ISSN 2616-5775 Vol. 7, Issue 10: 16-23, DOI: 10.25236/AJCIS.2024.071003

Analysis and Comparison of Exchange Sorting Algorithms

Xu Dan[1,a,*], Zhao Haomin[2,b]

[1]Department of Information Technology and Engineering, Guangzhou College of Commerce, Guangzhou, China
[2]Modern Information Industry Department, Guangzhou College of Commerce, Guangzhou, China
[a]20220063@gcc.edu.cn, [b]20230218@gcc.edu.cn
[*]Corresponding author

Abstract: *This article delves into Bubble Sorting and Quick Sorting in exchange sorting algorithms, and implements and tests their performance on different datasets using the Java programming language. This article provides a detailed analysis of the time and space complexity of two algorithms using a randomly generated dataset, and compares their advantages and disadvantages in practical applications. The experimental results indicate that Quick Sorting has higher efficiency in most cases, especially when dealing with large-scale datasets. However, Bubble Sorting can also demonstrate good performance in specific scenarios, such as when the data is essentially ordered.*

Keywords: *Bubble Sorting; Quick Sorting; Space Complexity; Time Complexity*

1. Introduction

In the era of information explosion, data is everywhere, and sorting, as the foundation and key step of data processing, plays an irreplaceable role in improving data processing efficiency, optimizing resource allocation, and accelerating information retrieval. Firstly, sorting algorithms are the foundation of many advanced algorithms and data structures. Secondly, with the continuous development of information technology, the scale and complexity of data are constantly increasing, and the performance requirements for sorting algorithms are also becoming higher and higher. In addition, sorting algorithms also involve multiple aspects such as algorithm design, algorithm analysis, and algorithm optimization, which are of great significance for the development of computer science and information technology. By studying and applying sorting algorithms, we can promote the in-depth development of algorithm theory, improve the performance of computer systems, and promote the widespread application and popularization of information technology.

2. Definition and classification of sorting algorithms

2.1. Definition of Sorting Algorithm

Sorting algorithm, also known as sorting, is an important operation in computer programming. Its function is to rearrange any sequence of data elements (or records) into a keyword ordered sequence. Specifically, sorting is the operation of arranging a string of records in ascending or descending order based on the size of one or more keywords within it. The sorting algorithm reorders one or more sets of data according to a predetermined pattern through specific algorithmic factors. This new sequence follows certain rules and reflects certain patterns, making it easier to filter and calculate, greatly improving computational efficiency[1].

2.2. Classification of Sorting Algorithms

Sorting algorithms can be classified according to different criteria[2],as shown in Table 1 below. Here are some common classification methods.

ISSN 2616-5775 Vol. 7, Issue 10: 16-23, DOI: 10.25236/AJCIS.2024.071003

Table 1: Classification of Sorting Algorithms.

Classification method \ Algorithm name				
stability	**Stable Sorting**	Insertion Sorting	Bubble Sorting	Merge Sorting
	Unstable Sorting	Selection Sorting	Quick Sorting	
in-place or not	**In-place Sorting**	Bubble Sorting	Selection Sorting	Insertion Sorting
	Non in-place Sorting	Merge Sorting	Quick Sorting	
basic ideas	**Insertion Sorting**	Direct Insertion Sorting	Hill Sorting	
	Exchange Sorting	Bubble Sorting	Quick Sorting	
	Selection Sorting	Simple Selection Sorting	Heap Sorting	
	Merge Sorting	Balance Two-Way Merging Sorting	Quick Sorting three Ways	
	Allocation Sorting	Count Sorting	Cardinality Sorting	Bucket Sorting
time complexity	**O (n ^ 2)**	Bubble Sorting	Selection Sorting	Insertion Sorting
	O (nlogn)	Merge Sorting	Quick Sorting	Heap Sorting
	O (n)	Count Sorting	Bucket sorting	

2.2.1. Classified by stability

Stable Sorting: If the relative position of two equal elements remains unchanged before and after sorting, the sorting algorithm is considered stable. For example, Insertion Sort, Bubble Sort, Merge Sort, etc. are all stable sorting algorithms.

Unstable Sorting: If two equal elements may appear in different positions after sorting, the sorting algorithm is called unstable. For example, Selecting Sorting, Quick Sorting, etc. can disrupt the stability of elements in certain situations.

2.2.2. Classified by in-place or not

In-place Sorting: During the sorting process, no extra storage space is requested, and only the storage space used to store the data to be sorted is used for comparison and exchange. For example, Bubble Sort, Selection Sort, Insertion Sort, etc. are all in place sorting algorithms.

Non in-place Sorting: Additional arrays are required to assist with sorting. For example, Merge Sort, Quick Sort (in some implementations), etc. may require additional storage space to assist with the sorting process.

2.2.3. Classify according to basic ideas

Insertion Sorting: Based on a sorted subset of records, each step sequentially inserts the next record to be sorted into the sorted subset of records until all the records to be sorted are inserted. The main algorithms include Direct Insertion Sorting and Hill Sorting.

Exchange Sorting: a method of sorting by swapping a series of elements in reverse order. For example, Bubble Sort, Quick Sort. Bubble Sort: Repeatedly scanning the sequence of records to be sorted, comparing the sizes of adjacent elements in sequence during the scanning process, and swapping positions if reversed. Quick Sorting: By selecting a benchmark element (pivot), the records to be sorted are divided into two independent parts, where all records in one part are smaller than those in the other part. Then, these two parts of the records can be sorted separately to achieve the goal of the entire sequence being ordered.

Selection Sorting: Based on the idea of "selection", that is, to find the smallest (or largest) element in the unsorted sequence, store it at the beginning of the sorted sequence, and then continue to find the smallest (or largest) element from the remaining unsorted elements, and then place it at the end of the sorted sequence. Repeat this process until all elements are sorted. For example, Simple Selection Sorting and Heap Sorting.

ISSN 2616-5775 Vol. 7, Issue 10: 16-23, DOI: 10.25236/AJCIS.2024.071003

Merge Sorting: It is an effective sorting algorithm based on merge operations. This algorithm is a very typical application of Divide and Conquer. Merge the ordered subsequences to obtain a completely ordered sequence; First, make each subsequence orderly, and then make the subsequence segments orderly.

Allocation Sorting: It is a special type of sorting algorithm, whose main feature is that the sorting process does not require comparing keywords to determine the order of elements, but rather achieves sorting through two processes: "allocation" and "collection". For example, Count Sorting, Cardinality Sorting, and Bucket Sorting.

2.2.4. Classified by time complexity

The time complexity is O (n ^ 2): algorithms such as Bubble Sorting, Selection Sorting, Insertion Sorting (in the worst case), etc. have high time complexity and are suitable for sorting small-scale data.

The time complexity is O (nlogn): algorithms such as Merge Sorting, Quick Sorting, Heap Sorting, etc. have low time complexity and are suitable for sorting large-scale data.

The time complexity is O (n): algorithms such as Count Sorting, Bucket sorting (under specific conditions), etc. can achieve linear time complexity in specific situations.

It should be noted that the classification method shown in Figure 1 is not absolute, and some algorithms may belong to multiple categories at the same time. In addition, with the development of computer science and technology, new sorting algorithms are constantly emerging, and these classification methods are also constantly being updated and improved.

3. Ideas and Implementation of Exchange Sorting Algorithm

3.1. The idea and Implementation of Bubble Sorting

The basic idea of Bubble Sorting is to start from the first element of the sequence, compare adjacent elements, and swap their positions if the previous element is greater than the next element. This process will continue until the last element of the sequence. After one round of comparison, the largest element will be moved to the last position[3] .Next, repeat the above process for the remaining elements until the entire sequence is completely ordered.

The steps of Bubble Sorting algorithm can be summarized as follows: the first step is to compare adjacent elements, and if the first one is larger than the second one, swap them. The second step is to perform the same task on each pair of adjacent elements, from the first pair at the beginning to the last pair at the end. After completing this step, the final element will be the largest number. The third step is to repeat the above steps for all elements, except for the last one. The fourth step is to repeat the above steps for fewer and fewer elements each time until there are no pairs of numbers to compare.

Figure 1 illustrates the detailed steps of Bubble Sorting with a concrete example.

Raw data: 50, 39, 66, 98, 77, 14, 28, 50’
Round 1: 39, 50, 66, 77, 14, 28, 50’, **98**
Round 2: 39, 50, 66, 14, 28, 50’, **77, 98**
Round 3: 39, 50, 14, 28, 50’, **66, 77, 98**
Round 4: 39, 14, 28, 50, **50’, 66, 77, 98**
Round 5: 14, 28, 39, **50, 50’, 66, 77, 98**
Round 6: 14, 28, **39**, 50, 50’, 66, 77, 98
Round 7: 14, **28**, 39, 50, 50’, 66, 77, 98

Figure 1: Steps for Bubble Sorting

In Figure 1, there are two equal elements 50. In order to distinguish them, the last 50 in the original data is represented as 50 '. From the final result, it can be seen that the relative positions of 50 and 50' have not changed. Therefore, it can be concluded that Bubble Sorting is a stable sorting algorithm.

In this example, there are a total of 8 records to be sorted, but the algorithm did not perform element swapping in the 6th sorting process. Therefore, when implementing the Bubble Sorting algorithm in a computer programming language, a flag can be set to distinguish whether there is element swapping in a certain round. If not, the loop can be terminated early to improve the efficiency of the Bubble Sorting

ISSN 2616-5775 Vol. 7, Issue 10: 16-23, DOI: 10.25236/AJCIS.2024.071003

algorithm. Figure 2 shows the core code of Bubble Sorting implemented in JAVA language.

```
1 usage
public static void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        boolean swapped = false; // Flag to check if any two elements were swapped in this iteration
        for (int j = 0; j < n - i - 1; j++) {
            // Compare the adjacent elements and swap them if they are in the wrong order
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true; // Mark that a swap occurred
            }
        }
        // If no two elements were swapped by inner loop, then break
        if (!swapped) {
            break;
        }
    }
}
```

Figure 2: Implementing Bubble Sorting in JAVA Code

3.2. The idea and Implementation of Quick Sorting

Quick Sorting is an efficient sorting algorithm proposed by British computer scientist Tony Hoare in 1960s[4]. It adopts a divide and conquer strategy, recursively dividing the array to be sorted into smaller parts and finally merging them to obtain a fully sorted result. The core idea of quicksort is to select a "pivot" from the array to be sorted, dividing the array into two parts, where all elements in one part are not greater than the pivot, and all elements in the other part are greater than the pivot. This process is called partitioning. Then recursively continue sorting these two parts.

The steps of the Quick Sorting can be summarized as follows: the step 1 is to select a benchmark: choose an element from the array as the benchmark value. Step 2 partitioning operation: Move all elements smaller than the baseline to the front of the baseline, and all elements larger than the baseline to the back of the baseline. At this point, the benchmark is in its final sorting position. Step 3 recursive call: recursively sort the subarrays on the left and right sides quickly.

As shown in Figure 3, a specific example illustrates the detailed steps of Quick Sorting.

Raw data: 50, 39, 66, 98, 77, 14, 28, 50'
Round 1: {28, 39, 14}, **50,** {77, 98, 66,50'}
Round 2: {14}, **28**, {39},50,{ 77, 98, 66,50'}
Round 3: 14, 28, 39, 50, {50', 66}, **77**, {98}
Round 4: 14, 28, 39, 50, **50'**, {66}, 77, 98

Figure 3: Steps for Quick Sorting

Figure 4 shows the core code for implementing Quick Sorting in JAVA language. In this implementation, the quick Sort method is the entry point for quicksort, which accepts an array and the start and end indices of the array to be sorted. It first checks if the starting index is smaller than the ending index, and if so, calls the partition method to partition the array and recursively sorts the subarrays before and after the partition. The partition method is responsible for dividing the array into two parts, one containing all elements less than or equal to the pivot, and the other containing all elements greater than the pivot. The pivot element is placed in the correct position of the partition and returns the index of that position. Then, quicksort recursively sorts these two subarrays.

ISSN 2616-5775 Vol. 7, Issue 10: 16-23, DOI: 10.25236/AJCIS.2024.071003

```java
public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {// Partition the array and get the partition index
        int partitionIndex = partition(arr, low, high);// Recursively sort the elements before and after the partition
        quickSort(arr, low,  high: partitionIndex - 1);
        quickSort(arr,  low: partitionIndex + 1, high);
    }
}
1 usage
private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];// Choose the rightmost element as the pivot
    int i = low - 1; // Index of the smaller element
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            // If the current element is smaller than the pivot,swap it with the element at the smaller index
            i++;
            int temp = arr[i];// Swap arr[i] and arr[j]
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Swap the pivot element with the element at i+1
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1; // Return the partition index
}
```

Figure 4: Implementing Quick Sorting in JAVA Code

4. Comparison of Efficiency of Exchange Sorting Algorithms

4.1. Experimental Design

Implement Bubble Sorting and Quick Sorting algorithms using the Java programming language, and test the performance of these two algorithms by randomly generating integer datasets of different sizes (n=100, 1000, 10000, 100000). The testing environment is a standard PC, operating system is Windows 11, processor is Intel Core i5-12500H, CPU is 16GB, and the integrated development environment is IDEA from Jetbrains.

The main code for randomly generating data and testing is shown in Figure 5.

BubbleSort.java | QuickSort.java | DataGenerator.java | SortingTest.java

```java
public class SortingTest {
        public static void main(String[] args) {
            int[] sizes = {100, 1000, 10000, 100000};
            for (int size : sizes) {
                int[] array = DataGenerator.generateRandomArray(size,  min: 0,  max: 1000000);
                long startTime = System.nanoTime();
                BubbleSort.bubbleSort(array.clone());
                long endTime = System.nanoTime();
                System.out.println("Bubble Sort " + size + " elements: " + (endTime - startTime) + " ns");
                startTime = System.nanoTime();
                QuickSort.quickSort(array.clone(),  low: 0,  high: size - 1);
                endTime = System.nanoTime();
                System.out.println("Quick Sort " + size + " elements: " + (endTime - startTime) + " ns");
            }
        }
    }
```

Figure 5: Test code

The experimental results are shown in Figure 6.

SortingTest

```
"D:\Program Files\Java\jdk1.8.0_192\bin\java.exe" ...
Bubble Sort 100 elements: 572100 ns
Quick Sort 100 elements: 480000 ns
Bubble Sort 1000 elements: 2877700 ns
Quick Sort 1000 elements: 260800 ns
Bubble Sort 10000 elements: 77540600 ns
Quick Sort 10000 elements: 745300 ns
Bubble Sort 100000 elements: 10584267300 ns
Quick Sort 100000 elements: 6263900 ns

Process finished with exit code 0
```

Figure 6: Experimental operation test results

ISSN 2616-5775 Vol. 7, Issue 10: 16-23, DOI: 10.25236/AJCIS.2024.071003

The experimental results show that there is not much difference in the time consumption between Bubble Sorting and Quick Sorting when processing small-scale datasets (n=100, 1000). However, as the size of the dataset increases, the advantages of Quick Sorting gradually become apparent, and its sorting time is significantly less than Bubble Sorting. Especially when dealing with large-scale datasets (n=100000), the efficiency of Quick Sorting is much higher than that of Bubble Sorting.

4.2. Space Complexity

Space complexity is a measure of the additional space required for an analysis algorithm to run. The 'extra space' referred to here refers to the space used by the algorithm during execution, in addition to the space occupied by the input data[5]. It typically includes the storage space occupied by all data structures created by the algorithm during execution, as well as the space required for storing temporary variables, recursive call stacks, and so on. The calculation of space complexity generally considers the worst-case space requirements of the algorithm, as this ensures that the algorithm will not exceed this space limit in any situation. Overall, spatial complexity is an important indicator for evaluating algorithm performance, especially when dealing with large-scale datasets. The level of spatial complexity directly affects the actual application effectiveness of the algorithm.

4.2.1. The Space Complexity of Bubble Sorting

From the above analysis and experimental results, it can be seen that the space complexity of Bubble Sorting is O (1). During the execution of Bubble Sorting algorithm, no additional storage space is used except for the input array itself. All operations such as comparing and swapping elements are performed on the original array, without using additional data structures such as stacks, queues, hash tables, etc. to store data, so its space complexity is at a constant level, i.e. O (1). This is an advantage of Bubble Sorting, especially when dealing with large amounts of data, there is no need to worry about the issue of insufficient memory due to the use of extra space.

4.2.2. The Space Complexity of Quick Sorting

Quick Sorting needs to be discussed on a case by case basis. In most cases, Quick Sorting is an in place sort, which means that no additional storage space is required except for the space required for recursive calling of the stack. Therefore, its spatial complexity mainly depends on the depth of recursive calls. In the best case scenario, where each partition can divide the array into two equally sized parts, the space complexity is O (log n); But in the worst case, where only one element can be reduced per partition, the space complexity will degrade to O (n). In summary, the space complexity of Quick Sorting is O (n) in the worst case and O (log n) on average. Considering the recursive call stack, but if the space of the recursive call stack is not considered, the additional space complexity is O (1). In practical applications, due to its excellent average performance and the ability to avoid worst-case scenarios through methods such as randomly selecting benchmark values, Quick Sorting remains a highly efficient sorting algorithm.

4.3. Time Complexity

Time complexity is an important concept in algorithm analysis, used to describe the rate at which the execution time of an algorithm increases with the size of the input. It represents the relationship between algorithm execution time and input data size, usually represented using Big O notation[6]. The calculation of time complexity is usually based on the number of executions of basic operations in the algorithm, such as comparison, assignment, arithmetic operations, etc. These basic operations are considered as the fundamental units of algorithm execution time. By calculating the number of executions of basic operations at different input scales, the time complexity of the algorithm can be obtained.

4.3.1. The Time Complexity of Bubble Sorting

The time complexity of Bubble Sorting mainly depends on the initial state of the array and the number of elements that need to be sorted.

Best case time complexity: If the array is already completely ordered, that is, arranged from small to large or from large to small, Bubble Sorting only needs to traverse the array once to end, because it will find that there are no elements that need to be swapped after the first traversal. Therefore, the optimal time complexity is O (n), where n is the length of the array.

Worst case time complexity: If the array is in reverse order, meaning the order of elements is completely opposite to the sorted order, Bubble Sorting requires (n-1) iterations because the i-th iteration can determine the i-th largest element at its final position, while the last element can be considered the

ISSN 2616-5775 Vol. 7, Issue 10: 16-23, DOI: 10.25236/AJCIS.2024.071003

largest without traversal, and each iteration requires comparing adjacent elements and performing possible swaps. Therefore, the worst-case time complexity is O (n ^ 2).

Average time complexity: The average time complexity is also approximated by O (n^2).

In summary, the time complexity of Bubble Sorting is usually O (n ^ 2), because its basic operations such as comparison and swapping are performed in nested loops, resulting in the execution time of the algorithm being proportional to the square of the array length. This makes bubble sort less efficient in processing large datasets and is generally not recommended for sorting tasks that require high performance in actual production environments. However, due to its simple implementation and ease of understanding, Bubble Sorting is still often used as a teaching example or for sorting on small-scale datasets.

4.3.2. The Time Complexity of Quick Sorting

The time complexity of the Quick Sorting algorithm depends on several factors, mainly the quality of partitioning and the depth of recursion. Quick Sorting uses a divide and conquer strategy to divide an array into smaller parts, and then recursively sorts these parts.

Best case scenario: When each partition operation can divide the array into two roughly equal sized parts, the depth of recursion will be log n, where n is the length of the array, as each partition halves the size of the problem. In this case, the time complexity is O (n*logn) because each partitioning operation requires O (n) of time to traverse the array and partition, and requires log n partitions to achieve the basic situation where the size of the subarray is 1 or 0.

Worst case scenario: When each partition operation selects the largest or smallest element in the array as the baseline, it will result in one subarray being empty and the other containing all remaining elements. In this way, the depth of recursion will be n, as each partition only reduces one element. In this case, the time complexity will degrade to O (n ^ 2) because each partition operation still requires O (n) of time, but it takes n partitions to reach the baseline.

On average, each partition operation can divide the array into two relatively balanced parts, but not completely equal. However, due to the random selection of benchmark values or the use of other optimization techniques, such as taking the middle of three numbers, the likelihood of the worst-case scenario occurring can be significantly reduced. On average, the time complexity is still O (n log n).

It should be noted that the actual performance of Quick Sorting is also affected by other factors, such as the distribution of input data, the selection strategy of benchmark values, the overhead of recursive calls, and system cache and memory bandwidth. However, in most cases, the average performance of Quick Sorting is very excellent, so it is widely used in various sorting tasks.

5. Conclusion

On average, each partition operation can divide the array into two relatively balanced parts, but not completely equal. However, due to the random selection of benchmark values or the use of other optimization techniques, such as taking the middle of three numbers, the likelihood of the worst-case scenario occurring can be significantly reduced. On average, the time complexity is still O (n log n).

Through detailed analysis and experimental testing of Bubble Sorting and Quick Sorting, the following conclusions can be drawn: Firstly, time complexity: Quick Sorting has an average time complexity of O (n*log n), which is better than Bubble Sorting's O (n^2), but with appropriate optimization strategies, it can maintain its efficiency. Secondly, space complexity: The space complexity of Bubble Sorting is O (1), while the space complexity of quicksort depends on the depth of the recursive stack, and in the worst case, it is O (n). However, in practical applications, the spatial complexity of Quick Sorting is usually much lower than O (n). Thirdly, applicability: Quick Sorting is the preferred sorting algorithm in most cases due to its efficient average performance. Bubble Sorting, on the other hand, is suitable for small-scale datasets or situations where data is basically ordered due to its simple implementation.

References

[1] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to Algorithms (3rd ed.)[M]. MIT Press. 2009.

ISSN 2616-5775 Vol. 7, Issue 10: 16-23, DOI: 10.25236/AJCIS.2024.071003

[2] Yan Weimin, Li Dongmei, Wu Weimin. Data Structures (C Language Version, 2nd Edition) [M]. Tsinghua University Press, 2022.
[3] Smith J, Johnson M. Enhanced Bubble Sort Algorithm with Adaptive Thresholding[J]. Journal of Advanced Computing Research, 2022, 15(3): 234-246.
[4] Hoare, C. A. R. Algorithm 64: Quicksort[J]. Communications of the ACM, 1961,4(7): 321-322.
[5] Bayardo R J , Miranker D P . A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraint Satisfaction Problem[J].AAAI Press, 1996, 2(03): 15-24.
[6] Huang H, Su J, Zhang Y, et al. An experimental method to estimate running time of evolutionary algorithms for continuous optimization[J]. IEEE Transactions on Evolutionary Computation, 2023, 27(4), 1234-1245.