UNC CHARLOTTE
College of Computing and Informatics

PROJECT: 1 REPORT

COMPARISION-BASED SORTING ALGORITHMS

TEAM MEMBERS:

KEERTHI REDDY KANDI (801149971)

AKSHARA GONE (801136090)

PROFESSOR:

Dr. DEWAN TANVIR AHMED

dahmed@uncc.edu

Contents

1. INSERTION SORT

Insertion Sort is an ‘In-place’ Sorting Algorithm which considers ‘Array’ data structure for sorting. Sorting is done iterating the array and the left subarray at each iteration will be sorted. At each iteration, Insertion Sort takes one element of the input data and finds its correct position in the array and inserts it there. After i iterations, array from index 1 to i-1 will be sorted. Insertion sort is efficient for small data sets, but in-case of large data sets this sorting algorithm is not suitable as the complexity in average and worst case is O(n^2). Insertion Sort works well for substantially sorted input [O(n)], the best case.

Insertion Sort works as follows:

Consider array: 7, 4, 6, 9, 2, 10

7	**4**	**6**	**9**	**2**	**10**

First Iteration: Compare element 7 and 4. 7<4 is false, so swap 4 and 7.

7	**4**	**6**	**9**	**2**	**10**

Iterator will be at 6 now and the array after swapping will be

4	**7**	**6**	**9**	**2**	**10**

Second Iteration: Next compare 6 with 7. 7<6 is false, so swap 7 and 6. And again compare 6 with 4. 4<6 so no swap is required.

4	**7**	**6**	**9**	**2**	**10**

Iterator will be at 9 now and the array after swapping will be

4	**6**	**7**	**9**	**2**	**10**

Third Iteration: Compare 9 with 7 and its preceding elements. Here 9 is greater than all the preceding elements. So, position of 9 remains same. And iterator will be at 2.

4	**6**	**7**	**9**	**2**	**1**

Fourth Iteration: Now, compare 2 with 9. 9<2 is false, so swap. Continue by comparing 2 with its preceding elements and 2 is finally placed at first position which is its correct position.

4	**6**	**7**	**9**	**2**	**10**

4	**6**	**7**	**2**	**9**	**10**

4	**6**	**2**	**7**	**9**	**10**

4	**2**	**6**	**7**	**9**	**10**

Iterator will be at 10 now and the array after swapping will be

2	**4**	**6**	**7**	**9**	**10**

Fifth Iteration: Compare 10 with its preceding elements. Here 10 is greater than all the preceding elements so no swap is needed. 10 will remain in its position

Final Sorted Array is

2	**4**	**6**	**7**	**9**	**10**

Psuedo-Code :

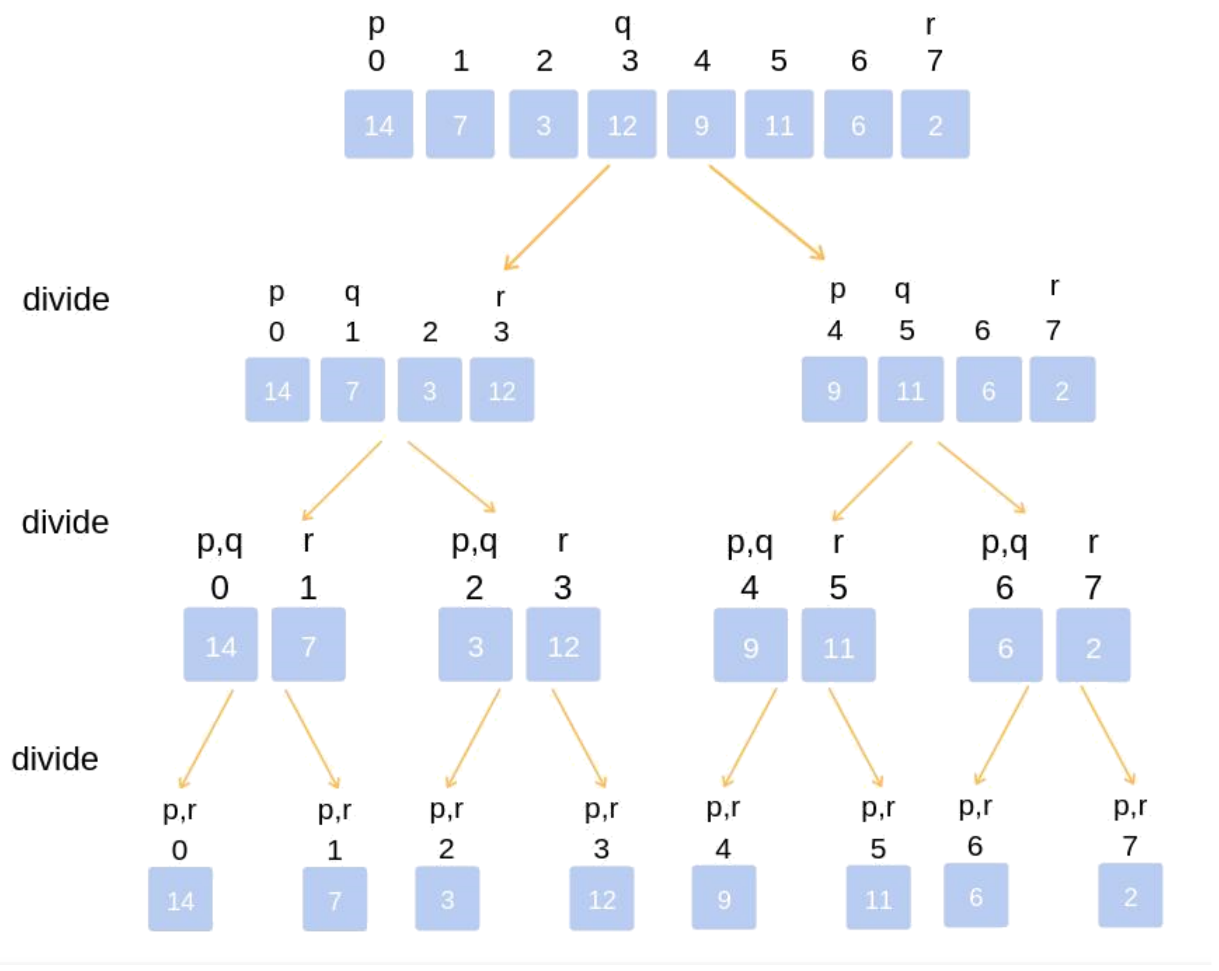
```
def InsertionSort(arr):
    for index in range(1, len(arr)):
        key = arr[index]
        j = index - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j=j-1
        arr[j + 1] = key
```

2.MERGE SORT

Merge Sort works on the principle of Divide and Conquer. This sorting algorithm repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists results into a sorted list. The worst case time complexity of Merge Sort is O(nlogn),which makes it a efficient algorithm.

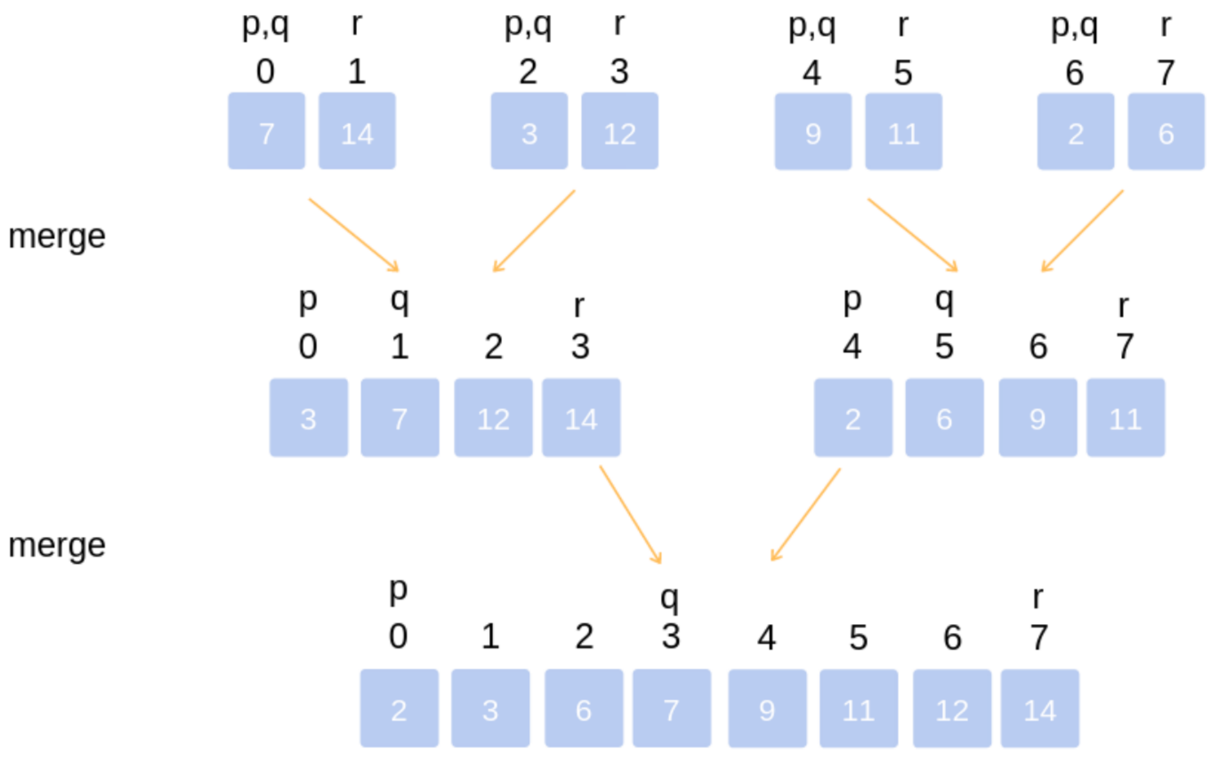
Merge Sort works as follows:

1. Recursively divide the unsorted list into two sublists, until we get each sublist comprising 1 element (a list of 1 element is supposed sorted).



2. Now repeatedly merge sublists to produce newly sorted list. The merging is done as follows:

   The first element of both lists is compared. If sorting in ascending order, the smaller element among two becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the newly combined sublist covers all the elements of both the sublists.



Pseudocode for Merge Sort :

```
MergeSort(inputarray[], left,  right)
If right > left:
     1. Find the middle point to divide the array into two
            halves: middle = (left+right)/2
     2. Call mergeSort for first half:

     3. Call mergeSort for second half:

     4. Merge the two halves sorted in step 2 and 3: Call
            merge(inputarray, left, middle, right)
```

3.HEAP SORT

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. Heap Sort divides its input into sorted and unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. Vector Based implementation is used to represent the binary heap. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort works as follows:

Basically there are two phases in sorting of elements using heap sort, they are :

1. Building a Heap: In this phase, the elements are inserted one at a time into the heap array. After the insertion of a new key, the heap order property may be violated. Upheap algorithm is used to restore this property.
2. Removing the root: Once the heap is created, repeatedly eliminate the root element of the heap (i.e First element of the array), replace the root with last element of the array and then store the heap order property with remaining elements.

Consider an array and Build Min Heap : [4,10,3,5]

1. 4 is inserted in the array.

4			

2. 10 is inserted and compared with its parent 4. As 4<10 the heap order property is not violated and hence there is no swapping.

4	10		

3. 3 is inserted and compared with its parent 4. As 4>3 the heap order property is violated and hence 3 and 4 are swapped.

4	10	3	

3	10	4	

4. 5 is inserted and compared with its parent 10. As 10>5 the heap order property is violated and hence 10 and 5 are swapped.

3	10	4	5

3	5	4	10

Removing the Minimum Element (Root):

1. Root 3 is removed and replaced with 10, and then 4 and 10 are swapped to restore the heap order property.

4	5	10	

2. Root 4 is removed and replaced with 10, and then 5 and 10 are swapped to restore the heap order property.

5	10		

3. Root 5 is removed and replaced with 10.

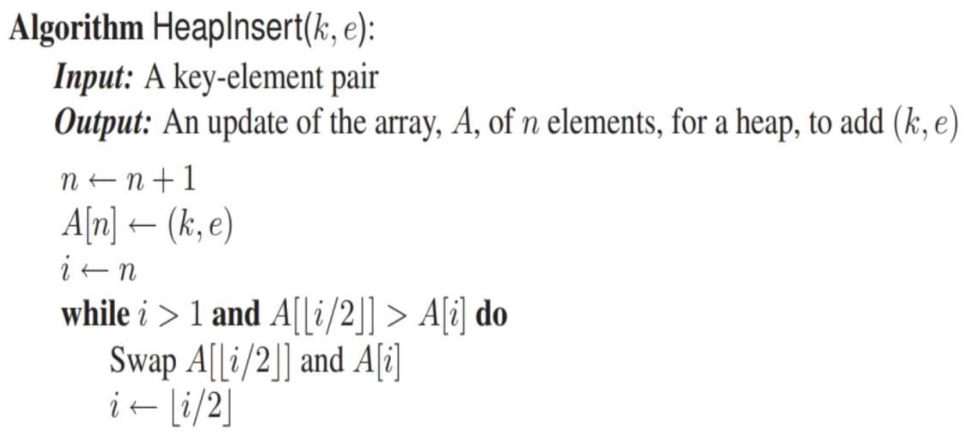
10			

4. Finally Root 10 is removed and the sorted output array is obtained.

3	4	5	10

<u>Pseudocode:</u>

Phase 1 : Build Heap



```
Algorithm HeapInsert($k, e$):
    Input: A key-element pair
    Output: An update of the array, $A$, of $n$ elements, for a heap, to add $(k, e)$
    $n \leftarrow n + 1$
    $A[n] \leftarrow (k, e)$
    $i \leftarrow n$
    while $i > 1$ and $A[\lfloor i/2 \rfloor] > A[i]$ do
        Swap $A[\lfloor i/2 \rfloor]$ and $A[i]$
        $i \leftarrow \lfloor i/2 \rfloor$
```

Phase 2 : Remove Minimum

```
Algorithm HeapRemoveMin():
  Input: None
  Output: An update of the array, $A$, of $n$ elements, for a heap, to remove and
    return an item with smallest key
  $temp \leftarrow A[1]$
  $A[1] \leftarrow A[n]$
  $n \leftarrow n - 1$
  $i \leftarrow 1$
  while $i < n$ do
      if $2i + 1 \leq n$ then   // this node has two internal children
          if $A[i] \leq A[2i]$ and $A[i] \leq A[2i + 1]$ then
              return $temp$        // we have restored the heap-order property
          else
              Let $j$ be the index of the smaller of $A[2i]$ and $A[2i + 1]$
              Swap $A[i]$ and $A[j]$
              $i \leftarrow j$
      else   // this node has zero or one internal child
          if $2i \leq n$ then   // this node has one internal child (the last node)
              if $A[i] > A[2i]$ then
                  Swap $A[i]$ and $A[2i]$
          return $temp$        // we have restored the heap-order property
  return $temp$        // we reached the last node or an external node
```

4.In-Place Quick Sort

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

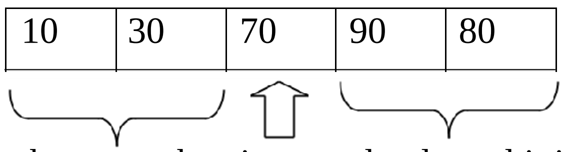
In-Place quick sort works as follows:

1. An element called pivot is randomly picked from the array.
2. Partitioning: Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After this partitioning, the pivot is in its correct position. This is called the Partition Function.
3. Recursively apply the above steps to the two sub arrays - one array containing smaller elements than pivot and the other array containing elements grater than pivot.

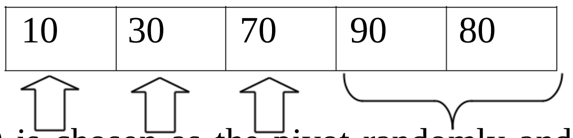
Consider an array:

10	80	30	70	90

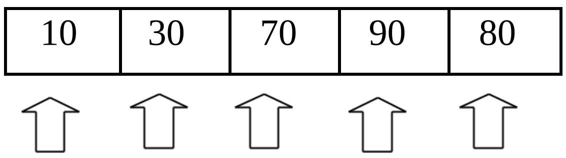
1. 70 is selected as the pivot randomly. It is placed in the position such that all the elements to its left side are less than 70 and all the elements to its right side are greater than 70.



2. In the left subarray 10 is chosen as the pivot randomly and it is placed in its position such that all the elements to its left side are less than 10 and all the elements to its right side are greater than 10.



3. In the right subarray 90 is chosen as the pivot randomly and it is placed in its position such that all the elements to its left side are less than 90 and all the elements to its right side are greater than 90.



4. Finally, the sorted output array is obtained.

Pseudocode:

```
Algorithm InplaceQuickSort(array):
  function sorting(input, left, right):
     if right <= left:
        return
     pivot_index = random.randint(left, right)
     input[left], input[pivot_index] = input[pivot_index], input[left]
     i = left
     for j in range(left + 1, right + 1):
        if input[j] < input[left]:
           i += 1
           input[i], input[j] = input[j], input[i]
     input[i], input[left] = input[left], input[i]
     sorting(input, left, i - 1) sorting(input, i
     + 1, right)
  sorting(array, 0, len(array) - 1)
```

5.MODIFIED QUICK SORT (Median-of-three)

An optimal quick sort, which uses median of array as pivot. Quick sort is an fastest sorting algorithm.

This quick sort is efficient as it calls Insertion sort when the subarray size is <10 because quick sort does not perform efficiently for small inputs.

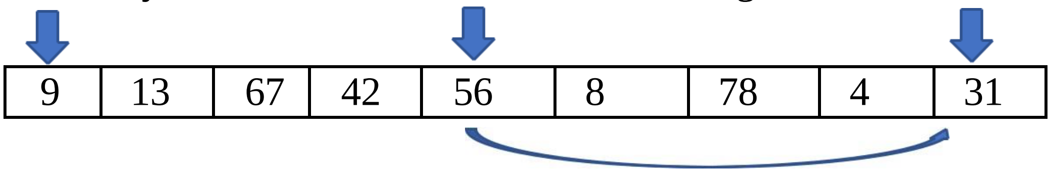
For large inputs, quick sort performs efficiently.

<u>Median-of-three rule:</u>

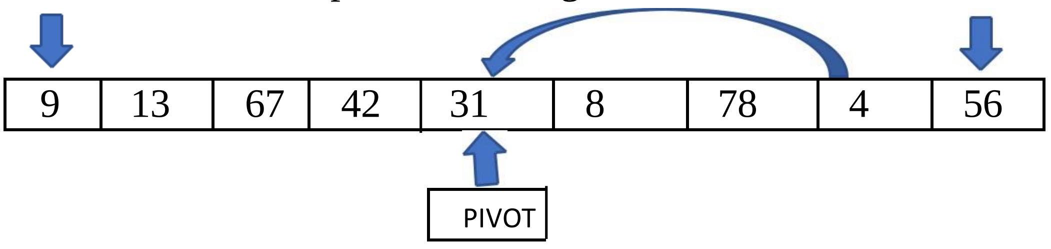
From the leftmost, middle and rightmost elements of the list select the one with median key as pivot. Swap the elements if necesary so that smallest element will be in the left position, largest at right and median at center.

Quick Sort works as follows:

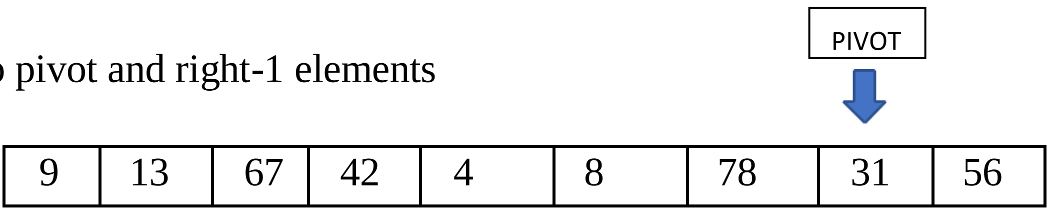
Consider an array: 9,13,67,42,56,8,78,4,31 [left=9, right=31, center=56]



1. Perform Median-of-three rule: Left element is the smallest and its position is retained. 31<56 so swap center and right element.



2. Swap pivot and right-1 elements



3. Partition takes place now at left+1 to right-2 to get sorted list.

<u>Psuedo-Code:</u>

```
Algorithm Modified_Quicksort(input,left,right):
    If (left+10<=right):
        Pivot=medianofthree(input,left,right)
        i=left
        j=right- 2
        done= False
        while not done:
            while(input[i]<pivot):
                i=i+1
            while(input[j]>pivot):
                j=j-1
            if i<j:
                swap input[i],input[j]
            else:
                done=True
        swap input[i],input[right-1]
        Modified_Quicksort(input,left,i-1)
        Modified_Quicksort(input,i+1,right)
    Else:
        InsertionSort(input,left,right)
```

6. Special Cases (Sorted, Reverse Sorted Input)

Case1: Sorted Input:

Using “sorted” function in python. The code is same as random input, just added one line to the code to get the sorted list of randomly generated input. we passed this sorted input to the sorting algorithms.

```
Sorted_Input=sorted(Random_Input)
```

Case2: Reverse Sorted Input:

Using “sorted” function in python and “reverse=True” in python. The code is same as random input, just added one line to the code to get the reversely sorted list of randomly generated input. we passed this reversely sorted input to the sorting algorithms.

```
Reverse_Sorted_Input=sorted(Random_Input, reverse=True)
```

DATA STRUCTURES CHOOSEN:

Data Structure used in sorting algorithms is “array” (list). For Heap sort, used heap data structure implemented using vector(array).

7.CODE

```
import random
import time
from sys import setrecursionlimit
setrecursionlimit(100000)

### INSERTION SORT FUNCTION ###
def InsertionSort(array):
    for index in range(1, len(array)):
        key = array[index]
        j = index - 1
        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j=j-1
        array[j + 1] = key

### INSERTION SORT FOR MODIFIED QUICK SORT ###
def InsertionSort_10subarrays(alist,left,right):
    for k in range(left,right+1):
        key1=alist[k]
        l=k-1
        while l>=0 and key1 < alist[l]:
            alist[l+1]=alist[l]
            l=l-1
        alist[l+1]=key1

### MERGE SORT FUNCTION ###
def MergeSort(array):
    if len(array) > 1:
        middle = len(array) // 2
        Left = array[:middle]
        Right = array[middle:]
        MergeSort(Left)
        MergeSort(Right)
        i=0
        j=0
        k=0
        while i < len(Left) and j < len(Right):
            if Left[i] < Right[j]:
                array[k] = Left[i]
                i += 1
            else:
                array[k] = Right[j]
                j += 1
            k += 1
        while i < len(Left):
            array[k] = Left[i]
            i += 1
            k += 1
        while j < len(Right):
            array[k] = Right[j]
            j += 1
            k +=1

### HEAP SORT FUNCTION(BUILD AND REMOVE) ###
class HEAP:
    def_init_(self):
        self.heaplist = [0]
        self.length = 0

    def buildheap(self, item):
        self.heaplist.append(item)
        self.length = self.length + 1
        self.UpHeap(self.length)

    def UpHeap(self, j):
        while j // 2 > 0:
            if self.heaplist[j] < self.heaplist[j // 2]:
                self.heaplist[j // 2], self.heaplist[j] = self.heaplist[j],
self.heaplist[j // 2]
            j= j // 2

    def removemin(self):
        root = self.heaplist[1]
        self.heaplist[1] = self.heaplist[self.length]
        self.length = self.length - 1
        self.heaplist.pop()
        self.Downheap(1)
        return root

    def Downheap(self, i):
        while (2*i) <= self.length:
            m = self.minimum(i)
            if self.heaplist[i] > self.heaplist[m]:
                self.heaplist[i], self.heaplist[m] = self.heaplist[m],
self.heaplist[i]
            i = m

    def minimum(self, l):
        if 2*l+1 > self.length:
            return 2*l
        else:
            if self.heaplist[2*l] < self.heaplist[2*l+1]:
                return 2*l
            else:
                return 2*l+1

def heapSort(input):
    hlist = HEAP()
    o = []
    i = 0
    for item in input:
        hlist.buildheap(item)
    while i < len(input):
        min = hlist.removemin()
        o.append(min)
        i += 1
    return o

### INPLACE QUICK SORT FUNCTION ###
def InplaceQuickSort(array):
    def sorting(input, left, right):
        if right <= left:
            return
        pivot_index = random.randint(left, right)
        input[left], input[pivot_index] = input[pivot_index], input[left]
        i = left
        for j in range(left + 1, right + 1):
            if input[j] < input[left]:
                i += 1
                input[i], input[j] = input[j],
        input[i] input[i], input[left] = input[left],
        input[i] sorting(input, left, i - 1)
        sorting(input, i + 1, right)
    sorting(array, 0, len(array) - 1)

### MODIFIED QUICK SORT FUNCTION ###
def ModifiedQuickSort(input,left,right):
    if(left+10<=right):
        pivot=median(input,left,right)
        i=left
        j=right-2
        done=True
        while done:
            while(input[i]<pivot):
                i=i+1
            while(pivot<input[j]):
                j=j-1
            if(i<j):
                input[i],input[j]=input[j],input[i]
            else:
                done=False
        input[i],input[right-1]=input[right-1],input[i]
        ModifiedQuickSort(input,left,i-1)
        ModifiedQuickSort(input,i+1,right)
    else:
        InsertionSort_10subarrays(input,left,right)

def median(input,left,right):
    center=(left+right)//2
    if(input[center]<input[left]):
        input[left],input[center]=input[center],input[left]
    if (input[right] < input[left]):
        input[left], input[right] = input[right], input[left]
    if (input[right] < input[center]):
        input[right], input[center] = input[center], input[right]
    input[center],input[right-1]=input[right-1],input[center]
    return input[right-1]
```

```
### DRIVER AND MAIN CODE ###
def callingsorts(randominput,Start,n):
    InsertionInput=list(randominput)
    InplaceQuickInput = list(randominput)
    MergeInput=list(randominput)
    ModifiedQuickInput = list(randominput)
    l = len(ModifiedQuickInput)

    ### CALLING INSERTION SORT ###
    print("\n\n FOR " + str(Start) + " INPUT SIZE : \n")
    print(" Random Input : " + str(InsertionInput))
    InsertStart = time.time()
    InsertionSort(InsertionInput)
    InsertEnd = time.time()
    print(" Insertion Sort : " + str(InsertionInput))
    print(" Execution Time : " + str((InsertEnd - InsertStart) * 1000) + " ms
")

print("*********************************************************************"
)

    ### CALLING MERGE SORT ###
    print(" Random Input : " + str(MergeInput))
    MergeStart = time.time()
    MergeSort(MergeInput)
    MergeEnd = time.time()
    print(" Merge Sort : " + str(MergeInput))
    print(" Execution Time : " + str((MergeEnd - MergeStart) * 1000) + " ms ")

print("*********************************************************************"
)

    ### CALLING HEAP SORT ###
    global output
    output = []
    HeapStart =  time.time()
    output = heapSort(randominput)
    HeapEnd= time.time()
    print(" Random Input : " + str(randominput))
    print(" Heap Sort : " + str(output))
    print(" Execution Time : " + str((HeapEnd - HeapStart) * 1000) + " ms ")

print("*********************************************************************"
)

    ### CALLING INPLACE QUICK SORT ###
    print(" Random Input : " + str(InplaceQuickInput))
    InplaceQuickStart = time.time()
    InplaceQuickSort(InplaceQuickInput)
    InplaceQuickEnd = time.time()
    print(" Inplace Quick Sort : " + str(InplaceQuickInput))
    print(" Execution Time : " + str((InplaceQuickEnd - InplaceQuickStart) *
1000) + "ms")
```

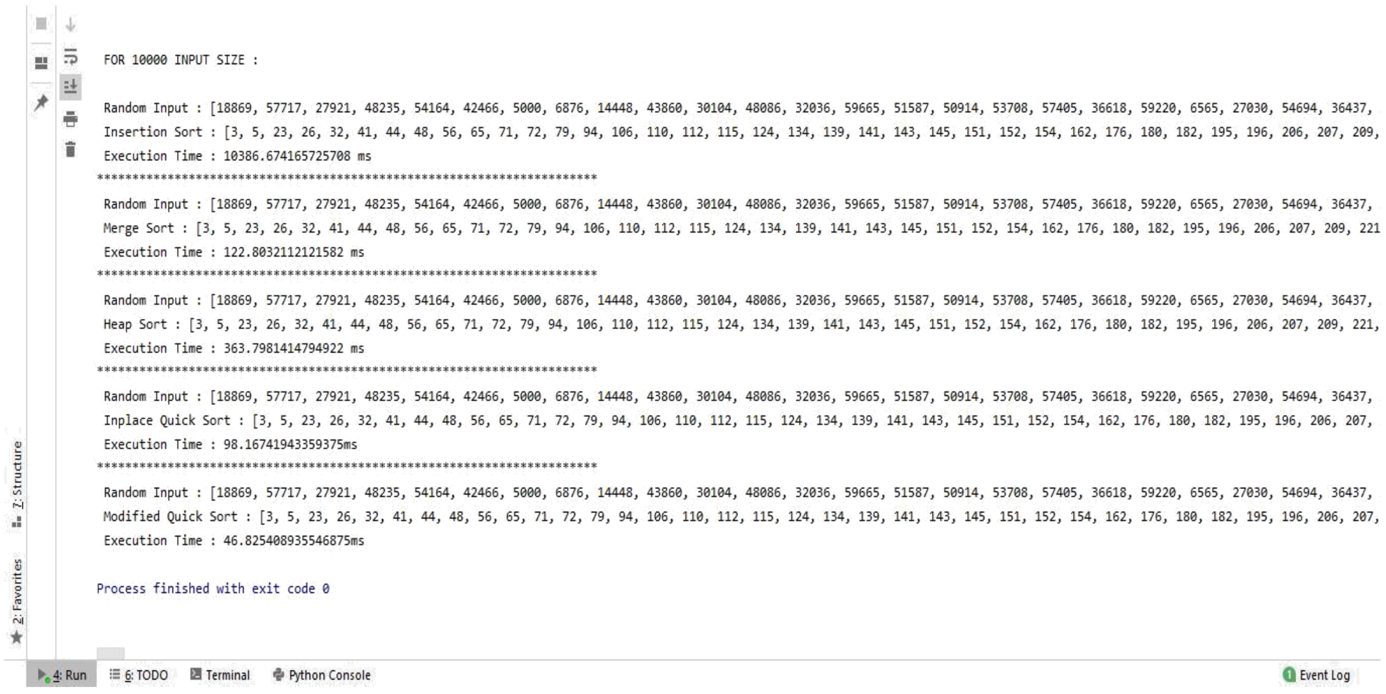
```
print("********************************************************************"
)

    ### CALLING MODIFIED QUICK SORT ###
    print(" Random Input : " + str(ModifiedQuickInput))
    ModifiedQuickStart = time.time()
    ModifiedQuickSort(ModifiedQuickInput, 0, l - 1)
    ModifiedQuickEnd = time.time()
    print(" Modified Quick Sort : " + str(ModifiedQuickInput))
    print(" Execution Time : " + str((ModifiedQuickEnd - ModifiedQuickStart) *
1000) + "ms")

def main(input_sizes_range,startrange,endrange):
    for i in range(startrange,endrange):
        INPUT = random.sample(range(1, 60000), input_sizes_range[i])
        n=len(INPUT)
        global heaplist
        heaplist = []
        callingsorts(INPUT,input_sizes_range[i],n)

randominput=[]
input_sizes_range=[1000,2000,4000,5000,10000,20000,30000,40000,50000]
main(input_sizes_range,0,len(input_sizes_range))
```

8.OUTPUT

- Output is displayed for all input size range as mentioned in the code.
- For each sorting algorithm output displays Input, Sorted output and execution time.
- We have run the code 3 times and considered the average of execution times for each sorting algorithm to plot the graph.

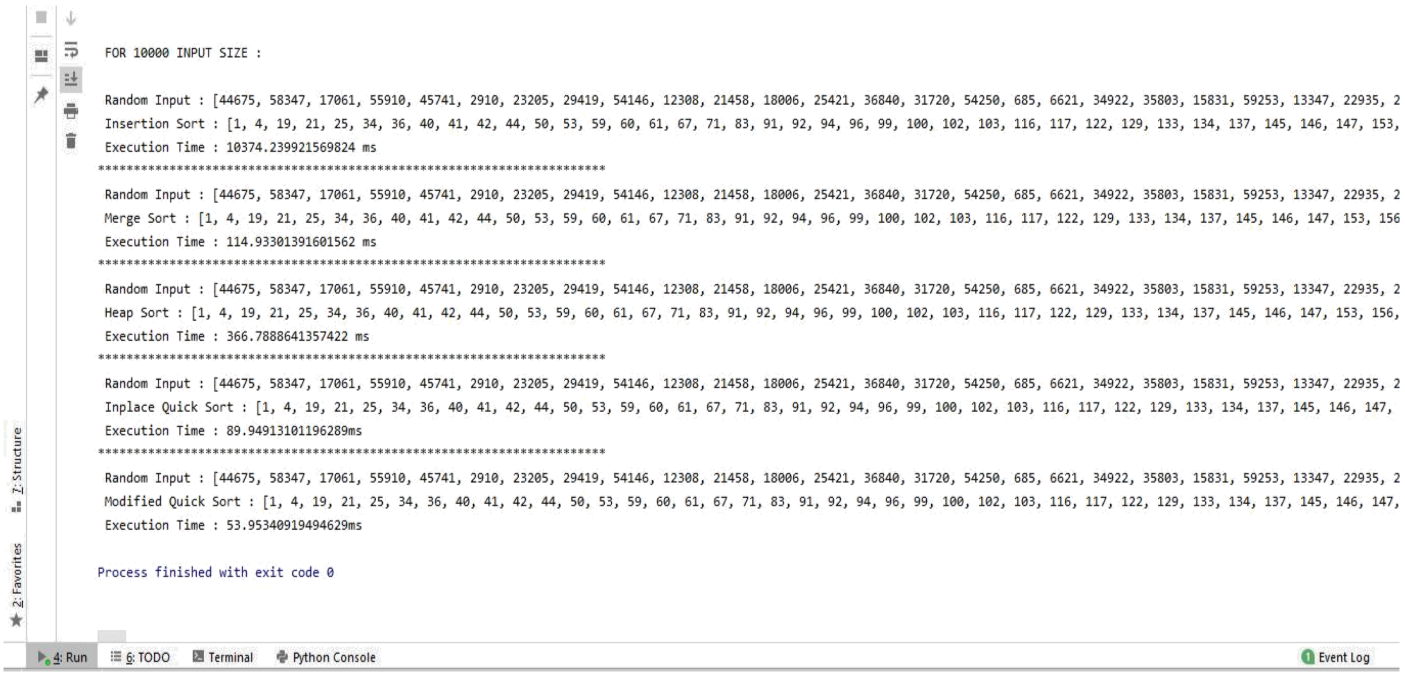
  Following are output screenshots for 10,000 input size for randomly generated input which is sorted by each sorting algorithm using the same input:

Output when run for 1st time :



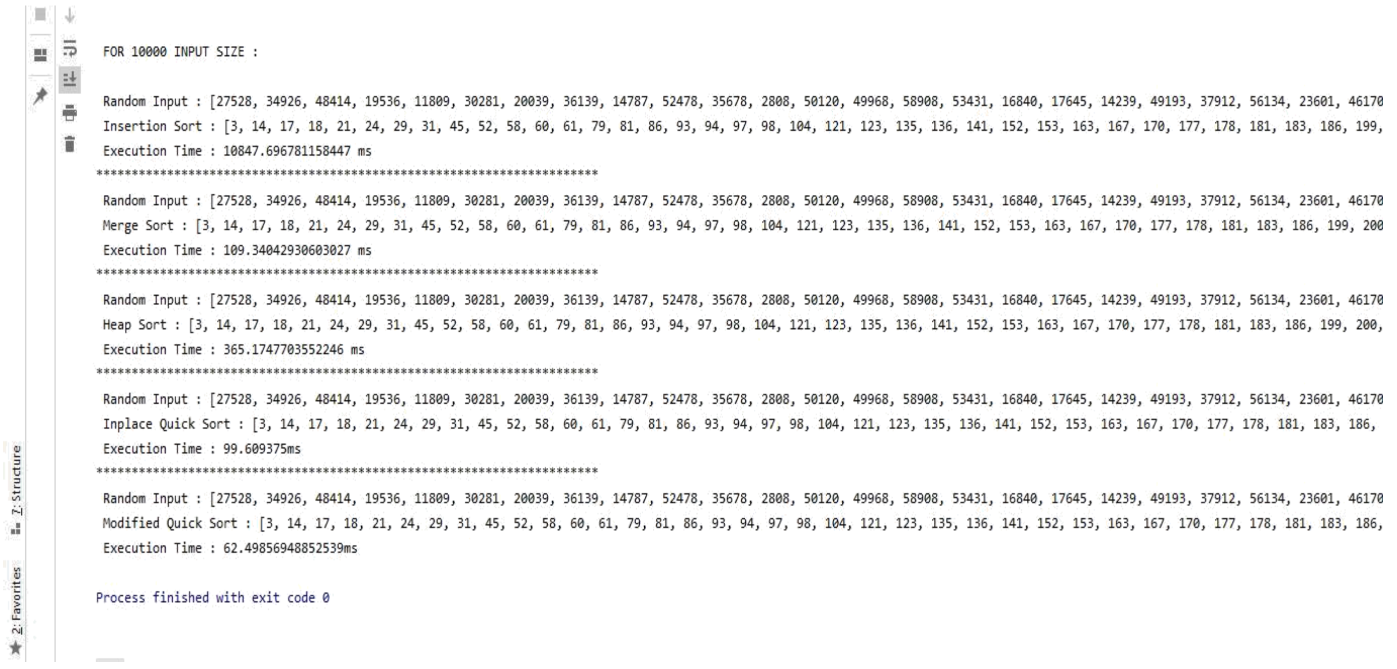
```
 FOR 10000 INPUT SIZE :

 Random Input : [18869, 57717, 27921, 48235, 54164, 42466, 5000, 6876, 14448, 43860, 30104, 48086, 32036, 59665, 51587, 50914, 53708, 57405, 36618, 59220, 6565, 27030, 54694, 36437,
 Insertion Sort : [3, 5, 23, 26, 32, 41, 44, 48, 56, 65, 71, 72, 79, 94, 106, 110, 112, 115, 124, 134, 139, 141, 143, 145, 151, 152, 154, 162, 176, 180, 182, 195, 196, 206, 207, 209,
 Execution Time : 10386.674165725708 ms
**********************************************************************
 Random Input : [18869, 57717, 27921, 48235, 54164, 42466, 5000, 6876, 14448, 43860, 30104, 48086, 32036, 59665, 51587, 50914, 53708, 57405, 36618, 59220, 6565, 27030, 54694, 36437,
 Merge Sort : [3, 5, 23, 26, 32, 41, 44, 48, 56, 65, 71, 72, 79, 94, 106, 110, 112, 115, 124, 134, 139, 141, 143, 145, 151, 152, 154, 162, 176, 180, 182, 195, 196, 206, 207, 209, 221
 Execution Time : 122.8032112121582 ms
**********************************************************************
 Random Input : [18869, 57717, 27921, 48235, 54164, 42466, 5000, 6876, 14448, 43860, 30104, 48086, 32036, 59665, 51587, 50914, 53708, 57405, 36618, 59220, 6565, 27030, 54694, 36437,
 Heap Sort : [3, 5, 23, 26, 32, 41, 44, 48, 56, 65, 71, 72, 79, 94, 106, 110, 112, 115, 124, 134, 139, 141, 143, 145, 151, 152, 154, 162, 176, 180, 182, 195, 196, 206, 207, 209, 221,
 Execution Time : 363.7981414794922 ms
**********************************************************************
 Random Input : [18869, 57717, 27921, 48235, 54164, 42466, 5000, 6876, 14448, 43860, 30104, 48086, 32036, 59665, 51587, 50914, 53708, 57405, 36618, 59220, 6565, 27030, 54694, 36437,
 Inplace Quick Sort : [3, 5, 23, 26, 32, 41, 44, 48, 56, 65, 71, 72, 79, 94, 106, 110, 112, 115, 124, 134, 139, 141, 143, 145, 151, 152, 154, 162, 176, 180, 182, 195, 196, 206, 207,
 Execution Time : 98.16741943359375ms
**********************************************************************
 Random Input : [18869, 57717, 27921, 48235, 54164, 42466, 5000, 6876, 14448, 43860, 30104, 48086, 32036, 59665, 51587, 50914, 53708, 57405, 36618, 59220, 6565, 27030, 54694, 36437,
 Modified Quick Sort : [3, 5, 23, 26, 32, 41, 44, 48, 56, 65, 71, 72, 79, 94, 106, 110, 112, 115, 124, 134, 139, 141, 143, 145, 151, 152, 154, 162, 176, 180, 182, 195, 196, 206, 207,
 Execution Time : 46.825408935546875ms

Process finished with exit code 0
```

Output when run for 2nd time:



```
 FOR 10000 INPUT SIZE :

 Random Input : [44675, 58347, 17061, 55910, 45741, 2910, 23205, 29419, 54146, 12308, 21458, 18006, 25421, 36840, 31720, 54250, 685, 6621, 34922, 35803, 15831, 59253, 13347, 22935, 2
 Insertion Sort : [1, 4, 19, 21, 25, 34, 36, 40, 41, 42, 44, 50, 53, 59, 60, 61, 67, 71, 83, 91, 92, 94, 96, 99, 100, 102, 103, 116, 117, 122, 129, 133, 134, 137, 145, 146, 147, 153,
 Execution Time : 10374.239921569824 ms
************************************************************************
 Random Input : [44675, 58347, 17061, 55910, 45741, 2910, 23205, 29419, 54146, 12308, 21458, 18006, 25421, 36840, 31720, 54250, 685, 6621, 34922, 35803, 15831, 59253, 13347, 22935, 2
 Merge Sort : [1, 4, 19, 21, 25, 34, 36, 40, 41, 42, 44, 50, 53, 59, 60, 61, 67, 71, 83, 91, 92, 94, 96, 99, 100, 102, 103, 116, 117, 122, 129, 133, 134, 137, 145, 146, 147, 153, 156
 Execution Time : 114.93301391601562 ms
************************************************************************
 Random Input : [44675, 58347, 17061, 55910, 45741, 2910, 23205, 29419, 54146, 12308, 21458, 18006, 25421, 36840, 31720, 54250, 685, 6621, 34922, 35803, 15831, 59253, 13347, 22935, 2
 Heap Sort : [1, 4, 19, 21, 25, 34, 36, 40, 41, 42, 44, 50, 53, 59, 60, 61, 67, 71, 83, 91, 92, 94, 96, 99, 100, 102, 103, 116, 117, 122, 129, 133, 134, 137, 145, 146, 147, 153, 156,
 Execution Time : 366.7888641357422 ms
************************************************************************
 Random Input : [44675, 58347, 17061, 55910, 45741, 2910, 23205, 29419, 54146, 12308, 21458, 18006, 25421, 36840, 31720, 54250, 685, 6621, 34922, 35803, 15831, 59253, 13347, 22935, 2
 Inplace Quick Sort : [1, 4, 19, 21, 25, 34, 36, 40, 41, 42, 44, 50, 53, 59, 60, 61, 67, 71, 83, 91, 92, 94, 96, 99, 100, 102, 103, 116, 117, 122, 129, 133, 134, 137, 145, 146, 147,
 Execution Time : 89.94913101196289ms
************************************************************************
 Random Input : [44675, 58347, 17061, 55910, 45741, 2910, 23205, 29419, 54146, 12308, 21458, 18006, 25421, 36840, 31720, 54250, 685, 6621, 34922, 35803, 15831, 59253, 13347, 22935, 2
 Modified Quick Sort : [1, 4, 19, 21, 25, 34, 36, 40, 41, 42, 44, 50, 53, 59, 60, 61, 67, 71, 83, 91, 92, 94, 96, 99, 100, 102, 103, 116, 117, 122, 129, 133, 134, 137, 145, 146, 147,
 Execution Time : 53.95340919494629ms

Process finished with exit code 0
```

Output when run for 3rd time:



```
 FOR 10000 INPUT SIZE :

 Random Input : [27528, 34926, 48414, 19536, 11809, 30281, 20039, 36139, 14787, 52478, 35678, 2808, 50120, 49968, 58908, 53431, 16840, 17645, 14239, 49193, 37912, 56134, 23601, 46170
 Insertion Sort : [3, 14, 17, 18, 21, 24, 29, 31, 45, 52, 58, 60, 61, 79, 81, 86, 93, 94, 97, 98, 104, 121, 123, 135, 136, 141, 152, 153, 163, 167, 170, 177, 178, 181, 183, 186, 199,
 Execution Time : 10847.696781158447 ms
************************************************************************
 Random Input : [27528, 34926, 48414, 19536, 11809, 30281, 20039, 36139, 14787, 52478, 35678, 2808, 50120, 49968, 58908, 53431, 16840, 17645, 14239, 49193, 37912, 56134, 23601, 46170
 Merge Sort : [3, 14, 17, 18, 21, 24, 29, 31, 45, 52, 58, 60, 61, 79, 81, 86, 93, 94, 97, 98, 104, 121, 123, 135, 136, 141, 152, 153, 163, 167, 170, 177, 178, 181, 183, 186, 199, 200
 Execution Time : 109.34042930603027 ms
************************************************************************
 Random Input : [27528, 34926, 48414, 19536, 11809, 30281, 20039, 36139, 14787, 52478, 35678, 2808, 50120, 49968, 58908, 53431, 16840, 17645, 14239, 49193, 37912, 56134, 23601, 46170
 Heap Sort : [3, 14, 17, 18, 21, 24, 29, 31, 45, 52, 58, 60, 61, 79, 81, 86, 93, 94, 97, 98, 104, 121, 123, 135, 136, 141, 152, 153, 163, 167, 170, 177, 178, 181, 183, 186, 199, 200,
 Execution Time : 365.1747703552246 ms
************************************************************************
 Random Input : [27528, 34926, 48414, 19536, 11809, 30281, 20039, 36139, 14787, 52478, 35678, 2808, 50120, 49968, 58908, 53431, 16840, 17645, 14239, 49193, 37912, 56134, 23601, 46170
 Inplace Quick Sort : [3, 14, 17, 18, 21, 24, 29, 31, 45, 52, 58, 60, 61, 79, 81, 86, 93, 94, 97, 98, 104, 121, 123, 135, 136, 141, 152, 153, 163, 167, 170, 177, 178, 181, 183, 186,
 Execution Time : 99.609375ms
************************************************************************
 Random Input : [27528, 34926, 48414, 19536, 11809, 30281, 20039, 36139, 14787, 52478, 35678, 2808, 50120, 49968, 58908, 53431, 16840, 17645, 14239, 49193, 37912, 56134, 23601, 46170
 Modified Quick Sort : [3, 14, 17, 18, 21, 24, 29, 31, 45, 52, 58, 60, 61, 79, 81, 86, 93, 94, 97, 98, 104, 121, 123, 135, 136, 141, 152, 153, 163, 167, 170, 177, 178, 181, 183, 186,
 Execution Time : 62.49856948852539ms

Process finished with exit code 0
```

9.GRAPHS

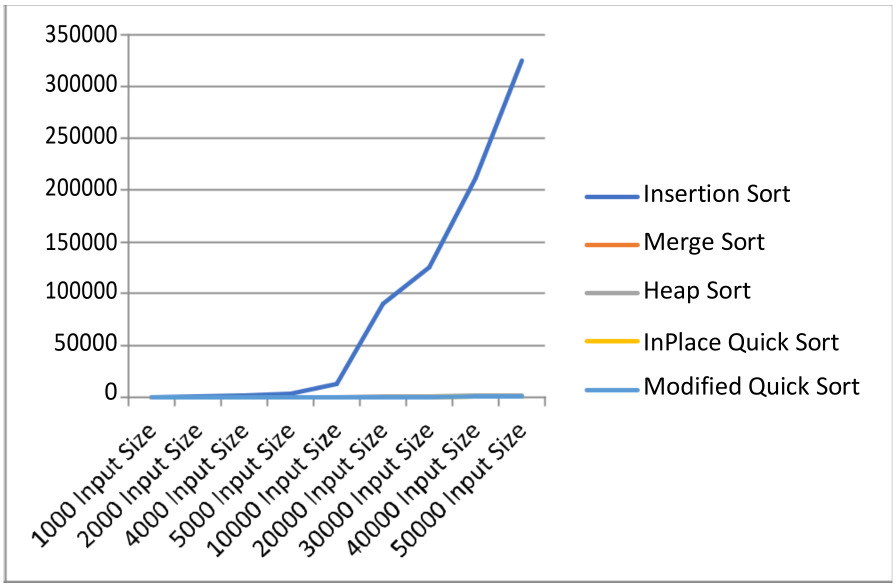
Quick Sort is faster for random and reverse sorted input, whereas Insertion Sort is efficient in case of sorted input. This can be observed from following graphs:

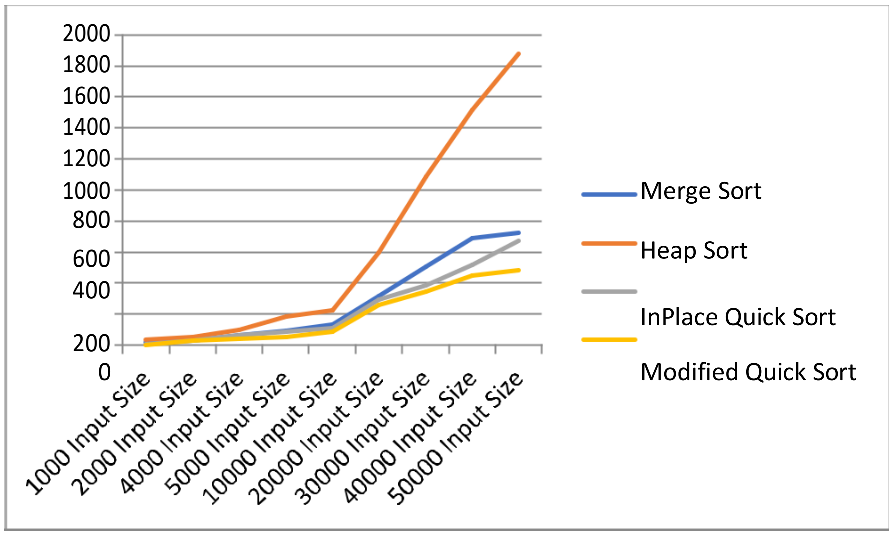
1. For Random Input :

Execution Times –

Input Size	Insertion Sort	Merge Sort	Heap Sort	InPlace Quick Sort	Modified Quick Sort
1000 Input Size	110.12	15.52	32.34	7.55	0
2000 Input Size	546.7	30.46	54.23	31.13	28.76
4000 Input Size	2150.83	62.19	96.78	62.4	38.63
5000 Input Size	3278.79	93.64	183.34	86.32	52.31
10000 Input Size	10533.76	118.95	365.78	109.76	62.34
20000 Input Size	89765.34	318.34	596.32	291.12	258.21
30000 Input Size	125742.9	503.78	1087.9	383.88	347.91
40000 Input Size	211360.89	687.93	1516.09	514.32	445.42
50000 Input Size	324568.4	723.43	1876.89	671.06	480.01

Graphs comparing all sorting algorithms:



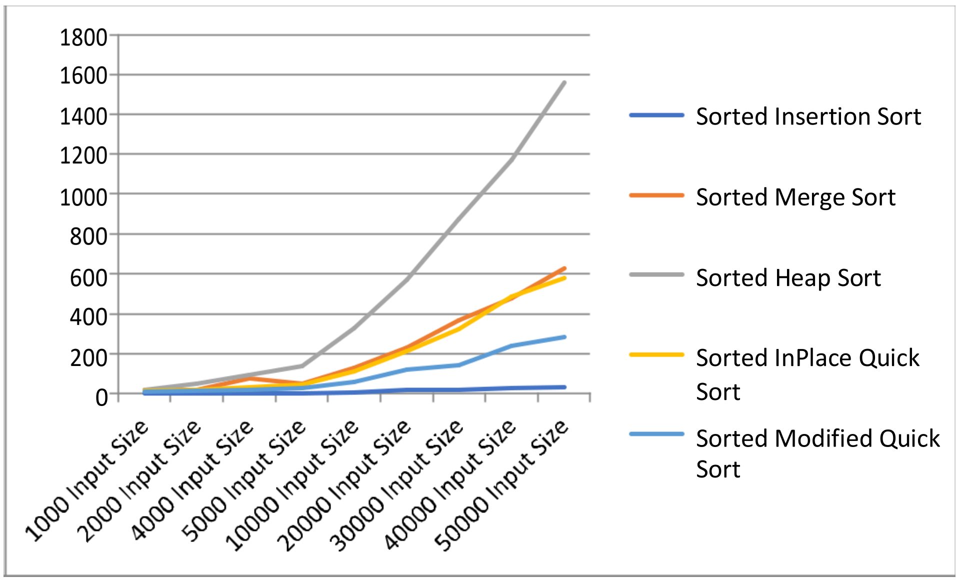


2. For Sorted Input :

Execution Times –

Input Size	Sorted Insertion Sort	Sorted Merge Sort	Sorted Heap Sort	Sorted InPlace Quick Sort	Sorted Modified Quick Sort
1000 Input Size	0	15.63	18.97	15.11	7.04
2000 Input Size	0	18.46	48.76	16.87	12.34
4000 Input Size	0.04	77.08	91.77	31.23	15.67
5000 Input Size	0.6	46.87	138.99	45.41	27.34
10000 Input Size	3.99	128.54	324.83	109.78	55.85
20000 Input Size	15.62	229.34	568.84	210.65	118.79
30000 Input Size	16.86	364.32	876.45	323.14	140.6
40000 Input Size	26.46	478.86	1168.79	487.13	238.9
50000 Input Size	31.24	627.46	1560.38	579.45	284.29

Graph comparing sorting algorithms:

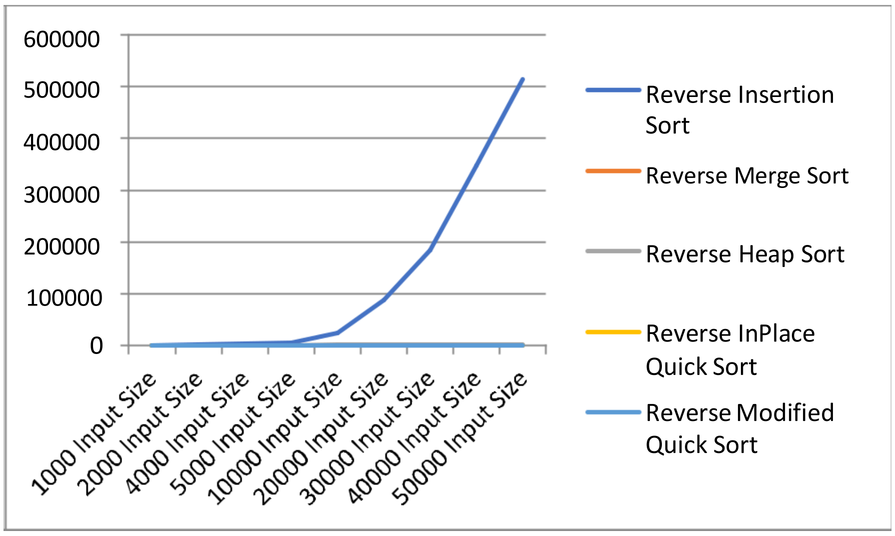


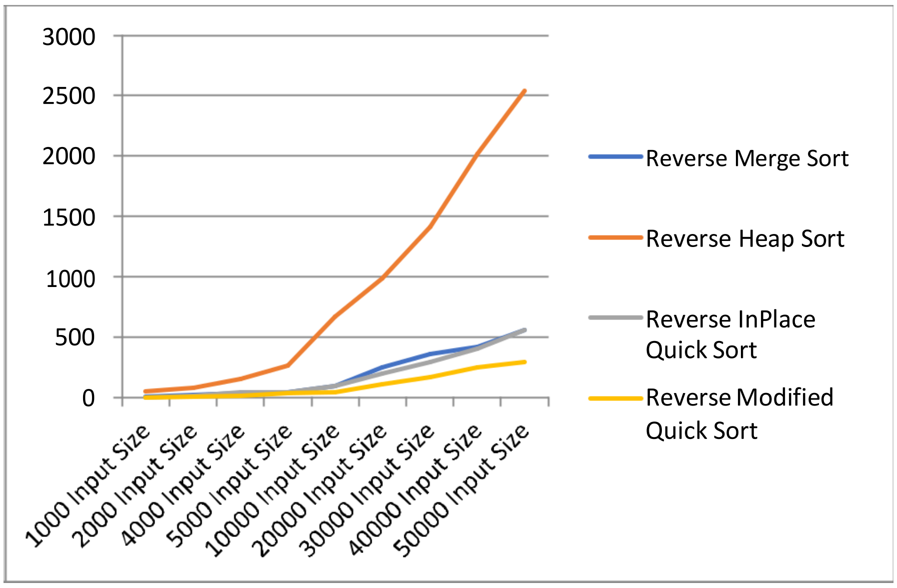
3. For Reverse Sorted Input

Execution Times –

Input Size	Reverse Insertion Sort	Reverse Merge Sort	Reverse Heap Sort	Reverse InPlace Quick Sort	Reverse Modified Quick Sort
1000 Input Size	228.59	7.68	48.74	0	0
2000 Input Size	876.84	18.75	78.12	15.63	7.29
4000 Input Size	3376.37	41.55	151.1	46.85	15.62
5000 Input Size	5469.91	46.88	265.62	46.85	38.25
10000 Input Size	23990.31	93.73	672.47	97.13	46.28
20000 Input Size	86879.97	250.02	986.22	201.62	109.37
30000 Input Size	183369.75	360.07	1416.93	296.83	173.95
40000 Input Size	347871.43	421.63	2018.1	406.03	249.96
50000 Input Size	513471.73	557.99	2540.03	563.22	296.85

Graph comparing each sorting algorithms:





10. Complexity Analysis

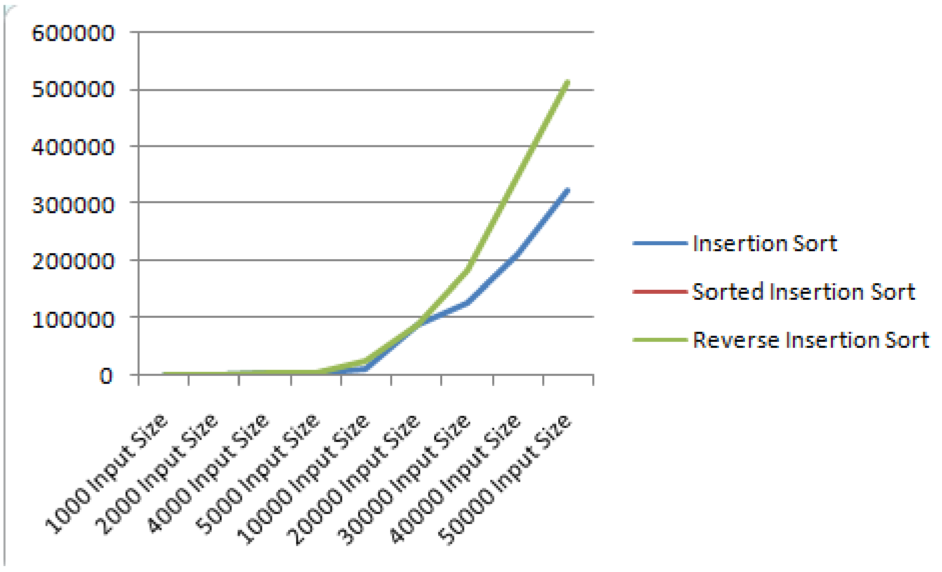
Insertion Sort:

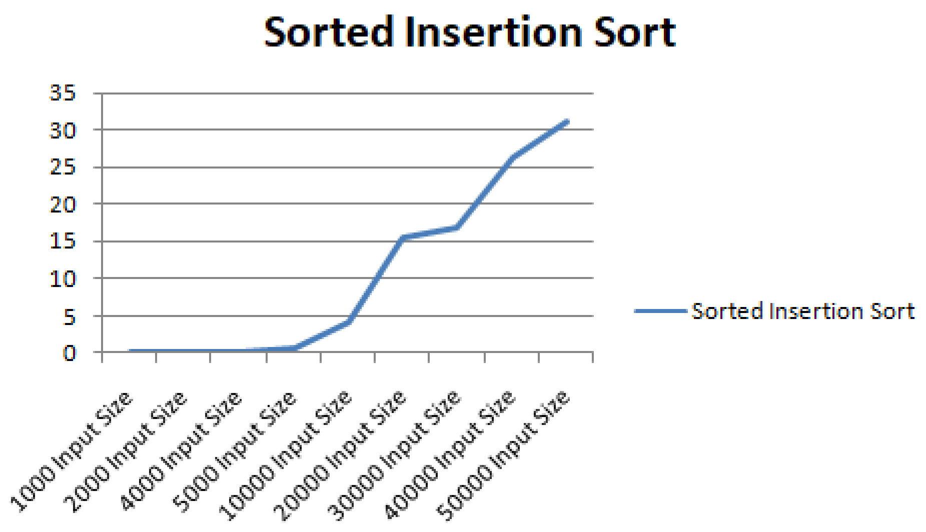
- The "worst-case" in insertion sort occurs when the input is in reverse order. The reason is, in such case the element has to be shifted to the beginning of array each time. So, the time for the worst case will be **O(n^2)**.
- The "Best-case" in insertion sort occurs when the input is in sorted order. In such case, there will be no swaps, so the entire array is iterated only once. So, the time for best case will be **O(n)**.
- The "Average-case" in insertion sort occurs when half of the elements are less than an particular element and other half are greater. The time is same as worst case which is **O(n^2)**.

Execution Times:

Input Size	Insertion Sort	Sorted Insertion Sort	Reverse Insertion Sort
1000 Input Size	110.12	0	228.59
2000 Input Size	546.7	0	876.84
4000 Input Size	2150.83	0.04	3376.37
5000 Input Size	3278.79	0.6	5469.91
10000 Input Size	10533.76	3.99	23990.31
20000 Input Size	89765.34	15.62	86879.97
30000 Input Size	125742.9	16.86	183369.75
40000 Input Size	211360.89	26.46	347871.43
50000 Input Size	324568.4	31.24	513471.73

Graph:





Sorted Insertion Sort
35
30
25
20
15
10
5
0
1000 Input Size
2000 Input Size
4000 Input Size
5000 Input Size
10000 Input Size
20000 Input Size
30000 Input Size
40000 Input Size
50000 Input Size
Sorted Insertion Sort

Merge Sort :

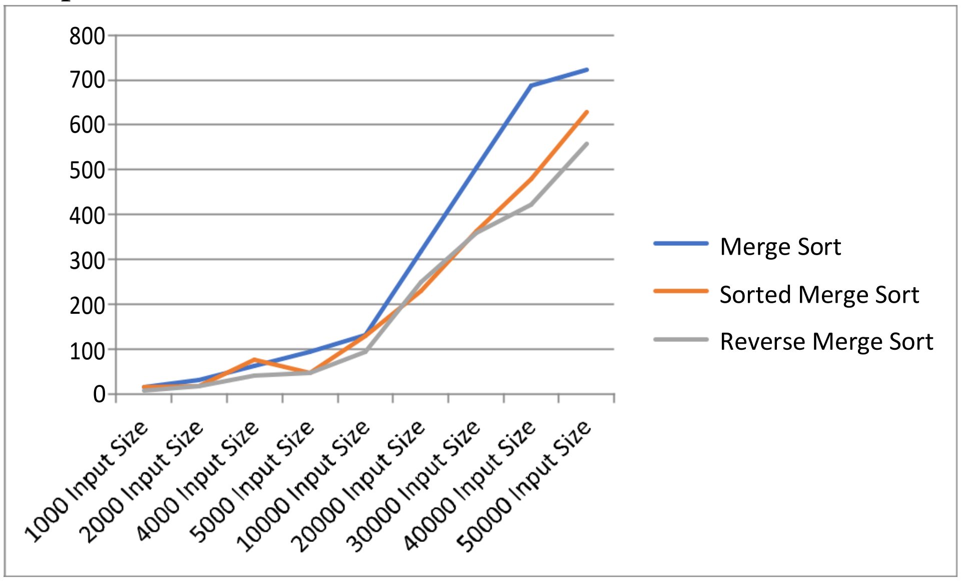
In merge sort, it follows Divide and Conquer approach.

- The input array will be divided into halves. The time to do such operation(split) is ‘logn’, where n is the input size.
- After dividing, to merge it will be linear time i.e ‘n’ as there will be n operations.
- So, Merge sort does logn splits for which n is the cost to merge them which is **O(nlogn)** for entire sorting.

Execution Times –

Input Size	Merge Sort	Sorted Merge Sort	Reverse Merge Sort
1000 Input Size	15.52	15.63	7.68
2000 Input Size	30.46	18.46	18.75
4000 Input Size	62.19	77.08	41.55
5000 Input Size	93.64	46.87	46.88
10000 Input Size	118.95	128.54	93.73
20000 Input Size	318.34	229.34	250.02
30000 Input Size	503.78	364.32	360.07
40000 Input Size	687.93	478.86	421.63
50000 Input Size	723.43	627.46	557.99

Graph –



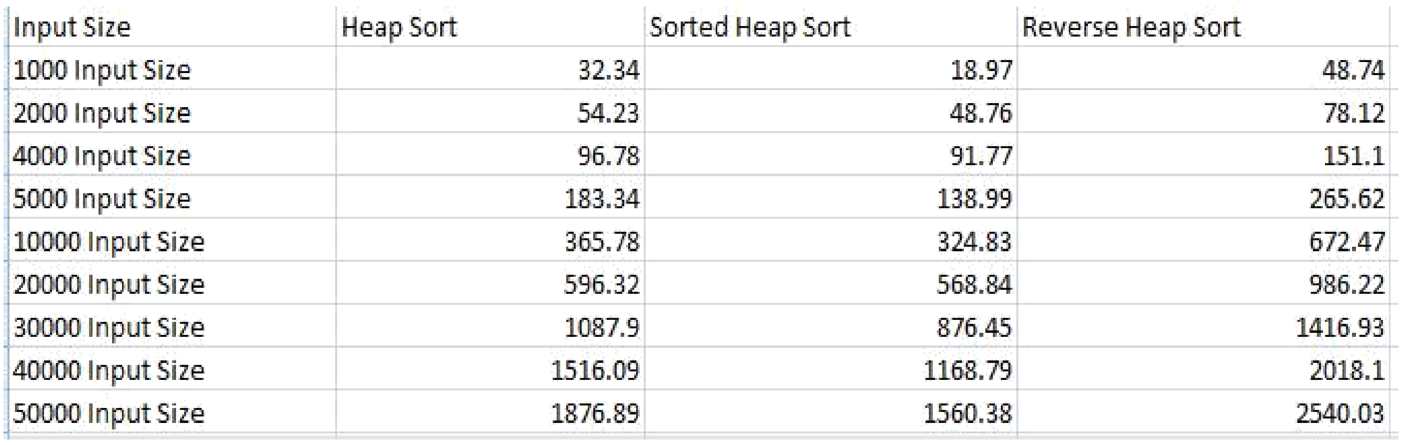
Heap Sort :

In heap sort, there are two phases – building heap and removing min.

- For building heap, ‘n’ elements are inserted into the heap array and after every insertion, the array is heapified to maintain the heap order property. So, for ‘n’ elements it takes ‘logn’ time to heapify. So, the time complexity will be O(nlogn).
- For removing min, first element is removed and then heapified. So, the total ‘n’ elements will be removed in the sorted order and heapified after every removal inorder to maintain heap property. So, the time is O(nlogn).

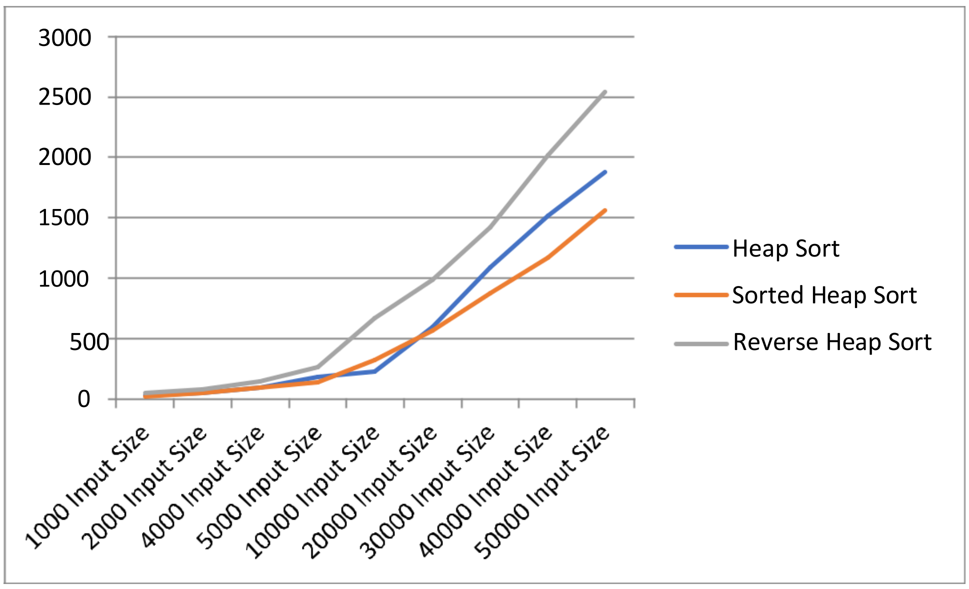
The time complexity for all the cases in heap sort is **O(nlogn)**.

Execution Time –



Input Size	Heap Sort	Sorted Heap Sort	Reverse Heap Sort
1000 Input Size	32.34	18.97	48.74
2000 Input Size	54.23	48.76	78.12
4000 Input Size	96.78	91.77	151.1
5000 Input Size	183.34	138.99	265.62
10000 Input Size	365.78	324.83	672.47
20000 Input Size	596.32	568.84	986.22
30000 Input Size	1087.9	876.45	1416.93
40000 Input Size	1516.09	1168.79	2018.1
50000 Input Size	1876.89	1560.38	2540.03

Graph –



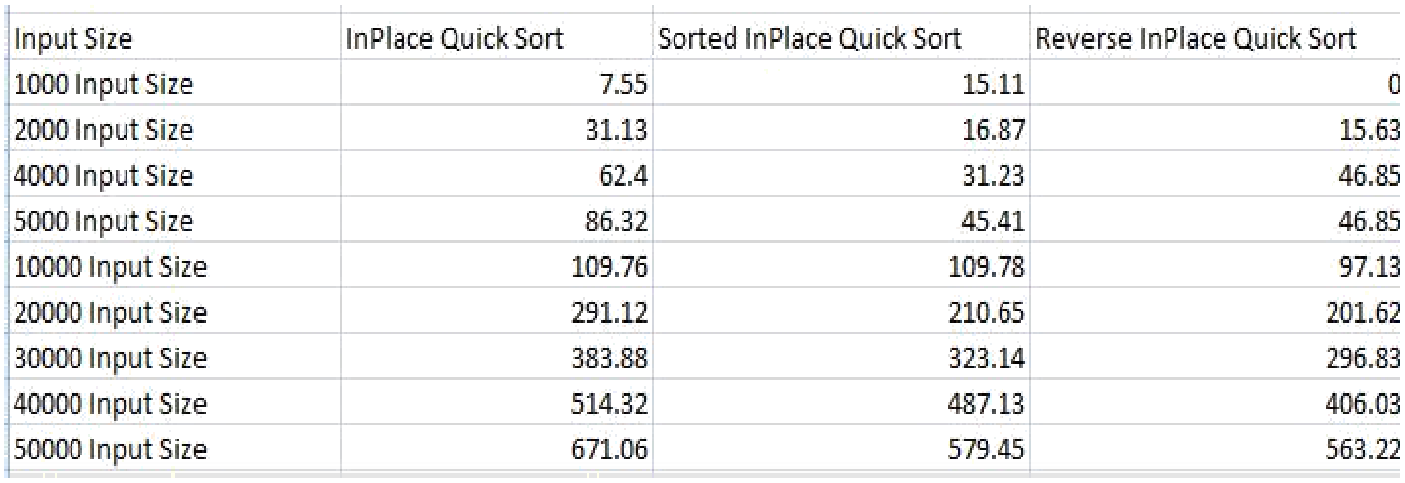
<u>In-Place Quick Sort :</u>

- The Worst-case for In-place quick sort occurs when pivot is either the smallest element in the array or the largest element. When the array is sorted or reversely sorted, In-place happens in worst case time. The time complexity in worst case is **O(n^2)**.

- The Best-Case for In-place quick sort occurs when pivot is the middle element and the time complexity is **O(nlogn)**.

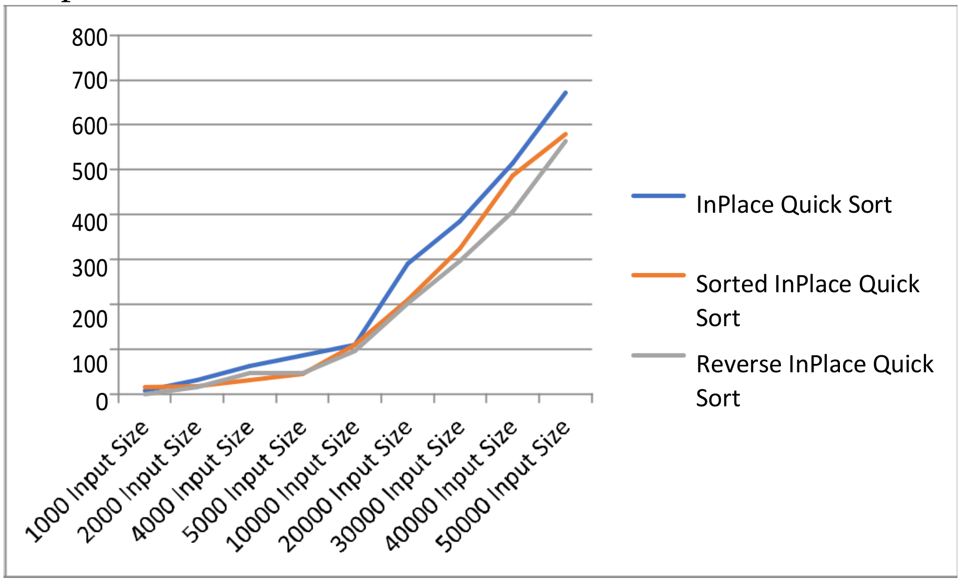
- The Average-Case for In-place quick sort is

  O(nlogn). Execution Time :



Input Size	InPlace Quick Sort	Sorted InPlace Quick Sort	Reverse InPlace Quick Sort
1000 Input Size	7.55	15.11	0
2000 Input Size	31.13	16.87	15.63
4000 Input Size	62.4	31.23	46.85
5000 Input Size	86.32	45.41	46.85
10000 Input Size	109.76	109.78	97.13
20000 Input Size	291.12	210.65	201.62
30000 Input Size	383.88	323.14	296.83
40000 Input Size	514.32	487.13	406.03
50000 Input Size	671.06	579.45	563.22

Graph :



<u>**Modified-Quick Sort(Median-of-three):**</u>

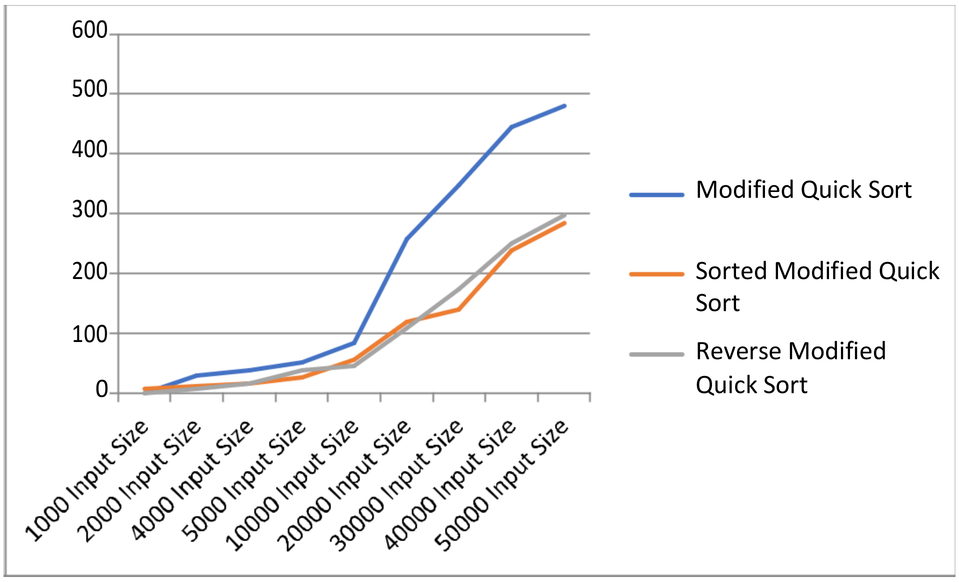
In modified quick sort, we use median-of-three to choose pivot and also when the subarray is of size 10, insertion sort is used to sort because quick sort does not perform well for small inputs.

- The Worst-case for this quick sort happens when the pivot is either the smallest or the largest element in the array. The time complexity is **O(n^2)**.
- The Best-case for this quick sort happens when the pivot choosen divides the array into equal halves. Time complexity is **O(nlogn)**.
- The Average-case for this quick sort considers the sizes are equally likely and this assumption is valid for median-of-three strategy. The time complexity is **O(nlogn)**.

Execution Time :

Input Size	Modified Quick Sort	Sorted Modified Quick Sort	Reverse Modified Quick Sort
1000 Input Size	0	7.04	0
2000 Input Size	28.76	12.34	7.29
4000 Input Size	38.63	15.67	15.62
5000 Input Size	52.31	27.34	38.25
10000 Input Size	62.34	55.85	46.28
20000 Input Size	258.21	118.79	109.37
30000 Input Size	347.91	140.6	173.95
40000 Input Size	445.42	238.9	249.96
50000 Input Size	480.01	284.29	296.85

Graph :


1000 Input Size
2000 Input Size
4000 Input Size
5000 Input Size
10000 Input Size
20000 Input Size
30000 Input Size
40000 Input Size
50000 Input Size
Modified Quick Sort
Sorted Modified Quick Sort
Reverse Modified Quick Sort