第10章　概率方法*

本章主要介绍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
- 蒙特卡罗算法

10.1 概率算法的大致类型

前面各章中所讨论算法的每一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地选择下一个计算步骤。在许多情况下，当算法在执行过程中面临一个选择时，随机选择常比最优选择省时。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。一般情况下，概率算法大致分为数值概率算法、蒙特卡罗算法、拉斯维加斯算法和舍伍德算法这四种类型。

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. 蒙特卡罗算法

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为“是”或“否”，不存在任何近似解。用蒙特卡罗算法能够求得问题的一个解，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。蒙特卡罗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。与蒙特卡罗算法类似，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构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。

10.2 数值概率算法*

10.2.1 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，如图10-1所示。向该正方形随机投掷 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，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 $\pi \approx \frac{4k}{n}$。

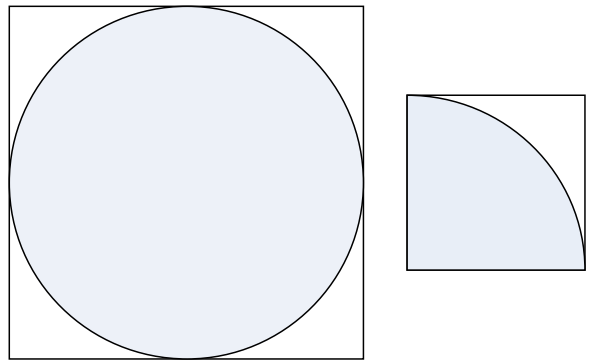


图10-1 用随机投点法计算π值

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

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

```
int main()
{   for(int i = 0; i < 5; ++i)
        cout << Darts(314000) << " ";
}
```

```
3.14113     3.14048     3.13721     3.14122     3.13961
```

10.2.2 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10-2中的面积 G 。

在图10-2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地投入 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 $I \approx \frac{m}{n}$ 。

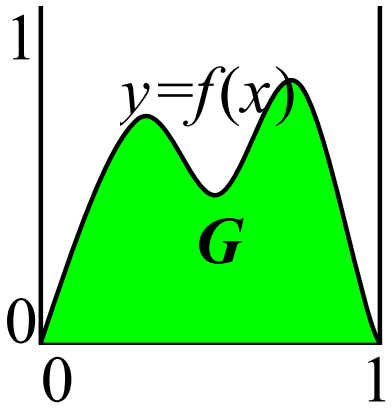


图10-2 计算定积分

```
#include "algorithm.h"
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)
    {   double x = (double)rand() / RAND_MAX; // [0, 1]内的随机实数
        double y = (double)rand() / RAND_MAX;
        if(y <= g(x)) ++k;
    }
    return (double)k / n;
}
int main()
{   auto func = [](double x) { return x * x; };
    for(int i = 0; i < 5; ++i)
        cout << Darts(1000000, func) << " ";
}
```

```
0.33304    0.333338    0.332908    0.333222    0.333037
```

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

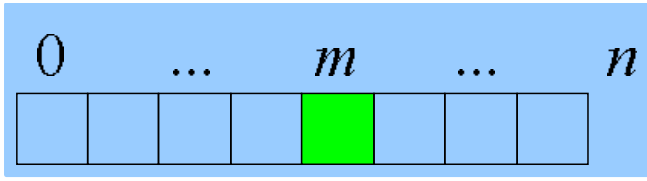
显然，可能存在 $x\in X_n$ 使得 $t_A(x)>>\overline{t}_A(n)$。舍伍德算法的基本思想是希望获得一个概率算法 B，使得当 $x\in X_n$ 时，有 $t_B(x)=\overline{t}_A(n)+s(n)$。当 $s(n)$ 与 $\overline{t}_A(n)$ 相比可忽略时，舍伍德算法可获得很好的平均性能。

10.3.1 线性时间选择算法

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。这两个算法的随机化版本就是舍伍德型概率算法。这两个算法的核心都在于选择合适的划分基准。对于选择问题而言，如果简单地用数组的首元素或尾元素作为划分基准，则算法的平均性能较好，而在最坏情况下需要 $O(n^2)$ 计算时间。舍伍德型选择算法则随机地选择一个数组元素作为划分基准，以保证算法的线性时间平均性能。

1. 划分程序

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



2. 选择程序

在数组X[n]中找第k(0 <= k < n)小元素，即X[k]，其余元素按划分元素为X[k]的划分规则存放。

```
template<class T> // 在数组X[n]中找第k(0 <= k < n)小元素
const T &PartSelect2(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
    }
}
```

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

3. 测试程序*

```
int main()
{   int n = 9;
    double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    cout << to_string(X, n) << endl; // 6 7 9 8 4 3 2 9 6
    cout << PartSelect2(X, n, 4) << endl; // 6
    cout << to_string(X, n) << endl; // 2 3 4 6 6 7 8 9 9
}
```

10.3.2 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型算法。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，对于确定性选择算法，可以用下列洗牌算法将数组中的元素随机排列，然后用确定性选择算法求解。

```
template<class T>
void Shuffle(T X[], int n)
{   for(; n > 0; --n)
    {   int i = rand() % n, j = n - 1;
        if(i != j) swap(X[i], X[j]);
    }
}
```

10.4 拉斯维加斯算法

10.4.1 基本思想

舍伍德型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地改进算法的有效性。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。

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true，否则返回false。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true时，y返回问题的解。当success为false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为

```
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)){}
}
```

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)){}
}
```

10.4.2 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. 算法描述

```
function LV_TSP(G, X, t)
{   // G是一个有n个顶点的无向图, X为解向量
    '随机选择一个以0开头的排列X'; // 耗时O(n)
    '计算相应回路的费用s'; // 耗时O(n)
    if(s > t) return false; // 不满足要求
    return true; // 满足要求
}
```

执行一次耗时 $O(n)$。

2. 算法的C++描述*

① 拉斯维加斯算法的调用形式。保存在文件LasVegas.h中。

```
// LasVegas.h
#pragma once
#include "algorithm.h"
```

```
template<class Func, class ... Args>
int LasVegas(Func &&g, Args && ...arg)
{   int n = 1; //  调用次数
    while(!g(arg...)) ++n;
    return n;
}
```

② 算法描述。算法的C++描述如下。

```
#include "LasVegas.h"
bool LV_TSP(const Matrix<double> &G, double t, int X[])
{   int n = G.Rows();
    iota(X, X + n, 0); // 构造自然排列
    random_shuffle(X + 1, X + n); // 随机选择一个以0开头的排列
    double s = 0;
    for(int k = 0; s <= t && k < n; ++k) // 计算回路费用
    {   int i = X[k], j = X[(k + 1) % n];
        s += G[i][j];
    }
    if(s > t) return false; // 不满足要求
    return true; // 满足要求
}
```

③ 测试程序。使用如图10-3所示的网络。

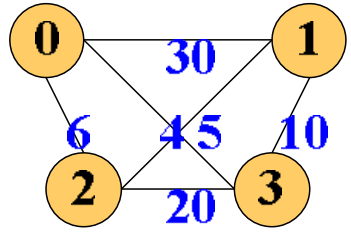


图10-3　一个TSP实例

```
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();
    int X[n]; // 解向量
    int m = LasVegas(LV_TSP, G, 25, (int *)X);
    cout << to_string(X, n) << endl; // 0 3 1 2
    cout << "执行次数: " << m; // 执行次数: 4
}
```

10.4.3 0-1背包问题

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

1. 算法描述

```
#include "KNAP.h"
#include "LasVegas.h"
bool LV_BKNAP(double V[], double W[], int n, double c, double t, bool X[])
{   // 效益数组V, 重量数组W, 背包容量c
    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
    {   X[i] = rand() % 2, fv += V[i] * X[i], fw += W[i] * X[i];
        if(fw > c) return false; // 已经超重
    }
    return fv >= t;
}
```

执行一次耗时 $O(n)$ 。

2. 测试程序*

```
int main()
{   int n = 3; // 物品数量
    double V[n] = {120, 60, 100}, t = 220; // 效益数组, 指定效益值
    double W[n] = {30, 10, 20}, M = 50; // 重量数组, 背包容量
    bool X[n]; // 解向量
    int m = LasVegas(LV_BKNAP, +V, +W, n, M, t, +X);
    PrintSolution(V, W, +X, n);
    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: 7
```

10.5 蒙特卡罗算法

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。蒙特卡罗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，但是通常无法判定一个具体解是否正确。

10.5.1 基本思想

设 p 是一个实数，且 $0.5 < p < 1$。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，且称 $p - 0.5$ 是该算法的优势。

如果对于问题的同一实例，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。

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

在实际使用中，大多数蒙特卡罗算法经重复调用后正确率提高很快。

设 $\mathrm{MC}(x)$ 是解某个判定问题的蒙特卡罗算法。如果当 $\mathrm{MC}(x)$ 返回true时解总是正确的，仅当它返回false时有可能产生错误的解，则称这类蒙特卡罗算法为偏真算法。

多次调用一个偏真蒙特卡罗算法，只要有一次调用返回true就可以断定相应的解为true。

重复调用一个一致的 p 正确的偏真的蒙特卡罗算法 k 次，可得到一个一致的 $1-(1-p)^k$ 正确的偏真的蒙特卡罗算法。例如，只要重复调用一个一致的55%正确的偏真的蒙特卡罗算法4次，就可以将解的正确率从55%提高到95％，重复调用6次，可以提高到99%（0.45^4=0.041006, 0.45^6=0.00830377）。偏真蒙特卡罗算法对正确率 p 的要求可以从 $p>0.5$ 放松到 $p>0$。

如果当 $\mathrm{MC}(x)$ 返回false时解总是正确的，仅当它返回true时有可能产生错误的解，则称这类蒙特卡罗算法为偏假算法。

对于偏假蒙特卡罗算法，可以同样讨论。

10.5.2 主元素问题

设 X 是一个含有 n 个元素的数组，若元素 e 在 X 中出现的次数超过 $n/2$，则称元素 e 是数组 X 的主元素。

1. 蒙特卡罗算法

```
#include "algorithm.h"
template<class T> // 判定是否存在主元素的蒙特卡罗算法
bool Majority(T X[], int n)
{   T &e = X[rand() % n]; // 随机选择数组元素e
    int k = 0; // 计算元素e出现的次数k
    for(int i = 0; i < n; ++i)
        if(X[i] == e) ++k;
    return k > n / 2; // k > n/2 时X含有主元素
}
```

显然，该算法所需的计算时间是 $O(n)$。它是一个偏真的蒙特卡罗算法，得到正确解的概率 $p > 0.5$，所以其错误概率 $(1-p) < 0.5$。

2. 重复调用

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

```
template<class T> // 重复log(1/e)次调用算法Majority
bool Majority(T X[], int n, double e)
{   for(int m = log2(1 / e); m > 0; --m)
        if(Majority(X, n)) return true;
    return false;
}
```

显然，该算法所需的计算时间是 $O(n\log(1/\varepsilon))$。它是一个偏真的蒙特卡罗算法，其错误概率 $(1-p)^{\log(1/\varepsilon)} < 0.5^{\log(1/\varepsilon)} = 2^{\log\varepsilon} = \varepsilon$。

3. 测试程序*

```
int main()
{   double X[] = {2, 1, 1, 3, 1, 4, 1};
    cout << boolalpha << Majority(X, 7, 0.001); // true
}
```

10.5.3 素数测试*

1. 利用定义构造确定性算法

```
#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
}
```

因为问题的输入规模为 $m=\lceil \log n \rceil$，所以使用该方法需要的计算时间为 $O(n^{0.5})=O(2^{0.5\log n})=O(2^{m/2})$，是指数时间算法。

2. 素数的2个性质

【费尔马小定理】若 n 是素数，且 $0<a<n$，则 $a^{n-1}\equiv 1(\mathrm{mod}\, n)$。也可以描述为：若奇数 n 是素数，且 $1<a<n-1$，则 $a^{n-1}\equiv 1(\mathrm{mod}\, n)$。

【二次探测定理】若 n 是素数，且 $0<x<n$，则方程 $x^2\equiv 1(\mathrm{mod}\, n)$ 的解为 $x=1,\ n-1$。

3. 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

$$n = 7, w = 1, z = x$$

$$n = 3, w = x, z = x^2$$

$$n = 1, w = x^3, z = x^4$$

$$n = 0, w = x^7, z = x^8$$

由此，可构造出下列计算模乘方的算法。

```
// 计算x ^ n mod m, n>0, m>0, 时间复杂度为O(log2(n))
template<class Int>
Int pow(Int a, Int n, Int m)
{   Int w = 1, z = a;
    // 结果存入w, z记录a^(2^k), k = 1 .. log2(n)
    for(; n > 0; n /= 2)
    {   if(n & 1) w *= z, w %= m; // 若n是奇数
        z *= z, z %= m;
    }
    return w;
}
```

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

4. 根据费马小定理构造随机算法

```
template<class Int>
bool Fermat(Int n) // n > 2
{   if(n % 2 == 0) return false;
    Int m = log2(n);
    for(Int i = 0; i < m; ++i) //  执行log2(n)次测试
    {   Int a = rand() % (n – 3) + 2; //  不测试0, 1和n-1
        if(pow(a, n - 1, n) != 1) return false; // n一定是合数
    }
    return true; // n高概率为素数
}
```

问题的输入规模为 $m = \lceil \log n \rceil$。调用pow需要 $O(m)$ 的计算时间，Fermat调用了 $O(m)$ 次pow，故Fermat的计算时间为 $O(m^2)$，是平方时间算法。

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

5. 带二次探测的随机算法

该算法是在pow的计算过程中实施二次探测而形成的。

```
template<class Int>
bool Witness(Int a, Int n)
{   // 计算a^(n - 1) mod n, 并在计算过程中实施二次探测
    Int w = 1, z = a, x; // 结果存入w, z记录a^(2^m), m = 1 to log(n-1)
    for(Int p = n - 1; p > 0; p /= 2)
    {   if(p & 1) w *= z, w %= n; // 若p是奇数
        x = z, z *= z, z %= n;
        // 实施二次探测, 结果为假时一定是合数
        if(z == 1 && x != 1 && x != n - 1) return false;
    }
    return w == 1; // 测试费尔马小定理的结论
}
```

6. MiilerRabin算法

对待测试整数 n，重复调用 $\log n$ 次Witness算法。

```
template<class Int>
bool MiilerRabin(Int n) // n > 2
{   if(n % 2 == 0) return false;
    Int m = log2(n);
    for(Int i = 0; i < m; ++i) //  执行log2(n)次测试
    {   Int a = rand() % (n – 3) + 2; //  不测试0, 1和n - 1
        if(!Witness(a, n)) return false; // n一定是合数
    }
    return true; // n高概率为素数
}
```

问题的输入规模为 $m = \lceil \log n \rceil$。调用Witness需要 $O(m)$ 的计算时间，MiilerRabin需要调用 $O(m)$ 次Witness，故MiilerRabin的计算时间为 $O(m^2)$，是平方时间算法。MiilerRabin中的for循环体是一个偏假3/4正确的蒙特卡罗算法。通过多次重复调用错误概率不超过 $(1/4)^{\log n}$。这是一个很保守的估计，实际使用的效果要好得多。

7. 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(i)) cout << i << '\t';
    cout << endl;
    cout << "MiilerRabin: " << endl;
    cout << 2 << '\t' << 3 << '\t';
    for(int i = 5; i < 10000; i += 2)
        if(MiilerRabin(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
……

MiilerRabin:

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
……

3种方法结果相同

10.6 练习题

1、请使用一种程序设计语言描述产生伪随机数的线性同余法方法。

2、请分别为下列问题设计一个拉斯维加斯算法。

Hamilton回路问题（是否有Hamilton回路）；皇后问题（是否有解）；旅行商问题（是否存在耗费不超过t的旅行）；子集和问题（是否存在和为t的子集）；团问题（是否存在大小不小于k的团）；0/1背包问题（是否存在效益和不少于t的装包方式）。

3、用随机方法改写快速排序程序和基于划分的选择程序。

4、请写出随机洗牌算法的程序。

5、请利用Fermat小定理构造素数测试的概率算法。

6、请写出Miiler-Rabin算法进行素数测试的程序。

7、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的蒙特卡罗算法，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。