

Tietorakenteiden harjoitustyö

Toteutusdokumentti

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

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

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

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 kasvaa 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 $O(n \log n)$ 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 $O(\log n)$:n suuruiseksi, ja quicksortin ja heapsortin aikavaativuudeksi todettiin $O(n \log(n))$, muodostuu koko introsortin lopulliseksi vaativuudeksi:

Aika: $O(n \log(n))$) Tila: $O(\log(n))$

1.2 Mergesort

Mergesortin aikavaativuudeksi tavoitellaan syötteestä riippumattomasti $O(n \log n)$. 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

In progress..