算法分析与设计

主讲教师：刘峤

第5章：随机化算法

（ Randomized Algorithm ）

知识要点

- 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

伪随机数生成算法

引言

- 随机化算法与前几类算法的区别
  - 分治法、动态规划、贪心算法、回溯法和分支限界法等
    - 算法的每一计算步骤都是确定的
  - 随机化算法允许执行过程中随机选择下一计算步骤
- 随机化算法的特点
  - 当算法执行过程中面临选择时随机选择通常比最优选择省时
    - 因此随机化算法可在很大程度上降低算法复杂性
  - 对所求解问题的同一实例用同一随机化算法求解两次
    - 可能得到完全不同的效果（所需时间和计算结果等）
  - 设计思想简单，易于实现

引言

- 本章介绍的随机化算法包括
  - 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，也非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结果的正确性

伪随机数

- 随机数在随机化算法设计中扮演着十分重要的角色
  - 随机数最重要的特性是：序列中前后的数值间毫无关系
- 当前的计算机无法产生真正的随机数
  - 真正的随机数是使用物理现象产生的
    - 如：掷钱币、骰子、转轮、使用电子元件的噪音、核裂变等
    - 样的随机数发生器叫做：物理性随机数发生器
    - 主要缺点是：技术要求比较高
    - 通常应用于一些关键性的应用中（如军事密码）
  - 在随机化算法中使用的随机数都是伪随机数
    - 伪随机数是通过一个固定的、可以重复的计算方法产生的
    - 计算机产生的随机数有很长的周期性
    - 且具有类似于随机数的统计特征

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：

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

  - 其中：b≥0，c ≥ 0，d ≤ m， **d** 称为该随机序列的种子
  - 随机序列的随机性能由常数b、c和m的选择决定
    - 这是随机性理论研究的主要内容之一
  - 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

随机化算法概述

- 随机化算法（randomized algorithm）
  - 是指需要利用随机数发生器的算法
  - 即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
- 概率算法（probabilistic algorithm）
  - 随机化算法有时也称概率算法（二者有细微的区别）
  - 随机算法：取得结果的途径是随机的
    - 如：拉斯维加斯算法
  - 概率算法：取得的解是否正确存在随机性
    - 如：蒙特卡罗算法
  - 本书中统一称为随机化算法

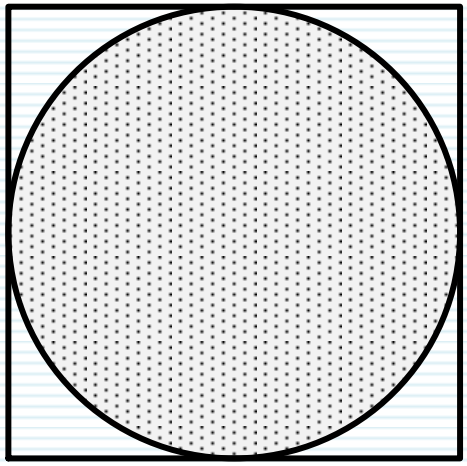
Classification of Probabilistic Algorithms

- **Karp-Type** : **Deterministic** algorithm whose behavior (performance) is probabilistically analyzed
  - **Richard M. Karp**,"The probabilistic analysis of some combinatorial search algorithms," in Algorithms and Complexity: New Directions and Recent Results, Academic Press, New York, 1976.
- **Rabin-Type** : **Nondeterministic** algorithm using randomization
  - **Michael O. Rabin**,"Probabilistic algorithms,"in Algorithms and Complexity: New Directions and Recent Results, 1976.
    - Numerical probabilistic algorithm
    - Monte Carlo algorithm
    - Las Vegas algorithm
    - Sherwood algorithm
- In order to distinguish these two types, probabilistic algorithms of the Rabin type are often called randomized algorithms.

数值随机化算法

（ Numerical Randomized Algorithm ）

用随机投点法计算π值



- 数值随机算法：用于求数值问题的近似解
- 示例1：用随机投点法计算n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
  - 因此有：$\pi \approx \frac{4k}{n}$

用随机投点法计算π值

```
double darts(int n){
    srand((unsigned)time(NULL));  // 初始化随机数发生器
    int k = 0;
    for (int i = 1; i <= n; i++) {
        double x = rand() / double(RAND_MAX);
        double y = rand() / double(RAND_MAX);
        if ((x*x + y*y)<=1) k++;
    }
    return 4*k/double(n);
}
```

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

用随机投点法计算π值：算法运行结果

刘徽（225～295）提出割圆术，推算π的近似值为3.14

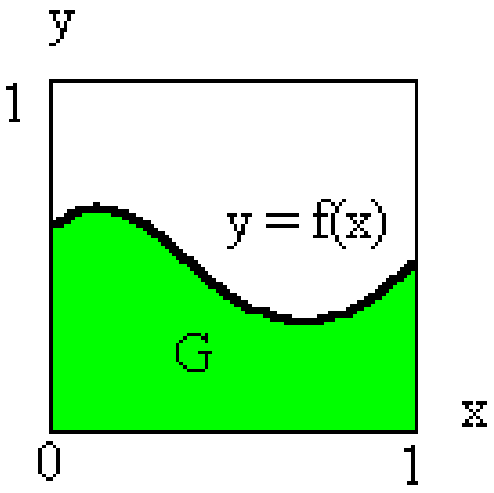
```
n = 5 x 10^2      pi = 3.112
n = 5 x 10^3      pi = 3.1152
n = 5 x 10^4      pi = 3.15
n = 5 x 10^5      pi = 3.14275
n = 5 x 10^6      pi = 3.14302
```

祖冲之（429-500）推算π的近似值在3.1415926和3.1415927间

```
= 5 x 10^1        pi = 3.36
= 5 x 10^2        pi = 3.096
= 5 x 10^3        pi = 3.1296
= 5 x 10^4        pi = 3.13544
= 5 x 10^5        pi = 3.13911
= 5 x 10^6        pi = 3.14182
```

用随机投点法计算定积分

- 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 $G = \int_0^1 f(x)dx$
  - 显然：积分G的值等于图中绿色面积
- 在图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)** 下的概率为



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

  - 假设：向单位正方形内随机地投入n个点(xi, yi)
  - 若有k个点落入G内，则随机点落入G内的概率： $G \approx \frac{k}{n}$

用随机投点法计算定积分

```
double darts(int n){
    srand((unsigned)time(NULL));  // 初始化随机数发生器
    int k = 0;
    for (int i = 1; i <= n; i++) {
        double x = rand() / double(RAND_MAX);
        double y = rand() / double(RAND_MAX);
        if ( y <= f(x) ) k++;
    }
    return  k/double(n);
}
```

$$G \approx \frac{k}{n}$$

用随机投点法求解非线性方程组

- 问题：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，…，x_n 是实变量
- $f_i()$ 是未知量 x_1，x_2，…，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

$$x_1^*, x_2^*, \cdots, x_n^*$$

非线性方程求根

- 线性系统
  - 线性：两个变量间可用直角坐标中一段直线表示的一种关系
  - 由线性关系描述的系统满足叠加原理
    - 对简单输入的响应叠加起来可导出其对其他输入的响应
- 非线性系统
  - 非线性：是指不满足线性叠加原理的性质
  - 非线性科学：研究各类系统中非线性现象共性和定量的规律
  - 非线性科学是当今科学发展的一个重要研究方向
  - 非线性科学中较成熟的部分是非线性动力学

非线性方程求根

- 非线性科学包括3个主要部分：孤立波，混沌，分形
  - 孤立波：是在传播中形状不变的单波（特别是孤立子）
    - 如光导纤维通信
  - 混沌：是一种由确定性规律支配却貌似无规的运动过程
    - 动物种群消长、脑电波、天气预报、社会经济活动等
  - 分形：是一个几何概念，由云彩、海岸线、树枝、闪电等不规则但具有某种无穷嵌套自相似性的几何图形抽象概括得出
- 非线性方程的求根是其中一个不可或缺的内容
  - 但是：非线性方程的求根非常复杂……

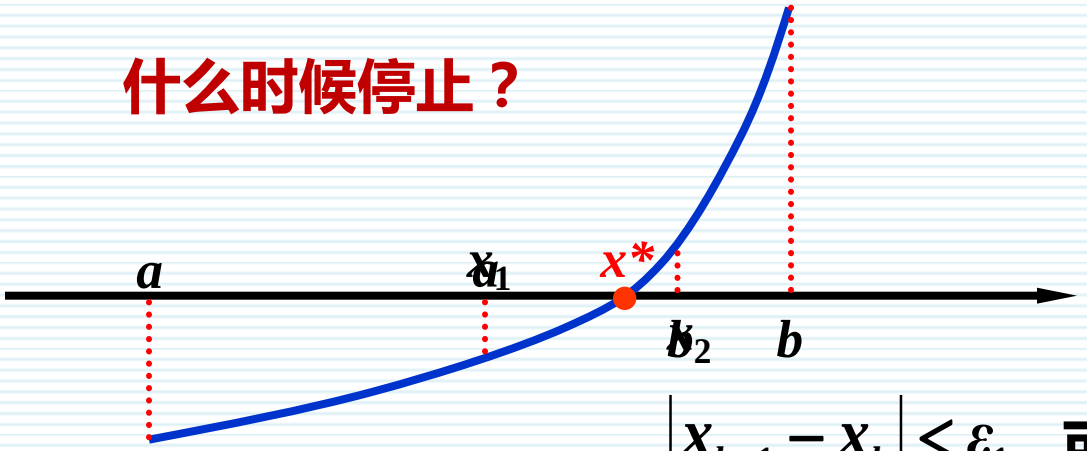
非线性方程求根

- 通常非线性方程的根的情况非常复杂：

$$\begin{cases} \sin(\dfrac{\pi}{2}x) = y \\ y = \dfrac{1}{2} \end{cases}$$ **无穷组解**

$$\begin{cases} y = x^2 + a \\ x = y^2 + a \end{cases} \Rightarrow \begin{cases} a = 1 & \text{无解} \\ a = \dfrac{1}{4} & \text{一个解} \\ a = 0 & \text{两个解} \\ a = -1 & \text{四个解} \end{cases}$$

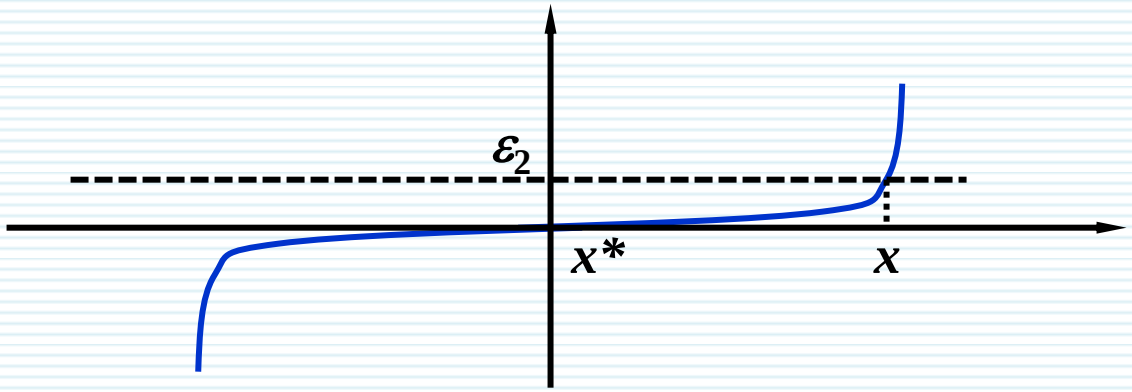
非线性方程求根



$$\left|x_{k+1}-x_k\right|<\varepsilon_1 \quad \text{或} \quad \left|f(x)\right|<\varepsilon_2$$

- 非线性方程通常只在某个区域内可能存在唯一解
- 并且通常难以得到非线性方程的精确解
  - 即便是很简单的方程式，例如：$e^x-\cos(\pi x)=0$
- 因此：通常采用迭代法求解非线性方程
- 迭代法的基本思想：$f(a)\cdot f(b)<0 \Rightarrow \exists x$ 使得 $f(x)=0$

采用对分法求解非线性方程



While(|a-b|>eps)

x = (a+b)/2

计算f(x)的值

若(|f(x)|<eps) 则以x为方程解

若f(x)*f(b)<0 修正区间为[x,b]

若f(a)*f(x)<0 修正区间为[a,x]

End while

主要问题

- **收敛速度为1/2（较慢）**
- **使用条件限制较大**
- **且只能求一个根**
- **不能保证 x 的精度**

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
- 按预先选定的分布（正态、均匀分布等）逐一选取随机点x
  - 计算目标函数Φ(x)，满足精度要求的随机点作为近似解
- 在算法的搜索过程中
  - 假设第k步随机搜索得到的随机搜索点为x_k
  - 在第k+1步，计算出下一步的增量Δx_k
  - 从当前点 x_k 依 Δx_k 得到第k+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
- 一般而言：随机化算法费时，但设计思想简单，易于实现
  - 对于精度要求高的问题，可以提供较好的初值

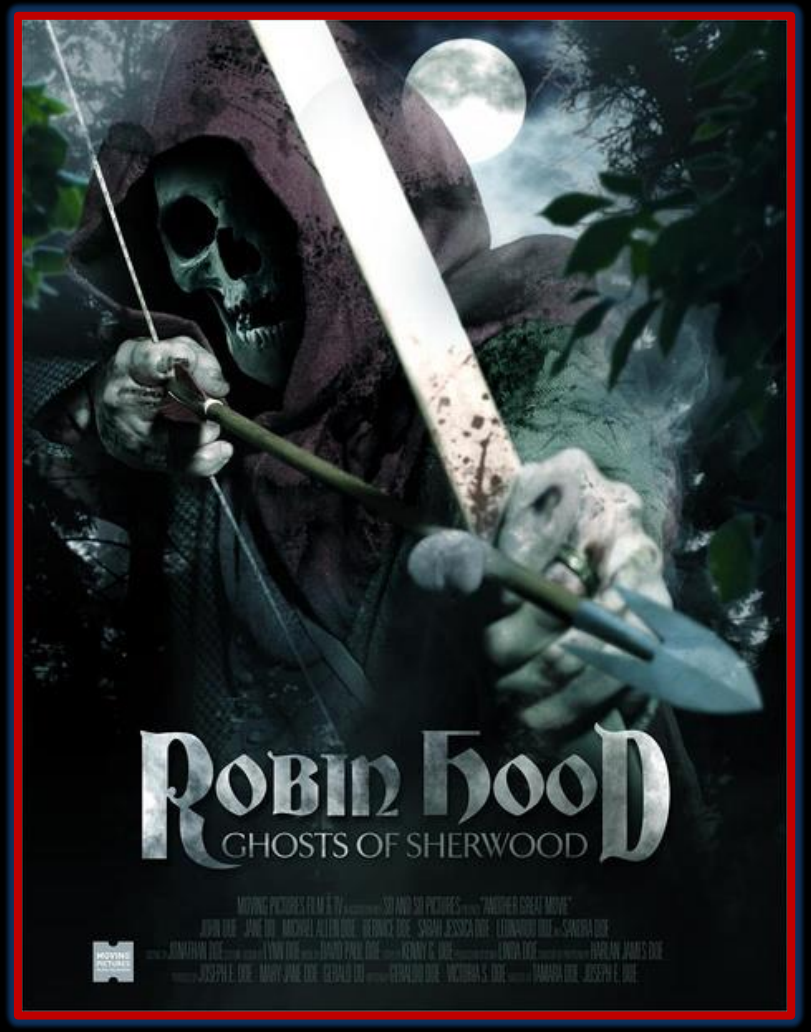
用随机投点法解非线性方程组

```
while((min > epsilon)&&(k < nstep)){ //a为步长因子
   if (fx < min){    // 搜索方向正确，增大步长因子
      min = fx;   a *= k;  }
   else{   // 搜索方向错误，减少步长因子
      bad++;   if (bad > M) a/= k;  }
   for (int i = 1; i  <= n; i++) {   // 生成随机搜索方向和增量
      r[i] = 2.0 * rand () – 1;      // r为搜索方向向量
      dx[i] = a * r[i];              // 计算随机搜索增量
      x[i] += dx[i];                 // 计算下一个随机搜索点
   }
   fx = f(x, n);   // 计算目标函数值
}
if (fx < epsilon)  return true;
else   return false;
```

舍伍德算法

（ Sherwood Algorithm ）

Robin Hood: Ghosts of Sherwood



导演: 奥利弗·克雷克尔　主演: Martin Thon / Ramona Kuen

上映日期: 2012-11-09（首映：德国）

舍伍德算法的基本设计思想

- 设：A是一个确定性算法
  - 设：X_n 是算法A的输入规模为n的实例的全体
  - 对于$x \in X_n$：算法A所需的计算时间记为 $t_A(x)$
  - 则：当问题的输入规模为n时，算法A所需的平均时间为

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

  - 显然可能存在$x \in Xn$ 使得： $t_A(x) >> \bar{t}_A(n)$
- 舍伍德算法的基本设计思想：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：

$$t_B(x) = \bar{t}_A(n) + s(n) \quad s.t. \;\; s(n) \ll \bar{t}_A(n)$$

舍伍德算法

- 其实我们已经接触过舍伍德算法思想
  - 分治法示例5：快速排序（Quick Sort）
  - 分治法示例6：线性时间选择
    - 给定线性序集中n个元素和一个整数 k（1≤k≤n）
    - 要求找出这n个元素中第k小的元素
- 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

```
int select(int A[], int start, int end, int k) {
    if (start ==end) return A[start];
    int i = Partition(A, start, end); //划分元位置
    int n = i-start+1;  // 左子数组A[start :i]的元素个数
    if (k<=n)
        return select (A, start, i, k);
    else
        return select(A, i+1, end, k-n);
}
```

最坏情况下，算法select需要O(n^2)计算时间

但可以证明，算法运行的平均时间为O(n)

舍伍德算法

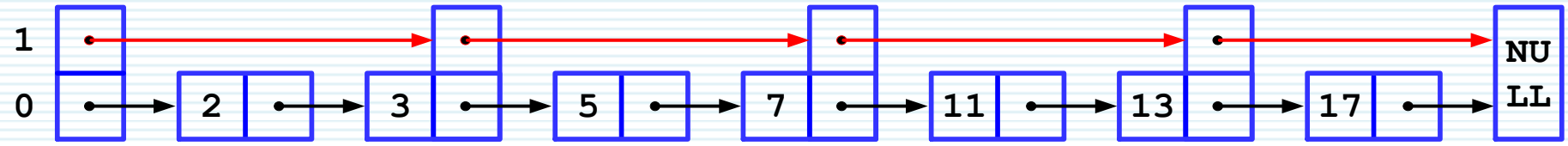
```
void shuffle(int a[], int n){
    srand((unsigned)time(NULL));
    for (int i=0; i<n; i++){
        int k = rand() % (n-i) + i;
        swap(a[i],  a[k]);
    }
}
```

- 常见情况：给定确定性算法无法直接改造成舍伍德型算法
- 此时可借助随机预处理技术帮助解决问题
  - 即：不改变原有的确定性算法，仅对其输入进行随机洗牌
  - 结果同样可收到舍伍德算法的效果

舍伍德算法示例

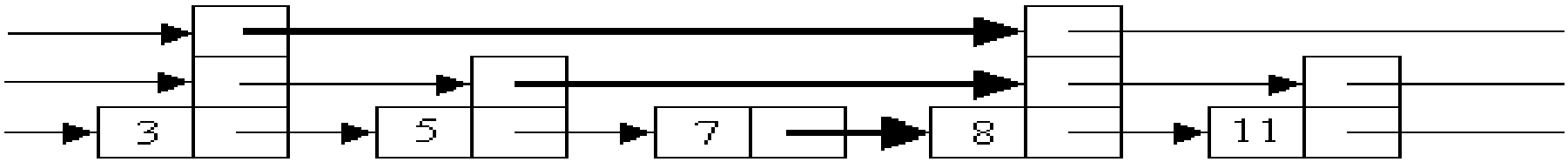
跳跃表

舍伍德算法示例：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 Ω(n) 计算时间
- 怎样提高有序链表的搜索效率？
  - 提示：可以在有序链表的部分结点处增设一个附加指针
  - 搜索时可借助附加指针跳过链表中若干结点以加快搜索速度
  - 这种增加了前向附加指针的有序链表称为：跳跃表

跳跃表



- 将一个含有n个元素的有序链表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 $i\times 2^k$ 处（$i\geq 0$）
  - 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$\lceil \log n \rceil$级结点

舍伍德算法示例：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关于有序集的成员搜索运算
    - 搜索、插入和删除等运算的平均时间为 O(logn)
  - 但它不适用于集合动态变化的情况
    - 元素的插入和删除会破坏完全跳跃表原有的平衡状态
    - 进而影响对后续元素的搜索效率
- 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n的一定比例范围内

舍伍德算法示例：跳跃表

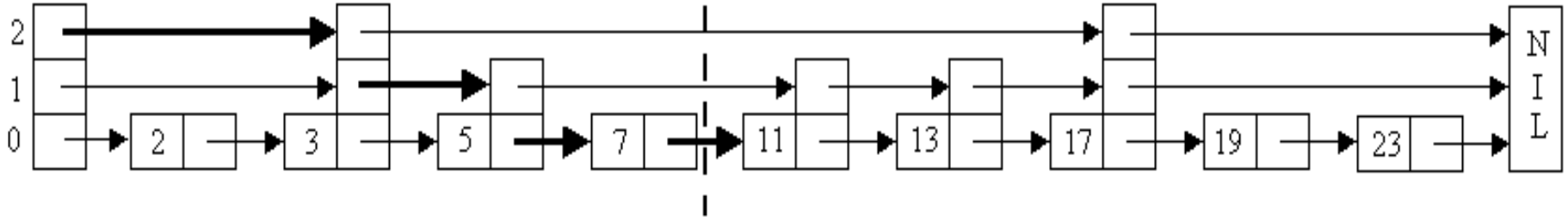
- 使跳跃表中k级结点数维持在n的一定比例范围内
  - 问题1：建表时应选择哪些结点为其增加附加指针？
  - 问题2：对每个选中的结点应设置多少个指针？
- 根据舍伍德算法思想：完全采用随机化方法来确定！
  - 提示：在一个完全跳跃表中
    - 50%的指针是0级指针
    - 25%的指针是1级指针…
    - $(100/2^{k+1})\%$的指针是k级指针
  - 思考：怎样建立和维护跳跃表？

舍伍德算法示例：跳跃表

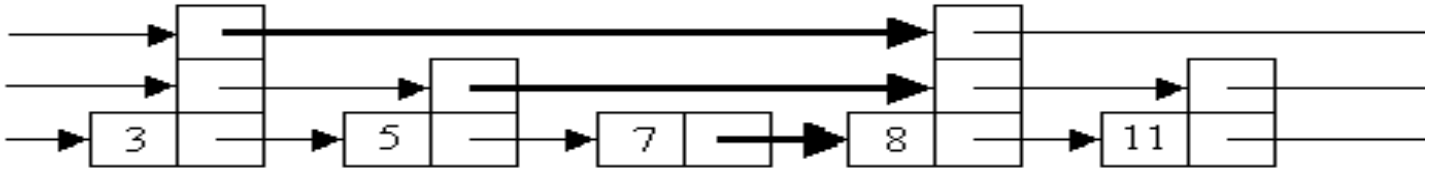
- 插入元素时维持完全跳跃表的指针平衡性
  - 在插入一个元素时
    - 以概率1/2引入一个0级结点
    - 以概率1/4引入一个1级结点 …
    - 以概率 $1/2^{k+1}$ 引入一个k级结点
    - 让一个 i 级结点指向下一个同级或更高级的结点
      - 它所跳过的结点数不再准确地维持在2^i-1
    - 由此实现元素插入或删除时维持其指针平衡性

舍伍德算法示例：跳跃表

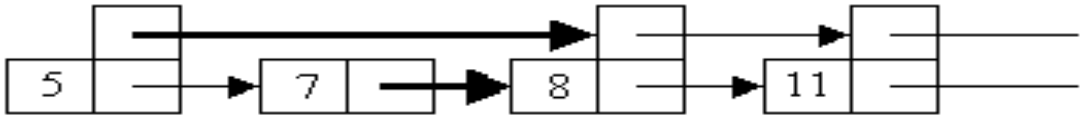
- 插入元素时维持完全跳跃表的指针平衡性



插入元素8



(a)



(b)

舍伍德算法示例：跳跃表

- 关键问题：确定新结点的级别
  - 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
  - 为此可以事先确定一个实数0<p<1
    - 要求在跳跃表中维持：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
  - 实现方法
    - 在插入新结点时首先将其级别初始化为0
    - 然后用随机数发生器反复地生成：q∈[0，1]
    - 如果q<p：则使新结点级别增加1（直至q≥p为止）

舍伍德算法示例：跳跃表

- 由上述确定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$p^i(1-p)$
  - 问题：有可能导致结点的级别过高
    - 设定新结点级别的上界为： $\log_{1/p} n$
    - 其中：n是当前跳跃表中结点个数
    - 则跳跃表中任一结点的级别均不超过 $\log_{1/p} n$

舍伍德算法小结

- 舍伍德算法总能求得问题的正确解
- 舍伍德算法的应用场景：确定性算法
  - 当算法在最坏情况下与平均情况下的计算复杂度相差较大时
  - 可通过引入随机性将它其改造成一个舍伍德算法
  - 目的是减少或消除问题的不同实例在计算时间上的差别
- 舍伍德算法的精髓
  - 不是避免算法行为的最坏情况
  - 而是设法消除这种最坏行为与特定实例之间的关联性

拉斯维加斯算法

（Las Vegas Algorithm）

拉斯维加斯算法

- 舍伍德算法
  - 计算复杂度对所有实例而言相对均匀
  - 但其平均时间复杂度不会比相应的确定性算法有所降低
- 拉斯维加斯算法
  - 是一种采用随机决策的搜索算法
    - 在搜索过程中不是通过计算得到局部的决策结果
    - 而是采用随机猜测的方式对分支的选择做出决策
  - 能够显著改进算法的计算性能
  - 对某些迄今为止找不到有效算法的问题也能得到满意结果

拉斯维加斯算法示例

n-皇后问题

拉斯维加斯算法求解n-皇后问题

- 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可言
- n-皇后问题的拉斯维加斯算法
  - 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

拉斯维加斯算法求解8-皇后问题

```
row = 1
repeat
   spotsPossible = 0   // 可用的位置
   for i = 1 to 8 do
      if i is not attacked then
         spotsPossible = spotsPossible + 1
         if uniform(1, spotsPossible) = 1 then
            try = i
         end if
      end if
   end for
   if spotsPossible > 0 then
       result[row] = try ;   row = row + 1
   end if
until spotsPossible = 0  or row = 9
return (spotsPossible > 0)
```

拉斯维加斯算法求解n-皇后问题

- 注意：拉斯维加斯算法有可能找不到问题的解 **如何改进？**
- 思路1：多次重复运行算法，直至找到问题的解为止
- 思路2：将随机放置策略与回溯法相结合
  - 首先在棋盘上的若干行中随机地放置几个互不攻击皇后
  - 然后在后继行中用回溯法继续放置剩余的皇后
  - 直至找到一个解或宣告失败为止
  - 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

拉斯维加斯算法

```
// 参数：x 为问题输入；y 为问题的解
void obstinate(Object x, Object y){
    bool success = false;
    while (!success){
        success = lasvegas(x,y);
    }
}
```

- 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
- 解决方案：反复调用拉斯维加斯算法，直到找到问题的一个解

拉斯维加斯算法

- 拉斯维加斯算法的平均运行时间估计
  - 设：p(x)是对输入x调用LV算法获得一个解的概率
    - 一个正确的LV算法应该对所有输入x均有：p(x)>0
  - 设：t(x)是obstinate找到实例x的一个解所需的平均时间
  - 设：s(x)和f(x)是算法对x求解成功或失败所需的平均时间
    - 则有：

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

  - 解此方程可得：

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

拉斯维加斯算法与回溯法结合求解8-皇后问题

#Placed	P	S	F	T
0	1.0000	114.00	--	114.00
3	0.4931	13.48	15.10	29.01
6	0.1375	9.05	6.98	53.50

- P：表示算法获得一个解的概率
- S：表示一次成功搜索访问的结点数的平均值
- F：表示一次不成功搜索访问的结点数的平均值
- T：表示反复调用算法最终找到一个解时访问的结点数的平均值

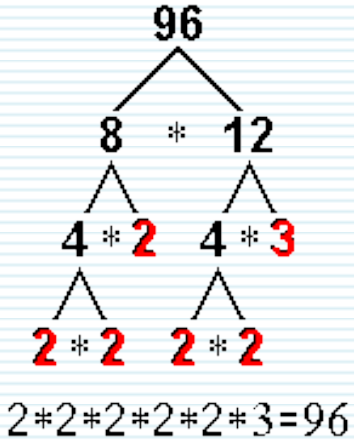
$$T = S + \frac{(1-p)\times F}{p}$$

拉斯维加斯算法示例

整数因子分解问题

整数因子分解的拉斯维加斯算法

- 整数n的因子分解问题：设n>1是一个整数
  - 要求找出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
    - 其中：$m_1, m_2, \ldots, m_k$ 是k个正整数



- 整数n的因子分割问题：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
  - 若：能够快速判定给定整数的素性？
  - 则可将整数因子分解问题转化为整数的素因子分割问题

整数因子分割的一般算法

```
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;
}
```

埃拉托斯特尼筛法

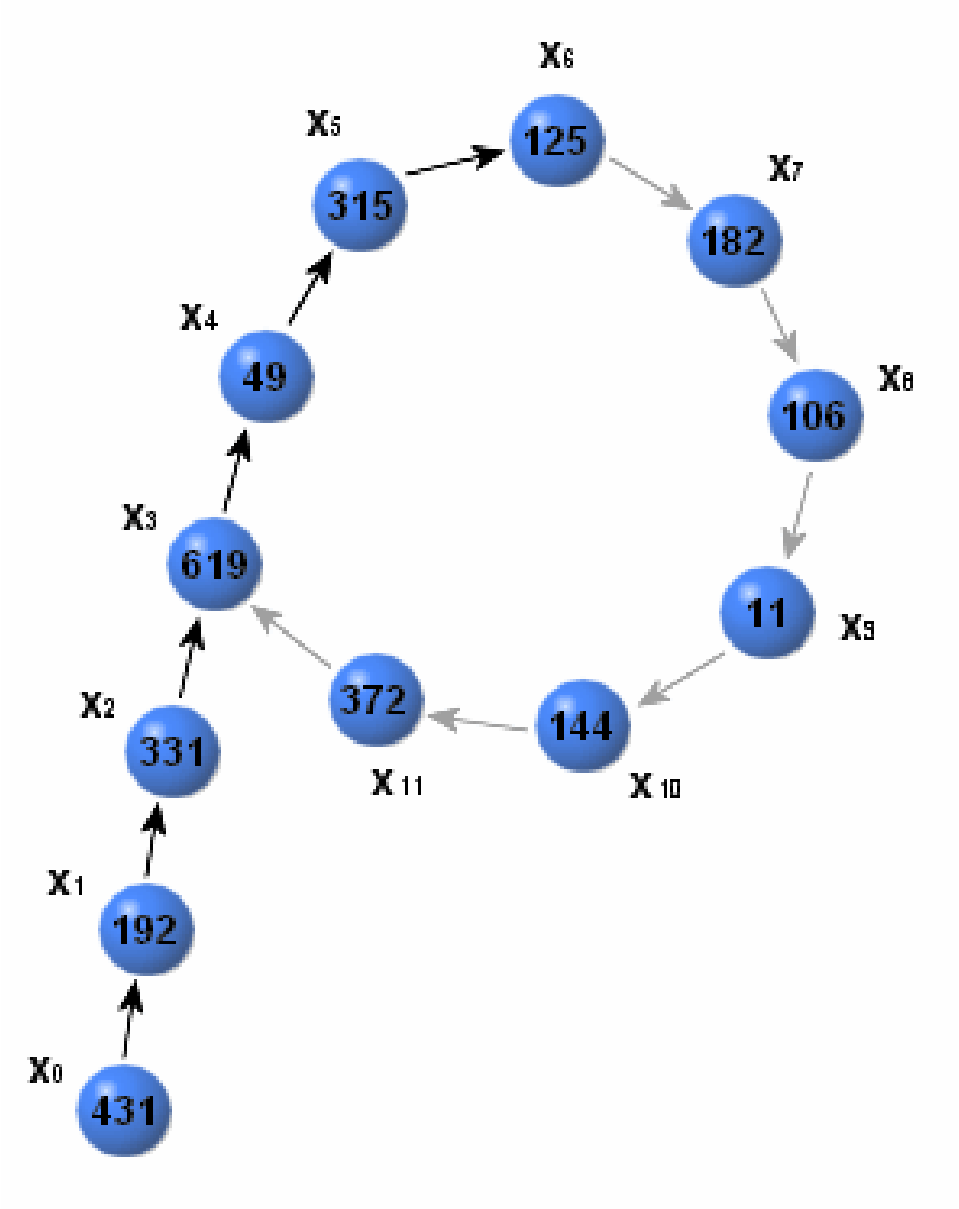
把不大于√n的所有素数的倍数剔除，剩下的就是素数

- 算法对范围在2～k区间内的所有整数进行了试除
  - 最坏情况下的算法复杂度为：$\Omega(\sqrt{n})$
  - n的位数：$m = \lceil \log_{10}(n+1) \rceil$ 算法复杂度：$\Theta(10^{m/2})$
  - 迄今为止尚未找到求解因子分割问题的多项式时间算法

Pollard-Rho 因子分解方法

- John M. Pollard (1975)
  - 生日悖论问题：当 $m>1.177n^{1/2}$ 时，碰撞的概率>50%
  - ρ-算法利用一个多项式g(x)函数模n作为伪随机序列生成器
    - 最常用的多项式方程是：$g(x) = (x^2+1) \bmod n$
    - 产生如下无穷序列： $x_1 = g(x_0),\ x_2 = g(x_1),\ \cdots$
  - 当碰撞发生时，该序列会产生环 …… 为什么？
    - 因为该序列每个元素的取值都依赖于前一个元素值
  - 之所以称为ρ-algorithm就是因为该序列画出来很像ρ

Pollard-Rho 因子分解方法



Pollard-Rho 因子分解方法

- ρ-算法常用的伪随机序列生成器：g(x) = （ x^2+1 ） mod n
  - 产生如下无穷序列： $x_1 = g(x_0),\ x_2 = g(x_1),\ \cdots$
    - 当碰撞发生时该序列会产生环
  - 思考：环的出现有何意义？
    - 设： x_k 和 x_m 为序列中发生碰撞的两个元素（ m>k ）
    - 有： gcd (x_m - x_k ，n) = p
    - 其中：p 为 n 的非平凡素因子（若n为合数）

- **Pollard, J. M. (1975), "A Monte Carlo method for factorization", BIT Numerical Mathematics 15 (3): 331–334, doi:10.1007/bf01933667**

Pollard-Rho 算法

- Pollard算法是整数因子分解的拉斯维加斯算法
  - 算法开始时：选取随机数 $x_1 \in [2, n-1]$
  - 然后递归地由公式： $x_i = (x_{i-1}^2 + 1) \bmod n$
    - 产生如下无穷序列： $x_1, x_2, \cdots, x_k, \cdots$
  - 对于其中下标为 $i = 2^k$ 和 $2^k < j \le 2^{k+1}$ 的元素
  - 计算出$x_j - x_i$与n的最大公因子：$p = \gcd(x_j - x_i, n)$
  - 如果p是n的非平凡因子（即：$p \neq 1$）
    - 则实现对n的一次分割：算法输出n的因子p

Pollard-Rho 算法

```
// 回顾：求最大公约数
int gcd( int a, int b) {
    int remainder;
    while (b != 0) {
        remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
}
```

Pollard-Rho 算法

```
int rho(int n) {
    srand((unsigned)time(NULL));
    int  x = rand() % n,  xfix = x,  cyc = 2,  p = 1;
    while (p == 1) {
        for (int i =1; i <= cyc && p == 1; i++) {
            x = (x*x+1) % n; p = gcd(x - xfix, n);
        }
```

- 为找到n的一个因子p： while循环需执行约 $p^{1/2}$ 次
- 由于 n 的最小素因子 $p \le n^{1/2}$
- 故Pollard算法可在 $O(n^{1/4})$ 时间内找到n的一个素因子

拉斯维加斯算法小结

- 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找不到问题的解
- 引入随机决策直接导致算法的准确性下降
  - 这一点可以由算法效率的巨大提升来弥补（多运行几次）
  - LV算法求得正确解的概率随计算时间的增加而增大
  - 因此为了提高拉斯维加斯算求解成功的概率
    - 可以使用该算法对同一实例重复多次执行求解
  - 还可以采用混合策略来改进算法性能
    - 通过随机决策迅速缩小搜索空间
    - 到达某一阀值后改用完全的搜索

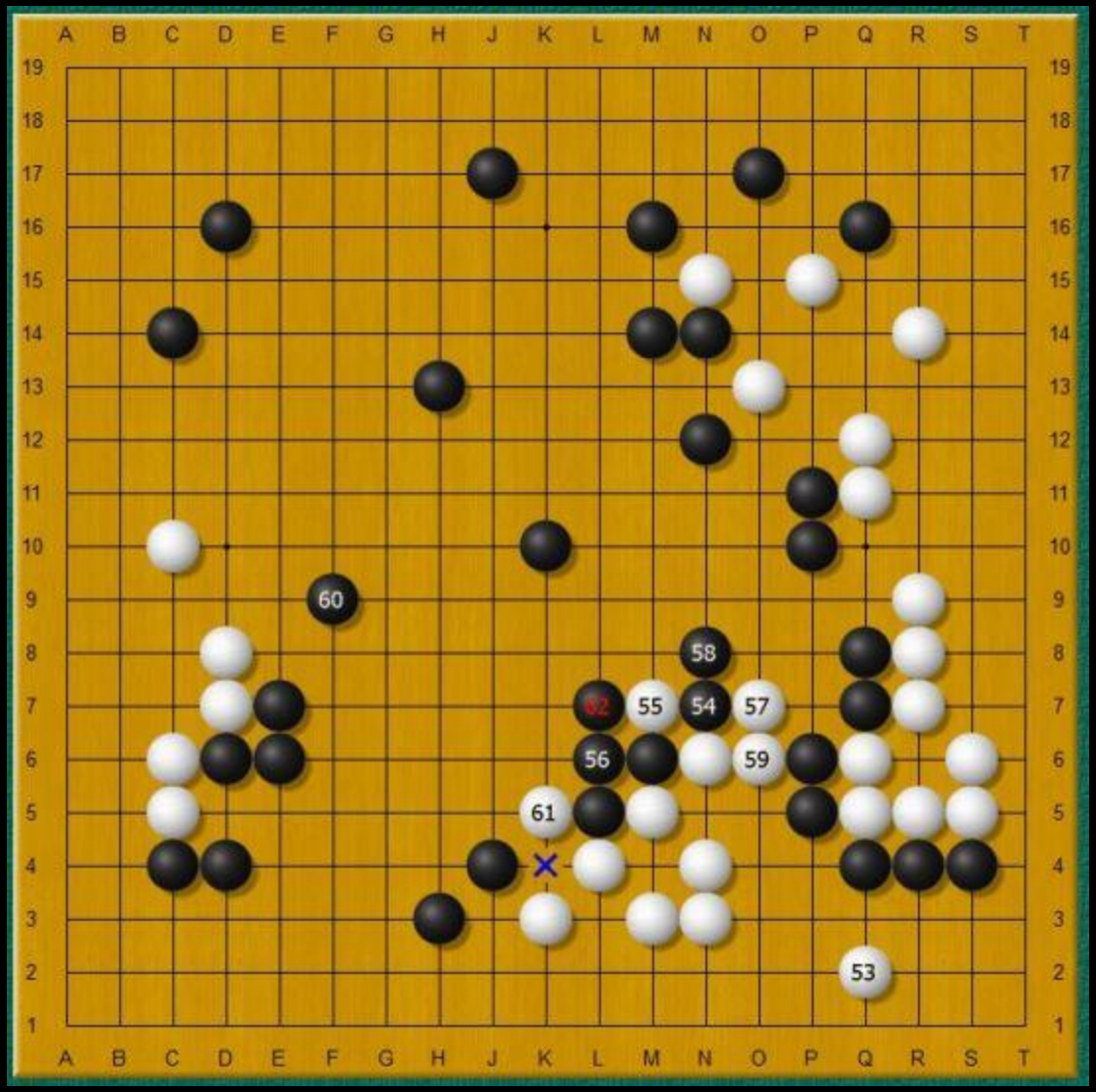
蒙特卡罗算法

（Monte Carlo Algorithm）

武宫正树让四子不敌软件 电脑大局观强擅弃子战术

2012年03月17日

第一局：武宫正树让ZEN五子



最终黑棋盘面11目，减去5目的还子，ZEN以6目的优势取胜

2016年3月5日：日本宣布启动DEEP ZEN GO项目

3月9日：AlphaGo与李世石的五番棋大战拉开帷幕

蒙特卡罗算法

- 常会遇到这样一些问题：不论采用确定性算法或随机化算法
  - 均无法保证每次计算都能得到问题的正确解答
- 蒙特卡罗算法也称为统计模拟方法
  - 是一种以概率统计理论为指导的重要数值计算方法
    - 使用（伪）随机数来解决计算问题……采样
  - 广泛应用于金融工程学，宏观经济学，计算物理学等领域
- 蒙特卡罗算法的两个主要特征
  - 结果正确的概率比错误的概率大（错误的概率有限）
  - 资源的使用是确定的（主要优点）

蒙特卡罗算法

- 蒙特卡罗算法的优势：设p是一个实数：0.5<p<1
  - 若一个MC算法对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
    - 并称(p – 0.5)是该算法的优势
- 蒙特卡罗算法的一致性
  - 如果蒙特卡罗算法对于同一实例不会给出不同的解答
    - 则称该蒙特卡罗算法是一致的
- 对于一个一致的、p正确的蒙特卡罗算法
  - 要提高获得正确解的概率，只要执行该算法若干次（采样）
  - 并选择出现频次最高的解即可

蒙特卡罗算法

- 设：ε和δ为两个正实数，且：ε+δ<0.5
  - 设：MC(x)是一个一致的（0.5+ε）正确的蒙特卡罗算法
  - 若：重复调用MC(x)算法2m-1次
  - 则通过MC算法得到正确解的概率至少为1-δ
  - 其中：

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

  - 在实际应用中大多数MC算法经重复调用后正确率提高很快

蒙特卡罗算法

- 偏真蒙特卡罗算法
  - 设MC(x)是求解某个判定问题的蒙特卡罗算法
    - 如果：当MC(x)返回True时，解总是正确的
    - 且仅当：MC(x)返回False时，有可能产生错误的解
  - 称这类蒙特卡罗算法为：偏真算法
- 偏真算法的意义：当多次调用偏真算法时
  - 只要有一次返回的结果为True，则相应的解为True
  - 可以证明：对于正确率为55%的偏真算法而言
    - 只需重复调用4次，即可将解的正确率提高到95%
    - 如果重复调用6次，可以将解的正确率提高到99%
  - 对于偏真算法而言：p正确的要求可以从p>0.5放松为p>0

蒙特卡罗算法

- 考查一般情况：所讨论的问题不一定是一个判定问题
  - 设： y_0 是所求解问题的一个特殊的解答
  - 对于算法MC(x)，若存在问题示例的子集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
    - 则称这类蒙特卡罗算法为：偏 y_0 的算法
- 重复调用一个一致的，p正确的偏 y_0 的蒙特卡罗算法 k 次
  - 所得算法仍是一个一致的偏 y_0 的蒙特卡罗算法
  - 注意：得到正确解的概率由p提高到： $1-(1-p)^k$
  - 即：得到一个 $1-(1-p)^k$ 正确的蒙特卡罗算法

蒙特卡罗算法示例

主元素问题

主元素问题

```
int majority(int *A, int n){    // 判定主元素的蒙特卡罗算法
    srand((unsigned)time(NULL));
    int k = rand() % n  , m = 0,  x = A[k];  // 随机选择元素
    for (int i = 1; i<=n; i++){ if (A[i]==x) m++; }
    return (m > n/2);  // m>n/2 时A含有主元素
}
```

- 问题描述：设A[1:n]是一个含有n个元素的数组
  - 当|{i|A[i]==x}|>n/2时，称元素x是数组T的主元素
  - 要求：对于给定的输入数组A，判定其是否包含主元素

主元素问题

```
int MonteCarlo(int *A, int n, double p){
    int k = ceil(log(1/p)/log(2));      // 思考：k如何确定？
    for (i = 1; i<=k; i++){  if (majority(A, n)) return 1; }
    return 0;
}
```

算法复杂度：O(nlog(1/p))

- 对于任何给定的 0 < p < 0.5
  - 算法MC重复调用⌈log(1/p)⌉次majority算法
  - 这是一个偏真蒙特卡罗算法：其错误概率小于p
    - 若：A含有主元素，则算法以大于0.5的概率返回1
    - 若：数组A不含有主元素，则算法必然返回0

蒙特卡罗算法示例

素数测试问题

素数测试问题

- 素数的测试是初等数论研究的一个重要的基本问题
  - 快速素数检验是目前大部分公钥密码体系的关键
- Wilson定理给出了一个数是素数的充要条件（1770年）
  - 同余式：如果两个正整数a和b之差能被n整除
    - 则称： a和b对模n同余
    - 记为： $a \equiv b \pmod{n}$
  - Wilson定理：判定给定的正整数n是素数的充要条件是
    - $(n-1)! \equiv -1 \pmod{n}$

素数测试问题

- 费马小定理（ Fermat Theory，1636年）
  - 若：p 是一个素数，a是一个整数（0<a<p）
    - 且：gcd(a, p) = 1 （ a与p互质）
    - 则：$a^{p-1} \equiv 1 (\text{mod } p)$
  - 例如：67为一个素数，且有：$2^{66} \text{ mod } 67 = 1$
  - 费马小定理为素数判定提供了一个有利工具
    - 注意费马小定理只是素数判定的一个必要条件
    - 例如：341是合数，且：$2^{340} \text{ mod } 341 = 1$
    - 然而：$3^{340} \equiv 56 (\text{mod } 341)$
    - 为提高测试准确性可随机取值进行测试

用费马小定理进行素数测试

```
int Fermat(int a,  int n,  int p){  // 计算a^n mod p
    int  res;
    if (n == 0) return 1;
    else{
        res = Fermat(a, n/2, p);
        res = (res * res)%p;
        if (n%2) res = (res*a)%p;
    }
    return res;
}
```

算法复杂度：O(log(n))

判定p是否为素数：res = Fermat(a, p-1, p);

素数测试问题

- 费马小定理只是素数判定的一个必要条件
  - 满足费马小定理的整数中的合数称为Carmichael数
    - 排名前三的Carmichael数为：561，1105，和1729
    - Carmichael数非常少，在前1亿个整数中只有255个
  - 为了避免将Carmichael数当作素数，引入二次探测定理
- 二次探测定理（Miller-Rabin算法）
  - 若：p是一个素数，且：0<x<p
    - 则方程：$x^2 \equiv 1 \pmod{p}$ 的解为：x=1，x=p-1

素数测试问题

- 理解二次探测定理：
  - $x^2 \equiv 1 \pmod{p}$ 等价于：$(x-1)(x+1) \bmod p = 0$
    - 所以：p必须整除(x-1)或(x+1)
  - 由于p是素数，且：$0<x<p$
    - 所以：$x = 1$ 或 $x = p - 1$
- Miller-Rabin算法是目前主流的基于概率的素数测试算法
  - 在构建密码安全体系中占有重要的地位
  - 是构建密码安全体系时完成素数测试的最佳选择

Miller-Rabin算法

```
// 计算a^n mod p，并实施对n的二次探测
int MillerRabin(int a,  int n,  int p, int *comp ){
    int  res, x;
    if (n == 0) return 1;
    else{
        x = MillerRabin (a, n/2, p);
        res = (x * x)%p;
        if ((res == 1)&&(x != 1)&&(x != p-1))
                *comp = 1;
        if (n%2) res = (res*a)%p;
    }
    return res;
}
```

Miller-Rabin算法

```
// 素数测试的蒙特卡罗算法
int primeTest(int p){
        int a,  composite = 0;
        srand((unsigned)time(NULL));
        a = rand() % (n-3) + 2;
        power(a, p-1, p, &composite);
        if (composite || (res != 1)){
                return 0;
        }
        else   return 1;
}
```

primeTest 算法分析

- 算法primeTest返回0时
  - 整数n必为合数（确定性结果）
- 算法primeTest返回1时
  - 整数n在高概率意义下是一个素数
  - 仍可能存在合数n对于随机选取的基数a使得算法返回1
- 研究表明：当n充分大时，这样的基数a不超过(n-9)/4个
  - 由此可知：该算法是一个偏假3/4正确的蒙特卡罗算法
- 思考：算法性能该如何改进？
  - 重复调用k次，使得算法结果错误的概率不超过$(1/4)^k$
  - 这是一个很保守的估计，实际使用的效果要好得多

Miller-Rabin算法

```
// 重复调用k次primeTest算法的MonteCarlo算法
bool MonteCarlo(int p, int k){
    bool composite = false;
    srand((unsigned)time(NULL));
    for(int i=1; i <= k; i++){
        a = rand() % (n-3) + 2;
        power(a, p-1, p, &composite);
        if (composite || (res != 1))   return 0;
    }
    return 1;
}
```

蒙特卡罗算法小结

- 该算法用于求问题的准确解（精确解而不是近似解）
- 蒙特卡罗算法不能保证所求得的解是正确的
  - 即：只能在一定的概率意义上保证得到正确解
  - 但由于通常可以设法控制这类算法得到错误解的概率
  - 而且蒙特卡罗算法通常是简单高效的（实用价值高）
- 关于蒙特卡罗算法求解的正确性
  - 一般情况下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的解究竟是否正确！
  - 这一点可以看做是蒙特卡罗算法的缺陷……瑕不掩瑜


UESTC
1956