USTC

Chapter 8
Stochastic Methods

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn/

学习要点

- 理解产生伪随机数的算法
- 掌握数值随机化算法的设计思想
- 掌握蒙特卡罗算法的设计思想
- 掌握拉斯维加斯算法的设计思想
- 掌握舍伍德算法的设计思想

随机数

随机数

- 在随机化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
  - 真正的物理随机数
- 在随机化算法中使用的随机数都是一定程度上随机的，即伪随机数
  - 更好的随机数
  - 服从特定分布的随机数

随机数

线性同余法

- 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

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

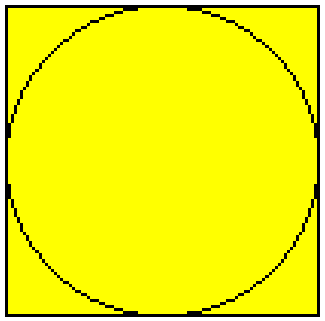
- 其中 $b \geq 0$，$c \geq 0$，$d \leq m$，d 称为该随机序列的种子
- 如何选取该方法中的常数 b、c 和 m 直接关系到所产生的随机序列的随机性能
  - 这是随机性理论研究的内容，已超出本书讨论的范围
- 从直观上看，m 应取得充分大，因此可取 m 为机器大数，另外应取 $\gcd(m,b)=1$，因此可取 b 为一素数

用随机投点法计算π值

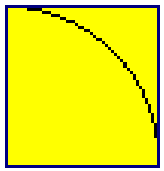
- 设有一半径为 r 的圆及其外切四边形，向该正方形随机地投掷 n 个点，设落入圆内的点数为 k。由于所投入的点在正方形上**均匀分布**，因而所投入的点落入圆内的概率为 $\frac{\pi r^2}{4r^2}=\frac{\pi}{4}$。所以，当 n 足够大时，k 与 n 之比就**逼近**这一概率，从而：

$$\pi \approx \frac{4k}{n}$$

```
1.  double Darts(int n)
2.  {   // 用随机投点法计算π值
3.      static RandomNumber dart;
4.      int k=0;
5.      for (int i=1;i <=n;i++) {
6.          double x=dart.fRandom();
7.          double y=dart.fRandom();
8.          if ((x*x+y*y)<=1) k++;
9.      }
10.     return 4*k/double(n);
11. }
```



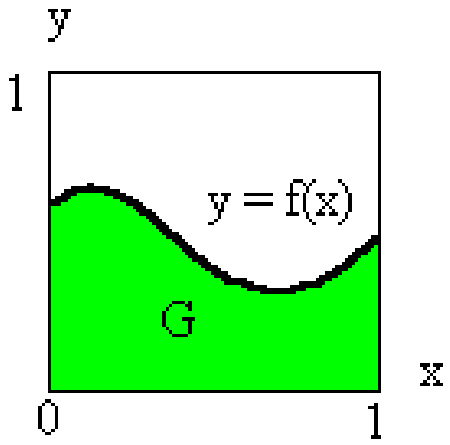
(a)



(b)

计算定积分

- 设 $f(x)$ 是 $[0, 1]$ 上的连续函数，且 $0 \le f(x) \le 1$
- 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 (x_i, y_i)，如果有 m 个点落入 G 内，则随机点落入 G 内的概率为：

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

解非线性方程组

求解下面的非线性方程组

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

其中，$x_1, x_2, \ldots, x_n$是实变量，f_i是未知量$x_1, x_2, \ldots, x_n$的非线性实函数

要求确定上述方程组在指定求根范围内的一组解？

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$\Phi(x)<\varepsilon$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

❖ 舍伍德算法设计的基本思想

- 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，希望获得一个随机化算法B，使得对问题的输入规模为 n 的每一个实例均有：

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

- 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

- 学习过的 Sherwood 算法：
  - （1）线性时间选择算法
  - （2）快速排序算法

  - 随机选取关键比较点 (pivot)

舍伍德（Sherwood）算法

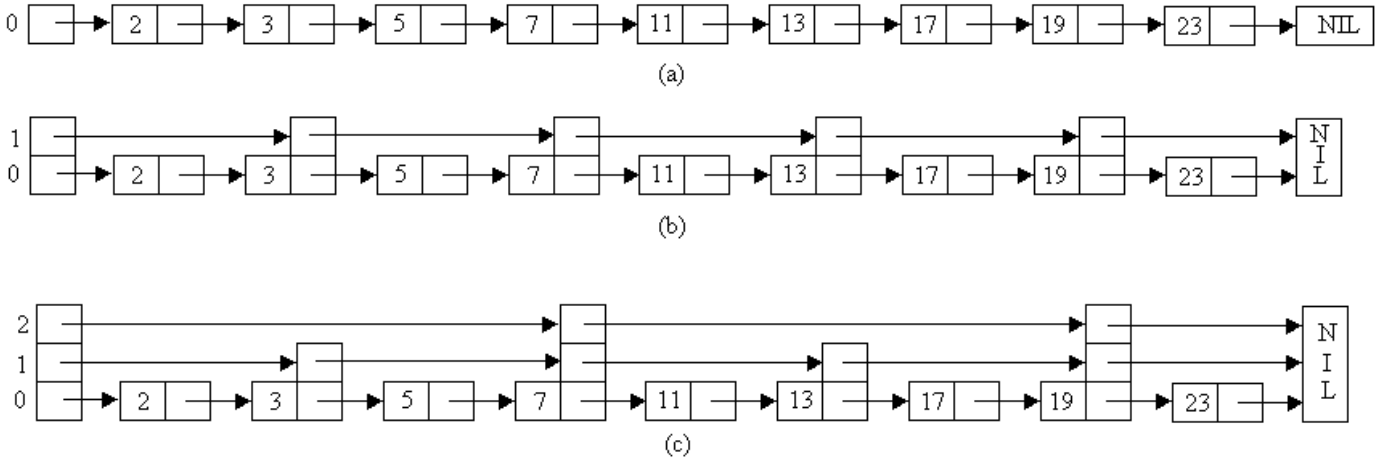
- 有时也会遇到这样的情况
  - 所给的确定性算法无法直接改造成舍伍德型算法
  - 此时借助于随机预处理技术，不改变原有的确定性算法，仅对其输入进行随机洗牌，同样可收到舍伍德算法的效果
- 例如
  - 对于确定性选择算法，可以用下面的洗牌算法 shuffle 将数组 a 中元素随机排列，然后用确定性选择算法求解

```
1.  template<class Type>
2.  void Shuffle(Type a[], int n)
3.  {    // 随机洗牌算法
4.       static RandomNumber rnd;
5.       for (int i=0;i<n;i++) {
6.            int j=rnd.Random(n-i)+i;
7.            Swap(a[i], a[j]);
8.       }
9.  }
```

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 $\Omega(n)$ 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

跳跃表

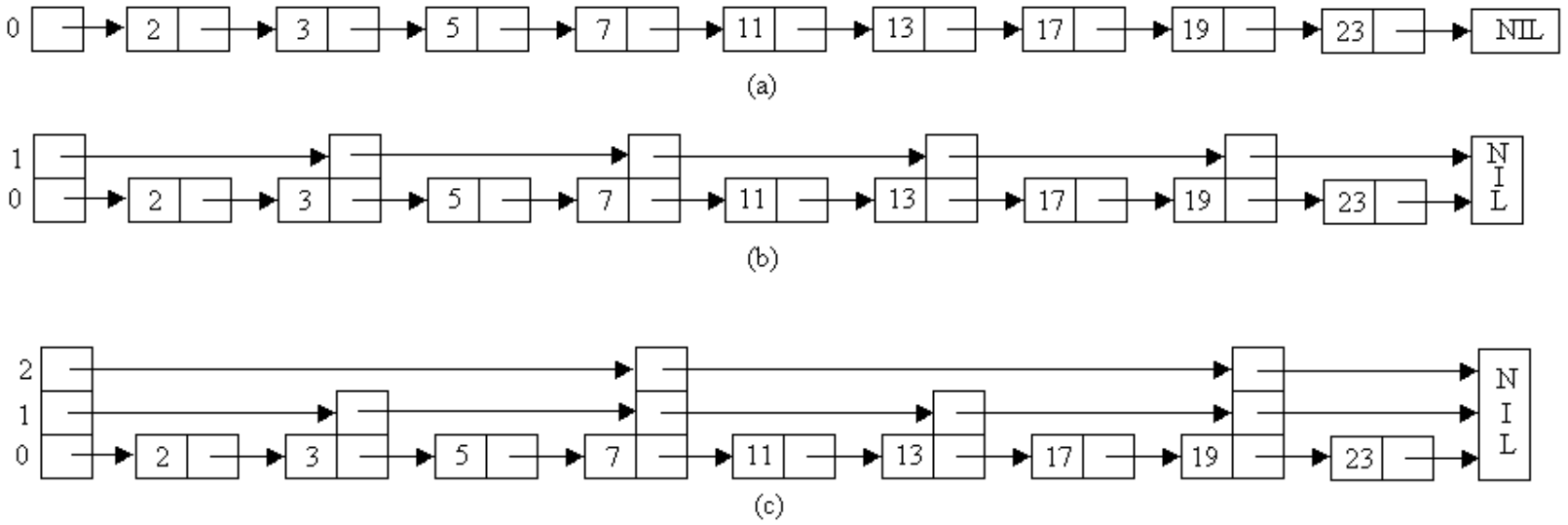


- 问题
  - 应在跳跃表的哪些结点增加附加指针？
  - 在该结点处应增加多少指针？
- 完全采用随机化方法来确定
  - 这使得跳跃表可在 $O(\log n)$ 平均时间内支持关于有序集的搜索、插入和删除等运算

跳跃表

- 跳跃表构造
  - 在一般情况下，给定一个含有 n 个元素的有序链表
  - 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
  - 第 i 个 k 级结点安排在跳跃表的位置 i^{2k} 处($i\geq 0$)，这样就可以在时间 $O(\log n)$ 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

跳跃表



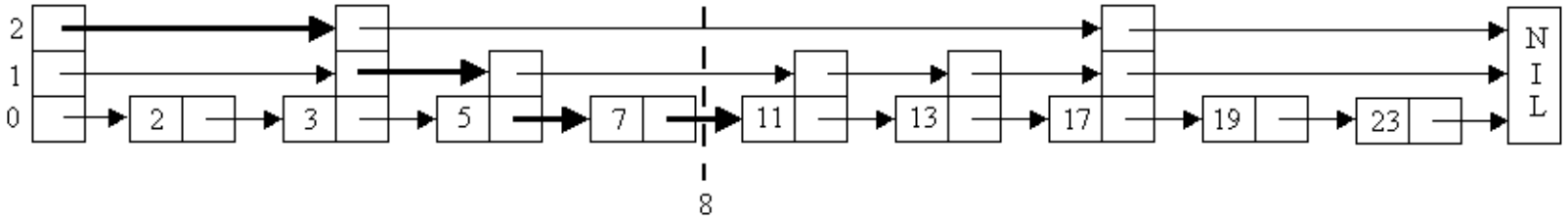
- 完全跳跃表与完全二叉搜索树的情形非常类似
  - 它可以有效地支持成员搜索运算，但不适应于集合动态变化的情况
  - 集合元素的插入和删除运算会破坏完全跳跃表原有的平衡状态，影响后继元素搜索的效率

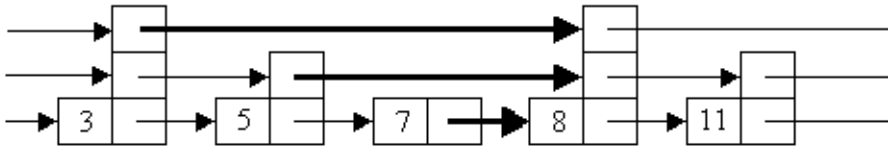
跳跃表

- 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
  - 在插入一个元素时，以概率 ½ 引入一个 0 级结点，以概率 ¼ 引入一个 1 级结点，…，以概率 $1/2^{k+1}$ 引入一个 k级结点
  - 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

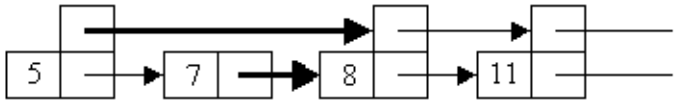
跳跃表

- 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





(a)



(b)

跳跃表

- 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 $0<p<1$
  - 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
- 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 $q\geq p$
  - 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，即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
  - 它所作的随机性决策有可能导致算法找不到所需的解
  - 但能够判断是否找到所需的解

```
1.  void obstinate(Object x, Object y)
2.  {    // 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
3.       bool success= false;
4.       while (!success) success=lv(x,y);
5.  }
```

- 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 $p(x)>0$

拉斯维加斯（Las Vegas）算法

时间分析

- 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，每一个皇后在棋盘上的位置无任何规律，不具有系统性，而更像是随机放置的，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

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**唯一**分解式：$n = p_1^{m_1} p_2^{m_2} \Lambda\ p_k^{m_k}$
  - 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 的一个非平凡因子问题称为**整数 n 的因子分割问题**

```
1.  int Split(int n)
2.  {
3.      int m = floor(sqrt(double(n)));
4.      for (int i=2; i<=m; i++)
5.          if (n%i==0) return i;
6.      return 1;
7.  }
```

事实上，算法 **split(*n*)** 是对范围在 $1\sim x$ 的所有整数进行了试除而得到范围在 $1\sim x^2$ 的任一整数的因子分割

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 $i=2^k$，及 $2^k<j\le 2^{k+1}$，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

```
1.  void Pollard(int n)
2.  {  // 求整数n因子分割的拉斯维加斯算法
3.     RandomNumber rnd;
4.     int i=1;
5.     int x=rnd.Random(n);  // 随机整数
6.     int y=x;
7.     int k=2;
8.     while (true) {
9.        i++;
10.       x=(x*x-1)%n;   //
11.       int d=gcd(y-x,n); // 求n的非平凡因子
12.       if ((d>1) && (d<n)) cout<<d<<endl;
13.       if (i==k) {
14.          y=x;
15.          k*=2;
16. } } }
```

对 Pollard 算法更深入的分析可知，执行算法的 while 循环约 $\sqrt{n}$ 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

- 问题背景
  - 在实际应用中常会遇到一些问题，不论采用确定性算法或随机化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
- 蒙特卡罗算法是 p 正确的
  - 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
  - 称 $p-1/2$ 是该算法的优势
  - 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

蒙特卡罗（Monte Carlo）算法

- 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
- 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 $(1/2+\varepsilon)$ 正确的蒙特卡罗算法 n 次，得到正确解的概率至少为 $1-\delta$，其中：

$$\varepsilon + \delta < \frac{1}{2}, n = -2\frac{\log(1/\delta)}{\log(1-4\varepsilon^2)}, p = 1/2 + \varepsilon, q = 1 - p = 1/2 - \varepsilon$$

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

```
1.  template<class Type>
2.  bool Majority(Type *T, int n)
3.  {   // 判定主元素的蒙特卡罗算法
4.      int i=rnd.Random(n)+1;
5.      Type x=T[i];   // 随机选择数组元素
6.      int k=0;
7.      for (int j=1;j<=n;j++)
8.          if (T[j]==x) k++;
9.      return (k>n/2);  // k>n/2 时T含有主元素
10. }
```

对于任何给定的 $\varepsilon>0$，算法 **majorityMC** 重复调用 $\lceil\log(1/\varepsilon)\rceil$ 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 $O(n\log(1/\varepsilon))$

```
1.  template<class Type>
2.  bool MajorityMC(Type *T, int n, double e)
3.  {   // 重复调用算法Majority
4.      int k=ceil(log(1/e)/log(2));
5.      for (int i=1;i<=k;i++)
6.          if (Majority(T,n)) return true;
7.      return false;
8.  }
```

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 $a^{p-1}=1 \pmod{p}$

必要条件

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 $x=1$，$p-1$

```
1.  void power( unsigned int a, unsigned int p,
2.          unsigned int n,  unsigned int &result,
3.          bool &composite)
4.  { // 计算mod n，并实施对n的二次探测
5.      unsigned int x;
6.      if (p==0) result=1;
7.      else {
8.          power(a,p/2,n,x,composite); // 递归计算
9.          result=(x*x)%n;          // 二次探测
10.         if ((result==1)&&(x!=1)&&(x!=n-1))
11.             composite=true;
12.         if ((p%2)==1)     // p是奇数
13.             result=(result*a)%n;
14.     }
15. }
```

```
1.  bool Prime(unsigned int n)
2.  {  // 素数测试的蒙特卡罗算法
3.      RandomNumber rnd;
4.      unsigned int a, result;
5.      bool composite=false;
6.      a=rnd.Random(n-3)+2;
7.      power(a,n-1,n,result,composite);
8.      if (composite || (result!=1)) return false;
9.      else return true;
10. }
```

算法 **prime** 是一个偏假¾正确的蒙特卡罗算法。通过多次重复调用错误概率不超过 $(1/4)^k$，这是一个保守的估计，实际使用的效果要好得多

Next

- 线性规划
  - Linear programming
  - Simplex algorithm