《现代密码学》实验报告

实验名称：有限域的实现	**实验时间：2022.10.9**
学生姓名：伍建霖	学号：20337251
学生班级：20网安	成绩评定：

一、实验目的

通过实现有限域，理解有限域的运算方式。

二、实验内容

有限域的实现

- 实验要求

1. 构造有限域GF(2^131)：不可约多项式为f(x) = $x^{131}+x^{13}+x^{2}+x+1$；

2. 给出GF(2^131)上的加法、乘法、平方以及两种求逆运算的子程序；

- 2.1 使用任意一个或两个不是0和1的元素，每种运算执行50000次，并统计以上运算时间的：下四分位、中位数、均值、上四分位（统计运行时间数据可以不使用C++，但子有限域运算需要使用C++实现）；
- 2.2 两种求逆运算分别基于扩展欧几里得算法和费马小定理

3. 将自己的学号转为二进制字符串，看作有限域中的元素a，计算a的逆元素a^{-1}以及$a^{21214928}$

注：二进制的学号每一位数字用四位二进制比特串表示，9位学号表示为36位二进制串（高次项在后低次项在前）。如123456789->000100100011010001010110011110001001

三、实验原理

加法

由于有限域中多项式的每一项系数都为0或1，即异或操作。

模运算

由于乘法之后的结果可能大于2的131次方，故需要实现mod运算。题目给出了不可约多项式，可得x^{131}和$x^{13}+x^2+x+1$同余，可借助此式降幂：$x^n=x^{131}*x^{n-131}=(x^{13}+x^2+x+1)*x^{n-131}$

乘法

两个131bit的数相乘，结果最长为262bit。运算过程和二进制乘法类似，一个不动，另一个移位，然后加上去，最后mod一下。

平方

自己乘自己。

基于扩展欧几里得算法的求逆

使用扩展欧几里得算法求逆，记任意多项式为g，不可约多项式为f，在GF(2^131)中，对于不可约多项式，任意一个多项式和它的最大公约式必定为1，则有：$gx+fy=gcd(g,f)=1$ 我们可以列出这样的一组式子

$$gx1+fy1=k1, gx2+fy2=k2$$

在初始条件下，我们令$x1=1, y1=0, x2=0, y2=1, k1=g, k2=f$ 在每一轮比较k2和k2的最大幂次，计算两式的最大幂次差为e，幂次大的式子减去幂次小的式子乘上x^e的结果，更新幂次大的式子的值。每一轮计算的时候都记录下x1和x2的值，最终能够得到k的幂次为0的式子，也就是等号右边为1，那么这时候我们就能够得到g的逆即x的值了。

基于费马小定理的求逆

由费马小定理$x^{2^m} \equiv x \ mod\ 2^m$ 可推出$x^{-1} \equiv \ x^{2^m-2}$，即计算$x^{2^m-2}$，用快速幂算法可实现指数运算。

四、实验步骤

```
#include <bitset>
#include <vector>
#include <iostream>
#include <chrono>
#include <numeric>
#include <algorithm>
using namespace std;
typedef chrono::steady_clock STEADY_CLOCK;
#define P 131
#define RSTR "10000000000111"

class gf
{
private:
    bitset<P> value;

public:
    void set(int);
    void set(bitset<P>);
    bitset<P> returnValue();
    bitset<P> add(gf);
    bitset<P> mul(bitset<P>);
    bitset<P> squ();
    bitset<P> in1();
    bitset<P> in2();
    bitset<P> mod(bitset<P * 2>, bitset<P>);
```

```
    bitset<P> exp(bitset<P>);
    int maxPower(bitset<P + 1>);
};

void printTime(vector<chrono::duration<int, nano>>);

int main()
{
    /*
    // int a = 9; // 1001    // int b = 1; // 0001  // cout << (a ^ b);
    // int t = 001111; //以0开头表示八进制 // cout << t;
    // bitset<P> a("11111110");
    // gf b, c;
    // b.set(a);
    // b.set(b.mul(b.returnValue()));
    // b.set(b.squ());
    // b.set(2);
    // c.set(2);
    // b.set(b.in1());
    // b.set(b.mul(c.returnValue()));
    // b.set(2);
    // cout << b.in2() << '\n';
    // cout << b.returnValue(); // a = 10, cout 01
    */
    gf a, b;
    a.set(20000000);
    b.set(30000000);
    vector<chrono::duration<int, nano>> timestorer;

    for (int i = 0; i < 50000; i++)
    {
        STEADY_CLOCK::time_point t1 = STEADY_CLOCK::now();
        a.set(a.add(b));
        STEADY_CLOCK::time_point t2 = STEADY_CLOCK::now();
        chrono::duration<int, nano> dTimeSpan = chrono::duration<int, nano>(t2 -
t1);
        timestorer.push_back(dTimeSpan);
    }
    cout << "加法运算表现如下: " << '\n';
    printTime(timestorer);
    timestorer.clear();

    for (int i = 0; i < 50000; i++)
    {
        STEADY_CLOCK::time_point t1 = STEADY_CLOCK::now();
        a.set(a.mul(b.returnValue()));
        STEADY_CLOCK::time_point t2 = STEADY_CLOCK::now();
        chrono::duration<int, nano> dTimeSpan = chrono::duration<int, nano>(t2 -
t1);
        timestorer.push_back(dTimeSpan);
    }
    cout << "乘法运算表现如下: " << '\n';
    printTime(timestorer);
    timestorer.clear();
```

```
    for (int i = 0; i < 50000; i++)
    {
        STEADY_CLOCK::time_point t1 = STEADY_CLOCK::now();
        a.set(a.squ());
        STEADY_CLOCK::time_point t2 = STEADY_CLOCK::now();
        chrono::duration<int, nano> dTimeSpan = chrono::duration<int, nano>(t2 -
t1);
        timestorer.push_back(dTimeSpan);
    }
    cout << "平方运算表现如下: " << '\n';
    printTime(timestorer);
    timestorer.clear();

    for (int i = 0; i < 50000; i++)
    {
        STEADY_CLOCK::time_point t1 = STEADY_CLOCK::now();
        a.in1();
        STEADY_CLOCK::time_point t2 = STEADY_CLOCK::now();
        chrono::duration<int, nano> dTimeSpan = chrono::duration<int, nano>(t2 -
t1);
        timestorer.push_back(dTimeSpan);
    }
    cout << "基于扩展欧几里得算法的求逆运算表现如下: " << '\n';
    printTime(timestorer);
    timestorer.clear();

    for (int i = 0; i < 50000; i++)
    {

        STEADY_CLOCK::time_point t1 = STEADY_CLOCK::now();
        b.in2();
        STEADY_CLOCK::time_point t2 = STEADY_CLOCK::now();
        chrono::duration<int, nano> dTimeSpan = chrono::duration<int, nano>(t2 -
t1);
        timestorer.push_back(dTimeSpan);
    }
    cout << "基于费马小定理的求逆运算表现如下: " << '\n';
    printTime(timestorer);
    timestorer.clear();

    // 20337251
    bitset<P> xh("00100000001100110111001001010001");
    bitset<P> ex(21214928);
    a.set(xh);
    cout << "学号为" << xh << '\n';
    cout << "逆为" << a.in1() << '\n';
    cout << "学号^21214928为" << a.exp(ex);
}

void printTime(vector<chrono::duration<int, nano>> timestorer)
{
    chrono::duration<int, nano> sum(0);
    sort(timestorer.begin(), timestorer.end());
    for (int i = 0; i < 50000; i++)
    {
```

```
        sum += timestorer[i];
    }
    cout << "下四分位: " << timestorer[12500].count() << "ns" << '\n';
    cout << "中位数: " << timestorer[25000].count() << "ns" << '\n';
    cout << "均值: " << (sum / 50000).count() << "ns" << '\n';
    cout << "上四分位: " << timestorer[37500].count() << "ns" << '\n';
}

// 模函数
bitset<P> gf::mod(bitset<P * 2> a, bitset<P> r)
{
    bitset<P> ret(0);
    bitset<P * 2> temp;
    bitset<P * 2> _extend(r.to_ulong());
    for (int i = P * 2 - 1; i >= P; i--)
    {
        if (a[i])
        {
            temp = _extend << (i - P);
            a ^= temp;
            a[i] = 0;
        }
    }
    for (int i = 0; i < P; i++)
    {
        ret[i] = a[i];
    }
    return ret;
}

/*
bitset有不少成员函数,
其中有一个<<=左移操作, 可以把0000001变成0010000, 可能用于实现乘法?
*/

// int set value
void gf::set(const int n)
{
    this->value = n;
}

// bitset<P> set value
void gf::set(bitset<P> n)
{
    this->value = n;
}

// return value
bitset<P> gf::returnValue()
{
    return this->value;
}

bitset<P> gf::add(gf a)
{
```

```
    return this->value ^ a.returnValue();
}

bitset<P> gf::mul(bitset<P> a)
{
    bitset<P * 2> _ret(0);
    bitset<P> ret(0);
    bitset<P> r(RSTR);
    bitset<P * 2> _extend(this->value.to_string());
    for (int i = 0; i < P; i++)
    {
        if (a[i])
        {
            _ret ^= (_extend << i);
        }
    }
    ret = mod(_ret, r);
    return ret;
}

bitset<P> gf::squ()
{
    return this->mul(this->value);
}

int gf::maxPower(bitset<P + 1> a)
{
    for (int i = P; i > -1; i--)
    {
        if (a[i])
        {
            return i;
        }
    }
    return 0;
}

// from Extend Euclidean...
bitset<P> gf::in1()
{
    bitset<P + 1> poly(RSTR);
    poly[131] = 1;
    bitset<P + 1> k1(this->value.to_string());
    bitset<P + 1> k2(poly.to_string());
    bitset<P + 1> ktemp;
    bitset<P> x1(0);
    bitset<P> x2(0);
    bitset<P> temp;
    x1[0] = 1;
    int difPower;
    while (maxPower(k1))
    {
        difPower = maxPower(k1) - maxPower(k2);
        if (difPower < 0)
        {
```

```
            difPower = -difPower;
            ktemp = k1;
            k1 = k2;
            k2 = ktemp;
            temp = x1;
            x1 = x2;
            x2 = temp;
        }
        k1 ^= (k2 << difPower);
        x1 ^= (x2 << difPower);
    }
    bitset<P> r(RSTR);
    return x1;
}

// from fermat...
bitset<P> gf::in2()
{
    // a = 2^P - 2
    bitset<P>
a("11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111110");
    return this->exp(a);
}

// 快速幂算法算指数
bitset<P> gf::exp(bitset<P> e)
{
    gf ret;
    ret.set(1);
    if (!e.any())
    {
        return ret.returnValue();
    }
    while (e.any())
    {
        if (e[0] & 1)
        {
            ret.set(ret.mul(this->value));
        }
        this->set(this->squ());
        e >>= 1;
    }
    return ret.returnValue();
}
```

五、实验结果

```
加法运算表现如下:
下四分位: 100ns
中位数: 100ns
均值: 91ns
上四分位: 100ns
乘法运算表现如下:
下四分位: 7000ns
中位数: 7200ns
均值: 7532ns
上四分位: 7400ns
平方运算表现如下:
下四分位: 9800ns
中位数: 10200ns
均值: 10382ns
上四分位: 10600ns
基于扩展欧几里得算法的求逆运算表现如下:
下四分位: 169600ns
中位数: 173100ns
均值: 4385ns
上四分位: 177900ns
基于费马小定理的求逆运算表现如下:
下四分位: 2807900ns
中位数: 2856100ns
均值: 26634ns
上四分位: 2895100ns
学号为00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000100000001100110111001001010001
逆为10111100000110111001100011110010110001011101110100011011000001001110010110010010011011110101010100010001101110100110001011011110001
学号^21214928为000010110101101001011000001001000001000000000001111000010000011011101101001111010001111011111000101100111000011100001110011101001100
(base) PS D:\CodeField\Cryptography\lab>
```

```
加法运算表现如下:
下四分位: 0ns
中位数: 0ns
均值: 40ns
上四分位: 100ns
乘法运算表现如下:
下四分位: 7400ns
中位数: 7500ns
均值: 7740ns
上四分位: 7700ns
平方运算表现如下:
下四分位: 10000ns
中位数: 10300ns
均值: 10577ns
上四分位: 10700ns
基于扩展欧几里得算法的求逆运算表现如下:
下四分位: 169500ns
中位数: 173900ns
均值: 4509ns
上四分位: 178800ns
基于费马小定理的求逆运算表现如下:
下四分位: 2801300ns
中位数: 2892500ns
均值: -39834ns
上四分位: 3038500ns
学号为00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000100000001100110111001001010001
逆为10111100000110111001100011110010110001011101110100011011000001001110010110010010011011110101010100010001101110100110001011011110001
学号^21214928为000010110101101001011000001001000001000000000001111000010000011011101101001111010001111011111000101100111000011100001110011101001100
```

```
加法运算表现如下:
下四分位: 100ns
中位数: 100ns
均值: 82ns
上四分位: 100ns
乘法运算表现如下:
下四分位: 6700ns
中位数: 6900ns
均值: 7147ns
上四分位: 7200ns
平方运算表现如下:
下四分位: 9400ns
中位数: 9800ns
均值: 9946ns
上四分位: 10200ns
基于扩展欧几里得算法的求逆运算表现如下:
下四分位: 173700ns
中位数: 177600ns
均值: 7480ns
上四分位: 182000ns
基于费马小定理的求逆运算表现如下:
下四分位: 2697800ns
中位数: 2786000ns
均值: 37406ns
上四分位: 2925700ns
```

六、实验总结

这次实验我实现了2的131次方大小的有限域的加法，乘法，平方，基于扩展欧几里得的求逆运算，基于费马小定理的求逆运算。通过实现这些运算，我对如何实现这些运算的细节有了更深的了解。除了学习到了有关实验内容的知识，我还学到了更多关于c++的知识，比如bitset的用法，比如如何在cpp中计算时间等等。