

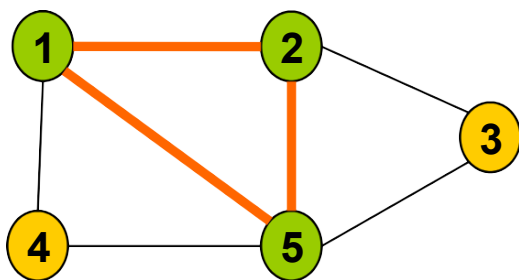
# 分支限界算法

陈长建

计算机科学系

# 分支限界法案例5- 最大团问题

- 团 (clique) : 社会团体, 团体中的个体互相认识。
- 最大团问题 (Maximum Clique Problem, MCP) 是图论中一个经典的组合优化问题, 也是一类NP完全问题。



例: 子集 $\{1, 2\}$ 是 $G$ 的大小为2的完全子图。这个完全子图不是团, 因为它被 $G$ 的更大的完全子图 $\{1, 2, 5\}$ 包含。 $\{1, 2, 5\}$ 是 $G$ 的最大团。 $\{1, 4, 5\}$ 和 $\{2, 3, 5\}$ 也是 $G$ 的最大团。

# 回顾：回溯思路

- 首先设最大团为一个空团，往其中加入一个顶点，然后依次考虑每个顶点，查看该顶点加入团之后是否仍然构成一个团，如果可以，考虑将该顶点加入团或者舍弃两种情况，如果不行，直接舍弃，然后递归判断下一顶点。
- 判断当前顶点加入团之后是否仍是一个团：只需要考虑该顶点和团中顶点是否都有连接。
- 剪枝策略：如果剩余未考虑的顶点数加上团中顶点数不大于当前解的顶点数，可停止继续深度搜索，否则继续深度递归当搜索到一个叶结点时，即可停止搜索，更新最优解和最优值。

# 上界函数及优先级

- $\text{cliqueSize}$ 表示与该结点相应的团的顶点数
- $\text{level}$ 表示结点在子集空间树中所处的层次
- $\text{cliqueSize} + n - \text{level} + 1$ 作为顶点数上界 $\text{upperSize}$ 的值
- $\text{upperSize}$ 实际上也是优先队列中元素的优先级
- 算法总是从活结点优先队列中抽取具有最大 $\text{upperSize}$ 值的元素作为下一个扩展元素

# 算法思想：子集树

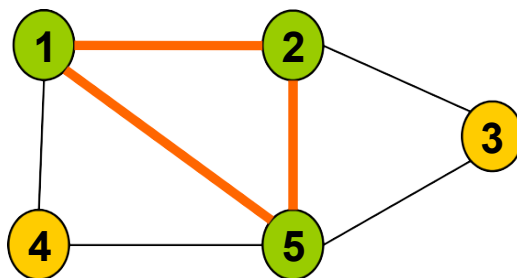
- 子集树的根结点是初始扩展结点，其cliqueSize的值为0
- 扩展内部结点时，
  - 先考察左子结点。将顶点  $i$  加入到当前团中，并检查可行性，即该顶点与当前团中其它顶点是否都有边相连。
    - 可行结点，将加入到子集树中并插入活结点优先队列。
  - 继续考察当前扩展结点的右子结点。
    - 当  $upperSize > bestn$  时，右子树中可能含有最优解，此时将右子结点加入到子集树中并插入到活结点优先队列中。

# 算法思想：while循环的终止条件

- 子集树中的一个叶结点(即 $n+1$ 层结点)成为当前扩展结点。
  - 对于子集树中的叶结点, 有 $\text{upperSize} = \text{cliqueSize}$ 。
  - 此时活结点优先队列中剩余结点的 $\text{upperSize}$ 值均不超过当前扩展结点的 $\text{upperSize}$ 值, 从而进一步搜索不可能得到更大的团, 此时算法已找到一个最优解。

# 练习

- 用优先队列方式求解下列最大团问题



# 随机化算法

陈长建

计算机科学系



# 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
- 掌握舍伍德算法的设计思想
- 掌握拉斯维加斯算法的设计思想
- 掌握蒙特卡罗算法的设计思想

# 引言

- 前面几章所讨论的分治、动态规划、贪心法、回溯和分支限界等算法的**每一计算步骤都是确定的**，本章所讨论的概率算法允许执行过程中随机选择下一计算步骤。
- 在多数情况下，当算法在执行过程中面临一个选择时，随机性选择常比最优选择**省时**，因此概率算法可在很大程度上**降低算法复杂性**。
- **概率算法的一个基本特征**是对所求解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(所需时间或计算结果)。

# 引言

- 本章将要介绍的概率算法包括：
  - 数值随机化算法 求解数值问题的近似解，精度随计算时间增加而不断提高。
  - 舍伍德算法 消除算法最坏情况行为与特定实例之间的关联性，并不提高平均性能，也不是刻意避免算法的最坏情况行为。
  - 拉斯维加斯算法 求解问题的正确解，但可能找不到解。
  - 蒙特卡罗算法 求解问题的准确解，但这个解未必正确，且一般情况下无法有效判定正确性。

# 随机化算法概述

- 一个**随机化算法** (randomized algorithm) 是指需要利用随机数发生器的算法, 算法执行的某些选择依赖于随机数发生器所产生的随机数。

# 随机化算法

- 随机化算法有时也称概率算法 (probabilistic algorithm) , 但也有人对两者这样区分:
- 如果取得结果的途径是随机的, 则称为随机算法, 如拉斯维加斯算法; 而如果取得的解是否正确存在随机性, 称为概率算法, 如蒙特卡罗算法。
- 本课中统一称为随机化算法。

# 随机化算法的特点

- 当算法执行过程中面临选择时，概率算法通常比最优选择算法**省时**。
- 对所求问题的**同一实例**用**同一概率算法**求解两次，两次求解所需的时间甚至所得的结果**可能有相当大的差别**。
- 设计思想简单，易于实现。

# 随机算法分类



# 数值随机算法

- 数值随机算法 (numerical randomized algorithm) 用于求数值问题的近似解。



# 数字炸弹游戏



# 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，因此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。

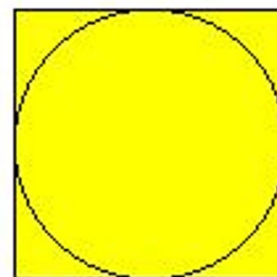
# 用随机投点法计算 $\pi$ 值

- 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

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



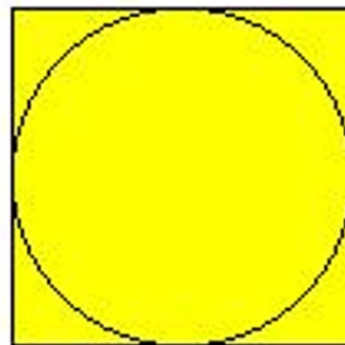
(a)



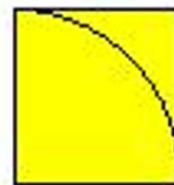
(b)

# 用随机投点法计算 $\pi$ 值

```
double Darts(int n)
{ // 用随机投点法计算 $\pi$ 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);
}
```



(a)



(b)

# 用随机投点法计算 $\pi$ 值的运行实例

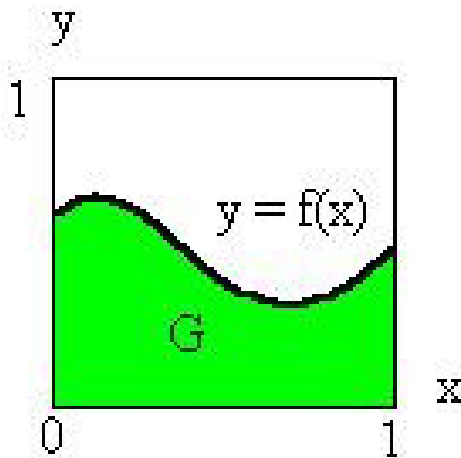
```
随机投掷500个点时, pi= 3.024  
随机投掷5000个点时, pi= 3.1808  
随机投掷50000个点时, pi= 3.14304  
随机投掷500000个点时, pi= 3.14256  
随机投掷5000000个点时, pi= 3.14208  
随机投掷50000000个点时, pi= 3.14123  
随机投掷500000000个点时, pi= 3.14155  
Press any key to continue
```

# 用随机投点法计算 $\pi$ 值的运行实例

```
随机投掷500个点时, pi= 3.176  
随机投掷5000个点时, pi= 3.1072  
随机投掷50000个点时, pi= 3.14056  
随机投掷500000个点时, pi= 3.14262  
随机投掷5000000个点时, pi= 3.14161  
随机投掷50000000个点时, pi= 3.14152  
Press any key to continue
```

# 计算定积分

- 设 $f(x)$ 是 $[0, 1]$ 上的连续函数, 且 $0 \leq f(x) \leq 1$
- 需要计算的积分为 $I = \int_0^1 f(x)dx$ , 积分 $I$ 等于图中的绿色面积 $G$



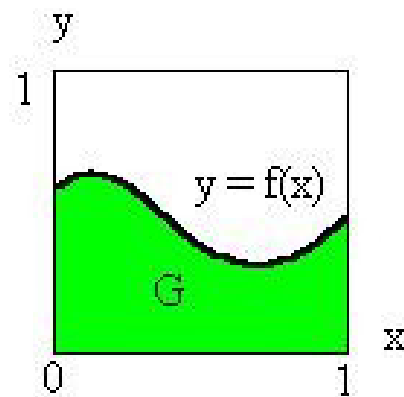
# 用随机投点法计算 $\pi$ 值

- 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  $y = f(x)$  下面的概率为：

$$P_r\{y \leq f(x)\} = \int_0^1 \int_0^{f(x)} dy dx = \int_0^1 f(x) dx$$

- 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：

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





# 计算定积分

程序代码如下：

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

# 计算定积分的运行实例

对 $g(x)=x$ ，在 $[2,4]$ 上积分值为6，下面是算法模拟10次的运行结果：

6.00203

6.01715

6.02954

5.99889

6.00622

6.00707

5.98352

6.0082

6.01024

5.97851

Press any key to continue\_

# 计算定积分的运行实例

对 $g(x)=x$ ，在 $[2,4]$ 上积分值为6，下面是算法模拟10次的运行结果：

6.02086

6.01481

6.01054

6.00136

6.01837

5.97965

5.99345

5.9922

6.00454

5.98965

Press any key to continue

# 解非线性方程组

- 求解下面的非线性方程组：

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

- 其中 $x_1, x_2, \dots, x_n$ 是实变量,  $f_i$ 是未知量 $x_1, x_2, \dots, x_n$ 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  
 $x_1^*, x_2^*, \dots, x_n^*$

# 解非线性方程组

- 方法一：线性化方法
- 方法二：求函数极小值方法

$$\Phi(x) = \sum f_i^2(x)$$

注：一般而言，概率算法费时，但设计思想简单，易实现。对于精度要求高的问题，可以提供较好的初值。

# 解非线性方程组的数值随机化算法

- 在求根区域  $D$  内, 选定随机点  $x_0$  作为随机搜索的出发点。
- 按照预先选定的分布 (正态分布、均匀分布等), 逐个选取随机点  $x_j$ , 计算目标函数, 满足精度要求的随机点就是近似解。
- 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, 生成随机搜索方向和步长, 计算出下一步的增量  $\Delta x_j$ 。从当前点  $x_j$  依  $\Delta x_j$  得到第  $j+1$  步的随机搜索点。当  $\Phi(x) < \epsilon$  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。

# 解非线性方程组的数值随机化算法

## 程序代码

```
while((min > epsilon)&&(j < Steps)){ //计算随机搜索步长因子
    if (fx < min){//搜索成功, 增大步长因子
        min = fx;// fx为目标函数值, min为当前最优值
        a* = k;// a 为步长因子
        success = true;}
    else{//搜索失败, 减少步长因子
        mm++;
        if (mm > M) a/= k; success = false;}
    for (int i = 1; i < n; i++) //计算随机搜索方向和增量
        r[i] = 2.0 * rnd.fRandom() - 1;//r为搜索方向向量 if
        (success)
            for(int i=1; i <= n; i++) dx[i] = a * r[i];
    else
        for(int i=1; i <= n; i++) dx[i] = a * r[i];
```

# 解非线性方程组的数值随机化算法

程序代码（续）

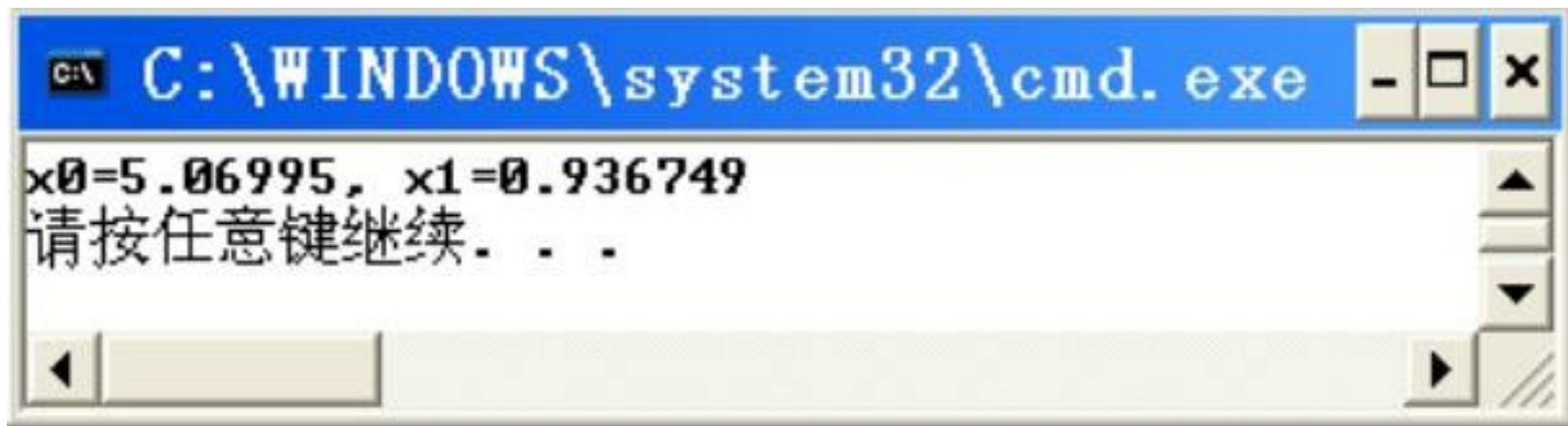
```
//计算下一个随机搜索点
    for(int i=1; i <= n; i++)
        x[i] += dx[i];
//计算目标函数值
    fx = f(x,n);
}
if (fx < epsilon)
    return true;
else
    return false;
```



# 解非线性方程组的运行实例

```
double func0(double x[],int n) {return 2*x[0]+x[1] * x[1] -11; }  
double func1(double x[],int n) {return x[0]+2*x[1] * x[1] -7;}
```

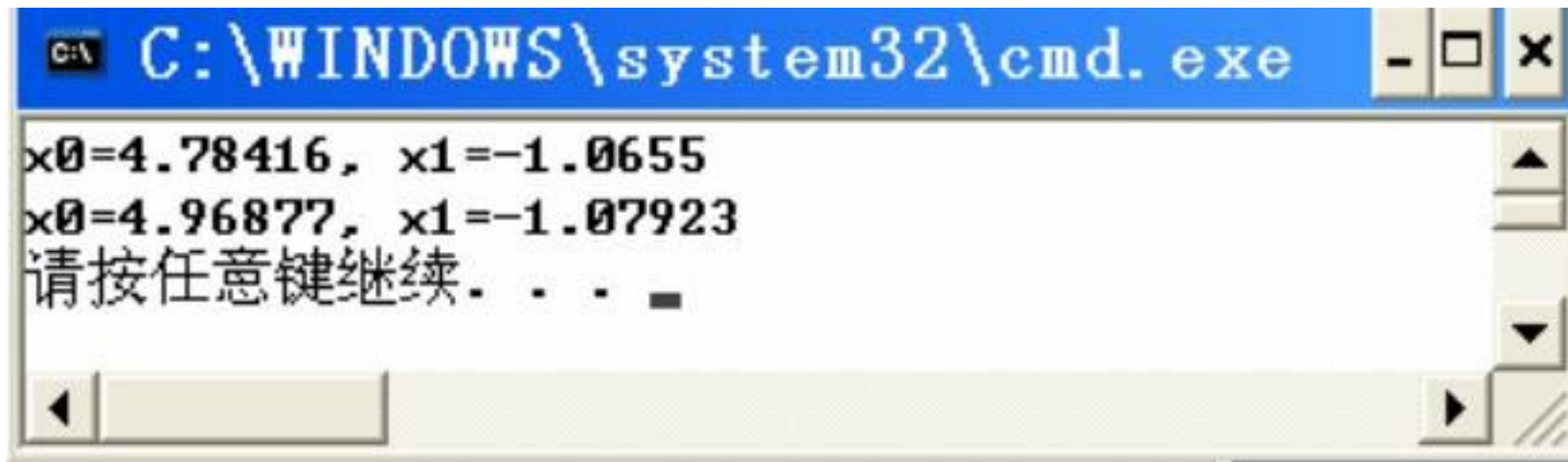
模拟求解1000次



# 解非线性方程组的运行实例

```
double func0(double x[],int n) {return 2*x[0]+x[1] * x[1] -11; }  
double func1(double x[],int n) {return x[0]+2*x[1] * x[1] -7;}
```

模拟求解1000次

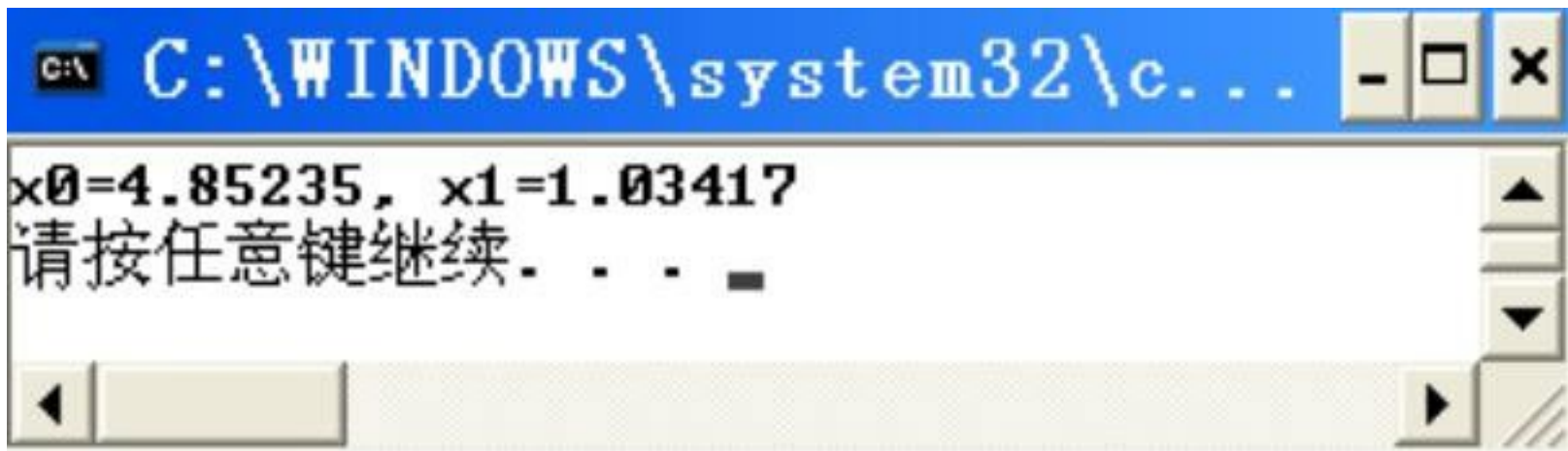


```
C:\WINDOWS\system32\cmd.exe  
x0=4.78416, x1=-1.0655  
x0=4.96877, x1=-1.07923  
请按任意键继续. . .
```

# 解非线性方程组的运行实例

```
double func0(double x[],int n) {return 2*x[0]+x[1] * x[1] -11; }  
double func1(double x[],int n) {return x[0]+2*x[1] * x[1] -7;}
```

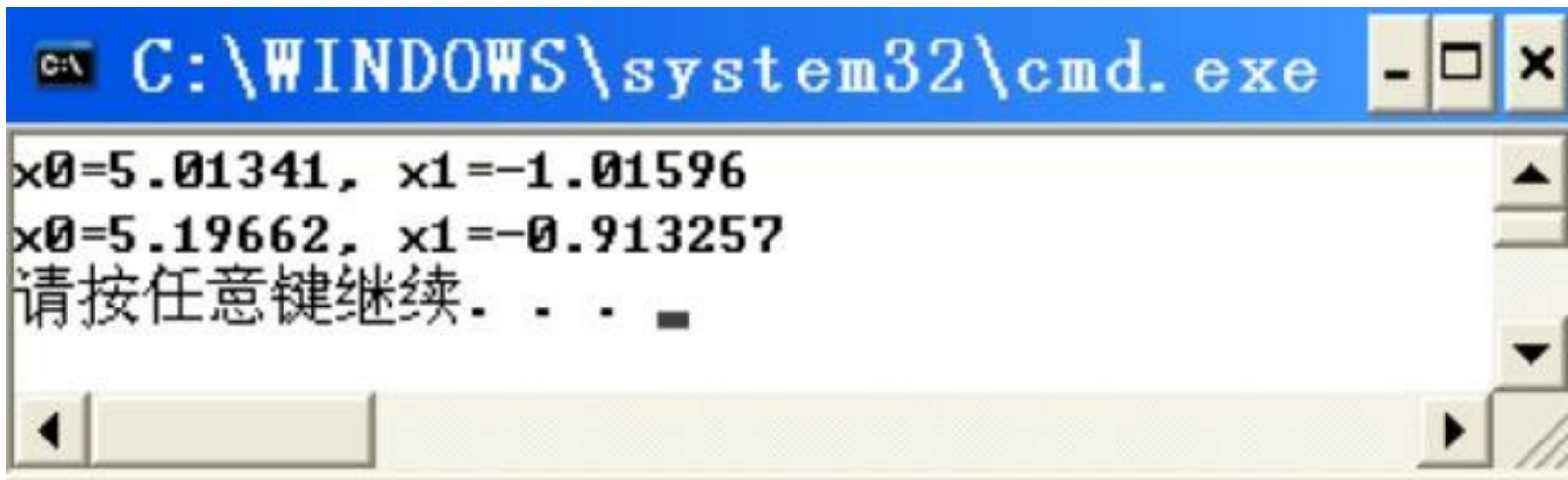
模拟求解1000次



# 解非线性方程组的运行实例

```
double func0(double x[],int n) {return 2*x[0]+x[1] * x[1] -11; }  
double func1(double x[],int n) {return x[0]+2*x[1] * x[1] -7;}
```

模拟求解1000次



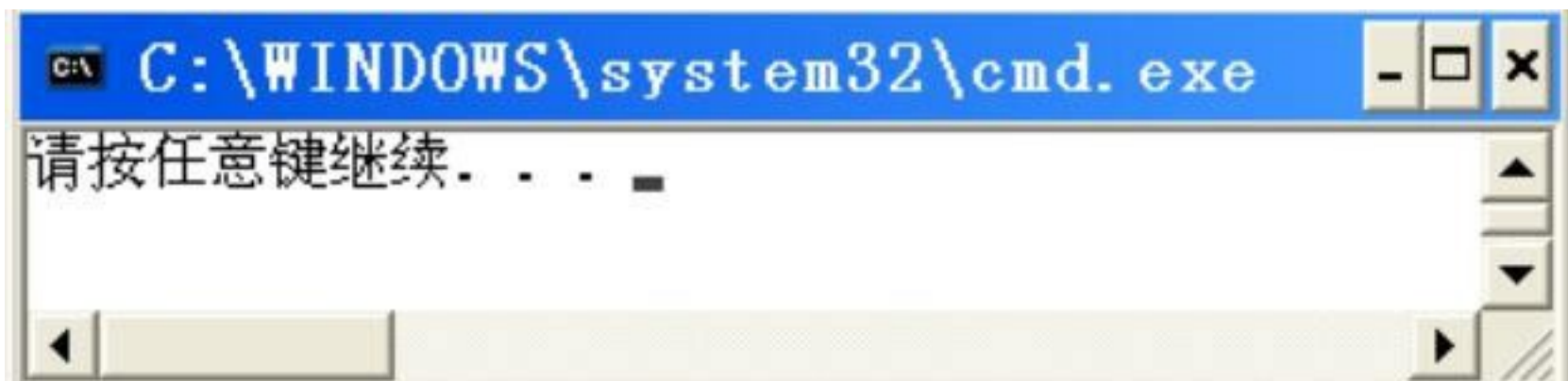
```
C:\WINDOWS\system32\cmd.exe  
x0=5.01341, x1=-1.01596  
x0=5.19662, x1=-0.913257  
请按任意键继续. . .
```

# 解非线性方程组的运行实例

```
double func0(double x[],int n) {return 2*x[0]+x[1] * x[1] -11; }
```

```
double func1(double x[],int n) {return x[0]+2*x[1] * x[1] -7;}
```

模拟求解100次(模拟次数少时可能得不到解)



# 舍伍德算法

- 总能求得问题的正确解。当一个确定性算法在最坏情况下的计算复杂度与其在平均情况下的计算复杂度两者相差较大时，可以在这个确定算法中引入随机性将它改造成一个舍伍德算法，用来消除或减少问题的不同实例之间在计算时间上的差别。舍伍德算法的精髓不是避免算法的最坏情况行为，而是设法消除这种最坏行为与特定实例之间的关联性。

# 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：

$$t_A(n) = \sum_{x \in X_n} \frac{t_A(x)}{|X_n|}$$

# 舍伍德(Sherwood)算法

- 这显然不能排除存在  $x \in X_n$  使得  $t_A(x) \gg \bar{t}_A(n)$  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。



# 舍伍德(Sherwood)算法

▪ No.1

线性时间选择算法

▪ No.2

快速排序算法

这两种算法的核心都在于选择合适的划分基准。舍伍德算法随机地选择一个数组元素作为划分基准。

# 线性时间选择算法的运行实例

```
///计算a[0: n-1]中第k小元素  
///假设a[n]是一个键值无穷大的元素  
///arr[7] = {3, 2, 5, 7, 10, INF}  
///执行Select(arr, 6, 4)后  
?  
Press any key to continue
```

# 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。

# 舍伍德(Sherwood)算法

- 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。

```
template<class Type>
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]);
    }
}
```

# 随机洗牌算法的运行实例

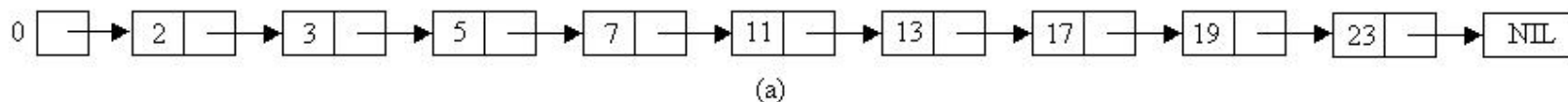
```
// 原先次序  
1 2 3 4 5 6 7 8 9 10  
// 第一次洗牌  
4 6 7 2 3 1 8 5 9 10  
// 第二次洗牌  
7 8 10 9 1 2 4 3 5 6  
Press any key to continue.
```

# 随机洗牌算法的运行实例

```
// 原先次序  
1 2 3 4 5 6 7 8 9 10  
// 第一次洗牌  
8 10 7 1 2 9 3 5 4 6  
// 第二次洗牌  
6 4 2 8 3 7 10 5 1 9  
Press any key to continue_
```

# 跳跃表

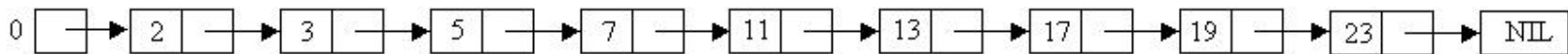
- 舍伍德型算法的设计思想还可用于设计高效的数据结构。
- 如果用有序链表来表示一个含有  $n$  个元素的有序集  $S$ ，则在最坏情况下，搜索  $S$  中一个元素需要  $\Omega(n)$  计算时间。



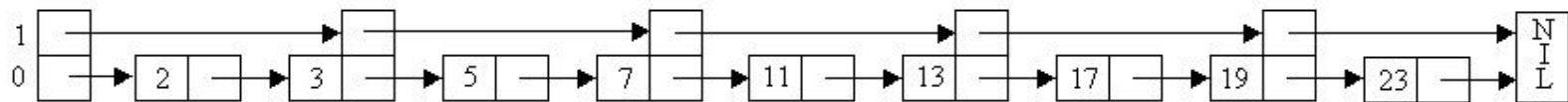
- 提高有序链表效率的一个技巧**是在有序链表的部分结点处增设**附加指针**以提高其搜索性能。在增设附加指针的有序链表中搜索一个元素时，可借助于附加指针跳过链表中若干结点，加快搜索速度。这种增加了向前附加指针的有序链表称为**跳跃表**。

# 跳跃表

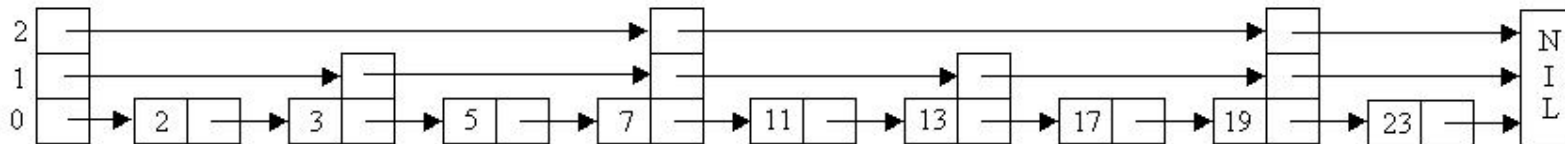
- 应在跳跃表的哪些结点增加附加指针以及在该结点处应增加多少指针完全采用随机化方法来确定。这使得跳跃表可在  $O(\log n)$  平均时间内支持关于有序集搜索、插入和删除等运算。



(a)



(b)

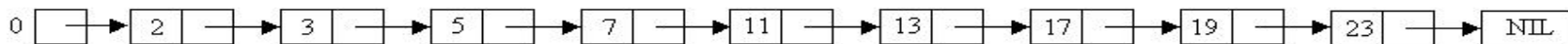


(c)

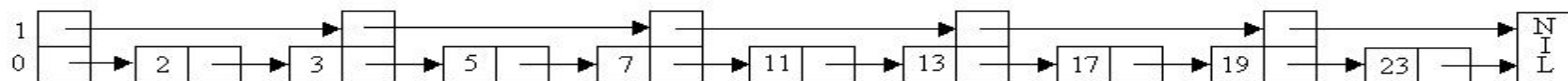


# 跳跃表

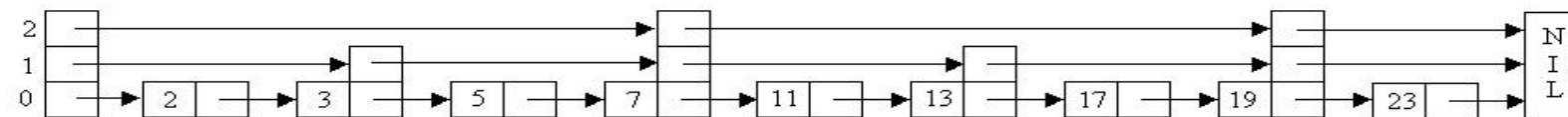
- 在一般情况下，给定一个含有  $n$  个元素的有序链表，可以把它改造成一个**完全跳跃表**，使得每一个  $k$  级结点含有  $k+1$  个指针，分别跳过  $2^k - 1, 2^{k-1} - 1, \dots, 2^0 - 1$  个中间结点。第  $i$  个  $k$  级结点安排在跳跃表的位置  $i2^k$  处,  $i \geq 0$ 。这样就可以在时间  $O(\log n)$  内完成集合成员的搜索运算。在一个完全跳跃表中，**最高级的结点是  $\lceil \log n \rceil$  级结点**。



(a)



(b)



(c)

# 跳跃表

- **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。

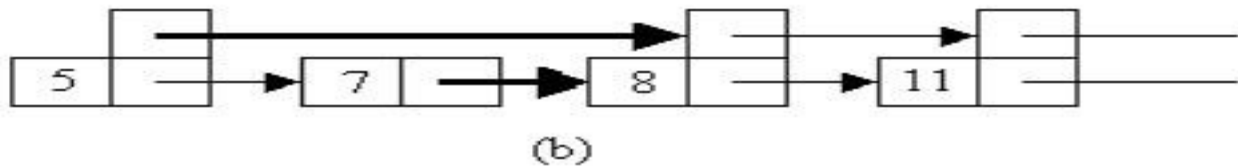
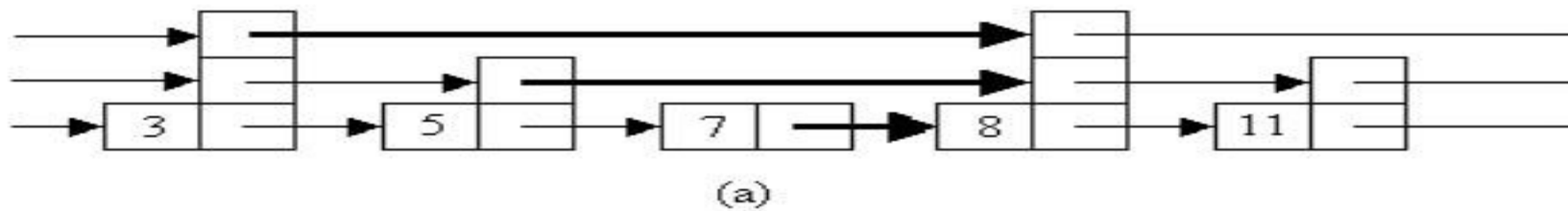
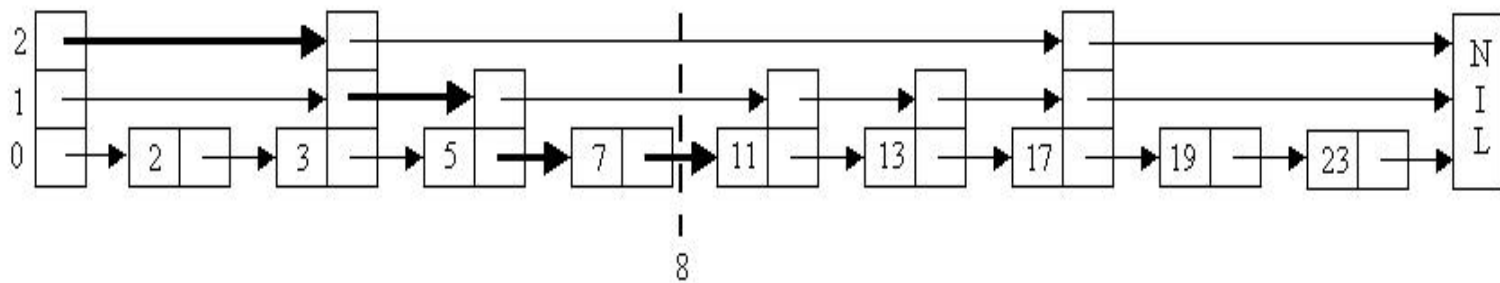
# 跳跃表

- 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。

# 跳跃表

- 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元素时，通过对跳跃表的局部修改来维持其平衡性。

# 跳跃表



# 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  $0 < p < 1$ ，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 $q \geq p$ 。

# 跳跃表

- 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 $p(1-p)$ ， $\dots$ ，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  $\log_{1/p} n$ 。

# 拉斯维加斯算法

- 求得的解总是正确的，但有时拉斯维加斯算法可能始终找不到解。一般情况下，求得正确解的概率随计算时间的增加而增大。因此，为了减少求解失败的概率，可以使用一个拉斯维加斯算法对同一实例，重复多次执行该算法。



# 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。

```
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);
}
```

# 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后问题的拉斯维加斯算法

- 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**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。

# n后问题

允许随机放置的皇后数

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

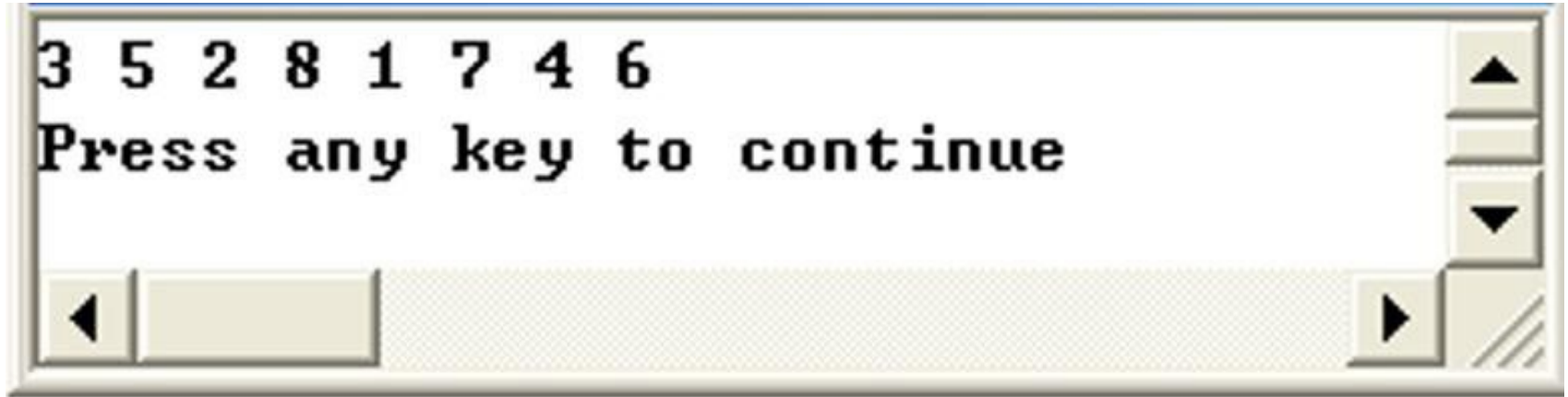
## 8后问题的运行实例



## 8后问题的运行实例



## 8后问题的运行实例





# 整数因子分解的拉斯维加斯算法

- 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：

$$n = p_1^{m_1} p_2^{m_2} \cdots p_n^{m_n}$$

- 其中， $p_1 < p_2 < \cdots < p_n$  是  $k$  个素数， $m_1, m_2, \dots, m_n$  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。

# 整数因子分解的一般算法

```
int Split(int n)
{
    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 $1 \sim x$ 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

- Pollard算法是整数因子分解的拉斯维加斯算法。
- 在开始时选取 $0 \sim n-1$ 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 $x_1, x_2, \dots, x_k, \dots$
- 对于 $i = 2^k$ ，以及 $2^k < j \leq 2^{k+1}$ ，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

```
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;
  int k=2;
  while (true) {
    i++;
    x=(x*x-1)%n; //  $x_i=(x_{i-1}^2-1) \bmod n$ 
    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;} } }
```

# Pollard算法

- 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  $p \leq \sqrt{p}$  故Pollard算法可在 $O(n^{1/4})$ 时间内找到n的一个素因子。

# Pollard算法的运行实例

```
执行Pollard(456)后  
24  
Press any key to continue
```

# Pollard算法的运行实例

执行Pollard<456>后  
19  
Press any key to continue

# Pollard算法的运行实例





# 蒙特卡罗算法

- 该法用于求问题的**准确解(或者说是精确解而不是近似解)**，但不保证所求得的解是正确的，也就是说，蒙特卡罗算法求得的解有时是错误的。不过，由于可以设法控制这类算法得到错误解的概率，并因它的**简单高效，是很有价值的一类随机算法。**

# 蒙特卡罗算法

- 一般情况下，蒙特卡罗算法求得正确解的概率随计算时间的增加而增大。但无论如何不能保证解的正确性，而且通常无法有效地判断所求得解究竟是否正确，这是蒙特卡罗算法的缺陷。

# 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。
- 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  $(1/2 + \epsilon)$  正确的蒙特卡罗算法  $2m-1$  次 ( $m = \text{floor}(n/2) + 1$ ), 得到正确解的概率至少为  $1 - \delta$ , 其中,

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

# 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：
- 当 $x \notin X$ 时， $MC(x)$ 返回的解是正确的；
- 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。

```
template<class Type>
bool Majority(Type *T, int n)
{// 判定主元素的蒙特卡罗算法
    int i=rnd.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
}
```

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

# 主元素问题

- 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 $\varepsilon$ 。算法MajorityMC所需的计算时间显然是 $O(n \log(1/\varepsilon))$ 。



# 主元素问题的运行实例

- **main**重复调用10次**MajorityMC**, 而**MajorityMC** 重复调用**k**次  
**Majority**算法
- 本例中 $\text{int } k = \text{ceil}(\log(1/e)/\log(2)); // e=0.1, k=4$

```
int main()
{
    int arr[10] = {3, 5, 7, 3, 9, 3, 3, 1, 3, 3};
    for (int var=0; var<10; var++)
        cout << boolalpha << MajorityMC(arr, 10, 0.1) << endl;
    return 0;
}
```





# 主元素问题的运行实例

```
true  
false  
true  
true  
true  
true  
true  
false  
true  
true  
Press any key to continue
```

# 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 $(n-1)! \equiv -1 \pmod{n}$ 。
- **费马小定理**: 如果 $p$ 是一个素数, 且 $0 < a < p$ , 则 $a^{p-1} \equiv 1 \pmod{p}$ 。
- **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

```
void power( unsigned int a, unsigned int p,  
            unsigned int n, unsigned int &result,  
            bool &composite)  
{// 计算 $a^p \bmod n$ , 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; }}
```

# 素数测试

```
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;
}
```

# 素数测试

```
bool PrimeMC(unsigned int n)
{ // 重复调用k次Prime算法的蒙特卡罗算法
    RandomNumber rnd;
    unsigned int a, result;
    bool composite=false;
    for(int i=1;i<=k;i++)
    { a=rnd.Random(n-3)+2;
      power(a,n-1,n,result,composite);
      if (composite||(result!=1)) return false;
    }
    return true;
}
```



# 素数测试

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

# 素数测试的运行实例

```
int main()
{
    unsigned int n;
    while(cin>>n)
    {
        if (PrimeMC(n))
            cout<<"YES"<<endl;
        else
            cout<<"NO"<<endl;
    }
    return 0;
}
```

# 素数测试的运行实例

```
29  
YES  
13  
YES  
457  
YES  
789  
NO  
987  
NO  
^Z  
Press any key to continue
```

# 素数测试的运行实例

对 $g(x)=x$ ，在 $[2,4]$ 上积分值为6，下面是算法模拟10次的运行结果：

6.00203

6.01715

6.02954

5.99889

6.00622

6.00707

5.98352

6.0082

6.01024

5.97851

Press any key to continue\_