Programming Assignment 1
Comparison-based Sorting Algorithms

Algorithms and Data Structures
ITCS 6114/8114, Fall 2019

Pooja Nikhare (801136236)

Rahul Sundaresan (801151472)

Programming Language Used :

Python

Data Structures used:

List - Variable length array.
Heap - Implemented using python lists.
Dictionary - To store the set of keys and their respective values.
Stack - used internally in recursive function calls.

Complexity Analysis

Sorting Algorithms :

1. Insertion Sort
2. Merge Sort
3. Heap Sort
4. Quick Sort (Taking first element as pivot)

5. Modified Quick Sort
    - Using median of First , middle, last element as pivot
    - Using insertion sort for input size <=10

Test Cases Observed :

1. Unsorted Arrays
2. Sorted Arrays
3. Reversely Sorted Arrays

The time complexity of the above algorithms are:

Algorithm	Best Case	Average Case	Worst Case
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Modified Quick Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))

Complexity analysis for unsorted arrays:

Insertion Sort:

When the input array is unsorted, insertion sort algorithm has to compare each element with all the elements before it. This results in a time complexity of O(n^2) . It takes space complexity of O(1) as no extra space is needed and insertion sort is a stable sorting technique i.e it maintains the order of elements.

Merge Sort:

When the input array is unsorted, merge sort divides the array into sub arrays till the sub array consists of single elements and merges them. Thus the time complexity is dependent on the merging of multiple sorted subarrays. The number of times the subarrays are divided is in the order of the height of the tree. The height of the tree is O(log(n)). The merging of two sub arrays takes O(n) time. Therefore the total running time is O(n*log(n)) .

The time complexity is O(n log(n)) . As a temporary array is used to merge the list space complexity is O(n). Merge sort is a stable sorting technique .

Heap Sort:

In heap sort, the elements are inserted one at a time into the heap and the heap property is restored. This restoration of heap property involves a minimum of log(n) comparisons for one element insertion. Thus, for n elements, the time complexity is O(n log(n)). It takes space complexity of O(1) as no extra space is needed .

Quick Sort :

In quick sort, the elements are compared with a pivot element and elements are partitioned into arrays that are greater than or less than the element. In our implementation the pivot is chosen as the first or leftmost element. With unsorted arrays, the partitions may have variable number of elements with each iteration. The partitioned array is executed in O(n) time as each element of the array needs to be compared with the pivot. The number of times the array gets partitioned is O(log(n)). Thus the total time required is O(nlog(n)) in the best and average case. In the worst case scenario where each partition results in 0 elements in one array and n-1 elements in the other (which occurs in sorted or reversely sorted arrays), the number of partitions is O(n) and the total running time becomes O(n^2). In place quicksort takes space complexity of O(1) as no extra space is needed .

Modified Quick Sort

- median-of-three as pivot:

In quick sort with median-of-three as pivot, the elements are compared with a pivot element which is the median of the first, last and middle elements of the array and partitioned into subarrays which are less than or greater than the pivot. With unsorted arrays, the partitions may have variable number of elements with each iteration. The time complexity is O(nlog(n))

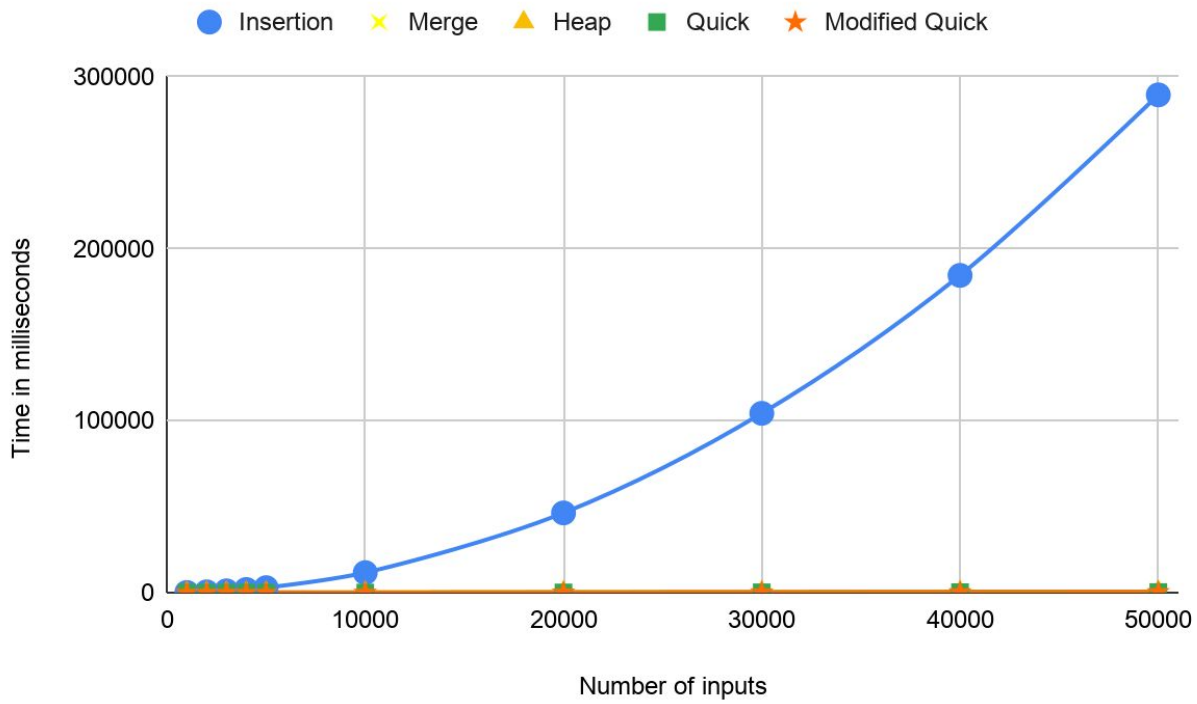
- insertion sort for input size <=10:

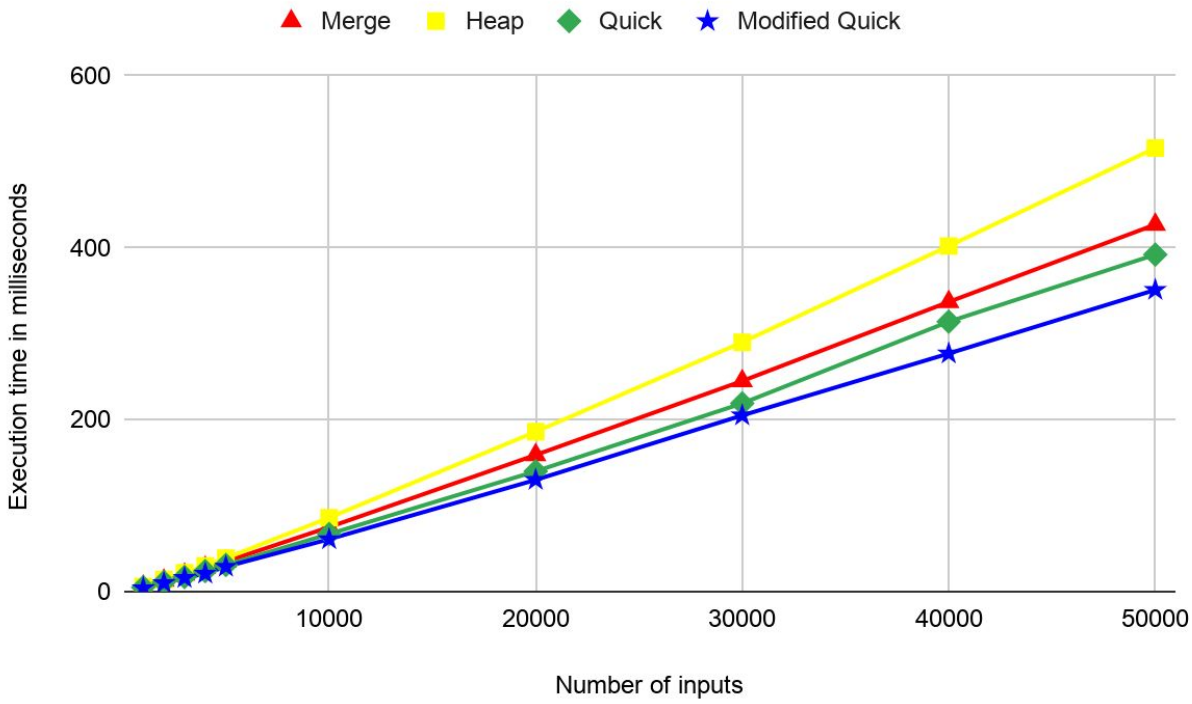
For input size <=10, since insertion sort is used, the time complexity is O(n^2). For input size >=10, regular quicksort is used and the time complexity is time complexity is O(nlog(n)). Modified quicksort takes space complexity of O(1) as no extra space is needed.

Results

Note: execution time is measured in milliseconds (rounded Up)

Number of inputs	Insertion	Merge	Heap	Quick	Modified Quick
1000	114	6	6	5	4
2000	464	13	14	11	10
3000	1045	20	22	17	16
4000	1846	28	30	24	21
5000	2901	35	39	31	29
10000	11643	75	86	67	61
20000	46366	159	186	140	130
30000	104278	245	290	219	205
40000	184516	337	402	314	277
50000	289507	427	516	392	351





Complexity analysis for sorted arrays:

Insertion Sort:

When the input array is sorted, insertion sort algorithm stops with a single comparison of the element and its previous element and no swapping is required. Thus the total number of comparisons is directly proportional to the number of elements. This results in best case for insertion sort and it's time complexity is $\Omega(n)$.

Merge Sort :

When the input array is sorted, merge sort divides the array into sub arrays till the sub array consists of single elements that still takes O(log n) and merges them in O(n) time , thus time complexity is cannot be better than $\Omega(n \log n)$. Efficiency of merge sort increases as the merging of arrays takes place in already sorted array.

Heap Sort:

In heap sort, even if the elements are sorted, building of heap property minimum takes O(log(n)) comparisons for each element insertion or deletion. To form a sorted array, whenever it removes a minimum element, each removal needs heapify which means we need to restore the heap property which will take minimum log n. For n elements, the time complexity will be O(n log(n)). Thus, sorting using heap sort for n elements, the time complexity remains $\Omega(n \log(n))$.

Quick Sort :

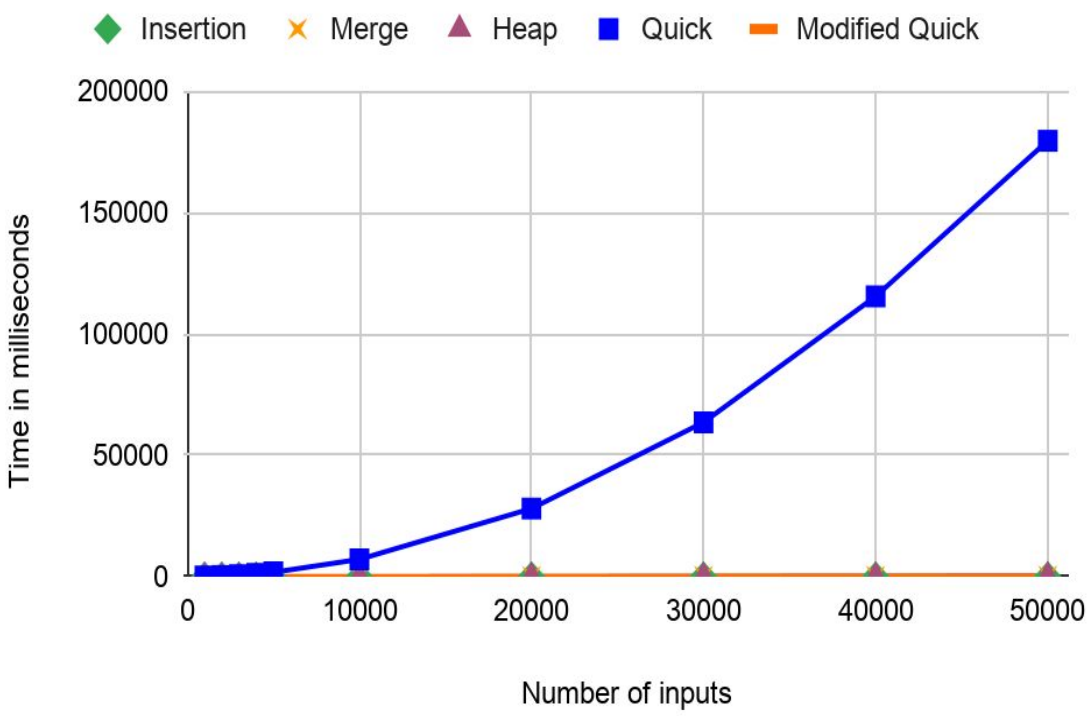
In quick sort, the elements are compared with a pivot element and elements are partitioned into arrays that are greater than or less than the element. In our implementation the pivot is chosen as the first element. With sorted arrays, the partition results in one partition being empty and the other having n-1 elements. This results in nested calls till the partition reaches size 1. Thus n-1 calls are made for each element and for n element, the time complexity is O(n^2) in the worst case. Efficiency of quicksort depends on pivot chosen.
This shows that if pivot is chosen randomly quicksort efficiency will be better.

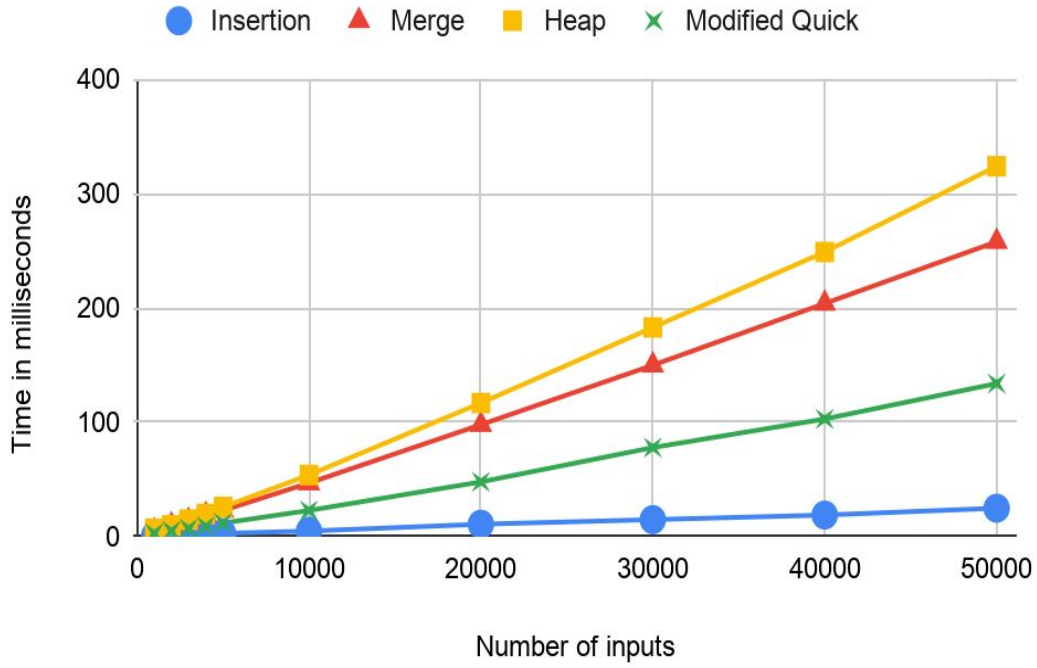
Modified Quick Sort (median-of-three as pivot):

In modified quicksort with median-of-three as pivot, the elements are compared with a pivot element which is the median of the first, last and middle elements of the array. This will never result in a partitioning case like in Quicksort and both sub arrays will have n > 0 elements. The partitioned array is executed in O(n) time as each element of the array needs to be compared with the pivot. The number of times the array gets partitioned is O(lon(n)).Thus the time complexity is θ(n log(n)).

Note: execution time is measured in milliseconds (rounded Up)

Number of inputs	Insertion	Merge	Heap	Quick	Modified Quick
1000	1	5	7	110	3
2000	1	10	10	329	5
3000	2	14	15	679	7
4000	2	19	20	1176	9
5000	3	23	26	1832	12
10000	5	47	54	7172	23
20000	11	98	117	28085	48
30000	15	150	183	63602	78
40000	19	204	249	115464	103
50000	25	258	324	179571	134


Insertion
Merge
Heap
Quick
Modified Quick
200000
150000
100000
50000
0
Time in milliseconds
0
10000
20000
30000
40000
50000
Number of inputs


Insertion
Merge
Heap
Modified Quick
400
300
200
100
0
Time in milliseconds
0
10000
20000
30000
40000
50000
Number of inputs

Complexity analysis for reversely sorted arrays:

Insertion Sort:

When the input array is sorted, insertion sort algorithm stops with a single comparison of the element and its previous element and no swapping is required. Thus the total number of comparisons is directly proportional to the number of elements. This results in best case for insertion sort and it's time complexity is O(n).

Merge Sort :

When the input array is reversely sorted, merge sort still divides the array into sub arrays till the sub array consists of single elements that takes Ω(log n) and merges them in Ω(n) time , thus time complexity in worst case remains O(n log n).

Heap Sort:

In heap sort, even if the elements are reversely sorted, building of heap property minimum takes O(log(n)) comparisons for each element insertion or deletion. To form a sorted array, whenever it removes a minimum element, each removal need heapify which means we need to restore the heap property which will take minimum log n. For n elements, the time complexity will be O(n log(n)). Thus, sorting using heap sort for n elements, the time complexity remains O(n log(n)).

Quick Sort :

In quick sort, the elements are compared with a pivot element and elements are partitioned into arrays that are greater than or less than the element. In our implementation the pivot is chosen as the first element. With reversely sorted arrays, the partition results in one partition being empty and the other having n-1 elements. This results in nested calls till the partition reaches size 1. Thus n-1 calls are made for each element and for n elements, the time complexity is O(n^2) in the worst case. Efficiency of quicksort depends on pivot chosen.
This shows that if pivot is chosen randomly quicksort efficiency will be better.

Modified Quicksort :

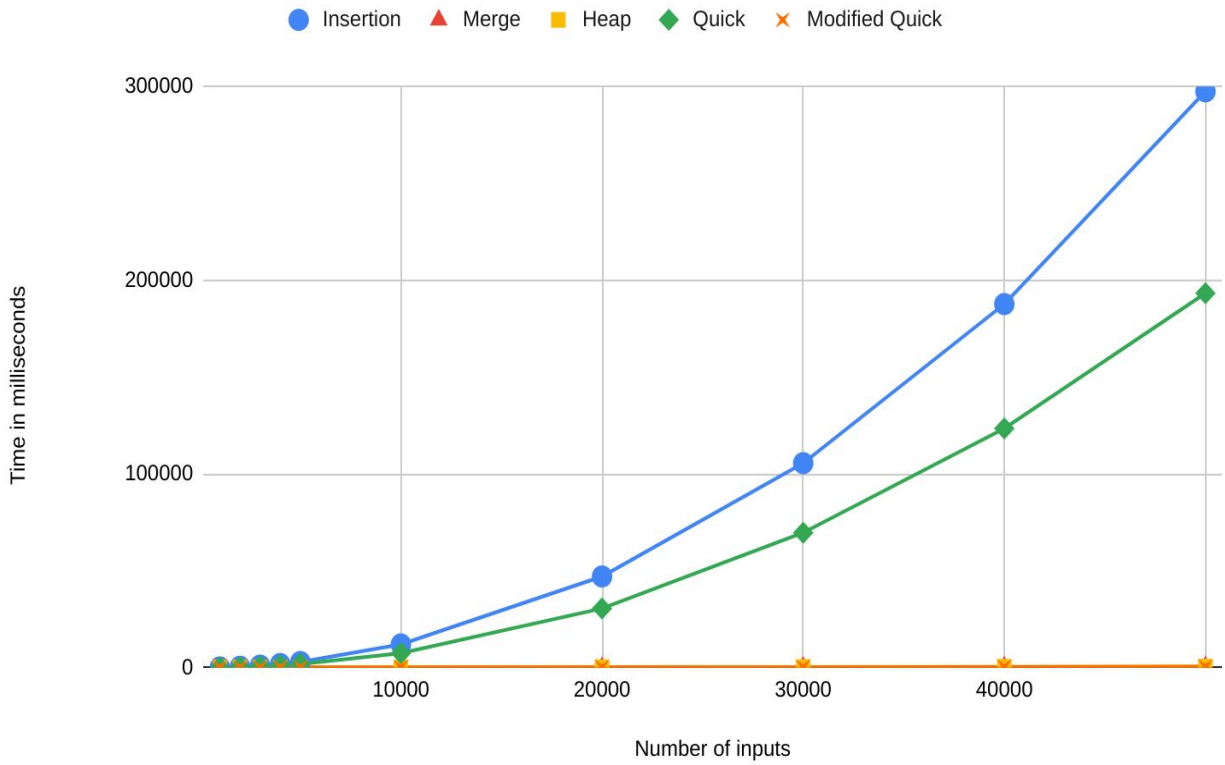
In modified quicksort with median-of-three as pivot, even if the elements are reversely sorted the pivot element chosen is the median of the first, last and middle elements of the array. This will never result in a partitioning case like in Quicksort and both sub arrays will have n>0 elements. The partitioned array is executed in O(n) time as each element of the array needs to be compared

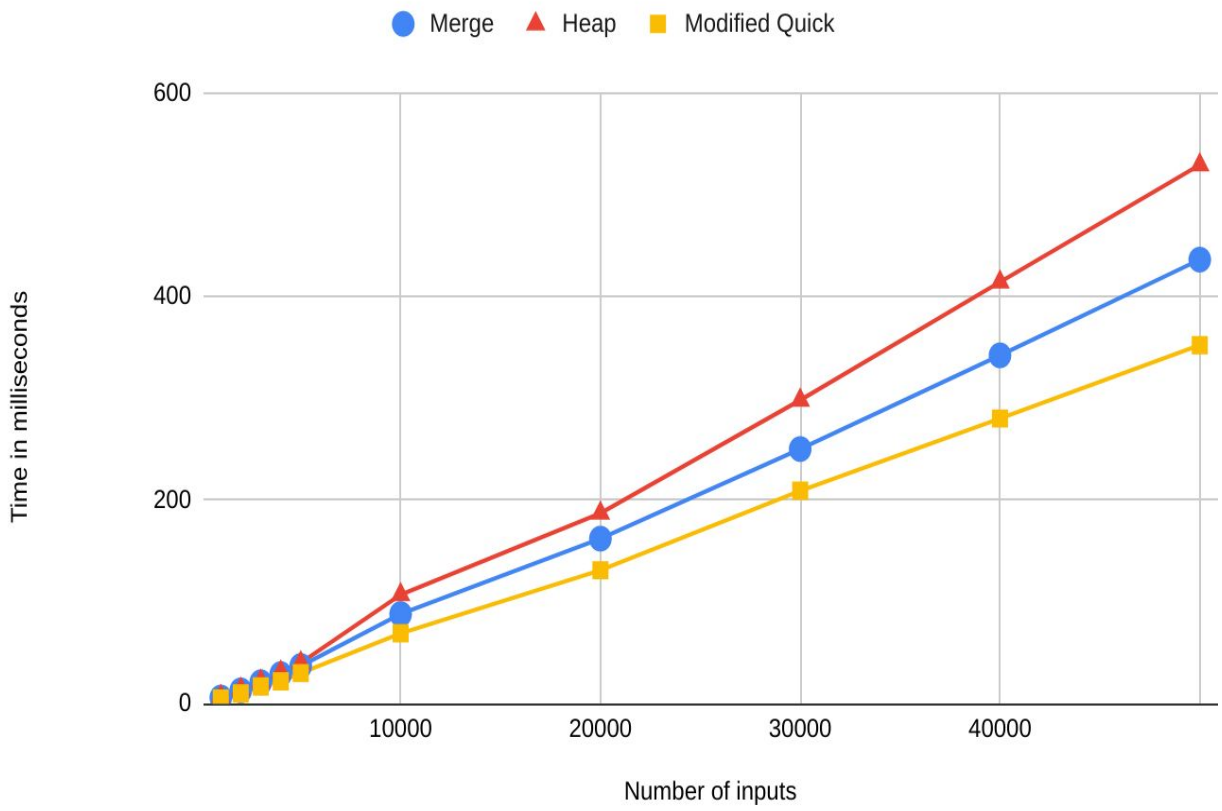
with the pivot. The number of times the array gets partitioned is O(log(n)). Thus the time complexity is O(n log(n)).

Results :

Note: execution time is measured in milliseconds (rounded Up)

Number of inputs	Insertion	Merge	Heap	Quick	Modified Quick
1000	116	6	7	75	5
2000	463	13	14	295	10
3000	1040	21	22	673	17
4000	1857	29	31	1175	22
5000	2905	37	40	1862	30
10000	11993	88	107	7469	69
20000	47055	162	187	30557	131
30000	105534	250	298	69653	209
40000	187668	342	414	123468	280
50000	297481	436	529	193309	352





Conclusion :

Note **:** To get more accurate results, we have set CPU affinity using taskset utility to avoid context switches while running the algorithm. This will benefit in reducing the cache misses due to context switch.

For ideally testing environment, we should use a machine (isolated testing environment) where no other process can intervene in the process resources like compute & memory. So, the results could have been affected as we have used our personal laptop as testing environment.

Efficiency of tested algorithms can be ordered as follows for the given cases :

Random/Unsorted Array :

Modified Quick > Quick > Merge > Heap > Insertion Sort

Sorted Array :

- If pivot element is first or leftmost element in quick sort then

Insertion > Modified Quick > Merge > Heap > Quick Sort

- If pivot element is randomly chosen in quick sort then

  Insertion > Modified Quick > Quick Sort > Merge > Heap Sort

Reverse Sorted Array :

Modified Quick > Merge > Heap > Quick (pivot=first element)> Insertion Sort

Array of large input size :

Modified Quick >Quick > Merge > Heap > Insertion sort