Sorting Algorithms - An Overview

Huynh Minh Tuan[1] and Bui Huy Thong[2]

[1,2]Ho Chi Minh University of Science

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)
for i ← 1 to N do
    minIndex ← i
    for j ← i + 1 to N do
        if a_minIndex > a_j then
            minIndex ← j
        end
    end
    swap(a_minIndex, a_i)
end
```

Complexity

In each stage, the algorithm takes the smallest unsorted element. In initial stage, the data has N elements, so the algorithm needs $n-1$ steps and moves the current smallest number to the top of the data. After this stage, number of unsorted elements remains to $n-1$, and this number will be reduced until it equal to one. Conclusion, the number of comparisions:

$$(n-1) + (n-2) + ... + 2 + 1 = \frac{n(n-1)}{2}$$

So the complexity of this algorithm is $O(N^2)$.
Best case time complexity: $O(N^2)$
Average 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)
for $i \leftarrow 2$ *to* N **do**
  $k \leftarrow i - 1$
  $key \leftarrow a_i$
  while $a_k > key$ *and* $k \geq 0$ **do**
    $a_{k+1} \leftarrow a_k$
    $k \leftarrow k - 1$
  end
  $a_{k+1} \leftarrow key$
end

Complexity

To evaluate the complexity of Insertion Sort, the number of comparisions and assignments need to be calculated. At each stage, the algorithm find a suitable position for an unsorted element. The finding part requires at most $O(N)$ comparisons and assignments, and the best case is $O(1)$ if the unsorted element is at the end of sorted section.
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)
for $i \leftarrow N$ *to* 1 **do**
  $isSwap \leftarrow False$
  for $j \leftarrow 1$ *to* $i - 1$ **do**
    if $a_j > a_{j+1}$ **then**
      $isSwap \leftarrow True$
      swap(a_j, a_{j+1})
    end
  end
  if $isSwap = False$ **then**
    stop algorithm
  end
end

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

Complexity

The number of comparisons is $n-1$ is first stage, and so on $n-2$, $n-1$, ..., 1. The number of swaps in the worst case is same as the number of comparisions and best case is 0 (sorted data). So on all best case and worst case the complexity of this algorithm is $O(N^2)$. But in reality, the algorithm runs faster in sorted array than in other data.
Best case time complexity: $O(N^2)$
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

Complexity

The complexity of the Shaker sort algorithm is exactly the same as that of Bubble sort, the only difference is that the running time is faster because the number of steps is less. In general, when talking about the complexity, it's still $O(N^2)$ in any case.
Best case time complexity: $O(N^2)$
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.

Algorithm 2.4: Shaker Sort

Input: $a_1, a_2, ..., a_N$

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

```
left ← 0
right ← N − 1
k ← 0
for i ← left to right do
    // phase 1
    isSwap ← False
    for j ← left to right − 1 do
        if a_j > a_{j+1} then
            isSwap ← True
            swap(a_j, a_{j+1})
            k ← j
        end
    end
    if isSwap = False then
        stop algorithm
    end
    right ← k
    // phase 2
    isSwap ← False
    for j ← right to left + 1 do
        if a_j < a_{j−1} then
            isSwap ← True
            swap(a_j, a_{j−1})
            k ← j
        end
    end
    if isSwap = False then
        stop algorithm
    end
    left ← k
end
```

Algorithm 2.5: Shell Sort

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

```
interval ← N/2
while interval > 0 do
    for i ← interval to N do
        temp ← a_i
        j ← i
        while interval ≤ j and a_{j−interval} > temp do
            a_j ← a_{j−interval}
            j ← j − interval
        end
    end
    a_j ← temp
    interval ← interval/2
end
```

Pseudo code

Complexity

The complexity of this algorithm bases on M, which is length of the interval between the elements. The evaluation of the complexity of Shell Sort is still an open problem, with no solution.
Worst case space complexity: $O(M)$

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

Complexity

In first stage, the complexity of the max-heap construction is $O(N)$. In second stage, the complexity is $O(N \log N)$ because the algorithm needs to take N heap adjustment, each adjustment takes at most $O(\log N)$.
Best case time complexity: $O(N)$
Average case time complexity: $O(N \log N)$
Worst case time complexity: $O(N \log N)$
Worst case space complexity: $O(1)$

Algorithm 2.6: Heap Sort

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

```
Function HeapRebuild(a, pos, N):
    while 2·pos + 1 <= N do
        j = 2·pos + 1
        if j < N then
            if a_j < a_{j+1} then
                j ← j + 1
            end
        end
        if a_pos ≥ a_j then
            return
        end
        swap(a_pos, a_j)
        pos ← j
    end
Function HeapConstruct(a, N):
    for i ← N/2 to 0 do
        HeapRebuild(a, i, n)
    end
Function HeapSort(a, N):
    HeapConstruct(a, N)
    r ← N
    while r > 0 do
        swap(a_1, a_N)
        HeapRebuild(a, 1, r)
        r ← r − 1
    end
```

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)

```
Function Merge(a, first, mid, last):
    n1 ← mid − first + 1
    n2 ← last − mid
    L ← a_first, a_first+1, ..., a_mid
    R ← a_mid+1, a_mid+2, ..., a_last
    // merge
    i ← 0
    j ← 0
    k ← first
    while i < n1 and j < n2 do
        if L_i < R_j then
            a_k ← L_i
            i ← i + 1
        else
            a_k ← R_j
            j ← j + 1
        end
        k ← k + 1
    end
    while j < n2 do
        a_k ← R_j
        k ← k + 1
        j ← j + 1
    end
    while i < n1 do
        a_k ← L_i
        k ← k + 1
        i ← i + 1
    end
Function MergeSort(a, first, last):
    if first < last then
        mid ← first + (last − first)/2
        MergeSort(a, first, mid)
        MergeSort(a, mid + 1, last)
        Merge(a, first, mid, last)
    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)
Function Partition(a, l, r):
    pivot ← a_{(l+r)/2}
    while l ≤ r do
        while a_l < pivot do
            l ← l + 1
        end
        while a_r > pivot do
            r ← r − 1
        end
        if l ≤ r then
            swap(a_l, a_r)
            l ← l + 1
            r ← r − 1
        end
    end
    return l
Function QuickSort(a, l, r):
    if l < r then
        mid ← Partition(a, l, r)
        QuickSort(a, l, mid − 1)
        QuickSort(a, mid, r)
    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: $O(\log N)$

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)
maxVal ← a_0
for i ← 1 to N do
    if a_i > maxVal then
        maxVal ← a_i
    end
end
count ← [0] * (maxVal + 1) // initialize 0-value counting array
foreach u ∈ a do
    count_u ← count_u + 1
end
// restore the elements to array
idx ← 0
for i ← 0 to maxVal do
    while count_i > 0 do
        a_idx ← i
        idx ← idx + 1
        count_i ← count_i − 1
    end
end
```

Complexity

The complexity of Counting Sort depends on the difference of the largest and smallest element of the input array.
Best case time complexity: $O(N + \mathrm{K})$
Average case time complexity: $O(N + \mathrm{K})$
Worst case time complexity: $O(N + \mathrm{K})$
Worst case space complexity: $O(\mathrm{K})$
where K is the difference of the largest and smallest element of the 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)
maxVal ← a_0
for i ← 1 to N do
    if a_i > maxVal then
        maxVal ← a_i
    end
end
exp ← 1
while maxVal / exp > 0 do
    digit ← array with of N elements
    for i ← 1 to N do
        // get corresponding digit
        digit_i ← a_i / exp  mod 10
    end
    // do counting sort of a[] according to the digit represented by exp
    CountingSort(a, n, digit)
    exp ← exp · 10
end
```

Complexity

In this paper, the algorithm is implemented in base 10. So the complexity is $O(N \log_{10} M)$, where Max is maximum value of input array.
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, equal to $(\log_{10} M)$.

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]
m ← d · n
// stage 2
minVal ← a_1
maxIndex ← 1
for i ← 1 to N do
    if a_i < minVal then
        minVal ← a_i
    end
    if a_maxIndex < a_i then
        maxIndex ← i
    end
end
if a_maxIndex == minVal then
    stop algorithm
end
// classify elements into corresponding partition
c ← (m − 1) / (a_maxIndex − minVal)
for i ← 1 to N do
    cls ← c · (a_i − minVal)
    L_cls ← L_cls + 1
end
for i ← 1 to m do
    L_i ← L_i + L_{i−1}
end
// stage 3
swap(a_maxIndex, a_1) nmove ← 0
j ← 0
k ← m − 1
t ← 0
while nmove < N do
    while j > L_k − 1 do
        j ← j + 1
        k ← c · (a_j − minVal)
    end
    flash ← a_j
    if k ¡ 0 then
        break
    end
    while j ≠ L_k do
        k ← c · (flash − minVal)
        L_k ← L_k − 1
        t ← L_k
        hold ← a_t
        a_t ← flash
        flash ← hold
        nmove ← nmove + 1
    end
    // stage 4
    InsertionSort(a, n)
end
```

Complexity

The time complexity of Flash Sort base on choosing value m. On average, there are $\frac{N}{m}$ elements in each partition, so the complexity for sorting each partition is $O(\frac{N^2}{m^2})$. The total time complexity is $O(\frac{N^2}{m^2} \cdot m)$ for m partitions. For example, if m is chosen proportional to $\sqrt{N}$, the time complexity is $O(N^{3/2})$. In this paper, I chose 0.43 for m, the average complexity will be approximately $O(N)$.

Space complexity: $O(m)$

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.

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

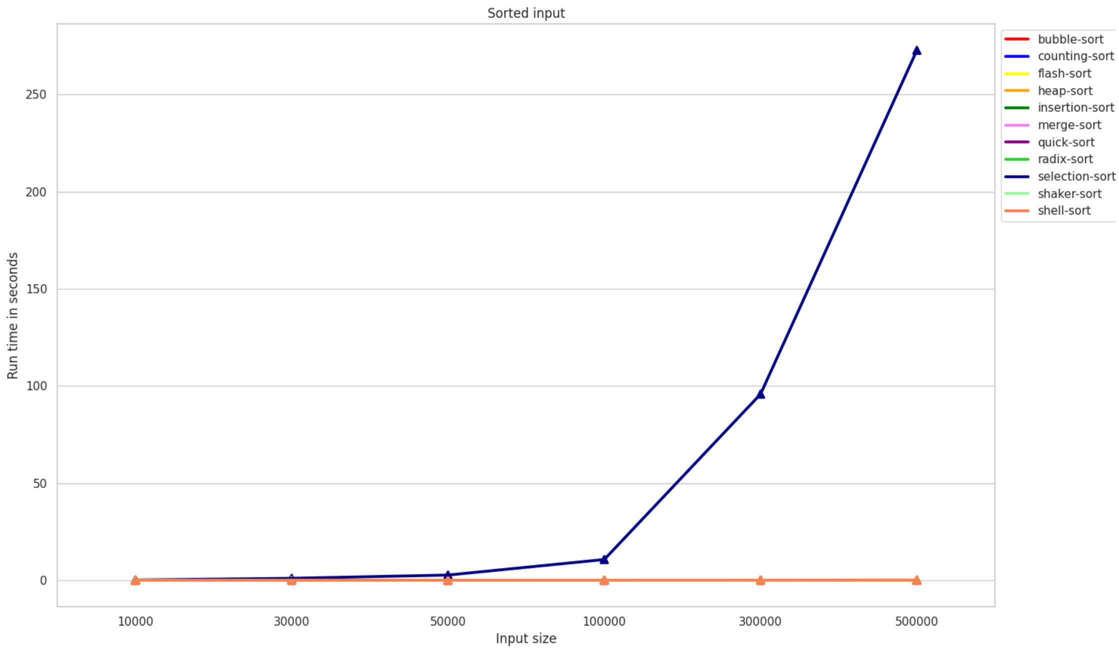
Table 2: Experimental results on nearly sorted data

Data Order: Reverse Sorted data						
Data size	10,000		30,000		50,000	
Result	Time (ms)	Comparision	Time (ms)	Comparision	Time (ms)	Comparision
Selection	114.984	50005001	1031.539	450015001	2892.041	1250025001
Insertion	130.475	100009999	1284.959	900029999	3295.286	2500049999
Bubble	313.577	100020000	2865.468	900060000	7252.795	2500100000
Shaker	315.843	100015000	3064.203	900045000	8452.464	2500075000
Shell	0.796	355157	2.491	1164030	6.851	2144607
Heap	1.503	476739	5.202	1622791	10.956	2848016
Merge	1.118	411833	3.672	1353961	6.06	2351433
Quick	0.586	203608	1.744	657224	2.843	1134456
Counting	0.098	60003	0.307	180003	0.602	300003
Radix	1.44	170106	5.409	630132	7.315	1050132
Flash	0.384	86006	1.197	258006	1.951	430006
Data size	100,000		300,000		500,000	
Result	Time (ms)	Comparision	Time (ms)	Comparision	Time (ms)	Comparision
Selection	10639.041	5000050001	94140.473	45000150001	254863.245	125000250001
Insertion	12228.275	10000099999	104455.571	90000299999	288450.598	250000499999
Bubble	29184.735	10000200000	245922.443	90000600000	678899.144	250001000000
Shaker	30121.16	10000150000	261173.905	90000450000	745604.906	250000750000
Shell	9.091	4589168	31.2	14901826	49.912	25357556
Heap	18.817	6087452	58.036	20187386	95.207	35135730
Merge	13.076	4952873	40.821	16029865	71.813	27643913
Quick	4.969	2368920	15.282	7575704	27.513	12975704
Counting	0.984	600003	3.02	1800003	5.921	3000003
Radix	12.885	2100132	40.176	7500158	70.372	12500158
Flash	3.859	860006	11.693	2580006	19.106	4300006

Table 3: Experimental results on reverse sorted data

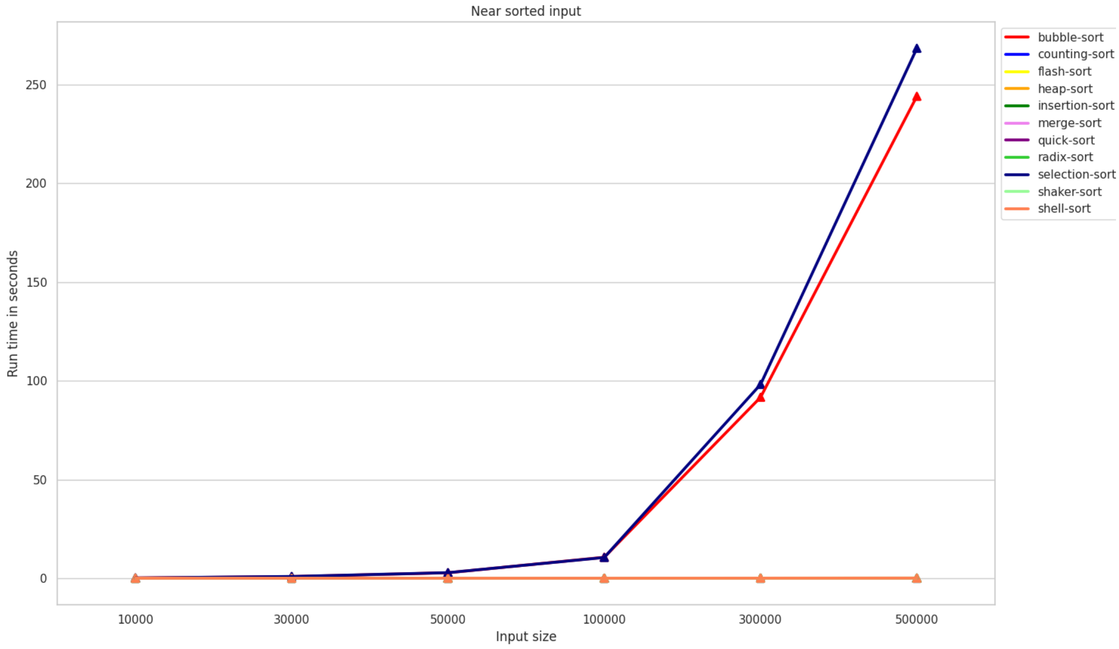
Data Order: Randomized data						
Data size	10,000		30,000		50,000	
Result	Time (ms)	Comparision	Time (ms)	Comparision	Time (ms)	Comparision
Selection	105.579	50005001	939.037	450015001	2628.237	1250025001
Insertion	58.019	50154899	527.028	450626857	1472.483	1252137825
Bubble	313.057	100014960	2911.841	900031101	8322.806	2499993072
Shaker	228.12	66809284	2131.684	600367899	6039.776	1666554000
Shell	1.773	509398	6.287	1854582	11.558	3789217
Heap	1.746	497238	5.839	1681366	13.595	2951638
Merge	1.593	463289	5.421	1528817	9.442	2664667
Quick	1.345	286986	4.395	933275	8.178	1611537
Counting	0.172	60001	0.549	180003	0.895	300001
Radix	0.919	170106	3.41	630132	6.978	1050132
Flash	1.746	497238	5.839	1681366	13.595	2951638
Data size	100,000		300,000		500,000	
Result	Time (ms)	Comparision	Time (ms)	Comparision	Time (ms)	Comparision
Selection	10471.188	5000050001	96069.631	45000150001	263963.237	125000250001
Insertion	5785.636	4977810633	53194.09	44944423565	147202.734	125106226143
Bubble	33661.256	9999933745	308169.164	89998878657	846705.019	250000620545
Shaker	24580.249	6650493563	228078.787	59935751830	620102.489	166715416590
Shell	27.963	8398740	86.959	29933626	172.871	58608607
Heap	22.234	6305394	77.598	20798645	149.053	36121064
Merge	18.848	5629341	61.769	18297818	106.604	31545308
Quick	16.023	3507410	50.365	11317513	87.032	19597319
Counting	1.88	600003	5.649	1800001	18.419	3000003
Radix	10.899	2100132	42.148	7500158	65.644	12500158
Flash	4.737	726414	16.351	2174658	57.569	3681221

Table 4: Experimental results on randomized data



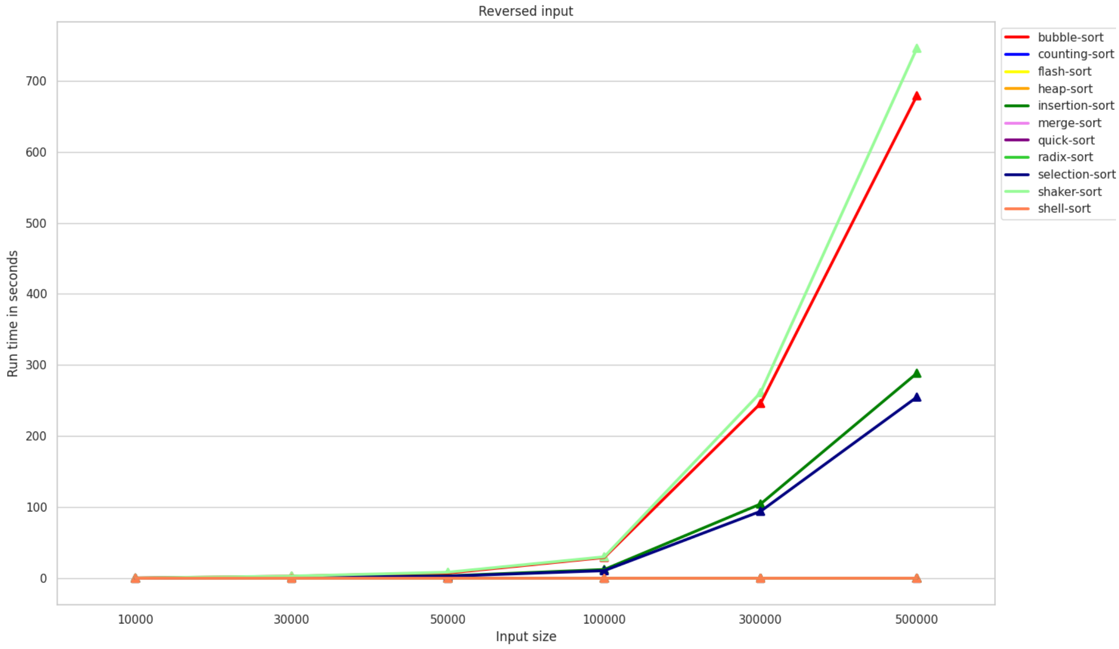
Note: Running sorting algorithms on sorted input data, almost all algorithms recognized that the data had been sorted except Selection Sort. So in the figure, the line of Selection Sort is significantly higher than the others.

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



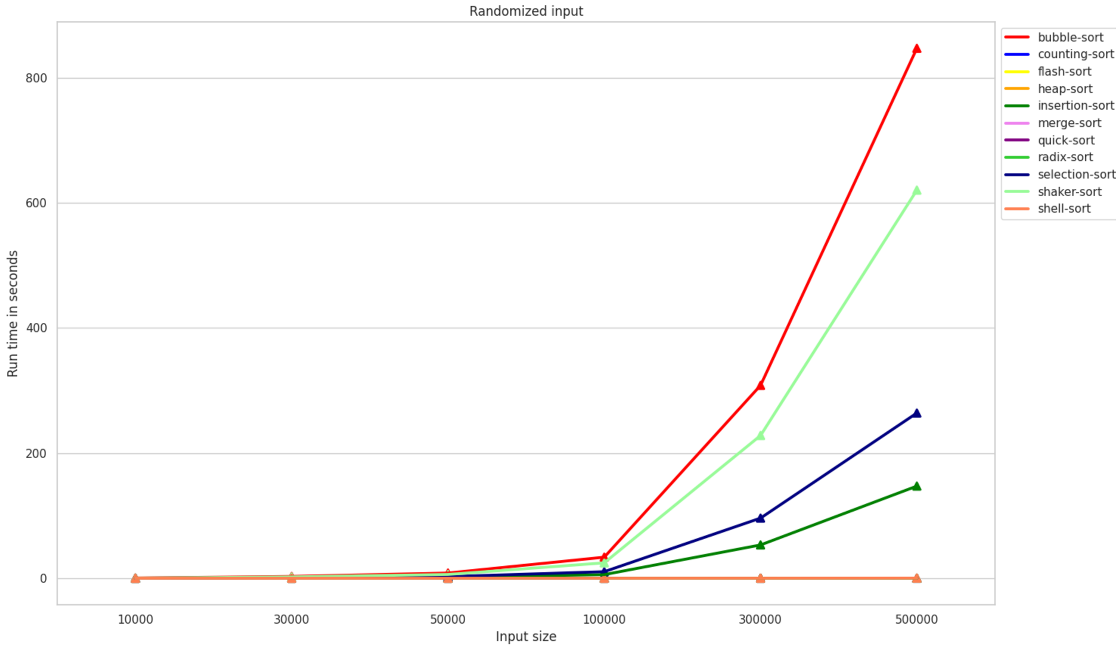
Note: In the figure, the lines of Selection Sort and Bubble Sort are higher than the others. With this data, Selection Sort worked better than Bubble Sort.

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



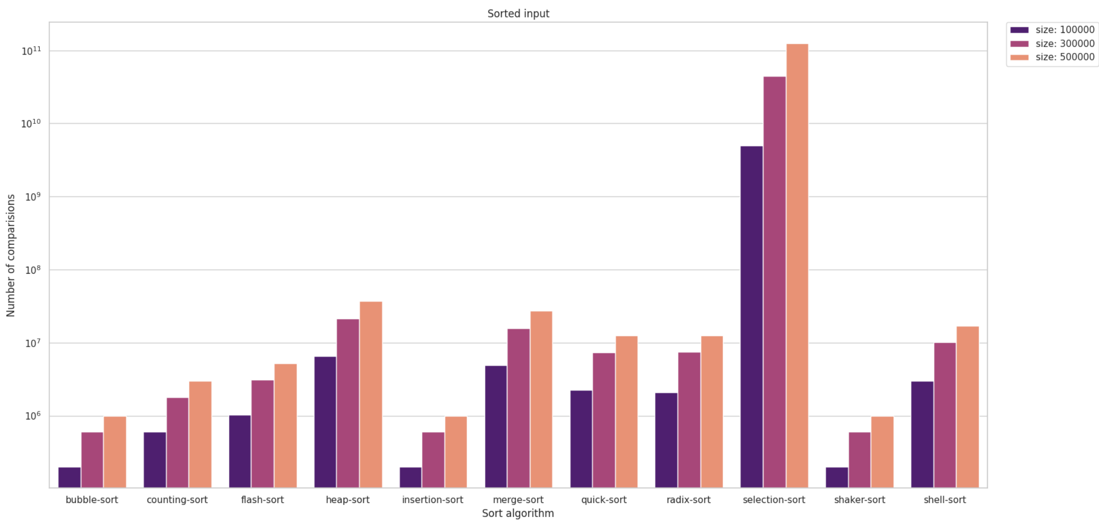
Note: Shaker and Bubble Sort have the line quite higher when Insertion Sort and Selection Sort are lower. Those show that Insertion and Selection Sort worked better than Shaker and Bubble Sort on reverse sorted data but not good enough when comparing with other algorithms.

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



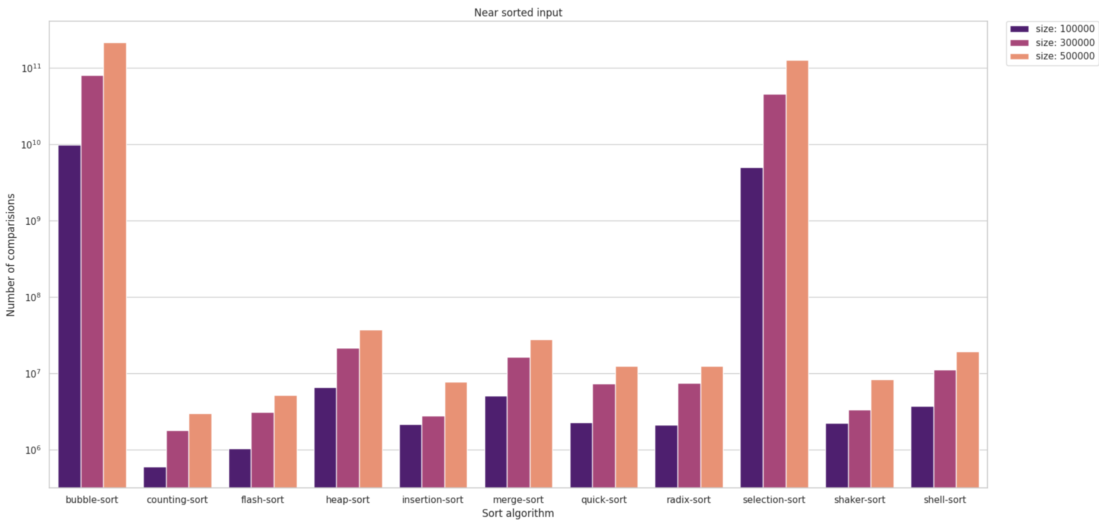
Note: Bubble and Shaker Sort have the worst running time on randomized data. Selection Sort is a little higher than Insertion Sort. Insertion Sort had proofed that it is the most stable algorithm of all simple sorting algorithm. In other that, those advanced sorting algorithms always work well.

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



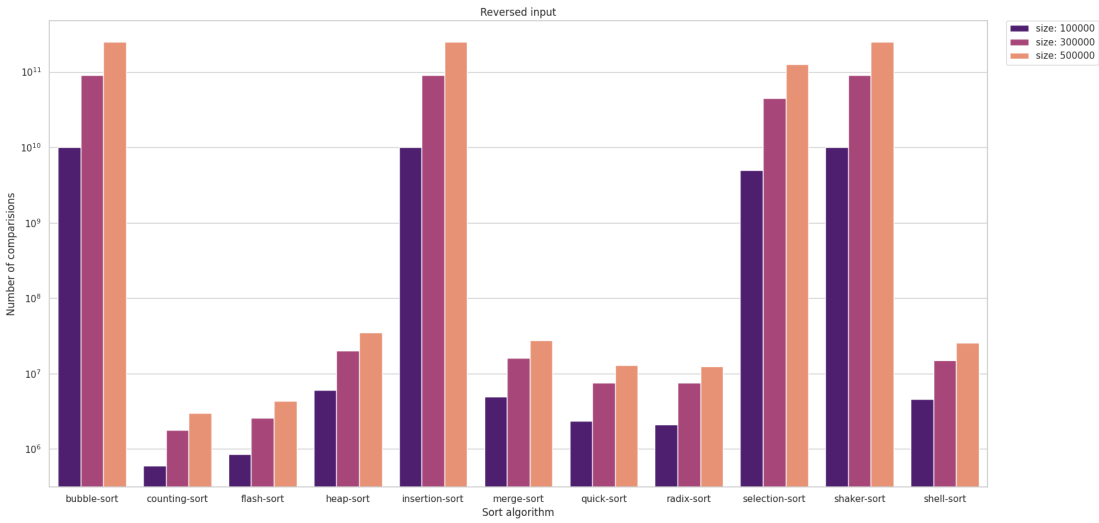
Note: Selection Sort has the most number of comparisons. Bubble, Shaker and Insertion Sort have the smallest comparisions since those algorithms can recognize the input data is sorted or not.

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



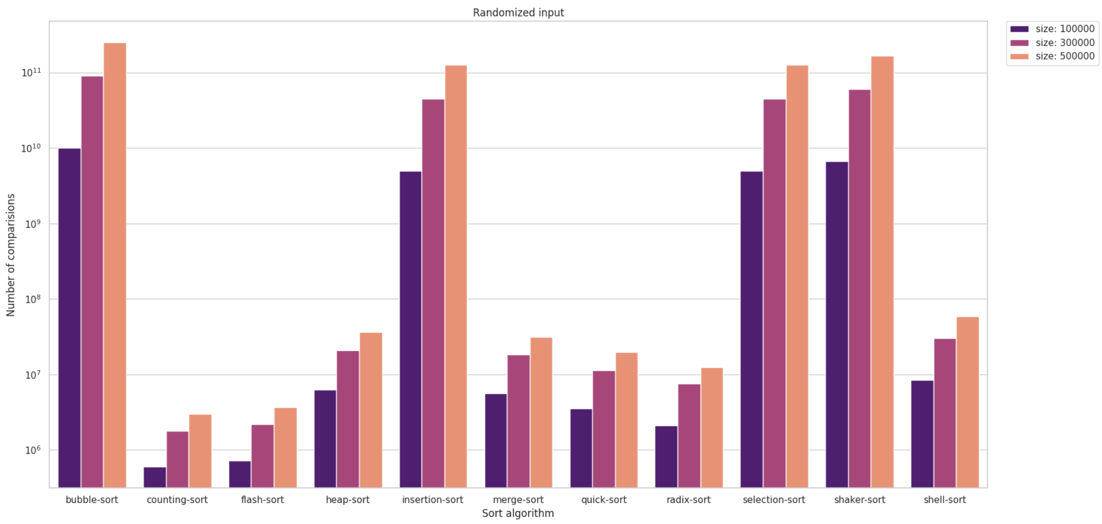
Note: Counting Sort has the least and Bubble Sort has the most number of comparisons. Counting Sort sorts elements by counting the number of occurrences of each unique element in the array, so the number of comparisons is not affected if the order of data changed. Opposite that, Selection, Bubble, Insertion Sort ... do have affected if the order changed, easy to recognize if comparing this figure to the above one.

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



Note: In reverse sorted data, Counting Sort still kept it as the least number of comparisions. When Bubble, Selection, Shaker and Insertion Sort have the most number of comparisions. This behavior can be explained that those above algorithms are simple, those are not implemented to be able to recognize if the data is reverse sorted or not.

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



Note: The behavior of algorithms is the same as reverse sorted data.

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

4. Conclusions

- Overall, the fastest algorithm is Counting Sort, the slowest is Bubble Sort and Selection Sort (Bubble Sort can recognize sorted data but the overall Selection Sort has the average time complexity better than Bubble Sort).
- For sorted data, Bubble Sort and Shaker Sort have fastest running time because the time complexity to know this is a sorted data of the two algorithms above is $O(N)$.
- For nearly sorted data, not consider Counting Sort, Insertion Sort is the best choice due to the small number of comparisions to be performed.
- Selection Sort always gives bad performance (slow running time), so this algorithm should ony be used for cases where the number of elements to be ordered is small.
- Shell Sort, Heap Sort, Merge Sort and Quick Sort have stable performances on all of data types.
- Counting Sort is the fastest, however there is a trade-off by using more memory.
- Flash Sort is not better than Counting Sort, but it is a fast algorithm and consumes very little memory.

4.1. Stable algorithms

In my implementation, the stable algorithms include: Insertion Sort, Bubble Sort, Shaker Sort, Merge Sort, Counting Sort. In these algorithms, Counting Sort has the lowest running time as well as complexity.

4.2. Unstable Algorithms

In my implementation, the unstable algorithms include: Selection Sort, Shell Sort, Heap Sort, Quick Sort, Radix Sort, Flash Sort. Of these algorithms, the fastest running algorithm is still Flash Sort.

5. 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`

6. 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`

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

5. Introduction to Algorithms (Third edition)