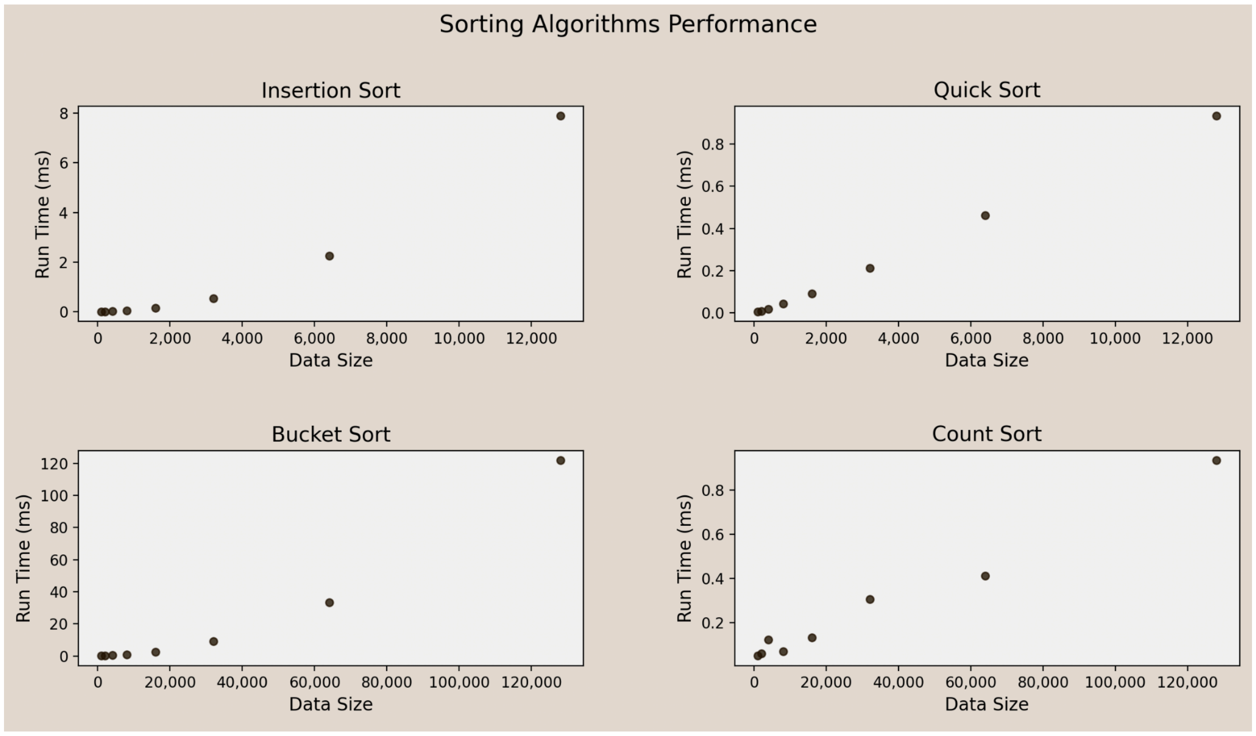
Ken Ogihara
A16969236
PA5

Part 2.2

Task (1)



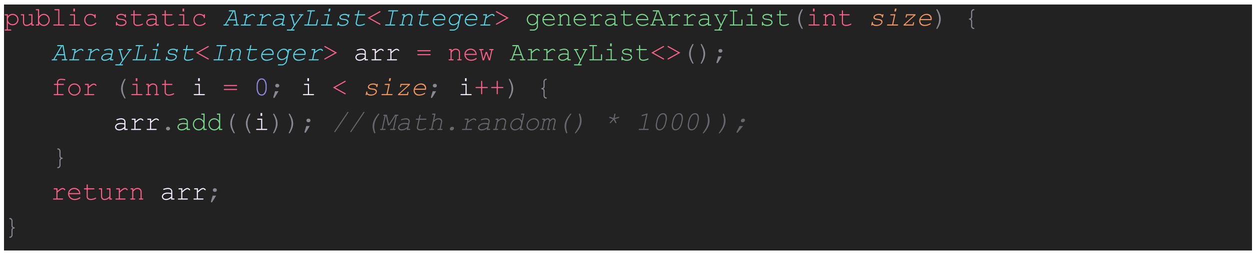
1. The small input sizes were used for insertion sort which may skew our analysis since we don't have a large enough input size that we can observe on. But according to the graph, it seems that as we increase the size of the input array, the time it takes to process them increases at a quadratic rate. In other words, the time complexity for insertion sort is O(n^2).

2. For quicksort, I used the small input sizes again because my computer was running out of memory. According to the graph, we can see that this function is running relatively fast at O(nlogn) time complexity.

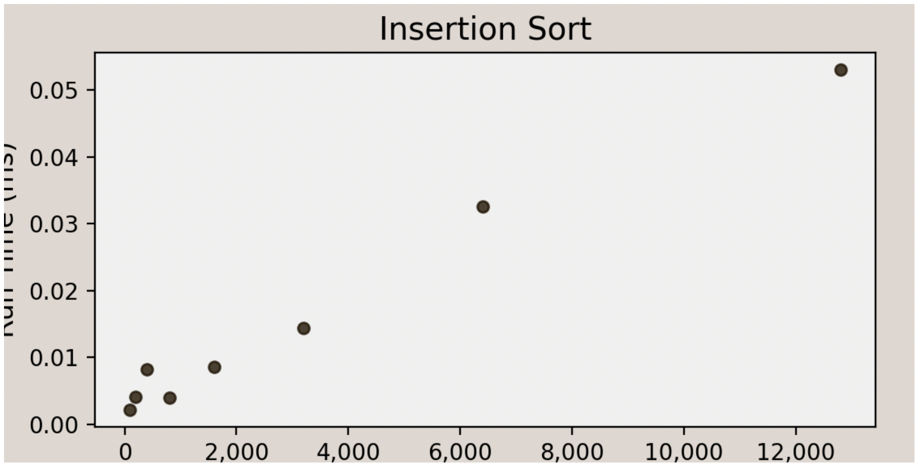
3. Bucket sort seems to be in O(n^2) time complexity. This time we are using large input sizes.

4. If we perform regression on our count sort graph, it would most likely be linear. When the input size is doubled, the run time also tends to double. Therefore, time complexity is O(n+k)

Task (2)

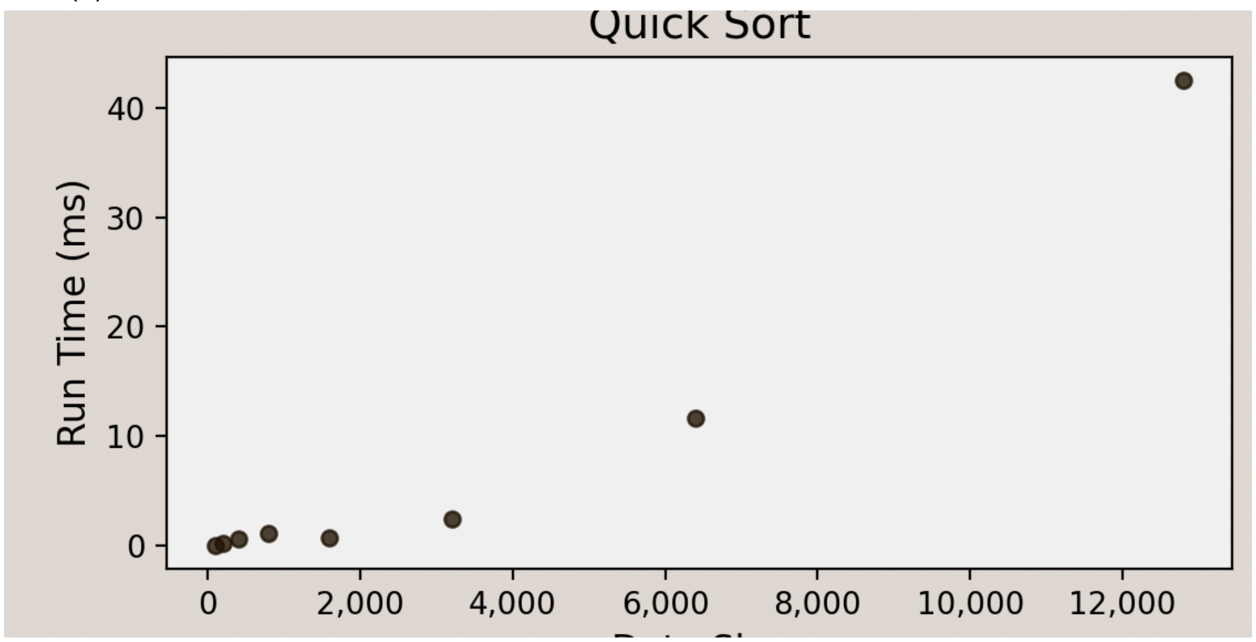


```java
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    for (int i = 0; i < size; i++) {
        arr.add((i)); //(Math.random() * 1000));
    }
    return arr;
}
```



Insertion sort is now operating in linear time complexity. This is because the array is already in sorted order. I simply commented out the Math.random() part which prevented the array from being unsorted.

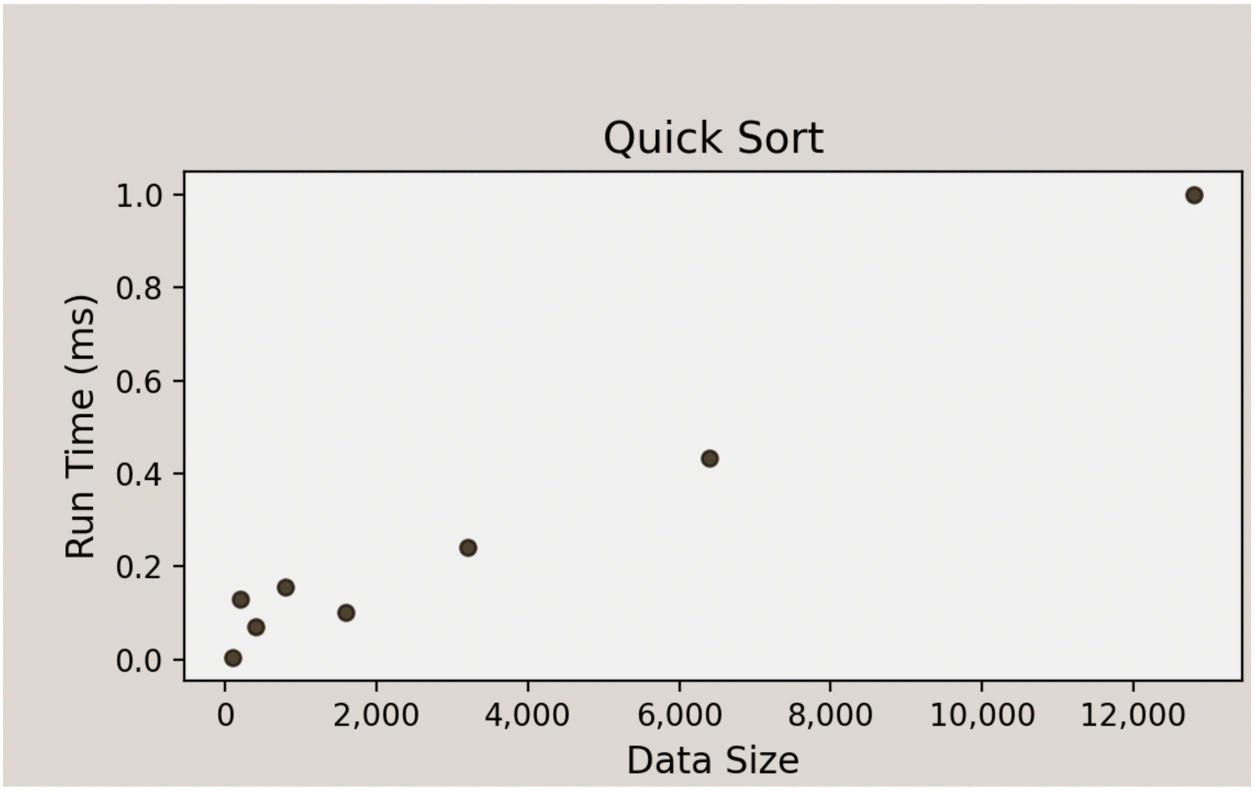
Task (3)



```
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    for (int i = 0; i < size; i++) {
        arr.add((i)); //(Math.random() * 1000));
    }
    return arr;
}
```

Now that elements are added to the array in ascending order, our quicksort method is slower because the pivot position is at the first index. Quicksort method performs fastest when the input is randomized and slowest when the pivot position is at the minimum value. This is because the nested recursion structure has to process every single element one by one. As a result, our new quicksort method is operating in O(n^2)

Task (4)

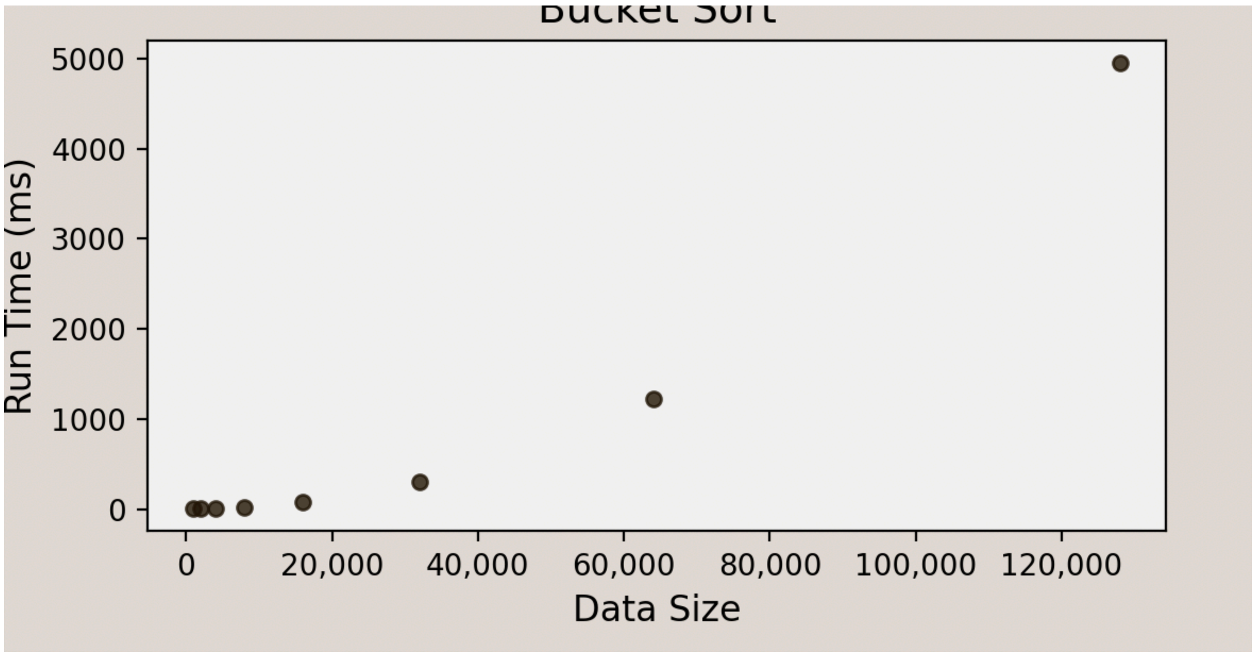


```
private static int inPlacePartition(ArrayList<Integer> arr, int start, int
stop, int pivotIx) {
    int pivot = arr.get(pivotIx + (stop - pivotIx) / 2);
    swap(arr, pivotIx, stop);
    int middleBarrier = start;
    for (int endBarrier = start; endBarrier < stop; endBarrier++) {
        if (arr.get(endBarrier) < pivot) {
            swap(arr, middleBarrier, endBarrier);
            middleBarrier++;
        }
    }
    swap(arr, middleBarrier, stop);
    return middleBarrier;
}
```

Quick sort is not in O(n^2) time complexity anymore. This is because we modified the pivot to be in the middle of the array. We achieved this by assigning pivot to be equal to the index of the difference between the end index and first index and dividing it by 2. This allows for a more balanced/efficient partitioning process. According to our findings, the new time complexity is O(nlogn).

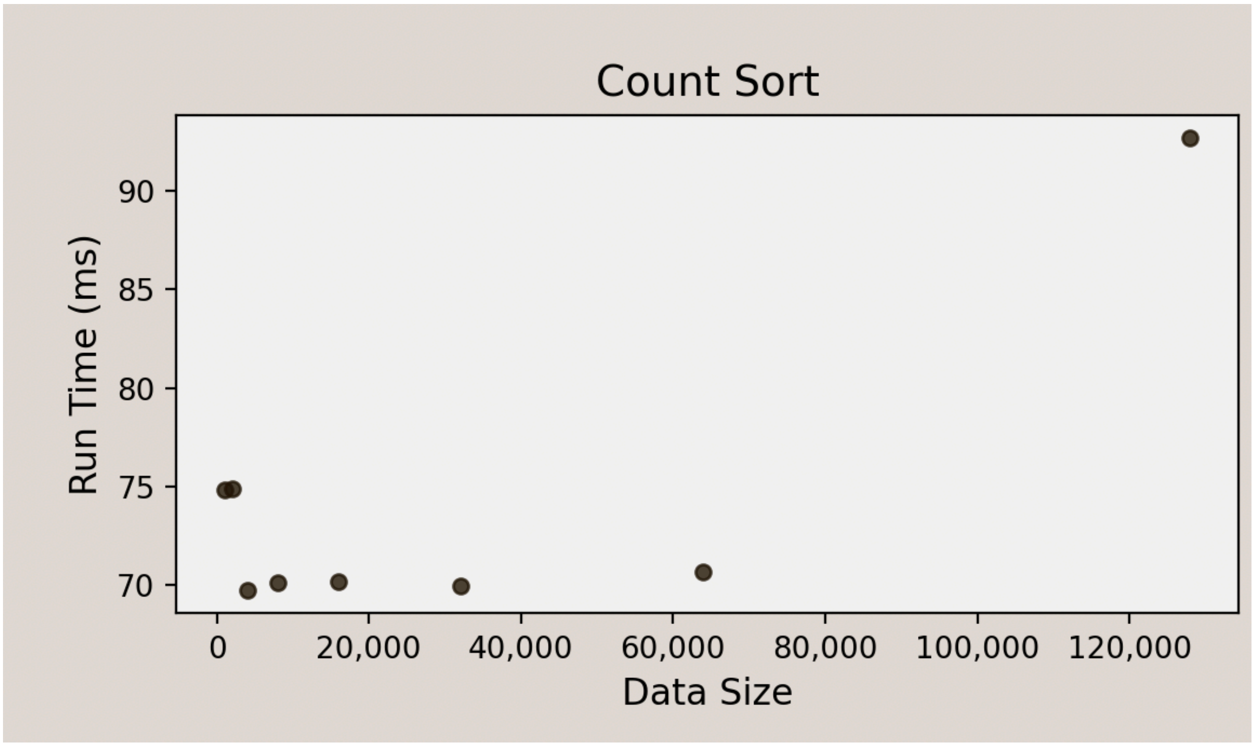
Task (5)

```
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    for (int i = 0; i < size; i++) {
        arr.add(i % 2); //(Math.random() * 1000));
    }
    return arr;
}
```



Although bucket sort is slower when there's a skewed distribution of elements among buckets, the reason why the algorithm got slower in my case is because the insertion sorting is taking longer. The elements in the array are alternating between 0 and 1, meaning there are only two values. This limits the amount of buckets we can use, thus feeding a bigger input size for the insertion sort. According to the graph, the total run time is about 40x slower than the original.

Task (6)



```java
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    for (int i = 0; i < size; i++) {
        arr.add((int) (Math.random() * 100000000));
    }
    return arr;
}
```

At every input size, the total run time significantly increases. This is because the range of values has increased. We can come to the conclusion that count sort is not ideal when the range of values is large.