Report

Introduction

Sorting is the process of taking in unordered elements and arranging those elements in an array so that they can be placed in ascending or descending order. Sorting data simplifies tasks and many computations. It is an essential data management task therefore sorting is a core topic in the study of algorithms.

The efficiency of a sorting algorithm is determined by the time and space complexity of the algorithm. Time complexity relates to the time it takes for an algorithm to execute with respect to the size of the input. It is commonly represented in the form of Big O Notation. This gives the worst-case complexity of an algorithm by representing the upper bound of the running time. It is an asymptotic notation that describes the running time of an algorithm when the input tends towards a particular value.

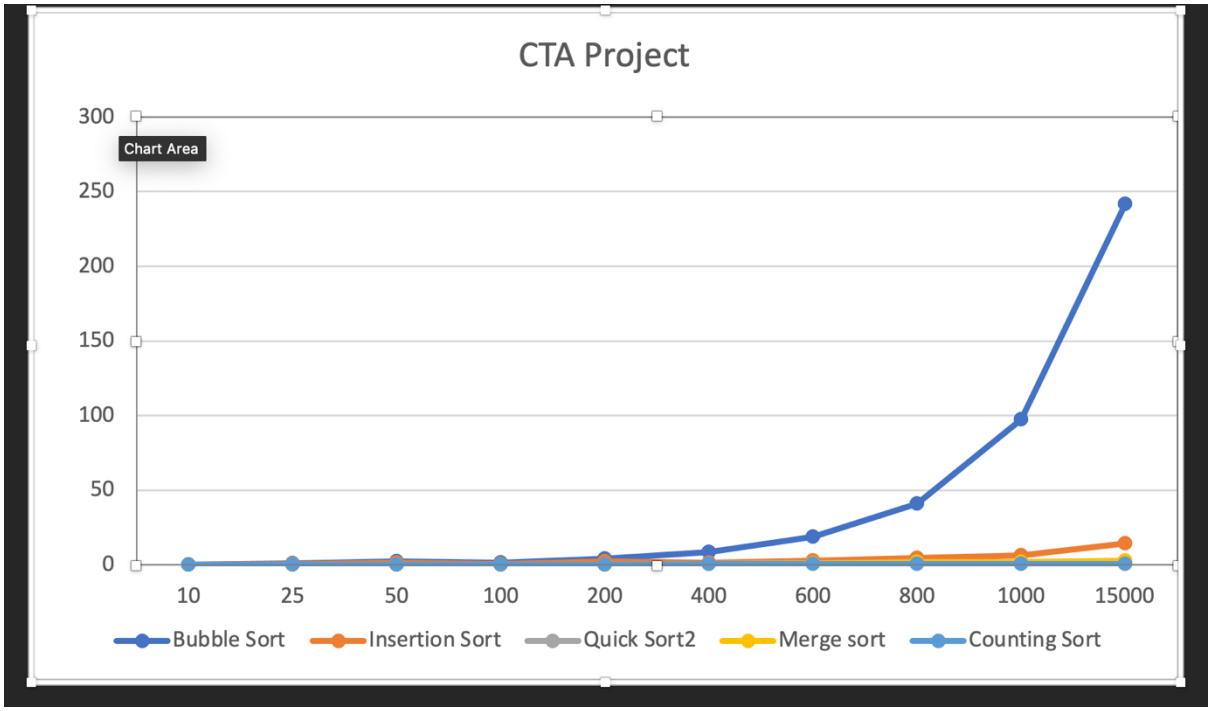
Space complexity relates to the total amount of memory used by an algorithm to complete execution. It includes both the auxiliary memory and input. A sorting algorithm is stable if two or more items with the same value maintain the same relative positions even after sorting. Below is a table with the stability of the algorithms used in this project.

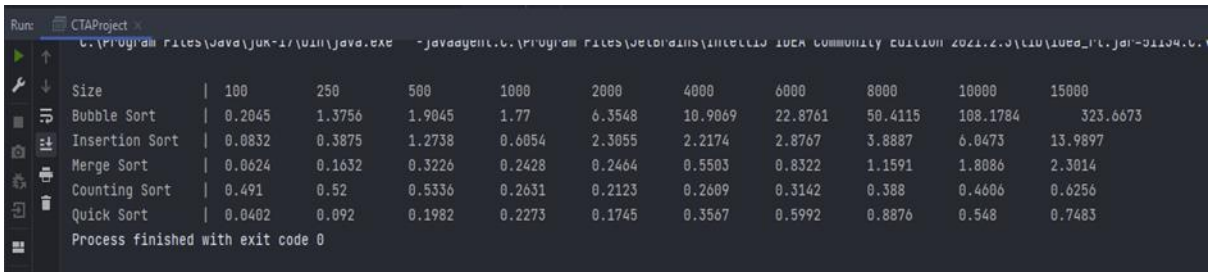
Sorting Algorithm	**Stability**
Bubble Sort	Yes
Insertion Sort	Yes
Merge Sort	Yes
Counting Sort	Yes
Quick Sort	No

Performance of an algorithm considers the amount of time each algorithm takes to complete in terms of the size of the original input N is expressed.

Sorting Algorithm	Avg & Worst	Best	Space Complexity
Bubble Sort	O(n^2)	O(n)	O(1)
Insertion Sort	O(n^2)	O(n)	O(1)
Merge Sort	O(n Log n)	O(n Log n)	O(n)
Counting Sort	O(k + n)	O(k + n)	O(n + k)
Quick Sort	O(n^2)	O(n Log n)	O(n)

Graphing allows us to visually compare algorithmic performance.





The scale along the x axis represents n while the y axis goes from 0 – 300. The value of n^2 anywhere on the x axis is always larger than O(n log n) while that functions output is always larger than n itself. We can infer that any function running in n^2 will take longer to complete than functions running in O(n log n) time. Likewise functions running in the order of O(n log n) are slower than those running in linear time O(n).

In place sorting does not use extra memory to sort an array. Sorting algorithms sometimes requires some additional space for comparison or to temporarily store data elements. These types of algorithms carry out their sorting in what is known as in-place or happening within the array. An example of in-place sorting is a bubble sort.

Stable sorting refers to a sorting algorithm that does not change the sequence of similar content after sorting. Several of the sorting algorithms used in this project are stable by nature, such as Merge sort, Counting sort, Insertion sort and Bubble sort. Quick sort is unstable, but we can modify it by using extra space to maintain stability.

In Java the comparator function orders collections of objects that don't have a natural ordering. It is included in the java.util package and contains two methods compare and equals. It is an interface used for rearranging an arraylist to a sorted manner. To do this, we must override the compare() method and call collections.sort(). Comparator is useful as it allows multiple sorting sequence. We can sort the collection based on multiple elements such as id, itemName, price.

To conclude this introduction, we will look at comparison and non-comparison-based sorting. Most of the sorting algorithms chosen for this project come from comparison-based sorting. This method intuitively compares elements of an array to find the sorted array. Bubble sort, insertion sort, merge and quick sort all belong to this category. Bubble and insertion have a worst-case time complexity of n^2 while the best case in n assuming the array is already sorted. Merge sorts time complexity is nlogn in every case and quick sort uses a divide, conquer, and compare approach with recurrence relation. Its worst case returns n^2and its best/average will be sorted in nlogn.

Non-comparison-based sorting does not compare elements to sort an array. Counting sorts time complexity is n+k where k is the size of count array. When you have an array that is near sorted, Insertion sort is preferable. Whereas, if the order is unknown – merge sort is best suitable as its worst-case time complexity is nlogn and it is stable. If the array is sorted, bubble sort and insertion sort give linear complexity while quick sort gives n^2 complexity.

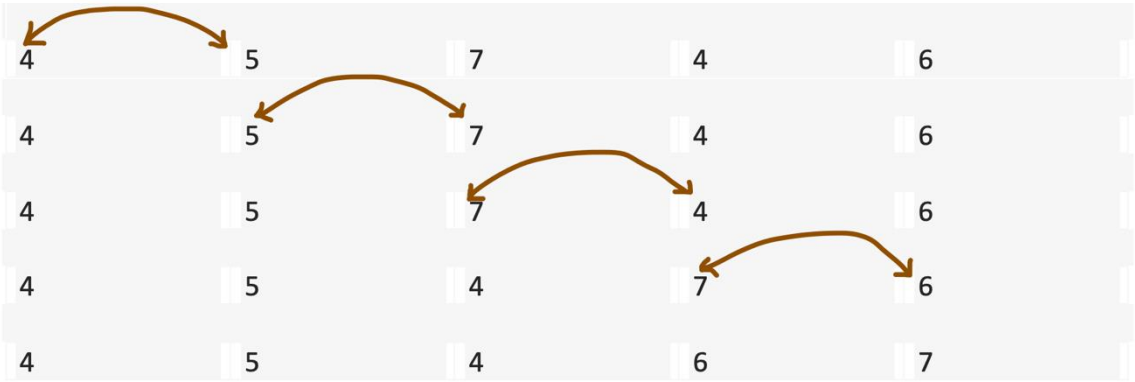
According to Fenyi et al, non-comparison-based algorithms are a better choice compared to comparison based in terms of time and space complexity. The high number of swapping elements in the array result in high time complexity. As the size of the array grows more swapping of elements is required increasing execution time further. For example, a bubble sort algorithm will never stop iterating until the highest number moves furthest right. Their studies also posit that non-comparison-based algorithms use less memory. Quick sort for example utilized the largest memory capacity due to its recursive nature when using pivot for each sub array.

Sorting Algorithms

Bubble Sort: (1962 Iverson)

This sorting algorithm is the easiest sorting algorithm, and it is regularly used to introduce the concept of sorting. The algorithm moves through a sequence of input data and rearranges them into ascending and descending order. The algorithm compares two adjacent pairs of elements in the array. If the elements are in an incorrect order, they are moved so that the largest moves right. This process continues until the array is sorted and the highest number is furthest right. The number of passes or each time bubble sort algorithm goes through the data element list can be represented by n – 1. N being the number of elements in the array.

Sorting Algorithm	Avg & Worst	Best	Space Complexity
Bubble Sort	O(n^2)	O(n^2)	O(1)



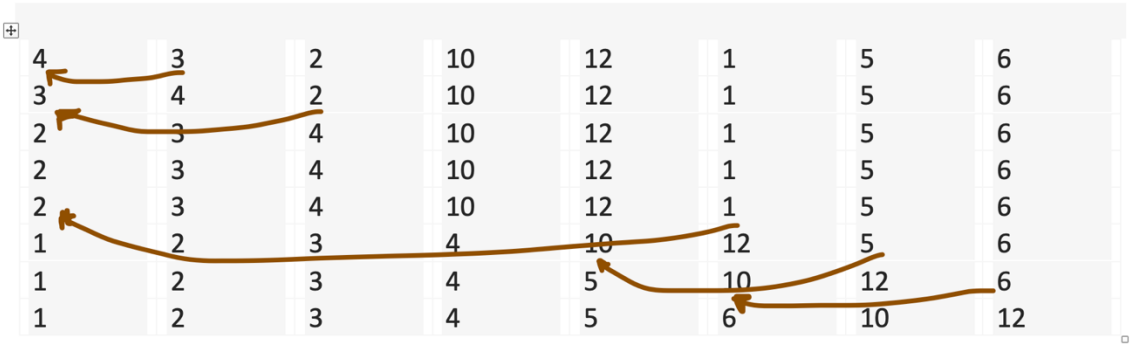
To implement the Bubble sort algorithm, we have int i that = 0, and j that varies from 0 – 5. Temp is where we put things when we want to swap two variables. Next, we have a loop within a loop where if compares the adjacent elements and swaps them if true. If no two elements were swapped by the inner loop, then break.

```
static int[] bubbleSort(int[] arr) {
    int n = arr.length;
    int i, j, temp;
    boolean swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // swap arr[j] and arr[j+1]
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }

        // If no two elements were swapped by inner loop, then break
        if (!swapped)
            break;
    }
    return arr;
```

Insertion Sort: (1946 John Mauchly)

Insertion sort is a simple inefficient sorting algorithm that can be likened to manually sorting a hand of playing cards. To begin you always start your first iteration with index of 1 not with 0. You start with one and check the previous number. If the previous number, it larger you swap with index 1. Now compare third element with second and move right as appropriate. Then check second element with first in the array. Essentially you are moving each element in turn towards the start, checking each as you progress. Like bubble sort, if you double the number of elements, it will quadruple the running time. With insertion sort the run time will be shorter to begin with. This will take O(n) time when the array is sorted otherwise it will be n^2.

Sorting Algorithm	Avg & Worst	Best	Space Complexity
Insertion Sort	O(n^2)	O(n)	O(1)



To implement the Insertion Sort algorithm, we begin by traversing the array from 1 to n – 1. We next compare the element with its predecessor. If the element is smaller, then we move previous element up by one place and insert the current element at previous elements position. We then print the sorted array once the outer loop is completed.

```
static int[] insertionSort(int[] arr)
{
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
```

Inside the InsertionSort function we must traverse from 1 to n – 1. The current element is the key and the previous element is j = i – 1. We move elements of arr[0...i – 1], that are greater than the key, to one position ahead of their current position.

```
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
    return arr;
}
```

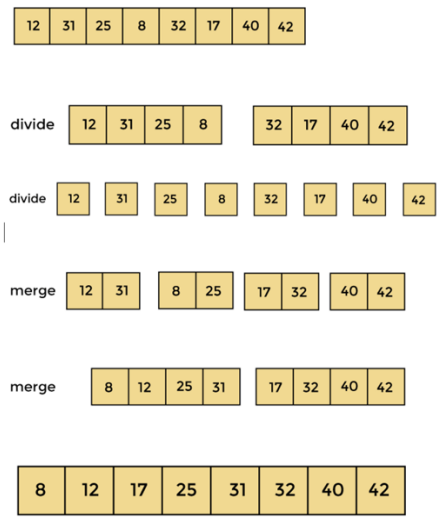
J + 1 that is the next position that becomes the current position of the previous element. The key then becomes the next element. These steps will continue until the entire array is traversed.

Merge Sort: (1945 John Von Neumann)

Merge sort like quick sort is a divide and conquer sorting algorithm. It works by dividing into two halves, calling itself for each half and then it uses the merge() function to merge the two halves together. Merge (l,m,r) assumes that the two arrays are sorted and merges the two sub arrays to create one single sorted array.

Sorting Algorithm	**Avg & Worst**	**Best**	**Space Complexity**
Merge Sort	O(n Log n)	O(n Log n)	O(n)

Merge sort is a recursive algorithm expressed as following recurrence relation. Time complexity of merge sort is O(n Log n) in worst, avg and best case as it always divides the array and linearly merges both halves. This algorithm would be considered stable and in a typical implementation it would not sort in place. Its temp array requires auxiliary space making its space complexity O(n).



To implement merge sort, we declare a left and right variable to mark the extreme indices of the array. Left is assigning 0 and right is assigned n – 1. We mind mid by (left + right)/2. Next we call mergeSort on (left,mid) & (mid + 1 rear). These steps continue until sub problems are sorted. Finally, we merge the two sub problems to form a sorted array.

```java
static void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        // Find the middle point
        int m =l+ (r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m); // left most to middle
        mergeSort(arr, m + 1, r); // middle + 1 to rightmost

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

```java
static void merge(int arr[], int l, int m, int r)
{
    // Find sizes of two subarrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[] = new int[n1];
    int R[] = new int[n2];

    /*Copy data to temp arrays*/
    for (int i = 0; i < n1; ++i)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[m + 1 + j];
```

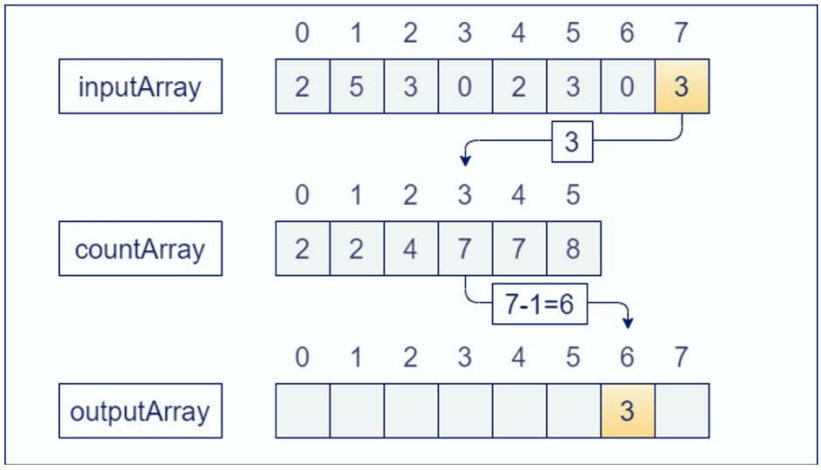
```java
static int largest(int[] arr)
{
    int i;
    // Initialize maximum element as first element of array
    int max = arr[0];
    // Traverse array elements from second position and compare every element with current max
    for (i = 1; i < arr.length; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Counting Sort:
(1954 Harold H Seward)

This sorting technique uses keys with specific ranges by counting the number of objects with distinct values, like hashing. Then finally we calculate each objects position in the output sequence. We do this by indexing the input data and using a count array to store the count of each distinct object. You would then modify the count array so that each element at each index stores the sum of the previous counts. The modified count array indicates the position of each object in the output sequence. Finally, we output each object from the input sequence followed by increasing its count by 1.

Sorting Algorithm	**Avg & Worst**	**Best**	**Space Complexity**
Counting Sort	O(k + n)	O(k + n)	O(n + k)

Time complexity for counting sort is O(k + n) where n is equal to the number of elements in the input array and k is the range of the input. Its auxiliary complexity is also O(n + k) .



To implement Counting sort, you create a count array to store the count of each unique object. Then modify the count array such that each element at each index stores the sum of the previous counts. The modified count array indicates the position of each object in the output sequence. After rotating the array clockwise for one time you output each object from the input sequence followed by increasing its count by 1.

```
static int[] countingSort(int[] input) {
    int k = largest(input);
    int[] c = countElements(input, k);

    int[] sorted = new int[input.length];

    // Once you have the element counts, place elements on their positions
    // and decrement current value in count array by 1
    for (int i = input.length - 1; i >= 0; i--) {
        int current = input[i];
        sorted[c[current] - 1] = current;
        c[current] -= 1;
    }

    return sorted;
}
```

```
static int[] countElements(int[] input, int k) {
    int[] count = new int[k + 1];
    // Fill count array with 0
    Arrays.fill(count, val: 0);

    // Count occurrence of each element in the input array and save in the count array
    for (int i : input) {
        count[i] += 1;
    }

    // Modify count array as addition of preceding counts
    for (int i = 1; i < count.length; i++) {
        count[i] += count[i - 1];
    }

    // return the final count array
    return count;
}
```

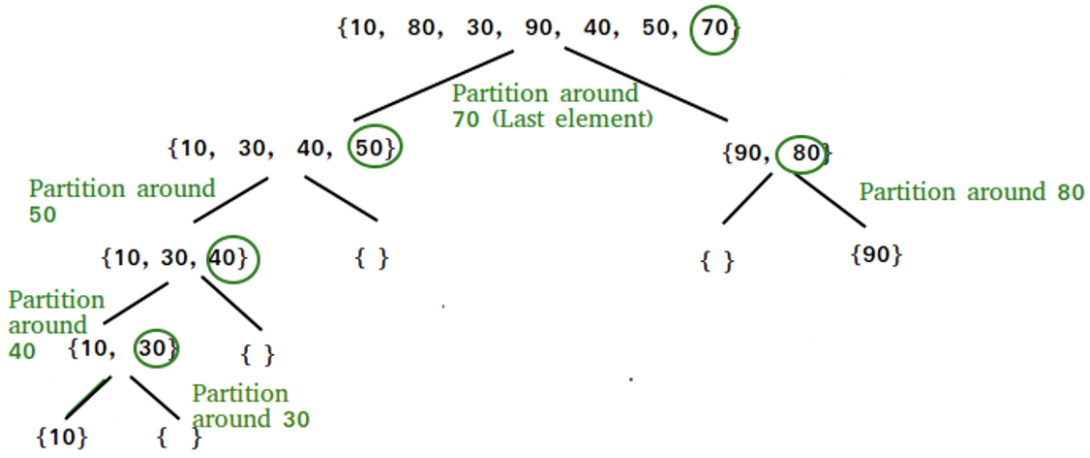
Quick Sort: (1962 C.A.R. Hoare)

This sorting algorithm also leverages the divide and conquer principle. Its best time complexity is O(n log n) and it is commonly used when sorting big data volumes. Quicksort is not a stable algorithm as it does as the elements with the same values do not appear in the same order in the sorted output as

they do in the input list. The input list here is divided into two sub-lists by the pivot. One sub-list with elements less than the pivot and another sub-list for elements greater than the pivot. Each sub class repeats this process. When all sub lists are sorted they merge to form the final sorted output.

Sorting Algorithm	Avg & Worst	Best	Space Complexity
Quick Sort	O(n^2)	O(n Log n)	O(n)

In the best case, Quick sort divides the list into two equal size sub lists. The first iteration of the full n sized lists will be done in O(n). Sorting the remaining two sub-lists takes O(n/2) each. As a result, the algorithm has the complexity of O(n log n). In the worst case, Quick sort only selects one element in each iteration equaling O(n^2). On the average Quick sort will have O(n Log n) complexity making it suitable for big volumes of data.

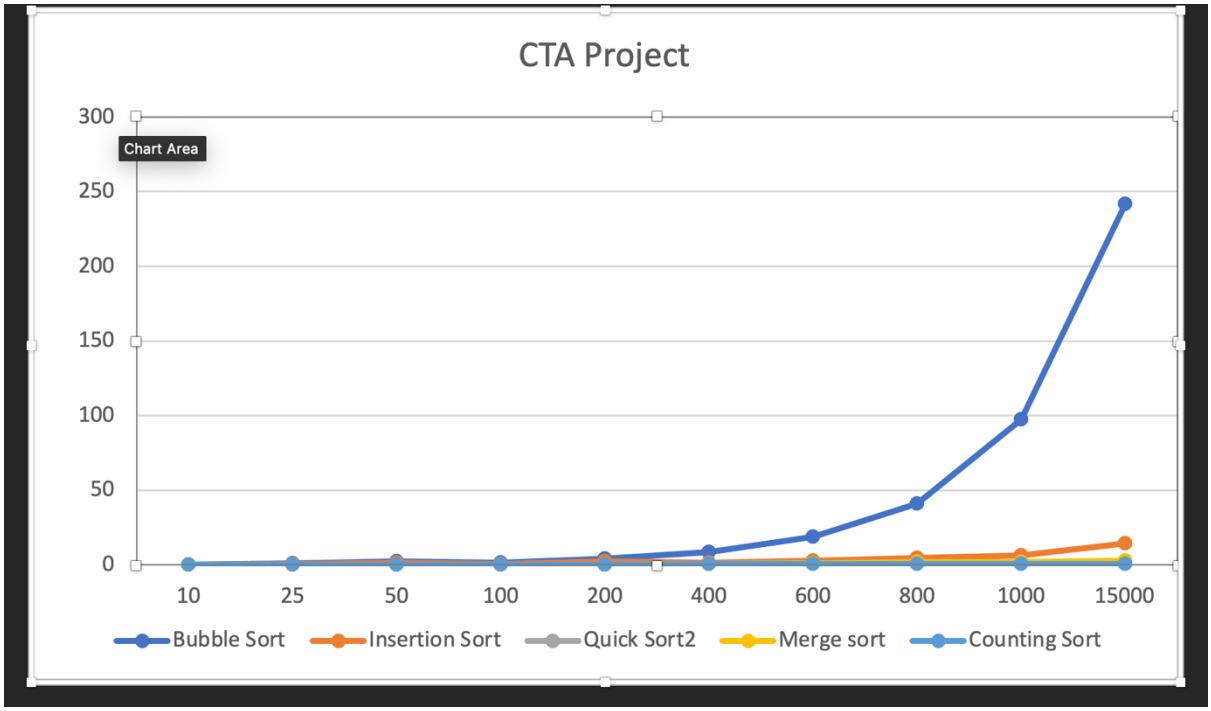


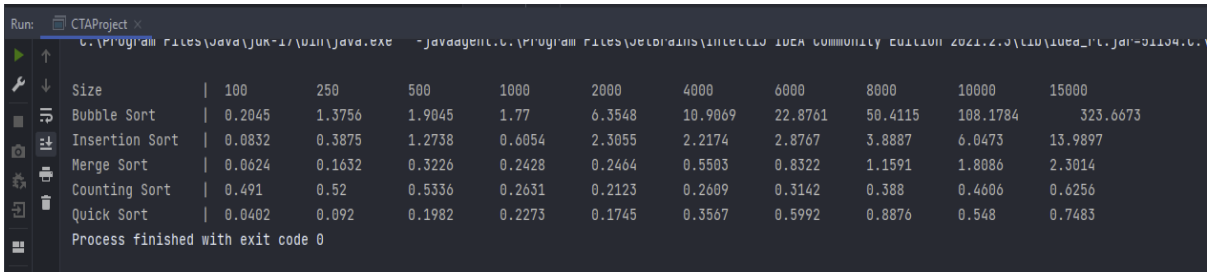
Like Merge sort, QuickSort is a divide and conquer algorithm. It picks a pivot and partitions a given array around the picked array. The key process in quicksort is partition() which is done in O(n) linear time. The logic is to start at the leftmost element and keep track of index of smaller elements. While traversing, if the smaller element is found it is swapped with current element with arr[i]. Otherwise the current element is ignored.

```java
static int partition(int[] arr, int low, int high)
{

    // pivot
    int pivot = arr[high];

    // Index of smaller element and indicates the right position of pivot found so far
    int i = (low - 1);

    for(int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            // Increment index of smaller element
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return (i + 1);
}
```

```java
static void quickSort(int[] arr, int low, int high)
{
    if (low < high)
    {

        // pi is partitioning index, arr[p] is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before partition and after partition
        quickSort(arr, low, high: pi - 1);
        quickSort(arr, low: pi + 1, high);
    }
}
```

Implementation and Benchmarking:





The bubble sort algorithm used in our performance benchmark has a predictably simple inefficient computational complexity of O(n^2). Meaning, if bubble sort has an input data size of 15, it will take a time factor of 15 squared to run, which is 225. Bubble sort increases at the fastest rate followed by insertion sort as these algorithms both display O(n^2) growth rate. It is hard to distinguish Counting Sort, Merge Sort and quick Sort given the scale on the y-axis is dominated by the slow times for bubble sort. The difference between bubble sort and insertion sort lies in the number of comparisons. Specifically, bubble sort carries out many comparisons that may not result in a swap. Overall, the results produced what we would expect. Quicksort is the overall leader having a very small execution time compared to every other algorithm apart from counting sort which came in at a close second place. Merge sort performed well almost on par with the two best performing algorithms.