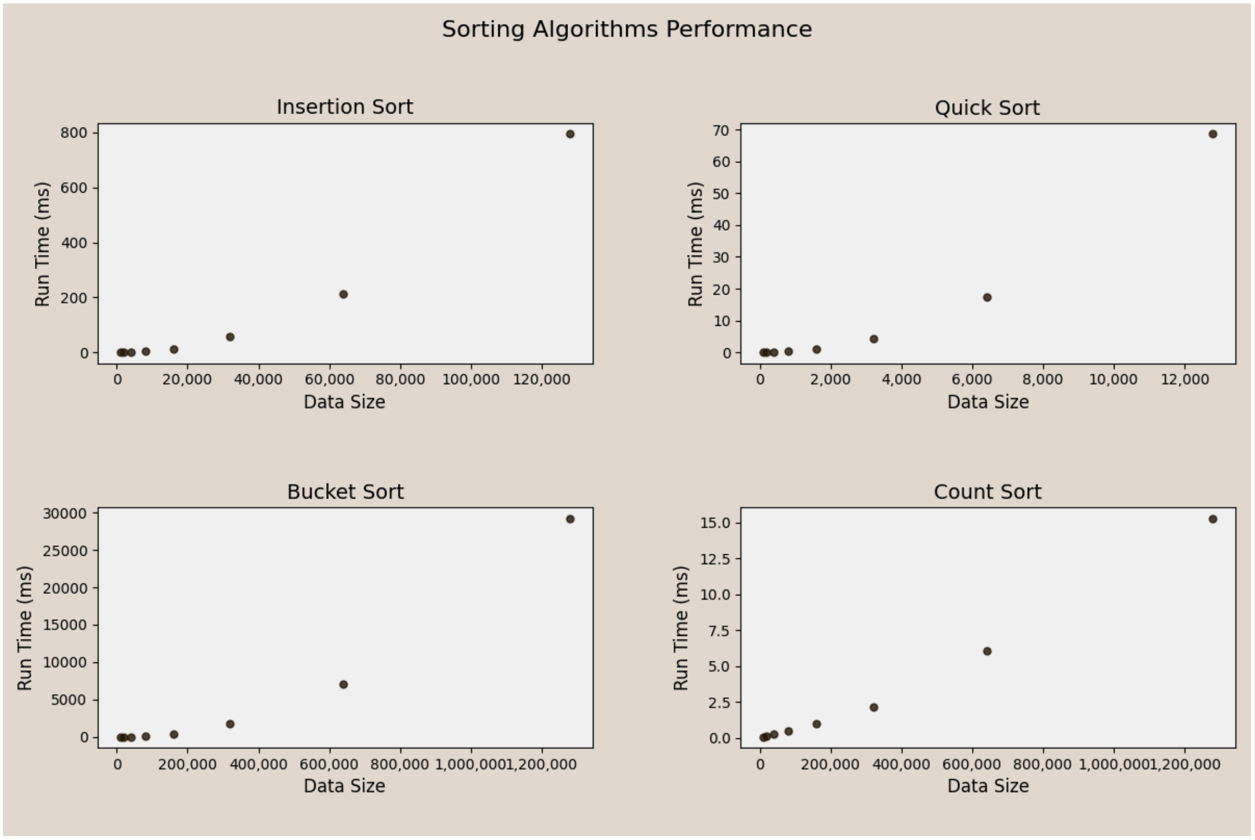


Task(1)

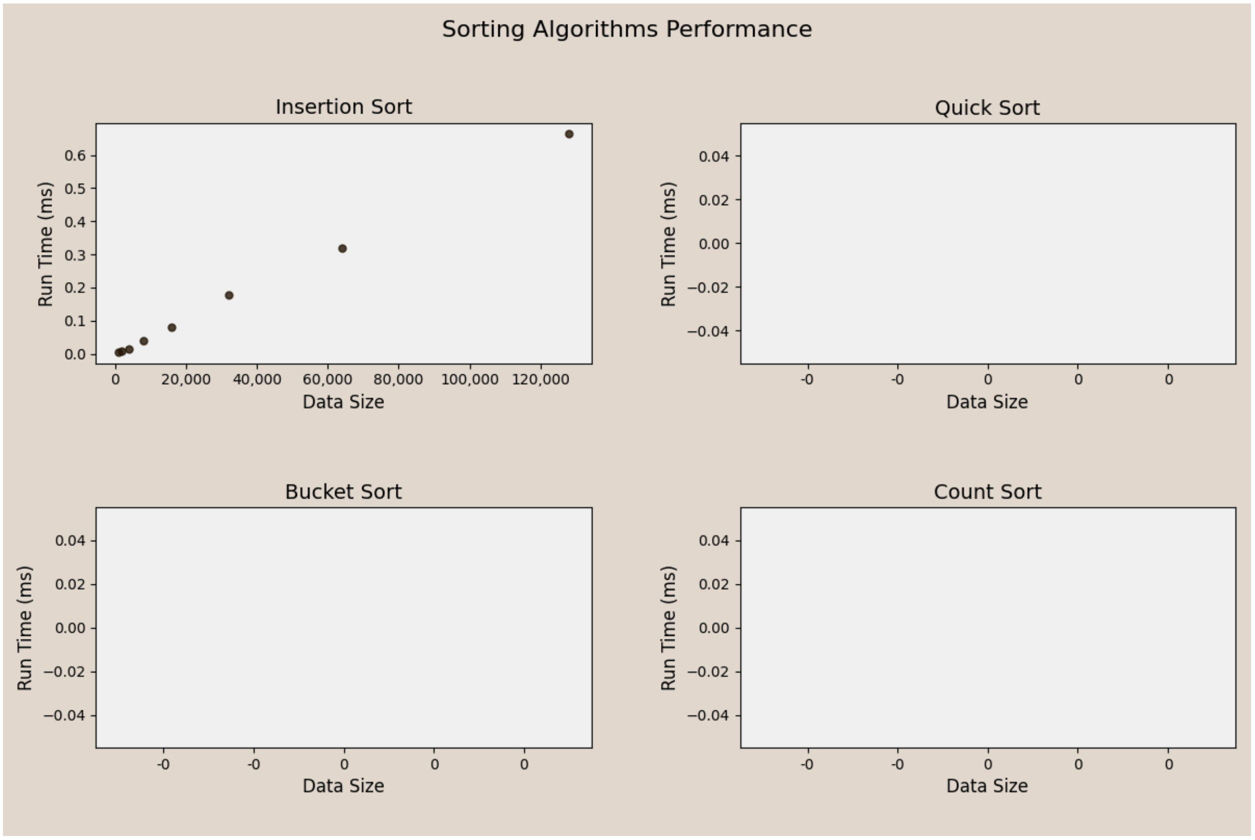
Insertion Sort:
In the plot, the runtime increases significantly as data size increases. This indicates that the time complexity of Insertion Sort is O(n^2). Therefore the graph also resembles a quadratic graph because the greater the data size, the greater the runtime.

Quick Sort:
In the plot, the runtime increases by a relative amount as data size increases. This indicates that the time complexity of Quick Sort is O(nlogn). Therefore the graph is also a curve similar to insertion sort where the greater the data size, the greater the runtime. However it is not as exponential as insertion sort with a quadratic growth, quick sort has a flatter curve with logarithmic growth. Quick Sort was also run on smaller data sizes due to its recursive nature.

Bucket Sort:
In the plot, the runtime before 1.2 million data size has a relatively linear curve. This indicates that when the amount of buckets are small, the time complexity of bucket sort is O(n) but as the amount of buckets becomes significantly large, the algorithm becomes inefficient as seen on the spike in the graph.

Count Sort:
In the plot, the runtime increases at the same scale as data size. This indicates that the time complexity of count sort is relatively O(n), although there is a spike at 1.2 million data size. Therefore the graph should appear the most linear and efficient with a slight curve at the end point of 1.2 million data size.

Overall, the axes of each sort scales differently, from these figures, it appears that bucket sort and count sort are more suitable for larger data sizes, insertion sort struggles with randomly sorted data and quick sort is more suitable for smaller data sizes due to its recursive nature.

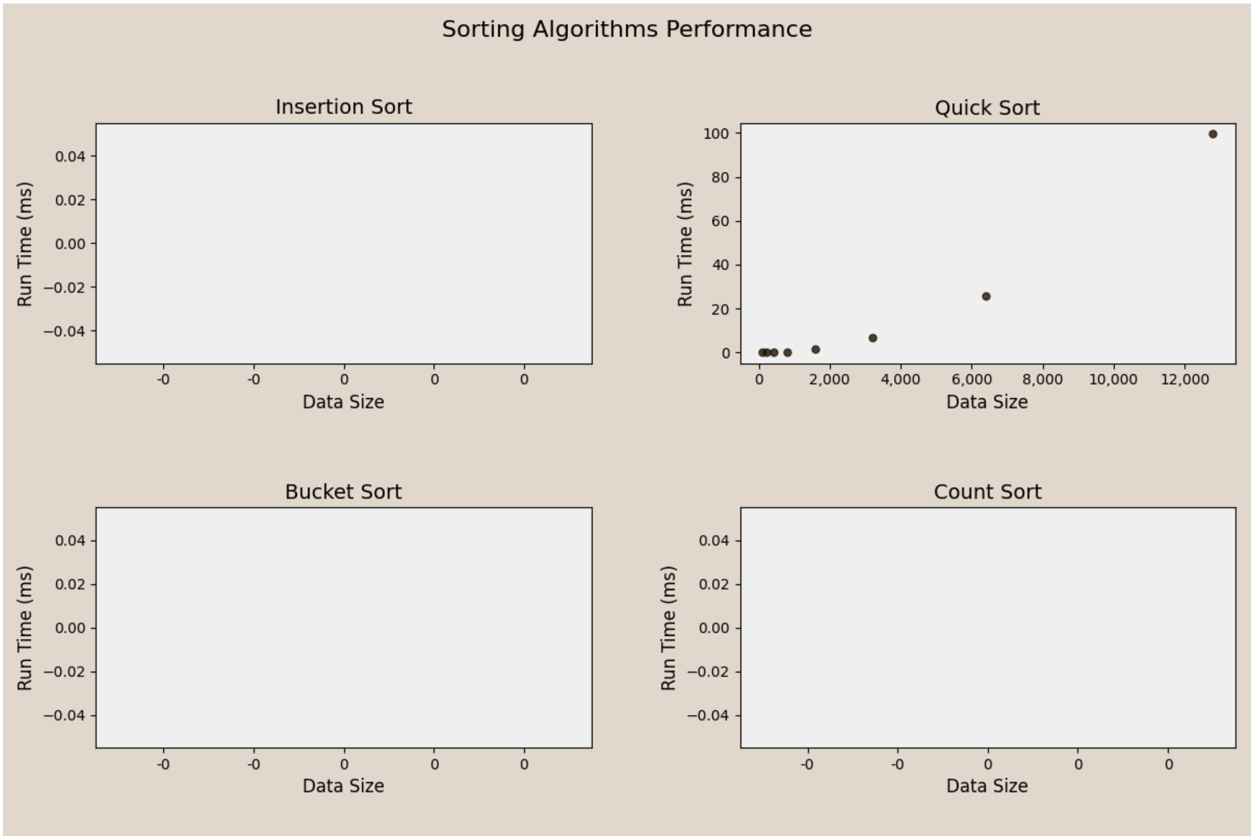


Task(2)

Code:

```
public static ArrayList<Integer> generateArrayList(int size) {
   ArrayList<Integer> arr = new ArrayList<>();
   for (int i = 0; i < size; i++) {
       arr.add((int) (Math.random() * 5) + i);
   }
   return arr;
}
```

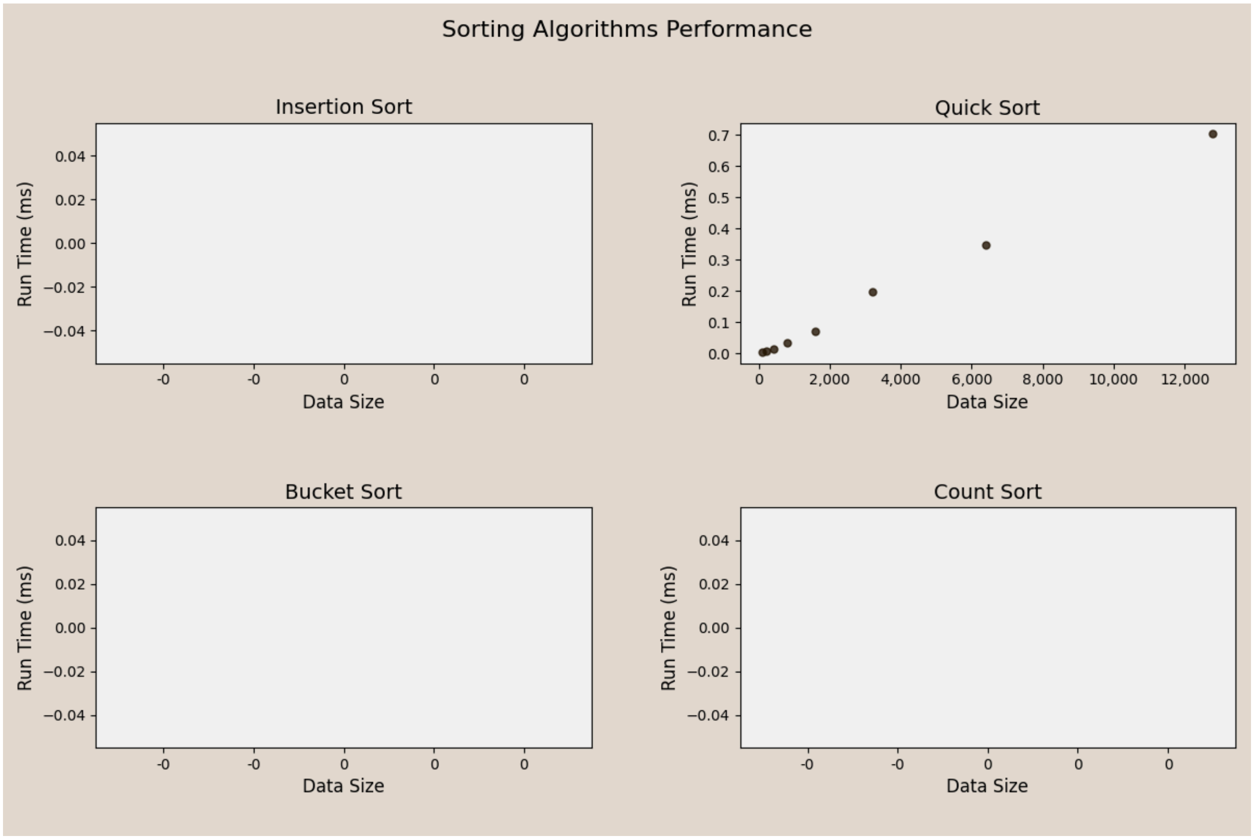
To achieve a faster insertion sort, I modified generateArrayList to generate integer arrays that were partially sorted, with overall ascending values so that each number will not be compared to many other numbers. In the plot, the runtime increases linearly with data size therefore, its graph also has a linear shape. Therefore, compared to its previous graph of sorting a random integer array, it is more efficient. Additionally it is also way faster, with runtimes being less than a hundredth of the previous graph at the same data size.



Task(3)

```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;
}
```

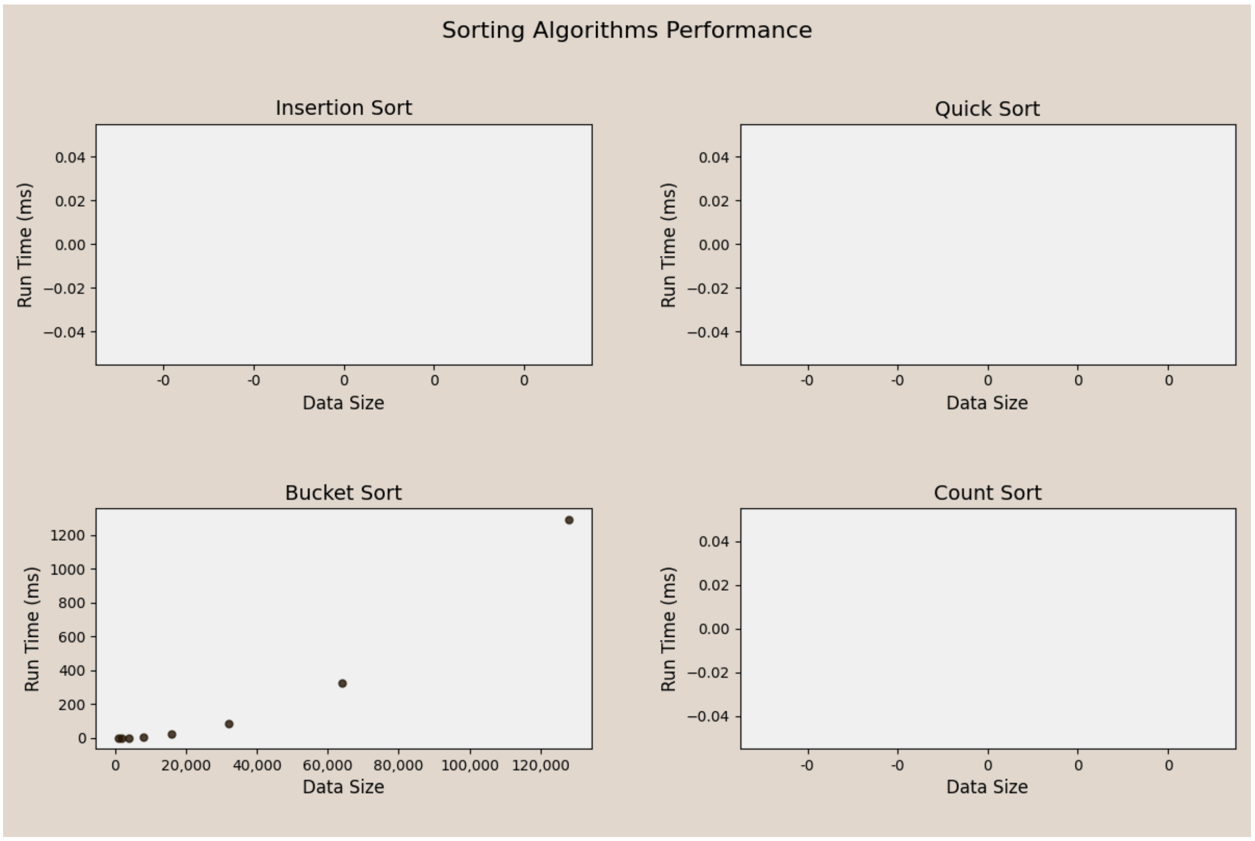
To get a slower Quick Sort, I modified generateArrayList to create an integer array that is in descending order, this makes for the most uneven split each iteration of the algorithm. Time complexity is now O(n^2), and will have a quadratic shape compared to the previous graph's logarithmic shape, making it much slower as can be seen through the average increase of around 30-50% in the runtime with the same data size.



Task(4)

```
pivotIx = start + (stop - start) / 2;
```

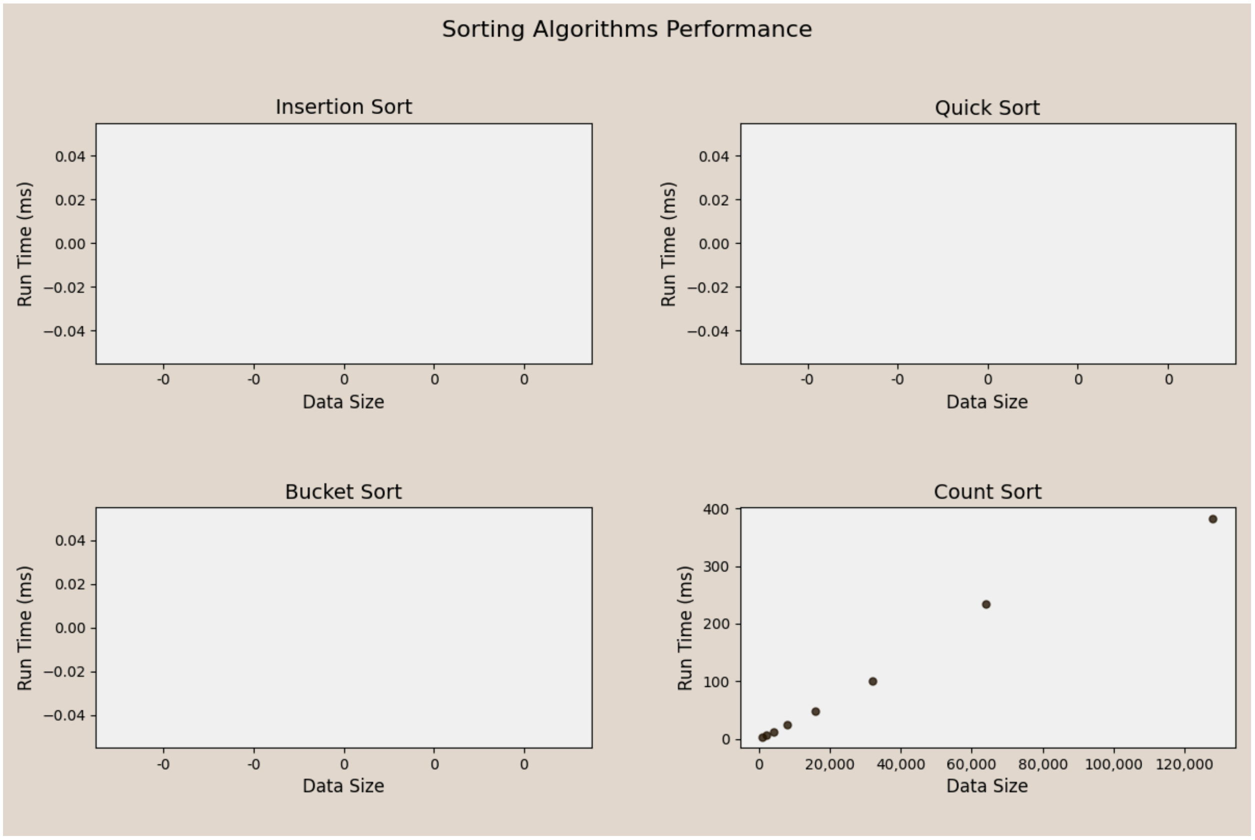
By assigning the pivot index to the middle index of a descending integer array, the pivot will always be the middle value, this creates an even split in all iterations creating the most efficient scenario for Quick Sort, as seen on the graph the time complexity is now O(nlogn) and seemingly flatter than the original graph in task 1. It also becomes much faster roughly a tenth in runtime compared to the original graph in task 1 due to the pivot always being the most ideal pivot instead of random.



Task(5)

```
public static ArrayList<Integer> generateArrayList(int size) {
   ArrayList<Integer> arr = new ArrayList<>();
   for (int i = size; i > 0; i--) {
       arr.add((int)(Math.random() * 50));
   }
   return arr;
}
```

To slow down bucket sort, I created an integer array that all had similar values, therefore all the numbers would cluster in a few buckets only. This causes an increased time as more internal sorting will be required. I even had to lower the data size as the original data size took too long. As seen when compared to the original graph in task 1, the difference in runtimes is about only 200-300% less while the data size has been reduced by 1000%.



Task(6)

```
public static ArrayList<Integer> generateArrayList(int size) {
   ArrayList<Integer> arr = new ArrayList<>();
   for (int i = size; i > 0; i--) {
       arr.add(i * 5000);
   }
   return arr;
}
```

To slow down Count Sort, I created an integer array with big value differences between them. This would cause a significantly large counting array making the algorithm super slow as well as inefficient. Compared to the original graph in task 1, even though I decreased the data size by 1000%, the runtime still increased by 200-300%.