计算理论学

通常, 对于一个给定的算法, 我们要做两项分析:

- 第一是从数学上证明算法的**正确性**
- 第二步就是分析算法的**复杂度**。

算法复杂度

- 算法复杂度: 时间复杂度和空间复杂度。
- 研究实验告诉我们，对于大多数问题来说，我们在速度上能够取得的进展要远大于在空间上的进展，所以我们把主要精力集中在时间复杂度上。

时间复杂度和算法的运行速度有关

```
LocationN(A[1..n], K)
// Find the location of K in array A
// Input: an real array A with size n, K is a real number
// Output: the position of K in A, -1: not found
 for i←1 to n do
     if  A[i]==K
         return i
return -1
```

讨论：如何分析算法的时间复杂度？

```
LocationN(A[1.. n], K)
// Find the location of K in array A
 for i←1 to n do
     if  A[i]==K
         return i
 return -1
```

1、算法的运行时间受什么影响？

2、只考虑运行时间可以比较时间复杂度吗？

2.1 算法效率分析框架

- （1）输入规模的度量
  - Measuring an input's size
- （2）运行时间的度量
  - Measuring the running time
- （3）增长次数
  - Orders of growth
- （4）算法的最优、最差和平均效率
  - Worst-case, best-case and average efficiency

分析框架（1）——输入规模度量

- 输入规模度量
  - 算法的**时间效率**和**空间效率**都用输入规模的函数进行度量。
  - 经常使用一个**以输入规模为参数的函数来研究算法的效率**。
  - 选择**输入规模**的**合适量度**，要受到所讨论算法的**操作细节影响**。

- 这里我们首先要明确**输入规模**的概念。

- 关于**输入规模**，不是很好下定义，非严格的讲，**输入规模**是指算法A所接受输入的**自然独立体的大小**。
- 例如，对于排序算法来说，**输入规模**一般就是待排序元素的个数。

分析框架（2）——运行时间的度量单位

- 把*基本操作*作为算法运行时间的度量单位。

- 基本操作：**就是算法中最重要的操作。**
  - 它对总运行时间的**贡献最大**。
  - **基本操作**通常是算法**最内层循环**中**最费时**的操作。

分析框架（2）——运行时间的度量单位

算法运行时间的估计：

用基本操作的执行次数来度量算法的**时间复杂度**。

$$T(n) \approx c_{op}C(n)$$

- n 是该算法的输入规模
- C_{op} 是特定计算机上一个算法基本操作的执行时间
- $C(n)$ 是该算法需要执行的基本操作的次数

分析框架（3）——增长次数

为什么要强调**执行次数**的**增长次数**呢？

这是因为**小规模输入**在运行时间上差别不足以将高效的算法和低效的算法区分开来。

对于算法分析具有重要意义的函数值（有些是近似值）

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

一个需要指数级操作次数的算法只能用来解决规模非常小的问题

分析框架（4）
——算法的最优、最差和平均效率

- 最差效率：是指在输入规模为n时，算法在**最坏情况下**的效率
- 最优效率：是指在输入规模为n时，算法在**最优情况下**的效率
- 平均效率：是指在“**典型**”或“**随机**”输入的情况下，算法具有的效率

LocationN(A[1..n], K)

```
// Find the location of K in array A
// Input: an real array A with size n, K is a real number
// Output: the position of K in A, -1: not found
 for i←1 to n do
     if  A[i]==K
         return i
 return -1
```

输入规模： n

基本操作： ==

增长次数： 线性 **n**

LocationN(A[1..n], K)

// Find the location of K in array A
// Input: an real array **A** with size **n**, **K** is a real number
// Output: the position of K in A, -1: not found

```
for i←1 to n do        //      运行时间 t1
    if  A[i]==v        //      运行时间 t2
        return i       //      运行时间 t3
return -1              //      运行时间 t3
```

算法的最优、最差和平均效率

最好情况：K排在A的第一位，运行时间　　t1+t2+t3

最坏情况：K不在A中，　　运行时间　(t1+t2)*n+t3

平均情况：　t1+t2 的平均运行次数：（1+2+3+…+n) *1/n

=1/n*(1+n)*n/2=(n+1)/2

平均运行时间 (t1+t2)*(n+1)/2+t3

LocationN(A[1..n], K)

// Find the location of K in array A
// Input: an real array **A** with size **n**, **K** is a real number
// Output: the position of K in A, -1: not found

```
for i←1 to n do        //      运行时间 t1
    if  A[i]==v        //      运行时间 t2
        return i       //      运行时间 t3
return -1              //      运行时间 t3
```

只考虑基本操作的增长次数

最好情况：v排在A的第一位，运行时间　　t2，**增长次数：1**

最坏情况：v不在A中，　运行时间　　t2*n ，　**增长次数：n**

平均情况：　　运行时间　t2*(n+1)/2 ，　**增长次数：n**

分析框架概要

- 1 算法的**时间效率**和**空间效率**都用**输入规模 n** 的**函数**进行度量。
- 2.1 我们用算法**基本操作**的**执行次数**来度量算法的**时间效率**。
- 2.2 通过计算算法消耗的**额外存储单元**的**数量**来度量**空间效率**。
- 3 在**输入规模**相同的情况下，有些算法的效率仍会有显著差异。对于这样的算法，我们需要区分**最差效率**，**平均效率**和**最优效率**。
- 4 本框架**主要关心**，当算法的**输入规模**趋向于**无穷大**的时候，其**运行时间函数**的**增长次数**。

- 如何表示算法的时间复杂度？
  - 利用数学上的渐进符号

2.2 渐进符号和基本效率类型

- 算法效率的主要指标是基本操作次数的**增长次数**。

- 为了对这些**增长次数**进行比较和归类，计算机科学家们使用了**3种符号：**
  - O（读“O″）：**渐近上界记号-**上界
  - Ω（读”omega″）：**渐近下界记号-**下界
  - Θ（读”theta″）：**紧渐近界记号-**近似

2.2.1 非正式介绍

- O(g(n)) 是**增长次数小于等于**g(n) (以及其常数倍，n趋向于无穷大)的函数集合。

  $n \in O(n^2), 100n+5 \in O(n^2), 1/2(n(n-1)) \in O(n^2), n^3 \notin O(n^2)$,

- Ω(g(n)) 代表**增长次数大于等于**g(n)(以及其常数倍,n趋向于无穷大)的函数集合。

  $n^3 \in \Omega(n^2)$， $1/2(n(n-1)) \in \Omega(n^2)$，但是$100n+5 \notin \Omega(n^2)$

- Θ(g(n))是**增长次数等于**g(n) (以及其常数倍,n趋向于无穷大)的函数集合。

符号O

- **定义1**：对于足够大的*n*，*t(n)*的上界由*g(n)*的常数倍来确定，即：

$$t(n) \le cg(n)，c为正的常数$$

$$\exists n_0 > 0, \exists c > 0 \left(\forall n > n_0 \left(t(n) \le cg(n) \right) \right)$$

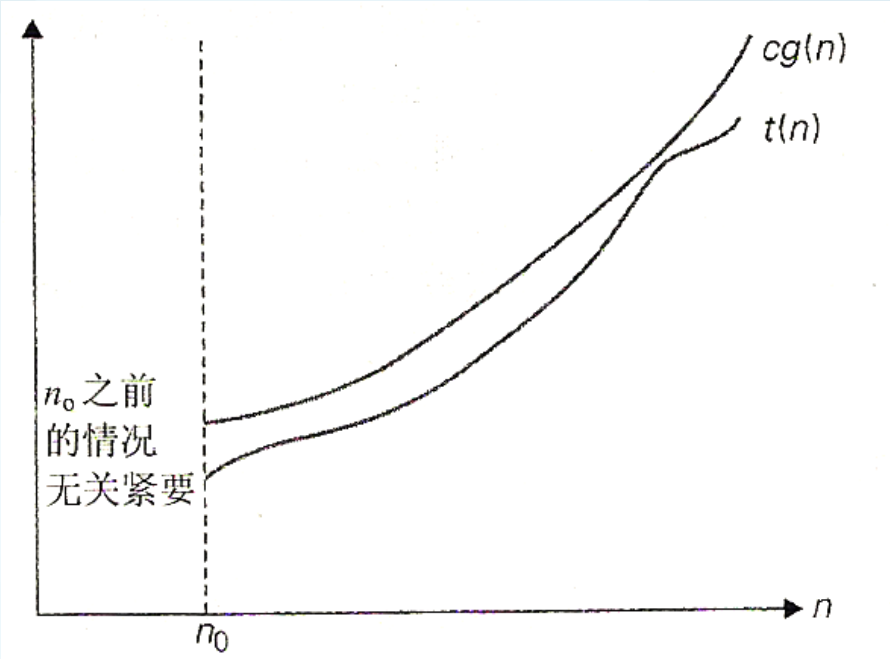
- 记为：$t(n) \in O(g(n))$

$3n \in O(n)$

$n \in O(n^2)$

$100n+5 \in O(n^2)$

$n(n-1)/2 \in O(n^2)$



符号Ω

- **定义2**: 对于足够大的*n*，*t(n)*的下界由*g(n)*的常数倍来确定，即：

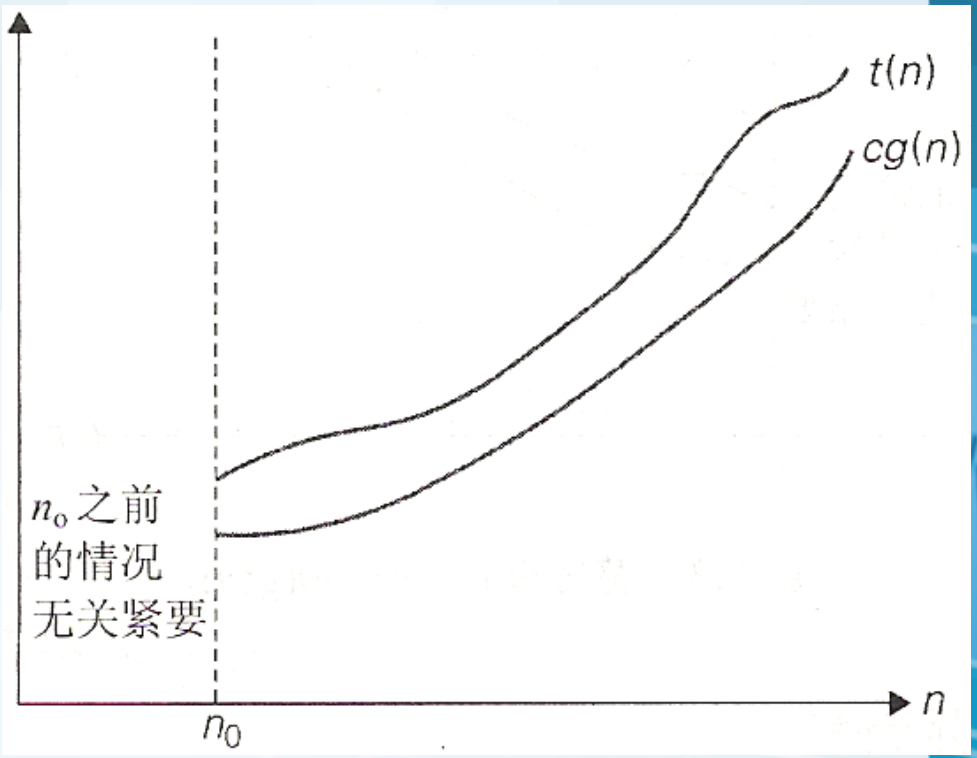
$$t(n) \geq cg(n)，c为正的常数$$

- 记为: $t(n) \in \Omega(g(n))$

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

$n(n+1) \in \Omega(n^2)$

$4n^2+5 \in \Omega(n^2)$



符号Θ

- **定义3:** 对于足够大的n，$t(n)$的上界和下界都由$g(n)$的常数倍来确定，即：

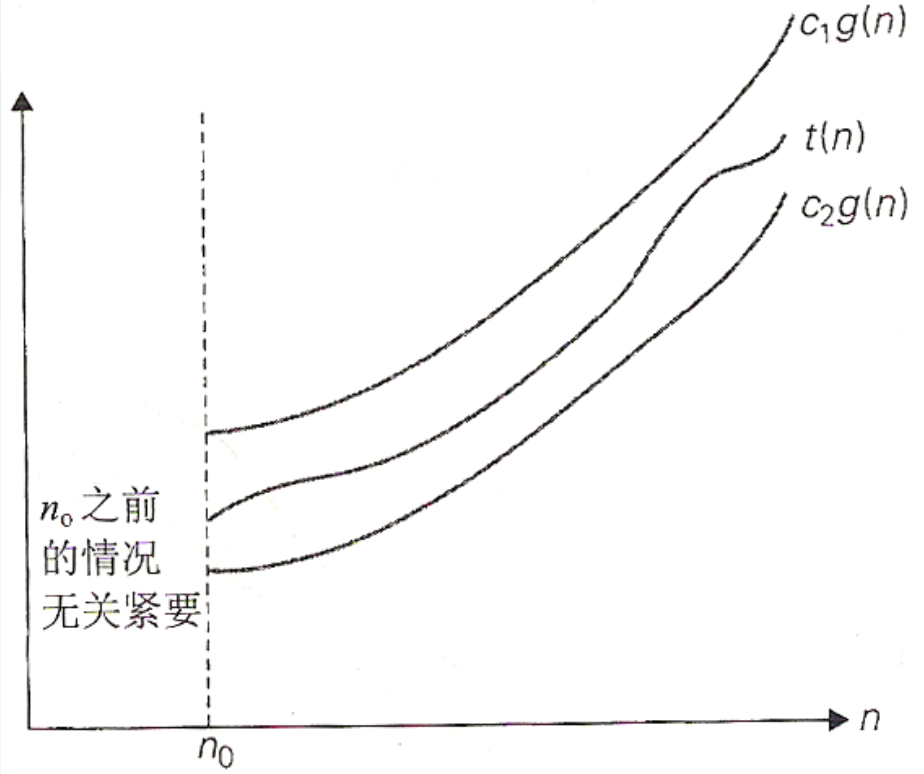
$$c_2 g(n) \le t(n) \le c_1 g(n)，\ c_1, c_2 \text{为正的常数}$$

- 记为: $t(n) \in \Theta(g(n))$

$n^2+3n+2 \in \Theta(n^2)$

$n(n-1)/2 \in \Theta(n^2)$

$4n^2+5 \in \Theta(n^2)$



渐进符号的有用特性

- **定理:** 如果 $t_1(n) \in O(g_1(n))$ 并且
  $t_2(n) \in O(g_2(n))$，则
  $t_1(n) + t_2(n) \in O(\max\{(g_1(n), g_2(n)\})$

- 对于符号Ω和Θ，该定理也成立。
- 该定理表明：**当算法由两个连续执行部分组成时，该算法的整体效率由具有较大增长次数的那部分所决定。**

2.2.6 利用极限比较增长次数

基于对所计论的两个函数的比率求极限，
有3种极限情况会发生：

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & \text{表明}t(n)\text{的增长次数比}g(n)\text{小}\\ c & \text{表明}t(n)\text{的增长次数和}g(n)\text{相同}\\ \infty & \text{表明}t(n)\text{的增长次数比}g(n)\text{大}\end{cases}$$

前两种情况意味着 $t(n) \in O(g(n))$

后两种情况意味着 $t(n) \in \Omega(g(n))$

第二种情况意味着 $t(n) \in \Theta(g(n))$

2.2.6 利用极限比较增长次数

求极限时，可以采用洛必达法则：

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

以及史特林公式：

当n足够大时，$n!\approx\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

2.2.7 基本的效率类型

常量(1)、

对数(logn)、

线性(n)、

nlogn、

平方(n^2)、

立方(n^3)、

指数(2^n)、

阶乘(n!)

1	constant
$\log n$	logarithmic
n	linear
$n \log n$	$n \log n$
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Orders of Growth

指数级增长函数

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

增长次数:

- 仅考虑公式中的主导因子
- 忽略常量因子

LocationN(A[1..n], K)

// Find the location of K in array A
// Input: an real array **A** with size **n**, **K** is a real number
// Output: the position of K in A, -1: not found

```
for i←1 to n do          //      运行时间 t1
    if  A[i]==K          //      运行时间 t2
        return i         //      运行时间 t3
return -1                //      运行时间 t3
```

只考虑基本操作的增长次数

平均情况： 运行时间 t2*(n+1)/2 ， 增长次数 ： n

所以此算法平均的时间复杂度为: O(n) 或者说 Θ(n)

练习

比较 增长次数:

- $n\ln(n)$, $n^{4/3}$

算法效率分析基础

- 非递归算法的效率分析
- 递归算法的效率分析

算法可分为：非递归算法和递归算法

非递归算法 可以调用别的算法，但不会再调用自身。

递归算法 是一种直接或者间接地调用自身算法的过程。

2.3 非递归算法的数学分析

例1： 考虑下面算法的效率

元素唯一性问题：验证给定数组中的元素**是否全部唯一**。

算法 *UniqueElements*(A[0..n-1])

```
//验证给定数组中的无素是否全部唯一
//输入：数组A[0..n-1]
//输出：如果A中的元素全部唯一，返回“true”
//        否则，返回“false”.
 for i←0 to n-2 do
      for j←i+1 to n-1 do
          if A[i]=A[j]
              return false
Return true
```

例1： 元素唯一性问题

```
for i←0 to n-2 do
    for j←i+1 to n-1 do
        if A[i]=A[j]
            return false
Return true
```

- **输入规模：**n
- **基本操作：**两个元素的**比较**
- 除了和n有关外，还取决于数组中是否有相同的元素，以及它们在数组中的位置
- 必须研究其**最优，平均**和**最差效率**

例1： 元素唯一性问题的最差效率

```
for i←0 to n-2 do
    for j←i+1 to n-1 do
        if A[i]=A[j]
            return false
Return true
```

$$C_{worst}(n)=\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}[(n-1)-(i+1)+1]=\sum_{i=0}^{n-2}(n-1-i)$$

$$=\sum_{i=0}^{n-2}(n-1)-\sum_{i=0}^{n-2}i=(n-1)\sum_{i=0}^{n-2}1-\frac{(n-2)(n-1)}{2}$$

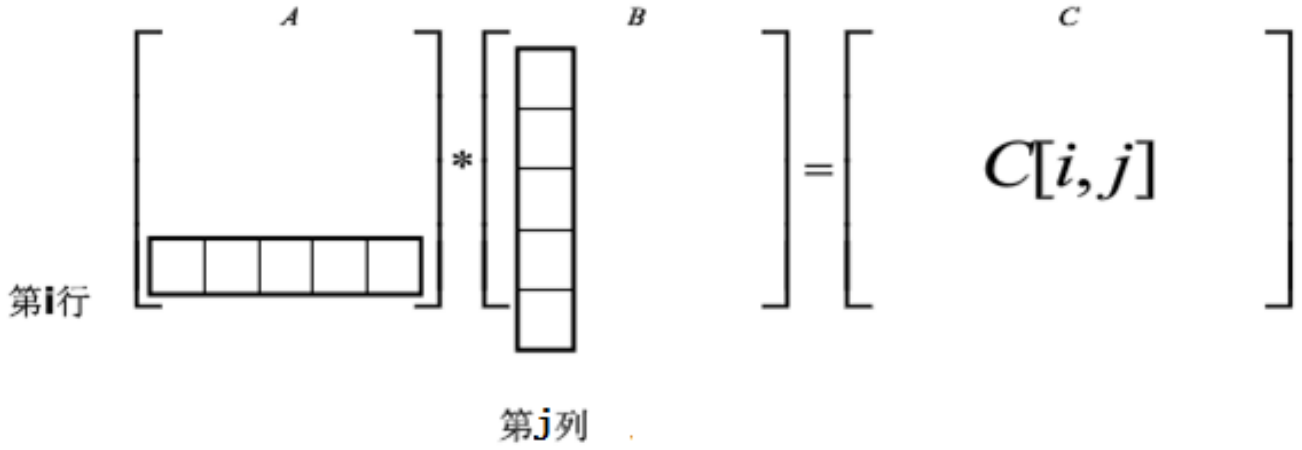
$$=(n-1)^2-\frac{(n-2)(n-1)}{2}=\frac{(n-1)n}{2}\approx\frac{1}{2}n^2\in\Theta(n^2)$$

分析非递归算法效率的通用方案

1. 决定用那些参数作为**输入规模**的度量。
2. 找出算法的**基本操作**。
   - 作为一规律，它总是位于算法的最内层循环中
3. 检查基本操作的**执行次数**是否只依赖输入规模。
   - 如果它还依赖一些其他的特性，则**最差效率**、**平均效率**以及**最优效率**需要分别研究
4. **建立一个算法基本操作执行次数的求和表达式**。
5. 利用**求和运算的标准公式**和**法则**来建立一个操作次数的**闭合公式**，确定它的**增长次数**。

例2： 矩阵相乘

定义，C是一个n阶方阵，它的每个元素都是矩阵A的行和B的列的点积：



对于 $i >= 0$ 和 $j <= 0$ 的每一对下标，

$$C[i,j]=A[i,0]B[0,j]+\ldots+A[i,k]B[k,j]+\ldots+A[i,n-1]B[n-1,j]$$

例2： 矩阵相乘

```
算法 MatrixMuti(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])
//根据定义计算两个n阶矩阵的乘积
//输入：两个n阶矩阵
//输出：矩阵C=AB
for i←0 to n-1 do
    for j←0 to n-1 do
        C[i,j] ←0
        for k ←0 to n-1 do
            C[i,j] ←C[i,j] + A[i,k] * B[k,j]
return C
```

分析

```
for i←0 to n-1 do
    for j←0 to n-1 do
        C[i,j] ←0
        for k ←0 to n-1 do
            C[i,j] ←C[i,j] + A[i,k] * B [k,j]
return C
```

输入规模：n

基本操作：乘法

基本操作执行次数表达式：

$$\mathrm{M}(n)=\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}1=\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}n=\sum_{i=0}^{n-1}n^2=n^3$$

增长次数：$\Theta(n^3)$

经常用到的求和公式

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \in \Theta(n^2)$$

$$\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6} \in \Theta(n^3)$$

$$\sum_{j=0}^{n} c^j = \frac{c^{n+1}-1}{c-1} \in \Theta(c^n),\ c \neq 1$$

$$\sum_{j=0}^{n} jc^j \in \Theta(nc^n),\ c \neq 1$$

$$\sum_{j=0}^{\infty} c^j = \frac{1}{1-c} \in \Theta(1),\ |c|<1$$

$$\sum_{j=0}^{\infty} jc^j = \frac{c}{(1-c)^2} \in \Theta(1),\ |c|<1$$

$$\sum_{j=0}^{n} 2^j = \frac{2^{n+1}-1}{2-1} \in \Theta(2^n)$$

$$\sum_{j=0}^{\infty} \frac{1}{2^j} = 2 \in \Theta(1)$$

其他的重要公式

欧拉常数

$$\sum_{j=1}^{n} j^k \in \Theta(n^{k+1}),\ k \geq 1$$

$$\sum_{j=1}^{n} \frac{1}{j} \approx \ln n + 0.5772 \in \Theta(\log n)$$

$$\sum_{j=1}^{n} \log j \in \Theta(n \log n)$$

$$\ln x = \int_1^x \frac{1}{t} dt$$

$$e(\frac{n}{e})^n \leq n! \leq ne(\frac{n}{e})^n$$

$$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n \text{ (Stirling's formula)}$$

$$e = \lim_{n\to\infty}(1+\tfrac{1}{n})^n = 1 + \tfrac{1}{1!} + \tfrac{1}{2!} + \tfrac{1}{3!} + \ldots = 2.7182818\ldots$$

课堂练习

- 对于下列每一种函数，指出他们属于哪一种Θ(g(n))类型：

(1). $\sqrt{7n^2+4n+3}$

(2). $2n\lg(n+2)^2+(n+2)^2\lg n$

(3). $\sum_{i=0}^{n-1} i(i-1)$

(4). $\sum_{i=2}^{n-1} \log_2 i^2$

2.4 递归算法的数学分析

- 例：对于任意非负整数*n*，计算***F(n)=n!***的值。

F(n)= F(n-1)*n

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

```
算法  F(n)
//递归计算n!
//输入：非负整数n
//输出：n!的值
if n=0
        return 1
else
        return F(n-1)*n
```

- 该算法的**基本运算**是：**乘法**
- 把执行次数记作M(n)，有：

$$M(n)=\begin{cases} M(n-1)+1 & ,\ n\geq 1 \\ 0 & ,\ n=0 \end{cases}$$

- M(n-1) 用于计算 F(n-1)
- 1 用于将 F(n-1) 乘以n

```
算法  F(n)
//递归计算n!
//输入：非负整数n
//输出：n!的值
if n=0
        return 1
else
        return F(n-1)*n
```

反向替换法

- M(n)=M(n-1)+1
- =[M(n-2)+1]+1=M(n-2)+2
- =[M(n-3)+1]+2=M(n-3)+3
- ..... =M(n-n)+n=M(0)+n=n

- 所以算法复杂度为：**Θ(n)**

分析递归算法效率的通用方案

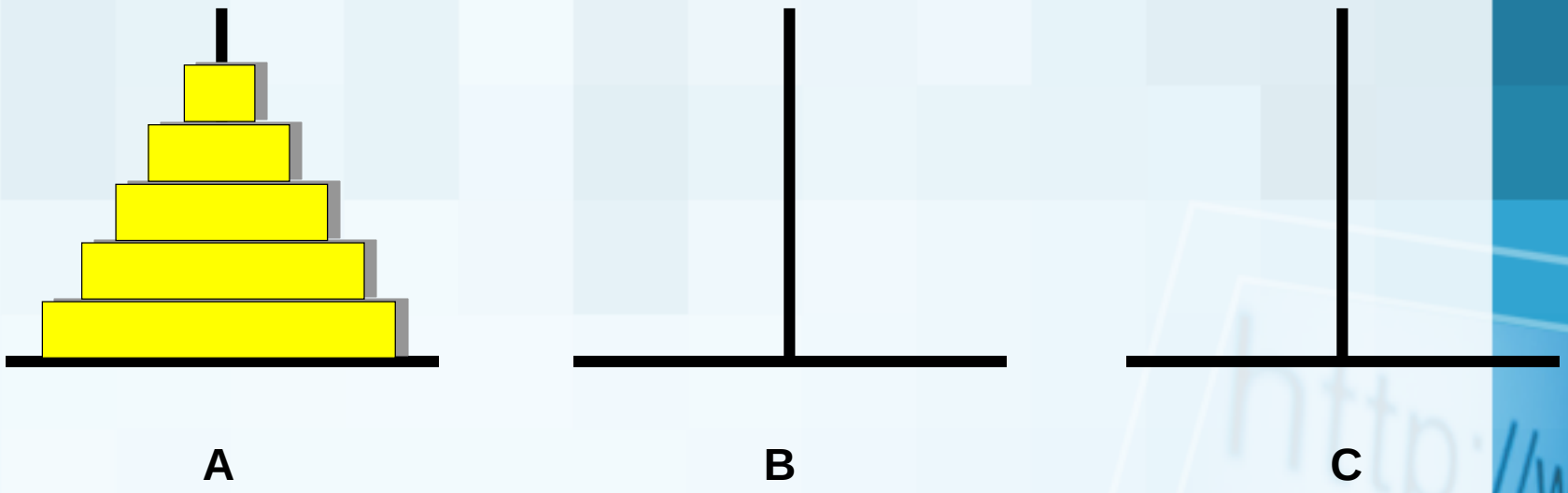
- 决定用哪个参数作为**输入规模的度量**
- 找出算法的**基本操作**
- 检查对相同规模的输入，基本操作的执行次数是否相同，如果不同，必须对**最差、平均**及**最优效率**单独研究

◆对于算法**基本操作的执行次数**，建立一个**递推关系**以及**相应的初始条件**

◆**求解**这个**递推关系式**，确定**增长次数**

汉诺塔： 源于印度一个古老传说的益智玩具

从A移到C，利用B



关键假设： 已知如何移动4个圆盘，现在来移动5个圆盘

汉诺塔

定义 Move(SET, from, through, to)

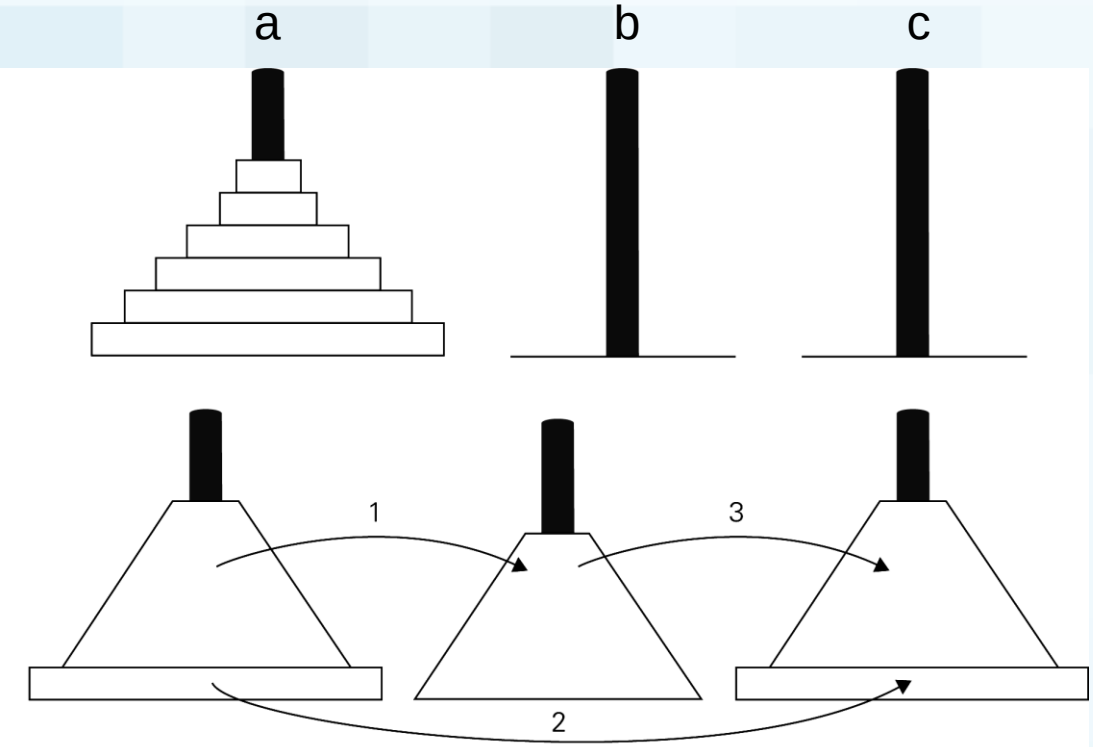


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle

算法描述：

```
Move([1..n], a, b, c)
{
1. If n=1 Then move(n, a, c)
2. Else Move([1..n-1], a, c, b)
3.          move(n, a, c)
4.          Move([1..n-1], b, a, c)
}
```

递归调用

```
Move([1..n-1], a, c, b)
Move([1..n-2], a, b, c)
Move(n-1,a,b)
Move([1..n-2], c, a, b)
…
```

汉诺塔

递归算法：

```
Move([1..n], a, b, c)
{
1. If n=1 Then move(n, a, c)
2. Else Move([1..n-1], a, c, b)
3.         move(n, a, c)
4.         Move([1..n-1], b, a, c)
}
```

递推关系：

$$M(n)=\begin{cases}2M(n-1)+1 & ,\ n>1\\ 1 & ,\ n=1\end{cases}$$

反向替换求解

得：$M(n)=2^n-1 \in \Theta(2^n)$

经验：我们应该谨慎使用递归算法，
因为他们的简洁可能会掩盖他们的低效率。

算法效率分析基础

- 斐波那契（ Fibonacci ）数列
- 求解递推关系式
- **2.5 算法的经验分析**

2.5 斐波那契（ Fibonacci ）数列

Fibonacci 数列—0,1,1,2,3,5,8,13,21,34,…

这个数列可以用一个**简单的递推式**定义：

$$F(n)=\begin{cases} F(n-1)+F(n-2) & , n>1 \\ 1 & , n=1 \\ 0 & , n=0 \end{cases}$$

利用求解**常系数齐次二阶线性递推关系**的方法得：

$$F(n)=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$$F(n)=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n=\frac{1}{\sqrt{5}}(\varphi^n-\hat{\varphi}^n)$$

$$F(n)\approx\frac{1}{\sqrt{5}}\varphi^n$$

$\varphi=\frac{1+\sqrt{5}}{2}\approx 1.61803$ 就是常说的黄金分割比

$F(n)\in\Theta(1.61803^n)$

Fibonacci 数列 算法

由于这个数列可以用一个**简单的递推式**和**两个初始条件**来定　　**当n>1时，F(n)=F(n-1)+F(n-2)**

初始条件： F(0)=0,F(1)=1

递归算法 F(n)

//根据定义，递归计算第n个斐波那契数

//输入：一个非负整数n

//输出：第n个斐波那契数

if n≤1

return n

else

*return **F(n-1)+F(n-2)***

算法效率分析

```
算法 F(n)
if n≤1
    return n
else
    return F(n-1)+F(n-2)
```

基本操作：加法

A(n)：这个算法在**计算F(n)的过程中**所做的**加法次数**。

当n>1时，A(n)=A(n-1)+A(n-2)+1

初始条件：A(0)=0; A(1)=0

比较A(n)和Fibonacci数列F(n)，可知：

$$A(n)=F(n+1)-1 \in \Theta(1.61803^n)$$

算法效率分析

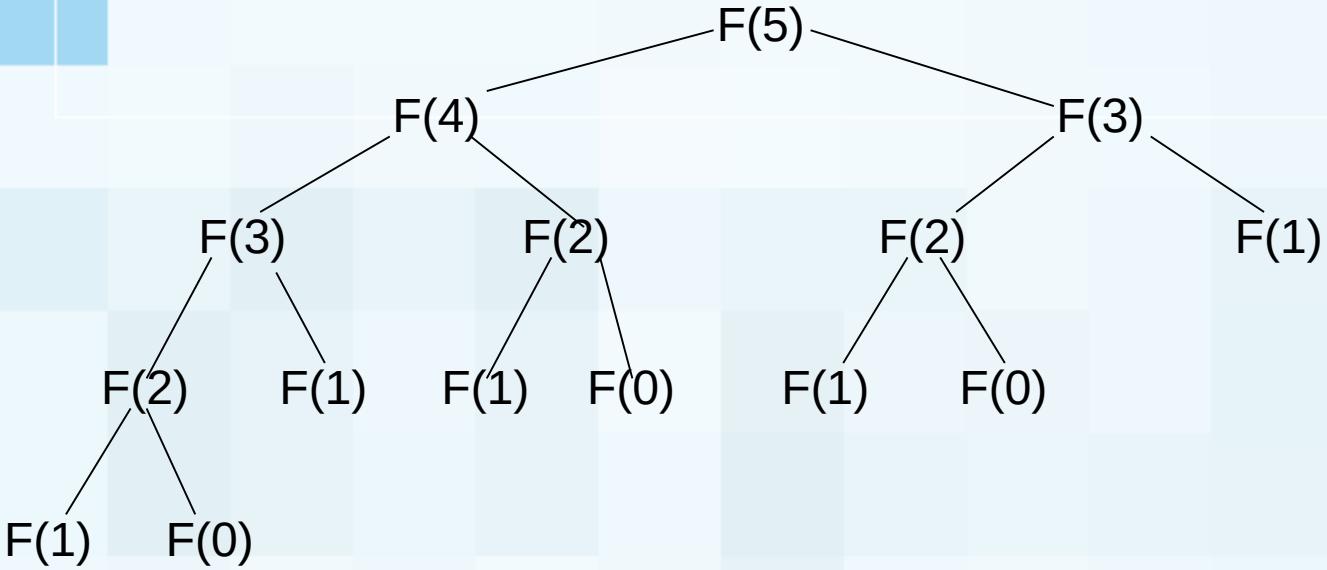
- $A(n) \in \Theta(1.61803^n)$

结论：

该计算Fibonacci 数列的算法效率很低！

问题：此算法为何低效？

能否改进此算法？

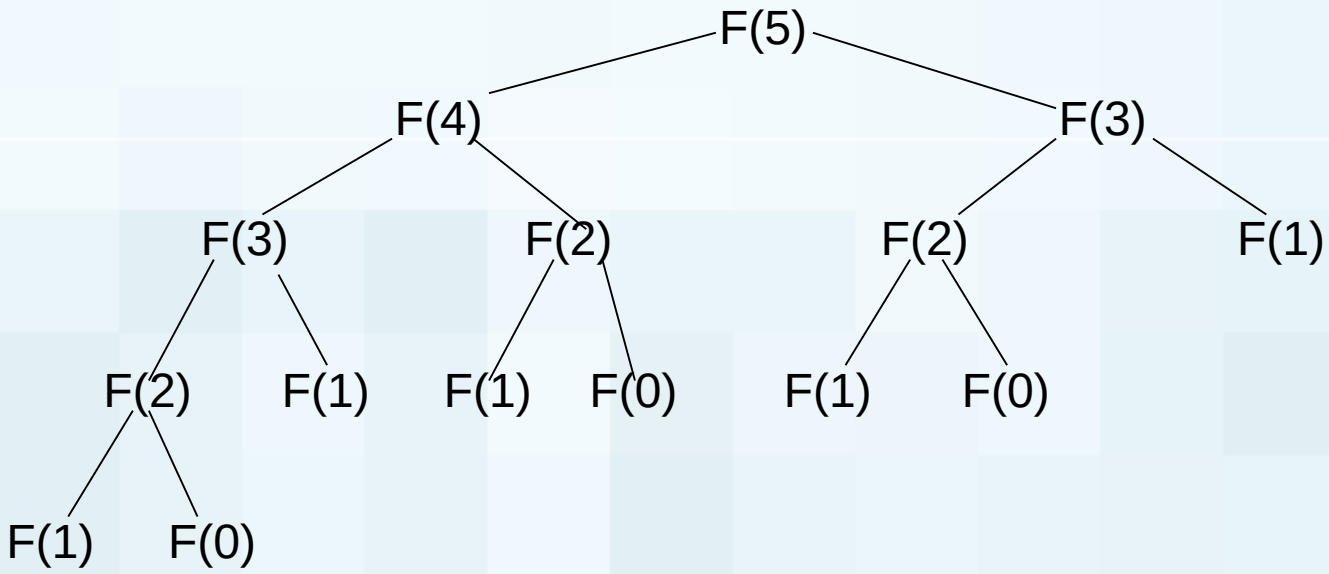



```
算法 F(n)
if n≤1
    return n
else
    return F(n-1)+F(n-2)
```

n=5时，计算Fibonacci 数列的递归调用树

通过观察该算法的递归调用树，我们可以发现该算法效率低下的原因：

相同的函数值被一遍一遍地重复计算！



n=5时，计算Fibonacci 数列的递归调用树

问题：如何避免相同的函数值被重复计算？

Fibonacci 数列高效算法

通过对Fibonacci 数列的**元素连续进行自底向上的迭代计算**，可得到一个高效的算法：

算法 Fib(n)
//根据定义，迭代计算第n个Fibonacci数
//输入：一个非负整数n
//输出：第n个Fibonacci数
F[0]←0; F[1]←1
for i←2 to n do
    F[i]←F[i-1]+F[i-2]
return F(n)

基本操作：加法

运算次数：**n-2+1=n-1**

算法效率类型：**Θ(n)**

求 Fibonacci 数列的各种算法

1. 递归算法：

   Definition-based recursive algorithm

   效率：$\Theta(1.61803^{n})$

2. 非递归迭代算法：

   Nonrecursive definition-based algorithm

   效率：$\Theta(n)$

求 Fibonacci 数列的高效算法

3. 给定公式算法：Explicit formula algorithm，

$$F(n)=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

效率：取决于计算 c^n

利用好的算法可得**Θ(log n)的效率，但会受计算精度的影响！**

4. 更好的算法：矩阵算法

$$\begin{pmatrix} F(n\text{-}1) & F(n) \\ F(n) & F(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$$

效率： **Θ(log n)**

计算整数次幂

- C^n =Y*Y （n为偶数）
- C^n =Y*Y*C （n为奇数）
- Y=C^(⌊n/2⌋)
- M(1)=0
- M(n)<=M(⌊n/2⌋)+2
- n=2^k => k=log2(n)
- M(n)=M(n/2)+2=M(n/2^2)+2+2
  =M(n/2^i)+2*i = 2*k =2*log(n)

如何求解递推关系式

- 反向替换法
  - 从 T(n) 开始替换
- 正向替换法
  - 从 T(1) 开始替换
- 公式法

- **反向替换法**

已知：*x(n)=3 x(n-1), x(1)=4*

$$x(n) = 3x(n-1)$$

$$= 3[3x(n-2)] = 3^2 x(n-2)$$

$$= 3^2[3x(n-3)] = 3^3 x(n-3)$$

$$= \ldots$$

$$= 3^i x(n-i)$$

(当i=n-1时)

$$= 3^{n-1} x(1) = 4 \cdot 3^{n-1}$$

- **正向替换法：**

$x(n)=x(n-1)+n,\ x(0)=0$

- $x(1)=x(0)+1$
- $x(2)=x(1)+2=x(0)+1+2$
- $x(3)=x(2)+3=x(0)+1+2+3$
- $x(n)=x(0)+1+2+3+\ldots+n=n(n+1)/2$

公式法：两种重要的递归类型

- **减一类型 Decrease-by-one**
  - 对应第5章的**减治法**
  - Example: n!
  - **典型递推式：** $T(n) = T(n-1) + f(n)$
- **减常因子类型 Decrease-by-a-constant-factor**
  - 对应第4章的**分治法**
  - Example: binary search.
  - **典型递推式：** $T(n) = aT(n/b) + f(n)$

减一类型（减治法）
Decrease-by-one Recurrences

- One (constant) operation reduces problem size by one.

  $T(n) = T(n-1) + c$ $\qquad$ $T(1) = d$

  公式: $\qquad$ $\mathbf{T(n) = (n-1)c + d}$

- A pass through input reduces problem size by one.

  $T(n) = T(n-1) + c\,n$ $\qquad$ $T(1) = d$

  公式: $\qquad$ $\mathbf{T(n) = [n(n+1)/2 - 1]\,c + d}$

减常因子类型 (分治法)

$T(n) = aT(n/b) + f(n)$, where $f(n) \in \Theta(n^k)$,

要求： $k \geq 0$, $a \geq 1$, $b > 1$

附录B3 定理5 （364页）

The Master Theorem（主定理，重要！）

$$T(n) = \begin{cases} \Theta(n^k) & a < b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^{\log_b a}) & a > b^k \end{cases}$$

The Master Theorem 的应用

$T(n) = aT(n/b) + f(n)$, where $f(n) \in \Theta(n^k)$, $k \geq 0$，$a \geq 1$，$b > 1$

The Master Theorem（主定理）

$$T(n) = \begin{cases} \Theta(n^k) & a < b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^{\log_b a}) & a > b^k \end{cases}$$

- T(n) = T(n/2) + n — **a=1, b=2, f(n)=n, k=1, $a<b^k$** $\Theta(n)$
- T(n) = 2 T(n/2) + n — **a=2, b=2, f(n)=n, k=1, $a=b^k$** $\Theta(n\log n)$
- T(n) = 3 T(n/2) + n — **a=3, b=2, f(n)=n, k=1, $a>b^k$** $\Theta(n^{\log_2 3})$

递推关系： $x(n)=x(n/3)+1,\ n>1;\ x(1)=1$

解法1：反向替换法

需用到　附录B3 定理4　平滑法则

平滑法则：设T(n)是一个**最终非递减**的函数，而f(n)是**平滑函数**。

如果 **n是b的幂时，$T(n)\in\Theta(f(n))$**，其中 $b\geq 2$，

那么 **$T(n)\in\Theta(f(n))$**

（此定理对于O和Ω的情况也成立）

定义：

最终非递减：$\exists n_0 \geq 0$，使得$n>n_0$时，函数是非递减的。

平滑函数：f(n)最终非递减，并且 $f(2n)\in\Theta(f(n))$，

例如：$\log n$, n, n^a 等增长慢的函数都是，但 $a^n(a>1)$和n!都不是。

递推关系： $x(n)=x(n/3)+1,\ n>1;\ x(1)=1$

解法1：反向替换法

Only solve for $n=3^k$!

$$
\begin{aligned}
x(3^k) &= x(3^{k-1})+1 \\
&= [x(3^{k-2})+1]+1 = x(3^{k-2})+2 \\
&= [x(3^{k-3})+1]+2 = x(3^{k-3})+3 \\
&= \ldots \\
&= x(3^{k-i})+i \quad = \ldots \qquad (let\ \ i=k) \\
&= x(3^{k-k})+k = x(1)+k = 1+\log_3 n
\end{aligned}
$$

$log_3 n = logn / log3$

复杂度： $\Theta(logn)$

递推关系： $x(n)=x(n/3)+1,\ n>1;\ x(1)=1$

解法2：公式法

利用 主定理 （**The Master Theorem**）

$T(n) = aT(n/b) + f(n)$,
where $f(n) \in \Theta(n^k)$

a=1, b=3,

f(n)=1, k=0,

$a=b^k$, $n^0 \log n = \log n$

复杂度： $\Theta(\log n)$

$$T(n)=\begin{cases}\Theta(n^k) & a<b^k \\ \Theta(n^k \log n) & a=b^k \\ \Theta(n^{\log_b a}) & a>b^k\end{cases}$$

2.6 算法的经验分析

做实验分析**实验结果**。。。

- 对算法效率做经验分析的**通用方案**
  - 了解试验的**目的**
    - 验证算法效率理论结果的准确性及精确性
    - 相同问题不同算法
    - 相同算法不同程序的实现
  - 决定用来度量效率的**度量单位**
    - **基本操作次数**
    - 程序的**运行时间**
    - 不同的硬件运行环境（系统时间，系统分时操作）
    - 多次运行，求均值

2.6 算法的经验分析

- 对算法效率做经验分析的通用方案
  - 决定**输入样本的特性**
    - **样本的规模（小->大）**
    - 样本的**随机生成**（典型输入）
  - **算法的程序实现**
  - **生成输入样本**
  - 对输入样本**进行计算，**并**记录结果**
  - **分析获得的实验数据**
    - **散点图**

小结

- 算法效率包括**时间效率**和**空间效率**。
- **时间效率**主要用它的**输入规模的函数**来度量，该函数计算算法**基本操作**的**执行次数**。
- 最差、最优与平均效率
- **增长次数**
- **符号O，Ω，Θ**
- **非递归算法**和**递归算法**的效率分析
- 递归算法简洁性可能会掩盖它的低效率
- 算法的**经验分析**是针对一个输入样本，运行算法的一个程序实现，然后分析观测到的数据。

课堂练习

1、利用反向替换求解：

$T(n) = T(n-1)+2, \quad T(1)=0$

2、利用 The Master Theorem 求解：

$T(n) = 4T(n/2)+n^2+3n$