Project #4

2018044893 이수정

<함수 설명>

- **static uint64_t gcd(uint64_t a, uint64_t b)** - review_project#1

```c
static uint64_t gcd(uint64_t a, uint64_t b)
{
    uint64_t tmp;
    while (b != 0)
    {
            tmp = a % b;
            a = b;
            b = tmp;
    }
        return a;//b == 0 일 때 a값이 최대공약수
}
```

gcd(a,b) = gcd(b,a%b) = gcd(a%b,b%(a%b)) = … 을 만족하고 b == 0일 때 a가 최대공약수가 되므로 a가 return되도록 구현하였다.

- **static uint64_t mul_inv(uint64_t a, uint64_t m)** - review_project#1

```c
static uint64_t mul_inv(uint64_t a, uint64_t m)
{
    uint64_t d0 = a, d1 = m;
    uint64_t x0 = 1, x1 = 0;
    uint64_t q, tmp;
    int sign = -1;

    while (d1 > 1)
    {
        q = d0 / d1;

        tmp = d0 - q * d1; // d(n+2) = d(n) - d(n)/d(n+1) * d(n+1)
        d0 = d1;
        d1 = tmp;

        tmp = x0 + q * x1; // x(n+2) = x(n) - d(n)/d(n+1) * d(n+1)
        x0 = x1;
        x1 = tmp;

        sign = ~sign; //x1값이 while문을 돌 때마다 부호가 바뀜 unsign int를 고려함.
    }
    if (d1 == 1)
        return (sign ? m - x1 : x1); // sign변수를 통해 부호 결정.
    else
        return 0;
}
```

① ② ③

유클리드 알고리즘 gcd(a,b) = gcd(b,a%b) = gcd(a%b,b%(a%b)) 에서 a = d0, b = d1, a%b = d2, b%(a%b) = d3라고 하자.

이때 $d_2 = d_0 - (d_0/ d_1) * d_1$, $d_3 = d_1 - (d_1/ d_2) * d_2$ 라 할수있다.

일반화 하면 $d_n/ d_{n+1} = q_n$ … ①,

$d_{n+2} = d_n - q_n * d_{n+1}$, $x_{n+2} = x_n - q_n * x_{n+1}$, $y_{n+2} = y_n - q_n * y_{n+1}$ 이다.

이를 통해 d_{n+1} == 0 일때 gcd(a,b) = d_n = a * x_n + b * y_n임을 알 수 있다.

gcd(a,b) == 1을 만족할 때 1 = a * x_n + b * y_n각항을 modulo b를 한다면 a^{-1}mod b = x_n이다.

a^{-1}mod m 은 x_n 을 계산하면 됨을 알 수있다 … ②.

x_0 = 1, x_1 = 0, x_2 = (양수) – 0, x_3 = 0 – (양수), x_4 = (양수) – (음수), x_5 = (음수) – (양수)

즉 홀수 번째 x에 대해 0또는 음수 짝수 번째 x에 대해 양수를 가짐을 알 수 있다.

unsigned int의 특성을 고려해 sign변수이용하여 음수 양수를 판별한다. … ③

- **static uint64_t mod_add(uint64_t a, uint64_t b, uint64_t m)** - review_project#3

```
55 static uint64_t mod_add(uint64_t a, uint64_t b, uint64_t m)
56 {
57     a = a % m; //a modulo m
58     b = b % m; //b modulo m
59     return (a >= (m-b) ? a-(m-b) : a + b); // (a + b) >= m ? a + b - m : a + b)
60 }
61
```

a (mod m) → a = a % m;, b (mod m) → b = b % m;으로 미리 처리해준다

이때 0 ≤ a < m이고, 0 ≤ b < m이므로 0 ≤ a + b < 2m인 것을 알 수 있다. ... ①

a + b ≥ m 라면 ①에 의해 m ≤ a + b < 2m → 0 ≤ a + b - m < m이므로 a + b - m을 return 하면 되고,... ②

a + b < m 라면 ①에 따라 0 ≤ a + b < m이므로 a + b를 return하면 된다.

② 에서 a + b는 m의 값에 따라 overflow가 발생할 수 있기 때문에 a + b ≥ m → a ≥ m – b을 이용하고

a + b – m → a – (m - b)으로 연산하면 overflow를 방지할 수 있다

- **static uint64_t mod_mul(uint64_t a, uint64_t b, uint64_t m)** - review_project#1

```
62 static uint64_t mod_mul(uint64_t a, uint64_t b, uint64_t m)
63 {
64     uint64_t r = 0;//
65     while (b > 0)
66     {
67         if (b & 1)
68             r = mod_add(r, a, m); //b의 비트가 1이라면 결과에 a를 더해줌.
69         b = b >> 1;
70         a = mod_add(a, a, m); // binary b << 1 에 해당하는 a값은 a + a 에 해당함
71     }
72     return r;
73 }
```

double addition 알고리즘을 이용한다. 이를 설명하자면 먼저 a * b = a + a + ... + a (a를 b번 더한 것)임을 인지한다.

b를 binary로 나타낼 때 b = 10010이라고 한다면 b = 2^4 + 2^1이므로 a를 2^4 번 더한 후 a를 2^1번 더한 것을 알 수 있다.

a를 2^1번 더한 것은 a를 2^0번 더한 것을 두 번 더한 것이고 a를 2^4 번 더한 것은 a를 2^3번 더한 것을 두 번 더한 것이므로 a = mod_add(a, a, m)을 해준 후 b에 해당하는 비트가 1일경우 이를 더해준다.

- **static uint64_t mod_pow(uint64_t a, uint64_t b, uint64_t m)** - review_project#1

```
75 static uint64_t mod_pow(uint64_t a, uint64_t b, uint64_t m)
76 {
77     uint64_t r = 1;
78     while (b > 0)
79     {
80         if (b & 1)
81             r = mod_mul(r, a, m); //b의 비트가 1일때 r에 a를 곱해줌.
82         b = b >> 1;
83         a = mod_mul(a, a, m); //binary b << 1 에 해당하는 a값은 a * a에 해당함.
84     }
85     return r;
86 }
```

square multiplication 알고리즘을 이용한다. 마찬가지로 a^b= a * a * ... * a (a를 b번 곱한 것)임을 인지한다.

b를 binary로 나타낼 때 b = 10110이라고 한다면 b = 2^4 + 2^2 + 2^1이므로 a를 2^4 번 곱하고 2^2 번 곱한 후 a를 2^1번 곱한 것을 알 수 있다.

a를 2^1번 곱한 것은 a를 2^0번 곱한 것의 제곱이고 a를 2^2번 곱한 것은 a를 2^1번 곱한 것의 제곱이다. 마찬가지로 a를 2^4 번 곱한 것은 a를 2^3번 곱한 것을 제곱한 것이므로 a = mod_mul(a, a, m)을 해준 후 b에 해당하는 비트가 1일경우 이를 곱해준다면 a^bmod m에 해당하는 값을 구할 수 있다.

- **static int miller_rabin(uint64_t n)** – review_project#3

```
14 const uint64_t a[ALEN] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
```

ALEN = 12

```
88 static int miller_rabin(uint64_t n)
89 {
90     uint64_t q, k, tmp;
91     int is_prime = 0;
92     int index = 0;
93
94     if (n == 2 || n == 3) return PRIME;// 2는 짝수지만 prime, 3은 while값에 들어갈 수 없으므로 미리 예외처>
   리한다.
95     k = 0;
96     q = n - 1;
97     while (q % 2 == 0)
98     {
99         q = q / 2;
100        k++;
101    } //q,k구하기
102    while (a[index] < (n-1) && index < 12)
103    {
104        is_prime = COMPOSITE; //소수 판별 초기화
105        if (mod_pow(a[index], q, n) == 1)
106            is_prime = PRIME;
107        else
108        {
109            for (int j = 0; j < k; j++)
110            {
111                if (j == 0)
112                    tmp = mod_pow(a[index], q, n); //a^q mod n 저장
113                else
114                    tmp = mod_mul(tmp, tmp, n); //q(t+1) = q(t)*q(t) mod n
115                if (tmp == n - 1)
116                    is_prime = PRIME;
117            }
118        }
119        if (is_prime == COMPOSITE)
120            return COMPOSITE;//확정적인 합성수임.
121        index++;
122    }
123    return is_prime;//while문을 모두 통과하면 99.99...%확률로 prime임.
124 }
```

① n = 2인 경우 Miller Rabin의 테스트조건 n이 홀수에 해당하지 않으므로 예외처리를 한다.
n = 3인 경우 k = 1, q = 1이며 ③ 에 대한 결과는 2 mod 3 = 1로 해당하지 않고, ④에 대한 결과는 1 mod 3 = 2로 해당하지 않는다. 또한 while문의 조건에 해당하지 않기 때문에 예외처리를 하였다.

② (n – 1) = 2^k * q (k>0, q는 홀수)를 만족해야 한다.
Prime n은 2를 제외하면 모두 홀수이기 때문에 n - 1이 짝수임을 알 수 있다.
n – 1는 2를 k번 곱하고 홀수 q를 곱한 값이기 때문에 2로 나눈 나머지가 1이 되기 전까지 k번 나누어 주면 q가 남게 된다.

③ if (a^q mod n = 1) then return (“inconclusive”)에 해당한다.
n이 소수(prime)이고 a가 정수일 때 $a^n \equiv a \bmod n$, $a^{n-1} \equiv 1 \bmod n$ 를 만족한다.

④ if($a^{q*(2^j)}$ mod n = n-1) then return (“inconclusive”)에 해당한다.
0보다 큰 k와 홀수 q라는 조건을 이용하여 n이 소수(prime)라면 양의 정수 a에 대해 $a^n \equiv a \bmod n$, $a^{n-1} \equiv 1 \bmod n$ 를 만족한다.
n - 1 = q * 2^k 이므로 $a^{q*(2^k)} \equiv 1 \bmod n$ 을 만족하고 $(a^{q*(2^{k-1})})^2 \equiv 1 \bmod n$이므로 $a^{q*(2^{k-1})} \equiv -1 \bmod n$ ⇔ ($a^{q*(2^j)}$ mod n = n-1) 에 해당한다.
for loop에서 C(j) = $a^{q*(2^j)}$ 라고 하면 C(j+1) = $a^{q*(2^{j+1})}$= $a^{q*(2^j)}$ * $a^{q*(2^j)}$ 이므로 C(j+1) = C(j) *C(j)임을 확인할 수 있다. 초기값 j = 0일 때 값을 tmp에 초기화 시키고 mod_pow를 통해 매번 새롭게 연산하는 것보다 이전의 C(j)값을 이용하여 값을 누적시키면 속도가 빨라진다

- **void mRSA_generate_key(uint64_t *e, uint64_t *d, uint64_t *n)**

```
126 void mRSA_generate_key(uint64_t *e, uint64_t *d, uint64_t *n)
127 {
128     uint64_t p, q, tmp, theta;
129     uint64_t fbit = 1;
130
131
132   ① fbit = (fbit<<31) + 1;//p와 q가 확정적으로 32비트로 만들기 위함
133     tmp = 0;//p*q값 초기화
134
135   ② while(tmp < MINIMUM_N){//n은 ninimun_n보다 커야 64비트를 만족함.
136       ③ p = fbit | arc4random(); //32th = 1, 1st = 1 고정, fit or arc4random()
137         while (miller_rabin(p) == COMPOSITE)    p = fbit | arc4random();// prime p 찾기
138         q = fbit | arc4random(); //32th = 1, 1st = 1 고정, fit or arc4random()
139         while (miller_rabin(q) == COMPOSITE)    q = fbit | arc4random();// prime q 찾기
140         tmp = p * q;
141     }
142     *n = tmp;
143
144   ① theta = (p-1) * (q-1)/gcd(p-1,q-1);//계산의 양을 줄이기 위해 lcm 최소공약수 이용
145
146     tmp = arc4random_uniform(theta);//theta보다 작은 e값 선정
147   ② while(gcd(theta,tmp) != 1 || tmp <= 1)  tmp = arc4random_uniform(theta);//e값의 조건을 만족할때까지.
148     *e = tmp;
149   ③ *d = mul_inv(*e,theta);
150 }
```

1. Large prime인 p와q를 선택하고 n = p * q 으로 계산한다.

   ① p와q는 길이가 비슷할수록 large prime이므로 computing하기 어렵다.
   arc4random()는 최대 32bit의 random #를 생성하므로 만들 수 있는 p, q인 arc4random() $<2^{32}$ 이므로 최소한 $2^{31}<$ p와 $2^{31}<$ q를 만족해야 MINIMUN_N 이상의 n값이 생성된다.

   P와 q의 32번째 비트를 1로 만들기 위해 fbit = 2^{31}을 만들어준다.
   large prime인 p와 q는 항상 홀수이므로 1번째 비트를 1로 만들기위해 fbit + 1을 해준다.

   ② n은 MINIMUM_N인 64bit(2^{64})이상을 만족해야 하므로 tmp(n값) ≥ MINIMUM_N(2^{63})을 만족할 때까지 while문을 돌린다.

   ③ Arc4random()을 통해 최대 32비트 random #를 생성해주고 fbit와 or 연산(fbit or arc4rnadom())을 하여 항상 $2^{31}<$ p와 $2^{31}<$ q 를 만족하고 홀수인 random #를 완성한다.
   이를 miller_rabin()을 통해 PRIME아 나올 때 까지 반복한다.

2.

   ① 오일러 function theta을 대신해 Carmichael's totient function ramda(최소공배수)를 이용하여 계산의 양을 줄여준다.
   ramda = lcm(p - 1, q – 1) = (p - 1) * (q - 1) / gcd(p - 1, q - 1)

   ② 2 < e < ramda, gcd(ramda, e) == 1을 만족할 때까지 random # e를 선택한다.

   ③ e * d ≡ 1 mod ramda ⇔ d ≡ e^{-1}mod ramda이므로 d = mul_inv(e, ramda)를 통해 구한다.

- **int mRSA_cipher(uint64_t *m, uint64_t k, uint64_t n)**

```
164 int mRSA_cipher(uint64_t *m, uint64_t k, uint64_t n)
165 {
166     *m = mod_pow(*m, k, n);// m^k mod n
167     if(*m >= n) return 1;// data >= n then error
168     else return 0; // success
169 }
```

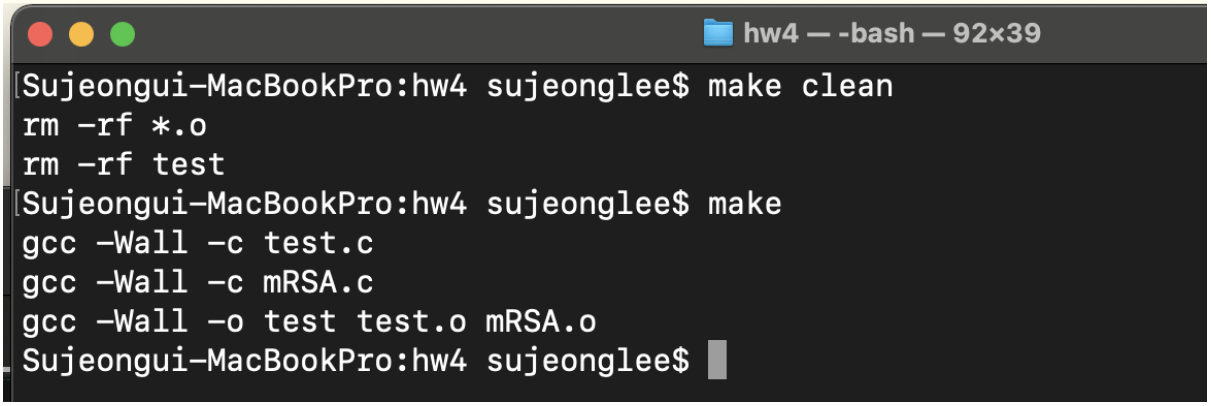
Message M을 암호화하여 sender은 PU (public key) = {e, n}를 이용하여 C(암호문) = M^emod n 이때 (0 ≤ M < n) 조건을 만족할 때 암호화가 가능하다. 마찬가지로 암호문C를 복호화 할 때 PR(private key) = {d, n}을 이용하여 M

= C^dmod n 이다.

※ 메시지 M은 항상 modulo n보다 크면 메시지가 소실, 변형되므로 항상 0 ≤ M < n를 만족하지 않을 때 error를 return하고 0 ≤ M < n를 만족할 때 success를 return한다.

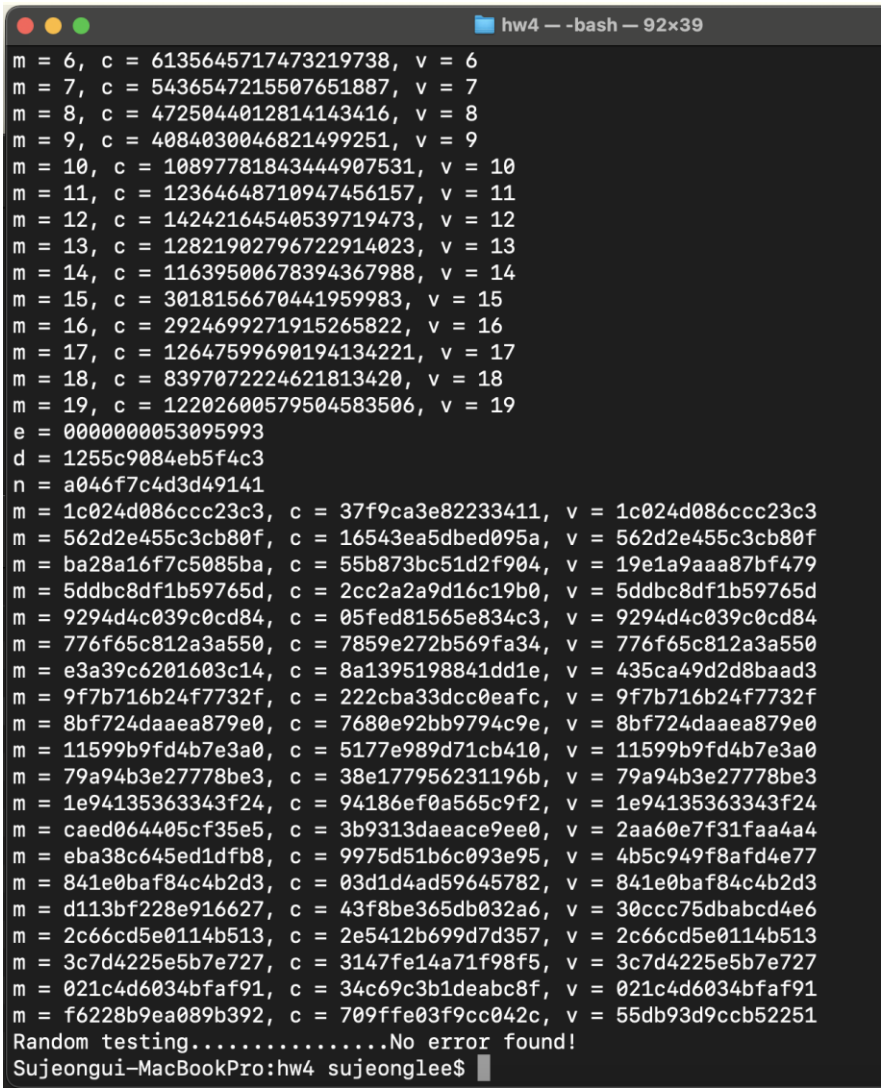
<컴파일 과정>

환경 : macOS Monterey 12.0.1



문제없이 실행됨을 확인하였다.

<실행 결과물>



No error found!를 확인하였다.

<소감/어려웠던 점>

1. 이전 project에서 이용한 함수들 update

mul_inv(uint64_t a, uint64_t m) - sign변수 이용

update 전

```
111 uint64_t umul_inv(uint64_t a, uint64_t m) // unsigned 64 비트 정수에서 mul_inv와 같은 방식을 사용하여 a^-1 mo
    d m을 구함
112 {
113     uint64_t d0,d1,x0,x1,q,tmp4;
114     d0 = a;
115     d1 = m;
116     x0 = 1;
117     x1 = 0;
118     while(d1>1)
119     {
120         q = d0/d1;
121 
122         tmp4 = d0 - q*d1;
123         d0 = d1;
124         d1 = tmp4;
125 
126         tmp4 = x0 - q*x1;
127         x0 = x1;
128         x1 = tmp4;
129     }
130     if(d1 == 1) return((x1>>63)==0 ? x1 : x1+m);//첫번째 비트를 통해 부호를 확인함.
131     else    return 0;
132 
133     // 여기를 완성하세요
134 }
```

update 후

```
 28 static uint64_t mul_inv(uint64_t a, uint64_t m)
 29 {
 30     uint64_t d0 = a, d1 = m;
 31     uint64_t x0 = 1, x1 = 0;
 32     uint64_t q, tmp;
 33     int sign = -1;
 34 
 35     while (d1 > 1)
 36     {
 37         q = d0 / d1;
 38 
 39         tmp = d0 - q * d1; // d(n+2) = d(n) - d(n)/d(n+1) * d(n+1)
 40         d0 = d1;
 41         d1 = tmp;
 42 
 43         tmp = x0 + q * x1; // x(n+2) = x(n) - d(n)/d(n+1) * d(n+1)
 44         x0 = x1;
 45         x1 = tmp;
 46 
 47         sign = ~sign; //x1값이 while문을 돌 때마다 부호가 바뀜 unsign int를 고려함.
 48     }
 49     if (d1 == 1)
 50         return (sign ? m - x1 : x1); // sign변수를 통해 부호 결정.
 51     else
 52         return 0;
 53 }
```

x1의 첫번째 비트를 확인하여 양수 음수를 확인하는 방법 대신 x1의 부호가 매 while문마다 바뀐다는 특성을 통하여 sign변수를 이용하였다. → project#1제출 후 수업시간 전체 피드백 내용

2. arc4random 이용

man arc4random을 통해 활용방법에 대해 고민해보았다.

```
SYNOPSIS
     #include <stdlib.h>
     (See libbsd(7) for include usage.)

     uint32_t
     arc4random(void);

     void
     arc4random_buf(void *buf, size_t nbytes);

     uint32_t
     arc4random_uniform(uint32_t upper_bound);

     void
     arc4random_stir(void);

     void
     arc4random_addrandom(unsigned char *dat, int datlen);

DESCRIPTION
     This family of functions provides higher quality data than those described in
     rand(3), random(3), and rand48(3).

     Use of these functions is encouraged for almost all random number consumption
     because the other interfaces are deficient in either quality, portability, stan -
     dardization, or availability.  These functions can be called in almost all cod -
     ing environments, including pthreads(3) and chroot(2).
```

```
157     while(*n<MINIMUM_N){
158         arc4random_buf(&p,4);
159         while (miller_rabin(p) != PRIME)
160         {
161             arc4random_buf(&p,4);
162         }
163         arc4random_buf(&q,4);
164         while (miller_rabin(q) != PRIME)
165         {
166             arc4random_buf(&q,4);
167         }
168         *n = p * q;
169     }
Random testingError: your RSA key generation may be wrong.
Sujeongui-MacBookPro:hw4 sujeonglee$
```

```
137     while(tmp < MINIMUM_N){
138         p = 0x80000000 + arc4random_uniform(0x20000000)*2 + 1;
139         while (miller_rabin(p) == COMPOSITE)
140         {
141             p = 0x80000000 + arc4random_uniform(0x20000000)*2 + 1;
142
143
144         //  printf("%llu\n",p >> 31);
145         }
146         q = 0x80000000 + arc4random_uniform(0x20000000)*2 + 1;
147         while (miller_rabin(q) == COMPOSITE)    q =0x80000000 + arc4random_uniform(0x20000000)*2 + 1;
148         tmp = p * q;
149     }
150     *n = tmp;
Random testing.....No error found!
it takes 236342768.0
Sujeongui-MacBookPro:hw4 sujeonglee$
```

arc4random_buf()와, arc4random_uniform()으로 고민해 보았고 arc4random_buf()는 포인터를 이용해야 했기 때문에 적합하지 않다고 판단하였다.

arc4random_uniform()은 32비트 위치를 1로 고정하고 최대 30비트까지 random #를 넣고 2배한 이후 1을 더해 홀수인 최솟값이 $2^{31}+1$인 random #을 만드는 방법을 생각했지만 Random testing속도가 너무 느려(약 10배) 다른 방법을 고민해 보았다. (속도에 관해서 뒤에서 설명)

3. 속도증진

- Miller_rabin함수
  공지에서 언급했듯이 prime p, q를 얼마나 빨리 찾느냐가 성능을 결정하는 최대 요인이므로 먼저 prime 임을 확인하는 miller_rabin함수의 속도가 중요하다.

if($a^{q*(2^j)}$ mod n = n-1) then return ("inconclusive")에 해당하는 부분code

```
109             for (int j = 0; j < k; j++)
110             {
111                 if (j == 0)
112                     tmp = mod_pow(a[index], q, n); //a^q mod n 저장
113                 else
114                     tmp = mod_mul(tmp, tmp, n); //q(t+1) = q(t)*q(t) mod n
115                 if (tmp == n - 1)
116                     is_prime = PRIME;
117             }
```

a^{q*2^j}를 연산할 때 j값이 증가할 때마다 2중 mul_pow연산인 a^{q*2^j}를 반복하지 않고 square multiplication 알고리즘을 이용하여 매번 값을 누적 시켜 pow연산에서 mul연산으로 대체하였다.

- arc4random()을 이용한 random # p, q생성
  정확한 처리속도를 확인하기 위해 clock함수를 추가하여 비교하였다.

```
135     while(tmp < MINIMUM_N){//n은 ninimun_n보다 커야 64비트를 만족함.
136         p = arc4random();
137         while (miller_rabin(p) == COMPOSITE)    p = arc4random();// prime p 찾기
138         q = arc4random();
139         while (miller_rabin(q) == COMPOSITE)    q = arc4random();// prime q 찾기
140         tmp = p * q;
141     }
142     *n = tmp;
```

```
Random testing.....................No error found!
it takes 117 sec
Sujeongui-MacBookPro:hw4 sujeonglee$
```

P와 q의 범위를 지정하지 않았을 때 p는 $p < 2^{31}$인 경우까지 포함하여 생성되는데, 이때 n값이 MINIMUM_N을 만족하는 p, q가 나올 가능성은 $p > 2^{31}$ 일때보다 절반이상 낮을것으로 예상하였고 117초가 걸린 것을 확인하였다.

```
137     while(tmp < MINIMUM_N){
138         p = 0x80000000 + arc4random_uniform(0x20000000)*2 + 1;
139         while (miller_rabin(p) == COMPOSITE)
140         {
141             p = 0x80000000 + arc4random_uniform(0x20000000)*2 + 1;
142
143
144         //  printf("%llu\n",p >> 31);
145         }
146         q = 0x80000000 + arc4random_uniform(0x20000000)*2 + 1;
147         while (miller_rabin(q) == COMPOSITE)    q =0x80000000 + arc4random_uniform(0x20000000)*2 + 1;
148         tmp = p * q;
149     }
150     *n = tmp;
```

```
Random testing.....................No error found!
it takes 241 sec
Sujeongui-MacBookPro:hw4 sujeonglee$
```

$p > 2^{31}$, $q > 2^{31}$으로 고정하였고 arc4random_uinform()을 이용하였을 때 짝수경우를 제외하였지만 p와 q를 정의하기까지 큰수의 덧셈 곱셈이 많아 241초로 오히려 속도가 느려짐을 확인하였다.ㄴ

```
132     uint64_t fbit = 1;
133
134
135     fbit = (fbit<<31);//p와 q가 확정적으로 32비트로 만들기 위함
136     tmp = 0;//p*q값 초기화
```

```
135     while(tmp < MINIMUM_N){//n은 ninimun_n보다 커야 64비트를 만족함.
136         p = fbit + (arc4random()>>1);
137         while (miller_rabin(p) == COMPOSITE)    p = fbit + (arc4random()>>1);
138         q = fbit + (arc4random()>>1);
139         while (miller_rabin(q) == COMPOSITE)    q = fbit + (arc4random()>>1);       tmp = p * q;
140     }
141     *n = tmp;
```

```
Random testing.....................No error found!
it takes 31 sec
Sujeongui-MacBookPro:hw4 sujeonglee$
```

32번째 비트값이 1인 fbit를 생성하고 32비트 이하의 random#를 생성한 뒤 31비트로 줄여주고 fbit와 합쳐주는 방법으로 p와 q를 생성하였을 때 31초로 시간이 감소하였다.

```
144     theta = (p-1) * (q-1);
```

```
147     theta = (p-1) * (q-1)/gcd(p-1,q-1);//계산의 양을 줄이기 위해 lcm 최소공약수 이용
```

```
Random testing.....................No error found!
it takes 30 sec
Sujeongui-MacBookPro:hw4 sujeonglee$
```

오일러 function theta을 대신해 Carmichael's totient function ramda(최소공배수)를 이용하여 계산의 양을 줄여준다 1초정도 줄어들었지만 random으로 생성되는 p와 q의 값에 따라 1초는 충분히 차이 날 수 있음을 인지하고 있다.

```
135     fbit = (fbit<<31) + 1;//p와 q가 확정적으로 32비트로 만들기 위함
136     tmp = 0;//p*q값 초기화
137
138     while(tmp < MINIMUM_N){//n은 ninimun_n보다 커야 64비트를 만족함.
139         p = fbit | (arc4random()>>1); //32th bit =  1, 2~31th bit = radnom, 1st bit = 1 고정.
140         while (miller_rabin(p) == COMPOSITE)    p = fbit | (arc4random()>>1);// prime p 찾기
141         q = fbit | (arc4random()>>1);//32th bit =  1, 2~31th bit = radnom, 1st bit = 1 고정.
142         while (miller_rabin(q) == COMPOSITE)    q = fbit | (arc4random()>>1);// prime q 찾기
143         tmp = p * q;
144     }
145     *n = tmp;
```

P와 q는 large prime으로 항상 **홀수** 이므로 fbit을 생성할 때 2^{31}대신$2^{31}+1$을 하여 p와 q가 짝수가 생성되는 경우를 제거해 주었고 (arc4random()>>1)의 값과 관련없이 항상 2^0에 해당하는 값이 1이여야 하기 때문에 | (or)연산을 이용하였다.

```
132     fbit = (fbit<<31) + 1;//p와 q가 확정적으로 32비트로 만들기 위함
133     tmp = 0;//p*q값 초기화
134
135     while(tmp < MINIMUM_N){//n은 ninimun_n보다 커야 64비트를 만족함.
136         p = fbit | arc4random(); //32th = 1, 1st = 1 고정, fit or arc4random()
137         while (miller_rabin(p) == COMPOSITE)    p = fbit | arc4random();// prime p 찾기
138         q = fbit | arc4random(); //32th = 1, 1st = 1 고정, fit or arc4random()
139         while (miller_rabin(q) == COMPOSITE)    q = fbit | arc4random();// prime q 찾기
140         tmp = p * q;
141     }
142     *n = tmp;
```

```
Random testing.....................No error found!
it takes 24 sec
Sujeongui-MacBookPro:hw4 sujeonglee$
```

Fbit과 |연산을 이용하면 arc4random()값과 상관없이 2^{31}값이 1이므로 (arc4rabdin()>>1)하는 과정이 필요없기 때문에 fbit | (arc4rabdin()>>1) → fbit | arc4rabdin()로 수정하였다. 최종 결과 24초까지 시간을 줄였다.

2^{31}를 고정하지 않았을 때 (117초) → 2^{31}고정 (31초) → lamda사용 (30초) → 짝수생성 제외(24초)

Project#4를 통해 project#1과 project#3의 코드를 다시 돌아보고 속도 증진과 오류 최소화를 위해 수정해 보았고 arc4random()함수에 대해 알아보는 기회가 되었다. 2^{31}을 고정하기 위한 시도를 해 보았고 비트연산을 통해 고정하는 방법을 이용하였다. 이를 이해하는 과정에서 더 쉽게 홀수값을 만드는 아이디어를 생각할 수 있었다.