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) ——“算法”的来源

- 算法(Algorithm) 一词来源于著名的波斯帝国塔吉克族数学家、天文学家、地理学家——阿尔·花刺子模(Al-Khwarizmi)，其拉丁名为 阿尔戈利兹姆(Algorismus)
- 他是代数与算术的创立人，被誉为代数之父。代数(Algebra)源于他的一本书的标题，而算法(Algorism、Algorithm)源于他的拉丁文名

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 mod n)

m mod 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 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≠0 do
    r ←m mod n
    m ←n
    n ←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 = 2×2×3×5
24= 2×2×2×3
gcd(60, 24) = 2×2×3 = 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中找出所有的公因数（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。

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。

描述算法的形式

1、自然语言

优点:通俗易懂　　　　　　　　　　缺点:但缺乏直观性和简洁性，且易产生歧义。

“南京市长江大桥”、“这个人连我都不认识”、“我曾经喜欢一个人，我现在喜欢一个人"……。

2、程序流程图

优点:描述算法形象、直观，容易理解　　缺点:绘制过程比较费时费力，难以修改。

3、伪代码

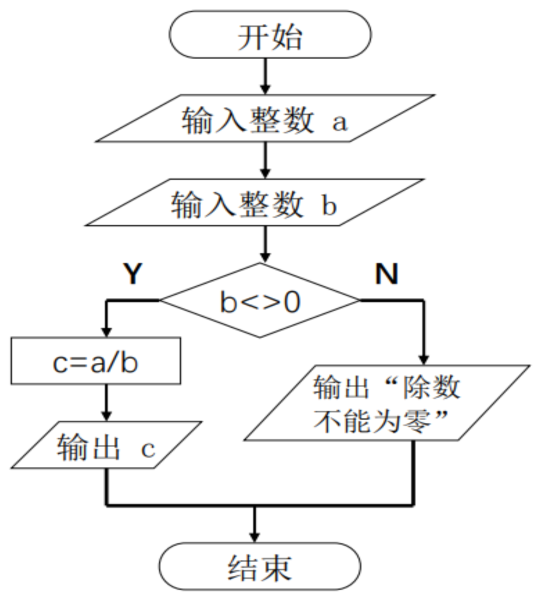
优点:简单易懂，修改容易，易于转化为程序语言代码

缺点:伪代码格式难以规范

自然语言

1	输入整数 a
2	输入整数 b
3	如果 b=0 转到第7步
4	计算 c=a/b
5	输出 c
6	转到第8步
7	输出 “除数不能为零”
8	结束

程序流程图



伪代码

```
read  a
read  b
if  b ≠ 0
    c ← a / b
    print c
else
    print "除数不能为零"
```

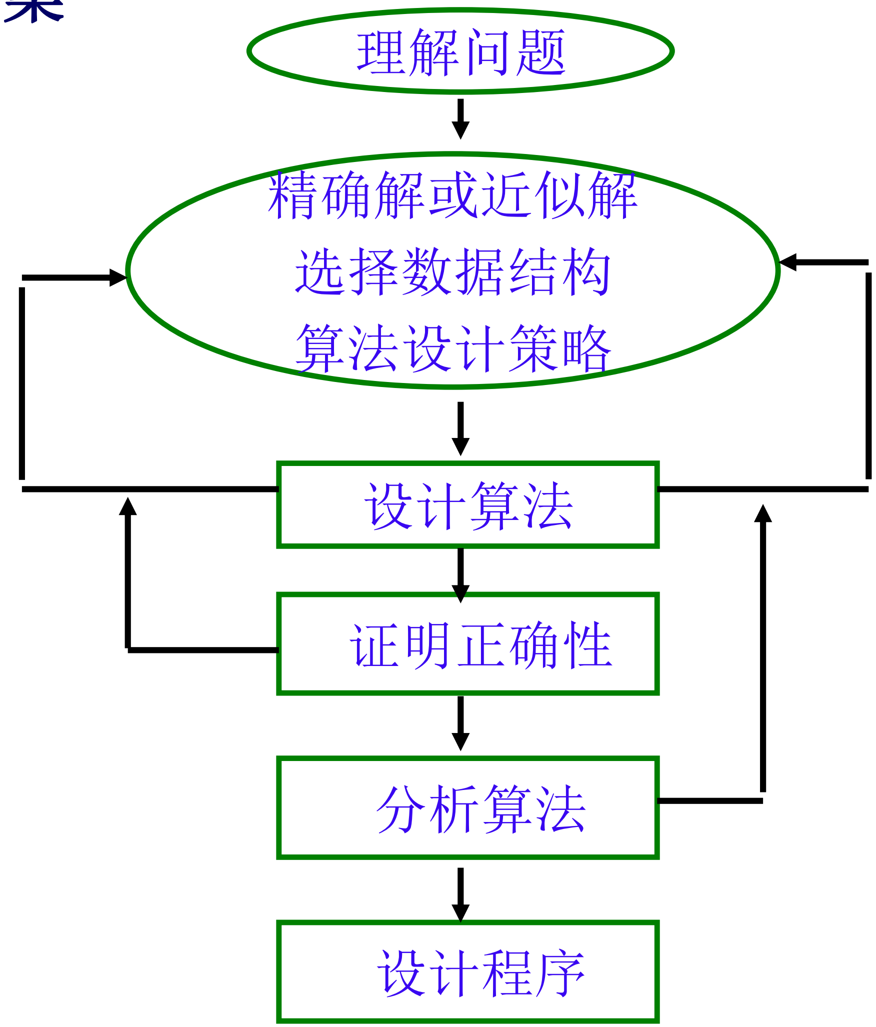
问题求解(Problem Solving)

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

和具体输入有关

（1）**最坏情况**下的时间复杂性

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

（2）**最好情况**下的时间复杂性

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

（3）**平均情况**下的时间复杂性

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

其中I是问题的规模为n的实例，$p(I)$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) | size(I)=n }=$O(n)$

（2）$T_{min}(n)$ = min{ T(I) | size(I)=n }=$O(1)$

（3）在平均情况下，假设：

(a) 搜索成功的概率为p ($0 \le p \le 1$)；

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

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

$$= \left(1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + 3 \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right) + n \cdot (1-p)$$

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

- 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

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

$T(n) \to \infty$， as $n \to \infty$；$(T(n) - t(n))/ T(n) \to 0$， 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，$f(n) \geq 0$，$g(n) \geq 0$。

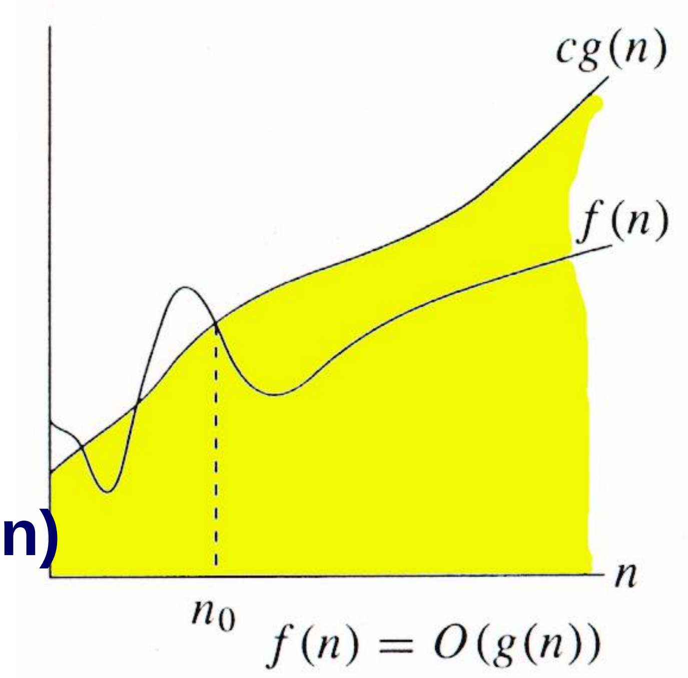
（1）**渐近上界记号O**

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

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

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

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



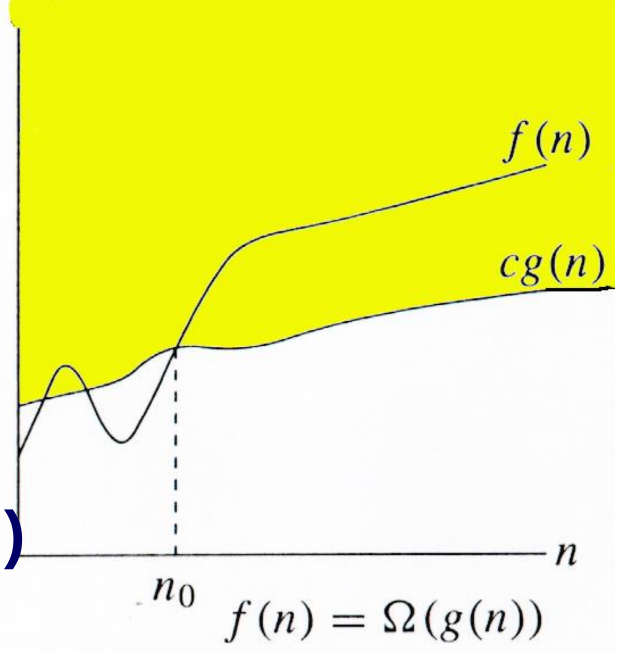
O(g(n))是增长次数小于或等于g(n)的函数集合

- 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。
- $O(g(n))=\{f(n)|f(n)$满足：存在正的常数c和n_0，使得当$n>=n_0$时$f(n)<=cg(n)\}$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

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

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



Ω(g(n))是增长次数大于或等于g(n)的函数集合

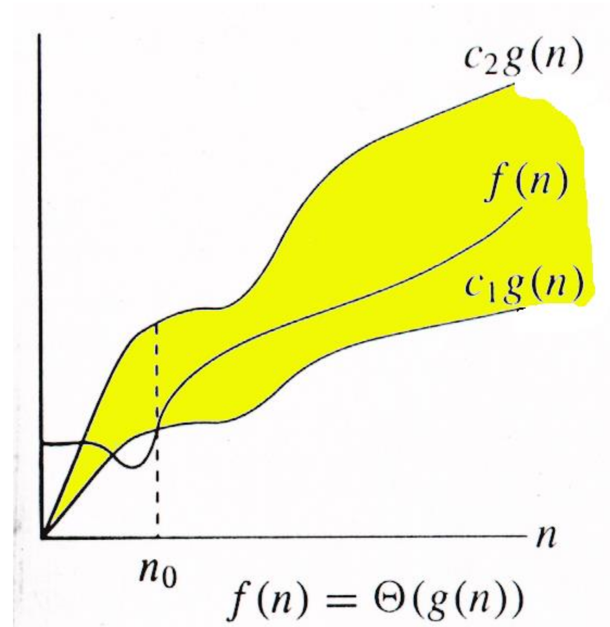
（3）**紧渐近界记号Θ**

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

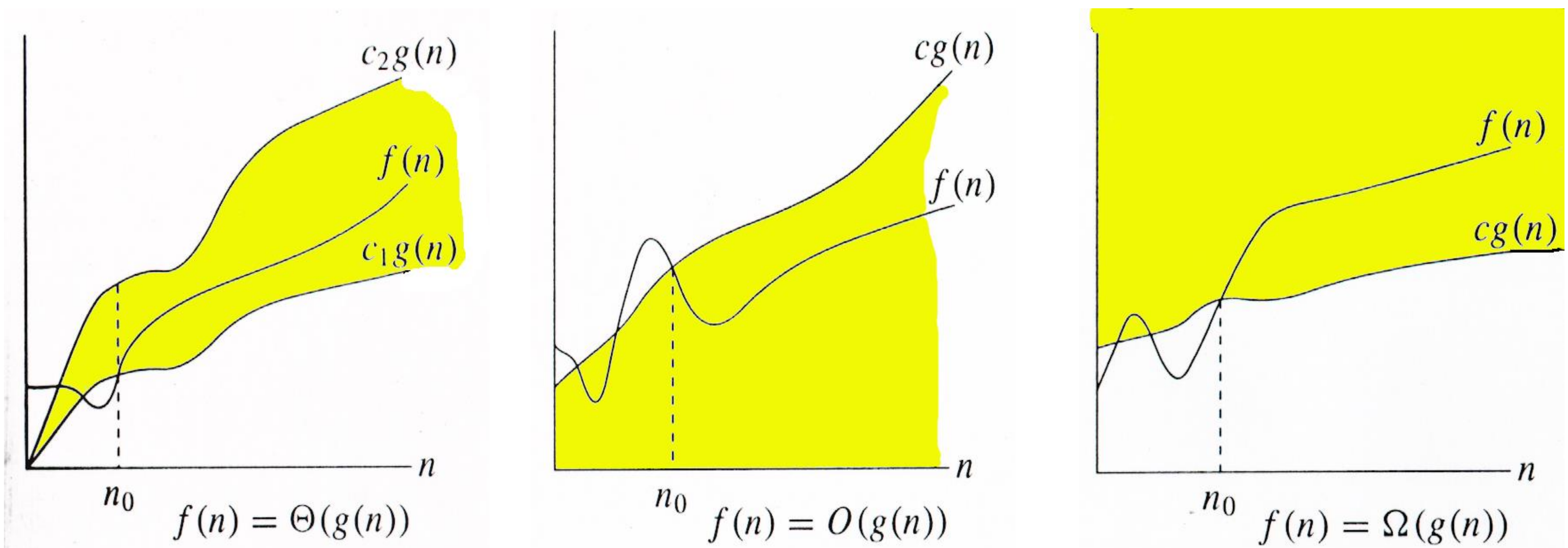
定理1： $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

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

Θ(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

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

$2n^2 + 4n = O(n^2)$ ✓

$O(n^2) = 2n^2 + 4n$ ✘

Big-O 实例

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

$(n^2 + 100\,n) \le 2\,n^2 \quad \text{for } n \ge 10$

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

$n^2 + 100\,n) \ge 1\,n^2 \quad \text{for } n \ge 0$

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

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

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

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

- **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

$o(g(n)) = \{ f(n) \mid$ 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 $f(n) / g(n) \rightarrow 0$， as $n \rightarrow \infty$。

（5）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 $f(n) / g(n) \rightarrow \infty$， 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

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

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

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

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

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

渐近分析记号的若干性质

（1）传递性：

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

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

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

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

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

（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：

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

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

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

$O(cf(n)) = O(f(\mathrm{n}))$；

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

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

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

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 $O(n^2)$

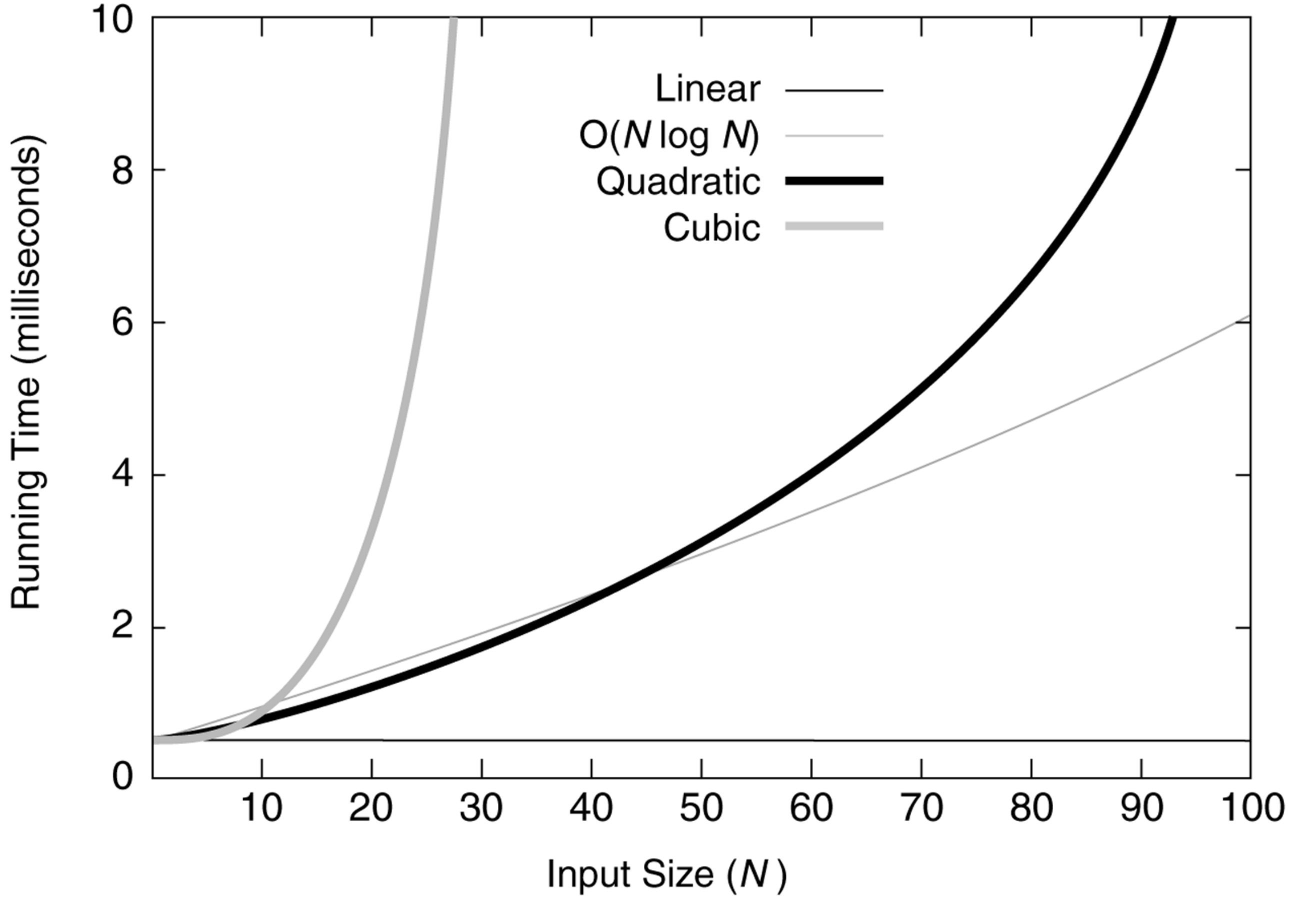
第二部分的比较次数不会超过n-1,属于 O(n)

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

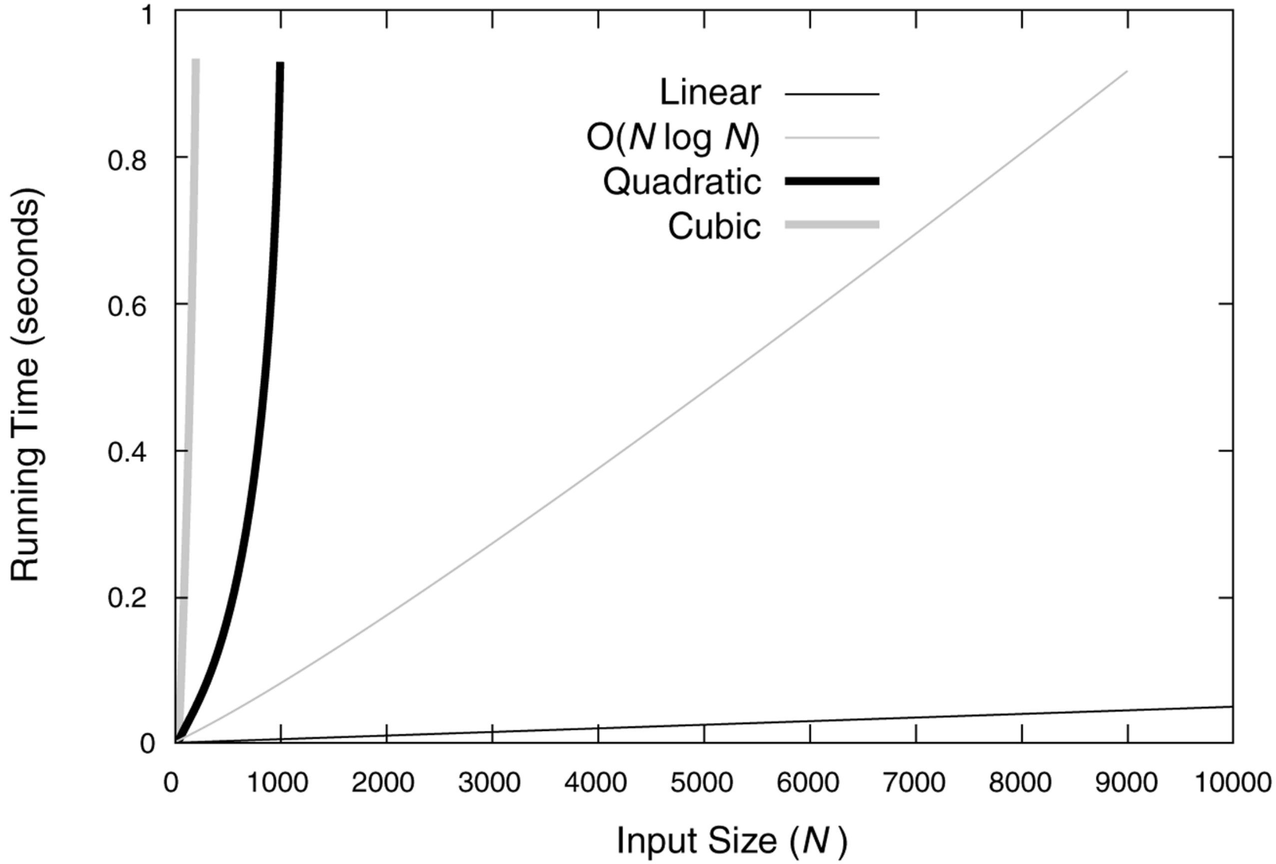
FUNCTION	NAME
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

小规模数据



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

中等规模数据



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

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(subproblems) + T(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:

$T(1) \le b$

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

求解:

$$
\begin{aligned}
T(n) &\le T(n/2) + c \\
&\le T(n/4) + c + c \\
&\le T(n/8) + c + c + c \\
&\le T(n/2^k) + kc \\
&\le T(1) + c \log n \text{，} \quad k = \log n \\
&\le b + c \log n \quad = \quad O(\log n)
\end{aligned}
$$

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
```

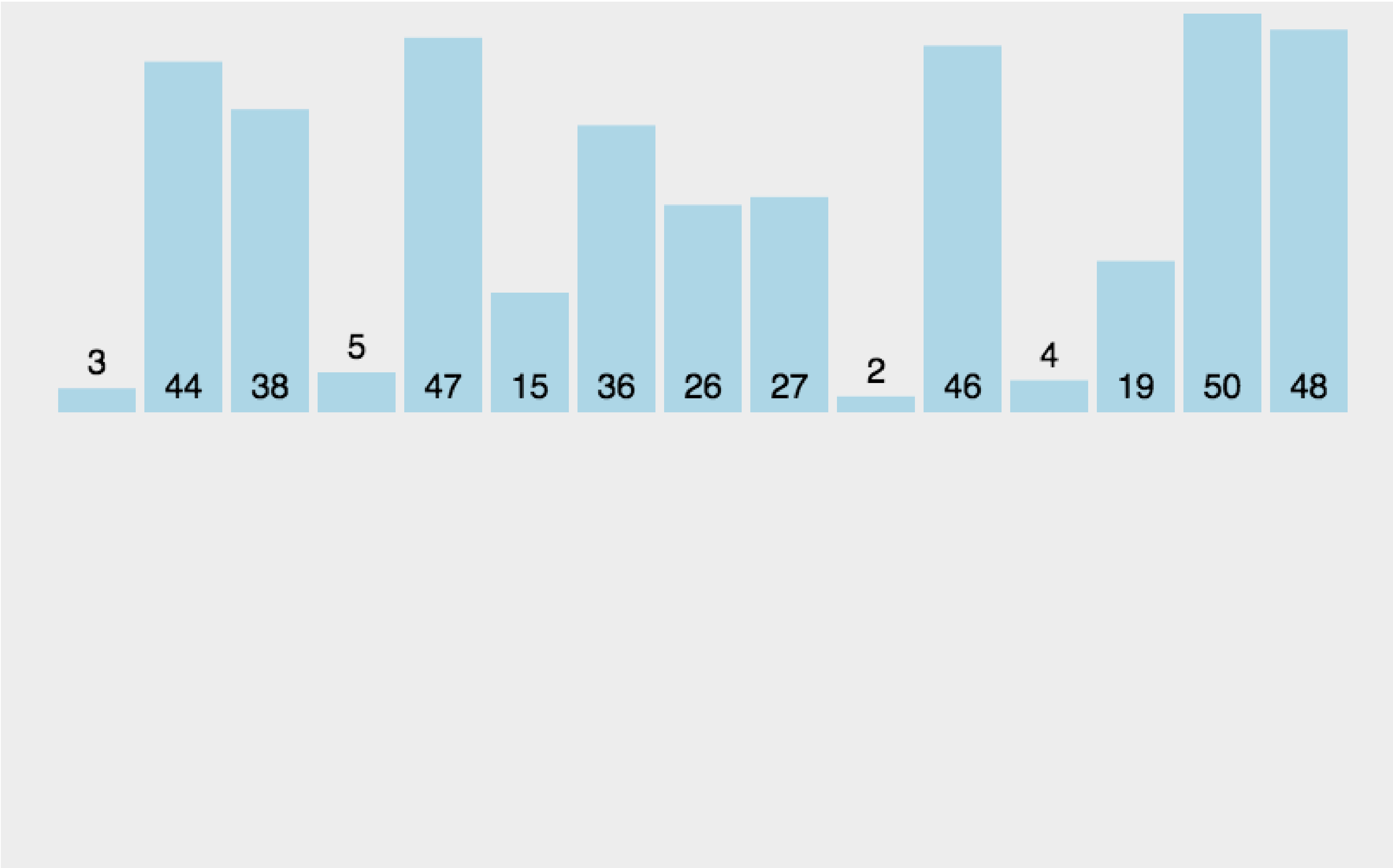
$$\sum_{i=1}^{n} \sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2$$

```
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
    }
}
```

$$T(n)=c_1n+c_2(n-1)+c_3(n-1)+c_4\sum_{i=1}^{n-1}t_i+c_5\sum_{i=1}^{n-1}(t_i-1)+c_6\sum_{i=1}^{n-1}(t_i-1)+c_7(n-1)$$

- 在最好情况下，$t_i=1$, for $1 \le i < n$;

$\boxed{\text{已排好序}}$

$$T_{\min}(n)=c_1n+c_2(n-1)+c_3(n-1)+c_4(n-1)+c_7(n-1)$$

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

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

$\boxed{\text{倒序}}$

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

$$T_{\max}(n)\le c_1n+c_2(n-1)+c_3(n-1)+$$
$$c_4\left(\frac{n(n-1)}{2}\right)+c_5\left(\frac{n(n-2)}{2}+1\right)+c_6\left(\frac{n(n-2)}{2}+1\right)+c_7(n-1)$$
$$=\frac{c_4+c_5+c_6}{2}n^2+\left(c_1+c_2+c_3-\frac{c_4}{2}-c_5-c_6+c_7\right)n-(c_2+c_3-c_5+c_7)$$
$$=O(n^2)$$

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

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

由此可见，$T_{\max}(n)=\Theta(n^2)$

最优算法

- 问题的计算时间下界为Ω(f(n))，则计算时间复杂性为O(f(n))的算法是最优算法。
- 例如，基于数值比较的排序问题的计算时间下界为Ω($n$log$n$)，计算时间复杂性为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。

算法渐近复杂性分析中常用函数

（1）单调函数

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

单调递减：$m \le n \Rightarrow f(m) \ge f(n)$;

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

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

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

对于$n \ge 0$，$a,b>0$，有：

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

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

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

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

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

（3）多项式函数

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

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

$f(n) = O(n^k) \Leftrightarrow f(n)$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;

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

可证明：多项式函数的阶低于指数函数的阶

$$\boxed{n^d = o(r^n),\ r>1,\ d>0}$$

证 不妨设d为正整数，

$$\lim_{n\to\infty}\frac{n^d}{r^n} = \lim_{n\to\infty}\frac{dn^{d-1}}{r^n \ln r} = \lim_{n\to\infty}\frac{d(d-1)n^{d-2}}{r^n(\ln r)^2}$$

$$= \dots = \lim_{n\to\infty}\frac{d!}{r^n(\ln r)^d} = 0$$

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

$e^x \geq 1+x$;

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

$e^x = 1+x+ \Theta(x^2)$, as $x \to 0$;

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

（5）对数函数

$\log n = \log_2 n$;

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

$\ln n = \log_e n$;

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

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

for a>0,b>0,c>0

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

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

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

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

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

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

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

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

for $x > -1$, $$\frac{x}{1+x} \le \ln(1+x) \le x$$

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

可证明：对数函数的阶低于幂函数的阶

$$\boxed{\ln n = o\,(n^d)\text{，}\ d > 0}$$

证

$$\lim_{n\to\infty} \frac{\ln n}{n^d} = \lim_{n\to\infty} \frac{\frac{1}{n}}{dn^{d-1}}$$

$$= \lim_{n\to\infty} \frac{1}{dn^d} = 0$$

（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

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

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

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

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

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

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

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

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

存在c和n_0

(1). 在第一种情况，$f(n)$不仅小于$n^{\log_b a}$，必须多项式地小于，即对于一个常数$\varepsilon > 0$，$f(n) = O(\frac{n^{\log_b a}}{n^{\varepsilon}})$.

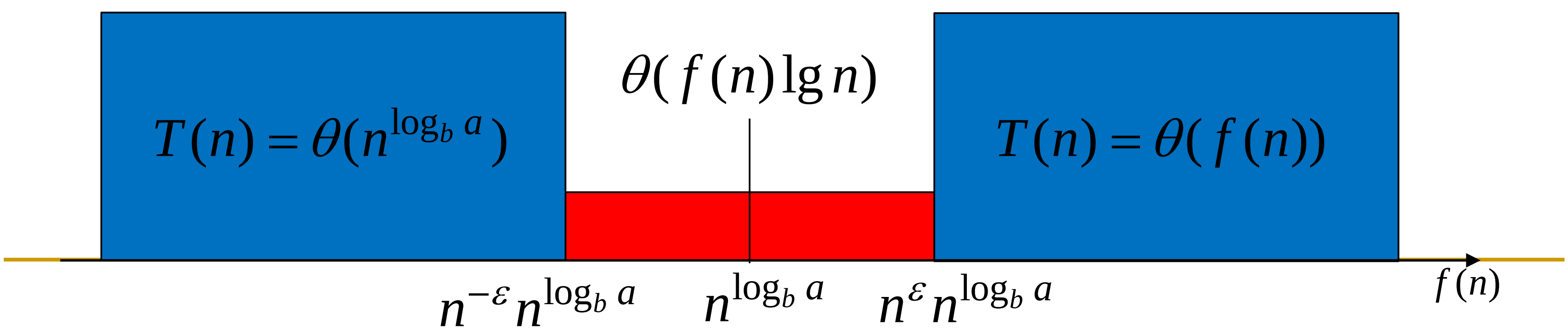
(2). 在第三种情况，$f(n)$不仅大于$n^{\log_b a}$，必须多项式地大于，即对于一个常数$\varepsilon > 0$，$f(n) = \Omega(n^{\log_b a} \cdot n^{\varepsilon})$.

*直观地：我们用$f(n)$与$n^{\log_b a}$比较

(1). 若$n^{\log_b a}$大，则$T(n) = \theta(n^{\log_b a})$

(2). 若$f(n)$大，则$T(n) = \theta(f(n))$

(3). 若$f(n)$与$n^{\log_b a}$同阶，则$T(n) = \theta\left(n^{\log_b a} \lg n\right) = \theta(f(n) \lg n)$.



主定理（Master Theorem）

***a*≥1, *b*>1**

$$T(n) = aT(n/b) + f(n)$$

二分查找(Binary search):

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

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 $T(n) = \Theta(logn)$;

二叉树遍历(Binary tree traversal):

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

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

求解递推方程：

例1　求解递推方程

$$T(n) = 9T(n/3) + n$$

解　上述递推方程中的

$$a = 9,\quad b = 3,\ f(n) = n$$

$$n^{\log_3 9} = n^2,\ f(n) = O(n^{\log_3 9-1})$$

相当于主定理的case1，其中ε=1.

根据定理得到　$T(n) = \Theta(n^2)$

求解递推方程:

例2 求解递推方程

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

解 上述递推方程中的

$$a = 1,\ b = 3/2,\ f(n) = 1,$$

$$n^{\log_{3/2} 1} = n^0 = 1$$

相当于主定理的Case2 .

根据定理得到 $T(n) = \Theta(\log n)$

求解递推方程：

例3 求解递推方程

$$T(n) = 3T(n/4) + n\log n$$

解 上述递推方程中的

$a = 3$, $b = 4$, $f(n) = n\log n$

$n\log n = \Omega(n^{\log_4 3+\varepsilon}) \approx \Omega(n^{0.793+\varepsilon})$

取 $\varepsilon = 0.2$ 即可.

要使 $a f(n/b) \le c f(n)$ 成立，

代入 $f(n) = n \log n$，得到

$$3(n/4)\log(n/4) \le c n \log n$$

只要 $c \ge 3/4$，上述不等式可以对所有充分大的n 成立. 相当于主定理的 Case3.

因此有 $T(n) = \Theta(f(n)) = \Theta(n\log n)$

不能使用主定理的例子

例4 求解 $T(n)=2T(n/2)+n\log n$

解 $a=b=2$, $n^{\log_b a}=n$, $f(n)=n\log n$

不存在$\varepsilon>0$ 使下式成立

$$n\log n = \Omega(n^{1+\varepsilon})$$

不存在 $c<1$ 使 $af(n/b)\le cf(n)$对所有充分大的 n 成立

$$2(n/2)\log(n/2)=n(\log n-1)\le cn\log n$$

If there exists a constant $k \ge 0$ such that $f(n) = \Theta(n^{\log_b a}\lg^k n)$, then $T(n) = \Theta(n^{\log_b a}\lg^{k+1} n)$.

$T(n)=2T(n/2)+n\log n$

递归树求解

$n\log n$

$\frac{n(\log n-1)}{2}$ $\frac{n(\log n-1)}{2}$

$\frac{n(\log n-2)}{4}$ $\frac{n(\log n-2)}{4}$ $\frac{n(\log n-2)}{4}$ $\frac{n(\log n-2)}{4}$

....

$n\log n$

$n(\log n-1)$

$n(\log n-2)$

$\vdots$

$n(\log n-k+1)$

求和

$$
\begin{aligned}
T(n) &= n\log n + n(\log n-1) + n(\log n-2) \\
&\quad + \ldots + n(\log n-k+1) \\
&= (n\log n)\log n - n(\underline{1+2+\ldots+k-1)} \\
&= n\log^2 n - nk\underline{(k-1)/2} = O(n\log^2 n)
\end{aligned}
$$

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。

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。