Tavallisimmat lajittelumenetelmät testattavina

TURUN YLIOPISTO
Informaatioteknologian laitos
LuK-tutkielma
Tietojenkäsittelytiede
Tero Yli-Sipilä
Huhtikuu 2013

TURUN YLIOPISTO
Informaatioteknologian laitos
Matemaattis-luonnontieteellinen tiedekunta

YLI-SIPILÄ, TERO: Tavallisimmat lajittelumenetelmät testattavina

LuK-tutkielma: 29 s., 10 liites.
Tietojenkäsittelytiede
Huhtikuu 2013

Joukkojen lajittelemiseen on useita eri lähestymistapoja. Algoritmien aikavaativuudet alkavat vaikuttaa suoritusaikoihin jo muutaman tuhannen alkion lajittelemisessa. Tämän vuoksi algoritmit, jotka jakavat syötteen tehokkaimmin pieniin alijoukkoihin, ovat myös selvästi nopeimpia.

Tässä tutkielmassa esitellään ja vertaillaan kuutta erilaista lajitteluun tarkoitettua algoritmia. Esittelyosiossa kuvataan algoritmien toimintaperiaatteet esimerkkien avulla. Lajittelualgoritmeista on hyvin niukalti kattavia, useita menetelmiä sisältäviä vertailuja. Tämän vuoksi vertailuosiossa algoritmit suoritetaan samalla koneella ja samoilla syötteillä, jotta tulokset ovat vertailukelpoisia keskenään.

Vertailun tuloksissa hajota-ja-hallitse -menetelmiin kuuluvat rekursiivinen ja iteratiivinen pikalajittelu sekä lomituslajittelu ovat selvästi nopeimpia menetelmiä. Pelkkiin yksinkertaisiin vertailuihin perustuvien lisäyslajittelun ja valintalajittelun aikavaativuudet ovat neliöllisiä, mikä tekee näistä käyttökelvottomia syötekoon kasvaessa isoksi. Näitäkin menetelmiä optimoimalla voidaan saada kohtuullisen nopea menetelmä, kuten lisäyslajittelun ideasta kehitetty Shellin lajittelu osoittaa. Sen suorituskyky ei kuitenkaan yllä isoilla syötteillä samaan luokkaan nopeimpien menetelmien kanssa.

Asiasanat: lajittelualgoritmi, vertailu

Sisällys

1. Johdanto

Lajittelumenetelmillä tarkoitetaan algoritmeja, joiden avulla pystyy lajittelemaan keskenään vertailukelpoisia alkioita haluttuun järjestykseen. Tässä tutkielmassa selvitetään viiden tavallisen lajittelumenetelmän toimintaperiaatteet, ja näistä yhdelle esitetään kaksi erilaista versiota. Algoritmeja ja niistä tiettyyn tarkoitukseen tehtyjä yhdistelmiä löytyisi vielä paljon enemmänkin, mutta tässä tutkielmassa keskitytään yleisimmin käytössä oleviin sekä pariin vaihtoehtoiseen lajittelumenetelmään. Algoritmien käyttöä, tehokkuutta ja monimutkaisuutta tarkastellaan satunnaisen ohjelmoijan näkökulmasta. Tämän vuoksi ensimmäisenä esitellään algoritmien toimintaperiaatteet.

Toimintaperiaatteiden selvittämisen jälkeen algoritmien käyttäytymistä verrataan käytännön testeissä. Niissä testataan, kuinka kauan eri algoritmit lajittelevat samaa syötettä ja kuinka paljon ne käyttävät tilaa tietokoneen keskusmuistista. Testaamisen toisena ideana on selvittää, miten syötteiden koon kasvaminen vaikuttaa suoritusaikoihin eri algoritmeilla. Testaaminen tehdään useilla selvästi eri suuruusluokkiin kuuluvilla satunnaisilla kokonaislukuryhmillä. Tässä tutkielmassa esiteltävien menetelmien lisäksi vertailuun otetaan mukaan myös Javan oma sisäänrakennettu lajittelumetodi, jotta nähdään, miten nämä vanhemmat algoritmit pärjäävät osittain useaa ydintä käyttävään uudempaan menetelmään verrattuna.

Lajittelumenetelmille on tehty melko niukalti aikaisempia tutkimuksia, joissa olisi kattavasti vertailtu useaa algoritmia samoilla syötteillä. Joissain tutkimuksissa on keskitytty tietyn algoritmin toimintaan ja testattu sitä pienillä muutoksilla, kun taas toisissa käytetään mittareina tehtyjen vertailujen tai alkioiden siirtojen määrää. Tässä tutkielmassa erilaiset lajittelumenetelmät suorittavat samat testit samalla koneella ja samoilla satunnaissyötteillä, jolloin suoritusaikojen tulokset ovat vertailukelpoisia keskenään. Samalla testataan, miten paria jo valmiiksi nopeaa menetelmää voi vielä hiukan nopeuttaa hyvin pienillä muutoksilla. Suoritusajat riippuvat käytettävän tietokoneen tai muun laitteen komponenteista ja ominaisuuksista, mutta lajittelunopeuden järjestys ja suoritusaikojen suuruusluokat ovat samat hieman hitaammilla tai nopeammillakin koneilla.

2. Lajittelumenetelmät

Lajittelumenetelmien tarkoitus on lajitella annetut lukujonot. Menetelmiä on olemassa useita erilaisia, joista jokainen toimii omalla tavallaan. Ne sopivat eri tavoin erilaisiin tilanteisiin. Tietynlaiset syötteet sopivat joillekin menetelmistä todella hyvin; jotkut menetelmät taas ovat yhtä nopeita kaikenlaisilla syötteillä.

Algoritmit voidaan jakaa kahteen ryhmään kahden eri piirteen mukaan. Ensimmäinen näistä piirteistä on lajittelualgoritmin *vakaus*. Vakaa algoritmi säilyttää keskenään samojen alkioiden keskenäisen järjestyksen koko ajan. Jos lajiteltava ominaisuus (arvo) on yhtä suuri molemmissa alkioissa, ne ovat samassa järjestyksessä lajittelun jälkeen kuin ennen sitä. Epävakaassa keskinäinen järjestys saattaa muuttua algoritmin suorittamisen aikana. Vakaus on tärkeä ominaisuus, jos alkioita tai olioita lajitellaan monen eri ominaisuuden suhteen peräkkäin. Esimerkiksi tietokannassa on henkilöiden pituudet ja painot omissa kentissään. Jos halutaan tietokannasta tiedot ensisijaisesti pituuden ja toissijaisesti painon mukaan järjestykseen, voidaan ensin lajitella painon mukaan ja tämän jälkeen vakaalla algoritmilla pituuden mukaan. Tällöin henkilöiden pituuksien ollessa samat heidän tietonsa esiintyvät painojärjestyksessä.

Toinen ominaisuus, jonka perusteella algoritmit voidaan jakaa kahtia, on *minimitilassa toimiminen*. Jos menetelmä toimii minimitilassa, silloin se tarvitsee tietyn kiinteän määrän muistia lisää lajiteltavien alkioiden lukumäärästä riippumatta. Tällöin algoritmi toimii paikallaan, eli sen lisätilavaatimus on luokkaa O(1). Tämä on tärkeä ominaisuus, jos muistia on rajoitetusti, kuten esimerkiksi mobiililaitteissa ja vanhoissa, pieniä yksinkertaisia järjestelmiä ylläpitävissä koneissa saattaa olla. Jos menetelmä ei toimi minimitilassa, sen muistin tarve kasvaa alkioiden lukumäärän kasvaessa, minkä lisäksi se tarvitsee isommilla syötteillä enemmän lisämuistia lajittelun läpikäymiseen.

Suurien tietokantojen lajittelussa on tärkeää optimoida muistinkäyttöä ja jakaa lajittelusta aiheutuvaa kuormaa usealle suorittimelle. Tällaiset lajittelumenetelmät ovat yleensä enemmän tai vähemmän tapauskohtaisesti räätälöityjä ja huomattavasti monimutkaisempia kuin tässä tutkielmassa esitettävät menetelmät. Eri menetelmien toimintatapoja saatetaan yhdistellä, jotta saavutettaisiin sekä hyvä suoritusnopeus että kohtuullinen muistintarve. Luvun 3.3. suoritusaikatesteissä testataan kahta algoritmia hiukan muutettuina.

Seuraavissa alaluvuissa esitetään lajittelun kulkua eri algoritmeilla ja tutustutaan tarkemmin niiden toimintaperiaatteisiin. Menetelmät ovat lisäyslajittelu, valintalajittelu, pikalajittelusta rekursiivinen ja iteratiivinen versio, lomituslajittelu sekä Shellin lajittelu.

2.1. Lisäyslajittelu

Lisäyslajittelu on eräs yksinkertaisimmista lajittelualgoritmeista. Se on vakaa ja toimii minimitilassa, joten se sopii moniin erilaisiin käyttötarkoituksiin. Kohtuullisen pienillä syötemäärillä nopeusero muihin lajittelumenetelmiin on käytännössä huomaamattoman pieni, joten se on ominaisuuksiensa puolesta hyvä valinta moneen lajittelutehtävään.

Lisäyslajittelu kuuluu pienennä ja hallitse (engl. decrease-and-conquer) -tekniikoihin. Tekniikassa käytetään hyväksi pienemmän osaongelman ratkaisua isomman ongelman ratkaisemisessa. N-alkioisen taulukon ratkaisemiseen käytetään N – 1 ensimmäisen

alkion valmiiksi lajiteltua ratkaisua. N – 1 alkion ongelman ratkaisemiseksi käytetään ratkaisua N – 2 ensimmäisen alkion valmiista lajittelusta jne. Eli ensimmäiseksi ratkaistavaksi jää yhden alkion lajittelu. Sen jälkeen toinen alkio lisätään lajiteltujen joukkoon oikealle paikalleen ensimmäisen alkion edelle tai jälkeen. Tämän jälkeen alkioita lisätään lajiteltujen joukkoon yksi kerrallaan oikeille paikoilleen. Lisäyksessä verrataan lisättävää aina edelliseen jo lajiteltuun, kunnes löytyy yhtä suuri tai pienempi alkio. Silloin kaikkien taulukossa edeltävien alkioiden tiedetään olevan korkeintaan yhtä suuria kuin lisättävä. Jos saavutetaan taulukon alku, eli ei ole enää vertailtavia alkioita, lisättävästä tulee ensimmäinen eli pienin alkio. Alkioita lisätään samalla tavalla yksitellen vertailemalla, kunnes kaikki alkiot ovat järjestyksessä. Vaikka algoritmin idea on rekursiivisen oloinen, algoritmi toteutetaan iteratiivisena, joka on helpompi, tehokkaampi ja vähemmän muistia vaativa tapa.

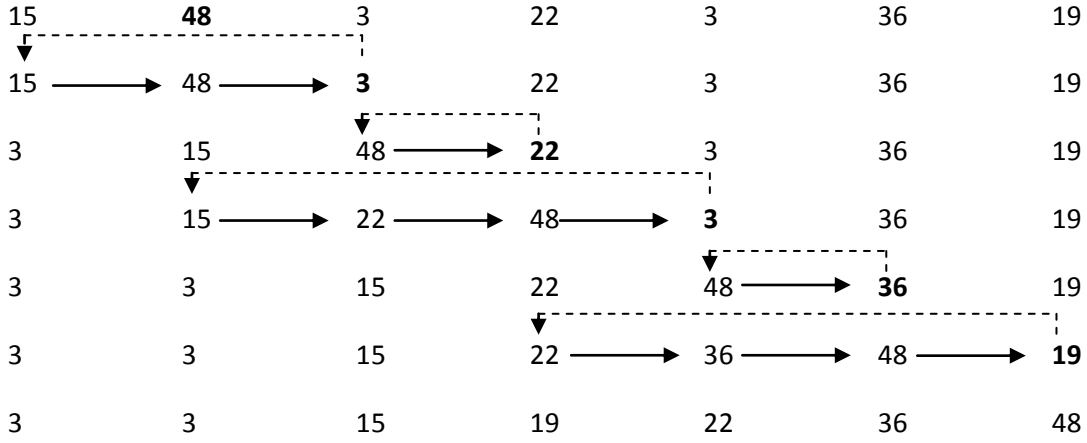
Lisäyslajittelun ideaa voi havainnollistaa pelikorttiesimerkillä. Sekaisin olevista korteista tehdään pakka, josta otettaessa kortti asetetaan käteen numerojärjestyksessä oikeaan paikkaan. Lopulta kun pakka on loppunut, kortit ovat kädessä numerojärjestyksessä.

Lisäyslajittelun pseudokoodi: (Levitin 2007)

```
LisäysLajittele(A[0..N – 1])
    // Lajittelee annetun syötteen paikallaan lisäyslajittelulla.
    // Syöte: Taulukko A[0..N – 1], jossa N kpl järjestettäviä alkioita.
    // Tulos: Taulukko A[0..N – 1] lajiteltuna pienimmästä suurimpaan.
    for i ← 1 to N – 1 do
        v ← A[i]
        j ← i – 1
        while j ≥ 0 ja A[j] > v do
            A[j + 1] ← A[j]
            j ← j – 1
        A[j + 1] ← v
```

Kuvassa 1 on esitetty esimerkki lisäyslajittelun etenemisestä positiivisilla kokonaisluvuilla. Ensimmäistä lukua ei tarvitse verrata mihinkään, sillä edellisiä alkioita ei ole. Ensimmäisenä operaationa lisätään toinen alkio 48 vertaamalla sitä edelliseen valmiiksi lajiteltuun joukkoon, jossa on vain yksi alkio, jonka arvo on 15. Koska lisättävä on näistä suurempi, se lisätään lajitellun joukon viimeiseksi, eli käytännössä ei tehdä mitään. Seuraavaksi lisätään alkio 3, joka on pienempi verrattaessa edellisiin, joiden arvot ovat 48 ja 15 tässä vertailujärjestyksessä. Alkio 3 on tämän jälkeen joukon ensimmäisenä. Seuraavaksi alkiota 22 vertaillaan edellä oleviin, kunnes löytyy pienempi tai yhtä suuri. Ainoastaan 48 on isompi, joten se vaihtaa paikkaa alkion 22 kanssa. Seuraava alkio, arvoltaan 3, on pienempi kuin 48, 22 ja 15. Näiden kolmen jälkeen sitä verrataan joukon ensimmäiseen alkioon, joka on myös arvoltaan 3. Koska lisättävä ei ole pienempi, alkiot eivät vaihda paikkaa. Näin siis alkuperäisessä syötteessä edempänä ollut arvoltaan yhtä suuri alkio on toisen edellä myös valmiiksi lajitellussa

joukossa. Samalla tavalla lisätään vielä alkiot 36 ja 19, minkä jälkeen alimmalla rivillä on alkuperäinen syöte lajiteltuna pienimmästä isoimpaan.



Kuva 1. Lisäyslajittelun kulku. Lisättävä alkio on tummennettu, vertailtavien alkioiden siirrot on merkitty mustilla nuolilla ja lisättävän alkion siirto omalle paikalleen mustalla katkoviivoitetulla nuolella.

Lisäyslajittelussa paras tapaus on tilanne, jossa kaikki alkiot ovat jo valmiiksi järjestyksessä. Toisesta alkaen viimeiseen alkioon asti kaikkia vertaillaan vain yhden kerran ja mitään ei siirretä minnekään. Tällöin tarvitaan siis N – 1 operaatiota, eli tapauksen aikavaativuus on luokkaa Θ(N). Tämä tarkoittaa, että aikavaativuus on suoraan verrannollinen järjestettävien alkioiden lukumäärään. Tapauksen kompleksisuus on $C_{paras}(N) = \sum_{i=1}^{N-1} 1 = N - 1 \in \Theta(N)$.

Pahimmassa tapauksessa suoritusaika on merkittävästi huonompi. Huonoin tapaus on päinvastaisessa järjestyksessä oleva syöte, jossa ei ole samanarvoisia alkioita. Järjestettäessä jokaista lisättävää alkiota täytyy verrata aina kaikkiin edelläoleviin alkioihin. Tämän tapauksen kompleksisuus on

$$C_{pahin}(N) = \sum_{i=1}^{N-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} \in \Theta(N^2).$$

Pahimman tapauksen aikavaativuus on *neliöllinen*.

Keskimääräisen tapauksen aikavaativuus voidaan arvioida olettamalla, että jokaisella kierroksella lisättävä alkio osuisi puoliväliin lajiteltua syötettä. Tällöin lisättävä olisi suurempi kuin lajitellun osan alkupään alkiot ja pienempi kuin loppupään alkiot. Vertailuja tulisi tällöin $k_i \approx i / 2$ kappaletta joka kierroksella. Tällöin kompleksisuudeksi saadaan

$$C_{keskim.}(N) = \sum_{i=1}^{N-1} k_i \approx \sum_{i=0}^{N-1} i / 2 = \frac{1}{2} \sum_{i=1}^{N-1} i = \frac{(N-1)N}{4} \in \Theta(N^2).$$

Keskimääräisen tapauksen aikavaativuus on siis neliöllinen, mikä on samaa suuruusluokkaa kuin pahimmalla tapauksella.

2.2. Valintalajittelu

Valintalajittelu on yksi helpoimmin ymmärrettävistä lajittelumenetelmistä. Se on brute force -tekniikka, joka vaatii suomennetun nimensä mukaisesti raakaa voimaa, eli paljon laskentatehoa. Valintalajittelu toimii minimitilassa, joten pienillä syötemäärillä ja pienellä muistimäärällä se on vielä käyttökelpoinen vaihtoehto.

Valintalajittelussa etsitään aina jäljelläolevasta alkiojoukosta pienintä alkiota, joka lisätään lajiteltujen joukkoon viimeiseksi vaihtamalla ensimmäisen lajittelemattoman ja pienimmän paikkaa keskenään. Ensimmäisessä vaiheessa käydään kaikki N alkiota kertaalleen läpi, jotta saadaan selville, mikä niistä on pienin. Tämän jälkeen joudutaan käymään läpi koko N – 1 alkion joukko, josta etsitään pienin. Toisena etsittävä on kuitenkin välttämättä vähintään yhtä suuri kuin ensimmäisellä kierroksella löydetty pienin alkio, joten nyt löytyneestä pienimmästä alkiosta tulee lajitellun joukon toinen alkio. Tällaisia kierroksia tehdään N – 1 kappaletta, minkä jälkeen viimeinen jäljelle jäänyt on joukon isoin alkio, jonka annetaan olla omalla paikallaan joukon viimeisenä.

Valintalajittelun pseudokoodi: (Levitin 2007)

```
ValintaLajittele(A[0..N – 1])
    // Lajittelee annetun syötteen paikallaan valintalajittelulla.
    // Syöte: Taulukko A[0..N – 1], jossa N kpl järjestettäviä alkioita.
    // Tulos: Taulukko A[0..N – 1] lajiteltuna pienimmästä suurimpaan.
    for i ← 0 to N – 2 do
        min ← i
        for j ← i + 1 to N – 1 do
            if A[j] < A[min] then min ← j
        vaihda A[i] ja A[min]
```

15	48	**3**	22	3	36	19
3 |	48	15	22	**3**	36	19
3	3 |	**15**	22	48	36	19
3	3	15 |	22	48	36	**19**
3	3	15	19 |	48	36	**22**
3	3	15	19	22 |	**36**	48
3	3	15	19	22	36 |	48

Kuva 2. Valintalajittelun kulku. Pystyviivan vasemmalla puolella lajiteltu syöte, oikealla puolella lajittelematon. Lajittelemattomista pienin alkio on tummennettu.

Valintalajittelun kulku on esitelty kuvassa 2. Yksi rivi vastaa yhtä valintakierrosta, jossa etsitään pienin lajittelematon alkio. Ensimmäisellä kierroksella käydään kaikki alkiot läpi ja etsitään niistä pienin. Syötteessä on kaksi yhtä suurta pienintä alkiota, joten niistä algoritmin perusteella ensimmäisenä huomattu valitaan pienimmäksi. Tämän jälkeen pienimmän ja ensimmäisen alkion paikkoja vaihdetaan keskenään. Seuraavalla kierroksella käydään läpi kaikki jäljellä olevat lajittelemattomat alkiot, joista etsitään taas pienin. Tälläkin kierroksella pienimmän alkion arvo on 3, jonka paikka vaihdetaan alkion 48 kanssa.

Seuraavaksi käydään läpi alkiot 15, 22, 48, 36 ja 19, joista ensimmäinen on pienin. Sen paikka vaihdetaan itsensä kanssa – toisin sanoen se pysyy paikallaan. Paikan vaihtaminen itsensä kanssa vaatii keskimääräisellä syötteellä vähemmän operaatioita kuin mitä yhden vertailun lisääminen jokaiseen paikan vaihtamiseen vaatisi. Yhden oikealla paikalla olevan kohdalla säästettäisiin kaksi operaatiota (kolme kopioimista paikanvaihdoista – yksi vertailu), mutta tästä olisi hyötyä vain yli kolmasosan ollessa oikealla paikallaan. Normaalissa satunnaisessa syötteessä tämä ei yleensä toteudu. Tämän lisäksi saatava hyöty olisi häviävän pieni verrattuna kokonaissuoritusaikaan. Kuuden pienimmän luvun etsimisen jälkeen on jäljellä enää yksi alkio. Koska se on jäänyt viimeiseksi, se on suurin ja se on valmiiksi oikealla paikallaan viimeisenä.

Valintalajittelun aikavaativuus on sama syötteen luonteesta riippumatta, sillä aina etsitään pienintä arvoa kaikista jäljellä olevista. Valmiiksi järjestetyn läpikäyminen vaatii saman työn kuin päinvastaisessa tai satunnaisessa järjestyksessä olevan järjestäminen. Vain syötteen koko vaikuttaa suoritusaikaan. Näin ollen paras ja keskimääräinen tapaus ovat yhtä vaativia kuin pahin tapaus.

$$C(N) = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} 1 = \sum_{i=0}^{N-2} (N - 1 - i) = \frac{(N-1)N}{2} \in \Theta(N^2).$$

Valintalajittelusta saisi hyvin helposti vakaan algoritmin poistamalla pienimmän alkion ja lisäämällä sen alkion A[i] kohdalle ja siirtämällä loppuja alkuperäiseen paikkaan asti yhden paikan eteenpäin. Javan ArrayList-luokassa, jonka avulla tämän tutkielman algoritmit on koodattu, on olemassa tähän tarkoitukseen valmis metodi add(int index, E element) (Oracle 2012).

2.3. Pikalajittelu

Pikalajittelun kehitti britti Sir Charles Antony Richard Hoare, joka tunnetaan myös nimillä Tony Hoare tai C. A. R. Hoare. Hän kehitti algoritmin vuonna 1961 (Hoare 1962). Pikalajittelu kuuluu divide-and-conquer eli hajota-ja-hallitse -menetelmiin, joissa ongelma jaetaan pienemmiksi aliongelmiksi. Alkuperäinen versio on toteutettu rekursiivisesti, mutta pikalajittelusta on myös iteratiivinen versio (Ďurian 1986).

Pikalajittelu on epävakaa, mutta nopeutensa ansiosta se on hyvä vaihtoehto isojen tietokantojen järjestämisessä. Seuraavissa aliluvuissa esitellään rekursiivinen versio (luku 2.3.1.) ja iteratiivinen versio (luku 2.3.2.) pikalajittelusta.

2.3.1. Rekursiivinen pikalajittelu

Pikalajittelun rekursiivisessa versiossa joukko jaetaan kahteen alijoukkoon, joista toinen jää odottamaan ensimmäisen lajittelua. Ensimmäisen alijoukon lajittelemisessa kutsutaan uudestaan pikalajittelua, joka jakaa tämän alijoukon taas kahteen alijoukkoon, joista jälkimmäinen jää odottamaan. Lopulta lajitellaan yhden alkion kokoista tai tyhjää joukkoa, kun loput alkiot odottavat rekursiopinoissaan edellisten lajitteluiden valmistumista. Tämän vuoksi rekursiivinen pikalajittelu vaatii lisämuistia algoritmin suorittamiseen, joten se ei toimi minimitilassa.

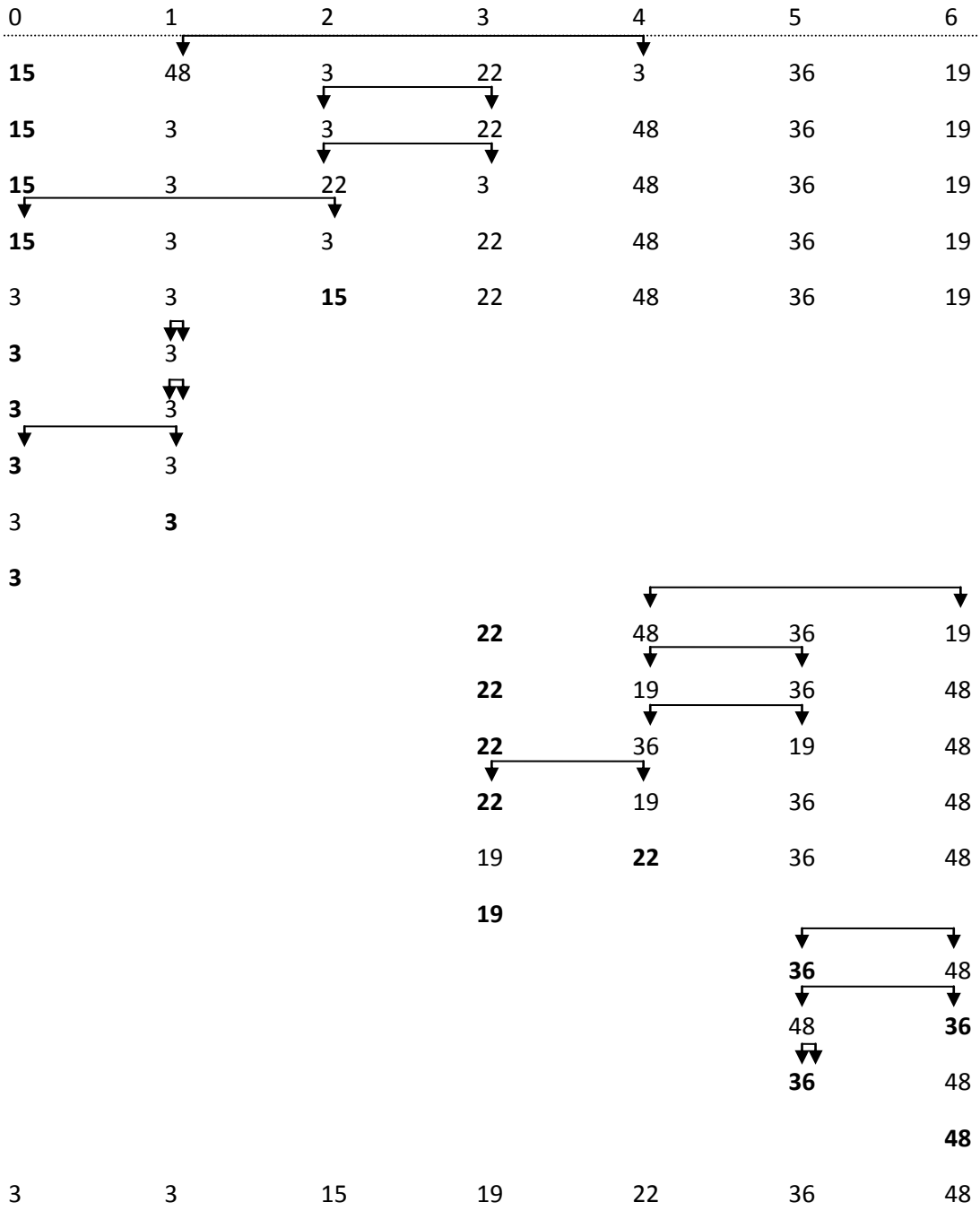
Rekursiivisen lajitteun pseudokoodi: (Levitin 2007)

```
RekPikaLajittele(A[v...o])
    // Lajittelee annetun syötteen rekursiivisella pikalajittelulla.
    // Syöte: Taulukko A[v...o], joka on taulukon A[0..N – 1] alitaulukko, jossa
alitaulukon reunat ovat: v = vasen reuna ja o = oikea reuna.
    // Tulos: Taulukko A[0..N – 1] lajiteltuna pienimmästä suurimpaan.
    if v < o then
        s ← Partitioi(A[v…o])  // s = sarana-alkion paikka lajitellussa syötteessä
        RekPikaLajittele(A[v...s – 1])
        RekPikaLajittele(A[s + 1...o])
```

Pikalajittelu kutsuu partitiointia, joka valitsee *pivot*- eli *sarana*-alkion, jonka avulla lajiteltava joukko jaetaan kahdeksi alijoukoksi. Toisessa ryhmässä kaikki alkiot ovat pienempiä tai yhtä suuria ja toisessa suurempia tai yhtä suuria kuin sarana-alkio. Alijoukko voi olla myös tyhjä, jos sarana-alkio on pienin tai suurin alkio.

Partitioinnin pseudokoodi: (Levitin 2007)

```
Partitioi(A[v...o])
    // Jakaa annetun syötteen kahteen alijoukkoon ja palauttaa sarana-alkion paikan.
    // Syöte: Taulukko A[v...o], joka on taulukon A[0..N – 1] alitaulukko, jossa
        alitaulukon reunat ovat: v = vasen reuna ja o = oikea reuna, v < o.
    // Tulos: Taulukosta A[0..N – 1] etsitty paikka sarana-alkiolle, sitä pienemmät
        jäävät sen vasemmalle puolelle ja suuremmat oikealle puolelle.
    s ← A[v]
    i ← v;  j ← o + 1
    repeat
        repeat i ← i + 1 until A[i] ≥ s or i = o
        repeat j ← j – 1 until A[j] ≤ s
        vaihda A[i] ja A[j]
    until i ≥ j
    vaihda A[i] ja A[j]   // kumotaan viimeinen vaihto ( i ≥ j )
    vaihda A[v] ja A[j]
    return j
```



Kuva 3. Rekursiivisen pikalajittelun kulku. Sarana-alkiot on merkitty tummentamalla ja vaihdot nuolilla. Ylimmällä rivillä on alkioiden paikkojen numerointi.

Pikalajittelun eteneminen esitetään kuvassa 3, jossa ensimmäisellä rivillä on lähtötilanne ja alimmalla rivillä lopputulos. Ensimmäiseksi sarana-alkioksi valitaan partitiointialgoritmin mukaan koko alkiojoukosta ensimmäinen alkio, jonka arvo on 15. Muuttujaa i kasvatetaan, kunnes löytyy suurempi tai yhtä suuri alkio ja muuttujaa j pienennetään viimeisestä lähtien, kunnes löytyy pienempi tai yhtä suuri kuin sarana-alkio. Jos i saavuttaa lajiteltavan joukon oikean reunan, sitä ei enää kasvateta. Tämän jälkeen alkiot paikoilla i ja j vaihdetaan keskenään, eli alkiot 3 ja 48 vaihtavat paikkoja keskenään. Jatketaan i:n kasvattamista ja j:n pienentämistä samalla tavalla kuin aikaisemmin, jolloin i:n arvoksi tulee 3 ja j:n arvoksi tulee 2, joten vaihdetaan näillä

paikoilla olevat alkiot keskenään. Koska i on j:tä suurempi, suoritus poistuu silmukasta. Seuraavaksi kumotaan viimeinen vaihto, jossa alkiot menivät väärään järjestykseen. Yksi kumoava operaatio vie huomattavasti vähemmän aikaa kuin jokaisessa kohdassa i:n ja j:n vertaileminen keskenään. Nyt alkiot on jaettu kahteen alijoukkoon, joten ensimmäisenä oleva sarana-alkio vaihdetaan oikealle paikalleen ja tällä paikalla oleva alkio sarana-alkion paikalle ensimmäiseksi. Partitiointi palauttaa arvon j, jonka avulla määritetään seuraavat alijoukkojen rekursiiviset pikalajittelukutsut.

Ensimmäisen partitioinnin päätyttyä kutsutaan uudelleen pikalajittelukutsua, joka saa argumenttina sarana-alkiota edeltävän alijoukon {3, 3}. Seuraavaksi kutsutaan partitiointialgoritmia, joka valitsee ensimmäisen alkion sarana-alkioksi. Muuttujia i ja j muutetaan taas, kunnes löydetään ehdot täyttävät alkiot. Tässä tapauksessa molempien arvoksi tulee yksi. Paikassa yksi olevan alkion paikkaa vaihdetaan itsensä kanssa, jonka jälkeen poistutaan silmukasta ja kumotaan tämä vaihto. Edellä tuli kaksi alkion paikan vaihtoa itsensä kanssa, joka tarkoittaa kuutta ylimääräistä operaatiota. Näiden vaatima suoritusaika on suuressa mittakaavassa kuitenkin vain hyvin pieni osa kokonaissuoritusajasta. Lopuksi j:n kohdalla oleva alkio vaihdetaan sarana-alkion kanssa.

Sarana-alkion vasemmalle puolelle jää yhden alkion alijoukko ja oikealle puolelle ei jää mitään. Tämän jälkeen pikalajittelua kutsutaan edellämainituille alijoukoille, mutta tämän mittaiset joukot eivät täytä ehtoa vasen reuna < oikea reuna, joten lajitteluista poistutaan tekemättä mitään.

Alkuperäisen pikalajittelun ensimmäinen rekursiivinen kutsu on suoritettu loppuun, joten seuraavana kutsutaan rekursiivista pikalajittelua alkioille paikoilla 3 – 6. Lajitteleminen aloitetaan partitioinnilla, jossa sarana-alkioksi tulee alkio 22. Ensimmäinen suurempi tai yhtä suuri on alkio 48 paikassa 4 ja loppupäästä ensimmäinen pienempi tai yhtä suuri on alkio 19 paikassa 6. Näiden kahden paikat vaihdetaan keskenään ja jatketaan i:n kasvattamista ja j:n pienentämistä. Seuraavaksi i:n arvoksi tulee viisi ja j:n arvoksi neljä. Näiden paikkojen alkiot 19 ja 36 vaihdetaan keskenään, jolloin ne menevät väärään järjestykseen. Tämän jälkeen poistutaan silmukasta ja tehdään viimeisen kumoava operaatio, eli alkiot 36 ja 19 vaihdetaan takaisin oikeaan järjestykseen. Tämän jälkeen sarana-alkion paikkaa vaihdetaan j:n kohdalla olevan kanssa ja palautetaan sarana-alkion uusi paikka pikalajittelualgoritmille.

Pikalajittelua kutsutaan seuraavaksi yhden kokoiselle joukolle, joten sieltä palataan heti tekemättä mitään. Tämän jälkeen lajitellaan viimeinen alijoukko rekursiivisella pikalajittelulla. Partitioinnissa sarana-alkioksi valitaan taas joukon ensimmäinen, alkio 36. Etsitään muuttujille i ja j oikeat arvot, jonka jälkeen vaihdetaan kyseisten paikkojen alkiot keskenään. Tällä kerralla i ja j ohittavat toisensa, joten alkiot 36 ja 48 menevät väärään järjestykseen. Seuraavaksi poistutaan taas silmukasta ja kumotaan viimeinen vaihto. Tämän jälkeen vaihdetaan vielä sarana-alkion ja j:n paikat. Tässä tapauksessa ne ovat samat, joten alkio 36 vaihdetaan itsensä kanssa. Tämä tarkoittaa, että tehdään taas kolme turhaa asettamisoperaatiota ennen arvon palauttamista. Viimeisenä on vuorossa

tyhjän ja yksialkioisen alijoukon pikalajittelut, joista palataan tekemättä mitään. Tämän jälkeen rekursiivinen pikalajittelu on valmis.

Partitioinnissa tehdään N – 1 vertailua, joten sen aikavaatimus on $\Theta(N)$. Tämän lisäksi pikalajittelua kutsutaan kahteen kertaan rekursiivisesti. Parhaassa tapauksessa pikalajittelun partitiointi jakaa lajiteltavan joka kerralla kahteen yhtä suureen osaan. Rekursiokutsussa lajiteltava voidaan arvioida ylöspäin arvoon N / 2.

$$C_{paras}(N) = 2\,C_{paras}(N/2) + N, \text{ kun } N > 1,\ C_{paras}(1) = 0.$$

Kun $N = 2^k$, eli N on kahden potenssi, voidaan ylläolevasta kaavasta saada rekursioyhtälö ratkaisemalla tulos: $C_{paras}(N) \in \Theta(N \log_2 N)$.

Pikalajittelun huonoimmassa tapauksessa partitioinnin jälkeen toinen alijoukoista on tyhjä. Esimerkiksi valmiiksi lajitellut tai pelkästään samoja alkioita sisältävät joukot ovat tällaisia. Rekursiivinen pikalajittelu lajittelee ja partitioi näissä tapauksissa aina vain yhtä pienempää joukkoa, kunnes syöte on kokonaan lajiteltu. Rekursioyhtälöstä $C_{pahin}(N) = C_{pahin}(N - 1) + N$ saadaan summakaava vertailujen määrälle:

$$C_{pahin}(N) = \sum_{i=1}^{N} i = \frac{(N+1)N}{2} \in \Theta(N^2).$$

Keskimääräisessä tapauksessa sarana-alkion toiselle puolelle tulee k alkiota ja toiselle N – 1 – k alkiota. Jos kaikki vaihtoehdot ovat yhtä todennäköisiä, niiden todennäköisyys on 1 / N.

$$C_{keskim.}(N) = \frac{1}{N} \sum_{k=1}^{N} \Big((N - 1) + C_{keskim.}(k) + C_{keskim.}(N - 1 - k) \Big), \text{ kun } N > 1$$

$$C_{keskim.}(0) = C_{keskim.}(1) = 0$$

Keskimääräisen tapauksen kaavasta voidaan ratkaista aikavaativuus, jonka huomataan olevan samaa aikavaativuusluokkaa kuin paras tapaus.

$$C_{keskim.}(N) = \Theta(N \ln N) \in \Theta(N \log_2 N).$$

Partitioimiseen on myös useita muita tapoja, mutta niitä ei käsitellä tässä tutkielmassa. Rekursiivinen pikalajittelu valitsee sarana-alkioksi aina lajiteltavan joukon ensimmäisen alkion. Tämä on hyvin harvoin paras vaihtoehto, minkä takia on kehitetty erilaisia versioita sarana-alkion valintaan. Sarana-alkion paikka voidaan valita esimerkiksi satunnaisesti lajiteltavasta joukosta, kuten satunnaistetussa pikalajittelussa tehdään. Siinä valitaan satunnainen indeksi, jonka kohdalla oleva alkio vaihdetaan ensimmäiseksi, ja tämän jälkeen jatketaan normaalisti samalla tavalla. Satunnaistettu versio voi tehdä jopa pahimmasta tapauksesta keskimääräisen tapauksen. Toinen hyvä tapa on ottaa kolme alkiota ja valita niistä keskimmäinen sarana-alkioksi. Kolme alkiota voivat olla esimerkiksi kolme ensimmäistä tai vaikka ensimmäinen, keskimmäinen ja viimeinen. Keskimmäinen arvo vaihdetaan ensimmäiseksi ja lajittelua jatketaan normaalisti. Hyöty satunnaisten tai keskimmäisten alkioiden vaihtamisesta on paljon suurempi kuin niiden vaihtamiseen kuluva aika, jolloin pikalajittelusta tulee entistä tehokkaampi. Ainoastaan samaa alkiota sisältävän joukon lajittelu hidastuu, mutta hidastusvaikutus on häviävän pieni.

2.3.2. Iteratiivinen pikalajittelu

Rekursiivinen pikalajittelu on nopea lajittelualgoritmi, mutta lisämuistin tarve kasvaa lajiteltavan syötteen kasvaessa. Tämän vuoksi on kehitetty minimitilassa toimiva iteratiivinen versio pikalajittelusta. Se toimii samalla periaatteella kuin rekursiivinen versio, mutta siinä rekursiiviset kutsut on korvattu iteratiivisilla silmukoilla. Alkiot lajitellaan sarana-alkion avulla kahteen alijoukkoon, joista toisessa kaikki alkiot ovat korkeintaan yhtä suuria ja toisessa vähintään yhtä suuria kuin sarana-alkio. Ilman rekursiopinoja sarana- ja apualkioita joudutaan siirtämään edestakaisin merkeiksi, jotta tiedetään, missä vaiheessa suoritus on menossa. Tämän vuoksi iteratiivinen versio ei ole aivan yhtä nopea kuin rekursiivinen versio. Koska algoritmin toimintaperiaate on sama kuin rekursiivisessakin versiossa, iteratiivinenkin versio on epävakaa.

Iteratiivisena versiona käytetään muokattua versiota Branislav Ďurianin vuonna 1986 kehittämästä menetelmästä QuickSort without a stack (Ďurian 1986). Alkuperäisessä versiossa muuttuja m alustetaan alussa arvolla yhdeksän, joten lajittelua ei jatketa alijoukkojen ollessa tasan tai alle kymmenen alkion kokoisia. Alkuperäisen version lopussa algoritmissa kutsutaan lisäyslajittelua, joka lajittelee tässä tapauksessa lähes valmiiksi lajitellun joukon todella nopeasti valmiiksi. Kappaleessa 3 tutkitaan eri lajittelumenetelmien eroja, joten tässä muuttuja m alustetaan arvolla nolla, jolloin algoritmi lajittelee koko joukon loppuun asti iteratiivisella pikalajitelulla.

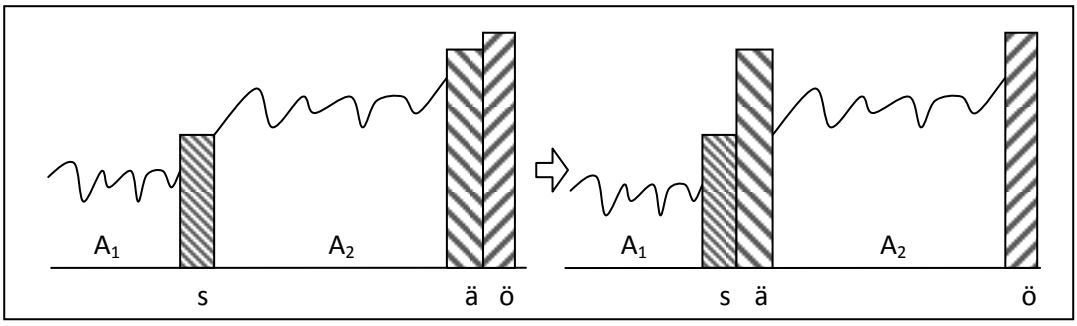
Iteratiivisen pikalajittelun pseudokoodi: (Ďurian 1986)

```
IterPikaLajittele(A[0...N – 1])
    // Lajittelee annetun syötteen iteratiivisella pikalajittelulla.
    // Syöte: Taulukko A[0...N – 1], jossa N kappaletta alkioita.
    // Tulos: Taulukko A[0..N – 1], alkuperäinen taulukko lajiteltuna.
    // Lisätään kaksi äärettömän suurta arvoa taulukon viimeisiksi alkioiksi
    A[N] ← ∞ - 1; A[N + 1] ← ∞; k ← 0; r ← N; m ← 0
    repeat
        while r – k > m  do
            i ← k; j ← r
            p ← A[k]
            repeat
                repeat i ← i + 1 until A[i] ≥ p
                repeat j ← j – 1 until A[j] ≤ p
                if i < j then vaihda A[i] ja A[j]
            until i ≥ j
            A[k] ← A[j]; A[j] ← p
            vaihda A[i] ja A[r]
            r ← j
        k ← r
        repeat k ← k + 1 until A[k] > p
        if k ≤ N then
```

```
            p ← A[k];  r ← k
            repeat r ← r + 1 until A[r] > p
            r ← r – 1; A[k] ← A[r]; A[r] ← p
    until k > N
    poista A[N + 1] ja A[N]
```

Iteratiivisen pikalajittelun esittäminen pienen esimerkin avulla ei ole mielekästä, sillä sen seuraaminen olisi todella hankalaa merkkialkioiden jatkuvan edestakaisin siirtämisten vuoksi. Siksi tässä esitetään seuraavaksi loppuun asti lajiteltavan esimerkin sijaan algoritmin toimintaperiaatteen idea. Iteratiivisen pikalajittelussa joukon perään lisätään alussa kaksi äärettömän suurta arvoa. Joukon toiseksi viimeiseksi tuleva on pienempi kuin viimeiseksi tuleva. Näitä on merkitty kuvassa 4 kirjaimilla ä ja ö. Niitä käytetään apuna lajittelun edetessä, jotta algoritmi osaa tehdä oikeat asiat oikeassa järjestyksessä.

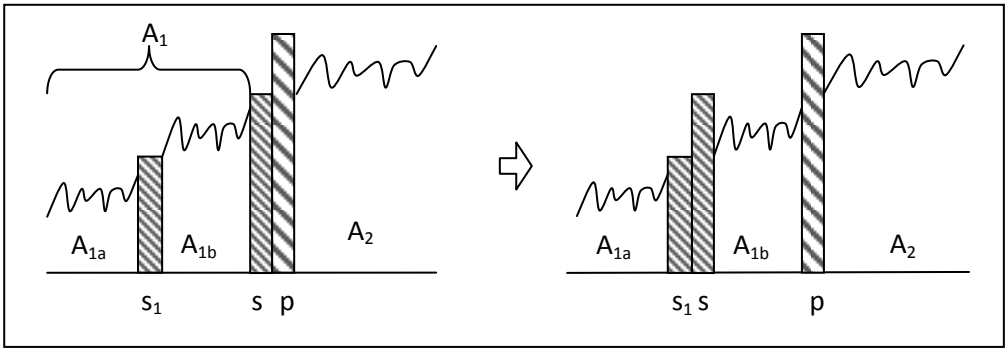
Ensimmäisenä valitaan vasemmanpuoleisin alkio sarana-alkioksi ja alustetaan apumuuttujat i ja j. Muuttujaa i kasvatetaan vasemmalta, kunnes löytyy suurempi tai yhtä suuri arvo kuin sarana-alkiolla. Muuttujaa j pienennetään oikealta lähtien, kunnes löytyy pienempi tai yhtä suuri arvo kuin sarana-alkiolla. Jos i < j, näillä paikoilla olevat alkiot vaihdetaan keskenään. Tämän jälkeen i:n kasvattamista, j:n pienentämistä ja näillä paikoilla olevien vaihteluja jatketaan, kunnes $i \geq j$. Tämä partitiointi tehdään siis samalla tavalla kuin rekursiivisessakin pikalajittelussa. Sarana-alkio vaihdetaan partitioinnin lopuksi j:n kohdalla olevan kanssa, jolloin se on oikealla paikalla. Tämän jälkeen vaihdetaan pienempi äärettömän isoista merkkialkioista sarana-alkion oikealla puolella olevan alkion kanssa. Tämän vaihdon jälkeen sarana-alkio s ja apualkio ä jakavat muun syötteen kahteen osaan kuvan 4 mukaisesti. Muuttuja r muutetaan samaksi kuin j, jonka jälkeen siirrytään partitioimaan seuraavaa aluetta A_1.



Kuva 4. Iteratiivisen pikalajittelun tilanne ensimmäisen partitioinnin jälkeen.

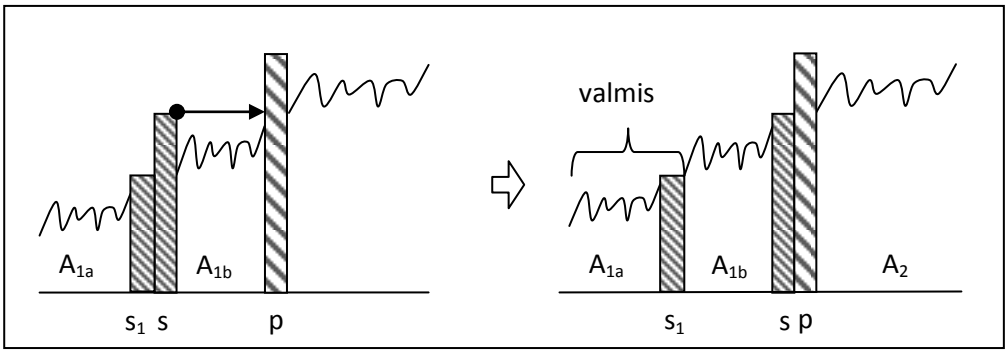
Partitiointia jatketaan alijoukolla A_1, jonka alkiot ovat edellisen partitioinnin jäljiltä pienempiä tai yhtä suuria kuin sarana-alkio s. Valitaan A_1:n ensimmäinen alkio sarana-alkioksi s_1, jonka avulla suoritetaan partitiointi samalla tavalla kuin edellä. Kun partitiointi on suoritettu ja sarana-alkio vaihdettu oikealle paikalleen, siirretään edellisessä partitioinnissa käytetty sarana-alkio s merkiksi s_1:n oikealle puolelle. Tämä on esitetty kuvassa 5, jossa p:llä on kuvattu s:ää edeltävää sarana-alkiota tai kuten tässä tapauksessa, äärettömän suurta apualkiota. Näin s_1 ja s jakavat alijoukon A_1 kahteen osaan A_{1a} ja A_{1b}.

Samaa partitiointia ja sarana-alkioiden siirtelyä jatketaan eteenpäin algoritmin mukaan, kunnes vasen ja oikea reuna osoittavat samaan alkioon. Alijoukosta A_{1a} tulee aina seuraavan vaiheen A_1, joka jaetaan samalla tavalla vasemmanpuoleisen alkion avulla kahteen alijoukkoon. Tämän voi rinnastaa rekursiivisen pikalajittelun rekursiokutsuihin, jossa aina alkupään alijoukko lajitellaan ensin loppupään odottaessa vuoroaan.



Kuva 5. Iteratiivisen pikalajittelun eteneminen yleisesti partitioinnin jälkeen.

Kun A_{1a} on tarpeeksi pieni eikä sitä voida enää jakaa, siirrytään viimeisestä partitioinnista saatuun alijoukkoon A_{1b}. Alijoukko A_{1a} on nyt valmis ja sen koko riippuu muuttujasta m. Tämän jälkeen etsitään ensimmäiseksi edellisen partitioinnin sarana-alkiota s edeltävää sarana-alkiota p, joka on seuraava s:ää suurempi alkio. Tämän jälkeen siirretään sarana-alkio s takaisin edellisen partitioinnin sarana-alkion p viereen oikealle paikalleen, kuten kuvassa 6 näytetään. Muutetaan vasemman reunan paikkaa tallentava muuttuja k ja oikean reunan muuttuja r oikeiksi vastaamaan joukkoa A_{1b}. Seuraavaksi aloitetaan taas partitiointi, tällä kertaa A_{1b}:lle, jonka vasemmanpuoleisesta alkiosta tulee sarana-alkio. Tästä jatketaan kuten aikaisemmin A_{1a}:n kanssa.



Kuva 6. Iteratiivisen pikalajittelun eteneminen seuraavan alijoukon lajittelemiseen.

Kun partitiointeja ei voi enää jatkaa pidemmälle, siirrytään taas oikealle. Etsitään uusi suurempi sarana-alkio s:n oikealta puolelta, siirretään s omalle paikalleen p:n viereen, muutetaan muuttujat oikeiksi ja aloitetaan partitiointi. Tätä jatketaan, kunnes vasemman reunan paikka on varsinaisen lajiteltavan joukon oikealla puolella. Koko taulukko on nyt lajiteltu iteratiivisella pikalajittelulla, joten poistetaan kaksi apualkiota.

Iteratiivisessa pikalajittelussa on sama idea kuin rekursiivisessa versiossa, joten partitioinnin vertailujen ja siirtojen aikavaativuus on yhtä suuri. Näiden lisäksi tulee sarana- ja apualkioiden siirtoja, jotta algoritmi osaa toimia oikein. Siirtojen määrä on

kuitenkin pienempää kertaluokkaa kuin muiden operaatioiden aiheuttama kompleksisuus, joten keskimääräinen aikavaativuus on samaa luokkaa kuin rekursiivisenkin pikalajittelun: $C_{keskim.}(N) \in \Theta(N \log_2 N)$.

Iteratiivisessakin versiossa sarana-alkion voi valita satunnaisesti lajiteltavana olevasta joukosta tai valita keskimmäisen arvon esimerkiksi kolmesta ensimmäisestä alkiosta tai vaikka ensimmäisestä, keskimmäisestä ja viimeisestä alkiosta. Tämä nopeuttaa algoritmia ja tekee pahimmastakin tapauksesta keskimääräisen, paitsi jos kaikki alkiot ovat yhtä suuria.

2.4. Lomituslajittelu

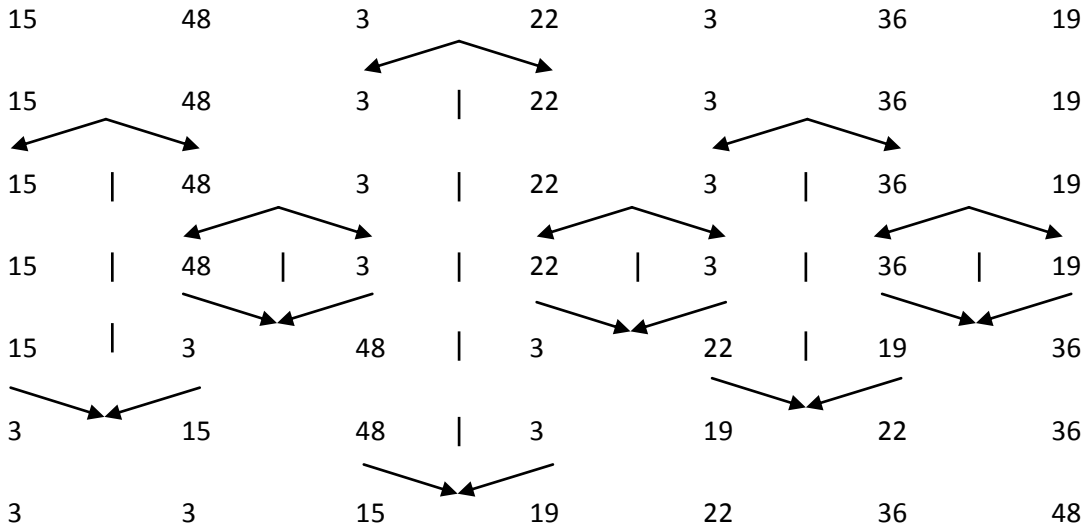
Lomituslajittelualgoritmi on idealtaan yksinkertainen hajota-ja-hallitse -menetelmä, jossa lajiteltava joukko jaetaan rekursiivisesti kahtia ja lopuksi yhdistetään pienet alijoukot lomittamalla. Algoritmi on vakaa, joten se pitää samanarvoisten alkioiden järjestyksen ennallaan. Lomituslajittelu kutsuu itseään rekursiivisesti molemmille alijoukoille, joten se vaatii paljon lisämuistia. Lajiteltava jaetaan aina puoliksi, joten muistitilaa tarvitaan lisää N / 2 rekursiotasolle ylöspäin pyöristettynä.

Lomituslajittelun pseudokoodi: (Levitin 2007)

```
LomitusLajittele(A[0...N – 1])
    // Lajittelee annetun syötteen lomituslajittelulla.
    // Syöte: Lajiteltava taulukko A[0...N – 1].
    // Tulos: Taulukko A[0..N – 1] lajiteltuna pienimmästä suurimpaan.
    if N > 1 then
        kopioi B[0... ⌊N/2⌋ – 1] ← A[0...⌊N/2⌋ – 1]
        kopioi C[0... ⌊N/2⌋ – 1] ← A[⌊N/2⌋...N – 1]
        LomitusLajittele(B[0... ⌊N/2⌋ – 1])
        LomitusLajittele(C[0... ⌊N/2⌋ – 1])
        Yhdistä(B, C, A)

Yhdistä(B[0...p – 1], C[0...q – 1], A[0...p + q – 1])
    // Yhdistää kaksi lajiteltua joukkoa yhdeksi lajitelluksi listaksi.
    // Syöte: Lajitelut taulukot B[0...p – 1] ja C[0...q – 1].
    // Tulos: B:stä ja C:stä yhdistetty lajiteltu taulukko A[0..p + q – 1].
    i ← 0; j ← 0; k ← 0;
    while i < p and j < q do
        if B[i] ≤ C[j] then
            A[k] ← B[i]; i ← i + 1
        else A[k] ← C[j]; j ← j + 1
        k ← k + 1
    if i = p then
        kopioi A[k...p + q – 1] ← C[j...q – 1]
    else   kopioi A[k...p + q – 1] ← B[i...p – 1]
```

Kuvassa 7 esitetään, miten lomittelulajittelu käytännössä toimii. Alussa alkiojoukko jaetaan kahteen osaan. Parittomassa tapauksessa ensimmäinen joukko on algoritmin mukaan kooltaan yhden pienempi kuin jälkimmäinen. Ensimmäinen jakaminen tapahtuu siis alkioiden 3 ja 22 välistä. Nämä kaksi alijoukkoa annetaan parametreinä rekursiivisille lomituslajittelukutsuille. Jos luetaan algoritmin koodia tarkasti, toinen rekursiivinen kutsu jää odottamaan edellisen valmistumista lajitelluksi joukoksi. Lomittelulajittelussa idean selventämiseksi käsitellään yleensä molempia rekursioita sekä muita tulevia rekursiokutsuja rinnakkain.



Kuva 7. Lomituslajittelun kulku. Pystyviivat erottavat jaetut alijoukot toisistaan, nuolet esittävät joukkojen jakamista sekä yhdistämistä lomittamalla.

Ensimmäisestä jaosta saadut kaksi alijoukkoa jaetaan uudelleen kahtia pienemmiksi alijoukoiksi, jolloin saadaan alijoukot {15}, {48, 3}, {22, 3} ja {36, 19}. Yhden kokoista joukkoa ei voi jakaa, joten se jää odottamaan muiden jakamisten valmistumista. Loput joukot jaetaan puoliksi, minkä jälkeen jäljellä on pelkkiä yhden alkion joukkoja.

Alkuperäinen syöte on monen jakamisen seurauksena pieninä yhden alkion joukkoina, joten seuraavaksi niitä yhdistetään lomittamalla ne suuremmiksi joukoiksi. Yhdistämisessä kutsutaan yhdistä-algoritmia, joka saa pienet alijoukot ja alkuperäisen joukon parametreinä. Yhdistäminen tapahtuu käänteisessä järjestyksessä jakamiseen nähden, joten ensimmäinen joukko {15} jää odottamaan. Ensimmäiseksi yhdistetään joukot {48} ja {3}, joista huomataan jälkimmäisen olevan pienempi. Lajiteltu yhdistetty joukko {3, 48} kopioidaan alkuperäiseen syötteeseen tämänhetkisille oikeille paikoilleen. Sama tehdään yhdistetyille joukoille {3, 22} ja {19, 36}.

Tämän jälkeen siirrytään rekursiopinossa yksi askel taaksepäin ja yhdistetään edellä saadut yhdistetyt joukot isommiksi joukoiksi. Nyt tulee mukaan myös yhden alkion joukko {15}, jonka ainoata alkiota verrataan joukon {3, 48} ensimmäiseen. Koska $3 < 15$, alkiosta 3 tulee yhdistetyn joukon ensimmäinen alkio. Seuraavaksi verrataan

jäljellä olevia kahta alkiota, jotka menevät myös suuruusjärjestykseen. Yhdistetyn joukon {3, 15, 48} alkiot kopioidaan taas alkuperäiseen joukkoon tämänhetkisille oikeille paikoilleen ja siirrytään seuraavaan yhdistämiseen.

Seuraavana yhdistetään joukot {3, 22} ja {19, 36}. Verrataan aina ensimmäisenä olevia alkioita, joista pienempi siirretään aina tulokseksi alkuperäiseen syötejoukkoon oikealle paikalleen. Nämä neljä päätyvät järjestykseen {3, 19, 22, 36}. Nyt jäljellä on enää kaksi joukkoa. Ne yhdistetään samalla tavalla vertailemalla aina molempien joukkojen ensimmäisiä alkioita. Ensimmäisenä vertaillaan kahta samanarvoista alkiota, joista valitaan ensimmäisen joukon alkio. Näin vakausominaisuus toteutuu ja alkuperäisessä syötteessä aikaisemmin ollut alkio tulee vastaan aikaisemmin myös lajitellussa joukossa. Tämän jälkeen valitaan muut alkiot taas joukkojen pienimpiä vertailemalla. Lopputuloksena on alkuperäinen syöte lajiteltuna pienimmästä suurimpaan.

Lajittelussa joukko jaetaan aina samalla tavalla uudelleen ja uudelleen pienempiin alijoukkoihin, kunnes jäljellä on yhden kokoiset alijoukot. Yhdistämisessä tulee pieniä eroja vertailujen määrässä alkuperäisen syötteen järjestyksen takia, mutta kaikenlaisilla syötteillä suuruusluokka pysyy samana. Yhdistämisessä tehdään pahimmassa tapauksessa N – 1 vertailua, jossa N on kahden joukon alkioiden yhteenlaskettu määrä. Oletetaan, että lajiteltavan koko on joku kahden potenssi eli $N = 2^k$. Tällöin lomituslajittelun kompleksisuus on

$$C(N) = 2C(N / 2) + C_{yhdist.}(N), \quad \text{kun } N > 1, \qquad C(1) = 0.$$
$$C_{pahin}(N) = C_{pahin}(N / 2) + N - 1, \text{ kun } N > 1, \qquad C_{pahin}(1) = 0.$$

Ratkaistaan rekursiokaava ja saadaan pahimman tapauksen kompleksisuudeksi

$$C_{pahin}(N) = N \log_2 N - N + 1 \in \Theta(N \log_2 N).$$

2.5. Shellin lajittelu

Shellin lajittelun kehitti vuonna 1959 Donald Shell, jonka mukaan algoritmi on nimetty (Knuth 1998). Algoritmista on olemassa myös muita nimityksiä kuten shell-lajittelu ja jopa shellistä suomennettu versio kuorilajittelu. Shellin lajittelu perustuu alkioiden vertailuihin ja vaihtoihin. Käytännössä se on lisäyslajittelun optimoitu versio, jossa vertailtavien alkioiden välimatkoja muutetaan. Aluksi verrataan alkiopareja pitkällä välillä, tämän jälkeen väliä lyhennetään asteittain. Lopulta lajitellaan yhden kokoisella välillä, joka vastaa lähes valmiiksi lajitellun joukon lajittelemista lisäyslajittelulla. Shellin lajittelu on epävakaa lajittelualgoritmi, mutta se toimii minimitilassa.

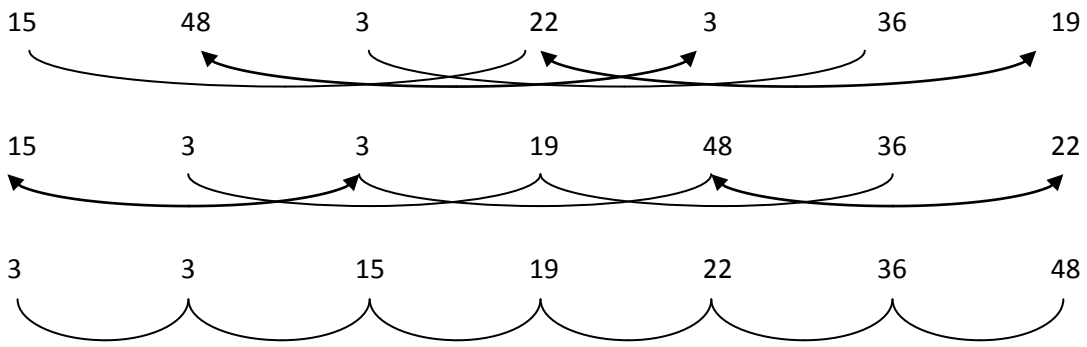
Shellin lajittelun algoritmi on muokattu selvempään ja käyttökelpoisempaan muotoon Donald E. Knuthin teoksesta The Art of Computer Programming (Knuth 1998).

Shellin alkuperäinen idea oli, että ensimmäinen vertailuväli olisi puolet syötteen koosta ja seuraavat välit olisivat aina puolet edellisestä, tarvittaessa alaspäin pyöristettynä. Tämä ei ole kuitenkaan paras vaihtoehto, sillä esimerkiksi 16 alkion kokoisella joukolla välit olisivat 8, 4, 2 ja 1, joista vasta viimeisellä vertailtaisiin parittomia ja parillisia alkioita keskenään. Ensimmäisen välin kannattaa olla maksimissaan 1 / 3 lajiteltavan joukon koosta, sillä suuremmilla väleillä vertailut vievät enemmän aikaa kuin tuovat

hyötyä (Knuth 1998). Algoritmissa käytetään Mark Weissin empiiristen testien perusteella ehdottamia välejä $2^k - 1$, ..., 15, 7, 3, 1 (Weiss 1991).

Shellin lajittelun pseudokoodi:

```
ShellLajittele (A[0...N – 1])
    // Lajittelee annetun syötteen Shellin lajittelulla.
    // Syöte: Lajiteltava taulukko A[0...N – 1].
    // Tulos: Taulukko A[0..N – 1] lajiteltuna pienimmästä suurimpaan.
    a; h; k
    s ← 2
    while 2^s – 1 ≤ N / 3 do s ← s + 1
    while s > 1 do
        s ← s – 1
        h ← 2^s – 1
        k ← h
        while k < N do
            a ← k
            while A[a – h] > A[a] do
                vaihda A[a] ja A[a – h]
                a ← a – h
            k ← k + 1
```



Kuva 8. Shellin lajittelun kulku. Vertailtavat alkiot yhdistetty viivoilla, vaihdot merkitty nuolilla.

Kuvassa 8 on esimerkki Shellin lajittelun kulun ideasta. Tässä esimerkissä ei edetä aivan algoritmin mukaan, sillä se lajittelisi näin lyhyen joukon ainoastaan yhden pituisella välillä. Tällöin se toimisi samalla tavalla kuin lisäyslajittelu. Muuttujaa h ei lasketa tässä tapauksessa siis algoritmin mukaan vaan esimerkissä joukko lajitellaan väleillä 3, 2 ja 1.

Lajittelu aloitetaan välillä, jonka pituus on 3. Vertaillaan pareja {15, 22}, {48, 3}, {3, 36}, {22, 19} sekä {15, 19}. Parien luvut järjestetään niin, että pienempi on ensimmäinen. Parin {22, 19} alkioiden paikkojen vaihtamisen jälkeen saadaan

viimeinen vertailtava pari, jossa vasemmalle siirtynyttä alkiota 19 verrataan ensimmäisenä olevaan alkioon 15.

Tämän jälkeen väliä pienennetään, jolloin vertaillaan kahden askeleen päässä olevia alkioita toisiinsa. Ensimmäisessä parissa {15, 3} alkioiden paikat vaidetaan, koska jälkimmäinen on pienempi. Seuraavat kolme paria {3, 19}, {15, 48} ja {19, 36} ovat jo oikeassa järjestyksessä. Tämän jälkeen vaihdetaan alkioparin {48, 22} alkiot keskenään, jonka jälkeen alkiota 22 verrataan vielä alkioon 15. Nyt joukko on lajiteltu väleillä 3 ja 2. Viimeisenä joukon vierekkäisiä alkioita verrataan, eli väli on 1. Joukko on kuitenkin jo oikeassa järjestyksessä, joten mitään ei enää tarvitse vaihtaa ja lajittelu on valmis.

Toinen näkökulma lajittelun ideaan on, että alkioista voidaan myös ajatella muodostuvan alijoukkoja, jotka lajitellaan lisäyslajittelulla. Suurimmalla välillä nämä joukot ovat {15, 22, 19}, {48, 3} ja {3, 36}. Kahden kokoisella välillä alijoukot ovat {15, 3, 48, 22} ja {3, 19, 36}. Yhden kokoisella välillä ei ole enää alijoukkoja, vaan koko alkuperäinen joukko lajitellaan vertailemalla vierekkäisiä alkioita samalla tavalla kuin lisäyslajittelussa.

Shellin lajittelulle ei ole yksiselitteistä aikavaativuutta, vaan siihen vaikuttavat lajiteltavan joukon koko ja tyyppi sekä käytettävien välien lukumäärä sekä niiden arvot. Parhaassa tapauksessa suoritusaika on valmiiksi lajitellulle syötteelle luokkaa O(N). Huonoimmassa tapauksessa pieni päinvastaisessa järjestyksessä olevan joukon lajittelu vie $O(N^2)$ ajan, mutta pienellä syötteellä lajitteluun käytettävä aika jää kuitenkin pieneksi. Jo kahdella erilaisella välillä suoritusaika saadaan tiputettua kuitenkin luokkaan $O(N^{5/3})$ (Knuth 1998), ja lisäämällä hyvin valittuja välejä algoritmista tulee vielä hiukan tehokkaampi.

Shellin lajittelulle löytyisi vielä Weissinkin ehdottamia välejä tehokkaampia välien laskukaavoja. Arvojen valinta ei ole kuitenkaan yksikertainen tehtävä. Asiasta on tehty jonkin verran testejä; esimerkiksi edellä mainitut Mark Weissin ja myös Donald Knuthin testit. Robert Sedgewick on saanut laskettua Shellin lajittelun pahimmalle tapaukselle aikavaativuudeksi $O(n^{4/3})$ omilla väleillään (Knuth 1998). Noiden välien kaava on kuitenkin huomattavasti monimutkaisempi. Tässä ei ole tarkoituksenmukaista tehdä mahdollisimman tehokasta monimutkaista algoritmia, vaan käytetään selvyyden ja yksikertaisuuden vuoksi helpompaa ja lähes yhtä tehokasta Weissin ehdottamaa tapaa laskea välit.

3. Lajittelumenetelmien empiirinen vertailu

Tässä kappaleessa testataan lajittelualgoritmien suoritusaikoja sekä muistinkäyttöä käytännössä. Kaikki algoritmit järjestävät samat kokonaislukujoukot, jotta menetelmien erot tulisivat esille. Joukkoja on viisi erilaista, ensimmäisessä joukossa on 1 000, toisessa 10 000, kolmannessa 100 000, neljännessä 1 000 000 ja viidennessä 10 000 000 alkiota. Lajittelua koitettiin myös 100 000 000 alkiolla, mutta Javan ArrayList-luokkaa ei ole tarkoitettu noin suurille joukoille, joten suoritus päättyi virheilmoitukseen. Osan algoritmeista olisi saanut muutettua sellaiseen muotoon, että niillä saisi lajiteltua suurempiakin joukkoja. Tällöin algoritmien toteutusta joutuisi kuitenkin muuttamaan reilusti, jolloin tämän vertailun idea häviäisi.

3.1. Testausmenetelmät ja kokoonpano

Suoritusajat on mitattu koodiin sisäänrakennetulla laskurilla, joka käynnistyy juuri ennen lajittelun aloittamista ja pysähtyy heti lajittelun päättymisen jälkeen. 100 000 ja enemmän alkioita sisältäville joukoille aikaa mitattiin System.currentTimeMillis() -metodilla, jonka antamista arvoista saatiin laskettua näiden kahden hetken välinen aika millisekunteina. Metodi on kuitenkin aivan liian epätarkka vielä muutamien kymmenienkin millisekuntien mittaamisessa, joten 1 000 ja 10 000 alkion lajittelussa ajanotossa käytettiin tarkempaa System.nanoTime() -metodia, josta nanosekunteina saadut tulokset muutettiin millisekunneiksi.

Lajittelualgoritmien muistinkäyttöä arvioitiin Windows 7:n omilla Suorituskyvyn valvonta ja Resurssienvalvonta-ohjelmilla. Näiden ohjelmien tihein päivitysnopeus on yksi hertsi eli kerran sekunnissa, joten reilusti alle sekunnin mittaisten lajitteluiden muistinkäyttöä ei näillä työkaluilla saa tarkasti testattua. Ohjelmat antoivat näistä lyhyistä suorituksista joitain suuntaa antavia tuloksia, mutta ne vaihtelivat huomattavasti meneillä olevasta vaiheesta riippuen. Tuloksia koitettiin tarkentaa suorittamalla samaa algoritmia monia kertoja peräkkäin ja ottamalla arvoista suurin. Vain reilusti yli sekunnin kestäneistä lajitteluista mitattuja muistinkäyttötuloksia voidaan pitää suurin piirtein oikeina ja luotettavina.

Testikoneena toimivassa pöytäkoneessa on seuraavanlainen kokoonpano:

Emolevy: MSI P45-Neo3-FR
Prosessori: Intel Core 2 Duo E8500 @3,17 GHz (kaksi prosessoriydintä)
Keskusmuisti: 2 x Team Elite 2048 MB DDR2 (5-5-5-15) 800 MHz
Kiintolevyt: C: 60 GB G.Skill SSD (SandForce SF-1200), Sata II (Windows, Java)
K: 1000 GB Samsung SpinPoint F3, HD103SJ, Sata II (lajittelualgoritmit)
Näytönohjain: Asus Radeon HD5870 1GB
Käyttöjärjestelmä: Windows 7 Ultimate 64-bittinen, Service Pack 1

Testin algoritmit on kirjoitettu Javalla. Ne suoritettiin komentoriviltä, ja niistä otettiin viiden suoritusajan keskiarvot, jotta muista taustalla suoritettavista ohjelmista aiheutuvat häiriöt saatiin minimoitua. Suorittimia ja taustaprosesseja ei voi kuitenkaan

valvoa kovin hyvin, joten niistä aiheutuneet muutamat selvästi virheelliset arvot jätettiin huomioimatta. Testien syötteinä käytettiin tekstitiedostoja, joissa oli erilaisia määriä erisuuruisia kokonaislukuja. Tiedostot luotiin tätä tarkoitusta varten tehdyllä pikkuohjelmalla, joka loi halutun määrän satunnaislukuja Javan Math.Random() -metodilla.

Lajittelumenetelmien koodit on tehty edellisessä luvussa esiteltyjen algoritmien pohjalta. Algoritmien kaikki taulukot on korvattu ArrayList-listarakenteella, joka on joustavampi käyttää kuin taulukko. Sen ansiosta esimerkiksi iteratiiviseen pikalajitteluun lisättävät kaksi apualkiota on helppo lisätä ja poistaa. Algoritmit käyttävät vain yhtä prosessoriydintä, eli suorittavat vain yhden operaation kerrallaan. Toinen prosessoriydin on siis käytettävissä muihin prosesseihin algoritmeja suorittaessa, joten taustalla suoritettavien ohjelmien ei pitäisi vaikuttaa merkittävästi lajitteluun käytettyyn aikaan.

Algoritmit testattiin ennen mittauksia toimiviksi testejä huomattavasti pienemmillä syötemäärillä. Lajittelun jälkeen tarkastettiin lajittelutuloksen oikeellisuus sekä algoritmin oikea toimintalogiikka välitulosteiden avulla. ArrayList-tyypillä on olemassa myös oma sisäänrakennettu lajittelumetodinsa, jonka kutsu on Collections.sort([lajiteltava_lista]). Tämä otetaan mukaan vertailuun työssä esiteltyjen algoritmien ulkopuolelta. Se osaa hyödyntää useita prosessorin ytimiä ainakin osittain, joten se poikkeaa muista vertailun algoritmeista. Menetelmän toimintatapaa ei selvitetä, mutta lajittelun tuottaman lopputuloksen tarkastettiin olevan oikein.

3.2. Lajittelualgoritmien nopeustesti

Alkioiden lkm.	1 000	10 000	100 000	1 000 000	10 000 000
Lisäyslajittelu	14 ms	71 ms	6,6 s	n. 41 min	ei tulosta*
Valintalajittelu	12 ms	260 ms	49 s	n. 2 h 7 min	ei tulosta*
Rek. pikalajittelu	4,8 ms	25 ms	72 ms	490 ms	7,1 s
Iter. pikalajittelu	6,6 ms	39 ms	110 ms	630 ms	9,3 s
Lomituslajittelu	8,6 ms	48 ms	97 ms	720 ms	12 s
Shellin lajittelu	7,5 ms	18 ms	110 ms	2,2 s	53 s
Collections.sort()	2,4 ms	22 ms	84 ms	450 ms	5,4 s

Taulukko 1. Lajittelualgoritmien testien kestojen keskiarvot erikokoisilla syötteillä. * Lisäys- ja valintalajittelualgoritmit keskeytetty, kun testit ovat kestäneet yli 24 h.

Lajittelumenetelmien testeistä saadut keskiarvot suoritusajoille ovat taulukossa 1. Pienillä syötteillä lajittelumenetelmien suoritusajat eivät poikkea toisistaan kovin paljon. 1 000-alkioisen joukon lajitteleminen kesti kaikilla menetelmillä vain jokusia millisekunteja, joten mikä tahansa menetelmä sopii vielä tuhannen alkion lajittelemiseen jopa tehottomammallakin suorittimella. Rekursiivinen pikalajittelu oli odotetusti nopein algoritmi 4,8 ms suoritusajalla. Sen perässä tulivat iteratiivinen pikalajittelu, Shellin lajittelu ja lomituslajittelu muutaman millisekunnin sisällä. Lisäyslajittelu oli hieman yllättävästi hitain kestäen 2 ms kauemmin kuin

valintalajittelu. Ulkopuolisena vertailuun tullut lajittelumetodi oli selvästi vielä rekursiivista pikalajitteluakin nopeampi suoriutuen puolet lyhyemmässä ajassa. Käytännössä nämä erot ovat kuitenkin niin pieniä, ettei niitä voi huomata. Edes Javan millisekunneissa aikaa mittaava menetelmä ei pystynyt mittaamaan näin pieniä aikoja, minkä vuoksi arvot piti mitata nanosekunteina.

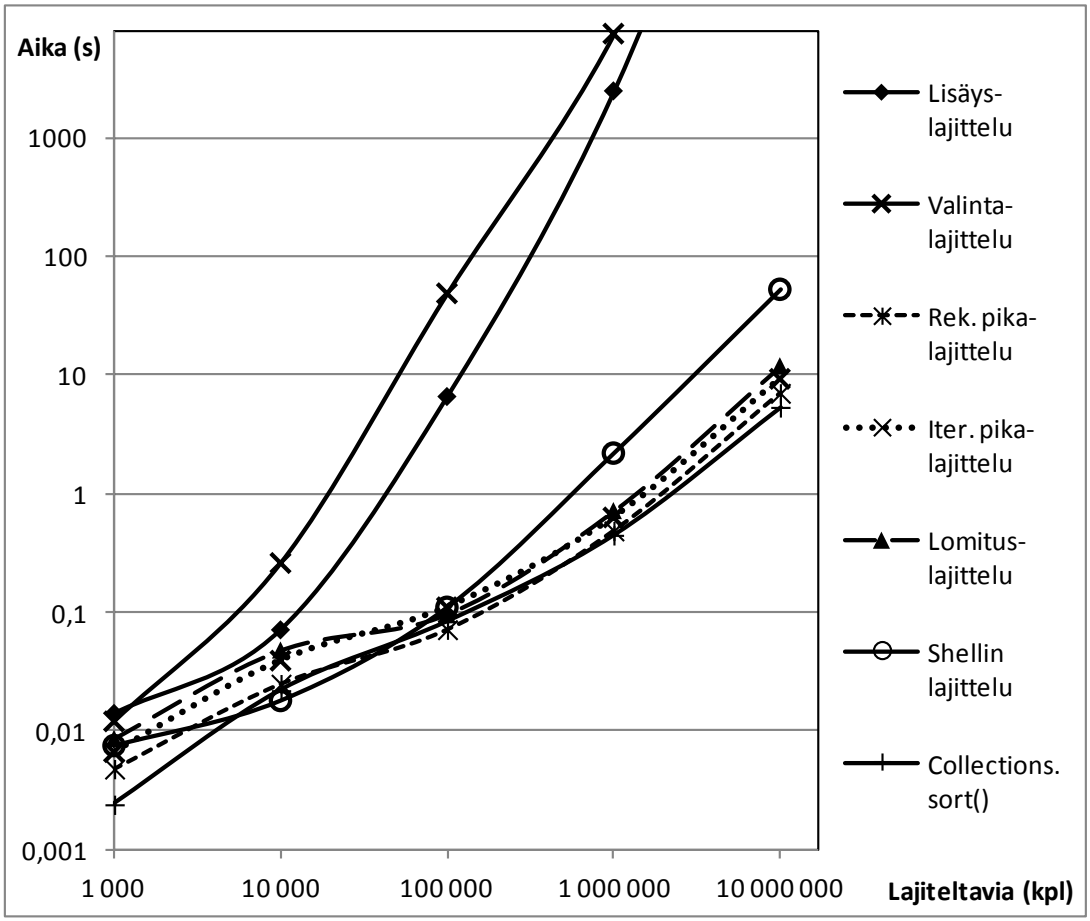
Lajiteltavan joukon koon kasvaessa 10 000 alkioon suoritusajat ja erot moninkertaistuvat. Shellin lajittelun aika kasvaa vain hieman yli kaksinkertaiseksi 18 ms:iin, joka on yllättäen jopa sisäänrakennettua menetelmää neljä millisekuntia nopeampi. Rekursiivisen pikalajittelun aika viisinkertaistuu syötteen koon kymmenkertaistuessa, jolloin se on Shellin lajittelua seitsemän millisekuntia hitaampi. Iteratiivinen pikalajittelu ja lomituslajittelu käyttävät yli kaksi kertaa niin paljon aikaa kuin nopeimmat menetelmät, 39 ms ja 48 ms. Lisäyslajittelun suoritusaika on kasvanut jo 71 ms:iin, joka on jo yli kolminkertainen nopeimpiin verrattuna. Nämä ovat kuitenkin vielä niin pieniä aikaeroja, että niitä on vaikea huomata. Tämäkin vaihe mitattiin nanosekunteina, jotta saatiin eroja aikaiseksi. Valintalajittelu on jäänyt selvästi viimeiseksi 260 ms:lla, joka on enemmän kuin muiden menetelmien yhteenlaskettu aika. Muita menetelmiä hitaampi valintalajittelualgoritmi alkaa jo erottua selvästi joukosta. Sen suoritusajan pystyy jo erottamaan muista epätarkemmillakin testeillä.

Alkiomäärän kasvaessa 100 000:een viisi nopeinta menetelmää ovat vielä hyvin samassa suuruusluokassa. Nopeimmin suoriutuu tällä kertaa rekursiivinen pikalajittelu, joka käyttää aikaa vain 72 ms. ArrayListin oma lajittelumetodi on vasta toiseksi nopein 84 ms:n kestolla ja lomituslajittelu tulee perässä 97 ms:lla. Neljännen sijan jakavat iteratiivinen pikalajittelu ja Shellin lajittelu 110 ms:n ajalla. Nekin ovat vielä tarpeeksi nopeita ja käyttökelpoisia moneen tarkoitukseen. Toiseksi hitain on lisäyslajittelu 6,6 sekunnin suoritusajalla ja selvästi hitain on valintalajittelu 49 sekunnin ajalla. Näistä etenkin valintalajittelu on jo käyttökelvoton tässä kokoluokassa, ja lisäyslajittelullekin löytyy huomattavasti parempia vaihtoehtoja.

Miljoonan alkion lajittelussa kahta ydintä käyttävä ArrayListin oma metodi on odotetusti nopein 450 ms:n ajalla. Rekursiivinen pikalajittelu jää kuitenkin vain 40 ms hitaammaksi ja on selvästi algoritmeista nopein. Iteratiivinen versio pikalajittelusta käyttää lajittelemiseen aikaa keskimäärin 630 ms ja lomituslajittelu 720 ms, mitkä ovat vielä hyviä suoritusaikoja näin isolle joukolle. Shellin lajittelun 2,2 sekuntia alkaa olla jo hidas verrattaessa nopeimpiin. Lisäyslajittelun n. 41 minuuttia ja valintalajittelun yli kaksi tuntia kestävät suoritusajat tekevät niistä täysin käyttökelvottomia näin isoilla syötteillä.

Kymmenen miljoonan alkion lajitteleminen ei ole enää kovin nopeaa millään algoritmilla. Selvästi nopein on tässäkin useampaa ydintä hyödyntävä sisäänrakennettu metodi, joka suoriutuu tehtävästä 5,4 sekunnissa. Rekursiivinen ja iteratiivinen pikalajittelu ovat kaksi seuraavaa ajoilla 7,1 ja 9,3 sekuntia. Rekursiivinen on tällöin 31 % hitaampi ja iteratiivinen 72 % hitaampi kuin Javan oma metodi. Lomituslajittelu on lähes 3 sekuntia hitaampi kuin iteratiivinen pikalajittelu. Shellin lajittelu on jäänyt suoritusajassa jo kauas neljän parhaan joukosta, sillä 53 sekunnin kesto on jo liian pitkä

aika. Lisäys- ja valintalajittelut jakavat viimeisen sijan, sillä niiden suoritukset keskeytettiin 24 tunnin jälkeen tuloksettomina. Valintalajittelulle tehtiin karkea arvio suorituksen kestoajaksi. Suoritusajaksi näin isolla syötteellä arvitoitiin noin kaksi viikkoa.



Kuva 9. Lajittelumenetelmien suoritusaika lajiteltavan syötteen koon funktiona.

Kuvassa 9 ovat kaikki lajittelumenetelmät kuvattuina samassa kaaviossa. Lisäys ja valintalajittelu eroavat muusta joukosta jo pienillä syötteillä. Shellin lajittelu alkaa erottua muusta joukosta 100 000 alkion jälkeen, mutta suuremmilla syötteillä sen kasvuvauhti alkioiden lisääntyessä on samaa suuruusluokkaa kuin pikalajitteluilla ja lomituslajittelulla. Kaaviota tutkittaessa kannattaa huomata, että molemmat akselit kasvavat logaritmisesti. Kahden vaakaviivan välissä suoritusaika on jo kymmenkertaistunut.

Tässä testissä kaikki algoritmit suorittivat vain yhtä operaatiota kerrallaan, jotta eri menetelmien nopeuserot olisivat nähtävissä. Rekursiivisia kutsuja käyttävien rekursiivisen pikalajittelun ja lomituslajittelun suoritus olisi aika helppo jakaa usealle suoritinytimelle, jolloin niistä saisi helposti muokattua vielä huomattavasti nopeampia. Algoritmeja voi nopeuttaa myös poistamalla turhia operaatioita, kun lajiteltava alijoukko on pieni. Tätä käsitellään tarkemmin seuraavassa luvussa (luku 3.3).

Nopeimmat algoritmit pärjäsivät hyvin Javan omalle lajittelumenetelmälle, joka on helppokäyttöisyytensä takia hyvä valinta. Jos haluaa olla varma, että syöte lajitellaan vakaalla menetelmällä, siinä tapauksessa lomituslajittelu on paras vaihtoehto. Jos taas haluaa optimoida suoritusaikoja, pikalajittelut ovat hyviä vaihtoehtoja.

3.3. Pikalajitteluiden nopeuttaminen

Luvussa 2 esiteltiin algoritmien toimintaa esimerkkisyötteillä. Rekursiivisessa ja iteratiivisessa pikalajittelussa esiintyy pienillä syötteillä paljon alkion vaihtamista edestakaisin sekä paikan vaihtamista itsensä kanssa. Molemmat pikalajittelut ovat tehokkaita ja nopeita isoilla syötteillä, mutta niitä voi helposti vielä nopeuttaa. Muita algoritmeja ei voi nopeuttaa yhtä helposti, joten tässä käsitellään vain pikalajittelumenetelmien tehostamista.

Iteratiivisen version alkuperäisessä versiossa jätettiin kymmenen alkion kokoiset ja pienemmät lajittelematta, ja vasta algoritmin lopussa koko joukko lajiteltiin nopeasti lisäyslajittelulla. Myös rekursiivisen pikalajittelun saa helposti lajittelemaan vain tiettyyn pisteeseen asti.

Iteratiivisessa pikalajittelussa nopeuttaminen on helppoa alustamalla muuttuja m jollain nollaa suuremmalla arvolla ja lisäämällä loppuun kutsu lisäyslajittelulle. Rekursiivisessa versiossa koodia joutuu hieman muokkaamaan, jotta siitä tulisi järkevä ja tehokas. Jos joukon lajittelisi vain lisäyslajittelulla lopuksi, kuten iteratiivisen version kohdalla, ei saisi kovin paljon hyötyä. Sarana-alkioita on turha lajitella, sillä ne ovat jo omilla paikoillaan. Koko joukon lajittelua parempi tapa on lisätä algoritmiin lisäyslajittelu niin, että tarpeeksi pienet ositteet lajitellaan sillä rekursiivisen kutsun sijaan. Muutos ei koske partitiointia mitenkään, joten se pysyy muuttumattomana.

Rekursiivisen lajittelun muutettu koodi:

```
RekPikaLajittele(A[v...o])
    if o – v > m then
        s ← Partitioi(A[v…o])
        RekPikaLajittele(A[v...s – 1])
        RekPikaLajittele(A[s + 1...o])
    else
        for i ← v to o do
            v ← A[i];
            j ← i – 1;
            while j ≥ 0 and A[j] > v do
                A[j + 1] ← A[j]
                j ← j – 1
            A[j + 1] ← v
```

Lajitteluvertailu nopeutetuille pikalajitteluille tehtiin 10 miljoonan alkion joukolla, jotta nopeuden muutokset näkyisivät selvemmin. Suoritusaika on viiden suoritusajan keskimääräinen kesto, jotta pienet heitot saadaan minimoitua. m arvolla 0 on aika, joka kului lajiteltaessa pelkästään pikalajittelulla. Suoritusaikoja mitattiin m:n arvoilla 2, 3, 6 ja 9.

m	0	2	3	6	9
Rek. pikalajittelu	7,05 s	6,95 s	6,87 s	6,55 s	6,56 s
Iter. pikalajittelu	9,25 s	9,05 s	8,95 s	8,14 s	8,08 s

Taulukko 2. Pikalajittelun kesto 10 000 000 alkion lajittelussa, jossa m on alijoukon koko, joka lajitellaan lisäyslajittelulla.

Taulukkoon 2 on kerätty mittaustulosten keskiarvot. Jo m:n arvolla 2 huomaa pienen nopeuseron, sillä rekursiivinen pikalajittelu nopeutuu 0,10 s. Arvolla 3 rekursiivinen versio nopeutuu jo 0,18 s, joka on 2,5 % alkuperäisestä. Kun muuttujalle m annetaan arvoksi 6, rekursiivinen pikalajittelu on jo puoli sekuntia nopeampi kuin ilman lisäyslajittelua. Tämän jälkeen m:n kasvaessa lisäyslajittelulla täydennetty versio alkaa muuttua jo hitaammaksi. Rekursiivinen pikalajittelu nopeutuu parhaimmillaan 0,5 s, kun m saa arvon 6. Tällöin rekursiivinen pikalajittelualgoritmi on noin 7,1 % alkuperäistä nopeampi, jolloin se on vain 21 % hitaampi kuin Javan oma lajittelumetodi.

Iteratiivinen pikalajittelu nopeutuu 0,10 sekuntia m:n ollessa kaksi. Kun m:ksi asetetaan kolme, suoritusajasta häviää 0,20 sekuntia eli pari prosenttia. Kun muuttujaa m kasvatetaan arvoon 6, algoritmi on jo yli sekunnin alkuperäistä nopeampi, mikä tarkoittaa jo 12 %. Paras suoritusaika saadaan algoritmin alkuperäisellä ehdotuksella, jolloin m on 9. Tällöin suoritusajasta saadaan vielä muutamia sekunnin sadasosia pois. Suuremmilla m:n arvoilla lisäyslajittelua käyttävä versio alkaa jo hidastua reilusti, jolloin alkuperäinen tapa on nopeampi. Muuttujan m arvolla 9 iteratiivisen pikalajittelun suoritusaika parani 12,6 % alkuperäisestä, jolloin se on n. 50 % hitaampi kuin Javan oma lajittelumetodi.

3.4. Lajittelumenetelmien muistinkäyttötesti

Alkioiden lkm.	1 000	10 000	100 000	1 000 000	10 000 000
Tiedoston koko	6 kt	63 kt	720 kt	7,0 Mt	80 Mt
Lisäyslajittelu	**	18 Mt	27 Mt	82 Mt	480 Mt
Valintalajittelu	**	17 Mt	27 Mt	66 Mt	480 Mt
Rek. pikalajittelu	**	18 Mt	39 Mt	120 Mt	660 Mt
Iter. pikalajittelu	**	18 Mt	36 Mt	120 Mt	640 Mt
Lomituslajittelu	**	25 Mt	42 Mt	190 Mt	670 Mt
Shellin lajittelu	**	18 Mt	36 Mt	290 Mt	770 Mt
Collections.sort()	**	17 Mt	29 Mt	85 Mt	670 Mt

Taulukko 3. Lajittelualgoritmien muistinkäyttö testien aikana eri syötemäärillä.
** 1 000 alkion lajittelusta ei saatu arvoja lyhyen keston takia.

Muistinkäyttöä on vaikea mitata tarkasti tässä käytetyillä työkaluilla näin lyhyillä suoritusajoilla. Käyttöjärjestelmä varaa ohjelmalle muistia tietyn määrän, joka muuttuu koko ajan tarpeen mukaan. Tässä testissä käytetty Windows 7 ja Java eivät varmasti ole ideaalisin vaihtoehto muistinkäytön suhteen. Arvoista on otettu suurin arvo Javan työmuistin huipulle. Muita Javaa käyttäviä ohjelmia ei ollut päällä, joten Javalle varattu muisti on lajittelualgoritmien käytössä.

Taulukossa 3 on muistitestien tuloksien keskiarvot. Tuhannella alkiolla lajittelut kestävät vain millisekunteja, joten sekunnin välein mittaavalla menetelmällä ei saatu tulosta näin lyhyistä suorituksista. Kun alkioiden määrä kymmenkertaistui 10 000 alkiooon ja suoritusajatkin moninkertaistuivat, saatiin jo joitain tuloksia. Näitä tuloksia ei voi pitää täysin tarkkoina, sillä arvot heittelivät niin paljon. Lomituslajittelua lukuun ottamatta kaikki olivat yhden megatavun sisällä, eikä lomituslajittelukaan vienyt kuin seitsemän megatavua enemmän muistia. Muistin tarve yllätti suuruudellaan, sillä alkiot sisältävän tiedoston koko on vain kymmeniä kilotavuja, eli parin promillen luokkaa Javan muistinkäytöstä. Javan tavallisten luokkien sisällyttäminen koodiin vie todennäköisesti suurimman osan muistista.

Tiedoston ja joukon koon kymmenkertaistuessa muistinkäyttö luonnollisesti kasvaa. Pienimmällä muistilla selviävät lisäys- ja valintalajittelu, joiden pitäisikin viedä vähiten tilaa. Myös ArrayListin oman metodin muistinkäyttö pysyy hyvin kurissa ollen vain kaksi megatavua suurempi. Shellin lajittelu ja iteratiivinen pikalajittelu ovat odotetusti seuraavat, joskin minimitilassa toimivina ne tarvitsevat yllättävän paljon enemmän muistia lisäys- ja valintalajitteluun verrattuna. Rekursiivisia kutsuja käyttävät rekursiivinen pikalajittelu ja lomituslajittelu ovat odotetusti suurimpia muistinkäyttäjiä.

Miljoonan alkion lajittelussa suoritusajat ovat jo hiukan pidempiä, joten tulokset ovat hieman parempia kuin pienimmillä syötteillä. Tosin ohjelma saattaa käyttää eri vaiheissa eri määriä muistia, joten nämäkään eivät ole absoluuttisia totuuksia. Valintalajittelu käyttää selvästi vähiten muistia, 66 Mt. Seuraavina tulevat odotetusti lisäyslajittelu sekä sisäänrakennettu menetelmä reiluilla 80 Mt:n muistinkäytöillä. Molemmille pikalajitteluille on annettu yhtä paljon muistia käytettäväksi, joten iteratiivisuudesta ei näy olevan suurempaa hyötyä ainakaan tässä tapauksessa. Lomituslajittelu on odotetusti eniten muistia vaativien joukossa 190 Mt:lla. Yllättäen eniten muistia käyttää Shellin lajittelu, jonka pitäisi toimia minimitilassa. Shellin lajittelulle tietokone varasi 290 Mt keskusmuistia.

Kymmenelle miljoonalle alkiolle tiedostokoko on jo 80 Mt. Lajittelussa pienimmillä muistimäärillä toimivat lisäys- ja valintalajittelu 480 Mt:lla. Niille saatiin hyvä tasainen tulos, vaikka testit lopetettiin niiden liian pitkän suoritusajan vuoksi. Iteratiivinen pikalajittelu on näistä jo kaukana, sillä se käyttää jo 640 Mt:a muistia. Rekursiivinen versio käyttää tässä tapauksessa vain 20 Mt enemmän muistia, mikä on lopulta aika pieni ero. Lomituslajittelu ja sisäänrakennettu lajittelu käyttävät muistia vielä hiukan enemmän, 670 Mt:a. Eniten muistia käyttää tässäkin tapauksessa hiukan yllättäen Shellin lajittelu. Toisaalta nykyisten tietokoneiden keskusmuistit ovat yleensä vähintään useita gigatavuja ja servereistä muistia löytyy jo kymmeniä gigatavuja, joten Shellin

lajittelun 770 Mt:n muistinkäyttö ei pitäisi olla ongelma. Testikoneessa muistia oli vapaana testien aikanakin vielä noin gigatavun verran, joten Javalle on saatettu osittaa muistia enemmän kuin se on tarvinnut.

Yhteistä kaikille tuloksille on, että tarvittavan muistin määrä on moninkertainen syötetiedoston kokoon verrattuna. Muistinkäytöllisesti ei saatu kovin suuria eroja eri algoritmien välille. Algoritmien muistinkäytön järjestys oli suurinpiirtein odotettu Shellin lajittelua lukuun ottamatta. Tarkempaa analyysiä muistin käytöstä on vaikea tehdä, sillä käytetyt työkalut antoivat vain Javalle annetun kokonaismuistin määrän. Ei ole tiedossa, kuinka paljon muistia tästä määrästä oikeasti oli käytössä ja kuinka paljon sitä tarvitaan alkioiden tallentamiseen, apualkiohin ja esimerkiksi rekursiopinojen tallennukseen.

4. Yhteenveto

Lajittelumenetelmiä on monenlaisia ja ne lähestyvät samaa ongelmaa eri suunnilta. Lopputulos on kuitenkin useimmiten sama. Poikkeuksen muodostaa se, että jotkut muuttavat keskenään yhtä suurien alkioiden paikkaa. Jos haluaa järjestää ensisijaisesti arvon x mukaan ja toissijaisesti arvon y mukaan, on väliä millä lajittelee ja miten. Pienemmillä syötteillä on helppoa ja kannattavaa käyttää vakaata algoritmia ongelmien välttämiseksi. Suurilla syötteillä täytyy miettiä kompromissia suoritusajan, muistinkäytön ja monimutkaisuuden suhteen. Lajittelu vakaalla algoritmilla voi olla paljon hitaampaa kuin lajittelu nopeimmalla tavalla ja lajitella tämän jälkeen samanarvoiset toissijaisen arvon mukaan. Tällainen menetelmien yhdistely taas vaatii hiukan työtä, jotta saa kyseisen koodin varmasti toimivaksi. Jos tarvitsee lajitella toistuvasti todella suuria syötteitä vakaasti, yhdistely saattaa kannattaa, mutta satunnaista käyttöä ajatellen vain hiukan hitaampi vakaa menetelmä on usein paras vaihtoehto.

Rekursiivinen ja iteratiivinen pikalajittelu ovat nimiensä mukaisesti nopeita kaikilla testatuilla syötteillä. Ne eivät ole parhaimmillaan aivan pienimpien joukkojen kanssa, joten niitä pystyy vielä helposti nopeuttamaan lisäyslajittelun avulla. Lomituslajittelu jää nopeudessa hieman pikalajitteluita hitaammaksi, mutta vakautta vaadittaessa se on paras ja nopein vaihtoehto, paitsi jos muistia on todella vähän käytössä. Lisäyslajittelu on helppo ja kätevä algoritmi pienempien joukkojen lajittelemiseen. Se on vakaa ja toimii minimitilassa, joten se on usein paras vaihtoehto, jos syöte ei ole iso eikä haeta vain nopeinta mahdollista suorituskykyä. Valintalajittelun paras puoli on yksinkertainen idea. Sen pystyy koodaamaan koska vain pieneen projektiin ilman koodin etsimistä internetistä ja suurempaa algoritmeihin perehtymistä.

Alle muutaman tuhannen alkion lajittelussa ei ole oikeastaan väliä mitä algoritmia käyttää, sillä suoritusaikojen erot ovat huomaamattoman pieniä. Ensimmäisenä suoritusajoissa muista erottuu valintalajittelu, jonka suoritusajan saattaa huomata jo muutaman tuhannen alkion joukossa. Lisäyslajittelua voi käyttää helposti vielä reilun 10 000 alkion lajitteluissa. Shellin lajittelu alkaa erottua viimeisenä joukosta noin miljoonan alkion kokoluokassa. Rekursiivinen ja iteratiivinen pikalajittelu sekä lomituslajittelu pystyvät haastamaan kohtuullisesti useampaa ydintä hyödyntävän ArrayListin oman metodin vielä kymmenen miljoonan alkion kokoluokassakin.

Javalla ohjelmoitaessa muistia tarvitaan kertaluokkaa enemmän kuin mitä joukon tallentamiseen tekstitiedostossa vaaditaan. Muistinkäytön erot jäivät lopulta pieniksi, eikä pelkistä testin tuloksista pysty luotettavasti sanomaan, mikä algoritmeista toimii minimitilassa.

Lähteet

Ďurian, B. 1986, "Quicksort Without a Stack", *Lecture Notes in Computer Science*, vol. 233, s. 283 – 289.

Hoare, C.A. 1962, "Quicksort", *The Computer Journal*, vol. 5, no. 1, s. 10 – 16.

Knuth, D.E. 1998, *The art of computer programming. Volume 3 Sorting and searching*, Addison-Wesley, Reading (MA), s. 83 – 95.

Levitin, A. 2007, *Introduction to the design & analysis of algorithms*, Pearson/Addison-Wesley, Boston.

Oracle 2012, , *ArrayList (Java Platform SE 7)*. Viittauspäivämäärä: 19.3.2013 http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Weiss, M.A. 1991, "Short Note Empirical study of the expected running time of Shellsort", *The Computer Journal*, vol. 34, no. 1, s. 88 – 91.

Liitteet:

Lisäyslajittelun algoritmi:

```
import java.util.*;
import java.io.*;

public class InsertionSort {
	public static void main(String... args) throws Exception {
		long aika;
		BufferedReader lukija = new BufferedReader(
			new  FileReader("lajiteltava_taulukko.txt"));
		ArrayList<Integer> luvut = new ArrayList<Integer>();
		String rivi = lukija.readLine();
		while (rivi != null) {
			int luku = Integer.parseInt(rivi);
			luvut.add(luku);
			rivi = lukija.readLine();
		}
		lukija.close();
		aika = System.currentTimeMillis();
//		aika = System.nanoTime();
		lajittele(luvut);
//		aika = System.nanoTime() - aika;
		aika = System.currentTimeMillis() - aika;
		System.out.println(aika + " ms");
/*		System.out.println("lajiteltu syöte");
		for (int luku : luvut) {
			System.out.println(luku);
		}*/
	}
	public static void lajittele(ArrayList<Integer> lajiteltava) {
		for(int i = 0; i < lajiteltava.size(); i++){
			int v = lajiteltava.get(i);
			int j = i - 1;
			while (j >= 0 && lajiteltava.get(j) > v){
				lajiteltava.set(j + 1, lajiteltava.get(j));
				j = j - 1;
			}
			lajiteltava.set(j + 1, v);
		}
	}

}
```

Valintalajittelun algoritmi:

```
import java.util.*;
import java.io.*;
```

```
public class SelectionSort {
	public static void main(String... args) throws Exception {
		// Sama kuin lisäyslajittelussa
	}
	public static void lajittele(ArrayList<Integer> lajiteltava) {
		int koko = lajiteltava.size();
		for(int i = 0; i < koko - 1; i++){
			int min = i;
			int j = 0;
			int pienin = 0;
			for (j = i + 1; j < koko; j++){
				pienin = lajiteltava.get(min);
				if (lajiteltava.get(j) < pienin){
					min = j;
					pienin = lajiteltava.get(j);
				}
			}
			int apu = lajiteltava.get(i);
			lajiteltava.set(i, pienin);
			lajiteltava.set(min, apu);
		}
	}
}
```

Rekursiivisen pikalajittelun algoritmi:

```
import java.util.*;
import java.io.*;

public class RecQuickSort {
	public static void main(String... args) throws Exception {
		long aika;
		BufferedReader lukija = new BufferedReader(new
			FileReader("lajiteltava_taulukko.txt"));
		ArrayList<Integer> luvut = new ArrayList<Integer>();
		String rivi = lukija.readLine();
		while (rivi != null) {
			int luku = Integer.parseInt(rivi);
			luvut.add(luku);
			rivi = lukija.readLine();
		}
		lukija.close();
		aika = System.currentTimeMillis();
//		aika = System.nanoTime();
```

```
            lajittele(luvut, 0, luvut.size()-1);
//          aika = System.nanoTime() - aika;
            aika = System.currentTimeMillis() - aika;
            System.out.println(aika + " ms");
/*          System.out.println("lajiteltu syöte");
            for (int luku : luvut) {
                System.out.println(luku);
            }*/
        }
        public static void lajittele(ArrayList<Integer> lajiteltava, int vasen, int oikea) {
            int m = 0;
            if (oikea – vasen > m){
                int s = partitioi(lajiteltava, vasen, oikea);
                lajittele(lajiteltava, vasen, s - 1);
                lajittele(lajiteltava, s + 1, oikea);
            }
        }
        static int partitioi(ArrayList<Integer> lajiteltava, int vasen, int oikea){
            int s = lajiteltava.get(vasen);
            int i = vasen;
            int j = oikea+1;
            do{
                do{
                    i++;
                }while (lajiteltava.get(i) < s && i < oikea);
                do{
                    j--;
                }while (lajiteltava.get(j) > s);
                int a = lajiteltava.get(i);
                lajiteltava.set(i, lajiteltava.get(j));
                lajiteltava.set(j, a);
            }while (i < j);
            int a = lajiteltava.get(i);
            lajiteltava.set(i, lajiteltava.get(j));
            lajiteltava.set(j, a);
            a = lajiteltava.get(vasen);
            lajiteltava.set(vasen, lajiteltava.get(j));
            lajiteltava.set(j, a);
            return j;
        }
}
```

Iteratiivisen pikalajittelun algoritmi:

```
import java.util.*;
import java.io.*;

public class IterQuickSort {
	public static void main(String... args) throws Exception {
		// Sama kuin lisäyslajittelussa
	}
	public static void lajittele(ArrayList<Integer> lajiteltava) {
		int l = 0; int r = lajiteltava.size(); int m = 0;
		int i, j;
		int n = r; int p = -1;
		lajiteltava.add((int)Float.POSITIVE_INFINITY-1);
		lajiteltava.add((int)Float.POSITIVE_INFINITY);
		do{
			while(r-l > m){
				i = l; j = r;
				p = lajiteltava.get(l);
				do{
					do{i++; }while(lajiteltava.get(i) < p && i < r-1);
					do{j– –; }while(lajiteltava.get(j) > p && j >= l);
					if(i < j){
						int a = lajiteltava.get(i);
						lajiteltava.set(i, lajiteltava.get(j));
						lajiteltava.set(j, a);
					}
				}while (i < j);
				lajiteltava.set(l, lajiteltava.get(j));
				lajiteltava.set(j, p);
				int a = lajiteltava.get(i);
				lajiteltava.set(i, lajiteltava.get(r));
				lajiteltava.set(r, a);
				r = j;
			}
			l = r;
			do{l++; }while(lajiteltava.get(l) <= p);
			if(l <= n){
				p = lajiteltava.get(l);
				r = l;
				do{r++; }while(lajiteltava.get(r) <= p);
				r--;
				lajiteltava.set(l, lajiteltava.get(r));
				lajiteltava.set(r, p);
```

```
                }
        }while (l <= n - 1);
        lajiteltava.remove(lajiteltava.size()-1);
        lajiteltava.remove(lajiteltava.size()-1);
    }
}
```

Lomituslajittelun algoritmi:

```
import java.util.*;
import java.io.*;

public class MergeSort {
        public static void main(String... args) throws Exception {
                // Sama kuin lisäyslajittelussa
        }
        public static void lajittele(ArrayList<Integer> lajiteltava) {
                int n = lajiteltava.size();
                ArrayList<Integer> B = new ArrayList<Integer>();
                ArrayList<Integer> C = new ArrayList<Integer>();
                if(n > 1){
                        for(int i = 0; i < n/2; i++){
                                B.add(lajiteltava.get(i));
                        }
                        for(int i = n/2; i < n; i++){
                                C.add(lajiteltava.get(i));
                        }
                        lajittele(B);
                        lajittele(C);
                        yhdista(B, C, lajiteltava);
                }
        }
        static void yhdista(ArrayList<Integer> B, ArrayList<Integer> C,
                        ArrayList<Integer> lajiteltava){
                int i = 0; int j = 0; int k = 0; int p = B.size(); int q = C.size();
                while(i < p && j < q){
                        if(B.get(i) <= C.get(j)){
                                lajiteltava.set(k, B.get(i));
                                i = i + 1;
                        }else{
                                lajiteltava.set(k, C.get(j));
                                j = j + 1;
                        }
                        k = k + 1;
```

```
            }
            if(i == p){
                for(int y = 0; y < q - j; y++){
                    lajiteltava.set(k + y, C.get(j + y));
                }
            }else{
                for(int y = 0; y < p - i; y++){
                    lajiteltava.set(k + y, B.get(i + y));
                }
            }
        }
}
```

Shellin lajittelun algoritmi:

```
import java.util.*;
import java.io.*;
import java.lang.Math.*;

public class ShellSort {
    public static void main(String... args) throws Exception {
        // Sama kuin lisäyslajittelussa
    }
    public static void lajittele(ArrayList<Integer> lajiteltava) {
        int h = 0; int s = 2; int a = 0; int k = 0;
        int koko = lajiteltava.size();
        while((int)Math.pow(2, s) - 1 <= koko/3){s++;}
        while(s > 1){
            s--;
            h = (int)Math.pow(2, s) - 1;
            k = h;
            while(k < koko){
                a = k;
                int b = lajiteltava.get(a);
                while(a >= h && lajiteltava.get(a-h) > b){
                    lajiteltava.set(a, lajiteltava.get(a-h));
                    lajiteltava.set(a-h, b);
                    a = a - h;
                }
                k++;
            }
        }
    }
}
```

ArrayListin oman lajittelualgoritmin käyttäminen:

```
import java.util.*;
import java.io.*;

public class CollectionsSort {
   public static void main(String... args) throws Exception {
            long aika;
      BufferedReader lukija = new BufferedReader(new
FileReader("lajiteltava_taulukko.txt"));
      ArrayList<Integer> luvut = new ArrayList<Integer>();
      String rivi = lukija.readLine();
      while (rivi != null) {
         int luku = Integer.parseInt(rivi);
         luvut.add(luku);
         rivi = lukija.readLine();
      }
      lukija.close();
            aika = System.currentTimeMillis();
//          aika = System.nanoTime();
            Collections.sort(luvut);
//          aika = System.nanoTime() - aika;
            aika = System.currentTimeMillis() - aika;
            System.out.println(aika + " ms");
/*          System.out.println("lajiteltu syöte");
       for (int luku : luvut) {
       System.out.println(luku);
       }*/
   }
}
```

Satunnaislukutaulukoiden luomiseen käytetty algoritmi:

```
import java.io.*;
import java.lang.Math.*;

public class randomnumbergenerator {
   public static void main(String... args) throws Exception {
            int lkm = 100000;
            int alku = -100000;
            int loppu = 100000;
            luoTaulukkoTiedostoon("taulukko" + lkm + ".txt", lkm, alku, loppu);
   }
   /**
    * Luo tiedoston, jossa on tiedoston nimen määrä satunnaislukualkioita
    * annetulta väliltä.
    * AE: tiedostonimi annettu oikein, lkm > 0, lo ja hi != null.
    * LE: taulukossa lkm kpl satunnaisalkioita väliltä [lo,hi].
    */
   public static void luoTaulukkoTiedostoon(String tiedosto, int lkm, int lo, int hi)
throws Exception{
            PrintWriter kirj = new PrintWriter(new FileWriter(tiedosto));
            for (int i = 0; i < lkm; i--){
                  kirj.println(rand(lo,hi));
            }
            kirj.close();
   }
   /**
        * Arpoo satunnaisen kokonaisluvun väliltä [lo, hi].
        * AE: lo ja hi kokonaislukuja, lo < hi
        * LE: palauttaa kokonaisluvun väliltä [lo, hi]
        */
   public static int rand(int lo, int hi){
       int n = hi - lo + 1;
       return (int)(n * Math.random() + lo);
   }
}
```

Nopeustestien tulokset.

	Lisäyslajittelu	Valintalajittelu	Rek. Pikalajittelu	Iter. pikalajittelu	Lomituslajittelu	Shellin lajittelu	Collections.sort()
1 000 alkiota (ms)	14,1	12,4	5,1	6,7	8,5	8,5	2,5
1 000 alkiota (ms)	14,2	11,6	5,1	6,4	8,5	6,6	2,5
1 000 alkiota (ms)	14,1	11,7	4,7	6,7	9,0	8,3	2,4
1 000 alkiota (ms)	14,0	11,6	4,6	6,5	8,5	7,6	2,4
1 000 alkiota (ms)	14,1	12,1	4,4	6,5	8,6	6,6	2,4
1 000 alkiota (ms) ka	14,1	11,9	4,8	6,6	8,6	7,5	2,4
10 000 alkiota (ms)	71,2	251	29,7	37,4	28,3	18,1	25,8
10 000 alkiota (ms)	72,6	295	18,4	38,1	60,2	17,5	27,8
10 000 alkiota (ms)	70,9	262	28,6	40,2	56,2	16,1	20,1
10 000 alkiota (ms)	70,0	253	21,3	43,2	65,0	21,8	20,5
10 000 alkiota (ms)	71,6	257	28,9	37,8	29,5	16,7	18,2
10 000 alkiota (ms) ka	71,3	264	25,4	39,3	47,8	18,0	22,5
100 000 alkiota (ms)	6552	48719	78	109	109	110	78
100 000 alkiota (ms)	6552	50918	78	124	93	109	93
100 000 alkiota (ms)	6552	49015	63	109	94	124	94
100 000 alkiota (ms)	6771	48719	62	125	94	110	78
100 000 alkiota (ms)	6552	48891	78	94	93	109	78
100 000 alkiota (ms) ka	6596	49252	72	112	97	112	84
1 000 000 alkiota (ms)	2495 s	7573 s	484	640	718	2122	453
1 000 000 alkiota (ms)	2465 s	7578 s	483	624	717	2153	437
1 000 000 alkiota (ms)	2454 s	7576 s	499	624	717	2122	437
1 000 000 alkiota (ms)	2453 s	7583 s	484	624	733	2293	452
1 000 000 alkiota (ms)	2470 s	7641 s	484	640	733	2122	468
1 000 000 alkiota (ms) ka	2467 s	7590 s	487	630	724	2162	449
10 000 000 alkiota (ms)	liian kauan	liian kauan	7052	9267	11996	52666	5663
10 000 000 alkiota (ms)	> 24 h	> 24 h	7020	9219	11154	53727	5382
10 000 000 alkiota (ms)			7067	9204	11544	52728	5382
10 000 000 alkiota (ms)			7051	9360	11311	53118	5382
10 000 000 alkiota (ms)			7067	9204	11950	53866	5413
10 000 000 alkiota (ms) ka	ei tulosta	ei tulosta	7051	9251	11591	53221	5444,4

Muistinkäyttö testien aikana

	Lisäyslajittelu	Valintalajittelu	Rek. Pikalajittelu	Iter. pikalajittelu	Lomituslajittelu	Shellin lajittelu	Collections.sort()
10 000 alkiota	18144 kt	17352 kt	17992 kt	18892 kt	25115 kt	17984 kt	17904 kt
100 000 alkiota	27908 kt	27136 kt	39876 kt	36752 kt	43 119 kt	37480 kt	29664 kt
1 000 000 alkiota	84435 kt	67343 kt	125476 kt	127232 kt	197975 kt	297096 kt	87312 kt
10 000 000 alkiota	491779 kt	490528 kt	675796 kt	656248 kt	687075 kt	783508 kt	684604 kt

Pikalajitteluiden nopeuttaminen

Rekursiivinen pikalajittelu, 10 M alkiota			
2	3	6	9
6,926	6,817	6,537	6,568
7,004	6,879	6,521	6,568
6,88	6,895	6,552	6,567
6,958	6,879	6,599	6,552
7,004	6,864	6,52	6,52
6,9544	6,8668	6,5458	6,555

Iteratiivinen pikalajittelu, 10 M alkiota			
2	3	6	9
9,438	8,939	8,315	8,33
8,923	8,939	7,972	8,081
8,908	8,955	8,127	8,175
9,001	8,954	7,925	7,909
8,954	8,986	8,361	7,904
9,0448	8,9546	8,14	8,0798