

实验报告

【实验目的】

1. 通过本次实验，熟悉编程环境，为后续实验做好铺垫。
2. 回顾数论以及有限域上的基本算法，加深对其理解，为本学期密码学课程及实验课打好基础。
3. 通过本次实验，了解上元素的性质及其四则运算。
4. 掌握上的不可约多项式的判定和生成算法。
5. 【必做】部分第 4 题需要在报告中给出算法流程图和伪代码。

【实验环境】

1. 语言：C
2. 平台：clion 2021.2 版本

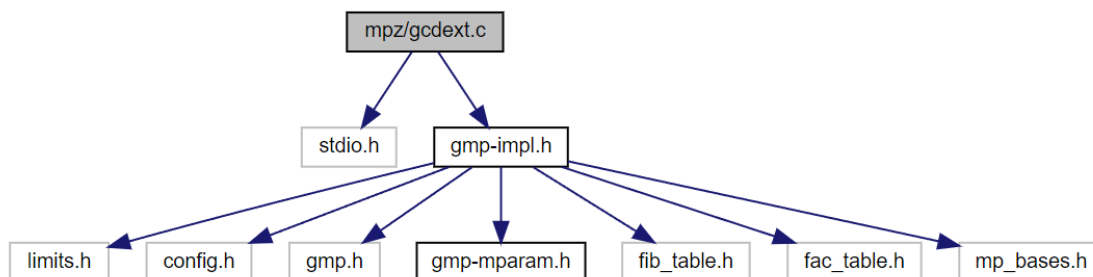
【实验内容】

一、欧几里得算法

1. 算法流程

在C语言自带的gmp大数据库中，调用mpz_gcdext函数即可实现对两个大整数的欧几里得算法。

其具体函数调用关系如下：



对于输入的大整数 a ， b 及欧几里得等式：

$$a * x + b * y = gcd(a, b)$$

由于gmp大数据库可以快速的计算出 $gcd(a, b)$ 及 x ， y 的一组解，从而我们可以对该式变换得到多组 x ， y 的解：也就是 x ， y 的通解：

$$x = x_0 - \frac{k * b}{gcd(a, b)}$$
$$y = y_0 + \frac{k * a}{gcd(a, b)}$$

因此，我们可以采用如下的方式将解得的 x 变为满足题意的最小正整数：

1.
$$flag = \frac{b}{gcd(a, b)}$$
2.
$$x = x_0 \% flag$$

这样就使得最后的 x 成为最小的正整数。

3.
$$k = \frac{x - x_0}{flag}$$

这样我们就解得了k, 进而可以求得符合题目条件的x, y的值。

2. 测试样例及结果截图

1. 样例1:

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"
31 -13
8 19 1
Process finished with exit code 0
```

2. 样例2:

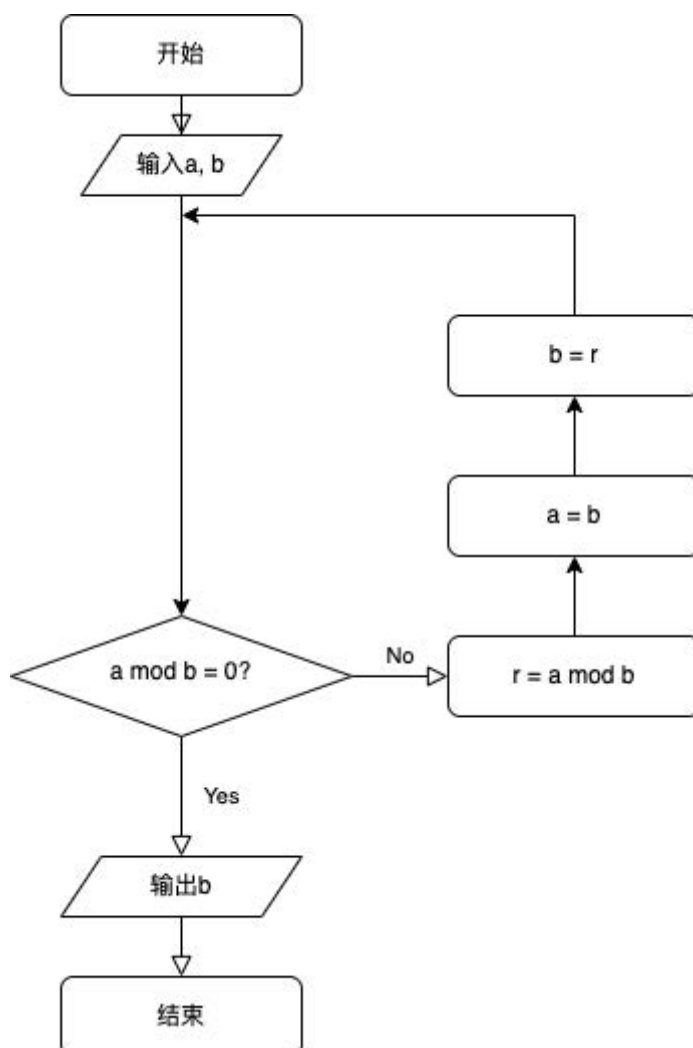
```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"
274891284672781637261476728489367864781 -8717238783782738921783782738273827
647772216679604987037235312010186 20426988549362888895096094620021204617 7
Process finished with exit code 0
```

3. 样例3:

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"
24 36
2 -1 12
Process finished with exit code 0
```

3. 讨论与思考

其实对于求最大公约数, 可以直接借用gmp大数库中的简单四则运算, 通过如下算法获得:



但考虑到这样的话在速度上可能不如直接使用gmp中的函数，所以就没有采用这种方法。

二、快速幂算法

1. 算法流程

```
input : B,P,M
if(P < 0){
    P = -P;
    B = invert(B, M);
}
temp = B % M;
flag = P % 2;
P = P / 2;
flag == 1? ans = temp : ans =1;
while(P != 0){
    flag = P % 2;
    P = P / 2;
    temp = temp^2 % M;
    if(flag == 1){
        ans = ans * temp % M;
    }
}
output : ans
```

2. 测试样例及结果截图

1. 样例1:

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"
7 16 3
1
Process finished with exit code 0
```

2. 样例2:

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"
5 1003 31
5
Process finished with exit code 0
```

3. 样例3:

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"
671827219 87218378217387283782 283892839281908
18945444165233
Process finished with exit code 0
```

3. 讨论与思考

其实在这里我们可以使用gmp大数库中自带的mpz_get_str函数将大数直接转化为二进制形式的字符串，这样也可以对上述的算法进行优化。

三、中国剩余定理：

1. 算法流程

Algorithm 1 中国剩余定理

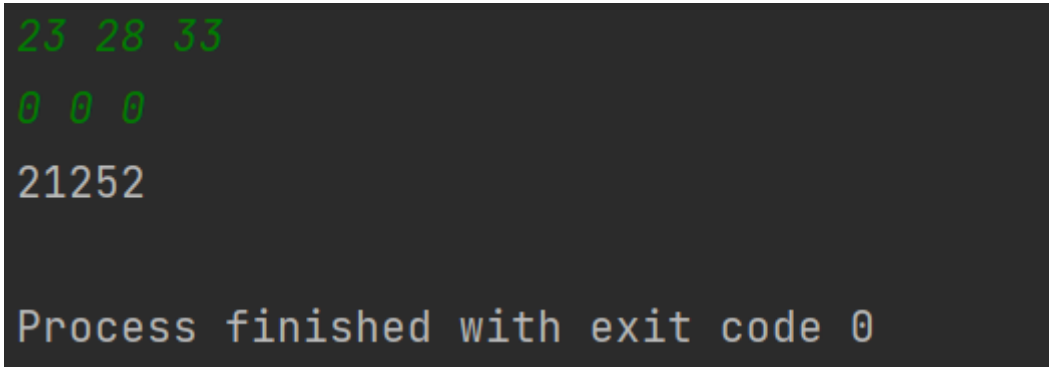
Input: $a_1, a_2, a_3, b_1, b_2, b_3$

Output: 同余方程组的解 m

```
1: function CRT( $a_1, a_2, a_3, b_1, b_2, b_3$ )
2:    $M \leftarrow a_1 * a_2 * a_3$ 
3:    $M_1 \leftarrow a_2 * a_3$ 
4:    $M_2 \leftarrow a_1 * a_3$ 
5:    $M_3 \leftarrow a_1 * a_2$ 
6:    $t_1 \leftarrow \text{INVMOD}(M_1, a_1)$ 
7:    $t_2 \leftarrow \text{INVMOD}(M_2, a_2)$ 
8:    $t_3 \leftarrow \text{INVMOD}(M_3, a_3)$ 
9:    $m \leftarrow b_1 * t_1 * M_1 + b_2 * t_2 * M_2 + b_3 * t_3 * M_3$ 
10:  return  $m$ 
11: end function
12:
13: function INVMOD( $a, n$ )
14:    $d, x, y \leftarrow \text{exEuclid}(a, n)$ 
15:   while  $d \neq 1$  do
16:     return  $x$ 
17:   end while
18: end function
```

2. 测试样例及结果截图

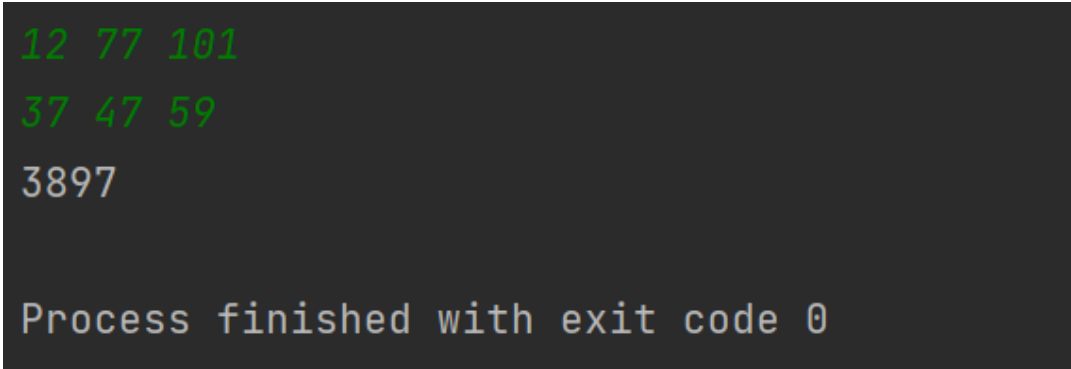
1. 样例1：



```
23 28 33
0 0 0
21252

Process finished with exit code 0
```

2. 样例2：



```
12 77 101
37 47 59
3897

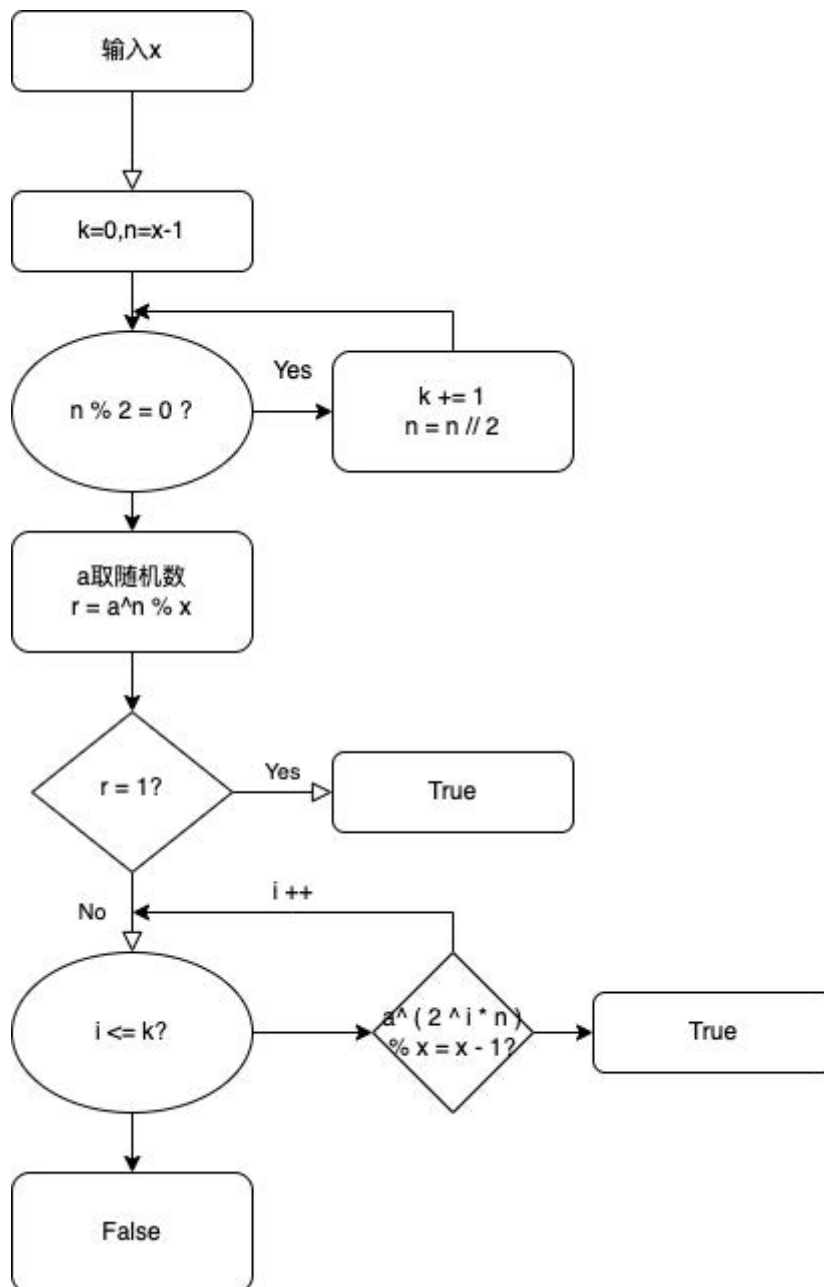
Process finished with exit code 0
```

3. 讨论与思考

遇到样例1的情况，需要在算出的结果中再加上模数便能得到符合题意的解。

四、Miller-Rabin 素性检测

1. 算法流程



2. 伪代码为：

Algorithm 1 MillerRabin 素性检测

Input: x **Output:** 是否为素数

```
1: function MILLERRABINTEST( $x$ )
2:    $k \leftarrow 0$ 
3:    $n \leftarrow x - 1$ 
4:   while  $n$  为偶数 do
5:      $k \leftarrow k + 1$ 
6:      $n \leftarrow (n/2)$ 
7:   end while
8:    $a \leftarrow \text{randominteger}$ 
9:    $r \leftarrow a^n \pmod{x}$ 
10:  if  $r = 1$  then
11:    return True
12:  end if
13:  while  $i \leq k$  do
14:    if  $a^{2^i \cdot n} \pmod{x} = x - 1$  then
15:      return True
16:    end if
17:     $i \leftarrow i + 1$ 
18:  end while
19:  return False
```

3. 测试样例及结果截图

1. 样例1:

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"
21789372186478136278463761847
NO
Process finished with exit code 0
```

2. 样例2:

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"
1000033
NO
Process finished with exit code 0
```

3. 样例3:

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"
1000033
YES
Process finished with exit code 0
```

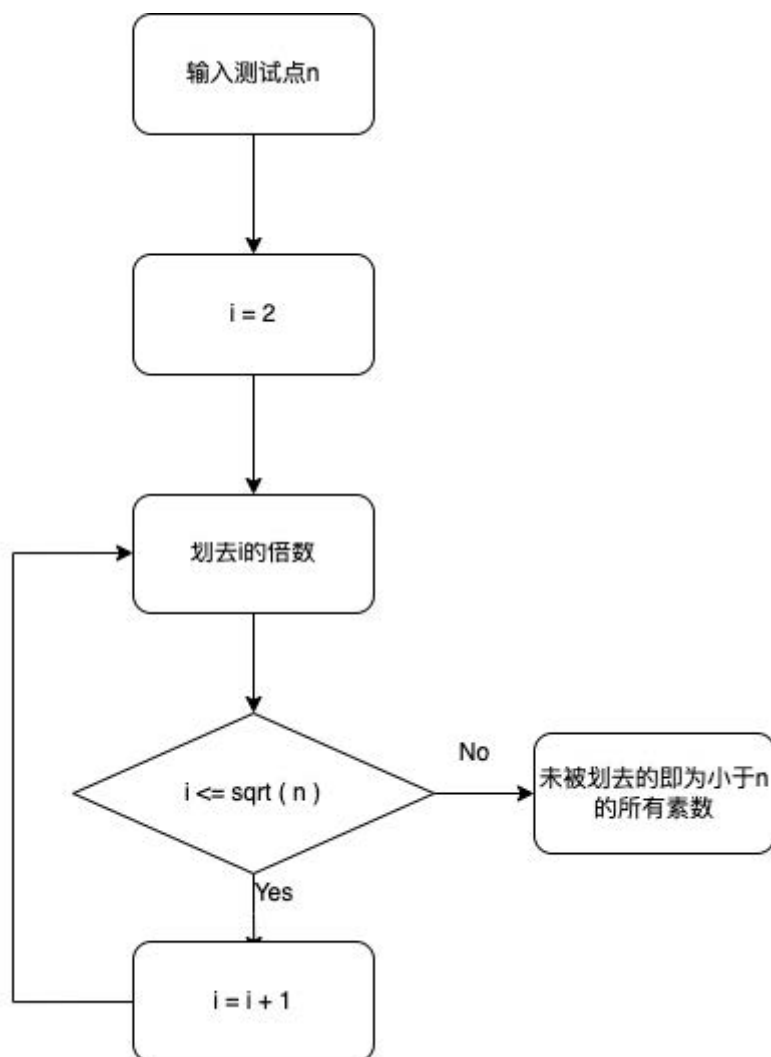
4. 讨论与思考

Miller-Rabin 素性检测只是一个概率性结果，得到的只能说是可能是素数或可能不是素数，所以当检测次数增加时可以一定程度增加检测有效的概率，一般检测到10次左右即可，当次数过多时反而会增加时间成本使得超时。在实际的实验中，我也观察到了一次判断异常的状况。

五、厄拉多塞筛法

1. 算法流程

大体的思路如下：



2. 测试样例及结果截图

1. 样例1：

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"  
23  
2 3 5 7 11 13 17 19 23  
Process finished with exit code 0
```

2. 样例2：

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"  
133  
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 1  
63 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241  
Process finished with exit code 0
```

3. 样例3：

```
"E:\E_drive\clion\Project List\cryptography\exp1\cmake-build-debug\exp1.exe"  
1234  
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 1  
63 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 3  
37 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 5  
21 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 7  
19 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 9  
29 937 941 947 953 967 971 977 983 991 997 1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091 1093 10  
97 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231  
Process finished with exit code 0
```

3. 讨论与思考

对于该算法的优化，通过利用之前所得的素数表可以用空间换取时间，使得计算效率大大提升。

六、有限域四则运算

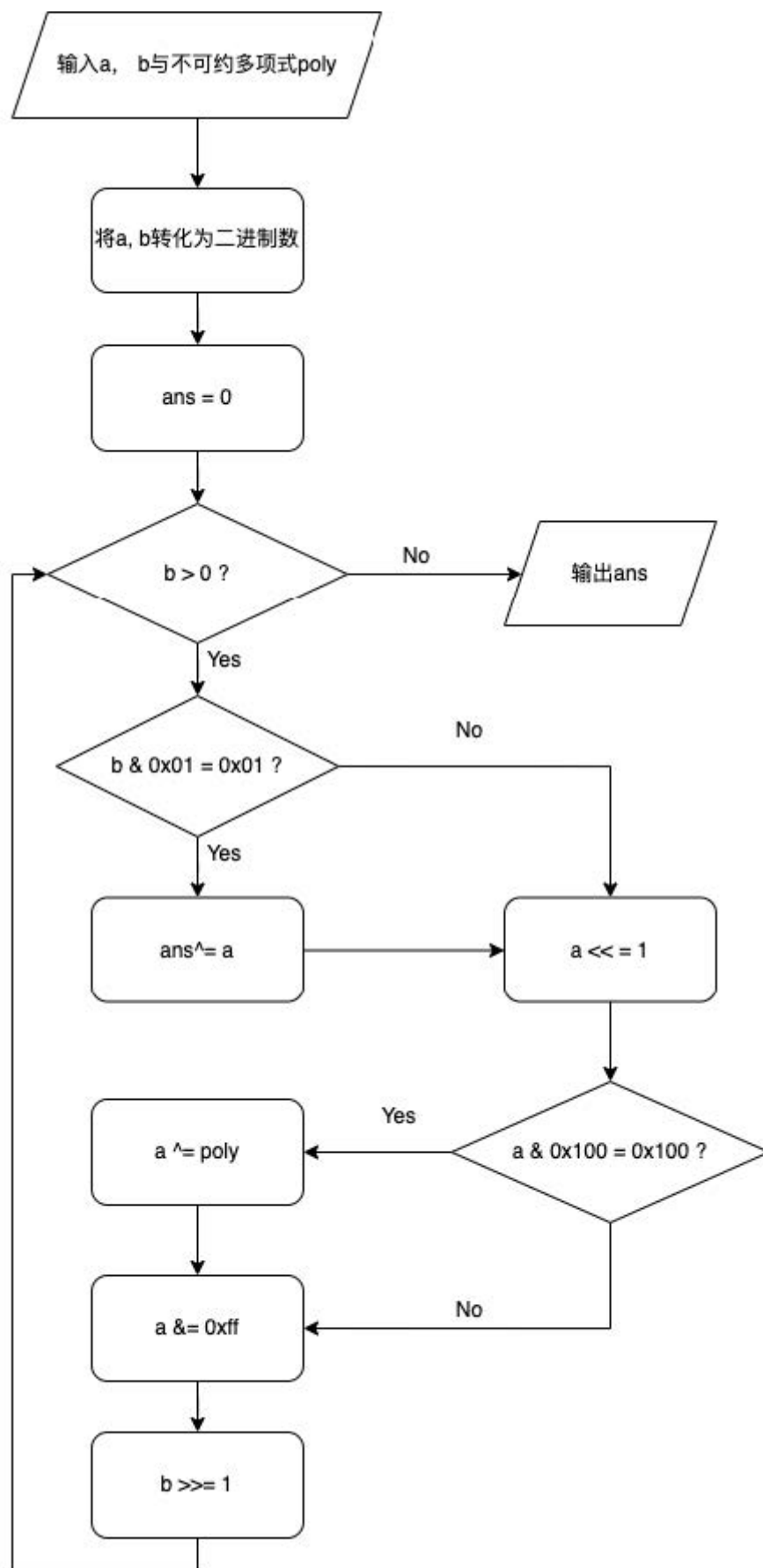
1. 算法流程

1. 有限域加法与减法：

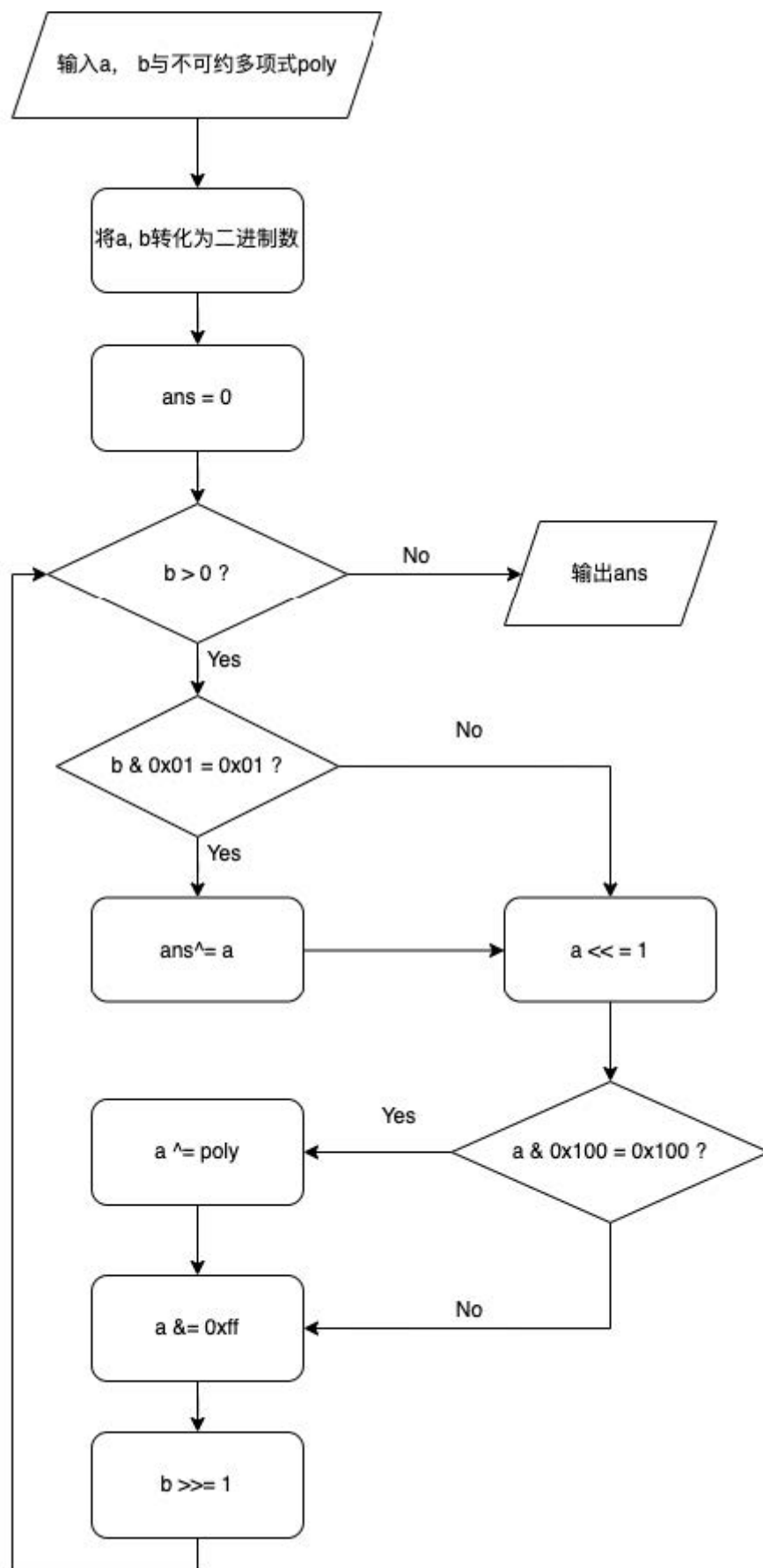
$GF(2^n)$ 上的加减法等价于 $GF(2)$ 上多项式的加减法，即系数的加减法，而对于 $GF(2)$ 而言，加减法即为异或运算，故加减法均为按位异或运算。

2. 有限域的乘法：

已知不可约多项式 (poly) 为0x11b:



3. 有限域的除法:



其实本质上就是对乘法与除法的竖式运算模拟，将加减操作变换为异或运算。

4. 在这里，我采用的是模拟竖式操作的方式实现的有限域的四则运算，在函数内部对二进制数进行操作时，采用如下标准化函数将其前置0去掉，从而方便函数关于接口的封装与后续的操作：

```
char *standard(char *binary_s)
{
    mpz_t s;
    mpz_init_set_str(s, binary_s, 2);
    char *ans = NULL;
    ans = mpz_get_str(ans, 2, s);
    return ans;
}
```

5. 异或模拟：

```
char *GF_2_xor(char *str1, char *str2)
{
    //对输入的二进制字符串标准化操作
    str1 = standard(str1);
    str2 = standard(str2);
    int len1 = (int)strlen(str1), len2 = (int)strlen(str2), cnt1, cnt2;
    int cnt = (len1 > len2 ? len1 : len2);
    char *re = malloc(sizeof(char) * (cnt + 1));
    re[cnt] = '\0';
    cnt--;
    for(cnt1 = len1 - 1, cnt2 = len2 - 1; cnt1 >= 0 && cnt2 >= 0; cnt1--, cnt2--)
        re[cnt--] = xor(str1[cnt1], str2[cnt2]);
    if(cnt1 == -1)
        while(cnt2 >= 0)
            re[cnt--] = str2[cnt2--];
    else
        while(cnt1 >= 0)
            re[cnt--] = str1[cnt1--];
    //对输出标准化
    re = standard(re);
    return re;
}
```

6. GF域内多项式（二进制数）的除法运算：

```
typedef struct result{
    char *ans;
    char *remain;
}re, *ptr;

//计算完成后返回商字符串的首地址，被除字符串变为余数字符串
re GF_2_poly_division(char *binary_dividend, char *binary_divisor)
{
    //对输入的二进制字符串标准化操作
    binary_dividend = standard(binary_dividend);
    binary_divisor = standard(binary_divisor);
    //设置结构体承载输出变量
    re result;
    int degree1 = (int)strlen(binary_dividend) - 1, degree2 =
(int)strlen(binary_divisor) - 1;
    //使用binary_dividend_re操作，防止改变binary_dividend
```

```

char *binary_dividend_re = malloc(sizeof(char)*LEN);
strcpy(binary_dividend_re, binary_dividend);
if(degree1 < degree2){
    char *none = malloc(sizeof(char)*LEN);
    none[0] = '0'; none[1] = '\0';
    result.ans = none;
    result.remain = binary_dividend_re;
    return result;
}else{
    char *ans_str = malloc(sizeof(char) * (degree1 - degree2 + 2));
    memset(ans_str, '0', degree1 - degree2 + 2);
    ans_str[degree1 - degree2 + 1] = '\0';
    int ansCnt = 0, cnt1 = 0, cnt2 = 0;
    for (; cnt1 <= degree1 - degree2; cnt1++) {
        ans_str[ansCnt] =
            (char) ((binary_dividend_re[cnt1] - '0') /
(binary_divisor[0] - '0') + '0');
        if (ans_str[ansCnt] == '1')
            for (cnt2 = 0; cnt2 < degree2 + 1; cnt2++)
                binary_dividend_re[cnt2 + cnt1] =
                    xor(binary_dividend_re[cnt2 + cnt1],
binary_divisor[cnt2]);
        ansCnt++;
    }
    //对输出标准化
    ans_str = standard(ans_str);
    binary_dividend_re = standard(binary_dividend_re);
    result.ans = ans_str;
    result.remain = binary_dividend_re;
    return result;
}
}

```

7. GF域内多项式（二进制数）的乘法运算：

```

char *GF_2_poly_multiplication(char *binary_s1, char *binary_s2) {
    //标准化输入
    binary_s1 = standard(binary_s1);
    binary_s2 = standard(binary_s2);
    int digit1 = (int)strlen(binary_s1), digit2 =
(int)strlen(binary_s2), cnt1, cnt2, time = 1;
    char *ans = malloc(sizeof(char) * (digit1 + digit2));
    memset(ans, '0', digit1 + digit2);
    ans[digit1 + digit2 - 1] = '\0';
    for (cnt1 = digit2 - 1; cnt1 >= 0; cnt1--, time++)
        if (binary_s2[cnt1] == '1')
            for (cnt2 = digit1 - 1; cnt2 >= 0; cnt2--)
                ans[cnt2 + digit2 - time] = xor(ans[cnt2 + digit2 -
time], binary_s1[cnt2]);
    //标准化输出
    ans = standard(ans);
    return ans;
}

```

8. GF域内加减法运算：

```
char *GF_2_add_sub(char *binary_s1, char *binary_s2, char
*binary_modstr)
{
    re result;
    //binary_modstr是域上的不可约多项式系数的二进制表示
    char *flag_str = NULL;
    flag_str = GF_2_xor(binary_s1, binary_s2);
    result = GF_2_poly_division(flag_str, binary_modstr);
    return result.remain;
}
```

9. GF域内乘法运算:

```
char *GF_2_mul(char *binary_s1, char *binary_s2, char *binary_modstr)
{
    re result;
    //binary_modstr是域上的不可约多项式系数的二进制表示
    char *flag_str;
    flag_str = GF_2_poly_multiplication(binary_s1, binary_s2);
    result = GF_2_poly_division(flag_str, binary_modstr);
    return result.remain;
}
```

10. 之后便是写16进制数的封装函数，只需要将读入的数转化为二进制，将二进制运算结果转化为16进制即可。

2. 测试样例及结果截图

1. 样例1:

```
"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"
c7 * 0d
ea

Process finished with exit code 0
```

2. 样例2:

```
"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"
0 /1c
0
0

Process finished with exit code 0
```

3. 讨论与思考

如果采用gmp中的mpz_xor函数可能会更快，在之后我会对这个进行优化。

七、有限域快速模幂算法

1. 算法流程

与二、快速幂算法流程相同，唯一不同的是这里采用的是有限域中的四则运算。

2. 测试样例及结果截图

1. 样例1:

```
"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"  
60 23084021985  
ef  
  
Process finished with exit code 0
```

2. 样例2:

```
"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"  
60 0  
1  
  
Process finished with exit code 0
```

3. 样例3:

```
"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"  
0 6217863782673812  
0  
  
Process finished with exit code 0
```

八、有限域欧几里得算法

1. 算法流程

主要是通过递归调用实现:

```
public int[] exEuclid(int a1, int a2) {  
    if (a2 == 0)  
        返回(a1, 1, 0);  
    else {  
        (GCD, xtmp, ytmp) = exEuclid(a2, mod(a1, a2));  
        x = ytmp;  
        y = xtmp ^ multiply(ytmp, divide(a1, a2));  
        返回(GCD, x, y);  
    }  
}
```

```
typedef struct gcd_result {  
    char *gcd;  
    char *x;  
    char *y;  
} gcd_re, *gcd_ptr;  
  
gcd_re GF_2_exEuclid(char *binary_s1, char *binary_s2, char *modstr)  
{  
    binary_s1 = standard(binary_s1);  
    binary_s2 = standard(binary_s2);  
    modstr = standard(modstr);
```

```

gcd_re result, temp;
if(binary_s2[0] == '0'){
    result.gcd = binary_s1;
    result.x = malloc(sizeof(char)*LEN);
    strcpy(result.x, "1");
    result.y = binary_s2;
    return result;
}else{
    temp = GF_2_exEuclid(binary_s2, GF_2_poly_division(binary_s1,
binary_s2).remain, modstr);
    result.gcd = temp.gcd;
    result.x = temp.y;
    result.y =
        GF_2_xor(temp.x, GF_2_mul(temp.y, GF_2_poly_division(binary_s1,
binary_s2).ans, modstr));
    return result;
}
}

```

2. 测试样例及结果截图

1. 样例1:

```

"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"
x 74
19 3 2
Process finished with exit code 0

```

2. 样例2:

```

"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"
0 74
0 1 74
Process finished with exit code 0

```

3. 样例3:

```

"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"
1 1
0 1 1
Process finished with exit code 0

```

九、有限域求逆元

1. 算法流程

借用上面的有限域拓展欧几里得算法即可。即将要求最大公约数的两个数中的一个换成不可约多项式0x11b即可。

```

char *s1 = malloc(sizeof(char)*LEN);
gcd_re re;
scanf("%s", s1);
re = GF_2_hex_exEuclid(s1, mod_str, mod_str);
puts(re.x);
return 0;

```

2. 测试样例及结果截图

1. 样例1:

```
"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"  
1  
  
Process finished with exit code 0
```

2. 样例2:

```
"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"  
0a  
d7  
  
Process finished with exit code 0
```

3. 样例3:

```
"E:\E_drive\clion\Project List\cryptography\Experiment1\cmake-build-debug\Experiment1.exe"  
11a  
1  
  
Process finished with exit code 0
```

十、本原多项式的生成

1. 算法流程

- 在对本原多项式进行判断时，采用的是如下判断的方法：

提示：

若 $f(x)$ 是 $GF(2)$ 上的不可约多项式，且 $f(x)$ 的根是 $GF(2^8)$ 的本原元，则称 $f(x)$ 为 $GF(2)$ 上的本原多项式；另一等价定义如下：

若一个 n 次多项式 $f(x)$ 满足以下条件：

1. $f(x)$ 为不可约多项式；
2. $f(x)$ 可整除 $x^m + 1$ ($m = 2^n - 1$)；
3. $f(x)$ 不能整除 $x^q + 1$ ($q < m$)；

则称 $f(x)$ 为 n 次本原多项式。

- 算法流程如下：

Algorithm 1 本原多项式的判定与生成

Input: none**Output:** 所有 8 次本原多项式

```
1: function PRIMITIVE.POLYNOMIAL.TEST(none)
2:    $lt \leftarrow IrreduciblePolynomial$ 
3:    $i \in lt$ 
4:   for  $i$  do
5:      $flag \leftarrow IOP(i)$ 
6:     if  $flag = 0$  then
7:        $print(i)$ 
8:     end if
9:   end for
10: end function
11: function IOP( $i$ )
12:    $b \leftarrow x^{255} + 1$ 
13:   if  $div(b, i) = 0$  then
14:      $flag \leftarrow 0$ 
15:   end if
16:   for  $q = 1$  to 255 do
17:     if  $div(x^q + 1, i) = 0$  then
18:        $flag \leftarrow 1$ 
19:       break
20:     end if
21:   end for
22:   return  $i$ 
23: end function
```

2. 测试样例及结果截图

1. 样例:

```
100011101
100101011
100101101
101001101
101011111
101100011
101100101
101101001
101110001
110000111
110001101
110101001
111000011
111001111
111100111
111110101

Process finished with exit code 0
```

3. 讨论与思考

在这里，先将所有小于等于4次的不可约多项式打印出来，再利用信安数基中学到的筛法可以轻易的得到所有的本原多项式。

十一、收获与建议

1. 本次实验中我有以下收获：

通过对上学期学到的有关数论知识动手复现，更加加深了我对这些密码学会用到的基础算法的理解。

使用C语言进行尝试也增强了我的编程能力。

2. 对于本次实验，我认为可以做这些改进：

将域中的四则运算所涉及到的相关运算的算法进行进一步的优化，具体上是不采用模拟的方式，而是利用gmp中已有的相关函数进行速度即准确度上的优化。