PROJECT 1

COMPARISON – BASED SORTING ALGORITHMS

ALGORITHMS AND DATA STRUCTURES ITCS 6114

Submitted By:

Sai Prasad Palli (801254453)

Shourya Reddy Katkam (801255429)

Project Objective:

Implement the following sorting algorithms:

- Insertion sort
- Merge sort
- Heapsort [vector based, and insert one item at a time]
- 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 <= 8, you must use insertion sort.

Execution instructions:

- Run these algorithms for different input sizes (e.g. = 1000, 2000, 4K, 5K, 10K, 20K, 40K, 50K, 60K, 80K, 100K). You will randomly generate input for execution. Record the execution time (need to take the average of several runs) and plot them all in a single graph against various input sizes. Note that, you will compare all sorting algorithms for the same data set.
- Also observe and present performance of the following two special cases:
    - Input array is already sorted.
    - Input array is reversely sorted.

DATA STRUCTURES

Data structures used in this project are

- Arrays
- Heap sort is executed with heap data structure implemented using vector-based array

LANGUAGE:

The language we used: Python 3

LIBRARIES:

The libraries we used are: Random, Sys, Matplotlib, Time

COMPLEXITY ANALYSIS:

INSERTION SORT

ALGORITHM

1. Iterate from index 1 to n over the array arr.

2. Compare the current element (key) to its predecessor.

3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

PSEUDO CODE

```
# Function to do insertion sort
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of array greater than key, to one position ahead of current position
        j = i-1
            while j >=0 and key < arr[j] : arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
```

Time Complexity: O (n^2)
Space Complexity: O (1)

Worst-case happens when the input is in reverse order, As the element will have to be relocated to the start of the array every time. Therefore, the time complexity with the worst case is O(n^2).

Best-case happens whenever the input is in sorted order. In such a case, there would be no swaps, so the whole array is iterated just once. Thus, the time complexity for best case is O (n).

Average-case happens if half of the elements are less than one element and then the other half is greater. Time complexity is similar to the worst-case O(n^2).

MERGE SORT

ALGORITHM

MergeSort(a[], l1, l2)
If l2 > l1

1. Find the middle point to divide the array into two halves:
middle m = l1+ (l2-l)/2

2. Call mergeSort for first half:
Call mergeSort(a, l1, m)

3. Call mergeSort for second half:
Call mergeSort(a, m+1, l2)

4. Merge the two halves sorted in step 2 and 3:
Call merge(a, l1, m, l2)

PSEUDO CODE

```
procedure mergesort( var a as array )
  if ( n == 1 ) return a

  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]

  l1 = mergesort( l1 )
  l2 = mergesort( l2 )

  return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while

  while ( a has elements )
```

```
    add a[0] to the end of c
    remove a[0] from a
  end while

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while

  return c

end procedure
```

Time complexity: O(nlogn)
Space complexity: O(n)

In merge sort, we use Divide and Conquer approach, where the input array will be split in half. The time to do this operation is 'log n, when the size of the input is n. After splitting, merge would be linear time i.e. 'n' since there will be n operations. As a result, Merge sort logs n splits in which n is the expense of merging it which is O(nlogn) for the whole sorting.

HEAP SORT

ALGORITHM

1. We start by using **Heapify** to build a max heap of elements present in an array **A**.
2. Once the heap is ready, the largest element will be present in the root node of the heap that is **A[1]**.
3. Now swap the element at **A[1]** with the last element of the array, and heapify the max heap excluding the last element.
4. Repeat steps 2 and 3 till all the elements in the array are sorted.

PSEUDO CODE

```
Heapify(A as array, n as int, i as int)
{
   max = i
   leftchild = 2i + 1
   rightchild = 2i + 2
   if (leftchild <= n) and (A[i] < A[leftchild])
      max = leftchild
   else
      max = i
   if (rightchild <= n) and (A[max]  > A[rightchild])
```

```
        max = rightchild
    if (max != i)
        swap(A[i], A[max])
        Heapify(A, n, max)
}
Heapsort(A as array)
{
  n = length(A)
  for i = n/2 downto 1
    Heapify(A, n ,i)

  for i = n downto 2
    exchange A[1] with A[i]
    A.heapsize = A.heapsize - 1
    Heapify(A, i, 0)
}
```

Time complexity: O(nlogn)
Auxiliary Space: O(1)

There are two steps in heap sorting – building a heap and removing a min.

- In order to build a heap, 'n' elements are inserted into the heap array after each insertion, the array is heaped to preserve the heap order property. So, it takes time for 'n' elements to heap.
- Then, the complexity of the time will be O (nlogn).

- In order to eliminate min, the first element was removed and heaped. Thus, the total 'n' elements would be removed in the sorted order and heaped after each removal order to preserve the heap property.

- So, it's time for O (nlogn).

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

IN PLACE AND MODIFIED QUICKSORT

ALGORITHMS

1. In Place: A pivot element for the in place quick sort is selected randomly.
Modified: The median of three is used to select the pivot for the modified quick sort. The modified quicksort is not recommended for an array of size less than 8. In that case, insertion sort is used.
2. The array is partitioned around the pivot element. Then the elements that are less than the pivot element are moved to the left and the elements greater than or equal to pivot are moved to the right side.
3. The pivot element will currently in the correct position.
4. The quicksort should be applied recursively on the left and right-side partition.
5. The recursion is stopped when the base case of an array of size one or zero is reached, when we will have a sorted array.

PESUDO CODE:

```
Algorithm InPlaceQS(K, n, r) if n >= r
        return

 i <- a random integer between n and r
 y <- S.elemAtRank(i)
 (h, j) <- inPlacePartition(y)

 InPlaceQS (K, n, h - 1) InPlaceQS (K, j + 1, r)
      Modified_QS (P, q, r) while(q < r)
      /* Partition and sort left subarray */ do
       s = PARTITION(S, q, r);

       Modified_QS (S, n, x-1); n = x+1;
```

In place quicksort

Worst case for In-place fast sort arises whenever the pivot could be the smallest or the largest element in the array. Whenever the list is sorted or retrieved, in-place occurs in the worst-case time. In the worst case, the time complexity is O(n^2).

Best case for In-place fast sort happens whenever the middle element is rotated as well as the time complexity is O. (nlogn).

Average-Case for In-place quicksort is O(nlogn).

Modified quicksort (Median-Of-Three)

- Use median-of-three as pivot.
- For a small sub-problem of size ≤ 8, you must use insertion sort.

In modified quick sort, we use median-of-three to select pivot, and even when the subarray is size 15, the insertion sort is being used to sort since quicksort may not work very well for small inputs.

Worst-case happens when the pivot is either the smallest or the largest element in the array. The time complexity is O(n^2).

Best-case happens when the pivot chosen divides the array into equal halves. Time complexity is O(nlogn).

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).

OUTPUTS AND GRAPHS

Sorting Algorithms time analysis:

Execution of each sort algorithm and comparison visualized in below tables:

Randomly Generated Array

Sorting Algorithm	1000	2000	4K	5K	10K	20K	40K	50K	60K	80K	100K
Insertion Sort	0.035	0.111	0.619	0.880	3.312	21.70	90.96	125.8	179.0	206.5	335.7
Merge Sort	0.003	0.007	0.015	0.022	0.066	0.141	0.211	0.387	0.261	0.371	0.468
Heap Sort	0.003	0.009	0.018	0.031	0.057	0.116	0.259	0.521	0.337	0.470	0.591
Quick Sort	0.037	0.06	0.164	0.188	0.397	0.891	1.669	3.601	2.427	3.505	4.415
Modified Quick Sort	0.011	0.037	0.078	0.097	0.196	0.536	0.964	1.878	1.282	1.814	2.388

Randomly Generated Array Sorted

Sorting Algorithm	1000	2000	4K	5K	10K	20K	40K	50K	60K	80K	100K
Insertion Sort	0.0001	0.0003	0.0006	0.0007	0.001	0.004	0.013	0.017	0.017	0.011	0.014
Merge Sort	0.002	0.005	0.014	0.031	0.053	0.090	0.171	0.370	0.270	0.337	0.447
Heap Sort	0.003	0.009	0.018	0.027	0.063	0.113	0.240	0.643	0.346	0.477	0.654
Quick Sort	0.045	0.068	0.153	0.171	0.409	0.774	1.667	3.17	2.401	3.506	4.379
Modified Quick Sort	0.010	0.040	0.077	0.102	0.185	0.586	1.120	2.160	1.334	1.768	2.406

Randomly Generated Array Reverse Sorted

Sorting Algorithm	1000	2000	4K	5K	10K	20K	40K	50K	60K	80K	100K
Insertion Sort	0.084	0.254	1.106	1.615	6.830	39.50	149.0	250.2	286.7	422.8	675.0
Merge Sort	0.002	0.006	0.015	0.022	0.044	0.083	0.195	0.474	0.302	0.372	0.476
Heap Sort	0.003	0.016	0.019	0.026	0.056	0.126	0.252	0.525	0.327	0.522	0.651
Quick Sort	0.029	0.081	0.158	0.176	0.382	1.062	1.738	3.638	2.411	3.348	4.357
Modified Quick Sort	0.014	0.045	0.148	0.094	0.182	0.543	1.429	2.082	1.279	1.898	2.402

Graphs - visualization of three types of different sort algorithm comparison:

- The program generates an irregular rundown and calculates runtime few times before storing it in a different cluster.
- The array is used as a runtime for performing a specific calculation for a given n value.
- After gathering all the runtimes for various arranging strategies and n esteems, the program displays the charts for each arranging procedure that includes an exceptional situation.
- Colors are used to differentiate different sorting algorithms as shown below

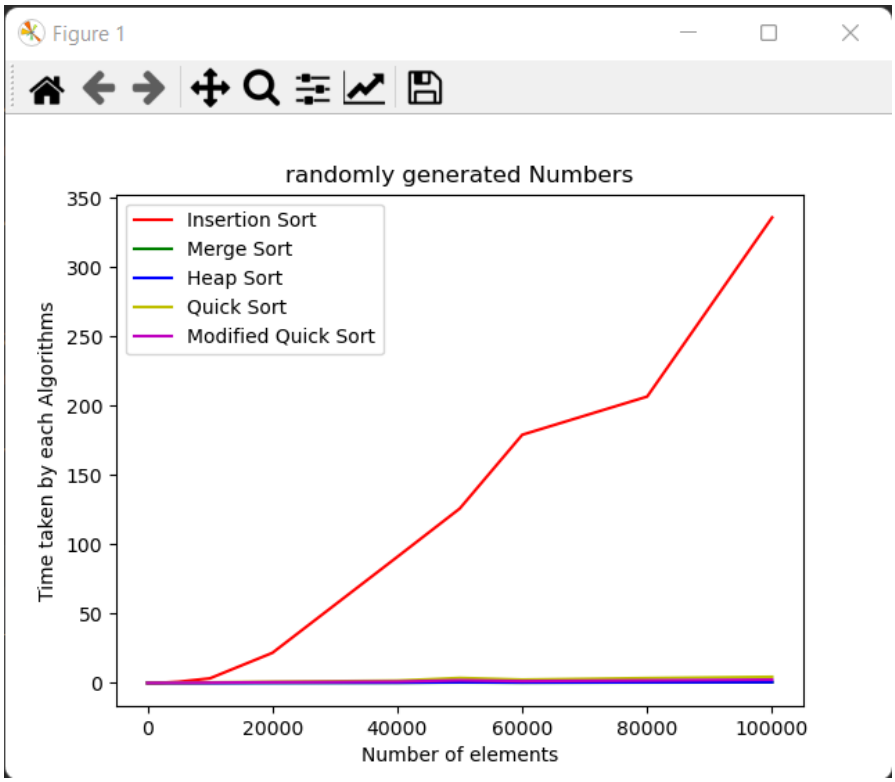
Graphs will represent the following:

The graphs are represented as the x axis is for input sizes and the y axis is for the time taken for each sort algorithm in seconds.

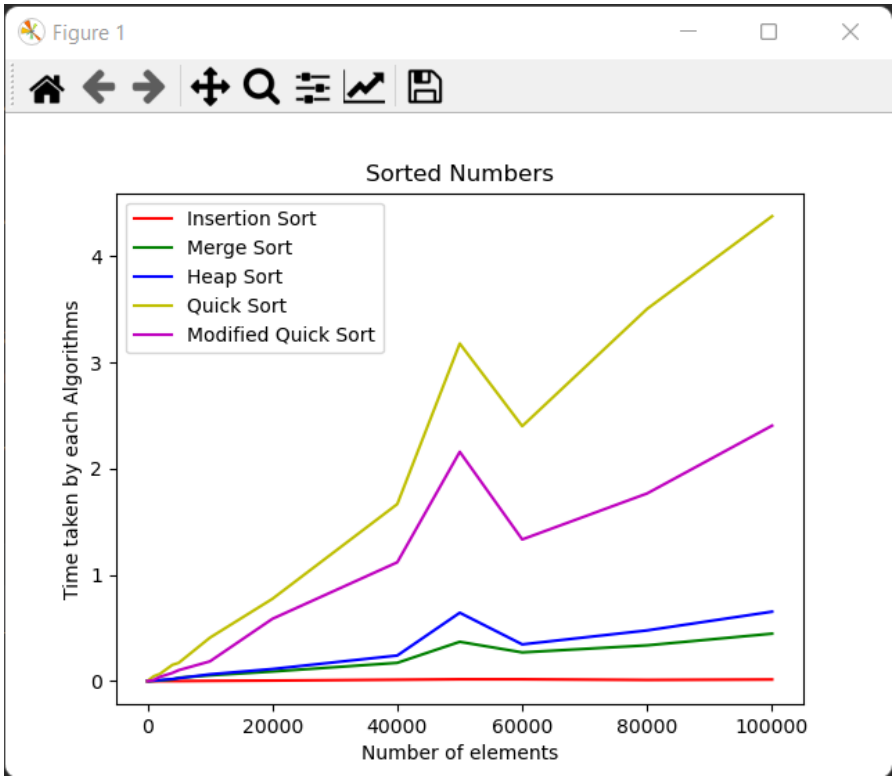
We have run the program twice with different sizes, the first run is for input sizes of

[0,1000,2000,4000,5000,10000,20000,40000,50000,60000,80000,100000]

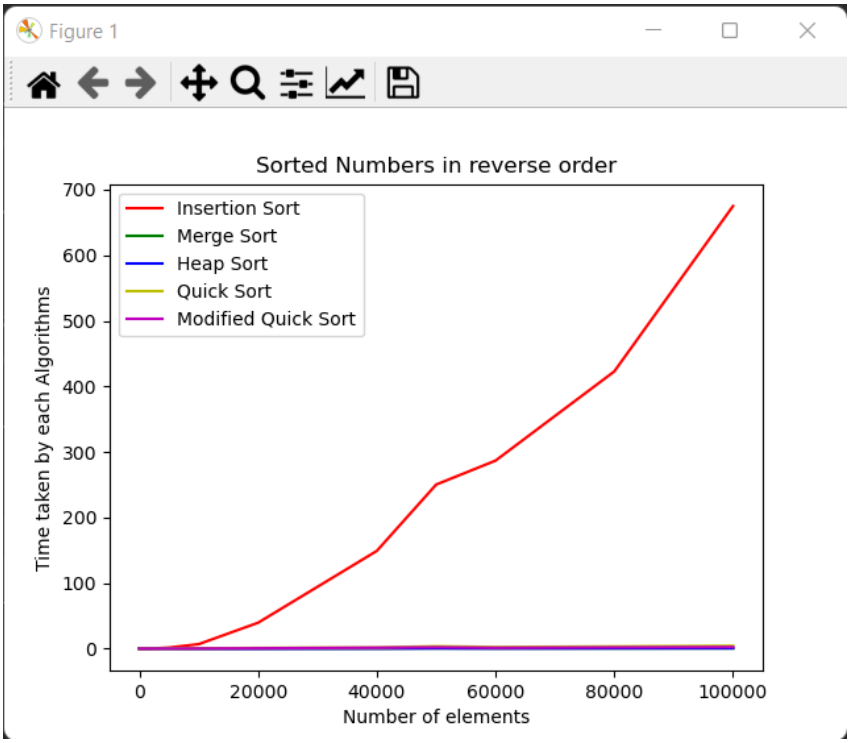
- Randomly generated Numbers



- Randomly generated Numbers Sorted



- Sorted Numbers in reverse order



Here in the randomly generated input for each size, the insertion sort is taking up more time compared to other sorting algorithms.

In the analysis of the second graph which is the sorted randomly generated input. We can see that the insertion sort takes less time compared to other sorting algorithms and the quick sort has the most time.

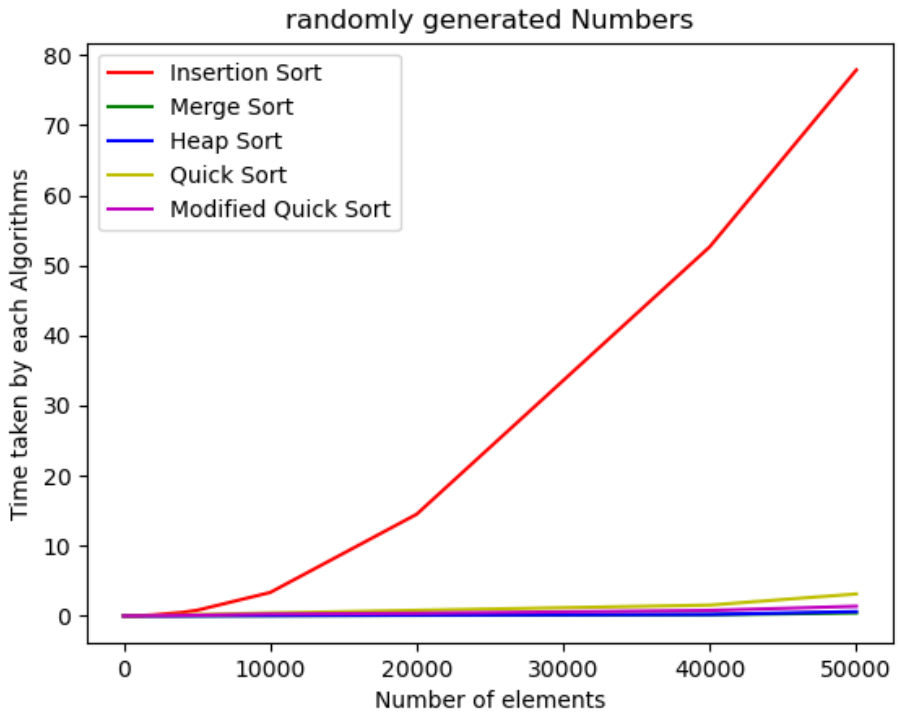
In the analysis of the third graph which is the reverse sorted randomly generated input. We can see that the insertion sort has the highest time compared to other algorithms.

The Second run is for input sizes of

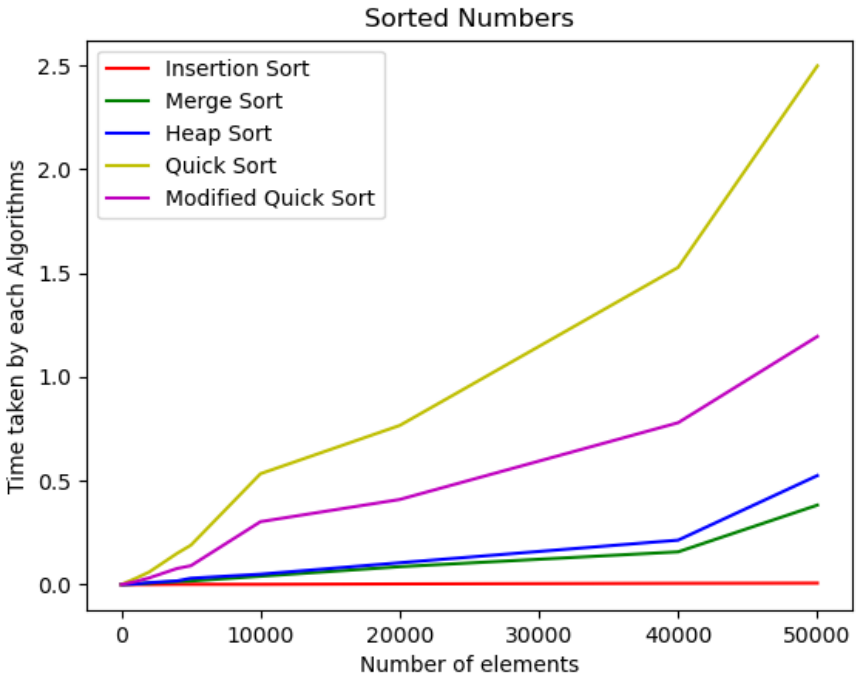
[0,1000,2000,4000,5000,10000,20000,40000,50000]

This run is for more understanding of each algorithm analysis. As we go for higher number, because of the insertion sort time consumption other sort algorithm lines are almost negligible in the above run analysis.

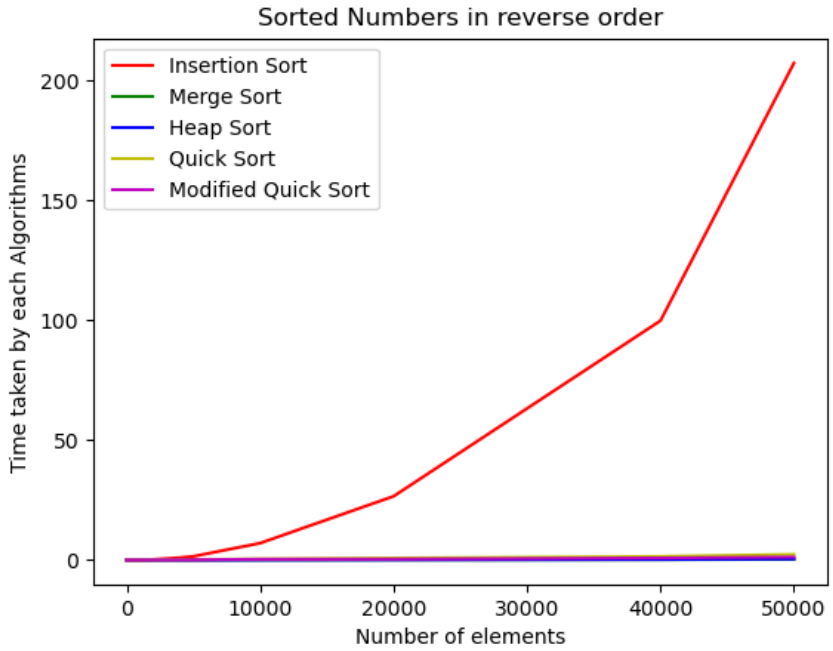
- Randomly generated Numbers



- Randomly generated Numbers Sorted



- Sorted Numbers in reverse order



Since we had taken less input size, we can visualize the analysis of the graphs more clearly.

Here in the randomly generated input for each size, the insertion sort is taking up more time compared to other sorting algorithms.

In the analysis of the second graph which is the sorted randomly generated input. We can see that the insertion sort takes less time compared to other sorting algorithms and the quick sort has the most time followed by each sort algorithm which are, modified quicksort, heap sort, merge sort and the last insertion sort.

Since the array is already sorted, we know that for sorted array insertion sort takes O(n) time complexity. So, it has the least time in sorted array analysis.

In the analysis of the third graph which is the reverse sorted randomly generated input. We can see that the insertion sort has the highest time compared to other algorithms. This is because it needs to go to the end of the array and make all the moves which takes $O(n^2)$ time complexity.

Project Code:

```
""" Authors:

    Sai Prasad Palli - 801254453

    Shourya Reddy Katkam - 801255429 """


# Comparison-based Sorting Algorithm

import random

import sys

sys.setrecursionlimit(10**6)

from time import time

import matplotlib.pyplot as myPlot


# Implementing INSERTION SORT

def insertion_sort(list_of_elements):
```

```
 for x in range(1, len(list_of_elements)):
    key = list_of_elements[x]
    y = x-1
    while y >= 0 and list_of_elements[y] > key:
        list_of_elements[y+1] = list_of_elements[y]
        y = y - 1
    list_of_elements[y+1] = key


#Implementing MERGE SORT
# Merge Function
def Merge(list_of_elements, l, m, h):
 n1 = m-l+1
 n2 = h-m
 left = [0]*n1
 right = [0]*n2
 for x in range(n1):
    left[x] = list_of_elements[x+l]
 for x in range(n2):
    right[x] = list_of_elements[x+1+m]
 x = 0
 y = 0
 z = l
 while x < n1 and y < n2:
```

```
    if left[x] < right[y]:
        list_of_elements[z] = left[x];
        x = x + 1
    else:
        list_of_elements[z] = right[y]
        y = y + 1
    z += 1
 while x < n1:
    list_of_elements[z] = left[x]
    z = z + 1
    x = x + 1
 while y < n2:
    list_of_elements[z] = right[y]
    z = z + 1
    y = y + 1


# Implementing MERGE SORT
def merge_sort(list_of_elements, l, h):
 if l < h:
    m = (l+h)//2
    merge_sort(list_of_elements, l, m)
    merge_sort(list_of_elements, m+1, h)
    Merge(list_of_elements, l, m, h)
```

```
# Implementing HEAP SORT

# heapify function

def heapify(list_of_elements, x, n):

    l = 2*x + 1

    h = 2*x + 2

    y = x

    if (l < n) and (list_of_elements[l] > list_of_elements[x]):

        x = l

    if (h < n) and (list_of_elements[h] > list_of_elements[x]):

        x = h

    if x != y:

        list_of_elements[x], list_of_elements[y] = list_of_elements[y], list_of_elements[x]

        heapify(list_of_elements, x, n)


# Implementing HEAP SORT

def heap_sort(list_of_elements, l, h):

    n = len(list_of_elements)

    for x in range(n//2, -1, -1):

        heapify(list_of_elements, x, n)

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

        list_of_elements[x], list_of_elements[0] = list_of_elements[0], list_of_elements[x]

        heapify(list_of_elements, 0, x)
```

```
# Implementing QUICK SORT

# quick sort partition function

def quick_sort_partition(list_of_elements, l, h):
    pivot = list_of_elements[l]
    x = l+1
    y = h
    while x < y:
        while x <= h and list_of_elements[x] <= pivot:
            x += 1
        while y >= 0 and list_of_elements[y] > pivot:
            y -= 1
        if x < y:
            temp = list_of_elements[x]
            list_of_elements[x] = list_of_elements[y]
            list_of_elements[y] = temp
        else:
            temp = list_of_elements[y]
            list_of_elements[y] = pivot
            list_of_elements[l] = temp
    return y


# Implementing QUICK SORT
```

```
def quick_sort(list_of_elements, l, h):
    if l < h:
        pi = quick_sort_partition(list_of_elements, l, h)
        quick_sort(list_of_elements, l, pi-1)
        quick_sort(list_of_elements, pi+1, h)

# Implementing MODIFIED QUICK SORT
# Median function (Use median-of-three as pivot.)
def Median(list_of_elements, low, high, mid):
    i = list_of_elements[low]
    j = list_of_elements[mid]
    k = list_of_elements[high]
    if i <= j <= k:
        return j, mid
    if k <= j <= i:
        return j, mid
    if i <= k <= j:
        return k, high
    if j <= k <= i:
        return k, high
    return i, low

# modified quick sort partition function
```

```
def modified_quick_sort_Partition(list_of_elements, l, h):
    pivot, idx = Median(list_of_elements, l, h, (l+h)//2)
    x = l+1
    y = h
    while x < y:
        while x <= h and list_of_elements[x] <= pivot:
            x += 1
        while y >= 0 and list_of_elements[y] > pivot:
            y -= 1
        if x < y:
            temp = list_of_elements[x]
            list_of_elements[x] = list_of_elements[y]
            list_of_elements[y] = temp

        else:
            temp = list_of_elements[y]
            list_of_elements[y] = pivot
            list_of_elements[idx] = temp
    return y

#Implementing MODIFIED QUICK SORT
def modified_quick_sort(list_of_elements, l, h):
    if l < h:
```

```
        if (h-l) >= 8:  #For small sub-problem of size <= 8 , you must use insertion sort.
            pi = modified_quick_sort_Partition(list_of_elements, l, h)
            quick_sort(list_of_elements, l, pi-1)
            quick_sort(list_of_elements, pi+1, h)
    else:
        for x in range(l+1, h+1):
            key = list_of_elements[x]
            y = x-1
            while y >= l and list_of_elements[y] > key:
                list_of_elements[y+1] = list_of_elements[y]
                y = y - 1
            list_of_elements[y+1] = key


# Variables declarations

# declaring the array sizes of the different inputs sizes  (e.g. n = 1000, 2000, 4K, 5K, 10K, 20K, 40K, 50K, 60K, 80K, 100K).

# sizes = [x for x in range(0,100000,1000)]

sizes = [0,1000,2000,4000,5000,10000,20000,40000,50000,60000,80000,100000]

# Algorithms time declaration

insertion_sort_time = {}

merge_sort_time = {}

heap_sort_time = {}

quick_sort_time = {}

modified_quick_sort_time = {}
```

```
# Sorted Algorithms time declaration (special case)

insertion_sort_time_sorted = {}

merge_sort_time_sorted = {}

heap_sort_time_sorted = {}

quick_sort_time_sorted = {}

modified_quick_sort_time_sorted = {}


# reverse sorted Algorithms time declaration (special case)

insertion_sort_time_reverse = {}

merge_sort_time_reverse = {}

heap_sort_time_reverse = {}

quick_sort_time_reverse = {}

modified_quick_sort_time_reverse = {}



for x in sizes:        # executing each sizes from the given instructions

    list_of_elements = []

    for xyz in range(x):

        list_of_elements.append(random.randint(1, 1000)) # generating random values in the range
of 1 to 1000

    print(list_of_elements)

    randomArray = list_of_elements

    sortedArray = sorted(list_of_elements)
```

```
reverseSortedArray = sorted(list_of_elements)[-1::-1]


t = time()
insertion_sort(randomArray)
t = time()-t
insertion_sort_time[x] = t


t = time()
insertion_sort(sortedArray)
t = time()-t
insertion_sort_time_sorted[x] = t


t = time()
insertion_sort(reverseSortedArray)
t = time()-t
insertion_sort_time_reverse[x] = t


t = time()
merge_sort(randomArray, 0, len(randomArray)-1)
t = time()-t
merge_sort_time[x] = t


t = time()
```

```
merge_sort(sortedArray, 0, len(randomArray)-1)
t = time()-t
merge_sort_time_sorted[x] = t

t = time()
merge_sort(reverseSortedArray, 0, len(randomArray)-1)
t = time()-t
merge_sort_time_reverse[x] = t

t = time()
heap_sort(randomArray, 0, len(randomArray)-1)
t = time()-t
heap_sort_time[x] = t

t = time()
heap_sort(sortedArray, 0, len(randomArray)-1)
t = time()-t
heap_sort_time_sorted[x] = t

t = time()
heap_sort(reverseSortedArray, 0, len(randomArray)-1)
t = time()-t
heap_sort_time_reverse[x] = t
```

```
randomArray = list_of_elements

t = time()

quick_sort(randomArray, 0, len(randomArray)-1)

t = time()-t

quick_sort_time[x] = t


t = time()

quick_sort(sortedArray, 0, len(randomArray)-1)

t = time()-t

quick_sort_time_sorted[x] = t


t = time()

quick_sort(reverseSortedArray, 0, len(randomArray)-1)

t = time()-t

quick_sort_time_reverse[x] = t


randomArray = list_of_elements

t = time()

modified_quick_sort(randomArray, 0, len(randomArray)-1)

t = time()-t

modified_quick_sort_time[x] = t
```

```
    t = time()
    modified_quick_sort(sortedArray, 0, len(randomArray)-1)
    t = time()-t
    modified_quick_sort_time_sorted[x] = t

    t = time()
    modified_quick_sort(reverseSortedArray, 0, len(randomArray)-1)
    t = time()-t
    modified_quick_sort_time_reverse[x] = t

print("Random Generated Values")

print("Insertion Sort")
for i in insertion_sort_time.items():
    print("%s\t%s"%(i[0], i[1]))

print("Merge Sort")
for i in merge_sort_time.items():
    print("%s\t%s"%(i[0], i[1]))

print("Heap Sort")
for i in heap_sort_time.items():
```

```
    print("%s\t%s"%(i[0], i[1]))


print("Quick Sort")
for i in quick_sort_time.items():
    print("%s\t%s"%(i[0], i[1]))


print("Modified Quick Sort")
for i in modified_quick_sort_time.items():
    print("%s\t%s"%(i[0], i[1]))


print("\n")


print("Random Generated Values Sorted")


print("Insertion Sort")
for i in insertion_sort_time_sorted.items():
    print("%s\t%s"%(i[0], i[1]))


print("Merge Sort")
for i in merge_sort_time_sorted.items():
    print("%s\t%s"%(i[0], i[1]))


print("Heap Sort")
```

```
for i in heap_sort_time_sorted.items():
    print("%s\t%s"%(i[0], i[1]))

print("Quick Sort")
for i in quick_sort_time_sorted.items():
    print("%s\t%s"%(i[0], i[1]))

print("Modified Quick Sort")
for i in modified_quick_sort_time_sorted.items():
    print("%s\t%s"%(i[0], i[1]))

print("\n")

print("Random Generated Values Reverse Sorted")

print("Insertion Sort")
for i in insertion_sort_time_reverse.items():
    print("%s\t%s"%(i[0], i[1]))

print("Merge Sort")
for i in merge_sort_time_reverse.items():
    print("%s\t%s"%(i[0], i[1]))
```

```
print("Heap Sort")
for i in heap_sort_time_reverse.items():
    print("%s\t%s"%(i[0], i[1]))

print("Quick Sort")
for i in quick_sort_time_reverse.items():
    print("%s\t%s"%(i[0], i[1]))

print("Modified Quick Sort")
for i in modified_quick_sort_time_reverse.items():
    print("%s\t%s"%(i[0], i[1]))

print("\n")

# Graph plotting for all algorithms time.
myPlot.plot(sizes, list(insertion_sort_time.values()), "r",label = "Insertion Sort" )
myPlot.plot(sizes, list(merge_sort_time.values()), "g", label = "Merge Sort" )
myPlot.plot(sizes, list(heap_sort_time.values()), "b",label="Heap Sort")
myPlot.plot(sizes, list(quick_sort_time.values()), "y",label = "Quick Sort")
myPlot.plot(sizes, list(modified_quick_sort_time.values()), "m",label="Modified Quick Sort")
myPlot.title("randomly generated Numbers")
myPlot.xlabel("Number of elements")
myPlot.ylabel("Time taken by each Algorithms")
```

```
myPlot.legend()

myPlot.show()


# Graph plotting for all sorted algorithms time.

myPlot.plot(sizes, list(insertion_sort_time_sorted.values()), "r",label = "Insertion Sort")

myPlot.plot(sizes, list(merge_sort_time_sorted.values()), "g",label = "Merge Sort")

myPlot.plot(sizes, list(heap_sort_time_sorted.values()), "b",label = "Heap Sort")

myPlot.plot(sizes, list(quick_sort_time_sorted.values()), "y",label = "Quick Sort")

myPlot.plot(sizes, list(modified_quick_sort_time_sorted.values()), "m",label = "Modified Quick
Sort")

myPlot.title("Sorted Numbers")

myPlot.xlabel("Number of elements")

myPlot.ylabel("Time taken by each Algorithms")

myPlot.legend()

myPlot.show()


# Graph plotting for all reverse sorted algorithms time.

myPlot.plot(sizes, list(insertion_sort_time_reverse.values()), "r",label = "Insertion Sort")

myPlot.plot(sizes, list(merge_sort_time_reverse.values()), "g",label = "Merge Sort")

myPlot.plot(sizes, list(heap_sort_time_reverse.values()), "b",label = "Heap Sort")

myPlot.plot(sizes, list(quick_sort_time_reverse.values()), "y",label = "Quick Sort")

myPlot.plot(sizes, list(modified_quick_sort_time_reverse.values()), "m",label = "Modified
Quick Sort")

myPlot.title("Sorted Numbers in reverse order")
```

```
myPlot.xlabel("Number of elements")

myPlot.ylabel("Time taken by each Algorithms")

myPlot.legend()

myPlot.show()
```