算法设计与分析

随机算法及实例分析

主要内容

- 随机算法的基本思想
- Las Vegas算法
- Monte Carlo算法
- 随机算法实例分析

随机数

- 随机序列
  - 概率相等(均匀随机)
  - 不可预测
  - 不可重现
- 随机数
  - 真随机数
  - 伪随机数
- 在目前的计算机中
  - 要产生真正的随机数是相当困难的，因为计算机是基于确定性的算法工作的，无法产生真正的随机数。
  - 因此在随机算法中使用的随机数都是一定程度上随机的，即伪随机数。

产生伪随机数的方法

- 线性同余法
- 进位乘数法
- 梅森旋转算法
- Box-Muller转换法

产生伪随机数的方法

- 线性同余法（Linear congruential generator）
  - $X_{n+1} = (a*X_n+c) \bmod m$
  - m>0，模数
  - 0<a<m，乘法因子
  - 0=<c<m，增量
  - 0=<X_0<m，种子，或初始值
- 工作原理
  - 通过递推公式不断生成新的伪随机数

产生伪随机数的方法

- 进位乘数法（multiply-with-carry）
  - George Marsaglia提出，和线性同余法类似
  - 可生成周期很长的随机整数序列，$2^{60} \sim 2^{2000000}$
  - 优点在于调用的是简单的计算机整数运算，可以以极快的速度产生随机数序列
- 工作原理

$$x_n = (ax_{n-1} + c_{n-1}) \bmod b,\ c_n = \left\lfloor \frac{ax_{n-1} + c_{n-1}}{b} \right\rfloor$$

  - a为乘法因子，b为模数，此外还有一个表示进位的变量c。

产生伪随机数的方法

- 梅森旋转算法（Mersenne twister）
  - Makoto Matsumoto和Takuji Nishimura于1997年提出
  - 可以以极快的速度生成高质量的伪随机整数
  - 算法的周期长度总是一个梅森素数
  - 梅森旋转算法具有较长的周期，适用于需要大量随机数的应用
- 工作原理
  - 通过位运算和旋转操作组合，产生新的伪随机数

产生伪随机数的方法

- Box-Muller转换法
  - 可将均匀分布随机数转换为正态分布随机数的一种算法。
  - 优点在于生成的随机数具有良好的统计性质，特别适用于需要正态分布随机数的模拟和实验
- 基本原理
  - 对于独立均匀分布的随机变量$U1$和$U2$，生成独立标准正态分布的随机变量$Z0$和$Z1$

$$Z_0 = R\cos(\Theta) = \sqrt{-2\ln U_1}\cos(2\pi U_2)$$

$$Z_1 = R\sin(\Theta) = \sqrt{-2\ln U_1}\sin(2\pi U_2).$$

确定性算法

- 输入确定
- 则对这个特定输入的每次运行过程是可重复的，运行结果是一样的
- 比如，对于输入<6,4,5,8,9,3>，正确的插入排序算法对这个输入，每次的运行过程是一样的，运行结果也是一样的

随机算法的基本思想

- 引入了随机因素
- 在随机算法中
  - 不要求算法对所有可能的输入均正确计算
  - 只要求出现错误的可能性小到可以忽略的程度
  - 另外也不要求对同一输入，算法每次执行时给出相同的结果

随机算法的特点

- 有不少问题，目前只有效率很差的确定性求解算法，但用随机算法去求解，可以（很快地）获得相当可信的结果

随机算法的应用领域

- 随机算法在分布式计算、通信、信息检索、计算几何、密码学等许多领域都有着广泛的应用
- 最著名的是在公开密钥体系、RSA算法方面的应用

随机算法的种类

- 通常可分为两类
  - Las Vegas算法
  - Monte Carlo算法

Las Vegas算法

- 在少数应用中，可能出现求不出解的情况
- 但一旦找到一个解，这个解一定是正确的
- 在求不出解时，需再次调用算法进行计算，直到获得解为止
- 对于此类算法，主要是分析算法的时间复杂度的期望值，以及调用一次产生失败（求不出解）的概率

Monte Carlo算法

- 通常不能保证计算出来的结果总是正确的，一般只能断定所给解的正确性不小于p（$1/2 < p < 1$）
- 通过算法的反复执行（即以增大算法的执行时间为代价），能够使发生错误的概率小到可以忽略的程度
- 由于每次执行的算法是独立的，故k次执行均发生错误的概率为$(1-p)^k$

Monte Carlo算法

- 对于判定问题（回答只能是“Yes”或“No”）
  - 带双错的（two-sided error）: 回答”Yes”或”No”都有可能错
  - 带单错的（one-sided error）：只有一种回答可能错
- Las Vegas算法可以看成是单错概率为0的Monte Carlo算法

两类随机算法的应用场景

- 使用时，选择哪一类随机算法，到底哪一种随机算法好呢？
  - 依赖于应用
- 在不允许发生错误的应用中（e.g. 电网控制等），Monte Carlo算法不可以使用
- 若小概率的出错允许的话，Monte Carlo算法比Las Vegas算法要节省许多时间，是人们常常采用的方法

随机算法的优点

- 对于某一给定的问题，随机算法所需的时间与空间复杂性，往往比当前已知的、最好的确定性算法要好
- 到目前为止设计出来的各种随机算法，无论是从理解上还是实现上，都是极为简单的
- 随机算法避免了去构造最坏情况的例子

找第k小元素的随机算法（Las Vegas算法）

- 在n个数中随机的找一个数A[i]=x, 然后将其余n-1个数与x比较，分别放入三个数组中：S_1(元素均<x), S_2(元素均=x), S_3(元素均＞x)
- 若$|S_1|\geq k$，则调用Select(k,S_1)
- 若（$|S_1|+|S_2|$）$\geq k$，则第k小元素就是x
- 否则就有（$|S_1|+|S_2|$）＜ k，此时调用Select(k-$|S_1|$-$|S_2|$,S_3)

找第k小元素的随机算法 分析（Las Vegas算法）

- 定理：若以等概率方法在n个数中随机取数，则该算法用到的比较数的期望值不超过4n
- 说明：如果假定n个数互不相同，如果有相同的数的话，落在S_2中的可能性会更大，比较数的期望值会更小一些

Sherwood随机化方法（属Las Vegas算法）

- 如果某个问题已经有了一个平均情况下较好的确定性算法，但是该算法在最坏情况下效率不高，此时引入一个随机数发生器（通常是服从均匀分布，根据问题需要也可以产生其他的分布），可将一个确定性算法改成一个随机算法，使得对于任何输入实例，该算法在概率意义下都有很好的性能
  - Select, Quicksort

Sherwood随机化方法（属Las Vegas算法）

- 如果算法（所给的确定性算法）无法直接使用Sherwood方法，则可以采用随机预处理的方法，使得输入对象服从均匀分布（或其他分布），然后再用确定性算法对其进行处理。所得效果在概率意义下与Sherwood型算法相同

Sherwood随机化方法（属Las Vegas算法）

- Sherwood算法总能求得问题的一个解，且所求得的解是正确的
- 当一个确定性算法在最坏情况和平均情况下的时间复杂度有较大差别时，可在确定性算法中引入随机性将其改造为Sherwood算法，以消除或减少问题的好坏输入实例间的差别

Testing String Equality （Monte Carlo算法）

- 问题描述
- 设A处有一个长字符串x（e.g. 长度为10^6），B处也有一个长字符串y，A将x发给B，由B判断是否有x=y

Testing String Equality（Monte Carlo算法）

- 算法：首先由A发一个x的长度给B，若长度不等，则$x \neq y$
- 若长度相等，则采用“取指纹”的方法：
  - A对x进行处理，取出x的“指纹”，然后将x的“指纹” 发给B
  - 由B检查x的“指纹”是否等于y 的“指纹”
  - 若取k次“指纹”（每次取法不同），每次两者结果均相同，则认为x与y是相等的
  - 随着k的增大，误判率可趋于0

Testing String Equality （Monte Carlo算法）

- 常用的指纹：
- 令I(x)是x的编码，取$Ip(x) \equiv I(x) \pmod{p}$作为x的指纹
- 这里的p是一个小于M的素数，M可根据具体需要调整

Testing String Equality（Monte Carlo算法）

- 错判率分析
- B接到指纹Ip(x)后与Ip(y)比较
  - 如果Ip(x)≠Ip(y)，当然有x≠y
  - 如果Ip(x)=Ip(y)而x≠y，则称此种情况为一个误匹配
- 现在需要确定：误匹配的概率有多大？
  - 若总是随机地去取一个小于M的素数p，则对于给定的x和y，Pr[failure] =（使得Ip(x)=Ip(y)但x≠y的素数p(p＜M)的个数）/（小于M的素数的总个数）

Testing String Equality（Monte Carlo算法）

- 误匹配的概率小于 $1/n$,当n很大时，误匹配的概率很小
- 设$x \neq y$，如果取k个不同的小于$2n^2$的素数来求Ip(x)和Ip(y)
- 即k次试验均有Ip(x)=Ip(y)但$x \neq y$（误匹配）的概率小于 $1/n^k$
- 当n较大、且重复了k次试验时，误匹配的概率趋于0

Pattern Matching
（Monte Carlo算法）

- 问题
- 给定两个字符串：$X=x_1,\ldots,x_n$，$Y=y_1,\ldots,y_m$，看Y是否为X的子串？（即Y是否为X中的一段）
- 可用KMP算法在O(m+n)时间内获得结果，但算法较为繁琐

Pattern Matching
（Monte Carlo算法）

- 考虑随机算法（用brute-force 思想）
- 记$X(j)=x_jx_{j+1}\cdots x_{j+m-1}$（从X的第j位开始、长度与Y一样的子串）
- 从起始位置j=1开始到j=n-m+1，不去逐一比较X(j)与Y，而仅逐一比较X(j)的指纹Ip（X(j)）与Y的指纹Ip（Y）
- 由于Ip（X(j+1)）可以很方便地根据Ip（X(j)）计算出来，故算法可以很快完成

Pattern Matching
（Monte Carlo算法）

1.随机取一个小于M的素数p，置j←1；

2.计算Ip(Y)、Ip(X(1))及$W_p(=2^m \bmod p)$；

3.While j≤n-m+1 do

{if Ip(X(j))=Ip(Y) then return j

/ * X(j)极有可能等于Y * /

else{根据Ip(X(j))计算出Ip(X(j+1))；j增1}

}

4.return 0;　　/ * X肯定没有子串等于Y * /

Pattern Matching
（Monte Carlo算法）

- 时间复杂度分析
- 计算Ip(Y)、Ip(X(1))及$2^m \bmod p$的时间不超过O(m)次运算
- Ip(X(j+1))的计算，只需用O(1)时间
- 由于循环最多执行n-m+1次，故这部分的时间复杂度为O(n)，于是，总的时间复杂性为O(m+n)

Pattern Matching
（Monte Carlo算法）

- 当$Y \neq X(j)$，但$Ip(Y)=Ip(X(j))$时产生失败
- 失败的概率$Pr[failure]<1/n$，即失败的概率只与X的长度有关，与Y的长度无关

Pattern Matching
（Monte Carlo算法）

- 本算法可以转成Las Vegas算法：
- 当Ip(Y)=Ip(X(j))时，不直接return j，而去比较Y和X(j)
- 即在return j之前加一个判断看Y和X(j)是否相等，相等则return j，否则继续执行循环
- 如果有子串X(j)与Y相匹配，该算法总能给出正确的位置（即算法出错的概率为0）

Random Sampling问题

- 问题描述
  - 设给定n个元素（为简单起见，设为1，2，…n）
  - 要求从n个数中随机地选取m个数（m≤n）

Random Sampling问题

- 求解该问题的Las Vegas算法
  - 可以用一个长为n的布尔数组B来标识i是否被选中
  - 初始时均表为“未选中”
  - 然后随机产生〔1，n〕之间的一个整数i，若B[i]为“未选中”，则将i加入被选中队列，同时把B[i]标识为“已选中”
  - 反复执行直到m个不同的数全部被选出为止

Random Sampling问题

- 上述算法存在的问题
  - 当n和m很接近时，产生最后几个随机数的时间可能很长（有95%以上的可能性是已选中的数）
  - 改进方法：当m>n/2时，先去生成(n-m)(<n/2)个随机数，然后再取剩下的m个数作为选出的数

Random Sampling问题

- 上述算法存在的问题
  - 当n与m相比大很多时（例：$n > m^2$），布尔数组B对空间浪费很多
  - 改进
    - 用一个允许冲突的、长为m的散列表，来存放产生的随机数
    - 产生一个数后，看其是否在散列表中：若不在则将其加入，若已在则抛弃该数，再去产生下一个数

Random Sampling问题

- 分析
  - 设随机变量X表示扔硬币时，第一次碰到“正面朝上”的情况时，已经试过的次数
  - X=k意味着前k-1次均向下，第k次向上
  - 概率表示为（p为正面朝上概率，q=1-p为背面朝上概率）
    - $\Pr[X=k]=\begin{cases} pq^{k-1} & k \geq 1 \\ 0 & k < 1 \end{cases}$
    - 前k-1次均向下，第k次向上（几何分布）
    - E(X)=1/p

Random Sampling问题

- 分析
  - 设p_j是这样的概率：在j-1个数已经选出的前提下，当前随机产生的数是以前尚未选过的数的概率
  - $p_j=(n-j+1)/n$
  - $q_j=1-p_j$表示前随机产生的数是以前选过的j-1个数之一的概率
  - $q_j=1-p_j=(j-1)/n$

Random Sampling问题

- 分析
  - 设随机变量X_j（$1 \le j \le m$）表示：在j-1个数已经选出后，再去找出第j个不同的数时，所需要产生的整数个数
  - 则与前述的X类似，X_j服从几何分布，$E(X_j)=1/p_j$
  - 设Y表示从n个数中选出m个数时（$m \le n/2$），所需产生的整数个数的总和的随机变量（目标）
  - 则有$Y=X_1+X_2+X_3+\ldots+X_m$
  - $E(Y) \le 1.69n$

Random Sampling问题

- 分析
  - 求解Random Sampling问题的算法的时间复杂度T(n)正比于算法产生整数的个数总和Y
  - 所以T(n)的期望值与E(Y)成常数比，即有$T(n)=\Theta(n)$（期望值）

估算π值

- 通过随机采样来对实际值进行估算，得到较好的估算结果
- 在正方形和圆形中随机放置一些点，利用圆内点比上正方形内全部点的比例就可以计算出PI值
- 圆形的面积公式为“π乘以半径的平方”，而该圆直径又等于正方形边长
- 随机的点越多，π的估计值就越精确

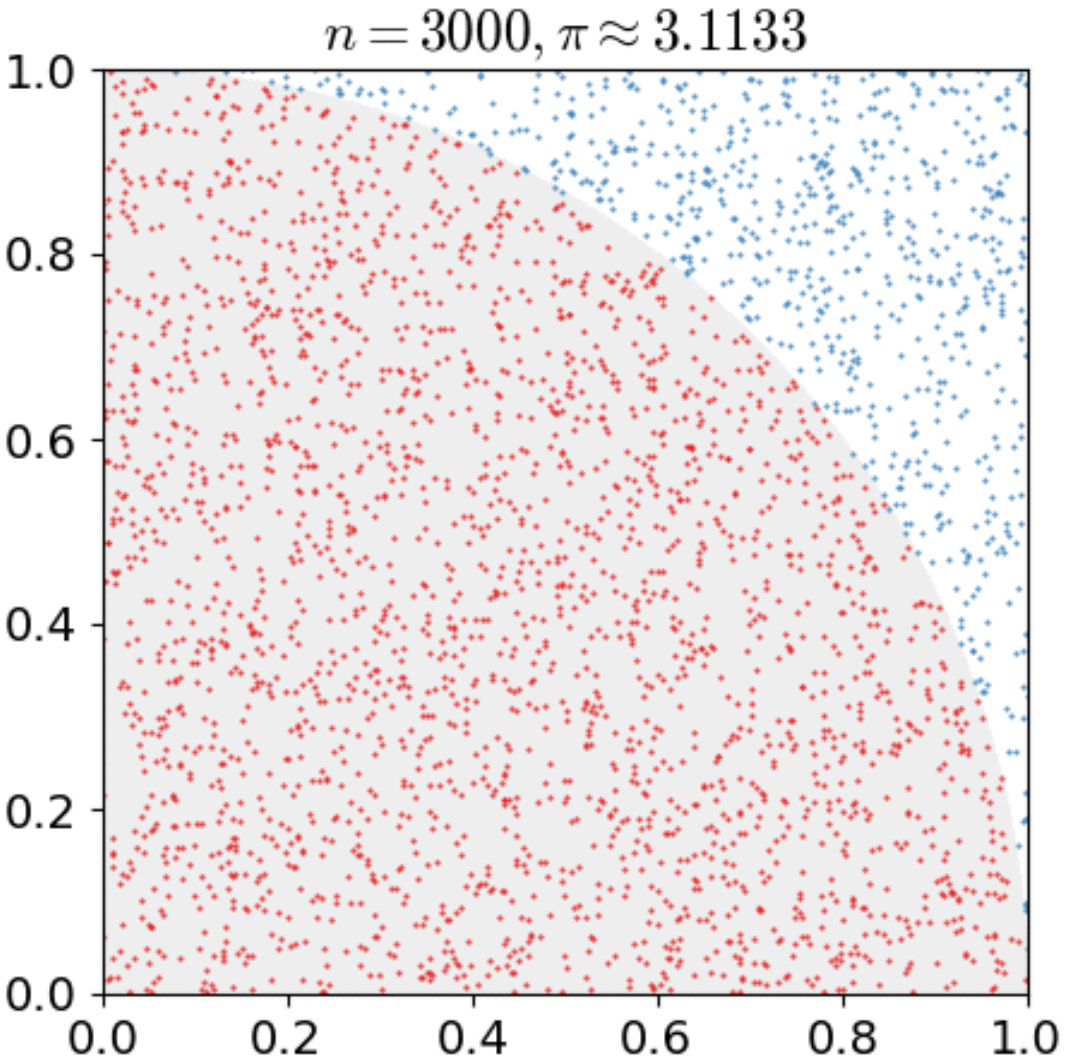
估算π值

- 基本步骤
  - 在单位正方形内生成随机点。
  - 确定有多少点落在单位圆内。
  - 利用比例估算π值。

估算π值

- tries = 0
- success = 0
- for i in from 0 to 1000{
- 	//随机取点
- 	x = rand()
- 	y = rand()
- 	tries = tries +1
- 	if(x*x+y*y<=1){
- 		success++
- 	}
- }
- pi = 4*success / tries

估算π值



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
- 问题描述
  - 对于给定的数组T，判定T数组中是否含有主元素

主元素问题

- 判定是否有主元素的Monte Carlo算法

```
public static boolean majority(int[]t, int n)
  {// 判定主元素的蒙特卡罗算法
     rnd = new Random();
     int i=rnd.random(n)+1;
     int x=t[i];    // 随机选择数组元素
     int k=0;
     for (int j=1;j<=n;j++)
       if (t[j]==x) k++;
     return (k>n/2);  // k>n/2 时t含有主元素
  }
```

主元素问题

- 多次重复调用

```
public static boolean majorityMC(int[]t, int n, double e)
  {// 重复多次调用算法majority
     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;
  }
```

- 计算时间和调用的次数相关

素数测试问题

- 问题描述，判断一个数是否为素数
- 费尔马（ Fermat ）小定理
  - 若n为素数，且0<a<n，有$a^{n-1}\equiv 1 \pmod n$
  - 即Fermat条件$a^{n-1}\equiv 1 \pmod n$是素数的必要条件
  - 反过来说，若$a^{n-1}\not\equiv 1 \pmod n$则n必为合数
  - 逆定理不成立，但反例不多，特别是当n很大且又是随机选取的时候

素数测试问题

- 素数判定方法
  - 直接用Fermat条件$2^{n-1}\equiv 1 \pmod n$来判断n是否为素数
  - 此时，若算法的回答是“合数”，则100%正确
  - 若算法的回答是“素数”，则出错概率很小（带单错）
  - 当n不太大时，满足条件$2^{n-1}\equiv 1 \pmod n$的合数n不多

素数测试问题

- 素数判定方法
  - 能否对其它的a去测试是否有$a^{n-1}\equiv 1 \pmod n$（a=3,4,…），从而再排除一些满足条件$2^{n-1}\equiv 1 \pmod n$的合数？
  - 回答是：可以排除掉一些，但不能完全排除
  - 满足Fermat条件的数未必全是素数
  - 有些合数也满足Fermat条件，这些合数被称为Carmichael数

素数测试问题

- 二次探测定理
  - 如果p是一个素数，且0<x<p，则方程$x^2 \equiv 1 (\mod p)$的解为x=1，p-1
- 利用二次探测定理，可以在基于Fermat条件判断时，增加二次探测，一旦违背二次探测条件，则可得出不是素数的结论

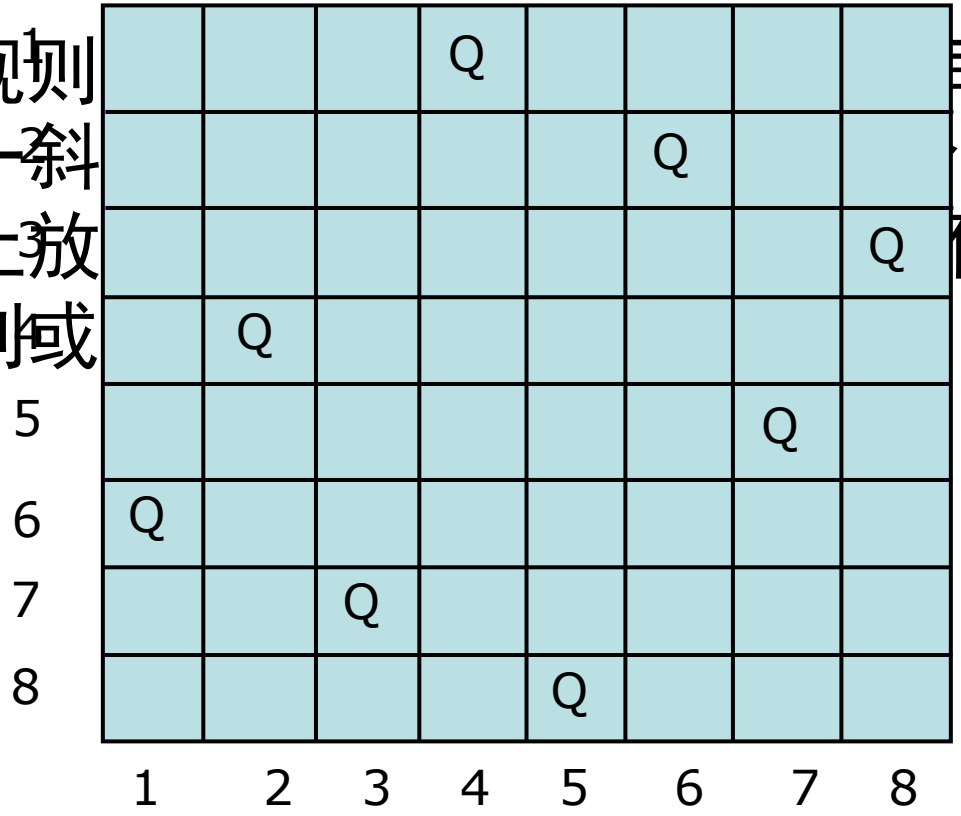
素数测试问题

```
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;}
```

```
public static boolean prime(int n)
   {// 素数测试的蒙特卡罗算法
     rnd = new Random();
     int a, result;
     composite=false;
     a=rnd.random(n-3)+2;
     result=power(a,n-1,n);
     if (composite||(result!=1)) return f
     else return true;
}
```

n后问题

- 在n×n格的棋盘上放置彼此不受攻击的n个皇后。
  - 按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。于是，n后问题等价于在n×n格的棋盘上放置n个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。



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。
- Las Vegas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

n后问题

- n后问题的Las Vegas算法思路：各行随机放置皇后，使新放的与已有的互不攻击，until(n皇后放好||无可供下一皇后放置的位置)

n后问题

- 1) *x*[8]←0; count←0;
- 2) for (*i*=1;*i*<=8; *i*++)
- 2.1) 产生一个[1,8]的随机数*j*;
- 2.2) count=count+1; //第count试探
- 2.3) *if* (皇后*i*放在位置*j*不发生冲突)
- 则*x*[*i*]=*j*; count=0; 转2)放下一个皇后;
- *if* (count==8) 算法失败;
- *else* 转2.1重新放皇后*i*;
- 3) *x*[1]~*x*[8]作为一个解输出;

n后问题

- 针对上述8皇后问题
  - 随机放两个皇后，再回溯比完全用回溯快大约两倍
  - 随机放3个皇后，再回溯比完全用回溯快大约一倍
  - 随机放所有皇后，比完全用回溯慢大约一倍

  - 产生随机数所需的时间导致