University of Science - VNUHCM

Faculty of Information Technology

PPOJECT REPORT

Data Structure Algorithm

Lab 3 - Sorting Algorithm

29th July 2023

Class 22CLC06

Group 01

Le Duy Anh - 22127012

Huynh Cao Tuan Kiet - 22127219

Ly Dinh Minh Man - 22127255

Vo Nguyen Phuong Quynh - 22127360

Instructors

Mr. Bui Huy Thong

Mrs. Tran Thi Thao Nhi

Contents

1 Introduction

1.1 Topic

This project aims to study 11 algorithms (Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort) on arrays with different data order and size, implement them using the C++ programming language, and provide an evaluation of their complexity and flexibility.

1.2 Objectives

The objectives of the project are as follows:

- Introduce the sorting algorithms.
- Analyze and evaluate the complexity of the sorting algorithms.
- Implement the algorithms using 5 given commands.
- Create visualizations of the algorithms using diagrams.

1.3 Research Methodology

Our research methodology involves the following steps:

- Present algorithm ideas.
- Study the algorithms.
- Analyze the complexity of the algorithms.
- Implement the algorithms.
- Debug and troubleshoot errors.
- Create graphs to illustrate the algorithms.

1.4 Achievements

The achieved results of our project are:

- Completed implementation of the 11 sorting algorithms.
- Successfully wrote programs using the 5 given commands.
- Accomplished analysis and evaluation of the complexity of the algorithms.
- Completed visualizations of the sorting algorithms.

1.5 Instructors

We cannot complete this project without the excellent help of:

- Mr. Le Thanh Tung (Theory Lecturer)
- Mr. Bui Huy Thong
- Mrs. Tran Thi Thao Nhi

We aim to express our sincerest gratitude for assigning us the sorting analysis project. Your guidance and support throughout this endeavor have been invaluable, and we are truly grateful for the opportunity to learn and grow under your tutelage. Thank you for being exceptional lecturers and for making this learning experience both enjoyable and rewarding. Your impact on our academic journey will stay with us for a lifetime.

1.6 Machine Configuration

We have implemented and evaluated the algorithms using Visual Studio Code (VS Code) on a computer system with the following specifications:

Processor	AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx 2.30 GHz
Installed RAM	8.00 GB (7.43 GB usable)
Device ID	83C8E803-2A45-49FF-9C6B-07AE939DCA03
Product ID	00327-30000-00000-AAOEM
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display

Figure 1: Machine Configuration

The program was compiled using the **GNU C++ Compiler (g++) version 12.1.0** with flag `-std=c++14`. This flag ensures that the code is compiled according to the C++14 language standard.

2 Algorithm Presentation

2.1 Presentation Note

For ease of presentation, here are some conventions in our reporting:

1. A: input array
2. n: size of the input array A
3. Elements in red : sorted elements
4. Elements in blue : target elements
5. [...] : subarray of the main array A
6. Max: maximum element of array A
7. Min: minimum element of array A

2.2 Selection sort

Selection sort is a simple comparison-based sorting algorithm that works by repeatedly finding the minimum (or maximum) element from the unsorted part of the list and swapping it with the first element of the unsorted part.

2.2.1 Idea

The selection sort algorithm works based on the idea of dividing the list into two parts: the sorted part and the unsorted part. The algorithm repeatedly finds the minimum element from the unsorted part and moves it to the end of the sorted part. This process is iteratively applied until the entire list is sorted.

2.2.2 Step-by-step description

1. Start with the entire list of elements to be sorted.
2. Find the minimum (or maximum) element in the unsorted part.
3. Once the end of the unsorted part is reached, swap the found minimum (or maximum) element with the first element of the unsorted part of the list.
4. The first element of the unsorted part is now considered sorted.
5. Repeat steps 2 to 4, with the new unsorted part.
6. Continue this process until the entire list is sorted.

Example

Stage	Array A	Description
0	38, 27, 43, 3, 9, 82, 10	Initial unsorted array
1	38, 27, 43, 3, 9, 82, 10	found 3 as the minimum element in unsorted part
2	3, 27, 43, 38, 9, 82, 10	swap 3 with the first element of unsorted part (which is the first element)
3	3, 27, 43, 38, 9, 82, 10	now [3] is considered as sorted
...	Then continue with the rest of array	

2.2.3 Complexity evaluations

- Best-case time complexity: O(n^2)
- Average-case time complexity: O(n^2)
- Worst-case time complexity: O(n^2)
- Space complexity: O(1) (it requires constant extra space regardless of the input size)

2.2.4 Variants & improvements

Double Selection Sort: Double selection sort is a variation of the basic selection sort algorithm that simultaneously finds both the minimum and maximum elements in each iteration and swaps them with the first unsorted element and the last unsorted element. This reduces the number of iterations and swaps, making it more efficient than the basic selection sort.

2.3 Insertion sort

Insertion Sort is a simple comparison-based sorting algorithm that builds the final sorted array one element at a time by repeatedly inserting elements into their correct positions.

2.3.1 Idea

The algorithm maintains a sorted subarray and iterates through the remaining unsorted elements to insert each them into its correct position in the sorted subarray.

2.3.2 Step-by-step description

1. Begin with the second element by comparing it to the previous element.
2. If the previous element is greater, swap the two elements.
3. Continue comparing the element with the previous elements and swapping them if necessary until it is in its correct position.
4. Repeat steps 1-3 for the remaining unsorted elements until the entire array is sorted.

Example

Stage	Array A	Description
0	38, 27, 43, 3, 9, 82, 10	Initial unsorted array
1	27, 38, 43, 3, 9, 82, 10	Inserted 27 at the correct position
2	27, 38, 43, 3, 9, 82, 10	43 is in the right position
3	3, 27, 38, 43, 9, 82, 10	Inserted 3 at the correct position
4	3, 9, 27, 38, 43, 82, 10	Inserted 9 at the correct position
5	3, 9, 27, 38, 43, 82, 10	82 is in the right position
6	3, 9, 10, 27, 38, 43, 82	Inserted 10 at the correct position

2.3.3 Complexity evaluations

- Best-case time complexity: O(n), which occurs when the input array is already sorted.
- Average-case time complexity: O(n^2)
- Worst-case time complexity: O(n^2), which occurs when the input array is sorted in reverse order.
- Space complexity: O(1)

2.3.4 Variants & improvements

We can consider using a binary search to find the correct position for inserting element, reducing the number of comparisons and improving the overall performance.

2.4 Bubble sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the entire array is sorted. The algorithm gets its name because smaller elements "bubble" to the top of the array as the sorting progresses.

2.4.1 Idea

In each pass through the list, the largest (or smallest, depending on the sorting order) element "bubbles up" to its correct position at the end of the list. The algorithm then focuses on the remaining unsorted portion of the list and repeats the process until the entire list is sorted.

2.4.2 Step-by-step description

1. Compare the first and second elements.
2. If they are in the wrong order (according to the desired sorting order, e.g., ascending or descending), swap them.
3. Move to the next pair of elements (second and third) and continue comparing and swapping if necessary.

4. Repeat this process until the last pair of elements is compared and swapped if needed.
5. Now, the largest element is in its correct position at the end of the list.
6. Repeat the same process for the remaining unsorted portion of the list until the entire list is sorted.

Example

Stage	Array A	Description
0	4, 3, 2, 5, 1	Initial unsorted array
1	3, 4, 2, 5, 1	Compare 4 and 3, swap is needed as 4 is greater than 3.
2	3, 2, 4, 5, 1	Compare 4 and 2, swap is needed as 4 is greater than 2.
3	3, 2, 4, 5, 1	Compare 4 and 5, no swap is needed as 4 is less than 5.
4	3, 2, 4, 1, 5	Compare 5 and 1, swap is needed as 5 is greater than 1.
5	4, 3, 2, 1, 5	The largest element, 5, has "bubbled up" to the last position of the array after the first pass.
...	Then continue with the rest of array	

2.4.3 Complexity evaluations

- Best-case time complexity: O(n^2)
- Average-case time complexity: O(n^2)
- Worst-case time complexity: O(n^2)
- Space complexity: O(1) (it requires constant extra space regardless of the input size)

2.4.4 Variants & improvements

- **Improved Bubble Sort**: We can keep track of whether any swaps were made during a pass. If no swaps were made, it means the array is already sorted, and we can terminate the sorting early. This helps avoid unnecessary iterations and reduces the time complexity for partially sorted arrays.
- Shaker sort
- **Odd-Even Sort**: Odd-Even Sort is a variation of Bubble Sort that works by comparing and swapping all odd-indexed pairs followed by comparing and swapping all even-indexed pairs. It repeatedly alternates between odd and even passes until the array is fully sorted. This approach can be parallelized since the odd and even passes are independent of each other, which can improve performance in certain scenarios.

2.5 Shaker sort

Shaker Sort, also known as Cocktail Sort or Bidirectional Bubble Sort, is a variation of the Bubble Sort algorithm. It is a simple comparison-based sorting algorithm that repeatedly steps through the list, comparing adjacent elements and swapping them if they are in the wrong order.

2.5.1 Idea

Same as bubble sort, but instead of using 1 loop from left to right, now we'll have a second loop from right to left for swapping the element to its correct position.

2.5.2 Step-by-step description

1. Start with the first element (index 0) and set the left and right boundaries of the unsorted portion of the list.
2. Repeat the following steps until the left boundary is no longer less than the right boundary:
   - Move from the left boundary to the right boundary, comparing adjacent elements, and swapping them if the left element is greater than the right element.
   - Decrement the right boundary as the largest element in the unsorted portion is now correctly placed at the end.
   - Move from the right boundary to the left boundary, comparing adjacent elements, and swapping them if the left element is greater than the right element.
   - Increment the left boundary as the smallest element in the unsorted portion is now correctly placed at the beginning.
3. Once the left boundary is no longer less than the right boundary, the list is sorted.

Example

Stage	Array A	Description
1	38, 27, 43, 3, 9	Initial unsorted array
2	27, 38, 3, 9, 43	loop from left to right in unsorted part (swap 27 with 38, swap 3 with 43, swap 9 with 43) and now last element is considered sorted
3	3, 27, 38, 9, 43	loop from right to left in unsorted part (swap 3 with 38 then swap 3 with 27) and now first element is considered sorted
...	Then continue with the rest of array	

2.5.3 Complexity evaluations

- Best-case time complexity: O(n)
- Average-case time complexity: O(n^2)
- Worst-case time complexity: O(n^2)
- Space complexity: O(1) (The algorithm requires a constant amount of additional memory for temporary variables, making it memory-efficient.)

2.6 Shell sort

Shell Sort is an efficient variation of the Insertion Sort algorithm that improves its performance by sorting elements that are far apart first and gradually reducing the gap between elements to perform smaller and more efficient insertions. This sorting algorithm is also known as "diminishing increment sort."

2.6.1 Idea

The main idea behind Shell Sort is to improve the performance of the Insertion Sort algorithm by sorting elements that are far apart first and then gradually reducing the gap between elements until the entire array is sorted. The fundamental concept is to minimize the number of comparisons and swaps required to sort the array efficiently.

2.6.2 Step-by-step description

1. **Choose an initial gap (increment) value**: The algorithm starts by selecting a gap value that determines how far apart elements will be compared during the first pass. This gap can be chosen based on some predefined sequence or heuristics, like the Knuth sequence or Sedgewick sequence. The goal is to select a gap that efficiently reduces the number of inversions in the array.

2. **Sort elements with the chosen gap**: The algorithm performs a series of insertion sorts on the array, but with a specific gap. It compares elements that are separated by the gap and swaps them if necessary to partially sort the array.

3. **Reduce the gap**: After completing the first pass, the algorithm reduces the gap value, typically by dividing it by a chosen factor (commonly 2). This reduces the gap between elements to focus on smaller subsets of the array.

4. **Repeat steps 2 and 3**: The algorithm continues to sort the array with the updated gap value and reduces the gap until it becomes 1. The final pass of the algorithm with a gap of 1 is a standard insertion sort, but the array is now preprocessed and partially sorted due to the previous passes.

5. **Complete the sort**: Finally, the algorithm performs a regular insertion sort with a gap of 1 to fully sort the array. At this point, the array is nearly sorted, and the standard insertion sort has reduced the remaining inversions efficiently.

Example

For the sake of simplicity, let's choose gap = 2

Stage	Array A	Description
0	4, 3, 2, 5, 1	Initial unsorted array
1	2, 3, 4, 5, 1	Comparing 4 and 2 (gap = 2), swap is needed.
2	2, 3, 4, 5, 1	Comparing 3 and 5, swap is not needed.
3	2, 3, 1, 5, 4	Comparing 4 and 1, swap is needed.
4	2, 3, 1, 5, 4	No more pairs, we continue with gap = gap / 2 = 1
...	Then continue with the rest of array	

2.6.3 Complexity evaluations

The time complexity of Shell Sort depends on the chosen gap sequence. Let's consider the worst-case and average-case time complexities for two popular gap sequences:

- The original Shell Sort: Uses a gap sequence of (n/2), (n/4), (n/8), ..., 1, where N is the number of elements in the array. In the worst-case scenario, this sequence can lead to a time complexity of approximately O(n^2).
- Knuth's sequence: Defined as $(3^k - 1)/2$, where k is the number of passes starting from 0 until the value exceeds n ⇒ Average-case time complexity: O($n^{\frac{3}{2}}$)
- **Ciura's sequence**: gaps = [1, 4, 10, 23, 57, 132, 301, 701, 1750]
- Best-case time complexity: O(n)
- Space complexity: O(1) (it requires little extra space for the gap sequence)

2.7 Heap sort

Heap Sort is a comparison-based sorting algorithm that utilizes a binary heap data structure to efficiently sort elements in an array. It has an average and worst-case time complexity of O(n log n), making it a relatively efficient sorting algorithm. Heap Sort works by building a max-heap or min-heap from the array elements and repeatedly extracting the root element to create a sorted array.

2.7.1 Idea

The algorithm first builds a max heap, repeatedly swapping the root element with the last element of the heap, reducing the heap size, and "heapifying" to maintain the max heap property. This process continues until the entire array is sorted.

2.7.2 Step-by-step description

1. **Build the Heap:** The first step of Heap Sort is to build a heap from the input array. The heap is a binary tree that satisfies the heap property, which is that for every node, its value must be greater (for max-heap) or smaller (for min-heap) than or equal to the values of its children. The heap can be built in place within the input array.
2. **Max-Heapify (or Min-Heapify):** To build a heap, we need to ensure the heap property is maintained for each subtree rooted at an index i. The Max-Heapify (or Min-Heapify) algorithm is used to achieve this. It compares the parent node with its children and swaps them if necessary to satisfy the heap property. The process continues recursively until the entire heap is built.
3. **Heapify the Array:** Starting from the last non-leaf node (index n/2 - 1) to the root (index 0), apply Max-Heapify (or Min-Heapify) to each node to convert the array into a valid heap.
4. **Extract Elements:** After building the heap, the largest (for max-heap) or smallest (for min-heap) element will be at the root of the heap (index 0). Swap this element with the last element in the array (index n-1).

5. **Heap Size Reduction:** Decrease the heap size by one (heap size = heap size - 1) to exclude the last element, which is now in its correct sorted position.

6. **Maintain Heap Property:** To maintain the heap property, apply Max-Heapify (or Min-Heapify) to the root element (index 0) again.

7. **Repeat Extraction:** Repeat steps 4 to 6 until the heap size is reduced to 1. After each iteration, the largest (for max-heap) or smallest (for min-heap) element will be placed at the end of the array.

Example

Build Heap

Stage	Array A	Description
0	38, 27, 43, 3, 5	Initial unsorted array
1	38, 27, 43, 3, 5	Heapify at index = 1 (already satisfy the heap at this node)
2	43, 27, 38, 3, 5	Heapify at index = 0 (swap 43 with 38)

Sort the array

Stage	Array A	Description
1	43, 27, 38, 3, 5	Max-heap array
2	5, 27, 38, 3, 43	swap first element (maximum element) with last element of unsorted part. And now last element is considered as sorted
3	38, 27, B, 3, 43	Heapify the first element (swap 38 with 5)
4	3, 27, B, 38, 43	swap first element (maximum element) with last element of unsorted part. And now last elements are considered as sorted
...	Then continue with the rest of array	

Note: Heap Sort is an in-place sorting algorithm. However, it is not a stable sorting algorithm, as equal elements may change their relative order during the sorting process.

2.7.3 Complexity evaluations

- Best-case time complexity: $O(n \log n)$
- Average-case time complexity: $O(n \log n)$
- Worst-case time complexity: $O(n \log n)$
- Space complexity: $O(1)$ (it requires constant extra space regardless of the input size)

2.7.4 Variants & improvements

Optimized Heapify: The heapify step in Heap Sort can be further optimized to reduce the number of comparisons and swaps. Techniques like sift-down (instead of sift-up) or using bitwise operations for index calculations can improve the performance of heapify operations.
Choosing Optimal Data Structures: Depending on the size and nature of the input data, you can consider using different heap data structures like binary heaps or Fibonacci heaps. For specific scenarios, Fibonacci heaps can provide better performance for certain heap operations.

2.8 Merge sort

Merge Sort is a comparison-based sorting algorithm that apply divide-and-conquer strategy. It dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

2.8.1 Idea

The algorithm divides the array into two smaller subarrays until each subarray contains only one element and it then merges these subarrays until the entire array is sorted.

2.8.2 Step-by-step description

1. Divide the input array into two smaller subarrays.
2. Recursively apply Merge Sort to each half, dividing them further until they contain only one element.
3. Merge the sorted subarrays by comparing the elements and placing them in the correct order.
4. Repeat the merging process until the entire array is sorted.

Example

Stage	Array A	Description
0	[38, 27, 43, 3, 9, 82, 10]	Initial unsorted array
1	[38, 27, 43, 3] & [9, 82, 10]	Divide the array into two smaller subarrays
2	[38, 27] & [43, 3] & [9, 82] & [10]	The two subarray are divided into smaller subarrays
3	[38] & [27] & [43] & [3] & [9] & [82] & [10]	Divided those subarray till its size reach 1
4	[27, 38], [3, 43], [9, 82], and [10]	The single-element sub-arrays are merged back in sorted order
5	[3, 27, 38, 43] & [9, 10, 82]	Continue to merge those sub-arrays in sorted order
6	[3, 9, 10, 27, 38, 43, 82]	Array after Merge Sort

2.8.3 Complexity evaluations

- Best-case time complexity: O($nlogn$)
- Average-case time complexity: O($nlogn$)
- Worst-case time complexity: O($nlogn$)
- Space complexity: O(n) for additional memory during the merging process.

2.8.4 Variants & improvements

One of the weakness of Merge Sort is that it requires additional memory to store subarray. To overcome this, we can consider using an iterative version of MergeSort, such as Bottom-Up MergeSort, which avoids recursion and reduces memory consumption.

2.9 Quick sort

Quick Sort is a widely used sorting algorithm known for its efficiency and performance. It follows the divide-and-conquer strategy to sort an array by selecting a pivot element and partitioning the array into two sub-arrays: one with elements less than the pivot and the other with elements greater than the pivot. It then recursively sorts the sub-arrays until the entire array is sorted.

2.9.1 Idea

The main idea behind Quick Sort is to efficiently sort an array by employing the divide-and-conquer approach. It accomplishes this by selecting a pivot element from the array and partitioning the array into two sub-arrays: one with elements less than the pivot and the other with elements greater than the pivot. It then recursively sorts the two sub-arrays until the entire array is sorted.

2.9.2 Step-by-step description

1. **Pivot selection**: Choose a pivot element from the array. The choice of the pivot can affect the efficiency of the algorithm. A common approach is to pick the last element in the array as the pivot, but other strategies, like choosing a random element or using the median of three elements, can also be used to improve performance.

2. **Partitioning**: Rearrange the elements in the array such that all elements less than the pivot are on the left side, and all elements greater than the pivot are on the right side. The pivot element is now in its final sorted position. This step is crucial for the effectiveness of the Quick Sort algorithm.

3. **Recursion**: Recursively apply Quick Sort to the two sub-arrays created in the partitioning step. One sub-array contains elements less than the pivot, and the other contains elements greater than the pivot.

4. **Combine**: After the recursion, the individual sub-arrays are sorted. Combine the sorted sub-arrays, and the entire array is now sorted.

Example

For the sake of simplicity, **pivot = last element** (bold)

Stage	Array A	Description
0	4, 3, 2, 5, 1	Initial unsorted array
1	4, 3, 2, 5, **1**	Choose 1 as the pivot
2	[ ] **1** [4, 3, 2, 5]	Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot
3	1 [4, 3, 2, 5]	1 now is sorted. Recursively apply the Quick Sort algorithm to the two sub-arrays
...	Then continue with the rest of array	

2.9.3 Complexity evaluations

The time complexity of Quick Sort depends on several factors, including the choice of the pivot element, the input data distribution, and whether the algorithm is implemented with optimizations.

- Best-case time complexity: O($n \log n$)
- Average-case time complexity: O($n \log n$)
- Worst-case time complexity: O(n^2)
- Space complexity: O($n \log n$)

2.10 Counting sort

Counting Sort is an integer sorting algorithm that works efficiently when the range of input values is not significantly greater than the number of elements to be sorted. It is a non-comparison-based sorting algorithm

2.10.1 Idea

The idea behind Counting Sort is to exploit the knowledge of the input range and the frequency of elements in the array to achieve a linear-time sorting algorithm, rather than relying on comparisons between elements. It is particularly useful when the range of input values is relatively small compared to the number of elements to be sorted.

2.10.2 Step-by-step description

1. **Find the Range:** Determine the range of input values in the array (k) by finding the minimum and maximum elements. This step helps in determining the size of the counting array.
2. **Count the Occurrences:** Create a counting array of size k+1 (to account for 0-based indexing) and initialize it with zeros. Count the occurrences of each element in the input array and store the counts in the counting array. Each index in the counting array represents a distinct value from the input range, and the value at that index represents the number of occurrences of that element in the input array.
3. **Calculate Cumulative Counts:** Modify the counting array to hold the cumulative counts. The value at each index in the cumulative counting array represents the position in the sorted array where elements with that value should be placed.
4. **Build the Sorted Array:** Create a new array to store the sorted elements. Iterate through the original array, use the cumulative counting array to determine the correct positions of elements in the sorted array, and place them accordingly. Decrement the cumulative count after placing an element to handle duplicate elements correctly.
5. **Final Sorted Array:** The sorted array is now constructed and represents the elements in their correct order.

Example

Array A: 4, 2, 1, 4, 3, 2

- First, we need to find the range of input values (k). In this case, the smallest element in the array is 1, and the largest element is 4, so k = 4 - 1 + 1 = 4.

- Create a counting array to keep track of the occurrences of each value in the input array. The counting array's size will be equal to the range of input values (k).
- **Counting array** (initialized with zeros): 0, 0, 0, 0 (blue)
- Counting array after counting occurrences: 1, 2, 1, 2
- Calculate the cumulative counts in the counting array. This step will help us determine the correct positions of each element in the sorted array.
- Cumulative counting array: 1, 3, 4, 6
- Create a sorted array that will store the elements in their correct order.
- **Sorted array** (initialized with zeros): 0, 0, 0, 0, 0, 0
- Now, iterate through the original array from right to left (to maintain stability) and use the cumulative counting array to place elements in their correct positions in the sorted array:

Current index	**Sorted Array**	**Explaination**
5	0, 0, 2, 0, 0, 0	Element 5 at index 2 (cumulative count for value 2 is 3) Update cumulative count: 1, 2, 4, 6
4	0, 0, 2, 3, 0, 0	Element 4 at index 3 (cumulative count for value 3 is 4) Update cumulative count: 1, 2, 3, 6
3	0, 0, 2, 3, 0, 4	Element 3 at index 5 (cumulative count for value 4 is 6) Update cumulative count: 1, 2, 3, 5
2	1, 0, 2, 3, 0, 4	Element 2 at index 0 (cumulative count for value 1 is 0) Update cumulative count: 0, 2, 3, 5
1	1, 2, 2, 3, 0, 4	Element 1 at index 1 (cumulative count for value 2 is 2) Update cumulative count: 0, 1, 3, 5
0	1, 2, 2, 3, 4, 4	Element 0 at index 4 (cumulative count for value 4 is 5) Update cumulative count: 0, 1, 3, 4

2.10.3 Complexity evaluations

- Time complexity for all cases: O($n + k$), where k is the range of input values.
- Space complexity: O($n + k$) for two arrays: sorted array and counting array

(k = (Max - Min) + 1)

2.11 Radix sort

Radix Sort is a non-comparative sorting algorithm that sorts numbers base on its digits. It sorts the numbers from the least significant digit to the most significant one.

2.11.1 Idea

The algorithm iterates the numbers digit by digit, sorting them based on each value. It repeats this process multiple times for each digit position until the entire array is sorted.

2.11.2 Step-by-step description

1. Determine the maximum number of digits in the array.
2. Starting from the least significant digit (rightmost), sort the numbers based on that digit using a stable sorting algorithm (counting sort or bucket sort).
3. Repeat step 2 for each subsequent digit position, moving towards the most significant digit.
4. After iterate all the digits, the array will be sorted.

Example

Stage	Array A	Description
0	170, 45, 75, 90, 802, 24, 2, 66	Initial unsorted array
1 (1's)	170, 90, 802, 2, 24, 45, 75, 66	Sorted based on 1's digit
2 (10's)	802, 02, 24, 45, 66, 170, 75, 90	Sorted based on 10's digit
3 (100's)	002, 024, 045, 066, 075, 090, 170, 802	Sorted based on 100's digit

2.11.3 Complexity evaluations

- Best-case time complexity: O($n * d$)
- Average-case time complexity: O($n * d$)
- Worst-case time complexity: O($n * d$)
- Space complexity: O(n) (additional memory to store the digits)

(d is the number of maximum digits and n the is number of elements)

2.11.4 Variants & improvements

Radix Sort with Counting Sort: Radix sort requires a sorting algorithm to sort those digits in the sorting order. One common improvement to this is to use counting sort, especially when the range of digits is small. Combining counting sort with radix sort can improve its performance.

2.12 Flash sort

Flash Sort is a non-comparative sorting algorithm that works based on the distribution of elements into equally spaced buckets. It is particularly effective when the input data is uniformly distributed.

2.12.1 Idea

The algorithm divides the input array into a fixed number of buckets, and it determines the range of values that each bucket should hold. It then distributes the elements into these buckets according to their values. After that, it applies a simple sorting algorithm (often insertion sort) to each bucket to sort the elements within it.

2.12.2 Step-by-step description

1. Determine the number of buckets based on the size of the input array and the desired performance. (base on experimentations, the number of buckets should equal to 0.45 * the size of the array)
2. Find the minimum and maximum values in the input array.
3. Calculate the range (difference between the maximum and minimum values) and the width of each bucket.
4. Create an array of empty buckets.
5. Iterate through the input array and distribute each element into the appropriate bucket based on its value.
6. Sort each non-empty bucket using a simple sorting algorithm like insertion sort.
7. Concatenate the sorted buckets to obtain the final sorted array.

Example

Stage	Array A	Description
0	-1, 5, -6, 2, 13, 7, 19	Initial unsorted array
1	buckets = $0.45 * 7 \approx 3$	Find the number of buckets
2	Bucket size = (Max - Min) / buckets = $25/3 \approx 8.33 = 9$ Bucket 1: [-6, 3] Bucket 2: [4, 13] Bucket 3: [14, 23]	Determine the range of elements and buckets
3	Bucket 1: [-1, -6, 2] Bucket 2: [5, 13, 7] Bucket 3: [19]	Distribute elements into the buckets
4	Bucket 1: [-6, -1, 2] Bucket 2: [5, 7, 13] Bucket 3: [19]	Sorting the bucket
5	-6, -1, 2, 5, 7, 13, 19	Concatenate the buckets

2.12.3 Complexity evaluations

- Best-case: $O(n + k)$, which occurs when the input data is uniformly distributed.
- Average-case: $O(n + k)$
- Worst-case: $O(n^2)$, which can happen when the input data is skew and requires many redistributions.
- Space complexity: $O(n + k)$

*(k is the number of buckets. For the best efficiency, we use k = 0.45 * n)*

2.12.4 Variants & improvements

Instead of using fixed number of buckets, we can improve the flash sort by dynamically adjusting the number of buckets based on the data distribution, which might lead to better results.

3 Experimental results and Comments

3.1 Data Tables

Note: The time unit used for the running time in our statistical tables is milliseconds.

3.1.1 Sorted data

Data order: **Sorted**						
Data size	10,000		10,000		50,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	307.25	100010001	3372.23	900030001	7927.42	2500050001
Insertion sort	0.1	29998	0.46	89998	0.5	149998
Bubble sort	358.05	100010001	3143.79	900030001	8623.72	2500050001
Shaker sort	0.07	20002	0.2	60002	0.33	100002
Shell sort	0.79	236321	2.54	716321	4.17	1196321
Heap sort	5.74	655331	19.04	2191650	33.33	3850353
Merge sort	5.92	485241	18.53	1589913	30.88	2772825
Quick sort	2.09	10007	11.26	30007	17.13	50007
Counting sort	1.95	310007	2.29	330007	2.56	350007
Radix sort	2.05	140056	7.71	510070	12.81	850070
Flash sort	1.2	127995	3.65	383995	6.02	639995
Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	21414.37	10000100001	265966.29	90000300001	724744.64	250000500001
Insertion sort	0.97	299998	2.92	899998	6.78	1499998
Bubble sort	34452.28	10000100001	280477.72	90000300001	728030.66	250000500001
Shaker sort	0.73	200002	1.68	600002	3.42	1000002
Shell sort	8.33	2396321	11.41	7196321	51.92	11996321
Heap sort	63.69	8215082	93.1	26963232	444.56	46654888
Merge sort	63.04	5845657	76.51	18945945	363.96	32517849
Quick sort	44.37	100007	118.97	300007	195.96	500007
Counting sort	3.45	400007	2.57	600007	13.72	1000012
Radix sort	26.25	1700070	35.32	6000084	189.57	10000084
Flash sort	12.07	1279995	12.36	3839995	65.95	6399995

3.1.2 Nearly sorted data

Data order: **Nearly sorted**						
Data size	10,000		10,000		50,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	303.59	100010001	3060.25	900030001	7846.93	2500050001
Insertion sort	0.67	190742	3.95	536858	1.02	299446
Bubble sort	348.91	100010001	3269.22	900030001	3092.02	2500050001
Shaker sort	1.82	175373	6.68	602302	8.56	800111
Shell sort	1.36	375805	4.4	1226797	5.46	1860861
Heap sort	5.73	654558	19.32	2191641	32.87	3849668
Merge sort	6.24	516399	18.56	1676738	25.65	2857789
Quick sort	2.81	10010	9.61	30020	16.03	50036
Counting sort	1.98	310007	2.31	330007	2.12	350007
Radix sort	2.02	140056	7.55	510070	10.59	850070
Flash sort	1.21	127971	3.75	383965	4.79	639967
Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	24066.44	10000100001	264832.32	90000300001	637539.48	250000500001
Insertion sort	0.81	678154	8.61	1250370	12.24	1878546
Bubble sort	31409.16	10000100001	306661.85	90000300001	660060.56	250000500001
Shaker sort	5.09	562701	6.57	1129064	10.44	1361312
Shell sort	9.83	2724033	10.99	7542693	57.74	12417285
Heap sort	74.26	8214581	202.73	26963350	229.98	46655123
Merge sort	66.35	5938235	71.89	19043767	369.91	32611899
Quick sort	41.8	100068	139.17	600004	187.85	1000004
Counting sort	3.41	400007	2.41	600007	20.95	1000012
Radix sort	26.75	1700070	37.78	6000084	192.02	10000084
Flash sort	12.49	1279961	12.96	3839963	77.17	6399965

3.1.3 Reverse sorted data

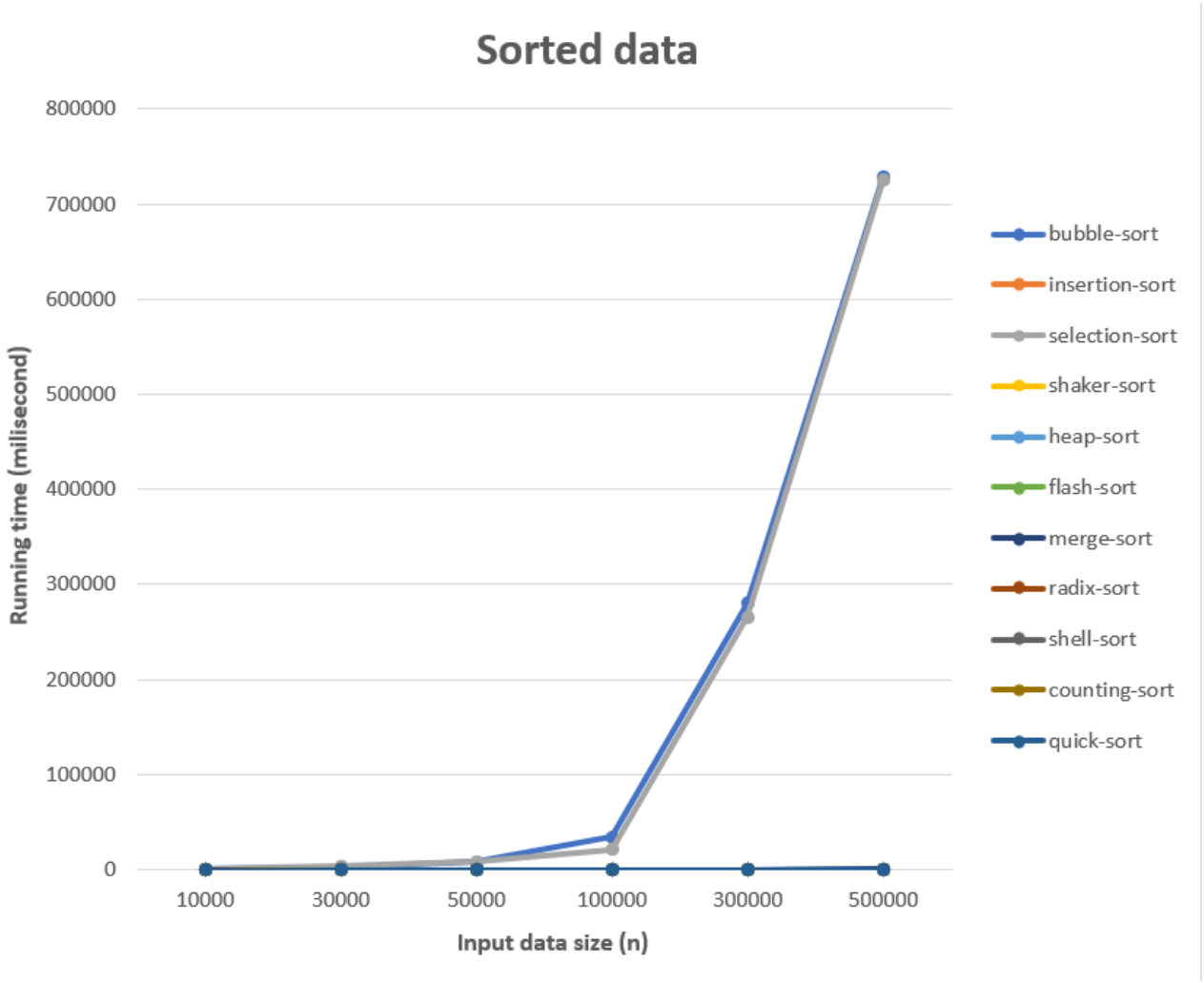
Data order: **Reverse sorted**						
Data size	10,000		10,000		50,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	339.85	100010001	2983.13	900030001	8132.33	2500050001
Insertion sort	707.87	100009999	3352.89	900029999	10016.98	2500049999
Bubble sort	607.81	100010001	5391.96	900030001	14934.35	2500050001
Shaker sort	1404.43	100005001	10916.32	900015001	28030.41	2500025001
Shell sort	486.11	100216322	3582.51	900656322	10089.9	2501096322
Heap sort	6.72	591773	18.22	2018326	32.08	3537726
Merge sort	6.95	486440	18.57	1603464	30.67	2783944
Quick sort	2.46	10010	10.66	30010	19.92	50010
Counting sort	1.95	310007	0.83	330007	2.6	350007
Radix sort	2.02	140056	7.6	510070	12.92	850070
Flash sort	1.18	110504	3.54	331504	5.79	552504
Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	32894.5	10000100001	289996.8	90000300001	785846.98	250000500001
Insertion sort	37798.7	10000099999	388519.42	90000299999	946032.32	250000499999
Bubble sort	75865.74	10000100001	511133.78	90000300001	1512940.23	250000500001
Shaker sort	124927.42	10000050001	1235109.51	90000150001	3236206.65	250000500001
Shell sort	39294.51	10002196322	440329.15	90006596322	981844.14	250010996322
Heap sort	71.52	7568945	229.91	25119381	421.78	43733350
Merge sort	64.27	5867896	198.81	19008312	363.32	32836408
Quick sort	38.14	100010	115.39	300010	194.3	500010
Counting sort	1.35	400007	3.93	600007	14.08	1000012
Radix sort	26.43	1700070	96.5	6000084	172.49	10000084
Flash sort	11.9	1105004	34.79	3315004	76.8	5525004

3.1.4 Randomize data

Data order: **Randomize**						
Data size	10,000		10,000		50,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	308.64	100010001	2940.82	900030001	7637.4	2500050001
Insertion sort	191.42	50524205	1813.83	450335661	5618.87	1256925961
Bubble sort	630.27	100010001	6462.74	900030001	18459.06	2500050001
Shaker sort	890.83	67122272	10154.57	597495417	24413.47	1667567899
Shell sort	208.03	50740804	2121.07	452053448	1754.48	1250767800
Heap sort	7.11	623040	25.62	2106122	15.79	3696636
Merge sort	7.75	593710	27.55	1967325	14.77	3432880
Quick sort	4.03	10016	8.49	30024	17.22	50042
Counting sort	2.21	310007	2.89	330007	1.18	350007
Radix sort	2.01	140056	7.46	510070	6.83	850070
Flash sort	1.24	95945	5.04	309898	2.43	499255
Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection sort	30929.05	10000100001	244555.11	90000300001	482789.3	250000100001
Insertion sort	18786.86	5010850376	165560.66	45003338332	381760.32	12485100000
Bubble sort	76975.76	10000100001	603457.7	90000300001	1290625.71	250000100001
Shaker sort	90635.62	6655583602	768385.03	59941076969	1529633.96	166670000000
Shell sort	19045.19	5005620872	172438.48	44996221769	391820.46	12516500000
Heap sort	88.94	7895793	275.61	26036872	256.03	45218638
Merge sort	85.76	7265608	313.06	23683252	205.46	40882951
Quick sort	35.78	100078	128.13	600004	179.89	1000004
Counting sort	4.62	400007	8.09	600007	24.03	800007
Radix sort	24.97	1700070	80.11	5100070	142.6	8500070
Flash sort	13.31	937028	46.66	2699690	56.11	4497770

3.2 Line graphs

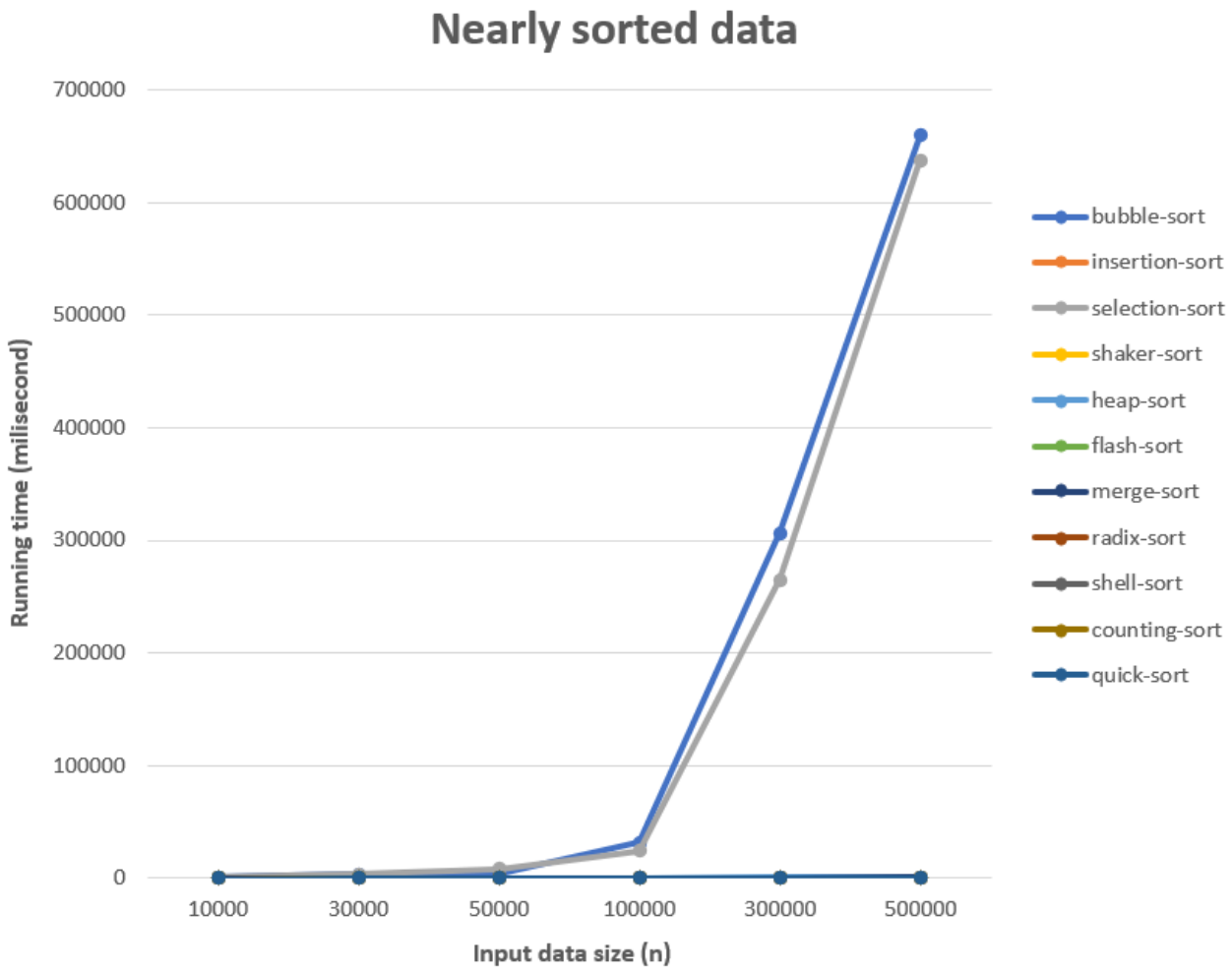
3.2.1 Sorted data



Comment

- Fastest Algorithm: Shaker Sort. The sorted input type represents the best-case scenario for each algorithm. With Shaker Sort, the best-case complexity is O(n). Therefore, only one pass is needed to sort the array.
- Slowest Algorithms: Selection Sort and Bubble Sort. Both of these algorithms have a complexity of O(n^2). Their sorting process continues to the last element even if the array is already sorted, which increases the execution time.

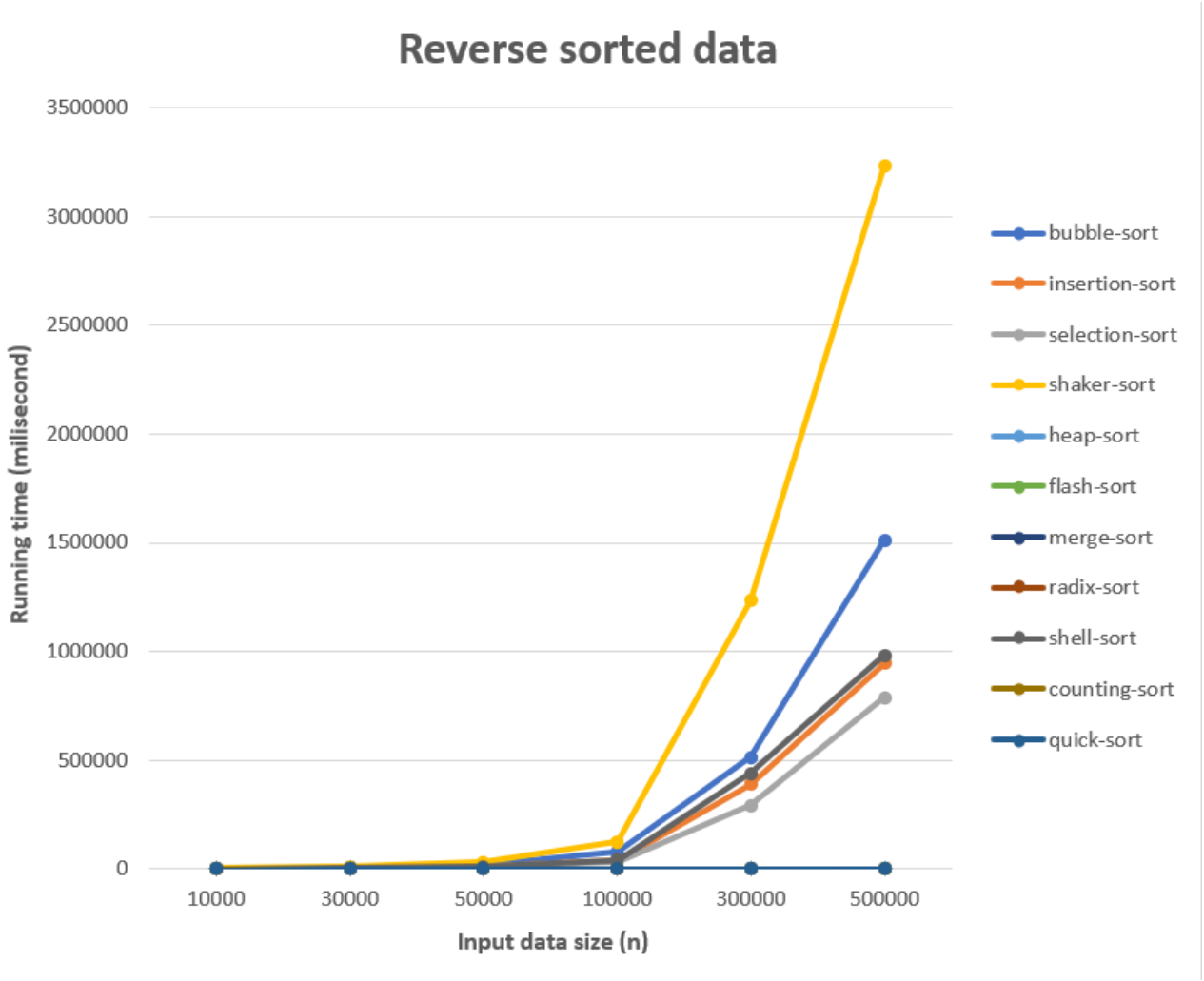
3.2.2 Nearly sorted data



Comment

- Fastest Algorithm: Counting Sort, because its complexity is O($n+k$) with $k = (Max - Min) + 1$, and interestingly, in this input, k is always smaller than n, resulting in a complexity of O(n). Therefore, Counting Sort runs extremely fast.
- Slowest Algorithm: Bubble Sort, because this algorithm has a complexity of O(n^2). Its sorting process continues to the last element even if the array is already sorted, which increases the execution time.

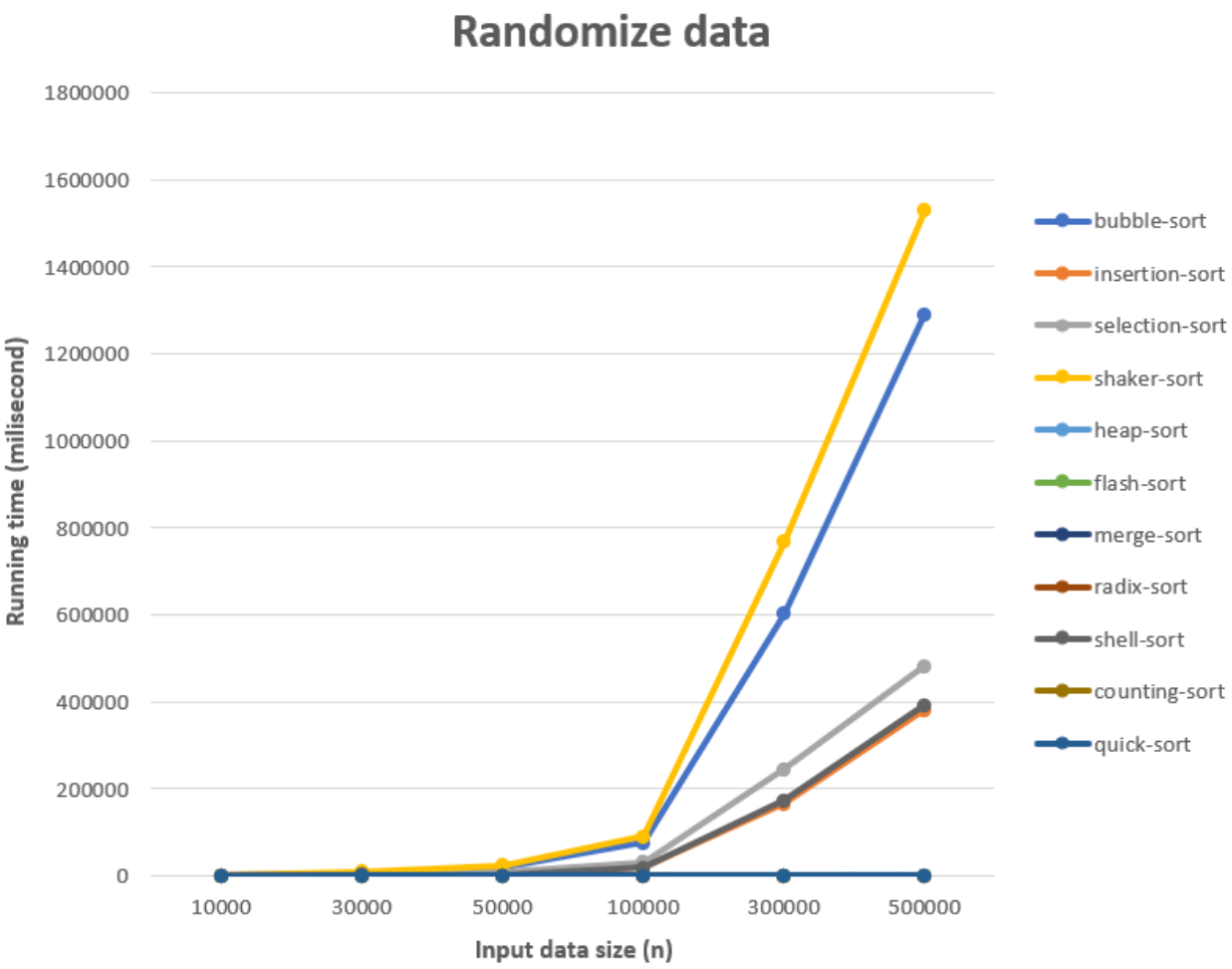
3.2.3 Reverse sorted data



Comment

- Fastest Algorithm: Counting Sort, because its complexity is $O(n+k)$ with $k = (Max - Min) + 1$, and interestingly, in this input, k is always smaller than n, resulting in a complexity of O(n). Therefore, Counting Sort runs extremely fast.
- Slowest Algorithm: Shaker Sort, because in this input, it falls into the worst-case scenario for Shaker Sort. The complexity of Shaker Sort in the worst-case scenario is O(n^2). Therefore, this algorithm runs very slowly.

3.2.4 Randomize data

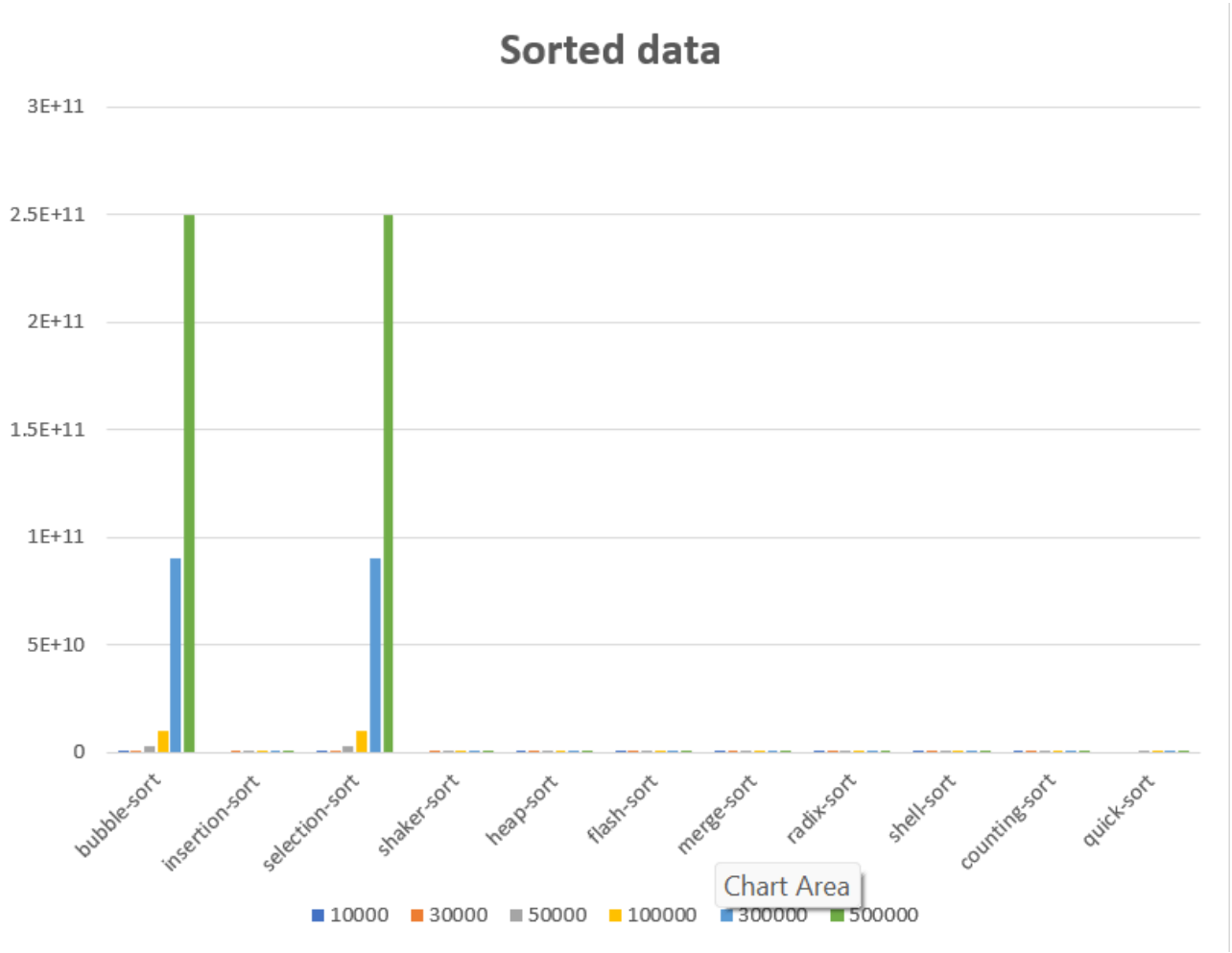


Comment:

- Fastest Algorithm: Counting Sort, because its complexity is $O(n+k)$ with $k = (Max - Min) + 1$, and in this random input, k is always smaller than n, resulting in a complexity of O(n). Therefore, Counting Sort runs extremely fast.
- Slowest Algorithm: Shaker Sort, because its sorting process continues even after the elements are already sorted, making it less efficient and increasing the execution time.

3.3 Bar charts

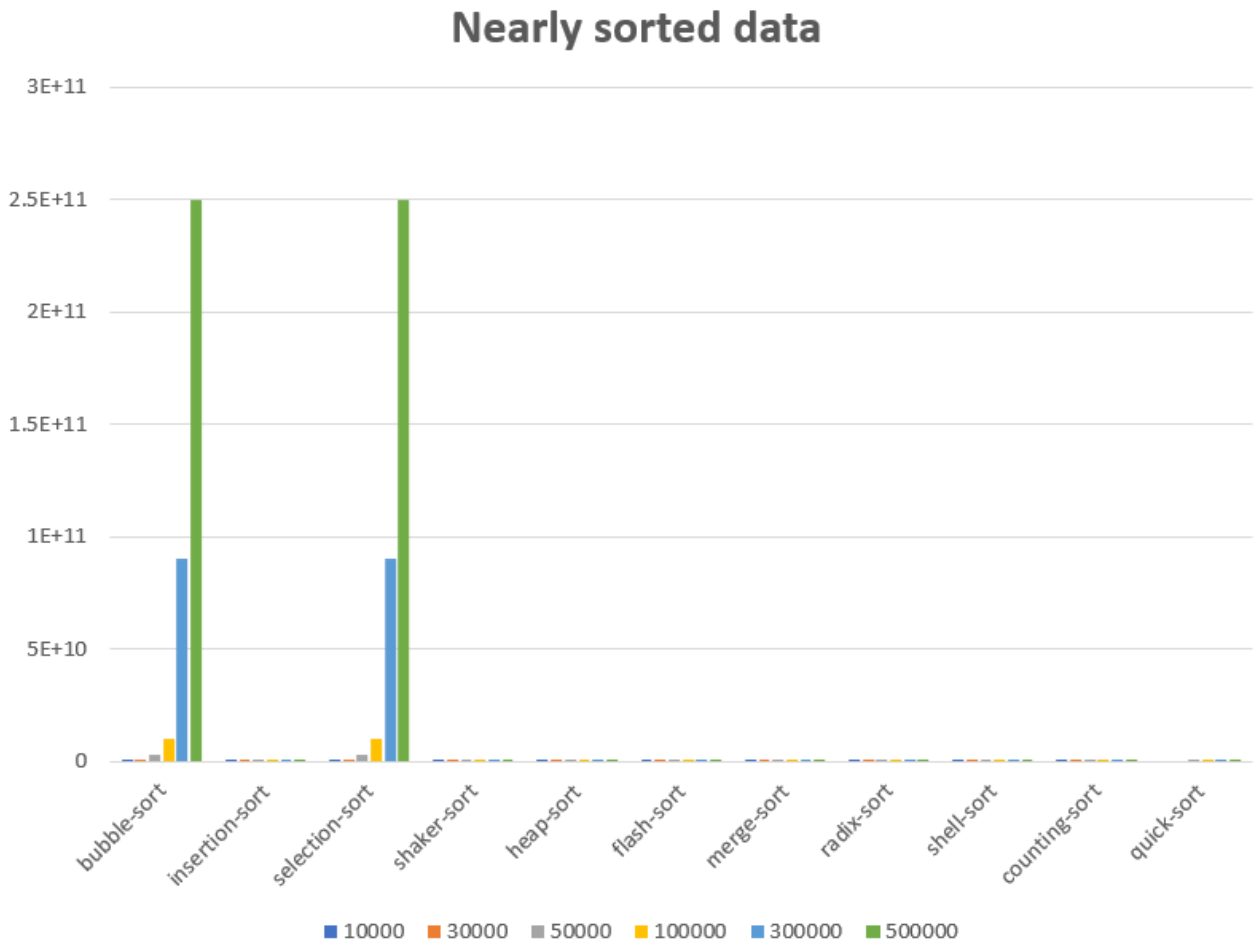
3.3.1 Sorted data



Sorted

- Most Comparisons: Bubble Sort and Selection Sort. These algorithms are comparison-based sorting techniques. When the data is already sorted, both Bubble Sort and Selection Sort still perform a significant number of comparisons to verify the order of elements, which makes them less efficient for sorted data.
- Least Comparisons: Quick Sort. In the case of already sorted data, it can achieve fewer comparisons compared to other sorting algorithms. It efficiently partitions the data and eliminates redundant comparisons, leading to improved performance on sorted data.

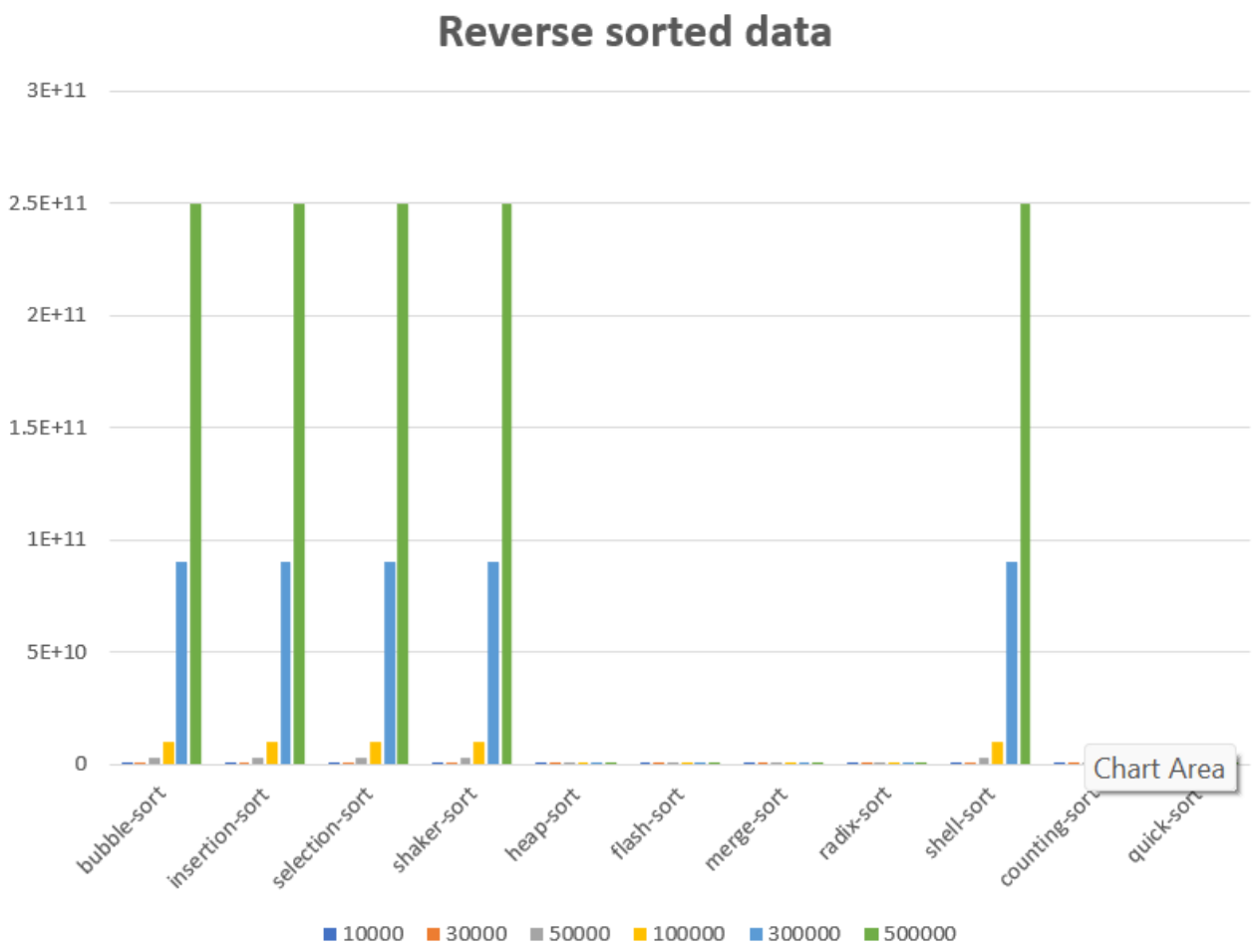
3.3.2 Nearly sorted data



Comment

- Most Comparisons: Bubble Sort and Selection Sort. These algorithms are comparison-based sorting techniques. When the data is nearly sorted, Bubble Sort and Selection Sort still perform a large number of comparisons to rearrange the elements, making them less efficient for nearly sorted data.
- Least Comparisons: Quick Sort. In the case of nearly sorted data, it can achieve fewer comparisons compared to other sorting algorithms. It efficiently partitions the data and performs fewer comparisons due to the partially sorted nature of the input, resulting in improved performance.

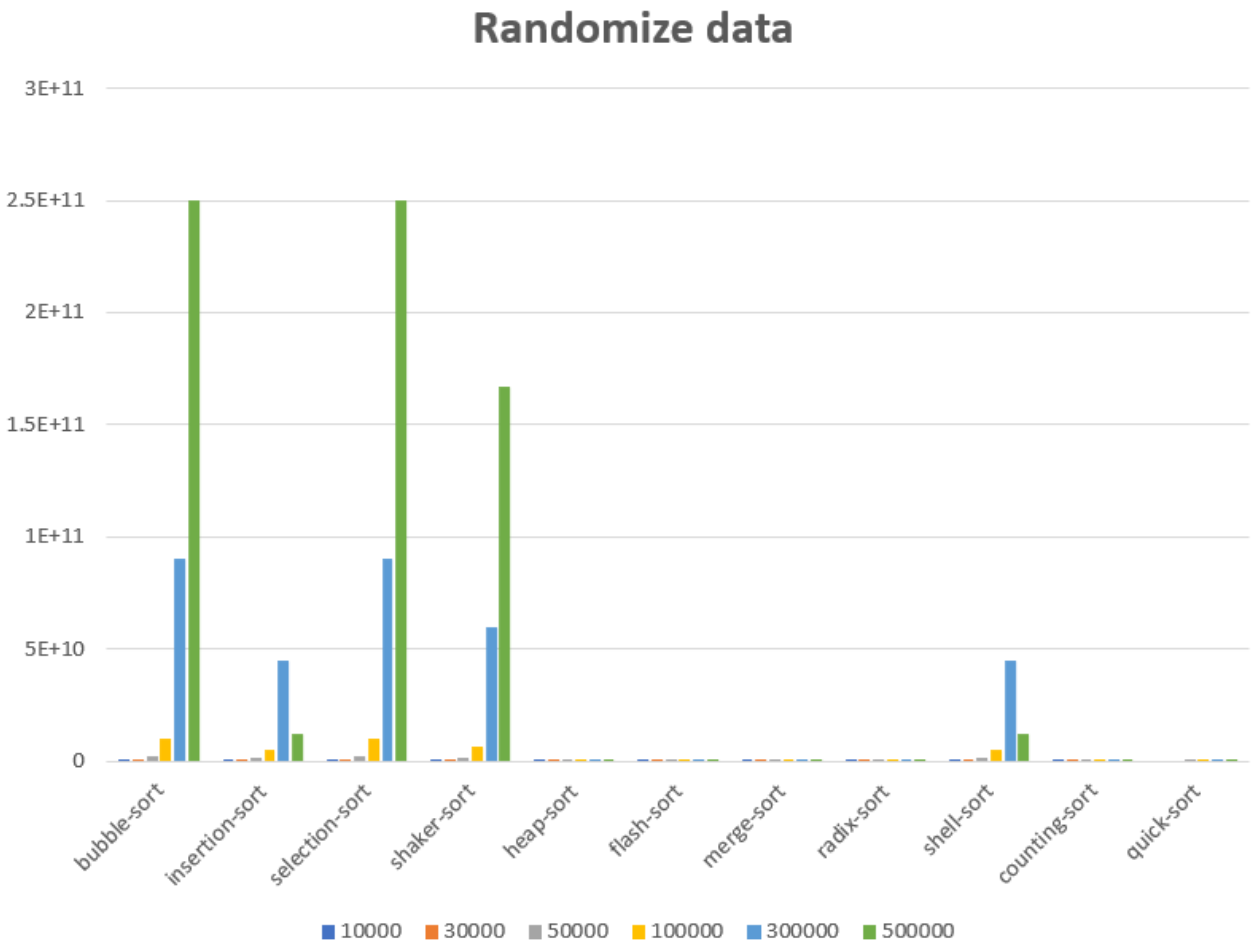
3.3.3 Reverse sorted data



Comment

- Most Comparisons: Bubble Sort, Selection Sort, Insertion Sort, Shaker Sort, and Shell Sort. These sorting algorithms perform a large number of comparisons when dealing with reverse sorted data. They need to repeatedly swap elements to correctly order the reversed input, resulting in higher comparisons.
- Least Comparisons: Quick Sort. Despite being an efficient algorithm for general cases, Quick Sort requires fewer comparisons compared to the mentioned algorithms when dealing with reverse sorted data. Its partitioning technique allows it to avoid unnecessary comparisons and make progress towards sorting the data efficiently.

3.3.4 Randomize data



Comment:

- Most Comparisons: Bubble Sort and Selection Sort. They are both comparison-based sorting algorithms. In the case of random data, these algorithms require a large number of comparisons to arrange the elements in the correct order.
- Least Comparisons: Quick Sort. In the case of random data, Quick Sort can achieve a relatively low number of comparisons compared to Bubble Sort and Selection Sort.

3.4 Conclusion

- Fastest Running Algorithm: Counting Sort.
- Slowest Running Algorithms: Bubble Sort, Selection Sort.
- Stable Sorting Algorithms: Heap Sort, Counting Sort, Flash Sort, Quick Sort, Radix Sort, Merge Sort
- Unstable Sorting Algorithms: Shaker Sort, Bubble Sort, Selection Sort, Insertion Sort, Shell Sort.

4 Project organization & Programming notes

4.1 File organization

In our project, we organized our code files using the common practice of separating the implementation and declaration into header files (.h files) and source files (.cpp files), respectively. This approach helps to maintain a modular and organized codebase.

All the header files are located inside `include/` folder.

4.1.1 `SortStrategy.h`

Declares a base class for other sorting classes to be inherited from.

Property	**Usage**
`int *m_array`	To store the current array
`int m_size`	To store the current array's size
`std::chrono::high_resolution_clock::time_point m_timeStart, m_timeEnd`	To help calculating the overlapsed time an algorithm used
`long long m_count_comparison = 0`	To count the number of comparison when using a specific algorithm

Method	**Usage**
`SortStrategy(int *a, int n)`	Constructor to initialize new object with a given array and its size
`~SortStrategy()`	Default destructor to free the used memory
`virtual void sort() = 0`	Abstract function which allows inherited classes to implement their own algorithm to sort an array
`virtual void sortWithComparison() = 0`	Same as `sort()` but every comparison is being counted
`int* getArray() const`	To get the array itself
`long long getComparison() const`	To get the number of comparison
`int getSize() const`	To get the array's size
`double getDuration() const`	To get the elapsed time

4.1.2 `SortContext.h`

This declare a general context to store the current in-use sorting algorithm. It provides different methods to initialize different algorithms, and switching between them.

Property	**Usage**
`SortStrategy *strategy`	A pointer holds the current sorting algorithm

Method	Usage
`SortStrategy(int *a, int n)`	Constructor to initialize new object with a given array and its size
`SortContext(SortStrategy* strat_)`	Constructor to initialize new object with a give pointer points to a sorting algorithm
`ŠortContext()`	Default destructor to free the used memory
`virtual void sortWithComparison() = 0`	Same as `sort()` but every comparison is being counted
`void setStrategy(SortingAlgorithm algo, int* a, int n)`	To set the current algorithm to a different one given an array and its size
`void sort()`	To sort the array using the current algorithm
`void sortWithComparison()`	Same as sort() but every comparison is being counted

4.1.3 Other `.h` files

They are sorting algorithms' header files, with proper constructors and methods.

The main implementations are located inside `src/name` folder, where `name` represents the main purpose of the file(s) inside it.

4.1.4 `src/SortingAlgorithm`

This directory contains all the implementations of 11 sorting algorithms.

4.1.5 `src/SortingContext`

This directory contains the SortContext's implementation

4.1.6 src/SortingStrategy

This directory contains the SortStrategy's implementation

4.1.7 src/utils

Contains utility files, like `DataGenerator.cpp`

4.1.8 `src/main.cpp`

The soul of this program. It will link and handle everything.

4.2 Libraries

In our project, we utilized several standard libraries in C++ to enhance the functionality and efficiency of our code. Some of the commonly used libraries include:

4.2.1 `iostream`

This library helps us perform basic input and output operations in our C++ program. We can use it to create a friendly interface, receive command line from the user and show results on the screen.

4.2.2 `fstream`

With `fstream`, we can read data from files and write data to files in our C++ program. It helps us store information that we need for our research or keep track of the results.

4.2.3 `chrono`

This library provides tools to measure time in our C++ program. We utilized it to measure the performance of our algorithms and track how long it takes to complete.

4.2.4 `cstring`

`cstring` has functions that help manipulate strings in C++. Since the command-line arguments are represented as `char*` strings, we can use this library to handle and modify them.

4.2.5 `iomanip`

This library assists us in controlling the formatting of output in C++. We applied `iomanip` to format the running time we collected as `double` values. This allowed us to show the results neatly with the desired level of accuracy.

4.2.6 `doctest.h`

`doctest.h` is a external header-only testing framework for C++ that we used to check if our sorting algorithms are working correctly. We wrote various test cases to ensure our sorting algorithms give the right results. It helped us identify and fix issues during the development process.

4.3 How to build

Our source code was organized in different folders and built using CMake (make sure that your computer has already installed CMake with version ≥ `3.25`). Below are the step-by-step instructions to build the project:

1. Navigate inside the `SOURCE` folder, which contains three different folders.
2. Create a new folder called `build` and navigate inside it.
3. Run `cmake -G "Unix Makefiles" ..` and make sure there is no error.
4. Run `cmake --build .` in the current directory (`SOURCE/build`).
5. There will be many new files/folders created, but we only need to care about the file called `sorting_analysis.exe` - which is our [Execution file]. We can move it to a different location and run it with proper command line arguments

OR we can run the already built file `sorting_analysis.exe` located inside `SOURCE/`.

5 References

To work out this project, we have taken many references from many sources. We would like to give credits to them in this section so as to pay our respects to their work:

5.1 Algorithm

1. **Selection Sort:** Selection Sort – Data Structure and Algorithm Tutorials - GeeksforGeeks
2. **Insertion Sort:** Bài 49. Thuat Toan Sap Xep Chen (Insertion sort) & Binary Insertion Sort - GeeksforGeeks
3. **Shaker Sort:** Bubble Sort và Shaker Sort - Artical
4. **Heap Sort:** Heap Sort - GeeksforGeeks
5. **Merge Sort:** Merge Sort - GeeksforGeeks & Iterative Merge Sort Algorithm (Bottom-up Merge Sort)
6. **Flash Sort:** Flash Sort - Thuat Toan Sap Xep Than Thanh & Flash SORT
7. **Radix Sort:** Radix Sort - GeeksforGeeks
8. **Counting Sort, Quick Sort & Shell Sort:** GitHub Sorting-Algorithms by Le Duy Thuc - K19 FIT HCMUS.

5.2 Report presentation

We also took inspiration from Mr. Le Duy Thuc's report and AI Chatbox ChatGPT for the presentation of step-by-step algorithms (Algorithm Presentation) and some remarks in our report.