Lin Tian

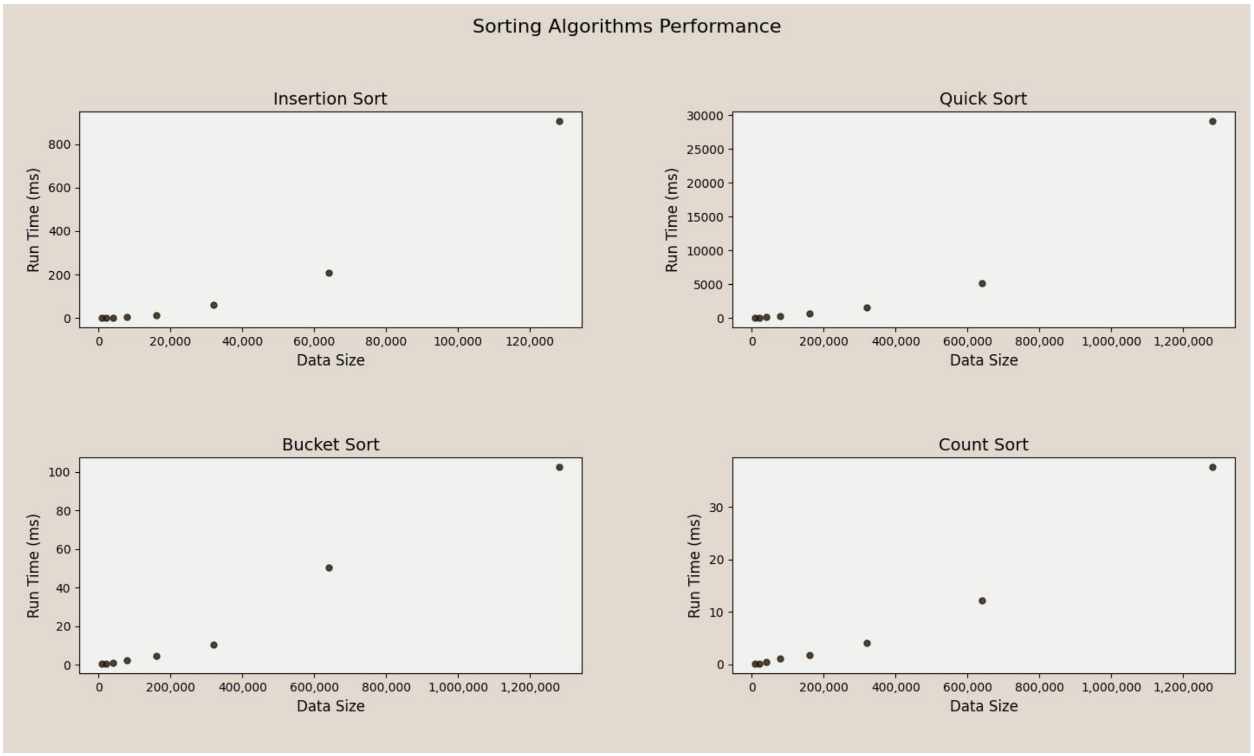
DSC 30

Marina Langlois

Nov 5th, 2023

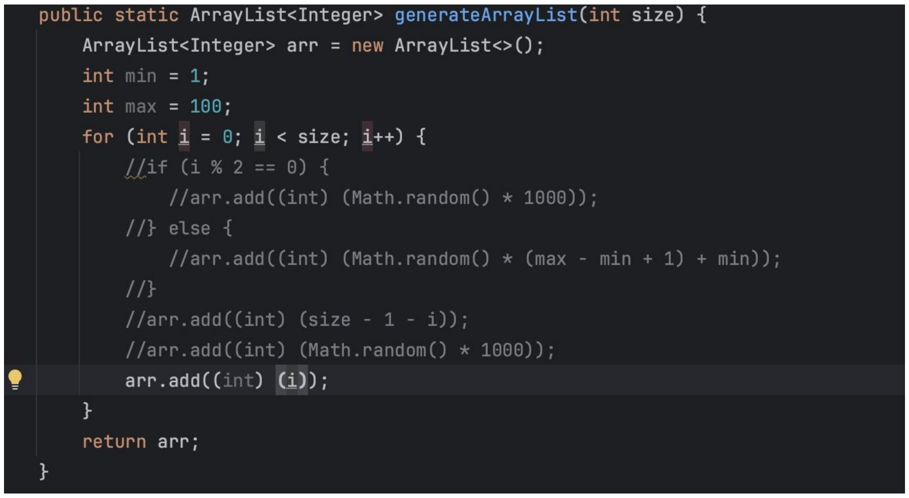
PA5(Part 2.2)

(1)



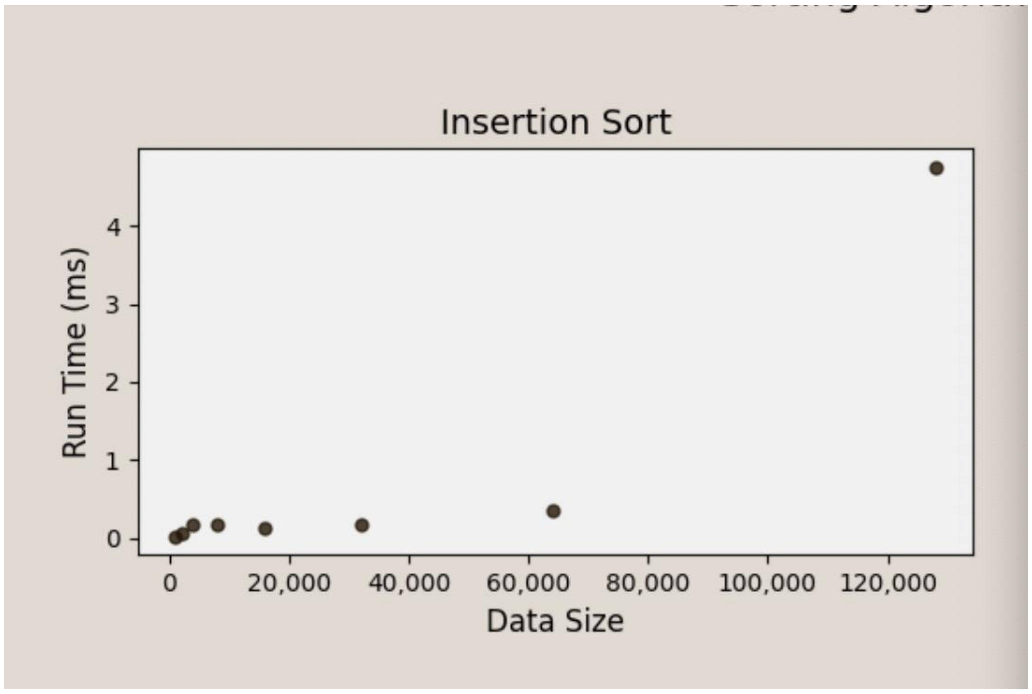
According to the graphs above, all sorting methods show a positively proportional relationship between the Data Size and the Run Time, which is the larger the Data Size, the more the Run Time. Noticeably, Insertion Sort was run on smaller sized input data because it may take too long for Insertion Sort to go through larger data sets like the other sorting methods. On the same level of data size, Count Sort appears to be the fastest that uses the least time among the others while Insertion Sort is the slowest that uses the most time among the others. Then Quicker sort is a little faster than Insertion Sort and Bucket Sort is faster than Quick Sort. The results make sense considering the average time complexity of each sorting methods: O(n) for Count Sort, O(n) - O(n^2) for Bucket Sort, O(n logn) for Quick Sort, and O(n^2) for Insertion Sort. For the Insertion Sort graph, the trend shows the function of $f(x) = x^2$ that matches with the empirical time complexity O(n^2). As for Quick Sort, the graph shows a trend of (n*logn). Both Buckets Sort and Count Sort shows a linear trend which correspond to the the empirical time complexity O(n). The big O time complexity parameters are empirical so it is not surprising that the actual run time differ a little from it, but the general order of each sorting methods in terms of running times matches with the empirical assumptions.

(2)



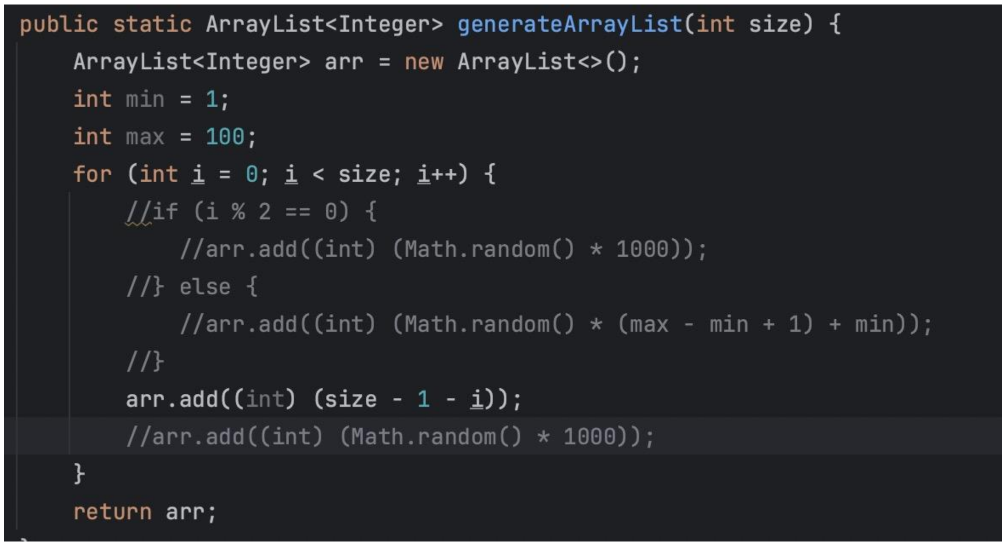
```java
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    int min = 1;
    int max = 100;
    for (int i = 0; i < size; i++) {
        //if (i % 2 == 0) {
            //arr.add((int) (Math.random() * 1000));
        //} else {
            //arr.add((int) (Math.random() * (max - min + 1) + min));
        //}
        //arr.add((int) (size - 1 - i));
        //arr.add((int) (Math.random() * 1000));
        arr.add((int) (i));
    }
    return arr;
}
```

For a faster general run time than the original method, I modified the passed-in list to be already sorted list that the Insertion Sort Method would reach the best case scenario where the time complexity is O(n), faster than the original method(O(n^2)).



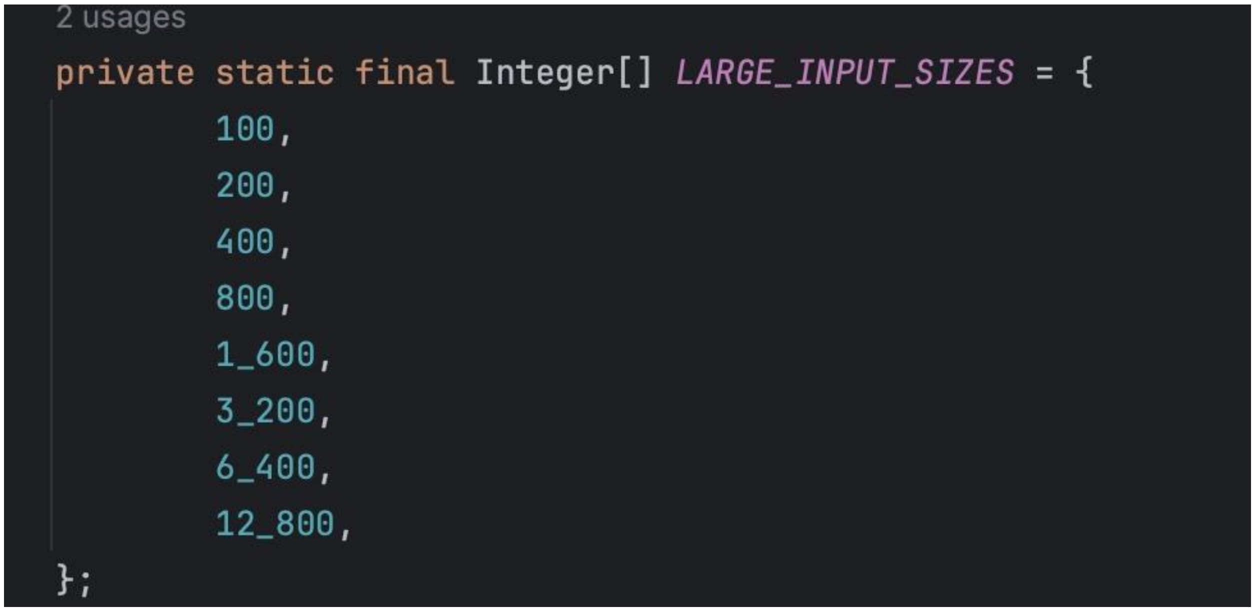
The graph is also telling us the same thing: Under the same data size(the observation around 60,000), this method(almost 0 ms) takes less run time than the original method(around 200 ms). Even though the trend doesn't look like a linear relationship, all observations are too small in terms of run time that could be all round down to almost 0 second, which would appears to be a linear relationship on a larger scaled graph.

(3)


```
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    int min = 1;
    int max = 100;
    for (int i = 0; i < size; i++) {
        //if (i % 2 == 0) {
            //arr.add((int) (Math.random() * 1000));
        //} else {
            //arr.add((int) (Math.random() * (max - min + 1) + min));
        //}
        arr.add((int) (size - 1 - i));
        //arr.add((int) (Math.random() * 1000));
    }
    return arr;
```

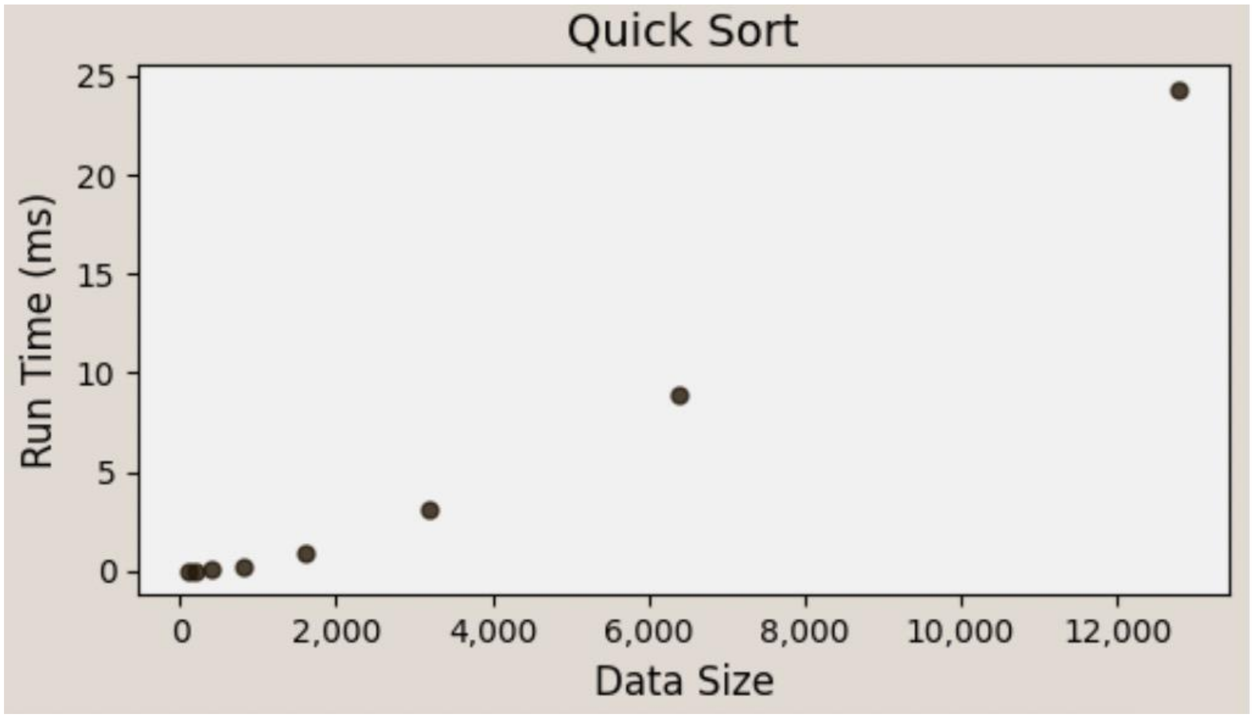
For a slower general run time than the original method, I modified the passed-in list to be already sorted list in an descending order that the Quick Sort Method would possibly reach the worst case scenario where the time complexity is O(n^2).

However, the input sizes were too big that java ran out of memories, so I also adjusted the sizes to be smaller.


```
2 usages
private static final Integer[] LARGE_INPUT_SIZES = {
        100,
        200,
        400,
        800,
        1_600,
        3_200,
        6_400,
        12_800,
};
```

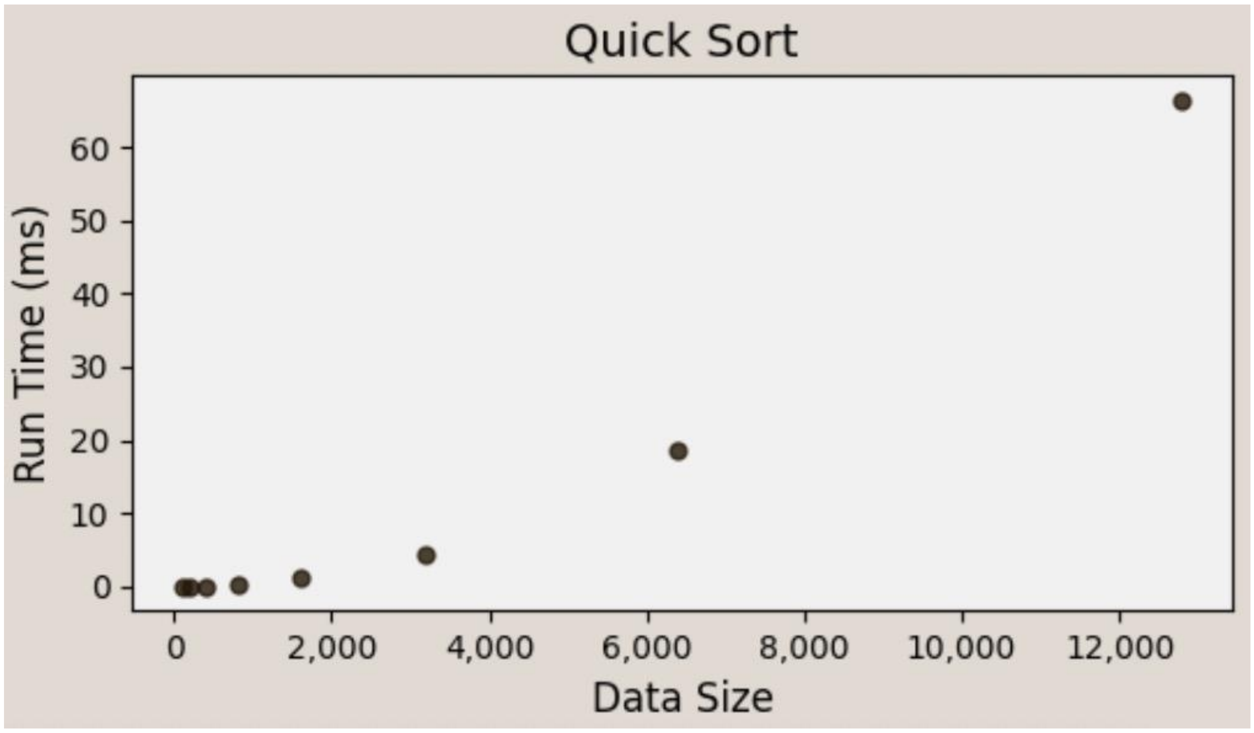
The graph of original method with the adjusted input

size:



The graph of the slower method with the adjusted input

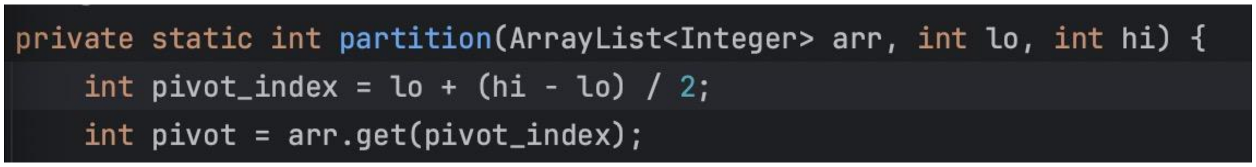
size:



The graph is also telling us the modified quick sort method is slower by showing the trend of the observations to be more like a function of f(x) = x^2 rather than f(x) = kx like shown in the original method. Under the same data size(take the observation around 6000), this method(almost 20 ms) takes more time than the original method(nearly 10 ms).

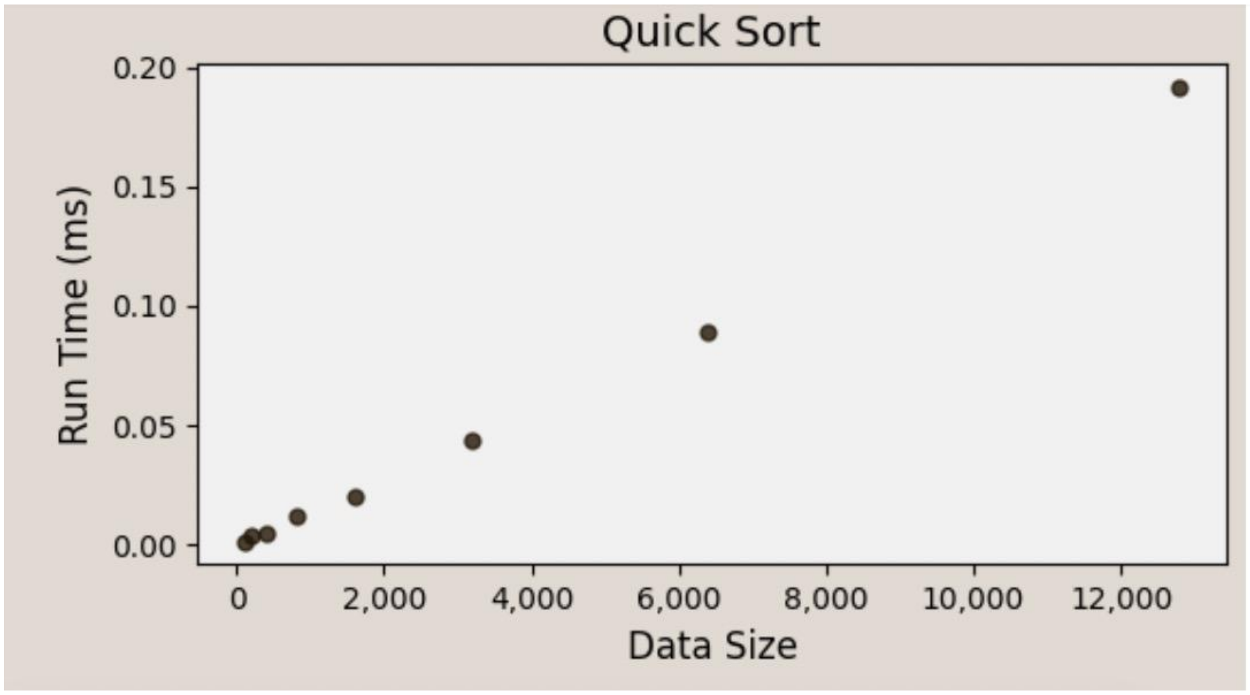
(4)

Using the same generateArrayList() I created in Task 3, I modified partition() in Sorts.java so that quickSort() uses the middle element as the pivot index rather than the start.



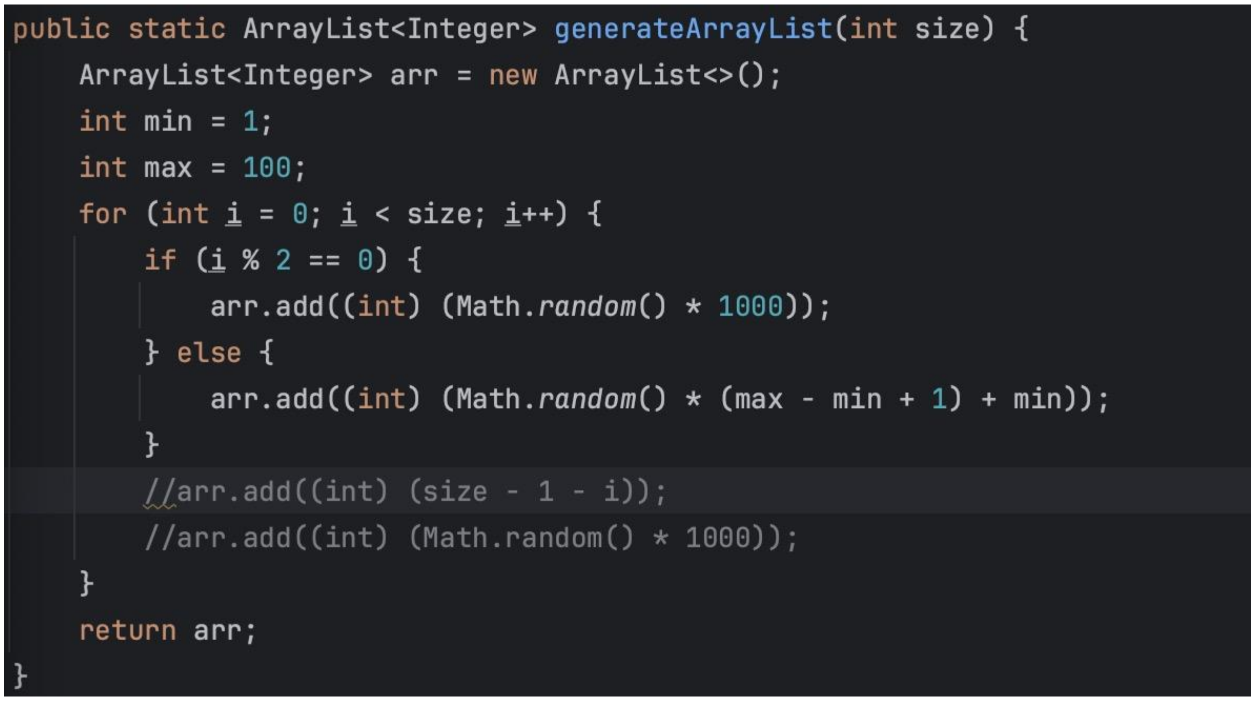
```
private static int partition(ArrayList<Integer> arr, int lo, int hi) {
    int pivot_index = lo + (hi - lo) / 2;
    int pivot = arr.get(pivot_index);
```

I created a new graph as shown below.



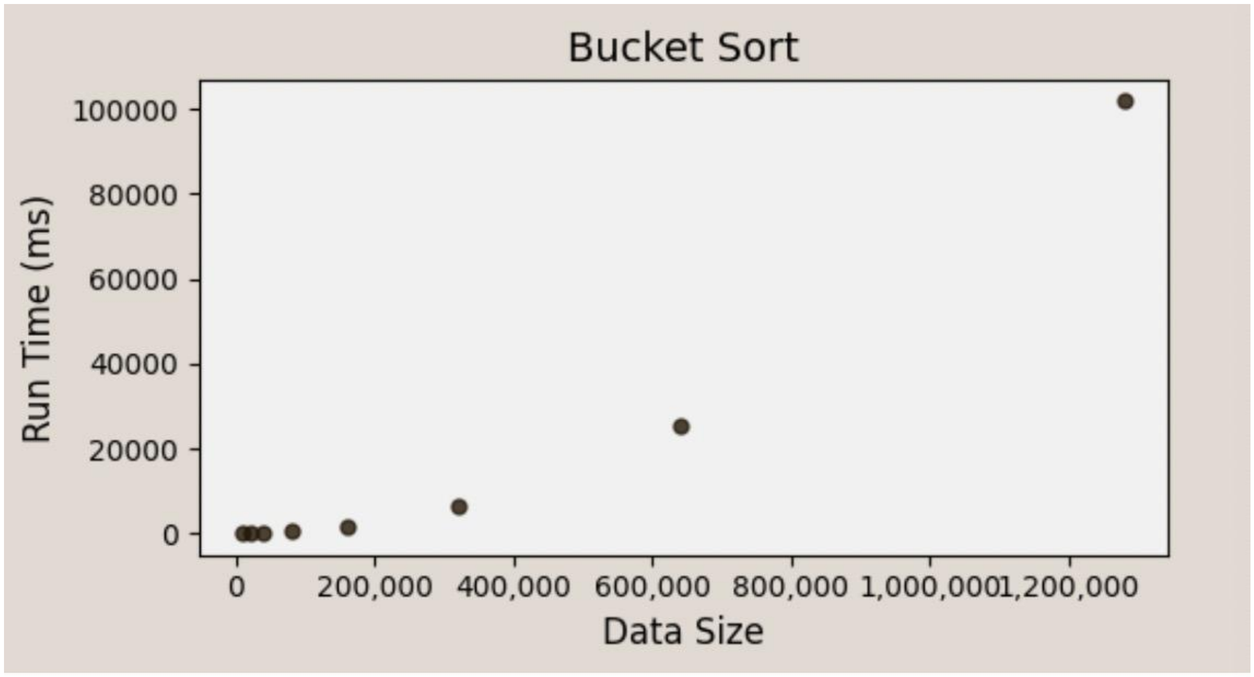
Comparing the new graph to the old graph, it seems like changing the pivot point of each partition from the first element to the middle element made the sorting process more efficient and cost less run time. The time complexity seems to be O(n) as you can tell from the graph. Under the same data size(take the observation around 6000) and input, this method(nearly 0.1 ms) takes far less time than the previous method(almost 20 ms).The efficiency of the sorting method got improved by switching the pivot point from the starting element to the middle one.

(5)



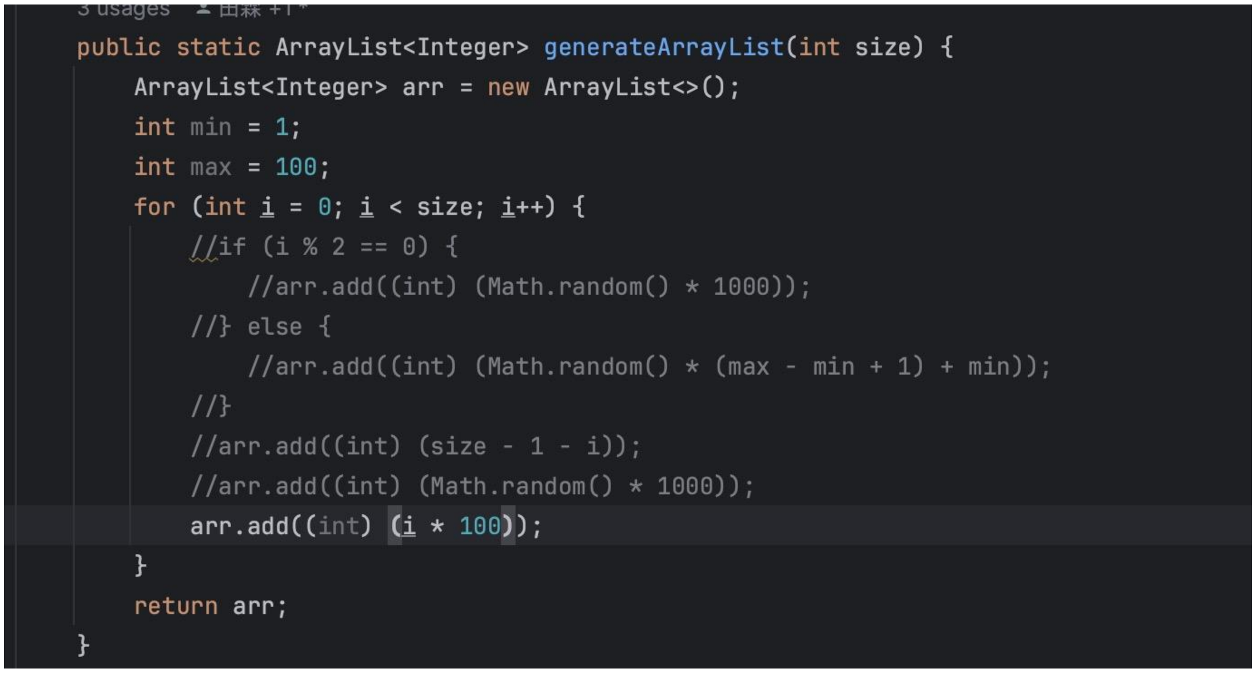
```java
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    int min = 1;
    int max = 100;
    for (int i = 0; i < size; i++) {
        if (i % 2 == 0) {
            arr.add((int) (Math.random() * 1000));
        } else {
            arr.add((int) (Math.random() * (max - min + 1) + min));
        }
        //arr.add((int) (size - 1 - i));
        //arr.add((int) (Math.random() * 1000));
    }
    return arr;
}
```

For a slower general run time than the original method, I modified the passed-in list to have a very uneven distribution of elements within the buckets that the Bucket Sort Method would possibly reach the worst case scenario. I managed to modify the passed in values : I assign a few elements with very high frequencies, maximizing the number of elements in a few buckets while leaving most buckets nearly empty. With this change in implementation, the Bucket Sort will just do Insertion Sort, which is slower.



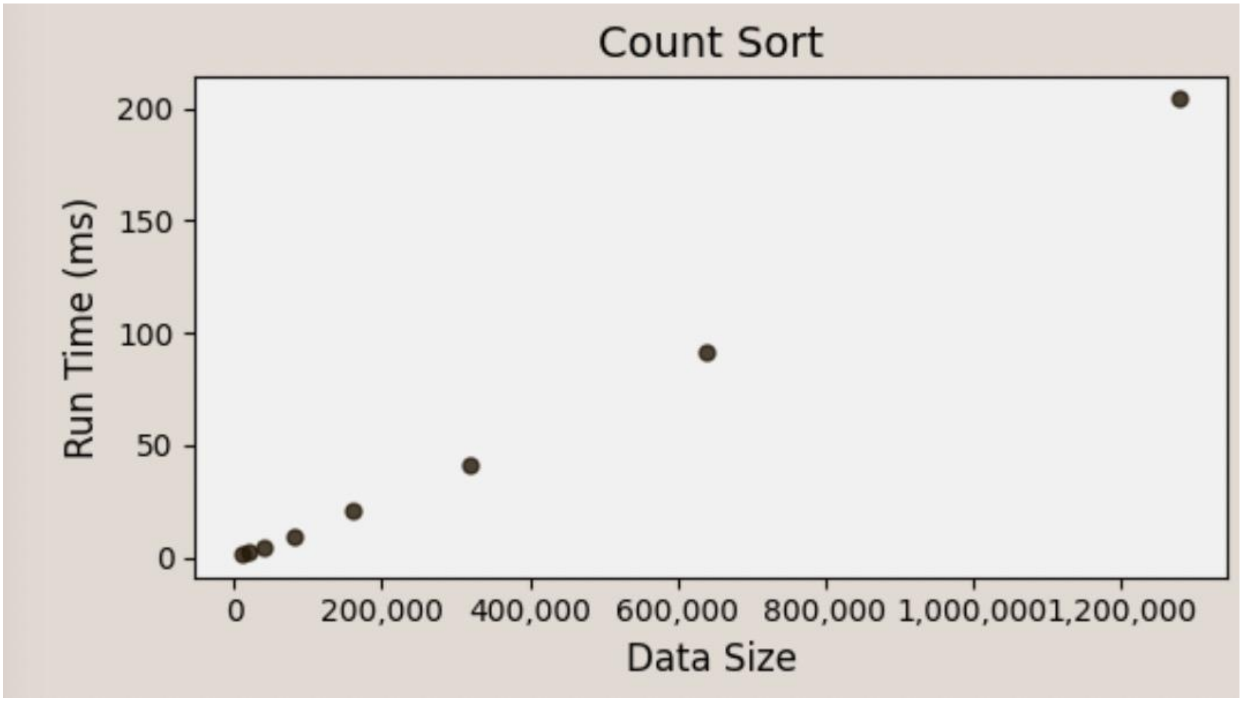
The graph is also telling us the modified Bucket Sort method is slower: Under the same data size(the observation around the data size of 600,000), this method(around 20000 ms) takes more time than the original method(around 50 ms).

(6)



```
3 usages   田森 +1 *
public static ArrayList<Integer> generateArrayList(int size) {
    ArrayList<Integer> arr = new ArrayList<>();
    int min = 1;
    int max = 100;
    for (int i = 0; i < size; i++) {
        //if (i % 2 == 0) {
            //arr.add((int) (Math.random() * 1000));
        //} else {
            //arr.add((int) (Math.random() * (max - min + 1) + min));
        //}
        //arr.add((int) (size - 1 - i));
        //arr.add((int) (Math.random() * 1000));
        arr.add((int) (i * 100));
    }
    return arr;
}
```

For a slower general run time than the original method, I modified the gap between the value of the elements in the passed-in list to be wider than the original list that the Count Sort Method would be slower since Counting Sort is not well-suited for sparse data. The count array will be unnecessarily large, and most of the entries will be empty.



The graph is also telling us the modified Count Sort method is slower: Under the same data size(the observation around the data size of 600,000), this method(almost 100 ms) takes more time than the original method(around 12 ms).