CSE-5311-Project

Name: Kanuganti Sreeja
UTA ID: 1001868336

Implement and compare the following sorting algorithm :

- **Mergesort**
- **Heapsort**
- **Quicksort (Regular quick sort* and quick sort using 3 medians)**
- **Insertion sort**
- **Selection sort**
- **Bubble sort**

Sorting Algorithms: A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

A **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists

Data Structure used: For all the sorting methods, I used Arrays as the data structures.

Merge Sort: Merge Sort is a **Divide and Conquer** algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves.

Divide the unsorted list into 2 sub lists each time. Do this until each sub list has 1 element. Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2. Repeat the process till a single sorted list of obtained..

Merge Sort Algorithm Time Complexity:

Best Case: $\Omega(n(\log n))$

Average Case: $\Theta(n(\log n))$

Worst Case: $O(n(\log n))$

```python
import time

# left-leftarray
# right - rightArray
#method to perform merge
def merge(left, right):
    ans = []
    i, j = 0, 0

    while i < len(left) and j < len(right):

        if left[i] < right[j]:
            ans.append(left[i])
            i += 1
        else:
            ans.append(right[j])
            j += 1

    while i < len(left):
             ans.append(left[i])
             i += 1

    while j < len(right):
             ans.append(right[j])
             j += 1

    return ans
#method to perform Merge Sort
def mergeSort(mArray):
    length=len(mArray)
    if length == 1:
        return mArray

    mid = length // 2

    lP = mergeSort(mArray[0:mid])       #lP=left Partition, rP=right Partition
    rP = mergeSort(mArray[mid:])

    return merge(lP, rP)
```

In this method we have used two method. mergeSort() method is used to divide the array in two halves by calling mergeSort() method recursively. merge() method is used to join the array after sorting the list.

Run time for different test cases:

WORST CASE:

```
Enter the array size : 7

Enter the  Element 0 : 7

Enter the  Element 1 : 6

Enter the  Element 2 : 5

Enter the  Element 3 : 4

Enter the  Element 4 : 3

Enter the  Element 5 : 2

Enter the  Element 6 : 1

Results after applying Merge sort: [1, 2, 3, 4, 5, 6, 7]

Runtime of Merge sort = 1.6000000002236447e-05
```

Best Case:

```
Enter the array size :  7

Enter the  Element 0 : 1

Enter the  Element 1 : 2

Enter the  Element 2 : 3

Enter the  Element 3 : 4

Enter the  Element 4 : 5

Enter the  Element 5 : 6

Enter the  Element 6 : 7

Results after applying Merge sort: [1, 2, 3, 4, 5, 6, 7]

Runtime of Merge sort =  5.1400000000256796e-05
```

Average Case:

```
Enter the array size : 7

Enter the  Element 0 : 4

Enter the  Element 1 : 1

Enter the  Element 2 : 5

Enter the  Element 3 : 3

Enter the  Element 4 : 2

Enter the  Element 5 : 7

Enter the  Element 6 : 6

Results after applying Merge sort: [1, 2, 3, 4, 5, 6, 7]

Runtime of Merge sort = 1.5199999999992997e-05
```

Heap Sort: Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

Heap Sort Time Complexity:

Best Case: Ω(n(logn))

Average Case: Θ(n(logn))

Worst Case: O(n(logn))

```python
import time

#method to heapify
def heapify(array, n, i):
    greatest= i
    left = 2 * i + 1
    right = 2 * i + 2



    if left < n and array[greatest] < array[left]:
        greatest = left
    if right < n and array[greatest] < array[right]:
        greatest = right


    if greatest != i:
        #swap
        array[i], array[greatest] = array[greatest], array[i]

        # Heapify the root.
        heapify(array, n, greatest)


# method to perform heap Sort


def heapSort(heaparr):
    length = len(heaparr)

    # Build a maxheap.
    for i in range(length // 2 - 1, -1, -1):
        heapify(heaparr, length, i)
    for i in range(length - 1, 0, -1):
        #swap
        heaparr[i], heaparr[0] = heaparr[0], heaparr[i]
        heapify(heaparr, i, 0)
    return heaparr
```

Here we implemented two methods .

One is heapify() method which is used to build a max heap using the given values. heapSort() method is used to arrange the values in a sorted order using the heapify() method.

Run time for different test cases:

Worst Case:

Enter the size of the array : 7

Enter the Element 0 : 7

Enter the Element 1 : 6

```
Enter the  Element 2  : 5

Enter the  Element 3  : 4

Enter the  Element 4  : 3

Enter the  Element 5  : 2

Enter the  Element 6  : 1

Results after applying heap Sort: [1, 2, 3, 4, 5, 6, 7]

Runtime=2.0100000000411455e-05

Best case:

Enter the size of the array : 7

Enter the  Element 0  : 1

Enter the  Element 1  : 2

Enter the  Element 2  : 3

Enter the  Element 3  : 4

Enter the  Element 4  : 5

Enter the  Element 5  : 6

Enter the  Element 6  : 7

Results after applying heap Sort: [1, 2, 3, 4, 5, 6, 7]

Runtime= 2.2400000000644127e-05

Average Case:

Enter the size of the array : 7

Enter the  Element 0  : 5

Enter the  Element 1  : 1

Enter the  Element 2  : 2

Enter the  Element 3  : 7

Enter the  Element 4  : 6
Enter the  Element 5  : 3

Enter the  Element 6  : 4

Results after applying heap Sort: [1, 2, 3, 4, 5, 6, 7]

Runtime =  2.3399999999895726e-05
```

Quick Sort: QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

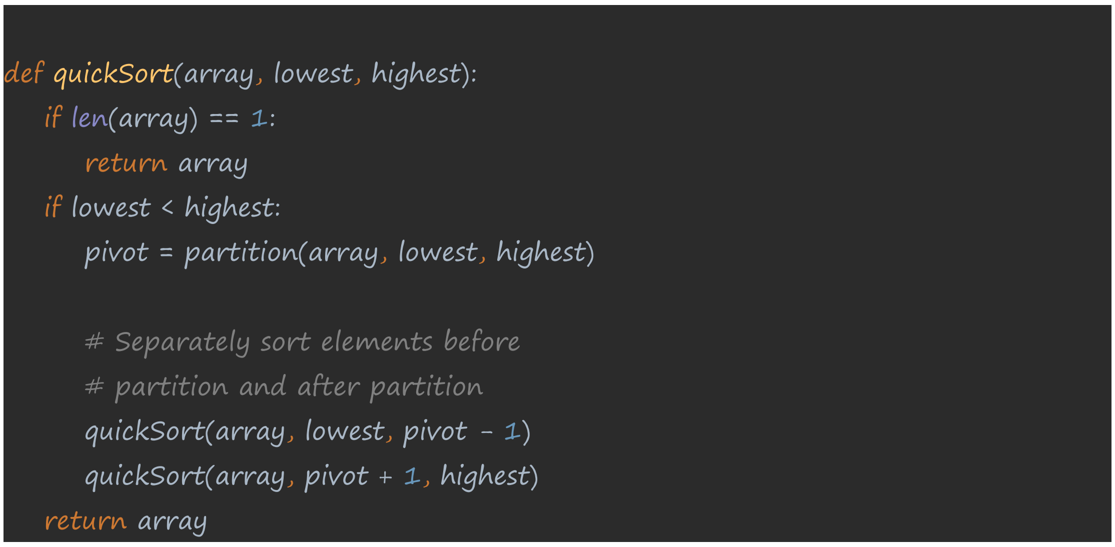
Quick Sort Time Complexity:

Best Case: Ω(n(logn))

Average Case: Θ(n(logn))

Worst Case: O(n^2)

```python
import time

#method to perform partition
def partition(inputArray, least, highest):            #l=low , h=high
    count = (least - 1)
    pivot = inputArray[highest]

    for j in range(least, highest):

        if inputArray[j] <= pivot:
            count += 1
            inputArray[count], inputArray[j] = inputArray[j], inputArray[count]

    inputArray[count + 1], inputArray[highest] = inputArray[highest],
inputArray[count + 1]
    return count + 1



# Method to do Quick sort
```



```
def quickSort(array, lowest, highest):
    if len(array) == 1:
        return array
    if lowest < highest:
        pivot = partition(array, lowest, highest)

        # Separately sort elements before
        # partition and after partition
        quickSort(array, lowest, pivot - 1)
        quickSort(array, pivot + 1, highest)
    return array
```

Here we used two methods.

partition() method is used to get the pivot value. quickSort() method is used to divide the array by calling quickSort recursively to sort the array and call the partition().

Run time for different test cases:

Worst Case:

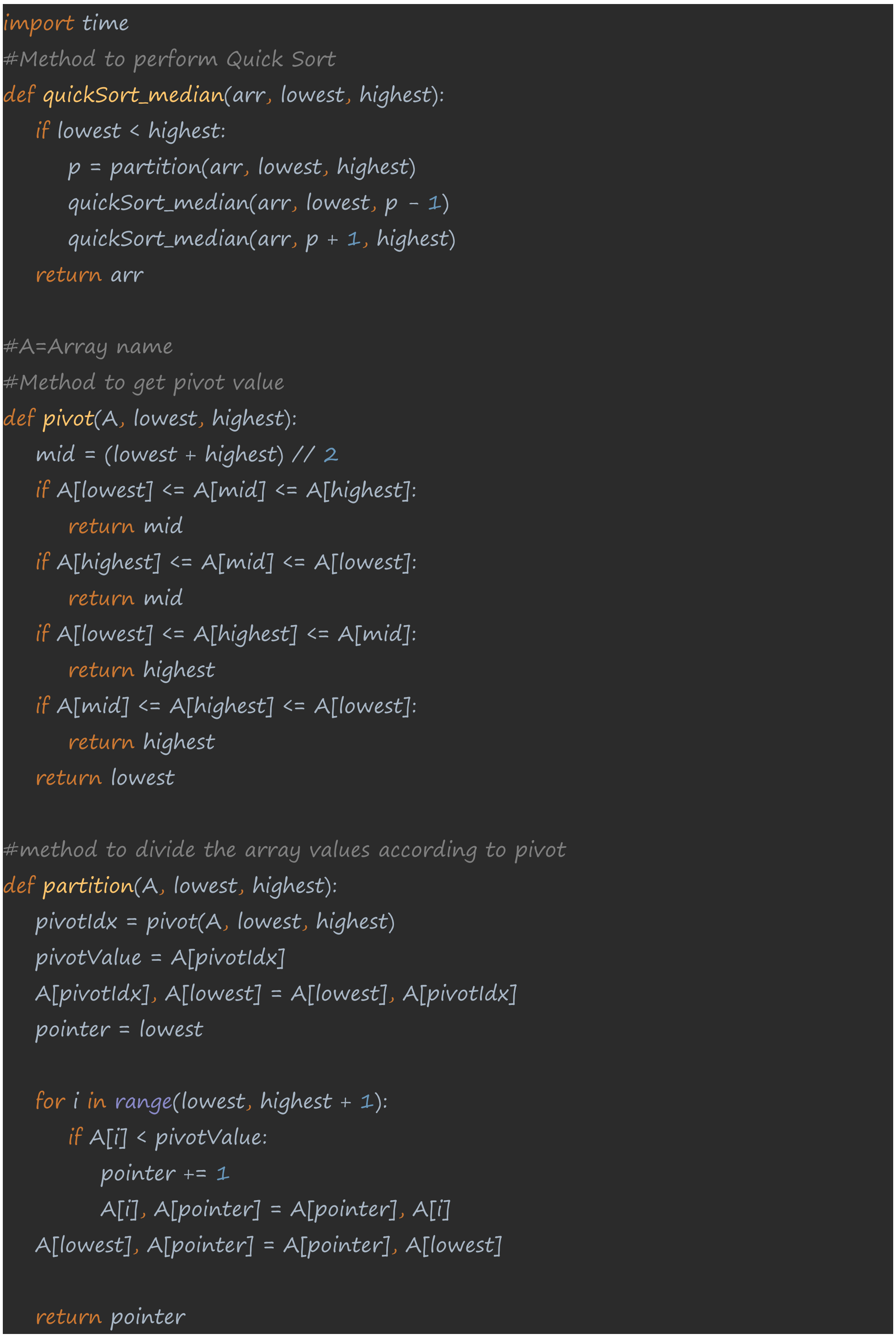
```
Enter the array size : 7

Enter the  Element 0 of List : 7

Enter the  Element 1 of List : 6

Enter the  Element 2 of List : 5

Enter the  Element 3 of List : 4

Enter the  Element 4 of List : 3

Enter the  Element 5 of List : 2

Enter the  Element 6 of List : 1

Results after performing Quick Sort: [1, 2, 3, 4, 5, 6, 7]

Quick sort Runtime= 1.5199999999992997e-05
```

Best Case:

```
Enter the array size : 7

Enter the  Element 0 of List : 1
Enter the  Element 1 of List : 2

Enter the  Element 2 of List : 3

Enter the  Element 3 of List : 4

Enter the  Element 4 of List : 5

Enter the  Element 5 of List : 6

Enter the  Element 6 of List : 7

Results after performing Quick Sort: [1, 2, 3, 4, 5, 6, 7]

Quick sort Runtime= 1.6099999999852344e-05

Average Case:

Enter the array size : 7

Enter the  Element 0 of List : 3

Enter the  Element 1 of List : 1

Enter the  Element 2 of List : 6

Enter the  Element 3 of List : 2

Enter the  Element 4 of List : 4

Enter the  Element 5 of List : 5

Enter the  Element 6 of List : 7

Results after performing Quick Sort: [1, 2, 3, 4, 5, 6, 7]

Quick sort Runtime=  1.9500000000505224e-05
```

Quick Sort Using Medians: The best case for quick sort is that if we could find the middle element. In this algorithm the pivot is picked as the median of the left most, centre and rightmost element in array. By selecting the median as the pivot we can, overcome the drawback in quick sort algorithm.

Here the drawback in the quick sort is ignored by selecting the pivot as median and hence the worst case in quick sort is converted to average case.



```
import time
#Method to perform Quick Sort
def quickSort_median(arr, lowest, highest):
    if lowest < highest:
        p = partition(arr, lowest, highest)
        quickSort_median(arr, lowest, p - 1)
        quickSort_median(arr, p + 1, highest)
    return arr

#A=Array name
#Method to get pivot value
def pivot(A, lowest, highest):
    mid = (lowest + highest) // 2
    if A[lowest] <= A[mid] <= A[highest]:
        return mid
    if A[highest] <= A[mid] <= A[lowest]:
        return mid
    if A[lowest] <= A[highest] <= A[mid]:
        return highest
    if A[mid] <= A[highest] <= A[lowest]:
        return highest
    return lowest

#method to divide the array values according to pivot
def partition(A, lowest, highest):
    pivotIdx = pivot(A, lowest, highest)
    pivotValue = A[pivotIdx]
    A[pivotIdx], A[lowest] = A[lowest], A[pivotIdx]
    pointer = lowest

    for i in range(lowest, highest + 1):
        if A[i] < pivotValue:
            pointer += 1
            A[i], A[pointer] = A[pointer], A[i]
    A[lowest], A[pointer] = A[pointer], A[lowest]

    return pointer
```

Here, we used three methods.

pivot() method is get the median value from three values which we have taken that is value at index 0 , value at index (size-1)and value at middl index . quickSort_median is used to divide the array into parts by calling the quickSort_median recursively. partition() method is used to get the pivot value.

Run time for different test cases:

Worst Case:

```
Enter the array size : 7

Enter the  Element 0 of List : 7

Enter the  Element 1 of List : 6

Enter the  Element 2 of List : 5

Enter the  Element 3 of List : 4

Enter the  Element 4 of List : 3

Enter the  Element 5 of List : 2

Enter the  Element 6 of List : 1

Results after applying quick sort using Median: [1, 2, 3, 4, 5, 6, 7]

Runtime=  1.3400000000274304e-05
```

Best Case:

```
Enter the array size : 7

Enter the  Element 0 of List : 1

Enter the  Element 1 of List : 2

Enter the  Element 2 of List : 3

Enter the  Element 3 of List : 4

Enter the  Element 4 of List : 5

Enter the  Element 5 of List : 6

Enter the  Element 6 of List : 7

Results after applying quick sort using Median: [1, 2, 3, 4, 5, 6, 7]

Runtime= 1.2500000000414957e-05
```

Average Case:

```
Enter the array size : 7

Enter the  Element 0 of List : 5

Enter the  Element 1 of List : 1

Enter the  Element 2 of List : 3

Enter the  Element 3 of List : 2

Enter the  Element 4 of List : 4

Enter the  Element 5 of List : 7

Enter the  Element 6 of List : 6

Results after applying quick sort using Median: [1, 2, 3, 4, 5, 6, 7]

Runtime= 2.0599999999149077e-05
```

Insertion Sort: Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

The input is an unsorted array. Find the correct position of xi in the list. Move other elements to Insert xi in the correct place

Insertion Sort Time Complexity:

Best Case: Ω(n)

Average Case: Θ(n^2)

Worst Case: O(n^2)

```python
import time

#insertion sort method
def insertionSort(iArray, n):   #name of the array

    for i in range(n):
        data = iArray[i]
        j = i-1
        while j >= 0 and data < arr[j]:
            iArray[j+1] = iArray[j]
            j -= 1
        iArray[j+1] = data
    return iArray
```

Here, we used insertionSort() method to sort the values.

Run time for different test cases:

Worst Case:

```
Enter the array size : 7

Enter the elements 0: 7

Enter the elements 1: 6

Enter the elements 2: 5

Enter the elements 3: 4

Enter the elements 4: 3

Enter the elements 5: 2

Enter the elements 6: 1

Results after applying insertion sort: [1, 2, 3, 4, 5, 6, 7]

 Runtime of insertion sort: 8.09999999873412e-06
```

Best Case:

```
Enter the array size : 7

Enter the elements 0: 1

Enter the elements 1: 2

Enter the elements 2: 3

Enter the elements 3: 4

Enter the elements 4: 5

Enter the elements 5: 6

Enter the elements 6: 7

Results after applying insertion sort: [1, 2, 3, 4, 5, 6, 7]
```

```
Runtime of insertion sort: 7.700000001165108e-06

Average Case:

Enter the array size : 7

Enter the elements 0: 5

Enter the elements 1: 1

Enter the elements 2: 2

Enter the elements 3: 7

Enter the elements 4: 6

Enter the elements 5: 3

Enter the elements 6: 4

Results after applying insertion sort: [1, 2, 3, 4, 5, 6, 7]

Runtime of insertion sort: 6.8000000013057615e-06
```

Selection Sort: Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

Initially, the sorted part is empty, and the unsorted part is the entire list. The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

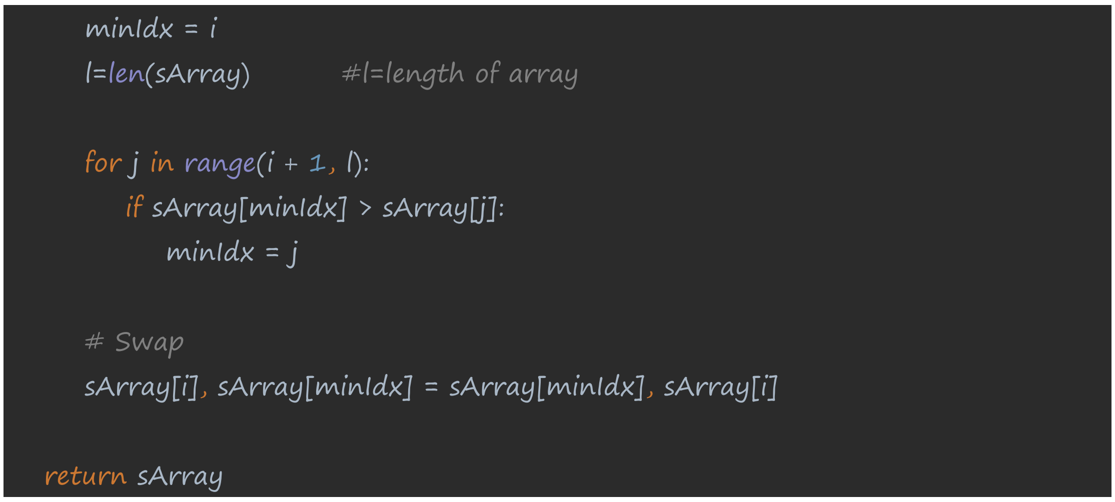
Selection Sort Time Complexity :

Best Case: Ω(n^2)

Average Case: Θ(n^2)

Worst Case: O(n^2)

```python
import time


# selection sort method
def selectionSort(sArray, length):  #sArray= name of the array
    for i in range(length):
```



```python
        minIdx = i
        l=len(sArray)          #l=length of array

        for j in range(i + 1, l):
            if sArray[minIdx] > sArray[j]:
                minIdx = j

        # Swap
        sArray[i], sArray[minIdx] = sArray[minIdx], sArray[i]

    return sArray
```

Here, we used selectionSort() method to sort the values.

Run time for different test cases:

Worst Case:

```
Enter the  Element 0  : 7

Enter the  Element 1  : 6

Enter the  Element 2  : 5

Enter the  Element 3  : 4

Enter the  Element 4  : 3

Enter the  Element 5  : 2

Enter the  Element 6  : 1

Results after applying selection sort: [1, 2, 3, 4, 5, 6, 7]

Run time of selection sort = 9.399999999715192e-06
```

Best Case:

```
Enter the size of the array: 7

Enter the  Element 0  : 1

Enter the  Element 1  : 2

Enter the  Element 2  : 3

Enter the  Element 3  : 4
Enter the  Element 4  : 5

Enter the  Element 5  : 6

Enter the  Element 6  : 7

Results after applying selection sort: [1, 2, 3, 4, 5, 6, 7]

Run time of selection sort = 1.2999999999152578e-05
```

Average Case:

```
Enter the size of the array: 7

Enter the  Element 0  : 4

Enter the  Element 1  : 1

Enter the  Element 2  : 3

Enter the  Element 3  : 6

Enter the  Element 4  : 7

Enter the  Element 5  : 2

Enter the  Element 6  : 5

Results after applying selection sort: [1, 2, 3, 4, 5, 6, 7]

Run time of selection sort = 9.90000000022917e-06
```

Bubble Sort: Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

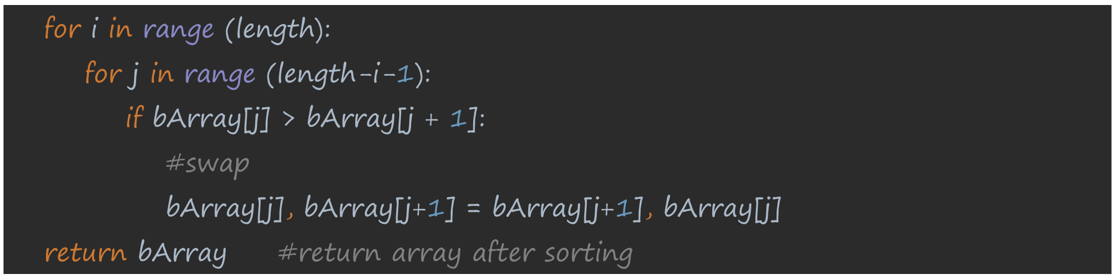
Bubble Sort Time Complexity:

Best Case: Ω(n)

Average Case: Θ(n^2)

Worst Case: O(n^2)

```python
import time

#bubble Sort method
def bubbleSort(bArray, length):
```



```
    for i in range (length):
        for j in range (length-i-1):
            if bArray[j] > bArray[j + 1]:
                #swap
                bArray[j], bArray[j+1] = bArray[j+1], bArray[j]
    return bArray     #return array after sorting
```

Run time for different test cases:

Worst Case:

```
Enter the size of the array: 7

Enter the  Elements 0 : 7

Enter the  Elements 1 : 6

Enter the  Elements 2 : 5

Enter the  Elements 3 : 4

Enter the  Elements 4 : 3

Enter the  Elements 5 : 2

Enter the  Elements 6 : 1

Results after performing Bubble sort: [1, 2, 3, 4, 5, 6, 7]

Bubble sort Runtime= 9.899999999340992e-06

Enter the size of the array: 7

Enter the  Elements 0 : 1

Enter the  Elements 1 : 2

Enter the  Elements 2 : 3

Enter the  Elements 3 : 4

Enter the  Elements 4 : 5

Enter the  Elements 5 : 6

Enter the  Elements 6 : 7

Results after performing Bubble sort: [1, 2, 3, 4, 5, 6, 7]
```

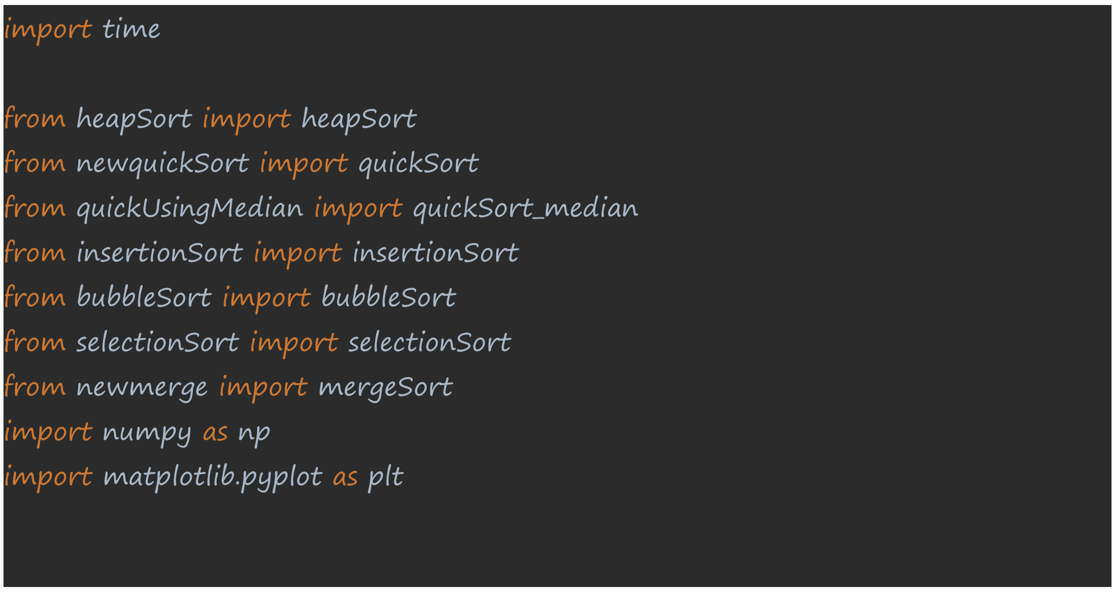
```
Bubble sort Runtime= 1.0899999999948077e-05

Average Case:

Enter the size of the array: 7

Enter the  Elements 0 : 5

Enter the  Elements 1 : 1

Enter the  Elements 2 : 3

Enter the  Elements 3 : 2

Enter the  Elements 4 : 4

Enter the  Elements 5 : 7

Enter the  Elements 6 : 6

Results after performing Bubble sort: [1, 2, 3, 4, 5, 6, 7]

Bubble sort Runtime=  8.90000000097757e-06
```

newmain.py : In this file we imported all the sorting methods from other files. This is the main file where we can execute all the sorting Algorithm.

main() method is used to call all the sorting algorithms

It takes the input one time and gives the sorted results and running time of all the algorithms.

Here Graph is also plotted to compare the run time of the algorithms in different cases.

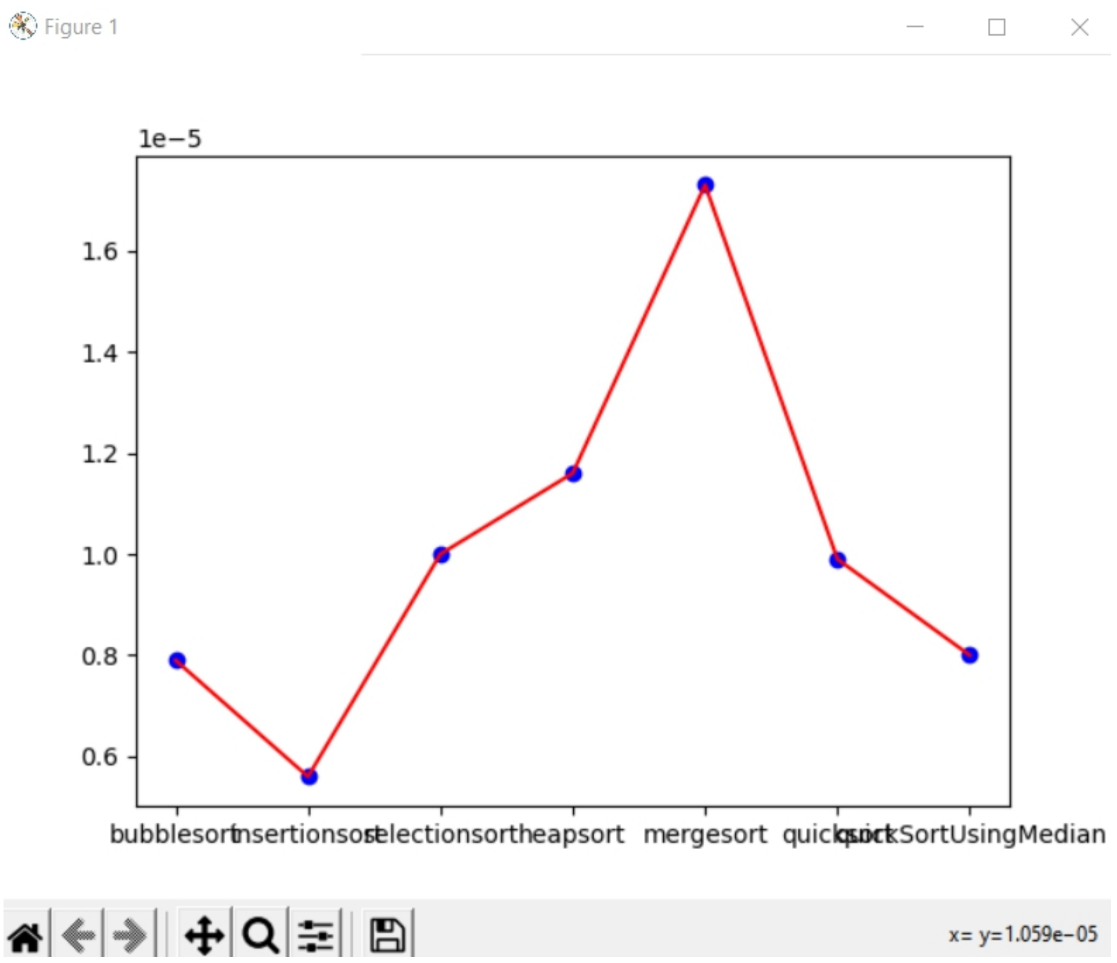


```python
import time

from heapSort import heapSort
from newquickSort import quickSort
from quickUsingMedian import quickSort_median
from insertionSort import insertionSort
from bubbleSort import bubbleSort
from selectionSort import selectionSort
from newmerge import mergeSort
import numpy as np
import matplotlib.pyplot as plt
```

```python
def main():
    # getting the input from the user
    globalArr = []
    size = int(input("Enter the size of the array : "))

    for i in range(size):
        value = int (input("Enter the  Element %d of array : " % i))
        globalArr.append(value)

    print(globalArr)
    arrayForSelectionSort = globalArr.copy()
    selection_start = time.perf_counter()
    r = selectionSort(arrayForSelectionSort, size)
    selection_time = time.perf_counter() - selection_start
    print("Results after performing Selection sort:", r)
    print("Run time of Selection Sort =", selection_time)

    arrayForbubbleSort = globalArr.copy()
    bubble_start = time.perf_counter()
    r = bubbleSort(arrayForbubbleSort, size)
    bubble_time = time.perf_counter() - bubble_start
    print("Results after performing Bubble sort:", r)
    print("Runtime of Bubble sort =", bubble_time)

    arrayForMergeSort = globalArr.copy()
    merge_start = time.perf_counter()
    r = mergeSort(arrayForMergeSort)
    merge_time = time.perf_counter() - merge_start
    print("Results after performing Merge sort:", r)
    print("Runtime of Merge sort=", merge_time)

    arrayForInsertionSort = globalArr.copy()
    insertion_start = time.perf_counter()
    r = insertionSort(arrayForInsertionSort, size)
    insertion_time = time.perf_counter() - insertion_start
    print("Results after performing Insertion sort:", r)
    print("Runtime of Insertion sort=", insertion_time)

    arrayForQuickSort = globalArr.copy()
    quick_start = time.perf_counter()
    r = quickSort(arrayForQuickSort, 0, len(arrayForQuickSort) - 1)
    quick_time = time.perf_counter() - quick_start
    print("Results after performing Quick sort:", r)
    print("Runtime of Quick sort=", quick_time)

    arrayForQuickSortMedian = globalArr.copy()
    quickstart_UsingMedain = time.perf_counter()
    r = quickSort_median(arrayForQuickSortMedian, 0, len(arrayForQuickSortMedian)
- 1)
    quickMedian_time = time.perf_counter() - quickstart_UsingMedain
    print("Result after performing Quick sort using Median:", r)
    print("Runtime of Quick sort using Median=", quickMedian_time)

    arrayForheapSort = globalArr.copy()
    heap_start = time.perf_counter()
    r = heapSort(arrayForheapSort)
    heap_time = time.perf_counter() - heap_start
    print("Results after performing Heap sort:", r)
    print("Runtime of Heap sort=", heap_time)

    x = [bubble_time, insertion_time, selection_time, heap_time, merge_time,
quick_time, quickMedian_time]
    plt.xticks(np.arange(7), (
    'bubblesort', 'insertionsort', 'selectionsort', 'heapsort', 'mergesort', 'quicksort',
'quickSortUsingMedian'))
    plt.plot(x, 'bo', x, 'r')
    plt.show()


main()
```

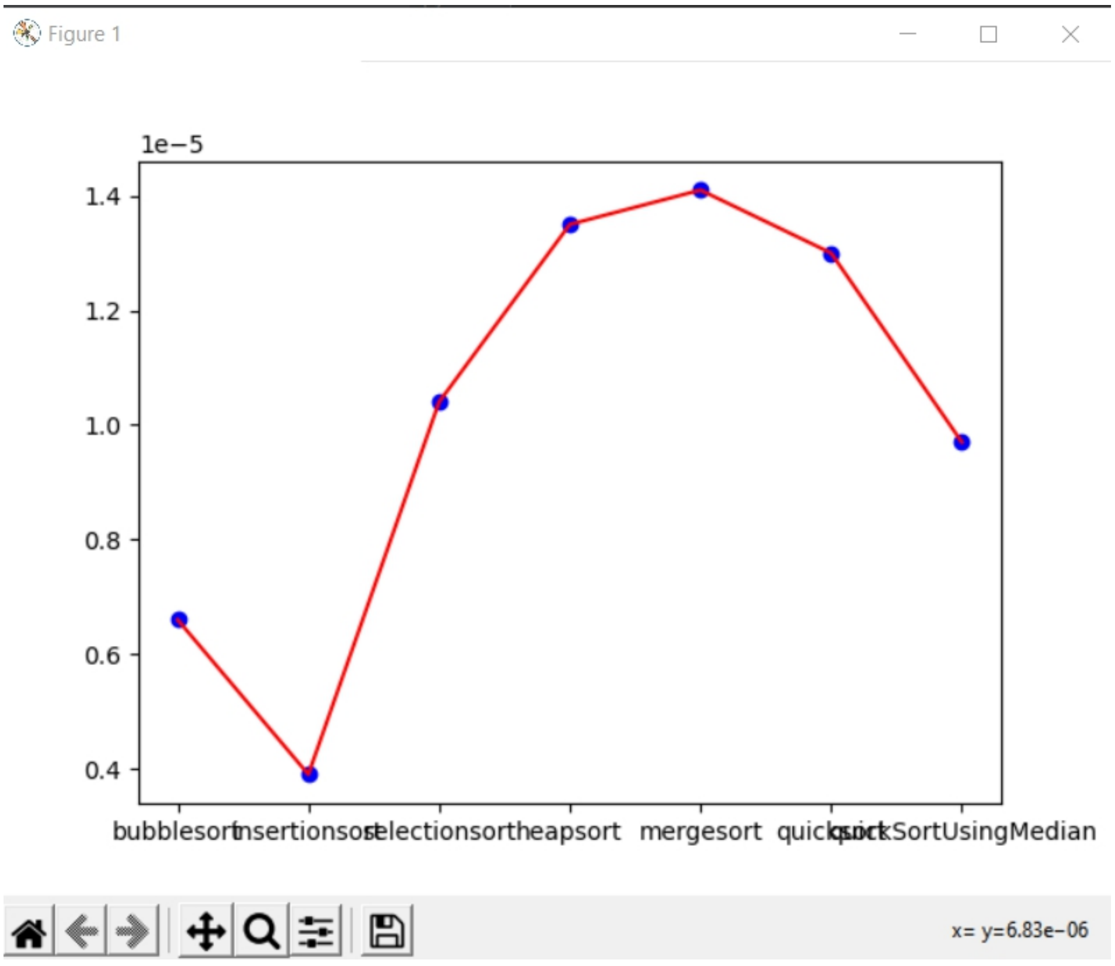
```
Enter the size of the array : 5

Enter the  Element 0 of array : 4

Enter the  Element 1 of array : 67

Enter the  Element 2 of array : 21

Enter the  Element 3 of array : 45

Enter the  Element 4 of array : 8
```

```
[4, 67, 21, 45, 8]

Results after performing Selection sort: [4, 8, 21, 45, 67]

Run time of Selection Sort = 9.999999999621423e-06

Results after performing Bubble sort: [4, 8, 21, 45, 67]

Runtime of Bubble sort = 7.899999999949614e-06

Results after performing Merge sort: [4, 8, 21, 45, 67]

Runtime of Merge sort= 1.7299999999664806e-05

Results after performing Insertion sort: [4, 8, 21, 45, 67]

Runtime of Insertion sort= 5.600000001493299e-06

Results after performing Quick sort: [4, 8, 21, 45, 67]

Runtime of Quick sort= 9.899999998452813e-06

Result after performing Quick sort using Median: [4, 8, 21, 45, 67]

Runtime of Quick sort using Median= 8.000000001118224e-06

Results after performing Heap sort: [4, 8, 21, 45, 67]

Runtime of Heap sort= 1.160000000055561e-05
```



```
Enter the size of the array : 5

Enter the  Element 0 of array : 1

Enter the  Element 1 of array : 2

Enter the  Element 2 of array : 3

Enter the  Element 3 of array : 4

Enter the  Element 4 of array : 5

[1, 2, 3, 4, 5]

Results after performing Selection sort: [1, 2, 3, 4, 5]

Run time of Selection Sort = 1.0400000000743148e-05

Results after performing Bubble sort: [1, 2, 3, 4, 5]

Runtime of Bubble sort = 6.600000006073969e-06

Results after performing Merge sort: [1, 2, 3, 4, 5]
```

```
Runtime of Merge sort= 1.4100000001349144e-05

Results after performing Insertion sort: [1, 2, 3, 4, 5]

Runtime of Insertion sort= 3.900000002943216e-06

Results after performing Quick sort: [1, 2, 3, 4, 5]

Runtime of Quick sort= 1.2999999995599865e-05

Result after performing Quick sort using Median: [1, 2, 3, 4, 5]

Runtime of Quick sort using Median= 9.699999999668307e-06

Results after performing Heap sort: [1, 2, 3, 4, 5]
```



```
Enter the size of the array : 40

Enter the  Element 0 of array : 6

Enter the  Element 1 of array : 12

Enter the  Element 2 of array : 98
Enter the  Element 3 of array : 534

Enter the  Element 4 of array : 234

Enter the  Element 5 of array : 61

Enter the  Element 6 of array : 90

Enter the  Element 7 of array : 29

Enter the  Element 8 of array : 11

Enter the  Element 9 of array : 456

Enter the  Element 10 of array : 75

Enter the  Element 11 of array : 13

Enter the  Element 12 of array : 42

Enter the  Element 13 of array : 78

Enter the  Element 14 of array : 19

Enter the  Element 15 of array : 33

Enter the  Element 16 of array : 26

Enter the  Element 17 of array : 84

Enter the  Element 18 of array : 93

Enter the  Element 19 of array : 25

Enter the  Element 20 of array : 71

Enter the  Element 21 of array : 43

Enter the  Element 22 of array : 81

Enter the  Element 23 of array : 22

Enter the  Element 24 of array : 56

Enter the  Element 25 of array : 39

Enter the  Element 26 of array : 11

Enter the  Element 27 of array : 77
Enter the  Element 28 of array : 239

Enter the  Element 29 of array : 871

Enter the  Element 30 of array : 123

Enter the  Element 31 of array : 44

Enter the  Element 32 of array : 64

Enter the  Element 33 of array : 37

Enter the  Element 34 of array : 114

Enter the  Element 35 of array : 121

Enter the  Element 36 of array : 2

Enter the  Element 37 of array : 1

Enter the  Element 38 of array : 5

Enter the  Element 39 of array : 9

[6, 12, 98, 534, 234, 61, 90, 29, 11, 456, 75, 13, 42, 78, 19, 33, 26, 84, 93, 25, 71, 43,
81, 22, 56, 39, 11, 77, 239, 871, 123, 44, 64, 37, 114, 121, 2, 1, 5, 9]

Results after performing Selection sort: [1, 2, 5, 6, 9, 11, 11, 12, 13, 19, 22, 25, 26, 29,
33, 37, 39, 42, 43, 44, 56, 61, 64, 71, 75, 77, 78, 81, 84, 90, 93, 98, 114, 121, 123, 234,
239, 456, 534, 871]

Run time of Selection Sort = 5.269999999997685154e-05

Results after performing Bubble sort: [1, 2, 5, 6, 9, 11, 11, 12, 13, 19, 22, 25, 26, 29,
33, 37, 39, 42, 43, 44, 56, 61, 64, 71, 75, 77, 78, 81, 84, 90, 93, 98, 114, 121, 123, 234,
239, 456, 534, 871]

Runtime of Bubble sort = 8.910000001094431e-05

Results after performing Merge sort: [1, 2, 5, 6, 9, 11, 11, 12, 13, 19, 22, 25, 26, 29, 33,
37, 39, 42, 43, 44, 56, 61, 64, 71, 75, 77, 78, 81, 84, 90, 93, 98, 114, 121, 123, 234,
239, 456, 534, 871]

Runtime of Merge sort= 6.84000000035212e-05

Results after performing Insertion sort: [1, 2, 5, 6, 9, 11, 11, 12, 13, 19, 22, 25, 26, 29,
33, 37, 39, 42, 43, 44, 56, 61, 64, 71, 75, 77, 78, 81, 84, 90, 93, 98, 114, 121, 123, 234,
239, 456, 534, 871]

Runtime of Insertion sort= 4.8599999999510146e-05
```

Results after performing Quick sort: [1, 2, 5, 6, 9, 11, 11, 12, 13, 19, 22, 25, 26, 29, 33, 37, 39, 42, 43, 44, 56, 61, 64, 71, 75, 77, 78, 81, 84, 90, 93, 98, 114, 121, 123, 234, 239, 456, 534, 871]

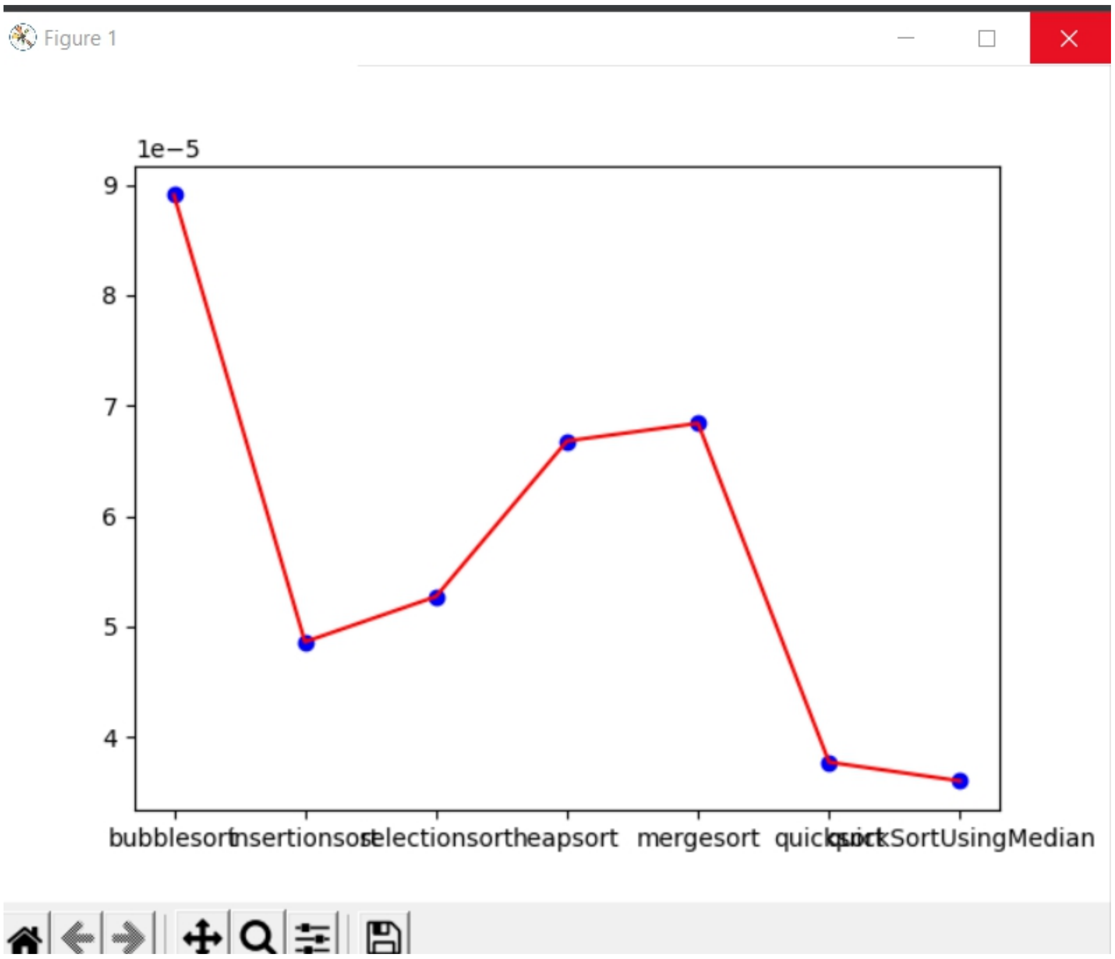
Runtime of Quick sort= 3.7700000007134804e-05

Result after performing Quick sort using Median: [1, 2, 5, 6, 9, 11, 11, 12, 13, 19, 22, 25, 26, 29, 33, 37, 39, 42, 43, 44, 56, 61, 64, 71, 75, 77, 78, 81, 84, 90, 93, 98, 114, 121, 123, 234, 239, 456, 534, 871]

Runtime of Quick sort using Median= 3.600000000858472e-05

Results after performing Heap sort: [1, 2, 5, 6, 9, 11, 11, 12, 13, 19, 22, 25, 26, 29, 33, 37, 39, 42, 43, 44, 56, 61, 64, 71, 75, 77, 78, 81, 84, 90, 93, 98, 114, 121, 123, 234, 239, 456, 534, 871]

Runtime of Heap sort= 6.679999999990343e-05



Conclusion:

From the above graphs, we observed that,

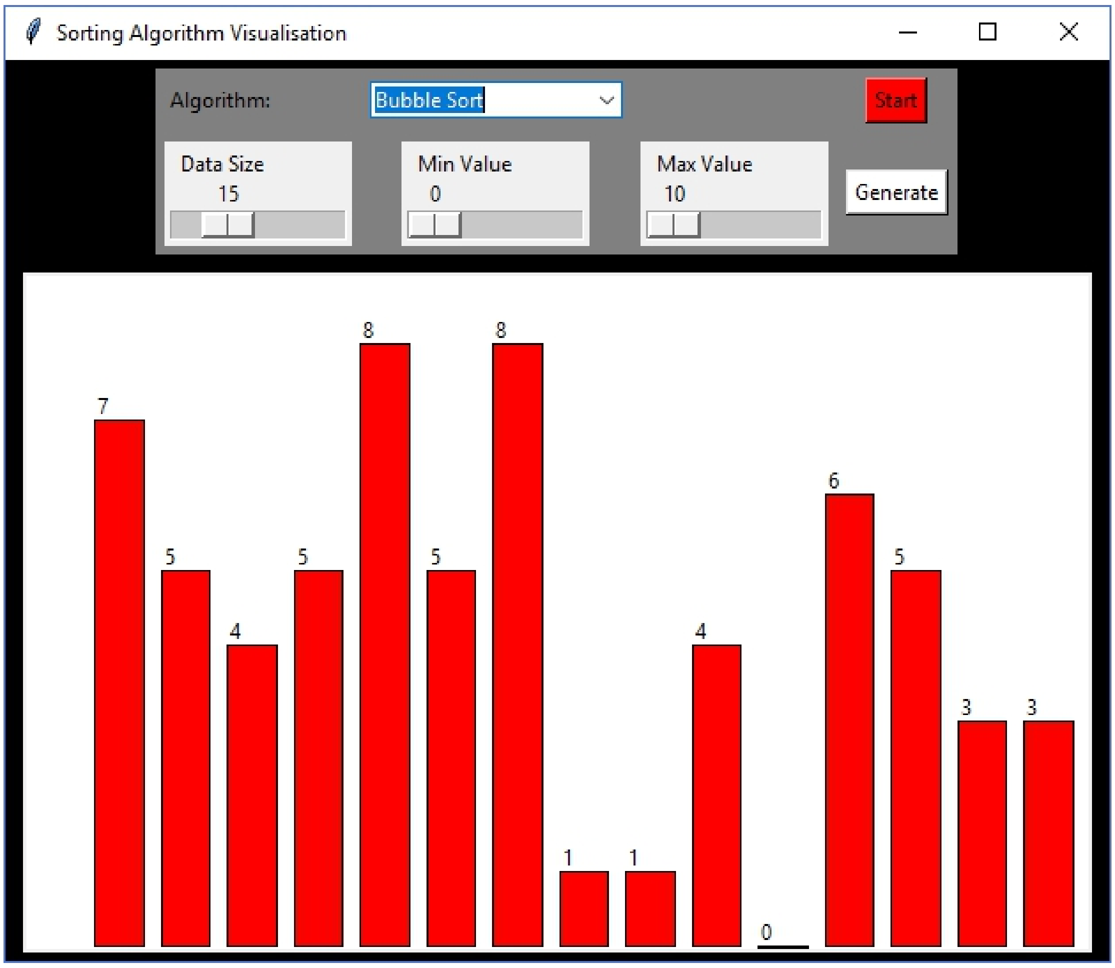
for smaller data Insertion sort is more efficient.

For larger data quick Sort Using Median is more efficient.

Time Complexity of all the above sorting Algorithms are given in the following table in Best Case, Average Case and Worst Case.

Name Algorithm	Time Complexity of Algorithm		
	Best	Average	Worst
Merge Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Heap Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)

GUI Pictures:

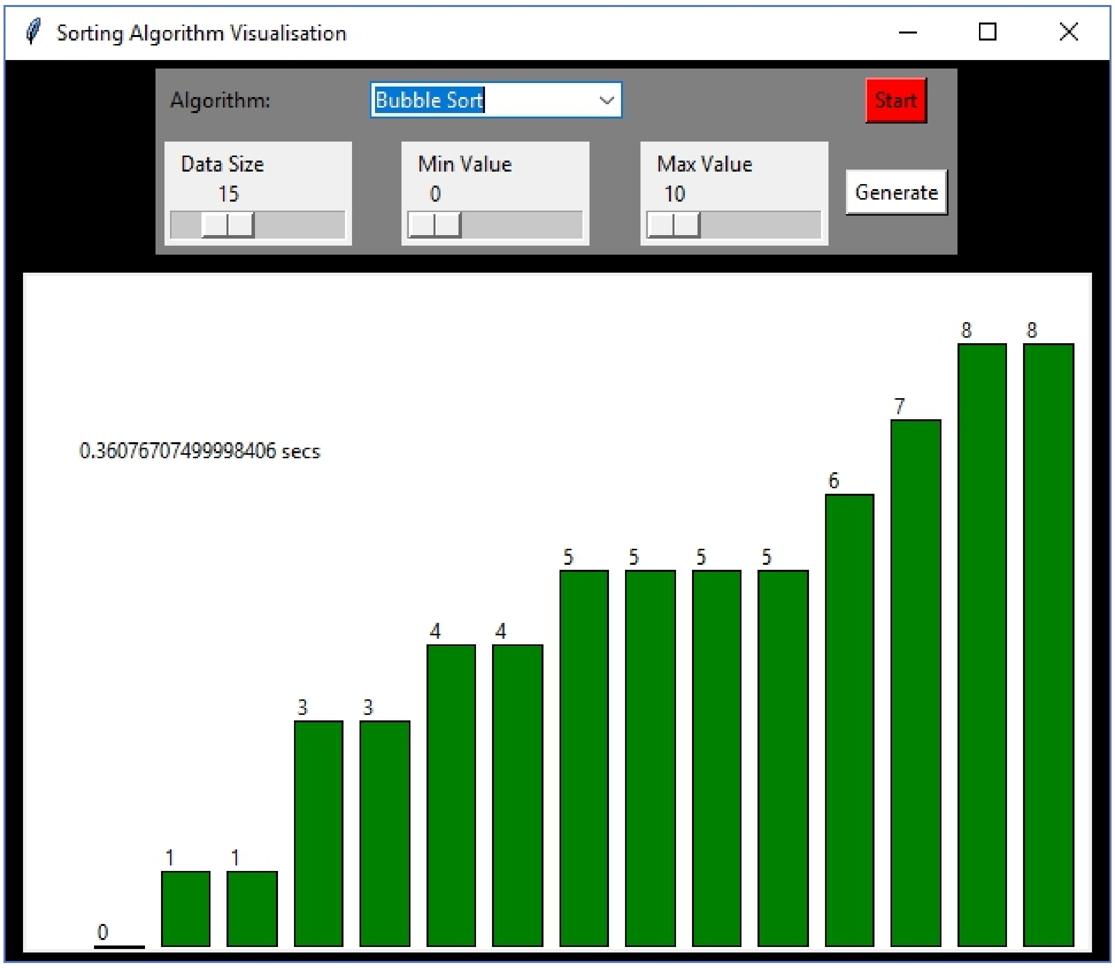

Sorting Algorithm Visualisation

Algorithm: Bubble Sort

Data Size 15

Min Value 0

Max Value 10



Here, I have built a simple GUI using tkinter package which is a Python interface to take the number of inputs, maximum value and minimum value.

First graph indicates values before sorting.

Second graph indicates values after sorting.

Data Size: Here we give the number of inputs.

Min Value: It takes the minimum value of the input data set.

Max Value: It takes the maximum value of the input data set.

Start: Start Button is used to start the sorting algorithm and to sort out.

Generate : Generate Button is used to generate the values between minimum and maximum values of the data set.