**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])

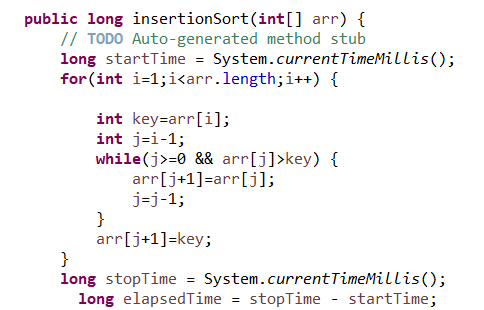
temp <= a[j] is true always in while loop. To insert the last element, we need at most n-1 comparisons and at most n-1 swaps. The number of operations needed to perform insertion sort is therefore: 2 \* (1+2+… +n-2+n-1). To calculate the recurrence relation for this algorithm, use the following summation: k=1Σn 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(n2) time.

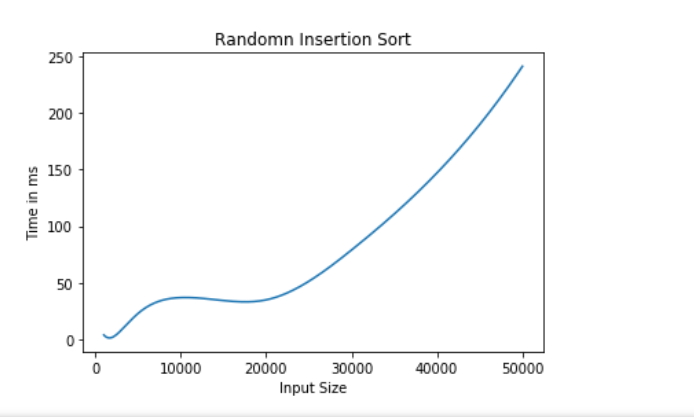
**Space complexity**: O(1)

Data structures chosen: Array

**CODE SNIPPET:**

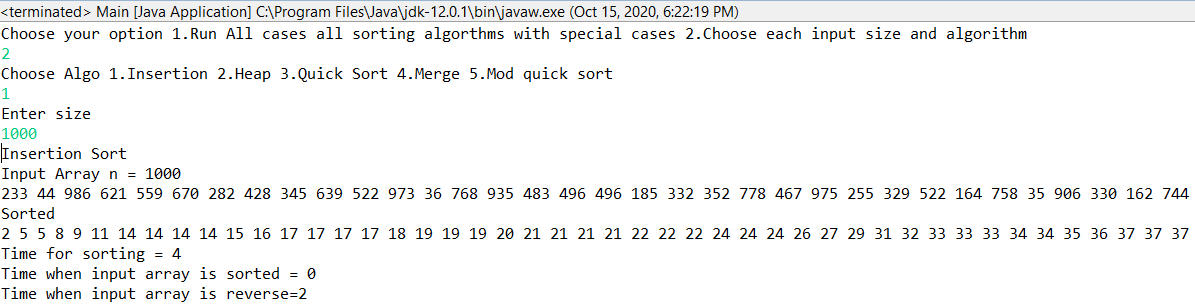
****

**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**



**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 2i sequences of size n/2i.

We make 2i + 1 recursive calls.

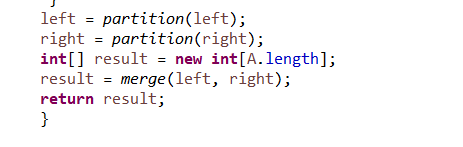
Thus, the total running time of merge-sort is O(nlog n).

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

**Space Complexity**: O(n)

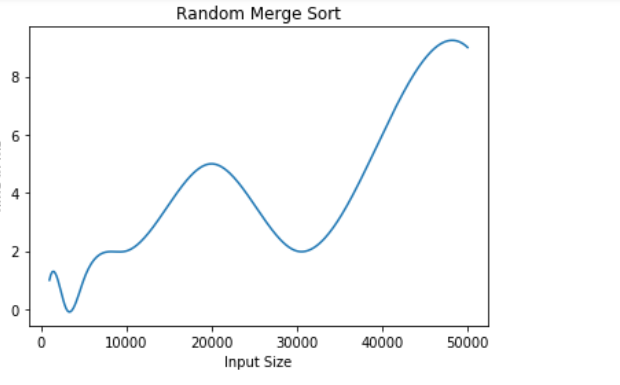
Data structures chosen: Array

**CODE SNIPPET:**

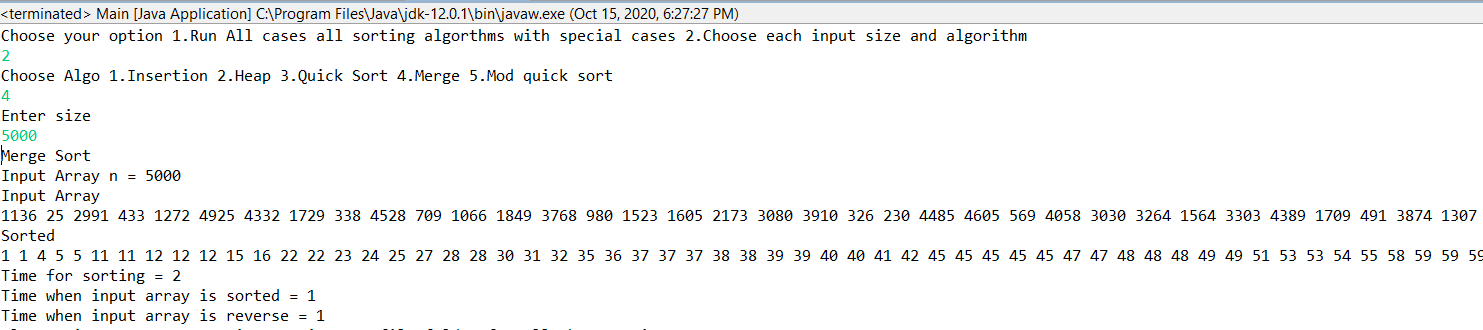
****

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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**

****

**HEAP SORT:**

**Complexity Analysis:**

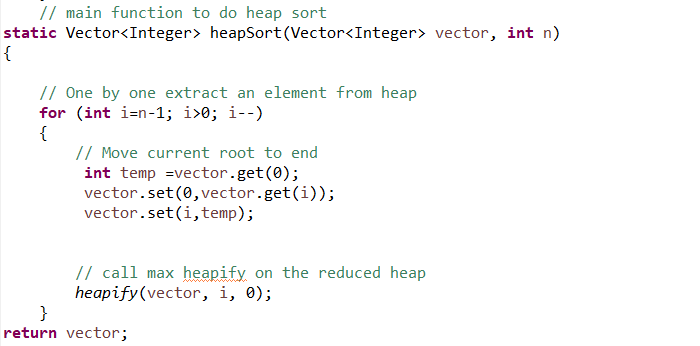
Heap sort has a running time of O(nlog n).

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.

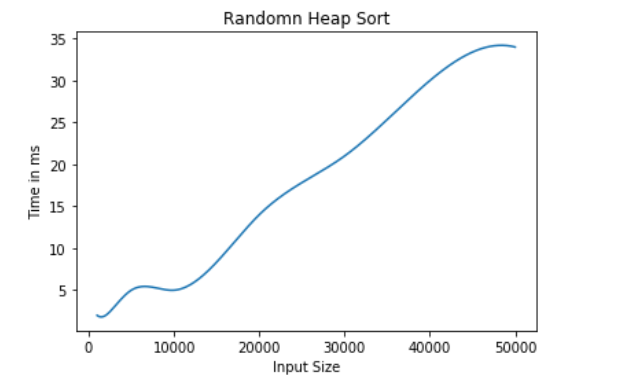
Data structures chosen: Vector

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

**CODE SNIPPET:**

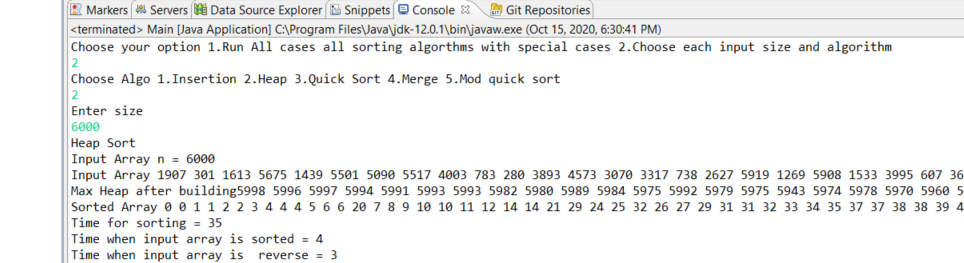
****

**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.

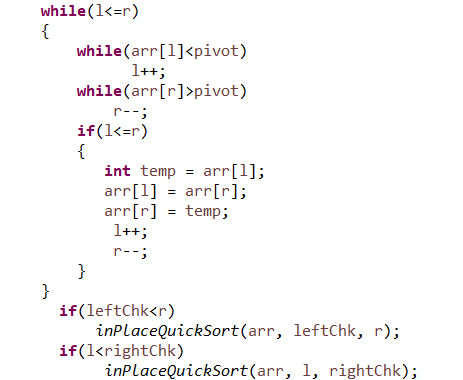
The running time is proportional to the sum: n + (n - 1) + ... + 2 + 1

Thus, the worst-case running time of quick sort is O(n2)

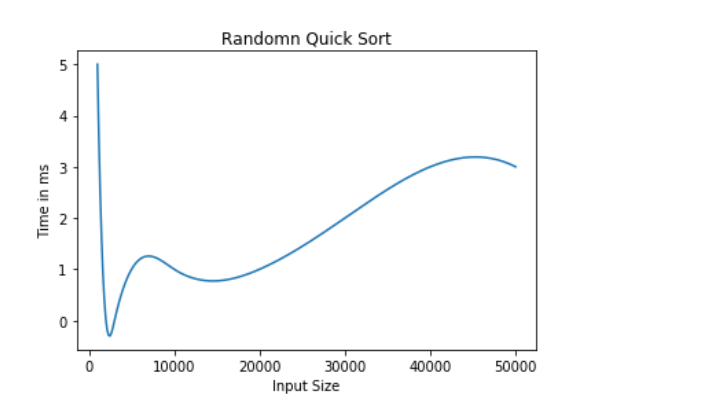
**Average case: O (n log n)**

Data structures chosen: Array

**CODE SNIPPET:**

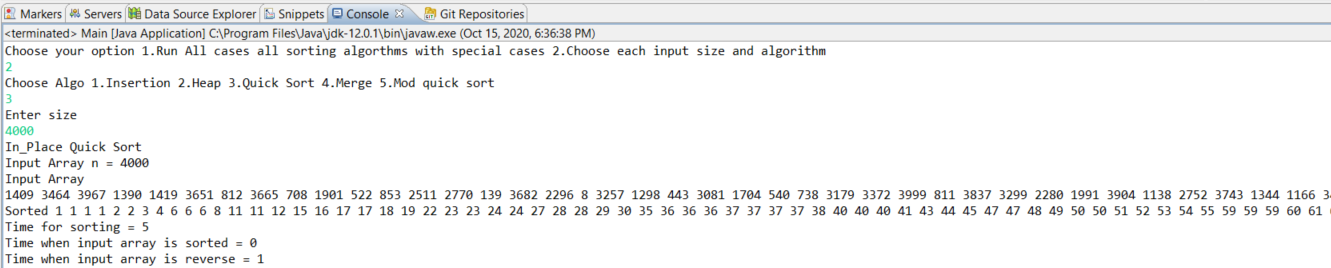


**GRAPH:**



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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\* i=2Σn i= O(n2)

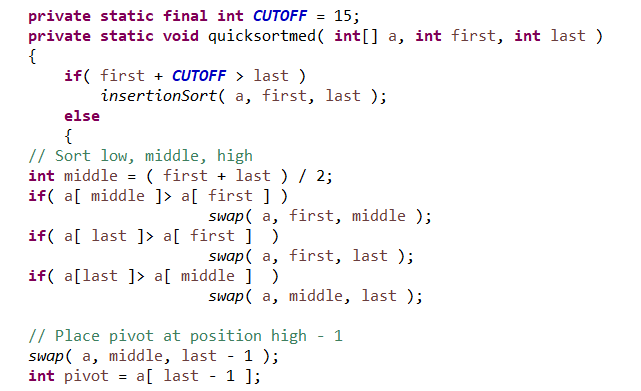
**Average case:**

Assume that each of the sizes for S1 is equally likely.

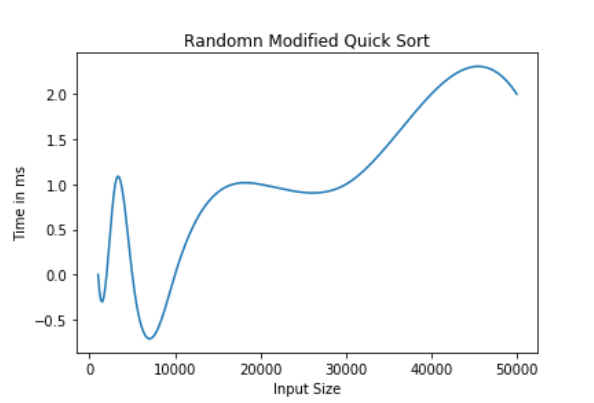
Hence, time complexity = **O(n log n)**

Data structures chosen: Array

**CODE SNIPPET:**

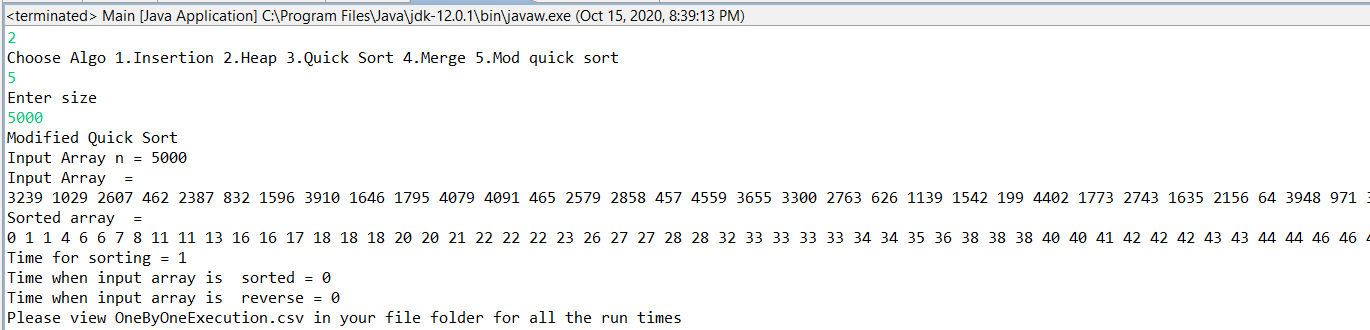


**GRAPH:**

****

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

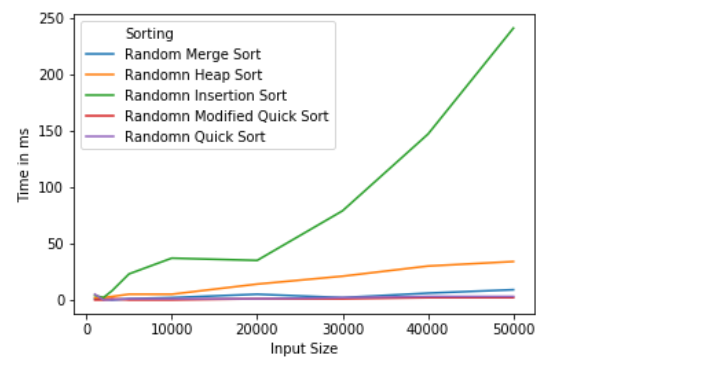
**OUTPUT**

****

**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(n2) .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(n2) | O(n2) | 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(n2) | O(n) |
| Modified Quicksort | Array | O(nlogn) | O(nlogn) | O(n2) | O(n) |

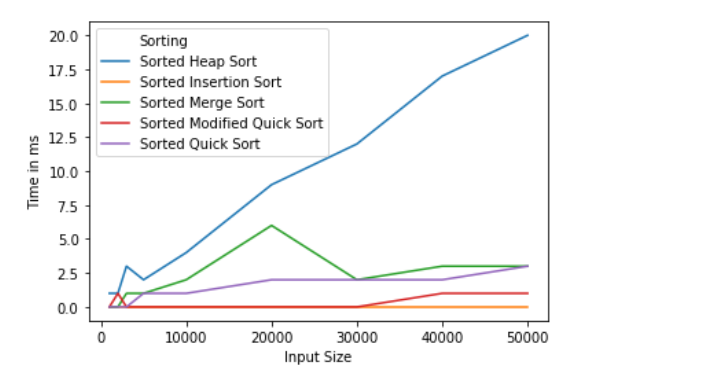
**COMPARING DIFFERENT ALGORTHMS ON THE BASIS OF TIME AND SPACE COMPLEXITY**

**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 |

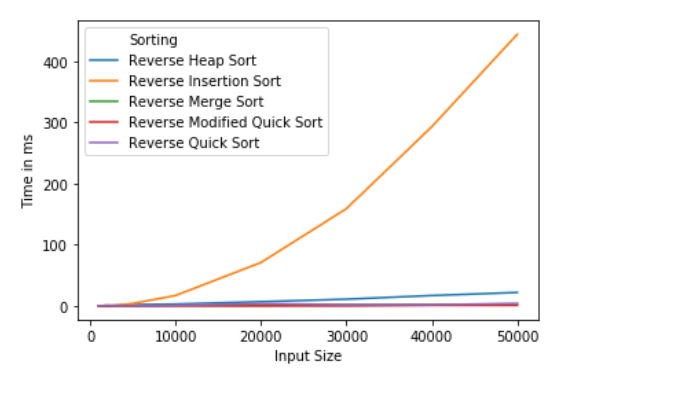


As observed here Insertion Sort has the best performance as the array is already sorted and no insertions have to take place, while the others perform similarly since they have a time complexity of O (nlog n) for best case.

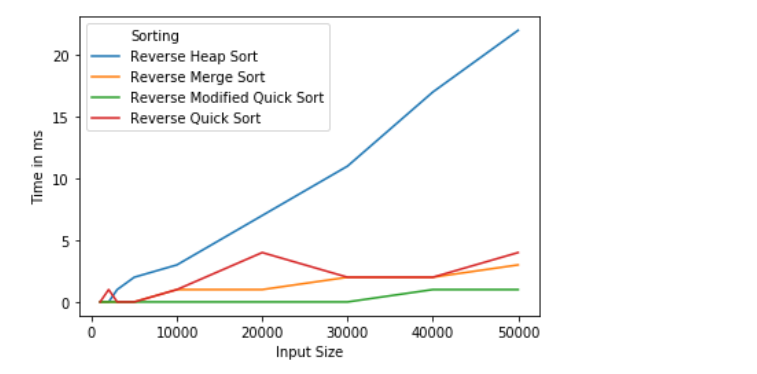
**CASE B: REVERSE SORTED INPUT**

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(n2) 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.