# 第六章 随机化算法

- 教学目标
  - 理解产生伪随机数的线性同余法
  - 掌握数值随机化算法的特点及应用
  - 掌握蒙特卡罗算法的特点及应用
  - 掌握拉斯维加斯算法的特点及应用
  - 掌握舍伍德算法的特点及应用

# 6.1概述

## 6.1.1随机化算法的类型及特点

- 类型
  - 数值随机化算法
  - 蒙特卡罗算法
  - 拉斯维加斯算法
  - 舍伍德算法
- 特点
  - 数值随机化算法常用于数值问题的求解，所得到的解往往都是近似解，而且近似解的精度随计算时间的增加不断提高。

- 蒙特卡罗算法能求得问题的一个解，但这个解未必是正确的。求得正确解的概率依赖于算法执行时所用的时间，**所用的时间越多得到正确解的概率就越高。**一般情况下，蒙特卡罗算法不能有效地确定求得的解是否正确。

- 拉斯维加斯算法不会得到错误解，一旦找到一个解，那么这个解肯定是正确的。但是有时候拉斯维加斯算法可能找不到解。拉斯维加斯算法**得到正确解的概率随着算法执行时间的增加而提高。**

- 舍伍德算法不会改变对应确定性算法的求解结果，每次运行都能够得到问题的解，并且所得到的解是正确的，**但是能提高时间复杂度的实际感受。**

# 6.1.2 随机数发生器

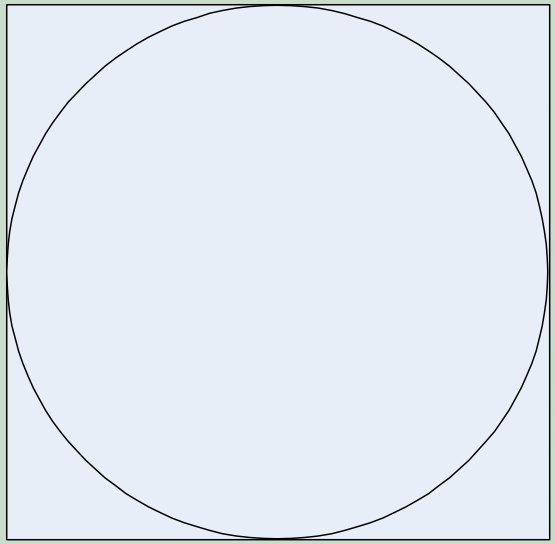
- **随机数发生器**
  - 产生随机数的方法
- **伪随机数发生器**
  - 通过一个固定的、可以重复的计算方法产生随机数的发生器
- **线性同余法**

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

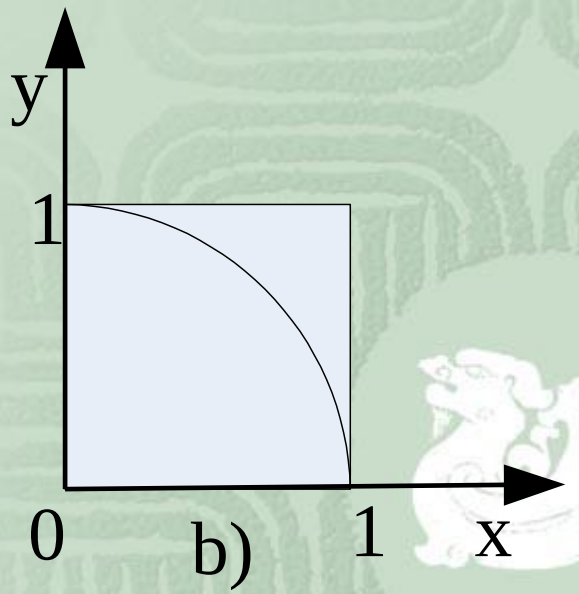
- d为种子；b为系数，满足b≥0；c为增量，满足c≥0；m为模数，满足m>0。b、c和m越大且b与m互质可使随机函数的随机性能更好

# 6.2数值随机化算法

- 6.2.1计算π的值
  - 将n个点随机投向一个正方形，设落入此正方形内切圆（半径为r）中的点的数目为k。



a)



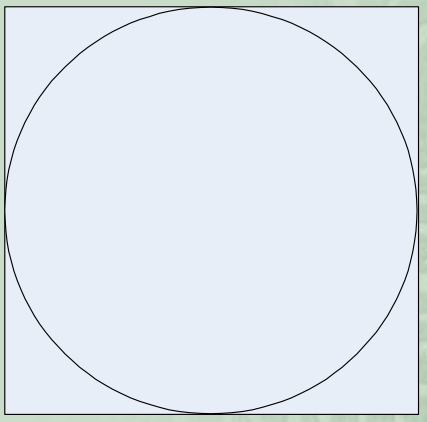

b)

- 假设所投入的点落入正方形的任一点的概率相等，则所投入的点落入圆内的概率为

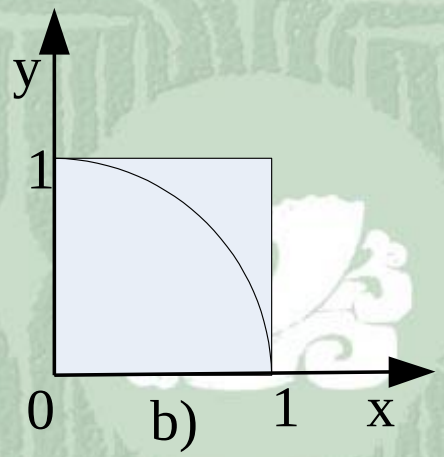
$$\frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

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

- n是点的总数，k是落入园中的点数。当n足够大时，k与n之比就逼近这一概率，在具体实现时只以第一象限为样本且r取值为1。



a)




b)

```
double Darts(int n)
{   //定义一个RandomNumber类的对象darts
    static RandomNumber darts;
    int k=0, i;
    double x, y;
    for( i=1; i<=n; i++)
    {   x=darts.fRandom();  //产生一个[0,1)之间的实数
        y=darts.fRandom();  //产生一个[0,1)之间的实数
        if( (x*x+y*y) <= 1 )
            k++;
    }
    return 4*k/double(n);
}
```

# 6.2.2计算定积分

- 设f(x)是[0,1]上的连续函数且0≤f(x)≤1，计算积分值

$$I = \int_0^1 f(x)dx$$

积分 $I$ 等于图中的阴影区域G的面积

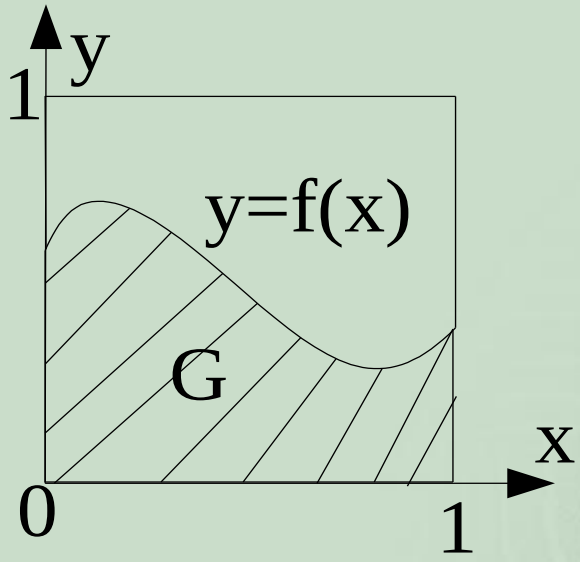



图6-2 随机投点实验估算I值示意图

- 向单位正方形内随机投入n个点，如果有m个点落入G内，则 $I$ 近似等于随机点落入G内的概率，即 $I \approx m/n$

```
double Darts(int n)
{   static RandomNumber dart;
    int k=0,i;double x,y;
    for( i=1 ;i<=n;i++)
    {   x=dart.fRandom ( );   //产生一个[0,1)之间的实数
        y=dart.fRandom ( );   //产生一个[0,1)之间的实数
        if( y<=f(x) )
            k++;
    }
    return k/double(n);
}
```

# 思考题

- 蒲丰投针
  - 蒲丰(C．Buffon)投针实验是运用实验法研究几何概率的典型范例． 1777年的一天，蒲丰邀请许多宾朋来家做客，并参观他的实验．他事先在白纸上画好了一条条等距离的平行线，然后将纸铺在桌上，又拿出一些质量均匀、长度为平行线间距离之半的小针，请客人把针一根根随便扔到纸上，蒲丰则在一旁计数．结果，共投了2122次，其中与任一平行线相交的有704次。
  - 蒲丰又做了一个简单的除法2212÷704≈3.142，最后他宣布，这就是圆周率π的近似值，还说投的次数越多越精确。

# 6.2数值随机化算法

- 基于概率的数值随机化算法适用于非自然科学领域，如社会学、心理学、经济学等。这类学科没有明确的数学模型，用概率模型则可以得到较理想的解。

# 6.3蒙特卡罗算法

- 设p是一个实数，且0.5<p<1。如果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，则称该算法是p正确的。
- 对于同一实例，蒙特卡罗算法不会给出两个不同的正确解，则称该算法是一致的。
- 而对于一个一致的p正确的蒙特卡罗算法，要想提高获得正确解的概率，只需执行该算法若干次，从中选择出现频率最高的解即可。

# 6.3蒙特卡罗算法

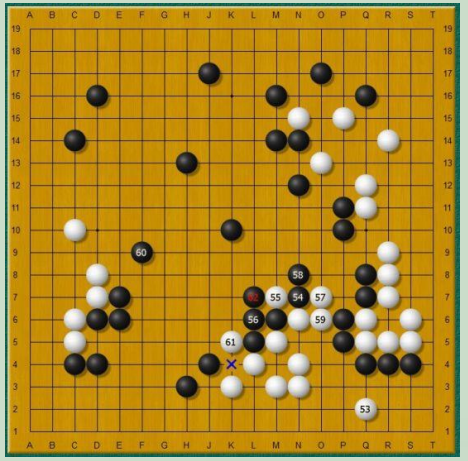
- 设蒙特卡罗算法是一致的p正确的。那么至少调用多少次蒙特卡罗算法，可以使得蒙特卡罗算法得到正确解的概率不低于 $1-\varepsilon\ (0<\varepsilon\leq 1-p)$ ？
- 假设至少调用x次，则

$$p+(1-p)p+(1-p)^2p\ldots+(1-p)^{x-1}p\geq 1-\varepsilon$$

$$1-(1-p)^x\geq 1-\varepsilon\quad (1-p)^x\leq\varepsilon\quad x\geq\log_{1-p}\varepsilon$$

**说明：当调用的次数大于某个值后，将以不低于1- ε的概率出现正确解。**

# 6.3蒙特卡罗算法



- 阿尔法狗的系统组成
  - 1. 走棋网络（Policy Network），给定当前局面，预测/采样下一步的走棋。
  - 2. 快速走子（Fast rollout），目标和1一样，但在适当牺牲走棋质量的条件下，速度要比1快1000倍。
  - 3. 估值网络（Value Network），给定当前局面，估计是白胜还是黑胜。
  - 4. 蒙特卡罗树搜索（Monte Carlo Tree Search，MCTS），把以上这三个部分连起来，形成一个完整的系统。

# 6.3.1主元素问题

- 问题描述
  - 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。例如，在数组T[7]={3, 2, 3, 2, 3, 3, 5}中，元素3就是主元素。

- 蒙特卡洛算法
  - 随机地选择数组中的一个元素T[i]进行统计，如果该元素出现的次数大于$n/2$，则该元素就是数组的主元素，算法返回true；否则随机选择的这个元素T[i]不是主元素，算法返回false。此时，数组中可能有主元素也可能没有主元素。如果数组中存在主元素，则非主元素的个数小于$n/2$。因此，算法将以大于1/2的概率返回true，以小于1/2的概率返回false，这说明算法出现错误的概率小于1/2。如果连续运行算法$k$次，算法返回false的概率将减少为$2^{-k}$，则算法发生错误的概率为$2^{-k}$。

# 6.3.1主元素问题

- 算法描述

```
bool majority(Type T[ ], int n)     // 判定主元素的蒙特卡罗算法
{    RandomNumber rnd;
     int i=rnd.random(n)+1;        //产生1~n之间的随机下标
     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);      //当 k>n/2 时，T含有主元素
}
```

```
bool majorityMC(Type T[ ], int n, double)
{    // 重复次调用多次算法majority
     int k= (int)ceil(log(ε)/log(1-p));
     for (int i=1; i<=k; i++)
          if (majority(T,n))
               return true;
     return false;
}
```

# 6.3.1主元素问题

- 如果数组里不存在主元素……
- 有没有更好的方法寻找主元素？
  - 先找中数，判断中数是否是主元素
  - 快排找中数

# 6.3.2素数测试

- 试除法

  用2,3,…, $\sqrt{n}$ 去除n，判断是否能被整除，如果能，则为合数，否则为素数。时间复杂度$O(n^{0.5})$

- 算法描述

```
bool Prime(unsigned int n)
{    int m=floor(sqrt(double(n)));
     for(int i=2;i<=m;i++)
          if(n%i==0)
               return false;
     return true;
 }
```

- **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$(n-1)! \equiv -1 \pmod{n}$。
  - 时间复杂度如何？O(n)

- **费尔马小定理**
  - 如果p是一个素数且a是整数，则 $a^p \equiv a \pmod p$。特别地，若a不能被p整除，则$a^{p-1} \equiv 1 \pmod p$。
  - 只是必要条件，但是出现例外的概率很小，所以可以使用蒙特卡洛算法。1~1000中不是的只有22个
  - 多次调用蒙特卡洛算法，提高准确性
  - 提高计算$a^p$的效率
    - 1反复平方
    - 2迭乘中只需保存余数即可
    - 改进后的时间复杂度$O(\log_2 n)$

- **二次探测定理**
  - 如果p是一个素数，x是整数且0<x<p，
  - 则方程$x^2 \equiv 1 (\mathrm{mod}\ p)$的解为：x=1、p-1。

  - 在反复平方迭乘中可以使用二次探测定理，从而进一步提高准确性。

# 6.4拉斯维加斯算法

- 拉斯维加斯型概率算法不时地做出可能导致算法陷入僵局的选择，并且算法能够检测是否陷入僵局，如果是，算法就承认失败。这种行为对于一个确定性算法是不能接受的，因为这意味着它不能解决相应的问题实例。但是，拉斯维加斯型概率算法的随机特性可以接受失败，只要这种行为出现的概率不占多数。当出现失败时，只要在相同的输入实例上再次运行概率算法，就又有成功的可能。

# 6.4拉斯维加斯算法

- 该算法的一个显著特征是，算法运行一次，或者得到一个正确的解，或者无解。因此，需要对同一输入实例反复多次运行算法，直到成功地获得问题的解。

- 因此，通常拉斯维加斯型概率算法的返回类型为bool，并且有两个参数：一个是算法的输入，另一个是当算法运行成功时保存问题的解。当算法运行失败时，可对同一输入实例再次运行，直到成功地获得问题的解。

**算法的一般形式**

```
void  Obstinate(input x, solution &y)
{    success=false;
     while (!success) success=LV(x, y);
}
```

有时候得到正确解的概率太小导致循环次数过多，则可以通过设置一个次数的上限。

# 6.4拉斯维加斯算法

- 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。
- 在更强意义下，要求存在一个常数δ>0，使得对问题的每一个实例x均有p(x)≥δ。
- 思考：给定一个拉斯维加斯算法，设p(x)是对输入x调用它获得问题的一个解的概率，s(x)和e(x)分别是算法对于具体实例x求解成功或失败所需的平均时间，算法找到具体实例x的一个解所需的平均时间是多少？

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

# 6.4.1整数因子分解

- 整数因子分解
  - 将大于1的整数n分解为 $n = p_1^{m_1} p_2^{m_2} \cdots p_k^{m_k}$ 的形式
  - $p_1,p_2,\ldots,p_k$是k个素数且$p_1<p_2<\ldots<p_k$，$m_1,m_2,\ldots,m_k$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。
- 结合上节的素数测试问题，整数因子分解问题实质上可以转化为整数的因子分割问题

- 试除法进行因子分割

```
int Split(int n)
{     int k=floor(sqrt(double(n)));
      for (int i=2; i<=k; i++)
            if (n%i==0)
                  return i;
      return 1;
}
```

- Pollard(n)算法进行因子分割 **Birthday Trick**

(1) 产生0~n-1范围内的一个随机数x，令y=x;

(2) 按照 $x_i = (x_{i-1}^2 - 1) \bmod n$ i=2,3,4,…

产生一系列的$x_i$

产生一组随机数

(3) 对于k=$2^j$（j=0,1,2,…），以及 $2^j < i \le 2^{j+1}$，

计算$x_i$-$x_k$与n的最大公因子 d=gcd($x_i$-$x_k$,n)，

如果$1 < d < n$，则实现对n的一次分割，输出d。

本质上就是找两个n以内的随机数

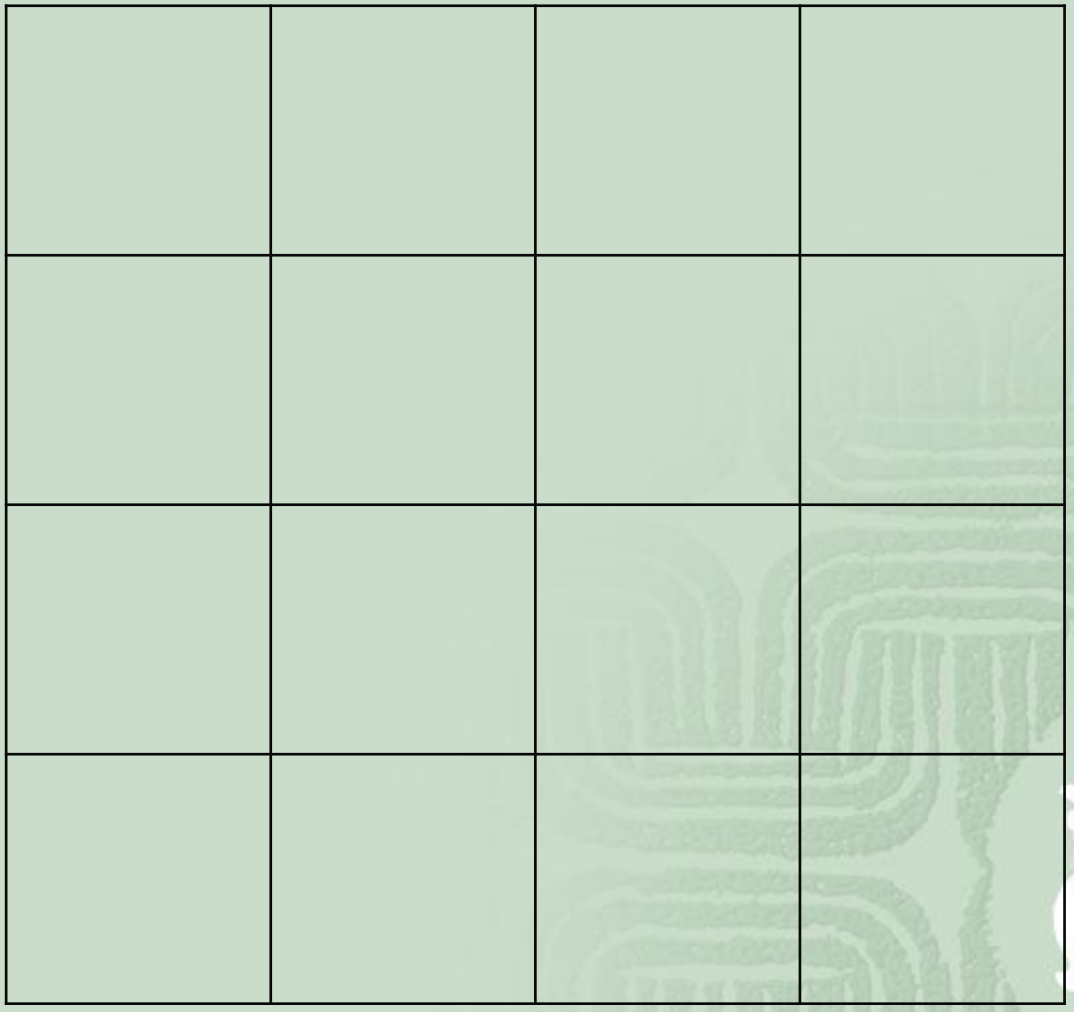
- Pollard(n)算法进行因子分割

```
void Pollard(int n)
{    RandomNumber rnd;
     int i=1;
     int x=rnd.Random(n);
     int y=x;
     int k=2;
     while (true)
     {     i++;
           x=(x*x-1)%n;
           int d=gcd(y-x,n);
           if ((d>1) && (d<n))    cout<<d<<endl;
           if (i==k)  { y=x;  k*=2; }
     }
}
```

# 6.4.2 n皇后问题

- 问题描述
  - n皇后问题要求将n个皇后放在n×n棋盘的不同行、不同列、不同斜线的位置，找出相应的放置方案
- 随机化措施
  - 对某行所有列位置进行随机
  - 对某行放置皇后的有效位置进行随机

# 6.4.2 n皇后问题



# 关键代码分析

```
bool Queen::QueensLV( )
{   RandomNumber rnd;
    int k=1;    //下一个放置的皇后编号
    int count=1;   //第k个皇后在第k行的有效位置数
    while((k<=n)&&(count>0))
    {   count=0;
        for(int i=1;i<=n;i++)
        {   x[k]=i;
            if(Place(k))
                //第k个皇后在第k行的有效位置存于y数组
                y[count++]=i;
        }
        //从有效位置中随机选取一个位置放置第k个皇后
        if(count>0)
            x[k++]=y[rnd.Random(count)];
     }
     return (count>0);     //count>0表示放置成功
}
```

先判断，再随机放置

```
bool Queen::QueensLV1(void)
{     RandomNumber rnd;       //随机数产生器
      int k=1;                //下一个放置的皇后编号
      //尝试产生随机位置的最大次数，用户根据需要设置
      int count=maxcout;
      while(k<=n)
      {    int i=0;
           for(i=1;i<=count;i++)
           {       x[k]=rnd.Random(n)+1;
                   if(Place(k))
                        break;    //第k个皇后在第k行的有效位置存于y数组
           }
           if(i<=count)
                k++;
           else
                break;
       }
       return (k>n);     //k>n表示放置成功
}
```

先随机放置，再判断

# 6.5舍伍德算法

- 分析确定性算法在平均情况下的时间复杂性时，通常假定算法的输入实例满足某一特定的概率分布。
- 事实上，很多算法对于不同的输入实例，其运行时间差别很大。此时，可以采用舍伍德型概率算法来消除算法的时间复杂性与输入实例间的这种联系。

# 6.5舍伍德算法

- 如果一个确定性算法无法直接改造成舍伍德型概率算法，可借助于随机预处理技术，即不改变原有的确定性算法，仅对其输入实例随机排列(称为洗牌)。假设输入实例为整型，下面的随机洗牌算法可在线性时间实现对输入实例的随机排列。

算法——随机洗牌

```
void RandomShuffle(int n, int r[ ])
{    for (i=0; i<n; i++)
     {    j=Random(0, n-i-1);
          r[i]←→r[j];        //交换r[i]和r[j]
     }
}
```

# 6.5舍伍德算法

- 舍伍德型概率算法总能求得问题的一个解，并且所求得的解总是正确的。但与其相对应的确定性算法相比，舍伍德型概率算法的平均时间复杂性没有改进。换言之，舍伍德型概率算法不是避免算法的最坏情况行为，而是设法消除了算法的不同输入实例对算法时间性能的影响，对所有输入实例而言，舍伍德型概率算法的运行时间相对比较均匀，其时间复杂性与原有的确定性算法在平均情况下的时间复杂性相当。

# 6.5舍伍德算法

- 随机快速排序
  - 在3.5节确定性算法的基础上，引入随机性操作。
  - 随机操作——随机选择基准元素
- 关键代码分析

```
int RandPartition(int a[ ],int low,int high)   //随机划分
{    RandomNumber random;
     int i=random(high-low+1)+low;
     swap(a[low],a[i]);
     int j=Partition(a,low,high);
     return j;
}
```

```
void rqs(int a[ ],int left,int right)  //随机快速排序
{
    if(left<right)
    {
        int p=RandPartition(a,left,right);
        rqs(a,left,p-1);
        rqs(a,p+1,right);
    }
}
```

# 6.5.2线性时间选择

- 线性时间选择问题——寻找第K大(小)的数。
- 确定性算法采用类似快排的方式找第K大的数。
- 引入随机性成分
  - 随机选择一个元素作为基准元素
- 关键代码分析

```
Type select(Type a[], int left, int right, int k)
{    RandomNumber rnd;
     if(left>=right)
          return a[left];
     int i=left, j=rnd.Random(right-left+1)+left;
     swap(a[i], a[j]);
     j=Partition(a,left,right);
     int count=j-left+1;
     if(count<k)
          select(a[],j+1, right, k-count);
     else
          select(a[],left, j, k);
}
```