密码学实验报告 实验二

2019 年 3 月 12 日

1 素性测试算法

1.1 算法原理

如果要判断一个超大整数是否为素数，用传统的素数判定方法是效率极低的，目前还没有简单高效的确定性算法来解决该问题，通常会使用概率算法，常见的素性测试算法有 Miller-Rabin 算法、Fermat 算法和 Solovay-Stassen 算法。

Miller-Rabin 算法利用了素数的如下性质：

设p是大于2的素数，有$p-1=2^k q$，其中$k>0$，q为奇数。设a是整数且$1<a<p-1$，则有下面两个条件之一成立：

(1) a^q模p和1同余，即$a^q \bmod p=1$，或等价地，$a^q \equiv 1(\bmod p)$。

(2) 整数$a^q, a^{2q}, a^{4q}, \cdots, a^{2^{k-1}q}$中存在一个数，模$p$时和$-1$同余。即存在一个$j(1 \le j \le k)$满足$a^{2^{j-1}q} \bmod p=-1$，或$a^{2^{j-1}q} \equiv -1(\bmod p)$。

我们可以利用上述性质做素性测试。给定一个整数n，若n不是素数，通过逆否命题即可判定，否则n既可能是素数，也可能不是素数。当测试次数足够多且总是返回不确定时，我们就能以很大的把握说n是素数。

Fermat 算法利用了 Fermat 小定理：

如果n是一个素数，则对任意整数b，$(b,n)=1$，有

$$b^{n-1} \equiv 1(\bmod n)$$

由此可得：如果有一个整数b，$(b,n)=1$使得

$$b^{n-1} \neq 1(\bmod n)$$

则n是一个合数。

Solovay-Stassen 算法利用了 Euler 判别法则：

设n是奇素数，有同余式

$$b^{\frac{n-1}{2}} \equiv \left(\frac{b}{n}\right)(\bmod n)$$

对任意整数b成立。因此，如果存在整数b，$(b,n)=1$，使得

$$b^{\frac{n-1}{2}} \neq \left(\frac{b}{n}\right) (\mathrm{mod}\ n)$$

则n不是一个素数。

1.2 算法实现的伪代码

Miller-Rabin 算法 对整数n做素性测试

输入： $n > 0$

输出： “合数”当n不是素数或“不确定”当n既可能是素数也可能是合数

```
function miller_rabin(n)
    找出整数k,q，其中k > 0，q是奇数，使n − 1 = 2^k q
    随机选取整数a, 1 < a < n − 1
    if a^q mod n = 1 then
        return “不确定”
    end if
    for j ← 0 to k − 1 do
        if a^(2^j q) mod n = n − 1 then
            return “不确定”
        end if
    end for
    return “合数”
end function
```

Fermat 算法 对整数n做素性测试

输入： $n > 0$

输出： “合数”当n不是素数或“不确定”当n既可能是素数也可能是合数

```
function fermat(n)
    随机选取整数b, 1 < b < n − 1
    求最大公约数d = gcd(b,n)
    if d > 1 then
        return “合数”
    end if
    if b^(n−1) mod n ≠ 1 then
        return “合数”
    end if
    return “不确定”
end function
```

Solovay-Stassen 算法 对整数n做素性测试

输入： $n > 0$

输出： “合数”当n不是素数或“不确定”当n既可能是素数也可能是合数

function $solovay\_stassen(n)$
随机选取整数$b, 1 < b < n-1$
计算$r = b^{\frac{(n-1)}{2}} (\mathrm{mod}\ n)$
if $r \neq 1$ **and** $r \neq n-1$ **then**
return “合数”
end if
计算 Jacobi 符号$s = \left(\frac{b}{n}\right)$
if $r \neq s$ **then**
return “合数”
end if
return “不确定”
end function

1.3 测试样例及运行结果

为了检验算法在超大素数上的效果，我们选取了梅森素数$M_p = 2^p - 1$来比较不同算法的效率，表中选取了第 14、15、16 个梅森素数M_{607}、M_{1279}、M_{2203}

表 1 三种算法时间效率比较（单位：毫秒）

待测素数	Miller-Rabin 法	Fermat 法	Solovay-Stassen 法
M_{607}	26.03	33.59	40.72
M_{1279}	170.06	215.86	235.66
M_{2203}	611.22	949.80	987.42

最后，为了检验素数筛选算法的正确性，我们使用两个相邻的梅森数M_{2281}和M_{2282}，其中M_{2281}是梅森素数，而M_{2282}是合数。

表 2 三种算法实验结果

待测素数	Miller-Rabin 法	Fermat 法	Solovay-Stassen 法
M_{2281}	True	True	True
M_{2282}	False	False	False

2 有限域的四则运算

2.1 有限域运算原理

一个元素个数有限的域称为有限域或伽罗华域。有限域中元素的个数为一个素数或一个素数的幂，记为$GF(p^n)$。有限域中运算满足交换律，结合律和分配律，加法的单位元是 0，乘法的单位元是 1，每个非零元素都有唯一的乘法逆元，我们将重点研究形如$GF(2^n)$的有限域，可以表达为模 2 系数且次数低于 n 的多项

式，可以用 n 次不可约多项式去模约以降次。此类多项式构成一个有限域，且非零元也总有唯一的逆元存在。由于系数仅为 0 或 1，可以用一个比特串来表示任何多项式，加法为比特串的逐位异或，乘法为移位和异或，可以用扩展欧几里得算法求乘法逆元。

为了利用 Python 面向对象编程的特性，我们用一个类来实现有限域，将该类的运算符重载，加入自定义的运算操作，可以和普通数字的运算一样来操作有限域中的元素，由于 Python 为弱类型语言，甚至可以直接使用原来的扩展欧几里得算法来求有限域元素的乘法逆元。

2.2 有限域运算实现的伪代码

有限域加法 计算两元素相加的和

输入： 比特串a, b
输出： $s = a + b$
function $add(a, b)$
    return $a \oplus b$
end function

有限域减法 计算两元素相减的差

输入： 比特串a, b
输出： $s = a - b$
function $sub(a, b)$
    return $a \oplus b$
end function

有限域乘法 计算两元素相乘的积

输入： 比特串a, b，不可约多项式m
输出： $s = a * b(\mathrm{mod}\ m)$
function $mul(a, b)$
    for $i \leftarrow 0\ to\ n$ **do**
        $temp[i] \leftarrow a^i(\mathrm{mod}\ m)$
    end for
    for $i \leftarrow 0\ to\ n$ **do**
        $multi \leftarrow multi \oplus (b[i] * temp[i])$
    end for
    return $multi$
end function

有限域除法 计算两元素相除的商

输入： 比特串a, b，不可约多项式m

输出： $s = \frac{a}{b}$

function $revElem(a, m)$
    $s_0 \leftarrow 1, t_0 \leftarrow 0$
    $s_1 \leftarrow 0, t_1 \leftarrow 1$
    $q \leftarrow \lfloor a/m \rfloor$
    $r_0 \leftarrow m, r_1 \leftarrow a \ \boldsymbol{mod} \ m$
    while $r_1 \neq 0$ **do**
        $x_0 \leftarrow x_1, x_1 \leftarrow x_0 - q * x_1$
        $y_0 \leftarrow y_1, y_1 \leftarrow y_0 - q * y_1$
        $q \leftarrow \lfloor r_0/r_1 \rfloor$
        $r_0 \leftarrow r_1, r_1 \leftarrow r_0 \ \boldsymbol{mod} \ r_1$
    end while
    return $x \leftarrow x_1$
end function
function $div(a, b)$
    return $mul(revElem(a, m), b)$
end function

2.3 测试样例及运行结果

我们实现了有限域内的加法、减法、乘法、整除、取模、除法、最大公因子和扩展欧几里得算法，支持$GF(2^1), GF(2^2), GF(2^3), GF(2^4), GF(2^8), GF(2^{16})$的相应运算，且只要给出一个$GF(2^n)$内的不可约多项式，便可以迅速地扩展到该域上的运算。以下选取$GF(2^8)$内的两个元素，对其进行实验验证：

表 3 有限域运算的实验验证

数据	运算结果
a	01110001
b	00001101
$a+b$	01111100
$a-b$	01111100
$a*b$	00001011
$\lfloor a/b \rfloor$	00001010
$a\%b$	00000011
a/b	10010010
a/a	00000001
$gcd(a,b)$	00000001
$exgcd(a,b)$	(00000001, 00000100, 00101001)

3 本原根生成算法

3.1 算法原理

由欧拉定理，若$(a,m)=1$，那么

$$a^{\varphi(m)} \equiv 1(\mathrm{mod}\ m)$$

满足

$$a^{d} \equiv 1(\mathrm{mod}\ m)$$

的最小正整数d_0称为a模m的阶，记为$\delta_m(a)$。

设m是正整数，a是整数，若$\delta_m(a)=\varphi(m)$，则称a为模m的一个本原根。

由定理知，m有本原根存在的充要条件是$m=2,4,p^l,2p^l$，其中p为奇素数。

求本原根首先要知道欧拉函数，由于我们已知存在本原根的数的一般形式，便可以快速计算出欧拉函数，然后通过遍历每个数以及它的幂，就能通过逐个验证得到一个数的所有本原根。

3.2 本原根生成算法实现的伪代码

本原根生成算法 计算在模m下的所有本原根

输入： m
输出： PR：m的所有本原根

```
function eular(n)
    m ← n
    if m mod 2 = 0 then
        m ← m/2
    end if
    for p ← 3 to m do
        if m mod p = 0 then
            while m mod p = 0 do
                m ← m/p
            end while
            break
        end if
    end for
    if m ≠ 1 then
        return None
    end if
    if n mod 2 = 0 then
        phi ← n/2/p * (p − 1)
```

```
        else
            phi ← n/p * (p − 1)
        end if
        return phi
    end function
    function prime_root(m)
        if m = 2 then
            return 1
        end if
        if m = 4 then
            return 3
        end if
        phi = eular(m)
        if phi = None then
            return None
        end if
        for i ← 2 to m − 1 do
            if i^j mod n ≠ 1 for ∀j = 2, ⋯ , phi − 1 then
                PR[] ← i
            end if
        end for
        return PR
    end function
```

3.3 算法优化原理

上述算法计算a模m的阶$\delta_m(a)$时使用的方法是依次计算$a^1, a^2, \cdots (\mathrm{mod}\ m)$，直到第一次出现$a^d \equiv 1(\mathrm{mod}\ m)$，则$\delta_m(a) = d$。这种方法效率较低，且浪费时间。由 Euler 定理可知$a^{\varphi(m)} \equiv 1(\mathrm{mod}\ m)$，再根据以下定理：

设$(a, m) = 1$，$d_0 = \delta_m(a)$，则$a^k \equiv 1(\mathrm{mod}\ m)$当且仅当$d_0 | k$。

可知$\delta_m(a)$一定是$\varphi(m)$的因子，将$\varphi(m)$分解，用$\varphi(m)$的每一个因子d，验证$a^d \equiv 1(\mathrm{mod}\ m)$是否成立，选择满足这个条件的最小因子$d$即可。

3.4 优化后算法实现的伪代码

本原根生成算法 计算在模m下的所有本原根

输入：m

输出：PR：m的所有本原根

```
function naive_fact(n)
    for i ← 2 to n do
        if n mod i = 0 then
```

```
            factors[] ← i
            while n mod i = 0 do
                n ← n/i
            end while
        end if
        if n = 1 then
            break
        end if
    end while
    return factors
end function
function advance_pr(m)
    if m = 2 then
        return 1
    end if
    if m = 4 then
        return 3
    end if
    phi = eular(m)
    if phi = None then
        return None
    end if
    factors ← naive_fact(phi)
    for i ← 2 to m − 1 do
        if i^(phi/j) mod n ≠ 1 for ∀j in factors then
            PR[] ← i
        end if
    end for
    return PR
end function
```

3.5 两种本原根生成算法的对比分析

对于原来的本原根生成算法，需要遍历n个数，且每个数的检验最差情况需要循环$\varphi(n)$次，考虑最差情况$\varphi(n) = n - 1$，时间复杂度为$O(n^2)$。

对于优化后的算法，同样要遍历n个数，但是每次的检验需要循环的次数仅仅是$\varphi(n)$的因子个数，考虑最差情况$\varphi(n) = 2 * 3 * 5 * \cdots$，大约为$\log n$次，时间复杂度约为$O(n * \log n)$。

对于较大的数字来说，显然经过优化后的算法性能会有明显提升。

3.6 测试样例及运行结果

我们选取了三个数字对两种算法效率进行比较，运行结果如下表所示。

表 4 两种本原根生成算法的运行时间（单位：毫秒）

输入数据	常规算法	优化后算法
81	5.74	0.23
361	220.65	1.80
3125	18867.65	17.82

最后再使用两种算法计算25的所有本原根，结果均为

$$[2, 3, 5, 8, 10, 12, 13, 15, 17, 20, 22, 23]$$

证明算法正确。

经过进一步的思考，我们发现该算法还有改进的余地，例如更换质因数分解的算法，如果使用 Pollard rho 因数分解方法，要比传统的遍历方法更快，可以使程序更高效地计算出本原根。

4 本原多项式生成算法

4.1 算法基本原理

设$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$是唯一分解整环$D$上的多项式，如果$\gcd(a_0, a_1, \cdots, a_n) = 1$，则称$f(x)$为$D$上的一个本原多项式。

在有限域$GF(2^n)$上，n阶本原多项式$f(x)$的充要条件是

(1) $f(x)$为不可约多项式

(2) $f(x)$整除$x^m + 1, m = 2^n - 1$

(3) $f(x)$不能整除$x^k + 1, k = 1,2, \cdots, 2^n - 2$

所以我们可以遍历所有n次多项式，依次对三个条件进行判定，若符合上述条件则为我们要求的本原多项式。

4.2 算法实现的伪代码

本原多项式生成算法 求所有的n次本原多项式

输入： 多项式次数n
输出： 所有的n次本原多项式P
function $find\_primePoly(n)$
    $r \leftarrow x^{2^n-1} + 1$
    for $p \leftarrow x^n \ to \ x^{n+1} - 1$ **do**

```
        for k ← 2 to p − 1 do
            if p mod k = 0 then
                验证下一个多项式p + 1
            end if
        end for
        if r mod p = 0 then
            if x^k + 1 mod p ≠ 0 for ∀k = 1, ⋯, 2^n − 2 then
                P[] ← p
            end if
        end if
    end for
    return P
end function
```

4.3 测试样例及运行结果

我们尝试让程序生成所有的 3 次本原多项式和 6 次本原多项式，程序均在 1 秒内给出了相应结果。

所有的 3 次本原多项式为：

$$x^3 + x + 1, x^3 + x^2 + 1$$

所有的 6 次本原多项式为：

$$x^6 + x + 1$$

$$x^6 + x^4 + x^3 + x + 1$$

$$x^6 + x^5 + 1$$

$$x^6 + x^5 + x^2 + x + 1$$

$$x^6 + x^5 + x^3 + x^2 + 1$$

$$x^6 + x^5 + x^4 + x + 1$$

根据进一步的研究和查阅文献，我们发现寻找有限域上所有本原多项式的时间复杂度均为指数级，即需要遍历$GF(2^n)$上的全部2^{2^n}个多项式来依次验证，在今后的工作中，我们还可以通过优化检验起点和分解质因式算法等进一步提高算法的效率。

5 总结

在本次实验中，我们依次实现了素性测试算法、有限域上的四则运算、本原根生成算法和本原多项式生成算法。在素性测试算法中，进一步巩固了数论中的知识，加深了对素数性质的理解，根据三条不同定理的逆否命题实现了三种不同的素性测试方法，经过对算法的测试分析，发现 Miller-Rabin 算法更为高效。在有限域上的四则运算中，我们利用了 Python 面向对象的特性，编写了有限域$GF(2^n)$的类，重载了运算符，只要给出一个不可约多项式，该算法便可以迅速地扩展到任意2^n阶的有限域四则运算。在本原根生成算法中，我们首先通过依次遍历的方法来验证本原根，发现算法的效率较低，进一步引入了 Euler 定理，通过分解$\varphi(n)$得到质因子再依次验证，可以优化算法的性能，经过实验分析，发现优化后算法的运行速度比原算法提高了 1000 倍。最后的本原多项式生成算法中，我们利用有限域$GF(2^n)$上本原多项式的充要条件，对所有的n次多项式进行逐个验证，可以得到正确的结果。