UNIVERSITY OF SCIENCE

VIETNAM NATIONAL UNIVERSITY - HCM CITY

LAB 3 - SORTING

GROUP 1 REPORT

Theory Lecturer	Nguyen Thanh Phuong	
Instructors	Bui Huy Thong	
	Nguyen Ngoc Thao	
Students	Vo Thanh Tu	21127469
	Nguyen Khanh Nhan	21127657
	Pham Thi Phuong Nam	21127364
	Nguyen Huynh Phu Qui	21127153

Data Structures and Algorithms
Third Term 2021-2022

Contents

1 Introduction and information

1.1 Members Information

- 21127469 - Vo Thanh Tu
- 21127657 - Nguyen Khanh Nhan
- 21127364 - Pham Thi Phuong Nam
- 21127153 - Nguyen Huynh Phu Qui

1.2 CPU Information

We use Google Colaboration platform with specification:

- Model name: Intel(R) Xeon(R) CPU @ 2.20GHz
- CPU MHz: 2200.194
- Cache size: 56320 KB

2 Algorithm Presentation

2.1 Selection Sort

2.1.1 Algorithmic ideas

Find the smallest element in the unsorted array, store it at the beginning of the sorted array, then continue to find the smallest element from the remaining unsorted elements, then place it at the end of the sorted sequence. And so on, until all the elements are sorted.

2.1.2 Operate the algorithm

Step 1: In the first turn of the external loop (except at $i = 0$), we assign the location of the smallest value min_index by the first index, any element $a[j]$ from 1 to $n - 1$ with a value less than $a[min\_index]$ is interchangeable. As a result, the item with a value less than the value $a[min\_index]$ is saved to $a[0]$.

Step 2: In the second turn with $i = 1$, the inner-loop starts from $a[2]$ Any element from 2 to $n - 1$ has value less than $a[1]$ (the $a[min\_index]$) is swapped for $a[min\_index]$. After this turn, the second smallest value element is saved to $a[1]$.

Step 3: The above process goes on continuously until it reaches $n-2$. At this time, the inner loop only repeats once, and the exchange occurs if $a[n - 1] < a[min\_index]$.

2.1.3 Time and space complexity

- Worst-case and average-case time complexity $O(N^2)$: The worst-case occurs when the array is already sorted (with one swap), but the smallest element is the last element
- Best-case time complexity $O(N^2)$: The best-case occurs when an array is already sorted.
- Auxiliary space: $O(1)$

2.2 Quick Sort

2.2.1 Algorithmic ideas

Quick Sort is an algorithm that applies the idea of divide-and-conquer algorithm. It first splits the input array into two sub-arrays, one half smaller and one half larger, based on an intermediate element. Then it recursively sorts the subarrays to solve the problem.
The key to the algorithm is sub-array division, also known as the segmentation algorithm. The solution can be summarized as follows:

- Choosing an element in the array as an intermediate element to split the array in half is called the pivot. Usually, we often choose the first element, the element in the middle of the array, or the last element of the array to do the pivot.
- Choosing an element in the array as an intermediate element to split the array in half is called the pivot. Usually, we often choose the first element, the element in the middle of the array, or the last element of the array to do the pivot.
- After the segmentation step, move the pivot to the correct middle position of the 2 subarrays
- Recursively apply the above segmentation steps to two sub-arrays to perform sorting

2.2.2 Operate the algorithm

Step 1: Initialize left and correct variables to save the top and end positions of the sequence to be sorted, identifying the median element in the sequence to divide the array into two halves.

Step 2: Use the i and j variables to divide the sequence into 2 parts that turn j to run from left to right, turning j to run from right to left. If $a[i] < mid-value$ and $a[j] > mid-value$ increases the variable value and reduces the variable value j by 1 until detected $a[i] > mid-value$ and $a[j] < mid-value$ we will swap positions i and j for each other.

Step 3: When $i >= j$, the arrangement array is divided into 2 parts. The range from 0 to j has a value smaller than the median element, and the sequence from i to $n-1$ is larger than the median.

Step 4: We continue to recursively sort the two parts of the array until the array is sorted in ascending order.

2.2.3 Time and space complexity

- The best-case $O(N \log N)$: occurs when the partition process always picks the middle element as the pivot.
- The worst-case $O(N^2)$: occurs when the partition process always picks the greatest or smallest element as the pivot. Let's consider the above partition strategy where the last element is consistently picked as the pivot. The worst case occurs when the array is already sorted in increasing or decreasing order.
- Auxiliary space: $O(1)$

2.3 Heap Sort

2.3.1 Algorithmic ideas

The Heap Sort algorithm takes the idea of solving from the heap structure, specifically:

- We consider the array to be sorted as a complete binary tree, then modify the tree to a heap structure.
- Based on the properties of the heap structure, we can get the array's largest or most minor part. This element is the root of the heap, reduces the number of elements of the binary tree, and refactors the heap.
- Return the top element of the heap to the correct position of the array at the end of the array, then reduce the number of elements of the array (no longer consider the last element)
- Refactor the heap and iteratively get the root of the heap structure until the original list has only 1 element left. Return this element to the correct position and end the algorithm.

2.3.2 Operate the algorithm

Step 1: The `HeapPartition` function checks the left-side and right-side node positions right at the i^{th} position to find the node with the largest value. If the element on the left node is larger than the element at the i^{th} position, then the position of the element with the largest value is equal to the element on the left-side node, the same we have with the right-side node.

Step 2: In case the node with the largest value is not the node at position i, the function swap the values of these 2 nodes and continue the `HeapPartition` recursively from the largest-value node to go to the following branches to ensure The nodes always have a greater value than their child nodes.

Step 3: The above process goes on continuously until it reaches $n-2$. At this time, the inner loop only repeats once, and the exchange occurs if $a[n-1] < a[min\_index]$. Finally, the HeapSort function receives the input as an array, and the first loop performs heap creation from the array's data and size. The loop from the middle of the array iterate and arrange the elements to create a heap that satisfies the required properties using the `HeapPartition` function.

Step 4: The second loop iterate from the end of the array $n - 1$ to swap the element values. After each iteration, the function rearranges the heap and reduces the heap's size to 1.

2.3.3 Time and space complexity

- Best-case, worst-case and average-case time complexity $O(N \log N)$: in all cases the algorithm executes with the same time value.
- Auxiliary space: $O(1)$

2.4 Counting Sort

2.4.1 Algorithmic ideas

Use a counting array to count the occurrences of the elements. This array is then used to place the elements in their correct positions.

2.4.2 Operate the algorithm

Step 1: Find the max and min elements of the array of length N.

Step 2: Initialize a count array of length $(max - min + 1)$ with all zero-value.

Step 3: Run a loop from 0 to $N - 1$. Each time element i appears, add 1 to the elements at position $(i - min)^{th}$ of the counting array.

Step 4: Store cumulative sum of the elements of the count array. It helps in-placing the elements into the correct index of the sorted array.

Step 5: Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated. After placing each element at its correct position, decrease its count by one.

Step 6: Return the array in ascending order.

2.4.3 Time and space complexity

- Running time is based on the range of elements in the array, so the overall time complexity becomes $O(max - min + 1)$
- Auxiliary space: $O(max - min + 1)$

2.5 Merge Sort

2.5.1 Algorithmic ideas

Using the Divide and Conquer technique, Merge Sort recursively divide the array into sub-arrays. When the sub-arrays size is 1, then merges those sub-arrays to produce a sorted array.

2.5.2 Operate the algorithm

Step 1: In the MergeSort function, use the Split function for the whole array with N elements in 2 halves. Then recursively call MergeSort to split the halves.

Step 2: After splitting the array into only one element, start merging the elements again based on a comparison of the size of the elements. Compare the element for each array and then combine them into another array.

Step 3: The array is in ascending order after the final merging.

2.5.3 Time and space complexity

- Best-case, worst-case and average-case time complexity $O(N \log N)$: in all cases the algorithm executes with the same amount of time. Merge Sort performs the same number of operations for any input array of a given size. In this algorithm, we divide the array into two recursively subarrays, creating $(O \log N)$ rows where each element is present in each row exactly once. Each row takes $O(N)$time to merge every pair of sub-arrays. So the overall time complexity becomes $O(N \log N)$.
- Auxiliary space: $O(N)$ because all elements are copied into a temporary array.

2.5.4 Improvements and variants

- **Improvements**
  Test whether the array is already in order. We can reduce the running time to be linear for arrays that are already in order by adding a test to skip the call to `merge()` if $a[mid]$ is less than or equal to $a[mid+1]$. With this change, we still do all the recursive calls, but the running time for any sorted subarray is linear.
  Eliminate the copy to the auxiliary array. It is possible to eliminate the

time (but not the space) taken to copy to the auxiliary array used for merging. To do so, we use two invocations of the sort method, one that takes its input from the given array and puts the sorted output in the auxiliary array; the other takes its input from the auxiliary array and puts the sorted output in the given array. With this approach, in a bit of mind-bending recursive trickery, we can arrange the recursive calls such that the computation switches the roles of the input array and the auxiliary array at each level.

- **Variants**
  - **Oscillating Merge Sort:** A variation of Merge Sort used with tape drives that can read backwards. Instead of doing a complete distribution as is done in a tape merge, the distribution of the input and the merging of runs are interspersed. The oscillating Merge Sort does not waste rewind time or have tape drives sit idle as in the conventional tape merge.
  - **Polyphase Merge Sort :**A variation of a bottom-up Merge Sort that sorts a list using an initial uneven distribution of sub-lists (runs), primarily used for external sorting, and is more efficient than an ordinary Merge Sort when there are fewer than eight external working files (such as a tape drive or a file on a hard drive). A polyphase Merge Sort is not a stable sort.
  - **3-way Merge Sort:** Merge Sort involves recursively splitting the array into 2 parts, sorting and finally merging them. A variant of Merge Sort is called 3-way Merge Sort where instead of splitting the array into 2 parts we split it into 3 parts.
    Merge Sort recursively breaks down the arrays to subarrays of size half. Similarly, 3-way Merge Sort breaks down the arrays to subarrays of size one third.

2.6 Radix Sort

2.6.1 Algorithmic ideas

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix Sort uses Counting Sort as a subroutine to sort.

2.6.2 Operate the algorithm

Step 1: Find the maximum value of the array to get the maximum number of meaning digits and store it into the variable $max - digit$

Step 2: Set a variable call $pos - digit$ equal to 0. Zero is the position of the least significant digit.

Step 3: Use Counting Sort to sort the array by digits at $pos - digit$ of each item.

Step 4: If $pos - digit$ is less than $max - digit$, return step 2. Otherwise, go to the next step.

Step 5: Step 5: Sorting the array by positive and negative.

2.6.3 Time and space complexity

Call n as the array's size, b as the base for representing numbers, and d as this array's maximum value.

- In most cases, the maximum number of digits is $\log_b d$, which is also the number of loops. Each loop has iterated to n items in the array. So, the complexity in most cases is $O(N \log_b d)$
- In specific cases, with limitation, d must be less or equal to n, and the algorithm use binary to resenting number, the complexity is $O(N \log N)$

2.6.4 Improvements and variants

- Because the idea of this algorithm is sort by digit, there are also many forms of this algorithm in the base 8, base 10, base 16, et cetera. In this lab, we use binary to resent the numbers.
- Radix Sort also has a version that start from most significant digit.
- Characters can also be treated as digits to sort strings.

- To sort a user-define type, the developer can use Radix Sort for each property step by step based on the priority comparison.
- Beside Counting Sort, Radix Sort can use another sort likes Insertion Sort, Quick Sort, et cetera, as a subroutine to sort.

2.7 Flash Sort

2.7.1 Algorithmic ideas

Flash Sort is an efficient in-place implementation of histogram sort, itself a type of bucket sort. It assigns each of the N input elements to one of m buckets, efficiently rearranges the input to place the buckets in the correct order, then sorts each bucket.

2.7.2 Operate the algorithm

Step 1: Using a first pass over the input, find the minimum and maximum sort keys.

Step 2: Linearly divide the range $[A_min, A_max]$ into m buckets.

Step 3: Make one pass over the input, counting the number of elements which fall into each bucket.

Step 4: Convert the counts of elements in each bucket to a prefix sum, where $L[b]$ is the number of elements in bucket b or less. ($L[0] = 0$ and $L[m] = n$)

Step 5: Rearrange the input to all elements of each bucket b are stored in positions $A[i]$ where $L[b-1] < i \leq L[b]$.

Step 6: Sort each bucket using Insertion Sort.

2.7.3 Time and space complexity

- As with all bucket sorts, performance depends critically on the balance of the buckets. In the ideal case of a balanced data set, each bucket will be approximately the same size. If the number m of buckets is linear in the input size N, each bucket has a constant size, so sorting a single bucket with an $O(N^2)$ algorithm like Insertion Sort has complexity $O(1^2)$ = $O(1)$. The running time of the final Insertion Sorts is therefore $m \times O(1) = O(m) = O(N)$.
- Choosing a value for m, the number of buckets, trades off time spent classifying elements (high m) and time spent in the final Insertion Sort step (low m). For example, if m is chosen proportional to $\sqrt{N}$, then the running time of the final Insertion Sorts is therefore $m \times O(\sqrt{N^2}) = O(N^{3/2})$.

- In the worst-case scenarios where almost all the elements are in a few buckets, the complexity of the algorithm is limited by the performance of the final bucket-sorting method, so degrades to (N^2). Variations of the algorithm improve worst-case performance by using better-performing sorts such as Quick Sort or recursive Flash Sort on buckets which exceed a certain size limit.

2.8 Insertion Sort

2.8.1 Algorithmic ideas

Insertion Sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position, the position to which it belongs in a sorted array. It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater than the largest, it leaves the element in its place and moves on to the next element. Else it finds its correct position in the sorted array and moves it to that position. This job is done by shifting all the elements larger than the current element in the sorted array to one position ahead.

2.8.2 Operate the algorithm

Step 1: The first element in the array is assumed to be sorted. Take the second element and store it separately in KEY. Then, Compare KEY with the first element. If the first element is greater than KEY, then KEY is placed in front of the first element.

Step 2: Now, the first two elements are sorted. Take the third element and compare it with the elements on its left. This element is placed just behind the element smaller than it. If there is no element smaller than it, place it at the beginning of the array.

Step 3: Similarly, place every unsorted element in its correct position. Finish when every element of the array is set in the correct position.

2.8.3 Time and space complexity

- The best-case time complexity of the Insertion Sort algorithm is $O(N)$ time complexity when the array is already sorted. It means that the time taken to sort a list is proportional to the number of elements in the list; this is the case when the list is already in the correct order. In this case, there is only one iteration since the inner loop operation is trivial when the list is already in order.
- The Insertion Sort algorithm's worst-case (and average-case) complexity is $O(N^2)$. In the worst-case, the time taken to sort a list is proportional to the square of the number of elements in the list.

2.8.4 Improvements and variants

The improvement of Insertion Sort is Binary Insertion Sort, which uses binary search instead of linear search to find the location where an element should be inserted.

- Best-case: The best case is when the element is already in its sorted position. In this case, we don't have to shift any elements; we can insert the element in $O(1)$. But we are using binary search to find the position where we need to insert. If the element is already in its sorted position, binary search take $\log i$ steps. Thus, we make $\log i$ operations for the i^{th} element, so its best-case time complexity is $O(N \log N)$. This occurs when the array is initially sorted in ascending order.
- Average-case: For average-case time complexity, we assume that the elements of the array are jumbled. Thus, on average, we need $O(\frac{i}{2})$ steps for inserting the ith element, so the average-case time complexity of binary Insertion Sort is $O(N^2)$.
- Worse-case: To insert the i^{th} element in its correct position in the sorted, find the position pos, and take $O(\log i)$ steps. However, to insert the element, we need to shift all the elements from pos to $i - 1$. This take i steps in the worst case (when we have to insert at the starting position). We make a total of N insertions. So, the worst-case time complexity of binary Insertion Sort is $O(N^2)$. The worst-case occurs when the array is initially sorted in descending order.

2.9 Bubble Sort

2.9.1 Algorithmic ideas

Bubble Sort is a simple algorithm used to sort a given set of N elements provided in the form of an array with N number of elements. As its name, with every complete iteration, the largest element in the given array bubbles up towards the last or highest index, just like a water bubble rises to the water surface. Starting from the last element of the list, we compare it with its left element. If the element under consideration has a key smaller than its left element, we move it to the left of the sequence by permutation with its left element. Continuing to do so for the problem with N elements, after $N-1$ steps, we get an increasing list.

2.9.2 Operate the algorithm

Step 1: Starting from the first index $i = 1$

Step 2: With $j = n$, Compare 2 adjacent elements in turn and swap places (if the left element $a[j-1]$ is larger than the right element $a[j]$). Reducing j by 1 after the comparison and repeating when $j > i$.

Step 3: After Step 2, the smallest element is in the first position of the array. We increase i by 1 to take the next smallest element to i^{th} position.

Step 4: Step 4: if i still less than N, return to Step 2. Else, stop the algorithm when the array is sorted.

2.9.3 Time and space complexity

It could be optimized by stopping the algorithm if the inner loop didn't cause any swap. If it isn't optimized by stopping the algorithm if the inner loop didn't cause any swap, the algorithm's complexity is $O(N^2)$ in all cases. Let's assume that Bubble Sort is optimized:

- Best-case time complexity: $O(N)$. The best-case occurs when an array is already sorted.
- Worst-case and average-case time complexity: $O(N^2)$. The worst-case occurs when an array is reverse sorted.
- Auxiliary space: $O(1)$

2.9.4 Improvements and variants

We call its result Shaker Sort.

- Any exchange that had taken place during a pass is remembered whether or not
- The last exchange's position is remembered.
- The direction of consecutive passes is alternated.

2.10 Shaker Sort

2.10.1 Algorithmic ideas

Shaker Sort is a variation of Bubble Sort. The Bubble Sort algorithm always traverses elements from the left. It moves the largest element to its correct position in the first iteration, the second-largest in the second iteration, et cetera. Shaker Sort traverses through a given array in both directions alternatively. Shaker Sort does not go through the unnecessary iteration, making it efficient for large arrays.
Shaker Sorts break down barriers that limit Bubble Sorts from being efficient enough on large arrays by not allowing them to go through unnecessary iterations on one specific region (or cluster) before moving onto another section of an array.

2.10.2 Operate the algorithm

There are two stages of the Shaker Sort in one iteration that are listed as follows:

Step 1: In the first stage, similar to the Bubble Sort, loop through the array from right to left. The adjacent elements are compared; if the left element is greater than the right one, we swap those elements. The smallest element of the list is placed at the first position of the array.

Step 2: In the second stage, loop through the array from the leftmost unsorted element to the right. The adjacent elements are compared, and if the right element is smaller than the left element, we swap those elements. The largest element of the list is placed at the last position of the array.

This process stops when the array elements are already sorted.

2.10.3 Time and space complexity

- Best-case: When there is no sorting required, the array is already sorted. The best-case time complexity of cocktail sort is $O(N)$.
- Average-case: It occurs when the array elements are in jumbled order that is not correctly ascending and not properly descending. The average-case time complexity of cocktail sort is $O(N^2)$.

- Worst-case: It occurs when the array elements must be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of cocktail sort is $O(N^2)$.

2.11 Shell Sort

2.11.1 Algorithmic ideas

Shell Sort is mainly a variation of Insertion Sort. In Insertion Sort, we move elements only one position ahead. Many movements are involved when an element has to be moved far ahead. The idea of Shell Sort is to allow the exchange of far items. 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 sub-lists of every h^{th} element are sorted.

2.11.2 Operate the algorithm

Step 1: If the array has N elements, the elements lie at the interval of $h = \frac{N}{2}$. The 0^{th} element is compared with h^{th} element. If the 0^{th} element is greater than the h^{th} element, they swap their value with each other.

Step 2: In the next loop, an interval of $h = \frac{h}{2}$ is taken, and the elements lying at these intervals are sorted.

Step 3: Repeat that process for the remaining elements.

Step 4: When the interval $h = 1$, the array elements lying at the interval of 1 are sorted. The process stops, and the array is sorted.

2.11.3 Time and space complexity

- Best-case: The total number of comparisons for each interval (or increment) is equal to the array's size when it is already sorted. So the time complexity of this case is $O(N \log N)$.
- Average-case: The degree of complexity is determined by the interval picked. The above complexity varies depending on the increment sequences used. The best increment sequence has yet to be discovered. The time complexity of this case is $O(N \log N)$.
- Worst-case: Shell Sort's worst-case complexity is always less than or equal to $O(N^2)$.

3 Experimental results and comments

Because of the wide data range, we use the logarithmic charts to see the difference between the sorting algorithms easier.

3.1 Randomized data

Randomized data												
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	133.4	100009999	1207	900029999	3352.4	2500049999	13428.4	10000099999	122081.6	90000299999	338413.8	250000000000
Insertion Sort	79.2	50384940	708.8	450262331	2139.6	1251931590	7898.2	4994545087	71335.4	45048072272	199670.6	125356000000
Bubble Sort	338.8	100009999	3109.4	900029999	8496.6	2500049999	28152.4	10000099999	172622	90000299999	412914	250000000000
Heap Sort	2	637345	7.8	2150272	14	3771391	31.8	8044933	109	26490560	201	45966586
Merge Sort	28.4	523579	78.4	1738165	135	3026031	271.2	6401895	834.8	20861557	1363.8	36084584
Quick Sort	1	293170	4	956980	7.2	1651077	15.6	3586469	48	11670096	79.8	19983785
Radix Sort	2	320036	7.2	1020038	12.2	1700040	26.2	3400040	77.2	10200040	132	17000040
Shaker Sort	220	67173534	2204.2	598954745	6324.6	1667786183	25846.4	6657154928	234400.4	60096115705	651844.2	167101000000
Shell Sort	2	656169	7.2	2263586	14.6	4497492	31.4	10148983	107.2	33891799	205.8	65925888
Counting Sort	0	60000	1	180000	1	282768	3	532768	10	1532768	17	2532768
Flash Sort	0	102050	1.2	313174	2.4	531030	6	1162094	27.6	5309394	60.8	11872790

Figure 1: Measurement results of randomized data

3.1.1 Running time graph

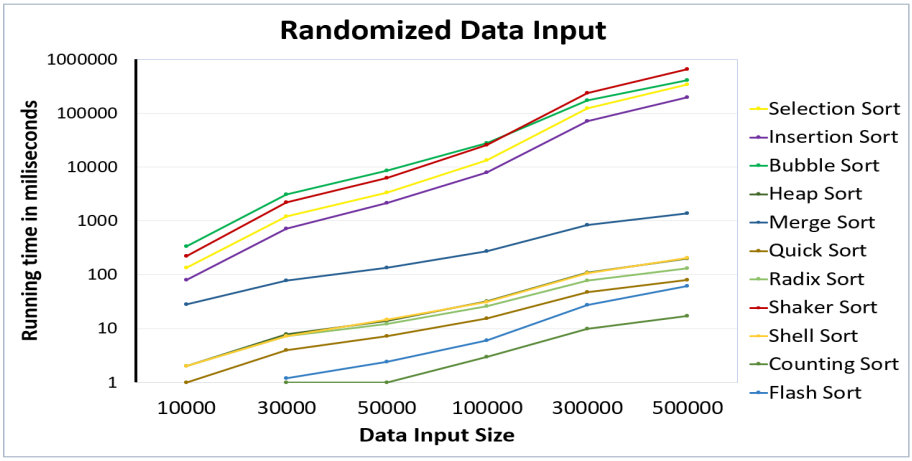

Figure 2: Running time of the algorithms with randomized data

With randomized data:

- The fastest algorithms: Counting Sort, Flash Sort

- The slowest algorithms: Shaker Sort, Bubble Sort, Selection Sort, Insertion Sort

3.1.2 Comparison counting chart

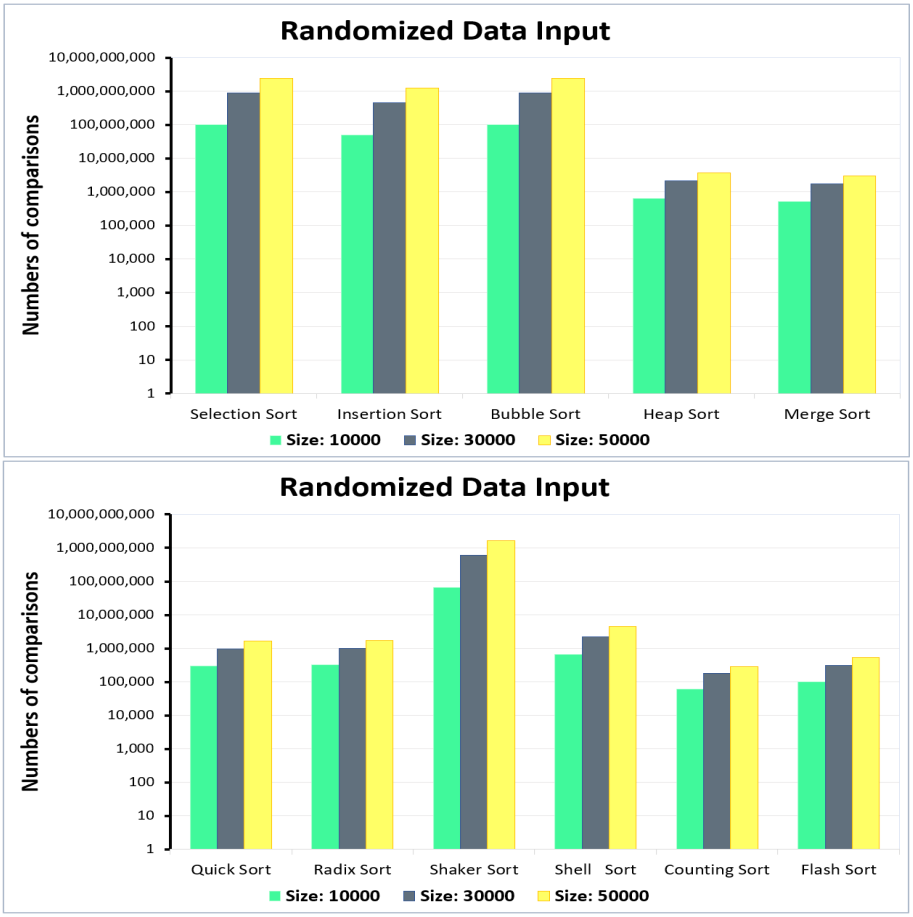

Figure 3: Number of comparisons of algorithms with randomized data

- The least comparisons algorithms: Counting Sort
- The most comparisons algorithms: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort

3.2 Sorted data

Sorted data												
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	142.8	100009999	1279.4	900029999	3343.2	2500049999	13404	10000099999	120943.8	90000299999	339066	250000000000
Insertion Sort	0	29998	0	89998	0	149998	0	299998	1	899998	2	1499998
Bubble Sort	116.4	100009999	1059.4	900029999	2949	2500049999	11740.2	10000099999	106676.4	90000299999	297285.8	250000000000
Heap Sort	2	670333	6	2236652	11	3925355	24.2	8365084	79.6	27413234	140.4	47404890
Merge Sort	24.6	407433	74.8	1345737	126.6	2331993	252.8	4913993	780.4	15911049	1272	27553193
Quick Sort	0	154959	1	501929	1.6	913850	4	1927691	13.2	6058228	22.6	10310733
Radix Sort	2	330028	7.2	1050030	13.2	1850032	28.4	3900034	98	12900038	168.8	21500038
Shaker Sort	0	20001	0	60001	0	100001	0	200001	0	600001	1	1000001
Shell Sort	0	360042	1	1170050	3	2100049	7	4500051	25.8	15300061	41.2	25500058
Counting Sort	0	60000	1	180000	2	300000	3	600000	11	1800000	20	3000000
Flash Sort	0	123491	1	370491	2	617491	4	1234991	13	3704991	21.4	6174991

Figure 4: Measurement results of sorted data

3.2.1 Running time graph

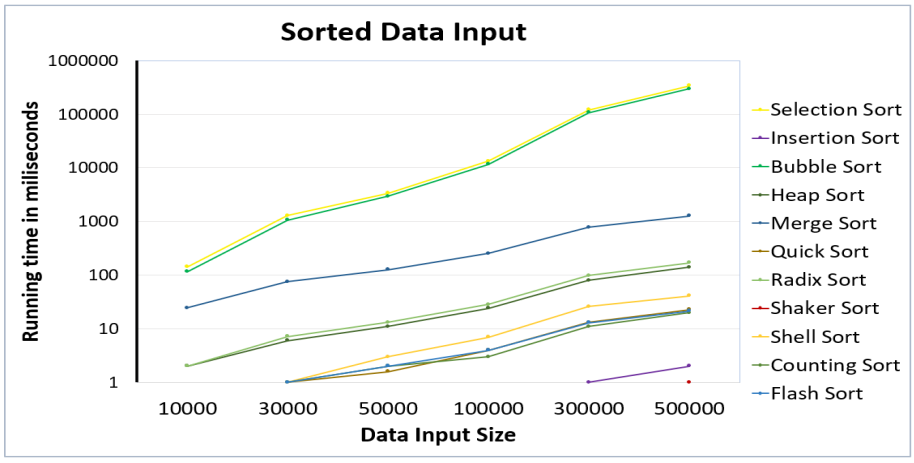

Figure 5: Running time of the algorithms with sorted data

With sorted data:

- The fastest algorithms: Counting Sort, Flash Sort
- The slowest algorithms: Selection Sort, Bubble Sort

3.2.2 Comparison counting chart

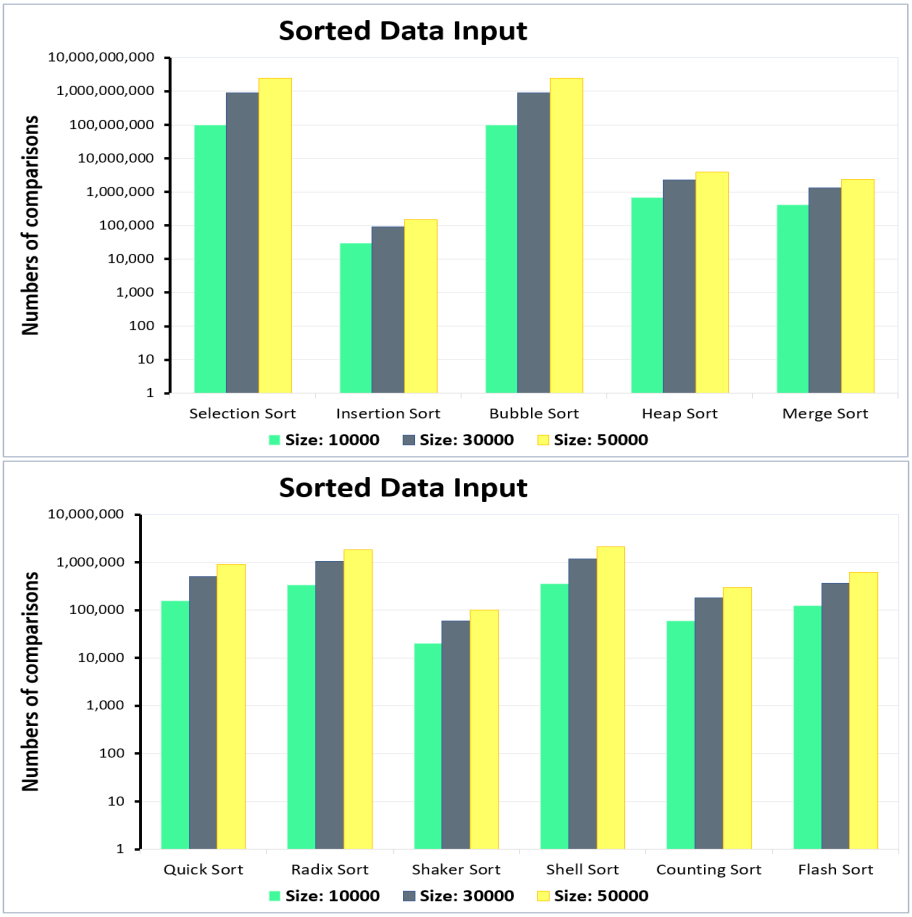

Figure 6: Number of comparisons of algorithms with sorted data

- The least comparisons algorithms: Shaker Sort, Insertion Sort, Counting Sort
- The most comparisons algorithms: Selection Sort, Bubble Sort, Shaker Sort

3.3 Reverse sorted data

Reverse sorted data												
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	253.4	100009999	2296.4	900029999	6389	2500049999	25539.8	10000099999	231340	90000299999	644396.6	250000000000
Insertion Sort	156.6	100009999	1417	900029999	3928.8	2500049999	16113.6	10000099999	142052	90000299999	396228.2	250000000000
Bubble Sort	226.8	100009999	2053.8	900029999	5698	2500049999	22843.8	10000099999	206938.2	90000299999	576604.2	250000000000
Heap Sort	1.2	606775	5.8	2063328	10.8	3612728	24.4	7718947	77.4	25569383	135.4	44483352
Merge Sort	24.4	426232	75.2	1392184	125.6	2420872	253	5091752	778.8	16448680	1265.8	28234632
Quick Sort	0	164975	1	531939	1.6	963861	4	2027703	13	6358249	23.8	10810747
Radix Sort	2	320029	7.6	1020031	13.2	1800033	28.4	3800035	96.4	12600039	164.8	21000039
Shaker Sort	301.4	100015000	2453.2	900045000	6469	2500075000	26017	10000150000	235717.2	90000450000	656649.4	250001000000
Shell Sort	0	475175	2	1554051	4	2844628	9.8	6089190	33.2	20001852	58.8	33857581
Counting Sort	0	60000	1	180000	1	300000	3	600000	17	1800000	19	3000000
Flash Sort	0	106000	1	318000	2	530000	4	1060000	12.4	3180000	21.2	5300000

Figure 7: Measurement results of reverse sorted data

3.3.1 Running time graph

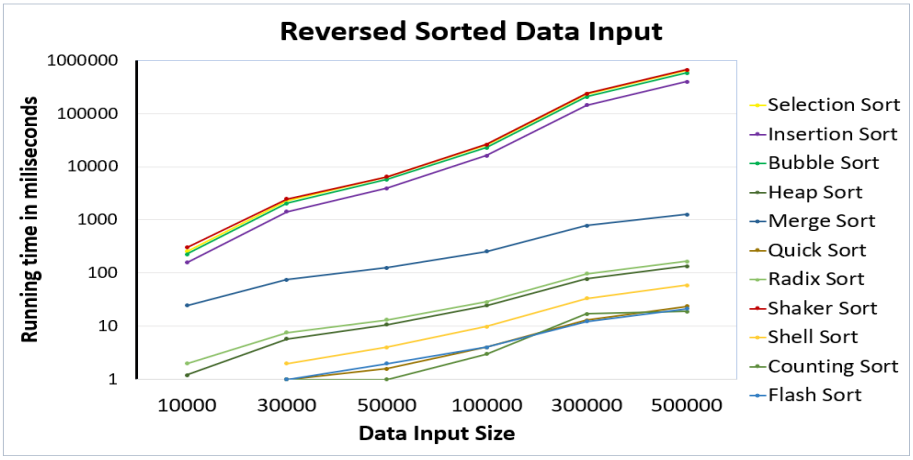

Figure 8: Running time of the algorithms with reverse sorted data

With reverse sorted data:

- The fastest algorithms: Counting Sort, Flash Sort, Quick Sort
- The slowest algorithms: Shaker Sort, Selection Sort, Bubble Sort, Insertion Sort

3.3.2 Comparison counting chart

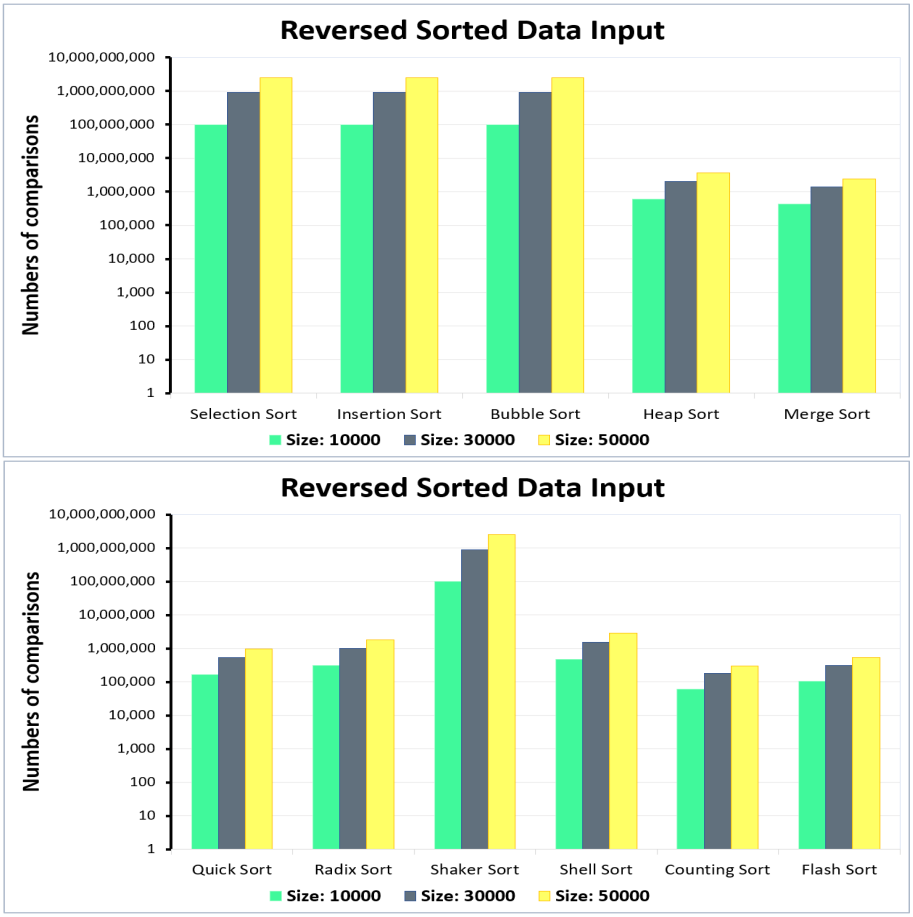

Figure 9: Number of comparisons of algorithms with reverse sorted data

- The least comparisons algorithm: Counting Sort
- The most comparisons algorithms: Shaker Sort, Selection Sort, Bubble Sort, Insertion Sort

3.4 Nearly sorted data

Nearly sorted data												
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	135.6	100009999	1228.2	900029999	3580.2	2500049999	13628	10000099999	123340.4	90000299999	341050	250000000000
Insertion Sort	0	159662	0	348914	0	408914	0	558914	1	1158914	2	1758914
Bubble Sort	120.4	100009999	1095.2	900029999	3034.2	2500049999	12436.4	10000099999	109924.8	90000299999	305704.4	250000000000
Heap Sort	2	669862	6	2236555	11	3925459	25	8365051	81	27413194	143	47404820
Merge Sort	24.6	435568	73.8	1418193	128.8	2416593	253.8	4998593	778.8	15990202	1269	27626648
Quick Sort	0	155011	1	501961	1.6	913874	4.2	1927715	13.8	6058260	23.4	10310757
Radix Sort	2.4	321753	8.4	1028020	15.4	1828022	33	3878024	112.2	12878028	189	21478028
Shaker Sort	0	167945	0.2	404270	0.8	484270	1.2	684270	2.2	1484270	3.4	2284270
Shell Sort	0	393993	2.2	1283208	3.6	2224064	7.6	4624066	26.6	15404023	44.6	25600995
Counting Sort	0	60000	1	180000	1	300000	3	600000	11	1800000	20	3000000
Flash Sort	0	123465	1	370461	2.2	617465	4.4	1234967	14.2	3704967	24.6	6174967

Figure 10: Measurement results of nearly sorted data

3.4.1 Running time graph

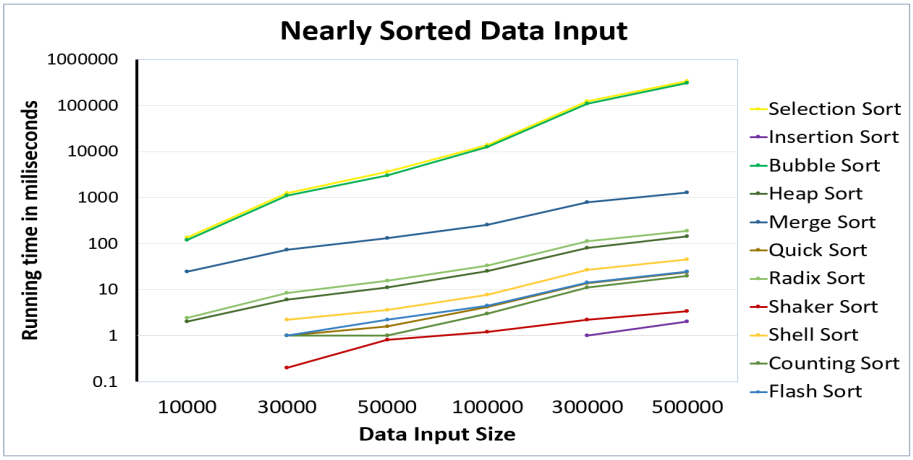

Figure 11: Running time of the algorithms with nearly sorted data

With nearly sorted data:

- The fastest algorithms: Insertion Sort, Shaker Sort, Counting Sort, Flash Sort
- The slowest algorithms: Selection Sort, Bubble Sort, Merge Sort

3.4.2 Comparison counting chart

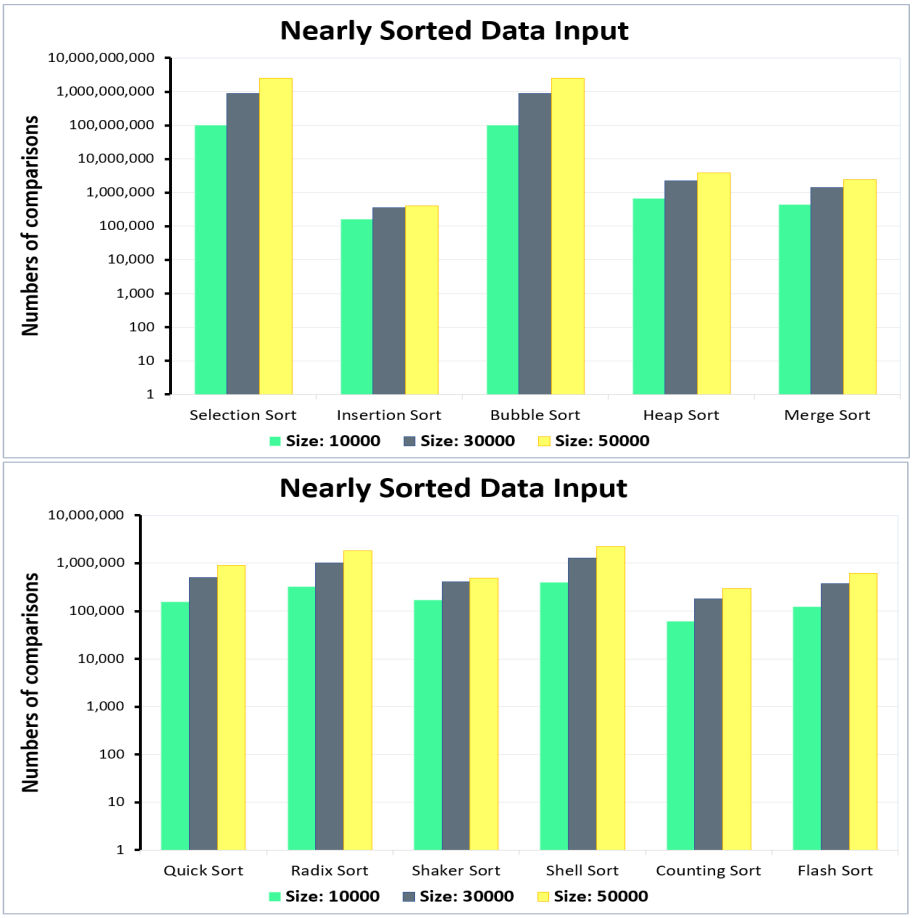

Figure 12: Number of comparisons of algorithms with nearly sorted data

- The least comparisons algorithm: Insertion Sort, Shaker Sort, Counting Sort
- The most comparisons algorithms: Selection Sort, Bubble Sort

3.5 Based on the Graphs and the Data tables

3.5.1 The fastest algorithms: Counting Sort, Flash Sort

Counting Sort is not a comparison-based algorithm. It sorts the elements of an array by counting the number of occurrences of each unique element in the array. So it doesn't take much time to make the comparisons.

Flash Sort as with all bucket sorts, performance depends critically on the balance of the buckets. In the ideal case of a balanced data set, each bucket will be approximately the same size. If the number m of buckets is linear in the input size N, each bucket has a constant size, so sorting a single bucket with an $O(N^2)$ algorithm like Insertion Sort has complexity $O(12) = O(1)$. The running time of the final Insertion Sorts is therefore $m \times O(1) = O(m) = O(N)$.

3.5.2 The slowest algorithms:,Selection Sort, Bubble Sort

The time complexity of both Selection Sort and Bubble Sort is $O(N^2)$.

Bubble Sort works by repeatedly swapping the adjacent elements if they are in the wrong order.
Selection Sort sorts an array by repeatedly finding the minimum element and putting it in the right place.
$\Rightarrow$ No matter how the data is sorted, both algorithms perform the same number of comparisons.

3.5.3 Insertion Sort, Shaker Sort

Insertion Sort virtually splits the array into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.
Shaker Sort is a variation of Bubble Sort. Shaker Sort traverses through a given array in both directions alternatively.
$\Rightarrow$ They will run faster if the data is sorted or nearly sorted.

3.5.4 Stable algorithms

Stable algorithm sort the data without changing the order of the elements with the same values. So, base on the algorithm description: Stable algorithms: Selection Sort, Bubble Sort, Shaker Sort, Counting Sort, Merge Sort,

Insertion Sort, Radix Sort, Flash Sort. Unstable algorithm: Quick Sort, Heap Sort, Shell Sort.

4 Project organization and Programming notes

4.1 Project organization

- Sorting.cpp: contains `main()` funciton
- DataGenerator.cpp and DataGenerator.h: contain all data generator function given by instructors.
- Command.h: contains all prototype of command and support functions.
- All file has suffix "_Command.cpp": Sorting function body, seperated by writers's name.
- Measure.cpp and Measure.h: contain all things for time measure and the definition of sorting function pointer.
- SortAlgorithms.h: contains all prototype of Sorting Algoritms.
- All file has suffix "_Algorithm.cpp": Sorting function body, seperated by writers's name.

4.2 Programing notes

Libraries are included:

- iostream: to use `std::cout` function to print output to console.
- fstream: to use type `std::ifstream` and `std::ofstream`, which support input, output with files.
- string: to use type `std::string` and its support method. It helps handle string easier.
- vector: to use type `std::vector` and its support method. It helps handle dynamic array easier.
- ctime/time.h: get time to measure sorting algortims and set random seed.
- cstdint/stdint.h: get exaact-byte integer to reduce the usage of memory.

Struct `MeasureInfo` included:

- `uint64_t time`: store the result after time-measurement, uint64_t type gives us the biggest limitation
- `uint64_t comparison_counter`: store the result after comparison-measurement, uint64_t type gives us the biggest limitation

5 List of references

- Measure Time: https://www.geeksforgeeks.org/how-to-measure-time-taken-by-a-program-in-c/
- Quick Sort: https://codelearn.io/learning/cau-truc-du-lieu-va-giai-thuat/858618
- Heap Sort: https://www.geeksforgeeks.org/cpp-program-for-heap-sort/
- Counting Sort: https://www.programiz.com/dsa/counting-sort
- Merge Sort:
  - Mr. Nguyen Minh Huy_slide AP-10-Advanced Topics-English_Page 7,8
  - https://www.geeksforgeeks.org/merge-sort/
  - Merge Sort Improvements: https://shorturl.at/ntzIZ
  - Oscillating Merge Sort: https://en.wikipedia.org/wiki/Oscillating_merge_sort
  - Polyphase Merge Sort: https://en.wikipedia.org/wiki/Polyphase_merge_sort
  - 3-way Merge Sort: https://www.geeksforgeeks.org/3-way-merge-sort/
- Radix Sort: https://www.geeksforgeeks.org/radix-sort/
- Flash Sort:
  - Wikipedia: https://en.wikipedia.org/wiki/Flashsort
  - CodeLearn: https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh
- Insertion Sort:
  - https://www.programiz.com/dsa/insertion-sort
  - https://cafedev.vn/thuat-toan-insertion-sort-gioi-thieu-chi-tiet-va-code-vi-du-tren-nhieu-ngon-ngu-lap-trinh/
  - https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/
  - https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/tutorial/
  - Binary Insertion Sort - GeeksforGeeks: https://www.geeksforgeeks.org/binary-insertion-sort/
  - Binary Insertion Sort - Interview Kickstart. https://www.interviewkickstart.com/learn/binary-insertion-sort
- Bubble Sort:
  - https://www.happycoders.eu/algorithms/bubble-sort/#Bubble_Sort_Time_Complexity
  - https://viblo.asia/p/sap-xep-noi-bot-bubble-sort-la-gi-1VgZvN82ZAw
  - https://www.geeksforgeeks.org/bubble-sort/
- Shaker Sort:
  - https://www.javatpoint.com/cocktail-sort
  - https://www.geeksforgeeks.org/cocktail-sort/
- Shell Sort:
  - Shell Sort Algorithm In Data Structures: Overview, Time Complexity https://www.simplilearn.com/tutorials/data-structure-tutorial/shell-sort
  - GeeksforGeeks https://www.geeksforgeeks.org/shellsort/