELO, akkor feketére színezzük a QT segítségével. Ellenkező esetben fehér lesz.

```
#include "sejtablak.h"

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

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

    cellaSzelesseg = 6;
    cellaMagassag = 6;

    setFixedSize(QSize(szelesseg*cellaSzelesseg, magassag*cellaMagassag));

    racso = new bool**[2];
    racso[0] = new bool*[magassag];
    for(int i=0; i<magassag; ++i)
        racso[0][i] = new bool [szelesseg];
```

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

    racsIndex = 0;
    racs = racsok[racsIndex];

    for(int i=0; i<magassag; ++i)
        for(int j=0; j<szelesseg; ++j)
            racs[i][j] = HALOTT;

    sikloKilovo(racs, 5, 60);

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

    eletjatek->start();
}

void SejtAblak::paintEvent(QPaintEvent*) {
    QPainter qpainter(this);

    bool **racs = racsok[racsIndex];
    for(int i=0; i<magassag; ++i) {
        for(int j=0; j<szelesseg; ++j) {
            if(racs[i][j] == ELO)
                qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                                   cellaSzelesseg, cellaMagassag, Qt::black);
            else
                qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                                   cellaSzelesseg, cellaMagassag, Qt::white);
            qpainter.setPen(QPen(Qt::gray, 1));

            qpainter.drawRect(j*cellaSzelesseg, i*cellaMagassag,
                               cellaSzelesseg, cellaMagassag);
        }
    }
    qpainter.end();
}

SejtAblak::~SejtAblak()
{
    delete eletjatek;

    for(int i=0; i<magassag; ++i) {
        delete[] racsok[0][i];
        delete[] racsok[1][i];
    }
}
```

```
delete[] racsok[0];
delete[] racsok[1];
delete[] racsok;

}

void SejtAblak::vissza(int racsIndex)
{
    this->racsIndex = racsIndex;
    update();
}

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

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

}

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

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

    ...
}
```

Most nézzük meg a `sejtszal.cpp` fájlt. Itt fogjuk megszámolni a szomszédokat, és megadjuk a szomszédok száma alapján, hogy hol lesz élő és halott cella. Ennek a felépítése megegyezik a java programunkban ismertetett módszerrel.

```
#include "sejtszal.h"

SejtSzal::SejtSzal(bool ***racsok, int szelesseg, int magassag, int ←
    varakozas, SejtAblak *sejtAblak)
{
    this->racsok = racsok;
    this->szelesseg = szelesseg;
    this->magassag = magassag;
    this->varakozas = varakozas;
    this->sejtAblak = sejtAblak;

    racsIndex = 0;
```

```
}

int SejtSzal::szomszedokSzama(bool **racs,
                              int sor, int oszlop, bool allapot) {
    int allapotuSzomszed = 0;
    for(int i=-1; i<2; ++i)
        for(int j=-1; j<2; ++j)

            if(!((i==0) && (j==0))) {

                int o = oszlop + j;
                if(o < 0)
                    o = szelesseg-1;
                else if(o >= szelesseg)
                    o = 0;

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

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

void SejtSzal::idoFejlodes() {

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

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

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

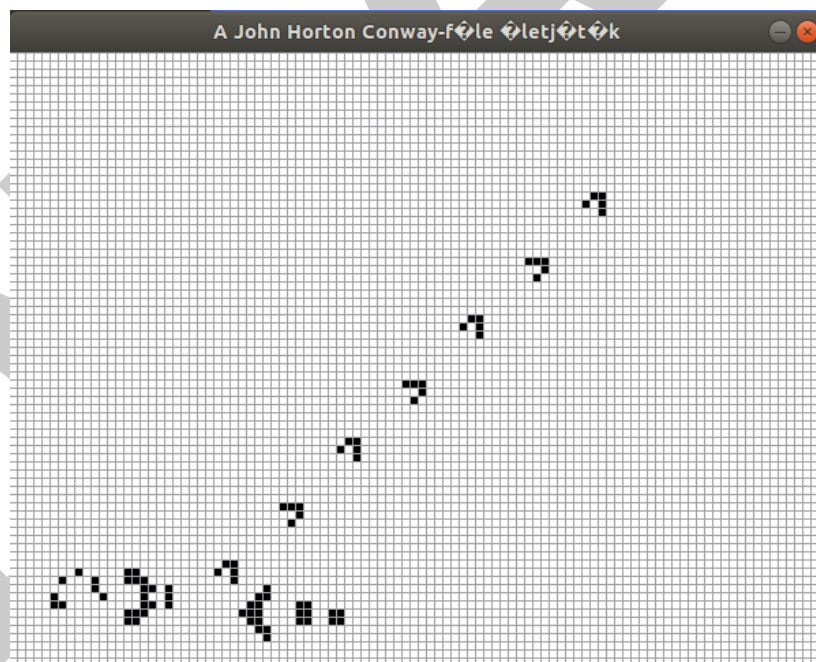
            if(racsElotte[i][j] == SejtAblak::ELO) {

                if(elok==2 || elok==3)
                    racsUtana[i][j] = SejtAblak::ELO;
                else
                    racsUtana[i][j] = SejtAblak::HALOTT;
            } else {

                if(elok==3)
                    racsUtana[i][j] = SejtAblak::ELO;
                else
                    racsUtana[i][j] = SejtAblak::HALOTT;
            }
        }
    }
}
```

```
        }  
    }  
    }  
    racsIndex = (racsIndex+1)%2;  
}  
  
void SejtSzal::run()  
{  
    while(true) {  
        QThread::msleep(varakozas);  
        idoFejlodes();  
        sejtAblak->vissza(racsIndex);  
    }  
}  
  
SejtSzal::~~SejtSzal()  
{  
}
```

A program futtatása a következő: a qmake -project létrehoz egy .pro végződésű fájlt, amibe bele kell írni, hogy QT += widgets. Majd a parancssorba beírjuk hogy qmake *.pro, ami létrehoz egy Makefile-t. make paranccsal bináris fájlt hozunk létre, majd futtatjuk a szokásos g++ programot. Ha mindent jól csináltunk, a következő képernyőt kell kapnunk:



7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

aa Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.2. Szoftmax R MNIST

R

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.4. Deep dream

Keras

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.5. Robotpszichológia

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó:

Megoldás forrása:

9.2. Weizenbaum Eliza programja

Éleszd fel Weizenbaum Eliza programját!

Megoldás videó:

Megoldás forrása:

9.3. Gimp Scheme Script-fu: króm effekt

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

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

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

Tanulságok, tapasztalatok, magyarázat...

9.4. Gimp Scheme Script-fu: név mandala

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

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

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

Tanulságok, tapasztalatok, magyarázat...

9.5. Lambda

Hasonlítsd össze a következő programokat!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9.6. Omega

Megoldás videó:

Megoldás forrása:

DRAFT

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

[?]

1. ALAPFOGALMAK A programozási nyelveknek három szintjét különböztethetjük meg: gépi nyelv, assembly szintű nyelv és magas szintű nyelv. A magas szintű nyelven megírt programot forrásprogramnak nevezzük. Ezek összeállítására vonatkozó formai szabályok összességét pedig szintaktikai szabályoknak. A tartalmi, értelmezési, jelentésbeli szabályok alkotják a szemantikai szabályokat. Ez a két szabály együttese határozza meg a magas szintű programozási nyelvet. Ahhoz, hogy az adott processzor végre tudja hajtani a programokat, szükség lesz fordító programokra vagy interpreterre, amik lefordítják a processzor saját gépi kódú nyelvére. Az a különbség a fordítóprogramok és az interpreterek között, hogy az utóbbi nem készít tárgyprogramot, hanem utasításokként értelmezi a forrásprogramot és rögtön végre is hajtja azokat. Egyébként mindkét program végez lexikális (forrásszöveg feldarabolása lexikális egységekre), szintaktikai (Teljesülnek-e az adott nyelv szintaktikai szabályai?) és szemantikai elemzés. Ahhoz, hogy egy programot helyesen írjunk meg, át kell tanulmányozni az adott programnyelvek szabványait (hivatkozási nyelv). A legnagyobb probléma a programozási nyelvekkel kapcsolatban, az a hordozhatóság, ami az egyik magas szintű programnyelvből egy másik programnyelvre történő implementációja, ugyanis ez a technika jelenleg nem megoldott. A programnyelvek osztályozása háromféleképpen történik: imperatív nyelvek, deklaratív nyelvek és máselví (egyéb) nyelvek. Az imperatív nyelvekhez tartoznak az algoritmikus nyelvek, a program utasítások sorozata. Legfőbb programozói eszköz a változó, amihez értékeket rendelhetünk, manipulálhatjuk az adatokat, tehát a tár közvetlen elérése. Ez a nyelv szorosan kötődik a Neumann-architektúrához. Két alcsoportja van: Eljárásorientált- és objektumorientált nyelvek. Ezzel szemben a deklaratív nyelvek nem algoritmikus nyelvek, nem kötődnek szorosan a Neumann architektúrához. A programozó csak a problémát adja meg, és nincs lehetősége memóriaműveletekre. Alcsoportjai a funkcionális- és logikai nyelvek. A másnyelvű(egyéb) nyelvekhez pedig azok tartoznak, amiket nem lehet besorolni a fenti két nyelv valamelyikébe.

2. Adattípusok

Az adattípus a programozási nyelvekben egy absztrakt programozási eszköz. A típuslétrehozás úgy történik, hogy megadjuk a tartományát, a műveleteit és a reprezentációját. Vannak esetek, amikor a saját típust a beépített és a már korábban definiált saját típusok segítségével adjuk meg. Nem minden programozási nyelv ismeri ezt az eszközt, ezért ennek megfelelően elkülöníthetünk típusos és nem típusos nyelveket. Egy adattípus három dolgot határoz meg: tartomány, műveletek, reprezentáció. A tartomány tartalmazza azokat

az elemeket, amelyeket az adott programozási eszköz felvehet értékként. Egyszerű típus lehet például a C nyelvben int típus, ami az egész számokra vonatkozik. Ott van továbbá a char, azaz a karakteres típus, amelynek elemei karakterek. De létezik még logikai, felsorolásos, sorszámozott típus stb. Az összetett típusokhoz tartoznak például a tömbök, amik lehetnek statikus vagy homogén összetett típus. attól függően, hogy a tömb elemeiben milyen típusú értékek fordulnak elő. Továbbá ide tartozik még a rekord típusú programozási eszköz is. Fontos még a mutató típus, amelyek tartományának elemei lényegében tárcímek, tehát az adott eszköz a tár adott területét címzi. A nevesített konstans olyan programozási eszköz, amelynek három komponense van: név, típus, érték. A C programban a nevesített konstans így kell elképzelni: #define név literál. Lényege, hogy előre definiáljuk a konstansok értékeit és a programban bárhol hivatkozhatunk rá a konstans nevének használatával, így betöltjük az ott lévő adatokat. Továbbá, ha meg akarjuk változtatni az adatot, akkor elég csak a deklarációs részben megváltoztatni, így az egész programban érvényes lesz a megváltoztatott érték. A változónak négy komponense van: név, attribútumok, cím, érték. A név egy azonosító, ahogy a nevesített konstansoknál is van. Az attribútumok pedig a változók futás közbeni viselkedését határozzák meg, például ha típust rendelünk hozzá (int, char stb). A cím a tárnak azt a részét határozza meg, ahol a változó értéke elhelyezkedik. Az értéket a típussal összhangban kell megadni. A C nyelv típusrendszere lehet aritmetikai és származtatott típusok. Az aritmetikai típusok tartományának elemeivel aritmetikai műveletek végezhetők:

- egész típus: int szam = 5
- karakter típus: char szoveg = 'valami'
- felsorolás: enum szinek { VOROS=11, NARANCS=9, SARGA=7, ZOLD=5, KEK=3, IBOLYA=3};
- valós (float, double, long double)

Származtatott típusok lehetnek:

- tömb: int tomb[5]
- mutató: int *mut
- függvény: fgv()
- struktúra: STRUCT [struktúrátípus_név] {mező_deklarációk} [változólista];
- union: UNION [uniontípus_név] {mező_deklarációk} [változólista];

3. Kifejezések

A kifejezések szintaktikai eszközök. Lényege, hogy ha a programban már ismert értékek segítségével új értékeket határozunk meg. Formálisan operandusokból, operátorokból és kerek zárójelekből állnak. A kifejezésnek három alakja lehet: pre-, in- és postfix. A prefix esetben az operátor az operandusok előtt áll (* 3 5). Infix-nél az operátor középen (3 * 5). Postfix-nél pedig az operandusok mögött áll az operátor (3 5 *). A C egy kifejezésorientált nyelv. Itt a kifejezéseket rekurzívan építjük fel, tehát a kifejezések egyszerű kifejezésekből épülnek fel, zárójelek és operátorok használatával, majd ezt az egészet lexikális egységek zárják le. Az aritmetikai típusoknál a típuskényszerítés elvét vallja, ami azt jelenti, hogy egy két operandusú operátornak különböző típusú operandusai lehetnek, de a műveletek csak azonos típusok között végezhetők

el. Bizonyos esetekben konverzió van, tehát a nyelv megvizsgálja, hogy milyen típuskombinációk megengedettek.

4. Utasítások

Az utasítások olyan program egységek, amelyekkel megadhatjuk az algoritmusok egyes lépéseit, továbbá a fordítóprogramok ezek segítségével generálja a tárgykódot. Két csoportból áll: deklarációs és végrehajtható utasítás. A deklarációs utasításnál nincs tárgykód, nem kerülnek lefordításra. Ez csak a fordítóprogramoktól kérnek valamilyen szolgáltatás, vagy olyan információkat ad át a fordítóprogramnak, amik szükségesek a tárgykód generálásánál. A végrehajtható utasításokból készül el a tárgykód a fordítóprogram által. Ezek tovább csoportosíthatók:

- értékadó utasítás: változó értékének beállítása a program futása során.
- üres utasítás: üres gépi utasítás
- ugró utasítás: feltétel nélküli vezérlés átadó utasítás során egy adott címkével ellátott végrehajtható utasításra adhatjuk át a vezérlést. Használt alakja: GOTO címke
- elágaztató utasítás:

Kétirányú elágaztató utasítás: ebben az esetben a program két tevékenység közül választ és a választott tevékenységet végrehajtja (vagy ellenkezőleg). A feltételes utasítás alakja: IF feltétel THEN tevékenység [ELSE tevékenység] Ha nem szerepel az ELSE-ág, akkor hosszú alakról van szó, ellenkező esetben rövid utasítás. Működése úgy zajlik, hogy előbb kiértékelődik a feltétel. Ha ez igaz, akkor az ott lévő tevékenység végrehajtott. Ha a feltétel hamis és van else ág, akkor az ott lévő feltétel is kiértékelődik. Szót kell még ejteni a „csellengő ELSE” problémáról is, ami akkor fordul elő, amikor egy rövid if utasításba be van ágyazva egy hosszú if utasítás is:

```
IF ... THEN
    IF ... THEN
        ELSE ...
```

A kérdés az, hogy vajon melyik IF feltételhez tartozik az ELSE-ág? A probléma egyszerűen kiküszöbölhető, ha egy hosszú utasításba ágyazzuk be a hosszú utasítást, úgy hogy a külső ELSE-ágba üres utasítás szerepel. De némely programnál nincs szükség erre, ugyanis az implementáció többsége azt mondja, hogy az értelmezés belülről kifelé történik, tehát minden egyes ELSE-ágot belülről kifelé párosítjuk az adott IF utasításokhoz.

Többirányú elágaztató utasítás: Itt a program egymást kölcsönösen kizáró akárhány tevékenység közül egyet választ. C esetében úgy néz ki a dolog, hogy először kiértékelődik a kifejezés, ami egy integer típusú számnak kell lennie. Ez összehasonlításra kerül a CASE ágak értékeivel. Ha van egyezés akkor az adott ponton végrehajtott a tevékenység. Ha nincs egyezés, akkor a DEFAULT ággal lépünk ki:

```
SWITCH (kifejezés) {
    CASE egész_konstans_kifejezés : [ tevékenység ]
    [ CASE egész_konstans_kifejezés : [tevékenység ]]...
    [ DEFAULT: tevékenység ]
};
```

- ciklusszervező utasítás: Ez az utasítás lehetővé teszi, hogy egy bizonyos tevékenységet akárhányszor megismételjünk. Felépítése: fej, mag, vég. Működés szerint 2 típusa van: az egyik az üres ciklus, amikor a mag egyszer sem fut le, a másik pedig a végtelen ciklus, amikor az ismétlődés soha nem áll le. Kezdőfeltételes ciklus esetében a feltétel a fejben jelenik meg. Tehát Először kiértékelődik a feltétel, Ha igaz, végrehajthó a ciklusmag, majd újra kiértékelődik a feltétel egészen addig, amíg hamis nem lesz. Ilyen a WHILE(feltétel) végrehajthó_utasítás. Végfeltételes ciklus esetében a feltétel a végben van, de vannak nyelvek amelyekben a fej tartalmazza azt. Tehát először végrehajthó a mag és csak utána vizsgálja meg a feltételt. DO végrehajthó_utasítás WHILE(feltétel); Előírt lépésszámú ciklus: Itt a fejben van a feltétel. Tartalmaz egy változót (ciklusváltozó) és a változó által felvett értékekre fut le a ciklus. FOR([kifejezés1]; [kifejezés2]; [kifejezés3]) végrehajthó_utasítás
- hívó utasítás
- vezérlésátadó utasítás
- I/O utasítás
- egyéb utasítások

5. Programok szerkezete

Az alprogramok olyan programozási eszközök, amelyeket akkor használunk ha egy programon belül ugyanaz a programrész többször megismétlődik, így nem kell minden esetben újra megírni a programot, hanem elég csak hivatkoznunk az adott alprogramra. A formális alprogram felépítése a következő: fej vagy specifikáció, törzs vagy implementáció és vég. Ez a programozási eszköz négy komponensből áll: név, formális paraméter lista, törzs, környezet. A név lesz az azonosító, tehát mindig ennek használatával tudunk majd hivatkozni. A formális paraméter listában azonosítók szerepelnek, melyek aktuális értékei a hívás helyén szerepelnek. A formális paraméter lista általában a név mellett zárójelben szerepel. Dönthetünk úgy is, hogy nem adunk meg paramétert, mert az adott alprogramnak nincs szüksége rá. Ekkor ez egy paraméter nélküli alprogram lesz. Ez a két komponens az alprogramban mindig a fejben szerepelnek. A törzsben deklarációs és végrehajthó utasításokat adunk meg. Az alprogram környezete a globális változók együttese. Globális nevek alatt azt értjük, hogy az alprogramon kívül neveket deklaráltunk, amik elérhetőek az alprogram számára. Van ugyanis lokális nevek is, amik csak az alprogramban érhető el és a külső környezet számára el van rejtve. Az alprogramnak két típusa van: eljárás és függvény. Az eljárás általában a paraméterek vagy a környezetének megváltoztatását, vagy a törzsben elhelyezett utasítások végrehajtását jelenti. Egy eljárás akkor nevezhető szabályosnak, ha a program futtatása közben elérjük a végét és külön utasítással befejeztetjük. Míg a függvények általában valamilyen értéket adnak vissza. Egy függvényt meghívni csak kifejezésben lehet, ennek alakja: függvénynév(aktuális_paraméter_lista). A blokk egy olyan programegység, amely csak egy másik programegységben helyezkedhet el, tehát külső szinten nem. Ezt a programegységet csak az eljárásorientált nyelvek egy része ismeri. Szerepe a nevek hatáskörének elhatárolása. A blokknak nincs paramétere, de lehetnek a törzsében végrehajthó vagy deklarációs utasítások és úgy lehet elkezdni, hogy szekvenciálisan kerül rá a vezetés vagy egy GOTO-utasítással ráugrunk a kezdetére.

6. Paraméterek

Paraméterkiértékelés során először mindig a formális paraméter lista lesz az elsődleges, majd ezután rendeljük hozzá az aktuális paramétereket. Annak módszerét, hogy melyik formális paraméterhez, melyik

aktuális paraméter fog tartozni, megkülönböztethetjük sorrendi kötés vagy név szerinti kötés szerint. Sorrendi kötésnél, mint ahogy a nevében is benn van, felsorolás sorrendjében rendelődnek hozzá. A név szerinti kötésnél pedig, az aktuális paraméter listában megadjuk a formális paraméter lista nevét és mellette pedig valamilyen szintaktikával az aktuális paramétert. A paraméter átadás is különböző módon történhet (érték-, cím-, eredmény-, érték-eredmény-, név-, szöveg szerint). Érték szerinti paraméterátadás

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

10.3. Programozás

[BMECPP]

A C és C++ nyelv közötti különbségek

Ha egy függvénynek nem adunk meg paramétert, akkor tetszőleges számú paraméterrel hívhatjuk meg. A C++ nyelvben meg kell adnunk a void-ot, aminek jelentése pontosan az, hogy nincs a függvénynek paramétere:

```
// C üres paraméter
void f() {
    //az F függvény törzse
}

//C++ üres paraméter
void f(...) {
    //Az f függvény törzse
}
```

A függvénynevek azonosítása is eltérően működik a két nyelvben. Amíg a C-nél maga a neve az azonosítója, így nem lehet két azonos nevű függvény, addig a C++ nyelvben a függvényeket a nevük és argumentumlistájuk együttesen azonosítja, így fordulhatnak elő azonos nevű függvények. A fordító ezt úgy oldja meg, hogy a függvényneveket kiegészíti a az argumentumtípus elő- vagy utótagjával, így a linker szintjén különböző nevekké jelennek meg. Ezt nevezzük névelferdítésnek. Ahhoz, hogy a C és a C++ közötti együttműködés, emiatt ne legyen problémás, használnunk kell az extern "C" deklarációt. Ez lehetővé teszi, hogy a már C fordítóval lefordított függvényeket felhasználhassuk a C++ programunkban is.

```
//Példa C++ nyelvben azonos azonosítónevű, de eltérő ↔
argumentumú függvények
void PrintTime(int hour, int minute) {
    ...
}

int PrintTime(int hour, int minute) {
    ...
}
```

C++ nyelvben van lehetőség alapértelmezett értékeket megadni a függvények argumentumában. Szerepe az, hogy függvényhívásnál megfelelő elemszámú/típusú paramétereket adjunk át. Fontos különbség még, hogy a C++ nyelvben van referenciatípus szerinti paraméterátadás is. Ez feleslegessé teszi a pointereket a cím szerinti paraméterátadásnál:

```
void f(int& i) {  
    i = i + 2;  
}  
  
int main(void) {  
    int a = 0;  
    f(a);  
    print("%\n", a);  
}
```

A fenti programban látható, hogy a referencianeve elé egy & jellel deklaráltuk a referenciát.

III. rész

Második felvonás

DRAFT

**Bátf41 Haxor Stream**

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

DRAFT

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

DRAFT

11.3. Általános

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

11.4. C

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

11.5. C++

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

11.6. Lisp

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

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

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