Computational Thinking with Algorithms Project 2024

Table of Contents

Introduction:

The important aim of this project is to know how much role these different sorting algorithms play in the current era and their use and to gain knowledge about them. In this "Computational Thinking with Algorithms" module we were given information on several sorting algorithms during the lecture series through online, establishing a foundation on this topic. However, before I move forward with this project, I want to do deeper research on the sorting algorithms I have chosen to compare. It will allow me to deeper understanding of sorting algorithms functionality and gives me prior knowledge of sorting algorithms current performance before I move forward and test them.

The concept of What is Sorting: To sort something refers to "arrange things systematically in groups or separate things into groups according to their type" (*Oxford Dictionary and Google, 2024*)

Neskkids.com.au 2024

Camelino.gr 2024

Above Pictures showing simple Sorting.

Judging by these simple toys sorted by young children above, humans have a great sorting ability that makes sorting very easy and clever. But this not the case with computers. A computer program must follow an order of sequence of instructions to achieve the same result. So, we use these algorithms to implement sorting in a computer-based environment. If we apply the same concept to an algorithm as we did earlier when defining what is meant by the term "sorting", *"In computer science, a sorting algorithm is an algorithm that puts elements of a list into an order" (Wikipedia, 2024).*

Sorting Algorithms:

There are several types of sorting algorithms. The main 3 types of sorting algorithms are:

1. **Comparison based Sorting Algorithms**:
   An algorithm used to sort a group of elements or objects by comparing them to one another is called a comparison-based sorting algorithm. It can read only the elements of a list through a single abstract comparison operation. It is often a "<" or "=" operator or a three-way comparison, that determines which of two elements should occur first in the final sorted list. The only requirement is that the operator forms a total preorder over the data, with: *(Wikipedia, 2024)*
      If $a \leq b$ and $b \leq c$ then $a \leq c$ (Transitivity)
      For all a and b, $a \leq b$ or $b \leq a$ (connexity).

   It is possible that both $a \leq b$ and $b \leq a$; in this case either may come first in the sorted list. In a stable sort, the input order determines the sorted order in this case. For comparison-based sorts of the decision to execute basic operations other than comparisons is based on the outcome of comparisons. Thus, in timing analysis, the number of executed comparisons is used to determine upper bound estimates for the number of executed basic operations such as swaps or assignments. *(Wikipedia, 2024)* Some examples of the comparison-based sorting algorithms are Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Quick Sort.

2. **Non – Comparison based Sorting Algorithms:**

   The algorithms that sort the data without comparing the elements are called non-comparison sorting algorithms. A non-comparison sort algorithm uses the internal character of the values to be sorted. It can only be applied to some cases and requires particular values. And the best complexity is probably better depending on cases, such as O(n). (*stackoverflow, 2024*) Some examples of the non-comparison-based sorting algorithms are Counting Sort, Bucket Sort and Radix Sort. Counting Sort is a linear sorting algorithm that sorts in O(n+k) time when elements are in the range from 1 to k. Radix Sort uses Counting Sort as a subroutine to sort.

3. **Hybrid Sorting Algorithms**:
   A hybrid sorting algorithm is one which combines two or more algorithms which are designed to solve the same problem. Hybrid algorithm either chooses one specific algorithm depending on the data and execution conditions, or switches between different algorithms according to some rule set. These algorithms aim to combine the desired features of each constituent algorithm to achieve a better algorithm in aggregate. Some examples of the non-comparison-based sorting algorithms are Intro Sort and Tim Sort. (*Dr. Carr, 2024*)

Performance:

The sorting algorithm depends on how much time/memory/disk/... is used when a program is run. And this depends on the speed of the computer and quality of the compiler as well as the code. Algorithms are platform independent. Any algorithm can be implemented in any programming language on any computing hardware running any operating system results obtained depend on the hardware and software used. We can compare algorithms by evaluating their running time complexity on input data of size n. Performance determines which algorithms scale well to solve problems of a nontrivial size, by evaluating the complexity the algorithm in relation to the size n of the provided input. (*Dr. Carr, 2024)*

In-place Sorting:

An in-place sorting algorithm is a sorting algorithm that sorts the input array in place, without using any additional memory. This means that the algorithm does not need to create a new array to store the sorted elements. In-place sorting algorithms are often used in applications where memory is limited. For example, in-place sorting algorithms are often used to sort lists of elements on a computer's memory. (*medium.com, 2024)*

Some of the examples of in-place sorting algorithms include:

- **Bubble Sort**: An in-place sorting algorithm that rearranges elements within the original array by comparing adjacent elements and swapping them if they are in the wrong order.
- **Selection Sort**: Also, an in-place sorting algorithm that uses a "gap" sequence to sort elements that are far apart and gradually reduces the gap until the entire list is sorted. (*medium.com, 2024*)

Stable Sorting:

In a stable sorting algorithm, data is sorted in a way that preserves the original order of elements having equal keys. This means that if two elements have the same key, the one that appeared earlier in the input will also appear earlier in the sorted output. For instance, if element A[i} and element A[j] have the same key and i < j, then in sorted order, A[i] should come before A[j]. (*educative.io, 2024*)

Stable Sorting Algorithms: Some of the examples of the most common Stable Sorting Algorithms include: (*educative.io, 2024*)

- **Bubble sort**: It iterates the array repeatedly until completely sorted, compares the adjacent elements in each iteration, and swaps them if they are not in the correct order. Its time complexity is $O(n^2)$.
- **Insertion sort**: It divides the array into sorted and unsorted portions. It compares the unsorted elements with the sorted

elements and places them in their correct positions. Its time complexity is O(n^2).

- **Merge sort**: It divides the dataset into smaller subarrays, sorts them individually, and then merges them to obtain the final sorted result. Its time complexity is O(n log n).
- **Counting sort**: It counts the occurrences of the elements in the dataset and uses this information to sort them in increasing order. Its time complexity is O(n + b).
- **Radix sort**: It sorts the numbers by sorting each digit from the left to right, resulting in the sorted data. Its time complexity is O(d*(n + b)). (*educative.io, 2024*)

Efficiency/Complexity (Time and Space):

Time and Space efficiency are the key variables looked at when evaluating efficiency of the algorithm.

Space efficiency looks at the computer's ability to run the algorithm in terms of the computational amount of memory and space/storage available. (*Dr. Carr, 2024*)

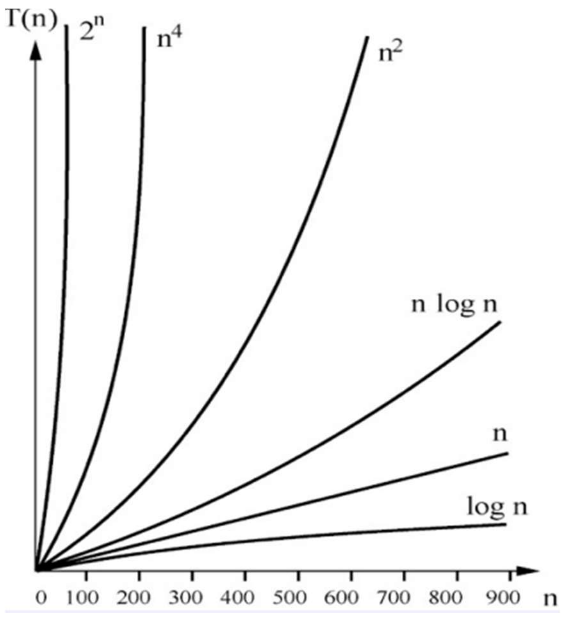
Time efficiency is evaluated by considering the length of time or number of operations required for a computer to run an algorithm. Algorithms can be compared to each other by evaluating their running time complexity on input data size 'n'. (*Dr. Carr, 2024*)

The efficiency of an algorithm with respect to internal factors such as the amount of time it takes to run is measured by its complexity. The complexity of the sorting algorithm effects its performance as the problem being solved or input dataset gets larger. Algorithm complexity results in the algorithm falling into one of number families which is determined by the growth in its execution time with respect to increasing input size 'n' is of a certain order. To evaluate algorithmic complexity the most expensive computation must be identified to determine classification. (*Dr. Carr, 2024*)

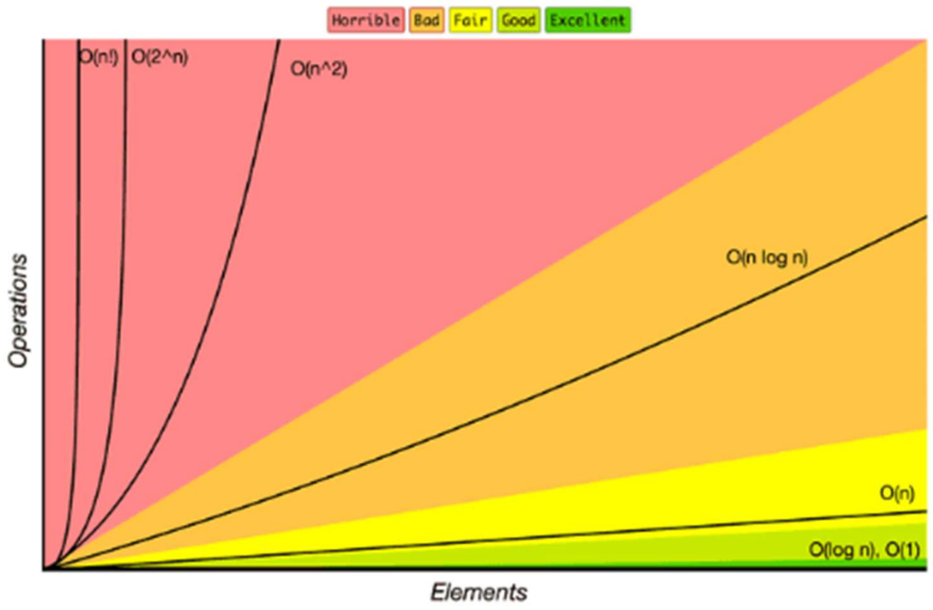
Complexity Curves:

Running time T (n) is proportional to:	Complexity
$T(n) \propto \log n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n \log n$	linearithmic
$T(n) \propto n^2$	quadratic
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n$; $k > 1$	exponential

Decreasing in efficiency



Above image showing Running times in relation to Complexity (*cs.auckland.ac.nz, 2024*)



(Dr. Carr, 2024)

There are 3 types of cases when analysing the complexity of an algorithm. These three cases can be described by Big O Notation. Another name for Big O notation is asymptotic notation. Big O notation measures how quickly a function grows and declines.

- **Best Case**: It defines the input for which the algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. (*Geeksforgeeks,2024*) Best case is rare. We can use Ω notation for best case complexity. It also displays a "linear growth in execution". (*Dr. Carr, 2024).*
- **Average Case:** It defines the expected behaviour when executing the algorithm on random input sizes. Some of the inputs required greater time to complete the algorithm because of some special cases, but majority of them don't need that much time to complete. The theta 'Θ' notation defines the average case of an algorithm's time complexity. (Dr. Carr, 2024)
- **Worst case:** It defines the input for which algorithm takes a long time or maximum time. In the worst-case scenario, the slowest time to complete an algorithm. In this case, it will take maximum time to execute the algorithm. Worst case behaviour is the easiest to calculate. We use Big O

notation in the worst-case scenario to describe the complexity of an algorithm. (*Dr. Carr, 2024*)

Comparator Functions:

A comparison function, which imposes a total ordering on some collection of objects. They can be passed to a sort of method. E.g. collections.sort or Arrays.sort. Comparators can also be used to control the order of certain data structures e.g. sorted sets or sorted maps, or natural order. (*docs.oracle.com, 2024)*

In comparator function, it can be a 3-way comparator that returns an integer, or a 2-way comparator that returns a Boolean. The default comparator compare works well for sorting numbers in increasing order, or strings, keywords, or symbols, in lexicographic order, and a few other cases. (*Closure, 2024)*

Sorting Algorithms:

This project focuses on evaluating the space and time complexity of five chosen sorting algorithms. To perform this benchmarking application, we were given the task to design and build a Java application that would do this.

The 5 Sorting Algorithms I have chosen to benchmark in this Project are:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Merge Sort
5. Counting Sort

In this aspect of the project, I will discuss each of the sorting algorithms separately under these topics:

a. Introduction to the chosen Sorting Algorithm
b. Its Space and Time Complexity
c. How the Sorting Algorithm works (with diagrams) using my student ID
d. Code

1. Bubble Sort:

Bubble Sort algorithm is a simple comparison-based sorting algorithm. In this algorithm, smaller values bubble up to the starting of the array, towards index 0 and larger values bubble up to the end of the array, towards index N-1. It was given the name **Bubble sort** to describe the way larger values "**bubble up**" to the end of the list during the sorting process. Bubble sort was created in 1956. It works by repeatedly swapping the two adjacent elements if they are in the wrong order. So that they are in the correct order. Bubble sort algorithm is an in-place and stable sorting algorithm. It is easy to understand and implement, but it is very slow and impractical choice of sorting algorithm for most of the problems. (*Dr. Carr*) Bubble sort is basically based on the idea of repeatedly swapping pairs of elements that is compared to each other until the entire input size is sorted.

Its Time and Space Complexity:

Its time complexity is 'n' in best case, and 'n^2' in worst and average cases.

Specifications Big O Notation

Best Case	Average Case	Worst Case	Space Complexity	Stable
O(n)	$O(n^2)$	$O(n^2)$	1	Yes

Above Table showing Bubble Sort Specifications (*Dr. Carr and geeksforgeeks, 2024*)

Best Case: In this scenario, where the array of elements is already sorted prior to running the sorting algorithm. Bubble sort performs O(n) comparisons.

Average & Worst Case: occur if the input array was in reverse order prior to sorting the input set. Bubble sort performs $O(n^2)$ comparisons in average and worst-case scenarios.

<u>Big O Analysis of Bubble Sort</u>: It is very slow and impractical choice of sorting algorithm for most of the problems. (*Dr. Carr, 2024*) These problems are arisen with greatly unsorted data and large input sizes. Bubble sort is not the most efficient sorting algorithm compare with other algorithms. Because it uses nested loops during iteration. Bubble sort often requires more iterations than necessary to complete sort a list. Bubble sort is not often used in the real-world applications because of its inefficiency. However, sometimes best-case scenario occurs when the input array is already sorted, and no element swap is needed. (*Dr. Carr, 2024*)

Sample Bubble Sort Java Code

Code adapted from: (*YouTube Video – Bubble Sort Algorithm, 2024*)

```
import java.util.Arrays;

public class BubbleSort {

  public static void main(String[] args) {

    int array[] = {4, 3, 8, 8, 0, 3};

    System.out.println ("Array before sorting:"+ Arrays.toString(a));

    int n = array. Length;

    for (int i = 0; i < n-1; i++) {  // Outer loop to access each array element

      for (int j = 0;  j < n-1;  j++) {  //Inner loop to compare array elements

        if (array [j]  > array [j+1]) {

          int temp = array [j];        // swapping occurs if elements are not in the
                                                intended order

          array [j] = array [j+1];

          array [j+1] = temp;

        }

      }

    }

    System.out.println("Array After sorting:"+ Arrays.toString(a));
    }
}
```

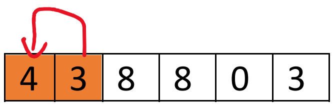
Diagram of Bubble Sort Working:

Index Number	0	1	2	3	4	5
Input Data	4	3	8	8	0	3

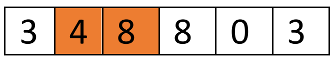
FISRT PASS: Largest element in the input bubbles to the right.

Steps:

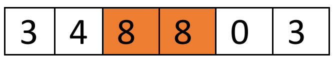
4 is greater than 3 – swap



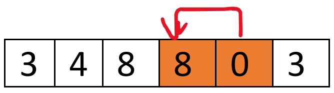
4 is less than 8 – no swap



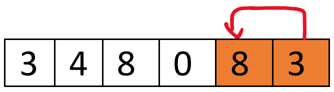
8 is equal to 8 – no swap



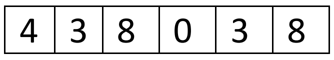
8 is greater than 0 – swap



8 is greater than 3 – swap



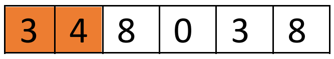
After First Pass



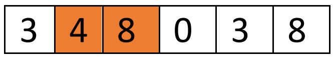
SECOND PASS: Largest element in the input bubbles to the right.

Steps:

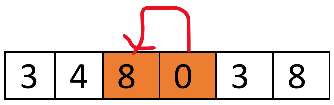
3 is less than 4 – no swap



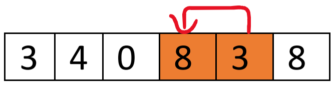
4 is less than 8 – no swap



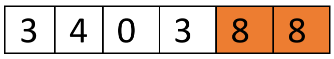
8 is greater than 0 –swap



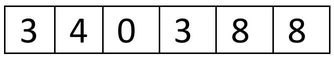
8 is greater than 3 – swap



8 is equal to 8 – no swap



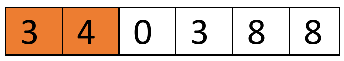
After Second Pass



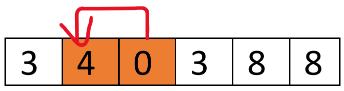
THIRD PASS: Largest element in the input bubbles to the right.

Steps:

3 is less than 4 – no swap



4 is greater than 0 – swap



4 is greater than 3 - swap

3	0	4	3	8	8

4 is less than 8 – no swap

3	0	3	4	8	8

8 is equal to 8 – no swap

3	0	3	4	8	8

After Third Pass

3	0	3	4	8	8

<u>FOURTH PASS:</u> Largest element in the input bubbles to the right.

Steps:

3 is greater than 0 - swap

3	0	3	4	8	8

3 is equal to 3 – no swap

0	3	3	4	8	8

3 is less than 4 – no swap

0	3	3	4	8	8

4 is less than 8 – no swap

3	0	3	4	8	8

8 is equal to 8 – no swap

3	0	3	4	8	8

After Fourth Pass

3	0	3	4	8	8

<u>Fifth PASS:</u> Largest element in the input bubbles to the right.

Steps: Largest element in the input bubbles up to the right.

3 is greater than 0 – swap

3	0	3	4	8	8

3 is equal to 3 – no swap

0	3	3	4	8	8

3 is less than 4 – no swap

0	3	3	4	8	8

4 is less than 8 – no swap

0	3	3	4	8	8

8 is equal to 8 – no swap

0	3	3	4	8	8

After Fifth Pass

0	3	3	4	8	8

Input data set is sorted: - Complete.

2. Insertion Sort:

Insertion sort is also a simple comparison-based sorting algorithm. It is a sorting algorithm that places an unsorted element at its suitable place in each iteration. Insertion sort algorithm works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side. Otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place. The same approach is applied to insertion sort. It is easy to implement. (*javapoint.com, 2024*)

It is simple to use and efficient for small data sets, not for larger data sets. Insertion sort is based on the idea of repeatedly moving higher ranked input integers to the end of the input list.

Its Space and Time Complexity:

Its time complexity is ‘n’ in best case, and ‘n^2’ in worst and average cases.

Specifications Big O Notation

Best Case	Average Case	Worst Case	Space Complexity	Stable
O(n)	$O(n^2)$	$O(n^2)$	1	Yes

Above Table showing Insertion sort specifications (Dr. Carr & geeksforgeeks,2024)

Best case: It occurs when there is no sorting required. The array is already sorted. The best-case time complexity of insertion sort is O(n) (*javapoint.com, 2024)*

Average case: It occurs when the array elements are in jumbled order, that is not properly ascending and descending. The average case time of insertion sort is $O(n^2)$. (*javapoint.com, 2024)*

Worst Case: It occurs when the array elements are required to be sorted in reverse order. That means suppose we must sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is $O(n^2)$. (*javapoint.com, 2024*)

Big O Analysis of Insertion Sort:

Insertion sort is a simple and efficient sorting algorithm for small input sizes. Insertion sort algorithm performs two operations: It scans through the list, comparing each pair of elements, and it swaps elements if they are out of order.

(*brilliant.org, 2024*) It is very useful for small datasets as best cases in n time. It also has an average and worst-case time complexity due to the nested loops for inserting elements. But it also has a best time complexity for an already sorted array. (*medium.com, 2024*)

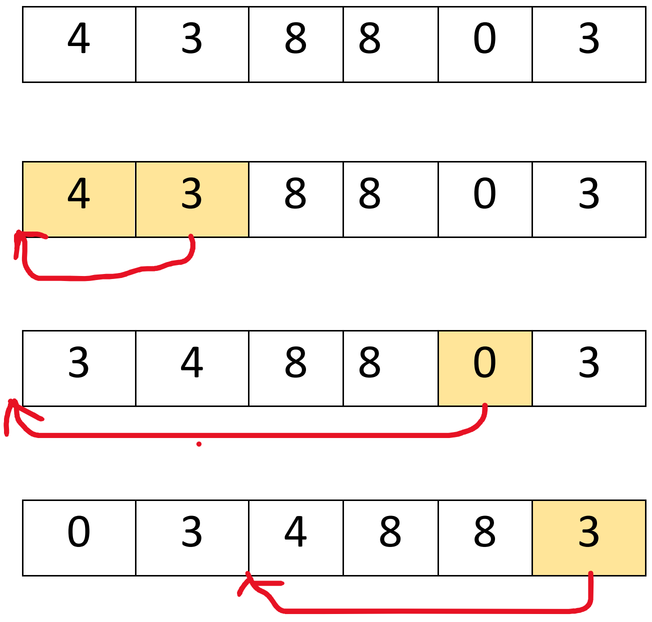
Diagrams of Insertion Sort Working:

Steps of Insertion Sort working:

1. Items in the input set are viewed as to whether they are sorted or not sorted.
2. Using insertion sort data is shuffled to the left if smaller than its comparator to the right. This rearranges the items into the correct order.

Index Number	0	1	2	3	4	5
Input Data	4	3	8	8	0	3

ARRAY



RESULT	0	3	3	4	8	8

Input data set is sorted: - Complete.

Sample Insertion Sort Java Code:

Code adapted from geeksforgeeks, 2024

```
import java.util.Arrays;
public class InsertionSort {
   void insertionSort(int arr[]) {
      int n = arr.length;
      for (int i = 1; i < n; ++i) {   //Outer loop over the array starting at index 1
         int key = arr[i];
         int j = i - 1;
          // Comparing key with each element on the left-hand side of it
          // until an element smaller than it is found
         while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1; // Move the index to be compared to the left by one position
         }
         arr[j + 1] = key;  // Complete the process again, this time one position to the right
      }
   }
```

3. Selection Sort:

Selection sort is a sorting algorithm that belongs to the group of simple and comparison-based algorithms. It is similar to Insertion sort algorithm. Selection sort is an In-place and unstable algorithm. It is easy and simple to implement.

We need to select the smallest element in the unsorted section of the array and move it to the beginning of the unsorted section of the array.

Selection sort is an effective and efficient sort algorithm. It gives better performance as compared to bubble sort, but still not practical for real world tasks with significant input size. The best, average and worst time complexity of selection sort is n^2. (*Dr. Carr, 2024*).

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. So, the process continues until this entire array is sorted. (*javapoint.com, 2024*)

Its Time and Space Complexity: The best, average and worst cases time complexity of selection sort is n^2. Its space complexity is 1.

Specifications Big O Notation

Best Case	Average Case	Worst Case	Space Complexity	Stable
$O(n^2)$	$O(n^2)$	$O(n^2)$	1	No

Above Table showing Insertion sort specifications (Dr. Carr & geeksforgeeks,2024)

Best case: Best case complexity occurs when there is no need for sorting, i.e. the arrays has already been sorted. The time complexity of selection sort algorithm in best case scenario is $O(n^2)$. (*simplilearn.com, 2024*)

Average case: Average case complexity occurs when the array elements are arranged in a jumbled order that is neither ascending nor descending correctly. The time complexity of selection sort algorithm in average case scenario is $O(n^2)$. (*simplilearn.com, 2024*)

Worst case: Worst case occurs when array elements must be sorted in reverse order. Assume we need to sort the array elements in ascending order, but they are in descending order. The time complexity of selection sort algorithm in worst case is $O(n^2)$. (*simplilearn.com, 2024*)

Big O Analysis of Selection Sort:

Selection sort algorithm is efficient for small data sets. It is simple and easy to implement. It is appropriate for data sets that are already substantially sorted. It adds one element in each iteration. (*javapoint.com, 2024*)

The best, average and worst-case time complexity of selection sort is $O(n^2)$. The selection sort algorithm is made up of two nested loops. It has an $O(n^2)$ time complexity due to the two nested loops. It performs all computations in the original array and does not use any other arrays. As a result, its space complexity is O(1). (*simplilearn.com, 2024*) Selection sort algorithm consistently outperforms bubble sort and gnome sort. When memory writing is a costly operation, this can be useful.

Diagrams of Selection Sort Working:

Index Number	0	1	2	3	4	5
Input Data	4	3	8	8	0	3

INPUT ARRAY

4	3	8	8	0	3

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 4 is stored presently, I search the whole list and find that 0 is the lowest value.

4	3	8	8	0	3

So, I replace 4 with 0. After one iteration 0, which happens to be the minimum value in the list, appears in the first position of the sorted list.

0	3	8	8	4	3

For the second position, where 3 is residing, I start scanning the rest of the list in a linear manner.

0	3	8	8	4	3

In the second position, where 3 is residing, I find that 3 is lowest value. So, 3 will be remain in the second position.

0	3	8	8	4	3

For the third position, where 8 is residing, I start scanning the rest of the list in a linear manner.

0	3	8	8	4	3

I find that 3 is the third lowest value in the list and it should appear at the third place. I swap these values.

0	3	3	8	4	8

For the fourth position, where 8 is residing, I start scanning the rest of the list in a linear manner.

0	3	3	8	4	8

I find that 4 is the fourth lowest value in the list and it should appear at the fourth place. I swap these values.

0	3	3	4	8	8

For the fifth position, where 8 is residing, I start scanning the rest of the list in a linear manner.

0	3	3	4	8	8

In the fifth position, where 8 is residing, I find that 8 is equal value with value in the sixth position. So, 8 will be remain in the fifth position.

0	3	3	4	8	8

The Array is sorted.

Sample Selection Sort Java Code:

Code adapted from *geeksforgeeks*, 2024.

```
import java.io.*;
public class SelectionSort {
   void selectionSort(int array[]) {
      int n = array.length;
      for (int i = 0; i < n-1; i++) {    // Outer loop
         int min_idx = i;
         for (int j = i+1; j < n; j++)    // Inner loop of unsorted array
            if (array[j] < array[min_idx]) // Locate the index of the min element in the
                                              unsorted subarray
            min_idx =  j;
         int temp = array[min_idx]; // Swapping elements
         array[min_idx] = array[i];
         array [i] = temp;
      }
   }
```

4. Merge Sort:

Merge sort is more complex comparison-based sorting algorithm that was developed by John Von Neumann in 1945. Merge sort uses a recursive "Divide and Conquer" approach to sort the elements. (Dr. Carr, 2024) Merge sort is similar to the quick sort algorithm. It is one of the most popular and efficient sorting algorithms. It divides the given list in to two halves, calls itself for two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.(*javapoint.com, 2024*)Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array. It is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. Time complexity of Merge sort algorithm in best, average and worst cases is very similar. It is O(n log n). (*Geeksforgeeks, 2024*)

Its Time and Space Complexity: The best case, average case and worst-case time complexity of Merge sort is very similar. It is O(n log n). Its space complexity is O(n).

Specifications Big O Notation

Best Case	Average Case	Worst Case	Space Complexity	Stable
O(n log n)	O(n log n)	O(n log n)	O(n)	Yes

Above Table showing Insertion sort specifications (Dr. Carr & geeksforgeeks,2024)

Best case: It occurs when the array is already sorted or nearly sorted. The best-case time complexity of Merge sort is O(n log n).(*geeksforgeeks,2024)*

Average case: It occurs when the array is randomly ordered. The average case time complexity of Merge sort is O(n log n). *(geeksforgeeks,2024)*

Worst case: It occurs when the array is sorted in reverse order. The worst-case time complexity of Merge sort is O(n log n).(*geeksforgeeks,2024*)

Big O Analysis of Merge Sort:

Merge sort versions are very good for sorting data with slow access times such as data not held in internal memory (RAM) or stored in linked lists. Merge sorts best, average, and worst-case time complexity is very similar. And merge sort gives a very good performance. Merge sort is a stable sort. (*Dr. Carr, 2024*) Merge sort has a worst-case time complexity of O(N log N), which means it performs well even on large datasets. Its space complexity is O(n). It requires additional space to store the merged sub-arrays during the sorting process. (*Geeksforgeeks, 2024*)

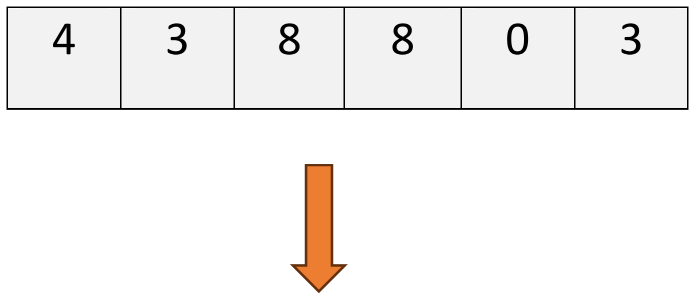
Diagrams of Merge Sort Working:

Steps of Merge Sort Workings:

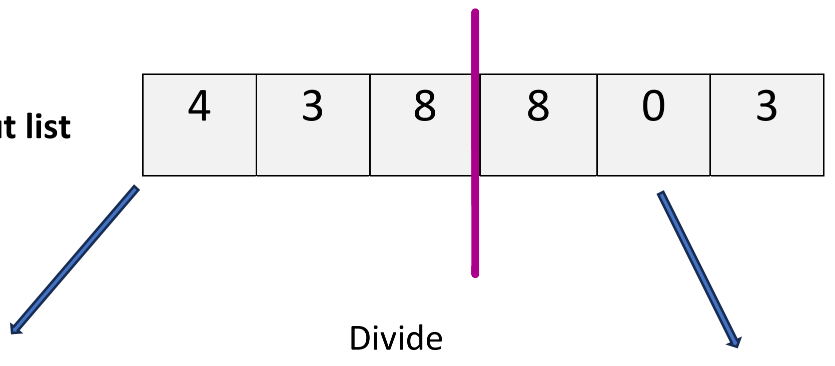
1. The input list is split into two separate smaller lists.
2. The two separate lists are further simplified into smaller more sorted lists.
3. Elements are sorted and rearranged in order. The smaller lists begin to merge back to bigger lists.
4. The final two lists are merged to form one sorted list.

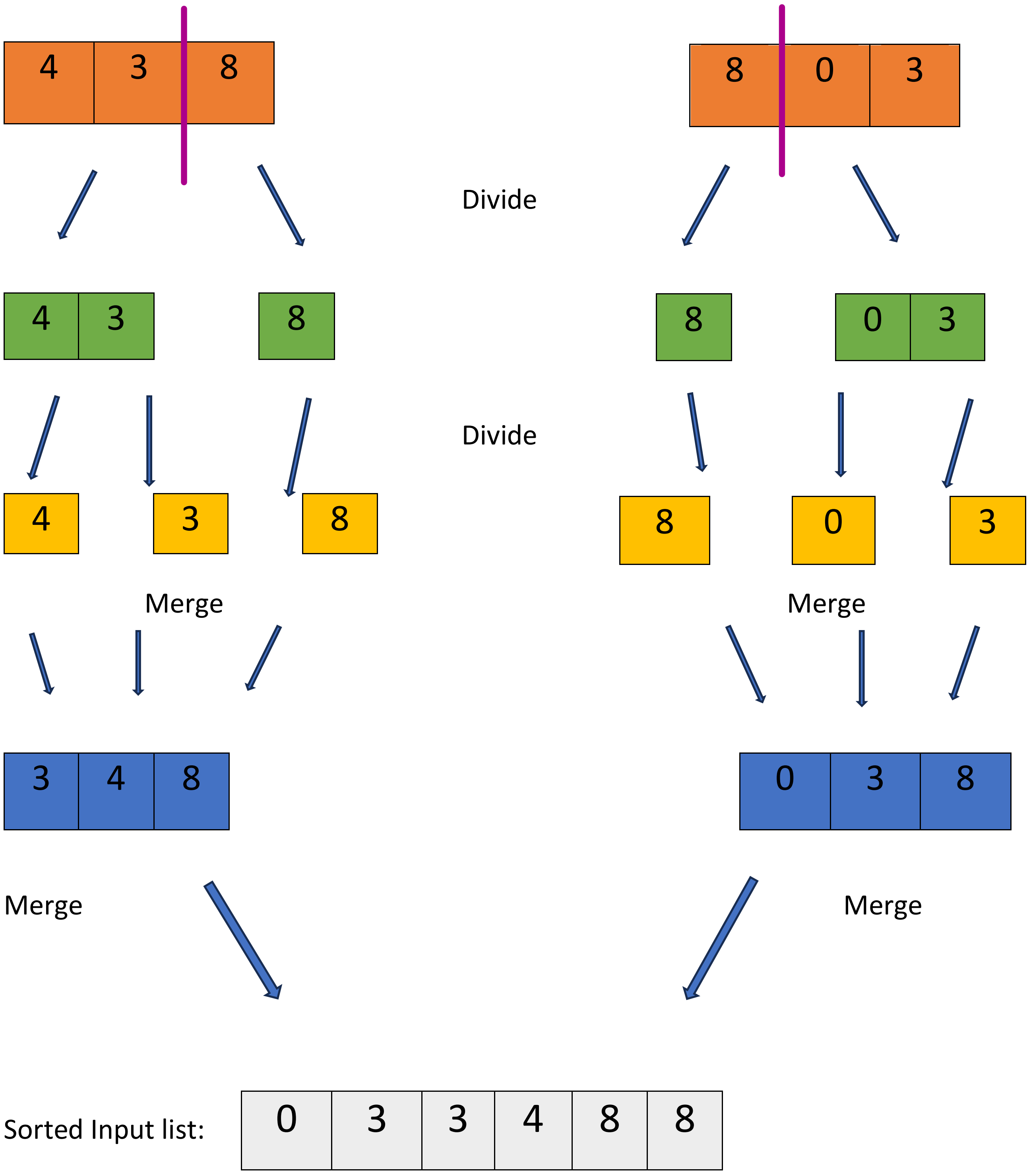
Index Number	0	1	2	3	4	5
Input Data	4	3	8	8	0	3

Input List of Integers:



Unsorted Input list




4
3
8
8
0
3
Divide
4
3
8
8
0
3
Divide
4
3
8
8
0
3
Merge
Merge
3
4
8
0
3
8
Merge
Merge
Sorted Input list:
0
3
3
4
8
8

Sample Merge Sort Java Code: (adapted from geeksforgeeks, 2024)

```
import java.io. *;
class MergeSort {

  // Function to merge the subarrays of array[]
  void merge(int arr[], int l, int m, int r){
    // Find sizes of two sub-arrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;

    // Temporary arrays
    int L[] = new int[n1];
    int R[] = new int[n2];

    // Copy data to temporary 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];




    int i = 0;     // Initial index of first sub-array
    int  j = 0;     // Initial index of second sub-array
    int k  = l;      // Initial index of merged sub-array

    while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
          arr[k] = L[i];
          i++;
      }
      else {
          arr[k] = R[j];
          j++;
      }
      k++;
    }

    // Copy the remaining elements of Left [] if there are any
    while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
    }

    // Copy the remaining elements of Right [] if there are any
    while (j < n2) {
      arr[k] = R[j];
      j++;
      k++;
    }
  }

  // Main function that sorts arr[l..r] using
  // merge()
  void sort(int arr[],  int l,  int r)
  {
    if (l < r) {

      // Find the middle point
      int m = l + (r - l) / 2;

      // Sort first and second halves
      sort(arr, l, m);
      sort(arr, m + 1, r);

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

5. Counting Sort:

Counting sort is a non-comparison-based sorting algorithm. It is invented by Harold H. Seward in 1954. (*Dr. Carr, 2024*) Counting sort is an integer sorting algorithm used in computer science to collect objects according to keys that are small positive integers. (*simplilearn.com. 2024*) This sorting technique doesn't perform sorting by comparing elements. It performs sorting by counting objects having distinct key values like hashing. After that, it performs some arithmetic operations to calculate each object's index position in the output sequence. Counting sort is not used as a general-purpose sorting algorithm. Counting sort is effective when range is not greater than number of objects to be sorted. It can be used to sort the negative input values. (javapoint.com, 2024) The best case, average case and worst-case time complexity of Counting sort algorithm is O(n + k). Counting sort is stable.

Its Time and Space Complexity: The best case, average case and worst-case time complexity of Counting sort algorithm is similar. It is O(n + k). Counting sort of space complexity is O(max)

Specifications Big O Notation

Best Case	Average Case	Worst Case	Space Complexity	Stable
O(n + k)	O(n + k)	O(n + k)	O(n + k)	Yes

Above Table showing Insertion sort specifications (*Dr. Carr, 2024*)

Best case: The best-case time complexity of counting sort is O(n + k).

Average case: The average case time complexity of counting sort is O(n + k).

Worst case: The worst-case time complexity of counting sort is O(n + k).

Big O Analysis of Merge Sort:

Counting sort is a linear sorting algorithm with asymptotic complexity O(n + k). The counting sort method is fast and reliable sorting algorithm. Counting sort unlike bubble sort and merge sort is not a comparison-based algorithm. It avoids comparisons and takes advantage of the array's O(1) time insertions and deletions. (*simplilearn.com, 2024*). The time complexity comes from counting the elements (O(n)time) and iterating over the range of input data (O(k)time). (*studysmarter,2024*) Its main strength in time efficiency is unfortunately shadowed by some of its other weaknesses.

Diagrams of Merge Sort Working:

Index Number	0	1	2	3	4	5
Input Data	4	3	8	8	0	3

1. Find out the maximum element (let it be max.) from the given array.

Input Array

0	1	2	3	4	5
4	3	8	8	0	3

Max
8

2. Initialize a count Array[] of length max+1 with all elements as 0. This array will be used for sorting the occurrences of the elements of the input array.

Count Array

0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0

3. Store the count of each unique element of the input array at their respective indices.

Count Array

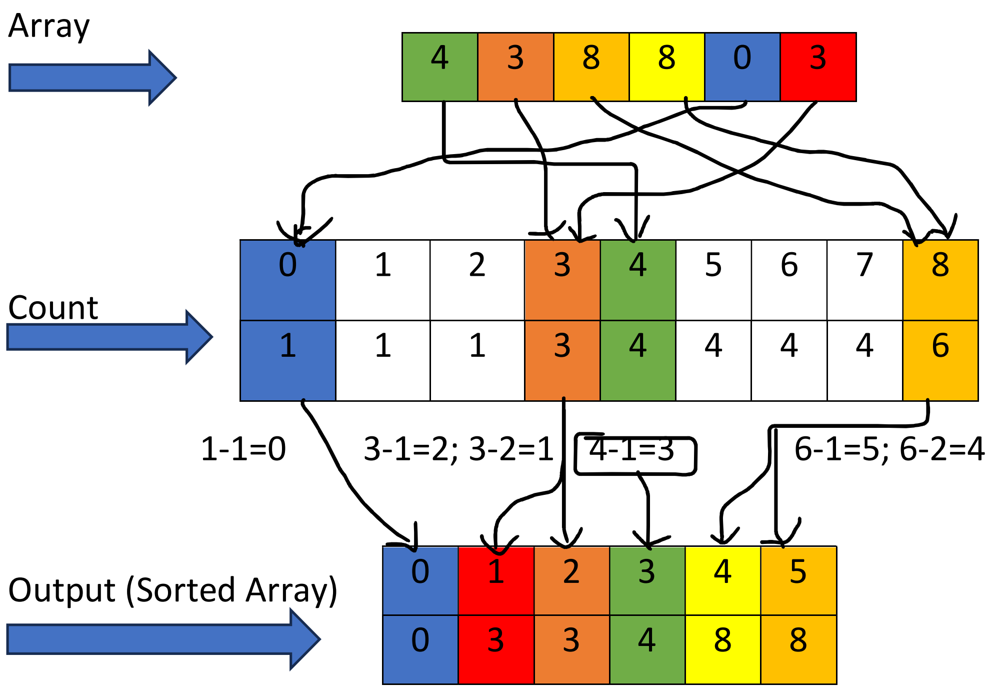
0	1	2	3	4	5	6	7	8
1	0	0	2	1	0	0	0	2

4. Store the cumulative sum of the elements of the countArray[]. It helps in placing the elements into the correct index of the sorted array.

Cumulative count

0	1	2	3	4	5	6	7	8
1	1	1	3	4	4	4	4	6

5. Find the index of each element of the original array in the count array. This gives the cumulative count.



The array is completely sorted.

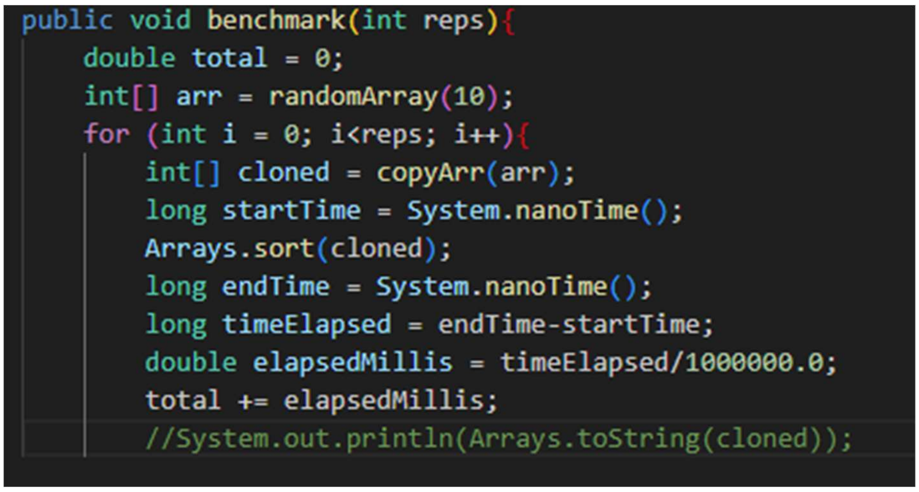
Sample Counting Sort Java Code: (adapted from geeksforgeeks, 2024)

```
import java.util.Arrays;


public class CountingSort {

        public static int[] countingSort(int[] inputArray) {

                int N = inputArray.length;    // Array size

                int M = 0;

                for (int i = 0; i < N; i++) {          // Finding the maximum element of the array

                        M = Math.max(M, inputArray[i]);

                }
                int[] countArray = new int[M + 1];  // Initializing the count array
                for (int i = 0; i < N; i++) {
                        countArray[inputArray[i]]++;
                }
                for (int i = 1; i <= M; i++) {      //Changing the count array for positions
                        countArray[i] += countArray[i - 1];
                }
                int[] outputArray = new int[N];
                for (int i = N - 1; i >= 0; i--) {
                        outputArray[countArray[inputArray[i]] - 1] = inputArray[i];
                        countArray[inputArray[i]]--;
                }
                return outputArray;  //Sorted output array returned
        }
```

Implementation and Benchmarking – Results:

All code for this project was written using the Java NetBeans package and Visual Studio Code. I have tried to implement and interpret as much as possible using the code taken from YouTube video and Geeks for Geeks (geeksforgeeks). I struggled a lot to understand these algorithms before implementing this code. After many attempts and understanding, finally managed to implement this benchmarking code for this project. It took a lot of effort to get the benchmarking results of different algorithms in tabular form. The source code provided by Dr. Carr was used to benchmark these 5 sorting algorithms. To benchmark the 5 different algorithms, I sued the arrays of randomly generated integers with different input sizes (n). Those input sizes (n) are: 100, 250, 500, 1000, 2000, 2500, 3000, 3500, 5000, 10000, 10001, to test the effect of the input size(n) on the running time of each algorithm. Each algorithm was run 10 times, its average running time was taken in milliseconds and rounded to 3 decimal places. So, this process is repeated for each different ‘n’ value.

Benchmarking Code



```
public void benchmark(int reps){
    double total = 0;
    int[] arr = randomArray(10);
    for (int i = 0; i<reps; i++){
        int[] cloned = copyArr(arr);
        long startTime = System.nanoTime();
        Arrays.sort(cloned);
        long endTime = System.nanoTime();
        long timeElapsed = endTime-startTime;
        double elapsedMillis = timeElapsed/1000000.0;
        total += elapsedMillis;
        //System.out.println(Arrays.toString(cloned));
```

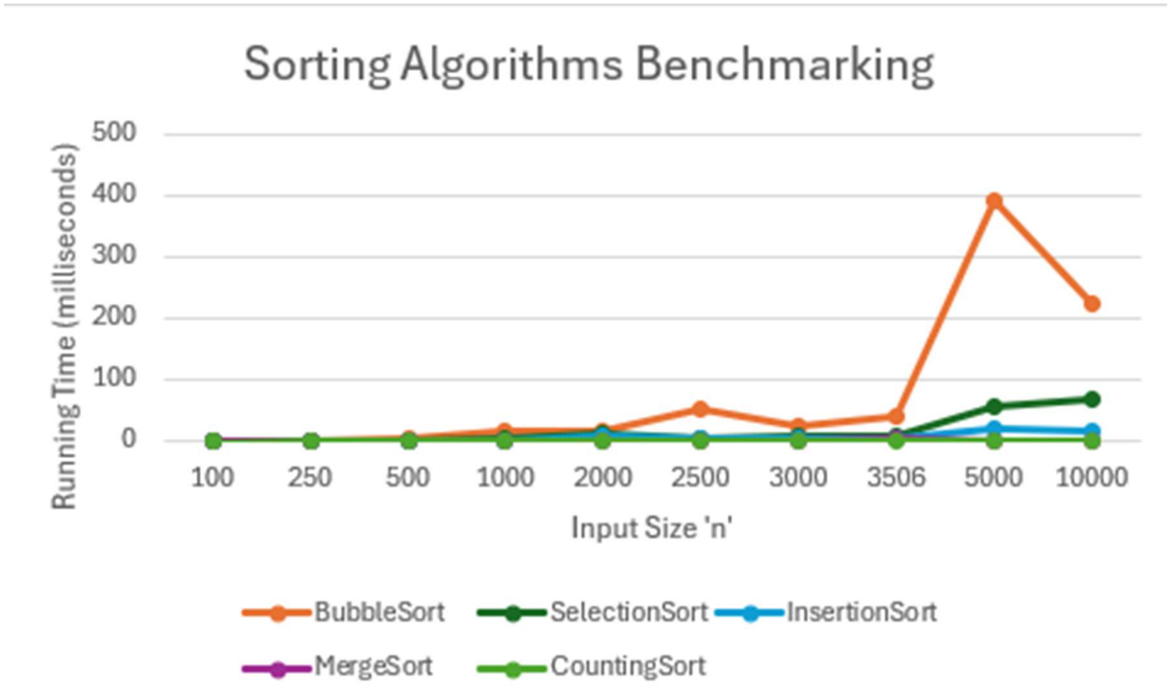
Image taken from (*Dr. Carr, 2024*)

Benchmarking Results:

The benchmarking results are as follows. The value indicated in the table is the average time measured in milliseconds of 10 randomly generated arrays of size ‘n’.

Size	100	250	500	1000	2000	2500	3000	3500	5000	10000
Bubble	0.214	0.914	2.832	16.731	13.659	53.136	23.795	38.429	389.423	221.849
Selection	0.041	0.234	0.743	3.381	11.646	5.188	7.582	8.342	54.91	68.971
Insertion	0.017	0.065	0.296	1.087	7.394	1.919	2.060	4.334	17.704	13.856
Merge	0.025	0.064	0.296	1.463	1.403	0.428	0.389	2.030	0.968	0.987
Counting	0.002	0.003	0.005	0.009	0.012	0.015	0.013	0.029	0.033	0.032

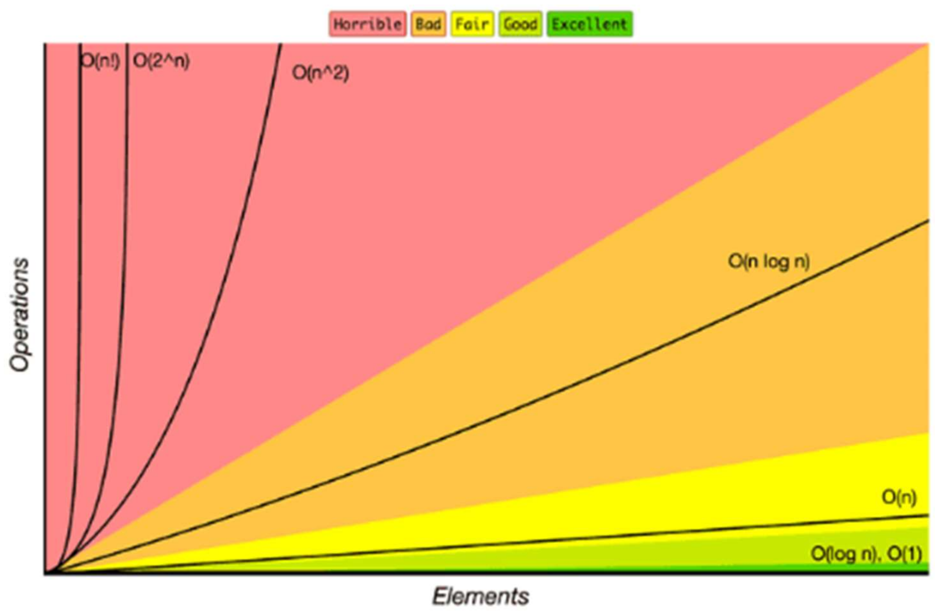
Graph representation of the Time Complexity of each sorting Algorithm Tested:



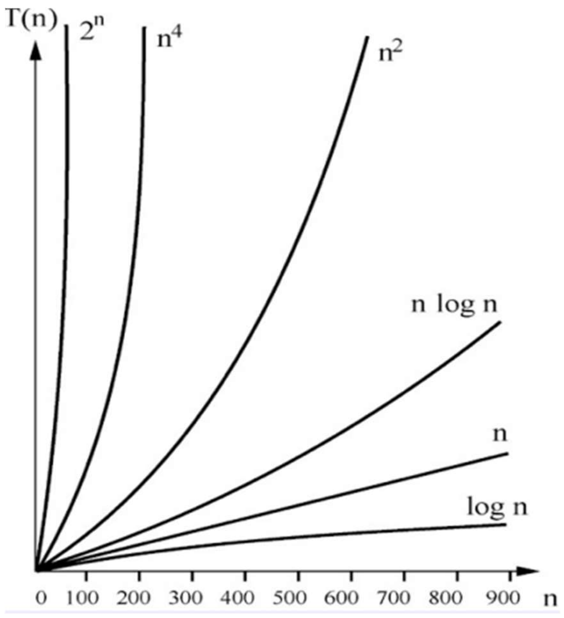
By analysing the time complexity of each of 5 sorting algorithms, one of the sorting algorithms was able to identify scalability with respect to increased input size. It can be clearly seen that each of the algorithms follows a unique pattern according to their ‘Big O time complexity’.

Algorithm	Best Case	Average Case	Worst Case	Space Complexity	Stable
Bubble Sort	O(n)	O(n^2)	O(n^2)	1	Yes
Selection Sort	O(n^2)	O(n^2)	O(n^2)	1	No
Insertion Sort	O(n)	O(n^2)	O(n^2)	1	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(n + k)	Yes

.



Running time T (n) is proportional to:	Complexity	
$T(n) \propto \log n$	logarithmic	Decreasing in efficiency
$T(n) \propto n$	linear	
$T(n) \propto n \log n$	linearithmic	
$T(n) \propto n^2$	quadratic	
$T(n) \propto n^3$	cubic	
$T(n) \propto n^k$	polynomial	
$T(n) \propto 2^n$	exponential	
$T(n) \propto k^n$; $k > 1$	exponential	



Above image showing Running times in relation to Complexity (*cs.auckland.ac.nz, 2024*)

Summary:

Finally, to conclude this project focused on evaluating the time complexity of the five sorting algorithms. These five sorting algorithms are Bubble sort, Insertion sort, Select sort, Merge sort and Counting sort. A comprehensive algorithm application was developed to benchmark each to estimate their time complexity. After doing research for this project and analysing these algorithms in different ways, I found their performance to be very interesting.

References:

Works Cited

"Big O Notation | Brilliant Math & Science Wiki." *Brilliant.org*, 2019, brilliant.org/wiki/big-o-notation/. Accessed May 2024.

"Bubble Sort: Algorithm, Example, Time Complexity & Advantages." *StudySmarter UK*, www.studysmarter.co.uk/explanations/computer-science/algorithms-in-computer-science/bubble-sort/#:~:text=The%20time%20complexity%20of%20Bubble%20Sort%20is%20always%20O(n. Accessed May 2024.

BubbleSort Code. "Frequently Asked Java Program 22: Sort Elements in Array | Bubble Sort." *Www.youtube.com*, www.youtube.com/watch?v=cJ2eMUiCFy4&t=245s. Accessed 13 May 2024.

Carr, Dr. Dominic. *Class Notes*. Accessed May 2024.

Carr, Dr. Dominic . *Bubble Sort, Class Notes*.

"Clojure - Comparators Guide." *Clojure.org*, clojure.org/guides/comparators#:~:text=A%20comparator%20is%20a%20function. Accessed May 2024.

"Comparator (Java Platform SE 8)." *Docs.oracle.com*, docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#:~:text=Comparators%20can%20also%20be%20used. Accessed 13 May 2024.

"Comparison among Bubble Sort, Selection Sort and Insertion Sort - GeeksforGeeks." *GeeksforGeeks*, Apr. 2019, www.geeksforgeeks.org/comparison-among-bubble-sort-selection-sort-and-insertion-sort/. Accessed May 2024.

"Counting Sort - Javatpoint." *Www.javatpoint.com*, www.javatpoint.com/counting-sort. Accessed May 2024.

"Counting Sort Algorithm: Overview, Time Complexity & More | Simplilearn." *Simplilearn.com*, www.simplilearn.com/tutorials/data-structure-tutorial/counting-sort-algorithm#:~:text=Counting%20sort%20is%20an%20integer. Accessed 13 May 2024.

Dictionary, Ocford. *Oxford Dictionary*.

"Educative Answers - Trusted Answers to Developer Questions." *Educative*, www.educative.io/answers/stable-and-unstable-sorting-algorithms. Accessed May 2024.

GeeksforGeeks. "Counting Sort." *GeeksforGeeks*, 18 Mar. 2013, www.geeksforgeeks.org/counting-sort/. Accessed May 2024.

---. "GeeksforGeeks | a Computer Science Portal for Geeks." *GeeksforGeeks*, www.geeksforgeeks.org/. Accessed May 2024.

---. "Insertion Sort Code - GeeksforGeeks." *GeeksforGeeks*, 7 Mar. 2013, www.geeksforgeeks.org/insertion-sort/. Accessed May 2024.

---. "Merge Sort - GeeksforGeeks." *GeeksforGeeks*, 31 Oct. 2018, www.geeksforgeeks.org/merge-sort/. Accessed May 2024.

geeksforgeeks. “Selection Sort Code - GeeksforGeeks.” *GeeksforGeeks*, 31 Jan. 2014, www.geeksforgeeks.org/selection-sort/. Accessed May 2024.

https://www.cs.auckland.ac.nz/compsci220s1t/lectures/lecturenotes/GG-lectures/220slides-lecture03.pdf. Accessed May 2024.

“Insertion Sort - Javatpoint.” *Www.javatpoint.com*, www.javatpoint.com/insertion-sort. Accessed May 2024.

InsertionSort. “Insertion Sort in Java - Javatpoint.” *Www.javatpoint.com*, www.javatpoint.com/insertion-sort-in-java. Accessed May 2024.

Kids, Nesk. “Why Is Classifying and Sorting Important for Young Children?” *Nesk Kids*, 2 Sept. 2019, www.neskkids.com.au/blogs/news/classifying-and-sorting. Accessed May 2024.

“Merge Sort - Javatpoint.” *Www.javatpoint.com*, www.javatpoint.com/merge-sort. Accessed May 2024.

“SORTING TOWERS 28X21X9 CM.” *Camelino: Εξοπλισμός Σχολείων, Παιδικών Σταθμών, ΚΔΑΠ*, camelino.gr/en/p/sorting-towers-28x21x9-cm. Accessed 13 May 2024.

StackOverFlow. “Stack Overflow - Where Developers Learn, Share, & Build Careers.” *Stack Overflow*, 2019, stackoverflow.com. Accessed May 2024.

Thakrani, Suhailt. “What Are Stable Sorting Algorithms and In-Place Sorting Algorithms?” *Medium*, 21 July 2023, medium.com/@suhailthakrani12/what-are-stable-sorting-algorithms-and-in-place-sorting-algorithms-672820a8e36c#:~:text=An%20in%2Dplace%20sorting%20algorithm%20is%20a%20sorting%20algorithm%20that. Accessed May 2024.

“What Is Selection Sort Algorithm in Data Structures? | Simplilearn.” *Simplilearn.com*, www.simplilearn.com/tutorials/data-structure-tutorial/selection-sort-algorithm#:~:text=in%20bubble%20sort).-. Accessed May 2024.

Wikipedia Contributors. "Sorting." *Wikipedia*, Wikimedia Foundation, 16 Jan. 2019, en.wikipedia.org/wiki/Sorting. Accessed May 2024.

---. "Sorting Algorithm." *Wikipedia*, Wikimedia Foundation, 8 Dec. 2018, en.wikipedia.org/wiki/Sorting_algorithm. Accessed May 2024.