Algoritmusok és adatszerkezetek

Rendezési algoritmusok elemzése

Kovács Norbert
ANOXWJ

2020. május 8.

Tartalomjegyzék

1. fejezet

Bevezetés

Rendezésnek nevezünk egy algoritmust, ha az valamilyen szempont alapján sorba állítja elemek egy listáját. A rendezési algoritmusok a programozás kezdete óta jelen vannak és az érdeklődés középpontjában állnak, mivel egy rendezett adathalmazzal több és hatékonyabb műveletek végezhetők, mint egy rendezetlennel. A rendezések hatékonyságát, általában a szükséges összehasonlítások és cserék átlagos/maximális száma alapján ítéljük meg. Általánosan elmondható, minél gyorsabb egy rendezési algoritmus, annál jobban szeretjük. A dokumentumban Python kódokkal fogjuk szemléltetni az algoritmusokat, és futási idejüket.

Környezet paraméterei:

- **OS**: Ubuntu 19.10 x86_64
- **Kernel**: 5.3.0-51-generic
- **Shell**: bash 5.0.3
- **Python version**: 3.7.5

2. fejezet

Rendezések elemzése

A következőkben sorra megyünk a különböző rendezési módszereken. Rövid leírás után, szemléltetjük az algoritmust. Ahhoz, hogy megfelelő becslő értékeket kaphassunk a futási időről, különböző elemszámokon teszteljük az algoritmusokat. Minden elemszámra minimum ötször elvégezzük a rendezést, egy véletlen számokkal feltöltött sorozaton. Ebből átlagolva, megkapjuk a közelítésünket az adott algoritmus sebességére. Ezeket táblázatba[1] rendezzük, és diagramot készítünk az adatokból. Így láthatóvá tesszük, a sejtéseinket, hogy az algoritmus milyen időbonyolultsággal rendelkezik.

Kiindulópontom az, hogy a rendezési algoritmusoknál, a legjobb eset, az ha a sorozat már eleve rendezett, így nincs rendezni valónk a tömbön. A legrosszabb eset, hogy ha a sorozat fordítva rendezett, mint ahogy szeretnénk. Mivel minden rendezési algoritmus kicsit más, ezért máshogy fognak reagálni (futási idő tekintetében) ezekre a speciális helyzetekre, és meg is cáfolhatják a kiindulási alapomat. Minden rendezésnél megpróbáljuk elemezni hogy miként reagálnak, és levonni a megfelelő következtetést.

2.1. Beszúró rendezés

A beszúró rendezés esetén, egy emberközeli gondolkodásmódot alkalmazunk. Alapvetően mindig, egy elem helyét keressük, és visszük a helyére. A sorozat egy részét rendezettnek tekintjük. Ez kiindulásként a lista nulladik eleme lesz. Egy elem önmagában rendezett, tehát nincs sok teendőnk. A következőkben, megvizsgáljuk, hogy a következő elem milyen relációban áll a már rendezett sorozatunkkal, és oda mozgatjuk, ahol lennie kell. Ne felejtsük el, hogy ez még igen kis valószínűséggel az adott elem legvégső pozíciója. Azonban, ezután rendezettnek tekinthető ismét a sorozat egy része, mégpedig az első két elem. Ismét megvizsgáljuk, a következő elem viszonyát a már rendezett sorozatunkkal, és a megfelelő pozícióra helyezzük. Ezt folytatva, egészen addig amíg az egész sorozatunkat rendezettnek nem tekinthetjük, érünk az algoritmus végére.

[1] A táblázatokat csak az első példánál jelenítjük meg, ezután a dokumentum terjedelme miatt csupán a diagramokat fogjuk szemléltetni.

Beszúró rendezés
Python nyelven implementálva

```
import time,random

def insertionSort(array):
        for j in range(len(array)-1):
                i = j
                x = array[j+1]

                while(i >= 0 and array[i] > x):
                        array[i + 1] = array[i]
                        i = i - 1
                array[i + 1] = x

testArray = []
for i in range(1000):
        testArray.append(random.randint(0,100))

beginning = time.time()
insertionSort(testArray)
end = time.time()
print(end - beginning)
```

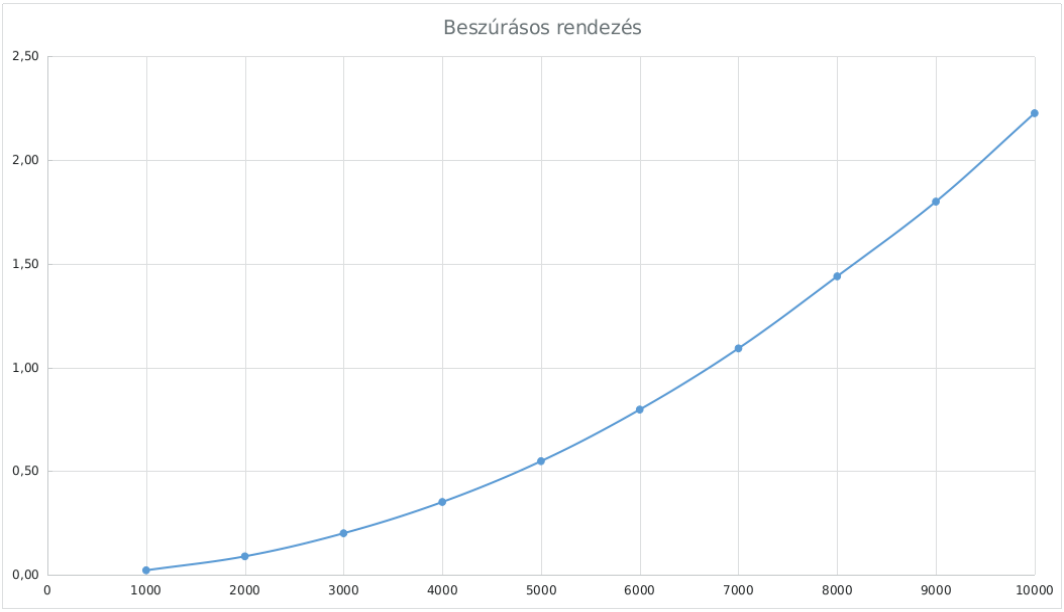
Az algoritmust egymás után lefuttatva, változó elemszámra (ezresenként), lejegyzetelve a futási időket, elkésztettünk egy táblázatot. Ez a folyamat igen időigényes. Sokkal kényelmesebb lenne, automatizálni a mérések folyamatát, ahol a program maga ellenőrzi a futási időt, futtatja le többször az algoritmust, veszi az átlagot, és írja ki az eredményt. Így lehetőségünk is nyílik arra hogy ne csupán öt darab mintavétel történjen, hanem amennyit szeretnénk. Ez mindenképp felgyorsítja, az elkészült algoritmus elemzését, azonban most, végezzük el a méréseket manuálisan.

2.1. táblázat. A futási idők

Elemszám	**Véletlenszerű elemek (sec)**	**Legrosszabb eset (sec)**
1000	0,021235	0,041438
2000	0,088552	0,169964
3000	0,199812	0,380816
4000	0,350660	0,687677
5000	0,547545	1,062555
6000	0,796298	1,530954
7000	1,091439	2,087298
8000	1,438931	2,754825
9000	1,799304	3,495871
10000	2,225763	4,299924

Az elkészült diagram

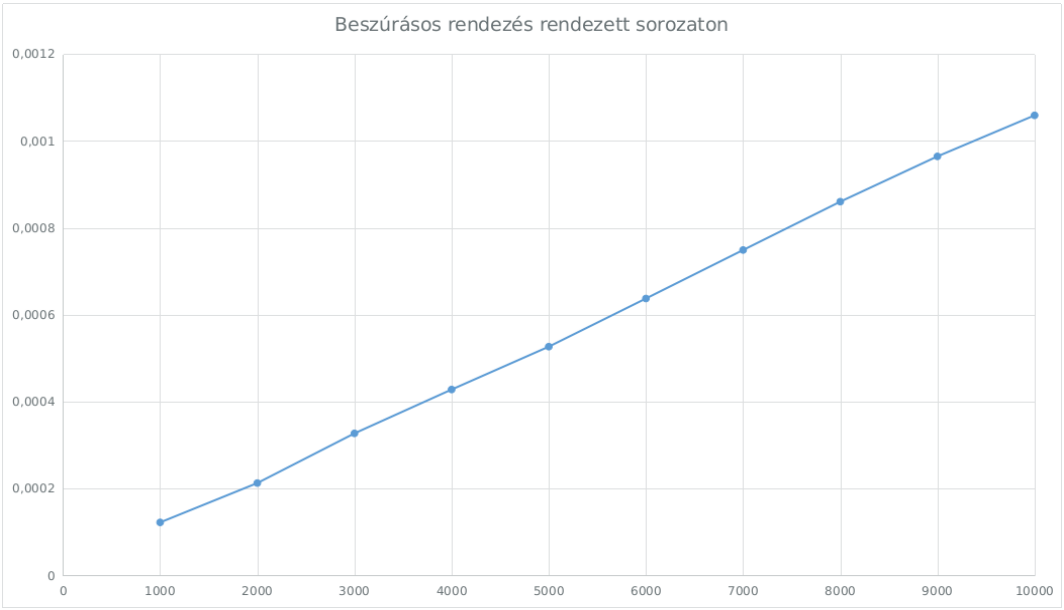
Futási idő különböző elemszámok esetén.



Látható hogy az algoritmus véletlenszerűen generált sorozatnál, négyzetes bonyolultsággal viselkedik.[2] Tízezer elemszámnál a rendezés futási ideje megközelíti a 2,5 másodpercet.

Rendezett sorozaton

Futási idő különböző elemszámok esetén.

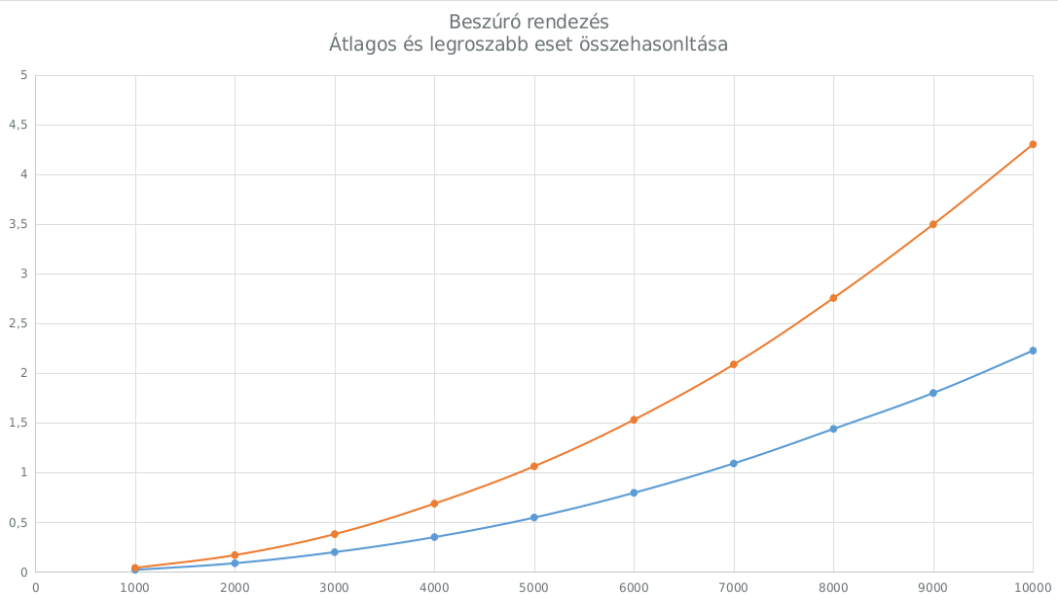


[2] A legtöbb rendezési algoritmusnál négyzetes időbonyolultságra számítunk.

Mivel az általam vélt ideális eset időegységei össze sem hasonlíthatóak az átlagos esetekhez szükséges időhöz, így azt külön diagramon tekintjük meg. Észrevehetjük, hogy a legmagasabb elemszám esetén is, csupán a másodperc töredéke, mire lefut az algoritmus.

Fordítva rendezett sorozaton

Összehasonlítás.



A következőkben többnyire összehasonlító diagramokon fogjuk szemléltetni az egymáshoz viszonyított futási időket. Jól látható hogy a legrosszabb esetnél (narancssárga) a rendezési idő az átlagoshoz (kék) képest megduplázódik.

2.2. Kiválasztásos rendezés

A módszer, két részre osztja a tömbünket. Egy rendezett, és egy még nem rendezett részre (A rendezett rész eleinte üres). Ezután a rendezetlen részben megkeresi az aktuális minimum értéket, és a „rendezett” lista végére teszi. A gyakorlatban ezt egy elemcserével oldjuk meg. Ezután, a rendezetlen részben, újra megkeressük az aktuális minimum értéket, és hozzáillesztjük a rendezett részsorozathoz. Amint végigérünk a tömbön, növekvő sorrendű sorozatot kapunk, hiszen biztosítottuk hogy mindig az aktuális legkisebb elem kerüljön a rendezett lista végére.

Kiválasztásos rendezés algoritmusa

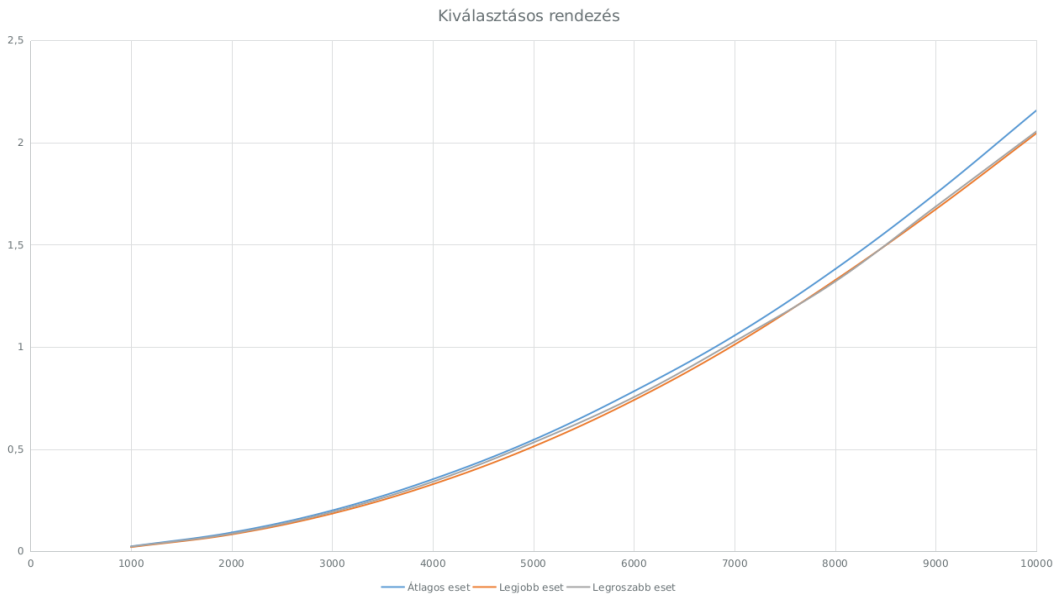
Egyszerű, manuális változat.

```
import random, time

def selectionSort(array):
        for j in range(len(array)):
                for i in range(j+1, len(array)):
                        if(array[i] < array[j]):
                                c = array[i]
                                array[i] = array[j]
                                array[j] = c

testArray = []
for i in range(1000):
        testArray.append(random.randint(0,100))

beginning = time.time()
selectionSort(testArray)
end = time.time()
print(end -  beginning)
```

Ez esetben, már nem szeretnénk megjeleníteni a mérési adatokat táblázatban, hiszen a következtetés könnyedén leolvasható a diagramról is.

Kiválasztásos rendezés

Összehasonlítás.



Elvégezve a méréseket, a látható diagramot kaptuk. Amit elsőként levonhatunk, hogy a futási idők összeolvadnak. A legjobb eset, és az átlagos eset alig tér el egymástól.

Ami azonban nagyobb érdekesség, hogy a „legrosszabb” eset, a kettő között, de inkább a legjobb esethez tapadva látható. Tehát, ennek a rendező algoritmusnak, a levont következtetések szerint, közel lényegtelen hogy eleve rendezett, fordítva rendezett, vagy véletlenül feltöltött sorozatot kap, az időbonyolultsága változatlan, és az általam gondolt kiindulási alap megcáfoltnak tűnik, hiszen nem egyértelműen kijelenthető hogy a fordítva rendezett sorozat, rontaná az algoritmus futási idejét. Azonban itt még annyira közel vannak egymáshoz az eredmények, és a mintavétel nem túl nagy, hogy nem vonnék le túl hamar következtetéseket.

2.3. Buborékrendezés

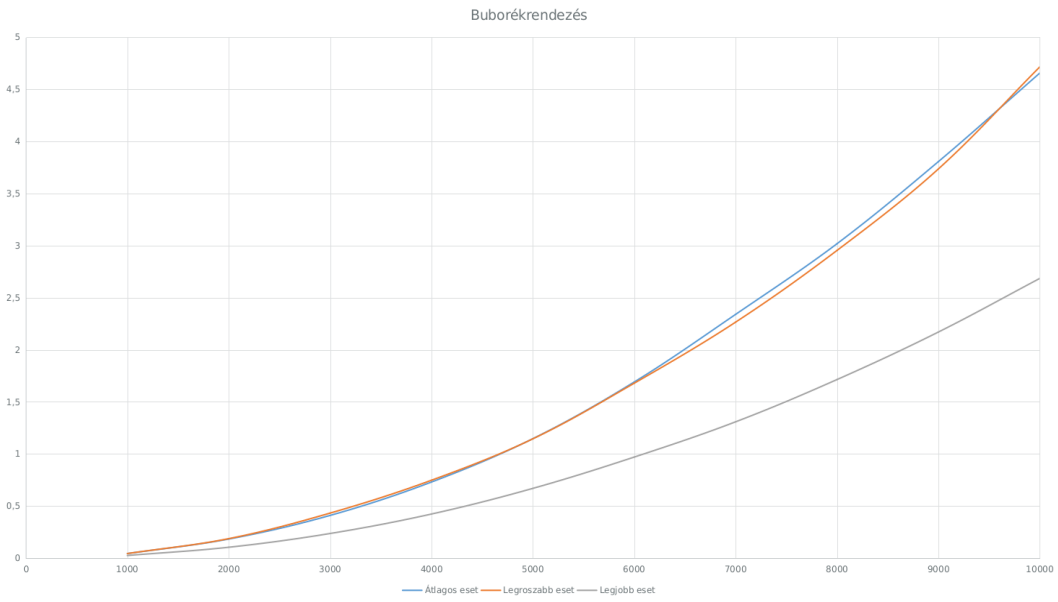
2.3.1. Buborékrendezés 1.

A sorozatok rendezettsége ellenőrizhető a szomszédos elemek összehasonlításával. Nevezetesen, ha bármely két szomszédos elem sorrendje megfelelő, akkor a sorozat is rendezett. Ezt használja a buborékrendezés algoritmusa is.

```
1 def bubbleSort(array):
2         for i in range(len(array)-1):
3                 for j in range(len(array)-i-1):
4                         if(array[j]>array[j+1]):
5                                 c = array[j]
6                                 array[j] = array[j+1]
7                                 array[j+1] = c
```

Buborékrendezés

Alapalgoritmus futási idők.



Ez esetben azt tapasztaljuk, hogy a legrosszabb (visszafelé rendezett), és az átlagos (véletlenszerű elemek) eset összefonódik, azonban jelentős sebességbeli növekedés látható amikor eleve rendezett sorozatot kapott az algoritmus.

2.3.2. Buborékrendezés 2.

Ugyanakkor vegyük észre, hogy a buborékrendezés során azon túl, hogy a nagyobb értékű elemek sorozat vége felé gyorsan vándorolnak, a kisebbek a sorozat eleje felé mozdulnak el egy hellyel. Ez azt jelenti, hogy miközben a sorozat végén a rendezett rész elemszáma eggyel nőtt, addig az ettől jobbra lévő elemek rendezettsége is megváltozik, esetleg rendezetté is válhat. Az algoritmus módosításával elérhető, hogy az „felismerje" azt, ha a rendezettség a kisebb indexű elemek körében is bekövetkezett.[3] Ekkor ugyanis a teljes sorozat rendezetté vált.

```
import time,random

def bubbleSort(array):
        j = 0
        exchange = True
        while(j <= len(array) and exchange):
                exchange = False
                for i in range(len(array)-j-1):
                        if(array[i]>array[i+1]):
                                c = array[i]
                                array[i] = array[i+1]
                                array[i+1] = c
                                exchange = True
                j += 1

def randomizer(array, size):
        for i in range(size):
                array.append(random.randint(0,100))

testArray = []
randomizer(testArray,1000)
beginning = time.time()
bubbleSort(testArray)
end = time.time()
print(end - beginning)
```

Hogy ne manuálisan kelljen elvégezni a méréseket, ezért automatizáljuk a folyamatot. Először is, szükségünk van egy függvényre, ami majd átlagolja egy sorozat elemeinek összegét, hiszen több mérés átlagát tekintjük hitelesnek. Ez kifejezetten egyszerű, a beépített függvények segítségével.

```
def average(array):
        return sum(array) / len(array)
```

[3] Ez a buborékrendezés első javítása

Ezután végezzük el a méréseket, és növeljük ezresenként az elemszámot. Ennek az automatizálásával sokkal hatékonyabbá, gyorsabbá tesszük a mérési folyamatokat. Csupán át kell irányítanunk egy fájlba az eredményeket, és könnyedén elkészíthetjük a diagramunkat. Ennek a megvalósításáért felelős a következő függvény:

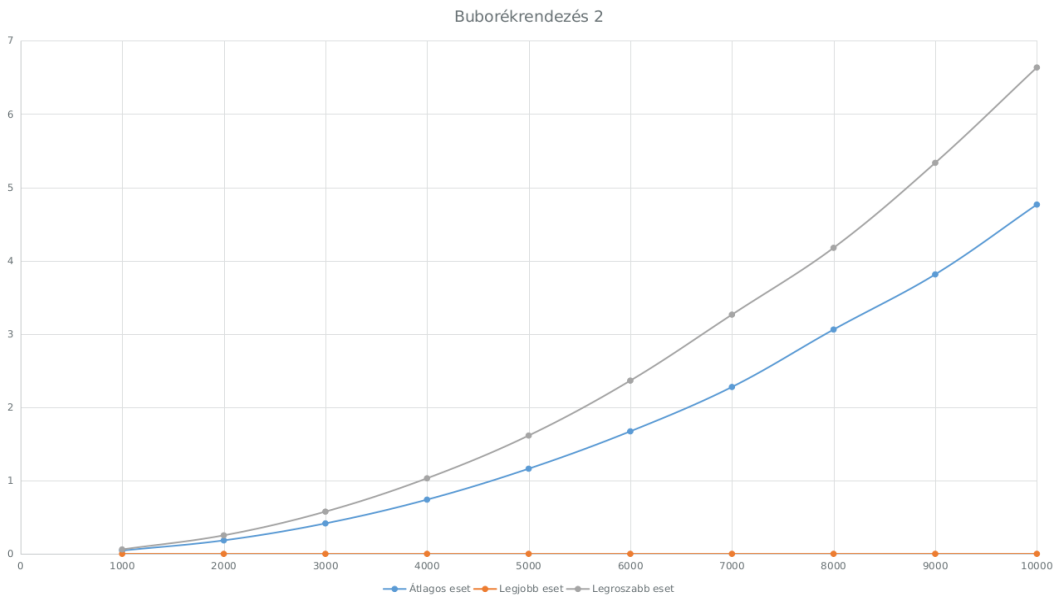
Automatizálást segítő függvény

Adatok létrehozása.

```
def createData(array, count):
        data = []
        for i in range(1000,11000,1000):
                times = []
                for j in range(count):
                        array = []
                        randomizer(array, i)
                        beginning = time.time()
                        bubbleSort(array)
                        end = time.time()
                        times.append(end - beginning)
                print(str(i) + " : " + str(average(times)))
                data.append(average(times))
        return data

testArray = []
data = createData(testArray,5)
```

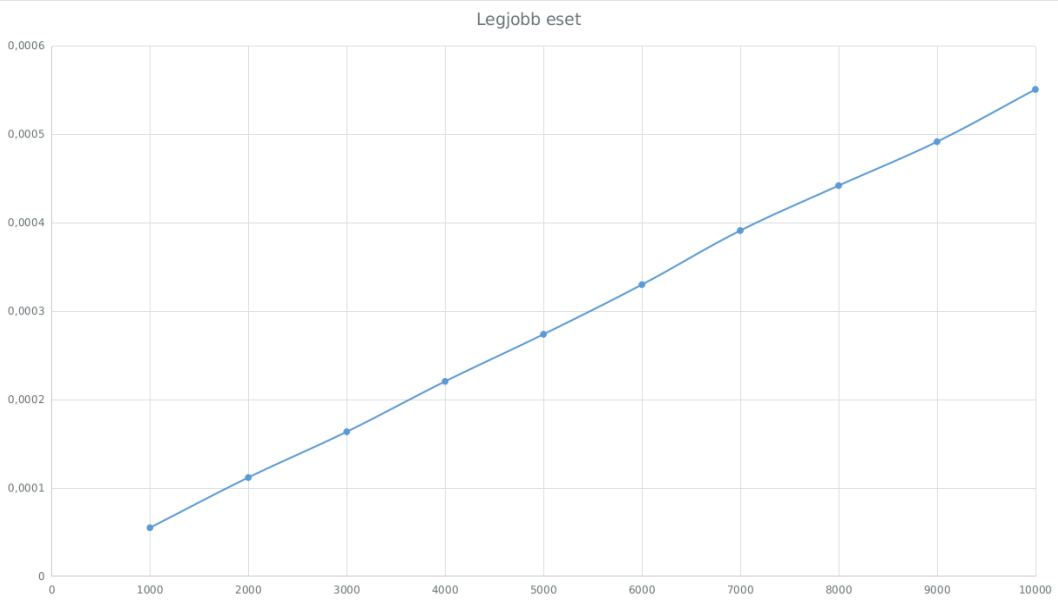
Buborékrendezés

Összehasonlítás.



Könnyedén belátható, hogy a rendezett tömb esetén, a szükséges idő mértéke össze sem hasonlítható az átlagos, vagy legrosszabb esetéhez. Azonban látható, hogy a legrosszabb esetben, az elemszám növekedésével egyre távolabbra kerülünk az átlagos végrehajtási időtől. Kifejezetten érdekes, amennyiben nem hibás az általam írt algoritmus, hogy a javított kód átlagos futási ideje nem gyorsult, a legrosszabb esetre pedig romlott is, az alap buborékrendezéshez viszonyítva.

Buborékrendezés - Legjobb eset
Önmagában.



2.3.3. Buborékrendezés 3.

A javított buborék-rendezés „csak” azt képes fölismerni, hogy a sorozat baloldali, rendezésre váró része teljesen rendezetté vált. Ezt az jelzi, hogy belső ciklus lefutása során nem volt szükség cserére. Ugyanakkor előfordulhat, hogy belső ciklusban jóval a ciklusváltozó végértékének elérése előtt történt az utolsó csere, azaz a sorozat az utolsó csere helyétől már rendezett. Ezt képes fölismerni a buborékrendezés 3.[4] algoritmusa:

[4] A buborékrendezés második javítása.

```
import time,random

def bubbleSort(array):
        j = 0
        while(j <= len(array)-1):
                lastExchangeIndex = 0
                for i in range(0,len(array)-j-1):
                        if(array[i] > array[i+1]):
                                c = array[i]
                                array[i] = array[i+1]
                                array[i+1] = c
                                lastExchangeIndex = i+1
                j = len(array) - lastExchangeIndex
```

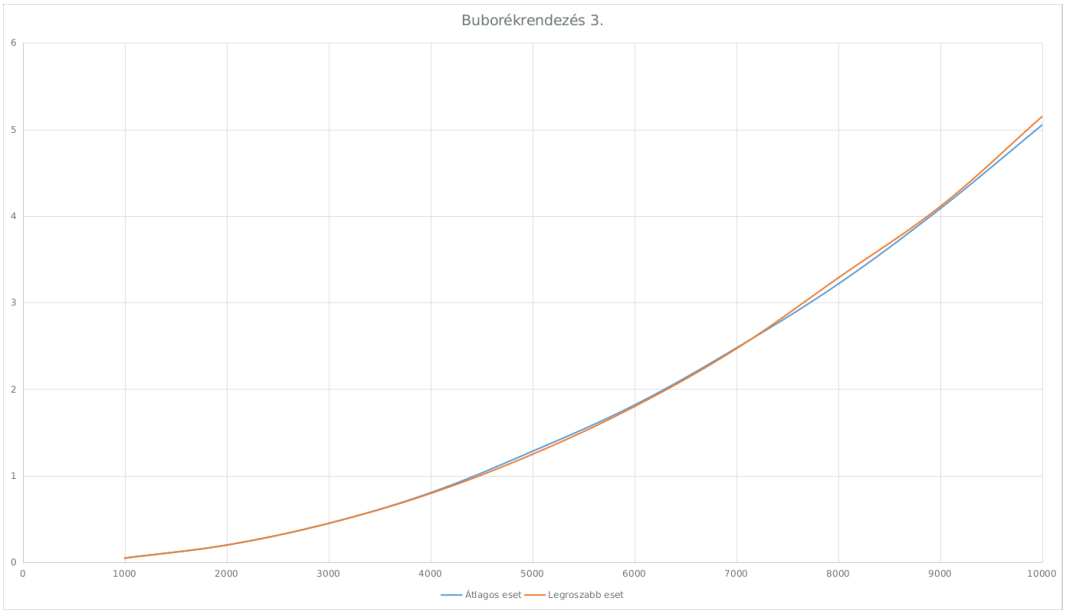
Lássuk az automatizálást segítő függvényeket:

```
def bubbleSortBack(array):
        for i in range(len(array)-1):
                for j in range(len(array)-i-1):
                        if(array[i]<array[i+1]):
                                c = array[i]
                                array[i] = array[i+1]
                                array[i+1] = c

def average(array):
        return (sum(array) / len(array))

def randomizer(array, size):
        for i in range(size):
                array.append(random.randint(0,100))

def createData(array, count):
        data = []
        for i in range(1000,11000,1000):
                times = []
                for j in range(count):
                        array = []
                        randomizer(array, i)
                        #bubbleSortBack(array)
                        beginning = time.time()
                        bubbleSort(array)
                        end = time.time()
                        times.append(end - beginning)
                print(str(i) + " ; " + str(average(times)))
                data.append(average(times))
        return data

testArray = []
data = createData(testArray,5)
```

Szükségünk van egy függvényre, ami visszafelé rendezi le a sorozatot. Ha az átlagos esetre vagyunk kíváncsiak, kikommenteljük a rendezési algoritmust az időmérés előtt, ha a legjobbra, megfelelő sorrendben berendezzük, és az időmért függvény már rendezett sorozatot kap.

Buborékrendezés 3

Összehasonlítás.



A legjobb esetet fel sem tüntetjük, nincs változás az előzőhöz képest. A legrosszabb eset, és az átlagos eset teljesen összeolvadnak, látható javulás az előző buborékrendezéshez képest, a legrosszabb esetben, a legnagyobb elemszámnál, másfél másodperccel gyorsult az algoritmus.

2.4. Koktélrendezés

A fentebb bemutatott buborékrendezés algoritmusának nagy hátránya, hogy a sorozat végéhez közel található kis értékű elemek (teknősök) nagyon lassan „találják meg” helyüket a sorozat elején, míg a sorozat elején lévő nagy értékű „nyulak” jóval gyorsabban a végleges helyükre kerülnek.

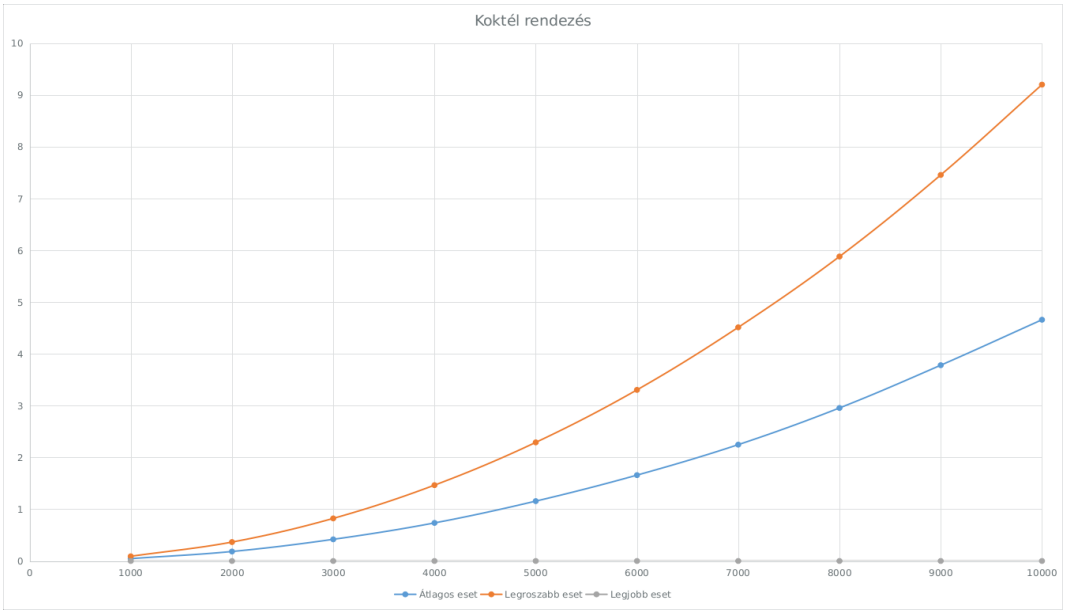
Koktélrendezés algoritmusa

```
def cocktailSort(array):
        up = range(len(array)-1)
        while(True):
                for i in (up, reversed(up)):
                        swapped = False
                        for j in i:
                                if(array[j] > array[j + 1]):
                                        c = array[j]
                                        array[j] = array[j + 1]
                                        array[j + 1] = c
                                        swapped = True
                        if not swapped:
                                return
```

A „Koktél rendezés” elvi alapja tehát a fentebb bemutatott buborékrendezés.

Koktélrendezés

Összehasonlítás.



A legjobb esetben, tehát ha a sorozat alapesetben rendezett, a futási idő elhanyagolható. Átlagos esetben összehasonlítható a buborékrendezéssel, de picivel hatékonyabb. Legrosszabb esetben viszont a rendezési idő megduplázódik.

2.5. Fésűs rendezés

A fésűs rendezés (combsort) algoritmus egy tömb elemeinek sorba rendezésére. A buborékrendezés egy javított módszere. A buborékrendezés talán az összes rendezési algorit-

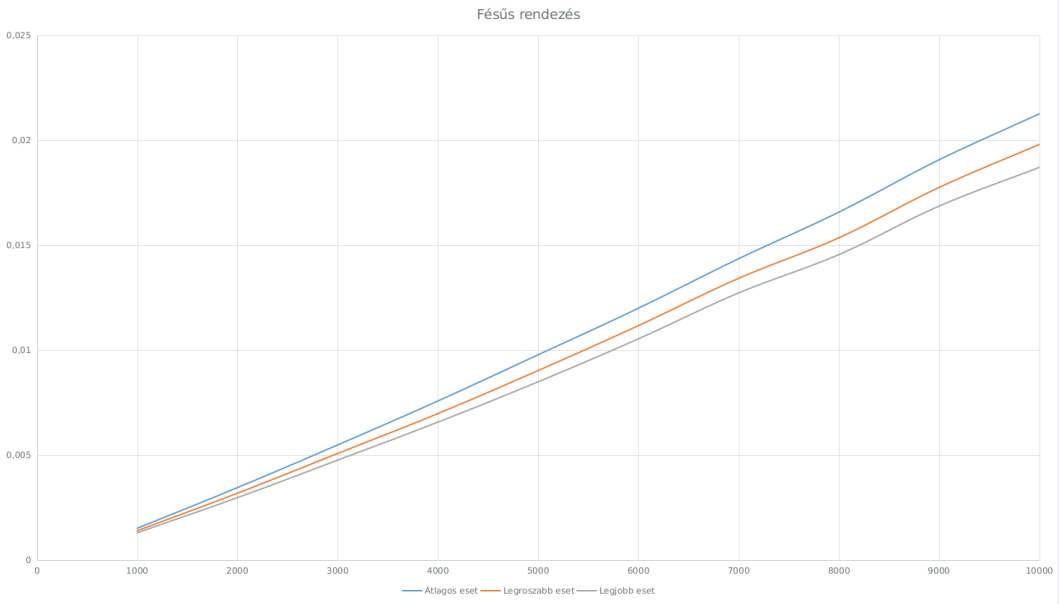
mus közül a legrosszabb, azonban egy egyszerű módosítással úgy felgyorsítható, hogy a gyorsrendezés sebességével vetekszik.

```
def combSort(array):
        gap = len(array)
        swaps = True
        while(gap > 1 or swaps):
                gap = max(1, int(gap / 1.25))
                swaps = False
                for i in range(len(array)-gap):
                        j = i + gap
                        if(array[i] > array[j]):
                                c = array[i]
                                array[i] = array[j]
                                array[j] = c
                                swaps = True
```

Ez esetben a mintavételt 5-ről, 100-ra növeltük, mivel annyira gyors az algoritmus, hogy így is a másodperc törtrésze alatt elkészül az összes mérés.

Fésűs rendezés

Összehasonlítás.



Érdekesség, hogy az átlagos eset került a legtöbb időbe, nem pedig a visszafelé rendezett sorozat. Ez szintén tovább növeli a gyanút, hogy nem minden algoritmus számára jelenti a legrosszabb esetet a visszafelé rendezettség. Az algoritmus lineáris futási idejű, és még tízezer elemszámnál is, csupán a másodperc törtrésze alatt elvégzi a feladatot.

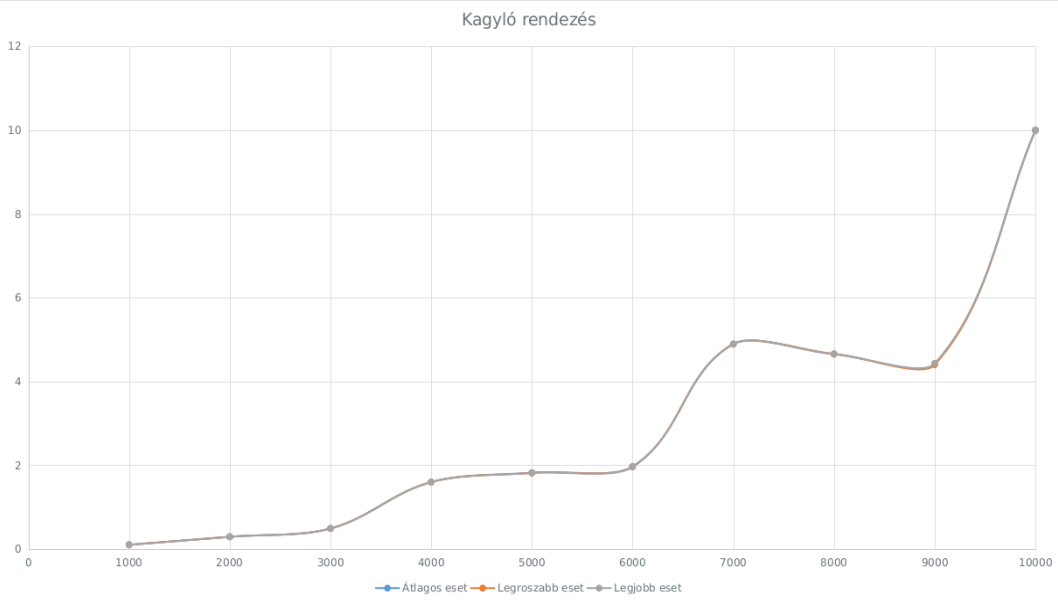
2.6. Kagyló rendezés

Az alapgondolat szerint kezdetben egymástól távolabbi elemeket hasonlítunk és mozgatunk, ami általában azt eredményezi, hogy az elemek gyorsabban közelítenek a végleges helyükhöz.

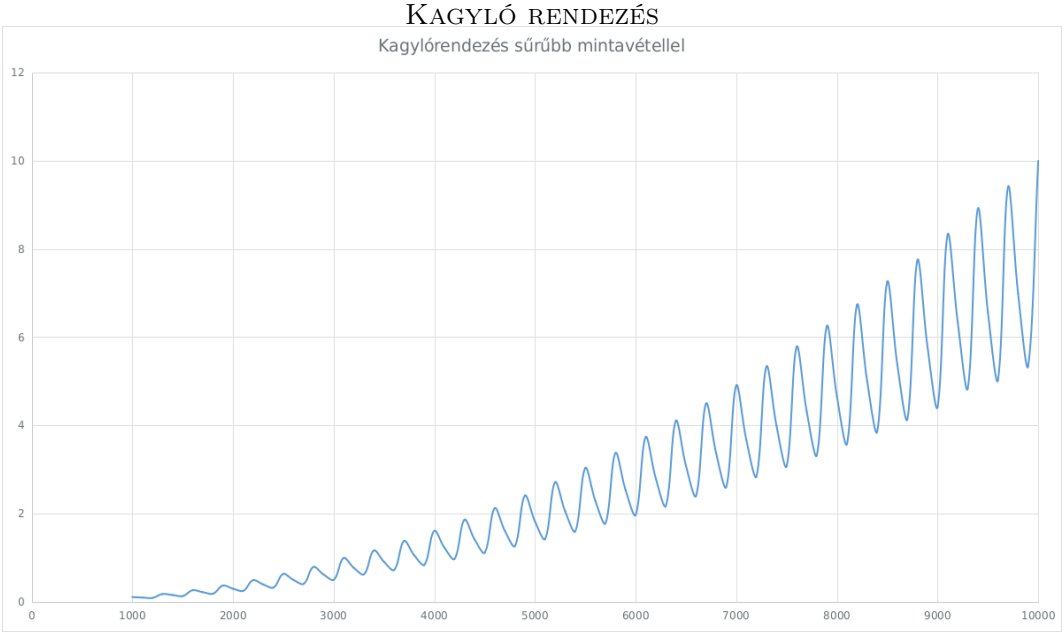
```
def shellSort(array):
        d = len(array)
        while(True):
                d = d % 3 + 1
                for k in range(d):
                        j = k
                        while(j <= len(array) - d):
                                minIndex = j
                                i = j + d
                                while(i <= len(array)-1):
                                        if(array[i] < array[minIndex]):
                                                minIndex = i
                                        i = i + d
                                if(j != minIndex):
                                        c = array[j]
                                        array[j] = array[minIndex]
                                        array[minIndex] = c
                                j = j + d
                if(d == 1):
                        return
```

Kagyló rendezés

Összehasonlítás.



A különböző esetek teljesen összeolvadnak, és mivel az megszokottól különlegesebb viselkedést tanúsít az algoritmus, ezért sűrítem a mintavétel gyakoriságát.



Ennél is sűrűbb mintavétellel, és nagyobb átlagolt mintából, még pontosabb diagramot kaphatnánk, azonban ez már elég részletes ahhoz, hogy lássuk az algoritmus viselkedését. Ha a függvényt középen nézzük, vagy minden „50”-edik elemnél vesszük az értékét, akkor egy négyzetes görbét kapunk. Megközelítőleg 5000 elemnél két másodpercre van szükségünk, 7000 elemnél pedig megközelítőleg négyre. Tehát a szükséges idő gyorsabban növekszik, mint az elemszámok.

2.7. Összefuttatás

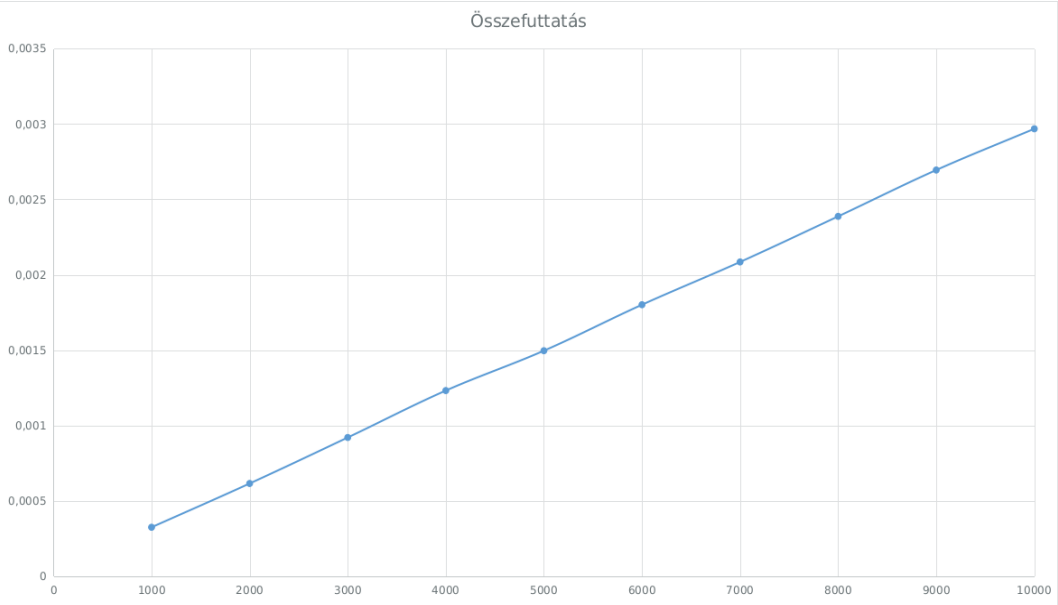
A rendezett sorozatok feldolgozása során szükségessé válhat két (azonos módon) rendezett sorozat elemeinek egy rendezett sorozatban való egyesítése. Kézenfekvő megoldásnak tűnne a sorozat elemeit egy adatszerkezetben egyszerűen egymás után másolni, majd az így kapott, most már rendezetlen sorozatot az egyik korábban megismert rendező algoritmussal rendezni. Az összefuttatás tétele egy ennél jóval hatékonyabb megoldást kínál.

Összefuttatás algoritmusa

```
def mergeSort(array_1, array_2, result):
        aI = 0
        bI = 0
        cI = -1
        while(aI < len(array_1) and bI < len(array_2)):
                cI += 1
                if(array_1[aI] < array_2[bI]):
                        result[cI] = array_1[aI]
                        aI += 1
                else:
                        if(array_1[aI] > array_2[bI]):
                                result[cI] = array_2[bI]
                                bI += 1
                        else:
                                result[cI] = array_1[aI]
                                aI += 1
                                cI +=1
                                result[cI] = array_2[bI]
                                bI += 1
        while(aI <= len(array_1)-1):
                cI += 1
                result[cI] = array_1[aI]
                aI += 1
        while(bI <= len(array_2)-1):
                cI += 1
                result[cI] = array_2[bI]
                bI += 1
```

Összefuttatás diagram

Futási idő.



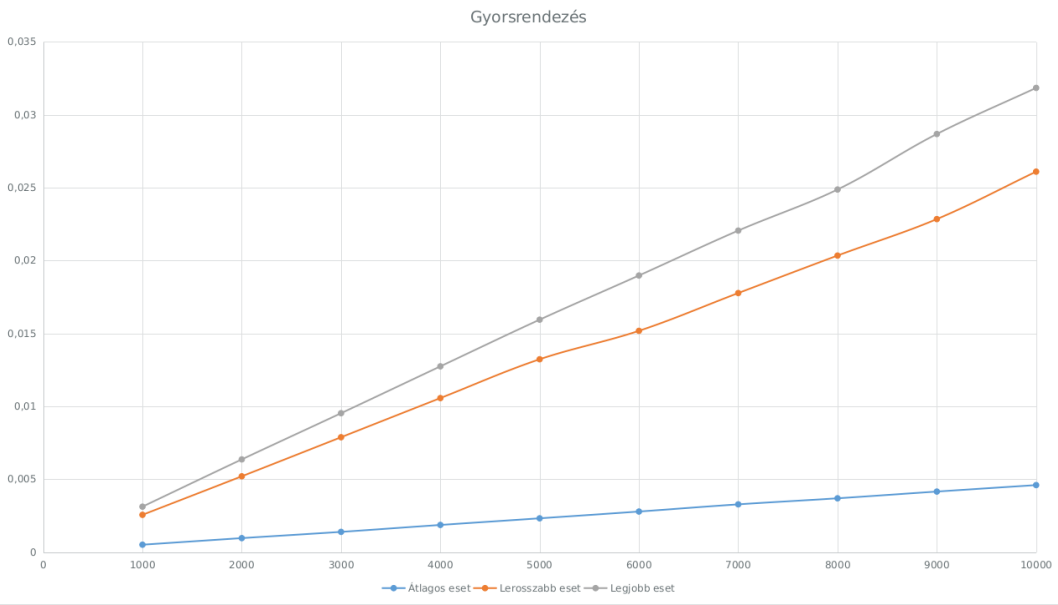
Belátható, hogy két eleve rendezett sorozat összefuttatása igen hatékony.

2.8. Gyors rendezés

Feltételezhetően a leggyorsabb rendező algoritmus, az elemzettek közül.

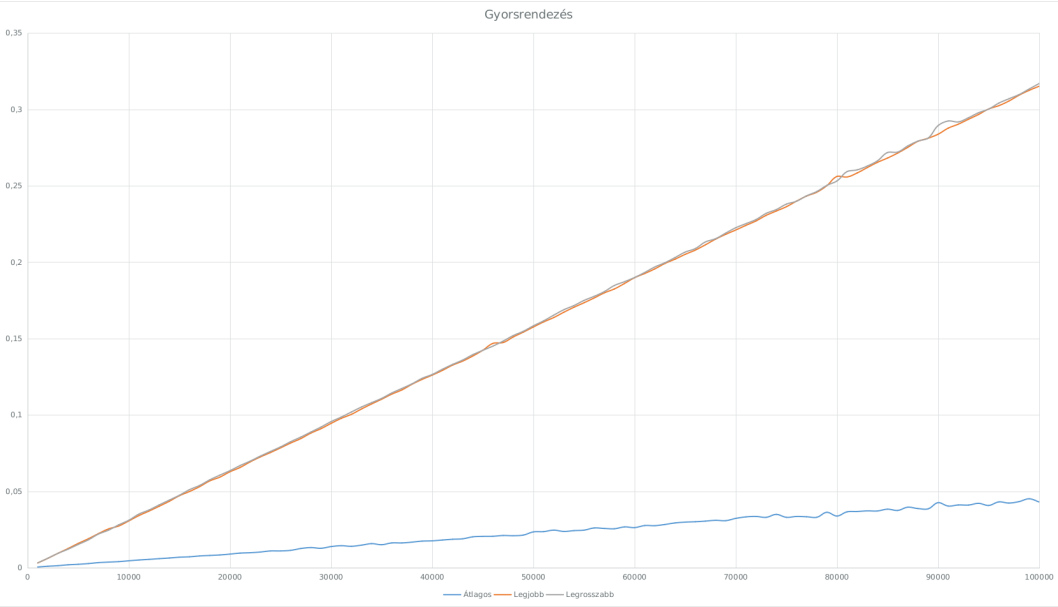
```
1  def quickSort(arr):
2          less = []
3          pivotList = []
4          more = []
5          if(len(arr) <= 1):
6                  return arr
7          else:
8                  pivot = arr[0]
9                  for i in arr:
10                         if(i < pivot):
11                                 less.append(i)
12                         elif(i > pivot):
13                                 more.append(i)
14                         else:
15                                 pivotList.append(i)
16                 less = quickSort(le      ss)
17                 more = quickSort(more)
18                 return less + pivotList + more
```

Gyorsrendezés
Összehasonlítás.



Mivel alapvetően egy igen gyors futási idejű algoritmusról van szó, növeljük a mintavétel határát, és az átlaghoz szükséges minták számát is. Vegyük 20 minta átlagát, és nézzük az algoritmus futási idejét 100 ezer elemszám esetén, milyen relációban állnak egymáshoz a különböző esetek.

Gyorsrendezés
Összehasonlítás.



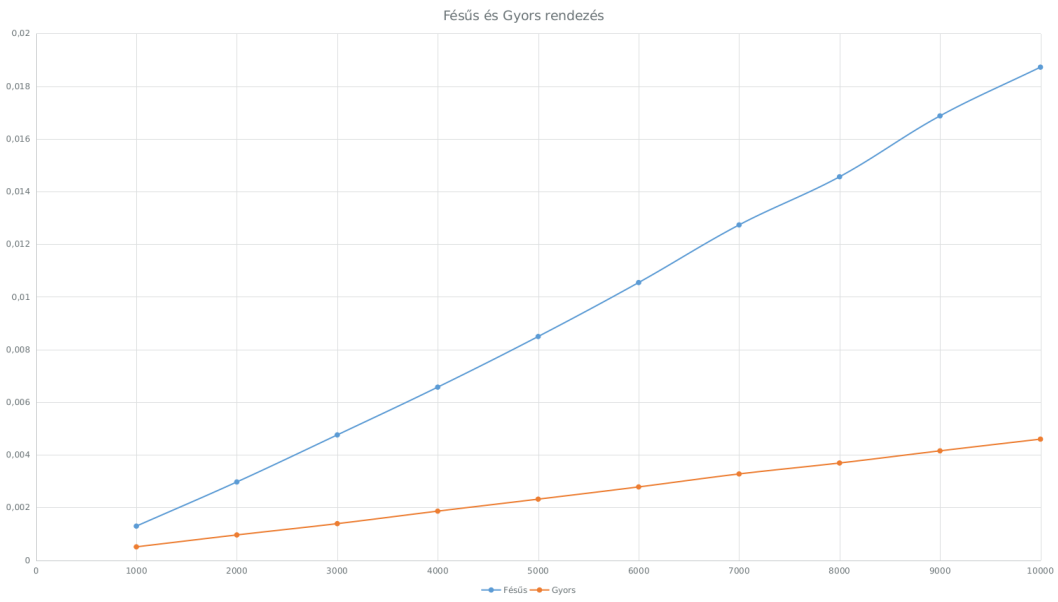
Az első diagram alapján azt mondtam volna, hogy legjobb eset, egyre inkább eltávolodik a legrosszabb esettől. Mindenképp érdekes kiemelni, hogy az átlagos eset futási ideje elképesztően alacsony a vélt legjobb, vagy legrosszabb esethez képest. Ránézve viszont a második diagramra, nagyobb elemszámra látható, hogy a legjobb és legrosszabb eset teljesen összefolynak. Persze figyelembe kell vennünk, hogy a legmagasabb elemszámra is, csupán 0,3 másodperc telt el a tényleges legrosszabb esetben, a legjobban pedig még százezer hosszú sorozatnál is csupán a másodperc huszad része. Az egyetlen olyan rendező algoritmusunk aminek időbonyolultsága $O(n\ log\ n)$

2.9. Összefoglalás

A leghatékonyabb algoritmusaink egyértelműen a fésűs, és a gyors rendezés. Nézzük, hogyan viszonyulnak egymáshoz.

Végső összehasonlítás

Fésűs és Gyors rendezés.



Ugyan a Fésűs rendezés igen hatékony, még mindig lineáris időbonyolultsággal rendelkezik, így egyértelmű hogy melyik a hatékonyabb algoritmus.

2.9.1. Záró gondolat

Látni, hogy a különböző algoritmusok különféleképpen reagálnak bizonyos helyzetekre, és hogy nem feltétlenül a legnagyobb probléma egy visszafelé rendezett sorozat, igen tanulságos. Látni, hogy bizonyos algoritmusok, mennyire rosszul teljesítenek, már alapból rendezett sorozatoknál, felveti a kérdést, hogy ezeknél az algoritmusoknál, érdemes-e, egy vizsgáló függvényt alkalmazni, ami eldönti egy sorozatról, hogy az rendezett-e. Hiszen feleslegesen indítunk el egy rendező algoritmust, egy rendezett sorozatra. És rengeteg időt pocsékolunk el egy rendezésre, ami valójában érdemi munkát nem végez, ebben az esetben. Természetesen ha a sorozatunk véletlen elemekkel van feltöltve, a rendezettség lehetősége alapból olyan kicsi, hogy nincs értelme számításba venni. Viszont, olyan algoritmusoknál, ahol alapból rendezett sorozatra, a másodperc törtrésze alatt végez az algoritmus, nincs értelme időt, és erőforrást pazarolni arra, hogy megpróbáljuk detektálni egy sorozat rendezettségét, a rendezési kísérlet előtt, így fel sem merül a kérdés, hogy van-e bármilyen értelme.