ITCS 6114 - Algorithms & Data Structures

Project 1: Comparison-based Sorting Algorithms

-Ajinkya Gadgil (801200445), Saumitra Apte (801202857)

Project Overview

This project includes implementation, comparison and analysis of different sorting algorithms for different input sizes and comparing the time required for the execution of the sorting algorithms for 3 cases.

1. Input is already sorted (Best Case)
2. Input is sorted reversely (Worst Case)
3. Average case:
    a. In average case we shuffle the input array and execute the sorting algorithms for 'number_of_repetitions' by default we have used 'number_of_repetitions' is 3. So, for average case for each algorithm, for each input size runs for 3 times and we take the average of all 3 times and that is how we get the average time.

Different sorting algorithms used for comparison are as below:

1)Insertion sort

2) Heap Sort

3) Merge Sort

4) In place Quick sort

5) Modified Quick sort

All the algorithms have been implemented in Python.

Data Structures used in the project are as below:

1) **Dictionary:**
    a. We have used dictionary for storing the key value pairs as {key:value}. Where key is the input size and value are the array generated for that input size. We have used random module of python to generate random numbers. We have used **random.sample(sequence,k)** which creates an array in a given sequence and k number of items in the array.
2) **List**: We have used list mainly for:
    a. We store input sizes in a list on which all the algorithms would run.
    b. We have used list as a value field in dictionary which stores the randomly generated array for the given input sizes.
    c. To store execution time for all 3 cases.

Execution Flow Understanding

1) We have created separate files for each sorting algorithm and created a file named **“sorting_algorithm_main.py”** which imports all the algorithm classes and we create object of each class and call the respective main methods of each sorting algorithms.
2) We are randomly generating the inputs using python random module for each input sizes and store it in the dictionary such as {10: [*random list of 10 elements*], 1000: [*random list of 1000 elements*]…..60000:[*random list of 1000 elements*]}
3) For each algorithm we pass three parameters –
   input_dictionary-which has random input arrays as per size
   input_sizes -which has input sizes on which we are running algorithms
   no_of_repetitions- no of times we want to run the algorithm for each input size to calculate average case execution time
4) Inside the algorithm, there are three cases for average, already sorted and reverse sorted array. In average case, as per given no of repetitions, array is shuffled and sorted and average is calculated.
5) As the array is sorted in average case same array is used for best case i.e. when the array is sorted
6) Array is reverse sorted and passed to the sorting algorithm to analyse the worst case.

Input: User needs to choose sorting algorithm to run from choices corresponding to each sorting algorithm. Also, there is a choice to run all algorithms at the same time as well.

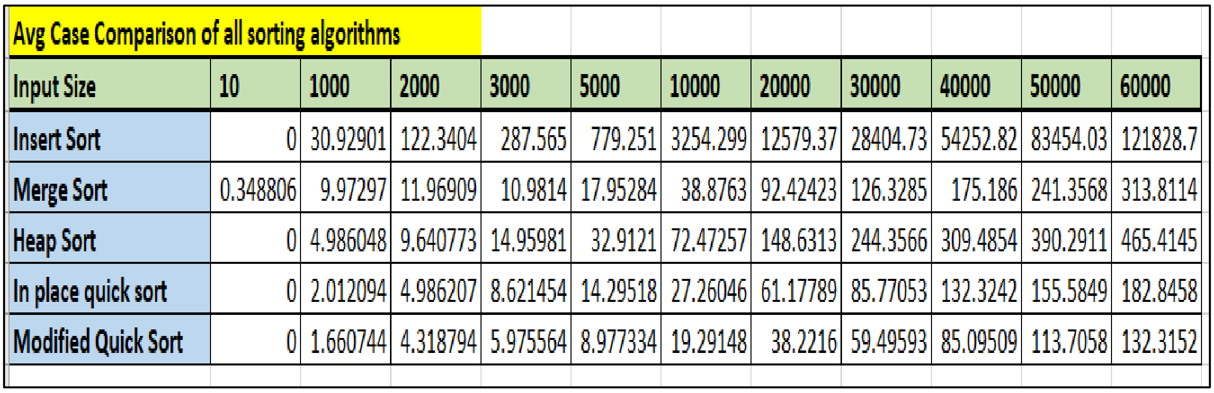
Output: There will be 3 lists for corresponding algorithm in the output which will show execution times for average, already sorted and reverse sorted case corresponding to the input sizes.

Complexity Analysis, Comparison and Graphs

Average case for all algorithms:

Insertion Sort	ϴ(n^2)
Merge Sort	ϴ(nlogn)
Heap Sort	ϴ(nlogn)
In place Quick Sort	ϴ(nlogn)
Modified Quick sort	ϴ(nlogn)

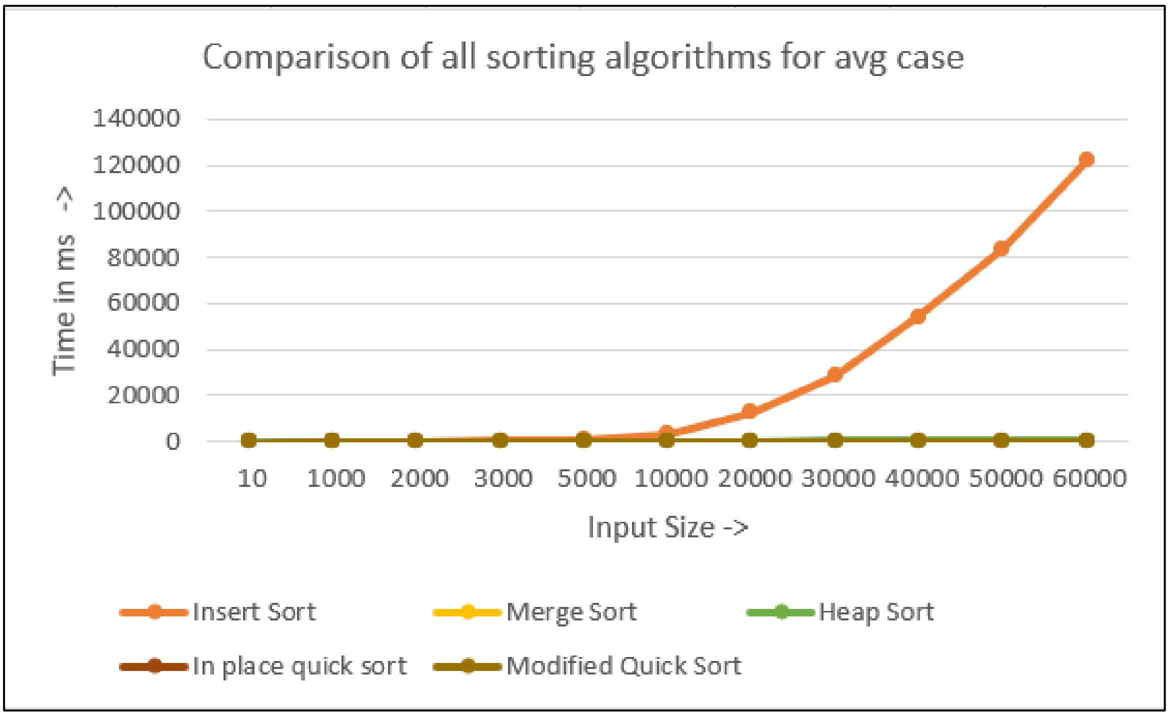
Time Comparisons (in ms):



Avg Case Comparison of all sorting algorithms

Input Size	10	1000	2000	3000	5000	10000	20000	30000	40000	50000	60000
Insert Sort	0	30.92901	122.3404	287.565	779.251	3254.299	12579.37	28404.73	54252.82	83454.03	121828.7
Merge Sort	0.348806	9.97297	11.96909	10.9814	17.95284	38.8763	92.42423	126.3285	175.186	241.3568	313.8114
Heap Sort	0	4.986048	9.640773	14.95981	32.9121	72.47257	148.6313	244.3566	309.4854	390.2911	465.4145
In place quick sort	0	2.012094	4.986207	8.621454	14.29518	27.26046	61.17789	85.77053	132.3242	155.5849	182.8458
Modified Quick Sort	0	1.660744	4.318794	5.975564	8.977334	19.29148	38.2216	59.49593	85.09509	113.7058	132.3152

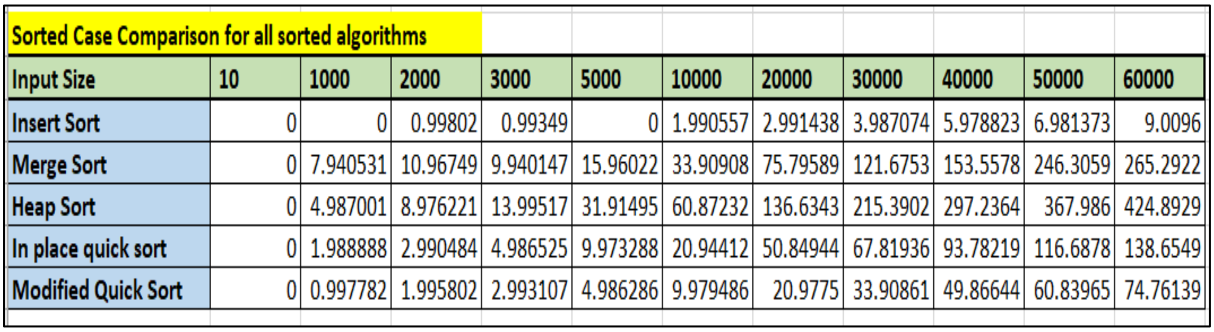
Graph:



Already Sorted Case (Best Case):

Insertion Sort	Ω (n)
Merge Sort	Ω (nlogn)
Heap Sort	Ω (nlogn)
In place Quick Sort	Ω (nlogn)
Modified Quick sort	Ω (nlogn)

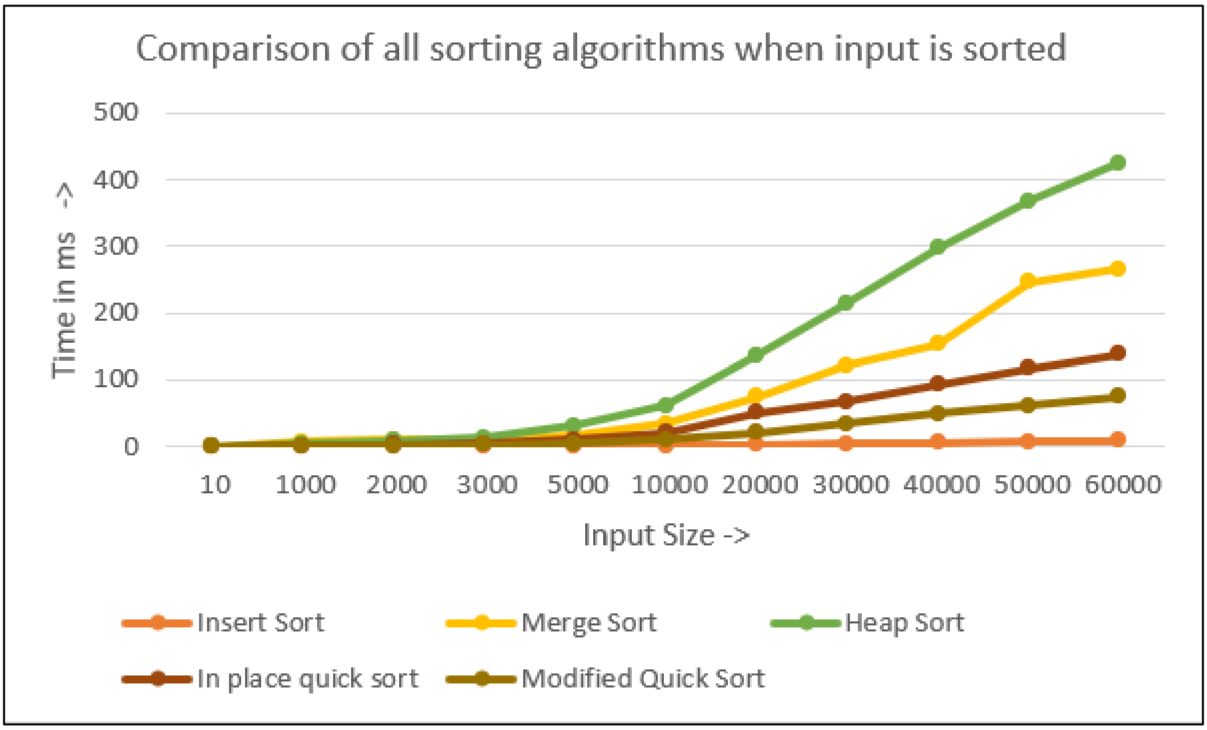
Time Comparisons (in ms):



Sorted Case Comparison for all sorted algorithms

Input Size	10	1000	2000	3000	5000	10000	20000	30000	40000	50000	60000
Insert Sort	0	0	0.99802	0.99349	0	1.990557	2.991438	3.987074	5.978823	6.981373	9.0096
Merge Sort	0	7.940531	10.96749	9.940147	15.96022	33.90908	75.79589	121.6753	153.5578	246.3059	265.2922
Heap Sort	0	4.987001	8.976221	13.99517	31.91495	60.87232	136.6343	215.3902	297.2364	367.986	424.8929
In place quick sort	0	1.988888	2.990484	4.986525	9.973288	20.94412	50.84944	67.81936	93.78219	116.6878	138.6549
Modified Quick Sort	0	0.997782	1.995802	2.993107	4.986286	9.979486	20.9775	33.90861	49.86644	60.83965	74.76139

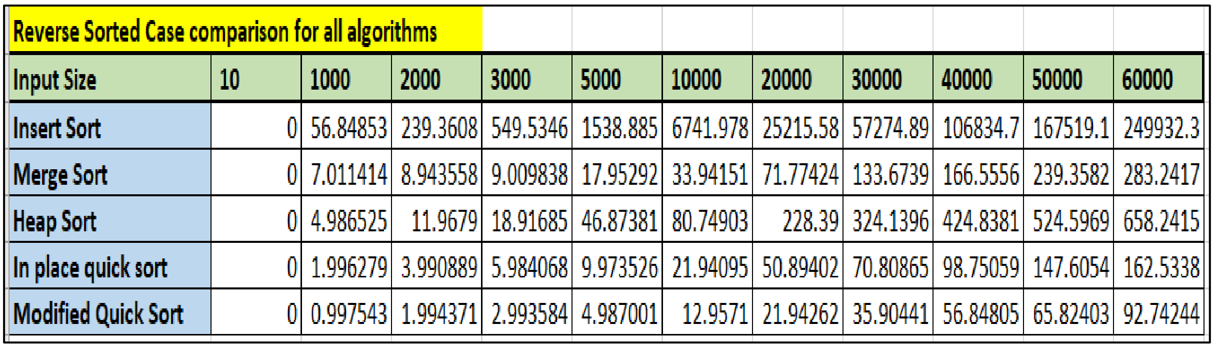
Graph:



Reverse Sorted Case (Worst Case)

Insertion Sort	O (n)
Merge Sort	O (n logn)
Heap Sort	O (n logn)
In place Quick Sort	O (n logn)
Modified Quick sort	O (n logn)

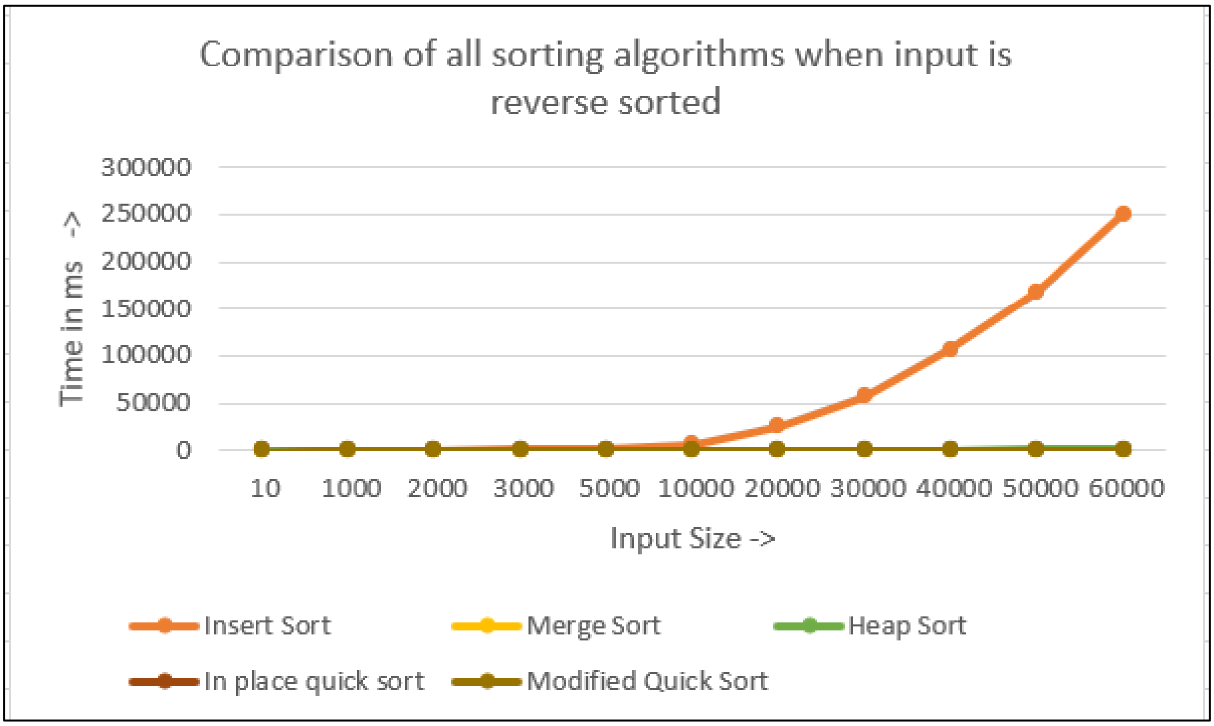
Time Comparisons (in ms):



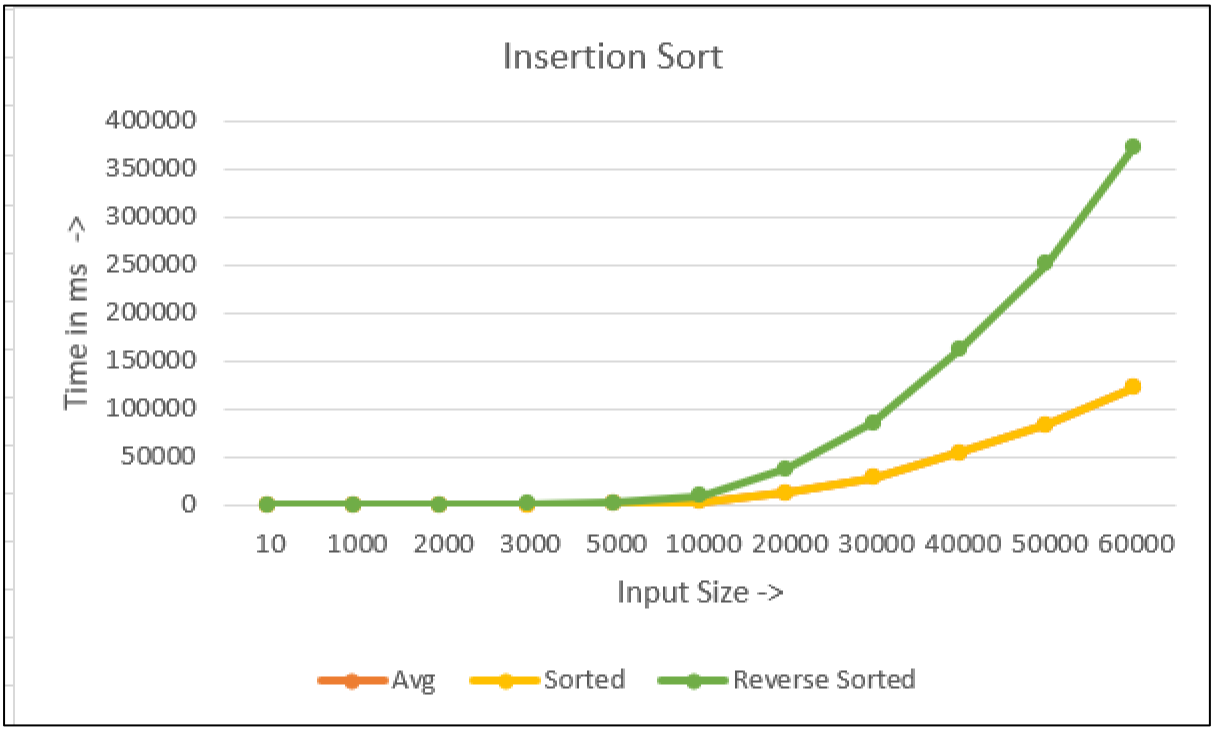
Reverse Sorted Case comparison for all algorithms

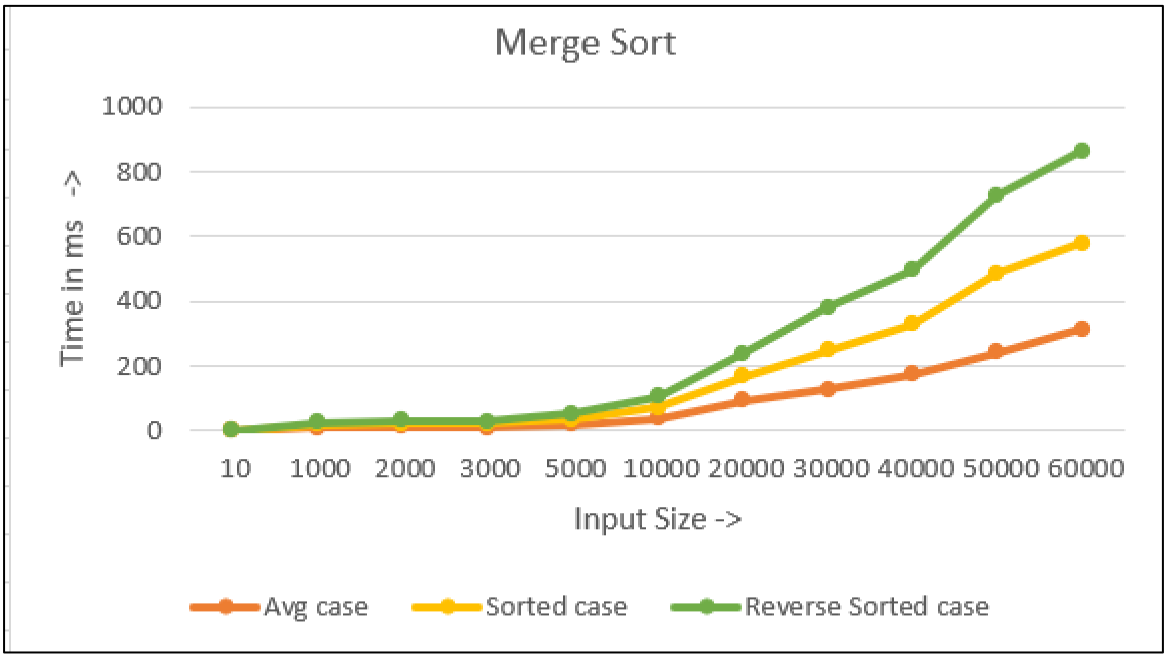
Input Size	10	1000	2000	3000	5000	10000	20000	30000	40000	50000	60000
Insert Sort	0	56.84853	239.3608	549.5346	1538.885	6741.978	25215.58	57274.89	106834.7	167519.1	249932.3
Merge Sort	0	7.011414	8.943558	9.009838	17.95292	33.94151	71.77424	133.6739	166.5556	239.3582	283.2417
Heap Sort	0	4.986525	11.9679	18.91685	46.87381	80.74903	228.39	324.1396	424.8381	524.5969	658.2415
In place quick sort	0	1.996279	3.990889	5.984068	9.973526	21.94095	50.89402	70.80865	98.75059	147.6054	162.5338
Modified Quick Sort	0	0.997543	1.994371	2.993584	4.987001	12.9571	21.94262	35.90441	56.84805	65.82403	92.74244

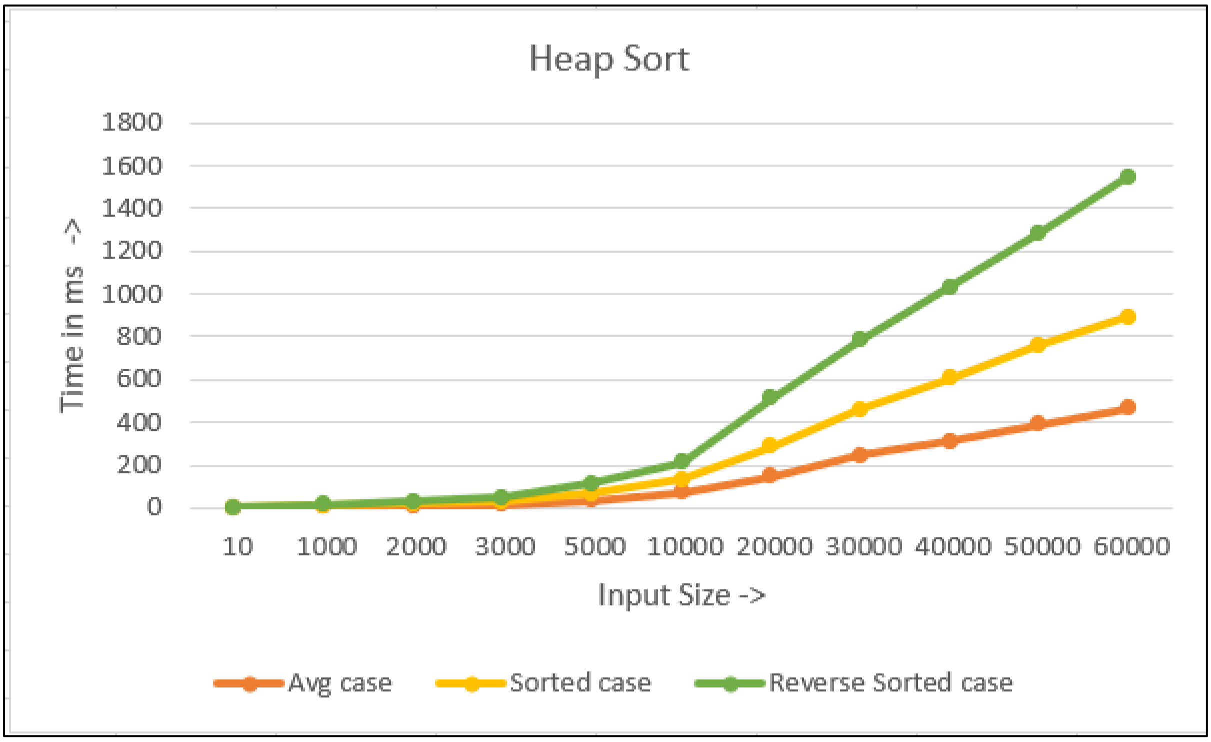
Graph:

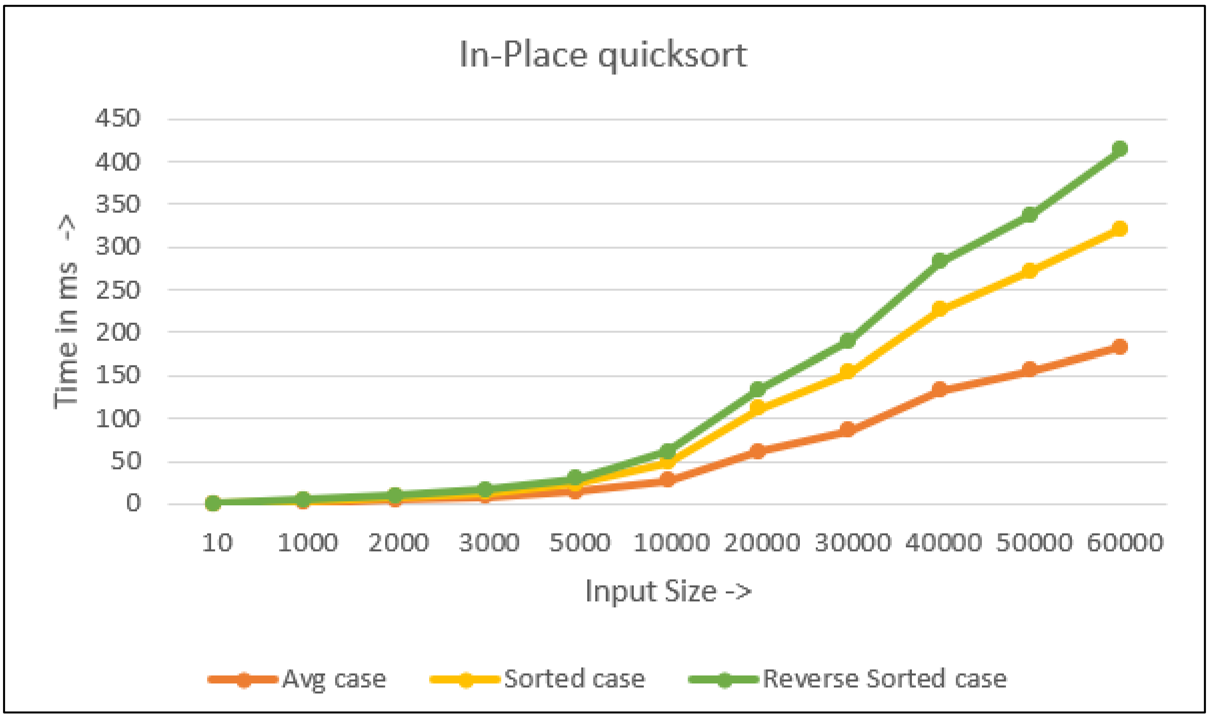


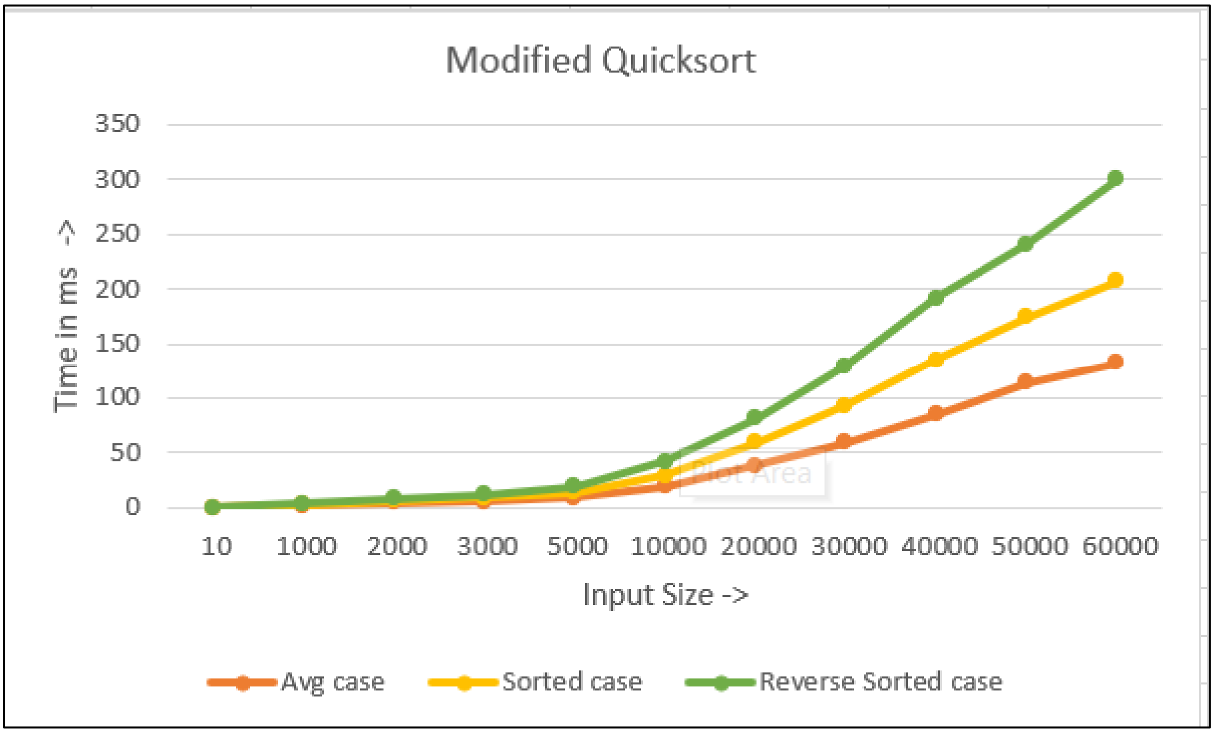
Individual Sorting Algorithms for different cases Graphs






Heap Sort
Time in ms ->
0
200
400
600
800
1000
1200
1400
1600
1800
10
1000
2000
3000
5000
10000
20000
30000
40000
50000
60000
Input Size ->
Avg case
Sorted case
Reverse Sorted case


In-Place quicksort
Time in ms ->
0
50
100
150
200
250
300
350
400
450
10
1000
2000
3000
5000
10000
20000
30000
40000
50000
60000
Input Size ->
Avg case
Sorted case
Reverse Sorted case


Modified Quicksort
Time in ms ->
350
300
250
200
150
100
50
0
10
1000
2000
3000
5000
10000
20000
30000
40000
50000
60000
Input Size ->
Avg case
Sorted case
Reverse Sorted case