算法设计与分析

主讲教师：邵蓥侠

联系方式：shaoyx@bupt.edu.cn

个人主页：https://shaoyx.github.io/

课程微信群

2021 算法设计与分析

- 腾讯会议：
  - https://meeting.tencent.com/dm/aPpDvqjkkSHd?rs=25
- 会议 ID：
  - 372 4325 2473

课程安排

16周/32学时

助教1： 李鸿政 (301班，302班)

邮箱： ethan.lee.qnl@gmail.com

助教2： 连金清 （303班）

邮箱： jinqinglian@bupt.edu.cn

助教3： 王子泓 （304班）

邮箱： wzhyt1@163.com

周次	课程内容
1	第1章 算法引论
2	第1章 算法引论
3	第2章 分治与递归
4	第2章 分治与递归
5	第2章 分治与递归
6	第3章 动态规划
7	第3章 动态规划
8	第3章 动态规划
9	第3章 动态规划
10	第4章 贪心
11	第4章 贪心
12	第5章 回溯
13	第5章 回溯
14	第5章 回溯
15	第6章 分支界限
16	第6章 分支界限

课程线上交流


课程简介
算法设计与分析
课程主页
课程简介
教师介绍
教学安排
教学大纲
课程资料
交流互动
课程小组
作业习题
在线测试
课程工具
微助教-课堂互动

爱课堂

课程评分方法

- 平时成绩(40%)
  - 理论作业 (10%)
  - 编程实验 (30%)
- 期末考试 (60%)
  - 闭卷考试

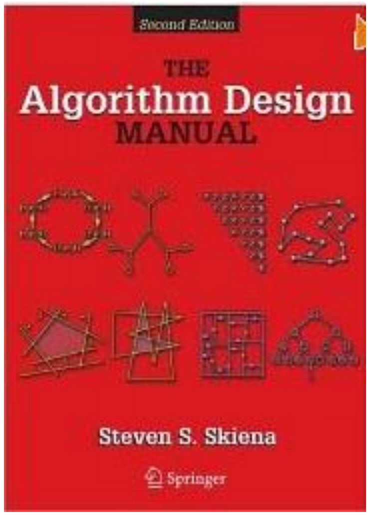
理论作业评分

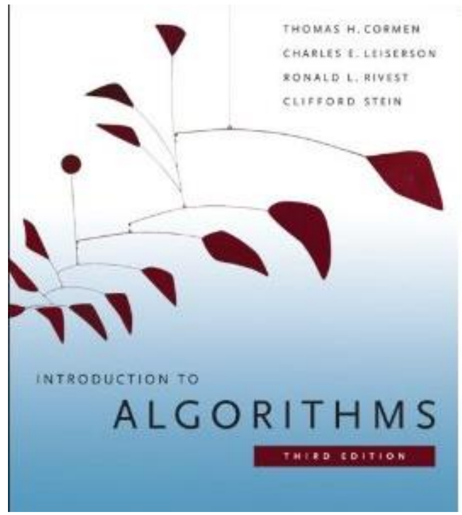
- 每次作业满分10分;
- 迟交罚扣为2 pts/wk；抄袭者0分;
- 做则有分，不计对错;
- 通过爱课堂提交。

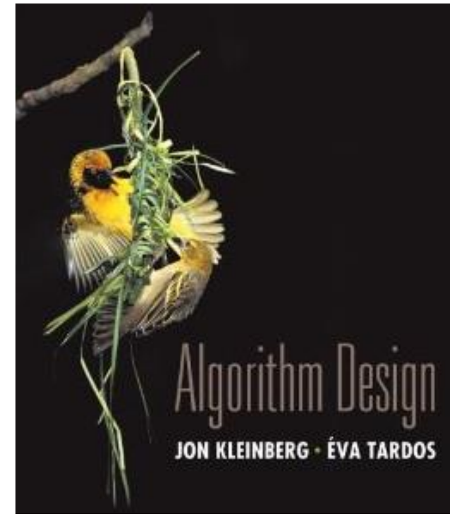
编程实验评分

- 算法代码建议使用C/C++;
- 迟交罚扣为 10% /day；抄袭者0分;
- 通过爱课堂提交。

参考书籍







选一本看！

不可贪婪 切忌！

第1章 算法引论

学习要点:

- 理解算法的概念。
- 理解什么是程序，程序与算法的区别和内在联系。
- 掌握算法的计算复杂性概念。
- 掌握算法渐近复杂性的数学表述。
- 掌握用C/C++语言描述算法的方法。

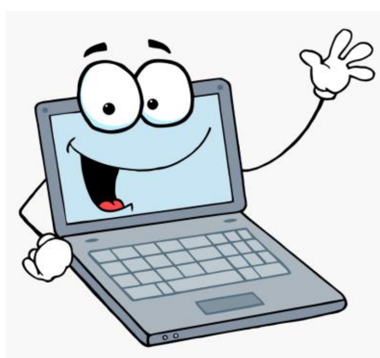
算法(Algorithm)

- 算法是指解决问题的一种方法或一个过程。

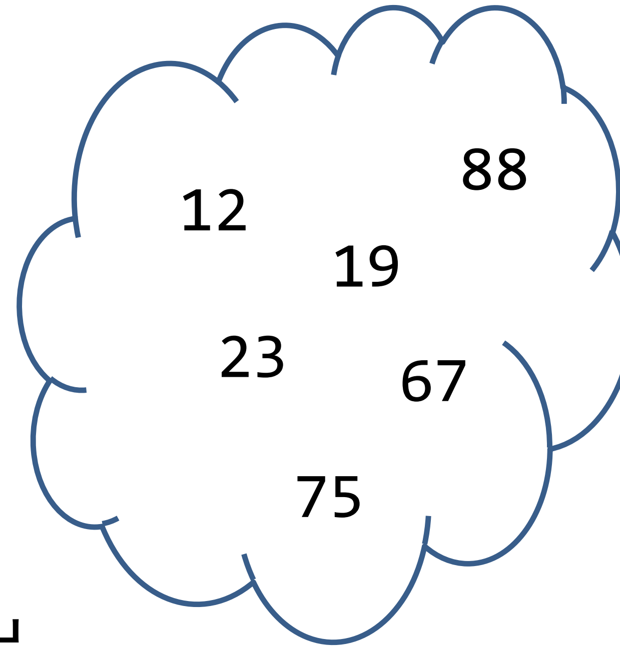
问题

算法

输入



输出



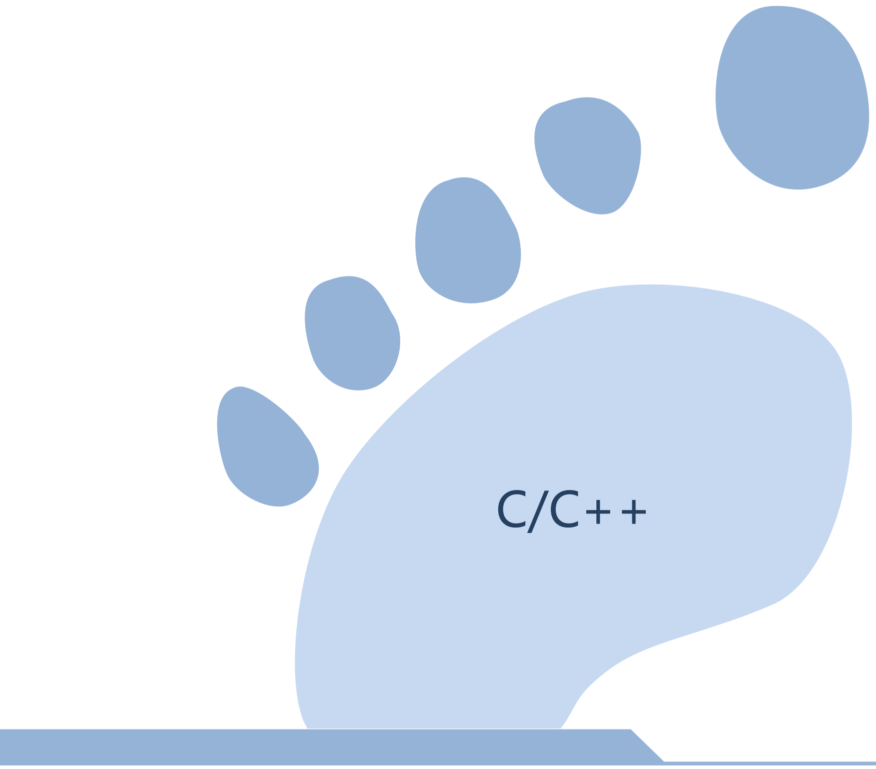
算法(Algorithm)

- 算法是指解决问题的一种方法或一个过程。
- 算法是若干指令的有穷序列，满足性质：
  1) **输入**：有外部提供的量作为算法的输入。
  2) **输出**：算法产生至少一个量作为输出。
  3) **确定性**：组成算法的每条指令是清晰，无歧义的。
  4) **有限性**：算法中每条指令的执行次数是有限的，执行每条指令的时间也是有限的。

程序(Program)

- 程序是算法用某种程序设计语言的具体实现。
- 程序可以*不满足*算法的*性质(4)有限性*。
  - 例如操作系统(OS)，是一个在无限循环中执行的程序，因而不是一个算法。
  - 操作系统的各种任务可看成是单独的问题，每一个问题由操作系统中的一个子程序通过特定的算法来实现。该子程序得到输出结果后便终止。

用C/C++描述算法



- C/C＋＋语言
  - 优点：大家学过，类型丰富，语句精炼，具有面向对象和面向过程的双重特点。
- C/C++语言与自然语言相结合

算法复杂性分析

算法的分析和评价

- ✓ 对于解决同一个问题，往往能够编写出许多不同的算法。
- ✓ 进行算法分析和评价的目的，即在于从解决同一问题的不同算法中选择较为合适的一种，也在于知道如何对现有算法进行改进，从而可能设计出更好的算法。

如何评价算法?

时间复杂性和空间复杂性

算法复杂性分析

- 算法复杂性 = 算法运行所需要的计算机资源的量
  - 需要时间资源的量称为**时间复杂性**;
  - 需要空间（存储器）资源的量称为**空间复杂性。**

- 算法的时间复杂性$T(n)$;
- 算法的空间复杂性$S(n)$。
- 其中n是问题的规模（输入大小）。

本课程所研究的算法复杂性

- 时间复杂性和空间复杂性的量计算方法相似，并且空间复杂性的计算更简单，因此本课程主要以时间复杂性计算为主。

时间复杂性 / 复杂度

- 又称**计算复杂度**，是一个算法运行时间的相对度量。

一个算法的运行时间是指在计算机上从开始到结束运行所花费的时间，它大致等于计算机执行一种简单操作（如赋值、比较、计算、转向、返回、输入、输出等）所需的时间与算法中进行简单操作次数的**乘积**。

由于执行一种简单操作所需的时间随着机器而异，它是机器本身硬软件环境决定的，与算法无关，所以我们只讨论影响运行时间的另一个因素——**算法中进行简单操作的次数。**

时间复杂性 / 复杂度

- 又称**计算复杂度**，是一个算法运行时间的相对度量。

不管一个算法是简单还是复杂，最终都是被分解成简单操作来具体执行，因此每个算法都对应着一定的简单操作的次数。显然，在一个算法中，进行简单操作的次数越少，其运行时间也就相对地减少；次数越多，其运行时间也就相对地越多。

所以，**通常把算法中包含简单操作次数的多少叫做算法的时间复杂度，用它来衡量一个算法的运行时间性能或称计算性能。**

时间复杂度函数*T(n)*

函数定义

若解决一个问题的规模为n，例如在排序问题中，n表示待排元素的个数；在矩阵计算中，n表示矩阵的阶数；在图的遍历中，n表示图中的顶点数。

T(n)

算法的时间复杂度就是n的一个函数，通常记作：T(n).

举例：累加求和

```
int Sum ( int b[], int n )
{
    int s = 0;
    for ( int i =0; i < n; i ++ )
        s += b[i];
    return s;
}
```

$T(n) = 4n + 4$

```
       int i = 0                     //1次
mark1: if( i >= n) goto mark 2;     //n+1次
       s += b[i];                    //n次
       i ++;                         //n次
       goto mark1;                   //n次
mark2: return s;
```

举例：简单选择排序

```
void SelectSort ( int b[], int n )
{
    int i, j, k, x;
    for ( i=0; i <n-1; i ++)
    {
        k = i;
        for ( j=i+1; j<n; j++)
            if (b[k] > b[j]) k =j;
        x = b[j];
        b[i] = b[k];
        b[k] = x;
    }
}
```

```
void SelectSort ( int b[], int n )
{
    int i, j, k, x;
    for ( i=0; i <n-1; i ++)
    {
        k = i;
        for ( j=i+1; j<n; j++)
            if (b[k] > b[j]) k =j;
        x = b[j];
        b[i] = b[k];
        b[k] = x;
    }
}
```

```
       i = 0;  //1次
mark1: if(!(i < n-1)) goto mark4;  //n次
       k =j ;    //n-1次
       j=i+1;    //n-1次
mark2: if (!(j<n)) goto mark3; //(n+2)(n-1)/2次
       if (b[k] > b[j]) k =j;  //n(n-1)/2次
       j++;  //n(n-1)/2次
       goto mark2;  //n(n-1)/2次

mark3: x = b[j];     //n-1次
       b[i] = b[k];  //n-1次
       b[k] = x;     //n-1次
       i++;          //n-1次
       goto mark1; //n-1次
mark4:
```

$T(n) = 2n^2+7n-7$

算法的时间复杂性

（1）**最坏情况**下的时间复杂性

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

（2）**最好情况**下的时间复杂性

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

（3）**平均情况**下的时间复杂性

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

其中，I是问题的规模为n的实例，$p(\mathrm{I})$是实例I出现的概率。

算法复杂性分析

渐进性复杂性分析

What to Analyze

- Machine & compiler-dependent run times.
- Time & space complexities : machine & compiler-independent.

• 假设(Assumptions):

① instructions are executed sequentially

② each instruction is simple, and takes exactly one time unit

③ integer size is fixed and we have infinite memory

• Typically the following two functions are analyzed:

$T_{avg}(N)$ & $T_{worst}(N)$ -- the average and worst case time complexities, respectively, as functions of input size N.

If there is more than one input, these functions may have more than one argument.

What to Analyze

〖Example〗 Matrix addition

```
void  add ( int  a[ ][ MAX_SIZE ],
            int  b[ ][ MAX_SIZE ],
            int  c[ ][ MAX_SIZE ],
            int  rows,  int  cols )
{
    int  i,  j ;
    for ( i = 0; i < rows; i++ )     /* rows + 1 */
        for ( j = 0; j < cols; j++ )     /* rows(cols+1) */
            c[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ];     /* rows · cols */
}
```

A: Exchange rows and cols.

$T(\text{rows, cols}) = 2\,\text{rows} \cdot \text{cols} + 2\text{rows} + 1$

What to Analyze

〖Example〗 Iterative function for summing a list of numbers

```
float  sum ( float  list[ ],  int  n )
{  /* add a list of numbers */
   float  tempsum = 0;   /* count = 1 */
   int  i ;
   for ( i = 0; i < n; i++ )
           /* count ++ */
     tempsum  += list [ i ] ;  /* count ++ */
    /* count ++ for last execution of for */
   return  tempsum;    /* count ++ */
}
```

$$T_{sum}(n) = 2n + 3$$

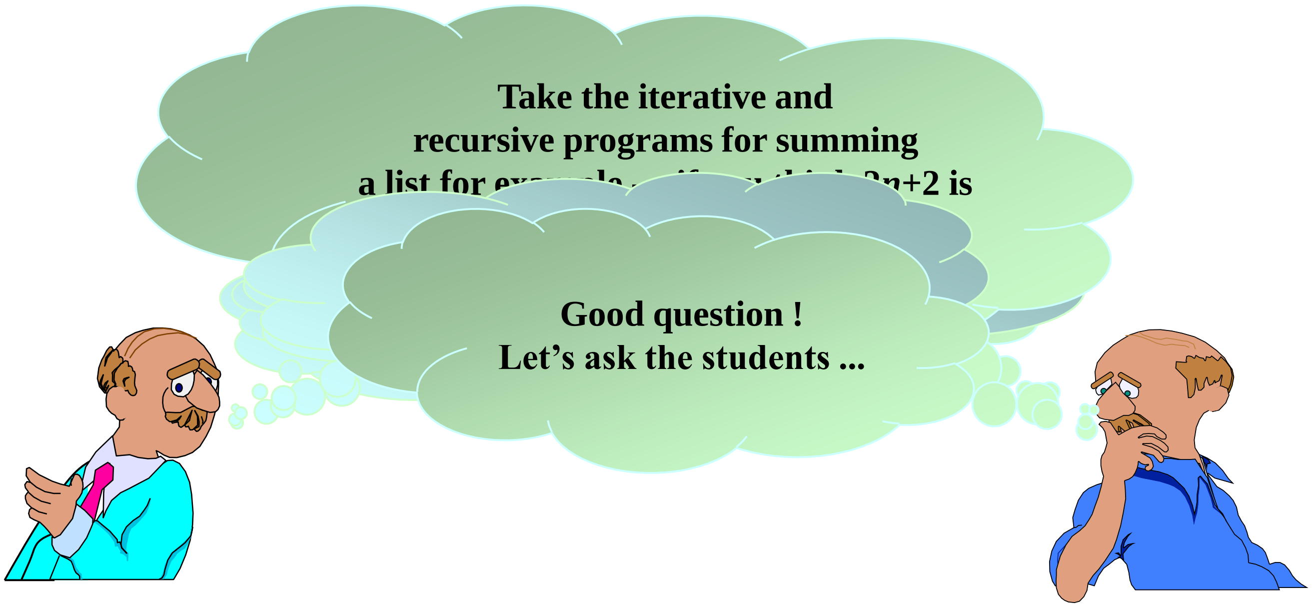
〖Example〗 Recursive function for summing a list of numbers

```
float  rsum ( float  list[ ],  int  n )
{  /* add a list of numbers */
   if ( n )  /* count ++ */
      return rsum(list, n–1) + list[n – 1];
       /* count ++ */
   return   0;  /* count ++ */
}
```

$$T_{rsum}(n) = 2n + 2$$

But it takes more time to compute each step.

What to Analyze

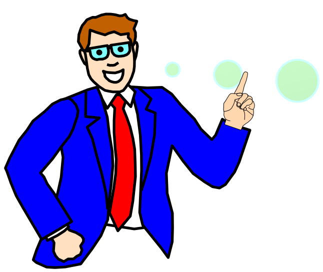


Asymptotic Notation（ O, Ω, Θ, o ）渐进性原理及表示符号

The point of counting the steps is to predict the growth in run time as the N change, and thereby compare the time complexities of two programs. So what we really want to know is the asymptotic behavior of T_p.

Suppose $T_{p1}(N) = c_1N^2 + c_2N$ and $T_{p2}(N) = c_3N$. Which one is faster?

No matter what c_1, c_2, and c_3 are, there will be an n_0 such that $T_{p1}(N) > T_{p2}(N)$ for all $N > n_0$.



I see! So as long as I know that T_{p1} is about N^2 and T_{p2} is about N, then for sufficiently large N, P2 will be faster!

算法渐近复杂性

- **衡量算法复杂性的规模 — 阶，Rate/Order of growth**
- **对时间复杂性函数$T(n)$，有可能，$T(n) \to \infty$，as $n \to \infty$；**
- 例：$T(n) = 2n^2 + 4n + 1$， $t(n)= 2n^2$
  - $(T(n) - t(n))/ T(n) \to 0$，as $n \to \infty$;
  - $t(n)$是$T(n)$的渐近性态，为算法的渐近复杂性。
  - 在数学上，$t(n)$是$T(n)$的渐近表达式，是$T(n)$略去低阶项留下的主项。它比$T(n)$ 简单。

渐近分析的记号

在下面的讨论中，对所有n，$f(n) \geq 0$，$g(n) \geq 0$。

（1）**渐近上界记号*O***

$O(g(n))$ = { $f(n)$ | 存在正常数c和n_0使得对所有$n \geq n_0$有：$0 \leq f(n) \leq cg(n)$ }

（2）**渐近下界记号Ω**

$\Omega(g(n))$ = { $f(n)$ | 存在正常数c和n_0使得对所有$n \geq n_0$有：$0 \leq cg(n) \leq f(n)$ }

渐近分析的记号

（3）**非紧上界记号*o***

$o(g(n)) = \{ f(n) \mid$ 对于任何正常数$c>0$，存在正数和$n_0>0$使得对所有$n \geq n_0$有：$0 \leq f(n) < cg(n) \}$

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

（4）**非紧下界记号*ω***

$\omega(g(n)) = \{ f(n) \mid$ 对于任何正常数$c>0$，存在正数和$n_0>0$使得对所有$n \geq n_0$有：$0 \leq cg(n) < f(n) \}$

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

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

渐近分析的记号

（5）**紧渐近界记号Θ**

$\Theta(g(n)) = \{ f(n) \mid$ 存在正常数c_1, c_2和n_0使得对所有$n \ge n_0$有：$c_1 g(n) \le f(n) \le c_2 g(n) \}$

定理1： $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

渐近分析记号在等式和不等式中的意义

$f(n)=\Theta(g(n))$的确切意义是：$f(n)\in\Theta(g(n))$。

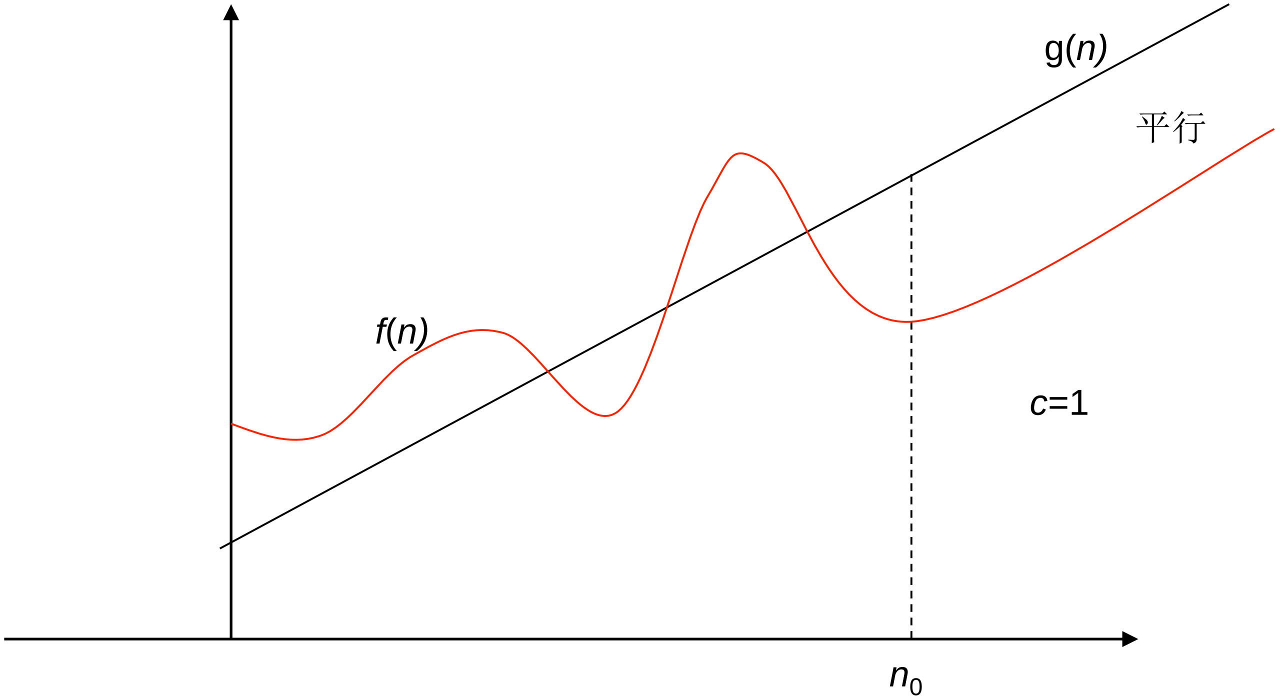
一般情况下，等式和不等式中的渐近记号$\Theta(g(n))$表示$\Theta(g(n))$中的某个函数。

例如：$2n^2+3n+1=2n^2+\Theta(n)$ 表示

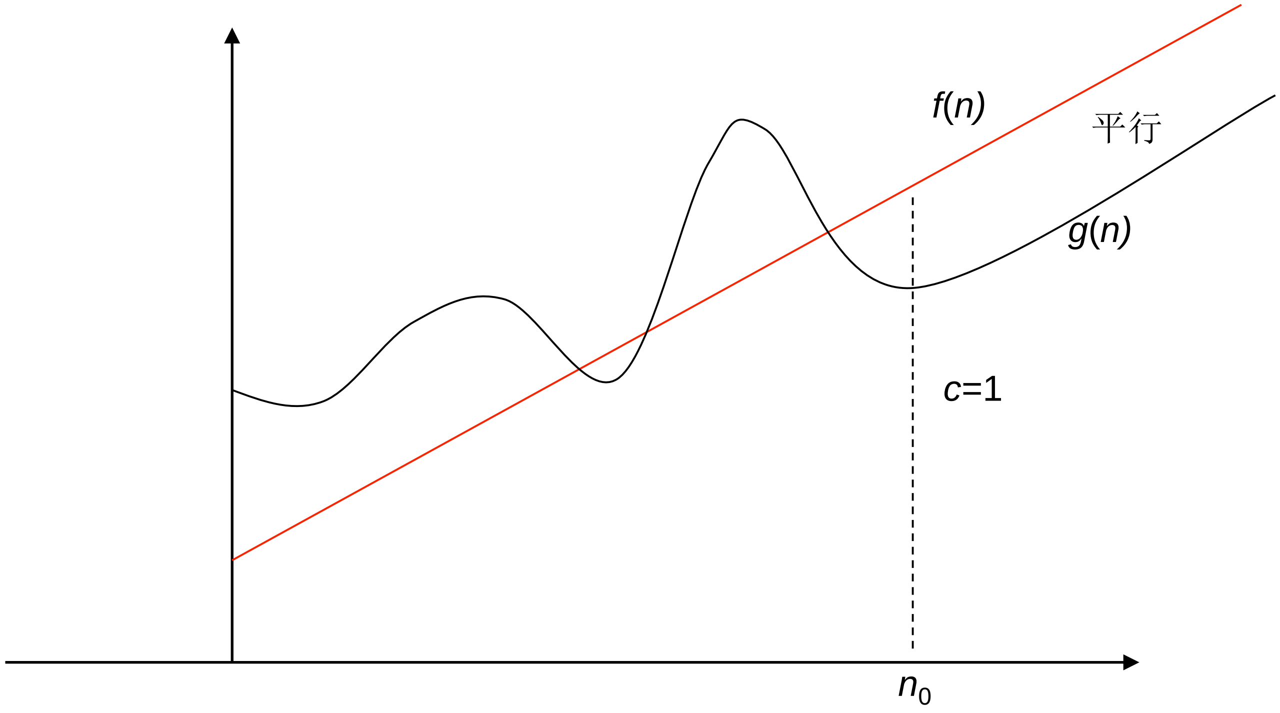
- $2n^2+3n+1=2n^2+f(n)$，其中$f(n)$ 是$\Theta(n)$中某个函数。

等式和不等式中渐近记号O, o, Ω和ω的意义是类似的。

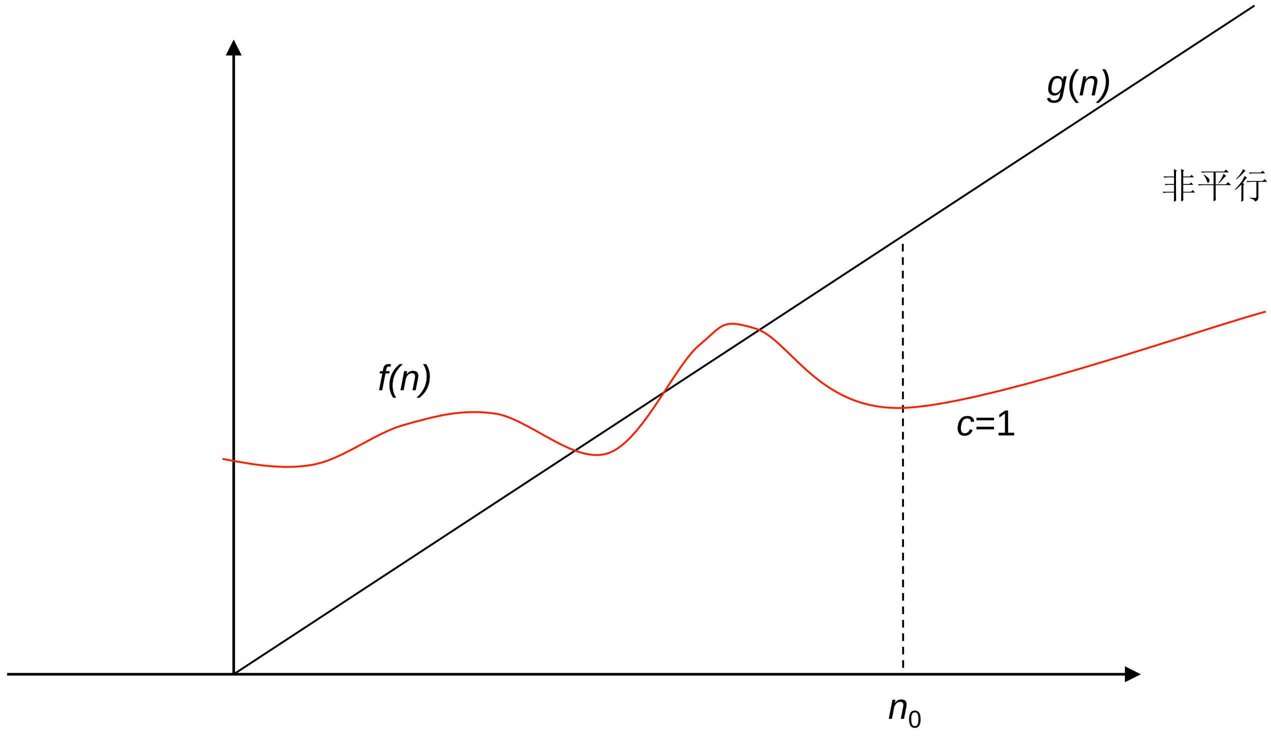
渐近上界记号*O*



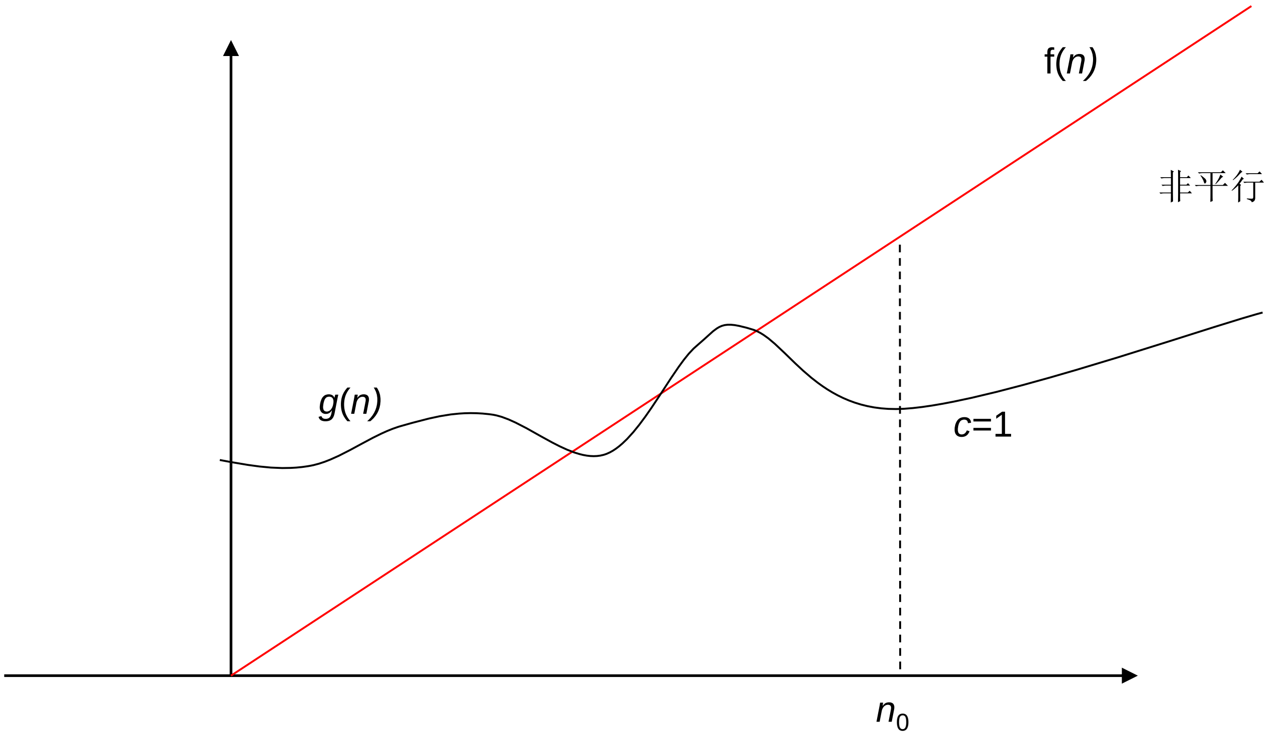
渐近下界记号Ω



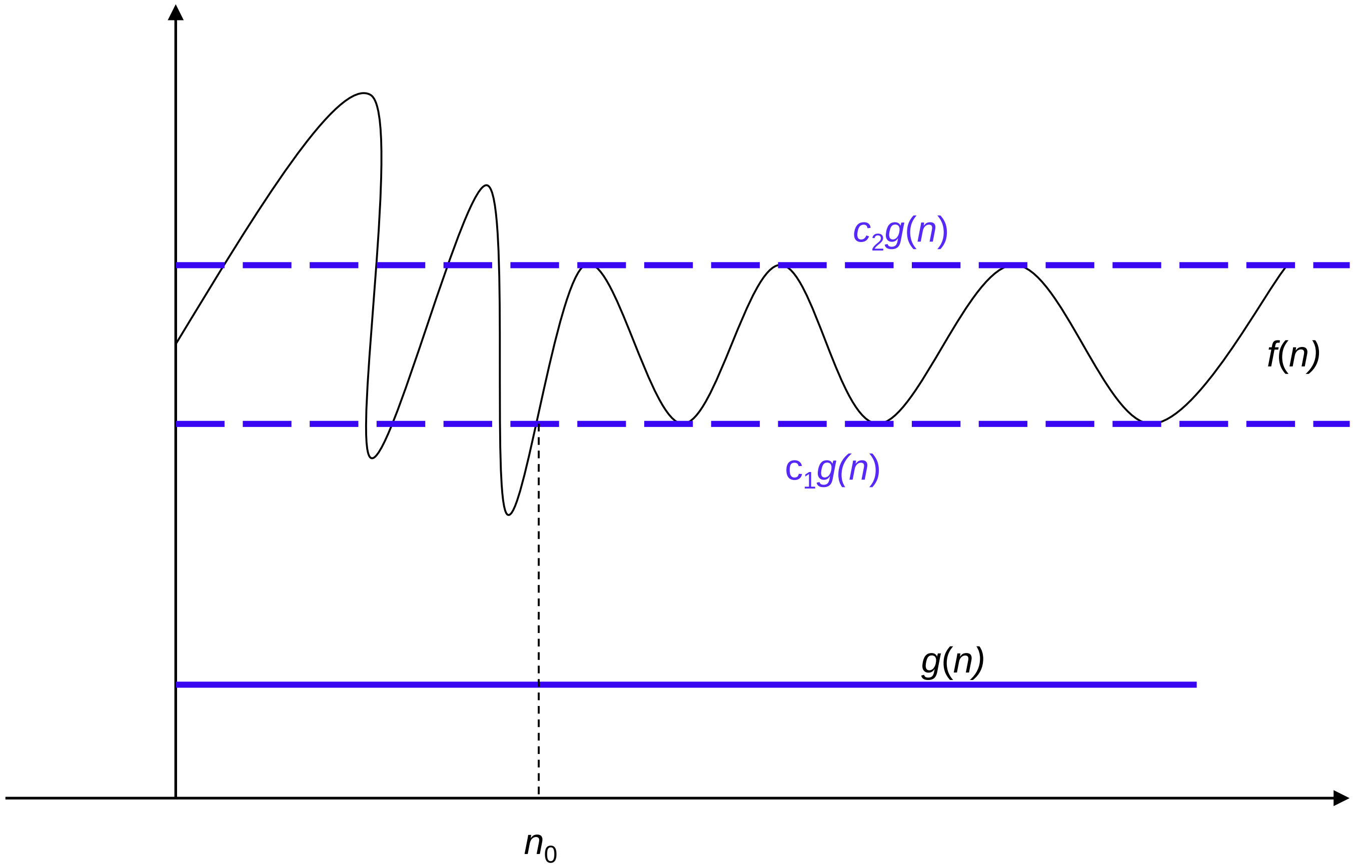
非紧上界记号o



非紧下界记号ω



紧渐近界记号Θ



注意

- $2N + 3 = O(N) = O(N^{k\geq 1}) = O(2^N) = \cdots$

We shall always take the smallest $f(N)$.

- $2^N + N^2 = \Omega(2^N) = \Omega(N^2) = \Omega(N) = \Omega(1) = \cdots$

We shall always take the largest $g(N)$.

渐近分析中函数比较

- $f(n) = O(g(n)) \approx \mathrm{a} \le \mathrm{b};$
- $f(n) = \Omega(g(n)) \approx \mathrm{a} \ge \mathrm{b};$
- $f(n) = \Theta(g(n)) \approx \mathrm{a} = \mathrm{b};$
- $f(n) = o(g(n)) \approx \mathrm{a} < \mathrm{b};$
- $f(n) = \omega(g(n)) \approx \mathrm{a} > \mathrm{b}.$

渐近分析记号的若干性质

（1）传递性：

- $f(n)=\Theta(g(n))$， $g(n)=\Theta(h(n)) \Rightarrow f(n)=\Theta(h(n))$；
- $f(n)=O(g(n))$， $g(n)=O(h(n)) \Rightarrow f(n)=O(h(n))$；
- $f(n)=\Omega(g(n))$， $g(n)=\Omega(h(n)) \Rightarrow f(n)=\Omega(h(n))$；
- $f(n)=o(g(n))$， $g(n)=o(h(n)) \Rightarrow f(n)=o(h(n))$；
- $f(n)=\omega(g(n))$， $g(n)=\omega(h(n)) \Rightarrow f(n)=\omega(h(n))$；

渐近分析记号的若干性质

（2）反身性：

- $f(n)=\Theta(f(n))$；
- $f(n)=O(f(n))$；
- $f(n)=\Omega(f(n))$.

（3）对称性：

- $f(n)=\Theta(g(n)) \Leftrightarrow g(n)=\Theta(f(n))$.

（4）互对称性：

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

渐近分析记号的若干性质

（5）算术运算：

- $O(f(n))+O(g(n)) = O(\max\{f(\mathrm{n}), g(n)\})$；
- $O(f(n))+O(g(n)) = O(f(\mathrm{n})+g(n))$；
- $O(f(n))*O(g(n)) = O(f(\mathrm{n})*g(n))$；
- $O(cf(n)) = O(f(\mathrm{n}))$；
- $g(n)= O(f(n)) \Rightarrow O(f(n))+O(g(n)) = O(f(\mathrm{n}))$。

证明举例

规则O(f(n))+O(g(n)) = O(max{f(n), g(n)}) 的证明:

- 对于任意$f_1(n) \in O(f(n))$，存在正常数c_1和自然数n_1，使得对所有$n \geq n_1$，有$f_1(n) \leq c_1 f(n)$。
- 类似地，对于任意$g_1(n) \in O(g(n))$，存在正常数c_2和自然数n_2，使得对所有$n \geq n_2$，有$g_1(n) \leq c_2 g(n)$。
- 令$c_3=\max\{c_1, c_2\}$，$n_3 =\max\{n_1, n_2\}$，$h(n)= \max\{f(\mathrm{n}), g(n)\}$。
- 则对所有的 $n \geq n_3$，有
- $f_1(n) +g_1(n) \leq c_1 f(n) + c_2 g(n)$

$\leq c_3 f(n) + c_3 g(n)= c_3(f(n) + g(n))$

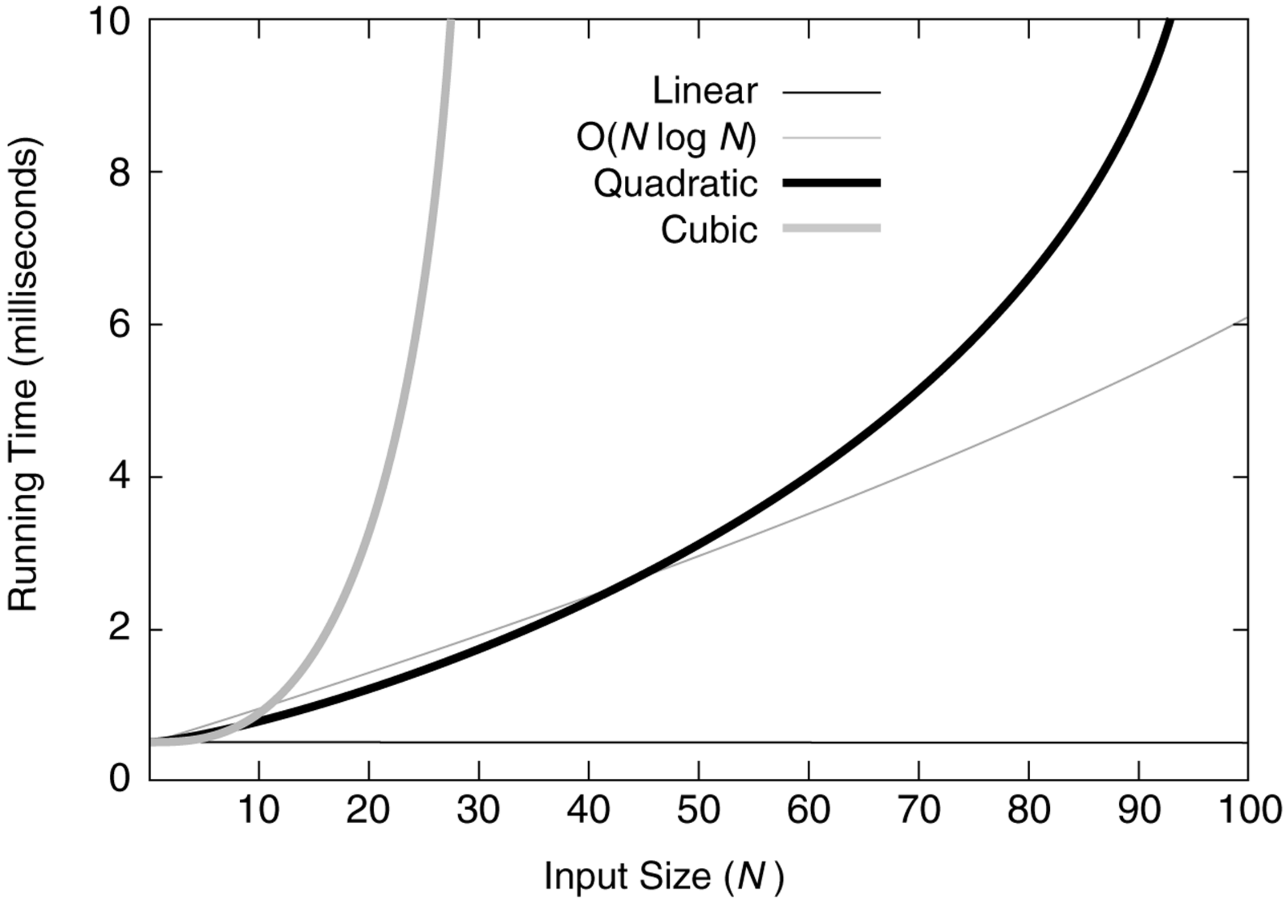
$\leq c_3 2 \max\{f(\mathrm{n}), g(n)\}$

$= 2c_3 h(n) = O(\max\{f(\mathrm{n}),g(n)\})$.

算法分析中常见的复杂性函数

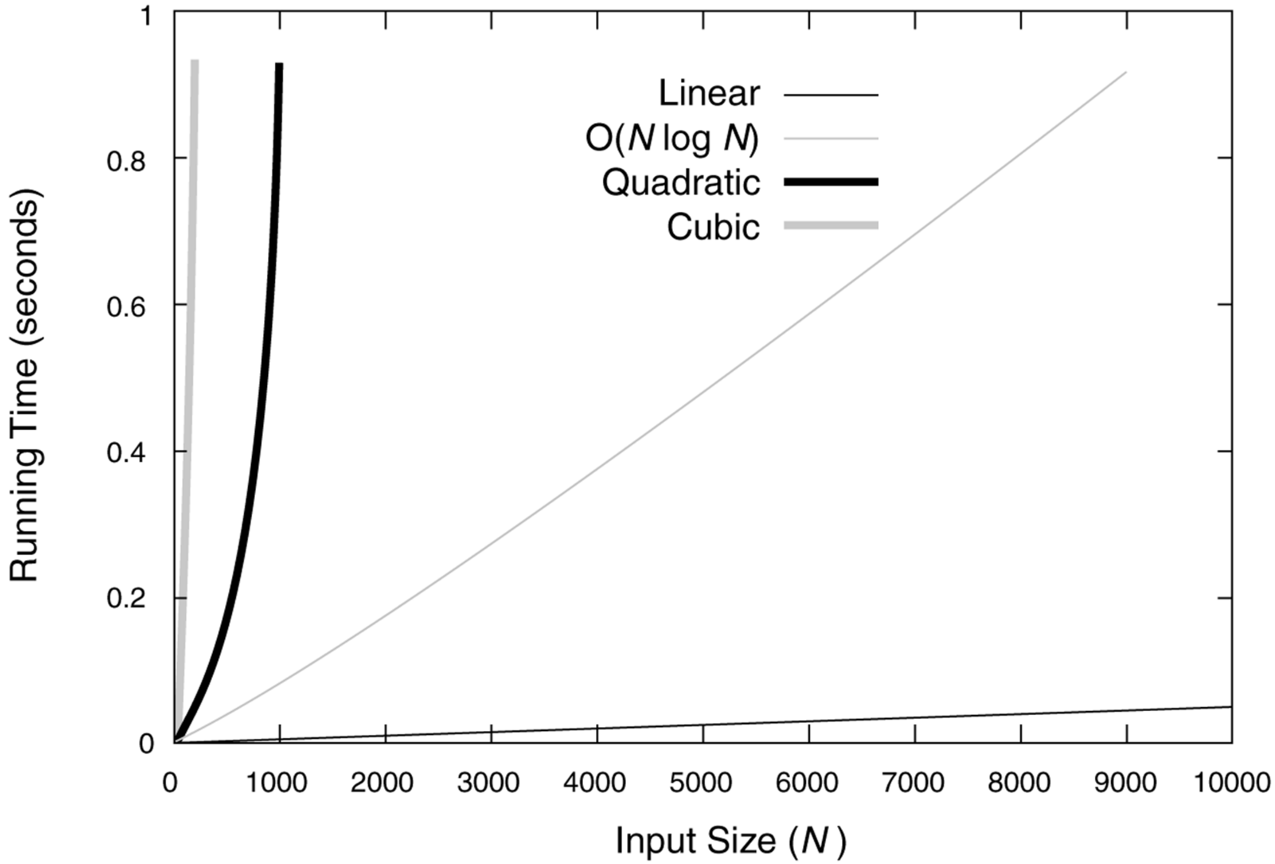
FUNCTION	NAME
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

小规模数据



Linear
O(N log N)
Quadratic
Cubic
Running Time (milliseconds)
Input Size (N)
0
2
4
6
8
10
10
20
30
40
50
60
70
80
90
100

中等规模数据



Linear
O(N log N)
Quadratic
Cubic
Running Time (seconds)
0
0.2
0.4
0.6
0.8
1
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Input Size (N)