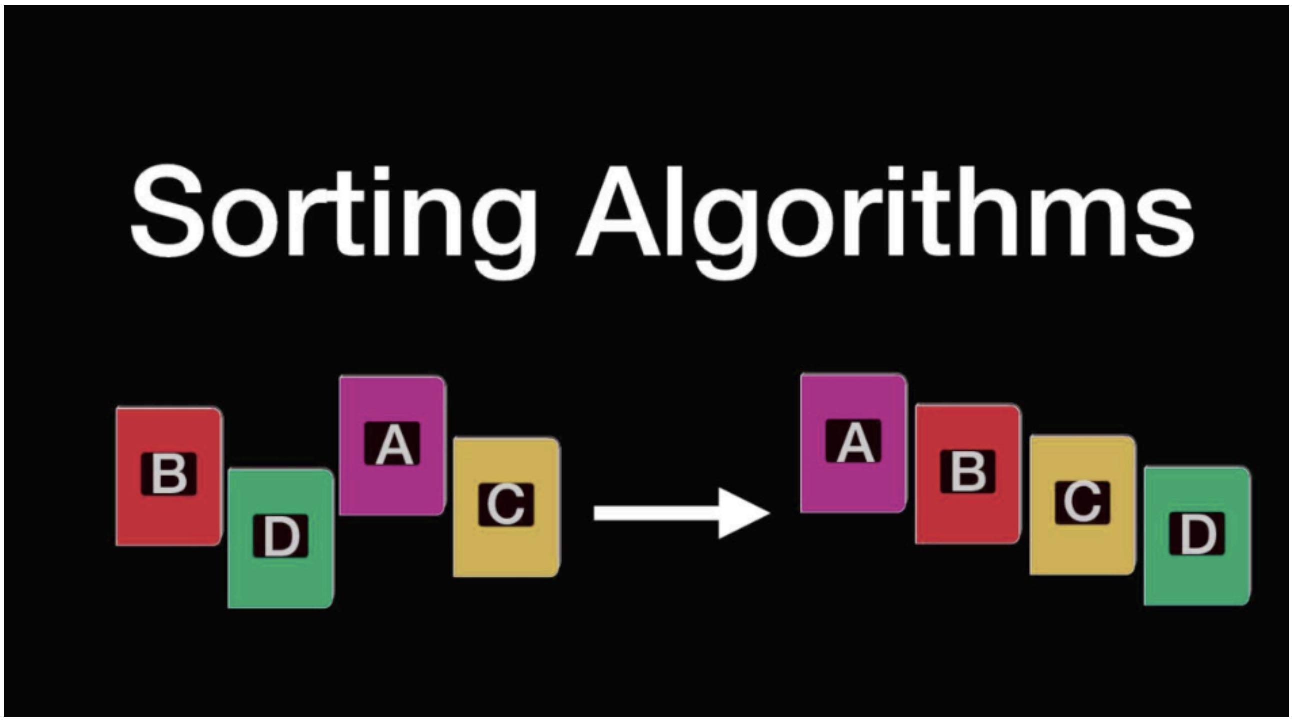
CSE-5311-002-DESIGN & ANALYSIS ALGORITHMS

Project Name: Sorting of an algorithms

Project Done by: Sayali Dilip Deshmukh

Email ID: sxd6629@mavs.uta.edu



Problem Statement of Project:

Implement and compare the following sorting algorithm :

- Merge sort
- Heapsort
- Quicksort
- Insertion sort
- Selection sort
- Bubble sort

Table of Contents:

1.Theory of sorting

2. Each individual sort program along with its theory

3. Project Specification and function

4. Project flow

5. Analysis of all graphs

6. conclusion

what is sorting exactly means?
Let's learn a bit more around what makes classifies a sorting algorithm. Here we are considering the 6 types of sorting and their implementation along with the comparison.
A Sorting techniques is utilised to modify a given cluster or list components . Sorting calculations are a set of informational that take a cluster or list as an input and orchestrate the things into a specific order.
The different sorting algorithms are a perfect example of how algorithm design can have such a strong effect on program complexity, speed, and efficiency.

- Merge sort
- Heapsort
- Quicksort
- Insertion sort
- Selection sort
- Bubble sort

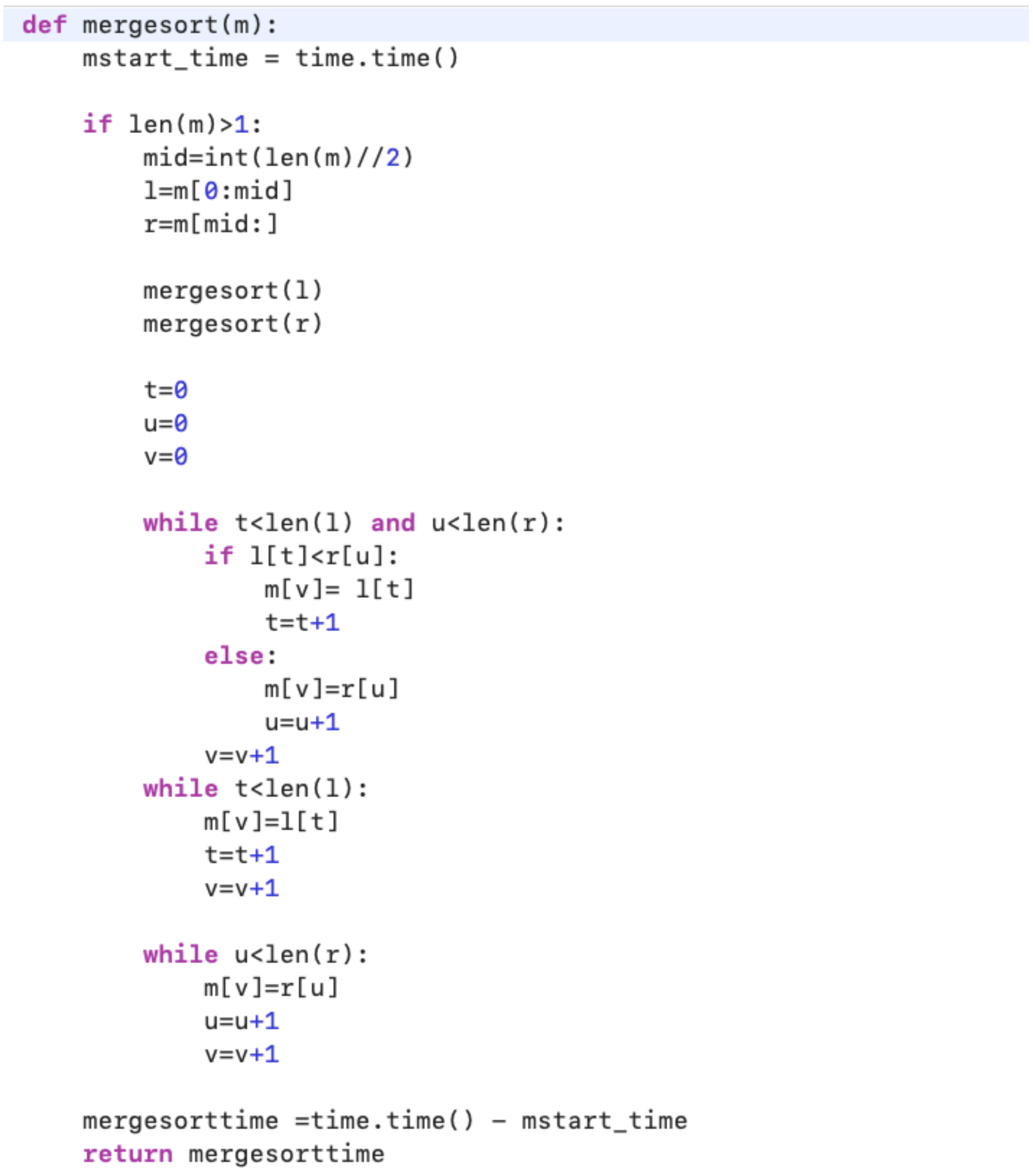
The given table shows the best average and worst case complexity:

Algorithms		Time Complexities	
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))
Quick Sort	O(n log(n))	O(n log(n))	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Bubble Sort	O(n)	O(n^2)	O(n^2)

1. Merge Sort

Merge sort is work on the principal of “divide and conquer”.

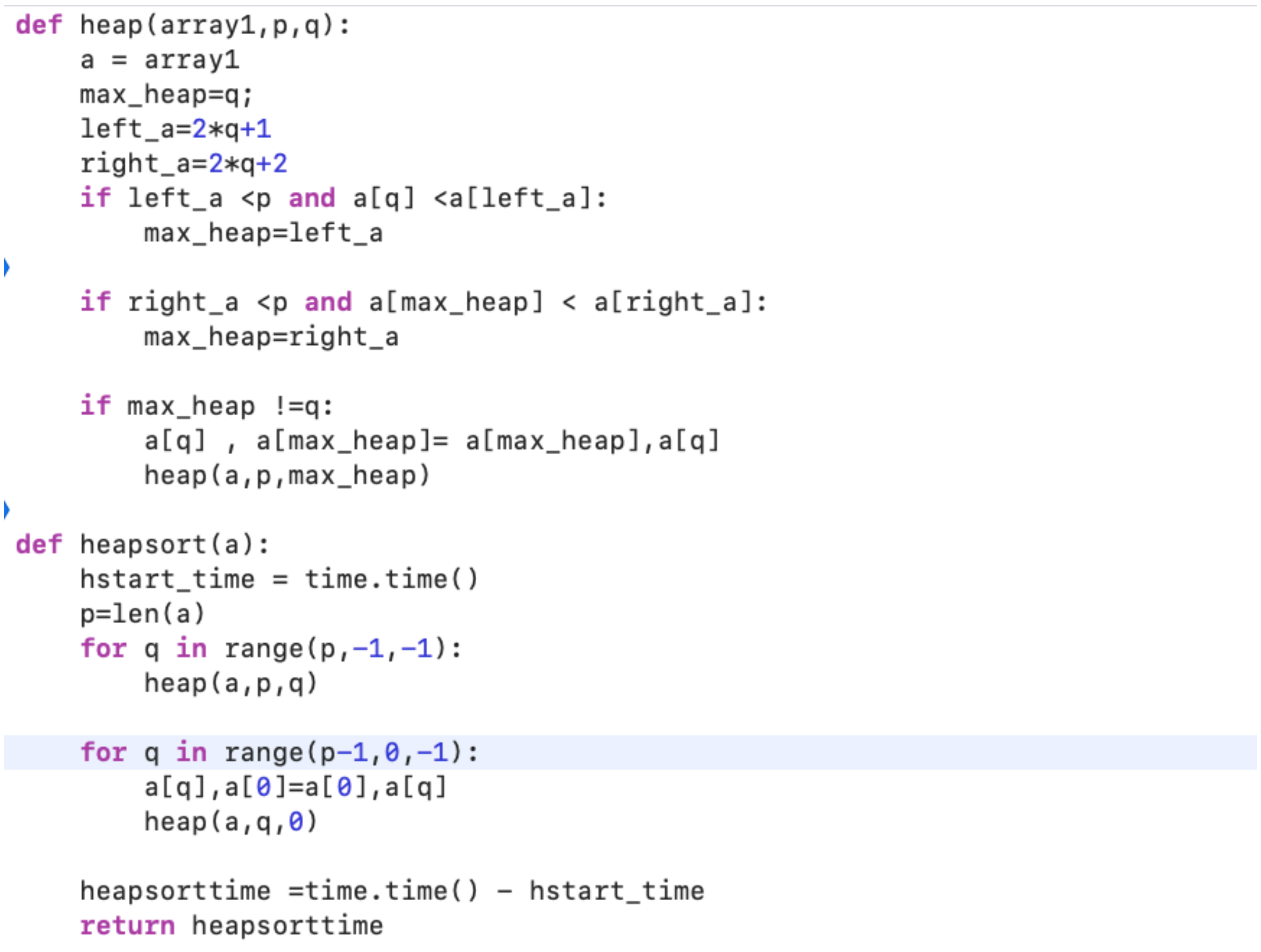
- To divide the given unsorted list into number of sublists
- Merge sublists to produce new sorted sublists
- We have to do until 1 sublist is remaining.
- We will obtain sorted list.



```
def mergesort(m):
    mstart_time = time.time()

    if len(m)>1:
        mid=int(len(m)//2)
        l=m[0:mid]
        r=m[mid:]

        mergesort(l)
        mergesort(r)

        t=0
        u=0
        v=0

        while t<len(l) and u<len(r):
            if l[t]<r[u]:
                m[v]= l[t]
                t=t+1
            else:
                m[v]=r[u]
                u=u+1
            v=v+1
        while t<len(l):
            m[v]=l[t]
            t=t+1
            v=v+1

        while u<len(r):
            m[v]=r[u]
            u=u+1
            v=v+1

    mergesorttime =time.time() - mstart_time
    return mergesorttime
```

2. Heap Sort

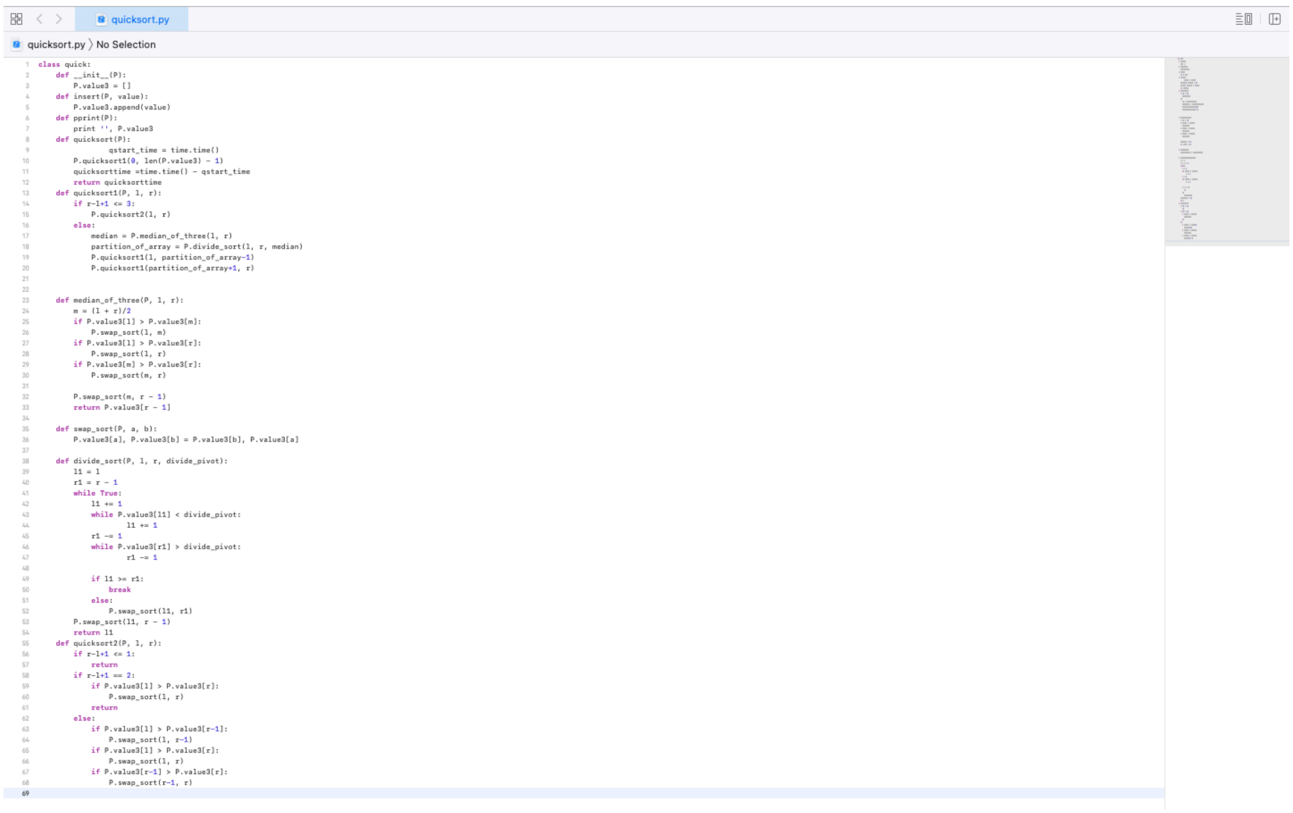
This comparison-based algorithm can be utilised for non-numerical information elements insofar as a few relation can be defined over the elements.This algorithm runs in O(nlogn) time and O(1) additional space O(n) including the space required to store the input data] since all operations are performed entirely in-place. After n iterations the Heap is empty, every iteration involves a swap and a heapify operation; hence it takes O(log n) time



```python
def heap(array1,p,q):
    a = array1
    max_heap=q;
    left_a=2*q+1
    right_a=2*q+2
    if left_a <p and a[q] <a[left_a]:
        max_heap=left_a

    if right_a <p and a[max_heap] < a[right_a]:
        max_heap=right_a

    if max_heap !=q:
        a[q] , a[max_heap]= a[max_heap],a[q]
        heap(a,p,max_heap)

def heapsort(a):
    hstart_time = time.time()
    p=len(a)
    for q in range(p,-1,-1):
        heap(a,p,q)

    for q in range(p-1,0,-1):
        a[q],a[0]=a[0],a[q]
        heap(a,q,0)

    heapsorttime =time.time() - hstart_time
    return heapsorttime
```

3.Quick Sort Using 3median:

Quick Sort is a Divide and Conquer algorithm.
Quicksort works in place and its worst-case running time is as bad as selection sort's and insertion sort's: O(n2). But its average-case running time is as good as merge sort's: o(nlogn).
In Quick sort most of the important task is done while dividing the array into subarrays. There are many ways to deals with the quick sort.



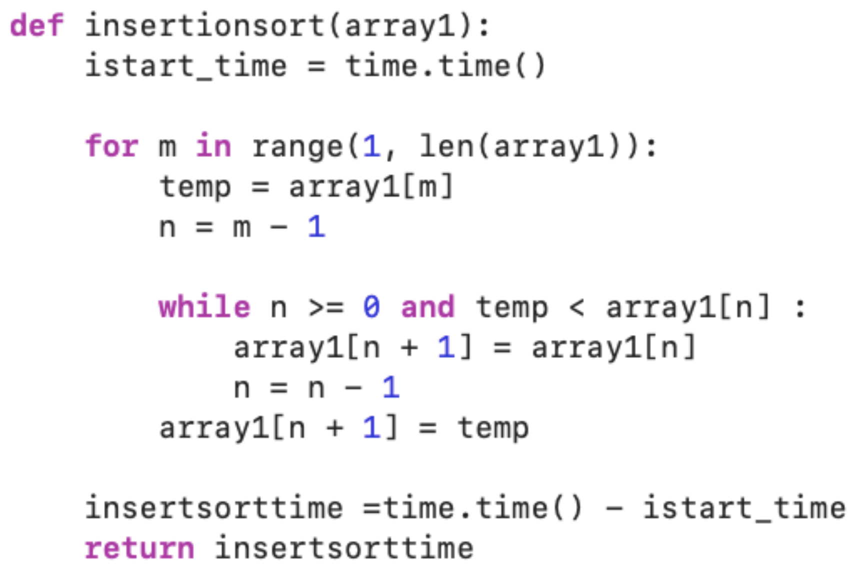
```
class quick:
    def __init__(P):
        P.value3 = []
    def insert(P, value):
        P.value3.append(value)
    def pprint(P):
        print '', P.value3
    def quicksort(P):
                qstart_time = time.time()
        P.quicksort1(0, len(P.value3) - 1)
        quicksorttime =time.time() - qstart_time
        return quicksorttime
    def quicksort1(P, l, r):
        if r-l+1 <= 3:
            P.quicksort2(l, r)
        else:
            median = P.median_of_three(l, r)
            partition_of_array = P.divide_sort(l, r, median)
            P.quicksort1(l, partition_of_array-1)
            P.quicksort1(partition_of_array+1, r)


    def median_of_three(P, l, r):
        m = (l + r)/2
        if P.value3[l] > P.value3[m]:
            P.swap_sort(l, m)
        if P.value3[l] > P.value3[r]:
            P.swap_sort(l, r)
        if P.value3[m] > P.value3[r]:
            P.swap_sort(m, r)

        P.swap_sort(m, r - 1)
        return P.value3[r - 1]

    def swap_sort(P, a, b):
        P.value3[a], P.value3[b] = P.value3[b], P.value3[a]

    def divide_sort(P, l, r, divide_pivot):
        l1 = l
        r1 = r - 1
        while True:
            l1 += 1
            while P.value3[l1] < divide_pivot:
                    l1 += 1
            r1 -= 1
            while P.value3[r1] > divide_pivot:
                    r1 -= 1

            if l1 >= r1:
                break
            else:
                P.swap_sort(l1, r1)
        P.swap_sort(l1, r - 1)
        return l1
    def quicksort2(P, l, r):
        if r-l+1 <= 1:
            return
        if r-l+1 == 2:
            if P.value3[l] > P.value3[r]:
                P.swap_sort(l, r)
            return
        else:
            if P.value3[l] > P.value3[r-1]:
                P.swap_sort(l, r-1)
            if P.value3[l] > P.value3[r]:
                P.swap_sort(l, r)
            if P.value3[r-1] > P.value3[r]:
                P.swap_sort(r-1, r)
```

4. Insertion Sort

- Insertion sort is the sorting algorithm that splits the given array into sorted and unsorted list.
- Values in an unsorted list are placed in proper position

  It works like:

1. Comparison of an element with the next neighbouring element
2. Elements are shifted to the right according to their position by creating space
3. Repeat step 1 and 2 until desired position of element is obtained.

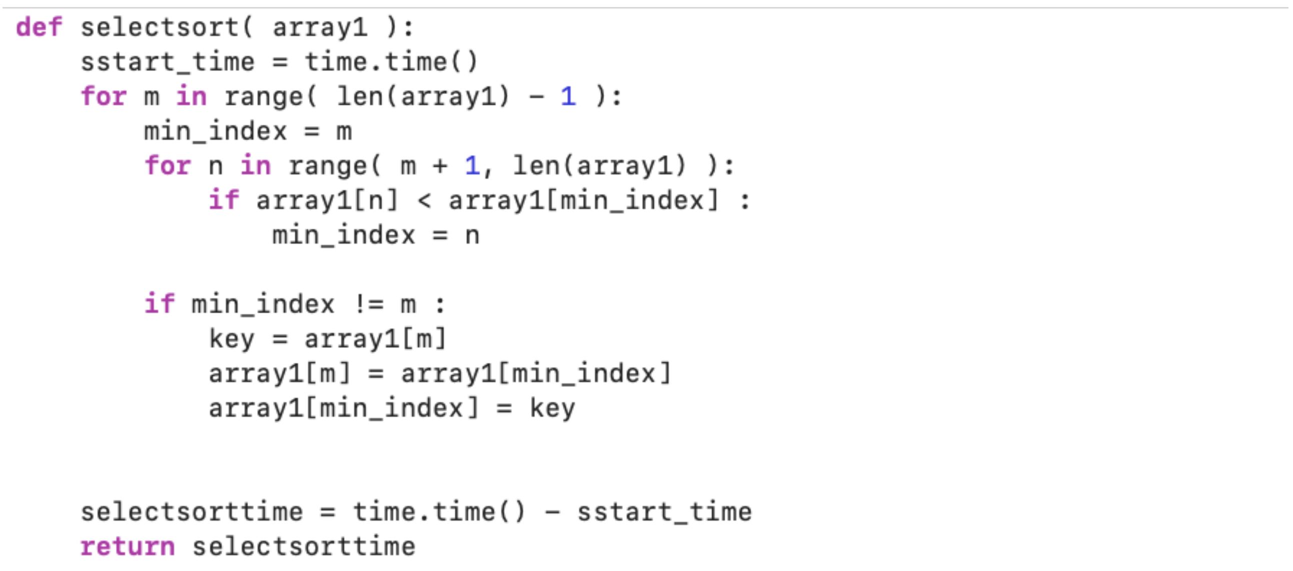


```python
def insertionsort(array1):
    istart_time = time.time()

    for m in range(1, len(array1)):
        temp = array1[m]
        n = m - 1

        while n >= 0 and temp < array1[n] :
            array1[n + 1] = array1[n]
            n = n - 1
        array1[n + 1] = temp

    insertsorttime =time.time() - istart_time
    return insertsorttime
```

5. Selection Sort

- To find the smallest element
- Fit the smallest element to the first position
- Another smallest element grater than first element into the next position
- Repeat the above steps to fit the each element to the correct position.

The main motto is to sort the elements according to the above procedure.



```
def selectsort( array1 ):
    sstart_time = time.time()
    for m in range( len(array1) - 1 ):
        min_index = m
        for n in range( m + 1, len(array1) ):
            if array1[n] < array1[min_index] :
                min_index = n

        if min_index != m :
            key = array1[m]
            array1[m] = array1[min_index]
            array1[min_index] = key


    selectsorttime = time.time() - sstart_time
    return selectsorttime
```

6. Bubble Sort

- Compares the given elements with adjacent pair elements
- To swap the pair of elements if they are not in the correct position
- It will pass the largest element to the end of the list.
- Scan again and repeat the above to find largest and position into second largest
- We will obtain the sorted list

Bubble sort repetitively compares adjacent pairs of elements and swaps if necessary.

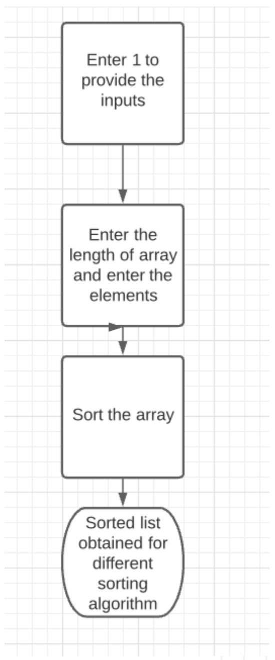
```python
def bubblesort(array1):
    p=len(array1)
    bstart_time = time.time()
    for i in range(len(array1)):
        array_sorted = True
        for j in range(len(array1) - i - 1):
            if array1[j]> array1[j+1]:
                array1[j],array1[j+1] = array1[j+1],array1[j]
                array_sorted = False

        if array_sorted:
            break
    bubblesorttime =time.time() - bstart_time
    return bubblesorttime
```

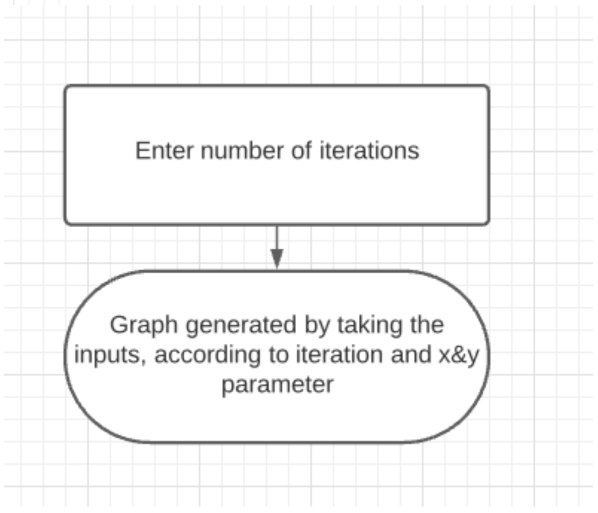
Project Specifications:

1. Enter the given option to provide inputs
2. Then enter the length of the array
3. Press Enter to provide the list of elements
4. The elements will be sorted according to their order.
5. Obtained the running time in seconds.
6. We can generate the graph by providing the iterations.
7. We obtained the graph and achieve exact runtime.

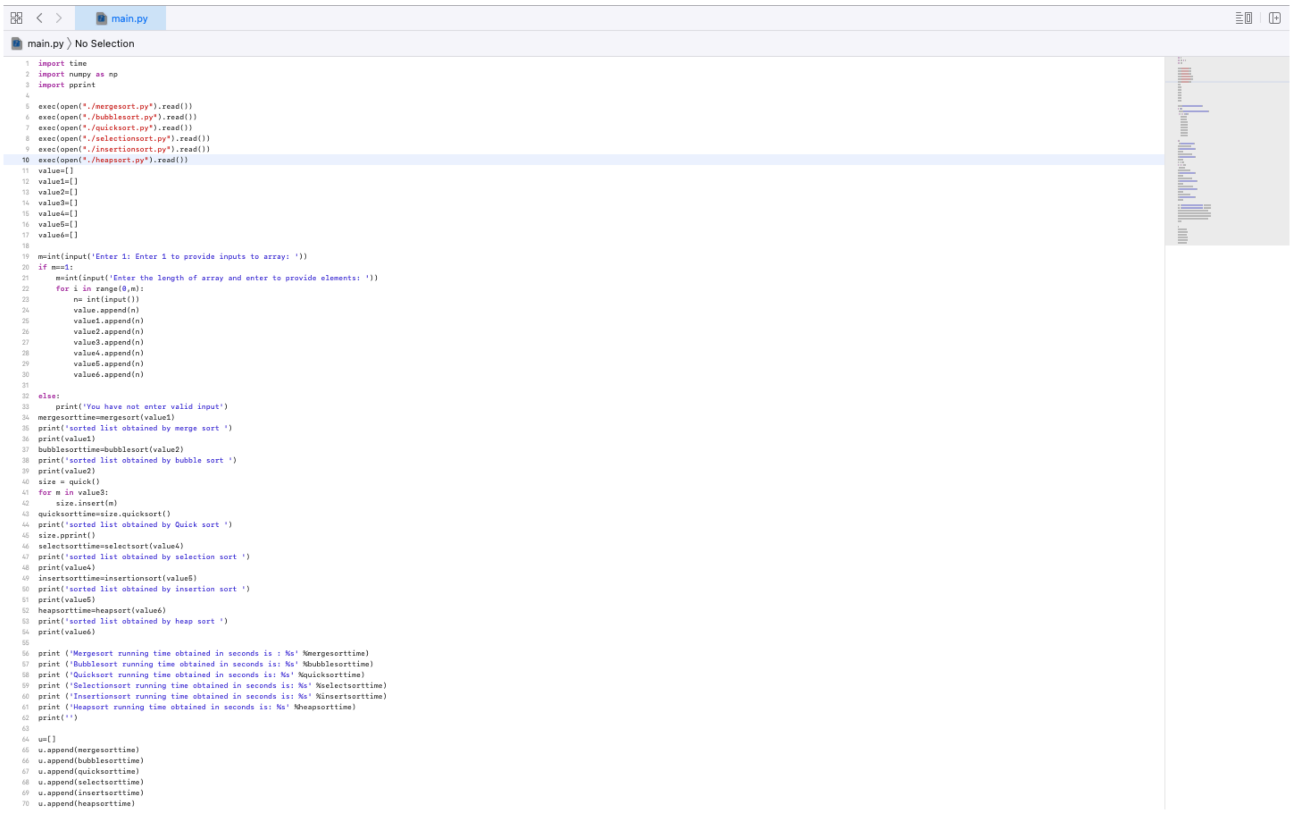
Project Flowchart:

1. Execution of program
2.Generation of graph


Enter 1 to provide the inputs
Enter the length of array and enter the elements
Sort the array
Sorted list obtained for different sorting algorithm

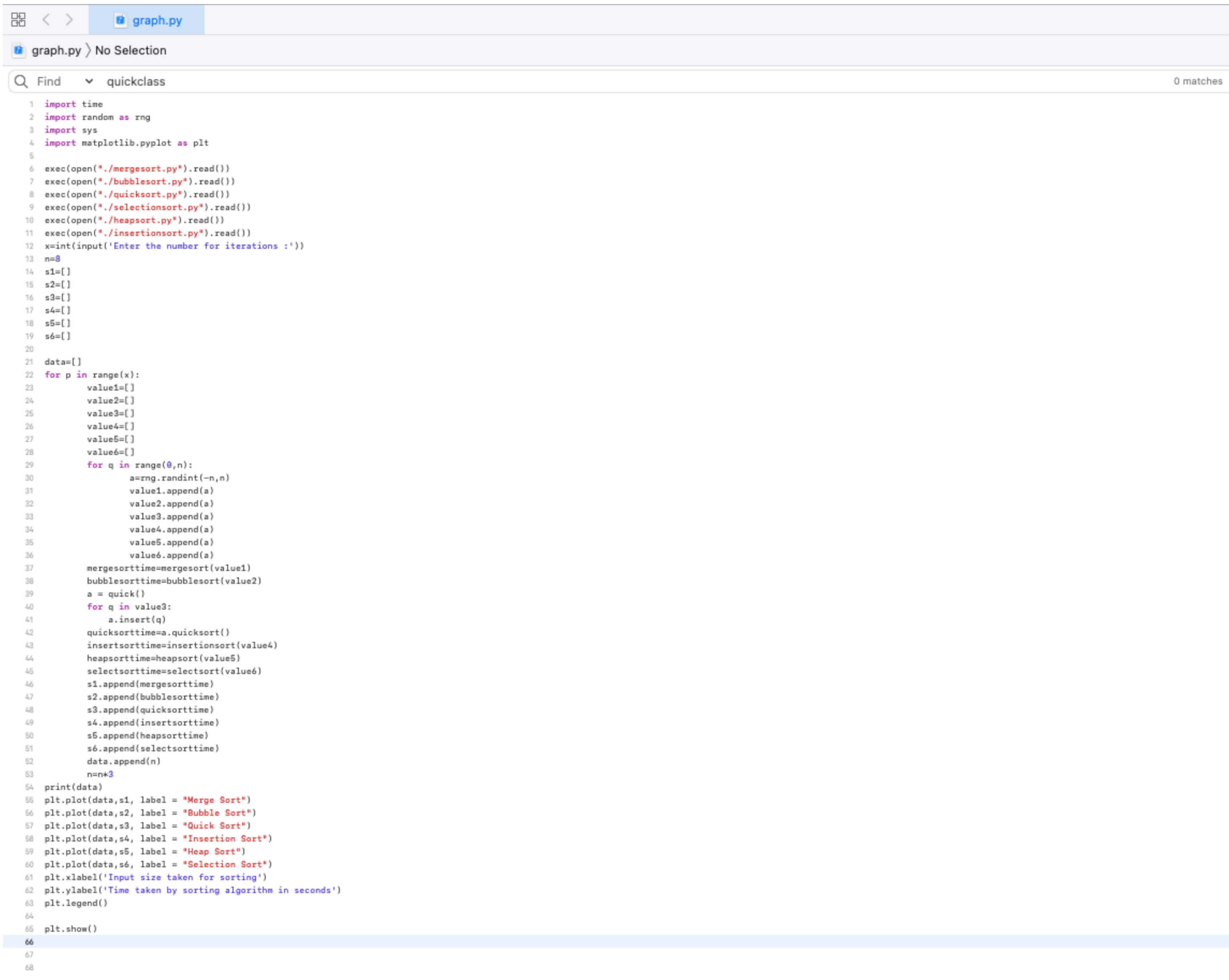

Enter number of iterations
Graph generated by taking the inputs, according to iteration and x&y parameter

Main file Program including functions:



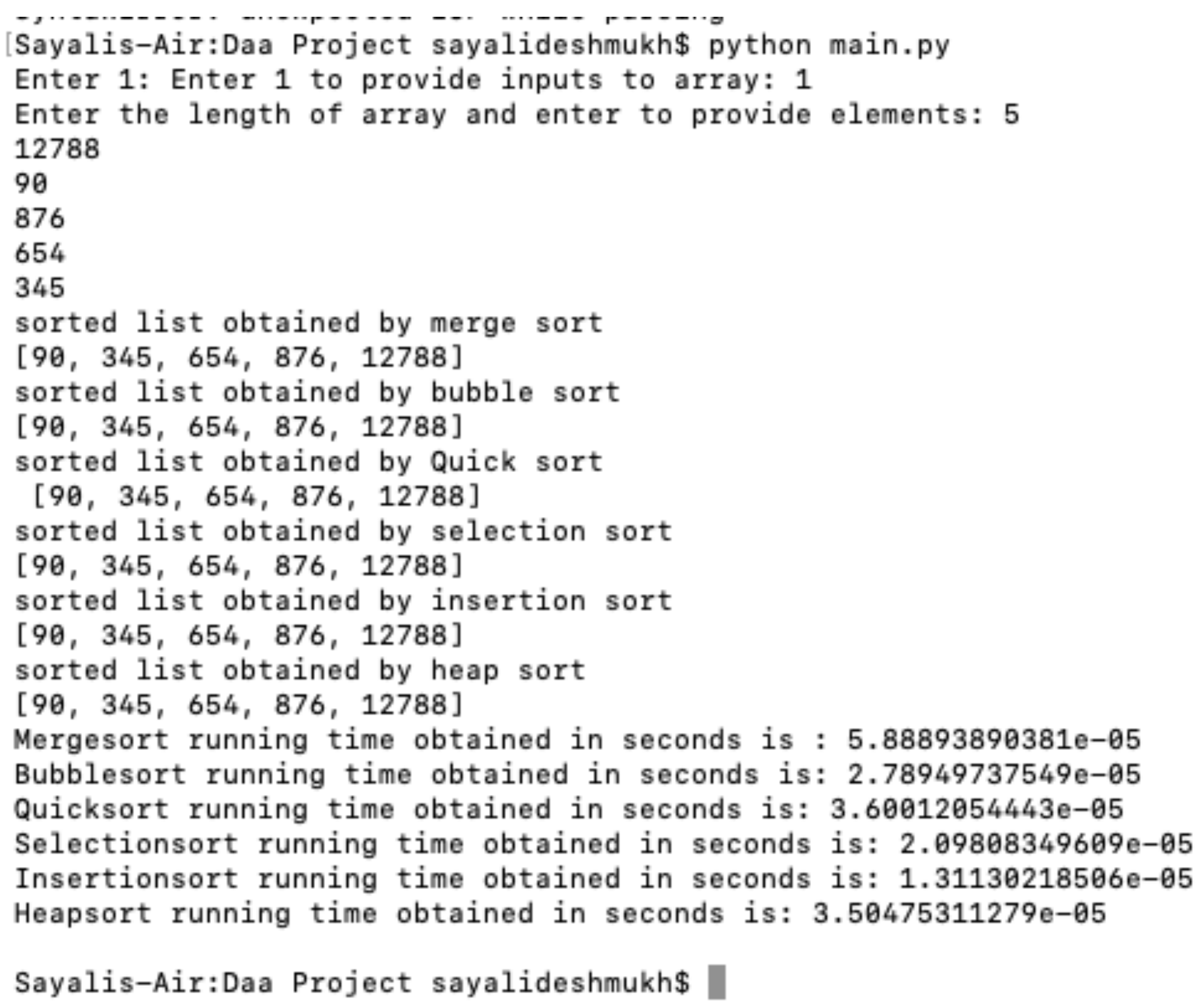
```python
import time
import numpy as np
import pprint

exec(open("./mergesort.py").read())
exec(open("./bubblesort.py").read())
exec(open("./quicksort.py").read())
exec(open("./selectionsort.py").read())
exec(open("./insertionsort.py").read())
exec(open("./heapsort.py").read())
value=[]
value1=[]
value2=[]
value3=[]
value4=[]
value5=[]
value6=[]

m=int(input('Enter 1: Enter 1 to provide inputs to array: '))
if m==1:
    m=int(input('Enter the length of array and enter to provide elements: '))
    for i in range(0,m):
        n= int(input())
        value.append(n)
        value1.append(n)
        value2.append(n)
        value3.append(n)
        value4.append(n)
        value5.append(n)
        value6.append(n)

else:
    print('You have not enter valid input')
mergesorttime=mergesort(value1)
print('sorted list obtained by merge sort ')
print(value1)
bubblesorttime=bubblesort(value2)
print('sorted list obtained by bubble sort ')
print(value2)
size = quick()
for m in value3:
    size.insert(m)
quicksorttime=size.quicksort()
print('sorted list obtained by Quick sort ')
size.pprint()
selectsorttime=selectsort(value4)
print('sorted list obtained by selection sort ')
print(value4)
insertsorttime=insertionsort(value5)
print('sorted list obtained by insertion sort ')
print(value5)
heapsorttime=heapsort(value6)
print('sorted list obtained by heap sort ')
print(value6)

print ('Mergesort running time obtained in seconds is : %s' %mergesorttime)
print ('Bubblesort running time obtained in seconds is: %s' %bubblesorttime)
print ('Quicksort running time obtained in seconds is: %s' %quicksorttime)
print ('Selectionsort running time obtained in seconds is: %s' %selectsorttime)
print ('Insertionsort running time obtained in seconds is: %s' %insertsorttime)
print ('Heapsort running time obtained in seconds is: %s' %heapsorttime)
print('')

u=[]
u.append(mergesorttime)
u.append(bubblesorttime)
u.append(quicksorttime)
u.append(selectsorttime)
u.append(insertsorttime)
u.append(heapsorttime)
```

GUI Program importing the other sorting files:



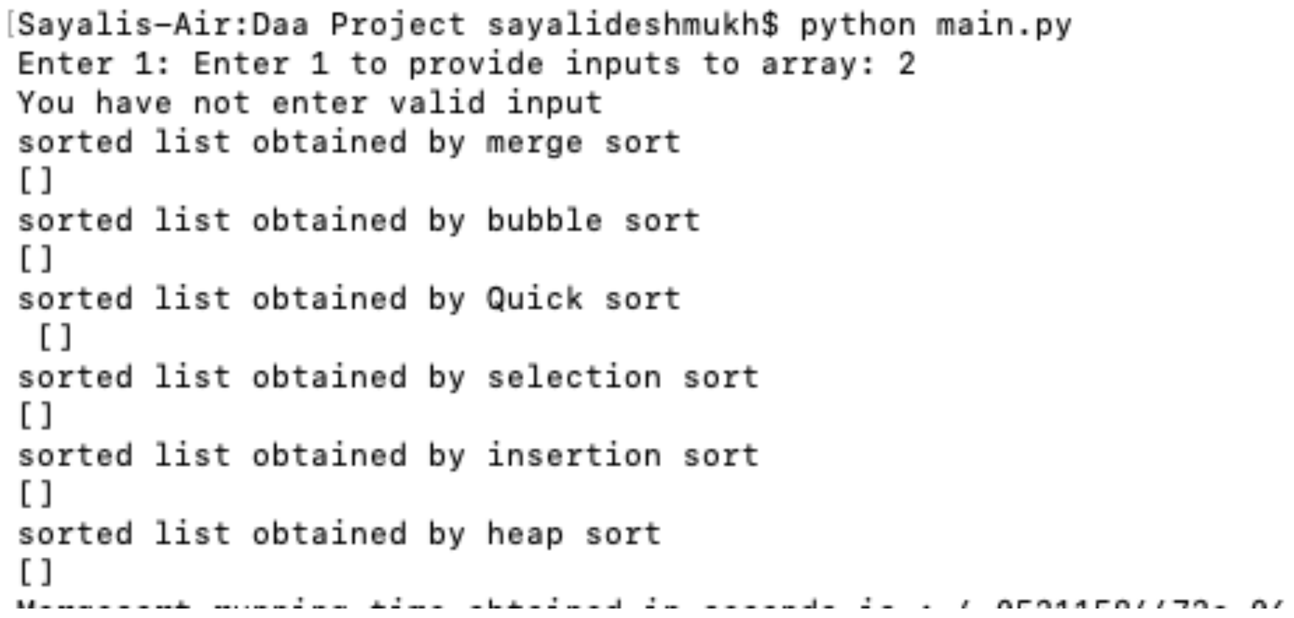
```python
import time
import random as rng
import sys
import matplotlib.pyplot as plt

exec(open("./mergesort.py").read())
exec(open("./bubblesort.py").read())
exec(open("./quicksort.py").read())
exec(open("./selectionsort.py").read())
exec(open("./heapsort.py").read())
exec(open("./insertionsort.py").read())
x=int(input('Enter the number for iterations :'))
n=8
s1=[]
s2=[]
s3=[]
s4=[]
s5=[]
s6=[]

data=[]
for p in range(x):
        value1=[]
        value2=[]
        value3=[]
        value4=[]
        value5=[]
        value6=[]
        for q in range(0,n):
                a=rng.randint(-n,n)
                value1.append(a)
                value2.append(a)
                value3.append(a)
                value4.append(a)
                value5.append(a)
                value6.append(a)
        mergesorttime=mergesort(value1)
        bubblesorttime=bubblesort(value2)
        a = quick()
        for q in value3:
            a.insert(q)
        quicksorttime=a.quicksort()
        insertsorttime=insertionsort(value4)
        heapsorttime=heapsort(value5)
        selectsorttime=selectsort(value6)
        s1.append(mergesorttime)
        s2.append(bubblesorttime)
        s3.append(quicksorttime)
        s4.append(insertsorttime)
        s5.append(heapsorttime)
        s6.append(selectsorttime)
        data.append(n)
        n=n*3
print(data)
plt.plot(data,s1, label = "Merge Sort")
plt.plot(data,s2, label = "Bubble Sort")
plt.plot(data,s3, label = "Quick Sort")
plt.plot(data,s4, label = "Insertion Sort")
plt.plot(data,s5, label = "Heap Sort")
plt.plot(data,s6, label = "Selection Sort")
plt.xlabel('Input size taken for sorting')
plt.ylabel('Time taken by sorting algorithm in seconds')
plt.legend()

plt.show()
```

Output of the console and to examine the running time according to the sorting algorithms



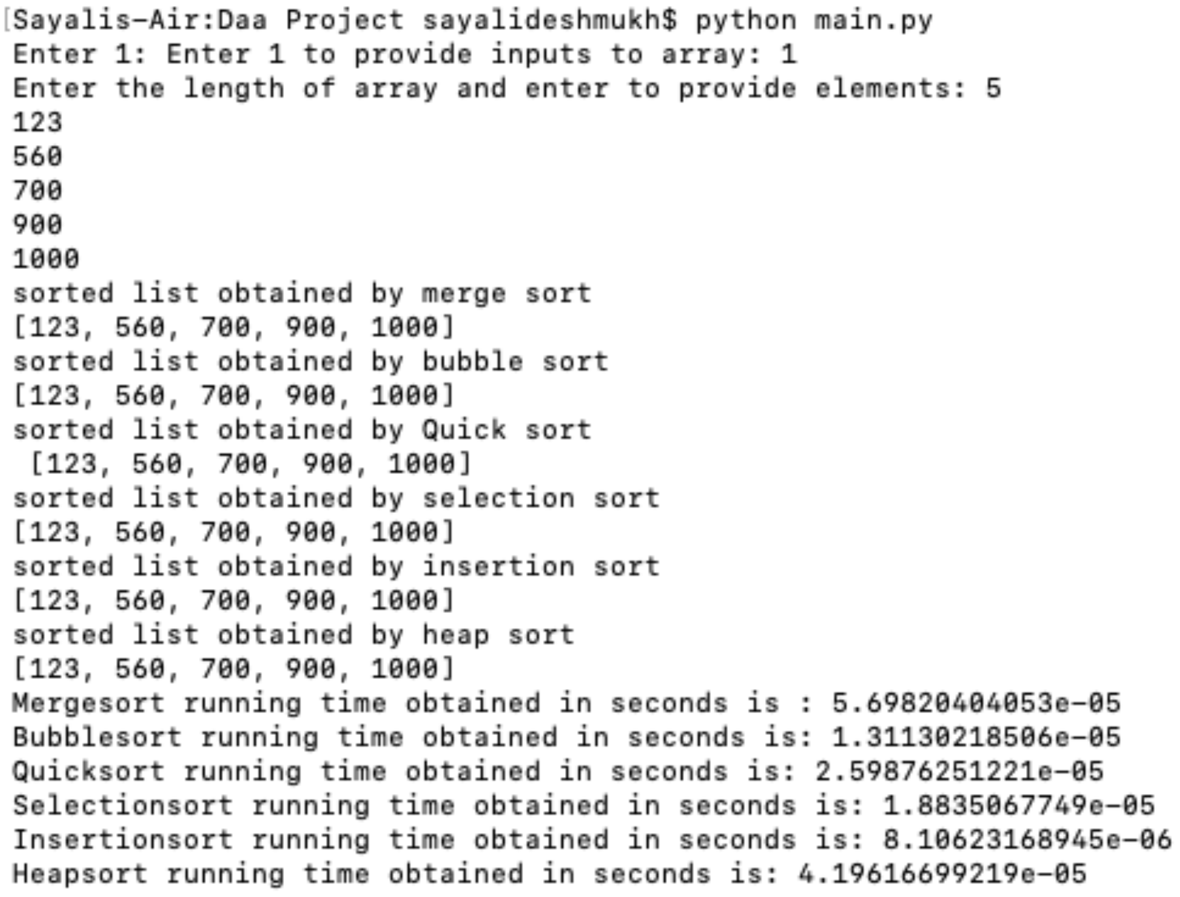
```
[Sayalis-Air:Daa Project sayalideshmukh$ python main.py
Enter 1: Enter 1 to provide inputs to array: 1
Enter the length of array and enter to provide elements: 5
12788
90
876
654
345
sorted list obtained by merge sort
[90, 345, 654, 876, 12788]
sorted list obtained by bubble sort
[90, 345, 654, 876, 12788]
sorted list obtained by Quick sort
 [90, 345, 654, 876, 12788]
sorted list obtained by selection sort
[90, 345, 654, 876, 12788]
sorted list obtained by insertion sort
[90, 345, 654, 876, 12788]
sorted list obtained by heap sort
[90, 345, 654, 876, 12788]
Mergesort running time obtained in seconds is : 5.88893890381e-05
Bubblesort running time obtained in seconds is: 2.78949737549e-05
Quicksort running time obtained in seconds is: 3.60012054443e-05
Selectionsort running time obtained in seconds is: 2.09808349609e-05
Insertionsort running time obtained in seconds is: 1.31130218506e-05
Heapsort running time obtained in seconds is: 3.50475311279e-05

Sayalis-Air:Daa Project sayalideshmukh$ █
```

It shows empty results when we are not properly providing the input properly :



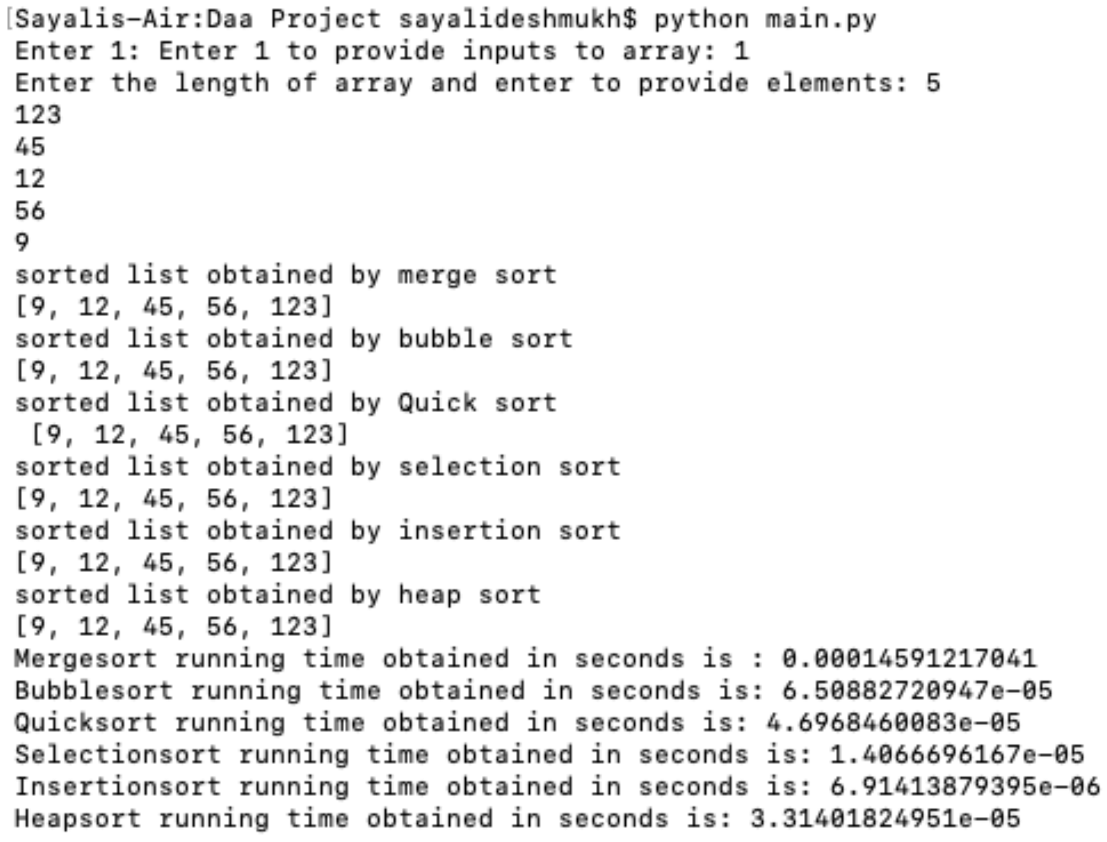
```
[Sayalis-Air:Daa Project sayalideshmukh$ python main.py
Enter 1: Enter 1 to provide inputs to array: 2
You have not enter valid input
sorted list obtained by merge sort
[]
sorted list obtained by bubble sort
[]
sorted list obtained by Quick sort
 []
sorted list obtained by selection sort
[]
sorted list obtained by insertion sort
[]
sorted list obtained by heap sort
[]
[illegible]
```

Here I have provided the elements according to their ascending order and find outs the perfect output and running time.



```
[Sayalis-Air:Daa Project sayalideshmukh$ python main.py
 Enter 1: Enter 1 to provide inputs to array: 1
 Enter the length of array and enter to provide elements: 5
 123
 560
 700
 900
 1000
 sorted list obtained by merge sort
 [123, 560, 700, 900, 1000]
 sorted list obtained by bubble sort
 [123, 560, 700, 900, 1000]
 sorted list obtained by Quick sort
  [123, 560, 700, 900, 1000]
 sorted list obtained by selection sort
 [123, 560, 700, 900, 1000]
 sorted list obtained by insertion sort
 [123, 560, 700, 900, 1000]
 sorted list obtained by heap sort
 [123, 560, 700, 900, 1000]
 Mergesort running time obtained in seconds is : 5.69820404053e-05
 Bubblesort running time obtained in seconds is: 1.31130218506e-05
 Quicksort running time obtained in seconds is: 2.59876251221e-05
 Selectionsort running time obtained in seconds is: 1.8835067749e-05
 Insertionsort running time obtained in seconds is: 8.10623168945e-06
 Heapsort running time obtained in seconds is: 4.19616699219e-05
```

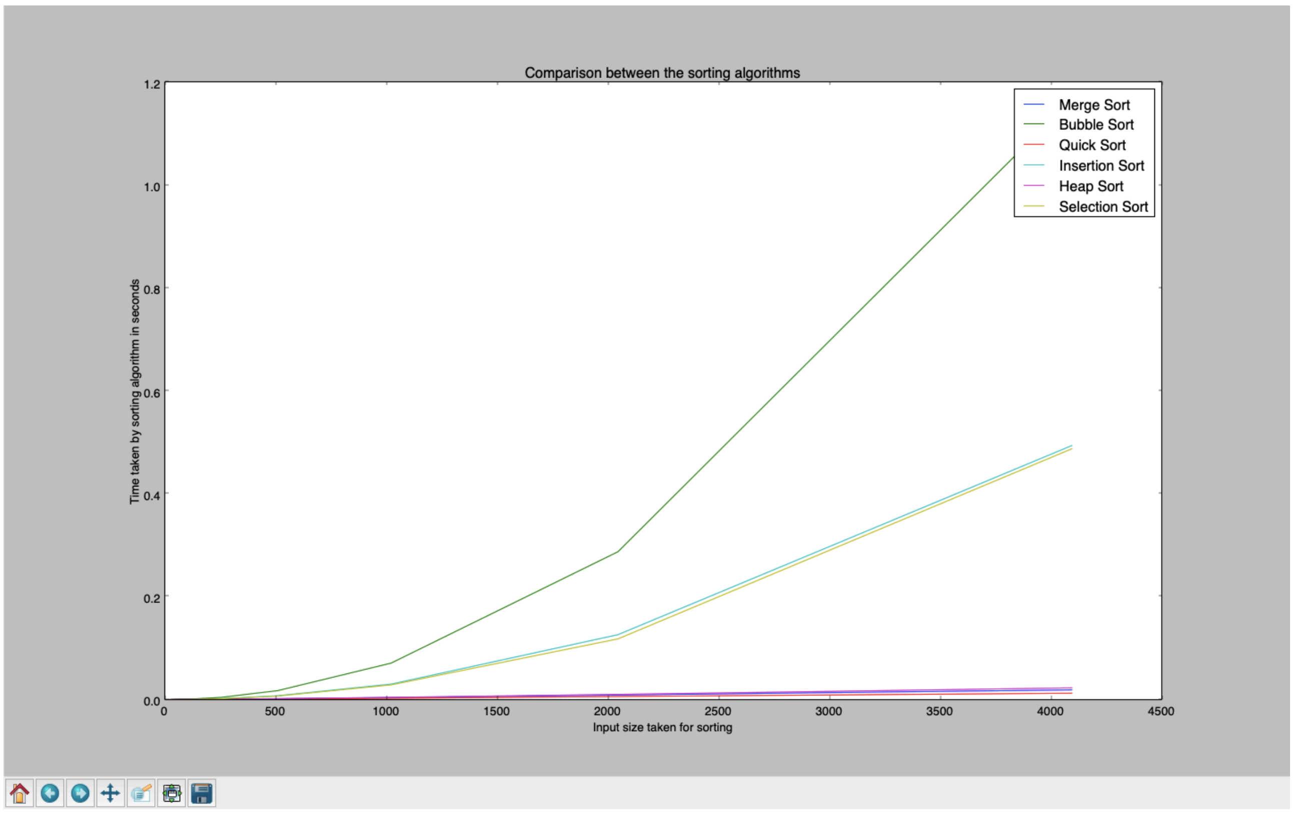
Here the provided elements are random i.e. not in any order of sorting.

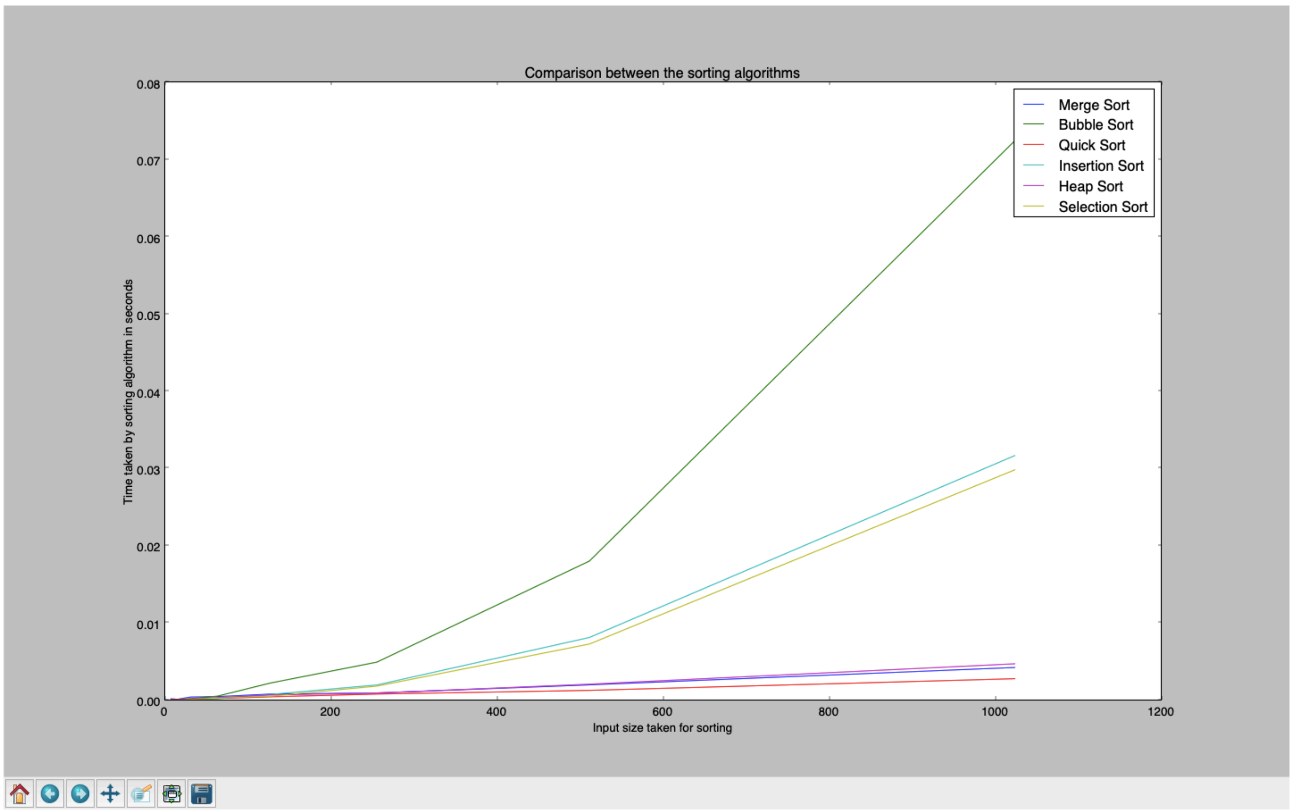


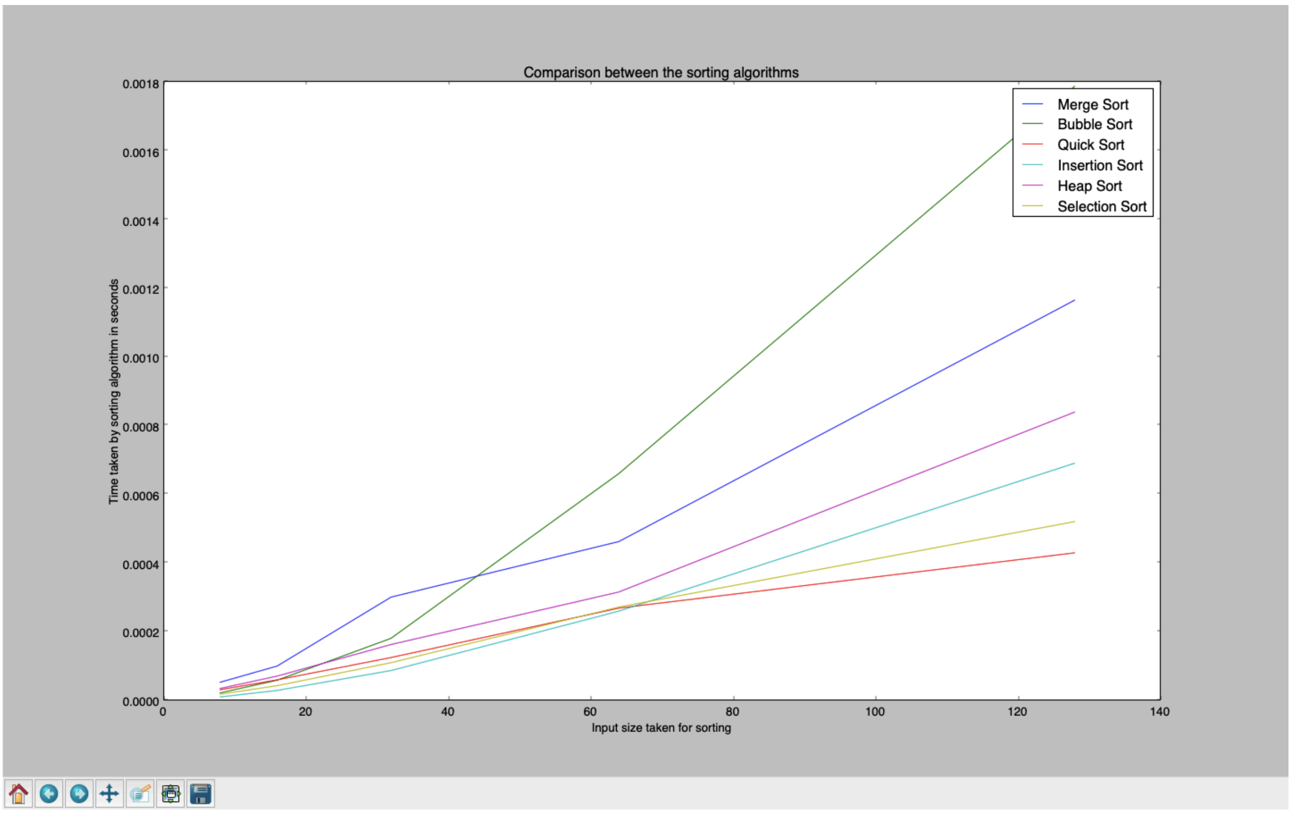
```
[Sayalis-Air:Daa Project sayalideshmukh$ python main.py
Enter 1: Enter 1 to provide inputs to array: 1
Enter the length of array and enter to provide elements: 5
123
45
12
56
9
sorted list obtained by merge sort
[9, 12, 45, 56, 123]
sorted list obtained by bubble sort
[9, 12, 45, 56, 123]
sorted list obtained by Quick sort
 [9, 12, 45, 56, 123]
sorted list obtained by selection sort
[9, 12, 45, 56, 123]
sorted list obtained by insertion sort
[9, 12, 45, 56, 123]
sorted list obtained by heap sort
[9, 12, 45, 56, 123]
Mergesort running time obtained in seconds is : 0.00014591217041
Bubblesort running time obtained in seconds is: 6.50882720947e-05
Quicksort running time obtained in seconds is: 4.6968460083e-05
Selectionsort running time obtained in seconds is: 1.4066696167e-05
Insertionsort running time obtained in seconds is: 6.91413879395e-06
Heapsort running time obtained in seconds is: 3.31401824951e-05
```

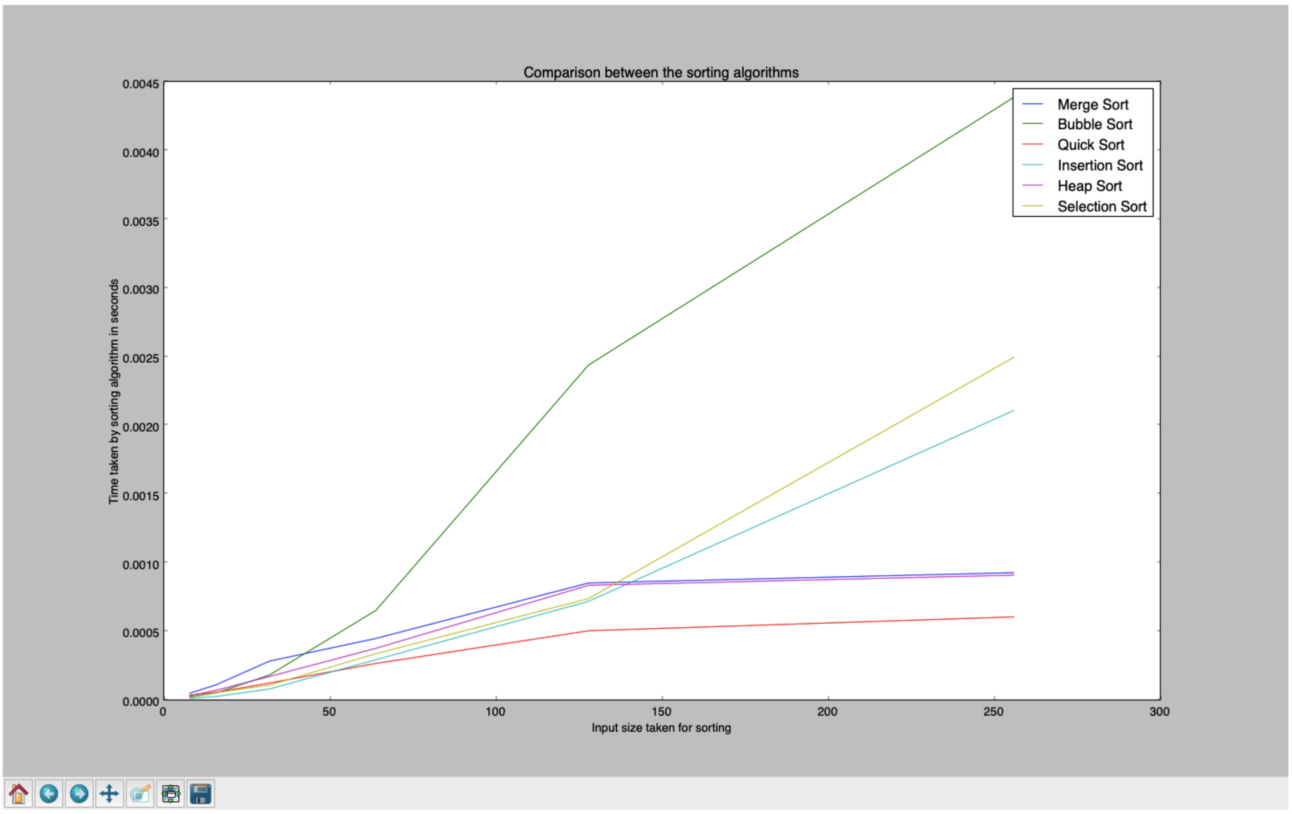
Analysis of all sorts:

It shows proper pictorial representation of sorting in terms of graph by considering the time in seconds.






Comparison between the sorting algorithms
Time taken by sorting algorithm in seconds
Input size taken for sorting
0.0000
0.0002
0.0004
0.0006
0.0008
0.0010
0.0012
0.0014
0.0016
0.0018
0
20
40
60
80
100
120
140
Merge Sort
Bubble Sort
Quick Sort
Insertion Sort
Heap Sort
Selection Sort


Comparison between the sorting algorithms
Time taken by sorting algorithm in seconds
Input size taken for sorting
0.0000
0.0005
0.0010
0.0015
0.0020
0.0025
0.0030
0.0035
0.0040
0.0045
0
50
100
150
200
250
300
Merge Sort
Bubble Sort
Quick Sort
Insertion Sort
Heap Sort
Selection Sort

Conclusion

Here time increases as no. of inputs provided. Among the different 6 sorting algorithms that are Merge sort, Heapsort, Quicksort, Insertion sort, Selection sort and Bubble sort we examine the proper running time complexity, their average, best and worst running time and according to finds out the best sorting algorithms depending on how the data is provided i.e. ascending order, descending order and random order. We have obtained bubble sort algorithm is slowest one algorithm and quick sort algorithm finds out to be fastest one.