암호학 HW1

2018037356 – 안동현

(아래 사진들은 코드의 캡쳐를 위해 주석을 지운 상태입니다. 본 파일에는 주석이 존재합니다)

1. int gcd(int a, int b);

```
17 int gcd(int a, int b)
18 {
19     while(b != 0){
20         int tem = b;
21         b = a % b;
22         a = tem;
23     }
24     return a;
25 }
26
```

gcd(a, b) = gcd(b, a % b) 와 같음을 이용한 방식이다.

재귀 호출은 일반적으로 스택에 계속해서 메모리를 쌓아가고 다시 복귀주소로 리턴해주는 과정을 가지기에 특별한 경우가 아니라면 반복문이 더 높은 성능을 발휘한다. 매 반복마다 a 에는 b를 넣고, b에는 a % b를 넣는 과정을 진행한다.

```
========== 기본 gcd 시험 ==========
gcd(28,0) = 28
gcd(0,32) = 32
gcd(41370,22386) = 42
gcd(22386,41371) = 1
```

매우 유명한 알고리즘 중 하나이고 이미 숙지하고 있던 알고리즘이라 어려움 없이 구현할 수 있었다.

2. int xgcd(int a, int b, int *x, int *y);

```
34 int xgcd(int a, int b, int *x, int *y)
35 {
36     int x0 = 1, y0 = 0;
37     int x1 = 0, y1 = 1;
38     int q, r, tem_x, tem_y;
39
40     while (b != 0){
41         q = a / b;
42         r = a % b;
43
44         tem_x = x1;
45         tem_y = y1;
46
47         x1 = x0 - q * x1;
48         y1 = y0 - q * y1;
49
50         x0 = tem_x;
51         y0 = tem_y;
52
53         a = b;
54         b = r;
55     }
56     *x = x0;
57     *y = y0;
58
59     return a;
60
61 }
```

gcd(a,b) = d = ax + by의 식을 만족한다고 했을 때, gcd(a,b) = gcd(b,a%b) = gcd(a%b,b%(a%b)) 과정으로 나아가며 이때 각각의 인자값들을 d0, d1, d2, d3 ..... 로 가정할 수 있다. 따라서 이걸 ax + by의 식으로 나타내면 이러하다.

d0 = a = ax0 + by0

d1 = b = ax1 + by1

d2 = d0 - (d0/d1)d1 = ax2 + by2 (d의 연산은 d0 mod d1)

d3 = d1 - (d1/d2)d2 = ax3 + by3 (d의 연산은 d1 mod d2)

위 과정을 반복했을 때 일반식이 나온다.

몫 = qi = di-1 div d1

나머지 = di+1 = (di-1) - qi * di

이 과정에서 x, y에 관해서도 똑같이 적용되는데,

xi+1 = (xi-1) - qi * xi

yi+1 = (yi-1) - qi * yi

의 과정을 반복한다. 이 과정을 끝까지 했을 때의 값 즉 dk + 1 = 0 -> gcd(a,b) = dk = axk + byk 일 때의 x와 y의 값을 구해주면 된다.

3. int mul_inv(int a, int m);

```
69 int mul_inv(int a, int m)
70 {
71     int d0 = a, d1 = m;
72     int x0 = 1, x1 = 0, q, tem;
73
74     while (d1 > 1){
75         q = d0 / d1;
76         tem = d0 - q * d1; d0 = d1; d1 = tem;
77         tem = x0 - q * x1; x0 = x1; x1 = tem;
78     }
79
80     if (d1 == 1){
81         return (x1 > 0 ? x1 : x1 + m);
82     }
83     else
84         return 0;
85 }
```

만약 a,b가 서로소라면 둘의 최대공약수는 1이다. 따라서 gcd(a,b) = 1 이다.

만약 dk = 1 이라면

1 = axk + byk

1 mod b = axk + byk mod b = 1

우리는 a의 xk에만 관심이 있기 때문에 b를 버리자. 따라서 axk mod b = 1 이므로 xk = a^-1 mod b 이다. 즉 a, m이 서로소임을 확인하고, xk의 값을 구해준다.

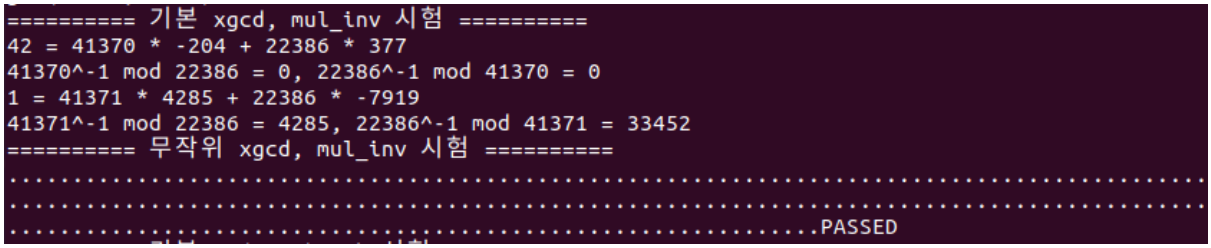
d1 > 1까지 진행하는 이유는 마지막 반복을 돌고 d1을 확인해봤을 때 d1 = 1이라면 둘의 공약수

가 1밖에 없다는 소리이고 d1 = 0이라면 둘의 공약수가 1이 아닌 무언가가 존재한다는 소리

가 되므로 그렇다.

마지막 리턴 부분을 보면

x는 음수와 양수가 왔다갔다 거린다. 만약 x가 음수라면 mod m 계산이니 m을 더해줘도 없는 취급이 되므로 동일한 값의 연산이나 마찬가지이다. 따라서 m을 더해 양수로 만들어준다.



위 결과에서 실제로 41370 * -204 + 22386 * 377 = 42임을 확인할 수 있고

그 밑에 mul_inv를 보면 서로소가 아닌 41370, 22386의 결과에서는 return 0을 통해 0이 출력되는 모습과 41371을 했을 때는 서로소를 만족해 모듈려 역원에 대한 값들이 제대로 양수로 출력되는 것을 확인하는 것이 가능하다.

4. uint64_t umul_inv(uint64_t a, uint64_t b);

```
94 uint64_t umul_inv(uint64_t a, uint64_t m)
95 {
96     uint64_t d0 = a, d1 = m;
97     uint64_t x0 = 1, x1 = 0, q, tem, t;
98     int isNx0 = 0, isNx1 = 0, isNt = 0;
99
100    while (d1 > 1){
101        q = d0 / d1;
102        tem = d0 - q * d1;
103        d0 = d1;
104        d1 = tem;
105
106        t = x1; isNt = isNx1;
107
108        if (isNx0 != isNx1){
109            x1 = x0 + q * x1;
110            isNx1 = isNx0;
111        }
112        else{
113            x1 = (x0 > q * x1) ? x0 - q * x1 : q * x1 - x0;
114            isNx1 = (x0 > q * x1) ? 0 : 1;
115        }
116        x0 = t; isNx0 = isNt;
117    }
118
119    if (d1 == 1){
120        return (isNx1 > 0 ? m - x1 : x1);
121    }
122    else
123        return 0;
124 }
```

64bit unsigned int의 특징은 음수를 표현하지 않는다.

하지만 x의 값은 계산 도중에 음수와 양수값을 둘 다 가질 수 있다는 점이다.

참고로 d값은 그냥 mod 연산이기에 절대 음수일 수 없다.

그럼 가장 간단하고 직관적으로 이해할 수 있는 방식을 생각해본다면 그냥 우리가 따로 음수 표현을 가지도록 만들어주고 계산해주면 된다. int inNx0, inNx1, isNt는 0, 1의 값만 가지는 변수로 각각 uint64_t의 x0, x1, t 값의 음수 표현을 도와주는 변수다. 0이면 양수, 1이면 해당 변수가 음수임을 나타낸다. 기본적으로 알고리즘의 진행 방식은 mul_inv와 같다. 딱 하나 다른점은 x값의 계산에서 isNx0, isNx1 값을 검사해 계산 방식을 다르게 해주면 된다.

d 계산은 이전 mul_inv와 같지만, x계산은 분기가 나뉘는 것을 볼 수 있다.

각 분기의 의미는 이러하다.

두 부호가 다르다면 - 연산이 아니라 + 연산을 해주고 isNx1 = isNx0으로 맞춰준다.

만약 isNx0 = 0, isNx1 = 1 일 경우 원래 연산은 - 이고 -의 -는 +이므로 +를 해준 다음 isNx1 은

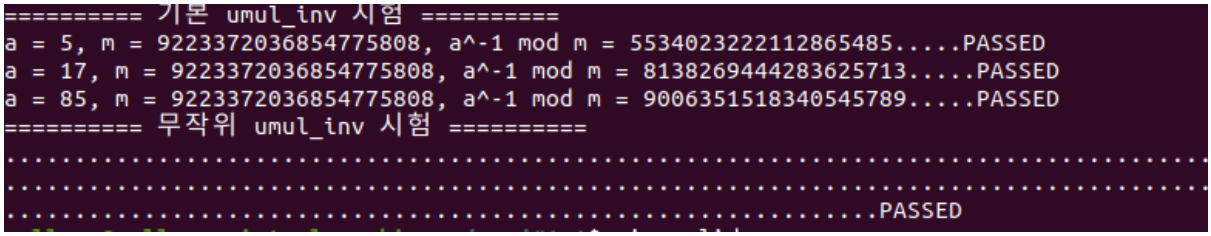
양수가 된다.

만약 isNx0 = 1, isNx1 = 0일 경우 원래 연산은 - 이고 이는 두 값을 더해준 다음 음수로 만들어 주는 것과 같기에 더해준 후 isNx1 은 음수가 된다.

두 부호가 같다면 두 값을 비교해줘야 한다.

x0가 더 큰 값이면 기존 연산과 같고 x1은 무조건 양수이며, q * x1이 더 크다면 반대로 계산해주고 x1은 무조건 음수가 나온다.

이렇게 분기만 나눠서 계산 처리를 해준다면 크게 어렵지 않게 함수를 완성할 수 있다.



제대로 PASSED 처리를 받는 것을 확인할 수 있다.

5. uint64_t gf16_mul(uint64_t a, uint64_t b);

```
132 uint16_t gf16_mul(uint16_t a, uint16_t b)
133 {
134     uint16_t r = 0;
135     while (b > 0) {
136         if (b & 1) r = r ^ a;
137         b = b >> 1;
138         a = ((a<<1) ^ ((a>>15) & 1 ? 0x2B : 0));
139     }
140     return r;
141 }
142
```

GF(2^n) 내에서의 연산은 mod 2 즉 XOR 연산과 같다. 우리는 곱셈을 구해야 하므로 shift 연산과 XOR 연산을 적절히 활용해야 한다. 또한 15차식의 곱셈의 결과를 16차식으로 나눈 나머지를 계산하고 위의 특성을 이용해야 한다.

임의로

b = x^4 + x + 1 = 10011

a = x^4 + 1 = 10001

이라고 해보자. a * b는 이렇게 계산될 것이다.

a * b = (x^4 + 1) * (x^4 + x + 1) = a * x^4 + a * 0 + a * 0 + a * x + a * 1

이후 값들은 mod 2로 계산되니 XOR로 덧셈을 진행해준다. 이를 오른쪽부터 잘 보면 a에 계속해서 x를 곱해주는데, b의 비트가 0이면 더해주지 않고, 부호가 1이면 더해주는 방식이다. 위 알고리즘도 이를 이용하고 mod 계산은 a에 x를 곱해줄 때 적용해준다. 이게 가능한 이유는 mod에 분배 법칙이 허용돼 최종값에 mod를 해줄 필요 없이 중간 과정에 mod를 해줘도 되기 때문이다. 또한 위 특성을 적용해서 mod 계산은 x^5 + x^3 + x + 1 = 0010 1011 = 0x2B를 적용해준다.

코드의 진행 과정을 간략하게 설명하자면

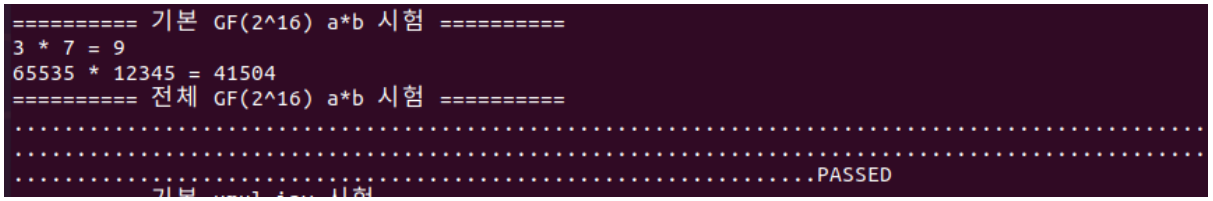
먼저 b의 마지막 비트가 1인지 0인지를 확인하고, 1이면 a를 r에 XOR 해준다. 최초값은 상수일 때를 말하고 이후부터는 x, x^2, x^3 ..... 가 될 것이다. 이후에는 b의 비트를 오른쪽으로 한칸 밀어 다음 항을 보도록 한다.

이제 a에 x를 곱해주고(왼쪽 shift) 만약 15차항 계수가 1이면(15비트 오른쪽으로 민 것이 1이라면) mod 계산을 진행해준다. 위 과정을 반복하면 계산이 완료된다.

6. uint64_t gf16_pow(uint64_t a, uint64_t b);

```
149 uint16_t gf16_pow(uint16_t a, uint16_t b)
150 {
151     uint16_t r = 1;
152     while (b > 0) {
153         if (b & 1) r = gf16_mul(r, a);
154         b = b >> 1;
155         a = gf16_mul(a,a);
156     }
157     return r;
158 }
159
```

위의 gf16_mul과, multiplication 알고리즘을 적절히 조합해 해당 함수를 구현해보자. 단순히 multiplication 알고리즘의 기본 모형에 곱셈 부분만 gf16_mul을 적용하였다. gf16_mul에서 어차피 mod 계산이 되었기에 따로 계산을 진행하지는 않았다.



3 * 7 = 0000 0000 0000 0011 * 0000 0000 0000 0111

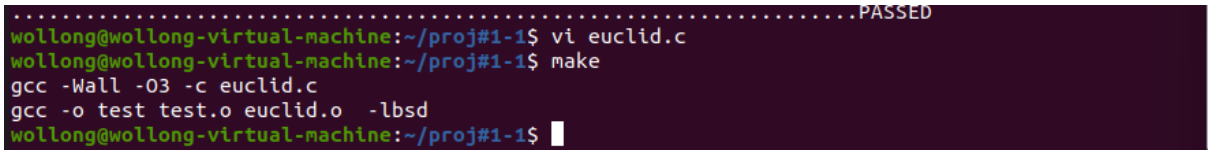
= (x + 1) * (x^2 + x + 1)

= 1110 ^ 0111

= 1001

= 9를 확인하니 제대로 나오는 것 같다.

7. 컴파일



make를 통해 컴파일을 진행해주고 실행을 해보면?

8. 결과

```
wollong@wollong-virtual-machine:~/proj#1-1$ ./test
========== 기본 gcd 시험 ==========
gcd(28,0) = 28
gcd(0,32) = 32
gcd(41370,22386) = 42
gcd(22386,41371) = 1
========== 기본 xgcd, mul_inv 시험 ==========
42 = 41370 * -204 + 22386 * 377
41370^-1 mod 22386 = 0, 22386^-1 mod 41370 = 0
1 = 41371 * 4285 + 22386 * -7919
41371^-1 mod 22386 = 4285, 22386^-1 mod 41371 = 33452
========== 무작위 xgcd, mul_inv 시험 ==========
................................................................................
................................................................................
.....................................................................PASSED
========== 기본 GF(2^16) a*b 시험 ==========
3 * 7 = 9
65535 * 12345 = 41504
========== 전체 GF(2^16) a*b 시험 ==========
................................................................................
................................................................................
.....................................................................PASSED
========== 기본 umul_inv 시험 ==========
a = 5, m = 9223372036854775808, a^-1 mod m = 5534023222112865485.....PASSED
a = 17, m = 9223372036854775808, a^-1 mod m = 8138269444283625713.....PASSED
a = 85, m = 9223372036854775808, a^-1 mod m = 9006351518340545789.....PASSED
========== 무작위 umul_inv 시험 ==========
................................................................................
................................................................................
.....................................................................PASSED
wollong@wollong-virtual-machine:~/proj#1-1$
```

이렇게 전체 결과가 잘 나오는 것을 확인할 수 있다.

```
  1 ========== 기본 gcd 시험 ==========
  2 gcd(28,0) = 28
  3 gcd(0,32) = 32
  4 gcd(41370,22386) = 42
  5 gcd(22386,41371) = 1
  6 ========== 기본 xgcd, mul_inv 시험 ==========
  7 42 = 41370 * -204 + 22386 * 377
  8 41370^-1 mod 22386 = 0, 22386^-1 mod 41370 = 0
  9 1 = 41371 * 4285 + 22386 * -7919
 10 41371^-1 mod 22386 = 4285, 22386^-1 mod 41371 = 33452
 11 ========== 무작위 xgcd, mul_inv 시험 ==========

 12 ................................................................................    ................
 ................................................................    ........................................
 ........................PASSED
 13 ========== 기본 GF(2^16) a*b 시험 ==========
 14 3 * 7 = 9
 15 65535 * 12345 = 41504
 16 ========== 전체 GF(2^16) a*b 시험 ==========

 17 ................................................................................    ................
 ................................................................    ........................................
 ........................PASSED
 18 ========== 기본 umul_inv 시험 ==========
 19 a = 5, m = 9223372036854775808, a^-1 mod m = 5534023222112865485.....PASSED
 20 a = 17, m = 9223372036854775808, a^-1 mod m = 8138269444283625713.....PASSED
 21 a = 85, m = 9223372036854775808, a^-1 mod m = 9006351518340545789.....PASSED
 22 ========== 무작위 umul_inv 시험 ==========

 23 ................................................................................    ................
 ................................................................    ........................................
 ........................PASSED
```

과제의 예상 결과와도 부합하는 것을 확인할 수 있다.

9. 느낀점

이번 과제를 진행하면서 쉬운 부분도 분명히 있었지만 어려웠던 부분도 많았다. 그건 내가 비트 마스킹 프로그래밍과, unsigned 자료형을 거의 해본적이 없어서라고 판단되는데, 이번 기회에 익숙해 질 수 있는 기회가 된 것 같고

개인적으로는 umul_inv 함수를 저렇게 flag 변수를 세우지 않고서도 가능할 것 같은데 아직까지는 머릿속에서 그려지지 않는 다는 점이 아쉬웠다. 다른 u 함수들 처럼 비트 계산으로만 해결하는 방법은 없을지 생각해봐야겠다고 느꼈다.

재밌었던 점은 GF에서의 다항식 계산 부분이었다. 수업을 들으면서도 신기하다고 생각하고 있었는데, 직접 과제로 구현해보니 처음에는 코드가 직관적으로 보이지 않아 힘들었지만 한번 깨닫고 나니 술술 읽혔다.