Oblig3 – Stian Grimsrud stiangri

SelectionSort:

```
private static void selectionSort(int[] array){
    int aLength = array.length;

    for(int i = 0; i < aLength -1; i++){
        int index = i;

        for(int j = i +1; j < aLength; j++){
            if(array[j] < array[index]){
                index = j;
            }
        }
    int tmp = array[index];
    array[index] = array[i];
    array[i] = tmp;
    }
}
```

InsertionSort:

```
private static void insertionSort(int[] array){
    int aLength = array.length;
    for(int i = 1; i < aLength; i++){
        int j = i-1;
        int key = array[i];
        while(j >= 0 && array[j] > key){
            array[j+1] = array[j];
            j = j-1;
        }
    array[j+1] = key;
    }
}
```

QuickSort:

```java
private static void quickSort(int[] array, int first, int last){
    if(first >= last){return;}

    int par = inPlacePartition(array, first, last);
    quickSort(array, first, par-1);
    quickSort(array, par +1, last);
}
```

```java
private static int inPlacePartition(int[] array, int first, int last){
    int pivot = random.nextInt(last + 1 - first) + first;
    int tmp = array[pivot];
    array[pivot] = array[last];
    array[last] = tmp;
    int p = array[last];
    int l = first;

    pivot = last -1;
    while(l <= pivot){
        while(l <= pivot && array[l] <= p){
            l = l+1;
        }
        while(pivot >= l && array[pivot] >= p){
            pivot = pivot -1;
        }
        if(l < pivot){
            tmp = array[l];
            array[l] = array[pivot];
            array[pivot] = tmp;
        }
    }
    tmp = array[l];
    array[l] = array[last];
    array[last] = tmp;
    return l;
}
```

HeapSort:

```java
private static void heapSort(int[] a1){
    int n = a1.length;

    //Bygger heap
    for(int i = n/2-1; i >= 0; i--){
        heapify(a1, n, i);
    }

    //tar ut ett og ett element fra heapen
    for(int i = n-1; i>=0; i--){
        int temp = a1[0];
        a1[0] = a1[i];
        a1[i] = temp;

        heapify(a1, i, 0);
    }
}
```

```java
private static void heapify(int[] a1, int n, int i){
    int largest = i;
    int left = 2*i +1;
    int right = 2*i +2;

    if(left < n && a1[left] > a1[largest]){
        largest = left;
    }

    if (right < n && a1[right] > a1[largest]) {
        largest = right;
    }

    if(largest != i){
        int swap = a1[i];
        a1[i] = a1[largest];
        a1[largest] = swap;

        heapify(a1, n, largest);
    }
}
```

BucketSort:

```java
private static void bucketSort(int[] array, int max){
    int[] bucket = new int[max + 1];

    for(int i = 0; i < bucket.length; i++){
        bucket[i] = 0;
    }

    for(int i = 0; i < array.length; i++){
        bucket[array[i]]++;
    }

    int index = 0;
    for(int i = 0; i < bucket.length; i++){
        for(int j = 0; j < bucket[i]; j++){
            array[index++] = i;
        }
    }
}
```

Oppgave 3.

Selection sort var greit, ingen betydelige utfordringer.

Insertion sort, det samme gjaldt her.

Quicksort gikk også relativt fint, der valgte jeg å sette pivotelementet til random. Hvis jeg hadde valgt å bruke medianen hadde det kanskje gitt en bedre kjøretid, men ønsket ikke å bruke så mye tid på å velge element.

Heapsort gikk fint. Dette har vi gått gjennom veldig godt i forelesning.

Bucketsort var mer vrien å forstå, men etter at jeg forstod hvordan den fungerer, gikk greit å implementere.

Oppgave 4

Jeg har kjørt tester med tallene 0-9.

Selectionsort itererer igjennom alle tallene, finner det laveste og bytter plass med det elementet som er på den første plassen. Så igjennom alle tallene bortsett fra de som allerede er sortert og finner det neste laveste. Dette gjør den til den til listen er sortert.
Ved sortert liste går den bare igjennom alle elementene 10, men foretar ingen bytter.
Ved fallende liste kan vi se at listen egentlig er ferdig sortert når den har kommet halvveis. Dette fordi den først bytter det største elementet (som er først i listen) med det laveste elementet (som er sist i listen). Begge elementene har da riktig plass og det samme vil skje med de resterende elementene.

Ser ikke noe spesielt mønster i Insertionsort når listen er sortert.
på Reversert ser vi at de høyeste tallene i riktig rekkefølge blir «dyttet» mot høyre/sin riktige plass.
Med random liste ser vi hvordan listen stadig blir mer og mer sortert fra venstre til høyre.
Algoritmen går igjennom tall for tall og sjekker om tallene før i listen (til venstre) er større. Hvis tallet er større blir den satt inn på riktig plass.

Quicksort virker lit vilkårlig, men det er fordi jeg bruker random.nextint() til pivot.
Algoritmen sjekker om tallene er større/mindre enn pivoten og blir deretter flyttet til riktig side av pivoten. Hvis det kun de er riktig plassert i forhold til pivoten går den videre.
Dette er sortering fra random liste og reversert liste.

Heapsort sjekker om tallet som er på indeks 0 er det største, hvis ikke ser den på barna sine og sjekker den om noen av barna er større. Dette gjelder barna og, de sjekker om de er større enn sine barn igjen. Etter vi har fått en maksheap, vil den ta ut det største tallet og kjøre algoritmen rekursivt til den er sortert.

Bucketsort
Ved reversert kunne jeg se at tallet 6 ikke skrives ut på linje 5, men det blir skrevet ut igjen på linje 6.

Oppgave 5 – Test av hastighet

Tiden er i millisekunder med random genererte tall mellom 0 og 100000.
Hver kjøring med x antall elementer inneholder de samme tallene, slik at alle algoritmene må sortere de samme tallene.
Mørkegrønn = raskes
Lysegrønn = raskes uten Arrays

Rekkefølge:	Elementer:	Arrays.sort(..)tid:	Selection tid:	Insertion tid:	Quick tid:	Heap tid:	Bucket tid:
Sortert	1000	0,03	8,6	0,06	1,1	2,07	2,3
Reversert	1000	0,03	9,4	8,3	0,5	2,72	2,1
Random	1000	0,5	0,47	1,8	0,4	1,45	2,3
Sortert	10000	0,2	29,8	0,5	3	2,15	1,9
Reversert	10000	0,3	116,3	43,5	1,6	16,75	0,9
Random	10000	6,7	38,9	23,3	2,3	17,77	1,2
Sortert	100000	2,1	1677	3,9	36,2	21,3	4,2
Reversert	100000	3,2	13650	6616	28,3	40,13	7,6
Random	100000	22	3923	1505	21,1	64,04	1,5
Sortert	500000	4,2	47595	7,3	58,2	20,02	9
Reversert	500000	7,6	302599	155747	60,9	34,4	3,3
Random	500000	163	96455	42824	60,6	25,2	7,5
Sortert	1000000	9,3	202870	7,4	85,5	28,13	16,8
Reversert	1000000	64,1	1083081	713395	81,9	41,76	9
Random	1000000	214	422309	174486	116	35,8	11,5
Sortert	10000000	18,4	-	-	484	85,04	35
Reversert	10000000	17,6	-	-	644	51,27	51
Random	10000000	824	-	-	1349	24,9	46
Sortert	30000000	52	-	-	1967	91,75	106
Reversert	30000000	51	-	-	2934	35,23	97
Random	30000000	2118	-	-	5062	23,63	131

Resultatet stemmer overens med O-notasjonen til grafene.
Vi ser at selection sort som har $O(n^2)$ er tregest i alle tilfeller bortsett fra en. Det ene tilfellet er ved 1000 elementer og det er tregeste sorteringen er under 2 ms tregere, noe som så si er ubetydelig.
Bucket Sort, som har gjennomsnittlig kompleksitet O(n+k) og verste $O(n^2)$, er raskest i de fleste tilfeller (bortsett fra Arrays.sort()). Dette bortsett fra når listen allerede er sortert eller ved få elementer.
Jeg er ikke overrasket over at disse selection sort er tregest og heap sort er raskest, men at det skulle være så stor forskjell visste jeg ikke.
Bucket klarer å sortere 10 000 000 elementer på nesten samme tid som selection klarer 10 000
(i random rekkefølge).
Insertion er raskest når listen allerede er sortert, det stemmer med kompleksiteten som er o(n) ved beste tilfellet. Ellers er den relativt treg, slik som o-notasjonen tilsier ($O(n^2)$).

Heapsort som har var også rask. Det stemmer godt med O-notasjonen som er O(nlogn).

Quick sort er også relativt raskt, men fortsatt en del tregere enn bucket. Quick har gjenomsnittlig kjøretid på 0(n log(n)), mens bucket har o(n+k).