Tampereen yliopisto

Kari Harmaahieta

LAJITTELUALGORITMIEN VERTAILUA

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Joulukuu 2020

Tiivistelmä

Kari Harmaahieta: Lajittelualgoritmien vertailua
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Joulukuu 2020

Lajittelu on ollut tietokoneiden tärkeimpiä tehtäviä alusta alkaen. Suuri osa vieläkin käytössä olevista lajittelualgoritmeista on kehitetty jo vuosikymmeniä sitten. Tässä tutkielmassa tutustutaan kuuteen tunnettuun lajittelualgoritmiin ja kolmeen tuoreeseen tutkimukseen lajittelualgoritmien suorituskyvystä. Tutkielmaan valitut lajittelualgoritmit ovat valintalajittelu, kuplalajittelu, lisäyslajittelu, lomituslajittelu, kekolajittelu sekä pikalajittelu. Tutkielmassa esitetään kaikkien edellämainittujen algoritmien toimintaperiaate sekä niiden asymptoottinen aikavaativuus. Tutkielmassa käsiteltävät tutkimukset ovat valittu niin, että niissä olisi tutkittu mahdollisimman paljon samoja lajittelualgoritmeja kokeellisin keinoin. Tutkielman tavoitteena on löytää nopeimmat lajittelualgoritmit tekemällä kirjallisuuskatsauksen lajittelualgoritmeista tehtyihin empiirisiin tutkimuksiin.

Tutkimuksissa vertaillaan lajittelualgoritmeja sekä teoreettisesti että empiirisesti. Tuloksista havaitaan, että pikalajittelun ja lomituslajittelun eri toteutukset ovat nopeimpia lajittelualgoritmeja sekä asymptoottisen suoritusajan että empiirisen kokeen mukaan. Tutkimuksissa löydetään myös monia optimointikeinoja pika- ja lomituslajittelulle. Loput tarkastelussa mukana olevat algoritmit eivät pääse lähellekään samoja suoritusaikoja. Tutkimuksista tehdään myös havainto, että lajittelualgoritmien suoritusaika ja keskinäinen nopeusjärjestys voi riippua myös niille lajiteltavaksi annettavasta syötteestä. Erään tutkimuksen mukaan esimerkiksi negatiivisten lukujen lajittelu oli nopeampaa kuin positiivisten lukujen lajittelu, vaikkakin tutkimuksen kuvauksesta ei selviä toistettiinko koe kuinka monta kertaa.

Tässä tutkielmassa vertailtavat algoritmit ovat aiheen rajauksesta johtuen perinteisiä yhden säikeen laskentaa hyödyntäviä algoritmeja. Erään tutkimuksen mukaan rinnakkain tapahtuva laskenta nopeuttaa lajittelu kuitenkin huomattavasti. Tiedon määrän kasvaessa ja tekniikan kehittyessä uusin kehitys lajittelun saralla tapahtuukin rinnakkaislaskentaa hyödyntävien algoritmien kehityksessä.

Avainsanat: lajittelualgoritmit, algoritmit, asymptoottinen suoritusaika,
algoritmitutkimus.

Sisällys

1 Johdanto

Lajittelu (sorting) tai *järjestäminen* (sequencing) tarkoittaa jonkin kokoelman elementtien järjestämistä vaadittuun järjestykseen (Buradagunta ja muut, 2020). Lajittelua on esimerkiksi nimilistan järjestäminen aakkosjärjestykseen tai numeroiden järjestäminen suuruusjärjestykseen. Lajittelua suoritetaan koneellisesti *lajittelualgoritmeilla* (sorting algorithm). Tässä tutkielmassa tutustutaan erilaisien lajittelualgoritmien tehokkuuteen niistä tehtyjen tutkimusten avulla.

Koneellisen lajittelun voidaan katsoa saaneen alkunsa vuonna 1890, kun Herman Hollerith kehitti koneen avustamaan Yhdysvaltain väestönlaskentaa. Väestönlaskennan tiedot kerättiin tällöin reikäkorteille. Kone toimi sähköllä ja se pystyi käsittelemään ja lajittelemaan 49 reikäkorttia minuutissa. Hollerith perusti yrityksen, joka lopulta päätyi osaksi IBM:ia. Ensimmäiset tietokoneet tulivat 1940-luvulla, jolloin tehtiin myös ensimmäiset lajitteluohjelmat tietokoneelle. John von Neumann kehitti ohjelman lomituslajitteluun vuonna 1945 EDVAC-tietokoneelle. Samoihin aikoihin saksalainen K. Zuse kehitti ohjelman lisäyslajitteluun. Ensimmäisten tietokoneiden sisäisen muistin määrä oli todella pieni, joka vaikutti myös lajittelualgoritmien kehitykseen. Magneettinauhoja tai kasetteja käytettiin nopeuttamaan lajittelua esimerkiksi lomituslajittelussa lisäämällä magneettinauhojen määrää. 1950-luvulle tultaessa monia lajittelualgoritmeja oli jo kehitetty, mutta niiden teoriaa oli kehitetty melko vähän. Vuonna 1952 julkaistiin yksi ensimmäisistä lajittelualgoritmien vertailuista, kun Daniel Goldenberg julkaisi tutkimuksensa "Time analyses of various methods of sorting data". Kyseisessä tutkimuksessa nopein algoritmi oli eräänlainen lomituslajittelu. Vuoteen 1970 mennessä suuri osa tärkeimmistä lajittelualgoritmeista oli jo kehitetty, kuten esimerkiksi kaikki tässä tutkielmassa käsiteltävät algoritmit. Tämän jälkeenkin kehitystä on tapahtunut, mutta se on pääasiassa ollut aiempien algoritmien variaatioiden kehitystä. (Knuth, 2014)

Lajittelu on ollut yksi tietokoneiden tärkeimpiä tehtäviä alusta saakka. Jo 1960-luvulla tietokonevalmistajat arvioivat, että yli 25 prosenttia kaikesta suoritusajasta kului lajitteluun, ja joissain järjestelmissä jopa yli puolet. (Knuth, 2014) Varhainen algoritmitutkimus keskittyikin erityisesti suurien tietomäärien lajitteluun (Heineman ja muut, 2016).

Lajittelua käytetään esimerkiksi ratkaisemaan *yhteenkuuluvuuden ongelmaa* (togetherness problem) sekä helpottamaan tiedon etsimistä (Knuth, 2014). Tiedon lajittelua tarvitaan erityisesti *ison datan* (big data) käsittelyyn. Viime vuosina lajittelualgoritmien kehitys onkin keskittynyt erityisesti luomaan uudenlaisia, ison datamäärän käsittelyyn sopivia algoritmeja tai parantamaan olemassaolevia algoritmeja. Algoritmien suorituskykyä voidaan parantaa esimerkiksi käyttämällä hyödyksi *rinnakkaislaskentaa* (parallel computing), kuten näytönohjaimia tai useampia tietokoneita (Shatnawi ja muut, 2018). Rinnakkaislaskenta soveltuu hyvin lajitteluun, sillä esimerkiksi lomitus- ja pikalajittelu hyödyntävät hajota ja hallitse -suunnittelumallia.

Lajittelualgoritmeja voi luokitella toimintaperiaatteen mukaan eri tavoilla. Jos algoritmi käyttää tietokoneen sisäisen muistin lisäksi myös ulkoista muistia, kyseessä on *ulkoinen*

lajittelualgoritmi (external sorting algorithm). Jos algoritmi pärjää pelkällä tietokoneen sisäisellä muistilla, se on silloin *sisäinen lajittelualgoritmi* (internal sorting algorithm) (Shatnawi ja muut, 2018). Jos algoritmi säilyttää yhtäsuurien elementtien järjestyksen, se on *vakaa* (stable), jos taas ei, se on *epävakaa* (unstable) (Heineman ja muut, 2016).

Tässä tutkielmassa tutustutaan perinteisiin, rinnakkaislaskentaa hyödyntämättömiin lajittelualgoritmeihin, sekä uusimpiin tutkimustuloksiin niiden tehokkuudesta. Tutkielman tavoitteena on tehdä kirjallisuuskatsaus lajittelualgoritmeista tehtyihin tutkimuksiin ja siten löytää nopeimmat lajittelualgoritmit sekä teoreettisesti että empiirisellä tutkimuksella vertailtuna.

Tutkielman toisessa luvussa kerrotaan kuinka algoritmien suorituskykyä voidaan mitata. Kolmannessa luvussa esitellään muutama yleisin perinteinen lajittelualgoritmi. Neljännessä luvussa esitellään tutkielman lähteiksi otetut tutkimukset ja viidennessä luvussa tutustutaan niiden tuloksiin tarkemmin. Lähteeksi on otettu kolme lajittelualgoritmien tehokkuudesta tehtyä tutkimusta, joissa tutkitaan lajittelualgoritmien tehokkuutta sekä empiirisesti että teoreettisesti. Tutkimukset on valittu siten, että ne ovat mahdollisimman tuoreita ja niissä on vertailtu mahdollisimman paljon samoja lajittelualgoritmeja. Lopuksi kuudennessa luvussa tehdään yhteenveto koko tutkielmasta ja pohditaan lajittelun tulevaisuutta.

2 Algoritmien tehokkuus

Algoritmien tehokkuutta voidaan tarkastella joku teoreettisesti tai kokeellisesti. Kappaleessa 2.1 käsitellään algoritmin tehokkuuden teoreettista tarkastelua ja kappaleessa 2.2 kokeellista.

2.1. Asymptoottinen kompleksisuus

Algoritmin *kompleksisuutta* (complexity) voidaan tarkastella kahdesta eri näkökulmasta. Algoritmin aikakompleksisuus tarkastelee algoritmin suoritusaikaa ja operaatioiden määrää. Algoritmin tilakompleksisuus tarkastelee algoritmin suorituksen aikana vaatimaa tilaa. Kompleksisuutta tarkasteltaessa oletetaan algoritmin syötteen kooksi n. (Singhal, 2018)

Aikakompleksisuutta määritettäessä lasketaan algoritmin suorittamien perusoperaatioiden määrää. Perusoperaatio on jokin muistia käyttävä tai vertaileva toiminto. Lisäksi huomioidaan algoritmin sisältämien silmukoiden suorituskerrat. (Singhal, 2018)

Tilakompleksisuus kuvaa algoritmin vaatiman muistin määrää tietynkokoisella syötteellä. Tilakompleksisuutta ei mitata yleensä bitteinä, vaan tietynkokoisten yksiköiden määränä. Tilakompleksisuudessa ei huomioida lajiteltavan datan viemää tilaa, vaan vain sen lisäksi vaadittu tila. (Singhal, 2018) Siten siis *ilman lisätilaa* (in-place) järjestelevät algoritmit ovat tilakompleksisuudeltaan aina O(1).

Kompleksisuutta ilmaistaan *asymptoottisella notaatiolla* (asymptotic notation). Yleisim-

min käytetyt notaatiot ovat *O-notaatio* (Big O-notation), *Ω-notaatio* (Big Omega-notation) sekä *Θ-notaatio* (Big Theta-notation). O-notaatiolla ilmaistaan algoritmin hitainta mahdollista suoritusaikaa, Ω-notaatiolla ilmaistaan algoritmin nopeinta mahdollista suoritusaikaa ja Θ-notaatiolla ilmaistaan algoritmin keskiarvoista suoritusaikaa. (Singhal, 2018)

Asymptoottinen notaatio sisältää vain suurimman asteen polynomisesta funktiosta. Kaava 1 sisältää erään algoritmin operaatioiden määrän syötteen koon *n* funktiona.

$$f(n) = 3n^2 + 2n + 22 \tag{1}$$

Syötteen koon kasvaessa funktion ensimmäinen osa ($3n^2$) kasvaa nopeiten, joten ainoastaan se huomioidaan asymptoottista suoritusaikaa määrittäessä. Algoritmin suoritusajan kasvunopeutta tarkasteltaessa myöskään polynomin kiinteä kerroin (3) ei ole kasvunopeuden kannalta merkityksellinen, joten algoritmin lopulliseksi asymptoottiseksi suoritusajaksi jää $O(n^2)$. (Bae, 2019) Taulukossa 1 on kuvattuna algoritmien suoritusaikaluokat nopeimmasta hitaimpaan. Taulukossa ylimpänä on nopein mahdollinen kompleksisuus, eli vakio, ja alimpana hitain, eli kertoma. (Singhal, 2018)

Taulukko 1. Algoritmien suoritusaikaluokat (taulukko muokattu lähteestä (Singhal, 2018)).

Suoritusaika	Luokka
1	Vakio
Log n	Logaritminen
n	Lineaarinen
n Log n	n log n
n^2	Neliöllinen
n^3	Kuutiollinen
n^k	Polynominen
2^n	Eksponentiaalinen
n!	Kertoma

2.2. Käytännön suoritusaika

Algoritmien käytännön suoritusaikaa voidaan mitata toteuttamalla algoritmi jollain ohjelmointikielellä ja suorittamalla algoritmi tietokoneella mitaten sen suoritusaikaa. Koska tietokoneet ovat suorituskyvyltään erilaisia, käytännön suoritusajan mittaukset eri tietokoneiden välillä eivät ole keskenään vertailukelpoisia. Samalla tietokoneella voidaan kuitenkin vertailla erilaisten algoritmien keskinäistä tehokkuutta. Myöskin algoritmien aikavaativuus on oletettavasti riippumaton toteutuksesta ja suoritusympäristöstä.

3 Lajittelualgoritmit

Seuraavaksi esitellään muutama yleisin lajittelualgoritmi. Esittelyssä käytetään apuna englanninkielistä pseudokoodia selkeyden ja lyhyyden vuoksi. Algoritmien kompleksisuut-

ta käsitellään huonoimman tapauksen (O-notaatio) kautta. Algoritmeissa oletetaan, että taulukon A ensimmäinen indeksi on 1. Käsiteltävistä algoritmeista kaikki paitsi lomituslajittelu ja pikalajittelu lajittelee listan *paikallaan* (in place), eli ne eivät vaadi ylimääräistä tilaa (Heineman ja muut, 2016).

3.1. Valintalajittelu

Algoritmi 1: Valintalajittelu (muokattu lähteestä (Verma ja Singh, 2015)).

```
1  funktio Selectionsort(A):
   Syöte          : Lista A, joka sisältää n kokonaislukua
   Palautusarvo: Lista A järjestettynä nousevaan numerojärjestykseen
2  for i ← 1 to n do
3  |  min ← i
4  |  for j ← i + 1 to n do
5  |  |  if A[j] < A[min] then
6  |  |  |  min ← j
7  |  |  end
8  |  end
9  |  SwapA[i]&A[min]
10 end
```

Valintalajittelu (selection sort) järjestää listan etsimällä sen pienimmän arvon, laittamalla sen listan ensimmäiseksi ja toistamalla tämän lopulle listalle, kunnes kaikki listan elementit ovat järjestyksessä. Valintalajittelun aikakompleksisuus on O(n^2) (Verma ja Singh, 2015). Suuren aikakompleksisuuden vuoksi valintalajittelu ei sovellu kuin todella pienille tietomäärille.

3.2. Kuplalajittelu

Algoritmi 2: Kuplalajittelu (muokattu lähteestä (Verma ja Singh, 2015)).

```
1  funktio Bubblesort(A):
   Syöte          : Lista A, joka sisältää n kokonaislukua
   Palautusarvo: Lista A järjestettynä nousevaan numerojärjestykseen
2  for i ← 1 to n do
3  |  for j ← n to i + 1 do // decrease j
4  |  |  if A[j] < A[j − 1] then
5  |  |  |  SwapA[j]&A[j − 1]
6  |  |  end
7  |  end
8  end
```

Kuplalajittelussa (bubble sort) lajittelu tapahtuu vertailemalla listan vierekkäisiä elementtejä pareittain. Jos ne eivät ole järjestyksessä, elementit vaihdetaan keskenään. Jokaisella vertailukierroksella listan viimeinen elementti pääsee paikalleen. Kuplalajittelun aikakom-

pleksisuus on O(n^2). (Verma ja Singh, 2015) Suuren aikakompleksisuuden vuoksi kuplalajittelu ei sovellu kuin todella pienille tietomäärille.

3.3. Lisäyslajittelu

Algoritmi 3: Lisäyslajittelu (muokattu lähteestä (Heineman ja muut, 2016)).

```
1 funktio Insertionsort(A):
  Syöte          : Lista A, joka sisältää n kokonaislukua
  Palautusarvo: Lista A järjestettynä nousevaan numerojärjestykseen
2 for i ← 2 to n do
3 |   j ← i − 1
4 |   while j ≥ 1 and A[j] > A[i] do
5 |   |   A[j + 1] ← A[j]
6 |   |   j ← j − 1
7 |   end
8 |   A[j + 1] ← A[i]
9 end
```

Lisäyslajittelussa (insertion sort) lista käydään elementti kerrallaan läpi. Listan alkuosa pidetään aina lajiteltuna. Lajitellun listan loppuun lisätään aina jäljellä olevista elementeistä pienin, kunnes koko lista on lajiteltu. Lisäyslajittelun aikakompleksisuus on O(n^2) (Heineman ja muut, 2016), jonka vuoksi lisäyslajittelu ei sovellu kuin todella pienille tietomäärille.

3.4. Lomituslajittelu

Algoritmi 4: Lomituslajittelu (muokattu lähteestä (Verma ja Singh, 2015)).

```
1 funktio Mergesort(A, p, r):
  Syöte          : Lista A, joka sisältää n kokonaislukua
                   Kokonaisluku p, alilistan ensimmäinen elementti
                   Kokonaisluku r, alilistan viimeinen elementti
  Palautusarvo: Lista A järjestettynä nousevaan numerojärjestykseen
2 if p < r then
3 |   q ← (p + r)/2
4 |   Mergesort(A, p, q)
5 |   Mergesort(A, q + 1, r)
6 |   Merge(A, p, q, r)
7 end
```

Lomituslajittelu (merge sort) perustuu *hajota ja hallitse* (divide and conquer) -suunnittelumalliin. Lomituslajittelussa listaa jaetaan pienempiin osiin kunnes se on helposti helposti lajiteltavissa. Sitten algoritmi *yhdistää* (merge) pienemmät lajitellut listat lopulliseen lajiteltuun listaan. Lomituslajittelun aikakompleksisuus on O($n \log_2 n$). (Verma ja Singh, 2015) Yllä oleva algoritmi käyttää lisäksi apufunktiota merge, joka yhdistää kaksi listaa järjestyksessä yhdeksi. (Heineman ja muut, 2016)

3.5. Kekolajittelu

Algoritmi 5: Kekolajittelu (muokattu lähteestä (Heineman ja muut, 2016)).

```
1 funktio Heapsort(A):
    Syöte          : Lista A, joka sisältää n kokonaislukua
    Palautusarvo: Lista A järjestettynä nousevaan numerojärjestykseen
2 BuildHeap(A)
3 for i ← n to 1 do // decrease i
4 |   SwapA[0]&A[i]
5 |   Heapify(A, 0, i)
6 end
```

Kekolajittelu (heap sort) käyttää hyväkseen *kekoja*. Keko on binääripuu, jolla on kaksi tärkeää ominaisuutta:

- 1.Puun syvyydellä $k > 0$ oleva solmu voi olla olemassa vain, jos kaikki 2^{k-1} solmua syvyydellä $k - 1$ ovat olemassa.
- 2. Puun jokaisen solmun arvo on suurempi tai yhtä suuri kuin sen lapsien.

Algoritmin aluksi lista muutetaan keoksi. Sitten keko käydään läpi lopusta alkuun elementti kerrallaan. Jokaisella kierroksella vaihdetaan juuressa oleva elementti viimeisimmän elementin kanssa, jonka jälkeen jäljelle jäävän keon keko-ominaisuus palautetaan, eli suurin elementti sijoitetaan puun juureen. Tämä toistetaan kunnes keko on kokonaan järjestyksessä. Koska keko on tässä toteutettu listalla, jäljelle jää valmiiksi lajiteltu lista. Kekolajittelun aikakompleksisuus on O($n \log_2 n$). Ylläoleva algoritmi käyttää lisäksi apufunktioita BuildHeap, joka muuttaa listan aluksi keoksi, ja Heapify, joka palauttaa keon ominaisuudet. (Heineman ja muut, 2016)

3.6. Pikalajittelu

Algoritmi 6: Pikalajittelu (muokattu lähteestä (Verma ja Singh, 2015)).

```
1 funktio Quicksort(A, p, r):
    Syöte          : Lista A, joka sisältää n kokonaislukua
                     Kokonaisluku p, alilistan ensimmäinen elementti
                     Kokonaisluku r, alilistan viimeinen elementti
    Palautusarvo: Lista A järjestettynä nousevaan numerojärjestykseen
2 if p < r then
3 |   q ← Partition(A, p, r)
4 |   Quicksort(A, p, q − 1)
5 |   Quicksort(A, q + 1, r)
6 end
```

Pikalajittelu (quick sort) perustuu lomituslajittelun tavoin hajota ja hallitse -suunnittelumalliin. Optimaalisesti toteutetussa pikalajittelussa listasta valitaan pivot-elementti

(A[q]) jakamaan lista kahteen lähes samankokoiseen listaan niin, että pivot-elementtiä pienemmät elementit ovat sitä ennen ja suuremmat sen jälkeen. Tämä toistetaan rekursiivisesti uusiin luotuihin alilistoihin, kunnes koko lista on järjestyksessä. Pikalajittelun aikakompleksisuus on O($n \log_2 n$). (Verma ja Singh, 2015) Ylläoleva algoritmi käyttää lisäksi apufunktiota Partition, joka jakaa listan kahteen lähes samankokoiseen listaan.

4 Tutkimukset

Tarkastellaan seuraavaksi kolmea erilaista lajittelualgoritmien tehokkuudesta tehtyä empiiristä tutkimusta. Tutkimukset on valittu siten, että ne olisivat mahdollisimman tuoreita ja niissä olisi vertailtu keskenään mahdollisimman paljon samoja algoritmeja.

Tutkimuksessa *A Comparative Analysis of Deterministic Sorting Algorithms based on Runtime and Count of Various Operations* Verma ja Singh (2015) vertailevat keskenään kuutta eri lajittelualgoritmia: valintalajittelu, kuplalajittelu, lisäyslajittelu, pikalajittelu, lomituslajittelu sekä kekolajittelu. Aluksi tutkielmassa esitellään algoritmit ja vertaillaan niiden aikakompleksisuutta keskenään. Seuraavaksi algoritmit toteutetaan Java-ohjelmointikielellä empiiristä osuutta varten. Algoritmeille annetaan järjestettäväksi 10 000 pseudosatunnaisen numeron lista. Algoritmi suoritetaan 500-10 000 kertaa, 500 suorituskerran välein. Algoritmin suorittamien eri operaatioiden (vertailu, vaihto, asetus) määrää laskettaessa syötteen koko on 1 000 satunnaislukua. Lisäksi tarkastellaan operaatioiden määrää valmiiksi järjestetyllä listalla sekä lähes kokonaan järjestetyllä listalla.

Tutkimuksessa *Performance Comparison of Sorting Algorithms with Random Numbers as Inputs* Buradagunta ja muut (2020) vertailevat edellisessä tutkimuksessa vertailtujen algoritmien lisäksi kahta muuta: *UNH-lajittelua* (UNH sort) ja *kantalukulajittelua* (radix sort). Aluksi algoritmien toimintaperiaate käydään läpi. Seuraavaksi algoritmeista toteutetaan kuusi: UNH-lajittelu, lisäyslajittelu, kuplalajittelu, valintalajittelu, lomituslajittelu sekä pikalajittelu. Algoritmeja testataan 100–200 000 satunnaisluvun syötteellä. Vertailu tehdään erikseen positiivisille ja negatiivisille syötteille.

Viimeinen tutkimus *Sort Race* on rakenteeltaan hyvin erilainen aiempiin verrattuna. Zhang ja muut (2016) etsivät tutkimuksessaan vastausta kysymykseen "Mikä on paras lajittelualgoritmi?". Tutkimuksen aluksi valitaan lajittelukisaan viisi erilaista lajittelualgoritmia: lisäyslajittelu, *kuorilajittelu* (shell sort), lomituslajittelu, kekolajittelu ja pikalajittelu. Algoritmit suorittavat 100 kertaa 2 000 000 elementin lajittelun. Lisäksi vertailuun oli otettu kaksi erilaista lomituslajittelua, qsort ja timsort, sekä *introlajittelu* (introsort). Tutkimuksessa määritellään seuraavaksi 12 erilaista syöteluokkaa, joilla lähdetään lopuksi etsimään pikalajittelun ja lomituslajittelun tehokkaimpia toteutustapoja.

5 Tulokset

Verma ja Singh (2015) vertailevat aluksi algoritmien suoritusaikaa ja operaatioiden määrää pseudosatunnaisella syötteellä. Keskimääräinen suoritusaika vaihtelee kuplalajittelun ja

Taulukko 2. Algoritmien sijoitus tutkimuksessa *A Comparative Analysis of Deterministic Sorting Algorithms based on Runtime and Count of Various Operations* (taulukko muokattu lähteestä (Verma ja Singh, 2015)).

Algoritmi	Vertailujen määrän ka.	Vaihtojen määrän ka.	Asetusten määrän ka.	Suoritusaika ka.
Valintalajittelu	5	2	6	6
Kuplalajittelu	6	5	4	5
Lisäyslajittelu	4	6	5	4
Pikalajittelu	1	3	1	1
Lomituslajittelu	3	1	3	3
Kekolajittelu	2	4	2	2

valintalajittelun noin 2 000 000 nanosekunnista kekolajittelun ja pikalajittelun noin 100 000 nanosekuntiin. Tutkimuksessa jätetään tarkat suoritusajat mainitsematta. Operaatioiden määrä satunnaissyötteellä on kaikista isoin kuplalajittelulla ja lisäyslajittelulla, ja pienin pikalajittelulla ja kekolajittelulla. Täysin järjestellyllä syötteellä eniten operaatioita tuli kupla-, valinta- ja pikalajitteluilla. Vähimmällä määrällä selvisi lisäyslajittelu ja lomituslajittelu. Lähes kokonaan järjestellyllä syötteellä eniten operaatioita suorittivat kuplalajittelu ja valintalajittelu, vähiten taas lomituslajittelu ja kekolajittelu. Taulukossa 2 on koottuna tutkimuksen tulokset. Algoritmeille on annettuna järjestysluvut 1–6, jossa 1 tarkoittaa parasta ja 6 huonointa. (Verma ja Singh, 2015) Tuloksista havaitaan, että pikalajittelu on tutkimuksessa (Verma ja Singh, 2015) kaikista nopein algoritmi, mutta kekolajittelu ja lomituslajittelu ovat lähellä samaa suoritusaikaa.

Taulukko 3. Algoritmien keskimääräinen sijoitus tutkimuksessa *Performance Comparison of Sorting Algorithms with Random Numbers as Inputs* syötteen koolla 100-5 000 (taulukko muokattu lähteestä (Buradagunta ja muut, 2020)).

Algoritmi	Sijoitus positiivisella syötteellä	Sijoitus negatiivisella syötteellä
UNH	6	6
Lisäyslajittelu	4	3
Kuplalajittelu	5	4
Valintalajittelu	3	5
Lomituslajittelu	2	2
Pikalajittelu	1	1

Tutkimuksessa *Performance Comparison of Sorting Algorithms with Random Numbers as Inputs* tarkastellaan algoritmien suoritusaikaa positiivisilla ja negatiivisilla luvuilla. Syötteen koko vaihtelee välillä 100–200 000. (Buradagunta ja muut, 2020) Taulukossa 3 tarkastellaan aluksi lajittelualgoritmien sijoituksia tässä tutkimuksessa pienellä syötekoolla ($n \leq 10\,000$) asteikolla 1–6, joissa 1 tarkoittaa parasta ja 6 huonointa.

Taulukko 4. Algoritmien keskimääräinen sijoitus tutkimuksessa *Performance Comparison of Sorting Algorithms with Random Numbers as Inputs* syötteen koolla 10 000-200 000 (taulukko muokattu lähteestä (Buradagunta ja muut, 2020)).

Algoritmi	Sijoitus positiivisella syötteellä	Sijoitus negatiivisella syötteellä
UNH	6	6
Lisäyslajittelu	4	3
Kuplalajittelu	5	5
Valintalajittelu	3	4
Lomituslajittelu	2	2
Pikalajittelu	1	1

Taulukossa 4 tarkastellaan lajittelualgoritmien sijoituksia tässä tutkimuksessa isommalla syötteellä. Algoritmien keskinäinen sijoitus ei juurikaan muuttunut suuren ja pienen syötemäärän välillä. Negatiivisen ja positiivisen syötteen välillä oli jonkin verran eroja algoritmien keskinäisessä suorituskyvyssä. Kaksi nopeinta, eli pikalajittelu ja lomituslajittelu, pysyivät kuitenkin samoilla sijoituksilla.

Taulukko 5. Algoritmien keskimääräinen suoritusaika tutkimuksessa *Performance Comparison of Sorting Algorithms with Random Numbers as Inputs* syötteen koolla 200 000 (taulukko muokattu lähteestä (Buradagunta ja muut, 2020)).

Algoritmi	Suoritusaika positiivisella syötteellä	Suoritusaika negatiivisella syötteellä
UNH	147.1	149.1
Lisäyslajittelu	67.401	74.5
Kuplalajittelu	150.51	150.2
Valintalajittelu	45.472	135.6
Lomituslajittelu	0.901	0.296
Pikalajittelu	0.3097	0.131

Taulukossa 5 on edellämainitun tutkimuksen algoritmien suoritusajat isoimmalla mahdollisella syötteellä. Algoritmien UNH, lisäyslajittelu ja kuplalajittelu suoritusajat olivat lähes samat sekä positiivisella että negatiivisella syötteellä. Valintalajittelun suoritusaika negatiivisella syötteellä nousi lähes kolminkertaiseksi. Lomituslajittelun ja pikalajittelun suoritusaika sensijaan laski kummallakin algoritmilla negatiivisella syötteellä noin kolmasosaan positiivisen syötteen suoritusajasta.

Taulukossa 6 on tutkimuksen *Sort Race* ensimmäisen vertailun tulokset. Vertailussa lajiteltiin erityyppisiä syötteitä 2 000 000 syötteen kokoisella listalla. Tuloksista havaitaan, että keskimäärin paras lajittelualgoritmi on pikalajittelu ja toiseksi paras lajittelualgoritmi on lomituslajittelu sekä timsort joka on lomituslajitteluun pohjautuva. Huomi-onarvoista

Taulukko 6. Algoritmien suhteellinen suoritusaika tutkimuksessa *Sort Race* syötteen koolla 2 000 000 erityyppisillä syötteillä (taulukko muokattu lähteestä (Zhang ja muut, 2016)).

Syötteen tyyppi	Valinta	Keko	Kuori	Lomitus	Pika	qsort	timsort	Intro
satunnainen	-	3.57	1.74	1.18	1.02	1.21	1.17	1.00
päinvastoin lajiteltu	-	113.78	26.34	1.00	1.78	19.74	1.07	7.67
melkein lajiteltu	1.62	5.73	1.92	1.00	1.54	1.50	1.11	1.30
muutama uniikki	-	5.88	2.39	1.64	1.00	1.73	1.61	1.26
keskimääräinen	-	5.18	1.98	1.14	1.00	1.42	1.13	1.21

tuloksista on myös se, että valintalajittelua käytettäessä ei näillä syötteillä saatu läheskään kaikkia tuloksia (Zhang ja muut, 2016). Syynä tähän lienee se, että valintalajittelun kompleksisuus on n^2, ja kyseisellä syötteen koolla $2000000^2 = 4000000000000$, eli operaatioiden määrä on aivan liian suuri.

Taulukko 7. Lomituslajitteluun perustuvien algoritmien keskimääräinen suhteellinen suoritusaika tutkimuksessa *Sort Race* syötteen koolla 2 000 000 (taulukko muokattu lähteestä (Zhang ja muut, 2016)).

Lajittelualgoritmi	Suhteellinen suoritusaika keskimäärin
qsort	1.24
mer2	1.56
mer3	1.64
mer4	1.57
mer5	1.11
mer6	1.09
timsort	1.07
neat	1.65

Taulukossa 7 on tutkimuksen *Sort Race* toisen vertailun tulokset. Vertailussa vertailtiin kahdeksaa erilaista lomituslajittelun toteutusta. Taulukkoon on kirjattu jokaisen lajittelualgoritmin keskimääräinen suhteellinen suoritusaika 12 erilaisella syöteluokalla mitattuna. Vertailusta havaitaan, että timsort on kaikista tehokkain lomituslajittelu. (Zhang ja muut, 2016)

Taulukossa 8 on tutkimuksen *Sort Race* kolmannen vertailun tulokset. Vertailuun otettiin seitsemän erilaista pikalajittelun toteutusta, introlajittelu sekä lomituslajittelun vertailusta algoritmit mer6, timsort ja qsort. Taulukkoon on kirjattu jokaisen lajittelualgo-

Taulukko 8. Pika- ja lomituslajitteluun perustuvien algoritmien keskimääräinen suhteellinen suoritusaika tutkimuksessa *Sort Race* syötteen koolla 2 000 000 (taulukko muokattu lähteestä (Zhang ja muut, 2016)).

Lajittelualgoritmi	Suhteellinen suoritusaika keskimäärin
B&M	1.23
3-way	1.06
2-way	1.04
hyb1	1.02
hyb2	1.00
hyb3	1.01
hyb4	1.01
intro	1.11
mer6	1.09
timsort	1.08
qsort	1.24

ritmin keskimääräinen suhteellinen suoritusaika 12 erilaisella syöteluokalla mitattuna. Taulukosta voidaan havaita, että kaikista tehokkain algoritmi on pikalajittelun toteutus hyb2. Tutkimuksessa nopeimmaksi algoritmiksi siis saadaan pikalajittelu. Lopuksi todetaan kuitenkin, että mikäli tarvitaan vakaata lajittelua tai syöte on osittain lajiteltu eikä muistin määrä ole ongelma, suositellaan käytettävän lomituslajittelua. (Zhang ja muut, 2016)

6 Pohdinta

Taulukko 9. Kolme nopeinta algoritmia jokaisesta tutkimuksesta.

Tutkimus	Nopein algoritmi	2. nopein algoritmi	3. nopein algoritmi
Tutkimus 1 (Verma ja Singh, 2015)	Pikalajittelu	Kekolajittelu	Lomituslajittelu
Tutkimus 2 (Buradagunta ja muut, 2020)	Pikalajittelu	Lomituslajittelu	Lisäyslajittelu
Tutkimus 2 (Zhang ja muut, 2016)	Pikalajittelun eri versiot	Lomituslajittelun eri versiot	Introlajittelu

Taulukossa 9 on koottuna kaikkien edellämainittujen tutkimuksien tulokset yhteen. Tutkielman tavoitteena oli löytää nopeimmat lajittelualgoritmit. Tutkimuksista löytyi melko selkeä näkemys nopeimmista lajittelualgoritmeista. Kaikissa kolmessa tutkimuksessa päädyttiin vastaaviin tuloksiin. Pikalajittelu eri muodoissaan on nimensä mukaisesti nopein lajittelualgoritmi ja lomituslajittelu toteutuksineen on toiseksi nopein lajittelualgoritmi. Tutkimus-

ten empiirisiä havaintoja tukee myös lajittelualgoritmien asymptoottisen suoritusajan tarkastelu: pika- ja lomituslajittelu ovat teoreettisestikin tarkasteltuna nopeimmat. Tutkimuksissa havaittiin myös, että syötteen koko ja laatu voi vaikuttaa lajittelualgoritmin nopeuteen huomattavasti. Kun valitaan käytettävää lajittelualgoritmia johonkin tarkoitukseen, tulisi siis aluksi tarkastella järjestettävää syötettä ja valita algoritmi sen perusteella. Muita huomioitavia asioita algoritmin valinnassa on myös käytettävissä olevan muistin määrä ja halutaanko lajittelualgoritmin olevan vakaa.

Tutkimuksessa *Performance Comparison of Sorting Algorithms with Random Numbers as Inputs* Buradagunta ja muut (2020) havaitsivat kokonaislukuja lajitellessa lomitus- ja pikalajittelun suoritusajan tippuvan noin kolmasosaan kun lajiteltiin positiivisien kokonaislukujen sijaan pelkkiä negatiivisia kokonaislukuja erityisesti suuria määriä. Tutkimuksesta ei kuitenkaan käy ilmi onko jokaisella syötekoolla kokeiltua vain yhtä vai montaa erilaista satunnaissyötettä negatiivisella ja positiivisella syötteellä. Tämä havainto voi siis johtua myös siitä, että negatiivisilla kokonaisluvuilla jotkin annetut syötteet ovat olleet valmiiksi lähempänä oikeaa järjestystä. Jotta tätä havaintoa voitaisiin hyödyntää lajittelualgoritmien optimoinnissa, tarvitaan siis vielä lisää jatkotutkimuksia aiheesta.

Tutkimuksessa *Sort Race* löydettiin useita tapoja optimoida pika- ja lomituslajittelua. Lomituslajittelualgoritmeista parhaiten optimoitu algoritmi on Tim Petersin luoma timsort. Timsort oli lomituslajittelualgoritmeista nopein kun tarkasteltiin suoritusaikoja 12 syöteluokalla, mutta Zhang ja muut (2016) onnistuivat luomaan vieläkin nopeamman lomituslajittelualgoritmin *mer6* kun suoritusaikoja mitattiin vain 8 syöteluokkaa käyttäen. Tämä algoritmi pärjäsi myös pikalajittelua vastaan käytettäessä 8 syöteluokkaa, mutta kaikilla 12 syöteluokalla koko vertailun voittajaksi selviytyi optimoitu pikalajittelualgoritmi *hyb2*. (Zhang ja muut, 2016) Tutkimuksen tulokset korostavat syötteen vaikutusta lajittelualgoritmin nopeuteen ja siten syötteen analysoinnin tärkeyttä lajittelualgoritmin valinnassa.

Tässä tutkielmassa otettiin tarkasteluun perinteiset, yhdellä prosessorilla ja säikeellä tapahtuvaa *CPU-laskentaa* hyödyntävät lajittelualgoritmit ja niiden optimointi. Tietotekniikan kehityksen myötä laskentaan käytettävien säikeiden määrä on kuitenkin lisääntynyt huomattavasti, ja tulevaisuudessa lajittelualgoritmien optimointi tapahtuukin erityisesti rinnakkaislaskentaa käyttämällä. Tästä hyvä esimerkki on näytönohjaimilla tapahtuva *GPU-laskenta*. Tutkimuksessa *Toward a new approach for sorting extremely large data files in the big data era* Shatnawi ja muut (2018) vertailevat keskenään lomituslajittelun CPU- ja GPU-toteutuksia ja havaitsevat, että GPU-toteutus voi olla jopa satoja kertoja nopeampi kuin CPU-toteutus.

Lähdeluettelo

Bae, S. (2019). Big-O Notation. In *JavaScript Data Structures and Algorithms* (pp. 1–11). Apress. https://doi.org/10.1007/978-1-4842-3988-9_1

Buradagunta, S., Bodapati, J., Mundukur, N., & Salma, S. (2020). Performance Comparison of Sorting Algorithms with Random Numbers as Inputs. *Ingénierie Des Systèmes d'Information*, 25(1), 113–117. https://doi.org/10.18280/isi.250115

Heineman, G., Pollice, G., & Selkow, S. (2016). *Algorithms in a nutshell* (Second edition.). O'Reilly.

Knuth, D. (2014). *Art of Computer Programming, Volume 3, The: Sorting and Searching*. Addison-Wesley.

Setiawan, R. (2016). Comparing sorting algorithm complexity based on control flow structure. *2016 International Conference on Information Management and Technology (ICIMTech)*, 224–228. https://doi.org/10.1109/ICIMTech.2016.7930334

Shatnawi, A., AlZahouri, Y., Shehab, M., Jararweh, Y., & Al-Ayyoub, M. (2018). Toward a new approach for sorting extremely large data files in the big data era. *Cluster Computing, 22*(3), 819–828. https://doi.org/10.1007/s10586-018-2860-1

Singhal, S. (2018). *Analysis and Design of Algorithms: A Beginner's Hope*. BPB Publications.

Verma, R., & Singh, J. (2015). A Comparative Analysis of Deterministic Sorting Algorithms based on Runtime and Count of Various Operations. *International Journal of Advanced Computer Research*, 5(21), 380–385.

Zhang, H., Meng, B., & Liang, Y. (2016). *Sort Race*. https://arxiv.org/abs/1609.04471