Course:	Design and Analysis of Algorithms CSE 5311 – Fall 2021
Module:	Sorting Algorithms
Deliverable:	Programming Project

Version: [1.0] **Date: [11/21/2021]**

Name: **Prithiviraj Eswaramoorthy**

Student Id: **1001860633**

Section**: 2218-CSE-5311-002**

Assignment: **Programming Project**

TABLE OF CONTENTS

1. Introduction and Scope of the project

Sorting algorithms are a sequence of well-defined instructions that puts elements of a list into an order. The most frequently used orders are numerical order either ascending or descending. Sorting is useful for representation of huge data in standardized and human readable output. There are several sorting techniques and here we are analyzing and comparing **Insertion sort, Selection sort, Bubble sort, Merge sort, Heap sort, quick sort and 3-way quick sort** algorithms. We will be computing the running time for these algorithms for different data sizes and so we can have a better understanding in usage of these algorithms in different scenarios.

The project is developed in Python 3.9 and PyCharm IDE for development and execution. The Matplotlib library is used to visualize and compare the running time of these algorithms with different size of input to these algorithms.

2. Sorting Algorithms

2.1 Insertion Sort

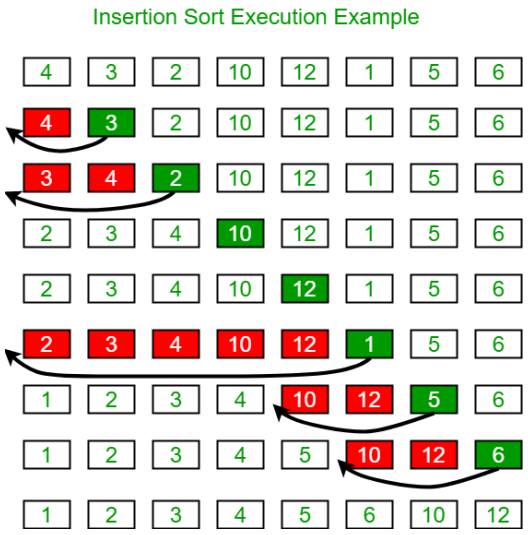
Insertion sort is a simple sorting algorithm that splits the input into sorted and unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

The Algorithm follows these steps:

To sort a list of size n in ascending order:

1. Iterate from list[1] to list[n] over the array.
2. Compare the current element to its predecessor.
3. If the element is smaller than the predecessor, compare it to the elements before.
4. Move the larger numbers one position up to make space for the swapped element.

Example:




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

The above algorithm is used in the project to sort the list and its functioning is:

- j will take the value of 0 and list[j] will take value of first element and key takes the value of second element of the list.
- While key is smaller than first element, it copies the value of first to the second element and decrements the j value. In the next iterations the key value is compared to all previous elements and copied to the correct position using the while loop above.

Best Case Runtime Complexity:

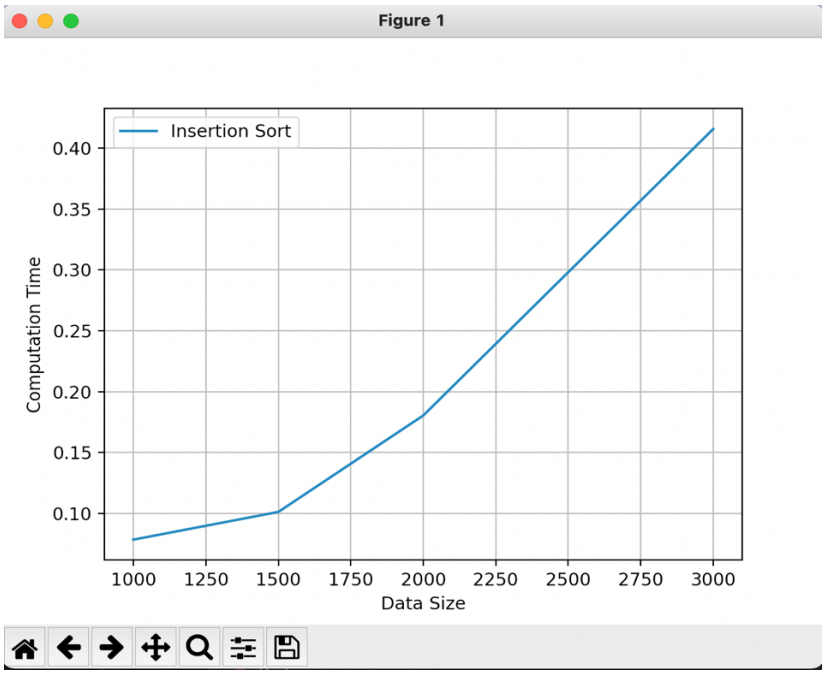
The best case of Insertion sort occurs when the list is already in a sorted list. If it is already sorted, the function runs through the list once to check all values and the time complexity is **0(n).**

Worst Case Runtime Complexity:

The worst case of Insertion sort occurs when the list is sorted in reverse order or descending order. In such case for iteration of each element, the loops runs for all previous elements of the list to find the position of the element, So the time complexity is **O(n²).**

It is an In-place algorithm and the same list is sorted without making any copy of it and the **space complexity** for it is O(1).

In this project random inputs of different sizes are generated and sorted, and their different running times are captured and plotted in a graph using matplotlib library in python. The obtained graph is:



2.2 SELECTION SORT

The selection sort algorithm sorts a list by repeatedly finding the minimum element from the unsorted part and putting it at the beginning.

The algorithm maintains two sub lists in a given list:

1. The subarray which is already sorted.
2. The subarray which is unsorted.

In every iteration of selection sort, the minimum element from the unsorted sub list is picked and moved to the sorted sub list.

The algorithm is coded as:

```
def selectionSort(list):
    length = len(list)
    for i in range(length):
        smallest_num = i
        for j in range (i+1, length):
            if list[smallest_num] > list[j]:
                smallest_num = j
        list[i], list[smallest_num] = list[smallest_num], list[i]
    return list
```

Example :

List[] = 85,75,10, 60, 5

Here, i will hold the index of 85 and compares with all other elements of the list. At the end of the j loop j holds the index of the smallest element and swaps it with the first element.

Step 2:

List[] = 5, 75,10, 60, 85

5 and 85 gets swapped, in the next step checks element less than 75 and swaps them.

Step 3:

List[] = 5, 10,75, 60, 85

Step 4:

List[] = 5, 10, 60, 75, 85

Step 5:

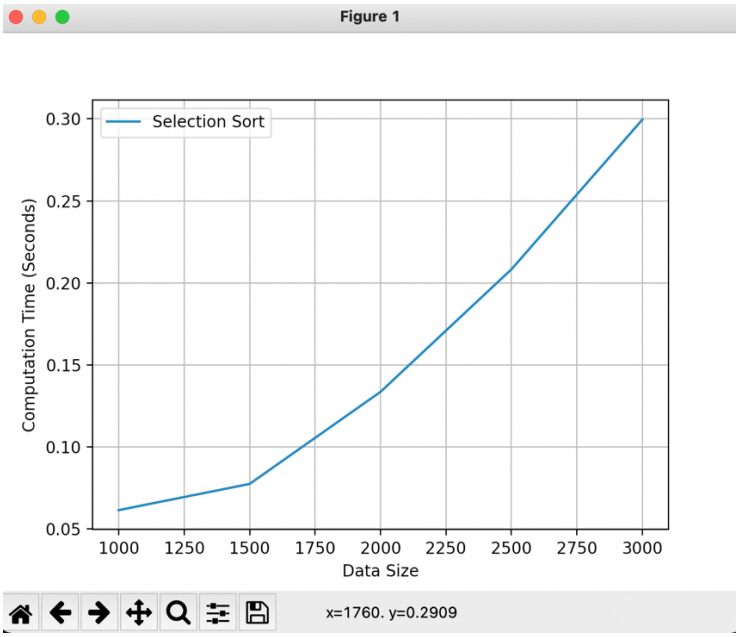
List[] = 5, 10, 60, 75, 85. (No change as 75<85).

Runtime Complexity:

In selection sort in all the cases irrespective of the inputs, there are two nested loops executed to compare one element with rest elements of the list. So, the runtime complexity of selection sort is **O(n²)** in all the cases of execution (Best, Average, Worst cases). Maximum there would be O(N) swaps involved in sorting. When swaps are costly, this algorithm can be used.

It is an In-place algorithm and the same list is sorted without making any copy of it and the **space complexity** for it is O(1).

In this project random inputs of different sizes are generated and sorted, and their different running times are captured and plotted in a graph using matplotlib library in python. The obtained graph is:



2.3 BUBBLE SORT

Bubble sort is an algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example : [6, 2, 5, 3, 9]

First iteration:

6,2 swapped → [2,6,5,3,9]

6,5 swapped →[2,5,6,3,9]

6,3 swapped →[2,5,3,6,9]

Second iteration:

5,3 swapped→[2,3,5,6,9]

Third Iteration:

Its already sorted but still the algorithm runs one more iteration and no swaps are needed here.

The algorithm is coded as:

```
ef bubbleSort(list):
    length = len(list)
    for i in range(length-1):
        for j in range(0, length-i-1):
            if list[j] > list[j+1]:
                list[j], list[j+1] = list[j+1], list[j]
    return list
```

Best Case Runtime Complexity:

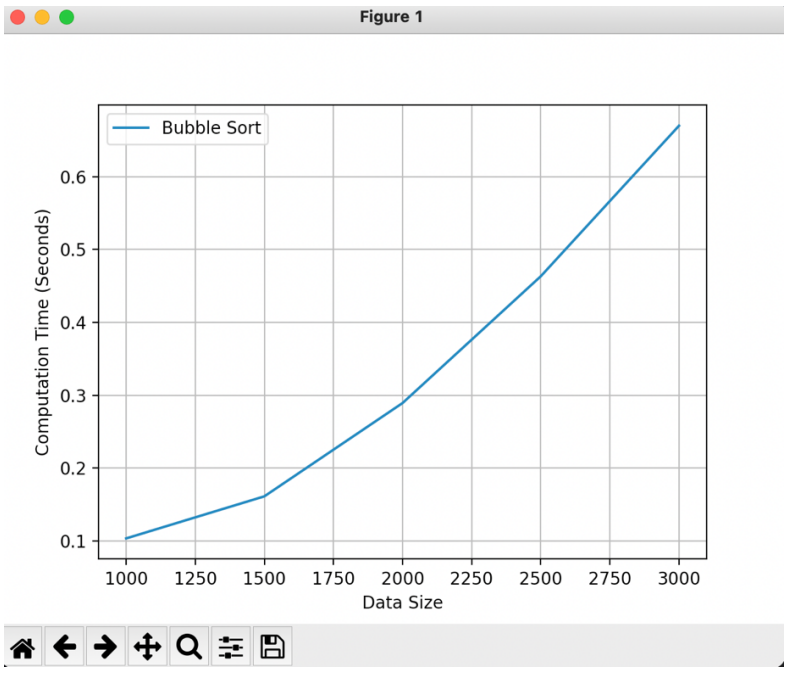
The best case of Bubble sort occurs when the list is already in a sorted list. If it is already sorted, the function runs through the list once to check all values and the time complexity is **0(n).**

Worst Case Runtime Complexity:

The worst case of Bubble sort occurs when the list is sorted in reverse order or descending order. In such case for iteration of each element, the loops runs for all previous elements of the list to find the position of the element, So the time complexity is **O(n²).** The average case is also **O(n²)** as the inner loop runs every time.

It is an In-place algorithm and the same list is sorted without making any copy of it and the **space complexity** for it is O(1).

In this project random inputs of different sizes are generated and sorted, and their different running times are captured and plotted in a graph using matplotlib library in python. The obtained graph is:



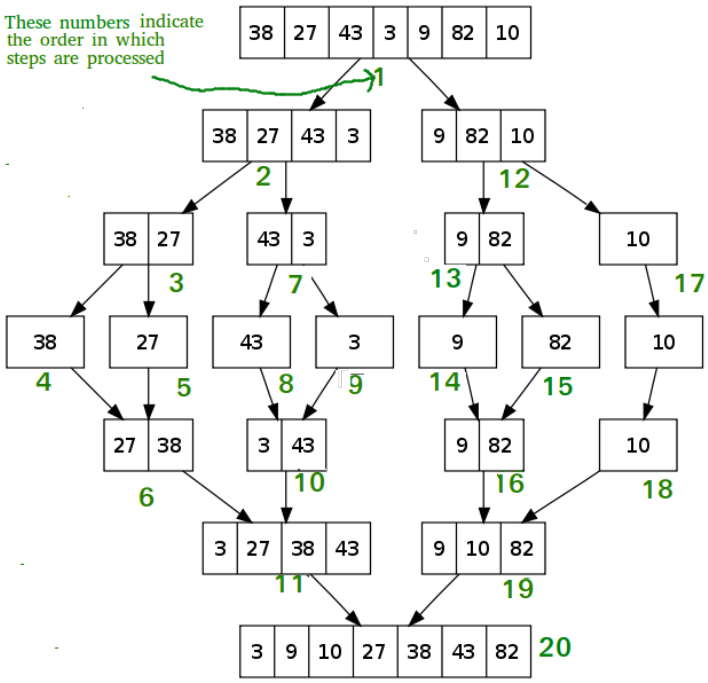
2.4 MERGE SORT

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. A merge function is used for merging the two halves.

The merge sort performs the following operations:

1. Divide unsorted array into two sublists in each iteration.
2. Repeats the above step until each sublist has one element.
3. Take adjacent pairs, compares and merges them in order to a list.
4. Repeats the above step until a single sorted list is obtained.

Merge sort is illustrated in the example below:



Algorithm Implementation:

Here in the code below, the list is split into halves repeatedly until the list is split into left and right list with one element in it.

The while loop will compare the left and right list and adds to the list from the starting index in sorted order. The next two while loops will add the remaining elements of the left and right list to the end of the list. Repeatedly it adds until the left and right lists merge to form a single sorted list.

The Algorithm is coded as :

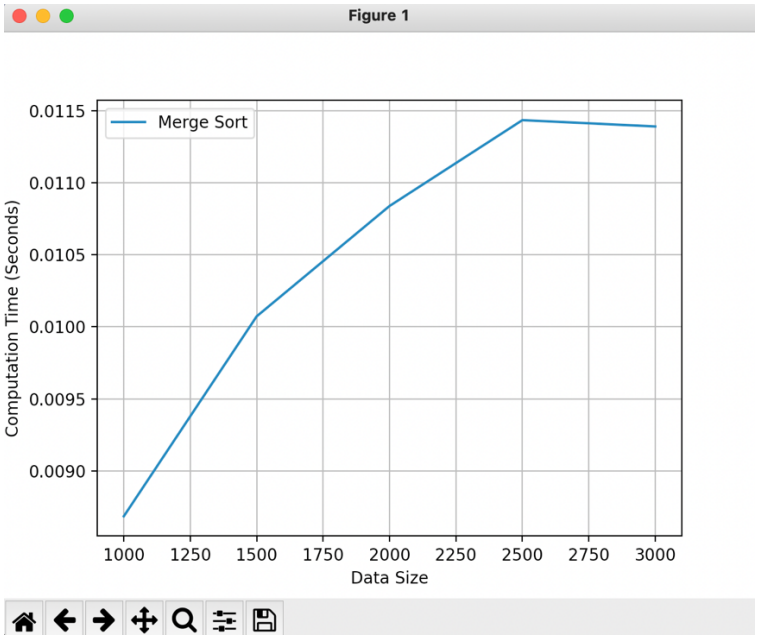
```
def mergeSort(list):

    if len(list) > 1:
        mid = len(list) // 2
        left_list = list[:mid]
        right_list = list[mid:]
        mergeSort(left_list)
        mergeSort(right_list)

        i = 0
        j = 0
        k = 0

        while i < len(left_list) and j < len(right_list):
            if left_list[i] <= right_list[j]:
                list[k] = left_list[i]
                i += 1
            else:
                list[k] = right_list[j]
                j += 1
            k += 1

        while i < len(left_list):
            list[k] = left_list[i]
            i += 1
            k += 1

        while j < len(right_list):
            list[k] = right_list[j]
            j += 1
            k += 1
    return list
```

Runtime Complexity:

In Merge sort in all the cases irrespective of the inputs, it involves divide and merge strategy. To divide the lists it takes O(log n) time and to merge it takes O(n). The overall time complexity for the complete sort in O(n log n). There is no best, average or worst cases and all cases takes similar time.

It is an Out-place algorithm since left list and right list are created and the **space complexity** for it is O(n).

In this project random inputs of different sizes are generated and sorted, and their different running times are captured and plotted in a graph using matplotlib library in python. The obtained graph is:



2.4 HEAP SORT

Heap sort is a comparison-based sorting technique based on binary heap data structure. It is like selection sort where we find the minimum element and place the minimum element at the beginning. The same process is repeated for the remaining elements.

Steps for Heap sort in Increasing order:

1. Build a max heap from the input data.
2. Now, the largest item is stored at the root of the heap. Replace with the last item of the heap. Heapify the root of the tree.
3. Repeat the above step when size of heap is greater than one.

The algorithm is coded as below:

```
def form_heap(list,length,i):
    highest = i
    left_node = 2 * i + 1
    right_node = 2 * i + 2

    if left_node < length and list[highest] < list[left_node]:
        highest = left_node
    if right_node < length and list[highest] < list[right_node]:
        highest = right_node

    if highest != i:
        list[i], list[highest] = list[highest], list[i]
        form_heap(list, length, highest)

def heapSort(list):
    length = len(list)
    for i in range(length//2 - 1, -1, -1):
        form_heap(list, length, i)
    for i in range(length-1, 0, -1):
```

```
        list[i], list[0] = list[0], list[i]
        form_heap(list, i, 0)
    return list
```

The form_heap method above initializes largest as root and builds a max heap of the elements present in the list.

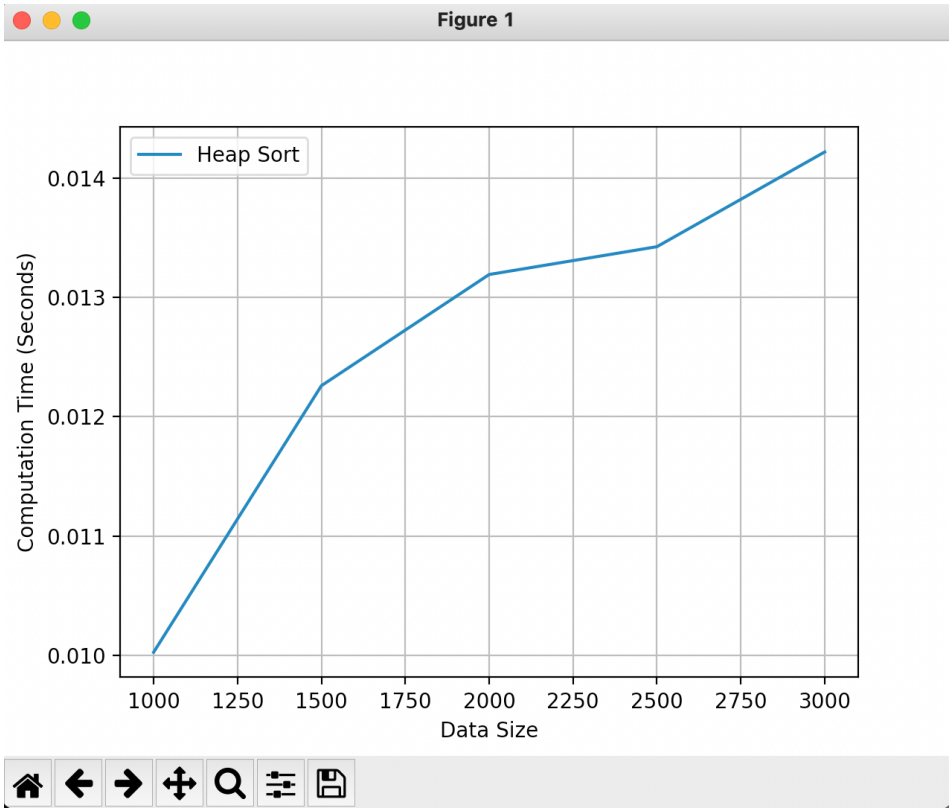
The heapSort method replaces the last item of the heap and heapify again and sorts the list.

Runtime Complexity:

In Heap sort in all the cases irrespective of the inputs, it involves creating heap and heapify. To heapify the list it takes O(log n) time and to build a heap it takes O(n). The overall time complexity for the complete sort in O(n log n). There is no best, average or worst cases and all cases takes similar time.

It is an In-place algorithm and the same list is sorted without making any copy of it and the **space complexity** for it is O(1).

In this project random inputs of different sizes are generated and sorted, and their different running times are captured and plotted in a graph using matplotlib library in python. The obtained graph is:



2.5 QUICK SORT

Quick sort is a Divide and Conquer Algorithm. It picks an element as pivot and partitions the list around the picked pivot. It can be performed in different ways:

1. Picking first element as pivot.
2. Picking last element as pivot.
3. Picking random element as pivot.
4. Picking median as pivot.

The algorithm is coded as below:

```
def partition(list, start, end):
    i = start - 1
    pivot = list[end]

    for j in range(start, end):
        if list[j] <= pivot:
            i = i+1
            list[i], list[j] = list[j], list[i]
    list[i+1], list[end] = list[end], list[i+1]
    return (i+1)

def quicksort(list, start, end):
    if len(list) == 1:
        return list
    if start < end:
        partIndex = partition(list, start, end)
        quicksort(list, start, partIndex-1)
        quicksort(list, partIndex+1, end)
    return list
```

- The algorithm partitions the list into two and swaps the element in a way that elements smaller than pivot are to the left list and the elements greater than the pivot are to the right list. Now the pivot is in the correct position.
- The above steps are repeated to the left and right list to the pivot.

Runtime complexity:

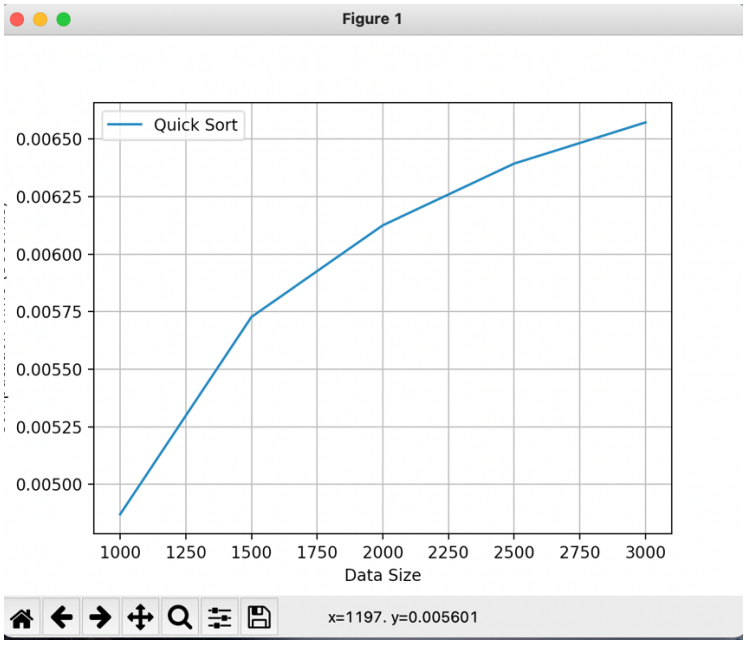
Best case: The best case occurs when middle element is chosen as pivot. The best case runtime complexity is **O(n log n)** and it is similar to merge sort.

Worst case: The worst case occurs when the partition method picks the greatest or smallest number as the pivot. The worst-case time complexity is **$O(n^2)$**

Average case: The average case runtime complexity is **O(n log n).**

It is an In-place algorithm and the same list is sorted without making any copy of it and the **space complexity** for it is O(1).

In this project random inputs of different sizes are generated and sorted, and their different running times are captured and plotted in a graph using matplotlib library in python. The obtained graph is:



2.6 3-WAY QUICK SORT

In 3-way quick sort algorithm, the array is divided into three parts:

1. Elements less than the pivot.
2. Elements equal to the pivot.
3. Elements greater than pivot.

The steps involved in 3-way quick sort is:

1. Choose three elements (first, last, middle) from the list. Assume middle as the pivot.
2. Move the smallest of these three elements to the first position.
3. Move the largest of these three elements to the middle position.
4. Find two entries: starting from the front, find element larger than pivot, starting from end, find element smaller than pivot.
5. Repeat the above steps until the elements near the middle element are swapped.
6. Insert pivot to the middle, move the middle element to the end. Now perform quick sort on the first and second half of the list recursively.

The algorithm is coded as:

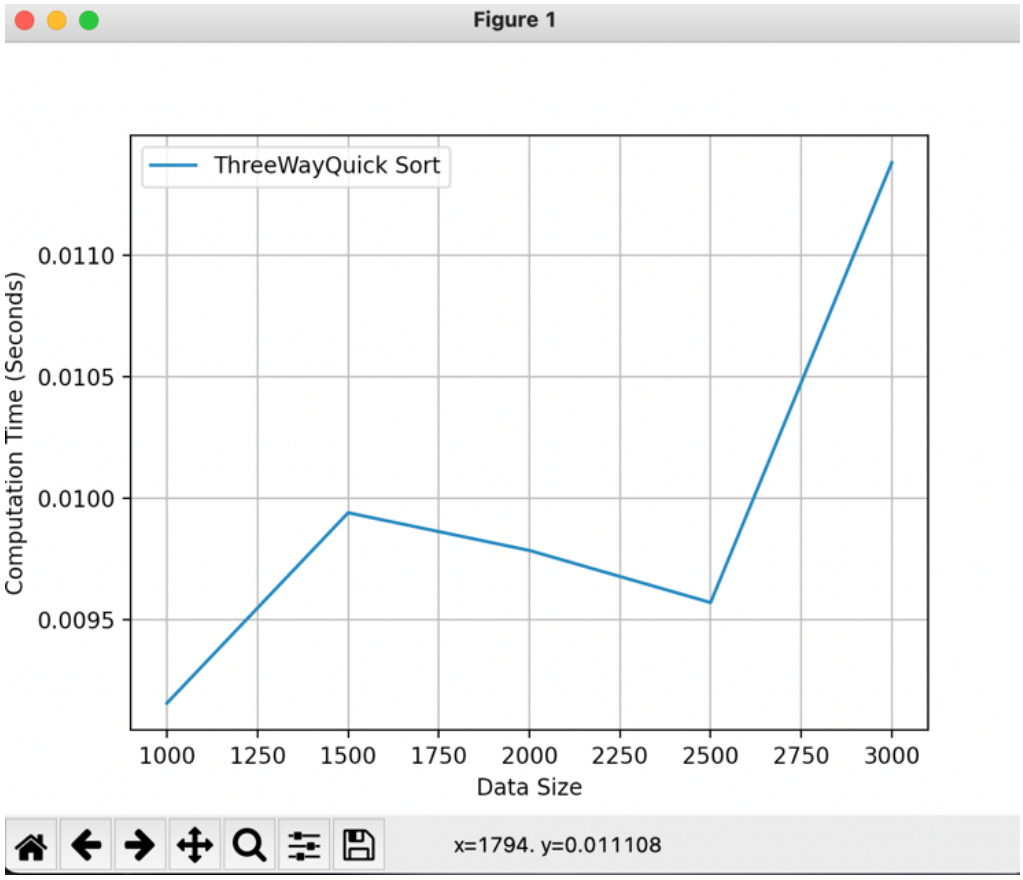
```
def threewaypartition(list, first, last, start, mid):
    pivot = list[last]
    end = last

    # Iterate while mid is not greater than end.
    while (mid[0] <= end):

        # Inter Change position of element at the starting if it's
value is less than pivot.
        if (list[mid[0]] < pivot):

            list[mid[0]], list[start[0]] = list[start[0]], list[mid[0]]

            mid[0] = mid[0] + 1
            start[0] = start[0] + 1

        # Inter Change position of element at the end if it's value is
greater than pivot.
        elif (list[mid[0]] > pivot):

            list[mid[0]], list[end] = list[end], list[mid[0]]

            end = end - 1

        else:
            mid[0] = mid[0] + 1


# Function to sort the array elements in 3 cases
def threewayQuicksort(list, first, last):
    # First case when an array contain only 1 element
    if (first >= last):
        return

    # Second case when an array contain only 2 elements
    if (last == first + 1):

        if (list[first] > list[last]):
            list[first], list[last] = list[last], list[first]

            return

    # Third case when an array contain more than 2 elements
    start = [first]
    mid = [first]

    # Function to partition the array.
    threewaypartition(list, first, last, start, mid)

    # Recursively sort sublist containing elements that are less than
the pivot.
    threewayQuicksort(list, first, start[0] - 1)

    # Recursively sort sublist containing elements that are more than
the pivot
    threewayQuicksort(list, mid[0], last)
    return list
```

Runtime Complexity:

Best case: The best-case runtime complexity is O(n) when median is picked as pivot.

Worst case: The worst case occurs when greatest or smallest number is picked as the pivot. The worst-case time complexity is **$O(n^2)$**

In this project random inputs of different sizes are generated and sorted, and their different running times are captured and plotted in a graph using matplotlib library in python. The obtained graph is:



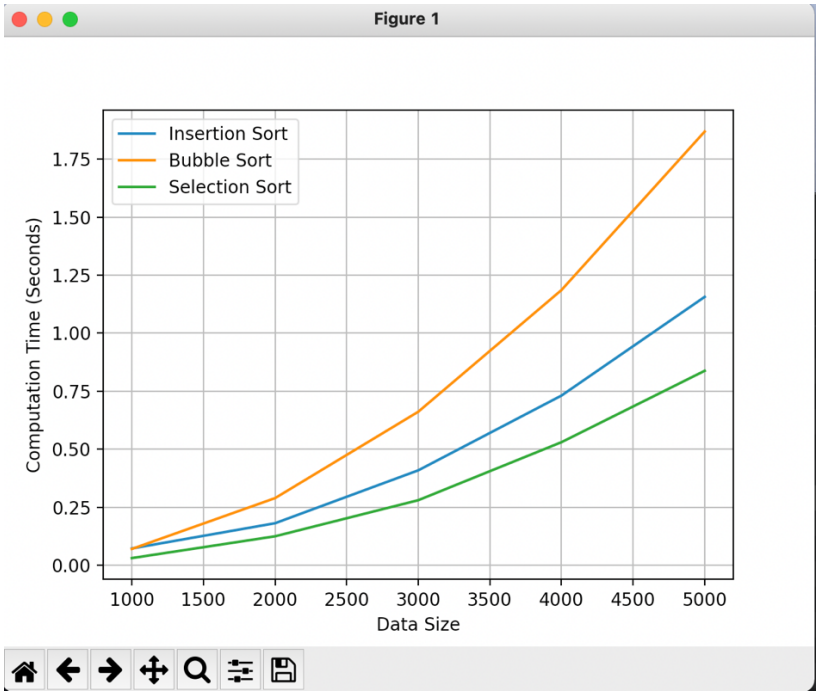
3 Algorithms Comparison

The running time of the algorithms are compared here for 5 lists of different sizes having 1000, 2000, 3000, 4000, 5000 elements in it.

The input to these elements in these lists is generated randomly within the numbers 1 to 1000 and the same set of random inputs is used to compare the running time between these three algorithms.

The below graphs are plotted for the size of the input and the computation time it takes for the algorithms using matplotlib library of python.

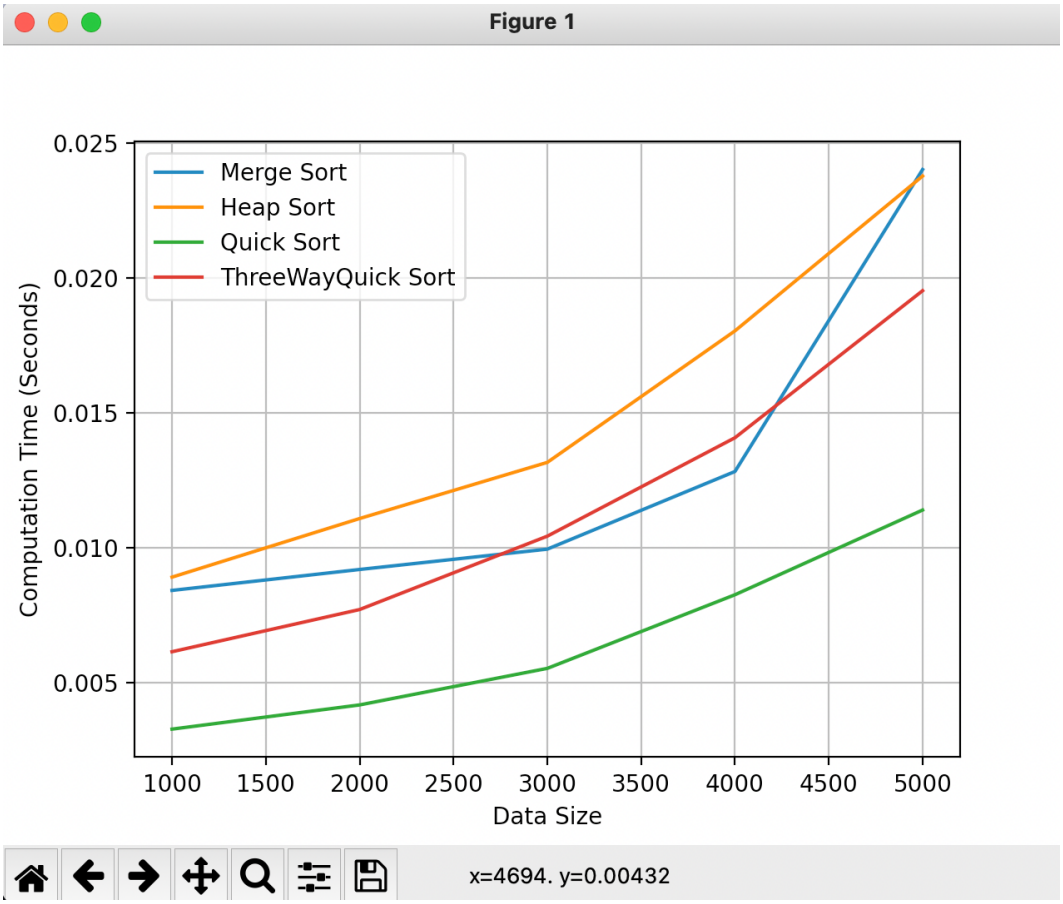
3.1 Insertion sort vs Selection vs Bubble sort



Here it is observed that the sorting time increases in all the three algorithms, as the size of the input to be sorted increases.

However, Selection sort takes the least computation time followed by Insertion sort, while Bubble sort is the slowest and takes the largest computation time.

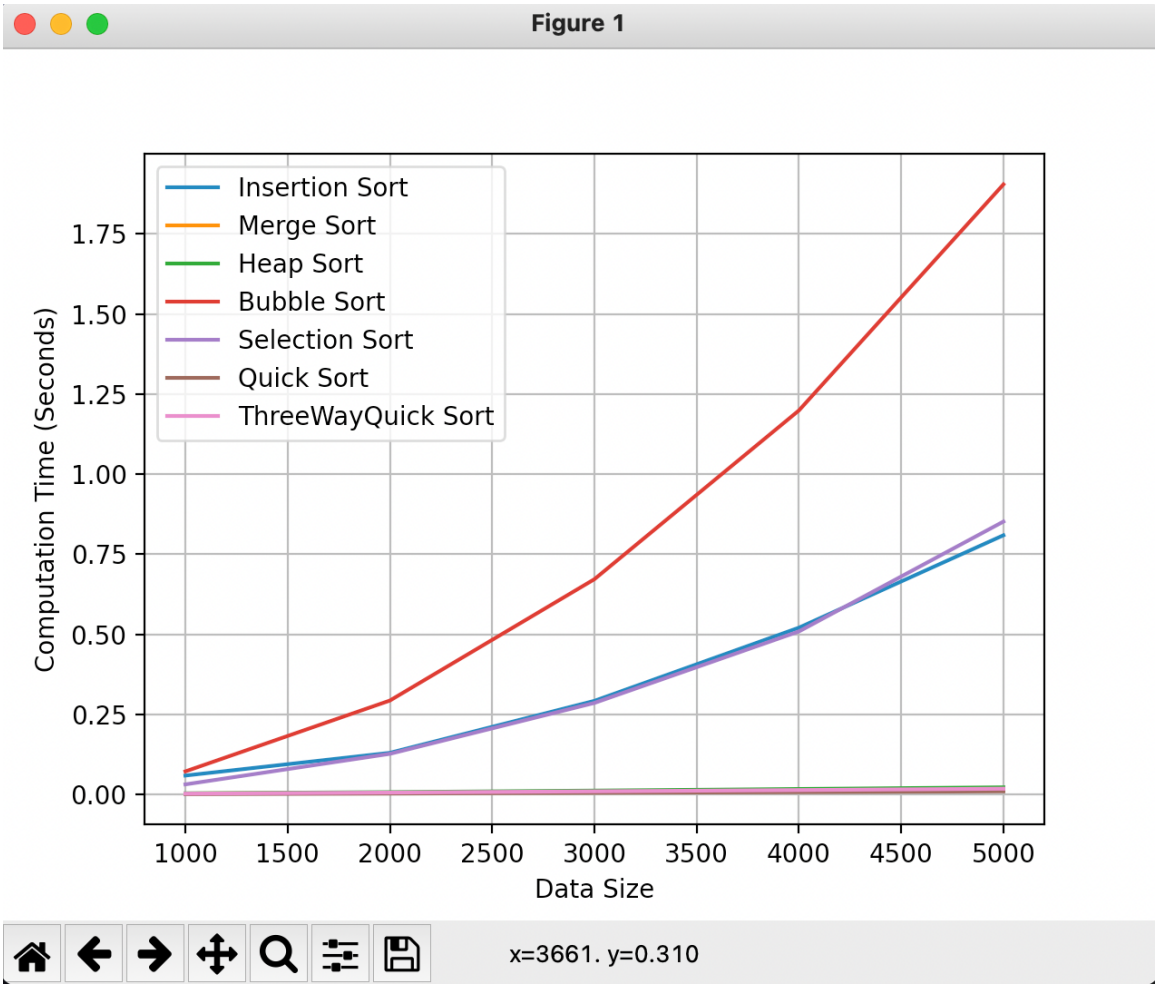
3.2 MERGE SORT VS HEAP SORT VS QUICK SORT VS 3-WAY QUICK SORT



Here it is observed that the sorting time increases in all the three algorithms, as the size of the input to be sorted increases.

However, Quick sort takes the least computation time followed by 3-way Quick sort, while Merge sort and Heap sort are the slowest and takes similar computation time.

3.3 COMPARISON OF ALL ALGORITHMS



The following are the observations when all the algorithms are executed with random similar inputs of varying sizes:

- Bubble sort is the least in terms of performance for larger inputs as it takes the longest time in comparison with other algorithms.
- The Selection and the Insertion sort have average performance in comparison with other algorithms having running time much lesser than Bubble sort.
- Heap sort, Merge sort, Quick and 3-way Quick sort have similar and quickest running time irrespective of the size of inputs and these algorithms have Divide and Conquer approach in common.

4 Conclusion and Recommendation

From the above observations, it is evident that the runtime of algorithm increases with increase in size of the input. We can have a conclusion that when the inputs are random and gets arbitrarily large, algorithms that follow Divide and Conquer approach like Merge, Heap, Quick, 3-way Quick sort has better performance in terms of time over other algorithms. Also, we can observe that Bubble sort is not so efficient in most of the cases.

When Random input is already sorted:

A sample execution is made when the random data is already sorted, and it is observed that Insertion sort is the quickest and 3-way quick sort is the slowest.

When Random input is reversed sorted:

A sample execution is made when the random data is reverse sorted, and it is observed that Merge sort and Heap sort are the quickest and 3-way quick sort is the slowest.

When Input is small:

A sample execution is made for smaller input size, and it is observed that Insertion sort is the quickest and Merge sort is the slowest.

Recommendation:

The recommendation would be to use algorithms that follow Divide and Conquer approach when the input size is large as Heap sort, Merge sort, Quick and 3-way Quick sort have faster execution time when input size is large.

Usage of Insertion or Selection sort when input is smaller is a better approach than choosing Divide and Conquer algorithms.

References

- https://www.geeksforgeeks.org/3-way-quicksort-dutch-national-flag/
- https://www.geeksforgeeks.org/quick-sort/
- https://www.algorithmsandme.com/
- https://matplotlib.org/