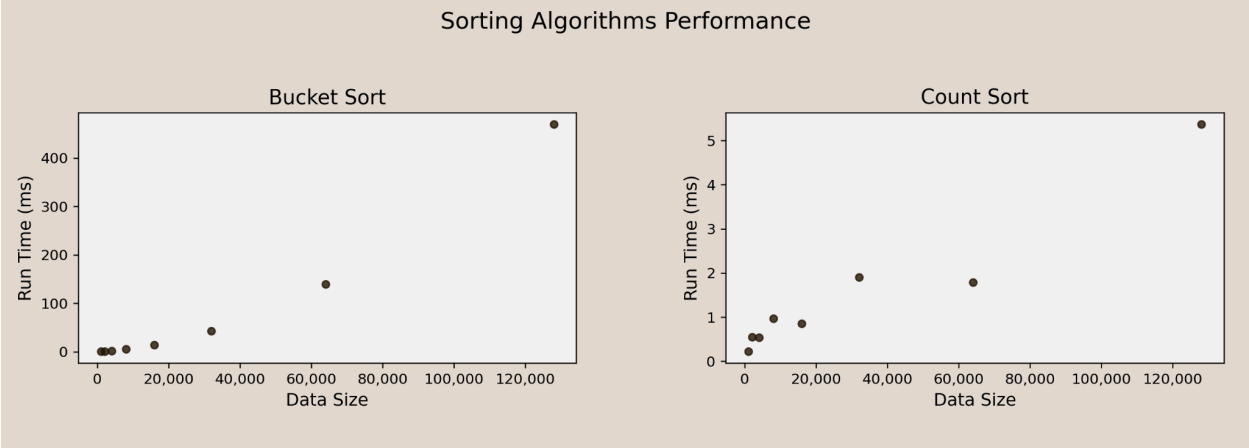
Rikako Ono
A17583780
8/23/24

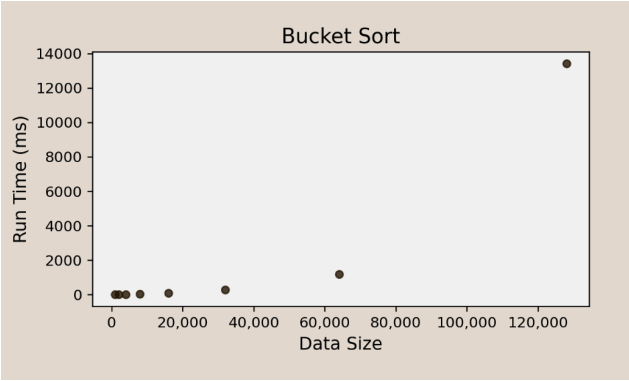
pa3_writeup

1. Original Speed



The best case scenario for bucket sort is O(n), where each bucket sorts values in an efficient way; the worst case scenario is O(n^2), where all elements fall in one bucket and have to be sorted afterwards. This can be seen in the graph where the slope gradually increases as the data size increases (without the range increasing).
For count sort, the base case time complexity is O(n), for counting each of the elements for the range of inputs. The worst case is O(2n) where the range of values is the same as n, but since constant coefficients are ignored for complexity the best and worst case is O(n). This can be seen in the graph where the trendline seems generally positively linear as the data size increases.

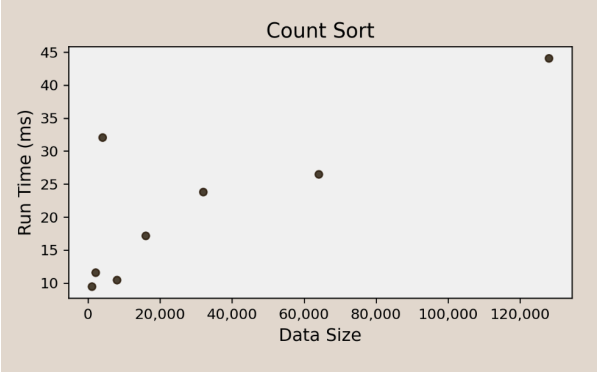
2. Making Bucket Sort Slower



```java
public static ArrayList<Integer> generateArrayList(int size) {  3 usages  rikakoono *
    ArrayList<Integer> arr = new ArrayList<>();
    for (int i = 0; i < size; i++) {
        arr.add((int) (Math.random() * 10));  // Limits the range, increasing bucket load.
    }
    return arr;
}
```

Instead of the original 1000 integers, I limited the random numbers to fall between 0-9 using (Math.random() * 10). This causes most elements to fall into just a few buckets, losing the advantage of being able to sort elements in a parallel way across multiple buckets. When a majority of elements end up in a few buckets, each of these require additional sorting and increases overall runtime compared to processing many buckets in parallel.

3. Making Count Sort Slower




```java
public static ArrayList<Integer> generateArrayList(int size) {  3 usages  rikakoono *
    ArrayList<Integer> arr = new ArrayList<>();
    for (int i = 0; i < size; i++) {
        arr.add((int) (Math.random() * 10000000));
    }
    return arr;
}
```

As aforementioned, countSort is reliant on the range of values for speed; this is because countSort relies on counting occurrences of each distinct value within a range, so the larger the range, the larger the counting array (count[]). By increasing the range of values, countSort must count occurrences of a larger range of values and therefore takes longer than the original range of 0-999.