算法分析与设计

第8章 随机算法

主讲人：甘文生 PhD

Email: wsgan001@gmail.com

暨南大学网络空间安全学院

Fall 2021

Jinan University, China

第8章 随机算法

内容提要：

- 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(Monte Carlo)算法的设计思想
- 掌握拉斯维加斯(Las Vegas)算法的设计思想
- 掌握舍伍德(Sherwood)算法的设计思想

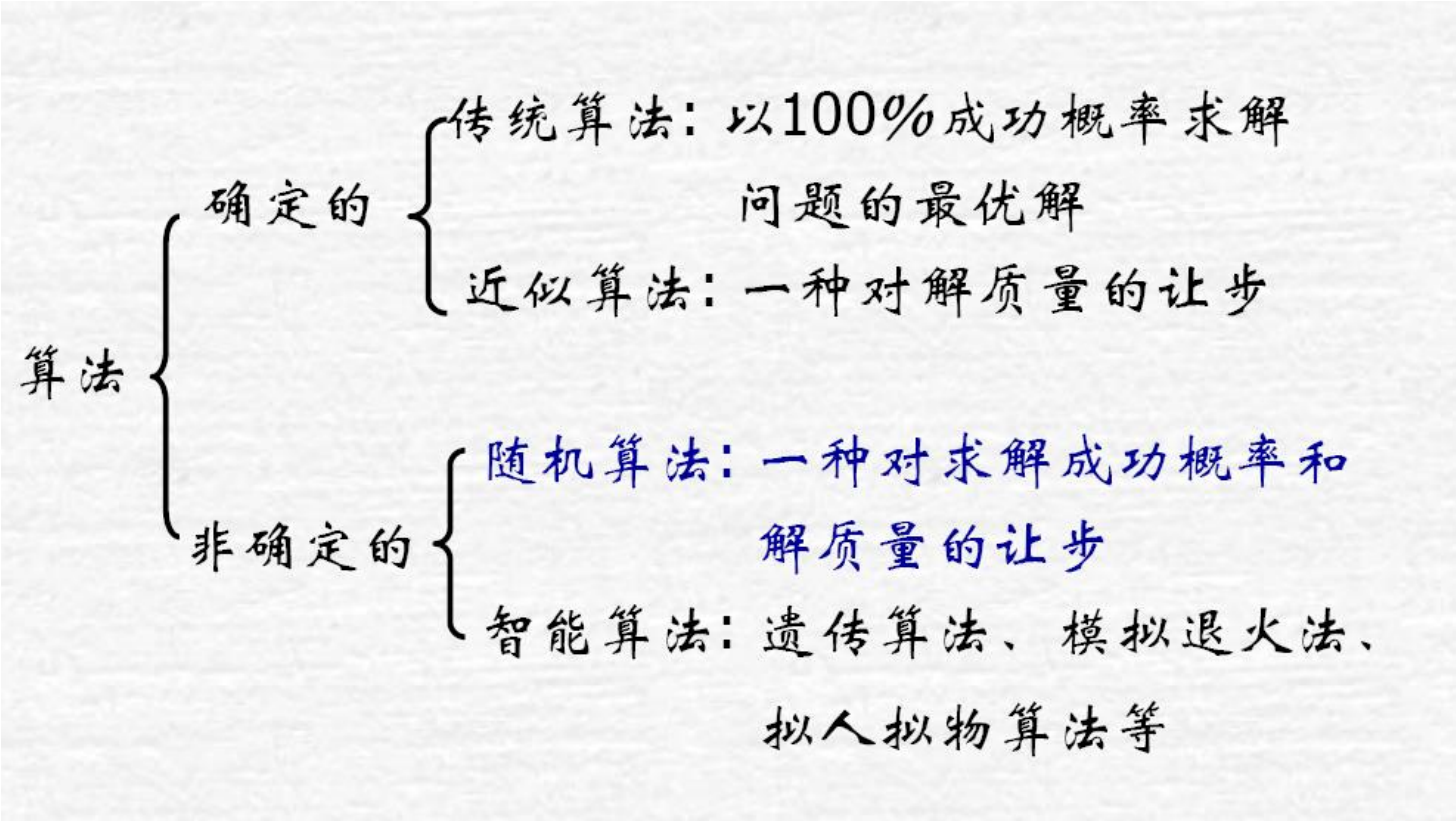
随机思想

- 在实际应用中，随机的概念在游戏软件中无所不在，为了增加游戏的趣味性、探索性和吸引力，游戏中的角色、奖品或武器等出现的位置与时间等都是随机的/不确定的。

- 在多数情况下，当算法在执行过程中面临一个选择时，随机性选择一般比最优选择省时，因此概率算法可在很大程度上降低算法复杂性。

- 本节将利用数据序列的随机性和概率分布等特点，设计解决问题的算法或提高已有算法的效率。

随机思想


算法
确定的
传统算法：以100%成功概率求解问题的最优解
近似算法：一种对解质量的让步
非确定的
随机算法：一种对求解成功概率和解质量的让步
智能算法：遗传算法、模拟退火法、拟人拟物算法等

随机算法的概述

- 随机算法是指在算法中执行某些步骤或某些动作时，所进行的选择是随机的。
- 三要素：输入实例、随机源和停止准则。
- 特点：简单、快速和易于并行化。
- 一种平衡：随机算法可以理解为在时间、空间和随机三大计算资源中的平衡 (Lu C.J. 博士论文, 1999)
- 重要文献：Motwani R. and Raghavan P., **Randomized Algorithms**. Cambridge University Press, New York, 1995

随机算法的概述

- 著名的例子

— Monte Carlo求定积分法

— 随机k-选择算法

— 随机快速排序

— 素性判定的随机算法

— 二阶段随机路由算法

- 重要人物和工作

— De Leeuw等人提出了概率图灵机, 1955

— John Gill的随机算法复杂性理论, 1977

— Rabin的数论和计算几何领域的工作, 1976

— Karp的算法概率分析方法, 1985

— Shor的素因子分解量子算法, 1994

方法概述

常见的随机算法分为4类：

① 数值随机化算法：常用于数值问题的求解，所得到的往往是近似解，解的精度随着计算时间增加而不断提高；

② 蒙特卡罗算法：用于求问题的准确解。该方法总可以得到问题的解，但是该解未必是正确的。求得正确解的概率依赖于算法所用的时间。比较难以判断解是否正确；

③ 拉斯维加斯算法：不会得到不正确的解，但是有时会找不到解。找到正确解的概率随着所用的计算时间的增加而提高。对任一实例，反复调用算法求解足够多次，可使求解失效的概率任意小；

④ 舍伍德算法：总能求得问题的一个解，且所求得的解总是正确的。当一个确定性算法最坏情况下的计算复杂性与其在平均情况下的计算复杂性有较大差别时，可在这个确定性算法中引入随机性将它改造成一个舍伍德算法，消除或者减少这种差别。核心思想：设法消除最坏情形行为与特定实例之间的关联性。

随机数

- 随机性(randomness)是偶然性的一种形式，是某一事件集合中的各个事件所表现出来的不确定性。产生某一随机性事件集合的过程，是一个不定因子不断产生的重复过程，但它可能遵循某个概率分布。

- 随机序列(random sequence)，或叫做随机变量序列，是随机变量形成的序列。一般的，如果用X1，X2……Xn代表随机变量，并按照一定顺序出现，就形成了随机序列。这种随机序列具备两个关键的特点：其一，序列中的每个变量都是随机的；其二，序列本身就是随机的。

随机数发生器

- 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，因此在随机化算法中使用的随机数都是在一定程度上随机的，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$a_0,a_1,\cdots,a_n$满足

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

其中$b \geq 0$，$c \geq 0$，$d \leq m$。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，已超出本章节讨论的范围。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。

随机数发生器

- **随机整数算法：**

利用线性同余式计算新的种子，将randSeed右移16位得到一个0~65535间的随机数，然后再将此随机整数映射到0~(n-1)范围内。

```
Random（n, s）
{  //产生 0 : n-1之间的随机整数，s为随机数发生器种子
   if( s==0)
         randSeed= time(0);   //用系统时间产生种子
   else
         randSeed = s;              //由用户提供种子
   randSeed = multiplier * randSeed + adder;
   return  (randSeed >>16) % n ;
}
```

其中： multiplier = 1194211693L， adder = 12345L

随机数发生器

- 随机实数生成算法

```
fRandom( s )
{  //产生[0，1)之间的随机实数
   return Random( maxshort ) / double( maxshort );
}
```

其中，maxshort = 65536L

思考：如何产生一个符合一定分布的随机数？

随机数发生器

- 假设抛10次硬币为一个事件，每次抛硬币得到正面和反面是随机的，利用计算机产生的伪随机数模拟抛硬币试验。

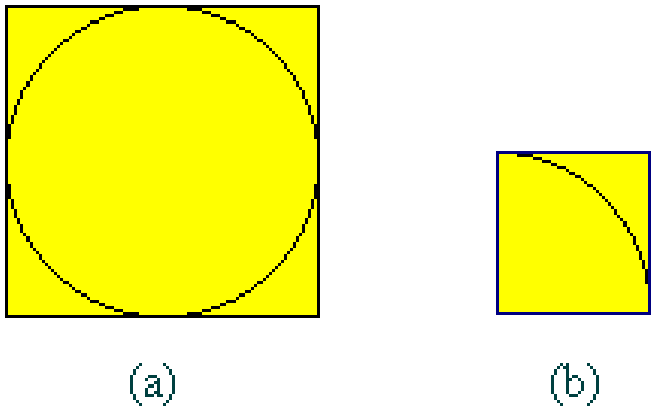
```
int TossCoins(int numberCoins)
{ // 随机抛numberCoins次硬币，返回得到正面的次数
    int i, tosses = 0;
    for( i = 0; i<numberCoins; i++ )
    {  // Random(2) = 1表示正面
        tosses += Random(2);
     }
     return tosses;
}

void main()
{  // 模拟随机抛硬币试验
   for( j= 0; j<11; j++ )
        heads[j] = 0;          // heads[i]得到i次正面的次数
    for( i = 0; i< 50000; i++ )
        heads[TossCoints(10)] ++ ;
}
```

数值随机化算法

例1：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 $\pi \approx \frac{4k}{n}$

```
double Darts(int n)
{ // 用随机投点法计算π值
   static RandomNumber dart;
   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);
}
```

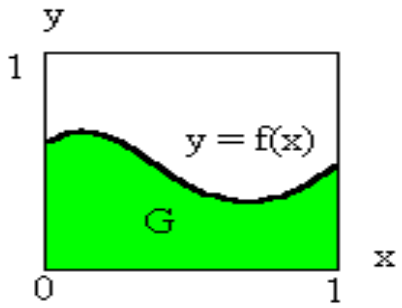


数值随机化算法

例2：计算定积分

设f(x)是[0，1]上的连续函数，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 $I \approx \frac{m}{n}$

数值随机化算法

```
double  Darts( int n)
{  //用随机投点法计算定积分
   int k=0;
   for( int i = 1; i<= n; i++)
   {
        double   x = dart.fRandom( );
        double   y = dart.fRandom( );
        if( y <= f(x) ) k++;
   }
   return  k/double(n);
}
```

思考：如果被积函数f(x)位于区间[a,b]中有界，并用M, L 表示其最大值和最小值。如何求定积分？

数值随机化算法

- 例3：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$x_1, x_2, \ldots, x_n$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。

数值随机化算法

思路： 构造一目标函数：$\Phi(x)=\sum f_i^{\,2}(x)$

式中，$x=(x_1, x_2, \ldots, x_n)$。该函数的极小值点是所求非线性方程组的一组解.

① **直观方法：** 在指定求根区域内，选定一个x0作为根的初值，按照预先选定的分布，逐个选取随机点x，计算目标函数Φ(x)，并把满足精度要求的随机点x作为所求非线性方程组的近似解。

② **改进方法：** 在指定求根区域D内，选定一个随机点x0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xj。在第j+1步，计算出下一步的随机搜索增量Δxj。从当前点xj依Δxj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。

随机算法的比较与总结

- 随机序列提高算法的平均复杂度—— 舍伍德(Sherwood)算法
- 随机生成答案并检测答案正确性—— 拉斯维加斯(Las Vegas)算法
- 考虑正确几率的算法—— 蒙特卡罗(Monte Carlo)算法

算法	有解	解正确	改进复杂性的优势	设计要点
Sherwood	有	是	可能改善最坏情况	随机选择
Las Vegas	不定	是	可能改善平均情况	与确定算法相结合
Monte Carlo	有	不定	解决目前困难的问题	概率>1/2 多次执行

- 蒙特卡罗算法：采样越多，越近似最优解；拉斯维加斯算法：采样越多，越有机会找到最优解

舍伍德(Sherwood)算法

- 算法的平均时间复杂度主要依赖数据的规模，其次很多算法还依赖于数据的初始状态，典型的有插入排序、快速排序和搜索算法等都是如此。特别是快速排序，平均时间复杂度为$O(nlogn)$，但在数据基本有序时最坏时间复杂度为$O(n^2)$。舍伍德算法的思想就是**应用随机序列提高算法的平均时间复杂度**。

- 舍伍德算法是概率算法的一种，该文在比较**线性表的顺序存储与链式存储**的特点之后，提出了一种较优的数据结构——**用数组模拟链表(时间复杂度为$0(n^{\wedge 1/2})$)**。

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∈Xn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

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

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。

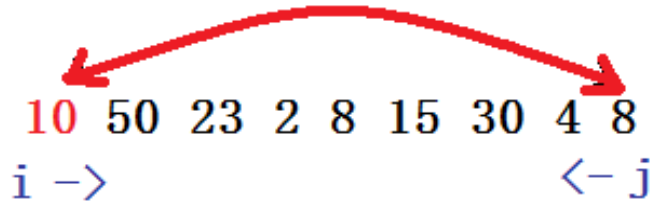
$$\bar{t}_B(n) = \sum_{x \in X_n} t_B(x) / |X_n| = \bar{t}_A(n) + s(n)$$

舍伍德算法的核心

- 总能求得问题的一个解，且所求得的解总是正确的。
- 当一个确定性算法最坏情况下的计算复杂性与其在平均情况下的计算复杂性有较大差别时，可在这个确定性算法中引入随机性将它改造成一个舍伍德算法，消除或者减少这种差别。
- 核心思想：设法消除最坏情形行为与特定实例之间的关联性。
- 快速排序中利用随机序列选取枢轴值，能提高快速排序的平均效率，避免最差情况的出现。快速排序算法的关键在于一次划分中选择合适的轴值作为划分的基准。

舍伍德算法

- 例[快速排序]



- 平均复杂度：O(nlogn)

2 8 4 8 10 50 23 15 30

一趟排序前后

- 最坏复杂度：O(n^2)
  - A = (n,n-1,n-2,…,1)，每次都取A1为枢轴值。
- 添加随机性后：“随机选择枢轴值”
  - 在取枢轴值之前，随机选择一个数Ai，与A1交换。
  - 使算法不受待排序序列初始状态的影响，在最坏情况下的时间性能趋近于平均情况的时间性能。
- 最坏复杂度：期望为O(nlogn)，达到最坏的概率非常小，可以忽略。

舍伍德算法

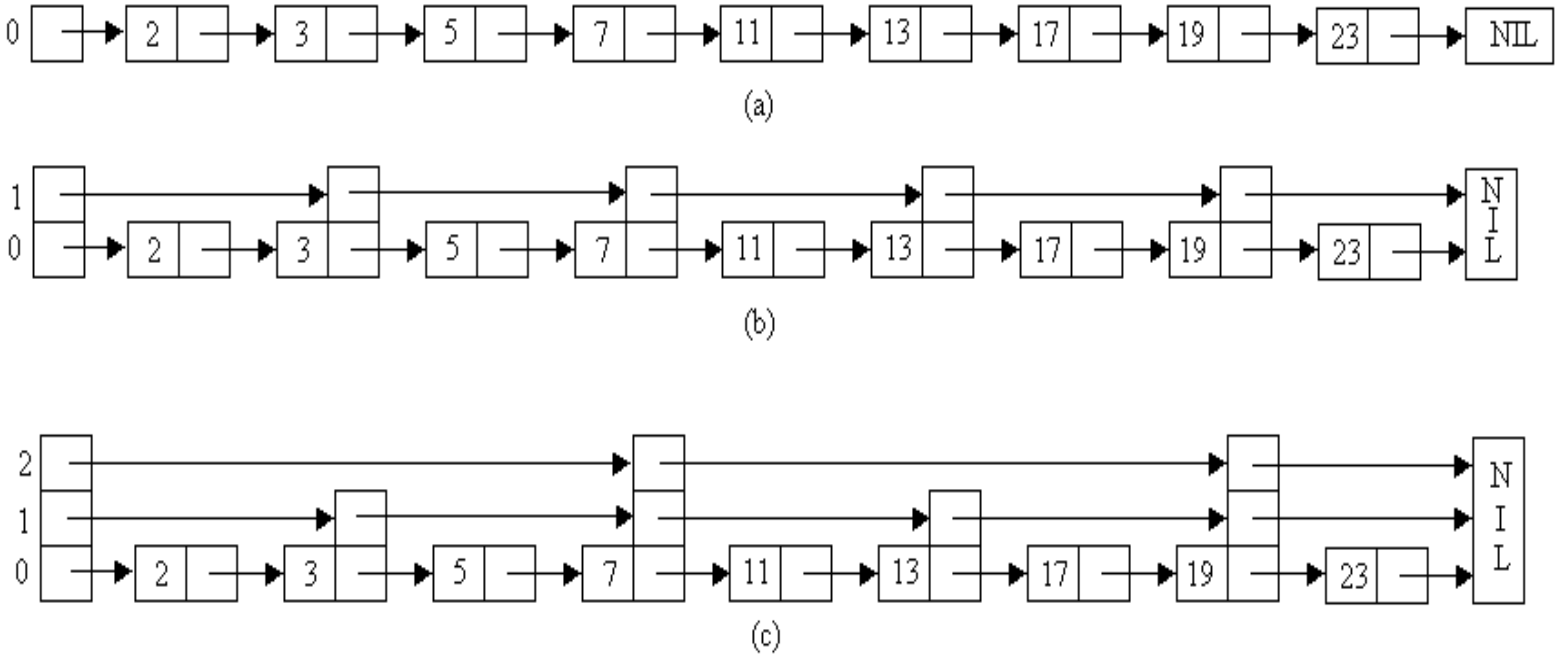
- 舍伍德 Sherwood算法：快速排序算法、最坏时间线性选择算法
- 有时也会遇到这样的情况，即所给的确定性算法无法直接改造成舍伍德型算法。此时可借助于随机预处理技术，不改变原有的确定性算法，仅对其输入实例进行随机排列/即洗牌，同样可收到舍伍德算法的效果。例如，对于确定性选择算法，可以用下面的洗牌算法shuffle将数组a中元素随机排列，然后用确定性选择算法求解。这样做所收到的效果与舍伍德型算法的效果是一样的。

```
void Shuffle( Type a[], int n)
{ // 随机洗牌算法
    static RandomNumber rnd ;
    for (int i=0; i<n; i++)
    {
       int j = rnd.Random(n-i) + i;
       Swap(a[i], a[j]);
    }
}
```

舍伍德算法——跳跃表

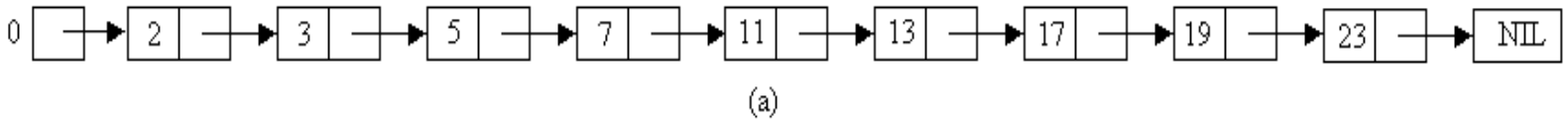
- 舍伍德型算法的设计思想可用于设计高效的数据结构；
- 如果用有序链表来表示一个含有n个元素的有序集S，则在最坏情况下，搜索S中的一个元素需要Ω(n)计算时间。
- 提高有序链表效率的一个技巧是在有序链表的部分结点处增设附加指针以提高其搜索性能。在增设附加指针的有序链表中搜索一个元素时，可借助于附加指针跳过链表中若干结点，加快搜索速度。这种增加了向前附加指针的有序链表称为跳跃表。
- 应在跳跃表的哪些结点增加附加指针以及在该结点处应增加多少指针完全采用随机化方法来确定。这使得跳跃表可在O(logn)平均时间内支持关于有序集的搜索、插入和删除等运算。

舍伍德算法——跳跃表

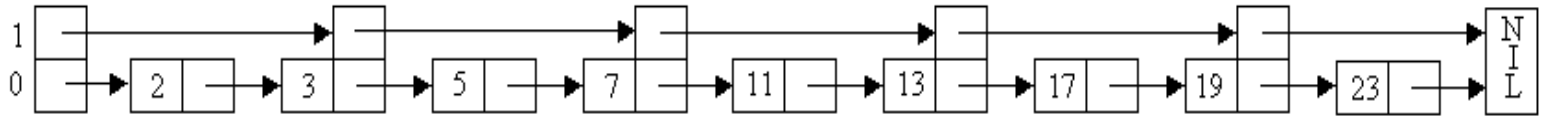

0
2
3
5
7
11
13
17
19
23
NIL
(a)
1
0
2
3
5
7
11
13
17
19
23
NIL
(b)
2
1
0
2
3
5
7
11
13
17
19
23
NIL
(c)

舍伍德算法——跳跃表

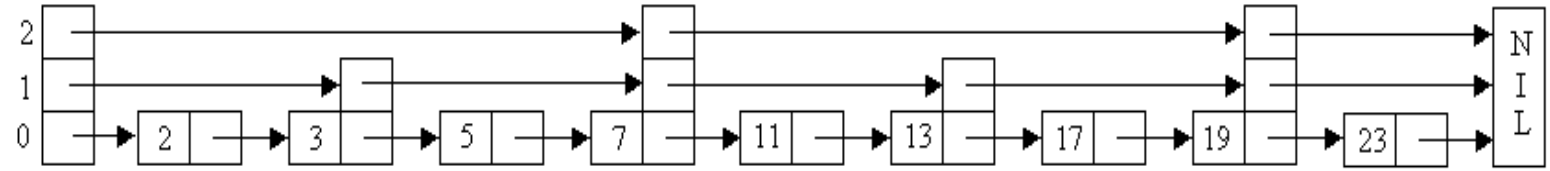
- 在一般情况下，给定一个含有n个元素的有序链表，可以将它改造成一个完全跳跃表，使得每一个k级结点含有k+1个指针，分别跳过2^k-1，$2^{k-1}-1$，…，2^0-1个中间结点。第i个k级结点安排在跳跃表的位置$I^{2\wedge k}$处，i≥0。则可以在时间O(logn)内完成集合成员的搜索运算。在一个完全跳跃表中，最高级的结点是⌈logn⌉级结点。



(a)



(b)

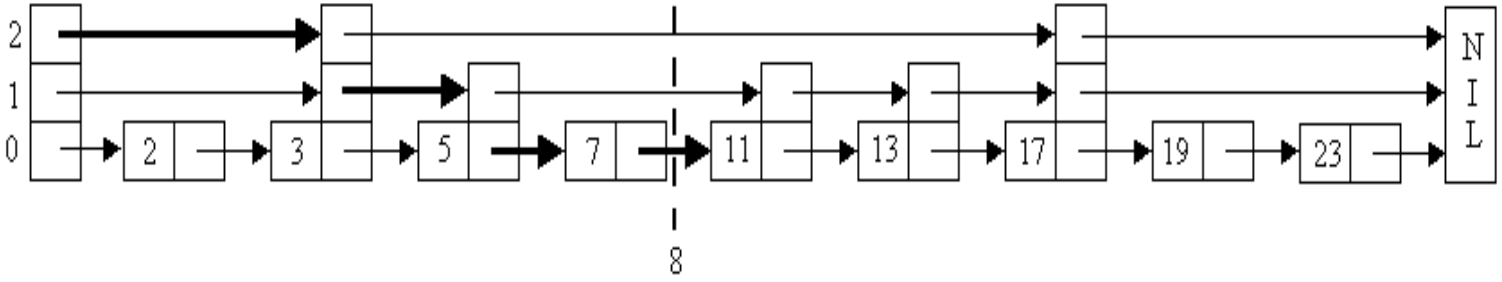


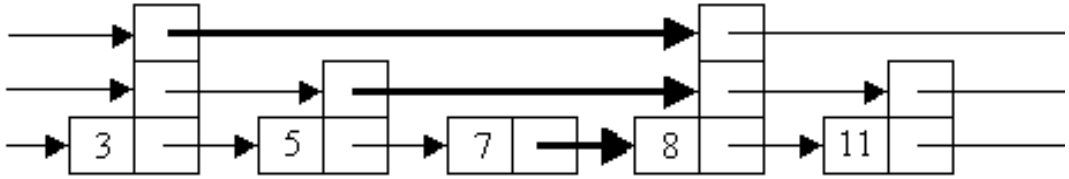
(c)

舍伍德算法——跳跃表

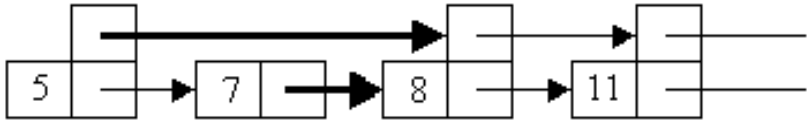
- 完全跳跃表与完全二叉搜索树的情形非常类似。它可以有效地支持成员搜索运算，但不适应于集合动态变化的情况。集合元素的插入和删除运算会破坏完全跳跃表原有的平衡状态，影响后继元素搜索的效率。
- 为了在动态变化中维持跳跃表中附加指针的平衡性，必须使跳跃表中k级结点数维持在总结点数的一定比例范围内。
- 注意到在一个完全跳跃表中，50%的指针是0级指针；25%的指针是1级指针；...；$(100/2^{k+1})\%$的指针是k级指针。
- 因此，在插入一个元素时，以概率1/2引入一个0级结点，以概率1/4引入一个1级结点，...，以概率$1/2^{k+1}$引入一个k级结点。
- 另一方面，一个i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.

舍伍德算法——跳跃表





(a)



(b)

舍伍德算法——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。
- 因此，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

- 现实中很多问题及其解无规律可言，无法使用递推、贪婪法、分治法或动态规划法解决，只能通过蛮力法来解决，也就是把所有可能的结果列举出来，逐一判断那个是正确结果。这种算法无论利用循环还是递归机制进行枚举，只能按一定规律从小到大或从大到小枚举，而问题的答案一般是中间的数据，这样，算法的效率往往较低。
- 拉斯维加斯算法的思想是用随机序列代替有规律的枚举，然后判断随机枚举结果是否问题的正确解。此方法在不需要全部解时，一般可以快速找到一个答案；当然也有可能在限定的随机枚举次数下找不到问题的解，只要这种情况出现的概率不占多数，就认为拉斯维加斯算法是可行的。当出现失败情况时，还可以再次运行概率算法，就又有成功的可能。
- 核心思想：随机生成答案并检测答案正确性。

拉斯维加斯(Las Vegas)算法

- 随机生成答案并检测答案正确性。
- 对于找不到有效算法的某些问题，可使用Las Vegas算法来求解，可能会很快找到问题的一个解。
- 一旦能得到问题的一个解，一定是正确的。但是，这种算法所作随机性决策可能导致找不到解。
- 可通过多次调用同一个Las Vegas算法来增加得到问题解的概率。

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做的随机性决策有可能导致算法找不到所需要的解。

```
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
  bool success= false;
  while (!success) success=lv(x,y);
}
```

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：

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

拉斯维加斯(Las Vegas)算法

例1：N-皇后问题

- **分析：**对于n皇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，像是随机放置的。由此容易想到下面的**拉斯维加斯算法**。

- **思路：**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。

拉斯维加斯(Las Vegas)算法

- 随机“爬山法”
  - 局部极大值 vs 全局最优值
  - 随机重新开始
  - 完备性接近1
  - 对于3,000,000个皇后，可在1分钟以内找到解

拉斯维加斯(Las Vegas)算法

```
QueensLV( n )
{  //随机放置n个皇后的拉斯维加斯算法
    int  k = 1;   //下一个放置的皇后编号
    int  count = 1;
    while((k <= n) && (count > 0)) {
        count = 0;
        //寻找第i个皇后可能放的所有合法位置
        for (int i = 1; i <= n; i++){
            x[k] = i;
            if(Place(k) )   y[count++] = i;
        }
        //从合法位置中随机选择一个位置
       if(count > 0) x[k++] = y[Random(count)];
   }
    return  (count>0);  //放置成功
}
```

```
Place( int k)
{ //判断皇后k置于第x[k]列的合法性
    for(int j=1; j<k; j++)
        if( (abs(k-j) == abs( x[j] – x[k]))
         || (x[j] == x[k]))
            return false;
    return  true;
}
```

拉斯维加斯(Las Vegas)算法

- 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。策略：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

拉斯维加斯(Las Vegas)算法

```
Backtrack( int t )
{ //解n后问题的回溯法
    if ( t> n )
        return  x;        //找到解
    else
        for( int i=1; i<=n; i++ )  {
            x[t] = i;
            if( Place(t) && Backtrack(t+1) )
                return true;
          }
    return false;
}
```

```
//与回溯法相结合的解n后问题的拉斯维加斯算法
void  nQueen( int n)
{
    //表示初始随机放置的皇后个数
    int stop = 5;
    if( n >15)  stop  = n-15;
    while( QueensLV( stop ) );

    //算法的回溯搜索部分
    if ( Backtrack( stop + 1) )
            打印出解;
 }
```

拉斯维加斯(Las Vegas)算法

例2：整数因子分解

设n>1是一个整数。关于整数n的因子分解问题是关于找出n的如下形式的唯一分解式：

$$n = p_1^{m_1} p_2^{m_2} \cdots p_k^{m_k}$$

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。

拉斯维加斯(Las Vegas)算法

一个数的因子没有统一的规律，只能通过枚举算法来枚举可能的因子，算法如下：

```
int Split(int n){
  // sqrt(n)为开方函数
  int m =  floor(sqrt(double(n)));
  for (int i=2; i<=m; i++)
      if (n%i==0) return i;
  return 1;
}
```

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

拉斯维加斯(Las Vegas)算法

Pollard算法：

利用拉斯维加斯算法的思想，在开始时选取0～n-1范围内的随机数，然后由 $x_i = (x_{i-1}^2 - 1) \bmod n$ 递推产生无穷序列 $x_1, x_2, \cdots, x_k, \cdots$

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。

拉斯维加斯(Las Vegas)算法

```
void Pollard(int n)
{ // 求整数n因子分割的拉斯维加斯算法
  RandomNumber rnd;
  int i=1;
  int x=rnd.Random(n); / 随机整数
  int y=x; k=2;
  while (true) {
    i++;
   // 求无穷序列x1, x2, …, xk
    x=(x*x-1) mod n;
   // gcd(a , b)求a , b 的最大公因子
   int d=gcd(y-x,n);
    // 求n的非平凡因子
    if ((d>1) && (d<n)) cout<<d<<endl;
    if (i==k) {
       y=x;
       k*=2;}
   }
}
```

对Pollard算法深入分析可知，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

哈希查找概述

例3 字符串匹配

- 哈希查找概述；简单的哈希函数
- 字符串匹配的拉斯维加斯算法
- 哈希查找算法是一个高效的算法，如果设计得当，可以通过O(1)的比较次数找到所需元素。哈希查找算法不可避免地会有地址冲突的问题。
- 哈希算法的思想有许多其他的应用，例如在挖掘关联规则的过程中利用哈希查找的办法发现二次的频繁/大项集(large itemsets)

简单的哈希函数

- 网络传输的冗余位就是一个哈希函数

1　　1　　0　　1　　0　　0　　1　　0

$$H(7)=\left(\sum_{i=1}^{6} f(i)\right) \bmod 2$$

- 虽然不能保证冗余位相同的两个字节相同，但冗余位不同的两个字节一定是不同的。
- 网络传输中的某一位出现了错误是能够发现的，而这又是最常见的情况。
- 字符串匹配算法总结：朴素的匹配算法，时间复杂度O(mn)。KMP算法，时间复杂度O(m+n)

字符串匹配的拉斯维加斯算法

一个简单例子：假设字符集是{a, b, c, d}。在母串"b c b c c d a b c"中查找子串"c d a"

- 任意取一个素数19 (在计算允许的情况下越大越好)
- 把子串"c d a"变换成4进制数2 3 0，这个数对应的十进制数是44，模19之后的数是6
- 计算母串的前3个字符的4进制模19的数值，是6。因此，前3个字符构成的子串有可能和模式串匹配，通过朴素的算法发现它们不匹配

字符串匹配的拉斯维加斯算法

字符集是{a, b, c, d}。在母串"b c b c c d a b c"中查找子串"c d a"

- 由"b c b"的哈希值可以花费O(1)的计算得到"c b c"的哈希值。计算过程如下：(6-16*1)%19 = 9；(9*4)%19 = 17, (17+2)%19 = 0，即"c b c"的哈希值是0。通过验证发现该计算是正确的。

- 依次可以计算"bcc"、"ccd"、……的哈希值。如果和模式串的哈希值6相同，则朴素匹配验证；否则，略过。

拉斯维加斯算法

- 基于哈希计算的字符串匹配算法是拉斯维加斯型概率算法，母串的子串与模式串的哈希值相等时并不能保证匹配成功，而需要用朴素的方式进行验证。

- 拉斯维加斯型概率算法的时间复杂度是**O(m+n)**,当伪匹配大量出现时,复杂度会有所增加。

- 所选素数足够大时，能够保证不出现伪匹配。但算法所能容忍的素数大小与计算机的处理位数有关。例如，64位机所能采用的素数不会超过264。否则，虽然减少了伪匹配的可能性，但增加每次移动窗口所需的计算量。

蒙特卡罗算法

- 算法特点
  - 在实际应用中(例如天气预报、机器问答系统)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
  - Monte Carlo算法可以高概率给出正确解：在一般情况下保证对问题的所有实例都以高概率给出正确解
  - 它通常无法判定一个具体解是否正确
  - 可通过重复调用同一个Monte Carlo算法多次来提高获得正确解的概率

蒙特卡罗算法

考虑正确几率的算法—蒙特卡罗(MC)算法的相关术语：

- p正确(p-correct)：设如果一个MC算法对于问题的任一实例得到正确解的概率不小于p，p是一个实数，且1/2≤p<1。且称p-1/2是该算法的优势(advantage)。
- 一致的(consistent)：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。
- 偏真(true-biased)算法：当MC是求解判定问题的算法，算法MC返回true时解总是正确的，当它返回false不一定正确。反之称为偏假(flase-biased)算法。

蒙特卡罗算法

考虑正确几率的算法—蒙特卡罗(MC)算法的相关术语：

- **p正确(p-correct)**：设如果一个MC算法对于问题的任一实例得到正确解的概率不小于p，p是一个实数，且1/2≤p<1。且称p-1/2是该算法的优势(advantage)。
- **一致的(consistent)**：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。
- **偏真(true-biased)算法**：当MC是求解判定问题的算法，算法MC返回true时解总是正确的，当它返回false不一定正确。反之称为偏假(flase-biased)算法。

蒙特卡罗算法

- 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。

- 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，

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

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。

蒙特卡罗算法

例1 [主元素问题]:

1 3 2 1 6 1 1 8 12 1 7 1 1

- 定义：在数列中，是否有出现次数超过一半的元素。
- 算法：随机选择一个数，扫描出现次数。若超过一半，输出“YES”，否则输出“NO”。
- 分析：算法时间复杂度为O(n)。正确的概率>1/2。数次调用，可使得正确概率接近1。
- 有确定性O(n)的算法

蒙特卡罗算法——主元素问题

- 例1[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。

```
bool Majority(Type *T, int n)
{ // 判定所给数组T是否包含主元素
  //的蒙特卡罗算法
    int i=Random(n)+1;
    Type 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
}
```

说明： 算法对随机选择的数组元素x，测试它是否为数组T的主元素。

- 如果算法返回true，则x肯定是数组T的主元素。
- 反之，返回false，数组T未必没有主元素。可能数组包含主元素，但是元素x不是主元素。
- 由于数组T的非主元素个数小于n/2，故这种情况发生的概率小于1/2，因此上述判定数组T的主元素存在性的算法是一个偏真的1/2正确算法。

蒙特卡罗算法——主元素问题

- 通过重复调用Majority算法可以把错误概率降低到任何能接受的范围

```
bool MajorityMC(Type *T, int n, double e)
{  // 重复调用算法Majority
   int k=ceil(log(1/e)/log(2));
   for (int i=1;i<=k;i++)
        if (Majority(T,n)) return true;
    return false;
}
```

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))。

蒙特卡罗算法——素数测试

例2[素数测试]：

算法设计1: 至今没有发现素数的解析式表示方法，判定一个给定的整数是否为素数，一般通过枚举算法来完成:

```
int  prime(int n){
  for (i=2; i <=n-1; i++)
    if (n mod i= =0)  return 0;
 return 1; }
```

算法的时间复杂性是O(n),可以缩小枚举范围为$2\text{-}\ n^{1/2}$，算法的时间复杂性是$O(n^{1/2})$。但是对于一个m位正整数n，算法的时间复杂性仍是$O(10^{m/2})$，即该算法的时间复杂性*关于位数是指数阶的*。

蒙特卡罗算法——素数测试

算法设计2: 简单的随机算法，随机选择一个数，若是n的因数，则下结论n不是素数；否则下结论n是素数。

```
prime(int n)
{   d = Random(2,sqrt(n));        // 产生2 – n^1/2的随机整数;
    if (n mod d ==0) return 0;
    else return 1; }
```

若返回0，则算法幸运地找到了n的一个非平凡因子，n为合数，算法完全正确。因此，这是一个偏假算法。若返回1，则未必是素数。实际上，若n是合数，prime 也可能以高概率返回1。

例如：n=61*43，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≠43，算法返回true(概率为98%)，结果错误。当n增大时，情况更差。

蒙特卡罗算法——素数测试

算法设计3: 为了提高算法的正确率，先看以下定理及分析。

(1) 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

$$(n-1)\ !\equiv -1 \pmod{n}$$

*其中，Wilson定理具有很高的理论价值，但是实际用于素性测试所需要计算量太大，无法实现对较大素数的测试。

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

*费尔马小定理只是给出了素数判定的一个必要条件，满足该条件的整数不一定就是素数。如n=341就是满足该条件的最小合数。

(3) 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。

蒙特卡罗算法

- 例2[素数测试]：
  - Miller-Rabin随机测试
  - 基于两个定理：
    - Fermat小定理
    - 二次探测定理
  - 过程：假设$n-1=2^r s$，随机产生a，依次考察

    $a^s \bmod n, a^{2s} \bmod n, \ldots, a^{n-1} \bmod n$，

    该序列必定以1结束，且在第一次出现1之前的值必定是n-1
  - 每次测试失败的概率小于1/4。
  - 多次重复可以极大概率得到正确解。

蒙特卡罗算法

算法power用于计算a^p mod n，并在计算过程中实施对n的二次探测！

```
void power (unsigned int a, unsigned int p,  unsigned int n,  unsigned
int & result, bool &composite)
{  // 计算mod n，并实施对n的二次探测
     unsigned int x;
     if (p==0) result=1;
    else
    {
         power(a,p/2,n,x,composite);  // 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;
     }
}
```

蒙特卡罗算法

在算法power基础上，可以设计素数测试的蒙特卡罗算法Prime如下：

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

说明： Prime算法返回false时，整数n一定是一个合数。当返回true时，整数n在高概率的意义下是一个素数。即仍然可能存在合数n，对于随机选取的基数a，算法返回true。但对于上述算法的深入分析表明，当n充分大时，这样的基数a不超过(n-9)/4个。即**算法prime是一个偏假3/4正确的蒙特卡罗算法。**

蒙特卡罗算法

上述Prime算法的错误概率可以通过多次重复调用而快速降低。重复k次调用算法prime的蒙特卡罗算法primeMC如下：

```
primeMC(int  n, int k)
{  //重复k次调用算法Prime的蒙特卡罗算法
    rnd = new Random( );
    int a, result;
    composite = false;
    for ( int i = 1;  i <= k;  i++ )
    {
       a = rnd.random(n-3) + 2;
       result = power( a, n-1, n);
       if( composite || (result != 1) )
             return false;
    }
    return true;
}
```

容易算出算法primeMC的错误概率不超过$(1/4)^k$。这是一个很保守的估计，实际使用的效果要好得多。

简单应用——洗牌

- 多种洗牌算法
- 常见的一种算法：
  - 保证每个位置出现任何一张牌的概率均相等

```
1    n ← length[A]
2    for i ← 1 to n
3        do swap A[i] ←→ A[Random(1,n)]
```

简单应用——s-t连通性

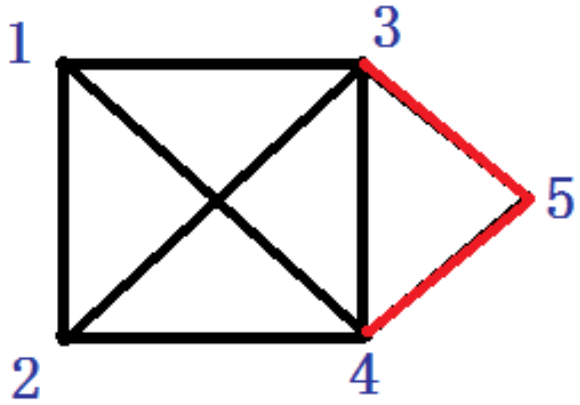
- **s-t连通性算法：** 无向图G=(V,E), s, t为G上两点。令n=|V|，m=|E|。希望确定是否存在一条连接s和t的路。S到T有路吗？

- **随机算法：** 从s开始随机游动，如果在$4n^3$步之内到达t，则返回存在一条路；否则，返回不存在路。

- 算法以1/2的概率返回正确结果。

简单应用: 最小割随机算法

- 无向图G = (V，E) 中找最小边割集。
- **最小割随机算法**：每次随机选一条边，合并该边对应的顶点。重复该过程n-2次。最后剩下两点之间的边，就是一个割集。
- 此方法至少以2/n(n-1)的概率输出最小割集。
- 重复上述方法n(n-1)lnn次，输出不是一个割集的概率≤$1/n^2$。



例题分析1

- 给出3个n*n (n≤500)的矩阵A，B，C，验证A*B = C?
  - n^3算法会超时
  - n^2的概率算法：
    - 随机置换法则：如果比特向量a ≠b, r为随机向量，那么

      $$Pr[(a,r) \neq (b,r)] \geq 1/2$$

    - 随机一个1*n的向量r，验证A*B*r=C*r，若成立，输出“YES”，否则输出“NO”。
    - 如果A*B≠C，算法以1/2的概率输出“NO”！

例题分析2

- 一堆数分为两堆。要求两堆之间元素个数差不超过1，并且两堆和的差值尽量小。(元素个数≤100,元素值≤450)

  - 动态规划可以解决

  - 随机算法如下：
    - 1. 计算两堆的大小后，随机分为两堆。
    - 2. 随机选择两堆中的数，如果交换后差变小，则交换。
    - 3. 重复2，直到多次交换未发生。更新答案。
    - 4. 回到1，重新开始。

随机算法小结

- **确定型算法 vs 随机算法**
  - 确定型算法：对某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。
  - 随机算法：对某个特定输入的每次运行过程是随机的，运行结果也可能是随机的。

- **随机算法的优势**
  - 在许多情况下，随机性选择通常比最优选择省时，可以很大程度上降低算法的计算复杂度。
  - 在运行时间或者空间需求上随机算法比确定型算法往往有较好的改进。
  - 随机算法设计简单。

随机算法的比较与总结

- **数值随机算法**常用于数值问题的求解。在许多情况下，计算问题的精确解是不可能或者没有必要的，数值随机算法能得到相当满意的解；
- **蒙特卡罗算法**一定可以得到问题的解，但是得到的解未必是正确的。解的正确概率依赖于算法所用的时间；
- **拉斯维加斯算法**所找到的解一定是正确的，但是有时该算法无法找到问题的解。算法找到问题的正确解的概率也依赖于所用的计算时间；
- **舍伍德算法**总能求得问题的正确解，它用于改善确定性算法的时间复杂度。其主要思想是：避免算法最坏情况，设法消除最坏情形和特定实例之间的关联。

随机算法的比较与总结

- 数值概率算法：数值问题的求解，最优化问题的近似解
- 蒙特卡罗算法：判定问题的准确解，不一定正确
- 拉斯维加斯算法：不一定会得到解，但得到的解一定是正确解
- 舍伍德算法：总能求得一个解，且一定是正确解

随机算法的比较与总结

- 随机序列提高算法的平均复杂度—— 舍伍德(Sherwood)算法
- 随机生成答案并检测答案正确性—— 拉斯维加斯(Las Vegas)算法
- 考虑正确几率的算法—— 蒙特卡罗(Monte Carlo)算法

算法	有解	解正确	改进复杂性的优势	设计要点
Sherwood	有	是	可能改善最坏情况	随机选择
Las Vegas	不定	是	可能改善平均情况	与确定算法相结合
Monte Carlo	有	不定	解决目前困难的问题	概率>1/2 多次执行

- 蒙特卡罗算法应用：测试串相等(通信纠错)、模式匹配、随机抽样

谢谢！

Q & A