Project 2: Sorting Algorithms

Basira Daqiq

Study Objective

This study aims to compare and contrast the performance of various sorting algorithms—Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort (with random pivot), and Quicksort with Median of Medians—across different input sizes and edge cases: best, worst, and average cases. Although exact runtimes may vary between executions on the same machine and across machines due to system variations, the overall trends should remain consistent as they depend on the underlying logic of the algorithms.

Methodology

To evaluate performance differences:

1. **Input Sizes**: Arrays ranging from 10,000 to 150,000 elements were tested in increments of 10,000. Significant performance differences were observed for arrays larger than a few thousand elements.
2. **Edge Cases**: Performance was measured for three input types to represent best, worst, and average cases.

Input Type 1: Sorted Arrays

Sorted arrays, by definition, require minimal or no sorting. Bubble Sort and Insertion Sort, in particular, can detect that the array is already sorted, which dramatically reduces the number of subsequent steps they take. The graph below shows the execution time vs size of the array.

Key Observations:

- **Selection Sort** consistently exhibits the highest runtimes due to its O(n^2) complexity.

- **Bubble Sort and Insertion Sort** perform significantly better in this case, as they leverage their ability to detect sorted arrays. This property allows them to achieve linear-time performance (O(n)) by determining early on that the array is sorted.
- In the graph, the runtime for Bubble Sort and Insertion Sort appears as a flat line zero, this is most likely because the run time is measured in milliseconds and the number and the actual run time is very small where it is rounded to zero in the selected units. Zooming in on the plot (left) reveals that these algorithms outperform all others on sorted inputs.

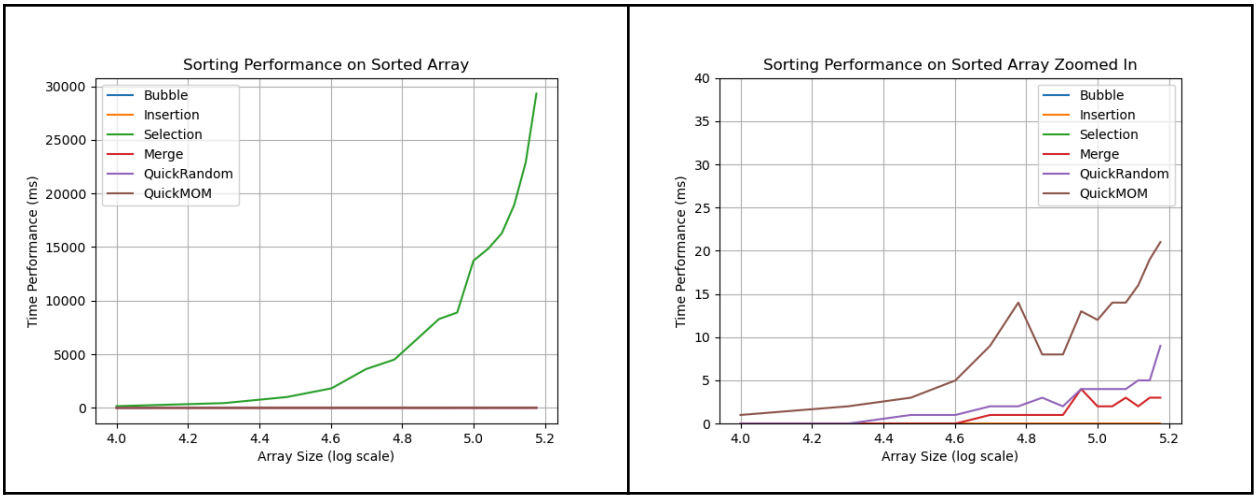


Figure 1. Execution time(ms) of sorting algorithm on already sorted arrays vs array size- log scale.

Table1: Size and time(ms) performance of sorting algorithms in sorted arrays

Size	Bubble	Insertion	Selection	Merge	QuickRandom	QuickMOM
10000	0	0	161	0	0	1
20000	0	0	445	0	0	2
30000	0	0	1014	0	1	3
40000	0	0	1822	0	1	5
50000	0	0	3622	1	2	9

Size	Bubble	Insertion	Selection	Merge	QuickRandom	QuickMOM
60000	0	0	4504	1	2	14
70000	0	0	6523	1	3	8
80000	0	0	8286	1	2	8
90000	0	0	8894	4	4	13
100000	0	0	13739	2	4	12
110000	0	0	14859	2	4	14
120000	0	0	16288	3	4	14
130000	0	0	18934	2	5	16
140000	0	0	22891	3	5	19
150000	0	0	29315	3	9	21

Input Type 2: Reverse Sorted Arrays

Reverse sorted arrays test the worst-case scenarios for many algorithms, requiring the maximum amount of work to sort the data. Most algorithms follow the same logic as usual, but Quick Sort faces its worst-case performance when the pivot is selected as either the smallest or largest element of the current array. The minimum of a partition was intentionally selected as the pivot element while sorting the reverse sorted array for quicksort.

In these scenarios:

- **Selection Sort** performs significantly worse than other algorithms consistent to its O(n^2) time complexity.
- Other algorithms also experience increased runtimes, but the specific impact depends on their implementation and handling of the input order.
- Merge sort out performs all other algorithms

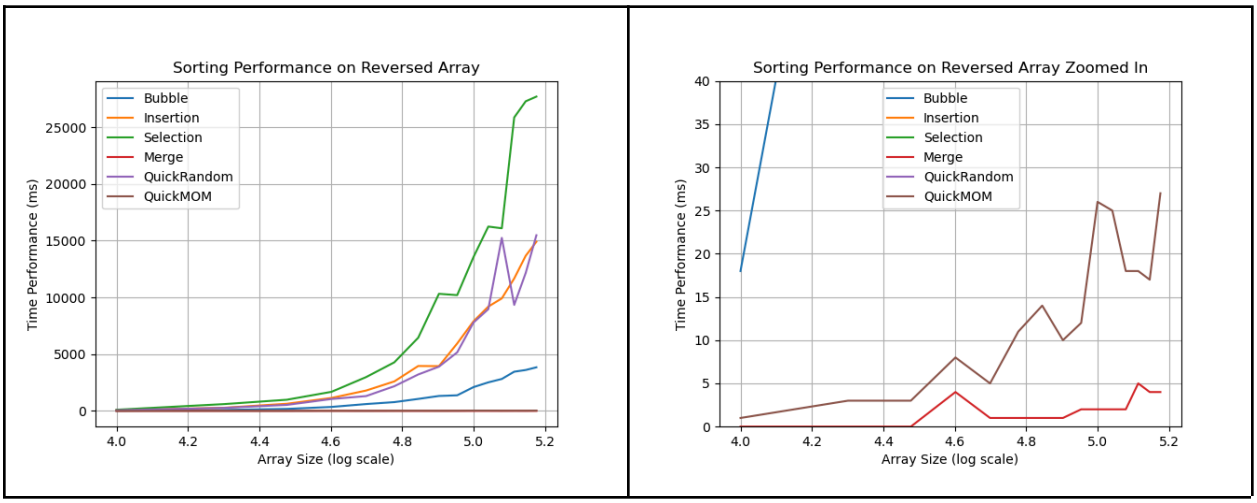


Table2: Size and time(ms) performance of sorting algorithms in reverse sorted arrays

Size	Bubble	Insertion	Selection	Merge	QuickRandom	QuickMOM
10000	18	65	101	0	57	1
20000	85	276	598	0	264	3
30000	174	630	984	0	529	3
40000	349	1140	1679	4	1048	8
50000	602	1792	2976	1	1304	5
60000	774	2597	4278	1	2174	11
70000	1055	3956	6447	1	3208	14
80000	1315	3943	10315	1	3897	10
90000	1373	5940	10199	2	5157	12
100000	2097	7895	13562	2	7786	26

110000	2514	9189	16240	2	8964	25
120000	2813	9911	16090	2	15245	18
130000	3454	11667	25855	5	9339	18
140000	3603	13666	27276	4	12152	17
150000	3840	14902	27690	4	15475	27

Input Type 3: Random Arrays (Average Case Performance)

Random arrays represent the average case scenario for sorting algorithms, calculated by sorting three independently generated random arrays and averaging their runtimes. This method ensures balanced evaluation for general use.

Observations from the Graphs

1. **Bubble Sort**:
    - Performs the worst with a steep curve reflecting its expected O(n^2) complexity..
2. **Insertion Sort**:
    - Slightly better than Bubble Sort but still inefficient due to O(n^2) time complexity. Suitable only for smaller arrays.
3. **Selection Sort**:
    - Similar poor performance to Bubble Sort, with a steep rise in runtime due to its O(n^2) complexity.
4. **Merge Sort**:
    - Outperforms all the other algorithms. Consistent with the O(nlogn) performance, making it scalable for large arrays.
5. **Quick Sort (Random Pivot)**:
    - Comparable performance and shape to the merge sort O(nlogn).
6. **Quicksort with Median of Medians (MoM)**:

- Underperformance compared to merge sort and quicksort with random pivot selection, however does much better than bubble sort, insertion sort and selection sort.
- The zoomed-in graph shows minimal differences between Merge Sort, Quick Sort (Random Pivot), and Quick Sort (MoM) for small arrays, but Quicksort random pivot and Merge Sort excel as arrays grow larger.

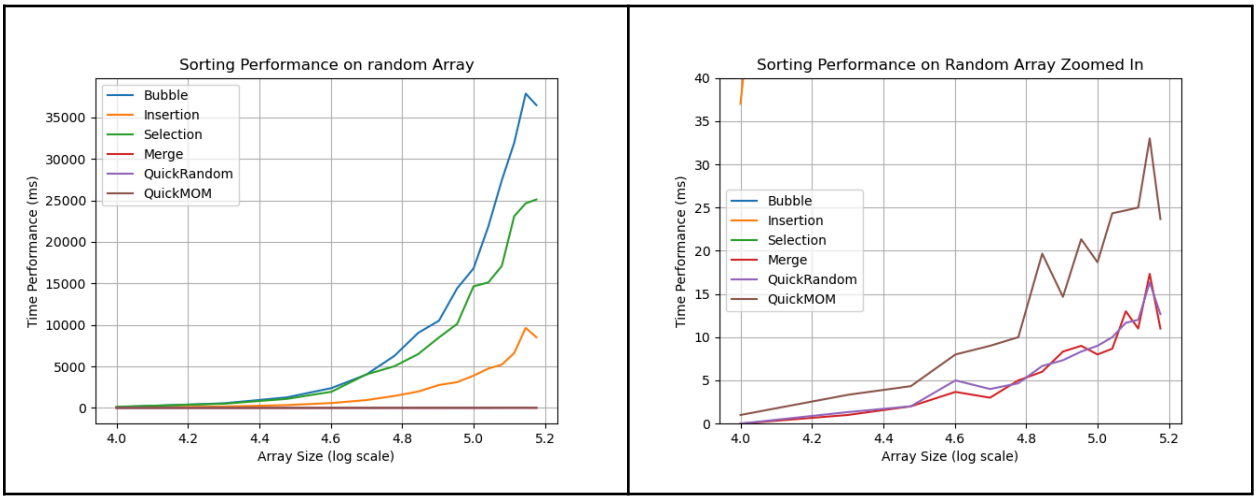


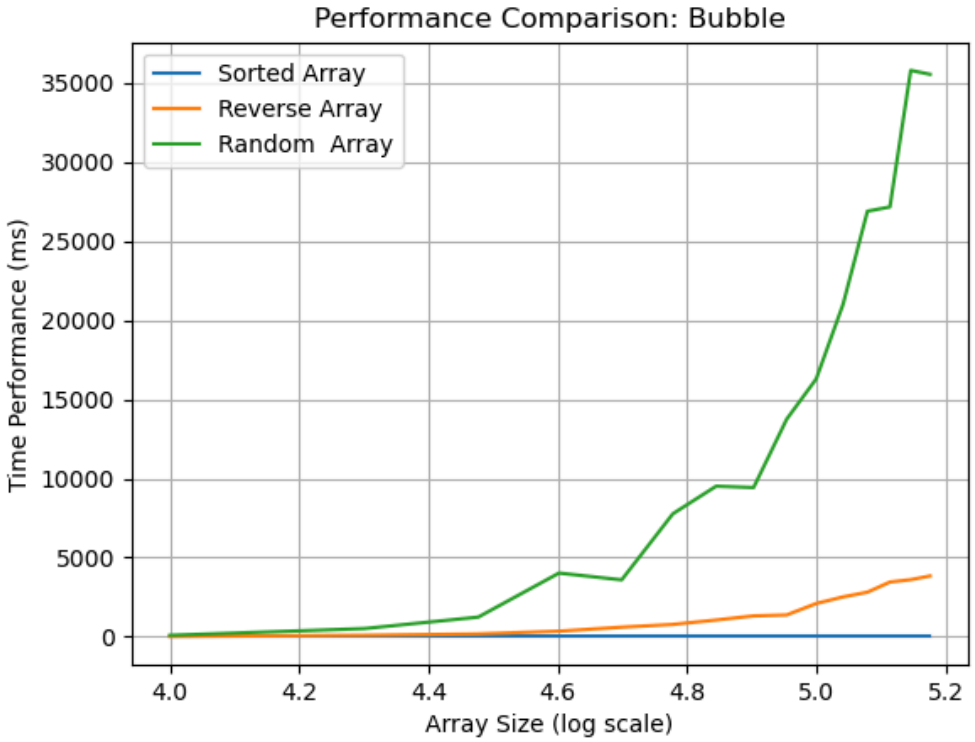
Table3: Size and time(ms) performance of sorting algorithms in reverse sorted arrays

Size	Bubble	Insertion	Selection	Merge	QuickRandom	QuickMOM
10000	86.33333	37	121.6667	0	0	1
20000	514	155.6667	459.6667	1	1.333333	3.333333
30000	1237.667	358.6667	1095	2	2	4.333333
40000	4022	650	2051.667	3.666667	5	8
50000	3592.333	905	3057	3	4	9
60000	7768	1504	4481.667	5	4.666667	10
70000	9512.333	1906	6196.333	6	6.666667	19.66667
80000	9428.333	2089	7520	8.333333	7.333333	14.66667
90000	13746	3128.333	10650.33	9	8.333333	21.33333

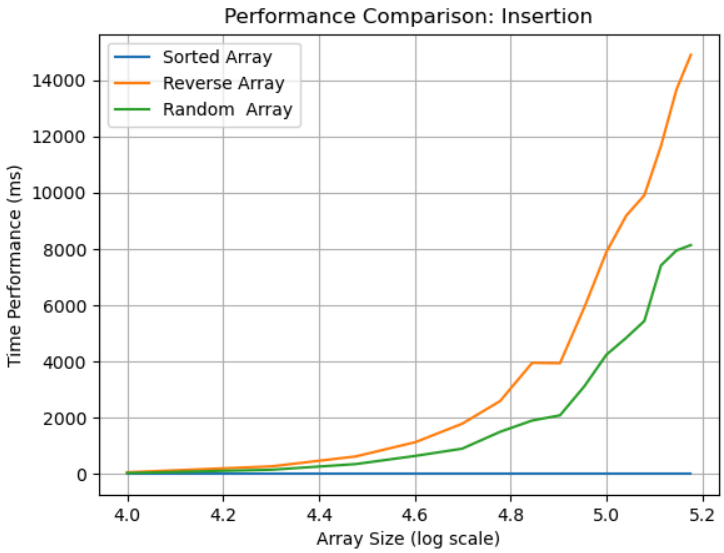
100000	16297.67	4245	12646	8	9	18.66667
110000	20999	4845	17291.67	8.666667	10	24.33333
120000	26892.33	5443.333	17090	13	11.66667	24.66667
130000	27162	7419.333	24860.33	11	12	25
140000	35793.33	7949	25875.67	17.33333	16.33333	33
150000	35546	8140.333	25095.33	11	12.66667	23.66667

Individual algorithm performance across variable input

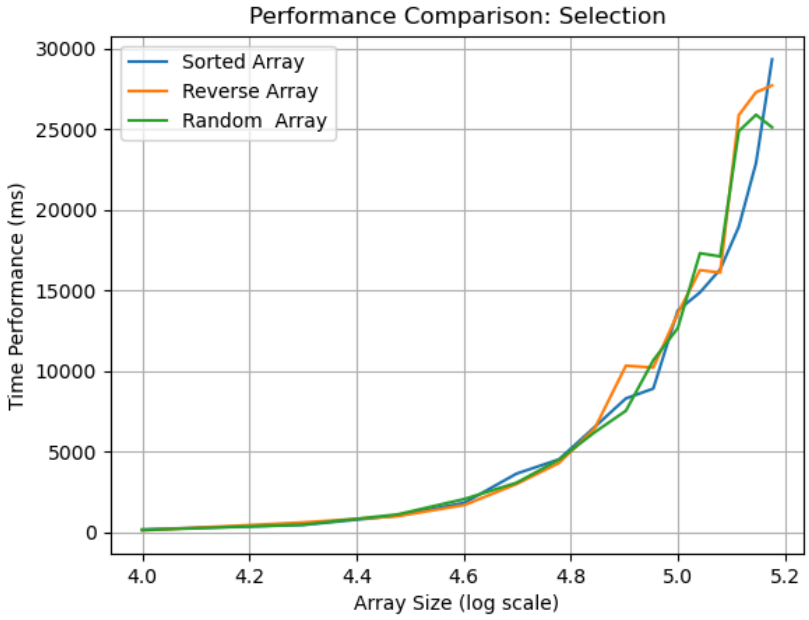
1. Bubble Sort: Bubble sort performs the best on already sorted arrays and the worst on randomly sorted arrays. However, I had expected to do better in randomly sorted arrays compared to reverse sorted arrays. Since for a reverse sorted array it has to make the most number of swaps for each element to move it in the correct position. Further, inspection is needed to figure out this behavior.



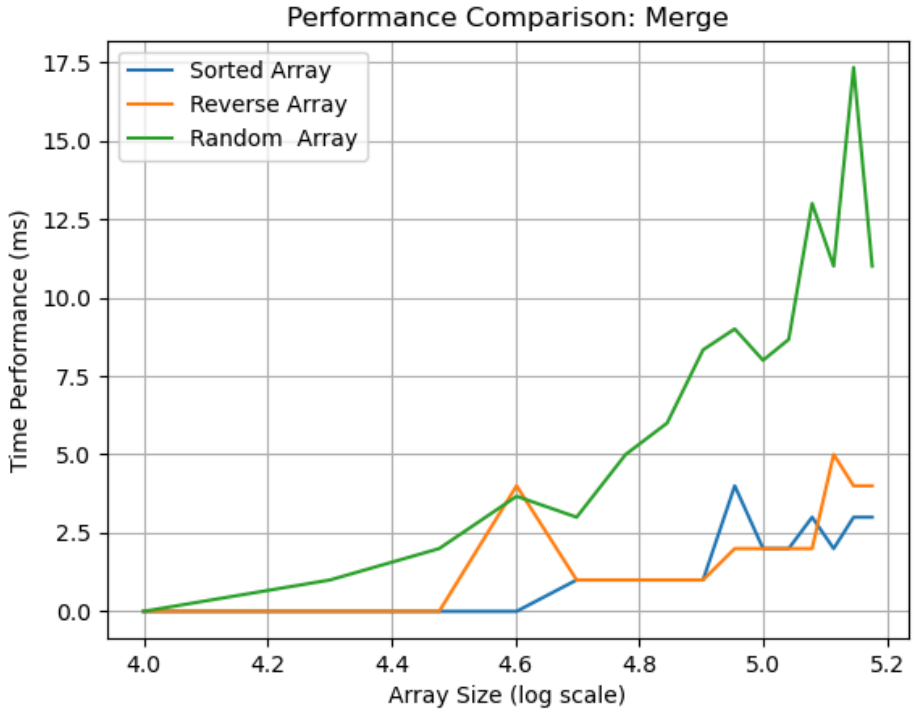
2. Insertion Sort: Insertion sort has similar characteristics as the bubble sort. Performs the best on already sorted arrays and worst on reversely sorted arrays, where the execution times increases exponentially as the array size increases.



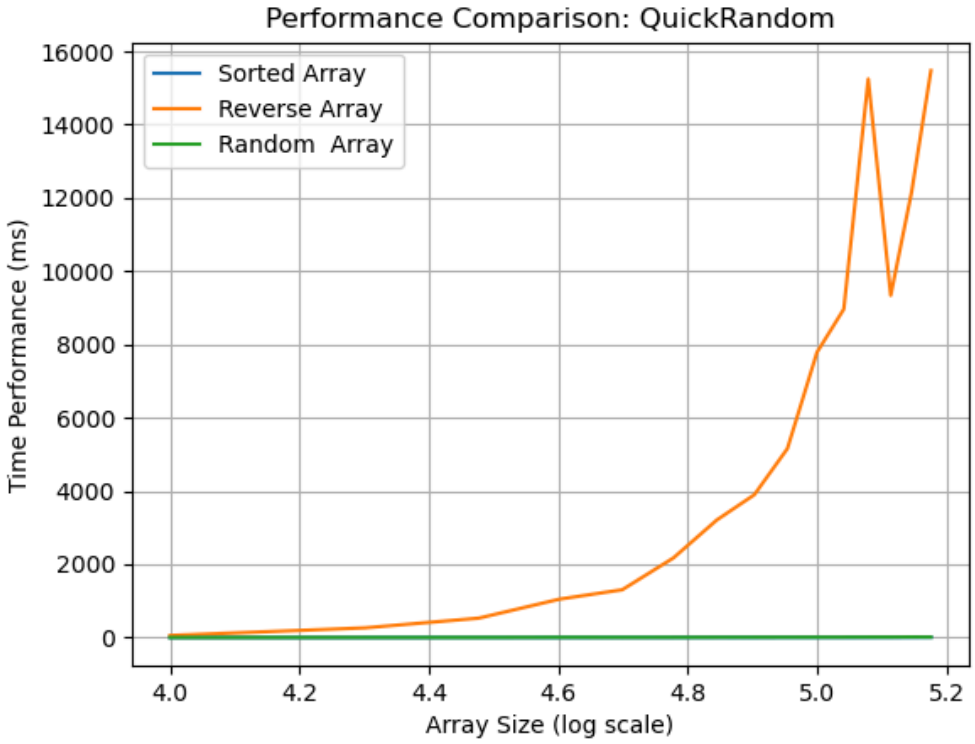
3. Selection Sort: Selection sort execution time increases exponentially as the array size grows, regardless of the input type.



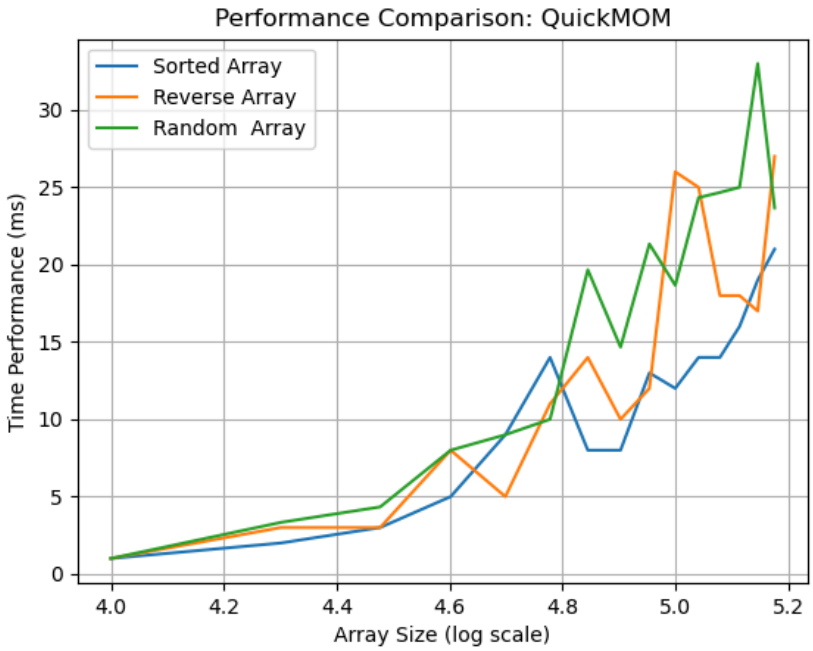
4. Merge Sort: Merge sort similarly for already sorted arrays and reverse sorted array, and the runtime is slightly higher for the random sorted arrays.



5. Quick Sort: the quick sort performance significantly when the pivot is selected as the minimum element of the partition indicated by the orange line.



6. Quicksort with Median of MediansInsertion sort



The best performing sorting algorithm as the array size increases is merge sort and the worst is selection sort regardless of the input type.