UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

CSC10004 - Data Structures and Algorithms

PROJECT REPORT

THE SORTING PROJECT

CLASS 23CLC10

Students:
Hà Bảo Ngọc – 23127300
Nguyễn Đỗ Bảo – 23127026
Lâm Vĩ Khang – 23127062
Đỗ Ngọc Minh Tuấn – 23127137

Instructors:
Mr. Bùi Huy Thông
Mr. Nguyễn Trần Duy Minh

Contents

List of Figures

1 Information

1.1 Project Overview

This project involves the implementation and detailed analysis of 11 sorting algorithms as part of our Data Structures & Algorithms course. Our objective is to understand the performance characteristics of each algorithm when applied to various datasets. The algorithms we have chosen to research from Set 2 include:

- Selection Sort
- Insertion Sort
- Bubble Sort
- Shaker Sort
- Shell Sort
- Heap Sort
- Merge Sort
- Quick Sort
- Counting Sort
- Radix Sort
- Flash Sort

1.2 Objectives

1. Implement the selected sorting algorithms in C/C++.
2. Evaluate each algorithm's performance in terms of running time and the number of comparisons.
3. Analyze the algorithms' efficiency on datasets of varying sizes (10,000, 30,000, 50,000, 100,000, 300,000, and 500,000 elements).
4. Assess the impact of different data orders (sorted, nearly sorted, reverse sorted, and randomized) on the performance of each algorithm.
5. Document the results through detailed tables and visualizations, including line graphs and bar charts.

1.3 Methodology

1. **Implementation:** Develop C/C++ code for each of the eleven sorting algorithms.
2. **Experimental Setup:** Create arrays of different sizes and orders, run each algorithm, and record the performance metrics.
3. **Data Collection:** Measure the running time in milliseconds and count the number of comparisons for each sorting operation.
4. **Analysis:** Organize the collected data into tables and generate visual graphs to facilitate a comparative analysis of the algorithms.
5. **Reporting:** Compile the findings into a comprehensive report, including observations and conclusions based on the experimental results.

1.4 Expected Outcomes

- A detailed understanding of the operational mechanisms of each sorting algorithm.
- Insights into the strengths and weaknesses of each algorithm concerning different data orders and sizes.
- Practical knowledge of the implementation and optimization of sorting algorithms in C/C++.
- A well-documented report that serves as a reference for understanding the comparative performance of sorting algorithms.

1.5 Deliverables

1. **Source Code:** C/C++ implementations of all eleven sorting algorithms.
2. **Experimental Data:** Collected data on running times and number of comparisons.
3. **Report:** A comprehensive PDF document including:
    - Information Page
    - Introduction
    - Algorithm Presentation
    - Experimental Results and Comments
    - Project Organization and Programming Notes
    - List of References
4. **Visualizations:** Line graphs and bar charts illustrate the performance metrics.

This project aims to deepen our understanding of sorting algorithms and enhance our ability to apply these fundamental techniques to real-world data processing challenges.

2 Introduction

Sorting algorithms are pivotal in computer science, and essential for efficient data management and retrieval. This research project, conducted by the Knowledge Engineering Department, focuses on implementing and analyzing a diverse set of sorting algorithms to evaluate their performance across various conditions. Our team has selected Set 2 for this study, which includes eleven algorithms: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort.

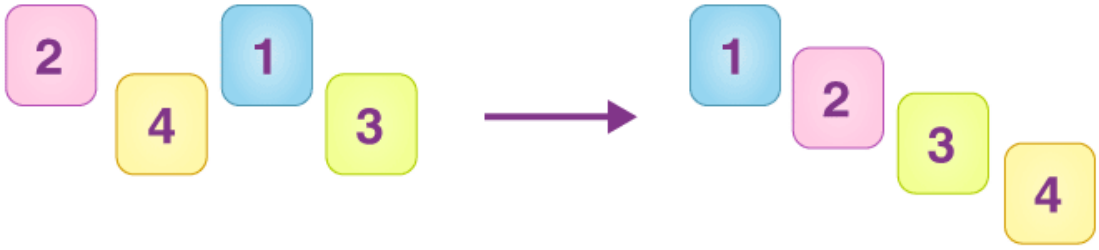


Figure 1: Sorting Algorithms

This project aims to provide a comprehensive analysis of these sorting algorithms by implementing them in C/C++ and conducting systematic experiments. The experiments are designed to measure the running time and the number of comparisons made by each algorithm under different scenarios. Specifically, we will assess their performance on datasets of varying sizes (ranging from 10,000 to 500,000 elements) and different data orders (sorted, nearly sorted, reverse sorted, and randomized).

Through this extensive evaluation, we aim to gain insights into the practical and theoretical efficiencies of each sorting algorithm. The results will be meticulously documented, including detailed tables and visual graphs that illustrate the performance metrics. This report will present the implemented algorithms, the experimental methodology, the results, and a thorough analysis, providing a clear understanding of the strengths and weaknesses of each sorting technique within Set 2.

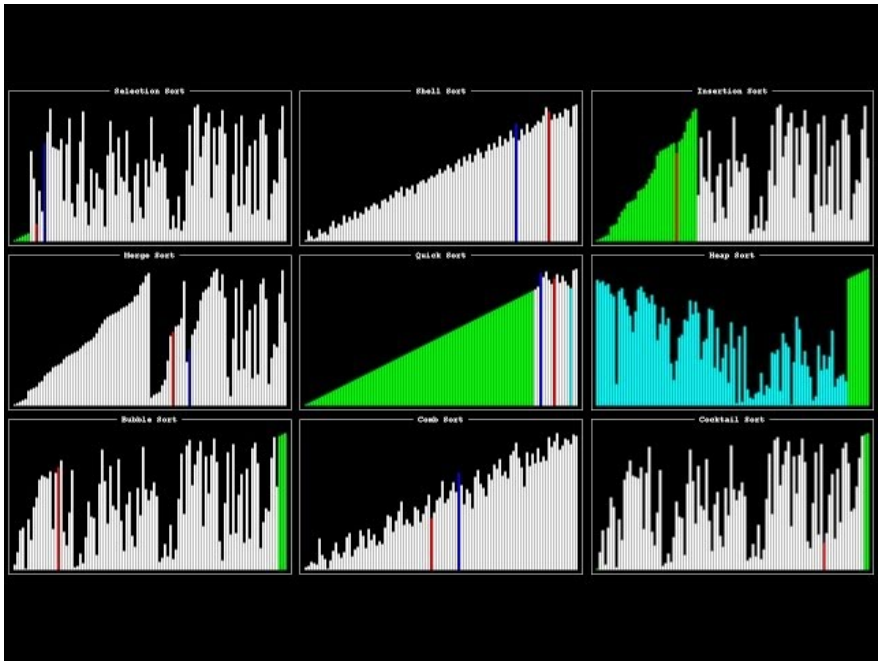


Figure 2: Sorting Algorithms Visualizations

By the end of this project, we hope to offer valuable conclusions that can inform the selection and application of sorting algorithms in various computational contexts, contributing to the broader knowledge base in the field of data structures and algorithms.

2.1 Symbols

Many symbols are used in this article, with certain popular symbols being repeated repeatedly, as mentioned below:

- a: input data array.
- n: size of the input data array ($1 \leq n \leq 5 \times 10^5$).
- a_i: the element at the i-th position of the array a. In this report, the array is indexed from 0, meaning the first element of the array is a_0 and the last element is a_{n-1} ($0 \leq a_i < n \, \forall \, i \in [1, n]$).

2.2 Compilation Configuration

The program is compiled using the GNU C++ Compiler. Therefore, any other compiler like Microsoft Visual C++ Compiler (MSVC), Clang, etc., cannot compile the source code and produce the executable file (.exe) to run the source code.

Specifically, the source code is compiled in the C++11 standard with the following command:

```
g++ -std=c++11 main.cpp -o main.exe
```

To ensure that the project is error-free, compile and run the program using the terminal commands as shown below. This confirms that the program operates correctly when all required files are present in the source code directory.

2.3 Remarks

- The following sorting algorithms are implemented to arrange array elements in increasing order.
- All execution times in the diagrams and tables are recorded in milliseconds, any exceptions will be explicitly noted.
- All algorithms are executed on identical machine configurations and run in a single-threaded mode.
- To provide deeper insights into the algorithms, we have included some enhanced or variant versions of the original sorting algorithms. These versions can be executed via terminal commands with the names:

3 Algorithm presentation

Sorting is one of the most fundamental operations in computer science, essential for organizing data in a meaningful order. From searching algorithms that depend on sorted data to database indexing and optimizing various computational tasks, sorting algorithms are at the heart of efficient data manipulation.

In this chapter, we embark on a journey through the realm of sorting algorithms, exploring their principles, implementations, and performance characteristics. Sorting algorithms come in various forms, each with unique strategies and efficiencies. Understanding these algorithms is crucial not only for theoretical knowledge but also for practical applications where performance and efficiency are paramount.

We will cover a diverse set of sorting algorithms, ranging from simple, intuitive methods to more complex, sophisticated techniques. This exploration includes: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, Flash Sort Each of these algorithms will be explored in detail, from their conceptual foundation and step-by-step process to their implementation in C/C++. We will also conduct experiments to measure their performance on various data sets and sizes, providing a comprehensive understanding of their strengths and weaknesses.

By the end of this chapter, readers will have a solid grasp of different sorting techniques, their theoretical complexities, practical implementations, and real-world applications. This foundational knowledge is essential for tackling more advanced topics in data structures and algorithms, and for applying efficient sorting methods in everyday programming tasks.

System Information: The sorting algorithms will be executed on the following system

- **Operating System:** Windows 11 Home Single Language 64-bit (10.0, Build 22631)
- **Language:** English (Regional Setting: English)
- **System Manufacturer:** ASUSTeK COMPUTER INC.
- **System Model:** ROG Strix G513RC_G513RC
- **BIOS:** G513RC.327
- **Processor:** AMD Ryzen 7 6800HS with Radeon Graphics (16 CPUs)
- **Memory:** 8192MB RAM
- **Page file:** 6846MB used, 8782MB available
- **DirectX Version:** DirectX 12

3.1 Selection Sort

3.1.1 Ideas

The idea behind selection sort is to repeatedly find the minimum element from the unsorted part of the array and swap it with the element at the beginning of the unsorted part. This process continues until the entire array is sorted.

3.1.2 Step-by-step descriptions

Assuming we have the array [2 5 1 9 6], n = 5. The insertion sort algorithm sorts the array as illustrated below:

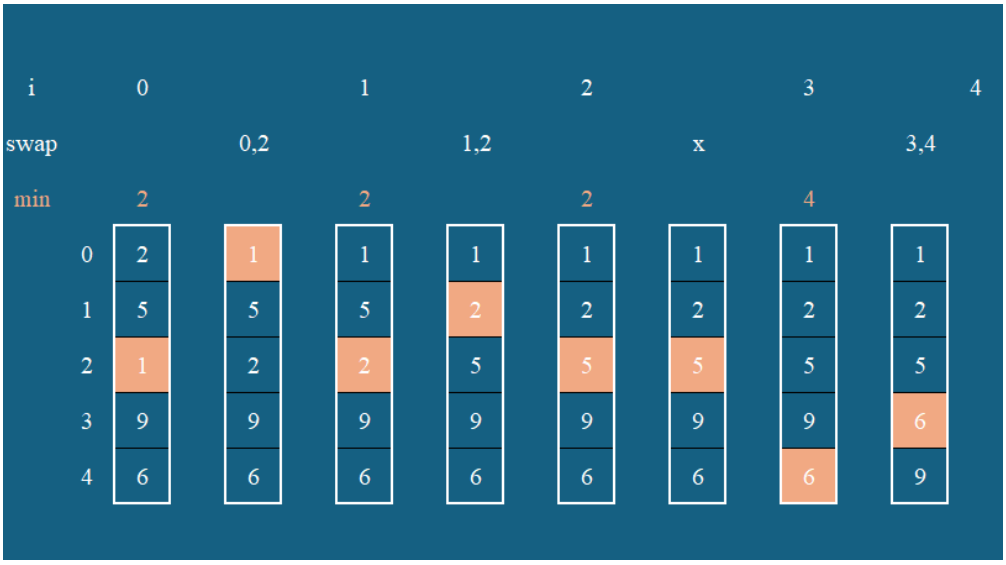


Figure 3: Selection Sort Visualization

Sorted array: [1 2 5 6 9]

3.1.3 Complexity Evaluations

- **Time Complexity:** The time complexity of Selection Sort is O(n^2)
  Choose key operation: comparison (compare min with elements in the unsorted array)
  Outer loop: run from 0 to n - 2.
  Inner loop:

$$
\begin{aligned}
&i = 0, && j \text{ runs from } 1 \text{ to } n-1 \Rightarrow n-1 \text{ comparisons}\\
&i = 1, && j \text{ runs from } 2 \text{ to } n-1 \Rightarrow n-2 \text{ comparisons}\\
&i = 2, && j \text{ runs from } 3 \text{ to } n-1 \Rightarrow n-3 \text{ comparisons}\\
&\vdots\\
&i = n-3, && j \text{ runs from } n-2 \text{ to } n-1 \Rightarrow 2 \text{ comparisons}\\
&i = n-2, && j \text{ runs from } n-1 \text{ to } n-1 \Rightarrow 1 \text{ comparison}
\end{aligned}
$$

Call $T(n)$ the time to execute the Selection Sort algorithm.

$$
\begin{aligned}
T(n) &= 1 + 2 + \cdots + (n-2) + (n-1)\\
&= \sum_{i=1}^{n-1} i\\
&= \frac{n(n-1)}{2}
\end{aligned}
$$

Therefore, the time complexity is $O(n^2)$.

  - Best case: The input array is in the correct order, and the time complexity is $O(n^2)$.
  - Average case: The input array is in random order, and the time complexity is $O(n^2)$.
  - Worst case: The array is in reverse order, and the time complexity is $O(n^2)$.

- **Space Complexity:** O(1), this is an in-place algorithm.

3.1.4 Variants/Improvements

In each iteration, instead of finding only the minimum element in the unsorted part, we find both the minimum and maximum elements. By moving the minimum element to the beginning and the maximum element to the end, we can reduce the number of iterations by half.

3.2 Insertion Sort

The algorithm will divide the array into two parts: the sorted array on the left and the unsorted array on the right. In each step, the algorithm will consider the leftmost element of the unsorted array and "insert" that element into its correct position in the sorted array until the entire array is sorted.

3.2.1 Ideas

The idea behind selection sort is to repeatedly find the minimum element from the unsorted part of the array and swap it with the element at the beginning of the unsorted part. This process continues until the entire array is sorted.

3.2.2 Step-by-step descriptions

Theoretically, the array is divided into two parts: the unsorted array and the sorted array. In this example, the sorted array is [5] and the unsorted array is [2 8 3 1].

1. Consider the leftmost element of the unsorted array, called `temp`.
2. Compare `temp` with each element of the sorted array starting from the right, simultaneously shifting the larger elements to the right by one position.
3. Insert `temp` into its correct position in the sorted array.
4. Repeat the above three steps until the entire array is sorted.

Assuming we have the array [5 2 8 3 1], n = 5. The insertion sort algorithm sorts the array as illustrated below:

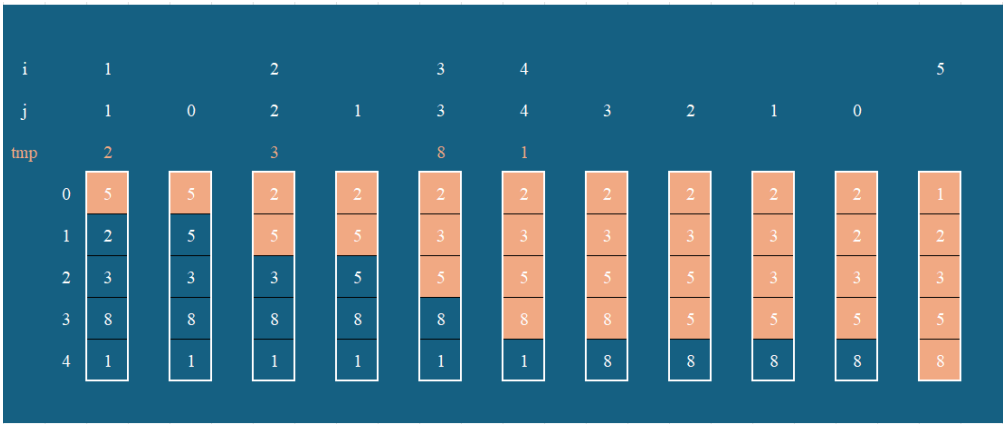


Figure 4: Insertion Sort Visualization

Sorted array: [1 2 3 5 8]

3.2.3 Complexity Evaluations

- **Time Complexity:**
  - Best case: The input array is already sorted in the correct order. Therefore, for each position i, only one comparison is needed, resulting in a total of $n˘1$ comparisons, i.e., $O(n)$.
  - Worst case: The input array is sorted in reverse order (i.e., descending). Therefore, the number of comparisons for each position i is i times. The total number of comparisons is:
$$1 + 2 + \ldots + (n˘1) = \frac{(n-1)n}{2} = O(n^2)$$
  - Average case: The array is randomly ordered. The number of comparisons for each position i is the distance of position i to its correct position in the sorted array. Therefore, the number of comparisons can range from 1 to i. So, in the average case, we need to calculate the average number of comparisons for each position i by summing up all possible cases and dividing by the total number of cases. We get:
$$\frac{1 + 2 + \ldots + i}{i} = \frac{i}{2} \cdot \frac{(i+1)}{i} = \frac{(i+1)}{2}$$
    Thus, the number of comparisons in the average case is:
$$\sum_{i=1}^{n-1} \frac{i+1}{2} = \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} \frac{1}{2} = \frac{1}{2}\frac{n(n-1)}{2} + \frac{1}{2}(n-1) = \frac{n^2+n-2}{4} = O(n^2)$$
- **Space Complexity:** O(1), this is an in-place algorithm.

3.2.4 Variants/Improvements

Some improvements of insertion sort include Shell Sort, which will be introduced later in the report, and Binary Insertion Sort.

Binary Insertion Sort: Since the idea of the insertion sort algorithm is to find the suitable position for the current element, we can apply the binary search method to improve the time required to find the position for the element.

3.3 Bubble Sort

3.3.1 Ideas

The idea of the algorithm is quite simple: swap two adjacent elements if they are in the wrong order. In each iteration, the algorithm will go through each element from the end of the array to the current element being considered. The algorithm stops when the last element of the array is considered.

3.3.2 Step-by-step descriptions

1. Select the current position.
2. Iterate from the end of the array to the current position and swap adjacent elements if they are in the wrong order.
3. Repeat steps 1 and 2 until the current position is the last element of the array, indicating the array is completely sorted.

Assuming we have the array [5 2 3 8 1], n = 5. The bubble sort algorithm sorts the array as illustrated below.

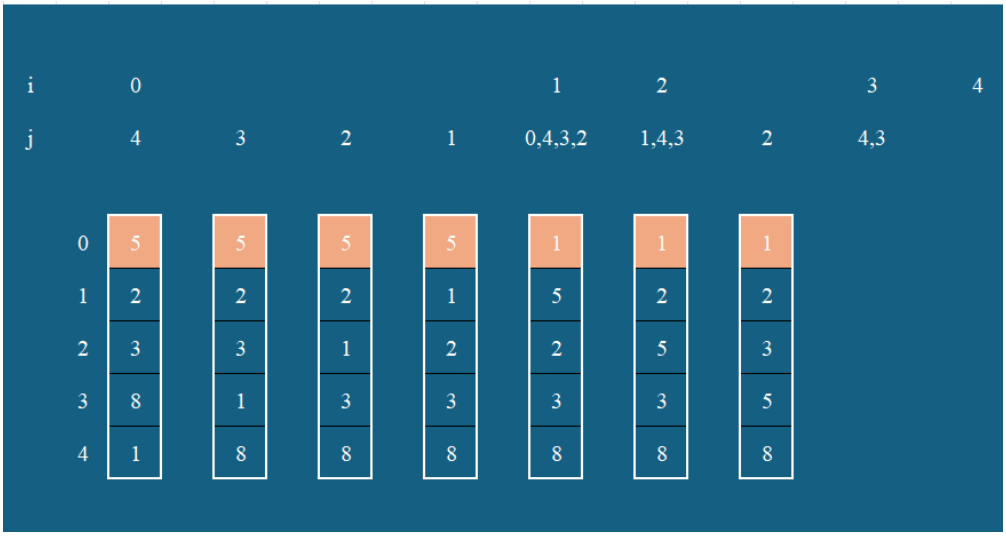


Figure 5: Bubble Sort Visualization

Sorted array: [1 2 3 5 8]

3.3.3 Complexity Evaluations

- **Time Complexity:** The number of comparisons for bubble sort is the same for all cases because the algorithm does not have a stopping condition for different input arrays.

$$\sum_{i=0}^{n-2}(n-1-i) = \frac{n(n-1)}{2} = O(n^2)$$

- **Space Complexity:** O(1), this is an in-place algorithm.

3.3.4 Variants/Improvements

Some variants and improvements of the bubble sort algorithm include Comb Sort, Bubble Sort with Flag, and Shaker Sort, which will be introduced later.

Bubble Sort with Flag: This is an improved version of Bubble Sort to reduce unnecessary comparisons by stopping early if the array is already sorted. Specifically, it uses a flag variable to track whether any elements were swapped in each iteration through the array. If no elements were swapped, it means the array is already sorted, and the algorithm can stop immediately.

Comb Sort: This algorithm uses an initial gap larger than Bubble Sort and gradually reduces this gap in subsequent iterations. This helps larger elements move faster to their correct positions in the array.

3.4 Shaker Sort

3.4.1 Ideas

This is an improved version of Bubble Sort that addresses the inefficiencies of Bubble Sort with large arrays. The Shaker Sort algorithm prevents iterating over already sorted arrays multiple times and examines both ends of the array in a single loop.

3.4.2 Step-by-step descriptions

1. The first loop examines the array from left to right and shifts two adjacent elements if they are in the wrong order, similar to the Bubble Sort algorithm. If no elements are shifted after the loop, the algorithm terminates. Otherwise, the examination limit contracts, and step 2 is executed.
2. The second loop examines the array in the opposite direction, from right to left, with the same algorithm: swapping two adjacent elements if they are in the wrong order and stopping the algorithm if necessary.

Assuming we have the array [5 1 2 3 8], n = 5. The Shaker Sort algorithm proceeds on the array as illustrated, with colored cells indicating the boundaries of the subarray being examined.

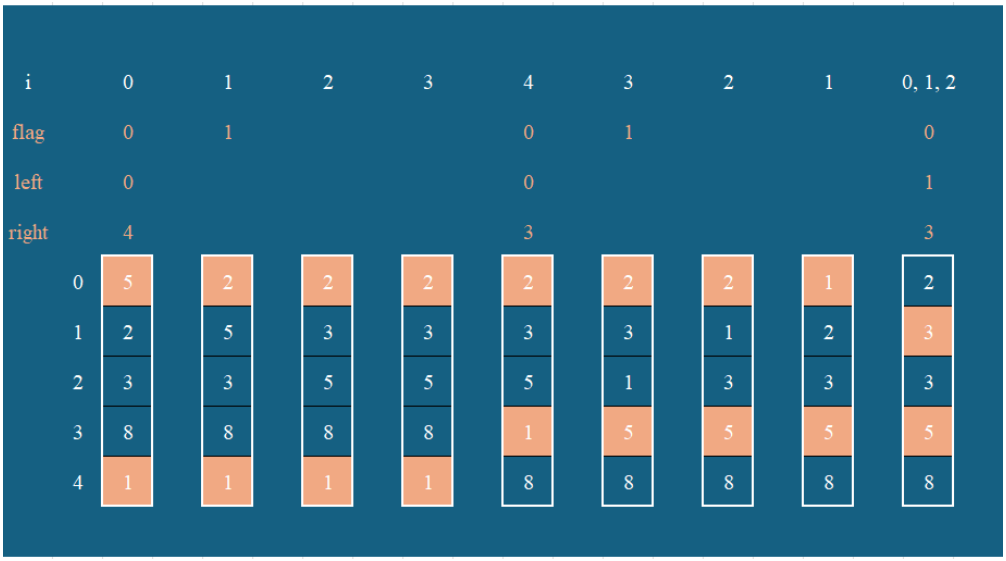


Figure 6: Shaker Sort Visualization

Sorted array: [1 2 3 5 8]

3.4.3 Complexity Evaluations

- **Time Complexity:**
  - Best case: The array is already sorted. Therefore, the algorithm will terminate after the first iteration. Complexity $O(n)$.
  - Average case: Similar to Bubble Sort, the average case for Shaker Sort occurs when the array is randomly sorted with alternating increasing and decreasing segments. In this situation, the algorithm complexity is still $O(n^2)$.
  - Worst case: The array is sorted in descending order. The number of comparisons will be the highest. Complexity $O(n^2)$.
- **Space Complexity:** O(1), this is an in-place algorithm.

3.5 Shell Sort

3.5.1 Ideas

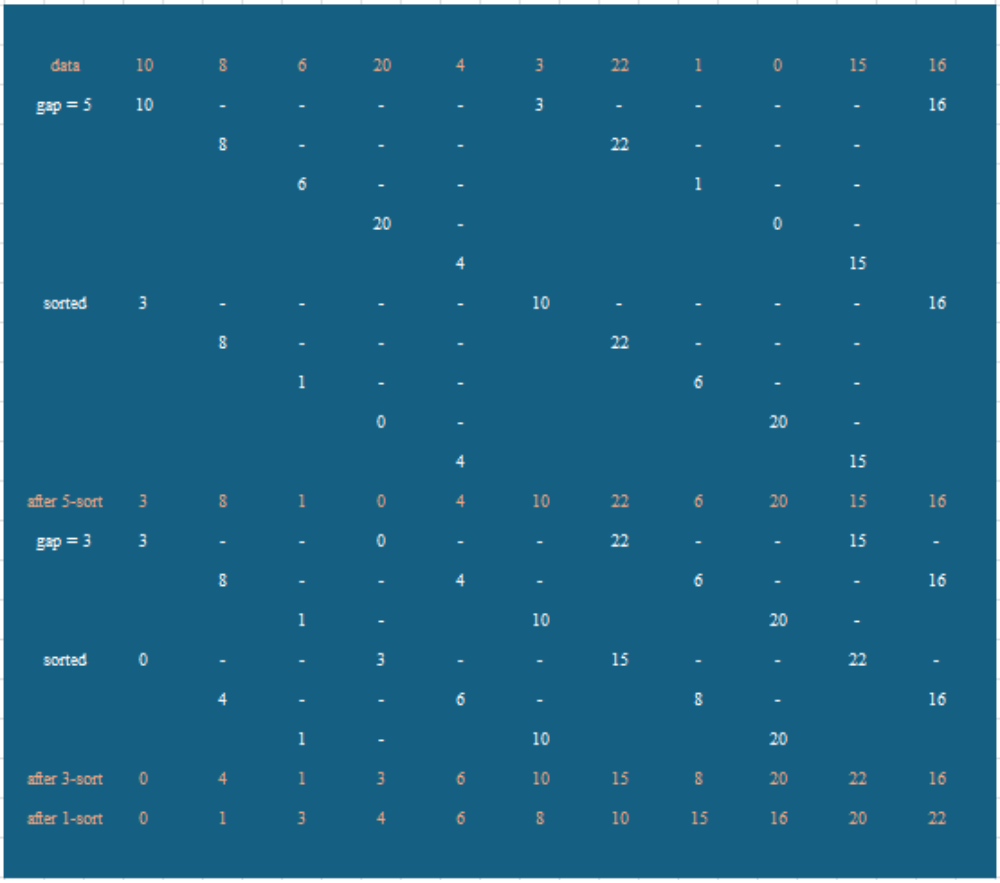
As introduced earlier, Shell Sort is an improved version of the insertion sort algorithm. The algorithm is based on the diminishing increment sort method, where the gap between elements to be compared and sorted decreases over steps. The algorithm can be implemented using various sorting methods, but the most common is insertion sort. The algorithm divides the initial array into many sub-arrays with elements far apart being compared first, then narrows the gap, and finally compares consecutive elements in the last iteration.

3.5.2 Step-by-step descriptions

The algorithm has three main steps:

1. Calculate the gap.
2. For the chosen gap, compare and sort the pairs of elements that satisfy that gap.
3. Repeat steps 1 and 2 with a smaller gap. The algorithm ends when the gap is 1.

Assuming we have the array [10 8 6 20 4 3 22 1 0 15 16], n = 11 with gaps [5 3 1]. The Shell Sort algorithm can be illustrated as follows:



data	10	8	6	20	4	3	22	1	0	15	16
gap = 5	10	-	-	-	-	3	-	-	-	-	16
		8	-	-	-		22	-	-	-	
			6	-	-			1	-	-	
				20	-				0	-	
					4					15	
sorted	3	-	-	-	-	10	-	-	-	-	16
		8	-	-	-		22	-	-	-	
			1	-	-			6	-	-	
				0	-				20	-	
					4					15	
after 5-sort	3	8	1	0	4	10	22	6	20	15	16
gap = 3	3	-	-	0	-	-	22	-	-	15	-
		8	-	-	4	-		6	-	-	16
			1	-		10			20	-	
sorted	0	-	-	3	-	-	15	-	-	22	-
		4	-	-	6	-		8	-		16
			1	-		10			20		
after 3-sort	0	4	1	3	6	10	15	8	20	22	16
after 1-sort	0	1	3	4	6	8	10	15	16	20	22

Figure 7: Shell Sort Visualization

Sorted array: [0 1 3 4 6 8 10 15 16 20 22]

3.5.3 Complexity Evaluations

- **Time Complexity:** The time complexity of the Shell Sort algorithm depends on the gap sequence used, and there are many different gap sequences available. The algorithm implemented in the file <shell_sort.h> uses the Knuth sequence for gap calculation. The complexity of the Shell Sort algorithm with the Knuth sequence is still a topic of research. The estimated complexity of the algorithm is:

  ‘‘‘

  - Best-case complexity: $O(n \log n)$
  - Average complexity: $O\left(n^{\frac{3}{2}}\right)$
  - Worst-case complexity: $O\left(n^{\frac{3}{2}}\right)$

- **Space Complexity:** O(1), this is an in-place algorithm.

3.6 Heap Sort

3.6.1 Ideas

The idea behind Heap Sort is similar to Selection Sort, where in each iteration we find the largest element and move it to the end of the array. However, Heap Sort is more efficient because the cost of finding the largest element is lower compared to Selection Sort.

Heap: A heap is a special type of binary tree where any given element is always larger than its two child elements. The input one-dimensional array will be considered a heap according to the following rule: "If the parent element is a[i], then it must be greater than or equal to its two child elements a[2i+1] and a[2i+2]". From this, we can see that the largest element will be at the beginning of the array.

Array a = [10 9 6 7 8 4 1 3 2 5] can be written in heap form as follows:

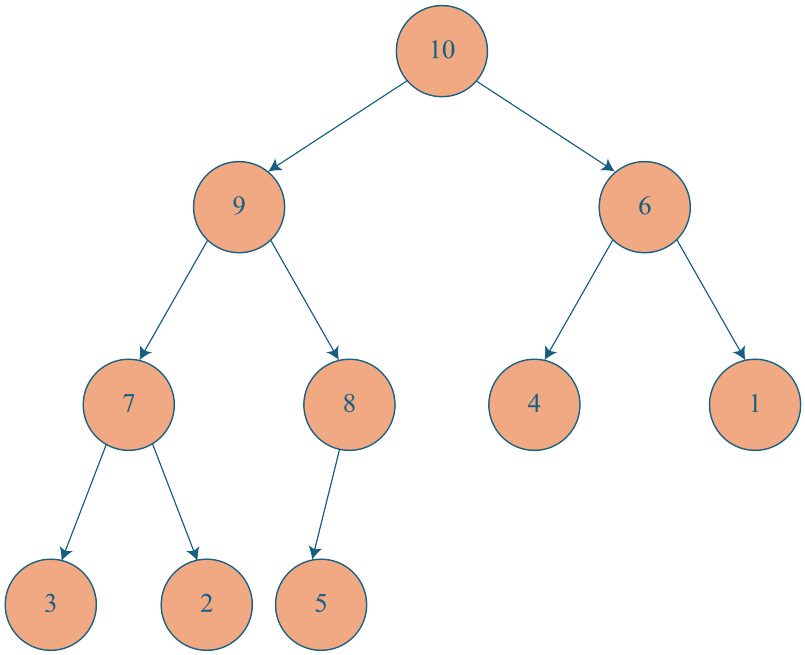


Figure 8: Heap Sort - Visualization

Heapify: We check whether the parent element is greater than its two child elements. If it is not, we swap the parent and child elements and then heapify the child element we just swapped. If it is, we check the other elements.

Build Max-Heap: We heapify $n/2$ elements from the bottom up. The complexity of this operation is $O(n)$.

3.6.2 Step-by-step descriptions

The Heap Sort algorithm consists of 2 stages:

1. ***Stage 1:*** Build the heap
   When considering any parent element a[i], we assume that its branches are already heaps (the parent elements are greater than or equal to their child elements) and heapify from the bottom up. During heapify, we check whether the parent element is greater than its two child elements. If it is not, we swap the parent and child elements and then heapify the child element we just swapped. If it is, we check other elements.
2. ***Stage 2:*** Move the first element (the largest element) to the end of the array, remove the new largest element from the heap, and then adjust the heap (heapify the first element).

For the input array: [2 5 6 1 4 9], n = 6

1. ***Stage 1:*** Build the heap, starting from $n/2$ - 1 to 0. Stage 1 can be visualized as follows:

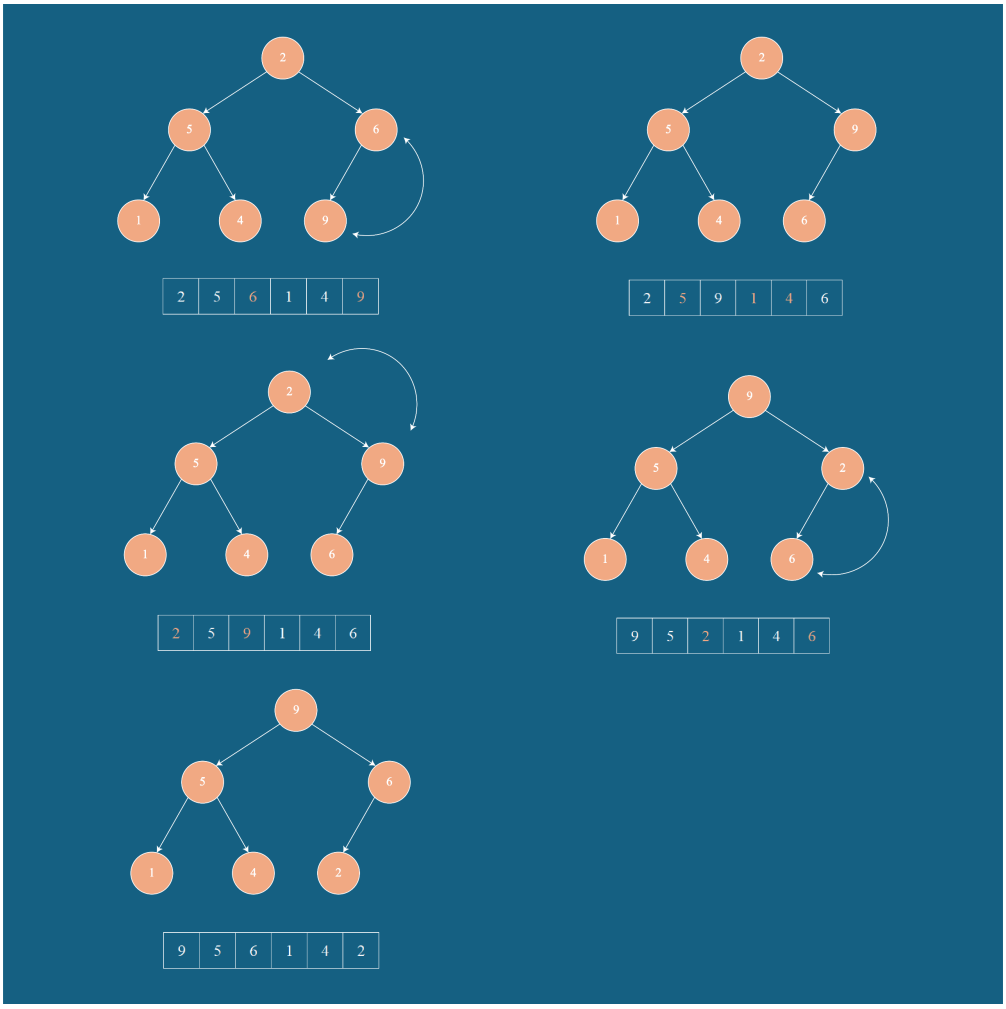


Figure 9: Heap Sort - Stage 1 visualization

2. ***Stage 2:*** Move the first element to the end of the array, remove it from the tree, and adjust the heap. Stage 2 can be visualized as follows:

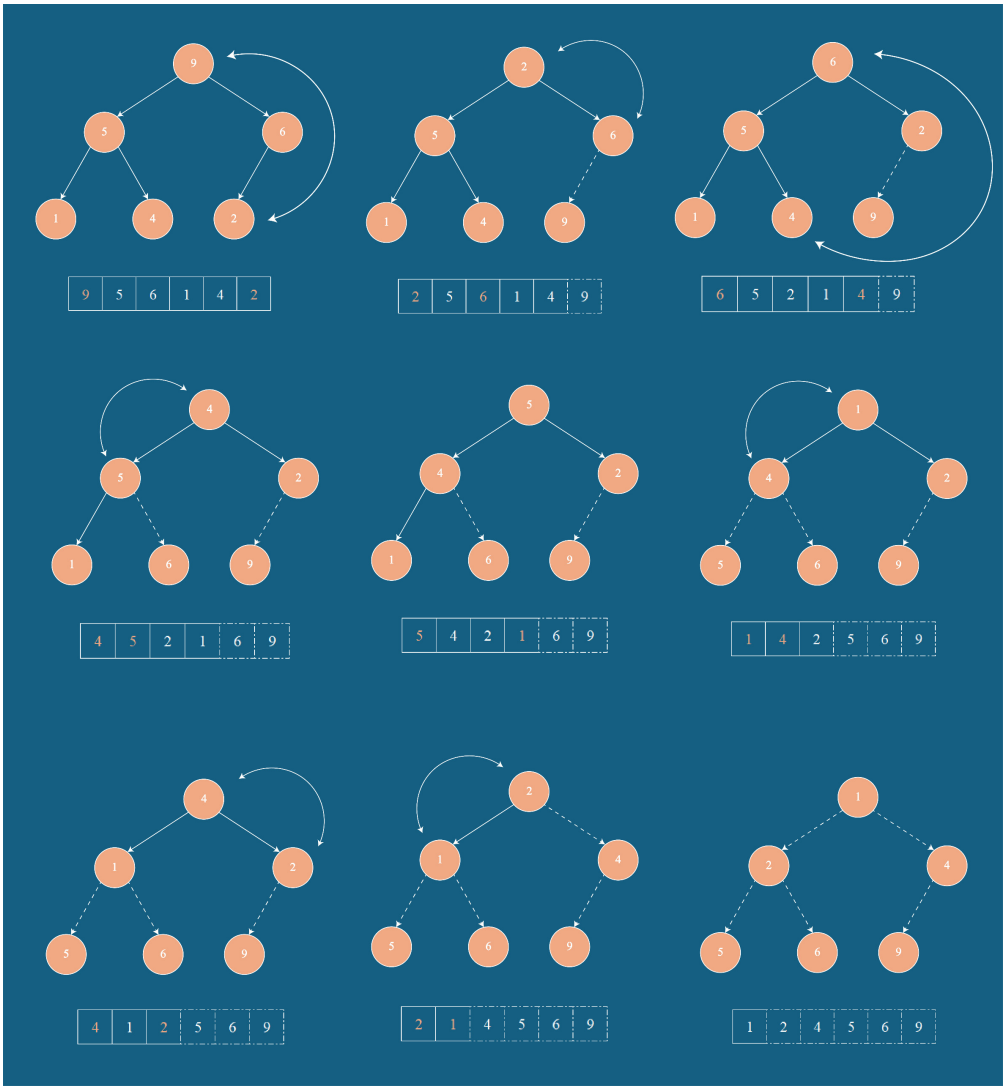


Figure 10: Heap Sort - Stage 2 visualization

3.6.3 Complexity Evaluations

- **Time Complexity:** The time complexity of Heap Sort is $O(nlogn)$ in all cases. Let's consider the time complexity at different stages:

  1. *Heapify:* The complexity of this operation is $O(logn)$ because we need to perform it at most $log_2 n$ times (equivalent to the height of the tree) to maintain the heap property.
  2. *Build Heap:* In this stage, we heapify $n/2$ times, so the complexity of this operation is: $O(nlogn)$.

  In the Heap Sort algorithm, we first build the heap. Then, we move the largest elements

to the end of the array and heapify the first element, repeating this n-1 times. Thus, the complexity of the Heap Sort algorithm is $O(nlogn) + O(nlogn) = O(nlogn)$.

- **Space Complexity:** $O(1)$, as all operations are performed in place within the array.

3.7 Merge Sort

3.7.1 Ideas

Merge Sort is a sorting algorithm based on the *Divide and Conquer principle*. In this algorithm, we repeatedly split an array into smaller halves, sort these smaller arrays, and then merge them back together. This process is repeated recursively until the sub-arrays contain zero or only one element.

The idea can be summarized as follows:

```
mergeSort(array a)
    if (a is empty or contains only one element)
        return
    else
        mergeSort(left half of a)
        mergeSort(right half of a)
        merge(the two halves into one larger array)
```

3.7.2 Step-by-step descriptions

Suppose we have an array as follows:

[26 4 76 96 35 44 18 13], n = 8. The Merge Sort algorithm sorts the array as illustrated below:

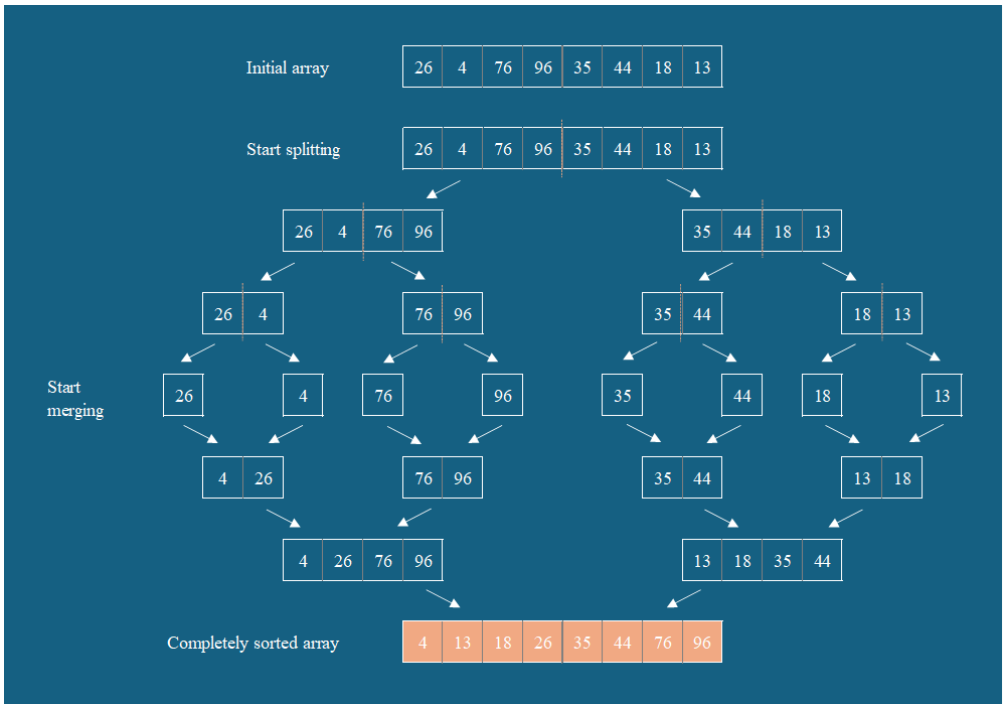


Figure 11: Merge Sort Visualization

Sorted array: [4 13 18 26 35 44 76 96]

3.7.3 Complexity Evaluations

- **Time Complexity:** Divide the time complexity into two processes: *splitting* and *merging*.

  1. *Splitting process*
     Each recursive call splits the array into two halves. This step has a complexity of $O(1)$ since it only needs to compute the midpoint index.
  2. *Merging process*
     To merge two sub-arrays, we need to compare their elements. This step has a complexity of $O(n)$.

  Since the array is split in half with each recursive call, the number of recursive calls will be $log_2(n)$.

  Thus, the time complexity of Merge Sort can be expressed as:

  $T(n) = 2T(\frac{n}{2}) + O(n)$

  Applying the Master Theorem to the above expression, the time complexity of Merge Sort is $O(nlogn)$.

Specifically:

  - Best case: $O(nlogn)$
  - Average case: $O(nlogn)$
  - Worst case: $O(nlogn)$

- **Space Complexity:** Merge Sort creates auxiliary arrays during the merging process, resulting in a space complexity of $O(n)$.

3.7.4 Variants/Improvements

- **Bottom-Up Merge Sort:** Instead of recursion, this variant merges the array iteratively. Elements are merged from individual elements upwards, rather than splitting and then merging as in the traditional approach.
- **Multi-Way Merge Sort:** Instead of splitting the array into two, this variant splits the array into more sub-arrays and merges them.
- **Timsort:** A hybrid sorting algorithm combining Merge Sort and Insertion Sort. It splits the array into sub-arrays and uses Insertion Sort for those sub-arrays, finally merging the sorted sub-arrays. This algorithm can achieve $O(n)$ time complexity when the original array is nearly sorted.

3.8 Quick Sort

3.8.1 Ideas

Quick Sort is a sorting algorithm based on the *Divide and Conquer principle*, similar to Merge Sort. First, a pivot element is selected, and other elements are partitioned around the pivot such that elements smaller than or equal to the pivot are on the left and elements larger than the pivot are on the right. After this step, the pivot is in its final position. Quick Sort is then applied recursively to the sub-arrays formed by partitioning around the pivot. The process can be outlined as follows:

```
quickSort(array a)
   if (a has 2 or more elements)
       select a pivot element
       while (elements remain in the array)
           sort elements into left and right subarrays around the pivot
       quickSort(left subarray)
       quickSort(right subarray)
```

3.8.2 Step-by-step descriptions

Note: In this description, the pivot is chosen as the first element of the array.

Suppose we have an array as follows:

[23 4 76 84 95 44 18 14], n = 8. The Quick Sort algorithm sorts the array as illustrated below:

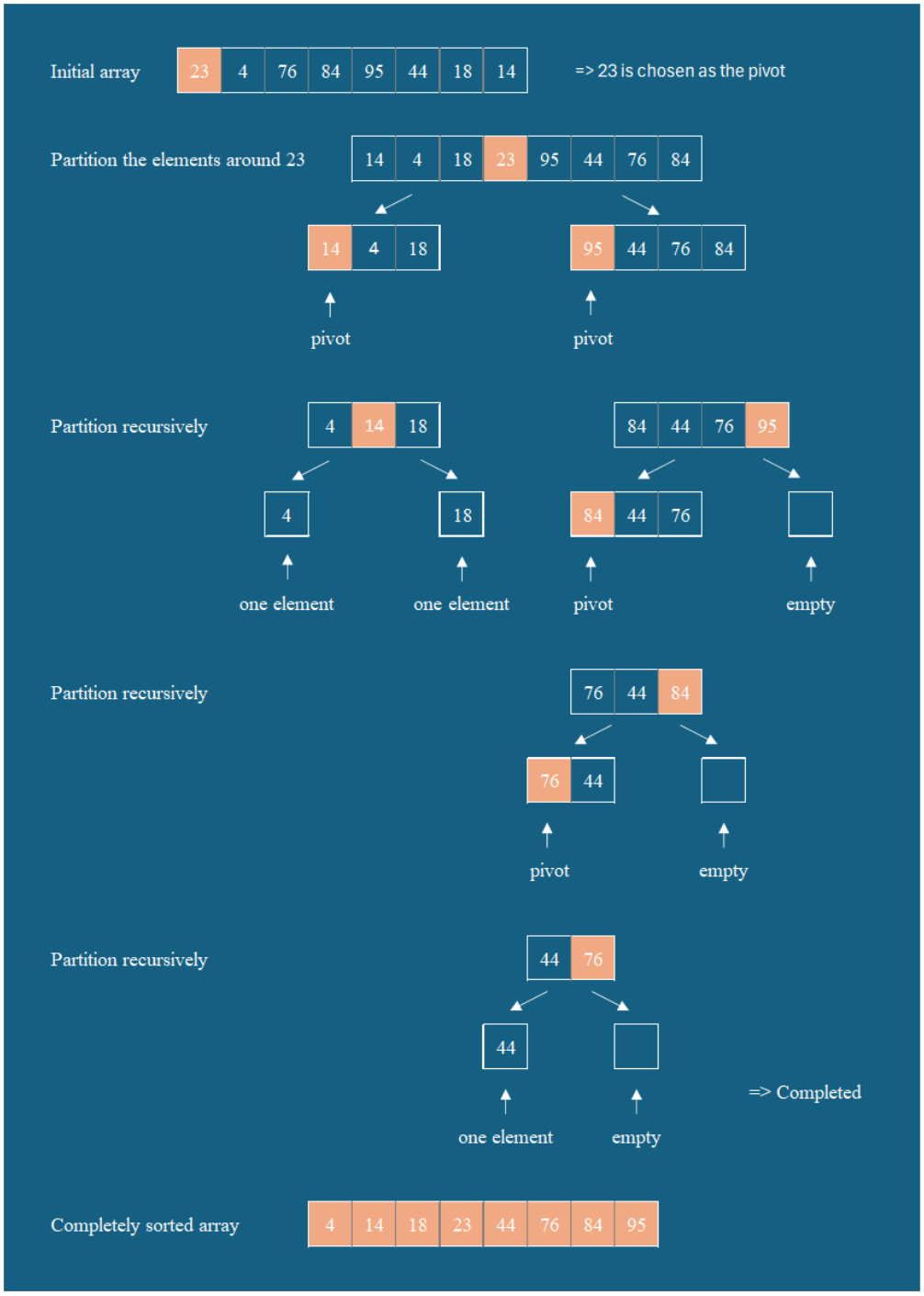


Figure 12: Quick Sort Visualization

Sorted array: [4 14 18 23 44 76 84 95]

3.8.3 Complexity Evaluations

- **Time Complexity:** The time complexity of Quick Sort depends on how the pivot element is selected.
  - *Best case:* This occurs when the pivot always splits the array into two equal or nearly equal sub-arrays. The pivot is the median or close to the median of the array. The partitioning process is performed $log_2(n)$ times, and each time it has a time complexity of $O(n)$ to iterate through the array elements.
    Thus, the best-case complexity of Quick Sort is $O(nlogn)$.
  - *Average case:* Typically, the pivot does not always split the array into two equal sub-arrays but still splits it into two sub-arrays with relatively similar sizes. The number of partitioning steps is approximately $log_2(n)$, and each step has a time complexity of $O(n)$.
    Therefore, the average case complexity is $O(nlogn)$.
  - *Worst case:* This occurs when the pivot continuously splits the array into two sub-arrays with significantly different sizes, often when the pivot is the smallest or largest element in the array. In this case, the number of partitioning steps is $O(n)$, and each step has a time complexity of $O(n)$.
    Thus, the worst-case complexity is $O(n^2)$.
- **Space Complexity:** The space complexity of Quick Sort mainly comes from recursive calls on the stack. Hence, it will be:
  - Best and Average cases: $O(nlogn)$.
  - Worst case: $O(n)$.

3.8.4 Variants/Improvements

- **Hybrid Quick Sort:** Combines Quick Sort with another sorting algorithm (like Insertion Sort) to leverage the advantages of both. Uses Quick Sort for larger arrays and switches to Insertion Sort for smaller arrays. This improves performance for nearly sorted input data.
- **Median-of-Three Quick Sort:** Selects the pivot as the median of the first, middle, and last elements of the array. This reduces the chance of choosing a poor pivot and avoids the worst-case time complexity of $O(n^2)$.
- **Randomized Quick Sort:** Randomly selects the pivot element instead of a fixed one (like the first, middle, or last element). Helps avoid the worst-case scenario of $O(n^2)$.

3.9 Counting Sort

3.9.1 Ideas

We will count the frequency of each element in the array and store these frequencies in a temporary array. Using this data, we sort the array based on the frequency of the elements.

3.9.2 Step-by-step descriptions

The algorithm has eight main steps:

1. Find the smallest and largest elements in the array.
2. Subtract the smallest element from each element in the array.
3. Create a count array `count_array` with `max - min + 1` elements, all initialized to 0.
4. Iterate through the elements in a. For each element $a[i]$, increment `count_array[a[i]]` by 1.
5. Compute the prefix sum of the count array to know the first position that each element appears in the array.
6. Create a temporary array `tmp` with n elements. Place $a[i]$ in its correct position in `tmp` according to the rule `tmp[count_array[a[i]] - 1] = a[i]`.
7. Update the array a based on the array `tmp` by adding `min` to each element, as we initially subtracted `min` from the elements.
8. Return the sorted array.

Assuming we have the array [5 3 -1 4 6 6], n = 6. The Counting Sort algorithm can be illustrated as follows:

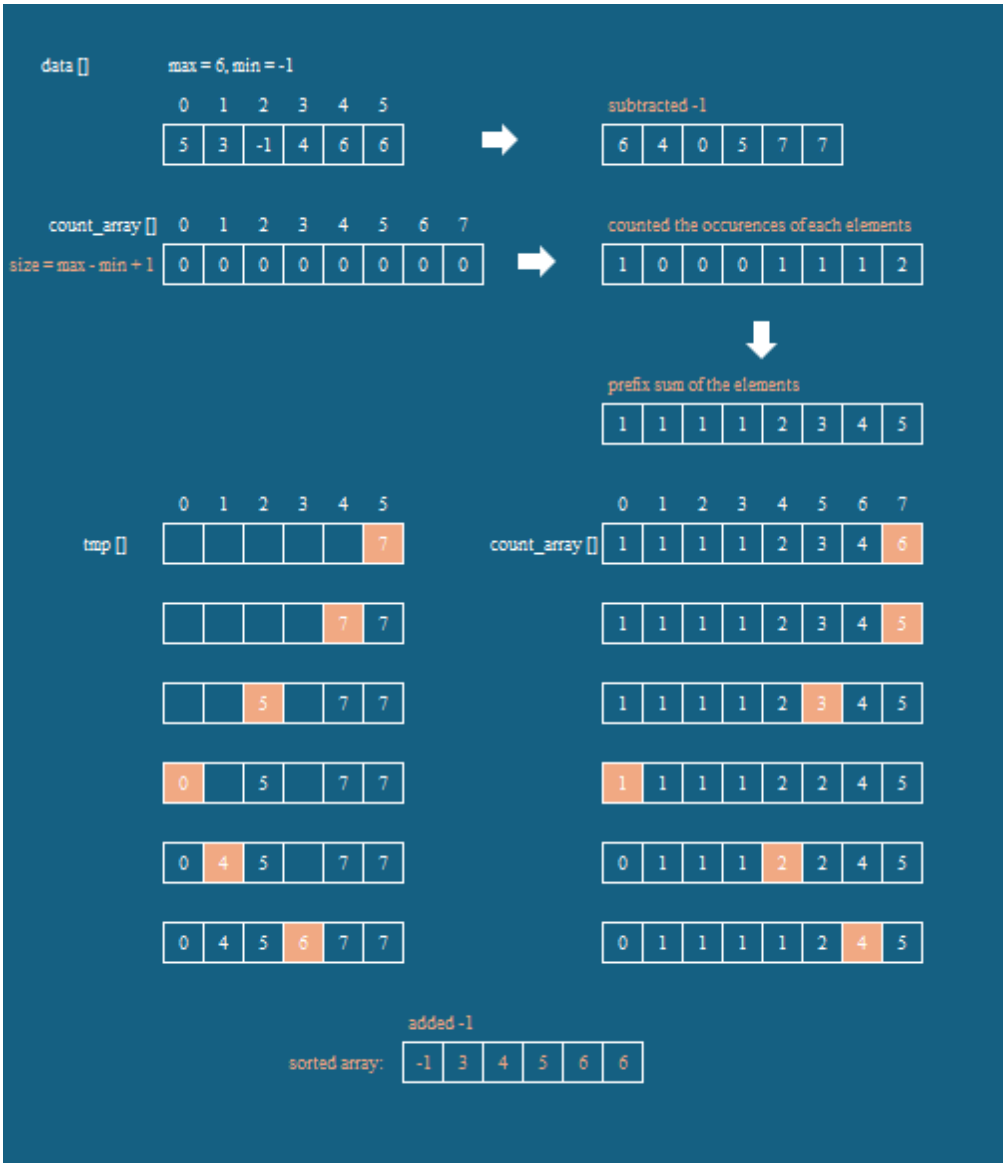


Figure 13: Counting Sort Visualization

Sorted array: [-1 3 4 5 6 6]

3.9.3 Complexity Evaluations

- **Time Complexity:**
  - Best case: $O(n)$ when `max` is less than or equal to the size n.
  - Average case: $O(n + \texttt{max})$ when `max` is not too large compared to the size n.

  - Worst case: $O(n^2)$ when `max` is very large compared to the size n.

- **Space Complexity:** $O(\texttt{max} - \texttt{min})$
  - `max`: The largest element in the array.
  - `min`: The smallest element in the array.

3.10 Radix Sort

3.10.1 Ideas

Radix Sort is a non-comparative sorting algorithm used for sorting integers or strings. This algorithm sorts elements by processing each digit or character one at a time, from the least significant digit or character to the most significant one (or vice versa). Below is a brief description of how Radix Sort works:

1. Determine the maximum number of digits in the dataset.
2. Sort according to each digit (or character) from least significant to most significant.
   - Start with the least significant digit, sort the numbers by this digit.
   - Use a stable sorting algorithm like Counting Sort or Bucket Sort to sort by each digit during this process.
3. Repeat for the next digits: Continue sorting by subsequent digits (tens, hundreds, etc.) until all digits are sorted.

3.10.2 Step-by-step descriptions

Note: In the following description, we use Counting Sort to sort each digit in Radix Sort.

Suppose we have an array as follows:

[65 9 74 96 35 3 89 18 46], n = 9. The Radix Sort algorithm sorts the array as illustrated below:

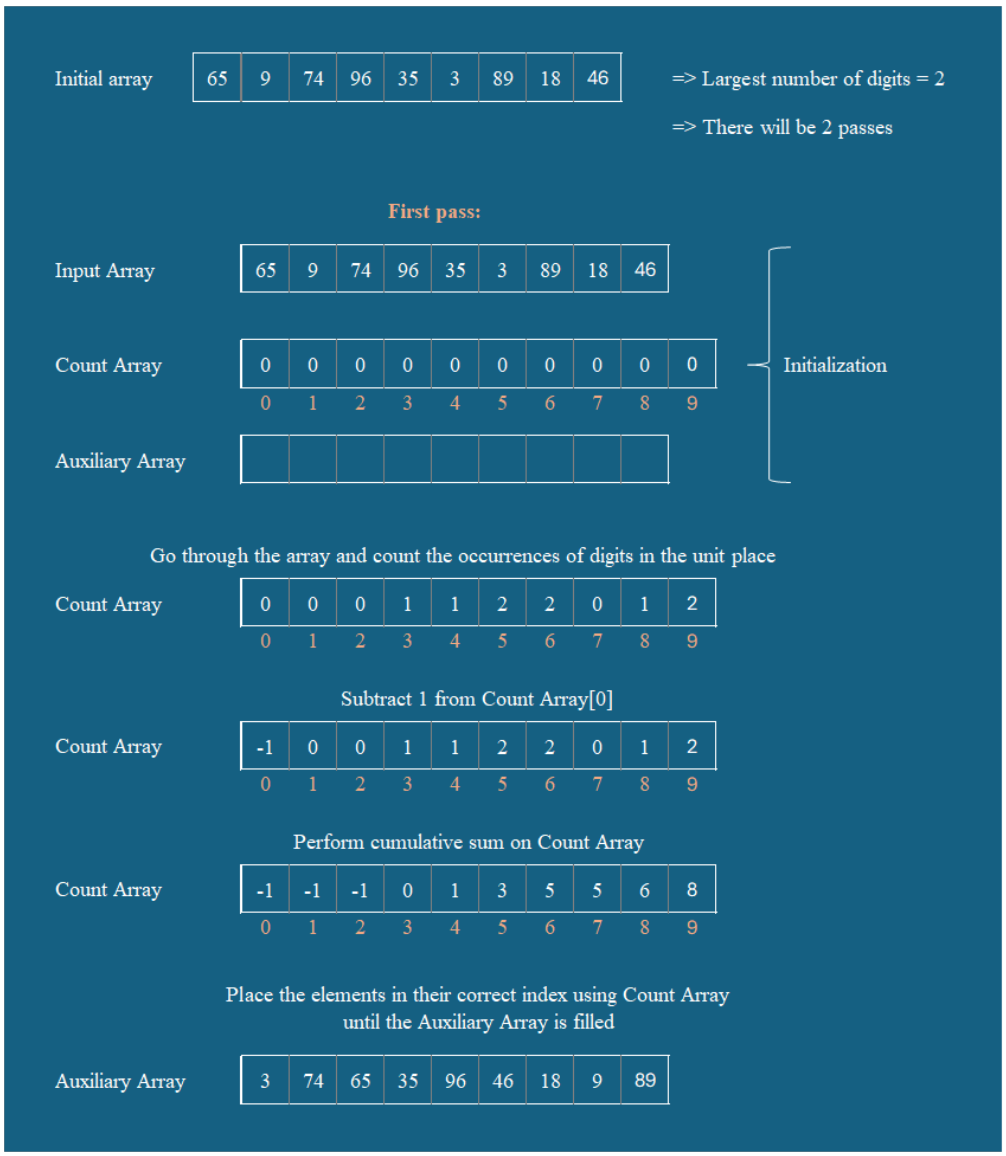


Figure 14: Radix Sort Visualization 1

Figure 15: Radix Sort Visualization 2

Sorted array: [3 9 18 35 46 65 74 89 96]

3.10.3 Complexity Evaluations

- **Time Complexity:**

  Assume that:

  - n is the number of elements in the array
  - d is the largest number of digits

  - k is the range of digits (for example, in the decimal system, k=10 because there are 10 digits from 0 to 9)

  The complexity of Radix Sort is based on two processes:

  1. Processing each digit/character
     There are d times we need to process.
  2. Sorting the digits/characters each time
     - For each digit/character, we need to sort n elements.
     - Counting Sort, commonly used at this step, has a linear complexity $O(n + k)$

  Combining the two processes, the complexity of Radix Sort is: $O(d(n + k))$
  In practice, k is often much smaller than n, so the average complexity is: $O(dn)$

- **Space Complexity:** The space complexity of Radix Sort depends on the memory used during sorting each digit/character. Counting Sort uses additional memory to create the Count Array and Auxiliary Array.

  - *Count Array:* The space complexity is $O(k)$ because it depends on the range of digits/characters.
  - *Auxiliary Array:* The space complexity is $O(n)$ because it depends on the number of elements to be sorted.

  Therefore, the total space complexity of Radix Sort is: $O(n + k)$.

3.10.4 Variants/Improvements

- **LSD (Least Significant Digit) Radix Sort:** Sorts by processing from the least significant digit to the most significant. The algorithm described above falls under this.
- **MSD (Most Significant Digit) Radix Sort:** Sorts by processing from the most significant digit to the least significant. Suitable for sorting strings or when the dataset has a fixed length.
- **Hybrid Radix Sort:** Combines Radix Sort with other sorting algorithms like Quick Sort or Merge Sort to handle certain data groups more efficiently.

3.11 Flash Sort

3.11.1 Ideas

Flash Sort is a linear sorting algorithm that relies on both the distribution of the dataset and comparison operations. Initially, the Flash Sort algorithm calculates the approximate position of each element in the original set relative to the sorted set. Simultaneously, the elements are partitioned and permuted into groups based on the calculated positions. Each group is then sorted using the Insertion Sort algorithm, and the groups are concatenated to form the fully sorted dataset.

3.11.2 Step-by-step descriptions

The algorithm has five main steps:

1. Determine the minimum and maximum values of the input array.
2. Create auxiliary array b to store the count of elements in each group.
   - The size of the auxiliary array (m) is chosen using the formula: $m = \lfloor 0.45n \rfloor$, where n is the size of the input array.
   - The value of m is typically chosen as a certain percentage of n based on the characteristics of the dataset and desired performance. Empirical evidence suggests that $m = \lfloor 0.45n \rfloor$ yields good performance in many real-world cases, so we use this value to implement the algorithm.
3. Determine the number of elements in each group.
   - The number of elements in each group can be determined by iterating through each element of the input array. Element a_i will be assigned to group b_k with $k = \lfloor \frac{(m-1)(a_i - \texttt{min})}{\texttt{max} - \texttt{min}} \rfloor$.
4. Partition elements into corresponding groups.
   - First, use the cumulative sum technique with the auxiliary array $b$to mark the last element of each group using the formula $b_i = b_i + b_{i-1}$. Here, b_i will be the position of the last unpartitioned element of the i-th group.
   - Then, iterate from the beginning of array a and permute elements into their correct groups from the end to the beginning of each group. Each time an element is permuted to a correct group, the displaced element is saved for further permutations in subsequent iterations.
5. Sort elements within each group using the Insertion Sort algorithm.

Assuming we have the array $a = [7\ 4\ 1\ 6\ 2]$, $n = 5$. The Flash Sort algorithm can be illustrated as follows:

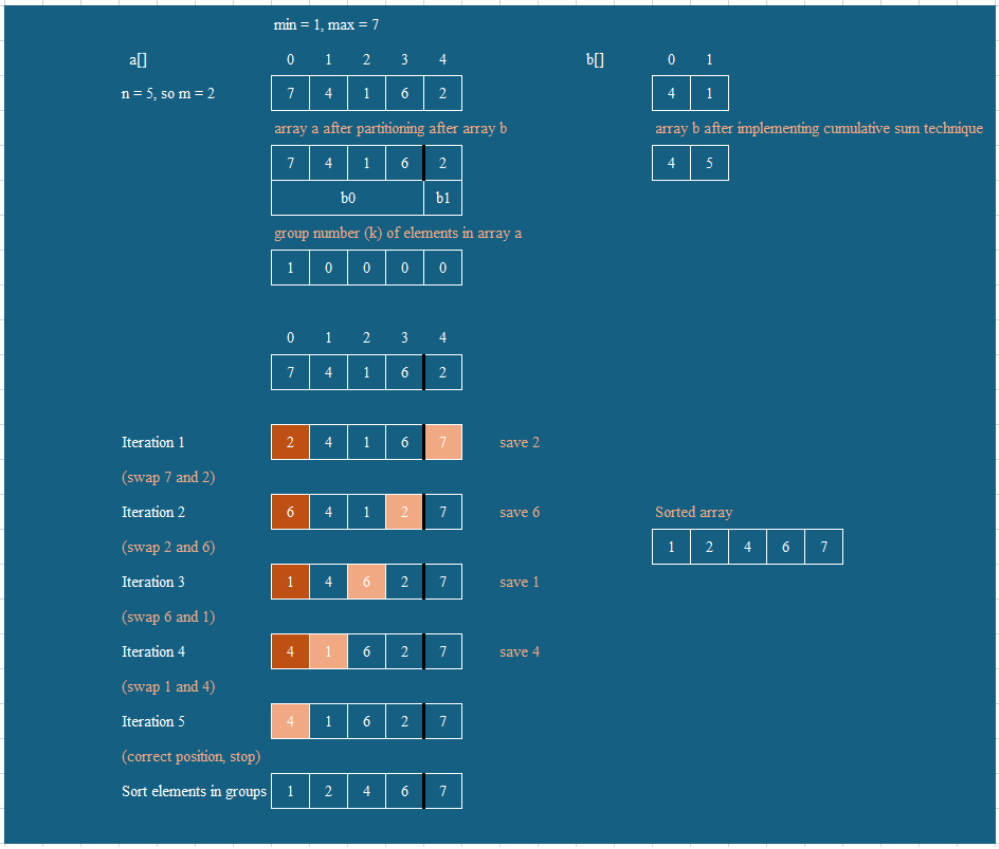


Figure 16: Flash Sort Visualization

Sorted array: [1 2 4 6 7]

3.11.3 Complexity Evaluations

- **Time Complexity:**
  - Partitioning phase: $O(n)$ as each element is examined once.
  - Sorting phase: On average, each group will have $\frac{n}{m}$ elements and the Insertion Sort algorithm has a time complexity of $O(n^2)$. Thus, sorting each group has a complexity of $O\left(\left(\frac{n}{m}\right)^2\right)$
  - There are m groups, so the total sorting complexity is $O\left(m \cdot \left(\frac{n}{m}\right)^2\right) = O\left(\frac{n^2}{m}\right)$. Given $m = 0.45n$, substituting in yields a complexity of $O\left(\frac{n^2}{0.45n}\right) = O(n)$.
  - Moreover, there are certain cases to be considered:
    - Best-case: The input dataset is already sorted, therefore the complexity is $O(n)$.
    - Average-case: Usually occurs when the input dataset is normally distributed, with a complexity of $O(n)$.

  - Worse-case: Occurs when elements within each group are in reverse order after partitioning phase. In this case, if m is chosen inappropriately, the complexity can reach $O(n^2)$. However, with $m = 0.45n$, the complexity is $O(n)$.

- **Space Complexity:** The length of the auxiliary array and input array are m and n, respectively. Hence, the complexity is $O(m + n)$, with $m \leq n$.

4 Experimental results and comments

4.1 Data Tables

4.1.1 Randomized Input

Data Order: **Randomized**						
Data size	10,000		30,000		50,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	63.585	100019998	430.156	900059998	1209.836	2500099998
Insertion Sort	33.183	49885961	312.006	451357782	845.370	1248633742
Bubble Sort	149.630	100009999	1701.177	900029999	5420.365	2500049999
Shaker Sort	136.516	74401341	1588.166	678154081	4520.708	1885332359
Shell Sort	1.019	567338	3.702	2169038	6.826	3941459
Heap Sort	1.456	637815	4.935	2150394	8.877	3772964
Merge Sort	1.652	583585	5.784	1937182	8.592	3382569
Quick Sort	0.899	355594	2.891	1145433	5.209	1945510
Counting Sort	0.149	89995	0.415	269991	0.610	415535
Radix Sort	0.399	140056	1.318	510070	2.211	850070
Flash Sort	1.091	90623	0.732	274113	1.288	424091

Data Order: **Randomized**						
Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	4935.908	10000199998	44368.234	90000599998	123016.711	250000999998
Insertion Sort	3429.945	4986090491	35180.983	44961535887	86771.725	124967753295
Bubble Sort	23595.855	10000099999	216285.720	90000299999	588301.326	250000499999
Shaker Sort	18859.819	7493145171	171499.146	67512148441	469620.872	187352162089
Shell Sort	14.036	8698821	50.016	31597141	85.873	58778577
Heap Sort	19.051	8046534	62.829	26489294	112.779	45967928
Merge Sort	23.347	7165823	58.524	23382209	103.012	40382022
Quick Sort	11.301	4294497	38.979	15111536	72.286	29342115
Counting Sort	1.510	765532	4.108	2165532	6.649	3565532
Radix Sort	4.562	1700070	14.856	5100070	23.898	8500070
Flash Sort	2.358	845042	9.931	2622730	16.974	4495718

4.1.2 Nearly Sorted Input

Data Order: **Nearly Sorted**						
Data size	10,000		30,000		50,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	47.904	100019998	443.005	900059998	1215.877	2500099998
Insertion Sort	34.915	49962847	316.031	450928263	849.960	1246043687
Bubble Sort	34.915	49962847	316.031	450928263	849.960	1246043687
Shaker Sort	139.431	75164761	1538.683	680797559	4525.849	1875674845
Shell Sort	1.045	578187	3.631	2091270	6.558	3709528
Heap Sort	1.451	638068	5.136	2150113	8.676	3772672
Merge Sort	1.642	583694	5.669	1937254	8.313	3383001
Quick Sort	0.872	329166	2.917	1137323	5.131	1925691
Counting Sort	0.209	135515	0.410	275532	0.682	415532
Radix Sort	0.423	170070	1.319	510070	2.124	850070
Flash Sort	0.239	92301	0.706	260461	1.168	449415

Data Order: **Nearly Sorted**						
Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	4877.527	10000199998	43399.939	90000599998	121410.613	250000999998
Insertion Sort	3393.258	4986090491	30820.528	44961535887	86226.453	124967753295
Bubble Sort	23093.274	10000099999	212165.981	90000299999	589331.443	250000499999
Shaker Sort	18545.606	7493145171	168474.391	67512148441	469501.573	187352162089
Shell Sort	14.891	8698821	48.555	31597141	91.786	58778577
Heap Sort	19.575	8046534	63.384	26489294	113.568	45967928
Merge Sort	17.717	7165823	57.213	23382209	99.926	40382022
Quick Sort	11.455	4294497	38.183	15111536	74.772	29342115
Counting Sort	1.260	765532	3.633	2165532	6.040	3565532
Radix Sort	4.213	1700070	14.710	5100070	25.034	8500070
Flash Sort	2.328	845042	10.114	2622730	18.601	4495718

4.1.3 Sorted Input

Data Order: **Sorted**						
Data size	10,000		30,000		50,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	49.209	100019998	440.342	900059998	1239.089	2500099998
Insertion Sort	0.019	29998	0.056	89998	0.096	149998
Bubble Sort	52.832	100009999	469.132	900029999	1301.463	2500049999
Shaker Sort	0.011	20001	0.031	60001	0.051	100001
Shell Sort	0.176	255272	0.599	855751	1.023	1455751
Heap Sort	1.263	670329	4.122	2236648	7.102	3925351
Merge Sort	1.231	475242	4.040	1559914	6.094	2722826
Quick Sort	0.376	244975	1.269	823646	2.315	1467264
Counting Sort	0.112	80004	0.360	240004	0.592	400004
Radix Sort	0.357	140056	1.353	510070	2.166	850070
Flash Sort	0.211	113506	0.631	340506	1.044	567506

Data Order: **Sorted**						
Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	4866.774	10000199998	43689.987	90000599998	121514.458	250000999998
Insertion Sort	0.192	299998	0.573	899998	0.995	1499998
Bubble Sort	5211.308	10000099999	46806.153	90000299999	130547.484	250000499999
Shaker Sort	0.102	200001	0.305	600001	0.559	1000001
Shell Sort	2.263	3167181	8.055	10401464	13.348	17601464
Heap Sort	14.770	8365080	48.248	27413230	85.697	47404886
Merge Sort	11.751	5745658	41.220	18645946	71.502	32017850
Quick Sort	4.738	3134498	15.384	10258249	25.879	17737894
Counting Sort	1.263	800004	3.587	2400004	5.946	4000004
Radix Sort	4.196	1700070	17.993	6000084	29.799	10000084
Flash Sort	2.010	1135006	6.230	3405006	10.225	5675006

4.1.4 Reverse Sorted Input

Data Order: **Reverse Sorted**						
Data size	10,000		30,000		50,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	55.341	100019998	478.261	900059998	1298.631	2500099998
Insertion Sort	67.567	100009999	629.969	900029999	1718.026	2500049999
Bubble Sort	235.625	100009999	2076.347	900029999	5809.991	2500049999
Shaker Sort	231.903	100010001	2123.250	900030001	5882.353	2500050001
Shell Sort	0.306	353923	1.013	1208495	1.615	1960018
Heap Sort	1.192	606771	3.913	2063324	6.935	3612724
Merge Sort	1.227	476441	3.823	1573465	6.002	2733945
Quick Sort	0.439	254764	1.368	855622	2.525	1508351
Counting Sort	0.131	90003	0.380	270003	0.659	450003
Radix Sort	0.343	140056	1.410	510070	2.137	850070
Flash Sort	0.182	97759	0.512	293259	0.884	488759

Data Order: **Reverse Sorted**						
Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	5341.974	10000199998	47664.711	90000599998	132284.278	250000999998
Insertion Sort	6844.257	10000099999	61880.995	90000299999	171915.619	250000499999
Bubble Sort	23255.552	10000099999	207461.866	90000299999	574084.220	250000499999
Shaker Sort	23555.262	10000100001	211041.244	90000300001	585980.487	250000500001
Shell Sort	3.536	4353183	12.009	14215027	20.245	24074412
Heap Sort	14.369	7718943	48.181	25569379	81.142	44483348
Merge Sort	12.056	5767897	42.270	18708313	71.756	32336409
Quick Sort	12.056	5767897	42.270	18708313	71.756	32336409
Counting Sort	1.198	900003	3.761	2700003	6.467	4500003
Radix Sort	4.130	1700070	17.460	6000084	29.453	10000084
Flash Sort	2.025	977509	5.954	2932509	8.889	4887509

4.2 Line Graphs

4.2.1 Randomized Input

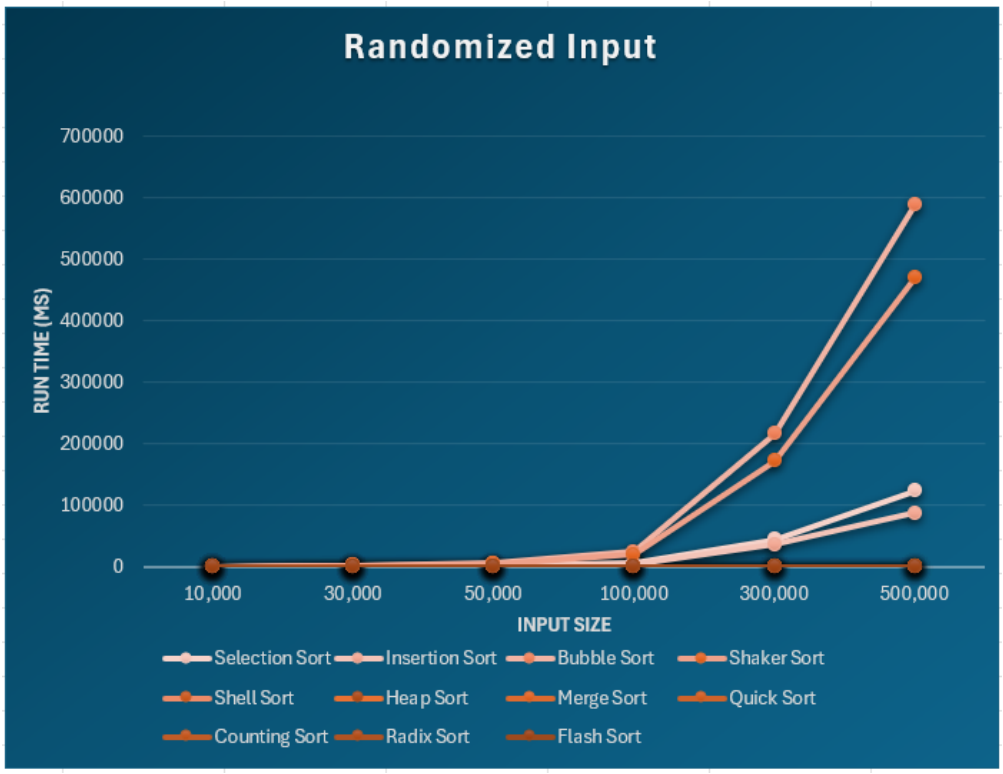


Figure 17: A Line Graph for Visualizing Algorithm Running Times on Randomized Input Data

- **Fastest Algorithms**: Quick Sort, Heap Sort, Merge Sort, Radix Sort, Counting Sort, and Flash Sort show significantly faster runtimes across all input sizes.
- **Slowest Algorithms**: Selection Sort, Insertion Sort, Bubble Sort, and Shaker Sort exhibit much slower performance, especially as input size increases.
- **Observation**: The performance gap between the efficient algorithms (like Quick Sort) and the less efficient ones (like Bubble Sort) widens significantly with larger input sizes.

4.2.2 Nearly Sorted Input

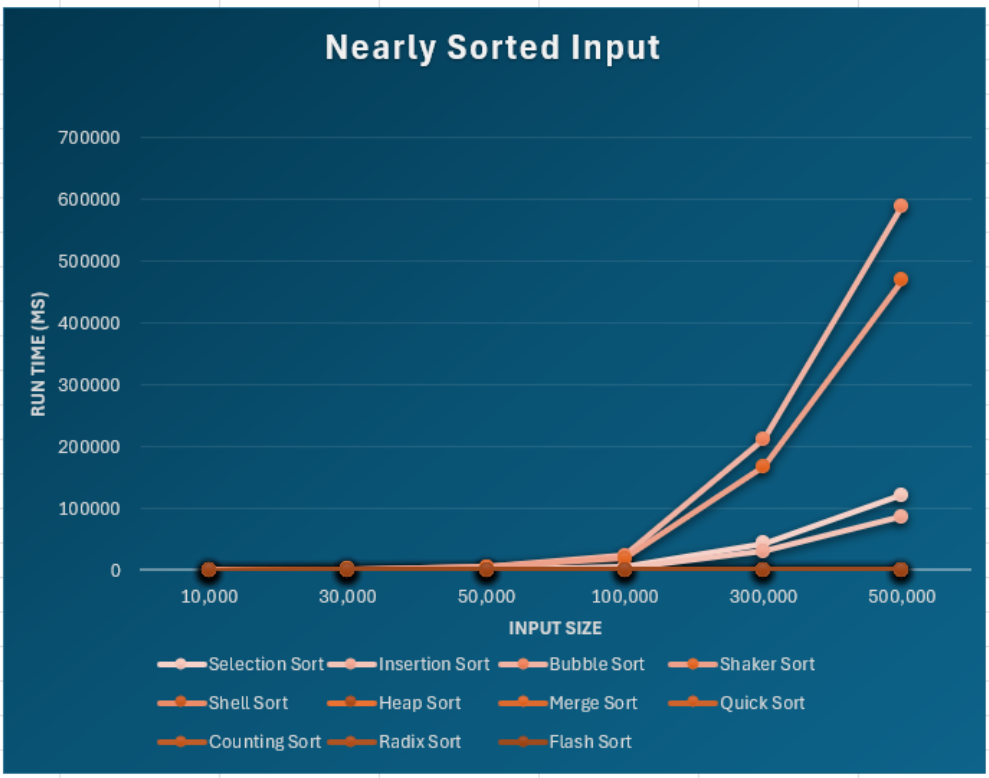


Figure 18: A Line Graph for Visualizing Algorithm Running Times on Nearly Sorted Input Data

- **Fastest Algorithms**: Quick Sort, Heap Sort, Merge Sort, Radix Sort, Counting Sort, Flash Sort, and Insertion Sort perform better compared to randomized input due to the efficiency of Insertion Sort with nearly sorted data.
- **Slowest Algorithms**: Selection Sort, Bubble Sort, and Shaker Sort continue to perform poorly.
- **Observation**: Algorithms that are efficient with randomized data also perform well with nearly sorted data. Insertion Sort shows improvement due to the nature of the input.

4.2.3 Sorted Input

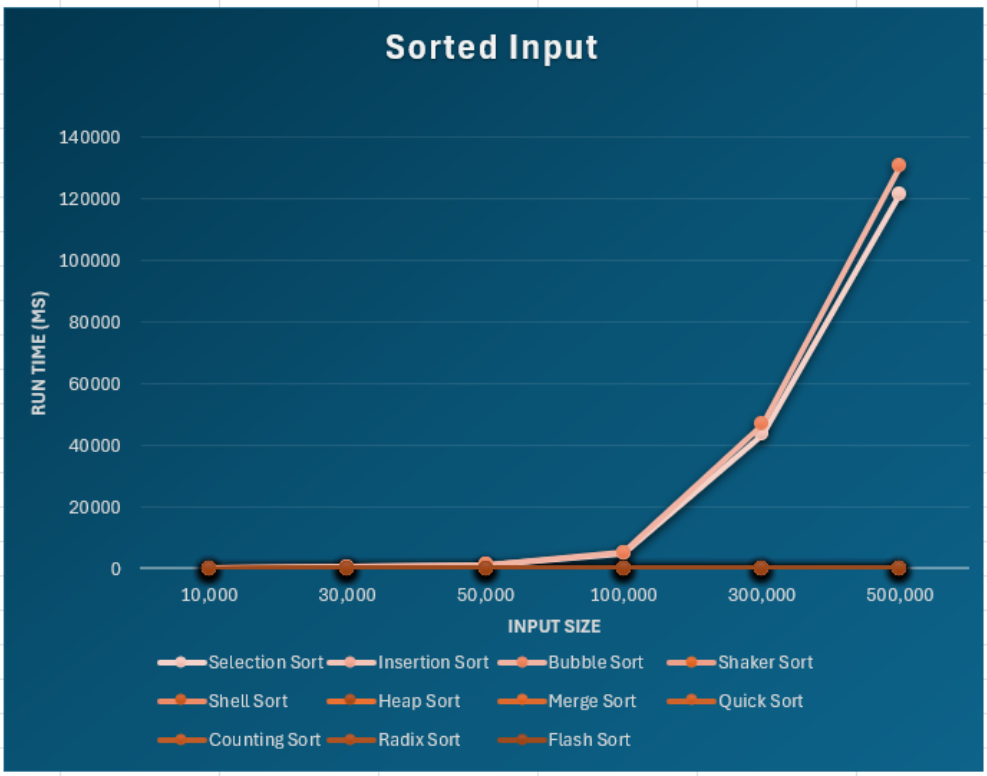


Figure 19: A Line Graph for Visualizing Algorithm Running Times on Sorted Input Data

- **Fastest Algorithms**: Quick Sort, Heap Sort, Merge Sort, Counting Sort, Radix Sort, Flash Sort, and Insertion Sort perform best. Insertion Sort performs exceptionally well with sorted data.
- **Slowest Algorithms**: Bubble Sort, Shaker Sort, and Selection Sort improve but still lag behind the most efficient algorithms.
- **Observation**: Insertion Sort's performance is optimal with sorted data, matching the efficiency of more complex algorithms.

4.2.4 Reverse Sorted Input

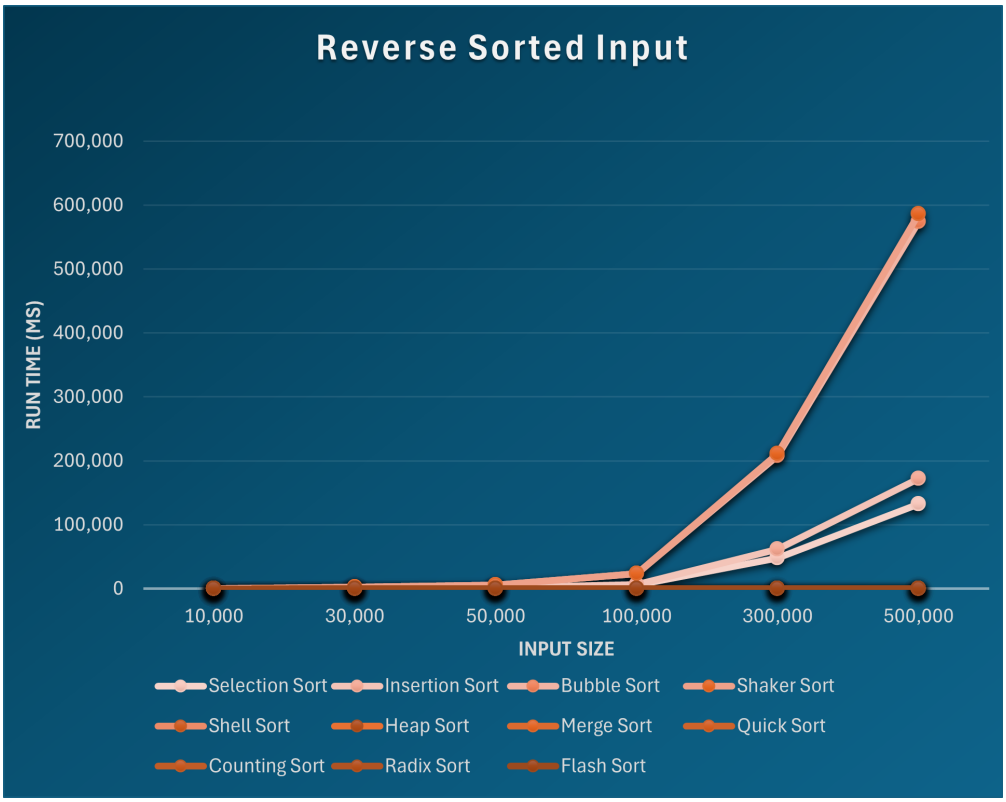


Figure 20: A Line Graph for Visualizing Algorithm Running Times on Reverse Sorted Input Data

- **Fastest Algorithms**: Heap Sort and Quick Sort: These algorithms show the best performance, with the lowest running times across all input sizes. They efficiently handle the reverse sorted input data, maintaining consistent performance. Merge Sort: Performs similarly to Heap Sort and Quick Sort, with relatively low running times, indicating good efficiency on reverse sorted input.
- **Slowest Algorithms**: Selection Sort and Bubble Sort: These algorithms have the highest running times, especially noticeable for larger input sizes. Their performance degrades significantly with increasing input sizes. Insertion Sort: Shows better performance than Selection and Bubble Sort but still has a high running time, which increases rapidly with larger input sizes.
- **Observation**: There is a clear disparity in performance between the efficient algorithms (Heap Sort, Quick Sort, Merge Sort) and the less efficient ones (Selection Sort, Insertion Sort, Bubble Sort). This disparity becomes more pronounced as the input size increases. The performance of the sorting algorithms on reverse sorted input data highlights the importance of input order on sorting efficiency. Algorithms with better average-case and worst-case complexities (like Quick Sort and Heap Sort) maintain their efficiency, while those with poor worst-case complexities (like Bubble Sort and Selection Sort) struggle significantly.

4.3 Bar Charts

4.3.1 Randomized Input

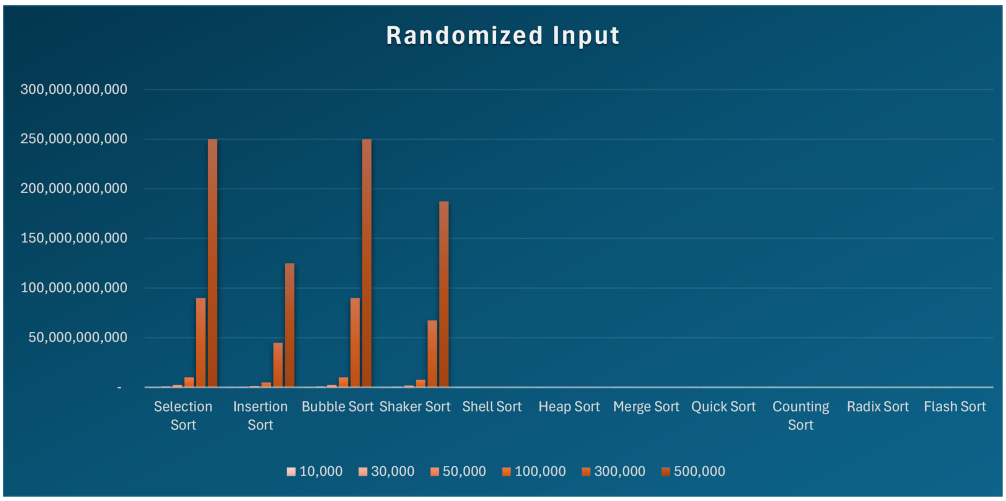


Figure 21: A Bar Chart for Visualizing Algorithms' Number of Comparisons made on Randomized Input Data

- **Algorithms with Fewest Comparisons**: Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort show significantly fewer comparisons across all input sizes.
- **Algorithms with Most Comparisons**: Selection Sort, Insertion Sort, Bubble Sort, and Shaker Sort exhibit a very high number of comparisons, especially as input size increases.
- **Observation**: The performance gap in terms of the number of comparisons is quite noticeable between efficient algorithms (like Quick Sort and Merge Sort) and less efficient ones (like Bubble Sort and Insertion Sort). This highlights the inefficiency of quadratic-time algorithms, which perform many more comparisons as the input size increases.

4.3.2 Nearly Sorted Input

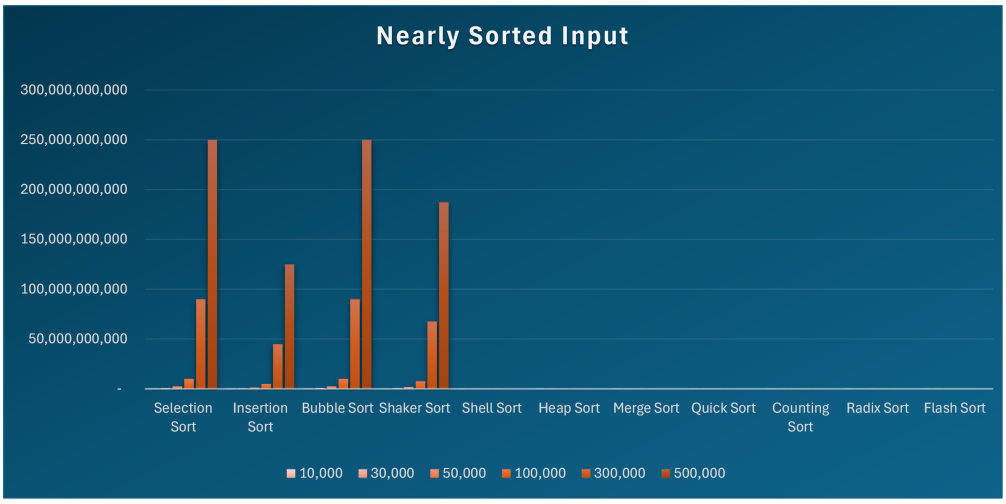


Figure 22: A Bar Chart for Visualizing Algorithms' Number of Comparisons made on Nearly Sorted Input Data

- **Algorithms with Fewest Comparisons**: Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort show significantly fewer comparisons across all input sizes.
- **Algorithms with Most Comparisons**: Selection Sort, Insertion Sort, Bubble Sort, and Shaker Sort exhibit a very high number of comparisons, especially as the input size increases.
- **Observation**: Algorithms that are efficient in terms of comparisons (like Shell Sort and Quick Sort) maintain a low number of comparisons even as input sizes grow. In contrast, the less efficient algorithms (like Bubble Sort and Insertion Sort) have a number of comparisons that scale poorly with input size, indicating their inefficiency even with nearly sorted data.

4.3.3 Sorted Input

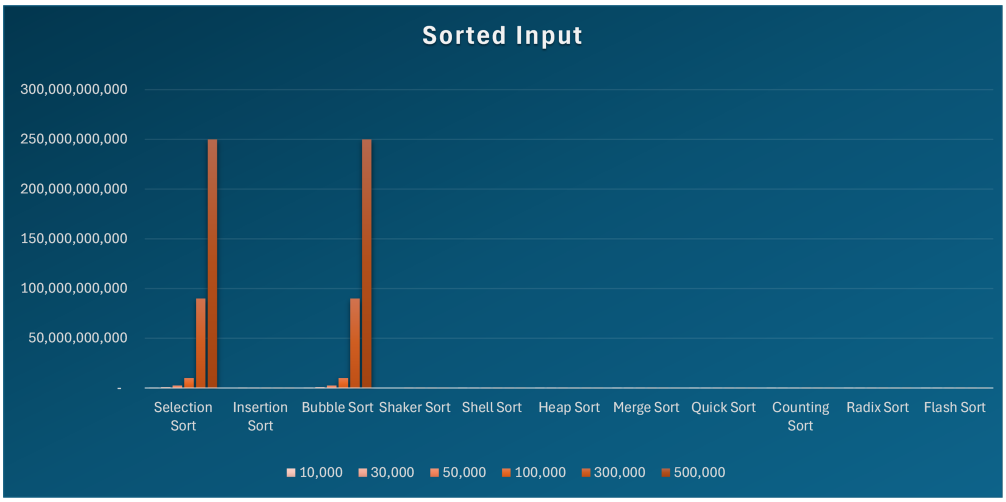


Figure 23: A Bar Chart for Visualizing Algorithms' Number of Comparisons made on Sorted Input Data

- **Algorithms with Fewest Comparisons**: Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort have the fewest comparisons across all input sizes. Insertion Sort also performs exceptionally well with a low number of comparisons for sorted input.
- **Algorithms with Most Comparisons**: Selection Sort and Bubble Sort exhibit a very high number of comparisons, especially as the input size increases. Shaker Sort also shows higher comparisons, though not as extreme as Bubble Sort.
- **Observation**: For sorted inputs, efficient algorithms like Heap Sort, Merge Sort, and Quick Sort maintain low comparison counts, demonstrating their effectiveness in handling already sorted data. In contrast, less efficient algorithms like Bubble Sort and Selection Sort scale very poorly with input size, leading to an exponentially increasing number of comparisons.

4.3.4 Reverse Sorted Input

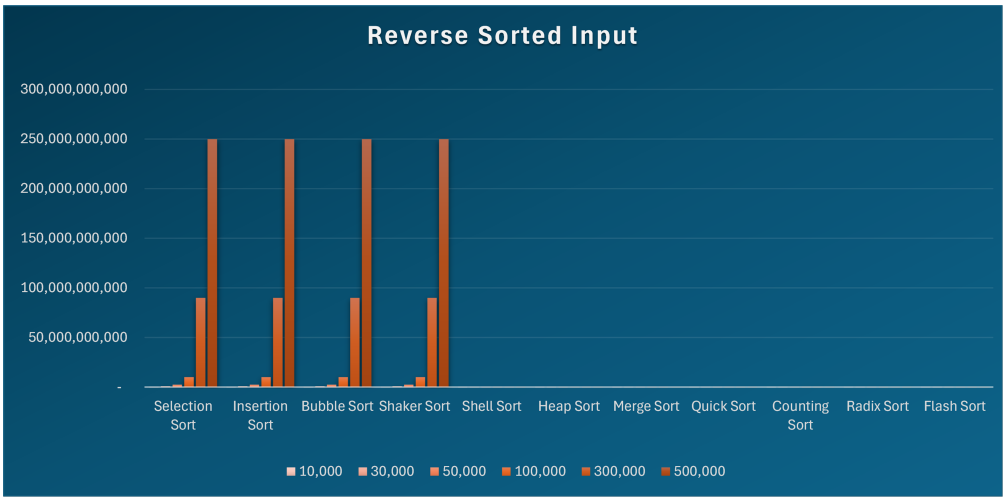


Figure 24: A Bar Chart for Visualizing Algorithms' Number of Comparisons made on Reverse Sorted Input Data

- **Algorithms with Fewest Comparisons**: Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort show significantly fewer comparisons across all input sizes.
- **Algorithms with Most Comparisons**: Selection Sort, Insertion Sort, Bubble Sort, and Shaker Sort exhibit a very high number of comparisons, especially as input size increases.
- **Observation**: The reverse sorted input is particularly challenging for many algorithms. However, the efficient algorithms (like Quick Sort and Merge Sort) still manage to keep the number of comparisons relatively low. In contrast, the less efficient algorithms (like Bubble Sort and Selection Sort) have a drastically higher number of comparisons, further demonstrating their inefficiency.

4.4 Overall Comment on All Data Orders and Sizes

- **Fastest Algorithms Overall**: Quick Sort, Heap Sort, Merge Sort, Radix Sort, Counting Sort, Shell Sort, and Flash Sort consistently show the best performance across all types of input data and sizes.
- **Slowest Algorithms Overall**: Selection Sort, Insertion Sort, Bubble Sort, and Shaker Sort consistently perform poorly, especially with larger input sizes and less favorable data orders (e.g., randomized and reversed).
- **Stable Algorithms**: Merge Sort, Insertion Sort, Bubble Sort, Shaker Sort, Counting Sort, and Radix Sort are stable sorting algorithms that maintain their relative order of equal elements. They show consistent performance across different data orders.

- **Unstable Algorithms**: Quick Sort, Heap Sort, Selection Sort, Shell Sort, and Flash Sort are generally faster but are unstable as they may not preserve the relative order of equal elements.
- **Efficient Group**: Quick Sort, Merge Sort, Heap Sort, Radix Sort, Counting Sort, Shell Sort, Flash Sort
  - Characteristics: Efficient in both average and worst-case scenarios, suitable for large datasets.
- **Inefficient Group**: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort
  - Characteristics: Poor performance with larger datasets and unfavorable data orders, best suited for small or nearly sorted datasets.

4.5 Conclusion

Efficient algorithms like Quick Sort and Merge Sort are preferable for large datasets and varying data orders due to their robustness and speed. In contrast, less efficient algorithms like Bubble Sort should be avoided for large datasets due to their significantly higher runtimes. Stable sorting algorithms are advantageous when the relative order of equal elements matters.

5 Project organization and Programming notes

5.1 Project Organization

Our Source folder consists of the Running files and two main groups of header files: the Standard Group and the Extra Group. This organizational structure aids us in effectively managing code implementation, error detection, and enhances readability. For more details, refer to the section below:

5.1.1 Running files

This includes a program that handles 5 commands required in the assignment and the file DataGenerator.cpp provided by the instructors.

1. <**DataGenerator.cpp**>
   Initializes datasets based on the number of elements and type of data (Random, Sorted, Reverse Sorted, Nearly Sorted).
   - HoanVi(T&a, T&b): A swap function used for initializing nearly sorted arrays.
   - GenerateRandomData(int a[], int n): Initializes a random array.
   - GenerateSortedData(int a[], int n): Initializes a sorted array.

- GenerateNearlySortedData(int a[], int n): Initializes a nearly sorted array.
- GenerateReverseData(int a[], int n): Initializes a reverse sorted array.

2. **<handle_cmd.h>**
   Contains prototypes for sorting functions, constants, array declarations, function pointers, etc., used in <handle_cmd.cpp>.

3. **<handle_cmd.cpp>**

   - GetSortName(const std::string &sortName): Returns the ID number of the requested sorting algorithm, which is then used in executeSort function.
   - executeSort(int a[], int inputSize, int sortFunction): Executes the sorting algorithm based on ID number (int sortFunction).
   - executeSort(int a[], int inputSize, int sortFunction, long long &comparisons): Executes the sorting algorithm and counts the number of comparisons based on ID number (int sortFunction).
   - GetDataType(const std::string &dataType): Returns the index of the requested data type, which is then used to return the corresponding data type name.
   - GetInputData(int inputData[], int &inputSize, const std::string fileName): Get the input data array and its size in the file fileName.
   - PrintRunTime(int a[], int inputSize, int sortFunction, bool isCompare): Measures and prints the runtime of the provided sorting algorithm to the console.
   - PrintComparisons(int a[], int inputSize, int sortFunction, bool isCompare): Measures and prints the number of comparisons made by the provided sorting algorithm to the console.
   - PrintArray(int a[], int inputSize, const std::string fileName): Outputs the provided dataset to the specified filename.
   - CopyArray(int u[], int v[], int inputSize): Copies the array v[] into the array u[].
   - AlgorithmMode(int argc, char **argv): Executes commands containing the -a flag.
   - ComparisonMode(int argc, char **argv): Executes commands containing the -c flag.

4. **<main.cpp>**
   The main function processes command line arguments and determines the mode of operation, either "AlgorithmMode" or "ComparisonMode". It also ensures the arguments are valid and provides error feedback ("Error: Invalid arguments!") if they are not.

5.1.2 Header Files - Standard Group: Standard Sorting Algorithms

There are 11 header files for the required sorting algorithms:

1. **<selection_sort.h>**
   This header file defines two functions:

   - *(1) selectionSort* implements the Selection Sort algorithm.

- *(2) selectionSort_cpr* counts the number of comparisons made during the sorting process.

2. <**insertion_sort.h**>
   This header file defines two functions:
   - *(1) insertionSort* implements the Insertion Sort algorithm.
   - *(2) insertionSort_cpr* counts the number of comparisons made during the sorting process.
3. <**bubble_sort.h**>
   This header file defines two functions:
   - *(1) bubbleSort* implements the Bubble Sort algorithm.
   - *(2) bubbleSort_cpr* counts the number of comparisons made during the sorting process.
4. <**shaker_sort.h**>
   This header file defines two functions:
   - *(1) shakerSort* implements the Shaker Sort algorithm.
   - *(2) shakerSort_cpr* counts the number of comparisons made during the sorting process.
5. <**shell_sort.h**>
   This header file defines two functions:
   - *(1) shellSort* implements the Shell Sort algorithm.
   - *(2) shellSort_cpr* counts the number of comparisons made during the sorting process.
6. <**heap_sort.h**>
   This header file defines six functions:
   - *(1) max_heapify* and *(2) build_max_heap* are used to maintain and build a max heap from an array, respectively.
   - *(3) heapSort* performs the actual sorting.
   - The corresponding functions *(4) max_heapify_cpr*, *(5) build_max_heap_cpr* and *(6) heapSort_cpr* count the number of comparisons made during the sorting process.
7. <**merge_sort.h**>
   This header file defines six functions:
   - *(1) merge* merges two sorted sub-arrays into one sorted array.
   - *(2) mergeSort* recursively divides the array into smaller sub-arrays until they are sorted and then merges them using merge.
   - *(3) mergeSort* (overloaded) is an overloaded function of (2) to keep the parameters uniform throughout our codes.
   - The corresponding functions *(4) merge_cpr*, *(5) mergeSort_cpr* and *(6) mergeSort_cpr* (overloaded) count the number of comparisons made during the sorting process.
8. <**quick_sort.h**>
   This header file defines six functions:

- *(1) partition* partitions the array around a pivot element.
- *(2) quickSort* performs the actual sorting.
- *(3) quickSort* (overloaded) is an overloaded function of (2) to keep the parameters uniform throughout our codes.
- The corresponding functions *(4) partition_ cpr*, *(5) quickSort_ cpr* and *(6) quickSort_ cpr* (overloaded) count the number of comparisons made during the sorting process.

9. <**counting_sort.h**>
   This header file defines two functions:
   - *(1) countingSort* implements the Counting Sort algorithm.
   - *(2) countingSort_ cpr* counts the number of comparisons made during the sorting process.
10. <**radix_sort.h**>
    This header file defines seven functions:
    - *(1) getMax* finds the maximum element in the array, and *(2) getDigit* extracts a specific digit from a number.
    - *(3) countSort* performs Counting Sort based on a specific digit, and *(4) radixSort* performs the Radix Sort algorithm.
    - The corresponding functions *(5) getMax_ cpr*, *(6) countSort_ cpr*, and *(7) radixSort_ cpr* count the number of comparisons made during the sorting process.
11. <**flash_sort.h**>
    This header file defines two functions:
    - *(1) flashSort* implements the Flash Sort algorithm.
    - *(2) flashSort_ cpr* counts the number of comparisons made during the sorting process.

5.1.3 Header files - Extra Group: Variants/Improved Versions

Besides the standard sorting algorithms, we have developed 6 variants and improved versions for some of them:

1. <**selection_sort_improved.h**>
   This header file defines two functions:
   - *(1) selectionSortImproved* implements the improved Selection Sort algorithm, which minimizes the number of swaps by moving the minimum element to the beginning and the maximum element to the end of the array in each pass.
   - *(2) selectionSortImproved_ cpr* counts the number of comparisons made during the sorting process.
2. <**insertion_sort_improved.h**>
   This header file defines two functions:

- *(1) binaryInsertionSort* implements the Binary Insertion Sort algorithm, which improves the efficiency of the standard insertion sort by using binary search to find the correct position of the element.
- *(2) binaryInsertionSort_ cpr* counts the number of comparisons made during the sorting process.

3. **<bubble_sort_improved.h>**
   This header file defines four functions:

   - *(1) bubbleSortImproved* implements the improved Bubble Sort algorithm, which includes a flag to detect if the array is already sorted.
   - *(2) combSort* implements the Comb Sort algorithm, which uses a shrinking gap to compare and swap elements, then finishes with a final pass similar to Bubble Sort.
   - *(3) bubbleSortImproved_ cpr* and *(4) combSort_ cpr* count the number of comparisons made during the sorting process.

4. **<merge_sort_improved.h>**
   This header file defines six functions:

   - *(1) insertionSort* sorts the sub-arrays using Insertion Sort.
   - *(2) merge_ improved* merges two sorted sub-arrays into one sorted array.
   - *(3) mergeSortImproved* implements the improved version of Merge Sort, specifically Timsort.
   - The corresponding functions *(4) insertionSort_ cpr*, *(5) merge_ improved_ cpr* and *(6) mergeSortImproved_ cpr* count the number of comparisons made during the sorting process.

5. **<quick_sort_improved.h>**
   This header file defines eight functions:

   - *(1) medianOfThree* finds the median of three elements (first, middle, and last) and uses it as the pivot.
   - *(2) partitionImproved* partitions the array around the pivot selected by *medianOfThree*.
   - *(3) quickSortImproved* implements the improved version of Quick Sort, specifically Median of Three Quicksort.
   - *(4) quickSortImproved* (overloaded) is an overloaded function of (3) to keep the parameters uniform throughout our codes.
   - The corresponding functions *(5) medianOfThree_ cpr*, *(6) partitionImproved_ cpr*, *(7) quickSortImproved_ cpr* and *(8) quickSortImproved_ cpr* (overloaded) count the number of comparisons made during the sorting process.

6. **<radix_sort_improved.h>**
   This header file defines seven functions:

   - *(1) getMax_ improved* finds the maximum element in the array, and *(2) getByte* extracts a specific byte from a given number based on the byte index.
   - *(3) countingSortImproved* performs counting sort on an array based on a specific byte index, and *(4) radixSortImproved* implements the improved version of Radix Sort.

- The corresponding functions *(5) getMax_ improved_ cpr*, *(6) countingSortImproved_ cpr*, and *(7) radixSortImproved_ cpr* count the number of comparisons made during the sorting process.

5.2 Libraries

5.2.1 <chrono>

This library is used for timing and clock operations. It provides classes for manipulating time, such as `std::chrono::high_resolution_clock`.

5.2.2 <string>

This library is used for working with strings. It provides classes like `std::string` for manipulating strings.

5.2.3 <iostream>

This library is used for input/output operations. It provides classes like `std::cin` and `std::cout`.

5.2.4 <fstream>

This library is used for input/output operations. It provides classes like `std::ifstream` and `std::ofstream` for reading and writing files.

5.2.5 <iomanip>

This library is used for input/output manipulation. It provides classes like `std::setprecision` and `std::fixed` for setting the precision of output.

6 List of references

Our project would not be complete without acknowledging the contributions from various sources. Here we express our gratitude and give credit to their valuable work:

1. Data Structures and Algorithms in C++ - Adam Drozdek
    - 9.3.3 Quicksort
    - 9.3.4 Mergesort

- 9.3.5 Radix Sort
- Implementation of Shell Sort using Knuth sequence
- Time complexity calculation of Insertion and Bubble Sort algorithms

2. GeeksforGeeks
   - Radix Sort Algorithm
   - Time complexity calculation of Shaker Sort algorithm
3. Master theorem
   - Youtube video
   - Youtube video
4. The Flashsort1 Algorithm: Dr Dobb's (drdobbs.com)
5. Time complexity calculation of Shell Sort algorithm: An Analysis of Shellsort and Related Algorithms by Robert Sedgewick