# Sorting Algorithms Analysis & Visualization

# Table of Contents

## Project Description:

5 different sorting algorithms namely Merge sort, Heap sort, quicksort, insertion sort and heap sort are compared and analyzed to get a better understanding of their execution time in relation to one another. From my observation, we will plot a graph to visualize the execution time. There are multiple sorting techniques such as the following :

**In-place/Outplace technique –**

A sorting technique is inplace if it does not use any extra memory to sort the array.

Among the comparison based techniques discussed, only merge sort is outplaced technique as it requires an extra array to merge the sorted subarrays.

Among the non-comparison based techniques discussed, all are outplaced techniques. Counting sort uses a counting array and bucket sort uses a hash table for sorting the array.

**Online/Offline technique –**

A sorting technique is considered Online if it can accept new data while the procedure is ongoing i.e. complete data is not required to start the sorting operation.

Among the comparison based techniques discussed, only Insertion Sort qualifies for this because of the underlying algorithm it uses i.e. it processes the array (not just elements) from left to right and if new elements are added to the right, it doesn't impact the ongoing operation.

**Stable/Unstable technique –**

A sorting technique is stable if it does not change the order of elements with the same value.

Out of comparison based techniques, bubble sort, insertion sort and merge sort are stable techniques. Selection sort is unstable as it may change the order of elements with the same value. For example, consider the array 4, 4, 1, 3.

In the first iteration, the minimum element found is 1 and it is swapped with 4 at 0th position. Therefore, the order of 4 with respect to 4 at the 1st position will change.

Similarly, quick sort and heap sort are also unstable.

Out of non-comparison based techniques, Counting sort and Bucket sort are stable sorting techniques whereas radix sort stability depends on the underlying algorithm used for sorting.

## 1.Project Overview:

### 1a. Programming Languages Used:

HTML and CSS were used to create a website to provide a User Interface. View.JS was used to code the various sorting algorithms and the proper functioning of the website. Google charts were used to graphically represent the execution times in order to visualize the output.

## 2. Purpose of the project:

The purpose of the project was to understand the working of the various sorting algorithms when different sizes of unsorted inputs were given. The random numbers are generated to form the unsorted list which is then passed to the various sorting algorithms to process.

The **performance.now()** is used before the execution of the sorting algorithm and after the sorting algorithm and is subtracted in order to find the difference between the starting and ending time. This difference is the **execution time.**

## 3. The scope of the work:

### 3a. Current situation:

The project is **complete** and the required output was achieved. A graph is generated which shows a visual comparison among all the execution times.

### 3b. Work Distribution:

This project was done by me alone and I set aside 2 hours every day to work on this.

## 4. Sorting Algorithms, Codes and Outputs:

In this section, we are going to take into consideration 5 different sorting algorithms (Merge sort, Insertion sort, Quick sort, Heap sort, Bubble sort) and compare their various execution times with different input sizes. I am then producing a chart which will provide a visual representation of their execution time.

### 4a. Merge Sort:

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The mergeSort() function** is used for merging two halves. The two sorted arrays are merged back and form a completely sorted array.

Code:

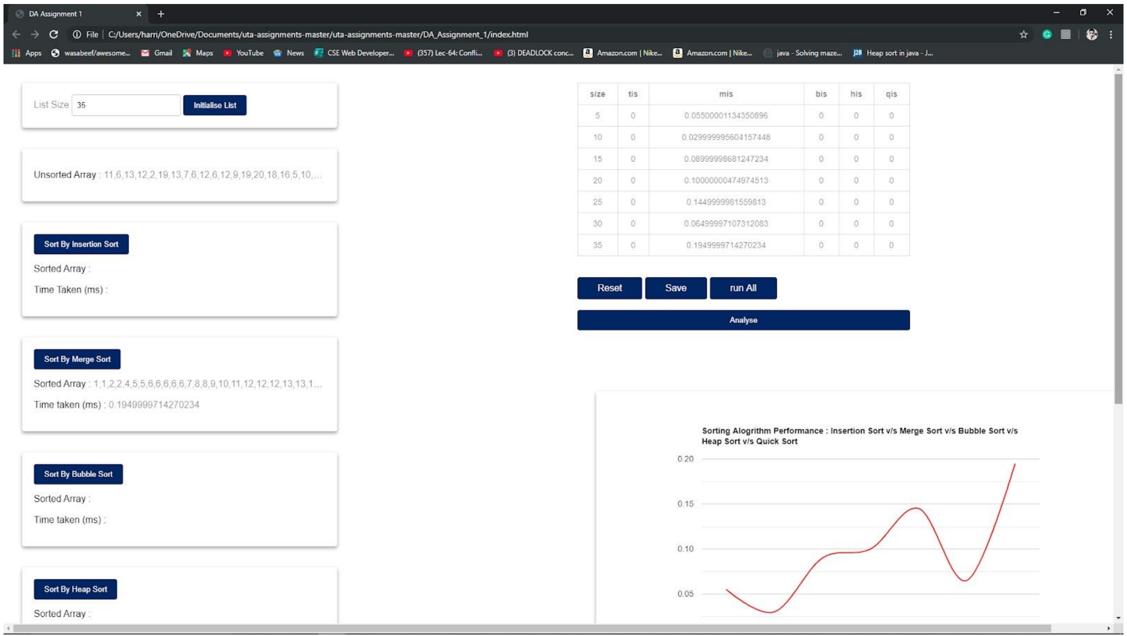
```
var initMergeSort = function(){
    let start = performance.now();
    let sortedArray = mergeSort(unsortedArray)
    let end = performance.now();

    mergeSortTimeTaken = (end-start)

    document.getElementById('mergeSort').innerHTML = sortedArray;
    document.getElementById('ms_time_taken').innerHTML = mergeSortTimeTaken
}

var mergeSort = function(array){

   if (array.length === 1) {
       return array
    } else {
       var split = Math.floor(array.length/2)
       var left = array.slice(0, split)
       var right = array.slice(split)

       left = mergeSort(left)
       right = mergeSort(right)

       var sorted = []
       while (left.length > 0 || right.length > 0) {
       if (right.length === 0 || left[0] <= right[0]) {
           sorted.push(left.shift())
       } else {
           sorted.push(right.shift())
       }
       }

       return sorted
    }
}
```

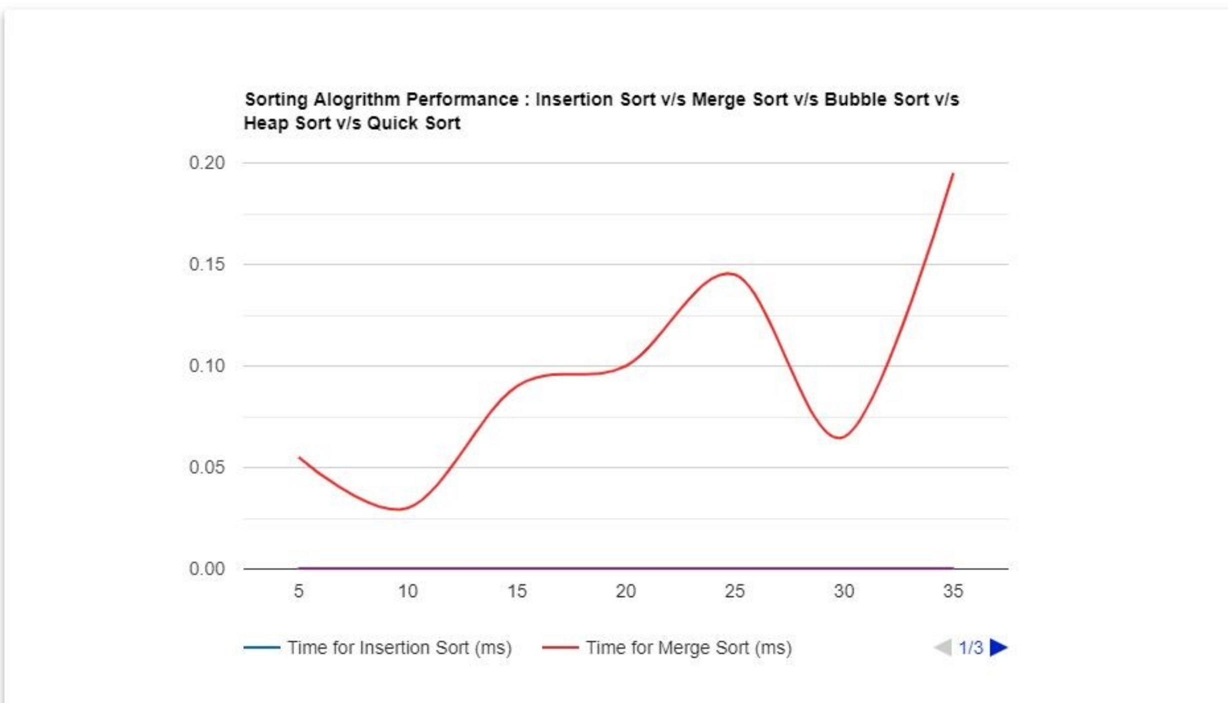
HTML code:

```html
<br><br>
<div class="box">
    <button class="primaryBtn" onclick="initMergeSort()">Sort By Merge Sort</button>
    <p class="truncate">Sorted Array : <label id="mergeSort"></label><br><br> Time taken (ms) : <label id="ms_time_taken"></label></p>
</div>

 <br><br>
```

This html code creates a button that would initiate the merge sort for the unsorted list.

OUTPUT:




The above images shows the execution time for Merge sort for different input sizes such as 5,10,15,20,25,30 and 35 elements. The graph shows a visual representation of the output.

**4b. 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 maximum element and place the maximum element at the end. We repeat the same process for remaining element.

CODE:

```
var heapSort = function(){
    let arr = [...unsortedArray];
    let n = arr.length;

    var i;

    let start = performance.now();


        // Build max heap
    for (i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

            // Heap sort
    for (i=n-1; i>=0; i--)
    {
        var temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Heapify root element
        heapify(arr, i, 0);
    }

    let end = performance.now();


    heapSortTimeTaken = (end-start)

    document.getElementById('heapsort').innerHTML = arr;
    document.getElementById('hs_time_taken').innerHTML = heapSortTimeTaken

}
```

```
var heapify = function(arr, n, i){

            // Find largest among root, left child and right child
    var largest = i;
    let l = 2*i + 1;
    let r = 2*i + 2;

    if (l < n && arr[l] > arr[largest])
        largest = l;

    if (r < n && arr[r] > arr[largest])
        largest = r;

    // Swap and continue heapifying if root is not largest
    if (largest != i)
    {
        var swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        heapify(arr, n, largest);
    }

}
```

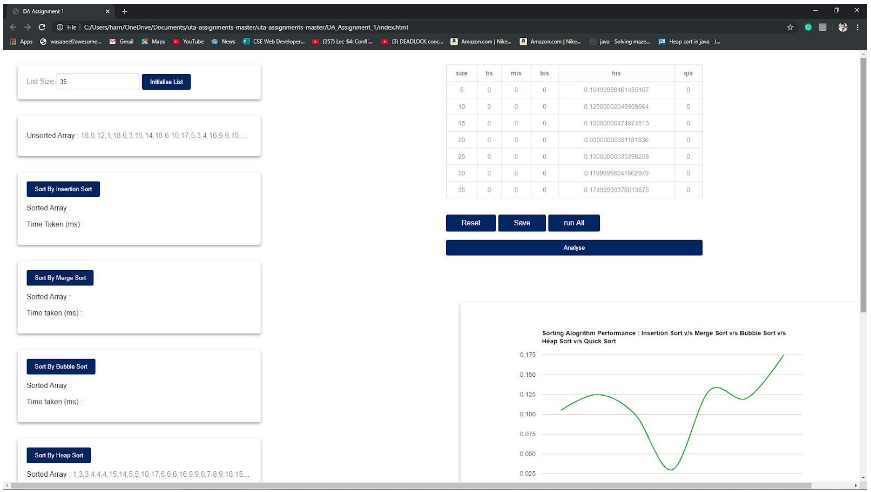
The heapify function is the process of converting a binary tree into a Heap data structure. A binary tree being a tree data structure where each node has at most two child nodes. A Heap must also satisfy the heap-order property, the value stored at each node is greater than or equal to it's children.

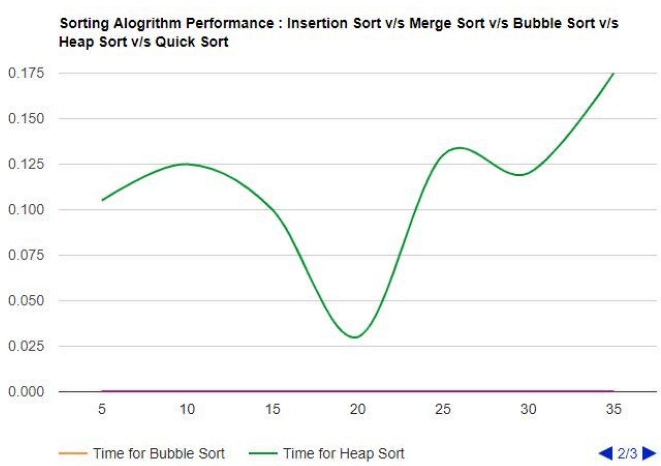
HTML code:

```
<br><br>
<div class="box">
    <button class="primaryBtn" onclick="heapSort()">Sort By Heap Sort</button>
    <p class="truncate">Sorted Array : <label id="heapsort"></label><br><br> Time taken (ms) : <label id="hs_time_taken"></label></p>
</div>

<br><br>
<div class="box">
```

When the button is pressed, the heapSort() is executed and the performance time is calculated.

OUTPUT:






The above output shows the various execution times for 5,10,15,20,25,30 and 35 unsorted elements and a graphical representation of its execution time.

### 4c. Quick Sort (Using median of 3):

Quicksort works recursively in order to sort a given array. These are the three basic steps of the Quicksort algorithm:

1. Partition the array into a left sub-array and a right sub-array, in which the items in the left sub-array are smaller than the specified item and the items in the right sub-array are greater than the specified item.
2. Recursively call the Quicksort to sort the left sub-array.
3. Recursively call the Quicksort to sort the right sub-array.

The partitioning step is the key, when sorting an array with Quicksort. Quicksort itself uses a Partition algorithm to partition the given array.

Median of 3:

The best approach to choose a pivot, is by choosing the median of the first, middle, and the last items of the array. This approach is known as the “median of three approach”.

Code:

```
var quickSort = function(){

    let array = [...unsortedArray];
    let length = array.length;

    let start = performance.now();


    recQuickSort(array, 0, length - 1);

    let end = performance.now();


    quickSortTimeTaken = (end-start)

    document.getElementById('quicksort').innerHTML = array;
    document.getElementById('qs_time_taken').innerHTML = quickSortTimeTaken




}

var recQuickSort = function(intArray, left, right){
    let size = right - left + 1;
    if (size <= 3)
        manualSort(intArray, left, right);
    else {
        let median = medianOf3(intArray, left, right);
        let partition = partitionIt(intArray, left, right, median);
        recQuickSort(intArray, left, partition - 1);
        recQuickSort(intArray, partition + 1, right);
    }


}
```

```
var partitionIt = function(intArray, left, right, pivot){

    let leftPtr = left;
    let rightPtr = right - 1;

    while (true) {
        while (intArray[++leftPtr] < pivot)
            ;
        while (intArray[--rightPtr] > pivot)
            ;
        if (leftPtr >= rightPtr)
            break;
        else
            swap(intArray, leftPtr, rightPtr);
    }
    swap(intArray, leftPtr, right - 1);
    return leftPtr;

}

var medianOf3 = function(intArray, left, right){

    let center = (left + right) / 2;

    if (intArray[left] > intArray[center])
        swap(intArray, left, center);

    if (intArray[left] > intArray[right])
        swap(intArray, left, right);

    if (intArray[center] > intArray[right])
        swap(intArray, center, right);

    swap(intArray, center, right - 1);
    return intArray[right - 1];

}

var manualSort = function(intArray, left, right){

    let size = right - left + 1;
    if (size <= 1)
        return;
    if (size == 2) {
        if (intArray[left] > intArray[right])
            swap(intArray, left, right);
        return;
    } else {
        if (intArray[left] > intArray[right - 1])
            swap(intArray, left, right - 1);
        if (intArray[left] > intArray[right])
            swap(intArray, left, right);
        if (intArray[right - 1] > intArray[right])
            swap(intArray, right - 1, right);
    }
}
```

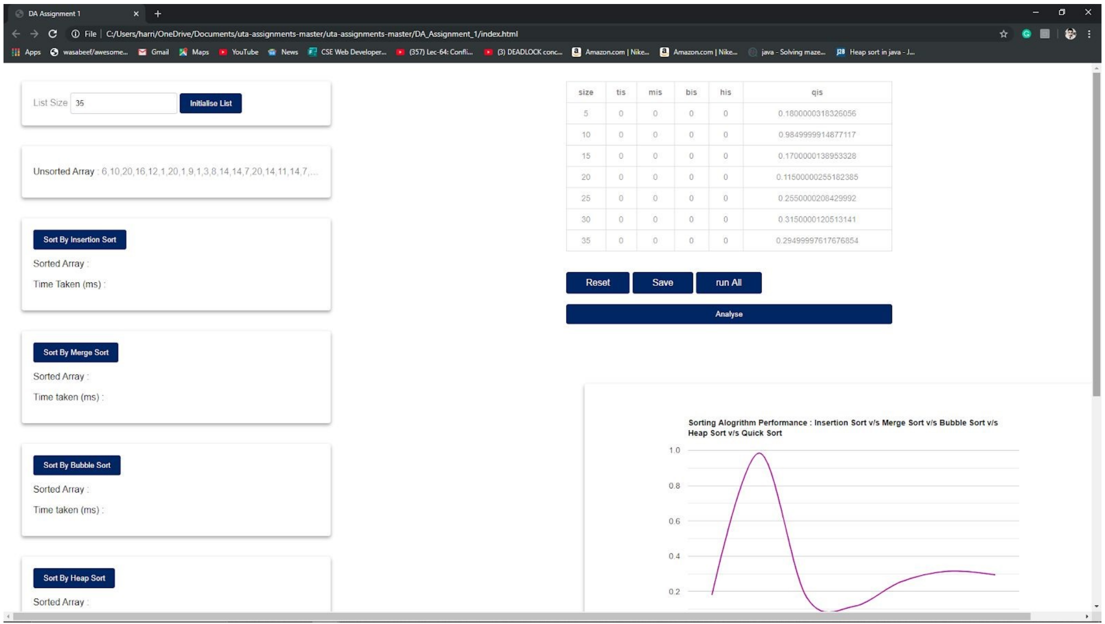
```
var swap = function(intArray, dex1, dex2){

    var temp = intArray[dex1];
    intArray[dex1] = intArray[dex2];
    intArray[dex2] = temp;
}

```

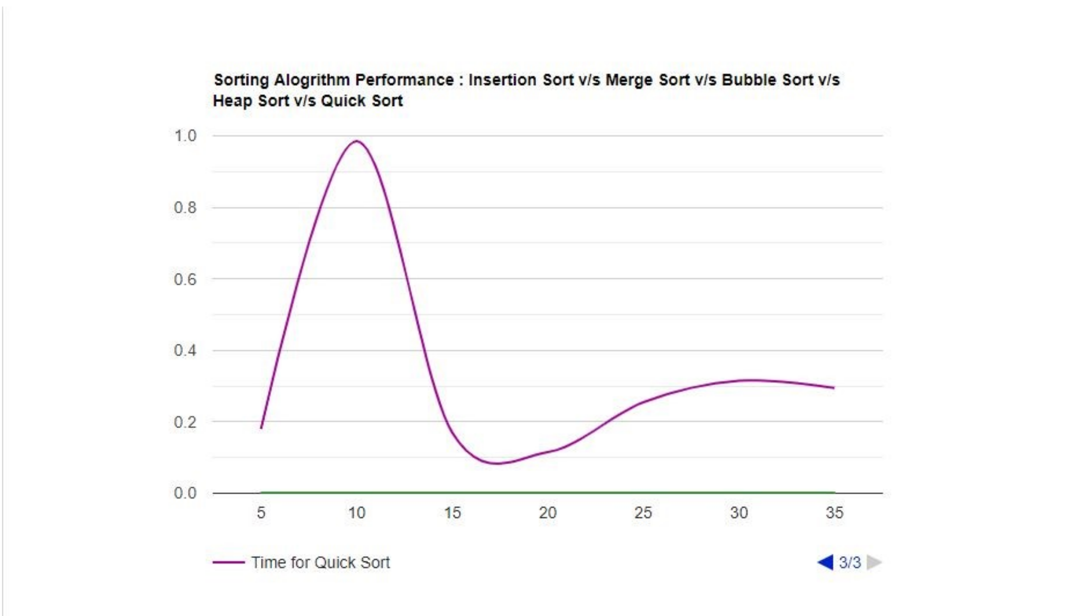
The above code runs the QuickSort() on the unsorted elements when the button is pressed.

HTML CODE:

```
<br><br>
<div class="box">
    <button class="primaryBtn" onclick="quickSort()">Sort By Quick Sort</button>
    <p class="truncate">Sorted Array : <label id="quicksort"></label><br><br> Time taken (ms) : <label id="qs_time_taken"></label></p>
</div>
<br><br><br>

```

OUTPUT:




The above is shows the execution time for 5,10,15,20,25,30 and 35 unsorted elements when it is sorted through quick sort.

### 4d. Insertion Sort:

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

CODE:

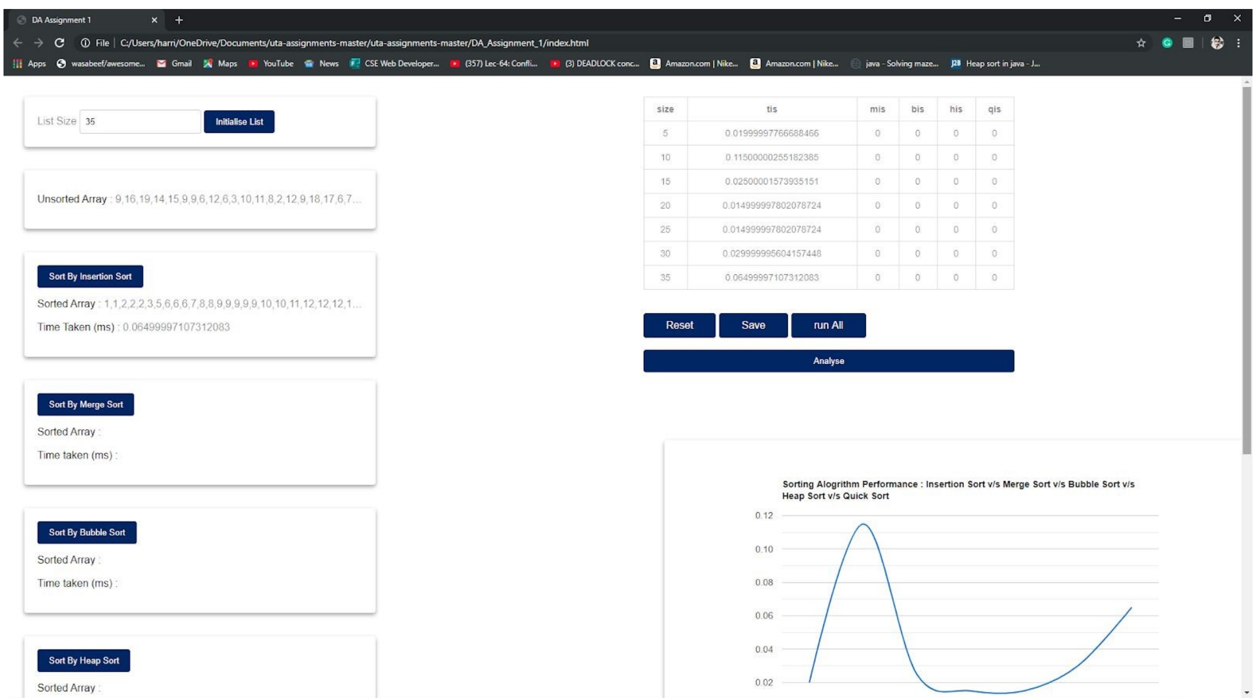
```
var insertionSort = function(){

    let array = [...unsortedArray];
    let length = array.length;

    let start = performance.now();

    for(var i = 1, j; i < length; i++) {
        var temp = array[i];
        for(var j = i - 1; j >= 0 && array[j] > temp; j--) {
        array[j+1] = array[j];
        }
        array[j+1] = temp;
    }

    let end = performance.now();

    insertionSortTimeTaken = (end-start)

    document.getElementById('insertionSort').innerHTML = array;
    document.getElementById('is_time_taken').innerHTML = insertionSortTimeTaken
}
```

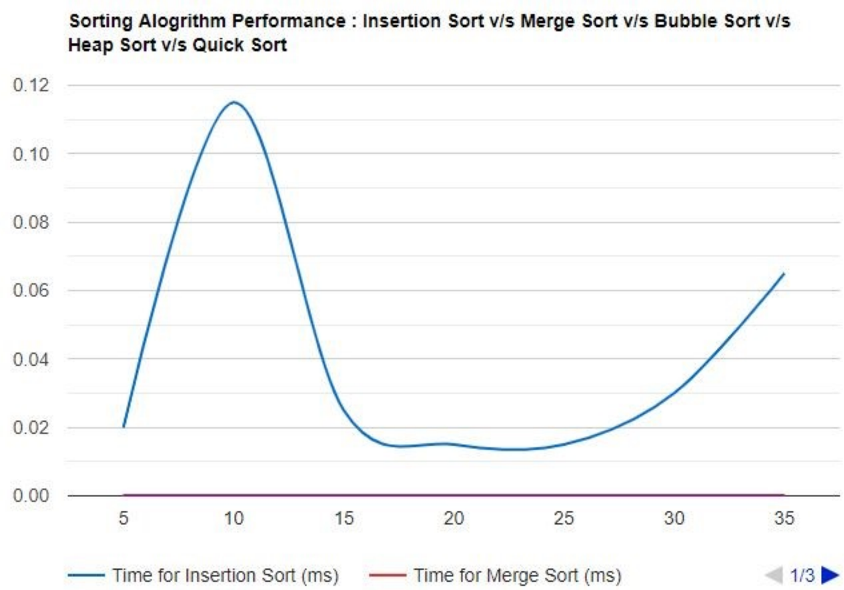
HTML code:

```
<br><br>
    <div class="box">
        <button class="primaryBtn" onclick="insertionSort()">Sort By Insertion Sort</button>
        <p class="truncate">Sorted Array : <label id="insertionSort"></label><br><br> Time Taken (ms) : <label id="is_time_taken"></label></p>
    </div>

    <br><br>
```

Pressing the button would sort 5,10,15,20,25,30 and 35 unsorted elements by invoking insertionSort().

OUTPUT:






**4e. Bubble Sort:**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

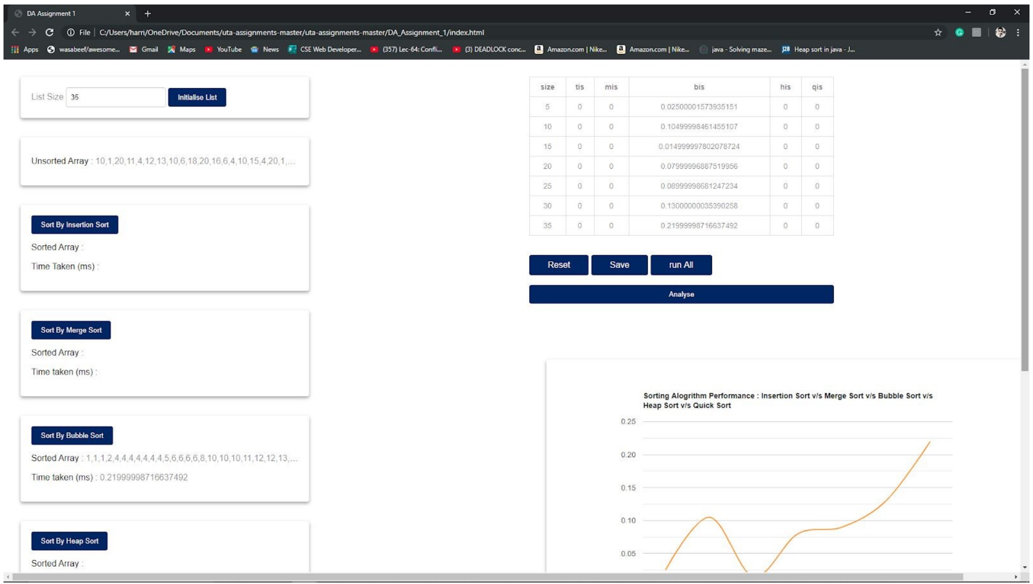
Code:

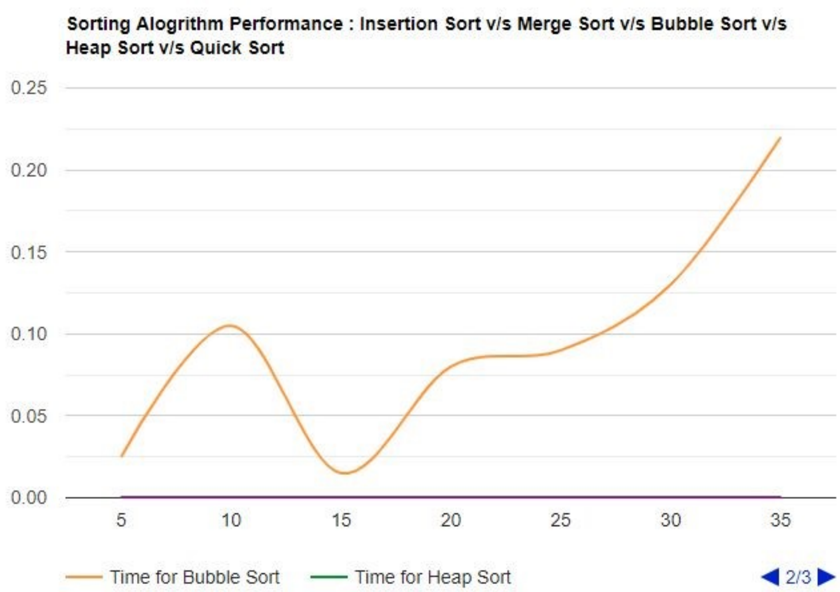
```
var bubbleSort = function(){

let array = [...unsortedArray];
let length = array.length;

let start = performance.now();

/*
for(var i = 1, j; i < length; i++) {
    var temp = array[i];
    for(var j = i - 1; j >= 0 && array[j] > temp; j--) {
    array[j+1] = array[j];
    }
    array[j+1] = temp;
}

*/




for(var i=0;i<length-1;i++){


    for(var j=0;j<length-i-1;j++){

        if(array[j]>array[j+1]){
            var temp=array[j];
            array[j]=array[j+1];
            array[j+1]=temp;

        }


    }



}

let end = performance.now();

bubbleSortTimeTaken = (end-start)

document.getElementById('bubblesort').innerHTML = array;
document.getElementById('bs_time_taken').innerHTML = bubbleSortTimeTaken
}
```

HTML code:

```
      <br><br>
    <div class="box">
        <button class="primaryBtn" onclick="bubbleSort()">Sort By Bubble Sort</button>
        <p class="truncate">Sorted Array : <label id="bubblesort"></label><br><br> Time taken (ms) : <label id="bs_time_taken"></label
    </div>
```

OUTPUT:



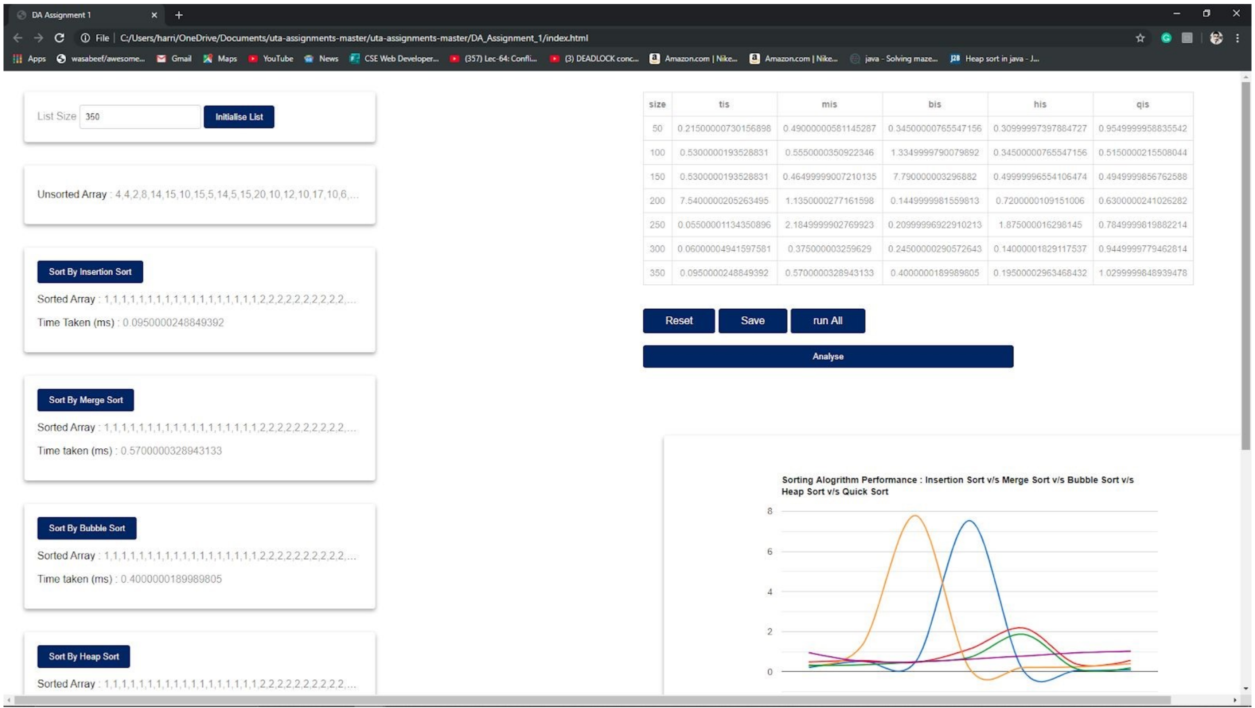




The above shows the execution time of 5,10,15,20,25,30 and 35 unsorted input elements and its graphical visualization.

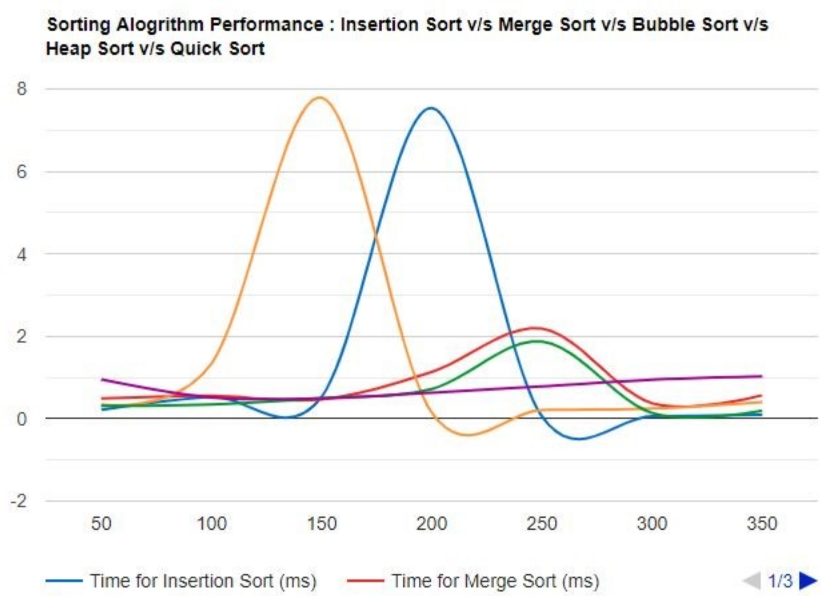
**4f. Sorting Alogrithm Performance : Insertion Sort v/s Merge Sort v/s Bubble Sort v/s Heap Sort v/s Quick Sort:**

Now, we are going to compare the various sorting algorithms and its execution times for larger unsorted input elements.

OUTPUT:



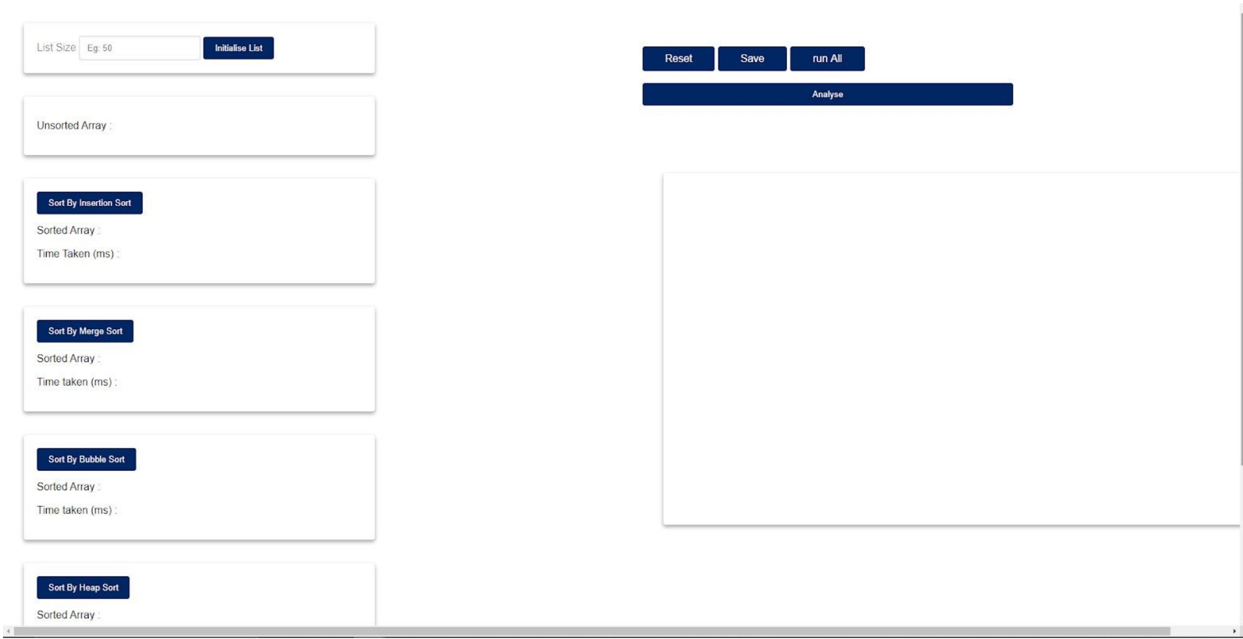
DA Assignment 1
C:/Users/harri/OneDrive/Documents/uta-assignments-master/uta-assignments-master/DA_Assignment_1/index.html
List Size
350
Initialise List
Unsorted Array : 4,4,2,8,14,15,10,15,5,14,5,15,20,10,12,10,17,10,6,...
Sort By Insertion Sort
Sorted Array : 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,...
Time Taken (ms) : 0.0950000248849392
Sort By Merge Sort
Sorted Array : 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,...
Time taken (ms) : 0.5700000328943133
Sort By Bubble Sort
Sorted Array : 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,...
Time taken (ms) : 0.4000000189989805
Sort By Heap Sort
Sorted Array : 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,...
size | tis | mis | bis | his | qis
50 | 0.2150000730156898 | 0.4900000581145287 | 0.3450000765547156 | 0.3099997397884727 | 0.9549999958835542
100 | 0.530000193528831 | 0.5550000350922346 | 1.3349999790079892 | 0.3450000765547156 | 0.515000215508044
150 | 0.530000193528831 | 0.4649999007210135 | 7.790000003296882 | 0.4999996554106474 | 0.4949999856762588
200 | 7.540000205263495 | 1.1350000277161598 | 0.1449999981559813 | 0.7200000109151006 | 0.630000241026282
250 | 0.0550001134350896 | 2.1849999902769923 | 0.2099996922910213 | 1.875000016298145 | 0.7849999819882214
300 | 0.0600004941597581 | 0.375000003259629 | 0.2450000290572643 | 0.1400001829117537 | 0.9449999779462814
350 | 0.095000248849392 | 0.570000328943133 | 0.400000189989805 | 0.1950002963468432 | 1.029999848939478
Reset
Save
run All
Analyse
Sorting Alogrithm Performance : Insertion Sort v/s Merge Sort v/s Bubble Sort v/s Heap Sort v/s Quick Sort

Sorting Alogrithm Performance : Insertion Sort v/s Merge Sort v/s Bubble Sort v/s Heap Sort v/s Quick Sort
8
6
4
2
0
-2
50
100
150
200
250
300
350
Time for Insertion Sort (ms)
Time for Merge Sort (ms)
1/3

From this, we can obtain this table comparing the various execution times of the sorting algorithms for 50,100,150,200,250,300 and 350 unsorted elements.

## 5. User-Interface:



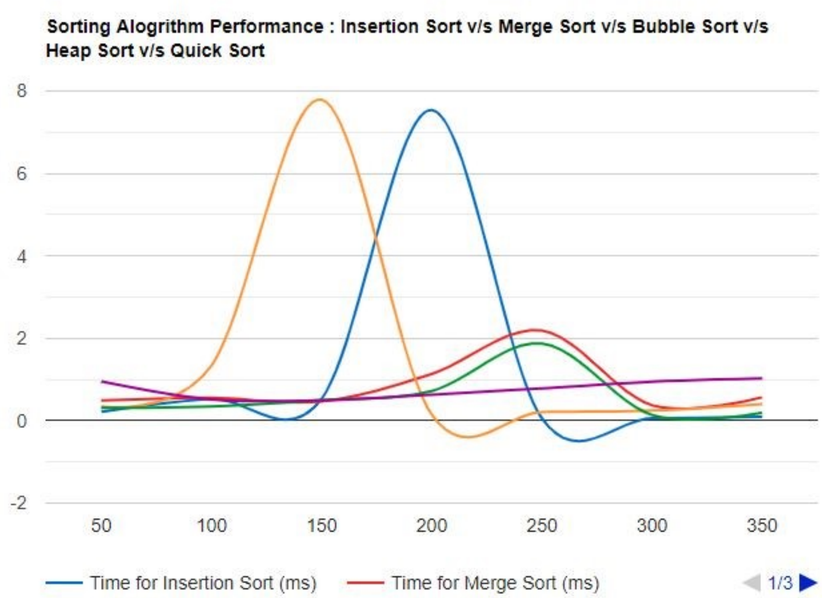

I have created a website using HTML, CSS and JS which provides an intuitive user-friendly experience. The only INPUT by the user is to indicate the size of the unsorted list.
From there, the initUnsortedArray() function takes that value as its input to generate a random unsorted list of values between 1 and 30 (as pre-determined by me).

The run All button can then be used to sort that unsorted list using all of the sorts and output the execution time in the table as well as producing a visualizable graph.

Every individual sort button can be pressed to execute its respective sort. The Reset button can also be pressed to clear everything that has been done.

## 6. Conclusion and Inference:

| SORTING ALGORITHM | | TIME COMPLEXITY | | SPACE COMPLEXITY |
|---|---|---|---|---|
| | BEST CASE | AVERAGE CASE | WORST CASE | WORST CASE |
| Bubble Sort | $\Omega(N)$ | $\Theta(N^2)$ | $O(N^2)$ | $O(1)$ |
| Selection Sort | $\Omega(N^2)$ | $\Theta(N^2)$ | $O(N^2)$ | $O(1)$ |
| Insertion Sort | $\Omega(N)$ | $\Theta(N^2)$ | $O(N^2)$ | $O(1)$ |
| Merge Sort | $\Omega(N \log N)$ | $\Theta(N \log N)$ | $O(N \log N)$ | $O(N)$ |
| Heap Sort | $\Omega(N \log N)$ | $\Theta(N \log N)$ | $O(N \log N)$ | $O(1)$ |
| Quick Sort | $\Omega(N \log N)$ | $\Theta(N \log N)$ | $O(N^2)$ | $O(N \log N)$ |




- When the array is almost sorted, insertion sort can be preferred.
- When order of input is not known, merge sort is preferred as it has worst case time complexity of nlogn and it is stable as well.
- When the array is sorted, insertion and bubble sort gives complexity of n but quick sort gives complexity of n^2.