MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
V.N. Karazin Kharkiv National University
School of Mathematics and Computer Science
Department of Theoretical and Applied Informatics

Master's Thesis

Calculating the Time Complexity of Algorithms: Substitution Method

Author:
Final year Master's Program student,
group MCS-53
specialty - Computer Sciences and
Information Technologies,
educational program: "Informatics"
XU LI
Supervisor: Anastasiia Morozova
Reviewer: Momot Myroslav, PhD
Adviser: lllia llin

Kharkiv, 2024

Table of Contents

Abstract :

This paper focuses on the replacement method to calculate the time complexity of algorithms, and deeply explores its application and significance in various classic algorithms. As the core of computer science, the time complexity of algorithms affects key aspects such as system performance, resource utilization and response speed. In this context, the replacement method becomes a key tool for accurately analyzing algorithm performance and guiding optimization directions. In the field of sorting algorithms, bubble sort is based on the comparison and exchange of adjacent elements. The time complexity in the best, worst and average cases is $O(\mathrm{n})$, $O(\mathrm{n}^2)$, $O(n^2)$, respectively. It is simple to implement but has poor efficiency in processing large-scale data. It is suitable for small-scale or partially ordered data scenarios and has stability; insertion sort inserts data into the sorted part. The time complexity is similar to bubble sort. It is efficient for small-scale data and performs well for partially ordered data, but its performance is limited for large-scale data; quick sort adopts a divide-and-conquer strategy, with an average time complexity of $O(\mathrm{n}\log n)$, low memory requirements, but the worst case is $O(\mathrm{n}^2)$, sensitive to data distribution, and often used for large-scale data sorting and memory-constrained environments. In the search algorithm, linear search checks elements in sequence, and the time complexity in the best and worst cases is $O(1)$ and $O(\mathrm{n})$, respectively. It is simple and general, but the efficiency of large-scale data search is low, and it is suitable for small-scale unordered data; binary search is based on the characteristics of ordered arrays, and the time complexity is $O(\log n)$. It is efficient but requires ordered data, and has significant advantages in large-scale ordered data search and frequent search scenarios. In terms of recursive algorithms, the Fibonacci sequence recursive algorithm is recursively calculated according to the definition, and the time complexity grows exponentially. Although it is intuitive and simple, redundant calculations cause a sharp increase in large-scale computing costs, and it is often used for small-scale calculations or theoretical teaching; the factorial recursive algorithm is recursive according to the mathematical definition, and the time complexity is $O(\mathrm{n})$. It is easy to understand but may cause stack overflow. It is applicable to small-scale

calculations or sub-modules. It is advisable to choose iterative algorithms for large-scale calculations. Comprehensively comparing the time complexity values of different algorithms, the performance of each algorithm varies with the change of data scale, and the applicable scenarios are different. Bubble sort and insertion sort are suitable for small-scale or specific sorting demand scenarios, quick sort is suitable for large-scale data, linear search is for small-scale unordered data, binary search is used for large-scale ordered data, Fibonacci sequence recursive algorithm serves small-scale computing, and factorial recursive algorithm meets small-scale computing or sub-module requirements. Algorithm optimization strategies are closely linked to the replacement method, such as cocktail sort optimization of bubble sort, randomized quick sort improvement of benchmark selection, and factorial recursive algorithm tail recursion optimization to reduce stack space consumption, all of which use the replacement method to verify the optimization effect. This article systematically analyzes the time complexity of common algorithms with the replacement method, reveals the performance characteristics of the algorithms in all aspects, and lays a solid theoretical foundation for algorithm selection and optimization. However, the research still has limitations. In the future, it is planned to expand the algorithm types, deepen the analysis dimensions, cover more application scenarios, integrate the replacement method with other analysis methods, and continuously improve the level of algorithm performance evaluation and optimization, keep up with the pace of computer science development, inject new vitality into algorithm research, and open up new horizons.

Keywords: algorithm time complexity; substitution method; sorting algorithm; search algorithm; recursive algorithm; algorithm optimization

1. Introduction

1.1 Research background and significance

In today's computer science field, algorithms are the core tools for solving various problems. Their efficiency directly affects many key aspects, such as the performance of computer systems, resource utilization, and the response speed of applications[1]. Algorithm time complexity is an important quantitative indicator for measuring algorithm efficiency. It describes the growth trend of the time required for algorithm execution as the input size increases. For algorithm designers, accurately evaluating the algorithm's time complexity helps optimize the algorithm structure during the design phase and avoid unnecessary waste of computing resources. For developers, understanding the algorithm's time complexity can help them reasonably choose the appropriate algorithm to meet the performance requirements of specific application scenarios. In the field of academic research, the analysis of algorithm time complexity is the basis for in-depth research on algorithm characteristics, comparison of the advantages and disadvantages of different algorithms, and promotion of algorithm innovation[2].

Among the many methods for calculating the time complexity of algorithms, the substitution method occupies an important position with its unique mathematical derivation method. The substitution method can accurately solve the asymptotic representation of the algorithm's time complexity by performing in-depth mathematical abstraction and variable replacement on the recursive or loop structure in the algorithm. This method can not only provide us with a rigorous theoretical analysis of the algorithm's performance, but also serve as an important guide in the algorithm optimization process, helping us determine the optimization direction and evaluate the optimization effect. For example, in large-scale data processing applications, such as database query optimization, massive data sorting and searching, etc., the time complexity of related algorithms can be accurately analyzed by the substitution method, which can effectively improve the overall operating efficiency of the system and reduce processing time and resource consumption. Therefore, in-depth research on the replacement method to calculate the time complexity of algorithms has extremely

important theoretical and practical significance, which is also the core theme of this paper.

1.2 Overview of the Replacement Method

The basic principle of the substitution method is based on the mathematical modeling of the recursive relationship or loop iteration process in the algorithm. For an algorithm with a recursive structure, we first need to determine its recursive equation, which usually describes the relationship between the problem size and the sub-problem size and the time cost required to solve these sub-problems. For example, for the classic Fibonacci sequence recursive algorithm, its recursive equation is $\mathrm{T(n)} = T(\mathrm{n}-1) + T(\mathrm{n}-2) + O(1)$, where $\mathrm{T(n)}$ represents n the time required to calculate the Fibonacci number, and $\mathrm{O(1)}$ represents the time complexity of basic operations (such as addition) [3].

The main steps of the substitution method include: first, make a reasonable guess for the unknown function (such as) in the recursive equation $\mathrm{T(n)}$. This guess is usually based on an intuitive understanding of the algorithm execution process and empirical judgment of the complexity of similar algorithms. Then, substitute the guessed function into the recursive equation to verify whether it meets the boundary conditions of the equation and its consistency throughout the recursive process. If the verification is successful, the guessed function is the asymptotic representation of the time complexity of the algorithm; if it is not satisfied, the guess needs to be adjusted and re-verified. For example, for the above Fibonacci sequence recursive equation, we may first guess $T(\mathrm{n}) = O(2^n)$ and then substitute it into the equation for verification and derivation.

The scope of application of the substitution method is relatively wide, especially for algorithms with obvious recursive structures and relatively simple and clear recursive relationships, such as many divide-and-conquer algorithms (such as merge sort, the recursive part of quick sort), recursive calculation series (such as Fibonacci series, factorial recursive calculation), etc. However, for some complex algorithms, such as those with multiple recursions, nonlinear recursion, or a mixture of recursion and loops, the application of the substitution method may face certain challenges and require more in-depth mathematical analysis and techniques. But overall, as a basic and

important time complexity analysis method, the substitution method provides us with a powerful tool for understanding and optimizing algorithms.

2.Analysis of sorting algorithms based on replacement method

2.1 Bubble sort

2.1.1 Algorithm Principle

Bubble sort is a simple comparison sorting algorithm, and its basic idea is based on comparing and exchanging adjacent elements. For a given array, starting from the first element of the array, two adjacent elements are compared in turn. If their order does not meet the requirements (such as the previous element is greater than the next element in ascending order), the positions of the two elements are swapped. Through such a round of comparison and exchange operations, the largest (or smallest, depending on the sorting order) element in the array will "bubble" to the end of the array. Then, this process is repeated for the remaining unsorted elements until the entire array is sorted. For example, for the array [5, 3, 4, 6, 2], in the first round of comparison, 5 and 3 will be compared first. Since 5 > 3, their positions are swapped to obtain [3, 5, 4, 6, 2]; then 5 and 4 are compared, and after the swap, it becomes [3, 4, 5, 6, 2]; and so on. After the first round, the array becomes [3, 4, 5, 2, 6], and the largest element 6 has reached the end. In subsequent rounds, this operation is repeated for the first n - 1 elements until the entire array is sorted [4].

2.1.2 Time complexity calculation (substitution method)

1. Best case: In the best case, the array is already ordered. At this time, only one round of comparison is needed, and the number of comparisons is n - 1 times (n is the number of array elements), and there is no element exchange operation. Set it as the time complexity function of bubble sort, then $T(n)$ $T(n) = O(n-1) = O(n)$. This is because when the array is ordered, the inner loop will end after only one comparison each time, and the outer loop will execute n - 1 times. Here we mainly rely on the algorithm logic of bubble sort. In the ordered case, the number of comparisons in the inner loop is constant at n - 1, which is the analysis of the best case of bubble sort.

2. Worst case: The worst case occurs when the array is in reverse order. For each round of comparison, n - i comparisons are required (i is the current round), and a total of n - 1 comparisons are required. The total number of comparisons is

$(n-1)+(n-2)+...+1=\sum_{i-1}^{n-1} i$. According to the arithmetic sequence summation formula $\frac{n(n-1)}{2}$, so $T(n)=O(\frac{n(n-1)}{2})=O(n^2)$. This is based on a detailed analysis of the bubble sort in the reverse order to derive the worst case time complexity.

3. Average case: On average, the disorder of the array is between the best and worst cases. It can be assumed that all permutations of the elements in the array have equal probability of appearing. For any two elements, there is a 50% probability that their order in the final ordered array is opposite, that is, they need to be swapped. So the average number of swaps is about $\frac{n(n-1)}{4}$. The number of comparisons is of the same order as the number of swaps, so the average time complexity is also $O(n^2)$.

2.1.3 Code Implementation The following is the Python code implementation of bubble sort:

```
def bubble_sort(arr):
n = len(arr)
# The outer loop controls the comparison rounds
for i in range(n - 1):  # Here the outer loop is executed n - 1 times, corresponding to the O(n) part of the time complexity.
# The inner loop compares and exchanges adjacent elements
for j in range(n - 1 - i):  # The number of comparisons in each round gradually decreases, corresponding to the (n - i) comparisons in the time complexity derivation.
if arr[j] > arr[j + 1]:
arr[j], arr[j + 1] = arr[j + 1], arr[j]
return arr
```

Fig.2.1 Python code implementation of bubble sort

2.1.4 Advantages and Disadvantages Analysis

1. Advantages: The algorithm logic of bubble sort is simple and intuitive, easy to understand and implement. Its code structure is clear, and even beginners can quickly grasp its principles and writing methods. This makes it have certain advantages in some simple sorting scenarios or teaching demonstrations, and can help learners quickly

establish the basic concepts of sorting algorithms. For example, it is often used in algorithm teaching in introductory computer science courses [5].

Bubble sort is a stable sorting algorithm. During the sorting process, if two elements are equal, their relative order will not be changed. This is very important in some application scenarios that require the order of equal elements, for example, when sorting a task list containing tasks of the same priority, the original order of the tasks of the same priority can be preserved [6].

2. Disadvantages: From the perspective of time complexity, the time complexity of bubble sort is 2.0 in both the worst and average cases $O(n^2)$, which makes it inefficient when processing large-scale data. As the data size n increases, the execution time of the algorithm will increase rapidly. Compared with some efficient sorting algorithms with a time complexity of 2.0 $O(nlogn)$ (such as merge sort and quick sort), it is obviously at a disadvantage in large-scale data sorting tasks. For example, it is rarely used in scenarios such as data analysis in the field of big data processing and database sorting operations [4].

In each round of comparison, bubble sort will still perform a large number of unnecessary comparison operations even if the array is already partially ordered. For example, when the first half of the array is already ordered, the comparison of the elements in the second half will still be performed in a complete round. There is no effective mechanism to terminate the comparison process early, which further reduces the efficiency of the algorithm [6].

2.2 Insertion Sort

2.2.1 Algorithm Principle

Insertion sort is a simple and intuitive sorting algorithm. Its basic idea is to insert a data into the appropriate position of an already sorted array, so as to gradually build a complete ordered array. In each round of iteration, it takes an element from the data to be sorted, and then scans from the back to the front in the sorted part of the array to find the appropriate insertion position of the element, and then moves the elements after this position backward one position in turn to make room for the inserted element. For example, for the given array [5, 2, 4, 6, 1, 3], initially the first element 5 is regarded as

the sorted part, and then the second element 2 is taken. Since 2 < 5, 5 is moved backward one position and 2 is inserted into the first position. At this time, the array becomes [2, 5, 4, 6, 1, 3]. Then the third element 4 is processed and inserted into the appropriate position in the sorted [2, 5], and so on, until the entire array is sorted [7].

2.2.2 Time complexity calculation (substitution method)

Assume that there are n elements in the array. The time complexity of insertion sort depends mainly on the number of comparisons and moves of elements. In the best case, that is, the array is already ordered, each insertion operation only requires one comparison and no element movement. The time complexity is $O(n)$. This is because for an already ordered array, the outer loop executes n - 1 times, but the inner loop only performs one comparison operation each time and ends. In the worst case, the array is in reverse order, such as [n, n - 1, n - 2, ..., 1]. For the i-th element, it is necessary to compare and move with the previous i - 1 elements, and the total number of comparisons is $T(n)=\sum_{i=2}^{n}(i-1)$. According to the formula for summing arithmetic progressions $\frac{n(n-1)}{2}$, the time complexity is $O(n^2)$. Here we use the substitution method for analysis. Assume that $T(n)=an^2+bn+c$ (a, b, c are constants), substitute them into the recursive formula or determine the coefficients by analyzing the law of the number of comparisons, and verify that the time complexity is $O(n^2)$, which is the same as the time complexity of bubble sort in the worst case. However, on average, the performance of insertion sort is slightly better than that of bubble sort, because bubble sort needs to compare and exchange adjacent elements in each round, while insertion sort only moves data when necessary [8].

2.2.3 Code Implementation The following is a Python code example for insertion sort:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
    # The inner loop is used to find the insertion position in the sorted part
and move the element
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

Fig.2.2 Python code example for insertion sort

In the above code, the outer loop controls the position of the element to be inserted, starting from the second element (index 1), and is executed n - 1 times in total. The inner loop is used to find the insertion position in the sorted part and move the elements. In the worst case, the inner loop needs to be executed i - 1 times for the i-th element, which results in an overall worst-case time complexity of $O(n^2)$. The data movement operation is reflected in the inner loop. When the insertion position is found, the elements after the insertion position need to be moved back one position in turn, which may involve multiple data movements, especially when the array is reversed.

2.2.4 Advantages and Disadvantages Analysis

1. Advantages: The advantages of insertion sort are obvious. For small-scale data, its simple and direct implementation makes it more efficient in actual operation and the code execution speed is fast. In the case of partially ordered data, since only a small number of unordered elements need to be inserted, performance can also be better guaranteed. For example, when processing some real-time data collection systems, the newly collected data may be only partially out of order. Insertion sort can quickly integrate new data into the sorted data sequence.

2. Disadvantages: Its worst-case time complexity is $O(n^2)$, which will significantly reduce performance when processing large-scale data. Compared with some efficient sorting algorithms (such as quick sort and merge sort), the efficiency gap is obvious

when the amount of data is large. Moreover, due to the large number of data movement operations, especially when the array is in reverse order, a large amount of computing resources may be consumed for data movement, which also affects the overall performance of the algorithm to a certain extent [9].

2.3 Quick sort

2.3.1 Algorithm Principle

Quick sort is an efficient sorting algorithm based on the divide-and-conquer strategy. Its core idea is to select a base element and divide the array into two sub-arrays, the left sub-array and the right sub-array, so that the elements in the left sub-array are less than or equal to the base element, and the elements in the right sub-array are greater than or equal to the base element. Then, the quick sort algorithm is recursively applied to the left and right sub-arrays respectively until the length of the sub-array is less than or equal to 1, at which point the entire array is sorted [10]. When selecting the base element, common strategies include selecting the first element, the last element, or a random element of the array. Taking the first element of the array as an example, the process of dividing the left and right sub-arrays is as follows: starting from the second element of the array, move the elements that are less than the base element to the left, and the elements that are greater than the base element to the right, and finally determine the correct position of the base element, thereby completing a division operation. For example, for the array [5, 3, 8, 4, 9, 1], if 5 is selected as the base element, after partitioning, we get [3, 4, 1, 5, 8, 9], and then recursively perform quick sort operations on [3, 4, 1] and [8, 9] respectively.

2.3.2 Time complexity calculation (substitution method)

Suppose the time complexity of the quick sort algorithm is $T(n)$, where n is the length of the array to be sorted. On average, each partition can roughly divide the array into two sub-arrays of equal length. Then, the time complexity of the partition operation is $O(n)$, because it is necessary to traverse the array once to determine the position of the element. The time complexity of recursively sorting the two sub-arrays is respectively

$T(\frac{n}{2})$. According to the divide-and-conquer strategy, the recursive formula for the time complexity in the average case is:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Apply the substitution method, assuming that $T(n) = O(n \log n)$, substitute it into the recursive formula:

$$\begin{aligned} T(n) &= 2O(\frac{n}{2}\log\frac{n}{2}) + O(n) \\ &= 2 * \frac{n}{2}(\log n - \log 2) + O(n) \\ &= n(\log n - 1) + O(n) \\ &= O(n \log n) \end{aligned}$$

In the worst case, for example, when the array is already in order or in reverse order, each partition will result in an empty subarray and a subarray of length n - 1. The time complexity recursive formula is:

$$T(n) = T(n-1) + O(n)$$

By expanding the recursive formula, we can get $T(n) = O(n^2)$: For example, for the ordered array [1, 2, 3, 4, 5], if 1 is selected as the base element, the left subarray is empty after the first partition, and the right subarray is [2, 3, 4, 5]. The subsequent recursive process is similar, resulting in the time complexity degenerating to $O(n^2)$ [11].

2.3.3 Code Implementation The following is the Python code implementation of quick sort:

```
import random

def quick_sort(arr):
    def partition(arr, low, high):
        # Randomly select the pivot element and swap it with the first element
        pivot_index = random.randint(low, high)
        arr[low], arr[pivot_index] = arr[pivot_index], arr[low]
        pivot = arr[low]
        i = low + 1
        j = high
        while True:
            # Find the first element greater than the pivot from left to right
            while i <= j and arr[i] <= pivot:
                i += 1
            # Find the first element smaller than the pivot from right to left
            while i <= j and arr[j] > pivot:
                j -= 1
            if i <= j:
# Swap two elements so that the left side is less than or equal to the reference and the
right side is greater than or equal to the reference
                arr[i], arr[j] = arr[j], arr[i]
            else:
                break
        # Put the datum element in the correct position
        arr[low], arr[j] = arr[j], arr[low]
        return j

    def quick_sort_helper(arr, low, high):
        if low < high:
            # Divide the operation to obtain the final position of the base element
            pivot_index = partition(arr, low, high)
            # Recursively sort the left and right subarrays
            quick_sort_helper(arr, low, pivot_index - 1)
            quick_sort_helper(arr, pivot_index + 1, high)

    quick_sort_helper(arr, 0, len(arr) - 1)
    return arr
```

Fig.2.4 Python code implementation of quick sort

In the above code, the `quick_sort` function is an external interface function, and the `partition` function implements the partition operation. It randomly selects the base element and swaps it to the beginning of the array, and then uses the double pointer method to divide the array into left and right parts. The `quick_sort_helper` function recursively sorts the left and right subarrays. Recursive calls are the key to implementing the divide-and-conquer idea. Each partition will generate two sub-problems and process them recursively. The time complexity of the partition operation is $O(n)$, and the recursive depth is on average $O(\log n)$. Therefore, the overall time complexity is on average $O(n\log n)$. Randomly selecting base elements helps reduce the probability of the worst case and improve the average performance of the algorithm .

2.3.4 Advantages and Disadvantages Analysis

1. Advantages: Quick sort has significant advantages . Its average performance is very good, and its time complexity is $O(n\log n)$. In practical applications, it is more efficient in sorting large-scale data. It is usually more efficient than some simple sorting algorithms (such as bubble sort and insertion sort).) much faster. It is an in-place sorting algorithm that does not require a large amount of additional storage space and only requires a small amount of auxiliary space for the recursive call stack.

2. Disadvantages: In the worst case, when the data is already in order or reverse order, the time complexity will degenerate to $O(n^2)$, which may cause a sharp drop in sorting performance. Due to its recursive implementation, when the data scale is very large, the recursive depth may be very deep, which may easily lead to stack overflow problems. In addition, quick sort is sensitive to the initial distribution of data. If the data distribution is uneven, it may affect the performance of the algorithm. However, this problem can be alleviated to a certain extent by optimizing strategies such as randomly selecting benchmark elements [12].

3. Analysis of search algorithm based on replacement method

3.1 Linear Search

3.1.1 Algorithm Principle

Linear search is the most basic and intuitive search algorithm. It starts from the beginning of the array, checks each element in the array in order, and compares the current element with the target element. If the two are equal, it means that the target element has been found; if no matching element is found after traversing the entire array, it means that the target element does not exist in the array. For example, given an integer array arr = [3, 7, 1, 9, 5], to search for the target element 7, the linear search will first check arr[0] = 3, and if it finds no match, it will continue to check arr[1] = 7, at which point the target element is found and the search stops. This element-by-element comparison process constitutes the core mechanism of linear search [13].

3.1.2 Time complexity calculation (substitution method)

Let the size of the array be n, that is, there are n elements in the array. In the best case, the target element happens to be the first element of the array, and only one comparison operation is required to find the target element, so the time complexity is $O(1)$. This can be simply analyzed by substitution method. Let $T(n)$ represent the time complexity of the linear search algorithm on an array of size n. When n = 1 (the best case), $T(1) = 1$, which meets $O(1)$ the time complexity representation.

In the worst case, the target element is at the last position of the array or does not exist in the array at all. In this case, the entire array needs to be traversed, and a total of n comparison operations are performed. Using the substitution method, it is assumed that for an array of size n, its time complexity $T(n)$ satisfies the recursive relation

$T(n) = T(n-1) + 1$,in $T(1) = 1$.

By continuously expanding this recursive formula, we can get:

$T(n) = T(n-1) + 1 = T(n-2) + 1 + 1 = \ldots = T(1) + (n-1) = n$, so the worst-case time complexity is $O(n)$, which clearly shows that linear search is linear in the array size in the worst case [14].

3.1.3 Code Implementation

Here is the Python code implementation of linear search:

```
def linear_search(arr, target):
    for i in range(len(arr)):  # Iterating over an array
        if arr[i] == target:  # Compares the current element with the target element
            return i  # If found, returns the index of the element
```

Fig.3.1 the Python code implementation of linear search

In the above code, the loop structure `for i in range(len(arr))` directly determines the time complexity of the algorithm. Because in the worst case, the loop needs to be executed n times (n is the length of the array), so the time complexity is $O(n)$. A simple optimization strategy is to terminate the loop early in certain specific scenarios, such as when the array elements are ordered. But for general unordered arrays, this basic linear search structure is more common.

3.1.4 Advantages and Disadvantages Analysis

1. Advantages: The advantages of linear search are significant. Its algorithm logic is simple and easy to understand and easy to implement. It does not require any preprocessing or special data structure requirements for the array. This makes it very suitable for processing small-scale data or data that does not require a specific order. For example, in some simple data query tasks, if the amount of data is small, linear search can give results quickly.

2. Disadvantages: Since its time complexity is in the worst case $O(n)$, when facing large-scale data, the search efficiency will become extremely low. As the size of data continues to increase, the time required for searching will increase linearly, which may lead to problems such as system response delays and excessive resource consumption. In the era of big data, this high time complexity characteristic limits the application of linear search in large-scale data search scenarios [15].

3.2 Binary search

3.2.1 Algorithm Principle

Binary search is an efficient search algorithm whose core idea is based on the characteristics of ordered arrays. When searching for a target element in a sorted array,

first determine the middle element of the array and compare it with the target element. If the middle element is equal to the target element, the search is successful and the index of the element is returned. If the target element is smaller than the middle element, it can be seen that the target element must be in the left half of the array, so the search range is narrowed to the left half; conversely, if the target element is larger than the middle element, the search range is narrowed to the right half. Then, repeat the above process in the new search range, halving the search interval each time, until the target element is found or it is determined that the target element does not exist in the array [16].

For example, for an array A = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19] in ascending order, to find the target element 7. First, calculate the index of the middle element ($mid = \left\lfloor \frac{0+9}{2} \right\rfloor = 4$ This$\lfloor \ \rfloor$ means round down here). At this time $A[4] = 9$. Since 7 < 9, the search range is narrowed to the left half [1, 3, 5, 7]. Calculate the middle element index again $mid = \left\lfloor \frac{0+3}{2} \right\rfloor = 1$, $A[1] = 3$, because 7 > 3, so the search range is narrowed to the right half [5, 7]. Continue to calculate the middle element index. $mid = \left\lfloor \frac{2+3}{2} \right\rfloor = 2$ At this time $A[2] = 7$, the target element is successfully found and the index 2 is returned.

3.2.2 Time complexity calculation (substitution method)

Let the length of the array be n. The time complexity of the binary search algorithm can be derived by substitution. In each iteration, the length of the search interval is halved. Let the number of iterations be k, then $n/2^k = 1$(the search interval ends when it is reduced to only one element). Solving this formula yields:

$$n / 2^k = 1$$

$$2^k = n$$

$$k = \log_2 n$$

Therefore, the time complexity of binary search is $O(\log n)$. From a mathematical point of view, since each iteration can halve the problem size, this exponential reduction in size leads to a logarithmic growth in time complexity. For example, when n = 16, the

search interval becomes 8 after the first iteration, 4 after the second, 2 after the third, and 1 after the fourth, for a total of $4 = \log_2 16$ iterations [17].

3.2.3 Code Implementation

Here is a sample Python code for binary search:

```
def binary_search(arr, target):
    left = 0
    right = len(arr) - 1

    while left <= right:
        # Calculate the middle element index
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            # The target element is in the right half, update the left
border
            left = mid + 1
        else:
            # The target element is in the left half, update the right
border
            right = mid - 1
    return -1  # Indicates that the target element was not found
```

Fig.3.2 the Python code implementation of linear search

In the above code, boundary condition processing is implemented by ` while left <= right ` to ensure that the loop continues when there are still elements in the search interval. The calculation of the middle element index `mid` uses `(left + right) // 2` . This calculation method ensures that the middle position can be correctly obtained under integer division. When `arr[mid] == target` , the middle element index is returned directly. If `arr[mid] < target` , the left boundary `left` is updated to `mid + 1` , because the left half has been searched at this time; otherwise, the right boundary `right` is updated to `mid - 1` . If the target element is not found at the end of the loop, -1 is returned. These code operations are all completed in constant time and will not affect the overall logarithmic time complexity of the algorithm.

3.2.4 Analysis of advantages and disadvantages

1. Advantages: Binary search has significant advantages. Its high search efficiency makes it perform well when processing large-scale ordered data. Compared with $O(n)$ the time complexity of linear search, binary search $O(\log n)$. Time complexity can greatly reduce search time when the data size is large. For example, to find an element in an ordered array containing millions of elements, binary search only requires about 20 comparisons at most ($\log_2 1000000 \approx 20$), while linear search requires an average of half a million comparisons.

2. Disadvantages: Its main disadvantage is that it requires data to be ordered, which may require additional sorting operations or data maintenance costs in practical applications. For dynamic data sets, every time a new element is inserted or an existing element is deleted, it may be necessary to reorder the data to maintain order, which will undoubtedly bring additional time and space overhead. In addition, if the data is unevenly distributed or there are special circumstances (such as a large number of repeated elements in the data and the target element is repeated multiple times), the performance of binary search may be affected to a certain extent, but overall, in the search scenario of ordered data, its advantages are still very obvious [18].

4. Analysis of recursive algorithm based on substitution method

4.1 Fibonacci sequence recursive algorithm

4.1.1 Algorithm Principle

The Fibonacci sequence is a classic mathematical sequence that is recursively defined as: $F(n) = F(n-1) + F(n-2)$, where $F(0) = 0), F(1) = 1$. This means that to calculate the nth Fibonacci number, you need to first calculate the n-1th and n-2th Fibonacci numbers, and so on, until you recurse to the base case $F(0)$ and $F(1)$. For example, $F(5)$ when calculating, the recursive call process is as follows:

$$F(5) = F(4) + F(3)$$
$$F(4) = F(3) + F(2)$$
$$F(3) = F(2) + F(1)$$
$$F(2) = F(1) + F(0)$$

In this process, many sub problems are calculated repeatedly, for $F(3)$ example, $F(4)$ and $F(5)$ are calculated twice [19].

4.1.2 Time complexity calculation (substitution method)

$T(n)$ the time complexity of $T(n)$ the calculation . According to the recursive definition, $F(n)$ the following recursive relation is satisfied: $T(n) = T(n-1) + T(n-2) + O(1)$, where $O(1)$ represents the time complexity of an addition operation.

In order to solve it using the substitution method, we first guess $T(n)$ the form. Since the Fibonacci sequence itself grows exponentially, we guess $T(n) = O(2^n)$.

We will now verify this conjecture by mathematical induction.

Base case: When n = 0 or n = 1, $T(0) = T(1) = O(1)$, which is obviously satisfied $T(n) = O(2^n)$.

Inductive hypothesis: Assume that for all k < n, $T(k) \le c*2^k$, where c is some constant.

For n, we have:

$$T(n) = T(n-1) + T(n-2) + O(1)$$
$$\leq c * 2^{n-1} + c * 2^{n-2} + d$$
$$= c * 2^{n-2}(2+1) + d$$
$$= c * 2^{n-2} * 3 + d$$
$$\leq c * 2^{n}$$

Where d is a constant, and this formula $O(1)$ holds true when c is large enough $\leq c * 2^{n}$.

From the perspective of the recursive tree, the recursive calculation of the Fibonacci sequence forms a binary tree structure. The number of nodes in each layer is approximately doubled, and the height of the tree is about n. Therefore, the total amount of calculation is approximately 2^{n}, which also intuitively shows that its time complexity is exponential [20].

4.1.3 Code Implementation

Here is the Python code for the Fibonacci sequence recursive algorithm:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Fig.4.1 the Python code for the Fibonacci sequence recursive algorithm

In this code, the recursive function `fibonacci` continuously calls itself to calculate the first two Fibonacci numbers and add them together. For example, when calculating `fibonacci(5)` , `fibonacci(4)` and `fibonacci(3)` are calculated first, and the calculation of `fibonacci(4)` will calculate `fibonacci(3)` and `fibonacci(2)`, resulting in repeated calculation of `fibonacci(3)`. This recursive calling mechanism causes the amount of calculation to grow exponentially with the increase of n, which seriously affects the time complexity of the algorithm [21].

4.1.4 Advantages and Disadvantages Analysis

1. Advantages: The advantage of this algorithm is that it reflects the mathematical definition of the Fibonacci sequence very intuitively, the code is concise and easy to understand. For small values of n, it can quickly give the correct result.

2. Disadvantages: Due to the existence of a large number of redundant calculations, the time complexity is extremely high, which is exponential $O(2^n)$. In practical applications, when n is slightly larger, the time and resources required for calculation will increase dramatically, and the efficiency is extremely low, which is not suitable for large-scale calculations. For example, the calculation $F(100)$ requires a lot of time and memory, and may even cause system resources to be exhausted [22].

4.2 Factorial Recursive Algorithm

4.2.1 Algorithm Principle

The recursive calculation method of factorial is based on the mathematical definition of factorial, that is $n!= n*(n-1)!$, the recursive basis is 0!=1. When calculating the factorial of n, the recursive algorithm will continuously call itself to calculate the factorial of (n - 1) until it reaches the recursive base 0. For example, to calculate 5!, the algorithm will first calculate 4!, then calculate 3!, and so on until it reaches 0!, and then gradually backtrack and multiply the results to obtain the final 5! value [23].

4.2.2 Time complexity calculation (substitution method)

$T(n)$ the time complexity of calculating n ! According to the algorithm principle, calculating n! requires first calculating (n - 1)!, and then performing a multiplication operation, so a recursive equation can be obtained $T(n) = T(n-1) + O(1)$. This $O(1)$ represents the time complexity of the multiplication operation, since its execution time does not change significantly as n changes.

Solving by substitution, we assume $T(n) = O(n)$ (this is based on an intuitive guess about the relationship between the recursive depth of the algorithm and the number of computational steps). Substituting it into the recursive equation:

$$
\begin{aligned}
&T(n) = T(n-1) + O(1) \\
&= O(n-1) + O(1) \\
&= O(n)
\end{aligned}
$$

This confirms our conjecture that the time complexity of the factorial recursive algorithm is $O(n)$, which clearly shows the linear relationship between the recursive depth and the time complexity. As n increases, the calculation time increases roughly linearly [24].

4.2.3 Code Implementation

Here is a Python code example of the factorial recursive algorithm:

```
def factorial(n):
    if n == 0:  # Recursive base
        return 1
    else:
        return n * factorial(n - 1)  # Recursive call to calculate (n-1)! and
multiply it by n
```

Fig.4.2 Python code example of the factorial recursive algorithm

In this code, each recursive call will add a layer of stack frames, and the depth of the recursive call stack is equal to the value of n. For example, when calculating 5!, the recursive call stack will contain stack frames for calculating 5!, 4!, 3!, 2!, 1!, and 0! in sequence. Although this recursive call method concisely implements factorial calculations, it may cause stack overflow problems for larger n. In order to reduce stack space consumption, tail recursion optimization can be used (some Python interpreters do not support tail recursion optimization, so only the optimization ideas are shown here):

```
def factorial_tail(n, accumulator=1):
    if n == 0:
        return accumulator
    else:
        return factorial_tail(n - 1, n * accumulator)  # Tail recursive form,
accumulating the calculation results in the accumulator variable
```

Fig.4.3 Python code example of the tail recursion optimization

This form of tail recursion accumulates the result of the multiplication operation during the recursive call, rather than calculating it during backtracking. In theory, this can reduce the use of stack space, but the actual effect depends on whether the Python interpreter supports tail recursion optimization [25].

4.2.4 Advantages and Disadvantages Analysis

1. Advantages: The advantage of this algorithm lies in its simplicity, which directly follows the mathematical definition of factorial. The code implementation is simple and intuitive, easy to understand and write, and is very consistent with the mathematical way of thinking.

2. Disadvantages: Due to the characteristics of recursive calls, each recursion will occupy a certain amount of stack space. When n is large enough, the stack space may be exhausted and cause a stack overflow error. For example, it may not run properly in some resource-constrained environments or when calculating factorials of large values (such as 10000!). Moreover, compared with iterative algorithms, recursive algorithms may suffer certain performance losses due to the overhead of function calls (such as saving stack frames, parameter transfer, etc.), especially in large-scale data processing scenarios with high performance requirements. , this performance difference may be more significant [26].

5. Comparison and comprehensive analysis of time complexity of substitution methods of different algorithms

5.1 Numerical comparison of time complexity

5.1.1 Sorting Algorithm

1. Bubble sort, insertion sort and quick sort (average case)

When the data size is small (such as $n < 50$), the time complexity curves of bubble sort and insertion sort are relatively close, and their execution time increases relatively slowly. For example, when $n = 10$, bubble sort and insertion sort may only need to perform dozens of comparison and exchange operations.

When the data size is small, the time complexity of quick sort (average case) is slightly higher than that of bubble sort and insertion sort due to the overhead of its recursive call and partition operation. However, as the data size increases, the advantages of quick sort gradually become apparent. When $n = 1000$, the execution time of quick sort is significantly lower than that of bubble sort and insertion sort.

When the data scale reaches $n = 10000$, the time complexity curves of bubble sort and insertion sort rise sharply, and their execution time may reach several seconds or even longer, while the time complexity of quick sort (average case) increases relatively slowly, and the execution time may be in milliseconds.

2. Quick sort (worst case)

In the worst case (such as when the data is already in order or reverse order), the time complexity curve of quick sort is similar to that of bubble sort and insertion sort in large-scale data, showing a steep upward trend. When $n = 10,000$, the execution time of quick sort (worst case) may be several orders of magnitude longer than the average case, and may even cause system freezes or memory overflows.

5.1.2 Search Algorithm

1. Linear search

The time complexity of linear search increases linearly with the growth of data size n. When $n = 1000$, if the target element is at the end of the array or does not exist, linear search may require 1000 comparison operations.

2.Binary Search

The time complexity curve of binary search is very flat, almost horizontal. Even when the data size reaches n = 10,000, binary search only needs to perform $\log_2 10000 \approx 13.29$ comparison operations at most (rounded up to 14 times), and its execution time remains basically at a very low level.

5.1.3 Recursive Algorithm

1. Fibonacci sequence recursive algorithm

The time complexity of the Fibonacci sequence recursive algorithm grows very quickly, exponentially. When n = 30, the time to calculate the Fibonacci sequence may be very long, because the amount of calculation 2^n increases with the increase of n.

2. Factorial recursive algorithm

The time complexity of the factorial recursive algorithm increases linearly, which is relatively gentle. However, when n is large (such as n = 1000), performance problems may occur due to the large depth of the recursive call stack, but it is much better than the Fibonacci sequence recursive algorithm.

Algorithm Name	Best case time complexity	Worst case time complexity	Average case time complexity
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$ (Better than the average case of bubble sort)
Quick Sort	O(nlogn)	$O(n^2)$	O(nlogn)
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(logn)	O(logn)	O(logn)
Fibonacci sequence recursive algorithm	O(1) (when n=0 or 1), Exponential growth (when n>1)	Exponential Growth	Exponential Growth
Factorial Recursive Algorithm	O(1) (when n=0), O(n) (when n>0)	O(n)	O(n)

Table 5.1 Time complexity of different algorithms in various situations

The table 5.1 clearly shows the differences in time complexity of different algorithms in various situations. When the data size n is small, for example $n \leq 10$, bubble sort, insertion sort, and linear search have low time complexity in the best case and may perform better. When n gradually increases, the advantages of quick sort and binary search gradually emerge. Due to its exponentially growing time complexity, the computational cost of the Fibonacci sequence recursive algorithm will increase sharply when n is slightly larger.

5.2 Analysis of applicable scenarios

5.2.1 Sorting Algorithm

1. Bubble Sort and Insertion Sort

Small data size (n < 100): In this case, the simplicity of bubble sort and insertion sort makes them cheap to implement and understand. For example, when sorting a small array (such as within 10 elements), they can complete the sorting task quickly, and the simplicity of the code makes debugging and maintenance relatively easy.

Partially ordered data: If the data is already nearly ordered, insertion sort performs better because it only needs to insert a small number of unordered elements. For example, in a real-time data acquisition system, the newly collected data may only be partially unordered, and insertion sort can efficiently integrate the new data into the sorted data sequence. Although bubble sort can also work in this case, it will perform more unnecessary comparison operations than insertion sort.

Scenarios with high stability requirements: Both sorting algorithms are stable, that is, the relative order of equal elements does not change after sorting. When sorting a task list containing tasks of the same priority, the original order of tasks of the same priority can be preserved.

2. Quick Sort

Large-scale data sorting (n>1000): The average time complexity of quick sort is $O(nlogn)$, which performs well in large-scale data sorting. For example, in scenarios such as database sorting operations and analysis and processing of large data sets, quick sort can quickly sort data and improve data processing efficiency.

Memory-constrained scenarios (in-place sorting feature): Quick sort is an in-place sorting algorithm that does not require a large amount of additional storage space, but only requires a small amount of auxiliary space for the recursive call stack. This allows it to effectively sort data in environments with limited memory resources, such as embedded systems or server applications with tight memory.

Data distribution is relatively uniform: When data distribution is relatively uniform, quick sort can better play the advantages of its divide-and-conquer strategy and quickly divide the array into roughly equal sub-arrays, thereby improving sorting efficiency. However, for uneven data distribution, performance can be improved by optimizing strategies such as randomly selecting benchmark elements.

5.2.2 Search Algorithm

1. Linear Search

Small or unordered data: When the data is small (e.g., $n < 100$) and the data is unordered, the simplicity and versatility of linear search make it a good choice. For example, to find a specific record in a small database table or to find an element in an unsorted simple data set, linear search can give results quickly and does not require preprocessing of the data.

Data changes frequently and has no sorting requirements: If data changes frequently (insertion and deletion of elements) and does not need to be kept in order, linear search can be performed at any time in the new data state without requiring additional maintenance costs to keep the data in order.

2. Binary Search

Large-scale ordered data search: When processing large-scale ordered data (such as ordered arrays, ordered lists, etc.), the efficiency of binary search makes it the first choice. For example, in scenarios such as searching for words in a large dictionary data structure or searching for records at a specific time point in an ordered log file, binary search can quickly locate the target element and greatly reduce search time.

Scenarios of frequent search operations: If an application requires frequent search operations, and the data can be pre-sorted and kept in order, then binary search can significantly improve search efficiency and reduce system response time. For example,

in scenarios such as search engine index search and game ranking query, the advantages of binary search can be fully demonstrated.

5.2.3 Recursive algorithm

1. Fibonacci Sequence Recursive Algorithm

Small-scale calculations (n < 30) with low efficiency requirements: Since its algorithm intuitively reflects the mathematical definition of the Fibonacci sequence, this algorithm can be used for scenarios where small-scale Fibonacci numbers (such as within the first 30 items) are calculated and low efficiency is required, such as simple math teaching demonstrations, small-scale math calculation tasks, etc. However, it should be noted that even in small-scale cases, as n increases, the calculation time will increase significantly.

Theoretical research and conceptual understanding: In algorithm theory research and teaching, the Fibonacci sequence recursive algorithm is a good example to explain theoretical knowledge such as recursive concepts, algorithm complexity analysis, and recursive tree structure, helping students understand the nature and characteristics of recursive algorithms.

2. Factorial recursive algorithm

Small-scale calculations (n < 100) and simplicity: The factorial recursive algorithm is a suitable choice for scenarios where you need to calculate factorials of smaller numbers (such as within 100!) and pursue code simplicity and mathematical intuition. For example, this algorithm can be used to calculate factorials of smaller numbers in some simple mathematical calculation programs or scripts.

Basic modules combined with other algorithms (small scale): In some complex algorithms, if you need to calculate a small-scale factorial as one of the sub modules, and the sub module is not called frequently, the factorial recursive algorithm can be used as a concise implementation method. However, in practical applications, if you need to calculate factorials on a large scale, you should consider using iterative algorithms or more efficient math library functions.

5.3 Relationship between Algorithm Optimization and Replacement Method

5.3.1 Bubble sort optimization variant - cocktail sort (bidirectional bubble sort)

1. Optimization strategy: Cocktail sort is an improvement on bubble sort. It first compares and exchanges adjacent elements from front to back, "bubbles" the largest element to the end of the array, and then compares and exchanges adjacent elements from back to front, "bubbles" the smallest element to the beginning of the array. In this way, a complete two-way traversal can determine the position of the largest and smallest elements, reducing the number of traversals compared to ordinary bubble sort.

2. Impact of time complexity and verification of substitution method: Suppose the time complexity function of cocktail sort is $T(n)$, where n is the number of array elements.

In the best case, the array is already ordered, and only one round of traversal from front to back and from back to front is needed, with n - 1 comparisons (n - 1 comparisons from front to back, n - 2 comparisons from back to front, but the constant term can be ignored), so $T(n) = O(n)$.

In the worst case, assume that the array is in reverse order. The first round of traversal from front to back requires n - 1 comparisons and exchanges, and traversal from back to front requires n - 2 comparisons and exchanges; the second round of traversal from front to back requires n - 3 comparisons and exchanges, and traversal from back to front requires n - 4 comparisons and exchanges, and so on. The total number of comparisons and exchanges is $(n-1)+(n-2)+(n-3)+...+1=\frac{n(n-1)}{2}$, so $T(n) = O(n^2)$, although it is still quadratic, the number of comparisons in the worst case is reduced by about half compared to ordinary bubble sort.

In the average case, mathematical analysis shows that its time complexity has also improved, getting closer $O(nlogn)$ but still $O(n^2)$ at the same level.

3. Python code implementation:

```python
def cocktail_sort(arr):
    n = len(arr)
    swapped = True
    start = 0
    end = n - 1
    while swapped:
        swapped = False
        # Traverse from front to back
        for i in range(start, end):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end -= 1
        # Traverse from back to front
        for i in range(end - 1, start - 1, -1):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        start += 1
    return arr
```

Fig.5.1 Python code implementation

5.3.2 Optimizing the base selection of quick sort - randomized quick sort

1. Optimization strategy: When selecting the benchmark element, randomized quick sort no longer selects the first or last element, but uses a random function to randomly select an element in the array range as the benchmark. This can reduce the impact of the initial order of data on the algorithm performance and reduce the probability of the worst case.

2. Impact of time complexity and verification by substitution method: Assume that the time complexity of randomized quick sort is $T(n)$. On average, due to the random selection of benchmark elements, the probability that the size of the subproblems is roughly equal after each division increases. Assuming that the size of the subproblems after each division is roughly $\frac{n}{2}$, then the time complexity of the division operation is $O(n)$, and the time complexity of recursively sorting the two subarrays is respectively

$T(\frac{n}{2})$. According to the divide-and-conquer strategy, the recursive formula for the time complexity in the average case is $T(n) = 2T(\frac{n}{2}) + O(n)$. Applying the substitution method, assume that $T(n) = O(n \log n)$, substituting it into the recursive formula can verify that the assumption is true.

In the worst case, if the randomly selected base element always leads to extremely unbalanced partitions (although the probability is extremely low), for example, when the array is already in order or reverse order, each partition will result in an empty subarray and a subarray of length n - 1. At this time, the time complexity recursive formula is $T(n) = T(n-1) + O(n)$, which can be obtained by expanding the recursive formula $T(n) = O(n^2)$, but this situation rarely occurs in practice.

3. Python code implementation (modify the benchmark selection part based on the previous quick sort code):

```python
import random
def quick_sort_randomized(arr):
    def partition(arr, low, high):
        # Randomly select the pivot element and swap it with the first element
        pivot_index = random.randint(low, high)
        arr[low], arr[pivot_index] = arr[pivot_index], arr[low]
        pivot = arr[low]
        i = low + 1
        j = high
        while True:
            # Find the first element greater than the pivot from left to right
            while i <= j and arr[i] <= pivot:
                i += 1
            # Find the first element smaller than the pivot from right to left
            while i <= j and arr[j] > pivot:
                j -= 1
            if i <= j:
 # Swap two elements so that the left side is less than or equal to the reference and the right side is greater than or equal to the reference
                arr[i], arr[j] = arr[j], arr[i]
            else:
                break
        # Put the datum element in the correct position
        arr[low], arr[j] = arr[j], arr[low]
        return j

    def quick_sort_helper(arr, low, high):
        if low < high:
            # Divide the operation to obtain the final position of the base element
            pivot_index = partition(arr, low, high)
            # Recursively sort the left and right subarrays
            quick_sort_helper(arr, low, pivot_index - 1)
            quick_sort_helper(arr, pivot_index + 1, high)

    quick_sort_helper(arr, 0, len(arr) - 1)
    return arr
```

Fig.5.2 Python code implementation

5.3.3 Tail recursion optimization of recursive algorithms - taking the factorial recursive algorithm as an example

1. Optimization strategy: The tail-recursive optimized factorial algorithm (such as the `factorial_tail` function mentioned above) accumulates the results of the

multiplication operation during the recursive call, avoiding the overhead of the ordinary recursive algorithm when performing calculations during backtracking, while reducing the use of stack space.

2. Impact of time complexity and verification by substitution method: Assume that the time complexity of the optimized factorial algorithm is $T(n)$. When calculating n!, each recursive call of the optimized algorithm will reduce the problem size n until n = 0. The time of each recursive call is mainly consumed in the multiplication operation, and its time complexity is $O(1)$. So the total time complexity is $T(n) = O(n)$), which is the same as the time complexity of the ordinary factorial recursive algorithm, but in actual execution, due to the reduction in the use of stack space, for larger n, the stack overflow problem can be avoided, so that the algorithm can process larger-scale data with limited resources. Through the analysis of the substitution method, the optimized algorithm does not require additional stack space to save intermediate results during the calculation process, and the essence of the recursive call is still linear, so the time complexity remains unchanged, but the performance is improved.

3. Python code implementation (comparison before and after optimization):

```python
# Ordinary factorial recursive algorithm
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
# Tail recursion optimized factorial algorithm
def factorial_tail(n, accumulator=1):
    if n == 0:
        return accumulator
    else:
        return factorial_tail(n - 1, n * accumulator)
```

Fig.5.2 Python code implementation

6. Conclusion and Outlook

6.1 Summary of Research Results

This article deeply explores the application of substitution method in calculating the time complexity of algorithms. Through the analysis of common algorithms such as sorting algorithms (bubble sort, insertion sort, quick sort), search algorithms (linear search, binary search) and recursive algorithms (Fibonacci sequence recursive algorithm, factorial recursive algorithm), it elaborates on how the substitution method can effectively derive the time complexity of these algorithms.

In the sorting algorithm analysis, we showed that the time complexity of bubble sort and insertion sort is 2.316 in the worst case $O(n^2)$, while quick sort can achieve excellent performance in the average case $O(nlogn)$. Through the substitution method, we accurately calculated the time complexity of these algorithms in different cases and compared their advantages and disadvantages. Although bubble sort and insertion sort are simple to implement, they are less efficient when processing large-scale data; while quick sort has become the preferred choice in practical applications due to its efficient average performance.

In terms of search algorithms, we compared the time complexity of linear search and binary search. The time complexity of linear search is $O(n)$, that is, as the size of the data increases, the search time increases linearly. Binary search $O(logn)$ is significantly better than linear search in terms of time complexity. Especially on large-scale data sets, the efficiency advantage of binary search is even more obvious.

For recursive algorithms, we analyze the time complexity of Fibonacci sequence recursive algorithms and factorial recursive algorithms. These algorithms have a typical recursive structure. Through the substitution method, we can clearly see how the recursive formula is transformed into an asymptotic representation of the time complexity. These analyzes not only deepen the understanding of recursive algorithms, but also provide important basis for algorithm optimization.

In summary, this article systematically analyzes the time complexity of common algorithms through the substitution method, reveals the performance characteristics of

different algorithms under different circumstances, and provides strong theoretical support for algorithm selection and optimization.

6.2 Research limitations and prospects

Although this paper has achieved certain research results in calculating the time complexity of algorithms using the substitution method, there are still some limitations. First, in terms of algorithm selection, this paper mainly focuses on some classic sorting, searching, and recursive algorithms. However, there are many types of algorithms in the field of computer science, and there are many other types of algorithms (such as dynamic programming algorithms, graph algorithms, etc.) that deserve in-depth study. In the future, we can apply the substitution method to more types of algorithms to expand its scope of application. Second, in terms of analysis depth, this paper mainly focuses on the derivation and comparison of algorithm time complexity. However, the performance of an algorithm depends not only on time complexity, but also on multiple aspects such as space complexity, stability, and adaptability. Therefore, future research can further explore the application of the substitution method in these aspects to provide a more comprehensive analysis of algorithm performance.

In addition, in terms of coverage of actual application scenarios, although the algorithms analyzed in this article are representative, they do not cover all possible actual application scenarios. For example, in the fields of big data processing, real-time systems, parallel computing, etc., the performance requirements of algorithms may be more complex and diverse. Therefore, future research can analyze the performance of algorithms for these specific application scenarios and explore how to combine the replacement method for algorithm optimization and selection.

Looking into the future, the research on substitution method in the field of algorithm time complexity calculation has broad prospects. With the continuous development of computer science and technology, new types of algorithms and more complex data structures continue to emerge, which puts higher requirements on the analysis and optimization of algorithm performance. As a basic and important time complexity analysis method, substitution method will continue to play an important role in this process. In the future, we can explore combining substitution method with other

analysis methods (such as experimental analysis, simulation, etc.) to provide more accurate and comprehensive algorithm performance evaluation and optimization strategies. At the same time, we can also pay attention to the performance of the algorithm in practical applications, and carry out targeted algorithm optimization and improvement according to actual needs.

References:

[1] Kimani, CPA John, and James Scott. *Introduction to Algorithms Professional Level* . Finstock Evarsity Publishers, 2023.

[2] Knuth, Donald Ervin. The art of computer programming. Vol. 3. Pearson Education, 1997.

[3] Kozen, Dexter C. *The design and analysis of algorithms*. Springer Science & Business Media, 2012.

[4] Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Clifford Stein.(2009). Introduction to algorithms."

[5] McKinney, Wes. Python for data analysis. " O'Reilly Media, Inc.", 2022.

[6] Sedgewick, Robert, and Kevin Wayne. *Algorithms*. Addison-wesley professional, 2011.

[7] Nicola, Stelian, Lacramioara Stoicu-Tivadar, and Alexandru Patrascoiu. "VR for Education in Information and Tehnology: application for Bubble Sort." *2018 international symposium on electronics and telecommunications (isetc)*. IEEE, 2018.

[8] Dasgupta, Sanjoy, Christos H. Papadimitriou, and Umesh Vazirani. Algorithms. McGraw-Hill, Inc., 2006.

[9] Lobb, Richard, and Jenny Harlow. "Coderunner: A tool for assessing computer programming skills." *ACM Inroads* 7.1 (2016): 47-51.

[10] Knuth, Donald E. *The Art of Computer Programming: Seminumerical Algorithms, Volume 2*. Addison-Wesley Professional, 2014.

[11] Wilf, Herbert S. *Algorithms and complexity*. AK Peters/CRC Press, 2002.

[12] Mayer, Richard E. *Teaching and learning computer programming: Multiple research perspectives*. Routledge, 2013.

[13] Cederman, Daniel, and Philippas Tsigas. "Gpu-quicksort: A practical quicksort algorithm for graphics processors." *Journal of Experimental Algorithmics (JEA)* 14 (2010): 1-4.

[14] Hadjerrouit, Said. "Towards a blended learning model for teaching and learning computer programming: A case study." *Informatics in Education-An International Journal* 7.2 (2008): 181-210.

[15] Szeliski, Richard. *Computer vision: algorithms and applications*. Springer Nature, 2022.

[16] Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2022.

[17] Mora, A., et al. "Logic-based functional dependencies programming." *Proceedings of the International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2010, Almeria, Spain*. Vol. 1. 2010.

[18] Takamatsu, Mizuyo, and Satoru Iwata. "Index reduction for differential–algebraic equations by substitution method." *Linear algebra and its applications* 429.8-9 (2008): 2268-2277.

[19] Shyam Sumukh, S. R., Raghav Gupta, and Jagadish Nayak. "A Comparative Analysis in Hardware Partitioning of a Steganographic based LSB-substitution Algorithm." *Proceedings of the World Congress on Engineering and Computer Science*. Vol. 1. 2014.

[20] Lin, Anthony. "Binary search algorithm." *WikiJournal of Science* 2.1 (2019): 1-13.

[21] Cholissodin, Imam, and Efi Riyandani. "A State-of-the-Art of Time Complexity (Non-Recursive and Recursive Fibonacci Algorithm)." *Journal of Information Technology and Computer Science* 1.1 (2016): 14-27.

[22] Xu, Hongquan. "Algorithmic construction of efficient fractional factorial designs with large run sizes." *Technometrics* 51.3 (2009): 262-277.

[23] Gu, Ming. "Recursive Algorithm and its Practice in C Language Online Course Teaching." *Advances in Science and Technology* 105 (2021): 341-347.

[24] Oliveto, Pietro S., and Carsten Witt. "Improved time complexity analysis of the simple genetic algorithm." *Theoretical Computer Science* 605 (2015): 21-41.

[25] Chong, Chee-Way, P. Raveendran, and Ramakrishnan Mukundan. "A comparative analysis of algorithms for fast computation of Zernike moments." *Pattern Recognition* 36.3 (2003): 731-742.

[26] McConnell, Jeffrey. *Analysis of algorithms*. Jones & Bartlett Publishers, 2007.