UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

SUBJECT: CSC10004 – Data structures and Algorithms
REPORT:

SORTING ALGORITHMS

Class: 22CLC04

Group 4:

- Nguyễn Gia Phúc – 22127482
- Trần Thị Thiên Kim – 22127225
- Võ Trung Tín – 22127417
- Bùi Ngô Quang Minh – 22127261

Professors:

Bùi Huy Thông
Nguyễn Ngọc Thảo

Ho Chi Minh city, 28th July 2023

TABLE OF CONTENTS

I. Information

GROUP 4

Member name	Student ID
Nguyễn Gia Phúc	22127482
Trần Thị Thiên Kim	22127225
Võ Trung Tín	22127417
Bùi Ngô Quang Minh	22127261

II. Introduction

1. Initial speech

Sorting is a fundamental task we encounter in everyday life, from organizing cards in a game of Bridge to arranging bills or jars of spices. In computing, sorting is essential for efficient data retrieval and processing. Many sorting algorithms have been devised, ranging from intuitive approaches like Insertion Sort to complex ones like Quicksort, designed for handling large datasets. Studying sorting algorithms not only introduces a central problem in computer science but also helps us understand various algorithm design and analysis techniques. From divide and conquer strategies to exploring best and worst-case behaviors, sorting offers valuable insights into the efficiency and limitations of different algorithms. It remains an active field of research with ongoing efforts to develop specialized sorting methods for specific applications, like External Sorting for handling large disk-stored files.

2. Experimental computer's specifications

Machine name: BUIMINH
Machine Id: {8DFC9189-86E0-4880-B813-4B3AD4AA1976}
Operating System: Windows 11 Home Single Language 64-bit (10.0, Build 22621) (22621.ni_release.220506-1250)
Language: English (Regional Setting: English)
System Manufacturer: HP
System Model: HP ENVY Laptop 13-ba1xxx
BIOS: F.35 (type: UEFI)
Processor: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz (8 CPUs), ~2.8GHz
Memory: 8192MB RAM
Available OS Memory: 7938MB RAM
Page File: 6524MB used, 9351MB available
Windows Dir: C:\WINDOWS
DirectX Version: DirectX 12
DX Setup Parameters: Not found
User DPI Setting: 120 DPI (125 percent)
System DPI Setting: 144 DPI (150 percent)
DWM DPI Scaling: Disabled
Miracast: Available, with HDCP
Microsoft Graphics Hybrid: Not Supported

DirectX Database Version: 1.4.7
DxDiag Version: 10.00.22621.0001 64bit Unicode

III. Algorithm presentation

1. Selection sort

Idea

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

Description

The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

Complexity evaluation

Time complexity			Space complexity
Best case	Average case	Worst case	Worst case
$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$

Suitable dataset

Selection sort works well with small datasets.

Variants

Heap sort: Heap sort greatly improves the basic algorithm by using an implicit heap data structure to speed up finding and removing the lowest datum.

A bidirectional variant of selection sort (called **double selection sort** or sometimes **cocktail sort** due to its similarity to cocktail shaker sort) finds both the minimum and maximum values in the list in every pass.

In the **bingo sort** variant, items are sorted by repeatedly looking through the remaining items to find the greatest value and moving all items with that value to their final location. Like counting sort, this is an efficient variant if there are many duplicate values: selection sort does one pass through the remaining items for each item moved, while Bingo sort does one pass for each value. After an initial pass to find the greatest value, subsequent passes move every item with that value to its final location while finding the next value.

2. Insertion sort

Idea

Insertion sort is a simple sorting algorithm that works similarly to the way playing cards are sorted in hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Description

To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Complexity evaluation

Time complexity			Space complexity
Best case	Average case	Worst case	Worst case
$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$

Suitable dataset

Basically, insertion sort is efficient for small data values. This algorithm is adaptive in nature as it is appropriate for data sets that are already partially sorted.

Variants

The improved version is called **Shell sort**. The sorting algorithm compares elements separated by a distance that decreases on each pass. Shell sort has distinctly improved running times in practical work, with two simple variants requiring $\Theta\left(n^{\frac{3}{2}}\right)$ and $\Theta\left(n^{\frac{4}{3}}\right)$ running time.

Binary insertion sort employs a binary search to determine the correct location to insert new elements, and therefore performs ⌈log2 n⌉ comparisons in the worst case. When each element in the array is searched for and inserted this is $\Theta(n\log n)$.

3. Bubble sort

Idea

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

Description

The algorithm has a straightforward idea. The algorithm will traverse through the array (n -1) times. In each traversal, if two adjacent elements are found where the element before is greater than the element after, the algorithm will swap these two elements. It should be noted that after each traversal, the current largest element will be brought to the end of the array, and we may exclude this element from the next traversal, similar to the Selection Sort.

Complexity evaluation

Time complexity			Space complexity
Best case	Average case	Worst case	Worst case
$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$

We can observe that the number of comparisons required is $n-1$ in the first iteration, then $n-2, n-3, \ldots, 1$. The number of swaps in the worst case will be equal to the number of comparisons, whereas in the best case, it is 0 swaps.

In the worst case, there will be $O(n^2)$ comparisons and $O(n^2)$ swaps. Therefore, the complexity of the algorithm is $O(n^2)$ in the worst case.

In the best case, we need $O(n^2)$ comparisons and only $O(1)$ swaps. Hence, the complexity of the algorithm is $\Omega(n)$ in the best case.

The average-case complexity of the algorithm is $\Theta(n^2)$.

Bubble Sort has a memory complexity of $O(1)$ since it doesn't require additional memory beyond the original array.

Suitable dataset

Bubble sort has a time complexity of $\Theta(n^2)$ which makes it very slow for large data sets. So, it is used to sort small data sets.

In fact, bubble sort is almost utilized for academic purposes and rarely utilized practically.

Variants

The improved version is called **Shell sort**. The sorting algorithm compares elements separated by a distance that decreases on each pass. Shell sort has distinctly improved running times in practical work, with two simple variants requiring $\Theta\left(n^{\frac{3}{2}}\right)$ and $\Theta\left(n^{\frac{4}{3}}\right)$ running time.

Binary insertion sort employs a binary search to determine the correct location to insert new elements, and therefore performs ⌈log2 n⌉ comparisons in the worst case. When each element in the array is searched for and inserted this is $\Theta(n\log n)$.

4. Heap sort

Idea

First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until the size of heap is greater than 1.

Description

- Build a heap from the given input array.
- Repeat the following steps until the heap contains only one element:
  - Swap the root element of the heap (which is the largest element) with the last element of the heap.
  - Remove the last element of the heap (which is now in the correct position).
  - Heapify the remaining elements of the heap.
- The sorted array is obtained by reversing the order of the elements in the input array.

Complexity evaluation

Time complexity			Space complexity
Best case	Average case	Worst case	Worst case
$\Omega(n\log n)$	$\Theta(n\log n)$	$O(n\log n)$	$O(1)$

Suitable dataset

The algorithm is suitable for large datasets.

When requiring the maximum or minimum value of a dataset in the fastest time, heap sort is usually used since it doesn't need the dataset to be sorted.

Variants

Floyd's heap construction is the most important variation to the basic algorithm, which is included in all practical implementations, is a heap-construction algorithm by Floyd which runs in $O(\boldsymbol{n})$time and uses sift-down rather than sift-up, avoiding the need to implement sift-up at all.

Ternary heapsort uses a ternary heap instead of a binary heap; that is, each element in the heap has three children. It is more complicated to program but does a constant number of times fewer swap and comparison operations. This is because each sift-down step in a ternary heap requires three comparisons and one swap, whereas in a binary heap two comparisons and one swap are required.

5. Merge sort

Idea

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

Description

Merge Sort is a divide-and-conquer algorithm, and it operates recursively as follows:

- If the array has fewer than 2 elements, meaning it is already sorted or contains only one element, then we do nothing because this array is considered sorted.
- Otherwise, we divide the array into two parts, each containing nearly an equal number of elements (i.e., the difference in the number of elements between the two parts is at most one).
- Next, we recursively apply the Merge Sort algorithm to each of these two subarrays. This recursive process continues until each subarray contains only one element (this is the base case of the recursion).
- After both subarrays are sorted, we merge them back together to create a fully sorted array.

The merging step is a crucial part of the Merge Sort algorithm. We compare the elements of the two subarrays and place them in the resulting array in the correct sorted order. This ensures that the final array is sorted correctly.

Complexity evaluation

Time complexity			**Space complexity**
Best case	Average case	Worst case	Worst case
$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$

The time complexity of the Merge Sort algorithm is indeed $\Theta(n \log n)$ in all cases. We can prove it as follows:

Assume $T(n)$ is the time taken for Merge Sort to sort an array of n elements.

We have the following recursive equation: $T(n) = 2 \times T\left(\frac{n}{2}\right) + n$. This is because we need to solve subproblems for two subarrays of size $\frac{n}{2}$, and merging the subarrays takes n operations.

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

$$= 2^{\log n} * T\left(\frac{n}{2^{\log n}}\right) + n \log n$$

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

Merge Sort's time complexity is $O(n \log n)$ because it divides the problem into smaller subproblems and then merges the sorted subarrays.

Regarding memory complexity, Merge sort requires $O(n)$ memory, where n is the length of the array, as it uses additional memory to store temporary arrays during the merge operation. The maximum depth of the recursive call stack is $\log n$, which also contributes to memory complexity. However, this is still considered reasonable, and Merge Sort is known for its stable and efficient performance for large datasets.

Suitable dataset

Sorting large datasets: Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of $O(\boldsymbol{n}\log\boldsymbol{n})$.

External sorting: Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.

Custom sorting: Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.

Variants

Instead of merging two blocks at a time, a **ping-pong merge** merges four blocks at a time. The four sorted blocks are merged simultaneously to auxiliary space into two sorted blocks, then the two sorted blocks are merged back to main memory. Doing so omits the copy operation and reduces the total number of moves by half.

6. Quick sort

Idea

The key process in quick sort is a partition.

Description

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.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

Complexity evaluation

Time complexity			Space complexity
Best case	Average case	Worst case	Worst case
$\Omega(n\log n)$	$\Theta(n\log n)$	$O(n^2)$	$O(n)$

Suitable dataset

Quick sort is efficiently used to sort large datasets. It is not a good choice for small datasets.

Variants

Quicksort is a space-optimized version of the **binary tree sort**. Instead of inserting items sequentially into an explicit tree, quicksort organizes them concurrently into a tree that is implied by the recursive calls.

Intro sort is a variant of quicksort that switches to heapsort when a bad case is detected to avoid quicksort's worst-case running time. Major programming languages, such as C++ (in the GNU and LLVM implementations), use intro sort.

Bucket sort with two buckets is very similar to quicksort; the pivot in this case is effectively the value in the middle of the value range, which does well on average for uniformly distributed inputs.

7. Radix sort

Idea

Radix sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

Description

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, radix sort achieves the final sorted order.

The key idea behind radix sort is to exploit the concept of place value. It assumes that sorting numbers digit by digit will eventually result in a fully sorted list. Radix sort can be performed using different variations, such as Least Significant Digit (LSD) Radix Sort or Most Significant Digit (MSD) Radix Sort.

Complexity evaluation

Time complexity			Space complexity
Best case	Average case	Worst case	Worst case
$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$

k is the number of digits of the maximum value in the dataset.

Suitable dataset

In practical implementations, radix sort is often faster than other comparison-based sorting algorithms, such as quicksort or merge sort, for large datasets, especially when the keys have many digits. However, its time complexity grows linearly with the number of digits, and so it is not as efficient for small datasets.

Variants

Binary MSD radix sort, also called binary quicksort, can be implemented in-place by splitting the input array into two bins - the 0s bin and the 1s bin.

MSD radix sort can be implemented as a stable algorithm but requires the use of a memory buffer of the same size as the input array.

Radix sort, such as the two-pass method where counting sort is used during the first pass of each level of recursion, has a large constant overhead. Thus, when the bins get small, other sorting algorithms should be used, such as insertion sort. A good implementation of insertion sort is fast for small arrays, stable, in-place, and can significantly speed up radix sort.

Radix sorting can also be accomplished by building a tree (or radix tree) from the input set, and doing a pre-order traversal.

8. Shaker sort

Idea

Shaker Sort is an improvement over Bubble Sort. In the Bubble Sort algorithm, each sorting pass only moves elements toward the end of the array. However, in Shaker Sort, after traversing the array from the beginning to the end, we perform an additional traversal from the end to the beginning. This way, elements can be moved in both directions, towards the front and the back of the array.

Description

In the Bubble Sort algorithm, each sorting pass only moves elements toward the end of the array. However, in Shaker Sort, after traversing the array from the beginning to the end, we perform an additional traversal from the end to the beginning. This way, elements can be moved in both directions, towards the front and the back of the array.

After each iteration, the largest element will be moved to the end, and the smallest element will be moved to the front. Consequently, we don't need to traverse the entire array from the beginning in each pass, similar to the optimization described earlier for the Selection Sort algorithm.

Complexity evaluation

Time complexity			Space complexity
Best case	Average case	Worst case	Worst case
$\Omega(n^2)$	$\Theta(n)$	$O(n^2)$	$O(1)$

The complexity of the Shaker Sort algorithm is indeed the same as Bubble Sort, which is $O(n^2)$ in all cases. The main difference between Shaker Sort and Bubble Sort lies in the number of steps performed during execution, leading to potentially faster running times for Shaker Sort due to its bidirectional passes.

However, when it comes to complexity analysis, they both have a quadratic time complexity of $O(n^2)$.

Regarding memory complexity, Shaker Sort, like Bubble Sort, has a memory complexity of $O(1)$ since it only requires a constant amount of additional memory to store temporary variables during the sorting process.

Suitable dataset

Shaker sort has a worst-case time complexity of $O(n^2)$, which means that it can be slow for large datasets or datasets that are already partially sorted.

Just like Bubble sort, this algorithm is primarily used for educational purposes.

Variants

Shaker sort is itself variant of Bubble sort.

Shaker sort has different names: Bidirectional Bubble Sort, Cocktail Sort, Cocktail Shaker Sort, Ripple Sort, Shuffle Sort or Shuttle Sort.

9. Shell sort

Idea

Shell Sort can be considered an improvement over the Insertion Sort algorithm.

Description

This algorithm allows comparing and swapping elements that are far apart, rather than just adjacent elements as in the Insertion Sort. Initially, we have a sequence of numbers called "gaps." We traverse through these gaps in decreasing order, and during each pass, we only compare elements that are separated by the corresponding gap distance.

Complexity evaluation

Time complexity			Space complexity
Best case	Average case	Worst case	Worst case
$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(n)$

The time complexity of the Shell Sort algorithm depends largely on how we choose the sequence of gaps. For some gap sequences, evaluating the complexity of Shell Sort remains an open problem with no definitive solution.

Different gap sequences can lead to different running times for Shell Sort, and there is ongoing research to find optimal gap sequences for certain types of data.

Regarding memory complexity, Shell Sort requires $O(1 + m)$ memory, where m is the length of the gap sequence. The additional memory required is for storing temporary variables or the gap sequence itself. In most practical cases, the memory overhead of Shell Sort is considered reasonable.

Suitable dataset

Shell sort is commonly used for from medium-sized to large-sized datasets.

Variants

Shell sort's variants were born by inventing different gap sequences which allow us to effectively sort datasets. Some popular gap sequences are:

- **Shell's original gap sequence**: gap = gap / 2, where the initial gap is the length of the array. This sequence has been proven to be not optimal but serves as the basis for other gap sequences.
- **Hibbard's gap sequence**: gap = 2^k - 1, where k starts at floor(log2(n)) and decrements by 1 at each step.
- **Sedgewick's gap sequence**: The gap sequence is calculated using a combination of powers of 2 and powers of 3: 1, 5, 19, 41, 109, ...
- **Knuth's gap sequence**: gap = (3^k - 1) / 2, where k starts at floor(log3(n)) and decrements by 1 at each step.

10. Counting sort

Idea

Counting sort is a sorting technique based on keys between a specific range.

Description

It operates by counting the number of objects that possess distinct key values and applying prefix sum on those counts to determine the positions of each key value in the output sequence. Thus, each element of the unsorted sequence will be put on a supplement sequence of the same size by using the positions calculated in the previous step. Finally, copy all elements of the supplement sequence to the initial one.

Complexity evaluation

Time complexity			Space complexity
Best case	Average case	Worst case	Worst case
$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(n+k)$

k is the range of input

Suitable dataset

Counting sort makes assumptions about the data, for example, it assumes that values are going to be in the range of 0 to 10 or 10 – 99, etc. Some other assumption counting sort makes is input data will be positive integers.

Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between the range 1 to 10K and the data is 10, 5, 10K, 5K.

It is often used as a sub-routine to another sorting algorithm like the radix sort.

The counting sort can be extended to work for negative inputs also.

Variants

Some variants of Counting sort are formed by adjusting the range of key or changing the counting process.

11. Flash sort

Idea

Flash Sort is an algorithm that relies on both data distribution and comparisons. The idea of the algorithm is to divide the elements into m different partitions. Element $a[i]$ will belong to the partition number $1 + \lfloor (m - 1) \times ((a[i] - a_{min})/(a_{max} - a_{min})) \rfloor$.

Description

- Find the minimum and maximum elements in the input array to determine the range of values.
- Calculate the number of partitions "m" based on a chosen fraction of the array size or a constant.
- Create an empty output array "b" and an auxiliary array "L" of size "m" to store the distribution counts of elements in each partition.
- Distribute elements into their respective partitions based on the calculated partition number "k" for each element.
- Calculate the starting positions of each partition in the output array "b" using partial sum computation.
- After partitioning, the algorithm proceeds to sort the elements within each partition using the Insertion Sort algorithm.

Complexity evaluation

The partitioning phase has a time complexity of $O(n)$ because each element is traversed only once.

During the sorting phase, on average, each partition will have n/m elements, and using the Insertion Sort algorithm, it will take $O\left(\frac{n^2}{m^2}\right)$ time to sort each partition. Since there are m partitions, the overall time complexity to sort m partitions is $O(nm^2)$

Based on empirical evidence, it is recommended to choose $m = 0.43n$. Substituting $m = 0.43n$, the time complexity of the sorting phase becomes $O(n^2 \times 0.43n) = O(n)$.

However, this analysis is based on average-case performance. In the worst-case scenario, when the data distribution is not even, the time complexity of Flash Sort can be $O(n^2)$, although this is rare when dealing with completely random data.

The memory complexity is $O(m) = O(n)$. However, due to specific characteristics of the data, the author chose $m = n$ to ensure that each partition contains the same elements (as all elements $a[i]$ are less than n), resulting in the algorithm mostly incurring the partitioning phase's time cost. This choice of $m = n$ is made possible within the limitations of the specific dataset.

It is important to note that the choice of $m = n$ is based on the dataset's constraints, and it helps in achieving better performance under those specific conditions.

Suitable dataset

Flash sort is appropriately used for uniformly distributed datasets.

IV. Experimental results and comments

1. Experimental results

Randomized datasets

Data order: Randomized												
Data size	**10000**		**30000**		**50000**		**100000**		**300000**		**500000**	
Resulting statics	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons
Selection sort	53	100009999	367	900029999	993	2500049999	4646	10000099999	43393	90000299999	113669	250000499999
Insertion sort	37	50042652	321	452418676	862	1252916442	4057	5001650759	36271	44974693564	96022	124873746030
Bubble sort	167	100009999	1822	900029999	6734	2500049999	26377	10000099999	242245	90000299999	647658	250000499999
Heap sort	1	496866	3	1680769	7	2951623	19	6304859	49	20796155	72	36120059
Merge sort	6	583504	25	1937307	54	3383489	97	7166098	275	23382892	500	40382427
Quick sort	1	284470	2	974190	5	1606081	10	3513390	31	11165698	73	20478008
Radix sort	1	140061	2	510076	4	850076	7	1700076	23	5100076	40	8500076
Shaker sort	161	66738687	1491	603044421	4958	1669252922	20508	6665693931	179029	59977677857	490594	166472278641
Shell sort	1	639419	4	2265635	7	4467281	16	9840271	72	34016964	108	61673823
Counting sort	0	40003	0	120003	1	200003	1	400003	3	1200003	9	2000003
Flash sort	1	94113	1	280281	3	462614	2	932029	8	2742613	20	4245019

Nearly sorted datasets

Data order: Nearly sorted												
Data size	**10000**		**30000**		**50000**		**100000**		**300000**		**500000**	
Resulting statics	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons
Selection sort	42	100009999	405	900029999	1127	2500049999	4407	10000099999	40131	90000299999	112519	250000499999
Insertion sort	0	169222	1	655062	0	580446	1	868462	1	1215070	2	1934858
Bubble sort	55	100009999	513	900029999	1417	2500049999	5782	10000099999	53503	90000299999	149323	250000499999
Heap sort	1	518588	2	1739614	4	3056454	7	6519719	25	21431722	43	37116337
Merge sort	6	505515	27	1645227	47	2817482	96	5829758	252	18745271	445	32124472
Quick sort	1	193666	1	627258	1	1084502	2	2268975	7	7275742	12	12475776
Radix sort	1	140061	3	510076	4	850076	7	1700076	29	6000091	49	10000091
Shaker sort	1	183619	1	674349	1	632083	2	944947	1	1505918	2	2465607
Shell sort	1	398885	2	1336184	2	2281984	4	4690883	11	15427455	18	25658592
Counting sort	0	40003	0	120003	0	200003	1	400003	3	1200003	6	2000003
Flash sort	0	123458	1	370458	1	617464	2	1234962	6	3704960	19	6174964

Sorted datasets

Data order: Sorted												
Data size	**10000**		**30000**		**50000**		**100000**		**300000**		**500000**	
Resulting statics	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons
Selection sort	41	100009999	402	900029999	1100	2500049999	4411	10000099999	40038	90000299999	112747	250000499999
Insertion sort	0	29998	0	89998	0	149998	0	299998	1	899998	2	1499998
Bubble sort	63	100009999	550	900029999	1481	2500049999	5895	10000099999	52405	90000299999	147990	250000499999
Heap sort	1	518705	3	1739633	7	3056481	20	6519813	25	21431637	55	37116275
Merge sort	6	475242	37	1559914	47	2722826	106	5745658	256	18645946	437	32017850
Quick sort	1	193610	1	627226	1	1084458	2	2268922	8	7275706	12	12475706
Radix sort	1	140061	3	510076	4	850076	7	1700076	38	6000091	49	10000091
Shaker sort	0	20001	0	60001	0	100001	0	200001	1	600001	1	1000001
Shell sort	1	360042	1	1170050	2	2100049	4	4500051	11	15300061	23	25500058
Counting sort	0	40003	1	120003	1	200003	0	400003	3	1200003	5	2000003
Flash sort	0	123490	1	370490	1	617490	2	1234990	6	3704990	10	6174990

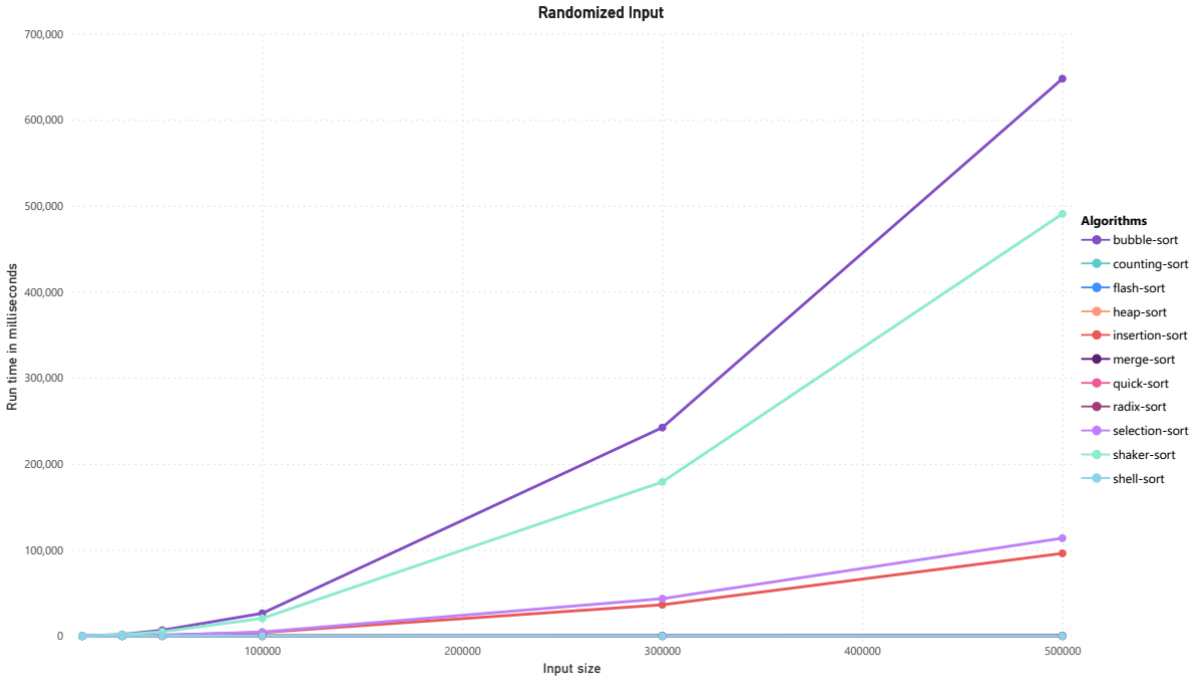
Reversed datasets

Data order: Reversed												
Data size	**10000**		**30000**		**50000**		**100000**		**300000**		**500000**	
Resulting statics	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons	Running time	Number of comparisons
Selection sort	54	100009999	520	900029999	1436	2500049999	5635	10000099999	51780	90000299999	145858	250000499999
Insertion sort	76	100009999	690	900029999	1924	2500049999	7721	10000099999	68588	90000299999	193457	250000499999
Bubble sort	197	100009999	1858	900029999	5190	2500049999	20602	10000099999	187480	90000299999	522978	250000499999
Heap sort	1	476739	2	1622791	4	2848016	8	6087452	41	20187386	49	35135730
Merge sort	6	476441	27	1573465	52	2733945	113	5767897	266	18708313	458	32336409
Quick sort	0	203607	1	657223	3	1134455	3	2368919	7	7575703	13	12975703
Radix sort	0	140061	5	510076	4	850076	7	1700076	31	6000091	47	10000091
Shaker sort	219	100005000	2087	900015000	5556	2500025000	22223	10000050000	199733	90000150000	558940	250000250000
Shell sort	0	475175	2	1554051	3	2844628	5	6089190	17	20001852	29	33857581
Counting sort	0	40003	0	120003	1	200003	1	400003	3	1200003	4	2000003
Flash sort	0	105999	1	317999	1	529999	2	1059999	6	3179999	10	5299999

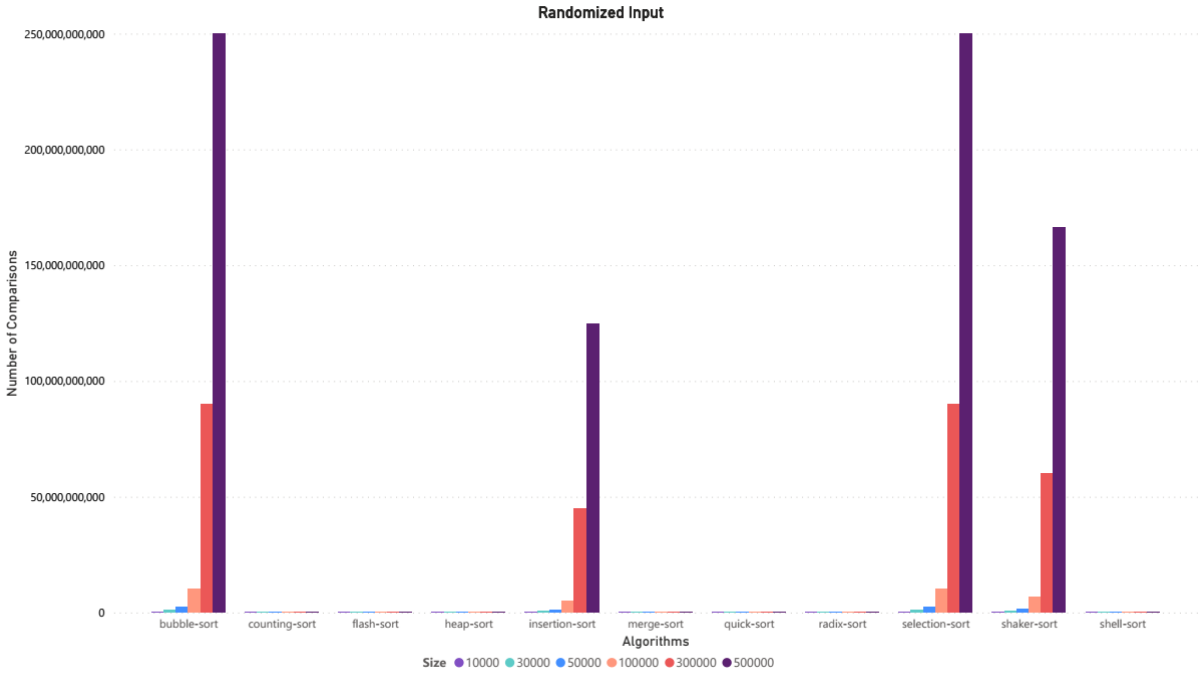
2. Result visualization

Data arrangement: Randomized

The line graph below demonstrates running times of 11 algorithms sorting 6 randomized datasets.

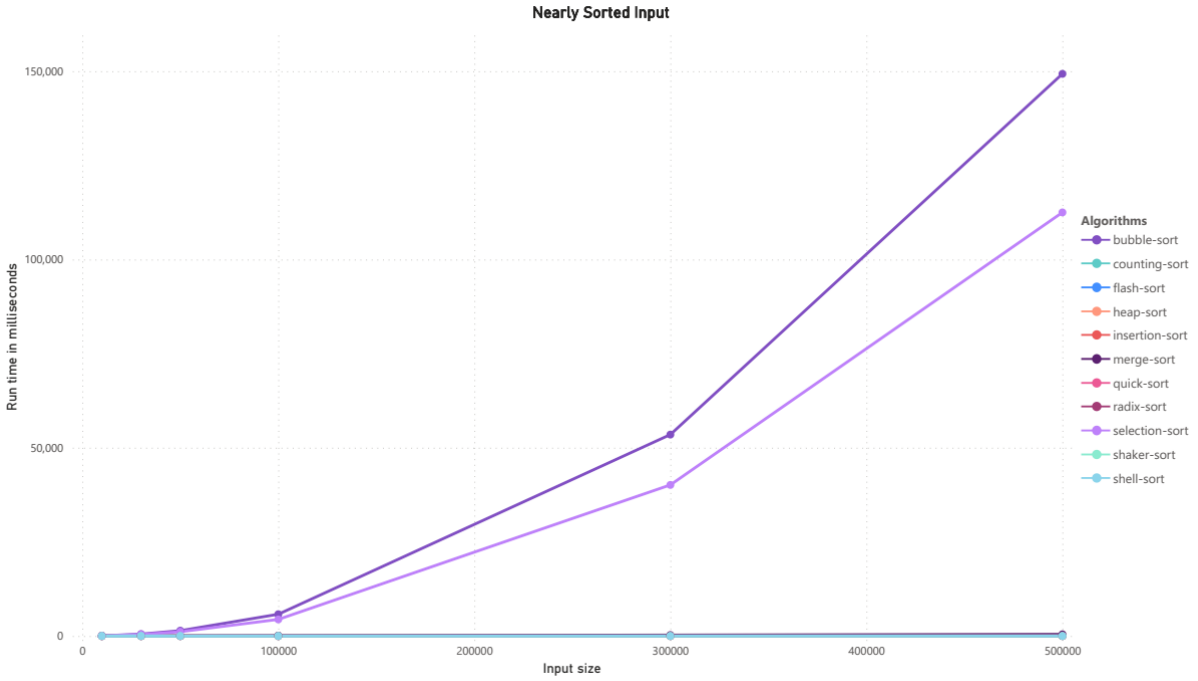


The bar chart below demonstrates numbers of comparisons of 11 algorithms sorting 6 randomized datasets.

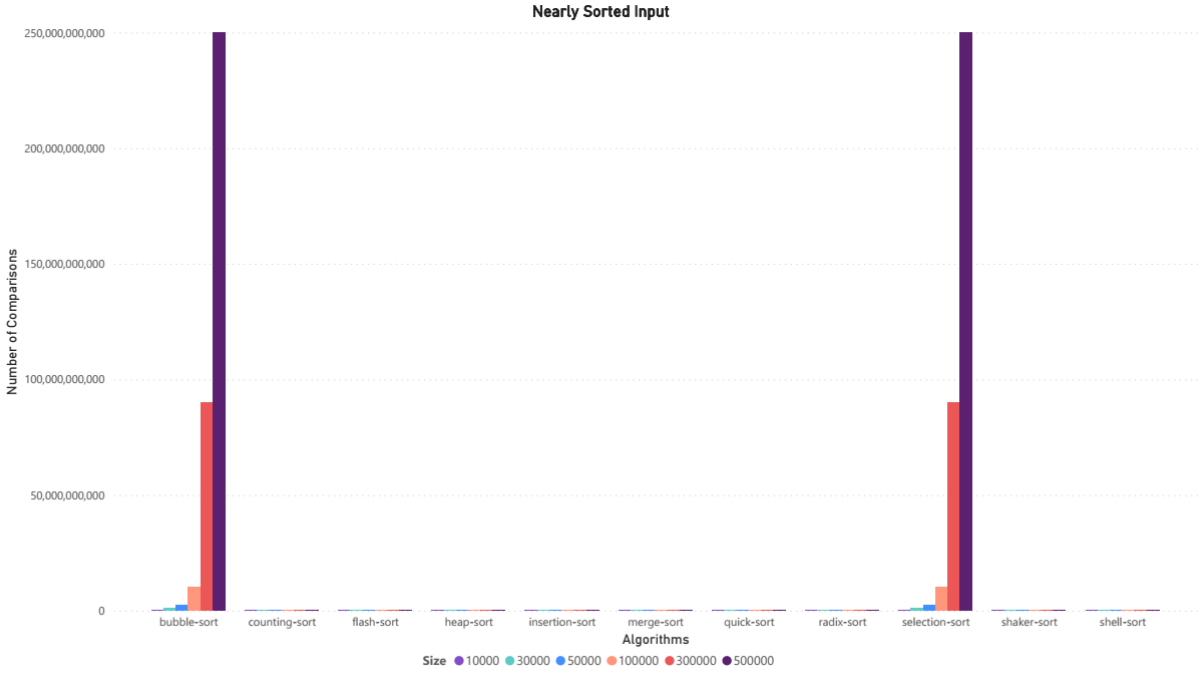


Data arrangement: Nearly sorted

The line graph below demonstrates running times of 11 algorithms sorting 6 nearly sorted datasets.

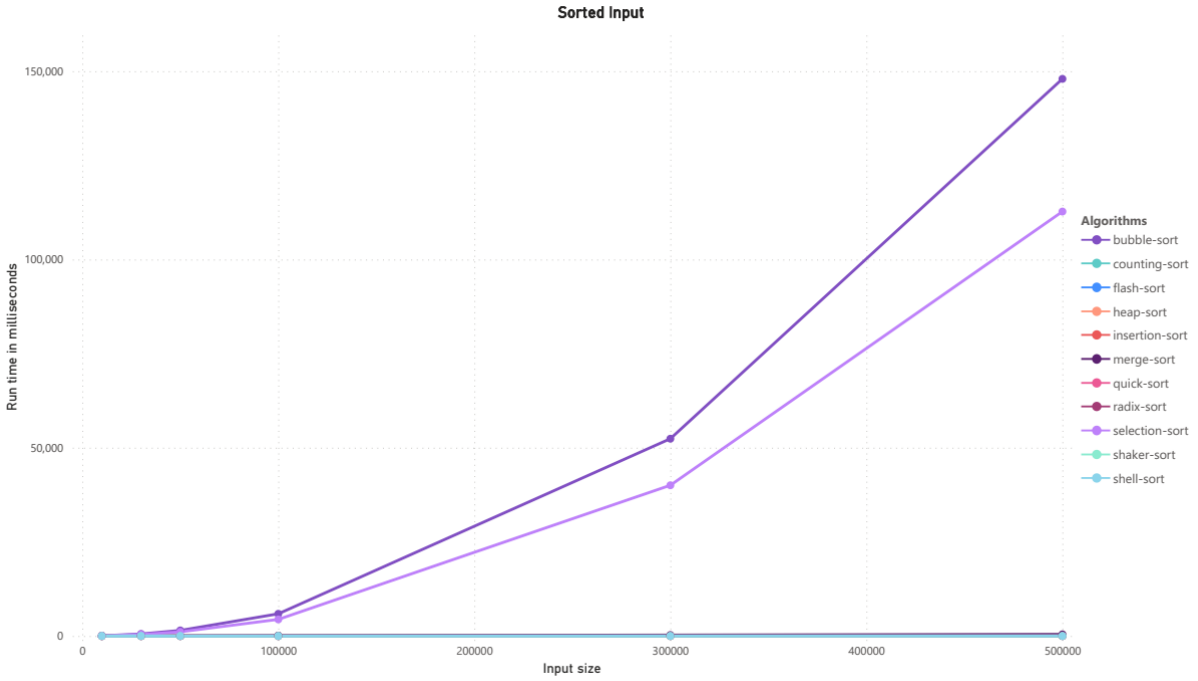


The bar chart below demonstrates numbers of comparisons of 11 algorithms sorting 6 nearly sorted datasets.



Data arrangement: Sorted

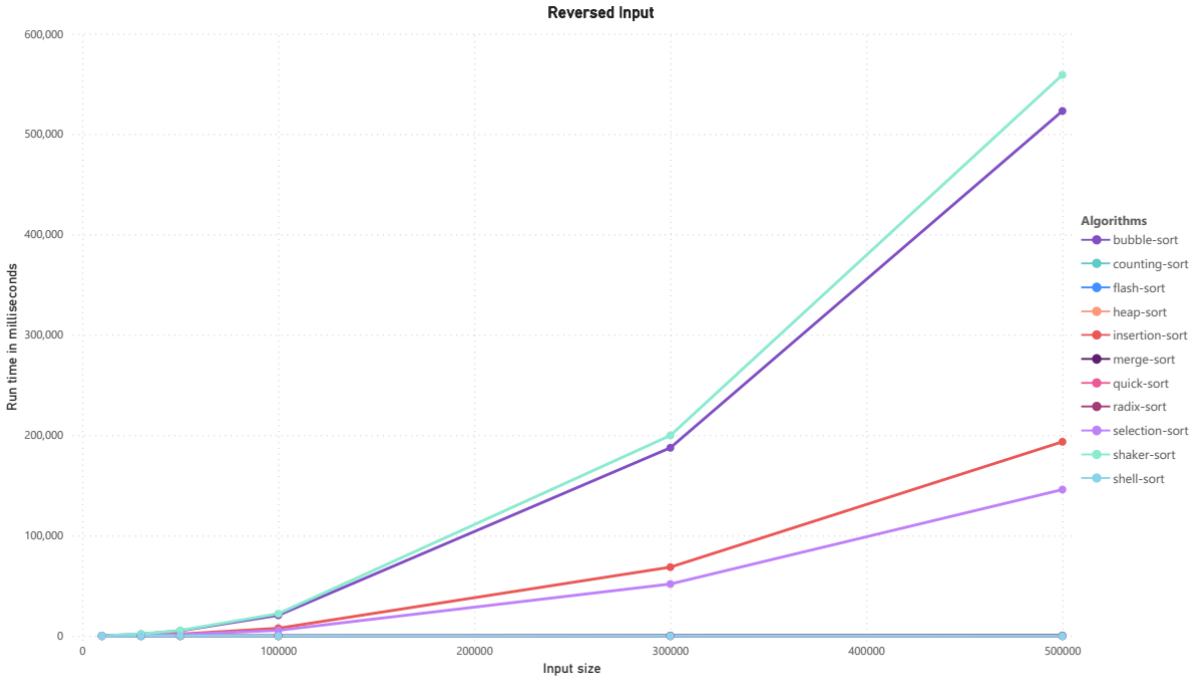
The line graph below demonstrates running times of 11 algorithms sorting 6 sorted datasets.



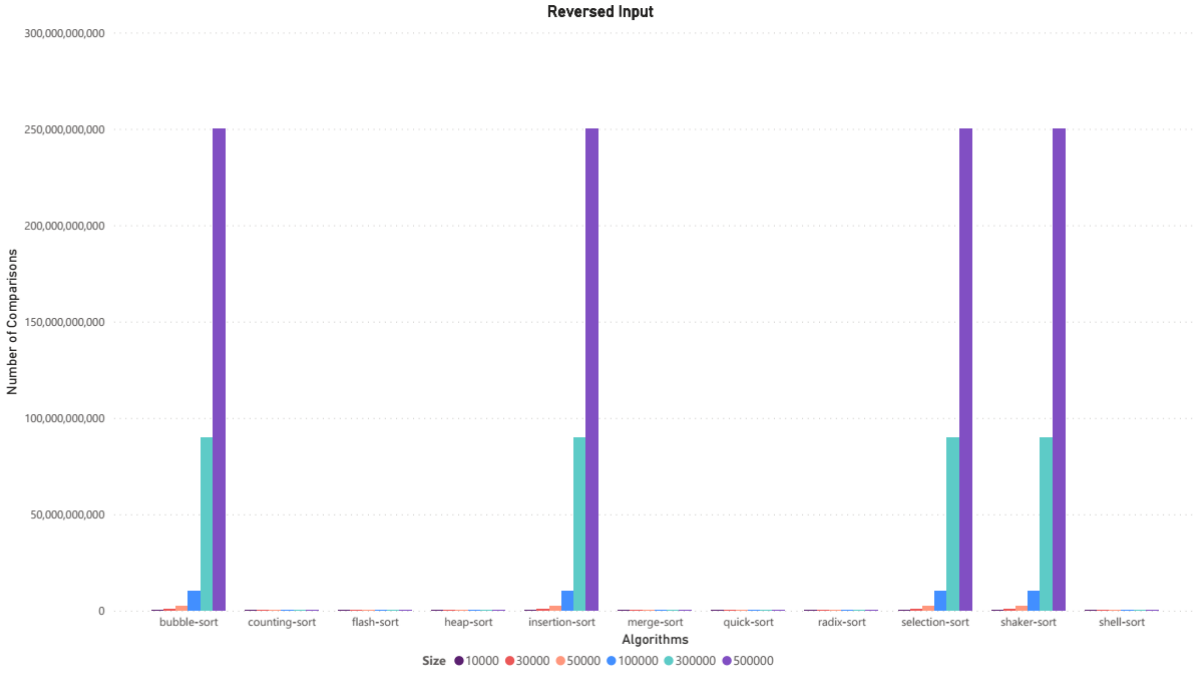
The bar chart below demonstrates numbers of comparisons of 11 algorithms sorting 6 sorted datasets.

Data arrangement: Reversed

The line graph below demonstrates running times of 11 algorithms sorting 6 reversed datasets.



The bar chart below demonstrates numbers of comparisons of 11 algorithms sorting 6 reversed datasets.



3. Comments on graphs

Observing the line graphs above demonstrating running times of 11 sorting algorithms, it is obvious for us to find out which algorithm works the fastest or the slowest with all 4 types of datasets.

Concerning the fastest algorithms which belong to time complexity $\Theta(n \log n)$ and $\Theta(n)$, namely Counting sort, Flash sort, Heap sort, Merge sort, Quick sort, Radix sort, and Shell sort, lines of these algorithms are virtually straight with all data sizes. It means that together with the upturn in data sizes, they still work at a stabilized amount of time. So, these are certainly the fastest among 11 sorting algorithms.

Concerning the slowest algorithms which belong to time complexity and $\Theta(n^2)$, namely Bubble sort, Insertion sort, Selection sort, and Shaker sort, their amount of running time increases in a quadratic line. It means the greater the data size is, the hugely increasing amount of time they will need to process. So, these are certainly the slowest among 11 sorting algorithms.

Regarding the time acceleration of 11 algorithms, Bubble sort, Insertion sort, Selection sort, and Shaker sort have running time increasing quadratically with sizes of datasets, which means they work dramatically slower when data sizes go bigger, whilst Counting sort, Flash sort, Heap sort, Merge sort, Quick sort, Radix sort, and Shell sort have running time virtually stabilizing.

As can be seen from the bar charts above illustrating the number of comparisons made by 11 sorting algorithms, we can find out which algorithms make the most comparisons or the least comparisons with all 4 types of datasets.

Talking about the algorithms making the most significant quantity of comparisons, it is clear to see that Bubble sort and Selection sort execute the most comparisons with whichever size of datasets or type of datasets. With randomized datasets and reversed datasets, Insertion sort and Shaker sort also make the considerable volume of comparing operations. To explain for these statistics, we know that whichever order the dataset has, Bubble sort and Selection sort always make the fixed number of comparisons whereas Bubble sort and Selection sort depend on data distribution in a dataset.

Talking about the algorithms making the least substantial volume of comparisons, we can obviously see that Counting sort, Flash sort and Radix sort make the least comparing operations with all types of datasets and size of datasets. The reason for that would be that these three algorithms work basing on distributing manner. What is worth noticing is that with sorted and nearly sorted input data, Insertion sort and Shaker sort have the marginal number of comparisons for these two depend on data distribution in datasets.

Regarding the acceleration of the number of comparisons made by 11 algorithms, we can simply see that the rates of Bubble sort and Selection sort escalate substantially with all types of data and data sizes increasing due to the fact that they belong to algorithm group $\Theta(n^2)$. It is the same for Insertion sort and Shaker sort with randomized input and reversed input. Meanwhile, Counting sort, Flash sort and Radix sort have the level of comparisons expanding gradually with data sizes.

4. Overall comments

Having found out the group of algorithms which work the fastest and the slowest above, we choose the overall fastest and the overall slowest algorithms.

Since the lines of the group of fastest algorithms are straight, it is difficult to just pictorially think. Yet according to time complexity calculated in theory, Counting Sort, Flash sort and Radix sort is the fastest.

In contrary, we obviously know that Bubble sort and Selection sort execute the slowest as lines of these two algorithms are always quadratic.

Sorting algorithm	**Stable**	**Unstable**
Selection sort		X
Insertion sort	X	
Bubble sort	X	
Heap sort		X
Merge sort	X	
Quick sort		X
Radix sort	X	
Shaker sort	X	
Shell sort		X
Counting sort	X	
Flash sort		X

V. Project organization and programming notes

1. Project organization

Our source code of this project is organized within 5 files described as below:

- **DataGenerator.cpp**: this C++ file given by professors contains relating functions used for data generating according to 4 types of datasets.
- **Header.h**: this header file is used for declaring all related libraries, structs, and functions defined inside other files.
- **sorting_algorithms.cpp**: this C++ file is used to define, in other words implement, sorting algorithms' function and related utility functions.
- **command.cpp**: this C++ file is used to implement necessary functions which serve to process input data and print out processed results, both in screen and in text file. Specifically, this file has 5 functions executing 5 command line features and 1 vital function to determine which function will be executed.
- **main.cpp**: this file is mainly used to run the program by calling the principal function in command.cpp file.

2. Programming notes

- Use Microsoft Power BI to visualize statistics measured.
- Devise a function to write down statistics measured by sorting functions into a CSV file which will be used for further jobs.

- Implement functions measuring running time and counting the number of comparisons in the same format: for each task, they return the same data type and use the same arguments. It is to define two types of function pointer and create two arrays of functions with them.
- Implement separately functions processing command line.
- Use function clock() in library time.h to measure running time.
- Define a struct to store results of running time and the number of comparisons.

VI. List of references

https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/
https://en.wikipedia.org/wiki/Selection_sort#
https://www.geeksforgeeks.org/selection-sort/
https://www.geeksforgeeks.org/insertion-sort/
https://en.wikipedia.org/wiki/Insertion_sort#
https://www.geeksforgeeks.org/bubble-sort/
https://en.wikipedia.org/wiki/Bubble_sort
https://www.geeksforgeeks.org/heap-sort/
https://data-flair.training/blogs/heapsort/
https://en.wikipedia.org/wiki/Heapsort#
https://en.wikipedia.org/wiki/Merge_sort
https://www.geeksforgeeks.org/merge-sort/
https://en.wikipedia.org/wiki/Quicksort#
https://www.geeksforgeeks.org/quick-sort/
https://en.wikipedia.org/wiki/Radix_sort
https://www.geeksforgeeks.org/radix-sort/
https://en.wikipedia.org/wiki/Cocktail_shaker_sort
https://www.geeksforgeeks.org/cocktail-sort/
https://en.wikipedia.org/wiki/Shellsort
https://www.geeksforgeeks.org/shellsort/
https://chat.openai.com/
https://en.wikipedia.org/wiki/Counting_sort
https://www.geeksforgeeks.org/counting-sort/
https://www.w3resource.com/javascript-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-12.php
https://www.iostream.vn/article/merge-sort-u1Ti3U
https://codelearn.io/learning/data-structure-and-algorithms/856660
https://www.algolist.net/Algorithms/Sorting/Quicksort
https://github.com/leduythuccs/Sorting-Algorithms
https://github.com/HaiDuc0147/sortingAlgorithm