

## INDIAN INSTITUTE OF TECHNOLOGY ROPAR

Data Structures & Algorithms (CS506)

### Lab Assignment 1

<b>Name</b>	Rajeshwar Singh
<b>Entry Number</b>	2025CSM1013
<b>Department</b>	Computer Science & Engineering
<b>Course ID</b>	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.

Bubble Sort-(On non-descending sorted array)		Bubble Sort-(On almost sorted array)	
Inputs: 10 =>	Time (sec): 0.000002	Inputs: 10 =>	Time (sec): 0.000002
Inputs: 100 =>	Time (sec): 0.000016	Inputs: 100 =>	Time (sec): 0.000019
Inputs: 1000 =>	Time (sec): 0.001262	Inputs: 1000 =>	Time (sec): 0.002057
Inputs: 5000 =>	Time (sec): 0.026524	Inputs: 5000 =>	Time (sec): 0.039167
Inputs: 10000 =>	Time (sec): 0.102090	Inputs: 10000 =>	Time (sec): 0.106400
Inputs: 50000 =>	Time (sec): 2.551436	Inputs: 50000 =>	Time (sec): 3.013988
Inputs: 100000 =>	Time (sec): 10.304379	Inputs: 100000 =>	Time (sec): 12.453471
Inputs: 500000 =>	Time (sec): 255.991770	Inputs: 500000 =>	Time (sec): 300.868011
Inputs: 1000000 =>	Time (sec): 1026.791877	Inputs: 1000000 =>	Time (sec): 1442.393944
○ raje@Rajeshwars-MacBook-Air assignment-1 % █		○ raje@Rajeshwars-MacBook-Air assignment-1 % █	
Bubble Sort-(On non-ascending sorted array)		Bubble Sort-(On random array)	
Inputs: 10 =>	Time (sec): 0.000003	Inputs: 10 =>	Time (sec): 0.000003
Inputs: 100 =>	Time (sec): 0.000035	Inputs: 100 =>	Time (sec): 0.000048
Inputs: 1000 =>	Time (sec): 0.003203	Inputs: 1000 =>	Time (sec): 0.002996
Inputs: 5000 =>	Time (sec): 0.049805	Inputs: 5000 =>	Time (sec): 0.055656
Inputs: 10000 =>	Time (sec): 0.189662	Inputs: 10000 =>	Time (sec): 0.247395
Inputs: 50000 =>	Time (sec): 4.772641	Inputs: 50000 =>	Time (sec): 7.701540
Inputs: 100000 =>	Time (sec): 19.028380	Inputs: 100000 =>	Time (sec): 32.928672
Inputs: 500000 =>	Time (sec): 477.593042	Inputs: 500000 =>	Time (sec): 845.425668
Inputs: 1000000 =>	Time (sec): 1485.443400	Inputs: 1000000 =>	Time (sec): 3388.307805
○ raje@Rajeshwars-MacBook-Air assignment-1 % █		○ raje@Rajeshwars-MacBook-Air assignment-1 % █	

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

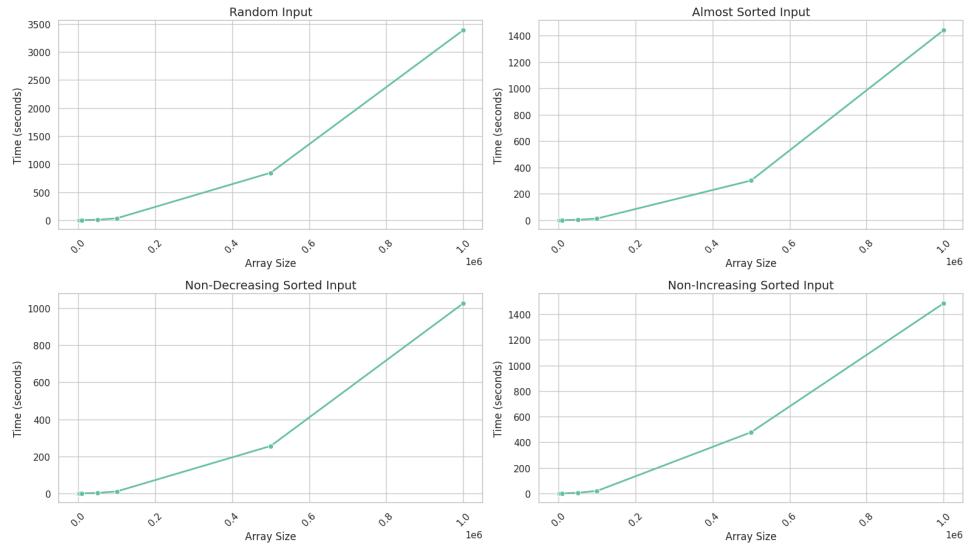


Fig. 2: Bubble sort performance graphs

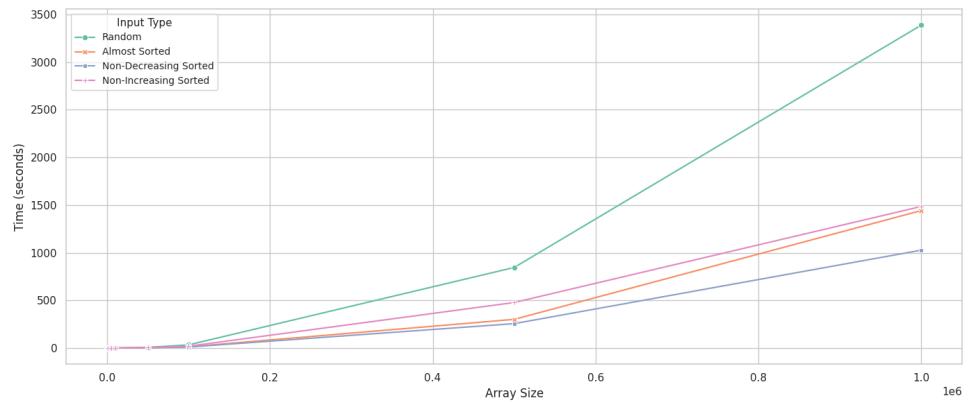


Fig. 3: Bubble sort performance comparison graph

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.

<b>Selection Sort-(On random array)</b> <pre>Inputs: 10      =&gt;      Time (sec): 0.000003 Inputs: 100     =&gt;      Time (sec): 0.000019 Inputs: 1000    =&gt;      Time (sec): 0.001278 Inputs: 5000    =&gt;      Time (sec): 0.023586 Inputs: 10000   =&gt;      Time (sec): 0.064977 Inputs: 50000   =&gt;      Time (sec): 1.464000 Inputs: 100000  =&gt;      Time (sec): 5.873618 Inputs: 500000  =&gt;      Time (sec): 147.579057 Inputs: 1000000 =&gt;      Time (sec): 590.549554 ○ raje@Rajeshwars-MacBook-Air assignment-1 % █</pre>	<b>Selection Sort-(On non-descending sorted array)</b> <pre>Inputs: 10      =&gt;      Time (sec): 0.000001 Inputs: 100     =&gt;      Time (sec): 0.000012 Inputs: 1000    =&gt;      Time (sec): 0.001029 Inputs: 5000    =&gt;      Time (sec): 0.026473 Inputs: 10000   =&gt;      Time (sec): 0.102224 Inputs: 50000   =&gt;      Time (sec): 2.548917 Inputs: 100000  =&gt;      Time (sec): 10.192528 Inputs: 500000  =&gt;      Time (sec): 255.473244 Inputs: 1000000 =&gt;      Time (sec): 1026.531240 ○ raje@Rajeshwars-MacBook-Air assignment-1 % █</pre>
<b>Selection Sort-(On almost sorted array)</b> <pre>Inputs: 10      =&gt;      Time (sec): 0.000002 Inputs: 100     =&gt;      Time (sec): 0.000014 Inputs: 1000    =&gt;      Time (sec): 0.001627 Inputs: 5000    =&gt;      Time (sec): 0.029396 Inputs: 10000   =&gt;      Time (sec): 0.067978 Inputs: 50000   =&gt;      Time (sec): 1.484208 Inputs: 100000  =&gt;      Time (sec): 5.927877 Inputs: 500000  =&gt;      Time (sec): 147.706901 Inputs: 1000000 =&gt;      Time (sec): 1003.997687 ○ raje@Rajeshwars-MacBook-Air assignment-1 % █</pre>	<b>Selection Sort-(On non-ascending sorted array)</b> <pre>Inputs: 10      =&gt;      Time (sec): 0.000003 Inputs: 100     =&gt;      Time (sec): 0.000020 Inputs: 1000    =&gt;      Time (sec): 0.001580 Inputs: 5000    =&gt;      Time (sec): 0.030265 Inputs: 10000   =&gt;      Time (sec): 0.115474 Inputs: 50000   =&gt;      Time (sec): 2.879510 Inputs: 100000  =&gt;      Time (sec): 11.585129 Inputs: 500000  =&gt;      Time (sec): 288.807635 Inputs: 1000000 =&gt;      Time (sec): 1150.234572 ○ raje@Rajeshwars-MacBook-Air assignment-1 % █</pre>

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

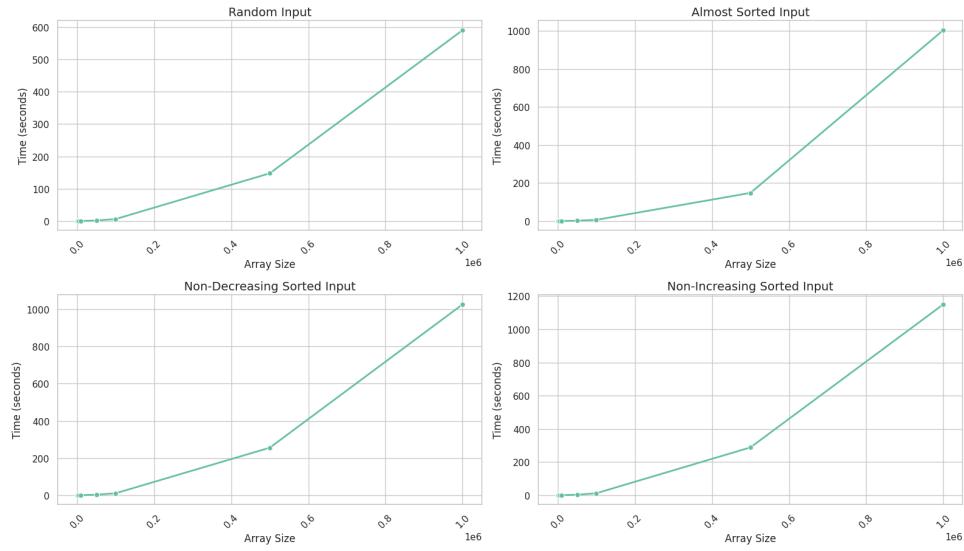


Fig. 5: Selection sort performance graphs

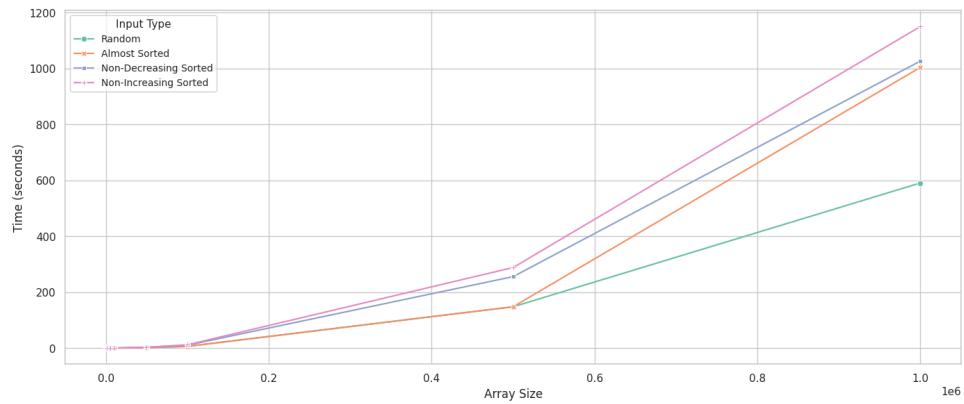


Fig. 6: Selection sort performance comparison graph

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.

Merge Sort-(On random array)	Merge Sort-(On non-descending sorted array)
<pre>Inputs: 10    =&gt;    Time (sec): 0.000013 Inputs: 100   =&gt;    Time (sec): 0.000024 Inputs: 1000  =&gt;    Time (sec): 0.000284 Inputs: 5000  =&gt;    Time (sec): 0.001394 Inputs: 10000 =&gt;    Time (sec): 0.002823 Inputs: 50000 =&gt;    Time (sec): 0.013286 Inputs: 100000 =&gt;    Time (sec): 0.024757 Inputs: 500000 =&gt;    Time (sec): 0.136559 Inputs: 1000000 =&gt;   Time (sec): 0.285054 o raje@Rajeshwars-MacBook-Air assignment-1 %</pre>	<pre>Inputs: 10    =&gt;    Time (sec): 0.000008 Inputs: 100   =&gt;    Time (sec): 0.000012 Inputs: 1000  =&gt;    Time (sec): 0.000124 Inputs: 5000  =&gt;    Time (sec): 0.000712 Inputs: 10000 =&gt;   Time (sec): 0.001519 Inputs: 50000 =&gt;   Time (sec): 0.007014 Inputs: 100000 =&gt;  Time (sec): 0.014111 Inputs: 500000 =&gt;  Time (sec): 0.078775 Inputs: 1000000 =&gt; Time (sec): 0.154810 o raje@Rajeshwars-MacBook-Air assignment-1 %</pre>
Merge Sort-(On almost sorted array)	Merge Sort-(On non-ascending sorted array)
<pre>Inputs: 10    =&gt;    Time (sec): 0.000014 Inputs: 100   =&gt;    Time (sec): 0.000023 Inputs: 1000  =&gt;    Time (sec): 0.000248 Inputs: 5000  =&gt;    Time (sec): 0.001453 Inputs: 10000 =&gt;   Time (sec): 0.002439 Inputs: 50000 =&gt;   Time (sec): 0.011305 Inputs: 100000 =&gt;  Time (sec): 0.020565 Inputs: 500000 =&gt;  Time (sec): 0.111600 Inputs: 1000000 =&gt; Time (sec): 0.233307 o raje@Rajeshwars-MacBook-Air assignment-1 %</pre>	<pre>Inputs: 10    =&gt;    Time (sec): 0.000012 Inputs: 100   =&gt;    Time (sec): 0.000011 Inputs: 1000  =&gt;    Time (sec): 0.000125 Inputs: 5000  =&gt;    Time (sec): 0.000687 Inputs: 10000 =&gt;   Time (sec): 0.001413 Inputs: 50000 =&gt;   Time (sec): 0.007012 Inputs: 100000 =&gt;  Time (sec): 0.013731 Inputs: 500000 =&gt;  Time (sec): 0.073525 Inputs: 1000000 =&gt; Time (sec): 0.152125 o raje@Rajeshwars-MacBook-Air assignment-1 %</pre>

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

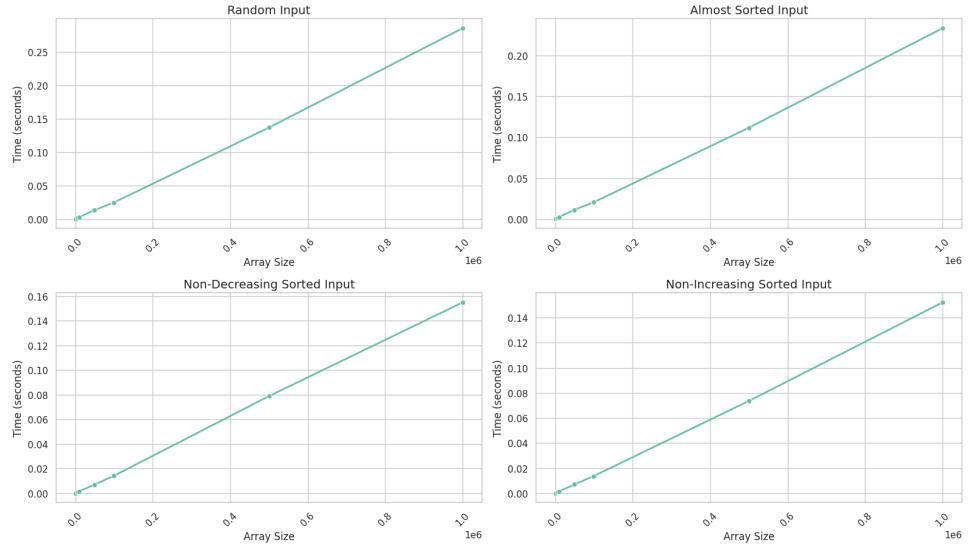


Fig. 8: Merge sort performance graphs

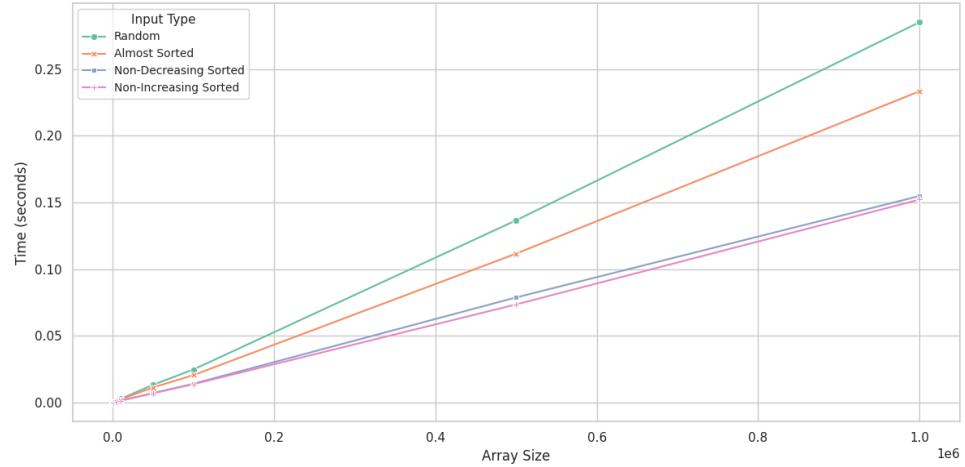


Fig. 9: Merge sort performance comparison graph

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.

<pre>Insertion Sort-(On non-descending sorted array)  Inputs: 10      =&gt;      Time (sec): 0.000002 Inputs: 100     =&gt;      Time (sec): 0.000001 Inputs: 1000    =&gt;      Time (sec): 0.000006 Inputs: 5000    =&gt;      Time (sec): 0.000024 Inputs: 10000   =&gt;      Time (sec): 0.000049 Inputs: 50000   =&gt;      Time (sec): 0.000243 Inputs: 100000  =&gt;      Time (sec): 0.000485 Inputs: 500000  =&gt;      Time (sec): 0.003716 Inputs: 1000000 =&gt;      Time (sec): 0.003678 ○ raje@Rajeshwars-MacBook-Air assignment-1 %</pre>	<pre>Insertion Sort-(On random array)  Inputs: 10      =&gt;      Time (sec): 0.000002 Inputs: 100     =&gt;      Time (sec): 0.000010 Inputs: 1000    =&gt;      Time (sec): 0.000610 Inputs: 5000    =&gt;      Time (sec): 0.015120 Inputs: 10000   =&gt;      Time (sec): 0.055247 Inputs: 50000   =&gt;      Time (sec): 1.364179 Inputs: 100000  =&gt;      Time (sec): 5.558610 Inputs: 500000  =&gt;      Time (sec): 139.086423 Inputs: 1000000 =&gt;      Time (sec): 557.181091 ○ raje@Rajeshwars-MacBook-Air assignment-1 %</pre>
<pre>Insertion Sort-(On non-ascending sorted array)  Inputs: 10      =&gt;      Time (sec): 0.000003 Inputs: 100     =&gt;      Time (sec): 0.000018 Inputs: 1000    =&gt;      Time (sec): 0.001511 Inputs: 5000    =&gt;      Time (sec): 0.029290 Inputs: 10000   =&gt;      Time (sec): 0.108989 Inputs: 50000   =&gt;      Time (sec): 2.761725 Inputs: 100000  =&gt;      Time (sec): 11.125204 Inputs: 500000  =&gt;      Time (sec): 277.563142 Inputs: 1000000 =&gt;      Time (sec): 1115.409816 ○ raje@Rajeshwars-MacBook-Air assignment-1 %</pre>	<pre>Insertion Sort-(On almost sorted array)  Inputs: 10      =&gt;      Time (sec): 0.000002 Inputs: 100     =&gt;      Time (sec): 0.000012 Inputs: 1000    =&gt;      Time (sec): 0.000691 Inputs: 5000    =&gt;      Time (sec): 0.013750 Inputs: 10000   =&gt;      Time (sec): 0.041519 Inputs: 50000   =&gt;      Time (sec): 1.010370 Inputs: 100000  =&gt;      Time (sec): 4.036749 Inputs: 500000  =&gt;      Time (sec): 92.990419 Inputs: 1000000 =&gt;      Time (sec): 325.684711 ○ raje@Rajeshwars-MacBook-Air assignment-1 %</pre>

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

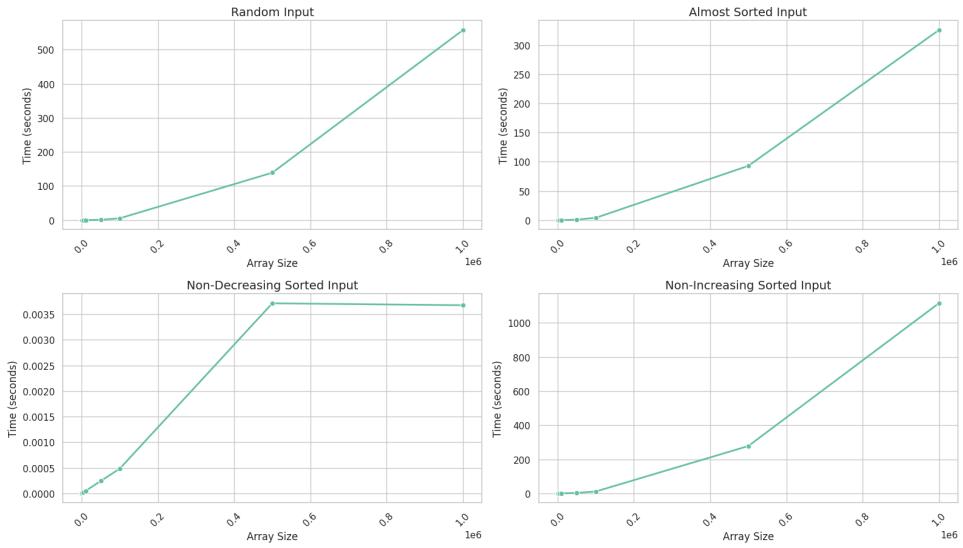


Fig. 11: Insertion sort performance graphs

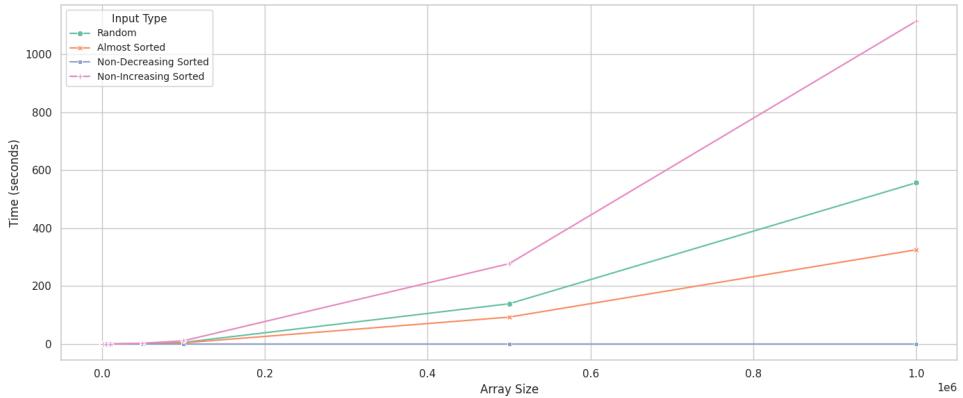


Fig. 12: Insertion sort performance comparison graph

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.

<b>Quick Sort-(On random array)</b> Inputs: 10 => Time (sec): 0.000003 Inputs: 100 => Time (sec): 0.000012 Inputs: 1000 => Time (sec): 0.000140 Inputs: 5000 => Time (sec): 0.000789 Inputs: 10000 => Time (sec): 0.001639 Inputs: 50000 => Time (sec): 0.008446 Inputs: 100000 => Time (sec): 0.014296 Inputs: 120000 => Time (sec): 0.017874 Inputs: 150000 => Time (sec): 0.022766 ○ raje@Rajeshwars-MacBook-Air assignment-1 %	<b>Quick Sort-(On non-descending sorted array)</b> Inputs: 10 => Time (sec): 0.000003 Inputs: 100 => Time (sec): 0.000016 Inputs: 1000 => Time (sec): 0.000971 Inputs: 5000 => Time (sec): 0.016639 Inputs: 10000 => Time (sec): 0.051517 Inputs: 50000 => Time (sec): 1.278211 Inputs: 100000 => Time (sec): 5.116817 Inputs: 120000 => Time (sec): 7.410994 Inputs: 150000 => Time (sec): 11.508029 ○ raje@Rajeshwars-MacBook-Air assignment-1 %
<b>Quick Sort-(On almost sorted array)</b> Inputs: 10 => Time (sec): 0.000002 Inputs: 100 => Time (sec): 0.000011 Inputs: 1000 => Time (sec): 0.000120 Inputs: 5000 => Time (sec): 0.000651 Inputs: 10000 => Time (sec): 0.001373 Inputs: 50000 => Time (sec): 0.007193 Inputs: 100000 => Time (sec): 0.014323 Inputs: 120000 => Time (sec): 0.017492 Inputs: 150000 => Time (sec): 0.021983 ○ raje@Rajeshwars-MacBook-Air assignment-1 %	<b>Quick Sort-(On non-ascending sorted array)</b> Inputs: 10 => Time (sec): 0.000002 Inputs: 100 => Time (sec): 0.000017 Inputs: 1000 => Time (sec): 0.000779 Inputs: 5000 => Time (sec): 0.015510 Inputs: 10000 => Time (sec): 0.051551 Inputs: 50000 => Time (sec): 1.281191 Inputs: 100000 => Time (sec): 5.109099 Inputs: 120000 => Time (sec): 7.361390 Inputs: 150000 => Time (sec): 11.510657 ○ raje@Rajeshwars-MacBook-Air assignment-1 %

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

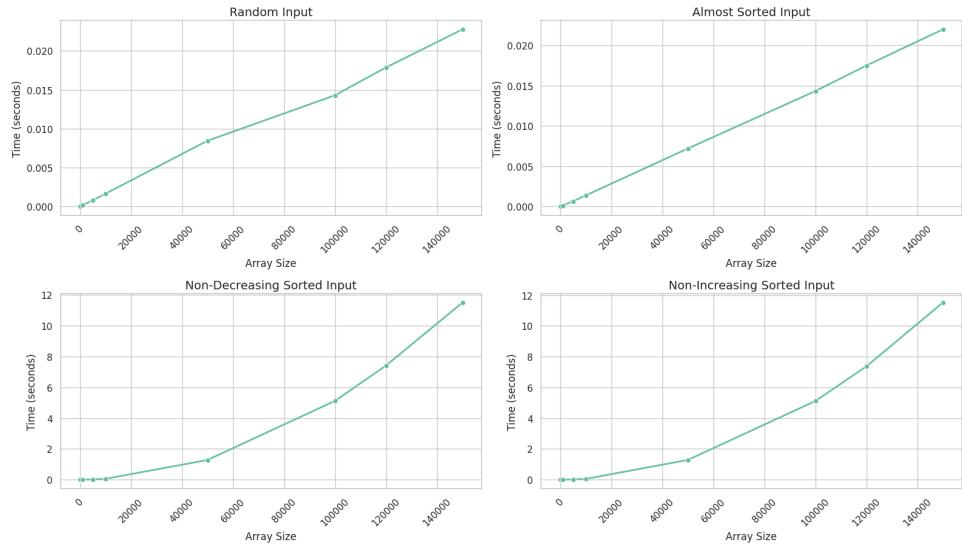


Fig. 14: Quick sort performance graphs

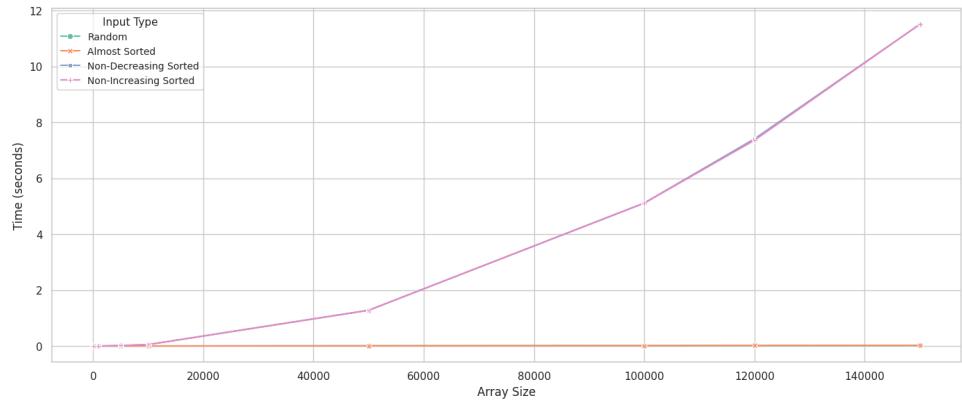


Fig. 15: Quick sort performance comparison graph

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  $O(n^2)$ .