# 随机化算法

| 算法 | 一定能得到解 | 解一定正确 | 备注 |
|---|---|---|---|
| 拉斯维加斯算法 | 否 | 是 | 计算越久，得到正确解概率越大 |
| 舍伍德算法 | 是 | 是 | 最坏/平均情况时间复杂性差别巨大 |
| 数值概率算法 | 是(近似解) | 近似解 | 解的精度随计算时间的增加而不断提高 |
| 蒙特卡罗算法 | 是(准确解) | 否 | 以正的概率给出正解，时间越久求得正解概率越大 |

# 1. (伪)随机数的生成：线性同余法

$$\begin{cases} a_0 = d \\ a_n = (ba_{n-1} + c) \mod m & n = 1, 2, \ldots \end{cases}$$

1. 其中$b \geq 0, c \geq 0, d \leq m$，且$d$为该随机序列的种子
2. $m$应该充分大，且$m, b$之间互质

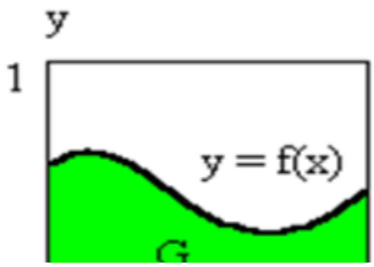
# 2. 数值随机化算法

## 2.1. 随机投点法计算$\pi$

内圆外方，投掷点落入圆的概率为$\frac{\pi}{4}$，设投掷$n$个点后$k$个落入圆中，则$\frac{\pi}{4} = \frac{k}{n}$

```
double calculatePi(int n)
{
 int k = 0;
 double x, y;
 for (int i = 0; i < n; ++i)
 {
     x = (double)rand() / RAND_MAX; // 生成0到1之间的随机数
     y = (double)rand() / RAND_MAX; // 生成0到1之间的随机数
     if (x * x + y * y <= 1) ++k;
 }
 return 4.0 * k / n;
}
```

## 2.2. 定积分计算



$0 \leq f(x) \leq 1, P\{y \leq f(x)\} = \int_0^1 \int_0^{f(x)} dydx = \int_0^1 f(x)dx$

向正方形内投入$n$个点，$k$个落入$G$，则面积为$\frac{k}{n}$

```
double calculatePi(int n)
{
 int k = 0;
 double x, y;
 for (int i = 0; i < n; ++i)
 {
     x = (double)rand() / RAND_MAX; // 生成0到1之间的随机数
     y = (double)rand() / RAND_MAX; // 生成0到1之间的随机数
     if (y<= f(x) ) k++;
 }
 return k / double(n);
}
```

## 2.3. 解非线性方程组：函数极小值法

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

1 目标函数：$\Phi(x) = \sum_{i=1}^{n} f_i^2(x)$

2 步骤

1. 更具预选分布(正态/均匀)，选随机点$x_j$计算目标函数，更新$x_{j+1} = x_i + \Delta x_j$再计算目标函数，以此类推
2. 直到满足$\varPhi(x) < \epsilon \approx 0$，算是求得解

# 3. 舍伍德算法

## 3.1. 算法概述

1 算法功能：消除算法所需时间与输入实例间的联系，就好比最坏情况输入时将输入随机打乱，避免了最坏

2 确定性算法：输入给定后，之后每一步都固定，可以完全预测执行过程和输出的算法

3 确定性算法改造成舍伍德算法

1. 直接改造：比如快速排序中，随机选择基准元素
2. 间接改造：采用随机预处理技术，不改变确定性算法，仅对输入洗牌(如下)

```cpp
void shuffle(int a[], int n) {
    srand(time(0)); // 使用当前时间作为随机数生成器的种子

    for (int i = 0; i < n; i++) {
        int j = i + rand() % (n - i); // 生成一个在[i, n-1]范围内的随机数
        std::swap(a[i], a[j]); // 交换元素
    }
}
```

## 3.2. 快排：随机选择基准点版本

1 分割

```cpp
int partition(int a[], int p, int r) {
 int x = a[p]; // x 是基准值，初始设置为子数组的第一个元素。
 int i = p;
 int j = r + 1;
 while (true) {
     while (a[++i] < x && i < r); // 右移，直到找到一个大于或等于基准值的元素。
     while (a[--j] > x); // 左移，直到找到一个小于或等于基准值的元素。
     if (i >= j) break; // 当两个指针相遇时，退出循环。
     swap(a, i, j); // 交换两个元素的位置。
 }
 a[p] = a[j]; // 把基准值放到正确的位置上。
 a[j] = x;
 return j; // 返回基准值的位置。
}
```

2 基准随机选择

```cpp
int randomizedPartition(int a[], int p, int r) {
 int i = rand() % (r - p + 1) + p; // 随机生成一个介于 p 和 r 之间的索引。
 swap(a, p, i); // 把随机选择的元素与子数组的第一个元素交换。
 return partition(a, p, r); // 调用 partition 函数进行分区操作。
}
```

3 开始快排

```cpp
void quickSort(int[] a, int p, int r) {
 if (p < r) {
     int q = randomizedPartition(a, p, r);
     quickSort(a, p, q - 1);
     quickSort(a, q + 1, r);
 }
}
```

## 3.3. 线性时间选择：随机版本

1 目的：求一个数组中第$k$小的元素

2 代码实现

```
void randomizedSelect(int p, int r, int k) //p,r为数组起始/终止索引
{
 if (p == r) return a[p];
 int i = randomizedPartition(p, r), //随机选一个基准, 划为左右两半, 返回基准索引
 j = i - p + 1; //计算左侧数组中的元素数量

 //第k小的数就是基准
 if (k == j) return a[i];
 //第k小的数必定在基准左边, 则递归查找左侧
 else if (k < j) return randomizedSelect(p, i, k);
 //第k小的数必定在基准右边, 则递归查找右侧
 else return randomizedSelect(i + 1, r, k - j);
}
```

## 3.4. 搜索有序表

1 有序集`s[]`的表示：模拟有序链表

1. `value[]`存放了`s[]`中所有的元素(但是不是有序的)，先假设`value[i]=s[k]`
2. `link[0]`指向`s[]`的第一个元素，然后`value[link[i]]=s[k+1]`
3. 示例：`s[]=[1,2,3,5,8,13,21]`

| 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 |

2 随机化算法改进搜索：假设要搜索的元素是$x$

1. 算法思想：随机抽取数组元素$k$次，从最接近元素$x$的位置开始顺序搜索
2. 顺序搜索的平均比较次数：$O(\frac{n}{k+1})$

3 函数实现

```
// n: 有序集合中元素的数量
// x: 需要搜索的元素
// index: 若找到x, index会被更新为x在value[]中的位置
bool Search(int value[], int link[], int n, int x, int& index)
{
    index = 0;
    int max = 0; //S集合中元素的下界
    int m = floor(sqrt(double(n)));//设定随机抽取次数

    //开始随机抽取
    for (int i = 1; i <= m; i++)
    {
        int j = rand() % n + 1; //随机产生一个位置j
        int y = value[j]; //获取随机位置j上的元素y
        if ((max < y) && (y < x)) //找到更接近x且大于max的y时, 更新max和index
        {
```

```
            max = y;
            index = j;
        }
    }

    // 从较接近 x 的位置开始顺序搜索
    while (value[link[index]] < x) {index = link[index];}
    // 检查是否找到元素 x
    return (value[link[index]] == x);
}
```

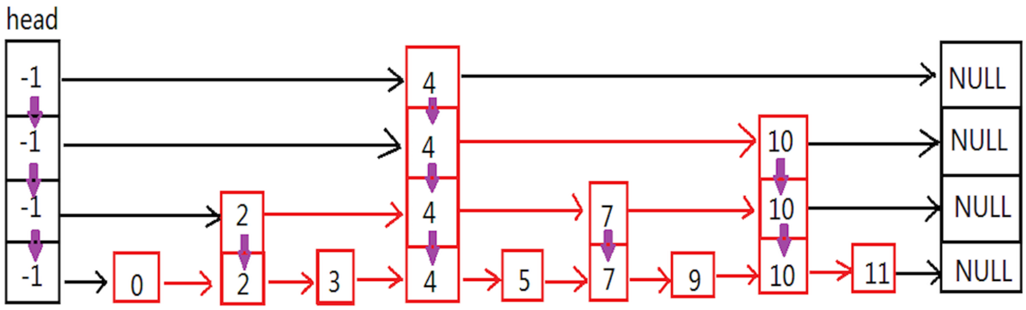
## 3.5. 跳跃表

1 场景：在一个有序链表中，搜索某一个值

2 附加指针：链表结点中，指向其他(而非下一个)结点的指针

3 跳跃表：在链表每个结点处增设一附加指针，使得搜索时直接可以跳过若干节点

4 完全跳跃表的结构




1. $k$级节点：1个向前的指针 + 有$k$个跳跃的指针，跳跃的距离为$2, 4, \dots, 2^k$
2. 结点的排布：第$i$个$k$级结点安排在跳跃表的$2ki$处

# 4. 拉斯维加斯算法

## 4.1. 算法概述

1 算法典型调用

```
void obstinate(InputType x, OutputType 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);
}
```

2 算法分析：$t(x) = p(x)s(x) + [1 - p(x)][e(x) + t(x)]$，解得

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

| 符号 | 含义 |
| --- | --- |
| $x$ | 算法实例 |
| $t(x)$ | 算法**obstinate**找到$x$一个解的平均时间 |

| 符号 | 含义 |
| --- | --- |
| $p(x)$ | 实例$x$获得解的概率 |
| $s(x)/e(x)$ | 实例$x$被成功/失败求解的平均时间 |

## 4.2. $n$后问题的拉斯维加斯法

1 思想：

1. 各行中，相继随机放置皇后，但注意新放的皇后不能冲突
2. 一直到$n$个皇后全放完了结束算法
3. 遇到无法再放置下一个皇后时，全盘重新开始

2 随机放置n个皇后的Las Vegas算法

```
//n为皇后个数
//x[]为解向量
//place(k)测试皇后k的放置是否会与之前的冲突
bool queensLV()
{
    int k = 1; //将要放置的皇后编号，从1→n
    int count = 1;//count记录了皇后放置的可能数，为了第一次能进入循环先初始化为1
    while ((k <= n) && (count > 0)) //当还有皇后，且皇后还可放置的时候
    {
        count = 0; //记录皇后放置的可能数
        int j = 0; //记录迭代选择的列位置
        for (int i = 1; i <= n; i++) //遍历每一列
        {
            x[k] = i; //皇后k先放到遍历所在的列
            if (place(k)) //如果放置合法
            {
                count++;
                if (rnd() % count == 0) j = i;
                //每次发现一个新的合法位置时，都以概率(1/count)选择他
            }
        }
        if (count > 0) {x[k] = j; k++}//若这一行可以合法放入则考虑下一行，否则退出
    }
    return (count > 0); //count>0表示放置成功
}
```

3 解n后问题的Las Vegas算法

```
public LVQueen(int n)
{
    x = new int[n + 1];
    for (int i = 0; i <= n; i++) x[i] = 0;//先全部初始化为0
    while (!queensLV());//不断调用拉斯维加斯方法，直到放置成功
    //输出x[].....
}
```

> > 4 改进：失败退出改为失败回溯

# 5. 蒙特卡罗算法

## 5.1. 算法特点

1 $p$正确的：即算法对于问题任一实例得到正确解的概率不小于$p$

2 一致性：一个实例只能得到一个答案

3 偏真性：有一定概率给出错误解

4 提高正确解概率的方法：多次执行，选取出现频率最高的解

## 5.2. 非判定问题

1 偏$y_0$的算法：

1. 符号含义：$y_0$是问题的特殊解，$MC(x)$是所给问题的蒙特卡罗算法
2. $MC(x)$是偏$y_0$的算法的条件：存在实例子集$X$使得
    - $x \notin X$时，$MC(x)$返回正确解
    - $x \in X$时，正解是$y_0$，但$MC(x)$不一定返回$y_0$

2 重复调用一个一致的+$p$正确+偏$y_0$蒙特卡罗算法$k$次：得到一个偏$y_0$的+$(1-(1-p)^k)$正确的蒙特卡罗算法