Project-I
COMPARISON -BASED SORTING ALGORITHM
ITCS-6114 FALL 2021

Submitted by

Ramanathan Sivaramakrishnan
rsivara1@unccc.edu

Sharat Sindoor
ssindoor@uncc.edu

Project Overview:

The following comparison-based sorting algorithms are implemented and performance of each of the sorting algorithm is observed for input size ranging from 1000 to 60000 on three different cases:

- ➔ Randomly generated input numbers for an array.
- ➔ Sorted array.
- ➔ Reversely sorted.

The sorting algorithms are:

- ➔ Insertion Sort.
- ➔ Merge Sort.
- ➔ Heap Sort.
- ➔ In-Place Quicksort(any random item or the first or the last item of your input can be pivot).

Modified Quicksort:

- ➔ Use median-of-three as pivot.
- ➔ For small sub-problem of size <=10 Use insertion sort.

And finding the run-time of each algorithm in nanoseconds and analysing them by plotting it in a graph.

Data Structures used:

List Data Structures from Python is used to implement various comparison-based sorting algorithms.

Comparison-Based Sorting Algorithms

Insertion Sort:

The insertion sort is one of the easy-to-understand comparison-based sorting algorithm which is similar to the way to sort the playing cards.When insertion sort is implemented the array gets virtually split into a sorted and an unsorted part.The elements in the unsorted part are compared with the elements before them and placed in the correct position of the sorted part.

Working Principle:

- ➔ Compare the element with its adjacent element.
- ➔ If the element is smaller than the other element compare with elements before until some smaller element or the start of the array is reached and swap with the element which fits correctly.
- ➔ Repeat the above steps until you place the last element of the unsorted part of an array in a correct position.

Advantage of Insertion Sort :

- ➔ It has a running time of O(n) for a partially sorted array.

Disadvantages of Insertion Sort:

- ➔ It has $O(n^2)$ running time in the average case and worst case(i.e,the array is reversely sorted).
- ➔ It takes $n^2/2$(approx)comparisons and exchanges.

Time Complexity:

Best case:O(n)

Worst case:$O(n^2)$
Auxiliary Space: O(n)
Merge Sort:
The Merge Sort is based on the divide and-conquer paradigm.The algorithm divides the input array into two halves if the input size is greater than certain threshold,calls itself recursively by splitting it into two halves until it reaches a single element and then merges the sorted sub-arrays that are made until a complete sorted array is formed.
Advantages of Merge Sort:
- ➔ It is quicker for larger lists,unlike insertion sort it doesn't traverse through the whole list several times
- ➔ It has a consistent running time.

Disadvantages of Merge Sort:
- ➔ It is a little slower for smaller tasks.
- ➔ It repeats the entire process even thought the list is sorted
- ➔ It takes additional memory to the elements of the splitted array.

Time Complexity: O(nlogn)
Auxiliary Space: O(n)

Heap Sort:
The Heap sort is a comparison-based sorting algorithm which is based on Binary Heap data Structure.A heap can be used to sort an array of element which is represented by an array where the parent node is at chiild_node(index)/2,parent's left child is at 2*(parent_node_index)+1 and the right child is at 2*(parent_node_index)+2.First,we need to do Heapify where we need to build a complete binary tree and represent it in the form of an array and satisfy the Heap condition in this case Max Heap (where the largest element is the root element).
Working Principle:
- ➔ Once the tree satisfies the complete binary tree condition and Heap condition(for example Max Heap),the largest element will be the root element.
- ➔ So,remove the root element and put at the end of the array.And place the last item into the vacant place.
- ➔ Reduce the heap size by 1.
- ➔ Repeat the heapify process so the largest element is at the root of the tree.
- ➔ The above steps are repeated until all the elements are sorted.

Advantages of Heap Sort:
- ➔ Time required to sort the algorithms increases logarithmically when the input size increases while in other algorithms increase exponentially when the input size increases.
- ➔ It is an in-place sorting algorithm.

Disadvantages of Heap Sort:
Heap sort is not stable when compared with the merge sort.
Time Complexity:O(nlogn).
Auxiliary Space:O(1).

Quick Sort:
The Quick Sort is similar in essence to Merge Sort such that it is based on the divide-and-conquer algorithm.The algorithm selects an element as pivot and divides the given

sequence/array around the selected pivot. In this particular Project, median as pivot and random number selection are implemented.

Advantages of Quick Sort:

- ➔ It does not require additional memory as it sorts in place.
- ➔ It has an extremely short inner loop.

Disadvantages of Quick Sort:

- ➔ It requires quadratic time in the worst case.
- ➔ It is recursive, especially if recursion is not available, the implementation is extremely complicated.
- ➔ A simple mistake in implementation can go unnoticed and cause it to perform badly.

Time Complexity:

Best case:O(nlogn)

Worst case:O(n^2)

Auxiliary Space: O(log n)

Modified Quick Sort:

Working Principle:

- ➔ In modified quicksort the pivot selection is done by taking the median of the first,last and middle element of the array.
- ➔ In this algorithm the first,last and middle are sorted according to their value and then the array[middle] is selected as pivot.So,by this method we have already sorted 3 elements hence the probability of choosing a bad pivot decreases.
- ➔ We will continue to perform the normal quicksort till the array length becomes 10.
- ➔ After the array length becomes 10 or less than 10 we can sort the array using insertion sort.

And the graph values are plotted by taking an average after running for several times.And the recursion limit depth is handled by setting the recursion depth limit to (10^6).

Source Code:

Insertion sort.py

```
def sorting_insertion(arr):
  #Traversing through the array from 0 to len(arr)
  for ax in range(1, len(arr)):
    bz = ax-1
    num = arr[ax]
    #swap the element if it is lower than the element before it.
    while bz>=0 and arr[bz]>num:
      arr[bz+1] = arr[bz]
      bz = bz-1
  arr[bz+1]=num
```

Merge_sort.py

```
def merge_sort(arr):
  if len(arr)>1:
    m = len(arr)//2
    larr = arr[:m]
    rarr = arr[m:]
    # Splitting the input array into two halves
```

```
        merge_sort(larr)
        merge_sort(rarr)

        ij=0
        kl=0
        mn=0
        # Comparing the values of both larr and rarr inorder to create a sorted
        # array
        while ij < len(larr) and kl < len(rarr):
            if larr[ij] < rarr[kl]:
                arr[mn]=larr[ij]
                ij=ij+1
            else:
                arr[mn]=rarr[kl]
                kl=kl+1
            mn=mn+1
        #Checking if there are any element present in either of the array and appending
        # them into a sorted array.
        while ij < len(larr):
            arr[mn]=larr[ij]
            ij=ij+1
            mn=mn+1

        while kl < len(rarr):
            arr[mn]=rarr[kl]
            kl=kl+1
            mn=mn+1
    return (arr)
```

Quick_sort.py

```
import random
def parting(seq, smallest, highest):
    ax = (smallest - 1)
    pivot_point = seq[random.randint(smallest,highest)]
    for bz in range(smallest, highest):
        if seq[bz] <= pivot_point:
            ax = ax + 1
            seq[ax], seq[bz] = seq[bz], seq[ax]
    seq[ax + 1], seq[highest] = seq[highest], seq[ax + 1]
    return (ax + 1)

def sortingquick(seq, smallest, highest):
    if smallest < highest:
        op = parting(seq, smallest, highest)
        sortingquick(seq, smallest, op - 1)
        sortingquick(seq, op + 1, highest)
```

```
def sorting_quick(numbers):
   seq = numbers
   sortingquick(seq, 0, len(seq) - 1)
   return seq

# Finding the middle element from the array.
middleC = 0
def middle1(a, b, c):
   if ( a - b) * (c - a) >= 0:
      return a
   elif (b - a) * (c - b) >= 0:
      return b
   else:
      return c
# Taking the median for takin
def partition_median(sequence, smallest_val_seq, highest_val_seq):
   small = sequence[smallest_val_seq]
   high = sequence[highest_val_seq - 1]
   length = highest_val_seq - smallest_val_seq
   middle = sequence[smallest_val_seq + length // 2]
   pivot_point = middle1(small, high, middle)
   pivot_index = sequence.index(pivot_point)
   sequence[pivot_index] = sequence[smallest_val_seq]
   sequence[smallest_val_seq] = pivot_point
   ax = smallest_val_seq + 1
   for bz in range(smallest_val_seq + 1, highest_val_seq):
      if sequence[bz] < pivot_point:
         temp_var = sequence[bz]
         sequence[bz] = sequence[ax]
         sequence[ax] = temp_var
         ax += 1

   high_End_Val = sequence[smallest_val_seq]
   sequence[smallest_val_seq] = sequence[ax - 1]
   sequence[ax - 1] = high_End_Val
   return ax - 1
# quicksort by taking the middle element
def quicksort_middle(sequence, smallest_index, highest_index):
   global middleC
   if smallest_index+ 10  <= highest_index:
      new_pivot_index = partition_median(sequence, smallest_index, highest_index)
      middleC += (highest_index - smallest_index - 1)
      quicksort_middle(sequence, smallest_index, new_pivot_index)
      quicksort_middle(sequence, new_pivot_index + 1, highest_index)

   else:
      insertion_sortting(sequence,smallest_index,highest_index)
```

```
# In modified quicksort if the array length comes to 10 we can perform insertion sort
def insertion_sortting(sequence,a,b):
   for ax in range(a, b):
      bz = ax
      while bz > 0 and sequence[bz] < sequence[bz-1]:
         sequence[bz],sequence[bz-1]=sequence[bz-1],sequence[bz]
         bz = bz - 1
#modified quick sort
def mquick_sort(seq):
   quicksort_middle(seq, 0, len(seq))
   return seq
```

Heap_sort.py

```
heap_entered=[]
arr_size=0
arr_sorted=[]
# creating a heap
def heap_generate(sequence):
   global heap_entered
   heap_entered = [0] * (len(sequence)+1)
   for ax in range(0,len(sequence)):
      enter(sequence[ax]);

def enter(x):
   global arr_size
   arr_size = arr_size +1
   po=arr_size
   heap_entered[po]=x
   up_shift_bubble(po)
# For generating a heap using bubble-up
def up_shift_bubble(place):
   x_id = place // 2;
   y_id = place;
   while (heap_entered[x_id] > heap_entered[y_id] and y_id > 0):
      shift(y_id, x_id);
      y_id = x_id;
      x_id = x_id // 2;
def shift(a,b):
   temp_var = heap_entered[a];
   heap_entered[a] = heap_entered[b];
   heap_entered[b] = temp_var;
# for generating the heap function call.
def sorting_heap1(sequence):
   heap_generate(sequence);
# finding the minimum for sorting purposes
def min_find():
   global arr_size
```

```
    small = heap_entered[1]
    heap_entered[1]=heap_entered[arr_size]
    heap_entered[arr_size] = 0
    down_shift(1)
    arr_size=arr_size -1
    return small
def down_shift(lm):
    small = lm
    LC_ID = 2 * lm
    RC_ID = 2 * lm + 1
    if (LC_ID < arr_size  and heap_entered[small] > heap_entered[LC_ID]):
        small = LC_ID
    if (RC_ID < arr_size  and heap_entered[small] > heap_entered[RC_ID]):
        small = RC_ID
    if (small != lm):
        shift(lm, small)
        down_shift(small)
#Heap sort driving function
def sorting_heap(sequence):
    arr_sorted=[0]*len(sequence)
    sorting_heap1(sequence)
    for ax in range(0,len(sequence)):
        arr_sorted[ax]=min_find()
    return arr_sorted
```

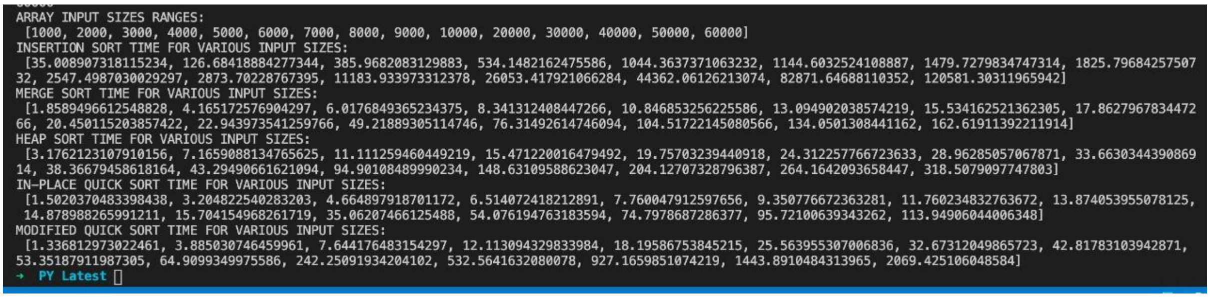
Driver Code for Random Number,Reverse Sorted Array,Sorted Array Input:
Random_numbers_driver.py

```
import random
import time
import sys
from statistics import mean
from heap_sort import sorting_heap
from insertion_sort import sorting_insertion
from merge_sort import merge_sort
from quick_sort import sorting_quick,mquick_sort

if _name_ == '_main_':
    insertion_sorting_average = []
    merge_sorting_average = []
    heap_sorting_average = []
    inplaceQuick_sorting_average = []
    medianQuick_sorting_average = []

   #Setting the recursion limit
    sys.setrecursionlimit(10**6)#For quick-sort to run properly for large data sets
```

```
arr=[]#INPUT SIZE ARRAY
for size in range(1000,10001,1000):
  arr.append(size)
for size in range(20000,60001,10000):
  arr.append(size)
print(arr)
#finding the running-time for the each of the sorting algorithm
for indexy in range(0, len(arr)):
  times_run = 1
  time_insertion_sorting = []
  time_merge_sorting = []
  time_inplaceQuick_sorting = []
  time_medianQuick_sorting = []
  time_heap_sorting = []
  #Running for 4 times to calculate the average for Different random inputs.
  while times_run <4:
    random_arr = []
    for a in range(0,arr[indexy]):
      print(arr[indexy])
      random_arr.append(random.randint(1,arr[indexy]+1))

    #calculating time for Insertion Sort
    then_time = time.time()
    sorting_insertion(random_arr[:])
    now_time = time.time()
    time_insertion_sorting.append((now_time-then_time)*1000)
    #calculating time for Merge Sort
    then_time = time.time()
    merge_sort(random_arr[:])
    now_time = time.time()
    time_merge_sorting.append((now_time-then_time)*1000)
    #calculating time for Heap Sort
    then_time = time.time()
    sorting_heap(random_arr[:])
    now_time = time.time()
    time_heap_sorting.append((now_time-then_time)*1000)
    #calculating time for In-Place Quick Sort
    then_time = time.time()
    sorting_quick(random_arr[:])
    now_time = time.time()
    time_inplaceQuick_sorting.append((now_time-then_time)*1000)
    #calculating time for Modified QuickSort
    then_time = time.time()
    mquick_sort(random_arr[:])
    now_time = time.time()
    time_medianQuick_sorting.append((now_time-then_time)*1000)
    times_run=times_run+1
```

```
        # Finding the mean for 4 runs and
        insertion_sorting_average.append(mean(time_insertion_sorting))
        merge_sorting_average.append(mean(time_merge_sorting))
        heap_sorting_average.append(mean(time_heap_sorting))
        inplaceQuick_sorting_average.append(mean(time_inplaceQuick_sorting))
        medianQuick_sorting_average.append(mean(time_medianQuick_sorting))

    print("ARRAY INPUT SIZES RANGES:\n",arr)
    print("INSERTION SORT TIME FOR VARIOUS INPUT SIZES:\n",insertion_sorting_average)
    print("MERGE SORT TIME FOR VARIOUS INPUT SIZES:\n",merge_sorting_average)
    print("HEAP SORT TIME FOR VARIOUS INPUT SIZES:\n",heap_sorting_average)
    print("IN-PLACE QUICK SORT TIME FOR VARIOUS INPUT
SIZES:\n",inplaceQuick_sorting_average)
    print("MODIFIED QUICK SORT TIME FOR VARIOUS INPUT
SIZES:\n",medianQuick_sorting_average)
```

Output Screenshot:



```
ARRAY INPUT SIZES RANGES:
 [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 20000, 30000, 40000, 50000, 60000]
INSERTION SORT TIME FOR VARIOUS INPUT SIZES:
 [35.008907318115234, 126.68418884277344, 385.9682083129883, 534.1482162475586, 1044.3637371063232, 1144.6032524108887, 1479.7279834747314, 1825.7968425750732, 2547.4987030029297, 2873.70228767395, 11183.933973312378, 26053.417921066284, 44362.06126213074, 82871.64688110352, 120581.30311965942]
MERGE SORT TIME FOR VARIOUS INPUT SIZES:
 [1.8589496612548828, 4.165172576904297, 6.0176849365234375, 8.341312408447266, 10.846853256225586, 13.094902038574219, 15.534162521362305, 17.862796783447266, 20.450115203857422, 22.943973541259766, 49.21889305114746, 76.31492614746094, 104.51722145080566, 134.0501308441162, 162.61911392211914]
HEAP SORT TIME FOR VARIOUS INPUT SIZES:
 [3.1762123107910156, 7.1659088134765625, 11.111259460449219, 15.471220016479492, 19.75703239440918, 24.312257766723633, 28.96285057067871, 33.663034439086914, 38.36679458618164, 43.29490661621094, 94.90108489990234, 148.63109588623047, 204.12707328796387, 264.1642093658447, 318.5079097747803]
IN-PLACE QUICK SORT TIME FOR VARIOUS INPUT SIZES:
 [1.5020370483398438, 3.204822540283203, 4.664897918701172, 6.514072418212891, 7.760047912597656, 9.350776672363281, 11.760234832763672, 13.874053955078125, 14.878988265991211, 15.704154968261719, 35.06207466125488, 54.076194763183594, 74.7978687286377, 95.72100639343262, 113.94906044006348]
MODIFIED QUICK SORT TIME FOR VARIOUS INPUT SIZES:
 [1.336812973022461, 3.885030746459961, 7.644176483154297, 12.113094329833984, 18.19586753845215, 25.563955307006836, 32.67312049865723, 42.81783103942871, 53.35187911987305, 64.9099349975586, 242.25091934204102, 532.5641632080078, 927.1659851074219, 1443.8910484313965, 2069.425106048584]
→  PY Latest
```

Reverse_sorted_array_driver.py

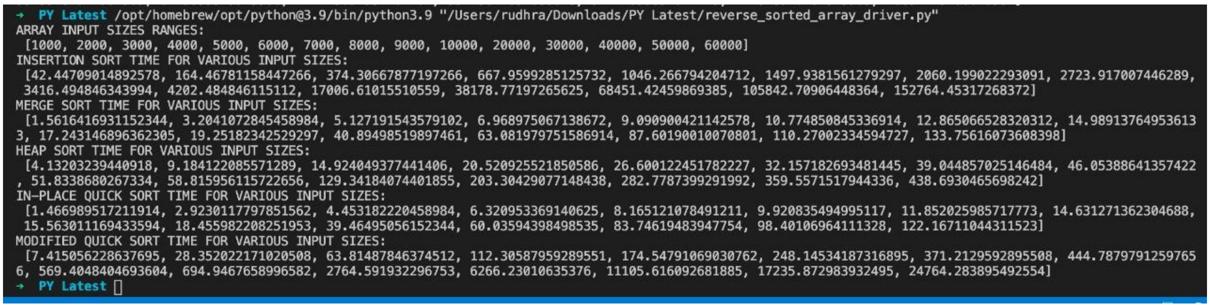
```
import random
import time
import sys
from statistics import mean
from heap_sort import sorting_heap
from insertion_sort import sorting_insertion
from merge_sort import merge_sort

def sortingquick(arr, smallest, highest):
    if smallest < highest:
        op = parting(arr, smallest, highest)
        sortingquick(arr, smallest, op - 1)
        sortingquick(arr, op + 1, highest)

def sorting_quick(numbers):
    arr = numbers
    sortingquick(arr, 0, len(arr) - 1)
    return arr

def parting(arr, smallest, highest):
```

```
    ax = (smallest - 1)
    pivot_point = arr[random.randint(smallest,highest)]
    for bz in range(smallest, highest):
        if arr[bz] <= pivot_point:
            ax = ax + 1
            arr[ax], arr[bz] = arr[bz], arr[ax]
    arr[ax + 1], arr[highest] = arr[highest], arr[ax + 1]
    return (ax + 1)
#Selecting the median element
middleC = 0
def middle1(a, b, c):
    if ( a - b) * (c - a) >= 0:
        return a
    elif (b - a) * (c - b) >= 0:
        return b
    else:
        return c

def quicksort_middle(arruence, smallest_index, highest_index):
    global middleC
    if smallest_index+ 10  <= highest_index:
        new_pivot_index = partition_median(arruence, smallest_index, highest_index)

        middleC += (highest_index - smallest_index - 1)
        quicksort_middle(arruence, smallest_index, new_pivot_index)
        quicksort_middle(arruence, new_pivot_index + 1, highest_index)

    else:
        insertion_sortting(arruence,smallest_index,highest_index)

def partition_median(arruence, smallest_val_arr, highest_val_arr):
    small_value= arruence[smallest_val_arr]
    high_value = arruence[highest_val_arr - 1]
    length = highest_val_arr - smallest_val_arr
    middle = arruence[smallest_val_arr + length // 2]
    pivot_point = middle1(small_value, high_value, middle)
    pivot_index = arruence.index(pivot_point)
    arruence[pivot_index] = arruence[smallest_val_arr]
    arruence[smallest_val_arr] = pivot_point
    ax = smallest_val_arr + 1
    for bz in range(smallest_val_arr + 1, highest_val_arr):
        if arruence[bz] < pivot_point:
            temp_var = arruence[bz]
            arruence[bz] = arruence[ax]
            arruence[ax] = temp_var
            ax += 1
```

```
    high_End_Val = arruence[smallest_val_arr]
    arruence[smallest_val_arr] = arruence[ax - 1]
    arruence[ax - 1] = high_End_Val
    return ax - 1

def mquick_sort(arr):
    quicksort_middle(arr, 0, len(arr))
    return arr

def insertion_sortting(arruence,a,b):
    for ax in range(a, b):
        bz = ax
        while bz > 0 and arruence[bz] < arruence[bz-1]:
            arruence[bz],arruence[bz-1]=arruence[bz-1],arruence[bz]
            bz = bz - 1

if _name_ == '_main_':
    #Setting the recursion Limit
    sys.setrecursionlimit(10**6)#For quick-sort to run properly for large data sets
    arr=[]#INPUT SIZE ARRAY
    for size in range(1000,10001,1000):
        arr.append(size)
    for size in range(20000,60001,10000):
        arr.append(size)
    insertion_sorting_average = []
    merge_sorting_average = []
    heap_sorting_average = []
    inplaceQuick_sorting_average = []
    medianQuick_sorting_average = []
    #finding the running time for each of the algorithm
    for index in range(0, len(arr)):
        runs = 1
        time_insertion_sorting = []
        time_merge_sorting = []
        time_inplaceQuick_sorting = []
        time_medianQuick_sorting = []
        time_heap_sorting = []
        #Reversed sorted array
        reversedArray = []
        for a in range(arr[index],0,-1):
            reversedArray.append(a)
        #calculating time for Insertion Sort
        then_time = time.time()
        sorting_insertion(reversedArray[:])
        now_time = time.time()
        time_insertion_sorting.append((now_time-then_time)*1000)
        #calculating time for Merge Sort
```

```
        then_time = time.time()
        merge_sort(reversedArray[:])
        now_time = time.time()
        time_merge_sorting.append((now_time-then_time)*1000)
         #calculating time for Heap Sort
        then_time = time.time()
        sorting_heap(reversedArray[:])
        now_time = time.time()
        time_heap_sorting.append((now_time-then_time)*1000)
        #calculating time for In-place Quick Sort
        then_time = time.time()
        sorting_quick(reversedArray[:])
        now_time = time.time()
        time_inplaceQuick_sorting.append((now_time-then_time)*1000)
        #calculating time for Modified Quick Sort
        then_time = time.time()
        mquick_sort(reversedArray[:])
        now_time = time.time()
        time_medianQuick_sorting.append((now_time-then_time)*1000)
        #Finding the mean for each of the running time complexity
        insertion_sorting_average.append(mean(time_insertion_sorting))
        merge_sorting_average.append(mean(time_merge_sorting))
        heap_sorting_average.append(mean(time_heap_sorting))
        inplaceQuick_sorting_average.append(mean(time_inplaceQuick_sorting))
        medianQuick_sorting_average.append(mean(time_medianQuick_sorting))

    #Printing Running time value for each of the input size
    print("ARRAY INPUT SIZES RANGES:\n",arr)
    print("INSERTION SORT TIME FOR VARIOUS INPUT SIZES:\n",insertion_sorting_average)
    print("MERGE SORT TIME FOR VARIOUS INPUT SIZES:\n",merge_sorting_average)
    print("HEAP SORT TIME FOR VARIOUS INPUT SIZES:\n",heap_sorting_average)
    print("IN-PLACE QUICK SORT TIME FOR VARIOUS INPUT
SIZES:\n",inplaceQuick_sorting_average)
    print("MODIFIED QUICK SORT TIME FOR VARIOUS INPUT
SIZES:\n",medianQuick_sorting_average)
```

Output Screenshot:



```
→  PY Latest /opt/homebrew/opt/python@3.9/bin/python3.9 "/Users/rudhra/Downloads/PY Latest/reverse_sorted_array_driver.py"
ARRAY INPUT SIZES RANGES:
 [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 20000, 30000, 40000, 50000, 60000]
INSERTION SORT TIME FOR VARIOUS INPUT SIZES:
 [42.44709014892578, 164.46781158447266, 374.30667877197266, 667.9599285125732, 1046.266794204712, 1497.9381561279297, 2060.199022293091, 2723.917007446289, 3416.494846343994, 4202.484846115112, 17006.61015510559, 38178.77197265625, 68451.42459869385, 105842.70906448364, 152764.45317268372]
MERGE SORT TIME FOR VARIOUS INPUT SIZES:
 [1.5616416931152344, 3.2041072845458984, 5.127191543579102, 6.968975067138672, 9.090900421142578, 10.774850845336914, 12.865066528320312, 14.989137649536133, 17.243146896362305, 19.25182342529297, 40.89498519897461, 63.081979751586914, 87.60190010070801, 110.27002334594727, 133.75616073608398]
HEAP SORT TIME FOR VARIOUS INPUT SIZES:
 [4.13203239440918, 9.184122085571289, 14.924049377441406, 20.520925521850586, 26.600122451782227, 32.157182693481445, 39.044857025146484, 46.05388641357422, 51.8338680267334, 58.815956115722656, 129.34184074401855, 203.30429077148438, 282.7787399291992, 359.5571517944336, 438.6930465698242]
IN-PLACE QUICK SORT TIME FOR VARIOUS INPUT SIZES:
 [1.466989517211914, 2.9230117797851562, 4.453182220458984, 6.320953369140625, 8.165121078491211, 9.920835494995117, 11.852025985717773, 14.631271362304688, 15.563011169433594, 18.455982208251953, 39.46495056152344, 60.03594398498535, 83.74619483947754, 98.40106964111328, 122.16711044311523]
MODIFIED QUICK SORT TIME FOR VARIOUS INPUT SIZES:
 [7.415056228637695, 28.352022171020508, 63.81487846374512, 112.30587959289551, 174.54791069030762, 248.14534187316895, 371.2129592895508, 444.7879791259766, 569.4048404693604, 694.9467658996582, 2764.591932296753, 6266.23010635376, 11105.616092681885, 17235.872983932495, 24764.283895492554]
→  PY Latest
```

Sorted_Array_driver.py

```
import random
```

```
import time
import sys
#import matplotlib.pyplot as plot
from statistics import mean
from statistics import mean
from heap_sort import sorting_heap
from insertion_sort import sorting_insertion
from merge_sort import merge_sort
from quick_sort import sorting_quick,mquick_sort

if _name_ == '_main_':
    sys.setrecursionlimit(10**6)#For quick-sort to run properly for large data sets
    arr=[]#INPUT SIZE ARRAY
    for size in range(1000,10001,1000):
        arr.append(size)
    for size in range(20000,60001,10000):
        arr.append(size)
    insertion_sorting_average = []
    merge_sorting_average = []
    heap_sorting_average = []
    inplaceQuick_sorting_average = []
    medianQuick_sorting_average = []
    #finding the running-time for the each of the sorting algorithm
    for index in range(0, len(arr)):
        runs = 1
        time_insertion_sorting = []
        time_merge_sorting = []
        time_inplaceQuick_sorting = []
        time_medianQuick_sorting = []
        time_heap_sorting = []
        sortedArray = []
        for a in range(0,arr[index]):
            sortedArray.append(a)
        #calculating time for Insertion Sort
        then_time = time.time()
        sorting_insertion(sortedArray[:])
        now_time = time.time()
        time_insertion_sorting.append((now_time-then_time)*1000)
        #calculating time for Merge Sort
        then_time = time.time()
        merge_sort(sortedArray[:])
        now_time = time.time()
        time_merge_sorting.append((now_time-then_time)*1000)
        #calculating time for Heap Sort
        then_time = time.time()
        sorting_heap(sortedArray[:])
        now_time = time.time()
```

```
    time_heap_sorting.append((now_time-then_time)*1000)
    #calculating time for In-Place Quick Sort
    then_time = time.time()
    sorting_quick(sortedArray[:])
    now_time = time.time()
    time_inplaceQuick_sorting.append((now_time-then_time)*1000)
    #calculating time for Modified QuickSort
    then_time = time.time()
    mquick_sort(sortedArray[:])
    now_time = time.time()
    time_medianQuick_sorting.append((now_time-then_time)*1000)
    #Finding the average value for each of the algorithm
    insertion_sorting_average.append(mean(time_insertion_sorting))
    merge_sorting_average.append(mean(time_merge_sorting))
    heap_sorting_average.append(mean(time_heap_sorting))
    inplaceQuick_sorting_average.append(mean(time_inplaceQuick_sorting))
    medianQuick_sorting_average.append(mean(time_medianQuick_sorting))
  #Printing the values
  print("ARRAY INPUT SIZES RANGES:\n",arr)
  print("INSERTION SORT TIME FOR VARIOUS INPUT SIZES:\n",insertion_sorting_average)
  print("MERGE SORT TIME FOR VARIOUS INPUT SIZES:\n",merge_sorting_average)
  print("HEAP SORT TIME FOR VARIOUS INPUT SIZES:\n",heap_sorting_average)
  print("IN-PLACE QUICK SORT TIME FOR VARIOUS INPUT
SIZES:\n",inplaceQuick_sorting_average)
  print("MODIFIED QUICK SORT TIME FOR VARIOUS INPUT
SIZES:\n",medianQuick_sorting_average)
```

Output Screenshot:

```
→  PY Latest /opt/homebrew/opt/python@3.9/bin/python3.9 "/Users/rudhra/Downloads/PY Latest/sorted_array_driver.py"
ARRAY INPUT SIZES RANGES:
 [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 20000, 30000, 40000, 50000, 60000]
INSERTION SORT TIME FOR VARIOUS INPUT SIZES:
 [0.08225440979003906, 0.15306472778320312, 0.2334117889404297, 0.2880096435546875, 0.37407875061035156, 0.4241466522216797, 0.49805641174316406, 0.5710124969482422, 0.6499290466308594, 0.72479248046875, 1.4429092407226562, 2.1543502807617188, 2.8738975524902344, 3.6547183990478516, 4.317045211791992]
MERGE SORT TIME FOR VARIOUS INPUT SIZES:
 [1.7881393432617188, 3.381013870239258, 5.0907135009765625, 6.783962249755859, 8.800029754638672, 10.637760162353516, 12.588024139404297, 14.5721435546875, 16.73293113708496, 18.514156341552734, 39.30234909057617, 61.202287673950195, 83.47272872924805, 106.2169075012207, 128.8161277770996]
HEAP SORT TIME FOR VARIOUS INPUT SIZES:
 [3.0410289764404297, 6.492853164672852, 10.06007194519043, 13.957977294921875, 17.9440975189209, 22.12977409362793, 26.67999267578125, 31.071901321411133, 35.215139389038086, 39.57724571228027, 87.89682388305664, 137.44211196899414, 188.80987167358398, 240.65685272216797, 296.92697525024414]
IN-PLACE QUICK SORT TIME FOR VARIOUS INPUT SIZES:
 [1.5492439270019531, 2.9020309448242188, 4.332065582275391, 6.165027618408203, 7.790088653564453, 8.688211441040039, 10.769128799438477, 13.115882873535156, 15.718936920166016, 16.229867935180664, 34.03878211975098, 51.27596855163574, 71.82908058166504, 90.78621864318848, 104.76493835449219]
MODIFIED QUICK SORT TIME FOR VARIOUS INPUT SIZES:
 [1.0466575622558594, 3.008127212524414, 6.8149566650390625, 9.001970291137695, 11.157989501953125, 23.34904670715332, 27.050018310546875, 31.104087829589844, 34.74998474121094, 37.9488468170166, 137.01796531677246, 388.72694969177246, 526.5510082244873, 1266.9858932495117, 1517.2078609466553]
→  PY Latest
```

Random Input

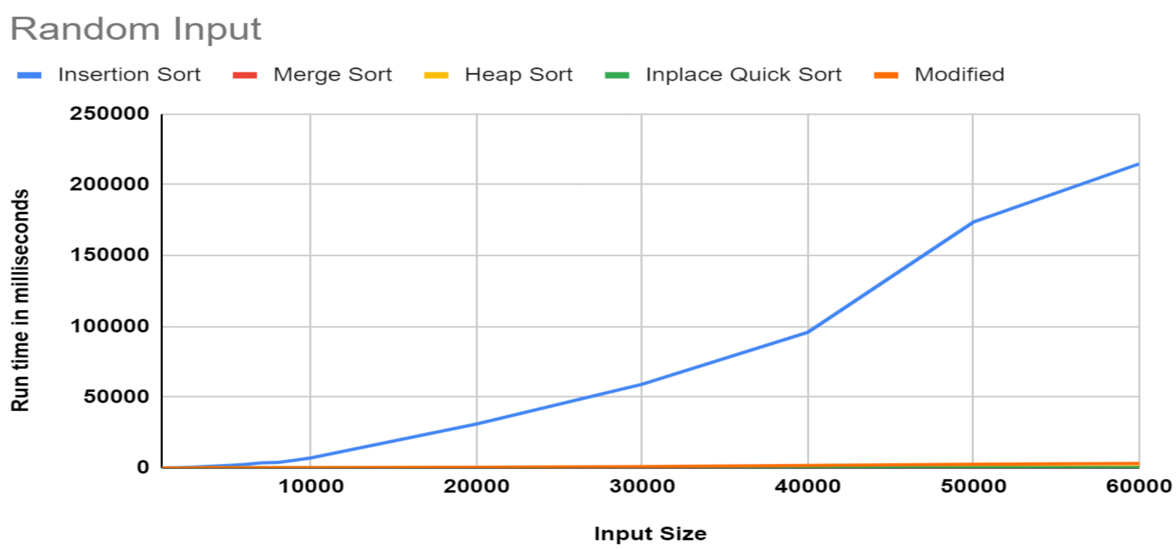
Input Size	Insertion Sort	Merge Sort	Heap Sort	Inplace Quick Sort	Modified Quick Sort
1000	72.4850495	3.32458814	7.29878743	2.6336511	3.68579229
2000	291.162967	7.63082504	13.6259396	5.6505998	6.9940884
3000	654.160022	11.2654368	21.0785071	8.63981247	13.31615448

4000	1163.33603	15.9436066	29.4656753	11.9694074	21.3023026
5000	1746.08087	20.7934379	37.9161834	14.2906506	30.9038957
6000	2406.65459	25.6282488	47.5424130	17.6316102	42.21272469
7000	3619.93447	30.9332211	55.2153587	20.6374327	56.90868696
8000	3824.26627	34.8876317	63.870649	25.7072448	72.13600477
9000	5373.24484	40.3192838	73.4365781	28.0006726	91.25630061
10000	7058.95511	44.9960231	81.8928082	30.6281248	111.8530432
20000	31019.5341	99.0788141	179.341634	83.1139882	499.4450410
30000	59067.5102	148.912668	278.076330	99.7260411	966.7311509
40000	95783.5803	211.422363	380.726575	141.848882	1742.850303
50000	173715.996	260.289271	503.654162	180.529197	2639.540592
6000	21475.896	290.388381	730.5654824	210.9325486	3124.641487

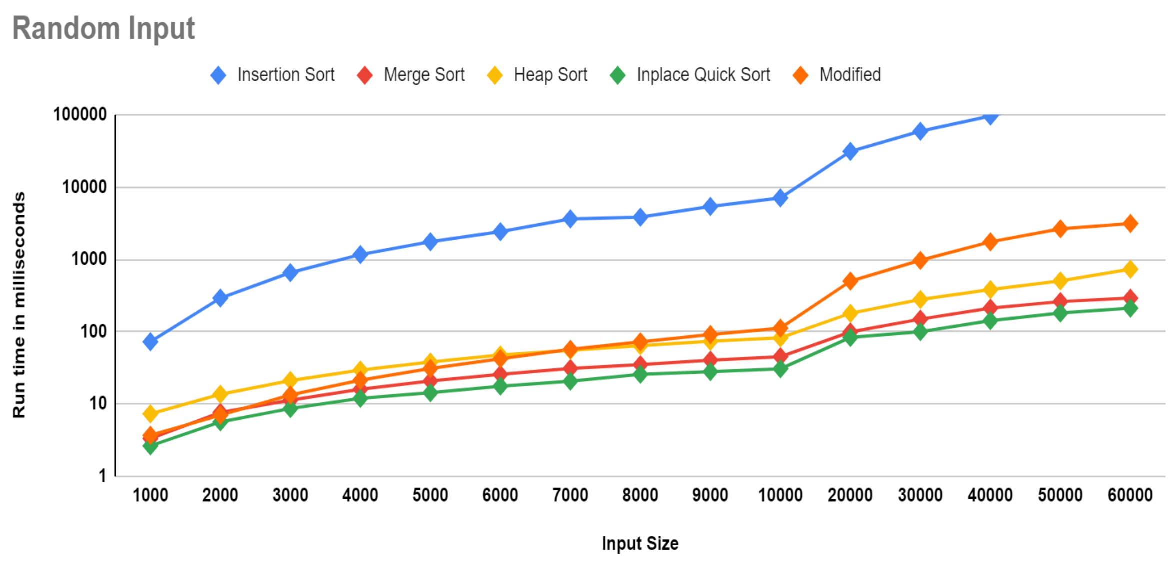
Line-chart graph Analysis of above mentioned algorithm with input size ranging from 1000 to 60000 for Random Input Array:

The graph has been displayed with y-Axis in logarithmic scale and in Arithmetic Scale for better understanding.

Y-AXIS is in Arithmetic Scale:



Y-AXIS is in Logarithmic Scale:

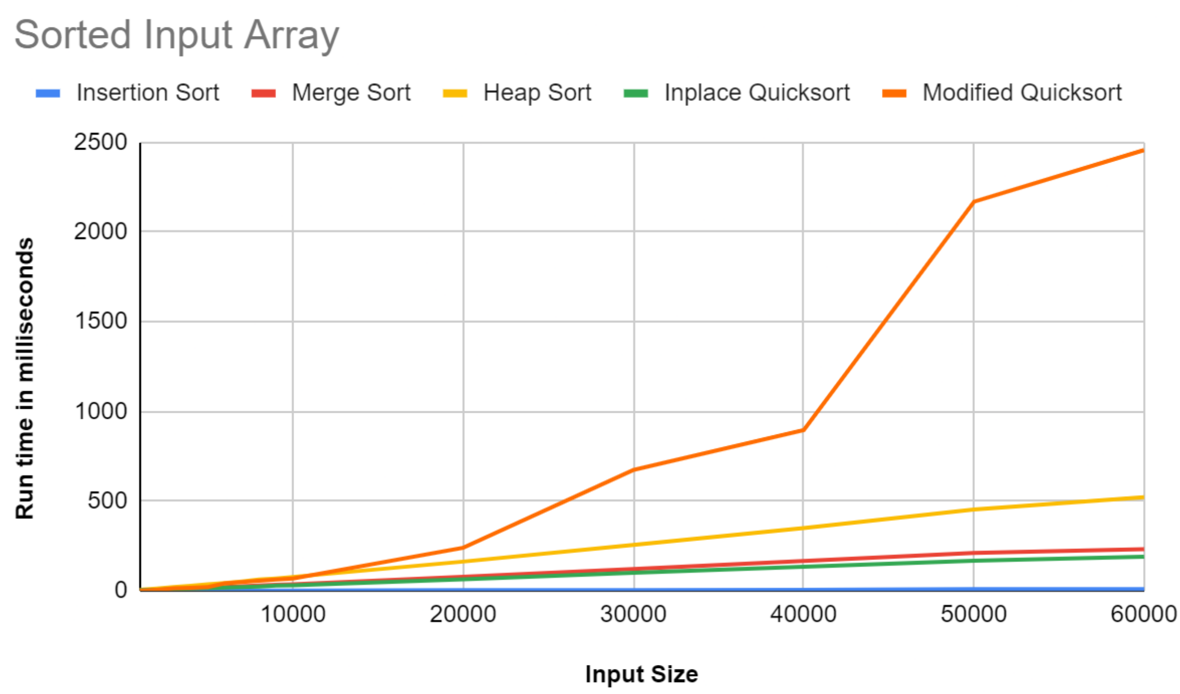


Sorted Input

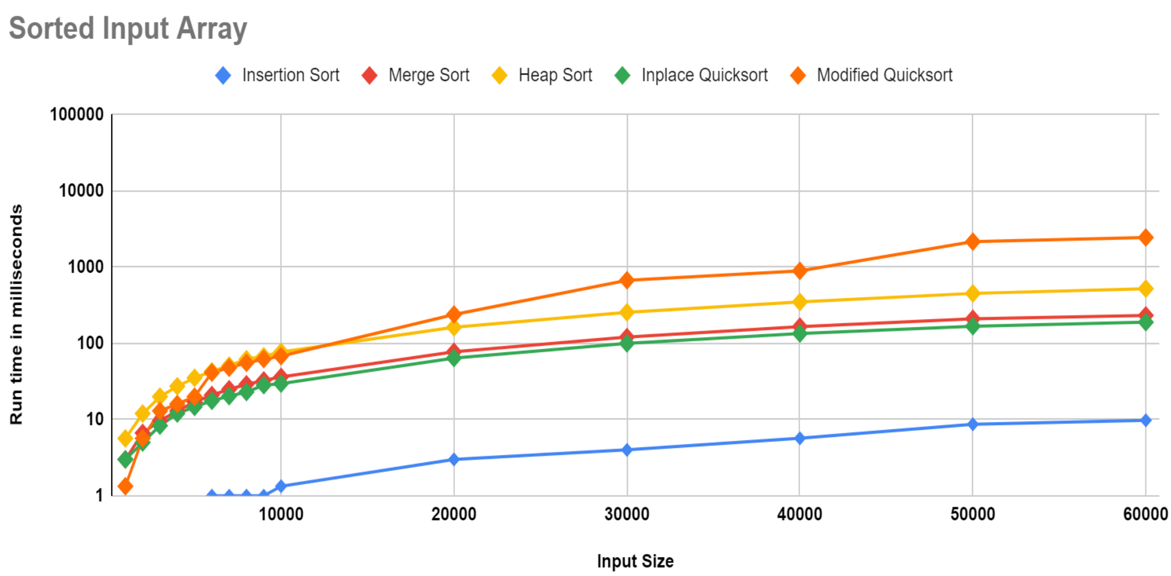
Input Size	Insertion Sort	Merge Sort	Heap Sort	Inplace Quicksort	Modified Quicksort
1000	0.0	2.99159686	5.651474	2.99191475	1.32981936
2000	0.0	6.64917628	11.96829478	4.98636564	5.65155347
3000	0.3323555	9.64975357	19.94633675	8.30189387	12.96544075
4000	0.99786123	12.97903061	27.26022402	11.96710269	15.95854759
5000	0.66685677	16.95084572	35.23755074	14.62745667	19.93393898
6000	0.99770228	20.94348272	42.89865494	17.60570208	41.22384389
7000	0.99738439	24.93341764	50.86334546	20.27885119	47.87198702
8000	0.99730492	28.93662453	61.82074547	22.94023832	55.5164814
9000	0.99778175	32.5782299	67.48231252	28.25840314	62.17503548
10000	1.32989883	36.22213999	77.15495427	29.58718936	67.81880061
20000	2.9908816	77.445666	162.5532309	64.17560577	239.7038141
30000	3.98842494	120.6764380	255.6505203	99.73033269	673.5335985

40000	5.66569964	165.8760706	349.7501214	134.3181133	895.2696323
50000	8.64545504	210.3641033	453.1114101	167.8847471	2168.755372
60000	13.76543675	232.4658742	522.2546229	199.6952346	2546.895621

Line-chart graph Analysis of above mentioned algorithm with input size ranging from 1000 to 60000 for Sorted Input Array:
Y-AXIS is in Arithmetic Scale:



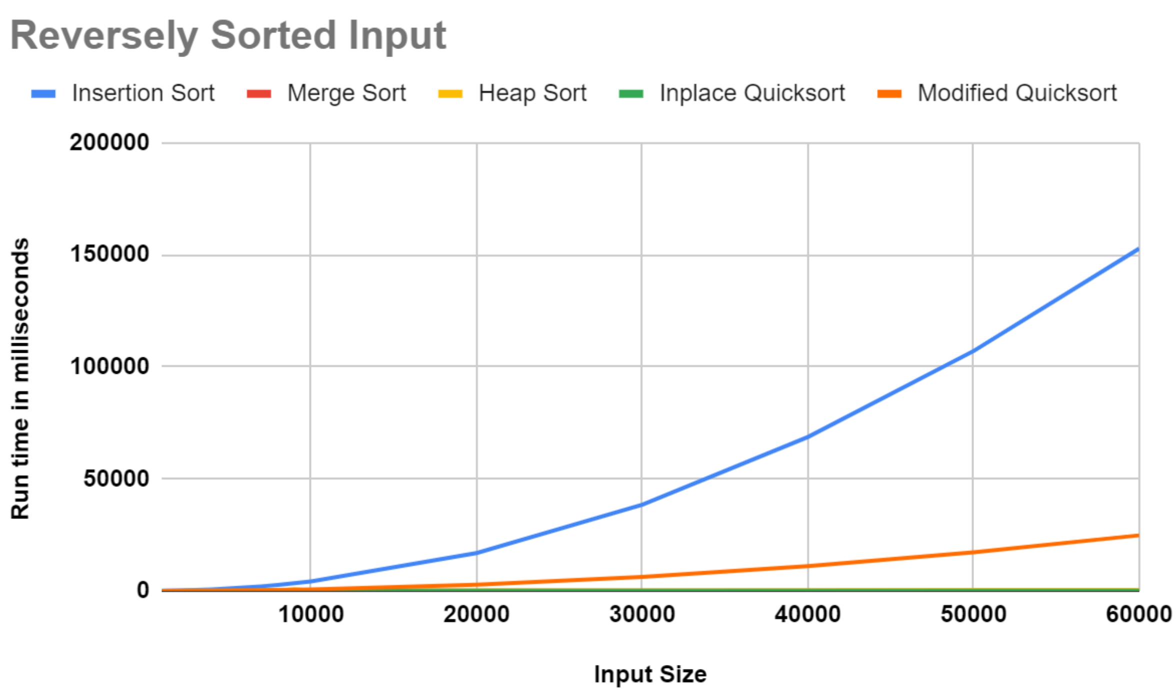
Y-AXIS is in Logarithmic Class:



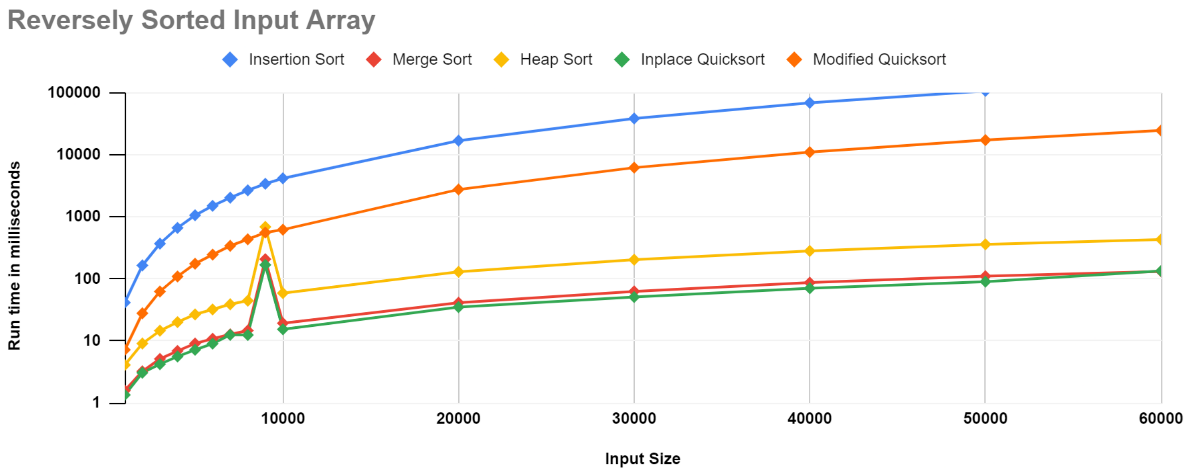
Reversely Sorted Input:

Input Size	Insertion Sort	Merge Sort	Heap Sort	Inplace Quicksort	Modified Quicksort
1000	41.48817062	1.582145	4.08792496	1.3449192	7.17306137
2000	164.7548676	3.2279491	9.05704498	3.06391716	27.83703804
3000	369.5421219	5.1140785	14.58573341	4.20880318	62.51311302
4000	662.3780727	6.8819522	20.05887032	5.59806824	109.0550422
5000	1059.494019	9.07588	26.65424347	7.18021393	175.1928329
6000	1501.678705	10.818004	32.08208084	9.01913643	245.8040714
7000	2029.650211	12.730836	38.90395164	12.48693466	340.4233455
8000	2669.193029	14.646768	44.38877106	12.41803169	433.5269928
9000	3401.17979	206.98285	686.8999004	167.9310798	557.6421785
10000	4167.819023	19.249916	58.88319016	15.36297798	619.3816185
20000	16887.09998	41.241168	129.7819614	35.0048542	2754.239082
30000	38434.86595	62.711	203.3371925	50.87685585	6193.816185
40000	68709.00822	86.865901	281.1479568	70.42002678	11012.542
50000	107000.9992	110.22305	359.9188327	89.92600441	17244.12107
6000	152764.453172	133.756160736	438.693046569	122.167110443	24764.283895

Line-chart of above mentioned algorithm with input size ranging from 1000 to 60000 for Reversely sorted Input Array:
Y-AXIS is in Arithmetic scale:



Y-AXIS is in Logarithmic Scale:



Conclusion:

Randomly generated Input Array:

- It can be inferred from the above graph ,that the insertion sort takes more time for sorting the randomly generated elements present in the input Array.
- Modified Quicksort takes next more time for sorting the randomly generated elements present in the input Array.
- In-place Quicksort takes less time than other sorting algorithms for sorting a randomly generated input Array.

Sorted Input Array:

- It can be inferred from the above graph,modified quicksort takes more time for sorting an array of sorted elements.
- Insertion sort takes less time when compared with other sorting algorithms for sorting an array of sorted elements.

Reversely Sorted Input Array:

- It can be inferred from the above graph,insertion sort takes more time for sorting an array of reversely sorted elements.
- Inplace Quicksort takes less time for sorting an array of reversely sorted elements.

➔ Heap Sort is the slowest O(N log N) of the sorting algorithms but unlike merge and quicksort, it does not require massive recursion or multiple arrays to work.

Reference:

➔ https://github.com.
➔ https://youtube.com.
➔ https://iq.opengenus.org/insertion-sort-analysis/.
➔ https://geeksforgeeks.com.
➔ Lectures slide from canvas.
➔ https://stackoverflow.com