Univerzális programozás

Írd meg a saját programozás tankönyvedet!

Ed. BHAX, DEBRECEN,
2019. május 10, v. 1.0.0

COLLABORATORS			
	TITLE : Univerzális programozás		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Bátfai, Norbert Ã©s Szilágyi, Csaba	2019. május 10.	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai
0.1.0	2019-02-27	Feladatok elkészítésének megkezdése.	nraccoon
0.1.1	2019-03-04	Első feladatcsokor (2.fejezet) befejezve.	nraccoon
0.1.2	2019-03-10	Harmadik fejezet elkészítve.	nraccoon
0.1.3	2019-03-16	Negyedik fejezet elkészítve	nraccoon

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.1.4	2019-03-26	Ötödik fejezet befejezve.	nraccoon
0.1.5	2019-04-02	Hatodik fejezet befejezve.	nraccoon
0.1.6	2019-04-07	Hetedik fejezet befejezve.	nraccoon
0.1.7	2019-04-18	Olvasónaplók kidolgozva.	nraccoon
0.1.8	2019-04-23	Nyolcadik fejezet elkészítve.	nraccoon
0.1.9	2019-04-27	Kilencedik fejezet elkészült.	nraccoon
0.2.0	2019-05-09	Feladatok finomításai, és véglegesítései.	nraccoon
1.0.0	2019-05-10	A könyv első teljes kiadása.	nraccoon

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

Ábrák jegyzéke

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!

Megoldás forrása:

100% 1 mag: `Ciklus1`

0% minden mag: `Ciklus2`

100% minden mag: `Ciklus3`

Ez a feladat 3 részre bontható. Az egyikben 100%-on kell dolgoztatni egy magot, a másikban 0%-on minden magot, az utolsóban pedig 100%-on minden magot.

Az első rész, egy egyszerű végtelen ciklus megírásával, már készenek is mondható. A *Megoldás forrásban* látható, hogy én a while használatával készítettem el a végtelen ciklusom. Persze, ezt másképp is meg lehet oldani például egy: "for(;;)" ciklus használatával, ami még talán jobb is lehet, bizonyos szempontokból, de a feladat megoldásához tökéletesen megfelel az én első gondolatom is ami egy egyszerű mindig igaz while ciklusra épül. Ugyanis ez tökéletesen 100%-on használ ki egy magot futásakor.

```
while(0<1){}
```

Mivel ez mindig igaz lesz ezért a végtelenségig fut a program, és egy magot teljesen lefog.

A második rész már bonyolultabb egy kicsit, mivel itt a while cikluson belül használnunk kell egy sleep() kifejezést.(A mellékelt módhoz hasonlóan!)

```
while(0<1){
sleep(1);}
```

Valamint a forrás elején includeolnunk kell az unistd.h header fájlt.

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

Ezen kívül viszont már nincs más dolgunk csak fordítani és futtatni, az eredmény a várt lett, 0% terhelést neheztel a magokra a program.

A feladat utolsó része tűnhet már első ránézésre is a legnehezebbnek, és ez nincs is máshogy.

Én ennek a feladatnak a megoldására az OpenMp-t használtam, mely minden magra kiterjeszti a terhelést, ezáltal elérhetjük hogy a program futásakor minden magot 100%-on pörgessen.

Az OpenMp használatához includeolnunk kell az omp.h könyvtárat!

```
#include <omp.h>
```

Valamint a végtelen ciklusunk elé kell írnunk a #pragma omp parallel kifejezést!

```
#pragma omp parallel
while(0<1){}
```

A kódunk fordításához a következő parancssort kell begépelnünk a terminálba:

gcc -fopenmp (fájlnév pl:) 3.c -o (fordítás utáni fájlnév pl:) 3

Ezután futtatjuk a programot és láthatóvá válik hogy minden magot (én esetemben 4 magot) 100%-on pörget.

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 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.

Tapasztalatok és megfigyelések: Ilyen programot nem lehet írni, ugyanis ellentmondásra jutunk. Egy program nem fogja tudni megmondani hogy a neki paraméteréül adott program végtelenségig fog-e futni vagy lesz olyan pont ahol megáll.

Legjobban talán az tudná szemléltetni ha paraméterként saját magát kapná meg a program.

Mi történne ilyenkor?

Ha megállna a program csak akkor tudná visszaadni az értéket, hogy a végtelenségig fut, de akkor ez nem lenne a valóságnak megfelelő adat, hiszen nem lenne igaz, mivel megállt. A másik eset az lenne, ha nem állna meg a program futása, de ez esetben pedig nem tudná megadni azt hogy Ő a végtelenségig fut, hiszen nem állt még meg. Ezáltal egy fajta paradoxonra jutunk, és el kell ismernünk, hogy nem létezik erre a problémára megoldás.

Eleinte talán nehezen érthető még magyarázzattal is, de többszöri átolvasás után véleményem szerint mindenkinek ez a példa triviális lesz. Bátfai Tanár úr kódját nézzük a magyarázat mellé a könnyebb megértés végett!

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:

Változók cseréje kivonás és összeadás műveletek segítségével: `Csere`

Két változó értékének a felcserélésére több lehetséges opciónk is van, az egyik legegyszerűbb megoldás erre egy új változó (ún. segéd változó) létrehozása lenne, Aminek értékül adnánk az egyik felcserélendő változót, aztán a két változó értékét egyenlővé tennénk, majd a segéd változóból átraknánk az értéket a másik felcserélendő változóba.

Viszont segéd változó bevezetése nélkül is megoldható, például kivonás és összeadás művelet segítségével.

Három lépésből megoldható a változók értékeinek cserélése a kivonás és az összeadás művelet segítségével is. Első lépésben vesszük a két változó összegét, tegyük fel a két változónk most "a" és "b". Ez alapján első lépésünk a következő legyen:

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

Ezáltal az új "a" értéke az összeg lesz. Ezután következő lépésünk legyen az alábbi:

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

Így "b" értéke már is az eredeti "a" értékét hordozza magába. Ugyanis az összegből vontuk ki a "b" értékét ami így a régi "a" értékét adja. Már csak a régi "b" értéket kell megadni az új "a"-nak amit a következő képpen tehetünk meg:

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

És kész is vagyunk az érték felcsereléssel, most már az "a" hordozza a régi "b" értéket és a "b" a régi "a"-t. Ugyanis az összegből kivontuk az új "b" értéket ami a régi "a" értékkel azonos, tehát a régi "b" értéket adja vissza.

Ez egy egyszerű ám de hasznos feladat, később sokszor alkalmazhatjuk, valamint megértése is triviális, egy általános iskolás számára is.

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.)

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

Megoldás forrása:

If segítségével való labda pattogtatás: `Labdaif`

Logikai utasítás vagy kifejezés használata nélkül való labda pattogtatás: `Labda`

If-ek használatával, egy fokkal könnyebb dolgunk van, mint ha nem használhatnánk őket. Ugyanis ennek a feladatnak az a lényege hogy kövessük a "labda" mozgását, és az amint eléri a konzolunknak a "falát" a mozgásának az iránya megforduljon/megváltozzon. If-ek segítségével ezt az ellenőrzést könnyebben vizsgálhatjuk mint a következő esetben ahol nem használhatunk logika utasítást vagy kifejezést.

```
        if ( x>=mx-1 ) {       // elérte-e a jobb oldalát?
            xnov = xnov * -1;
        }
        if ( x<=0 ) {          // elérte-e a bal oldalát?
            xnov = xnov * -1;
        }
        if ( y<=0 ) {          // elérte-e a tetejét?
            ynov = ynov * -1;
        }
        if ( y>=my-1 ) {        // elérte-e az alját?
            ynov = ynov * -1;
        }
```

A feladathoz a programnak tudnia kell a konzolunk méretét, ugyanis csak így tudja hol a konzol "fala". Úgy képzeljük el hogy a konzol mérete alapján létre hozz egy "táblázatot" ahol a határértékek megegyeznek, így ha oda ér a "labdánk" akkor megváltoztatja irányát, míg bárhol máshol a konzolon belül nem fogja megváltoztatni annak mozgását. A program folyamatosan updateli magát és ahol jár a "labda" azt kiprinteli, így tud a "labda" haladni a képernyőnkön. A program tehát nem csinál mást csak számolja a labda haladását mind x mind pedig y tengelyen való haladását és hogy az adott hely a konzol határa-e vagy sem, amint eléri a határt mozgási irányt változtat, végig követi és printeli a haladást. If-ek segítségével és if-ek nélkül is ugyan ez az alapja a programnak.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás videó:

Megoldás forrása:

Szóhossz: `Szóhossz`

A szóhosszt a bitshift operátor segítségével tudjuk elvégezni. Ami úgy működik hogy a forráskódban szereplő "a"-t lépteti balra és a jobb oldalára pedig 0 értéket ad. Mind addig folytatja ezt míg ki nem nulláza az egészet.

Hogyan tudjuk meg ebből a bitek számát?

Minden shiftelés(léptetés) után növeljük a forráskódban lévő "b" változó értékét 1-el, teljesen a ciklus végéig, tehát ha lezárul akkor a "b" értéke pontosan a bitek számával fog megegyezni, azaz megtudja azt mondani hogy hány bites a szó a számítógépünkön.

```
while(a != 0){
a<<=1;
b++;
}
```

A konzolon való megjelenítéshez pedig a printf-et fogjuk használni, a következő képpen:

```
printf("%d bites a szó.\n", b);
```

A következő képpen adja vissza tehát például: 32 bites a szó.

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:

Page-Rank: Page-Rank

A Page-Rank egy algoritmus. Nem véletlenül a "Helló, Google!" cím, ugyanis a Page-Rank a Google keresőmotorjának az egyik legfontosabb eleme. 1998-ban fejlesztették ki a google alapítói Larry Page és Sergey Brin. Az algoritmus használata előtt közel sem volt olyan pontos a keressési találat mint napjainkban, szinte ma már a keresőbe beírt szavak alapján az első három találtba benne van a keresett oldal, ritka az amikor a találatok között is keresnünk kellene a megfelelőt, régebben még nem volt ilyen tökéletes ez. Ez a pontosság napjainkban annak köszönhető hogy a Page-Rank sorba állítja az oldalakat az alapján hogy hány link mutat rájuk valamint a rájuk mutató oldalakra hány oldal mutat. De miért is jó ez?

Bonyolultnak hangzik ez és értelmetlennek viszont mint látjuk a google kereső motor példáján is, ugyancsak hasznos és működése is kiváló. Visszatérve tehát mit is jelent hogy sorba állítja az oldalakat a fent említettek alapján? Azt takrja hogy a legtöbb oldal hivatkozással rendelkező oldal lesz az első és minél kevesebb oldal hivatkozik rá annál lentebbre kerül az adott lap. Ezáltal a legtöbbet keresett oldalak/leghasználtabb/legnépszerűbb találatok lesznek legelöl, így a keresési találatok tetején 99,9% kb hogy közte van az általunk keresett oldal.

A feladatban egy 4 honlapból álló hálózatra nézzük meg a Page-Rank értéket.

Az alap érték 1/4-ed lesz minden oldalnak mivel ugye 4 oldalt nézünk most. Az oldalak közötti kapcsolatot egy mátrixban tároljuk el, a következőféleképpen:

```
{
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}
};
```

Egy fajta táblázati nézetként tudható ez be, a sorok és az oszlopok megfelelő találkozási pontjánál olvasható le a kapcsolatuk. Ezekel az adatokkal a megfelelő matematikai műveletekkel, mint például a mátrixszorzás, kapjuk meg az oldalak Page-Rank értékét. Ezek az értékek egy tömbe lesznek ami az eredmény tömbünk, ezt akár ki is lehet írtani, így láthatóvá téve az eredményeket a terminálon vagy egy output fájlba. A Page-Rank-ot lehet használjuk nap mint nap is akár keresőmotor használatával például, de magát a kódot nem kell alkalmaznunk, viszont érteni igen, tehát ezt a feladatot inkább megértésre és kód olvasásra ajánlanám, jó magam se írtam meg újra a kódót, internetről szedett kódot mellékeltem tanulmányozásra.

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

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

A Brun tétel a prímszámokkal azon belül is az iker prímekkel foglalkozik. De mik is a prímszámok illetve az ikerprímek?

A prímszámok azon természetes számok melyek csak eggyel és önmagukkal oszthatóak. Ilyen például a 2, az 5, a 7 stb. A 2 egyben az első prímszám is valamint az egyetlen páros prímszám is, értelemszerűen a kettő kivételével a többi párosszám nem lehet prímszám mivel kivétel nélkül oszthatóak kettővel. A prímszámok szorzataként minden természetes szám előállítható, ezért nevezik őket a természetes számok építőelemeinek is őket. Továbbá a prímszámok száma végtelen, ezt már Krisztus előtt is tudták.

Az ikerprímek azon prímszámpárok amelyeknek a különbsége kettő. Például 5 és 7 ikerprímek hiszen a különbségük kettővel egyenlő. A Brun tétel erre épül, ugyanis azt nem tudni hogy az ikerprímek száma is végtelen-e vagy van úgy mond egy "utolsó" ikerprím. (Ezt mái napig kutatják még.)

A Brun tétel azt mondja ki, hogy az ikerprímek reciprokaiból képzett sor összege véges vagy legalábbis végtelen sor konvergens, azaz az így képzett törtszámok egy határt képeznek amit nem lépnek át soha, ezt a határ értéket Brun konstansnak nevezik.

A mellékelt forrásban lévő kód ezt a "határértéket", ún. Brun konstanst próbálja elérni.

```
primes = primes(x)
```

Az x-nek megadott számig kiszámolja a prímeket.

```
diff = primes[2:length(primes)]-primes[1:length(primes)-1]
```

Megnézi a prímek közti különbséget, ez azért fontos mivel így kitudjuk szűrni az ikerprímeket ugyanis azok közti különbség mindig kettő.

```
idx = which(diff==2)
```

Kitudjuk szedni az első tagot, és az alapján +2-vel a hozzá tartozó párt, az így kapott számok ikerprímek lesznek tehát nekik kell a reciprokuk.

```
t1primes = primes[idx]
    t2primes = primes[idx]+2
    rt1plust2 = 1/t1primes+1/t2primes
```

Ezeket a törteket összeadva kell kapnunk azt az értéket mely körülbelül az 1,8-hoz közelít minél több számot vizsgálunk át. Ezt a sum-mal tehetjük meg.

```
return(sum(rt1plust2))
```

Persze szemléletesebb a dolog ha ki is rajzoltatjuk őket.

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

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 Monty Hall paradoxonra épülő "műsor" lényege az hogy van 3 ajtó, de csak az egyik mögött van értékes nyeremény, a másik kettő mögött értéktelen dolog van, vagy semmi, a paradoxon szempontjából nem lényeges, csak annyi hogy 1 jó és 2 rossz választás van. A játékosnak lehetősége van az egyik ajtót megjelölni, hogy azt szeretné majd kinyitni, de ezután a játékvezető kinyit egy rossz ajtót, és megengedi hogy változtasson a választásán a játékos, de ezt csak egyszer teheti meg. Nyilvánvalóan a játékvezető tudja hogy hol a nyeremény és csak azt az ajtót nyitja ki ami mögött nincs az értékes nyeremény, valamint a játékos által megjelölt (elsőnek kiválasztott) ajtót sem nyithatja a játékvezető.

A Monty Hall probléma/paradoxon tehát egy érdekes téma, ugyanis nem mindenkinek triviális elsőre az hogy érdemes változtatni. Jó magam se értettem hogy miért lenne érdemesebb változtatni a döntésen. Hisz 50-50% esély van hogy értékes vagy sem a mögötte lévő dolog. Hisz egy rossz ajtót "kizár" a játékvezető, aztán újra nálunk a döntés mintha alapból csak 2 ajtó lenne.

De ha jobban belegondolunk valóban igaz az hogy elsőre 66% eséllyel egyik rossz választásra bökünk rá, míg csak 33% az esély hogy egyből a jót találjuk el. Így a szabályoknak megfelelően ha kizárjuk az egyik rossz ajtót, 66% esély arra hogy a változtatás után nyerünk és csak 33% az esély arra hogy ha nem változtatunk akkor nyerünk.

A forrásban szereplő kód áttanulmányozása után is jól látható, hogy mi a "műsor" lényege. Láthatjuk azt is hogy a műsorvezetőnek adott az ajtó ha a játékos elsőre rosszat választ, mivel sem a jót sem a játékos ajtaját nem nyithatja ki.

```
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)]

}
```

Nagyon jó kis gondolkodós példa, a forráskód Bátfai Norbert Tanár úr-é és R nyelvben íródott, de R nyelvet sose látott olvasók számára is olvasható véleményem szerint.

Ehhez a feladathoz ajánlom a "21 Las Vegas ostroma" című filmet, mely az ajánlott filmek között már fentebb is megtalálható volt, ebben szerepelni fog a Monthy Hall probléma.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: Turing gép

A Turing-gép Alan Turing angol matematikus nevéhez kötődik közvetlen, Ő volt az aki megfogalmazta eme algoritmust és leírta, valamint megjelentette egyik cikkében ezt a jelenséget még 1936-ban, amikor már képes lett volna olyan mai számítógépes probléma megoldásra ami akkoriban még nem is létezett mint eszköz sem.

A Turing-gép úgynevezett absztrakt automata amely azt takarja, hogy valóságos digitális számítógépek nagyon leegyszerűsített verziójára hasonlít vagy nevezhető is annak. De bővebben a Turing gépről már magyar nyelven is olvashatunk a Wikipédia oldalán.

```
#include <stdio.h>

int main(){
  int szam;
  printf("Adj meg egy számot: ");
  scanf("%d", &szam);

  for(int i = 0; i < szam; i++) {
    printf("1");
    }
  printf("\n");
  return 0;
}
```

A fenti kód egy egyszerű kód. Hogy is működik és mit is csinál tehát?

Deklarálunk egy "szam" változót majd a standard outputra kiírjuk a felhasználónak hogy adjon meg egy számot, amit megad szám azt pedig a deklarált "szam" valtozóba mentjük el. Ezután egy for ciklus segítségével a kapott egész számmal megegyező karakter írunk ki a standard outputra. Az én kódomban ez a

karakter az egyes, de lehetne akár a "|", "/", "\" vagy egyébb szemléletes, könnyen számlálható karakter. Tehát a természetes számokat unáris alakba képes megadni, nyilván hátrány hogy csak természetes számokkal tud dolgozni, törteket és negatív számokat sem tud kezelni.

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 generatív grammatika, generatív nyelvtan azoknak a csoportosítása Noam Chomsky nevéhez köthető, aki az 1960-as években hozta létre ezeket a csoportokat, fő célja a nyelvek modellezése volt.

Az amerikai nyelvész 4 osztályba sorolta, ezt a csoportosítást Chomsky hierarchiának hívják, de hogy is néz ez ki?

- Az általános típus (0. típus).
- A környezetfüggő (1. típus), ez az a típus amivel a feladatsorán foglalkoznunk kell.
- A környezetfüggetlen típus (2. típus).
- Reguláris (3. típus).

Chomsky-féle nyelvosztályok: A nyelvtan és az általa generált nyelv definíciója szerint minden nyelvtanhoz egy egyértelműen meghatározott nyelv tartozik, de megfordítva ez már nem lesz igaz, ugyanis egy nyelvet nem csak egy nyelvtannal generálhatunk le. Erről részletesebben és a generatív nyelvtanról bővebben olvashatunk a következő oldalon.

Két környezetfüggő generatív grammatika, amely a nyelvet genrálja például:

```
S, X, Y      változók
a,b,c        konstansok

S - abc, S - aXbc, Xb - bX, Xc - Ybcc, bY - Yb, aY - aaX, aY - aa

S (S - axBc)
aXbc (Xb - bX)
abXc (Xc - Ybcc)
abYbcc (bY - Yb)
aYbbcc (aY - aa)
aabbcc

aYbbcc (aY - aaX)
aaXbbcc (Xb - bX)
aabXbcc (Xb - bX)
aabbXcc (Xc - Ybcc)
aabbYbccc (bY - Yb)
aabYbbccc (bY - Yb)
```

```
aaYbbbccc (aY - aa)
aaabbbccc
```

Vagy például:

```
A, B, C változók
a, b, c konstansok
A - aAB, A - aC, CB - bCc, cB - Bc, C - bc

A (A - aAB)
aAB ( A - aC)
aaCB (CB - bCc)
aabCc (C - bc)
aabbcc

A (A - aAB)
aAB ( A - aAB)
aaABB ( A - aAB)
aaaABBB ( A - aC)
aaaaCBBB (CB - bCc)
aaaabCcBB (cB - Bc)
aaaabCBcB (cB - Bc)
aaaabCBBc (CB - bCc)
aaaabbCcBc (cB - Bc)
aaaabbCBcc (CB - bCc)
aaaabbbCccc (C - bc)
aaaabbbbcccc
```

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:

`c89 vs c99`

Az utasítások, ha nem jelezzük hogy ne sorba fussanak le, akkor leírásuk sorrendjébe hajtódnak végre. Nem rendelkeznek értékekkel valamint több csoportjuk ismert C nyelvben, ilyenek például:

- Kifejezésutasítások: Az utasítások többsége ide tartozik. Legáltalánosabb kifejezésutasítás a függvényhívás valamint az értékadás.
- Összetett utasítások:Másik nevén a blokk, arra képes hogy egy utasításként kezeljen egyszerre több utasítást, ez azért jó mert vannak programkörnyezetek, ahol a fordítóprogram csak egyetlen utasítással tud dolgozni, egyszerre többet nem tud elfogadni.

- Iterációs utasítások:Egy ciklust határoznak meg. Ebből a típusból a legismertebb a while, do utasítások.

Lehetne sok ilyen kódot írni, de szemléltetésnek elég egy egyszerű for ciklust írnunk ugyanis c89-ben a c99-es for ciklus felépítés nem fog lefordulni.

```
for(int i=0; i<5; i++){}
```

Ugyan ezt c89-ben így írhatjuk meg:

```
int i=0;
for(i<5; i++){}
```

Ez számomra meglepő volt, ugyanis én még nem használtam c89-et, én már megszokásból is a c99 féle for ciklust írnám meg, ami persze fordításnál egyből hibát lökne.

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

Megoldás forrása:

Lexikális elemző: `Valós számok`

A forrás 3 részre van bontva, ennek mentén haladok a magyarázattal is.

Az első részben includeoljuk az stdio.h könyvtárat, valamint létrehozunk egy változót amely számolja nekünk azt, hogy hány számot olvass be a program.

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

Majd a digitnek megadjuk hogy a 0-9-ig terjedő számjegyeket tartalmazza.

```
digit [0-9]
```

A második részében a kódnak, rögzítjük a szabályokat. Azaz itt lesz megadva hogy nagyjából mit csinál a függvény. Segítségünkre lesz ebben például az informatikában sűrűn használt csillag mely tetszőleges számú tetszőleges karaktert jelent általában. (A forrásban a digit mivel előtte áll így tetszőleges számú tetszőleges számjegyet fog takarni.) Majd ugyanebben a részben ki is íratjuk a stringet a printf segítségével, valamint az atof segítségével a string alapján készített számot elkészítjük és ugyan úgy a printf használatával ki is íratjuk.

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

A 3. rész pedig maga a program, mely ugye az int main-től indul, hívja a lexikális elemzőt, majd pedig kiíratja a "realnumber" értékét, ami a végeredmény.

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

Összeségében tehát nem egy bonyolult kód, de hogy tudjuk működtetni?

Először is a .l fileunkat lexeljük a következő képpen:

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

Ezután fordítjuk következőképpen az előbb létrehozott .c fileunkat:

```
gcc realnumber.c -o realnumber -lfl
```

Majd futtatjuk, és a példában szereplő forrás a terminálból fogadd inputot.

```
./realnumber
```

3.5. l33t.l

Lexelj össze egy l33t ciphert!

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

Megoldás forrása:

Leet chiper: `L33t`

A feladat megoldásában tutoriált engem: Dankó Zsolt

Flex lexikális elemzővel tudjuk ezt a feladatot is elvégezni, az előző feladatra épül rá igazából, a fordítást és a futattást is ahhoz hasonlóan kell elvégeznünk, tehát valahogy így:

```
lex -o l33t.c l33t.l
```

```
gcc l33t.c -o l33t
```

```
./l33t
```

A kód főeleme egy chiper lesz, ezért először is ennek a struktúráját kell felépítenünk, de ne essünk kétségben ehhez a segítségünkre lesz a következő oldal amelyen találhatunk helyettesítő karaktereket, amiket majd felhasználhatunk a programunkban. A struktúránkban deklarálunk kell egy karaktert(én esetemben ez most a c lesz) és azt is meg kell adnunk, hogy az adott karaktert mire tudjuk cserélni majd. A programnak viszont megengedjük, hogy magának válassza ki a négy lehetőség közül, hogy melyik karakterre cserélje, nem pedig mi mondjuk meg neki, hogy mire cserélje az adott betűket vagy számokat.

Felmerülhet továbbá az a kérdés, hogy hogyan tudjuk kezelni a kis- és nagybetűket hogy ez ne legyen probléma?

A `tolower()` függvényt használva nem lesz ilyen gondunk. Ugyanis ez átalakítja az összes nagybetűnket kisbetűvé, így már nem ütközhetünk emiatt problémába.

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?

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

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

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

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

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

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

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

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

Megoldás forrása: Signal

Megoldás videó:

Mi ennek a kódnak a feladata/lényege?

A parancsul adott signal-t kapja el ami a jelen esetben a kettes tehát a Ctrl+C, ami ugye alap esetben "bezárná" programunk futását.

Ahhoz hogy signalokkal foglalkozzunk a kódunkban includeolnunk kell a signal.h könyvtárat

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

Megtudjuk még azt is tenni hogy elkapáskor csináljon valamit, például a forrásomban azt írja ki hogy "Elkapva 2" ahol a 2 a signalnak a száma.

```
printf("Elkapva! %d\n", sig);
```

Fontos azt is megjegyezni hogy nem a program lesz "megállíthatatlan" csak a Ctrl+C signalt kapja el, tehát például a Ctrl+Z segítségével a programunk megfog állni ugyan úgy mint eddig.

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

Minden x-re igaz az hogy van olyan y, hogy y nagyobb mint x és y prím. Tehát a prímek száma végtelen.

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

Minden x-ra igaz az hogy létezik olyan y, hogy y nagyobb mint x és y prímszám, valamint y rákövetkezőjének a rákövetkezője is prímszám. Tehát az ikerprímek száma végtelen sok.

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

Létezik olyan y hogy minden x-re igaz az, hogy ha x prím akkor kisebb mint y. Tehát minden prímszám esetén találunk olyan számot mely nagyobb mint Ő, tehát a prímszámok száma véges sok.

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

Létezik olyan y amely minden x-ra igaz az hogy ha x nagyobb mint y és x nem prímszám. Tehát van olyan szám amitől minden nem prím szám nagyobb, azaz nincs olyan szám amitől nem lenne nagyobb nem prímszám, ha y prím lenne akkor az előzővel ekvivalens lenne a megoldás.

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

A feladat megoldása számomra nem okozott gondot, mivel első félévben tanultam logikát mint tantárgyat, de azok számára akik nincsennek tisztában ezzel a tantárggyal és az ilyen típusú feladatokkal, azok számára egy érdekes és elgondolkodtató feladat lehet ez.

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

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

Megoldás videó:

Megoldás forrása: Deklaráció

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
  ```

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

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

Megoldás videó: https://youtu.be/1MRTuKwRsB0, https://youtu.be/RKbX5-EWpzA.

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/tm.c

Bátfai Norbert Tanár úr kódját fogom bemutatni, valamint elemzeni.

Először is includeolni kell a szükséges könyvtárakat ami a szokásos stdio könyvtár lesz valamint mellette az stdlib könyvtár is kell ugyanis a mallocot majd így érhetjük el.

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

Majd mainen belül az első amit teszünk az az hogy deklarálunk egy integer típusú "nr" változót amelyben majd az alsó háromszög mátrixunknak a sorainak a számát fogjuk tárolni. A forrásban ez 5 lesz, tehát 5 soros lesz az alsó háromszög mátrixunk, tehát 15 értéke lesz.

```
int nr = 5;
```

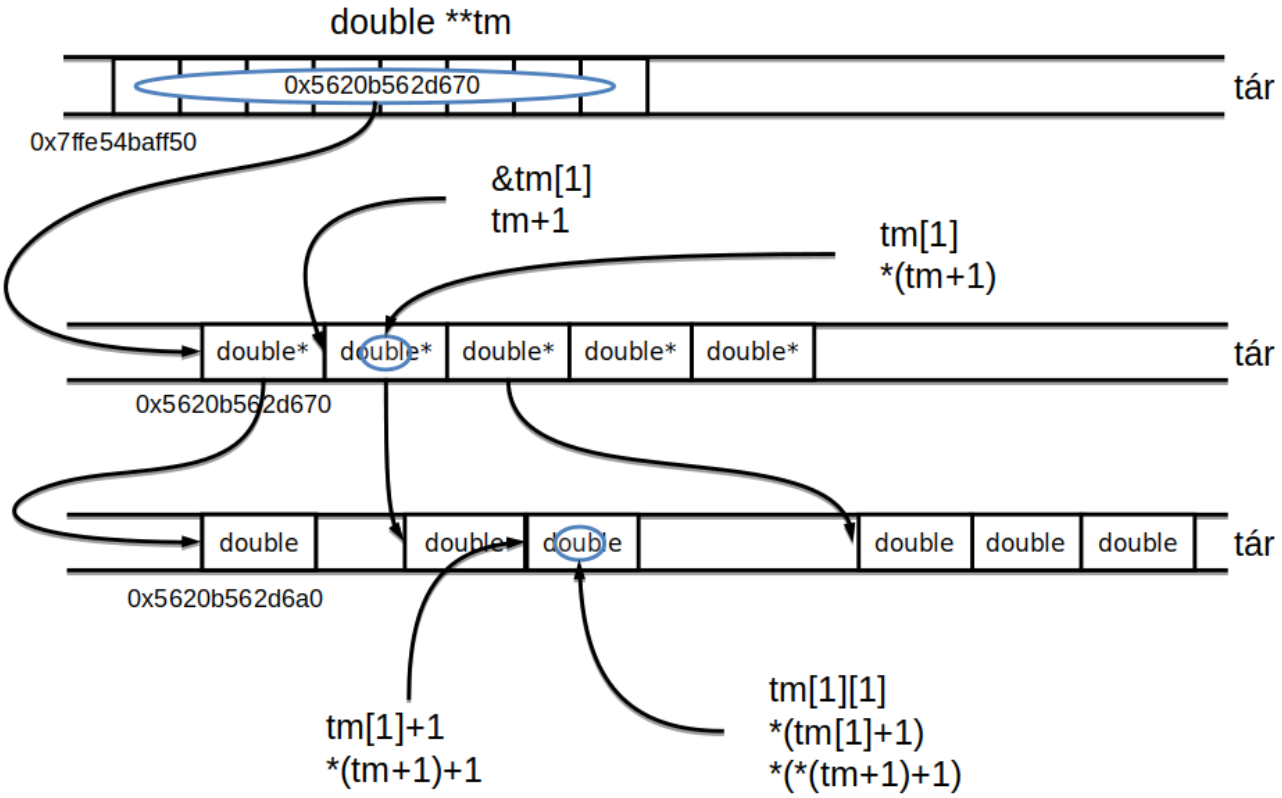
Valamint emellett még deklaráljuk a double **tm-et is. Valamint a tm-nek az aktuális memória címét kiíratjuk.

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

A man 3 malloccal megtudjuk nézni hogy a malloc mit ad vissza. Láthatjuk hogy visszaad egy pointert, ha valami hibát észlel akkor pedig NULL-t add vissza, így ha azzal egyenlő akkor return -1-elünk így a hiba esetén "kilök" a program.

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

Ezzel ráállítjuk a tm mutatót a memóriában malloc által lefoglalt terültre, persze ha az tudott foglalni, értelem szerűen ha nem tudott a program hibát fog lökni, viszont ha tudott akkor pedig kiírja a lefoglalt tár memóriacímét.



4.1. ábra. Double **

A képet Bátfai Norbert készítette, ami a következő oldalon fellelhető: https://youtu.be/1MRTuKwRsB0

A következő kódcsipet pedig, egy for ciklus ami maga a foglalás igazából, ha nr értéke 5 akkor 5-ször fog lefutni és mindegyiknek megfelelőnyi bájtot foglal le, első sorban ugye i=0 azaz 1 doublenek 8 bájtnyit, i=1 esetén a második sorban két doublenek 2*8=16 bájtnyit stb...

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

A legvégén pedig felszabadítjuk a lefoglalt memóriát, ezt a free függvény segítségével tesszük meg valami a for ciklusok használatával, majd magát a tm mutatót is felszabadítjuk a free használatával.

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

    free (tm);
```

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás forrása: Titkosító

Titkosítandó szöveg

Titkosított szöveg

A titkosító azt csinálja hogy a bemenetként kapott szöveget emberi szem számára olvashatatlanná változtatja még pedig a bitek száma alapján. Ehhez egy úgynevezett kulcsot használ amit szintén a felhasználó ad meg, a kulcs és a törő segítségével pedig az olvashatatlan karakter halmazból újra visszatudjuk majd állítani a szövegünket.

```
#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);

    }
}
```

Mint láthatjuk maga a titkosító, nem túl bonyolult és hosszúnak sem mondható egyáltalán. Valamint maga a titkosítás ideje se huzamos, emberek számára szinte azonnali. Ez majd a törésnél már nem lesz megfigyelhető.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: Java titkosító

Az előző feladat javás megvalósítása szükséges, ez alapjába véve nem lenne annyira nehéz feladat, mert ugyan vannak különbségek C és java között de átírni egy ekkora kódcsípetet nem sok idő ha ismerjük a javát és a c-t is. Számomra azért volt nehezebb feladat ez, mivel nem ismertem a javát, így először is a javával kellett ismerkednem utána a feladat forráskódjával.

Elsőnek is a legszembe tűnőbb az hogy a java objektum-orientált programozási nyelv, így már egy "Hello, World!" megírása is teljesen más mint C-ben, ugyanis classokat kell létrehozni és abban kell deklarálni/kidolgozni mindent.

```
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);

        }

    }

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

```
        try {
            new ExorTitkosító(args[0], System.in, System.out);
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
    }
}
```

Végül, ahhoz hogy futtassuk telepítenünk kell az openjdk-8-at amit a következőképpen tudunk leszedni és feltelepíteni:

```
sudo apt-get install openjdk-8-jdk
```

A futtatáshoz pedig ezt kell begépelni a terminálba:

```
javac ExorTitkosító.java
java ExorTitkosító kulcs (pl:12345678) <tiszta.txt >titkos.txt
```

4.4. C EXOR törő

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

Megoldás forrása: Törő

Titkosítandó szöveg

Titkosított szöveg

A törő feladata a fentebb említett titkosító által létrehozott "titkos" szöveg visszafejtése, ugyanis a titkosító által kreált karakter halmaz értelmetlen és olvashatatlan az emberek számára.

A törő kódja már picit bonyolultabb és jóval hosszabb a titkosító társánál. Valamint futási ideje nagyságrendekkel nagyobb a titkosításnál. A törés a fordítás módjától, a szöveg és a kulcs hosszától is nagyban függ. A forrásoknál az általam titkosított szöveget is linkeltem, valamint annak a titkosított verzióját is, ezekben jól megfigyelhető milyen is az átalakítás. Ennek a szövegnek a titkosításának a törése körülbelül 20 percet vett/vesz igénybe ha a következő módon fordítjuk a törőt:

```
gcc -o t t.c
```

És csak körülbelül 3 percbe telik ugyan ez ha így fordítjuk, és azután futtatjuk csak:

```
gcc t.c -O3 -o t -std=c99
```

Ez nagyszerűen szemlélteti hogy a fordítás módja már önmaga is meghatározó tud lenni a törési idő csökkentésére/növelésére.

4.5. Neurális OR, AND és EXOR kapu

R

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

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

Ez a feladat az R programozáshoz kapcsolódik megint. Itt építünk fel egy tanulni képes neurális hálót, mely a neuronról kapta nevét, ami az agyunkban lévő sejt, ami az információ feldolgozásáért és kezeléséért felelős. Itt is ugyan ez lesz a helyzet, a feladatban a VAGY az ÉS és a kizáró vagy azaz EXOR-t fogjuk neki úgy mond megtanítani.

A VAGY művelet betanítása lesz az első, ahhoz hogy tudjunk dolgozni a neuralnet könyvtárra lesz szükségünk. A VAGY-nak a működését szemléltetjük az a1 a2 és az OR segítségével, ha az a1 és az a2 is 0 akkor az OR is 0 lesz, tehát ha mindakét operandus 0 akkor a VAGY logikai művelet is 0-át ad vissza, egyébként pedig 1-et fog vissza adni. Azaz ha legalább az egyik operandus 1 akkor az OR is 1 lesz. (Nyilván ez magába foglalja azt is hogy mind az a1 mind az a2 1 akkor is 1 lesz az OR.)

```
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])
```

Az ÉS művelet is hasonlóképpen betanítható és működtethető, annyi eltéréssel hogy itt akkor fog 1-et visszaadni a logikai művelet ha mind a két operandus 1, különben 0-val tér vissza. Tehát ha legalább az egyik operandus 0 akkor az ÉS is 0 lesz. (Ez pedig magába foglalja azt is ha mind az a1 mind az a2 0 akkor az ÉS is 0.)

```
library(neuralnet)

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])
```

Az EXOR már kicsit másabb, eleinte mivel nem működött megfelelően fel is hagytak a neurális hálókkal, mert úgy voltak vele az emberek hogy ha egy ilyen egyszerű és sűrűn használt logikai műveletre nem lehet megtanítani akkor nem is annyira érdemes használni. Persze azóta megfejtették hogy mi is lehet a hiba, és most már képesek lehetünk neki megtanítani az EXOR azaz a kizáró vagy műveletet. Mindössze létre kell hozni rejtett neuronokat amelyek segítik a tanulásban.

```
library(neuralnet)

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 ábra igaz bonyolultabbá válik a rejtett neuronok miatt, viszont csak így érhetjük el a helyes eredményeket, te hát a megfelelő helyeken az 1-et és a 0-át, ha nem használnánk rejtett neuronokat és a VAGY vagy az ÉS logikai művelthez hasonlóan próbálnánk megoldani a tanítását, olyan 0,5 körüli értékeket kapnánk ami számunkra nem megfelelő. De nézzük meg a "hibás" forráskódot is:

```
library(neuralnet)


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])
```

Ez az ami nem működne megfelelően, de hála a rejtett neuronoknak mára már az EXOR művelet megtanítása se egy lehetetlen dolog. Az utolsó "hibás" kódot tehát el is felejthetjük.

4.6. Hiba-visszaterjesztéses perceptron

C++

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

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

A feladat értelmezésében és megoldásában tutoriáltam Nagy Krisztiánt!

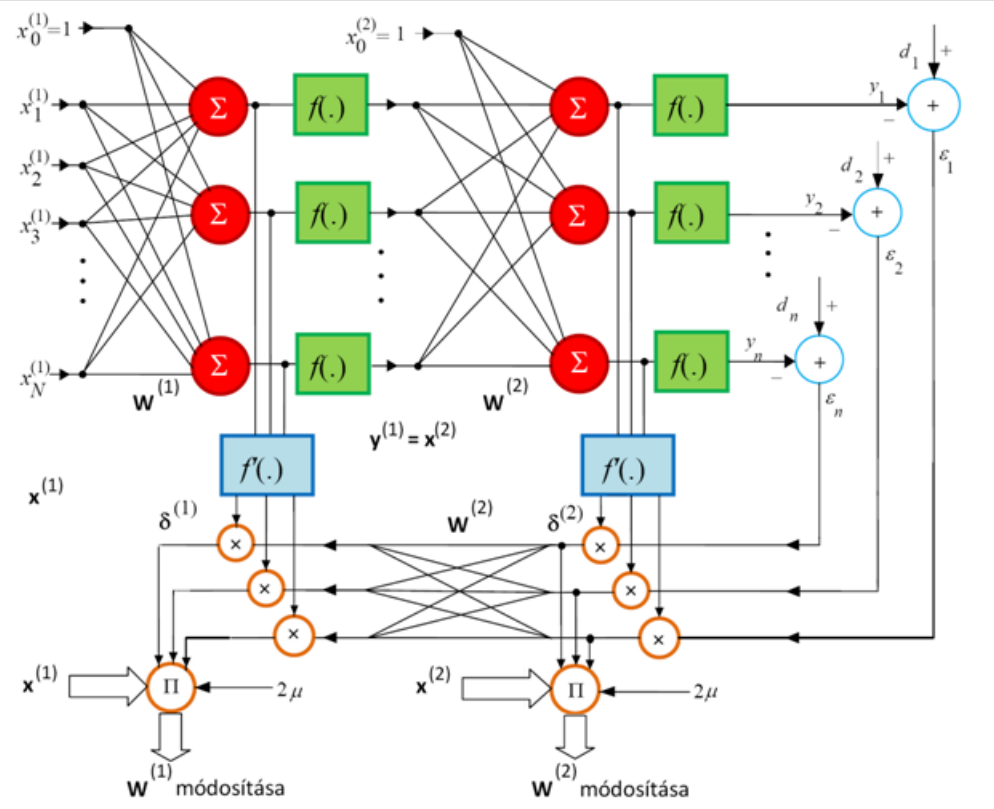
A perceptron egy algoritmus ami a gépi tanulásban játszik fontos szerepet. A bináris osztályzók tanulásában is fontos szerepet játszanak, ugyanis ezek munkaköre az hogy eltudja dönteni/el is dönti, hogy az input specifikus osztályhoz tartozik-e vagy sem. A perceptronról bővebben angolul olvashatunk a Wikipédia oldalán.

Továbbá még kiemelném a perceptron felépítését, mely 3 fő részből áll:

- A retinának nevezett első elem, ami a bemeneti jeleket fogadó cellákat tartalmazza.
- Az asszociatív cellák, melyek összegzik a hozzájuk érkező jeleket, impulzusokat.
- A döntési cellák rétege a perceptronok kimenetele. Az asszociatív cellákhoz hasonló képpen működnek ezek is.

Részletesebben erről és egyéb perceptron információról olvashatunk magyar nyelven a következő oldalon.

A hibavisszaterjesztéses algoritmus az MLP architektúra alapján származtatható. Maga az algoritmus egy tanuló eljárás. Az MLP tanítása az algoritmussal egy szemléletes képen:



4.2. ábra. MLP tanítása hibavisszaterjesztéses algoritmussal

A képet a következő oldalról szúrtam be: http://mialmanach.mit.bme.hu/neuralis/ch04s02

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

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

Megoldás forrása: `bhax/attention_raising/CUDA/mandelpngt.c++` nevű állománya.

A Mandelbrot halmaz Benoit Mandelbrot nevéhez fűződik, aki még 1980-ban fedezte fel azt a komplex számsíkon, akkor el is nevezték róla.

De mit is nevezünk komplex számoknak?

Azok a számok, amelyek "nem létező számok" úgynevezett imaginárius azaz "képzeletbeli" számok. Ezekre azért van szükségünk hogy kitudjuk fejezni magunkat olyan helyzetben is amire nem létezik valós megoldás. Mint például a páros gyökkitevőjű gyök alatti negatív szám értéke. Ugyanis még középiskolában is azt tanítják, hogy a páros gyökkitevőjű gyök alatti negatív számot nem tudjuk értelmezni, de egyetemen már a komplex számok bevezetésével ezeket is tudjuk értelmezni.

Hogyan láthatjuk meg a Mandelbrot halmazt?

Hát úgy, hogy az origó középpontú négyzetbe lefektetünk egy négyzet rácsot például 600x600 vagy 800x800 stb. Valamint ezután kiszámoljuk, hogy a lefektett rács pontjai mely komplex számoknak felelnek meg. A $z_{n+1}=z_n^2+c$, (0<=n) képletet felhasználva kapjuk majd meg, még pedig úgy hogy a c lesz a képletdben a vizsgált rácspont, míg a z0 lesz az origó. Alkalmazva a képletet a továbbiakban:

- $z_0 = 0$
- $z_1 = 0^2+c = c$
- $z_2 = c^2+c$
- $z_3 = (c^2+c)^2+c$
- $z_4 = ((c^2+c)^2+c)^2+c$
- ... s így tovább.

Vagyis a Mandelbrot halmaz megkeresése és kirajzolása úgy működik hogy kiindulunk az origóból (z_0-ból) és onnan ugrunk a rács első pontjába a z_1 = c-be, aztán ettől a c-től függően a további z-kbe fogunk ugrani és azokat is vizsgáljuk. Mind addig ugrálunk míg ki nem érünk a 2 sugarú körből, ha ez megtörténik akkor a vizsgált rácspont nem lesz eleme a Mandelbrot halmaznak. Mivel végtelenségig nem tudjuk megvizsgálni ezért csak véges számú z elemet nézünk meg minden rácsponthoz. Viszont ha ez idő alatt nem lép ki a körből akkor feketére színezzük a rácspontot, ezzel jelezve azt hogy az a c rácspont a Mandelbrot halmaz része. Aztán lépünk tovább a következő rácspontra majd a következőre és így tovább. Végeredményként a Mandelbrot halmaz elemei fekete színnel lesznek jelölve.

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

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

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

Megoldás forrása:

A Mandelbrot halmaz pontban vázolt ismert algoritmust valósítja meg a repó `bhax/attention_raising/Mandelbrot/3.1.2.cpp` nevű állománya.

Egyik legfontosabb különbség az hogy c++-ban tudjuk includeolni a complex könyvtárat ami nagyban megkönnyíti a munkánkat ugyanis biztosít nekünk complex típust így nem kell komplex számot "tartanunk". De nézzük is meg a kódot!

```
#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;

}
```

Ehhez a feladathoz én Bátfai Norbert Tanár úr kódját használtam és értelmeztem. Szerintem a kommentelésből és a videóból könnyen megérthető hogy mit is csinál a program. Valamint az előző feladat alapján már nem fog nehezünkre esni a kód értelmezése. Hisz igazából ugyan az pár kisebb nagyobb különbséggel.

A CUDA-s verzióhoz képest ez annyival másabb (főként), hogy ez csak egy magszálon fut míg a CUDA-s megvalósításnál felossza rácsra, és minden rácsban vannak blokkok, és a blokkokon belül van például 100 szál, ahol végig egyszerre fognak dolgozni, ezáltal sokkal, de sokkal gyorsabb lesz maga a folyamat a CUDA-s megoldásban.

A CUDA-s megvalósításról későbbi feladatban részletesebben fogunk beszélni! Így ha még nem világos mi is az a CUDA vagy milyen a CUDA-s megoldás nem kell kétségbeesni.

5.3. Biomorfok

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

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

A biomorfokra rátaláló Clifford Pickover meg volt győződve hogy természeti törvényre bukkant. 1986-ban volt ez, akkor valóban természeti formáknak tűntek ezek, ezért is "bio"morfok.

Az előző kódhoz hasonló lesz ez is, egy két kisebb-nagyobb változtatás kell csak mindössze.

Mint például a következő kódcsipetben jól látható hogy az előző kód a,b,c,d változói itt is jelen vannak csak xmin, xmax, ymin és ymax néven:

Valamint még hozzá adunk egy "reC", egy "imC" és egy R változót is, így már az "argc==9" helyett is "argc==12"-őt írunk.

```
    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] );

    }
```

Az előzőnek említett 3 új váltpzó közül kettő a komplexikáló függványnek a paramétere fog majd lenni, a következő féleképpen:

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

A változtatások miatt a számláló for ciklusunkat is kicsit át kell írni és a beléágyazott while ciklusunkat is átírjuk for ciklusra, ahol hasznosítjuk a 3. új változónkat az R-t, ez a következő képpen fog kinézni a kódunkban:

```
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 ));
        }
```

Ha z_n valós vagy imaginárius része nagyobb mint R akkor az iteráció a ciklus i változója lesz.

5.4. A Mandelbrot halmaz CUDA megvalósítása

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

Megoldás forrása: `bhax/attention_raising/CUDA/mandelpngc_60x60_100.cu` nevű állománya.

Ehhez a feladathoz fontos, hogy milyen hardverünk van, ugyanis ehhez a feladathoz nvidia kártyára lesz szükségünk, mivel csak azok a videókártyák rendelkeznek Cudacoreral.

A CUDA kártya ellentétben a fenti feladattal, Ő nem CPU-n dolgozik hanem GPU-n (a GPU=Graphics processing unit azaz grafikai processzor/feldolgozó egység). Olyan műveleteket is megtud csinálni a CPU helyett manapság mint például a videókódolás, és nem csak hogy áttudja venni, de sokkal gyorsabban végzi is el azt. Ugyanis ahogy már a fentebbi feladatban említettem a CUDA-s megvalósításnál felossza rácsokra, és minden rácsban vannak blokkok, amelyeken belül van például 100 szál, ahol végig egyszerre

fognak dolgozni. Azaz szinte mondható az hogy külön pontonként/pixelenként fog számolni, persze ez nem teljesen igaz, de egy erős túlzással állíthatjuk ezt.

Hogyan tudjuk fordítani?

nvcc fájlnév (pl. mandelpngc_60x60_100.cu) -lpng16 -O3 -o kimeneti fájlnév (például: mandelpngc)

Ezután futtathatjuk is, érdemes a 2. feladat forrásával futtatni, és megnézni a két forrás futtása közti különbséget. Mit fogunk tapasztalni?

Azt tapasztaljuk hogy többszörösen gyorsabb a CUDA-s megoldás, a cudecore-tól is számít persze ez, de 50-100x-os sebsséggel végzi el ugyan azt amit a 2. feladat.

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

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

Már a nevéből is látható hogy ez bizony valamit nagyítani fog nekünk, de hogy és mit?

Először is a libqt4-dev csomagra lesz szükségünk, ennek telepítése után tudjuk csak elvégezni a feladatot.

Ha viszont ezzel rendelkezünk és egy mappában van az összes szükséges fájl, akkor a qmake paranccsal készíthetünk egy makefile-t, amit a make paranccsal lefuttathatunk. Majd a programunkat tudjuk futtatni a ./Frak pranccsal.

Az eredmény frakablakok lesznek, amelyeken különféle fraktált alakzatok láthatóak, köztük kinagyított képek az eredeti halmazból, innen a nagyító kifejezés.

Továbbá meg lehet azt oldani hogy egy bizonyos gomb lenyomásakor nagyítson a képünkön, ezáltal még részletesebben lássuk a kirajzolódást.

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás videó: https://youtu.be/Ui3B6IJnssY, 4:27-től. Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_a

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#id570518

Alapvetően ugyan az mint az előző feladat, annyi különbséggel ugye bár hogy más a programozási nyelv így nem árt a kódot átírni/újraírni különben nem is működne az.

Először is szükségünk lesz a jdk8 csomagra a javás verzió elkészítéséhez és használatához, ezt a következőképpen tudjuk beszerezni:

```
sudo apt-get install openjdk-8-jdk
```

A javában include-álni/importolni máshogyan kell mint c++-ban, itt jelen esetben az extend-del fogunk. A MandelbrotHalmaz.javát úgy tudjuk használni a kódunkban ha azt a következőképpen importáljuk bele:

```
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz{...}
```

Az előző feladatban már említett nagyítást szintén itt is be bindelhetjük valamelyik gombra, annak lenyomása után pedig egy "közelebbi" részletesebb képet kaphatunk a kirajzolt alakzatunkról.

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: Polargen C++-ba , Polargen Javába

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!

Ehhez a feladathoz szükségünk lesz a randomizálásra. A randomizálást c++ a következőkép tudjuk alkalmazni, először is a cstdlib mellett szükségünk van a ctime-ra is, tehát includeolni kell ezt, ugyanis az idő függvényében tudunk teljesen random számot készíteni az srandom által, különben a program mindig ugyan azt a random számot generálná, így viszont a futtatás ideje alapján mindig újat és újat.

A kódban látható hogy folyamatosan figyeljük a tárolt számokat is, ezáltal azt is elkerüljük ha véletelenül már szerepelne a random számunk.

A matematikai háttér jelenleg nem érdekel minket, legalábbis a feladat is megkér arra hogy erre most ne térjünk ki, tehát nem túl részletesen csak lényegre törően azt csinálja a programunk, hogy ha nincs eltárolva a randomizált számunk akkor lefut az algoritmus és vissza ad egy random számot két különböző változóba. Ahonnan az egyik ki lesz íratva a másik pedig el lesz tárolva. Amennyiben el van tárolva egyszerűen kiíratjuk.

Valamint azt is megfigyelhetjük, ami a feladatban szerepel, tehát a JDK forrásaiban is hasonó felépítéssel vannak meg a kódok a miénkhez, tehát egy tapasztalt programozó is úgy oldaná meg ezt mint mi a jelenlegi tudásunkkal. Tehát mondhatni hogy ha ezt értjük akkor ugyanolyan természetes ez nekünk mint a Sun programozóinak.

6.2. LZW

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

Megoldás videó:

Megoldás forrása: C Binfa

Az LZW (Lempel-Ziv-Welch nevéhez köthető, innen ered) egy veszteségmentes tömörítési algoritmus. A tömörítés alapja az hogy a kódoló csak egy szótárbeli indexet küld át. Erről részletesebben magyar nyelven olvashatunk a Wikipédia oldalán.

A bináris fa egy olyan adatszerkezet amely belső és külső csúcsok hierarchikus elrendezéséből áll, ezzel megalkotva a szülő-gyermek elrendezést, mint az általános fa adatszerkezet. Fontos viszont kiemelni hogy minden "szülőnek" maximuma két gyermeke lehet, ez az amivel eltér a többi fa adatszerkezettől, valamint a gyermekeket megkülönböztetni úgy tudjuk a fában hogy bal vagy jobb oldalágat képezik-e a szűlőnek. Tehát fontos a sorrendjük is, bejárásra több példát is fogunk látni a következő feladatokban, ott majd részletesebben megismerkedünk a bejárás fogalmával és mintjével.

Hogyan tudjuk fordítani a binfát?

Egyszerűen annyi mint bármelyik másik c forráskódot, tehát valahogy így:

```
g++ z.c -o z
```

Hogyan tudjuk futtatni?

A futtatás se különbözik igazán, mindössze megkell neki adni egy bemeneti és kimeneti fájlt is, valahogy így:

```
./z bemenetifájl -o kimeneteifájl
```

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása: Preorder bejárás , Postorder bejárás

Az alap, tehát a fentebbi forráskód szerinti inorder bejárás:

```
void
kiir (BINFA_PTR elem)
{
  if (elem != NULL)
    {
      ++melyseg;
      _melyseif (melyseg > maxg);
  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;
    }
}
```

A bejárás megváltoztatása a forróskódban egy egyszerű sor kicseréléssel valósítható meg, ugyanis a program lineárisan halad a végrehajtáson és ha előre rakjuk pl a gyökér elem vizsgálatát és csak utána a bal, s majd jobb ág vizsgálatát már is preorderben lesz a bejárás. Ha végére rakjuk a gyökér elem vizsgálatát nyilván posztorder bejárást fog eredményezni.

Preorder bejárás:

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

      // 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);
  kiir (elem->jobb_egy);
--melyseg;
    }
}
```

És posztorder bejárással is:

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

  kiir (elem->jobb_egy);
  kiir (elem->bal_nulla);
      // 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);
--melyseg;
    }
}
```

A név nem véletlenszerű hiszen az angol szavakból ered, "pre" mint elől "post" mint hátul és "in" mint benne/középen, és a gyök elem helyzetét árulja el a bejárás során.

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: Tag a gyökér

```
LZWBinFa ()
    {
  gyoker = new Csomopont;
  gyoker = fa;
    }
```

A kód alapján jól látható hogy nem mutató a gyökér (mint ahogy az a következő feladatban szerepelni fog), hanem a feladat leírása szerint egy Tree és egy beágyazott Csomópont osztály.

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

class LZWBinFa
{
public:

    LZWBinFa ()
    {
  gyoker = new Csomopont;
  gyoker = fa;
    }
//~LZWBinFa......
//a Csompópont class:
private:
    class Csomopont
    {
    public:
        Csomopont (char b = '/'):betu (b), balNulla (0), jobbEgy (0)
        {};
```

```
        ~Csomopont ()
        {};
        {return balNulla;}
        {return jobbEgy;}
        {balNulla = 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->egyesGyermek (), os);
            for (int i = 0; i < melyseg; ++i)
                os << "---";
            os << elem->getBetu () << "(" << melyseg - 1 << ")" << std:: ↩
               endl;
            kiir (elem->nullasGyermek (), os);
            --melyseg;
        }
    }
    void szabadit (Csomopont * elem)
    {
        if (elem != NULL)
        {
            szabadit (elem->egyesGyermek ());
            szabadit (elem->nullasGyermek ());
            delete elem;
    delete Csomopont;
        }
    }
```

6.5. Mutató a gyökér

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

Megoldás videó:

A feladat megoldásában tutoriáltam: Győri Márk Patrikot.

Megoldás forrása: Mutató a gyökér

Első lépésként át kell írnunk a gyökeret hogy mutató legyen, ehez mindössze annyit kell tennünk hogy csillagot teszünk elé.

```
Csomopont *gyoker;
```

Ezután ha megpróbáljuk fordítani láthatjuk hogy kismilliónyi hibát jelez, ezeket kell javítanunk.

Kezdjük már is a legelején, a konstruktort írjuk át valahogy így:

```
    LZWBinFa ()
    {
  gyoker = new Csomopont();
  fa = gyoker;
    }
```

Ezt ha megléptük akkor sikeresen helyet foglaltunk a memóriában a csomópontnak amire a gyökér mutat, továbbá a fa mutatót is ráállítottuk a gyökérre.

Mivel már pointer a gyökér ezért a szabadításnál '.' helyett '->' operátort kell alkalmaznunk, írjuk hát át ezt is!

```
{
        szabadit (gyoker ->egyesGyermek ());
        szabadit (gyoker ->nullasGyermek ());
    }
```

Valamint, szintén a gyökér pointer léte miatt a kódban szereplő

```
"&gyoker"
```

kifejezéseket is át kell írnunk simán "gyoker"-re, ugyanis nekünk nem a pointer címére lesz szükségünk sehol hanem arra a címre amire mutat, tegyük meg hát ezeket a lépéseket is!

6.6. Mozgató szemantika

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

Megoldás videó:

A feladat megoldásában tutoriált engem: Győri Márk Patrik

Megoldás forrása: Mozgató szemantika

```
LZWBinFa & operator= (const LZWBinFa & cp) {
    if(&cp != this)
        rekurzioIndutasa(cp.gyoker);
    return *this;
};
```

Ehhez a feladathoz azt kell tennünk hogy létrehozunk egy operátort, aminek az a feladata hogy másolja le magát ha nem a gyökeret tartalmazza. Valamint a másolás rekurzióval végződik, mely azt fogja tenni hogy a fa minden egyes ágát újra létrehozza viszont egy másik gyökérre. A kód csípetekből jól fog látszani, viszont a kódott a tutoromtól (Győri Márk Patriktól) kaptam, nem pedig saját.

```
void rekurzioIndutasa(Csomopont csm){
    if(csm.nullasGyermek()){
        fa = &gyoker;
        Csomopont *uj = new Csomopont ('0');
        fa->ujNullasGyermek (uj);
        fa = fa->nullasGyermek();
        std::cout << "GYOKER: nullas van"  << std::endl;
        rekurzioAzAgakon(csm.nullasGyermek());
    }
    if(csm.egyesGyermek()){
        fa = &gyoker;
        Csomopont *uj = new Csomopont ('1');
        fa->ujEgyesGyermek (uj);
        fa = fa->egyesGyermek();
        std::cout << "GYOKER: egyes van"  << std::endl;
        rekurzioAzAgakon(csm.egyesGyermek());
    }
}

void rekurzioAzAgakon(Csomopont * csm){
    if (csm->nullasGyermek()) {
        std::cout << "====van nullas" << std::endl;
        Csomopont *uj = new Csomopont ('0');
        fa->ujNullasGyermek(uj);
    }
    if (csm->egyesGyermek()){
        std::cout << "====van egyes" << std::endl;
        Csomopont *uj = new Csomopont ('1');
        fa->ujEgyesGyermek(uj);
    }
    Csomopont * nullas = fa->nullasGyermek();
    Csomopont * egyes = fa->egyesGyermek();
    if(nullas){
        fa = nullas;
        rekurzioAzAgakon(csm->nullasGyermek());
    }
    if(egyes){
        fa = egyes;
        rekurzioAzAgakon(csm->egyesGyermek());
```

```
        }
    }
```

A rekurzioInditasa függvény fogja elindítani a rekurziót, ha van nullás gyermeke akkor azon fog elsőként tovább futni, majd ha van egyes gyermeke akkor arra is meghívásra fog kerülni. A fő eljárást maga a rekurzioAzAgakon függvény fogja elvégezni nekünk, ez fog átfutni az összes ágon, és majd Ő fogja létrehozni az új csomópontokat.

```
LZWBinFa binFa2;
    binFa2 = binFa;
```

A másolás az egyenlőség jel operátorral meghívva történik, így az alap binFa átmásolódik a binFa2-be.

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:

Hangya osztály

AntWin osztály

AntWin

Antthread osztály

Antthread

Main

A hangya osztályban a hangyának a tulajdonságai vannak. A hangyának van oszlopa és sora ami a kódban x és y, valamint a kódban lévő dir ami a hangya irányát fogja képezni. Ezekből fogja tudni a program hogy hol van és merre tart a "hangyánk".

```
class Ant
{

public:
    int x;
    int y;
    int dir;

    Ant(int x, int y): x(x), y(y) {

        dir = qrand() % 8;

    }

};
```

Nem pixelekkel dolgozik a program hanem cellákkal. Alapértelmezetten 6x6 pixeles egy cella.

```
    cellWidth = 6;
    cellHeight = 6;
```

Az AntWinbe van az AntThread osztályra mutató. A grind int*** a két rácsra mutat. Azért van két rácsunk hogy minden hangya egyszerre lépjen ehhez majd szükségünk lesz a gridldx-re. Valamint még a feromonnak a maximum és minimum értékét tartalamazza az AntWin. Az AntWinnek van konstruktor és destruktorja is, valamint van close evetnje ami a bezáráshoz kell, és keypresseventje ami a pauseolásra képes, ami a kódban P-re van bindelve.

7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása:

Ehhez a feladathoz is szükségünk lesz a jdk8-ra, de ezzel már rendelkezünk, korábbi fejezetben ennek beszerzése levan írva, ugyanis már kellett használnunk javás feladathoz.

Az életjáték elvet John Horton Conway nevéhez kötjük. Aki 1970-ben létrehozta saját sejtautomatáját, aminek megadhatunk bizonyos feltételeket az életbe maradáshoz. Így egy fajta életet hozzunk létre, amit megfigyelhetünk, ugyanis dimanikusan változik, kihalnak, születnek újak, vannak olyanok is amelyek stagnálnak. És a legritkább az amikor egyfajta ismétlődéses stagnálást eredményeznek a csoportulásuk ezeknek a kis "élőlényeknek".

Több szabályrendszer is létezik, a mi példákban Conway 3 szabályát feldolgozó automatát használjuk. Ennek szabályai a következők:

1.szabály: Csak 2 vagy 3 szomszéddal rendelkező sejtek maradnak életben.

2.szabály: Ha egy szomszédnak 3+ szomszédja van túlnépesedés miatt kihal. Valamint ha 2-nél kevesebb akkor pedig szintén meghal, magányosság miatt.

3.szabály: Megszületik egy sejt ha üres a cella és 3 élő sejt szomszédja van a cellának.

Ezen szabályok mellett az egyik legjobb és legérdekesebb eset az úgynevezett "siklőkilövő" kódját linkelem ide, a feladathoz hűen java nyelven, majd a következő feladatban szintén ez megtalálható lesz c++ nyelven is. Valamint a következő feladatban a program futásakor készített kép is megtalálható lesz.

Forráskód: Siklókilövő

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

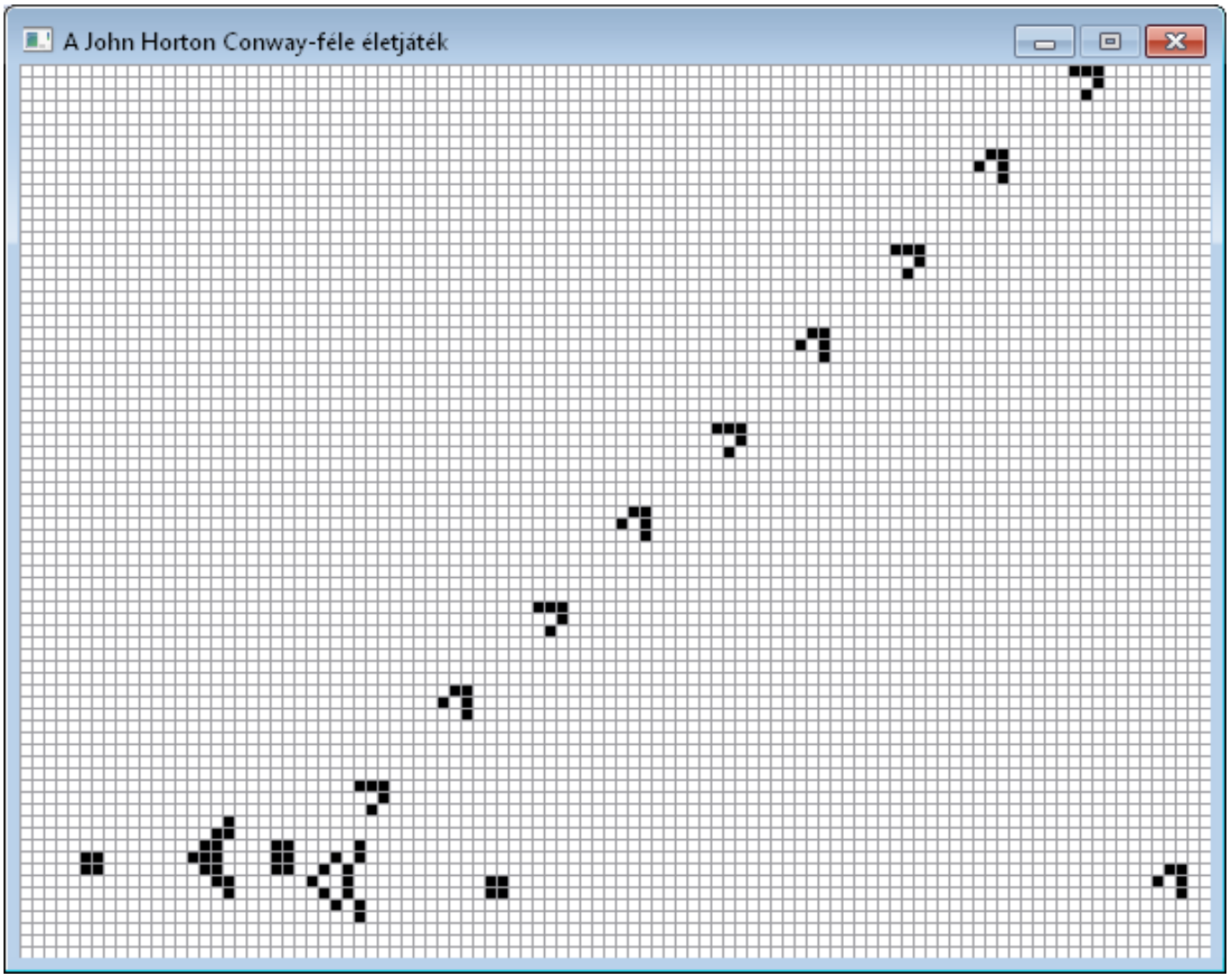
Megoldás forrása: Sejtablak, Sejtablak header

A feladat ugyan az mint az előző csak c++-ban. Itt is szintén a Conway 3-as szabályrendszert fogjuk alkalmazni, ezáltal a szabályok ugyanazok, tehát a születés az életbenmaradás és a kihalás esete ugyan úgy fordul elő mint előbbiekben.

Így például egy 2x2-es cella az élni fog és meg is marad nem fog változni se, míg egy 1x3-as cella pedig 3x1 lesz majd megint 1x3 és így tovább, viszont nem hal meg az egész alakzat.

Hogy is néz ki ez a cellákból álló ablak?

Valahogy így:



7.1. ábra. Sejtablak

A képet a következő oldalon fellelhetjük, valamint láthatunk még további állapotairól képeket a "siklókilövőnek". Az oldal: https://progpater.blog.hu/2011/03/03/fegyvert_a_nepnek

Maga a forrás pedig úgy működik hogy ilyen "siklókat" hozz létre amik folyamatosan haladnak és betöltik a rendelkezésre álló teret, ha ez a tér véges, akkor addig fognak haladni hogy a "szülők"-ig érnek azok pedig így meghalnak, de a siklók örökre megmaradnak és haladnak.

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása:

https://github.com/nbatfai/SamuBrain

A játékos feladata az hogy a Samu Entrophy-n belül lévő kék körben kell a kurzort lenyomva tartani, ez eleinte nem bonyolult, de minél tovább tartjuk rajta annál jobban elkezdenek mozogni a pontok, így követni hol jár egyre nehezebbé válik. A konzolon kiírja ha hibázunk valamint folyamatosan méri hogy mennyi bit/sec a reakció időnk. Tehát érdekes egy program, főként az esportban jártasak számára.

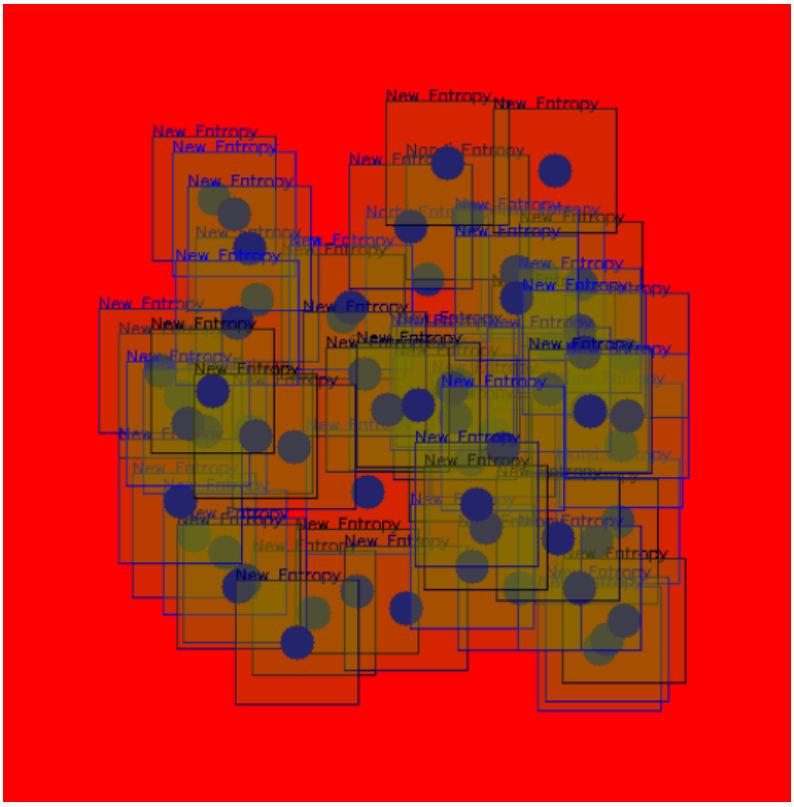
A játék 10 percig fut, de lehetőségünk van előtte is kilépni, és ha hamarabb befejezzük akkor is láthatjuk az eredményeinket.

Fordítani és futtatni úgy tudjuk ha a szükséges fájlok mind egy mappába vannak és a szokásos qmake-s eljárást kell alkalmaznunk itt is. Valahogy így:

```
qmake BrainB.pro--
make--
./BrainB--
```

Eleinte kevesebb "ablakkal" kezd, az idő múltával lesz egyre többb.

Kezelő felületete így néz ki futás közben:



7.2. ábra. BrainB Benchmark

A kép egy régebbi verziót ábrázol de az újabb belinkelt forráskódú "játék"/"benchmark" is ugyanolyan felépítésű. A képet az interneten a következő oldalon lelhetjük fel: https://github.com/nbatfai/esport-talent-search

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

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

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

Forráskód: `Szoftmax`

Ebben a feladatban Python nyelvben fogjuk megírni a MNIST-et amivel egy képfelismerő programot készítünk, melynek feladata az lesz hogy felismerje hány pont van a képen. Ehhez viszont szükségünk lesz a Python3 fejlesztő csomagra és a Tensorflowra is.

Hogyan tudjuk telepíteni a Tensorflowt és a Python3 fejlesztői csomagot?

A Python pip package segítségével a következőképpen tehetjük meg ezt:

```
sudo apt update
sudo apt install python3-dev python3-pip
sudo apt-get install python3-matplotlib
sudo pip3 install -U virtualenv
virtualenv --system-site-packages python3
source ./venv/bin/activate
pip install --upgrade pip
pip install --upgrade tensorflow
python -c "import tensorflow as tf; tf.enable_eager_execution(); print(tf. ↩
   reduce_sum(tf.random_normal([1000, 1000])))"
deactivate
```

A MNIST egy adatbázis mely tanítóanyagokat tartalmazz, olyan hálózatok tanítására aminek a feladata valamilyen képfelismeréshez vagy képelemzéshez köthető. Az SMNIST ennek úgynevezett "gyermeke" lett. A Bátfai Tanár úr féle nyílt forráskódú SMNIST-el "játszani" és tesztelgetni szabadon lehetett/kellett. Mi saját SMNIST for human próbálkozást is csináltunk/teszteltünk pár szakon lévő sráccal. Érdekes tud lenni ez, valamint versenyezni is lehet benne. Későbbiekben ez a képfelismerés, arcfelismerés stb gépek számára szerintem nagyobb teret fog behódítani jelenlegi állapotához képest. Ugyanis hasznos funkció lehet akár egy telefonban is. (Már léteznek próbálkozások kihasználására.)

Számomra a Python nyelv és annak használata kevésbé ismert, de más nyelvhez pl C vagy C++-hoz hasonlóan itt is elsőként importáljuk a szükséges könyvtárakat. Ilyen például ebben a feladatban használt Tensorflow könyvtár is.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse

from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf
old_v = tf.logging.get_verbosity()
tf.logging.set_verbosity(tf.logging.ERROR)

import matplotlib.pyplot

import tensorflow as tf
old_v = tf.logging.get_verbosity()
tf.logging.set_verbosity(tf.logging.ERROR)
```

```
FLAGS = None

def main(_):
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
```

A modell elkészítése.

```
x = tf.placeholder(tf.float32, [None, 784])
  W = tf.Variable(tf.zeros([784, 10]))
  b = tf.Variable(tf.zeros([10]))
  y = tf.matmul(x, W) + b


  y_ = tf.placeholder(tf.float32, [None, 10])

  cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits( ↩
     labels = y_, logits = y))
  train_step = tf.train.GradientDescentOptimizer(0.5).minimize( ↩
     cross_entropy)

  sess = tf.InteractiveSession()
```

A tanítása:

```
  tf.initialize_all_variables().run(session=sess)
  print("-- A halozat tanitasa")
  for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

```
    if i % 100 == 0:
      print(i/10, "%")
  print("------------------------------------------------------------")

  print("-- A halozat tesztelese")
  correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
  print("-- Pontossag: ", sess.run(accuracy, feed_dict={x: mnist.test. ↩
     images,
                                    y_: mnist.test.labels}))
  print("------------------------------------------------------------")

  print("-- A MNIST 42. tesztkepenek felismerese, mutatom a szamot, a  ↩
     tovabblepeshez csukd be az ablakat")

  img = mnist.test.images[42]
  image = img

  matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot.cm ↩
     .binary)
  matplotlib.pyplot.savefig("4.png")
  matplotlib.pyplot.show()

  classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

  print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
  print("------------------------------------------------------------")

  print("-- A MNIST 11. tesztkepenek felismerese, mutatom a szamot, a  ↩
     tovabblepeshez csukd be az ablakat")

  img = mnist.test.images[11]
  image = img
  matplotlib.pyplot.imshow(image.reshape(28,28), cmap=matplotlib.pyplot.cm. ↩
     binary)
  matplotlib.pyplot.savefig("8.png")
  matplotlib.pyplot.show()

  classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

  print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
  print("------------------------------------------------------------")

if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/ ↩
     mnist/input_data',
                      help='Directory for storing input data')
  FLAGS = parser.parse_args()
  tf.app.run()
```

Bonyolult kissé a kód egyrészt mert a hivatalos MNIST példaprogramja az alapja másrészt számomra azért is mert a Python forráskód olvasata még nem megszokott.

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Ezt a feladatot passzolnám!

8.3. Minecraft-MALMÖ

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

Megoldás forrása:

Minecraft-MALMÖ

A Malmo egy platform amely segít nekünk kutatásban a mesterséges intelligencia területén. A feladatban az ágens-programozásba csöppenhetünk bele, melyet a népszerű játékkal a Minecrafttal ötvözve probálhatunk ki. A feladat olyan Ai-t írni Stevenek hogy kikerülje az elé kerülő akadályokat, ezáltal nem csapdába esve haladni a mapon és felfedezni azt.

Tesztelni a mellékelt kódot nem tudtam, ugyanis nem rendelkezem a Minecrafttal, de működnie kell.

Mint mindenhol itt is elsőnek a szükséges könyvtárak importálásával kell kezdenünk:

```
from __future__ import print_function

from builtins import range
import MalmoPython
import os
import sys
import time
```

Majd létre hozunk egy alapértelmezett Malmo objektet:

```
agent_host = MalmoPython.AgentHost()
try:
    agent_host.parse( sys.argv )
except RuntimeError as e:
    print('ERROR:',e)
    print(agent_host.getUsage())
    exit(1)
if agent_host.receivedArgument("help"):
    print(agent_host.getUsage())
    exit(0)
```

```
my_mission = MalmoPython.MissionSpec(missionXML, True)
my_mission_record = MalmoPython.MissionRecordSpec()
```

Majd megkíséreljük a "küldetésünket" elindítani:

```
max_retries = 3
for retry in range(max_retries):
    try:
        agent_host.startMission( my_mission, my_mission_record )
        break
    except RuntimeError as e:
        if retry == max_retries - 1:
            print("Error starting mission:",e)
            exit(1)
        else:
            time.sleep(2)
```

Majd loopolnunk kell addig amíg a mi kis küldetésünk el nem indul:

```
print("Waiting for the mission to start ", end=' ')
world_state = agent_host.getWorldState()
while not world_state.has_mission_begun:
    print(".", end="")
    time.sleep(0.1)
    world_state = agent_host.getWorldState()
    for error in world_state.errors:
        print("Error:",error.text)

print()
print("Mission running ", end=' ')

    agent_host.sendCommand( "move 1" )

stevex = 0
stevez = 0
stevey = 0
steveyaw = 0
stevepitch = 0
elotteidx = 0
elotteidxj = 0
elotteidxb = 0
akadaly = 0
```

Ezután a kód többi része maga az a kódrész ami loopolva lesz, tehát ami majd akkor fut folyamatosan amíg a küldetésünk tart.

```
while world_state.is_mission_running:
.
.
.
#(a forráskód fent szerepel nem csúnyítanám vele a könyvet)
```

```
.
.
.
print()
print("Mission ended")
```

A kódban látható hogy a koordináták segítségével valamint a kód részében az if-ek segítségével tudjuk neki megmondani mikor mit csináljon. Például hogy mikor forduljon vagy mikor kezdjen ugrálni a karakterünk. Ezeknek a finom módosítgatásával tudjuk biztosítani a küldetésünk sikerét, akár különböző biomokon vagy különböző akadályokon való "túlélésen", és leküzdésében.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása:

A feladatot Lisp-ben kell megoldanunk, de hogyan férhetünk hozzá?

Mindössze elég hozzá egy GIMP-et indítani, azon belül pedig nyitni kell egy Script-fu konzolt, és már kezdhetjük is beírni a Lisp-es kódunkat. Na de hogyan is néz ki egy Lisp-es kód?

Először is azt ki kell emelni hogy ezen a nyelven nagyon figyelnünk kell a zárójelezésre, mivel azt könnyű elbénázni, és nagyon fontos, mindennek zárójelben kell lennie. De nézzünk is egy egyszerű összeadást Lisp-ben:

```
(+ 2 2)
```

Ez négyet fog nekünk adni. Egy preorder bejárással történt műveletnek tűnik, de egy bonyulultabb egyenletnél láthatjuk hogy nem erről van szó, nézzük is meg!

```
(+ 2 2 2)
```

Ez lesz a Lisp kódban a 2+2+2, míg preorder bejárással + + 2 2 2 lenne.

Na most hogy kicsit megismertük a Lispet kezdjünk neki a feladatnak!

Először is a faktoriális függvényünket "meg kell tanítanunk". Amit a következő kód beírásával tudunk megtenni:

```
(define (fakt n) (if (< n 1) 1 (* n (fakt(- n 1)))))
```

Most már tudja hogy mi a "fakt", így kitudja számolni egy szám faktoriálisát, a példában nézzük meg hogy kell a 4 faktoriálist ezután kiszámolni.

```
(fakt 4)
```

Ez 24-et fog adni, azaz a fakt definiálása és alkalmazása is tökéletes lett ugyanis ez megegyezik 4!-sal.

9.2. Gimp Scheme Script-fu: króm effekt

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

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

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

A GIMP script mappájából tudjuk csak a kódunkat működésre bírni, valamint a programon belül is megadjuk elérési helyét ezután már is képesek leszünk kevünk szerint alakítani a színeket a szövegen. Ha futtatjuk a programot akkor kapunk egy default beállítást mindenre de persze ezeken kedvünk szerint módosíthatunk, nem köt minket semmihez. Viszont a feladat köt minket még pedig a króm effektet kell elérnünk.

Na de térjünk a feladatra, első lépésként egy fekete színű hátteret hozunk létre amin a szövegünk fehér színű lesz. Ezt így érjük el:

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

    (set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) ↩
       ))
    (gimp-image-insert-layer image textfs 0 0)
    (gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (- (/  ↩
       height 2) (/ text-height 2)))

    (set! layer (car(gimp-image-merge-down image textfs CLIP-TO-BOTTOM- ↩
       LAYER)))
```

Ha ezzel megvagyunk második lépésként elmossuk a szövegünket, még pedig az úgynevezett Gaussian eljárással.

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 15 TRUE TRUE)
```

3. lépésként a szövegnek az éleit görbítjük le, majd 4. lépésként pedig ezt mossuk el az előbbi módszerrel.

```
    (gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE)
    (plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE)
```

A következő lépésekben lehetőségünk lesz az elején létrehozott fekete hátteret kitörölni a kép invertálása mellett, ami átlátszó lesz tehát maga a kép fog látszani ezáltal, a szöveggel pedig kedvünk szerint foglalkozhatunk, majd az utolsó (a forráskódban 9.) lépés alkalmával lesz lehetőségünk megadni a gradient effektet, amivel sikeresen késznek tudhatjuk a feladatot.

```
(gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))
```

9.3. Gimp Scheme Script-fu: név mandala

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

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

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

Ezt a feladatot passzolnám!

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

[**?**]

Ezt az olvasónaplót a megadott oldal számok és témák elolvasása alapján írom, alcímként megadom hogy melyik oldalról/melyik fejezet részről szól az adott szakasz. Annak a szakasznak röviden lényegre törően a tartalmát és a vele kapcsolatos egyéb meglátásokat közlöm az olvasónaplóban.

(11.-15. o.) 1.2 Alapfogalmak:

Az alap fogalmakat foglalja össze, viszont ellentétben a Keringhan-Ritchie könyvvel, itt nem kódokkal és azoknak az elmagyarázásával mutatja be, hanem inkább ilyen elméet szerűen tanulás nem gyakorlatiasan, tehát a fogalmak pontos korrekt megfogalmazásával. Így már én az első fejezet után is úgy vélem hogy a KernighanRitchie féle oktatás és magyarázás számomra sokkal kedvezőbb és könnyebben elsíjátítható mint ennél a könyvnél, de persze minden embernél, más, ezért főként azoknak ajánlanám ezt a könyvet aki nem a gyakorlatiasabb példákon átívelő tanulást részesíti előnyben. De persze még a végén erre kitérünk haladjunk is tovább.

(46.-55. o.) Kifejezések:

A kifejezések azaz a 3. fejezetben, a címből következtetve is helyesen gondolhatjuk azt hogy a kifejezésekről lesz szó. A kifejezések elméleti tudnivalójával kezd, megtudjuk mit nevezünk például operandusnak, operátornak stb. Valamint még az alakokról is szó esik (prefix,postfix,infix) ugye ezek között mindössze az operátorok és operandusok helyzete a különbség, hogy milyen sorrendbe írjuk őket fel. Azaz ha például egy kifejezés operátorait és operandusait egy bináris fával rajzolnánk fel akkor milyen bejárást alkalmazva jutunk el a felírásokhoz. (értsd úgy hogy preorder=prefix....)

A 3.1 "Kifejezés a C-ben" már inkább gyakorlatiasabb szemléletesebb fejezet rész, itt a címből is addódóan a C-ben lévő kifejezéseket mutatjabe azoknak a helyes használatát, kötési irányát stb.

(55.-57. o.) Utasítások:

Ezek az oldalak az utasításokról fognak szólni. Az utasítások építik fel a programot ezért fontos az ismeretük. Utasításoknak több fajtája van, mint például értékadó-, ciklusszervező-, hívó-, i/o-, egyéb utasítások. A kijelölt oldalakon ezeknek még nem található részletezése, de a rákövetkező pár oldal taglalja, szerintem érdemes azt is elolvasni hozzá, hisz ott ismerjük meg csak részletesen ezeket az utasításokat valamint ott már forrás kódot is kapunk mellé hogy láthassuk miről is van szó.

(58.-82. o.) LISP:

Mint az fentebb említettem érdemes elolvasni, igazából ki is lett jelölve csak másik csokorba. Egészen a 72. oldalig az utasításoknak a részletes szemléltetéséről szól a könyv, én ezt még az előző kijelöléshez soroltam volna. Ugyanis 72. odlalnál a programok szerkezetére tér át a könyv. Ábrás itt-ott forrás kódos módon mutatja be de még mindig inkább elméleti szinten, a paraméterekre is kitér bőségesen, azok megadására és átadására is egyaránt.

(56.-71. o.)

Még egyszer fel lett adva, ez a rész összefoglalóan részletesn az utasításokról szól, hasznos és fontos rész a könyvben a két oldalszám közé eső részek.

(72.-78. o.) és (78.-82. o.)

Ezek is újra fel lettek adva, nem írnám le még egyszer, fentebb olvasható miről is szól ez az intervallum.

(82.-85. o.)

Ez a rész még szintén a programok szerkezetéhez köthető. A blokkal ismerkedünk meg, valamitn a hatáskör fogalmát vesszük át. A hatáskör a c++-ban például fontos ugyanis nem mindegy hogy akár egy osztálynál vagy egy function-nél hol deklarálunk, hol használunk függvényeket stb, ezért érdemes megismerkedni vele.

(112.-113. o.)

Ez a rész a kivételkezeléssel foglalkozik. Megszakítások kezelését felhozzuk a program szintjére, ennek a részletes ismertetőjét olvashatjuk/tanulmányozhatjuk át ebben a részben.

(134.-138. o.)

Ez a rész pedig az I/O-val foglalkozik, tehát az input és output-tal. Azok ismertetésével (input/output állomány), mikor jön létre, mikor szükséges stb. Állományok deklarációjáról, összerendeléséről, megnyitásáról, feldolgozásáról, lezárásáról esik még szó. Majd a 13.1-ben a különböző nyelveknek az I/O eszközeit mutatja be nekünk az író.

Összesítés:

Alapjába véve egy hasznos könyv, mely rengeteg információt hordozz magában, de számomra kicsit száraz, szerintem jobb a Kernighan féle könyv mely példákon keresztül tanít. De ennek ellenére ezt is ajánlanám azoknak akik a C nyelvről szeretnének minél több tudást a magukénak érezni.

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

Én fejezetenként adok egy kis összefoglalót hogy miről is szól az adott fejezet, így teljesíteném az olvasónapló megírását ennél a feladatnál.

1. Fejezet:

A könyv az alapfogalmakkal és alapismeretekkel indít. Az 1.1-ben a szokásos "Hello, World!" programot írja meg és magyarázza el a "Halló, mindenki" példán keresztül. Majd a változókat és annak típusait nézi át részletesebben az 1.2-ben. Valamint még ugyan itt a kommentekről is olvashatunk továbbá a while

ciklus is megjelenik. Míg az 1.3-ban a for ciklust részletezi a könyv, amit az előző whilehoz hasonlítva mutat be. 1.4-be szimbolikus állandókat mutatja be az olvasóknak a könyv. Az 1.5-ben a karakterekkel foglalkozó függvényekről kapunk egy részletes összefoglalót amely bemutatja többek közt a beolvasást, kiírást, sorok/szavak számlálását. Az 1.6 alfejezetben a tömbökről lesz szó, azoknak az alkalmazásáról, a könyv karakter megszámlálása valamint üres hely megszámlálása példáján mutatja be hasznosságukat. Ezután az 1.7-be a függvényeké lesz a főszerep, a hatványozást mutatja be a könyv függvények segítségével. 1.8-ba pedig az argumentumokat taglalja a könyv. A fejezet végén pedig újra, csak azúttal részletesebben még egyszer megnézzük a tömböket és változókat.

2. Fejezet:

A 2.fejezet címként (tehát a könyv alcímként) megkapta a "Típusok, operátorok és kifejezések" címet, mely szerintem nagyon lényegretörő, ugyanis valóban ezekről lesz szó. A fejezet nagyon nagy részletességgel taglalja a változónevek, adattípusok és méretek, állandók, aritmetikai operátorok, logikai és relációs operátorok használatát fontosságát. A deklaráció fontosságáról és annak ki nem hagyhatóságát is ki emeli a 2.4-be. Megismerkedhetünk a típus konverziókkal, valamint különféle operátorokról és kifejezésekről olvashatunk a 2. fejezet második felében, ami nagyon fontos, valamint szerintem a programozás alapjai, ezért érdemes elolvasni figyelmesen.

3. Fejezet:

Ez a fejezet a vezérlési szerkezetekkel fog foglalkozni. Az olvasó számára bemutatja a leghasznosabb és legfontosabb utasításokat mint például az if, else if, switch, break, continue, goto. Valamint a ciklusszervezést mutatja meg az elején említett while és for utasítással, valamint a do-while-al ami a névből is láthatóan a whilehoz hasonló felépítésű és működésű, csak egy kisebb változtatással. Még pedig az hogy a do-while egyszer mindenképpen le fog futni, de nem is írok többet, a könyv nagyon jól bemutatja ezt is, szintén egy hasznos és "programozás alapja" fejezetnek mondanám ezt is.

4. Fejezet:

A 4. fejezet a függvényekkel kapcsolatos alapfogalmakkal kezd, ugyanis a fejezet a függvények és programok szerekezetét taglalja. Szintén precíz részletes leírást kapunk a dolgokról az eddig megszokott módon. Először az egyszerőbb függvényeket mutatja be majd a nem egésszel visszatérő függvényeket is. A külső változókról is szót ejt valamint a statikus változókat is kifejti a 4.6-ba, míg a 4.7-be a regiszter változókkal ismerkedhet meg az olvasó. Szintén ebben a fejezetben még bemutatásra kerül a tartományok érvényességi szabálya, a blokkstruktúrák és header állományok. Változók incializálását részletesebben újra elismétli, a fejezet végén pedig a rekurziót mutatja be valamint a C előfeldolgozó rendszert, ahol az állományok beépítése a #define által való makróhelyettesítés a valamint a feltételes fordítás kap szerepet.

5. Fejezet:

Az ötödik fejezetben a főtéma a mutatók és a tömbök lesznek. Elsőnek is a mutatókat a címzéseket mutatja be, majd később a mutatók és tömbök kapcoslatát is, változásként az emelhető ki hogy ábrákkal segíti a megértést, ami szerintem nagyon jó, ugyanis könnyebb úgy megérteni ha az ember nem csak olvasni tudja a kódcsipeteket és a magyarázó szöveget hanem kis ábrákat mellé nézve értelmezi a mutatókat például vagy a tömbök számozását 0-tól a memóriában foglalt területen. Ugyan ebben a fejezetben lesz még szó címaritmetikáról karaktermutatókról és függvényekről. Valamint a mutatóra mutató mutatókról is, ami számomra régebben bonyolultnak tűnt, ezt elolvasva hasonló helyzetben lévő embereknek sokat segíthet. Mutatótömbök és több dimenziós többök bemutatása incializálása, értelmezése több alfejezet is taglalja ezeket. Majd a fejezet végén előkerülnek a parancssor-argumentumok és a bonyolultabb deklarációk is. Ez a fejezet már nem annyira az alapokat részletezi hanem azokra már építve kissé de még mindig új "alap" fogalmakat bevezetve és elmagyarázva mutatja azt be.

6. Fejezet:

A 6. fejezet a struktúrák köré épül. Mélyebben belemegyünk ebbe a témába, és részletesen, ábrákkal is segítve, a struktúrák működését veszi át, valamint az azokra épülő dolgokat mint például a struktúratömböket, a struktúra és függvények viszonyát, valamint a struktúra és mutatók együttesét, önhivatkozó struktúrákat. Ez a fejezet már egy haladóbb szint szerintem, itt már számomra is sok új dolog volt és ha csak a mellékelt kódokat nézzük is, azok is már bonyolultabbak mint az előző fejezetben mellékelt kódok, viszont nem kell megijedni ugyanis a szokásos módon ezt is jól elmagyarázza a könyv és meg lehet érteni elolvasás után.

7. Fejezet:

A hetedik fejezet főként az adat ki- és bevitelről szól. Annak részletesebb elismétléséről. Megjelenek a függvények részletes leírása, olyanoké mint például a printf vagy scanf, valamint a fejezetben a karaktervizsgáló és -átalakító függvényekről is kapunk egy kis táblázatot. A hibakezelésről is szó esik az stderr és exit függvények mellett. Matematikai függvények és egyéb függvényekről is szó esik még, hogy milyen könyvtárba tartoznak, azaz mit kell includeolnunk hogy elérjük őket, mit tudnak kezdeni a bemenetként akpott dolgokkal stb. Valamint a legutolsó alfejezetben a 7.8.7-ben a véletlenszerű számgenerálást ismerjük meg.

8. Fejezet:

A 8. fejezetben a UNIX rendszer adatbevitellel és kivitellel ismerkedünk meg. Különböző függvényeket mutat be, számomra ez a fejezet már nem volt annyira érdekes, de azért érdemes ezt is végig olvasni sose tudni mikor vesszük hasznát, de véleményem szerint ez már tényleg inkább csak plusz infók hogy a tudásunkat bővítsük.

Összesítés:

Összeségében szerintem ez egy nagyon hasznos és precízen összeállított könyv, nagyon sokat tud segíteni a kezdő C programozók számára. Ezért én mindenképpen ajánlanám. Hátrányként esetleg azt tudnám megemlíteni hogy itt-ott vannak hibák elírások, de ez nagyon pici hátrány a sok-sok előny mellett.

A könyvnek van függeléke is mely még több hasznos és fontos dolgot vesz számon. Nem tudtom hogy az olvasónapló vonatkozik rá, de szerintem mindenképpen érdemes beleolvasni. Mivel még több hasznos információval rendelkezik. Egy szó mint száz érdemes olvasgatni mind a fő fejezeteket mind a függelékeket is, nagyon hasznos, szemléletes a nyelv bemutatása benne, valamint figyelmet fentartó módon mutatja be a dolgokat nem pedig untat minket.

10.3. Programozás

[BMECPP]

Ezt az olvasónaplót a kijelölt oldalszámok elolvasása után írom meg.

(17.-59. o.):

Ezek az oldalak az osztályokat taglalják valamint az objektumokat. Nagyon részletesen, példákkal, kódokkal dusítva, személy szerint nekem ez a rész sokat segítet, ugyanis rengeteg kérdőjel volt bennem mind az osztályokkal mind az objektumorientáltsággal kapcsolatban, ezek az oldalak pedig választ tudnak adni. Tipikusan olyan rész ahol még tapasztaltként is fellapozhatjuk ha valami nem tiszta.

(187.-197. o.):

Ez a rész a kivételkezelésről szól. A könyv ezen az oldalalakon keresztül végig azt taglalja, eléggé részletesen valamint forráskódokkal és feladatokkal tűzdeli meg az információ csokrokat, ezáltal az olvasó számára még jobban értelmezhetőek a dolgok. Try-catch blokkokat mutatja be azoknak egymásba ágyazásáról is szó esik valamint az elkapott kivétel újradobásáról is. Hasznos rész, nagyon részletes és szemléletes módon mutatja be a kivételkezelést c++-ban.

190. példa:

A példában láthatóvá válik az hogy mi történik meg ha a felhasználó bemenetként 0-át ad meg vagy ha éppen ellenkezőleg bemenetnek nem nullát ad meg. Mivel a kódunk a beérkező szám reciprokát számolja ki és írja ki, ezért ugye 0 nem lehet mivel nem értelmezünk olyan törtet aminek 0 a nevezője. Ennek segítségével mutatja be hogy lehet használni a try-catchet, ugyanis ha 0-át adunk bemenetnek akkor dobni fog egy hibát amit elkapunk és kiírjuk hogy a bemenetnek nem nullának kell lennie.

197. példa::

A példában azt láthatjuk, hogy az f2 kivételt dob, így f2-ben a definált i lokális változó felszabadul, míg az f1-ben lefoglalt Fifo fifo objektum felszabadul, mert meghívódik a destruktora. Majd lefut a main függvényben lévő catch blokk. Ki emeli továbbá a könyv is azt a fontos tényt hogy ugyan a hiba dobás és elkapás között kód fut le, de semmiképpen ne dobjunk új hibát míg az elsőt el nem kaptuk, mert annak kezelése nem lesz lehetséges.

(211.o) 58-as fólia kapcsán:

Üzenet kezeléssel foglalkozik, valamint az előző try-catch témára épül, egy újabb jó példa eme téma feldolgozására. Valamint részletesen taglalja a megoldást, ezzel megértetve az olvasóval a try-catch lényegi működését.

Összesítés:

Összesítésben egy nagyon jó és hasznos könyv, szemléletesen és részletesen mutatja be a C++ tualjdonságait és használatát. Ajánlott belőle szemezgetni, tanulgatni. Hasznos, lényegretörő és szemléletes. Az alapoktól kezdődően egy magasabb szintre fel tudunk fejlődni csak ezen könyv mellett is c++ programozási nyelvben.

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. Ã©s Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

11.5. C++

[BMECPP] Benedek, Zoltán Ã©s 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.