Algorithms and Data Structures
ITCS6114
Comparison Based Sorting Algorithms

Project Report

Submitted By:
Sai Harika Paluri(801151378)
Utkarsha Gurjar(801149356)
Mohammad Saif(801133807)

INTRODUCTION

We are using Java to implement various sorting techniques. We have implemented each sorting algorithm as a separate class file. We use the same dataset to compare against different sorting techniques. The Main class contains the driver function to run all algorithms.

There are two options in the Main class:
1> Run all the algorithms from input size [1000, 2000, 3000, 5000, 10000,20000,30000,40000, 50000] including the special cases such as reversed input array and sorted input array.
2> For each input size and sorting technique chosen we display the execution times for sorting as well as for special cases.

The execution times, name of the technique (where the prefix of the sorting technique name such as random, sorted and reversed indicate the kind of input array) and input sizes for the above choices are saved in csv files that are ExecutionTimes.csv and OneByOneExecution.csv .Graphs are generated from these csv files for analysis using Python.

We have implemented different sorting algorithms such as Insertion sort, Heap sort, Merge sort, In-place Quick sort and Modified quick sort. There is a special case for Modified quick sort. If the input size is lesser than 15 then, it will sort using Insertion sort.

The analysis of each algorithm is shown in this report along with its run time and output is given below.

INSERTION SORT:

Complexity Analysis:

Time complexity:

Best Case: If the array is already sorted.

while (j >= 0 && temp <= a[j])

if temp>a[j] for the first time the while loop is run when j=k-1. Insertion sort performs two operations: it scans and compares through the list and swaps elements if they are out of order. Each operation contributes to the running time of the algorithm. If the input array is already in sorted order, insertion sort compares O(n) elements and performs no swaps. Therefore, in the best case, insertion sort runs in O(n) time.

Worst Case: If the input array elements are reversely sorted.

while (j >= 0 && temp <= a[j])

temp <= a[j] is true always in while loop. To insert the last element, we need at most n-1 comparisons and at most n-1 swaps. The number of operations needed to perform insertion sort is therefore: 2 * (1+2+… +n-2+n-1). To calculate the recurrence relation for this algorithm, use the following summation: $k=1\Sigma n\ t = (n(n+1))/2$.

When analysing algorithms, the **Average case** often has the same complexity as the worst case. So insertion sort, on average, takes $O(n^2)$ time.

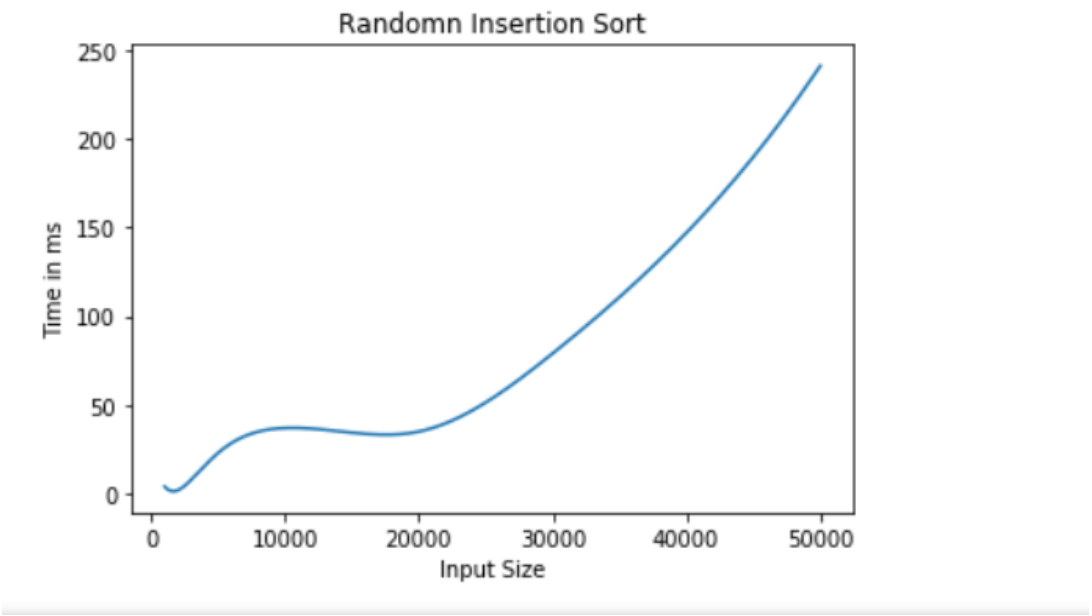
Space complexity: O(1)

Data structures chosen: Array

CODE SNIPPET:

```
public long insertionSort(int[] arr) {
    // TODO Auto-generated method stub
    long startTime = System.currentTimeMillis();
    for(int i=1;i<arr.length;i++) {

        int key=arr[i];
        int j=i-1;
        while(j>=0 && arr[j]>key) {
            arr[j+1]=arr[j];
            j=j-1;
        }
        arr[j+1]=key;
    }
    long stopTime = System.currentTimeMillis();
      long elapsedTime = stopTime - startTime;
```

GRAPH:



Input Size	1000	2000	4000	5000	10000	20000	30000	40000	50000
Time in ms	4.5	11	9.5	12	20	35	94.5	152.5	223

OUTPUT

```
<terminated> Main [Java Application] C:\Program Files\Java\jdk-12.0.1\bin\javaw.exe (Oct 15, 2020, 6:22:19 PM)
Choose your option 1.Run All cases all sorting algorthms with special cases 2.Choose each input size and algorithm
2
Choose Algo 1.Insertion 2.Heap 3.Quick Sort 4.Merge 5.Mod quick sort
1
Enter size
1000
Insertion Sort
Input Array n = 1000
233 44 986 621 559 670 282 428 345 639 522 973 36 768 935 483 496 496 185 332 352 778 467 975 255 329 522 164 758 35 906 330 162 744
Sorted
2 5 5 8 9 11 14 14 14 14 15 16 17 17 17 17 18 19 19 19 20 21 21 21 21 22 22 22 24 24 24 26 27 29 31 32 33 33 33 34 34 35 36 37 37 37
Time for sorting = 4
Time when input array is sorted = 0
Time when input array is reverse=2
```

MERGE SORT:
Complexity Analysis:

Time complexity:
The height h of the merge-sort tree is O(log n).
At each recursive call we divide in half the sequence.
The overall amount of work done at the nodes of depth i is O(n).
We partition and merge 2^i sequences of size $n/2^i$.
We make $2^i + 1$ recursive calls.
Thus, the total running time of merge-sort is O(nlog n).

Best case=Worst case=Average case= O(nlog n)
Space Complexity: O(n)

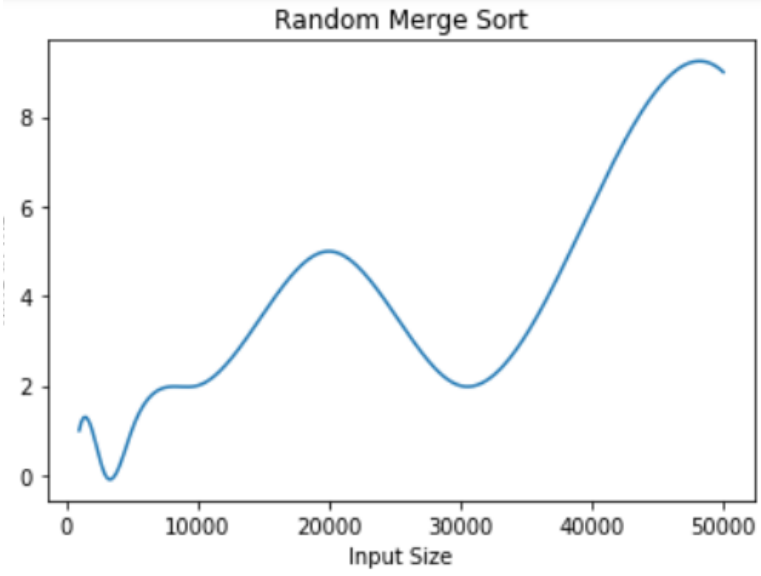
Data structures chosen: Array

CODE SNIPPET:

```
left = partition(left);
right = partition(right);
int[] result = new int[A.length];
result = merge(left, right);
return result;
}
```

Input Size	1000	2000	4000	5000	10000	20000	30000	40000	50000
Time in ms	1	1	1	1	1.5	4.5	5	6.5	9

GRAPH:



OUTPUT

```
<terminated> Main [Java Application] C:\Program Files\Java\jdk-12.0.1\bin\javaw.exe (Oct 15, 2020, 6:27:27 PM)
Choose your option 1.Run All cases all sorting algorthms with special cases 2.Choose each input size and algorithm
2
Choose Algo 1.Insertion 2.Heap 3.Quick Sort 4.Merge 5.Mod quick sort
4
Enter size
5000
Merge Sort
Input Array n = 5000
Input Array
1136 25 2991 433 1272 4925 4332 1729 338 4528 709 1066 1849 3768 980 1523 1605 2173 3080 3910 326 230 4485 4605 569 4058 3030 3264 1564 3303 4389 1709 491 3874 1307
Sorted
1 1 4 5 5 11 11 12 12 12 15 16 22 22 23 24 25 27 28 28 30 31 32 35 36 37 37 37 38 38 39 39 40 40 41 42 45 45 45 45 45 47 47 48 48 48 49 49 51 53 53 54 55 58 59 59 59
Time for sorting = 2
Time when input array is sorted = 1
Time when input array is reverse = 1
```

HEAP SORT:

Complexity Analysis:

Heap sort has a running time of O(nlog n).

To build the max-heap from the unsorted array of elements, it requires O(n) calls to the function heapify(), each of which takes O(log n) time.

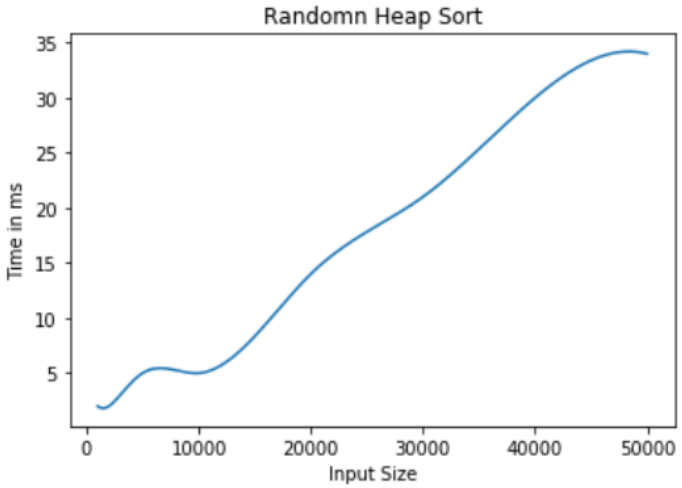
Data structures chosen: Vector

Best case = Average case = Worst case = O(nlog n)

CODE SNIPPET:

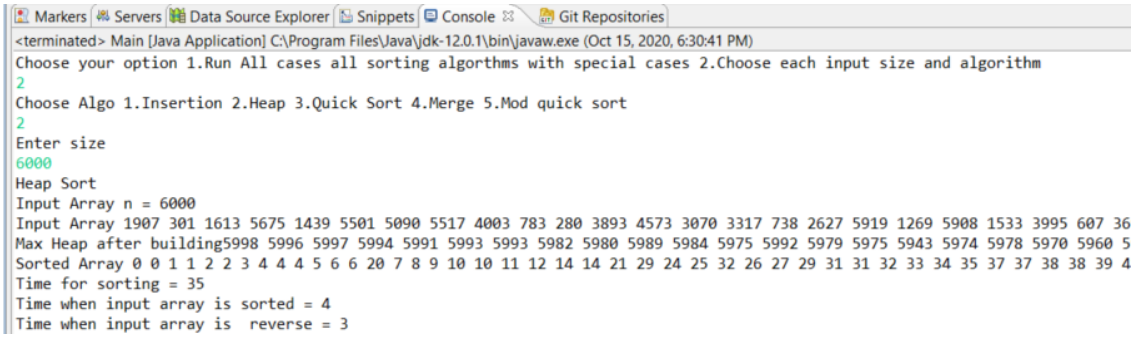
```
    // main function to do heap sort
static Vector<Integer> heapSort(Vector<Integer> vector, int n)
{

    // One by one extract an element from heap
    for (int i=n-1; i>0; i--)
    {
        // Move current root to end
         int temp =vector.get(0);
         vector.set(0,vector.get(i));
         vector.set(i,temp);


        // call max heapify on the reduced heap
        heapify(vector, i, 0);
    }
return vector;
```

GRAPH:



Input Size	1000	2000	4000	5000	10000	20000	30000	40000	50000
Time in ms	2	3	5	5	5	10	18.5	18.5	27

OUTPUT



QUICK SORT:

Complexity Analysis:

Best case:
Cut the array size in half each time.
So, the depth of the recursion is log n.
At each level of the recursion, all the partitions at that level do work that is linear in n.
Hence in the best case, quicksort has time complexity O(log n) * O(n) = O(nlog n)

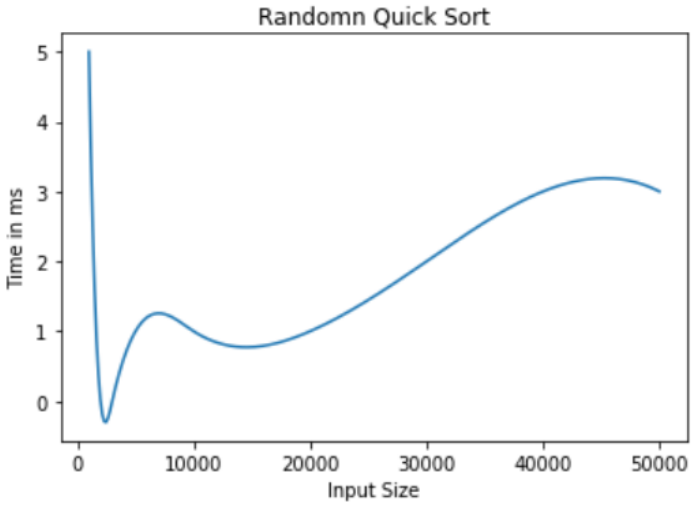
Worst case:
The worst case for quick sort occurs when the pivot is the unique minimum or maximum element.
The running time is proportional to the sum: n + (n - 1) + ... + 2 + 1
Thus, the worst-case running time of quick sort is $O(n^2)$

Average case: O (n log n)
Data structures chosen: Array

CODE SNIPPET:

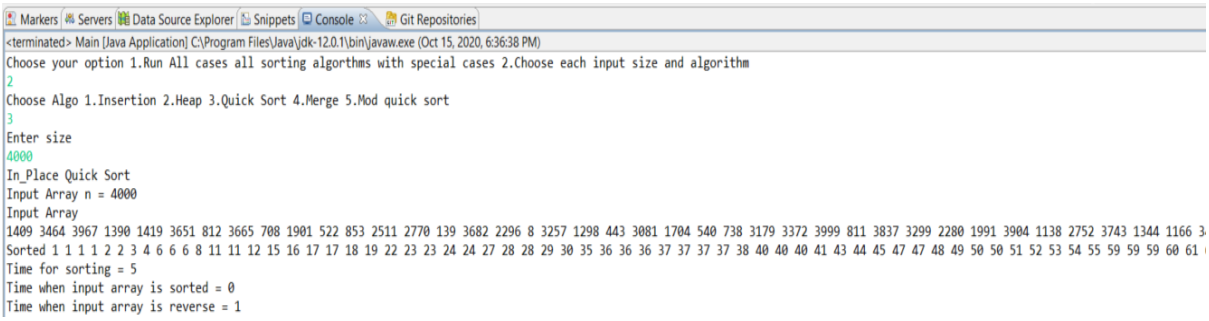
```
while(l<=r)
{
    while(arr[l]<pivot)
          l++;
    while(arr[r]>pivot)
        r--;
    if(l<=r)
    {
       int temp = arr[l];
       arr[l] = arr[r];
       arr[r] = temp;
        l++;
        r--;
    }
}
  if(leftChk<r)
      inPlaceQuickSort(arr, leftChk, r);
  if(l<rightChk)
       inPlaceQuickSort(arr, l, rightChk);
```

GRAPH:



Input Size	1000	2000	4000	5000	10000	20000	30000	40000	50000
Time in ms	0.5	0.5	1	1	1	2	3	3	3

OUTPUT



MODIFIED QUICKSORT:

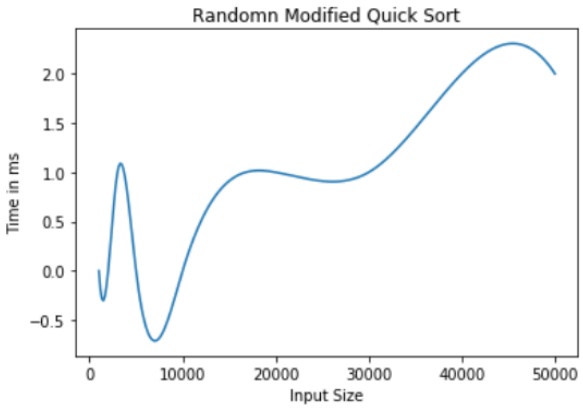
Complexity Analysis:

Best case:
Partition is perfectly balanced. Pivot is always in the middle (median of the array)
$T(n)/n = T(1)/1 + c * \log n$
$T(n)= c*n*\log n + n=$ $O(n \log n)$
Worst case: The pivot is the smallest element, all the time. Partition is always unbalanced.
$T(n)=T(n-1) +c*n$
$T(n-1)=T(n-2) +c*(n-1)$
.
.
.
$T(2)=T(1) +c*2$
$T(n)=T(1) +c* \sum_{i=2}^{n} i=$ $O(n^2)$
Average case:
Assume that each of the sizes for S1 is equally likely.
Hence, time complexity = **O(n log n)**
Data structures chosen: Array

CODE SNIPPET:

```
private static final int CUTOFF = 15;
private static void quicksortmed( int[] a, int first, int last )
{
    if( first + CUTOFF > last )
        insertionSort( a, first, last );
    else
    {
// Sort low, middle, high
int middle = ( first + last ) / 2;
if( a[ middle ]> a[ first ] )
                    swap( a, first, middle );
if( a[ last ]> a[ first ]  )
                    swap( a, first, last );
if( a[last ]> a[ middle ]  )
                    swap( a, middle, last );

// Place pivot at position high - 1
swap( a, middle, last - 1 );
int pivot = a[ last - 1 ];
```

GRAPH:



Input Size	1000	2000	4000	5000	10000	20000	30000	40000	50000
Time in ms	0	0	0	0	0	1	1	1	1

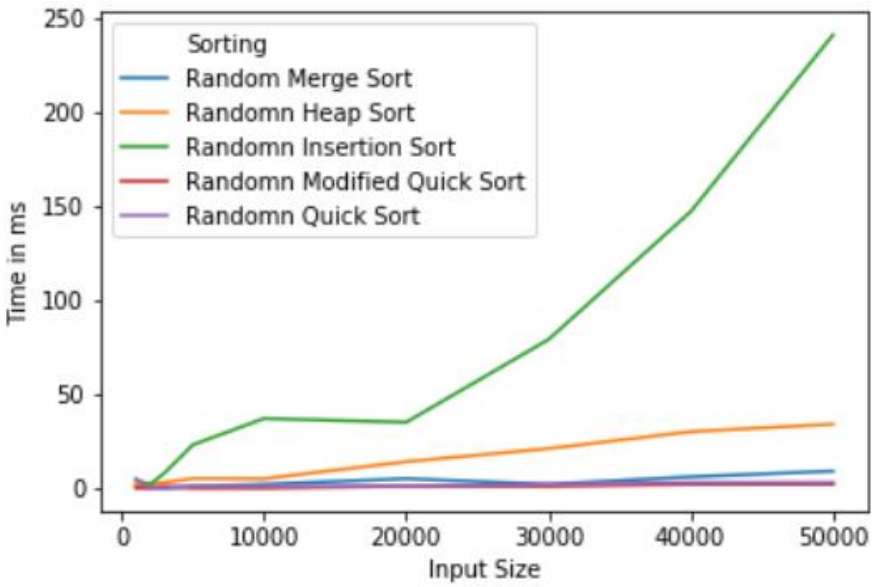
OUTPUT

```
<terminated> Main [Java Application] C:\Program Files\Java\jdk-12.0.1\bin\javaw.exe (Oct 15, 2020, 8:39:13 PM)
2
Choose Algo 1.Insertion 2.Heap 3.Quick Sort 4.Merge 5.Mod quick sort
5
Enter size
5000
Modified Quick Sort
Input Array n = 5000
Input Array  =
3239 1029 2607 462 2387 832 1596 3910 1646 1795 4079 4091 465 2579 2858 457 4559 3655 3300 2763 626 1139 1542 199 4402 1773 2743 1635 2156 64 3948 971
Sorted array  =
0 1 1 4 6 6 7 8 11 11 13 16 16 17 18 18 18 20 20 21 22 22 22 23 26 27 27 28 28 32 33 33 33 33 34 34 35 36 38 38 38 40 40 41 42 42 42 43 43 44 44 46 46
Time for sorting = 1
Time when input array is  sorted = 0
Time when input array is  reverse = 0
Please view OneByOneExecution.csv in your file folder for all the run times
```

INSTRUCTION 1

Comparison of Input size vs Execution time for various sorting algorithms

Input Size	1000	2000	4000	5000	10000	20000	30000	40000	50000
Insertion Sort	4.5	11	9.5	12	20	35	94.5	152.5	223
Merge Sort	1	1	1	1	1.5	4.5	5	6.5	9
Heap Sort	2	3	5	5	5	10	18.5	18.5	27
Quicksort	0.5	0.5	1	1	1	2	3	3	3
Modified Quicksort	0	0	0	0	0	1	1	1	1



As seen above, in the random input array case where the same dataset is used for all algorithms we observe that the algorithms perform similarly except Insertion Sort because of its Time Complexity being $O(n^2)$ .We observe that on an average Modified quick sort performs better than the rest.

Sorting Algorithm	Data Structure	Time Complexity: Best	Time Complexity: Average	Time Complexity: Worst	Space Complexity: Worst
Insertion Sort	Array	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	Array	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Heap Sort	Vector	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Quick Sort	Array	O(nlogn)	O(nlogn)	$O(n^2)$	O(n)
Modified Quicksort	Array	O(nlogn)	O(nlogn)	$O(n^2)$	O(n)

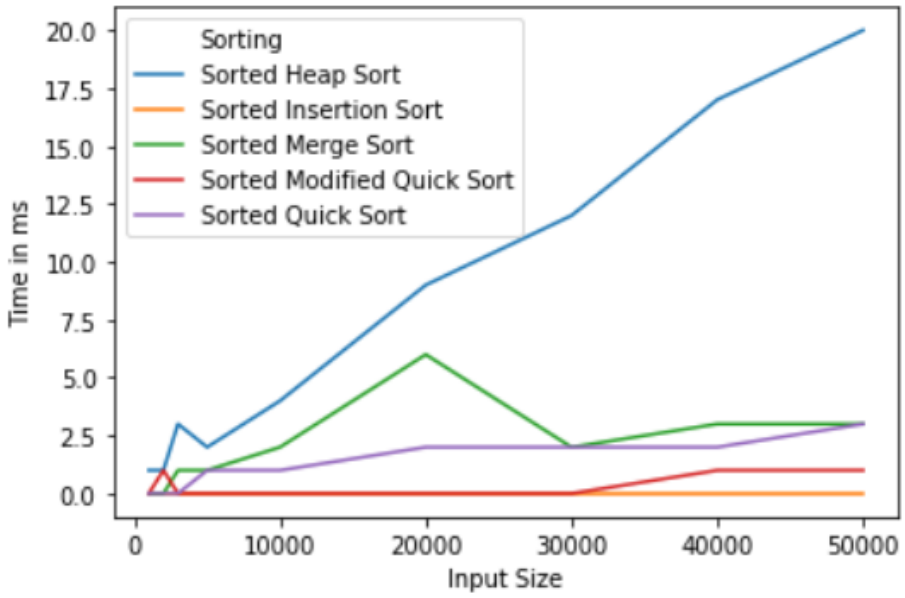
COMPARING DIFFERENT ALGORTHMS ON THE BASIS OF TIME AND SPACE COMPLEXITY

INSTRUCTION 2

CASE A: SORTED INPUT ARRAY

Comparison of Input size vs Execution time for various sorting algorithms

Input Size	1000	2000	4000	5000	10000	20000	30000	40000	50000
Insertion Sort	0	0	0	0	0	0	0	0	0
Merge Sort	0.5	1	1	1.5	6	5.5	6	6.5	7
Heap Sort	0.5	1.5	1.5	2.5	6	7.5	10.5	14	20.5
Quicksort	0	0.5	0.5	0.5	1.5	1.5	1.5	2.5	3
Modified Quicksort	0	0	0	0	0	0.5	1	1	1

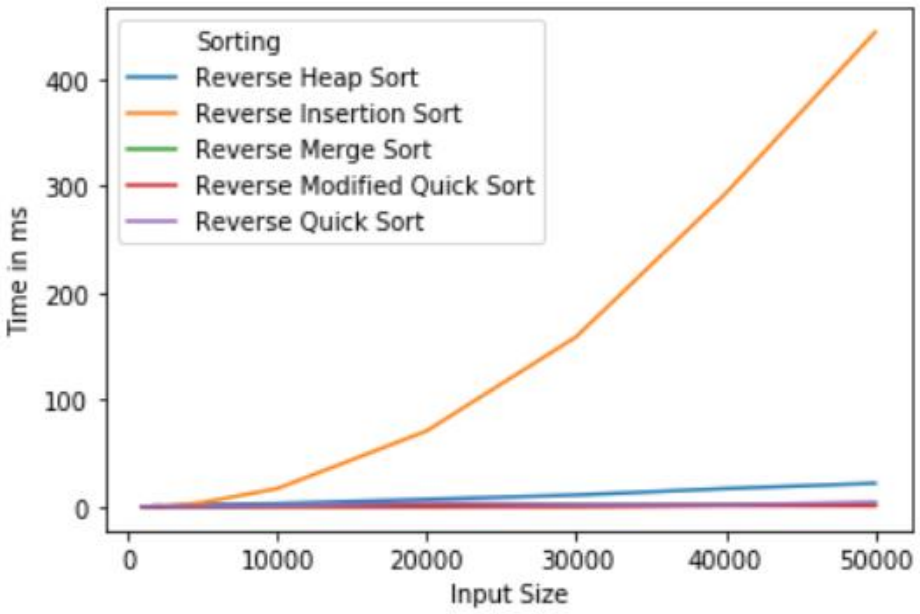


As observed here Insertion Sort has the best performance as the array is already sorted and no insertions have to take place, while the others perform similarly since they have a time complexity of O (nlog n) for best case.

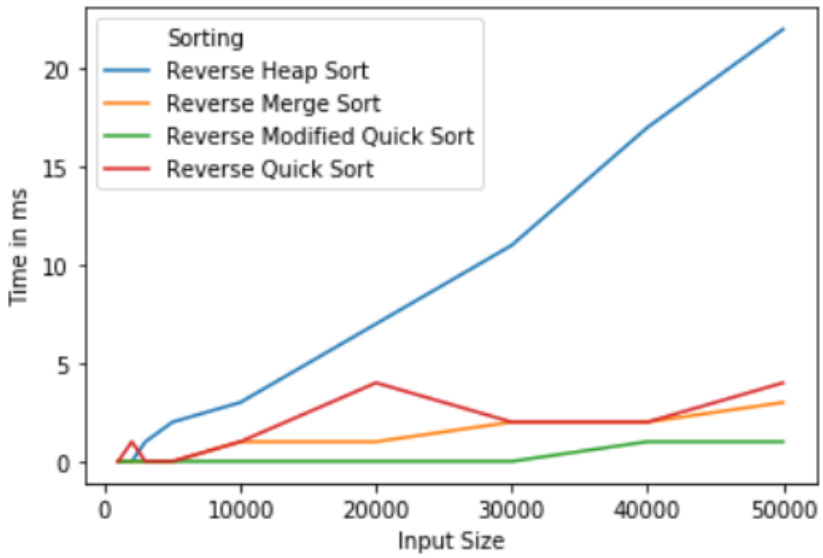
CASE B: REVERSE SORTED INPUT

Comparison of Input size vs Execution time for various sorting algorithms.

Input Size	1000	2000	4000	5000	10000	20000	30000	40000	50000
Insertion Sort	1	1.5	3	7	21.5	70.5	159.5	286.5	461.5
Merge Sort	0	0.5	0	1	2	3.5	2.5	3	3
Heap Sort	0.5	1	1	2.5	3	8	10	13.5	17.5
Quicksort	0.5	0	0.5	0.5	1	1	2	2.5	3
Modified Quicksort	0	0	0	0	0	0	0.5	0.5	1



In a worst-case scenario Insertion sort performs the worst of all of them due to the array being completely reverse and its $O(n^2)$ time complexity. We remove Insertion sort from the graph to better compare other algorithms



Here we get a better picture with the algorithms performing similarly within a margin of error and modified quick sort being the fastest.