

算法理论

汪炆

中国科学技术大学软件学院

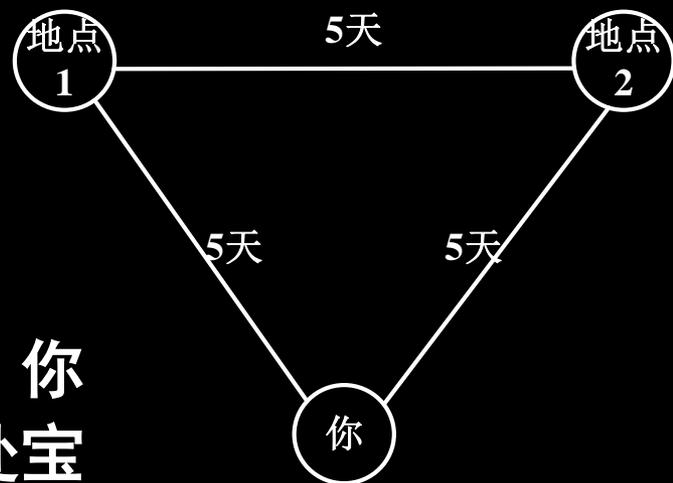
第一部分

概率算法

Ch.1 绪论

§ 1.1 引言

- 故事：**想象自己是神话故事的主人公，你有一张不易懂的地图，上面描述了一处宝藏的藏宝地点。经分析你能确定最有可能的两个地点是藏宝地点，但二者相距甚远。假设你如果已到达其中一处，就立即知道该处是否为藏宝地点。你到达两处之一地点，以及从其中一处到另一处的距离是5天的行程。进一步假设有一条恶龙，每晚光顾宝藏并从中拿走一部分财宝。假设你取宝藏的方案有两种：



§ 1.1 引言

方案1. 花4天的时间计算出准确的藏宝地点，然后出发寻宝，一旦出发不能重新计算

方案2. 有一个小精灵告诉你地图的秘密，但你必须付给他报酬，相当于龙3晚上拿走的财宝

Prob 1.1.1 若忽略可能的冒险和出发寻宝的代价，你是否接受小精灵的帮助？

显然，应该接受小精灵的帮助，因为你只需给出3晚上被盗窃的财宝量，否则你将失去4晚被盗财宝量。

但是，若冒险，你可能做得更好！

§ 1.1 引言

设 x 是你决定之前当日的宝藏价值，设 y 是恶龙每晚盗走的宝藏价值，并设 $x > 9y$

方案1： 4天计算确定地址，行程5天，你得到的宝藏价值为： $x - 9y$

方案2： $3y$ 付给精灵，行程5天失去 $5y$ ，你得到的宝藏价值为： $x - 8y$

方案3： 投硬币决定先到一处，失败后到另一处(冒险方案)

一次成功所得： $x - 5y$ ，机会 $1/2$
二次成功所得： $x - 10y$ ，机会 $1/2$ } 期望赢利： $x - 7.5y$

2. 意义

该故事告诉我们：当一个算法面临某种选择时，有时随机选择比耗时做最优选择更好，尤其是当最优选择所花的时间大于随机选择的平均时间的时候。

显然，概率算法只能是期望的时间更有效，但它有可能遭受到最坏的可能性。

3. 期望时间和平均时间的区别

- ❖ **确定算法的平均执行时间**

输入规模一定的所有输入实例是等概率出现时，算法的平均执行时间。

- ❖ **概率算法的期望执行时间**

反复解同一个输入实例所花的平均执行时间。

因此，对概率算法可以讨论如下两种期望时间

- ① **平均的期望时间**：所有输入实例上平均的期望执行时间
- ② **最坏的期望时间**：最坏的输入实例上的期望执行时间

4. 例子

① 快速排序中的随机划分

要求学生写一算法，由老师给出输入实例，按运行时间打分，大部分学生均不敢用简单的划分，运行时间在1500-2600ms，三个学生用概率的方法划分，运行时间平均为300ms。

② 8皇后问题

系统的方法放置皇后(回溯法)较合适，找出所有92个解 $O(2^n)$ ，若只找92个其中的任何一个解可在线性时间内完成 $O(n)$ 。

随机法：随机地放置若干皇后能够改进回溯法，特别是当n较大时

③ 判断大整数是否为素数

确定算法无法在可行的时间内判断一个数百位十进制数是否素数，否则密码就不安全。

概率算法将有所作为：若能接受一个任意小的错误的概率

5. 概率算法的特点

(1) 不可再现性

在同一个输入实例上，每次执行结果不尽相同，例如

① N-皇后问题

概率算法运行不同次将会找到不同的正确解

② 找一给定合数的非平凡因子

每次运行的结果不尽相同，但确定算法每次运行结果必定相同

(2) 分析困难

要求有概率论，统计学和数论的知识

6. 约定

随机函数uniform: 随机, 均匀, 独立

- ① 设 a, b 为实数, $a < b$,
uniform(a, b) 返回 x , $a \leq x < b$
- ② 设 i, j 为整数, $i \leq j$,
uniform($i..j$)= k , $i \leq k \leq j$
- ③ 设 X 是非空有限集,
uniform(X) $\in X$

例1: 设 p 是一个素数, a 是一个整数满足 $1 \leq a < p$, a 模除 p 的指数(index)是满足 $a^i \equiv 1 \pmod{p}$ 的最小正整数 i 。它等于集合 $X = \{a^j \pmod{p} \mid j \geq 1\}$ 的势, 即 $i = |X|$ 。

例如, 2模除31的指数等于5: $2^5 \pmod{31} = 1$,

$X = \{2^1 \pmod{31}, 2^2 \pmod{31}, 2^3 \pmod{31}, 2^4 \pmod{31}, 2^5 \pmod{31}\}$;

5模除31的指数是3, 即 $5^3 \pmod{31} = 1$,

3模除31的指数是30。

由费马(Fermat)定理($a^{p-1} \equiv 1 \pmod{p}$)可知, a 模 p 的指数总是恰好整除 $p-1$ 。

例如, 设 $p=31$, 若 $a=2$, 则 $30 \div 5=6$;

若 $a=5$, 则 $30 \div 3=10$ 。

因此, X 中的 j 至多为 $p-1$, 由此可得一种在 X 中随机, 均匀和独立地取一个元素的算法。

```
ModularExponent(a, j, p){  
  //求方幂模 $s=a^j \bmod p$ , 注意先求 $a^j$ 可能会溢出  
  s ← 1;  
  while j>0 do {  
    if (j is odd) s ← s·a mod p;  
    a ← a2 mod p;  
    j ← j div 2;  
  }  
  return s;  
}
```

```
Draw (a, p) {  
  // 在X中随机取一元素  
  j ← uniform(1..p-1);  
  return ModularExponent(a, j, p); // 在X中随机取一元素  
}
```

■ 伪随机数发生器

在实用中不可能有真正的随机数发生器，多数情况下是用伪随机数发生器代替。

大多数伪随机数发生器是基于一对函数：

S: $X \rightarrow X$, 这里 X 足够大，它是种子的值域

R: $X \rightarrow Y$, Y 是伪随机数函数的值域

使用s获得种子序列： $x_0=g$, $x_i=S(x_{i-1})$, $i>0$

然后使用R获得伪随机序列： $y_i=R(x_i)$, $i \geq 0$

该序列必然是周期性的，但只要S和R选得合适，该周期长度会非常长。

TC中可用rand()和srand(time), 用GNU C更好

§ 1.2 概率算法的分类

1. 基本特征

随机决策

在同一实例上执行两次其结果(或过程)可能不同

在同一实例上执行两次的时间亦可能不太相同

2. 分类

Numerical, Monte Carlo, Las Vegas, Sherwood.

很多人将所有概率算法(尤其是数字的概率算法)称为Monte Carlo算法

§ 1.2 概率算法的分类

① 数字算法

随机性被最早用于求数字问题的近似解

例如，求一个系统中队列的平均长度的问题，确定算法很难得到答案

- 概率算法获得的答案一般是近似的，但通常算法执行的时间越长，精度就越高，误差就越小
- 使用的理由
 - 现实世界中的问题在原理上可能就不存在精确解
例如，实验数据本身就是近似的，一个无理数在计算机中只能近似地表示
 - 精确解存在但无法在可行的时间内求得
有时答案是以置信区间的形式给出的

§ 1.2 概率算法的分类

② Monte Carlo算法 (MC算法)

蒙特卡洛算法1945年由J. Von Neumann进行核武模拟提出的。它是以概率和统计的理论与方法为基础的一种数值计算方法，它是双重近似：一是用概率模型模拟近似的数值计算，二是用伪随机数模拟真正的随机变量的样本。

这里我们指的MC算法是：**若问题只有1个正确的解，而无近似解的可能时使用MC算法**

例如，判定问题只有真或假两种可能性，没有近似解
因式分解，我们不能说某数几乎是一个因子

- **特点：**MC算法总是给出一个答案，但该答案未必正确，成功(即答案是正确的)的概率正比于算法执行的时间
- **缺点：**一般不能有效地确定算法的答案是否正确

§ 1.2 概率算法的分类

③ Las Vegas算法 (LV算法)

LV算法绝不返回错误的答案。

■ **特点：**获得的答案必定正确，但有时它仍根本就找不到答案。

和MC算法一样，成功的概率亦随算法执行时间增加而增加。无论输入何种实例，只要算法在该实例上运行足够的次数，则算法失败的概率就任意小。

④ Sherwood算法

Sherwood算法总是给出正确的答案。

当某些确定算法解决一个特殊问题平均的时间比最坏时间快得多时，我们可以使用Sherwood算法来减少，甚至是消除好的和坏的实例之间的差别。

Ch.2 数字概率算法

这类算法主要用于找到一个数字问题的近似解

§ 2.1 π 值计算

- 实验：将 n 根飞镖随机投向一正方形的靶子，计算落入此正方形的内切圆中的飞镖数目 k 。

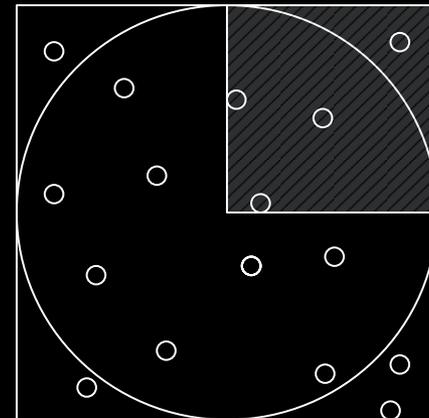
假定飞镖击中方形靶子任一点的概率相等(用计算机模拟比任一飞镖高手更能保证此假设成立)

设圆的半径为 r ，面积 $s_1 = \pi r^2$ ；方靶面积 $s_2 = 4r^2$

由等概率假设可知落入圆中的飞镖和正方形内的飞镖平均比为：

$$\frac{k}{n} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

由此知： $\pi \approx 4k / n$ $k = \frac{\pi n}{4}$



§ 2.1 π 值计算

■ 求 π 近似值的算法

为简单起见，只以上图的右上1/4象限为样本

```
Darts (n) {  
    k ← 0;  
    for i ← 1 to n do {  
        x ← uniform(0, 1);  
        y ← uniform(0, 1); // 随机产生点(x,y)  
        if (x2 + y2 ≤ 1) then k++; //圆内  
    }  
    return 4k/n;  
}
```

实验结果： $\pi=3.141592654$

n = 1000万: 3.140740, 3.142568 (2位精确)

n = 1亿: 3.141691, 3.141363 (3位精确)

n = 10亿: 3.141527, 3.141507 (4位精确)

§ 2.1 π 值计算

求 π 近似值的算法

Ex.1 若将 $y \leftarrow \text{uniform}(0, 1)$ 改为 $y \leftarrow x$, 则上述的算法估计的值是什么?

§ 2.2 数字积分 (计算定积分的值)

Monte Carlo积分(但不是指我们定义的MC算法)

1、概率算法1

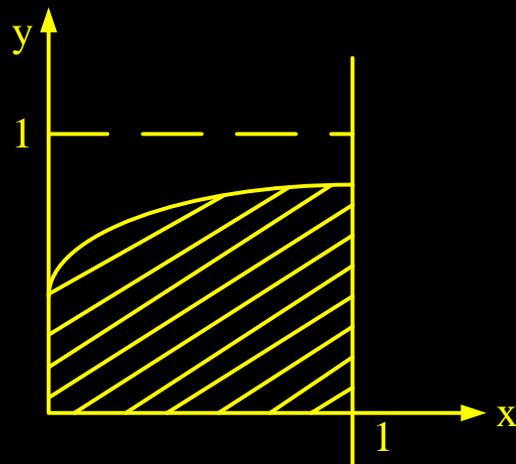
设 $f: [0, 1] \rightarrow [0, 1]$ 是一个连续函数，则由曲线 $y=f(x)$, x 轴, y 轴和直线 $x=1$ 围成的面积由下述积分给出：

$$S = \int_0^1 f(x) dx$$

向单位面积的正方形内投镖 n 次，落入阴影部分的镖的数目为 k ，则

$$\frac{k}{n} = \frac{S}{1} \Rightarrow S = k/n$$

显然，只要 n 足够大 $S \rightarrow k/n$



§ 2.2 数字积分 (计算定积分的值)

1. 概率算法1

HitorMiss (f, n) {

 k ← 0;

 for i ← 1 to n do {

 x ← uniform(0, 1);

 y ← uniform(0, 1);

 if y ≤ f(x) then k++;

 }

 return k/n;

}

Note: $\int_0^1 \sqrt{1-x^2} dx$ 是S/4的面积, $\because \pi = S, \therefore \pi = 4 \int_0^1 \sqrt{1-x^2} dx$

§ 2.2 数字积分 (计算定积分的值)

1. 概率算法1

Ex2. 在机器上用 $4 \int_0^1 \sqrt{1-x^2} dx$ 估计 π 值, 给出不同的 n 值及精度。

Ex3. 设 a, b, c 和 d 是实数, 且 $a \leq b, c \leq d, f:[a, b] \rightarrow [c, d]$ 是一个连续函数, 写一概率算法计算积分:

$$\int_a^b f(x)dx$$

注意, 函数的参数是 a, b, c, d, n 和 f , 其中 f 用函数指针实现, 请选一连续函数做实验, 并给出实验结果。

§ 2.2 数字积分 (计算定积分的值)

1. 概率算法1

*Ex4. 设 ε, δ 是 $(0,1)$ 之间的常数, 证明:

若 I 是 $\int_0^1 f(x)dx$ 的正确值, h 是由Hit or Miss算法返回的值, 则当 $n \geq I(1-I)/\varepsilon^2\delta$ 时有:

$$\text{Prob}[|h-I| < \varepsilon] \geq 1 - \delta$$

上述的意义告诉我们: $\text{Prob}[|h-I| \geq \varepsilon] \leq \delta$, 即: 当 $n \geq I(1-I)/\varepsilon^2\delta$ 时, 算法的计算结果的绝对误差超过 ε 的概率不超过 δ , 因此我们根据给定 ε 和 δ 可以确定算法迭代的次数

$$n = \frac{I(1-I)}{\varepsilon^2\delta} \leq \left\lceil \frac{1}{4\varepsilon^2\delta} \right\rceil \quad (\because I(1-I) \leq \frac{1}{4})$$

解此问题时可用切比雪夫不等式, 将 I 看作是数学期望。

§ 2.2 数字积分 (计算定积分的值)

2. 概率算法2

更有效的概率算法是：在积分区间上随机均匀地产生点，求出这些点上的函数值的算术平均值，再乘以区间的宽度：

$$\int_a^b f(x)dx = (b-a) \frac{1}{n} \sum_{i=1}^n f(x_i), \quad a \leq x_i \leq b$$

```
Crude (f, n, a, b) {  
    sum ← 0;  
    for i ← 1 to n do {  
        x ← uniform(a, b);  
        sum ← sum + f(x);  
    }  
    return (b-a)sum/n;  
}
```

§ 2.2 数字积分 (计算定积分的值)

2. 概率算法2

用HitorMiss和Crude运行三次的结果为：

$$n = 1\text{亿} \begin{cases} \text{hit} & 3.141855, 3.141422, 3.141434 \\ \text{crude} & 3.141662, 3.141486, 3.141527 \end{cases}$$

假定 $\int_a^b f(x)dx$ 和 $\int_a^b f^2(x)dx$ 存在，由算法求得的估算值的方差反比于点数 n 。当 n 足够大时，估计的分布近似为正态分布。

对于给定的迭代次数 n ，Crude算法的方差不会大于HitorMiss的方差。但不能说，Crude算法总是优于HitorMiss。因为后者在给定的时间内能迭代的次数更多。例如，计算 π 值时，Crude需计算平方根，而用投镖算法darts时无需计算平方根。

§ 2.2 数字积分 (计算定积分的值)

3. 确定的算法

梯形算法

将区间分为 $n-1$ 个子区间，每个子区间内的长度为 δ ，

$$\text{积分值} = \delta(f(a+\delta) + f(a+2\delta) + \dots + \frac{f(a) + f(b)}{2})$$

```
Trapezoid (f, n, a, b) {
```

```
  // 假设  $n \geq 2$ 
```

```
  delta  $\leftarrow$  (b-a)/(n-1);
```

```
  sum  $\leftarrow$  (f(a) + f(b))/2;
```

```
  for x  $\leftarrow$  a+delta step delta to b - delta do
```

```
    sum  $\leftarrow$  sum + f(x)
```

```
  return sum  $\times$  delta;
```

```
}
```

§ 2.2 数字积分 (计算定积分的值)

3. 确定的算法

当 $n=100$, $\pi=3.140399$

当 $n=1,000$, $\pi=3.141555$

当 $n=10,000$, $\pi=3.141586$

当 $n=100,000$, $\pi=3.141593$

一般地, 在同样的精度下, 梯形算法的迭代次数少于MC积分, 但是

① 有时确定型积分算法求不出解: 例如,

$$f(x)=\sin^2((100)! \pi x), \quad \int_0^1 f(x)dx = \frac{1}{2}。$$

但若用梯形算法, 当 $2 \leq n \leq 101$ 时, 返回值是0。若用MC积分则不会发生该类问题, 或虽然发生, 但概率小得多。

§ 2.2 数字积分 (计算定积分的值)

② 多重积分

在确定算法中，为了达到一定的精度，采样点的数目随着积分维数成指数增长，例如，一维积分若有100个点可达到一定的精度，则二维积分可能要计算 100^2 个点才能达到同样的精度，三维积分则需计算 100^3 个点。(系统的方法)

但概率算法对维数的敏感度不大，仅是每次迭代中计算的量稍增一点，实际上，MC积分特别适合用于计算4或更高维数的定积分。

若要提高精度，则可用混合技术：部分采用系统的方法，部分采用概率的方法

§ 2.3 概率计数

上一节可以认为，数字概率算法被用来近似一个实数，本节可用它们来估计一个整数值。例如，设 X 为有限集，若要求 X 的势 $|X|$ ，则当 X 较大时，枚举显然不现实。

1. 问题：随机选出25人，你是否愿意赌其中至少有两个人生日相同吗？直觉告诉我们，一般人都不愿意赌其成立，但实际上成立的概率大于50%。

§ 2.3 概率计数

一般地，从 n 个对象中选出 k 个互不相同的对象，若考虑选择的次序，则不同的选择有 $\frac{n!}{(n-k)!}$ 种；若允许重复选取同一对象，则不同的选法共有 n^k 种。

因此，从 n 个对象(允许同一对象重复取多次)中随机均匀地选择出的 k 个对象互不相同的概率是： $\frac{n!}{(n-k)!n^k}$ ，注意 a ， b 和 b ， a 是不同的取法。由此可知，上述问题中，25个人生日互不相同的概率是： $\frac{365!}{340!365^{25}}$ $n = 365, k = 25$

这里假设：不考虑闰年，一年中人的生日是均匀分布的。

§ 2.3 概率计数

■ 由Stirling公式知: $n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{\theta_n}{12n}}$, $0 < \theta_n < 1$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \theta(x^4) \text{ when } -1 < x < 1$$

可得
$$\frac{n!}{(n-k)!n^k} = e^{-k(k-1)/2n - k^3/6n^2 \pm O(\max(k^2/n^2, k^4/n^3))}$$

假定 $1 \ll k \ll n$ 近似地
$$\frac{n!}{(n-k)!n^k} \approx e^{-k^2/2n}$$

实际上, 若 $k \approx \alpha\sqrt{n}$, $\alpha = \sqrt{2\ln 2} \approx 1.177$, $k = 1.177\sqrt{365} \approx 22.5$

$$e^{-k^2/2n} = e^{-(2\ln 2)n/2n} = e^{-\ln 2} = 50\%$$

当 $n = 365, k = 25$ 时 $e^{25^2/730} \approx e^{-0.856} \approx 0.43$

取23个互不相同的人的概率为50%

§ 2.3 概率计数

因此，随机选出25个人中生日互不相同的概率约43%，由此可知至少有两人生日相同的概率约为57%。

此例提示：有回放抽样，大集合通过第一次重复来估计集合大小。

§ 2.3 概率计数

2. 求集合X的势

设X是具有n个元素的集合，我们有回放地随机，均匀和独立地从X中选取元素，设k是出现第1次重复之前所选出的元素数目，则当n足够大时，k的期望值趋近为 $\beta\sqrt{n}$ ，这里

$$\beta = \sqrt{\pi/2} \approx 1.253$$

利用此结论可以得出估计|X|的概率算法：

$$\beta\sqrt{n} = \sqrt{n\pi/2} = k \quad n = \frac{2k^2}{\pi}$$

2. 求集合X的势

```
SetCount (X) {  
    k ← 0; S ← Φ;  
    a ← uniform(X);  
    do {  
        k++;  
        S ← S ∪ {a}; a ← uniform(X);  
    } while (a ∉ S)  
    return 2k2/π  
}
```

注意：∵k的期望值是 $\sqrt{n\pi/2}$ ，∴上述算法n需足够大，且运行多次后才能确定 $n=|X|$ ，即取多次运行后的平均值才能是n。

该算法的时间和空间均为 $\theta(\sqrt{n})$ ，因为 $k = \theta(\sqrt{n})$

§ 2.3 概率计数

EX. 用上述算法，估计整数子集 $1\sim n$ 的大小，并分析 n 对估计值的影响。

§ 2.3 概率计数

3. 多重集合中不同对象数目的估计

假设磁带上记录有Shakespeare全集，如何统计其中使用了多少个不同的单词？为简单起见，同一词的复数，被动语态等可作为不同项。

设 N 是磁带上总的单词数， n 是其中不同词的数目

- ❖ **方法一**：对磁带上的单词进行外部排序，时间 $\theta(N \lg N)$,空间需求较大
- ❖ **方法二**：在内存中建一散列表，表中只存储首次出现的单词，平均时间 $O(N)$ ，空间 $\Omega(n)$

§ 2.3 概率计数

3. 多重集合中不同对象数目的估计

- ❖ 方法三：若能忍受某种误差及已知 n 或 N 的上界 M ，则存在一个时空性能更好的概率算法解此问题。

设 U 是单词序列的集合，设参数 m 稍大于 $\lg M$ ，可令：

$$m = 5 + \lceil \lg M \rceil$$

设 $h: U \rightarrow \{0, 1\}^m$ 是一个散列函数，它用伪随机数的方法将 U 中的单词映射为长度为 m 的位串。(目的，减少存储量)

若 y 是一个长度为 k 的位串，用 $y[i]$ 表示 y 的第 i 位， $1 \leq i \leq k$ ；

用 $\pi(y, b)$, $b \in \{0, 1\}$ 来表示满足 $y[i]=b$ 的最小的 i ，若 y 的位中没有哪一位等于 b ，则 $\pi = k+1$

3. 多重集合中不同对象数目的估计

$$\text{即: } \pi(y, b) = \begin{cases} i & \text{使得 } y[i] = b \text{ 的最小 } i, & 1 \leq i \leq k; \\ k+1 & y[i] \neq b, & 1 \leq i \leq k; \end{cases} \quad b = 0 \text{ or } 1$$

WordCount () {

y[1..m+1] ← 0; // 初始化

for 磁带上每个单词x do { //顺序读磁带

i ← π(h(x), 1); // x的散列值中等于1的最小位置, 表示x是
//以 $\underbrace{00\dots 01}_{i\text{位}}$ 打头的

y[i] ← 1; // 将该位置置为1

}

return π(y, 0); // 返回y中等于0的最小位置

}

§ 2.3 概率计数

3. 多重集合中不同对象数目的估计

❖ 上界估计

例，不妨设 $m=4$ ， $h(x_1)=0011$ ， $h(x_2)=1010$ ， $h(x_3)=0110$ ， $h(x_4)=0010$ ，

	1	2	3	4	5	
y	1	1	1	0	0	算法返回： k=4
	x_2	x_3	x_1			
			x_4			

也就是说，若算法返回4，说明磁带上至少有3个单词的散列地址是以1，01，001打头的，但绝没有以0001打头的单词。

∴一个以0001开始的随机二进制串的概率是 $2^{-4}=1/16$

∴磁带上不太可能有多于16个互不相同的单词，即：**互异单词的上界 2^k**

因为只要 h 的随机性好，则对16个不同的单词 x_i ， $\pi(h(x_i), 1) \neq 4$ (**这些单词的散列值等于1的最小位置均不为4**)的概率是 $(15/16)^{16} \approx 35.6\% \approx e^{-1}$ (**每个 x_i 不等于0001的概率的15/16，16个单词均不以0001开头的概率为35.6%**)，只有此时算法才可能返回4。

§ 2.3 概率计数

3. 多重集合中不同对象数目的估计

实际上，若算法的返回值 k 为4，则 $n=16$ 的概率为：

$$\text{Prob}[k=4 \mid n=16] = 31.75\%$$

❖ 下界估计

∵一个以001开始的随机二进制串的概率是 2^{-3}

∴在磁带上互不相同的单词数目少于4的可能性不大，即：**互不相同单词的下界 2^{k-2}**

因为，对4个互不相同的单词中，至少有一个是以001打头的概率为 $1 - (7/8)^4 \approx 41.4\%$ 。实际上，若算法的返回值 k 为4，则 $n=4$ 的概率为：

$$\text{Prob}[k=4 \mid n=4] = 18.75\%$$

粗略的分析告诉我们：

磁带上互不相同的单词数目为： $2^{k-2} \sim 2^k$

实际上，算法WordCount估计的 n 应等于 $2^k/1.54703$

❖ 性能：时间 $O(N)$ ，空间： $O(\lg N)$

§ 2.4 线性代数中的数字问题

例如，矩阵乘法，求逆，计算特征值和特征向量
只有一些特殊的应用，概率算法会执行得比确定性
算法要好。

Ch.3 Sherwood算法

Sherwood算法能够平滑不同输入实例的执行时间

- 设A是一个**确定算法**， $t_A(x)$ 是解某个实例x的执行时间，设n是一整数， X_n 是大小为n的实例的集合

假定 X_n 中每一个实例是等可能出现的，则算法A解一个大小为n的实例的平均执行时间是：

$$\bar{t}_A(n) = \sum_{x \in X_n} t_A(x) / |X_n|$$

这里无法消除这样的可能性，存在一个size为n的实例x使得：

$$t_A(x) \gg \bar{t}_A(n)$$

- 设B是一个**概率算法**，对每个size为n的实例x满足：

$$t_B(x) \approx \bar{t}_A(n) + s(n)$$

这里 $t_B(x)$ 是算法B在实例x上的期望值， $s(n)$ 是概率算法B为了取得均匀性所付出的成本。

Ch.3 Sherwood算法

虽然算法B的执行时间也可能偶然地在某一个实例x上大于 $\bar{t}_A(n) + s(n)$ ，但这种偶然性行为只是由于算法所做的概率选择引起的，与实例x本身没有关系。因此，不再有最坏情况的实例，但有最坏的执行时间。

算法B在一个size为n的实例集上的平均期望时间可定义为：
$$\bar{t}_B(n) = \sum_{x \in X_n} t_B(x) / |X_n|$$

很清楚 $\bar{t}_B(n) \approx \bar{t}_A(n) + s(n)$

也就是说Sherwood算法的平均执行时间略为增加。

§ 3.1 选择与排序

在 n 个元素中选择第 k 个最小元素的算法关键在于选择划分元，有两种常用的方法：

- ① 精心挑选划分元，使之是一个伪中值的元素，这样可使算法的最坏执行时间是 $O(n)$
- ② 取当前搜索区间的第一个元素作划分元，平均时间为 $O(n)$ ，但最坏时间是 $O(n^2)$ 。由于此方法简单，故平均性能较前者好。

该类确定算法的特点：设 $T[1..n]$ 互不相同，算法的执行时间不是依赖于数组元素的值，而是依赖于元素间的相对次序，因此，表达时间的函数不只是依赖于 n ，而且还依赖于数组元素的排列 δ

设 $t_p(n, \delta)$ —— 使用伪中值算法的执行时间

$t_s(n, \delta)$ —— 使用简单算法的执行时间

对多数的 δ ， $\forall n$ ，有 $t_s(n, \delta) < t_p(n, \delta)$

但对有的 δ ， $t_s(n, \delta) > t_p(n, \delta)$

§ 3.1 选择与排序

更精确地，设 S_n 是 T 中前 n 个数的排列的集合， $|S_n|=n!$ ，

定义 $\bar{t}_s(n) = \sum_{\delta \in S_n} t_s(n, \delta) / n!$ ，于是有：

$$(\exists C_p)(\exists n_1 \in N)(\forall n \geq n_1)(\forall \delta \in S_n)[t_p(n, \delta) \leq C_p n]$$

// 伪中值算法最坏时间是线性的

$$(\exists C_s \ll C_p)(\exists n_2 \in N)(\forall n \geq n_2)[\bar{t}_s(n) \leq C_s n]$$

// $\bar{t}_s(n)$ 小于 $t_p(n, \delta)$ ，简单算法的平均时间远小于 $t_p(n, \delta)$

但是：

$$(\exists C'_s)(\exists n_3 \in N)(\forall n \geq n_3)(\exists \delta \in S_n)[t_s(n, \delta) \geq C'_s n^2 \gg C_p n \geq t_p(n, \delta)]$$

// 存在某个实例，或说实例的某个排列使得简单算法的执行时间远远大于 t_p

§ 3.1 选择与排序

- 概率算法

随机选择T中的元素作为划分元

期望时间为 $O(n)$ ，独立于输入实例

注意：算法的某次执行有可能达到 $O(n^2)$ ，但这种可能性与实例无关

随着 n 的增大，这种可能性会很小。

设 $t_r(n, \delta)$ 是Sherwood算法的平均时间，则

$$(\exists n_0 \in N)(\forall n \geq n_0)(\forall \delta \in S_n)[t_r(n, \delta) < t_p(n, \delta)]$$

§ 3.2 随机的预处理

将选择和排序的确定算法修改为Sherwood算法很简单，但是当算法较复杂，例如它是一个缺乏文档资料的软件包的一部分时，就很难对其进行修改。注意，只有当该算法平均时间性能较优，但最坏性能较差时，才有修改的价值。

一般方法是：

- ① 将被解的实例变换到一个随机实例。// 预处理
- ② 用确定算法解此随机实例，得到一个解。
- ③ 将此解变换为对原实例的解。// 后处理

§ 3.2 随机的预处理

设： $f: X \rightarrow Y$ 是解某问题用到的一个函数，且平均性能较优(指相应的算法)；

$\forall n \in \mathbb{N}$ ， X_n 是size为n的实例的集合

A_n 是一个大小和 X_n 大小相同的集合，

假定在 A_n 中能够有效地均匀随机抽样

$$A = \cup A_n$$

则随机的预处理由一对函数构成：

$$u: X \times A \rightarrow Y$$

$$v: A \times Y \rightarrow X$$

u 和 v 满足三个性质：

① $(\forall n \in \mathbb{N})(\forall x, y \in X_n)(\exists! r \in A_n)[u(x, r) = y]$

此性质说明原实例 x 可通过随机抽样变换成另一个实例 y ，

$\exists!$ 表示存在且只存在一个

② $(\forall n \in \mathbb{N})(\forall x \in X_n)(\forall r \in A_n)[f(x) = v(r, f(u(x, r)))]$

此性质表示对 y 的解可变换为对原实例 x 的解

③ 函数 u 和 v 在最坏情况下能够有效计算

§ 3.2 随机的预处理

于是确定算法 $f(x)$ 可改造为Sherwood算法：

$RH(x)$ {

 // 用Sherwood算法计算 $f(x)$

$n \leftarrow \text{length}[x]$; // x 的size为 n

$r \leftarrow \text{uniform}(A_n)$; // 随机取一元素

$y \leftarrow u(x, r)$; // 将原实例 x 转化为随机实例 y

$s \leftarrow f(y)$; // 用确定算法求 y 的解 s

 return $v(r, s)$; // 将 s 的解变换为 x 的解

}

§ 3.2 随机的预处理

例1：选择和排序的Sherwood算法

- 只需进行随机预处理

将输入实例中元素打乱即可，相当于洗牌

后处理无需进行

只需调用确定的算法前先调用下述算法：

Shuffle (T) {

$n \leftarrow \text{length}[T];$

 for $i \leftarrow 1$ to $n-1$ do {

 // 在 $T[i..n]$ 中随机选1元素放在 $T[i]$ 上

$j \leftarrow \text{uniform}(i..n);$

$T[i] \leftrightarrow T[j];$

 }

}

例2：离散对数计算

- 离散对数计算困难使其可用于密码算法，数字签名等
- 定义：设 $a=g^x \bmod p$ ，记 $\log_{g,p}a=x$ ，称 x 为 a 的(以 g 为底模除 p)对数。从 p, g, a 计算 x 称为**离散对数**问题。
- 简单算法
 - ① 计算 g^x 对所有的 x ，最多计算 $0 \leq x \leq p-1$ 或 $1 \leq x \leq p$ ，因为实际上离散对数 $\langle g \rangle$ 是循环群；
 - ② 验证 $a=g^x \bmod p$ 是否成立。

`dlog(g, a, p) { // 当这样的对数不存在时，算法返回p`

`x ← 0; y ← 1;`

`do { x++;`

`y ← y*g; // 计算y=gx`

`} while (a ≠ y mod p) and (x ≠ p);`

`return x`

`}`

例 $\log_{2,7} 3$ 无解，不存在
 x 使 $3 = 2^x \bmod 7$

例2：离散对数计算

- 问题：最坏 $O(p)$

循环次数难以预料，当满足一定条件时平均循环 $p/2$ 次

当 $p=24$ 位十进制数，循环 10^{24} 次，千万亿次/秒 (10^{16} 次/秒) 大约算1年(10^8 秒/年)

若 p 是数百位十进制？随机选择都可能无法在可行的时间内求解。

- 假设有一个平均时间性能很好，但最坏情况差的确定算法求 $\log_{g,p}a$ ，怎样用Sherwood算法求解该问题？

设 $p=19, g=2$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$a=g^i \bmod 19$	2	4	8	16	13	17	14	9	18	17	15	11	3	6	12	5	10	1	11

当 $a=14, 6$ 时， $\log_{2,19}14 = 7, \log_{2,19}6=14$ ，即用dlog求14和6的离散对数时，分别要循环7和14次，执行时间与 a 的取值相关。

用sherwood算法应该使得与 a 无关，用随机预处理的方法计算 $\log_{g,p}a$

§ 3.2 随机的预处理

- 定理(见p877, § 31.6)

① $\log_{g,p}(st \bmod p) = (\log_{g,p} s + \log_{g,p} t) \bmod (p-1)$

② $\log_{g,p}(g^r \bmod p) = r$ for $0 \leq r \leq p-2$

dlogRH(g, a, p) { // 求 $\log_{g,p} a$, $a = g^x \bmod p$, 求x

// Sherwood算法

r ← uniform(0..p-2);

b ← ModularExponent(g, r, p); //求幂模 $b = g^r \bmod p$

c ← ba mod p; //(($g^r \bmod p$)($g^x \bmod p$)) mod p = $g^{r+x} \bmod p = c$

y ← $\log_{g,p} c$; // 使用确定性算法求 $\log_{p,g} c$, $y = r + x$

return (y-r) mod (p-1); // 求x

}

Ex. 分析dlogRH的工作原理，指出该算法相应的u和v

§ 3.2 随机的预处理

- 随机预处理提供了一种加密计算的可能性

假定你想计算某个实例 x ，通过 $f(x)$ 计算，但你缺乏计算能力或是缺乏有效算法，而别人有相应的计算能力，愿意为你计算(可能会收费)，若你愿意别人帮你计算，但你不愿意泄露你的输入实例 x ，你将如何做？

可将随机预处理使用到 f 的计算上：

- ① 使用函数 u 将 x 加密为某一随机实例 y
- ② 将 y 提交给 f 计算出 $f(y)$ 的值
- ③ 使用函数 v 转换为 $f(x)$

上述过程，他人除了知道你的实例大小外，不知道 x 的任何信息，因为 $u(x,r)$ 的概率分布(只要 r 是随机均匀选择的)是独立于 x 的。

§ 3.3 搜索有序表

设两个数组 $val[1..n]$ 和 $ptr[1..n]$ 及 $head$ 构成一个有序的静态链表:

$$val[head] \leq val[ptr[head]] \leq val[ptr[ptr[head]]] \leq \dots \leq val[ptr^{n-1}[head]]$$

即: $ptr[i] \begin{cases} \text{给出下一个关键字的下标} & \text{if } val[i] \text{非最大关键字} \\ 0 & \text{if } val[i] \text{是最大关键字} \end{cases}$

例:

i	1	2	3	4	5	6	7
val[i]	2	3	13	1	5	21	8
ptr[i]	2	5	6	1	7	0	3
rank	2	3	6	1	4	7	5

head=4 有序表: 1, 2, 3, 5, 8, 13, 21

§ 3.3 搜索有序表

- **折半查找：** 若 $val[1..n]$ 本身有序，可用折半查找找某个给定的key，时间为 $O(\lg n)$ 。
- **顺序查找：** 但此表为链式结构，故最坏时间是 $\Omega(n)$ 。尽管如此，我们能够找到一个确定性算法，平均时间为 $O(\sqrt{n})$ 。

相应的Sherwood算法的期望时间是 $O(\sqrt{n})$ ，它虽然并不比确定性算法快，但他消除了最坏实例。

假定表中元素互不相同，且所求的关键字在表中存在，则给定 x ，我们是求下标 i ，使 $val[i]=x$ ，这里 $1 \leq i \leq n$ 。

任何实例可以由两个参数刻画：

- ①前 n 个整数的排列 δ
- ② x 的rank

§ 3.3 搜索有序表

设 S_n 是所有 $n!$ 个排列的集合，设A是一个确定性算法

(1) $t_A(n, k, \delta)$ 表示算法A的执行时间，此时间与被查找元素的秩 k ，以及 val 的排列 δ 相关。若A是一个概率算法，则 $t_A(n, k, \delta)$ 表示算法的期望值。无论算法是确定的还是概率的， $w_A(n)$ 和 $m_A(n)$ 分别表示最坏时间和平均时间，因此有：

$$(2) w_A(n) = \max \{ t(n, k, \delta) \mid 1 \leq k \leq n \text{ and } \delta \in S_n \}$$

$$(3) m_A(n) = \frac{1}{n \times n!} \sum_{k=1}^n \sum_{\delta \in S_n} t_A(n, k, \delta)$$

$k = 1, 2, \dots, n$ 的概率是 $1/n$ ，在 S_n 中每个排列的概率是 $\frac{1}{n!}$

§ 3.3 搜索有序表

1. 时间为 $O(n)$ 的确定算法

■ 算法

设 $x \geq \text{val}[i]$ 且 x 在表中，则从位置 i 开始查找 x 的算法为

```
Search(x, i) { //仍可改进
    while x > val[i] do
        i ← ptr[i];
    return i;
}
```

在表 $\text{val}[1..n]$ 中查找 x 的算法为：

```
A(x) {
    return Search(x, head);
}
```

§ 3.3 搜索有序表

■ 性能分析

设 $\text{rank}(x)=k$, 则:

$\hat{t}_A(n, k)$ —— 算法A在 n 个元素的表中查找 x 所需的访问数组元素的次数, 显然与 δ 无关

$w_A(n)$ —— 算法A最坏时的访问次数

$m_A(n)$ —— 算法A平均的访问次数

$$\textcircled{1} \quad \forall n \in N, \forall k \in [1, n], \hat{t}_A(n, k) = k$$

$$\textcircled{2} \quad \forall n \in N, \text{若 } k = n \text{ 时为最坏情况, 此时 } w_A(n) = n$$

$\textcircled{3} \quad \forall n \in N, \text{设 } k = 1, 2, \dots, n \text{ 的概率相等, 则}$

$$m_A(n) = \frac{1}{n} \sum_{k=1}^n \hat{t}_A(n, k) = \frac{n+1}{2}$$

综上所述, $T(n) = O(n)$

§ 3.3 搜索有序表

2. 时间为 $O(n)$ 的概率算法

■ 算法

```
D(x) {  
    i ← uniform(1..n);  
    y ← val[i];  
    case {  
        x < y: return Search(x, head); // case1  
        x > y: return Search(x, ptr[i]); // case2  
        otherwise: return i; // case3, x = y  
    }  
}
```

§ 3.3 搜索有序表

■ 性能分析(D访问数组次数)

① 一般情况

设 $\text{rank}(x)=k$, $\text{rank}(\text{val}[i])=j$

若 $k < j$, 则 $\hat{t}_D(n, k) = k$, 属于case1, 从头搜索

若 $k > j$, 则 $\hat{t}_D(n, k) = k - j$, 属于case2, 从 j^{th} 最小元之后搜索

若 $k = j$, 则 $\hat{t}_D(n, k) = 0$, 属于case3

② 最坏情况

$\forall n \in \mathbb{N}, w_D(n)?$

当 $j=1, k=n$ 时, Search执行次数为 $n-1$, $w_D(n) = n - 1$

§ 3.3 搜索有序表

③平均情况

$\forall n \in N, m_D(n)?$

$j = 1, 2, \dots, n$ 及 $k = 1, 2, \dots, n$ 的概率均为 $\frac{1}{n}$

$$\begin{aligned} m_D(n) &= \frac{1}{n \times n} \sum_{j=1}^n \sum_{k=1}^n \hat{t}_D(n, k) = \frac{1}{n^2} \sum_{j=1}^n \left(\sum_{k=1}^{j-1} k + \sum_{k=j+1}^n (k-j) \right) \\ &= \frac{1}{n^2} \sum_{j=1}^n \left(\frac{j(j-1)}{2} + \frac{(n-j+1)(n-j)}{2} \right) = \frac{1}{3}n - \frac{1}{3n} \approx \frac{n}{3} \end{aligned}$$

显然平均时间性能优于确定算法

§ 3.3 搜索有序表

3. 平均时间为 $O(\sqrt{n})$ 的确定算法

■ 算法

B(x) { //设x在val[1..n]中

 i ← head;

 max ← val[i]; // max初值是表val中最小值

 for j ← 1 to $\lfloor \sqrt{n} \rfloor$ do { // 在val的前 $\lfloor \sqrt{n} \rfloor$ 个数中找不大于x

 y ← val[j]; // 的最大整数y相应的下标i

 if max < y ≤ x then {

 i ← j;

 max ← y;

 } //endif

 } // endfor

 return Search(x, i); // 从y开始继续搜索

}

§ 3.3 搜索有序表

■ 性能分析

for循环的目的：找不超过x的最大整数y，使搜索从y开始，若将val[1..n]中的n个整数看作是均匀随机分布的，则在val[1..n]中求y值就相当于在n个整数中，随机地取l个整数，求这l个整数中不大于x的最大整数y。

可用一个与l和n相关的随机变量来分析，更简单的分析如下：

设n个整数的排列满足： $a_1 < a_2 < \dots < a_n$

将其等分为l个区间：

$$\underbrace{[a_1, \dots, a_{\frac{n}{l}}] [a_{\frac{n}{l}+1}, \dots, a_{\frac{2n}{l}}] \dots [a_{\frac{(l-1)n}{l}+1}, \dots, a_n]}_{l \text{ 个区间}}$$

§ 3.3 搜索有序表

若均匀随机地从上述表中取 l 个数，则平均每个区间中被选到1个元素（注意：因为 val 的随机均匀性，这里所取的 l 个数相当于 $val[1.. l]$ 中的 l 个数），因此无论 x 是处在哪一个区间，其平均的执行时间为：

i) 若在 x 的同一区间中取到的数小于等于 x ，则它是算法中的 y ，那么Search的比较次数不超过区间长度 n/l 。

ii) 若在 x 的同一区间中取到的数大于 x ，则在 x 的前一区间中的取到的数必为算法中的 y ，它必定小于 x ，且 x 和 y 的距离平均为 n/l ，此时Search的比较次数平均为 n/l 。

§ 3.3 搜索有序表

注意，在Search前需执行次循环，故有

$$m_B(n) = l + \frac{n}{l}$$

$$\text{因为} \left(l + \frac{n}{l}\right)' = 1 - \frac{n}{l^2}, \left(l + \frac{n}{l}\right)'' \geq 0$$

$$\therefore 1 - \frac{n}{l^2} = 0, \text{即} l = \sqrt{n} \text{时} m_B(n) \text{最小, 其值为} 2\sqrt{n}$$

因此，确定性算法中for的次数为 \sqrt{n} ，此时算法的平均时间 $2\sqrt{n}$ 最小。

Ex. 写一Sherwood算法C，与算法A, B, D比较，给出实验结果。

Ch.4 Las Vegas 算法

■ Las Vegas和Sherwood算法比较

- { Sherwood算法一般并不比相应的确定算法的平均性能优
- { Las Vegas一般能获得更有效率的算法，有时甚至是对每个实例皆如此
- { Sherwood算法可以计算出一个给定实例的执行时间上界
- { Las Vegas算法的时间上界可能不存在，即使对每个较小实例的期望时间，以及对特别耗时的实例的概率较小可忽略不计。

■ Las Vegas 特点

可能不时地要冒着找不到解的风险，算法要么返回正确的解，要么随机决策导致一个僵局。

若算法陷入僵局，则使用同一实例运行同一算法，有独立的机会求出解。

成功的概率随着执行时间的增加而增加。

Ch.4 Las Vegas 算法

■ 算法的一般形式

$LV(x, y, \text{success})$ —— x 是输入的实例， y 是返回的参数， success 是布尔值， true 表示成功， false 表示失败

$p(x)$ —— 对于实例 x ，算法成功的概率

$s(x)$ —— 算法成功时的期望时间

$e(x)$ —— 算法失败时的期望时间

一个正确的算法，要求对每个实例 x ， $p(x) > 0$ ，更好的情况是：

$$\exists \text{常数 } \delta > 0, p(x) \geq \delta$$

Ch.4 Las Vegas 算法

```
Obstinate(x) {  
    repeat  
        LV(x, y, success);  
    until success;  
    return y;  
}
```

设 $t(x)$ 是算法obstinate找到一个正确解的期望时间，则

$$t(x) = p(x)s(x) + (1 - p(x))(e(x) + t(x))$$

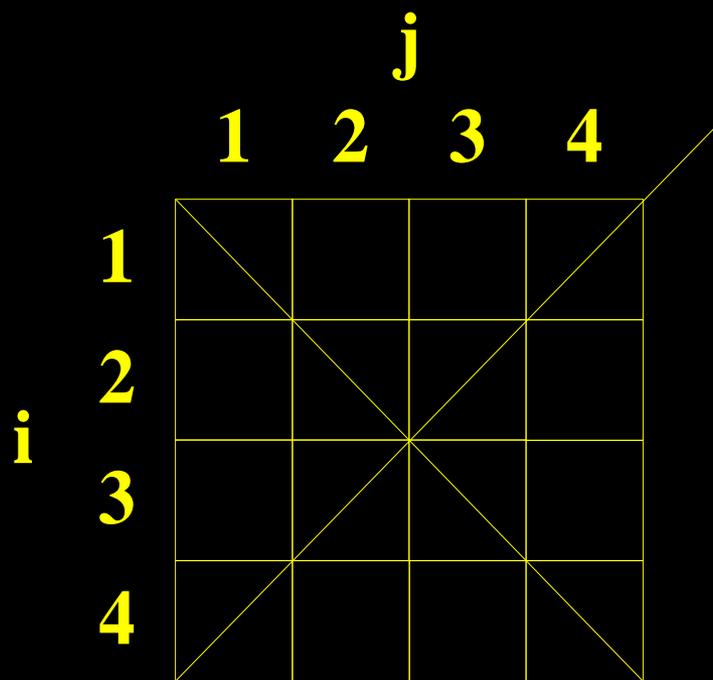
LV成功的概率 LV失败的概率

$$t(x) = s(x) + \frac{1 - p(x)}{p(x)} e(x)$$

若要最小化 $t(x)$ ，则需在 $p(x)$ ， $s(x)$ 和 $e(x)$ 之间进行某种折衷，例如，若要减少失败的时间，则可降低成功的概率 $p(x)$ 。 70

§ 4.1 8后问题

1. 传统的回溯法



$i+j$ 135度对角线: 同一条对角线上的元素的行列号之和相等

$i-j$ 或 $j-i$ 45度对角线: 同一条对角线上的元素的行列号之差相等

§ 4.1 8后问题

置当前行为第1行，当前列为第1列，即 $i \leftarrow j \leftarrow 1$;

while $i \leq 8$ do { // 当前行号 $i \leq 8$

 检查当前行 i : 从当前列 j 起向后逐列试探，寻找安全列号;

 if 找到安全列号 then {

 放置皇后，将列号记入栈，并将下一行置为当前行($i++$);

$j \leftarrow 1$; // 当前列置为1

 } else {

 退栈回溯到上一行，即 $i--$;

 移去该行已放置的皇后，以该皇后所在列的下一列作为当前列;

 }

}

§ 4.1 8后问题

2.Las Vegas方法

- ❖ 向量try[1..8]中存放结果

try[i]——表示第i个皇后放在 (i, try[i])位置上

- ❖ try[1..k]称为k-promising是指：

若k个皇后的位置($0 \leq k \leq 8$): (1,try[1]), (2,try[2]), ..., (k,try[k])互相不攻击, 则称try[1..k]是k-promising的.

形式化: 对 $\forall i, j \in [1, k]$, 若 $i \neq j$ 有

$$\text{try}[i] - \text{try}[j] \notin \{i - j, 0, j - i\} \quad (\text{式1})$$

若式1成立, 则:

- ① **无行冲突**: 无须考虑, 因为第i个皇后放在第i行, 故同一行不会有两皇后

§ 4.1 8后问题

② **无列冲突**：若对任意不同的两行*i*、*j*，因为其列数之差不为0，故任意两皇后不可能在同一列上。

③ **135° 对角线无冲突**： $a_{i,try[i]}$ 和 $a_{j,try[j]}$ 冲突时有：

$$i + try[i] = j + try[j] \quad \text{即} \quad try[i] - try[j] = j - i$$

故任两皇后不会在135° 对角线上冲突。

④ **45° 对角线无冲突**：

$$a_{i,try[i]} \text{ 和 } a_{j,try[j]} \text{ 冲突时有： } i - try[i] = j - try[j]$$

$$\text{即 } try[i] - try[j] = i - j$$

故任两皇后不会在45° 对角线上冲突。

综上所述，式1成立时 $try[1..k]$ 是k-promising。

显然，若 $k \leq 1$ ，则向量 $try[1..k]$ 是k-promising的，对8后问题，解是8-promising的。

❖ 算法

```

QueensLv (success){ //贪心的LV算法，所有皇后都是随机放置
    //若Success=true，则try[1..8]包含8后问题的一个解。
    col,diag45,diag135←Φ; //列及两对角线集合初值为空
    k ←0; //行号
    repeat //try[1..k]是k-promising，考虑放第k+1个皇后
        nb ←0; //计数器，nb值为 (k+1)th皇后的open位置总数
        for i ←1 to 8 do { //i是列号，试探 (k+1,i) 安全否?
            if (i∉col) and (i-k-1∉diag45) and (i+k+1∉diag135) then{
                //列i对 (k+1) th皇后可用，但不一定马上将其放在第i列
                nb ←nb+1;
                if uniform(1..nb)=1 then //或许放在第i列
                    j ←i; //注意第一次uniform一定返回1，即j一定有值i
                }//endif
            }
        }//endfor, 在nb个安全的位置上随机选择1个位置j放置之
    }
}

```

§ 4.1 8后问题

```
if(nb > 0) then{ //nb=0时无安全位置，第k+1个皇后尚未放好
//在所有nb个安全位置上，(k+1)th皇后选择位置j的概率为1/nb
    k←k+1; //try[1..k+1]是(k+1)-promising
    try[k] ←j; //放置(k+1)th个皇后
    col ←col ∪ { j };
    diag45 ←diag45 ∪ { j-k };
    diag135 ←diag135 ∪ { j+k };
} //endif
until (nb=0) or (k=8) ; //当前皇后找不到合适的位置或try是
// 8-promising时结束。
success ← (nb>0) ;
}
```

§ 4.1 8后问题

❖ 分析

设 p 是成功的概率（一次成功）

s : 成功时搜索的结点的平均数(1个皇后放好算是搜索树上的1个结点)

e : 失败时搜索的结点的平均数。

显然 $s=9$ （空向量try算在内），

p 和 e 理论上难于计算，但实验用计算机可以计算出：

$$p=0.1293\dots$$

$$e=6.971\dots$$

在重复上述算法，直至成功时(相当于obstinate的时间)，所搜索的平均结点数：

$$t = s + (1 - p)e / p = 55.927\dots$$

大大优于回溯法，回溯法约为114个结点才能求出一个解。

Ex.证明：当放置 $(k+1)$ th 皇后时，若有多个位置是开放的,则算法 QueensLV 选中其中任一位置的概率相等。

§ 4.1 8后问题

❖ 问题及改进

- **消极**: LV算法过于消极, 一旦失败, 从头再来
- **乐观**: 回溯法过于乐观, 一旦放置某个皇后失败, 就进行系统回退一步的策略, 而这一步往往不一定有效。
- **折中**: 会更好吗? 一般情况下为此。

先用LV方法随机地放置前若干个结点, 例如 k 个。

然后使用回溯法放置后若干个结点, 但不考虑重放前 k 个结点。

若前面的随机选择位置不好, 可能使得后面的位置不成功, 如若前两个皇后的位置是1、3。

随机放置的皇后越多, 则后续回溯阶段的平均时间就越少, 失败的概率也就越大。

§ 4.1 8后问题

➤ 改进算法

折中算法只需将QueensLV的最后两行改为：

```
until nb = 0 or k = stepVegas;
```

```
if (nb>0) then //已随机放好stepVegas个皇后
```

```
    backtrace (k, col, diag45, diag135,success);
```

```
else
```

```
    success ←false;
```

stepVegas——控制随机放置皇后的个数，如何选择？

改进算法的试验结果：

Step	Vegas	p	s	e	t	
0		1	114	—	114	← 搜索的平均节点数 ← 完全回溯
1		1	39.63	—	39.63	
2		0.875	22.53	39.67	28.20	
3		0.4931	13.48	15.10	29.01	
4		0.2618	10.31	8.79	35.10	
5		0.1624	9.33	7.29	46.92	
6		0.1357	9.05	6.98	53.50	
7		0.1293	9	6.97	55.93	
8		0.1293	9	6.97	55.93	← 完全随机

纯回溯时间：40ms

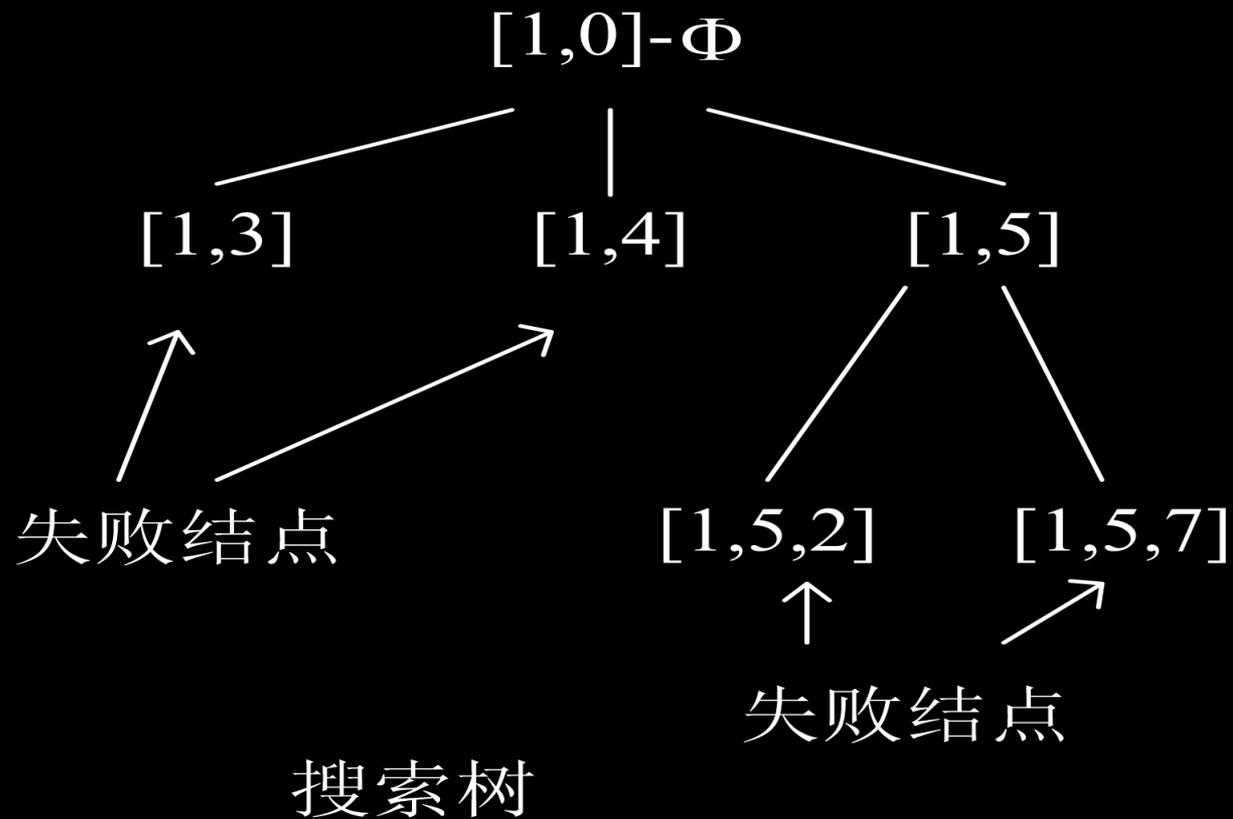
stepVegas=2 or 3：10ms（平均）

纯贪心LV：23ms（平均）

结论：一半略少的皇后随机放置较好。

§ 4.1 8后问题

-**问题1**: 为什么仅随机放一个皇后, 其效果就会大大优于纯回溯方法?



§ 4.1 8后问题

-问题2：预先随机选几个皇后放置为好？

由于解缺乏规律性（至少在皇后数不等于 $4k+2$ ， k 为某整数时），故求出stepVegas的最优值较难，但是找到一个较好（不一定是最优）的stepVegas还是可以的。

12皇后：

StepVegas	p	s	e	t	时间
0	1	262	—	262	125ms
5	0.5039	33.88	47.23	80.39	37ms
12	0.0465	13	10.2	222.11	基本与纯回溯相同

§ 4.1 8后问题

在Apple II机器上，20个皇后：

- ① 确定性的回溯算法：找到第一个解的时间大于2个小时。
- ② 概率算法，随机地放置前10个皇后：5分半钟找到36个不同的解。

后者找一个解比前者大约快1000倍！

-Obstinate算法在何时无限循环？

当问题不存在解时。

对于n皇后，要求 $n \geq 4$ ，即问题至少有一个解存在时，Obstinate算法才有可能结束。

Ex. 写一算法，求 $n=12 \sim 20$ 时最优的StepVegas值。

§ 4.2 模p平方根

1. **定义1:** 设 p 是一个奇素数, 若整数 $x \in [1, p-1]$ 且存在一个整数 y , 使
- $$x \equiv y^2 \pmod{p}$$

则称 x 为模 p 的二次剩余 (quadratic residue), 若 $y \in [1, p-1]$, 则 y 称为 x 模 p 的平方根。

- ❖ 例: 63是55模103的平方根, 55是模103的二次剩余。

$$\because 55 \bmod 103 = 55 \quad 63^2 \bmod 103 = 3969 \bmod 103 = 55$$

$$\therefore 55 \equiv 63^2 \pmod{103},$$

2. **定义2:** 若整数 $z \in [1, p-1]$, 且 z 不是模 p 的二次剩余, 则 z 是模 p 的非二次剩余。

§ 4.2 模p平方根

3. **定理1**: 任何一个模p的二次剩余至少有两个不同的平方根

pf: 设x是模p的二次剩余, y是x模p的平方根。

$$\text{因为 } (p-y)^2 = p^2 - 2py + y^2 \equiv y^2 \pmod{p}$$

$$\text{故 } x \equiv (p-y)^2 \pmod{p}$$

只要证 $p-y \neq y$ 且 $1 \leq p-y \leq p-1$ 就可证明 $p-y$ 是不同于 y 的x模p的另一个平方根。

$\because p$ 是奇数, $\therefore p-y \neq y$, 否则 p 是偶数。

另一方面,

$\because 1 \leq y \leq p-1$, $\therefore p-y \geq p - (p-1) = 1$ // $y \leq p-1$

$p-y \leq p-1$ // $y \geq 1$

§ 4.2 模p平方根

4. **定理2:** 任一模p的二次剩余至多有两个不同的平方根

pf: 设 $\forall a$ 和 b 是 x 模 p 的平方根。

$$\because a^2 \equiv b^2 \pmod{p} \quad a^2 = k_1 p + r, b^2 = k_2 p + r$$

$$\therefore a^2 - b^2 \equiv 0 \pmod{p}$$

即 $p \mid (a^2 - b^2)$ 成立。 $a^2 - b^2 = k \cdot p$

① 若 $k=0$, 则 $a=b$

② 若 $k>0$, 则 $a \neq b$

不妨设 $a > b$. $\because 1 \leq a - b < p \therefore p \nmid (a - b)$

又 $\because p \mid (a + b)(a - b)$, \therefore 因此可知 $p \mid (a + b)$

$$\text{即 } (a + b) = kp \quad \because (a + b) < 2p \therefore k = 1$$

$$\text{即 } a + b = p, \quad a = p - b$$

也就是说任意两个不同的平方根, 均只有 b 和 $(p-b)$ 两种不同形式。

§ 4.2 模 p 平方根

5. **定理3:** 1到 $p-1$ 之间的整数恰有一半是模 p 的二次剩余.

pf: 由定理1和定理2知, 任一模 p 的二次剩余恰有两个不同的平方根, 即: 任取二次剩余 $x \in [1, p-1]$, 只有 y 和 $p-y$ 这两个不同的平方根 $1 \leq y, p-y \leq p-1$

$$\because y^2 \equiv (p-y)^2 \pmod{p}$$

\therefore 在 $[1, p-1]$ 中恰有 $(p-1)/2$ 对不同的平方根, 每对平方根对应的模 p 的余数 x 必定在 $[1, p-1]$ 中, 即此区间上恰有 $(p-1)/2$ 个模 p 的二次剩余.

6. **定理4:** 对 $\forall x \in [1, p-1]$, p 是任一奇素数, 有 $x^{(p-1)/2} \equiv \pm 1 \pmod{p}$

且 x 是模 p 的二次剩余当且仅当 $x^{(p-1)/2} \equiv +1 \pmod{p}$

pf: 略 (可用费马定理)

§ 4.2 模 p 平方根

7. 如何判定 x 是否为模 p 的二次剩余？

只要利用定理4和计算方幂模 $x^{(p-1)/2} \bmod p$ 即可。

8. 已知 p 是奇素数， x 是模 p 的二次剩余，如何计算 x 模 p 的两个平方根？

- 当 $p \equiv 3 \pmod{4}$ 时，两平方根易求，分别是 $\pm x^{(p+1)/4}$
- 当 $p \equiv 1 \pmod{4}$ 时，没有有效的确定性算法，只能借助于Las Vegas算法。

§ 4.2 模p平方根

9. Las Vegas算法

用 \sqrt{x} 表示 x 的两个平方根中较小的一个。

■ **Def:** 模p乘法（类似于复数乘法）

$$a, b, c, d \in [0, p-1]$$

$$(a + b\sqrt{x})(c + d\sqrt{x}) \bmod p$$

$$= (ac + bdx + (ad + bc)\sqrt{x}) \bmod p$$

$$= ((ac + bdx) \bmod p + ((ad + bc) \bmod p)\sqrt{x}) \bmod p$$

//See p.863,(31.18)式

§ 4.2 模p平方根

❖ 例：设 $p = 53 \equiv 1 \pmod{4}$, $x = 7$. 求7的平方根

$$\because 7^{26} \equiv 1 \pmod{53} \quad // 26 = (53-1)/2$$

∴由定理4可知，7是模53的二次剩余，求7模53的平方根。

当省略模53符号时， $(1 + \sqrt{7})^{26} \pmod{53}$ 计算过程如下：

$$(1 + \sqrt{7})^2 = (1 + \sqrt{7})(1 + \sqrt{7}) = 8 + 2\sqrt{7}$$

$$(1 + \sqrt{7})^3 = (1 + \sqrt{7})(8 + 2\sqrt{7}) = 22 + 10\sqrt{7}$$

$$(1 + \sqrt{7})^6 = (22 + 10\sqrt{7})(22 + 10\sqrt{7}) = 18 + 16\sqrt{7}$$

$$(1 + \sqrt{7})^{12} = (18 + 16\sqrt{7})(18 + 16\sqrt{7}) = 49 + 46\sqrt{7}$$

$$(1 + \sqrt{7})^{13} = (1 + \sqrt{7})(49 + 46\sqrt{7}) = 0 + 42\sqrt{7}$$

$$(1 + \sqrt{7})^{26} = (0 + 42\sqrt{7})(0 + 42\sqrt{7}) = 52 + 0\sqrt{7}$$

§ 4.2 模p平方根

注： $(1 + \sqrt{7})^{26} \pmod{53} = (0 + 42\sqrt{7})(0 + 42\sqrt{7}) \pmod{53}$
 $= (42 \times 42 \times 7 \pmod{53} + 0\sqrt{7}) \pmod{53}$
 $= (12348 \pmod{53} + 0\sqrt{7}) \pmod{53}$
 $= (52 + 0\sqrt{7}) \pmod{53}$

上例中, $(1 + \sqrt{7})^{26} \equiv -1 \pmod{53}$, $\because 26 = (p-1)/2$

\therefore 由定理4知, $1 + \sqrt{7}$ 是模53的非二次剩余。

同样可知 $1 - \sqrt{7}$ 亦是模53的非二次剩余。

。

§ 4.2 模p平方根

- 若计算知当 $(a + \sqrt{7})^{26} \equiv c + d\sqrt{7} \pmod{53}$ 时, 已知 $(a + \sqrt{7}) \pmod{53}$ 和 $(a - \sqrt{7}) \pmod{53}$ 中有一个是模p的二次剩余, 而另一个不是二次剩余, 会怎样呢?

例如, 假定

$$c + d\sqrt{7} \equiv 1 \pmod{53} \quad // \text{即 } (a + \sqrt{7}) \pmod{53} \text{ 是二次剩余, 定理4}$$

$$c - d\sqrt{7} \equiv -1 \pmod{53} \quad // \text{即 } (a - \sqrt{7}) \pmod{53} \text{ 不是二次剩余}$$

两等式相加得: $2c \equiv 0 \pmod{53}$

$$\because 0 \leq c \leq 52 \therefore c = 0$$

两式相减得: $2d\sqrt{7} \equiv 2 \pmod{53}$

$$\therefore d\sqrt{7} \equiv 1 \pmod{53}$$

§ 4.2 模 p 平方根

- 例：通过计算可知 $(2 + \sqrt{7})^{26} \equiv 0 + 41\sqrt{7} \pmod{53}$

为了获得7的一个平方根，需要找唯一的一个整数 y 使得 $1 \leq y \leq 52$ ， $41y \equiv 1 \pmod{53}$ 。

这可使用一个Euclid算法解决

$\because 41 \times 22 \equiv 1 \pmod{53}$ ，故 $y=22$ 。它是7模53一个平方根

$\because 7 \equiv 22^2 \pmod{53}$ ，另一平方根为 $53-22=31$

- 算法

设 x 是模 p 的二次剩余， p 是素数且

$$p \equiv 1 \pmod{4}, \text{找 } y^2 \equiv x \pmod{p}$$

§ 4.2 模p平方根

```
rootLV(x, p, y, success){//计算y
  a ← uniform(1..p-1);//我们并不知道a应取多少
  if  $a^2 \equiv x \pmod{p}$  then { //可能性很小
    success ← true; y ← a;
  }else{
    计算c, d使得  $0 \leq c, d \leq p-1, (a+\sqrt{x})^{(p-1)/2} \equiv c+d\sqrt{x} \pmod{p}$ ;
    if d=0 then
      success ← false; //无法求出 $\sqrt{x}$ 
    else{ //c=0
      success ← true;
      计算y使  $d \cdot y \equiv 1 \pmod{p}, 1 \leq y \leq p-1$ //修改Euclid算法可求y
    }
  }
}
```

算法成功的概率 >0.5 ,接近0.5。故平均调用两次即可求得x的平方根

§ 4.3 整数的因数分解

设 n 是一个大于1的整数，因数分解问题是找到 n 的一个唯一分解： $n = p_1^{m_1} p_2^{m_2} \cdots p_k^{m_k}$

这里 m_i 是正整数，且 $p_1 < p_2 < \cdots < p_k$ 均为素数。

若 n 是合数，则至少有1个非平凡的因数(不是1和 n 本身)。

设 n 是一个合数， n 的因数分解问题，即找 n 的非平凡因数，它由两部分构成：

- ① $\text{prime}(n)$ ——判定 n 是否为素数，可由Monte Carlo算法确定。
- ② $\text{split}(n)$ ——当 n 为合数时，找 n 的一个非平凡的因数。

§ 4.3 整数的因数分解

1. 朴素的split算法

```
split(n) {  
    //n是素数,返回1, 否则返回找到的n的最小非平凡因数  
    for i ← 2 to  $\lfloor \sqrt{n} \rfloor$  do  
        if (n mod i) = 0 then  
            return i; //i ≥ 2  
    return 1; //返回平凡因数  
}
```

§ 4.3 整数的因数分解

- 性能分析: $T(n) = \Omega(\sqrt{n})$ ——最坏情况。

当 n 是一个中等规模的整数（如大约50位十进制整数）时，最坏情况的计算时间亦不可接受。

$\therefore n$ 的位数 $m = \lceil \log_{10}(n+1) \rceil$

$\therefore \sqrt{n} \approx 10^{m/2}$ ；当 $m=50$ 时，上述算法的时间约为 10^{25}

无论是确定性的还是概率的，没有算法能够在多项式时间 $O(p(m))$ 内分解 n 。Dixon的概率算法分解 n 的时间为 $O(2^{\sqrt{m \log_{10} m}})$

Note: 无论 k 和 b 是何正常数，均有：

$$O(m^k) \subset O(2^{O(\sqrt{m \log_{10} m})}) \subset O(10^{m/b})$$

§ 4.3 整数的因数分解

2. 合数的二次剩余（模素数到模合数的推广）

设 n 是任一正整数，整数 $x \in [1, n-1]$ 。若 x 和 n 互素，且存在一整数 $y \in [1, n-1]$ 使 $x \equiv y^2 \pmod{n}$ ，则称 x 为模 n 的二次剩余，称 y 为 x 模 n 的平方根。

一个模 p 的二次剩余，当 p 为素数时，恰有两个不同的平方根，但 p 为合数，且至少有两个奇素数因子时，不再为真。例： $8^2 \equiv 13^2 \equiv 22^2 \equiv 27^2 \equiv 29 \pmod{35}$ ，注意29应与35互素，才有可能为模35的二次剩余。

- 定理：若 $n=pq$ ， p 、 q 是两个互不相同的素数，则每一个模 n 的二次剩余恰有4个平方根。

§ 4.3 整数的因数分解

上节的测试 x 是否是模 p 的二次剩余及找 x 的平方根的方法是一个有效的算法（指rootLV），当 n 是一个合数，且 n 的因子分解给定时，同样存在有效的算法。但 n 的因数分解未给定时，目前还没有有效算法测试 x 是否为二次剩余及找 x 的平方根。

3. Dixon因数分解算法

基本思想，找两个与 n 互素的整数 a 和 b ，使 $a^2 \equiv b^2 \pmod{n}$ 但 $a \not\equiv \pm b \pmod{n}$ 蕴含着 $a^2 - b^2 = (a-b)(a+b) \equiv 0 \pmod{n}$ 即 $n \mid (a+b)(a-b)$

假定 $n \nmid (a+b)$ ， $n \nmid (a-b)$ ，则 n 的某一非平凡因子 x 满足：

$$x \mid (a+b), (n/x) \mid (a-b)$$

$\therefore n$ 和 $a+b$ 的最大公因子是 n 的一个非平凡因子。

例如： $a=8, b=13, n=35$.

$a+b=21$ 和 $n=35$ 的gcd是 $x=7$ ， x 是35的一个非平凡因子

§ 4.3 整数的因数分解

```
Dixon (n, x, success){//找合数n的某一非平凡因子x
  if n是偶数 then{
    x←2; success ←true;
  }else{
    for i ←2 to  $\lfloor \log_3 n \rfloor$  do
      if  $n^{1/i}$  是整数 then{
        x ←  $n^{1/i}$ ; success ←true; return;
      }//:n是合数且为奇数, 现在知道它至少有2个不同的奇素数因子
      a, b ←两个使得  $a^2 \equiv b^2 \pmod{n}$  的整数
      if  $a \equiv \pm b \pmod{n}$  then
        success ←false;
      else{
        x ←gcd (a+b,n); success ← true;
      }
    }
  }
}
```

§ 4.3 整数的因数分解

4. 如何确定a和b使 $a^2 \equiv b^2 \pmod{n}$ ，来对n因数分解。

- Def. k-平滑:

若一个整数x的所有素因子均在前k个素数之中，则x称为k-平滑的。

- 例如： $120 = 2^3 \times 3 \times 5$ 是3-平滑的

$35 = 5 \times 7$ 不是3-平滑的， $\because 7$ 是第四个素数

\therefore 它是4-平滑的，也是5-平滑的...

当k较小时，k-平滑的整数可用朴素的split算法进行有效的因数分解。Dixon算法可以分为3步确定a和b。

§ 4.3 整数的因数分解

Step1: 在 $1 \sim n-1$ 之间随机选择 x

- i) 若 x 碰巧不与 n 互素, 则已找到 n 的一个非平凡因子(即为 x)
- ii) 否则设 $y = x^2 \bmod n$, 若 y 是 k -平滑, 则将 x 和 y 的因数分解保存在表里。

此过程重复直至选择了 $k+1$ 个互不相同的整数, 并且这些整数的平方模 n 的因数已分解(当 k 较小时, 用 $\text{split}(n)$ 分解)

例1: 设 $n=2537$, $k=7$.

前7个整数为: 2, 3, 5, 7, 11, 13, 17

若随机选取 $x=1769$, $y = 1769^2 \bmod 2537 = 1240$

$$\because 1240 = 2^3 \times 5 \times 31$$

$\therefore 1240$ 不是7-平滑的

§ 4.3 整数的因数分解

下述8个x的平方模n是7-平滑的:

$$x_1 = 2455, y_1 = 1650 = 2 \times 3 \times 5^2 \times 11$$

$$x_2 = 970, y_2 = 2210 = 2 \times 5 \times 13 \times 17$$

$$x_3 = 1105, y_3 = 728 = 2^3 \times 7 \times 13$$

$$x_4 = 1458, y_4 = 2295 = 3^3 \times 5 \times 17$$

$$x_5 = 216, y_5 = 990 = 2 \times 3^2 \times 5 \times 11$$

$$x_6 = 80, y_6 = 1326 = 2 \times 3 \times 13 \times 17$$

$$x_7 = 1844, y_7 = 756 = 2^2 \times 3^3 \times 7$$

$$x_8 = 433, y_8 = 2288 = 2^4 \times 11 \times 13$$

§ 4.3 整数的因数分解

Step2: 在 $k+1$ 个等式之中找一个非空子集，使相应的因数分解的积中前 k 个素数的指数均为偶数(包含0)

例2: 在上例的8个等式中，有7个积符合要求：

$$y_1 y_2 y_4 y_8 = 2^6 \times 3^4 \times 5^4 \times 7^0 \times 11^2 \times 13^2 \times 17^2 \text{ (解一)}$$

$$y_1 y_3 y_4 y_5 y_6 y_7 = 2^8 \times 3^{10} \times 5^4 \times 7^2 \times 11^2 \times 13^2 \times 17^2 \text{ (解二)}$$

- 可以证明，在 $k+1$ 个等式中，至少存在这样一个解，如何找到一个解？

构造一个0-1矩阵 A : $(k+1) \times k$

矩阵的行对应 $k+1$ 个 y_i ，列对应前 k 个素数。

$$a_{ij} = \begin{cases} 0 & \text{若 } y_i \text{ 的第 } j \text{ 个素数的指数为偶数} \\ 1 & \text{若 } y_i \text{ 的第 } j \text{ 个素数的指数为奇数} \end{cases}$$

§ 4.3 整数的因数分解

∴矩阵的行数大于列数。

∴在模2意义下，矩阵的行之间不可能均是相互独立的。

例如在例2中，第一个解就是线性相关的：

$$y_1 = 2^1 \times 3^1 \times 5^2 \times 7^0 \times 11^1 \times 13^0 \times 17^0$$

$$(1, 1, 0, 0, 1, 0, 0) + (1, 0, 1, 0, 0, 1, 1) +$$

$$(0, 1, 1, 0, 0, 0, 1) + (0, 0, 0, 0, 1, 1, 0) \equiv (0, 0, 0, 0, 0, 0, 0) \pmod{2}$$

使用Gauss-Jordan消去法可找到线性相关的行。

Step3: 在step2中找到线性相关的行后：

1) 令a为相应 x_i 的乘积

2) 令b是 y_i 的乘积开平方

若 $a \not\equiv \pm b \pmod{n}$ ，则只需求a+b和n的最大公因子即可获得n的非平凡因子。

§ 4.3 整数的因数分解

例3: 对于例2中的第1个解有:

$$a = x_1 x_2 x_4 x_8 \bmod n = 2455 \times 970 \times 1458 \times 433 \bmod 2537 = 1127$$

$$b = 2^3 \times 3^2 \times 5^2 \times 7^0 \times 11 \times 13 \times 17 \bmod 2537 = 2012 \not\equiv \pm a \pmod{n}$$

$a+b=3139$ 和 $n=2537$ 的最大公因子是43, 它是 n 的一个非平凡因子。

对于例2中的第2个解有:

$$a = x_1 x_3 x_4 x_5 x_6 x_7 \bmod n = 564$$

$$b = 2^4 \times 3^5 \times 5^2 \times 7 \times 11 \times 13 \times 17 \bmod n = 1973 \equiv -a \pmod{n}$$

此解不能求因子。

实际上 $a \not\equiv \pm b \pmod{n}$ 的概率至少为1/2

§ 4.3 整数的因数分解

5. 时间分析

如何选择 k .

1) k 越大, $x^2 \bmod n$ 是 k -平滑的可能性越大(x 是随机选取的)

2) k 越小, 测试 k -平滑及因数分解 y_i 的时间越小, 确定 y_i 是否线性相关的时间也越少, 但 $x^2 \bmod n$ 不是 k -平滑的概率也就较大。

设 $L = e^{\sqrt{\ln n \ln \ln n}}$, $b \in R^+$

通常取 $k \approx \sqrt{L}$ 时较好, 此时Dixon算法分裂 n 的期望时间为 $O(L^2) = O(e^{2\sqrt{\ln n \ln \ln n}})$, 成功的概率至少为 $1/2$.

Ch.5 Monte Carlo算法

存在某些问题，无论是确定的还是概率的，均找不到有效的算法获得正确的答案。

Monte Carlo算法偶尔会犯错，但它无论对何实例均能以高概率找到正确解。当算法出错时，没有警告信息。

1. 基本概念

- **Def1**: 设 p 是一个实数，且 $1/2 < p < 1$ ，若一个MC算法以不小于 p 的概率返回一个正确的解，则该MC算法称为 p -正确，算法的优势（advantage）是 $p-1/2$ 。
- **Def2**: 若一个MC算法对同一实例决不给出两个不同的正确解，则该算法称是相容的（consistent）或一致的。

Ch.5 Monte Carlo算法

某些MC算法的参数不仅包括被解的实例，还包括错误概率的上界。因此，这样算法的时间被表示为实例大小及相关可接受的错误概率的函数。

- **基本思想**：为了增加一个一致的、 p -正确算法成功的概率，只需多次调用同一算法，然后选择出现**次数最多**的解。

例：设MC(x)是一个一致、75%-correct的MC算法，考虑下述算法：

```
MC3(x){  
    t←MC(x); u←MC(x); v←MC(x);  
    if t=u or t=v then return t;  
    else return v;  
}
```

Ch.5 Monte Carlo算法

该算法是一致的和27/32-correct的(约84%)

pf: 相容性 (一致性) 易证。

\therefore t、u、v正确的概率为75%=3/4=p

\therefore 错误的概率为q=1/4.

1) 若t、u、v均正确, \therefore MC是一致的 \therefore t=u=v, 则MC3返回的t正确, 此概率为: $(3/4)^3$

2) 若t、u、v恰有两个正确则MC3返回 $\begin{cases} t \text{ 正确 if } t = u \text{ or } t = v \\ v \text{ 正确 if } u = v \end{cases}$
此概率为:

$$C_3^2 p^2 q^1 = 3 \times (3/4)^2 (1/4)$$

3) 若t、u、v恰有一个正确, 则只有v正确时, MC3返回正确答案, 此概率为: $p q^2 = (3/4) (1/4)^2$

Ch.5 Monte Carlo算法

严格的说，当v正确，只有两个错误的解t和u不相等时，才有可能成功，因此MC3成功的概率为：

$$\left(\frac{3}{4}\right)^3 + 3\left(\frac{3}{4}\right)^2\left(\frac{1}{4}\right) + \frac{3}{4}\left(\frac{1}{4}\right)^2 = \frac{27}{32} + \frac{3}{64} > \frac{27}{32} \approx 84\%$$

多运行2次（共3次）使成功率 $75\% \nearrow 84\%$

- **Theorem:** 设2个正实数之和 $\varepsilon+\delta<0.5$ ，MC(x)是一个一致的、 $(0.5+\varepsilon)$ -correct 的蒙特卡洛算法，设 $C_\varepsilon = -2 / \lg(1-4\varepsilon^2)$ ，x是某一被解实例，若调用MC(x)至少 $\lceil C_\varepsilon \lg(1/\delta) \rceil$ 次，并返回出现频数最高的解，则可得到一个解同样实例的一致的 $(1-\delta)$ -correct的新MC算法

Ch.5 Monte Carlo算法

由此可见，无论原算法MC(x)的赢面(优势) $\epsilon > 0$ 是多小，均可通过反复调用来扩大其优势，使得最终的算法具有可接受的错误概率 δ ，可达到任意小（选定的）。

pf: 设 $n \geq C_\epsilon / \lg 1 / \delta$ 是调用MC(x)的次数， $m = \lfloor n/2 \rfloor + 1$

$p = 1/2 + \epsilon$ //MC成功的概率

$q = 1 - p = 1/2 - \epsilon$ //MC失败的概率

当重复调用MC(x)算法n次时，若正确解至少出现m次，则新算法返回频度最高的解必为正确解；若正确解出现的次数不超过n/2时，不能保证新算法找到了正确解。因此，出错概率至多为：

$$\sum_{i=0}^{m-1} \Pr [n次调用中出现i次正确解]$$

Ch.5 Monte Carlo算法

$$\leq \sum_{i=0}^{m-1} \binom{n}{i} p^i q^{n-i} \quad // \text{实际上, 当正确解出现次数} < m \text{时,}$$

// 亦可能返回正确解。

$$= (pq)^{n/2} \sum_{i=0}^{m-1} \binom{n}{i} (q/p)^{n/2-i}$$

$$\leq (pq)^{n/2} \sum_{i=0}^n \binom{n}{i} \quad // q < p, n/2 - i \geq 0$$

$$= (pq)^{n/2} \times 2^n$$

$$= (4pq)^{n/2} = (1 - 4\varepsilon^2)^{n/2}$$

$$\leq (1 - 4\varepsilon^2)^{(C_\varepsilon/2) \lg(1/\delta)} \quad // 0 \leq 1 - 4\varepsilon^2 < 1, \text{用 } C_\varepsilon \lg(1/\delta) \text{ 取代 } n$$

$$= 2^{-\lg(1/\delta)} \quad // C_\varepsilon / 2 = -1 / \lg(1 - 4\varepsilon^2), \text{对任意 } x > 0, x^{1/\lg x} = 2$$

$$= 2^{\lg \delta} = \delta \text{ 由此可知, 重复MC(x) } n \text{ 次成功的概率至少为 } 1 - \delta. \quad 113$$

Ch.5 Monte Carlo算法

例：假定有一个一致的55%赢面的MC算法，若希望获得一个错误概率小于5%的算法，则相当于将55%-correct的算法改造成95%-correct的算法。

上述定理告诉我们：大约要调用MC算法600次才能达到相应的精度（在同一实例上， $n \geq C_{\epsilon} \lg 1/\delta$ ）

上述证明太过粗略，更精确的证明表示：

如果重复调用一个一致的， $(1/2+\epsilon)$ -correct的MC算法 $2m-1$ 次，则可得到一个 $(1-\delta)$ -correct的最终算法，其中：

$$\delta = \left(\frac{1}{2} - \epsilon\right) \sum_{i=0}^{m-1} \binom{2i}{i} \left(\frac{1}{4} - \epsilon^2\right)^i \leq \frac{\left(1 - 4\epsilon^2\right)^m}{4\epsilon\sqrt{\pi m}}$$

$$m = \left\lceil x / 4\epsilon^2 \right\rceil, \text{ 须先确定 } x \text{ 使 } e^{\sqrt{x}} \geq 1 / (2\delta\sqrt{\pi})$$

Ch.5 Monte Carlo算法

2.有偏算法

重复一个算法几百次来获得较小的出错概率是没有吸引力的，幸运地，大多数MC算法实际上随着重复次数的增加，正确概率增长会更快。

- **Def:** (偏真算法)为简单起见，设MC(x)是解某个判定问题，对任何x，若当MC(x)返回true时解总是正确的，仅当它返回false时才有可能产生错误的解，则称此算法为偏真的(true-biased)。

显然，在偏真的MC算法里，没有必要返回频数最高的解，因为一次true超过任何次数的false。

对于偏真的MC算法，重复调用4次，就可将55%-正确的算法改进到95%正确。6次重复就可得到99%正确的算法，且对 $p > 1/2$ 的要求可以放宽到 $p > 0$ 即可。

Ch.5 Monte Carlo算法

Def: (偏 y_0 算法)更一般的情况不再限定是判定问题, 一个MC是偏 y_0 的(y_0 是某个特定解), 如果存在问题实例的子集 X 使得:

- ① 若被解实例 $x \in X$, 则算法 $MC(x)$ 返回的解总是正确的(无论返回 y_0 还是非 y_0)
- ② 若 $\forall x \in X$, 正确解是 y_0 , 但MC并非对所有这样的实例 x 都返回正确解。

虽然 y_0 是必须知道的, 但无需测试 x 是否是 X 的成员, 下面解释若算法返回 y_0 时, 它总是正确的。

即算法返回 y_0 时总是正确的, 返回非 y_0 时以 p 为概率正确。

Ch.5 Monte Carlo算法

设MC是一个一致的、 p -correct和偏 y_0 的蒙特卡洛算法， x 是一个实例， y 是MC(x)返回的解，可分为如下2种情形讨论：

case1: $y = y_0$

- 若 $x \notin X$ ，则算法MC总是返回正确解，因此 y_0 确实是正确的。
- 若 $x \in X$ ，算法返回的正确解必定是 y_0

这两种情况均可得到结论： y_0 是一个正确解。

case2: $y \neq y_0$

- 若 $x \notin X$ ，则 y 是正确解。
- 若 $x \in X$ ，因为正确解是 y_0 ，故 y 是错误解，此出错概率不超过 $1-p$ 。

Ch.5 Monte Carlo算法

■ 有偏算法重复调用MC: 优先返回 y_0

若 k 次调用MC(x)所返回解依次是 y_1, y_2, \dots, y_k , 则:

(1) 若存在 i 使 $y_i = y_0$, 则前面的讨论已知, 它是一个正确解(y_i 是正确解).

(2) 若存在 $i \neq j$ 使 $y_i \neq y_j$, 由于MC是一致的, 故必有 $x \in X$ 。因此正确解仍是 y_0 。

(3) 若对所有的 i 均有 $y_i = y$, 但 $y \neq y_0$, 则 y_0 仍有可能是正确解。实际上, 若 $x \in X$, 则 y_0 是正确解, 此时 y 是错误的, 其错误概率不超过 $(1-p)^k$ 。

由上面的讨论可知, 重复调用一个一致的, p -正确的, 偏 y_0 的MC算法 k 次, 可得到一个 $(1-(1-p)^k)$ -正确的算法(对偏真算法亦适合).

例: $p=0.55$, $k=4$, 即可将算法正确率提高到95%

§ 5.1 主元素问题

Def: 设 $T[1..n]$ 是 n 个元素的数组，若 T 中等于 x 的元素个数大于 $n/2$ (即 $|\{i \mid T[i]=x\}| > n/2$), 则称 x 是数组 T 的主元素。

(Note: 若存在, 则只可能有1个主元素)

1. 判 T 是否有主元素

$\text{maj}(T)$ { //测试随机元素 $x \in T$ 是否为 T 的主元素

$i \leftarrow \text{uniform}(1..n);$

$x \leftarrow T[i];$

$k \leftarrow 0;$

 for $j \leftarrow 1$ to n do

 if $T[j]=x$ then

$k \leftarrow k+1;$

 return $(k > n/2);$

}

§ 5.1 主元素问题

- 若算法返回true，则T含有主元素，所选择的元素即为主元素，算法一定正确。
- 若算法返回false，则T仍有可能含有主元素，只是所选元素x不是T的主元素而已。
- 若T确实包含一个主元素，则随机选择到一个非主元素的概率小于1/2，这是因为主元素占T的一半以上。
- 算法是一个偏真的1/2正确的算法：
 - ① 若maj返回true，则T必有主元素，解一定正确。因为随机选中主元素的概率大于1/2，故该算法是1/2-correct.
 - ② 若maj返回false，则T可能没有主元素。当然，若T没有主元素，则必返回false。

§ 5.1 主元素问题

实际使用时，50%的错误概率是不可容忍的。而对有偏算法，可以通过重复调用技术来使错误概率降低到任何值。

```
maj2(T) {  
    if maj (T) then  
        return true; //1次成功  
    else //1次失败后调用第2次  
        return maj(T); //调用2次  
}
```

§ 5.1 主元素问题

2. 分析:

1) 若T无主元素, 则maj每次均返回false, maj2亦肯定返回false。此时算法返回值正确 (成功)。

2) 若T有主元素, 则:

① 第一次调用maj返回真的概率是 $p > 1/2$, 此时maj2亦返回真。

② 第一次调用maj返回false的概率为 $1-p$, 第2次调用maj仍以概率 p 返回true, maj2亦返回真, 其概率为: $(1-p)p$. 此时算法返回值正确 (成功)。

总结: 当T有主元时, maj2返回真的概率是:

$$p + (1-p)p = 1 - (1-p)^2 > 3/4.$$

即: maj2是一个偏真的3/4正确的MC算法。

§ 5.1 主元素问题

3. 算法改进

错误的概率能够迅速减小，主要是因为重复调用maj的结果是相互独立的，即：对同一实例T，某次maj返回false，并不会影响继续调用返回true的概率。

因此，当T含有主元素时，k次重复调用maj均返回false的概率小于 2^{-k} 。

另一方面，在k次调用中，只要有一次maj返回真，即可判定T有主元素。

§ 5.1 主元素问题

- 当需要控制算法出错概率小于 $\varepsilon > 0$ 时，相应算法调用maj的次数为：

$$\varepsilon = 2^{-k} \Rightarrow k = \left\lceil \lg \frac{1}{\varepsilon} \right\rceil$$

majMC (T, ε) {

$k \leftarrow \lceil \lg(1/\varepsilon) \rceil$;

 for $i \leftarrow 1$ to k do

 if maj(T) then return true; //成功

 return false; //可能失败

}

该算法的时间为 $O(n \lg(1/\varepsilon))$ 。注意，这里只是用此问题来说明MC算法，实际上对于判定主元素问题存在 $O(n)$ 的确定性算法。

§ 5.2 素数测定(数的素性测定)

- 判定一个给定的整数是否为素数，到目前为止尚未找到有效的确定性算法或Las Vegas算法。

1. 简单的概率算法。

```
prime(n) {  
    d ← uniform(2.. $\lfloor \sqrt{n} \rfloor$  );  
    return (n mod d) ≠ 0  
}
```

- 若返回false，则算法幸运地找到了n的一个非平凡因子，n为合数。
- 若返回true，则未必是素数。实际上，若n是合数，prime亦以高概率返回true。

§ 5.2 素数测定(数的素性测定)

例如: $n = 2623 = 43 \times 61$, $\lfloor \sqrt{2623} \rfloor = 51$

prime在2~51内随机选一整数d

❖成功: $d=43$, 算法返回false(概率为2%), 结果正确

❖失败: $d \neq 43$, 算法返回true(概率为98%), 结果错误

当n增大时, 情况更差。

2. Fermat小定理

若n是素数, 则 $\forall a \in [1, n-1]$, 有 $a^{n-1} \bmod n = 1$

■ 变换命题 (逆否定理)

设n和a是整数, 若 $\exists a \in [1, n-1]$ 使 $a^{n-1} \bmod n \neq 1$, 则n不是素数。

§ 5.2 素数测定(数的素性测定)

- 素性测定算法（偏假的）：

```
Fermat(n) {  
    a ← uniform(1..n-1);  
    if  $a^{n-1} \bmod n = 1$  then  
        return true; //未必正确，n未必为素数  
    else  
        return false; //正确，n一定是合数  
}
```

- Fermat定理的逆命题成立吗？

即是否只要 $a^{n-1} \bmod n = 1$ for all $a \in [1, n-1]$ 成立，n就是素数？早期学者认为成立，甚至认为只要验证了 $a=2$ 即可。

§ 5.2 素数测定(数的素性测定)

当 n 为合数, 对 $\forall a \in [1, n-1]$, 有 $a^{n-1} \bmod n \neq 1$ 吗?

若成立, 则只要 n 是合数, $\forall a \in [1, n-1]$, 均有 $a^{n-1} \bmod n \neq 1$
否则 n 为素数。遗憾的是此命题亦不成立。

❖**结论:** 我们不能通过验证 $a^{n-1} \bmod n$ 是否为1来判定 n 是否为素数

例如:

$$1^{n-1} \bmod n = 1 \text{ for all } n \geq 1$$

$$(n-1)^{n-1} \bmod n = 1 \text{ if } n \geq 3.$$

若是 $a \in [2, n-2]$ 时呢?

设 $n=15$ (合数), $a=4$, $4^{14} \bmod 15 = 1$.

§ 5.2 素数测定(数的素性测定)

3. 伪素数和素性伪证据

设 $2 \leq a \leq n-2$ ，一个满足 $a^{n-1} \bmod n = 1$ (即 n 可整除 $a^{n-1}-1$) 的合数 n 称为以 a 为底的**伪素数**， a 称为 n 的素性**伪证据**。

实际上，符合费马小定理逆命题的数，我们称为拟素数 (Probable Prime)。拟素数一是素数，二是伪素数。

后来数学家渐把符合一些素数性质的逆命题的合数称为伪素数

■ 伪素数有多少？

在前10亿个自然数中，有50,847,534个素数，而以2为底的伪素数则只有5597个。因此在 n 整除 $2^{n-1}-1$ 的情况下，出现合数的机会仅有 $5597/(5597+50847534) = 0.00011$ 。

而我们同时考虑以2和3为底的伪素数，则只有1272个。因此 n 同时可以整除 $2^{n-1}-1$ 和整除 $3^{n-1}-1$ 的情况下，碰上合数的机会便更低了，仅有 $1272/(1272+50847534) = 0.000025$ 。

若我们把该测试扩张至其他底，自然会将找到伪素数的机会降低，会降至0吗？**否！**

§ 5.2 素数测定(数的素性测定)

若将Fermat测试改为从 $2 \sim n-2$ 之间随机选 a ，则只有选到一个伪证据时，对合数的测试失败(返回true).

■ 伪证据有多少？

➤ 总体情况是伪证据相当少

1000之内的奇合数测试误差概率 $<3.3\%$ 。n较大时，概率更小。

➤ 有些合数伪证据比例相当高

如561，有318个伪证据，超过证据数的一半（ $2 \sim 559$ ）。
极端情况：Fermat(651693055693681)返回true的概率 $>99.9965\%$

➤ 一般地，对 $\forall \delta > 0$ ，存在无穷多个合数，使得Fermat测试发现他们是合数的概率小于 δ

即：对任意的 $p > 0$ ，Fermat测试都不是 p -正确的。因此，以前将Fermat测试重复固定次数，并不能将误差降到任意小的 ϵ 内。

§ 5.2 素数测定(数的素性测定)

4. Fermat测试改进

■ 强伪素数

设 n 是一个大于4的奇整数， s 和 t 是使得 $n-1=2^s t$ 的正整数，其中 t 为奇数，设 $B(n)$ 是如下定义的整数集合：

$a \in B(n)$ 当且仅当 $2 \leq a \leq n-2$ 且满足下述2个条件之一：

① $a^t \bmod n = 1$ // $a^t \equiv 1 \pmod{n}$

或

② \exists 整数 i , $0 \leq i < s$ 满足 $a^{2^i t} \bmod n = n-1$ // $a^{2^i t} \equiv -1 \pmod{n}$

当 n 为素数时， $\forall a \in [2, n-2]$ ，均有 $a \in B(n)$

当 n 为合数时，若 $a \in B(n)$ ，则称 n 为一个以 a 为底的**强伪素数**，称 a 为 n 素性的**强伪证据**。

n 为素数，说明它对所有底均为强伪素数。

§ 5.2 素数测定(数的素性测定)

```
Btest(a, n){//n为奇数, a ∈ [2, n-2], 返回 true ⇔ a ∈ B(n)。即返回  
//真说明n是强伪素数或素数  
    s ← 0; t ← n-1; // t开始为偶数  
    repeat  
        s++; t ← t ÷ 2;  
    until t mod 2 = 1; //n-1=2st, t为奇数  
    x ← at mod n;  
    if x=1 or x=n-1 then return true; //满足①or②, a ∈ B(n)  
    for i ← 1 to s-1 do{ //验证 a2it mod n = n-1  
        x ← x2 mod n;  
        if x=n-1 then return true; //满足②, a ∈ B(n)  
    }  
    return false;  
}
```

§ 5.2 素数测定(数的素性测定)

例: $158 \in B(289)$?

$$\because n-1=288=2^5 \times 9 \quad //289=17 \times 17 \text{ 为合数}$$

$$\therefore s=5, t=9.$$

$$\text{计算 } x = a^t \bmod n = 158^9 \bmod 289 = 131$$

执行for循环4次(只要3次)。

$$a^{2t} \bmod n = 131^2 \bmod 289 = 110$$

$$a^{2^2 t} \bmod n = 110^2 \bmod 289 = 251$$

$$a^{2^3 t} \bmod n = 251^2 \bmod 289 = 288$$

$\therefore 158 \in B(289)$. 158是一强伪证据

§ 5.2 素数测定(数的素性测定)

- 强伪证据数目比伪证据数目少很多

强伪证据是伪证据，但反之不然。

例：4是15的素性伪证据 $4^{14} \bmod 15 = 1$

但它不是一个强伪证据， $\because 4^7 \bmod 15 = 4 \nexists a \in B(n)$

561有318个伪证据，但只有8个是强伪证据。

小于1000的奇合数中，随机选到一个强伪证据的概率小于1%

更重要的是，对任一奇合数，强伪证据比例都很小。

§ 5.2 素数测定(数的素性测定)

- **Th1.** 设 n 是任一大于4的奇数。

① 若 n 是素数, 则 $B(n) = \{a \mid 2 \leq a \leq n-2\}$

② 若 n 是合数, 则 $|B(n)| \leq (n-a)/4$

即, 当 n 为合数时, 强伪证据数目 $<1/4$ 。因此, 当随机选 a 时, 它返回false的概率 $>3/4$, 正确的概率 $>75\%$ 。

当 n 为素数时, Btest总返回真。

- **Miller-Rabin测试**

MillRab(n) { //奇 $n > 4$, 返回真时表示素数, 假表示合数

$a \leftarrow \text{uniform}(2..n-2);$

return Btest(a, n); //测试 n 是否为强伪素数

}

§ 5.2 素数测定(数的素性测定)

说明：该算法是3/4-正确，偏假的。

- ① 返回真时，它可能是伪素数，但是随机选到的强伪证据的概率 $<1/4$ ，出错概率 $<1/4$
- ② 返回假时， n 为合数，它必正确。

∵若 n 是素数，由定理1知，它必定返回真，任何 $a \in B(n)$.

∴ n 必定是一个合数。

```
RepeatMillRab(n,k){  
    for i ← 1 to k do  
        if MillRab(n) = false then  
            return false; //一定是合数  
    return true;  
}
```

§ 5.2 素数测定(数的素性测定)

重复调用 k 次之后返回true, 若错误则表示连续 k 次碰到强伪证据, 概率 $<(1/4)^k$ 。只要取 $k=10$, 错误概率 $<$ 百万分之一

即RepeatMiller(\bullet , k)是 $(1-4^{-k})$ -正确的MC算法。

■ 时间

若要求出错概率不超过 ϵ , 则 $4^{-k} \leq \epsilon$, $2^{2k} \geq 1/\epsilon$, 重复Miller-

Rabin测试次数: $k = \lceil \lg(1/\epsilon) / 2 \rceil$

每次调用Miller时间:

- ① 模幂运算: $a^t \bmod n$ —— 模乘法和模平方运算次数 $O(\lg t)$
- ② $s-1$ 次模平方运算: $x^2 \bmod n$ —— 模平方与乘法类似

§ 5.2 素数测定(数的素性测定)

$\because \lg n > \lg(n-1) = \lg 2^s t = s + \lg t$

\therefore 一次Miller-Rabin执行时间主要是进行 $O(\lg n)$ 次模乘法。

通过传统的算法实现，每次模乘时间为 $O(\lg^2 n)$ 。

-**结论**：确定 n 素性时间为： $O(\lg^3 n \lg \frac{1}{\varepsilon})$

应用中， n 为上千位数字， $\varepsilon = 10^{-100}$ ，该时间完全合理。

■ 问题

\therefore Miller-Rabin测试是偏假的

\therefore 当返回false肯定正确，即判定 n 是合数是完全正确的。

但返回真时，只能说 n 是以高概率为素数。

例如， $k=10$ ，出错概率 $\varepsilon = 4^{-10} = 2^{-20} < \text{百万分之一}$

§ 5.2 素数测定(数的素性测定)

可以说 n 是素数，但有 2^{-20} 的概率它可能是一个伪素数(合数)不能令人放心。即我们对这类判定问题不能100%相信，可能会冒风险，因此，是否采用确定性算法更好呢？

-分析：

取 $k=150$ ，概率算法出错概率 $4^{-150} \approx 10^{-100}$ 。

用一个确定性算法花费更多的时间是完全能确定 n 是否为素数，但是长时间计算过程中，硬件错误率可能高于 4^{-150} 。

§ 5.3 矩阵乘法验证

■ 问题

设A, B, C为 $n \times n$ 矩阵, 判定 $AB=C$?通过 $A \cdot B$ 的结果与C比较。

-传统方法 $O(n^3)$

-当n非常大时, 确定型算法的时间 $\Omega(n^{2.37})$

-用MC算法, 可在 $O(n^2)$ 内解此问题, 但要接受一个很小的误差 ϵ 。

■ MC算法

设X是一个长度为n的二值向量(0/1行向量)。

将判断 $AB=C$ 改为判断 $XAB=XC$?

§ 5.3 矩阵乘法验证

- ① 先计算XA $X_{1n}A_{mn} \Rightarrow 1 \times n$ 向量 n^2 次数乘
 - ② 再计算XA与B乘积。 n^2 次数乘
 - ③ 计算XC n^2 次数乘
- 时间: $O(n^2)$

} $3n^2$ 次乘法

```
goodproduct( A, B, C, n){  
    for i←1 to n do  
        x [ i ] ←uniform(0..1);  
    if (XA)B=XC then  
        return true;  
    else return false;  
}
```

§ 5.3 矩阵乘法验证

■ 分析

-例:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{bmatrix} \quad C = \begin{bmatrix} 11 & 29 & 37 \\ 29 & 65 & 91 \\ 47 & 99 & 45 \end{bmatrix}$$

$AB \rightarrow 101 \quad 145$

① 设 $X=(1, 1, 0)$

XA 相当于将 A 的第1行和第2行相加: $XA=(5, 7, 9)$

$(XA)B=(40, 94, 128)$, 相当于是 AB 的第1行+第2行。
(满足结合率)。

XC 相当于将 C 的第1行和第2行相加: $XC=(40, 94, 128)$ 。

算法返回true, **错误!**

§ 5.3 矩阵乘法验证

② 设 $X=(0, 1, 1)$

$XA=(11, 13, 15)$ (A的第2行+第3行)

$(XA)B=(76, 166, 236)$. (AB的第2行+第3行)

$XC=(76, 164, 136)$. (C的第2行+第3行)

∴AB和C的第3行不等, 即 $AB \neq C$

∴算法返回false, 正确!

-考虑两种情况

① 设 $AB=C$, 则无论 X 为何值, 必有 $XAB=XC$

② 设 $AB \neq C$, 若AB与C的第 i 行不同, 且 $X_i=0$ 则出错! 即误判 $AB=C$, 出错概率 $\leq 1/2$; 否则无论 X_i 为何值, 不影响判定结果。

§ 5.3 矩阵乘法验证

偏真还是偏假？

➤ 若算法返回false，则存在向量X使

$$XAB \neq XC \Rightarrow AB \neq C, \text{ 必正确。}$$

➤ 若算法返回true，则

{ 当 $AB = C$ 时，正确

{ 当 $AB \neq C$ 时，错误，发生在AB与C的第i行不等，但 $x_i = 0$ 时

➤ **结论：**偏假的，1/2-correct.

§ 5.3 矩阵乘法验证

- 改进

```
RepeatGoodProduct( A, B, C, n, k) {  
    for i ← 1 to k do //重复k次  
        if GoodProduct(A, B, C, n) = false then  
            return false; //偏假的, 有一次假即可返回  
    return true;  
}
```

此算法是偏假的 $(1-2^{-k})$ -correct的

当 $k=10$, 0.99-正确。

$k=20$, 出错概率 $<1/$ 百万。

§ 5.3 矩阵乘法验证

若给出出错概率 ϵ , 则 $2^{-k} = \epsilon$:

GP(A, B, C, n, ϵ) {

$$k \leftarrow \left\lceil \lg \frac{1}{\epsilon} \right\rceil ;$$

return RepeatGoodProduct (A, B, C, n, k);

}

时间: $O(n^2 \log \frac{1}{\epsilon})$

∵ 计算XAB和XC需要 $3n^2$ 次数字乘, 若 $k=20$, 则共需 $60n^2$ 次数乘。

∴ 当 n 很大时($n \gg 60$), 它远远快于确定性算法。

§ 5.3 矩阵乘法验证

■ 习题

```
PrintPrimes{ //打印1万以内的素数
```

```
    print 2, 3;
```

```
    n ← 5;
```

```
    repeat
```

```
        if RepeatMillRab(n,  $\lfloor \lg n \rfloor$  ) then print n;
```

```
        n ← n+2;
```

```
    until n=10000;
```

```
}
```

与确定性算法相比较，并给出100~10000以内错误的比例。