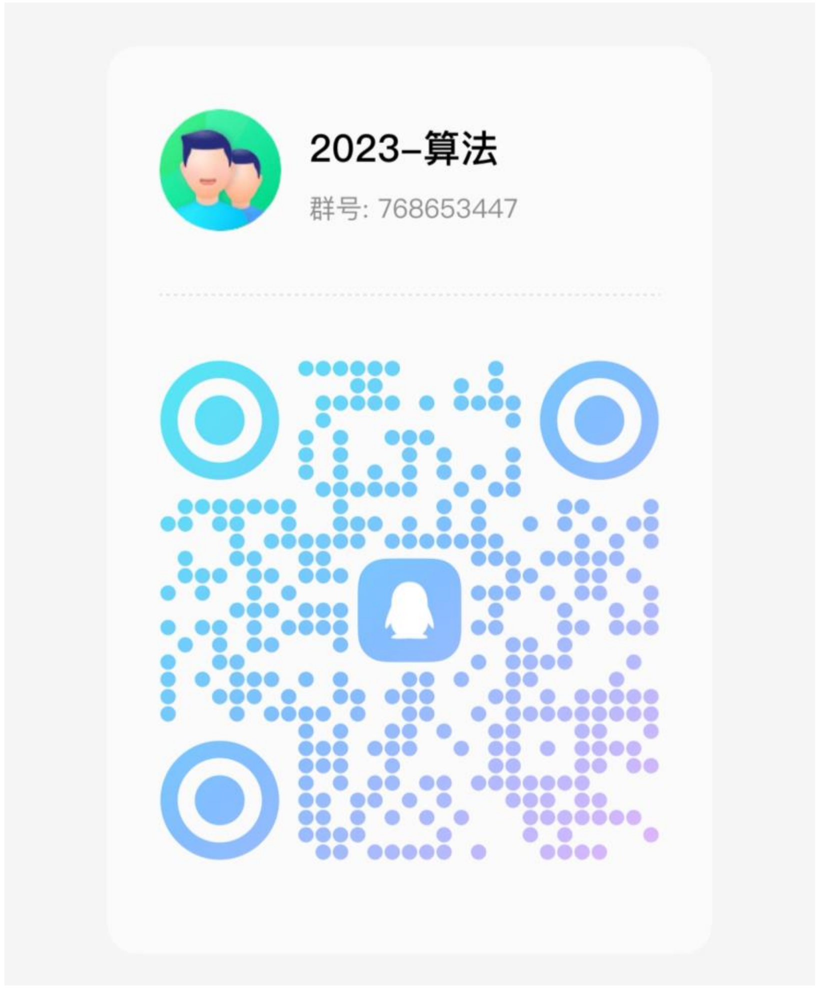
Algorithm Design and Analysis

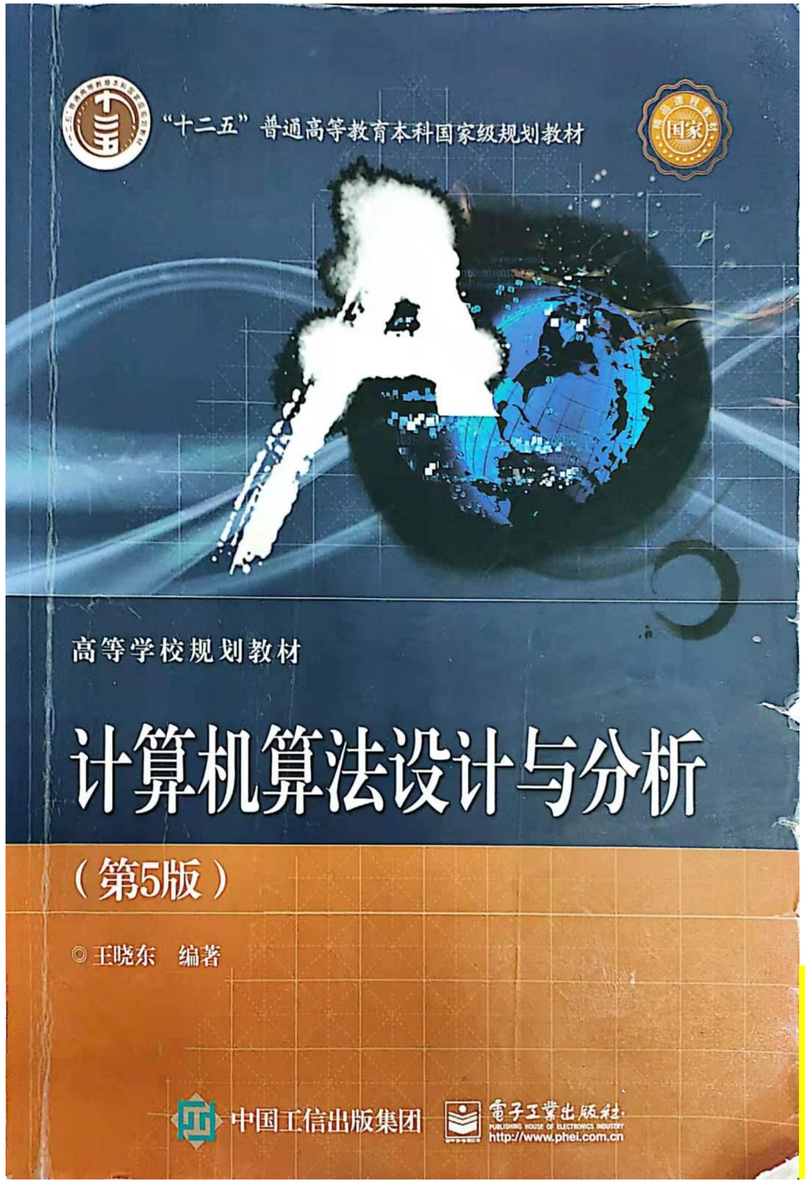
张书妍（360191917）



2023-算法

群号: 768653447

Teaching materials


"十二五"普通高等教育本科国家级规划教材

国家

高等学校规划教材

计算机算法设计与分析

(第5版)

◎王晓东 编著

中国工信出版集团 電子工業出版社 http://www.phei.com.cn

Chapter 1 Algorithm overview

Chapter 2 Recursion and divide-and-conquer strategy

Chapter 3 Dynamic programming

Chapter 4 Greedy algorithm

Chapter 5 Backtracking

Chapter 6 Branch and limit method

Theory class: 48 credit hours

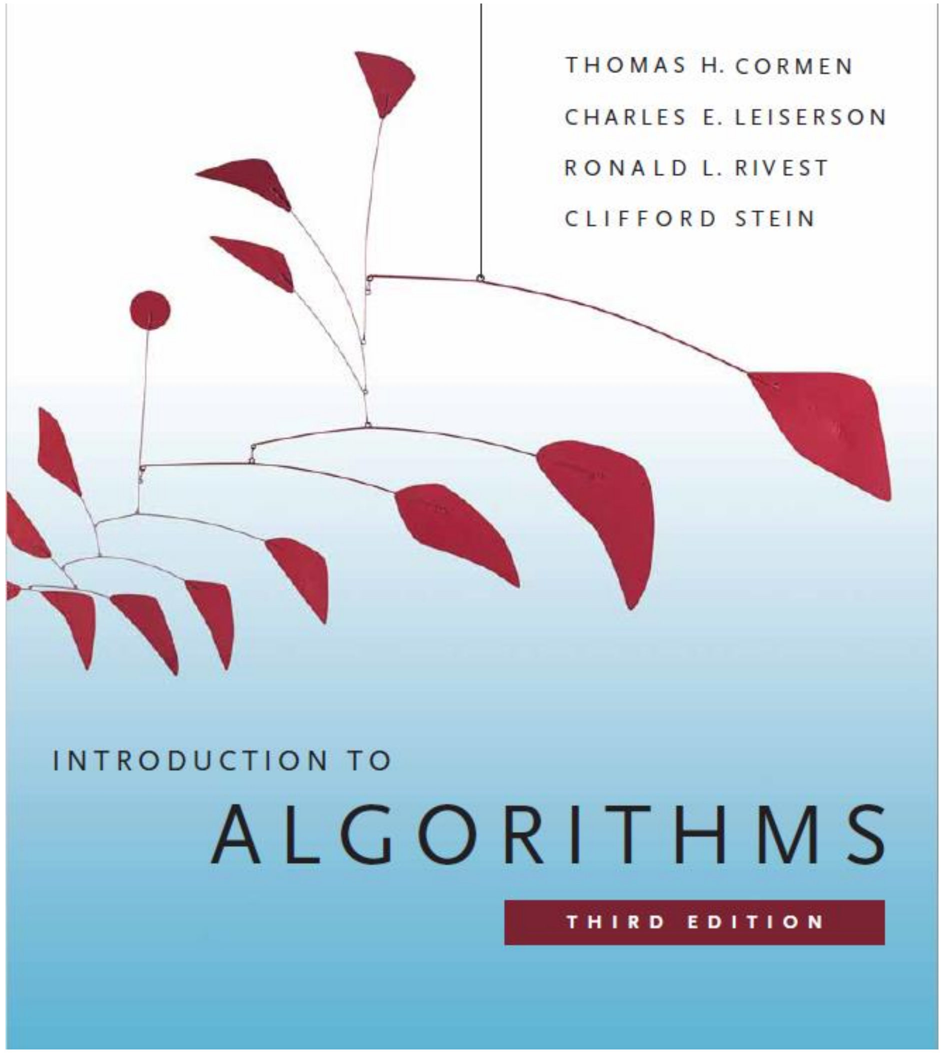
Time: 12 weeks

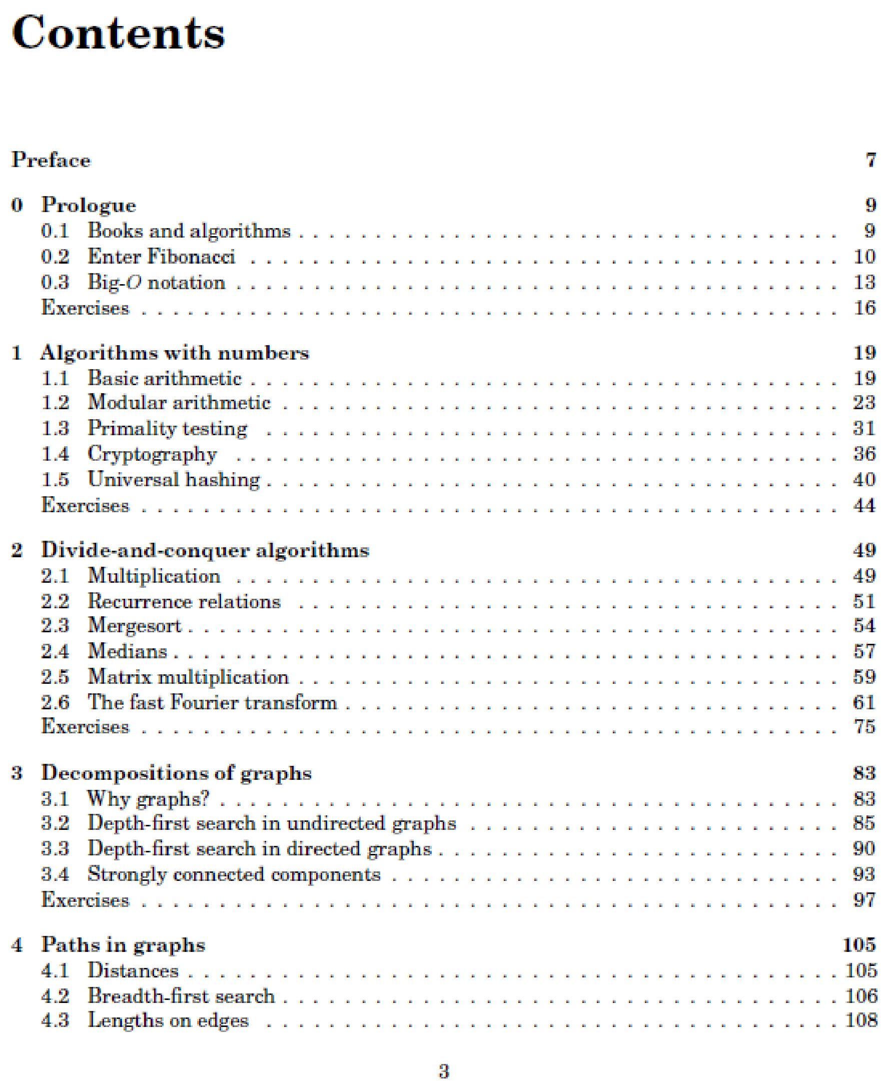
Usual score: 40 points

Final exam: 60 points

Attendance (5%), Course assignments(15%), Computer experiment(20%), Final examination (60%, Closed book examination)

Teaching materials





Contents

Preface 7

0 Prologue 9
0.1 Books and algorithms . . . 9
0.2 Enter Fibonacci . . . 10
0.3 Big-O notation . . . 13
Exercises . . . 16

1 Algorithms with numbers 19
1.1 Basic arithmetic . . . 19
1.2 Modular arithmetic . . . 23
1.3 Primality testing . . . 31
1.4 Cryptography . . . 36
1.5 Universal hashing . . . 40
Exercises . . . 44

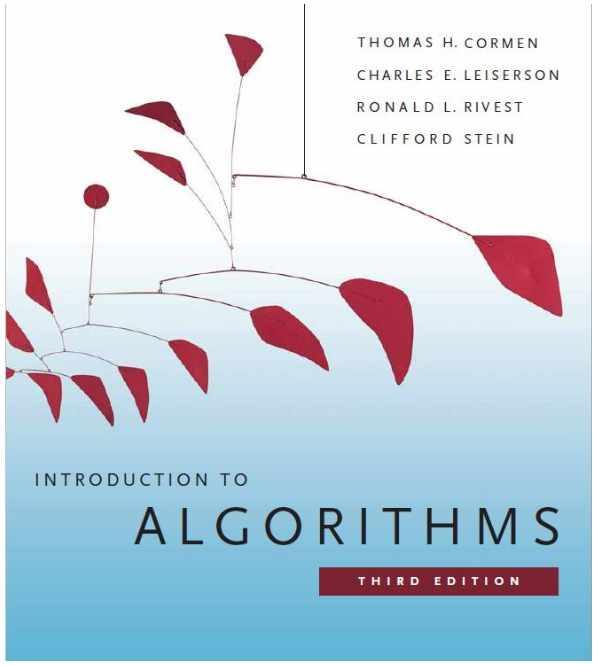
2 Divide-and-conquer algorithms 49
2.1 Multiplication . . . 49
2.2 Recurrence relations . . . 51
2.3 Mergesort . . . 54
2.4 Medians . . . 57
2.5 Matrix multiplication . . . 59
2.6 The fast Fourier transform . . . 61
Exercises . . . 75

3 Decompositions of graphs 83
3.1 Why graphs? . . . 83
3.2 Depth-first search in undirected graphs . . . 85
3.3 Depth-first search in directed graphs . . . 90
3.4 Strongly connected components . . . 93
Exercises . . . 97

4 Paths in graphs 105
4.1 Distances . . . 105
4.2 Breadth-first search . . . 106
4.3 Lengths on edges . . . 108

3

Before there were computers, there were algorithms. But now that there are computers, there are even more algorithms, and algorithms lie at the heart of computing.



Learning Points:

- Understand the concept of algorithms.
- Understand what is the program, and difference and inner link of the algorithm.
- Grasp the concept of **computational complexity** of algorithms.
- Grasp the mathematical representation of **asymptotic complexity** of algorithms.
- Master the method of describing algorithm with **C language.**

Chapter 1 Algorithm overview

What are algorithms? Why is the study of algorithms worthwhile? What is the role of algorithms relative to other technologies used in computers?

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

1.1 Algorithms and procedures

►An algorithm is a method or process for solving a problem. An algorithm is a finite sequence of instructions, which satisfies the properties:

(1)**Input**: an external quantity is provided as the input of the algorithm.

(2)**Output**: The algorithm produces at least one quantity as output.

(3)**Determinacy**: Each instruction that makes up the algorithm is clear and unambiguous.

(4)**Finiteness**: the execution times of each instruction in the algorithm are limited, and the execution time of each instruction is also limited.

1.1 Algorithms and procedures

►Program is a concrete realization of algorithm in some programming language. The nature of the program can not satisfy the algorithm (4).

An operating system, for example, is a program that executes in an infinite loop, not an algorithm. The various tasks of the operating system can be viewed as separate problems, and each problem is implemented by a subroutine of the operating system through a specific algorithm. The subroutine gets the output and then terminates.

1.2 Algorithm complexity analysis

- Algorithm complexity = computer resources required by the algorithm: time and space (i.e. memory) resources;
- The time complexity of the algorithm is T and the space complexity of the algorithm is S.

1.2 Algorithm complexity analysis

T and *S* are functions of *N*, *I* and *A*.

N: problem size (input size); I: input of the algorithm; *A*: The algorithm itself

Let *T(N,I)* be the required time, and there are k kinds of meta-algorithms provided by the computer, denoted as *O1,O2... Ok*, the time required for each meta-computation is denoted as *T1, T2... Tk*, for a given algorithm A, the number of meta-operation *Oi* used is *ei(I =1,2... , k)*, then

$$T(N,I) = \sum_{i=1}^{k} t_i e_i(N,I)$$

1.2 Algorithm complexity analysis

Considering the time complexity of the three situations, that is the worst, the best conditions and the time complexity of the average case

- （1）Worst-case time complexity

$T_{\max}(n) = \max\{\ T(\mathrm{I}) \mid \mathrm{size}(\mathrm{I})=n\ \}$

- （2）Best-case time complexity

$T_{\min}(n) = \min\{\ T(\mathrm{I}) \mid \mathrm{size}(\mathrm{I})=n\ \}$

- （3）Time complexity in the average case

$$T_{\mathrm{avg}}(n) = \sum_{size(I)=n} p(I)T(I)$$

Where I is the instance of the problem (size n), and p(I) is the probability of occurrence of real example I.

The most maneuverable and practical value is the worst-case time complexity.

1.2 Algorithm complexity analysis

Time Complexity

```
int fun(int n)
{
    int i=n;
    int j=3*n;
    return i+j;
}
```

O(1)

```
int fun(int n)
{
    int i=1;
    while(i<=n)
        i=i*2;
    return i;
}
```

O(log$n$)

```
int fun(int n)
{
    sum=0;
    for(int i=0;i<n;i++)
        sum+=i;
    return sum;
}
```

O(n)

```
int fun(int m,int n)
{
    int sum=0;
    for(int i=1;i<=m;i++)
        sum+=i;
    for(int i=2;i<=n;i++)
        sum+=i;
    return sum;
}
```

O(m+n)

1.2 Algorithm complexity analysis

Time Complexity

```
int fun(int m,int n)
{
   sum=0;
   for(int i=0;i<m;i++)
   {
        for(int j=0;j<n;j++)
       {
           sum+=i*j;
           j=j*2;
       }
   }
   return sum;
}
```

$\mathrm{O}(\mathrm{mlog}n)$

```
int fun(int n)
{
   sum=0;
   for(int i=0;i<n;i++)
   {
        for(int j=0;j<n;j++)
       {
           sum+=i*j;
       }
   }
   return sum;
}
```

$\mathrm{O}(n^2)$

$O(1) < O(\log n) < O(n) < O(n\log n)$

$< O(n^2) < O(2^n) < O(n!)$

Space Complexity

```
int fun(int n)
{
    sum=0;
    for(int i=0;i<n;i++)
        sum+=i;
    return sum;
}
```

$O(1)$

```
int fun(int n)
{
    int arr[N];
    while(i<=N)
        i=i*2;
    return i;
}
```

$O(n)$

```
int fun(int m,int n)
{
    int arr[M][N];
    for(int i=1;i<=m;i++)
        for(int j=1;j<=n;j++)
            sum+=arr[i][j];
   return sum;
}
```

$O(MN)$

$O(1) < O(n) < O(n^2)$

1.2 Algorithm complexity analysis

Asymptotic complexity of the algorithm

$T(n) \to \infty$, as $n \to \infty$;

$(T(n) - \tilde{T}(n))/\, T(n) \to 0$, as $n \to \infty$;

$\tilde{T}(n)$is the asymptotic state of T(n) and is the asymptotic complexity of the algorithm.

In mathematics, $\tilde{T}(n)$ is the asymptotic expression of T(n), which is the principal term left by T(n) omitting the lower order terms. It's simpler than T(n).

Comparing the efficiency of the two algorithms, comparing the progressive complexity of the two algorithms, and determine the highest order of each, without caring about the constant factor. (When the scale of the problem is sufficiently large)

Notation for asymptotic analysis

In the following discussion, for all n，$f(n) \geq 0$，$g(n) \geq 0$。

（1）**Asymptotic upper bound notation*O* – Give an upper bound on the function f**

$O(g(n))$ = { $f(n)$ | there are positive constant c and n_0 such that for all $n \geq n_0$ there is： $0 \leq f(n) \leq cg(n)$ }

E.g.： f(n) = 3n+2

When n>=2， 3n+2 <= 3n+n = 4n， so f(n) = O(n)， f(n) is a linearly varying function.

Similarly, 100n+6 = O(n).

In particular, when f(n) is a constant c， for example f(n)=9， can be written as f(n) = O(1)

1.2 Algorithm complexity analysis

$f(n^2) = 10n^2 + 4n + 2$

When $n>=2$， $10n^2 + 4n + 2 <= 10n^2 + 5n$

When $n>=5$， $10n^2 + 5n <= 10n^2 + n^2 = 11n^2$ ， so $f(n) = O(n^2)$

1.2 Algorithm complexity analysis

（2）**Asymptotic lower bound notation Ω**

$\Omega(g(n))$ = { $f(n)$ | there are positive constant c and n_0 such that for all $n \geq n_0$ there is： $0 \leq cg(n) \leq f(n)$ }

$3n+2 >= 3n$ $3n+2 = \Omega(n)$

$100n+6 >= 100n$ $100n+6 = \Omega(n)$

$10n^2+4n+2 >= 10n^2$ $10n^2+4n+2 = \Omega(n^2)$

1.2 Algorithm complexity analysis

（3）**Noncompact upper bound notation *o***

$o(g(n))$ = { $f(n)$ | for any positive constant c>0, there are positive numbers n_0 >0 such that for all $n \geq n_0$, there is： $0 \leq f(n) < cg(n)$ }

Equivalent to $f(n) / g(n) \to 0$， as $n \to \infty$。

（4）**Noncompact lower bound notation *ω***

$\omega(g(n))$ = { $f(n)$ | for any positive constant c>0, there are positive numbers n_0 >0 such that for all $n \geq n_0$, there is： $0 \leq cg(n) < f(n)$ }

Equivalent to $f(n) / g(n) \to \infty$， as $n \to \infty$。

$$f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$$

1.2 Algorithm complexity analysis

（5）**Compact asymptotic bound notation θ**

$\theta\ (g(n)) = \{\ f(n)$ | there are positive constant c_1,c_2 and n_0 such that for all $n \geq n_0$ there is： $c_1 g(n) \leq f(n) \leq c_2 g(n)\ \}$

Theory 1： $\theta(g(n)) = O\ (g(n)) \cap \Omega\ (g(n))$

1.2 Algorithm complexity analysis

Comparison of functions in asymptotic analysis

- *The rank of* $f(n)= O(g(n)) \approx f(n)$ *is not higher than the rank of* g(n)
- *The rank of* $f(n)= \Omega(g(n)) \approx f(n)$ is *not lower than the rank of* g(n)
- *The rank of* $f(n)= \theta(g(n)) \approx f(n)$ equals to the rank of g(n)
- *The rank of* $f(n)= o(g(n)) \approx f(n)$ is less than the rank of g(n)
- *The rank of* $f(n)= \omega(g(n)) \approx f(n)$ is greater than the rank of g(n)

1.2 Algorithm complexity analysis

Some properties of asymptotic analysis notation

（1）Transitivity：

$f(n)=\theta(g(n))$， $g(n)=\theta(h(n)) \Rightarrow f(n)=\theta(h(n))$；

$f(n)=O(g(n))$， $g(n)=O(h(n)) \Rightarrow f(n)=O(h(n))$；

$f(n)=\Omega(g(n))$， $g(n)=\Omega(h(n)) \Rightarrow f(n)=\Omega(h(n))$；

$f(n)=o(g(n))$， $g(n)=o(h(n)) \Rightarrow f(n)=o(h(n))$；

$f(n)=\omega(g(n))$， $g(n)=\omega(h(n)) \Rightarrow f(n)=\omega(h(n))$；

1.2 Algorithm complexity analysis

（2）Reflexivity：

$f(n)= \theta(f(n))$；

$f(n)= O(f(n))$；

$f(n)= \Omega(f(n))$.

（3）Symmetry：

$f(n)= \theta(g(n)) \Leftrightarrow g(n)= \theta(f(n))$.

（4）Mutual symmetry：

$f(n)= O(g(n)) \Leftrightarrow g(n)= \Omega(f(n))$ ；

$f(n)= o(g(n)) \Leftrightarrow g(n)= \omega(f(n))$ ；

1.2 Algorithm complexity analysis

（5）Arithmetic operations：

$O(f(n))+O(g(n)) = O(\max\{f(\mathrm{n}),g(n)\})$ ；

$O(f(n))+O(g(n)) = O(f(\mathrm{n})+g(n))$ ；

$O(f(n))*O(g(n)) = O(f(\mathrm{n})*g(n))$ ；

$O(cf(n)) = O(f(\mathrm{n}))$ ；

$g(n)= O(f(n)) \Rightarrow O(f(n))+O(g(n)) = O(f(\mathrm{n}))$ 。

Basic principles of algorithm analysis

Non-recursive algorithm:

(1) For/while loop

Circulatory body calculation time * number of cycles;

(2) Nested loop

Circulatory body calculation time * all cycles;

(3) Sequential statement

Sum the computation time of each statement;

(4) If-else statements

iThe greater of the if statement computation time and the else statement computation time.

1.2 Algorithm complexity analysis

- Recursive algorithm: establish the recursive relation of the execution times of the basic operation of the algorithm, and then determine its increment times

```
int factorial(int n)
 {
       if (n == 0) return 1;
       return n*factorial(n-1);
  }
```

$$T(n)=\begin{cases} 0 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

NP-complete theory

All decision problems that can be solved in polynomial time constitute class P problems. Polynomial time is in view of the problem scale, that is, the time required to solve the problem is the size of the problem of polynomial.

NP problems: nondeterministic polynomial problems.

All deterministic polynomial-time solvable decision problems constitute NP class problems.

NP-complete theory

Nondeterministic algorithm: Problem solving is divided into two stages: guess and verify. The guess phase of the algorithm is nondeterministic and gives a guess of the problem. The verification stage of the algorithm is deterministic and verifies the correctness of the solution of the previous stage. If the verification stage can be completed in polynomial, the algorithm is said to be a polynomial-time nondeterministic algorithm, and the problem is also said to be nondeterministic polynomial-time solvable.

Summary

- An "algorithm" is a clear sequence of instructions for solving a problem in limited time. The input of the algorithm determines an instance of the problem solved by the algorithm.
- The algorithm can be described in detail in natural language or pseudocode, or it can be implemented as a computer program.
- There are two kinds of algorithm complexity (efficiency) : time complexity and space complexity.

Summary

- There are three types of time complexity: worst, best, and average
- The complexity of most algorithms falls into several categories: constant (1), logarithmic (logn), linear (n), nearly linear (nlogn), square (n^2), cubic (n^3), exponential (m^n, n!). .
- 1, logn, n, nlogn, n^2, n^3 are collectively called polynomial time efficient algorithms or good algorithms; m^n, n! collectively known as exponential time invalid algorithms or