<u>Computational Thinking and Algorithms Project 2024</u>

Dylan Boyle

Student ID: G00438786

ATU Galway-Mayo – 2024

Higher Diploma in Software Development – CTA

Table of Contents

Introduction

As part of this project I will be writing about the five sorting algorithms, the five sorting algorithms are as follows:

- Bubble Sort
- Selection Sort
- Insertion Sort
- A non-comparison sort: Counting Sort.
- An efficient comparison-based sort: Merge Sort.

Concept of Sort:

Sorting are operation of arranging data in some given sequence such as increasing order (A-Z) or decreasing order (Z-A). The data items with smaller or minimum value to the left end and items with larger or maximum value to the right end. (https://en.wikipedia.org/wiki/Sorting) An example would be if given a stack of books from the library to organise them in alphabetical order. The computer does this all the time by organising and managing large datasets by improving search performance and displaying them in an organised manner, making it easier for users to read and interact with.

The Purpose of Sorting Algorithms:

1. Data Organisation: Sorting is important for organising data for users to help them find what they are looking for, usually in alphabetical order. For example, listing items for phone numbers in alphabetical or numerical order for users to find with ease.
2. Data Analysis: Is used a lot in social media such as Instagram, Facebook, YouTube, etc. When a user watches or like videos or photos, the sorting algorithms identify a pattern and trend, and present it to the users to see or watch the content.
3. Efficient Searching: Sorted data allows for effective searching algorithms like binary search, which lower the time complexity. (https://academicworks.cuny.edu/cgi/viewcontent.cgi?article=1911&context=ny_pubs) (https://www.programiz.com/dsa/sorting-algorithm)

Three Main Type of Sorting Algorithms:

1. Comparison-based Sorting: The comparison-based sort consists of algorithms, such as Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, etc. These algorithms compare elements in pairs and rearrange them in order of similar characteristics.

The Time-Complexity in comparison-based algorithms have an average time complexity of O(n log n), where n is the number of elements to be sorted, which makes this very efficient for large dataset.
It also has greater Stability for some of the comparison-based algorithms providing relative order and preservation of equal elements after sorting. Bubble Sort and Merge Sort are great stable algorithms. Certain comparison-based algorithms can sort the elements in-place, which means they do not require additional memory for sorting, which Bubble Sort, Insertion Sort, and Selection Sort are known for.
(https://www.cs.cornell.edu/courses/JavaAndDS/files/comparisonSorting.pdf)

2. Non-Comparison-based Sorting: The non-comparison-based sorting consists of algorithms, such as Counting Sort, Radix Sort, and Bucket Sort. These algorithms do not rely on pairs of comparing elements to sort them, instead they use different methods to rearrange items effectively.
The Time-Complexity in non-comparison-based algorithms have linear time complexity, O(n), making them effective in certain scenarios where comparison-based algorithms may be less ideal.
The non-comparison-based algorithms can be space-efficient when the range of values is limited, as they often require additional space relative to the range size.
(https://academicworks.cuny.edu/cgi/viewcontent.cgi?article=1911&context=ny_pubs)

3. Hybrid: The hybrid consists of combination of different sorting techniques, such as Timsort and Introsort. It aims to achieve a balance between effectively and flexibility by combining the strengths of different sorting methods.
By joining different sorting approaches, hybrid algorithms can enhance performance over basic sorting methods in some scenarios. Hybrid sorting algorithms may have greater implementation complexity compared to traditional sorting algorithms due to the need to manage the combination of multiple sorting methods.

When choosing a sorting algorithm, use the following categories to achieve the most:

Space Complexity: Space complexity is the amount of memory that an algorithm uses to solve a particular problem in relation to the size of its input. The efficiency of an algorithm is generally expressed using its time complexity.

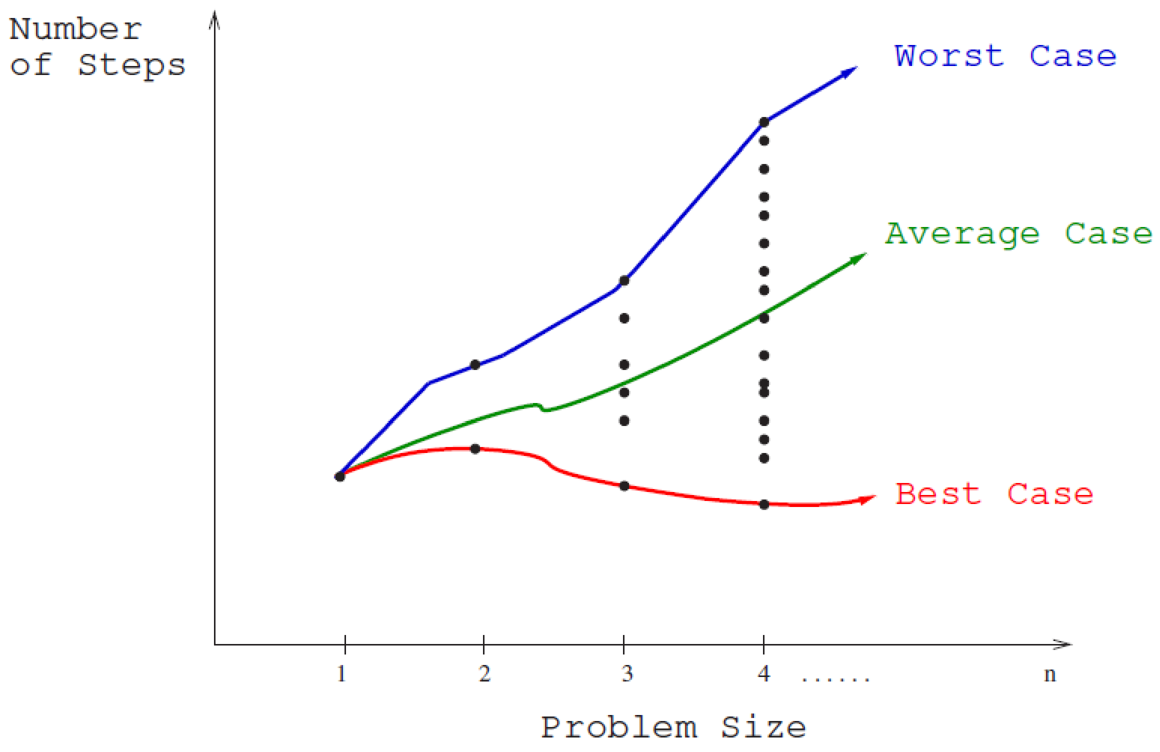
Time Complexity: The time complexity of a computer sorting algorithm is a measure of the amount of time it's takes to complete on the size of the list being sorted. There are three cases to consider when accessing the complexity of an algorithm, all of which can be expressed using Big O notation. '*In Big O notation, "O" represents the order of the* *function, and "f(n)" represents the function describing the algorithm's time complexity in terms of the input size "n."' (https://www.simplilearn.com/big-o-notation-in-data-structure-article#:~:text=O(1)%3A%20Constant%20time,linearly%20with%20the%20input%20size.)*

Best Case: The best-case scenario indicates the ideal performance of an algorithm, where it obtains the lowest time complexity for a given input size. It represents the most beneficial conditions for the algorithm, leading to the fastest execution time or the most effective. It occurs when the inputs data is already in a beneficial position.

Average Case: The average-case scenario relates to the expected performance of an overall possible inputs. It offers a more balanced view of the algorithm's efficiency. It considers the likelihood of different inputs. Analysing the average case performance of an algorithm can produce a more realistic view of its efficiency.

Worst Case: The worst-case scenario represents the least efficient performance of an algorithm, characterised by the highest possible time complexity for a given input sizes. It occurs when the input data is in the worst possible state or when the algorithm come across the most complex inputs. Acknowledging this scenario helps find the inputs that cause the most inefficient behaviour.

(https://en.wikipedia.org/wiki/Best,_worst_and_average_case)



(Image taken from this link: https://ai.plainenglish.io/algorithm-analysis-aa19be7abf8)

Performance: The performance in sort algorithm consists of speed of execution and efficiency, to neatly organise a given set of data. Efficient sorting algorithms need lower time complexity when sorting data even when the size of the input increases, it requires lower space complexity. Other performance considerations include whether the sorting algorithm is in-place, this means that the elements within the input data structure are rearranges and require no additional space for sorting.

(https://www.codeproject.com/Articles/26048/Fastest-In-Place-Stable-Sort)

Stability: The stability in sorting algorithms consists of data sorted in a way that retains the original order and value of elements ensuring 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. Stable sorting algorithms provide predictability and consistency in sorting outcomes, allowing users to rely on the sorted output.

(https://www.educative.io/answers/stable-and-unstable-sorting-algorithms)

Sorting Algorithms

This section provides a short overview of the five sorting algorithms, their time and space complexities and uses examples and diagrams to explain how each of the algorithm function. The five sorting algorithms in this section are as follow: Bubble Sort, Selection Sort, Insertion Sort, Counting Sort and Merge Sort.

Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through the list to be stored. It compares each pair of adjacent elements, and simply swap them if they are in the wrong order. It continues to move down the list until no swaps are needed, illustrating that the list is sorted. Bubble Sort has a time complexity of O(n2) in average and worst-case scenario, making it less ideal for large dataset. O(n) is Bubble Sort best-case scenario. (https://en.wikipedia.org/wiki/Bubble_sort)

Diagram of how Bubble Sort work (using the last six digits of my Student ID (438786) as Input Data Sample):

Index Number:	0	1	2	3	4	5
Input Data:	4	3	8	7	8	6

Index 0:

Index Number:	0	1	2	3	4	5
Input Data:	4	3	8	7	8	6
	3	4	8	7	8	6
	3	4	7	8	8	6
End of Index 0:	3	4	7	8	6	8

For each index, we will move from left to right swapping adjacent elements. Each index moves the largest element into its final position.

<u>Steps:</u>

- 4 is greater than 3 – swap. 348786
- 4 is less than 8 – no swap.
- 8 is greater than 7 – swap. 347886
- 8 is the same as 8 – no swap.
- 8 is greater than 6 – swap. 347868
- At the end of index 0 pass, the largest element is in its correct position.

Index 1:

Index Number:	0	1	2	3	4	5
Input Data:	3	4	7	8	6	8
End of Index 1:	3	4	7	6	8	8

<u>Steps:</u>

- 3 is less than 4 – no swap.
- 4 is less than 7 – no swap.
- 7 is less than 8 – no swap.
- 8 is greater than 6 – swap. 347688
- At the end of index 1 pass, the largest element is in its correct position.

Index 2:

Index Number:	0	1	2	3	4	5
Input Data:	3	4	7	6	8	8
End of Index 2:	3	4	6	7	8	8

<u>Steps:</u>

- 3 is less than 4 – no swap.
- 4 is less than 7 – no swap.
- 7 is greater than 6 – swap.
- At the end of index 2 pass, the largest element is in its correct position.

Result:	3	4	6	7	8	8

There is no need for further steps for next the index number, as the array is now sorted with the result: 346788.

Selection Sort

Selection sort is a straightforward comparison-based sorting algorithm that works by dividing the input list into two parts: the sorted sub list and the unsorted sub list. The algorithm repeatedly selects the minimum (or maximum, depending on the sorting order) element from the unsorted part of the array and swaps it to the beginning. Selection Sort has a time complexity of O(n2) in all cases (worst-case, average-case, and best-case scenarios) While it is not the most efficient sorting algorithm for large datasets due to it quadratic time complexity, Selection Sort is easy to implement and work well for small arrays.

<u>Diagram of how Selection Sort work (using the last six digits of my Student ID (438786) as Input Data Sample):</u>

Index 0:

Index Number:	0	1	2	3	4	5
Input Data:	4	3	8	7	8	6
End of Index 0:	4	3	6	7	8	8

For each index, we will move from left to right looking for the next largest value. Once that is found, it will be swapped into its final position.

Steps:

- Starting with biggest value – 4.
- Next thing is to compare the next biggest value, which is 8.
- Compare 8 to the next biggest value.
- There is no bigger value than 8, so 8 will switch to the last position. And the last position value 6 will be switch to the position 8 was in.
- At the end of index 0 pass, the largest element is in its correct position.

Index 1:

Index Number:	0	1	2	3	4	5
Input Data:	4	3	6	7	8	8
End of Index 0:	4	3	6	7	8	8

Steps:

- Starting with biggest value – 4.
- Next thing is to compare the next biggest value, which is 6.
- Compare 6 to the next biggest value, which is 7.
- Compare 7 to the next biggest value, which is 8.
- There is no bigger value than 8, so 8 will stay in the same position.
- At the end of index 1 pass, the largest element is in its correct position (No changes were made).

(Note: Nothing happens until Index number 4, where changes happen)

Index 4:

Index Number:	0	1	2	3	4	5
Input Data:	4	3	6	7	8	8
End of Index 4:	3	4	6	7	8	8

Steps:

- Starting with biggest value – 4.
- There is no bigger value than 4, so 4 will switch to the last position. And the last position value 3 will be switch to the position 4 was in.
- At the end of index 4 pass, the largest element is in its correct position.

There are no further steps for next index number, as the array is now sorted with the result: 346788.

Insertion Sort

Insertion Sort is a simple comparison-based sorting algorithm that builds the final sorted list one element at a time. It works by iterating through unsorted list and placing each element into its correct position in a new sorted list. It does this by comparing each element to the elements before it and swapping them to the right if they are larger, until the correct position is found. This cycle is repeated until all elements have been inserted into the new list, resulting in a sorted list.

Insertion Sort has a time complexity of O(n2) in the worst-case scenario and is effective for sorting small datasets or almost sorted lists. It is particularly useful when the input list is almost sorted, as it has linear time complexity of O(n) in the best-case scenario when elements are already in order. Insertion Sort is also stable, which means it look after the comparative order of elements in the sorted output.

Diagram of how Insertion Sort work (using the last six digits of my Student ID (438786) as Input Data Sample):

Index Number:	0	1	2	3	4	5
Input Data:	4	3	8	7	8	6
Index 0:	4	3	8	7	8	6
Index 1:	3	4	8	7	8	6
Index 2:	3	4	8	7	8	6
Index 3:	3	4	7	8	8	6
Index 4:	3	4	7	8	8	6
Index 5:	3	4	7	8	6	8
	3	4	7	6	8	8
	3	4	6	7	8	8
Sorted Array:	3	4	6	7	8	8

Highlighted green records to the left are always sorted. We start with the record in position 0 in the sorted part, and we will be moving the record in position 1 (in blue) to the left until it is sorted.

Move the blue record to the left until it reaches the correct position. In this position 1 is 3, so we swap 3 with 4. 3 is now in position 0 (index 0). Now we are on position 2, move the blue record to the left until it reaches the correct position. Same thing for position 3, move the blue record to the left until it reaches the correct position. In this position 3 is 7, swap 7 with 8 (position 2). 7 is now on position 2, and 8 is now on position 3. Now on position 4, move the blue record to the left until it reaches the correct position. Same thing for position 5. In this position 5 is 6, so we swap 6 with 8 (position 4). Now the blue record move to the left until it reaches the correct position. 6 (position 4) is swapped with 8 (position 3). 6 (position 3) is swapped with 7 (position 2). The array is now sorted: 346788.

Counting Sort

Counting Sort is a non-comparison-based sorting algorithm that sort the elements by counting the number of occurrences for each individual element in the list. It records how many times each individual element value occurs. it uses this information to determine the correct position of each element in the sorted output. Counting Sort assumes that the input elements are non-negative integers. It finds the least and highest values in the input list to determine the range of elements. It then calculates the sum of counts in the count array to establish the positions of elements in the sorted output. The sorted output is listed by placing each element in its right position based on the information from the count array.

Counting Sort time complexity is O(n+k), where n is the number of elements in the array and k is the range of elements. It is useful for sorting integers when the range is not bigger than the number of elements. In time complexity, the Counting Sort is stable and linear, making it useful for some cases where its input aligns with its requirements.

<u>Diagram of how Counting Sort work (using the last six digits of my Student ID (438786) as Input Data Sample):</u>

Input Number:

4	3	8	7	8	6

Step 1: Find the highest number

Max:

8

Step 2: Fill Index array with the number of occurrences

Index:	1	2	3	4	5	6	7	8
Value:	0	0	1	1	0	1	1	2

Number of occurrences of 1 is 0.

Number of occurrences of 2 is 0.

Number of occurrences of 3 is 1.

Number of occurrences of 4 is 1.

Number of occurrences of 5 is 0.

Number of occurrences of 6 is 1.

Number of occurrences of 7 is 1.

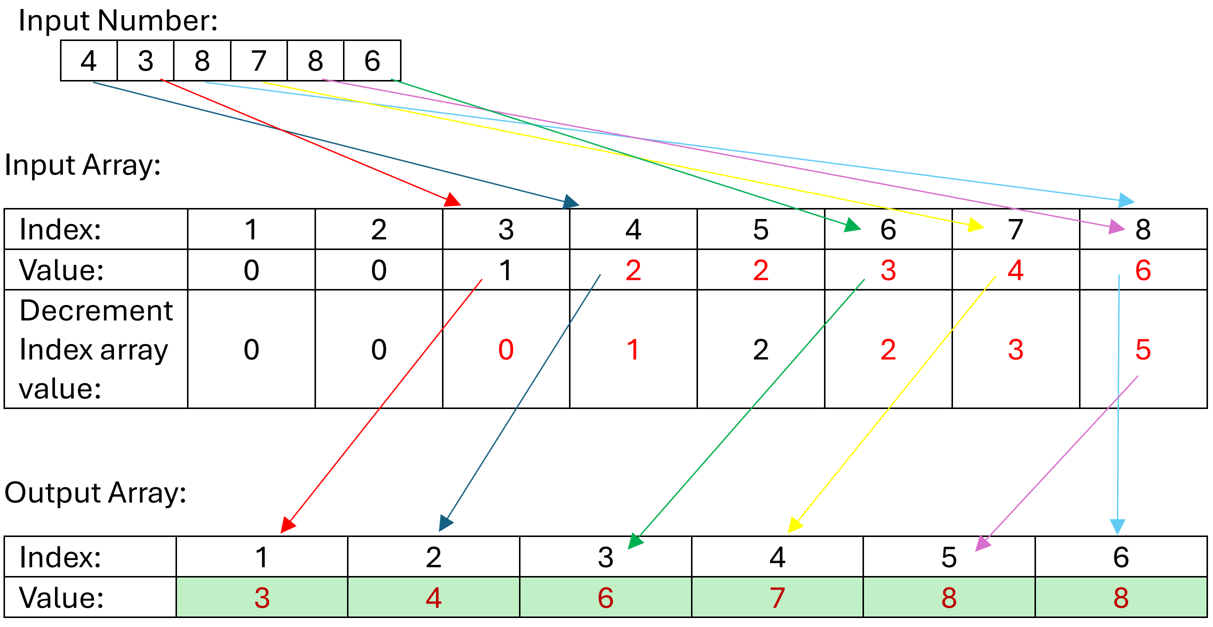
Number of occurrences of 8 is 2.

Step 3: Sum up Index array values with predecessor values

Index:	1	2	3	4	5	6	7	8
Value:	0	0	1	2	2	3	4	6

First occurrence is 0. Second occurrence are 0+0=0. Third occurrence are 0+1=1. Fourth occurrence are 1+1=2, which sum up to 2. Fifth occurrence are 2+0=2, which sum up to 2. Sixth occurrence are 2+1=3, which sum up to 3. Seventh occurrence are 3+1=4, which sum up to 4. Eighth occurrence are 4+2=6, which sum up to 6.

Step 4: Map the input-index-output arrays:



Input Number:

4	3	8	7	8	6

Input Array:

Index:	1	2	3	4	5	6	7	8
Value:	0	0	1	2	2	3	4	6
Decrement Index array value:	0	0	0	1	2	2	3	5

Output Array:

Index:	1	2	3	4	5	6
Value:	3	4	6	7	8	8

The first element is 4, find index 4 in the input array. The value of input array is 2, find index 2 in the output array and fill the value of the output array, which is 4. After this, the input array value decreases by 1, in this case 2-1=1 (Decrement index array value).

The second element is 3, find index 3 in the input array. The value of input array is 1, find index 1 in the output array and fill the value of the output array, which is 3. After this, the input array value decreases by 1, in this case 1-1=0 (Decrement index array value).

The third element is 8, find index 8 in the input array. The value of input array is 6, find index 6 in the output array and fill the value of the output array, which is 8. After this, the input array value decreases by, in this case 6-1=5 (Decrement index array value).

The fourth element is 7, find index 7 in the input array. The value of input array is 4, find index 4 in the output array and fill the value of the output array, which is 7. After this, the input array value decreases by 1, in this case 4-1=3 (Decrement index array value).

The fifth element is 8, find index 8 in the input array. The value of input array is 5 (Decrement index array value), find index 5 in the output array and fill the value of the output array, which is 8. After this, the input array value decreases by 1, in this case 5-1=4 (2nd Decrement index array value).

The sixth element is 6, find index 6 in the input array. The value of input array is 3, find index 3 in the output array and fill the value of the output array, which is 6. After this, the input array value decreases by 1, in this case 3-1=2 (Decrement index array value).

Sorted Array:

3	4	6	7	8	8

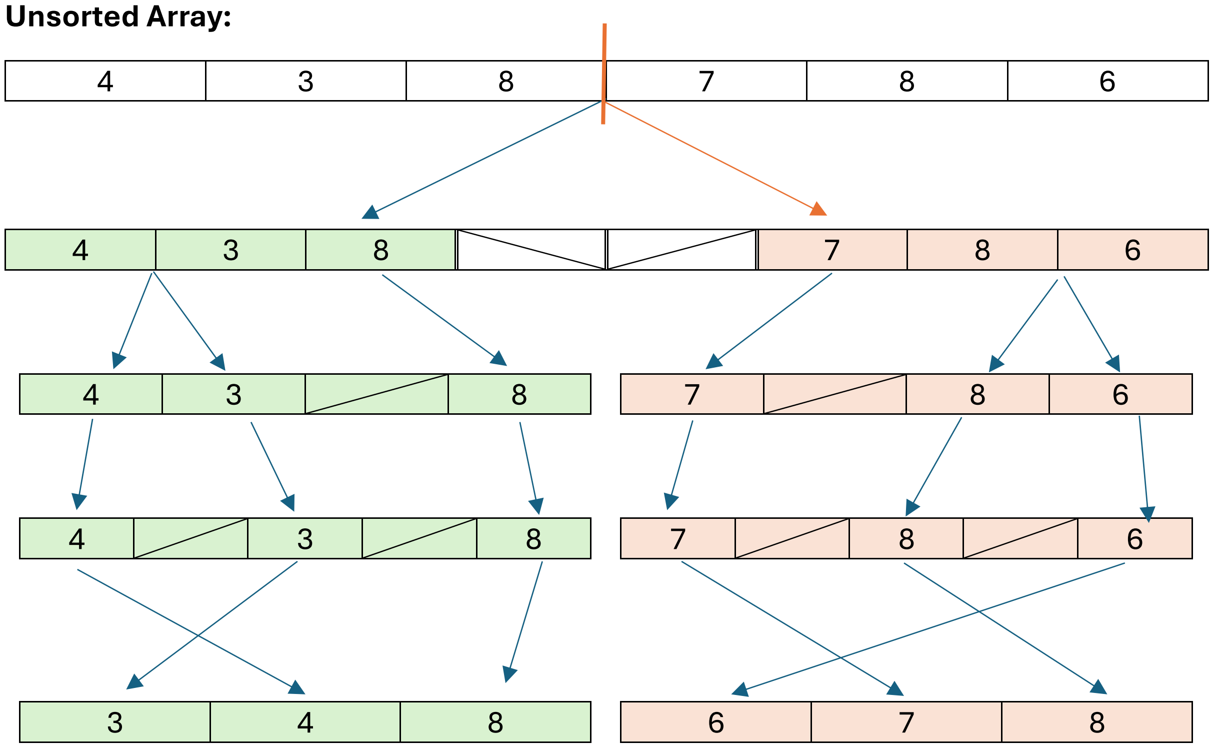
Merge Sort

Merge Sort is a popular comparison-based sorting algorithm known for its efficiency and stable performance. It uses the divide-and-conquer approach to sorting, where it divides the input list into two halves. It then combines the sub list back together in the correct order to produce the sorted output list. The input is divided into two parts and the process is repeated recursively until there is only one element left in every part. Merge then sorted halves back together to create a final list.

The Merge Sort has a time complexity of O(n log n) in all cases (best, average and worst-case scenarios). It is useful for sorting large datasets and produces stable performance across various input sizes. Merge Sort also has space complexity of O(n).

Diagram of how Merge Sort work (using the last six digits of my Student ID (438786) as Input Data Sample):

Unsorted Array:



Sorted Array:

3	4	6	7	8	8

Steps:

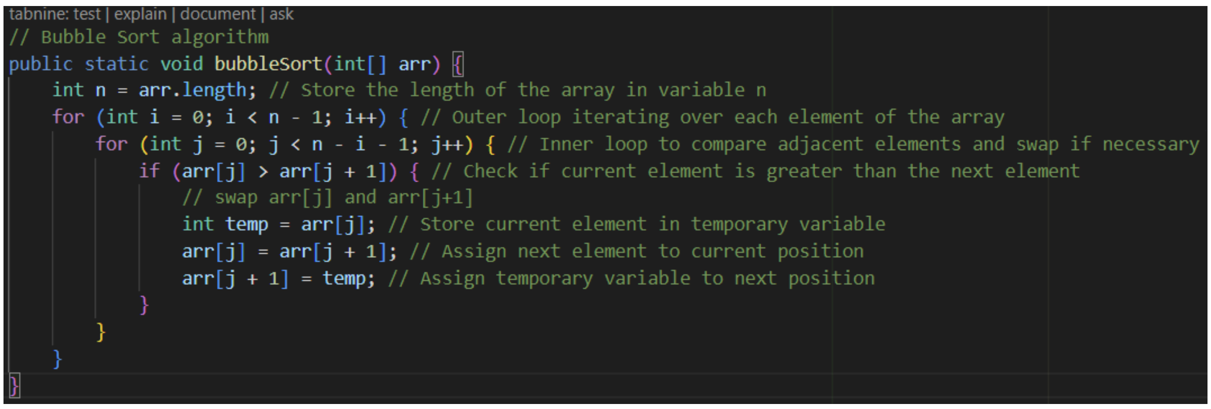
- Split the 6 elements into half.
- Split the selected elements (as evenly as possible)
- Split the sub arrays into two halves.
- Merge it into sorted sub arrays.
- Merge the sorted sub arrays into sorted arrays.

Implementation & Benchmarking

This section contains details of the implementation process of the application and introduces the benchmarking results. The application includes implementations of five sorting algorithms, each being tested using a main method. To benchmark the algorithms, the arrays of randomly generated integers with varying input sizes ‘n’ (e.g., n=10, n=100, n=500, …, n=10000) were used to examine the effect of input size on runtime. The program calculated the runtime of each algorithm 10 times and produced the result of the average runtime (3 decimal places, milliseconds) for each individual algorithm and input the size to the console at the end of completion. For the benchmarking outcomes, the value portrays the average time, measured in milliseconds, of randomly generated arrays of size ‘n’.

This part of the sections shows how the implementation of each sorting algorithm featured in this report is done, along with a clear explanation. Here, you'll see the code snippets in Java for each algorithm, followed by brief explanations of the variables and structures employed, along with their specific purposes.

Bubble Sort Implementation:

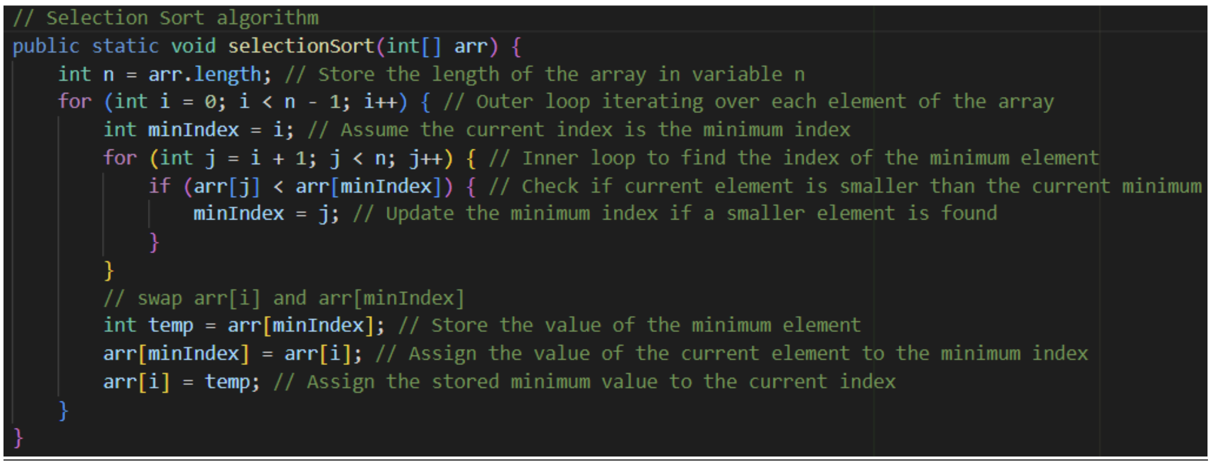


```
tabnine: test | explain | document | ask
// Bubble Sort algorithm
public static void bubbleSort(int[] arr) {
    int n = arr.length; // Store the length of the array in variable n
    for (int i = 0; i < n - 1; i++) { // Outer loop iterating over each element of the array
        for (int j = 0; j < n - i - 1; j++) { // Inner loop to compare adjacent elements and swap if necessary
            if (arr[j] > arr[j + 1]) { // Check if current element is greater than the next element
                // swap arr[j] and arr[j+1]
                int temp = arr[j]; // Store current element in temporary variable
                arr[j] = arr[j + 1]; // Assign next element to current position
                arr[j + 1] = temp; // Assign temporary variable to next position
            }
        }
    }
}
```

- ‘public static void bubbleSort(int[] arr) {‘: This states a method named ‘bubbleSort’ that takes an integer array ‘arr’ as input and doesn’t bring back any value. It’s a static method, meaning it is part of to the class itself.
- ‘int n = arr.length;’ This line computes the length of the input array ‘arr’ and keeps it in the variable ‘n’. This length portrays the number of elements in the array.
- ‘for (int I = 0; I < n – 1; i++) { ‘: This starts a loop that repeats over each element of the array except for the last one. It starts from the first element (index 0) and goes up to the second-to-last element (index ‘n – 2’).
- ‘for (int I = 0; j < n – I – 1; j++){‘: Inside the outer loop, another loop is started to compare elements and perform the swapping operation. It also starts from the first element (index 0) and iterates up to ‘n – I -2’.
- ‘if (arr[j] > arr[j + 1]) {‘: This condition examines if the current element (‘arr[j]’) is greater than the next element (‘arr[j + 1]’). If this situation is true, it means the elements are in the wrong order and need to be swapped.

- ‘int temp = arr[j + 1];’: If the circumstance in the previous line is true, the present element is stored in temporary variable ‘temp’.
- ‘arr[j] = arr[j + 1];’: The value of the next element is allocated to the current element, successfully swapping their positions.
- ‘arr[j + 1] = temp;’: Finally, the value is kept in the temporary variable ‘temp’ is allocated to the next element, bringing the swapping operation to a completion.
- These steps are performed again for each pair of close elements in the array, with the outer loop assuring that the largest element “bubbles up” to its correct position in each cycle.

Selection Sort Implementation:

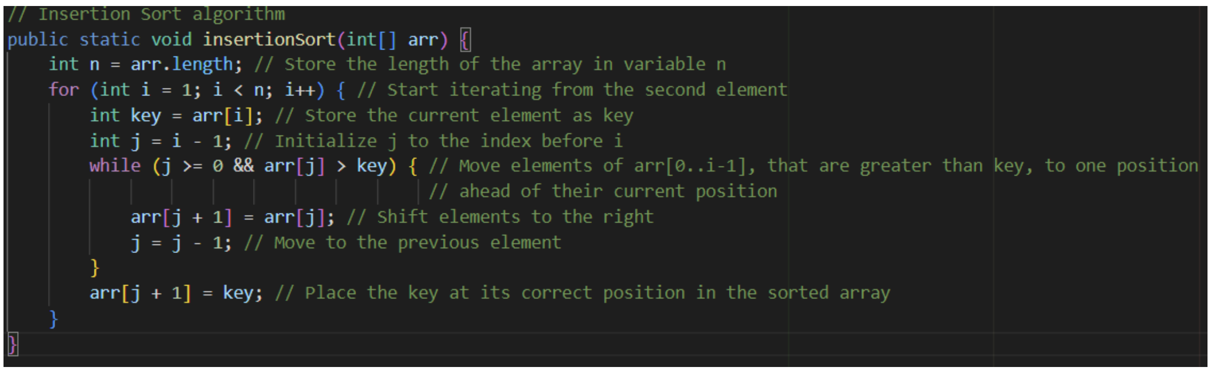


```java
// Selection Sort algorithm
public static void selectionSort(int[] arr) {
    int n = arr.length; // Store the length of the array in variable n
    for (int i = 0; i < n - 1; i++) { // Outer loop iterating over each element of the array
        int minIndex = i; // Assume the current index is the minimum index
        for (int j = i + 1; j < n; j++) { // Inner loop to find the index of the minimum element
            if (arr[j] < arr[minIndex]) { // Check if current element is smaller than the current minimum
                minIndex = j; // Update the minimum index if a smaller element is found
            }
        }
        // swap arr[i] and arr[minIndex]
        int temp = arr[minIndex]; // Store the value of the minimum element
        arr[minIndex] = arr[i]; // Assign the value of the current element to the minimum index
        arr[i] = temp; // Assign the stored minimum value to the current index
    }
}
```

- ‘int n = arr.length;’: This line computes the length of the input array ‘arr’ and keeps it in the variable ‘n’. This length portrays the number of elements in the array.
- ‘for (int I = 0; I < n – 1; i++) {‘: This starts a loop that repeats over each element of the array except for the last one. It starts from the first element of the array (index 0) and goes up to the second-to-last element (index ‘n – 2’).
- ‘int minIndex = I;’: Inside the outer loop, a variable ‘minIndex’ is set up to the current index ‘I’, assuming it to be the index of the smallest element.
- ‘for (int j = I + 1; j < n; j++) {‘: Another loop commenced with the outer loop to find the index of the minimum element in the unsorted part of the array. It starts from the element next to the current element (‘I + 1’) and goes up to the last element (‘n – 1’).
- ‘if (arr[j] < arr[minIndex]) {‘: This condition checks if the present element (‘arr[j]’) is lesser than the element at the current minimum index (‘arr[minIndex]’). If this state is true, it means that the smaller element is found, and the ‘minIndex’ is refreshed to ‘j’.
- ‘int temp = arr[minIndex];’: After searching the index of the smallest element, its value is saved in a temporary variable ‘temp’.

- ‘arr[minIndex] = arr[i];’: The value of the current element (‘arr[i]’) is then assigned to the position of the smallest element, successfully swapping their positions.
- ‘arr[i] = temp;’: Finally, the value is kept in the temporary variable ‘temp’ (the smallest value) is designated to the position previously occupied by the current element, completing the swap.
- These steps are performed again for each cycle of the outer loop, concluding in the array being sorted in rising order.

Insertion Sort Implementation:

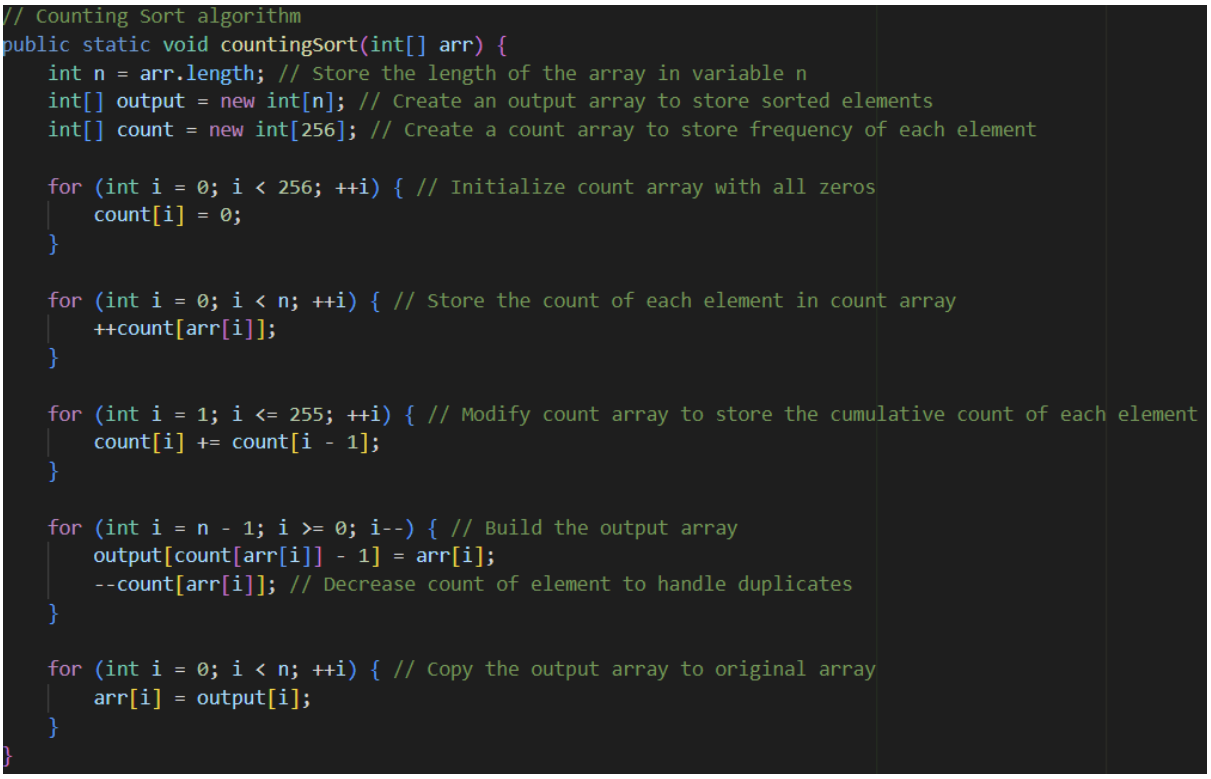


```
// Insertion Sort algorithm
public static void insertionSort(int[] arr) {
    int n = arr.length; // Store the length of the array in variable n
    for (int i = 1; i < n; i++) { // Start iterating from the second element
        int key = arr[i]; // Store the current element as key
        int j = i - 1; // Initialize j to the index before i
        while (j >= 0 && arr[j] > key) { // Move elements of arr[0..i-1], that are greater than key, to one position
                                         // ahead of their current position
            arr[j + 1] = arr[j]; // Shift elements to the right
            j = j - 1; // Move to the previous element
        }
        arr[j + 1] = key; // Place the key at its correct position in the sorted array
    }
}
```

- ‘int n = arr.length;’: This line computes the length of the input array `arr` and stores it in the variable `n`. This length portrays the number of elements in the array.
- ‘for (int i = 1; i < n; i++) {‘: This starts a loop that repeats over each element of the array commencing from the second element (index 1). The first element (index 0) is considered already sorted.
- ‘int key = arr[i];’: Inside the loop, the current element (‘arr[i]’) is placed in a variable ‘key’. This element will be added into its correct position in the sorted subarray.
- ‘int j = i - 1;’: Another variable ‘j’ is set up to the index before ‘I’, portraying the last index of the sorted subarray.
- ‘while (j >= 0 && arr[j] > key) {‘: This starts a loop that keep going as long as ‘j’ is larger than or equal to 0 (making sure the loop doesn't go out of bounds) and the element at index ‘j’ is greater than the ‘key’. This loop changes the elements of the sorted subarray that are larger than ‘key’ to the right to make room for insertion.
- ‘arr[j + 1] = arr[j];’: Inside the loop, each element that are larger than ‘key’ is moved one position to the right to make room for insertion.
- ‘j = j - 1;’: After moving, ‘j’ is reduced to move to the previous element in the sorted subarray.
- ‘arr[j + 1] = key;’: Once the correct position for ‘key’ is found (where ‘arr[j]’ is no longer larger than ‘key’), ‘key’ is placed into the sorted subarray at index ‘j + 1’.

- These steps are performed again for each cycle of the outer loop, resulting in the array being sorted in rising order.

Counting Sort Implementation:

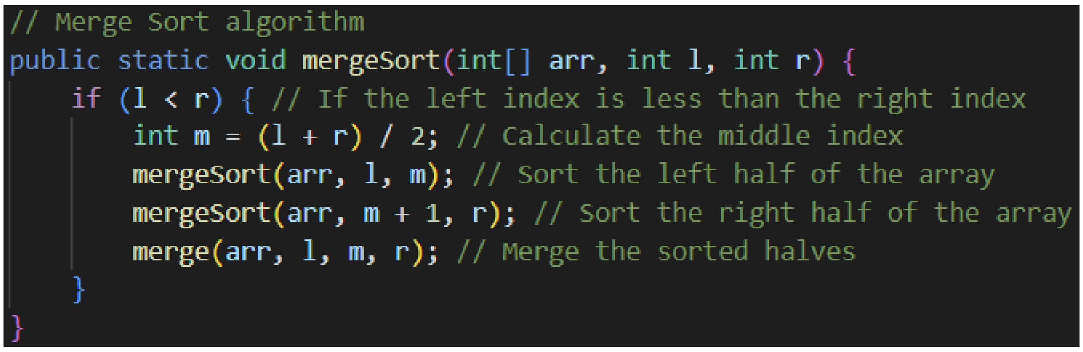


```java
// Counting Sort algorithm
public static void countingSort(int[] arr) {
    int n = arr.length; // Store the length of the array in variable n
    int[] output = new int[n]; // Create an output array to store sorted elements
    int[] count = new int[256]; // Create a count array to store frequency of each element

    for (int i = 0; i < 256; ++i) { // Initialize count array with all zeros
        count[i] = 0;
    }

    for (int i = 0; i < n; ++i) { // Store the count of each element in count array
        ++count[arr[i]];
    }

    for (int i = 1; i <= 255; ++i) { // Modify count array to store the cumulative count of each element
        count[i] += count[i - 1];
    }

    for (int i = n - 1; i >= 0; i--) { // Build the output array
        output[count[arr[i]] - 1] = arr[i];
        --count[arr[i]]; // Decrease count of element to handle duplicates
    }

    for (int i = 0; i < n; ++i) { // Copy the output array to original array
        arr[i] = output[i];
    }
}
```

- ‘int n = arr.length;’: This line computes the length of the input array ‘arr’ and stores it in the variable ‘n’. This length portrays the number of elements in the array.
- ‘int[] output = new int[n];’: This produces a new integer array named ‘output’ with the same length as the input array. This array will keep the sorted elements.
- ‘int[] count = new int[256];’: This produces a new integer array named ‘count’ with a length of 256. This array will keep the frequency of each element in the input array.
- ‘for (int i = 0; i < 256; ++i) {‘: This loop set up all elements of the ‘count’ array to zero. It make sure that the count array is unoccupied and set to store the frequency of each element.
- ‘for (int i = 0; i < n; ++i) {‘: This loop repeats through each element of the input array ‘arr’. For each element, it increases the matching count in the ‘count’ array.
- ‘for (int i = 1; i <= 255; ++i) {‘: This loop adjusts the ‘count’ array to keep track of the total count of each element. It makes sure that each element's count portrays the number of elements fewer than or equal to it.
- ‘for (int i = n - 1; i >= 0; i--) {‘: This loop creates the output array by placing each element from the input array at its right position in the output array, based on the total count stored in the ‘count’ array.

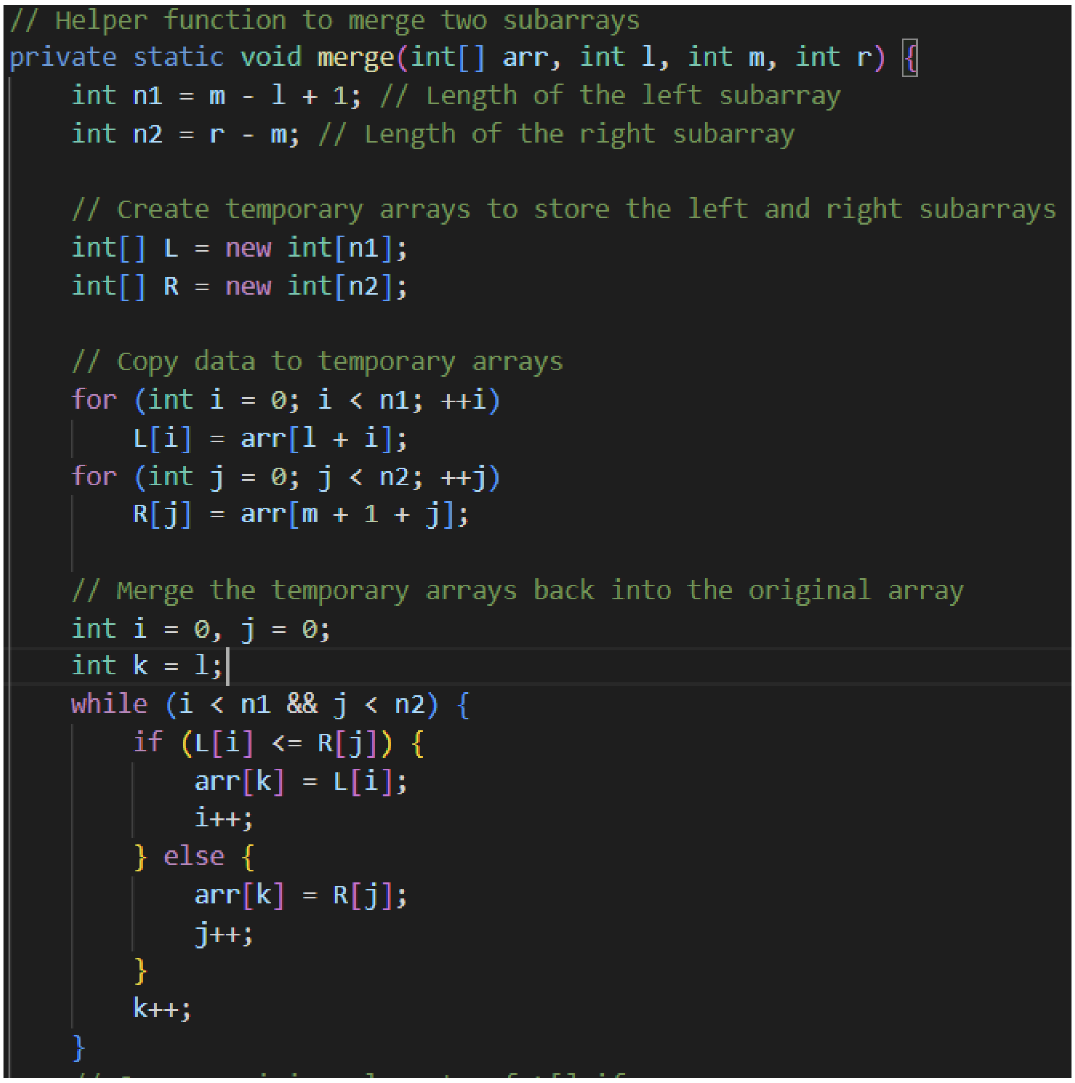
- ‘output[count[arr[i]] - 1] = arr[i];’: Inside the loop, it designated the current element from the input array to its right position in the output array, using the total count stored in the ‘count’ array.
- ‘--count[arr[i]];’: It reduces the count of the current element in the ‘count’ array to manage copy elements.
- ‘for (int i = 0; i < n; ++i) {‘: Finally, this loop copies the sorted elements from the output array back to the initial input array.

Merge Sort Implementation:



```java
// Merge Sort algorithm
public static void mergeSort(int[] arr, int l, int r) {
    if (l < r) { // If the left index is less than the right index
        int m = (l + r) / 2; // Calculate the middle index
        mergeSort(arr, l, m); // Sort the left half of the array
        mergeSort(arr, m + 1, r); // Sort the right half of the array
        merge(arr, l, m, r); // Merge the sorted halves
    }
}
```

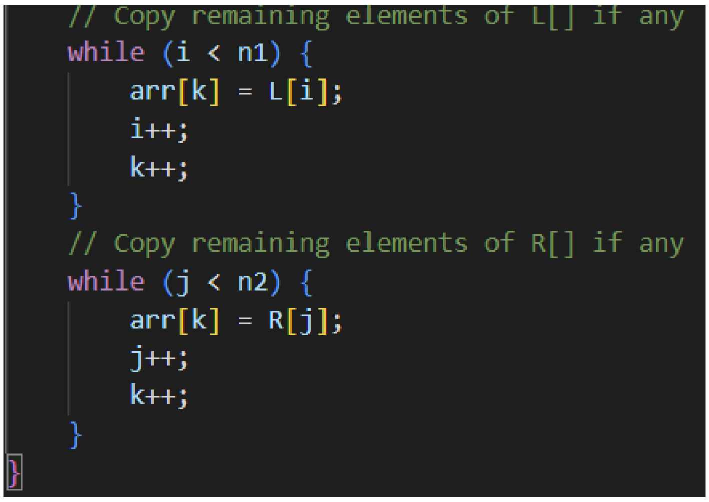
- ‘public static void mergeSort(int[] arr, int l, int r) {‘: This states a method named ‘mergeSort’ that takes an integer array ‘arr’ and two integer parameters ‘l’ and ‘r’, portraying the left and right indexes of the array to be sorted, in that order. It doesn’t bring back any value.
- ‘if (l < r) {‘: This examines if the left index ‘l’ is fewer than the right index ‘r’, suggesting that there are elements to be sorted between these indexes.
- ‘int m = (l + r) / 2;’: This computes the middle index ‘m’ of the array to separate it into two halves.
- ‘mergeSort(arr, l, m);’: This continuously calls the ‘mergeSort’ method on the left half of the array to organise it.
- ‘mergeSort(arr, m + 1, r);’: This continuously calls the ‘mergeSort’ method on the right half of the array to organise it.
- ‘merge(arr, l, m, r);’: This merges the organised left and right halves of the array by using the ‘merge’ helper function.



```java
// Helper function to merge two subarrays
private static void merge(int[] arr, int l, int m, int r) {
    int n1 = m - l + 1; // Length of the left subarray
    int n2 = r - m; // Length of the right subarray

    // Create temporary arrays to store the left and right subarrays
    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];

    // Merge the temporary arrays back into the original array
    int i = 0, j = 0;
    int k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

- 'private static void merge(int[] arr, int l, int m, int r){': This states a private helper method named 'merge' that obtains an integer array 'arr' and three integer parameters 'l', 'm', and 'r', portraying the indexes of the subarrays to be merged.
- 'int n1 = m – l + 1;': This computes the length of the left subarray.
- 'int n2 = r – m;': This computes the length of the right subarray.
- 'int[] L = new int[n1];': This produces a temporary array 'L' to keep the left subarray.
- 'int[] R = new int[n2];': This produces a temporary array 'R' to keep the right subarray.
- 'for (int I = 0; I < n1; ++i) L[i] = arr [l +i];': This replicates the elements of the left subarray from the initial array to the temporary array 'L'.
- 'for (int j = 0; j < n2; ++j) R[j] = arr [m + 1 + j];': This replicates the elements of the right subarray from the initial array to the temporary array 'R'.
- This following part of the section of the code is accountable for combining the two sorted subarrays 'L' and 'R' back into the initial array 'arr'.
- 'int I = 0, j = 0;': It set up two pointers 'I' and 'j' to the beginning of the left and right subarrays, in that order.

- ‘int k = l;’: It set up a third pointer ‘k’ to the start index of the initial array where the combined elements will be positioned.
- ‘while (I < n1 && j < n2){‘: This loop keeps going until either of the subarrays ‘L’ and ‘R’ is fully explored.
- ‘if (L[i] <= R[j]) { arr[k] = L[i]; i++; } else { arr[k] = R[j]; j++; }’: It compares the elements at indexes ‘I’ and ‘j’ of the left and right subarrays in that order. The lesser element in ‘L[i]’ and ‘R[j]’ is replicated into the intial array ‘arr’ at index ‘k’, and the related pointer (‘I’ or ‘j’) is increased to shift to the next element in that subarray.
- ‘k++;’: After replicating an element from either ‘L’ or ‘R’ into ‘arr’, the index ‘k’ is increased to point to the next area in ‘arr’ where the next combined element will be positioned.
- This loop successfully combines the two sorted subarrays ‘L’ and ‘R’ into the initial array ‘arr’ while upholding the sorted order.



```
    // Copy remaining elements of L[] if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    // Copy remaining elements of R[] if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

- ‘while (I < n1) { arr [k] = L[i]; i++; k++; }’: This loop replicates any leftover elements in the left subarray ‘L’ back into the initial array ‘arr’. It cycles while there are still elements left in the left subarray (‘I < n1’), replicating each element from ‘L’ to ‘arr’ at index ‘k’, then expanding both ‘I’ and ‘k’ to shift to the next element.
- ‘while (j < n2) { arr[k] = R[j]; j++; k++; }’: Similarly, this loop replicates any leftover elements in the right subarray ‘R’ back into the initial array ‘arr’. It cycles while there are still elements left in the right subarray (‘j < n2’), replicating each element from ‘R’ to ‘arr’ at index ‘k’’, then expanding both ‘j’ and ‘k’ to move to the next element.
- These two loops make sure that any leftover elements in the left or right subarrays, which were not yet replicated during the combining process, are correctly positioned in the final sorted array ‘arr’.

Benchmarking Results:

The benchmarking outcomes from the five sorting algorithms are displayed beneath Each value aligns to the average runtime, measured in milliseconds, throughout ten randomly generated arrays of size 'n'.

Bubble Sort Result Table:

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble	0.123	0.16	0.347	0.479	0.66	0.855	2.976	6.944	11.994	21.532	36.631	54.798	78.474

Selection Sort Result Table:

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Selection	0.039	0.096	0.21	0.204	0.21	0.495	1.3	2.567	4.383	6.873	9.256	12.843	16.81

Insertion Sort Result Table:

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Insertion	0.039	0.106	0.119	0.176	0.171	0.133	0.233	0.572	1.022	1.585	2.966	2.987	3.958

Counting Sort Result Table:

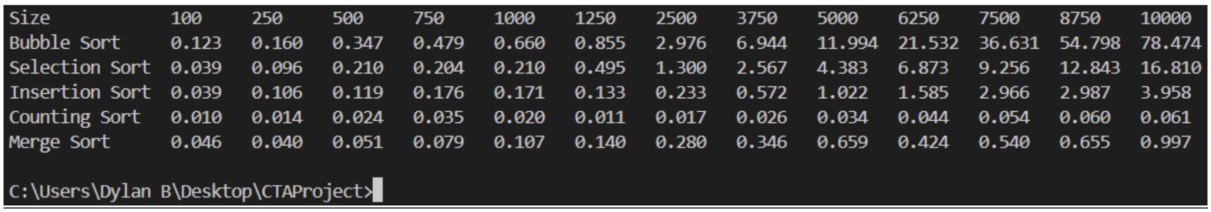
Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Counting	0.01	0.014	0.024	0.035	0.02	0.011	0.017	0.026	0.034	0.044	0.054	0.06	0.061

Merge Sort Result Table:

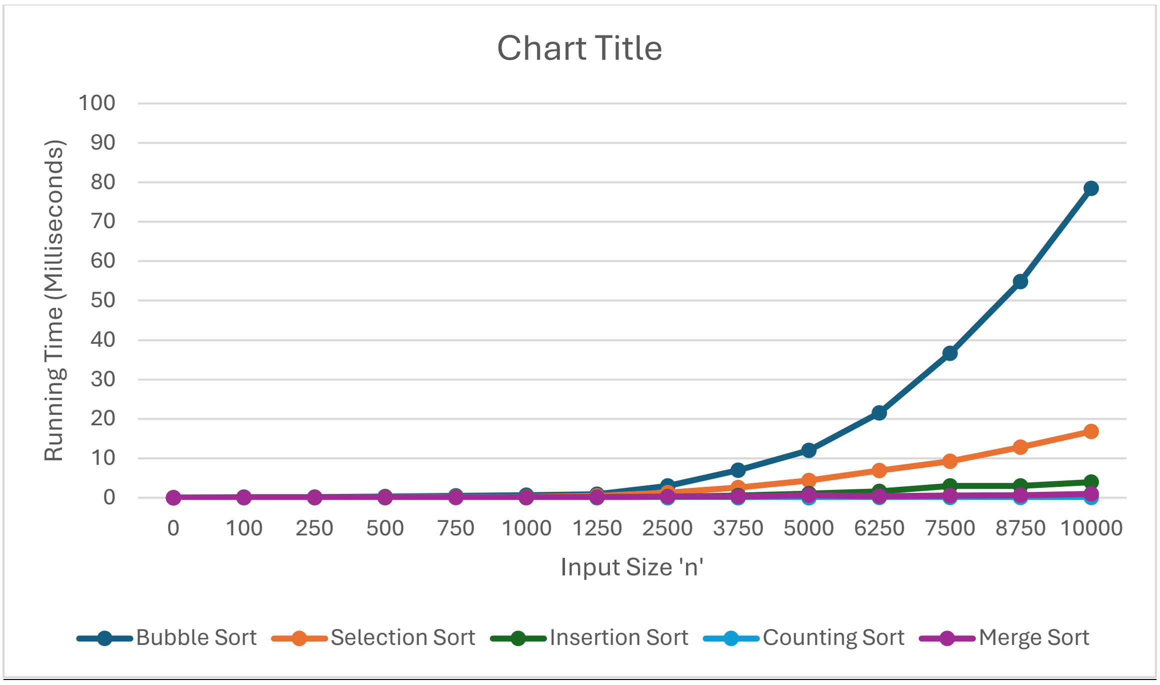
Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Merge	0.046	0.04	0.051	0.079	0.107	0.14	0.28	0.346	0.659	0.424	0.54	0.655	0.977

Five Sort Result Table:

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble	0.123	0.16	0.347	0.479	0.66	0.855	2.976	6.944	11.994	21.532	36.631	54.798	78.474
Selection	0.039	0.096	0.21	0.204	0.21	0.495	1.3	2.567	4.383	6.873	9.256	12.843	16.81
Insertion	0.039	0.106	0.119	0.176	0.171	0.133	0.233	0.572	1.022	1.585	2.966	2.987	3.958
Counting	0.01	0.014	0.024	0.035	0.02	0.011	0.017	0.026	0.034	0.044	0.054	0.06	0.061
Merge	0.046	0.04	0.051	0.079	0.107	0.14	0.28	0.346	0.659	0.424	0.54	0.655	0.977



```
Size            100    250    500    750    1000   1250   2500   3750   5000   6250   7500   8750   10000
Bubble Sort     0.123  0.160  0.347  0.479  0.660  0.855  2.976  6.944  11.994 21.532 36.631 54.798 78.474
Selection Sort  0.039  0.096  0.210  0.204  0.210  0.495  1.300  2.567  4.383  6.873  9.256  12.843 16.810
Insertion Sort  0.039  0.106  0.119  0.176  0.171  0.133  0.233  0.572  1.022  1.585  2.966  2.987  3.958
Counting Sort   0.010  0.014  0.024  0.035  0.020  0.011  0.017  0.026  0.034  0.044  0.054  0.060  0.061
Merge Sort      0.046  0.040  0.051  0.079  0.107  0.140  0.280  0.346  0.659  0.424  0.540  0.655  0.997

C:\Users\Dylan B\Desktop\CTAProject>
```



According to the results of the time complexity of each of the five algorithms, we can see the following differences between the five sorting algorithms throughout distinct input sizes:

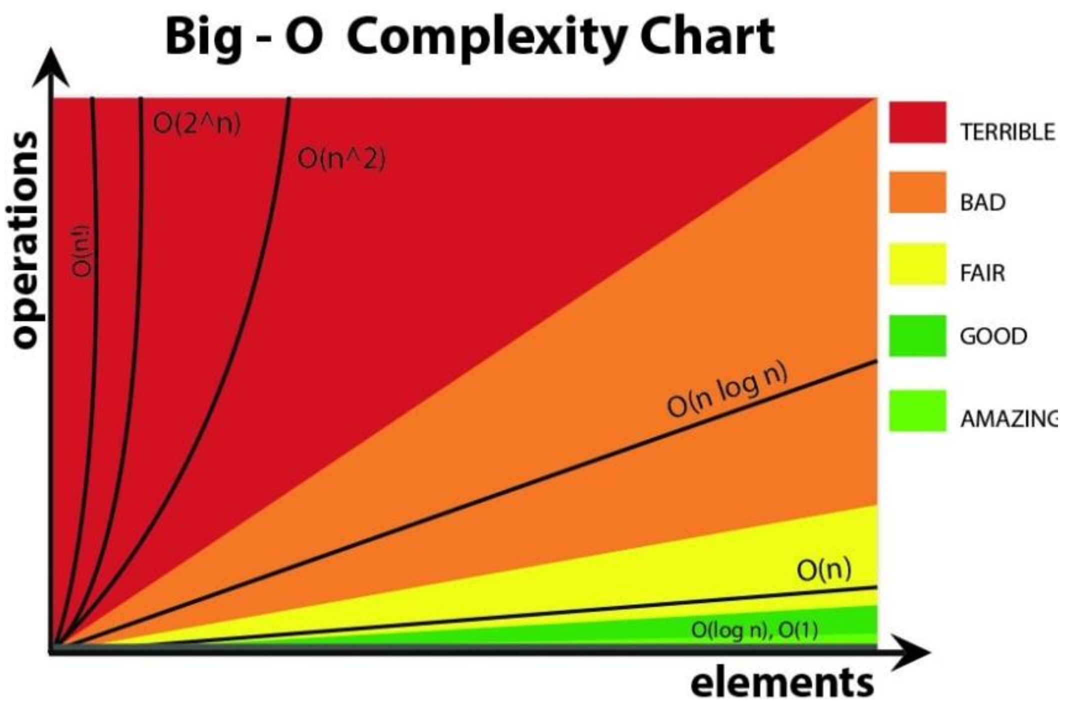
1. Bubble Sort: Bubble Sort commonly displays the highest runtime among the five algorithms for all input sizes. Its runtime rises considerably with larger input sizes, showing signs of poor flexibility.

2. Selection Sort: Selection sort reveals reasonable runtime compared to Bubble Sort, but still has a proportionally high runtime for larger input sizes.

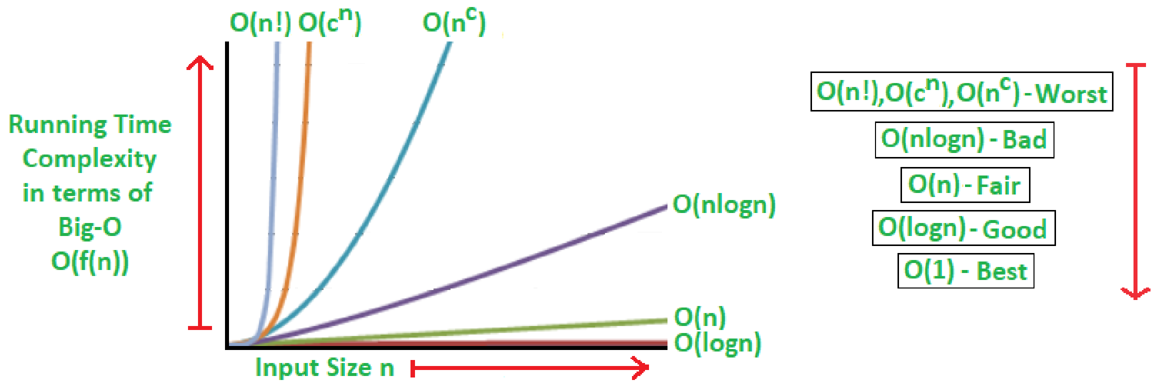
3. Insertion Sort: Insertion Sort typically performs better in comparison to Bubble and Selection Sorts, particularly in relation to smaller input sizes. However, its runtime also rises with larger input sizes, although not as drastically as Bubble Sort.

4. Counting Sort: Counting Sort shows the lowest runtime throughout all input sizes, highlighting a superior performance in comparison to the other algorithms. It maintains a steadily low runtime even as the input size rises.

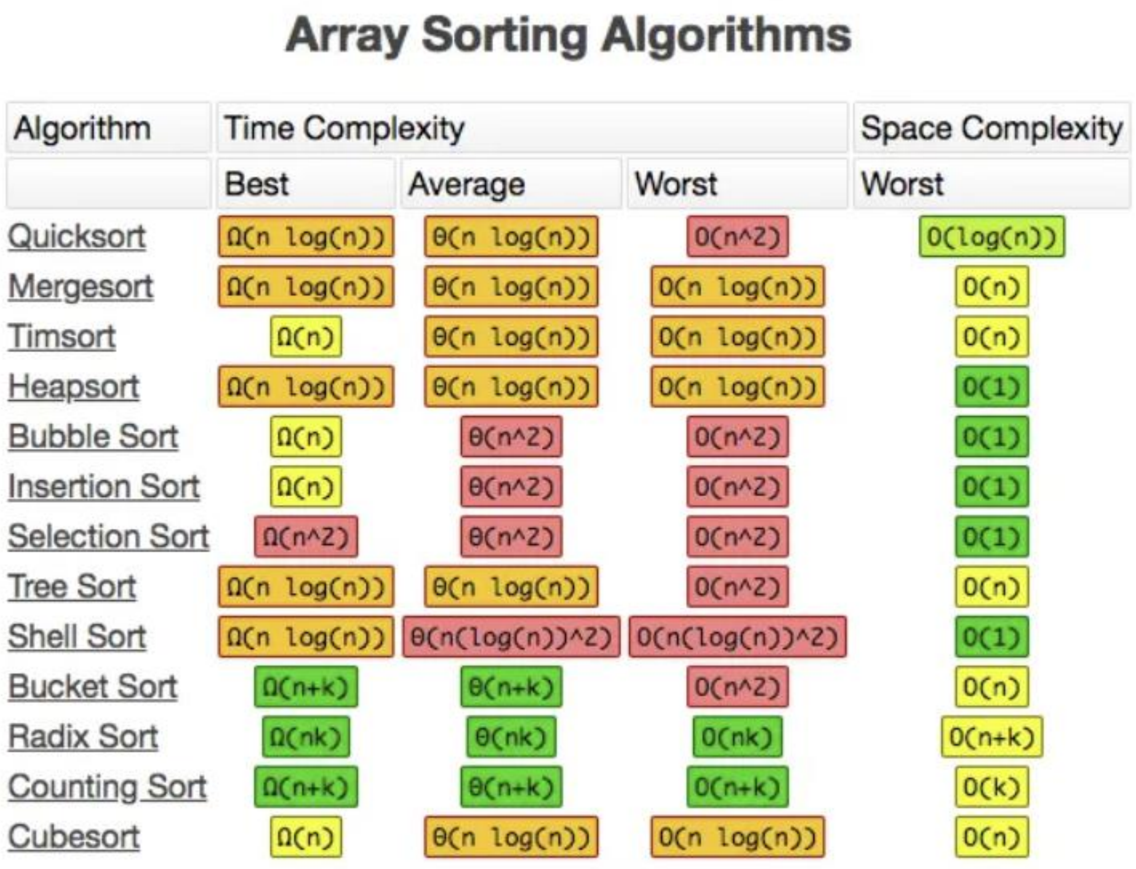
5. Merge Sort: Merge Sort displays reasonable to low runtime throughout different input sizes. It executes relatively well for larger input sizes in comparison to Bubble and Selection Sorts but is marginally slower than Insertion Sort for smaller input sizes.



https://dev.to/madisonstankevich/big-o-notation-a-brief-overview-for-the-beginner-1o13



https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

https://levelup.gitconnected.com/algorithm-complexity-and-the-big-o-cb703e093faa

In summary, Counting Sort and Merge Sorts seems to be the most effective algorithm regarding runtime across all input sizes, while Bubble Sort steadily executes the least favourable.

References:

- https://www.geeksforgeeks.org/sorting-algorithms/
- https://www.geeksforgeeks.org/bubble-sort/?ref=lbp
- https://www.geeksforgeeks.org/stable-selection-sort/?ref=lbp
- https://www.youtube.com/watch?v=nmhjrI-aW5o&ab_channel=GeeksforGeeks
- https://www.youtube.com/watch?v=xWBP4lzkoyM&ab_channel=GeeksforGeeks
- https://www.youtube.com/watch?v=OGzPmgsI-pQ&ab_channel=GeeksforGeeks
- https://www.youtube.com/watch?v=JSceec-wEyw&ab_channel=GeeksforGeeks
- https://www.youtube.com/watch?v=rbbTd-gkajw&t=427s&ab_channel=CodingwithLewis
- https://www.youtube.com/watch?v=7zuGmKfUt7s&ab_channel=GeeksforGeeks
- https://www.youtube.com/watch?v=peLS-S23TvE&ab_channel=Codearchery
- https://www.youtube.com/watch?v=UqmKiz2P0Lw&ab_channel=Geekific
- https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/
- https://en.wikibooks.org/wiki/A-level_Computing_2009/AQA/Problem_Solving,_Programming,_Operating_Systems,_Databases_and_Networking/Problem_Solving/Big_O_Notation
- https://fastercapital.com/topics/the-importance-of-optimal-time-complexity-in-sorting-algorithms.html
- https://www.knowledgehut.com/blog/programming/time-complexity