《现代密码学》实验报告

实验名称：有限域运算	**实验时间：2024年10月23日**
学生姓名：黄集瑞	学号：22336090
学生班级：22保密管理	费马定理ID:1354 欧几里得ID:1142 麻烦助教忽略我的其他提交成果,已有事先告知不采用1000ms数量级以下的结果

一、实验目的

> 本次实验通过实现有限域$F_{2^{131}}$上的加法、乘法、平方以及求逆运算有利于我们理解有限域的基本操作，并且能够提高我们对多项式计算的理解以及运用。除此之外，本实验采用二进制的输入以及输出流也让我们学会了如何处理二进制以及十六进制的数据，提高我们日后的编码能力。

二、实验内容

> - 用C++实现有限域上的各种运算操作
> - 输入：`uint32_t`类型用来表示需要进行的运算数量，`uint8_t`类型来表示选择的操作，然后用 `uint64_t[2][3]`来表示进行运算的两个有限域元素（在实验网站上写的是 `uint64_t[3][2]`感觉这样的表达有误，或者理解为列主序才可以解释得通）
> - 输出：将结果以 `uint64_t[3]`输出，不需使用的高位则全部补0。
> - 注意：输入以及输出均为二进制流，并且是以小端序存储的。

三、实验原理

我认为该实验的主体共分为两部分：一部分是解决对应的输入输出问题，在处理该问题时，我们需要考虑采取怎样的数据类型来存储对应的二进制流数据，并且能够方便我们对其进行运算；而另外一个部分就是要解决有限域上的各种运算问题，并且要考虑如何对这些内容进行优化以及提升。

- 输入输出以及数据类型的选择： 在刚开始做题的时候，本人并不能很好的处理好输入以及输出的问题，于是便选择先解决实验0-2的输入输出问题来为本实验做铺垫，但是我一开始无法理解样例给出的数据，直到询问了同学后才明白这是十六进制的**小端存储形式。** 之后便根据网站上的提示解决了输入输出问题。

```
1 5C
2 20 00 00 00
```

将后面这段 32 bytes（也就是 `0x20`）的数据与 `0x5C` 逐字节异

可是，当面对本实验时我们需要考虑用什么数据类型来存储可以更方便于我们的操作。在一开始，我使用的是 `uint64_t`的数据类型来存储以及操作，可是后面却发现该数据类型就连调试都十分困难。在寻找材料后，便选定了 `bitset`这个数据类型来作为本次实验操作的数据类型。`bitset`是C++标准库中提供的一种模板类，非常适合需要处理大量二进制数据的场景。由于我们是使用 `uint64_t`的数据类型来表示一个多项式，那么只需要在对应的x^i的i值处置1就可以解决。

- 有限域操作的实现以及提升： 在本次实验中，关于有限域的操作实现以及提升大部分都展示在了网站上，本人对一些trivial的做法作出了一些提升，具体的内容解释放在后面的实验步骤中。

四、实验步骤

- 输入输出部分 首先，先使用**条件编译**来配置线上测试以及线下调试，以便于我们能够在不同的编译环境以及平台上进行编写代码。在线上评测环境中，就是直接使用标准输入输出，而在本地调试环境中，就是通过网站的十六进制转二进制工具，从二进制文件中读取数据。

```
#ifdef ONLINE_JUDGE
#define in cin
#define out cout
#else
std::ifstream input("D:/Cryptolab/dump.bin", std::ios::binary);
std::ofstream output("output.txt");
#define in input
#define out output
#endif
```

其次，我们便要对读入的数据进行处理，将其转换成 bitset类型来处理。如代码所示，当我们读入二进制数据时，就将其存储在 uint64_t的数组中，并且使用 to_bitset函数来将其转换成对应的bitset类型。

```
        uint8_t type; // 需要进行的运算类型
        in.read((char *)&type, sizeof(type));
        uint64_t data[2][3] = {0};
        // 读取二进制数据
        for (int i = 0; i < 2; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                in.read((char *)&data[i][j], sizeof(data[i][j]));
            }
        }
        // 将第一个多项式转换为bitset并输出
        gf poly0 = to_bitset(data[0]);
        // 将第二个多项式转换为bitset并输出
        gf poly1 = to_bitset(data[1]);
```

对应的 to_bitset函数定义如下，同时也交代本程序设定的一些变量例如 M代表着一个多项式的位数，然后 m_bitset代表着进行运算时实际有的位数，最后就是将该类型定义成 gf有限域，使得表示更加规范。

```
const int M = 131;           // 定义多项式的位数
const int m_bitset = 2 * M;  // 因为后面进行乘法运算时，多项式最高位可以达到2*130
typedef std::bitset<m_bitset> gf;    // 用bitset来表示多项式

gf to_bitset(uint64_t data[3])
{
    gf result;
    // 将每个 uint64_t 的64位填入 bitset 的相应位置
    for (int i = 0; i < 3; ++i)
    {
        std::bitset<64> part(data[i]); // 每个64位部分
        for (int j = 0; j < 64; ++j)
        {
            result[i * 64 + j] = part[j]; // 将64位数填入bitset的相应位置
        }
    }
    return result;
}
```

最后，对于输出而言，因为最终要输出2进制流，所以便选择将bitset类型再次转化为 uint64_t数组，然后再进行输出。

```
void to_uint64_array(const gf &poly, uint64_t data[3])
{
    for (int i = 0; i < 3; ++i)
    {
        data[i] = 0; // 初始化为0
        for (int j = 0; j < 64; ++j)
        {
            data[i] |= (static_cast<uint64_t>(poly[i * 64 + j]) << j);
        }
    }
}

void gf_print(const uint64_t data[3])
{

     for (int i = 0; i < 3; i++)
     {
         out.write((char *)&(data[i]), sizeof(data[i]));
     }
}
```

至此输入输出部分就已经解决完毕。

- 有限域运算部分

1. 加法的实现 由于在该有限域下，每一项的系数只有可能是0或者1，所以加法只需要用位与位之间的异或就可以解决。

```
gf gf_add(const gf &a, const gf &b)
{
    return a ^ b; // 对两个位数相同的bitset执行按位异或
}
```

2. 取模操作的实现 为了能够让计算后的多项式都能表示成$F_{2^{131}}$有限域下的元素，我们需要实现取模运算，该运算也是其他运算的一个核心点，对于平方以及乘法这些运算会产生进位的操作而言，取模运算是不可或缺的。我们基于题目所给出的不可约多项式可得$x^{131}+x^{13}+x^2+x+1\equiv 0 \pmod{f(x)}$然后在两边同时加上$x^{13}+x^2+x+1$，便可以得到$x^{131}\equiv x^{13}+x^2+x+1 \pmod{f(x)}$，那我们便可以利用该性质，令$r=x^{13}+x^2+x+1$作为模多项式去消去多项式的那些高次项。具体代码如下：

```
gf gf_mod(gf a) // 取模操作
{
    gf r;
```

```
    r.set(13);
    r.set(2);
    r.set(1);
    r.set(0);
    //利用bitset来实现模多项式
    for (int i = 2 * M - 1; i >= M; i--) //将多余的部分都约分
    {
        if (a.test(i)) //判断此时的位是否为1
        {
            a.reset(i);
            gf tmp = r << (i - M);
            a = gf_add(a,tmp);
        }
    }
    return a;
}
```

在这其中，我们判断的范围是(2*M-1~M)这样就是把所有高于 m=131 的项直接约掉，才满足求模的定义，而后面的代码实现逻辑则是参考于给出的论文资料：从高位开始检查，如果为1，那么就代表着要把这位给清掉，并且将该模多项式移动到对应的地方进行相消，然后再实现多项式的相消过程。

Algorithm 2.40 Modular reduction (one bit at a time)

INPUT: A binary polynomial $c(z)$ of degree at most $2m-2$.
OUTPUT: $c(z) \bmod f(z)$.

1. *Precomputation*. Compute $u_k(z) = z^k r(z), 0 \le k \le W-1$.
2. For i from $2m-2$ downto m do
   2.1 If $c_i = 1$ then
   Let $j = \lfloor (i-m)/W \rfloor$ and $k = (i-m) - Wj$.
   Add $u_k(z)$ to $C\{j\}$.
3. Return$(C[t-1], \ldots, C[1], C[0])$.

3. 乘法操作的实现 对于乘法运算而言，多项式运算中难免会存在最高次项超出有限域的情况，所以在进行乘法操作时，我们要注意处理进位问题。处理乘法时，核心思想是通过按位扫描多项式的每一位，逐步构造出乘积多项式 ans，并且在乘法过程中进行必要的模多项式约减操作，以保证结果仍在有限域内。具体代码如下：

```
gf mul(gf a, const gf &b) // 乘法
{
    gf ans(0);                          // 初始化结果为 0
    for (int i = 0; i < b.size(); i++) // 遍历 b 的每一位
    {
        if (b[i])
            ans ^= a; // 这个操作可以用加法替代，但是直接使用感觉会更加明了

        a <<= 1;       // 移位操作是一定要执行的
        if (a.test(M)) // 如果左移后，a 的最高位（第 M 位）为 1
```

```
            a ^= f;     // 用模多项式 f 进行约减
        }
        return gf_mod(ans);
    }
```

当 b[i]为1时，就说明此时要进行乘法，将a的当前值加到 ans 中；而当该值不为1时就不进行其他操作，在判断结束后，一定要将a移位来进行乘法，如果此时a的最高位超出有限域的范围的话，同样也需要对其进行约减。

4. 平方操作的实现 平方操作最简单的实现思路就是直接调用乘法，但是网站指出可以不直接调用乘法。平方操作意味着将多项式中每个项的指数加倍，具体而言多项式中的每个幂次项x^i变成x^{2i}，所以我们可以不用调用乘法的内容而是将对应 i位的数据赋值给 2i就行。具体代码如下：

```
gf square(const gf &a) // 平方的结果是将多项式的每个项的指数加倍
{
    gf b;
    for (int i = 0; i < M; i++)
        b[i * 2] = a[i];
    return gf_mod(b);
}
```

5.求逆操作的实现 求逆操作按照本次实验要求而言，需要实现两种方法：一种是扩展欧几里得算法，而另外一种则是基于费马小定理，接下来会分别介绍二者： 扩展欧几里得： 该算法的目标是找到一个多项式 b,使得$a * b \equiv 1 \pmod{p}$,首先定义一个 degree函数,该函数用来计算多项式 a的最高次幂。

```
int degree(const gf &a)
{
    for (int i = 2 * M - 1; i >= 0; i--)
        if (a[i] || i == 0)
            return i;
}
```

而扩展欧几里得算法的主要实现,就是通过一系列的多项式除法操作，逐步逼近逆元。

```
gf Euc(const gf &a, const bitset<M + 1> &p)
{
    gf b;                       // 用于存储逆元
    gf c;                       // 辅助多项式
    gf u = a;                   // 将 a 赋值给 u
    gf v = gf(p.to_string()); // 将模多项式 p 赋值给 v
    b[0] = 1;                   // 初始化 b 为 1，表示常数多项式
    int degU, degV;

    // 计算次数并且循环直至 u 的次数为零
    while ((degU = degree(u)) > 0)
    {
        degV = degree(v);

        // 保证 degU >= degV
        if (degU < degV)
        {
            std::swap(u, v);
            std::swap(b, c);
            std::swap(degU, degV); // 同步更新次数
        }

        int j = degU - degV;

        // 使用异或进行多项式减法
        u ^= (v << j);
        b ^= (c << j);
    }

    return gf_mod(b); // 返回经过模操作后的逆元
}
```

费马小定理: 关于费马小定理的代码实现，本人是直接参考网站里面给出的Itoh-Tsujii算法，通过将其伪代码实现而直接完成的。

ALGORITHM 1.

S1. $y := x$
S2. for $k := 1$ to $m - 2$ do
S3. begin
S4. $z := y^2$ (one cyclic shift)
S5. $y := zx$ (multiplication in GF(2^m))
S6. end
S7. $y := y^2$ (one cyclic shift)
S8. write y

```
gf Fer(const gf &x)
{
    gf y(x);
    for (int i = 1; i <= M - 2; i++)
    {
        gf z = square(y); //只需要构造一个2^m-1即可
        y = mul(z, x);
```

```
    }
    y = square(y);
    return y;
}
```

但是在做思考题的时候,根据公式进行推导可以得到费马定理的一个简化形式,此时完成多次的平方以及乘法就可以实现对应的要求:而这个代码中的 `gf_pow2`以及 `gf_mul`其实就是先前实现的mul以及square,只不过这里为了要实现对应的逻辑而多增加了一些参数而已。

```
// 使用费马小定理求逆元
void gf_inverse(gf &result, const gf input)
{
    // 定义一个辅助函数来进行多次平方操作
    auto multiple_square = [](gf &output, const gf &value,
int times)
    {
        output = value;
        for (int i = 0; i < times; i++)
        {
            gf_pow2(output, output);
        }
    };

    // step1 = input^3
    gf_pow2(result, input);
    gf_mul(result, result, input);
    gf step1 = result;

    // step2 = step1^5
    multiple_square(result, result, 2);
    gf_mul(result, result, step1);
    gf step2 = result;

    // step3 = step2^17
    multiple_square(result, result, 4);
    gf_mul(result, result, step2);
    gf step3 = result;

    // step4 = step3^257
    multiple_square(result, result, 8);
    gf_mul(result, result, step3);
    gf step4 = result;

    // step5 = step4^65537
    multiple_square(result, result, 16);
    gf_mul(result, result, step4);
    gf step5 = result;

    // step6 = step5^(2^32 + 1)
    multiple_square(result, result, 32);
```

```
    gf_mul(result, result, step5);
    gf_pow2(result, result);
    gf_mul(result, result, input);
    gf step6 = result;

    // step7 = step6^(2^65 + 1)
    multiple_square(result, result, 65);
    gf_mul(result, result, step6);
    gf_pow2(result, result);

    return;
}
```

五、实验结果

- 输入:

```
05 20 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
21 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
04 00 00 00 00 00 00 00
```

- 输出: 加法:
```
24 20 00 00 00 00 00 00
00 00 00 00 00 00 00 00
04 00 00 00 00 00 00 00
```
乘法:
```
ab 10 04 02 00 00 00 00
00 00 00 00 00 00 00 00
04 00 00 00 00 00 00 00
```
平方:
```
11 00 00 04 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```
求逆:
```
7f 01 dd 8a ed da 46 92
42 ef 37 99 f4 d0 f9 0d
03 00 00 00 00 00 00 00
```

  可以看到运算结果全部正确。

六、实验总结

通过本次实验，我成功实现了有限域中多项式的基本运算，掌握了位集在数学运算中的有效应用。该实验不仅加深了我对有限域理论的理解，也为我实现将来的相关加密算法实现奠定了基础。在本次实验中，我遇到的了许多问题包括如何解决二进制流的输入输出以及要如何将论文中的伪代码实现变为可供运行的代码,在通过查询资料、询问同学以及自己的努力下也是成功克服了这些问题，自己也是感觉收益匪浅。

参考资料:F(2^131)下有限域基本运算的实现(c++)_有限域运算编程-CSDN博客

七、思考题

- 共用了8次乘法以及130次平方,调整对应代码使得其能做到单独的调试

```
// 将第一个多项式转换为bitset并输出
gf gf0 = to_bitset(data[0]);
std::cout << "gf 0: " << gf0 << std::endl;

// 将第二个多项式转换为bitset并输出
gf gf1 = to_bitset(data[1]);
std::cout << "gf 1: " << gf1 << std::endl;
// 将 gf 转换回 uint64_t 数组
auto ts = std::chrono::high_resolution_clock::now();
gf res = mul(gf0,gf1);
auto te = std::chrono::high_resolution_clock::now();
std::cout
    << "Duration: "
    << std::chrono::duration_cast<std::chrono::nanoseconds>(te - ts).count()
    << "ns"
    << std::endl;
uint64_t converted_data[3];
to_uint64_array(res, converted_data);
```

乘法运行时间:`Duration: 22900ns`, 平方运行时间:`Duration: 24400ns`,总时间计算就带入统计的几次乘法以及平方即可解决: `3355200ns`,实际时间:
`Duration: 2319100ns`