Design and Analysis of Algorithms

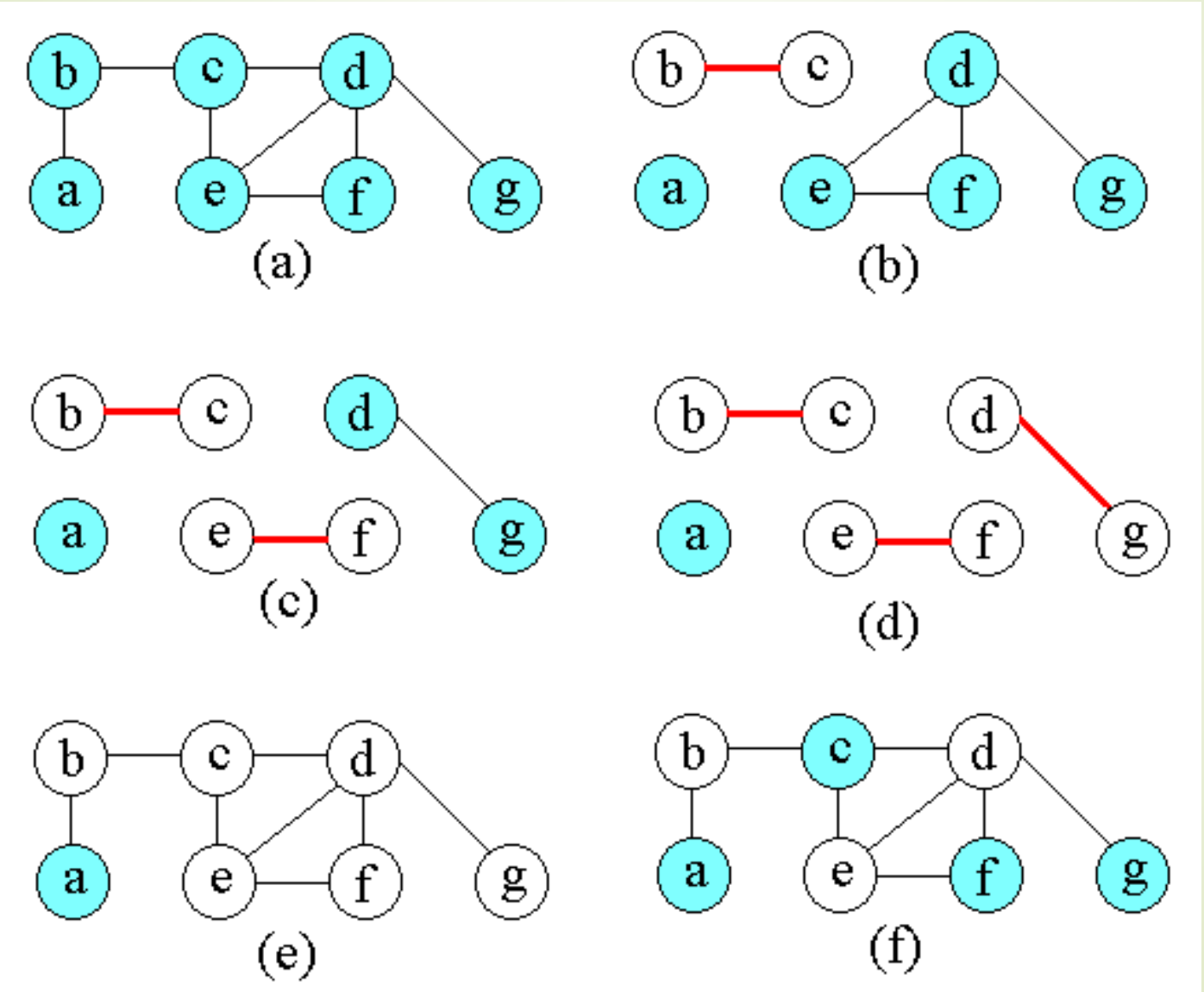
Hao Sheng

shenghao@buaa.edu.cn

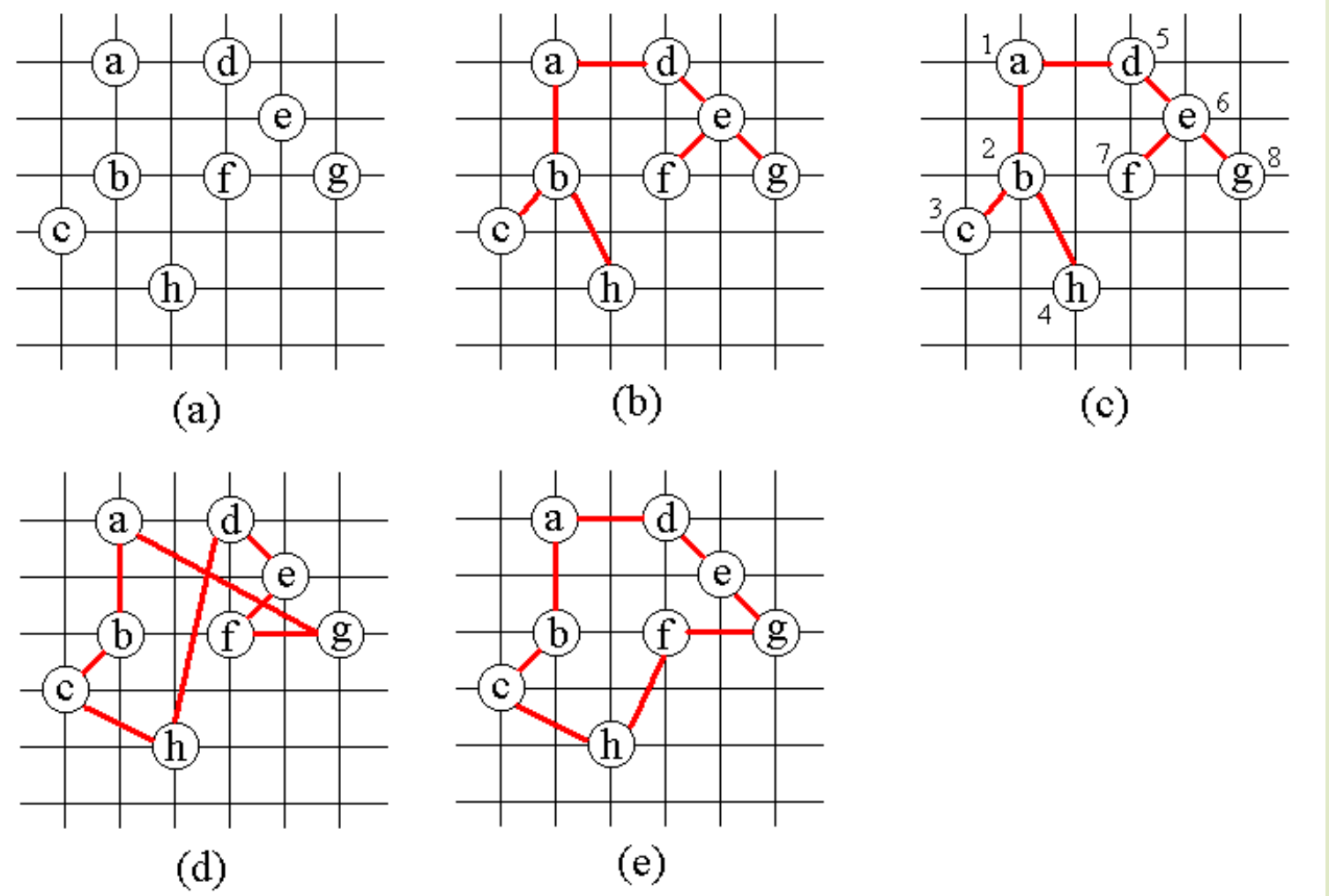
Last Section: Approximation Algorithm

1. 近似算法的**性能比** $r(s_a) = \max\left(\frac{f(s_a)}{f(s*)}, \frac{f(s*)}{f(s_a)}\right)$
2. 算法举例：
   1. 顶点覆盖问题
   2. 售货员旅行问题
   3. 集合覆盖问题

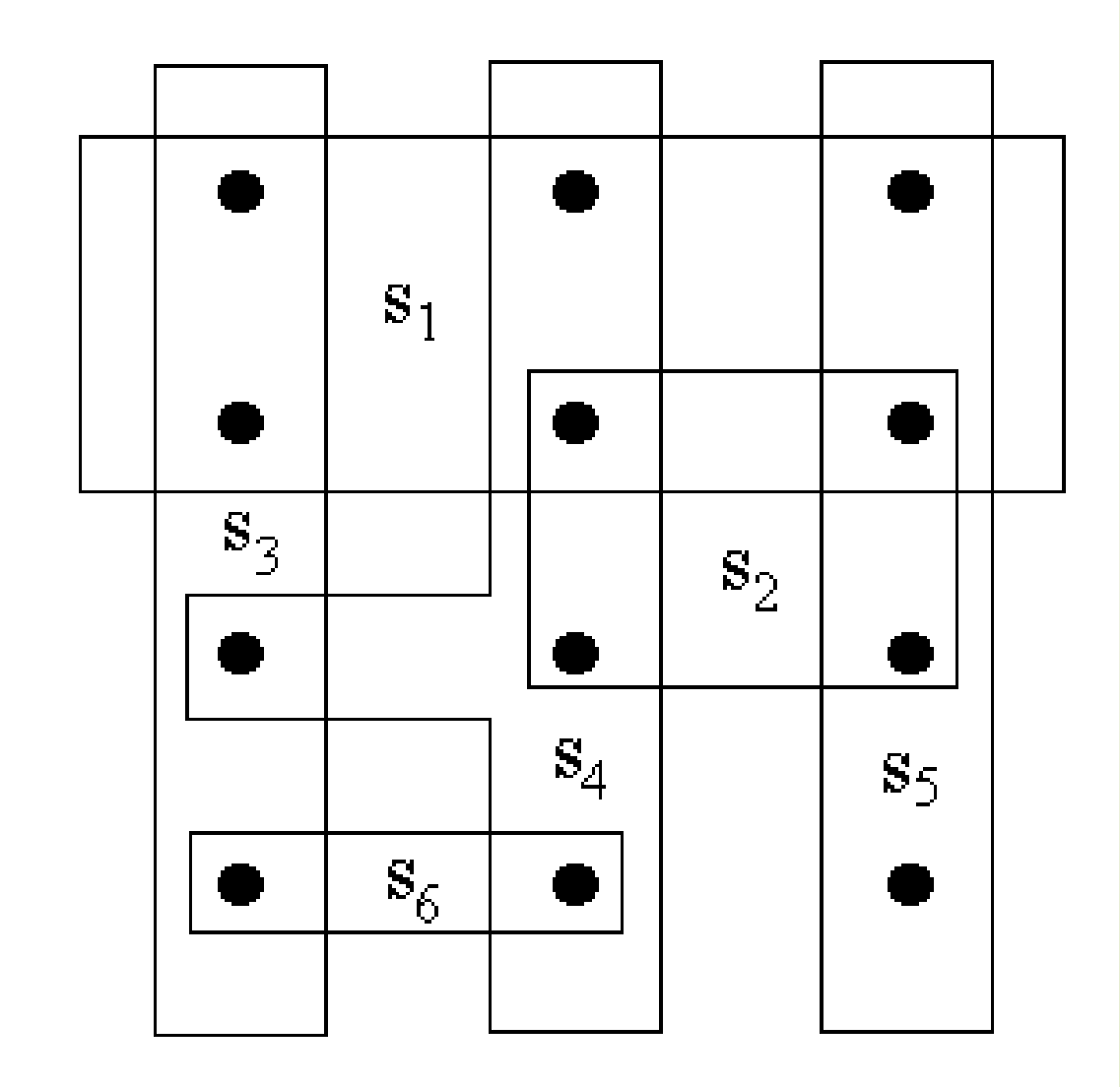
顶点覆盖问题的例子

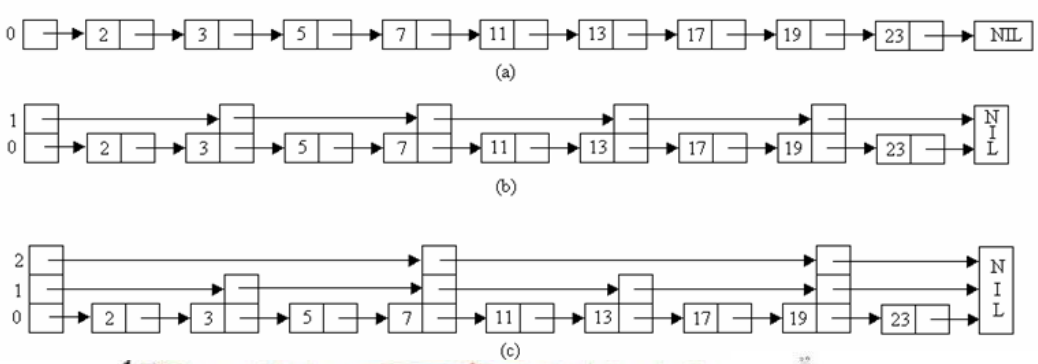


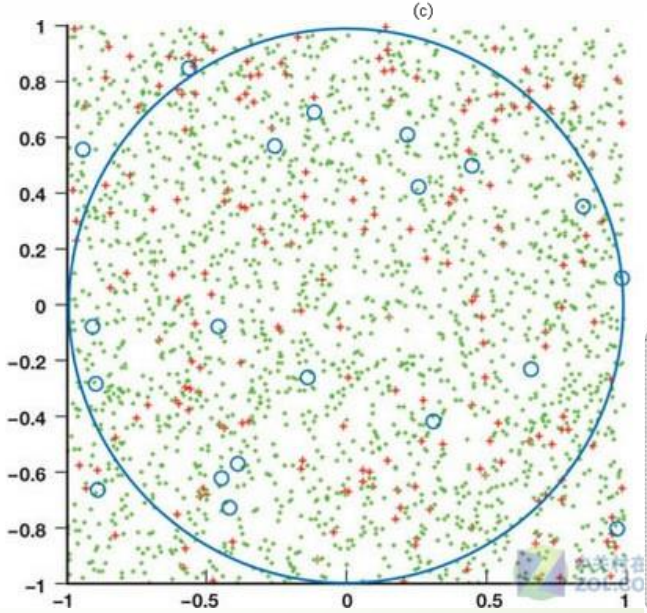
满足三角不等式的售货员旅行问题欧几里德类型实例

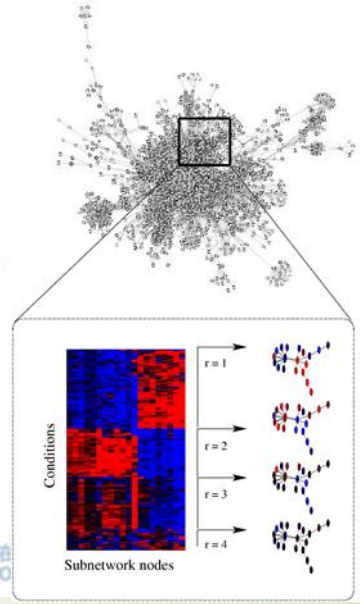


集合覆盖问题的近似算法









7. Probabilistic Algorithms
随机（概率）算法

Content

- 了解随机算法的基本特征
- 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

随机算法概述

- 随机算法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。

随机算法的特点

1. 当算法执行过程中面临选择时，随机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随机算法求解两次，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。

- 包括
  - 数值概率算法
  - 蒙特卡罗（Monte Carlo）算法
  - 拉斯维加斯（Las Vegas）算法
  - 舍伍德（Sherwood）算法

例：中华人民共和国成立的时间？

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。

•数值概率算法：

连续调用5次，得到的解：{1948±2， 1949±2， 1949±2， 1951±2， 1949±2}，解落入置信区间的概率是60%

随机算法的分类

- 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）。

•舍伍德算法：

连续调用5次，得到的解：{1949，1949， 1949，1949， 1949}，正确率100%

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。

•拉斯维加斯算法：

连续调用5次，得到的解：{1949， 1949， 1949， 1949， failure}，没有解的概率是20%

随机算法的分类

- 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。

•蒙特卡罗算法：

{1949， 2000， 1949， 1949， 1949}，解的正确率是80%

7.1 基本概念：随机数

7.1 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。
- 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a0, a1, …, an，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。如何选取该方法中的常数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，从直观上看，m应取得充分大，因此可取m为机器大数，如2^{32}或2^{64} 。

```
//随机数类
Const unsigned long maxshort = 65536L;
Const unsigned long multiplier = 1194211693L;
Const unsigned long adder = 12345L;

class RandomNumber
 {
  private:
        //当前种子
        unsigned long randSeed;
  public:
        //构造函数，默认值0表示由系统自动产生种子
        RandomNumber (unsigned long s = 0);
        //产生0到n-1之间的随机整数
        unsigned short Random (unsigned long n);
        //产生[0,1)之间的随机实数
        double fRandom (void);
  }
```

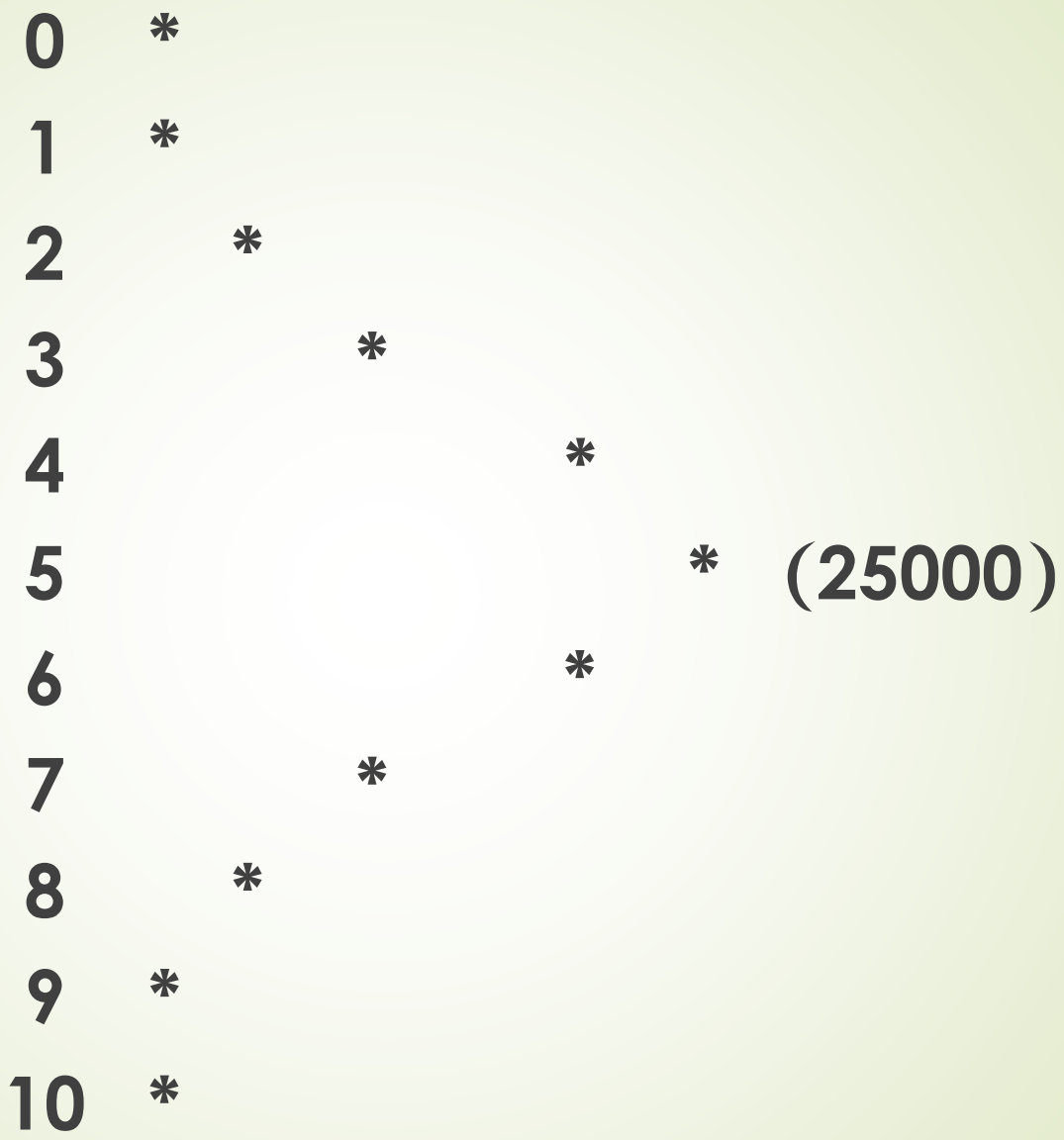
```
//产生种子
RandomNumber:: RandomNumber(unsigned long s)
{
    if (s == 0) randSeed = time (0);   //用系统时间产生种子
   else randSeed = s;   //由用户提供种子
}

//产生0~n-1之间的随机数
Unsigned short RandomNumber:: Random (unsigned long n)
{
    randSeed = multiplier * randSeed + adder;
    return (unsigned short) ( (randSeed >>16) % n );
}

//产生[0，1)之间的随机实数
Double RandomNumber:: fRandom (void)
{
   return Random (maxshort) / double (maxshort);
```

抛硬币实验


```
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

7.2 数值概率算法

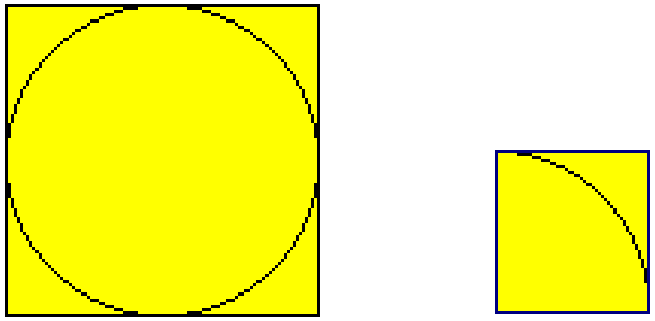
7.2 数值概率算法

数值概率算法（numerical randomized algorithm）用于求数值问题的近似解。

7.2.1 用随机投点法计算π值

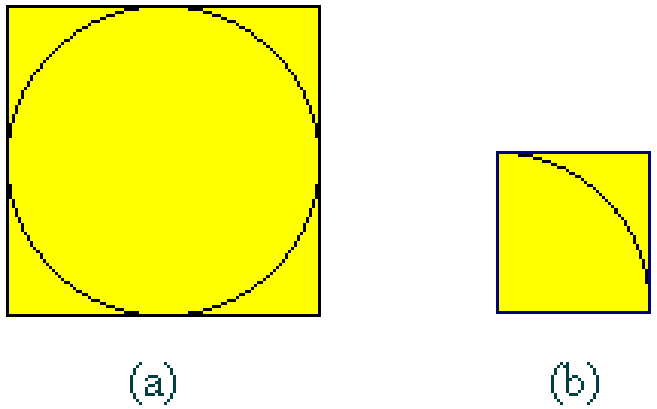
设有一半径为r的圆及其外切四边形。向该正方形随机地投掷n个点。设落入圆内的点数为k。由于所投入的点在正方形上均匀分布，因而所投入的点落入圆内的概率为 $\frac{\pi r^2}{4r^2}=\frac{\pi}{4}$ 。

所以当n足够大时，k与n之比就逼近这一概率。从而 $\pi \approx \frac{4k}{n}$ 。



在具体实现时，只要在第一象限计算即可：

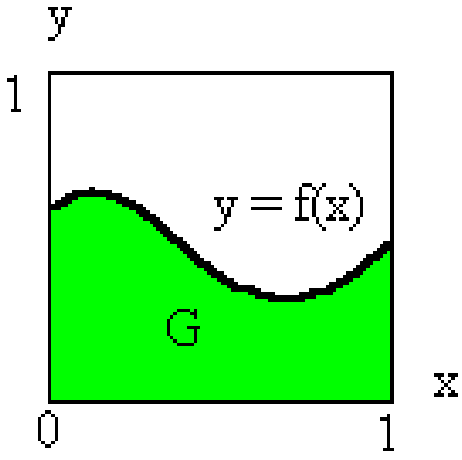
```
double Darts (int n)
{ // 用随机投点法计算π值
   static RandomNumber dart;
   int k=0;
   for (int i=1;i <=n;i++)
     {
     double x=dart.fRandom();
     double y=dart.fRandom();
     if ((x*x+y*y)<=1) k++;
     }
   return 4*k/double(n);
}
```



7.2.2 计算定积分

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 $I = \int_0^1 f(x)dx$ ，积分 I 等于图中的面积G。

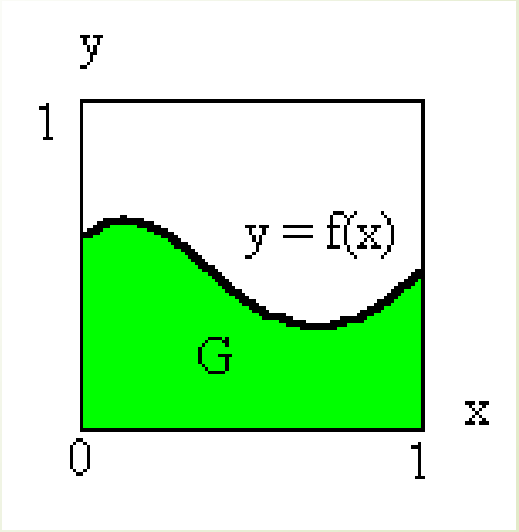


在图所示单位正方形内均匀地作投点试验，则随机点落在曲线下面的概率为：

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

假设向单位正方形内随机地投入n个点(xi，yi)，i = 1,2，…，n。随机点（xi，yi）落入G内，则yi ≤f（xi）。如果有m个点落入G内，则随机点落入G内的概率，即 $I \approx \frac{m}{n}$

```
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)
}
```



7.3 舍伍德算法

7.3 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。

7.3 舍伍德算法

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

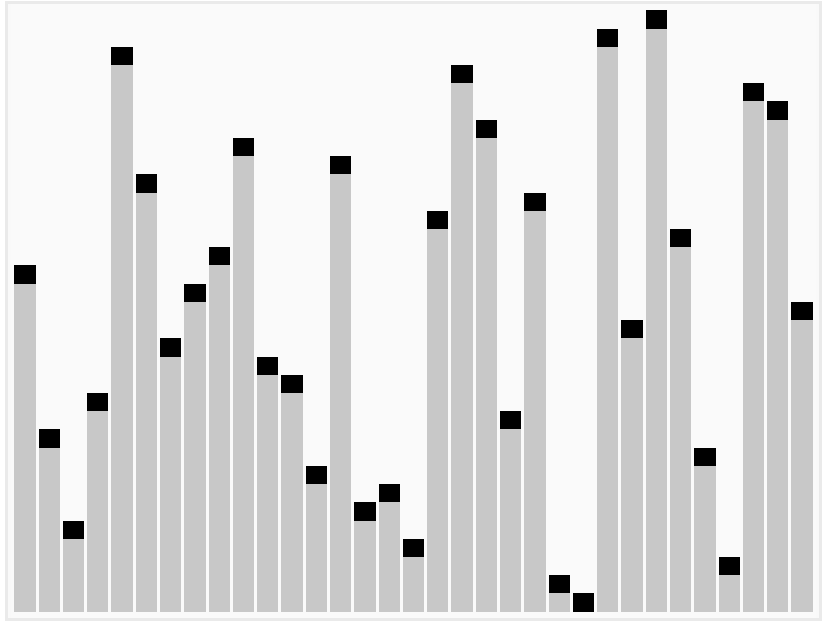
这显然不能排除存在$x \in Xn$使得 $t_A(x) >> \bar{t}_A(n)$ 的可能性。希望获得一随机化算法B，使得对问题的输入规模为n的每一个实例均有个 $t_B(x) = \bar{t}_A(n) + s(n)$

这就是舍伍德算法设计的基本思想。当s(n)与 $\bar{t}_A(n)$ 相比可忽略时，舍伍德算法可获得很好的平均性能。

7.3.1 随机快速排序算法

随机快速排序算法

- 快速排序算法
  - 算法的核心在于选择合适的划分基准
- 改变快速排序算法性能不稳定，即输入相关的问题



快速排序的步骤：

- ✓ 从数列中挑出一个元素，称为 "基准“
- ✓ 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区操作
- ✓ 递归地把小于基准值元素的子数列和大于基准值元素的子数列排序

随机快速排序算法

```
template <class Type>
void quicksort_random(Type A[],int low,int high)
{
    random_seed(0); /* 选择系统当前时间作为随机数种子 */
    r_quicksort(A,low,high);   /* 递归调用随机快速排序算法 */
}
void r_quicksort(Type A[],int low,int high)
{
    int k;
    if (low<high) {
        k = random(low,high);   /* 产生low到high之间的随机数k */
        swap(A[low],A[k]);          /* 把元素A[k]交换到数组的第一个位置*/
        k = split(A,low,high);    /* 按元素A[low]把数组划分为两个 */
        r_quicksort(A,low,k-1); /* 排序第一个子数组 */
        r_quicksort(A,k+1,high);     /* 排序第二个子数组 */
    }
}
```

- 最坏时间复杂度仍是：$O(n^2)$
- 最坏情况：当随机数发生器第i次随机产生的基准点元素恰恰就是数组中第i大或第i小的元素时造成最坏情况
- 与输入无关
- 最坏情况是微乎其微的
- 输入元素的任何排列顺序，都不可能使算法行为处于最坏的情况
- 期望运行时间是O(nlogn)

7.4 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。

7.4 拉斯维加斯算法概述

- 拉斯维加斯型概率算法有时成功，有时失败
- 当出现失败时，只要在相同的输入实例上再次运行相同的拉斯维加斯概率算法，就又有成功的可能
- 假设 BOOL las_vegas(P(x))是针对问题P的某个实例x的运行代码，则拉斯维加斯执行下面代码：

  while(!las_vegas(P(x))

- 假设p(x)是输入实例x和运行成功的概率之间的关系，如果存在常数δ，使得$p(x) \geq \delta$，就认为该拉斯维加斯算法正确，随着时间增加，可以让算法的失败概率$(1-\delta)^k$任意小

7.4 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);
}
```

7.4 拉斯维加斯(Las Vegas)算法

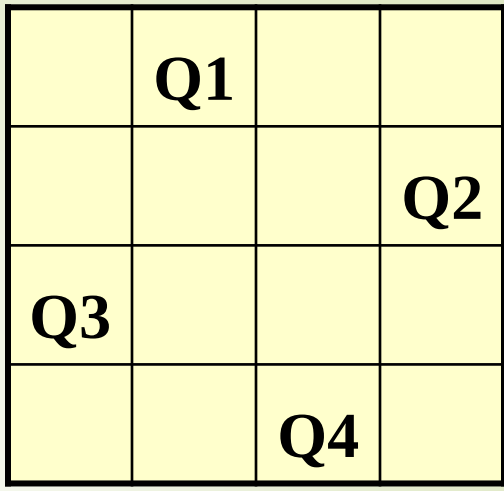
设p(x)是对输入x调用拉斯维加斯算法获得问题的一个解的概率。一个正确的拉斯维加斯算法应该对所有输入x均有p(x)>0。设t(x)是算法obstinate找到具体实例x的一个解所需的平均时间，s(x)和e(x)分别是算法对于具体实例x求解成功或求解失败所需的平均时间，则有：

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

解此方程可得：

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

7.4 N皇后问题



在$n \times n$的棋盘上摆放n个皇后，使任意两个皇后都不能处于同一行、同一列或同一斜线上。

- 确定约束条件：
  - 任意两个皇后不能位于同一行上
  - 任意两个皇后不能位于同一列上
  - 任意两个皇后不能在同一条斜线上

7.4 N皇后问题的拉斯维加斯算法

对于N皇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。

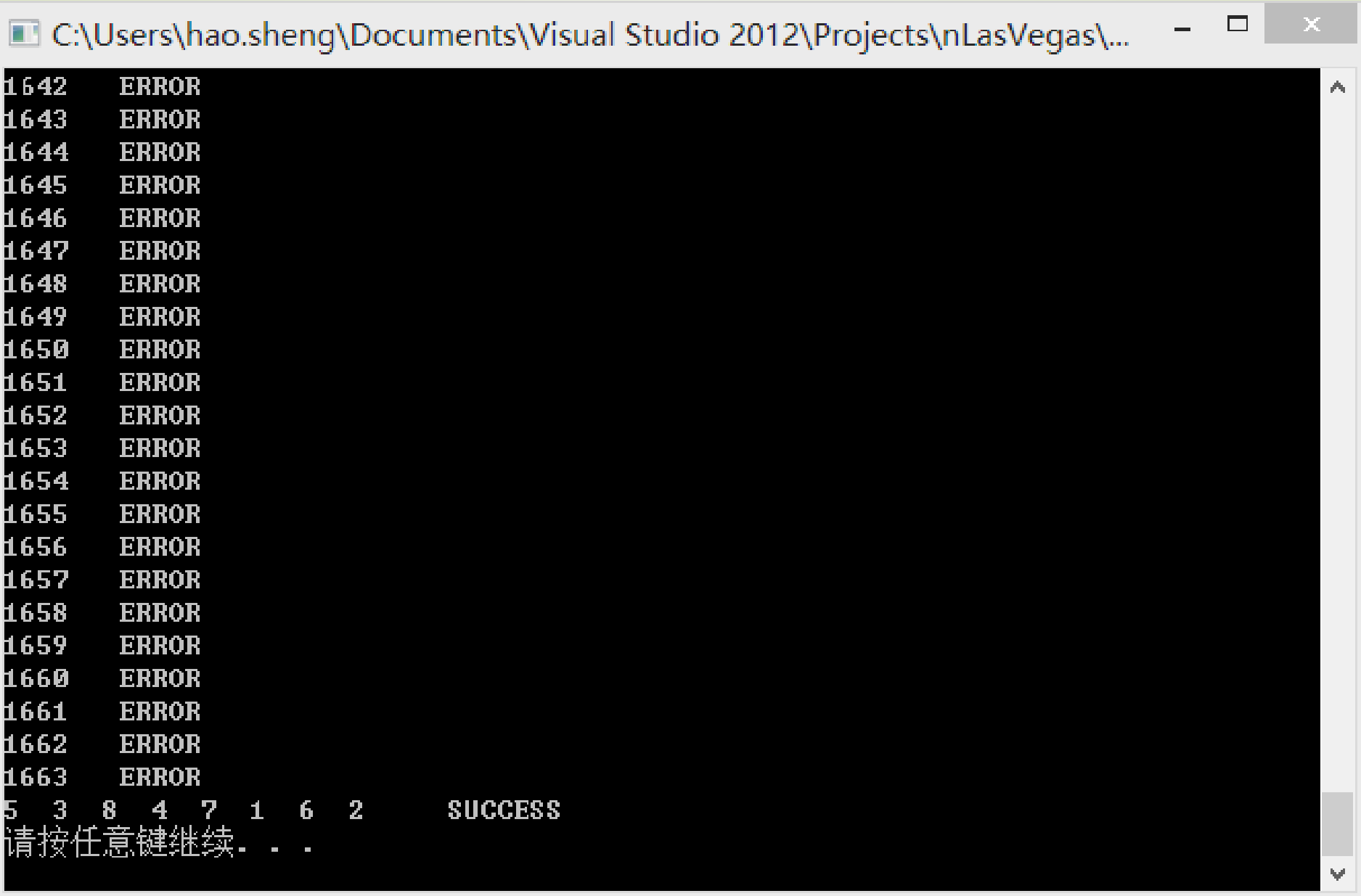
7.4 N皇后问题的拉斯维加斯算法

```
bool Queen::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((abs(k-j) == abs(x[j]-x[k])) || x[j]==x[k]))
            return false;
    return true;
}
```

```
//随机放置n个皇后的拉斯维加斯算法
bool Queen::QueensLv(void)
{
//随机数产生器
RandomNumber rnd;
//下一个皇后的编号
int k = 1;
//在一列中可以放置皇后的个数
int count = 1;

while((k<=n)&&(count>0))
{
count = 0;
for(int i=1; i<=n; i++)
{
x[k] = i;//位置
if(Place(k))
{
y[count++] = i;
}
}
if(count>0)
{
//随机位置
x[k++] = y[rnd.Random(count)];
}
}
//count>0 表示放置成功
return (count>0);
}
```

```
//解n后问题的拉斯维加斯算法
void nQueen(int n)
{
Queen X;
int nCount;
X.n = n;
nCount = 0;

int *p = new int[n+1];
for(int i=0; i<=n; i++)
{
p[i] = 0;
}
X.x = p;
X.y = new int[n+1];

//反复调用随机放置n个皇后的拉斯维加斯算法，直到放置成功
while(!X.QueensLv())
{
nCount++;
cout<<nCount<<"   ERROR \n";
}
for(int i=1; i<=n; i++)
{
cout<<p[i]<<"  ";
}
cout<<"   SUCCESS";
cout<<endl;
delete []p;
}
```

```
class Queen
{
friend void nQueen(int);
private:
bool Place(int k);//测试皇后k置于第x[k]列的合法性
bool QueensLv(void);//随机放置n个皇后拉斯维加斯算法
int n;//皇后个数
int *x,*y;//解向量
};

int main()
{
int n=8;
    cout<<n<<"皇后问题的解为： "<<endl;
    nQueen(n);
system("PAUSE");
    return 0;
}
```


C:\Users\hao.sheng\Documents\Visual Studio 2012\Projects\nLasVegas\...
1642 ERROR
1643 ERROR
1644 ERROR
1645 ERROR
1646 ERROR
1647 ERROR
1648 ERROR
1649 ERROR
1650 ERROR
1651 ERROR
1652 ERROR
1653 ERROR
1654 ERROR
1655 ERROR
1656 ERROR
1657 ERROR
1658 ERROR
1659 ERROR
1660 ERROR
1661 ERROR
1662 ERROR
1663 ERROR
5 3 8 4 7 1 6 2 SUCCESS
请按任意键继续. . .

7.5 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。

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的解究竟是否正确，这是蒙特卡罗算法的缺陷。

7.5 蒙特卡罗算法

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。

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。

7.5.1 主元素问题描述

定义：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。

- 设T[n]是一个含有n个元素的数组，x是数组T的一个元素，如果数组中有一半与x相同，则称x是T[n]的主元素
- 一个数组可以不存在主元素。如果存在，则至多有一个
- 例如，在数组T[7] = {3, 2, 3, 2, 3, 3, 5}中，元素3就是主元素

7.2.2 主元素问题的蒙特卡罗型概率算法求解

(1) 随机找出一个元素

(2) 遍历整个数组，验证这个元素是不是数组的主元素。如果是，运行成功，返回这个解；否则返回第(1)步

```
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;
}
```

7.2.3 主元素问题的蒙特卡罗型概率算法求解的时间复杂度估计

如果主元素在整个数组中的比例是p（显然p>1/2），那么每一次挑出的元素不是主元素的概率小于1/2；

连续运行k次，结果返回false的概率小于2^{-k}；

对于给定的任何e>0，如果要求算法错误概率小于e，则需要运行$\log_2(1/e)$次。每次需要一趟验证。因此，整体的时间复杂度是$O(n \log_2(1/e))$

随机算法的应用

碰到下列情况可以考虑用随机算法：

（1）当确定性算法遇到非常困难的实例，运行时间非常慢，甚至无法处理时，借助随机算法可以避免这种最差情况的发生，从而减小或消除这些实例所带来的影响。

（2）如果一个搜索空间包含大量的可接受的解，可以从这些解空间中随机选择一部分，构成一个小的搜索空间，然后对这个解空间进行搜索，从而可以更快地找到所需要的目标解或者可以接受的近似解。

（3）对一些死锁或对称问题，引进随机算法有助于负载平衡资源，可以避免或中断死锁的发生。

（4）在多样化或者不确定的环境里，有必要引进随机性，才能有效地估计和模仿现实世界的何种不确定性和多样性。

（5）模仿某种假设的情况和得到统计数据，随机算法提供必要的机制去实现。

（6）当需要奇迹出现时。例如在人工智能领域，可以利用随机性来决定确定性算法所无能为力的事情或者提供可控制的噪声等。

END