VIETNAM NATIONAL UNIVERSITY HO CHI MINH
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

PROJECT REPORT

Lab 3: Sorting Algorithms

Course: Data Structures & Algorithms

Presented by:
22127121 - Đào Việt Hoàng
22127213 - Võ Minh Khôi
22127273 - Phan Hải Minh
22127383 - Nguyễn Thành Thái

Instructors:
Mr. Bùi Huy Thông
Mrs. Trần Thị Thảo Nhi

Ho Chi Minh City, 29th July 2023

Contents

I. Information

Student's ID	Full Name	Github's username
22127121	Đào Việt Hoàng	hoangdaoviet
22127213	Võ Minh Khôi	HyperionV
22127273	Phan Hải Minh	hydroshiba
22127383	Nguyễn Thành Thái	Banhmikepthit0105

Table 1: Information of group members

II. Introduction

This Data Structures & Algorithms lab report introduces and performs measurements on various aspects of 11 sorting algorithms, namely *Bubble Sort*, *Counting Sort*, *Flash Sort*, *Heap Sort*, *Insertion Sort*, *Merge Sort*, *Quick Sort*, *Radix Sort*, *Selection Sort*, *Shaker Sort* and *Shell Sort*.

The program used in this report is implemented to measure the execution time & comparisons of the aforementioned sorting algorithms with input data of various orders: *Randomized*, *Nearly Sorted*, *Sorted*, *Reversed*; sizes: 10000, 30000, 50000, 100000, 300000, 500000. The algorithms can be measured separately, in comparison to another algorithm, or all at once.

1. Notices

For each algorithm report, the **Implemented optimizations** section reports the improvements implemented in this project while the **Further improvements & variants** section introduces various improvement ideas that are not implemented. The **Complexity** section is also based on the implementation of this project, which is the version of the algorithm in the **Implemented optimizations** section.

2. Notations

In this report many notations are used, here are some that are commonly used in pseudocode:

- a: Italicized letter denotes a variable.
- A: Capital italicized letter denotes an array. Capital A denotes the input array that needs to be sorted.
- $|A|$: Size of the array A.
- $A[i]$: The element at the i^{th} element of array A. The arrays in the implementation and this report are indexed from 0.
- $A[i, j]$: The subarray from the i^{th} to the j^{th} element of the array A.

3. Charts & graphs

If not further specified, all execution time data used in charts & graphs is measured in milliseconds.

4. Technical specifications

Algorithms are executed & measured single-threadedly using the following specifications:

- CPU: 11th Gen Intel Core i7-11370H 3.30GHz
- RAM: 16GB
- OS: Windows 11 Home 64-bit

The program is compiled with the GNU C++ Compiler version 12.2.0 using the following compile line:

```
g++ -std=c++14 -Ofast main.cpp -o main
```

III. Algorithm Presentation

1. Bubble Sort

1.1 Idea

Bubble Sort is a simple sorting algorithm that repeatedly iterates through the list, compares adjacent elements, and swaps them if they are in the wrong order. The algorithm gets its name from the way smaller elements *bubble* to the top of the list while larger elements *sink* to the bottom. It is straightforward to understand but is not very efficient, especially for large datasets.

1.2 Step-by-step description

1. Start with the first element as the current element.
2. Compare it to the next element, swap them if the next is greater
3. Move to the next element and repeat step 2 until the end of the list. After one pass through the list, the largest element will be at the last position.
4. Repeat the process for the remaining unsorted portion of the list.
5. Continue this process until the entire list becomes sorted.

1.3 Pseudocode

Algorithm 1 Algorithm for Bubble Sort - Written by Nguyen Thanh Thai

```
function BUBBLESORT($A$)
    for $i \in [0,\ |A|-2]$ do
        for $j \in [0,\ |A|-i-1]$ do
            if $A[j] > A[j+1]$ then
                swap($A[j],\ A[j+1]$)
            end if
        end for
    end for
end function
```

1.4 Implemented optimizations

In the implementation, a flag is used to keep track of whether any swaps are made during a pass through the array. If no swaps are made, it means that the array is already sorted and there is no need to do the rest of the work, thus the process ends immediately. This optimization improves the best case complexity of this algorithm to $O(N)$, when the array is already sorted.

1.5 Complexity

Average case	Best case	Worst case	Space Complexity	Stability
$O(N^2)$	$O(N)$	$O(N^2)$	$O(1)$	Yes

1.6 Further improvements & variants

- **Comb Sort** is another improvement over **Bubble Sort** that addresses some of its limitations. Instead of comparing and swapping adjacent elements, **Comb Sort** uses a "gap" value that starts as the length of the array and reduces in each pass until it becomes 1 (similar to the idea of **Shell Sort**). The gap value determines the distance between elements being compared and swapped. By using a larger gap initially, **Comb Sort** can quickly eliminate small elements at the end of the array and then progressively reduce the gap to perform smaller, more fine-grained comparisons and swaps.
- Instead of just sweeping from the beginning, **Shaker Sort** also iterates the list from the end. This sorting algorithm will be introduced further down the report.

2. Insertion Sort

2.1 Idea

The basic idea behind **Insertion Sort** is to take an element from the unsorted portion of the array and insert it into its correct position in the sorted portion of the array. This process is repeated until the entire array is sorted.

2.2 Step-by-step description

1. Start with the first element of the array, considering it as the sorted subarray.
2. Take the next element from the unsorted portion of the array.
3. Search for the correct position of this element in the sorted subarray by iterating it. The correct position is the position where the next element is larger or equal to the current one and the current one is larger or equal to its last element.
4. Shift all elements in the sorted portion from that position one step to the right to make space for the selected element.
5. Insert the selected element into its correct position in the sorted portion.
6. Continue this process until the entire list becomes sorted.

2.3 Pseudocode

Algorithm 2 Algorithm for Insertion Sort - Written by Nguyen Thanh Thai

```
function INSERTIONSORT($A$)
    for $i \in [0,\ |A| - 1]$ do
        $temp \leftarrow A[i]$
        $pos \leftarrow i - 1$
        while $pos \geq 0\ \ \&\ \ A[pos] > temp$ do
            $A[pos + 1] \leftarrow A[pos]$
            $pos \leftarrow pos - 1$
        end while
        $A[pos + 1] \leftarrow temp$
    end for
end function
```

2.4 Implemented optimizations

For finding the correct position of the selected element, instead of iterating from the back, a *binary search* is performed, reducing the complexity of this step from $O(N)$ to $O(\log N)$. This is possible since this subarray is sorted. However, the overall time complexity of **Binary Insertion Sort** remains $O(N^2)$ due to the shifting of elements, while the best complexity when the array is already sorted increases to $O(N \cdot \log N)$.

2.5 Complexity

Average case	**Best case**	**Worst case**	**Space Complexity**	**Stability**
$O(N^2)$	$O(N \cdot \log N)$	$O(N^2)$	$O(1)$	Yes

2.6 Further improvements & variants

- **Shell Sort** is a variant of **Insertion Sort**, which performs **Insertion Sort** on subsequences of elements separated by a certain *gap* length. The report will later introduce this algorithm.
- **Block Sort** is another variant of **Insertion Sort** that sorts elements in blocks of fixed size. The sorting process is performed within these blocks using **Insertion Sort**. Once the blocks are sorted, they are merged to produce the final sorted array. **Block Sort** can improve the performance of **Insertion Sort** for large arrays by reducing the number of comparisons and movements.

3. Shaker Sort

3.1 Idea

Shaker Sort or **Bidirectional Bubble Sort** is an improvement over the standard **Bubble Sort** algorithm. The idea behind **Shaker Sort** is to address one of the **Bubble Sort**'s limitations, which is its inefficiency in sorting elements located towards the end of the array.

3.2 Step-by-step description

1. Start with the assumption that the entire array is unsorted.
2. Set two pointers, *left* and *right*, initially pointing to the first and last elements of the array respectively.
3. Perform a *forward pass*through the array similarly to the **Bubble Sort** algorithm. Compare adjacent element and swap them if they are in the wrong order. After this pass, the largest element will *bubble up* to the last position.
4. Decrease the *right pointer* to exclude the last element, as it is already in its correct position.
5. Perform a *backward pass* through the remaining unsorted portion of the array. Compare adjacent elements and swap them if they are in the wrong order.
6. Increase the *left pointer* to exclude the first element, as it is already in its correct position.
7. Repeat steps 3 to 6 until the *left* and *right* pointers meet each other or cross each other. In each pass, one more element from each end of the array becomes sorted.

3.3 Pseudocode

Algorithm 3 Algorithm for Shaker Sort - Written by Nguyen Thanh Thai

```
function SHAKERSORT(A)
    $surface \leftarrow 0$
    $bottom \leftarrow |A| - 1$
    $k \leftarrow |A| - 1$
    while $surface < bottom$ do
        for $j \in [bottom,\ surface + 1]$ do
            if $A[j-1] > A[j]$ then
                swap($A[j-1],\ A[j]$)
                $k \leftarrow j$
            end if
        end for
        for $i \in [surface, bottom - 1]$ do
            if $A[i] \geq A[i+1]$ then
                $A[i+1] \leftarrow a[i]$
                $k \leftarrow j$
            end if
        end for
        $bottom \leftarrow k$
    end while
end function
```

3.4 Implemented optimizations

Similarly to **Bubble Sort**, a flag is also used to check whether a swap was performed, and the algorithm breaks early if there is no swap. The best case complexity is therefore $O(N)$ for sorted arrays.

3.5 Complexity

Average case	**Best case**	**Worst case**	**Space Complexity**	**Stability**
$O(N^2)$	$O(N)$	$O(N^2)$	$O(1)$	Yes

3.6 Further improvements & variants

- **Library Sort** is a hybrid sorting algorithm that combines **Insertion Sort**, **Bubble Sort**, and **Merge Sort**. It is considered an improvement because it utilizes the strengths of **Bubble Sort** for small sub-arrays and switches to **Merge Sort** for larger sub-arrays.

4. Counting Sort

4.1 Idea

The main idea behind **Counting Sort** is to sort an array of integers by counting the occurrences of each unique element in the input array, then determine the element's relative position in the output array. The algorithm assumes that the input array contains only integers, and that the integers are in a certain range.

4.2 Step-by-step description

1. Determine the range of input values, which is the size of the counting array using the maximum value.
2. Create the counting array and initialize all its elements to 0. Count the occurrences of each value.
3. Compute the cumulative sum of the counting array by adding the previous element of the counting array into the current one.
4. Loop the input array and use the counting array to put the elements at their correct positions in a temporary array, then decrease the corresponding element in the counting array after each placement.
5. Copy the temporary array to the original one.

4.3 Pseudocode

Algorithm 4 Algorithm for Counting Sort - Written by Vo Minh Khoi

```
function COUNTINGSORT($A$)
    $k \leftarrow 0$, $T \leftarrow$ [new array of size$|A|$]
    for $i \in [0, |A| - 1]$ do
        if $A[i] > k$ then
            $k \leftarrow A[i]$
        end if
    end for
    $C \leftarrow$ [new array of size $k + 1$]
    for $i \in [0, k]$ do
        $C[i] \leftarrow 0$
    end for
    for $i \in [0, |A| - 1]$ do
        $C[A[i]] \leftarrow C[A[i]] + 1$
    end for
    for $i \in [0, k]$ do
        $C[i] \leftarrow C[i] + C[i - 1]$
    end for
    for $i \in [|A| - 1, 0]$ do
        $T[C[A[i]]] \leftarrow A[i]$
        $C[A[i]] \leftarrow C[A[i]] - 1$
    end for
    for $i \in [0, |A| - 1]$ do
        $A[i] \leftarrow T[i]$
    end for
end function
```

4.4 Implemented optimizations

In the fourth step, the input array is iterated from the back to retain relative order of elements before sorting, hence making the algorithm stable.

4.5 Complexity

Let K be the range of input.

Average case	Best case	Worst case	Space Complexity	Stability
$O(N + K)$	$O(N + K)$	$O(N + K)$	$O(K + N)$	Yes

4.6 Further improvements & variants

- With **Parallel Processing**, **Counting Sort** can be parallelized by dividing the dataset into smaller segments and performing the counting step independently for each segment. The counts can then be combined and used to reconstruct the sorted array. This approach can lead to significant speedup, especially for large datasets, when using multi-core processors or distributed computing systems.
- If the range of input values is much smaller than the size of the counting array, it is possible to use a **Compressed Counting Array** to reduce memory usage. This can be achieved by mapping the input values to a smaller range before counting them.
- In some cases, it might be beneficial to combine **Counting Sort** with another sorting algorithm to handle datasets with a mix of small and large value ranges. **Adaptive Counting Sort** partitions the dataset into smaller segments based on the range of values and different sorting algorithms can be applied to each segment depending on the value range and distribution.
- The performance of **Counting Sort** can be improved by using **Optimized Memory Allocation** techniques. For example, allocate the counting array directly from the stack instead of the heap to reduce allocating and de-allocating memory overhead.

5. Flash Sort

5.1 Idea

Flash Sort is an efficient sorting algorithm that is especially well-suited for large datasets with a uniform value distribution or a small data range. The main idea behind **Flash Sort** is to use a combination of bucketing and partitioning to efficiently sort the data, taking advantage of the distribution of the datasets.

5.2 Step-by-step description

1. Partition the data into *buckets*. The number of buckets chosen is usually linear to the size of the array.
2. Count the number of elements in each bucket by calculating the *bucket index*[1] for each element in the array.
3. Accumulate the bucket size array to make it a prefix sum.
4. Permute the array elements to their correct positions according to the buckets.
5. Sort the elements within each bucket independently, this is done by using **Insertion Sort**.

5.3 Pseudocode

Algorithm 5 Algorithm for Flash Sort - Written by Vo Minh Khoi

```
function FLASHSORT($A$)
    $min \leftarrow \min A$, $max \leftarrow \max A$
    $B \leftarrow$ [new array of size $0.43 \cdot |A|$]
    $range \leftarrow max - min + 1$
    for $i \in [0, |B| - 1]$ do
        $B[i] \leftarrow 0$
    end for
    for $i \in [0, |A| - 1]$ do
        $B[|B| \cdot (A[i] - min)/range] \leftarrow B[|B| \cdot (A[i] - min)/range] + 1$
    end for
    for $i \in [1, |B| - 1]$ do
        $B[i] \leftarrow B[i] + B[i - 1]$
    end for
    for $i \in [0, |A| - 1]$ do
        $j \leftarrow |B| \cdot (A[i] - min)/range$
        if $i \geq B[j]$ then
            continue
        end if
        $B[j] \leftarrow B[j] - 1$
        while $i \neq B[j]$ do
            swap($A[i]$, $A[B[j]]$)
            $j \leftarrow |B| \cdot (A[i] - min)/range$
            $B[j] \leftarrow B[j] - 1$
        end while
    end for
    $start \leftarrow 0$
    for $i \in [1, |A| - 1]$ do
        $j \leftarrow |B| \cdot (A[i] - min)/range$
        if $|B| \cdot (A[start] - min)/range < |B| \cdot (A[i] - min)/range$ then
            insertionSort($A[start, i - 1]$)
            $start \leftarrow i$
        end if
    end for
    insertionSort($A[start, |A| - 1]$)
end function
```

[1]which is calculated as (maximum value - minimum value) / number of buckets

5.4 Implemented optimizations

In the last step of the sorting algorithm, each partition is sorted using **Insertion Sort**. However, in this implementation, each partition is sorted recursively using **Flash Sort** itself. The sort also exits early if all elements are equal ($min = max$).

5.5 Complexity

Average case	**Best case**	**Worst case**	**Space Complexity**	**Stability**
$O(N)$	$O(N)$	$O(N^2)$	$O(N)$	No

5.6 Variants & Impromements

- Use different sorting algorithms for different buckets based on their size of the distribution of their elements. For example, use **Quick Sort** for larger buckets and **Insertion Sort** for smaller buckets.
- **Parallel Processing** by perform the local sorting step in parallel for different buckets using multi-threading or distributed computing technique.
- **Pre-process** the data to make it more suitable before sorting. This action can remove duplicates, filter out outliers to obtain a more uniform distribution.

6. Shell Sort

6.1 Idea

Shell sort is a sorting algorithm which is a variation of **Insertion Sort**. The algorithm works by comparing elements that are distant from each other rather than adjacent and gradually decreasing the distant between the elements begin compared.
The main idea behind **Shell Sort** is that it reduces the number of swaps needed to sort the array compared to **Insertion Sort**. By sorting the elements in larger gaps first, the algorithm can move elements farther from their final position, reducing the number of swaps needed to bring them into place.

6.2 Step-by-step description

1. The algorithm start by dividing the input array into smaller sub-arrays of a certain gap size, often determined by a sequence of gap values.
2. The sub-arrays are then sorted using **Insertion Sort** with the gap size decreasing in each iteration.
3. The process continues until the gap size is reduced to 1, at which point the algorithm performs a final **Insertion Sort** on the entire array.

6.3 Pseudocode

Algorithm 6 Algorithm for Shell Sort - Written by Vo Minh Khoi

```
function SHELLSORT($A$)
    $gap \leftarrow 0$
    while $gap < (size/3)$ do
        $gap \leftarrow (interval * 3) + 1$
    end while
    while $gap > 0$ do
        for $outer \in [gap, |A| - 1]$ do
            $InsertionValue \leftarrow A[outer]$
            $inner \leftarrow outer$
            while $inner > gap - 1 \& A[inner - gap] \geq InsertionValue$ do
                $array[inner] \leftarrow A[inner{-}gap]$
                $inner \leftarrow inner - gap$
            end while
        end for
        $gap \leftarrow (gap - 1)/3$
    end while
end function
```

6.4 Implemented optimizations

The above pseudocode uses the Knuth 1973 sequence of gaps, however this sequence is not the fastest. The Sedgewick 1982 sequence of gaps is used in the implementation instead because of its better efficiency while still being relatively easy to implement. The expected average and worst complexity is $O(N^{\frac{4}{3}})$.

6.5 Complexity

Average case	Best case	Worst case	Space Complexity	Stability
$O(N^{\frac{4}{3}})$	$O(N \cdot \log N)$	$O(N^{\frac{4}{3}})$	$O(1)$	No

6.6 Further improvements & variants

1. Use Hybrid Sorting Algorithm

Shell Sort can be combined with another sorting algorithm such as **Quick Sort**, **Merge Sort** to improve its performance. This can help take advantage of **Shell Sort**'s strengths while avoiding its weaknesses.

2. Use Optimized Insertion Sort

The performance of the final **Insertion Sort** can be improved by using an optimized version of the algorithm, for instance, **Binary Insertion Sort**.

3. Use Parallel Processing

Shell Sort can be parallelized to take advantage of multiple processors or threads so it will run faster on systems with multiple processors or cores.

7. Selection Sort

7.1 Idea

The main idea behind **Selection Sort** is to divide the input list into two parts: the *sorted* part and the *unsorted* part. The algorithm repeatedly selects the *minimum* (or *maximum*) element from the unsorted part and places it at the *beginning*(or *end*) of the sorted part.

This process continues until the entire array is sorted. **Selection Sort** is a simple but not very efficient sorting algorithm, especially for large datasets.

7.2 Step-by-step description

1. Start with the first element as the minimum (or maximum) in the unsorted part.
2. Iterate through the unsorted part to find the minimum (or maximum) element.
3. Swap the found minimum (or maximum) element with the first element of the unsorted part.
4. Increment the boundary between the sorted and unsorted parts by moving it one position to the right.
5. Repeat steps 1 to 4 until the entire array is sorted.

7.3 Pseudocode

Algorithm 7 Algorithm for Selection Sort - Written by Dao Viet Hoang

```
function SELECTIONSORT( *array, size)
    for $i \in [0, size - 1]$ do
        $MinIndex \leftarrow i$
        for $j \in [i + 1, size - 1]$ do
            if $array[j] < array[MinIndex]$ then
                $MinIndex \leftarrow j$
            end if
        end for
        $swap(array[i], array[MinIndex])$
    end for
    return array
end function
```

7.4 Complexity

Average case	Best case	Worst case	Space Complexity	Stability
$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$	No

7.5 Further improvements & variants

1. Heap Sort

Heap sort is an improvement over **Selection Sort** that uses a *binary heap* data structure to efficiently find and remove the maximum (or minimum) element from the unsorted part of the array. By using a heap, the time complexity for selecting the maximum element is reduced to $O(\log N)$, resulting in an overall time complexity of $O(N \log N)$. **Heap Sort** is an in-place sorting algorithm, and its main advantage is its improved time complexity compared to **Selection Sort**.

2. Tournament Sort

Tournament Sort (also known as the **Tournament Tree Sort**) is a variation of **Selection Sort** that uses a tournament tree data structure to select the minimum element. It has a time complexity of $O(N \log N)$, which is better than the original **Selection Sort**.

8. Heap Sort

8.1 Idea

Heap Sort is a comparison-based sorting technique based on *Binary Heap* data structure. It is similar to the **Selection Sort** where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

8.2 Step-by-step description

1. Build Max Heap: Convert the input array into a max heap. Start by calling the *heapify* function for each non-leaf node, starting from the last non-leaf node and moving upwards towards the root. This ensures that the entire array satisfies the max heap property.

2. Heapify (Sift Down): The heapify operation involves *sifting down* an element to its correct position in the max heap. To heapify a node at index i, compare it with its children. If the node is smaller than any of its children, swap it with the larger child. Repeat this process recursively for the swapped child until the node is in its correct position, and the sub-tree rooted at that element satisfies the max heap property.

3. Extract Maximum: After building the max heap, the maximum element is at the root (index 0). Swap this element with the last element in the unsorted part of the array (last element of the heap). The largest element is now at its correct position in the sorted part of the array.

4. Decrease Heap Size: Decrease the heap size by one (excluding the last element, which is now sorted). This effectively shrinks the unsorted part of the heap.

5. Repeat: Repeat steps 2 to 4 until the entire array becomes sorted. In each iteration, the maximum element is extracted and placed at the end of the array, and the heap size is reduced.

6. Sorted Array: Once all iterations are complete, the input array is now sorted in ascending order.

8.3 Pseudocode

Algorithm 8 Algorithm for Heap Sort - Written by Dao Viet Hoang

```
function HEAPIFY(*array, n, root)
    $largest \leftarrow root$
    $leftChild \leftarrow 2 * root + 1$
    $rightChild \leftarrow 2 * root + 2$
    if $leftChild < n$ & $array[leftChild] > array[largest]$ then
        $largest \leftarrow leftChild$
    end if
    if $rightChild < n$ & $array[rightChild] > array[largest]$ then
        $largest \leftarrow rightChild$
    end if
    if $largest \neq root$ then
        swap($array[root]$, $array[largest]$)
        heapify($array$, $n$, $largest$)
    end if
end function

function HEAPSORT(*array)
    $n \leftarrow$ length($array$)
    for $i \in [n/2 - 1,\ 0]$ do
        heapify($array$, $n$, $i$)
    end for
    for $i \in [n - 1,\ 1]$ do
        swap($array[0]$, $array[i]$)
        heapify($array$, $i$, $0$)
    end for
    return $array$
end function
```

8.4 Complexity

Average case	**Best case**	**Worst case**	**Space Complexity**	**Stability**
$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(1)$	No

8.5 Further improvements & variants

1. Binary Heap Optimizations
Some optimizations can be applied to the binary heap data structure used in **Heap Sort**. For example, binary heap operations can be implemented more efficiently using bitwise operations for calculating the indices of parent and child nodes.

2. Ternary Heap Sort
Ternary Heap Sort is a variant that uses a *ternary* (3-ary) heap instead of a binary heap. This reduces the number of comparisons and swaps required during heapification.

9. Merge Sort

9.1 Idea

The main idea behind **Merge Sort** is to repeatedly divide the input array into smaller sub-arrays, sort these sub-arrays (which is relatively easy since they are small), and then merge them back together to form the final sorted array. This process continues until the entire array becomes sorted.
hspace0.5cm **Merge Sort**'s stability and efficiency make it a popular choice for sorting tasks.

9.2 Step-by-step description

1. Divide: Divide the input array into two equal (or almost equal) halves. This can be done by finding the middle index of the array.
2. Recursively Sort: Recursively apply **Merge Sort** to each of the two halves obtained in the previous step. Keep dividing the subarrays until each subarray contains only one element. By definition, single-element arrays are considered sorted.
3. Merge: Merge the two sorted subarrays obtained from the previous step into a single sorted array. This involves comparing elements from the two subarrays and placing them in order into an output array.
4. Combine Sorted Subarrays: Continue merging the sorted subarrays until all the elements from both halves are merged into a single sorted array.
5. Final Result: The final output array will be a sorted version of the input array.

9.3 Pseudocode

9.4 Complexity

Average case	**Best case**	**Worst case**	**Space Complexity**	**Stability**
$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	Yes

9.5 Further improvements & variants

1. Iterative Merge Sort
This variant of **Merge Sort** avoids the use of recursion and instead implements the algorithm iteratively using a loop. It divides the array into smaller subarrays, merges them, and then doubles the size of the subarrays until the entire array is sorted. **Iterative Merge Sort** can be more memory-efficient for large arrays compared to the recursive version.

2. Block Merge Sort
Block Merge Sort is a modification of **Merge Sort** that uses a two-step approach. In the first step, it sorts small blocks of elements within the array using a different sorting algorithm (like **Insertion Sort**). Then, it performs the merge step on the sorted blocks to create the final sorted array. **Block Merge Sort** is designed to minimize the overhead of the merge step for small blocks.

Algorithm 9 Algorithm for Merge Sort - Written by Dao Viet Hoang

```
function MERGE(left, right)
    result ← [empty list]
    $leftIndex \leftarrow 0$
    $rightIndex \leftarrow 0$
    while $leftIndex <$ length(left) & $rightIndex <$ length(right) do
        if $left[leftIndex] \leq right[rightIndex]$ then
            result.append($left[leftIndex]$)
            $leftIndex \leftarrow leftIndex + 1$
        else
            result.append($right[rightIndex]$)
            $rightIndex \leftarrow rightIndex$
        end if
        $result \leftarrow result + left[leftIndex]$
        $result \leftarrow result + right[rightIndex]$
    end while
    return result
end function

function MERGESORT( *array)
    if length(array) $\leq 1$ then
        return array
    end if
    $n \leftarrow$ length(array)
    $middle \leftarrow n/2$
    $leftHalf \leftarrow$ mergeSort($array[0 : middle]$)
    $rightHalf \leftarrow$ mergeSort($array[middle : n]$)
    return merge($leftHalf$, $rightHalf$)
end function
```

10. Quick Sort

10.1 Idea

The main idea behind **Quick Sort** is to efficiently partition the input array using a pivot element, recursively sort the two resulting sub-arrays, and combine them to obtain the final sorted array.

The efficiency and in-place nature of **Quick Sort** make it a popular choice for various sorting tasks. However, it may suffer from poor performance in certain scenarios when the pivot selection is not optimized.

10.2 Step-by-step description

1. Choose Pivot Element: Select a pivot element from the input array. The pivot element can be any element from the array, but its choice significantly affects the efficiency of the algorithm. Common strategies for selecting the pivot include choosing the first element, the last element, the middle element, or a random element.

2. Partitioning: Rearrange the elements of the array such that all elements less than the pivot come before it, and all elements greater than the pivot come after it. The pivot element is now in its correct sorted position.

3. Recursive Calls: After partitioning, Quick sort is recursively applied to the two subarrays formed on either side of the pivot. This process continues until the subarrays have only one element.

4. Combine Results: Once the recursive calls are complete, the subarrays will be sorted. Combine them along with the pivot element to form the final sorted array.

5. Base Case: The base case of the recursion is when the subarray has zero or one element, in which case it is already considered sorted.

10.3 Pseudocode

10.4 Complexity

Average case	Best case	Worst case	Space Complexity	Stability
$O(N \log N)$	$O(N \log N)$	$O(N^2)$	$O(N \log N)$	No

Algorithm 10 Algorithm for Quick Sort - Written by Phan Hai Minh

```
function PARTITION($*array, low, high$)
    $pivot \leftarrow array[high]$
    $i \leftarrow low - 1$
    $rightIndex \leftarrow 0$
    for $i \in [low,\ high - 1]$ do
        if $arr[j] \leq pivot$ then
            $i \leftarrow i + 1$
            swap($array[i],\ array[j]$)
        end if
    end for
    swap($array[i + 1],\ array[high]$)
    return $i + 1$
end function

function QUICKSORT($*array, low, high$)
    if $low < high$ then
        $pivotIndex \leftarrow$ partition($array,\ low,\ high$)
        quicksort($array,\ low,\ pivotIndex - 1$)
        quickSort($array,\ pivotIndex + 1,\ high$)
    end if
end function
```

10.5 Further improvements & variants

1. Three-Way Quick Sort
In the standard **Quick Sort**, elements equal to the pivot are not directly sorted and end up on either side of the pivot. **Three-Way Quick Sort** is an improvement that partitions the array into three sections: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot. This optimization is particularly useful when the input array contains many duplicate elements.

2. Tail Call Optimization
Tail Call Optimization (*TCO*) is an optimization technique that avoids creating a new stack frame for certain recursive calls. Applying *TCO* to **Quick Sort** can help reduce the overhead of recursion and improve performance.

3. Intro Sort
Intro Sort is a hybrid sorting algorithm that starts with **Quick Sort** but switches to **Heap Sort** when the recursion depth exceeds a certain threshold. This prevents the worst-case behavior of **Quick Sort** while still maintaining good performance on average.

11. Radix Sort

11.1 Idea

The idea of **Radix Sort** is to do digit-by-digit sorting starting from the least significant digit to the most significant digit. **Radix Sort** uses counting sort as a subroutine to sort.

11.2 Step-by-step description

1. Find the Maximum Element: Find the maximum element in the input array.
2. Initialize Buckets: Prepare the buckets to hold the elements during each pass.
3. Least Significant Digit (LSD) Pass: Start sorting the input array from the least significant digit (rightmost digit). For each element in the array, extract the least significant digit and place it into the corresponding bucket.
4. Stable Sorting: Use a stable sorting algorithm (e.g., counting sort or bucket sort) to sort the elements within each bucket based on the current digit position. Ensure that the order of elements with equal digits in the current position is preserved.
5. Recombine Elements: After sorting the elements based on the current digit position, combine them back into the original array, keeping the relative order of elements within each bucket.
6. Repeat for All Digits: Continue the above process for each subsequent digit position, moving from the least significant to the most significant digit. Each pass will sort the elements based on the current digit position.

11.3 Pseudocode

Algorithm 11 Algorithm for Radix Sort - Written by Phan Hai Minh

```
function RADIXSORT(*array)
    $n \leftarrow$ length($array$)
    $maxElement \leftarrow$ findMax($array$)
    $numDigits \leftarrow$ countDigits($maxElement$)
    for $digitPosition \in [1, numDigits]$ do
        $buckets \leftarrow initializeBuckets()$
        for $i \in [0, n-1]$ do
            $digit \leftarrow$ getDigit($array[i]$, $digitPosition$)
            $buckets[digit]$.append($element$)
        end for
        $array \leftarrow$ concatenateBuckets($buckets$)
    end for
    return $array$
end function
```

11.4 Complexity

Let K represents the maximum number of digits in the input elements.

Average case	**Best case**	**Worst case**	**Space Complexity**	**Stability**
$O(N * K)$	$O(N * K)$	$O(N * K)$	$O(N + K)$	Yes

11.5 Further improvements & variants

1. Least Significant Digit (LSD) Radix Sort
This is the standard implementation of **Radix Sort**, where the sorting starts from the least significant digit and progresses to the most significant digit. It is suitable for sorting numbers in ascending order.

2. Most Significant Digit (MSD) Radix Sort
MSD Radix Sort starts the sorting process from the most significant digit and moves towards the least significant digit. **MSD Radix Sort** is typically used for sorting numbers in descending order.

3. Two-Pass Radix Sort
In some cases, a **Two-Pass Radix Sort** can be used to sort elements with two different keys. For example, in a database table, if you want to sort first by one field and then by another, you can perform two passes of **Radix Sort** to achieve the desired result.

4. Parallel Radix Sort
Parallel Radix Sort is an optimization that exploits parallel processing techniques to speed up the sorting process. By distributing the sorting workload across multiple processors or threads, **Parallel Radix Sort** can significantly reduce the sorting time for large datasets.

5. Variable-Length Radix Sort
Standard **Radix Sort** assumes that all elements have the same number of digits. **Variable-Length Radix Sort** extends the algorithm to handle elements with varying numbers of digits. It efficiently sorts integers with different digit lengths.

It's important to note that the choice of the Radix sort variant depends on the specific requirements of the sorting task and the characteristics of the input data. While Radix sort can be very efficient for sorting integers with a fixed or limited range of digits, other sorting algorithms may be more appropriate for other data types or scenarios.

IV. Experimental Results & Comments

1. Experimental Results

The experimental results for 11 algorithms are presented into 4 tables with different data orders (see in next pages).

There are also 4 line graphs, each of which corresponds to a table of running times. In each graph, the horizontal axis is for data size and the vertical axis is for running time

There will be 4 bar charts, each of which corresponds to a table of numbers of comparisons. In every graph, the horizontal axis is for data size, and the vertical axis is for the numbers of comparisons.

2. Overall Comments

As can be seen from the 4 line graphs, selection sort is the worst sorting algorithm according to existing with explicit lines. While, on the contrary, counting sort ranks first when the algorithm always has a steadiness with trivial growth. Regarding radomized and reversed input, bubble sort and shaker sort are the slowest ones. However, shaker sort witnessed a dramatic changes when it comes to 2 remaining input order meanwhile bubble sort is unchanged in nearly sorted input - It is clear that shaker sort is the improvement of bubble sort.

As is observed to the 4 bar charts, Selection Sort has the longest running time regardless of input order. In randomized and reversed order, bubble sort, insertion sort and shaker sort are the next ranking on running time with approximation, which are far exceed than remaining. However, when it comes to 2 other bar charts, the insertion and shaker saw roughly a twofold decrease while the bubble sort does not have much fluctuation in nearly sorted input

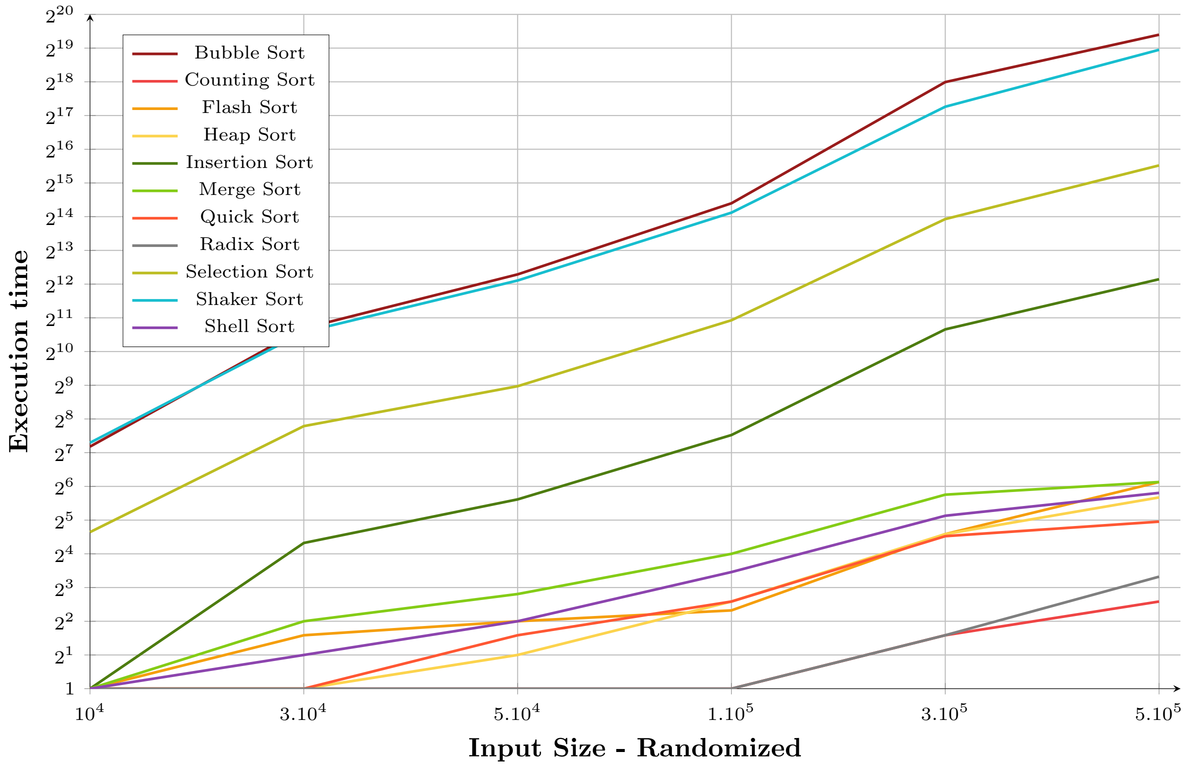
In conclusion, all the algorithms, which are except bubble sort, shaker sort, insertion sort and selection sort, are consistent algorithms over the data size and data order.

Data order: *Randomized data*						
Data size	**10,000**		**30,000**		**50,000**	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	25	100010001	221	900030001	502	2500050001
Insertion Sort	1	25901098	20	226949001	49	629652775
Bubble Sort	145	99971388	1651	900016194	5002	2500035480
Shaker Sort	157	74580715	1526	678086943	4411	1883880905
Shell Sort	0	522790	2	1830943	4	3233258
Heap Sort	0	626040	1	2116089	2	3715159
Merge Sort	1	582176	4	1885881	7	3341199
Quick Sort	0	289762	1	941244	3	1661738
Counting Sort	0	69999	0	210001	0	332769
Radix Sort	0	121042	0	361042	0	601042
Flash Sort	1	209761	3	629850	4	911184
	100,000		**300,000**		**500,000**	
	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	1948	10000100001	15594	90000300001	46998	250000500001
Insertion Sort	184	2500064254	1615	22504077502	4528	62513409779
Bubble Sort	21578	9999462398	260709	89999770744	690434	249999667344
Shaker Sort	17804	7505122425	157098	67361562249	506279	187534120410
Shell Sort	11	6928227	35	22942134	56	39118494
Heap Sort	6	7929176	24	26141978	51	45393457
Merge Sort	16	7082776	54	23514993	70	39464525
Quick Sort	6	3467378	23	11126179	31	19703884
Counting Sort	0	632769	3	1832769	6	3032769
Radix Sort	1	1201042	3	3601042	10	6001042
Flash Sort	5	1286385	24	3794377	70	6280546

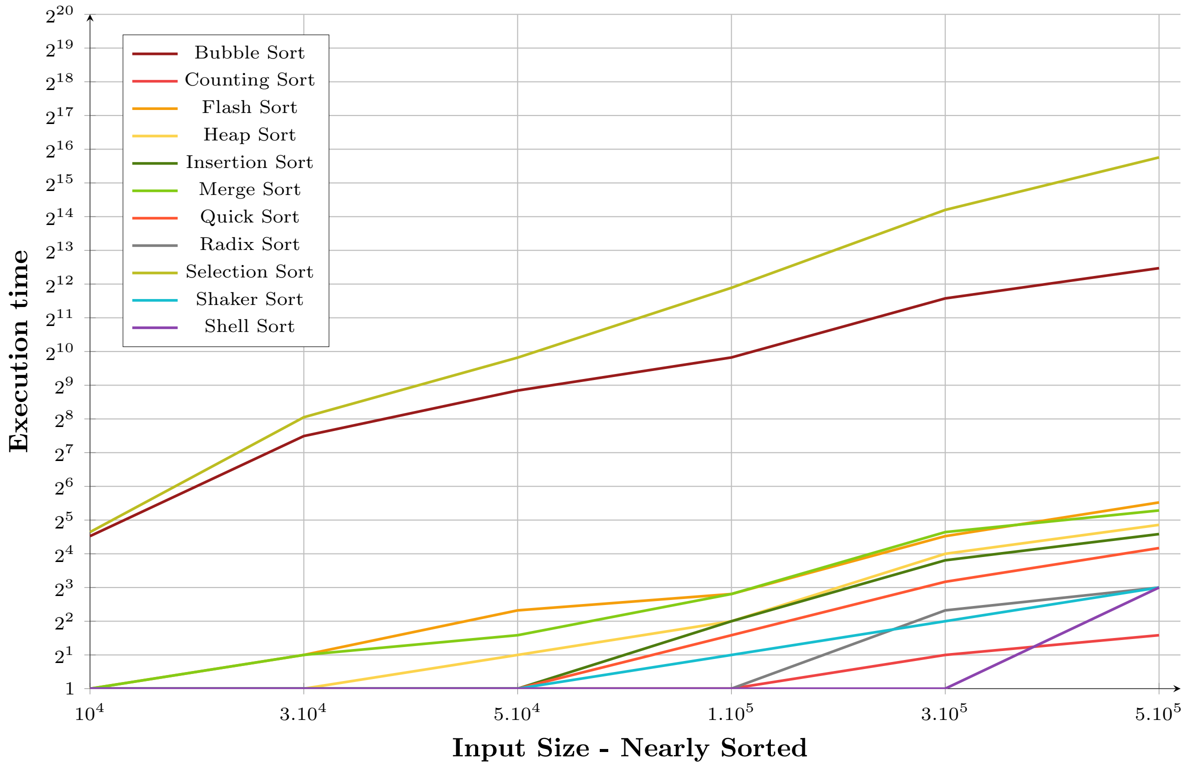
Data order: *Nearly sorted data*						
Data size	**10,000**		**30,000**		**50,000**	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	25	100010001	265	900030001	903	2500050001
Insertion Sort	0	568591	0	1950451	1	3266785
Bubble Sort	23	93545694	180	742514950	459	1876374298
Shaker Sort	0	299783	1	899783	1	1299838
Shell Sort	0	254011	0	846603	0	1357875
Heap Sort	0	655295	1	2194774	2	3854975
Merge Sort	0	506464	2	1593725	3	2791216
Quick Sort	0	244649	0	794592	1	1388677
Counting Sort	0	70001	0	210001	0	350001
Radix Sort	0	121042	0	361042	0	601042
Flash Sort	0	253696	2	761100	5	1268500
	100,000		**300,000**		**500,000**	
	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	3794	10000100001	18819	90000300001	55387	250000500001
Insertion Sort	4	6552688	14	20636462	24	35450511
Bubble Sort	906	2903638160	3051	9756424198	5681	21202346088
Shaker Sort	2	2599838	4	7799838	8	12999838
Shell Sort	1	2660117	1	8171383	8	14171383
Heap Sort	4	8220458	16	26992823	29	46722209
Merge Sort	7	5769186	25	18880550	39	31208812
Quick Sort	3	2929591	9	9381140	18	15880628
Counting Sort	0	700001	2	2100001	3	3500001
Radix Sort	1	1201042	5	3601042	8	6001042
Flash Sort	7	2537000	23	7611000	46	12685000

Data order: *Sorted data*						
Data size	**10,000**		**30,000**		**50,000**	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	15	100010001	122	900030001	359	2500050001
Insertion Sort	0	505445	1	1681187	2	2935892
Bubble Sort	0	20000	0	60000	0	100000
Shaker Sort	0	20000	0	60000	0	100000
Shell Sort	0	193070	0	653344	0	1133344
Heap Sort	0	655665	1	2194737	2	3854789
Merge Sort	1	473485	2	1507561	4	2678039
Quick Sort	0	234123	1	769003	1	1374998
Counting Sort	0	234123	1	769003	1	1374998
Radix Sort	0	121042	0	361042	0	601042
Flash Sort	1	253702	6	761102	7	1268502
	100,000		**300,000**		**500,000**	
	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	1352	10000100001	12999	90000300001	49077	250000500001
Insertion Sort	4	6221795	13	20305569	22	35119618
Bubble Sort	0	200000	0	600000	0	1000000
Shaker Sort	0	200000	0	600000	0	1000000
Shell Sort	0	2435586	1	7946852	3	13946852
Heap Sort	4	8220671	14	26992801	26	46721979
Merge Sort	8	5656009	27	18767373	37	31095635
Quick Sort	3	2831877	9	9264220	18	16033322
Counting Sort	0	700001	3	2100001	3	3500001
Radix Sort	1	1201042	5	3601042	8	6001042
Flash Sort	18	2537002	34	7611002	42	12685002

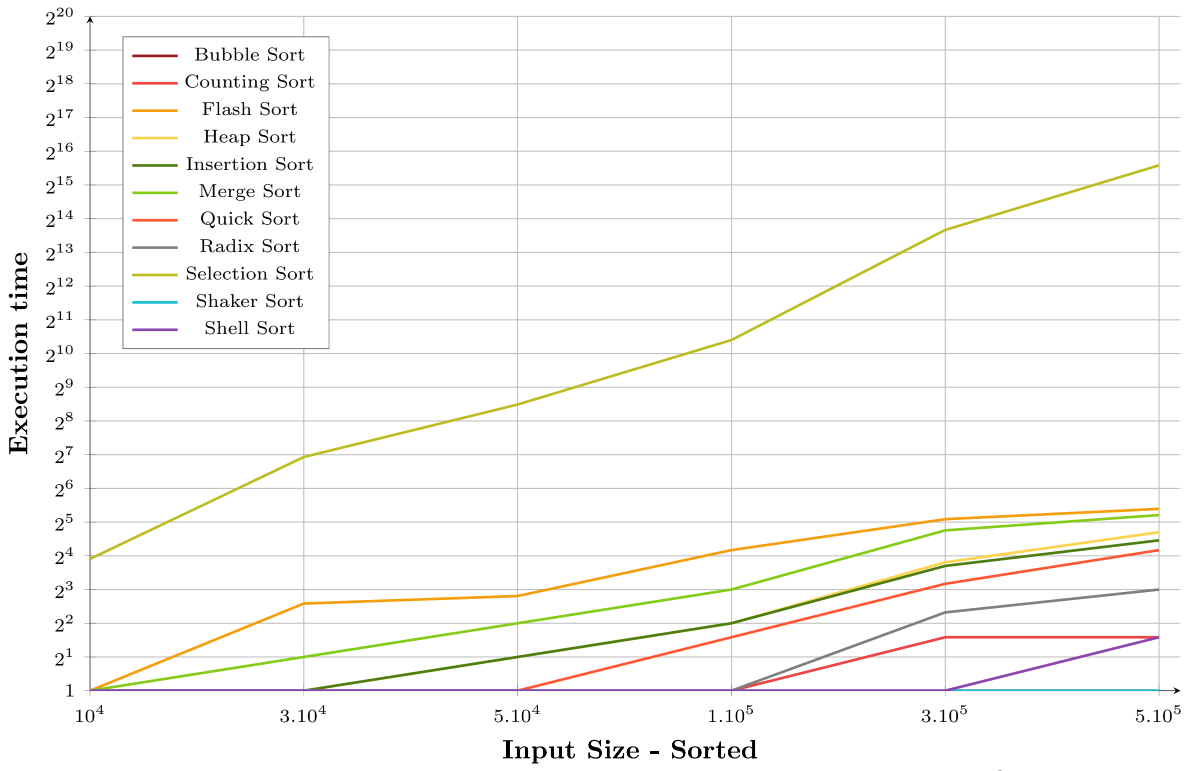
Data order: *Reversed sorted data*						
Data size	**10,000**		**30,000**		**50,000**	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	25	100010001	278	900030001	799	2500050001
Insertion Sort	3	50559448	32	451863911	92	1253262838
Bubble Sort	241	100009999	2171	900029999	6501	2500049999
Shaker Sort	280	100005000	2195	900015000	6115	2500025000
Shell Sort	0	302545	0	1055382	0	1925670
Heap Sort	0	598439	1	2037007	2	3571912
Merge Sort	0	469276	1	1519064	5	2665094
Quick Sort	0	248574	1	817021	1	1455716
Counting Sort	0	70001	0	210001	0	350001
Radix Sort	0	121042	0	361042	1	601042
Flash Sort	0	253002	3	759002	4	1265002
	100,000		**300,000**		**500,000**	
	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	3018	10000100001	26800	90000300001	70351	250000500001
Insertion Sort	358	5006925693	3168	45022652827	10099	125039152827
Bubble Sort	24282	10000099999	274323	90000299999	698190	250000499999
Shaker Sort	30807	10000050000	250556	90000150000	653257	250000500001
Shell Sort	1	3866572	4	12763939	6	21782465
Heap Sort	4	7634890	16	25313636	27	44054184
Merge Sort	8	5630120	27	18556972	37	31385074
Quick Sort	3	2941355	10	9568719	20	16248810
Counting Sort	0	700001	1	2100001	4	3500001
Radix Sort	1	1201042	10	3601042	15	6001042
Flash Sort	6	2530002	24	7590002	44	12650002



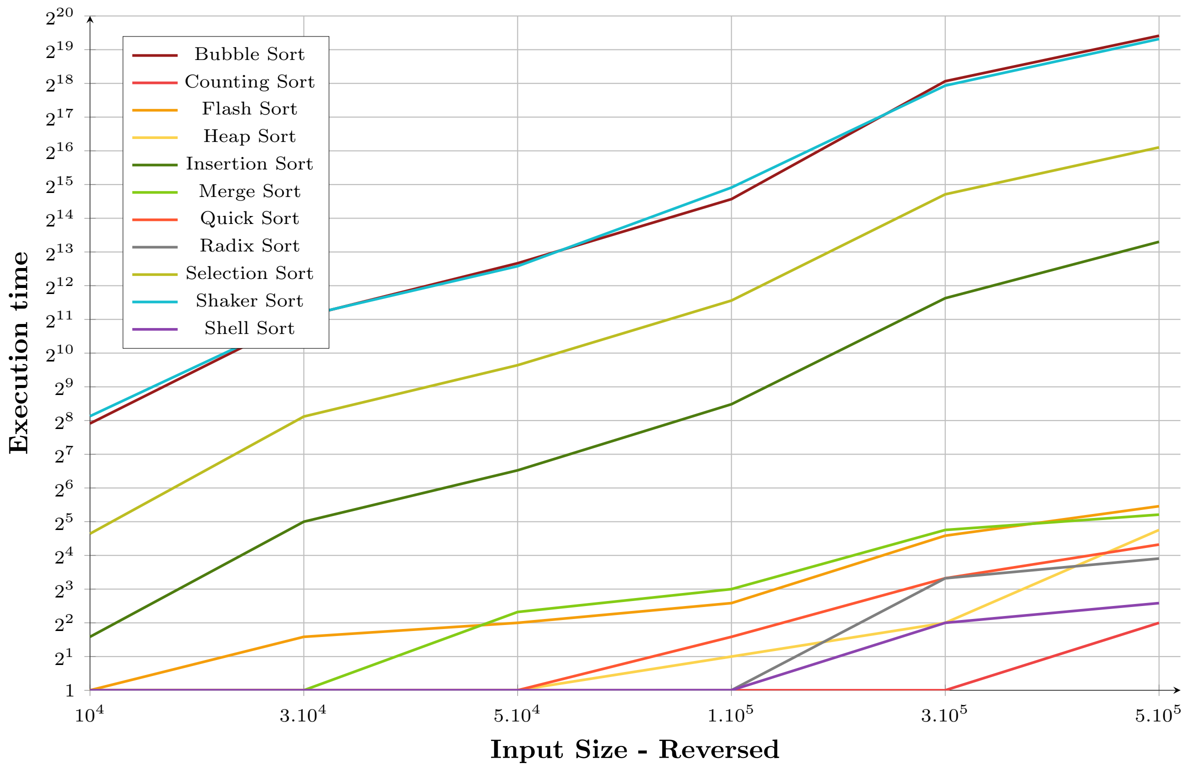
The line graphs illustrate that shaker sort and bubble sort are the two most consuming time because of $O(N^2)$ time complexity and comparison time, while quick sort and radix sort are the fastest due to $O(N \log N)$ and $O(N * K)$ time complexity with less comparison



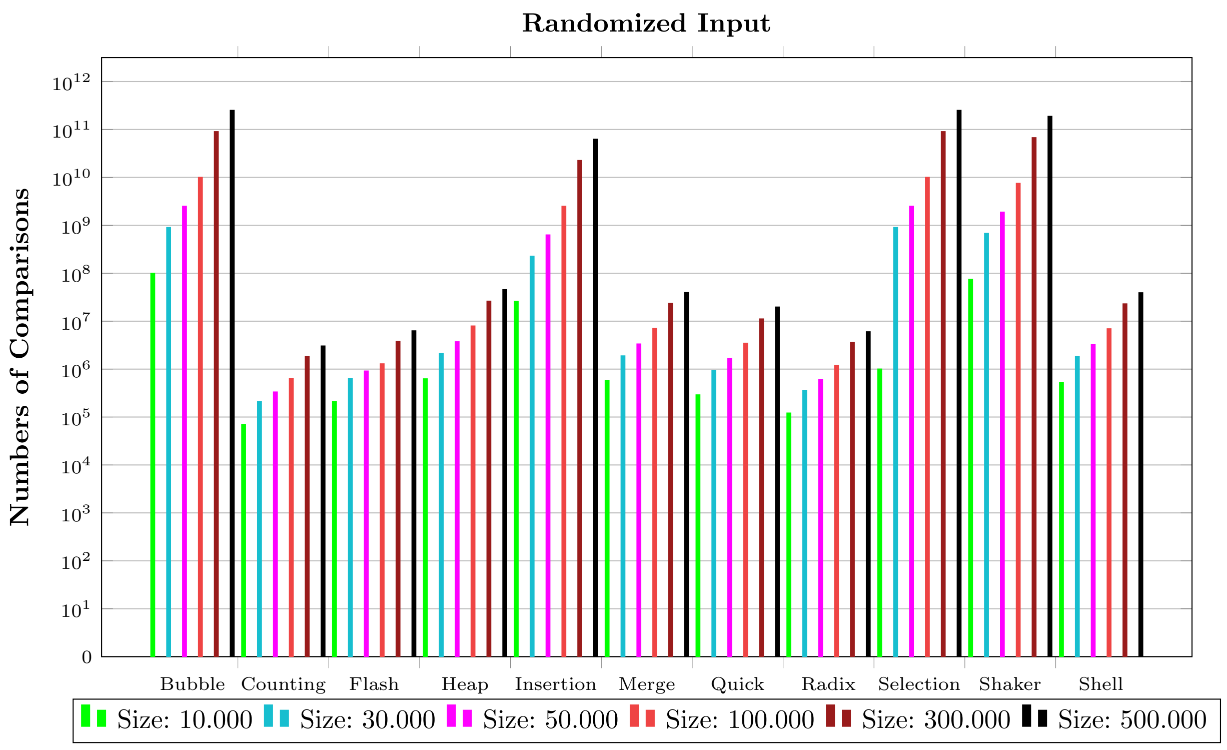
According to the chart, selection sort and bubble sort are slowest algorithms whose excution time are far exceed than the others due to the complexity of $O(N^2)$



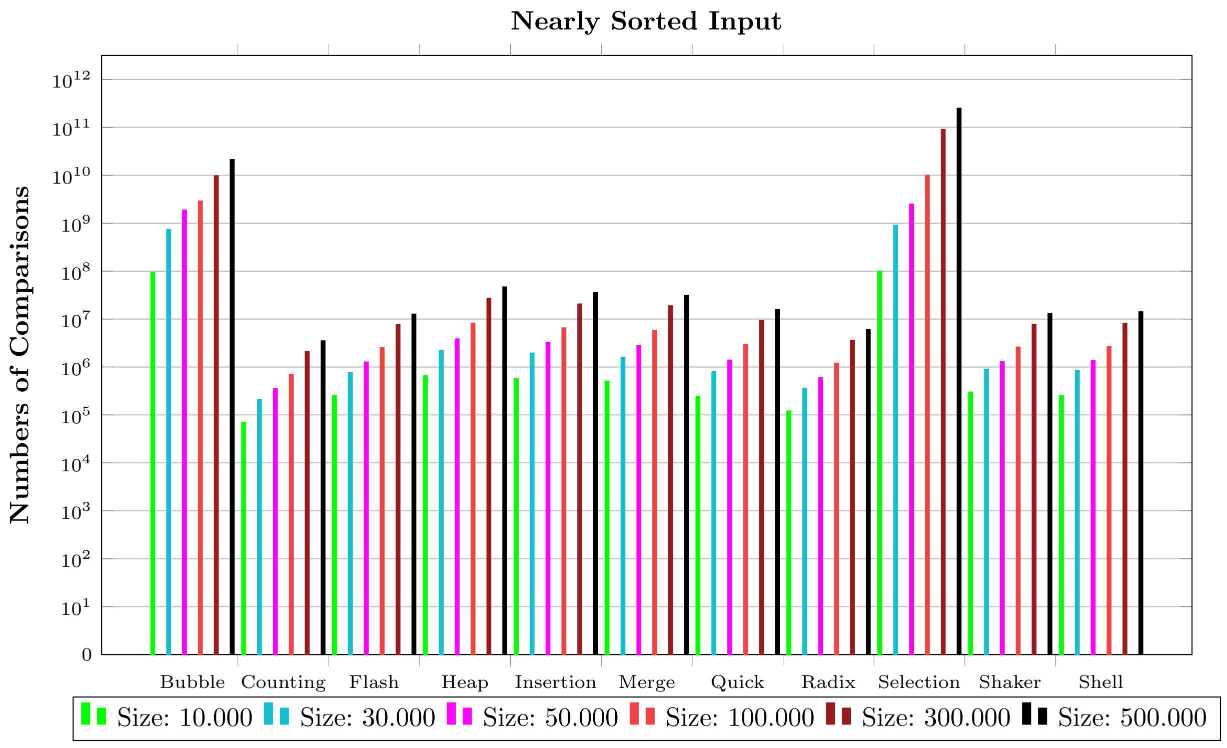
The line graphs illustrate that selection sort is the most consuming time because of $O(N^2)$ time complexity, while the others, e.g bubble sort, also have the complexity of $O(N^2)$ but they are faster due to sorted input order.



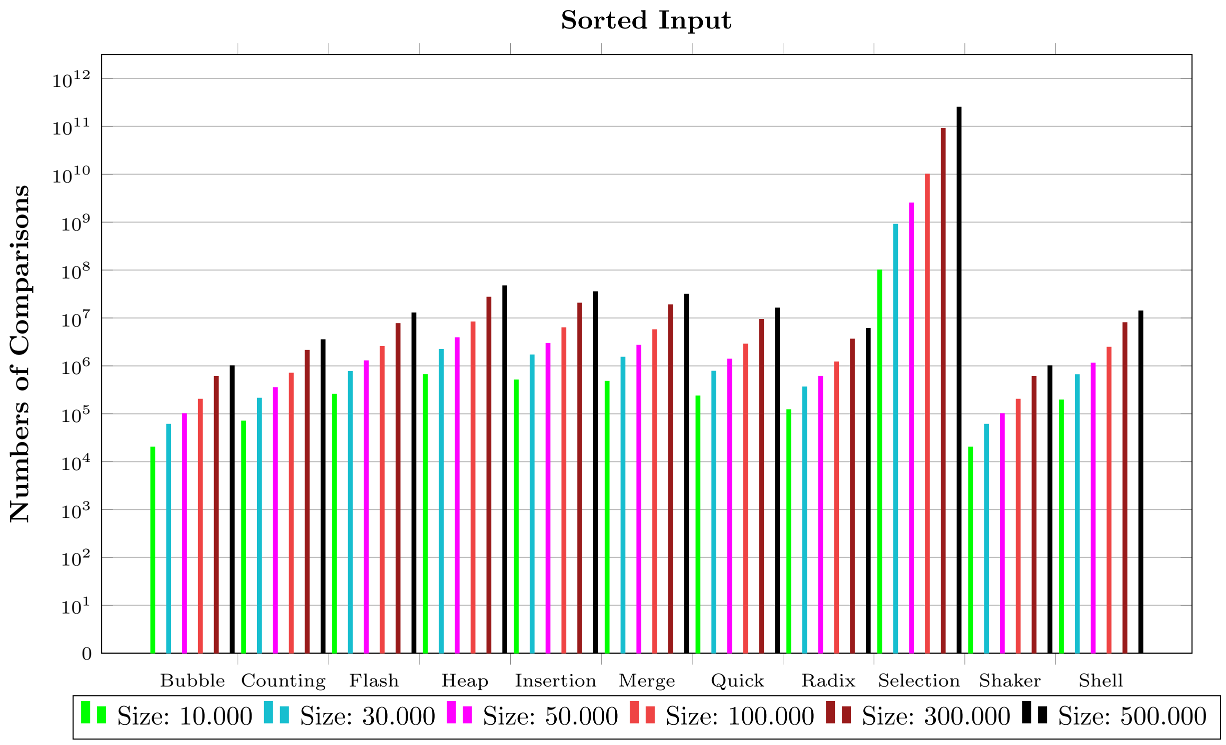
The line graphs illustrate that shaker sort and bubble sort are the two most consuming time because of $O(N^2)$ time complexity and comparison time, while quick sort and radix sort are the fastest due to $O(N \log N)$ and $O(N * K)$ time complexity with less comparison



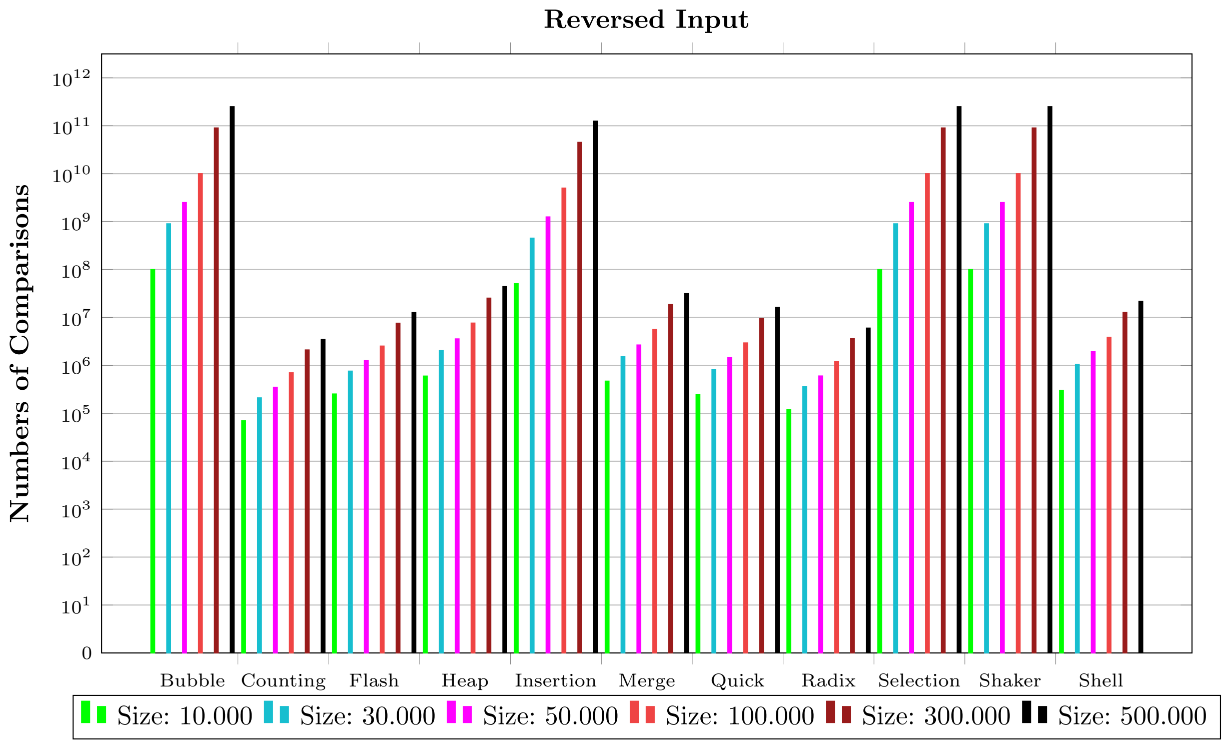
As can be seen from the bar chart, bubble sort, insertion sort, selection sort and shaker sort are ranked top, which are approximately twofold than the remaining algorithms. Meanwhile, the remaining algorithms illustrate almost the same data, especially counting sort has the smallest number of comparisons.



It can be clearly seen from the bar chart that selection sort is the most taking comparisons, which is roughly a tiny fraction further than bubble sort and it is twofold than the remaining algorithms. Meanwhile, the remaining algorithms illustrate almost the same data, especially counting sort has the smallest number of comparisons if having a closer view



According to the bar chart, it is obvious that selection sort is the most taking comparisons which is roughly twofold than the remaining. Meanwhile, the remaining algorithms illustrate almost the same data, especially bubble sort and shaker sort both have the smallest number of comparisons if having a closer view. The reason for proving this due to the input order - sorted, in which shaker sort is the better variants of bubble sort



The bar chart illustrate the number of comparisons of reversed input order. Since reversed order does not get much change compared with randomized order, the staticstics of two input order are equivalented

V. Project organization & implementation

1. File Organization

The project is divided into `docs`, `files` and `source` folders, each contains files of its category.

- `docs`: Contains the PDF version of this project report and files used to compile the PDF.
- `files`: Contains the input and exported output files throughout the execution of the program. If this folder does not exist yet, the program creates one.
- `source`: Except `main.cpp`, all C++ source files and headers are categorized into this folder.

Several source files of the `source` folder are further categorized into 2 folders:

- `sort`: Contains 11 headers of 11 sorting algorithms.
- `utility`: Contains headers for miscellaneous self-defined data types and functions, including the given `DataGenerator.cpp` source file.
- These source files are not categorized into either of those 2 folders since they serve their own purposes: `command.hpp`, `function.hpp`, `sort.hpp`, `utility.hpp`.

2. Notable implementation details

The sorting implementation takes 2 pointers *begin* and *end* then sorts the range $[begin, end)$, similarly to `std::sort`. This project heavily utilizes the use of templates, making the sorting algorithms adaptive to various data types (`int`, `float`, `std::string`) and array pointers (C-style arrays, C++ `std::vector`). Templates also enable the use of custom comparators, thus one can be passed to count comparisons without creating a separate function to achieve that purpose.

Each sorting algorithm implementation has these function overloads:

- `sort(iter begin, iter end, Compare& comp, CompareLoop& loop)`: The core implementation of the algorithms that performs the sorting. Receives additional types `Compare` to compare array elements and `CompareLoop` to specifically perform comparison in looping (`for`, `while` statements).
- `sort(iter begin, iter end, Compare& comp)`: A function overload that takes an additional custom comparator type `Compare` to compare elements. A default comparator `std::less<size_t>` is created for loop comparison and with `Compare` this is passed into the call to the first function overload. For non-comparison sorts, passing a custom comparator that changes the order of comparison may cause the algorithm to work unexpectedly.
- `sort(iter begin, iter end)`: A function overload that takes only 2 pointers to the sorting range of the array. A default comparator `std::less<Type>` (with `Type` is the value type of the pointer) is created for comparing elements and passed into the call to the second function overload.

Note that the keyword `sort` is used generically for the function name of 11 sorting algorithms: `bubbleSort`, `countingSort`, `flashSort`, `heapSort`, `insertionSort`, `mergeSort`, `quickSort`, `radixSort`, `selectionSort`, `shakerSort` and `shellSort`.

A few utility data structures are created to support algorithm measuring and simplify the code:

- `Counter`: A simple type that has a functor to do comparison. Every time it is called, it increases an inner counting variable then returns the comparison result. With templates many data types can be used for counter, so a `Counter<int>` is used to count element comparisons and a `Counter<size_t>` for looping comparisons.
- `Timer`: A simple structure that times the duration of some actions. Call `start()` to start timing and `get()` to receive the elapsed time in milliseconds since the last `start()` is called.

The random functions are implemented in the `random.hpp` header for simpler syntax and a better random engine `std::mt19937_64`. The `DataGenerator.cpp` still uses the old `rand()` function as provided by the lecturers.

These C++ standard libraries are used throughout the implementation: `<unordered_map>`, `<random>`, `<chrono>`, `<climits>`, `<iterator>`, `<utility>`,`<string>`, `<iostream>`.

References

[1] https://www.sitepoint.com/best-sorting-algorithms/.

[2] https://logicmojo.com/insertion-sort-problem.

[3] https://codedamn.com/news/algorithms/divide-conquer-merge-sort-quick-sort.

[4] https://www.altcademy.com/blog/quick-sort/.

[5] https://unstop.com/blog/competitive-coding-questions-with-solutions.

[6] https://justinriggio.com/blog/sorting-algorithms-guide/.

[7] https://the-algorithms.com/algorithm/radix-sort.

[8] https://www.simplilearn.com/tutorials/data-structure-tutorial/shell-sort.

[9] https://www.programiz.com/dsa/counting-sort.

[10] https://www.geeksforgeeks.org/radix-sort-vs-bucket-sort/.

[11] https://www.geeksforgeeks.org/comparison-among-bubble-sort-selection-sort-and-insertion-sort/.

[12] https://www.programiz.com/dsa/heap-sort.

[13] https://en.wikipedia.org/wiki/Flashsort.

[14] https://condor.depaul.edu/ichu/csc383/notes/notes2/sorting.pdf.

[15] https://daynhauhoc.com/t/cau-truc-du-lieu-va-giai-thuat-giai-thuat-sap-xep-rung-lac-shaker-sort/126359.

[16] https://www.programiz.com/dsa/heap-sort.

[17] https://dasarpai.com/dsblog/sorting-algorithm-a-summary.

[18] https://codedamn.com/news/algorithms/sorting-algorithms-comparison.

[19] https://edurev.in/t/83448/2--Sorting--Algorithms--GATE.

[20] https://www.webdatapandas.com/solution/0004%20-%20sorting_algorithm/index.html.

[21] https://irp-cdn.multiscreensite.com/58b08bff/files/uploaded/renazavisidisorunuseviti.pdf.

[22] https://www.cnblogs.com/papering/p/6038382.html.

[23] Codelearn. https://codelearn.io/.

[24] cplusplus.com. https://cplusplus.com/.

[25] GeekforGeeks. https://www.geeksforgeeks.org/.

[26] ChatGPT: Model GPT-3.5. https://chat.openai.com/.

[27] hoanghimself. Free HCMUT Report Template on LaTex. https://www.overleaf.com/project/6474d6411a2955d0bf8b8cd4.

[28] Tran Hoang Quan. Free HCMUS Report Template on LaTex. https://www.overleaf.com/latex/templates/hcmus-report-template/zyrhmsxynwqs.

[29] VNOI Wiki. https://vnoi.info/wiki/Home.

[30] Wikipedia. https://vi.wikipedia.org/wiki/.