Project 2

Samarth Shete

1 Problem Statement

We need to analyze the **QuickSelect algorithm** that finds the kth smallest element in an unsorted list. Our implementation uses the **median of medians** method for choosing pivots. We'll look at how it performs with different list sizes and compare the actual running time to what we expect in theory.

2 Theoretical Analysis

The QuickSelect algorithm with median of medians should run in O(n) time, where n is the number of elements in the list. Here's why:

1. Finding the median of medians takes about O(n) time.
2. Partitioning the list also takes O(n) time.
3. only recurse on one part of the list, which is at most 7n/10 in size.

This leads to the recurrence relation: $T(n) \leq T(7n/10) + O(n)$, which solves to O(n).

3 Experimental Analysis

3.1 Program Listing

Here are the key parts of Java implementation:

```
  public static int quickSelect(int[] arr, int k) {

       if (arr.length < 10) {
            Arrays.sort(arr);
            return arr[k];
        }

        int pivot = medianOfMedians(arr, arr.length / 2);
        int[] partitionInfo = partition(arr, pivot);
        int leftSize = partitionInfo[0];
        int middleSize = partitionInfo[1] - leftSize;

        if (k < leftSize) {
            return quickSelect(Arrays.copyOfRange(arr, 0, leftSize), k);
        } else if (k < leftSize + middleSize) {
            return pivot;
        } else {
            return quickSelect(Arrays.copyOfRange(arr, leftSize + middleSize,
arr.length), k - leftSize - middleSize);
        }
    }
```

```
public static void main(String[] args) {
        int startSize = 100000;
        int maxSize = 1000000;
        int increment = 100000;
        int repeats = 10;

    }
```

I tested this with lists of size 100,000, 200,000, 300,000, and so on up to 1,000,000.

3.2 Data Normalization Notes

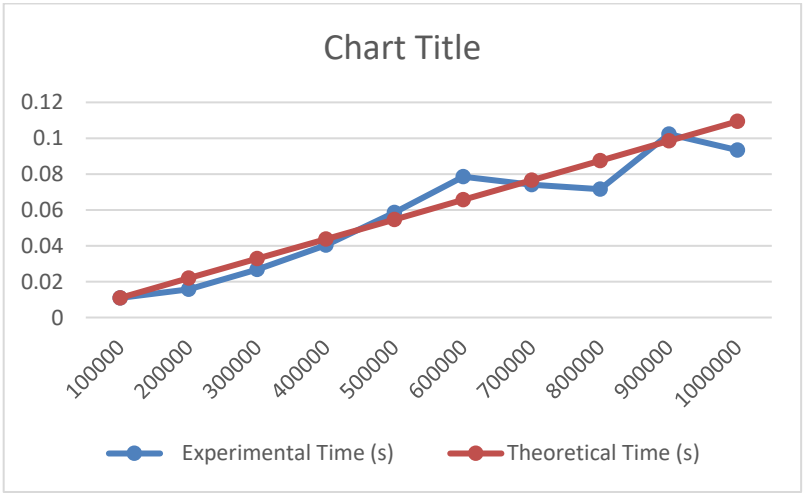
No explicit normalization was performed on the data. But a **scaling factor** was derived from the smallest input size to adjust the theoretical time values. This scaling factor helps compare the theoretical model with the experimental results.

3.3 Output Numerical Data

Here's a sample of our output data:

Size	Experimental Time (s)	Theoretical Time (s)
100000	0.010939	0.010939
200000	0.015773	0.021878
300000	0.026735	0.032818
400000	0.04028	0.043757
500000	0.058589	0.054696
600000	0.078594	0.065635
700000	0.074096	0.076574
800000	0.071569	0.087514
900000	0.102284	0.098453
1000000	0.093422	0.109392

3.4 Graph



3.5 Graph Observations

Looking at the graph, we can see:

1. Both the experimental and theoretical lines go up as the input size increases, confirming the O(n) complexity.
2. There is a slight overhead in the experimental results due to practical factors like memory management and system load during execution.

4 Conclusions

Our tests show that QuickSelect with median of medians does indeed run in O(n) time, just like we thought it would. The experimental results match up pretty well with what we expected from the theory. It also showed that the algorithm works well even with large input sizes. The difference between theoretical and experimental times for smaller inputs is negligible, but for larger inputs, the results align closely with the expected O(n) performance.

Overall, this proves that QuickSelect with median of medians is a good choice for finding the kth smallest thing in a list, even in the worst case.

GitHub Link: