A report on comparison of different sorting algorithms.

By:

Vishwanth Reddy Muchantula

TABLE OF CONTENTS

CHAPTER-1
INTRODUCTION

1.1 What is sorting

In computer science, a sorting algorithm is an algorithm that arranges elements of a collection in a certain order. Numerical order and Lexicographical order are most frequently used orders.

CHAPTER-2
BUBBLE SORT

2.1 Introduction

The name bubble sort has its name from the way this sorting technique is implemented. It follows a process where adjacent elements are compared and swapped if they are in the wrong order. Bubblesort, is also known as a sinking sort.

2.2 How it is implemented

```
while (n != 0):
    for i in range(n):
        if arr[i] > arr[i + 1]:
            arr[i], arr[i + 1] = arr[i + 1], arr[i]
    n = n - 1
```

- Initially we start a while loop that runs the algorithm n-1 times.
- Then, start the inner loop that compares all the values has range equal to n length of the list.
- The if statement to checks if the value on the value on the right is great than value on left, If the condition is not false swap the values.
- This process continues and So, with each outer loop iteration the number of elements that needs to be included in the inner loop iteration decreases giving us

2.3 Run time comparisons.

Average case: $O(n^2)$

Worst case: $O(n^2)$

Best Case: O(n) and the best case would be if the list were already sorted

Space Complexity: O(1)

CHAPTER-3

SELECTION SORT

3.1 Introduction

Selection sort is an in-place sorting algorithm which traverses through the list to find the smallest element and places it in the beginning of the list. The selection sort algorithm recursively finds the smallest element from unordered part and putting it at the first position.

3.2 How it is implemented

```
for i in range(len(A)):
    min = i
    for j in range(i + 1, len(A)):
        if A[j] < A[min]:
            min = j
    if min != i:
        tmp = A[min]
        A[min] = A[i]
        A[i] = tmp
```

The algorithm maintains two sub arrays in each list.

1) The sublist which is already sorted.
2) Remaining sublist which is unsorted.

In every step of selection sort algorithm, the least element from the unsorted sublist is picked and moved to the sorted sublist.

- In the implementation, the outer loop iterates over the entire list starting from the first element.

- Initially it considers first element as smallest and it swaps this smallest element
- index with the index of a smaller element if found any.
- Once the outer loop selects the smallest element, the inner loop iterates through the entire list from the next element to validate if the selected element is smallest or not.
- If found any smallest element, it swaps them. Else, it will exit the inner loop and select the next element as smallest using outer loop.
- Hence, with each outer loop iteration, we have a sorted list being formed on the left side and the un-sorted list remains on the right

3.3 Run time comparisons.

Average case: $O(n^2)$

Worst case: $O(n^2)$

Best Case: $O(n^2)$

Space Complexity: O (1)

CHAPTER-4

INSERTION SORT

4.1 Introduction

Insertion sort is a simple sorting technique which works by placing a element at its correct position in every iteration.

4.2 How it is implemented

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

In the implementation,

- Pick next element
- This element is compared with values in the other sorted list.
- Find the correct position to be inserted and then Insert the value
- Repeat until list is sorted

4.3 Run time comparisons.

Average case: $O(n^2)$

Worst case: $O(n^2)$

Best Case: O(n)

Space Complexity: O (1)

CHAPTER-5

MERGE SORT

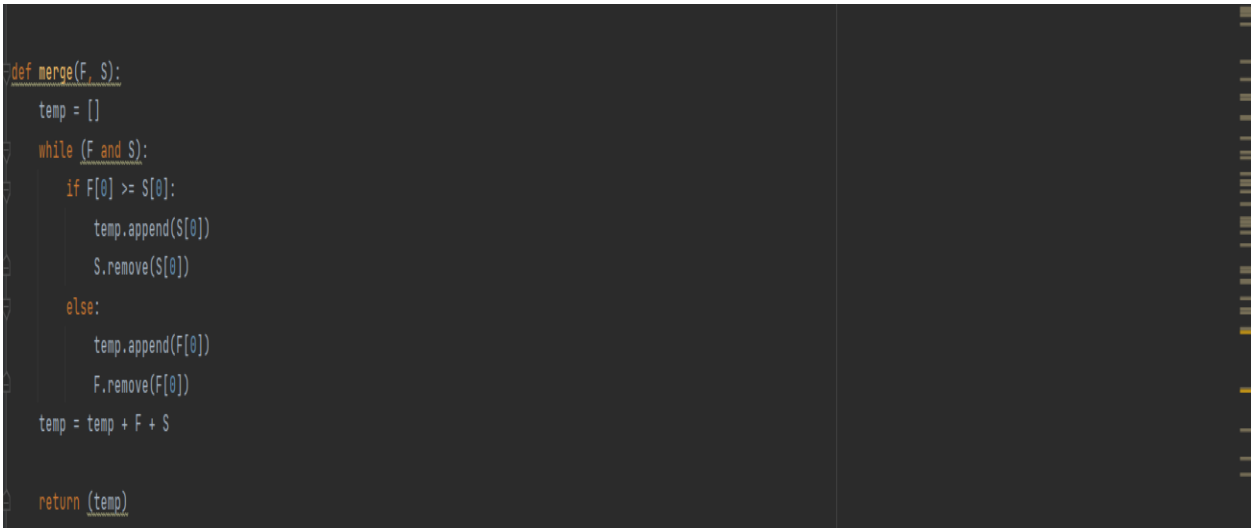
5.1 Introduction

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-arrays till till sublist consists of a single component and then merging back those subarrays in a manner that results into a sorted list.

5.2 How it is implemented

```
if len(arr) == 1:
    return arr
arr = mergearray(arr)
```

```
def mergearray(arr):
    if len(arr) == 1:
        return arr
    n = len(arr)

    F = arr[0:(n // 2)]
    S = arr[(n // 2)::]
    F = mergearray(F)
    S = mergearray(S)
    return merge(F, S)
```



```
def merge(F, S):
    temp = []
    while (F and S):
        if F[0] >= S[0]:
            temp.append(S[0])
            S.remove(S[0])
        else:
            temp.append(F[0])
            F.remove(F[0])
    temp = temp + F + S

    return (temp)
```

In the implementation,

- Initially, the merge sort function is called which keeps on dividing the list into equal sublists until it can no more be divided
- The merge function is used for merging two `sublists`. The merge (F,S) is a key process that assumes that F and S are sorted and merges the two sorted sub-arrays into one. Once the array contains only one element it sorts and compares the elements of the two smaller arrays and forms a new list which is sorted.

5.3 Run time comparisons.

Average case: O (n log(n))

Worst case: O(n^2). The worst case occurs when the picked pivot is always an extreme (smallest or largest)

Best Case O (n log(n)). The best case occurs when the partition process always picks the middle element as pivot.

Space Complexity: O (1)

CHAPTER-6
QUICK SORT

6.1 Introduction

Quicksort is a divide-and-conquer method for sorting. It works by *partitioning* a list into two parts, then sorting the parts independently.

6.2 How it is implemented

Quicksort algorithm is a sorting algorithm based on the divide and conquer approach where
An array is divided into subarrays by selecting a pivot element
While dividing the list, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
The left and right subarrays are recursively divided using the same approach. This process continues until each subarray contains a single element.

```python
def quick_reg(A, low, high):  # first element as pivot
    if low < high:
        j = partition(A, low, high)
        quick_reg(A, low, j - 1)
        quick_reg(A, j + 1, high)
    return A


def partition(arr, low, high):
    pivot = arr[low]
    i = low
    j = high
    while i < j:
        while i < (len(arr) - 1) and arr[i + 1] < pivot:
            i = i + 1
        while arr[j] > pivot:
            j = j - 1
        if i < j:
            tmp = arr[i + 1]
            arr[i + 1] = arr[j]
            arr[j] = tmp
            j = j - 1
    arr[low], arr[j] = arr[j], arr[low]
    return j
```

In the implementation,

- Initially we call the quick the quick_req function which calls the partition function and then recursively calls itself.
- In the partition function, we start from the first element and keep track of index of smaller (or equal to) elements as i .While traversing, if we find a element with lesser value than pivot value , we replace current element with jth element. Otherwise we ignore current element.
- The list is partitioned in a way that the pivot element is positioned in a such a way that it follows second point and then we call the quick_req function and Pivot elements are again chosen for the left and the right sub-parts separately.

6.3 Run time comparisons.

Average case: O(nlogn)

Worst case: $O(n^2)$ The worst case happens when the list is already sorted and pick the first or the last element as pivot

Best Case: O(nlogn) and the best case would be if the list were already sorted

Space Complexity: O(logn)

CHAPTER-7
QUICK SORT WITH 3 MEDIANS

7.1 Introduction

Quicksort with 3 medians can be alternate way to improve the efficiency of quicksort, we choose the first last and middle element of the initial array then to use the median of the three for the partitioning element. The median-of-three method helps quicksort in 2 ways.

- This quick sort makes sure that the worst case don't occur in quadratic time complexity
- This quick sort also reduces the running time of the algorithm by about 8 percent.

7.2 How it is implemented

- Initially, quicksort_mmedian function is called with the array,0,len(arr),0 as parameters.
- Median comparison acts as a global variable here
- The median of the 3 variables is calculated by using the statistics module.
- We will move the smallest entry to the first of the array
- And the largest entry to the middle.
- The array is rearranged in a way that we find two entries one larger that pivot (starting from the first) and one smaller than pivot (starting from the last). These two elements are swapped.
- Finally, the middle element is placed at last and the median element is placed at middle
- This process is continued until the array is sorted

```python
def partition_median(array, leftend, rightend):
    length = rightend - leftend
    if length % 2 == 0:
        middle = array[int(leftend + length / 2 - 1)]
    else:
        middle = array[int(leftend + length / 2)]

    pivot = statistics.median([array[leftend], array[rightend - 1], middle])

    pivotindex = array.index(pivot)  # only works if all values in array unique

    array[pivotindex] = array[leftend]
    array[leftend] = pivot

    i = leftend + 1
    for j in range(leftend + 1, rightend):
        if array[j] < pivot:
            temp = array[j]
            array[j] = array[i]
            array[i] = temp
            i += 1

    leftendval = array[leftend]
    array[leftend] = array[i - 1]
    array[i - 1] = leftendval
    return i - 1
```

```python
def quicksort_median(array, leftindex, rightindex,mediancomparison):
    if leftindex < rightindex:
        newpivotindex = partition_median(array, leftindex, rightindex)

        mediancomparison += (rightindex - leftindex - 1)
        quicksort_median(array, leftindex, newpivotindex,mediancomparison)


        quicksort_median(array, newpivotindex + 1, rightindex,mediancomparison)
    return array
```

```python
start_time = timeit.default_timer()

array=quicksort_median(arr, 0, len(arr),mediancomparison)
end_time = timeit.default_timer() - start_time
```

7.3 Run time comparisons.

Average case: O ($n$log(n))

Worst case: O(nlog(n))
Best Case O(nlog(n))

Space Complexity: O (1)

CHAPTER-8
HEAP SORT

8.1 Introduction

Heap Sort is one of the best sorting methods as it is in-place and with worst-case running time of logarithmic function. Heap sort involves building a Heap data structure from the given list and then utilizing the Heap to sort the list.

8.2 How it is implemented

Heap sort algorithm is divided into basic parts:

- Heapify the unsorted list/list.
- Then a sorted list is created by repeatedly removing the largest element from the heap and inserting it into the list. The heap is recursively reconstructed using the heapify function

```
    l = len(A) - 1
    for i in range((l // 2), 0, -1):
        max_heapify(A, l, i)
    start_time = timeit.default_timer()
    for i in range(l, 0, -1):
        A[l], A[1] = A[1], A[l]
        l = l - 1
        max_heapify(A, l, 1)
```

```
def max_heapify(A, l, i):
    large = i
    left = 2 * i
    right = (2 * i) + 1
    while left <= l and A[left] > A[large]:
        large = left
    while right <= l and A[right] > A[large]:
        large = right
    if large != i:
        A[large], A[i] = A[i], A[large]
        max_heapify(A, l, large)
```

In the implementation,

- First step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap).
- The heapfiy function is creating a max heap structure.
- Maximum element is found and placed at the end of the array.
- Then heapfiy is again called and the step 3 is called until whole of the array is sorted

8.3 Run time comparisons.

Average case: $O(n \log(n))$

Worst case: O(nlog(n))
Best Case:O(n*log(n))

Space Complexity: O (1)

CHAPTER-9

TKINTER AND SAMPLE GUI

INTRODUCTION

The tkinter package is the standard Python interface to the Tk GUI toolkit and it is not the only GUI Programming toolkit for Python. It is however the most used one.

GUI CODE EXPLANATION

```
import ...

root = tk.Tk()

# setting the windows size
root.geometry("600x400")
```

Explanation: Setting the windows size.

```
# declaring string variable
# for storing name and password
name_var = tk.StringVar()
passw_var = tk.StringVar()
```

Explanation: Declaring the string variables.

```
23    name = name_var.get()
24
25    arr = name.split(',')
```

Explanation: get function is used to get the variable from textbox and store from the name. The split function is used to convert the string which is stored in the name to list

```
26      for i in range(0,len(arr)):
27          arr[i]=int(arr[i])
```

Explanation: As the array's variables stored in string format, we use int function to convert to the array.

```
label = tk.Label(root, text="Array is sorted and the final out put is " + str(
    arr) + " and The time taken to excute the insertion sort alogirthm is " + str(time))
label.place(x=500, y=500)
```

Explanation: The tk.label is used to show the output in the GUI.The place attributes is used to place in root window.

```
start_time = timeit.default_timer()

if len(arr) == 1:
    return arr
arr = mergearray(arr)
time = timeit.default_timer() - start_time
```

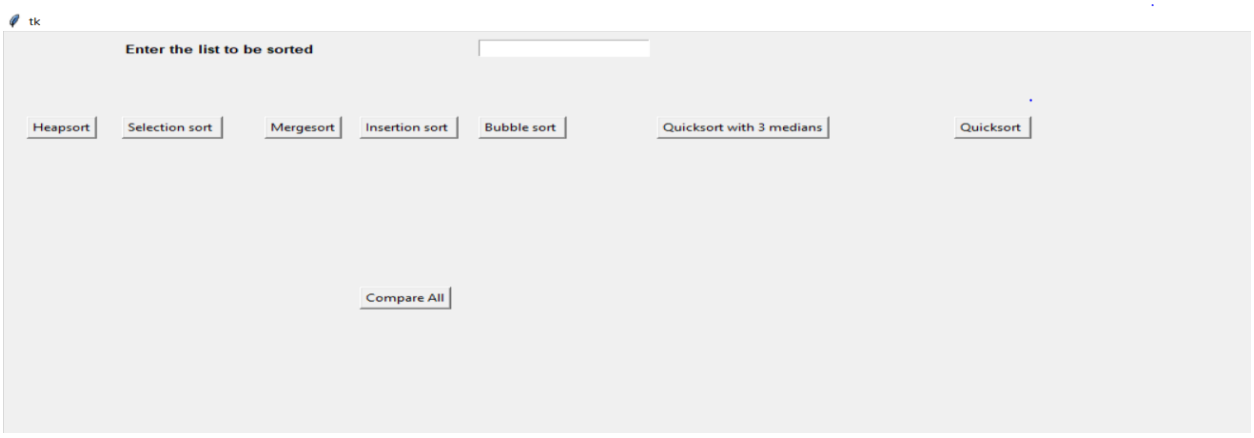
Explanation: The time taken to execute the above function is calculated using the above code. The time taken is displayed in seconds. To get the time taken we use python timeit function and its attribute default_timer().

```
# method
name_label.place(x=100, y=10)
compareall.place(x=300, y=300)
name_entry.place(x=400, y=10)
threequicksort.place(x=550, y=100)

Mergesort.place(x=220, y=100)

Bubblesort.place(x=400, y=100)
Heapsort.place(x=20, y=100)
Quicksort.place(x=800, y=100)
Insertionsort.place(x=300, y=100)

Selectionsort.place(x=100, y=100)
```

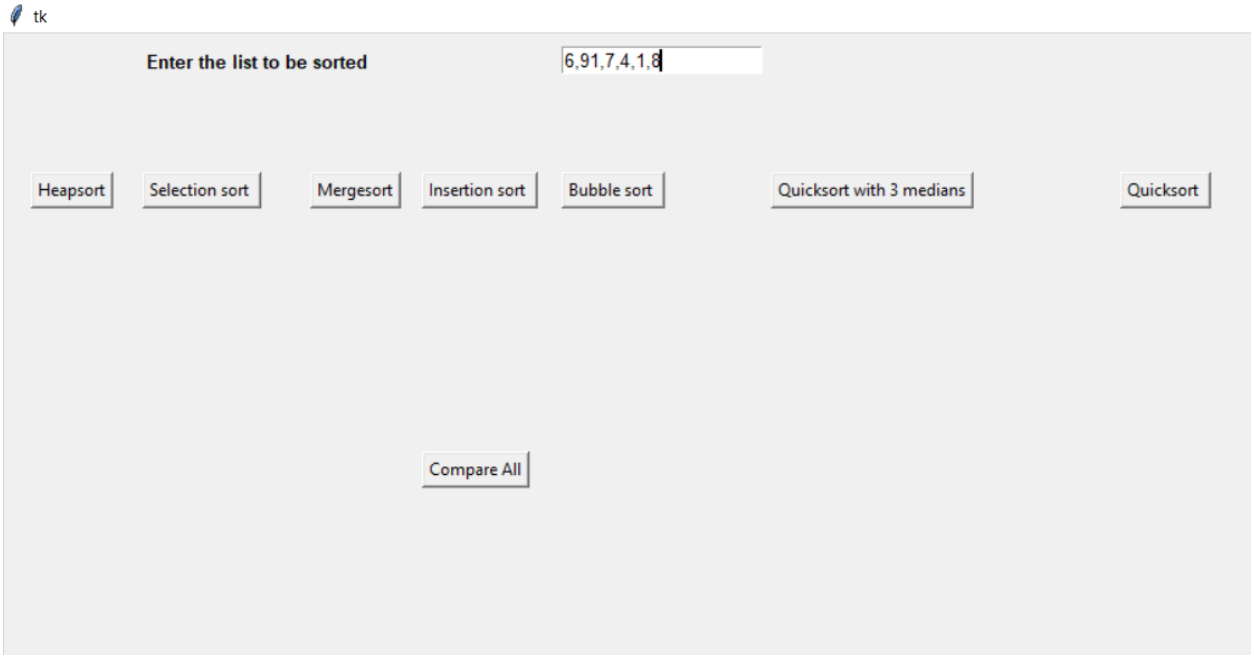
Explanation: All the labels are placed in a way that it follows the simple graphical structure which has x-axis and y-axis points.

SAMPLE RUNS AND OUTPUTS:

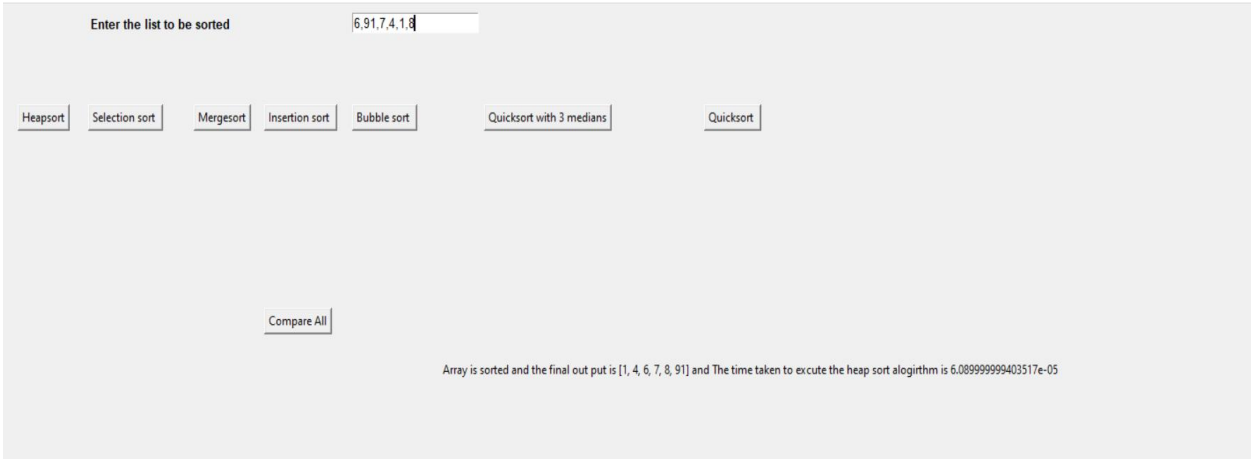
Step 1: Run the code in the console/Preferred GUI



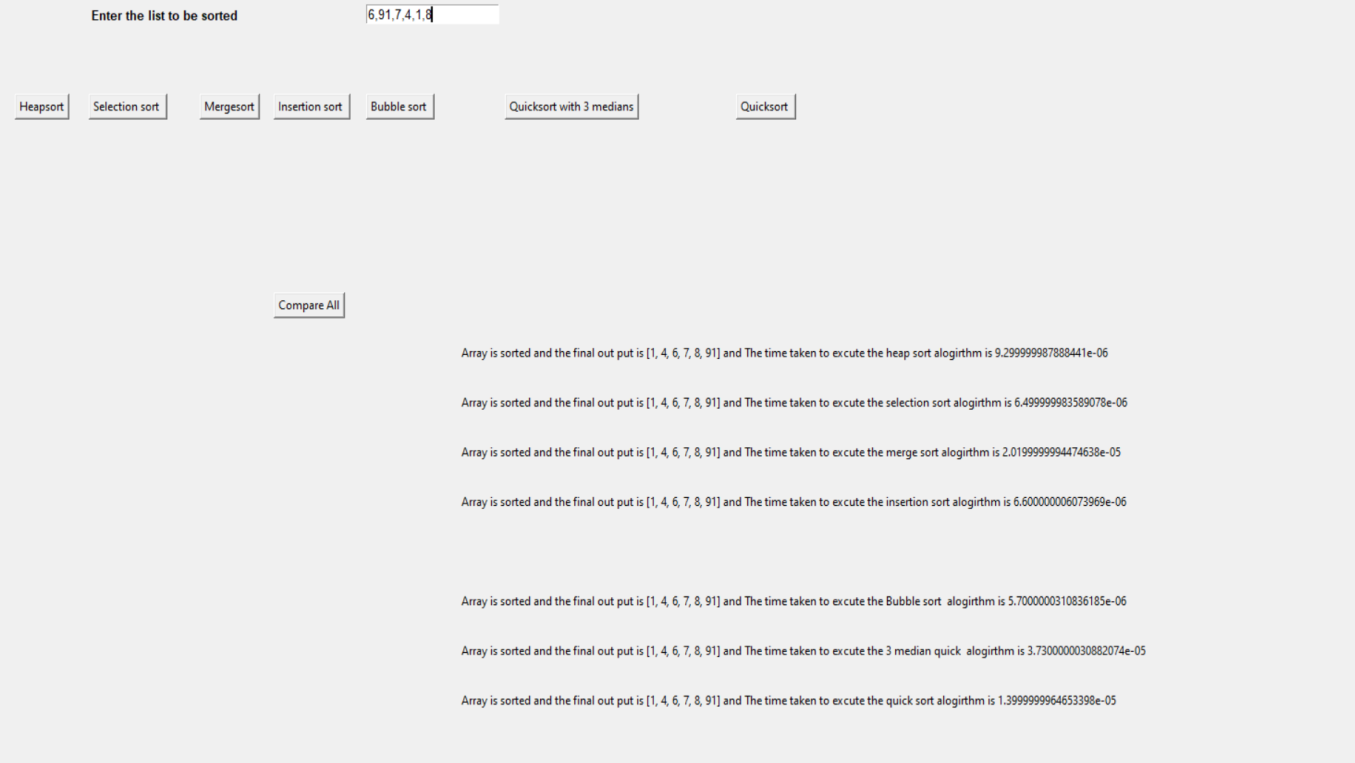
Step 2: Enter the numbers in the textbox



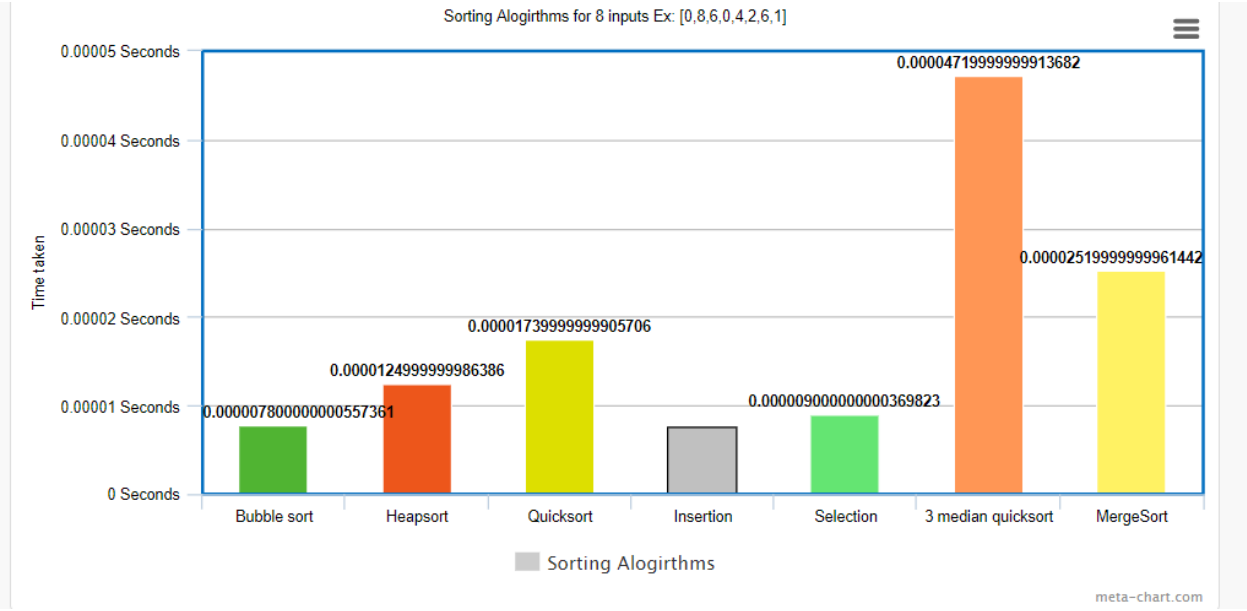
Step 3: Press the preferred button to start the corresponding sorting process.



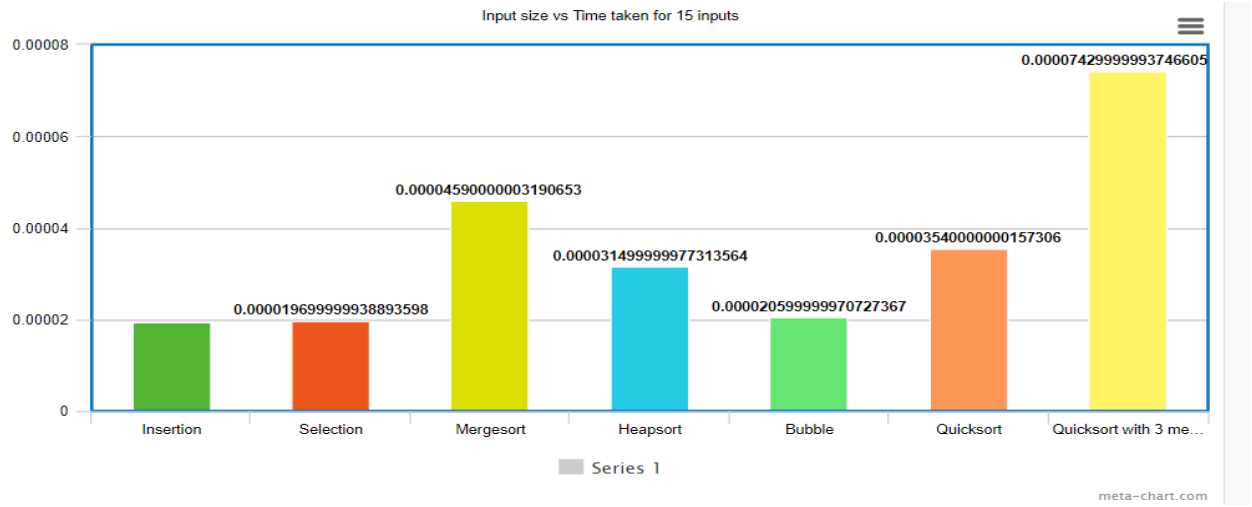
Step 4: Press on Compare All to compare the output of all sorting algorithms



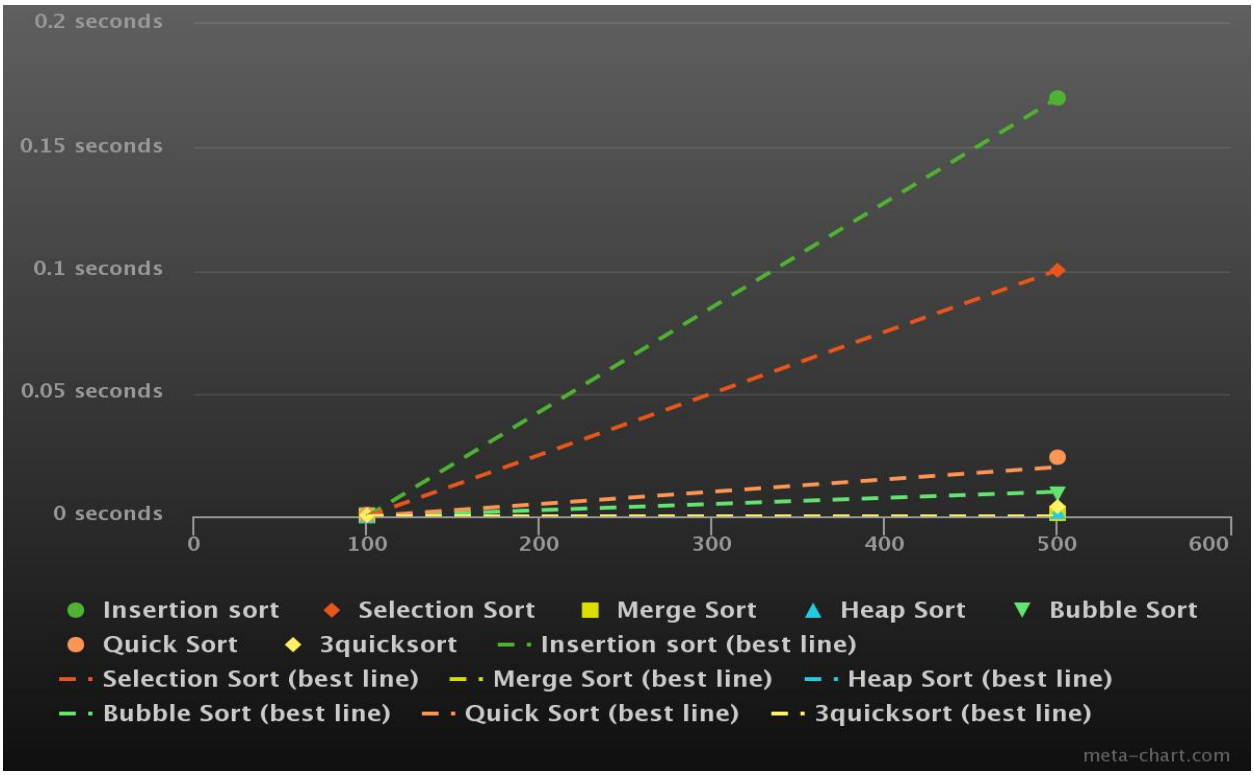
Comparison of the sorting algorithms



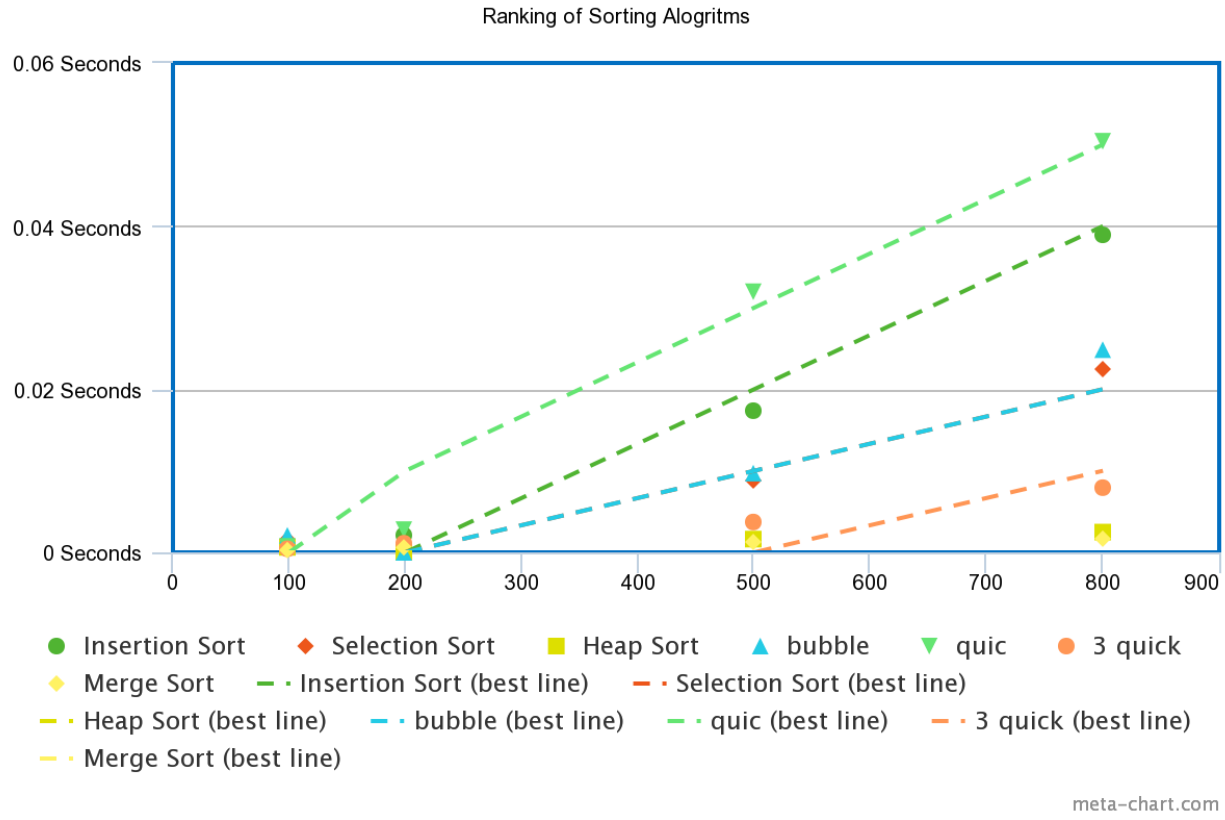
A graph of the 7 sorting algorithms for 8 inputs.



A graph of the 7 sorting algorithms for 15 inputs.



A comparison graph for 7 sorting algorithms. Y axis represents the input size whereas X-axis represents amount of time taken to sort the list.



The date used to generate above chart is randomly generated numbers from 1-10.

CHAPTER 10

RANKING OF THE SORTING ALOGRITHMS BASED ON TIME TAKEN IN SECONDS FOR AN INPUT SIZE OF 500.

S.NO	NAME	RANK	TIME TAKEN(sec)
1	Merge Sort	1	0.001
2	Heap Sort	2	0.002
3	3Quick Sort	3	0.003
4	Quick	4	0.009
5	Selection Sort	5	0.017
6	Insertion Sort	6	0.01
7	Bubble Sort	7	0.02