# Tietorakenteiden harjoitustyö

## Toteutusdokumentti

Nikke Kostiainen, 2014

# 1 Algoritmien vaativuusanalyysi

Käydään seuraavaksi läpi kaikki harjoitustyössä toteutetut algoritmit ja arvioidaan niiden toteuttamisessa saavutetut tila- sekä aikavaativuudet.

## *1.1 Introsort*

Introsortin tavoiteaikavaativuus on O(n log n) ja tavoitetilavaativuus on O(log n). Tutkitaan seuraavaksi vaativuuksien toteutumista. Tutkitaan aluksi tapauksia, jossa järjestettävän taulukon koko on alle 16 alkiota. Tällöin oletuksena järjestetään taulukko suoraan käyttäen lisäysjärjestämistä, joka käy pahimmassa tapauksessa taulukon läpi n kertaa ja siirtää jokaisen alkion n kertaa oikealle paikalleen. Tällöin lopulliseksi aikavaativuudeksi tulee O(n^2). Koska lisäysjärjestäminen käyttää vain vakiomäärän apumuuttujia, eikä toimi rekursion avulla on lopullinen tilavaativuus O(1).

```
intro_InsertionSort(int[] arrayToSort, int first, int last) {
    int j, temp;
    for (int i = first; i < last; i++) {
      j = i;
      temp = arrayToSort[i];
      while (j != first && temp < arrayToSort[j - 1]) {
        arrayToSort[j] = arrayToSort[j - 1];
        j--;
      }
      arrayToSort[j] = temp;
    }
  }
```

Kuitenkin, koska kyseessä on toimenpide, joka suoritetaan vain taulukon ollessa todella pieni, lisäysjärjestämisen käyttö ei ole haitallista koko algoritmin varsinaiselle aikavaativuudelle.

Siinä tapauksessa, että järjestettävä taulukko on suurempi kuin 16 alkiota, introsort jäjestää taulukon niin, että aluksi taulukkoa järjestetään quicksortilla niin, että pivotiksi valitaan taulukon kolmen alkion mediaani. Siinä tapauksessa, että quicksortin rekursiosyvyys kasvaa määriteltyä suuremmaksi, siirtyy algoritmi käyttämään quicksortin sijasta heapsortia lopputaulukon järjestämiseen. Näin introsort välttää quicksortin mahdollistavat pahimman tapauksen taulukot. Pikajärjestämisosan keskimääräinen aikavaativuus on O(n log n) sillloinkin, kun partition jakaa alkioita huonohkosti. Viitataan tässä Tietorakenteet ja algoritmit -kurssin kurssimateriaaliin, jossa tällainen todistus esitetään:

Oletetaan, että k -alkioinen taulukko jakautuisi huonoimmillaan siten ,että kooltaan pienempi osa on vain k/10 -kokoinen ja suurempi 9k/10. Saadaan kaava

$$T_u(n) \leq cn(\log_{10/9} n + 1) = cn(\frac{\log_2 n}{\log_2 10/9} + 1) \leq cn(7 \times \log_2 n + 1) = 7 \times cn(\log_2 n \ + \frac{1}{7}) = \mathcal{O}(n \log n)$$

Näin voidaan olettaa, että tapauksessa, jossa järjestäminen tapahtuu puhtaasti pikajärjestämisellä, aikavaativuus on O(n log n). Koska quicksort käyttää hyväkseen rekursiopuuta, jäisi tilavaativuuden suuruudeksi pahimmassa mahdollisessa tapauksessa O(n). Kuitenkin, koska introsort ei koskaan anna rekursiopuun kasva yli määritellyn rajan, jää lopullisen algoritmin tilavaativuus kuitenkin pienemmäksi suuremmilla taulukoilla.

Tarkastellaan vielä tapausta, jossa quicksortin rekursiopuu ylittyy ja päädymme järjestämään loput taulukosta käyttäen heapsorttia. Heapsortin aikavaativuudeksi on tavoiteltu O(n logn) ja tilavaativuudeksi O(1). Heapsort etenee niin, että (jo hieman järjestyksessä oleva) taulukko rakennetaan aluksi metodilla buildheap. Buildheap kutsuu metodia heapify n/2 kertaa n-alkioiselle taulukolle. Koska heapifyn suoritusaika riippuu keon korkeudesta, ja keon tiedetään olevan lähes täydellinen binääripuu on keon korkeus siis log n n-alkioisessa taulukossa.

```
private void intro_buildHeap(int[] array, int last, int first) {
      for (int i = last / 2; i >= 1; i = i - 1) {
         heapify(array, i, last, first);
      }
   }
private void heapify(int[] array, int index, int last, int first) {
      int d = array[first + index - 1];
      int child;
      while (index <= last / 2) {
         child = 2 * index;
         if (child < last && array[first + child - 1] < array[first + child]) {
            child++;
         }
         if (d >= array[first + child - 1]) {
            break;
         }
         array[first + index - 1] = array[first + child - 1];
         index = child;
      }
      array[first + index - 1] = d;
   }
```

Täten metodin heapify aikavaativuus on O(log n) ja buildheapin kutsuessa metodia n kertaa tulee buildheapin aikavaativuudeksi (n log n). Algoritmin lopuksi kutsutaan algoritmi heapify n – 1 kertaa, joten lopullinen algoritmin aikavaativuus on noin O(2* n * log(n)) eli O(n log n). Koska heapsort toteutetaan käyttäen implisiittistä kekoa, eikä algoritmi käytä rekursiota ja ainoastaan metodi heapify käyttää vakiomäärää apumuuttujia, on heapsortin lopullinen tilavaativuus O(1).

```
 public void intro_HeapSort(int[] array, int first, int last) {
      int gap = last - first;
      //build heap
      intro_buildHeap(array, gap, first);
      //sort the array
      for (int i = gap; i > 1; i = i - 1) {
         GlobalMethods.exchange(array, first, first + i - 1);
         heapify(array, 1, i - 1, first);

      }
}
```

Lopuksi mietitään koko introsortin pahimman tapauksen aikavaativuutta. Koska pahimmassa tapauksessa quicksortin rekursiosyvyys kasvaa noin O(logn):n suuruiseksi, ja quicksortin ja heapsortin aikavaativuudeksi todettin O(n log(n)), muodostuu koko introsortin lopulliseksi vaativuudeksi:

**Aika: O( n log(n) )  Tila: O(log (n))**

Nikke Kostiainen, 2014

## *1.2 Mergesort*

Mergesortin aikavaativuudeksi tavoitellaan syötteestä riippumattomasti O(n logn). Mergesort toimii niin, että joka kierroksella taulukko jaetaan kahtia ja rekursiivisesti suoritetaan kummallekin puoliskolle oma mergesort, jonka jälkeen puoliskot yhdistetään. Taulukon jako tapahtuu triviaalisti ajassa O(1). Tutkitaan siis joka kierroksella tapahtuvaa yhdistämisoperaatiota.

```
while (leftArrayIndex < left.length && rightArrayIndex < right.length) {
        if (left[leftArrayIndex] < right[rightArrayIndex]) {
          newArray[newArrayIndex] = left[leftArrayIndex];
          newArrayIndex++;
          leftArrayIndex++;
        } else {
          newArray[newArrayIndex] = right[rightArrayIndex];
          newArrayIndex++;
          rightArrayIndex++;
        }
}
while (leftArrayIndex < left.length) {
        newArray[newArrayIndex] = left[leftArrayIndex];
        leftArrayIndex++;
        newArrayIndex++;
      }
 while (rightArrayIndex < right.length) {
        newArray[newArrayIndex] = right[rightArrayIndex];
        rightArrayIndex++;
        newArrayIndex++;
      }
 return newArray;
  }
```

Merge käy taulukon molemmat puoliskot läpi lineaarisesti ja lisää numerot järjestyksessä uuteen taulukkoon. Koska taulukko jakaa n -alkioisen taulukon kahtia, ja käy lopuksi tämän taulukon kaikki alkiot kertaalleen läpi, tulee yhden kierroksen aikavaativuudeksi O(n). Tämän jälkeen laskemme mukaan rekursion kutsujen määrän. Koska joka kutsulla taulukko jaetaan kahdeksi, joudutaan rekursiokutsu suorittamaan taulukon kokoon nähden log(n) kertaa. Näin ollen koko mergesortin aikavaativuudeksi saadaan O(n log(n)). Koska mergesort käyttää rekursiopuuta, jonka syvyys on korkeintaan O(log n) ja aputilaa luodaan O(n):n verran, jää lopulliseksi tilavaativuudeksi myös O(n).

Algoritmin lopullinen vaativuus on siis:

**Aika: O( n log(n) )  Tila: O(n)**

## *1.3 Smoothsort*

Smoothsort järjestää taulukon keoista muodostuvan metsän avulla. Aluksi järjestettävä taulukko muutetaan useamman keon metsäksi operaatiolla formLeonardoHeap. Tämä metodi käy läpi jokaisen taulukon alkion ja korjaa sen alueen kekoehdon. Kekoehdon korjaus vie pahimmillaan O(log n) aikaa ja koska operaatio suoritetaan jokaiselle taulukon alkiolle kestää koko operaatio O(n log n).

```
private int formLeonardoHeap(int[] arrayToSort, int[] orders, int trees) {
    for (int i = 0; i < arrayToSort.length; i++) {
      if (trees > 1 && orders[trees - 2] == orders[trees - 1] + 1) {
        trees--;
        orders[trees - 1]++;
      } else if (trees > 0 && orders[trees - 1] == 1) {
        orders[trees++] = 0;
      } else {
        orders[trees++] = 1;
      }
      findAndSift(arrayToSort, i, trees - 1, orders);
    }
    return trees;
  }
```

Tämän jälkeen suoritetaan lopullinen järjestäminen, jossa otetaan kekorakenteen suurin alkio, siirretään se taulukon seuraavaksi alkioksi ja pienennetään rakenteen kokoa yhdellä. Tämän 'dequeue' -operaation aikavaativuus tulee samalla tavalla kekoehdon korjaamisesta kuten edellisesäkin kohdassa kestäen O(log n) per asetettava alkio. Algoritmin ei kuitenkaan tarvitse suorittaa kekoehdon korjausta tilanteessa, jossa kekoehto säilyy dequeuen yhteydessä. Siispä tilanteessal, jossa taulukko onkin jo järjestyksessä voidaan dequeue -operaatio suorittaa vakioajassa. Näinpä parhaimmassa tapauksessa päästään aikavaativuuteen O(n) ja toisaalta pahimmassakin tapauksessa vaativuudeksi jää O(n log n).

```
private void breakDown(int[] arrayToSort, int[] orders, int trees) {
    for (int i = arrayToSort.length - 1; i > 0; i--) {
      if (orders[trees - 1] <= 1) {
        trees--;
      } else {
        int rightIndex = i - 1;
        int leftIndex = rightIndex - leonardoNumbers[orders[trees - 1] - 2];

        trees++;
        orders[trees - 2]--;
        orders[trees - 1] = orders[trees - 2] - 1;

        findAndSift(arrayToSort, leftIndex, trees - 2, orders);
        findAndSift(arrayToSort, rightIndex, trees - 1, orders);
      }
    }
```

```
private void findAndSift(int[] arrayToSort, int index, int tree, int[] orders) {
      int value = arrayToSort[index];
      while (tree > 0) {
         int pivot = index - leonardoNumbers[orders[tree]];
         if (arrayToSort[pivot] <= value) {
            break;
         } else if (orders[tree] > 1) {
            int rightIndex = index - 1;
            int leftIndex = rightIndex - leonardoNumbers[orders[tree] - 2];
            if (arrayToSort[pivot] <= arrayToSort[leftIndex] || arrayToSort[pivot] <= arrayToSort[rightIndex]) {
               break;
            }

         }
         arrayToSort[index] = arrayToSort[pivot];
         index = pivot;
         tree--;
      }
      arrayToSort[index] = value;
      siftDown(arrayToSort, index, orders[tree]);
   }

public void siftDown(int[] arrayToSort, int index, int order) {
      int value = arrayToSort[index];
      while (order > 1) {
         int rightIndex = index - 1;
         int leftIndex = rightIndex - leonardoNumbers[order - 2];
         if (value >= arrayToSort[leftIndex] && value >= arrayToSort[rightIndex]) {
            break;
         } else if (arrayToSort[leftIndex] <= arrayToSort[rightIndex]) {
            arrayToSort[index] = arrayToSort[rightIndex];
            index = rightIndex;
            order = order - 2;
         } else {
            arrayToSort[index] = arrayToSort[leftIndex];
            index = leftIndex;
            order--;
         }
      }
      arrayToSort[index] = value;
   }
```

Pahimmassa tapauksessa smoothsortin aikavaativuus on siis **O(n log n)**. Koska aputietorakenteena käytetään taulukkoa, jonka pituus perustuu tarvittavien kekojen määrään ja määrä voidaan laskea logaritmisesti, on tilavaativuus implementaatiossamme **O(log n)**. Implementaatio on myös mahdollista suorittaa käyttäen puhtaasti implisiittistä tietorakennetta, mutta sen implementoiminen on huomattavan monimutkaista.

Nikke Kostiainen, 2014

## 2 Algoritmien nopea vertailu

Empiirisen ja analyysin perusteella voidaan todeta, että algoritmeillä on omat erityisominaisuutensa ja jokainen algoritmi voi toimia muita paremmin tietyissä erikoistilanteissa.

**Introsort** toimii algoritmeistä nopeiten satunnaisilla syötteillä ja on siis ideaalinen valinta siinä tilanteessa, kun järjestettävästä syötteestä ei tiedetä paljoa ja se voi olla hyvinkin satunnainen mutta tarvitaan ajallisesti mahdollisimman tehokas järjestäjä.

**Mergesort** taas toimii algoritmeistä kaikkein vakaimmin. Algoritmi järjestää minkä tahansa syötteen samassa ajassa, joten algoritmi on erittäin hyödyllinen silloin kun on erittäin kriittistä ettei millään syötteellä ole suurta vaikutusta algoritmin suoritusaikaan. Mergesort on myös vertailtavista algoritmeistä ainoa **vakaa** algoritmi, joten tilanteessa, jossa vakaus on kriittistä on mergesort luonnollinen valinta.

**SmoothSort** toimii ehdottomasti parhaiten kun kyse on jo valmiiksi järjestetyistä tai melkein järjestetyistä syötteistä, lähennellen melkein lineaarista järjestysaikaa. SmoothSort on siis hyvä valinta kun tiedetään että suuri osa syötteistä on jo lähes järjestyksessä ja halutaan mahdollisimman tehokas järjestysalgoritmini vertailu

## 3 Puutteet ja parannukset

Smoothsortin implementaatiossa on parantamisen varaa. Tilavaativuus on mahdollista nipistää vakiolliseksi. Myös mergesortin voisi muutta niin, että se vain implisiittisesti jakaa taulukon, vähentäen näin tilavaativuutta. Ohjelmaan olisi myös voitu implementoida jonkinnäköinen visuaalinen kuvaus järjestämisestä, ja raporttiin olisi ollut kiva tuoda hieman lisää visuaalista dataa.

## 4 Lähteet


Tietorakenteet ja algoritmit -kurssimateriaali

http://en.wikipedia.org/wiki/Sorting_algorithm

http://en.wikipedia.org/wiki/Merge_sort

http://en.wikipedia.org/wiki/Smoothsort

http://en.wikipedia.org/wiki/Introsort

http://www.keithschwarz.com/smoothsort/



Nikke Kostiainen, 2014