

# 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 number, 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, Reverse 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:**  $a_1, a_2, \dots, a_N$   
**Output:**  $a_1, a_2, \dots, a_N$  (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:**  $a_1, a_2, \dots, a_N$ **Output:**  $a_1, a_2, \dots, a_N$  (in sorted)

```

1 for  $i \leftarrow 2$  to  $N$  do
2    $k \leftarrow i - 1$ 
3    $key \leftarrow a_i$ 
4   while  $a_k > key$  and  $k \geq 0$  do
5      $a_{k+1} \leftarrow a_k$ 
6      $k \leftarrow k - 1$ 
7   end
8    $a_{k+1} \leftarrow key$ 
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:**  $a_1, a_2, \dots, a_N$ **Output:**  $a_1, a_2, \dots, a_N$  (in sorted)

```

1 for  $i \leftarrow N$  to 1 do
2    $isSwap \leftarrow False$ 
3   for  $j \leftarrow 1$  to  $i - 1$  do
4     if  $a_j > a_{j+1}$  then
5        $isSwap \leftarrow True$ 
6       swap( $a_j, a_{j+1}$ )
7     end
8   end
9   if  $isSwap = False$  then
10    stop algorithm
11  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:  $a_1, a_2, \dots, a_N$ 
Output:  $a_1, a_2, \dots, a_N$  (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:**  $a_1, a_2, \dots, a_N$

**Output:**  $a_1, a_2, \dots, a_N$  (in sorted)

```

1  $interval \leftarrow \frac{N}{2}$ 
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   $interval \leftarrow \frac{interval}{2}$ 
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:  $a_1, a_2, \dots, a_N$ 
Output:  $a_1, a_2, \dots, a_N$  (in sorted)
1 Function HeapRebuild( $a, pos, N$ ):
2   while  $2 \cdot pos + 1 \leq N$  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:**  $a_1, a_2, \dots, a_N$

**Output:**  $a_1, a_2, \dots, a_N$  (in sorted)

```

1 Function Merge( $a, first, mid, last$ ):
2    $n_1 \leftarrow mid - first + 1$ 
3    $n_2 \leftarrow last - mid$ 
4    $L \leftarrow a_{first}, a_{first+1}, \dots, a_{mid}$ 
5    $R \leftarrow a_{mid+1}, a_{mid+2}, \dots, a_{last}$ 
   // 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:**  $a_1, a_2, \dots, a_N$

**Output:**  $a_1, a_2, \dots, a_N$  (in sorted)

```

1 Function Partition( $a, l, r$ ):
2    $pivot \leftarrow a_{(l+r)/2}$ 
3   while  $l \leq r$  do
4     while  $a_l < pivot$  do
5        $l \leftarrow l + 1$ 
6     end
7     while  $a_r > pivot$  do
8        $r \leftarrow r - 1$ 
9     end
10    if  $l \leq r$  then
11      swap( $a_l, a_r$ )
12       $l \leftarrow l + 1$ 
13       $r \leftarrow r - 1$ 
14    end
15  end
16  return  $l$ 
17 Function QuickSort( $a, l, r$ ):
18  if  $l < r$  then
19     $mid \leftarrow$  PARTITION( $a, l, r$ )
20    QUICKSORT( $a, l, mid - 1$ )
21    QUICKSORT( $a, mid, r$ )
22  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:  $a_1, a_2, \dots, a_N$ 
Output:  $a_1, a_2, \dots, a_N$  (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  $count \leftarrow [0] * (maxVal + 1)$  // initialize 0-value counting array
8 foreach  $u \in a$  do
9    $count_u \leftarrow count_u + 1$ 
10 end
    // restore the elements to array
11  $idx \leftarrow 0$ 
12 for  $i \leftarrow 0$  to  $maxVal$  do
13   while  $count_i > 0$  do
14      $a_{idx} \leftarrow i$ 
15      $idx \leftarrow idx + 1$ 
16      $count_i \leftarrow count_i - 1$ 
17   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:**  $a_1, a_2, \dots, a_N$   
**Output:**  $a_1, a_2, \dots, a_N$  (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  $\frac{maxVal}{exp} > 0$  do
9    $digit \leftarrow$  array with of  $N$  elements
10  for  $i \leftarrow 1$  to  $N$  do
11    // get corresponding digit
12     $digit_i \leftarrow \frac{a_i}{exp} \bmod 10$ 
13  end
14  // do counting sort of  $a[]$  according to the digit represented by  $exp$ 
15  COUNTINGSORT( $a, n, digit$ )
16   $exp \leftarrow exp \cdot 10$ 
17 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:  $a_1, a_2, \dots, a_N$ 
Output:  $a_1, a_2, \dots, a_N$  (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
     $m \leftarrow m - 1$ 
15  $c \leftarrow \frac{m - 1}{a_{maxIndex} - minVal}$ 
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  $swap(a_{maxIndex}, a_1)$   $nmove \leftarrow 0$ 
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  $k \neq 0$  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    $InsertionSort(a, n)$ 
46 end

```

---

## 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 chose 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 lenth 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 representation was developed using Python programming language.

Data Order: Sorted data						
Data size	10,000		30,000		50,000	
Result	Time (ms)	Comparision	Time (ms)	Comparision	Time (ms)	Comparision
Selection	112.733	50005001	993.92	450015001	2661.613	1250025001
Insertion	0.052	19999	0.119	59999	0.177	99999
Bubble	0.029	20001	0.067	60001	0.114	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

Data size	100,000		300,000		500,000	
Result	Time (ms)	Comparision	Time (ms)	Comparision	Time (ms)	Comparision
Selection	11578.892	5000050001	101293.857	45000150001	281740.886	125000250001
Insertion	0.411	199999	1.044	599999	1.829	999999
Bubble	0.282	200001	0.765	600001	1.303	1000001
Shaker	0.245	200001	0.778	600001	1.272	1000001
Shell	7.336	3000029	25.944	10200035	41.04	17000033
Heap	19.542	6519813	60.3	21431637	102.634	37116275
Merge	1.694	406234	3.632	1332186	6.21	2320874
Quick	13.286	4891754	43.119	15848682	71.659	27234634
Counting	1.267	600003	3.35	1800003	5.708	3000003
Radix	13	2100132	44.787	7500158	76.2	12500158
Flash	4.02	1034996	11.634	3104996	19.422	5174996

Table 1: Experimental results on sorted data

Data Order: Nearly Sorted data						
Data size	10,000		30,000		50,000	
Result	Time (ms)	Comparision	Time (ms)	Comparision	Time (ms)	Comparision
Selection	114.804	50005001	892.508	450015001	2815.055	1250025001
Insertion	0.272	186007	0.533	421299	1.59	792443
Bubble	138.364	95109345	825.406	773520000	2760.609	2463630480
Shaker	0.538	195793	1.251	470236	2.147	833872
Shell	0.675	288983	2.146	907030	4.659	1684266
Heap	1.407	518491	5.226	1739623	11.406	3056352
Merge	1.052	421044	3.477	1381719	5.788	2407406
Quick	0.422	193651	1.343	627279	2.527	1084495
Counting	0.122	60003	0.364	180003	0.519	300003
Radix	1.026	170106	3.725	630132	6.265	1050132
Flash	0.455	103470	1.063	310464	1.951	517470

Data size	100,000		300,000		500,000	
Result	Time (ms)	Comparision	Time (ms)	Comparision	Time (ms)	Comparision
Selection	10517.345	5000050001	98131.833	45000150001	268308.098	125000250001
Insertion	2.862	2143771	3.917	2784471	9.615	7687035
Bubble	10614.255	9835488445	91618.611	80529222960	244003.655	214177110017
Shaker	6.193	2239517	9.282	3358098	21.366	8256564
Shell	9.193	3710600	28.038	11204764	39.989	19115054
Heap	18.311	6519703	57.692	21431472	90.107	37116054
Merge	13.929	5052616	40.513	16239347	63.774	27981864
Quick	4.53	2268955	15.264	7275735	22.691	12475755
Counting	0.962	600003	3.102	1800003	4.906	3000003
Radix	12.661	2100132	46.058	7500158	65.558	12500158
Flash	3.757	1034972	10.606	3104966	17.504	5174966

Table 2: Experimental results on nearly sorted data

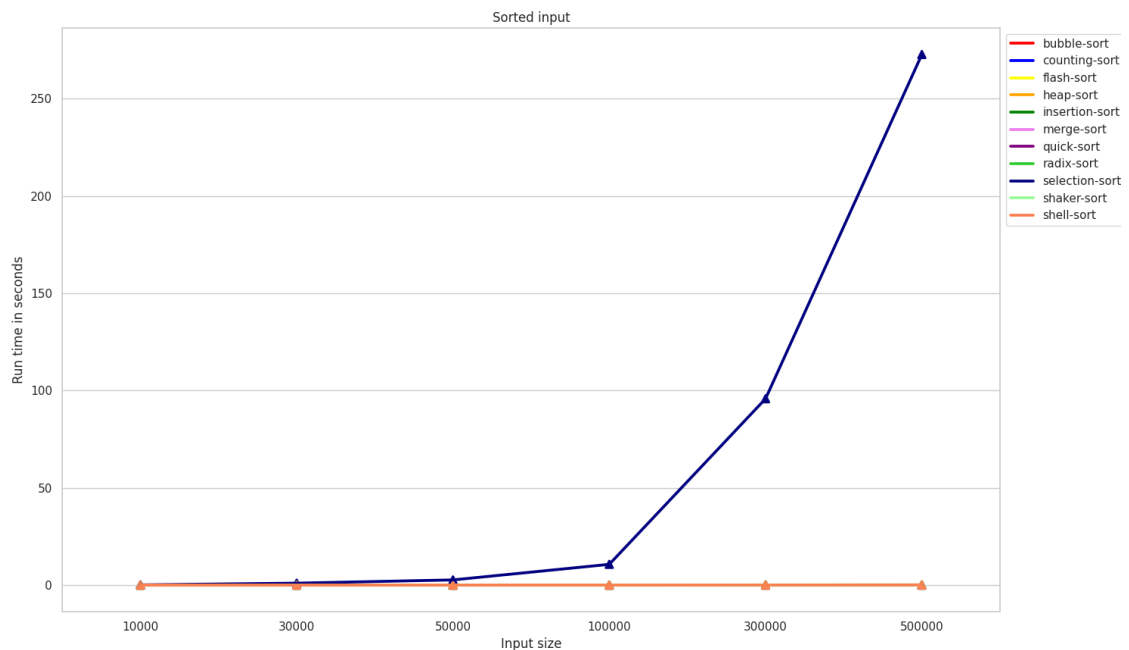


Figure 1: Visualizing the algorithms' running times on sorted data

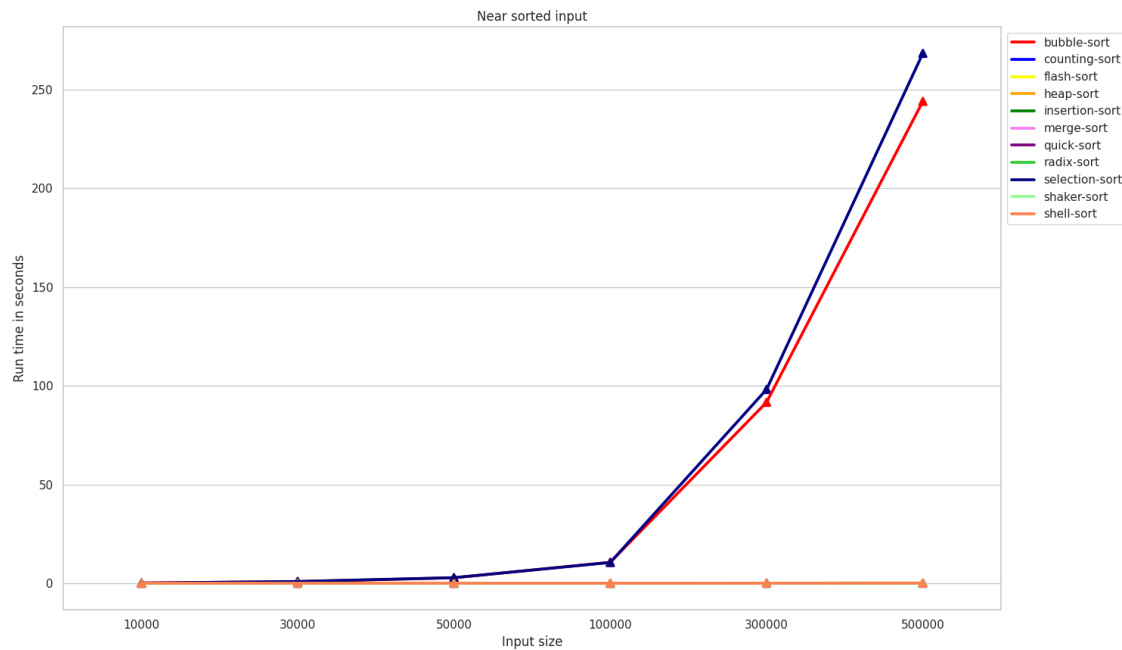


Figure 2: Visualizing the algorithms' running times on nearly sorted data

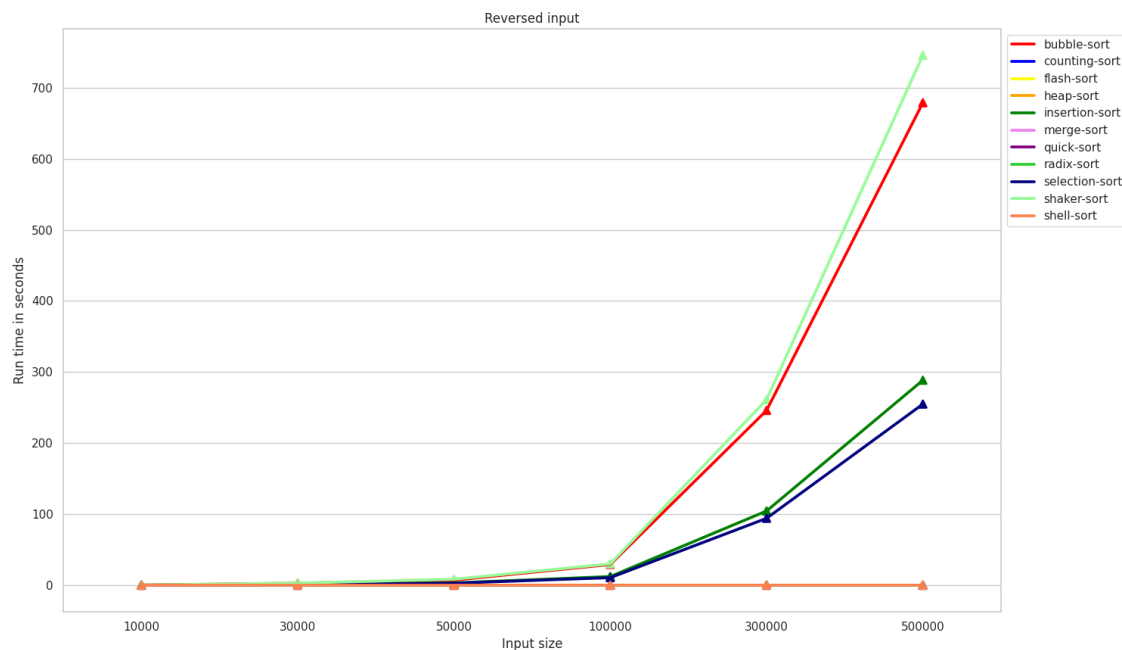


Figure 3: Visualizing the algorithms' running times on reverse sorted data

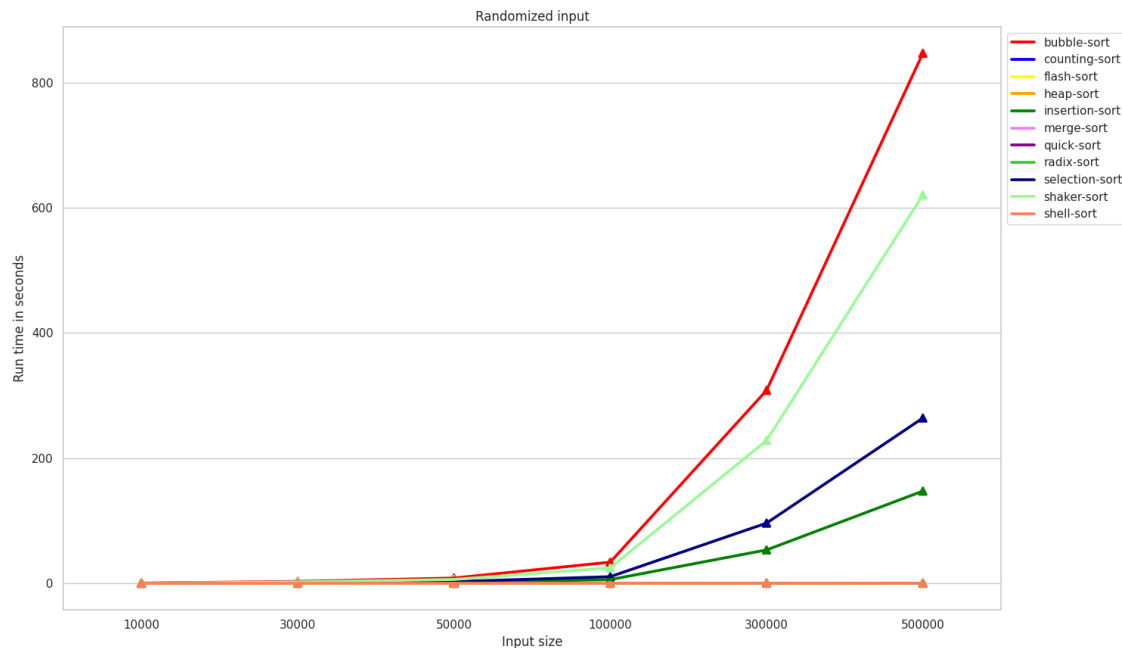


Figure 4: Visualizing the algorithms’ running times on randomized data

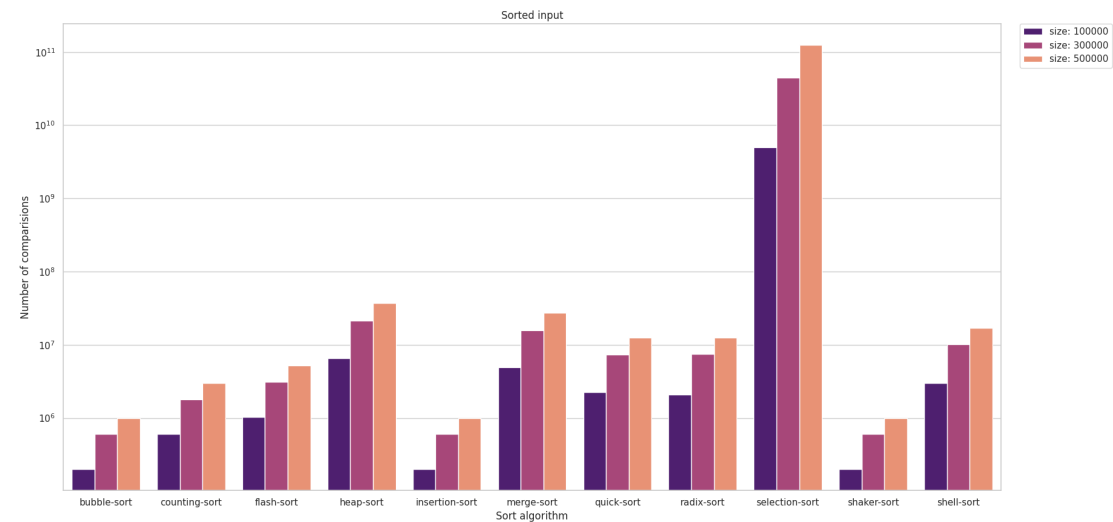


Figure 5: Visualizing the algorithms’ numbers of comparisons on sorted data

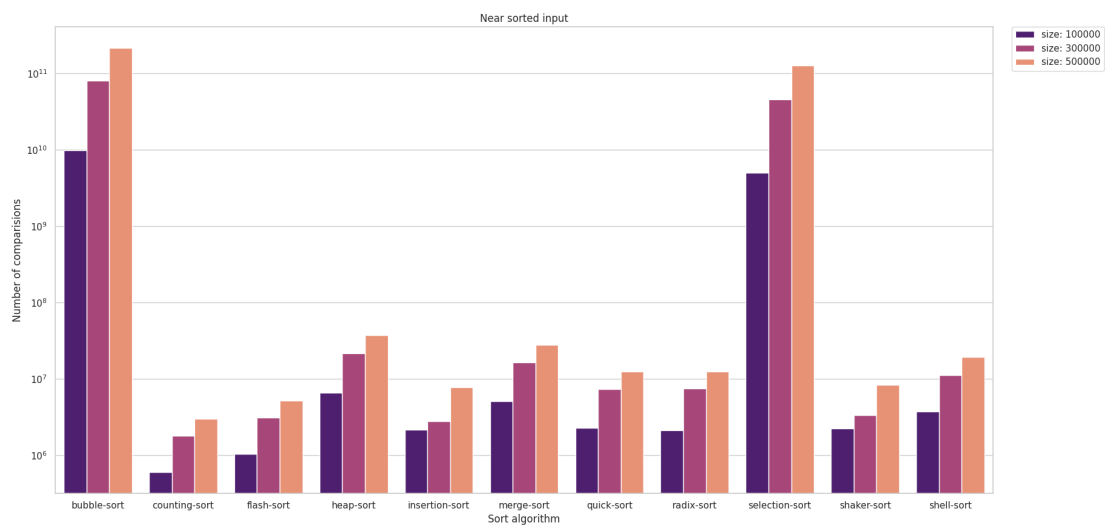


Figure 6: Visualizing the algorithms' numbers of comparisons on nearly sorted data

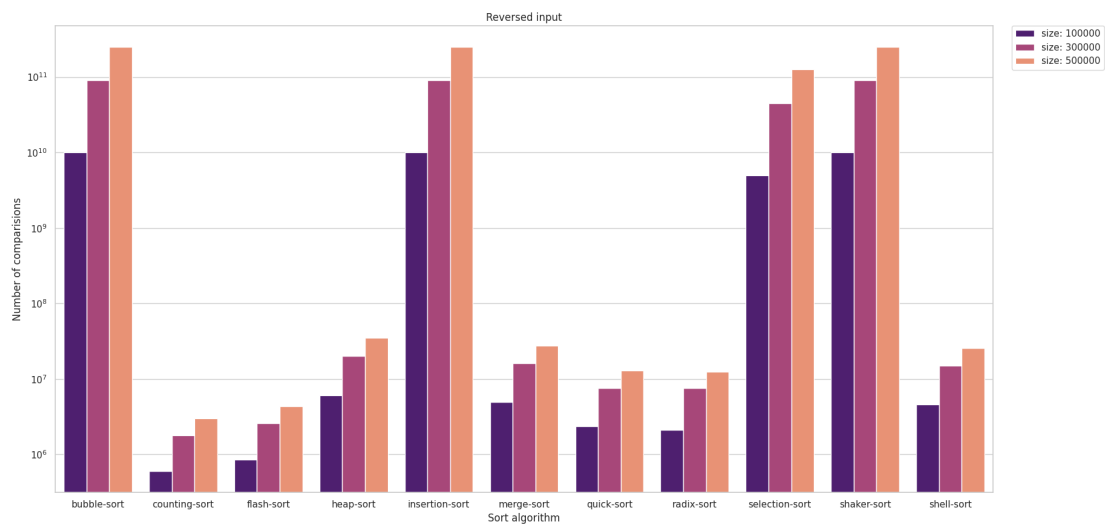


Figure 7: Visualizing the algorithms' numbers of comparisons on reverse sorted data



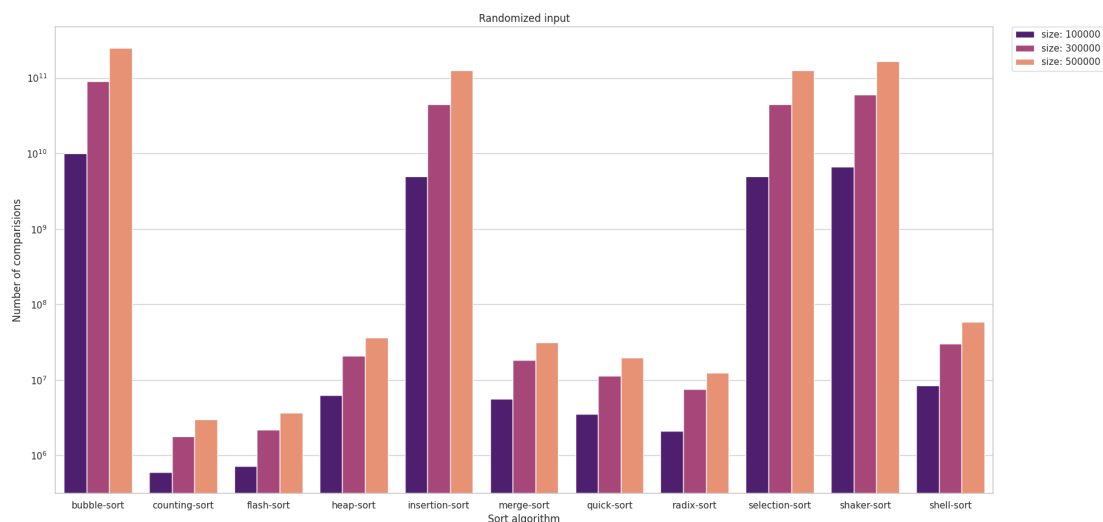


Figure 8: Visualizing the algorithms' numbers of comparisons on randomized data

## 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/> (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](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](https://www.researchgate.net/publication/259911982_Review_on_Sorting_Algorithms_A_Comparative_Study)
5. Introduction to Algorithms (Third edition)