Keko järjestäminen - Heap Sort

Määrittely dokumentissa annan kekojärjestämiselle tavoite aikavaativuuden O(n log n) ja toteutuksessa tähän ollaan päästy sillä:

Heapify:

- Heapify-operaation suoritusaika riippuu puun korkeudesta ja koska keko on lähes täydellinen binääripuu on keon korkeus O(log n) täten n alkiota sisältävälle puulle heapify pahimman tapauksen aikavaativuus on O(log n)

```
Heapify(A, index)
if (right <= heapSize)
        if(A[left] > A[right])
                largest = left
        else
                largest = right
        if(A[index] < A[largest])
                swap(A, index, largest)
                heapify(A, largest)
```

BuildHeap

- BuildHeap-operaation rungossa kutsutaan heapify n/2 kertaa keolle jossa on n alkiota, täten BuildHeap käyttää aikaa korkeintaan O(n log n)

Lopuksi sortin aikana kutsutaan vielä heapify-operaatiota n - 1 kertaa, joten kokonaisuus aikavaativuudeksi saadaan **O(n log n)**

```
Sort(A)
        for i = heapSize down to 2
                swap(1, i);
                heapSize - 1;
                heapify(A, 1)
```

Tilavaativuus kekojärjestämisellä on Heapify-operaation rekursio kutsujen takio **O(log n)**

Kupla järjestäminen - Bubble Sort

Määrittely dokumentissa annan kupla järjestämiselle aikavaativuuden O(n^2), ja toteutetussa koodissa aikavaativuus on O(n^2) sillä:

- For-loop sisältää valinnan ja vakioajan vieviä sijoitusoperaatioita. For-loopin runko on siis vakioaikainen O(1)
- For-loopin runko sen sijaan suoritetaan pahimmillaan n - 1 kertaa, sillä for-loop suoritetaan niin kauan että ollaan listan läpi menty kertaalleen ilman että listan sisällä on tehty swappauksia.
- Eli runko suoritetaan pahimmillaan 1 + 2 + 3 + … + (n - 1) täten aikavaativuus on **O(n^2)**

Tilavaativuus järjestämiselle on **O(1)** eli vakio, sillä järjestämisen aikana käytetään vain vakioaikaisia apumuuttujia.

Pseudokoodina kuplajärjestäminen

```
BubbleSort(A)
while not sorted
    sorted = true
    for i = 1 up to A.length - 1
        if(A[i] > A[i + 1])
            swap(A[i], A[i + 1]
            sorted = false
```

Lomitus järjestäminen - MergeSort

Määrittely dokumentissa annan kupla järjestämiselle aikavaativuuden O(n log n), ja toteutuksessa tähän on päästy sillä:

Järjestämisalgoritmin alussa luodaan yksi aputaulukko jonka koko on n, missä n = järjestettävän listan kokonaispituus.

Merge-operaation vaativuus:

Merge-operaatiossa ei luoda uutta apumuuttujaa vaan käytetään lomitukseen alussa luotua aputaulukkoa hyväksi ja lomitetaan aputaulukon vasen - oikea välillä olevia alkioita. Lomitusjärjestämisessä suurimmassa osassa on merge-operaatio:

- Mergen alussa sijoitetaan taulukosta A taulukkoon temp alkiot vasen ja oikea väliltä

  ```
  for i = left up to right
      temp[i] = A[i]
  ```

  Tämän operaation aikavaativuus on täten O(k), missä k = lomitettavan taulukonosan pituus
- Tämän jälkeen sijoitetaan alkiot temp taulukosta takaisin A taulukkoon valittaen aina temp[left] ja temp[middle] väliltä pienempi arvo. Tässä middle kuvastaa oikean lohkon valinnan sijaintia.

  ```
  while left <= leftEnd and middle <= right
      if(temp[left] <= temp[middle])
          A[i] = temp[left]
          kasvata i ja left yhdellä
      else
          A[i] = temp[middle]
          kasvata i ja middle yhdellä
  ```

  Koska on mahdollista että lohkot eivät mene tasan lopussa vielä asetetaan loput alkiot väliltä left - leftEnd takaisin taulukkoon A

```
while left <= leftEnd
    A[i] = temp[left]
    kasvata i ja left yhdellä
```

Näiden kahden while loopin läpi menemiseen käytetään jälleen aikaa O(k), missä k = lomitettavan taulukonosan pituus. Täten koko merge-operaatioon käytetään yhteensä aikaa **O(k).**

Lopullisen lomitusjärjestämisen aikavaativuuden saamiseen täytyy ottaa huomioon rekursio kutsujen määrä. Jokainen sort kutsu puolittaa syötteensä koon, täten kun syöte on aluksi n. Niin kun puolitus on tehty $\log_2$ n-kertaa on syötteen koko enään 1, eli kun mukaan luetaan alin taso, jolla ei enään rekursio kutsua tehdä on rekursiotasoja $\log_2 n + 1$ kappaletta.

```
int middle = (left + right) / 2
sort(A, left, middle)
sort(A, middle + 1, right)
```

Tästä nähdään että lomitusjärjestämisen kokonais aikavaativuus on **O(n log n)**

Lomitusjärjestämisen tilavaativuus:

- Koska järjestämisalgoritmissa käytän yhtä apumuuttujaa jonka koko on n, voi todeta että algoritmin tilavaativuus on **O(n).**