INDIAN INSTITUTE OF TECHNOLOGY ROPAR

Data Structures & Algorithms (CS506)

Lab Assignment 1

Name	Rajeshwar Singh
Entry Number	2025CSM1013
Department	Computer Science & Engineering
Course ID	CS506

SYSTEM CONFIGURATION

Device	MacBook Air (M2, 2022)
Processor	Apple M2 Chip
CPU Core	8 core
RAM	8 GB
GPU	8 core
Storage	256 GB SSD
OS	macOS Sequoia 15.5

1. BUBBLE SORT

Bubble sort is one of the simplest algorithms which works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order.

Time complexity: O(n) in best case (with optimization), O(n^2) in average & worst case.

Array sizes used: 10, 100, 1000, 5000, 10000, 50000, 100000, 500000, 1000000.

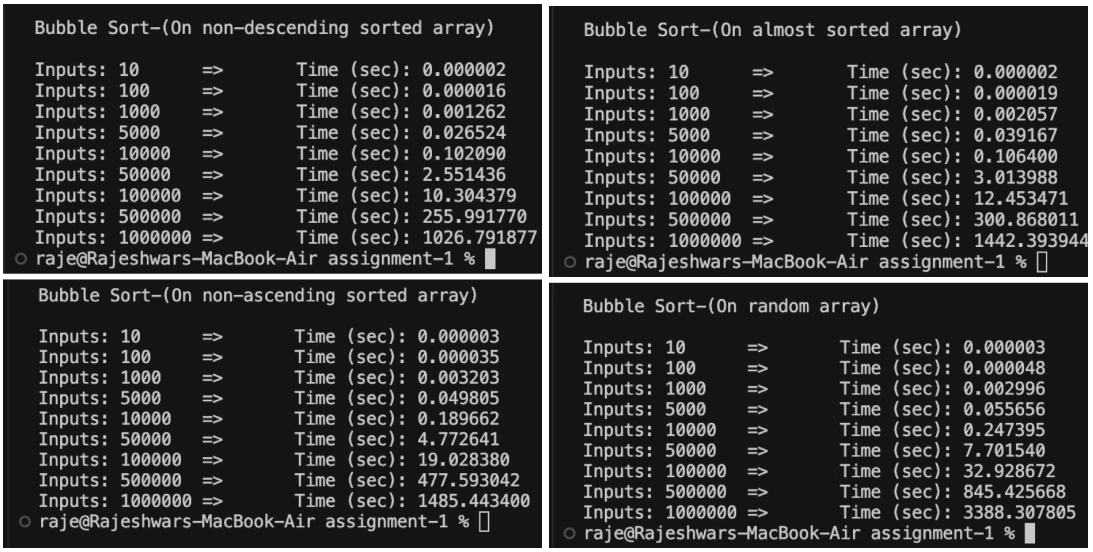

Fig. 1: Bubble sort time data

1.1. OBSERVATIONS

1.1.1. **Random Array:** The performance is very poor for large array sizes showing the "worst-case" O(n^2) complexity.

1.1.2. **Non-Decreasing Sorted Array:** It performed better than all other types of arrays with different sizes. With an optimized algorithm, the time complexity will be reduced to "best-case" O(n).

1.1.3. **Non-Increasing Sorted Array:** The performance is similar to that of a random array due to the large number of swaps showing O(n^2) complexity.

1.1.4. **Almost Sorted Array:** Performed better than random array but still took considerable time. There was no improvement over the sorted array.

1.2. COMPARATIVE ANALYSIS

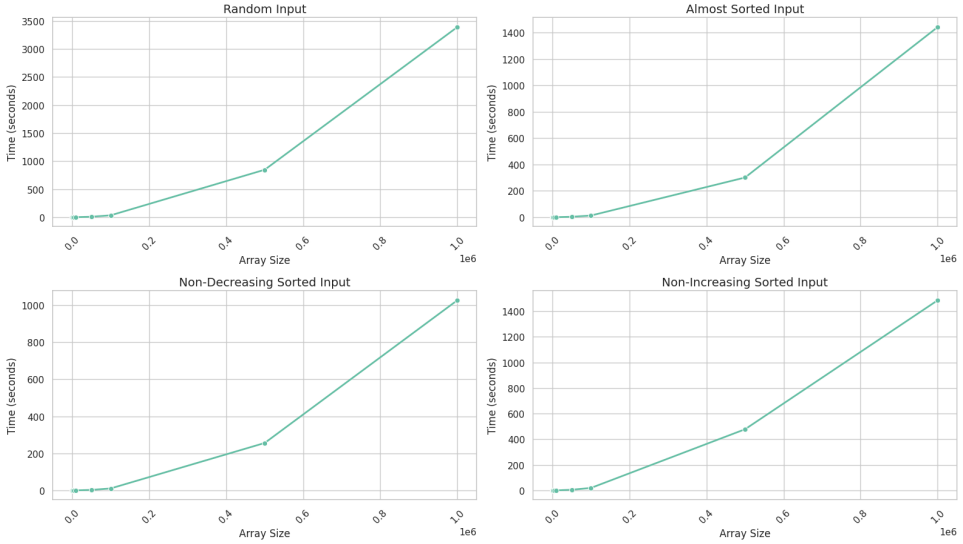

Fig. 2: Bubble sort performance graphs

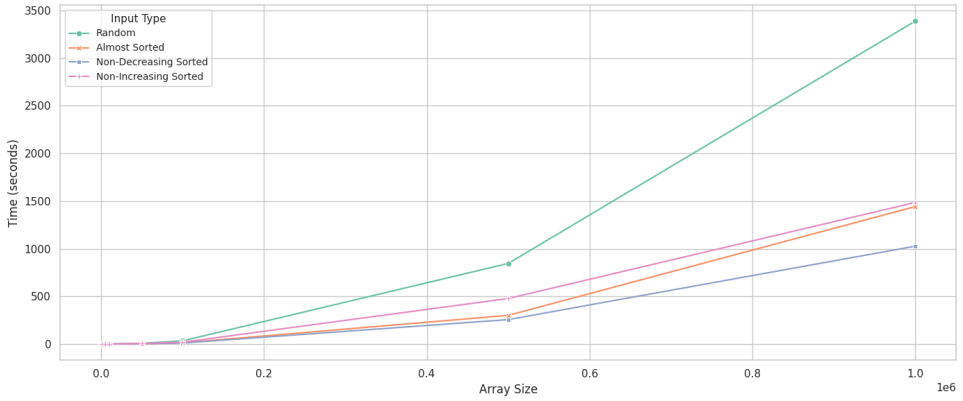

Fig. 3: Bubble sort performance comparison graph

From Fig. 2 and Fig. 3, it can be concluded that bubble sort should not be used for large datasets. The only case where it can perform better is when the array is already sorted and an optimized version of bubble sort is used on it.

2. SELECTION SORT

Selection sort is a simple comparison-based algorithm to sort elements. It repeatedly selects the minimum element from the array and places it at the suitable position in order to sort the array.

Time complexity: $O(n^2)$ in best, average and worst case.

Array sizes used: 10, 100, 1000, 5000, 10000, 50000, 100000, 500000, 1000000.

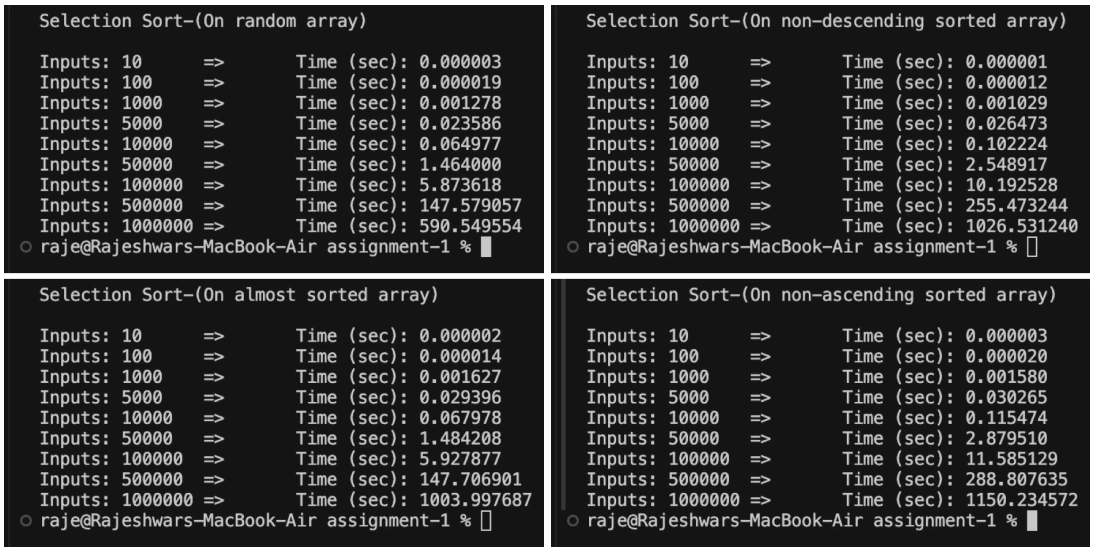

Fig. 4: Selection sort time data

2.1. OBSERVATIONS

2.1.1. **Random Array:** Selection sort performs slightly better when compared to other arrays but the time complexity is still $O(n^2)$.

2.1.2. **Non-Decreasing Sorted Array:** The performance does not improve even when sorted array provided and showed time complexity as $O(n^2)$.

2.1.3. **Non-Increasing Sorted Array:** The algorithm took the longest time to sort the array due to large number of comparisons and swapping.

2.1.4. **Almost Sorted Array:** There was no benefit with almost sorted arrays as it took considerable time to sort the elements with $O(n^2)$ complexity.

2.2. COMPARATIVE ANALYSIS

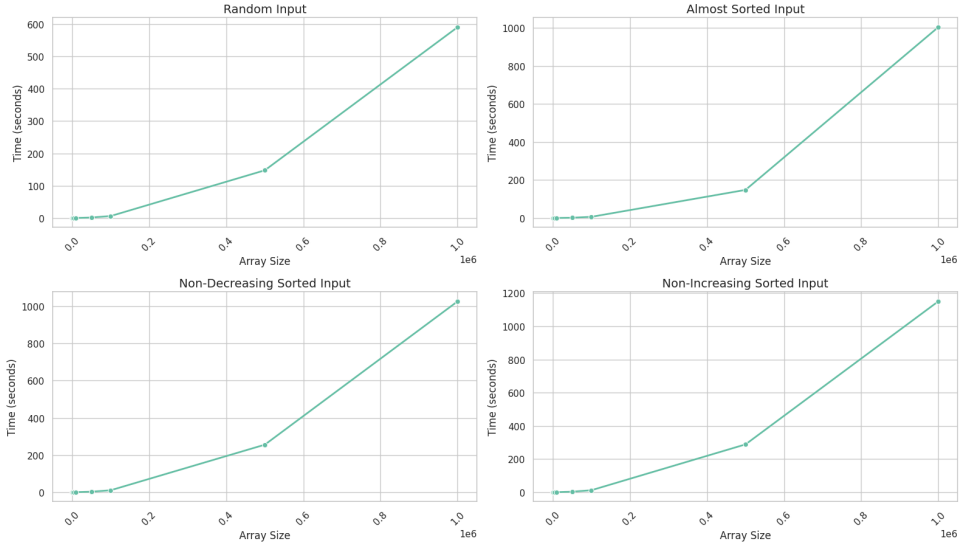

Fig. 5: Selection sort performance graphs

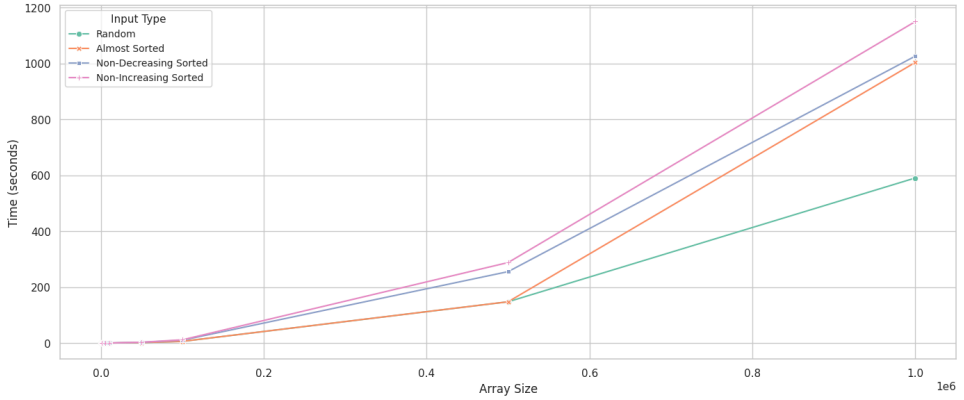

Fig. 6: Selection sort performance comparison graph

From Fig. 5 and Fig. 6, it was observed that the selection sort always performs $O(n^2)$ comparisons. For non-increasing sorted data, the performance is slightly worse due to more swaps that needed to be performed.

3. MERGE SORT

Merge sort is a divide-and-conquer sorting algorithm. It divides the array into two halves, sort each half and then merge them.

Time complexity: O(n log n) in best, average and worst case.

Array sizes used: 10, 100, 1000, 5000, 10000, 50000, 100000, 500000, 1000000.

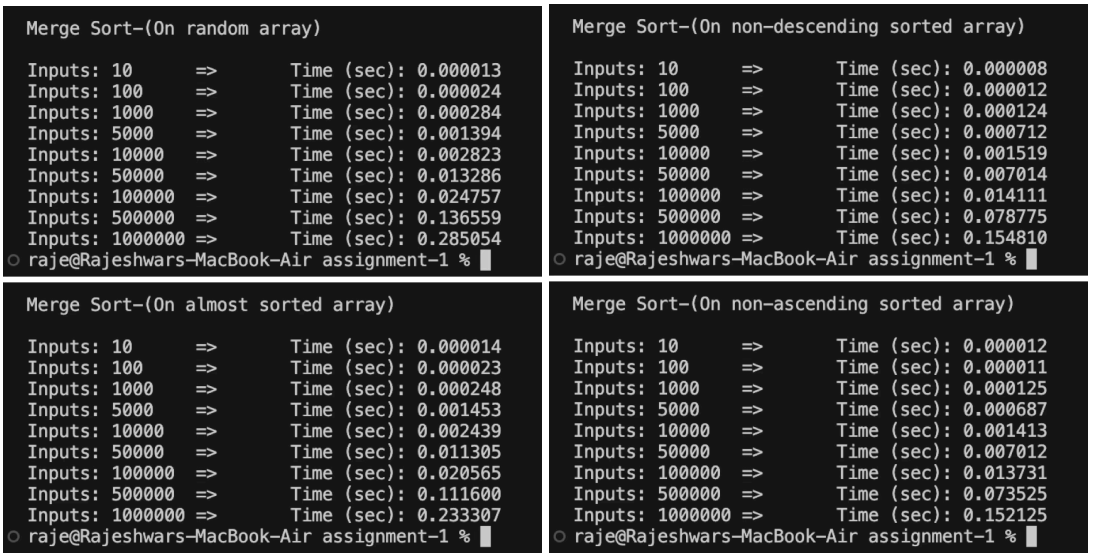

Fig. 7: Merge sort time data

3.1. OBSERVATIONS

3.1.1. **Random Array:** The algorithm performed as expected with time complexity of O(n log n). The time is slightly higher because of more splits and merges.

3.1.2. **Non-Decreasing Sorted Array:** The algorithm was fastest in terms of performance compared to others and showed the required behaviour with O(n log n) complexity.

3.1.3. **Non-Increasing Sorted Array:** The algorithm performed exactly as it performed on non-decreasing array. Here, the time complexity is the same as O (n log n).

3.1.4. **Almost Sorted Array:** Because there were few out of place elements in the array, the algorithm took slightly longer than sorted array but performed better compared to random array with time complexity O(n log n).

3.2. COMPARATIVE ANALYSIS

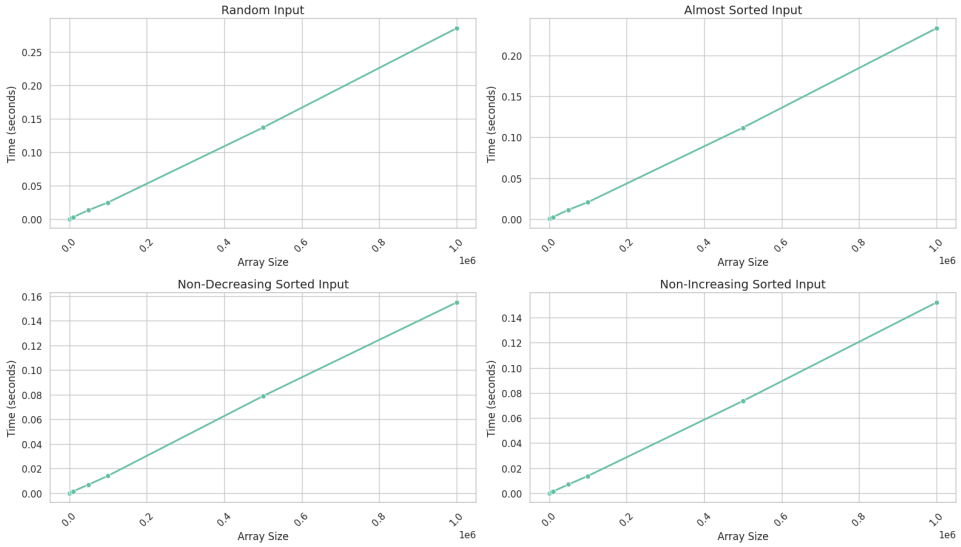

Fig. 8: Merge sort performance graphs

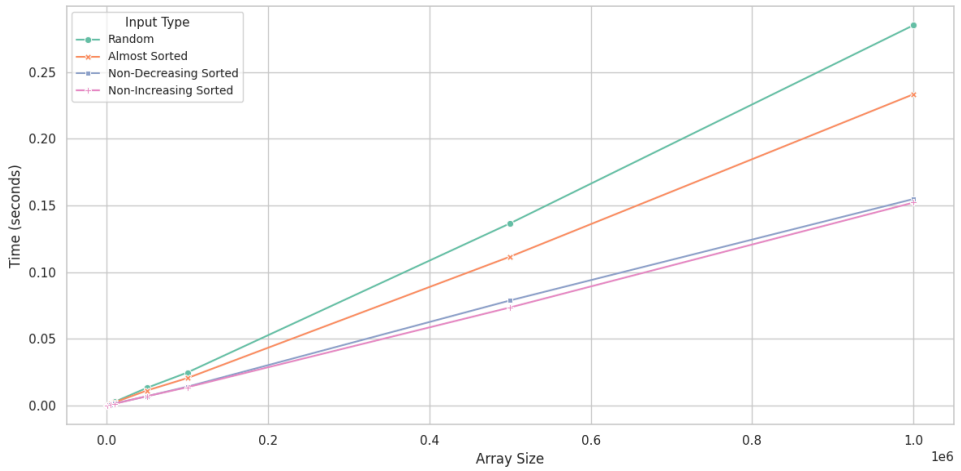

Fig. 9: Merge sort performance comparison graph

From fig. 8 and fig. 9, it is observed that both sorted arrays (non-decreasing and non-increasing) took less time than random array input due to more merge steps. Other than that, no anomalies were observed. Merge sort is stable in all different types of arrays used.

4. INSERTION SORT

Insertion sort is a comparison-based sorting algorithm. It creates a sorted array by picking one element at a time and placing it in the correct position.

Time complexity: O(n) in best case, $O(n^2)$ in average & worst case

Array sizes used: 10, 100, 1000, 5000, 10000, 50000, 100000, 500000, 1000000.

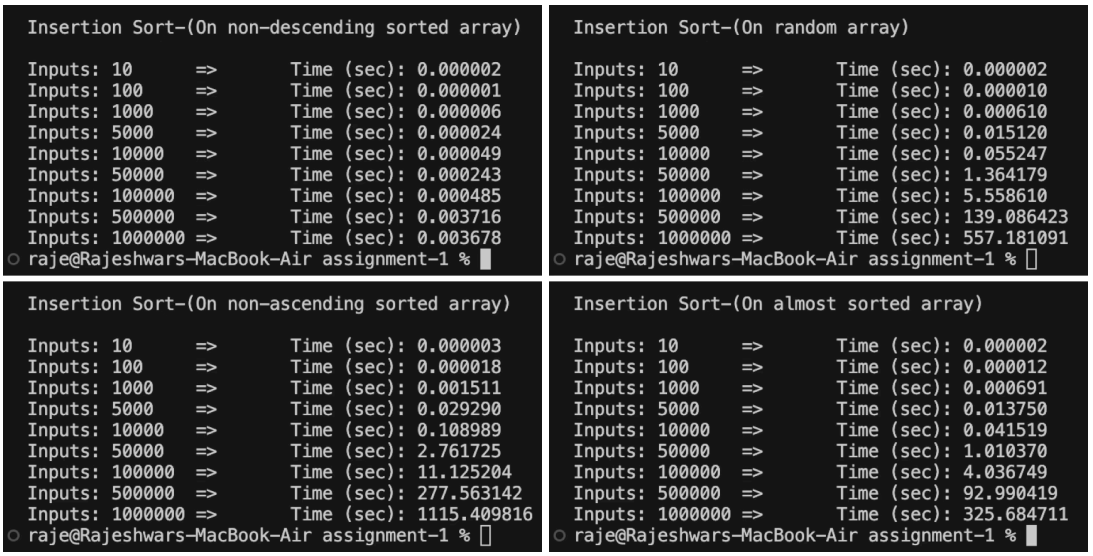

Fig. 10: Insertion sort time data

4.1. OBSERVATIONS

4.1.1. **Random Array:** As the size of the array increases, the increase of time proves the $O(n^2)$ time complexity.

4.1.2. **Non-Decreasing Sorted Array:** The execution time of the algorithm is very fast even for 10^6 input sizes making it a “best-case” scenario with time complexity O(n).

4.1.3. **Non-Increasing Sorted Array:** For input size of order 10^6, the execution time crosses 1000 seconds and shows the “worst-case” performance among all with time complexity $O(n^2)$.

4.1.4. **Almost Sorted Array:** The performance is slightly better than the random input but the time complexity is still $O(n^2)$.

4.2. COMPARATIVE ANALYSIS

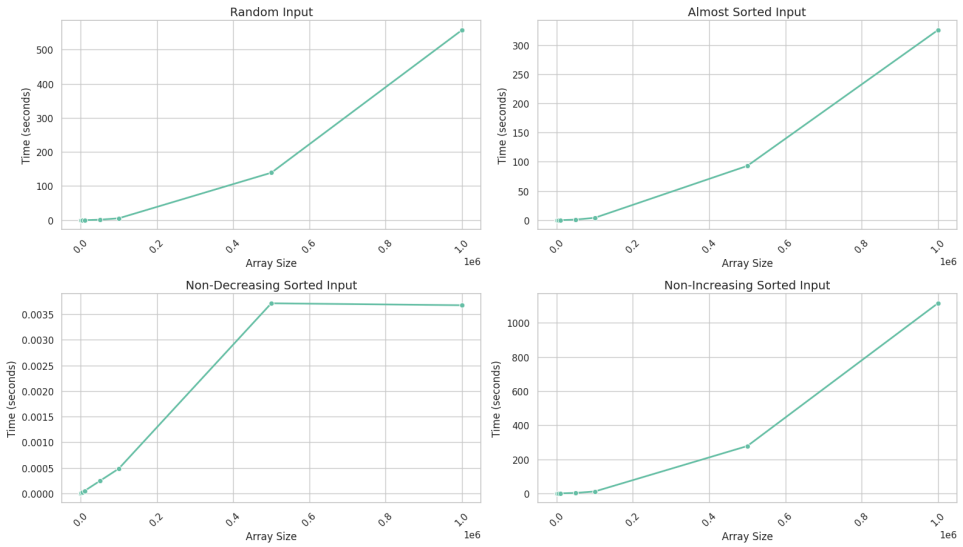

Fig. 11: Insertion sort performance graphs

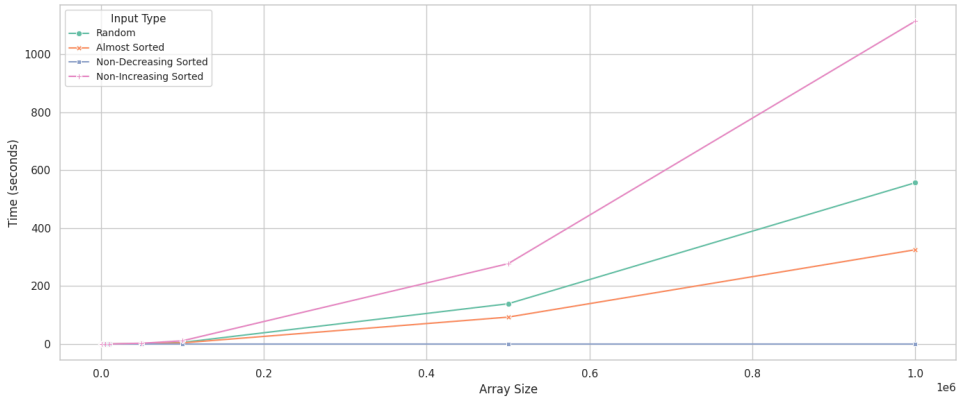

Fig. 12: Insertion sort performance comparison graph

From Fig. 11 and Fig. 12, we can observe that as the input size increases, time taken by the insertion sort algorithm increases sharply for most types (except non-decreasing sorted arrays). Insertion sort is suitable for small sizes or nearly sorted data but slow for large or reverse sorted arrays.

5. QUICK SORT

Quick sort is divide-and-conquer sorting algorithm. It first selects a pivot element (start, middle, end or median) and then creates partitions to sort the elements. Elements less than pivot go to the left of pivot and elements greater than pivot go to the right side.

In my analysis, I have used the start element as the pivot element.

Time complexity: O(n log n) in best & average case, $O(n^2)$ in worst case.

Array sizes used: 10, 100, 1000, 5000, 10000, 50000, 100000, 120000, 150000.

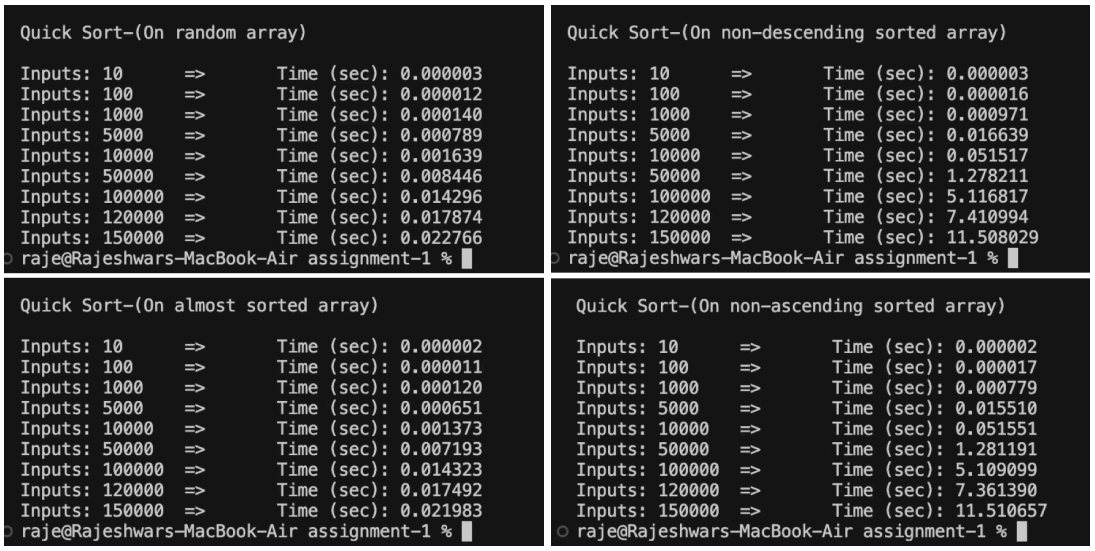

Fig. 13: Quick sort time data

5.1. OBSERVATIONS

5.1.1. **Random Array:** The time value is following the curve of order O(n log n) means that pivot splits the array in a balanced manner.

5.1.2. **Non-Decreasing Sorted Array:** "Worst-case" time complexity of $O(n^2)$ is observed when the start element is taken as pivot.

5.1.3. **Non-Increasing Sorted Array:** Similar "Worst-case" time complexity of $O(n^2)$ is observed.

5.1.4. **Almost Sorted Array:** Similar performance as random array with time complexity O(n log n)

5.2. COMPARATIVE ANALYSIS

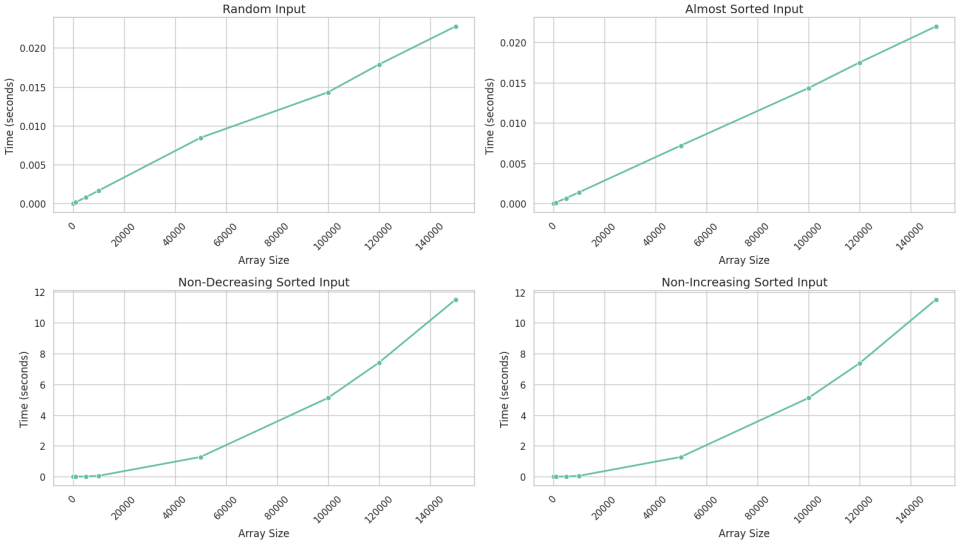

Fig. 14: Quick sort performance graphs

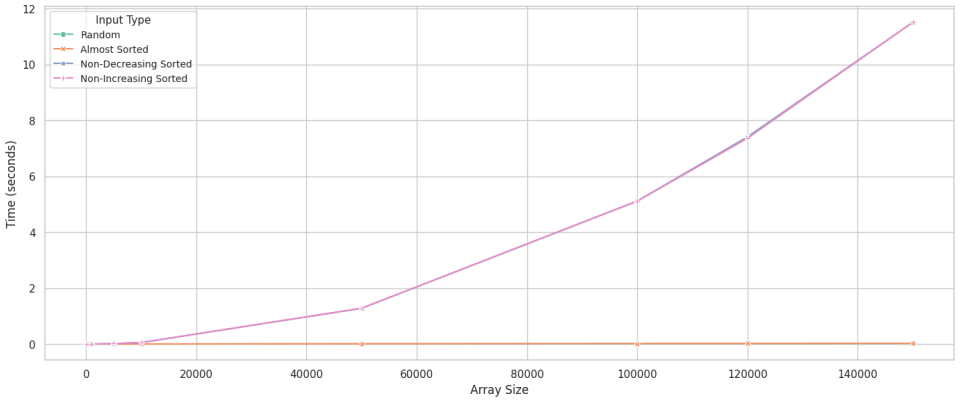

Fig. 15: Quick sort performance comparison graph

From Fig. 14 and Fig. 15, it can be observed that pivot choice can affect negatively on performance of quick sort. With pivot as the start element, applying a quick sort on sorted array (non-decreasing & non-increasing) creates unbalanced partitions leading to time complexity of $O(n^2)$.