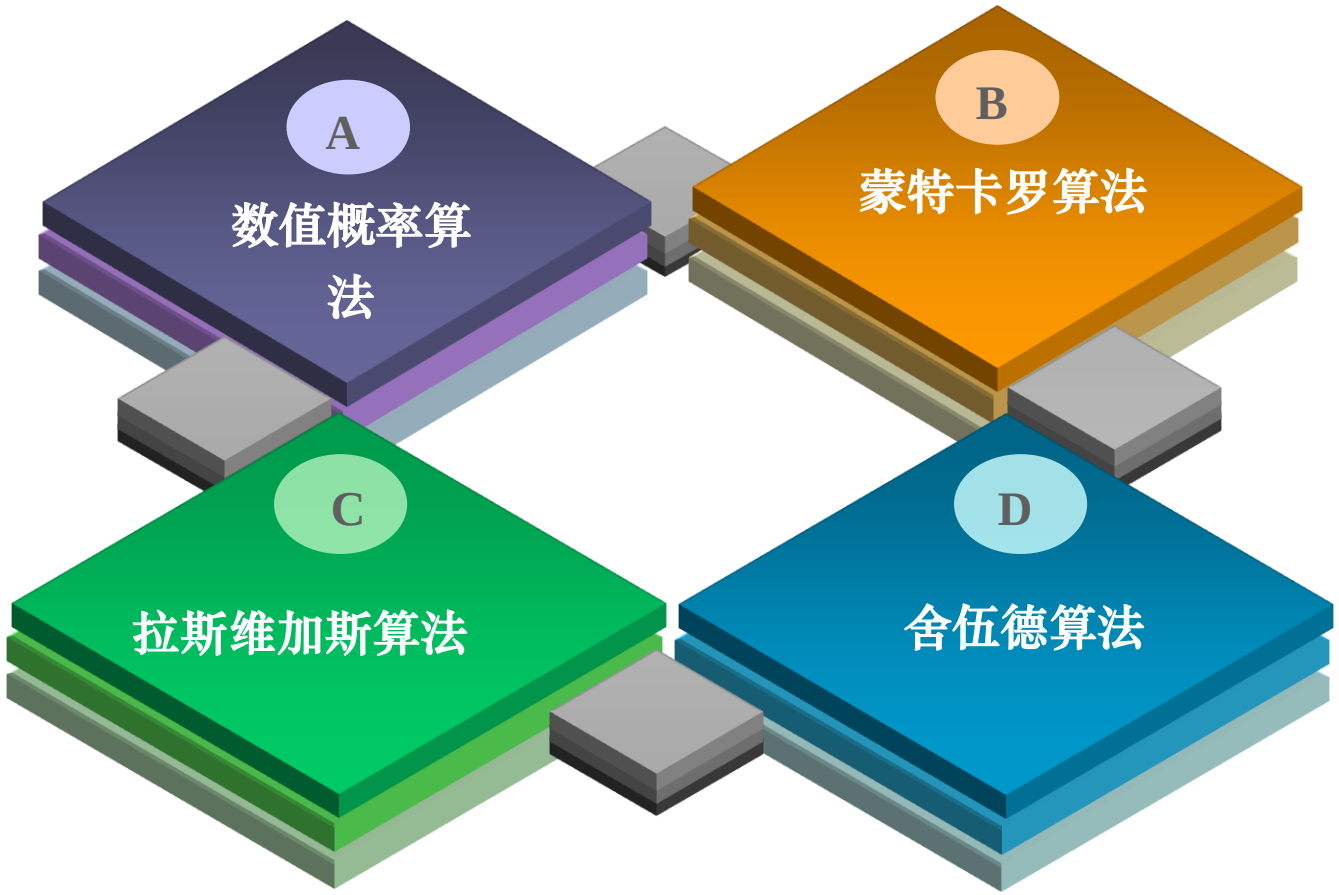
第7章 概率算法

- **学习要点**
- 理解产生伪随机数的算法
- 掌握数值概率算法的设计思想
- 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

概率算法的特点

1. 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。
2. 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。
3. 设计思想简单，易于实现。

概率算法的分类



随机算法的分类

- 数值概率算法常用于数值问题的求解。将一个问题的计算与某个概率分布已经确定的事件联系起来，求问题的近似解。这类算法所得到的往往是近似解，且近似解的精度随计算时间的增加而不断提高。在许多情况下，要计算出问题的精确解是不可能或没有必要的，因此可以用数值随机化算法得到相当满意的解。

- 蒙特卡罗算法用于求问题的准确解，但得到的解未必是正确的。蒙特卡罗算法以正的概率给出正解，求得正确解的概率依赖于算法所用的时间。算法所用的时间越多，得到正确解的概率就越高。一般给定执行步骤的上界，给定一个输入，算法都是在一个固定的步数内停止的。

随机算法的分类

- 拉斯维加斯算法不会得到不正确的解。一旦用拉斯维加斯算法找到一个解，这个解就一定是正确解。但有时可能找不到解。拉斯维加斯算法找到正确解的概率随着它所用的计算时间的增加而提高。对于所求解问题的任意实例，用同一拉斯维加斯算法反复对它求解，可以使求解失效的概率任意小。

- 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。当一个确定性算法在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，消除或减少问题的好坏实例间的这种差别（精髓所在）。

7.1 随机数

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。

Gcd: greatest common divider

```
class RandomNumber{
private:
        unsigned long randSeed;
public:
        RandomNumber(unsigned long s=0);
        unsigned short Random(unsigned long n);
        double fRandom(void);
};
```

```
RandomNumber::RandomNumber(unsigned long s/*=0*/)
{
        if (s == 0)
                randSeed = time(0);
        else
                randSeed = s;
}

unsigned short RandomNumber::Random(unsigned long n)
{
        randSeed = multiplier*randSeed+adder;
        return (unsigned short)((randSeed>>16) % n);
}

double RandomNumber::fRandom(void)
{
        return Random(maxshort)/double(maxshort);
}
```

下面用计算机产生的伪随机数来模拟抛硬币实验。假设抛10次硬币构成一个事件。调用Random(2)返回一个二值结果。返回0表示抛硬币得到反面，返回1表示得到正面。下面的算法TossCoins模拟抛10次硬币这一事件50000次。用head[i] (0≤i≤10)记录这50000此模拟恰好得到i次正面的次数。最终输出模拟抛硬币事件得到正面事件的频率图。

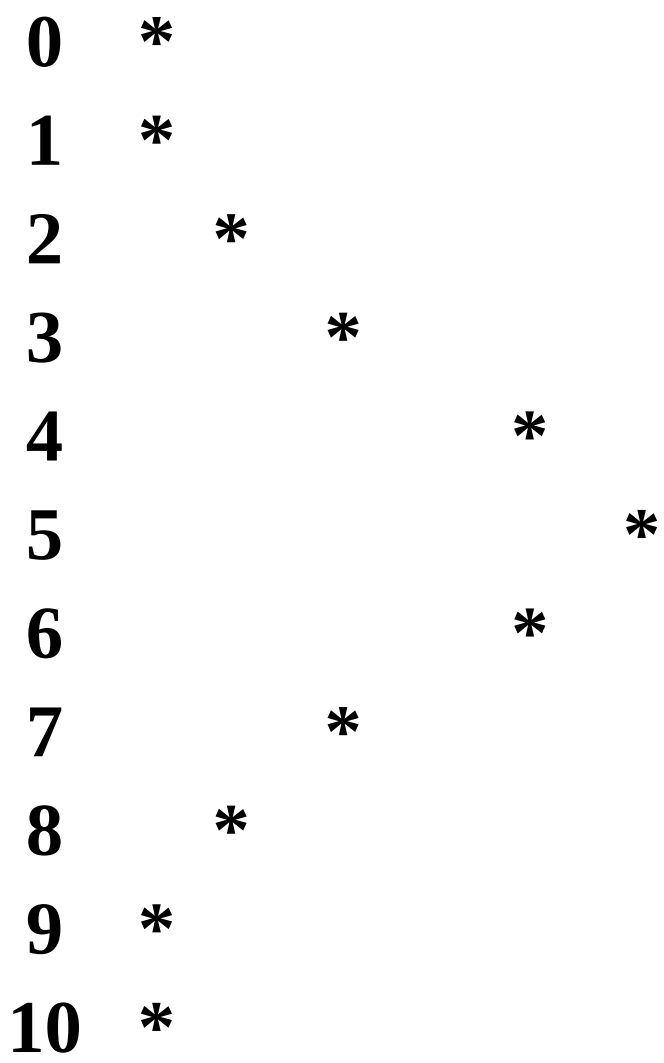
head tail

head tail

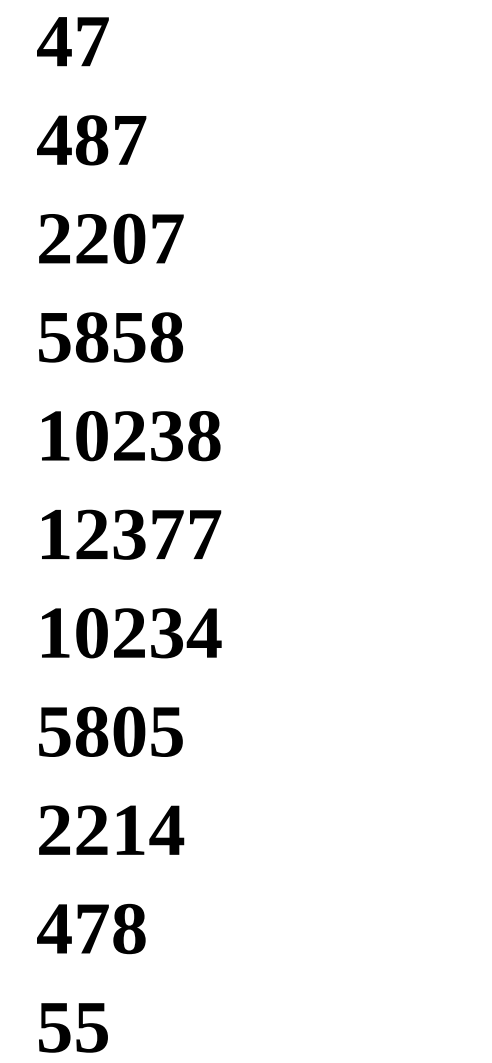
```
void main (void)
{ // 模拟随机抛硬币事件
        const int NCOINS = 10;
        const long NTOSSES = 50000L;
        // heads[i]是得到i次正面的次数
        long i, heads[NCOINS+1];
        int j, position;
        // 初始化数组heads
        for (j=0;j< NCOINS+1; j++)
                heads[j] = 0;
        // 重复50000次模拟事件
        for (i=0;i< NTOSSES; i++)
                heads[TossCoins(NCOINS)]++;
        // 输出频率图
        for (i=0;i<=NCOINS; i++)
        {
                position = int (float (heads[i])/NTOSSES*72);
                cout<<setw(6)<<i<<" ";
                for (j=0;j<position-1;j++)
                        cout<<" ";
                cout<<"*"<<endl;
        }
}
```

```
int TossCoins (int numberCoins)
{ //随机抛硬币
  static RandomNumber coinToss;
  int  i, tosses = 0;
  for (i = 0; i < numberCoins; i++)
      // Random (2) = 1 表示正面
      tosses += coinToss.Random (2);
  return tosses;
}
```

```
int TossCoins (int numberCoins)
{ //随机抛硬币
   static RandomNumber coinToss;
   int  i, tosses = 0;
   for (i = 0; i < numberCoins; i++)
         // Random (2) = 1 表示正面
         tosses += coinToss.Random (2);
   return tosses;
}
```



模拟抛硬币得到的正面事件频率图



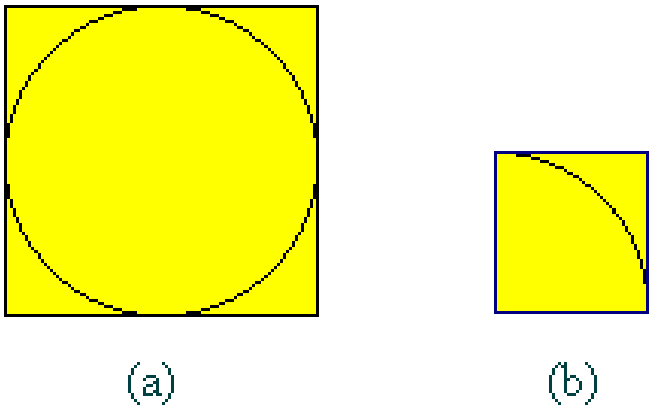
模拟抛硬币得到的次数

7.2 数值随机化算法

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。从而 $\pi \approx \frac{4k}{n}$

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



7.2 数值随机化算法

实验次数	计算结果
13350000000	3.141592841947565 6
13360000000	3.14159316676646 7
13370000000	3.141592942707554 4
13380000000	3.141593063378176 3
13390000000	3.141592515907393 4
13400000000	3.14159208985074 6
13410000000	3.141592214765100 5
13420000000	3.14159214783904 6
13430000000	3.141592191511541 5
13440000000	3.14159239375
13450000000	3.141592264981412 5

数学上可以证明，蒙特卡洛法每提高一位精度，需要提高100倍的计算量，因此达到上述小数点后面5位的精度如果花了1小时的话，6位精度要100小时，7位要一年多，8位要100多年。可见，用蒙特卡洛方法要把π算到小数点后面10位以上是很艰难的。即使用效率高得多的C语言，同样时间内也顶多再提高一两位的精度。即使放到大型计算机上去，也提高不了几位。

7.2 数值随机化算法

梅钦级数法

$$\pi = 16arctan(1/5) - 4arctan(1/239)$$

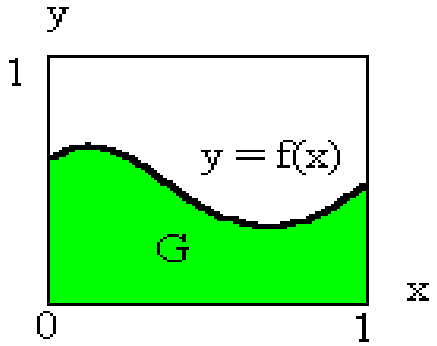
$$\frac{\pi}{4} = 4 \times (\frac{1}{1 \times 5^1} - \frac{1}{3 \times 5^3} + \frac{1}{5 \times 5^5} - \frac{1}{7 \times 5^7} + ..)$$
$$- (\frac{1}{1 \times 239^1} - \frac{1}{3 \times 239^3} + \frac{1}{5 \times 239^3} - \frac{1}{7 \times 239^7} + ..)$$

为了能做到高精度快速计算，我们把所有的级数都变成整数运算，因此，如果我们希望计算到π的小数点后面n位，我就在上述公式两边乘以10的n次方，并全部运行整除运算（运算符号是“//”）。最后我们得到的一个π乘以10^n所形成的一个巨大的整数来表示计算结果

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。



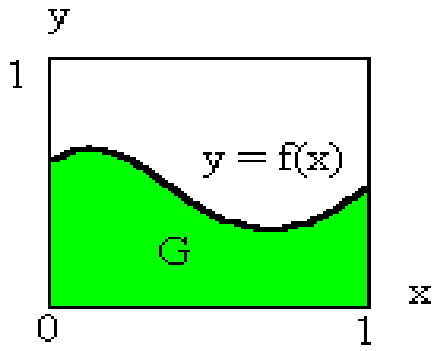
在图所示单位正方形内均匀地作投点试验，则随机点落在曲线下面的概率为

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

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

计算定积分

```
public  static double darts1(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);
}
```



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，$x_1,x_2,\cdots,x_n$是实变量，f_i是未知量$x_1,x_2,\cdots,x_n$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 $x_1^*, x_2^*, \cdots, x_n^*$

解非线性方程组

- **线性化方法**
- **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

在求根区域D内，选定随机点x_0作为随机搜索的出发点。

1. 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。
2. 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。

解非线性方程组

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;
        a* = k;
        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;
    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]-dx[i];
```

解非线性方程组

程序代码（续）

```
//计算下一个搜索点
        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;
```

7.3 舍伍德(Sherwood)算法

作用:

消除算法所需计算时间与输入实例间的联系。

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:

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

7.3 舍伍德(Sherwood)算法

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。

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

学过的Sherwood算法:

（1）快速排序算法

（2）线性时间选择算法

当所给的确定性算法无法直接改造成舍伍德型算法时，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。

回顾快速排序

视频来源：BiliBili，Up:老陈打码，BV1K44y1k79z

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。

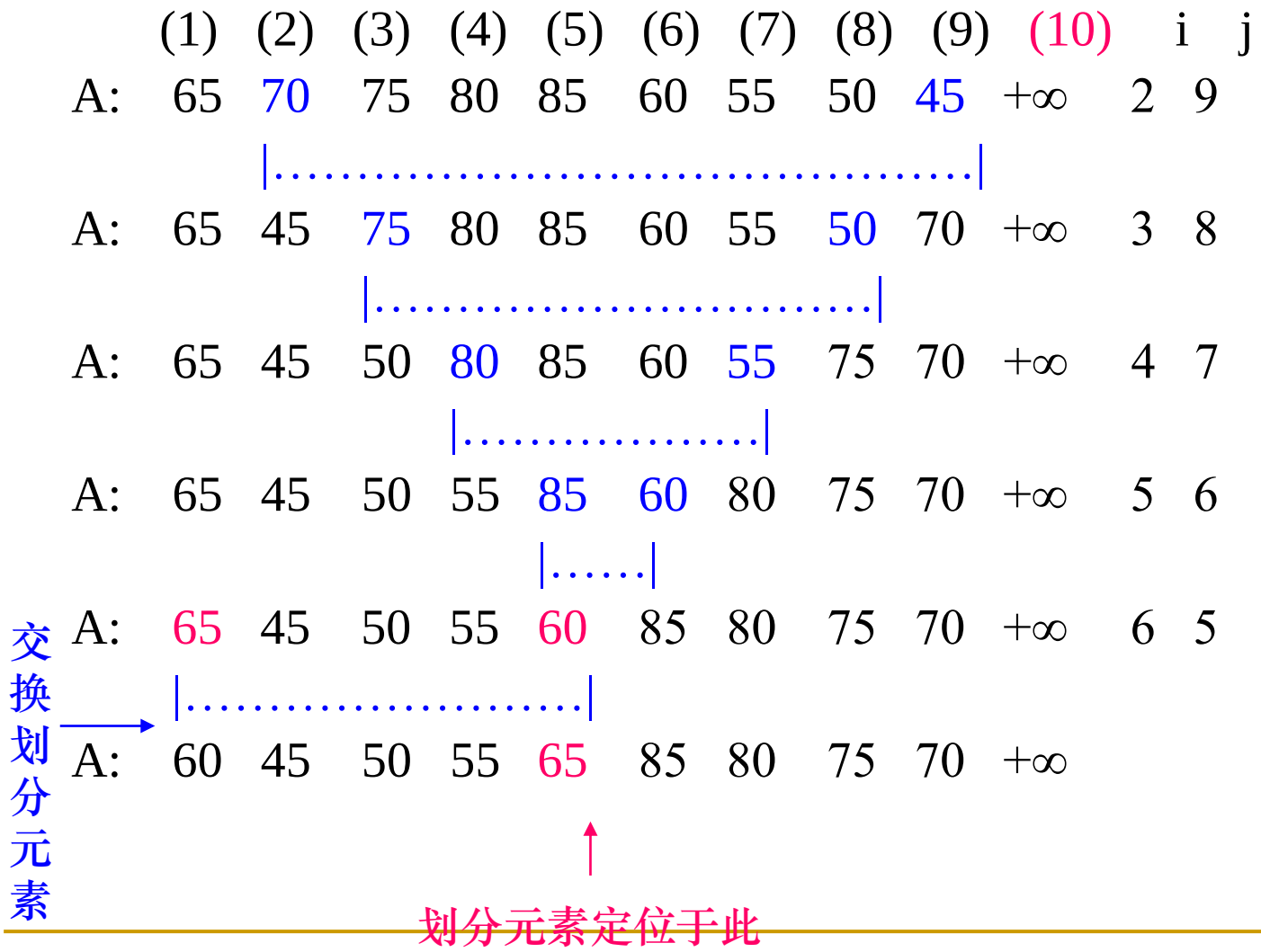
```
public static void shuffle(Comparable []a, int n)
{// 随机洗牌算法
  rnd=new Random();
  for (int i=1;i<n;i++) {
    int j=rnd.Random(n-i+1)+i;
    Swap(a[i], a[j]);
    }
}
```

划分过程（partition）的算法描述

用元素 a[p] 划分集合 a[p : r]

```
public static int partition (int p, int r)
{
    int i = p,  j = r + 1;
    Comparable x = a[p];
    // 将>= x的元素交换到左边区域，<= x的元素交换到右边区域
   while (true) {
        while (a[++i].compareTo(x) < 0); // i由左向右移
        while (a[--j].compareTo(x) > 0);  // j由右向左移
        if (i >= j) break;
        MyMath.swap(a, i, j);
         }
        a[p] = a[j];
        a[j] = x;
     return j;   //划分元素在位置j
    }
```

例子 划分实例


(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) i j
A: 65 70 75 80 85 60 55 50 45 +∞ 2 9
A: 65 45 75 80 85 60 55 50 70 +∞ 3 8
A: 65 45 50 80 85 60 55 75 70 +∞ 4 7
A: 65 45 50 55 85 60 80 75 70 +∞ 5 6
A: 65 45 50 55 60 85 80 75 70 +∞ 6 5
交换划分元素
A: 60 45 50 55 65 85 80 75 70 +∞
划分元素定位于此

快速排序算法

经过一次“划分”后，实现了对集合元素的调整：其中一个子集合的所有元素均小于等于另外一个子集合的所有元素。

按同样的策略对两个子集合进行分类处理。当子集合分类完毕后，整个集合的分类也完成了。这一过程避免了子集合的归并操作。这一分类过程称为快速分类。

通过反复使用划分算法 **partition** 实现对集合元素的排序。

以a[p]为基准元素将a[p:r]划分成3段:

a[p:q-1]，a[q]，a[q+1:r]，

使得：a[p:q-1]中任何元素小于等于a[q]，

a[q+1:r]中任何元素大于等于a[q]，

下标q在划分过程中确定。

```
public  static void qSort(int p, int r)
 {
    if (p<r) {
    int q=partition(p,r);
     qSort (p,q-1); //对左半段排序
     qSort (q+1,r); //对右半段排序
     }
 }
```

在快速排序中，记录的比较和交换是从两端向中间进行的，关键字较大的记录一次就能交换到后面单元，关键字较小的记录一次就能交换到前面单元，记录每次移动的距离较大，因而总的比较和移动次数较少。

快速排序算法的性能取决于划分的对称性。通过修改算法**partition**，可以设计出采用随机选择策略的快速排序算法。在快速排序算法的每一步中，当数组还没有被划分时，可以在a[p:r]中随机选出一个元素作为划分基准，这样可以使划分基准的选择是随机的，从而可以期望划分是较对称的。

```
public static int randomizedPartition (int p, int r)
 {
    int i = random(p,r);
    MyMath.swap(a, i, p);
    return partition (p, r);
    }
public static void randomizedQuickSort (int p, int r)
 {
    if( p<r ) {
    int q = randomizedPartition(p,r);
    randomizedQuickSort(p, q-1);//对左半段排序
    randomizedQuickSort(q+1, r); //对右半段排序
    }
}
```

线性时间选择

利用**partition**函数。如果划分元素v测定在A(j)的位置上，则有j-1个元素小于或等于A(j)，且有n-j个元素大于或等于A(j)。此时，

- 若k=j，则A(j)即是第k小元素；否则，
- 若k<j，则第k小元素将出现在A(0:j-1)中；
- 若k>j，则第k小元素将出现在A(j+1:n-1)中。

选择算法

- 算法描述

```
public static void randomizedSelect (int p, int r,int k)
{
   if( p==r ) return a[p];
   int i = randomizedPartition(p,r),
        j=i-p+1;
   if( k<=j )  return randomizedSelect (p, i, k);//在左半段
   else return randomizedSelect (i+1, r, k-j); //在右半段
}
```

教材216页

搜索有序表

用两个数组来表示所给的含有n个元素的有序集S。用value[0:n]存储有序集中的元素，link[0:n]存储有序集中元素在数组value中位置的指针。Link[0]指向有序集中第1个元素，即value[link[0]]是集合中的最小元素。一般地，如果value[i]是所给有序集S中的第k个元素，则value[link[i]]是S中的第k+1个元素。S中元素的有序性表现为，对于任意1≤i≤n有value[i]≤ value[link[i]]。对集合S中的最大元素value[k]有，link[k]=0且value[0]是一个大数。例如，有序集S={1,2,3,5,8,13,21}的一种表示方式如下图:

i	0	1	2	3	4	5	6	7
Value[i]	∞	2	3	13	1	5	21	8
Link[i]	4	2	5	6	1	7	0	3

在此例中，link[0]=4指向S中最小元素value[4]=1。显而易见，这种表示有序集的方法实际上是用数组来模拟有序链表。对于有序链表，可采用顺序搜索的方式在所给的有序集S中搜索链值为x的元素。如果有序集S中含有n个元素，则在最坏情况下，顺序搜索算法所需的计算时间为O(n)。

i	0	1	2	3	4	5	6	7
Value[i]	∞	2	3	13	1	5	21	8
Link[i]	4	2	5	6	1	7	0	3

利用数组下标的索引性质，可以设计一个随机化搜索算法，以改进算法的搜索时间复杂性。算法的基本思想是，随机抽取数组元素若干次，从较接近搜索元素x的位置开始做顺序搜索。可以证明，如果随机抽取数组元素k次，则其后顺序搜索所需的平均比较次数为O(n/(k+1))，如果k=√n，则比较次数O(√n)。

为实现上述算法，用数组来表示的有序链表需通过一个OrderedList类来进行定义，其中包含有Search、Insert、Delete等多个函数。在此类中，Small和TailKey分别是全集合中元素的下界和上界， MaxLength是集合中元素个数的上限。

OrderedList类的共享成员函数Search用来搜索当前集合中的元素X。当搜索到元素X时，将该元素在数组value中的位置范围到index中，并返回true，否则返回false。

```
template <class Type>
bool OrderedList <Type>::Search(Type x,int& index)
{ // 搜索集合中指定元素 k
        index = 0;
        Type max = Small;
        int m = floor(sqrt(double(n))); //随机抽取数组元素次数
        for (int i=1; i<=m; i++) {
                int j = rnd.Random(n)+1; //随机产生数组元素位置
                Type y=value[j];
                if( (max<y) && (y<x) ) {
                        max = y;
                        index = j;}
        }
        // 顺序搜索
        while ( value[link[index]] < x )  index = link[index];
        return (value[link[index]] == x);
}
```

插入运算，插入数字k至第n个位置

i	0	1	2	3	4	5	6	7	8
Value[i]	∞	2	3	13	1	5	21	8	10
Link[i]	4	2	5	6	1	7	0	3	

搜索出index=7

Link[n]=index

i	0	1	2	3	4	5	6	7	8
Value[i]	∞	2	3	13	1	5	21	8	10
Link[i]	4	2	5	6	1	7	0	3	3

Link[index]=n

i	0	1	2	3	4	5	6	7	8
Value[i]	∞	2	3	13	1	5	21	8	10
Link[i]	4	2	5	6	1	7	0	8	3

对于插入运算，首先用Search函数确认待插入元素K不在当前集合中，然后将新插入的元素存储在value[n+1]中，并修改相应的指针。

```
template <class Type>
void OrderedList <Type>::Insert(Type k)
{ // 插入集合中指定元素
        if ( (n==MaxLength) || (k>=TailKey) ) return;
        int index;
        if (!Search(k,index)) { //小于k的最大数的index
                value[++n] = k; //第n+1个value加进去
                link[n] =link[index];
                 //新的指向下一个的link更新为index本身对应的link
                link[index] = n;} //index本身对应的link更新为n
}
```

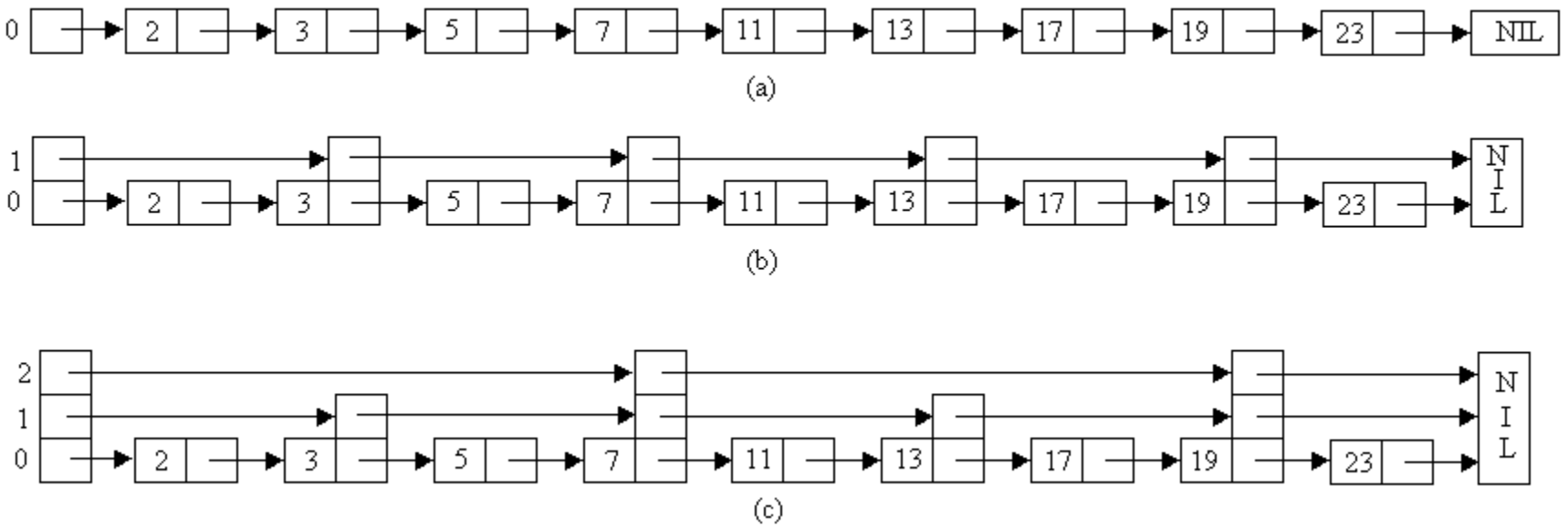
删除元算首先用函数Search找到待删除元素K在当前集合中的位置，然后修改待删除元素K的前驱元素的link指针，使其指向待删除元素K的后继元素。被删除元素K在有序表中产生的空洞，由当前集合中的最大元素来补填。搜索当前集合中的最大元素的任务由函数SearchLast来完成。

教材218页

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。
- 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。

跳跃表



搜索元素8，从最高级（2）开始。

跳跃表

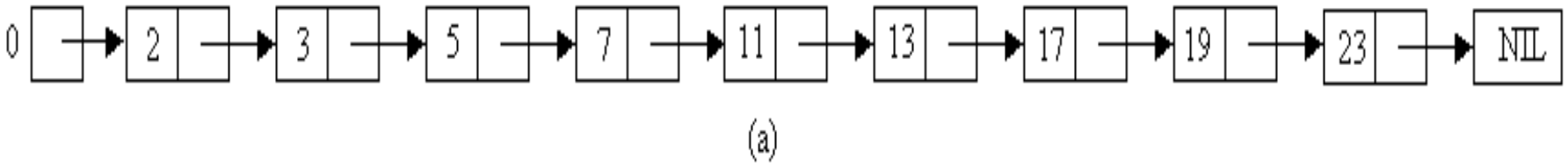
- 应在跳跃表的哪些结点增加附加指针？
- 在该结点处应增加多少指针？

- 完全采用随机化方法来确定。
- 使跳跃表可在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。
- k 级结点：含有k+1个指针的结点

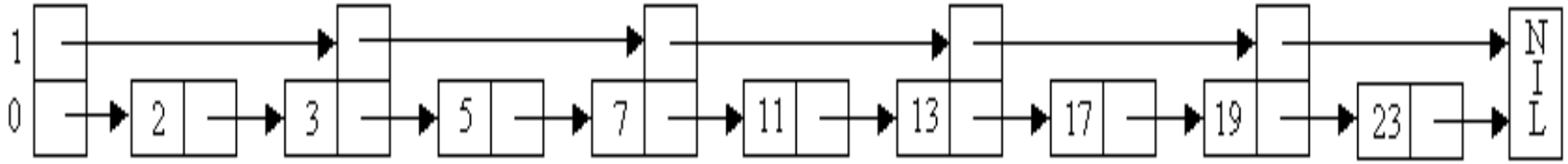
在一般情况下，给定一个含有n个元素的有序链表，可以将它改造成一个完全跳跃表。使得每一个k级结点含有k+1个指针，分别跳过2^k-1，$2^{k-1}-1$，…，2^0-1个中间结点。第i个k级结点安排在跳跃表的位置 i x 2^k处，i≥0。这样就可以在时间O(logn)内完成集合成员的搜索运算。在一个完全跳跃表中，最高级的结点是logn级结点。

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。

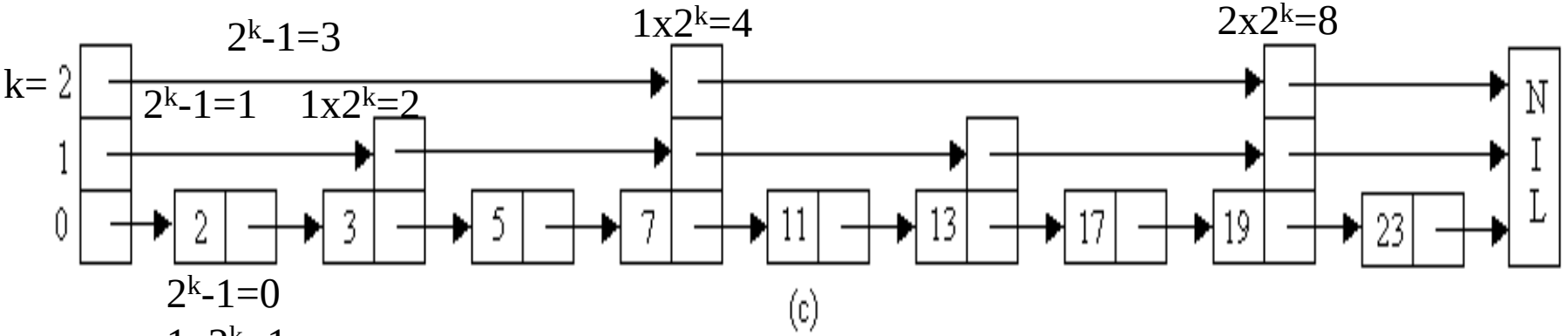
实例：n=9


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)


k= 2
1
0
2^k-1=3
2^k-1=1
1x2^k=2
2^k-1=0
1x2^k=1
1x2^k=4
2x2^k=8
2
3
5
7
11
13
17
19
23
NIL

(c)

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。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。

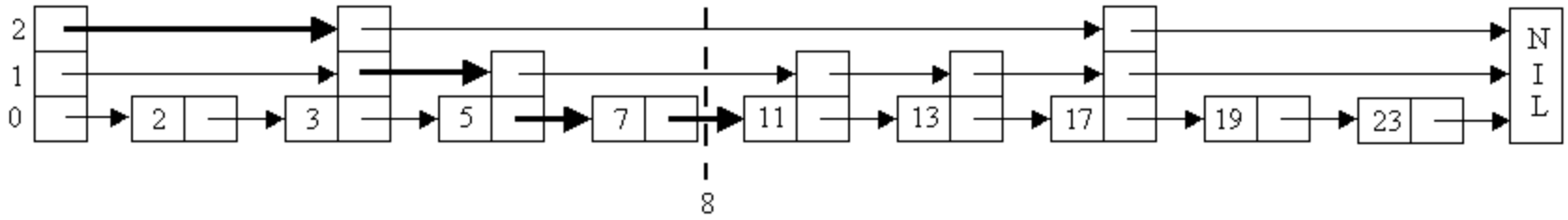
因此，在插入一个元素时：

以概率1/2引入一个0级结点；

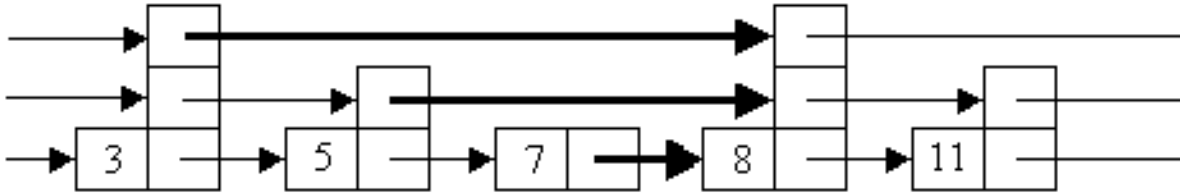
以概率1/4引入一个1级结点；…；

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。

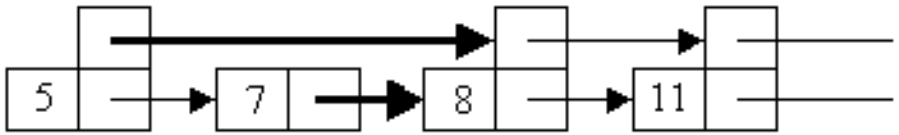


(a)元素8作为二级结点插入



(a)

(b)元素8作为一级结点插入



(b)

如何随机地生成新插入结点的级别?

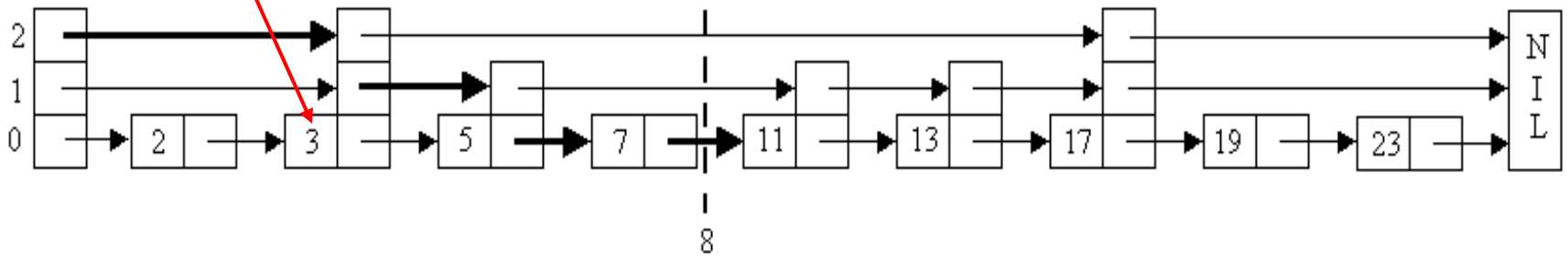
分析:

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。

步骤：

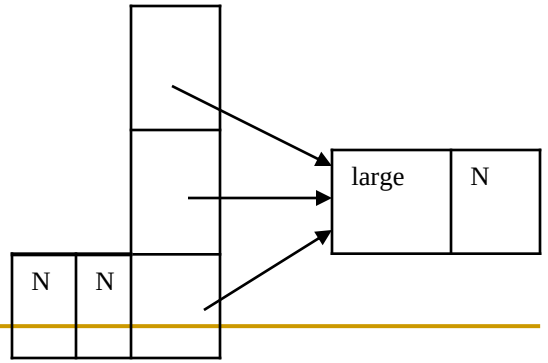
- 在插入一个新结点时，先将其结点级别初始化为0
- 然后用随机数生成器反复地产生一个[0,1)间的随机实数q。
  - 如果q<p，则使新结点级别增加1，直至q≥p。由此产生新结点级别的过程可知，所产生的新结点的级别为0的概率为1-p，级别为1的概率为p(1-p)，…，级别为i的概率为$p^i(1-p)$。
  - 如此产生的新结点的级别有可能是一个很大的数，甚至远远超过表中元素的个数。为了避免这种情况，用$\log_{1/p}n$ 作为新结点级别的上界。其中n是当前跳跃表中结点个数。当前跳跃表中任一结点的级别不超过 $\log_{1/p}n$

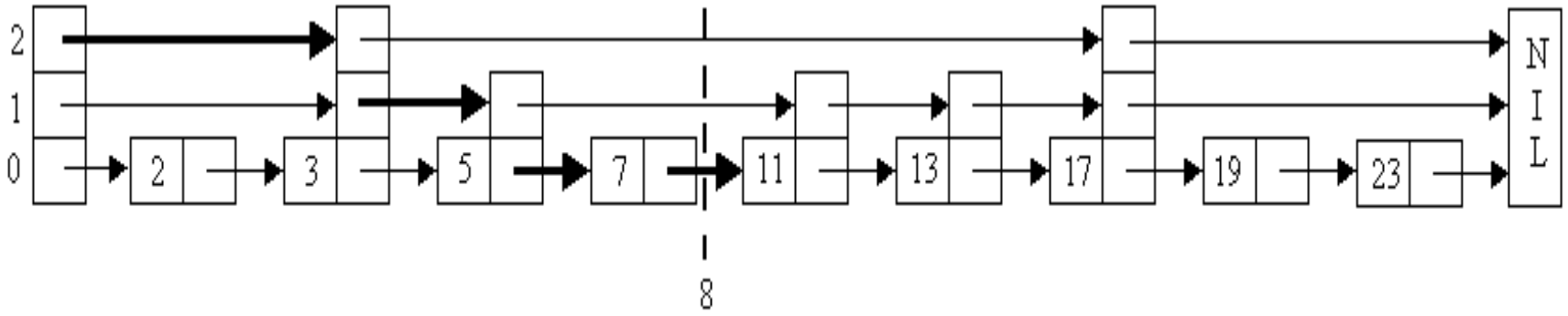
```
protected static class SkipNode
{
 protected Comparable key;
 protected Object element;
 protected SkipNode [] next; //指针数组
 protected SkipNode(Object k, Object e, int size)
 {//构造方法
   key=(Comparable)k;
   element=e;
   next=new SkipNode[size];        }
}
```



```
public  class SkipList
{
 protected float prob;//用于分配结点级别
 protected int maxLevel;//跳跃表级上界
 protected int levels; //当前最大级别
 protected int size;    //当前元素个数
 protected Comparable tailKey; //元素键值上界
 protected SkipNode head; //头结点指针
 protected SkipNode tail; //尾结点指针
 protected SkipNode [] last; //指针数组
 protected Random r; //随机数产生器
```

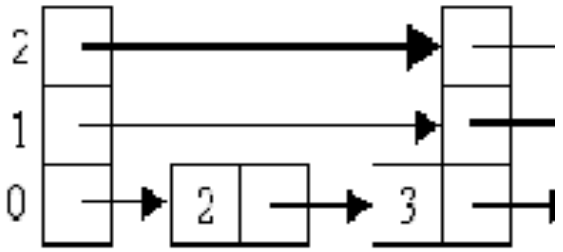
```
//构造方法……
public SkipList(Comparable large, int maxE, float p)
{
  prob=p;
  maxLevel=(int)Math.round(Math.log(maxE)/Math.log(1/prob))-1;
  tailKey=large;
  //创建头、尾结点和数组last
head=new SkipNode(null,null,MaxLevel+1);
tail =new SkipNode(tailKey,null,0);
last =new SkipNode[MaxLevel+1];
  //初始化跳跃表为空表
for (int i=0;i<= MaxLevel;i++)
head.next[i]=tail;
r=new Random( );
}
```



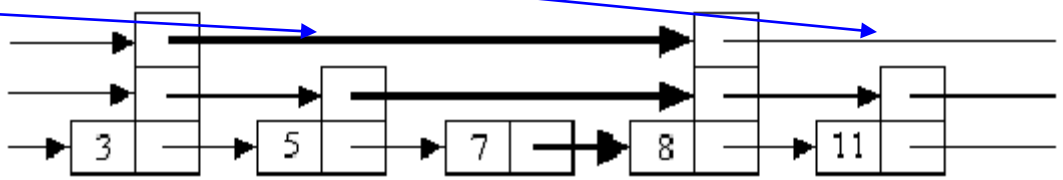

2
1
0
2
3
5
7
11
13
17
19
23
NIL
8

在跳跃表中搜索一个元素：

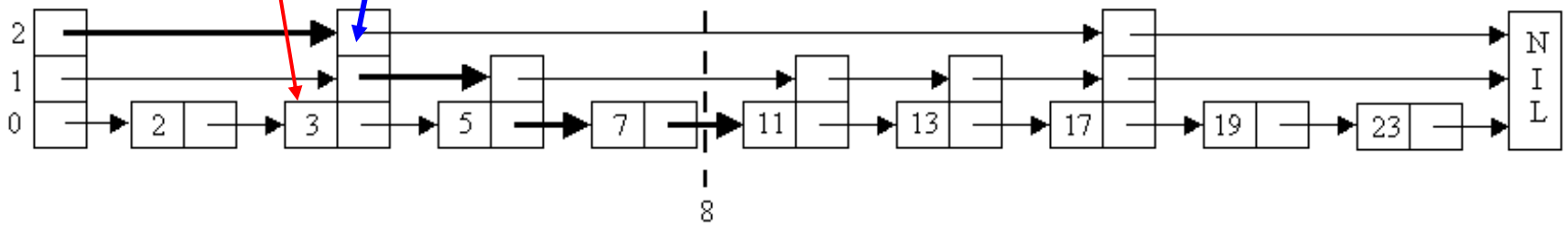
```
SkipNode search(Object k)
{//从最高级到0级指针搜索指定元素k
  SkipNode p=head;
for(int i=levels; i>=0; i--) {
    while(p.next[i].key.compareTo(k)<0) //在第i级链中搜索
        p=p.next;
    last[i]=p; //上一个第i级结点
    }
    return(p.next[0]);
}
int  level()
{ //产生不超过MaxLevel的随机级别
  int lev=0;
  while(r.fRandom()<=prob)
          lev++;
return (lev<=maxlevel)?lev:maxLevel;
}
```



```
Public Object  put(Object  k, Object e)
{//在跳跃表中插入指定元素e
  if( tailKey.compareTo(k)<0 ) //元素键值超界
      throw new IllegalArgumentException("key is too large");
      SkipNode p=search(k);  //检查元素是否已存在
      if(p.key.equals(k)){        //元素已存在
        Object  ee=p.element;
        p.element=e;
        return ee;
       }
      int lev=level( );              //元素不存在，确定新结点级别
      if (lev>levels){
         lev=++levels;
         last[lev]=head;
      }
  SkipNode y=new SkipNode(k,e,lev+1);
  for (int i=0; i<=lev;i++){
     y.next[i]=last[i].next[i];
     last[i].next[i]=y;      }
   size++;
   return null;
}
```



```
Public Object  remove(Object  k)
{
   if( tailKey.compareTo(k) <0 ) //元素键值超界
     return null;
//搜索待删除元素
   SkipNode p=search(k);
        if(!p.key.equals(k)){        //未找到
           return null;
  //从跳跃表中删除结点
  for (int i=0; i<=levels&&last[i].next[i]==p;i++)
       last[i].next[i]=p.next[i];
    while(levels>0&&head.next[levels]==tail)levels--;
    size--;
    return p.element;
```



算法复杂度分析

空间：

1级链：n*p个元素

2级链：$n*p^2$个元素

i 级链：$n*p^i$ 个元素

Σ ：n/(1-p)

近似：O(n)

时间：

最坏情况： O(n+maxLevel) 跳跃表退化为有序链表
只有1个1级结点

随机技术：O(log n)

Las Vegas

拉斯维加斯
美国内华达州的最大城市，以赌博业为中心的庞大的旅游、购物、度假产业而著名，是世界知名的度假圣地之一。

Las Vegas algorithms were introduced by László Babai in 1979, in the context of the graph isomorphism problem, as a dual to Monte Carlo algorithms. Babai introduced the term "Las Vegas algorithm" alongside an example involving coin flips: the algorithm depends on a series of independent coin flips, and there is a small chance of failure (no result).

拉斯维加斯(Las Vegas)算法

拉斯维加斯算法可以显著地改进算法的有效性，它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。其典型调用形式：

```
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:

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

解此方程可得:

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

n后问题

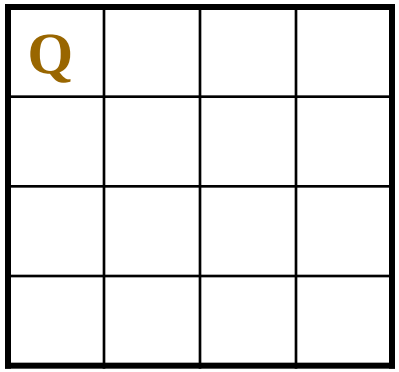
对于n后问题的任何一个解而言，每一个皇后在棋盘上的位置无任何规律，不具有系统性，而更象是随机放置的。由此容易想到下面的拉斯维加斯算法。

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。

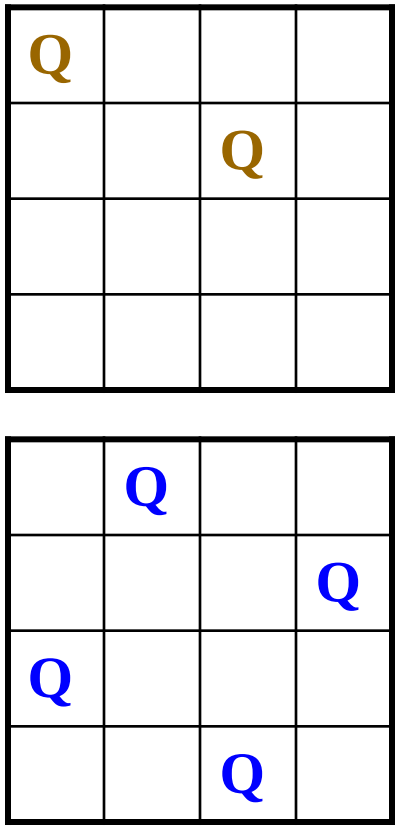
算法描述:

```
public class LVQueen
{
    static Random rnd;  //随机数产生器
    static  int n;              //皇后个数
    static  int[] x;            //解向量
}
private static boolean place(int k)
{//测试皇后k置于第x[k]列的合法性
   for ( int j=1; j<k; j++ )
     if(Math.abs(k-j)==Math.abs(x[j]-x[k]))||(x[j]==x[k]))
         return false;
     return true;
}
```

```
private static boolean queensLV( )
{//随机放置n个皇后的Las Vegas算法
   rnd=new Random();
   int k=1; //将放置的皇后编号
   int count=1;
   while( (k<=n)&&(count>0) ) {
      count=0;
      int j=0;
      for ( int i=1;i<=n;i++){
         x[k]=i;
         if(place(k))
            if(rnd.random(++count)==0)j=i;  //随机位置
       }
      if(count>0) x[k++]=j;
    }
    return(count>0); //count>0表示放置成功
}
```



```
public static void nQueen( )
{//解n后问题的Las Vegas算法
  //初始化
    x=new int [n+1];
    for ( int i=0;i<=n;i++)x[i]=0;
//反复调用随机放置n个皇后的Las Vegas算法
//直至放置成功
    while( !queensLV() );
}
```

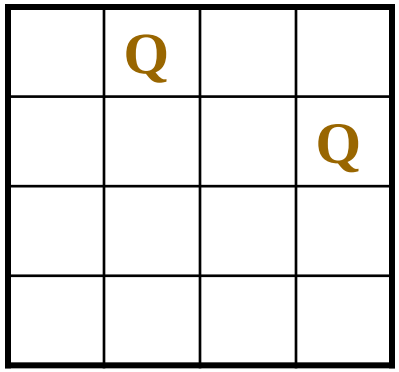


如果将上述随机放置策略与回溯法相结合，可能会获得更好的效果。可以先在棋盘的若干行中随机地放置皇后，然后在后继行中用回溯法继续放置，直至找到一个解或宣告失败。随机放置的皇后越多，后继回溯搜索所需的时间就越少，但失败的概率也就越大。

```
public class LVQueen1
{
  static Random rnd;
  static int n;
  static int []x;
  static int []y;
}
```

```
static void  backtrack(int t)
{
  if (t>n) {
     for (int i=1;i<=n;i++)
         y[i]=x[i];
     return;
   }
   else
      for (int i=1;i<=n;i++) {
        x[t]=i;
        if (place(t))backtrack(t+1);
      }
 }
```

```
private static boolean queensLV(int stopVegas )
{//随机放置m个皇后的Las Vegas算法
   rnd=new Random();
   int k=1;
   int count=1;
   while((k<= stopVegas)&&(count>0)) {
      count=0;
      int j=0;
      for ( int i=1;i<=n;i++){
         x[k]=i;
         if(place(k))
            if(rnd.random(++count)==0)j=i;  //随机位置
       }
      if(count>0) x[k++]=j;
    }
    return(count>0); //count>0表示放置成功
}
```



```
public static void nQueen(int stop)
{//与回溯法相结合的解n后问题的Las Vegas算法
   x = new int [n+1];
   y = new int [n+1];
   for ( int i=1;i<=n;i++){
        x[i]=0;
        y[i]=0;
   }
   while(!queensLV(stop));
   backtrack(stop+1);  //算法的回溯搜索部分
}
```

stopVegas	成功概率p	1次成功搜索访问的结点数平均值s	1次不成功搜索访问的结点平均值e	最终找到1个解访问的结点数t
8 后　0	1.0000	114.00	--	114.00
1	1.0000	39.63	--	39.63
2	0.8750	22.53	39.67	28.20
3	0.4931	13.48	15.10	29.01
4	0.2618	10.31	8.79	35.10
5	0.1624	9.33	7.29	46.92
6	0.1375	9.05	6.98	53.50
7	0.1293	9.00	6.97	55.93
8	0.1293	9.00	6.97	55.93

12后

stopVegas	p	s	e	t=s+(1-p)e/p
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:

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

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。

整数因子分解

```
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;
   }
```

计算时间： $\Omega(\sqrt{n})$

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

Pollard算法

在开始时选取0～n-1范围内的随机数x_1，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 $i=2^k$，$k=0,1,\ldots,$ 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。

```
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

识别重复元素

问题：一个有n 个数字的数组A[ ]，其中有n/2个不同的元素，其余元素是另一个元素的拷贝，即数组中有(n/2)+1个不同的元素。

要求：识别重复的元素。

解决该问题的确定性算法至少需要(n/2)+2个时间步。

```
private static int RepeatedElement(int []a, int n)
  {// Find the repeated element from a[1:n]
     int i, j;
     while (true) {
       i= random()%n+1;
       j= random()%n+1;
       if ((i!=j) && (a[i]==a[j])) return (i) ; }
}
```

识别重复元素

Õ（）定义：如果存在一个常数c, 使得对任何输入规模 n 的一个Las Vegas算法所使用的资源不超过cag(n)的概率为≥1-1/n , 则称该Las Vegas算法具有Õ(g(n))的资源（时间、空间等）界。称这些界为高概率界。

相似的定义可用于Θ’( )、Ω’( )、 Õ( )等函数。

Las Vegas算法只需花费Õ(logn)的时间。

证明：

如果 i 为重复元素所对应的n/2个数组下标中的任何一个，而 j 为除 i 之外这 n/2 个下标中的任何一个。那么，算法中 while 循环在任何一次迭代中退出的概率为：

$$P = \frac{n/2*(n/2-1)}{n^2}$$

对于所有的n ≥ 10，其值≥ 1/5，这表明算法在一个给定迭代中不退出的概率<4/5。

10次迭代，不退出的概率 $< (4/5)^{10} < 0.1074$

100次迭代，不退出的概率 $< (4/5)^{100} < 2.04*10^{-10}$

如果 $n=2*10^6$

确定性算法需要100万个时间步。

Las Vegas算法只需迭代100次。

算法在前$c\alpha\log n$(c为固定常数）次迭代内：

不退出的概率$<(4/5)^{c\alpha\log n} = n^{-c\alpha\log(5/4)}$

若取$c \geq 1/\log(5/4)$

不退出的概率$< n^{-\alpha}$

因此：算法在$\alpha\log n * 1/\log(5/4)$ 次迭代以内终止的概率$\geq 1 - n^{-\alpha}$

Monte Carlo

蒙特卡洛是摩纳哥公国的一座城市，位于欧洲地中海之滨、法国的东南方，属于一个版图很小的国家摩纳哥公国，世人称之为“赌博之国”、“袖珍之国”、“邮票小国”。

20世纪40年代，在科学家冯·诺伊曼、斯塔尼斯拉夫·乌拉姆和尼古拉斯·梅特罗波利斯于洛斯阿拉莫斯国家实验室为核武器计划工作时，发明了蒙特卡罗方法。因为乌拉姆的叔叔经常在摩纳哥的蒙特卡洛赌场输钱得名，而蒙特卡罗方法正是以概率为基础的方法。

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。
- 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。

蒙特卡罗(Monte Carlo)算法

- 有10个苹果，其中有6个是好的，4个是坏的。现在要你有放回的取出好的苹果，如果给你一次机会，你取出好的苹果的概率显而易见为3/5；现在给你两次机会，要求为有放回（相互独立）且取到好的苹果就停下，那么分为以下三种情况：

- 1.第一次就取到好的苹果 概率为3/5；
- 2.第一次取到坏的苹果，第二次取到好的苹果 概率为2/5*3/5=6/25；
- 3.第一次取到坏的苹果，第二次也取到坏的苹果 概率为2/5*2/5=4/25；

- 以上便是完整的三种情况，且概率之和为1，其中能取出好苹果的概率为3/5+6/25=21/25=84%

蒙特卡罗(Monte Carlo)算法

- 这个例子中得到好苹果的概率为3/5，即为p。我们取苹果的动作可以看作就是蒙特卡罗算法，当我们调用一次蒙特卡罗算法时，返回正确结果（即取得好苹果）的概率为3/5，算法优势为3/5-1/2=0.1。
- 当我们有两次机会抓取苹果时，我们可以看作调用了两次MC算法，此时返回正确结果的概率为21/25，算法优势为21/25-1/2=0.34
- 由此可见，蒙特卡罗算法其实就是一个通过增加调用MC的次数来不断提高获取正确解概率的方法。且无论一开始的优势有多小，我们都可以通过反复调用来扩大其算法优势，最终得到的算法具有可接受的错误概率。

蒙特卡罗(Monte Carlo)算法

- 在一般情况下，如果设蒙特卡罗算法是一致的p正确的。那么至少调用多少次蒙特卡罗算法，可以使得蒙特卡罗算法得到正确解的概率不低于1-ε(0<ε≤1-p)?
- 假设至少调用x次，则$p+(1-p)p+(1-p)^2p+\ldots+(1-p)^{x-1}p\geq 1-\varepsilon$，

即$1-(1-p)^x\geq 1-\varepsilon$（等比数列求和公式： $S = a_1\frac{1-q^n}{1-q}$ ）q为比例）

- $x \leq \frac{log\varepsilon}{\log(1-p)}$
- 抓取苹果问题：p=0.6，抓几次，可以达到0.9?
- $x \leq \frac{\log(1-0.9)}{\log(1-0.6)} \approx 2.5$

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x
    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

情况1：算法返回结果为true， 则数组T含有主元素。

情况2：算法返回结果为false，则数组T未必没有主元素。
由于数组T的非主元素个数小于n/2，
情况2发生的概率<1/2。

判定数组T的主元素存在性算法是一个偏真的1/2正确算法。
即如果数组T含有主元素，则算法以>1/2的概率返回true，
　如果数组T没有主元素，则算法肯定返回false 。

采用重复调用技术:

```
public static boolean  Majority2(int []t, int n)
{// 重复2次调用算法Majority
   if (Majority(t,n)) return true;
   else return Majority(t,n);
}
```

分析:

如果数组T没有主元素，

算法Majority(t,n)返回false，

Majority2(t, n)返回false。

如果数组T含有主元素，

算法Majority(t,n)返回true的概率p>1/2，

Majority2(t, n)也返回true。

分析:

如果第一次调用Majority(t,n)返回false的概率为1-p

第二次调用Majority(t,n)以概率p返回true 。

当数组T含有主元素，

算法Majority2(t, n)返回true 的概率为:

$p+(1-p)p=1-(1-p)^2 > 3/4$

即算法Majority2是一个偏真的3/4正确的蒙特卡罗算法

k 次重复调用Majority(t,n)均返回false的概率$<1/2^k$

k 次调用中，只要有一次返回true ，即可判定数组T含有主元素。

采用重复调用技术:

```
public static boolean majorityMC(int[]t,int n,double e)
  {// 重复log(1/e)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;
  }
```

对于任何给定的ε>0，算法**majorityMC**重复调用floor(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))，其中majority为O(n)。

素数测试

素数：任何大于1的整数，如果其只能被1和自身整除，那么称其为素数。通常认为1不是素数。

给定一个整数n，确定n是否为素数的问题即素数测试问题。

素数测试的简单算法： 检查每个$l \in [2, [n^{1/2}]]$是否能整除n。如果这些数中没有数能够整除n，那么n为素数，否则n为合数。算法的运行时间为$O(n^{1/2})$。

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 $(n-1)! \equiv -1 \pmod{n}$。

素数测试

```
public static boolean prime(int n)
    {
       rnd = new Random();
       int m=(int) Math.floor(Math.sqrt((double) n));
       int a=rnd.random(m-2)+2;
       return (n%a!=0);
}
```

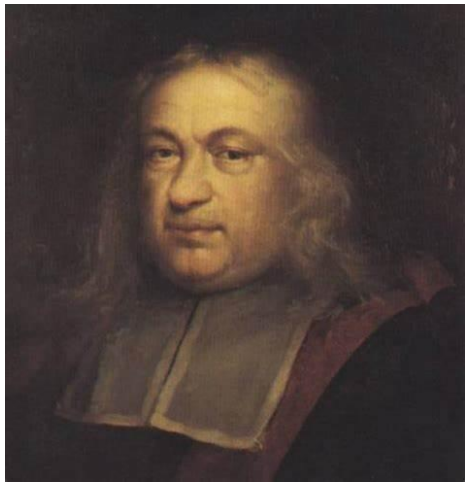
n=2623=43*61, d∈[2,51],

仅当d=43, 算法返回false, 其他情况返回true.

d=43的概率仅为2%, 算法返回错误结果概率为98%。

素数测试

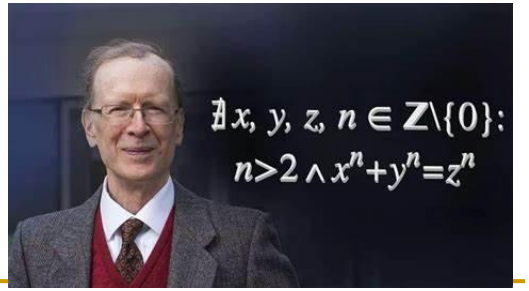
- **费马大定理：当整数n>2时，关于$x^n+y^n=z^n$的方程没有正整数解。**



- 1637年，费马在书本空白处提出费马猜想。
- 1770年，欧拉证明n=3时定理成立。
- 1823年，勒让德证明n=5时定理成立。
- 1832年，狄利克雷试图证明n=7失败，但证明 n=14时定理成立。
- 1839年，拉梅证明n=7时定理成立。
- 1850年，库默尔证明2<n<100时除37、59、67三数外定理成立。
- 1955年，范迪维尔以电脑计算证明了 2<n<4002时定理成立。
- 1976年，瓦格斯塔夫以电脑计算证明 2<n<125000时定理成立。
- 1985年，罗瑟以电脑计算证明2<n<41000000时定理成立。
- 1987年，格朗维尔以电脑计算证明了 2<n<10^1800000时定理成立。
- **1995年，怀尔斯证明 n>2时定理成立。**
- **1953年4月11日出生于英国剑桥，数学家，爵士，**

菲尔兹奖银质奖章获得者，美国国家科学院外籍院士，

牛津大学教授



素数测试

- **费马小定理**：如果p是一个素数，且0<a<p，则$(a^{p-1})\equiv 1 (\text{mod } p)$。
- 例如：67是一个素数，$2^{66} \text{ mod } 67 = 1$
- 费尔马小定理只是素数判定的一个必要条件：费马小定理的逆叙述不成立，即假如$a^{p-1}-1$是p的倍数，p不一定是素数（费马伪素数）。
- 341是一个合数341=11*31，但是$2^{340} \text{ mod } 341 = 1$

- 利用费尔马小定理，对于给定的整数n，可以设计素数判定算法。通过计算$d=2^{n-1} \text{mod } n$来判定整数n的素性。当d≠1时，n肯定不是素数；当d=1时，n很可能是素数。但也存在合数n使得$2^{n-1} \equiv 1 (\text{mod } n)$。例如当n=341时。
- 为了提高测试准确性，可以随机选取整数1<a<n-1，（而不是每次都计算a=2），然后用条件$a^{n-1} \equiv 1 (\text{mod } n)$判定整数n的素性。

费马小定理的证明

欧拉（Euler）：一些与素数有关的定理的证明（1736）

莱布尼兹（Leibniz）：未发表的手稿中发现他在1683年以前已经得到几乎是相同的证明。

素数测试

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或x=p-1。

因为$x^2 \equiv 1 \pmod p$ 等价于$x^2-1 \equiv 0 \pmod p$

则$(x-1)(x+1) \equiv 0 \pmod p$

故p必须整除x-1或者x+1。由于p是素数且0<x<p，可得x=1或x=p-1。

因此利用费马小定理计算$(a^{p-1}) \bmod p$的时候，可以加入对p的二次探测，判断$(a^{\frac{p-1}{2}})^2 \equiv 1$，就可以使用二次检测定理来判断。

如果$a^{\frac{p-1}{2}}$在mod p时的解不是1或者p-1，那么p就不是素数。

如果$a^{\frac{p-1}{2}} \equiv 1 \pmod p$，可进一步检验判断$a^{\frac{p-1}{4}}$，直到指数项为奇数。

素数测试

```
private static int power(int a, int p, int n, int &result, bool &composite)
 {// 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;
 }
```

在算法power的基础上，设计素数测试的蒙特卡罗算法：

```
public static boolean prime(int n)
    {
      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 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。

小结

概率算法：降低算法的复杂性，随机性选择常比最优选择省时。

4种常用的概率算法：

- 数值概率算法常用于数值问题的求解。
- 舍伍德算法总能求得问题的正确解。消除最坏情形与特定实例之间的关联。
- 蒙特卡罗算法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。（缺点）
- 拉斯维加斯算法找到的解一定是正确解，但有时找不到解，找到正确解的概率依赖于算法所用的时间。