Computational Thinking - Sorting Algorithms Project

Introduction

Overview of Sorting Algorithms

An algorithm is a series of steps designed to solve a computational problem, like following a recipe in a cookbook. Elements in a list are sorted using sorting algorithms and are fundamental to programming as they are used in many applications. Algorithms can handle various element types, including numbers, letters, complex data structures and objects, making them essential for applications ranging from e-commerce platforms to search engines. (Eldridge 2024)

This report will explore the nature of five sorting algorithms, focusing on computational complexity, meaning how much time they take to complete and how much space they take up while running. By understanding these complexities, the report aims to demonstrate how algorithms can be compared and how the results can be predicted to show their effectiveness in real-world applications.

Basic Concepts in Sorting

Time & Space Complexity

Firstly, we must discuss the computational complexities, time and space, which underpin the effectiveness of algorithms in real-world applications. The time complexity of an algorithm measures how the algorithm's running time increases with the size of the input data. This is described using Big-O notation, which assesses the algorithm's best, average, and worst-case performance times. Measuring time complexity has practical implications, as an algorithm that scales poorly with increasing input size can become impractical to use with large datasets. This is a critical aspect of deciding which algorithm would be suitable for real-world applications.

Alongside time complexity, space complexity measures to the amount of memory that an algorithm requires in relation to the size of the input data. This is crucial as many applications operate under memory constrictions. By determining space complexity, we can make decisions on algorithms, ensuring that memory is handled effectively so the application runs smoothly. (Great Learning Team 2024)

Types of Sorting Algorithms

Comparison-Based vs. Non-Comparison-Based Sorting

We can categorise sorting algorithms into two broad categories, comparison-based and non-comparison-based.

Comparison-based sorting algorithms rely on comparator functions to establish the order of elements by comparing them directly. These algorithms can effectively sort various types of data, whether they are in numerical or lexical order.(Eldridge 2024).

Conversely, non-comparison-based sorting algorithms like Counting Sort work differently. They don't directly compare elements but instead depend on integer arithmetic operations on keys. This is why they are also called integer sorting algorithms. This method allows these algorithms to achieve linear time complexity, measured as O(n), resulting in faster sorting times compared to the O(n log n) complexity that comparison-based methods reach in their worst-case scenarios. (Paul 2021)

While non-comparison-based algorithms offer advantages regarding time efficiency, they often require adjustments to the data types they can process. For instance, Counting Sort is highly efficient with faster running times than any comparison-based method, but it is limited to sorting integer keys. (Eldridge 2024)

Another significant difference is in space complexity. Comparison-based algorithms can sort data in place, achieving a space complexity of O(1). Non-comparison-based algorithms typically need extra storage, usually O(n), to accommodate intermediate counting or bucketing steps. (Paul 2021)

Despite the potential for faster performance, choosing an algorithm in real-world applications often relies on the nature of the data and operational demands. While comparison-based algorithms like Merge Sort are favoured for their versatility, non-comparison algorithms can provide advantages when their input requirements are met. These factors are essential when considering which sorting algorithm to use, as they impact how elements are compared, how memory is utilised, and how the order of the data is maintained during the sorting process.

In-Place & Stable Sorting Considerations

In the context of sorting algorithms, it is essential to consider whether a sorting algorithm is sorted in-place, stable, or both, as these traits significantly impact their memory efficiency and data integrity. In-place sorting algorithms work directly within the original data structure, using only a minimal amount of additional space. This is especially advantageous in settings where conserving memory is a priority, making these algorithms highly useful regarding space usage. Examples of in-place sorting are Bubble Sort , Selection Sort and Insertion Sort. (Thakrani 2023)

Stable sorting is important because, while sorting, it preserves the original order of equal elements. This is crucial in applications where the sequence of data is meaningful. Algorithms like Merge Sort demonstrate stable sorting by making sure that elements with equal values retain their original positions in the final sorted array. (Thakrani 2023)

Performance Metrics

Performance metrics help us understand how well sorting algorithms work with different types of data and predict how algorithms will function in real-world applications. An algorithm's efficiency is determined by balancing its space and time complexities, which assess how quickly it completes tasks and how much memory it uses during the process. The most efficient algorithms perform operations quickly while using the least amount of space. This is crucial for managing large datasets, where speed and resource utilisation significantly impact system performance.

When selecting a sorting algorithm, practical considerations come into play; for instance, systems with limited memory require algorithms that have lower space complexities to avoid memory overflow and ensure smooth operation. On the other hand, applications using large volumes of data are suited to algorithms with lower time complexities to minimise processing time. Understanding the behaviour of algorithms through asymptotic notations is essential; the Best Case scenario (Big Omega, Ω(n)) represents optimal performance, usually seen when inputs are already nearly sorted. The Average Case (Big Theta, Θ(n)) provides a realistic view of algorithm performance under normal conditions. However, developers usually plan for the Worst Case (Big O Notation, O(n)), which outlines the maximum time an algorithm will take regardless of input condition, ensuring that the algorithm remains efficient under the most demanding circumstances. (Educative 2019)

Complexity Classes

Classification of Algorithms by Complexity

Complexity classes categorise algorithms according to their time and space demands as input sizes increase, offering a reference for selecting the algorithms for specific computational tasks.

- **Quadratic Complexity O(n^2):** Bubble, Insertion, and Selection Sort are in this category. They are characterised by their execution time increasing quadratically with the input data size. While these algorithms are straightforward to implement, they can become inefficient for large datasets as their performance significantly deteriorates with larger volumes of data.
- **Log-Linear Complexity O(n log n):** An example of this class is Merge Sort, which balances efficiency and simplicity for comparison-based sorting. This class is particularly beneficial for larger datasets, as it effectively breaks down the data into manageable, sorted, and merged

segments, resulting in much quicker processing times than algorithms with quadratic complexities.

- **Linear Complexity O(n):** This class can sort data in linear time and is represented by algorithms such as Counting Sort. This method is generally best suited for sorting integers and is highly efficient for datasets with a predetermined and limited range of input values. (Educative 2019)

In this report, we will explore these complexity classes in greater detail, demonstrating how each algorithm handles different data sizes.

Bubble Sort

Algorithm Explanation

Bubble Sort is a simple comparison-based sorting algorithm often used in educational settings due to its simplicity in demonstrating basic sorting principles. The algorithm repeatedly scans through the list of elements, comparing adjacent ones and swapping them if they are out of order. This continues until no further swaps are necessary, showing the list is sorted. Due to this method, it becomes inefficient for large datasets. The reason it's called "Bubble Sort" is that smaller elements slowly move ("bubble") up to the top of the list as it is sorted, like bubbles rising to the top of water. (Alake 2023)

Code & Explanation

Code

The understanding and development of the Bubble Sort code presented in this section were informed by insights from college notes (Carr, n.d., 4-8) and the detailed examples provided by (Geeks For Geeks 2024) and (Alake 2023).

```
public static void sort(int[] array) {
   boolean swapped; // Flag to track if a swap has occurred during the pass
   int length = array.length; // Initially set the entire array for sorting
   do {
       swapped = false; // Reset swap flag at the start of each pass
       for (int i = 0; i < length - 1; i++) { // Only iterate up to the
unsorted part of the array
           if (array[i] > array[i + 1]) { // Compare adjacent elements
               // Swap elements
               SortUtils.swap(array, i, i + 1);
               swapped = true; // Indicate a swap occurred
           }
       }
       length--; // Reduce the sorting range since the last element is now
sorted
   } while (swapped); // If no swaps occurred, the array is sorted
}
```

Detailed Explanation of the Code

- **Initialisation:** The method `sort` starts by initialising a boolean variable `swapped` to track if any swaps occur during each iteration. This ensures that the algorithm only continues when necessary. The `length` variable represents the

total number of elements for sorting. This is reduced for each pass to improve efficiency.

- **Outer Loop (do-while loop):** This loop ensures that the sorting continues only if a swap occurs in the previous iteration. If no swaps occur, it indicates that the array is sorted, and the loop exits.
- **Inner Loop (for loop):** Starts from the first to the last unsorted element of the array, which decreases with each pass as the largest elements find their positions.
- **Comparison & Swapping:** In the inner loop, adjacent elements that are out of order are compared and swapped using the SortUtils.swap method. The swapped flag is set to true whenever a swap occurs, indicating that another pass through the array is necessary.
- **Reducing the Sorting Range:** After each iteration, the `length` is reduced by one to exclude the sorted elements.
- **Termination:** The process terminates when no swaps are made during an iteration, as all elements are sorted.

Complexity Analysis

Time Complexity

- **Best Case:** O(n) - Sorted arrays terminate after one iteration. This allows Bubble Sort to stop early, saving processing time.
- **Average and WorstCase:** $O(n^2)$ is typical because each element must be compared with others multiple times using nested loops, with the worst case occurring with reverse sorted arrays requiring maximum swaps. (Geeks For Geeks 2024)

Space Complexity

- Bubble Sort is an in-place sorting algorithm with a space complexity of O(1). It uses a very small amount of extra storage space for swapping elements, and this space does not increase with the size of the input array. (Geeks For Geeks 2024)

Factors Affecting Running Time

The efficiency of Bubble Sort can vary based on the starting order of the elements. Bubble Sort is effective for arrays that are already partially sorted. Its optimisation is an early termination that stops the sorting process when no swaps are needed during a new iteration. This reduces unnecessary comparisons, making Bubble Sort more

effective at handling nearly sorted data, unlike more rigid algorithms like Selection Sort, which iterates through all elements regardless of order. (Geeks For Geeks 2024)

Advantages & Disadvantages

Advantages

- **Simplicity in Understanding and Implementation:** Bubble Sort is known for its simplicity, making it an excellent teaching tool in introductory computer science courses.
- **No Additional Memory Required:** Operating in place, Bubble Sort does not need extra memory beyond the initial array, making it a viable option for systems with limited memory.
- **Stability:** It maintains the original order of equal keys, which is important when the order of the data matters.

Disadvantages

- **Inefficient Time Complexity:** Bubble Sort is impractical for sorting large datasets because it has a time complexity of $O(n^2)$.
- **Limitations:** Its reliance on comparing elements for sorting is not as efficient compared to algorithms like MergeSort, which use techniques to minimise the number of comparisons needed (Geeks For Geeks 2024)

Conclusion

Bubble Sort remains an important algorithm in computer science education due to its simplicity and educational value. As an introductory tool, it offers a clear insight into sorting algorithm processes. While more efficient algorithms like MergeSort and QuickSort overshadow Bubble Sort when high performance is essential, Bubble Sort's relevance in education ensures its continued importance in computer science.

Sorting Student Number Using Bubble Sort

Selection Sort

Algorithm Explanation

Another straightforward sorting algorithm based on comparisons is Selection Sort, often used in educational settings because of its simplicity. Unlike Bubble Sort, which repeatedly swaps elements as it iterates through the list, Selection Sort enhances this by selecting the smallest element from the unsorted part of the array and moving it to the end of the sorted section. This minimises the number of swaps; however, similar to Bubble Sort, its performance decreases with larger datasets. The term "Selection Sort" is derived from continuously selecting the smallest and arranging it into the right position. (Sehgal 2018)

Code & Explanation

Code

The understanding and development of the Selection Sort code presented in this section were informed by insights from college notes (Carr, n.d., 14-18) and detailed examples from (Geeks For Geeks 2024).

```
public static void sort(int[] arr) {
    int length = arr.length;
    // Iterate over the array
    for (int i = 0; i < length - 1; i++) {
        // Assume the first element of the unsorted part as the minimum
        int minIndex = i;
        // Check the rest of the array to find the true minimum
        for (int j = i + 1; j < length; j++) {
            // If we find an element smaller than the current minimum, update
minIndex
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap the found minimum element with the first element of the unsorted
part
        // Only perform the swap if the minimum is not already in place
        if (minIndex != i) {
            SortUtils.swap(arr, minIndex, i);
        }
    }
}
```

Detailed Explanation of the Code

- **Initialisation:** The `sort` method begins by setting the array's length.
- **Outer Loop:** This loop iterates from the first element up to the second last element. This marks the beginning of the unsorted segment of the list where the minimum value needs to be identified.
- **Inner Loop:** Starting from the next element of the sorted segment, this loop searches through the remaining unsorted segment to find the smallest element and assigned to `minIndex`.
- **Swapping:** The minimum element discovered is swapped with the ith element of the unsorted segment, expanding the sorted portion of the array with each iteration.
- **Termination:** This process is repeated, expanding the sorted segment with each cycle of the outer loop until the entire array is sorted.

Complexity Analysis

Time Complexity

- **Best Case, Average, and Worst Case:** $O(n^2)$ consistently performs a fixed number of comparisons, as it must find the smallest item in the remaining unsorted section, regardless of the starting order of the elements. (Geeks For Geeks 2023).

Space Complexity

- *O(1)* Like Bubble Sort, it operates in-place and doesn't require extra memory beyond what's used for the input. (Geeks For Geeks 2023)

Factors Affecting Running Time

Selection Sort is quadratic complexity, $O(n^2)$, regardless of how the elements are initially ordered. This is because each iteration through the array involves finding the smallest element. Unlike Bubble Sort, which ends early if no swaps are necessary, Selection Sort completes a set amount of comparisons and swaps in every situation. This approach leads to a reliable but sometimes less efficient performance than other sorting techniques that minimise the total number of comparisons.

Advantages & Disadvantages

Advantages

- **Simplicity in Understanding and Implementation:** The straightforward approach of Selection Sort is an excellent introductory algorithm for computer science students.
- **No Additional Memory Required:** Since it sorts in place, Selection Sort doesn't need extra memory beyond the space for the original array.
- **Predictability:** Performs consistently regardless of the order of data.

Disadvantages

- **Inefficient Time Complexity:** Selection sort is ineffective for large datasets with a time complexity of $O(n^2)$.
- **Outperformed by Advanced Algorithms:** Algorithms like MergeSort, which utilise more advanced techniques for reducing sorting times, generally outperform Selection Sort in speed and efficiency. (Ghosh 2024)

Conclusion

Though it may not be the first choice for most applications due to its inability to scale with large datasets, like Bubble Sort, Selection Sort's role in teaching computer science fundamentals remains important. Its straightforward nature makes it a valuable educational tool.

Sorting Student Number Using Selection Sort

Insertion Sort

Algorithm Explanation

Like Bubble Sort and Selection Sort, it is often used to teach basic sorting principles as it is a simple algorithm. However, Insertion Sort is effective when applied to nearly sorted lists or small datasets, outperforming Bubble and Selection Sort in these scenarios. Insertion Sort creates a sorted array one item at a time, much like organising a deck of cards by hand. Each element from the unsorted section is placed into the right position in the sorted part. This approach involves shifting elements greater than the new element to create space for it to be inserted correctly. (Geeks For Geeks 2024)

Code & Explanation

Code

The understanding and development of the Insertion Sort code presented in this section were informed by insights from college notes (Carr, n.d., 9-13) and detailed examples from (Geeks For Geeks 2024)

```
public static void sort(int[] array) {
    for (int i = 1; i < array.length; i++) {
        int current = array[i]; // The current element to be positioned
        int j = i - 1; // The last index of the sorted section

        // Shift elements of the sorted section that are greater than the
current element to one position ahead of their current position
        while (j >= 0 && array[j] > current) {
            array[j + 1] = array[j];
            j--;
        }

        // Place the current element at its correct position in the sorted
section
        array[j + 1] = current;
    }
}
```

Detailed Explanation of the Code

- **Initialisation:** It begins with the second element because the first element is already considered sorted.

- **Outer Loop:** This loop runs from the second element to the last element of the array.
- **Current Element:** The current element to be sorted within the already sorted portion of the array.
- **Inner Loop:** Shifts elements larger than the `current` one position to the right, making space for the new element.
- **Insertion:** Positions the `current` element into the new spot.
- **Progression:** With each iteration, the sorted section expands while the unsorted segment shrinks until the algorithm completes.

Complexity Analysis

Time Complexity

- **Best Case:** O(n) occurs for sorted arrays, reducing the inner loop's activity, as no shifting is needed.
- **Average and Worst Case:** $O(n^2)$, like Bubbled sort, is often seen with small to moderately sized or partially sorted arrays. It becomes less efficient as the dataset grows.

Space Complexity

- *O(1)* - Like selection and Bubble Sort, Insertion Sort performs in-place, needing no additional space beyond the input array. (Geeks For Geeks 2024)

Factors Affecting Running Time

The initial state of the array influences Insertion Sort efficiency, similar to Bubble Sort. Arrays that are nearly sorted at the start generally need fewer adjustments, which can speed up the process considerably. However, if the array is in reverse order or contains large unsorted sections, the algorithm becomes less efficient, requiring more element shifts and comparisons to sort each item correctly. (Alake 2021)

Advantages & Disadvantages

Advantages

- **Simplicity in Understanding and Implementation:** One of the main advantages of Insertion Sort is its simple logic and incremental approach, making it an excellent tool for education.

- **Adaptive Performance:** Efficient for nearly sorted or small datasets, requiring fewer adjustments.
- **In-Place Sorting:** Insertion Sort requires minimal additional memory beyond the original array.
- **Stability:** Maintains the original order of equal elements.

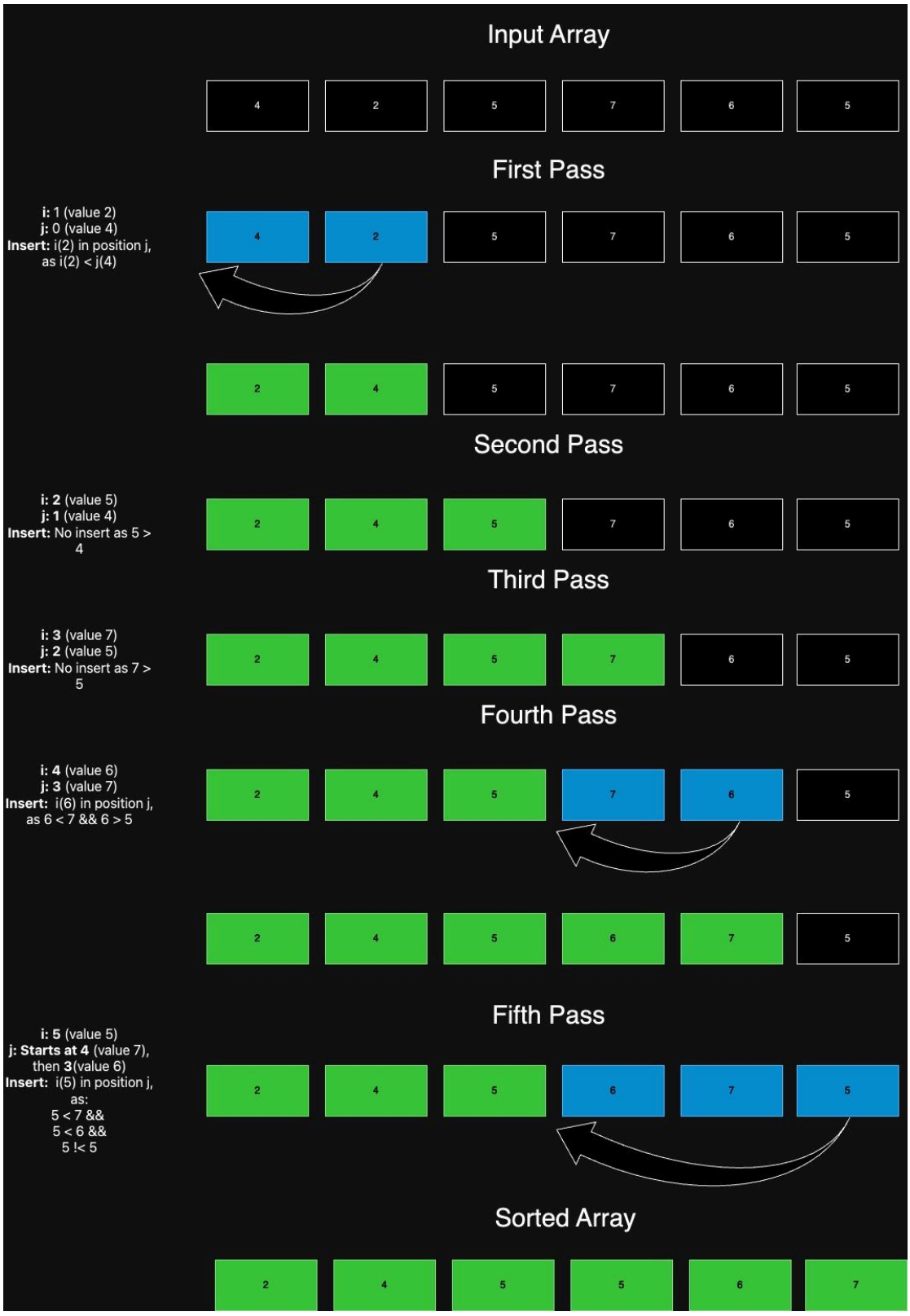
Disadvantages

- **Inefficient Time Complexity:** Although simple, the average and worst-case $O(n^2)$ scenarios make Insertion Sort slow for larger arrays.
- **Limited Application Scope**: Best suited for smaller datasets; performance drops significantly with larger or unsorted arrays. (DBMSpoly 2017)

Conclusion

Like Bubble and Selection Sort, Insertion Sort is an important part of computer science education due to its educational value. It stands out, as discussed, as it mimics how people often sort items manually, by comparing and placing each item into its correct position relative to already sorted items. While not always the best choice for large-scale applications, its role in helping new programmers understand the fundamentals of algorithmic sorting ensures its continued relevance in the academic world.

Sorting Student Number Using Insertion Sort



Merge Sort

Algorithm Explanation

Merge Sort stands out in both educational environments and real-world applications due to its efficient divide-and-conquer approach. Unlike simpler sorting methods such as Bubble or Selection Sort, it handles large datasets well. It breaks down a problem into smaller parts by continuously dividing the input array in half until each part has only one element. During the merging phase, these elements are combined in a manner that keeps them in a sorted order. (Geeks For Geeks 2024)

Code & Explanation

Code

The understanding and development of the Merge Sort code presented in this section were informed by insights from college notes (Carr, n.d., 4-5) and the detailed examples provided by (Geeks For Geeks 2024) and (W3Schools, n.d.).

```
public static void sort(int[] array) {
    if (array.length > 1) {
        performMergeSort(array, 0, array.length - 1);
    }
}

private static void performMergeSort(int[] array, int start, int end) {
    // Recursively divide the array if the current segment has more than one
element
    if (start < end) {
        int middle = (start + end) / 2; // Calculate the middle point of the
current segment

        // Recursively sort the left half of the array
        performMergeSort(array, start, middle);
        // Recursively sort the right half of the array
        performMergeSort(array, middle + 1, end);

        // Merge the sorted halves
        mergeSubArrays(array, start, middle, end);
    }
}
private static void mergeSubArrays(int[] array, int start, int middle, int end)
{
    // Initialise the sizes of temporary arrays to hold the subarrays
    int sizeLeft = middle - start + 1;
```

```
    int sizeRight = end - middle;

    // Create temporary arrays
    int[] leftArray = new int[sizeLeft];
    int[] rightArray = new int[sizeRight];

    // Copy the elements into the temporary arrays
    System.arraycopy(array, start, leftArray, 0, sizeLeft);
    System.arraycopy(array, middle + 1, rightArray, 0, sizeRight);

    // Merge the temporary arrays back into the original array
    int i = 0, j = 0; // Indices for left and right subarrays
    int k = start; // Index of merged subarray
    while (i < sizeLeft && j < sizeRight) {
        if (leftArray[i] <= rightArray[j]) {
            array[k] = leftArray[i++];
        } else {
            array[k] = rightArray[j++];
        }
        k++;
    }

    // Copy the remaining elements of leftArray, if any
    while (i < sizeLeft) {
        array[k++] = leftArray[i++];
    }

    // Copy the remaining elements of rightArray, if any
    while (j < sizeRight) {
        array[k++] = rightArray[j++];
    }
}
```

Detailed Explanation of the Code

- **Initialisation:** The `sort` method checks if the array length is greater than one, only recursive sorting only if necessary.
- **Recursive Division:** In the `performMergeSort` method, the array is repeatedly divided into halves until each segment has only one element.
- **Merging:** The `mergeSubArrays` method handles the merging of two pre-sorted halves of the array. This function is a key component of the merge sort's efficiency, as it arranges elements from two sorted subarrays, `array[start..middle]` and `array[middle+1..end]`, into a single sorted sequence. The merging process involves:

    - Comparing elements from the left and right temporary arrays using indices `i` (for the left array) and `j` (for the right array).
    - The smallest elements from these comparisons are placed back into the original array at the correct positions, starting from index `k`.
    - This step is critical to ensure the overall array is fully sorted.
- **Completing the Sort:** After the recursive division and merging, the entire array becomes sorted. The method ensures that all elements are correctly placed following the merge sort, leading to a sorted array. The recursion base case, where a subarray reduces to one element and the efficient merging of these elements parts, completes the sorting process.

Complexity Analysis

Time Complexity

- **Best, Average, and Worst Cases:** O(n log n). Owing to its divide-and-conquer approach, Merge Sort maintains its efficiency in best, average, and worst-case scenarios.

Space Complexity

- O(n). The merging process needs temporary arrays that are proportional to the original array. (Geeks For Geeks 2024)

Factors Affecting Running Time

Merge Sort maintains its O(n log n) running time across all scenarios. This attribute contrasts with algorithms like Bubble Sort, which can have quadratic running times under certain conditions. (Geeks For Geeks 2024)

Advantages & Disadvantages

Advantages:

- **Time Complexity:** Merge Sort consistently achieves an O(n log n) time complexity, marking it as one of the most efficient sorting algorithms.
- **Stability:** Maintains the order of equal elements.
- **Divide and Conquer Strategy:** Excellently handles large datasets by dividing them into smaller, more manageable subproblems, which are then conquered individually.

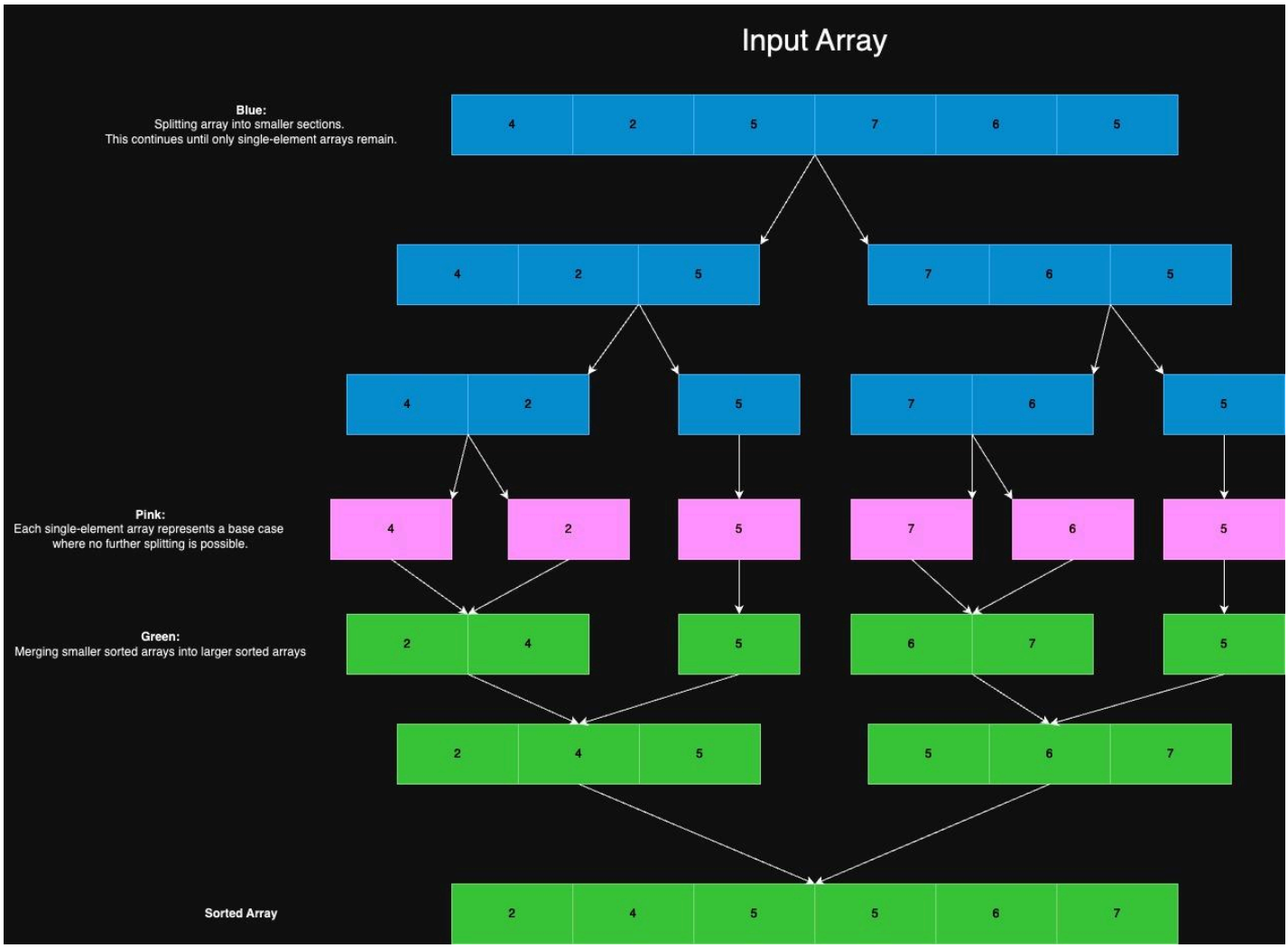
Disadvantages

- **Space Complexity:** Needs more space as the array size increases, which can be a disadvantage in environments where memory is limited.
- **Complex Implementation:** More difficult to implement than simpler sorting algorithms like Bubble Sort, particularly when dealing with in-place merging techniques. (Mondal 2024)

Conclusion

Merge Sort remains highly relevantas it ensures efficiency and reliability in sorting large datasets or maintaining the order of elements. Although it may not suit environments with strict memory constraints, its ability to handle complex sorting tasks efficiently makes it an indispensable tool in both academic and practical computer science applications.

Sorting Student Number Using Merge Sort



Counting Sort

Algorithm Explanation

Counting Sort is a highly efficient, non-comparison-based sorting algorithm, particularly good at sorting integers. It works best when the range of input values is not much larger than the number of elements. Unlike the previously discussed comparison-based methods like Bubble, Selection, etc, Counting Sort does not sort by comparing elements. Instead, it determines each element's position based on each value's count, which can be completed in linear time in optimal conditions. (Geeks For Geeks 2024)

Code & Explanation

Code

The understanding and development of the Counting Sort code presented in this section were informed by insights from college notes (Carr, n.d., 10-12) and the detailed examples provided by (Geeks For Geeks 2024).

```
public static void sort(int[] arr) {
    int max = findMax(arr); //Find the maximum value in the array
    int[] count = new int[max + 1]; // Initialise the count array
    // Count each element's frequency
    for (int num : arr) {
        count[num]++;
    }
    // Accumulate the counts
    for (int i = 1; i < count.length; i++) {
        count[i] += count[i - 1];
    }
    // Place elements into the sorted order
    int[] output = new int[arr.length];
    for (int i = arr.length - 1; i >= 0; i--) {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }
    // Copy sorted elements back to the original array
    System.arraycopy(output, 0, arr, 0, arr.length);
}

private static int findMax(int[] arr) {
    int max = arr[0]; // Assume the first element is the maximum
    for (int num : arr) {
        if (num > max) {
            max = num; // Update max if current element is greater
```

```
        }
    }
    return max; // Return the maximum value found
}
```

Detailed Explanation of the Code

- **Initialisation:** The `sort` method begins by using the `findMax` method to find the maximum value in the array, which determines the size of the `count` array. This array is used to keep a record of how often each unique element appears.
- **Counting Elements:** This process involves going through the input array and increasing the count for each element at its corresponding index in the `count` array.
- **Accumulating Counts:** The counts are then collected, changing the `count` array into a cumulative total that signifies each element's final sorted position.
- **Rebuilding the Array:** Elements are temporarily placed in an `output` array based on their positions from the cumulative counts, ensuring correct order.
- **Copying Back to the Original Array:** The data that has been sorted in the 'output' array is then moved back into the original array, updating it to show the sorted order.

Complexity Analysis

Time Complexity

- **Best, Average, and Worst Cases:** The algorithm runs in O(n+k) time, where `n` is the number of elements and `k` is the range of input values.

Space Complexity

- O(k) space due to the extra space needed for the count array. (Simplilearn 2023)

Factors Affecting Running Time

Unlike algorithms like Bubble Sort, which are affected by the initial order of elements, Counting Sort's performance mainly depends on the range of input values, specifically the difference between the maximum and minimum values. This allows it to operate efficiently by counting each element's occurrence, a process that remains unaffected by the elements' original order in the input array. (Study Smarter, n.d.)

Advantages & Disadvantages

Advantages

- **Linear Time Complexity**: The algorithm runs with a time complexity of O(n + k), where n is the number of elements and k is the range of input values.
- **Stable Sorting:** Maintains the relative order of equal elements
- **Not Based on Comparison:** Useful in situations where comparing elements is time-consuming.

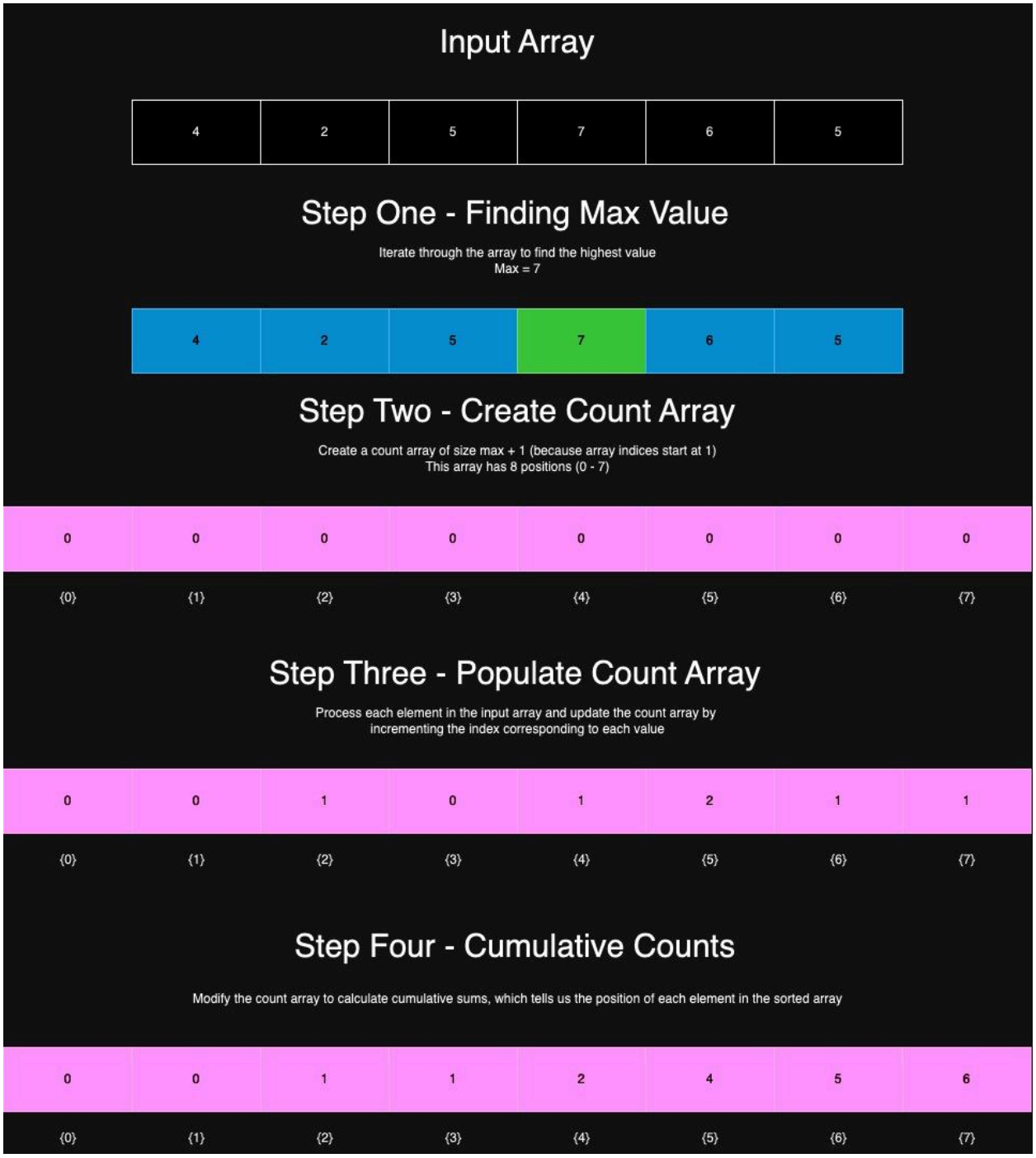
Disadvantages

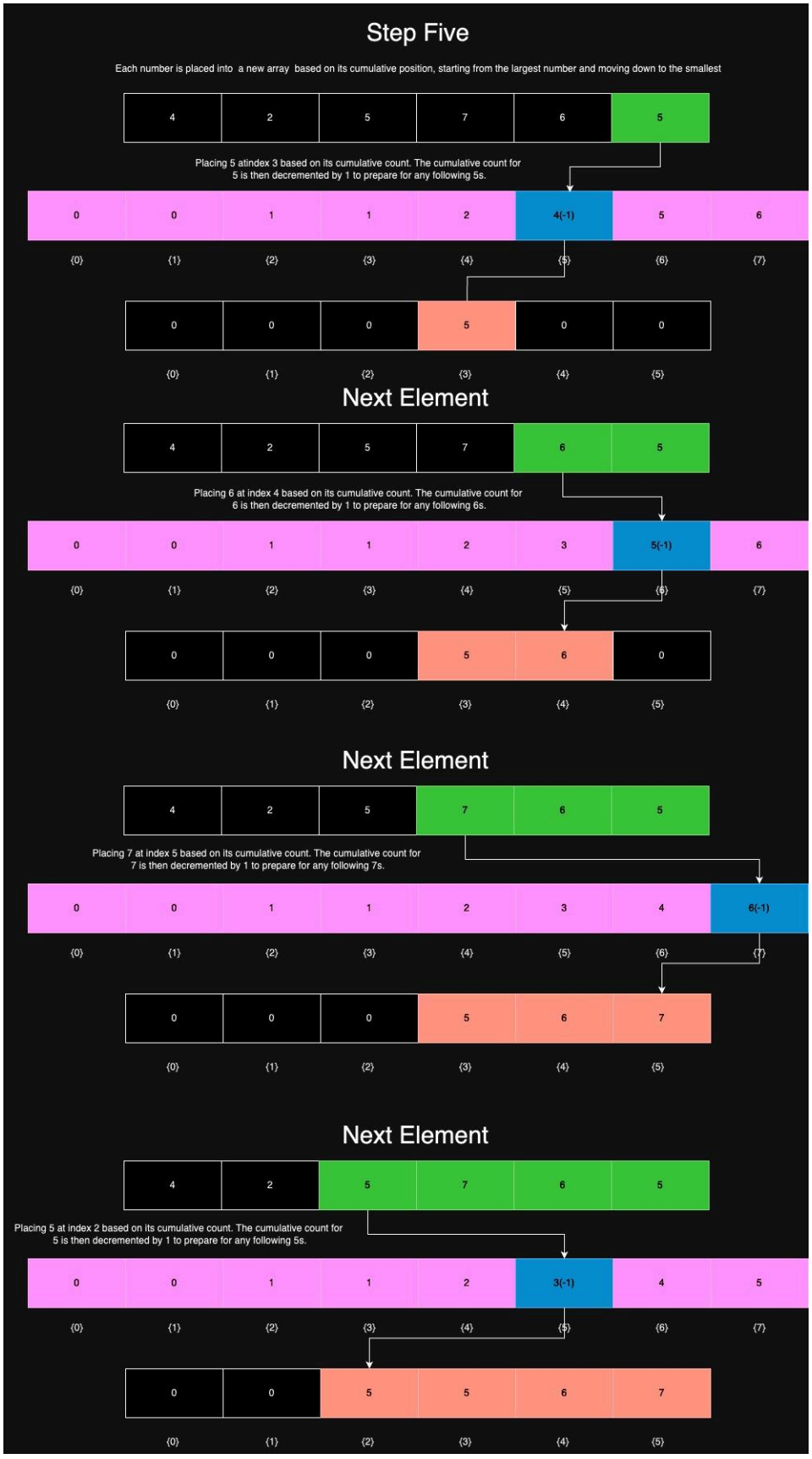
- **Space Complexity:** It may require large amounts of memory if the range of the input values is extensive.
- **Limited to Integers:** Effective for integers or scenarios where elements can be represented as integers. (Geeks For Geeks 2024)

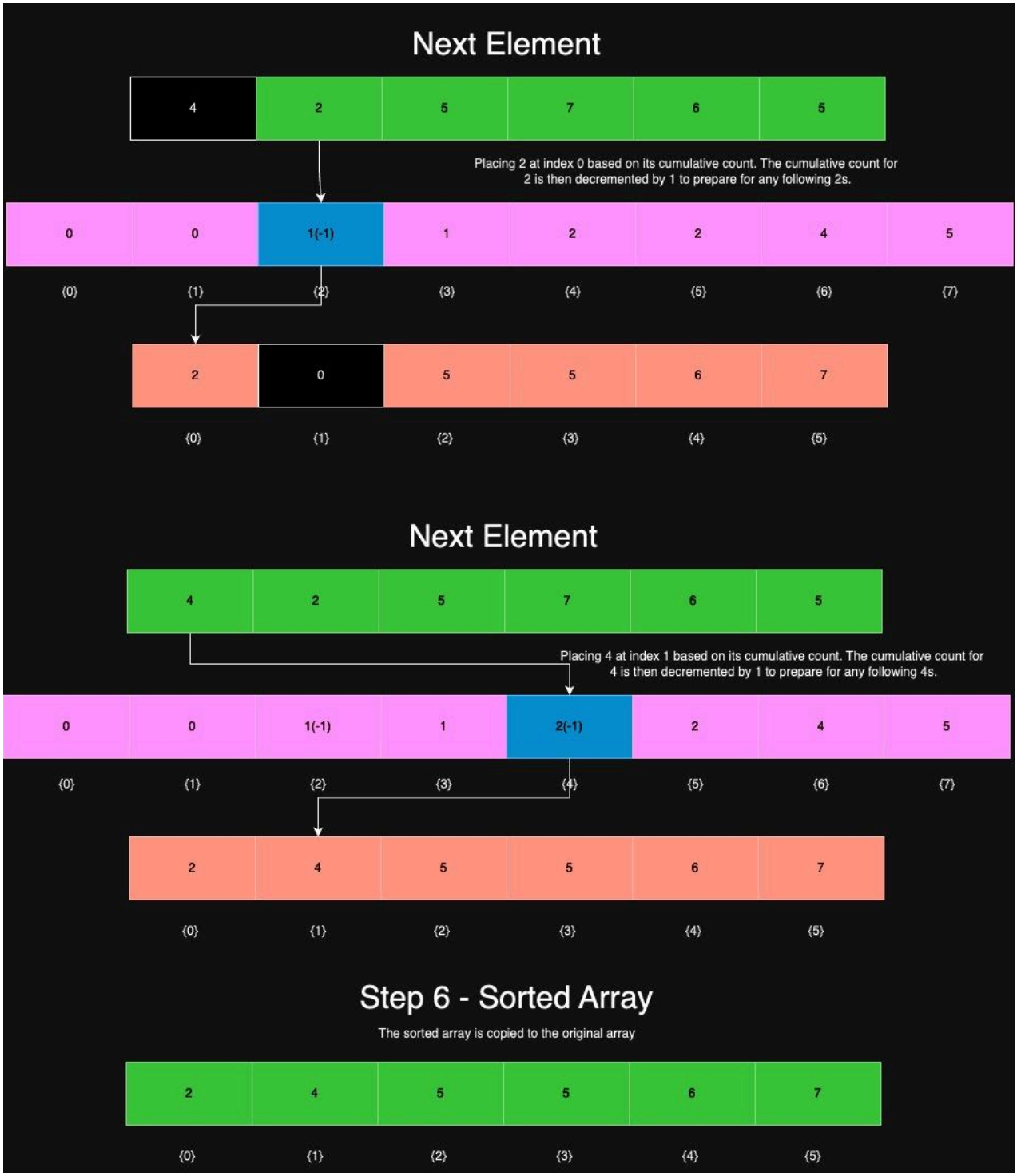
Conclusion

Counting Sort, with its unique approach to sorting by counting occurrences rather than performing direct element comparisons, offers advantages in speed and simplicity under the right conditions. It serves as an excellent educational tool for demonstrating linear-time sorting algorithms, providing valuable insights into efficient data handling techniques. While it faces limitations regarding applicability to data types and large data ranges, Counting Sort remains an excellent choice for specific applications where its constraints are met.

Sorting Student Number Using Counting Sort




Step Five
Each number is placed into a new array based on its cumulative position, starting from the largest number and moving down to the smallest
4 2 5 7 6 5
Placing 5 atindex 3 based on its cumulative count. The cumulative count for 5 is then decremented by 1 to prepare for any following 5s.
0 0 1 1 2 4(-1) 5 6
{0} {1} {2} {3} {4} {5} {6} {7}
0 0 0 5 0 0
{0} {1} {2} {3} {4} {5}
Next Element
4 2 5 7 6 5
Placing 6 at index 4 based on its cumulative count. The cumulative count for 6 is then decremented by 1 to prepare for any following 6s.
0 0 1 1 2 3 5(-1) 6
{0} {1} {2} {3} {4} {5} {6} {7}
0 0 0 5 6 0
{0} {1} {2} {3} {4} {5}
Next Element
4 2 5 7 6 5
Placing 7 at index 5 based on its cumulative count. The cumulative count for 7 is then decremented by 1 to prepare for any following 7s.
0 0 1 1 2 3 4 6(-1)
{0} {1} {2} {3} {4} {5} {6} {7}
0 0 0 5 6 7
{0} {1} {2} {3} {4} {5}
Next Element
4 2 5 7 6 5
Placing 5 at index 2 based on its cumulative count. The cumulative count for 5 is then decremented by 1 to prepare for any following 5s.
0 0 1 1 2 3(-1) 4 5
{0} {1} {2} {3} {4} {5} {6} {7}
0 0 5 5 6 7
{0} {1} {2} {3} {4} {5}


Next Element
4 2 5 7 6 5
Placing 2 at index 0 based on its cumulative count. The cumulative count for 2 is then decremented by 1 to prepare for any following 2s.
0 0 1(-1) 1 2 2 4 5
{0} {1} {2} {3} {4} {5} {6} {7}
2 0 5 5 6 7
{0} {1} {2} {3} {4} {5}
Next Element
4 2 5 7 6 5
Placing 4 at index 1 based on its cumulative count. The cumulative count for 4 is then decremented by 1 to prepare for any following 4s.
0 0 1(-1) 1 2(-1) 2 4 5
{0} {1} {2} {3} {4} {5} {6} {7}
2 4 5 5 6 7
{0} {1} {2} {3} {4} {5}
Step 6 - Sorted Array
The sorted array is copied to the original array
2 4 5 5 6 7
{0} {1} {2} {3} {4} {5}

Implementation & Benchmarking

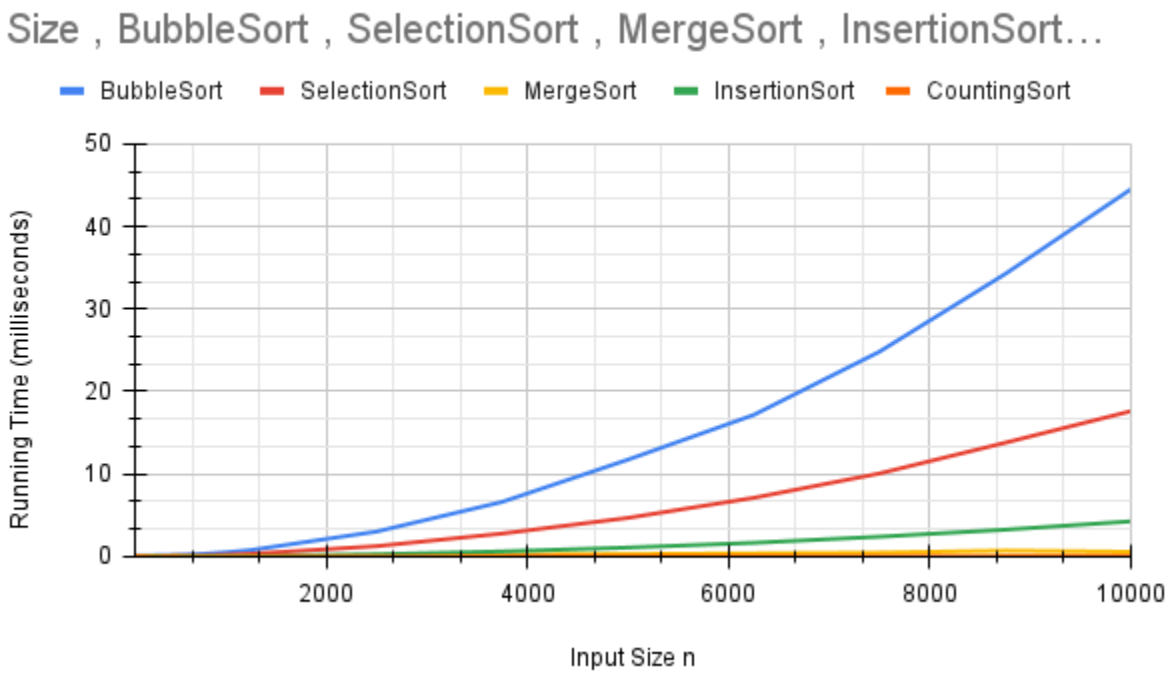
Benchmarking Methodology

- **Test Setup:** To guarantee the accuracy of the benchmark results, all non-essential applications were turned off during the testing period, except IntelliJ IDEA, which was used for running the tests. The benchmarking was performed on an Apple M2 Max with 64 GB of RAM and macOS Sonoma 14.0. This setup provided a high-performance computing environment that minimised external disturbances, ensuring consistency throughout the testing process.
- **Data Distribution:** The datasets used for benchmarking consisted entirely of random integers, which were selected to test the algorithms under standard conditions and avoid any bias towards specific data patterns. While this approach simplifies the initial testing phase, more benchmarks could incorporate reverse-ordered or partially ordered datasets to further demonstrate different real-world scenarios.
- **Repetition and Averaging:** Each sorting test was conducted ten times for every dataset size, with the execution times averaged. This helps ensure that the performance metrics reported are a reliable average.

Comprehensive Analysis and Conclusion of Benchmarking Results

Graph & Results in Milliseconds

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
BubbleSort	0.122	0.15	0.176	0.316	0.519	0.794	3.036	6.632	11.75	17.165	24.82	34.233	44.495
SelectionSort	0.059	0.094	0.13	0.131	0.217	0.339	1.272	2.799	4.694	7.125	10.071	13.797	17.62
MergeSort	0.039	0.013	0.028	0.038	0.045	0.06	0.144	0.241	0.323	0.395	0.488	0.733	0.588
InsertionSort	0.037	0.089	0.134	0.088	0.049	0.074	0.285	0.631	1.103	1.675	2.417	3.268	4.275
CountingSort	0.073	0.03	0.037	0.016	0.014	0.016	0.025	0.033	0.044	0.047	0.059	0.064	0.071



Analysis of Benchmark Results Across Various Sorting Algorithms:

- **BubbleSort:** Known for its reduced efficiency with increasing dataset sizes, the test results show BubbleSort's execution times went from 0.122 ms for a dataset of 100 elements to 44.495 ms for a dataset of 10,000 elements. This performance decline highlights its impracticality in handling large datasets due to its $O(n^2)$ complexity.
- **SelectionSort:** SelectionSort's performance starts faster than BubbleSort at 0.059 ms for 100 elements and escalates to 17.62 ms for 10,000 elements.

Although it scales slightly better than BubbleSort, its $O(n^2)$ time complexity still limits its practicality for larger datasets.

- **InsertionSort:** This algorithm performs well on smaller datasets, with an initial time of 0.037 ms for 100 elements, comparable to MergeSort and CountingSort. However, as the dataset size increases, its time increases to 4.275 ms for 10,000 elements, demonstrating its decreased efficiency with larger and less sorted datasets.
- **MergeSort:** Showing terrific efficiency and scalability, MergeSort starts at 0.039 ms for 100 elements and only increases to 0.588 ms for 10,000 elements. Its divide-and-conquer approach enables it to manage large datasets more effectively than simpler quadratic algorithms.
- **CountingSort:** CountingSort begins with 0.073 ms for 100 elements and remains efficient up to 0.071 ms for 10,000 elements, demonstrating its effectiveness under the right conditions. This consistent performance across dataset sizes demonstrates its effectiveness under ideal conditions.

Comparative Analysis:

- **MergeSort and CountingSort:** Both MergeSort and CountingSort show excellent performance as dataset sizes increase, demonstrating their effectiveness in handling larger volumes of data. However, there are differences between them:
    - MergeSort's divide-and-conquer approach minimises computational overhead and consistently improves performance, making it highly adaptable and effective for a broad range of applications. It shows only a small increase in execution times, from 0.039 ms for 100 elements to 0.588 ms for 10,000 elements.
    - CountingSort also maintains low execution times (from 0.073 ms for 100 elements to 0.071 ms for 10,000 elements) but requires specific conditions to run. This limitation means that while CountingSort is efficient under the right circumstances, its application is more specialised than MergeSort, which does not depend on the nature of the dataset's value range.
- **BubbleSort and SelectionSort:**
    - BubbleSort performance decreases with increasing dataset sizes, with execution times escalating from 0.122 ms for 100 elements to 44.495 ms for 10,000 elements. This increase demonstrates the challenges of algorithms with an $O(n^2)$ time complexity in scaling with larger datasets, making BubbleSort impractical for such applications.
    - Although SelectionSort performs slightly better than Bubble Sort, it faces similar scalability issues. Its execution times rise from 0.059 ms for 100 elements to 17.62 ms for 10,000 elements. Although it is more efficient

than BubbleSort, SelectionSort still performs poorly with large datasets, again demonstrating the issues with $O(n^2)$ time complexity.

- **InsertionSort:** This algorithm performs well in smaller and nearly sorted datasets. Its times range from 0.037 ms to 4.275 ms as dataset sizes increase, highlighting its suitability for applications where data is frequently added in a nearly sorted order.

Conclusion

Our benchmarking study thoroughly compares various sorting algorithms across different dataset sizes, confirming theoretical performance expectations with practical results. MergeSort and CountingSort are the top performers, showing excellent scalability and efficiency with large datasets and adhering to their O(n log n) and O(n+k) complexities. Their ability to swiftly handle complex, data-intensive tasks makes them ideal for applications requiring rapid data processing.

On the other hand, Bubble and Selection Sort show significant performance declines with larger datasets due to their $O(n^2)$ complexity, making them less suitable for large-scale applications but still valuable as educational tools for basic sorting principles. InsertionSort, also simple in design, performs well with small or nearly sorted datasets but declines as size and disorder increase.

The practical results from our tests closely match what we expected based on each algorithm's properties. For instance, MergeSort is excellent for managing large databases due to its effective merging processes, while CountingSort is ideally suited for environments with limited and predictable data ranges, where its speed and efficiency are most beneficial.

In conclusion, choosing a sorting algorithm should depend on specific computational requirements, considering the application's data size, complexity, and performance needs. This approach not only effectively uses computational resources but also boosts overall system efficiency and performance. Our analysis offers clear guidance for developers in choosing the most suitable algorithms to optimise their operations and achieve better performance outcomes.

Bibliography

Alake, Richmond. 2021. “Insertion Sort Explained–A Data Scientists Algorithm Guide | NVIDIA Technical Blog.” NVIDIA Developer. https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/.

Alake, Richmond. 2022. “Merge Sort Explained: A Data Scientist's Algorithm Guide | NVIDIA Technical Blog.” NVIDIA Developer. https://developer.nvidia.com/blog/merge-sort-explained-a-data-scientists-algorithm-guide/.

Alake, Richmond. 2023. “Bubble Sort Time Complexity and Algorithm Explained.” Built In. https://builtin.com/data-science/bubble-sort-time-complexity.

Astrachan, Owen. n.d. “Bubble Sort: An Archaeological Algorithmic Analysis.” Duke Computer Science. Accessed April 25, 2024. https://users.cs.duke.edu/~ola/bubble/bubble.html.

Broussard, Ashley Broussard. 2017. “Understanding Selection Sort: A Simple Sorting Algorithm.” linkedin.com. https://www.linkedin.com/pulse/understanding-selection-sort-simple-sorting-algorithm-broussard-9vwne/.

Carr, Dominic. n.d. *Computational Thinking with Algorithms - Simple Sorts*.

Carr, Dominic. n.d. “Sorting Algorithms.” Accessed April 26, 2024.

DBMSpoly. 2017. “Advantage & Disadvantages of Sort.” YouTube: Home. https://dbmspoly.blogspot.com/p/advantage-disadvantages-of-sort.html.

Educative. 2019. “Big O Notation: A primer for beginning devs.” Educative.io. https://www.educative.io/blog/a-big-o-primer-for-beginning-devs.

Eldridge, Stephen. 2024. “Sorting algorithm | Definition, Time Complexity, & Facts.” Britannica. https://www.britannica.com/technology/sorting-algorithm.

Entrustech Inc. 2017. “Mastering the Merge Sort: The Power of Efficient Sorting.” YouTube: Home. https://medium.com/@entrustech/mastering-the-merge-sort-the-power-of-efficient-sorting-5c572acb6f3c.

Geeks For Geeks. 2023. “Time and Space complexity analysis of Selection Sort.” GeeksforGeeks. https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-selection-sort/.

Geeks For Geeks. 2024. “Counting Sort - Data Structures and Algorithms Tutorials.” GeeksforGeeks. https://www.geeksforgeeks.org/counting-sort/.

Geeks For Geeks. 2024. “Time and Space Complexity Analysis of Bubble Sort.” GeeksforGeeks. https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/.

Geeks For Geeks. 2024. “Insertion Sort - Data Structure and Algorithm Tutorials.” GeeksforGeeks. https://www.geeksforgeeks.org/insertion-sort/.

Geeks For Geeks. 2024. “Merge Sort - Data Structure and Algorithms Tutorials.” GeeksforGeeks. https://www.geeksforgeeks.org/merge-sort/.

Geeks For Geeks. 2024. “Merge Sort - Data Structure and Algorithms Tutorials.” GeeksforGeeks. https://www.geeksforgeeks.org/merge-sort/.

Geeks For Geeks. 2024. “Bubble Sort - Data Structure and Algorithm Tutorials.” GeeksforGeeks. https://www.geeksforgeeks.org/bubble-sort/.

Geeks For Geeks. 2024. “Selection Sort – Data Structure and Algorithm Tutorials.” GeeksforGeeks. https://www.geeksforgeeks.org/selection-sort/.

Ghosh, Amit Kumar. 2024. “Selection Sort in Data Structures: An Overview.” ScholarHat. https://www.scholarhat.com/tutorial/datastructures/selection-sort-in-data-structures.

Great Learning Team. 2024. “Understanding Time Complexity with Examples - 2024.” Great Learning. https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/.

Halli, Omkareshwar Halli Omkareshwar. 2017. “Bubble Sort in C++ (code with explanation).” YouTube: Home. https://medium.com/@omkareshwarhalli/bubble-sort-in-c-code-with-explanation-ca8b01f2f3be.

Mondal, Akash. 2024. “Mastering Merge Sort: Algorithm, Implementation, Advantages.” Intellipaat. https://intellipaat.com/blog/merge-sort-algorithm/.

Paul, Javin. 2021. “Difference between Comparison (QuickSort) and Non-Comparison (Counting Sort) based Sorting Algorithms? Example.” Javarevisited. https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html.

Rodreguaze, Sophia. 2023. “Intro to Insertion Sort Algorithm With Code Examples.” HackerNoon. https://hackernoon.com/intro-to-insertion-sort-algorithm-with-code-examples.

Sandoval, Andrés. 2023. “Optimizing Database Performance: Exploring Indexing Techniques in DBMS.” DEV Community. https://dev.to/abaron10/optimizing-database-performance-exploring-indexing-techniques-in-dbms-1emj.

Sehgal, Karuna. 2018. “An Introduction to Selection Sort | by Karuna Sehgal | Karuna Sehgal.” Medium. https://medium.com/karuna-sehgal/an-introduction-to-selection-sort-f27ae31317dc.

Simplilearn. 2023. “Counting Sort Algorithm: Overview, Time Complexity & More.” Simplilearn.com. https://www.simplilearn.com/tutorials/data-structure-tutorial/counting-sort-algorithm.

Srivastava, Alankrit. 2024. “Binary Insertion Sort.” Naukri.com. https://www.naukri.com/code360/library/what-is-binary-insertion-sort.

Study Smarter. n.d. “Bubble Sort: Algorithm, Example, Time Complexity & Advantages.” StudySmarter. Accessed April 25, 2024. https://www.studysmarter.co.uk/explanations/computer-science/algorithms-in-computer-science/bubble-sort/.

Study Smarter. n.d. “Counting Sort: Algorithm, Time Complexity.” StudySmarter. Accessed April 27, 2024.

https://www.studysmarter.co.uk/explanations/computer-science/algorithms-in-computer-science/counting-sort/.

Study Smater. n.d. “Merge Sort: Algorithm, Advantages, Examples & Time Complexity.” StudySmarter. Accessed April 27, 2024. https://www.studysmarter.co.uk/explanations/computer-science/algorithms-in-computer-science/merge-sort/.

Study Tonight. n.d. “Merge Sort Algorithm.” Studytonight. Accessed April 27, 2024. https://www.studytonight.com/data-structures/merge-sort.

Thakrani, Suhailt. 2023. “What are stable sorting algorithms and in-place sorting algorithms?” Medium. https://medium.com/@suhailthakrani12/what-are-stable-sorting-algorithms-and-in-place-sorting-algorithms-672820a8e36c.

Topper World. 2017. “Insertion Sort.” YouTube: Home. https://topperworld.in/insertion-sort/.

Vishwakarma, Deepak. 2017. “Counting Sort Algorithm.” YouTube: Home. https://www.codinginterviewpro.com/data-structure-and-algorithms-tutorial/counting-sort-algorithm/#section-14.