Comparison-based Sorting Algorithms

Project -1 (Fall 2019)

By : Anish Bhattacharyya

&

Abhilash Mandlekar

Table of Contents

1 INTRODUCTION

Implement the following sorting algorithms and compare them.

1. Insertion sort
2. Merge sort
3. Heapsort [vector based, and insert one item at a time]
4. In-place quicksort (any random item or the first or the last item of your input can be pivot).
5. Modified quicksort
   - Use median-of-three as pivot.
   - For small sub-problem of size ≤ 10, use insertion sort.

Execution instructions:

1. Run these algorithms for different input sizes (e.g. n = 1000, 2000, 4000, 5000, 10,000 .. 40,000, 50000). You will randomly generate numbers for your input array. Record the execution time (need to take the average as discussed in the class) and plot them all in a single graph against input size. Note that you will compare these sorting algorithms for the same data set.
2. Also observe and present performance of the following two special cases:
   - Input array is already sorted.
   - Input array is reversely sorted.

Introduction:

We choose python programming language for this project. To make the process easier we are taking system generated random numbers as input to the sorting algorithms. To do this, library random is used which gives the random number between specified range.

We checked the time taken by each algorithm by measuring time took by program to finish the particular method. Time library is used to get the exact run time.

The library matplotlib is used for plotting the graphs of time vs input size array. By plotting the graphs we have compared the performance of the sorting algorithms.

Data structure Used

In insertion sort algorithm, since we are selecting random numbers that are integer values, we have considered **list** in python to sort the numbers. The method append is used to insert elements in the list. Python provides very efficient way to reverse list by simply typing [::-1], e.g. mylist[::-1] will reverse the contents of mylist.

2 Insertion Sort

2.1 Code:

```
def iSort( a):
    for j in range(1,len(a)):
        key=a[j]
        i=j-1
        while i>=0:
            if key<a[i]:
                a[i+1]=a[i]
                a[i]=key
                i-=1
            else:
                break
    return a
```

2.2 Complexity analysis:

Here we iterate through **all elements** in list for each selected **key** element in the array and hence the complexity of insertion sort becomes $\boldsymbol{O(n^2)}$**.** It is also because we visit every element twice.

Insertion sort performs best when the list of elements are already sorted. In this case since the elements on the left of the key are already sorted, there is no need to swap the elements . Hence for the best case the complexity of insertion sort becomes $\boldsymbol{O(n)}$**.**

3 Merge Sort

3.1 Code:

```
def mergeSort(a):
    if len(a) >1:
        mid = len(a)//2
        Left_a = a[:mid]
        Right_a = a[mid:]
        mergeSort(Left_a)
        mergeSort(Right_a)

        i = j = k = 0
        while i < len(Left_a) and j < len(Right_a):
            if Left_a[i] < Right_a[j]:
                a[k] = Left_a[i]
                i+=1
            else:
                a[k] = Right_a[j]
                j+=1
            k+=1

        while i < len(Left_a):
            a[k] = Left_a[i]
            i+=1
```

```
            k+=1

        while j < len(Right_a):
            a[k] = Right_a[j]
            j+=1
            k+=1
    return a
```

3.2 Complexity analysis:

Merge sort is divide and conquer algorithm, and hence the list is partitioned in half recursively. To partitioned the list in halves it takes $\boldsymbol{O(log(n))}$ time. But still after partitioning the list we need to check the left and right element in the finally obtained list. To perform this we are visiting every element in list, which take $\boldsymbol{O(n)}$ time. Hence the total run-time of this algorithm becomes $\boldsymbol{O(nlog(n))}$**.**

The best case and worst case complexity of merge sort remains the same, as it requires the list to be partitioned in every case. The complexity comparing the last 2 elements is constant.

4 Heap Sort

4.1 Code:

```
def Heap_order(arr, n, i):

    largest = i

    l = 2 * i + 1

    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:

        largest = l

    if r < n and arr[largest] < arr[r]:

        largest = r

    if largest != i:

        arr[i],arr[largest] = arr[largest],arr[i]

        Heap_order(arr, n, largest)


def heapsort(arr):

    n = len(arr)

    for i in range(n, -1, -1):

        Heap_order(arr, n, i)

    for i in range(n-1, 0, -1):

        arr[i], arr[0] = arr[0], arr[i]

        Heap_order(arr, i, 0)
```

4.2 Complexity analysis

The complexity of this algorithm depends upon heapify and createHeap methods. if we create a heap with Top down or bottom up approach the complexity to perform this is $\boldsymbol{O(n)}$**.** To heapify it always takes $\boldsymbol{O(log(n))}$ time**,** because of the property of min-heap that the root element is always lesser than both of the children. So, if the new element is inserted, we can quickly heapify it by comparing its parent.

Hence the best case and worst case of the heapsort becomes $\boldsymbol{O(nlog(n))}$**.**

5 In-place Quick sort

5.1 Code:

```
def divide(arr, low, high, piv):
    arr[low], arr[piv] = arr[piv], arr[low]
    pivot = arr[low]
    i = low + 1
    j = low + 1

    while j <= high:
        if arr[j] <= pivot:
            arr[j], arr[i] = arr[i], arr[j]
            i += 1
        j += 1

    arr[low], arr[i - 1] = arr[i - 1], arr[low]
    return i - 1

def quicksort(arr, low, high):
    if high - low < 1:
        return
    piv = random.randint(low, high)
    i = divide(arr, low, high, piv)
    quicksort(arr, low, i - 1)
    quicksort(arr, i + 1, high)
```

5.2 Complexity analysis

Quick sort is faster than most of the sorting algorithms, since its inner loop can be optimised by picking up the correct pivot element. Since quick sort also needs the partition, the partitioning algorithm has logarithmic complexity $\boldsymbol{O(log(n))}$**.** At each level of recursion, all partitions at that level takes linear time complexity $\boldsymbol{O(n)}$**.** So the algorithm has the complexity of $\boldsymbol{O(nlog(n))}$ best and average case.

The complexity of this algorithm might vary for worst case. It is because if the pivot is chosen in such a way that it is either minimum or the maximum element in the array. In this case, all the elements to the right or to the left of pivot needs to be sorted and then it takes $\boldsymbol{O(n^2)}$ time.

6 Modified Quick sort

6.1 Code

```
def divide(array, start, end, idx_pivot):
    array[start], array[idx_pivot] = array[idx_pivot], array[start]
    pivot = array[start]
    i = start + 1
    j = start + 1

    while j <= end:
        if array[j] <= pivot:
            array[j], array[i] = array[i], array[j]
            i += 1
        j += 1

    array[start], array[i - 1] = array[i - 1], array[start]
    return i-1

def MedianOfThree(array, start, end):
    mid = (start + end)//2
    if array[end] < array[start]:
        array[end], array[start] =  array[start],  array[end]
    if array[mid] < array[start]:
        array[mid], array[start] =  array[start],  array[mid]
    if array[end] < array[mid]:
        array[end], array[mid] =  array[mid],  array[end]
    return mid

def iSort_modified( array, start, end):
```

```
    for j in range(1,end):
        key=array[j]
        i=j-1
        while i>=0:
            if key<array[i]:
                array[i+1]=array[i]
                array[i]=key
                i-=1
            else:
                break

def modified_quicksort(array, start, end):
    if end is None:
        end = len(array) - 1
    if end - start < 1:
        return
    idx_pivot = MedianOfThree(array, start, end)
    i = divide(array, start, end, idx_pivot)

    if (i-1-start <= 10):
        iSort_modified(array, start, i)
    else:
        quicksort(array, start, i-1)

    if (end-i-1 <= 10):
        iSort_modified(array, i+1, end)
    else:
        modified_quicksort(array, i+1, end)
```

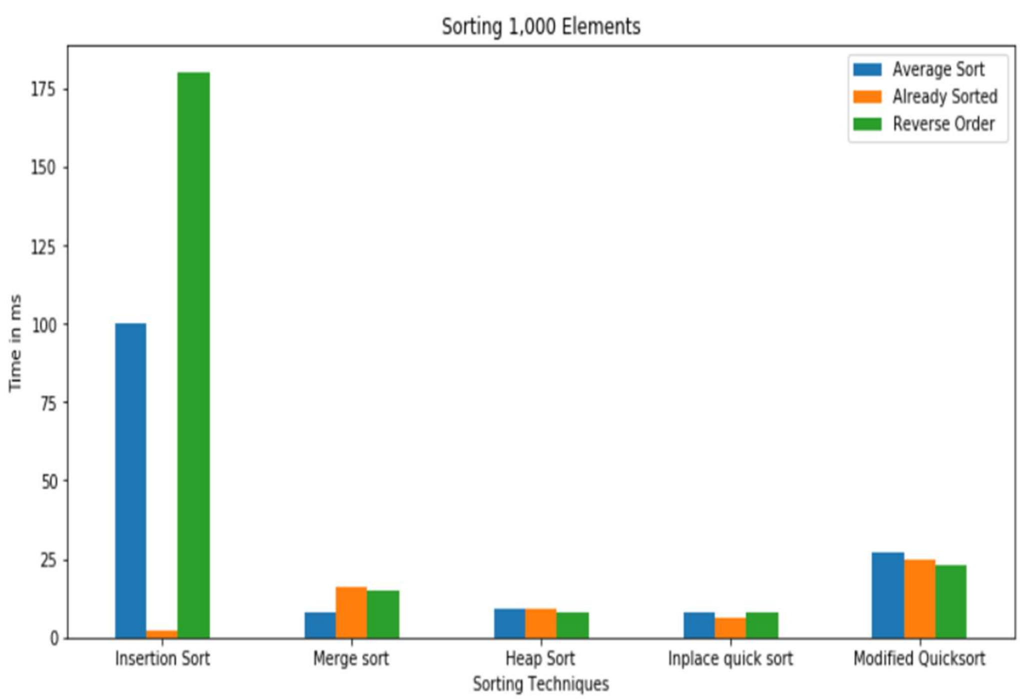
6.2 Complexity analysis

In modified quick sort, instead of randomly choosing the pivot element, we use median of three algorithm. In this technique, we sort the first, middle and last element in the list and rearrange these elements. We can select the element as a pivot with median key. This algorithm works for every partition of list and hence prevents from picking up the bad pivot. Hence even for the worst case the time complexity remains $\boldsymbol{O(nlog(n))}$**.**

7 Results and Graphs

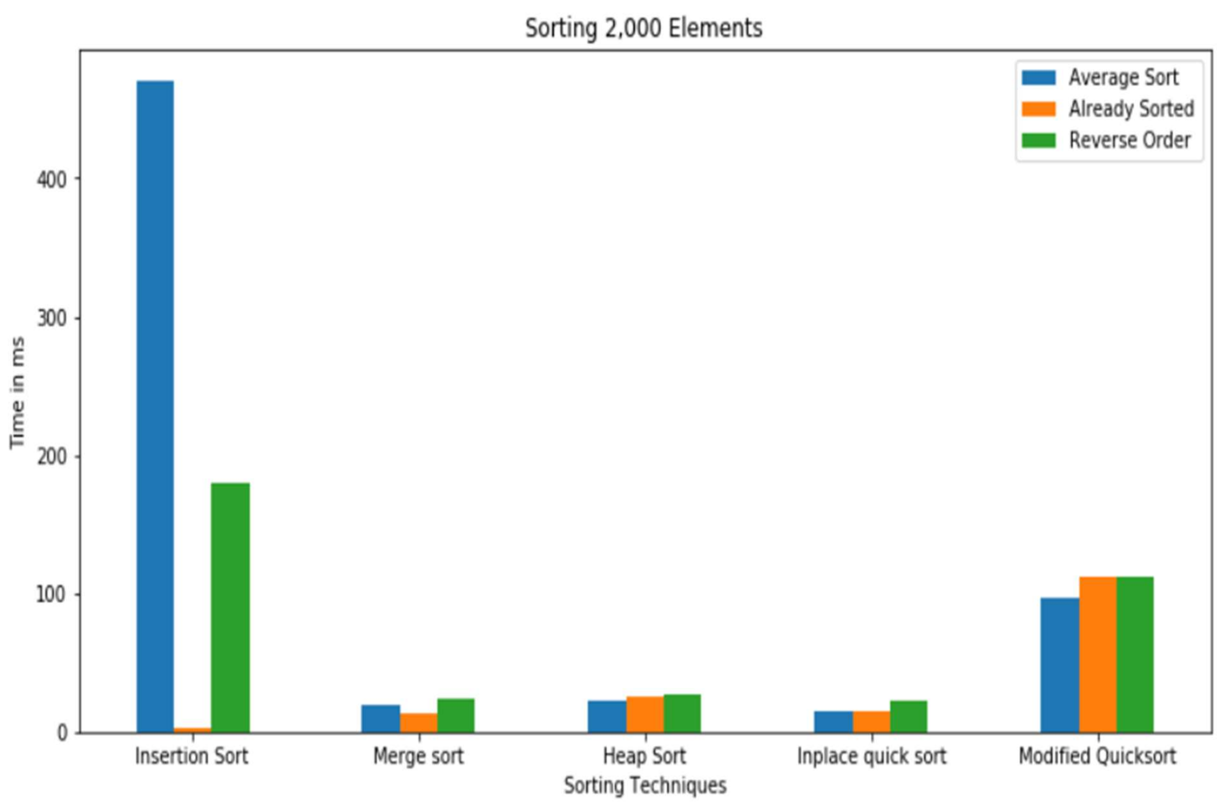
For input size 1000

Sorting Techniques	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Sort	Already Sorted	Reverse Order
Insertion Sort	95.94011307	104.9358845	97.9373455	100	1.998662949	179.8870564
Merge sort	6.994009018	6.995439529	10.9937191	8	15.98858833	14.99271393
Heap Sort	8.991479874	10.99181175	7.994174957	9	8.992433548	7.992267609
Inplace quick sort	6.995201111	9.99212265	5.993843079	8	5.997896194	7.998466492
Modified Quicksort	31.98170662	27.98008919	25.98261833	27	24.98412132	22.98355103



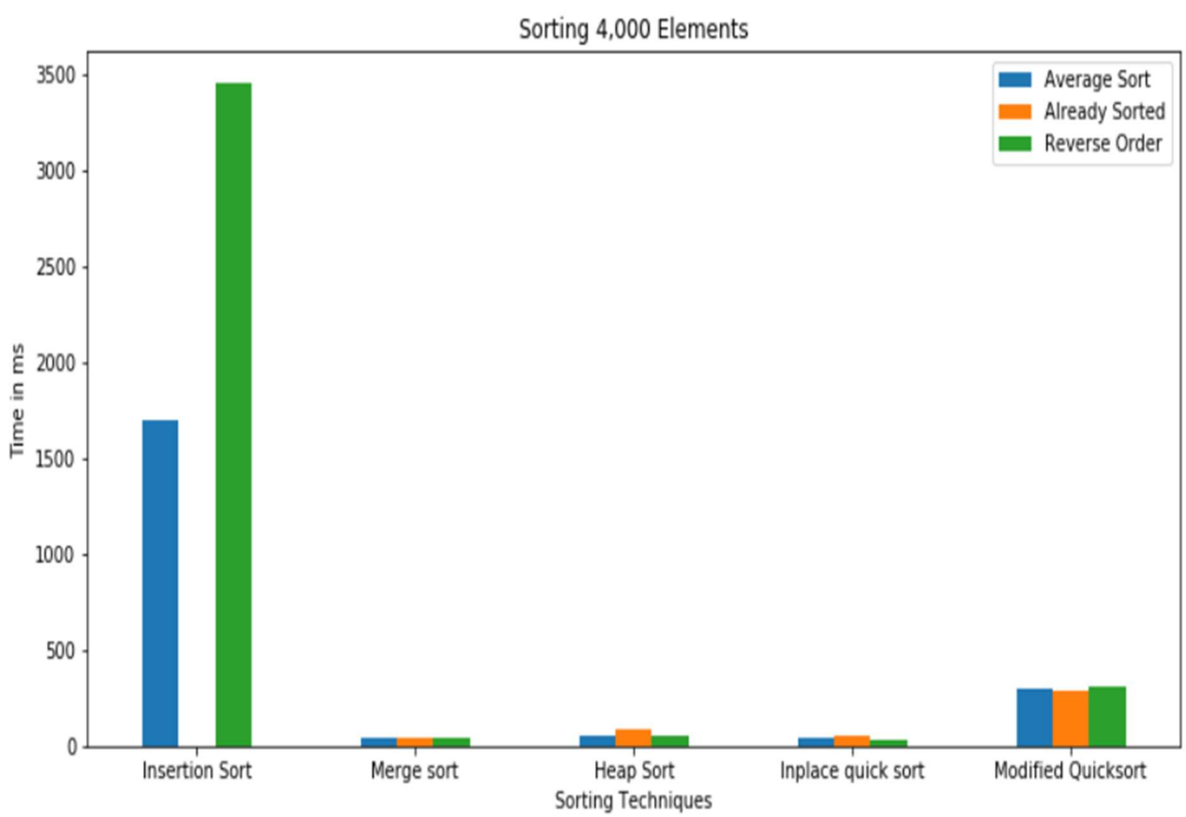
For input size 2000

Sorting Techniques	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Sort	Already Sorted	Reverse Order
Insertion Sort	357.7785492	687.7858639	365.7717705	470	1.998186111	179.8870564
Merge sort	21.99077606	14.99128342	20.98608017	19	12.98999786	23.98443222
Heap Sort	20.98369598	23.98276329	24.9838829	23	25.98142624	26.98135376
Inplace quick sort	18.98860931	12.98928261	14.99199867	15	14.991045	22.98521996
Modified Quicksort	82.94653893	99.93910789	109.9317074	97	111.9289398	111.928463



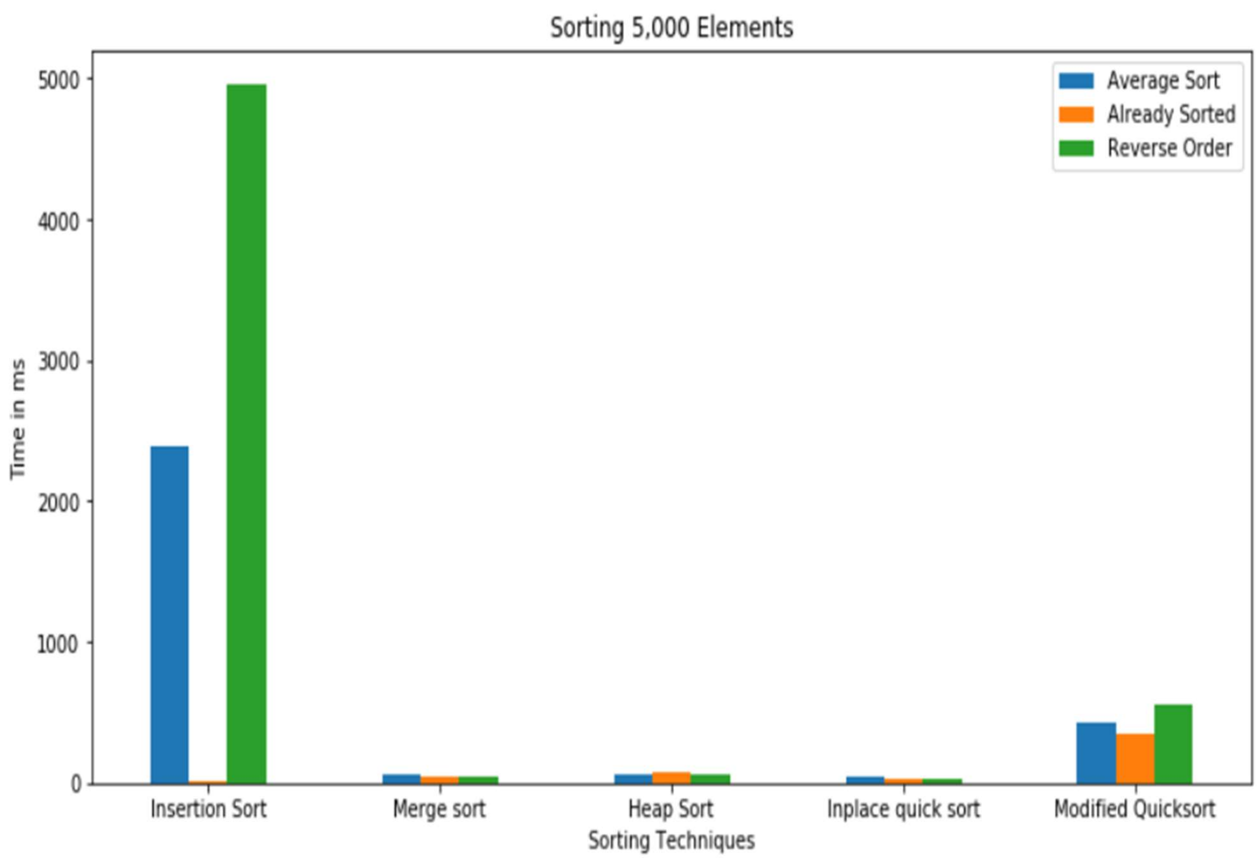
For input size 4000

	A	B	C	D	E	F	G
1	Sorting Techniques	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Sort	Already Sorted	Reverse Order
2	Insertion Sort	1730.935335	1659.5366	1694.974661	1694	1.996994019	3449.99218
3	Merge sort	38.9752388	49.97014999	43.9722538	44	37.97483444	36.97800636
4	Heap Sort	50.96912384	48.97069931	51.96666718	50	82.94820786	53.96842957
5	Inplace quick sort	34.97767448	43.76511185	50.96888542	42	52.96516418	34.97958183
6	Modified Quicksort	335.7944489	292.8166389	284.8274708	303	286.823988	304.8110008
7							
8							



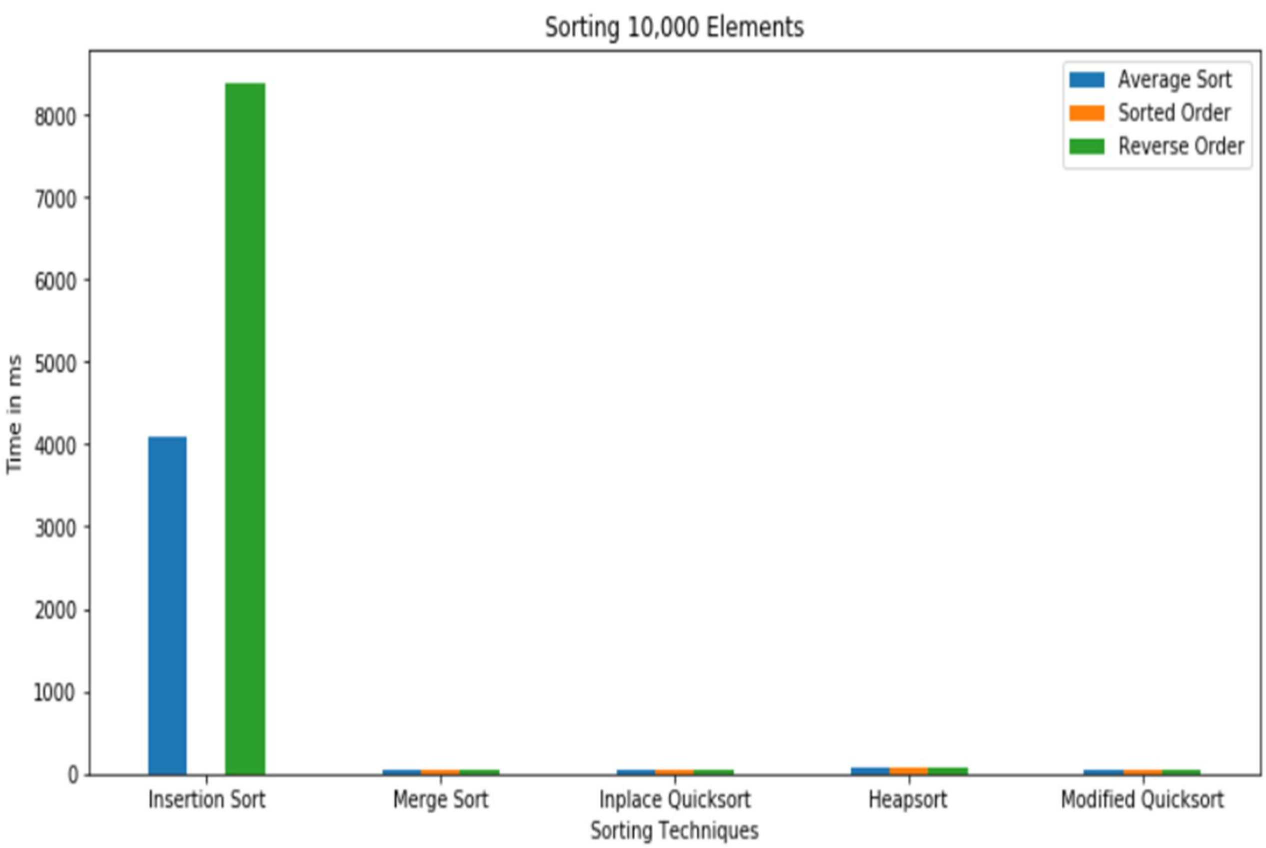
For input size 5000

	A	B	C	D	E	F	G
1	Sorting Techniques	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Sort	Already Sorted	Reverse Order
2	Insertion Sort	2382.776737	2422.891378	2352.97823	2385	5.996465683	4956.434965
3	Merge sort	62.96157837	58.961629	59.96322632	60	38.97428513	41.97525978
4	Heap Sort	72.95370102	66.96152687	63.95721436	68	69.95415687	58.96449089
5	Inplace quick sort	36.97514534	41.97454453	33.97655487	37	35.97545624	35.97712517
6	Modified Quicksort	443.4783459	428.8237095	426.138401	432	340.1606083	559.6501827
7							
8							



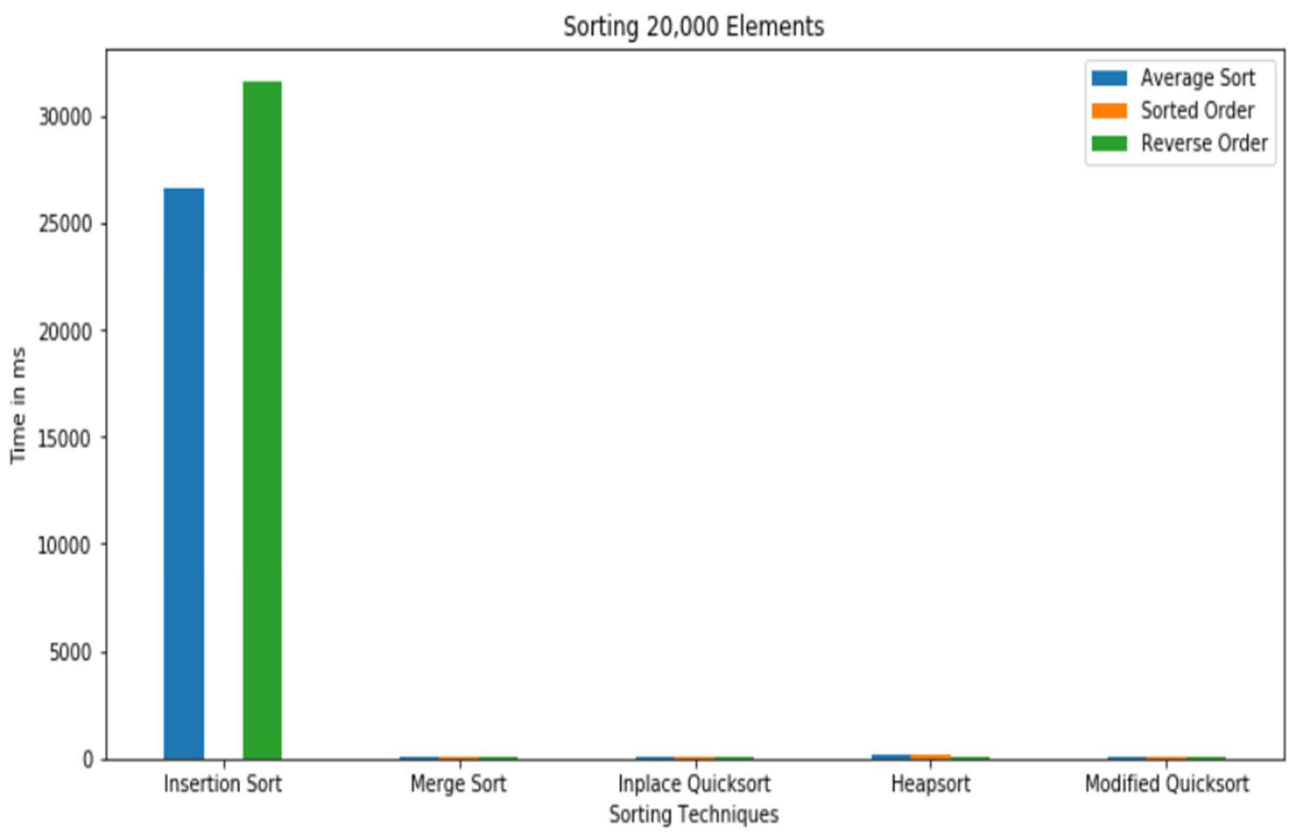
For input size 10000

Sorting Techniques	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Sort	Already Sorted	Reverse Order
Insertion Sort	4021	4028	4187	4078	3	8372
Merge Sort	55	58	48	53	50	42
Inplace Quicksort	37	49	36	41	49	40
Heapsort	67	79	59	68	68	62
Modified Quicksort	41	51	42	44	40	45



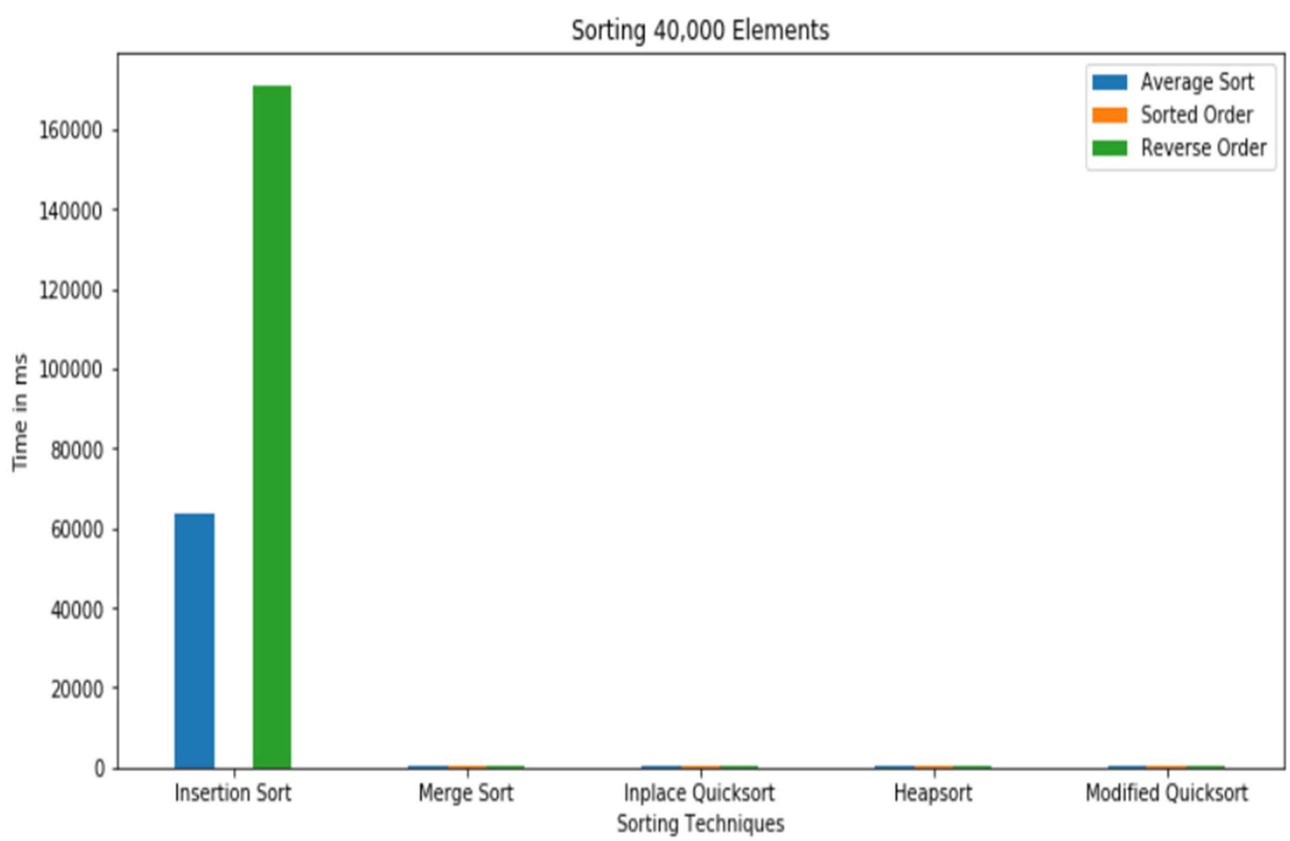
For input size 20000

Sorting Techniques	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Sort	Sorted Order	Reverse Order
Insertion Sort	48227	15466	16058	26583	3	31560
Merge Sort	107	127	93	109	83	87
Inplace Quicksort	74	91	78	80	90	82
Heapsort	132	251	113	165	135	127
Modified Quicksort	87	104	81	90	85	87



For input size 40000

	A	B	C	D	E	F	G
1	Sorting Techniques	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Sort	Sorted Order	Reverse Order
2	Insertion Sort	62407	65749	63679	63945	12	170800
3	Merge Sort	216	228	531	328	291	278
4	Inplace Quicksort	154	181	423	252	250	266
5	Heapsort	300	309	735	448	457	527
6	Modified Quicksort	199	190	431	273	291	320
7							



For input size 50000

	A	B	C	D	E	F	G
1	Sorting Techniques	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Sort	Sorted Order	Reverse Order
2	Insertion Sort	102950	99582	100902	101144	15	211007
3	Merge Sort	287	280	325	297	241	258
4	Inplace Quicksort	237	252	250	246	252	267
5	Heapsort	411	387	399	399	384	359
6	Modified Quicksort	242	241	243	242	250	259
7