Rendezési algoritmusok áttekintése

Szakdolgozat

Írta: **Dobrádi Anna**

Matematika BSc szak – elemző szakirány

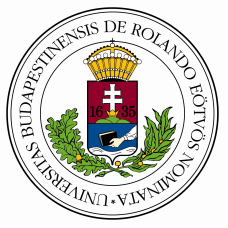
Témavezető:

Dr. Tichler Krisztián

adjunktus

Algoritmusok és Alkalmazásaik Tanszék

Eötvös Loránd Tudományegyetem, Informatika Kar



Eötvös Loránd Tudományegyetem
Természettudományi Kar
2017

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek Tichler Krisztiánnak, amiért elvállalta a témavezetésem és Fekete István tanár úrnak, aki önzetlenül segített, mikor nagy szükségem volt rá.

Továbbá szeretném megköszönni a családomnak a rengeteg támogatást és biztatást amit tőlük kaptam végig az egyetemi éveim alatt, és köszönettel tartozom újdonsült kollégáimnak, amiért segítettek és bátorítottak, külön kiemelve Levit, aki rendszeres kérdéseivel sosem hagyta nyugodni a lelkiismeretemet.

Tartalomjegyzék

Áttekintés

Dolgozatomban a rendező algoritmusokkal fogok foglalkozni, amelyek egy tetszőleges halmaz elemeit rendezik sorba, valamilyen rendszer szerint. Az első fejezetben bemutatásra kerülnek a rendezéshez leggyakrabban használt adatszerkezeteket és ezek műveleteit. Ezt követően formalizáljuk a rendezés fogalmát, és rendszerezzük, hogy milyen szempontok szerint lehet a rendező algoritmusokat csoportokba sorolni. A második fejezetben ezen kívül kimondjuk és bebizonyítjuk az összehasonlító algoritmusok műveletigényére vonatkozó tételeket. A következő részekben bemutatunk néhány összehasonlító, nem összehasonlító és rendező hálózat alapú algortimust, valamint megvizsgáljuk őket a műveletigényük szempontjából. Minden bemutatott algoritmushoz struktogram is készült, amelyen keresztül könnyebben elválaszthatóak az egymást követő lépések, ehhez felhasználom a Lőrentey Károly által készített LaTeX struktogram szerkesztő stílust, amely az alábbi oldalon érhető el: `http://aszt.inf.elte.hu/~lorentey/mirror/downloads/stuki/`. Végül a bemutatott algoritmusok egy részét a gyakorlatban is megvizsgáljuk a műveletigényük szempontjából. Ezek a Matlabban készített forráskódok megtalálhatóak a dolgozat végén.

1. Bevezetés

Algoritmus alatt egy olyan utasítássorozatot értünk, amely egy adott probléma megoldására alkalmas, legyen az a vacsora elkészítése, két szám legnagyobb közös osztójának meghatározása, vagy akár egy úticélhoz való eljutás. Minden algoritmus előre meghatározott, elemi lépésekből épül fel, amelyek maguk is lehetnek külön elkészített algoritmusok.

Bár a fenti példákból látható, hogy az élet minden területén használunk algoritmusokat, elsősorban mégis a matematikában és az informatikában hangsúlyozzuk ki és vizsgáljuk őket. Vizsgálatukhoz törekednünk kell a lehető legkevésbé nyelvspecifikus, minél formalizáltabb felírásra, hiszen így lehet őket matematikai szempontból elemezni. Az algoritmusokat vizsgáljuk a lépésszámuk és a tárhelyigényük szempontjából.

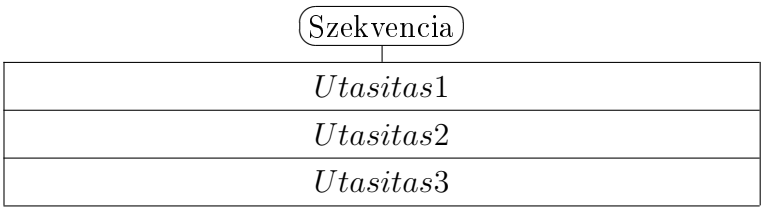
2. Ábrázolások

2.1. Struktogram

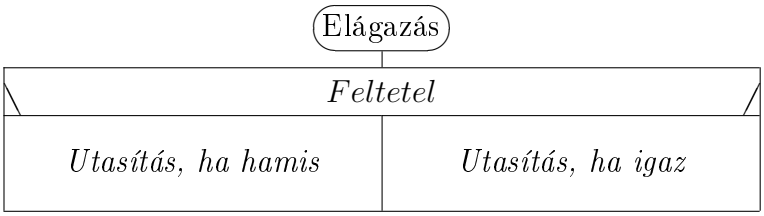
A struktogram a számítógépes algoritmusok egy olyan általános nyelven, egy egységként és meghatározott keretek között történő leírása, amely könnyen olvasható és bármely programozási nyelvre egyértelműen átkódolható.

Elemei

- Szekvencia: utasítások egymás után végrehajtandó sorozata.

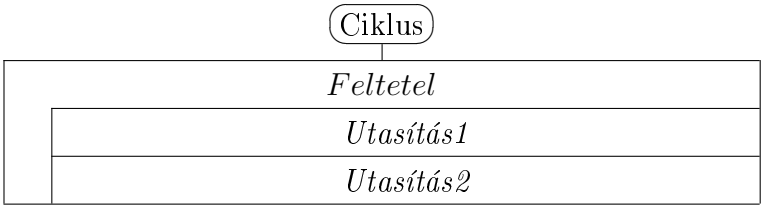


- Elágazás: feltétel, amelynek hamis eredményeként a bal oldali, igaz eredményeként a jobb oldali utasítások hajtódnak végre.



- Többágú elágazás: olyan feltételrendszer, amelyek közül mindig csak az egyik teljesülhet.

- Ciklus: valamilyen feltétel teljesüléséig ismétlődő utasítások sorozata



2.2. Adatszerkezetek

Adatok reprenzentálására számtalan adattípust alkalmazhatunk, például tömböt, sort, vermet, listát, vagy bináris fát; mindegyiknek vannak előnyei és

hátrányai, például a rajtuk elvégezhető műveletek, vagy abennük tárolt adatok elérhetősége szempontjából. Az általam bemutatott rendező algoritmusok jellemzően tömböket, listákat, és bináris fákat fognak használni, ezért a következőkben bemutatom ezeknek a típusait, formális leírását, valamint a rajtuk elvégezhető műveleteket.

Listák

1. Definíció (Lista). *A lista egy olyan absztrakt lineáris adatszerkezet, amely véges sok elem rendezett tárolására alkalmas. Megkülönböztethetjük őket aszerint, hogy van-e fejelemük, ciklikusak-e, és hogy egy, vagy két irányba bejárhatóak.*

2. Definíció (Láncolt lista). *Az egyszeresen láncolt listák olyan listák, amelyek elemei az adaton kívül a következő elemre mutató pointert is tárolnak. A kétszeresen láncolt listák elemei is rendelkeznek az előbbi tulajdonsággal, ezen felül még van a megelőző elemre vonatkozó mutatójuk is.*

Fák

3. Definíció (Fa). *Adatszerkezetként fának nevezzük egy csomópontok élekkel összekötött halmazát, ha létezik egy kitüntetett gyökér pont, minden a gyökértől különböző pont pontosan egy éllel kapcsolódik a szülőjéhez, és összefüggenek a pontok, azaz bármely nem gyökérpontból kiindulva a szülőkön keresztül a gyökérhez juthatunk.*

Jellemzői:

- Pont mélysége: A gyökértől a ponthoz vezető út hossza.
- Fa magassága: A gyökér és a legtávolabbi levél távolsága.
- Pont szintje: A fa magasságának és a pont mélységének különbsége.

4. Definíció (Bináris fa). *Bináris fák azok a fák, amelyekben minden pontnak legfeljebb két leszármazottja van.*

5. Definíció (Balra tömörített bináris fa). *Balra tömörítettnek nevezzük azokat a bináris fákat, amelyekben a levelek balról-jobbra hézagmentesen helyezkednek el.*

6. Definíció (Kupac). *Kupacnak nevezzük azokat a balra tömörített bináris fákat, amelyeknek minden belső pontjában lévő érték nagyobb, vagy egyenlő, mint a gyerekeiben lévő érték, és minden pontnak legfeljebb 2 leszármazottja van.*

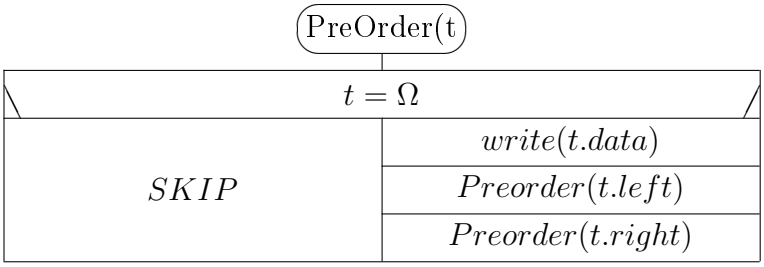
Kupacműveletek:

- Empty(Heap): egy üres kupac létrehozása,
  műveletigénye: $\Theta(1)$
- IsEmpty(Heap): logikai; ellenőrzi, hogy van-e elem a kupacban,
  műveletigénye: $\Theta(1)$
- First(Heap): Element, vagy NIL; megadja a kupac első elemét,
  műveletigénye: $\Theta(1)$
- Insert(Heap, Element): Kupac, vagy NIL; beszúr egy új elemet a kupacba,
  műveletigénye: $\Theta(1)$
- TakeOut(Heap, Element): (Heap $\times$ Element) vagy NIL; kivesz egy adott elemet a kupacból,
  műveletigénye: $\Theta(1)$
- MoveUp(Heap, Index): Heap; az adott indexű elemet megcseréli a szülőjével,
  műveletigénye: $\Theta(\log(n))$
- MoveDown(Heap, Index): Heap; az adott indexű elemet megcseréli a nagyobb leszármazottjával,
  műveletigénye: $\Theta(\log(n))$

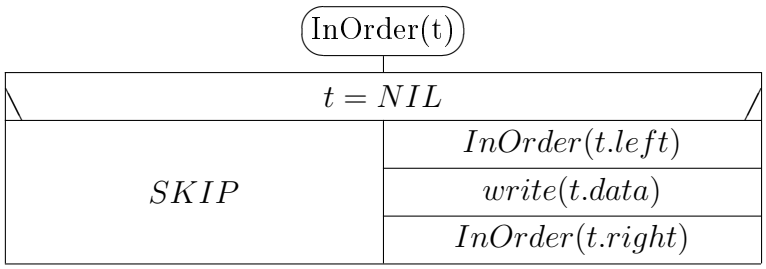
A majdnem teljes bináris fákat algoritmusok implementációjában általában valamilyen tömbszerű adatszerkezetben tároljuk el, így könnyebb implementálni és kezelni is őket, ám ehhez szükség van a fák valamilyen módszerrel történő következetes (minden elemére ugyanúgy lefutó) bejárására. Ha a bináris fa erősen hiányos, akkor pointeresen ábrázoljuk, a pontok adatával, valamint bal- és jobboldali leszármazottaival.

Bináris fák bejárása: [4]

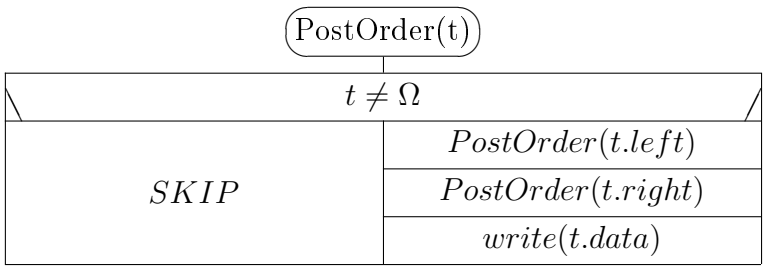
- Preorder:



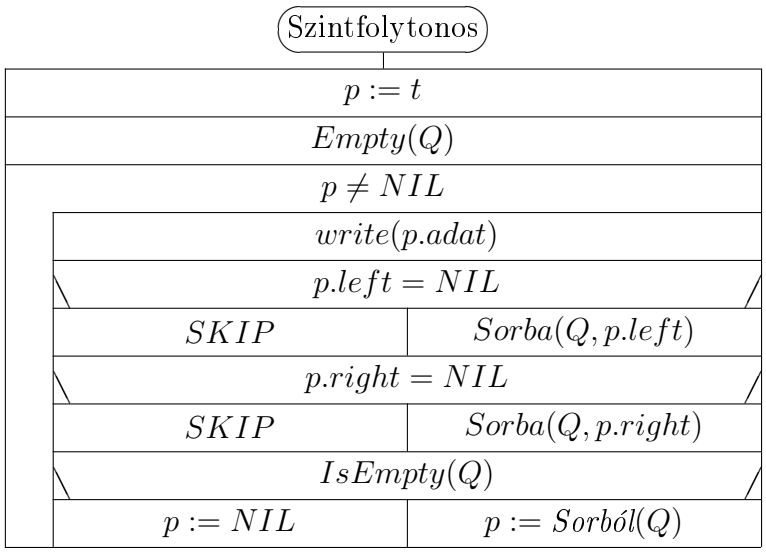
- Inorder:



- PostOrder



- Szintfolytonos(t)



A kupac adatszerkezetet a könnyebb átláthatóság érdekében szokás fa formában ábrázolni, ám algoritmusban történő felhasználásakor a tömbös megvalósítását alkalmazzuk. A két változat közötti ekvivalenciát a kupac szitfolytonos bejárásával lehet belátni, ugyanis a kupac definíciójából következik, hogy a balra tömörítés miatt egyik szinten sem lesz „lyuk”, vagyis üres elem a fában. Egy ilyen módon létrehozott tömbben pontosan dekódolhatjuk, hogy

az egyes elemek hogyan kapcsolódnak egymáshoz, hiszen minden elemre a baloldali leszármazott indexe a szülő indexének kétszerese, jobboldali leszármazotté pedig a szülő indexének kétszerese+1, vagyis ha a szülő a k-adik elem a tömbben, akkor a leszármazottai biztosan a $2k$ és $2k+1$-edik elemek lesznek, míg az elem szülőjének indexe minden esetben az elem indexének alsó egészrésze lesz.

Prioritásos sor

A prioritásos, vagy elsőbbségi sor kilóg a felsorolt adatszerkezetek közül, mert megfeleltethető neki minden olyan struktúra, amelyből a minden lépésben a legnagyobb, vagy meghatározástól függően a legkisebb elemet tudjuk kivenni. Reprezentálására három adatszerkezetet fogok bemutatni, ezek jól használhatóak tömbök hatékony rendezésére.

Rendezetlen tömb: Ha az elsőbbségi sort rendezetlen tömbként ábrázoljuk, akkor mint azt a neve is mutatja, az elemek sorrendjében nincsen logika. Új elem beszúrásakor a tömb végéhez lineárisan hozzáírjuk az új elemet, ez lesz a $PrSorba()$ művelet. A maximális elem tömbből kivételéhez, azaz a $PrSorbol()$ művelethez maximumkereséssel (a $PrMax()$ művelet) megkeressük a legnagyobbat, megcseréljük az utolsóval és kivesszük a tömbből. Rendezetlen tömb prioritásos sorral való rendezése megegyezik a kiválasztó rendezés algoritmusával.

A $PrSorba(n)$ műveletigénye: $\Theta(1)$.
A $PrSorbl(n)$ műveletigénye: $\Theta(n)$.
A $PrMax(n)$ műveletigénye: $\Theta(n)$.

Rendezett tömb: Ebben az esetben feltételezzük, hogy a tömb elemei eleve rendezve vannak. Ekkor új elem hozzátételekor ($PrSorba()$) az elemet beszúrjuk a tömb megfelelő helyére, kivételkor ($PrSorbol()$) pedig lineárisan az utolsót tudjuk kivenni. A $PrMax()$ művelet is az utolsó elemet fogja visszaadni. Rendezett tömb prioritásos sorral való rendezése azonos a beszúró rendezés algoritmusával.

A $PrSorba(n)$ műveletigénye: $\Theta(n)$.
A $PrSorbl(n)$ műveletigénye: $\Theta(1)$.
A $PrMax(n)$ műveletigénye: $\Theta(1)$.

Kupac: Kupacos ábrázolás esetén az új elemet az első üres helyre szúrjuk be, majd a MoveUp() kupacművelettel a helyére léptetjük, ez lesz a $PrSorba()$. A $PrSorbl()$ művelet kiírja a gyökérben lévő elemet, a helyére beírja az utolsó elemet, amit a MoveDown() kupacművelettel a helyére visz. A $PrMax()$ a kupac-tulajdonság miatt a gyökérelem. Ez a rendezés az

előző kettőnek optimalizálása, és kupacok elsőbbségi sorral történő rendezése megegyezik a kupacrendezés algoritmussal.

A $PrSorba(n)$ műveletigénye: $\Theta(\log(n))$.

A $PrSorbl(n)$ műveletigénye: $\Theta(\log(n))$.

A $PrMax(n)$ műveletigénye: $\Theta(1)$.

3. Rendezések

3.1. Formális bevezetés

7. Definíció (Gyenge rendezés). *Legyen A halmaz és legyen $\leq$ egy kétváltozós reláció. Ekkor egy $\leq$ kétváltozós reláció rendezési reláció az A halmazon, ha $\forall a, b \in A$-ra:*

Reflexív, azaz relációban áll önmagával:

$$\forall a \in A\text{-ra } a \leq a. \tag{1}$$

Antiszimmetrikus, azaz ha:

$$\forall a, b \in A\text{-ra } a \leq b \text{ vagy } b \leq a. \tag{2}$$

Tranzitív, azaz ha:

$$\forall a, b, c \in A\text{-ra } a \leq b \text{ és } b \leq c, \text{ akkor } a \leq c. \tag{3}$$

8. Megjegyzés. *Szigorú rendezésről beszélünk, ha $<$ olyan kétváltozós reláció, amelyre teljesül (1)-(3) úgy, hogy az $a < b := a \leq b$ és $a \neq b$ feltétel mellett*

$$a < b \text{ vagy } b < a. \tag{4}$$

9. Definíció (Rendezett halmaz). *Egy $(A, \leq)$ párt rendezett halmaznak nevezünk, ha A nemüres halmaz és $\leq$ egy rendezési reláció az A halmazon.*

3.2. Csoportosíthatóság

Az algoritmusokat, köztük a rendezéseket is az alább definiált szempontok szerint lehet csoportokba sorolni.

10. Definíció (Stabilitás). *Legyen A egy halmaz, és $<$ gyenge rendezés A elemein. Egy rendező algoritmus stabil, ha $\forall i < j$ és $A[i] \sim A[j]$-ből következik, hogy $\pi(i) < \pi(j)$, ahol π a rendező permutáció, melyben a rendezés az $A[i]$ elemet a $\pi(i)$ helyre mozgatja.*

Ez másképpen fogalmazva azt jelenti, hogy az azonos, ekvivalens elemek megtartják az egymáshoz viszonyított helyüket a rendezés után is.

11. Példa (Stabil rendezés). *Buborékrendezés, beszúró rendezés, összefésülő rendezés, leszámoló rendezés, edényrendezés.*

12. Példa (Instabil rendezés). *Gyorsrendezés, kupacrendezés, versenyrendezés, kiválasztásos rendezés.*

Bármely rendező algoritmus stabillá tehető, ám ez tárhely- és műveletigény növekedéssel oldható csak meg, például egy új azonosító oszlop hozzávételével. Ez azt jelenti, hogy ebben az esetben pontosan $O(n)$ extra tárhelyet fog még elfoglalni az adat.

Programozási szempontból a legtöbb nyelveen van lehetőség eldönteni, hogy stabil, vagy nem stabil rendezést szeretnénk-e alkalmazni (pl. c++: stable_sort() - sort()). Ez azért fontos szempont, mert a választástól függően változhat az algoritmus futásideje és tárhelyigénye.

13. Definíció (Determinisztikus algoritmus). *Egy algoritmus determinisztikus, ha ugyanarra a bemenetre mindig ugyanazt az eredményt adja, és minden időpontban egyértelműen adott a következő lépés.*

14. Definíció (Nemdeterminisztikus algoritmus). *Nemdeterminisztikusnak akkor nevezünk egy algoritmust, ha egy adott bemenetre több lehetséges eredményt is visszaadhat, és egy állapotból többféleképpen is lehetséges a továbblépés.*

15. Definíció (Randomizált algoritmus). *Olyan algoritmus, amelynek lépéseiben a véletlen meghatározó szerepet játszik. Két változata ismert, a Las Vegas típusú és a Monte Carlo típusú algoritmus.*

3.3. Rendezések műveletigénye

Egy algoritmus műveletigényét csak közelítéssel szokás meghatározni, mert ha minden műveletet figyelembe vennénk, akkor lényegesen megnőne a számítás bonyolultsága. A kiszámításhoz általában legfeljebb néhány meghatározó, „domináns” műveletet választunk ki, és azt vesszük figyelembe, hogy ez a művelet hányszor hajtódik végre az algoritmus futása során. A domináns művelet kiválasztásakor fontos szempot a többihez képest mért futásideje, illetve a végrehajtásainak száma az algoritmusban. Mivel minden gépen más a műveletek tényleges futásideje, ezért ezzel a nagyságrendi közelítéssel már jól jellemezhetőek és összehasonlíthatóak az algoritmusok. Műveletigény vizsgálatkor az alsó és a felső korlátot, valamint az átlagos esetet vesszük figyelembe. Alsó korlátnak nevezzük azt az értéket, amelynél kevesebb művelettel nem végezhető el a rendezés. A felső korlát pedig az a művelet mennyiség, amely a lehető legrosszabbul rendezett kiindulási tömb rendezéséhez kell.

16. Definíció. *Egy összehasonlító rendezési algoritmus bemenetnek egy n hosszú $[a_1, a_2, \ldots, a_n]$ tömböt kap, az elemekről információt csak a páronkénti összehasonlításukkal tud szerezni, ez az összehasonlítás (tehát hogy $a_i \leq a_j$ igaz-e) egy „igen", vagy egy „nem" válasszal tud visszatérni. Az összehasonlítás művelete egy időegységig tart, ez fogja meghatározni a lépésszámot. Ezen kívül az algoritmusban szerepel még egy újrarendezés lépés is, amely az összehasonlítás eredményétől függően átrendezi a tömböt, ám ezt nem vesszük figyelembe a lépésszám számításnál, mert legfeljebb annyiszor következhet be, mint az összehasonlítás. Az algoritmus kimenete mindig a bemeneti tömb elemeinek egy permutációja, amelyben az elemek már sorban vannak.*

17. Tétel. *Az összehasonlító determinisztikus rendezések műveletigényének alsó korlátja egy n elemű halmazon a legrosszabb esetben $\Omega(n \cdot \log n)$. Speciálisan: $\forall$ A összehasonlító determinisztikus rendezési algoritmus esetén, $\forall n \geq 2$-re $\exists$ egy n nagyságú I bemenet, amelyet A legkevesebb $log_2(n!) = \Omega(n \cdot \log n)$ összehasonlítással rendez.*

Bizonyítás. [2] Tegyük fel, hogy az algoritmus bemenete az $1, 2, \ldots, n$ elemek egy tetszőleges permutációja. Mivel általános esetre bizonyítjuk az állítást, ezért az algoritmus a Barkochba játékot alkalmazza a bemenetre, ennek lényege, hogy eldöntendő kérdésekkel szűkítjük a lehetséges megoldások halmazát, amíg elérjük a választ. Tudjuk, hogy:

1. két különböző bemeneti sorrendet nem lehet helyesen rendezni ugyanazzal a kérdéssorozattal, és

2. $n!$ különböző lehetséges sorrendje van az elemeknek, tehát ennyi különféle input létezik.

Tegyük fel, hogy I_1 és I_2 két különböző kezdeti rendezése a számoknak, amelyek az algoritmus által eddig elvégzett összehasonlításokban megegyeznek. Ekkor a rendezés még nem fejeződhetett be, hiszen ekkor I_1 és I_2 bemenetre is azonos lépésekkel jutnánk ugyanarra a végeredményre. Emiatt az algoritmushoz ki kell jelölni, hogy az $1, \ldots, n$ sorozat permutációi közül melyik volt a megadott bemenet. Legyen S az összes bemeneti sorrendek gyűjteménye, ahol az eddigi összehasonlításokra adott válaszok megegyeznek, tehát kezdetben S az összes $n!$ féle lehetséges bemenet. Egy következő összehasonlítás S-et kettébontja, egy olyan csoportra, ahol a következő válasz „igen", és egy olyanra, ahol ez a válasz „nem". Tegyük fel, hogy minden összehasonlításra az a válasz, amely a nagyobb csoportba sorolja S-et, tehát a bontás után a nagyobbik megmaradó részt vesszük figyelembe. Ekkor minden összehasonlítás legfeljebb felére csökkenti S méretét. Mivel S eredeti mérete $n!$, és az algoritmus futásának végére $|S|$-ről 1-re kell csökkenie, ezért legalább $log_2(n!)$

darab összehasonlítást kell elvégezni, amelyből már következni fog az állítás:

$$\begin{aligned} log_2(n!) &= log_2(n) + log_2(n-1) + \ldots + log_2(2) \\ &= \Omega(n \cdot logn). \end{aligned}$$

■

18. Példa. *$n = 3$-ra az összes lehetséges bemeneti sorrend (S):*

$$\{123\}, \{132\}, \{231\}, \{213\}, \{321\}, \{312\}.$$

Legyen az első lépés az első két elem összehasonlítása. A ha a válasz az, hogy $A[2] > A[1]$, akkor a lehetséges inputok a

$$\{123\}, \{132\}, \{231\}$$

maradnak. A következő lépésben összehasonlítva $A[3]$-at és $A[2]$-őt, az $A[2] > A[3]$ döntés következménye lesz a nagyobb csoport:

$$\{132\}, \{231\}$$

Ekkor már csak egy összehasonlítást kell végezni, hogy megkapjuk a kiinduló bementeti tömböt.[2]

19. Definíció (Döntési fa). *Olyan bináris fák, amelyek tökéletesek, azaz minden pontjuknak pontosan 0 vagy 2 leszármazottja van; minden belső pont egy eldöntendő kérdést tartalmaz, a levelekben pedig az eredmények találhatóak.*

20. Definíció (Kiegyensúlyozott bináris fa). *Egy bináris keresőfa kiegyensúlyozott, ha a legnagyobb és a legkisebb mélységű leveleinek a távolsága legfeljebb 1, azaz a levelek két szomszédos szinten helyezkednek el.*

21. Megjegyzés. *A bináris fa **tökéletesen kiegyensúlyozott**, ha minden levele azonos szinten van. Ha van olyan levele, amelyik más szinten van, mint a többi, akkor **majdnem teljesen kiegyensúlyozott** bináris fának nevezzük.*

22. Tétel. *Az összehasonlító determinisztikus rendezések esetében az összehasonlítások száma egy n elemű halmazon az átlagos esetben legalább $\lfloor log_2(n!) \rfloor$, azaz ebben az esetben is $\Omega(n \cdot logn)$ lesz a rendezések műveletigénye.*

Bizonyítás. [2] Az állítást döntési fa segitségével fogjuk belátni. Építsük fel a döntési fát úgy, hogy minden lehetséges válasz-sorozatot figyelembe veszünk a bemenet egy sorrendje alapján. Ekkor a fának minden levele megfeleltethető a bemenet egy permutációjának. A levél mélysége az adott permutáción végzett összehasonlítások számát mutatja. Ha a fa teljesen kiegyensúlyozott, akkor minden levél mélysége $\lceil log_2(n!) \rceil$ vagy $\lfloor log_2(n!) \rfloor$ lesz, amivel kész vagyunk. A tétel bizonyításához be kell látnunk a következő állitást:

23. Állítás. *Adott levélszámú döntési fák közül a teljesen kiegyensúkyozott bináris fák csúcsainak a legkisebb az átlagos mélysége, azaz csak ez minimalizálhatja a csúcsok átlagos mélységét.*

Vegyük egy nem kiegyensúlyozott fa két szomszédos, legmélyebb levelét, és rakjuk át őket a legkisebb mélységű levél leszármazottainak. Mivel ezeknek a különbsége minimum 2 (különben kiegyensúlyozott fa lenne), ezért ez a művelet csökkenti a fa leveleinek átlagos mélységét, hiszen ha a legkisebb mélység d, a legnagyobb pedig D, akkor az előző művelettel két levelet raktunk a $d+1$-edik szintre, és egy levelet adtunk a $D-1$-edikre.

Mivel minden kiegyensúlyozatlan fa módosítható úgy, hogy csökkenjen az átlagos mélysége, ezért egy ilyen fa nem minimalizálhatja az átlagos mélységet, és ebből következik, hogy a legkisebb átlagos mélységű fának kiegyensúlyozottnak kell lennie. ■

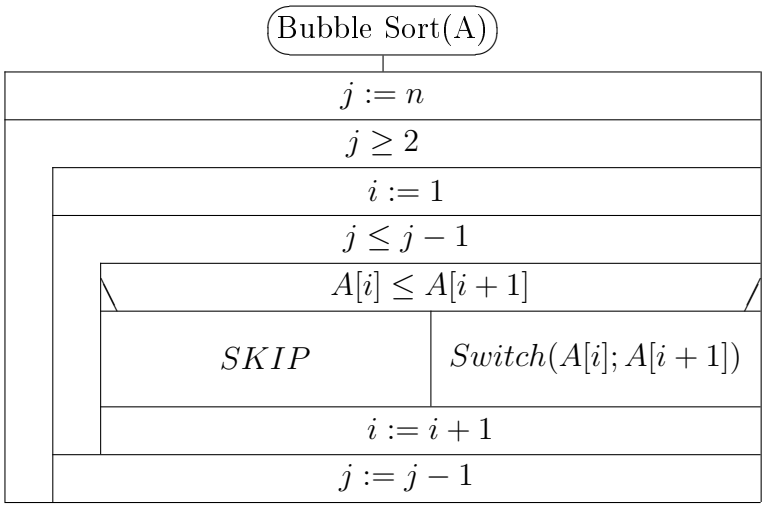
4. Összehasonlító rendezések

A következő rendezési algoritmusok megegyeznek, hogy a rendezendő elemeket közvetlenül hasonlítjuk össze egymással, ez alapján kapjuk meg a sorrendjüket. Közös tulajdonságuk még az előzőekben belátott tétel, azaz hogy egy n méretű input rendezéséhez szükséges lépések számának alsó korlátja $n \cdot \log n$.

4.1. Buborékrendezés [Bubble sort]

Az algoritmus

A buborékrendezés talán az egyik legismertebb rendezés, mert könnyen megérthető és egyszerűen programozható. Az összehasonlító rendezések családjába tartozik, mert futása során két szomszédos elemet hasonlítunk össze. Nevét is innen kapta, hiszen a kezdeti állapotból sorban „felbuborékoltatjuk" a nagyobb elemeket. Az algoritmus működése: a kezdeti állapotban nincs rendezettség, kiindulunk egy n hosszú tömbből. Első lépésként az első két elemet hasonlítjuk össze. Ha a kisebb indexű nagyobb, mint a másik, akkor megcseréljük őket és továbblépünk, ha nem akkor csak továbbmegyünk és a második és harmadik elemmel végezzük el az összehasonlítást. Mire elérünk a tömb végébe, biztosan a legnagyobb elem fog az utolsó helyre kerülni. Ezután újra kezdjük az összehasonlításokat a tömb elejéről, ám már csak egy $n-1$-es tömböt kell vizsgálni, hiszen az utolsó elem már a helyén van.



Műveletigény[4]

24. Állítás. *Az algoritmus műveletigénye:*

- *Legrosszabb esetben:* $\Theta(n^2)$.
- *Átlagos esetben:* $\Theta(n^2)$.

25. Definíció (Inverziószám). *Egy permutációban két elem inverzióban áll egymással, ha a nagyobb megelőzi a kisebbet. Az összes ilyen felcserélt elempárok száma a permutáció inverziószáma.*

26. Megjegyzés. *Két szomszédos elem cseréjekor az inverziószám pontosan eggyel változik.*

27. Megjegyzés. *Két elem az elemek permutációja és a permutáció inverze közül pontosan az egyikben állhat csak inverzióban*

Bizonyítás. Az összehasonlítások száma:

$$OHas(n) = (n-1) + (n-2) + \ldots + 1 = \tag{5}$$

$$= \frac{n \cdot (n-1)}{2} = \Theta(n^2) - \Theta(n) = \Theta(n^2) \tag{6}$$

A cserék száma megegyezik az inverziószámmal:

$$Cs(n) = inv(n) \tag{7}$$

A legrosszabb esetben, azaz ha a tömb éppen fordított sorrendben van:

$$MaxCs(n) = \frac{n \cdot (n-1)}{2} \tag{8}$$

Az átlagos esetben az átlagos csereszámmal kell számolnunk, azaz minden lehetséges input (n elem permutációja) inverziószámának átlagával:

$$ACs(n) = \frac{1}{n!} \cdot \sum_{p} inv(p) \tag{9}$$

$$ACs(n) = \frac{\sum inv(p) + \sum inv(p^R)}{2n!} = \tag{10}$$

$$= n! \cdot \frac{\binom{n}{2}}{2n!} = \frac{\binom{n}{2}}{2} = \tag{11}$$

$$= \frac{n(n-1)}{4} = \Theta(n^2), \tag{12}$$

ahol $Perm(n)$ az n elem összes lehetséges permutációinak halmaza p^R pedig a permutáció inverze. ■

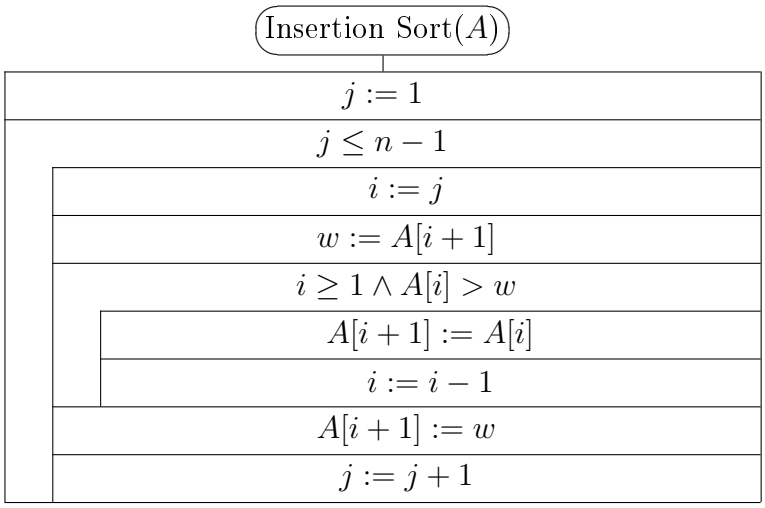
Változata: Koktélrendezés

Kétirányú buborékrendezésnek is nevezik, mert a buborékrendezés algoritmusát oda-vissza végzi el a rendezendő tömbön úgy, hogy mindig az utolsó még nem rendezett elemig fut, így egy teljes ciklus alatt pontosan 2 elem kerül a helyére, tehát minden futás után már csak egy 2-vel kisebb méretű tömböt kell rendezni.

4.2. Beszúró rendezés [Insertion sort]

Az algoritmus

Ez a rendezés futása a második elemtől indul úgy, az aktuális elemet kiemeli egy ideiglenes változóba, majd a korábbi elemeket összehasonlítja a kiemelttel, és jobbra mozgatja, amíg meg nem találja az aktuális elem helyét. Ekkor beszúrja az elemet a kijelölt helyre, annak a rendezettségét megtartva. Ezután továbblép a következő elemre. Az algoritmus addig fut, amíg el nem éri a tömb végét. Mivel a tömb elején a rendezett résztömb minden új elemmel bővülve továbbra is rendezett marad, ezért az utolsó elem beszúrása után a teljes tömb rendezve lesz.



Műveletigény[4]

28. Állítás. *Az algoritmus műveletigénye:*

- *Legrosszabb esetben:* $\Theta(n^2)$.
- *Átlagos esetben:* $\Theta(n^2)$.

Bizonyítás.

$$MaxOHas(n) = \sum_{i=1}^{n-1} i = \frac{(1 + (n-1)) \cdot (n-1)}{2} = \tag{13}$$

$$= \frac{n(n-1)}{2} = \Theta(n^2) \tag{14}$$

Mivel a külső ciklus pontosan $n-1$-szer fut le (ami $\Theta(n)$ idejű), ezért elég a belső ciklust vizsgálni részletesen, mivel ez fogja meghatározni a műveletigényt, tehát:

$$MaxInsert(n) = \Theta(n^2) \tag{15}$$

Az átlagos esetet nem bizonyítjuk. ■

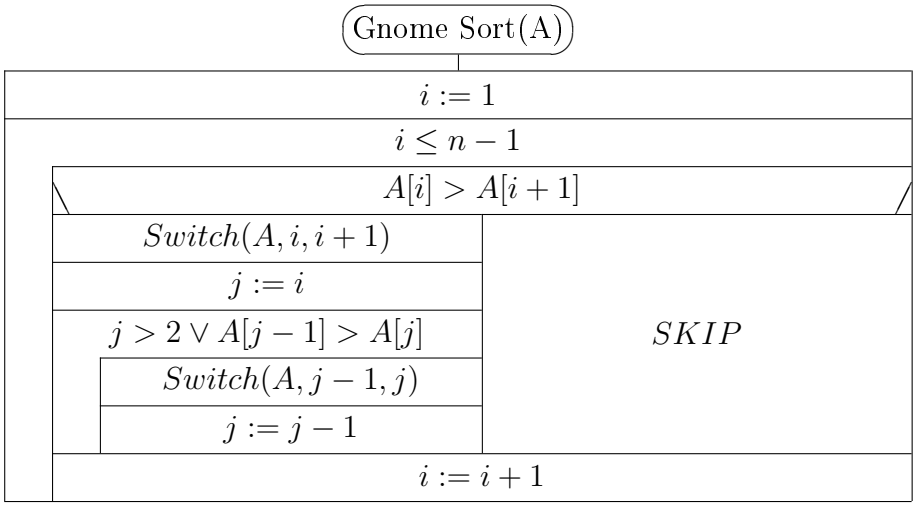
Változata: Beszúró rendezés bináris kereséssel

Ebben az esetben az eredeti rendezést úgy módosítjuk, hogy az új elem helyét nem egyenkénti összehasonlítással, hanem bináris kereséssel határozzuk meg. A változtatással a rendezés átlagos műveletigénye $O(n \cdot \log n)$-re csökken, ám a legrosszabb eseti műveletigénye nem változik.

4.3. Kertitörpe rendezés [Gnome sort]

Az algoritmus

A buborékrendezés és a beszúró rendezés egyfajta kombinációja. Futása során mindig az első elemtől indulva végigmegy a tömb elemein, megkeresve az első két olyan elemet, amelyek rossz sorrendben vannak, és megcseréli őket. Ezzel a cserével kizárólag csak az aktuális, és az azt megelőző elem között alakulhatott ki rossz sorrend, tehát ezeket is megvizsgáljuk, és cserélünk, ha szükséges. Addig megyünk visszafelé a tömbön újravizsgálva az elemeket, ameddig eljutunk egy olyan párhoz, amelynek jó a sorrendje, mert ebből következik, hogy a többi megelőző elem is már rendezve van. Ez után tovább folytatjuk előre a keresést, a következő inverzióban lévő elempárig, és ismét cserélünk. Az algoritmus futása akkor ér véget, ha az előre felé keresésnél nincs inverzióban lévő elempár, mert ekkor az input biztosan rendezett lesz.



Műveletigény

29. Állítás. *Az algoritmus műveletigénye:*

- *Legrosszabb esetben:* $\Theta(n^2)$.
- *Átlagos esetben:* $\Theta(n^2)$.

Bizonyítás. A legrosszabb esetben a tömb elemei éppen fordított sorrendben vannak. Ekkor az első iterációban 1, a másodikban 2, az n-edikben pedig összesen n csere történik, tehát a műveletigény $\Theta(n^2)$.

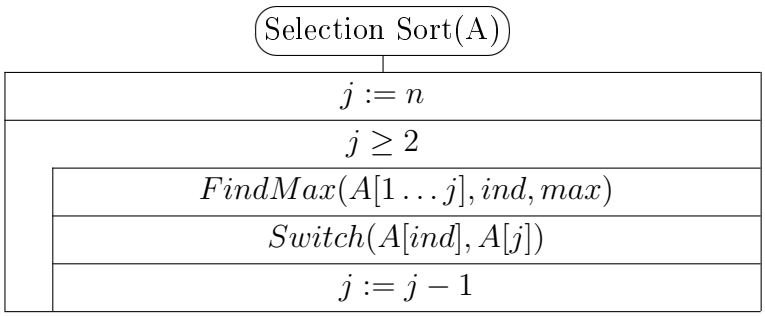
Az átlagos esetet nem bizonyítjuk. ■

4.4. Maximum kiválasztásos rendezés [Selection sort (max or min)]

Az algoritmus

A maximum kiválasztás algoritmusa két résztömbre osztja a bemenetet, az egyik rész a már rendezett, a másik rész pedig a még rendezetlen elemekből áll. Kezdetben a rendezett résztömb üres, a rendezetlen pedig a teljes tömb. Ez után minden lépésben megkeresi a legnagyobb, még rendezetlen elemet, megcseréli a rendezetlen résztömb utolsó elemével, ezzel eggyel növelve a rendezett résztömb méretét. Az algoritmus akkor ér véget, ha a rendezetlen rész üres lesz, a rendezett pedig a teljes, növekvő sorrendbe rendezett tömb.

Az algoritmus minimum kereséssel is futtatható, ebben az esetben a kimenet a csökkenő sorrendbe rendezett tömb lesz.



Műveletigény[4]

30. Állítás. *Az algoritmus műveletigénye:*

- *Legrosszabb esetben:* $\Theta(n^2)$.
- *Átlagos esetben:* $\Theta(n^2)$.

Bizonyítás. $n-1$ lépésben rendez, és minden lépésben maximumot keres és összehasonlít, ezért:

$$OHas(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2) \tag{16}$$

és

$$Cs(n) = n - 1 \tag{17}$$

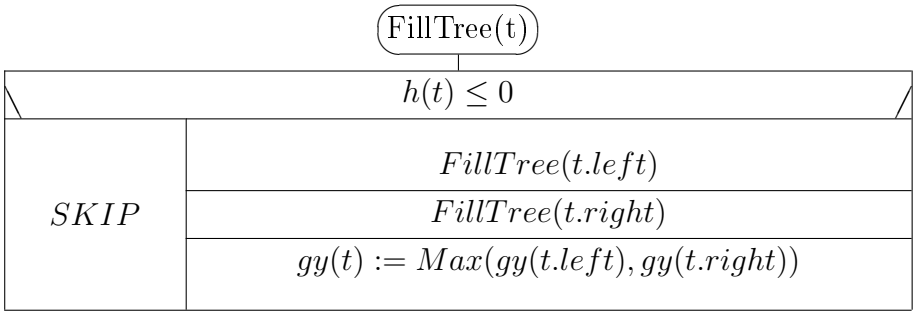
Az átlagos esetet nem bizonyítjuk. ■

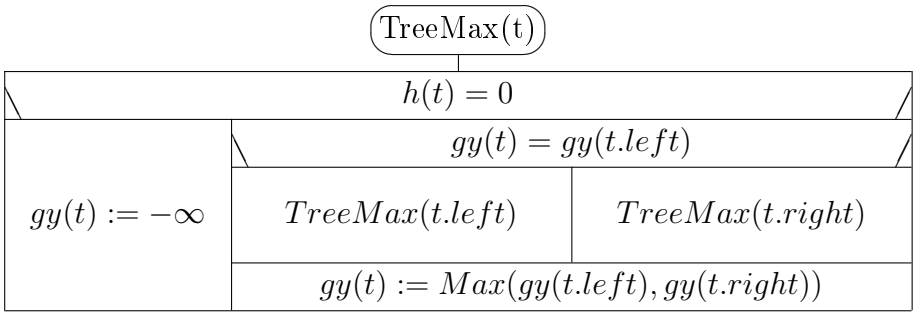
4.5. Versenyrendezés [Tournament sort]

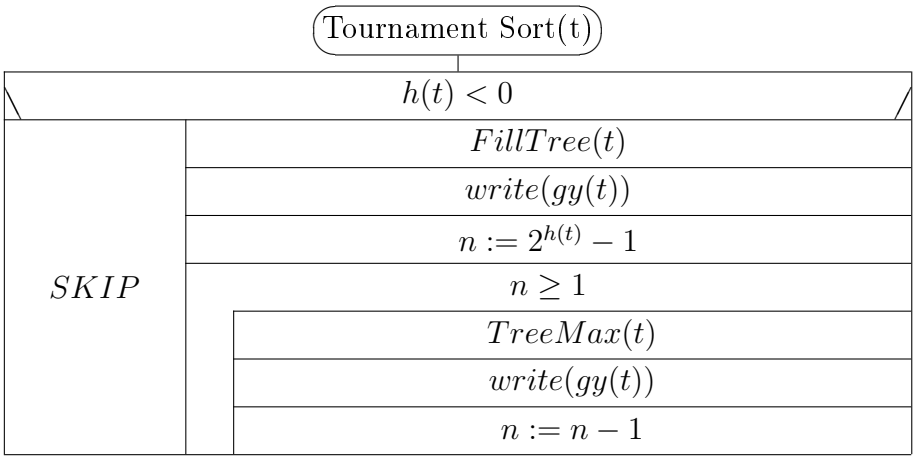
Az algoritmus

A versenyrendezés algoritmusa egy két ágon futó egyenes kieséses verseny szimulációja. Futása során első lépésként a kiinduló rendezőfa (egy teljes bináris fa) leveleiben elhelyezzük a rendezendő elemeket. Ezután páronkénti összehasonlítással „felléptetjük" a nagyobb elemet a következő szintre, majd tovább ismételjük ezt addig, amíg a legnagyobb elem el nem jut a gyökérbe. Ekkor a gyökérben lévő elemet kiírva egy kimeneti tömbbe újra lefuttatja a maximum kiválasztást a „győztes" ágán úgy, hogy a megtalált elem helyére $-\infty$ kerül, így az összehasonlításokkor biztosan nem előzhet meg olyan elemet, amely eredetileg is a fában volt. A versenyrendezés algoritmus addig fut, amíg minden elem bekerül a kimeneti tömbbe.

Az algoritmus három fő részre bontható, ezeket külön struktogramban ábrázolom az átláthatóság érdekében. Az első rész a rendezőfa kezdeti kitöltése, a második a maximum keresés, a harmadik pedig maga az algoritmus, amely felhasználja az első két részalgoritmust.







Műveletigény[4]

31. Állítás. *Az algoritmus műveletigénye:*

- *Legrosszabb esetben:* $\Theta(n \cdot \log n)$.
- *Átlagos esetben:* $\Theta(n \cdot \log n)$.

Bizonyítás. Az algoritmus kezdeti fő művelete (a kezdeti FillTree(t)) $n-1$ összehasonlítást és mozgatást végez a fa kitöltése során. Mivel a fa teljes bináris fa, ezért a magassága $\log_2(n)$, ahol n kettőhatvány. Minden további iterációban ezért $2 \cdot \log_2(n)$ összehasonlítás, és fele ennyi mozgatás történik, mert „oda-vissza" járjuk be a fát.

$$OHas = (n-1) + (n-1) \cdot 2 \cdot \log_2(n) \tag{18}$$

és

$$Mozg = (n-1) + (n-1) \cdot \log_2(n) \tag{19}$$

Mivel a csererendezők alaptétele kimondja, hogy $\Theta(n \cdot \log n)$-nál kevesebb nem lehet a műveletigény és ezt a legrosszabb esetnél már elértük, ebből következik, hogy az átlagos esetben lehet gyorsabb az algoritmus. ■

4.6. Kupacrendezés [Heapsort]

Az algoritmus

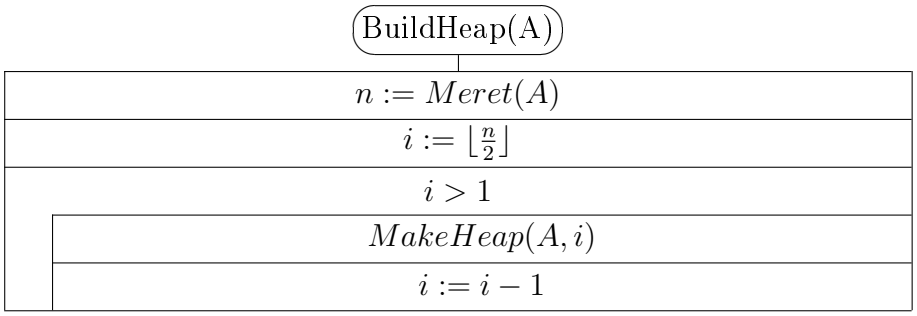
A kupacrendezés algoritmusa két fő részből áll. Az első részben felépíti az inputból a kupacot, a másodikban pedig visszaírja a kimenetbe a kupacból az elemeket. A kupac felépítése az elemek egyenkénti beszúrásával történik az Insert és a MoveUp kupacműveletek segítségével úgy, hogy minden lépésben megmaradjon a kupac-tulajdonság. A beszúrásokat addig folytatja, amíg a bemeneti tömb kiürül. A második lépésben a TakeOut és MoveDown

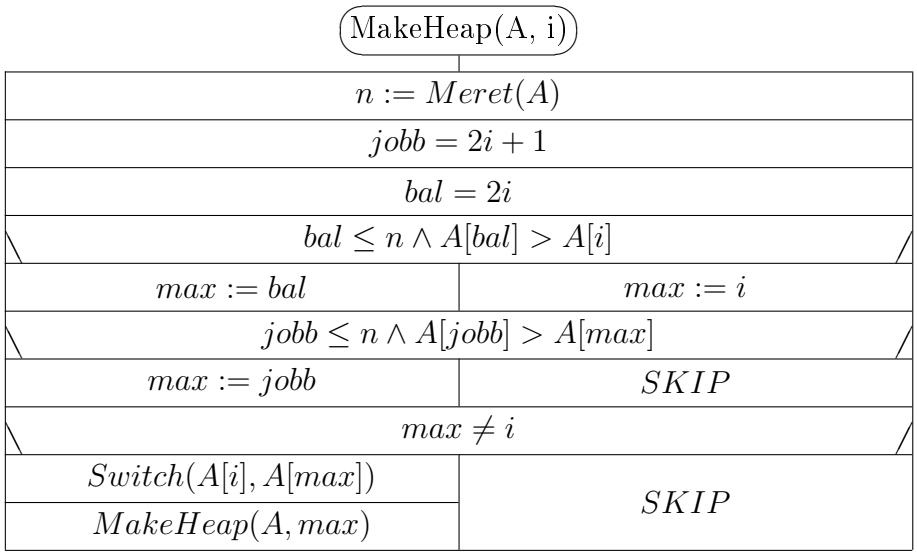
műveletekkel kiírja az aktuális gyökérelemet a kimenetbe, a helyére beszúrja a legszélső levél tartalmát, és rendezi a kupacot. Addig ismétli ezt a lépést, amíg üres kupacot nem kapunk.

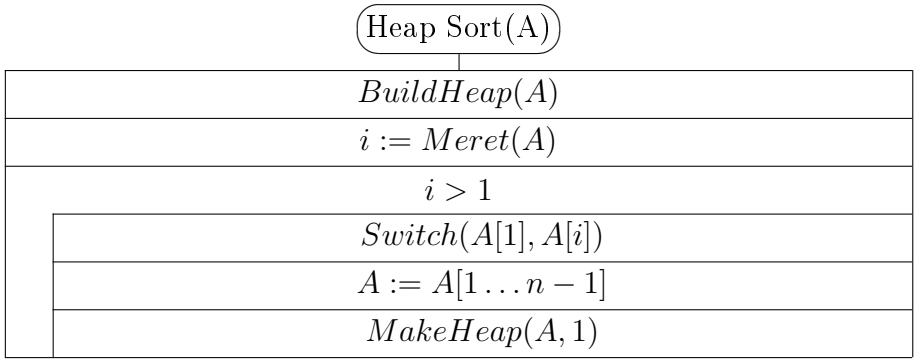
Tömbös megvalósítás

Mivel programozásban egyszerűbb a kupac tömbös megvalósításával dolgozni, ezért a rendezést úgy hajtjuk végre, mintha egy szintfolytonos bejárással létrehozott tömbön végeznénk. Ebben a felfogásban első lépésként úgy rendezzük az inputot, hogy végig megyünk a tömbön, megvizsgáljuk az aktuális elem és a leszármazottjai viszonyát, és ahol nem teljesül a kupac tulajdonság, ott a nagyobbikkal megcseréljük (ez felel meg a *MoveDown* műveletnek). Addig folytatjuk a cseréket, amíg egy kupacnak megfeleltethető tömböt kapunk. A második szakaszban a „gyökérelem", vagyis az éppan aktuális maximum mindig a tömb elején fog elhelyezkedni, míg a „jobb alsó" elem a tömb még rendezetlen részének végén lesz.

A „gyökérelem" kivétele és a „jobb alsó" elem beszúrása megfeleltethető a két elem cseréjének, ami után a tömb rendezetlen részének mérete eggyel csökken. Ha a beszúrt elem kisebb, mint a nagyobbik leszármazottja, akkor átrendezzük a tömböt úgy, hogy ismét megfeleljen a kupac definíciónak. Az újrarendezés után ismét az első és a rendezetlen rész utolsó elemének cseréjével folytatódik az algoritmus. A rendezés akkor ér véget, ha a tömb rendezetlen részének mérete 1, vagyis a teljes tömb rendezve van.







Műveletigény

32. Állítás. *Az algoritmus műveletigénye:*

- *Legrosszabb esetben:* $\Theta(n \cdot \log n)$.
- *Átlagos esetben:* $\Theta(n \cdot \log n)$.

Bizonyítás. A legrosszabb esetben a $BuildHeap$ művelete $\Theta(n)$, a $MakeHeap$ pedig $\Theta(\log n)$, így ekkor a rendezés teljes műveletigénye $\Theta(n \cdot \log n)$. Mivel a csererendezők alaptétele kimondja, hogy ennél nem lehet kevesebb, ezért az átlagos eset műveletigénye is $\Theta(n \cdot \log n)$. ■

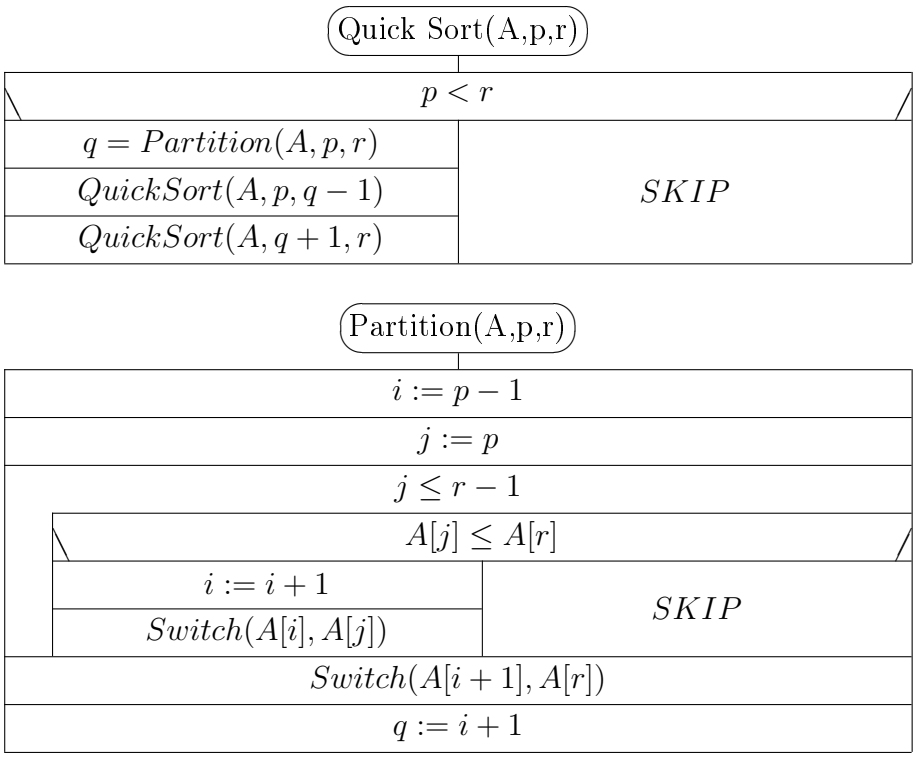
4.7. Gyorsrendezés [Quicksort]

Az algoritmus

Az algoritmus Az „oszd meg és uralkodj” elvet használja, azaz a problémát kisebb méretű, azonos problémákra bontja, amelyek rekurzívan megoldhatók,

majd a megoldásokat egyesíti. Első lépéseként kiválasztunk egy tetszőleges főelemet, ez után a tömböt három részre particionáljuk úgy, hogy a kiválasztott főelemnél kisebb elemeket a főelem elé, a nála nem kisebbeket pedig mögé mozgatjuk. Az így kialakuló három résztömbön – amiket a főelem, a nála kisebb, valamint a nála nagyobb elemek alkotnak – újra futtatjuk a gyorsrendezést.

Ez az algoritmus két fő részre bontható, az egyik maga a rekurzív rendezés, a másik pedig a particionálás.



ahol p, r a résztömb első és utolsó eleme, q pedig kiválasztott a főelem.

Műveletigény

33. Állítás. *Az algoritmus műveletigénye:*

- *Legrosszabb esetben:* $\Theta(n^2)$.
- *Átlagos esetben:* $\Theta(n \cdot \log n)$.

Bizonyítás. A legrosszabb esetben a particionálás művelete úgy bontja fel a tömböt, hogy a főelemen kívüli két rész mérete $n-1$ és 0 lesz minden lépésben. Ekkor a felbontás műveletigénye:

$$P(n) = \Theta(n). \tag{20}$$

Így a futásidő rekurzív képlete a legrosszabb esetben:

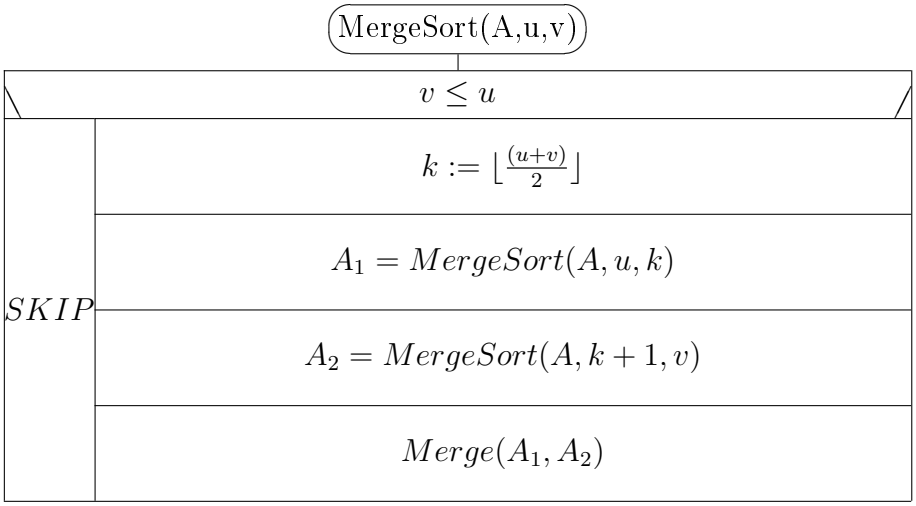
$$S(n) = S(n-1) + \Theta(n) = \sum_{i=1}^{n} \Theta(i) = \Theta(\sum_{i=1}^{n} i) = \Theta(n^2). \qquad (21)$$

Az átlagos esetet nem bizonyítjuk. ■

4.8. Összefésülő rendezés [Merge sort]

Az algoritmus

Ez a rendezés két már rendezett tömböt „fésül" össze úgy, hogy a végeredmény is egy rendezett tömb legyen. Az összefésülés folyamata úgy zajlik, hogy mindkét tömb első elemét összehasonlítjuk, és a kisebbet beírjuk a kimeneti tömb első szabad helyére, majd abból a tömbből, amelyikből ez kikerült, vesszük a következő elemet, és újra elvégezzük az összehasonlítást. Ezt addig folytatjuk, amíg valamelyik kezdeti tömbből el nem fogynak az elemek. Végül másik tömb maradék elemeit is sorban hozzáírjuk az eredményhez, így alakul ki a végleges, rendezett kimeneti tömb. Az algoritmus az összefésülés előtt az inputot rekurzívan kettébontja addig, amíg végül csak rendezett résztömbök lesznek (ez általában az egy elemű résztömb), majd ezeken végzi el az összefésülést.



Műveletigény

34. Állítás. *Az algoritmus műveletigénye:*

- *Legrosszabb esetben:* $\Theta(n \cdot \log n)$.
- *Átlagos esetben:* $\Theta(n \cdot \log n)$.

Bizonyítás. Az algoritmus három fő részből áll. Ezekből a felosztás $\Theta(1)$ művelet, a végső összefésülés pedig $\Theta(n)$. A rekurzívan hívott *MergeSort* minden lépésben $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ műveletet hajt végre, amelyről belátható, hogy $T(n) = \Theta(n \cdot \log(n))$. ■

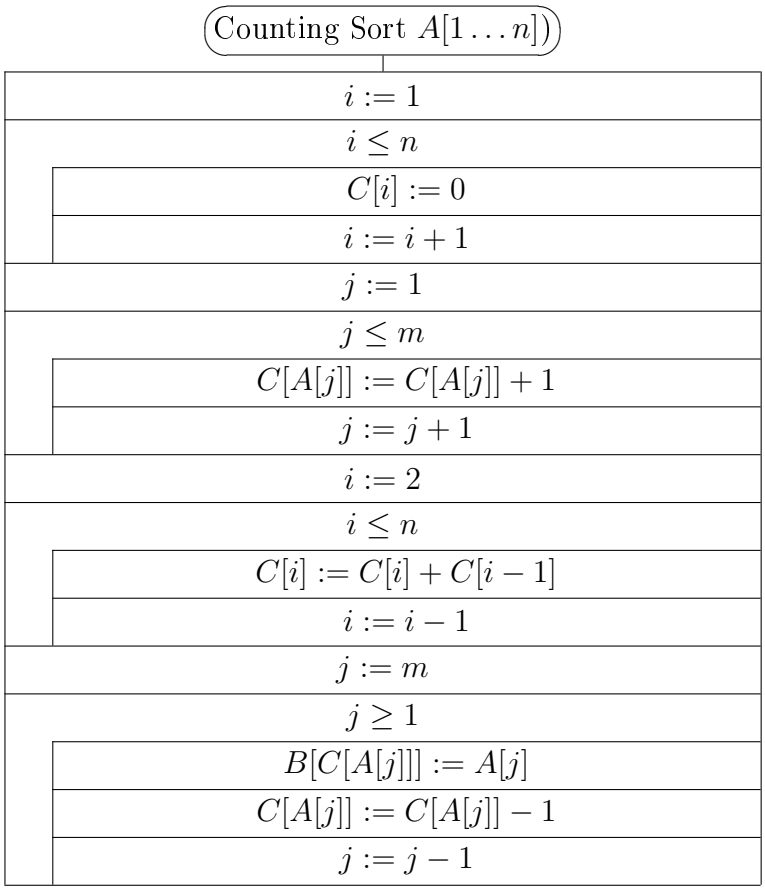
5. Nem összehasonlító rendezések

A most következő rendezések nem úgy adják meg az elemek végleges sorrendjét, hogy összehasonlítják őket, hanem minden elemhez rendelnek egy olyan kulcsot, amely valamilyen közös tulajdonságukon alapul (például szavak esetén a betűk ábécében elfoglalt helye), és ezeknek a kulcsoknak a sorrendje fogja adni az elemek végleges sorrendjét. Fontos, hogy olyan kulcsot találjunk, amely az input minden elemére jellemző. Mivel az input minden elemének valamilyen speciális tulajdonságát használják ki, így a korábbiakban belátott $\Omega(n \cdot \log(n))$-nél gyorsabb algoritmusokat kapunk.

5.1. Leszámoló rendezés [Counting sort]

Az algoritmus

Az algoritmus minden elemre meghatározza a nála kisebb elemek számát, ez alapján tudja az elemet a kimeneti tömb megfelelő helyére tenni. Bemenete 1 és n közötti tetszőleges egész számok egy m méretű halmaza, amelyeket egy A tömbben tárolunk, a kimenete egy B tömb lesz, ennek mérete szintén m lesz. A végrehajtás során szükséges lesz még egy C n méretű segédtömb is, amelyben előbb a benenetben előforduló elemek darabszáma, majd később az ezeknél az elemeknél nemnagyobb elemek darabszáma található meg. A rendezés ez alapján adja meg az elemek végleges helyét a kimeneti tömbben, mivel ekkor ha minden elem különböző, akkor az $A[i]$ elem helyzete pontosan $C[A[i]]$ lesz (hiszen pontosan $C[A[i]]$ elem kisebb nála). Ha megengedjük az egyenlőséget, akkor az algortimus annyiban módosul, hogy amikor egy $A[i]$ elemet beteszünk a kimeneti tömbbe, eggyel csökkentjük $C[A[i]]$ értékét, így a következő azonos elem pontosan elé fog kerülni.[?]



Műveletigény

35. Állítás. *Az algoritmus műveletigénye:*

- *Legrosszabb esetben:* $O(n + m)$.
- *Átlagos esetben:* $O(n + m)$.

Bizonyítás. Az első for ciklusban n hosszú tömböt hozunk létre és minden elemét 0-ra állítjuk, ez n lépést jelent. A második ciklusban végigmegyünk az m hosszú inputon m lépéssel. A harmadik ciklus ismét C elemein fut végig, n lépésben Végül az utolsó ciklus pedig B elemeit tölti fel, szintén m lépéssel.

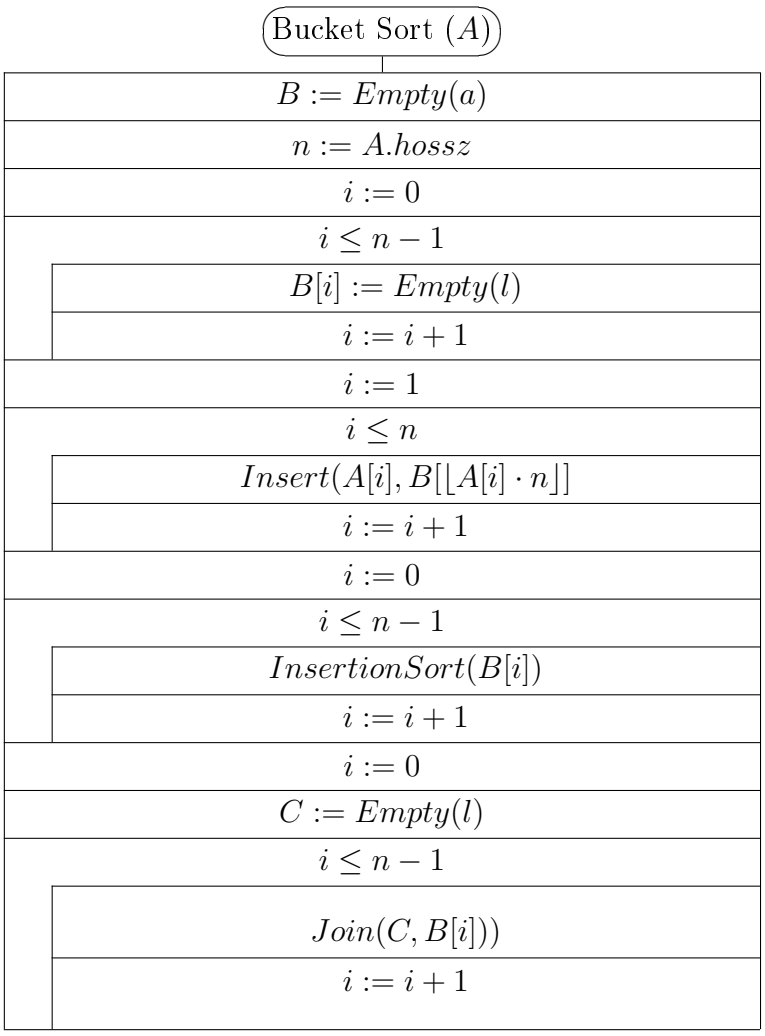
$$Lep(n) = n + m + n + m = 2(n + m) = \Theta(n + m) \tag{22}$$

■

5.2. Edényrendezés [Bucket sort]

Az algoritmus

Az edényrendezés algoritmusában feltételezzük, hogy az input minden eleme független, azonos eloszlású szám a $[0, 1)$ intervallumon. A módszer lényege az, hogy a $[0, 1)$ intervellumot felosztjuk n egyenlő részre (ezek lesznek az edények), és az input minden elemét besoroljuk az $\lfloor elem \cdot n \rfloor$-nek megfelelő edénybe, közben az edény tartalmát beszúró rendezéssel rendezi, majd az így rendezett edényeket összerakja egy kimeneti tömbbe.



Műveletigény

36. Állítás. *Az algoritmus műveletigénye:*

- *Legrosszabb esetben:* $O(n^2)$.
- *Átlagos esetben:* $O(n)$.

Bizonyítás. A legrosszabb esetben minden elem egy edénybe kerül, ekkor a beszúró rendezés műveletigénye miatt a teljes műveletigény $O(n^2)$ lesz. Az átlagos esethez legyen n_i a $B[i]$ edéynbe kerülő elemek számát jelölő valószínűségi változó. Ekkor a futásidő

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2), \tag{23}$$

amelyek várható értéke

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]). \tag{24}$$

Belátható, hogy

$$E[n_i^2] = 2 - \frac{1}{n}. \tag{25}$$

Ezt behelyettesítve kapjuk, hogy

$$\begin{aligned} E[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - \frac{1}{n}) && (26)\\ &= \Theta(n) + n \cdot O(2 - \frac{1}{n}) && (27)\\ &= \Theta(n), && (28) \end{aligned}$$

tehát a rendezés az átlagos esetben lineáris lesz. ■

5.3. Számjegyes rendezés [Radix]

Azonos hosszúságú stringek rendezésére használható, ahol k a szó hossza, d az egy karakteren előforduló lehetséges jegyek, n pedig a bemenő adatok száma.

Az algoritmus[1]

A futás során a karaktereik mentén sorbarendezzük a bemenetet valamilyen stabil rendezéssel. Többféle változata is létezik. Van, amelyik az inputot az első elemtől vizsgálva rendezi úgy, hogy először az első karakterek mentén rendez, majd ezeknek a sorrendjét megtartva a második elemek mentén, és így tovább a bemenetek végéig. Egy másik változata pedig ugyanezzel az elvvel működik, de a stringeket nem elölről, hanem hátulról kezdi rendezni.

Műveletigény

Az algoritmus műveletigénye erősen függ attól hogy milyen stabil rendezést alkalmazunk a szavak belső rendezésére. Ha a bemeneti stringek elemei megfelelnek a feltételnek, akkor például a leszámláló rendezéssel egy lépésben $\Theta(n+d)$ a műveletigény. Mivel minden karakteren végrehajtjuk a rendezést, ezért a teljes műveletigény ebben az esetben $\Theta(k \cdot (n+d))$ lesz.

6. Rendező hálózatok[1]

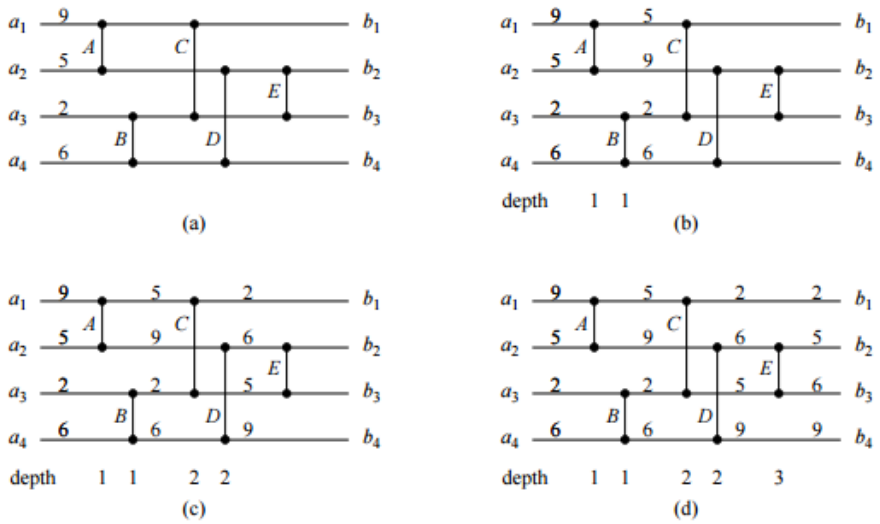
Az eddig bemutatott csererendezők esetében minden lépésben pontosan egy összehasonlítás (és csere) hajtódhatott végre az inputon, ám annak ellenére, hogy ezzel is megfelelő eredményt kapunk, látható, hogy nem a leghatékonyabb eljárás. Ezeknek az algoritmusoknak a javítására jöttek létre a rendező hálózatok, amelyeknek két fontos tulajdonsága van: csak összehasonlító rendezéseket lehet velük javítani, és ellentétben a csererendezőkkel, az összehasonlítások párhuzamosan, azonos időben is megtörténhetnek, ezzel gyorsítva az eljárást. A hálózatok hátránya viszont éppen a párhuzamosíthatóságból következik, ugyanis a felépítésükből látható lesz, hogy legfeljebb egy adott maximális elemszámú inputot képesek rendezni.

6.1. Felépítés és működés

A rendező hálózatok két részből állnak, vezetékekből és komparátorokból. A komparátorok egy ütemben egy páronként független kapukból álló kapurendeszert alkotnak, ezek végzik az input elemeinek összehasonlítását és szükség esetén cseréjét. A kapuk a vezetékek mentén helyezkednek el, a rendezendő bemenet pedig a szintén a vezetékek mentén halad végig a kapukon, ezért korlátozza a vezetékek száma a rendezhető input méretét. Minden kapu egy kétváltozós művelet, amelynek például a bemenete (x, y), a kimenete pedig (x', y'), ahol $x' = \min(x, y), y' = \max(x, y)$. Amikor az input átmegy egy kapurendszeren, csak azokat az elemeket hasonlítja össze, amelyek az adott kapuk két végén állnak.

37. Tétel (0-1 elv). *Ha egy összehasonlító hálózat n hosszú bemenettel helyesen rendezi a* 0*-ák és* 1*-ek 2^n összes lehetséges sorrendjét, akkor tetszőleges számok összes lehetséges sorrendjét is helyesen rendezi.*

Bizonyítás. Tegyük fel indirekt, hogy a hálózat helyesen rendezi az összes $0-1$ sorrendet, de van olyan tetszőleges számokból álló sorrend, amelyet nem. Ekkor $\exists$ olyan $\langle a_1, a_2, \ldots, a_n \rangle$, amelyben $a_i < a_j$, de az algoritmus fordított sorrendbe rakta őket a kimenetben. Legyen f egy monoton növő függvény,



1. ábra. Rendező hálózat[1]

amelyre:

$$f(x) = \begin{cases} 0, & \text{ha } x \leq a_i \\ 1, & \text{ha } x > a_i \end{cases}$$

Mivel a_i és a_j fordított sorrendben lesz, ebből következik, hogy $f(a_i)$ és $f(a_j)$ sorrendje is rossz lesz, ha $\langle f(a_1), \ldots, f(a_n) \rangle$ az input. Ám mivel $f(a_j) = 1$ és $f(a_i) = 0$, ebből következik, hogy van egy olyan $\langle f(a_1), \ldots, f(a_n) \rangle$ $0-1$ sorozat, amelyet a hálózat rosszul rendezett, ez pedig ellentmondás. ■

38. Tétel (Párhuzamosított rendezők alaptétele). *A párhuzamos csererendezők ütemszáma, ahol egy ütem egy kapurendszert jelent:* $U_{PrCsereRend}(n) = \Omega(\log(n))$

Bizonyítás. Ha egy rendező hálózatban az egy ütemben lévő összehasonításokat egymás után végezzük, akkor a rendezés megegyezik egy általános (szekvenciális) csererendezővel, amelyről tudjuk, hogy legkevesebb $\Omega(n \cdot \log(n))$ összehasonlítással tud rendezni egy n hosszú bemenetet. Egy ekkora bemeneten legfeljebb $\left[\frac{n}{2}\right]$ csere végezhető egy ütemben. Ebből következik:

$$\frac{n \cdot \log n}{\frac{n}{2}} = 2 \log n = \Omega(\log n) \tag{29}$$

■

6.2. Batcher-féle páros-páratlan összefésülés

Az algoritmus[3]

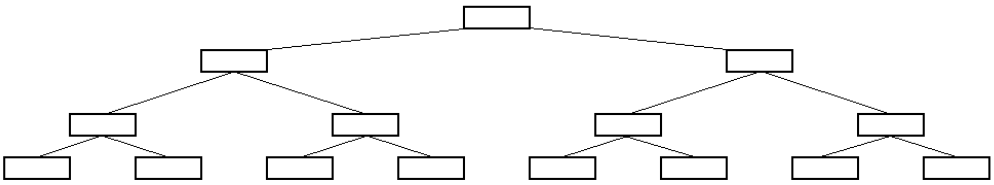
Az algoritmus hatékonyabbá teszi az összefésülő rendezést az összefésülő lépés párhuzamosításával. Itt is feltesszük, hogy két, már rendezett részsorozatunk van (A és B). Mindkettőt kettébontjuk a páros és páratlan indexű elemeik mentén (A_{ps} és A_{pltn} valamint B_{ps} és B_{pltn}), majd keresztben összefésüljük őket, azaz A_{ps}-t B_{pltn}-nal (ennek eredménye legyen C_1) és A_{pltn}-t B_{ps}-sal (eredménye C_2). Végül C_1 és C_2 összefésülése adja a végeredményt.

Az algoritmus műveletigénye: $O(\log^2(n))$.

6.3. Párhuzamos Versenyrendezés

Az algoritmus

Az versenyrendező algoritmus azonos szinten lévő összehasonlító lépései párhuzamosan, egyszerre több szálon végezhetőek, ahogy az szemléletesen látszik a 2 ábrán.



2. ábra. Tournament tree

6.4. Párhuzamos Kiválasztásos rendezés

Az algoritmus

A kiválasztásos rendezés párhuzamosítható, ha minden lépésben egyszerre keresünk minimumot és maximumot is úgy, hogy a minimumot a rendezendő tömb első, a maximumot pedig az utolsó elemével cseréljük meg, így a következő ciklusban már egy 2-vel kisebb tömbön fogunk dolgozni. Az eljárás tovább módosítható úgy, hogy a kezdeti tömböt kettébontjuk és mindkét részen párhuzamosan elvégezzük a kiválasztásokat, majd a összehasonlítjuk a megtalált két-két minimumot és maximumot. A minimumok közül a kisebb, a maximumok közül pedig a nagyobb lesz a végleges elem. Ezt ismételjük addig, amíg a teljes bemenet rendezve nem lesz.

6.5. Párhuzamos buborék rendezés

Az algoritmus

Ebben az esetben úgy párhuzamosíthatjuk az eredeti algoritmust, hogy minden lépésben a páronként független elemeket egyszerre hasonlítjuk össze, és szükség esetén cseréljük meg. Ezzel a módszerrel egy lépésben nem egy elemet mozdítunk el, hanem az inputtól függően akár $n/2$ csere is lehetséges.

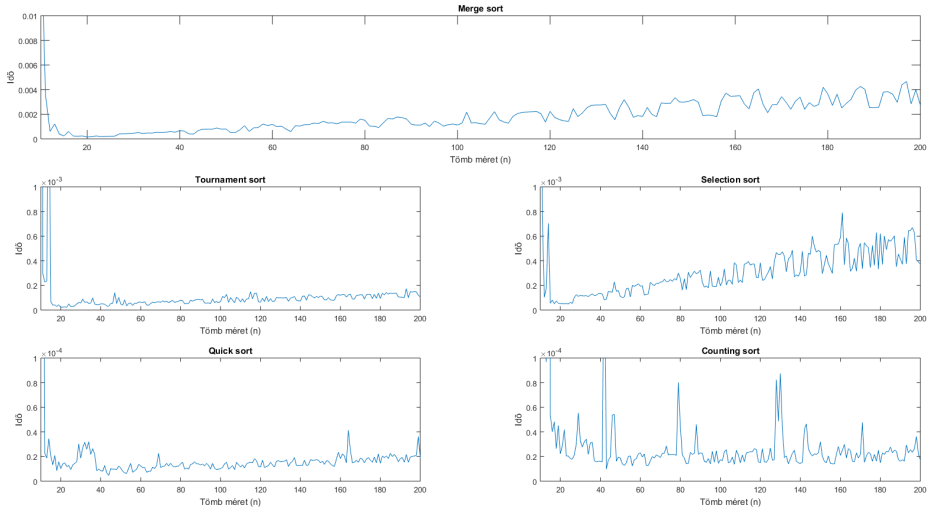
Az algoritmus műveletigénye: $\Omega(n)$.

7. Futási eredmények

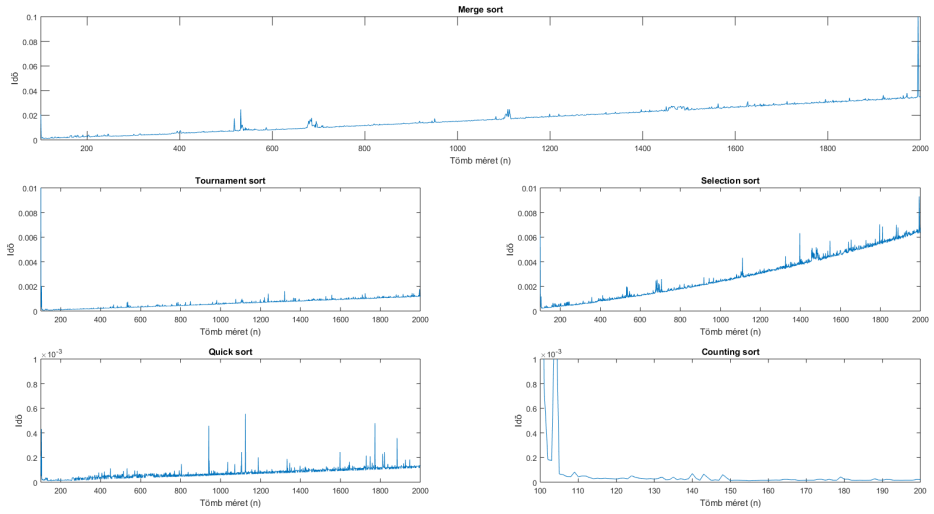
7.1. Szekvenciális rendezések

Az eddigiekben megvizsgáltuk, hogy elméletben melyik rendező mennyire gyorsan rendezi a bemenetként kapott permutációt. A most következő részben ábrákon is összehasonlítjuk a gyorsrendezés, a versenyrendezés, a kiválasztó rendezés, az összefésülő rendezés és a leszámoló rendezés teljesítményét, amelyek az algoritmusok leírásánál található struktogrammok alapján, Matlabban kerültek implementálásra, kivéve a gyorsrendezést, ami a Matlab saját beépített függvénye. Minden ábrán az algoritmusok futásideje látható az input méretének függvényében. A rendezések bemenete minden esetben 1-nél nagyobb egész számok egy permutációja volt, amelyeket a $[0, 1)$ intervallumban generált egyenletes eloszlású számokból hoztunk létre; tehát az input a leszámoló rendezés kezdeti feltételének is megfelelt. Mindkét ábrán az egy sorban lévő grafikonok idő tengelyei megegyeznek, soronként pedig 1 nagyságrendet nőnek; a nagyságrendek a tengely tetején vannak feltüntetve. Az eltérő beosztásra a könnyebb olvashatóság érdekében volt szükség. Az ábrákhoz algoritmusok futásidejét mértem le, nem pedig az általuk elvégzett lépésszámot, ezért a korábban definiált műveletigényhez képest a számítógép teljesítménye miatt torzabb eredményt kapunk.

A 3 ábrán a bemenet 10-ről 200 elemre növekszik egyesével, a 4 ábrán pedig 100-ról 2000-re. Mindkét ábrán jól látható, hogy az elemszám növekedésével növekszik a futásidő is, kivéve a leszámoló rendezés esetében, ahol a 4 ábrán nagyon jól látszik, hogy bár az input mérete radikálisan megnőtt, a futásidő alig változott.



3. ábra. 10-200 elemű véletlen számokból álló inputokra



4. ábra. 100-2000 elemű véletlen számokból álló inputokra

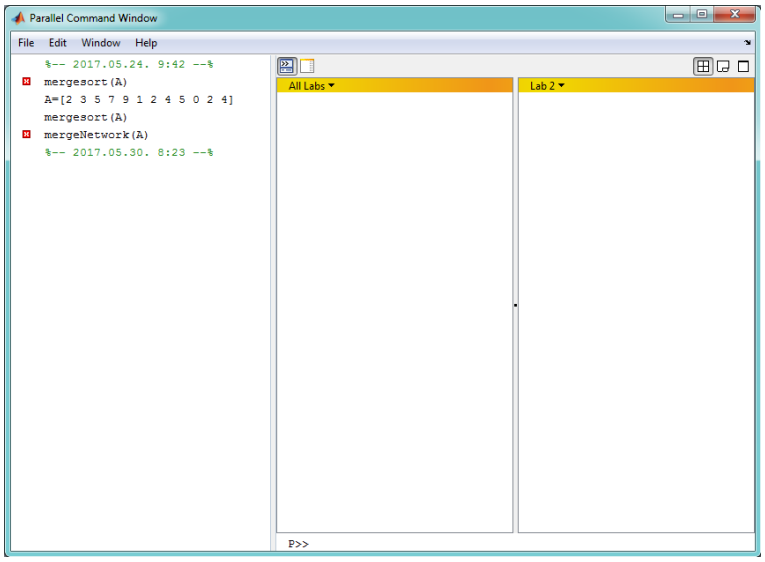
7.2. Párhuzamos rendezések

A Matlabhoz az R2010b verziótól kezdve létezik a *Parallel Computing Toolbox (PCT)*, amellyel már több szálon futtatható programok is létrehozhatóak, többmagos processzorok, GPU-k (graphics processing unit) és egymással összekapcsolt számítógépfürtök (clusters) segítségével. A Matlab korábban is alkalmazott párhuzamosítást bizonyos feladatok esetén, azonban ezek nem külső utasításra, hanem a beépített végrehajtási környezet hatására való-

sultak meg. A PCT eszközeivel azonban már végrehajthatóak szekvenciális munkák párhuzamosan (5. ábra), általános párhuzamos programozási fogalmak, párhuzamosított for ciklusok, valamint szerepel benne egy interaktív környezet (6. ábra), amely párhuzamos algoritmusok feljesztésére és tesztelésére alkalmas.[5]

```
Trial>> parpool
Starting parallel pool (parpool) using the 'local' profile ...
connected to 2 workers.

ans =

 Pool with properties:

            Connected: true
           NumWorkers: 2
              Cluster: local
        AttachedFiles: {}
          IdleTimeout: 30 minutes (30 minutes remaining)
          SpmdEnabled: true

Trial>>
```

5. ábra. *A „**parpool**” parancs indítja a párhuzamos környezetet; ez paraméterezhető például a dolgozók számával.*



6. ábra. *A „**pmode**” paranccsal nyitható meg az interaktív környezet, amelyben minden dolgozóra meg lehet határozni az elvégzendő feladatát.*

A teljes Matlab verzióban lokális, és virtuális szálakon, úgy nevezett „dolgozókon” futtathatóak az alkalmazások, azonban az ingyenes próbaverzió csak a lokális dolgozókat engedélyezi, ez általában 2-8 dolgozót jelent. Az általam használt számítógépen 2 dolgozó volt elérhető. Ezekkel a párhuzamosítás csak részben hajtható végre, mert az algoritmust legfeljebb két szálon

lehet egyszerre futtatni. A két szálon történő párhuzamosítás azonban nem gyorsított a kipróbált algoritmuson akkora mértékben, hogy érdemes legyen további algoritmusokon is letesztelni, különösen azért, mert a párhuzamosító környezet létrehozása több időt vett igénybe a futtatás elején, mint a szekvenciális rendezés teljes futása, egy 20000 elemből álló, random számokból létrehozott tömbön. Ebből a tapasztalatból kiindulva tehát ha a rendezési algoritmusokat a tényleges (számítógép teljesítménytől függő) futásidejük, és nem az elvégzett műveleteik alapján hasonlítanánk össze, ahhoz olyan méretű inputot kellene rendezni, amelynek a szekvenciális módszerrel történő rendezése lényegesen hosszabb ideig tartana, mint a program által használt párhuzamos környezet létrehozása, és az algoritmus lefuttatása. A Matlabon kívül természetesen léteznek még kimondottan párhuzamos programozást megvalósító programnyelvek is, de az általam ismertek közül a Matlab tartalmazza a legtöbb alapból beépített, ennélfogva optimalizált matematikai eljárást.

8. Függelék

Segédfüggvények

Mivel a Matlabban sok előre beépített matematikai függvény van – például a maximum keresés –, így azokat nem hoztam létre külön, csak felhasználtam az algoritmusokban.

```
function tomb=csere(tomb,index1,index2)
temp=tomb(index1);
tomb(index1)=tomb(index2);
tomb(index2)=temp;

function out=merge(left, right)
out=[];
while ~isempty(left) && ~isempty(right)
    if left(1) <=right(1)
        out=[out,left(1)];
        left=left(2:length(left));
    else
        out=[out,right(1)];
        right=right(2:length(right));
    end
end
if ~isempty(left)
    out=[out,left];
end
if ~isempty(right)
    out=[out,right];
end

function out=fillTree( input )
    j=length(input)-1;
    out=[zeros(1,j) input];
    while j >= 1
        out(j)=max(out(2*j),out(2*j+1));
        j=j-1;
    end
end

function out = treemax( treeData )
j=1;
n=(length(treeData)+1)/2;
while j < n
    if treeData(j)==treeData(2*j)
```

```
        j=2*j;
    else
        j=2*j+1;
    end
end
treeData(j)=-Inf;
j=floor(j/2);
while j>=1
    treeData(j)=max(treeData(2*j),treeData(2*j+1));
        j=floor(j/2);
end
out=treeData;
end
```

Kiválasztó algoritmus

```
function input=selection(input)
n=length(input);
for j=n:-1:2
    %maximumkereses az n-j meretu tombben
    [~,index]=max(input(1:end-(n-j)));
    %a maximum es a j-edik elem csereje az inputban
    input=csere(input,index,j);
end
```

Versenyrendező algoritmus

```
function out = tournament( input )
n=length(input);
out=zeros(1,n);
sort=fillTree(input);
i=n;
out(i)=sort(1);
r=n-1;
while r >= 1
    sort=treemax(sort);
    i=i-1;
    out(i)=sort(1);
    r=r-1;
end
```

Összefésülő algoritmus

```
function [out]=mergesort(m)
len=length(m);
    if len<=1
        out=m;
    else
        k=floor(len/2);
        left=mergesort(m(1:k));
        right=mergesort(m(k+1:len));
        out=merge(left,right);
    end
```

Leszámoló algoritmus

```
function out=counting(input)
n=max(input);
C=zeros(1,n);
m=length(input);
for j=1:m
    C(input(j))=C(input(j))+1;
end
for i=2:n
    C(i)=C(i)+C(i-1);
end
out=zeros(1,m);
for j=m:-1:1
    out(C(input(j)))=input(j);
    C(input(j))=C(input(j))-1;
end
```

Keret

```
clear all
close all

figure
x=zeros(1,900);y=zeros(1,900);a=zeros(1,900);
b=zeros(1,900);c=zeros(1,900);d=zeros(1,900);
for m=100:2000
    x(m-9)=m;
    in= round( m * rand(1, m)+1);
    tic
    sort(in);
    y(m-9)=toc;
    tic
```

```
    tournament(in);
    a(m-9)=toc;
    tic
    counting(in);
    b(m-9)=toc;
    tic
    mergesort(in);
    c(m-9)=toc;
    tic
    selection(in);
    d(m-9)=toc;

end

subplot(3,2,[1 2])
plot(x,c)
title('Merge sort')
xlabel('Input size (n)')
ylabel('Time')
axis([100 2000 0 0.1]);

subplot(3,2,3)
plot(x,a)
title('Tournament sort')
xlabel('Input size (n)')
ylabel('Time')
axis([100 2000 0 0.01]);

subplot(3,2,4)
plot(x,d)
title('Selection sort')
xlabel('Input size (n)')
ylabel('Time')
axis([100 2000 0 0.01]);

subplot(3,2,5)
plot(x,y)
title('Quick sort')
xlabel('Input size (n)')
ylabel('Time')
axis([100 2000 0 0.001]);

subplot(3,2,6)
```

```
plot(x,b)
title('Counting sort')
xlabel('Input size (n)')
ylabel('Time')
axis([100 200 0 0.001]);
```

Hivatkozások

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein **Új Algoritmusok** (Scolar Kiadó, 2003)

[2] Author: Avrim Blum, Instructor: Manuel Blum **Algorithms** (Lecture 5), url: `https://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0130.pdf`

[3] Attila Házy, Ferenc Nagy **Adatstruktúrák és algoritmusok** (jegyzet, 2011. április 6)

[4] Fekete István, Hunyadvári László, Nagy Tibor, Giachetta Roberto, Danyluk Tamás, Bartha Dénes, Ilonczsai Zsolt, **Algoritmusok és adatszerkezetek: Jegyzet az egyetemi informatikus alapképzéshez**, (ELTE Informatikai Kar; Budapest, 2014)

[5] Galántai Aurél, Hegedűs Csaba, Sram Norbert, **A numerikus lineáris algebra párhuzamos algoritmusai** (jegyzet, Óbudai Egyetem, 2015, 3.fejezet); url: `http://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_03_numerikus_linearis_algebra_parhuzamos_algoritmusai/ar01s03.html`

[6] `http://homepages.math.uic.edu/~leon/cs-mcs401-s08/handouts/stability.pdf` Utolsó látogatás: 2017.05.20.