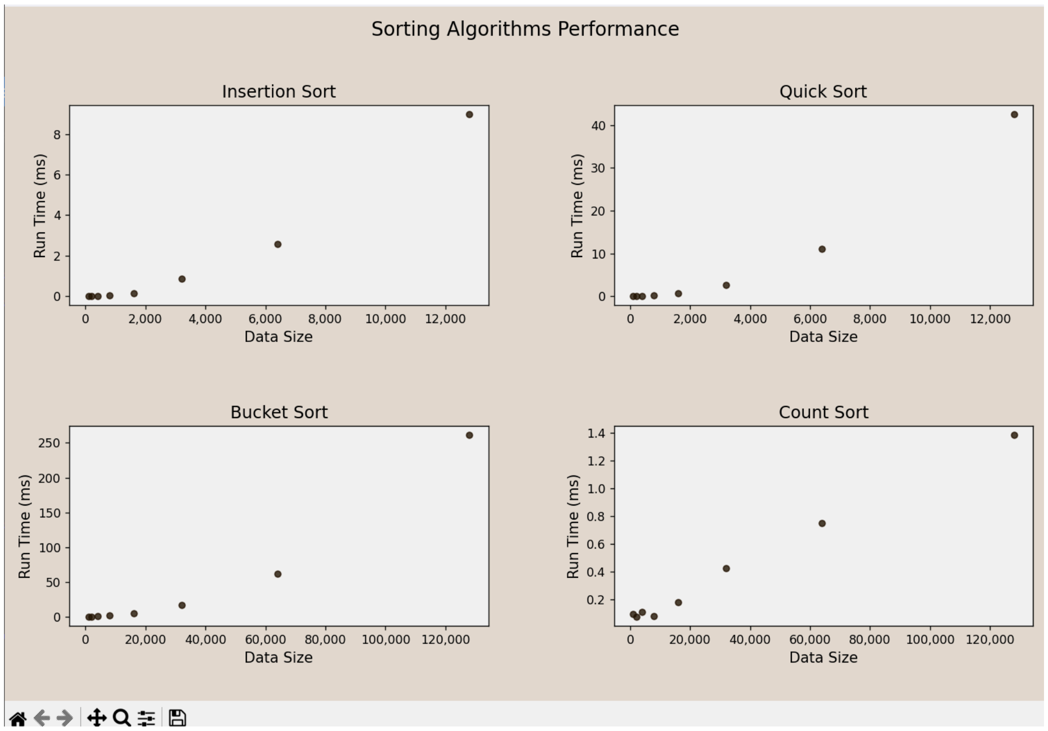


1)

a) Insertion Sort

i) For insertion sort, the complexity should be O(n^2) with it being O(n) in the best case. We can see the shape of the plot to be a curve up similar to a quadratic function and that there is an upward trend as a curve as the data size goes up. Due to the smaller size of the data that was used, we can see the shape of the quadratic fairly easily and we can make out the curve. If we also look at the points and estimate using the data size we can see the n^2 relationship, for example, the largest data point is about half of the second largest and we can see that the runtime of the largest data point is n^2 the one of the second largest, however, this is also a consequence of the smaller dataset, but most likely since insertion sort is n^2 the trend should follow through for larger datasets.

b) Quick Sort

    i) For quick sorting, the complexity should be O(logn) but can sometimes be O(n^2) in the worst-case scenario. Although the shape of the quick sort graph is seemingly n^2, I believe that is only due to the scale of the graph itself, or due to the fact that I had to create the quicksort graph with the small list instead which could be a reason why the graph looks n^2. The shape however could also be interpreted as nlogn and we can test these by doing the same test as I did previously by comparing the largest and second-largest we can clearly see that the runtime is not n^2 based of these two points. I also see that the runtime for this sort method was longer than the insertion sort method according to the scales.

c) Bucket Sort

    i) The complexity for this sort should be o(n+k) but in the worst case, it can be o(n^2) due to it using insertion sort and o(n) in the best case. In my case, It looks a lot more like a quadratic function rather than a linear function which leads to the assumption that the runtime in this case is o(n^2) which was the worst case but if we look at the scales it does not seem to be o(n^2) but o(n+k) instead.

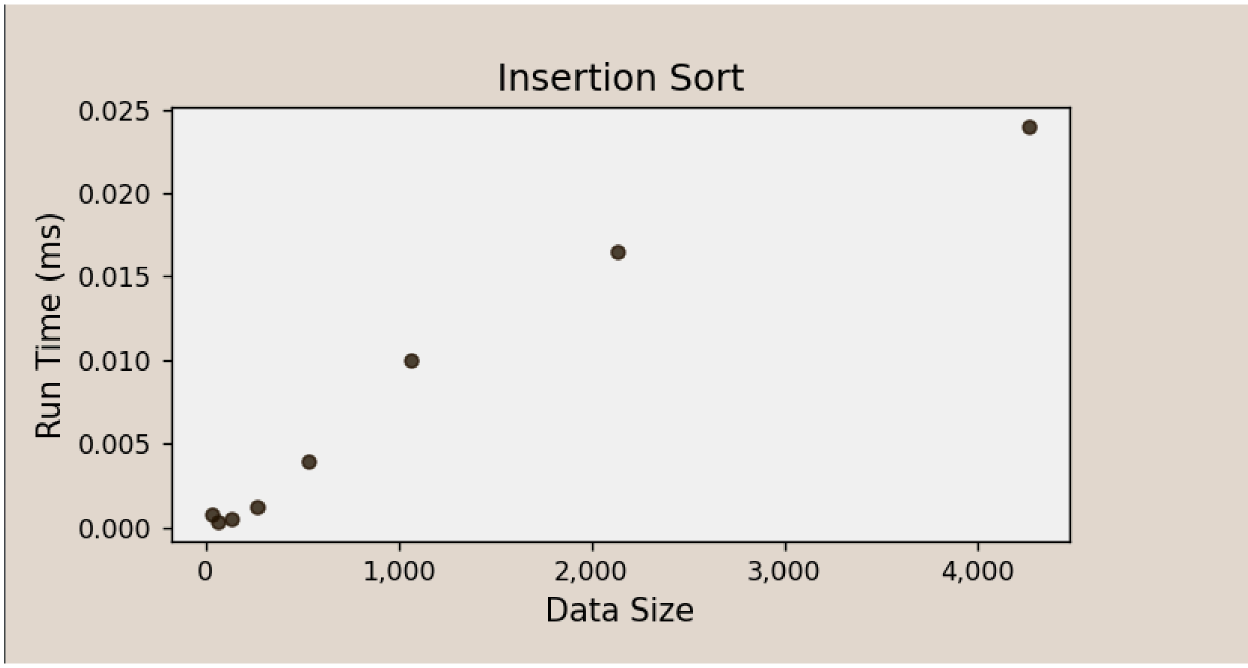
The scale of this graph is also much larger due to it the data size being much larger.

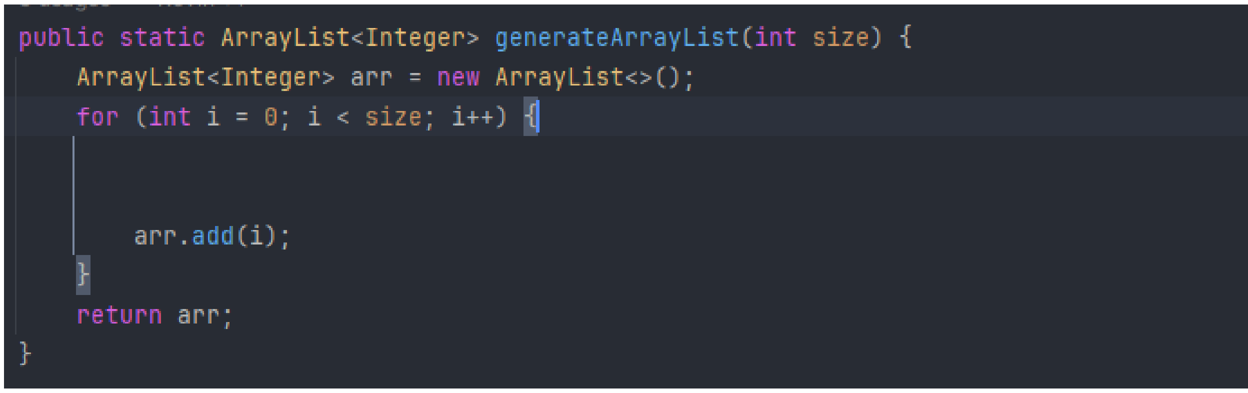
d) Count Sort

i) Count sort should have a complexity of O(n+k) all around. Akin to the complexity the graph of count sort displays a nearly linear plot. The runtimes scale for count sort is also much smaller compared to the other 3 sorts

2) Modified Insertion

a)

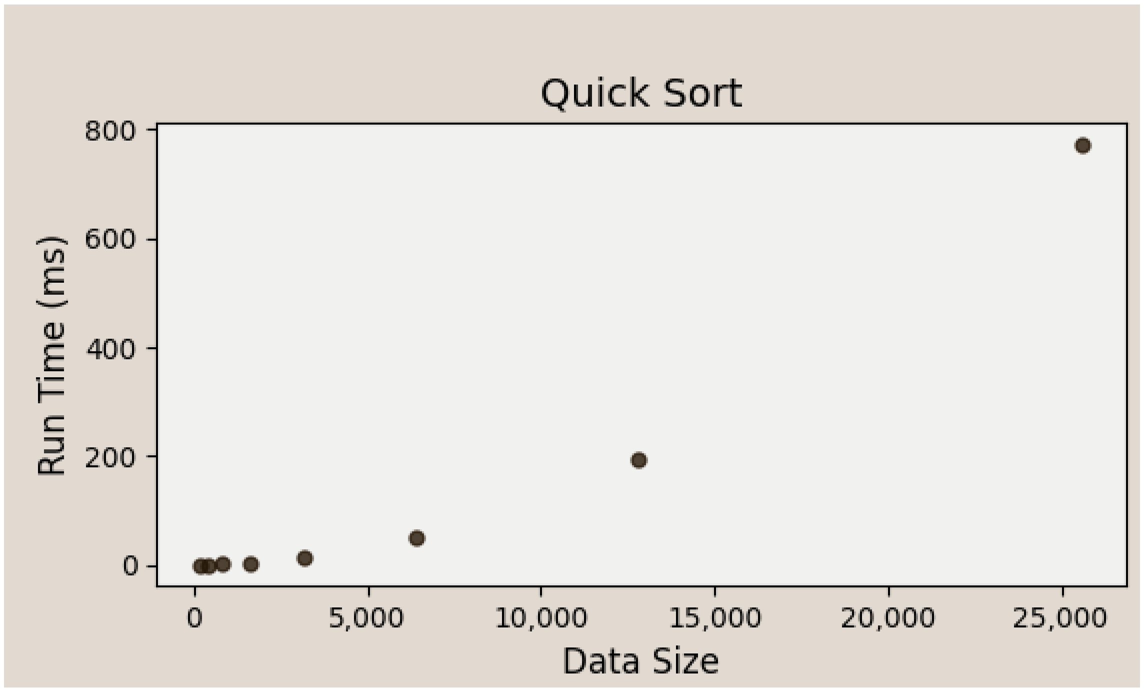


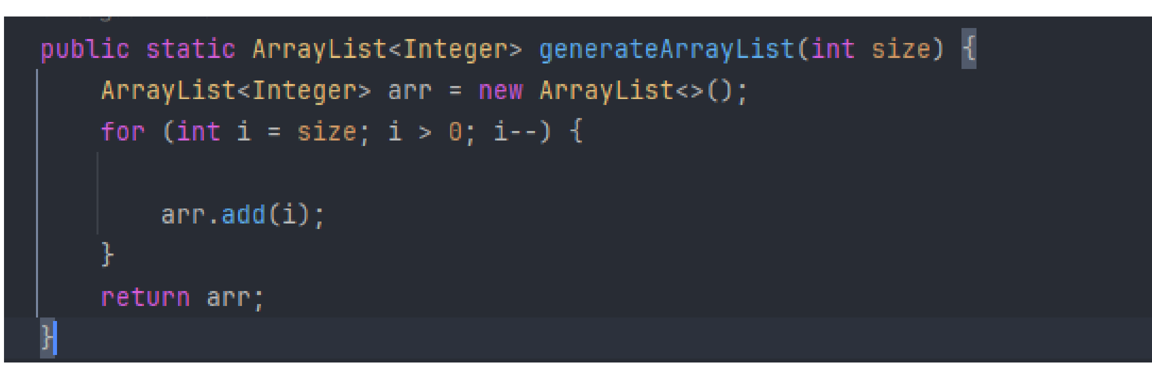
b) 

```java
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    for (int i = 0; i < size; i++) {


        arr.add(i);
    }
    return arr;
}
```

c) Since the best runtime for Insertion is O(n) which happens when the list is already in increasing order we can see that the runtime in this graph is much smaller in the previous one and it shows a more linear plot compared to the plot before.

3) Quicksort Modified

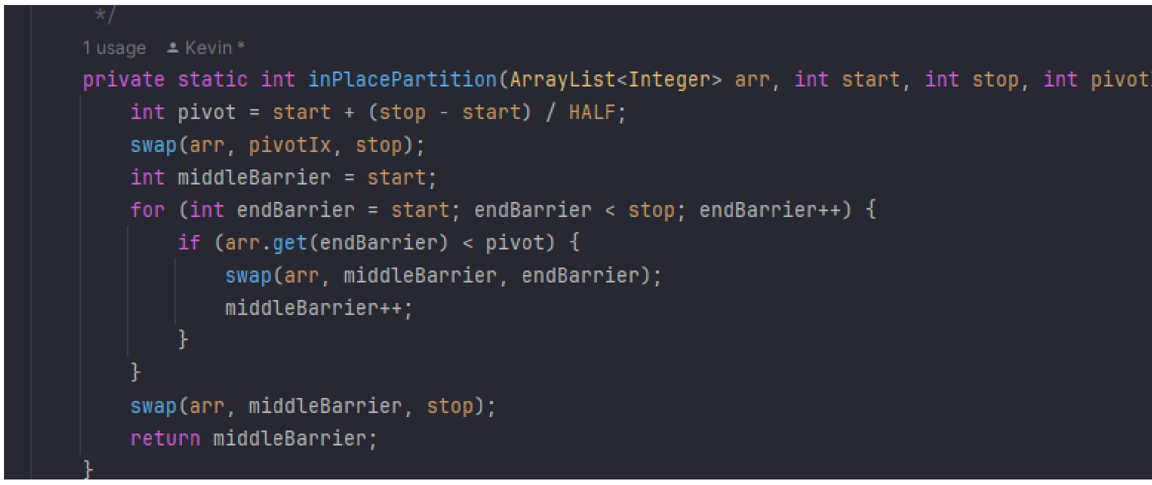
a) 

i) 

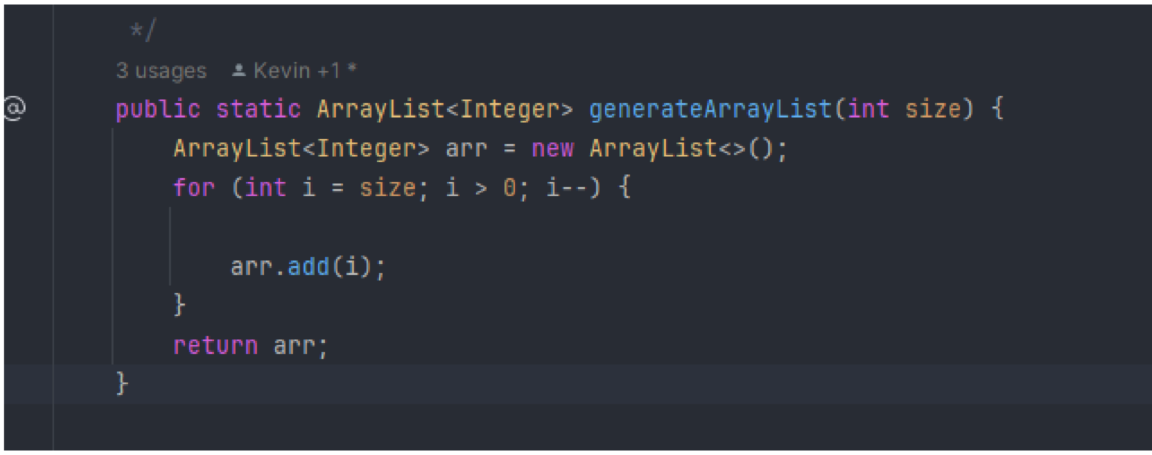
```java
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    for (int i = size; i > 0; i--) {

        arr.add(i);
    }
    return arr;
}
```

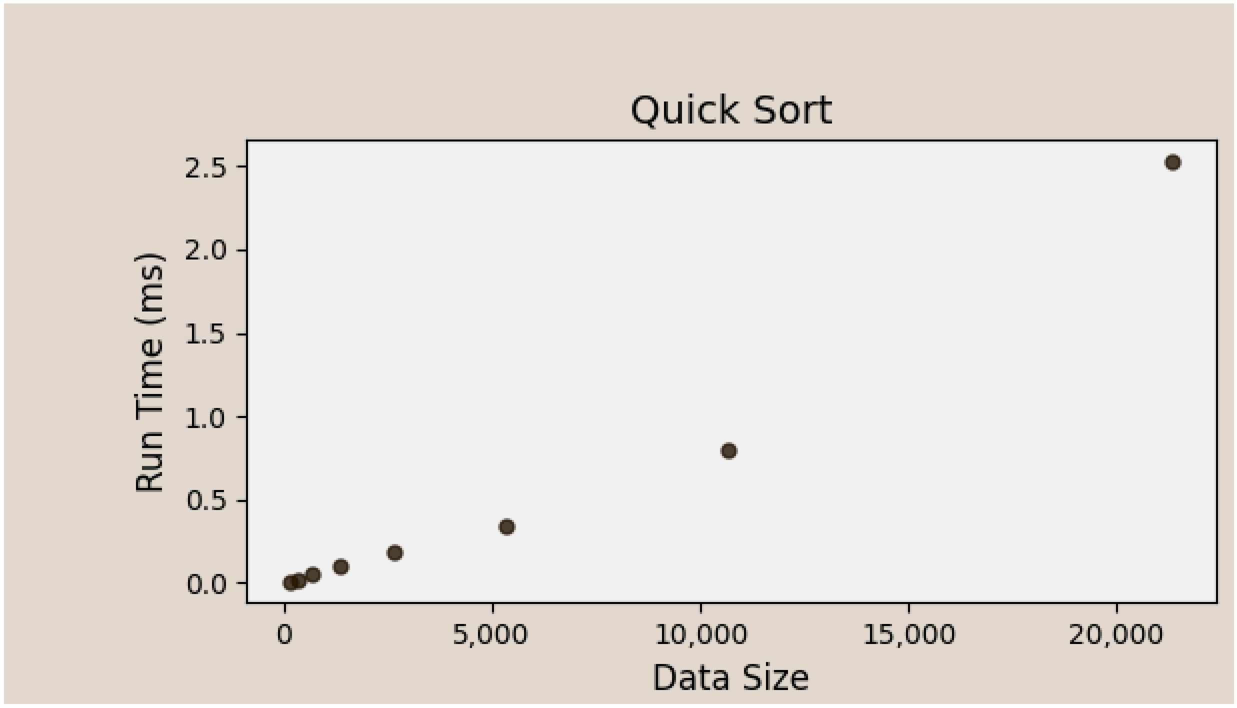
ii) This quick sort plot is slower and the time complexity is higher than the original graph. This happened because the input array was in descending order and since this version of quicksort, it chooses the start element which is the biggest element, which makes it choose the largest pivot.

4) Quicksort Modified pt2

i) 

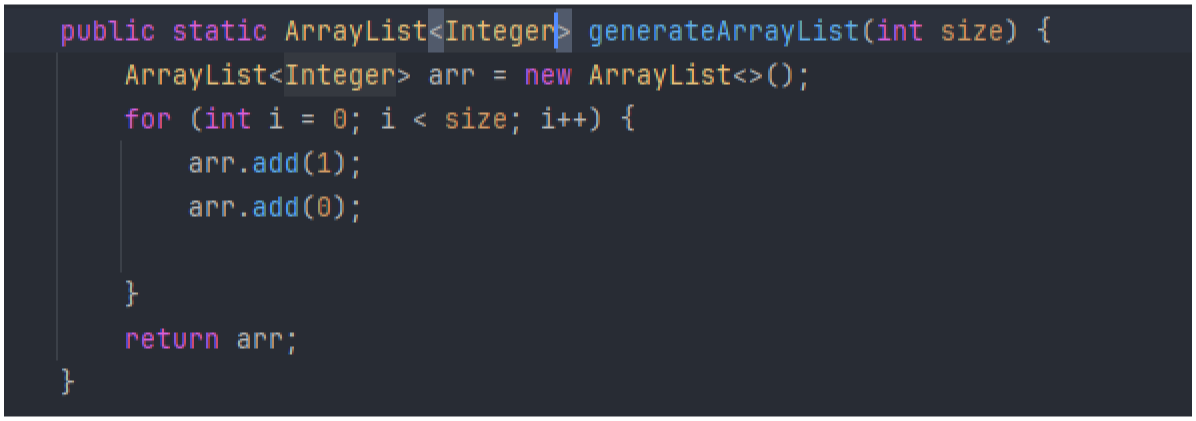
```java
    */
    private static int inPlacePartition(ArrayList<Integer> arr, int start, int stop, int pivot
        int pivot = start + (stop - start) / HALF;
        swap(arr, pivotIx, stop);
        int middleBarrier = start;
        for (int endBarrier = start; endBarrier < stop; endBarrier++) {
            if (arr.get(endBarrier) < pivot) {
                swap(arr, middleBarrier, endBarrier);
                middleBarrier++;
            }
        }
        swap(arr, middleBarrier, stop);
        return middleBarrier;
    }
```

ii) 
```
    */
    3 usages   Kevin +1 *
    public static ArrayList<Integer> generateArrayList(int size) {
        ArrayList<Integer> arr = new ArrayList<>();
        for (int i = size; i > 0; i--) {

            arr.add(i);
        }
        return arr;
    }
```

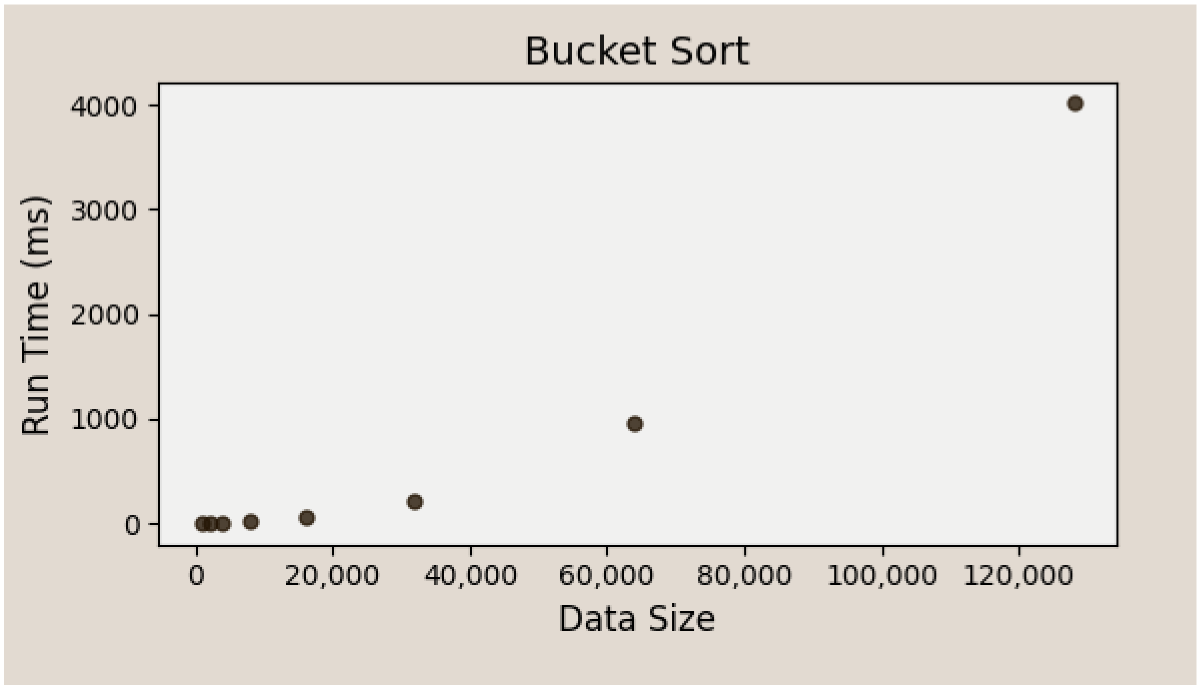
b) 

i) Changing the pivot to be the middle element decreases the runtime by a lot because the runtime for quicksort depends largely on whether the pivot is good or not and when the pivot is in the middle it is the ideal pivot which is why the runtime is much smaller compared to choosing the start element as the pivot.

5) Modified bucket

a) 

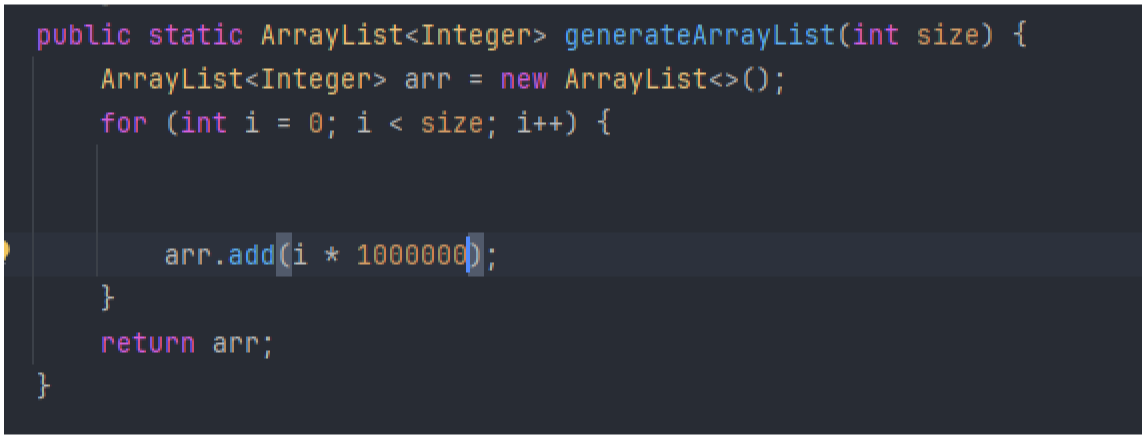
```java
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    for (int i = 0; i < size; i++) {
        arr.add(1);
        arr.add(0);

    }
    return arr;
}
```

b) 

i) The runtimes now are much larger than before because the new array I made put the points in a way where there would be a lot more data in one bucket which would cause insertion sort to run for a lot longer than if the buckets were evenly distributed

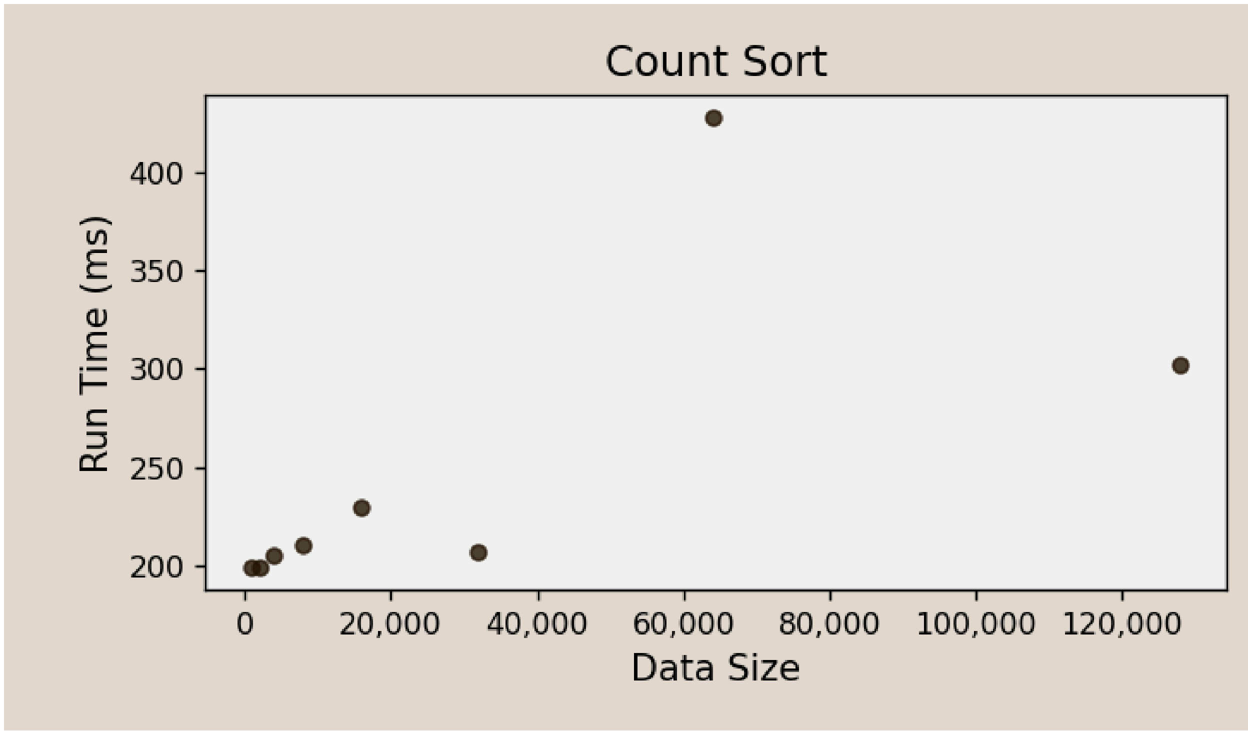
6) Count sort modified

a)



```
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    for (int i = 0; i < size; i++) {


        arr.add(i * 1000000);
    }
    return arr;
}
```

b)



i) This count sort is much slower than the original one because the range of values I provided was much larger than the input size, meaning since the runtime is O(n + k) where n is the size and k is the max range of the values and if k was very large it would add to the runtime depending on the size of k.