UNIVERSITY OF SCIENCE

Sorting Algorithms - An Overview

Data Structures and Algorithms

Huynh Minh Tuan - 20120024@student.hcmus.edu.vn

November 2021

Abstract

Sorting is nothing but alphabetizing, categorizing, arranging, or putting items in an ordered sequence. It is a key fundamental operation in the field of computer science. It is of extreme importance because it adds usefulness to data. In this report, I have compared eleven common sorting algorithms (Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort). I have developed a program in C++, Python and experimented with several input sizes 10,000, 30,000, 50,000, 100,000, 300,000, and 500,000 elements. The performance and efficiency of these algorithms in terms of CPU time consumption as well as the number of comparisons that have been recorded and presented in tabular and graphical form.

1. Introduction

Sorting is not a leap but it has emerged in parallel with the development of the human mind. In computer science, alphabetizing, arranging, categorizing, or putting data items in an ordered sequence on the basis of similar properties is called sorting. Sorting is of key importance because it optimizes the usefulness of data. We can observe plenty of sorting examples in our daily life, e.g. we can easily find required items in a shopping mall or utility store because the items are kept categorically.

The items to be sorted may be in various forms i.e. random as a whole, already sorted, very small or extremely large in numer, sorted in reverse order etc. There is no algorithm that is best for sorting all types of data. We must be familiar with sorting algorithms in terms of their suitability in a particular situation.

In this paper, I am going to compare eleven common sorting algorithms (Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort) for their CPU time consumption and number of compared operations on four different data arrangements (Sorted data (in ascending order), Nearly sorted data, Revherse sorted data, and Randomized data).

2.   Algorithm presentation

2.1.   Selection Sort

Idea

The Selection Sort is based on the idea of finding the minimum element in an unsorted array and then putting it in its correct position in a sorted array.

Pseudo code

Algorithm 2.1: Selection Sort

   Input: $a_1, a_2, ..., a_N$
   Output: $a_1, a_2, ..., a_N$ (in sorted)
1 **for** $i \leftarrow 1$ *to* N **do**
2 $minIndex \leftarrow i$
3   **for** $j \leftarrow i + 1$ *to* N **do**
4   **if** $a_{minIndex} > a_j$ **then**
5 $minIndex \leftarrow j$
6   **end**
7   **end**
8   swap($a_{minIndex}, a_i$)
9 **end**

Complexity

Best case time complexity: $O(N^2)$
Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.2.   Insertion Sort

Idea

The main idea of insertion sort is that array is divided in two parts which left part is already sorted, and right part is unsorted. Values from the unsorted part are picked and placed at the correct position in the sorted part. So, at every iteration sorted part grows by one element which is called key. During an iteration, if compared element is greater than key then compared element has to shift to right to open a position for key.

Pseudo code

Algorithm 2.2: Insertion Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

1 **for** $i \leftarrow 2$ *to* N **do**
2 $\quad$ $k \leftarrow i - 1$
3 $\quad$ $key \leftarrow a_i$
4 $\quad$ **while** $a_k > key$ *and* $k \geq 0$ **do**
5 $\quad\quad$ $a_{k+1} \leftarrow a_k$
6 $\quad\quad$ $k \leftarrow k - 1$
7 $\quad$ **end**
8 $\quad$ $a_{k+1} \leftarrow key$
9 **end**

Complexity

Best case time complexity: $O(N)$
Average case time complexity: $O(N^2)$
Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.3. Bubble Sort

Idea

Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

Pseudo code

Algorithm 2.3: Bubble Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

1 **for** $i \leftarrow N$ *to* 1 **do**
2 $\quad$ $isSwap \leftarrow False$
3 $\quad$ **for** $j \leftarrow 1$ *to* $i - 1$ **do**
4 $\quad\quad$ **if** $a_j > a_{j+1}$ **then**
5 $\quad\quad\quad$ $isSwap \leftarrow True$
6 $\quad\quad\quad$ swap(a_j, a_{j+1})
7 $\quad\quad$ **end**
8 $\quad$ **end**
9 $\quad$ **if** $isSwap = False$ **then**
10 $\quad\quad$ stop algorithm
11 $\quad$ **end**
12 **end**

In this paper, I implemented bubble sort with a flag $isSwap$ to stop the algorithm early when the array is sorted.

Complexity

Best case time complexity: $O(N)$
Average case time complexity: $O(N^2)$

Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.4. Shaker Sort

Idea

Shaker sort is a bidirectional version of bubble sort. The Bubble sort algorithm always traverses elements from left and moves the largest element to its correct position in first iteration and second largest in second iteration and so on. Shaker sort orders the array in both directions. Hence every iteration of the algorithm consists of two phases. In the first one, the lightest bubble ascends to the end of the array, in the second phase the heaviest bubble descends to the beginning of the array.

Pseudo code

Algorithm 2.4: Shaker Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

1   $left \leftarrow 0$
2   $right \leftarrow N - 1$
3   $k \leftarrow 0$
4   **for** $i \leftarrow left$ *to* $right$ **do**
     // phase 1
5     $isSwap \leftarrow False$
6   **for** $j \leftarrow left$ *to* $right - 1$ **do**
7   **if** $a_j > a_{j+1}$ **then**
8         $isSwap \leftarrow True$
9         $swap(a_j, a_{j+1})$
10        $k \leftarrow j$
11   **end**
12   **end**
13   **if** $isSwap = False$ **then**
14   stop algorithm
15   **end**
16     $right \leftarrow k$
     // phase 2
17     $isSwap \leftarrow False$
18   **for** $j \leftarrow right$ *to* $left + 1$ **do**
19   **if** $a_j < a_{j-1}$ **then**
20         $isSwap \leftarrow True$
21         $swap(a_j, a_{j-1})$
22        $k \leftarrow j$
23   **end**
24   **end**
25   **if** $isSwap = False$ **then**
26   stop algorithm
27   **end**
28     $left \leftarrow k$
29 **end**

Complexity

Best case time complexity: $O(N)$
Average case time complexity: $O(N^2)$
Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.5.   Shell Sort

Idea

Shell sort is a generalized version of the insertion sort algorithm. It first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted.

The interval between the elements is reduced based on the sequence used. Some of the optimal sequences that can be used in the shell sort algorithm are:

- Shell's original sequence

- Knuth's increments

- Sedgewick's increments

- Hibbard's increments

- ...

In this paper, I only implemented the algorithm with optimal sequence based on Shell's original sequence.

Pseudo code

Algorithm 2.5: Shell Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

1   $interval \leftarrow \dfrac{N}{2}$
2   **while** $interval > 0$ **do**
3   **for** $i \leftarrow interval\ to\ N$ **do**
4        $temp \leftarrow a_i$
5        $j \leftarrow i$
6   **while** $interval \leq j\ and\ a_{j-interval} > temp$ **do**
7          $a_j \leftarrow a_{j-interval}$
8          $j \leftarrow j - interval$
9   **end**
10   **end**
11      $a_j \leftarrow temp$
12      $interval \leftarrow \dfrac{interval}{2}$
13 **end**

Complexity

Best case time complexity: $O(N)$
Average case time complexity: $O(N \log N)$
Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.6.   Heap Sort

Idea

Heap sort is a comparison-based sorting algorithm. Heap sort can be thought of as an improved selection sort: like selection sort, heap sort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region.

Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a **heap data structure** to more quickly find the largest element in each step.

Pseudo code

Algorithm 2.6: Heap Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

1 **Function** `HeapRebuild`(a, *pos*, N)**:**
2   **while** $2 \cdot pos + 1 <= N$ **do**
3        $j = 2 \cdot pos + 1$
4   **if** $j < N$ **then**
5   **if** $a_j < a_{j+1}$ **then**
6              $j \leftarrow j + 1$
7   **end**
8   **end**
9   **if** $a_{pos} \geq a_j$ **then**
10   **return**
11   **end**
12   swap(a_{pos}, a_j)
13        $pos \leftarrow j$
14   **end**
15 **Function** `HeapConstruct`(a, N)**:**
16   **for** $i \leftarrow N/2$ *to* 0 **do**
17   HeapRebuild(a, i, n)
18   **end**
19 **Function** `HeapSort`(a, N)**:**
20   HeapConstruct(a, N)
21     $r \leftarrow N$
22   **while** $r > 0$ **do**
23   swap(a_1, a_N)
24   HeapRebuild(a, 1, r)
25        $r \leftarrow r - 1$
26   **end**

Complexity

Best case time complexity: $O(N \log N)$
Average case time complexity: $O(N \log N)$
Worst case time complexity: $O(N \log N)$
Worst case space complexity: $O(1)$

2.7. Merge Sort

Idea

Merge sort is a recursive sorting algorithm based on a "divide and conquer" approach. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

Pseudo code

Algorithm 2.7: Merge Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

1 Function Merge($a, first, mid, last$):
2 | $n_1 \leftarrow mid - first + 1$
3 | $n_2 \leftarrow last - mid$
4 | $L \leftarrow a_{first}, a_{first+1}, ..., a_{mid}$
5 | $R \leftarrow a_{mid+1}, a_{mid+2}, ..., a_{last}$
   | // merge
6 | $i \leftarrow 0$
7 | $j \leftarrow 0$
8 | $k \leftarrow first$
9 | **while** $i < n_1$ *and* $j < n_2$ **do**
10 | | **if** $L_i < R_j$ **then**
11 | | | $a_k \leftarrow L_i$
12 | | | $i \leftarrow i + 1$
13 | | **else**
14 | | | $a_k \leftarrow R_j$
15 | | | $j \leftarrow j + 1$
16 | | **end**
17 | | $k \leftarrow k + 1$
18 | **end**
19 | **while** $j < n_2$ **do**
20 | | $a_k \leftarrow R_j$
21 | | $k \leftarrow k + 1$
22 | | $j \leftarrow j + 1$
23 | **end**
24 | **while** $i < n_1$ **do**
25 | | $a_k \leftarrow L_i$
26 | | $k \leftarrow k + 1$
27 | | $i \leftarrow i + 1$
28 | **end**
29 Function MergeSort($a, first, last$):
30 | **if** $first < last$ **then**
31 | | $mid \leftarrow first + (last - first)/2$
32 | | MergeSort($a, first, mid$)
33 | | MergeSort($a, mid + 1, last$)
34 | | Merge($a, first, mid, last$)
35 | **end**

Complexity

Best case time complexity: $O(N \log N)$
Average case time complexity: $O(N \log N)$
Worst case time complexity: $O(N \log N)$
Worst case space complexity: $O(N)$

2.8.   Quick Sort

Idea

Like Merge Sort, Quick Sort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Pick first element as pivot.

- Pick last element as pivot

- Pick a random element as pivot.

- Pick median as pivot.

In this paper, I implemented the algorithm with pivot is a median of array.

Pseudo code

Algorithm 2.8: Quick Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

1 Function Partition(a, l, r):
2 $\quad$ $pivot \leftarrow a_{(l+r)/2}$
3 $\quad$ **while** $l \leq r$ **do**
4 $\quad\quad$ **while** $a_l < pivot$ **do**
5 $\quad\quad\quad$ $l \leftarrow l + 1$
6 $\quad\quad$ **end**
7 $\quad\quad$ **while** $a_r > pivot$ **do**
8 $\quad\quad\quad$ $r \leftarrow r - 1$
9 $\quad\quad$ **end**
10 $\quad\quad$ **if** $l \leq r$ **then**
11 $\quad\quad\quad$ swap(a_l, a_r)
12 $\quad\quad\quad$ $l \leftarrow l + 1$
13 $\quad\quad\quad$ $r \leftarrow r - 1$
14 $\quad\quad$ **end**
15 $\quad$ **end**
16 $\quad$ **return** l
17 Function QuickSort(a, l, r):
18 $\quad$ **if** $l < r$ **then**
19 $\quad\quad$ $mid \leftarrow$ Partition(a, l, r)
20 $\quad\quad$ QuickSort(a, l, $mid - 1$)
21 $\quad\quad$ QuickSort(a, mid, r)
22 $\quad$ **end**

Complexity

Best case time complexity: $O(N)$
Average case time complexity: $O(N \log N)$
Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.9. Counting Sort

Idea

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.

Pseudo code

Algorithm 2.9: Counting Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

1 $maxVal \leftarrow a_0$
2 **for** $i \leftarrow 1$ to N **do**
3 $\quad$ **if** $a_i > maxVal$ **then**
4 $\quad\quad$ $maxVal \leftarrow a_i$
5 $\quad$ **end**
6 **end**
7 $count \leftarrow [0] * (maxVal + 1)$ `// initialize 0-value counting array`
8 **foreach** $u \in a$ **do**
9 $\quad$ $count_u \leftarrow count_u + 1$
10 **end**
`// restore the elements to array`
11 $idx \leftarrow 0$
12 **for** $i \leftarrow 0$ to $maxVal$ **do**
13 $\quad$ **while** $count_i > 0$ **do**
14 $\quad\quad$ $a_{idx} \leftarrow i$
15 $\quad\quad$ $idx \leftarrow idx + 1$
16 $\quad\quad$ $count_i \leftarrow count_i - 1$
17 $\quad$ **end**
18 **end**

Complexity

Best case time complexity: $O(\text{MaxValue})$
Average case time complexity: $O(\text{MaxValue})$
Worst case time complexity: $O(\text{MaxValue})$
Worst case space complexity: $O(\text{MaxValue})$
where MaxValue is the maximum value of input array.

2.10. Radix Sort

Idea

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.

Pseudo code

Algorithm 2.10: Radix Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

1   $maxVal \leftarrow a_0$
2   **for** $i \leftarrow 1$ *to* N **do**
3   **if** $a_i > maxVal$ **then**
4       $maxVal \leftarrow a_i$
5   **end**
6   **end**
7   $exp \leftarrow 1$
8   **while** $\dfrac{maxVal}{exp} > 0$ **do**
9     $digit \leftarrow$ array with of N elements
10   **for** $i \leftarrow 1$ *to* N **do**
       // get corresponding digit
11      $digit_i \leftarrow \dfrac{a_i}{exp} \bmod 10$
12   **end**
     // do counting sort of a[] according to the digit represented by exp
13   CountingSort(a, n, $digit$)
14    $exp \leftarrow exp \cdot 10$
15 **end**

Complexity

Best case time complexity: $O(N \cdot d)$
Average case time complexity: $O(N \cdot d)$
Worst case time complexity: $O(N \cdot d)$
Worst case space complexity: $O(N)$
where d is the maximum number of digits

2.11.   Flash Sort

Idea

The main idea of Flash Sort is to assign each of the n input elements to one of m partitions, efficiently rearranges the input to place the partitions in the correct order, then sorts each partition.

The algorithm can be represented as four stages:

1. The number of partitions is calculated.

2. Set clear boundaries in our original array for every partitions.

3. Rearrange the elements in the original array so that each of them was in its place, in its partition.

4. Do Insertion Sort for sorting locally.

Pseudo code

Algorithm 2.11: Flash Sort

    Input: $a_1, a_2, ..., a_N$
    Output: $a_1, a_2, ..., a_N$ (in sorted)
    // stage 1
    // d should be in range [0.4, 0.6]
 1 $m \leftarrow d \cdot n$
    // stage 2
 2 $minVal \leftarrow a_1$
 3 $maxIndex \leftarrow 1$
 4 **for** $i \leftarrow 1$ *to* N **do**
 5   **if** $a_i < minVal$ **then**
 6 $minVal \leftarrow a_i$
 7   **end**
 8   **if** $a_{maxIndex} < a_i$ **then**
 9 $maxIndex \leftarrow i$
10   **end**
11 **end**
12 **if** $a_{maxIndex} == minVal$ **then**
13   stop algorithm
14 **end**
    // classify elements into corresponding partition
15 $c \leftarrow \dfrac{m - 1}{a_{maxIndex} - minVal}$
16 **for** $i \leftarrow 1$ *to* N **do**
17 $cls \leftarrow c \cdot (a_i - minVal)$
18 $L_{cls} \leftarrow L_{cls} + 1$
19 **end**
20 **for** $i \leftarrow 1$ *to* m **do**
21 $L_i \leftarrow L_i + L_{i-1}$
22 **end**
    // stage 3
23 $\text{swap}(a_{maxIndex}, a_1)$ $nmove \leftarrow 0$
24 $j \leftarrow 0$
25 $k \leftarrow m - 1$
26 $t \leftarrow 0$
27 **while** $nmove < N$ **do**
28   **while** $j > L_k - 1$ **do**
29 $j \leftarrow j + 1$
30 $k \leftarrow c \cdot (a_j - minVal)$
31   **end**
32 $flash \leftarrow a_j$
33   **if** $k \ ¡ \ 0$ **then**
34   **break**
35   **end**
36   **while** $j \neq L_k$ **do**
37 $k \leftarrow c \cdot (flash - minVal)$
38 $L_k \leftarrow L_k - 1$
39 $t \leftarrow L_k$
40 $hold \leftarrow a_t$
41 $a_t \leftarrow flash$
42 $flash \leftarrow hold$
43 $nmove \leftarrow nmove + 1$
44   **end**
        // stage 4
45 $\text{InsertionSort}(a, n)$
46 **end**

Data Order: Sorted data					
Data size	10,000		30,000		50,000	
Result	Time (ms)	Comparision	Time (ms)	Comparision	Time (ms)	Comparision
Selection	134.179	50005001	1010.114	450015001	3161.295	1250025001
Insertion	0.055	19999	0.155	59999	0.179000	99999
Bubble	0.036	20001	0.075	60001	0.123	100001
Shaker	0.026	20001	0.073	60001	0.129	100001
Shell	0.558	240024	1.762000	780029	3.423000	1400028
Heap	2.333	518705	5.418	1739633	9.129000	3056481
Merge	1.694	406234	3.632	1332186	6.21	2320874
Quick	0.598	193611	1.327	627227	2.218	1084459
Counting	0.162	60003	0.369	180003	0.572	300003
Radix	1.489	170106	3.738	630132	7.198	1050132
Flash	0.597	103496	1.118	310496	2.359	517496

Table 1: Experimental results on sorted data

Complexity

The time complexity of Flash Sort base on choosing value m. For example, if m is chosen proportional to $\sqrt{n}$, the time complexity is $O(n^{3/2})$. In this paper, I chosen 0.45 for m.
Space complexity: $O(1)$

3. Experimental results

All the eleven sorting algorithms were implemented in C++ programming language and tested on six input of lenght 10000, 30000, 50000, 100000, 300000, and 500000 of four data orders (Sorted data, Nearly sorted data, Reverse sorted data and Randomized data). All experiments were executed on machine Operating System having Intel(R) Core(TM) i5-10210U CPU @ 1.60Ghz (8 CPUs) and installed memory (RAM) 8GB. The results were calculated after tabulation and their graphical represention was developed using Python programming language.
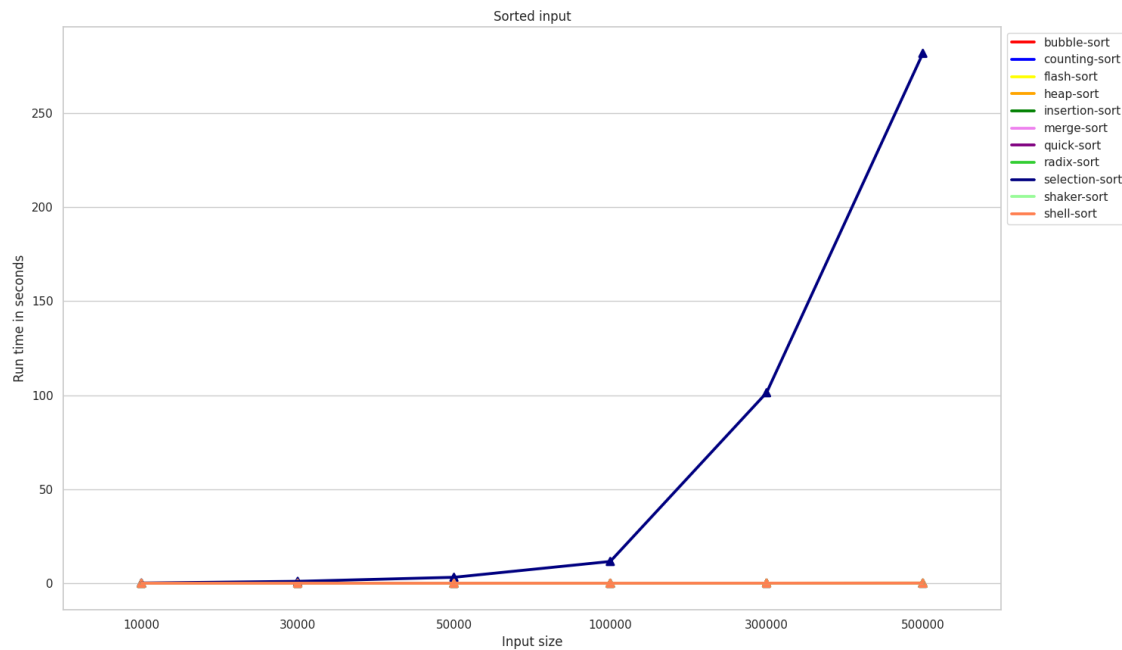
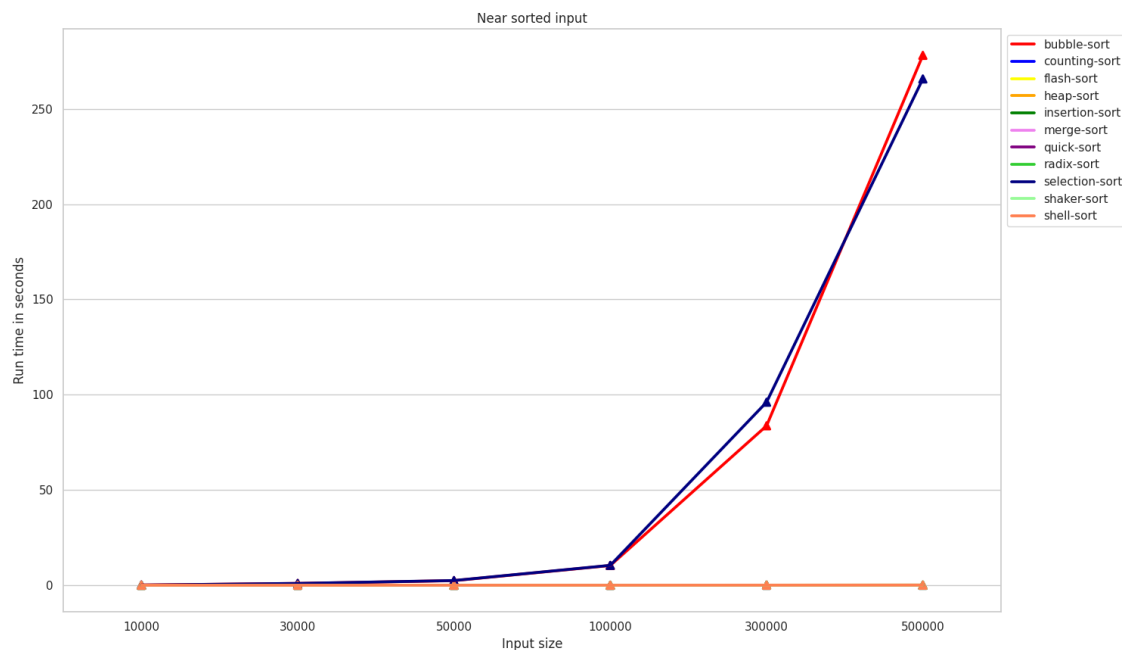Figure 1: Visualizing the algorithms' running times on sorted data

Figure 2: Visualizing the algorithms' running times on nearly sorted data
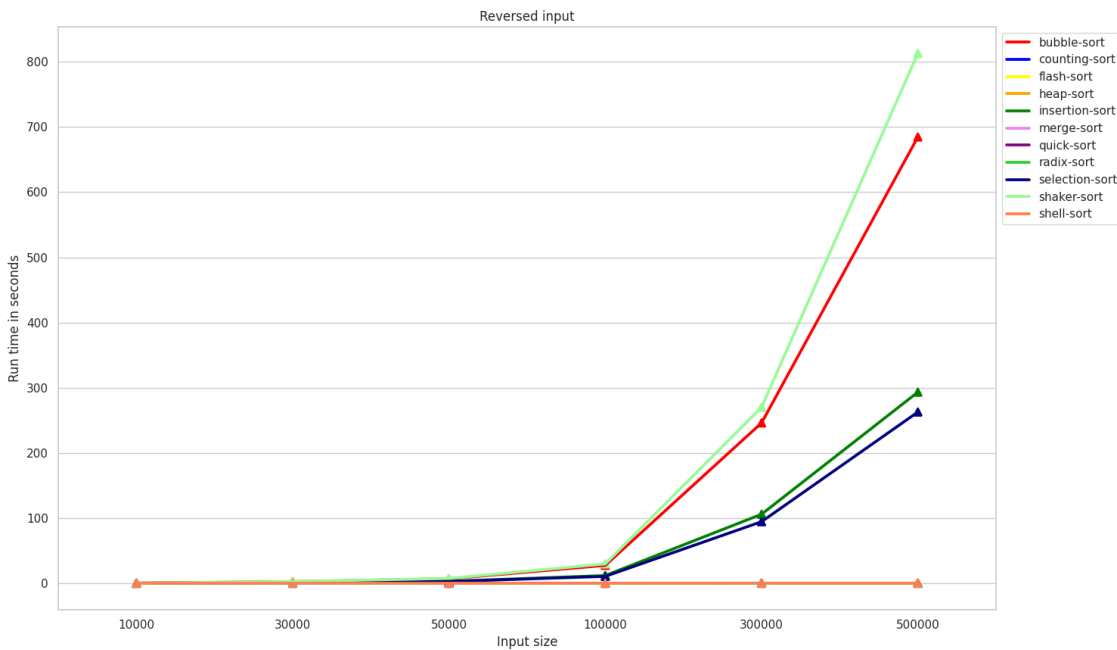
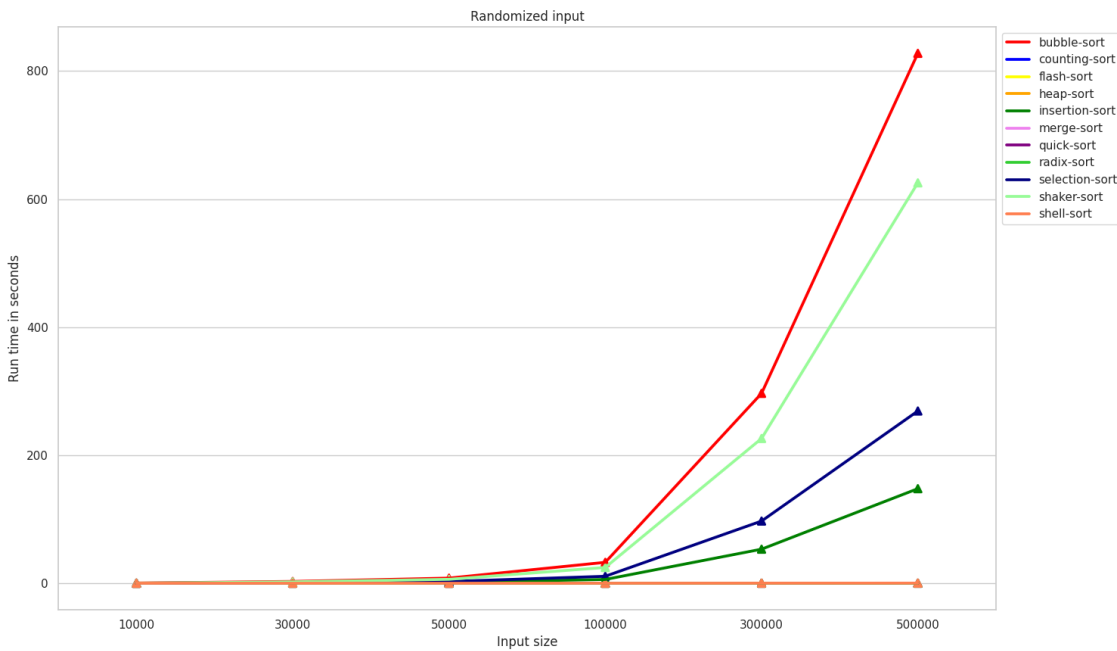Figure 3: Visualizing the algorithms' running times on reverse sorted data

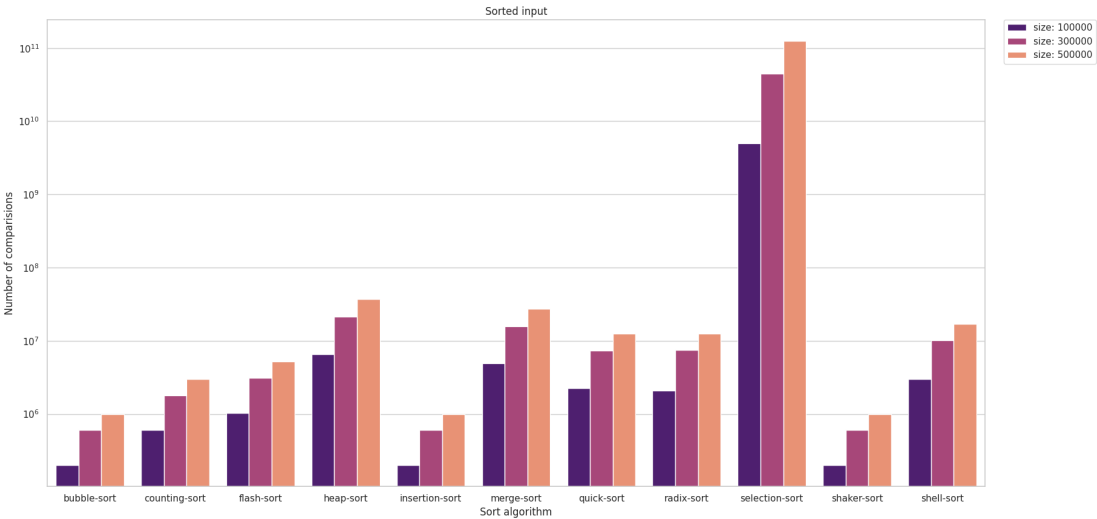Figure 4: Visualizing the algorithms' running times on randomized data

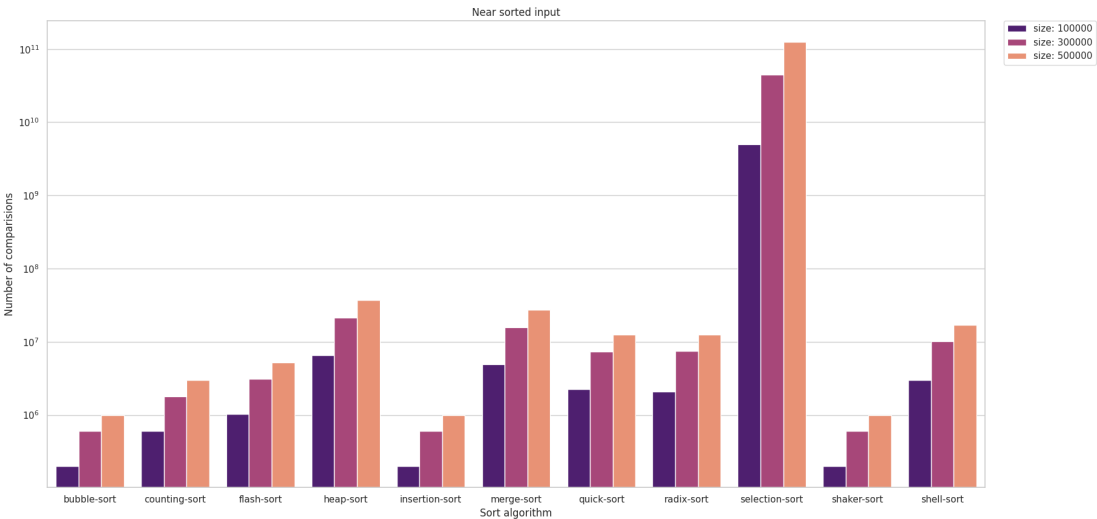Figure 5: Visualizing the algorithms' numbers of comparisons on sorted data

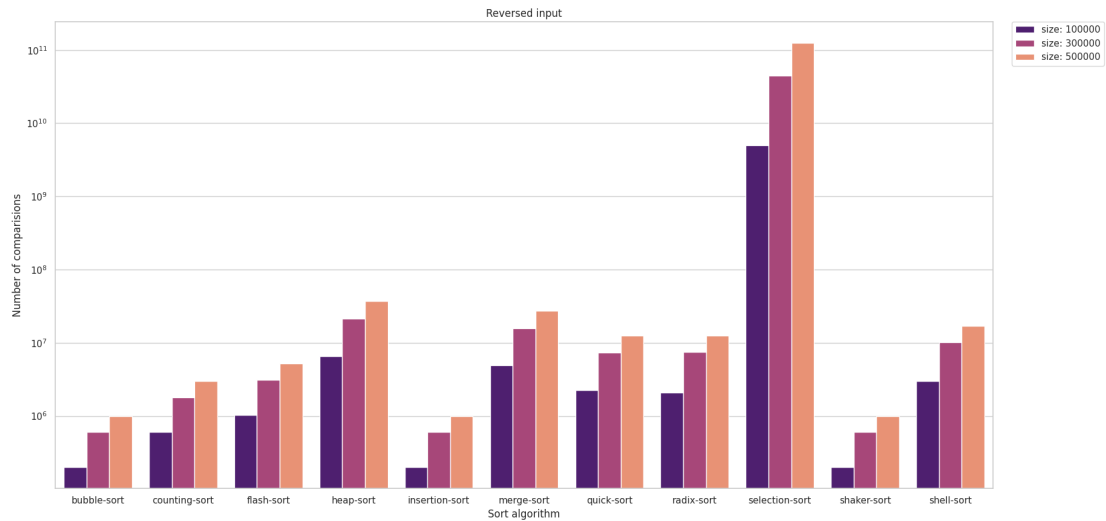Figure 6: Visualizing the algorithms' numbers of comparisons on nearly sorted data

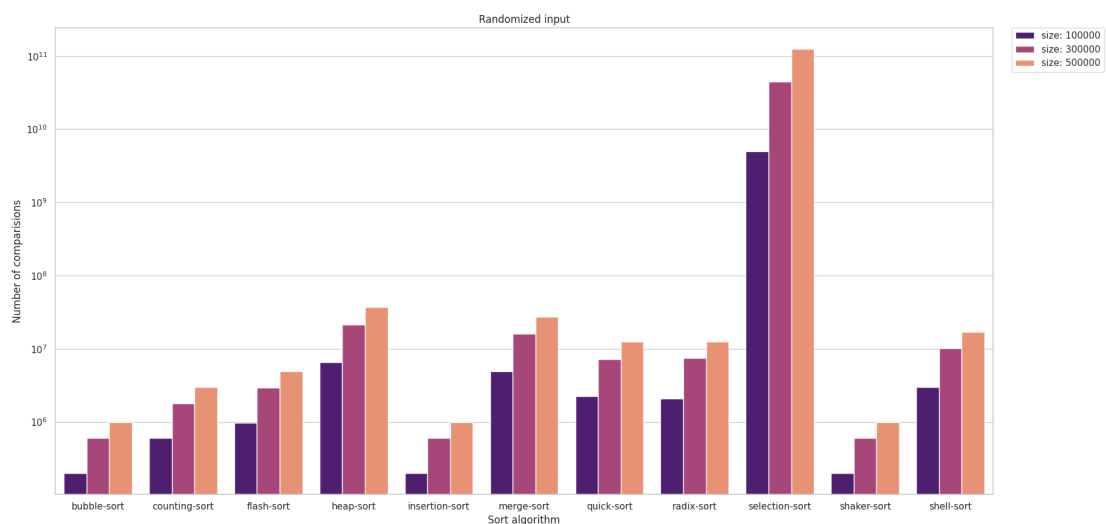Figure 7: Visualizing the algorithms' numbers of comparisons on reverse sorted data

Figure 8: Visualizing the algorithms' numbers of comparisons on randomized data

4. Project organization

C++ programming language was used in sorting algorithms' implementation. Python programming language and open libraries (Pandas, Matplotlib, Seaborn) were used in processing data and graphical visualizing.

Source code: `https://github.com/huynhtuan17ti/Sorting-Overview`

5. References

1. `https://www.geeksforgeeks.org/` (For nice explanations and source code of several sorting algorithms)

2. `https://www.wikipedia.org/` (Scientific explanations of all sorting algorithms)

3. `https://www.researchgate.net/publication/315662067_Sorting_Algorithms_-_A_Comparative_Study`

4. `https://www.researchgate.net/publication/259911982_Review_on_Sorting_Algorithms_A_Comparative_Study`

5. Introduction to Algorithms (Third edition)