HO CHI MINH CITY - UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

LAB 3: SORTING

Under the instructions of:

Nguyen Ngoc Thao	Lecturer
Tran Thi Thao Nhi	Instructor
Bui Huy Thong	Instructor

Performed by students:

Van Ba Duc Kien	22127218	22CLC10
Nguyen Anh Kiet	22127220	22CLC10
Vo Viet Long	22127252	22CLC10
Nguyen Le Thanh Duy	22127474	22CLC10

Ho Chi Minh City, 2023

Contents

1) Introduction

Arranging is a simple concept that takes place everywhere at anytime in this world. It refers to arranging a series of items based on a criteria into a desired sequence - ascending or descending, e.g. ordering a line of people according to their height from the shortest to the tallest, in order to facilitate other executions which function best on ordered items .

This "arranging" concept plays an primary role in terms of information and data. In Computer Science and Information Technology, arranging data to an ordered sequence is called "sorting". Sorting is a common operation in many applications, thus there have been numerous sorting algorithms developed in response to the the need of ordering complex and great sets of data.

2) In this report

In this report for Lab 3: Sorting, we make efforts to demonstrate these following sorting algorithms with regard to theory, implementation in form of pseudo-code and statistical demonstration as graph:

Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort.

3) Selection Sort

3.1) Theory

3.1.1) Idea:

Selection sort is a simple and effiecient sorting algorithm based on finding the largest value in the array repeatedly. This algorithms run consecutively on the unsorted portion of the array to find the biggest/smallest value (according to wanted descending/ascending order) and move it to the sorted part of the array.

3.1.2) Time and space complexity:

- **Time complexity**: For an array of size **N**, this algorithm has the **big O notation** of $\boldsymbol{N^2}$ and it has two nested loop:
  - One loop to select an element of Array one by one = $\boldsymbol{O(N)}$
  - Another loop to compare that element with every other Array element = $\boldsymbol{O(N)}$
  - Therefore overall complexity is $\boldsymbol{O(N) * O(N) = O(N * N) = O(N^2)}$
- **Space complexity**: The space complexity of this algorithm is O(1) as the only extra variable it uses is the variable to hold the value of the currently processed element and Selection Sort never exceeds **N** swaps. Therefore, this algorithm is very helpful in the case when space writing can be costly.

3.2) Implementation

In this pseudo code, we implement the Selection Sort to sort an array of size **N**, the first element lies at index 1, in the ascending order:

- **Function Selection_Sort(array of size N):**
  - Step 1: Set the increment variable **i = 1**
  - Step 2: Find the smallest value in the array start from **i** to **N**:
    - Swap with the element at **i**
    - Increase **i** by 1
  - Step 3: If the **i** has not reached **N (i < N)**, repeat from step 1, other wise, stop the function.

4) Insertion Sort

4.1) Theory

4.1.1) Idea:

Insertion sort is a simple sorting algorithm and its concept is like how we sort our playing cards. This algorithm treats the array as two parts - sorted half and unsorted half, it continuously picks the first element of the unsorted half of the array and places it in the right position in the sorted half until there is no element left in the unsorted half.

4.1.2) Time and space complexity:

- **Time complexity:**
  - The worst-case time complexity of the Insertion sort is $\boldsymbol{O(N^2)}$ when the array is sorted in reversed.
  - The average case time complexity of the Insertion sort is $\boldsymbol{O(N^2)}$
  - The time complexity of the best case is $\boldsymbol{O(N)}$ when the array is sorted.
- **Space complexity:** The auxiliary space complexity of Insertion Sort is $\boldsymbol{O(1)}$

4.2) Implementaion

In this pseudo code, we implement the Insertion Sort to sort an array of size **N**, the first element lies at index 1, in the ascending order:

- **Function Insertion_Sort(array of size N):**
  - Step 1: Assume that the unsorted half starts from index 2, set **i = 2** and create a variabele **v** to hold value of element at **i**
  - Step 2: Assign value at **i** to **v**.
  - Step 3: Compare **v** with value of elements at **index** from **i - 1** to **1**:
    - If the value at i of that element is greater than **v**, move it onẻ position up.
    - Decrease index by one, continue the loop until meet the leftmost end (there is no more element to compare) or meet a element smaller than **v**.
  - Step 4: Assign value of **v** to the position **index + 1**, increase **i** by 1.
  - Step 5: Check **i <= N**:
    - If true, it means unsorted half still has elements, repeat from step 2.
    - Otherwise, stop the function.

Insertion Sort has another improved version when Binary Sort is applied to find the place of the unordered element of the second half in the first ordered half of the array. This improved version of Insertion Sort saves a great amount of comparison. The following pseudo code is how Binary Insertion Sort is implemented:

Assume that we have Function **BinarySearch(Array of size N, left, right (for range of searching))** that returns the index of correct position.

- **Function Binary_Insertion_Sort(array S of size N):**
  - Step 1: Assume that the unsorted half starts from index 2, set **i = 2** and create a variabele **v** to hold value of element at **i**
  - Step 2: Assign value at **i** to **v**.
  - Step 3: Create pos to hold index returned from BinarySearch(S,1,i)
  - Step 4: For **j = i**, if **j >= pos + 1**:
    - **S[j]** = **S[j-1]**
    - Decrease **j** by **1**, check if **j >= pos + 1**, continue the loop if yes, break otherwise.
  - Step 5: Assign value of **v** to index **pos**
  - Step 6: Check **i <= N**:
    - If true, it means unsorted half still has elements, repeat from step 2.
    - Otherwise, stop the function.

5) Bubble Sort

5.1) Theory:

5.1.1) Idea:

Bubble Sort is one of the simpliest sorting algorithms. This algorithm traverses throughout the array of items and repeatedly swap two adjecent elements if they are in the wrong order, after each pass, an element is brought to its rightful position. This operation is re-executed until there are no two items in unordered positions.

5.1.2) Time and space complexity:

- **Time Complexity**: For an array of size **N**, Bubble Sort has an time complexity as $\mathbf{O}(\boldsymbol{N}^2)$ because it has two nested loops:
  - The outer loop indicates the number of passes to traverse through out the array of **N** items.
  - The inner loop is used to execute traversals with comparisons and swaps (if needed) for each adjecent items.
- We can improve the best case with a boolean variable to check whether there is no swap executed in the inner loop. With this improvement, the best case complexity is $O(N)$.
- **Space Complexity**: $O(1)$, this algorithm does not require more space used for additional arrays or the likes.

5.2) Implementaion

In this pseudo code, we implement the Bubble Sort to sort an array of size **N**, the first element lies at index 1, in the ascending order:

- **Function Bubble_Sort(array of size N):**
  - The outer loop counts the traversals as **i** from **1** to **N - 1**:
    - The inner loop traverses the array as index **j** from **N** to **i + 1**:
      - Check if the $\mathbf{j_{th}}$ element being processed is smaller than the $(\boldsymbol{j}-\mathbf{1})^{\mathbf{th}}$ element (in wrong order), if yes, swap those two element.
      - Decrease **j** by one, check if **j >= i+1**, if yes, continue the inner loop.
    - Increase **i** by one, check if $\boldsymbol{i} > \boldsymbol{N}-\mathbf{1}$, if yes, stop the function, else continue the outer loop

This algorithm can be improved by creating a boolean **isSwap** variable to check whether there is no swap executed in any passes of the outer loop, if isSwap is true, we end the algorithm immediately. This version of Bubble Sort has the best case of ($\boldsymbol{O}(\boldsymbol{N})$ if the first pass has no swap.The improved Bubble Sort can be implemented as the following pseudo code:

```
• Function Improved_Bubble_Sort(array of size N):
    • Create boolean variable ifSwap
    • The outer loop counts the traversals as i from 1 to N-1 passes:
        • Set isSwap to false
        • The inner loop traverses the array as index j from N to i + 1:
            • Check if the j_th element being processed is smaller than the $(j-1)^{th}$ element (in wrong order), if yes, swap those two element and set isSwap to true.
            • Decrease j by one, check if $j \geq i+1$, continue the loop.
        • Check if isSwap is true, if yes, stop the function.
        • Increase i by one, check if $i > N-1$, if yes, stop the function, else continue the outer loop
```

In the programming part, we choose the **Improved Shaker Sort** to implement.

6) Shaker Sort

6.1) Theory

6.1.1) Idea:

Shaker Sort is a variant of Bubble Sort that we have demonstrated above. Basically, Shaker Sort is Bubble Sort but in a pass of the outer loop, we run two traversals of opposite directions to sort unsorted array. Shaker Sort is also called Cocktail Sort as a metaphor for how the unordered items move in this algorithm.

6.1.2) Time and space complexity:

Because Shaker Sort is primarily another form of Bubble Sort, this algorithm still has the similar time and space complexity with Bubble Sort:

- **Time Complexity**: For an array of size **N**, Shaker Sort has an time complexity as O(N^2).
- **Space Complexity**: $O(1)$, this algorithm does not require more space used for additional arrays or the likes.

6.2) Implementaion

Another difference of Shaker Sort compared with Bubble Sort is that it uses additional variables **end and start** to regulate the two opposite traversals in each pass.

In this pseudo code, we implement the Shaker Sort to sort an array of size **N**, the first element lies at index 1, in the ascending order:

```
• Function Shaker_Sort(array of size N):
    • Create end and start, set N to end and set 1 to start.
    • The outer loop: Check if end ≤ start, if yes, do the following steps:
        • The first inner loop: For index i running from start to end - 1:
            • If $i^{th}$ element is larger than the $(i+1)^{th}$ element, swap these two elements
            • Increase i by one, check if $i <$ end $-1$, if yes, continue , else break the first inner loop.
        • Decrease end by one
        • The second inner loop: For index i running from end - 1 to start:
            • If $i^{th}$ element is larger than the $(i+1)^{th}$ element, swap these two elements
```

    - Decrease **i** by one, check if $\boldsymbol{i} \geq$ **start**, if yes, continue , else break the second inner loop.
  - Increase **start** by one, back to checking the condition in the outer loop.

This algorithm can also be improved in the same way as we mentioned in Bubble Sort by using a boolean variable:

- **Function Improved_Shaker_Sort(array of size N):**
  - Create **end** and **start**, set **N** to **end** and set **1** to **start**.
  - Create boolean variable **isSwap**, set **isSwap** to **true** (assume that there will be swaps)
  - The outer loop: Check if **isSwap** is **true**, if yes, do the following steps:
    - Set **isSwap** to **false**
    - The first inner loop: For index **i** running from **start** to **end - 1**:
      - If $\boldsymbol{i}^{\mathbf{th}}$ element > $(\boldsymbol{i}+\mathbf{1})^{\mathbf{th}}$ element, swap these two elements and set **isSwap** to **true**.
      - Increase **i** by one, check if **i** < **end - 1**, if yes, continue , else break the first inner loop.
    - Check if **isSwap** is **false**, stop the function here if yes.
    - Set **isSwap** to **false**
    - Decrease **end** by one
    - The second inner loop: For index **i** running from **end - 1** to **start**:
      - If $\boldsymbol{i}^{\mathbf{th}}$ element > $(\boldsymbol{i}+\mathbf{1})^{\mathbf{th}}$ element, swap these two elements and set **isSwap** to **true**.
      - Decrease **i** by one, check if $\boldsymbol{i} \geq$ **start**, if yes, continue , else break the second inner loop.
    - Increase **start** by one.
    - Check if **isSwap** is **false**, stop the function if **yes**, continue the loop otherwise.

In the programming part, we choose the **Improved Bubble Sort** to implement.

7) Shell Sort

7.1) Theory

7.1.1) Idea:

Shell Sort can be considered as a variant of Insertion Sort, it can surmount the weakness of Insertion Sort when the unordered element has to move far away from its current positions and costs servere amount of movements and comparisons. The idea of Shell Sort allow the swaps of far items by sorting the array around a gap **h**. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h_{th} element are sorted.

7.1.2) Time and Space complexity:

- **Time Complexity**: Time complexity of the Shell sort is $\mathbf{O}(\boldsymbol{N}^{\mathbf{2}})$**.**
- **The space complexity** of the shell sort is $\mathbf{O(1)}$**.**

7.2) Implementaion

In this pseudo code, we implement the Shell Sort to sort an array of size **N**, the first element lies at index 0, the last element is at N - 1, in the ascending order:

```
• Function Shell_Sort(array S of size N, N):
  • Let gap $h = \frac{N}{2}$, if $h > 0$:
    • For $i = h$; if $i < n$:
      • Create temp to hold value of element at i.
      • For $j = i$, check if $j >= h$ and $S[j - h] > temp$, if true:
        • Let $S[j] = S[j - h]$ (swap the higher value to the right of temp's precise position).
        • Decrease j by h, check if $j >= h$, continue the loop if true, break other wise.
      • Let $S[j] = temp$.
      • Increase i by 1, check if Keiweik, continue the loop if true, break otherwise.
    • Decrease h by half, check if $h > 0$, if true, continue the loop, break otherwise.
```

8) Heap Sort

8.1) Theory

8.1.1) Idea:

Heap Sort is a comparison-based algorithm that relies on Binary Heap data structure. Generally, it is similar to Selection Sort as we find the largest/smallest element in the array and then place it at the end. This operation is repeated on the remaining elements.

In order to find the largest or smallest element in the array, we create a max heap or a min heap, this data structure will make sure that the biggest/smallest element will always be at the first element of the array.

8.1.2) Time and space complexity:

- **Time Complexity:** This algorithm has time complexity of $\boldsymbol{O(N \log_2 N)}$.
- **Auxiliary Space:** $\boldsymbol{O(1)}$. Heap Sort is an in-place sorting algorithm.

8.2) Implementaion

Usually, Heap Sort has a subfunction that runs recursively called **heapify** or **HeapRebuild** to build and update the **Binary Heap**.

In this pseudo code, we implement the Heap Sort to sort an array of size **N**, the first element lies at index 1, in the ascending order by building a **Max Heap**:

```
• Function heapify(array of size N, N, index K ):
  • Find the left child and right child index of element K, the indexes of its left child and right child are consequently $2K + 1$ and $2K + 2$.
  • Verify if each child exist, if a child's index is bigger than the size of the array, it do not exist.
  • Find the child node containing larger value, if that child's value is larger than its father node, swap their values and recursively call the function heapify starting from that child.
```

- **Function Heap_Sort(array of size N, N):**
  - We first build a heap by using a loop with an index **i** from $\frac{N-1}{2}$ to **1**, call **heapify(array of size N, i)**
  - Create variable **heapSize**, set **heapSize = N**
  - Use a while loop with condition (**heapSize** > 0):
    - Decrease **heapSize** by one
    - Swap the first element with the last element inside the heap.
    - Call **heapify(array of size N, heapSize, 0)** to maintain the heap.

9) Merge Sort

9.1) Theory

9.1.1) Idea:

Merge Sort is a recursive and comparison-based algorithm that works simply by dividing the array into two subarrays, sorting two subarrays and merging them together. Usually Merge Sort is recalled by itself until the divided subarrays contain only one element.

9.1.2) Time and space complexity:

- **Time complexity:** $O(N \log_2 N)$ The time complexity of Merge Sort is $O(N \log_2 N)$ as merge sort always divides the array into two halves and takes linear time to merge two halves.
- **Auxiliary Space: O(N),** In merge sort when merging two subarrays, all elements are put into an additional array and then is copied back into the original array. So N auxiliary space is required for merge sort.

9.2) Implementaion

As mentioned above, Merge Sort is a recursive sorting algorithm and it usually has two functions for inplementation, which are the **merge()** and **MergeSort()**.

In this pseudo code, we implement the Merge Sort to sort an array of size **N**, the first element lies at index 1, in the ascending order:

- **Function merge(array A of size N, left, right (to indicate the range of the subarray))**:
  - This array **consists of two smaller subarray**, from the **leftmost element to the middle element** and the rest of the array. Both of these smaller subarrays are already **sorted**.
  - We need to **merge** these two smaller subarrays to create a sorted array from the leftmost element to the rightmost element. See the following pseudo code to know how to implement this idea:

```
Create a new temporary array B with right - left + 1 elements.
let left_index = 0, right_index = 0

For (i = 0; i < size of the subarray; i++):
  if ((right_index is out of the right part range) or
      (left_index is in the left part range
       and array[left_index] > array[right_index])
     ):
      B[i] = array[left_index]
      left_index++
  else:
      B[i] = array[right_index]
      right_index++

Copy the content of the temporary array B to the original array
```

- **Function Merge_Sort(array A of size N, left, right (to indicate the range of the subarray))**:
  - Check if the number of elements of the subarray is larger than 1 (or left < right), if yes:
    - Divide the subarray from left to right into two smaller subarrays
    - Let **middle** = $\frac{\textbf{right + left}}{\textbf{2}}$
    - Recursively call Merge_sort for the first half and second half:
      - **Merge_Sort(A, left, middle)**
      - **Merge_Sort(A, middle + 1, right)**
    - Call **merge(A, left, right)** to merge the sorted subarray.

10) Counting Sort

10.1) Theory

10.1.1) Idea:

Counting sort is a non-comparison sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (a kind of hashing). Then do some arithmetic operations to calculate the position of each object in the output sequence. Usually, for each element, its final position in the array is determined by the numbers of smaller elements.

10.1.2) Time and space complextity:

- Time complexity: $\boldsymbol{O(N+R)}$ where N is the number of elements in the input array and R is the range of input.
- Space Complexity: $\boldsymbol{O(R)}$ where R is the range of input.

10.2) Implementaion

There is two versions of counting sort, the first version requires calculating the prefix sum of the counting array and use an extra array of size N to temporarily store the sorted array. The second version is much more simple and efficient without the need of calculating the prefix sum and N extra auxiliary space.

We chose the second version of the counting sort to implement in order to have a better diversity in the code but the first version of counting sort is used inside of both of our radix sort and flash sort.

In this pseudo code, we implement the Counting Sort to sort an array of size **N**, the first element lies at index 0 and the last lies at $\boldsymbol{N-1}$, in ascending order:

```
• Function Counting_Sort(array A of size N)
    • Find the maximum value of A.
    • Create a counting array C of size max + 1, every element of this array is set to 0,
       we assume that value in that array is in range [0 ... max] and S2 is used to store the
      occurances of each element.
    • Use a for loop from $i = 0$ to $N - 1$:
        • Increase C[value of A[i]] by one.
    • After the previous loop, array C's elements contain how many times a value exist inside
      of the original array.
    • Use a for loop with $i = 0$ and index = 0, while $i \leq$ max and index < N:
        • While (C[i] > 0):
            • A[index] = i
            • Increment index and i by one
```

11) Quick Sort

11.1) Theory:

11.1.1) Idea:

Quick Sort is a recursive sorting algorithm that bases on the Divide and Conquer algorithm and revolves around a value called pivot. This algorithm picks an element in the array to be the pivot and starts making partitions of the array around the pivot by placing it in the correct position.

The key process in quickSort is the partition process. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot (the desired order is ascending order).

11.1.2) Time and Space complexity:

- Time complexity: This algorithm has an average case of $\mathbf{O}(\boldsymbol{N} * \mathbf{log}(\boldsymbol{N}))$ and worst case of $\mathbf{O}(\boldsymbol{N}^{\mathbf{2}})$. The worst case happens when the pivot is chosen poorly, e.g. always the largest element or the smallest element.
- Space complexity: $\mathbf{O}(\mathbf{1})$ if the recursive stack space is not considered.

11.2) Implementation:

There are couples of ways to pick an element to be **pivot**: the last element, the first element, a random element or the middle element. Quick Sort can also be implemented in many ways. In three kinds of Quick Sort introduced to us in the course of Data Structures and Algorithms, we chose the first version to implement where the chosen element is the middle one.

In this pseudo code, we implement the Quick Sort to sort an array of size **N**, the first element lies at index 0 and the last lies at $N - 1$, in ascending order:

Note: left and right indicate the start, end or the size of the array in process

- **Function Quick_Sort(array S of size N, left, right)**
  - Pick the middle element to be the pivot $(\mathbf{middle} = \frac{\mathbf{right} - \mathbf{left}}{\mathbf{2}})$
  - Create leftindex, rightindex to traverse the array rightward and leftward
  - While **leftindex <= rightindex**:
    - Keep traversing **rightward** and increasing **leftindex** by one until find an element that is **greater** than **pivot**
    - Keep traversing **leftward** and decreasing **rightindex** by one until find an element that is **smaller** than **pivot**
    - Check if **leftindex <= rightindex**, if yes:
      - Swap those elements at **leftindex and rightindex**
      - Decrease **rightindex** by one, increase **leftindex** by one.
    - Check the while loop condition, continue if true, break if false.
  - If **rightindex > left**:
    - Call **Quick_Sort(N, left, rightindex)**
  - If **leftindex < right**:
    - Call **Quick_Sort(N, leftindex, right)**

12) Radix Sort

12.1) Theory

12.1.1) Idea:

Radix Sort is another non-comparison sorting algorithm. Instead of comparing the array's entries, this algorithm form groups of these items by processing them digit by digit. This algorithm works best on intergers or strings with fixed-sized keys.

Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit's value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant or reversed, Radix Sort achieves the final sorted order.

12.1.2) Time and space complexity:

- **Time Complexity:**

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping the keys by the individual digits which share the same significant position and value. It has a time complexity of $\boldsymbol{O(d * (n + b))}$, where d is the number of digits, n is the number of elements, and b is the base of the number system being used.

- **Space Complexity:**

Radix sort also has a space complexity of $\boldsymbol{O(n + b)}$, where n is the number of elements and b is the base of the number system. This space complexity comes from the need to create buckets for each digit value and to copy the elements back to the original array after each digit has been sorted.

12.2) Implementaion

We implemented the LSD Radix Sort, which sorts the elements according to their least significant digit to their most significant digit.

In this pseudo code, we implement the Radix Sort by processing the digits of the arrays from the rightmost to leftmost digits to sort an array of size **N**, the first element lies at index 0 and the last lies at $N-1$, in ascending order:

- **Function Radix_Sort(array A of size N):**
  - Find the maximum value in array A and store in max.
  - For $\mathbf{k = 10^{h}}$, **h** is less or equal to the number of digits of the maximum value in array A:
    - Sort all the number according to the $(\boldsymbol{h}+\mathbf{1})^{\mathbf{th}}$ digit from right to left, we use the prefix sum version of counting sort to do this.
    - Create array **Bucket** of size 10, each element $\mathbf{Bucket}_{\boldsymbol{i}}$ represents the number of elements in the array A that have their $(\boldsymbol{h}+\mathbf{1})^{\mathbf{th}}$ digits from right to left equal to $\boldsymbol{i}$. We first set all of this array's elements to naught.
    - Traverse the original array and increment the elements of the array Bucket. Each element $\boldsymbol{A_i}$ will cause $\mathbf{Bucket}_{\boldsymbol{j}}$ to increase with $\boldsymbol{j} = \frac{\boldsymbol{A_i}}{\boldsymbol{k}} \bmod \mathbf{10}$.
    - Turn the Bucket array into a **prefix sum array** that will now give new indexes to the original array's elements after sorting them by their $(\boldsymbol{h}+\mathbf{1})^{\mathbf{th}}$ digits. We can achieve this by using a for loop and assign $\mathbf{Bucket}_{\boldsymbol{i}} = \mathbf{Bucket}_{\boldsymbol{i}} + \mathbf{Bucket}_{\boldsymbol{i-1}}$ with $\mathbf{1} \le \boldsymbol{i} \le \mathbf{9}$
    - Sort the array by their $(\boldsymbol{h}+\mathbf{1})^{\mathbf{th}}$ digits from right to left using an temporary array B, with $\mathbf{0} \le \boldsymbol{i} \le \boldsymbol{N}$, each element $\boldsymbol{A_i}$ will be assigned to element $\boldsymbol{B_m}$ with $\boldsymbol{m} = \mathbf{Bucket}_{\boldsymbol{j}}$ with $\boldsymbol{j} = \frac{\boldsymbol{A_i}}{\boldsymbol{k}} \bmod \mathbf{10}$ and then decrement $\mathbf{Bucket}_{\boldsymbol{j}}$ by one.
    - Copy the temporary array B back into the original array A.

13) Flash Sort

13.1) Theory

13.1.1) Idea:

Flash sort is a unstable sort that is very similar to counting sort but instead of counting how many copies of a value are in the array, we count how many elements in the array fall into a range of values (also known as bucket or class) to reduce space complexity. Then, an efficient in-place algorithm is used to "sort" the buckets and then use any other sorting algorithm or call flash sort recursively to sort the elements inside each bucket.

13.1.2) Time and space complexity

- **Time complexity:**
  - The **worst case** time complexity of flash sort is $\boldsymbol{O}(\boldsymbol{N}^{\mathbf{2}})$.
  - The **average case** time complexity of flash sort is $\boldsymbol{O}(\boldsymbol{N})$.
  - The **best case** time complexity of flash sort is $\boldsymbol{O}(\boldsymbol{N})$.

13.2) Implementation

- **Function Flash_Sort(array A of size N):**
  - Perform one pass through the array to find the **minimum value** and the **index of the maximum value**.
  - Divide the range $[A_{\min}, A_{\max}]$ into **M** buckets. Create an array **B** with size **M** with each element contains the number of elements fall into that bucket. **M** can be a predefined constant or can be calculated using the size of the array or the range of the values.

- Perform one pass through the array, we can know which bucket an element A_i should be in by using the formula below and increment B_j by one for each A_i.

$$\boldsymbol{j = \frac{(M-1)(A_i - A_{\min})}{A_{\max} - A_{\min}}}$$

- Perform one pass through the bucket array starting from the second element and set $\boldsymbol{B_i = B_i + B_{i-1}}$.This converts the counts of elements in each bucket to a prefix sum, now **B** is an array where $\boldsymbol{B_j}$ is the number of elements $\boldsymbol{A_i}$ in bucket $\boldsymbol{j}$ or less. Or you can say $\boldsymbol{B_j = \sum_{k=0}^{j} B'_k}$ with $\boldsymbol{B'}$ is the old bucket array.
- Rearrange the input array **A** to all elements of each bucket **j** are stored in positions $\boldsymbol{A_i}$ where $\boldsymbol{B_{j-1} \le i \le B_j}$. This is the pseudo code to implement this idea:

```
swap(first_element, max)
While (number_of_moves < N):
  // find the element which is not in the correct range
  While (current_element is in the correct range):
    current_element := next_element
  While (current_element is not in the correct range):
    bucket_index = (m - 1)*(current_element - min)/(max - min)
    swap_index := prefix_sum[bucket_index]
    swap(current_element, swap_element)
    prefix_sum[bucket_index] -= 1
    number_of_moves += 1
```

- Sort each bucket using any sorting algorithm you want or you can call the flash sort function recursively.

- **Notes:**
  - For our implementation, we chose $\boldsymbol{M = 0.1N}$ and because of this, the range of each bucket is exactly 10. We also chose insertion sort to sort each bucket because when the array needed to be sort are small, insertion sort runs quickest and the implementation is short.

14) Practical Experiment:

14.1) Programming:

Here is how we split files and handle the programming part for experiment:

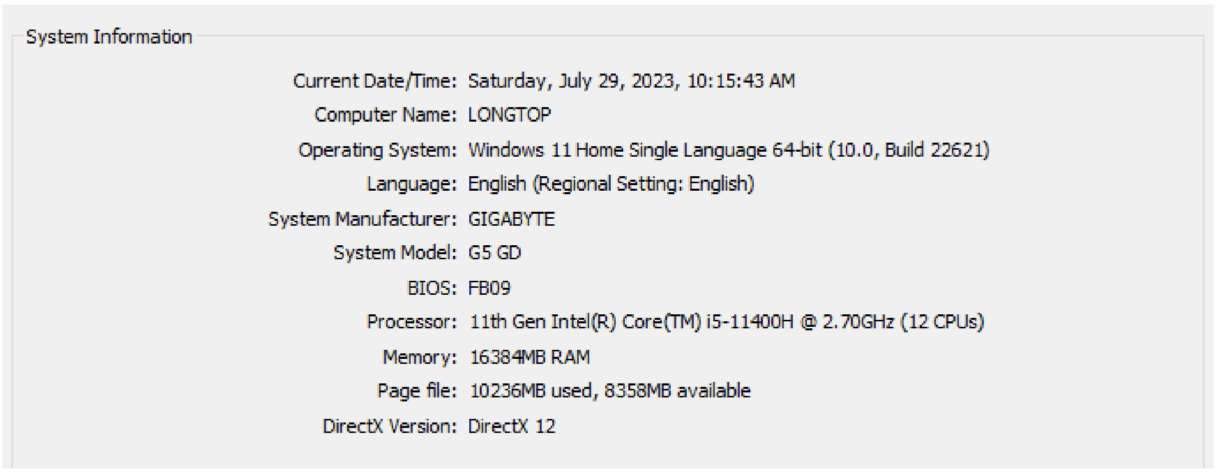
- <**DataGenerator**.**cpp**>: This is the provided **.cpp** file including all data generating functions that create array in **randomly generated order, nearly sorted order, sorted order and reverse order.**
- <**sort**.**cpp**>: We prepare our sorting algorithms in this file, there are two versions of each algorithm: one for time calculation and one for comparisons calculation.
- <**main**.**cpp**>: The main file is where we handle CLI commands and call the crucial functions from the two former **.cpp** files to compile a complete program.
- <**input_x.txt**>: The input file for our testruns and experiments. For each size there will be four files as there are four orders to test:
  - 1. Randomly Generated.
  - 2. Nearly Sorted.
  - 3. Sorted.
  - 4. Reverse.

14.2) Records:

All of the stastics shown below are the average value of five attemps for two categories - runtime and comparisons - in consideration of these following **sizes of data: 10000, 30000, 50000, 100000, 300000, 500000** and four different kinds of data entries for each size, respectively **randomly generated, nearly sorted, sorted and reverse.**

The tables named subsequently **Randomly Generated, Nearly Sorted, Sorted and Reverse** show the records of our practical experiments run on a laptop.

Specs of Laptop used for experiment:



System Information

Current Date/Time:	Saturday, July 29, 2023, 10:15:43 AM
Computer Name:	LONGTOP
Operating System:	Windows 11 Home Single Language 64-bit (10.0, Build 22621)
Language:	English (Regional Setting: English)
System Manufacturer:	GIGABYTE
System Model:	G5 GD
BIOS:	FB09
Processor:	11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz (12 CPUs)
Memory:	16384MB RAM
Page file:	10236MB used, 8358MB available
DirectX Version:	DirectX 12

Figure 1: Specs of Laptop used for running code

Support Application

- CMD: For running programs.
- Excel: For aggregating outputs and graph. drawing

Execution:

- Step 1: Use Command 3 to get the input files:

[Execution file] -a [Algorithm] [Input size] [Output parameter(s)]

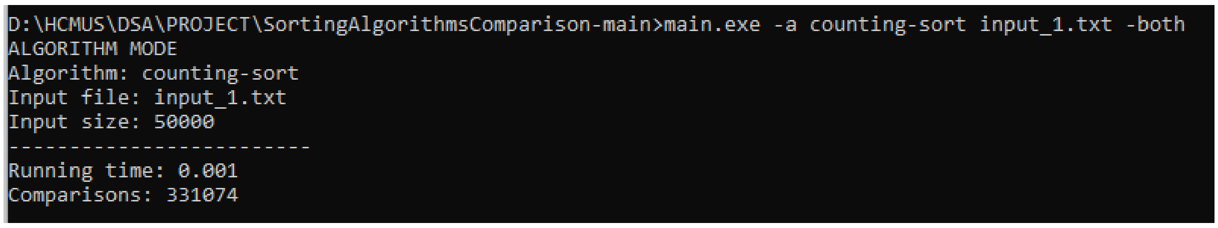
We call algorithm quick-sort to get the input files and output parameters is **-both**

Example: **main.exe -a quick-sort 500000 -both**

- Step 2: Use Command 1 for each algorithm to process the input files and get the outputs of comparisons and runtime

[Execution file] -a [Algorithm] [Given input] [Output parameter(s)]

Example: **main.exe -a counting-sort input_1.txt -both**:

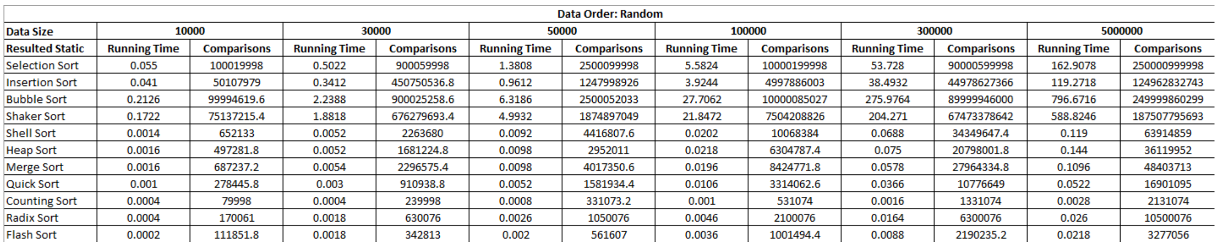


```
D:\HCMUS\DSA\PROJECT\SortingAlgorithmsComparison-main>main.exe -a counting-sort input_1.txt -both
ALGORITHM MODE
Algorithm: counting-sort
Input file: input_1.txt
Input size: 50000
-------------------------
Running time: 0.001
Comparisons: 331074
```

- Step 3: Put the outputs in Excel sheets.

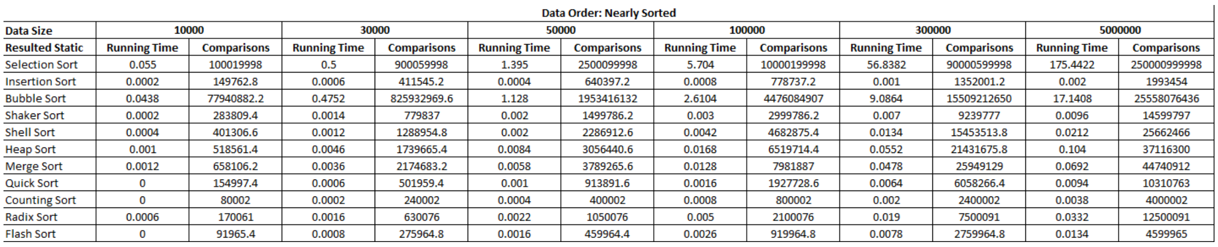
We repeat the process 5 times for each size of data and get the average values by AVERAGE().

Practical output:



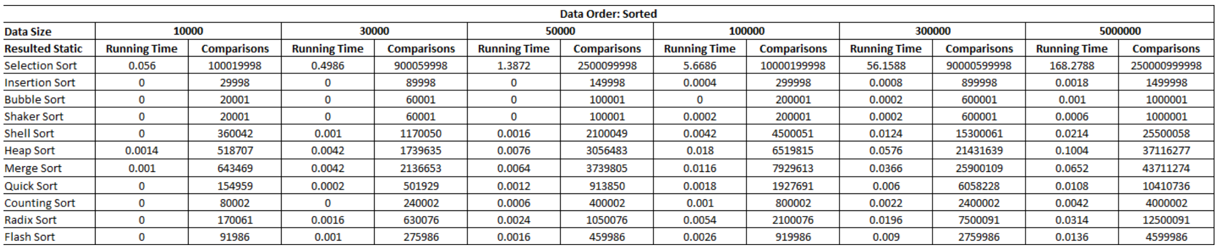
Data Order: Random												
Data Size	**10000**		**30000**		**50000**		**100000**		**300000**		**5000000**	
Resulted Static	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**
Selection Sort	0.055	100019998	0.5022	900059998	1.3808	2500099998	5.5824	10000199998	53.728	90000599998	162.9078	250000999998
Insertion Sort	0.041	50107979	0.3412	450750536.8	0.9612	1247998926	3.9244	4997886003	38.4932	44978627366	119.2718	124962832743
Bubble Sort	0.2126	99994619.6	2.2388	900025258.6	6.3186	2500052033	27.7062	10000085027	275.9764	89999946000	796.6716	249999860299
Shaker Sort	0.1722	75137215.4	1.8818	676279693.4	4.9932	1874897049	21.8472	7504208826	204.271	67473378642	588.8246	187507795693
Shell Sort	0.0014	652133	0.0052	2263680	0.0092	4416807.6	0.0202	10068384	0.0688	34349647.4	0.119	63914859
Heap Sort	0.0016	497281.8	0.0052	1681224.8	0.0098	2952011	0.0218	6304787.4	0.075	20798001.8	0.144	36119952
Merge Sort	0.0016	687237.2	0.0054	2296575.4	0.0098	4017350.6	0.0196	8424771.8	0.0578	27964334.8	0.1096	48403713
Quick Sort	0.001	278445.8	0.003	910938.8	0.0052	1581934.4	0.0106	3314062.6	0.0366	10776649	0.0522	16901095
Counting Sort	0.0004	79998	0.0004	239998	0.0008	331073.2	0.001	531074	0.0016	1331074	0.0028	2131074
Radix Sort	0.0004	170061	0.0018	630076	0.0026	1050076	0.0046	2100076	0.0164	6300076	0.026	10500076
Flash Sort	0.0002	111851.8	0.0018	342813	0.002	561607	0.0036	1001494.4	0.0088	2190235.2	0.0218	3277056

Figure 3: Randomly Generated



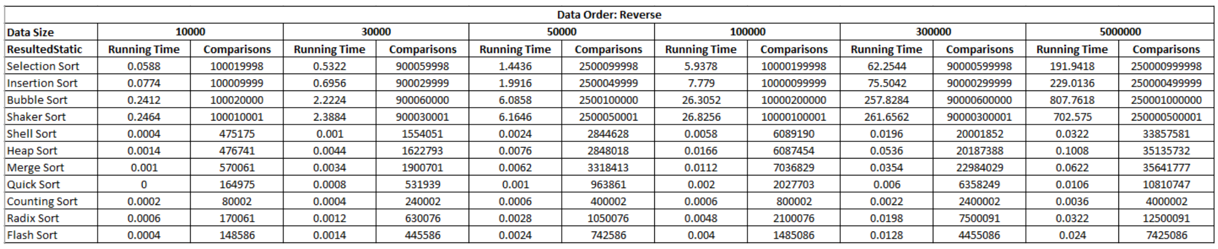
Data Order: Nearly Sorted												
Data Size	**10000**		**30000**		**50000**		**100000**		**300000**		**5000000**	
Resulted Static	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**
Selection Sort	0.055	100019998	0.5	900059998	1.395	2500099998	5.704	10000199998	56.8382	90000599998	175.4422	250000999998
Insertion Sort	0.0002	149762.8	0.0006	411545.2	0.0004	640397.2	0.0008	778737.2	0.001	1352001.2	0.002	1993454
Bubble Sort	0.0438	77940882.2	0.4752	825932969.6	1.128	1953416132	2.6104	4476084907	9.0864	15509212650	17.1408	25558076436
Shaker Sort	0.0002	283809.4	0.0014	779837	0.002	1499786.2	0.003	2999786.2	0.007	9239777	0.0096	14599797
Shell Sort	0.0004	401306.6	0.0012	1288954.8	0.002	2286912.6	0.0042	4682875.4	0.0134	15453513.8	0.0212	25662466
Heap Sort	0.001	518561.4	0.0046	1739665.4	0.0084	3056440.6	0.0168	6519714.4	0.0552	21431675.8	0.104	37116300
Merge Sort	0.0012	658106.2	0.0036	2174683.2	0.0058	3789265.6	0.0128	7981887	0.0478	25949129	0.0692	44740912
Quick Sort	0	154997.4	0.0006	501959.4	0.001	913891.6	0.0016	1927728.6	0.0064	6058266.4	0.0094	10310763
Counting Sort	0	80002	0.0002	240002	0.0004	400002	0.0008	800002	0.002	2400002	0.0038	4000002
Radix Sort	0.0006	170061	0.0016	630076	0.0022	1050076	0.005	2100076	0.019	7500091	0.0332	12500091
Flash Sort	0	91965.4	0.0008	275964.8	0.0016	459964.4	0.0026	919964.8	0.0078	2759964.8	0.0134	4599965

Figure 4: Nearly Sorted



Data Order: Sorted												
Data Size	**10000**		**30000**		**50000**		**100000**		**300000**		**5000000**	
Resulted Static	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**
Selection Sort	0.056	100019998	0.4986	900059998	1.3872	2500099998	5.6686	10000199998	56.1588	90000599998	168.2788	250000999998
Insertion Sort	0	29998	0	89998	0	149998	0.0004	299998	0.0008	899998	0.0018	1499998
Bubble Sort	0	20001	0	60001	0	100001	0	200001	0.0002	600001	0.001	1000001
Shaker Sort	0	20001	0	60001	0	100001	0.0002	200001	0.0002	600001	0.0006	1000001
Shell Sort	0	360042	0.001	1170050	0.0016	2100049	0.0042	4500051	0.0124	15300061	0.0214	25500058
Heap Sort	0.0014	518707	0.0042	1739635	0.0076	3056483	0.018	6519815	0.0576	21431639	0.1004	37116277
Merge Sort	0.001	643469	0.0042	2136653	0.0064	3739805	0.0116	7929613	0.0366	25900109	0.0652	43711274
Quick Sort	0	154959	0.0002	501929	0.0012	913850	0.0018	1927691	0.006	6058228	0.0108	10410736
Counting Sort	0	80002	0	240002	0.0006	400002	0.001	800002	0.0022	2400002	0.0042	4000002
Radix Sort	0	170061	0.0016	630076	0.0024	1050076	0.0054	2100076	0.0196	7500091	0.0314	12500091
Flash Sort	0	91986	0.001	275986	0.0016	459986	0.0026	919986	0.009	2759986	0.0136	4599986

Figure 5: Sorted



Data Order: Reverse												
Data Size	**10000**		**30000**		**50000**		**100000**		**300000**		**5000000**	
ResultedStatic	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**	**Running Time**	**Comparisons**
Selection Sort	0.0588	100019998	0.5322	900059998	1.4436	2500099998	5.9378	10000199998	62.2544	90000599998	191.9418	250000999998
Insertion Sort	0.0774	100009999	0.6956	900029999	1.9916	2500049999	7.779	10000099999	75.5042	90000299999	229.0136	250000499999
Bubble Sort	0.2412	100020000	2.2224	900060000	6.0858	2500100000	26.3052	10000200000	257.8284	90000600000	807.7618	250001000000
Shaker Sort	0.2464	100010001	2.3884	900030001	6.1646	2500050001	26.8256	10000100001	261.6562	90000300001	702.575	250000500001
Shell Sort	0.0004	475175	0.001	1554051	0.0024	2844628	0.0058	6089190	0.0196	20001852	0.0322	33857581
Heap Sort	0.0014	476741	0.0044	1622793	0.0076	2848018	0.0166	6087454	0.0536	20187388	0.1008	35135732
Merge Sort	0.001	570061	0.0034	1900701	0.0062	3318413	0.0112	7036829	0.0354	22984029	0.0622	35641777
Quick Sort	0	164975	0.0008	531939	0.001	963861	0.002	2027703	0.006	6358249	0.0106	10810747
Counting Sort	0.0002	80002	0.0004	240002	0.0006	400002	0.0006	800002	0.0022	2400002	0.0036	4000002
Radix Sort	0.0006	170061	0.0012	630076	0.0028	1050076	0.0048	2100076	0.0198	7500091	0.0322	12500091
Flash Sort	0.0004	148586	0.0014	445586	0.0024	742586	0.004	1485086	0.0128	4455086	0.024	7425086

Figure 6: Reverse

15) Statistical Visualization and Analysis:

15.1) Run-time:

In this part, we will demonstrate the efficiency of those sorting algorithms that we have presented theoretically in forms of line graphs considering the runtime.

In each case, we represent experimented outputs in two graphs for more convenient analysis. We consider **Improved Bubble Sort, Selection Sort, Improved Shaker Sort and Insertion Sort** to be **$\mathbf{O(N^2)}$ sorting algorithms.**

The reason for this arrangement will be explain in detailed in the first case - Randomly Generated Array.

15.1.1) Randomly Generated Array:

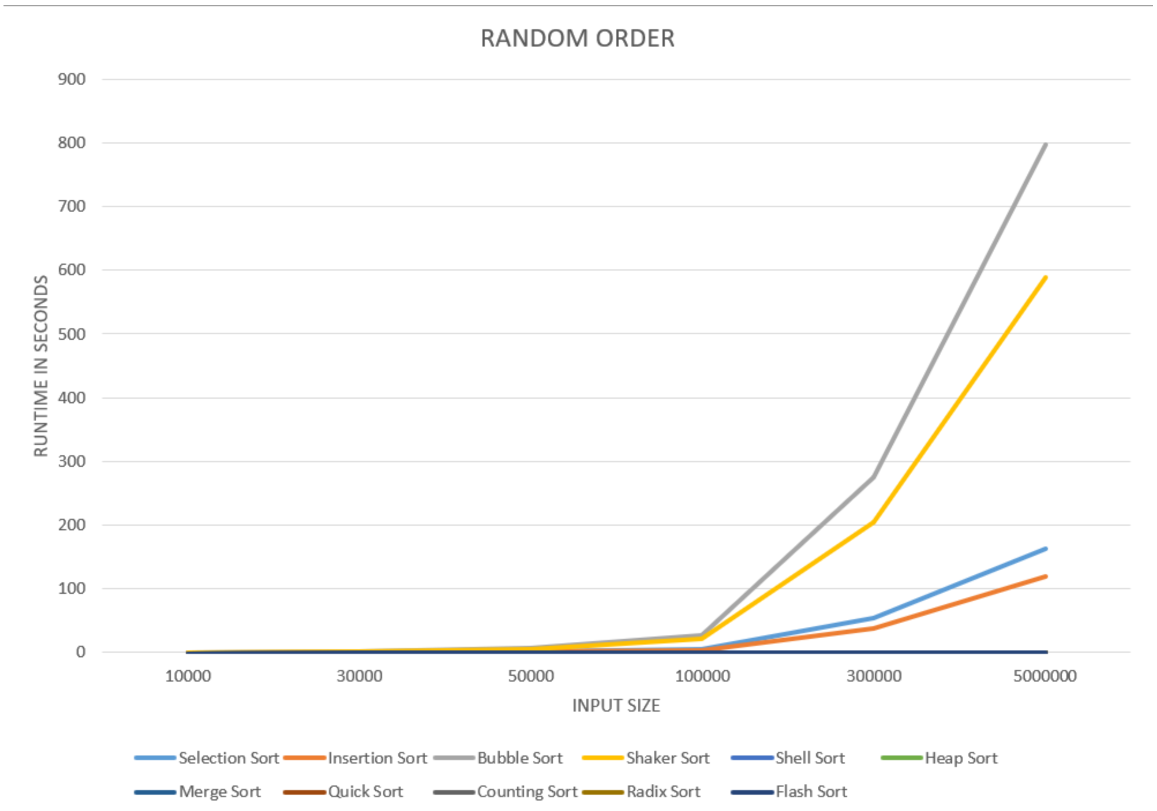


Figure 7: Randomly Generated Data (With O(N^2))

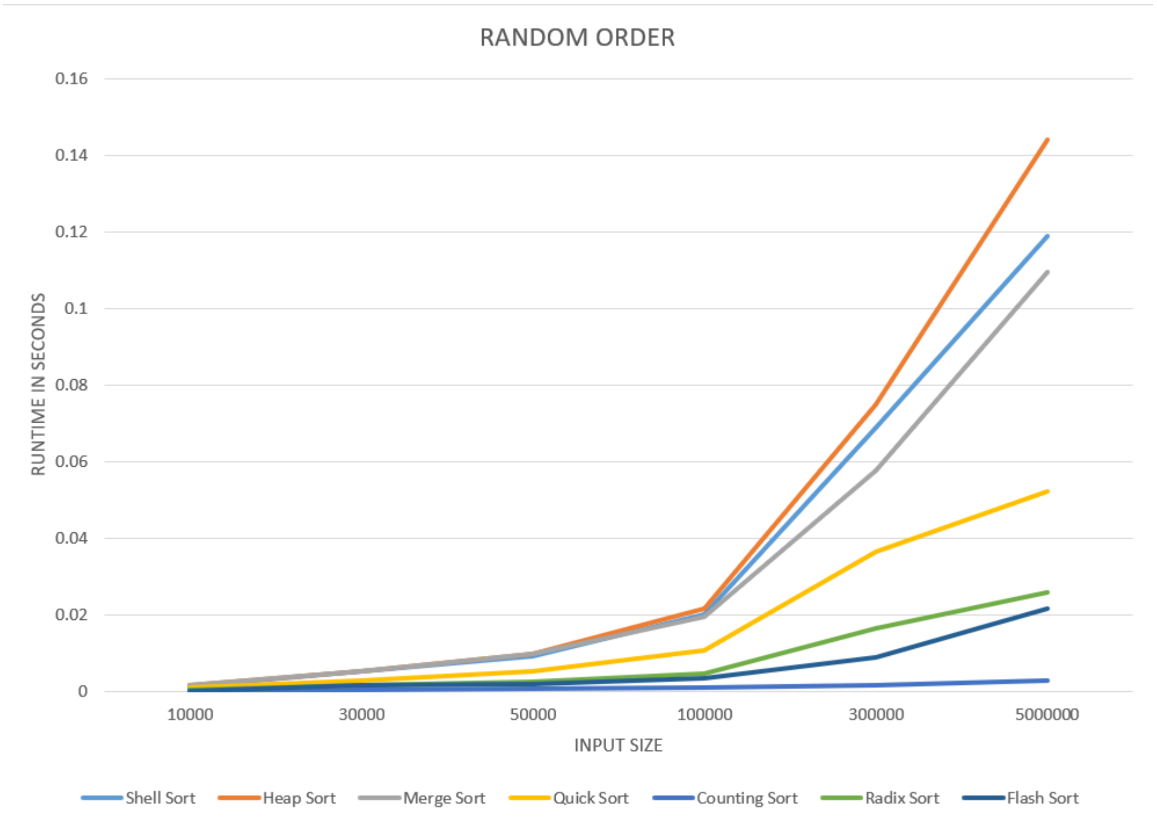


Figure 8: Randomly Generated Data (Without O(N^2))

First of all, the reason why we have to create two graphs for the same case is how inefficiently all **O(N^2)** sorting algorithms perform when the size of the array rockets so it ís difficult to see and compare ***O*(*N*)** and ***O*(*N* log *N*)** sorting algorithms.

Considering the first graph, when the size of the array is not greater than 100000 entries, all the **O(N^2)** sorting algorithms give fairly feasible results and the longest runtime, which is registered by Bubble Sort, does not exceed 40 seconds. But with triple the size of this input, 300000 randomly generated elements really create a hard-won challenge for this group of algorithms, leading to them having to spend nearly 300 seconds to sort the array.

The final size is really a nemesis for **Improved Bubble Sort** as it needs roughly 800 seconds to put the array in order and its variant - **Improved Shaker Sort** also needs approximately 600 seconds to finish the tasks. On the other hand, Insertion Sort and Selection Sort do spend remarkable amount of time to arrange arrays of more than 100000 elements, but they never reach 200 seconds.

In the chart of no **O(N^2)** sorting algorithms, the vertical axis is scaled on a smaller range of under 1 seconds unlike the first graph, whose highest reachable time can be 900 seconds. This feature is the primary reason why we have to seperate the demonstration into two graphs, it is almost impossible to dissect the perfomance of other more efficient sorting algorithms without this division. **Counting Sort** gives the phenomenal results as it has the fastest output in all cases, no matter how large the size of the arrays, and it is the only algorithm to spend under 0.02 seconds on the biggest size. In contrast, the worst performance in the second graph belongs to **Heap Sort**. In all tested sizes, **Heap Sort** is always the one to spend the longest time to sort a randomly generated array. In the case of 500000 elements, **Heap Sort** surpasses all other candidates to be the only sorting algorithms in this graph to run more than 0.12 seconds (actually even 0.14 seconds).

15.1.2) Nearly Sorted Array:

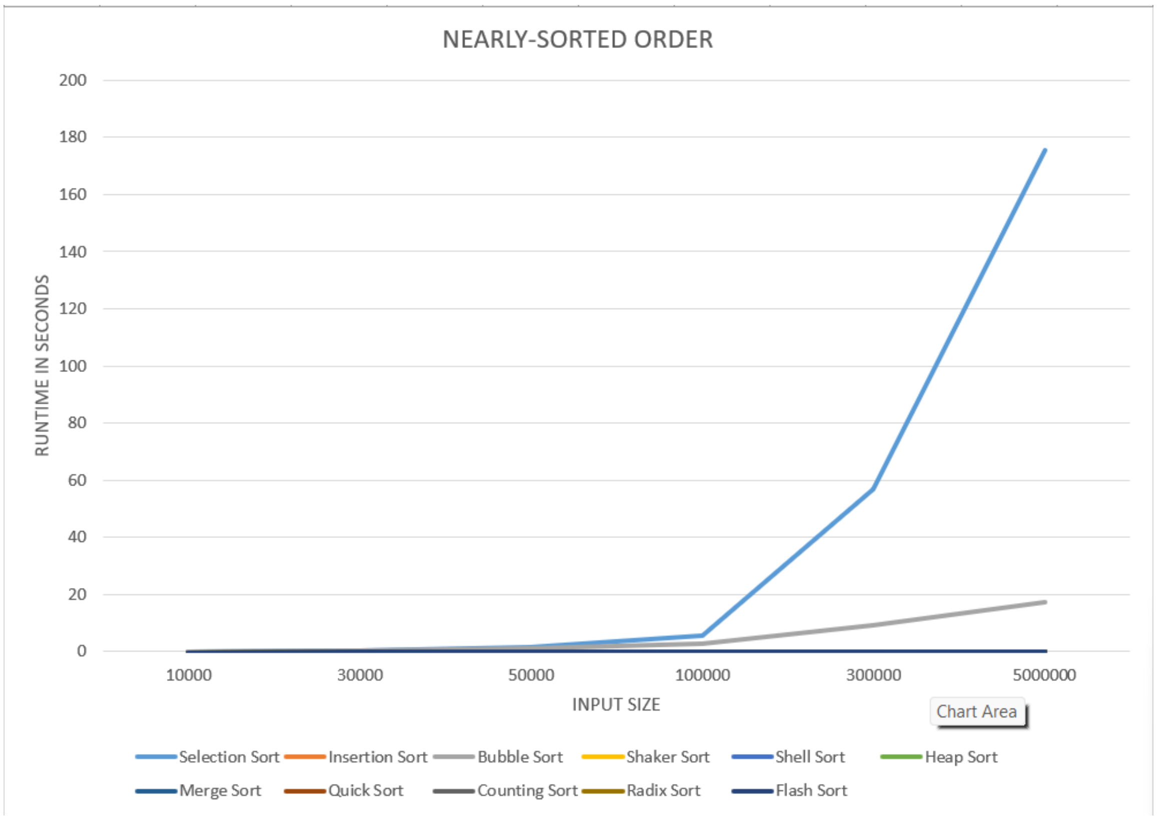


Figure 9: Nearly Sorted Array (With O(N^2))

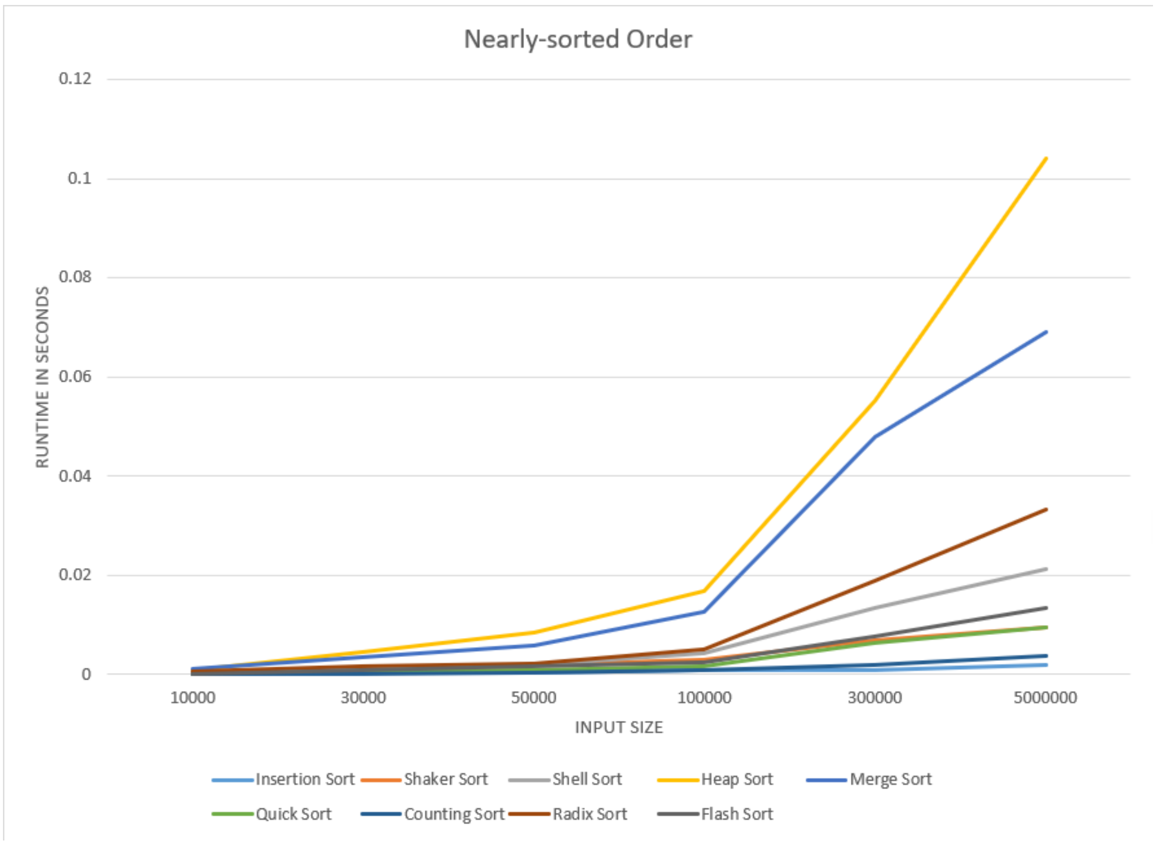


Figure 10: Nearly Sorted Array (Without Seclection Sort and Bubble Sort)

When the array is nearly sorted, the result can be quite strange at the first look. Selection Sort this time is the lowest sorting algorithm off all because it always has to perform at $\mathbf{O(N^2)}$ with no favorable breakpoint like **Improved Bubble Sort** and **Improved Shaker Sort**.Although **Improved Bubble Sort** runs obviously faster in this case, it still almost reaches 20 seconds while sorting a nearly sorted array of 500000 elements.

Regarding the second graph, other algorithms are not linear and they have no breakpoint like those **$\mathbf{O(N^2)}$ algorithms**, this can explain why many of those **O(NlogN)** and **O(N) algorithms** are a little bit slower than them. The positions of other lines in the **Without $\mathbf{O(N^2)}$** graph remain mostly unchanged, except there is a switch between **Flash Sort** and **Quick Sort**. The reason mostly lies in how the elements are ordered in a nearly sorted sequence, this faciltates the process around the pivot of **Quick Sort** which leads to less time taken to sort the array.

15.1.3) Sorted Array:

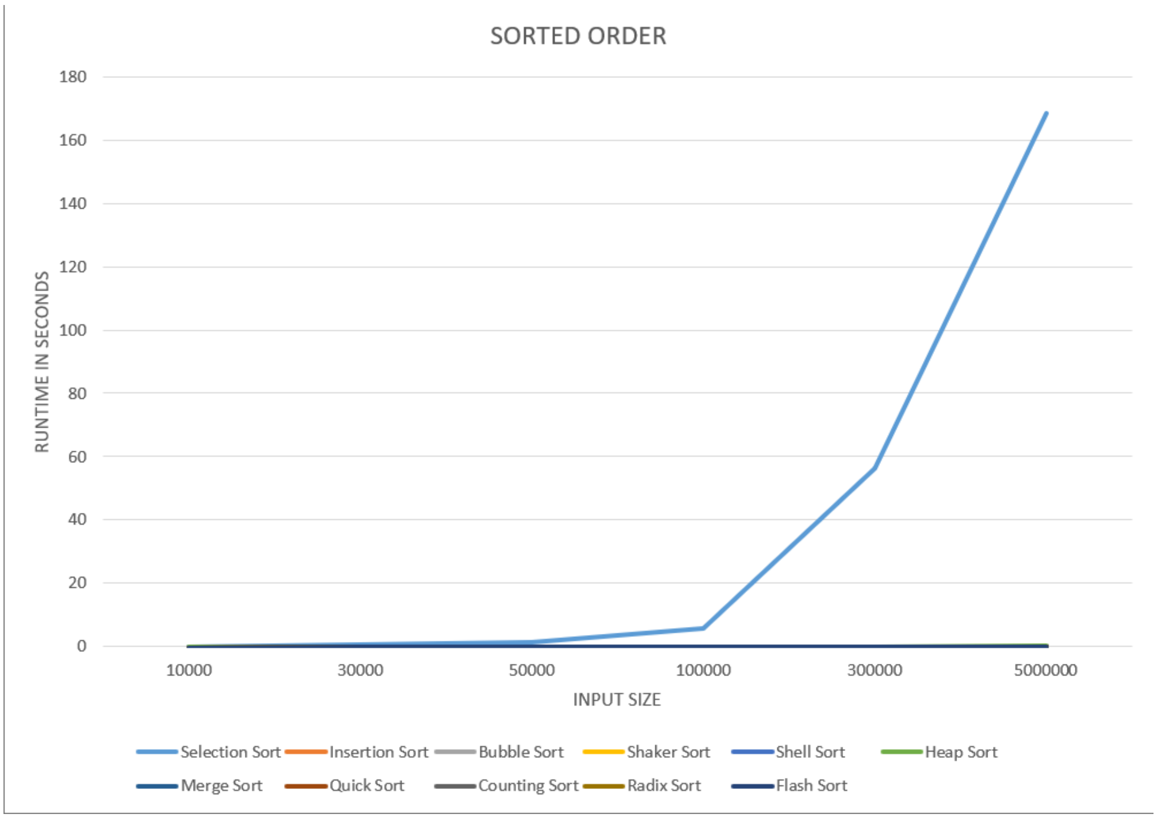


Figure 11: Sorted Array (With O(N^2))

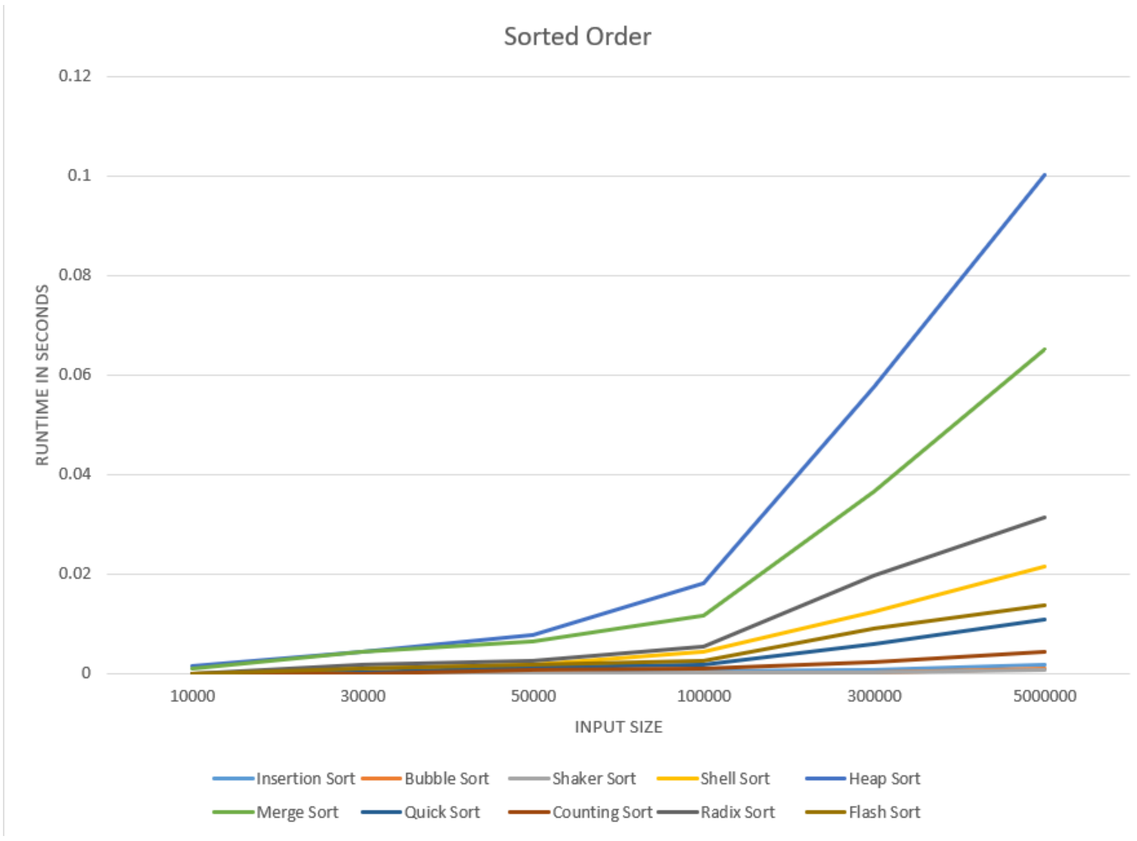


Figure 12: Sorted Array (Without Selection Sort)

In this usually beneficial kind of array, it is predictable to only see a single line of Selection Sort in the **With O(N^2)** graph. The reason is the same as mentioned in **Nearly Sorted Array**, it always has to perform at **O(N^2)** with no favorable breakpoint like **Improved Bubble Sort** and **Improved Shaker Sort**. Other algorithms in **O(N^2)** group are extremely facilitated by this order of input data, **Improved Bubble Sort** and **Improve Shaker Sort** always stop in the first loop when the first traversal notice that the array is already sorted, while the Insertion Sort just process and compare every element linearly without making any swap.

In the next graph where Selection Sort is not considered, the reason why some **O(NlogN) and O(N) algorithms** are a little bit slower than some other **O(N^2)** algorithms in processing **Sorted Array** is similar with the explaination in the section **Nearly Sorted Array**. Heap Sort and Merge Sort always maintain an **O(NlogN)** runtime no matter what size or what kind of the data entries, and they are likely to be the two slowest algorithms outside the **O(N^2)** group. Just like the case of **Nearly Sorted Array**, **Quick Sort** has better runtime than **Flash Sort**, and **Counting Sort** stays at the place of the fastest algorithm of all.

15.1.4) Reverse Array:

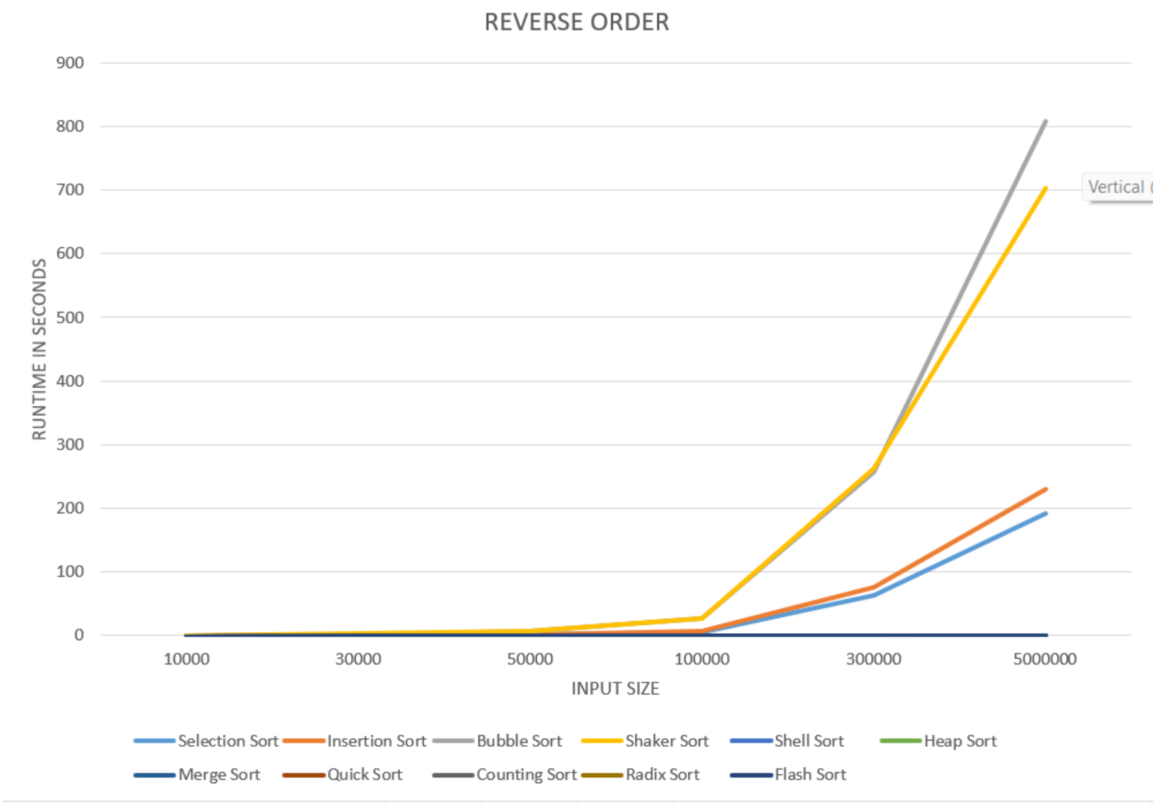


Figure 13: Reverse Array (With O(N^2))

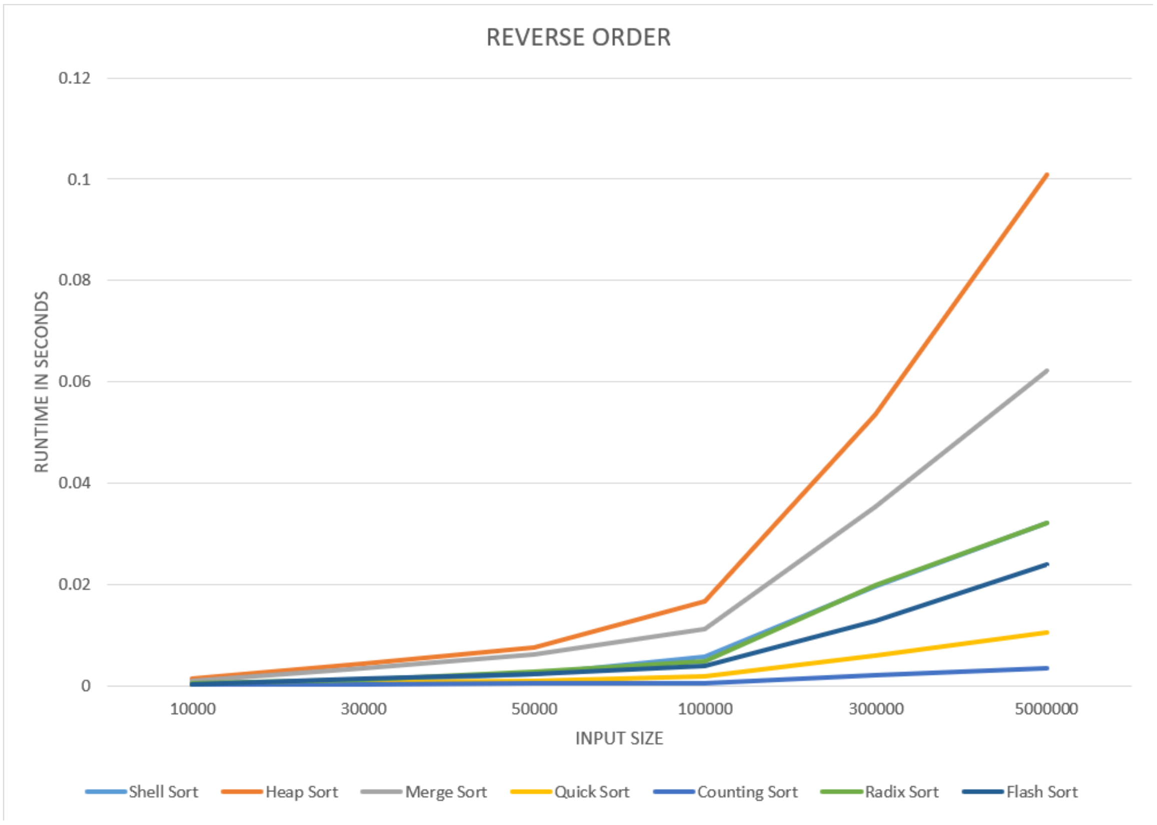


Figure 14: Reverse Array (Without O(N^2))

This order of unsorted array is mostly the worst case of many algorithms discussed in this report, especially the $\mathbf{O(N^2)}$ group.

As can be seen from both graphs, they share plenty of traits with two graphs of in the case **Randomly Generated Array** in term of the movements of the lines.

In **Reverse Array (With $\mathbf{O(N^2)}$)**, Bubble Sort again leads as the slowest sorting algorithms in all sizes of array, and it even spends more than 800 seconds on sorting the array in the last situation, which is higher than the longest processing time in **Randomly Generated Array** section of approximately 800 seconds. The other condidates result in no change in ranking, following Bubble Sort as the second slowest algorithm is its variant - Shaker Sort, and Selection Sort still triumphs as the fastest of all $\mathbf{O(N^2)}$ sorting algorithms despite having to spend nearly 200 seconds on the last size.

On a smaller scale where we do not consider $\mathbf{O(N^2)}$ algorithms, the slowest algorithm and the fastest algorithm remain unchanged as **Heap Sort** and **Flash Sort** respectively and their overall results in all cases are more faster than those of them in the **Randomly Generated Array** The only shift in position in this graph compared to others above is how **Shell Sort** and **Radix Sort** share roughly the same outputs for all sizes as their lines on the graph obviously overlay each other.

15.2) Comparisons:

In this part, we visualization how many comparisons each sorting algorithms perform for each kind of data of all sizes in form of bar charts.

The highest recored comparisons can exceed billions; therefore, for more convenient visualization, we have decided to scale all the numeric values down to $\log_{10}$ of orginal total comparisons, determined as:

- **visualized_value = $\mathbf{\log_{10}}$(original_value)**

Although we only generate a graph for each kind of dataset, **Selection Sort, Insertion Sort, Improved Bubble Sort and Improved Shaker Sort** are still be classified into $\mathbf{O(N^2)}$ sorting algorithms for easier explainations.

15.2.1) Randomly Generated Array:

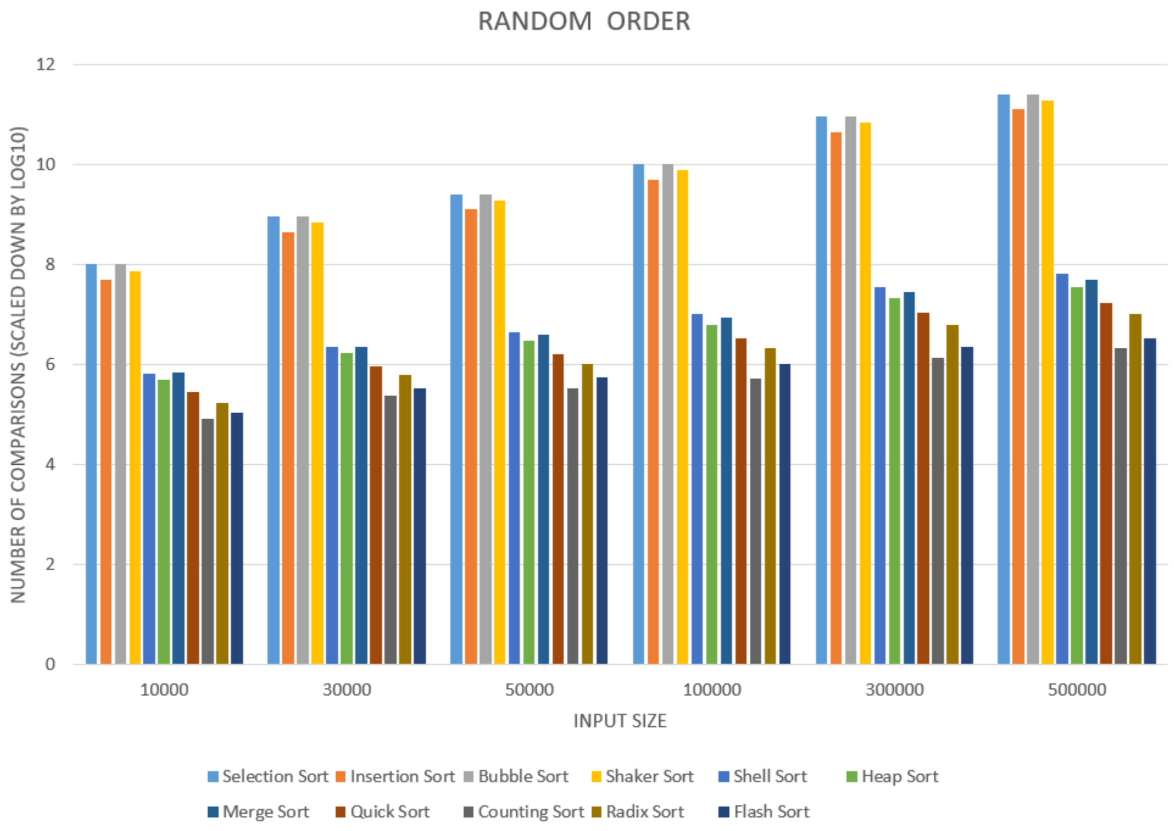


Figure 15: Randomly Generated Array

In general, the rankings of all sorting algorithms are unchangeable no matter how great the size of the input array is. The only obvious shift in the graph is the height of each vertical bar in the graph as the size of the array increases.

As expected from the essences of each sorting algorithms, the $\mathbf{O(N^2)}$ algorithms always lead with largest quantities of comparisons. Bubble Sort and Selection Sort register as the richest of all when they always have the highest amount of comparing executions at N^2 or more for every size.

For other sorting algorithms, **Counting Sort** and **Flash Sort** respectively result in the fewest and second fewest quantity of comparisons. The reason for Counting Sort's outcome lies in its nature as a non-comparison sorting algorithms, leading to its expectable small amount of processed comparisons that mostly come from the loops to process the array. Although **Flash Sort** still bases on comparing elements, it performs best on a uniformly distributed data entries, which is the characteristic of the array resulted from the provided data generator. Other sorting algorithms execute as expected when their quantities revolve around **Nlog(N)** comparisons for every size.

15.2.2) Nearly Sorted Array:

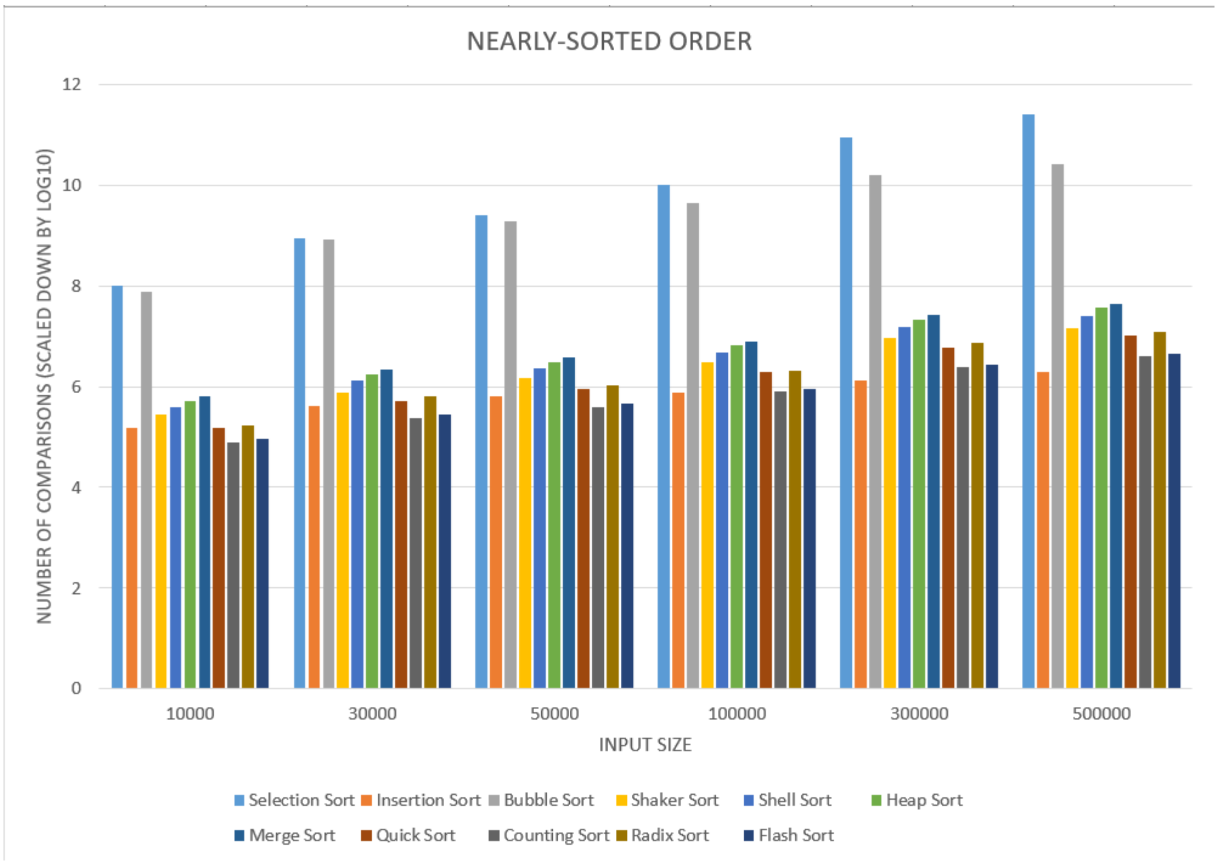


Figure 16: Nearly Sorted Array

In this order of input array, most of the algorithms do not have any noteable diffences besides relatively fewer comparisons spent. The only remarkable change conpared with the previous graph is how the quantities of comparison of **Insertion Sort** and **Improved Shaker Sort** plummet significantly when the array is already nearly sorted.

The version of **Shaker Sort** that we use obviously plays the primary role for this change. In the operation of **Improved Shaker Sort**, the sorting process stops when the algorithm detects that there is no swap made in a traversal of any directions, thus when the array is nearly ordered, there will be a point when the algorithm finishes sorting the array after a traversal of one direction, leading to the next traversal of the opposite direction recognize that the desired sequence is achieved ad stop the process immidiately. Another reason can come from the essence of **Shaker Sort** itself as it sorts the array in both directions, so it can reach that stop point sooner. Unlike **Improved Shaker Sort**, Insertion do not detects whether there is a swap, it just simply spend fewer comparisons on a nearly sorted array, which is its favorable input.

15.2.3) Sorted Array:

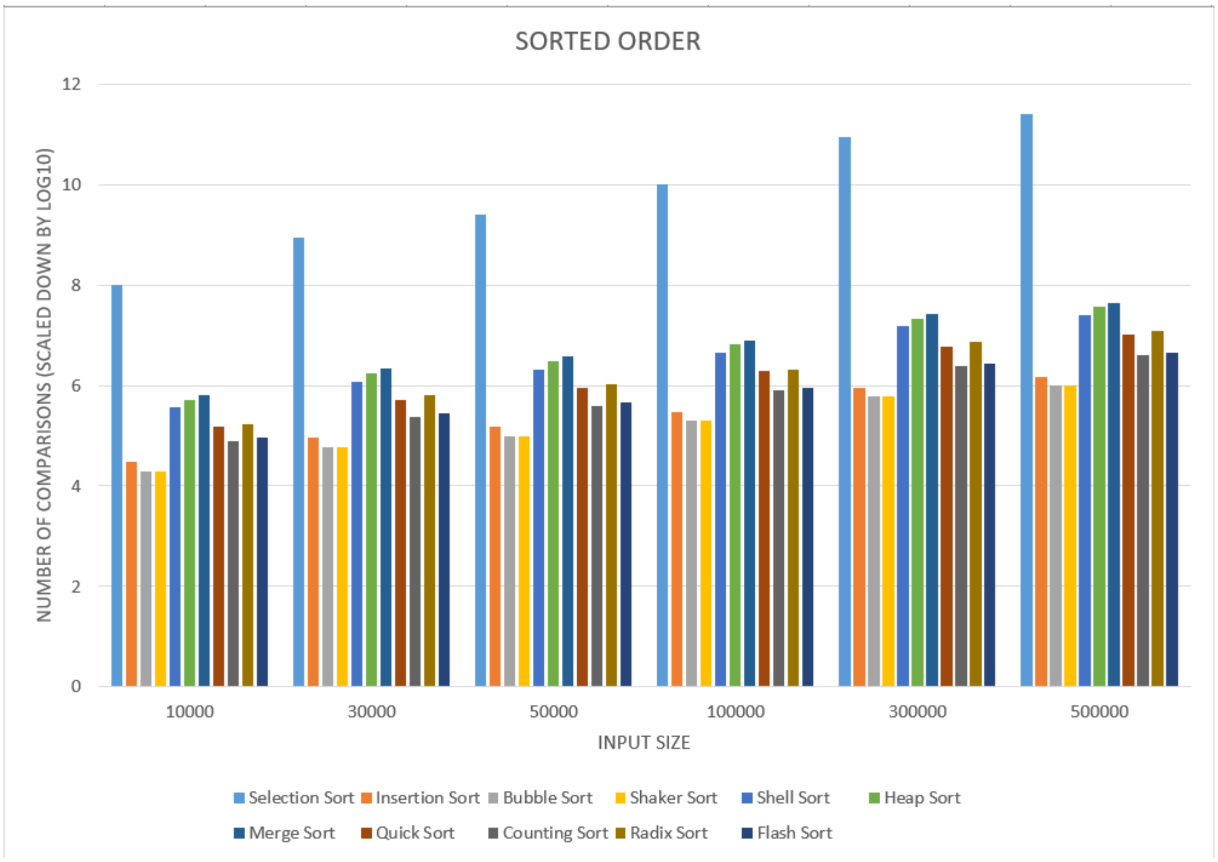


Figure 17: Sorted Array

Sorted Array is usually the favorable condition for many sorting algorithms.

In the $\mathbf{O(N^2)}$ group, **Selection Sort** again is the richest in comparisons of all algorithms as it always spends N^2 comparisons no matter what order of the array. As expected, **Improved Bubble Sort** and **Improved Shaker Sort** thrive as having the smallest amoung of comparison being made. There is only one explaination for the trait of the graph is that they are destined to stop in the first outer loop if the array is already sorted, leading to their predicted small numbers of comparison. **Insertion Sort** meets the similar case when it just need a loop to traverse through an array in desired order and just one comparison is performed at max for each processed element.

Other algorithms do not process the array linearly and they usually maintain the same big O notation in most cases, this can explain why although the array is already sorted but their quantity of comparisons show no evident changes.

15.2.4) Reverse Array:

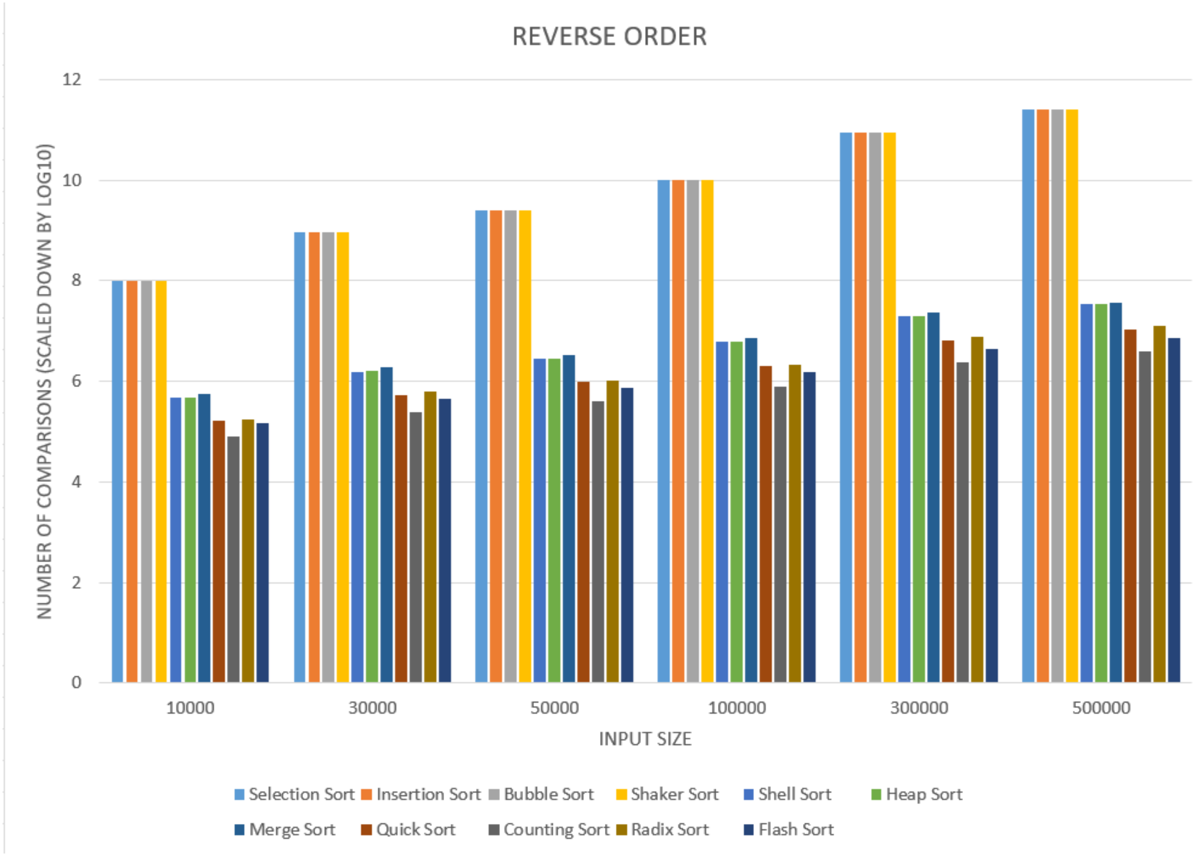


Figure 18: Reverse Array

Reverse order - the worst case for all $\mathbf{O}(\boldsymbol{N}^2)$ sorting algorithms really show how struggling the input can be. In this kind of data only do all member of the $\mathbf{O}(\boldsymbol{N}^2)$ share roughly the same results for all sizes of data, they also own the highest numbers of comparisons for each size of data at always $\boldsymbol{N}^2$.

Other algorithms do not receive much modification afterall. The only prevalent difference that they all share is the increased height of the bar when the size of the array grows. If there is one noticeable feature to point out, it appears that **Shell Sort** and **Heap Sort** have somewhat identical performances as their bars are at always at the same heights in the graph.

16) References

- Course resources: Google Drive for course activity
- Website:
  - Sorting Algorithms
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Shaker Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort
  - Counting Sort
  - Radix Sort
  - Shell Sort
  - Flash Sort