CS 3310 Design and Analysis of Algorithms
Project #2

Abstract

This project asks to implement, test, and analyze the three different Selection Problem algorithms that find the kth smallest element in the list of n elements. The class discussed three algorithms: Merge sort, Quick sort, and Median of Medians of algorithms.

- Algorithm 1, Merge Sort, sorts the list and then returns the kth smallest element.
- Algorithm 2, Quick Sort, applies the procedure Partition used in Quicksort. The procedure partitions pick a pivot value in an array and make a partition based on the pivot value. All elements smaller than some pivot item come before it in the array. Similarly, all elements larger than that pivot item go after it. The second algorithm recursively calls itself by partitioning until the pivot index = kth index.
- Algorithm 3, Median of Medians, applies the Partition algorithm by making sub-1D arrays the size of any odd number. It sorts each variety and makes a new array with all the median values in arrays.

Part 1: Design & Theoretical Analysis (30 points)

a. Complete the following table for theoretical worst-case complexity of each algorithm. Also need to describe how the worst-case input of each algorithm should be.

Algorithm	theoretical worst-case complexity	describe the worst-case input
Algorithm 1	O(n log n)	O(n log n)
Algorithm 2	O(n)	O(n^2)
Algorithm 3	O(n)	O(n)

b. Design the program by providing pseudocode or flowchart for each sorting algorithm.

2. Algorithm> Project2 > Documentation > DesignDocs

Part 2: Implementation (35 points)

a. Code each program based on the design (pseudocode or flow chart) given in Part 1(b).

b. Test you program using the designed testing input data given in the table in Part 1(c), Make sure each program generates the correct answer by marking a "√" if it is correct for each testing case for each program column in the table. Repeat the process of debugging if necessary.

The test data execution sample run in main

c. For each program, capture a screen shot of the execution (Compile&Run) using one testing case to show how this program works properly

Part 3: Comparative Analysis (35 points)

a. Run each program with the designed randomly generated input data given in Part 1(d). Generate a table for all the experimental results as follows.

millisecond second (ms) (second = 1000 ms)

Problem size	Algorithm1 (Merge)	Algorithm2 (Quick)	Algorithm3 (MM)
2	0	0	0
4	0	0	0
8	0	0	0
16	0	0	0.020408163
32	0	0.020408163	0
64	0	0	0
128	0	0	0
256	0.020408163	0	0
512	0.040816327	0	0.040816327
1024	0.204081633	0	0.020408163
2048	0.163265306	0	0.040816327
4096	0.367346939	0.020408163	0.06122449
8192	0.653061224	0	0.224489796
16384	1.428571429	0.040816327	0.510204082
32768	3.040816327	0.020408163	0.87755102
65536	5.959183673	0.081632653	1.510204082
131072	12.2244898	0.12244898	2.734693878
262144	25.67346939	0.244897959	5.244897959
524288	56.57142857	0.632653061	14.69387755
1048576	199.4489796	2.408163265	50.16326531
2097152	544.3469388	6.285714286	140.2653061
4194304	1037.795918	9.734693878	267.6734694
8388608	2295.183673	25.08163265	697.122449

b. Plot a graph of each algorithm and summarize the performance of each algorithm based on its own graph.

<Insert - totally three graphs, one for each program, here>

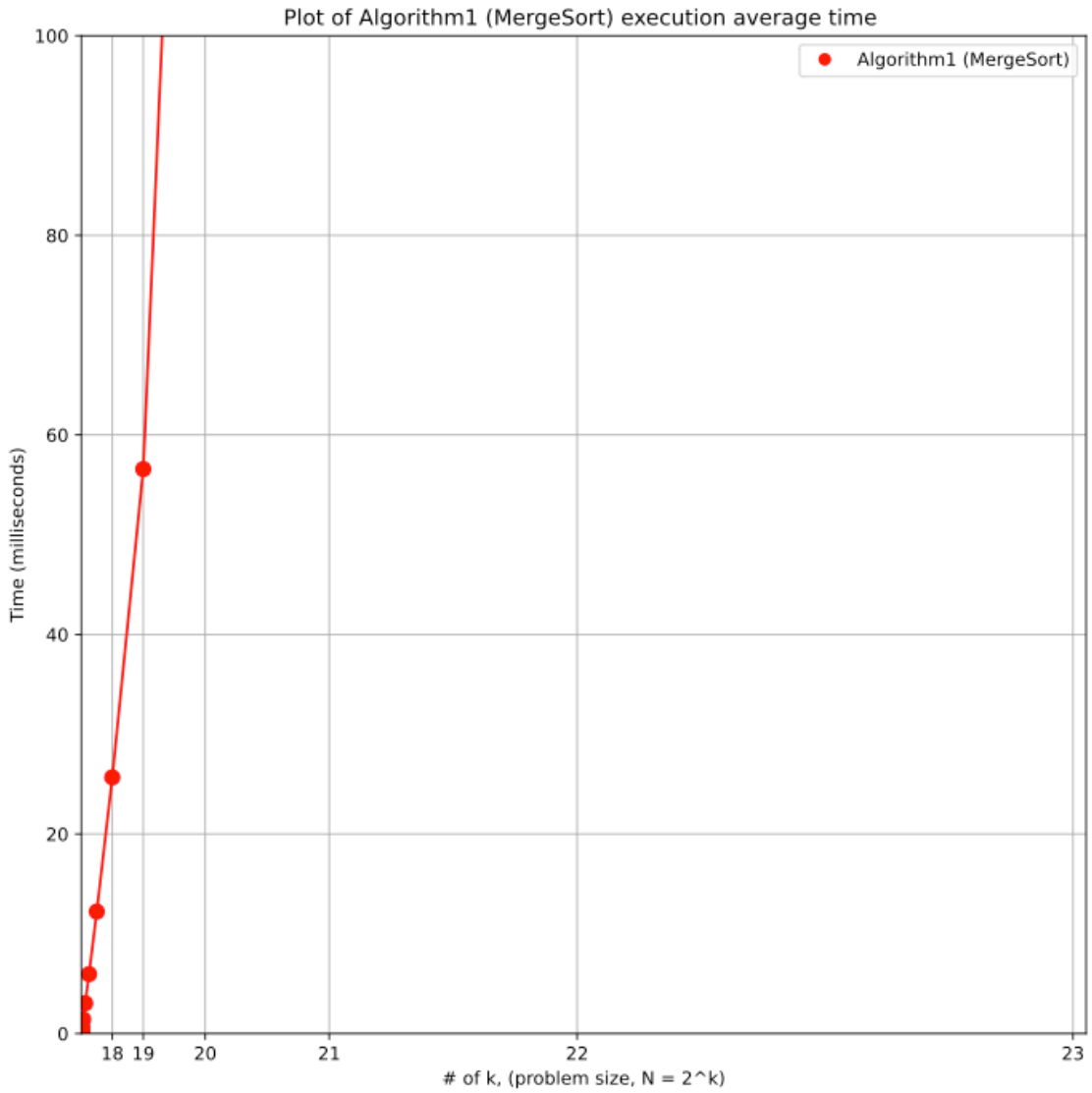
See the following pages.

1. Algorithm 1 (Merge Sort search)

   The plot shows the merge sort execution time grows linearly. The reason may be considered as follows.

   - The algorithm every time divides half and sorts the original array.
   - Since my implementation doesn't include finding the pivot value and partitioning the array, it will not be affected by the pivot value quality.
   - The merge sort does not require a swap, and it will not access two long-distanced memory locations to swap value.

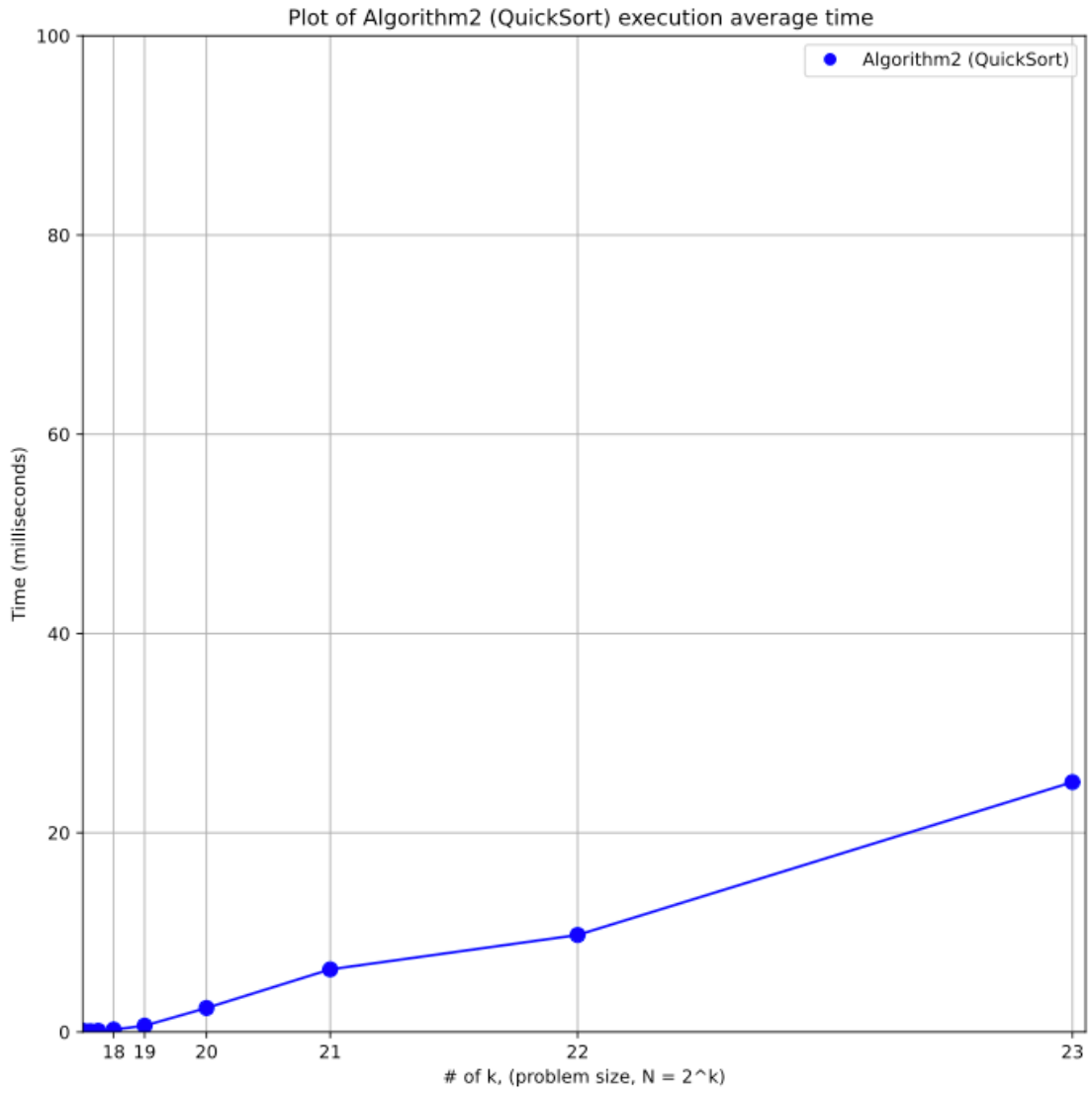
   These facts follow the execution time will be stable without the result of pivot value and long-distanced memory access to swap.



2. Algorithm 2 (Quick Sort)
Generally speaking, the quick sort algorithm is one of the most efficient algorithms. The plot shows this algorithm is fastest in the all input size in this project. The potential reasons are as follows.

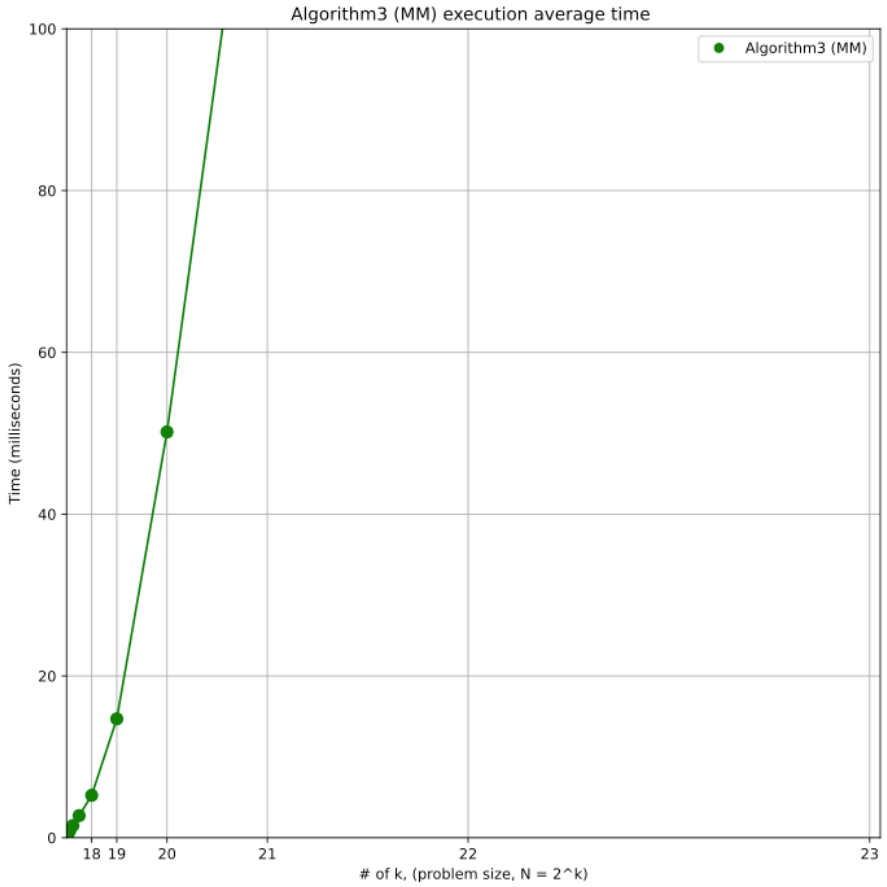
- My implementation quick sort takes 3 samples in the array to find a slightly better pivot, which is a smaller computation performance compared to how the median of medians finds a pivot.
- When it partitions the array with pivot, it directly swaps the array to sort, while the merge sort compares two values in the separated array and stores the result to the new array. It costs extra space.

As the plot shows, the execution time grows gradually. It may be considered accessing two long-distanced memory locations to swap values and make a partition. However, it does not affect the result significantly.

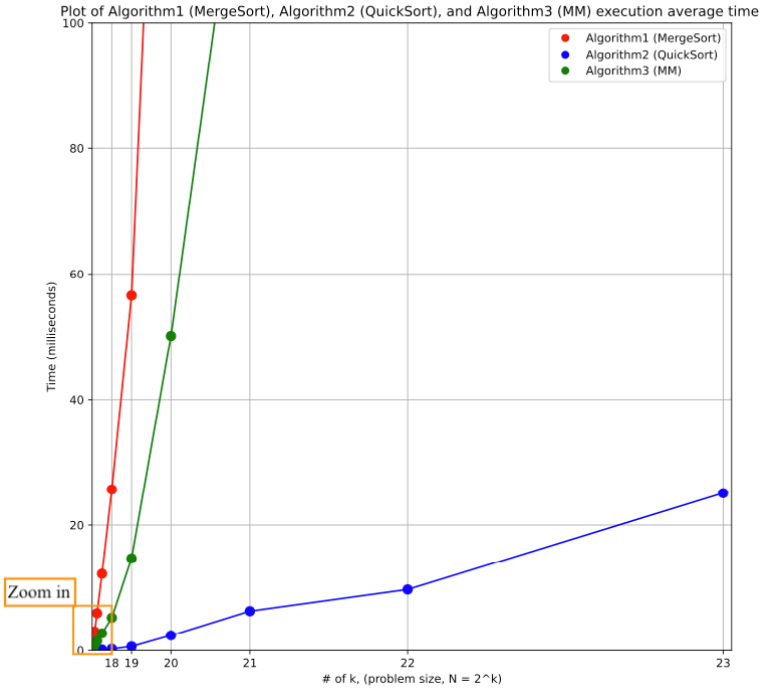


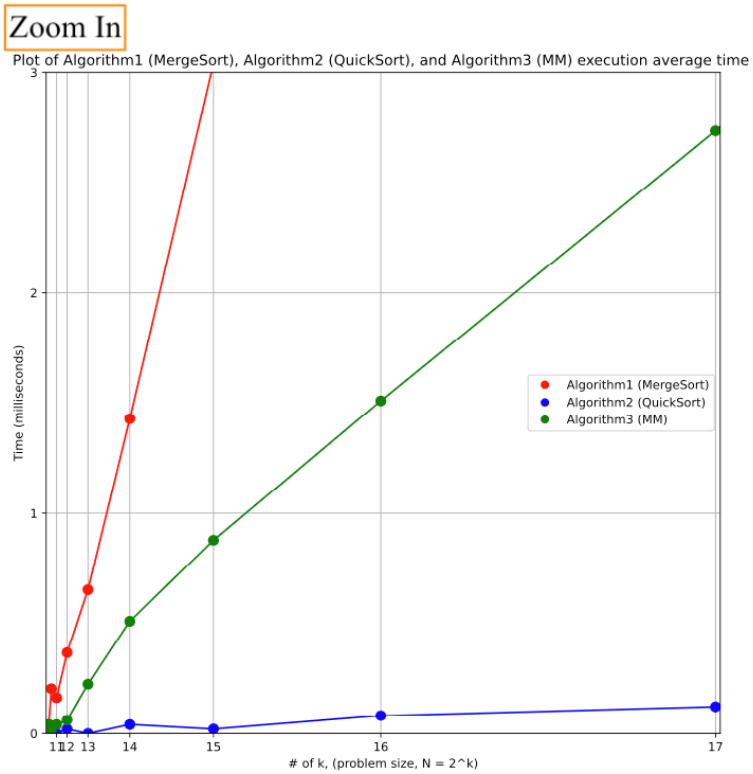
3. Algorithm 3 (Median of Medians)
The plot shows this algorithm's execution time is the 2nd fastest. The potential reasons are as follows.
    - Even though the worst-case time complexity is O(n), it may have a larger constant factor.
    - The constant factor in the linear term can be larger due to the overhead of these sub-tasks.
        i. Repeatedly finding medians of smaller subgroups.
        ii. Calculating pivots.
        iii. Finding pivot index.
        iv. The array is partitioned with pivot value.
        v. Managing the recursive structure.
    - During recursive calls, the cash miss likely occurs especially getting larger input size.
    - At the same time, my implementation makes a copy of medians and stores them in another array. It costs extra space.

    In the analysis phase, we can find median of medians worst and best time complexity is O(n). I expected this algorithm is fastest. But the larger constant factor, overhead of sub-tasks, potential cache misses, and extra space usage collectively costs more execution time.



Plot all three graphs on the same graph and compare the performance of all three algorithms. Explain the reasons for which algorithm is faster than others.





<Insert - about explaining the results>

As I state above, we can interpretate each fact as follow.

- Observation 1: Merge sort VS Quick sort
  - The quick sort operates in-place, directly manipulating the original array through swaps without requiring additional array space. In contrast, Merge Sort creates temporary arrays during the splitting and merging phases until it reaches the base case, contributing to additional space complexity.

- Observation 2: Merge Sort VS Median of Medians
  - As we derived the theriacal time complexity in the class, it is understandable Median of median is faster than Merge Sort. Although the finding pivot computation in the Median of Medians is not stable, the plot shows the Median of Medians always faster in this project.
  - Merge sort is stable because it does not need to find pivot. Median of Medians change execution time depends on how quickly it picks a good pivot.
  - During the recursive call in Median of Medians, it likely causes the cache miss. However, the median of medians still holds faster execution time.

- Observation 3: Quick sort vs Median of Medians
  - In the analysis phase, we can find median of medians worst and best time complexity is O(n) and the quick sort worst case complexity is O(n^2) under certain condition. However, the plot shows the quick sort algorithm is fastest, ever though it has the worst time complexity among three algorithms. It is important to note that the execution time is vary depends on environment and inputs.
  - Compared to the quick sort, the median of medians costs the larger constant factor, overhead of sub-tasks, potential cache misses, and extra space usage collectively costs more execution time.

c. Compare the theoretical results in Part 1(a) and empirical results here. Explain the possible factors that cause the difference.

<Insert - report the findings and explain>

In the analysis phase, we can find the median of medians worst and best time complexity is O(n). It is natural to expect this algorithm to be the fastest. But the larger constant factor, overhead of sub-tasks, potential cache misses, and extra space usage collectively cost more execution time. Managing the recursive structure can cause the cash miss, especially getting a larger input size. This project and observation encourage students to consider not only theoretical complexity, but also practical factors such as constant factors, overhead, and memory management in algorithm design.

d. Give a spec of your computing environment, e.g. computer model, OS, hardware/software info, processor model and speed, memory size, ...

<Insert - spec of your computing environment>

- Model Name: Mac mini
- Mac OS Sonoma 14.0 (23A344)
- Chip: Apple M1
- Total Number of Cores: 8 (4 performance and 4 efficiency)
- Memory: 16 GB

e. Conclude your report with the strength and constraints of your work. At least 200 words. Note: It is reflection of this project. Ask yourself if you have a chance to re-do this project again, what you will do differently (e.g. your computing environment, hardware/software, programing language, data structure, data set generation, …) in order to design a better performance evaluation experiment.

<Insert - write a conclusion about strength and constraints of your work here.>

d. This project is another fun project in this class. Through this project, I learned how important it is to include practical portions I discovered in the project to sophisticated and efficient algorithms.

The one thing I can improve for the future project is designing simpler algorithms. I implemented two different partition methods for the quick sort and the median of medians. In the first version, I implemented taking 3 samples in the value to find a slightly better pivot value. The second one costs more swap execution time but is simpler. I compare each partition method, and the first one has a faster execution time. However, it is a complex algorithm and takes a lot of time to debug issues.

Usually, a project needs to be maintained and developed in the future. It is a better method to understand and customize easily than I can only understand complex algorithms. As I went through this project, I learned that neutral decision-making is important. I spent a lot of time implementing the first implementation, and the result of the execution was better than the second one. However, for the time and future development, it is better to adopt a simple and sustainable one.