

算法设计与分析

刘钦源

liuqy@tongji.edu.cn

为什么要学习算法？

有两种思想，就像摆放在天鹅绒上的宝石那样熠熠生辉，一个是微积分，另一个就是算法。微积分以及在微积分基础上建立起来的数学分析体系造就了现代科学，而算法则造就了现代世界。

——大卫·柏林斯基《算法的诞生》

为什么要学习算法？

受过良好训练的计算机科学家知道怎样处理算法：如何构造算法、操作算法、理解算法以及分析算法。这些知识远不止为了写出良好的计算机程序而准备的。算法是一种一般性的智能工具，它必定有助于我们对其他学科的理解，不管是化学、语言学或音乐，还是其他学科。为什么算法会有这种作用呢？我们可以这样理解：人们常说，一个人只有把知识教给别人，才能真正掌握它。实际上，一个人只有把知识教给“计算机”，才能“真正”掌握它，也就是说，将知识表述为一种算法……比起简单按照常规去理解事物，尝试用算法将其形式化能使我们的理解更加深刻。

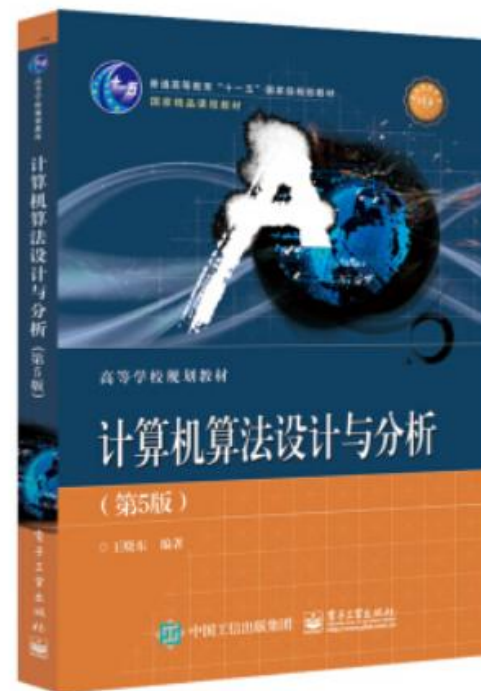
——唐纳德·克努特

课程目标

- 了解计算机应用中的各种常用算法
- 掌握设计和分析各种算法的基本原理、方法和技巧
- 能运用所学到的知识熟练地分析各种算法，并能指出解决同一问题的各种算法的好坏。

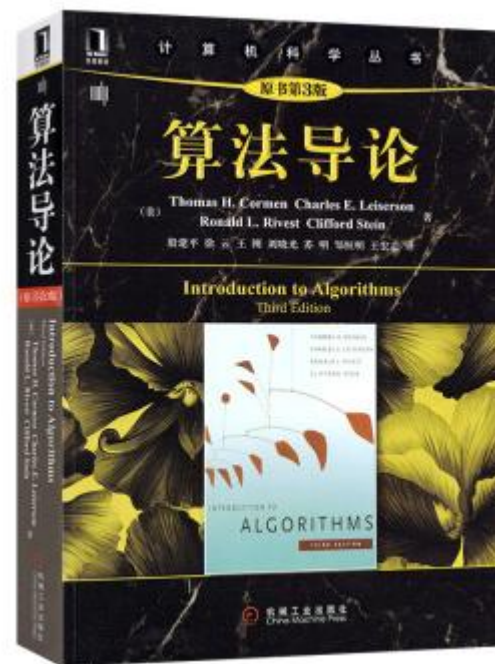
教材

- 王晓东，计算机算法设计与分析（第4、5版），电子工业出版社，2012，2018



课外阅读书

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, Third Edition, The MIT Press, 2009



课程考核方式

考勤 10%

平时作业 30%

期中考试 10%

期末考试 50%

课程QQ群：759352950

命名规则：姓名 + 学号



群名称:算法设计与分析 2022-2023...
群 号:759352950

第1章 算法概述

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。

算法(Algorithm) ——理解算法的概念

最大公约数: 两个不全为 0 的非负整数 m 和 n 的最大公约数记为 $\gcd(m,n)$, 代表能够整除 (即余数为 0)

1-欧几里得算法 (Euclid's algorithm)

$$\gcd(m,n) = \gcd(n, m \bmod n)$$

$m \bmod n$ 表示 m 除以 n 之后的余数

$$\gcd(m,0)=m$$

$$\gcd(60,24)= \gcd(24, 12)= \gcd(12,0)= 12$$



《几何原本》

算法(Algorithm)

欧几里得算法 (Euclid's algorithm)

第一步： 如果 $n=0$ ，返回 m 的值作为结果，同时过程结束；
否则，进入第二步。

第二步： m 除以 n ，将余数赋给 r 。

第三步： 将 n 的值赋给 m ，将 r 的值赋给 n ，返回第一步

算法 Euclid(m,n)

//使用欧几里得算法计算 $\text{gcd}(m,n)$

//输入： 两个不全为0的非负整数 m,n

//输出： m, n 的最大公约数

while $n \neq 0$ **do**

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

算法(Algorithm)

2-基于最大公约数的定义： m 和 n 的最大公约数就是能够同时整除它们的最大正整数。

第一步：将 $\min\{m,n\}$ 的值赋给 t 。

第二步： m 除以 t 。如果余数为 0，进入第三步;否则，进入第四步。

第三步： n 除以 t 。如果余数为 0，返回 t 的值作为结果;否则，进入第四步。

第四步：把 r 的值减1。返回第二步。

当一个输入为0时？

算法(Algorithm)

3-中学时计算 $\gcd(m,n)$

$$60 = 2 \times 2 \times 3 \times 5$$

$$24 = 2 \times 2 \times 2 \times 3$$

$$\gcd(60, 24) = 2 \times 2 \times 3 = 12$$

第一步：找到 m 的所有质因数。

第二步：找到 n 的所有质因数。

第三步：从第一步和第二步求得的质因数分解式找出所有的公因数（如果 p 是一个公因数，而且在 m 和 n 的质因数分解式分别出现过 p_m 和 p_n 次，那么应该将 p 重复 $\min\{p_m, p_n\}$ 次）。

第四步：将第三步中找到的质因数相乘，其结果作为给定数字的最大公约数。

能否称之为算法？

算法概念

- 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。

算法(Algorithm)

- 算法的每一个步骤都必须没有歧义，不能有半点儿含糊。
- 必须认真确定算法所处理的输入的值域。
- 同一算法可以用几种不同的形式来描述。
- 同一问题，可能存在几种不同的算法。
- 针对同一问题的算法可能基于完全不同的解题思路，而且解题速度也会有显著不同。

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)。
 - 例如操作系统，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。
-

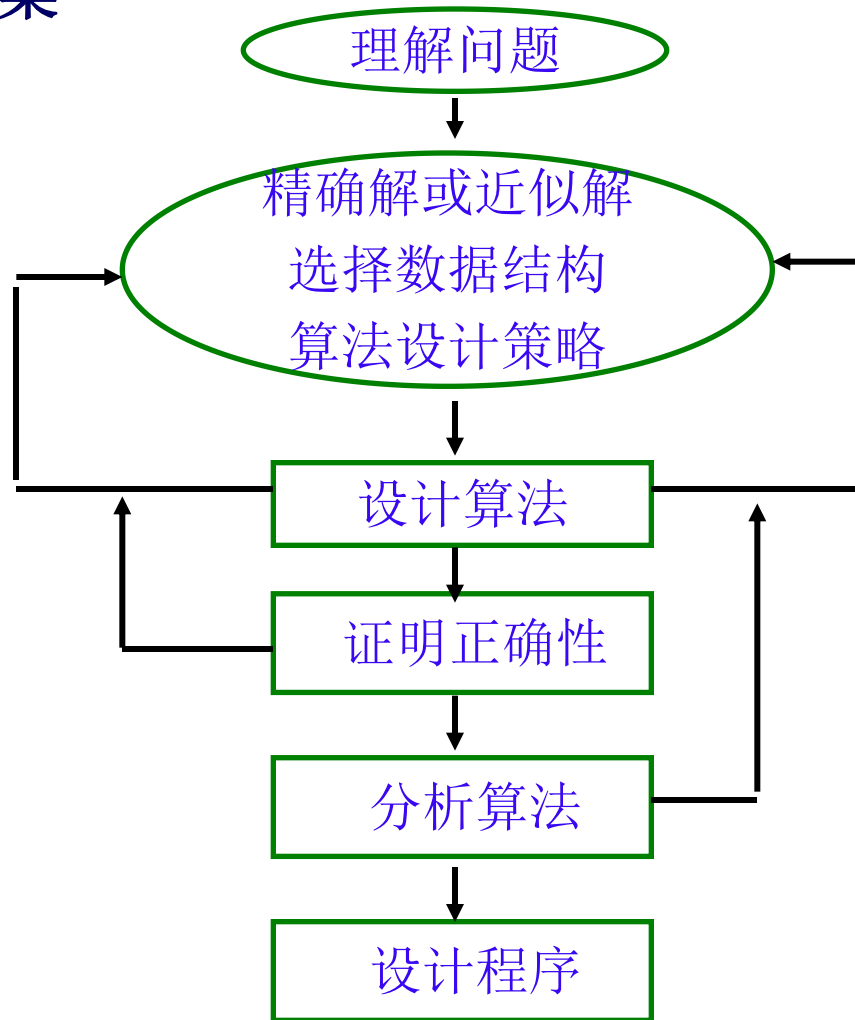
问题求解(Problem Solving)

算法是问题的程序化解决方案

正确的算法不仅应该能处理大多数常见情况，而且应该能正确处理所有合法的输入

重要的问题在很多情况下的确无法求得精确解，例如求平方根、解非线性方程和求定积分

证明正确性的一般方法是数学归纳法；证明不正确只需要一个反例



算法复杂性分析

- 算法复杂性 = 算法所需要的计算机资源
- 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）。

时间复杂性 VS 空间复杂性？

算法复杂性分析

事后统计

将算法先实现，直接将算法程序在计算机上运行，测算其时间和空间的开销

事前分析

对算法所消耗的资源进行估算，具体讲如何进行估算？

算法运行时间= 每条语句频度之和 * 该语句执行一次所需的时间

算法复杂性分析

for(i=1;i<=n;i++)	n+1
for(j=1;j<=n;j++){	n(n+1)
c[i][j]=0;	n*n
for(k=0;k<n;k++)	n*n*(n+1)
c[i][j]=c[i][j]+a[i][k]*b[k][j];	n*n*n
}	

把算法所耗费的时间定义为该算法中每条语句的频度之和把所有语句的执行次数全部加起来

$$T(n) = 2n^3 + 3n^2 + 2n + 1$$

算法的时间复杂性

顺序查找算法复杂性？

和具体输入有关

(1) 最坏情况下的时间复杂性

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

(2) 最好情况下的时间复杂性

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

(3) 平均情况下的时间复杂性

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

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

顺序查找算法复杂性

(1) $T_{\max}(n) = \max\{ T(I) \mid \text{size}(I)=n \} = O(n)$

(2) $T_{\min}(n) = \min\{ T(I) \mid \text{size}(I)=n \} = O(1)$

(3) 在平均情况下，假设：

(a) 搜索成功的概率为 p ($0 \leq p \leq 1$)；

(b) 在数组的每个位置 i ($0 \leq i < n$) 搜索成功的概率相同，均为 p/n 。

$$\begin{aligned} T_{\text{avg}}(n) &= \sum_{\text{size}(I)=n} p(I)T(I) \\ &= \left(1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + 3 \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n} \right) + n \cdot (1 - p) \\ &= \frac{p}{n} \sum_{i=1}^n i + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p) \end{aligned}$$

- 最优效率分析不如最差效率分析重要
- 算法的最优效率都无法满足要求，则不需要考虑
- 平均效率虽然有意义，但是难以计算，需要输入特征的概率分布

算法渐近复杂性

不一定要计算精确执行次数，只需要大概执行次数

$$T(n) = 2n^3 + 3n^2 + 2n + 1$$

$T(n) \rightarrow \infty$, as $n \rightarrow \infty$; $(T(n) - t(n)) / T(n) \rightarrow 0$, as $n \rightarrow \infty$;

- $t(n)$ 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 Θ , O , Ω , o , ω

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

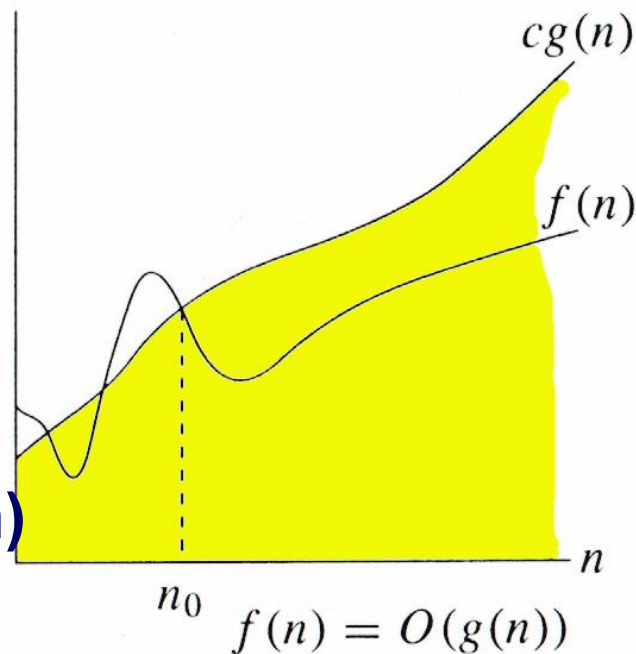
$O(g(n)) = \{ f(n) \mid \text{存在正常数 } c \text{ 和 } n_0 \text{ 使得对所有 } n \geq n_0 \text{ 有:}$

$$0 \leq f(n) \leq cg(n) \}$$

$$2n + 3 = O(n^2).$$

$$3n^3 = O(n^4)$$

$O(g(n))$ 是增长次数小于或等于 $g(n)$ 的函数集合



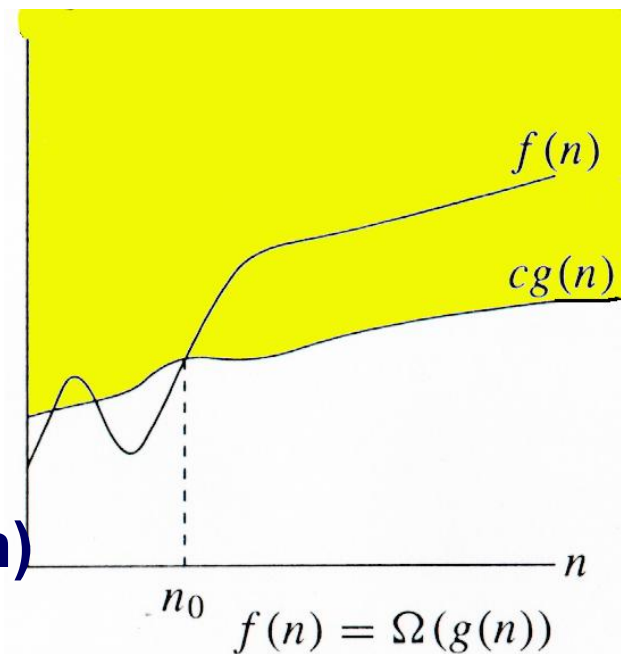
- 用来作比较的函数 $g(n)$ 应该尽量接近所考虑的函数 $f(n)$. $2n+3=O(n^2)$ 松散的界限; $2n+3=O(n)$ 较好的界限。
- $f(n)=O(g(n))$ 不能写成 $g(n)=O(f(n))$, 因为两者并不等价。实际上, 这里的等号并不是通常相等的含义。按照定义, 用集合符号更准确些。
- $O(g(n))=\{f(n)|f(n)\text{满足: 存在正的常数}c\text{和}n_0, \text{使得当}n\geq n_0\text{时}f(n)\leq cg(n)\}$ 所以, 人们常常把 $f(n)=O(g(n))$ 读作: “ $f(n)$ 是 $g(n)$ 的一个大O成员”。

(2) 渐近下界记号 Ω

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

$$3n^3 = \Omega(n^2)$$

$\Omega(g(n))$ 是增长次数大于或等于 $g(n)$ 的函数集合



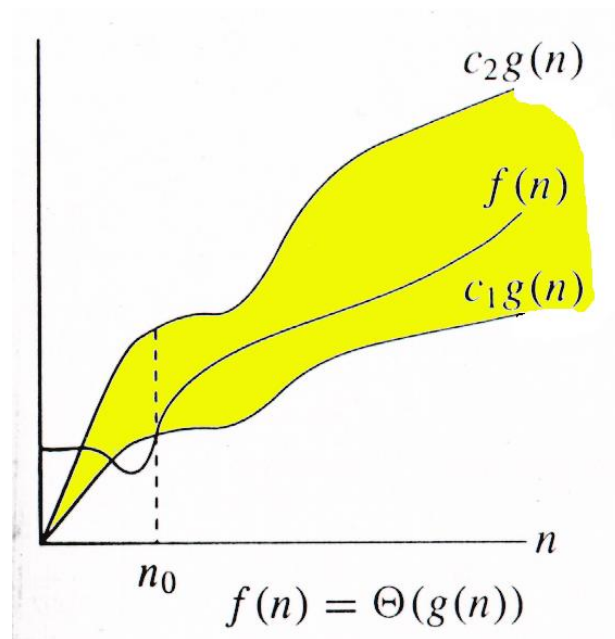
(3) 紧渐近界记号 Θ

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

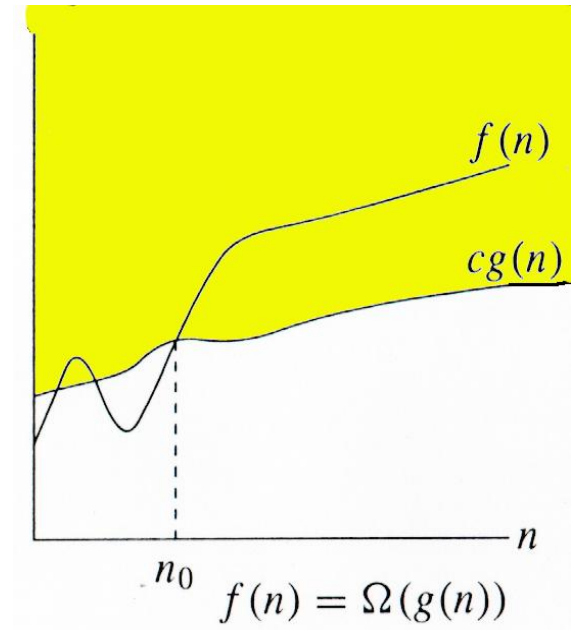
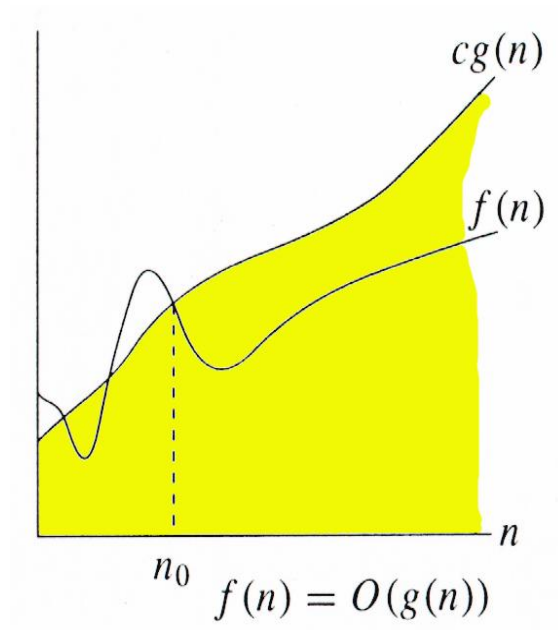
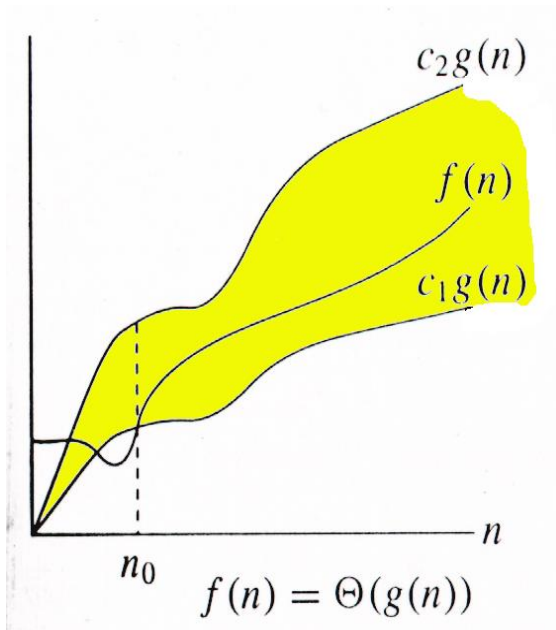
定理1: $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

$$0.5n(n-1) = \Theta(n^2)$$

$\Theta(g(n))$ 是增长次数等于 $g(n)$ 的函数集合



Θ , O , Ω 之间的关系



Big-O

略去低阶项和常数系数项留下的主项

略去低阶项

- $4n + 5 \Rightarrow 4n$
- $0.5 n \log n - 2n + 7 \Rightarrow 0.5 n \log n$

略去常数系数项

- $4n \Rightarrow n$
- $0.5 n \log n \Rightarrow n \log n$
- $\log n^2 = 2 \log n \Rightarrow \log n$
- $\log_3 n = (\log_3 2) \log n \Rightarrow \log n$

$$2n^2 + 4n = O(n^2) \quad \checkmark$$

$$O(n^2) = 2n^2 + 4n \quad \times$$

Big-O 实例

$$n^2 + 100n = O(n^2)$$

$$(n^2 + 100n) \leq 2n^2 \quad \text{for } n \geq 10$$

$$n^2 + 100n = \Omega(n^2)$$

$$(n^2 + 100n) \geq 1n^2 \quad \text{for } n \geq 0$$

$$n^2 + 100n = \Theta(n^2)$$

$$n \log n = O(n^2)$$

$$n \log n = \Theta(n \log n)$$

$$n \log n = \Omega(n)$$

- ♦ 插入排序在最坏的情况下需要 $\Theta(n^2)$ ，所以排序是 $O(n^2)$
- ♦ 任意的排序算法都需要查看每个元素，所以排序是 $\Omega(n)$.
- ♦ 实际上，合并排序在最坏的情况下是 $\Theta(n \log n)$

(4) 非紧上界记号 o

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

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

(5) 非紧下界记号 ω

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

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

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

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。

渐近分析中函数比较

$$f(n) = O(g(n)) \approx a \leq b;$$

$$f(n) = \Omega(g(n)) \approx a \geq b;$$

$$f(n) = \Theta(g(n)) \approx a = b;$$

$$f(n) = o(g(n)) \approx a < b;$$

$$f(n) = \omega(g(n)) \approx a > b.$$

渐近分析记号的若干性质

(1) 传递性:

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

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

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

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

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

(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:

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

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

(5) 算术运算:

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

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)) ;$$

$$O(f(n)) * O(g(n)) = O(f(n) * g(n)) ;$$

$$O(cf(n)) = O(f(n)) ;$$

$$g(n) = O(f(n)) \Rightarrow O(f(n)) + O(g(n)) = O(f(n)) .$$

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 $c_3 = \max\{c_1, c_2\}$ ， $n_3 = \max\{n_1, n_2\}$ ， $h(n) = \max\{f(n), g(n)\}$ 。

则对所有的 $n \geq n_3$ ，有

$$\begin{aligned} f_1(n) + g_1(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq c_3 f(n) + c_3 g(n) = c_3 (f(n) + g(n)) \\ &\leq c_3 2 \max\{f(n), g(n)\} \\ &= 2c_3 h(n) = O(\max\{f(n), g(n)\}) . \end{aligned}$$

有什么用？

由两个连续执行部分组成的算法

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

第一部分，应用某种已知的排序算法对数组排序；

第二部分，连续扫描该有序数组的元素，比较是否和指定元素相等

假设第一部分使用的排序算法的比较次数不会超过 $n(n-1)$ ，属于集合 $O(n^2)$

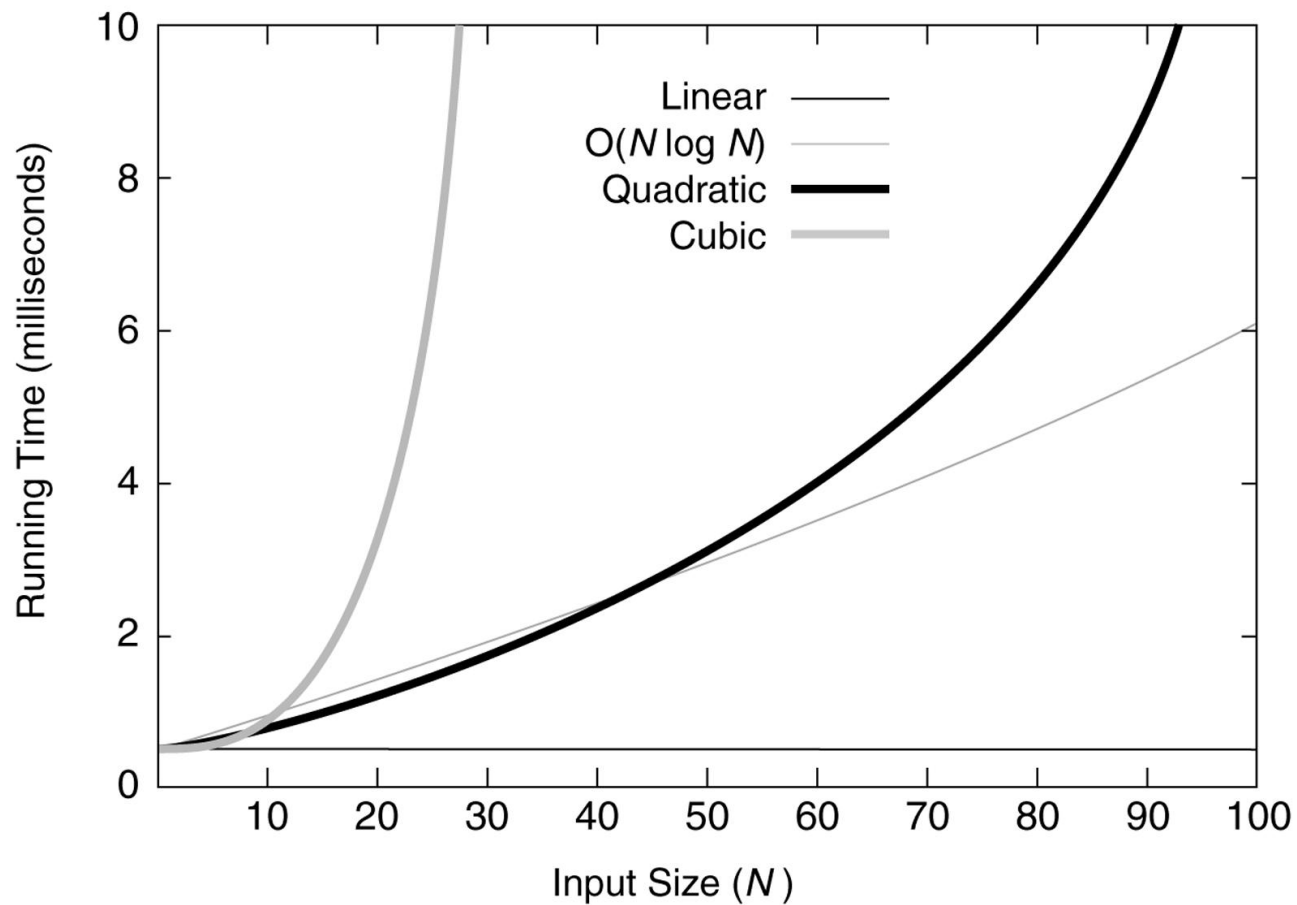
第二部分的比较次数不会超过 $n-1$ ，属于 $O(n)$

那么，算法的整体效率应该属于集合 $O(\max\{n^2, n\}) = O(n^2)$

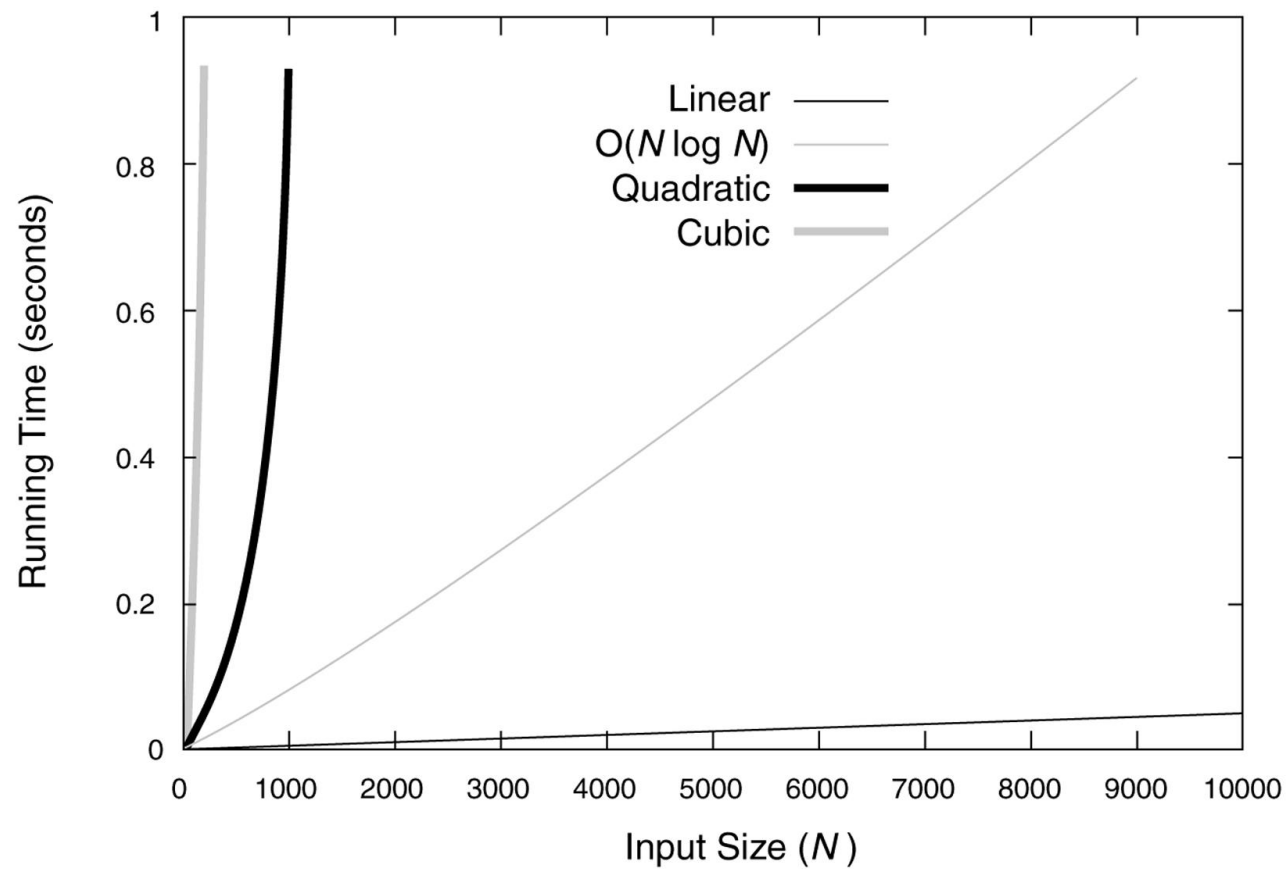
算法分析中常见的复杂性函数

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



中等规模数据



用C++描述算法

CATEGORY	EXAMPLES	ASSOCIATIVITY
Operations on References	. []	Left to right
Unary	++ -- ! - (type)	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift (bitwise)	<< >>	Left to right
Relational	< <= > >= instanceof	Left to right
Equality	== !=	Left to right
Boolean (or bitwise) AND	&	Left to right
Boolean (or bitwise) XOR	^	Left to right
Boolean (or bitwise) OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= *= /= %= += -=	Right to left

分析代码

C++ 操作	常数时间
顺序语句	语句时间和
条件语句	较大分支+条件测试
循环	迭代和
函数调用	函数体代价
递归函数	求解递归方程

递归 (Recursion)

- 递归过程一般可以通过解递归方程进行分析

- 基本形式:

$$T(n) =$$

base case: some constant

recursive case: $T(\text{subproblems}) + T(\text{combine})$

- 结果依赖于

- 子问题的个数
- 子问题的规模
- 子问题的解如何合并形成整个问题的解

二分查找

BinarySearch (A, x)

在有序数组A中查询x

7	12	30	35	75	83	87	90	97	99
---	----	----	----	----	----	----	----	----	----

子问题规模是原来的一半

方程:

$$T(1) \leq b$$

$$T(n) \leq T(n/2) + c \quad \text{for } n > 1$$

二分查找

方程:

$$T(1) \leq b$$

$$T(n) \leq T(n/2) + c \quad \text{for } n > 1$$

求解:

$$T(n) \leq T(n/2) + c$$

$$\leq T(n/4) + c + c$$

$$\leq T(n/8) + c + c + c$$

$$\leq T(n/2^k) + kc$$

$$\leq T(1) + c \log n, \quad k = \log n$$

$$\leq b + c \log n = O(\log n)$$

算法分析的基本法则

非递归算法：

(1) **for / while** 循环

循环体内计算时间*循环次数；

(2) 嵌套循环

循环体内计算时间*所有循环次数；

(3) 顺序语句

各语句计算时间相加；

(4) **if-else**语句

if语句计算时间和**else**语句计算时间的较大者。

嵌套循环

```
for i = 1 to n do  
  for j = 1 to n do  
    sum = sum + 1
```

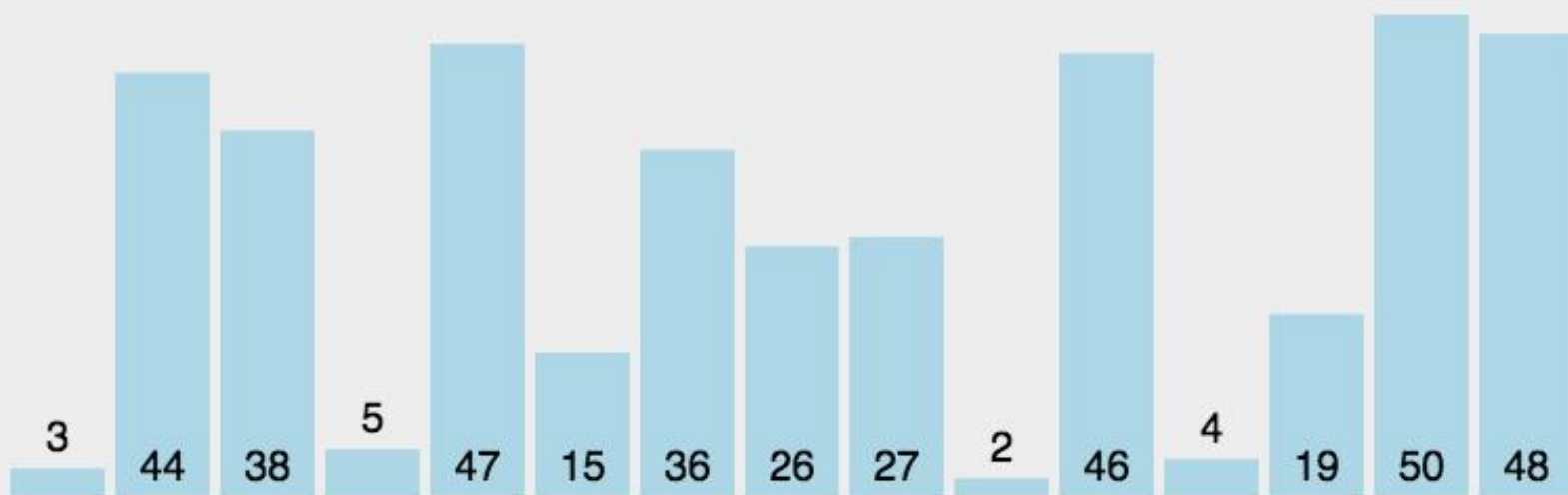
$$\sum_{i=1}^n \sum_{j=1}^n 1 = \sum_{i=1}^n n = n^2$$

```
for i = 1 to n do
  for j = i to n do
    sum = sum + 1
```

$$\sum_{i=1}^n \sum_{j=i}^n 1 = \sum_{i=1}^n (n - i + 1) = \sum_{i=1}^n (n + 1) - \sum_{i=1}^n i =$$

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

插入排序 (Insertion Sort)



```

template<class Type>
void insertion_sort(Type *a, int n)
{
    Type key;                                // cost    times
    for (int i = 1; i < n; i++){             // c1      n
        key=a[i];                            // c2      n-1
        int j=i-1;                          // c3      n-1
        while( j>=0 && a[j]>key ){            // c4      sum of ti
            a[j+1]=a[j];                     // c5      sum of (ti-1)
            j--;                             // c6      sum of (ti-1)
        }
        a[j+1]=key;                          // c7      n-1
    }
}

```

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n-1} (t_i - 1) + c_7(n-1)$$

- 在最好情况下， $t_i=1$, for $1 \leq i < n$;

已排好序

$$T_{\min}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) = O(n)$$

- 在最坏情况下， $t_i \leq i$, for $1 \leq i < n$;

倒序

$$\sum_{i=1}^{n-1} (i+1) = \frac{n(n-1)}{2} \quad \sum_{i=1}^{n-1} i = \frac{n(n-2)}{2} + 1$$

$$T_{\max}(n) \leq c_1 n + c_2(n-1) + c_3(n-1) +$$

$$c_4 \left(\frac{n(n-1)}{2} \right) + c_5 \left(\frac{n(n-2)}{2} + 1 \right) + c_6 \left(\frac{n(n-2)}{2} + 1 \right) + c_7(n-1)$$

$$= \frac{c_4 + c_5 + c_6}{2} n^2 + \left(c_1 + c_2 + c_3 - \frac{c_4}{2} - c_5 - c_6 + c_7 \right) n - (c_2 + c_3 - c_5 + c_7)$$

$$= O(n^2)$$

对于输入数据 $a[i]=n-i, i=0,1,\dots,n-1$ ，算法insertion_sort 达到其最坏情形。因此，

$$\begin{aligned} T_{\max}(n) &\geq \frac{c_4 + c_5 + c_6}{2} n^2 + \left(c_1 + c_2 + c_3 - \frac{c_4}{2} - c_5 - c_6 + c_7 \right) n - (c_2 + c_3 - c_5 + c_7) \\ &= \Omega(n^2) \end{aligned}$$

由此可见， $T_{\max}(n) = \Theta(n^2)$

最优算法

- 问题的计算时间下界为 $\Omega(f(n))$ ，则计算时间复杂度为 $O(f(n))$ 的算法是最优算法。
- 例如，排序问题的计算时间下界为 $\Omega(n\log n)$ ，计算时间复杂度为 $O(n\log n)$ 的排序算法是最优算法。
- 堆排序算法 $O(n\log n)$ 是最优算法。

NP完全性理论

如何区分一个问题的难易？

- **多项式时间：**在计算复杂度理论中，指的是一个问题的计算时间不大于问题规模 n 的多项式倍数。即多项式时间就是指时间复杂度是个多项式。程序运行的时间随着数据规模 n 变化的函数为 $f(n)$ ， $f(n)$ 是个多项式函数，那么就可以说是控制在多项式之内。

NP完全性理论

- **P类问题：**所有可以在多项式时间内求解的判定问题构成P类问题。**判定问题：**判断是否有一种能够解决某一类问题的能行算法的研究课题。
- 时间复杂度如(n^2 , n^4 , $n(\log(n))$)都是P时间的，指数级别的如(2^n , n^n)这些就不是P时间。

- **NP类问题：**所有的非确定性多项式时间可解的判定问题构成NP类问题。(Non-deterministic polynomial)
- 给定一个问题，我们可能不知道如何解，但如果通过连蒙带猜，得到了一个解，对于这个解，我们可以在P时间内验证它正确与否的一类问题，成为NP问题。

主定理 (Master Theorem)

主定理适用于求解递归式算法的时间复杂度

定理： 设 $a \geq 1, b > 1$ 为常数, $f(n)$ 为函数, $T(n)$ 为非负整数, 且 $T(n) = aT(n/b) + f(n)$, 则

1. 若 $f(n) = O(n^{\log_b a - \varepsilon})$, $\varepsilon > 0$, 那么

$$T(n) = \Theta(n^{\log_b a})$$

存在 ε

$$f(n) < n^{\log_b a}$$

2. 若 $f(n) = \Theta(n^{\log_b a})$, 那么

$$T(n) = \Theta(n^{\log_b a} \log n)$$

存在 ε

$$f(n) = n^{\log_b a}$$

3. 若 $f(n) = \Omega(n^{\log_b a + \varepsilon})$, $\varepsilon > 0$, 且对于某个常数 $c < 1$ 和充分大的 n 有 $af(n/b) \leq cf(n)$, 那么

$$T(n) = \Theta(f(n))$$

存在 c
和 n_0

$$f(n) > n^{\log_b a}$$

主定理 (Master Theorem)

二分查找(Binary search):

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

分析: $a = 1, b = 2, f(n) = \Theta(1)$, 基准函数 $\Theta(n^{\log_2 1}) = \Theta(1)$, 适用于Case2, 所以 $T(n) = \Theta(\log n)$;

二叉树遍历(Binary tree traversal):

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

分析: $a = b = 2, f(n) = O(1)$, 基准函数 $\Theta(n^{\log_2 2}) = \Theta(n)$, 适用于Case1, 所以 $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$

算法渐近复杂性分析中常用函数

(1) 单调函数

单调递增: $m \leq n \Rightarrow f(m) \leq f(n)$;

单调递减: $m \leq n \Rightarrow f(m) \geq f(n)$;

严格单调递增: $m < n \Rightarrow f(m) < f(n)$;

严格单调递减: $m < n \Rightarrow f(m) > f(n)$.

(2) 取整函数

$\lfloor x \rfloor$: 不大于 x 的最大整数;

$\lceil x \rceil$: 不小于 x 的最小整数。

取整函数的若干性质

$$x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1;$$

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n;$$

对于 $n \geq 0$, $a, b > 0$, 有:

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil;$$

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor;$$

$$\lceil a/b \rceil \leq (a+(b-1))/b;$$

$$\lfloor a/b \rfloor \geq (a-(b-1))/b;$$

$f(x) = \lfloor x \rfloor$, $g(x) = \lceil x \rceil$ 为单调递增函数。

(3) 多项式函数

$$p(n) = a_0 + a_1n + a_2n^2 + \dots + a_dn^d; \quad a_d > 0;$$

$$p(n) = \Theta(n^d);$$

$$f(n) = O(n^k) \Leftrightarrow f(n) \text{ 多项式有界};$$

$$f(n) = O(1) \Leftrightarrow f(n) \leq c;$$

$$k \geq d \Rightarrow p(n) = O(n^k);$$

$$k \leq d \Rightarrow p(n) = \Omega(n^k);$$

$$k > d \Rightarrow p(n) = o(n^k);$$

$$k < d \Rightarrow p(n) = \omega(n^k).$$

(4) 指数函数

对于正整数 m, n 和实数 $a > 0$:

$$a^0 = 1;$$

$$a^1 = a;$$

$$a^{-1} = 1/a;$$

$$(a^m)^n = a^{mn};$$

$$(a^m)^n = (a^n)^m;$$

$$a^m a^n = a^{m+n};$$

$a > 1 \Rightarrow a^n$ 为单调递增函数;

$$a > 1 \Rightarrow \lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0 \Rightarrow n^b = o(a^n)$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

$$e^x \geq 1+x;$$

$$|x| \leq 1 \Rightarrow 1+x \leq e^x \leq 1+x+x^2 ;$$

$$e^x = 1+x+ \Theta(x^2), \text{ as } x \rightarrow 0;$$

$$\lim_{n \rightarrow \infty} \left(1 + \frac{x}{n}\right)^n = e^x$$

(5) 对数函数

$$\log n = \log_2 n;$$

$$\lg n = \log_{10} n;$$

$$\ln n = \log_e n;$$

$$\log^k n = (\log n)^k;$$

$$\log \log n = \log(\log n);$$

$$\text{for } a > 0, b > 0, c > 0$$

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b (1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

$$|x| \leq 1 \Rightarrow \ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \dots.$$

$$\text{for } x > -1, \quad \frac{x}{1+x} \leq \ln(1+x) \leq x$$

$$\text{for any } a > 0, \quad \lim_{n \rightarrow \infty} \frac{\log^b n}{(2^a)^{\log n}} = \lim_{n \rightarrow \infty} \frac{\log^b n}{n^a} = 0 \Rightarrow \log^b n = o(n^a)$$

(6) 阶层函数

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

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

Stirling's approximation

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\alpha_n} \quad \frac{1}{12n+1} < \alpha_n < \frac{1}{12n}$$

$$n! = o(n^n)$$

$$n! = \omega(2^n)$$

$$\log(n!) = \Theta(n \log n)$$

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。

作业

- 证明 $\gcd(m,n) = \gcd(n, m \bmod n)$
- 设计计算 $\lfloor \sqrt{n} \rfloor$ 的算法， n 为任意整数，除了赋值与比较运算，只能用到基本四则运算
- 证明主定理
- 1-6, 1-7