Project#3

2018044893 이수정

<함수설명>

mod.c file

- **uint_64t mod_add(uint64_t a, uint64_t b, uint64_t m)**

```c
uint64_t mod_add(uint64_t a, uint64_t b, uint64_t m)
{
        a = a % m;//a mod m
        b = b % m;// b mod m
        return (a >= m-b ? a - (m - b) : (a + b));// a + b >= m 이면 a + b - m, a + b < m이면 a + b리턴
// 여기를 완성하세요
}
```

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.

- **uint_64t mod_sub(uint_64t a, uint_64t b, uint_64t m)**

```c
uint64_t mod_sub(uint64_t a, uint64_t b, uint64_t m)
{
        a = a % m;// a mod m
        b = b % m;// b mod m
        return(a < b ? (a + (m - b)) : a - b); // a - b < 0이면  a - b + m, a - b >= 0 이면 a - b리턴
// 여기를 완성하세요
}
```

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

이때 0 <= a < m이고, 0 <= b < m이므로 -m < a - b < m인 것을 알 수 있다. … ③

a – b < 0이면 ③에 따라 -m < a - b< 0 → 0< a – b + m < m이므로 모듈로 성질에 따라 a – b + m를 return하고

a - b >= 0이면 ③에 따라 0 <= a -b < m이므로 a - b를 그대로 return해주면 된다.

- **uint_64t mod_mul(uint_64t a, uint_64t b, uint_64t m)**

```c
uint64_t mod_mul(uint64_t a, uint64_t b, uint64_t m)
{
        uint64_t r = 0;
        while(b>0){
                if(b & 1)
                        r = mod_add(r,a,m);
                b = b >> 1;
                a = mod_add(a,a,m);
        }
        return r;
// 여기를 완성하세요
}
```

a * b = a + a + … + a (b번) → a * b는 a를 b번 더한것이다.

예를 들어 $a=1001$, $b=1111$ 라고 한다면

$b=1111 = 2^3+2^2+2^1+2^0$ 이므로

$$a\times b = a(2^3+2^2+2^1+2^0)$$
$$= \underbrace{a\cdot 2^3}_{a+a+a+a+a+a+a+a} + \underbrace{a\cdot 2^2}_{a+a+a+a} + \underbrace{a\cdot 2^1}_{a+a} + \underbrace{a\cdot 2^0}_{a}$$ 이라고 할 수 있다

$a\cdot 2^2+a\cdot 2^2 \bmod m$ $\quad$ $a\cdot 2^1+a\cdot 2^1 \bmod$ $\quad$ $a\cdot 2^0+a\cdot 2^0 \bmod m$

→ a=mod_add(a,a,m); 에 해당한다.

(b&1) 가 아닌경우 0 이고 해당하는 경우 2^n 이므로 이때만 result r 값에 연산된 a를 mod_add 해준다.

- **uint_64t mod_pow(uint_64t a, uint_64t b, uint_64t m)**

```
uint64_t mod_pow(uint64_t a, uint64_t b, uint64_t m)
{
        uint64_t r = 1;
        while(b > 0){
                if(b & 1)
                        r = mod_mul(r,a,m);
                b = b >> 1;
                a = mod_mul(a,a,m);
        }
        return r;
// 여기를 완성하세요
}
```

a ^ b = a * a * a * ... * a(b번) → a를b번 곱한것이다.

예를 들어 $a=1001$, $b=1101$ 라고 하면

$b= 2^3\times 1 + 2^2\times 1 + 2^1\times 0 + 2^0\times 1$ 이므로

$$a^b = a^{(2^3+2^2+0+2^0)}$$
$$= a^{2^3} \times a^{2^2} \times a^0 \times 2^{2^0}$$

$a^{2^2}\times a^{2^2}\times 1$ $\quad$ $a^{2^1}\times a^{2^1}\times 1$ $\quad$ $a^{2^0}\times a^{2^0}\times 0$ $\quad$ $2^{2^0}\times 1$

즉 mod_mul과 마찬가지로 if(b&1)인경우 a값을 곱해주고 매번 a=a*a; → a=mod_mul(a,a,m)을 해준다.

miller_rabin.c file

- **int miller_rabin(uint_64t n)**

```c
int miller_rabin(uint64_t n)
{
        uint64_t k,q,index,is_prime;
        q = n-1;
        k = 0;
        while(q%2==0){//q값이 홀수가 될때까지 나누어줌
                q = q/2;
                k++;
        }

        uint64_t bin_j;
        index = 0;
        if(n == a[0] || n == a[1])      return PRIME; // 2, 3인경우 while문에 들어가지 않음
        while(a[index] < n-1 && index<12){
                is_prime = COMPOSITE; //is_prime초기화
                if(mod_pow(a[index],q,n) == 1)  is_prime = PRIME;
                else{
                        for(uint64_t j = 0; j < k; j++)
                        {
                                if(j == 0)      bin_j = mod_pow(a[index],q,n);
                                else    bin_j = mod_mul(bin_j,bin_j,n);

                                if(bin_j == n-1)        is_prime = PRIME;
                        }
                }
                if(is_prime == COMPOSITE)       return is_prime;
                index++;
        }
        return is_prime;//모든 a값에 대해 PRIME으로 연산되는 경우만 PRIME이라고 확신할 수 있다.

// 여기를 완성하세요
}
```

a[index] < n-1에 해당하는 경우에 miller_rabin연산을 하므로 배열 모두를 사용하는 for 문보다는 while문이 적합하다고 생각하였다.

①. (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.

② 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.

③ if (a^q mod n = 1) then return (“inconclusive”)에 해당한다.

④ if(a^q*(2^j) mod n = n-1) then return (“inconclusive”)에 해당한다.
(수정 전) 2^j연산을 간단하게 하기 위해서 위의 mod_mul과 mod_pow에서 이용한 bit 연산을 사용하였다. 2^j 를 이진값 bin_j라고 정의하고 j가 증가하는 만큼 비트를 (bin_j << j)이동시켜 굳이 2^j를 연산할 필요가 없다
(수정 후) 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 bin_j 에 초기화 시키고 mod_pow를 통해 매번 새롭게 연산하는 것보다 이전의 C(j)값을 이용하여 값을 누적시키면 속도가 빨라진다.

<컴파일과정>

환경 : ubuntu-20.04.3-desktop-amd64(리눅스)

컴파일 결과

```
      |                  ~~~^                            ~
      |                     |                            |
      |                     long long unsigned int       uint64_t {aka long unsigned int}
      |                  %lu
test.c:46:23: warning: format ‘%llu’ expects argument of type ‘long long unsigned int’, but argument 3 has type ‘u
int64_t’ {aka ‘long unsigned int’} [-Wformat=]
   46 |     printf("%llu ^ %llu mod %llu = %llu\n", a, b, m, mod_pow(a,b,m));
      |                         ~~~^                            ~
      |                            |                            |
      |                            long long unsigned int       uint64_t {aka long unsigned int}
      |                         %lu
test.c:46:32: warning: format ‘%llu’ expects argument of type ‘long long unsigned int’, but argument 4 has type ‘u
int64_t’ {aka ‘long unsigned int’} [-Wformat=]
   46 |     printf("%llu ^ %llu mod %llu = %llu\n", a, b, m, mod_pow(a,b,m));
      |                                  ~~~^                  ~
      |                                     |                  |
      |                                     |                  uint64_t {aka long unsigned int}
      |                                     long long unsigned int
      |                                  %lu
test.c:46:39: warning: format ‘%llu’ expects argument of type ‘long long unsigned int’, but argument 5 has type ‘u
int64_t’ {aka ‘long unsigned int’} [-Wformat=]
   46 |     printf("%llu ^ %llu mod %llu = %llu\n", a, b, m, mod_pow(a,b,m));
      |                                         ~~~^                  ~~~~~~~~~~~~~~
      |                                            |                  |
      |                                            |                  uint64_t {aka long unsigned int}
      |                                            long long unsigned int
      |                                         %lu
test.c:55:24: warning: format ‘%llu’ expects argument of type ‘long long unsigned int’, but argument 2 has type ‘u
int64_t’ {aka ‘long unsigned int’} [-Wformat=]
   55 |             printf("%llu ", x);
      |                     ~~~^    ~
      |                        |    |
      |                        |    uint64_t {aka long unsigned int}
      |                        long long unsigned int
      |                     %lu
test.c:70:24: warning: format ‘%llu’ expects argument of type ‘long long unsigned int’, but argument 2 has type ‘u
int64_t’ {aka ‘long unsigned int’} [-Wformat=]
   70 |             printf("%llu ", x);
      |                     ~~~^    ~
      |                        |    |
      |                        |    uint64_t {aka long unsigned int}
      |                        long long unsigned int
      |                     %lu
gcc -Wall -c miller_rabin.c
gcc -Wall -c mod.c
gcc -Wall -o test test.o miller_rabin.o mod.o
soo@soo-VirtualBox:~/Documents/proj3$
```

리눅스 환경에서 %llu를 사용할 때 정의 된 변수가 64비트를 모두 사용하지 않고 32비트로 커버가 가능한 경우 %lu를 이용하도록 warning이 생긴다.

이 경우 %lu로 바꾸어 주거나 64비트 사용 매크로를 정의하여 %llu대신 %"PRIu64"를 이용하면 warning문제를 해결할 수 있다.

%lu로 수정 후 컴파일 결과

```
soo@soo-VirtualBox: ~/Documents/proj3
soo@soo-VirtualBox:~/Documents/proj3$ make clean
rm -rf *.o
rm -rf test
soo@soo-VirtualBox:~/Documents/proj3$ make
gcc -Wall -c test.c
gcc -Wall -c miller_rabin.c
gcc -Wall -c mod.c
gcc -Wall -o test test.o miller_rabin.o mod.o
soo@soo-VirtualBox:~/Documents/proj3$
```

<실행 결과물>

soo@soo-VirtualBox: ~/Documents/proj3

```
gcc -Wall -o test test.o miller_rabin.o mod.o
soo@soo-VirtualBox:~/Documents/proj3$ ./test
<덧셈> 1234 + 5678 mod 3456 = 0
<뺄셈> 1234 - 5678 mod 3456 = 2468
<곱셈> 1234 * 5678 mod 3456 = 1340
<지수> 1234 ^ 5678 mod 3456 = 1792
---
<덧셈> 3684901700 + 3904801120 mod 4294901760 = 3294801060
<뺄셈> 3684901700 - 3904801120 mod 4294901760 = 4075002340
<곱셈> 3684901700 * 3904801120 mod 4294901760 = 2417663360
<지수> 3684901700 ^ 3904801120 mod 4294901760 = 1734737920
---
<덧셈> 18446744073709551360 + 18446744073709551598 mod 18441921395520346504 = 9645356378409950
<뺄셈> 18446744073709551360 - 18446744073709551598 mod 18441921395520346504 = 18441921395520346266
<곱셈> 18446744073709551360 * 18446744073709551598 mod 18441921395520346504 = 14923616227936587640
<지수> 18446744073709551360 ^ 18446744073709551598 mod 18441921395520346504 = 6550219153064247488
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
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541
547 557 563 569 571 577 587 593 599 601
607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733
739 743 751 757 761 769 773 787 797 809
811 821 823 827 829 839 853 857 859 863
877 881 883 887 907 911 919 929 937 941
947 953 967 971 977 983 991 997 1009 1013
1019 1021 1031 1033 1039 1049 1051 1061 1063 1069
1087 1091 1093 1097 1103 1109 1117 1123 1129 1151
1153 1163 1171 1181 1187 1193 1201 1213 1217 1223
1229 1231 1237 1249 1259 1277 1279 1283 1289 1291
1297 1301 1303 1307 1319 1321 1327 1361 1367 1373
1381 1399 1409 1423 1427 1429 1433 1439 1447 1451
1453 1459 1471 1481 1483 1487 1489 1493 1499 1511
1523 1531 1543 1549 1553 1559 1567 1571 1579 1583
1597 1601 1607 1609 1613 1619 1621 1627 1637 1657
1663 1667 1669 1693 1697 1699 1709 1721 1723 1733
1741 1747 1753 1759 1777 1783 1787 1789 1801 1811
1823 1831 1847 1861 1867 1871 1873 1877 1879 1889
1901 1907 1913 1931 1933 1949 1951 1973 1979 1987
1993 1997 1999 2003 2011 2017 2027 2029 2039 2053
```

TRUE	<덧셈> 1234 + 5678 mod 3456 = 0	<덧셈> 1234 + 5678 mod 3456 = 0
TRUE	<뺄셈> 1234 - 5678 mod 3456 = 2468	<뺄셈> 1234 - 5678 mod 3456 = 2468
TRUE	<곱셈> 1234 * 5678 mod 3456 = 1340	<곱셈> 1234 * 5678 mod 3456 = 1340
TRUE	<지수> 1234 ^ 5678 mod 3456 = 1792	<지수> 1234 ^ 5678 mod 3456 = 1792
TRUE	---	---
TRUE	<덧셈> 3684901700 + 3904801120 mod 4294901	<덧셈> 3684901700 + 3904801120 mod 4294901760 = 329480
TRUE	<뺄셈> 3684901700 - 3904801120 mod 4294901	<뺄셈> 3684901700 - 3904801120 mod 4294901760 = 407500
TRUE	<곱셈> 3684901700 * 3904801120 mod 4294901	<곱셈> 3684901700 * 3904801120 mod 4294901760 = 241766
TRUE	<지수> 3684901700 ^ 3904801120 mod 4294901	<지수> 3684901700 ^ 3904801120 mod 4294901760 = 173473
TRUE	---	---
TRUE	<덧셈> 18446744073709551360 + 184467440737	<덧셈> 18446744073709551360 + 18446744073709551598 mo
TRUE	<뺄셈> 18446744073709551360 - 184467440737	<뺄셈> 18446744073709551360 - 18446744073709551598 mo
TRUE	<곱셈> 18446744073709551360 * 184467440737	<곱셈> 18446744073709551360 * 18446744073709551598 mo
TRUE	<지수> 18446744073709551360 ^ 184467440737	<지수> 18446744073709551360 ^ 18446744073709551598 mo
TRUE	2 3 5 7 11 13 17 19 23 29	2 3 5 7 11 13 17 19 23 29
TRUE	31 37 41 43 47 53 59 61 67 71	31 37 41 43 47 53 59 61 67 71
TRUE	73 79 83 89 97 101 103 107 109 113	73 79 83 89 97 101 103 107 109 113
TRUE	127 131 137 139 149 151 157 163 167 173	127 131 137 139 149 151 157 163 167 173
TRUE	179 181 191 193 197 199 211 223 227 229	179 181 191 193 197 199 211 223 227 229
TRUE	233 239 241 251 257 263 269 271 277 281	233 239 241 251 257 263 269 271 277 281
TRUE	283 293 307 311 313 317 331 337 347 349	283 293 307 311 313 317 331 337 347 349
TRUE	353 359 367 373 379 383 389 397 401 409	353 359 367 373 379 383 389 397 401 409
TRUE	419 421 431 433 439 443 449 457 461 463	419 421 431 433 439 443 449 457 461 463
TRUE	467 479 487 491 499 503 509 521 523 541	467 479 487 491 499 503 509 521 523 541
TRUE	547 557 563 569 571 577 587 593 599 601	547 557 563 569 571 577 587 593 599 601
TRUE	607 613 617 619 631 641 643 647 653 659	607 613 617 619 631 641 643 647 653 659
TRUE	661 673 677 683 691 701 709 719 727 733	661 673 677 683 691 701 709 719 727 733
TRUE	739 743 751 757 761 769 773 787 797 809	739 743 751 757 761 769 773 787 797 809
TRUE	811 821 823 827 829 839 853 857 859 863	811 821 823 827 829 839 853 857 859 863

엑셀을 이용하여 모든 값이 예상출력-2.txt와 일치하는지 확인하였다.

<소감/어려웠던 점>

1. mod_add

```c
uint64_t mod_add(uint64_t a, uint64_t b, uint64_t m)
{
        a = a % m;
        b = b % m;
        return (a > m-b ? a - (m - b) : (a + b)%m);
// 여기를 완성하세요
}
```

덧셈에서 a+b = m일 때 return값이 m이 나와 (a+b)%m을 해주었다.(연산의 중복에 대해 불편함을 느낌)

```c
uint64_t mod_add(uint64_t a, uint64_t b, uint64_t m)
{
        a = a % m;
        b = b % m;
        return (a >= m-b ? a - (m - b) : a + b);
// 여기를 완성하세요
}
```

보고서를 작성하면서 값의 범위에 대해 다시 생각해 보게 되었는데 a + b = m인경우 (a + b) mod m = a + b - m 이므로 a > m – b 대신 a >= m – b 를 이용하고 (a + b)%m을 a + b로 변경하는 방법을 통해 문제를 해결하였다.

2. miller_rabin

19	TRUE	1087	1091	1087	1091	1093	1097	1103	1109	1117	1123	1129	1151
20	TRUE	1153	1163	1153	1163	1171	1181	1187	1193	1201	1213	1217	1223
21	TRUE	1229	1231	1229	1231	1237	1249	1259	1277	1279	1283	1289	1291
22	TRUE	1297	1301	1297	1301	1303	1307	1319	1321	1327	1361	1367	1373
23	TRUE	1381	1399	1381	1399	1409	1423	1427	1429	1433	1439	1447	1451
24	TRUE	1453	1459	1453	1459	1471	1481	1483	1487	1489	1493	1499	1511
25	TRUE	1523	1531	1523	1531	1543	1549	1553	1559	1567	1571	1579	1583
26	TRUE	1597	1601	1597	1601	1607	1609	1613	1619	1621	1627	1637	1657
27	TRUE	1663	1667	1663	1667	1669	1693	1697	1699	1709	1721	1723	1733
28	TRUE	1741	1747	1741	1747	1753	1759	1777	1783	1787	1789	1801	1811
29	TRUE	1823	1831	1823	1831	1847	1861	1867	1871	1873	1877	1879	1889
30	TRUE	1901	1907	1901	1907	1913	1931	1933	1949	1951	1973	1979	1987
31	FALSE	1993	1997	1993	1997	1999	2003	2011	2017	2027	2029	2039	2047
32	FALSE	2063	2069	2053	2063	2069	2081	2083	2087	2089	2099	2111	2113
33	FALSE	2131	2137	2129	2131	2137	2141	2143	2153	2161	2179	2203	2207
34	FALSE	2221	2237	2213	2221	2237	2239	2243	2251	2267	2269	2273	2281
35	FALSE	2293	2297	2287	2293	2297	2309	2311	2333	2339	2341	2347	2351
36	FALSE	2371	2377	2357	2371	2377	2381	2383	2389	2393	2399	2411	2417
37	FALSE	2437	2441	2423	2437	2441	2447	2459	2467	2473	2477	2503	2521
38	FALSE	2539	2543	2531	2539	2543	2549	2551	2557	2579	2591	2593	2609
39	FALSE	2621	2633	2617	2621	2633	2647	2657	2659	2663	2671	2677	2683
40	FALSE	2689	2693	2687	2689	2693	2699	2707	2711	2713	2719	2729	2731
41	FALSE	2749	2753	2741	2749	2753	2767	2777	2789	2791	2797	2801	2803
42	FALSE	2833	2837	2819	2833	2837	2843	2851	2857	2861	2879	2887	2897
43	FALSE	2909	2917	2903	2909	2917	2927	2939	2953	2957	2963	2969	2971
44	FALSE	3001	3011	2999	3001	3011	3019	3023	3037	3041	3049	3061	3067
45	FALSE	3083	3089	3079	3083	3089	3109	3119	3121	3137	3163	3167	3169

```
soo@soo-VirtualBox: ~/D
739 743 751 757 761 769 773 787 797 809
811 821 823 827 829 839 853 857 859 863
877 881 883 887 907 911 919 929 937 941
947 953 967 971 977 983 991 997 1009 1013
1019 1021 1031 1033 1039 1049 1051 1061 1063 1069
1087 1091 1093 1097 1103 1109 1117 1123 1129 1151
1153 1163 1171 1181 1187 1193 1201 1213 1217 1223
1229 1231 1237 1249 1259 1277 1279 1283 1289 1291
1297 1301 1303 1307 1319 1321 1327 1361 1367 1373
1381 1399 1409 1423 1427 1429 1433 1439 1447 1451
1453 1459 1471 1481 1483 1487 1489 1493 1499 1511
1523 1531 1543 1549 1553 1559 1567 1571 1579 1583
1597 1601 1607 1609 1613 1619 1621 1627 1637 1657
1663 1667 1669 1693 1697 1699 1709 1721 1723 1733
1741 1747 1753 1759 1777 1783 1787 1789 1801 1811
1823 1831 1847 1861 1867 1871 1873 1877 1879 1889
1901 1907 1913 1931 1933 1949 1951 1973 1979 1987
1993 1997 1999 2003 2011 2017 2027 2029 2039 2047
2053 2063 2069 2081 2083 2087 2089 2099 2111 2113
2129 2131 2137 2141 2143 2153 2161 2179 2203 2207
```

결과값 확인을 위하여 excel을 활용하여 확인해본 결과 2047이 prime으로 인식되는 것을 확인하였다.

```c
        uint64_t bin_j = 1;
        index = 0;
        if(n == a[0] || n == a[1])      return PRIME;
        while(a[index] < n-1 && index<12){
                is_prime = 0;
                if(a[index] == n)       return PRIME;
                if(mod_pow(a[index],q,n) == 1)  is_prime = 1;
                else{
                        for(uint64_t j = 0; j < k; j++){
                                if(mod_pow(a[index],q*(bin_j<<j),n) == n-1)     is_prime = 1;
                        }
                }
                if(is_prime == 0)       return is_prime;
                index++;
        }
        return is_prime;
```

if(a^q*(2^j) mod n = n-1) then return ("inconclusive")에 해당하는 부분에서 q*(2^j)를 modulo연산이 아닌 일반 곱셈을 이용하여 값의 오류가 생겼다.

```
        if(n == a[0] || n == a[1])       return PRIME;
        while(a[index] < n-1 && index<12){
                is_prime = 0;
                if(a[index] == n)        return PRIME;
                if(mod_pow(a[index],q,n) == 1)  is_prime = 1;
                else{
                        for(uint64_t j = 0; j < k; j++){
                                if(mod_pow(a[index],mod_mul(q,(bin_j<<j),n),n) == n-1)  is_prime = 1;
                        }
                }
                if(is_prime == 0)        return is_prime;
                index++;
```

q * (bin_j<<j)대신 mod_mul(q,(bin_j<<j),n)을 이용하여 문제를 해결하였다.

3. miller_rabin()의 a^q*(2^j) mod n = n-1 연산을 위해 pow를 두번 사용해야해 속도의 약점이 있다.

```
for(uint64_t j = 0; j < k; j++){
        if(mod_pow(a[index],mod_mul(q,(bin_j<<j),n),n) == n-1)   is_prime = PRIME;
}
```

2^j를 빠르게 연산하기위해 비트를 이용하는 bin_j << j 아이디어를 생각해내 a^q*(2^j) = a^q*(bin_j<<j) 를 하여 2^j의 연산은 빠르지만 a에 대한 제곱수는 해결하지 못했다.

```
for(uint64_t j = 0; j < k; j++)
{
        if(j == 0)      bin_j = mod_pow(a[index],q,n);
        else    bin_j = mod_mul(bin_j,bin_j,n);

        if(bin_j == n-1)        is_prime = PRIME;
}
```

(피드백 후) 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. 이를 이용하여 a에 대한 지수연산 대신 누적연산으로 수정하였다.

mod.c를 완성하면서 overflow에 유의하며 변수들을 처리하니 후의 miller_rabin연산에서 overflow가 발생하지 않는다는 확신이 생겼고 실제 overflow가 발생하지 않는 것을 확인할 수 있었다. 아직도 부족하지만 이전의 project보다 비트연산에 대한 이해도가 높아짐을 느낄 수 있었다.