Univerzális programozás

Programkód vadászok

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

KÖZREMÜKÖDTEK			
	CÍM : Univerzális programozás		
HOZZÁJÁRULÁS	*NÉV*	*DÁTUM*	*ALÁÍRÁS*
ÍRTA	Bátfai Norbert és Lovász Botond	2019. március 24.	

VERZIÓTÖRTÉNET			
VERZIÓ	DÁTUM	LEÍRÁS	NÉV
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	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

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?

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

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

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

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

A DocBook XML 5.1 új neked?
Ez esetben forgasd a https://tdg.docbook.org/tdg/5.1/ könyvet, a végén találod az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

Bátf41 Haxor Stream
A feladatokkal kapcsolatos élő adásokat sugároz a https://www.twitch.tv/nbatfai csatorna, melynek permanens archívuma a https://www.youtube.com/c/nbatfai csatornán található.

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

```
#include <stdio.h>

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

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

int main ()
{
for (;;) {
sleep (1);
printf("Vegtelen ciklus\n");
}
return 0;
}
```

```
#include <unistd.h>

int main()
{
int a1,a2,a3;
if(!(a1=fork()))
{
  for(;;);
}
if(!(a2=fork()))
{
  for(;;);
}
if(!(a3=fork()))
{
  for(;;);
}
for(;;);
}
```

Megoldás videó:

Megoldások forrása:

https://github.com/lovaszbotond/Turing/blob/master/Vegtelen%20ciklus

https://github.com/lovaszbotond/Turing/blob/master/Vegtelen%20ciklus2%20(0%25)

https://github.com/lovaszbotond/Turing/blob/master/Vegtelen%20ciklus%203%20(4%20100%25)

A feladatunk az volt , hogy írjunk három végtelen ciklust melyek 1 mag 0%-os,1 mag 100%-os illetve 4 mag 100%-os igénybevételét követelte. Az 1 mag 100%-os megoldásnál háromféleképpen is el lehetett volna készíteni a programunkat viszont mi a 'for' ciklust választottuk ami számomra a legszimpatikusabb volt, de kiválóan működött volna a (while(1){}/while(true){}) vagy a (do {} while(1) / do {} while(true)) programkód is.Jelen esetben nem adtunk meg tömböt, hogy honnan kezdjen számolni (minek a függvényében) meddig menjen és mekkorákat.Így a program megállás nélkül tud futni , és nem áll le míg mi nem adunk meg egy erre megfelelő parancsot.Ez esetben 1 szálat használ és az pörgeti 100%-on a processzort.

A következő esetben 0%on szeretnénk használni , amit úgy oldottunk meg, hogy a sleep parancsot vettük elő melynek a paraméterét másodpercben kell számolnunk. Mi az '1'-et adtuk meg neki .Ekkor a program alvó állapotba kerül és a magasabb prioritású programok előnyt élveznek vele szemben.Amint nem lesz olyan futó folyamat melyre érvényesülne a korábban leírt kijelentésem , a program feléled.

A harmadik verzióban mind a négy szálunkat 100%-on szeretnénk dolgoztatni.Létrehoztunk 3 változót.Feltételnek készítünk egy gyerek programot melyet igazzá téve egy végtelen ciklusba teszünk bele, 1 változónkat felhasználva.A másik kettővel is így teszünk , a 4. esetben már erre nincs szükségem.Nem adunk meg neki utasítást melyre ki léphetne vagy le állhatna a programunk. Miután le generáltuk , lehet látni , hogy mindegyik szál 100%-os terheltségen dolgozik . Remélem lehet érteni amit , írtam . Én örülök , hogy jobban elmélyedtem a ciklusok világában, és együtt tanulhattunk egy újabb érdekességet.

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 vlgtelen ciklus:

```
Program T100
{

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

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

```
Program T1000
{

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

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

  main(Input Q)
  {
    Lefagy2(Q)
  }

}
```

Mit for kiírni erre a `T1000(T1000)` futtatásra?

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

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

Értelmezzük tehát mi is történik. A T100 as program kap egy másik programot bemenetként , hogy döntse el le fog e fagyni , vagy nem fog lefagyni a program. Megnézi észlel-e benne végtelen ciklust. Ha észlel akkor igaz ha nem akkor hamis értékkel tér vissza.A T1000-es program szintén reprezentálja a korábbiakat melyre ha igaz értéket ad megáll ha hamisat akkor végtelen ciklusba kerül.Az ellentmondása a programnak , ha saját magára kell tesztett futtatnia , azaz ha nincs végtelen ciklus a programban akkor végtelen ciklusba kerül , ha pedig van benne végtelen ciklus akkor megfog állni. Jól látható, hogy ellentmondásba ütköztünk aminek a következménye , hogy ezért nem sikerült még megvalósítanunk ilyen programot. Alan Turing nevéhez fűződik egyébként ez a probléma aki a XX. század az egyik legmeghatározóbb hanem a legmeghatározóbb angol matematikusa/modern számítógép-tudomány királya. Számomra legjelentősebb munkássága , az Enigma feltörése volt , mely majdhogynem 2 évvel rövidítette meg a második világháborút.

2.3. Változók értékének felcserélése

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

```
  #include <stdio.h>

  int main()
  {
```

```
    int x = 10, y = 5;

printf("x=%d, y=%d\n" , x, y);

    x = x + y;
    y = x - y;
    x = x - y;

    printf("x=%d, y=%d\n" , x, y);


    return 0;
}
```

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

Megoldás forrása:https://github.com/lovaszbotond/Turing/blob/master/Valtozocsere

A program esetében mivel , nem használhatunk logikai utasításokat , matematikai háttértudásunkat kell elő bányásznunk.A megoldáshoz szimpla matematikát használtam , azaz :

x=10,y=5

x = 10 + 5 = 15

y = 15 - 5 = 10

x = 15 - 10 = 5

Láthatjuk , hogy a megoldás során a két kezdeti értéket sikerült fel cserélnünk , logikai utasítás nélkül. Sokan a XOR műveletet szokták ilyenkor felhasználni , ami szintén helyes megoldása lehet a feladatnak . (XOR -> kizáró vagy , bitekkel valú művelet, egyforma terjedelmű bitek sorozatain végezi el a műveletet. Ha a bitek nem egyeznek meg az adott helyen akkor 1-et ha megegyeznek 0-át ad vissza.)

Példa:

```
    #include <stdio.h>

int main()
{
    int x = 10, y = 5;

printf("x=%d, y=%d\n" , x, y);

    y = x ^ y;
    x = x ^ y;
    y = x ^ y;

    printf("x=%d, y=%d\n" , x, y);


    return 0;
}
```

Ha a programot lefuttatjuk , észlelhetjük , hogy a két változó ismét felcserélődött.

Tehát :

x = 1100

y = 0111

Akkor y = 1100 ^ 0111 = 1011

x = 1100 ^ 1011 = 0111

y = 0111 ^ 1011 = 1100

Így kell valójában elképzelnünk ahogy a XOR művelet működik.

1 - 1 -> 0

0 - 0 -> 0

1 - 0 -> 1

0 - 1 -> 1

Ismét kiemelve , fontos , hogy a művelet , azonos terjedelmű bitsorozatokon működik helyesen !

2.4. Labdapattogás

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

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

int
main ( void )
{
    WINDOW *ablak;
  ablak = initscr ();

    int x = 0;
    int y = 0;

    int deltax = 1;
    int deltay = 1;

    int mx;
    int my;

    for ( ;; ) {

      getmaxyx ( ablak, my , mx );
```

```
        mvprintw ( y, x, "@" );

        refresh ();
        usleep ( 100000 );

        x = x + deltax;
        y = y + deltay;

        if ( x>=mx-1 ) {
            deltax = deltax * -1;
        }
        if ( x<=0 ) {
            deltax = deltax * -1;
        }
        if ( y<=0 ) {
            deltay = deltay * -1;
        }
        if ( y>=my-1 ) {
            deltay = deltay * -1;
        }

      }

      return 0;
  }
```

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

int
main (void)
{
    int xj = 0, xk = 0, yj = 0, yk = 0;
    int mx, my;

    WINDOW *ablak;
    ablak = initscr ();
    noecho ();
    cbreak ();
    nodelay (ablak, true);

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

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

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

        refresh ();
        usleep (150000);

    }
    return 0;
}
```

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

Megoldás forrása:https://github.com/lovaszbotond/Turing/blob/master/Labdapattoga20if%20nelkul

A feladat megoldásánál két lehetséges opciót vizsgáltunk , oldottunk meg . Az első az if logikai utasítást felhasználva készült a második az if elhagyásával a for ciklusba belenyúlva valósult meg.

Include-olva a megfelelő csomagokat **curses.h , unistd.h** el is kezdhetünk dolgozni a feladat megoldásán. Az **ablak = initscr ();** megadja a futtatás környezetének megfelelően az adott ablakunknak az adatait , amelybe utána szépen sorban elkezdhetem be táplálni a rendelkezésre álló terünknek az információit melyek alapján tökéletesen fog tudni dolgozni a program és pattogni a labda.Az x->oszlop | y->sor | deltax->lépéstáv(oszlop) | deltay->lépéstáv(sor) | mx->oszlopok száma | my->sorok száma

Láthatjuk korábbról , hogy a for ciklusunk egy végtelen ciklus lesz melyben a printf"curses" által kiterjesztett parancsait használjuk . **void getmaxyx(WINDOW *win, int y, int x);** itt utalunk vissza a **WINDOW *ablak**-ra ahol átadjuk paraméterként az ablakban eltárolt értékeket és hogy hol tárolja el az ablakunk szélességét/hosszúságát.A **mvprintw(y,x,"@");** megadja az adott ablakban használt karakteremet , mit is szeretnék majd látni a monitoromon, illetve felhasználja a meglévő koordinátákat ahol fogja mozgatni a karaktert. A **refresh()** az aktuális kimenetunk a terminalban. A **usleep()** alapértelmezett értelemben mikrosecundumban számol ami a másodperc egy milliomod része , melyet egy tizedre állítunk a programunkban . Minél kisebbre állítjuk ezt az értéket annál gyorsabban fog haladni a kijelzőnkön az adott jelöléssel ellátott objektumunk.

Az if részlegben négy különböző feltételt szabunk meg . Elérte a jobb oldalt ? Elérte a bal oldalt? Elérte a tetejét ? Elérte az alját? - kérdéseket tehetnénk fel melyek rávezetnek a feladat megoldására. A jobb oldal esetében az oszlopokhoz felhasznált "mx" lesz a segítségünkre. Azaz a maximális oszlopszám -1-nél ha nagyobb vagy egyenlő az x, akkor megszorozva -1-el , elindítjuk a labdánkat visszafele. Míg a bal oldal esetében ha az x 0-nál kisebb vagy egyenlő , akkor ugye -1*-1 +1 lesz így elindítjuk ellentétes irányba.A tetejét tekintve az y érték fog segíteni. Ahogyan korábban is említettük , ha az y kisebb vagy egyenlő 0-nál akkor -1-el való szorzás után pozitívot varázsolva belőle elindíthatjuk vele ellentétes irányba. Alját tekintve pedig mint ahogy az oszlopok maximális számánál is csináltuk a sorok maximális értékénél szorzom be -1-el , így visszafordítva irányát újra a helyes irányt veszi fel kijelzőnkön.

A második felének a feladatban ahogy írtuk korábban is "if"-ek nélkül kell dolgoznom , melyhez a "for" ciklusba kell belenyúlnom.A "%" a maradékos osztásnak a jele , melyre szükségünk van ahoz hogy megoldjuk

a feladatot.Mindig egészen addig kell osztanunk a modulót , míg vissza nem adja annak a számnak az értékét melyet elosztunk egészen addig , míg egyenlő nem lesz az ablakunk hosszúságával/szélességével.Ekkor vissza áll 1/-1-re aminek a következtében elkezd visszafele pattogni a karakterünk.Mivel egy végtelen ciklusban vagyunk benne ezért egy véget nem érő pattogásról beszélünk , egészen addig míg manuálisan mi magunk le nem lőjük a programunkat.

Szemfüles olvasóink , kiszúrhatták , hogy a "for" cikluson kívül , még 3 függvényt is meghívunk , ahoz , hogy tökéletesen működjön a programunk.A **noecho();** segítségével , kikapcsoljuk a karakterünk visszhangját. A tty driver echo systemét inaktiváljuk. A tty driver folyton buffereli a karaktersorozatomat egészen addig amíg új sor nem lesz vagy a szállítási értéke 0 nem lesz. A **cbreak();** alap esetben törli,megszakítja a karakterfolyamatot. A **nodelay();** hívásra kerül , és nem állít be időzítőt. Az időtullépés célja , hogy hogy a funkcióbillentyűtől kapott és a felhasználó által beírt szekvenciák között kiszűrje a különbséget.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

```
#include <stdio.h>

int main()
{
    int a=1,i=1;
      while(a>0)
      {
      a<<=1;
      i++;
      }
      printf("%d",i);
    return 0;
}
```

Megoldás videó:

Megoldás forrása: https://github.com/lovaszbotond/Turing/blob/master/Gepiszo

A gépi szó , int típusú változója 4 byte-on tárolódik , ezért 32-őt fog vissza adni,ha 8 byte-on tárolódna akkor 256 hosszúságú szót kapnánk vissza eredményül. A bitshift operátorral dolgozunk, azon belül is a left shift operátort alkalmazzuk. A while ciklusban haladunk balra shiftelunk 1 et. Addig növeljük , amíg végig nulla bitet nem fog tartalmazni az "a" változonk. Az "i" változónk tárolja a lépéseknek a darabszámát.(32)

2.6. Helló, Google!

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

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

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

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

double
tavolsag (double PR[], double PRv[], int n)
{
  double osszeg = 0.0;
  int i;

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

  return sqrt(osszeg);
}

int
main (void)
{

  double L[4][4] = {
    {0.0, 0.0, 1.0 / 3.0, 0.0},
    {1.0, 1.0 / 2.0, 1.0 / 3.0, 1.0},
    {0.0, 1.0 / 2.0, 0.0, 0.0},
    {0.0, 0.0, 1.0 / 3.0, 0.0}
  };

  double PR[4] = { 0.0, 0.0, 0.0, 0.0 };
  double PRv[4] = { 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0 };

  int i, j;

  for (;;)
    {

      for (i = 0; i < 4; ++i)
  {
    PR[i] = 0.0;
    for (j = 0; j < 4; ++j)
      PR[i] += (L[i][j] * PRv[j]);
  }
```

```
      if (tavolsag (PR, PRv, 4) < 0.00000001)
  break;

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

    }

  kiir (PR, 4);

  return 0;
}
```

Megoldás videó:

Megoldás forrása: https://github.com/lovaszbotond/Turing/blob/master/Pagerank

A PageRank egy olyan algoritmus amellyel weboldalak relatív fontosságát lehet megállapítani. Azt adja meg, hogy véletlenszerű böngészés esetén mekkora az esélye annak, hogy az adott oldalra találunk. Alapja,hogy egy oldalon minden hivatkozás egy-egy "szavazat" a hivatkozott oldalra. Az alapján meg lehet állapítani egy oldal relatív fontosságát, hogy hány az oldalra mutató hivatkozás van a többi oldalon, illetve, hogy hány oldalra hivatkozik az adott oldal. Az algoritmusban egy jobb minőségű oldal "szavazata" erősebbnek számít, mint egy kis relatív fontosságúé. Mivel a felhasználó általában nem fogja végignézni az összes linket a weboldalon, ezért bevezettek a képletbe egy csillapító faktort.

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

```
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")
```

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

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

A Brun tétel az ikerprímszámokkal foglalkozik . Ahoz hogy tudjunk az ikerprímszámokról beszélni , még előtte ismertetnünk kell pár alap definíciót.

A prímszám : Azok a természetes számok , melyeknek pontosan két osztójuk van a természetes számok között - > 1 és önmaga.

Ikerprím:Egymást követő prímszámok különbsége pontosan 2 lesz. Példa: (3-5) (5-7) (11-13) (101-103)

Az alapvető probléma amit pedzegetünk , illetve a Brun tétel is erre hajaz , hogy a prímszámok a végtelenbe tartanak , akkor talán az ikerprímeknek is a végtelenbe kellene hogy tartsanak . Viszont mikor elkezdjük számolni , azt vesszük észre , hogy az ikerprímek inkább konvergálnak egy szám felé . Az elején észleljük , hogy nagyok a különbségek , viszont ahogy haladunk előre és egyre több prímszámot vizsgálunk meg és adunk össze egyre kisebb eltérést tapasztalhatunk. Ha a koordináta rendszeren ezt ábrázolnom kellene, akkor egy észrevehető éles görbülés után , már szabad szemmel ki mernénk jelenteni , hogy konstans ,pedig nem az. Ez az eltérés irracionális értelemben a végtelenségig csökken és a feltételezésünktől nem lesz eltérő.

Nézzük is meg , hogyan működik a program.

A `primes(x)` vektor mögötti zárójelen belül megadom , hogy mekkora is legyen a vektorom , hány darab prímszámot tápláljak bele.

A `diff` egy új vektor lesz amely az ikerprímeket fogja nekem csak megjeleníteni. A**diff = primes[2:length(prim**
primes[1:length(primes)-1] sorban adom meg , hogyan is adom meg az ikerprímeket. Primes - > prímeken belül-> []-zárójelben adom meg az indexet, 2-es , hogy a másodiktól számolja, "length" pedig hogy végéig a prímeknek, tehát az egész hosszát vegye.Az `idx` megmondja hanyadik helyen lesz a a különbség 2.A t1primes illetve a t2primes egyértelmű. Csinálunk két azonos 1 tömbös vektort, ennek hála átláthatóbb lesz ha külön egymás alatt szeretném vizsgálni őket oszloposan. A t2 primes-hoz azért adunk hozzá 2-őt , mert úgy fogom megkapni a t1 primes közvetlen ikerpárját. A `rt1plust2` a t1primes illetve a t2primes reciprokait adja össze. A `return`-ben a "sum" segítségével összegezzük az rt1plust2-ben kapott összeget. A végén pedig kirajzoltatjuk a függvényünket.

A programot lefuttatva látjuk , hogy ha az "x" helyére minél nagyobb számot írunk be , akkor az ikerprímeink összege egyre kisebb lesz és konvergál egy szám felé . Ezen tapasztalataim következtében nem jelenteném ki , hogy ez a szám a végtelenbe tart, maximum hogy a végtelenségig csökken a rá következő számmal alkotott különbségük.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

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)]

}

nemvaltoztatesnyer= which(kiserlet==jatekos)
valtoztat=vector(length = kiserletek_szama)

for (i in 1:kiserletek_szama) {

    holvalt = setdiff(c(1,2,3), c(musorvezeto[i], jatekos[i]))
    valtoztat[i] = holvalt[sample(1:length(holvalt),1)]

}

valtoztatesnyer = which(kiserlet==valtoztat)


sprintf("Kiserletek szama: %i", kiserletek_szama)
length(nemvaltoztatesnyer)
length(valtoztatesnyer)
length(nemvaltoztatesnyer)/length(valtoztatesnyer)
length(nemvaltoztatesnyer)+length(valtoztatesnyer)
```

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 Monty Hall probléma/paradoxon egy valószínűségi paradoxon , ami az Amerikai Egyesült Államokban futott "Let's Make Deal" című televíziós vetélkedő utolsó feladatán alapul, nevét a vetélkedő műsorvezető-jéről "Monty Hall"-ról kapta.

A műsor végén a játékosnak mutatnak három csukott ajtót, amelyek közül kettő mögött egy-egy kecske van, a harmadik mögött viszont egy vadonatúj autó. A játékos nyereménye az, ami az általa kiválasztott ajtó mögött van. Azonban a választás meg van egy kicsit bonyolítva. Először a játékos csak rámutat az egyik ajtóra, de mielőtt valóban kinyitná, a műsorvezető a másik két ajtó közül kinyit egyet, amelyik mögött nem az autó van (a játékvezető tudja, melyik ajtó mögött mi van), majd megkérdezi a játékost, hogy akar-e módosítani a választásán. A játékos ezután vagy változtat, vagy nem, végül kinyílik az így kiválasztott ajtó, mögötte a nyereménnyel. A paradoxon nagy kérdése az, hogy érdemes-e változtatni, illetve hogy számít-e ez egyáltalán.

A kérdésre a válasz: igen számít. Amikor elkezdődik a játék 1/3-ad esélyem van hogy a jó ajtót választottam és nyerek , 2/3-ad esélyem van a bukásra. Miután a műsorvezető kinyit egy ajtót 1/2 az esélye , hogy helyes amit választottam matematikailag.Viszont ha mögé nézünk egy kicsit jobban láthatjuk , hogy az eredetileg 0.33%al bíró önmagam miután kinyílt egy ajtó a műsorvezetőnek hála gyarapodott 0.33% ot , ami azt jelenti, hogy megéri változtatnom hisz arra , hogy a megfelelő ajtót választom már 0.66% az esélyem.Még így is elképzelhető , hogy bukok , viszont ha végijátszunk a gondolattal,hogy mindhárom esetet megvizsgálva , hogyan járok jobban , látszik , hogy 2/3 az esélye annak , hogy nyerek és 1/3 ad az esélye annak , hogy elbukok mindent és hazaviszem a kecskét legelni.Ha könnyűsúlyú vagyok még őt is meglovagolom .. De egy brand new 2007-es BMW E92 M3-asnak inkább hallgatom a búgását.

A programunk működése:

Alapvetően véletlen eshetőségeket szeretnénk generálni, majd eldönteni a program segítségével , valóban igaz-e amit írtunk korábban a matematikai háttérről. A kísérletek száma egyértelmű, a zárójelben bármi szerelephet, amit beírunk , annyiszor próbálkozunk. A kísérlet vektor 1-2-3 közül választ , a játékos vektor szintén , a műsorvezető vektornak a hossza pedig a kisérletek száma.A for ciklussal végighaladok minden egyes vektoron és megnézem , ha a kísérlet egyenlő a játékossal, akkor jók vagyunk és nyertünk. A "mibol" vektornál 1-2-3-ból kiveszem a kísérletet az if-en belül else ágon pedig kiveszi a kisérletet+játékost.A műsorvezető a "mibol" hossza lesz. Ha nem talalta el akkor a műsorvezető nyer. Van egy másik águnk is a "nemvaltoztatesnyer" ahol szintén ha a kisérlet==játékos akkor megvan a kincs és nyertünk.A "valtoztat" a kisérletek mennyiségével lesz egyenlő ami a hosszt illeti. A folytatásban ha valtoztat kivesz kettőt szintén mint korábban a program. Kiértékeljük , hogy hányszor nyertünk változtatással és nélküle majd a végén ki írtajuk a kapott eredményeket.

3. fejezet

Helló, Chomsky!

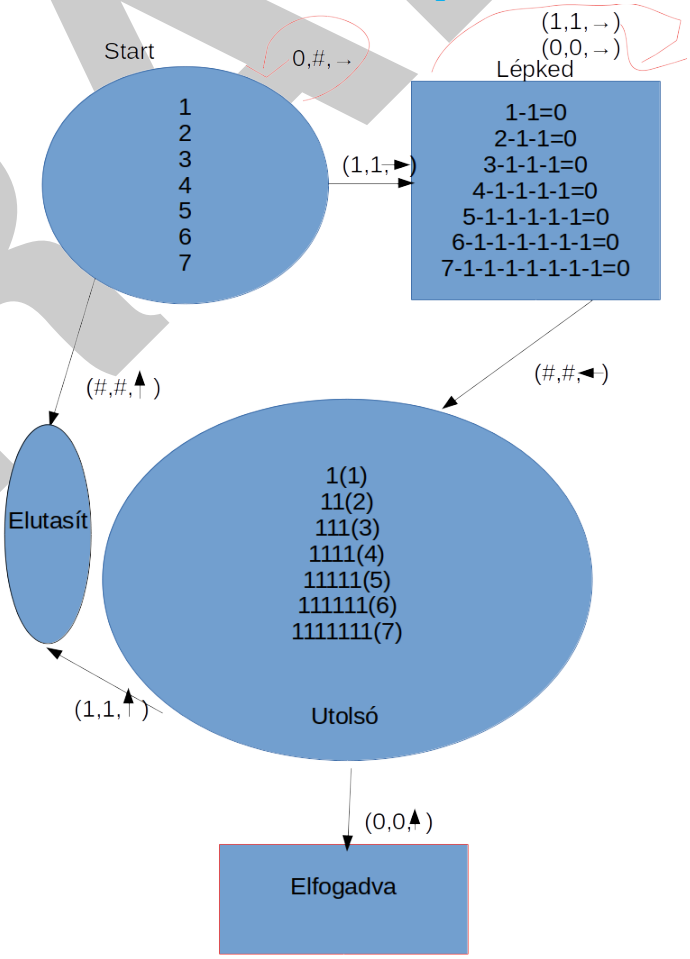
3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: https://github.com/lovaszbotond/Chomsky/blob/master/images/

Turin1.png



A Turing gép három nagyobb fizikai részből áll:

Egy író-olvasó fejből- ez a szimbólumokkal foglalkozik.(írja vagy olvassa)

A végetelen szalag formályában létező részekre osztott memóriából.

Valamint a gép programját tartalmazó vezérlőegységből.

A Turing gép alapvető működési elve , mikor decimálisból váltunk unárisba , hogy a meglévő számomból folyamatosan egyeseket vonunk ki, és ezeket a levont egyeseket tároljuk.Maga az unáris rendszerben való ábrázolás x darab egyforma jel/karakter egymás utáni leírásával történik . Az "x" a decimális szám.

3.2. Az $a^nb^nc^n$ nyelv nem környezetfüggetlen

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

1. PÉLDA

Kezdő szimbólumok / változók : S, Z, X

szimbólumok / konstansok : a, b, c

Szabályok: S->abc , S->aZScc , Z->aXa , Xa->aZ , Za->aabb

```
S->aZScc->aZabccc->aaabbbccc
```

2. PÉLDA

Kezdő szimbólumok / változók : S, X, Y

szimbólumok / konstansok : a, b, c

Szabályok: S->abc, S->aXaYS, Yab->bcc , Xa->aabb

```
S->aXaYS->aXaYabc->aXabccc->aaabbbccc
```

Megoldás videó:

Megoldás forrása:https://github.com/lovaszbotond/Chomsky/blob/master/Generat%C3%ADv

A formális nyelvek és formális nyelvtanok vizsgálatának egyik legjelentősebb úttörője Noam Chomsky, akinek a munkássága egyaránt hatott a formális nyelvek és a természetes nyelvek kutatására is.

A generatív nyelv az a legismertebb kategória, amely azoknak a szabályoknak a halmaza , amelyekkel a nyelben minden lehetséges jelsorozat előállítható, azaz hogy egy átírási eljárással ,hogyan is állíthatjuk elő a kitüntetett kezdő szombólumból a többi jelsorozatot , a szabályok egymás utáni alkalmazásával. Ezek variációja véges számű lehet , fent 2 példán át próbáljuk megmutatni , hogyan is kell érteni , alkalmazni ezt a nyelvtant.

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása: https://github.com/lovaszbotond/Chomsky/blob/master/C89-C99

Egy nyelv vezérlésátadó utasításai az egyes műveletek végrehajtási sorrendjét határozzák meg. Jelenleg a szerkezetekről fogunk tárgyalni .

Több utasítás fajtánk is van a C programnyelvünkben melyeknek típusai is vannak.

Ezek :

```
*kifejezés utasítás
*összetett utasítás
*iterációs utasítás
*vezérlésátadó utasítás
*kiválasztó utasítás
*cimkézett utasítás
```

Kifejezés :

```
x = 0, i++ vagy printf(...)
x=0;
i++;
printf(...)
```

Utasítássá válik ha egy pontosvesszőt írunk utána, ez az utasítás lezáró jel.A {} kapcsos zárójelekkel deklarációk és utasítások csoportját fogjuk össze egyetlen összetett utasításba vagy blokkba, ami szintaktikailag egyenértékű egyetlen utasítással.

If-Else utasítás: Döntés kifejezésére használjuk.

```
if \\(kifejezés)
\\1.utasítás
else
\\2.utasítás
```

Ahol az else rész az opcionális.Ha igaz a kiértékelés akkor az első utasítás, ha nem igaz akkor az else ág hajtódik végre.Az else mindig a hozzá legközelebb eső, else ág nélküli if utasításhoz tartozik.Ha nem így szeretnénk, akkor a kívánt összerendelés kapcsos zárójelekkel érhető el.

```
if (n > 0) {
   if (a > b)
      z = a;
}
else
   z = b;
```

Az Else-If utasítás:

```
if \\(kifejezés)
   \\utasítás
else if (kifejezés)
   \\utasítás
else if (kifejezés)
   \\utasítás
else if \\(kifejezés)
   \\utasítás
.
.
else
   \\utasítás
```

Ez a szerkezet adja a többszörös döntések általános szerkezetét. Agép sorra kiértékeli a kifejezéseket és ha bármelyik ezek közül igaz, akkor végrehajtja a megfelelő utasítást, majd befejezi az egész vizsgáló láncot.

A Switch utasítás:

A switch utasítás is a többirányú programelágaztatás egyik eszköze.Úgy működik, hogy összehasonlítja egy kifejezés értékét több egész értékű állandó kifejezés értékével, és az ennek megfelelő utasítást hajtja végre.

```
\\Általános felépítés
witch \\(kifejezés)
 {
   case \\állandó kifejezés: utasítások
   case \\állandó kifejezés: utasítások
   .
   .
   default: \\utasítások
}
```

Mindegyik case ágban egy egész állandó vagy állandó értékű kifejezés található, és ha ennek értéke megegyezik a switch utáni kifejezés értékével, akkor végrehajtódik a case ágban elhelyezett egy vagy több utasítás.Az utolsó, default ág akkor hajtódik végre, ha egyetlen case ághoz tartozó feltétel sem teljesült.A default és case ágak tetszőleges sorrendben követhetik egymást , viszont ha elhagyjuk a default ágat , azaz nincs, akkor nem hajtődik végre semmi sem.

```
#include <stdio.h>

main( ) /* számok, üres helyek és mások számolása */
{
   int c, i, nures, nmas, nszam[4];

   nures = nmas = 0;
   for (i = 0; i < 4; i++)
      nszam[i] = 0;
   while ((c = getchar( )) != EOF) {
      switch (c) {
         case '0': case '1': case '2':  case '3':
```

```
            nszam[c-'0']++;
            break;
         case ' ':
         case '\n':
         case '\t':
            nures++;
            break;
         default:
            nmas++;
            break;
      }
}
printf("számok =");
for (i = 0; i < 4; i++)
   printf(" %d", nszam[i]);
printf(", üres hely = %d, más = %d\n", nures, nmas);
return 0;
}
```

A break utasítás hatására a vezérlés azonnal abbahagyja a további vizsgálatokat és kilép a switch utasításból.Az egyes case esetek címkeként viselkednek, és miután valamelyik case ág utasításait a program végrehajtotta, a vezérlés azonnal a következő case ágra kerül, hacsak explicit módon nem gondoskodunk a kilépésről.

While - For utasítás:

```
while \\kifejezés
  \\utasítás
```

A program először kiértékeli a kifejezést.Ha annak értéke nem nulla (igaz), akkor az utasítást végrehajtja, majd a kifejezés újra kiértékelődik. Ez a ciklus mindaddig folytatódik, amíg a kifejezés nullává (hamissá) nem válik, és ilyen esetben a program végrehajtása az utasítás utáni helyen folytatódik.

```
for \\(1. kifejezés; 2. kifejezés; 3. kifejezés)
   \\utasítás
 \\ami teljesen egyenértékű a while utasítással megvalósított
\\1. kifejezés
while \\(2. kifejezés) {
   \\utasítás
   \\3. kifejezés
}
```

Ciklusszervezés do-while utasítással:

A do-while utasítás a ciklus leállításának feltételét a ciklusmag végrehajtása után ellenőrzi, így a ciklusmag egyszer garantáltan végrehajtódik.

```
do
   \\utasítás
while \\(kifejezés);
```

A gép először végrehajtja az utasítást és csak utána értékeli ki a kifejezést. Ez így megy mindaddig, amíg a kifejezés értéke hamis nem lesz, ekkor a ciklus lezárul és a végrehajtás az utána következő utasítással folytatódik.

A break és continue utasítások:

A break utasítás lehetővé teszi a for, while vagy do utasításokkal szervezett ciklusok idő előtti elhagyását, valamint a switch utasításból való kilépést.A break mindig a legbelső ciklusból lép ki.

A continue utasítás a break utasításhoz kapcsolódik, de annál ritkábban használjuk. A ciklusmagban található continue utasítás hatására azonnal (a ciklusmagból még hátralévő utasításokat figyelmen kívül hagyva) megkezdődik a következő iterációs lépés.

A goto utasítás és a címkék:

A C nyelvben a goto utasítás, amellyel megadott címkékre ugorhatunk.A goto használatának egyik legelterjedtebb esete, amikor több szinten egymásba ágyazott szerkezet belsejében kívánjuk abbahagyni a feldolgozást és egyszerre több, egymásba ágyazott ciklusból szeretnénk kilépni.

```
for(...)
   for(...) {

      if (zavar)
         goto hiba;
   }
hiba:
   \\a hiba kezelése
```

A szerkezetben előnyös a hibakezelő eljárást egyszer megírni és a különböző hibaeseteknél a vezérlést a közös hibakezelő eljárásnak átadni, bárhol is tartott a feldolgozás.A címke ugyanolyan szabályok szerint alakítható ki, mint a változók neve és mindig kettőspont zárja.

A feladatunk kérte, hogy mutassunk be olyan kódcsipeteket, melyek bizonyos esetekben lefordulhatnak, más esetekben viszont nem. C89/C99 -->

```
#include <stdio.h>

    int main ()
    {
      // Print string to screen.
      printf ("Hello World\n");
return 0;
    }
// C89-C99 es hiba / pipa
```

Az egészet egy egyszerű Hello World-ön is be lehet mutatni. A **"gcc -std=c89"** esetében nem fordul le a program mert a komment nem támogatott míg a**"gcc -std=c99"**esetében hibátlanul lefordul. Tökéletesen látszik ha lefuttatjuk, hogy valóban nem mindegy hogyan is fordítunk. A standard ANSI egyébként a C11 a C programnyelvben.

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása:https://github.com/lovaszbotond/Chomsky/blob/master/Elemzo

```
/*** definíciós rész ***/

%{
/* Ez a kód bemásolódik a generált C forrásba*/
#include <stdio.h>
int valos=0;
%}

/* Ez az opció azt mondja meg, hogy csak egy input file kerüljön ↩
   beolvasásra. */
%option noyywrap

%%
 /*** Szabályok ***/


[:digit:]+  { valos++; }
[A-Za-z][A-Za-z0-9]*       {   /* Minden más karaktert ignorálunk. */   }

%%
/*** C kód. Ez is bemásolódik a generált C forrásba. ***/

int main()
{
    /* Meghívjuk az elemzőt, majd kilépünk.*/
        yylex();
  printf("%d valós számot talált a lexer: \n",valos);
    return 0;
}
```

A kommentek alapján lehet olvasni mi is történik a programban.A lex sajátosságait látni, hogyan is épül fel a szerkezete.

Első körben a függvénykönyvtár behívása a fontos amit szeretnék használni a stdoutra való ki íratásra->"%{%}". Ezután jön a számláló amit a lexer észlel(számok). A második etapot a %%..%% jelöli.Itt adjuk meg a szabályokat.A `[[:digit:]]+`megad két vagy több számot egymás után és ha ez megtörtént akkor növelünk egyet. A `[A-Za-z][A-Za-z0-9]*` az összes alfanumerikus karakterláncot jelöli. Ez egy tipikus kifejezés a számítógép nyelvén található azonosítók felismerésére. A `main` függvényben a lexert hívjuk segítségül ami bájtonként haladva végigmegy a bemeneten.A lexer elő idézéséhez a `yylex` segít.Amint a lexer lefutott , ki írom az eredményt amit kaptunk a számláló által.A return 0 jelzi , a rendszer felé , hogy a program véget ért.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó: https://youtu.be/2E2tJf5yyTc

Megoldás forrása:

```
/*
  Ez egy lex kód. A fordításhoz először "lex leet.c" parancsot kell kiadni, ↩
      amely a gcc által fordítható forráskódot fog generálni.
*/
%{
  #include <stdio.h>
  #include <ctype.h>
  #include <string.h>
  #include <time.h>
  #include <stdlib.h>
  int x=0;
  typedef struct{
    char c;
    char *d[7];
  } cipher;
  cipher L337[] = {
    {'a',{"4","/-\\","a","/\\","a","a","a"}},
    {'b',{"!3","|3","8","ß","b","b","b"}},
    {'c',{"[","<","{","c","c","c","c"}},
    {'d',{"|}","d","|>","T)","d","d","d"}},
    {'e',{"3","&","e","€","e","e","e"}},
    {'f',{"|=","|#","/=","f","f","f","f"}},
    {'g',{"&","6","g","(_+","g","g","g"}},
    {'h',{"|-|",")-(","[-]","h","h","h","h"}},
    {'i',{"1","[]","!","|","i","i","i"}},
    {'j',{"_|",";","1","j","j","j","j"}},
    {'k',{">|","1<","|c","k","k","k","k"}},
    {'l',{"1","|_","l","|","l","l","l"}},
    {'m',{"/\\/\\","/V\\","[V]","m","m","m","m"}},
    {'n',{"</>","n","|\\|","^/","n","n","n"}},
    {'o',{"0","Q","o","<>","o","o","o"}},
    {'p',{"|*","|>","p","|7","p","p","p"}},
```

```
    {'q',{"(_,)","q","9","&","q","q","q"}},
    {'r',{"I2","|?","Iz","r","r","r","r"}},
    {'s',{"s","5","z","§","s","s","s"}},
    {'t',{"4","-|-","7","t","t","t","t"}},
    {'u',{"(_)","u","v","L|","u","u","u"}},
    {'v',{"v","\\/","|/","\\|","v","v","v"}},
    {'w',{"\\/\\/","w","\\x/","\\\\//\\\\//","w","w","w"}},
    {'x',{"4","}{","><","x","x","x","x"}},
    {'y',{"y","j","`/","\\|/","y","y","y"}},
    {'z',{"2","-/_","z",">_","z","z","z"}},

    {'1',{"I","1","L","I"}},
    {'2',{"R","2","2","Z"}},
    {'3',{"E","3","E","3"}},
    {'4',{"4","A","A","4"}},
    {'5',{"S","5","S","5"}},
    {'6',{"b","6","G","6"}},
    {'7',{"7","7","L","T"}},
    {'8',{"8","B","8","B"}},
    {'9',{"g","q","9","9"}},
    {'0',{"0","()","[]","0"}},
  };
%}
%option noyywrap
%%
\n  {
    printf("\n");
  }
.   {
    srand(time(0)+x++);
      char c = tolower(*yytext);

      int i=0;
      while(i<36 && L337[i++].c!=c);
      if(i<36)
      {
        char *s=L337[i-1].d[rand()%7];
        printf("%s",s);
      }
    else
    {
      printf("%c",c);
    }

  }
%%

int main()
{
  yylex();
```

```
  return 0;
}
```

A feladat megoldásánál mesterem és segítőm volt Petrus József Tamás! A lex első részében a program által látható függvénykönyvtárak include-jai láthatók. Ezután egy int tí- pusú változó létrehozása áll, amely a random számok generálásának beállításához szükséges. A program működésének alapja a cipher típusú tömb létrehozása, ami a különböző betűkhöz és számokhoz tartozó lehetséges leet kódokat tartalmazza, többnyire három kódolt betűt és 4 "eredeti" betűt, hogy kisebb eséllyel legyen minden betű átalakítva. A kód következő részében minden a lexer által beolvasott karakterre megnézi a program, hogy benne van-e a cipher típusú tömb kódolandó karakterei között. Ha megtalálja, akkor ahhoz a karakterhez tartozó egyik kódolást véletlenszerűen kiválasztja, majd a standard kimenetre kiírja. Ha nem találta meg, akkor az eredeti karaktert kiírja a standard kimenetre. A main függvényben a program meghívja a yylex függvényt, azaz magát a lexert. Ha a lexer futása véget ér, akkor a program 0-val tér vissza, amely azt jelzi az operációs rendszernek, hogy a program futása sikeresen véget ért.

3.6. A források olvasása

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

```
if(signal(SIGINT, jelkezelo)==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 jelkezelo függvény kezelje. (Miután a **man 7 signal** lapon megismertem a SIGINT jelet, a **man 2 signal** lapon pedig a használt rendszerhívást.)

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

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


void jelkezelo(int sig)
{
  printf("Off %d\n",sig);
}
int main () {
for(;;){
  if(signal(SIGINT, jelkezelo)==SIG_IGN)
     signal(SIGINT, SIG_IGN);
  }
return 0;
}
```

A **#include signal.h** csomagra szükségünk van , hogy meg tudjuk hívni a megfelelő parancsokat melyek kellenek a feladat megoldásához.

Létrehozzuk a "jelkezelő" függvényünket , melyet később fogunk majd érvényesíteni. A lényege , hogy amint befut a jel , megérkezik , ki írja a terminálunkra, vagy a környezetre amelyen lefuttatjuk , hogy "OFF" jelen esetben.

A `main` -en belül , van egy for ciklusunk mely egy végtelen ciklus. Az `if` -en belül , láthatjuk , hogy a `SIG_IGN` signal ignorance -t vizsgáljuk. Ha a SIGINT jelkezelése nincs elutasítva akkor innentől fogva `jelkezelo` végezze a kezelést. Egészen amíg a jelet mi nem közvetítjük neki egy " CATCH CTRL-C EFFECT"-en kersztül , addig a program csak egy végtelen ciklusban fut , viszont amint megérkezik , a jelkezelő függvényünknek hála , jelet elkapja , és ki is írja az előbb említett "OFF"-ot. Ez az alap programunk , most pedig vizsgáljuk meg a további kódcsipeteket , melyeket felírtunk. A csipetek alatt tovább elemezzük mivel egészül ki a program , vagy miben is változik meg, esetleg hibásak-e.

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

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

A korábban leírt alap programunkat , elvégezzük ötször.Változás , hogy a for ciklusunk már nincs végtelenítve. A végeredményünk nem változik. Az "i" eredménye 5 az utolsó vizsgálatnál.A pre increment miatt (++i). A jelkezelő szempontjából ez nem számít.

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

Semmit nem változott az előző kódcsipethez képest , kivéve , hogy post increment (i++) azaz az "i" eredménye 4 az utolsó viszgálatnál.A jelkezelő szempontjából viszont ez nem számít.

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

A for cikluson belül a tömb első öt elemét mindenütt eggyel növeljük. Viszont a tomb[i] = i++ kifejezés hiba lehet mert a program végrehajtási sorrendje nem megfelelően definiált,más fordító programot vagy másik számítógépet használva hibát/más eredményt kaphatunk.

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

Ez a for ciklus 0-tól n-ig tart és amire az "s" pointer mutat annak az értékét hozzárendeli ahoz a memóriához ahova "d" pointer mutat, a pointereket egyel lépteti, és minden iteráció végén növeli "i" értékét egyel.

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

Ki íratunk két egészet amiknek meghatározója a `f(a, ++a), f(++a, a)` lesznek. Az argumentumok kiértékelés sorrendje nincs meghatározva, így hiba lehet.Olyan program kódot ne írjunk ami a kiértékelési sorrendet előre megszerenté határozni mert hibát visz a programba.

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

Semmi extra nem történik , csak ki íratunk két egészet. Az egyiket az "f" függvényünk fogja vissza adni, a másikat pedig az "a" változó értéke.

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

Ebben az esetben , mivel az f függvény közvetlenül tudja variálni az "a" változót , ezért a kiértékelődési sorrend megint felborulhat.

Megoldás forrása:https://github.com/lovaszbotond/Chomsky/blob/master/Jelkezelo

Megoldás videó:

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

$(\forall x \exists y ((x<y)\wedge(y \text{ prím})\wedge(SSy \text{ prím})) ↩
   )$

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

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

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

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

A feladatunk az volt , hogy kiolvassuk helyesen a felírt kifejezéseinket.

1.- Minden x esetén létezik olyan y , ami nagyobb mint x és y prím. Ebből következik , hogy végtelen sok prímszámunk van.

2.- Minden x esetén létezik olyan y , ami nagyobb mint x és y prím valamint ikerprím (SS0->2("ssy -> 2 prím ikerprím) . Tehát végtelen sok ikerprím számunk van.

3.- Létezik olyan y , ami minden x esetén nagyobb ha x prímszám .A supset az az "implikáció" a nyelvünkben. Tehát véges sok prímszámunk van.

4.- Létezik olyan y , amely minden x esetén igaz , hogy ha y kisebb mint x akkor x nem prím. a \neg a ("negáció"-tagadás)-ahol tagadom , hogy x prím.Következmény: Véges sok prímszám van.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató

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

Mit vezetnek be a programba a következő nevek?

- ```
  int a; // az "a" egy egész
  ```
- ```
  int *b = &a; // egészre mutató mutató
  ```
- ```
  int &r = a; // egész referenciája
  ```
- ```
  int c[5]; // 5 elemű tömb
  ```
- ```
  int (&tr)[5] = c; // rekurzivan hivatkozik a "c" 5 elemu tombre
  ```
- ```
  int *d[5]; // egészre mutató 5 elemű tömbre mutatók tömbje
  ```
- ```
  int *h (); // a függvény egészre mutató mutatót ad vissza
  ```
- ```
  int *(*l) (); //egészre mutató mutatót visszaadó                ↩
     függvényre mutató mutató
  ```
- ```
  int (*v (int c)) (int a, int b) // egészet visszaadó és két egészet kapó  ↩
     függvényre mutató mutatót visszaadó, egészet kapó függvény
  ```
- ```
  int (*(*z) (int)) (int, int); // függvénymutató egy egészet visszaadó és  ↩
     két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó  ↩
     függvényre
  ```

Megoldás videó:

Megoldás forrása: https://github.com/lovaszbotond/Chomsky/tree/master/Deklar%C3%A1ci%C3%B3

A továbbiakban szépen végigmegyünk a kódcsipeteken.

```
#include <stdio.h>

int main()
{
int a=4;
printf("%d az a szám \n",a);
return 0; // ez az első (int a;)
}
```

Létrehoztam az 'a' egész típusú változót , és ki írattam.

```
#include <stdio.h>

int main()
{
int a=4;
int *b = &a;
*b=10;
printf("%d az 'a' szám \n",a);
return 0; // ez a második ( int *b = &a )
}
```

Az egészre mutató mutatóm létrejött . A "b" felülírja az "a" értékét.

```
#include <stdio.h>

int main()
{
int a=4;
int *b = &a;
*b=10;
int &r=a;
r=15;
printf("%d az 'a' szám \n",a);
return 0; // ez a harmadik ( int &r = a)
}
```

A gcc fordító jelenleg nem segít mert a C nyelvben nincs referencia így csak a g++ tudja a programot helyesen lefordítani. Az

```
 &r
```

jelenleg egy hivatkozás lesz, az egész referenciája.

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

```
int main()
{
int a[5]={1,10,100,1000,10000};

printf("%d %d %d %d %d , az a tömb elemei \n",a[0],a[1],a[2],a[3],a[4]);;
return 0; // ez a negyedik ( int a[5])
}
```

Létrehoztam az 5 elemű tömbömet , majd ki írattam.

```
#include <stdio.h>

int main()
{
int a[5]={1,10,100,1000,10000};
int (&tr)[5] = a;
  for (int i=0;i<5;i++)
    {
    printf("%d \n",tr[i]);
    }
return 0; // ez a az ötödik ( int (&tr)[5]=a)
}
```

Szintén a g++-t kell használnunk a korábban leírt problémák miatt.Ebben a részben az egészek tömbjének a referenciáját csináltuk meg.

```
#include <stdio.h>

int main()
{
int a[5]={1,10,100,1000,10000};
int (&tr)[5] = a;
int *d[5];
  for (int i=0;i<5;i++)
    {
    printf("%d \n",tr[i]);
    }
  for (int i=0;i<5;i++)
    {
    printf("%p \n",d[i]);
    }
return 0; // ez a hatodik (int *d[5])
}
```

Elkészítettem az egészre mutató mutatók tömbjét.

```
#include <stdio.h>

int *h()
{ int a=8; int *b = &a;
return b; }
```

```
int main()

{printf("%p \n",h());
return 0;} // ez a hetedik (int *h())
```

Ha szépen sorban kiolvassuk mit is csinál a program láthatjuk , hogy a függvény egészre mutató mutatót ad vissza.

```
#include <stdio.h>

int *h()
{ int a=8; int *b = &a;
return b;

}
int main(){
int* (*c)() = h;

printf("%p \n",c());
return 0; // ez a nyolcadik ( int *(*l)())
}
```

Szintén csak sorban kell olvasni mit is csinálunk , kivéve ami újdonság lehet , hogy behoztunk egy `int*`-ot ami egy függvényre mutató pointer. Tehát egészre mutató mutatót visszaadó függvényre mutató mutató a programunk.

```
#include <stdio.h>

int
sum (int a, int b)
{
    return a+b;
}
int
mul (int a, int b)
{
    return a*b;
}
int (*sumul (int c))(int a , int b)
{
    if (c)
        return mul;
    else
        return sum;
}
int
main ()
{
printf("%d \n",sumul(12)(0,5));
```

```
    return 0; // ez a kilencedik (int (*v (int c)) (int a, int b))
}
```

Sum / Multiply --> összeadás,összegzés/szorzás. A feladatban létrehoztuk az egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre programunkat. A `sumul(12)` ha `sumul(0)` akkor fog az else ágra lépni az "if"-en belül.

```
#include <stdio.h>

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

}
int
main ()
{

int (*(*z) (int)) (int, int)=sumul;

printf("%d \n",z(0)(0,5));

    return 0; // ez a tizedik (int (*(*z) (int)) (int, int))
}
```

Sikerrel vettül az akadályokat és megoldottuk a függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre programunkat. Látható hogy a ki íratásnál írtuk át illetve behoztuk a feladatban adott sort és egyenlővé tettük a sumul függvényünkkel.Az előző feladatnak függvényére mutatót leprogramoztuk.

4. fejezet

Helló, Caesar!

4.1. int *** háromszögmátrix

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

int
main ()
{
    int rs = 5;
    double **tb;

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

    if ((tb = (double **) malloc (rs * sizeof (double *))) == NULL)
    {
        return -1;
    }

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

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

    }

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

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

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

    tb[3][0] = 42.0;
    (*(tb + 3))[1] = 43.0;
    *(tb[3] + 2) = 44.0;
    *(*(tb + 3) + 3) = 45.0;

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

    for (int i = 0; i < rs; ++i)
        free (tb[i]);

    free (tb);

    return 0;
}
```

Megoldás videó:

Megoldás forrása: https://github.com/lovaszbotond/Caesar/blob/master/int**

A dinamikus memóriakezeléssel fogunk foglalkozni a feladat megoldása során.A feladatot a szokásos módon, a headerben a megfelelő függvénykönyvtárak behívásával kezdtük el.Létrehoztuk a `rs = 5` -el a sorainknak a számát amire szükségünk van.A `double **tb` deklarálunk és lefoglalunk a memóriában 8 byte-ot jelen esetben.Majd szimplán egy kiíratást végzünk . A `%p` segítségével egy hexadecimális számot fogunk ki írni mely a pointer számát fogja adni. Az adott sorban a vessző után látjuk , hogy a cimkéző operátor segítségével 1 bájt címét fogjuk ki íratni.Konkrétan lekérdezzük a címét. A `malloc` vissza ad egy pointert a lefoglalt területre. Memóriát foglal a szabad tárban. Kap egy paramétert ami az rs függvényében zajlik.(5)Következőkben látjuk , hogy a `size of (double*)` megmondja hogy mennyi hely(byte) kell a double* típusnak, ami a feladatban 5*8 azaz 40 byte helyet fog lefoglalni.Egyébként ami vissza kapunk ponter bármire mutathat a malloc esetében.Típus kényszerítjük ami miatt a "size of" méretét fogja vissza adni. A `tb`egyenlő a malloc által vissza adott értékkel memória foglalás szempontjából.Ha bármilyen hiba merülne fel akkor pedig a program kilép. Ezt biztosítjuk a `==NULL` szekcióban.A `NULL` nem mutat sehova.A folytatásban megkreáljuk az 5 sorból álló háromszögünket.Nagyon jól lehet szemléltetni a hivatkozásainkat , hogyan és miként működnek. Lehet látni , hogy a `tb[3][0] = 42.0`a harmadik sor első elemét egyenlővé tesszük 42.00-val. Azért az első eleme, erről még nem beszáltünk, mert 0-tól kezdjük a számolást/indexelést.Az alatta lévő sorban a második elemre hivatkozunk, alatta a harmadik elemre, majd a negyedik elemre.A "*" al hivatkozok azon a címen tárolt pointerre.Tökéletesen reprezentálja a példa mily

módon és hányféleképpen tudunk hivatkozni.A `for`ciklusukhoz különösképpen nem tudunk kiegészítést írni . A Chomsky fejezetünkben a hivatkozásos példánál kiveséztük nagy részét és a lényeg megtalálható benne ami ezen feladat megértéséhez kell.Amit esetlek meglehet említeni a `tb[i][j]`ahol az "i" a sor a "j" pedig az oszlopot jelzik.Fontos hogy vizsgáljuk a kapcsos zárójelek elhelyezkedését,hisz úgy fogjuk meg érteni mikor mi hajtódik végbe.A `%f` egy float típusú változót fog vissza adni.A végén a `free`-ről kell pár sort írnunk.Lényegében felszabadítja a memóriában tárolt/lévő pointerek által lefoglalt címeket. Ez fontos, hogy ne maradjon meg az adat amit tárol , mert ha a program még dolgozna tovább és ezt nem tennénk meg akkor egy idő után betelne a ram.

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

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

#define MAX_KULCS 100
#define BUFFER_MERET 256


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

  char kulcs[MAX_KULCS];
  char buffer[BUFFER_MERET];

  int kulcs_index = 0;
  int olvasott_bajtok = 0;

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

  while ((olvasott_bajtok = read (0, (void *) buffer, BUFFER_MERET)))
    {

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

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

  }

      write (1, buffer, olvasott_bajtok);

    }
```

```
}
```

Megoldás videó:

Megoldás forrása:https://github.com/lovaszbotond/Caesar/blob/master/Exor%20titkos%C3%ADt%C3%B3%20e.c

Maga a program ahogy a feladat is kéri egy titkosító lesz melyhez egy x hosszu kulcsot megadba titkosíthatunk szövegeket.Megismerkedünk a headerben egy nem "#include"-al kezdődő pre-compiler által fordított függvénykönyvtárral,ami a nevéből adódóan is látjuk , hogy definíció.Nem is kell túlgondolnunk definiáljuk , hogy a `MAX_KULCS` = 100 és ezt nem fogjuk tudnu a mainen belül megváltoztatni.Szintúgy a `BUFFER_MERET`-et 256-al tesszük egyenlővé.Megadja mennyit tud egyszerre beolvasni(hány darab karaktert).Az `argc` megmondja hogy a programot hány szóval hívtam meg/fordítottam le. Tegyük fel |gcc e.c -o k.c| akkor a spacek száma plusz 1 lesz az argc, ami jelenleg a példánkban 4.Az `argv` tárolja az argumentumokat.Tehát a parancssori argumentumok kezeléséhez szükségesek.A `char` variable egy nagyon pici 1 byte-os történet.Karaktereket deklarálunk vele.A programban a "max_kulcs", illetve a buffer amit beállítottunk neki.A kulcs méret amivel folytathatjuk. A `strlen` a string.h-ban találjuk meg , és a megadott string, hogy hány karakter hosszú azt mondja meg. A `strncpy` "string copy" pedig hogy mibe, mit , milyen hosszút. A `while` ciklusban végigolvassa a szöveget a "read" segítségével ami egészen addig olvassa a fájlt, míg a fájl vége jellel nem találkozik.Majd ha megtalálta lelövi a programot. A "buffer" méretnyit olvassa be. A write pedig ki írja az olvasott byteokat.Tehát "titkosítunk".A `for` cikluson belül a ciklus 0-tól indul , az olvasott byte-ok számáig és mindig növeljük egyel.A cikluson belül a magban a buffer "i"-edik elemét exorozzuk az adott kulcsindexel.A kulcs indexhez hozzá adunk egyet majd maradékosz osztást végzunk rajta.Ez azért kell nekünk, mert ha eléri a kulcs adott méretét akkor ismét nulláról fog indulni.A "return 0;" nem kell a program végére, mert elvégzi a feladatot és off.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

```
import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;
  class exor
    {
        public static void main(String[] args)
      {
        FileInputStream fin=null;
        FileOutputStream fout=null;
        try
      {
          fin=new FileInputStream("tiszta.txt");
          int c;
        ArrayList<Integer> in_v = new ArrayList<Integer>();
        ArrayList<Integer> out_v = new ArrayList<Integer>();
          Scanner in = new Scanner(System.in);
          System.out.println("Enter key:");
          String key = in.nextLine();
```

```
            if(key.length() !=9)
        {
            System.out.println("Bad key");
            return;
        }
            while((c=fin.read())!=-1)
        {
                in_v.add(c);
        }
            for(int i=0;i<in_v.size();i++)
        {
                 out_v.add(in_v.get(i)^key.charAt(i%9));
        }
            fout = new FileOutputStream("titkos.txt");
                for(int i=0;i<out_v.size();i++)
        {
          System.out.printf("%c",out_v.get(i));
        }

      }
      catch(Exception ex)
      {

      }
      finally
      {
        try
        {
          if(fin!=null) fin.close();
          if(fout!=null) fout.close();
        }catch(Exception ex)
        {

        }
      }
    }
}
```

Megoldás videó:

Megoldás forrása:https://github.com/lovaszbotond/Caesar/blob/master/java%20exor

A feladat megoldásánál Petrus József Tamás segített. A program az előző feladat megoldása Java környezetben. A különbség a két program között , hogy az alap szöveget tartalmazó fájl neve tiszta.txt, amely egy mappában kell , hogy legyen a programunkkal illetve a kulcsot a standard inputrol kéri be , nem pedig parancssori agumentumként mint ahogyan az korábban volt.

4.4. C EXOR törő

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

```
#define MAX_TITKOS 4096
#define OLVASAS_BUFFER 256
#define KULCS_MERET 5
#define _GNU_SOURCE

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

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

  return (double) titkos_meret / sz;
}

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

  double szohossz = atlagos_szohossz (titkos, titkos_meret);

  return szohossz > 5.0 && szohossz < 9.0
    && strcasestr (titkos, "hogy") && strcasestr (titkos, "nem")
    && strcasestr (titkos, "az") && strcasestr (titkos, "ha");

}

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);

}

int
main (void)
{

  char kulcs[KULCS_MERET];
  char titkos[MAX_TITKOS];
  char *p = titkos;
  int olvasott_bajtok;

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

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

  // osszes kulcs eloallitasa
  for (int ii = 'a'; ii <= 'z'; ++ii)
    for (int ji = 'a'; ji <= 'z'; ++ji)
      for (int ki = 'a'; ki <= 'z'; ++ki)
  for (int li = 'a'; li <= 'z'; ++li)
    for (int mi = 'a'; mi <= 'z'; ++mi)

      {
        kulcs[0] = ii;
        kulcs[1] = ji;
        kulcs[2] = ki;
        kulcs[3] = li;
        kulcs[4] = mi;
```

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

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

  return 0;
}
```

Megoldás videó:

Megoldás forrása:https://github.com/lovaszbotond/Caesar/blob/master/Exor%20titko
C3%ADt%C3%B3%20t.c

A korábban megírt 4.2-es feladatunkban látott program titkosított szövegét hivatott feltörni.Először megcsinálják a kötelező, feladat megoldásánál fő szempontot játszó függvényeket , melyek segítenek a feltörésben.Nézzük őket darabonként.

Az `atlagos_szohossz`-on belül láthatjuk a `const char *titkos`-t ami egy későbbiekben meg nem változtatható mutató.Beállítom hova mutasson a pointer és úgy is marad. A `for` cikluson belül az "i" 0-tól indul egészen míg a `titkos_meret`-et el nem éri.Az `if`-en belül megszamoljuk a karaktereket szavankent.Ha space jelenséget észlel tehát az "i"-edik eleme szóköz akkor növeljük az "sz"-t.Ami a "titkos" fakkba fog menni. A returnel pedig szimplan atlagot szamolunk.Elosztjuk a `titkos_meret`-et az `sz` szóközök számával.

A `tiszta_lehet` függvény ahogy a komment szekcióban is írjuk a programon belül , a potenciális töréseket csökkenthetjük.Jelen esetben be is állítjuk, hogy az átlagos szóhossz 5 és 9 közés esik , a leggyakoribb magyar szavaink pedig , hogy - az - nem - ha , melyeket az `strcasestr` a `strstr()`-el ellentétben megvizsgálja ezeknek az előfordulását és rögzítik is azt." Tű a szénakazalban " ahogyan a manuál is írja.A lényege pedig , hogy a `szohossz`-ban tároljuk az `atlagos_szohossz` által kapott értékeket.

A mainen belül , szintén olvashatjuk a kommenteket melyek segítenek a feladat megértésében. Megadjuk hogy jelen esetben a kulcsunk öt hosszú legyen , de lehetne harminc akár száz hosszú is.Egyébként a kulcs a-z-ig illetve 0-9-ig tartalmazhatja a karaktereket. Speciális karaktereket nem tartalmazhatna.A program sebességén tudunk növelni , viszont akkor át is kellene írnunk a forrásunkat. Egyenlőre ismerkedjünk ezzel a forrással, és a későbbiekben még találkozhatunk , ezen forrás továbbfejlesztett felgyorsított változatával.

4.5. Neurális OR, AND és EXOR kapu

R

```
library(neuralnet)

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)

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


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])



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=0, linear.output=FALSE,  ↩
   stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)

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



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])
```

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

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

A neuron egy olyan agysejt, amelynek alapfeladata az elektromos jelek összegyűjtése, feldolgozása és szétterjesztése.Azt gondoljuk, hogy az agy információfeldolgozó kapacitása elsősorban ilyen neuronok hálózatából alakul ki.A korai MI néhány kutatása mesterséges neurális hálók létrehozására irányult.Elnagyolva az mondható, hogy a neuron akkor "tüzel", amikor a bemeneti értékek súlyozott összege meghalad egy küszöböt.A neurális háló a tanuló rendszerek egyik leghatékonyabb és legnépszerűbb formája, ezért megéri külön tárgyalni.A neurális hálók irányított kapcsolatokkal összekötött csomópontokból vagy egységekből állnak. Az "i"-edik egységtől az "i"-edik felé vezető kapcsolat hivatott az aktivizációt terjeszteni. Minden egyes kapcsolat rendelkezik egy hozzá asszociált numerikus súllyal , ami meghatározza a kapcsolat egységét és előjelét.Minden egyes "i" egység először a bemeneteinek egy súlyozott összegét számítja ki. A neurális hálózat bemeneti kapcsolatokból , függvényekből , aktivációs függvényekből , egy kimenetből és kimeneti kapcsolatokból áll.A kimenetét úgy kapja, hogy egy g aktivációs függvényt alkalmaz a kapott összegre.Később le is írjuk , mi is ez a függvény.

A feladat a neurális OR , AND , EXOR kapukat kéri tőlönk.Megfelelő bemeneti és eltolássúlyokkal rendelkező, küszöbaktivációjú egységek képesek logikai kapuként működni.Ez azért fontos, mert azt jelenti, hogy ezen egységek felhasználásával tetszőleges logikai függvény kiszámítására tudunk hálózatot építeni.

A `library(neuralnet)`-es csomagot betöltjük a programunkba.Három részre bontva elmagyarazzuk/-megtanítjuk , hogyan is alkalmazza a különböző bementeimet és hogy dolgozzon vele. A logikai kapuk különböző logikai igaz/hamis kiemeneteket adnak mely az OR esetében :

```
0-0->0
0-1->1
1-0->1
1-1->1
```

Az AND :

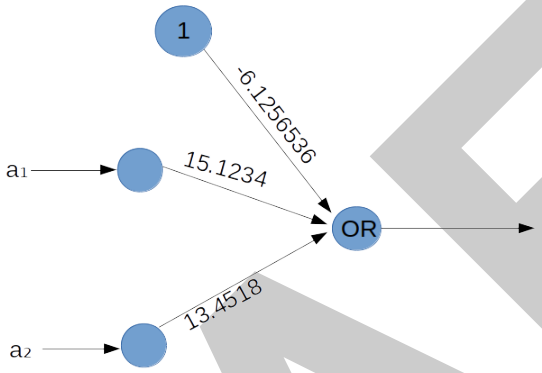
```
0-0->0
0-1->0
1-0->0
1-1->1
```

Az EXOR pedig :

```
0-0->0
0-1->1
1-0->1
1-1->0
```

Tehát ezt tanítjuk meg neki , hogy ezekkel a paraméterekkel dolgozzon.Az `or.data` sorunkban csinálunk belőle adatot.Átadjuk ezeket az adatokat a neurálnetnek.Szintén az `nn.orand` illetve az `nn.exor`sorokban is ezt tesszük meg. Az OR és az AND esetében kiválóan működik a programunk.Miután megattuk neki a

paramétereket a neurális háló elkezdi saját magát tanítani.Beállítja a megfelelő súlyokat amivel dolgozni fog és amikor azok kijönnek , akkor a tanítás befejeződik. A `compute` paranccsal beadjuk neki hogy ellenőrizze le magát és látjuk , hogy az eredményben a táblázatokhoz megfelelően egy nagyon közeli értéket kapunk, amiből már tudjuk , hogy helyes a megoldás és a művelet eredménye.Az EXOR esetében a súlyozás nem megfelelő nagyjából mindenött 0.5-ös eredményt kapunk pedig ahogy szemléltettük '0,1,1,0' a végkifejlett.Így hát megoldásnak azt választjuk , hogy a `hidden` azaz rejtett neuronokat nem 0-val hanem például 2-vel vagy programunk esetében egy kis sorozattal tesszül egyenlővé `hidden=c(6,4,6)`.Így azért is fog működni , mert rájöttünk , hogy a többrétegű hálózatokat lehet jól tanítani.A következő ábra amit szemléltetni fogok a program egy lehetséges megvalósulása lesz és remélem segít megérteni mégjobban , hogyan is működik a folyamat.A plot fügvénnyel a program végén egy hasonló ábrát rajzoltatunk ki, az adatok függvényében.



a1 – bemenet 0 1 0 1
a2 – bemenet 0 0 1 1
or – kimenet 0 1 1 1

A bemenetről láthatjuk a súlyozását a programunknak az éleken , illetve az 1-es bemeneten a negatív értékkel.

1- 0.001091766 – 0
2-0.999991787 – 1
3-0.999124123 – 1
4-0.999999999 – 1

4.6. Hiba-visszaterjesztéses perceptron

C++

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

```
using namespace std;

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

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

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

  double* image = new double[size];

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

  double value = (*p) ( image );

  cout << value << endl;

  delete p;
  delete [] image;

return 0;
}
```

Megoldás videó:

Megoldás forrása: https://github.com/lovaszbotond/KONYVALL/blob/master/perceptron
hpp(perceptron.hpp forrása)

Megoldás forrása 2:https://github.com/lovaszbotond/KONYVALL/blob/master/perceptro
cpp(perceptron.cpp forrása)

A neuron a mesterséges inteligenciában használt legelterjettebb változata,a perceptron, amit egy alakfelismerő fépnek tekinthetünk.A nullából és egyesekből álló bemeneti mintázatokat hivatott megtanulni számos kísérlet árán.Ezeknek a bemeneteknek a súlyozott összegzését végzi, amelyet egy nem lineáris leképezés követ.

A perceptron három fő elemből áll:

A retina: Bemeneti jeleket fogadó cellákat tartalmazza, melyek a legtöbb modellnél igen/nem válaszokat szolgáltattak, de lehetőség volt a stimulus intenzitásával arányos jelek generálására is.

Az asszociatív cella: Egyes cellái csatlakozhatnak a retinacellákhoz, más asszociatív cellákhoz, és az alábbiakban ismertetett harmadik réteg döntési celláihoz. Ezek a cellák összegzik a hozzájuk érkező jeleket, impulzusokat.

A döntés cella: Ez a réteg a perceptronnak a kimenete.Az asszociatív réteggel azonosan működnek, csak a bemenetek származhatnak akár egy, vagy több döntési cellától, akár az asszociatív celláktól.

A feladatunk megoldásánál a bull.png segített, hogy megkapjuk az eredmenyünket az stdoutra. Nézzük részenként hogyan is működik a perceptron.cpp

A headerben includeáljuk a "perceptron.hpp" csomagunkat melyben a perceptron osztályt megtalálhatjuk , lényegében a perceptronnak a definíciója. Írunk egy maint. A mainben az első soromban beállítom , hogy nem én szeretnék képet létrehozni , hanem fájlból szeretném importálni.Ugye paraméterenként fogom átadni a nevét ami jelen esetünkben az első argumentumunk lesz majd --> `png::image pngrgb_pixel png_image ( argv[1] );`.A kép méretét megadom az `int size`-al ahol a kép szélességét szorzom össze a kép magasságával. Majd foglalunk a szabad táron helyet a perceptronnak. Elnevezzük p-nek. A paraméterei : 3 layer, az első rétegnek size darab neuront akarok adni tehát azt állítom be, a másodiknak 256 , a harmadiknak pedig egy szám lesz az eredménye tehát azt állítom be.Ha már foglaltunk a perceptronnak helyet a memóriában akkor a végén töröljük is azt ki , nehogy elfollyon a végén. Ezt a `delete p`-vel tesszük meg. Egy olyan osztályunk van a továbbiakban ami tartalmazza a függvényhívás operátort.Ennek átadjuk az image-t `double* image = new double[size]`. Így helyet foglaltunk az imagenek is. A `for` ciklust egyértelműen lehet olvasni korábbi tanulmányainkból kifolyólag , ahogy haladtunk a könyvvel.Első for ciklus a szélesség , második magasság.Majd ugye az image hez a sor oszlopot hozzá társítjuk.Az "i" sor "j" oszlop ahogy a "height"-"width" is mutatja.Ez akkor egyenlő az RGB objektumnak a jelen esetben red komponensével. De ez lehetne green .. blue .. vagy színek szorzata... stb. Majd kiszámoljuk az értéket `double value` , amit stdoutra ki is íratunk `cout value`.Magát a programot az alábbi paranccsal tudjuk fordítani : `g++ perceptron.hpp perceptron.cpp -o perc(bármi) -lpng`

Futtatni pedig : `./perc kép_neve.png`

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása:https://github.com/lovaszbotond/Mandelbrot/blob/master/Mandelsi

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

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

void GeneratePNG(int tomb[N][M]) {
  png::image<png::rgb_pixel> image(N, M);

  for (int x = 0; x < N; ++x) {
    for (int y = 0; y < M; ++y) {
      image[x][y] = png::rgb_pixel(tomb[x][y], tomb[x][y], tomb[x][y]);
    }
  }
  image.write("kimenet.png");
}

struct Komplex {
  double re, im;
};


int main() {

  int tomb[N][M];
```

```
  int i, j, k;
  double dx = (MAXX - MINX) / N;
  double dy = (MAXY - MINY) / M;

  struct Komplex C, Z, Zuj;

  int iteracio;

  for (i = 0; i < M; ++i) {
    for (j = 0; j < N; ++j) {
      C.re = MINX + j * dx;
      C.im = MAXY - i * dy;

      Z.re = 0;
      Z.im = 0;
      iteracio = 0;

      while (Z.re * Z.re + Z.im * Z.im < 4 && iteracio++ < 255) {
        Zuj.re = Z.re * Z.re - Z.im * Z.im + C.re;
        Zuj.im = 2 * Z.re * Z.im + C.im;
        Z.re = Zuj.re;
        Z.im = Zuj.im;

      }
      tomb[i][j] = 256 - iteracio;

    }
  }

  GeneratePNG(tomb);

  return 0;
}
```

A feladat megoldásához , köszönöm a sok segítséget az UDPROG-os kódoknak , melyeket sourceforge.net weboldalán meglehet találni és későbbi csokorbeli példák megoldásánál is lehet használni az általuk megszerzett tapasztalatot.

Ezen mandelbrot halmaz részletesebb leírását meg lehet találni a csokor második feladatánál ahol már be include-áljuk a complex függvénykönyvtárat.A feladatunk , majdnem ki lehet jelenteni , hogy teljes mértékben hasonlít a másodikhoz. A különbség kettejük , között , hogy ebben a szakaszban a nehézséget , az adja , hogy saját magunknak kell le implementálnunk a különböző funkciókat/függvényeket amiket , alapvetően megkapunk a complex csomagban. A `#define` stabil definiálással az exor töréseknél találkozhattunk először. Ezek lefixált elemek amelyek értékein később a programban nem tudunk változtatni.A `GeneratePNG` függvény mint ahogyan a nevéből is lehet következtetni , fogja nekünk elkészíteni a mandelbrot képet.A stílusosabb megoldást complex osztályos mandelbrot halmaz nyújtja , mert abban mi adhatjuk , meg mi legyen a képünk-nek a neve, míg itt a kimenet.png nevet fogja kapni. (`image.write("kimenet.png");`)

x->szélesség y->magasság

N->szélesség M->magasság

a MIN/MAX magától értetődően - minimum / maximum

re - valós rész , im - képzetes rész

for ciklus i - kisebb mint #define M , for ciklus j - kisebb mint #define N

A programban megtalálható számításokról részletesen a második példán keresztül fogunk megismerkedni.

A két programkódot összehasonlítva lehet látni , ,hogy a `while` ciklusuon belül a komplex résznél nem kell megírnunk a számításokat.

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

Megoldás videó:

Megoldás forrása:https://github.com/lovaszbotond/Mandelbrot/blob/master/Mandelco

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

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

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

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

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

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

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

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

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

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

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

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

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

              ++iteracio;
            }

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

      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 Mandelbrot-halmazt Benoit Mandelbrot fedezte fel, és Adrien Douady és John Hamal Hubbard nevezte el róla 1982-ben.A matematikában a Mandelbrot-halmaz azon c komplex számokból áll , melyekre az alábbi

x'n rekurziv sorozat:

x1:=c , x'n+1 := (x'n)^2 + c

nem tart végtelenbe, azaz abszolút értékben (a hosszára nézve) korlátos. A Mandelbrot-halmazt a komplex számsíkon ábrázolva, egy nevezetes fraktálalakzat adódik.

Most , hogy megvolt a tiszteletkörünk nézzük meg , hogyan is működik a programunk amit fentebb láthatunk.

Az előző feladat nehézségeit kiküszöbölve ebben a megoldásban segítségül hívjuk a headerben az `#include complex`könyvtárat amelyben már megtalálhatóak a korábaan leírt függvények. Láthatjuk , hogy az elején deklaráljuk a megfeleleő egységeinket `int/double` típusú változóinkat felhasználva.Az előző csokrokban is találkoztunk már a main melletti argumentumokkal. Ugye az `argc` utal arra , hogy hány szóval hívom meg a programunkat , illetve az `argv` amely a parancssori argumentumok kezeléséhez szükséges.Az `if` utasításblokkon belül léthatjuk , hogy az argc amennyiben 9 szóval hívjuk meg a programot az esetben a szélesség lesz a harmadik a magasság(ez a kettő lesz a komplex sík vizsgált tartományára feszített háló) lesz a negyedik az iterációs határ(tehát hogy maximum hány lépésig tudom nagyítani(nagyítási pontosság)) lesz az ötödik.Az a,b,c,d maga a komplex síkunknak a vizsgált tartománya amiket megadhatunk.Az `atoi` egy string típusu változót átkonvertál int típusúvá. Az `atof` pedig hasonlóan csak double típusúvá konvertál.Az `else` ágon arról lenne szó , ha véletlen rosszul futtatnánk vagy rosszul írnánk be, rossz sorrendben a paramétereket , akkor segít nekünk és az stoutra ki írja a helyes használatnak a feltételét.Ez utobbi esetben a program jelzi is az operációs rendszer felé hogy véget értem. (`return -1`) A `png::image` a png könyvtár alapkészletében találjuk , mellyel az adott sorban megadjuk a képünknek a szélességét,magasságát. A `dx/dy` esetében megadjuk , hogy az [a,b]x[c,d] tartományon milyen sűrű a megadott szélesség/magasság háló.Alatta találjuk a valós komplex illetve imaginárius komplex részt, valós egészeket , imaginárius egészeket. Mint ahogy a komment is írja a j-vel végigmegyünk a sorokon a k-val pedig az oszlopokon egy `for` ciklus keretein belül.A `while`-on belül ha a z_n kisebb mint 4 akkor a feltétel nem teljesült.A program az iteráció kisebb iterációshatár sérülésével lépett ki.Tehát feltesszük, hogy a c-ről ,hogy itt a`z_{n+1} = z_n * z_n + c` sorozat konvergens , azaz iteráció= iterációshatár.Az `iteracio%255` tehát ez miatt egyenéő 255-el.Ekkor az iteráció az esetlges nagyítások során valahányszor *256 + 255.Ezt követően beállítjuk , hogy a felhasználó lássa a program , hogyan is halad az`int szazalek = ( double ) j / ( double ) magassag * 100.0; std::cout "\r" szazalek "%" std::flush;` csipetnél, így az stdouton , jelezve "1%""2%"stb.."98%""99%""100%".ha elértük a "100%"-ot akkor a program ki írja , hogy az `argv[2]`-es helyen megnevezett programom ," mentve ".

5.3. Biomorfok

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

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

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

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

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

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

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

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

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

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

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

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

        for ( int x = 0; x < 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;
                //z_n = std::pow(z_n, 2) + std::sin(z_n) + cc;
                if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
                {
                    iteracio = i;
                    break;
                }
            }

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

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

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

Eredetileg 1986-ba kell visszautazzunk , Pickover-hez aki először foglalkozott ezzel bár ma már bebizonyítottuk hogy az akkori kijelentések ma már nem helytállóak.A biomorfológia a biológiai morfológiákat jelenti. A biomorf algoritmus a gerinctelen szervezetekre hasonlító változatos és bonyolult formák létrehozásában használható. Számos technikát és módosítást vezettünk be, hogy ezeket a fraktálokat kapjuk.Részletes leírásukat , és algoritmikus gondolkodásukat a hasonló feladat típusokhoz az alábbi weboldal nyújt kimerítő és hasznos információkkal- többek között a Biomorfok különböző típusú iterációival.

https://pdfs.semanticscholar.org/f54b/0b315d979142d7d33b8e69cc8942bad1f60d.pdf

A csokor második feladványához hasonlóan indul programunk.Deklaráljuk a megfelelő egységeinket , melyekkel felokosítjuk a programunkat , hogy tudjon mivel dolgozni. Kiegészülve három számmal: "reC" "imC" és "R". A "reC" a C komplex számunk valós része az "imC" pedig a C komplex számunknak az imaginárius része.Az R pedig a valós szám.Ha azt át lépjük akkor a végtelenbe fogunk elszállni.Ezekkel természetesen a parancssori argumentumunkat is ki kell egészítenünk.Ami még újdonság a kódunkban az az `std::pow` ami a complex könyvtárban található. Elvégez helyettünk műveleteket , jelen esetben z_n-t a 3.ra emeljük vele és nem kell leírnunk háromszor egymés után szorzás formájában.Az `std::real/std::imag` a valós illetve imaginárius rész. Feltételünkön belül találhatóak meg és rájuk érvényesek a korábban kitett végtelenbe kirepülős kijelentésem. A `break`-re azért van szükségünk , mert ha már minden rácspontot megvizsgáltunk , akkor ki kell léptetni az iterációból.A `kep.set_pixel ( x, y, png::rgb_pixel ( (iteracio*20)%255, (iteracio*40)%255, (iteracio*60)%255 ));` kódban színezhetjük illetve a formáján is mahinálhatjuk kedvünkre ,a lefutás után megkapott ábrát.

A misszión során a fentebb említett/linkelt cikk ami a fő vonulatát képezte programunk megvalósulásában.

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása:

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 iteréció bejárta z_n komplex számokat!

Megoldás forrása:

Megoldás videó:

Megoldás forrása:

5.6. Mandelbrot nagyító és utazó Java nyelven

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás videó:

Megoldás forrása:

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

6.2. LZW

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

Megoldás videó:

Megoldás forrása:

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása:

6.4. Tag a gyökér

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

Megoldás videó:

Megoldás forrása:

6.5. Mutató a gyökér

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

Megoldás videó:

Megoldás forrása:

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása:

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:

Tanulságok, tapasztalatok, magyarázat...

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

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...

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:

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

[?]

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

10.3. Programozás

[BMECPP]

III. rész

Második felvonás

Bátf41 Haxor Stream
A feladatokkal kapcsolatos élő adásokat sugároz a https://www.twitch.tv/nbatfai csatorna, melynek permanens archívuma a https://www.youtube.com/c/nbatfai csatornán található.

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

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

11.4. C

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

11.5. C++

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

11.6. Lisp

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

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.