程序设计方法与实践

——算法效率分析基础

算法效率分析基础

- 1、效率分析基本概念
- 2、渐进符号和基本效率类型
- 3、非递归算法的数学分析
- 4、递归算法的数学分析

效率分析基本概念

- 算法的特征
  - **1.可行性：**针对实际问题设计的算法，人们总是希望能够得到满意的结果。
  - **2.确定性：**算法的确定性，是指算法中的每一个步骤都必须是有明确定义的，不允许有模棱两可的解释，也不允许有多义性。
  - **3.有穷性：**算法的有穷性，是指算法必须能在有限的时间内做完，即算法必须能在执行有限个步骤之后终止。
  - **4.输入：**通常，算法中的各种运算总是要施加到各个运算对象上，而这些运算对象又可能具有某种初始状态，这是算法执行的起点或是依据。
  - **5.输出：**一个算法有一个或多个输出，以反映对输入数据加工后的结果。

效率分析基本概念

- 算法的评价
  - 1.正确性

    正确性是指算法的执行结果应该满足预先规定的功能和性能要求
  - 2.可读性

    一个算法应该思路清晰、层次分明、简单明了、易读易懂
  - 3.健壮性

    算法的健壮性指的是，算法应对非法输入的数据做出恰当反映或进行相应处理
  - 4.复杂性

    算法的复杂性是算法**效率**的度量，是评价算法优劣的重要依据，算法的复杂性有**时间复杂性和空间复杂性**之分

效率分析基本概念

- 算法分析是对算法利用两种资源的效率做研究
  - 运行时间——时间复杂度
  - 存储空间——空间复杂度
  - ~~简单性、一般性-难以衡量~~
- 效率度量函数
  - 规模越大的输入，耗费时长越长，时间效率越低
  - 效率度量函数*T(n)*，算法输入规模***n***为参数的函数
    - 排序查找算法。规模是列表长度
    - 多项式$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$。多项式次数或系数个数
    - 两个n阶矩阵乘积。矩阵阶数n，或参加运算的矩阵元素个数N
    - 拼写检查算法。字符个数，或词个数
    - 数字特征相关。如计算数字的二进制位数，b=$\lfloor \log_2 n \rfloor$ +1

效率分析基本概念

- 运行时间的度量单位
  - 时间的标准度量单位：秒，毫秒等。
  - 统计算法每一步操作的执行次数。
  - 基本操作(Basic Operation)：算法中最重要和最主要的操作。

基本操作：算法最内层循环中最费时的操作

- 算法分析基本框架
  - 对于输入规模为n的算法，统计他的基本操作执行次数，对效率进行度量。

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

  - 考虑规模n增大，运行时间的变化。

效率分析基本概念

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

- 考虑当输入规模n增大时，运行时间的如何变化。

例如：$C(n) = \frac{1}{2}n(n-1)$

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

$$\frac{T(2n)}{T(n)} = \frac{c_{op}C(2n)}{c_{op}C(n)} = \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4$$

c_{op}被约去了，$\frac{1}{2}n$被忽略了，$\frac{1}{2}n^2$的系数$\frac{1}{2}$被约去了。

效率分析基本概念

- 增长次数
  - 为什么对于大规模的输入要强调执行次数的增长次数呢？

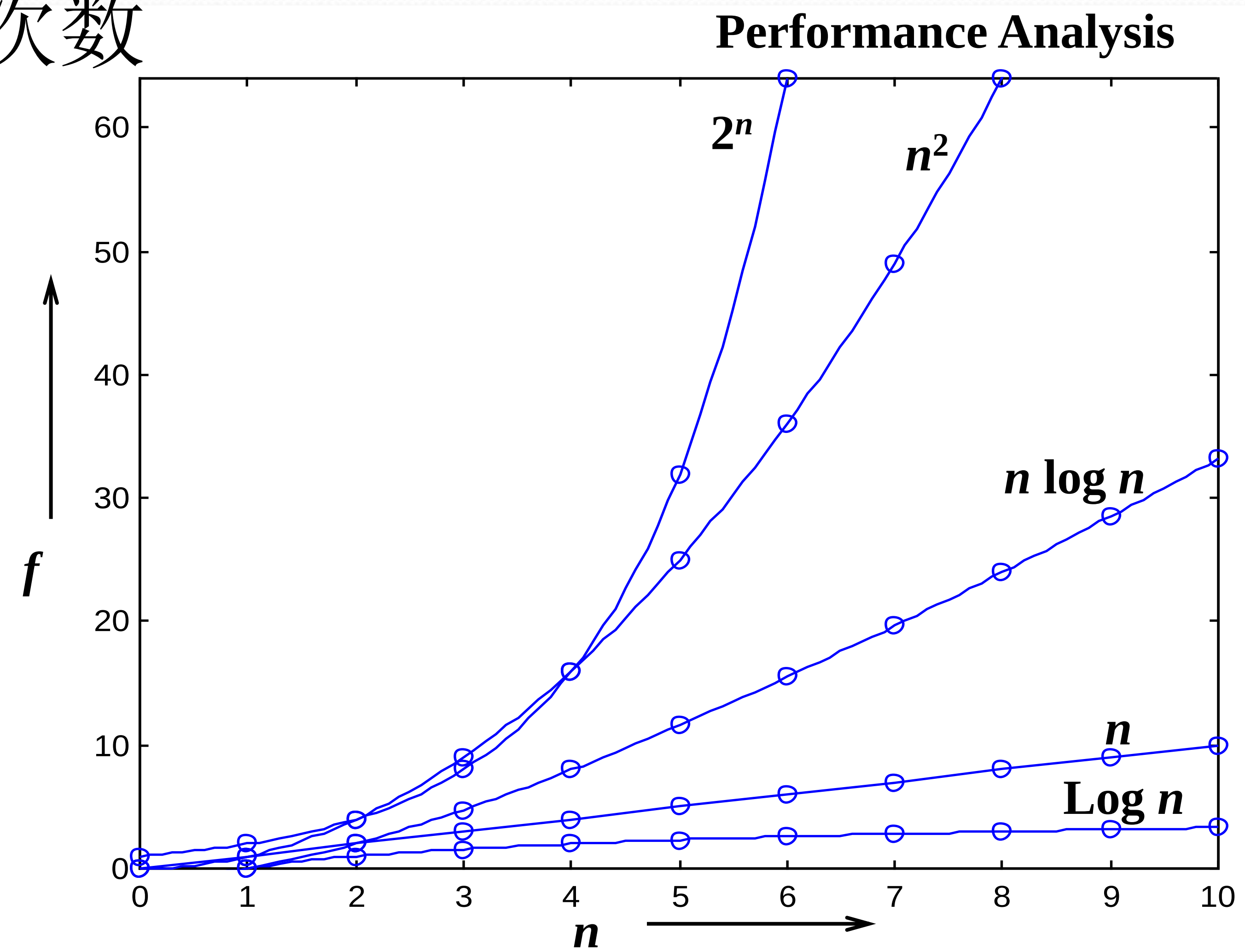
    这是因为小规模输入在运行时间上差别不足以将高效的算法和低效的算法法区分开来。

表 2.1　对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	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	1.0×10^4	10^6	10^9		
10^4	13	10^4	1.3×10^5	10^8	10^{12}		
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}		
10^6	20	10^6	2.0×10^7	10^{12}	10^{18}		

效率分析基本概念

- 增长次数



效率分析基本概念

8. 对于下列每种函数，请指出当参数值增加到 4 倍时，函数值会改变多少。

a. $\log_2 n$　**b.** $\sqrt{n}$　**c.** n　**d.** n^2　**e.** n^3　**f.** 2^n

9. 请指出下面每一对函数中，第一个函数的增长次数(包括其常数倍)比第二个函数的增长次数大还是小，还是二者相同。

a. $n(n+1)$和 $2000n^2$　**b.** $100n^2$ 和 $0.01n^3$

c. $\log_2 n$ 和 $\ln n$　**d.** $\log_2^2 n$ 和 $\log_2 n^2$

e. 2^{n-1} 和 2^n　**f.** $(n-1)!$和 $n!$

效率分析基本概念

8. a. $\log_2 4n - \log_2 n = (\log_2 4 + \log_2 n) - \log_2 n = 2.$

   b. $\dfrac{\sqrt{4n}}{\sqrt{n}} = 2.$

   c. $\dfrac{4n}{n} = 4.$

   d. $\dfrac{(4n)^2}{n^2} = 4^2.$

   e. $\dfrac{(4n)^3}{n^3} = 4^3.$

   f. $\dfrac{2^{4n}}{2^n} = 2^{3n} = (2^n)^3.$

效率分析基本概念

9. a. $n(n+1) \approx n^2$ has the same order of growth (quadratic) as $2000n^2$ to within a constant multiple.

b. $100n^2$ (quadratic) has a lower order of growth than $0.01n^3$ (cubic).

c. Since changing a logarithm's base can be done by the formula

$$\log_a n = \log_a b \log_b n,$$

all logarithmic functions have the same order of growth to within a constant multiple.

d. $\log_2^2 n = \log_2 n \log_2 n$ and $\log_2 n^2 = 2 \log n$. Hence $\log_2^2 n$ has a higher order of growth than $\log_2 n^2$.

e. $2^{n-1} = \frac{1}{2} 2^n$ has the same order of growth as 2^n to within a constant multiple.

f. $(n-1)!$ has a lower order of growth than $n! = (n-1)!n$.

效率分析基本概念

- 运行时间不仅取决于输入的规模，而且取决于特定输入细节，则需分析算法最优、最差和平均效率。
  - 最差效率：当输入规模为n时算法在最坏情况下的效率。
  - 最优效率：当输入规模为n时算法在最优情况下的效率。
  - 平均效率：在“典型”和随机输入情况下，算法的行为和效率。

效率分析基本概念

- 算法最优、最差和平均效率
  - 运行时间不仅取决于输入的规模，而且取决于特定输入细节。
  - 例如：顺序查找

算法 SequentialSearch($A[0 \ldots n-1], K$)

//用顺序查找在给定的数组中查找给定的值

//输入：数组A和查找键K

//输出：返回第一个匹配K的元素下标

$i \leftarrow 0$

While $i < n$ and $A[i] \neq K$ **do**

$i \leftarrow i + 1$

$\boldsymbol{if}$ $i < n$ **return** i

Else return -1.

效率分析基本概念

- 算法最优、最差和平均效率
  - 最差效率：当输入规模为n时算法在最坏情况下的效率。 (没有匹配元素或是最后一个元素) $C_{worst}=n$
  - 最优效率：当输入规模为n时算法在最优情况下的效率。 (第一个元素就是键K) $C_{best}=1$
  - 平均效率：在“典型”和随机输入情况下，算法的行为和效率。(随机输入的情况下) 设成功的概率p

$$C_{avg}(n)=\left[1\times\frac{p}{n}+2\times\frac{p}{n}+\cdots+n\times\frac{p}{n}\right]+n\times(1-p)$$

$$C_{avg}(n)=\begin{cases}(n+1)/2 & p=1\\ n & p=0\\ \frac{p(n+1)}{2}+n(1-p) & else\end{cases}$$

效率分析基本概念

- 效率分析基础总结
  - 算法的时间效率用**输入规模的函数**度量。
  - **输入数据**经过算法**各种操作**处理后输出结果。
  - 在所有操作中，**基本操作**是最主要和最核心的。
  - 算法**基本操作的执行次数**度量算法的时间效率。
  - 输入规模确定时，有些算法效率会有差异，则需要**区分最优、最差和平均效率**。

算法效率分析基础

- 1、效率分析基本概念
- 2、渐进符号和基本效率类型
- 3、非递归算法的数学分析
- 4、递归算法的数学分析

渐进符号和基本效率类型

- 符号介绍
  - $t(n)$表示算法运行时间（常用基本操作次数C(n) 表示）
  - $g(n)$是用于和操作次数做比较的函数。(简单易理解)
  - O(读作O)，$O(g(n))$是增长次数小于等于$g(n)$（及其常数倍,n趋向于无穷大）的函数集合。$n \in O(n^2)$
  - Ω (读作omega), $\Omega(g(n))$是增长次数大于等于$g(n)$（及其常数倍,n趋向于无穷大）的函数集合。$n^3 \in \Omega(n^2)$
  - Θ (读作$theta$)，$\Theta(g(n))$是增长次数等于$g(n)$（及其常数倍,n趋向于无穷大）的函数集合。$an^2 + bn + c \in \Theta(n^2)$

渐进符号和基本效率类型

- 大O表示法

定义:

如果函数$t(n)$包含在$O\big(g(n)\big)$中，记作$t(n) \in O(g(n))$。它的成立条件是：对于所有足够大的n，$t(n)$的上界由$g(n)$的常数倍所确定，也就是说，存在大于0的常数c和非负的整数n_0，使得：

对于所有的$n \geq n_0$来说，$t(n) \leq cg(n)$

例如：

$$100n + 5 \leq 100n + n(\text{当}n \geq 5) = 101n \leq 101n^2$$

因此，$100n + 5 \in O(n^2)$

理解：我们的算法比（n^2）规模增长慢，效率更高

渐进符号和基本效率类型

- 大Ω表示法

定义:

如果函数$t(n)$包含在$\Omega\left(g(n)\right)$中。它的成立条件是：对于所有足够大的n，$t(n)$的下界由$g(n)$的常数倍所确定，也就是说，存在大于0的常数c和非负的整数n_0，使得：

对于所有的$n \geq n_0$来说，$t(n) \geq cg(n)$

例如：

当$n \geq 0$时，$n^3 \geq n^2$，也就是说，可以选择$c = 1, n_0 = 0$，从而，$n^3 \in \Omega(n^2)$

理解：我们的算法比（n^2）规模增长快，效率更低

渐进符号和基本效率类型

- 小o表示法:严格小于
  - $o\big(g(n)\big) = \{f(n)$,对于任意c，存在n_0,使得 $f(n) < cg(n)$ 当$n \geq n_0$时候$\}$

$$2n^2 \in o(n^3) \qquad n_0\text{取 } 2/c$$

- 小ω表示法:严格大于
  - $\omega\big(g(n)\big) = \{f(n)$,对于任意$c$，存在$n_0$,使得 $f(n) > cg(n)$ 当$n \geq n_0$时候$\}$

$$2n^3 \in \omega(n^2) \qquad n_0\text{取 } c/2$$

渐进符号和基本效率类型

- 大Θ表示法

定义:

如果函数$t(n)$包含在$\Theta\left(g(n)\right)$中，它的成立条件是：对于所有足够大的n，$t(n)$的上界和下界由$g(n)$的常数倍所确定，也就是说，存在大于0的常数c_1、c_2和非负的整数n_0，使得：

对于所有$n \geq n_0$，$c_2 g(n) \leq t(n) \leq c_1 g(n)$

例如：

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

理解：我们的算法跟n^2差不多

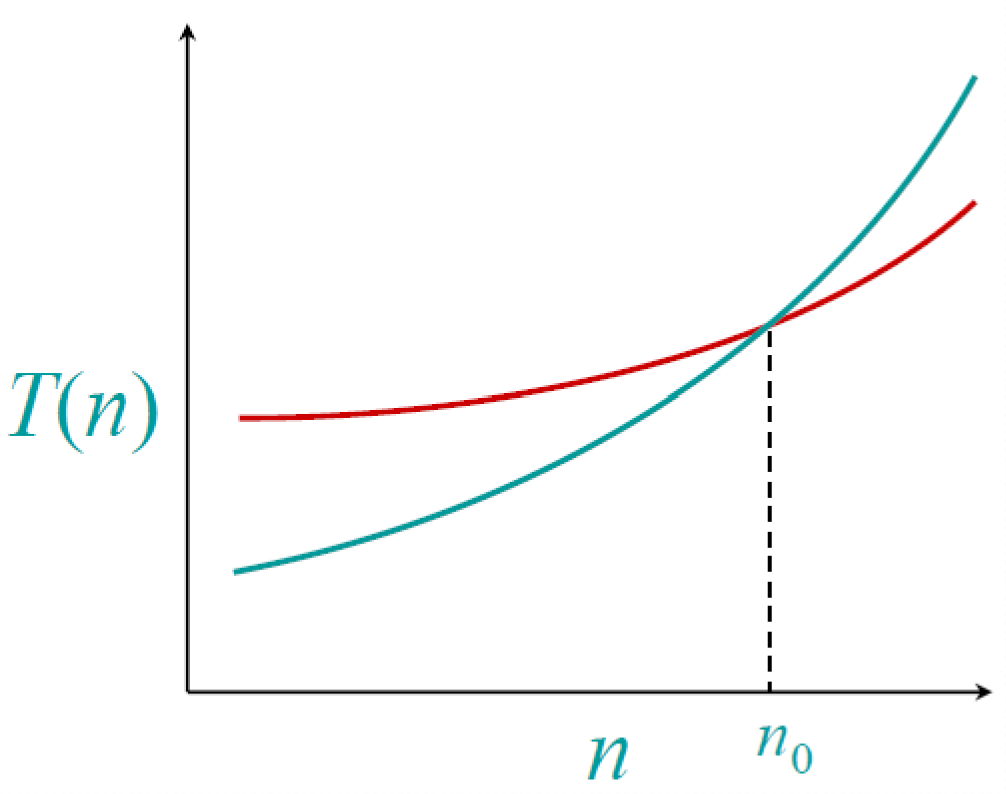
Θ(g(n)) = O(g(n)) ∩ Ω(g(n))

丢弃低次项，忽略常数项

例如：$\boldsymbol{3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)}$

渐进符号和基本效率类型

- 渐近分析
  - 当n足够大时，$\Theta(n^2)$算法总是优于$\Theta(n^3)$.



算法设计和**工程目的**
之间的平衡

渐进符号和基本效率类型

- 渐进符号的有用特性—加法规则

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Ω和Θ符号，类似的断言也为真)

对于两个连续执行部分组成的算法，应该如何应用这个特性呢？它意味着该算法的整体效率是由具有较大的增长次数的部分所决定的，即它的效率较差的部分.

渐进符号和基本效率类型

$t(n, m)=t_1(n)+t_2(m) \in O(\max(f(n), g(m)))$

两个并列循环的例子

```
void example (float x[ ][ ], int m, int n, int k)
{
        float sum [ ];
        for ( int i=0; i<m; i++ ) {   //x[][]中各行
                sum[i] = 0.0;                  //数据累加
                for ( int j=0; j<n; j++ ) sum[i]+=x[i][j];
        }
        for ( i = 0; i < m; i++ )   //打印各行数据和
                cout << “Line ” << i << “ : ” <<sum [i] << endl;
}
```

渐进时间复杂度$O(max(m*n, m)) = O(m*n)$

渐进符号和基本效率类型

- 渐进符号的有用特性—乘法规则

$t(n,m) = t_1(n) \times t_2(m) \in O(f(n) \times g(m))$

例：求两个n阶方阵的乘积 C＝A∗B

```
#define  n  自然数
MATRIXMLT(float A[n][n],float B[n][n],float C[n][n])
{
    int  i,j,k;
    for(i=0;i<n;i++)                              //n
       for(j=0;j<n;j++) {                         //n*n
          C[i][j]=0;                              //n*n
          for( k=0;k<n;k++)                       //n*n*n
             C[i][j]+=A[i][k]*B[k][j]             //n*n*n
       }
}
```

O(n)

O(n)

O(n)

$t(n) = n^3 \in O(n^3)$

渐进符号和基本效率类型

- 利用极限比较增长次数
- 虽然符号O， Ω和Θ的正式定义对于证明它们的抽象性质是不可缺少的，但我们很少直接用它们来比较两个特定函数的增长次数。
- 有一种较为简便的比较方法，它是基于对所讨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$t(n)\in O(g(n))$,后两种$t(n)\in \Omega(g(n))$,中间$t(n)\in \Theta(g(n))$

渐进符号和基本效率类型

- 利用极限比较增长次数
  - 罗必塔法则

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

  - 史特林公式

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

例1：比较 $\frac{1}{2}n(n-1)$和n^2

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

渐进符号和基本效率类型

- 利用极限比较增长次数
  - 罗必塔法则

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

  - 史特林公式

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

例2：比较$\log_2 n$ 和 $\sqrt{n}$

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{\left(\sqrt{n}\right)'}=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}= 2\log_2 e\lim_{n\to\infty}\frac{1}{\sqrt{n}} = 0$$

渐进符号和基本效率类型

- 利用极限比较增长次数
  - 罗必塔法则

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

  - 史特林公式

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

例3：比较$n!$和2^n

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\,\frac{n^n}{2^n e^n}=\infty$$

渐进符号和基本效率类型

表 2.2　基本的渐近效率类型

类　型	名　称	注　　释
1	常量	为数很少的效率最高的算法，很难举出几个合适的例子，因为典型情况下，当输入的规模变得无穷大时，算法的运行时间也会趋向于无穷大
$\log n$	对数	一般来说，算法的每一次循环都会消去问题规模的一个常数因子(参见 4.4 节)。注意，一个对数算法不可能关注它的输入的每一个部分(哪怕是输入的一个固定部分)：任何能做到这一点的算法最起码拥有线性运行时间
n	线性	扫描规模为 n 的列表(例如，顺序查找)的算法属于这个类型
$n \log n$	线性对数	许多分治算法(参见第 5 章)，包括合并排序和快速排序的平均效率，都属于这个类型
n^2	平方	一般来说，这是包含两重嵌套循环的算法的典型效率(参见下一节)。基本排序算法和 n 阶方阵的某些特定操作都是标准的例子
n^3	立方	一般来说，这是包含三重嵌套循环的算法的典型效率(参见下一节)。线性代数中的一些著名的算法属于这一类型
2^n	指数	求 n 个元素集合的所有子集的算法是这种类型的典型例子。“指数”这个术语常常被用在一个更广的层面上，不仅包括这种类型，还包括那些增长速度更快的类型
$n!$	阶乘	求 n 个元素集合的完全排列的算法是这种类型的典型例子

渐进符号和基本效率类型

1. 从 O, Ω 和 Θ 中选择最合适的符号，指出顺序查找算法的时间效率类型(参见 2.1 节)。
   - **a.** 在最差情况下
   - **b.** 在最优情况下
   - **c.** 在平均情况下

1. a. Since $C_{worst}(n) = n$, $C_{worst}(n) \in \Theta(n)$.

   b. Since $C_{best}(n) = 1$, $C_{best}(1) \in \Theta(1)$.

   c. Since $C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p) = (1-\frac{p}{2})n + \frac{p}{2}$ where $0 \le p \le 1$, $C_{avg}(n) \in \Theta(n)$.

渐进符号和基本效率类型

2. 请用 O，Ω 和 Θ 的非正式定义来判断下列断言是真还是假。

a. $n(n+1)/2 \in O(n^3)$

b. $n(n+1)/2 \in O(n^2)$

c. $n(n+1)/2 \in \Theta(n^3)$

d. $n(n+1)/2 \in \Omega(n)$

2. $n(n+1)/2 \approx n^2/2$ is quadratic. Therefore

a. $n(n+1)/2 \in O(n^3)$ is true.

b. $n(n+1)/2 \in O(n^2)$ is true.

c. $n(n+1)/2 \in \Theta(n^3)$ is false.

d. $n(n+1)/2 \in \Omega(n)$ is true.

算法效率分析基础

- 1、效率分析基本概念
- 2、渐进符号和基本效率类型
- 3、非递归算法的数学分析
- 4、递归算法的数学分析

非递归算法的数学分析

- 例1. 从n个元素的列表中查找最大值。

```
算法 MaxElement A[0 . . n-1]
    maxval ← A [0]
    for i ← 1 to n − 1 do
        if A[i]>maxval
            maxval ← A[i]
    return maxval
```

- 基本操作：即最频繁的操作，循环体内的操作。

$$T(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

非递归算法的数学分析

- 例2. 验证给定数组中n个元素是否全部唯一。

算法 **UniqueElements** $A[0..n\text{-}1]$

for $i \leftarrow 0\ to\ n-2$ do
  for $j \leftarrow i+1\ to\ n-1$ do
    if $A[i]=A[j]$ return false
return true.

- 基本操作：循环体内的操作，两个元素的比较。
- 基本操作次数取决输入规模n和是否有相同元素。
- 最优、最差和平均效率各不相同。
- 最差输入：不包含相同元素或最后两个元素是唯一相同元素的数组。

非递归算法的数学分析

- **例2.** 验证给定数组中n个元素是否全部唯一。

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

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

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

非递归算法的数学分析

- **例3. 计算n阶方阵A和B的乘积C 。**

算法 **MatrixMultiplication** $A[0..n-1,0..n-1]$, $B[0..n-1,0..n-1]$

```
    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.
```

- 乘法计算次数$\boldsymbol{M}(\boldsymbol{n}) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} \mathbf{1} = \boldsymbol{n}^{3}$
- 实际运行时间$\boldsymbol{T}(\boldsymbol{n}) \approx c_m \boldsymbol{M}(\boldsymbol{n}) = c_m \boldsymbol{n}^{3}$
- 精确估值时间$\boldsymbol{T}(\boldsymbol{n}) \approx c_m \boldsymbol{M}(\boldsymbol{n}) + c_a \boldsymbol{A}(\boldsymbol{n}) = (c_m + c_a)\boldsymbol{n}^{3}$

非递归算法的数学分析

- **例4. 十进制正整数用二进制表示的二进制数字个数。**

> 算法 **Binary** n
>
> $count \leftarrow 1$
>
> while $n > 1$ do
>
> $count \leftarrow count + 1$
>
> $n \leftarrow \lfloor n/2 \rfloor$
>
> return $count$.

- 最频繁的操作不是while内部，而是n>1的比较操作。
- 循环次数不是n,而是由n折半次数来确定的
- 每次减少一半$C(n) = \lfloor \log_2 n \rfloor + 1 \approx \log_2 n$

非递归算法的数学分析

- 非递归算法效率分析的通用方案
  1. 寻找算法的<u>**基本操作**</u>（一般位于最内层循环）
  2. 检查基本操作执行次数是否<u>**只依赖于输入规模**</u>。如果她还依赖于其他特性，则需分别研究最差、最优（如有必要）和平均效率。
  3. 建立基本操作执行次数的<u>**求和表达式**</u>。
  4. 利用求和表达式的标准公式和法则建立操作次数的闭合公式。

算法效率分析基础

- 1、效率分析基本概念
- 2、渐进符号和基本效率类型
- 3、非递归算法的数学分析
- 4、递归算法的数学分析

递归算法的数学分析

- 递归相关算法：
  - 反向替换法——找递推关系
  - 替换法——猜测后证明
  - 递归树方法——绘递归树，找关系并证明
  - 主方法Master method——套用3种公式

递归算法的数学分析

- 例1：对任意非负整数n,计算阶乘$F(n) = n!$

```
算法 F(n)
    if n = 0 return 1
    else return F(n-1)*n
```

- 函数关系：当$n > 0$时 $F(n) = F(n-1) \times n$

  $F(0) = 1$
- 乘法数量：当$n > 0$时 $M(n) = M(n-1) + 1$

  $M(0) = 0$
- 反向替换：$M(n) = M(n-1) + 1 = [M(n-2) + 1] + 1$

  $= M(n-2) + 2 = M(n-3) + 3$

  $= M(n-i) + i = n$

递归算法的数学分析

- 例2：汉诺塔。
- 思路：a)把前n-1个盘子从柱1移动到柱2，b)把第n个盘子从柱1移动到柱3，c)把前n-1个盘子从柱2移动到柱3。
- 移动次数：$M(n) = M(n-1) + 1 + M(n-1)$

$$M(1) = 1$$

- 反向替换：$M(n) = 2M(n-1) + 1 = 2[M(n-2) + 1] + 1$

$$= 2^2M(n-2) + 2 + 1 = 2^2[2M(n-3) + 1] + 2 + 1$$
$$= 2^3M(n-3) + 2^2 + 2 + 1$$
$$\ldots$$
$$= 2^iM(n-i) + 2^i - 1 \quad 当i = n-1有效$$
$$= 2^n - 1$$

递归算法的数学分析

- 例3：求十进制正整数的二进制位数。

> 算法 **BinRec(n)**
>   if $n = 1$ return 1
>   else return BinRec($\lfloor n/2 \rfloor$)+1

- 加法次数：$\mathrm{A}(n) = \mathrm{A}(n/2) + 1 \quad A(1) = 0$

当n不是2的乘法时
难以反向替换

- 仅计算n=2^k的情况：

$$\begin{aligned}
\mathrm{A}(2^k) &= \mathrm{A}(2^{k-1}) + 1 \qquad\qquad A(2^0) = 0 \\
&= [\mathrm{A}(2^{k-2}) + 1] + 1 = \mathrm{A}(2^{k-2}) + 2 \\
&= [\mathrm{A}(2^{k-3}) + 1] + 2 = \mathrm{A}(2^{k-3}) + 3 \\
&= \mathrm{A}(2^{k-k}) + k = k
\end{aligned}$$

$$\mathrm{A}(n) = \log_2 n \in \Theta(\log n)$$

递归算法的数学分析

- 递归相关算法：
  - 反向替换法——找递推关系
  - 替换法——猜测后证明
  - 递归树方法——绘递归树，找关系并证明
  - 主方法Master method——套用3种公式

递归算法的数学分析

- 替换法：
  - 1、猜测解的形式。例如n^2
  - 2、归纳法验证是否符合条件
    - 假设规模为k<n时满足条件
      - 验证n时满足条件
      - 验证n=1时满足条件。
  - 3、想办法解出系数

递归算法的数学分析

- 替换法：检查是否正确比较容易，但需要猜测规模

例如：T(n)=4T(n/2)+n

- 1、先简单猜测是$\boldsymbol{n^3}$规模的。
- 2、验证$\mathrm{T(n)} \in O(n^3)$。数学归纳法

假设 $\mathrm{T(k)} \le ck^3$　　　　当$k < n$时满足

$\mathrm{T}(n) = 4\mathrm{T}(n/2) + n$

$\le 4c(n/2)^3 + n = 0.5cn^3 + n = cn^3 - (0.5cn^3 - n)$

$\le cn^3$　　　　当$c \ge 2, n \ge 1$时满足

验证n=1时候

$\mathrm{T}(1) \le c$　　　　满足条件

因此$\mathrm{T(n)} \in O(n^3)$

递归算法的数学分析

替换法 T(n)=4T(n/2)+n

- 1、猜测是$\boldsymbol{n^2}$规模的。
- 2、验证$\mathrm{T(n)} \in O(n^2)$。数学归纳法

假设 $\mathrm{T(k)} \le ck^2$ 当$k < n$时满足

$\mathrm{T}(n) = 4\mathrm{T}(n/2) + n$

$\le 4c(n/2)^2 + n$

$= cn^2 + n$

因为n>0,无法证明$\mathrm{T(n)} \le cn^2$。重新猜测，考虑低阶项

递归算法的数学分析

替换法 $\mathrm{T}(n) = 4\mathrm{T}(n/2) + n$

假设 $\mathrm{T}(\mathrm{k}) \le c_1 k^2 \text{-} c_2 k$　　　当$k < n$时满足

$$\mathrm{T}(n) = 4\mathrm{T}(n/2) + n$$

$$\le 4(c_1(n/2)^2 - c_2(n/2)) + n$$

$$= c_1 n^2 - 2c_2 n + n$$

$$= c_1 n^2 - c_2 n - (c_2 n - n)$$

$\le c_1 n^2 - c_2 n$　　　　　　当$c_2 > 1$时$, c_2 n - n > 0$

验证当n=1时

$\mathrm{T}(1) = \Theta(1) \le c_1 - c_2$　　当$c_1 > c_2$

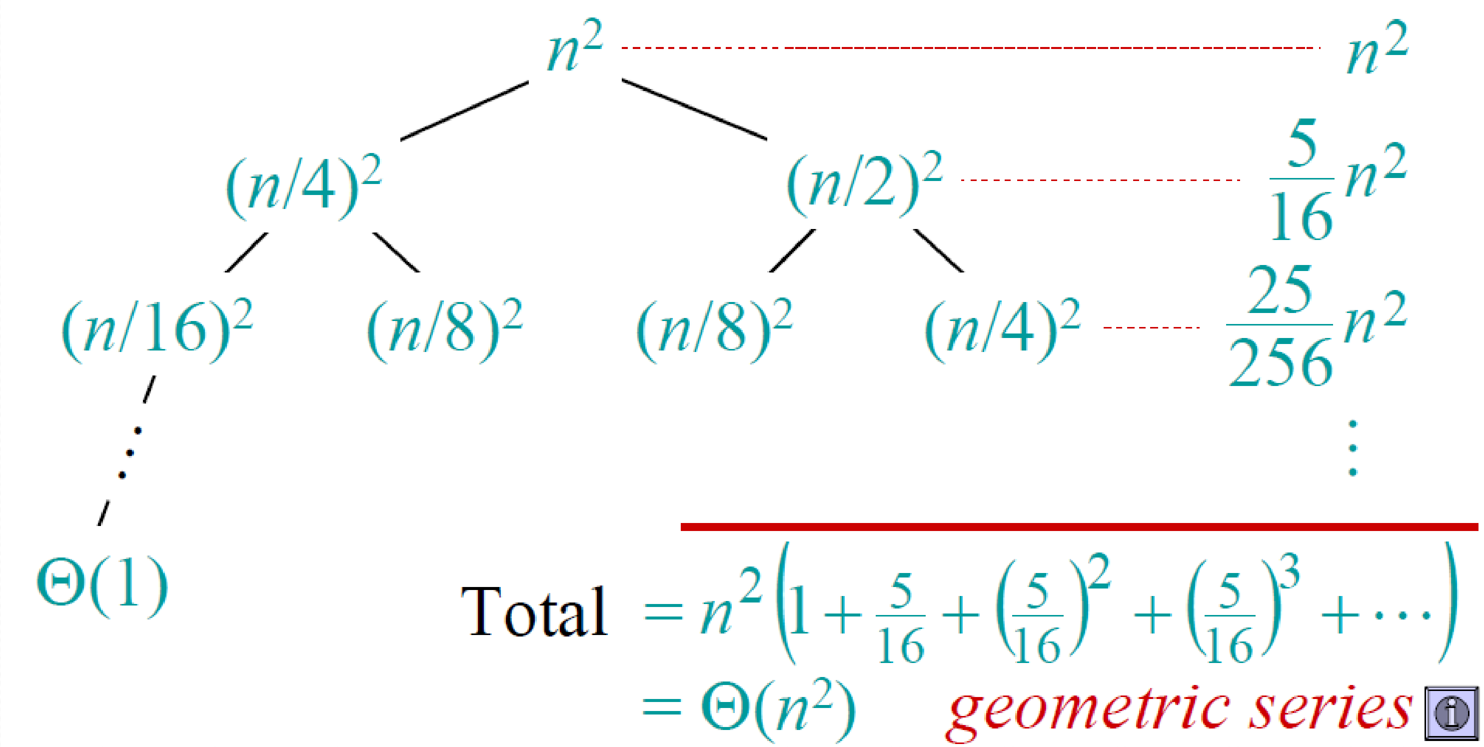
递归算法的数学分析

- 递归相关算法：
  - 反向替换法——找递推关系
  - 替换法——猜测后证明
  - 递归树方法——绘递归树，找关系并证明
  - 主方法Master method——套用3种公式

递归算法的数学分析

- 递归树方法。容易看出结果，需用替换法证明

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



递归算法的数学分析

- 递归相关算法：
  - 反向替换法——找递推关系
  - 替换法——猜测后证明
  - 递归树方法——绘递归树，找关系并证明
  - 主方法Master method——套用3种公式

递归算法的数学分析

- 主方法。Master method
  - 只能用在递归形式上
  - 符合$T(n) = aT(n/b) + f(n), a \geq 1$且$b > 1, f(n)$渐进趋正 $(n \geq n_0$时候$f(n) > 0)$
- 通过比较$f(n)$和$n^{\log_b a}$, 直接套用响应公式得到$T(n)$。其中$n^{\log_b a}$是递归树叶个数

递归算法的数学分析

- 主方法。Master method
  - 1、$f(n) \in O\left(n^{\log_b a-\varepsilon}\right)$ $\qquad \varepsilon > 0$

    $$T(n) = \Theta\left(n^{\log_b a}\right)$$
  - 2、$f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$ $\qquad k \geq 0$

    $$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$$
  - 3、$f(n) \in \Omega\left(n^{\log_b a+\varepsilon}\right)$ $\qquad \varepsilon > 0$

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

以上过程都要保证$af(n/b) \leq (1-\varepsilon')f(n)$ $\qquad (\varepsilon' > 0)$
即保证在递归过程中$f(n)$不断变小

递归算法的数学分析

- 主方法。Master method
  - 例子1 $\mathrm{T}(n) = 4\mathrm{T}(n/2) + n$
    - $a = 4, b = 2, f(n) = n$
    - $n^{\log_b a}$=n^2
    - 比较n与n^2，小于关系，套用公式1
    - $\mathrm{T}(n) = \Theta(n^2)$

1、$f(n) \in \mathrm{O}\left(n^{\log_b a-\varepsilon}\right)$ $\varepsilon > 0$

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

递归算法的数学分析

- 主方法。Master method
  - 例子2 $\mathrm{T}(n) = 4\mathrm{T}(n/2) + n^2$
    - $a = 4, b = 2, f(n) = n^2$
    - $n^{\log_b a}$=n^2
    - 比较n^2与n^2，等于关系，套用公式2，k=0
    - $\mathrm{T}(n) = \Theta(n^2 \log \mathrm{n})$

2、$f(n) \in \Theta\left(n^{\log_b a} \log^k n\right) \qquad k \geq 0$

$$\mathrm{T}(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$$

递归算法的数学分析

- 主方法。Master method
  - 例子3 $T(n) = 4T(n/2) + n^3$
    - $a = 4, b = 2, f(n) = n^3$
    - $n^{\log_b a}$=n^2
    - 比较n^3与n^2，大于关系，套用公式3
    - $T(n) = \Theta(n^3)$

$$3、f(n) \in \Omega\left(n^{\log_b a+\varepsilon}\right) \qquad \varepsilon > 0$$
$$T(n) = \Theta(f(n))$$

递归算法的数学分析

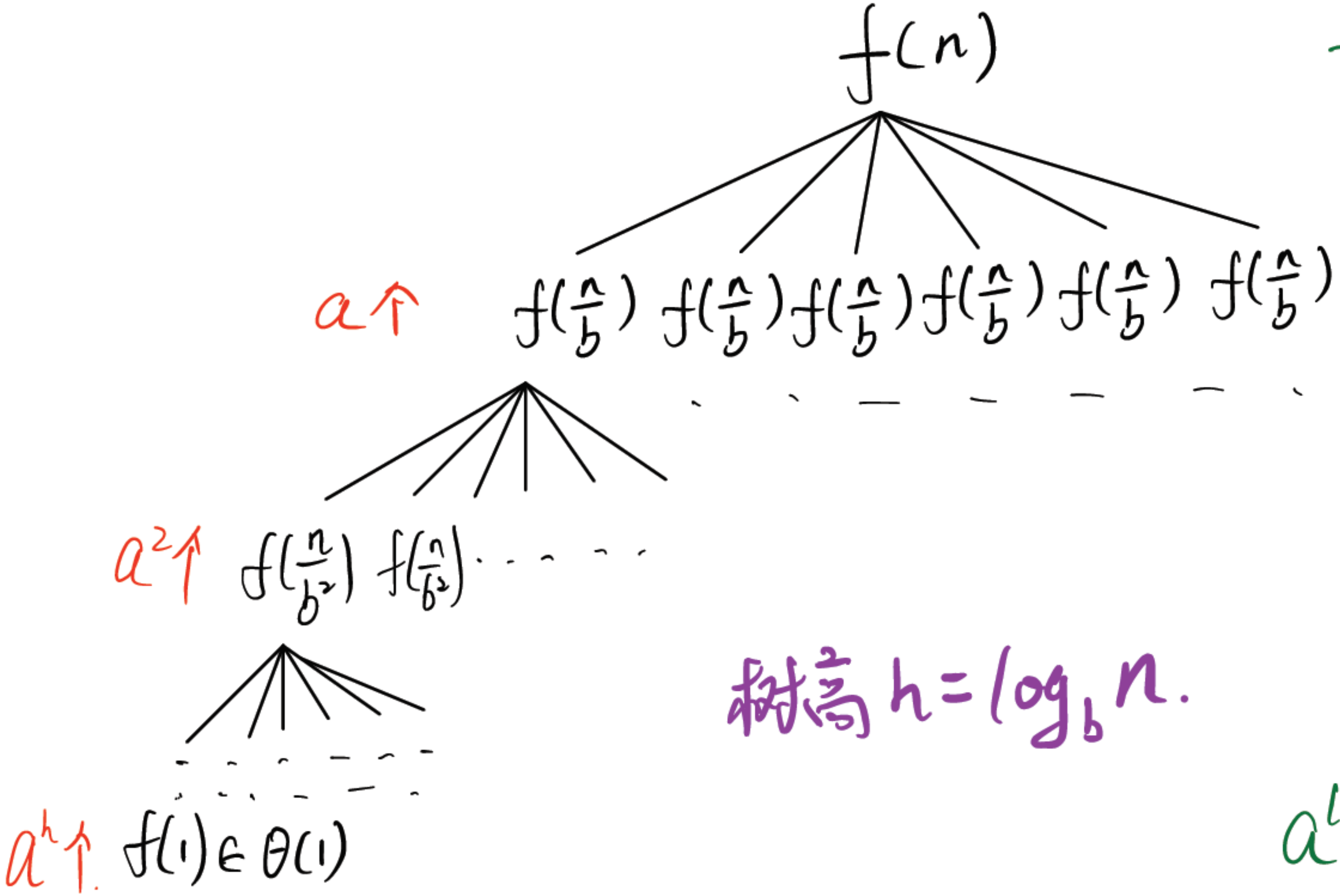
- 主方法。Master method
  - 例子4 $\mathrm{T}(n) = 4\mathrm{T}(n/2) + n^2/\log \mathrm{n}$
    - $a = 4, b = 2, f(n) = n^2/\log \mathrm{n}$
    - $n^{\log_b a}$=n^2
    - $n^2/\log \mathrm{n}$ 等于$n^2 \log^{-1} n$，k=-1<0不满足条件
    - 不能使用主方法

2、$f(n) \in \Theta\left(n^{\log_b a} \log^k n\right) \qquad k \geq 0$

$\mathrm{T}(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$

递归算法的数学分析

- 用递归树验证主方法。



$T(n) = f(n)$

$a f(\frac{n}{b})$

$a^2 f(\frac{n}{b^2})$

$\vdots$

树高 $h = \log_b n$.

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

递归算法的数学分析

- 用递归树验证主方法。
- 树的高度$h = \log_b n$
- 叶节点的个数是$a^h = a^{\log_b n} = n^{\log_b a}$
  - 若各层代价几何增长很快，(树上层代价$f(n)$小于下层代价$n^{\log_b a}$)，$n^{\log_b a}$占主导，采用公式1
  - 若各层代价逐渐减少，(树上层代价$f(n)$大于下层代价$n^{\log_b a}$)，$f(n)$占主导，采用公式3
  - 顶层与底层基本一样，每次大致相同，渐进相等。总代价是f乘以树高h， $T(n) = f(n)\log_b n = f(n)\Theta(\log n)$

递归算法的数学分析

- 递归算法效率分析的通用方案
  1. 决定输入规模度量参数，并确定<u>**基本操作**</u>。
  2. 检查相同规模的不同输入，<u>**基本操作执行次数是否可能不同**</u>，从而决定是否需要分别研究最差、最优（如有必要）和平均效率。
  3. 对基本操作执行次数<u>**建立递推关系及初始条件**</u>。
  4. <u>**解递推式**</u>，或者至少确定解的增长次数。