第7章　概率算法

概率算法

前面所讨论的算法的每一步计算都是确定的，概率算法允许算法在执行过程中可随机地选择下一个计算步骤。

特征：对所求问题的同一实例用同一概率算法求解两次可能得到完全不同的效果所需的时间和结果可能都有很大的差别。

概率算法可分为4类：

数值概率算法：常用于求解数值问题，一般得到一个近似解。

蒙特卡罗算法：用于求解问题的准确解。求得正确解的概率依赖于算法所用的时间。

拉斯维加斯算法：找到的解一定是一个正确解，但不保证一定能找到解。

舍伍德算法：总能求得一个解，且所求解总是正确的。

随机数

随机数在概率算法设计中扮演着十分重要的角色。在现实计算机上无法产生真正的随机数，因此在概率算法中使用的随机数都是一定程度上随机的，即伪随机数。

线性同余法是产生伪随机数的最常用的方法。由线性同余法产生的随机序列$a_0,a_1,\ldots,a_n$满足

$$\begin{cases} a_0 = d \\ a_n = (ba_{n-1} + c) \bmod m \qquad n = 1,2,\cdots \end{cases}$$

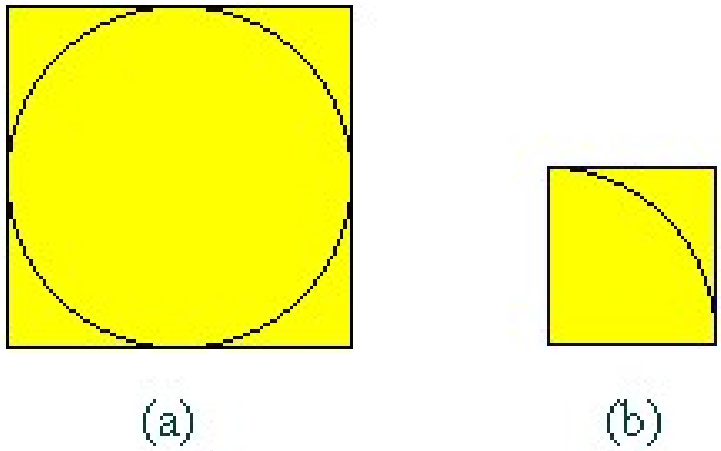
其中b≥0，c≥0，d≤m。d称为该随机序列的种子。如何选取该方法中的常数b、c和m直接关系到所产生的随机序列的随机性能。

数值概率算法

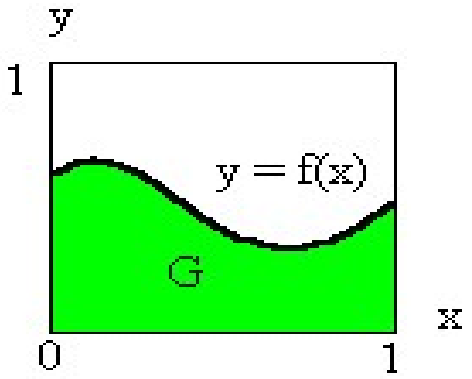
用随机投点法计算π值

设有一半径为r的圆及其外切四边形。向该正方形随机地投掷n个点。设落入圆内的点数为k。由于所投入的点在正方形上均匀分布，因而所投入的点落入圆内的概率为 $\frac{\pi r^2}{4r^2}=\frac{\pi}{4}$ 。所以当n足够大时，k与n之比就逼近这一概率。

```
public static double darts(int n)
   { // 用随机投点法计算π值
     int k=0;
     for (int i=1;i <=n;i++) {
       double x=dart.fRandom();
       double y=dart.fRandom();
       if ((x*x+y*y)<=1) k++;
       }
     return 4*k/(double)n;
   }
```



计算定积分



设f(x)是[0，1]上的连续函数，且0≤f(x)≤1。

需要计算的积分为$I=\int_0^1 f(x)dx$,积分I等于图中的面积G。

在图所示单位正方形内均匀地作投点试验，则随机点落在曲线下面的概率为

$$P_r\{y \le f(x)\} = \int_0^1 \int_0^{f(x)} dydx = \int_0^1 f(x)dx$$

假设向单位正方形内随机地投入n个点(xi,yi)。如果有m个点落入G内，则随机点落入G内的概率

$$\mathrm{I} \approx \frac{m}{n}$$

解非线性方程组

求解下面的非线性方程组

$$\begin{cases} f_1(x_1, x_2, \cdots, x_n) = 0 \\ f_2(x_1, x_2, \cdots, x_n) = 0 \\ \cdots\cdots\cdots\cdots\cdots\cdots \\ f_n(x_1, x_2, \cdots, x_n) = 0 \end{cases}$$

其中，$x_1, x_2, \ldots, x_n$是实变量，f_i是未知量$x_i \ldots, x_n$的非线性实函数。要求确定在指定求根范围内的一组解 $x_1^*, x_2^*, \cdots, x_n^*$

为求解给定的非线性方程组，构造函数：

$$\Phi(X) = \sum_{i=1}^{n} f_i^2(X)$$

其中：$X = (x_1, x_2, \cdots, x_n)$

该函数$\Phi(X)$的零点即为方程组的一组解。

解非线性方程组

求Φ(X)=0的解时，可在指定求根区域内选择一个X_0作为初值，按预先选定的分布逐个选取随即点X，并计算Φ(X)的值，把满足精度要求的X作为方程组的近似解。

这种方法简单直观，但计算工作量较大。

下面介绍一种随机搜索算法：

在指定求根区域D内，选定一个随机点X_0作为随机搜索的出发点。在算法的搜索过程中，假设第j步随机搜索得到的随机搜索点为X_j。在第j+1步，计算出下一步的随机搜索增量ΔX_j。从当前点X_j依ΔX_j得到第j+1步的随机搜索点。当Φ(x)<ε时，取为所求非线性方程组的近似解。否则进行下一步新的随机搜索过程。

舍伍德(Sherwood)算法

设A是一个确定性算法，当它的输入实例为x时所需的计算时间记为$t_A(x)$。设X_n是算法A的输入规模为n的实例的全体，则当问题的输入规模为n时，算法A所需的平均时间为

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

这显然不能排除存在$x \in X_n$使得 $t_A(x) >> \bar{t}_A(n)$ 的可能性。希望获得一个概率算法B，使得对问题的输入规模为n的每一个实例均有

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

这就是舍伍德算法设计的基本思想。当$s(n)$与$t_{A(n)}$相比可忽略时，舍伍德算法可获得很好的平均性能。

舍伍德(Sherwood)算法

复习学过的Sherwood算法：

（1）线性时间选择算法

（2）快速排序算法

```
int Select(a[],k) {
    int L=0, R=a.length-1;
    while (L<R){
        int j=random(R-1)+1;
        swap(a, L, j);
        j=partition(a, L, R);
        if (j==k) return a[j];
        if (j<k) L=j+1;
        else R=j-1;
    }
}
```

舍伍德(Sherwood)算法

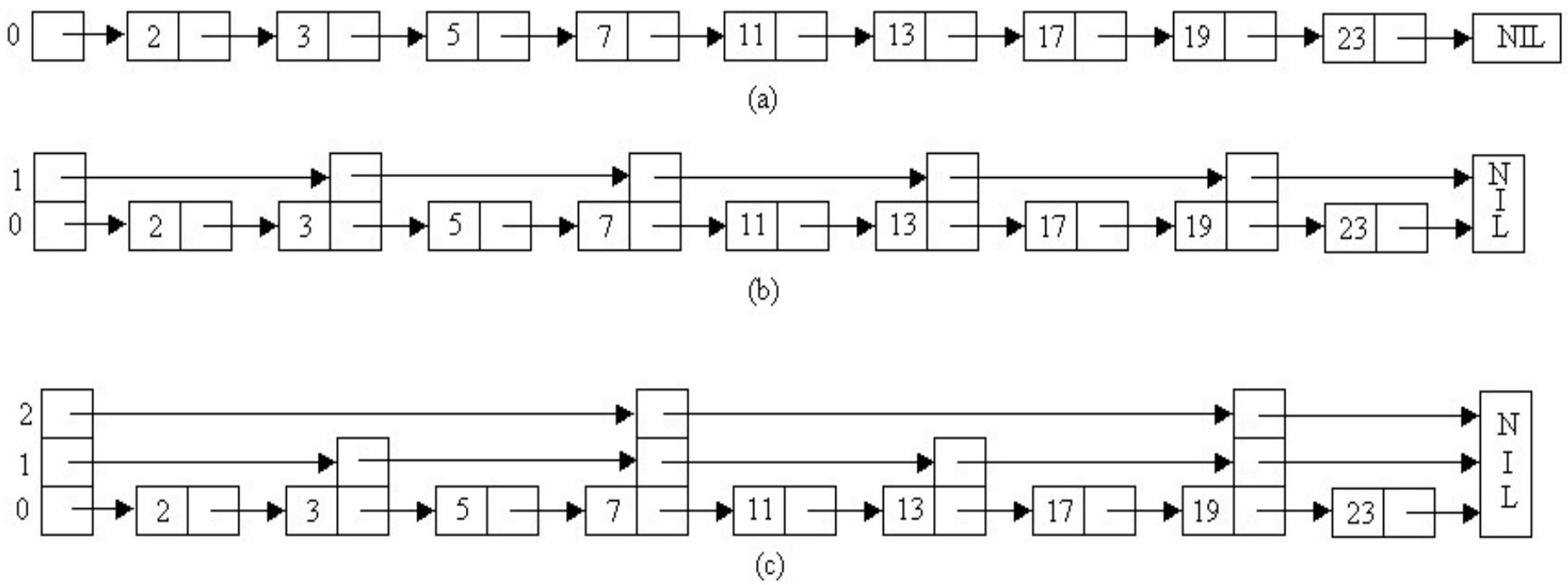
有时也会遇到这样的情况，即所给的确定性算法无法直接改造成舍伍德型算法。此时可借助于随机预处理技术，不改变原有的确定性算法，仅对其输入进行随机洗牌，同样可收到舍伍德算法的效果。例如，对于确定性选择算法，可以用下面的洗牌算法**shuffle**将数组a中元素随机排列，然后用确定性选择算法求解。

```
public static void shuffle(Comparable []a, int n)
  {// 随机洗牌算法
    rnd = new Random();
    for (int i=1;i<n;i++) {
      int j=rnd.random(n-i+1)+i;
      MyMath.swap(a, i, j);
      }
}
```

跳跃表

- 舍伍德型算法的设计思想还可用于设计高效的数据结构。
- 如果用有序链表来表示一个含有n个元素的有序集S，则在最坏情况下，搜索S中一个元素需要Ω(n)计算时间。
- 提高有序链表效率的一个技巧是在有序链表的部分结点处增设附加指针以提高其搜索性能。在增设附加指针的有序链表中搜索一个元素时，可借助于附加指针跳过链表中若干结点，加快搜索速度。这种增加了向前附加指针的有序链表称为**跳跃表**。
- 应在跳跃表的哪些结点增加附加指针以及在该结点处应增加多少指针完全采用随机化方法来确定。这使得跳跃表可在O(logn)平均时间内支持关于有序集的搜索、插入和删除等运算。

跳跃表


0
2
3
5
7
11
13
17
19
23
NIL
(a)
1
0
2
3
5
7
11
13
17
19
23
NIL
(b)
2
1
0
2
3
5
7
11
13
17
19
23
NIL
(c)

跳跃表

在一般情况下，给定一个含有n个元素的有序链表，可以将它改造成一个完全跳跃表，使得每一个k级结点含有k+1个指针，分别跳过2^k-1，$2^{k-1}-1$，…，2^0-1个中间结点。这样就可以在时间O(logn)内完成集合成员的搜索运算。在一个完全跳跃表中，最高级的结点是$\lceil \log n \rceil$级结点。

完全跳跃表与完全二叉搜索树的情形非常类似。它虽然可以有效地支持成员搜索运算，但不适应于集合动态变化的情况。集合元素的插入和删除运算会破坏完全跳跃表原有的平衡状态，影响后继元素搜索的效率。

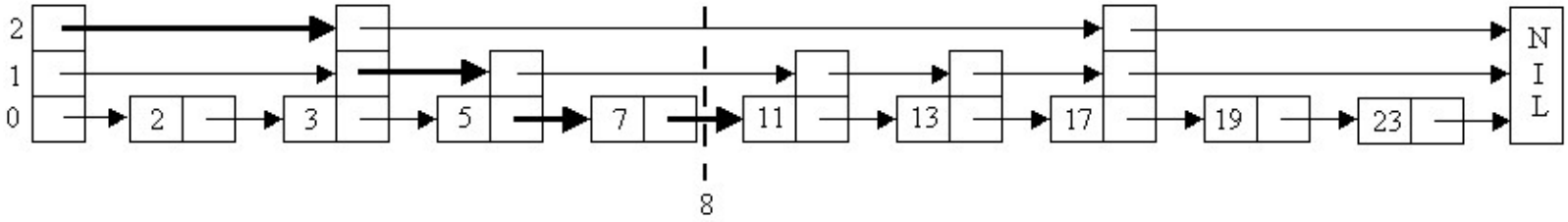
跳跃表

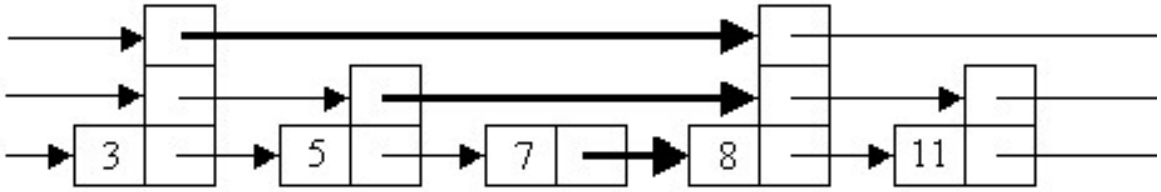
为了在动态变化中维持跳跃表中附加指针的平衡性，必须使跳跃表中k级结点数维持在总结点数的一定比例范围内。注意到在一个完全跳跃表中，50%的指针是0级指针；25%的指针是1级指针；...；$(100/2^{k+1})\%$的指针是k级指针。

因此，在插入一个元素时，以概率1/2引入一个0级结点，以概率1/4引入一个1级结点，...，以概率$1/2^{k+1}$引入一个k级结点。

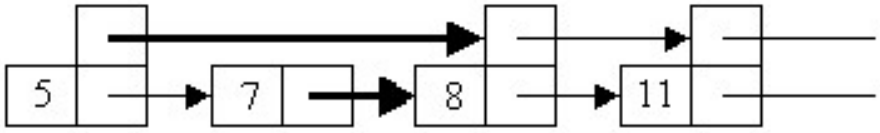
另一方面，一个i级结点指向下一个同级或更高级的结点，它所跳过的结点数不再准确地维持在2^i-1。经过这样的修改，就可以在插入或删除一个元素时，通过对跳跃表的局部修改来维持其平衡性。

跳跃表





(a)



(b)

跳跃表

注意到，在一个完全跳跃表中，具有i级指针的结点中有一半同时具有i+1级指针。为了维持跳跃表的平衡性，可以事先确定一个实数0<p<1，并要求在跳跃表中维持在具有i级指针的结点中同时具有i+1级指针的结点所占比例约为p。为此目的，在插入一个新结点时，先将其结点级别初始化为0，然后用随机数生成器反复地产生一个[0，1]间的随机实数q。如果q<p，则使新结点级别增加1，直至q≥p。由此产生新结点级别的过程可知，所产生的新结点的级别为0的概率为1-p，级别为1的概率为p(1-p)，…，级别为i的概率为$p^i(1-p)$。如此产生的新结点的级别有可能是一个很大的数，甚至远远超过表中元素的个数。为了避免这种情况，用 $\log_{1/p} n$ 作为新结点级别的上界。其中n是当前跳跃表中结点个数。当前跳跃表中任一结点的级别不超过 $\log_{1/p} n$

拉斯维加斯(Las Vegas)算法

拉斯维加斯算法的一个显著特征是它所作的随机性决策有可能导致算法找不到所需的解。

```
public static void obstinate(Object x, Object y)
  {// 反复调用拉斯维加斯算法LV(x,y)，直到找到问题的一个解y
     boolean success= false;
     while (!success) success=lv(x,y);
  }
```

拉斯维加斯(Las Vegas)算法

设p(x)是对输入x调用拉斯维加斯算法获得问题的一个解的概率。一个正确的拉斯维加斯算法应该对所有输入x均有p(x)>0。

设t(x)是算法**obstinate**找到具体实例x的一个解所需的平均时间 ,s(x)和e(x)分别是算法对于具体实例x求解成功或求解失败所需的平均时间，则有：

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

解此方程可得：

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

n后问题

对于n后问题的任何一个解而言，每一个皇后在棋盘上的位置无任何规律，不具有系统性，而更象是随机放置的。由此容易想到下面的**拉斯维加斯算法**。

在棋盘上相继的各行中随机地放置皇后，并注意使新放置的皇后与已放置的皇后互不攻击，直至n个皇后均已相容地放置好，或已没有下一个皇后的可放置位置时为止。

算法见P260

n后问题

如果将上述随机放置策略与回溯法相结合，可能会获得更好的效果。可以先在棋盘的若干行中随机地放置皇后，然后在后继行中用回溯法继续放置，直至找到一个解或宣告失败。随机放置的皇后越多，后继回溯搜索所需的时间就越少，但失败的概率也就越大。

stopVegas	p	s	e	t
0	1.0000	262.00	--	262.00
5	0.5039	33.88	47.23	80.39
12	0.0465	13.00	10.20	222.11

整数因子分解

设n>1是一个整数。关于整数n的因子分解问题是找出n的如下形式的惟一分解式： $n = p_1^{m_1} p_2^{m_2} \cdots p_k^{m_k}$

其中，$p_1<p_2<\ldots<p_k$是k个素数，$m_1,m_2,\ldots,m_k$是k个正整数。

如果n是一个合数，则n必有一个非平凡因子x，1<x<n，使得x可以整除n。给定一个合数n，求n的一个非平凡因子的问题称为整数n的因子分割问题。

```
private static int split(int n)
  {
    int m = (int) Math.floor(Math.sqrt((double)n));
    for (int i=2; i<=m; i++)
    if (n%i==0) return i;
    return 1;
  }
```

Pollard算法

在开始时选取0～n-1范围内的随机数，然后递归地由 $x_i = (x_{i-1}^2 - 1) \bmod n$

产生无穷序列 $x_1, x_2, \ldots, x_k, \ldots$

对于$i=2^k$，以及$2^k<j\le 2^{k+1}$，算法计算出x_j-x_i与n的最大公因子$d=\gcd(x_j-x_i, n)$。如果d是n的非平凡因子，则实现对n的一次分割，算法输出n的因子d。

Pollard算法

```
private static void pollard(int n)
  {// 求整数n因子分割的拉斯维加斯算法
     rnd = new Random();  // 初始化随机数
     int i=1,k=2;
     int x=rnd.random(n),y=x; // 随机整数
     while (true) {
       i++;              x=(x*x-1)%n;
       int d=gcd(y-x,n);  // 求n的非平凡因子
       if ((d>1) && (d<n)) System.out.println(d);
       if (i==k) {  y=x;   k*=2;}
     }
  }
```

对Pollard算法更深入的分析可知，执行算法的while循环约$\sqrt{p}$次后，Pollard算法会输出n的一个因子p。由于n的最小素因子$p \le \sqrt{n}$，故Pollard算法可在$O(n^{1/4})$时间内找到n的一个素因子。

蒙特卡罗(Monte Carlo)算法

- 在实际应用中常会遇到一些问题，不论采用确定性算法或概率算法都无法保证每次都能得到正确的解答。蒙特卡罗算法则在一般情况下可以保证对问题的所有实例都以高概率给出正确解，但是通常无法判定一个具体解是否正确。

- 设p是一个实数，且1/2<p<1。如果一个蒙特卡罗算法对于问题的任一实例得到正确解的概率不小于p，则称该蒙特卡罗算法是**p正确**的，且称p-1/2是该算法的**优势**。

蒙特卡罗(Monte Carlo)算法

- 如果对于同一实例，蒙特卡罗算法不会给出2个不同的正确解答，则称该蒙特卡罗算法是**一致的**。

- 有些蒙特卡罗算法除了具有描述问题实例的输入参数外，还具有描述错误解可接受概率的参数。这类算法的计算时间复杂性通常由问题的实例规模以及错误解可接受概率的函数来描述。

蒙特卡罗(Monte Carlo)算法

对于一个一致的p正确蒙特卡罗算法，要提高获得正确解的概率，只要执行该算法若干次，并选择出现频次最高的解即可。如果重复调用一个一致的(1/2+ε)正确的蒙特卡罗算法2m-1次，得到正确解的概率至少为1-δ，其中，

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

蒙特卡罗(Monte Carlo)算法

对于一个解所给问题的蒙特卡罗算法MC(x)，如果存在问题实例的子集X使得：

(1)当$x \notin X$时，MC(x)返回的解是正确的；

(2)当$x \in X$时，正确解是y_0，但MC(x)返回的解未必是y_0。

称上述算法MC(x)是偏y_0的算法。

重复调用一个一致的，p正确偏y_0蒙特卡罗算法k次，可得到一个$O(1-(1-p)^k)$正确的蒙特卡罗算法，且所得算法仍是一个一致的偏y_0蒙特卡罗算法。

主元素问题

设T[1:n]是一个含有n个元素的数组。当|{i|T[i]=x}|>n/2时，称元素x是数组T的主元素。

```
public static boolean majority(int[]t, int n)
  {// 判定主元
     rnd = ne
     int i=rnd
     int x=t[i]
     int k=0;
     for (int j=
       if (t[j]==
     return (k
  }
```

对于任何给定的ε>0，算法**majorityMC**重复调用⌈log(1/ε)⌉次算法**majority**。它是一个偏真蒙特卡罗算法，且其错误概率小于ε。算法**majorityMC**所需的计算时间显然是O(nlog(1/ ε))。

```
public static boolean majorityMC(int[]t, int n, double e)
  {  int k= (int) Math.ceil(Math.log(1/e)/Math.log(2));
     for (int i=1;i<=k;i++)
       if (majority(t,n)) return true;
     return false;
  }
```

素数测试

Wilson定理：对于给定的正整数n，判定n是一个素数的充要条件是$(n-1)!\equiv -1 \pmod n$。

费尔马小定理：如果p是一个素数，且0<a<p，则$a^{p-1} \pmod p = 1$。

二次探测定理：如果p是一个素数，且0<x<p，则方程$x^2 \equiv 1 \pmod p$的解为x=1，p-1。

素数测试

```
private static int power(int a, int p, int n)
 {// 计算 a^p mod n，并实施对n的二次探测
        int x, result;
        if (p==0) result=1;
        else {
              x=power(a,p/2,n);  // 递归计算
              result=(x*x)%n;          // 二次探测
              if ((result==1)&&(x!=1)&&(x!=n-1))
                    composite=true;
              if ((p%2)==1)     // p是奇数
                    result=(result*a)%n;
         }
      return result;
}
```

素数测试

```
public static boolean prime(int n)
    {// 素数测试的蒙特卡罗算法
        rnd = new Rand
        int a, result;
        composite=false
        a=rnd.random(n
        result=power(a,r
        if (composite||(result!=1))
              return false;
        else return true;
}
```

算法**prime**是一个偏假3/4正确的蒙特卡罗算法。通过多次重复调用错误概率不超过$(1/4)^k$。这是一个很保守的估计，实际使用的效果要好得多。