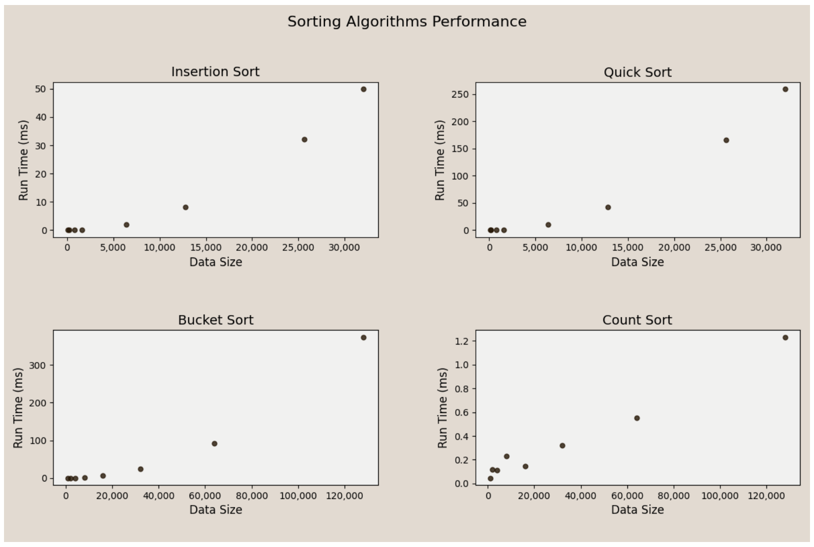
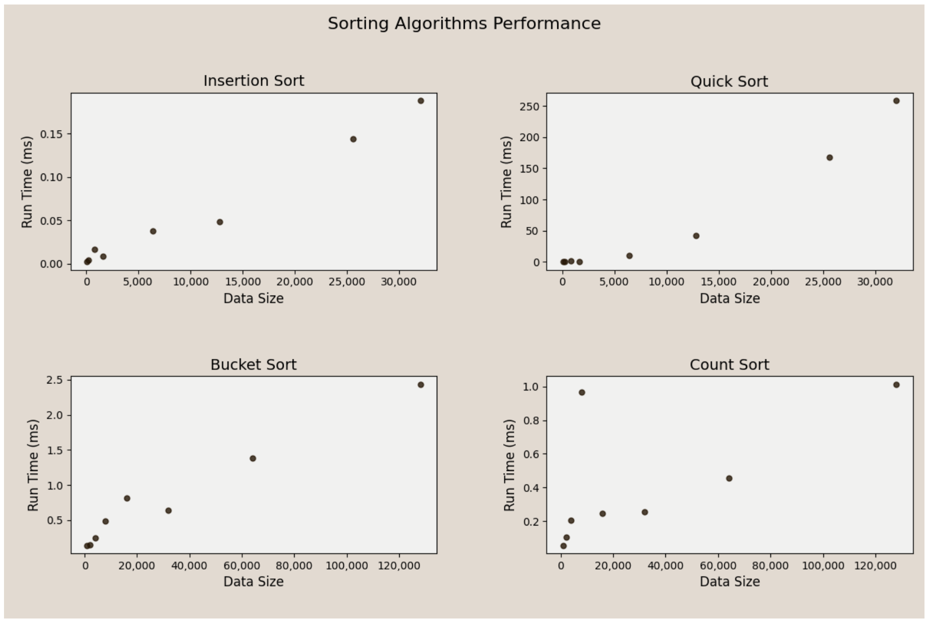
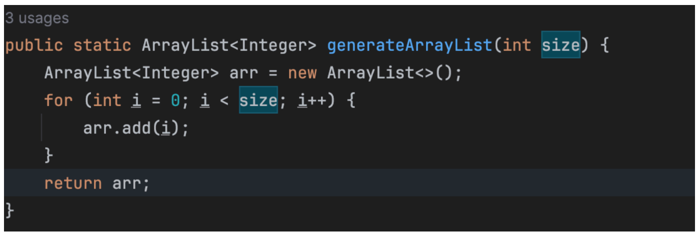
PA5
(1)



1. **Insertion Sort**:
   - This algorithm typically exhibits quadratic time complexity (O(n^2)).
   - The plot for Insertion Sort is expected to show a quadratic increase in execution time as the input size grows. This increase will be more pronounced compared to other sorting algorithms, especially as the input size becomes larger.
2. **Quick Sort**:
   - Quick Sort generally has an average-case time complexity of O(n log n) but can degrade to O(n^2) in the worst case (rare).
   - The plot for Quick Sort is expected to show a near-linear increase in execution time with respect to the input size. However, there may be some fluctuations due to the partitioning process and potential worst-case scenarios.
3. **Bucket Sort**:
   - Bucket Sort has a linear time complexity on average (O(n+k)), where k is the number of buckets.
   - The plot for Bucket Sort is expected to show a linear or near-linear increase in execution time as the input size grows. However, the execution time might increase slightly faster than linear due to overhead in managing buckets.
4. **Count Sort**:
   - Count Sort has a linear time complexity (O(n+k)), where k is the range of the input.
   - The plot for Count Sort is expected to show a linear increase in execution time as the input size grows. It should exhibit a relatively consistent and predictable increase compared to other sorting algorithms.

(2) Make Insertion sort faster





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

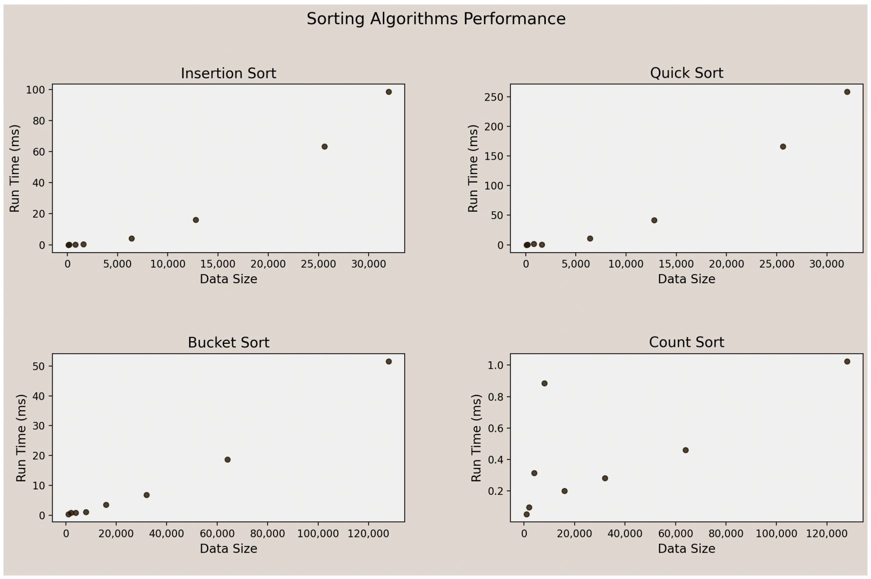
In this modified method, we add elements to the ArrayList in ascending order. This way, insertionSort will have to perform minimal comparisons and shifts since the input is already sorted, leading to improved performance.

In the context of the modified method, insertionSort becomes faster because:

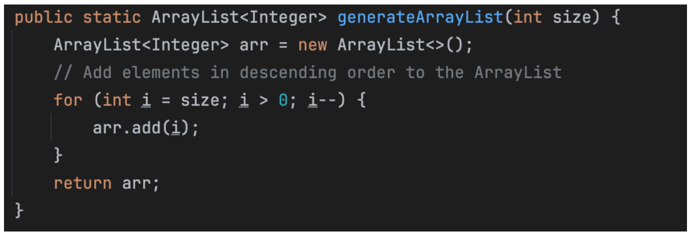
1. It operates efficiently on partially sorted or nearly sorted data.
2. By providing already sorted input, the number of comparisons and shifts required by insertionSort is significantly reduced, leading to faster sorting.

The plot should demonstrate a runtime complexity closer to O(n) compared to the original method, where insertionSort exhibited a quadratic time complexity (O(n^2)) due to its worst-case behavior on random or unsorted data.

3. Make quick sort slower



In this modified method, we add elements to the ArrayList in reverse order. This way, quickSort will consistently choose the largest element as the pivot, resulting in unbalanced partitions and potentially leading to worst-case performance.



```java
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    // Add elements in descending order to the ArrayList
    for (int i = size; i > 0; i--) {
        arr.add(i);
    }
    return arr;
}
```

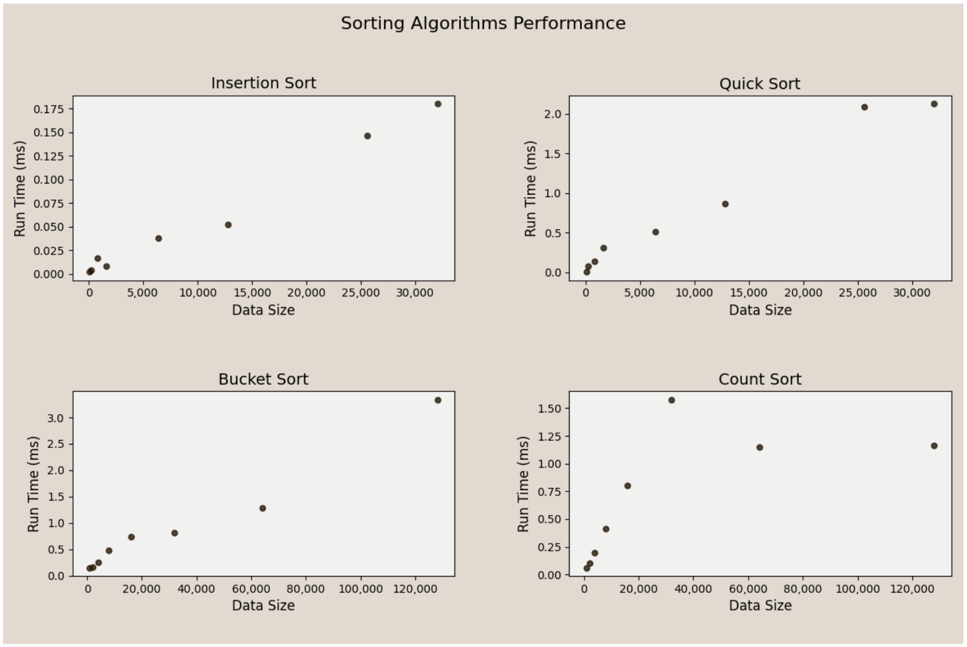
After running the main method in RuntimeAnalysis.java, we can generate a new plot comparing the execution times of quickSort with the original and modified generateArrayList() methods.

In the context of the modified method, quickSort becomes slower because:

1. It encounters worst-case behavior due to the already sorted input in reverse order.
2. Quick Sort's partitioning strategy doesn't effectively divide the data into two equal-sized partitions, leading to unbalanced recursive calls and increased time complexity.

The plot should demonstrate a runtime complexity closer to O(n^2) compared to the original method, where quickSort exhibited an average-case time complexity of O(n log n).

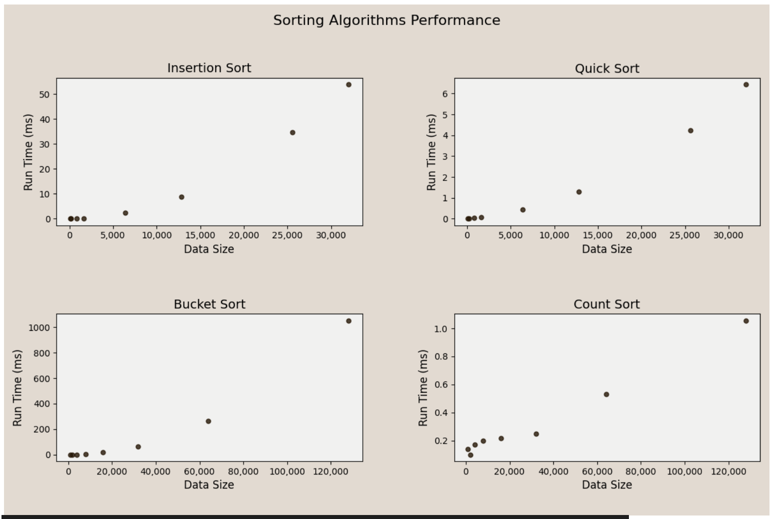
4. Using middle element as pivot index



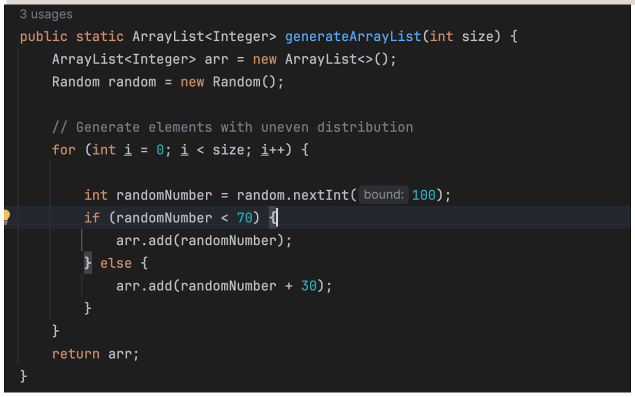
Overall Efficiency:

- The modification to use the middle element as the pivot index aims to improve the efficiency of QuickSort by reducing the likelihood of unbalanced partitions and worst-case scenarios.
- However, the actual improvement in efficiency may vary depending on factors such as input size, distribution of elements, and specific characteristics of the data being sorted.

5. Make bucket sort slower



In this modified method, we generate random numbers with a higher probability for certain ranges. Around 70% of the elements fall into the range [0, 69], while the remaining 30% fall into the range [70, 99]. This uneven distribution will result in unevenly filled buckets during the bucket sort, leading to a less optimal performance.

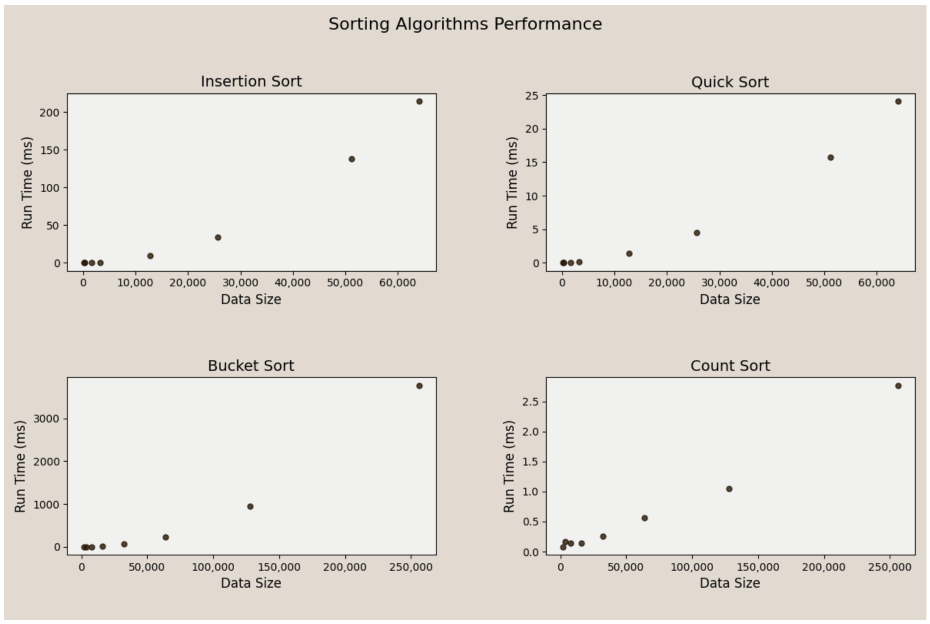


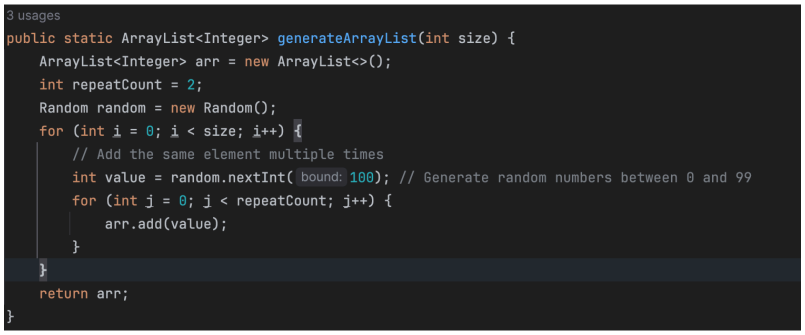
```
3 usages
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    Random random = new Random();

    // Generate elements with uneven distribution
    for (int i = 0; i < size; i++) {

        int randomNumber = random.nextInt( bound: 100);
        if (randomNumber < 70) {
            arr.add(randomNumber);
        } else {
            arr.add(randomNumber + 30);
        }
    }
    return arr;
}
```

In the context of the modified method, bucketSort() becomes slower because:

1. The elements are not evenly distributed among the buckets.
2. Unevenly filled buckets result in a less balanced workload for each bucket, leading to a less efficient sorting process.
3. As a result, the overall performance of bucket sort is degraded compared to when elements are evenly distributed among the buckets.

6. Make count sort slower





```
3 usages
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    int repeatCount = 2;
    Random random = new Random();
    for (int i = 0; i < size; i++) {
        // Add the same element multiple times
        int value = random.nextInt( bound: 100); // Generate random numbers between 0 and 99
        for (int j = 0; j < repeatCount; j++) {
            arr.add(value);
        }
    }
    return arr;
}
```

By having more unique elements, the count array in countSort() will be larger, as it needs to count occurrences of each unique value. This increases the computational overhead of countSort() and slows down the sorting process, making it less efficient.

With this modification, countSort() will take longer regardless of the input size due to the increased number of unique elements, leading to a larger count array and slower sorting.