Tampereen yliopisto

Iiro Koskinen

JÄRJESTYSALGORITMIEN TEHOKKUUDEN VERTAILU

Pika-, lomitus- ja kuplalajittelun erot

Kandidaatintutkielma
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Maarit Harsu
Marraskuu 2023

TIIVISTELMÄ

Iiro Koskinen: Järjestysalgoritmien tehokkuuden vertailu. Pika-, lomitus- ja kuplalajittelun erot
Kandidaatintutkielma
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma, tietotekniikka
Marraskuu 2023

Järjestysalgoritmien merkitys yhteiskunnalle on kasvanut viime vuosina datan määrän räjähdysmäisen kasvun takia. Datan käsittely vaatii usein datan järjestämistä, sillä monet algoritmit vaativat syötteeksi ottamansa datan olevan tietyssä järjestyksessä. Näin ollen järjestysalgoritmin suoritusaika voi vaikuttaa paljon itse algoritmia suurempien kokonaisuuksien suoritustehoon. Järjestysalgoritmeja on kuitenkin monia, ja niiden tehokkuuksissa voi olla suuria eroja. Tämän takia oikean järjestysalgoritmin valinta ei aina ole helppoa.

Tämän työn tarkoitus on helpottaa valintaa työssä käsiteltävien algoritmien osalta. Työssä vertaillaan kolmea järjestysalgoritmia, keskittyen erityisesti niiden tehokkuuteen ja tehokkuuksien eroihin. Algoritmit ovat pikalajittelu (engl. quicksort), lomituslajittelu (merge sort) ja kuplalajittelu (bubble sort). Vertailu on toteutettu osittain kirjallisuuskatsauksena ja osittain empiirisenä tutkimuksena. Kirjallisuuskatsauksessa käsitellään aiempien, kyseisiä algoritmeja vertailevien tutkimusten tuloksia. Omassa tutkimuksessa mitattiin algoritmien tehokkuuksia eri kokoisilla standardoiduilla datajoukoilla. Joukot koostuivat satunnaisesta positiivisesta datasta, satunnaisesta negatiivisesta datasta, järjestetystä ja käänteisesti järjestetystä datasta sekä lähes järjestetystä datasta.

Tutkielman perusteella kuplalajittelu on keskimääräisessä tapauksessa huomattavasti hitaampi kuin pikalajittelu tai lomituslajittelu. Pika- ja lomituslajittelun välinen ero ei ollut yhtä selvä. Lomituslajittelu oli suurimmassa osassa tutkimuksia nopeampi kuin pikalajittelu. Kuitenkin osassa tutkimuksista järjestys oli päinvastainen. Pikalajittelun suurin heikkous on sen tehokkuus huonoimmassa mahdollisessa tilanteessa. Tällöin sen tehokkuus on jopa kuplalajittelua huonompi. Lomituslajittelun tehokkuus taas pysyi lähes vakiona datan muodosta riippumatta. Näin ollen lomituslajittelua voidaan pitää yleisesti parhaana tutkielmassa käsitellyistä algoritmeista.

Avainsanat: algoritmi, järjestysalgoritmi, tehokkuus, pikalajittelu, kuplalajittelu, lomituslajittelu, quicksort, merge sort, bubble sort

SISÄLLYSLUETTELO

.

1. JOHDANTO

Monet tietojenkäsittelytieteilijät pitävät järjestämistä jopa algoritmien tutkimuksen keskeisimpänä osa-alueena. Tätä väitettä tukee useampi asia. Monesti järjestysalgoritmeja ajatellaan vain yksinkertaisimmassa tilanteessa, jossa niitä käytetään. Esimerkki tästä voisi olla pankki, joka käsittelee tilitapahtumia asiakasnumeron perusteella. Järjestysalgoritmit ovat kuitenkin usein myös toisten algoritmien alarutiineja, jotka ne suorittavat ennen itse algoritmin suoritusta. Esimerkki tästä voisi olla ohjelma, joka hahmontaa (engl. render) pinottuja graafisia esineitä. Ennen hahmonnuksen aloittamista esineet pitää järjestää niin, että ohjelma voi piirtää esineet alhaalta ylöspäin. [1, Luku 2]

Tässä tutkielmassa vertaillaan keskenään kolmen järjestysalgoritmin tehokkuutta ja pyritään löytämään niistä paras. Hajota ja hallitse on yleinen algoritmien suunnitteluperiaate. Tutkielmassa käsiteltävistä kolmesta algoritmista kaksi on toteutettu tällä periaatteella. Näin päästään näkemään onko kyseisestä suunnitteluperiaatteesta hyötyä tehokkuuden kannalta. Tutkielman tavoitteena on selvittää, mikä näistä kolmesta järjestysalgoritmista on yleisessä tapauksessa tehokkain. Tutkielmassa keskitytään algoritmien todelliseen, aikayksiköissä mitattuun tehokkuuteen asymptoottisen tehokkuuden sijaan. Tehokkuuksia vertaillaan sekä kirjallisuuslähteiden pohjalta että itse toteutettuna vertailuna. Vertailu tehdään C++-kielellä kirjoitettujen ohjelmien pohjalta. Algoritmeja verrataan pienellä, keskisuurella ja suurella datan määrällä.

Luvussa 2 käsitellään yleisesti algoritmeja, käydään läpi asymptoottisen tehokkuuden käsite sekä perehdytään tarkemmin hajota ja hallitse-suunnitteluperiaatteeseen. Luvussa 3 kuvataan tutkittavien algoritmien toimintaperiaatteet ja niiden asymptoottiset tehokkuudet. Luvuissa 4 ja 5 vertaillaan luvussa 3 esiteltyjen algoritmien tehokkuuksia. Luku 4 perustuu aiempiin tutkimuksiin. Luku 5 käsittelee omaa tutkimustani. Luku 6 on yhteenvetoluku.

2. YLEISTÄ ALGORITMEISTA

Algoritmi on sarja askeleita, joiden avulla ratkaistaan jokin ongelma. Usein algoritmeja käytetään kuvaamaan jotain loogista tai matemaattista prosessia. Jotta prosessi voidaan lukea algoritmiksi, tulee sen täyttää seuraavat kolme ehtoa:

- Prosessin tulee olla äärellinen, eli sen tulee lopulta ratkaista ratkomansa ongelma.
- Prosessin tulee olla tarkasti määritelty. Askeleiden tulee olla tarkkoja sekä ymmärrettäviä. Tämä on erittäin tärkeää varsinkin, kun on kyse tietokoneella suoritettavista algoritmeista.
- Prosessin tulee olla tehokas, eli sen pitää toimia kaikissa tilanteissa ja ratkaista aina sille annettu ongelma. [2, Luku 1]

Tässä luvussa käsitellään algoritmien asymptoottista tehokkuutta sekä algoritmien suunnitteluperiaatteita. Luvussa 2.1 käydään läpi asymptoottinen tehokkuus sekä siihen liittyviä termejä. Luku 2.2 käsittelee hajota ja hallitse- periaatetta, joka on yksi yleisimmistä algoritmien suunnitteluperiaatteista.

2.1 Asymptoottinen tehokkuus

Algoritmin asymptoottinen suoritusaika kertoo, kuinka paljon algoritmin suoritusaika kasvaa syötteiden määrän kasvaessa. Asymptoottinen suoritusaika saadaan, kun algoritmista poistetaan kaikki paitsi suoritusajan kasvun kannalta merkittävin osa. Otetaan esimerkiksi algoritmi, jonka suoritus, syötteiden määrän ollessa n, vaatii $5n^2 + 300n + 100$ konekäskyä. Kun syötteiden määrä kasvaa, kasvaa myös konekäskyjen määrä. Pienillä n:n arvoilla termi $300n$ on suurempi kuin $5n^2$. Kuitenkin syötteiden määrän kasvaessa riittävän suureksi, ohittaa $5n^2$ $300n$:n, tässä tapauksessa kun $n > 60$. Tämä on totta riippumatta termien kertoimista. Jos konekäskyjen määrä on muotoa $an^2 + bn + c$ ja $a > 0$, löytyy aina jokin n:n arvo, jolla $an^2 > bn$. Termi an^2 on siis suorituksen hidastumisen kannalta merkittävin termi. Kun jätämme pois vähemmän merkitsevät muuttujat sekä kertoimet, voimme keskittyä asymptoottisen suoritusajan kannalta olennaisimpaan asiaan eli sen kasvuun syötteen kasvaessa. Esimerkin algoritmin asymptoottiseksi suoritusajaksi jää siis vain n^2. [3]

Asymptoottisen tehokkuuden kuvaamiseen on olemassa kolme yleisesti käytettyä asymptoottista notaatiota: omega-notaatio(Ω), theeta-notaatio(Θ) ja O-notaatio[3].

Omega-notaatiota käytetään, kun halutaan tietää kuinka kauan algoritmin suorittaminen vähintään kestää. Ω-notaatio kertoo siis mikä on algoritmin tehokkuus parhaassa mahdollisessa tilanteessa. Jos algoritmin asymptoottiseksi tehokkuudeksi on ilmoitettu esimerkiksi Ω(log(n)), algoritmin suorittaminen vaatii aina vähintään log(n) konekäskyä.[4] Jos taas ollaan kiinnostuneita algoritmin tehokkuudesta huonoimmassa mahdollisessa tilanteessa, voidaan hyödyntää O-notaatiota. O-notaatio kertoo ehdottoman ylärajan algoritmin asymptoottiselle tehokkuudelle. Esimerkiksi jos algoritmin asymptoottinen tehokkuus on $O(n^2)$, algoritmin suorittaminen ei koskaan vaadi enempää kuin n^2 konekäskyä.[5]

Algoritmien suoritukseen kuluva aika voi vaihdella paljonkin, vaikka syötteiden määrä pysyisikin samana. Tämän takia Ω- ja O-notaatio eivät kumpikaan kerro koko totuutta algoritmin tehokkuudesta. Usein algoritmien tehokkuutta vertaillessa hyödyllisin notaatio on theeta-notaatio. Θ-notaatio kertoo algoritmin asymptoottisen tehokkuuden keskimääräisessä tapauksessa[6]. Algoritmia valitessa voidaan päätyä tilanteeseen, jossa kahden algoritmin asymptoottiseksi tehokkuudeksi on ilmoitettu $O(n^2)$. Tässä tapauksessa voi olla hyödyllistä vertailla algoritmien tehokkuuksia Θ-notaatiolla. Jos toisen algoritmin asymptoottinen tehokkuus on $\Theta(\mathrm{n}^2)$ ja toisen $\Theta\big(\mathrm{nlog(n)}\big)$, kannattaa näistä valita jälkimmäinen, sillä se on keskimäärin nopeampi kuin ensimmäinen algoritmi.

2.2 Hajota ja hallitse

Hajota ja hallitse (engl. divide and conquer) on algoritmien suunnitteluperiaate, joka koostuu kolmesta osasta:

1. ongelman jakamisesta pienempiin osiin
2. osaongelmien ratkaisemisesta
3. ratkottujen osien yhdistämisestä.

Hajota ja hallitse-algoritmit toteutetaan usein rekursiota hyödyntämällä. [7] Rekursiolla tarkoitetaan tietotekniikassa tilannetta, jossa funktio kutsuu itseään. Hajota ja hallitse-algoritmin tapauksessa funktio jakaa syötteensä pienempiin osiin ja kutsuu sen jälkeen itseään, antaen syötteeksi nämä pienemmät osat.

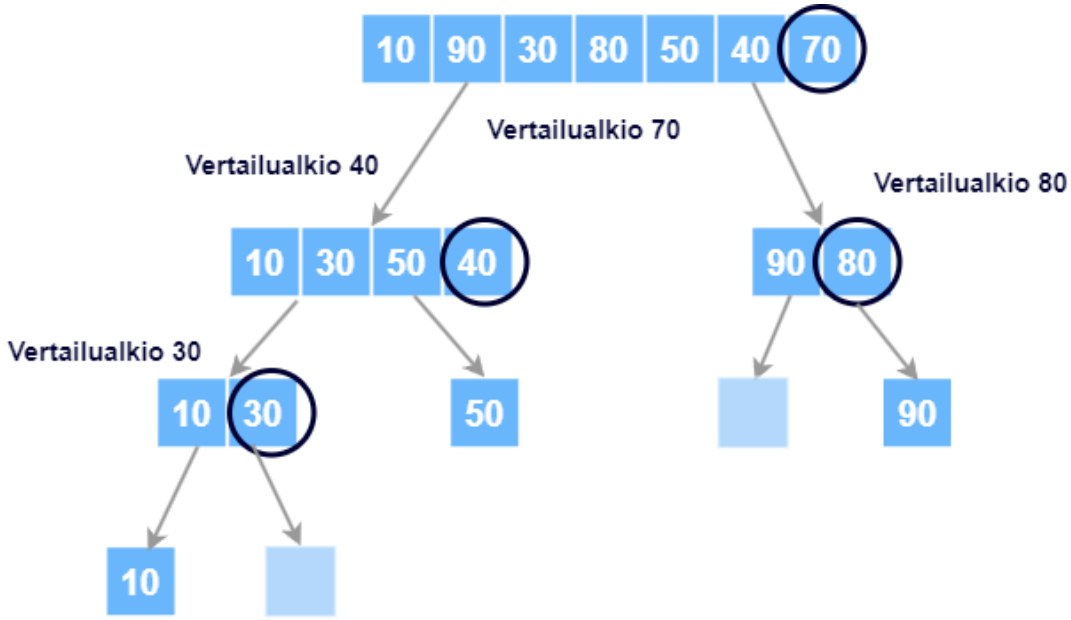
Järjestäminen on hyvä esimerkki ongelmasta, joka voidaan ratkaista hajoa ja hallitse -menetelmällä. Taulukon järjestäminen voidaan ajatella alitaulukkojen järjestämisenä. Kun kaikki alitaulukot on järjestetty, on myös alkuperäinen taulukko järjestetty. Seuraavassa luvussa käsiteltävistä kolmesta järjestysalgoritmista kaksi on toteutettu hajota ja hallitse -suunnitteluperiaatteella.

3. VERTAILTAVAT JÄRJESTYSALGORITMIT

Seuraavissa alaluvuissa käsitellään työssä tarkasteltavia järjestysalgoritmeja. Algoritmit ovat pikalajittelu (engl. quicksort), lomituslajittelu (engl. merge sort) ja kuplalajittelu (engl. bubble sort). Näistä varsinkin pikalajittelu ja lomituslajittelu ovat laajalti käytettyjä. Kuplalajittelu taas on yksinkertaisempi algoritmi, ja se on usein ensimmäinen järjestysalgoritmi, joka oppimateriaaleissa otetaan esille.

3.1 Pikalajittelu

Pikalajittelu toimii hajota ja hallitse -toimintaperiaatteella[8]. Algoritmi on usein tehokkainta toteuttaa rekursion avulla. Sen toimintaperiaate perustuu listan osiin jakamiseen vertailualkion perusteella. Kaikki vertailualkiota pienemmät alkiot laitetaan yhteen osioon ja vertailualkiota suuremmat alkiot laitetaan toiseen osioon. Vertailualkio itsessään laitetaan oikealle paikalle alkuperäiseen listaan. Tämä prosessi toistetaan kaikille syntyville osille, kunnes alkiot ovat järjestyksessä.[9, Luku 6.10] Vertailualkion valintaan on monia vaihtoehtoja. Yksi vaihtoehto on valita aina tietty alkio, esimerkiksi ensimmäinen tai keskimmäinen alkio. Toinen vaihtoehto on satunnaisen alkion valitseminen.[10] Kuvassa 1 on esimerkki pikalajittelun toiminnasta, kun vertailualkioksi valitaan viimeinen alkio.



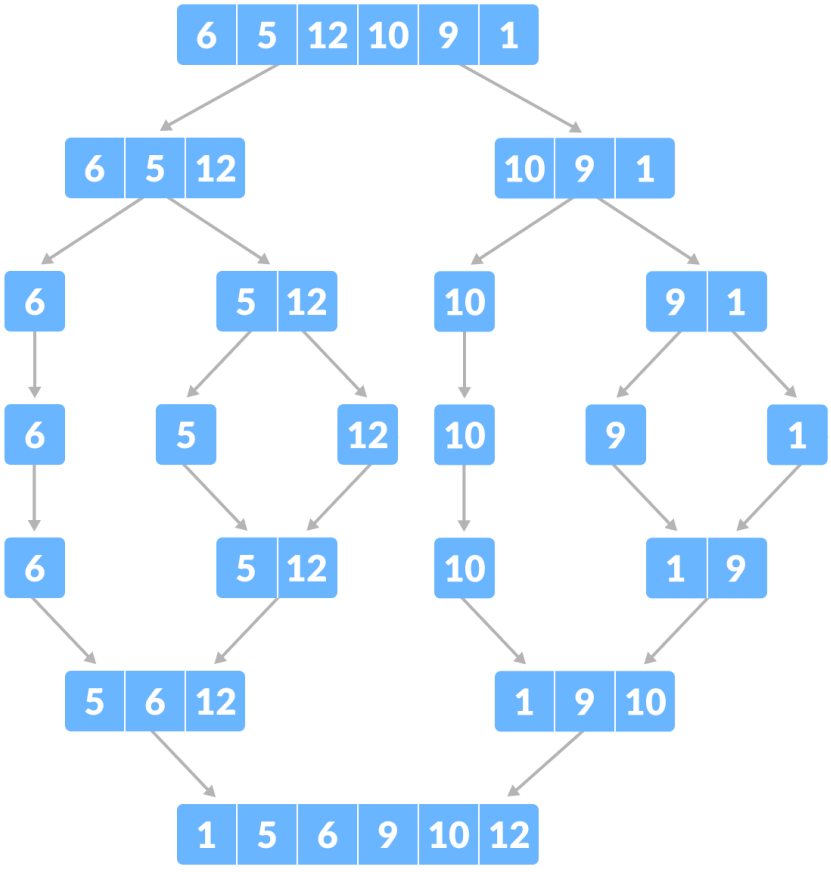
Kuva 1. *Pikalajittelun toimintaperiaate, perustuu lähteeseen [8].*

Pikalajittelun asymptoottinen tehokkuus keskimääräisessä tapauksessa on $\theta(n * log(n))$. Tämä on myös sen tehokkuus parhaassa mahdollisessa tapauksessa, jossa valittu vertailualkio on suuruudeltaan keskimmäinen luku. Huonoimmassa tapauksessa

vertailualkioksi valikoituu suurin tai pienin luku. Tällöin algoritmin asymptoottinen tehokkuus on $O(n^2)$. [8]

3.2 Lomituslajittelu

Lomituslajittelu toimii myös hajota ja hallitse -toimintaperiaatteella. Lomituslajittelu toteutetaan samoin yleensä rekursion avulla. Sen toimintaperiaate on yksinkertainen. Algoritmi jakaa järjestettävät alkiot kahteen samankokoiseen osaan. Tämä toistetaan niin moneen kertaan, että jokaisessa osiossa on enää yksi alkio. Tämän jälkeen algoritmi alkaa yhdistämään osioita asettaen alkiot suuruusjärjestyksessä uuteen osioon. Tätä toistetaan, kunnes jäljellä on enää yksi osio, joka sisältää kaikki alkuperäiset alkiot järjestettynä. [11] Algoritmi tarvitsee lisämuistia alkuperäisten alkioiden koon verran. Tehokkain tapa toteuttaa algoritmi on varata heti sen suorituksen alussa tämä tarvittava määrä muistia. [12, Luku 5] Kuvassa 2 on havainnollistettu lomituslajittelun toimintaperiaatetta



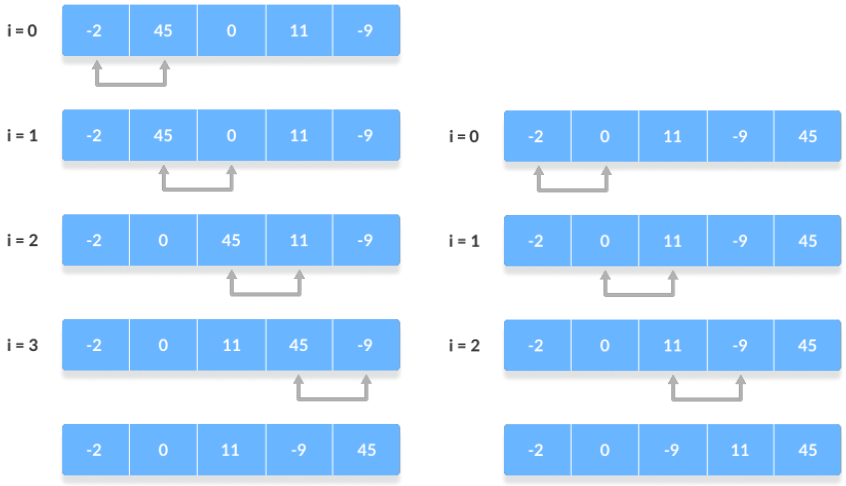
Kuva 2. *Lomituslajittelun toimintaperiaate [13]*

Lomituslajittelun asymptoottinen tehokkuus kaikissa tapauksissa on $O(n * log(n))$[11]. Se on siis parhaassa ja keskimääräisessä tapauksessa yhtä tehokas kuin pikalajittelu. Huonoimmassa tapauksessa lomituslajittelu on kuitenkin pikalajittelua tehokkaampi.

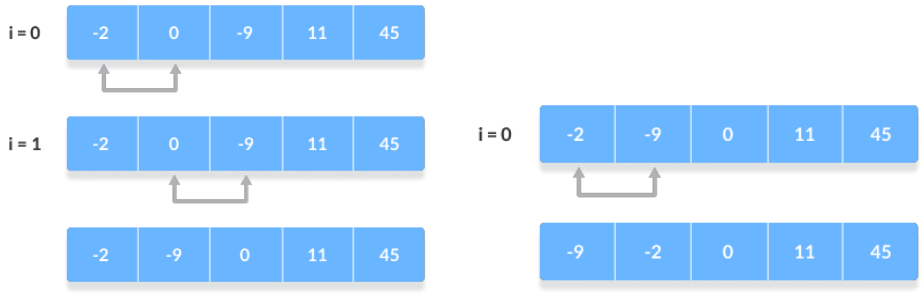
Tällä voi olla merkitystä esimerkiksi tilanteessa, jossa algoritmin tehokkuus on erittäin kriittistä. Tällöin lomituslajittelu on turvallisempi valinta kuin pikalajittelu.

3.3 Kuplalajittelu

Kuplalajittelu-algoritmi vertailee vierekkäisiä alkioita ja tarvittaessa vaihtaa niiden paikkaa keskenään. Prosessin aikana pienet arvot liikkuvat kohti toista päätä ja suuret arvot toista. Algoritmi saa nimensä tästä toimintaperiaatteestaan, joka vastaa vesitankissa oman tasonsa löytäviä ilmakuplia. Algoritmi koostuu kahdesta sisäkkäisestä for-silmukasta, jonka takia sitä ei suositella käytettävän suurilla datan määrillä.[9, Luku 5.2] Alapuolella kuvissa 3 ja 4 on kuvattu kuplalajittelun toimintaperiaate. Jokainen kuva esittää aina yhden suorituskierroksen tapahtumia.



Kuva 3. *Kuplalajittelun toimintaperiaate kahdella ensimmäisellä kierroksella, perustuu lähteeseen[14].*



Kuva 4. *Kuplalajittelun toimintaperiaate kahdella jälkimmäisellä kierroksella, perustuu lähteeseen[14].*

Kuplalajittelun tehokkuuden kannalta paras tilanne on silloin, kun alkiot ovat jo valmiiksi oikeassa järjestyksessä. Tällöin algoritmin asymptoottinen tehokkuus on $O(n)$. Tämän

perusteella voisi siis olettaa kuplalajittelun olevan aiemmin esiteltyjä algoritmeja parempi. Kuplalajittelun asymptoottinen tehokkuus on kuitenkin sekä keskimääräisessä että huonoimmassa tilanteessa $O(n^2)$. Eli huonoimmassa tapauksessa kuplalajittelu on yhtä tehokas kuin pikalajittelu, mutta hitaampi kuin lomituslajittelu. Keskimääräisessä tapauksessa kuplalajittelu on taas hitaampi kuin kumpikaan näistä kahdesta muusta algoritmista.[15]

4. AIEMMAT TUTKIMUKSET

Tässä luvussa käsitellään tutkimuksia, joissa vertaillaan pikalajittelun, lomituslajittelun ja kuplalajittelun todellisia tehokkuuksia. Kuplalajittelun tehokkuutta vertailevia tutkimuksia löytyi verrattain vähän. Oletettavasti siitä syystä, että kuplalajittelun asymptoottinen tehokkuus on huomattavasti huonompi kuin muiden tämän tutkimuksen algoritmien, kuten luvussa 3 mainittiin. Luvussa 4.1 käsitellään neljää tutkimusta, joissa on vertailtu kaikkia kolmea algoritmia. Luvussa 4.2 käsitellään vielä kahta tutkimusta, joissa on vertailtu vain pikalajittelua ja lomituslajittelua.

4.1 Kaikkia algoritmeja käsittelevät tutkimukset

Ensimmäinen tarkasteltu tutkimus on Faujdarin ja Ghreran tutkimus, jossa vertaillaan yhdeksän eri järjestysalgoritmin tehokkuuksia standardoidulla datalla. Tutkimuksessa vertailtiin algoritmeja neljällä eri tyylisellä datalla: satunnaisella, käänteisesti järjestetyllä, järjestetyllä ja lähes järjestetyllä. Taulukossa 1 näkyy tutkimuksessa saadut tulokset pikalajittelulle, lomituslajittelulle ja kuplalajittelulle. Taulukon ajat ovat mikrosekunteina.[16]

Taulukko 1. *Standardoidun datan järjestämiseen kulunut aika, perustuu lähteeseen [16]*

Datan tyyppi	**Pikalajittelu**	**Lomituslajittelu**	**Kuplalajittelu**
Satunnainen data	1043904	120233	1443038403
Lähes järjestetty data	1219802	54803	398798336
Järjestetty data	1263220	61900	892
Käänteisesti järjestetty data	72089548	69185	1118144524

Taulukosta nähdään lomituslajittelun olleen keskimäärin tehokkain näistä kolmesta algoritmista. Lomituslajittelun tehokkuus parani, kun data oli järjestetty tai lähes järjestetty. Myös käänteinen järjestys nopeutti lomituslajittelun suoritusta. Kuplalajittelu oli paljon muita algoritmeja hitaampi, paitsi silloin kun data oli valmiiksi järjestetty. Siinä tapauksessa kuplalajittelu oli todella paljon nopeampi kuin muut algoritmit. Pikalajittelu oli kaikissa tilanteissa hitaampi kuin lomituslajittelu ja kaikissa muissa tilanteissa, paitsi valmiiksi järjestetyn datan kanssa, nopeampi kuin kuplalajittelu. Pikalajittelu kärsi suuresti, kun data oli käänteisesti järjestettyä.

Toisena tarkastellaan Fenyin et al. tutkimusta, jossa tutkitaan vertailuperusteisten (engl. comparison based) ja ei-vertailuperusteisten (engl. non-comparison based) algoritmien

tehokkuuksia. Tässä tutkielmassa ollaan kiinnostuneita vain vertailuperusteisista algoritmeista, jotka kyseisessä tutkimuksessa olivat juuri pika-, lomitus- ja kuplalajittelu. Tutkimuksen vertailu toteutettiin tietokoneella, jonka suorittimen nopeus oli 2,7 GHz ja muistin määrä oli 8 GB. Taulukossa 2 näkyy tutkimuksen tulokset pika-, lomitus- ja kuplalajittelulle. Taulukon ajat ovat mikrosekunteina. [17]

Taulukko 2. *Datan järjestämiseen kulunut aika, perustuu lähteeseen [17]*

Alkioita	Pikalajittelu	Lomituslajittelu	Kuplalajittelu
1000	68400	73000	334000
5000	1448000	1318000	8517000
10000	6343000	5516000	34912000
20000	32029000	28239000	144865000
30000	83186000	64896000	332707000
35000	122690000	93527000	469331000

Taulukosta nähdään tälläkin kertaa lomituslajittelun olleen keskimäärin tehokkain vertailtavista algoritmeista. Pienimmällä alkioiden määrällä pikalajittelu oli nopeampi kuin lomituslajittelu. Lopuissa tilanteissa lomituslajittelu oli nopeampi näistä kahdesta, vaikka erot eivät olleet suuria. Kuplalajittelu oli jokaisessa tilanteessa huomattavasti muita algoritmeja hitaampi.

Kolmantena käsitellään Dharmajee Raon ja Rameshin tutkimusta, jossa vertailtiin järjestysalgoritmien tehokkuuksia sekä positiivisilla että negatiivisilla luvuilla. Vertailu toteutettiin tietokoneella, jonka suorittimen nopeus oli 2,4 GHz ja muistin määrä 2 GB. Taulukossa 3 on tutkimuksen tulokset pika-, lomitus- ja kuplalajittelulle positiivisella datalla ja taulukossa 4 tulokset negatiivisella datalla. Molempien taulukoiden ajat ovat mikrosekunteina. [18]

Taulukko 3. *Positiivisen datan järjestämiseen kulunut aika, perustuu lähteeseen [18]*

Alkioita	Pikalajittelu	Lomituslajittelu	Kuplalajittelu
1000	300	29300	900
10000	900	117100	2300
30000	5700	1218200	4702000
60000	12700	1748500	18628200
80000	15600	3223000	34581300
99000	19400	2724500	52344200

Taulukosta 3 nähdään pikalajittelun olleen selkeästi tehokkain algoritmi. Kuplalajittelu oli lomituslajittelua nopeampi 10 000 alkioon asti, mutta suuremmilla alkioiden määrillä lomituslajittelu oli reilusti nopeampi.

Taulukko 4. *Negatiivisen datan järjestämiseen kulunut aika, perustuu lähteeseen [18]*

Alkioita	Pikalajittelu	Lomituslajittelu	Kuplalajittelu
1000	300	26300	11710
10000	1800	237900	510880
30000	5800	694900	4612300
60000	12500	2461300	18964800
80000	15600	1873800	32693000
99000	19300	2312800	50824200

Taulukosta 4 nähdään, että negatiivinen data ei pikalajittelun ja lomituslajittelun osalta näytä vaikuttaneen tehokkuuteen lainkaan. Pikalajittelu oli tässäkin tapauksessa selkeästi nopein algoritmi. Kuplalajittelu taas näytti kärsineen negatiivisesta datasta pienemmillä alkioiden määrällä. Tällä kertaa lomituslajittelu oli kuplalajittelua tehokkaampi riippumatta alkioiden määrästä.

Neljäntenä käsitellään Buradaguntan et al. tutkimusta, jossa vertailtiin edellisen tutkimuksen tavoin järjestysalgoritmien tehokkuuksia positiivisella ja negatiivisella datalla. Taulukossa 5 on tutkimuksen tulokset pika-, lomitus- ja kuplalajitteluille positiivisella datalla ja taulukossa 4 samat tulokset negatiivisella datalla. Taulukoiden ajat ovat mikrosekunteina. [19]

Taulukko 5. *Positiivisen datan järjestämiseen kulunut aika, perustuu lähteeseen [19]*

Alkioita	Pikalajittelu	Lomituslajittelu	Kuplalajittelu
100	200	0	300
1000	1300	2000	6200
2000	1700	6000	11400
5000	11000	8000	86600
10000	22500	30000	352600
20000	49600	40000	1488000
40000	74600	103000	5945800
60000	120700	169000	13573000
80000	128000	239000	24170000
100000	143500	302000	38007000
200000	309700	901000	150510000

Taulukosta 5 nähdään pikalajittelun olleen selkeästi nopein algoritmi. Lomituslajittelu oli oletettavasti nopein 100 alkiolla, koska sen ajaksi oli merkitty vain nolla. Muissa tilanteissa se oli pikalajittelua hitaampi. Kuplalajittelu oli hitain algoritmi riippumatta alkioiden määrästä.

Taulukko 6. *Negatiivisen datan järjestämiseen kulunut aika, perustuu lähteeseen [19]*

Alkioita	Pikalajittelu	Lomituslajittelu	Kuplalajittelu
100	0	0	0
1000	0	2000	6000
2000	0	2000	29000
5000	4000	10000	84000
10000	8000	12000	388000
20000	17000	43000	1494000
40000	32000	57000	5978000
60000	44000	124000	13460000
80000	63000	125000	24140000
100000	72000	172000	37620000
200000	131000	296000	150200000

Taulukosta 6 nähdään, että negatiivisella datalla algoritmien keskeinen järjestys pysyi samana kuin positiivisella datalla. Kuplalajittelun tehokkuuteen negatiivisella datalla ei näytä olleen mitään merkitystä. Pikalajittelu ja lomituslajittelu taas nopeutuivat huomattavasti, mutta pikalajittelu oli edelleen nopein.

4.2 Vain pika- ja lomituslajittelua käsittelevät tutkimukset

Käydään vielä läpi kaksi tutkimusta, joissa vertaillaan pika- ja lomituslajittelua, mutta ei kuplalajittelua. Ensimmäinen näistä on Esau Taiwon et al. tutkimus, jossa vertailtiin hyvin tarkasti näitä kahta järjestysalgoritmia. Vertailussa oli muun muassa järjestyksen toteutus, muistin käyttö, tehokkuus sekä vakaus, eli säilyykö samansuuruisten alkioiden keskinäinen järjestys algoritmin suorituksen aikana. Tässä tutkielmassa ollaan kuitenkin kiinnostuneita vain algoritmien tehokkuudesta. Alla taulukossa 7 on esitetty tutkimuksen tulokset 10–500 alkiolla ja taulukossa 8 puolestaan 1000–1 000 000 alkiolla. Molempien taulukoiden ajat ovat mikrosekunneissa. [20]

Taulukko 7. *Pienen datan järjestämiseen kulunut aika, perustuu lähteeseen [20]*

Alkioita	Pikalajittelu	Lomituslajittelu
10	989	1383
15	1813	2013
20	2692	3340
25	2677	3564
50	5854	7322
100	10904	13661
200	24704	26567
500	69018	67672

Taulukko 8. *Suuren datan järjestämiseen kulunut aika, perustuu lähteeseen [20]*

Alkioita	Pikalajittelu	Lomituslajittelu
1000	149370	141230
2000	317360	273090
3000	476180	406540
10000	476180	406540
20000	3641300	2687900
40000	6664700	2501200
90000	16104500	6627700
1000000	85053400	46551000

Taulukoista nähdään pikalajittelun olleen nopeampi 200 alkioon asti, jonka jälkeen lomituslajittelu oli nopeampi kaikissa tapauksissa. Alkioiden määrän kasvaessa ero kasvoi, eli pikalajittelu hidastui huomattavasti enemmän. Miljoonan alkion järjestämiseen kulunut aika oli pikalajittelulla jo lähes kaksinkertainen lomituslajittelulla kuluneeseen aikaan verrattuna.

Viimeisenä käsitellään Marcellinon et al. tutkimusta, jossa vertailtiin viiden järjestysalgoritmin tehokkuutta ja muistin käyttöä standardoidulla datalla. Vertailu toteutettiin tietokoneella, jonka prosessorin nopeus oli 1,10 GHz ja muistin määrä oli 16 GB. Alla taulukossa 9 on vertailun tulokset pika- ja lomituslajittelulle. Taulukon ajat ovat mikrosekunneissa. [21]

Taulukko 9. *Datan järjestämiseen kulunut aika, perustuu lähteeseen [21]*

Alkioita	Pikalajittelu	Lomituslajittelu
2500	47897	70964
5000	84829	160250
7500	104820	173030
11000	204360	371740

Taulukosta nähdään pikalajittelun olleen nopeampi kaikissa tapauksissa. 5000 alkiolla ero on lähes kaksinkertainen. Tätä voitaneen pitää jossain määrin poikkeavana. Kuitenkin datan määrän kasvaessa ero näyttää kasvavan. Eli lomituslajittelu on hitaampi, minkä lisäksi se myös hidastuu pikalajittelua nopeammin.

5. OMA TUTKIMUS

Edellisen luvun tutkimusten perusteella on oletettavissa kuplalajittelun olevan huomattavasti muita hitaampi algoritmi. Pikalajittelun ja lomituslajittelun järjestys taas vaihteli tutkimusten välillä. Näiden tietojen pohjalta lähdin toteuttamaan omia mittauksiani.

5.1 Mittausjärjestelyt

Algoritmeja vertailtiin satunnaisesti luodulla datalla, joka koostui kokonaisluvuista. Dataa oli viittä eri tyyppiä:

1. positiivisista satunnaisessa järjestyksessä olevista alkioista koostuva data
2. positiivisista valmiiksi järjestetyistä alkioista koostuva data
3. positiivisista lähes järjestetyistä alkioista koostuva data
4. positiivisista käänteisesti järjestetyistä alkioista koostuva data
5. negatiivisista satunnaisessa järjestyksessä olevista alkioista koostuva data.

Kaikista viidestä datatyypistä luotiin kolme eri kokoista taulukkoa. Pienin taulukko oli 100-alkioinen, keskikokoinen 10 000-alkioinen ja suurin 200 000-alkioinen.

Vertailu suoritettiin pöytäkoneella, jonka suoritin on Intel Core i5-6600K ylikellotettuna 4,25 GHz:iin. Koneessa on 16 GB muistia. Käyttöjärjestelmänä on Windows 10. Ohjelmat suoritettiin Microsoft Visual Studio Code -editorin kautta. Prosessien suorituksen katkeamista vuorontajan (engl. scheduler) tekemien päätösten takia ei pystytty estämään. Tämä on huomioitu suorittamalla jokainen järjestäminen kolmeen kertaan ja laskemalla näiden kolmen suorituskerran keston keskiarvo.

Järjestettävät taulukot luotiin C++:n standardikirjaston funktiota rand() hyödyntämällä. Siemenlukuna käytettiin oletusarvoa. Käytetty data oli siis jokaisella suorituskerralla täysin identtistä. Tähän lähtödataan tehdyt muutokset on selitetty tarkemmin seuraavassa luvussa jokaisen vertailun kohdalla. Ajanotto toteutettiin C++:n standardikirjaston chrono-alikirjaston avulla.

Käytettyjen algoritmien toteutukset löytyvät liitteistä. Liite 1 on pikalajittelu, liite 2 lomituslajittelu ja liite 3 kuplalajittelu. Pikalajittelun vertailualkio voidaan valita monella tapaa. Tällä kertaa vertailualkioksi valitaan aina viimeinen alkio. Vertailualkion valinnalla voi olla vaikutusta algoritmin suoritusaikaan ja se tullaan näkemään mittauksissa.

5.2 Mittaukset ja niiden tulokset

Ensimmäisessä mittauksessa satunnaisesti luotuun dataan ei koskettu ennen järjestämistä. Alla olevasta taulukosta löytyy järjestämiseen kulunut aika mikrosekunteina.

Taulukko 10. *Satunnaisen datan järjestämiseen kulunut aika mikrosekunteina*

Alkioita	Pikalajittelu	Lomituslajittelu	Kuplalajittelu
100	6,50	7,87	19,27
10000	1201	1140	224926
200000	30167	28906	92153000

Taulukosta nähdään, että pikalajittelu ja lomituslajittelu ovat tehokkuudeltaan hyvin samankaltaisia. Kuplalajittelu taas on huomattavasti hitaampi, varsinkin alkioiden määrän kasvaessa. Tämä oli toki odotettavissakin, sillä sekä pikalajittelun että lomituslajittelun asymptoottinen tehokkuus keskimääräisessä tapauksessa on $\Theta(n * \log(n))$. Kuplalajittelun asymptoottinen tehokkuus keskimääräisessä tapauksessa taas on $\Theta(n^2)$.

Toisessa mittauksessa sama data järjestettiin C++:n standardikirjaston funktiolla sort(). Tämän jälkeen järjestetty data annettiin algoritmeille syötteeksi. Alla olevasta taulukosta löytyy mittauksen tulokset.

Taulukko 11. *Valmiiksi järjestetyn datan järjestämiseen kulunut aika mikrosekunteina*

Alkioita	Pikalajittelu	Lomituslajittelu	Kuplalajittelu
100	32,00	5,10	9,97
10000	276562	665	87616
200000	-	16447	34691000

Pikalajittelu hidastui tässä tilanteessa todella paljon. 200 000 alkiolla muisti loppui kesken (segmentation fault). Syy tähän on valitussa pikalajittelun toteutuksessa. Vertailualkioksi valitaan aina viimeinen alkio, joka on tässä tapauksessa aina suurin alkio. Tämä on pikalajittelun tehokkuuden kannalta huonoin tapaus[10]. Lomituslajittelun käyttämä aika lähes puolittui verrattuna järjestämättömään dataan. Se oli algoritmeista selkeästi nopein.

Toisin kuin Faujdarin ja Ghreran tutkimuksessa [16] omassa mittauksessani kuplalajittelu ei ollut erityisen nopea. Syy tähän lienee kuplalajittelun toteutusten erossa. Kuplalajittelu on mahdollista toteuttaa myös tavalla, jossa suoritus keskeytetään, jos ensimmäisellä iteraatiolla ei tapahdu muutoksia [14]. Käyttämässäni toteutuksessa tätä ei ole huomioitu. Tästä huolimatta kuplalajittelu oli pikalajittelua nopeampi.

Kolmannessa mittauksessa data järjestettiin taas sort()-funktiolla. Tämän jälkeen kahden mielivaltaisesti valitun alkion paikkaa vaihdettiin keskenään. Tämä lähes järjestetty data annettiin algoritmeille syötteeksi. Taulukossa 12 on mittauksen tulokset.

Taulukko 12. *Lähes järjestetyn datan järjestämiseen kulunut aika mikrosekunteina*

Alkioita	Pikalajittelu	Lomituslajittelu	Kuplalajittelu
100	21,57	5,73	10,33
10000	276562	641	87125
200000	-	16618	34726000

Taulukosta nähdään pikalajittelun olleen hyvin hidas tälläkin kertaa. 200 000 alkiolla ongelmaksi muodostui jälleen muistin loppuminen. Lomituslajittelu toimi tälläkin kertaa nopeimmin ja alkuperäistä mittausta nopeammin. Kuplalajittelu oli jälleen pikalajittelua nopeampi, mutta reilusti hitaampi kuin lomituslajittelu.

Neljännessä mittauksessa data järjestettiin käänteisesti sort()-funktiota hyödyntäen. Käänteisesti järjestetty data annettiin algoritmeille syötteenä. Tulokset löytyvät alla olevasta taulukosta.

Taulukko 13. *Käänteisesti järjestetyn datan järjestämiseen kulunut aika mikrosekunteina*

Alkioita	Pikalajittelu	Lomituslajittelu	Kuplalajittelu
100	22,57	4,93	16,40
10000	187386	656	155139
200000	-	19288	61398000

Pikalajittelussa muisti loppui jälleen kesken 200 000 alkion syötteellä. Lomituslajittelu oli tässäkin tapauksessa nopeampi kuin ensimmäisessä mittauksessa. Kuplalajittelu oli hitaampi kuin kahdessa edellisessä mittauksessa, mutta kuitenkin nopeampi kuin pikalajittelu.

Viimeisessä mittauksessa saman satunnaisen datan alkioiden etumerkki vaihdettiin negatiiviseksi. Tämän jälkeen muuten koskematon data annettiin algoritmeille syötteeksi. Negatiivisen syötteen valinnan syynä oli Buradaguntan et al. tutkimuksessa [19] ollut virhe, jossa kuvaajan mukaan pikalajittelun ja lomituslajittelun keskinäinen järjestys olisi vaihtunut täysin päinvastaiseksi. Selitys oli lopulta vain kuvaajan virheellinen selite. Virheen huomattuani päätin kuitenkin toteuttaa mittauksen myös negatiivisella datalla, nähdäkseni olisiko sillä vaikutusta. Tulokset löytyvät taulukosta 14.

Taulukko 14. *Negatiivisista kokonaisluvuista koostuvan satunnaisen datan järjestämiseen kulunut aika mikrosekunteina*

Alkioita	Pikalajittelu	Lomituslajittelu	Kuplalajittelu
100	6,70	7,93	16,33
10000	1206	1148	157484
200000	30351	29283	61423000

Kuten aiemmin käsitellyissä tutkimuksissakin [19] [18], myös omassa mittauksessani huomattiin, että negatiivisen datan järjestämisen tehokkuudessa ei ole merkittävää eroa positiiviseen dataan verrattuna. Pikalajittelu ja lomituslajittelu olivat hyvin lähellä toisiaan. Pikalajittelu oli hieman nopeampi pienemmällä datalla ja lomituslajittelu taas suurella datalla. Kuplalajittelu oli kaikissa tapauksissa huomattavasti muita algoritmeja hitaampi.

6. YHTEENVETO

Tässä työssä tutkittiin pikalajittelua, lomituslajittelua ja kuplalajittelua. Pääpiirteinä olivat näiden algoritmien tehokkuudet ja niiden väliset erot. Vertailu toteutettiin perustuen relevantteihin kirjallisuuslähteisiin sekä itse toteutettuun empiiriseen tutkimukseen. Työn tavoite oli selvittää, mikä näistä kolmesta algoritmista on yleisessä tapauksessa tehokkain.

Tulosten perusteella kuplalajittelu on yleisessä tilanteessa huomattavasti hitaampi kuin pika- tai lomituslajittelu. Näiden tulosten perusteella hajota ja hallitse -periaatetta voidaan pitää oikein toimivana lähtökohtana järjestysalgoritmien suunnitteluun. Kuplalajittelu on näistä algoritmeista ainut, jota ei ole suunniteltu hajota ja hallitse -periaatteella. Pikalajittelun ja lomituslajittelun välillä valinta ei ole itsestään selvä. Omassa tutkimuksessani sekä suurimmassa osassa muista käsitellyistä tutkimuksista lomituslajittelu oli hieman pikalajittelua tehokkaampi, varsinkin järjestettävien alkioiden määrän kasvaessa suureksi. Kuitenkin useammassa tutkimuksessa pikalajittelu oli lomituslajittelua tehokkaampi myös suurilla datan määrillä. Lopulta yleisessä tilanteessa parempi valinta näistä kahdesta on lomituslajittelu, sillä huonoimmassa tapauksessa pikalajittelu hidastuu todella paljon. Lomituslajittelun tehokkuus taas pysyy lähes samana syötteestä riippumatta.

Tämän tutkielman vertailua voisi viedä pidemmälle huomioimalla esimerkiksi rinnakkaisuuden vaikutuksen tehokkuuteen. Ainakin pikalajittelu ja lomituslajittelu voidaan toteuttaa myös rinnakkaisesti. Toinen jatkotutkimusidea voisi olla kuplalajittelun korvaaminen jollain toisella järjestysalgoritmilla. Kuplalajittelua tehokkaamman algoritmin valitseminen tekisi tutkimuksesta vielä relevantimman.

LÄHTEET

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest ja C. Stein, *Introduction to Algorithms*. Cambridge, UNITED STATES: MIT Press, 2009. Saatavissa: http://ebookcentral.proquest.com/lib/tampere/detail.action?docID=3339142. [Viitattu: 19. lokakuuta 2023]

[2] J. P. Mueller ja L. Massaron, *Algorithms For Dummies, 2nd Edition*. Saatavissa: https://learning.oreilly.com/library/view/algorithms-for-dummies/9781119869986/. [Viitattu: 7. lokakuuta 2023]

[3] ”Asymptotic notation”, *Khan Academy*. Saatavissa: https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation. [Viitattu: 7. lokakuuta 2023]

[4] ”Big-Ω (Big-Omega) notation”, *Khan Academy*. Saatavissa: https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation. [Viitattu: 7. lokakuuta 2023]

[5] ”Big-O notation”, *Khan Academy*. Saatavissa: https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation. [Viitattu: 7. lokakuuta 2023]

[6] ”Analysis of Algorithms | Big - Θ (Big Theta) Notation”, *GeeksforGeeks*, 11. heinäkuuta 2021. Saatavissa: https://www.geeksforgeeks.org/analysis-of-algorithms-big-theta-notation/. [Viitattu: 7. lokakuuta 2023]

[7] ”Divide and Conquer Algorithm”. Saatavissa: https://www.programiz.com/dsa/divide-and-conquer. [Viitattu: 18. lokakuuta 2023]

[8] ”QuickSort - Data Structure and Algorithm Tutorials”, *GeeksforGeeks*, 7. tammikuuta 2014. Saatavissa: https://www.geeksforgeeks.org/quick-sort/. [Viitattu: 29. syyskuuta 2023]

[9] H. Schildt, *Java: A Beginner’s Guide*, 9th Edition. McGraw-Hill Education, 2022. Saatavissa: https://www.accessengineeringlibrary.com/content/book/9781260463552. [Viitattu: 29. syyskuuta 2023]

[10] ”Time and Space Complexity Analysis of Quick Sort”, *GeeksforGeeks*, 24. toukokuuta 2023. Saatavissa: https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-quick-sort/. [Viitattu: 11. lokakuuta 2023]

[11] ”Merge Sort - javatpoint”, *www.javatpoint.com*. Saatavissa: https://www.javatpoint.com/merge-sort. [Viitattu: 29. syyskuuta 2023]

[12] G. Heineman, *Learning Algorithms*. Saatavissa: https://learning.oreilly.com/library/view/learning-algorithms/9781492091059/ch05.html. [Viitattu: 28. syyskuuta 2023]

[13] ”Merge Sort (With Code in Python/C++/Java/C)”. Saatavissa: https://www.programiz.com/dsa/merge-sort. [Viitattu: 18. lokakuuta 2023]

[14] ”Bubble Sort (With Code in Python/C++/Java/C)”. Saatavissa: https://www.programiz.com/dsa/bubble-sort. [Viitattu: 18. lokakuuta 2023]

[15] ”Time and Space Complexity Analysis of Bubble Sort”, *GeeksforGeeks*, 22. toukokuuta 2023. Saatavissa: https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/. [Viitattu: 17. lokakuuta 2023]

[16] N. Faujdar ja S. P. Ghrera, ”Analysis and Testing of Sorting Algorithms on a Standard Dataset”, teoksessa *2015 Fifth International Conference on Communication Systems and Network Technologies*, huhti 2015, s. 962–967. doi: 10.1109/CSNT.2015.98. Saatavissa: https://ieeexplore.ieee.org/document/7280062. [Viitattu: 28. lokakuuta 2023]

[17] A. Fenyi, M. Fosu ja B. Appiah, ”Comparative Analysis of Comparison and Non Comparison based Sorting Algorithms”, *IJCA*, vsk. 175, nro 28, s. 22–25, loka 2020, doi: 10.5120/ijca2020920813

[18] D. T. V. Dharmajee Rao ja B. Ramesh, ”Experimental Based Selection of Best Sorting Algorithm”, *International Journal of Modern Engineering Research (IJMER)*, vsk. 2, nro 4, s. 2908–2912, elo 2012.

[19] S. Buradagunta, J. D. Bodapati, N. B. Mundukur ja S. Salma, ”Performance Comparison of Sorting Algorithms with Random Numbers as Inputs”, *Ingénierie des systèmes d’Information*, vsk. 25, nro 1, s. 113–117, 2020, doi: 10.18280/isi.250115

[20] O. Esau Taiwo, A. O. Christianah, A. N. Oluwatobi, K. A. Aderonke ja A. J. kehinde, ”Comparative study of two divide and conquer sorting algorithms: Quicksort and Mergesort”, *Procedia Computer Science*, vsk. 171, s. 2532–2540, tammi 2020, doi: 10.1016/j.procs.2020.04.274

[21] M. Marcellino, D. W. Pratama, S. S. Suntiarko ja K. Margi, ”Comparative of Advanced Sorting Algorithms (Quick Sort, Heap Sort, Merge Sort, Intro Sort, Radix Sort) Based on Time and Memory Usage”, Piscataway: IEEE, 2021, s. 154–160. doi: 10.1109/ICCSAI53272.2021.9609715

LIITE 1: PIKALAJITTELUN TOTEUTUS C++:LLA

```
int ositus(int taulukko[],int alaraja,int ylaraja)
{
    //Valitaan vertailualkio
    int vertailu=taulukko[ylaraja];
    //alarajan indeksi ja arvio vertailualkion oikeasta paikasta
    int i=(alaraja-1);

    for(int j=alaraja;j<=ylaraja;j++)
    {
        //jos nykyinen alkio on pienempi kuin vertailualkio

        if(taulukko[j]<vertailu
    )
        {
        //kasvatetaan pienemmän alkion indeksiä
        i++;
        swap(taulukko[i],taulukko[j]);
        }
    }
    swap(taulukko[i+1],taulukko[ylaraja]);
    return (i+1);
}

// Pikalajittelufunktio

void pikalajittelu(int taulukko[],int alaraja,int ylaraja)
{

    if(alaraja<ylaraja)
    {
        // pi on vertailualkion oikea paikka


        int pi=ositus(taulukko,alaraja,ylaraja);

        //rekursiokutsu
        //vertailualkiota pienemmät alkiot menevät vasemmalle
        //ja suuremmat oikealle
        pikalajittelu(taulukko,alaraja,pi-1);
        pikalajittelu(taulukko,pi+1,ylaraja);
    }
}
```

Perustuu lähteeseen[8]

LIITE 2: LOMITUSLAJITTELUN TOTEUTUS C++:LLA

```
// Yhdistetään kaksi alitaulukkoa yhdeksi taulukoksi
void lomita(int taulukko[], int p, int q, int r) {

  // luodaan L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = taulukko[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = taulukko[q + 1 + j];

  // Talletetaan alitaulukoiden ja päätaulukon indeksit
  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  // Valitaan L:n ja M:n alkioista suurempi, ja siirretään se
  // taulukkoon oikealle paikalle
  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      taulukko[k] = L[i];
      i++;
    } else {
      taulukko[k] = M[j];
      j++;
    }
    k++;
  }

  // Kun L tai M tyhjentyy, lisätään toisen loput alkiot taulukkoon
  while (i < n1) {
    taulukko[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    taulukko[k] = M[j];
    j++;
    k++;
```

```
  }
}

// Jaetaan taulukko osiin, järjestetään ne ja yhdistetään osat
void lomituslajittelu(int taulukko[], int l, int r) {
  if (l < r) {
    // m on kohta, mistä taulukko jaetaan kahtia
    int m = l + (r - l) / 2;

    lomituslajittelu(taulukko, l, m);
    lomituslajittelu(taulukko, m + 1, r);

    // yhdistetään järjestetyt alitaulukot
    lomita(taulukko, l, m, r);
  }
}
```

Perustuu lähteeseen[13]

LIITE 3: KUPLALAJITTELUN TOTEUTUS C++:LLA

```
void kuplalajittelu(int taulukko[], int koko) {

  // silmukka, joka käy läpi jokaisen alkion
  for (int askel = 0; askel < koko; ++askel) {

    // silmukka, jossa alkioita vertaillaan keskenään
    for (int i = 0; i < koko - askel; ++i) {

      // vertaillaan kahta vierekkäistä alkiota
      if (taulukko[i] > taulukko[i + 1]) {

        // jos järjestys ei ole oikea
        // vaihdetaan alkioiden järjestys
        int valiaikainen = taulukko[i];
        taulukko[i] = taulukko[i + 1];
        taulukko[i + 1] = valiaikainen;
      }
    }
  }
}
```

Perustuu lähteeseen[14]