VIETNAM NATIONAL UNIVERSITY HO CHI MINH

UNIVERSITY OF SCIENCE

---o0o---

DATA STRUCTURES AND ALGORITHMS – CSC10004

REPORT

Lab 03

SORTING ALGORITHMS

Instructors: Văn Chí Nam — Lê Thanh Tùng
Bùi Huy Thông — Trần Thị Thảo Nhi

Group 9's members:

22127275	–	Trần Anh Minh
22127280	–	Đoàn Đặng Phương Nam
22127465	–	Bùi Nguyễn Lan Vy
22127475	–	Diệp Gia Huy

HO CHI MINH CITY, 2023

TABLES OF CONTENTS

FOREWORD

A sorting algorithm is an algorithm that puts elements of a list into an order. The most frequently used orders are numerical order and lexicographical order, and either ascending or descending.

Sorting algorithms help a lot in sorting a list of people in real life, either put into alphabetical by surname or into numerical by age. Sorting algorithms can also be used in library management systems to sort books. This proves the importance of sorting algorithms in creating and managing various kinds of management systems, including school, library, hospital, etc.

In this report, we present a C++ program that implements 11 sorting algorithms, describe the ideas and how they work step-by-step, estimate time and space complexity.

We welcome feedback and constructive criticism from our teachers who will be grading our project. This will help us to identify any areas where improvements can be made and allow us to refine our system. We are very open to suggestions to improve our program.

Group of authors

I. Introduction and information

1. Members Information

• 22127275 - Trần Anh Minh

• 22127280 - Đoàn Đặng Phương Nam

• 22127465 - Bùi Nguyễn Lan Vy

• 22127475 - Diệp Gia Huy

2. CPU Information

We use laptop with the following configuration:

• Processor: AMD Ryzen 7 4800H with Radeon Graphics

• CPU clock: 2.9 GHz

II. Algorithm presentation

1. Bubble Sort

1.1. Algorithmic ideas

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent element.

It gets its name from the way the smaller elements “bubble” up (move to the left) during each pass. The algorithm compares each element with its adjacent element and swaps them if they are in the wrong order. This process continues until the largest element "bubbles" to the end of the array.

After one pass, the largest element is in its final position. The algorithm then repeats this process for the remaining elements until the array is fully sorted.

1.2. Step-by-step descriptions

Step 1: Initializing **i = 0**, as the size of the sorted part of the array.

Step 2: Initialize **j = 0**, increasing towards the end of the unsorted part of the array, comparing the **2** adjacent elements, and performing a swap if the first element is larger than the second one.

Step 3: After **step 2**, the current largest will be swapped to its place at the end of the array, increasing the size of the sorted part.

Step 4: Until all the array is sorted, back to **step 2**.

1.3. Time and space complexity

- Best-case time complexity is $\boldsymbol{O(n)}$, occurring when the array is already sorted.
- Both the worst and average case is $\boldsymbol{O(n^2)}$, in which the worst-case scenario happens when the array is in reversed order.
- Space complexity: $\boldsymbol{O(1)}$, we don’t need to use any additional memory.

1.4. Variants/Improvements

- **Odd – even Sort** (**Brick Sort**)**:** For message passing systems. Time complexity is **O(N^2)** and auxiliary space is **O(1).**
- **Cocktail Sort** (**Cocktail Shaker Sort**): This sorting algorithm traverses through the given array in both directions alternatively. It is also efficient for large array, rather than Bubble Sort. Time complexity is **O(N)** in *best case*, **O(N^2)** in *average* and *worse case*. Auxiliary space required is **O(1)**.
- We can put a flag inside the loop of bubble sort to break out in case that loop has no swap,

which means the array is already sorted. This way can reduce the number of passes.

2. Counting Sort

2.1. Algorithmic ideas

Counting Sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

2.2. Step-by-step descriptions

Step 1: Find the maximum (called **max**) and minimum (called **min**) elements of the array of length **N** (called ***arr***).

Step 2: Initialize an array of length **max – min + 1** with all elements **0**. This array is used for storing the count of the elements in the array ***arr***.

Step 3: Run a loop from **0** to **N – 1**, at $\mathbf{i^{th}}$ element of array ***arr***, supposing that the value of that element is **X**, add **1** to the $\mathbf{(X - min)^{th}}$ element in the counting array.

Step 4: Initialize a variable called **índex**, assign **index** to **0**

Step 5: Run a loop from **0** to **max – min**, at $\mathbf{i^{th}}$ element of the counting array, supposing that the value of that element is **Y**. Compare **Y** with **0**, if **Y = 0**, continue the loop, otherwise, initialize another loop from **1** to **Y**, in each attempt, replace the $\mathbf{(index)^{th}}$ element with **i** (**i** is the index of **Y** in the counting array) and add **1** to **index** .

2.3. Time and space complexity

With **N** is the number of elements in the array, **max** is the maximum value and **min** is the minimum value in the array.

- Time complexity: **O(max(N, max – min + 1))** for all cases.
- Auxiliary space: **O(max(N, max – min + 1))**

3. Flash Sort

3.1. 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.

3.2. Step-by-step descriptions

Supposing that we are having an array **arr** of length **N** needing rearranging.

Step 1: Get the number of buckets (called **M**) used for this algorithm, normally, we will create the number of buckets by multiplying **N** by **0.45.**

Step 2: Compare **M** with **0**, if **M = 0**, using **Insertion Sort** algorithm to rearrange the array, otherwise, move to **step 3.**

Step 3: Find the minimum value (called **min**) and the position of the maximum value (called **indexmax**) in the array.

Step 4: Create an array saving the position of the last element in each bucket (called **EndOfBucket**), firstly, initialize the value **0** for all elements of this array, then with a loop from **0** to **N – 1**, with **ith** element, identify the index of bucket corresponding with it through the calculation below:

$$X = \left[\frac{(M-1)*(arr[i]-min)}{arr[indexmax]-min}\right] \textbf{ (1)}$$

Note that **[x]** is the maximum integer which is less than or equal to **x**, and **X** is the index of bucket corresponding with **ith** element.

Then, add **1** to the **Xth** element in the **EndOfBucket** array and continue the loop.

Step 5: Use prefix sum with the **EndOfBucket** array to identify precisely the position of the last element in each bucket

Step 6: Move the maximum number to the first position, then run a loop from **0** to **N – 1**, in each attempt, use **(1)** to identify the respective bucket of **arr[0]** and swap **arr[0]** and **arr[EndOfBucket[X] - 1]**, along with substract the **Xth** element in the **EndOfBucket** array by **1**.

Step 7: Using **Insertion Sort** algorithm to sort each bucket.

3.3. 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 $\mathbf{O(N^2)}$ algorithm like insertion sort has complexity $\mathbf{O(1^2) = O(1).}$ The running time of the final insertion sorts is therefore $\mathbf{M \cdot O(1) = O(M) = O(N).}$

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 $\mathbf{O(N^{\frac{3}{2}})}$.

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 $\mathbf{O(N^2).}$ Variations of the algorithm improve worst-case performance by using better-performing sorts such as **Quicksort** or **recursive Flashsort** on buckets which exceed a certain size limit.

4. Heap Sort

4.1. Algorithmic ideas

Heap Sort is based on **Binary Heap** data structure. First, *heapify* the array to convert it into heap data structure, build a *max-heap* where the largest element is at the root. Then repeatedly swap the root with the last element of the heap, reducing the size. Now heapify the remaining elements of the heap and do the previous work again. The process is done once heap contains only one element.

4.2. Step-by-step descriptions

Step 1: Building a max-heap from the input array. (Start from the middle moving to the first element, and perform a swap if the parent is less than its children)

Step 2: Swap the root element with the last element of the heap.

Step 3: Reduce the size of the heap by one (exclude the recently swapped element)

Step 4: Restore the heap property by heapify the new root.

Step 5: Until the heap size is **1**, back to **step 2**.

4.3. Time and space complexity

- Time complexity: **O(N log N)** in *all cases*.
- Auxiliary space: **O(1)**

5. Insertion Sort

5.1. Algorithmic ideas

The list of elements is virtually splited into two parts: unsorted and sorted elements. The value from the unsorted part will be picked up and then placed exactly at the correct position in the sorted part. It starts by considering the first element as the sorted portion and then inserts the other element into its correct position within the sorted part. Each time, the algorithm compares the current element to its predecessor or the elements before (only if the current element is smaller than its predecessor) in the sorted portion, shifting larger elements to the right until it finds the correct position for insertion.

5.2. Step-by-step descriptions

Step 1 : Consider first element of the list is in sorted part

Step 2 : Take the element next to sorted part and save it as a key

Step 3 : Compare the key with the whole sorted list and place it in correct order

Step 4 : Back to **Step 2** until the whole list in sorted part

5.3. Time and space complexity

- Time complexity:
  - Best case: **O(N).**
  - Average case and Worst case: **O(N^2).**
- Auxiliary space: **O(1).**

5.4. Variants/Improvements

Binary Insertion Sort:

The idea is to use Binary Search to find where to place each element. The goal is to reduce the number of comparisons because **Binary Insertion Sort** compares fewer elements using binary search instead of linear search.

Time complexity is **O(N*logN)** in *best case*, and is **O(N^2)** in *average case* and *worst case.*

Auxiliary space required is **O(1)** if use *iterative binary search*, is **O(N*logN)** if use *recursive binary search* (because of **O(logN)** recursive calls).

6. Merge Sort

6.1. Algorithmic ideas

Following the concept of the divide-and-conquer technique, the algorithm works by:

- Repeatedly divide the input array into smaller halves until the sublists contain only one element (a list of one element is considered sorted).
- Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

6.2. Step-by-step descriptions

Step 1: Recursively calls the function to split the array into **2** halves until every sub-array only contains **1** element, which is now considered sorted.

Step 2: Merge the adjacent sub-arrays into a bigger sorted array.

Step 3: Recursively combine them until there is only one left, the array itself.

6.3. Time and space complexity

- Time complexity: **O(N log N)** in *all cases* because divide the array into two halves, and take linear time to merge two halves.
- Auxiliary space: **O(N)**, require additional memory to store the subarrays.

6.4. Variants/Improvements

- **In-place merge sort**: it is possible to modify merge sort to perform in-place, improving the space complexity to $\boldsymbol{O}(\mathbf{1})$. Though it will cause a slighter higher time complexity due to extra swaps and comparisons.
- **3-way merge sort**: merge sort involves splitting the array into **2** parts, though this variant will break the array into 3 parts respectively.

7. Quick Sort

7.1. Algorithmic ideas

- **Quick Sort** applies the idea of divide-and-conquer algorithm.
- Choosing the pivot element from the input list, and partition it into 2 sublists, one with elements less than the pivot and the other with elements greater than or equal to the pivot.
- This partitioning process is repeatedly applied to sublists until the sublist only contains 1 element and the whole array is sorted.

7.2. Step-by-step descriptions

Step 1: Choosing a pivot element from the array. The choice of pivot can be a matter of the algorithm's performance.

Step 2: Partition the array around the chosen pivot. Reordering elements that are less than the pivot will come before it and all others are placed behind.

Step 3: Now the pivot element is in the correct sorted position. Recursively apply the algorithm to the 2 halves of the array separated by the pivot element.

Step 4: The recursion continues until each sub-array contains only one element. Now the whole array is sorted.

7.3. Time and space complexity

The time complexity largely depends on the way of choosing the pivot:

- The best and average case will be $\boldsymbol{O(N.\log_2 N)}$, in the situation in which the pivot is the middle element.
- The worst-case scenario is $\boldsymbol{O(N^2)}$, occurs when the algorithm always chooses the largest or smallest element as a pivot.

The space complexity: $\boldsymbol{O(1)}$, there is no need for external memory when swapping elements during the process.

7.4. Variants/Improvements

The decision of choosing the pivot matters a lot in the time complexity, therefore there are some methods to solving this:

- **Median-of-three**: choose the pivot from one of the following: first, middle, or last element.
- **Pick randomly**: This call **Randomize Quick Sort**, which should minimize the occurrence of worst-case.

Other improvements and variants can be:

- **Multi-pivot Quick sort**: choosing multiple pivots and partitioning them into compartments accordingly, which will improve the performances of data distributions.
- Instead of partitioning the array into 2 sub-arrays, the **Three-way Quick Sort** split the array into 3 parts: less than, equal to, and larger than the pivot itself. By this, the algorithm prevents redundant operations on duplicate elements.

8. Radix Sort

8.1. Algorithmic ideas

Radix Sort is a non-comparison-based sorting algorithm that operates on the individual digits of the input elements.

The idea is to sort elements according to the least significant digit and progressivvely toward the more significant ones. move to the more significant ones by putting them into different buckets according to the value of the current digit.

8.2. Step-by-step descriptions

Step 1: This process continues until all digits or bits have been considered, resulting in a fully sorted array.Find the maximum number of the meaning digits. This will determine the number of passes required for the sorting process.

Step 2: Starting from the rightmost digit, perform placing elements into the "buckets" correspondingly to its digit.

Step 3: From the order of the buckets and the order the element itself was put inside each bucket, copy it back to an array.

Step 4: Move to the next digit on the left for consideration, and back to **step 2**.

8.3. Time and space complexity

With N is the number of elements in the list, K is the number of digits of the largest element in the list.

- Time complexity: **O(NK)**
- Auxiliary space: **O(N),** in consideration of creating buckets for each digit's value and copying the sorted elements back to the array.

Radix Sort is often faster than other comparison-based sorting algorithms for large datasets. Its time complexity grows linearly with the number of digits so it is not efficient for small datasets.

8.4. Variants/Improvements

Radix Sort can be performed by using different variations as Least Significant Digit (LSD) Radix Sort of Most Significant Digit (MSD) Radix Sort.

- **LSD Radix Sort**: Started sorting at the least significant digit and worked up digit by digit to the most significant digit.
- **MSD Radix Sort**: Alternatively, we start at the most significant digit. MSD requires that after the hundreads place, we must sort the hundreads place buckets within themselves instead of the whole input list. It means we sort the buckets recursively.

Radix Sort has a version that starts with the largest digit and works way down, which can be beneficial for sorting strings,…

Multiple-key Radix Sort: when elements can have multiple keys or attributes, the algorithm can sort the array simultaneously, partitioning in buckets and performing radix sort on the corresponding key.

9. Selection Sort

9.1. Algorithmic ideas

Selection Sort works by repeatedly selecting the smallest from the unsorted part of the list and then moving it to the sorted part of the list. The largest element can be chosen first instead based on the approach. The moving process is to swap the chosen element with the first element of the unsorted part. This process is repeated until the entire list is sorted.

9.2. Step-by-step descriptions

Step 1 : Initialize **cur_idx = 0**

Step 2 : From **cur_idx + 1** to **N – 1**, find out the element has smallest value and save its index to **min_idx**

Step 3 : Swap value of array at indexes **cur_idx** and **min_idx**

Step 4 : If **cur_idx < N**, increase **cur_idx** by **1** and back to **step 2**. If not the whole list has already sorted.

9.3. Time and space complexity

- Time complexity: **$O(N^2)$** in *all cases,* as there are two nested loops.
  - Best case occurs when the array is already sorted.
  - Worst case occurs when the array is in descending order but need to be sorted in ascending order, and vice versa.
- Auxiliary space: **O(1)**.

9.4. Variants/Improvements

- **Bingo Sort**: Each distinct element is considered a Bingo value and called out in increasing order. During each pass, if the array element is equal to Bingo element then it will be shifted to its correct position. While **Selection Sort** does one pass through the remaining elements for each element moved, **Bingo Sort** does one pass for each distinct value and moves every elements (which store the value chosen) to its final position.
  - With M is the number of distinct elements, N is the size of the array: Time complexity is **O(MN)** in *average* and *worst case,* **$O(N + M^2)$** in *best case.*
  - **Bingo Sort** should be used if the repetition of every element is large because of better time complexity.
- **Recursive Selection Sort**: Putting the chosen minimum element at the end of the sorted part (considering ascending order).

10. Shaker Sort

10.1. Algorithmic ideas

Knowing as **Bidirectional Bubble Sort**, this algorithm is an improved version of **Bubble Sort**, which allows the sorting process to work in both directions. Similar to **Bubble Sort**, after each pass, the large element will be moved to the back and the smaller ones will end up at the front. This alternating continues until no more swaps are needed.

10.2. Step-by-step descriptions

Step 1: Set the left and right pointers to the beginning and end of the array.

Step 2: Move from left to right, comparing the adjacent elements and performing a swap if the first element is large than the second one.

Step 3: Then in the opposite, move from right to left, continue comparing each pair of adjacent elements, and swap correspondingly.

Step 4: After each pass, increase the left pointer and decrease the right one. Until the left and right pointers meet up, back to **step 2**.

Step 5: The process ends when no more swaps occur and the array is sorted.

10.3. Time and space complexity

Although Shaker Sort reduces the number of unnecessary comparisons and swaps, which makes it more efficient than its ancestor, Shaker Sort still has the time complexity of $\boldsymbol{O(N^2)}$ for both the average and worst-case scenario and it is $\boldsymbol{O(N)}$ for the best case where all elements are in their correct position.

No extra space is needed so the space complexity is $\boldsymbol{O(1)}$.

11. Shell Sort

11.1. Algorithmic ideas

Shell Sort is a variation of Insertion Sort. This algorithm starts by sorting pairs of elements far apart from each other, then reducing the gap between those elements progressively until the gap becomes 1. Call the gap size is **h**, first initialize the value of **h**, divide the list into sublists in which have equal intervals to **h** and sort these sublists using inserting sort. Repeat this process until the list is fully sorted.

11.2. Step-by-step descriptions

Step 1: Define the sequence that the elements are being compared with during the process. (Usually based on the array size)

Step 2: Start with the largest gap, iterate over the array

Step 3: Compare and perform a swap, if necessary, the **2** elements that are "gap" positions apart. Repeat the process for all elements in the array.

Step 4: Reduce the gap following the chosen sequence, back to **step 3** until the gap is **1**.

Step 5: Once the gap is **1**, it resembles Insertion Sort, then the array is sorted.

11.3. Time and space complexity

- Time complexity:
  - Best case: **O(N log N)**, occurs then the array is already sorted, the total number of comparisons of each interval is equal to the size of the array.
  - Average case: This depends on the gap size (which is chosen by programmer)
  - Worst case: **O(N^2).**
- Auxiliary space: **O(1).**

11.4. Variants/Improvements

Dobosiewicz Sort: improves on bubble sort in the same way that Shellsort improves on insertion sort. The underlying idea of both the sorting algorithms is same. In the bubble sort method, the elements are compared with the succeeding elements. In the comb sort, the elements are sorted in a certain gap, decreasing the gap after each iteration by dividing the gap factor by the decreasing factor (also called the shrink factor) which is **1.3**.

III. Experimental results and comments

1. Experimental results

1.1. Randomized data.

1.1.1. Statistic table

Data order: RANDOMIZE												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting Statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Bubble Sort	595.479	99,942,160	5418.95	900,050,001	14570.4	2,500,032,920	57382.9	10,000,066,045	513952	90,000,391,152	1436930	249,999,489,560
Counting Sort	0.3999	40,000	1.3949	120,000	1.4642	182,768	2.191	332,768	5.2879	932,768	9.3971	1,532,768
Flash Sort	0.417	103,308	1.1798	330,477	1.8513	544,705	3.5362	1,047,503	10.4275	3,525,683	17.5506	6,045,207
Heap Sort	2.0641	383,118	7.1269	1,290,735	16.8049	2,264,877	26.8435	4,829,709	97.1371	15,902,520	164.846	27,596,964
Insertion Sort	141.416	50,126,147	1398.92	451,707,034	3573.9	1,248,152,397	14544.9	4,996,899,593	137411	44,960,162,813	359524	125,068,299,550
Merge Sort	2.1608	506,975	6.9666	1,684,990	7.9585	2,942,139	16.3373	6,233,869	55.28	20,347,429	106.672	35,177,282
Quick Sort	1.1716	309,131	3.9237	1,064,575	8.2133	1,788,759	9.832	3,948,979	44.8994	14,531,101	72.2425	26,904,195
Radix Sort	1.8909	110,097	7.2897	390,120	12.3142	650,120	23.2561	1,300,120	80.4659	3,900,120	131.492	6,500,120
Selection Sort	231.893	100,019,998	2173.15	900,059,998	6039.12	2,500,099,998	24135.1	10,000,199,998	215658	90,000,599,998	598300	250,000,999,998
Shaker Sort	416.982	75,264,350	3983.38	674,834,958	11003.3	1,880,513,010	42259.4	7,484,826,744	382225	67,311,957,498	1038750	187,697,095,158
Shell Sort	0.5436	560,940	1.8014	1,985,441	3.2747	3,828,031	6.5782	8,344,529	20.6372	31,787,500	27.9495	61,559,473

Figure 1: Measurement results of randomized data input

1.1.2. Running time line graph

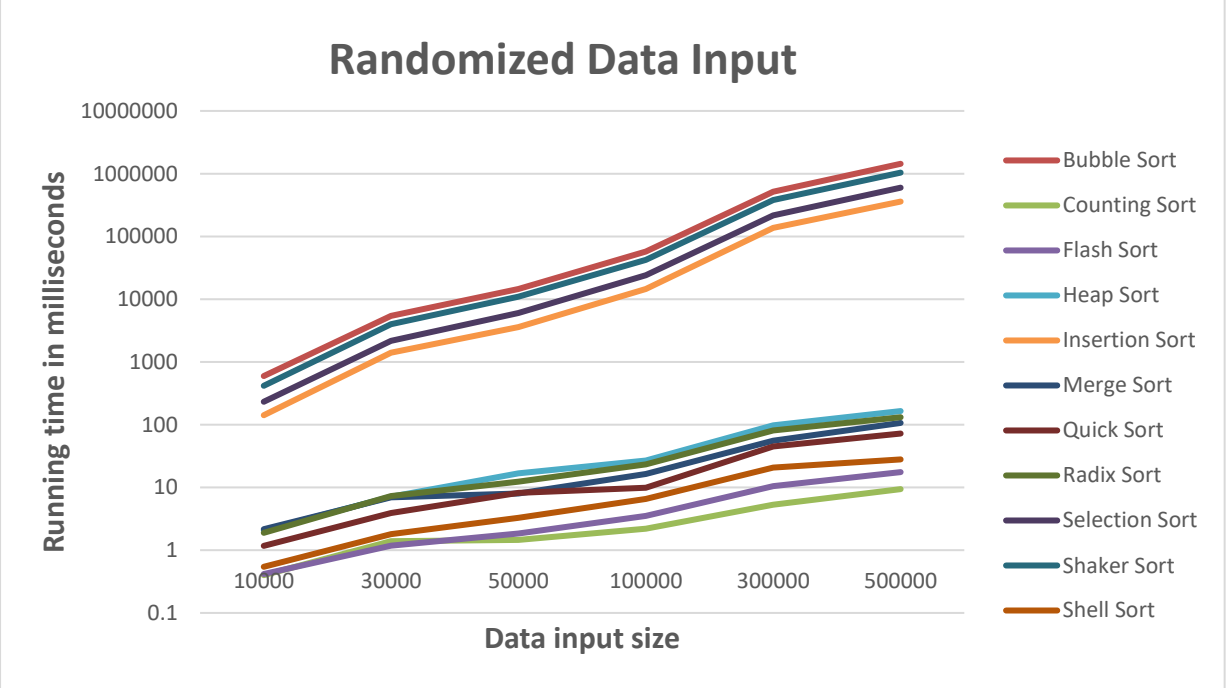


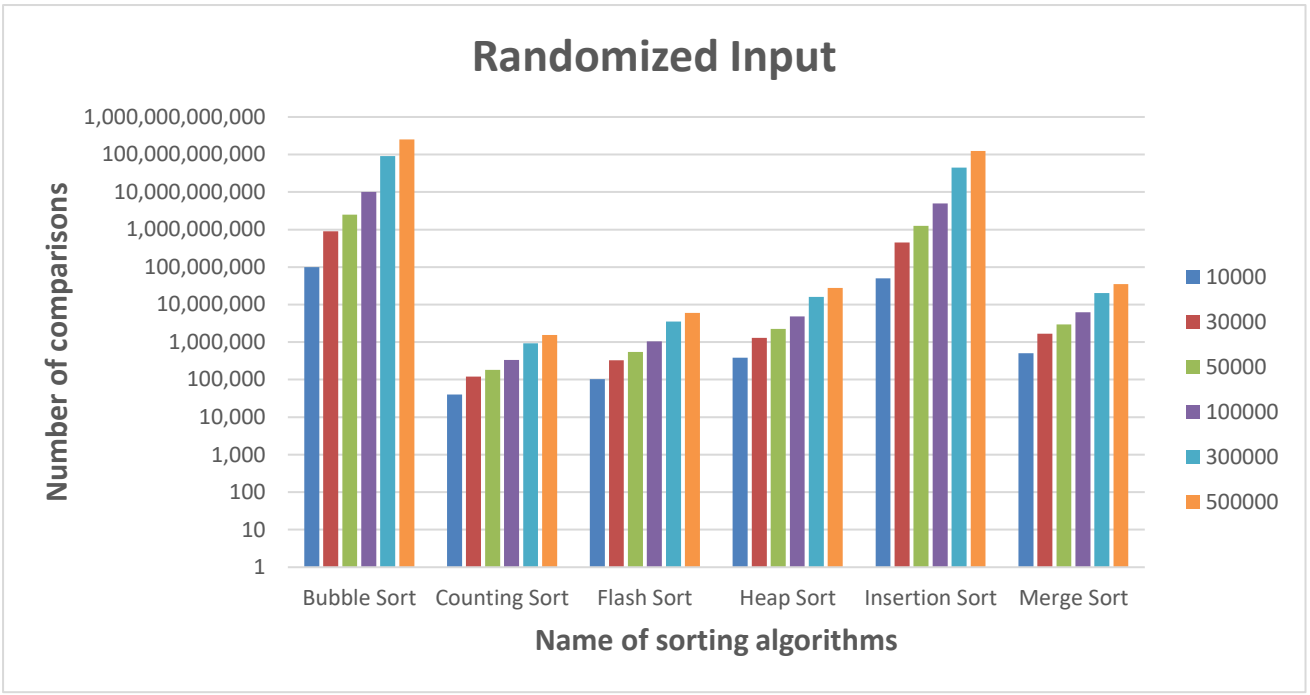
Figure 2: Running time line graph of algorithms with randomized data input

From the running time line graph of algorithms with randomized data input, we can see:

+ The fastest algorithms: Counting Sort, Flash Sort.

+ The slowest algorithms: Bubble Sort, Insertion Sort, Selection Sort, Shaker Sort.

1.1.3. Comparison bar chart



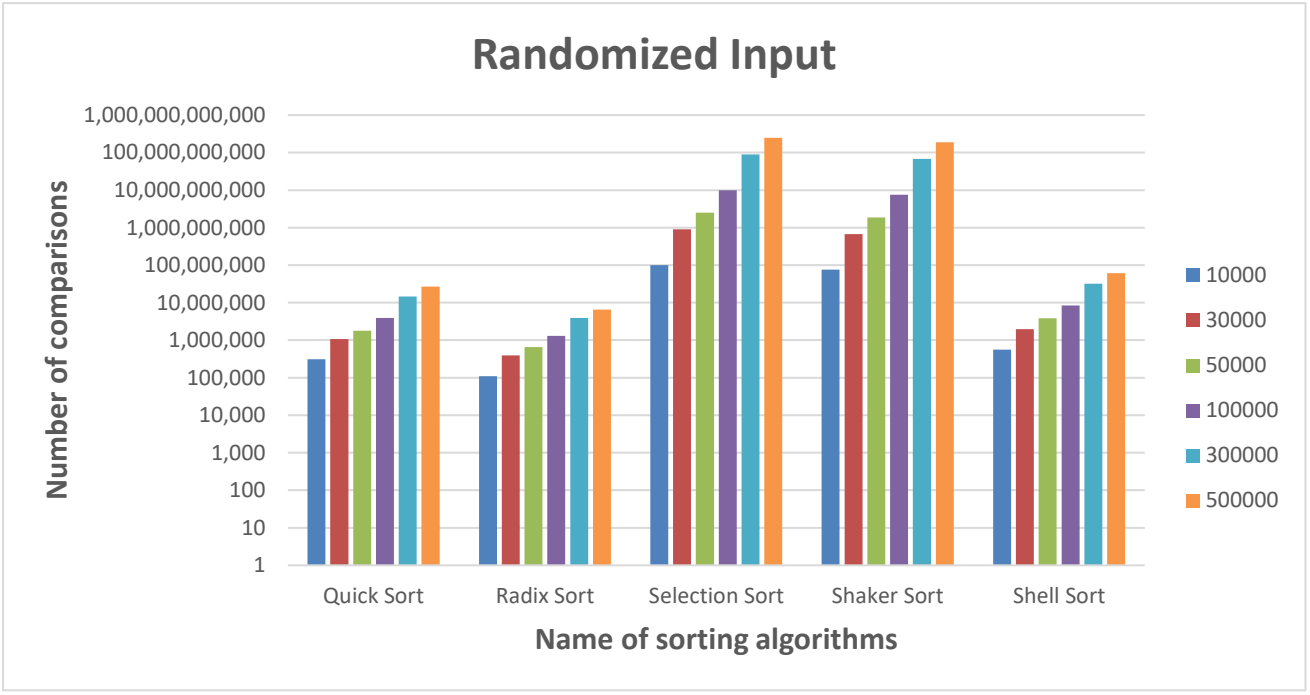


Figure 3: The number of comparisons of algorithms with randomized input

From the bar chart for counting comparisons of algorithms with randomized data input, we can see:

+ The least comparisons algorithms: Counting Sort.

+ The most comparisons algorithms: Bubble Sort, Insertion Sort, Selection Sort, Shaker Sort.

1.2. Sorted data.

1.2.1 Statistic table

Data order: SORTED												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting Statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Bubble Sort	0.044	20,001	0.2241	60,001	0.1412	100,001	0.4479	200,001	0.8897	600,001	3.1476	1,000,001
Counting Sort	0.3592	40,000	0.9064	120,000	1.7485	200,000	3.5226	400,000	8.7234	1,200,000	11.7491	2,000,000
Flash Sort	0.3583	83,514	0.9284	250,514	1.4194	417,514	3.4619	835,014	8.7412	2,505,014	14.8648	4,175,014
Heap Sort	2.2151	396,864	6.9638	1,329,555	19.7353	2,333,214	24.82	4,969,323	95.5671	16,322,181	149.207	28,268,367
Insertion Sort	0.058	29,998	0.1997	89,998	0.2317	149,998	0.6268	299,998	1.3141	899,998	5.3912	1,499,998
Merge Sort	2.1136	396,235	5.5106	1,302,187	8.1428	2,270,875	15.7371	4,791,755	52.1527	15,548,683	92.5694	26,734,635
Quick Sort	0.7322	262,447	4.4576	880,323	4.5689	1,548,623	9.912	3,297,215	29.4856	10,805,247	63.8083	18,708,747
Radix Sort	2.7611	110,097	6.9178	390,120	11.3901	650,120	24.9151	1,300,120	101.139	4,500,143	130.256	7,500,143
Selection Sort	145.441	100,019,998	1335.84	900,059,998	3679.51	2,500,099,998	15141	10,000,199,998	143358	90,000,599,998	584693	250,000,999,998
Shaker Sort	0.075	39,998	0.2455	119,998	0.4577	199,998	0.5621	399,998	2.6645	1,199,998	7.3479	1,999,998
Shell Sort	0.492	225,757	1.2222	767,188	2.5586	1,367,188	4.7265	2,901,472	15.3919	9,604,315	34.473	16,804,315

Figure 4: Measurement results of sorted data input

1.2.2 Running time line graph

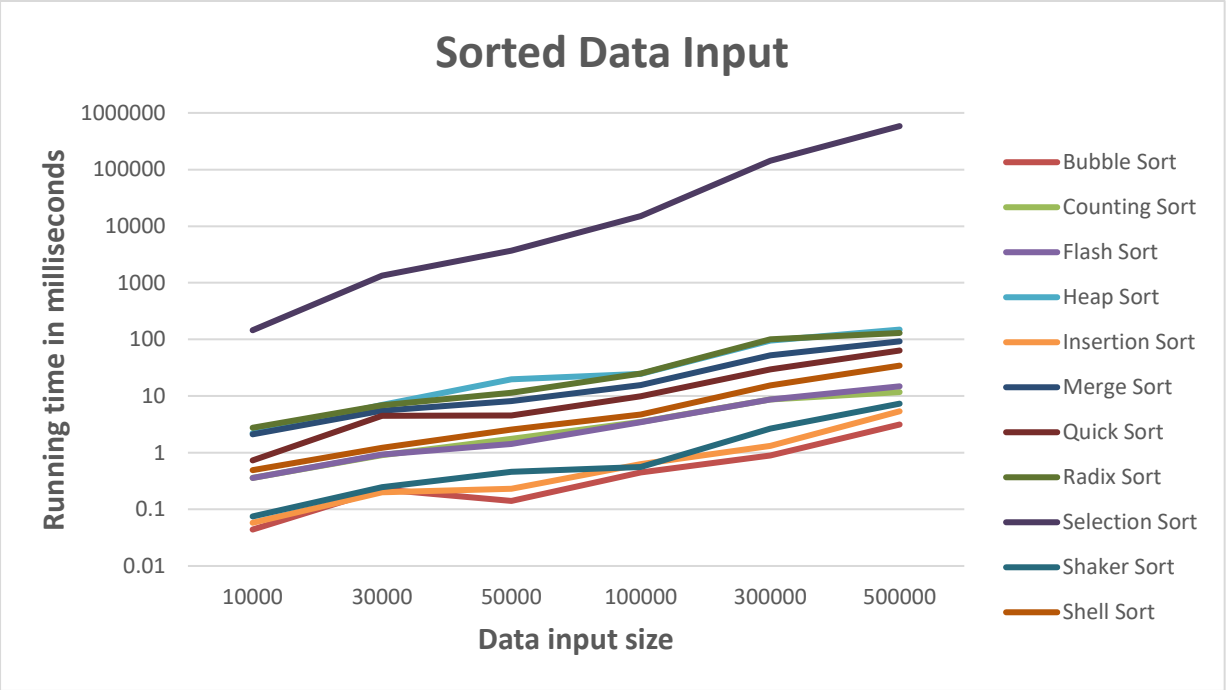


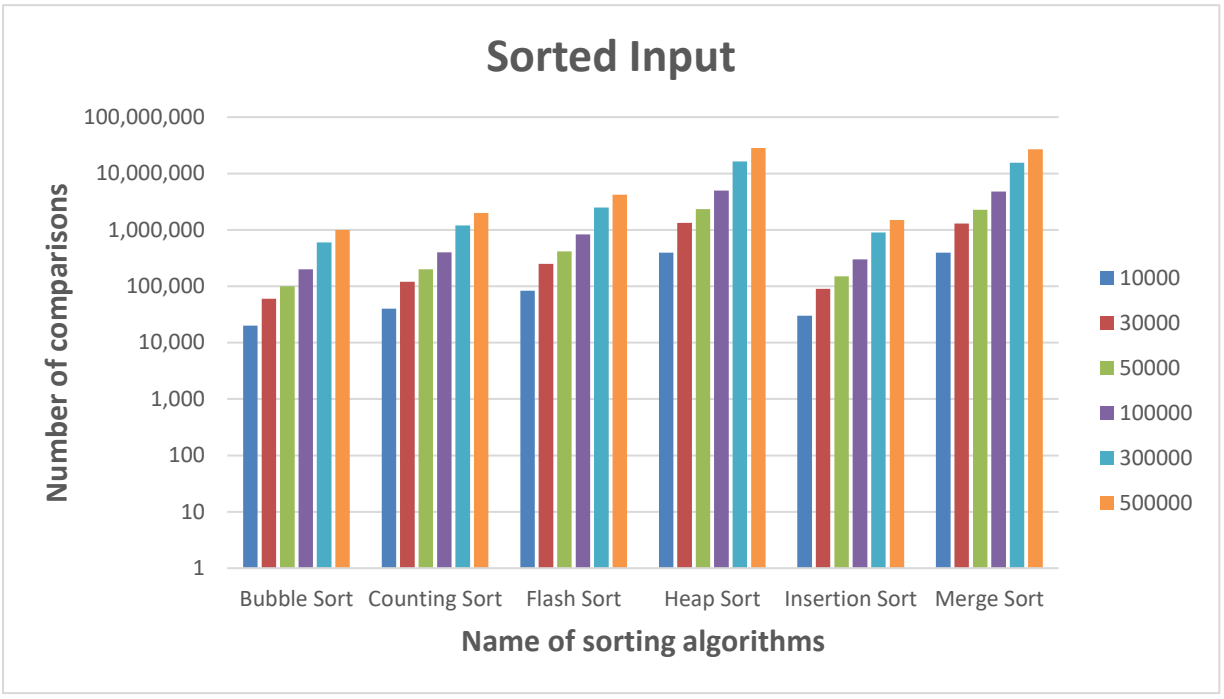
Figure 5: Running time line graph of algorithms with sorted data input

From the running time line graph of algorithms with sorted data input, we can see:

+ The fastest algorithms: Bubble Sort, Insertion Sort.

+ The slowest algorithms: Selection Sort.

1.2.3. Comparison bar chart



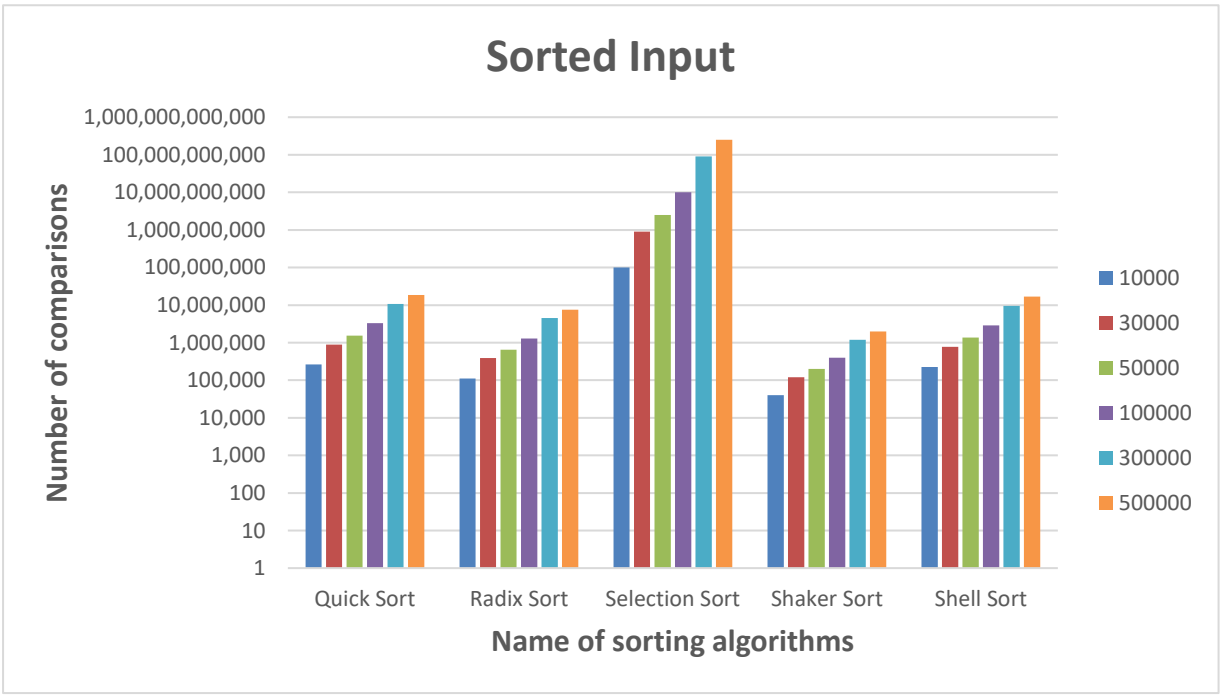


Figure 6: The number of comparisons of algorithms with sorted input

From the bar chart for counting comparisons of algorithms with sorted data input, we can see:

+ The least comparisons algorithms: Bubble Sort, Counting Sort, Insertion Sort, Shaker Sort.

+ The most comparisons algorithms: Selection Sort.

1.3. Reversed data.

1.3.1. Statistic table

Data order: REVERSE												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting Statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Bubble Sort	588.37	100,019,998	5516.61	900,059,998	15161.6	2,500,099,998	58480.6	10,000,199,998	537406	26,026,918,920	1498580	250,000,999,998
Counting Sort	0.3404	40,000	1.1556	120,000	1.7565	200,000	3.4279	400,000	6.3216	1,200,000	11.2048	2,000,000
Flash Sort	0.3163	81,018	1.2052	243,018	1.9901	405,018	2.9008	810,018	8.6959	2,430,018	20.2652	4,050,018
Heap Sort	1.9754	370,080	7.3794	1,253,673	11.7237	2,199,228	26.1695	4,690,131	81.5534	15,511,497	163.987	26,967,846
Insertion Sort	289.847	100,009,999	2712.29	900,029,999	7500.66	2,500,049,999	28214.9	10,000,099,999	266376	90,000,299,999	743420	250,000,499,999
Merge Sort	2.4905	456,443	5.8478	1,513,467	10.0129	2,633,947	16.9156	5,567,899	51.9998	18,108,315	105.747	31,336,411
Quick Sort	0.7639	460,819	3.7727	1,622,379	6.1699	2,885,671	16.6186	6,270,885	29.8542	21,176,461	51.997	37,174,571
Radix Sort	2.7971	110,097	7.1496	390,120	12.6686	650,120	22.1657	1,300,120	84.5399	4,500,143	144.549	7,500,143
Selection Sort	145.781	100,019,998	1316.46	900,059,998	3869.88	2,500,099,998	15108	10,000,199,998	217916	90,000,599,998	618605	250,000,999,998
Shaker Sort	598.225	100,010,001	5509.93	900,030,001	15460.6	2,500,050,001	57659.8	10,000,100,001	532843	90,000,300,001	1526810	250,000,500,001
Shell Sort	0.5491	324,408	1.9179	1,119,932	2.1554	1,871,455	8.6318	4,087,474	24.0237	13,417,878	26.9219	23,277,263

Figure 7: Measurement results of reversed data input

1.3.2. Running time line graph

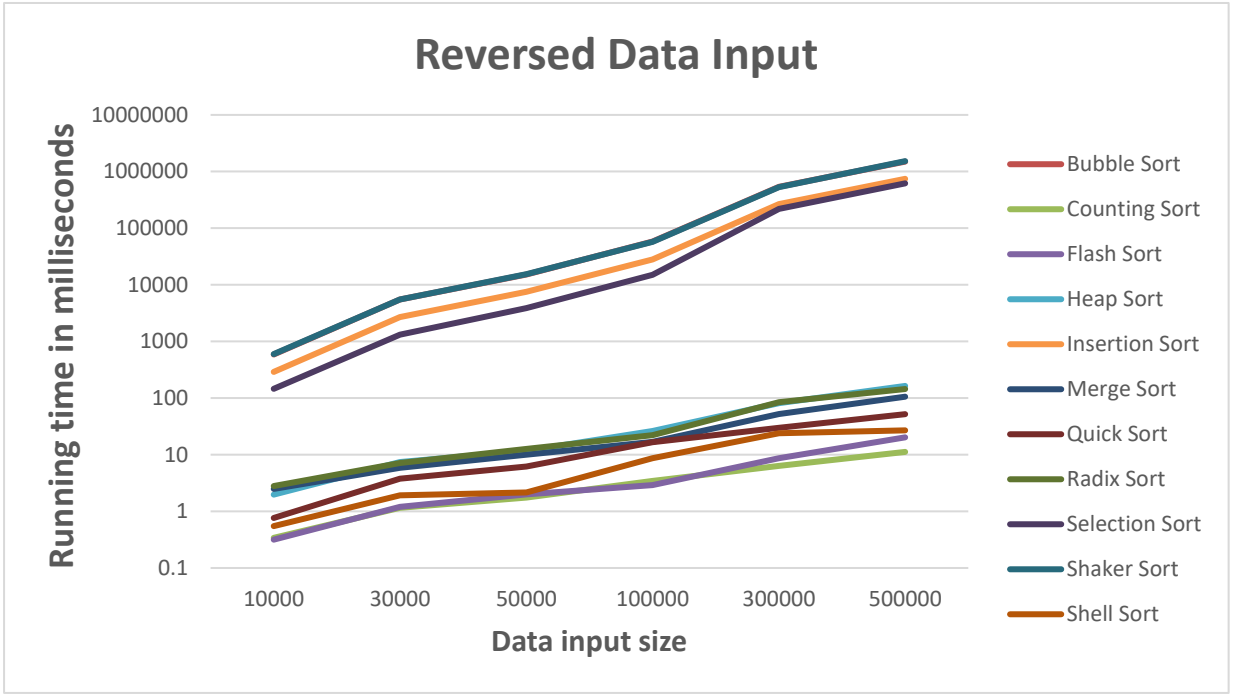


Figure 8: Running time line graph of algorithms with reversed data input

From the running time line graph of algorithms with reversed data input, we can see:

+ The fastest algorithms: Counting Sort, Flash Sort.

+ The slowest algorithms: Bubble Sort, Shaker Sort.

1.3.3. Comparison bar chart

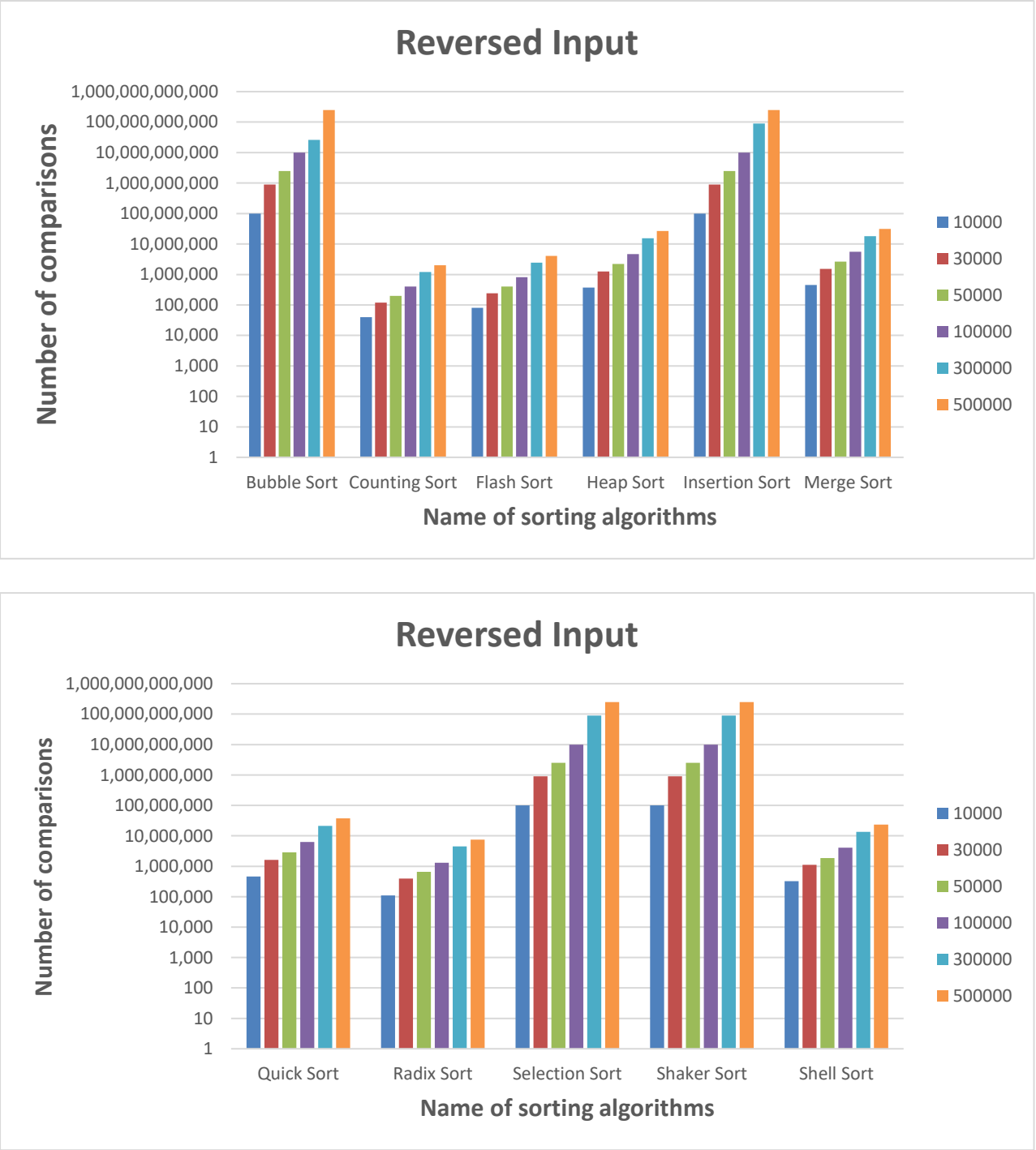


Figure 9: The number of comparisons of algorithms with reversed input

From the bar chart for counting comparisons of algorithms with reversed data input, we can see:

+ The least comparisons algorithms: Counting Sort.

+ The most comparisons algorithms: Bubble Sort, Insertion Sort, Selection Sort, Shaker Sort.

1.4. Nearly sorted data

1.4.1. Statistic table

Data order: NEARLY SORTED												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting Statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Bubble Sort	223.253	98,706,685	2162.17	796,183,137	4739.1	1,535,873,297	8831.21	3,624,496,897	40094	14,630,035,632	27860.6	26,570,836,145
Counting Sort	0.3585	40,000	0.6198	120,000	1.7519	200,000	3.4671	400,000	6.5416	1,200,000	17.8103	2,000,000
Flash Sort	0.3696	83,529	1.0401	250,532	1.8027	417,529	2.8537	835,023	11.9085	2,505,034	16.835	4,175,025
Heap Sort	2.2008	396,813	10.5699	1,329,591	15.0629	2,333,193	26.1841	4,969,320	89.6752	16,322,175	175.978	28,268,334
Insertion Sort	0.043	164,326	0.1957	306,838	0.3642	623,226	0.6868	743,862	2.0972	1,362,474	4.9542	1,690,026
Merge Sort	1.4287	424,995	5.1811	1,373,054	13.4871	2,367,677	17.5438	4,900,830	54.9375	15,650,920	89.0088	26,806,893
Quick Sort	0.7631	302,725	2.8053	1,054,227	5.6386	1,784,119	9.333	3,579,619	30.0721	11,069,303	64.4819	18,905,233
Radix Sort	2.6927	110,097	6.8063	390,120	10.7951	650,120	26.7965	1,300,120	92.461	4,500,143	148.53	7,500,143
Selection Sort	145.924	100,019,998	1324.77	900,059,998	3647.5	2,500,099,998	15469.2	10,000,199,998	142863	90,000,599,998	582223	250,000,999,998
Shaker Sort	1.6079	279,818	4.2555	959,760	6.0677	1,199,868	8.738	2,799,818	30.553	11,999,620	51.5832	17,999,694
Shell Sort	0.5187	279,806	1.9363	926,244	3.2791	1,543,380	5.1582	3,062,589	18.7247	9,816,956	35.1771	16,954,347

Figure 10: Measurement results of nearly sorted data input

1.4.2. Running time line graph

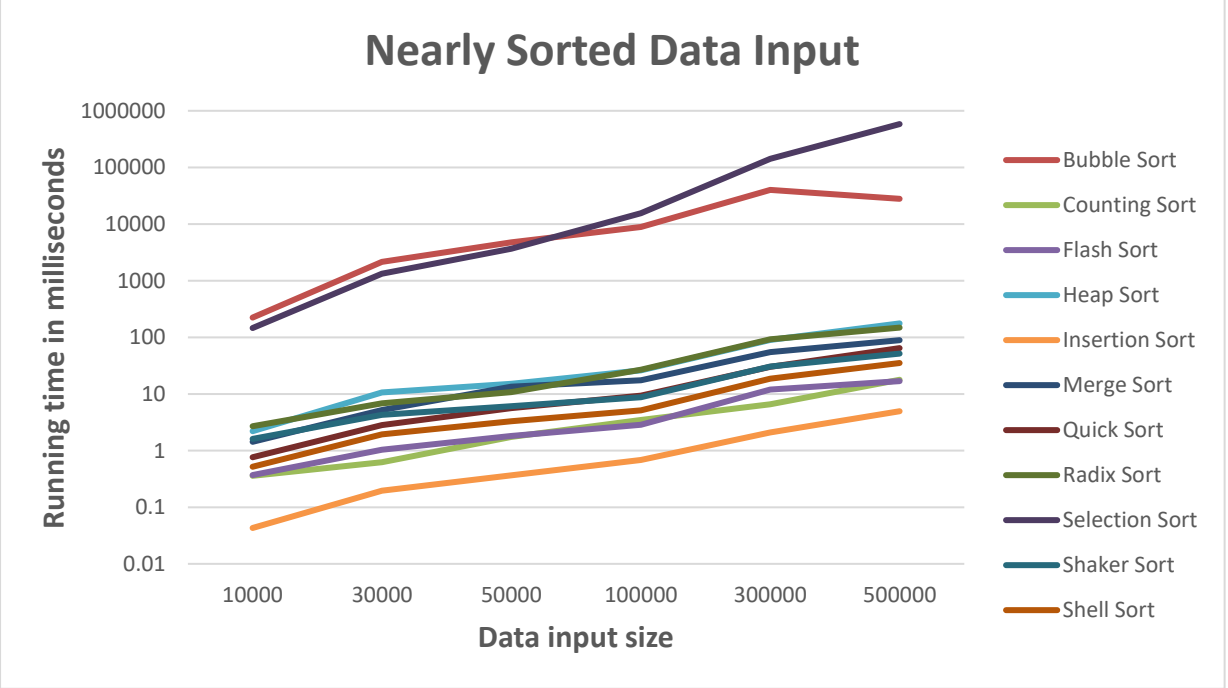


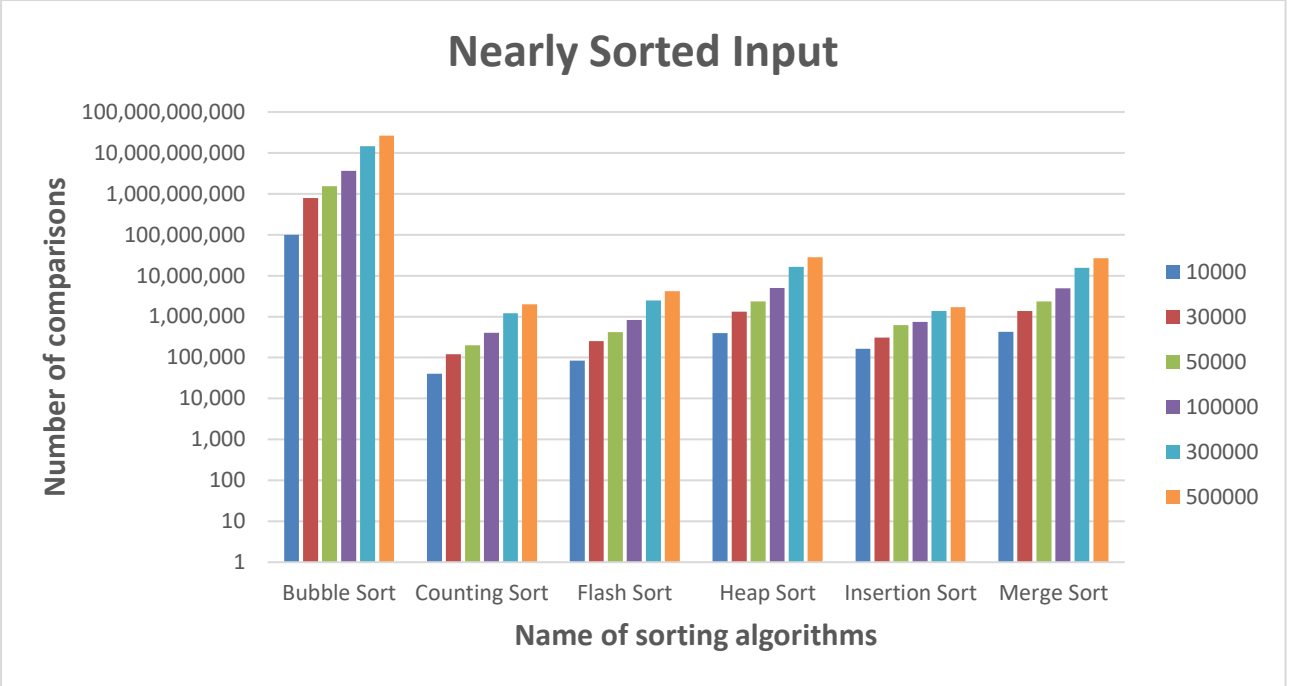
Figure 11: Running time line graph of algorithms with nearly sorted data input

From the running time line graph of algorithms with nearly sorted data input, we can see:

+ The fastest algorithm: Insertion Sort.

+ The slowest algorithms: Bubble Sort, Selection Sort.

1.4.3. Comparison bar chart



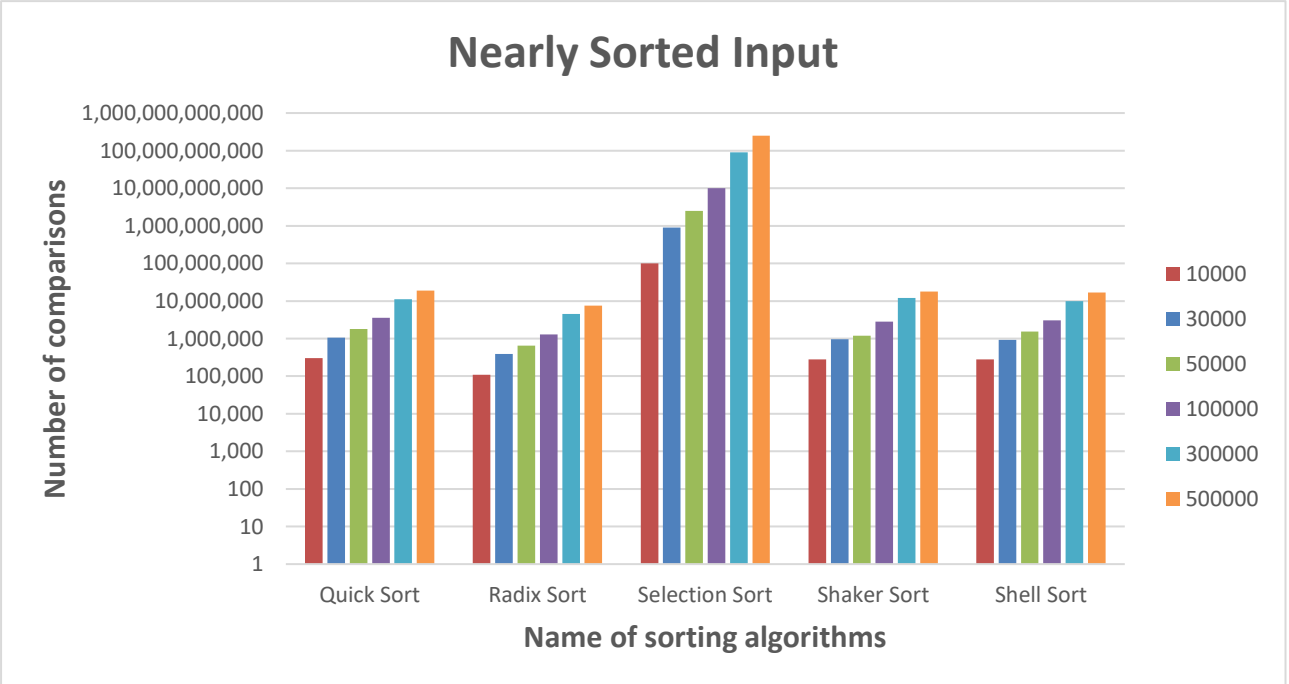


Figure 12: The number of comparisons of algorithms with nearly sorted input

From the bar chart for counting comparisons of algorithms with nearly sorted data input, we can see:

+ The least comparisons algorithms: Counting Sort, Insertion Sort.

+ The most comparisons algorithms: Bubble Sort, Selection Sort.

2. Experimental comments

2.1. The fastest algorithms: Couting Sort, Flash Sort, Radix Sort

Counting Sort is a non-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 comparisons.

As with all **Bucket Sorts**, performance of **Flash Sort** 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 $\mathbf{O(}N^{\mathbf{2}}\mathbf{)}$ algorithm like insertion sort has complexity $\mathbf{O(1^2) = O(1).}$ The running time of the final insertion sorts is therefore $\mathbf{M \cdot O(1) = O(M) = O(N).}$

From running time line graphs and counting comparisons through bar charts, we also witness the quickness of **Radix Sort**, in reality, the time complexity of this algorithm is **O(NK)**, with N is the number of elements in the list, K is the number of digits of the largest element in the list.

2.2. The slowest algorithms: Bubble Sort, Selection Sort, Insertion Sort, Shaker Sort

In original, if there are not any improvements on programming or main ideas, the time complexity of all 4 sorting algorithms is $\mathbf{O(N^2).}$

While **Bubble Sort** works by repeatedly swapping the adjacent element, **Selection Sort** sorts an array by repeatedly finding the minimum value in unsorted path and putting it in the right place, so if there are not any improvements, regard the number of comparisons, both these algorithms have the nearly similarity in value.

With sorted data input, it is evident that if we put a flag for checking whether there are any swappings in the process of moving through the array, **Bubble Sort** and **Insertion Sort** can obtain the order of **O(N)**, meanwhile, with nearly sorted data, **Shaker Sort** shows the efficiency in running. Though, in many cases, the number of comparisons when running 4 above algorithms with a large data is an excessive number, so we choose these algorithms as the slowest algorithms.

2.3. The stable algorithms

Stable algorithms sort the data without changing the order of the elements with the same values. So, based on the main idea of each algorithm, we can evaluate that:

Stable algorithms: Bubble Sort, Shaker Sort, Counting Sort, Merge Sort, Insertion Sort, Radix Sort.

Unstable algorithms: Selection Sort, Quick Sort, Heap Sort, Shell Sort, Flash Sort.

IV. Project organization and Programming notes

1. Project organization

- **Cpp_Files folder**: contain all .cpp files, .exe file, input and output file.
  - **main.cpp**: contains the *main()* function of the whole program
  - **DataGenerator.cpp**: contains all the data generator functions given by the instructors.
  - **[Author's name]_Sorts.cpp**: implementation of sorting functions written by author's names
  - **[Author's name]_Command.cp** : implementation of command functions written by author's name
  - **CmdHandling.cpp**: holds functions to extract information from the command line to a defined structure for further usage.
- **Header_Files**: contain all header files:
  - **DataGenerator.h**: stores the prototypes of functions that generate data.
  - **Command.h**: all command line arguments and their support prototype
  - **SortingAlgorithms.h**: names all the prototypes of the sorting algorithm within the program.
  - **Library.h**: includes all libraries and store structure.
  - **All.h**: contain all other header files, for ease of including throughout the program.

2. Programming notes

Libraries included in project:

- **iostream**: to use the cout function printing output to console terminal.
- **fstream**: to use functions to handle reading and writing to files.
- **string and cstring**: to use the string manipulation methods.
- **vector**: used to handle dynamic array more easily.
- **chrono**: used to calculate runtime of sorting algorithms.
- **cstdlib** and **ctime**: used to create random input.
- **queue**: using queue, pop and push functions to implement radix sort.

Structure: "Task"

- **int command**: the order of command following the requirement.
- **string mode**: "-a", "-c", getting the mode the program will run.
- **string al1, al2**: the mentioned sorting algorithms will be stored here.
- **int indexAlgo1, indexAlgo2**: the index of the function in the alphabetical order.

- **string inFile**: stores the file name.
- **string outPara**: controlling what to output: time, comparisons, or both.
- **bool useFile**: to check whether the command use file or not.
- **string inOrder**: the data will be generated with the following attribute.

Measuring time

- Each time the function is called, we put a checkpoint at the beginning of the function using *chrono::high_resolution_clock::now()*.
- Once the function completed its process, we mark the time again also by using the above function.
- After subtracting them to get the gap in between, casting it back to double type number *chrono::duration<double, milli>* and *count()* method.
- The output will be the running time measuring in milliseconds.

Function pointer

- This can be considered an extra understanding which is used to make our group's source code briefer and more flexible. In this project, we realized that after finding the name of sorting algorithms from handling command line argument, instead of seeking respective functions sequentially by using many conditional statements, we thought whether we could create an array for functions of counting comparisons and evaluating time. That's the reason why we went for **function pointer.**
- The difficulty when using **function pointer** is parameters of each function. Functions which are merged into a dynamic array must have the same parameters in both order and quantity.
- In our source code:
  - With each function for counting comparisons, we declare the same parameters which are **vector<int> &arr, long long &comparisons**.
  - With each function for judging time, we declare the same parameters which are **vector<int> &arr, double &time**.
  - In **library.h**, we create two structures **MeasureComp** and **MeasureTime**,
  - Finally, in **SortingAlgorithms.h**, we create two arrays **Algo_Measuring_Comp** and **Algo_Measuring_Time.**

V. List of references

Github (Reference from Võ Thanh Tú – a student of 21CLC02):

https://github.com/Noboroto/Lab-3-Sorting/tree/master

Bubble Sort:

https://www.geeksforgeeks.org/bubble-sort/

https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/

https://www.geeksforgeeks.org/odd-even-sort-brick-sort/

https://www.geeksforgeeks.org/cocktail-sort/

https://anisoftblog.wordpress.com/category/algorithms/sorting-algorithm/bubble-sort-and-variants/

https://en.wikipedia.org/wiki/Odd%E2%80%93even_sort

Counting Sort:

https://www.geeksforgeeks.org/counting-sort/

https://www.programiz.com/dsa/counting-sort

https://brilliant.org/wiki/counting-sort/

Flash Sort:

https://codelearn.io/sharing/dau-moi-la-thuat-toan-sap-xep-tot-nhat

https://en.wikipedia.org/wiki/Flashsort

https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh

https://steemit.com/popularscience/@krishtopa/the-fastest-sort-method-flashsort

https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

Heap Sort:

https://iq.opengenus.org/time-complexity-of-heap-sort/

https://www.geeksforgeeks.org/heap-sort/

https://www.programiz.com/dsa/heap-sort

Insertion Sort:

https://www.programiz.com/dsa/insertion-sort

https://www.geeksforgeeks.org/insertion-sort/

https://www.baeldung.com/cs/binary-insertion-sort

https://www.interviewkickstart.com/learn/binary-insertion-sort

Merge Sort:

https://www.geeksforgeeks.org/merge-sort/

https://www.geeksforgeeks.org/3-way-merge-sort/

https://en.wikipedia.org/wiki/Merge_sort

Quick Sort:

https://www.geeksforgeeks.org/quick-sort/

https://en.wikipedia.org/wiki/Quicksort

https://fptshop.com.vn/tin-tuc/danh-gia/quick-sort-la-gi-155097

https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-quick-sort/

Radix Sort:

https://www.programiz.com/dsa/radix-sort

https://www.geeksforgeeks.org/radix-sort/

https://www.topcoder.com/thrive/articles/Sorting

https://www.happycoders.eu/algorithms/radix-sort/

Selection Sort:

https://www.geeksforgeeks.org/selection-sort/

https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-selection-sort/

https://www.geeksforgeeks.org/recursive-selection-sort/?ref=lbp

https://en.wikipedia.org/wiki/Selection_sort

https://www.geeksforgeeks.org/bingo-sort-algorithm/

https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm

https://www.programiz.com/dsa/selection-sort/

Shaker Sort:

https://www.iostream.vn/article/bubble-sort-va-shaker-sort-01Si3U

https://en.wikipedia.org/wiki/Cocktail_shaker_sort

Shell Sort:

http://thomas.baudel.name/Visualisation/VisuTri/dobosort.html

https://en.wikipedia.org/wiki/Shellsort

https://www.geeksforgeeks.org/shellsort/

https://iq.opengenus.org/time-and-space-complexity-of-comb-sort/