湖南科技大学课程教案
（章节、专题首页）

授课教师：王志喜　　职称：副教授　　单位：计算机科学与工程学院

课 程 名 称	算法设计与分析
章节、专题	概率方法
教学目标及基 本 要 求	掌握数值概率算法、舍伍德算法、拉斯维加斯算法和蒙特卡洛算法等4种类型概率算法的一般设计方法。
教 学 重 点	舍伍德算法、拉斯维加斯算法和蒙特卡洛算法
教 学 难 点	蒙特卡洛算法
教学内容与时 间 分 配	（1）概率算法的大致类型（0.4课时） （2）数值概率算法（0.3课时） （3）舍伍德算法（0.3课时） （4）拉斯维加斯算法（0.6课时） （5）蒙特卡洛算法（0.9课时） 共计2.5课时。
习　　题	第9.6.1节（基础知识题），第9.6.2节（实验题）。

第9章　概率方法

本章主要介绍数值概率算法、舍伍德算法、拉斯维加斯算法和蒙特卡洛算法等4种类型的概率算法，每种类型的概率算法分别给出2个实例。

- 概率算法的大致类型
- 数值概率算法
- 舍伍德算法
- 拉斯维加斯算法
- 蒙特卡洛算法

9.1 概率算法的大致类型

前面各章所讨论的算法的每一个计算步骤都是确定的，本章所讨论的概率方法允许算法在执行过程中随机选取下一个计算步骤。在许多情况下，当算法在执行过程中面临一个选择时，随机选取通常比最优选择省时。因此概率方法可以在很大程度上降低算法的复杂度。

概率方法的一个基本特征是对所求解问题的同一实例用同一算法求解两次可能产生完全不同的效果，这两次求解所需的时间甚至得到的结果可能会有很大的差别。

使用概率方法设计的算法称为概率算法或随机算法。一般情况下，概率算法大致分为数值概率算法、蒙特卡洛算法、拉斯维加斯算法和舍伍德算法等4种类型。

1. 数值概率算法

常用于数值问题的求解。用数值概率算法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，用数值概率算法往往可以得到相当满意的结果。

2. 蒙特卡洛算法

用于求问题的准确解。对于许多问题来说，近似解毫无意义。例如，一个判定问题的解只能是真或假，不存在任何近似解。用蒙特卡洛算法能够求得问题的一个解，但这个解不能保证是正确的。求得正确解的概率依赖于算法所用的时间。算法用的时间越多，得到正确解的概率就越高。蒙特卡洛算法的主要缺点在于“一般不能有效地判定所得到的解是否正确”。

3. 拉斯维加斯算法

不会得到错误解。一旦用拉斯维加斯算法找到一个解，这个解一定是正确解，但有时会找不到解。与蒙特卡洛算法类似，用拉斯维加斯算法找到正确解的概率随着所用计算时间的增加而提高。对于问题的任一实例，用同一算法反复对该实例求解足够多次，可使求解失效的概率足够小。

4. 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平均情况下的计算复杂性有较大差别时，可在确定性算法中引入随机因素，将它改造成一个舍伍德算法，以消除或减少问题的好坏实例间的差别。舍伍德算法的精髓不是避免算法的最坏情况，而是设法消除这种最坏情形与特定实例之间的关联性。

9.2 数值概率算法

9.2.1 用随机投点法计算π值

设有一半径为 r 的圆及其外切正方形，如图9-1所示。向该正方形内随机投入 n 个点。设落入圆内的点数为 k。由于所投入的点在正方形内均匀分布，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 之比逼近这一概率，从而 $\pi \approx \frac{4k}{n}$。

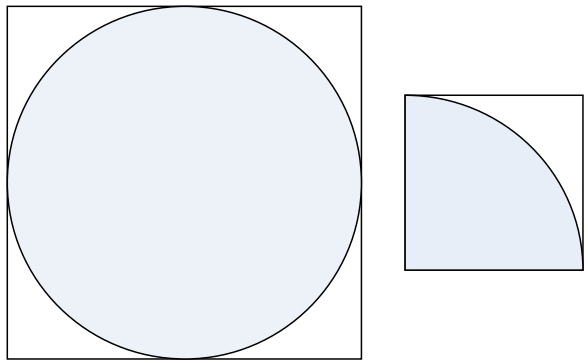

图9-1 用随机投点法计算π值

```
#include "algorithm.h"
```

```
double randf() // 生成一个[0 .. 1]中的随机小数
{   return (double)rand() / RAND_MAX;
}
```

```
double Darts(int n) // 用随机投点法计算π值
{   int k = 0;
    for(int i = 0; i < n; ++i)
    {   auto x = randf(), y = randf(); // [0, 1]内的随机实数
        if((x * x + y * y) <= 1) ++k;
    }
    return (double)4 * k / n;
}
```

```
int main()
{   for(int i = 0; i < 5; ++i)
        cout << Darts(314159 * 5) << " ";
}
```

```
3.13994 3.14044 3.14301 3.1414 3.14215
```

9.2.2 计算定积分

设 $f(x)$ 是 $[0,1]$ 上的连续函数，且 $0 \le f(x) \le 1$。需要计算的积分为 $I = \int_0^1 f(x)dx$，积分 I 等于图9-2中的面积 G。

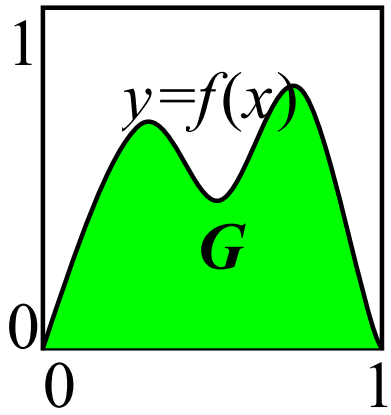

图9-2 计算定积分

在图9-2所示的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 $P(y \le f(x)) = \int_0^1 f(x)dx$。假设向单位正方形内随机投入 n 个点 (x_i, y_i)。如果有 m 个点落入 G 内，则随机点落入 G 内的概率 $I \approx \frac{m}{n}$。

```
#include "algorithm.h"
```

```
double randf() // 生成一个[0 .. 1]中的随机小数
{   return (double)rand() / RAND_MAX;
}
```

```
template<class Func> // 用随机投点法计算定积分
double Darts(int n, Func g)
{   int k = 0;
    for(int i = 0; i < n; ++i)
    {   auto x = randf(), y = randf(); // [0, 1]内的随机实数
        if(y <= g(x)) ++k;
    }
    return (double)k / n;
}
```

```
int main()
{   auto g = [](double x) { return x * x; };
    for(int i = 0; i < 5; ++i)
        cout << Darts(314159 * 3, g) << " ";
}
```

```
0.333364 0.333826 0.333524 0.333849 0.333544
```

9.3 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 $t_A(x)$。若 X_n 是算法 A 的输入规模为 n 的全体实例的集合，则当问题的输入规模为 n 时，算法 A 所需的平均时间为 $\overline{t}_A(n)=\sum_{x\in X_n} t_A(x)/|X_n|$。

显然，可能存在 $x\in X_n$ 使得 $t_A(x)>>\overline{t}_A(n)$。舍伍德算法的基本思想是找到一个概率算法 B，使得当 $x\in X_n$ 时，有 $t_B(x)=\overline{t}_A(n)+s(n)$。当 $s(n)$ 与 $\overline{t}_A(n)$ 相比可以忽略时，舍伍德算法可以获得很好的平均性能。

在舍伍德算法中，随机因素的引入消除了最坏情形与特定实例之间的关联性，消除或减少了好坏实例之间的差别，使得几乎所有实例都可以在平均时间内完成，可以获得很好的平均性能。

9.3.1 快速排序和线性时间选择

在分治方法一章中讨论了快速排序算法和线性时间选择算法，这两个算法的核心都在于选择合适的划分元素。对于这两个算法，如果简单地用数组的首元素或尾元素作为划分元素，则算法的平均性能较好，而在最坏情况下都需要$O(n^2)$计算时间。舍伍德算法则随机选取一个数组元素作为划分元素，以保证算法可以获得很好的平均性能。

1. 划分

下面给出划分算法的随机化版本，保存在文件Partition.h中。

```
template<class T>
int Partition2(T X[], int low, int up)
{   int m = rand() % (up - low ) + low;
    swap(X[up - 1], X[m]); // Partition用尾部元素作为划分元素
    return Partition(X, low, up);
}
```

2. 快速排序

```
#include "Partition.h"
template<class T> // 对范围[low, up)排序
void QuickSort(T X[], int low, int up)
{   if(up - low <= 1) return; // 最多一个元素
    int m = Partition2(X, low, up); // 划分位置
    QuickSort(X, low, m); // 左侧排序
    QuickSort(X, m + 1, up); // 右侧排序
} // 平均耗时O(n*log(n))
```

该舍伍德排序算法可以在$O(n\log n)$平均时间内将n个输入元素排序。

3. 线性时间选择※

```
#include "Partition.h" // 划分程序
template<class T> // 在数组X[n]中找第k(0 <= k < n)小元素，即X[k]
auto Select(T X[], int n, int k)
{   int low = 0, up = n; // 待查找区间为[low, up)
    for(;;)
    {   int m = Partition2(X, low, up); // 划分元素位置
        if(k == m) return X[m]; // m是第m(0 <= m < n)小元素的位置
        else if(k < m) up = m; // 左侧继续寻找
        else low = m + 1; // 右侧继续寻找
    }
}
```

该舍伍德选择算法可以在$O(n)$平均时间内找出n个输入元素中的第k小元素。

9.3.2 随机洗牌算法

有时也会遇到这样的情况，即所给的确定性算法很难直接改造成舍伍德算法。此时可借助于随机预处理技术，不改变原有的确定性算法，仅对输入进行随机洗牌，同样可以收到舍伍德算法的效果。例如，对于确定性的选择算法，可以使用下列洗牌算法将数组中的元素随机排列，然后使用确定性的选择算法求解。

```
template<class T>
void Shuffle(T X[], int n)
{   for(--n; n > 0; --n)
        swap(X[n], X[rand() % n]);
}
```

9.4 拉斯维加斯算法

9.4.1 基本思想

舍伍德算法的优点是其计算时间复杂性对所有实例而言相对均匀。与相应的确定性算法相比，平均时间复杂性没有改进。拉斯维加斯算法则能显著改进算法的有效性。甚至对某些迄今为止找不到有效算法的问题，也能得到满意的结果。

1. 拉斯维加斯算法的调用形式

拉斯维加斯算法的一个显著特征是它的随机性决策有可能导致算法找不到所需的解。因此通常用一个bool型函数表示拉斯维加斯型算法。当算法找到一个解时返回真，否则返回假。

拉斯维加斯算法的典型调用形式为“success = LV(*x*, *y*);”，其中*x*是输入参数，当success的值为真时，*y*返回问题的解。当success为假时，算法未能找到问题的一个解。此时可对同一实例再次独立地调用相同的算法。拉斯维加斯算法的调用形式通常如下所示。

```
function LasVegas(x, y)
{   // 反复调用拉斯维加斯算法LV(x, y), 直到找到一个解y
    while(not LV(x, y)){}
}
```

2. 拉斯维加斯算法找到一个解的平均耗时

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 $p(x)>0$。$t(x)$ 是算法LasVegas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（注意，求解失败后仍需找到一个解），则有

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

解得

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

容易看出，如果LV(x,y)获得成功解的概率较高，则LasVegas所用的时间较短。

```
function LasVegas(x, y)
{   while(not LV(x, y)){}
}
```

9.4.2 旅行商问题TSP※

给定一个边权值非负的无向加权图 G 和一个非负数 t，从 G 中找一个费用不超过 t 的旅行。当前最好的确定性算法的时间复杂度为 $O(n^2 \times 2^n)$。

1. 拉斯维加斯算法

```
#include "algorithm.h"
bool TSP(const Matrix<double> &G, double t, vector<int> &X)
{   // 邻接矩阵, t, 旅行
    int n = G.rows(); // 顶点数
    iota(begin(X), end(X), 0); // 构造自然排列, 耗时O(n)
    random_shuffle(begin(X), end(X)); // 随机选择一个排列, 耗时O(n)
    // 计算回路费用, 耗时O(n)
    double s = 0;
    for(int k = 0; k < n; ++k)
    {   s += G(X[k % n], X[(k + 1) % n]);
        if(s > t) return false; // 不满足要求
    }
    return true; // 满足要求
}
```

【复杂性】执行一次耗时 $O(n)$ 。

2. 反复执行

```
int TSP_m(const Matrix<double> &G, double t, vector<int> &X)
{   int n = G.rows(), m = 0; // 顶点数, 执行次数
    while(not TSP(G, t, X)) ++m; // 反复执行
    return m + 1; // 返回执行次数
}
```

3. 测试程序☼

使用如图9-3所示的网络。

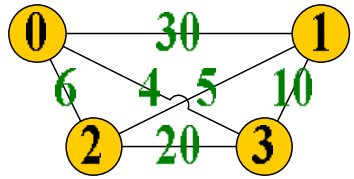

图9-3　一个TSP实例

```
int main()
{   Matrix<double> G = // 邻接矩阵
    {   {0, 30, 6, 4},
        {30, 0, 5, 10},
        {6, 5, 0, 20},
        {4, 10, 20, 0}
    };
    int n = G.rows(); // 顶点数
    vector<int> X(n); // 旅行
    int m = TSP_m(G, 25, X); // 反复执行, 返回执行次数
    cout << X << endl; // 3 0 2 1 <=> 0 2 1 3
    cout << "执行次数: " << m; // 执行次数: 2
}
```

9.4.3 0-1背包问题

给定正数 M 和 t 及正数数组 $\{w_i\}_{i=0}^{n-1}$ 和 $\{v_i\}_{i=0}^{n-1}$，求满足 $\sum\{w_i x_i\}_{i=0}^{n-1} \leq M$ 的0/1数组 $\{x_i\}_{i=0}^{n-1}$，使得 $\sum\{v_i x_i\}_{i=0}^{n-1} \geq t$。回溯算法的时间复杂度为 $O(n^2 \times 2^n)$。

1. 拉斯维加斯算法

```
#include "algorithm.h"
#include "Knap.h"
bool Knap(const vector<double> &V, const vector<double> &W,
                double c, double t, vector<bool> &X)
{   // 效益数组, 重量数组, 背包容量, t, 装包方案
    int n = min(V.size(), W.size()); // 物品数
    double fv = 0, fw = 0; // 最后效益, 最后重量
    for(int i = 0; i < n; ++i) // 随机决定物品i的选取
    {   if(rand() % 2 == 1) X[i] = 1, fv += V[i], fw += W[i];
        else X[i] = 0;
        if(fw > c) return false; // 已经超重
    }
    return fv >= t;
}
```

【复杂性】执行一次耗时 $O(n)$ 。

2. 反复执行

```
bool Knap_m(const vector<double> &V, const vector<double> &W,
            double c, double t, vector<bool> &X)
{   int n = min(V.size(), W.size()), m = 0; // 物品数, 执行次数
    while (not Knap(V, W, c, t, X)) ++m; // 反复执行
    return m + 1; // 返回执行次数
}
```

3. 测试程序

```
int main()
{   int n = 3; // 物品数
    vector<double> V = {120, 60, 100}; // 效益数组
    vector<double> W = {30, 10, 20}; // 重量数组
    double c = 50, t = 220; // 重量数组，背包容量
    vector<bool> X(n); // 装包方案
    int m = Knap_m(V, W, c, t, X); //反复执行，返回执行次数
    PrintSolution(V, W, X); // 显示结果
    cout << "执行次数: " << m << endl;
}
```

```
效益: 120 60 100
重量: 30 10 20
答案: 1 0 1
执行次数: 1
```

9.5 蒙特卡洛算法

在实际应用中经常会遇到一些问题，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解答。蒙特卡洛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，但是通常不能保证一个具体解是否正确。

9.5.1 基本思想

1. 正确率

设p是一个实数，且$0.5<p<1$（高概率）。如果一个蒙特卡洛算法对于问题的任一实例得到正确解的概率不小于p，则称该蒙特卡洛算法是p正确的，p是该蒙特卡洛算法的正确率。

2. 提高正确率的办法

如果一个蒙特卡洛算法对于问题的同一实例不可能给出2个不同的正确解（例如，对同一实例，不可能真和假都正确），则称该蒙特卡洛算法是一致的。

对于一个一致的p正确的蒙特卡洛算法，要提高获得正确解的概率，只要执行该算法若干次，然后选择出现频次最高的解。

例如，对于问题“从奇数长度的0/1数组中选取出现次数较多的元素”，蒙特卡洛算法“从数组中随机选取一个元素”就是一个一致的正确率高于0.5的蒙特卡洛算法。多次重复执行该算法并选取频次最高的解可以提高正确率。

在实际使用中，大多数蒙特卡洛算法经重复调用后正确率提高很快。

3. 偏真算法

设M是求解某个判定问题的蒙特卡洛算法。如果当M返回真时解总是正确的，仅当它返回假时有可能产生错误的解，则称M是一个偏真算法。

对于偏真算法，可以将正确率要求从$0.5<p<1$放宽到$0<p<1$。

多次调用一个偏真的蒙特卡洛算法，只要有一次调用返回真就可以断定相应的解为真。重复调用一个一致的p正确的偏真的蒙特卡洛算法k次，可得到一个一致的$1-(1-p)^k$正确的偏真的蒙特卡洛算法，该算法得到错误解的概率不会高于$(1-p)^k$。例如，只要重复调用一个一致的55%正确的偏真的蒙特卡洛算法4次，就可以将解的正确率从55%提高到95%（$0.45^4=0.041006$），重复调用6次，可以提高到99%（$0.45^6=0.00830377$）。

4. 偏假算法

设M是求解某个判定问题的蒙特卡洛算法。如果当M返回假时解总是正确的，仅当它返回真时有可能产生错误的解，则称M是一个偏假算法。

对于偏假蒙特卡洛算法，可以与偏真算法同样讨论，不再赘述。

9.5.2 团问题

这里讨论判定形式的团问题。

【问题】一个无向图是否存在顶点数不小于k（$k\geq 3$）的团。

1. 蒙特卡洛算法

```
#include "Clique.h"
bool Clique(const Matrix<bool> &G, int k, vector<bool> &X)
{   // 邻接矩阵, k, 当前团
    int n = G.rows(); // 图顶点数
    vector<int> V(n); // 图顶点集
    iota(begin(V), end(V), 0); // 图顶点集初始化为自然排列, 耗时O(n)
    random_shuffle(begin(V), end(V)); // 图顶点集洗牌, 耗时O(n)
    X.assign(n, 0); // 当前团初始化为空集, 耗时O(n)
    int cn = 0; // 团顶点数, 初始值为0
    for(int t : V) // 选取顶点, 共O(n)次
        if(Connected(G, X, t)) // t可行, 耗时O(n)
            X[t] = 1, ++cn; // 选取顶点t
    return cn >= k;
} // 耗时O(n^2)
```

【复杂性】Clique()需要调用$O(n)$次Connected()，而调用Connected()需要耗时$O(n)$，所以Clique()需要耗时$O(n^2)$，是平方时间算法。

2. 重复调用

对于任何给定的$\varepsilon>0$，下列算法重复调用$\log(1/\varepsilon)$次Clique()。

```
bool Clique_m(const Matrix<bool> &G, int k, vector<bool> &X, double e)
{   // 邻接矩阵, k, 当前团, 错误率
    int m = ceil(log2(1 / e)); // Clique()重复执行log(1/e)次
    for(int i = 0; i < m; ++i)
        if(Clique(G, k, X)) return true;
    return false;
} // 耗时O(n^2*log(1/e))
```

【正确率】Clique()是一个偏真的蒙特卡洛算法。因为执行一次Clique()有很大的机会得到最大团，所以Clique()得到正确解的概率$p>0.75$（返回真时一定正确，返回假时超过一半机会正确），从而其错误概率$(1-p)<0.25$。对于给定的$\varepsilon>0$，重复调用$\log(1/\varepsilon)$次Clique()的错误概率$(1-p)^{\log(1/\varepsilon)}<0.25^{\log(1/\varepsilon)}<0.5^{\log(1/\varepsilon)}=2^{\log(\varepsilon)}=\varepsilon$。

3. 测试程序☼

使用如图9-4所示的无向图。

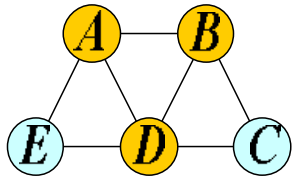

图9-4　一个团问题实例

```
int main()
{   Matrix<bool> G = // 邻接矩阵
    {   {0, 1, 0, 1, 1},
        {1, 0, 1, 1, 0},
        {0, 1, 0, 1, 0},
        {1, 1, 1, 0, 1},
        {1, 0, 0, 1, 0}
    };
    vector<bool> X; // 当前团
    cout << Clique_m(G, 3, X, 0.001) << ": " << X << endl; // 1: 1 0 0 1 1
    cout << Clique_m(G, 4, X, 0.001) << ": " << X << endl; // 0: 0 1 1 1 0
}
```

9.5.3 素数测试※

1. 确定性算法

```
#include "algorithm.h"
template<class Int>
bool Prime(Int n) // n > 2
{   if(n % 2 == 0) return false;
    Int m = sqrt(n); // n的平方根
    for(Int i = 3; i <= m; i += 2)
        if(n % i == 0) return false;
    return true; // 素数
}
```

【复杂性】问题的输入规模为 $m=|\log(n)|$，使用该方法需耗时 $O(n^{0.5})=O(2^{0.5\log(n)})=O(2^{m/2})$，是指数时间算法。

2. 费马小定理

（1）定理描述。若 n 是素数，则对任何 $0<x<n$，都有 $x^{n-1}\equiv 1(\mathrm{mod}\, n)$。

（2）模乘方的计算方法。乘方的计算方法为

$$x^n=\begin{cases}x^{n/2}\times x^{n/2} & n\%2=0\\ x\times x^{n/2}\times x^{n/2} & n\%2=1\end{cases}$$

例如，若用 w 记录结果，z 记录 x^{2^k}（$k\geq 1$），则 x^{10} 的计算过程如下所示。

初始	$w=1$	$z=x$
$n=10, k=1$		$z=x^2$
$n=5, k=2$	$w=x^2$	$z=x^4$
$n=2, k=3$		$z=x^8$
$n=1, k=4$	$w=x^{10}$	

由此，可构造出下列计算模乘方的算法。

$$x^n = \begin{cases} x^{n/2} \times x^{n/2} & n\%2 = 0 \\ x \times x^{n/2} \times x^{n/2} & n\%2 = 1 \end{cases}$$

```
// 计算x ^ n mod m, n>0, m>0, 耗时O(log(n))
template<class Int>
Int pow(Int x, Int n, Int m)
{   Int w = 1, z = x;
    // 结果存入w, z记录x^(2^k), k in [1 .. log(n)]
    for(; n > 0; n /= 2)
    {   if(n & 1) w *= z, w %= m; // 若n是奇数
        z *= z, z %= m;
    }
    return w;
}
```

【复杂性】问题的输入规模为$m=|\log(n)|$，调用pow()需耗时$O(m)$。

3. 蒙特卡洛算法

实际上，对任何大于2的奇数n，都有$0^{n-1}\equiv 1(\mathrm{mod}\ n)$、$1^{n-1}\equiv 1(\mathrm{mod}\ n)$和$(n-1)^{n-1}\equiv 1(\mathrm{mod}\ n)$，所以在测试费马小定理的结论时不需要测试0、1和$n$-1。由此，在模乘方的基础上，可以构造出下列素数测试的蒙特卡洛算法(只考虑大于3的数)。

```
template<class Int> // 一次执行
bool Fermat(Int n) // 奇数n > 3
{   Int x = rand() % (n - 3) + 2; // 不测试0, 1和n-1
    if(pow(x, n - 1, n) != 1) return false; // n一定是合数
    return true; // n很可能是素数
}
```

【复杂性】这是一个偏假的蒙特卡洛算法。问题的输入规模为$m=\lceil \log(n) \rceil$。由Fermat()调用一次pow()可知，调用Fermat()需耗时$O(m)$。

4. 重复调用

对于大于3的奇数n，下列算法重复调用$\log(n)$次Fermat()。

```
template<class Int> // 重复调用
bool Fermat_m(Int n) // 奇数n > 3
{   if(n % 2 == 0 or n % 3 == 0) return false;
    Int m = ceil(log2(n));
    for(Int i = 0; i < m; ++i) // 执行log(n)次测试
        if(Fermat(n) == false) return false; // n一定是合数
    return true; // n高概率为素数
}
```

【复杂性】问题的输入规模为$m=\lfloor\log(n)\rfloor$。该算法调用了$O(m)$次Fermat()，而调用Fermat()需耗时$O(m)$，所以该算法的计算时间为$O(m^2)$，是平方时间算法。

【正确率】Fermat()是一个偏假的3/4正确（返回假一定正确，返回真算一半机会正确）的蒙特卡洛算法，其错误概率不超过1/4。对于给定的奇数n，重复调用$\log(n)$次Fermat()的错误概率不超过$(1/4)^{\log(n)}$。这是一个很保守的估计（因为返回真时正确机会远不止一半），实际效果要好得多。

5. 测试程序

```
int main()
{   cout << "Prime: " << endl;
    cout << 2 << '\t' << 3 << '\t';
    for(int i = 5; i < 10000; i += 2)
        if(Prime(i)) cout << i << '\t';
    cout << endl;
    cout << "Fermat: " << endl;
    cout << 2 << '\t' << 3 << '\t';
    for(int i = 5; i < 10000; i += 2)
        if(Fermat_m(i)) cout << i << '\t';
    cout << endl;
}
```

Prime: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 ……
Fermat: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 ……
2种方法结果相同

【注】如果不准备完整阅读“难问题的近似算法”一章，则可在了解相关概念后选择补充阅读“10.5　带随机因素的近似算法”一节并选作相关练习题。

9.6 练习题

9.6.1 基础知识题

1. 用随机方法改写快速排序程序和基于划分的选择程序。
2. 请写出随机洗牌算法的程序。
3. 请为旅行商问题（是否存在耗费不超过t的旅行）设计一个拉斯维加斯算法。
4. 请为0/1背包问题（是否存在效益和不少于t的装包方式）设计一个拉斯维加斯算法。
5. 请利用费马小定理构造素数测试的概率算法。

9.6.2 实验题

1. 请为Hamilton回路问题（是否有Hamilton回路）设计一个拉斯维加斯算法。
2. 请为皇后问题（是否有解）设计一个拉斯维加斯算法。
3. 请为子集和问题（是否存在和为t的子集）设计一个拉斯维加斯算法。
4. 请为团问题（是否存在顶点数不小于k的团）设计一个拉斯维加斯算法。
5. 请使用概率方法设计一个判断一个给定的整数函数$f(x)$是否恒等于0的蒙特卡洛算法，并使用一种程序设计语言描述该算法，其C/C++的函数原型为“bool iszero(int f(int), int n);”，其中n表示最多进行n次比较后给出答案。
6. 请参阅相关材料写出Miiler-Rabin算法进行素数测试的程序。