VNUHCM - UNIVERSITY OF SCIENCE

FALCULTY OF INFORMATION TECHNOLOGY

PROJECT
DATA STRUCTURES AND ALGORITHMS

TOPIC:

SORTING

Supervisor: TRAN THI THAO NHI
BUI HUY THONG

Students:

Name	*ID*	*Class*
AN TIEN NGUYEN AN	23127148	23CLC01
NGUYEN TUAN ANH	23127152	23CLC01
TRAM HUU NHAN	23127442	23CLC01
HO THAI SON	23127469	23CLC01

Ho Chi Minh City, 2024

Contents

1 Introduction

1.1 Project Topic

This project researches 11 sorting algorithms introduced in the curriculum of CSC10004 – Data Structures and Algorithms, guided by the lecturers and practical instructors. The algorithms include: **Selection sort**, **Insertion sort**, **Shell sort**, **Bubble sort**, **Shaker sort**, **Heap sort**, **Merge sort**, **Quick sort**, **Counting sort**, **Radix sort**, **Flash sort**. These sorting algorithms are applied to arrays of different sizes and properties to evaluate execution time, algorithm complexity, and flexibility in various cases.

1.2 Input Data

This project involves several common symbols repeated throughout the report. Below are the annotations:

- a: input data array.
- n: size of the input data array (specifically, arrays of 10,000; 30,000; 50,000; 100,000; 300,000; and 500,000 elements).
- Input order: randomized, nearly sorted, sorted, and reverse sorted data.

1.3 Project Objectives

- Present the sorting algorithms (Idea, descriptions, and implementations).
- Analyze and evaluate the complexity of the sorting algorithms.
- Write a program to measure execution time (in seconds, double type) and the number of comparisons according to 5 commands required by the instructor.
- Draw line charts for execution time and bar charts for the number of comparisons of the sorting algorithms to provide an overall assessment of the strengths and weaknesses of each algorithm.

1.4 Research Methods

- Study the algorithms (Idea, descriptions, variations, and improvements).
- Analyze the complexity of the algorithms (time and space complexity).

1.5 Results Achieved

- Successfully researched and implemented 11 sorting algorithms.
- Completed the program to execute 5 commands as required by the instructor.
- Completed the introduction, analysis, and evaluation of the complexity of the sorting algorithms.
- Completed drawing and commenting on the charts representing each algorithm.

1.6 Program Implementation

To ensure accuracy and objectivity when comparing sorting algorithms, the algorithm execution will be based on the following common factors:

1.7 Computer Configuration

- Processor: AMD Ryzen 7 6800H with Radeon Graphics 3201 MHz, 8 Core(s), 16 Logical Processor(s).
- Memory: Installed Physical Memory (RAM) 16.0 GB.
- Operating System: Microsoft Windows 11 Home Single Language.

1.8 Compilation Configuration

The program is compiled using the GNU C++ Compiler. Therefore, other compilers like Microsoft Visual C++ Compiler (MSVC) or Clang may not compile the source code and generate executable (.exe) files to activate the source code. Specifically, the source code is written in C++11 with the following compilation command:

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

Additionally, to execute the program after downloading the compressed file of the project, extract the file, compile the program, and run `main.exe` using the terminal with the commands specified in the assignment.

1.9 Notes

- The sorting algorithms mentioned above are used to sort the elements of the array in ascending order.
- All execution times in the charts and tables are calculated in seconds.
- The algorithms are run on the same computer configuration and are single-threaded.
- To delve deeper into the algorithms, the group has included some improved or variant versions of the original sorting algorithms. These variations and improvements (if any) are presented and implemented in the group's source code.

2 Algorithm Representation

2.1 Insertion sort

2.1.1 Idea

The **Insertion sort** algorithm divides the array into two parts similar to the **Selection sort** algorithm, the sorted part and the unsorted part. The algorithm iterates through $n-1$ stages, at each stage, the algorithm takes an element from the unsorted part and finds the "appropriate" position for this element in the sorted part to insert it while maintaining the sorted order of the array.

2.1.2 Algorithm Description

Suppose we have an unsorted array as follows: [7, 4, 2, 1, 5]

Step 1: Iterate through each element in the array, starting from the second element (position 1). Call the current element *key*.

Step 2: Compare *key* with the elements to its left (iterate from right to left) to find the appropriate position to insert *key* into the sorted array. A position is considered appropriate when it is the first position where the element *key* is smaller than the element after it, in other words, find the largest j such that $a_j < key$.

Step 3: While finding the position, simultaneously move the elements greater than *key* one position to the right to create space for *key* to be inserted. Repeat this step until *key* is inserted at the correct position.

- Array after completing step 3 (iteration 1): [4, 7, 2, 1, 5]
- Array after completing step 3 (iteration 2): [2, 4, 7, 1, 5]
- Array after completing step 4 (iteration 3): [1, 2, 4, 7, 5]

Step 4: Repeat steps 1 to 3 until all elements in the original array have been iterated through and inserted into the appropriate positions in the sorted array.

Result: Array after completing all steps: [1, 2, 4, 5, 7]

2.1.3 Complexity Analysis

2.1.3.1 Time Complexity

The complexity of the **Insertion sort** algorithm depends significantly on the initial arrangement of the elements in the array. Let $T(n)$ be the execution time of the **Insertion sort** algorithm for an array of length n.

- **Best Case:** The best case is when the data are already in order. Only one comparison is made for each position i, so there are $n-1$ comparisons, which is $O(n)$, and $2(n-1)$ moves, all of them redundant.

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

- **Average Case:** This case often occurs when the elements in the array are distributed randomly: neither increasing nor decreasing. The time complexity in this case is $T(n) = \frac{n^2+n}{4} + \frac{n}{2} - ln(n) - 0.5772$, which belongs to $O(n^2)$.
- **Worst Case:** The worst case is when the data are in reverse order. In this case, for each i, the item `data[i]` is less than every item `data[0], ..., data[i-1]`, and each of them is moved by one position. For each iteration i of the outer `for` loop, there are i comparisons, and the total number of comparisons for all iterations of this loop is

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

The number of times the assignment in the inner `for` loop is executed can be computed using the same formula. The number of times `tmp` is loaded and unloaded in the outer `for` loop is added to that, resulting in the total number of moves:

$$\frac{n(n-1)}{2} + 2(n-1) = \frac{n^2+3n-4}{2} = O(n^2)$$

2.1.3.2 Space Complexity

The **Insertion sort** algorithm performs in-place swaps of two elements in the array. So, the space complexity of **Insertion sort** would be $O(1)$ (in-place algorithm).

2.1.3.3 Actual Running Time

According to data for $10,000; 30,000; 50,000; 100,000; 300,000; 500,000$ elements.

n	10,000	30,000	50,000	100,000	300,000	500,000
Already Sorted	0	0	0	0	0	0.001001
Nearly Sorted	0	0.001101	0	0	0.001334	0.002004
Reversed Data	0.069177	0.548178	1.71796	6.01659	52.9224	153.364
Random Data	0.03524	0.287411	0.783311	2.96092	25.3842	76.0444

Table 1: Summary of **Insertion sort** result

2.1.4 Improvement and Optimization

According to the initial idea of the algorithm, we divide the array into 2 parts: the unsorted part and the sorted part. We sequentially take each element from the unsorted part and find its correct position in the sorted part using Linear Search with a complexity of $O(n)$. This process is slow, so we take advantage of the sorted part of the array to use Binary Search with a complexity of $O(\log(n))$, which is much faster than linear search. Therefore, the algorithm is named **Binary Insertion sort** [2].

n	10,000	30,000	50,000	100,000	300,000	500,000
Insertion sort	0.03524	0.287411	0.783311	2.96092	25.3842	76.0444
Binary Insertion sort	0.028216	0.229755	0.610769	2.39891	20.6278	56.3191

Table 2: **Insertion sort** and binary **Insertion sort**

Note: Similar to the **Insertion sort** algorithm, the Binary **Insertion sort** algorithm highly depends on the distribution of the input data. When the input data is already sorted or nearly sorted, the algorithm runs very fast, $O(n\log(n))$, but if the data is random or in reverse order, the algorithm will run slow with the worst-case complexity of $O(n^2)$. However, the experimental results still show that Binary **Insertion sort** runs faster than **Insertion sort** in all cases.

2.2 Counting sort

2.2.1 Idea

The **Counting sort** algorithm [6] is an efficient sorting algorithm used to sort elements from an array with values within a specific range (usually non-negative and limited). **Counting sort** does not use direct comparisons between elements like algorithms such as **Quick sort** or **Merge sort**. Instead, it uses an auxiliary array to count the occurrences of each value in the original array, then uses this count array to place the elements in sorted order.

2.2.2 Algorithm Description

Assume we have the following initial array with non-negative integers: [4, 2, 1, 5, 3, 3, 1]

Step 1: Create a count array (auxiliary array) to count the occurrences frequency of each value in the initial array:

Count array: [0, 2, 1, 2, 1, 1]

In the count array, the element at index i is the number of times the value i appears in the initial array.

Step 2: Calculate the cumulative sum of the count array to determine the first position of each value in the result array:

Count array after cumulative sum: [0, 2, 3, 5, 6, 7]

In the count array after the cumulative sum, the element at index i is the first position of the value i in the result array.

Step 3: Scan the initial array again and place each element at the appropriate position in the result array based on the count array:

Result: Array after the scan: [1, 1, 2, 3, 3, 4, 5]

2.2.3 Complexity Analysis

2.2.3.1 Time Complexity

Fundamentally, **Counting sort** uses independent loops to solve the problem. We can divide the process into several loops and calculate the time complexity for each step as follows:

- Loop 1: Find the largest value in the array and assign it to the variable MAX. This loop takes n steps.
- Loop 2: Count the occurrences of each value in the array and store it in the count array. This loop takes n steps.
- Loop 3: Calculate the cumulative sum of the count array. This loop takes MAX steps.
- Loop 4: Sort the elements into the auxiliary array S using the assignment: $S[count[A[i]] - 1] = A[i]$. This loop takes n steps.
- Loop 5: Copy the auxiliary array back to the original array, this loop also takes n steps. Thus, the time complexity of the **Counting sort** algorithm is $O(size + MAX)$, where MAX is the largest value in the array.

In conclusion, time complexity of the **Counting sort** algorithm is $O(size + MAX)$ with MAX is the largest value in the array.

- **Best case:** When MAX is small or equal to the size of the array n, then $O(n + MAX)$ is equivalent to $O(n)$. In this case, the algorithm's complexity is linear $O(n)$.
- **Average case:** For any values of MAX not too large compared to n, the complexity of the algorithm is considered to be $O(n + k)$.
- **Worst case:** The worst case occurs when the array is skewed and the difference between the smallest and largest elements is large. It has been proven that the complexity of the **Counting sort** algorithm is $O(n+MAX)$, where MAX has a size equivalent to n^2, making the complexity $O(n+n^2)$, equivalent to $O(n^2)$. Thus, as the value of MAX increases significantly, it negatively affects the complexity of the algorithm.

2.2.3.2 Space Complexity

$O(MAX)$, where MAX is the largest element in the array.

2.2.3.3 Actual Running Time

According to the data: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000.

n	10,000	30,000	50,000	100,000	300,000	500,000
Already Sorted	0.001393	0.001002	0.001001	0.002014	0.00511	0.01004
Nearly Sorted	0	0	0.001098	0.002325	0.005866	0.009844
Reversed Data	0	0.001001	0.00101	0.001376	0.006144	0.009526
Random Data	0.001017	0	0.001002	0.001003	0.004412	0.00752

Table 3: Summary of **Counting sort**

2.2.4 Improvement and Optimization

Counting sort is only suitable for non-negative number arrays and the minimum value should not be too large compared to 0. To overcome this, we need to find the smallest value in the array (MIN) and the largest value in the array (MAX), then subtract the smallest value just found from all values in the array. After the array has been sorted, subtract MIN from all numbers in the array to return to the original array. This way, we can use **Counting sort** for arrays that have negative numbers or have a very large minimum value. The main idea of the method is to bring the negative range to $[0, MAX+|MIN|]$ or very large positive ranges (should be less than 10^6) to $[0, MAX - MIN]$.

In the source code of my team project, we have done both types of **Counting sort** implementation.

2.3 Merge sort

2.3.1 Idea

Merge sort [6] is a divide and conquer algorithm. The key process in mergesort is merging sorted halves of an array into one sorted array. However, these halves have to be sorted first, which is accomplished by merging the already sorted halves of these halves. This process of dividing arrays into two halves stops when the array has fewer than two elements. The algorithm is recursive in nature.

2.3.2 Algorithm Description

Suppose we have an unsorted array as follows: $[11, 17, 5, 1, 13, 9, 10, 7]$.

Step 1: Split the array into halves (two subarrays) evenly.

- Split into: [11, 17, 5, 1] and [13, 9, 10, 7]

Step 2: Apply the divide step to each subarray, continue splitting each subarray into smaller subarrays until each subarray contains only one element.

- First split: [11, 17] and [5, 1] and [13, 9] and [10, 7]
- Second split: [11] [17] [5] [1] [13] [9] [10] [7]

Step 3: Merge the subarrays to form sorted arrays. Specifically, compare the first elements of the two subarrays, pick the smaller element and put it into the result array. Continue comparing the next elements of the subarrays and put them into the result array in sorted order. Repeat the process until no elements are left in any subarray.

- Merge 1: [11, 17] and [1, 5] and [9, 13] and [7, 10]
- Merge 2: [1, 5, 11, 17] and [7, 9, 10, 13]
- Merge 3: [1, 5, 7, 9, 10, 11, 13, 17]

Result: [1, 5, 7, 9, 10, 11, 13, 17]

2.3.3 Complexity Analysis

2.3.3.1 Time Complexity

The time complexity of the **Merge sort** algorithm is $O(n \log n)$. This can be demonstrated as follows:

Let $T(n)$ be the running time of the **Merge sort** algorithm for an array of length n. We have the recurrence:

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + n$$

Solving the recurrence, we get:

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + n = 4 \times T\left(\frac{n}{4}\right) + 2n = 2^k \times T\left(\frac{n}{2^k}\right) + kn$$

When $k = \log n$, $T\left(\frac{n}{2^k}\right) = T(1) = 1$:

$$T(n) = n \log n + n = O(n \log n)$$

- **Best Case**: This occurs when the initial array is already sorted. The time complexity in this case is $O(n \log n)$.
- **Average Case**: This usually occurs when the elements in the array are randomly distributed. The time complexity in this case is $O(n \log n)$.
- **Worst Case**: This occurs when the initial array is sorted in reverse order. The time complexity in this case is $O(n \log n)$.

2.3.3.2 Space Complexity

The **Merge sort** algorithm requires additional space to store the merged subarrays, the largest subarray being of length n and the stack depth being $\log n$. Thus, the space complexity of the **Merge sort** algorithm is $O(n + \log n) = O(n)$.

2.3.3.3 Actual Running Time

According to the data: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000.

n	10,000	30,000	50,000	100,000	300,000	500,000
Already Sorted	0.001251	0.004121	0.005881	0.011925	0.033282	0.054641
Nearly Sorted	0.001002	0.003423	0.005494	0.011333	0.033173	0.054205
Reversed Data	0.001311	0.004118	0.005414	0.011029	0.034379	0.054184
Random Data	0.002005	0.00504	0.00902	0.018035	0.051827	0.085167

Table 4: Summary of **Merge sort**

2.4 Selection sort

2.4.1 Idea

Selection sort is a simple sorting algorithm. The algorithm will go through $n-1$ periods; each period will find the smallest (or largest) element starting with the index ith, then selecting and swapping it to the ith element (a sorted portion of the array). After that, the length of the sorted array will be increased by 1 (corresponding to the size of the unsorted array will be decreased by 1).

2.4.2 Algorithm Description

Assume that we have an unsorted array: [68, 23, 15, 20, 12]

Step 1: Finding the smallest element from the initial array

- Choose the first element ($a[0] = 68$) as the smallest element and assign `min_index` $= 0$ (a variable that stores the position of the smallest element).

- Iteratively compare with another element in the array to find the smallest element.
- Compare the current element with the smallest element, and if the current element is smaller, assign `min_index` to the position of the current element. In this problem, we find `min_index` = 4 (equivalent $a[4] = 12$).

Step 2: Swap the smallest element found in step 1 with the first element in the array.

Step 3: Continue finding and swapping the smallest element in the array (on the unsorted portion). Choose the unsorted portion of the array starting with the 2nd element and follow steps 1 and 2 for this array. Iterate until the array is sorted.

Array after steps 1 and 2:

- 1st period: [12, 23, 15, 20, 68]
- 2nd period: [12, 15, 23, 20, 68]
- 3rd period: [12, 15, 20, 23, 68]
- 4th period: [12, 15, 20, 23, 68]
- 5th period: [12, 15, 20, 23, 68]

Result: Sorted array [12, 15, 20, 23, 68]

2.4.3 Complexity Analysis

2.4.3.1 Time complexity

The time complexity of the **Selection sort** algorithm is $O(n^2)$ in all cases. We can prove this:

The **Selection sort** algorithm uses two nested loops to sort elements. The outer loop runs from $i = 0$ to $i = n - 2$. For each i, the inner loop runs from $j = i + 1$ to $j = n - 1$. The basic operation of this algorithm is the comparison between two elements at positions i and j to find the smaller component of the segment $[i + 1, n - 1]$.

In other words, at each stage, we need to find the smallest element in the unsorted part, which initially has n elements and requires $n - 1$ comparisons. Then we move this element to the front, and the unsorted part now has $n - 1$ elements, and so on, decreasing down to 1. From this, we can see:

- For $i = 0$, comparisons are made $n - 1$ times.
- For $i = 1$, comparisons are made $n - 2$ times.
- For $i = 2$, comparisons are made $n - 3$ times.
- ...
- For $i = n - 3$, comparisons are made 2 times.
- For $i = n - 2$, comparisons are made 1 time.

Let $T(n)$ be the execution time of the **Selection sort** algorithm for an array of length n elements. We have:

$$T(n) = 1 + 2 + 3 + \ldots + (n3) + (n2) + (n1) = \frac{n \times (n-1)}{2}$$

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

- **Best case:** This case occurs when the array is already sorted. The time complexity of the algorithm in this case is $O(n^2)$.
- **Average case:** This case typically occurs when the elements in the array are randomly distributed; neither increasing nor decreasing. The time complexity of the algorithm in this case is $O(n^2)$.
- **Worst case:** This case occurs when the array is sorted in reverse order. The time complexity of the algorithm in this case is $O(n^2)$.

2.4.3.2 Space complexity

The algorithm is $O(1)$ because **Selection sort** performs direct in-place swaps of elements in the array.

2.4.3.3 Actual Running Time

According to the data: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000.

n	**10,000**	**30,000**	**50,000**	**100,000**	**300,000**	**500,000**
Already Sorted	0.0563039	0.492681	1.33368	5.4627	50.4805	136.891
Nearly Sorted	0.0516898	0.497756	1.38788	5.38187	48.4697	131.623
Reversed Data	0.0576112	0.516835	1.4896	5.68196	50.5547	138.586
Random Data	0.0555399	0.50901	1.35655	5.43091	49.2843	131.725

Table 5: Summary of **Selection sort**

Observation: The **Selection sort** algorithm is entirely independent of the input data, meaning the running time is not affected by the way the data is distributed.

2.4.4 Improvement and Optimization

Instead of finding only the smallest value in the range $[i+1, n]$ in each loop, we will also find the largest value to place at position $n-i-1$. This allows us to sort both ends simultaneously after each iteration. This way, we only have $n/2$ stages and in each stage, we traverse the range $[i+1, n-i-1]$ which allows us to complete the array sorting.

n	**10,000**	**30,000**	**50,000**	**100,000**	**300,000**	**500,000**
Selection sort	0.0555399	0.50901	1.35655	5.43091	49.2843	131.725
Selection sort Optimize	0.034365	0.30633	0.862376	3.33919	30.7311	84.8078

Table 6: **Selection sort** and **Selection sort** Optimize according to Randomized data

Observation: Although this improved **Selection sort** does not reduce the algorithm's complexity, we can see that the improvement helps reduce the running time by about 0.4 to 0.5 times compared to the original **Selection sort**. This is because the number of stages is now $n/2$ instead of $n-1$ as before.

2.5 Shell sort

2.5.1 Idea

The **Shell sort** algorithm [4] is named after its inventor Donald Shell and is an enhancement of the **Insertion sort** algorithm. **Shell sort** allows comparisons and swaps of elements separated by a larger gap, unlike **Insertion sort** which only considers adjacent elements. Initially, the algorithm divides the sequence of numbers into groups called gaps, where each gap specifies elements that are a certain distance apart. **Shell sort** then iterates through these gaps in decreasing order (from the largest to the smallest) and sorts elements that are a gap distance apart.

2.5.2 Algorithm Description

Assume that we have an unsorted array: [68, 23, 15, 20, 12, 11, 99]

Step 1: Define the sequence of gaps (h-gaps) for the **Shell sort** algorithm. The gaps are typically chosen as a series of positive integers in a decreasing sequence $h_1, h_2, h_3, \ldots, h_n$ ending with $h_n = 1$. To illustrate the algorithm effectively in this example, we decide to use the sequence of gaps $h = \{65, 33, 17, 9, 5, 3, 1\}$ (Papernov & Stasevich 1965).

Step 2: Apply the **Insertion sort** algorithm for each gap separately. Sort each subgroup where elements are spaced apart by the gap. For instance, with a gap of 3, we sort elements at positions 0, 3, 6, ... separately and similarly for a gap of 1, we sort the entire array.

- Array after Step 2 with a gap of 3: [20, 23, 15, 68, 12, 11, 99]
- Array after Step 2 with a gap of 1: [11, 12, 15, 20, 34, 68, 99]

Result: Sorted array [11, 12, 15, 20, 34, 68, 99]

2.5.3 Complexity Analysis

The time complexity of the **Shell sort** algorithm heavily depends on the selection of gap sequences for its execution. With different gap sequences, evaluating the complexity of **Shell sort** remains an open problem without a definitive solution.

2.5.3.1 Time complexity

The group selected a gap sequence Sedgewick 1982 as:

$$\text{Distance} = \{65921, 16577, 4193, 1073, 281, 77, 23, 8, 1\}$$

Therefore, subsequent phases are dependent on this gap sequence. Let $T(n)$ denote the execution time of the **Shell sort** algorithm for an array of length n.

- **Best case:** In this scenario, the array is already sorted and $m = 9$ (the number of elements in the Distance sequence). Since the array is sorted, **Shell sort** iterates through all 9 gaps. For each gap i, we traverse approximately n/gap_i elements. Since gap_i is fixed, we can calculate as follows:

$$T(n) = n + \frac{n}{8} + \frac{n}{23} + \frac{n}{77} + \frac{n}{281} + \frac{n}{1073} + \frac{n}{4193} + \frac{n}{16577} + \frac{n}{65921} = O(n)$$

  The time complexity of the algorithm in this best case is $O(n)$ since the dominant term n remains linear with respect to n in the calculations.

- **Average case:** In this case, which often occurs when the elements in the array are uniformly distributed (neither increasing nor decreasing), the time complexity of the **Shell sort** algorithm is $O(n^{4/3})$.

- **Worst case:** In this scenario, where the array is sorted in reverse order initially, the selection of the gap sequence has little effect. Each gap between consecutive elements will be 1, essentially turning the **Shell sort** algorithm into something resembling **Insertion sort**. The time complexity of the algorithm in this case is $O(n^2)$.

2.5.3.2 Space complexity

The time complexity of **Shell sort** is typically expressed as $O(m)$, where m represents the length of the gap sequence predetermined for the algorithm's execution.

2.5.3.3 Actual Running Time

According to the data: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000.

n	**10,000**	**30,000**	**50,000**	**100,000**	**300,000**	**500,000**
Already Sorted	0	0	0.001023	0.002212	0.006154	0.00965
Nearly Sorted	0	0	0	0.002442	0.006003	0.010045
Reversed Data	0.001097	0.001003	0.001003	0.003538	0.010323	0.018174
Random Data	0.001001	0.003086	0.004982	0.011531	0.036799	0.061284

Table 7: Summary of **Shell sort**

Observation: Although **Shell sort** is an improvement over **Insertion sort** with a worst-case time complexity still at $O(n^2)$, it proves to be more efficient and significantly faster compared to other $O(n^2)$ complexity algorithms such as BubbleSort and **Selection sort**. **Shell sort** achieves this by optimizing the sorting process through the strategic selection of gap sequences, which allows it to make larger leaps in sorting elements compared to **Insertion sort**.

2.6 Heap sort

2.6.1 Idea

The **Heap sort** algorithm [6] can be considered an improvement over the **Selection sort** algorithm. The main idea of **Selection sort** is to divide the array into two parts: the sorted part and the unsorted part. In each iteration, it finds the largest element in the unsorted section and moves it to its appropriate position.

Instead of performing a linear search to find this largest element, we use a max-heap to find it, where the root of the heap is the largest element (the element we need to find).

A max-heap is defined as a binary tree where the value of each node is greater than or equal to the values of its children, with the largest value residing at the root of the heap (or the root of the tree). Specifically, an array-implemented heap satisfies the following condition: Let h be an array representing a max-heap containing elements $h_1, h_2, \ldots, h_n$ such that: $h_i \geq h_{2i+1}$, $h_i \geq h_{2i+2}$ (with the array indexed from 0).

- h_0 is the largest element in the max-heap.
- Elements in the range $[n/2, n]$ are the leaf nodes in the max-heap.

The algorithm consists of two main phases:

1. Build the max-heap: We iterate through the array from the middle to the start (from $n/2$ to 0). For each element, we push it down to its appropriate position in the max-heap by comparing it with its two children. If it is smaller than either child, we swap it with the larger child and continue the process.
2. Repeatedly extract the root of the max-heap and adjust the heap: When extracting the root of the max-heap to the end of the array, we replace it with the last element of the current array and then push this element down to its appropriate position similar to Phase 1.

2.6.2 Algorithm Description

Assume that we have an unsorted array: [7, 2, 3, 9, 4, 1, 5]

Step 1: Build a max-heap from the initial array: Building a max-heap starts from the node $n/2$ to node 0 in the binary tree. We perform Step 2 for each node.

Step 2: Adjust the binary tree into a max-heap: For each current node, compare it with the largest child node (if it exists). If the current node's value is smaller than the largest child's value, swap the values of the two nodes. Continue adjusting the binary tree until the current node is no longer smaller than its children's values or it reaches a leaf node. After Steps 1 and 2, the largest node is always moved to the root.

- Iteration 1: Swap 3 and 5: [7, 2, 5, 9, 4, 1, 3].
- Iteration 2: Swap 2 and 9: [9, 7, 5, 2, 4, 1, 3].
- Iteration 3: Swap 7 and 9: [9, 7, 5, 2, 4, 1, 3].

Step 3: Swap and reduce heap size: Swap the first node (root) with the last node in the array, placing the largest node in its correct final position. Reduce the heap size by one unit (removing the last node which is now sorted).

Step 4: Repeat Step 3 until the heap is empty (heap size reduced to 1): Start iterating from node $n-1$ to node 1, performing the process from Step 3 for each node. Here we use the symbol '|' to indicate the heap size limit after each iteration.

- Iteration 1: [7, 4, 5, 2, 1, 3 | 9] swap 9 with 3 and push 3 down.
- Iteration 2: [5, 4, 1, 2, 3 | 7, 9] swap 7 with 1 and push 1 down.
- Iteration 3: [4, 3, 1, 2, 5 | 7, 9] swap 5 with 5 and push 3 down.

- Iteration 4: [3, 2, 1, 4, 5 | 7, 9] swap 4 with 2 and push 2 down.
- Iteration 5: [2, 1, 3, 4, 5 | 7, 9] swap 3 with 1 and push 1 down.
- Iteration 6: [1 | 2, 3, 4, 5, 7, 9] swap 1 with 2 end iteration.

When the heap size is reduced to 1 (meaning the heap contains only one element), the array is fully sorted. Result sorted array [1, 2, 3, 4, 5, 7, 9]

2.6.3 Complexity Analysis

2.6.3.1 Time complexity

The time complexity of the **Heap sort** algorithm is $O(n \log n)$ in all cases. We can prove this as follows:

Let $T(n)$ be the execution time of the **Heap sort** algorithm for an array of n elements.

Phase 1: Building the Max-Heap The time complexity for building the max-heap is $O(\log n)$. This involves iterating over n elements where each element takes $O(\log n)$ time to be inserted into the max-heap.

Phase 2: Heap sort In this phase, we swap the root element of the max-heap with the last element of the array, reduce the heap size by one, and then reheapify the heap. This reheapifying process also takes $O(\log n)$ time and it is repeated n times (once for each element in the heap). Thus the running time of this phase

$$T(n) = \log n + \log(n-1) + \log(n-2) + \ldots + \log 2 = \log n(n-1)(n-2)\ldots 2 = \log n! = n \log n - n$$

In conclusion, the running time of two phases is $T(n) = n \log n - n + \log n$ of complexity $O(n \log n)$.

- **Best case:** The best case occurs when the array is already sorted. The time complexity of the algorithm in this case is $O(n \log n)$.
- **Average case:** The average case typically occurs when the elements in the array are randomly distributed, neither increasing nor decreasing. The time complexity of the algorithm in this case is $O(n \log n)$.
- **Worst case:** The worst case occurs when the array is sorted in reverse order. The time complexity of the algorithm in this case is $O(n \log n)$.

2.6.3.2 Space complexity

The space complexity of the **Heap sort** algorithm is $O(1)$ because it performs the sorting by directly swapping the positions of two elements within the array (in-place algorithm).

2.6.3.3 Actual Running Time

According to the data: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000.

Observation: For datasets that are already sorted or nearly sorted, the runtime of the algorithm is nearly equivalent. The difference in runtime is only noticeable when the data is random. This can be explained by the fact that with random data, the operations required to update the max-heap are more frequent compared to other cases.

n	10,000	30,000	50,000	100,000	300,000	500,000
Already Sorted	0	0	0	0.008005	0.024018	0.040039
Nearly Sorted	0	0	0	0.008018	0.024036	0.034546
Reversed Data	0	0	0.005008	0.008007	0.024017	0.040028
Random Data	0	0	0.005008	0.008007	0.024017	0.040028

Table 8: Summary of **Heap sort**

2.7 Bubble sort

2.7.1 Idea

Bubble sort is a simple sorting algorithm that is effective in some cases. The main idea of **Bubble sort** is to repeatedly traverse the list, comparing each pair of adjacent elements and swapping them if they are in the wrong order. This process is repeated until the list is sorted.

2.7.2 Algorithm Description

Assume we have a list of numbers: $[5, 3, 8, 4, 2]$

Step 1: Initialization: Start with the first element of the list.

Step 2: Compare and Swap: Compare the current element with the next element. If the current element is larger, swap them.

Step 3: Repeat: Move to the next pair of elements and repeat step 2 until the end of the list.

Step 4: Repeat the Whole Process: Go back to step 1 and repeat the whole process until no swaps are made during a pass through the list, indicating that the list is sorted.

- **First Pass:**
  - Compare 5 and 3, swap $\Rightarrow [3, 5, 8, 4, 2]$
  - Compare 5 and 8, no swap $\Rightarrow [3, 5, 8, 4, 2]$
  - Compare 8 and 4, swap $\Rightarrow [3, 5, 4, 8, 2]$
  - Compare 8 and 2, swap $\Rightarrow [3, 5, 4, 2, 8]$
- **Second Pass:**
  - Compare 3 and 5, no swap $\Rightarrow [3, 5, 4, 2, 8]$
  - Compare 5 and 4, swap $\Rightarrow [3, 4, 5, 2, 8]$
  - Compare 5 and 2, swap $\Rightarrow [3, 4, 2, 5, 8]$
  - Compare 5 and 8, no swap $\Rightarrow [3, 4, 2, 5, 8]$
- **Third Pass:**
  - Compare 3 and 4, no swap $\Rightarrow [3, 4, 2, 5, 8]$
  - Compare 4 and 2, swap $\Rightarrow [3, 2, 4, 5, 8]$
  - Compare 4 and 5, no swap $\Rightarrow [3, 2, 4, 5, 8]$

- Compare 5 and 8, no swap $\Rightarrow [3, 2, 4, 5, 8]$

- **Fourth Pass:**
  - Compare 3 and 2, swap $\Rightarrow [2, 3, 4, 5, 8]$
  - Compare 3 and 4, no swap $\Rightarrow [2, 3, 4, 5, 8]$
  - Compare 4 and 5, no swap $\Rightarrow [2, 3, 4, 5, 8]$
  - Compare 5 and 8, no swap $\Rightarrow [2, 3, 4, 5, 8]$

Result: Sorted array $[2, 3, 4, 5, 8]$**.**

2.7.3 Complexity Analysis

2.7.3.1 Time Complexity

- **Best Case:** When the list is already sorted.
  - **Analysis:** No swaps are made in the first pass.
  - **Time Complexity:** $O(n)$
- **Average Case:** The list is randomly ordered.
  - **Analysis:** Each element might need to be swapped $n/2$ times per pass.
  - **Time Complexity:** $O(n^2)$
- **Worst Case:** The list is sorted in reverse order.
  - **Analysis:** Each element must be swapped with every other element.
  - **Time Complexity:** $O(n^2)$

2.7.3.2 Space Complexity:

$O(1)$ as **Bubble sort** is an in-place sorting algorithm.

2.7.3.3 Actual Running Time

According to the data: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000.

n	10,000	30,000	50,000	100,000	300,000	500,000
Already Sorted	0.052636	0.479217	1.3166	5.25519	48.0413	131.798
Nearly Sorted	0.054303	0.483558	1.33443	5.29114	47.6764	132.514
Reversed Data	0.23089	2.04854	5.87842	24.8993	208.515	580.079
Random Data	0.146788	1.6232	5.16784	23.3012	224.203	619.962

Table 9: Summary of **Bubble sort**

2.7.4 Improvement and Optimization

Early Termination: A simple yet effective improvement for traditional **Bubble sort**. It reduces the number of passes if the list is partially or fully sorted before completing the usual number of passes. This improvement tracks if any swaps were made during each pass. If no swaps were made, the list is sorted, and the algorithm can terminate early.

2.8 Quick sort

2.8.1 Idea

Quick sort [8] is a divide-and-conquer sorting algorithm. The main idea of **Quick sort** is to select a "pivot" element and partition the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

2.8.2 Algorithm Description

Assume we have a list of numbers: $[10, 7, 8, 9, 1, 5]$

Step 1: Choose Pivot: Select a pivot element from the array. Common choices include the first element, the last element, the middle element, or a random element (in this report, the pivot is chosen by the first element and the median element).

Step 2: Partition: Rearrange the array so that elements less than the pivot are on the left, and elements greater than the pivot are on the right.

Step 3: Recursion: Recursively apply **Quick sort** to the left and right sub-arrays.

Step 4: Combine: Combine the sub-arrays to form the final sorted array.

- **First Selection**
  - **Choose Pivot = 10**
  - **Partition**
    - Compare 7 with 10: 7 is less than 10, so swap 7 with the first element after the pivot.
    - Compare 8 with 10: 8 is less than 10, so swap 8 with the next element.
    - Compare 9 with 10: 9 is less than 10, so swap 9 with the next element.
    - Compare 1 with 10: 1 is less than 10, so swap 1 with the next element.
    - Compare 5 with 10: 5 is less than 10, so swap 5 with the next element.
    - The result of the partition is: $[7, 8, 9, 1, 5, 10]$
  - **Recursion**
    - Left sub-array (elements less than the pivot): $[7, 8, 9, 1, 5]$
    - Right sub-array (no elements greater than 10).
- **Second Selection** (for the left sub-array $[7, 8, 9, 1, 5]$)
  - **Choose Pivot = 7**
  - **Partition**
    - Compare 8 with 7: No swap since 8 is greater than 7.

        * Compare 9 with 7: No swap since 9 is greater than 7.
        * Compare 1 with 7: Swap 1 and 8.
        * Compare 5 with 7: Swap 5 and 9.
        * The result of the partition is: $[1, 5, 7, 9, 8]$
    - **Recursion**
        * Left sub-array: $[1, 5]$
        * Right sub-array: $[9, 8]$
- **Third Selection** (for the left sub-array $[1, 5]$)
    - **Choose Pivot = 1**
    - **Partition**
        * Compare 5 with 1: No swap since 5 is greater than 1.
        * The result of the partition is: $[1, 5]$
- **Fourth Selection** (for the right sub-array $[9, 8]$)
    - **Choose Pivot = 9**
    - **Partition**
        * Compare 8 with 9: Swap 8 and 9.
        * The result of the partition is: $[8, 9]$

Result: Sorted array $[1, 5, 7, 8, 9, 10]$**.**

2.8.3 Complexity Analysis

2.8.3.1 Time Complexity

- **Best Case:** Occurs when each pivot splits the array into two nearly equal parts.
    - **Analysis:** Similar to the average case, each partition requires $O(n)$ time, and the number of partitions is $\log n$.
    - **Time Complexity:** $O(n \log n)$
- **Average Case:** When the elements are evenly split by the pivot.
    - **Analysis:** Each partition requires $O(n)$ time, and the number of partitions is $\log n$.
    - **Time Complexity:** $O(n \log n)$
- **Worst Case:** Occurs when the pivot does not split the array into balanced parts, such as when the array is already sorted (in ascending or descending order) and the pivot is the first or last element.
    - **Analysis:** Each partition only reduces the array size by one element, leading to n partitions, each taking $O(n)$ time.
    - **Time Complexity:** $O(n^2)$

2.8.3.2 Space Complexity:

Quick sort is an in-place sorting algorithm but requires additional space for the call stack due to recursion. In the average case, the call stack depth is $\log n$, leading to $O(\log n)$ space complexity. In the worst case, the call stack depth can reach n, resulting in $O(n)$ space complexity.

2.8.3.3 Actual Running Time

According to the data: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000.

n	10,000	30,000	50,000	100,000	300,000	500,000
Already Sorted	0.000936	0.002007	0.003604	0.007548	0.018242	0.036012
Nearly Sorted	0	0.002004	0.00352	0.00657	0.018148	0.033933
Reversed Data	0.000996	0.002502	0.004	0.007053	0.017068	0.036159
Random Data	0	0	0	0.01609	0.040024	0.064049

Table 10: Summary of **Quick sort**

2.8.4 Improvement and Optimization

Median of Three: The method of selecting the pivot in **Quick sort** greatly affects its performance. Choosing the first element as the pivot is simple but can lead to poor performance in some cases. The Median of Three method reduces the likelihood of worst-case performance by selecting the median of the first, middle, and last elements as the pivot. This method improves **Quick sort** by reducing the chance of worst-case scenarios and creating more balanced partitions. Although more complex and requiring additional calculations, the benefits are often worth it in practical applications, especially when working with large or unevenly distributed arrays.

2.9 Flash sort

2.9.1 Idea

Flash sort [1] is a sorting algorithm designed to handle large arrays with uniformly or nearly uniformly distributed elements efficiently. It combines Idea from distribution sorting and **Insertion sort**ing to achieve an average time complexity of $O(n)$.

2.9.2 Algorithm Description

Assume we have a list of numbers:

Step 1: Determine min and max values: Find the minimum (min) and maximum (max) values in the array.

Step2: Distribute elements into classes: Divide the array into m classes. The number of classes m is typically chosen to be about 0.2 to 0.3 times the number of elements in the array.

- Use a formula to determine the class of each element:

$$\text{class} = \left\lfloor m \cdot \frac{A[i] - \min}{\max - \min} \right\rfloor$$

Step 3: Calculate the number of elements in each class: Create a count array to store the number of elements in each class and use this array to determine the starting position of each class in the original array.

Step 4: Move elements into their correct positions: Use a permutation approach to move each element into its correct class based on the count array.

Step 5: Sort the elements within each class: Once distributed, each class can be sorted using another sorting algorithm, such as **Insertion sort**.

2.9.3 Complexity Analysis

2.9.3.1 Time Complexity

- **Best Case:** When the data is uniformly distributed, **Flash sort** can achieve nearly linear time complexity, i.e., $O(n)$.
- **Average Case:** The average complexity of **Flash sort** is $O(n+m)$, where n is the number of elements to be sorted, and m is the number of classes. Typically, m is chosen proportional to n, so the average complexity is also $O(n)$.
- **Worst Case:** When the data is not uniformly distributed, the performance can degrade to $O(n^2)$. This happens when most elements fall into a few classes, increasing the time needed to sort within those classes.

2.9.3.2 Space Complexity:

Flash sort uses additional memory of $O(n + m)$. While m is usually chosen proportional to n, this still requires extra memory to store the classes and count array.

2.9.3.3 Actual Running Time

According to the data: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000.

n	**10,000**	**30,000**	**50,000**	**100,000**	**300,000**	**500,000**
Already Sorted	0	0	0.001049	0.002081	0.007007	0.011586
Nearly Sorted	0	0.001001	0.001996	0.002	0.0065	0.009788
Reversed Data	0	0.001576	0.001002	0.001669	0.005506	0.009638
Random Data	0	0.00996	0.001053	0.002542	0.008578	0.016676

Table 11: Summary of **Flash sort**

2.10 Radix sort

2.10.1 Idea

Radix sort is a non-comparative sorting algorithm. The idea is to sort numbers digit by digit, resulting in a fully sorted array after processing through all the elements in the array. There are two approaches to implement this algorithm: start from the Least Significant Digit (LSD) to the Most Significant Digit (MSD) or vice versa.

2.10.2 Algorithm Description

In this section, we will demonstrate the algorithm using the LSD to MSD approach. The subsorting algorithm must be a stable and linear sorting algorithm. **Counting sort** and Bucket Sort are often chosen.

Consider that the array `arr = [8, 75, 123, 0, 99, 32]` is the list to be sorted.

Step 1: Determine the maximum number of digit, which is based on the largest number of the array. For instant, in `arr`, the largest number is 123, so the maximum number of digit is 3.

Step 2: Sort by the first digit. Current state of the array is `0, 32, 123, 75, 8, 99`

Step 3: Sort by the second digit. Current state of the array is `0, 8, 123, 32, 75, 99`

Step 4: Sort by the third digit. Current state of the array is `0, 8, 32, 75, 99, 123`

2.10.3 Complexity Analysis

2.10.3.1 Time complexity

Let `d` be the number of digits of the elements in the array. The algorithm will perform `d` loops. In each loop, it call a stable linear sorting algorithm, in this case is ***Counting sort***. The time complexity of ***Counting sort*** is $O(n + k)$. Therefore, the time complexity of **Radix sort** is $O(d \times (n + k))$. [6, p. 423]

- **Best case:** If all the elements have the same number of digit (`d` is *constant*) and `k` $= O(n)$, the complexity can be simplify to $O(n)$.
- **Worst case:** $O(d \times (n + k))$
- **Average case:** $O(d \times (n + k))$

2.10.3.2 Space complexity

The space complexity of **raster classification** is $O(n + k)$, where `n` is the number of elements in the input array and `k` is the input range. [5]

2.10.3.3 Actual Running Time

According to the data: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000.

n	10,000	30,000	50,000	100,000	300,000	500,000
Already Sorted	0.001217	0.003007	0.004056	0.008536	0.031712	0.053353
Nearly Sorted	0.001084	0.003116	0.004692	0.008478	0.033191	0.052793
Reversed Data	0.001003	0.00325	0.004004	0.008709	0.032816	0.054928
Random Data	0.001	0.002511	0.004156	0.008517	0.026683	0.046573

Table 12: Summary of **Radix sort**

2.10.4 Improvement and Optimization

An improvement of **Radix sort**, named **Fastbit Radix sort** [7], was published in 2016 11th International Conference on Computer Engineering & Systems (ICCES) by **Anthony Vinay Kumar S** and **Arti Arya**. The method of this algorithm is to consider the number in bits and process **Radix sort** in bitwise operation.

Let N be the size of the dataset and W be the word size. The time complexity of the algorithm follow the equations below:

- **Best case:** $T(n) = W \times \frac{N}{2}$
- **Worst case:** $T(n) = W \times N$

2.11 Shaker sort

2.11.1 Idea

Shaker sort (or Cocktail Sort) is a variant of **Bubble sort**. It goes through the array all the elements in the array upward and backward using the technique of **Bubble sort** and swaps elements if needed. If the array is sorted, the algorithm know how to stop, make in faster **Bubble sort** in some cases.

2.11.2 Algorithm Description

Consider the array `arr = [3, 4, 1, 8, 2, 2]` is the array to be sorted, `l` is the pointer points to the first element, `r` points to the last element.

Step 1: Forward pass from `l` to `r`. Current state of the array is `[3, 1, 4, 2, 2, 8]`. Decrease `r`. **Step 2:** Backward pass. Current state of the array is `1, 3, 2, 4, 2, 8`. Increase `l`. **Step 3:** Forward pass from `l` to `r`. Current state of the array is `[1, 2, 3, 2, 4, 8]`. Decrease `r`. **Step 4:** Backward pass. Current state of the array is `[1, 2, 2, 3, 4, 8]`. Increase `l`. **Step 5:** Forward pass from `l` to `r`. No swap occurred. The array is sorted.

2.11.3 Complexity Analysis

2.11.3.1 Time complexity

- **Best case:** This happens when the given array is already sorted. Only the first forward pass happens. The time complexity for this scenario is $O(n)$.
- **Average case:** $O(^2)$ [3].
- **Worst case:** When the given array is *reversed sorted*, the algorithm has to put every elements in correct order, making it cost $O(n^2)$.

2.11.3.2 Space Complexity

The space complexity is $O(1)$ since it not taking any other space to contain the array.

2.11.3.3 Actual Running Time

According to the data: 10,000; 30,000; 50,000; 100,000; 300,000; 500,000.

n	**10,000**	**30,000**	**50,000**	**100,000**	**300,000**	**500,000**
Already Sorted	0	0	0	0	0	0
Nearly Sorted	0	0.00057	0.002	0.0037	0.00805	0.0426
Reversed Data	0.237133	2.134287	10.14407	39.26626	300.7647	1004.216
Random Data	0.185266	1.649652	4.610212	25.89011	285.3387	857.0113

Table 13: Summary of **Shaker sort**

3 Experimental Results

3.1 Table

3.1.1 Sorted data

Input size	**10000**		**30000**		**50000**	
Result	**Time**	**Comparisons**	**Time**	**Comparisons**	**Time**	**Comparisons**
Selection sort	0.056039	100020001	0.492681	900060001	1.33368	2500100001
Insertion sort	0	19999	0	59999	0	99999
Bubble sort	0.052636	100010001	0.479217	900030001	1.3166	2500050001
Shaker sort	0	20001	0	60001	0	100001
Shell sort	0	193051	0	653320	0.001023	1133320
Heap sort	0	518707	0	1739635	0	3056483
Merge sort	0.001251	465243	0.004121	1529915	0.005881	2672827
Quick sort	0.000936	330305	0.002007	1204602	0.003604	2238176
Counting sort	0.001393	70005	0.001002	210005	0.001001	350005
Radix sort	0.001217	120057	0.003007	450071	0.004056	750071
Flash sort	0	103503	0	310503	0.001049	517503

Table 14: Sorted data table

Input size	100000		300000		500000	
Result	**Time**	**Comparisons**	**Time**	**Comparisons**	**Time**	**Comparisons**
Selection sort	5.4627	10000200001	50.4805	90000600001	136.891	2.50001E+11
Insertion sort	0	199999	0	599999	0.001001	999999
Bubble sort	5.25519	10000100001	48.0413	90000300001	131.798	2.50001E+11
Shaker sort	0	200001	0	600001	0	1000001
Shell sort	0.002212	2435557	0.006154	7833557	0.00965	13235557
Heap sort	0.008005	6519815	0.024018	21431639	0.040039	37116277
Merge sort	0.011925	5645659	0.033282	18345947	0.054641	31517851
Quick sort	0.007548	4429006	0.018242	12908281	0.036012	27234280
Counting sort	0.002014	700005	0.0051	2100005	0.01004	3500005
Radix sort	0.008536	1500071	0.031712	5400085	0.053335	9000085
Flash sort	0.002081	1035003	0.007007	3105003	0.011586	5175003

Table 15: Sorted data table

3.1.2 Nearly sorted data

Input size	10000		30000		50000	
Result	**Time**	**Comparisons**	**Time**	**Comparisons**	**Time**	**Comparisons**
Selection sort	0.0516898	100020001	0.497756	900060001	1.83788	2500100001
Insertion sort	0	168383	0.001101	626367	0	662255
Bubble sort	0.054303	100010001	0.483558	900030001	1.33443	2500050001
Shaker sort	0	259989	0.00057	899987	0.002	1299989
Shell sort	0	264119	0	871256	0	1302042
Heap sort	0	518536	0	1739675	0	3056398
Merge sort	0.001002	491007	0.003423	1602076	0.005494	2766406
Quick sort	0	330214	0.002004	1204584	0.00352	2238156
Counting sort	0	70005	0	210005	0.001098	350005
Radix sort	0.001084	120057	0.003116	450071	0.004692	750071
Flash sort	0	112074	0.001001	338781	0.001996	545953

Table 16: Nearly sorted data table

Input size	100000		300000		500000	
Result	**Time**	**Comparisons**	**Time**	**Comparisons**	**Time**	**Comparisons**
Selection sort	5.38187	10000200001	48.4697	90000600001	131.623	2.50001E+11
Insertion sort	0	608879	0.001334	1180183	0.002004	1325803
Bubble sort	5.29114	10000100001	47.6764	90000300001	132.514	2.50001E+11
Shaker sort	0.0037	3399985	0.00805	8999987	0.0426	2099981
Shell sort	0.002442	2628943	0.006003	8053069	0.010045	13458828
Heap sort	0.008018	6519775	0.024036	21431649	0.034546	37116394
Merge sort	0.011333	5748152	0.033173	18454812	0.054205	31623566
Quick sort	0.00657	4428990	0.018148	13908267	0.033933	27496184
Counting sort	0.002325	700005	0.005866	2100005	0.009844	3500005
Radix sort	0.008478	1500071	0.033191	5400085	0.052793	9000085
Flash sort	0.002	1066269	0.0065	3133006	0.009788	5204417

Table 17: Nearly sorted data table

3.1.3 Reversed data

Input size	10000		30000		50000	
Result	**Time**	**Comparisons**	**Time**	**Comparisons**	**Time**	**Comparisons**
Selection sort	0.0576112	100020001	0.516835	900060001	1.4896	2500100001
Insertion sort	0.069177	100019998	0.548178	900059998	1.71796	2500099998
Bubble sort	0.23089	100010001	2.04854	900030001	5.87842	2500050001
Shaker sort	0.237133	200010001	2.134287	1800030001	10.14407	5000050001
Shell sort	0.001097	302526	0.001003	1055358	0.001003	1925646
Heap sort	0	476741	0	1622793	0.005008	2848018
Merge sort	0.001311	466442	0.004118	1543466	0.005414	2683946
Quick sort	0.000996	317862	0.002502	1167240	0.004	2207736
Counting sort	0	70005	0.001001	210005	0.001001	350005
Radix sort	0.001003	120057	0.00325	450071	0.004004	750071
Flash sort	0	97755	0.001576	293255	0.001002	488755

Table 18: Reversed data table

Input size	100000		300000		500000	
Result	**Time**	**Comparisons**	**Time**	**Comparisons**	**Time**	**Comparisons**
Selection sort	5.68196	10000200001	50.5547	90000600001	138.586	2.50001E+11
Insertion sort	6.01659	10000199998	52.9224	90000599998	153.364	2.50001E+11
Bubble sort	24.8993	10000100001	208.515	90000300001	580.079	2.50001E+11
Shaker sort	39.26626	20000100001	300.7647	1.8E+11	1004.216	5.00001E+11
Shell sort	0.003538	3866543	0.010323	12878764	0.018174	22579378
Heap sort	0.008007	6087454	0.024017	20187388	0.040028	35135732
Merge sort	0.011029	5667898	0.034379	18408314	0.054184	31836410
Quick sort	0.007053	4261461	0.017068	13742055	0.036159	25408800
Counting sort	0.001376	700005	0.006144	2100005	0.009526	3500005
Radix sort	0.008709	1500071	0.032816	5400085	0.054928	9000085
Flash sort	0.001669	977505	0.005506	2932505	0.009638	4887505

Table 19: Reversed data table

3.1.4 Randomized data

Input size	10000		30000		50000	
Result	**Time**	**Comparisons**	**Time**	**Comparisons**	**Time**	**Comparisons**
Selection sort	0.0555399	100020001	0.50901	900060001	1.35655	2500100001
Insertion sort	0.03524	50230758	0.287411	448793792	0.783311	1251256109
Bubble sort	0.146788	100010001	1.6232	900030001	5.16784	2500050001
Shaker sort	0.185266	101014951	1.649652	897885037	4.610212	2507474927
Shell sort	0.001001	524951	0.003086	1820394	0.004982	3248440
Heap sort	0	497320	0	1680643	0	2951774
Merge sort	0.002005	573699	0.00504	1907558	0.00902	3332775
Quick sort	0	648713	0	2077102	0	3955810
Counting sort	0.001017	70005	0	210005	0.001002	315541
Radix sort	0.001	120057	0.002511	450071	0.004156	750071
Flash sort	0	88556	0.00996	284990	0.001053	441524

Table 20: Randomized data table

Input size	100000		300000		500000	
Result	**Time**	**Comparisons**	**Time**	**Comparisons**	**Time**	**Comparisons**
Selection sort	5.43091	10000200001	49.2843	90000600001	131.725	2.50001E+11
Insertion sort	2.96092	498034072	25.3842	4491150950	76.0444	1.25092E+11
Bubble sort	23.3012	10000100001	224.203	90000300001	619.962	2.50001E+11
Shaker sort	25.89011	1040454799	285.3387	90257249573	857.0113	2.50119E+11
Shell sort	0.011531	6928295	0.036799	22874596	0.061284	39584175
Heap sort	0.00799	6304470	0.032027	20795856	0.064093	36117714
Merge sort	0.018035	7065251	0.051827	23082799	0.085167	39883048
Quick sort	0.01609	7972759	0.040024	26983635	0.064049	51330024
Counting sort	0.001003	565541	0.004412	1565541	0.00752	2565541
Radix sort	0.008517	1500071	0.026683	4500071	0.046573	7500071
Flash sort	0.002542	847954	0.008578	2686107	0.016676	4617225

Table 21: Randomized data table

3.2 Line chart

3.2.1 Sorted data

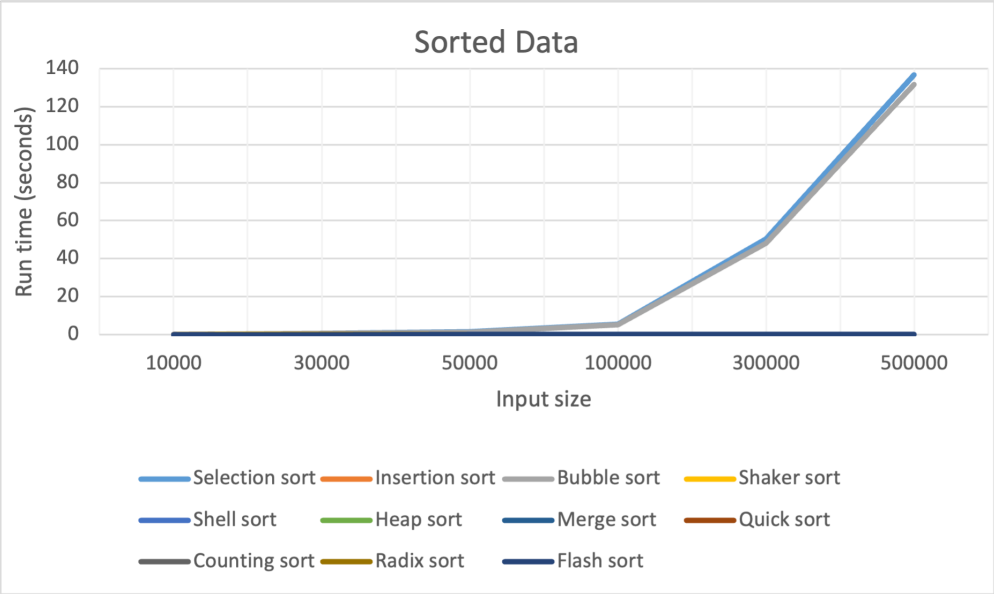

Figure 1: Sorted data line chart

Comments:

- For small input sizes (less than 100,000), the runtime of the algorithms is very small (less than 10 seconds) and there is no significant difference between the algorithms.
- When the input size increases (up to 500,000), the runtime of the algorithms increases significantly. However, some algorithms stand out with longer runtimes compared to others.
- Algorithms such as **Selection sort**, **Bubble sort**, and **Insertion sort** may have the least efficiency with large data sets, which is consistent with their theoretical time complexity of $O(n^2)$.
- Algorithms like **Quick sort**, **Merge sort**, and **Counting sort** tend to perform better with large data sets, but this is not clearly shown in the chart because the runtime of all algorithms increases significantly as the input size increases.

Overall, this chart clearly shows the growth in runtime of the sorting algorithms as the input size increases, and also highlights the differences in efficiency between the algorithms when dealing with large data sets.

3.2.2 Nearly sorted data

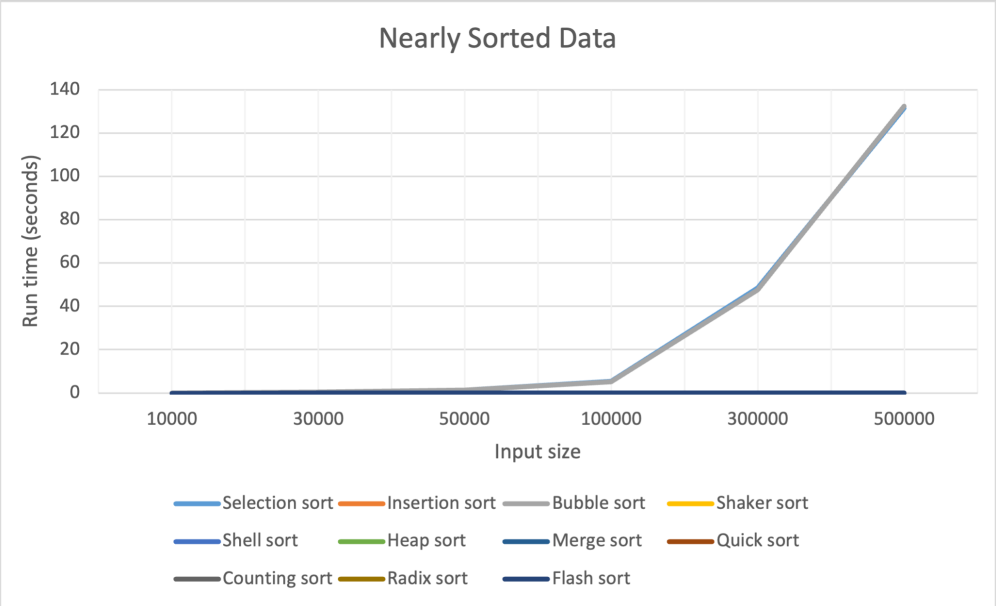

Figure 2: Nearly sorted data line chart

Comments:

- For small input sizes (less than 100,000), the runtime of the algorithms is still very small (less than 10 seconds) and there is no significant difference between the algorithms.
- When the input size increases (up to 500,000), the runtime of the algorithms increases, but the differences between the algorithms are not very clear. This can be seen with nearly sorted data, where the efficiency of the algorithms is not significantly different.
- At the largest input size (500,000), the runtime of the algorithms is nearly the same, ranging from about 120 to 140 seconds. This is similar to the previous chart, showing that the algorithms all struggle when handling large data sets that are nearly sorted.
- Some algorithms may perform better with nearly sorted data, such as **Insertion sort** and **Quick sort**, as their time complexity is better suited for these cases. However, this chart does not clearly show those differences.
- Algorithms such as **Bubble sort** and **Selection sort** may have less efficiency when handling large data sets, even if the data is nearly sorted, which is consistent with their theoretical time complexity of $O(n^2)$.

Overall, this chart shows the growth in runtime of the sorting algorithms when the input size increases, and emphasizes that even with nearly sorted data, the runtime still increases significantly with large input sizes.

3.2.3 Reversed data

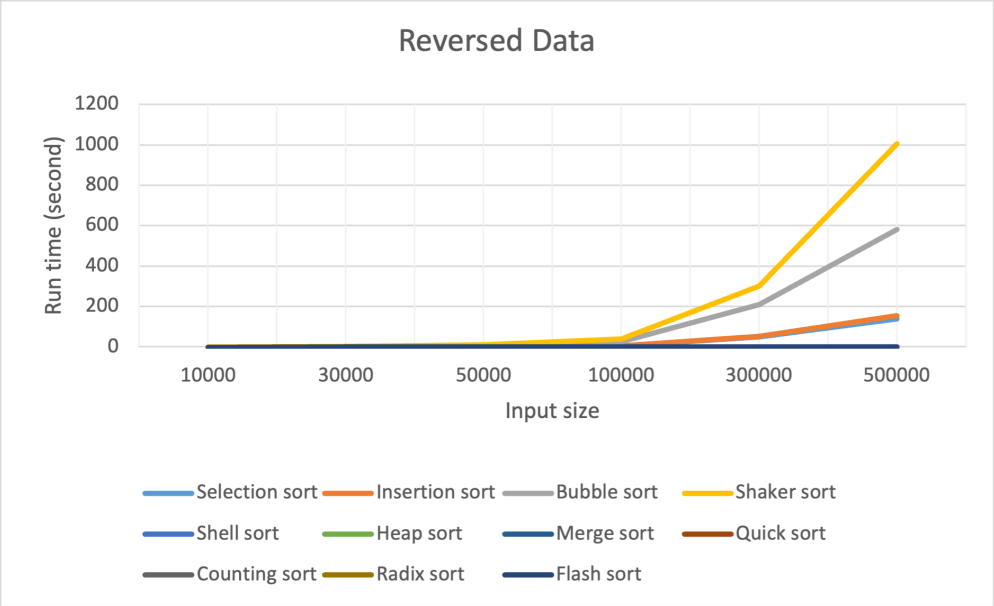

Figure 3: Reversed data line chart

Comments:

- Some algorithms like **Bubble sort**, **Insertion sort**, and **Selection sort** have significantly increased runtime.
- **Quick sort** has a sudden increase at the largest input size, indicating less efficiency when handling large and reversed data sets.
- **Counting sort** and **Radix sort** have the best efficiency, with runtime not significantly increasing even with large input sizes.
- **Shell sort**, **Heap sort**, and **Merge sort** have more stable efficiency compared to simple sorting algorithms like **Bubble sort** and **Selection sort**.
- This chart shows the growth in runtime of sorting algorithms as input size increases, especially with reversed data.
- Algorithms like **Bubble sort** and **Selection sort** are less efficient when handling large data sets, whereas **Counting sort** and **Radix sort** maintain good efficiency.

3.2.4 Randomized data

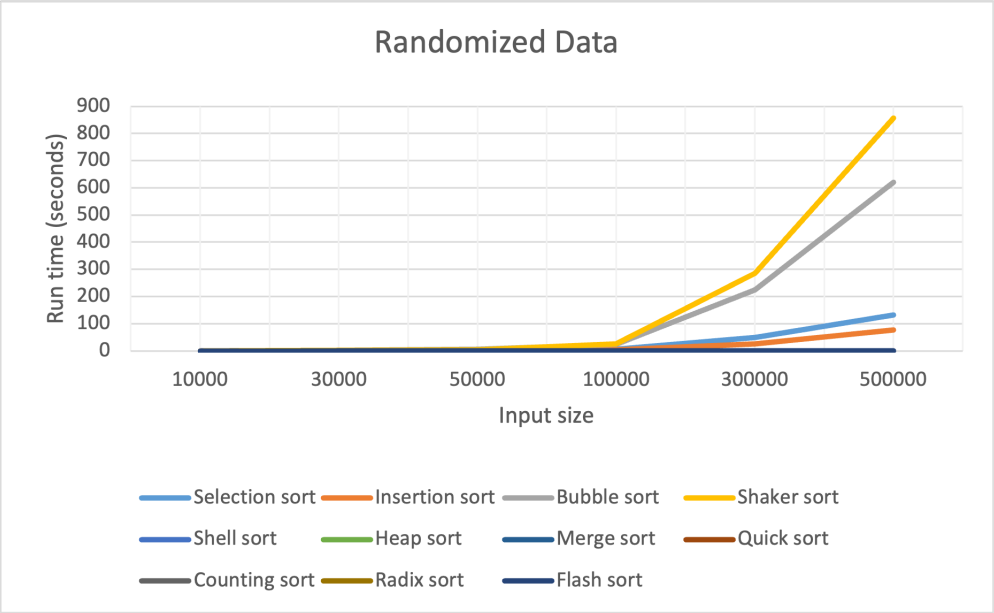

Figure 4: Randomized data line chart

Comments:

- **Shaker sort** and **Bubble sort** show a significant increase in runtime, outperforming other algorithms.
- **Counting sort** and **Radix sort** continue to show good efficiency with runtime not significantly increasing.
- **Shaker sort** and **Bubble sort** have the worst efficiency, especially with large input sizes.
- **Shell sort**, **Heap sort**, and **Merge sort** maintain more stable efficiency compared to simple sorting algorithms.
- **Shaker sort** and **Bubble sort** are the least efficient when handling large data sets.
- **Counting sort** and **Radix sort** maintain good efficiency with low runtime even with large input sizes.
- Algorithms like **Shell sort**, **Heap sort**, and **Merge sort** have more stable efficiency, suitable for handling randomized data.

3.3 Bar chart

3.3.1 Sorted data

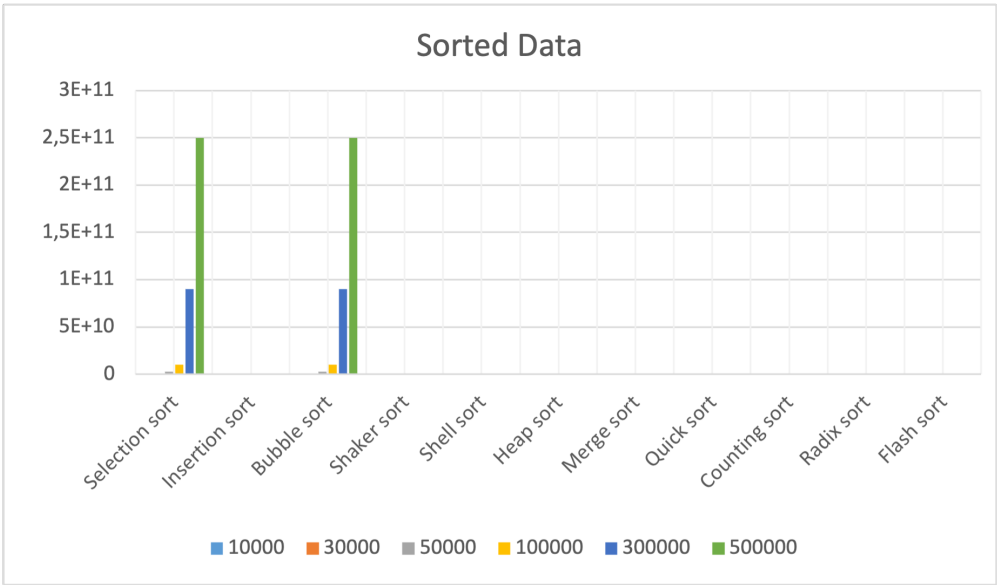

Figure 5: Sorted data bar chart

Comments:

- Other algorithms such as **Bubble sort**, **Shell sort**, **Heap sort**, **Merge sort**, **Quick sort**, **Counting sort**, **Radix sort**, and **Flash sort** have very low runtime, almost negligible compared to **Insertion sort** and **Shaker sort**.
- **Insertion sort** and **Shaker sort** have abnormally high runtimes, which does not reflect reality as for sorted data, **Insertion sort** usually has very good runtime (O(n)).
- Other algorithms like **Shell sort**, **Heap sort**, **Merge sort**, **Quick sort**, **Counting sort**, **Radix sort**, and **Flash sort** have very good efficiency when handling sorted data, with almost zero runtime.

3.3.2 Nearly sorted data

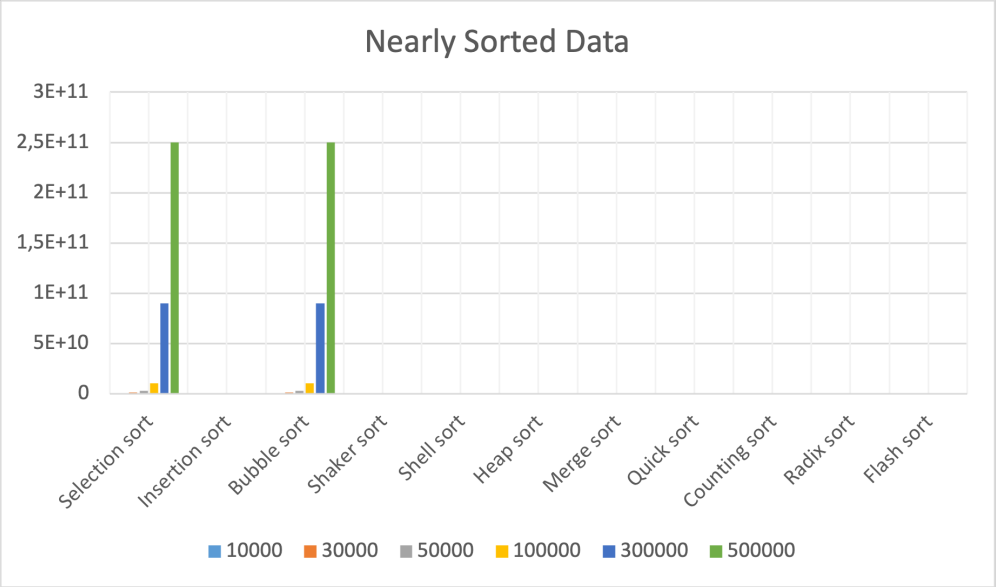

Figure 6: Nearly sorted data bar chart

Comments:

- **Insertion sort and Shaker sort:** Perform best on nearly sorted data, especially for small sizes (10,000 to 50,000 elements). This is because they work well on partially sorted data.
- **Selection sort and Bubble sort:** Take much more time compared to other algorithms, especially as the input size increases. They are not efficient with large data sets.
- **Shell sort, Heap sort, Merge sort, Quick sort:** Have stable performance, not significantly affected by input size, showing good capability to handle large data sets.
- **Counting sort, Radix sort, Flash sort:** Perform very well with large data sets. Their runtime is low and stable.
- **Insertion sort and Shaker sort** are good choices for working with nearly sorted data, especially for small sizes.
- Fast sorting algorithms like **Quick sort, Merge sort** have stable performance across all input sizes.
- **Counting sort and Radix sort** are very efficient with large data sets, due to their nature of being distribution sorts.

3.3.3 Reversed data

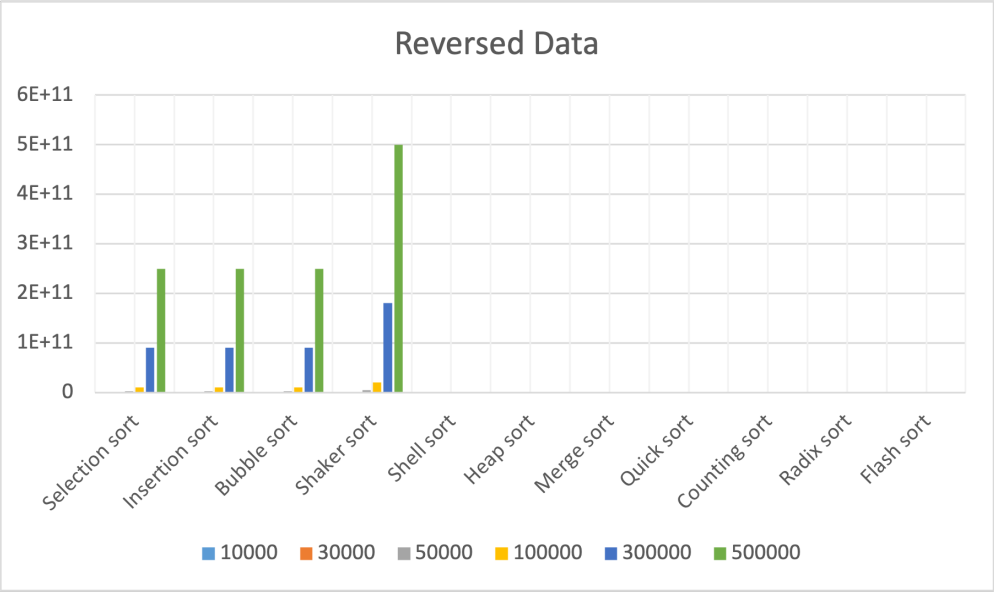

Figure 7: Reversed data bar chart

Comments:

- **Selection sort, Insertion sort, Bubble sort, and Shaker sort** perform poorly as the input size increases; these algorithms have very high runtime for large data sizes, especially 500,000 elements, where the runtime spikes.
- **Shell sort** performs better than the simple sorting algorithms above but still shows significant runtime increase for large data sizes.
- **Heap sort, Merge sort, Quick sort, Counting sort, Radix sort, and Flash sort** do not appear clearly on the chart, indicating that their runtimes are much smaller or too small to be displayed on the chart.
- This chart shows that simple sorting algorithms like **Selection sort, Insertion sort, Bubble sort, and Shaker sort** are not suitable for large data sizes, especially when the data is reversed. Efficient sorting algorithms like **Heap sort, Merge sort, Quick sort, Counting sort, Radix sort, and Flash sort** perform well in many cases.

3.3.4 Randomized data

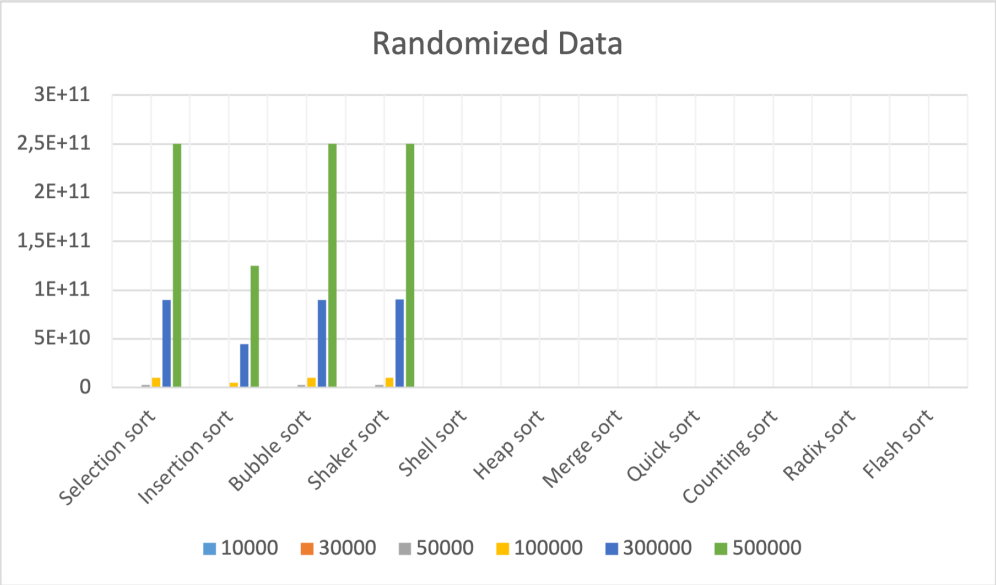

Figure 8: Randomized data bar chart

Comments:

- **Selection sort, Insertion sort, Bubble sort, and Shaker sort** perform poorly as the input size increases; these algorithms have very high runtime for large data sizes, especially 500,000 elements, where the runtime spikes.
- **Shell sort** performs better than the simple sorting algorithms above but still shows significant runtime increase for large data sizes.
- **Heap sort, Merge sort, Quick sort, Counting sort, Radix sort, and Flash sort** perform much faster and do not appear clearly on the chart due to very small runtimes compared to simple sorting algorithms.
- This chart once again confirms that simple sorting algorithms like **Selection sort, Insertion sort, Bubble sort, and Shaker sort** are not suitable for large data sets, especially when the data is randomized. Efficient sorting algorithms like **Heap sort, Merge sort, Quick sort, Counting sort, Radix sort, and Flash sort** perform well in many cases.

4 Algorithms Classification

4.1 In-place Sorting

In-place sorting algorithms include: **Selection sort**, **Insertion sort**, **Bubble sort**, **Shaker sort**, **Shell sort**, **Heap sort**, **Quick sort**.

4.2 Out-of-place Sorting

Out-of-place sorting algorithms include: **Merge sort**, **Counting sort**, **Radix sort**, **Flash sort** because they require additional arrays.

4.3 Comparison-based Sorting

Comparison-based sorting algorithms include: **Selection sort**, **Insertion sort**, **Bubble sort**, **Shaker sort**, **Shell sort**, **Heap sort**, **Merge sort**, **Quick sort**, **Flash sort**. Among these algorithms, **Flash sort** has the fastest running time in most cases, but **Heap sort** and **Merge sort** ensure a running time of $O(n \log n)$ in the worst case.

4.4 Non-comparison-based Sorting

Non-comparison-based sorting algorithms include: **Radix sort**, **Counting sort**. The common point of these algorithms is that they often have better complexity than most other algorithms, but sorting can be difficult if the data type is not an integer.

4.5 Stable Sorting

Stable sorting algorithms include: **Insertion sort**, **Bubble sort**, **Shaker sort**, **Merge sort**, **Counting sort**, **Radix sort**. Among these algorithms, **Counting sort** has the same running time and the lowest complexity.

4.6 Unstable Sorting

Unstable sorting algorithms include: **Selection sort**, **Shell sort**, **Heap sort**, **Quick sort**, **Flash sort**. Among these algorithms, **Flash sort** is the fastest.

5 Project organization and Programming Notes

5.1 Project organization

We organize our project followings these criteria: clarity and ease of navigation. This project is divided into 2 folders, `\src` and `\docs`, in which:

- `\src`: contains library declaration, library implementation and a `main.cpp` (main source code of the project).
- `\docs`: contains documentations of the project.

The source files have two main categories: Libraries of sorting algorithms and utilities to measure time and compare counts for these algorithms.

5.2 Programmings Notes

Below are our libraries of sorting algorithms.

5.2.1 BubbleSort.h

Implementation of the Bubble Sort algorithm. Contains the function `bubbleSort()` and its optimization `bubbleSort_optimized()`.

5.2.2 counting_sort.h

Implementation of Counting Sort algorithm. Contains `CountingSort()` function and its optimization `CountingSortImprovement()`.

5.2.3 FlashSort.h

Implementation of Flash Sort algorithm. Contains `flashSort()` function.

5.2.4 HeapSort.h

Implementation of Heap Sort algorithm. Contains algorithms `Heapify()` *(to build heap tree)* and `HeapSort()`.

5.2.5 merge_sort.h

Implementation of Merge Sort algorithm. Contains `Merge()` *(to merge two arrays* and `MergeSort()` *(main function of Merge Sort).*

5.2.6 insertion_sort.h

Implementation of Insertion Sort algorithm. Contains `InsertionSort()` function and its optimization `BinaryInsertionSort()`.

5.2.7 QuickSort.h

Implementation of Quick Sort algorithm. Contains `partition()` *(to make partition for Quick Sort* and `quickSort()` *main function of the algorithm.*

5.2.8 radix_sort.h

Implementation of Radix Sort algorithm. Contains `__countingSort()` *(a sub-sorting algorithm for Radix Sort that cost linear time)* and `radixSort()` *(main function of the algorithm).*

5.2.9 SelectionSort.h

Implementation of Selection Sort algorithm. Contains `SelectionSort()` function and its optimization `SelectionSortOptimized()`.

5.2.10 shaker_sort.h

Implementation of Shaker Sort algorithm. Contains `shakerSort()` function.

5.2.11 ShellSort.h

Implementation of Shell Sort algorithm. Contains `Shell()` function and its optimization `ShellSortOptimize()`.

5.2.12 SortAlgorithm.h

Manage 11 header files (.h) for 11 sorting algorithms.

In order to measuring algorithms and other utilities, we built some libraries below:

5.2.13 timeBenchmark.h

A library used to test the execution time of an algorithm. It contains `funcRunTime()` function that have 3 overloads. Each overload is for different types of parameters that the original sorting function requires.

5.2.14 output_command.h

The file <output_command.h> is used to contain prototypes of functions related to sorting algorithms, number handling, array definitions, pointer functions, etc.

5.2.15 output_command.cpp

- *GetSortName(const string & sortName):* Returns the order of the requested sorting algorithm. The returned number will be used to call the corresponding function using a function pointer.
- *GetInputOrder(const string & inputOrder):* Returns the order of the requested data type. The returned number will be used to return the corresponding name of that data type.
- *GetOutputParameter(const string &outputParameter):* Returns the type of output parameter (running time, comparisons, or both). The returned number will be used to return the corresponding name of that data type. `(-time, -comp, -both)`
- *OutputRuntime(int a[], int inputSize, const string & sortName, int command_type):* Measures and prints the runtime of the sorting algorithm.
- *OutputComparisons(int a[], int inputSize, const string & sortName, int command_type):* Counts the comparisons performed by the sorting algorithm and prints this count to the console.
- *OutputArray(const string &filePath, int a[], int inputSize):* Outputs the given data set `a[]` to a file named *filePath*
- *OutputParameter(int type, int A[], int inputSize, const string &sortName):* Print the runtime, count of comparisons or both of them of the sorting algorithm.
- *CopyArray(int a[], int data[], int inputSize):* Copies the contents of array `data[]` into array `a[]`.
- *AlgorithmMode(int argc, char *argv[]):* Executes commands that include the `-a` flag.
- *ComparisonMode(int argc, char *argv[]):* Executes commands that include the `-c` flag.

We utilized the following C++ libraries during the development of this project: `<algorithm>`, `<chrono>`, `<cmath>`, `<cstddef>`, `<cstdlib>`, `<fstream>`, `<iostream>`, `<time.h>`, `<utility>`, `<vector>`

References

[1] Educative. What is flash sort? https://www.educative.io/answers/what-is-flash-sort?fbclid=IwZXh0bgNhZW0CMTAAAR0d96Zj-6Xp1769yoNfRPYQo-7GBLg8kXZZIWrs2AdIOmtvCdJF35zoG1E_aem_8Y5emzqn6TNHnN_usfR4yA, 2024.

[2] GeeksForGeeks. Binary insertion sort. https://www.geeksforgeeks.org/binary-insertion-sort/, January 2023.

[3] GeeksForGeeks. Cocktail sort. https://www.geeksforgeeks.org/cocktail-sort/, September 2023.

[4] GeeksForGeeks. Shell sort. https://www.geeksforgeeks.org/shellsort/, March 2024.

[5] GeeksForGeeks. Time and space complexity of radix sort algorithm. https://www.geeksforgeeks.org/time-and-space-complexity-of-radix-sort-algorithm/, February 2024.

[6] Minh Hai Nguyen. Lecture 2: Sorting, June 2024.

[7] Anthony Vinay Kumar S and Arti Arya. Fastbit-radix sort: Optimized version of radix sort. In *2016 11th International Conference on Computer Engineering & Systems (ICCES)*, pages 305–312, 2016.

[8] Ronald L Rivest Clifford Stein Thomas H Cormen, Charles E Leiserson. In *Introduction to Algorithm, third edition*, 2009.