|  |
| --- |
| TKO\_2023 |
| Harjoitustyö |
| Dokumentti |

|  |
| --- |
| Tiina Nokelainen 503737  Veera Paananen 509803 |

# Tehtävän kuvaus

Harjoitustyön aiheeksi valikoitui viiden eri lajittelualgoritmin ohjelmointi Java -ohjelmointikielellä ja niiden toimivuuden sekä tehokkuuden testaus. Algoritmeiksi valitsimme kekolajittelun (eng. *heap sort*), kuplalajittelun (eng. *bubble sort*), pikalajittelun (eng. *quick sort*), lomituslajittelun (eng. *merge sort*) ja laskentalajittelun (eng. *counting sort*). Algoritmien tarkoitus on lajitella kokonaislukutaulukkoja suuruusjärjestykseen alkaen pienemmistä taulukon arvosta. Algoritmien testaukseen käytimme erilaisista kokonaisluvuista koostuvia taulukoita ja otimme erityisesti lajitteluun kuluvan ajan tarkasteltavaksi ominaisuudeksi algoritmien tehokkuuden vertailussa.

# Ratkaisuperiaate

Päätimme ohjelmoida meille tutulla Java -ohjelmointikielellä ja algoritmien implementoimisvaiheessa näimme järkeväksi luoda jokaiselle lajittelualgoritmille oma luokka. Luokista löytyy vain toiminnallisuudet taulukon lajittelun suorittamiseen. Itse testaaminen on toteutettu omassa luokassaan nimeltä Testing. Siellä randomize(int size) -metodilla luodaan yksi parametrina saadun kokonaisluvun kokoinen kokonaislukutaulukko, jonka arvot on satunnaisesti generoitu koodiin kirjoitetulta arvoväliltä, joka yhdessä testissä on esimerkiksi 0..99. Tämä kokonaislukutaulukko tallennetaan sitten Testing -luokan luokkamuuttujiin, joita on viisi kappaletta ja en toimivat testaustaulukoina jokaiselle viidelle algoritmille. Main -metodissa sitten jokaista algoritmia testataan ja välitulostusten avulla pääsee seuraamaan, toimiiko lajittelu oikein ja missä ajassa. Jokaiselle algoritmille siis käynnistetään ajan mittaamiseksi main-metodissa luotu Timer-olio, joka sitten mittaa testitaulukon lajitteluun kuluneen ajan. Jälleen välitulostuksilla voi seurata, miten algoritmien suoritukset eroavat ajallisesti toisistaan.

# Algoritmit

## Kekolajittelu - Heap sort

Kekolajittelussa taulukon alkioista muodostetaan ensin keko ja sen jälkeen muodostetaan maksimikeko, jotta kekoehto on voimassa. Maksimikeon juuresta löytyy aina taulukon suurin alkio. Päätimme muodostaa taulukon alkioista suoraan maksimikeon ja sille teimme Heapsort -luokkaan metodin heapify(). Keon muodostaminen tapahtuu kokonaislukutaulukko-olion arvojen uudelleenjärjestämisen avulla. Ne järjestetään kekomaisesti siinä järjestyksessä, missä ne taulukossa ovat aivan aluksi. Tämän jälkeen muodostetaan maksimikeko kutsumalla heapify() -metodia. Tarkastellaan siis alhaalta ylöspäin, että keossa olevat isäsolmut ovat aina lapsisolmuja suurempia. Kun keko on rakennettu, lajitellaan solmut siten, että juurisolmu vaihtaa viimeisen alkion kanssa paikkaa ja juureen vaihdetaan seuraavaksi suurin alkio. Vanha juuri poistuu keosta kokonaan. Vanhan juuren poistumisen jälkeen maksimikeko muodostetaan tarvittaessa uudelleen. Tätä jatketaan kunnes keko on tyhjä. Kun keko on tyhjä, lajittelu on valmis. Kekolajittelu ei ole stabiili lajittelualgoritmi ja sen aikavaativuus on T(n) = O(n) eli se suorituu lajittelusta lineaarisessa ajassa.

## Kuplalajittelu - Bubble sort

Kuplalajittelu on aikavaativuudeltaan O(n^2) eli hyvin hidas algoritmi. Lajittelussa taulukon vierekkäisiä alkioita vertaillaan keskenään ja jos niiden todetaan olevan väärinpäin, eli pienempi arvo on suuremman arvon oikealla puolella, vaihdetaan niiden paikkoja keskenään. Aina kun tällainen vaihto tapahtuu, toistetaan vertailu uudelleen. Vaihdon tapahtumisen seuraamiseen laitoimme luokkaan totuusarvomuuttujan *swapped*, joka saa arvon *true*, kun vaihto on tapahtunut. Algoritmi on hyvin yksinkertainen, muttei kovin tehokas. Varsinkaan, jos taulukon alkiot ovat epäjärjestyksessä erityisen paljon.

## Pikalajittelu – Quicksort

Pikalajittelu on aikavaativuudeltaan luokkaa O(n^2) mutta se suorituu lajittelusta keskimäärin ajassa O(n log2 n). Toimintaperiaate on, että taulukon alkiot partitioidaan kahteen osaan yhden määrätyn alkion suhteen, jota kutsutaan pivot- tai sarana-alkioksi. Pivot alkioksi voidaan valita esimerkiksi keskimmäinen tai viimeinen alkio. Päätimme itse käyttää pivot-alkiona taulukon viimeistä alkiota. Algoritmin partiointiosuuden aikakompleksisuus on Theta(n). Pahimmassa tapauksessa pikalajittelu tarvitsee neliöllisen suoritusajan eli T(n)= Theta(n^2). Parhaassa tapauksessa T(n)=O(nlog2 n). Satunnaistettu pikalajittelu toimii asymptoottisesti tarkkailtuna keskimääräisessä tapauksessa samassa ajassa kuin satunnaistamaton pikalajittelu parhaassa tapauksessa. Näin ollen satunnaistetut taulukot toimivat yhtä nopeasti kuin vastaava satunnaistamaton toimisi tehokkaimmassa suoritusajassa.

Meidän Quicksort -luokasta löytyy partition() -metodi, joka toteuttaa algoritmin paritiointiosuuden. Taulukko jaetaan siis kahteen osaan. Sitten siirretään pivot-alkion vasemmalle puolelle kaikki sitä pienemmät alkiot ja sen oikealle puolelle kaikki sitä suuremmat alkiot. Eli pivot-alkioon verrataan taulukon alkioita, mitkä ovat <=. Kun löydetään tällainen alkio, vaihdetaan sen paikka ensin indeksissä 1 olevan alkion kanssa, sitten seuraava vaihdetaan indeksissä 2 olevan alkion kanssa ja niin edelleen. Palautetaan lopulta indeksi, jossa pivot-alkio sijaitsee.

## Lomituslajittelu - Merge sort

Lomituslajittelu on aikavaativuudeltaan luokkaa O(nlogn) eli hyvinkin tehokas lajittelualgoritmi. Meillä testauksena olivat kokonaislukutaulukot, mutta parhaiten lomituslajittelusta on hyötyä linkitettyjen listojen lajitteluissa. Siinä myös taulukko jaetaan ensin kahteen osaan, jotka ovat kooltaan yhtäsuuria. Nämä taulukon osataulukot järjestetään rekursiivisesti ja järjestetys osataulukot limitetään sitten takaisin yhdeksi kokonaiseksi taulukoksi, josta löytyvät sitten taulukon arvot lajiteltuna suuruusjärjestykseen. Aivan kuten haluttiinkin.

Me teimme tämän algoritmin lajitteluun metodin, joka saa kaksi taulukkoa ja kaksi kokonaislukua parametreina. Ensimmäin taulukko on se, joka halutaan lajitella. Seuraavana on uusi yhtä suuri taulukko, jota käytetään apuna lajittelun aikana. Kokonaisluvut taas edustavat taulukon ensimmäisen alkion indeksiä ja taulukon viimeisen alkion indeksiä. Lomituslajittelussa metodi kutsuu rekursiivisesti itseään, joten nämä arvot tietenkin muuttuvat ohjelman suorituksen aikana. Taulukon jakamista osataulukoiksi jatketaan siihen saakka kunnes osataulukon pituus on vain yksi alkio. Sitten metodikutsuja lähdetään purkamaan takaisinpäin.

## Laskentalajittelu - Counting sort

Laskentalajittelu on aikavaativuudeltaan lineaarinen eli 0(n). Se perustuu jakauman laskemiseen Counting sort) on eräs lajittelualgoritmeista. Se perustuu jakauman laskemiseen ja on siksi hyvin tehokas lajittelualgoritmi tapauksiin, joissa n-suuruisen taulukon lajittelu halutaan tehdä ilman, että kokonaisia alkioita joudutaan vertaamaan jatkuvasti keskenään. Laskentalajittelussa käytimme apuna lomituslajittelun tapaan väliaikaistaulukkoa c, jota kutsutaan usein myös frekvenssivektoriksi.

Frekvenssitaulukkoonmuodostetaan niin, että sen ensimmäinen indeksi on lajiteltavan taulukon pienimmän alkion suuruinen ja sen viimeinen indeksi taas lajiteltavan taulukon suurimman alkion suuruinen. Jos luvut ovat siis väliltä 0…10, on frekvenssitaulukkoon koko tällöin 11. Kun frekvenssivektori on muodostettu, lasketaan jokaisen taulukon sisältämän kokonaisluvun esiintymien lukumäärät (jakauma) yhteen, ja tallennetaan tulos frekvenssitaulukkoon kyseisen luvun kohdalle. Tämän jälkeen tehdään arvojen kumulointi niin, että frekvenssitaulukon jokainen arvo sisältää kumuloinnin jälkeen itsensä summattuna kaikkiin edeltävin arvoihin. Esimerkiksi indeksissä 4 oleva arvo sisältäisi kumuloinnin jälkeen indeksin 4 olevan arvon ja indeksien 0, 1, 2 ja 3 olevien arvojen summan.

Kumuloinnin päätteeksi aluksi luotuun tulostaulukkoon, joka on siis samankokoinen kuin alkuperäinen lajiteltava taulukko, sijoitetaan frekvenssitaulukon arvojen perusteella alkuperäisen taulukon arvot. Lopuksi vielä tulostaulukon arvot tallennetaan lajiteltavaan alkuperäiseen taulukkoon, jolloin lajittelu on valmis.

# Testaukset

Tärkeimpänä ominaisuutena tarkastelimme algoritmien suoritusaikaa, kun niitä testataan erisuuruisilla kokonaislukutaulukoilla, joiden arvojen vaihteluväli myös vaihtelee. Testasimme esimerkiksi taulukkoa, jossa on kohtalaisen vähän alkioita, mutta jossa alkioiden vaihteluväli oli suuri. Sitten testasimme taulukolla, jossa on huomattavasti enemmän alkioita, mutta jossa alkioiden vaihteluväli on pieni. Kokosimme tuloksia testauksesta oheiseen taulukkoon. Jokaisella eri taulukon koolla sekä eri vaihteluvälillä testasimme jokaista algoritmia kymmenen kertaa, ja taulukosta löytyy näiden kymmenen testikerran keskiarvo.

Totesimme näiden testien perusteella, että kaikista tehokkain algoritmi oli laskentalajittelu, jos taulukon alkioiden vaihteluväli oli 0..99. Kun vaihteluväli pidettiin hyvin maltillisena eli välillä 0..9, laskentalajittelu oli kuitenkin hyvin paljon huonompi tehokkuudeltaan. Pelkästään jo tästä havainnosta voidaan päätellä, että jokaisella lajittelualgoritmilla on omat käyttötarkoituksena. Joihinkin tilanteisiin sopii paremmin toisenlainen algoritmi kun taas toisiin tilanteisiin käy parhaiten aivan eri tavalla toimiva algoritmi.

# Liitteet