Oppgave 3:

Implementasjon av selection sort:

```
public void selectionSort(int[] array){
  int i, j, value, index, teller = 0;
  for(i = 0; i < array.length; i++){
    value = array[i];
    index = i;
    for(j = i; j < array.length; j++){
    if(array[j] < value){
      value = array[j];
      index = j;
    }
  }
  //swap
  if(value < array[i]){
    teller = array[i];
    array[i] = array[index];
    array[index] = teller;
  }
 }
}
```

Implementasjon av insertion sort:

```
public class Insertion{
public void insertionArray(int[] array){
  int tall = array.length;

  for(int i =1; i < tall; i++){
    int test = array[i];
    int j = i -1;

    //Flytter alle elementer av array som er si

    while(j>=0 && array[j] > test){
      array[j+1] = array[j];
      j = j-1;
    }
    array[j+1] = test;
  }
}
```

pilasila

Implementasjon av quick sort:

```
public int[] quick(int array[], int start, int slutt){
  if(start < slutt){
    int index = partition(array,start, slutt);
    quick(array, slutt, index-1);
    quick(array,index+1, slutt);
  }
  return array;
 }

public int partition(int array[], int start, int slutt){
  int teller = array[slutt];
  int i = (start-1);
  for(int j = start; j< slutt; j++){
    if(array[j] <= teller){
      i++;
      int swap = array[i];
      array[i] = array[j];
      array[j] = swap;
    }
  }
  int swap = array[i+1];
  array[i+1] = array[slutt];
  array[slutt] = swap;
  return i+1;
}
```

Implementasjon av bucket sort:

```
class Bucketsort{
 void bucketSort(int array[], int n){
   int[] bucket = new int[n+1];

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

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

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

 }
```

Selection, insertion og quick var helt greit å forstå, og gjøre om til javakode, men synes det var vanskelig å forstå seg på bucket sort og implementere kode for dette. Hvis jeg forstod det riktig skal koden fungere nå. Møtte på noen utfordringer underveis, da jeg skulle teste quicksort med x-antall elementer, men det ordnet seg etterhvert.

Oppgave 4:

Utifra utskriften i ser jeg ikke noe forskjell, fordi alle algoritmene skriver ut tallene på samme måte, altså i sortert rekkefølge. Tror dermed tidskompleksiteten vil variere mellom hver av algoritmene fordi alle sorterer på forskjellige måter, dette kan man se ut ifra koden hvordan de flytter elementer foran eller bak for å sortere, eller finner miderste elementet for så å sortere. Oppgaven ba oss sortere på 3 forskjellige måter med sortert array, usortert og ved bruk av 10

tilfeldige tall(random)elementer. Jeg kunne ikke utifra utskriften anta hvordan algoritmene fungerte fordi alle skrev ut samme elementer, men koden sier mye om hvordan algoritmene vil fungere. **Selection sort** fungerer på måten at den gjentatte ganger går gjennom listen over elementer og for hver hang velger den et elementer i henhold til søket og plasserer elementet i riktig posisjon. Fordelen med selection sort vil være at den vil fungerer bra på lister som har få elementer i seg. Jeg vil tro den også da vil kreve liten lagringsplass fordi man beholder den opprinnelige listen og lager ikke en ny liste av elementer som blir sortert. Dermed tror jeg at ved sortering av lister med flere elementer vil den ikke være så effektiv, fordi nå må gå gjennom alle elementer i listen helt til den er sortert i riktig rekkefølge. Ved denne oppgaven vil jeg tro at den vil være mest effektiv ved liste som er sortert allerede, fordi den vil gå gjennom listen, og trenger ikke gjøre swap av elementer.

Ved **insertion sort** vil alle elementene gås gjennom før den for hver gang setter inn elementer som er i usortert posisjon til riktig posisjon. Akkurat som selection sort vil denne være tidskrevende for lister med flere elementer. Nettopp fordi hvert element som er usortert må flyttes til riktig posisjon. Tror derfor også denne vil kreve mye tid og plass for den reverserte listen og listen som er tilfeldig generert.

Bucket sort er en algoritme som ikke trenger å sammenligne tall for å sortere, som vil si at den lager en rekke bucket og plasserer elementer som skal sorteres inn i disse bucketsene basert på deres index. Indexene må ha egenskapen og tilfredstille at objekt a kommer før objekt b, derfor også at a sin bøtteindex ikke kan være større en b sin. Den sorterer så etter dette. Som nevnt for de andre tror jeg at denne typen algoritme vil være passende for lister med få elementer, men kreve plass og tid for lister med flere elementer.

Quick sort algoritmen fungerer på måten at den deler listen i to basert på miderste elementet. Alle elementer i den ene listen vil være mindre enn den andre, og elementer i den andre vil være høyere enn den første listen. Så vil elementer flyttes i hver av listene til den er sortert i riktig rekkefølge. Tror denne algoritmen vil være den mest effektive i de fleste tilfeller, fordi den sorterer i hver sin liste parallelt, og da vil det gå fortere å sortere elementene i listen. Tror kanskje den vil være mest effektiv ved en allerede sortert liste her også. Og bruke litt mere tid på reversert og tilfeldig sortert.

Oppgave 5:

Antok at det ville ta kortere tid å sortere arrayen som er stigende sortert. Fordi alle verdiene ligger på riktig plass som de skal være. Så da vil loopen gå gjennom tallene hvis det allerede er på den plassen det skal være på, så vil det ikke bruke til på å swap plass på tallet. Tror derfor også at fallende og random vil bruke mere tid fordi alle verdiene på gjøre plass bytte for å komme til sin riktige posisjon. Under er bilde av tiden det tok med x-antall elementer. Det som er markert i gult er de som gikk raskest, det som er markert rødt var tregest.

Faktisk kjøretid vs. forventet kjøretid:

Insertion: O(n^2), basert på kode.

Quick sort: generelt ville det tatt → Time(n) = T(k) + T(n-k-1) + (n), her er k antall elementer som er mindre enn pivot(midterste elementet) i koden. I verste tilfellet kan den bruke O(n^2). Og ved beste tilfellet kan den bruke O(nlgn).

Bucket sort: ved verste scenario vil den bruke O(n^2). Og ved vanlige tilfeller vil den bruke O(n + k) hvor n er lengden av elementer i array, og K er antall bucket som brukes.

Selection sort: O(n^2), basert på kode som også er verste tilfellet. Ved beste tilfellet vil den også være O(n^2).

Basert på dette vil jeg tro at teorien stemmer med den faktiske kjøretiden, for det første fordi quick sort er den raskeste ut ifra tabellen jeg lagde. Og for det andre hvis vi ser på selection sort for ekspemel så vil O(n^2) være både worst og best case, og utifra tabellen min så er selection sort den tregeeste algoritmen. Tror derfor at teorien stemmer endel med praksis.

pilasila

Reversert:

Elementer	Selection sort	Insertion sort	Bucket sort	Quick sort	Arraysort
1000	57.204843 ms	25.326014 ms	3.1229124 ms	0.3298542 ms	1.2983159 ms
5000	129.239783 ms	72.722328 ms	7.6622878 ms	6.3013693 ms	7.6717008 ms
10 000	410.203697 ms	230.879201 ms	19.2963234 ms	19.1655763 ms	20.3092668 ms
50 000	6484.320664 ms	5599.820697 ms	443.908643 ms	432.1546666 ms	475.1354973 ms
100 000	23978.012646 ms	24525.063573 ms	1975.2760417 ms	2228.0871544 ms	1823.2608321 ms

Tilfeldig:

Elementer	Selection sort	Insertion sort	Bucket sort	Quick sort	Arrays sort
1000	8.319114 ms	0.5383984 ms	0.8640762 ms	0.028584 ms	0.2985614 ms
5000	21.620319 ms	1.7507516 ms	0.144672 ms	0.0759382 ms	0.5068199 ms
10 000	23.926943 ms	1.7286852 ms	0.1438578 ms	0.1392212 ms	0.2413452 ms
50 000	677.210997 ms	46.3568012 ms	0.2179382 ms	0.5999032 ms	1.2563065 ms
100 000	2051.538676 ms	176.477812 ms	0.284129 ms	0.8840141 ms	1.1926467 ms

Er det noen av resultatene i tabellen som overrasker deg?
Ble overrasket over at ved noen tilfeller var bucket sort raskere enn quick sort. Usikker på hvorfor dette forekom. Ble også overrasket over hvor stor forskjell det var på tid mellom selection sort og insertion sort, hvor selection sort bruker mye lengere tid enn insertion sort.