Rapport i IN2010: Oblig 3 – Sortering

NB!
Jeg har kommentert vekk printingen (av iterasjonene) for å holde styr på tiden i stedet.

Selection sort

Javakode:

```
public int[] selectionSort(int[] array){
  int length = array.length;

  for(int i = 0; i < length-1; i++){
    System.out.println("Iteration " + (i+1) + ":");

    //finner minste tallet i det usorterte arrayet
    int min = i;

    for(int j = i+1; j < length; j++){

      if(array[j] < array[min]){
        min = j;
      }
    }

    //swapper det minste (funnet) elementet med det forste elementet.
    swap(array, min, i);
    System.out.println(Arrays.toString(array));

  }
  return array;
}
```

Endringer og utfordringer:
Jeg har ikke opplevd noen utfordringer gjennom å implementere selection sort. Siden swapping forekommer i nesten alle algoritmene, så bestemte jeg meg for å lage en egen swap-metode.

Mønster:
Med ”mønster”, så antar jeg at dere er ute etter hvordan algoritmene fungerer.

Tilfeldig:
Siden tallene er tilfeldige, så vil bytting av plassene ...

Sortert:
Siden arrayet allerede er sortert, så vil det ikke foregå noen ”swaps”.

Reversert:
Verdiene blir swappet én etter én (dermed blir alle verdiene swappet og alle verdiene vil ha vært på indeks 1, en eller annen gang gjennom kjøringen).

Hvordan den fungerer:
Finner den minste verdien i den usorterte delen av arrayet og plasserer det i begynnelsen.

Insertion sort

Javakode:

```
public int[] insertionSort(int[] array){
    int length = array.length;

    for(int i = 1; i < length; i++){
      System.out.println("Iteration " + i + ":");

      int key = array[i];
      int j = i-1;

      //flytter elementene som er storre enn key ett hakk opp
      while(j>= 0 && key < array[j]){
        array[j+1] = array[j];
        j = j-1;
      }
      array[j+1] = key;
      System.out.println(Arrays.toString(array));

    }
    return array;
  }
```

Endringer og utfordringer:
De fleste av endringene jeg har gjort, handler om likhetstegn. Ellers har det ikke vært noen utfordringer knyttet til implementasjonen

Mønster

> **Tilfeldig**
> Ingen spesielle mønster. Følger algoritmen som den skal.
>
> **Sortert:**
> Ingen spesielle mønster.
>
> **Reversert:**
> Følger samme mønster som selection sort på reversert array.

Hvordan den fungerer:
Tar ett element og sammenlikner det med de tidligere elementene, for å stå putte det inn i riktig plass.

Quick sort

Javakode:

```
public void inPlaceQuickSort(int[] array, int from, int to){
   if(from >= to){
     return;
   }
   int left = inPlacePartition(array, from, to);
   inPlaceQuickSort(array, from, left-1);
   inPlaceQuickSort(array,left+1, to);
  }

public int inPlacePartition(int[] array, int from, int to){
   int pivot = array[to];
   int left = from;
   int right = to -1;

   while(left <= right){
     i++;
     System.out.println("Iteration " + i + ":");

     while(left <= right && array[left] <= pivot){
       left++;
     }
     while(right >= left && array[right] >= pivot){
       right--;
     }
     if(left < right){
       swap(array, left, right);
     }
     System.out.println(Arrays.toString(array));


   }
   swap(array, left, to);
   return left;
  }
```

Endringer og utfordringer:
inPlaceQuickSort- metoden forble den samme, men jeg måtte gjøre noen endringer på inPlacePartition-metoden. Møtte på utfordringer rundt pivot-elementet. I stedet for å velge medianen mellom den første, siste og midterste indeksen, valgte jeg å sette pivoten til å være det siste elementet.

Mønster:
Quick-sort fungerer på samme måte, uavhengig av innholdet i arrayet.

Tilfeldig:

Sortert:

Reversert:

Hvordan den fungerer:
Quick sort er en divide-and-conquer algoritme. Den velger et element som pivot-en og deler arrayet rundt dette punktet. De elementene mindre enn pivot-en plasseres på høyre side, mens de større: på venstre side.

Bucket sort

Javakode:

```
public int[] bucketSort(int[] array){
  int min = array[0], max = array[0];
  for(int value : array){
    if(value > max){
      max = value;
    }
    if(value < min){
      min = value;
    }
  }

  List<List<Integer>> bucket = new ArrayList<List<Integer>>(max);
  for (int i = 0; i <= max; i++) {
    bucket.add(new ArrayList<Integer>());
  }

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

  int index = 0;
  for(int i = 0; i <= max; i++){
    System.out.println("Iteration " + i + ": ");
    List<Integer> list = bucket.get(i);
    while(!list.isEmpty()){
      int value = list.remove(0);
      array[index++] = value;
      System.out.println(Arrays.toString(array));
    }
  }
  return array;
}
```

Endringer og utfordringer:
Jeg har slitet mest med denne algoritmen. Det meste av utfordringer kom av at jeg ikke la til et begrenset verdiområde, og dermed jobbet jeg med altfor store tall. Etter å ha begrenset det til tall fra 0-9, så ble det lettere å arbeide med denne algoritmen. Jeg valgte å bruke arraylists med like mange bøtter som den høyeste verdien. Dvs. Hvis den høyeste verdien i arrayet er 9, så lager jeg 9 bøtter. Utenom dette, så likner koden min på pseudokoden i pensumboka.

Mønster:
Siden jeg initialiserer verdiene i arrayet til å være null, så kan man se at verdiene blir lagt inn i riktig rekkefølge, fra den laveste indeksen til den høyeste.

Tilfeldig:

Sortert:

Reservert:

Hvordan den fungerer:
Denne algoritmen distribuerer verdiene i ulike buckets. De tallene med samme verdi, vil bli satt i samme bucket. Denne algoritmen flytter på verdiene to ganger: en gang til bucketen, og tilbake til det originale arrayet.

Rød indikerer treigest, mens grønn indikerer raskest.

		Selection sort	**Insertion sort**	**Quick sort**	**Bucket sort**	**Arrays.sort**
1 000	***Tilfeldig***	0.4647471	0.5854981	0.2829049	0.2474485	0.3099118
	Sortert	0.135024	0.0016669	0.5479692	0.2911867	0.0090965
	Reversert	0.5194188	2.5032054	0.0297639	0.106098	0.014604
5 000	***Tilfeldig***	1.9976695	1.3800727	0.7136603	0.681251	0.224112
	Sortert	1.7221292	0.0052137	1.4901606	0.7839305	0.0571946
	Reversert	1.7648281	1.6397155	0.5151735	0.496106	0.0613362
50 000	***Tilfeldig***	161.4203152	26.4593058	7.1448655	4.25841	1.810524
	Sortert	170.7794762	0.0470493	37.5967936	2.152453	0.2328954
	Reversert	128.1691402	57.4571678	73.5113651	2.4390929	2.3216356
100 000	***Tilfeldig***	651.3788088	104.3413134	14.3405754	8.6709525	2.4491658
	Sortert	656.7351113	0.114199	136.7422929	4.5627985	0.4587564
	Reversert	510.2075737	241.7556795	159.7788534	4.7910506	0.4994775

Tabellen viser sorteringer av array med et begrenset verdiområde (0-9). Dersom jeg kjører algoritmene med ubegrenset verdiområde så får jeg følgende feilmelding:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.base/java.util.ArrayList.<init>(ArrayList.java:154)
        at Sorting.bucketSort(Sorting.java:265)
        at Sorting.bucketTest(Sorting.java:295)
        at Sorting.testRandom(Sorting.java:69)
        at Sorting.main(Sorting.java:20)
```

Årsaken til dette ligger i bucket sort. I bucket sort, så lager jeg en liste buckets på lengde lik den høyeste verdien i lista. Dersom den innebygde funksjonen Random generer det høyeste tallet til å være lik 9 999 999, så vil det si at vi har 9 999 999 bøtter, selv om det ikke er plass i minnet. Dermed får man ut OutOfMemory exception.

Jeg får kun sjekka tidene på array av random verdier, før programmet spyr ut exception, men her er det jeg får ut:

```
Array with random values:
TIME FOR SELECTION SORT:
0.0193205
TIME FOR INSERTION SORT:
0.012105
TIME FOR QUICK SORT:
0.0095053
TIME FOR BUCKET SORT:
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.base/java.util.ArrayList.<init>(ArrayList.java:154)
        at Sorting.bucketSort(Sorting.java:265)
        at Sorting.bucketTest(Sorting.java:295)
        at Sorting.testRandom(Sorting.java:69)
        at Sorting.main(Sorting.java:20)
```

Som vi kan se, så er det minimale forskjeller på tidsforbruket. Den eneste forskjellen er feilmeldingen grunnet minne.

Overrasket?

Jeg er overrasket over hvor bra Java sin innebygde sorteringsalgoritme gjør det. Arrays.sort bruker en form for quick sort, og dermed er jeg sjokkert over at tiden mellom Arrays.sort og quick sort ikke er på samme ”wave-length”. For eksempel: å sorterte en reversert liste med 100 000 elementer tar ca. 160 ms med quick sort, men kun ca. 0,5ms med Javas innebygde funksjon.

Praksis vs. Teori:

Bucket sort har et kjøretid på O(n + N), som er det raskeste vi får til. Vi kan se fra tabellen at denne algoritme gjør det ganske bra, uavhengig av antall elementer og type liste. Jeg vil dermed konkludere med at teorien stemmer ganske bra over rens med den faktiske kjøretiden. Derimot er det verdt å bemerke at selv om denne algoritmen er kjent for å være den raskeste, så er ikke dette tilfellet i virkeligheten.

Quick sort sin tidskompleksitet er på O(n^2). Man ville ha trodd at denne algoritmen (min implementasjon og javas innebygde versjon) ville ha vært ganske treg, men det viser seg å ikke stemme. Arrays.sort er den algoritmen som er raskest i de fleste tilfellene, mens quick sort har vært treigest kun én gang. Jeg vil dermed si at teorien ikke stemmer så godt overens med virkeligheten.

Insertion sort, lik quick sort, har en tidskompleksitet på O(n^2). Dermed vil man anta at denne algoritmen bruker svært lang tid. Ved å analysere tabellen, så kan man se at insertion sort har flere raske kjøretider, enn treige. Dermed stemmer heller ikke teorien bak denne algoritmen.

Selection sort er desidert den treigeste algoritmen (se tabell). Akkurat som quick- og insertion sort, så har denne algoritmen en kjøretid på O(n^2). Ut ifra tabellen, så vil jeg si at teorien stemmer med den virkelige kjøretiden.