Rapport - Obligatorisk innlevering 3

Algoritmer beskrevet i denne rapporten:

- Selection-sort
- Heap-sort
- Quick-sort

Resultater med value-limit (VL) vil si at alle elementer i arrayet har verdi mellom 0-99. Da vil det forekomme gjentatte verdier i arrayet. Random vil si at elementene i arrayet ikke følger noe spesielt mønster før sortering. Sortert betyr at alle elementene er sortert i riktig rekkefølge før sortering. Reversert vil si at elementene i arrayet er sortert i fallende rekkefølge før sortering.

Dersom en sortering bruker mer enn 5 minutter (300 sekunder) på å fullføre, vil det stå ' - ' i tabellen, hvis ikke er tid spesifisert i antall sekunder.

Selection-sort

```
public int[] selectionSort(int[] a) {

        //finner minste verdi ved å gå gjennom hvert element i arrayet
        for (int index = 0; index < a.length; index++) {

            int min = index;
            for (int j = index+1; j < a.length; j++) {
                if (a[j] < a[min]) {
                    min = j;
                }
            }

            //bytter plass mellom min og første element
            int temp = a[min];
            a[min] = a[index];
            a[index] = temp;
        }
        return a;
    }
```

Implementasjonen av denne algoritmen er in-place.

Mønster:

Selection-sort har samme type mønster for alle typer input. Algoritmen finner minste verdi ved å iterere gjennom hele arrayet, for så å bytte plass mellom det minste elementet den finner og første element. Lengden på arrayet som itereres vil minske med en plass for hver iterasjon. Problemet med denne algoritmen er at selv om arrayet i utgangspunktet allerede er sortert vil den uansett ha lik kjøretid. Alle posisjoner må sjekkes opp mot resten av arrayet før algoritmen er ferdig.

Kjøretid selection-sort:

	Random VL	Random	Sortert VL	Sortert	Rev. VL	Rev.
1 000	0.007	0.002	0.003	0.001	0.001	0.0
5 000	0.013	0.004	0.008	0.006	0.006	0.009
10 000	0.027	0.033	0.035	0.025	0.033	0.041
50 000	0.594	0.650	0.680	0.782	0.731	0.961
100 000	2.646	2.810	2.684	2.771	2.631	3.895
1 000 000	-	218.086	219.779	-	219.914	-
10 000 000	-	-	-	-	-	-
100 000 000	-	-	-	-	-	-

Heap-sort

```java
public int[] heapSort(int[] a) {
        int stopIndex = a.length;

        //oppretter maxheap. Begynner midt i arrayet, nodene etter er løvnoder
        for (int i = a.length/2; i >= 0; i--) {
            downHeap(a, i, stopIndex);
        }

        //bytter plass på siste og første(max-noden) element, og downheaper
        for (int i= a.length-1; i >= 0; i--) {
            //arbeidsområdet må minske med 1 hver iterasjon så ikke man havner
 i evig loop
            stopIndex--;

            int siste = a[i];
            a[i] = a[0];
            a[0] = siste;

            downHeap(a, 0, stopIndex);
        }
        return a;
    }
```

Implementasjonen av denne algoritmen er in-place. For å få til dette trengs det en rekursiv hjelpemetode downHeap(`int[] a, int i, int stopIndex`), som kan brukes i to ulike tilfeller. Først vil den bygge opp et maxHeap der man opererer på hele arrayet. Deretter kan samme metode brukes til å bytte plass på rotnoden/største element og siste node, dette for å sortere arrayet. Her vil man operere på en mindre del av arrayet for hver iterasjon.

Koden til downHeap() følger med i javafilen «Oblig3.java» vedlagt innleveringen.

Mønster:

Første del av heap-sort innebærer å opprette et max-heap. Denne prosessen vil gå mye raskere når arrayet allerede er sortert i fallende rekkefølge. Da vil alltid verdien til en forelder være større enn begge sine barn, og man har allerede et max-heap til å begynne med. Algoritmen kan da hoppe over prosessen med å bytte barn og forelder, og slipper da nye rekursjonskall på downHeap(). Mønsteret for fallende rekkefølge i arrayet forblir den

samme. For sortert vil elementene mot slutten av liste havne mot starten, og motsatt. Elementene rundt midten forblir det samme. Ved tilfeldig oppbygning vil mønsteret variere.

Når det gjelder andre del av heap-sort, bytte mellom rotnoden (max-noden) og siste node, vil mønsteret være så godt som det samme for alle typer input. Man begynner med et max-heap i alle tilfeller, og ender opp med et sortert array. For sortert vil gjennomføringen være optimal, for alle elementer ligger i rekkefølge. For fallende og tilfeldig sortering vil det være noen tall som ikke ligger i rekkefølge, men majoriteten vil likevel gjøre det, og utslagene er derfor relativt små.

Kjøretid heap-sort:

	Random VL	Random	Sortert VL	Sortert	Rev. VL	Rev.
1 000	0.0	0.0	0.0	0.0	0.0	0.0
5 000	0.0	0.001	0.001	0.0	0.001	0.001
10 000	0.001	0.002	0.001	0.001	0.0	0.001
50 000	0.002	0.005	0.005	0.004	0.006	0.002
100 000	0.010	0.012	0.004	0.005	0.005	0.007
1 000 000	0.103	0.154	0.085	0.085	0.079	0.094
10 000 000	1.404	2.991	0.859	0.876	0.832	0.899
100 000 000	13.587	48.062	9.493	9.557	9.201	10.165

Quick-sort

```java
public int[] quickSort2(int[] a, int start, int slutt, boolean print) {
        if (print) {print(a);}

        while (start < slutt) {
            int median = medianOf3(a, start, slutt); //pivot midterste element
            int index = partition2(a, start, slutt, median);
            if (index - start < slutt - index) {
                quickSort2(a, start, index-1, print);
                start = index + 1;
            }
            else {
                quickSort2(a, index + 1, slutt, print);
                slutt = index-1;
            }
        }

        if (print) {print(a);}
        return a;
}

public int partition2(int[] a, int start, int slutt, int pivot) {
        int left = start; //left gås gjennom riktig vei
        int right = slutt-1; //right gås gjennom motsatt vei

        while (left <= right) {
            while (left <= right && a[left] <= pivot) {
                left = left + 1;
            }
            while (right >= left && a[right] >= pivot) {
                right = right - 1;
            }
            if (left < right) {
                int temp = a[left];
                a[left] = a[right];
                a[right] = temp;
            }
        }
        swap(a, left, slutt);
        return left;
}
```

Implementasjonen av algoritmen er in-place. For å få til dette trengs det en rekursiv hjelpemetode partition(`int[] a, int start, int slutt, int pivot`), som kan brukes til

å finne en indeks. Denne indeksen brukes videre i quickSort() for å rekursivt kalle den på venstre og høyre side av denne indeksen. Her vil man operere på en mindre del av arrayet for hver iterasjon av quickSort().

medianOfThree() finner elementet i midten av arrayet, og setter det til pivot. Koden til medianOfThree() følger med i javafilen vedlagt innleveringen. En implementasjon av quick-sort der pivot er siste element (i arrayet) er også i java-filen. Dersom man velger pivot lik siste element vil kjøretiden øke.

Mønster:

Quick-sort algoritmen vil for hver iterasjon velge pivot lik median, og kalle partition() som plasserer alle elementer mindre enn pivot foran, og alle elementer større enn pivot bak. Returnerer så indeksen til pivot. Deretter gjennomføres det rekursive kall på quickSort() ettersom høyre eller venstre side av pivot er størst.

Fallende rekkefølge or tilfeldig rekkefølge følger samme type mønster i quickSort(). Først sorteres høyre side av arrayet (pivoten), for så å sortere venstre side (av pivoten). For disse vil høyre halvdel av arrayet være ferdig sortert i løpet av halve gjennomføringen, venstre side når algoritmen er fullført. For arrayet som allerede er sortert vil få endringer utføres, og man ender til slutt opp med samme rekkefølge som man begynte med.

Kjøretid for quick-sort med pivot lik siste element:

	Random VL	Random	Sortert VL	Sortert	Rev. VL	Rev.
1 000	0.001	0.0	0.001	0.001	0.001	0.0
5 000	0.0	0.0	0.017	0.022	0.004	0.004
10 000	0.001	0.0	0.016	0.020	0.022	0.014
50 000	0.006	0.008	0.233	0.247	0.338	0.380
100 000	0.016	0.008	2.133	2.133	2.741	1.752
1 000 000	0.957	0.087	159.311	224.667	285.015	283.707
10 000 000	220.898	2.543	-	-	-	-
100 000 000	-	28.257	-	-	-	-

Kjøretid for quick-sort med pivot lik midterste element:

	Random VL	Random	Sortert VL	Sortert	Rev. VL	Rev.
1 000	0.002	0.0	0.0	0.0	0.0	0.0
5 000	0.002	0.001	0.0	0.0	0.0	0.0
10 000	0.001	0.001	0.0	0.0	0.0	0.0
50 000	0.006	0.007	0.004	0.0	0.008	0.001
100 000	0.023	0.009	0.013	0.001	0.14	0.001
1 000 000	1.388	0.096	1.051	0.010	1.048	0.011
10 000 000	118.212	1.066	237.217	0.267	148.105	0.149
100 000 000	-	11.740	-	1.366	-	1.567

Arrays.sort()

Kjøretid for Arrays.sort():

	Random VL	Random	Sortert VL	Sortert	Rev. VL	Rev.
1 000	0.002	0.0	0.0	0.0	0.0	0.0
5 000	0.001	0.001	0.0	0.0	0.0	0.001
10 000	0.002	0.0	0.0	0.0	0.0	0.0
50 000	0.005	0.007	0.0	0.001	0.001	0.002
100 000	0.005	0.005	0.001	0.0	0.0	0.005
1 000 000	0.024	0.085	0.008	0.001	0.008	0.0
10 000 000	0.289	0.930	0.039	0.005	0.062	0.012
100 000 000	6.219	24.093	0.429	0.045	0.570	0.104

Diskusjon av resultater:

Vedlagt i innleveringen er det en Excel-fil med navnet «speedtable.xlxs». Den inneholder en tabell for stor til å representere i dette dokumentet. Diskusjonen under baseres på funnene fra tabellen.

For det første ble jeg overrasket over hvor stor innvirkning valg av pivot hadde for quick-sort. Særlig under sortering av 1 000 000 elementer eller mer begynner pivot lik siste element virkelig å bruke lang tid, mens pivot lik median fint klarer både 1 000 000 og 10 000 000 elementer. Uten en value-limit på 100 verdier klarer den til og med 100 millioner elementer uten problem.

Hvor stor innvirkning gjentatte verdier skulle ha på større utregninger i quick-sort var også overraskende. For selection- og heap-sort er forskjellene ikke særlig merkbare (foruten random heap, der opprettingen av max-heap etter hvert vil ta mye tid). Ved 10 millioner elementer tar det opptil 99299,33 % ganger lenger tid (0.149 sec mot 148.105 sec) å sortere elementene med value-limit 100 i forhold til å sortere med hvilke som helst verdier. Det viser seg at ved like verdier vil partition bruke kvadratisk tid (O (n^2) tid) på å sortere dem. Sammenlikner man faktisk kjøretid med forventet kjøretid er ikke resultatet like overaskende. For utregninger uten value-limit kjører implementasjonen i O (n log n) tid.

Selection-sort skal i teorien alltid kjøre i O (n^2) tid. Ved å studere resultatene kan man se at de stemmer godt overens med dette. Differansen mellom tester med og uten value-limit vil ikke gi store utslag på kjøretiden, for hele arrayet må gjennomgås uansett.

For heap-sort skal kjøretiden i teorien være O (n log n) tid for alle typer input. Resultatene stemmer overens med dette. Noe overaskende er heap-sort gjerne raskere enn quick-sort, selv om quick-sort ofte betegnes som den raskeste algoritmen. For få antall elementer er quick-sort meget rask, men desto flere elementer man legger til, desto lenger tid bruker algoritmen i forhold til heap-sort.