

Heikki Jetsonen

# QUICKSORTIN, RADIX SORTIN JA SHELLSORTIN VERTAILU

# TIIVISTELMÄ

Heikki Jetsonen: Quicksortin, Radix sortin ja Shellsortin vertailu  
Kandidaatintyö  
Tampereen yliopisto  
Tietotekniikan kandidaatin tutkinto-ohjelma  
Syyskuu 2019

---

Kandidaatintyön tarkoituksena oli selvittää, miten syöte vaikuttaa erilaisten järjestämisalgoritmien suoritusajajaan.

Työssä vertailtiin Quicksortia, LSD Radix sortia ja Shellsortia. Osassa testeistä algoritmien syötteet olivat valmiiksi järjestettyjä, kun taas toisissa testeissä alkiot olivat esimerkiksi uniikkeja. Näin pyrittiin saamaan aikaan muutoksia algoritmien suoritusajoissa.

Työn vertailun perusteella huomattiin, että järjestämisalgoritmeissa oli paljon eroja. Quicksortilla havaittiin selkeästi suurin suoritusajan vaihtelu, suoritusajan hidastuessa merkittävästi, kun syöte oli lähes järjestetty tai kun syöteen alkiot olivat uniikkeja. LSD Radix sortin havaittiin olevan selkeästi nopein uniikeilla alkioilla, kun taas Shellsort suoriutui jokaisesta testistä ilman suuria muutoksia suoritusajassa.

Avainsanat: Quicksort, Radix Sort, Shellsort, suoritusajaka.

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

# SISÄLLYSLUETTELO

1. JOHDANTO .....	1
2. JÄRJESTYSLUOKIT YLEISESTI .....	2
3. KÄYTETTÄVÄT ALGORITMIT .....	3
3.1 Quicksort.....	3
3.2 LSD Radix sort.....	5
3.3 Shellsort.....	7
4. ALGORITMIEN VERTAILU .....	9
5. YHTEENVETO.....	11
LÄHTEET .....	12

# LYHENTEET JA MERKINNÄT

LSD Least significant digit  
MSD Most significant digit

# 1. JOHDANTO

Järjestysalgoritmi järjestää sille annetun listan sovittuun järjestykseen; esimerkiksi aakkosjärjestykseen tai suuruusjärjestykseen. Esimerkiksi suurissa tietokannoissa datan säilyttäminen tai taulukointi valitussa järjestyksessä on tärkeää tiedon hakemisen nopeuttamiseksi. Tästä syystä erilaisia järjestysalgoritmeja on kehitetty paljon, ja niiden toimintaperiaatteiden vaihtelevuuden vuoksi myös niiden suoritusajat vaihtelevat keskenään. Suoritusajat vaihtelevat joskus myös algoritmille syötetyn listan sisällön vuoksi, ja tässä työssä tutkitaan algoritmin suoritusajan vaihtelua riippuen algoritmin saamasta syötteestä.

Työssä tutkitaan kolmea erilaista tunnettua ja kolmeen erilaiseen toimintatapaan perustuvaa järjestysalgoritmia. Algoritmeista on valittu tarkoituksella mahdollisimman erilaisia algoritmeja, jotta voidaan nähdä toimintatavan vaikutus syötteeseen reagoimiseen. Tarkoitus on selvittää, miten erilaiset syötteet vaikuttavat kunkin järjestysalgoritmin asymptoottiseen suoritusaikaan.

Työn alussa luvussa 2 kerrotaan yleisesti järjestysalgoritmeista ja selitetään asymptoottinen suoritusaika. Luvussa 3 esitellään valitut järjestysalgoritmit koodeineen. Luvussa myös kerrotaan valittujen järjestysalgoritmien yleisestä suoritusajasta ja siitä, kuinka niitä voitaisiin tehostaa. Luku 4 sisältää itse vertailun ja vertailun tuloksien käsittelyn. Luvussa 5 on lyhyt yhteenveto.

## 2. JÄRJESTYSALGORITMIT YLEISESTI

Järjestysalgoritmillla tarkoitetaan algoritmia, jonka tehtävänä on muokata saadun syötteen alkoiden järjestystä siten, että ne ovat järjestettynä halutun ominaisuuden mukaan, esimerkiksi aakkosjärjestyksessä. Järjestysalgoritmeista suurin osa perustuu vertailuun, eli järjestyksen saamiseksi alkioita verrataan keskenään, jotta varmistutaan siitä, että alkiot ovat oikeassa järjestyksessä suhteessa toisiinsa. On olemassa myös algoritmeja, jotka eivät vertaile alkioita toisiinsa, kuten Radix sort, jossa alkiot jaetaan eri ryhmiin niiden numeerisen arvon perusteella muista alkioista riippumatta. Vertailuun perustuvien algoritmien keskiarvoinen suoritusaika ei voi olla parempi kuin  $\Theta(n \log n)$ .

Järjestysalgoritmien suoritusaikaa mitataan suoritettavien operaatioiden määränä suhteessa alkoiden määrään  $n$ . Esimerkiksi algoritmin suorittamiseen voi mennä  $n^3 + n^2 + 100n + 100$  operaatiota. Asymptoottinen suoritusaika saadaan, kun suoritusaikasta poistetaan osat, joilla ei ole paljoa merkitystä kokonaissuoritusaikaan, kun järjestettävien alkoiden määrä kasvaa suureksi. Esimerkkinä olevan algoritmin suoritusaikan asymptoottinen suoritusaika on näin ollen  $n^3$ . Asymptoottista suoritusaikaa voidaan merkitä kolmella erilaisella notaatiolla. Iso-omega –notaatio ( $\Omega$ ) kertoo, kuinka monta operaatiota algoritmi vähintään suorittaa, eli kuinka tehokas algoritmi parhaimmillaan on. Theetanotaatio ( $\Theta$ ) taas kuvaa kuinka monta operaatiota algoritmi keskimäärin vaatii. Iso-O –notaatio ( $O$ ) kuvaa, kuinka monta operaatiota algoritmi enintään tarvitsee, eli kuinka hidas algoritmi huonoimmillaan on.

## 3. KÄYTETTÄVÄT ALGORITMIT

Työssä vertaillaan kolmea eri järjestysalgoritmia: Quicksort, Radix sort ja Shellsort. Quicksortista on vertailussa versio, jossa partition-funktiossa pivot-alkioksi valitaan funktiolle annetun välin viimeinen alkio. Radix sortista käytetään LSD-versiota. Shell sortista käytetään Shellin alkuperäistä versiota välisekvenssistä. Algoritmien valinnassa on pyritty valitsemaan suosittuja mutta myös toimintaperiaatteeltaan mahdollisimman erilaisia algoritmeja.

### 3.1 Quicksort

Quicksort on brittiläisen Tony Hoaren kehittämä järjestysalgoritmi, joka julkaistiin alun perin Communications of the ACM -lehdessä vuonna 1961 [1]. Algoritmi on valittu työhön sen laajan suosion perusteella.

Quicksort valitsee pivot-alkion, jota suuremmat alkio siirretään pivot-alkion oikealle puolelle ja pienemmät vasemmalle puolelle. Tämän jälkeen quicksortia kutsutaan rekursiivisesti molemmille puolille jääneille osille, kunnes lista on oikeassa järjestyksessä. Paras ja keskimääräinen suoritus aika quicksortille on  $\Theta(n \log n)$ . Quicksortin huonoin suoritus aika  $O(n^2)$  toteutuu, kun jokaisella partition-funktion kutsulla kaikki alkio jäävät samalle puolelle pivot-alkiosta nähden. Tämän takia algoritmin huonoimman suoritusajan todennäköisyys riippuu paljon siitä, miten pivot-alkion valinta tehdään. Huono suoritus aika toteutuu myös silloin, kun syöte sisältää paljon samanarvoisia alkioita. Quicksort on epävakaa algoritmi, eli se ei välttämättä pidä samanarvoisten alkioiden keskinäistä järjestystä samana kuin alkuperäisessä listassa.

Ohjelmassa 1 on toteutettu Quicksort C#-kielellä. Quicksort saa järjestettävän taulukon ja järjestettävän välin päätepisteet. Siirron jälkeen quicksort kutsuu itseään rekursiivisesti järjestääkseen pivot-alkion vasemmalle puolelle jääneet alkio. Tässä toteutuksessa oikealle puolelle jääneet alkio järjestetään ilman rekursiota, mikä säästää muistia mutta ei vaikuta algoritmin suoritus aikaan.

Samassa ohjelmassa oleva partition-funktio käy läpi sille annetun välin taulukosta. Funktio valitsee pivot-alkioksi välin viimeisen alkion ja siirtää kaikki pivot-alkiota pienemmät alkio vasemmalle puolelleen ja palauttaa pivot-alkion kohdan taulukossa.

```

void Quicksort(int[] arr, int low, int high)
2 {
    while (low < high)
4 {
        var pivot = Partition(arr, low, high);
6 // järjestetään pivotin molemmille puolille jäävät listat
        Quicksort(arr, low, pivot - 1);
8 low = pivot + 1;
    }
10 }

12 int Partition(int[] arr, int low, int high)
    {
14     var i = low;
        var pivot = arr[high];
16
        // Käydään kaikki low-high välin alkiot läpi
18     for (var j = low; j < high; j++)
        {
20         if (arr[j] < pivot)
            {
22             // Siirretään pivot alkiota pienemmät alkiot
                // pivotin vasemmalle puolelle
24             Swap(ref arr[i], ref arr[j]);
                i++;
26         }
        }
28
        Swap(ref arr[i], ref arr[high]);
30
        return i;
32 }

34 void Swap(ref int i, ref int j)
    {
36     int temp = i;
        i = j;
38     j = temp;
    }

```

**Ohjelma 1.** *Quicksortin ja partition-funktion C#-toteutus.*

Quicksortia on mahdollista tehostaa useilla eri tavoilla. Parasta suoritusaikaa ei voida parantaa, mutta huonoimman suoritusajan todennäköisyyttä voidaan laskea esimerkiksi valitsemalla pivot kolmen eri alkion mediaanina, jolloin todennäköisyys sille, että kaikki alkiot jäävät pivot-alkion oikealle puolelle pienenee. Toistuvien elementtien aiheuttamaa hidasta suoritusta voidaan estää palauttamalla yhden pivot-alkion sijaan väli, jossa alkiolla on sama arvo kuin pivot-alkiolla. Quicksort on hajota ja hallitse -algoritmi, joten sen rinnakaistaminen on yksinkertaista. Lisäksi quicksortin muistinkulutusta on mahdollista vähentää käyttämällä häntärekursiota normaalin rekursion sijaan.

## 3.2 LSD Radix sort

Radix sort järjestää syöteen numeeriset alkiot jakamalla ne ryhmiin luvuissa samalla kohdalla olevien numeroiden perusteella. Oikealta vasemmalle eli vähiten merkitseväästä kohti eniten merkitsevää etenevää versiota kutsutaan LSD Radix sortiksi ja päinvastaista MSD Radix sortiksi. Tämä työ käsittelee vain LSD versiota tästä algoritmista.

Alkioita ryhmiin jaettaessa säilytetään samanarvoisten lukujen alkuperäinen järjestys toisiinsa nähden, mikä tekee LSD Radix sortista vakaan algoritmin. Koska alkiot jaetaan ryhmiin toisistaan riippumatta, algoritmin ei tarvitse tehdä vertailuja alkioden välillä. Algoritmin huonoin mahdollinen suoritusaika on  $O(nw)$ , missä  $w$  kuvaa syöteen alkioden keskimääräistä pituutta.

Ohjelmassa 2 on toteutettu LSD Radix sort, jota ei ole optimoitu. Algoritmi selvittää suurimman syötteessä esiintyvän luvun, jotta saadaan suurimman alkion pituus. Tämän jälkeen algoritmi järjestää Counting sort -algoritmin avulla taulukon alkioden numeroiden perusteella aloittaen vähiten merkitseväästä numerosta.

Algoritmista saadaan tehokkaampi, kun syöte jaetaan pienempiin osajoukkoihin alkion numeroiden määrän perusteella ja suoritetaan LSD Radix sort jokaiselle osajoukolle erikseen. Toinen tapa tehostaa algoritmia on laskea esiintymismäärät jokaiselle alkion numerolle yhdellä syöteen läpikäynnillä. Esiintymismäärien laskeminen alkion eri numeroille voidaan myös rinnakkaistaa nopeuttaen algoritmia reaaliajassa, mutta tämä ei vaikuta algoritmin asympotoottisen suoritusaikaan.

```

    int GetMax(int[] arr, int n)
2   {
    int mx = arr[0];
4   for (var i = 1; i < n; i++)
        if (arr[i] > mx)
6       mx = arr[i];
    return mx;
8   }

10 void Countsort(int[] arr, int n, int exp)
    {
12     var output = new int[n];
        var count = new int[10];
14     int i;

16     // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
18         count[arr[i] / exp % 10]++;

20     // Change count[i] so that count[i] now contains actual
        // position of this digit in output[]
22     for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
24

        // Build the output array
26     for (i = n - 1; i >= 0; i--)
        {
28         output[count[arr[i] / exp % 10] - 1] = arr[i];
            count[arr[i] / exp % 10]--;
30     }

32     // Kopioidaan järjestetty taulukko alkuperäiseen taulukkoon
        for (i = 0; i < n; i++)
34         arr[i] = output[i];
    }
36

void Radixsort(int[] arr)
38 {
    int n = arr.Length;
40     // Find the maximum number to know number of digits
        int m = GetMax(arr, n);
42

        // Do counting sort for every digit. Note that instead
44     // of passing digit number, exp is passed. exp is 10^i
        // where i is current digit number
46     for (intr exp = 1; m / exp > 0; exp *= 10)
        Countsort(arr, n, exp);
    }
}

```

**Ohjelma 2.** LSD Radix sortin C#-toteutus. [2]

### 3.3 Shellsort

Shellsort on Donald Shellin kehittämä vertailuun perustuva järjestysalgoritmi, joka julkaistiin Communications of the ACM -lehdessä vuonna 1959. Algoritmi on valittu työhön sen yksinkertaisen toteutuksen takia.

Shellsortissa syöte järjestetään käymällä se useasti läpi Insertion sort -algoritmia muistuttavalla algoritmilla, jossa vertaillaan vain joka  $n$ :ttä alkia, jossa  $n$  on kullakin läpikäynnillä pienenevä luku. Esimerkiksi Shellin alkuperäisessä algoritmissa  $n = \frac{N}{2^k}$ , missä  $N$  on alkioden kokonaismäärä ja  $k$  läpikäyntikerran numero. Viimeisellä läpikäynnillä vertailtavien alkioden välinen pituus on 1, joten viimeinen läpikäynti vastaa normaalia Insertion sort -algoritmia. [3] Aiempien läpikäyntien takia syöte on jo lähes järjestetty, minkä vuoksi läpikäynti nopea.

```

void Shellsort(int[] arr)
2 {
    int n = arr.Length;
4
    for (int gap = n / 2; gap > 0; gap /= 2)
6 {
        // Do a gapped insertion Shellsort for this gap size.
8        // The first gap elements a[0..gap-1] are already in gapped
        // order. Keep adding one more element until the entire array
10       // is gap sorted
        for (int i = gap; i < n; i += 1)
12 {
            // add a[i] to the elements that have
14            // been gap sorted save a[i] in temp and
            // make a hole at position i
16            int temp = arr[i];

18            // shift earlier gap-sorted elements up until
            // the correct location for a[i] is found
20            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
22 {
                arr[j] = arr[j - gap];
24            }

26            // put temp (the original a[i]) in its correct location
            arr[j] = temp;
28        }
    }
}

```

**Ohjelma 3.** Shellsortin C#-toteutus Shellin alkuperäisellä välisekvenssillä. [5]

Shellsortin suoritus aika riippuu alkioden väleistä eri läpikäynneillä, ja algoritmin huonointa suoritus aikaa voidaan siis parantaa käyttämällä tehokkaampaa välisekvenssiä.

Alkuperäisessä Shellin välisekvenssissä huonoin suoritus aika on  $\Theta(n \log n)$ . Tehokkaammilla välisekvensseillä, kuten Sedgewickin 1986 ehdottamalla välisekvenssillä, huonoin suoritus aika on  $O(N^{4/3})$  [4].

## 4. ALGORITMIEN VERTAILU

Työssä vertailu on toteutettu C#-kielellä hyödyntäen ohjelmien suoritustehon mittaamista varten kehitettyä Benchmark.NET kirjastoa. Vertailussa on yritetty testata algoritmien suoritusnopeutta erilaisilla syötteillä. Kaikki järjestysalgoritmit toteutettiin hyväksymään syötteeksi vain kokonaislukuja ja tästä syystä kaikki syötteet sisältävät vain kokonaislukuja. Jokaisessa testissä syötteen pituus on 100 000 lukua ja kaikki luvut väliltä 0–10 000 ellei toisin mainittu.

Vertailussa testattiin algoritmeja seuraavilla syötteillä:

- Täysin satunnaisesti valittu syöte.
- Valmiiksi järjestetty syöte.
- Valmiiksi järjestetty syöte, joka on päinvastaisessa järjestyksessä haluttuun järjestykseen nähden.
- Syöte, josta ensimmäinen puolisko järjestetty ja toinen puolisko satunnaisessa järjestyksessä.
- Syöte, jossa on paljon samoja lukuja, tässä tapauksessa kaikki luvut olivat väliltä 0-5.
- Vain isoja lukuja sisältävä syöte, tässä tapauksessa kaikki luvut olivat väliltä 100 000 000–200 000 000.
- Syöte, jossa kaikki luvut olivat uniikkeja, tässä tapauksessa luvut 1–100 000 jotka on sekoitettu satunnaiseen järjestykseen.

Jokaista testiä varten luotiin uusi satunnainen syöte, jota muokattiin kutakin testiä varten tarpeen mukaan. Jokaista testiä ajettiin 1 000 kertaa ja taulukkoon 1 on listattu testien tuloksista saadut keskiarvot jokaiselle eri algoritmille.

**Taulukko 1.** Järjestysalgoritmien testien tulokset.

Testi	Quicksort (ms)	LSD Radix sort (ms)	Shellsort (ms)
Satunnainen	5,7	5,2	13,5
50% järjestetty	5,9	6,6	11,7
Järjestetty	<b>904,3</b>	5,2	3,8
Päinvastainen järjestys	<b>1056,0</b>	5,2	4,8
Paljon samoja lukuja	701,1	1,3	5,6
Vain isoja lukuja	6,3	12,0	13,5
Uniikit alkiot	<b>2404,5</b>	<b>0,08</b>	3,8

Testeistä havaitaan, että Quicksortilla on selkeästi suurin vaihtelu suoritusajassa syötteeseen nähden. Quicksort on nopea, kun sille annettu syöte on selkeästi satunnaisessa järjestyksessä, mutta erittäin hidas kun syöte on järjestettynä tai sisältää paljon uniikkeja alkioita. LSD Radix sortin suoritus aika ei vaihtelee merkittävästi syötteen järjestyksen mukaan, mutta hidastuu jonkin verran, kun syötteen alkiot ovat pitkiä. LSD Radix sort on myös erittäin nopea järjestysalgoritmi, kun syötteen alkiot ovat uniikkeja. Shellsortin suoritus aika vaihtelee vähän verrattuna muihin testattuihin järjestysalgoritmeihin, hitaimman suoritusajan ollessa vain noin kolme kertaa suurempi kuin nopein suoritus aika. Shellsortilla on testatuista järjestysalgoritmeista hitain suoritus aika täysin satunnaisella syötteellä, mutta se suoriutuu hyvin järjestetyistä tai paljon samoja lukuja sisältävistä syötteistä.

## 5. YHTEENVETO

Työssä vertailtiin kolmen erilaisen järjestysalgoritmin suorituskykyä erilaisilla syötteillä. Valitut algoritmit olivat quicksort, LSD Radix sort ja shellsort. Algoritmien toteutus pidettiin mahdollisimman yksinkertaisena, eikä niitä pyritty optimoimaan. Työssä koetettiin havaita eroavaisuuksia suoritusajassa, kun niille annetaan erilaisia syötteitä.

Vertailussa havaittiin, että quicksortin suoritusajaksi hidastuu merkittävästi syötteillä, jotka ovat jo suurimmaksi osaksi järjestettyjä tai sisältävät paljon samoja alkioita. Tämä johtuu erityisesti valitusta yksinkertaisesta partition-funktiosta, jota muuttamalla voitaisiin saada parempia tuloksia. LSD Radix sort pärjasi hyvin kaikilla syötteillä, mutta sen huomattiin olevan selkeästi nopein syötteellä, jossa on paljon uniikkeja alkioita. Näin ollen sen käyttö voi olla perusteltua, jos tiedetään että järjestettävänä on esimerkiksi käyttäjien uniikkeja tunnisteita. Shellsortin suoritusajaksi vaihteli vähiten tutkituista algoritmeista, riippuen lähinnä siitä, kuinka lähellä valmiiksi järjestettyä syöte oli.

Tulosten perusteella järjestysalgoritmia valittaessa on tärkeää miettiä, millaista järjestettävä data on, jotta järjestäminen on nopeaa ja suoritusajaksi käyttäytyy halutusti. Yksittäistä parasta algoritmia kaikkiin tilanteisiin ei ole, vaan paras algoritmi kuhunkin käyttötarkoitukseen riippuu paljon syötteestä.

# LÄHTEET

- [1] Hoare C. Algorithm 64: Quicksort. Communications of the ACM, Vol 4, 1961.
- [2] GeeksforGeeks, Radix Sort. Saatavissa (Viitattu 01.08.2019):  
<https://www.geeksforgeeks.org/radix-sort/>
- [3] Shell DL. A high-speed sorting procedure. Communications of the ACM, Vol 2, 1959.
- [4] Sedgewick R. A new upper bound for Shellsort. Journal of Algorithms, Vol 7, 1986. 159–173.
- [5] GeeksforGeeks, ShellSort. Saatavissa (Viitattu 01.08.2019):  
<https://www.geeksforgeeks.org/shellsort/>