



Eötvös Loránd Tudományegyetem

Informatikai Kar

Algoritmusok és Alkalmazásaik Tanszék

---

# Rendezési algoritmusok szemléltetése

Veszprémi Anna  
mestertanár

Márföldi Péter Bence  
programtervező informatikus BSc

Budapest, 2015

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>1</b>
1.1. A feladat és annak értelmezése . . . . .	1
1.2. Alkalmazott technológiák . . . . .	1
1.2.1. Java . . . . .	1
1.2.2. JavaFX . . . . .	2
1.2.3. JUnit . . . . .	2
<b>2. Felhasználói dokumentáció</b>	<b>3</b>
2.1. Rendszerkövetelmények . . . . .	3
2.1.1. Minimális rendszerkövetelmények . . . . .	3
2.1.2. Ajánlott rendszerkövetelmények . . . . .	4
2.1.3. Telepítés és eltávolítás . . . . .	4
2.2. Felhasználói felület bemutatása . . . . .	6
2.2.1. Főmenü . . . . .	6
2.2.2. Bemenet megadása panel . . . . .	7
2.2.3. Főpanel . . . . .	9
2.2.4. Megfigyelés panel . . . . .	10
2.2.5. Összehasonlítás panel . . . . .	12
2.3. A vizsgált algoritmusok . . . . .	14
2.3.1. Buborékrendezés . . . . .	14
2.3.2. Beszúró rendezés . . . . .	15
2.3.3. Shell rendezés . . . . .	17
2.3.4. Gyorsrendezés . . . . .	18
2.3.5. Kupacrendezés . . . . .	20
2.3.6. Versenyrendezés . . . . .	21
2.3.7. Radix "előre" . . . . .	23
2.3.8. Radix "vissza" . . . . .	24
<b>3. Fejlesztői dokumentáció</b>	<b>27</b>
3.1. Tervezés és megvalósítás . . . . .	27
3.1.1. Tervezés . . . . .	27

3.1.2. Megvalósítás . . . . .	28
3.1.3. Használt fejlesztőeszközök . . . . .	29
3.2. Használati esetek . . . . .	29
3.3. Osztályok leírása . . . . .	29
3.3.1. Modell osztályok . . . . .	30
<b>4. Irodalomjegyzék</b>	<b>31</b>

# 1. fejezet

## Bevezetés

Az bizonyos, hogy minden informatikus - beleértve a leendőket is - tanulmányaik kezdetén találkoztak a rendezési algoritmusokkal. Nagyszerű terület arra, hogy megérthessük a műveletigény kérdését, azt hogy mi számít igazán sok adatnak, vagy éppen, hogy mit értünk egy algoritmus stabilitásán.

### 1.1. A feladat és annak értelmezése

### 1.2. Alkalmazott technológiák

A Következőkben röviden összefoglaljuk a *Java*[1], *JavaFX*[2] és *JUnit*[3] jellegzetességeit.

#### 1.2.1. Java

A *Java* egy általános célú, objektumorientált programozási nyelv, melyet 2009-ig a *Sun Microsystems* fejlesztett, ezt követően pedig az *Oracle*. A szakdolgozatban használt 1.8-as verziót már az *Oracle* adta ki 2014-ben. A *Java* nyelv a szintaxisát a *C* és *C++* nyelvektől örökölte, azonban utóbbitól eltérően egyszerű objektummodellel rendelkezik.

A *Java* platformra készült programok túlnyomó többsége asztali alkalmazás. Ma-napság egyre több helyen találkozhatunk a *Java* nyelven írt programokkal, például mobil eszközökön, banki rendszereknél vagy akár egy szórakoztató elektronikai eszközön. Nagy előnye, hogy sok nyelvvel ellentétben platformfüggetlen, azaz egy adott platformról egy program minimális változtatással átültethető egy másik platformra.

A *Java* legfontosabb része a *Java virtuális gép (JVM)*. A *JVM*-et sokféle be-  
rendezés és szoftvercsomag tartalmazza, így a nyelv egyaránt platformként és köz-  
zépszintként is működik. Összefoglalva a *Java* program három fontos szerepet tölt  
be:

- programozási nyelv
- köztes réteg (middleware)
- platform

### 1.2.2. JavaFX

Olyan szoftverplatform, amelynek célja, hogy gazdag internetes alkalmazást lehessen készíteni és futtatni eszközök széles skáláján. Eredetileg a *Swing* könyvtárat váltotta volna fel, azonban jelenleg mindkettő része a *Jave SE*-nek.

A 2.0-ás verzióig a fejlesztők egy külön nyelvet használtak, amelyet *JavaFX Script*-nek neveznek. Azonban mivel ez szintén *Java* bájtkódot generál a későbbiekben megadatott a lehetőség, hogy a programozók *Java* kódot használjanak helyette. A *JavaFX* egyik legnagyobb előnye, hogy egy egyszerű *XML* struktúrában leírhatók a program grafikus felületének összetevői, melyhez ezt követően elegendő az egyes interakciókhoz tartozó funkciókat implementálni.

Az elterjedtebb operációs rendszerek mindegyikét támogatja. Ahogyan előnye, úgy hátránya is a *Swing*-hez képest az, hogy jelenleg is folyik a fejlesztése, ezért olykor csak hosszas utánajárást követően sikerül megoldást találni egy-egy problémára.

### 1.2.3. JUnit

Egy egységteszt keretrendszer a *Java* programozási nyelvhez. Az egységtesztek karbantartására, és futtatására kínál szolgáltatást. Gyakran a verzió kiadási folyamat részeként szokták beépíteni, azaz egy kiadás akkor hibátlan, ha ezen tesztek mindegyike hibátlanul lefut. Egy 2013-as felmérésben[10] tízezer *Java* technológiát használó *GitHub* projektet vizsgáltak. A projektek csaknem harmadánál használták a *JUnit*-ot, ezzel az egyik leggyakrabban használt függvénykönyvtár volt a felmérés során.

## 2. fejezet

# Felhasználói dokumentáció

### 2.1. Rendszerkövetelmények

Az elkövetkezőkben ismertetésre kerülnek a minimális és az ajánlott rendszerkövetelmények.

#### 2.1.1. Minimális rendszerkövetelmények

Mivel a program *Java* nyelven íródott, ezért elengedhetetlen, hogy a felhasználó számítógépén lehetőség legyen *Java* alkalmazások futtatására. Az alábbi operációs rendszereken érhető el a *Java Runtime Enviroment 8u45*-ös verziója:

- *Windows* 8
- *Windows* 7
- *Windows* Vista SP2
- *Windows Server* 2008 R2 SP1 (64-bit)
- *Windows Server* 2012 (64-bit)
- *Mac OS X* 10.8.3 vagy újabb
- *Suse Linux Enterprise Server* 10 SP2+, 11.x
- *Ubuntu Linux* 12.04 vagy újabb
- *Red Hat Enterprise Linux* 5.5+, 6.x
- *Oracle Linux* 5.5+; 6.x; 7.x

Hardverkövetelmények tekintetében a *Java JRE 8* futtatásához szükséges minimum követelményei az irányadóak. Azonban a program bizonyos esetekben több erőforrást is igényelhet, ezért ajánlott nagyobb memóriával és erősebb processzor rendelkező rendszer használata. A követelmények a következő táblázatban találhatók:

<b>Memória</b>	128 MB
<b>Szabad lemezterület</b>	124 (+2) MB
<b>Processzor</b>	<i>Pentium 2</i> 266 MHz

### 2.1.2. Ajánlott rendszerkövetelmények

A szoftver tökéletes működéséhez a legelterjedtebb operációs rendszer, a *Windows* ajánlott. Továbbá követelmény 16:9-es képaránnyal rendelkező monitor, és legalább  $1366 \times 768$  képernyőfelbontás használata.

A rendszer fejlesztése a következő ajánlott konfiguráción történt:

<b>Operációs rendszer</b>	<i>Windows 7</i>
<b>Memória</b>	4 GB
<b>Processzor</b>	<i>Intel</i> Core i5-2467M, 2000 MHz

### 2.1.3. Telepítés és eltávolítás

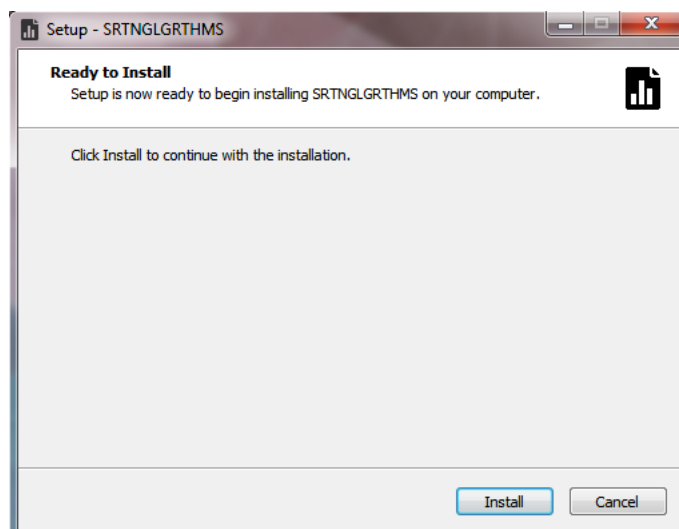
Alapvetően elegendő a *Java* futtatási környezet telepítéséről gondoskodnia a felhasználónak, ezen felül más program telepítésére nincs szükség. Azonban, mivel elsősorban *Windows* operációs rendszeren történő futtatásra lett felkészítve a program, ezt az operációs rendszert használók választhatják a kényelmesebb, natív telepítési megoldást. Elsőként a programhoz készült telepítővel történő konfigurálást vesszük végig, majd ezt követően a *Java* programokra inkább jellemzőbb, ám kissé körülményesebb telepítési mód kerül bemutatásra. Végül röviden összefoglaljuk a program eltávolításához szükséges lépéseket.

Bármely módszert is szándékozik követni a felhasználó, először győződjön meg róla, hogy az előzőekben ismertetett rendszerkövetelményeknek megfelel a számítógépe

#### Telepítés natív telepítővel

Ez a telepítési mód csak a *Windows*-t használók számára érhető el.

Az első lépés a telepítési varázsló elindítása. A megjelenő párbeszédpanelon kattintsunk a **Telepítés** gombra. Ezt követően elindul a telepítés, ami körülbelül fél percet vesz igénybe.



2.1. ábra. A telepítési párbeszédpanel

A telepítési panel bezárása után máris megjelenik a program főmenüje, így megkezdheti a felhasználó a használatát. A későbbiekben történő futtatáshoz a következő könyvtárba szükséges navigálni: **C:\Felhasználók\{Felhasználói név}\AppData\Local\SR**

A fenti útvonal akkor érvényes, ha **C:\** meghajtón található az operációs rendszer, néhány rendszeren más lehet ennek a meghajtónak a betűjele. Továbbá egyes számítógépeken az **AppData** mappa rejtett lehet, így érdemes valamilyen fájlböngészőt, például *Total Commander*-t használni.

### Hagyományos telepítés

A most ismertetésre kerülő telepítési mód minden operációs rendszeren elérhető.

Az első feladat a *Java* virtuális gép telepítése, amely letölthető a következő webcímről: <http://java.com/inc/BrowserRedirect1.jsp>

Fogadjuk el a licencszerződést, töltsük le a *JRE* telepítőfájlt. Ha frissíteni szeretnénk a jelenlegi *Java* verziót a rendszerünkön, akkor előbb célszerűbb eltávolítani a régi verziót. Miután feltelepítettük a futtatási környezetet, készen áll a szoftver futtatására a rendszerünk. A program könyvtárában található **SRTNGLGRTHMS.jar** fájl elindításával kezdetjük meg a szoftver használatát.

### Eltávolítás

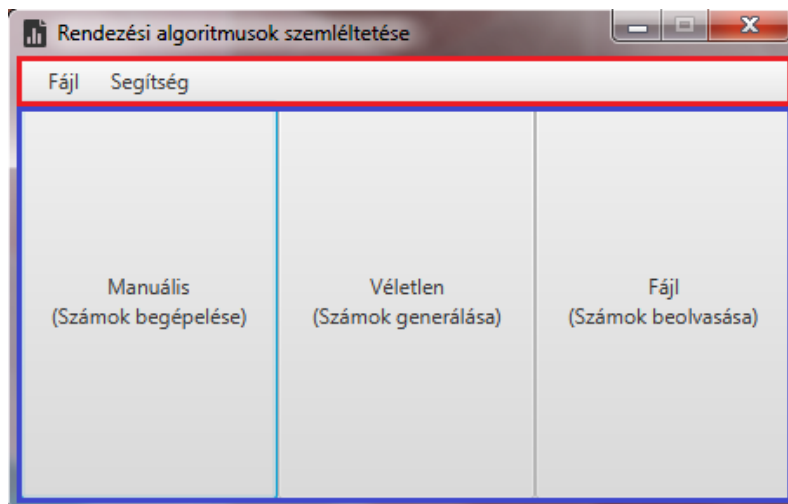
A program törlése az utóbb ismertetett telepítési mód esetében mindössze annyiból áll, hogy a **SRTNGLGRTHMS.jar** fájlt eltávolítjuk, valamint amennyiben a jövőben nincs igény a *Java* futtatási környezet használatára, úgy a felhasználó eltávolíthatja azt. *Windows*-on történő natív telepítést követően a program telepítési könyvtárában található **unins000.exe** fájlt futtatva, majd az **Igen** gombra kattintva a program törlődik a számítógépről.



## 2.2. Felhasználói felület bemutatása

### 2.2.1. Főmenü

A program indítását követően megjelenik a főmenü, mely két komponensből áll.

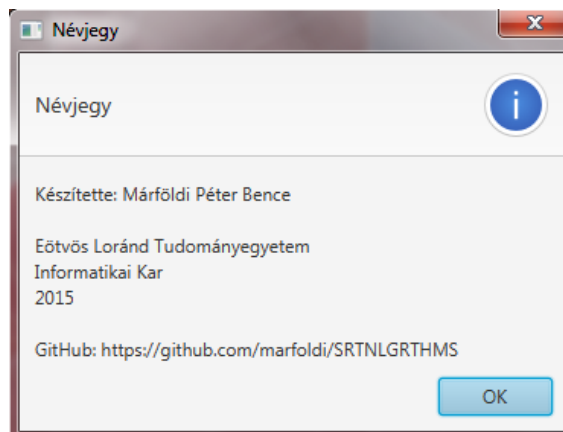


2.2. ábra. A program főmenüje

Az ablak felső részén található piros színnel jelölt rész az eszköztárat foglalja magában. A kék szín jelöli a főmenü központi paneljét, mely három gombból tevődik össze. Ezen gombokra történő kattintás után lehetőség nyílik a rendezendő számok megadására.

### Eszköztár

Az eszköztárat két menüpont alkotja, **Fájl** és **Segítség** címszóval ellátva. Az előzőben a program bezárásának lehetősége kapott helyet, míg utóbbiban a szoftver névjegye tekinthető meg. Az **Ok** gomb lenyomásával bezárható a névjegy.



2.3. ábra. A program névjegye

### Központi panel

A három gombból áll:

- Manuális (számok begépelése)
- Generálás (számok generálása)
- Fájl (számok beolvasása)

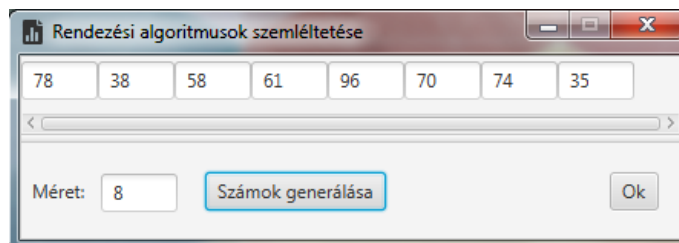
Ezek közül bármelyre kattintva átnavigálhatunk a bement megadását lehetővé tévő felületekre.

### 2.2.2. Bemenet megadása panel

Az előzőekben említett három lehetőség közül választhat a felhasználó.

#### Manuális

A megjelenő panelen két gomb található, melyek kezdetben inaktívak. A "Méret:" címke után található beviteli mezőbe egy pozitív egész szám megadásával és az **ENTER** billentyű leütésével megjelennek a számok bevitelére lehetőséget adó mezők. Ezt követően a panelen található két gomb már kattintható, a **Számok generálása** gombra kattintva 0 és 100 közötti véletlen számokkal töltődnek fel a mezők. Az **Ok**ra történő kattintás után megjelenik a program **Főpanelje**.

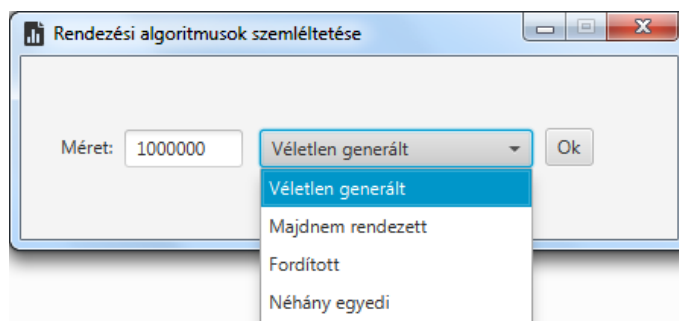


2.4. ábra. 8 elemű manuálisan megadott bemenet

Meg kell jegyezni, hogy ebben a módban legfeljebb száz érték adható meg, ennél nagyobb méretű bemenet manuális feltöltése túl körülményes lenne. Amennyiben több számot szándékozik megadni a felhasználó, válasszon a számok generálása vagy a fájlból történő beolvasás lehetőségek közül.

#### Generálás

A "Méret:" címke mellett megadva a bemeneti számok mennyiségét, és a legördülő menüből kiválasztva a generálás módját az **OK** gomb kattinthatóvá válik.



2.5. ábra. Egymillió véletlen generált érték megadása

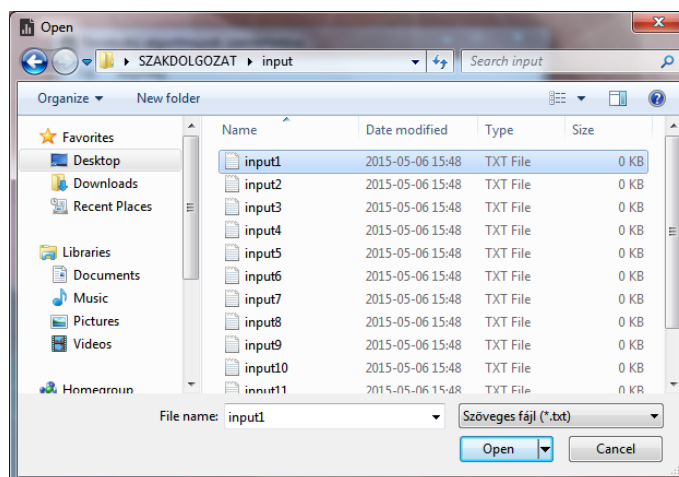
A legördülő menüből a következő négy típus közül lehet választani, azaz a generált tömb legyen:

- Véletlen generált - véletlenszerűen választott számokból álljon
- Majdnem rendezett - a tömb 80%-a már rendezve legyen
- Fordított - a tömb legyen csökkenőleg rendezett
- Néhány egyedi - a tömb elemei között sok azonos érték szerepeljen

Itt megjegyzendő, hogy amennyiben az input mérete az  $[1,100]$  intervallumban van, akkor a rendezendő számok a 0 és 100 közötti értékek közül kerülnek kiválasztásra. Ennek oka, hogy a túl nagy differencia az egyes értékek között sokat rontana az oszlopdiagramok megjelenésén. Ellenkező esetben a *Java* nyelv által definiált egész típus(*Integer*) maximális értékéig terjedhet a generált számok nagysága.

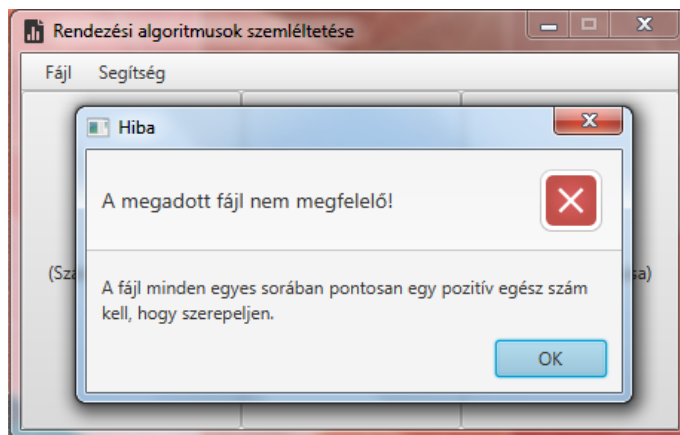
### Fájl - fájlból beolvasás

A gombra történő kattintás után megjelenik egy fájlállító. A tallózóban csak szöveges(*txt* kiterjesztésű) fájl választására van lehetőség.



2.6. ábra. Fájlállító

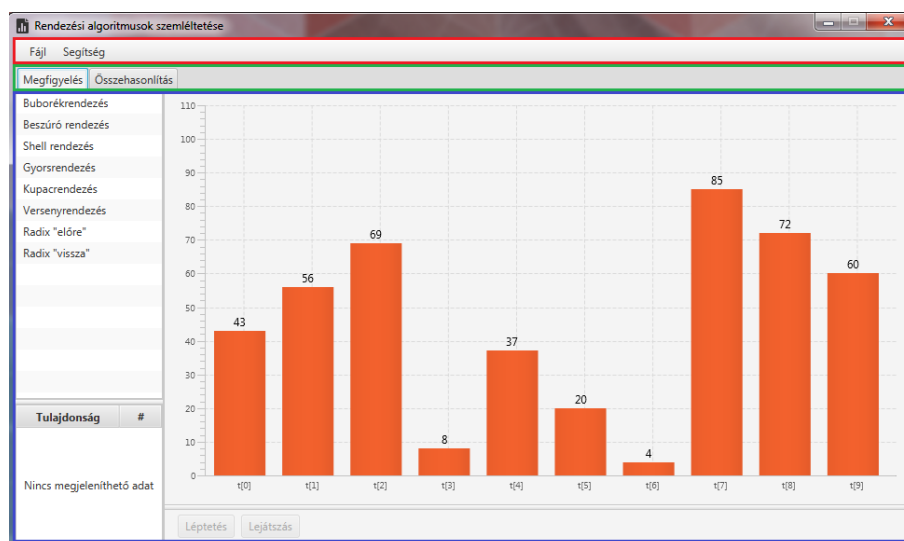
A fájl tartalmára történő megszorítások szigorúak. Minden sora legfeljebb egy pozitív egész számot tartalmazhat, ellenkező esetben a program hibaüzenet kíséretében visszatér a főmenübe. Fontos, hogy az előzőek értelmében az üres sorok sem megengedettek.



2.7. ábra. Nem megfelelő fájl esetén a hibaüzenet

### 2.2.3. Főpanel

A rendezendő számok sikeres megadása után megjelenik a program főpanelje, mely három logikai részből áll. A főpanel magában foglalja a **Megfigyelés** és **Összehasonlítás** paneleket, melyek összetettségükből fakadóan külön alfejezetekben kerülnek részletezésre.



2.8. ábra. A program főpanelja

## Eszköztár

A **Főmenü**höz hasonlóan itt is jelen van a piros színnel jelzett eszköztár sáv. Itt fontos kiemelni, hogy a **Fájl** és **Segítség** pontokon belül további alpontok is elérhetők:

A **Fájl** menüpont bővül a **Vissza a főmenübe** lehetőséggel, melynek segítségével a program újraindítása nélkül lehetőség van újabb rendezendő számsorozat megadására.

A **Segítség**re kattintva további lehetőségként választható az **Algoritmusról** pont. Itt elolvasható a rövid szöveges ismertetője a **Megfigyelés** panel listájából kiválasztott elemnek. Amennyiben pedig az **Összehasonlítás** panel aktív, akkor e panel táblázatából kiválasztott algoritmus leírása tekinthető meg. Az aktuális panelen ha nem került kiválasztásra sor, akkor a menüpontra történő kattintás után felugró ablak figyelmezteti a felhasználót arról, hogy e menüpont használatához előbb ki kell választani egy algoritmust.

## Panelválasztó

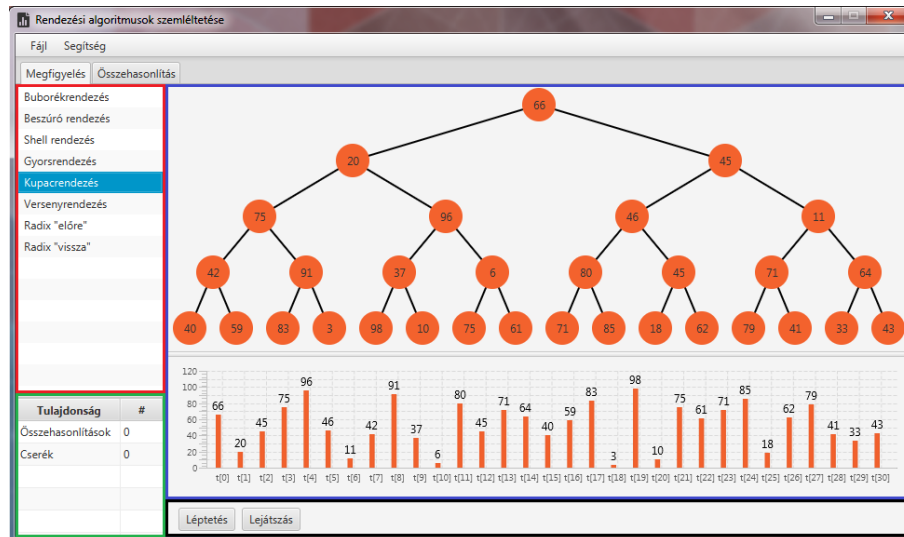
Amennyiben a bemenet megadásánál már ismertetett  $[1,100]$  intervallum magában foglalja a bemenet hosszát, akkor két elem látható a zöld jelölt részen. Így lehetősége van a felhasználónak navigálni a **Megfigyelés** és az **Összehasonlítás** panel között. Ellenkező esetben csak az utóbbi panel jelenik meg. Az aktuálisan megjelenített felület neve a listában kék kerettel jelenik meg, valamint a másik elemhez képest világosabb szürke színnel.

## Panel

A harmadik logikai egységet alkotják a panelek - kék színnel jelölve -, melyek közötti váltást a **Panelválasztó** teszi lehetővé. Mivel részletesebb leírást kíván a panelek ismertetése, ezért egy-egy külön alfejezetben kerülnek bemutatásra.

### 2.2.4. Megfigyelés panel

A főpanel középső területén foglal helyet, négy komponensből tevődik össze. A felhasználónak itt nyílik lehetősége az egyes algoritmusok megfigyelésére, tanulmányozására.



2.9. ábra. Megfigyelés panel

## Algoritmus lista

Az ábrán piros színnel jelölt rész, melynek elemeire kattintva kiválasztható, hogy mely algoritmust szeretné a felhasználó vizsgálni. A kiválasztott elem kék háttérszínt kap, alapértelmezett esetben nincs kijelölt elem.

## Állapotjelző táblázat

A kiválasztott algoritmus aktuális állapotához tartozó információk jelennek meg a táblázatban. Az ábrán zöld kerettel van jelölve a komponens. Amennyiben nincs kiválasztott elem a **Nincs megjeleníthető adat** szöveg jelenik meg.

Két oszlopa a **Tulajdonság** és a **#**, mely utóbbi az értéket jelöli. Alapvetően az összehasonlításon alapuló algoritmusoknál megjelenő adatok az összehasonlítások és cserék vagy mozgások száma. Az edényrendezéseknél pedig az aktuálisan vizsgált bit indexe és a vizsgálatok száma jelenik meg. Egyes algoritmusokhoz a jobb megérthetőség miatt további állapotjelző értékek is tartoznak, melyek a következők:

Algoritmus	Tulajdonság
Shell rendezés	Lépésköz
Gyorsrendezés	Vezérelem
Radix "előre"	Cserék száma

## Gombok

Alaphelyzetben a **Léptetés** és **Lejátszás** gombok inaktívak. A képen fekete keretet jelzi a helyüket. Amennyiben a felhasználó kiválaszt egy elemet az algoritmus listáról kattinthatóvá válnak. A léptetéssel egy következő állapotot tekinthet meg

a felhasználó. A **Lejátszás** gombra kattintva a program bemutatja az algoritmus működését. A felhasználó bármikor megállíthatja az animációt a **Lejátszás** gomb helyén található **Megállítás** gombra kattintva, majd ha kívánja innen folytathatja a vizsgálatot. Amennyiben a rendezés lezajlott, megjelenik az **Újraindítás** gomb, melynek megnyomásával az értékek visszakerülnek az eredeti helyükre.

### Állapotjelző felület

A fenti ábrán kék szín jelöli ezt a területet. Az állapotjelző felületen minden esetben legalább egy oszlopdiagram foglal helyet. Ettől eltérően a **Kupac**- és **Versenyrendezés** kiválasztásakor egy gráf is helyet kap. Továbbá a **Radix "vissza"** rendezés második tömbjének reprezentálásához egy plusz oszlopdiagram is megjelenik. Itt megjegyzendő, hogy a szemléltetéshez használt bináris fa legfeljebb 31 csúcsot tartalmazhat. Hosszabb bemenet megadásakor csak az első 31 elem vesz részt a megjelenítésben.

Az egyes műveleteket különböző színek is jelölik. Alapértelmezetten a rendezendő elemek színe a következőket jelentik:

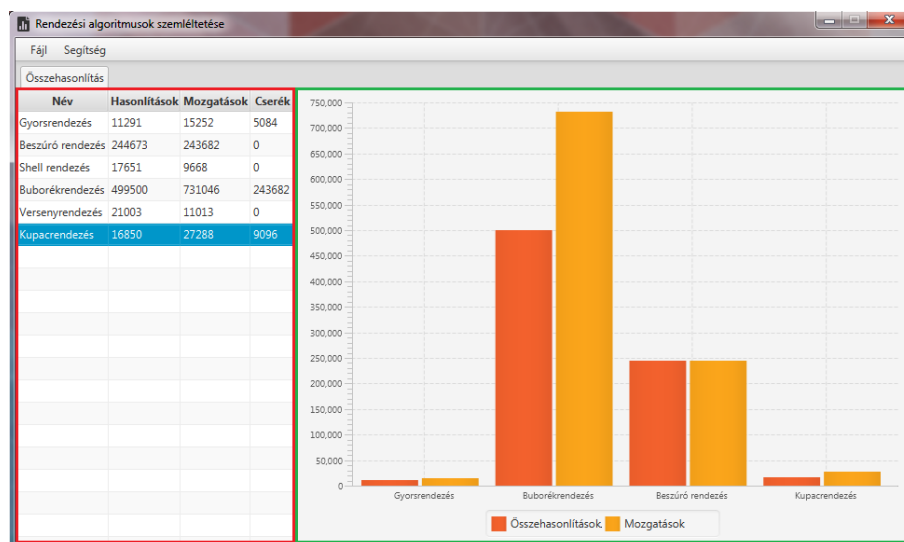
- Az alapértelmezett szín, nincs kijelölve az elem.
- Az elem cserére vagy mozgatásra van kijelölve.
- Az elemnek kitüntetett szerepe van.
- Az elem már a végleges helyére került.

A ● jelölés némi magyarázatra szorul. A kitüntetett szerepű elemnek számít például a gyorsrendezés vezéreleme, vagy a versenyrendezés fájának felépítésekor egy belső csúcsba kerülő elem.

Némely algoritmusnál a fentiekől eltérő lehet az egyes színek jelentése. A következő fejezetben(2.3), az algoritmusok ismertetésénél jelölve van minden ilyesfajta különbség.

### 2.2.5. Összehasonlítás panel

A **Megfigyelés** pannellel megegyezően a főpanel középső részén található, két logikai egységből épül fel. Lehetőséged ad az összehasonlításra alapuló rendezések műveletigényeinek a vizsgálására.



2.10. ábra. Összehasonlítás panel

### Elemzés táblázat

Piros kerettel jelölt rész az ábrán, mely táblázatnak négy oszlopa van:

- Név - az algoritmus neve
- Hasonlítások - a rendezés során végzett összehasonlítások szummája
- Mozgatások - az elemmozgatások számának összege
- Cserék - a végzett cserék szummája

A **Mozgatások** oszlop minden esetben kitöltésre kerül, még ha az algoritmus nem is mozgatásokat használ. Ekkor a mozgatások oszlopban a cserék számának háromszorosa jelenik meg. Ennél fogva egyszerűbb az algoritmusok vizsgálata.

### Elemzés diagram

A táblázathoz képest balra helyezkedik el a zöld kerettel jelölt oszlopdiagram. Kezdetben teljesen üres, csak a jelölések jelentése látható. Az **Elemzés táblázat** egy során történő dupla kattintás következtében megjelenik az algoritmus összehasonlításainak és mozgatásainak a száma. Előbbi narancs- utóbbi citromsárga színnel. Amennyiben olyan sorra kattint a felhasználó, mely már látható a diagramon, azon algoritmus adatai eltűnnek a felületről.



## 2.3. A vizsgált algoritmusok

### 2.3.1. Buborékredezés

#### Leírás

A legrégebbi és a legegyszerűbb rendezési algoritmus. Mindemellett a legtöbb esetben a leglassabb is. Már az 1965-ös évben megjelent egy teljes körű elemzése[4].

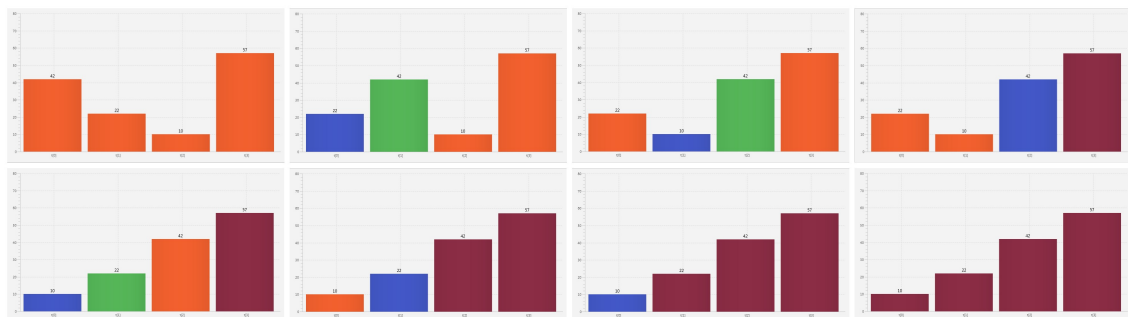
#### Működés

A rendezés minden egyes elemet összehasonlít a rákövetkező elemmel, és ha szükséges megcseréli őket. Ez azt eredményezi, hogy lépésenként a maximális elem "buborék" szerűen a lista végére kerül, ezzel egyidejűleg a kisebb elemek "lesüllyednek" a tömb elejére. Amennyiben egy menetben a maximális eleme elérte a helyét visszavezetjük a problémát az eggyel "rövidebb" rendezési feladatra[5]. Az algoritmus javítható azzal, ha figyeljük, hogy az egyes menetekben történt-e csere. Amennyiben egy olyan menet végére értünk, amelyben egy elem sem cserélt helyet, akkor a tömb már rendezve van. A program az eredeti, nem javított verziót mutatja be.

#### Példa

A rendezendő számok: 42, 22, 10, 57.

Az első ábrán szerepel a kezdeti állapot, a másodikon az első két értéket cseréje látható. A 42 ismét fentebb kerül egy pozícióval a harmadik ábrán. A negyedik ábrán csupán egy összehasonlítás történik, mivel az 57 nagyobb mint az öt megelőző érték. Az ötödik ábrán az előző állapot első két értékének(22 és 10) a cseréje látható. Az ezt követő összehasonlítás eredményeképp tudható, hogy a 42 érték a helyére került. Ezt követően már csak egy összehasonlítást történik, mely után az adatsor rendezve lesz.



2.11. ábra. Példa a buborékredezésre

## Műveletigény

Párosával haladunk végig az elemeken, és így vizsgáljuk őket. Mivel minden menet végén visszavezetjük a problémát az eggyel rövidebb feladatra, ezért az összehasonlítások száma  $n$  hosszú bemenetre:

$$\ddot{O}(n) = (n-1) + (n-2) + \dots + 1 = \frac{n \cdot (n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$$

A cserék száma már nem állandó, a bemenő adatok inverziószámával egyezik meg. Belátható, hogy minden egyes cserével egy inverzió szüntethető meg két elem között. A legtöbb cserét akkor szükséges eszközölni, ha a rendezendő elemek mindegyike inverzióban áll a rákövetkező elemmel, azaz a tömb nagyság szerint csökkenő sorrendben rendezett. Ekkor a cserék száma:

$$MCs(n) = \frac{n \cdot (n-1)}{2} = \Theta(n^2)$$

Amennyiben a tömb elemei már rendezettek, akkor egyetlen cserét sem szükséges végrehajtani. Az átlagos csereszám a maximális cserék számának fele[5], ám nagyságrendileg még ez is  $\Theta(n^2)$

## Jelölések az állapotjelző felületen

Kitüntetettnek (● színnel) jelöljük azt az elemet, amely egy csere következtében feljebb került. Nincs egyéb eltérés az eredeti jelölésekhez képest.

### 2.3.2. Beszúró rendezés

#### Leírás

A legtöbb esetben akár kétszer hatékonyabb a buborékrendezéshez képest[5]. Az elve egyszerűen megérthető egy hétköznapi példán keresztül: A kezünkben tartunk kártyalapokat, majd egy pakliból húzva az új lapot beszúrjuk a kezünkben lévő, már rendezett lapok közé. Ennélfogva szokás kártyás rendezésnek is nevezni.

#### Működés

Két részre bontjuk a tömböt, az egyik részén - a tömb bal szélé - a már rendezett, míg a másikon a még nem vizsgált elemek szerepelnek. Kezdetben a tömb első elemét tekintjük a rendezett résznek.

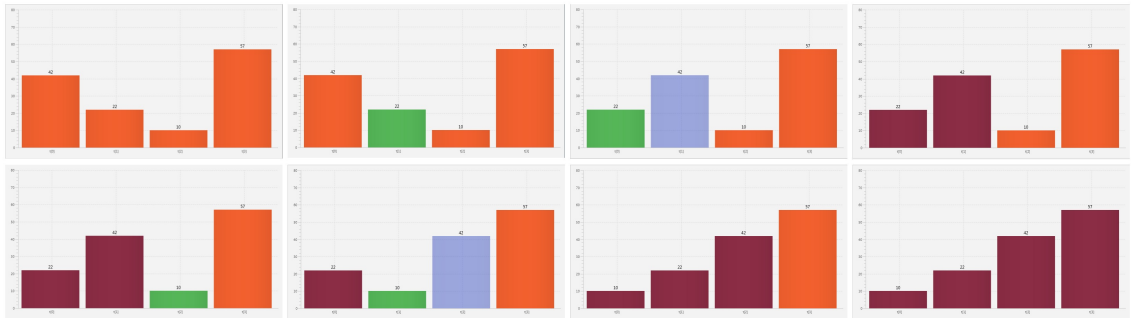
Lényege, hogy a soron következő elemet - a második elemtől kezdve - egy ideiglenes változóba mentjük, és a rendezett tömb rész elemeit jobbra csúsztatjuk mind-

addig, amíg a kiválasztott érték nem kerül a helyére. Ezt  $n-1$  alkalommal ismételve megkapjuk a rendezett tömböt.

### Példa

A rendezendő számok megegyeznek a buborékrendezés bemutatásánál használtakkal, azaz: 42, 22, 10, 57.

Az első ábrán szerepel a kezdeti állapot. A következő képen jelöljük azt, hogy a második elemet fogjuk beszúrni ezért elmentésre kerül. A harmadik ábrán felcsúsztatjuk a 42 értéket, majd a negyedik képen már látható, hogy a tömb első két eleme alkotja rendezett tömbrészt. Ezt követően a harmadik elem kerül kiválasztásra, a teljes rendezett tömbrészt jobbra tolódik, így az 7. képen már a tömb első három eleme rendezett. Végül a tömb utolsó eleme kerül összehasonlításra az öt megelőzővel, mivel nagyobb tőle, így kész a rendezési feladat.



2.12. ábra. Példa a beszúró rendezésre

### Műveletigény

Legjobb esetben elegendő a második elemtől mindegyik elemet az előtte található értékkel összehasonlítani, vagyis a tömb már eleve rendezett. Ez pontosan  $n-1$  összehasonlítást jelent, azaz:

$$m\ddot{O}(n) = (n - 1) = \Theta(n)$$

Ebben az esetben pedig egyetlen elemet sem kell mozgatni.




A legrosszabb eset akkor áll fenn, ha a beszúrandó elem minden alkalommal kisebb a már beszúrt elemeknél, azaz a tömb elemei csökkenőleg rendezettek. Ekkor az összehasonlítások száma:

$$M\ddot{O}(n) = \sum_{i=1}^n i = \frac{(1 + (n - 1)) \cdot (n - 1)}{2} = \frac{n \cdot (n - 1)}{2} = \Theta(n^2)$$

Továbbá ekkor a mozgatások száma:

$$MM(n) = \Theta(n^2)$$

### Jelölések az állapotjelző felületen

A már rendezett tömbrészt  szín jelöli. Az aktuálisan beszúrandó elem pedig a  jelölést kapja. A kiválasztott elemmel összehasonlított értékek  színt kapnak.

Megjegyzendő, hogy a kitüntetett szerepű avagy beszúrandó elem nem szerepel tömbelemként az összehasonlításakor, csupán a felületen van jelölve a jobb megérthetőség okán.

### 2.3.3. Shell rendezés

#### Leírás

**Donald Shell** nevéhez fűződik, a legtöbb esetben a leggyorsabb négyzetes idejű algoritmus. Az elve az, hogy célszerű lehet előbb a "távolabb" lévő elemeket hasonlítani és mozgatni, mivel így az elemek hamarabb közel kerülhetnek a végleges helyükhöz.

#### Működés

Többször vizsgálja a tömböt, és minden alkalommal egy részén beszúró rendezést hajt végre. Arra, hogy mekkora méretű résztömböt vizsgáljon az egyes lépésekben az algoritmus több javaslat is található. A teljesség igénye nélkül néhány ajánlás[7] erre vonatkozóan:

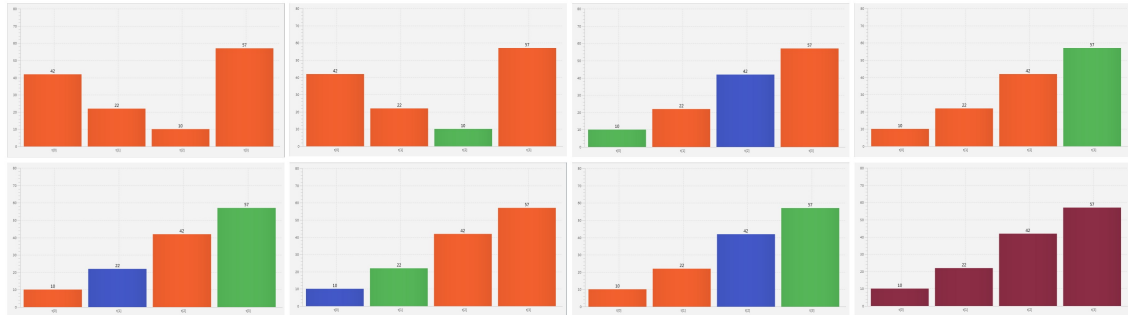
Szabály (k=1...)	Konkrét értékek	Műveletigény	Szerző
$\lfloor n/2^k \rfloor$	$\lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{4} \rfloor, \dots, 1$	$\Theta(n^2)$	Shell, 1959
$2^k - 1$	1, 3, 7, 11, ...	$\Theta(n^{\frac{3}{2}})$	Hibbard, 1963
$2^p 3^q$ váltakozva	1, 2, 3, 4, ...	$\Theta(n \log^2 n)$	Pratt, 1971

#### Példa

Az előző példákban használt számokat kívánjuk rendezni ismét, azaz a bemenet: 42, 22, 10, 57.

Az első képen látható a kezdeti állapot. A következő ábrán jelöljük azt, hogy a harmadik elemet fogjuk mozgatni ezért elmentésre kerül, ekkor a lépésköz kettő.

A harmadik ábrán látható az első elem átmozgatása. A negyedik képen vesszük a következő elemet, továbbra is 2 lépésközzel megvizsgáljuk a második indexen lévő elemet az ötödik ábrán. Itt nem történik mozgatás. Végül a hatodik megjelenített lépéstől kezdve beszűrő rendezést alkalmazunk.



2.13. ábra. Példa a Shell rendezésre

### Műveletigény

A fentebbi táblázatból látható, hogy az algoritmus sebessége nagyban függ a lépésköz megválasztásától. A program a legrégebbi, **Donald Shell** által javasolt értékeket[8] használja, így legrosszabb esetben  $\Theta(n^2)$  a műveletek száma.

A legjobb eset akkor áll fenn, ha a tömb elemei már rendezettek, ekkor nincs szükség mozgatásra, az algoritmus futási ideje  $\Theta(n)$  nagyságrendű.

### Jelölések az állapotjelző felületen

A felület bemutatásában szereplőkhöz képest nincs eltérés.

## 2.3.4. Gyorsrendezés

### Leírás

**C.A.R. Hoare**[9] alkotta meg 1965-ben. Az egyik leggyorsabb rendezési eljárás, ezért rendkívül gyakran alkalmazzák.

### Működés

Helyben rendező, oszd meg és uralkodj[6] elven működő rekurzív algoritmus. A következő négy lépésre bontható fel az rendezés:

- Ha csak egy vagy nulla elemű az elemzett rész, akkor ne tegyünk semmit.
- Válasszunk egy vezérelemet (legjobb oldalibb elem).
- Osszuk két részre a rendezendő részt, az egyik oldalára a vezérelemtől kisebb, míg a másikra a nagyobb elemek kerüljenek.

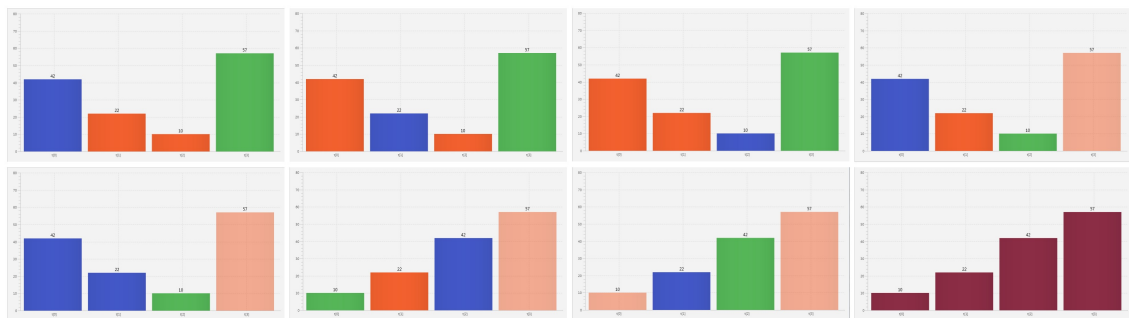
- Rekurzívan ismétljük meg az előbbi lépéseket a résztömbökön.

Gyakorlatilag szétválasztást eszközölünk, melynek eredményeképp a kiválasztott elemtől kisebb értékek tőle balra, a nagyobbak pedig jobbra helyezkednek el. Az egyszerűbb megérthetőséget szem előtt tartva a program mindig a legjobboldalibb elemet szelektálja.

### Példa

Az eddigi példákkal megegyezően a bemenet: 42, 22, 10, 57.

Az első képen látható a már kiválasztott vezérelem. A negyedik ábráig csak összehasonlítások történnek, mivel minden elem kisebb, mint a kiválasztott érték. A negyedik képen a három elemű résztömbön végezzük el a rendezést, a vezérelem ismét a legjobboldalibb érték. Az ötödik képen az első érték nagyobb mint a vezérelem, így ezt az elemet a hatodik képen megcseréljük a vezérelemmel. Ezt követően a középső két elem kerül összehasonlításra, mivel nincs szükség cserére így a rendezés befejeződött.



2.14. ábra. Példa a gyorsrendezésre

### Műveletigény

A rendezés műveletigényét befolyásolja, hogy hogyan választjuk meg a vezérelemet. Például a legnagyobb műveletigényt ( $\Theta(n^2)$ ) eredményezi, ha mindig a legjobboldalibb elemet választjuk vezérelemnek, és a tömb elemei csökkenő sorrendben vannak[6]. Éppen ezért a gyakorlatban javasolt ezen elem véletlenszerű megválasztása.

Ha feltételezzük azt, hogy minden rekurzív lépés felezi a tömböt, akkor egy  $n \log(n)$  magasságú fával lehet ábrázolni a rekurziót. Továbbá, mivel minden szinten az elemek száma  $n$ , és ezek particionálásához használt lépésszám  $\Theta(n)$  így a legjobb esetben a futási idő  $\Theta(n \cdot \log(n))$ .

A gyorsrendezés a legtöbb esetben(közepes és nagy méretű bemenetre) a legmegfelelőbb választás ha számít a rendezés sebessége, mivel az átlagos futási ideje -  $\mathcal{O}(n \cdot \log(n))$  - közel áll a legjobb futási időhöz[6].

Amennyiben a tömb elemei már eleve rendezettek vagy esetleg fordított sorrendben szerepelnek sajnos nem túl hatékony az eljárás.

### Jelölések az állapotjelző felületen

A vezérelemet ● szín jelöli. Annak érdekében, hogy jobban átlátható legyen, hogy éppen melyik résztömbön folyik a vizsgálat az éppen nem vizsgált rész haloványabb sárga színnel van jelölve. A többi jelölés a felület bemutatásában leírtaknak megfelelően történik.

### 2.3.5. Kupacrendezés

#### Leírás

Az  $\mathcal{O}(n \log(n))$  algoritmusok közül az egyik leglassabb, azonban előnye a gyorsrendezéssel szemben, hogy nem erőteljesen rekurzív. Ennél fogva jól alkalmazható milliós nagyságrendű bemenetre. Ahogyan a neve is sugallja, a rendezéshez egy kupac adatszerkezetet használ. Az algoritmus ismertetése előtt definiáljuk a kupac fogalmát[5]: Olyan bináris fa, amelyre a következők teljesülnek:

- Kizárólag a levelek szintjén hiányozhat csúcs, azaz "majdnem teljes".
- A levélszint csúcsai balra tömörítettek.
- Minden belső csúcs értéke nagyobb vagy egyenlő, mint a gyerekeinek értékei.

A második pont értelmében egyetlen olyan csúcs lehet, amelynek csak egy gyereke van, és az közvetlenül a levélszint felett kell, hogy elhelyezkedjen.

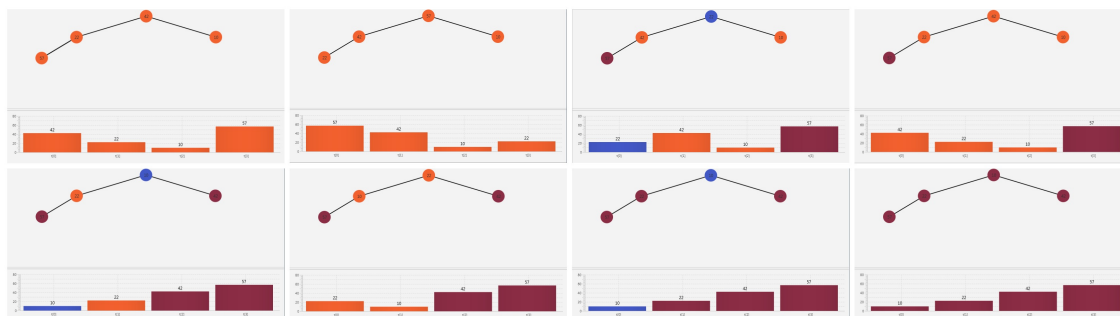
#### Működés

A rendezés az előbbi tulajdonságokra támaszkodik. Az algoritmus a bemeneti elemekből kupacot épít, majd a legfelsőbb elemét áthelyezi a kupac "végére". Ezt követően ellenőrzi a kupac tulajdonságokat, ahol szükséges cseréket hajt végre, hogy helyreálljon a kupac adatszerkezet, ekkor már az utolsó, legjobboldalibb levélelem nem vesz részt a kupacépítésben. A gyökérben található elemet a legjobboldalibb levélelem elé helyezi, és újraépíti a kupacot. Ezen lépések addig ismétlődnek, amíg már a maximális elem áthelyezése nem lehetséges, a kupac "végére" helyezési művelet elérte a gyökeret.

#### Példa

Az eddigi példákkal azonosan a rendezendő számsorozat: 42, 22, 10, 57.

Az első kép a kezdeti állapotot mutatja. Ezt követően a bemeneti adatokból kupacot építünk, melynek eredménye a második ábra. A gyökérelemet a levélszint utolsó elemével megcseréljük a harmadik képen. Ellenőrizzük, hogy teljesülnek-e a kupac tulajdonságok, amennyiben nem cserét hajtunk végre, ennek eredménye látható a 4. képen. Ismét a gyökérelemet lesüllyesztjük, majd ellenőrizzük a kupac tulajdonságot. Az előző két lépést megismételve rendezett tömböt kapunk.



2.15. ábra. Példa a kupacrendezésre

## Műveletigény

A kupacrendezés egyik további, hogy míg a gyorsrendezésnek legrosszabb esetben a futási ideje  $\mathcal{O}(n^2)$ , addig itt a futási idő továbbra is  $\mathcal{O}(n \cdot \log(n))$ . Ennél fogva azon rendszereknél, ahol a négyzetes futási idő elfogadhatatlan inkább kupacrendezést alkalmaznak.

## Jelölések az állapotjelző felületen

Az éppen összehasonlított értékeket ● jelöli. Amennyiben csere történt a belső csúcsba kerülő érték háttérszíne ● lesz. Ez alól kivétel, ha a gyökérbe került új érték, mivel ekkor már tudjuk az új legnagyobb értéket, ennek a színe ● lesz. Továbbá akkor is ezt a színt használjuk, ha már a tényleges, végleges helyére került egy elem.

### 2.3.6. Versenyrendezés

#### Leírás

A maximum-kiválasztó rendezések közé tartozik, minden egyes menetben kiválasztja a legnagyobb elemet, kiírja és végül eltávolítja. A maximum kiválasztásnak a gyakorlati háttérét a sportesemények lebonyolítási rendje adja, azaz meghatározza az elemek között a "nyertest"[5]. A módszert  $n=2^k$  inputhossz esetén érdemes alkalmazni, mivel ettől értéző bemenetre sokkal kedvezőbb eredményt lehet elérni a kupacrendezéssel[5].



## Működés

Az algoritmus által használt adatszerkezet egy teljes bináris fa. A bináris fa leveleiben szerepelnek a rendezendő elemek. Az első speciális menetben a fa belső pontjait kitöltjük, úgy, hogy a pontba a gyerekei közül nagyobb érték kerül.

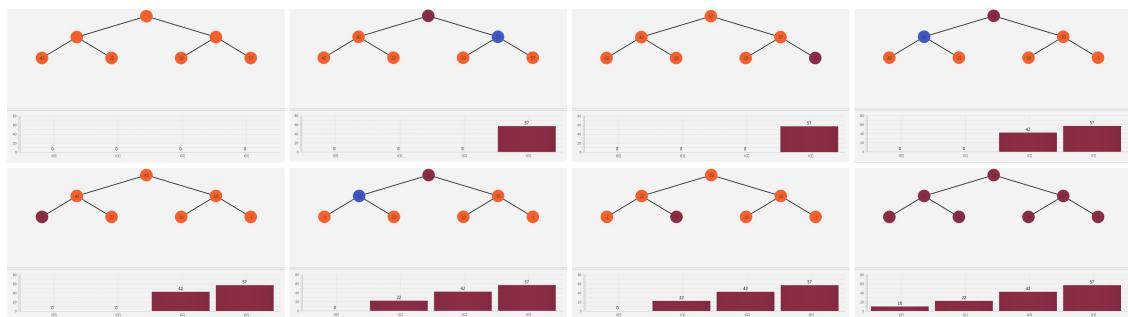
Ezt követően kerül sor az  $(n - 1)$  egyszerűbb menetre: A gyökérben található elemet keresve "lefelé" haladunk a bináris fában, majd megtalálva azt a levelet amelyben a gyökér értéke szerepel egy abszolút vesztest állítunk a helyére. Ez az érték a programban -1, mivel csak pozitív egészeket használunk a rendezések szemléltetésére. Ezzel ellentétben a gyakorlatban ez az érték  $-\infty$ . Majd ezen az "ágon" újrajátsszuk a mérkőzéseket.

Amennyiben a bemenet hossza nem kettő hatvány a program -1 értékekkel tölti fel a bináris fa további leveleit, amíg a bemenet hossza nem lesz megfelelő.

## Példa

Az eddigi példákkal megegyezően a bemenet: 42, 22, 10, 57.

Az első ábra a kezdőállapotot mutatja. A második kép a versenyfa kitöltését követő állapot, melynek eredményeképp megjelenik az abszolút maximum érték a diagramon. Ezt követően a harmadik képen látható a maximum értékhez tartozó levélelem megtalálásának állapota. A negyedik ábra szemlélteti az újrajátszás eredményét. Az előzőekhez hasonlóan szemléltetik a további képek a maximum érték megjelenítését, a levélelem megkeresését majd az újrajátszás eredményét.



2.16. ábra. Példa a versenyrendezésre

## Műveletigény




A rendezés egyetlen hátránya a tárigénye,  $n$  szám esetén további  $n - 1$  mezőre szükség van a versenyfa elkészítéséhez. Éppen emiatt a gyakorlatban nem sűrűn használt eljárás. Az első, speciális menet, a versenyfa kitöltése  $n - 1$  összehasonlítást és mozgatót használ ( $n - 1$  a belső csúcsok száma). Minden további menetben a fán kétszer kell végigmenni, melynek magassága  $\log_2(n)$ . Egyszer a maximális levél

megtalálásához, majd az újrajátszáshoz, így a ezen menetek  $2 \log_2(n)$  összehasonlítást végeznek. Mozgatás csak a második, újrajátszási művelethez tartozik. Így a műveletigények:

$$\ddot{O}(n) = n - 1 + (n - 1) \cdot 2 \cdot \log_2(n) = \Theta(n \cdot (\log(n)))$$

$$M(n) = n - 1 + (n - 1) \cdot \log_2(n) = \Theta(n \cdot (\log(n)))$$

### Jelölések az állapotjelző felületen

Az összehasonlításokat, valamint a maximális levélelem megkereséséhez bejárat utat  szín jelöli. A meccsek lejátszásakor a belső csúcsba kerülő érték  jelölést kap. Ez alól kivétel, amikor a gyökérbe kerül egy meccs győztese, ekkor  lesz az elem színe, továbbá akkor is ez a jelölés, amikor megtaláltuk a maximális levélcsúcsot.

### 2.3.7. Radix "előre"

#### Leírás

Az előzőekben ismertetett algoritmusok mindegyike összehasonlításra alapuló rendezés. A radix rendezés viszont az edényrendezések közé tartozik. Ezen rendezések nem hasonlítják össze az elemeket, hanem az elemek az értéküknek megfelelő edényekbe kerülnek. Az edényrendezések eredményeként rendezett adatsorozatot kapunk lineáris időben. A radix rendezés egy rekurzív algoritmus, melynek minden szintjén létrejönnek az edények.

Az általános edényrendezés egy speciális változata a radix előre rendezés, bináris,  $d$  hosszú számokra.

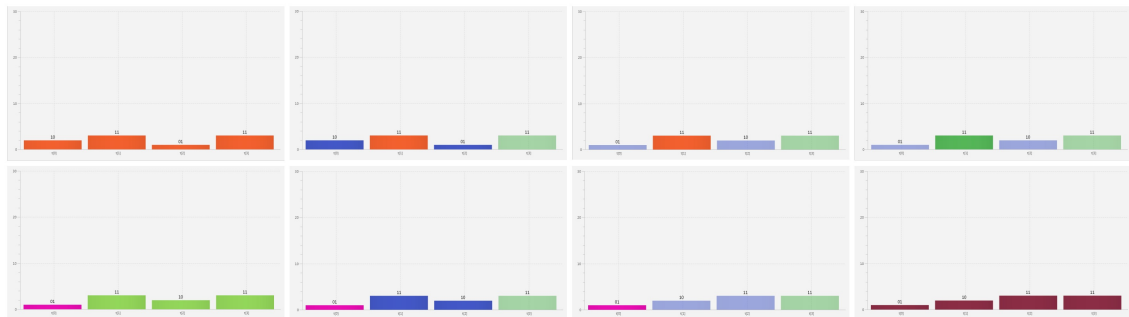
#### Működés

Az első menetben a rendezendő elemek első bitjét vizsgálja az algoritmus. A vizsgálat két mutatóval történik, melyek a tömb két végéről indulnak. A tömb elején addig halad a mutató, amíg a vizsgált elem első jegye nem 1, ezzel párhuzamosan a tömb végén olyan elemet keres a másik, melynek első jegye 0. Amennyiben talált ilyen elemeket megcseréli őket. Ezt mindaddig folytatódik, amíg a két mutató nem találkozik. Ekkor kialakul két edény, az elsőben a 0-ás kezdőbittel rendelkező számok, míg a másodikban az 1-essel kezdődő elemek foglalnak helyet. Ezt követően a második bit kerül vizsgálatra az "aledényekben", az előzővel azonos módon. A rendezés befejeződött, ha minden számjegy szerinti vizsgálat megtörtént, vagy ha mindegyik, a futás alatt kialakult edény már csak egy elemet tartalmaz.

**Példa**

Az rendezendő számok: 2, 3, 1, 3, azaz binárisan 10, 11, 01, 11.

Az első ábra a kezdőállapotot mutatja. Mivel az első érték valószínűleg nem megfelelő helyen van, ezért a tömb végéről kezdve keresünk egy olyan értéket, amely 0-ás jeggyel kezdődik. A harmadik elem pont ilyen, a második képen ez a két érték cserére van kijelölve, majd a negyedik ábrán megtörténik a két elem cseréje. Az ötödik képen látható a menet során kialakult két edény. Mivel az első edény egy elemű, így nem szükséges a vizsgálata. A második edényben az 10 értéket szeretnénk cserélni, így az edény végétől haladunk előre, amíg nem találunk olyan elemet, melynek második bitje egy. A 11 érték éppen ilyen, ezért megcseréljük őket. Az utolsó ábrán látható a rendezett sorozat.



2.17. ábra. Példa a radix "előre" rendezésre

**Műveletigény**

A rendezés lineáris időben történik, a számjegyek számát jelölje  $d$ , ekkor belátható, hogy a legrosszabb esetben is, azaz ha az összes szám minden bitjét meg kell vizsgálnunk az algoritmus futási ideje:

$$T(n) = \Theta(d \cdot n) = \Theta(n)$$

**Jelölések az állapotjelző felületen**

Az éppen vizsgált elem ● színnel van jelölve. Amennyiben két elemet fel kell cserélni ● háttérszín kapnak. Továbbá a már vizsgált elemek háttérszíne fakóbb lesz. Amennyiben egy új edény keletkezik annak a színe véletlenszerűen választódik ki.

**2.3.8. Radix "vissza"****Leírás**

Az algoritmus rövid ismertetője megegyezik a **Radix "előre"** rendezés leírásával.

## Működés

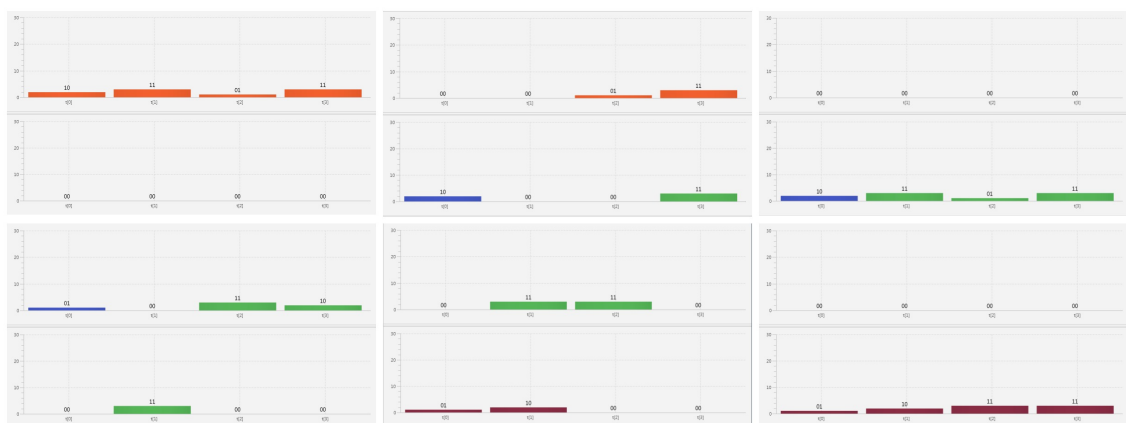
Az előző algoritmustól eltérően már nem helyben rendez, az eljárásnak két tömbre van szüksége. További különbség, hogy a kisebb helyiértéktől a nagyobb felé halad a vizsgálat.

Amennyiben az aktuálisan vizsgált bit értéke 0, akkor a "második" tömb elejére, ellenkező esetben a tömb végére töltjük át az aktuális bináris számot. Ezáltal minden egyes menetben két edény keletkezik: egy, melyben található számok aktuális bitje 0, s egy másik, melyek vizsgált számjegye 1-es. Ezt követően a 0-ás edényt az elejétől olvasva feltöltjük újra az eredeti tömböt az előbb ismertetett módon, majd a 1-es edényt elemeit az utolsó elemtől visszafelé haladva töltjük át az értékeket.

## Példa

A bemenet megegyezik a radix "előre" példájában használt számokkal, azaz: 2, 3, 1, 3, amelyek binárisan a 10, 11, 01, 11 értékek.

Az első ábra a kezdeti állapotot mutatja. A második képen a tömb első két eleme már áttöltésre került, a 10 az alsó tömb elejére, míg a 11 a végére. A harmadik ábrán látható a teljen áttöltött állapot. A következő ábrán visszatöltésre került az első érték, melynek első bitje 1, így az eredeti tömb végére került. Ekkor a második edény értékeit hátulról előre felé vizsgáljuk, így a 10 érték kerül először az eredeti tömb elejére, majd a 11 a 10 előtti helyre. Végül az utolsó 11-es érték is bekerül az eredeti tömbbe. Ezt követően az 5. ábrán már csak átmásolásra kerülnek az értékek a megfelelő sorrendben, azaz az egyes edény tartalma előről olvasva, míg a második edény értékei hátulról előre felé haladva.





2.18. ábra. Példa a radix "vissza" rendezésre

**Műveletigény**

Lineáris idejű a rendezés, amennyiben  $d$  jegyűek a rendezendő számok az algoritmus futási ideje:

$$T(n) = \Theta(d \cdot n) = \Theta(n)$$

**Jelölések az állapotjelző felületen**

Az aktuálisan vizsgált bit értékének megfelelően, amely bináris szám a 0-ás edénybe kerül , míg amely az 1-es edénybe kerül az  háttérszínt kap.

## 3. fejezet

# Fejlesztői dokumentáció

### 3.1. Tervezés és megvalósítás

A fejlesztés során több szempontot is figyelembe kell venni, úgy mint: művelet-igény, memóriaigény, jó megjelenés, egyszerű kezelhetőség, és átlátható-, bővíthető kód készítése. Mivel ezen kritériumok közül több is csak egy másik rovására javítható, ezért a tervezés során kompromisszumokat kell kötni. Továbbá fel kell készülni arra, hogy az eredeti terven a fejlesztés során módosításokat kell végezni, mivel egy-egy probléma megoldása más megközelítést kívánhat.

#### 3.1.1. Tervezés

A dolgozat fő célja egy olyan elsősorban hallgatóknak szánt program létrehozása, amellyel néhány rendezési algoritmus működése egy letisztult és egyszerű felhasználó felületen keresztül tanulmányozható.

A programnak három jól elkülönülő komponensből kell állnia: Egy logikai(modell) részből, ami gyakorlatilag a rendszer "motorja", itt kell, hogy történjen mindenféle számítási és adattárolási művelet. Egy megjelenítési rétegből, amely a logikai rész eredményeit jeleníti meg a felhasználó számára. Végül pedig egy kontroller szintből, amely kapcsolatot teremt a logikai- és a megjelenítési réteg között. A gyakorlatban ezt a fajta tagolást nevezik Modell-Nézet-Vezérlő (*MVC*) tervezési mintának.

Az elsődleges szempont az, hogy a felhasználó könnyedén tudja kezelni a programot, és segítségével megértse az algoritmusok működését. Így a felhasználói felület áttekinthetőségére és letisztultságára nagy hangsúlyt kell fektetni. Továbbá fontos az is, hogy a jövőben több rendezési eljárást is könnyedén meg lehessen jeleníteni a jelenlegiek mellett, így fontos szempont a kód egyszerű bővíthetősége.

### 3.1.2. Megvalósítás

Az első lépés a rendezési algoritmusok implementálása. Ezt követhette egyszerűbb felhasználói felület létrehozása. Kezdetben elegendő, ha csak egy grafikon jelenik meg, amely reprezentálja a tömbben található számokat.

Két algoritmushoz szükséges a gráfos megjelenítés, így a következő lépés egy gráf implementálása. Ezt követően a cél, hogy néhány "beégetett" elemre a rendezések lejátszhatóak legyenek, és az aktuális állapota a tömbnek szinkronban legyen a diagrammal valamint a gráffal. Később az egyes lépésekben történő összehasonlításokat/vizsgálatokat, mozgásokat, cseréket kell különböző színekkel jelölni az állapotjelző felületeken és számon tartani ezen műveletek összegeit.

Ezen a ponton az módosítás történt a projekt tervein. Eredetileg egy-egy külön szálon futottak volna az algoritmusok, és a felhasználói interakció hatására ezek állapota változott volna. Azonban a *JavaFX* szálkezelése jelentősen eltér az szokványos szálkezelésétől, ezért járhatóbb útnak bizonyult az, hogy kétszer kerüljenek implementálásra az algoritmusok.

Az egyik implementációban elmentjük az interakciót követő állapotot, és ez jelenik meg a felhasználói felületen. A másik megvalósításban pedig a rendezések azonnal lezajlanak, így képet kaphatunk arról, hogy mennyi műveletre volt szükség az egyes eljárások során. Ezen utóbbi implementációk mindegyike külön szálon fut, és ahogy valamelyik befejeződik figyelmezteti a főprogramot, hogy jelenítse meg a műveletek számát. Valamint minden szálhoz egy-egy egyszerűbb egységtesztet kell készíteni.

Miután a program alapjai elkészültek kezdetét veheti a felhasználói felület részletes kialakítása. Elsőként a diagram elhelyezése egy panelen, amely tartalmaz továbbá egy listát a választható algoritmusokról. Illetve egy táblázatot, melyben szerepelnek az aktuális állapot egyes tulajdonságai.

A programnak egy fontos szolgáltatása az, hogy a felhasználó különböző adatbeviteli mód közül választhat. A logikai réteget ki kell bővíteni ezekkel az esetekkel, továbbá a felhasználói felületen lehetőséget adni ezen módok kiválasztására.

Ezután az eszköztár kerül a helyére, mellyel párhuzamosan megtörténik az egyes műveletekhez tartozó eljárások implementálása.

Az utolsó teendő a felhasználói felületen egy rendezések összehasonlítására lehetőséget adó panel létrehozása, táblázattal, benne az algoritmusok műveletigényével. Továbbá egy diagrammal, amin megjelenik a táblázatból kiválasztott sor összehasonlításainak és mozgásainak a száma.

Végül, hogy a program egyszerűen használható legyen *Windows* környezetben egy telepítő fájl készítése, amellyel az előbb említett operációs rendszert használóknak nem szükséges külön *Java*-t telepíteniük.

### 3.1.3. Használt fejlesztőeszközök

A fejlesztés *Eclipse SDK 4.4* fejlesztői környezet keretei között történt. A program grafikus fejlesztői felületet ad alkalmazások készítéséhez.

A program elkészítése során a kódolást segítő funkció volt a kódkiegészítés, továbbá az egyik beépített projektmenedzsment eszköz(*EGit*).

A fejlesztéshez elengedhetetlen a *Java SE 8u40* vagy magasabb verziójú szoftver. Továbbá a fejlesztést nagyban elősegítette a *JavaFX Scene Builder 2.0*, melynek segítségével egyszerűen megtervezhetővé váltak a grafikus felület komponensei.

Végül a telepítési környezet létrehozásához *Ant* és *InnoSetup* eszközök kerültek felhasználásra. Az egész projekt, beleértve e dokumentumot is megtalálható, és az egyes verziók visszakövethetők a *GitHub*-on: <https://github.com/marfoldi/SRTNGLGRTHMS>

A program fejlesztése során egyedüli külső függvénykönyvtár a *JUnit* volt, melynek segítségével egységtesztek készültek.

## 3.2. Használati esetek

A következőkben a főbb használati esetek kerülnek bemutatásra. A felhasználónak tudnia kell:

- rendezendő számok sorozatát megadni;
- algoritmust lejátszani;
- műveletszámokat összehasonlítani;
- rendezés leírását elolvasni;
- program névjegyét megtekinteni;
- programot bezárni;
- esetleges hibákról értesülni;

## 3.3. Osztályok leírása

A program tervezésekor az egyik alapelv volt, hogy a grafikus megjelenítés, működés szintjén minél inkább különüljön el a rendszerlogikától. A grafikus felület már csak a modell programész által számított, a vezérlő programrésznek átadott adatokat jelenítse meg. Továbbá lehetőséget biztosítson a felhasználó számára, hogy vezérelje a program rendszerlogikáját a kontroller rétegen keresztül. Ez által, hogy a rendszerlogika lényegében független a grafikus interfésztől. Így a szoftver magasabb fokú bővíthetőséget, a kód pedig jobb átláthatóságot nyer.



Ez az elv jól megfigyelhető a programkód struktúrájában is, alapvetően a program három fő csomagra lett bontva: A **modell** csomag, melyben a logikai osztályok találhatóak, a **view**, melyben a grafikai interfészt leíró *fxml* fájlok foglalnak helyet, és végül a **controller** csomag, melyben található osztályok kapcsolatot teremtenek a logikai és a megjelenítési réteg között.

### 3.3.1. Modell osztályok

A modell osztályok a **modell** csomag foglalja magában. A pontosabb rendszerezést szem előtt tartva további alcsoomagok kerültek létrehozásra, így jobban áttekinthető az osztályhierarchia. Összességében ebben a csomagban foglalnak helyet azon osztályok, amelyek a rendszer adattároló, rendszerező és számítási feladatait ellátják. Az alábbi táblázat ad áttekintést az előzőleg említett alcsoomagokról:

Név	Leírás
algorithm	A lejátszható algoritmusok implementációját tartalmazza
algorithm.raw	Az "egyszerű" rendezést megvalósító osztályok csomagja
graph	A gráf adatszerkezet implementációja

Elsőként a **algorithm** csomagot részletezzük, röviden az alábbi táblázat foglalja össze a benne található osztályok feladatait:

## 4. fejezet

### Irodalomjegyzék

- [1] *Java (programming language)*, Wikipedia the free encyclopedia. [ONLINE] [Hivatkozva: 2015.04.21] [http://en.wikipedia.org/wiki/Java\\_\(programming\\_language\)/](http://en.wikipedia.org/wiki/Java_(programming_language)/)
- [2] *JavaFX*, Wikipedia the free encyclopedia. [ONLINE] [Hivatkozva: 2015.04.21] <http://en.wikipedia.org/wiki/JavaFX/>
- [3] *JUnit*, Wikipedia the free encyclopedia. [ONLINE] [Hivatkozva: 2015.05.01] <http://en.wikipedia.org/wiki/JUnit>
- [4] Demuth, H.: *Electronic Data Sorting*, PhD thesis, Stanford University, 1956, [184]
- [5] Dr. Fekete István: *Algoritmusok és adatszerkezetek I. jegyzet*, [ONLINE] [Hivatkozva: 2015.04.20] [http://people.inf.elte.hu/fekete/algoritmusok\\_bsc/alg\\_1\\_jegyzet/](http://people.inf.elte.hu/fekete/algoritmusok_bsc/alg_1_jegyzet/)
- [6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Új algoritmusok*, Sclolar kiadó, 2003, [992], 9789639193901
- [7] *Shellsort*, Wikipedia the free encyclopedia. [ONLINE] [Hivatkozva: 2015.04.25] <http://en.wikipedia.org/wiki/Shellsort/>
- [8] Donald Shell: *A high-speed sorting procedure* Communications of the ACM, 2, 7, 1959
- [9] C.A.R. Hoare: *Algorithm 64: Quicksort* Communications of the ACM, 4, 7, 1961
- [10] The Takipi Blog: *We Analyzed 30,000 GitHub Projects – Here Are The Top 100 Libraries in Java, JS and Ruby*, [ONLINE] [Hivatkozva: 2015.04.30] <http://blog.takipi.com/>

we-analyzed-30000-github-projects-here-are-the-top-100  
-libraries-in-java-js-and-ruby/