Assignment - 3

This optional assignment contains additional implementation based exercises.

Quadratic Equations over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$

Introduction. For a prime p, the *field* $\mathbb{F}_p$ can be modelled as the set $\{0, 1, \dots p-1\}$, where all operations (addition and multiplication) are defined with respect to mod p.

The field $\mathbb{F}_{p^2}$ can be modelled as the set $\{x + iy : x, y \in \mathbb{F}_p\}$, where i denotes the square root of '-1' in $\mathbb{F}_p$ (i.e. $i^2 = -1$). Note that this can only be done if $p \equiv -1 \pmod 4$. (*Why?*)

$\mathbb{F}_{p^2}$ can be thought of as an extension to $\mathbb{F}_p$ just like $\mathbb{C}$ is an extension to $\mathbb{R}$. Hence, the operations in $\mathbb{F}_p$ are extended to form the operations in $\mathbb{F}_{p^2}$ in the exact fashion in which the operations in $\mathbb{R}$ are extended to form the operations in $\mathbb{C}$.

Now consider a quadratic equation $ax^2 + bx + c = 0$ in some field $\mathbb{F}$, where x is the unknown. Since, all operations (addition, multiplication, division, etc.) are allowed in a field, we can do the same manipulation as we do for $\mathbb{R}$ to get,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Note that apart from the already defined operations (addition, multiplication), there is a *square root* operation. As an example, finding the square root of n in $\mathbb{F}_p$, is equivalent to solving the congruence $z^2 \equiv n \pmod p$ for the unknown z. Note that the existence of a solution of this congruence can be easily tested using *Euler's Criterion*, but it doesn't provide a way to obtain this solution.

Task. Design a program that can solve any given quadratic equation over any field $\mathbb{F}_p$ or $\mathbb{F}_{q^2}$, where $q \equiv -1 \pmod 4$ and p are primes.

Resources. You can go through the paper "*Square root computation over even extension fields*"., which discusses various techniques and algorithms for computing square roots in general fields.

Montgomery's Optimization

Introduction. In most cryptosystems like *RSA* and *Diffie Hellman* which are based on number theoretic problems, require a huge number of *modulo* operations. The modulo operation $x \pmod n$ is equivalent to evaluating $\left(x - n\lfloor \frac{x}{n} \rfloor\right)$. Since, n can be very large, the division operation here can be very costly. You can see the computational complexity of division algorithms *here*.

Montgomery Reduction is a technique which transforms the numbers into a special form, called the *Montgomery Form*, which allows you to compute modular operations much more efficiently as compared to usual operations. As an example, in multiplication, the idea is that instead of dividing the product by subtracting n multiple times, it adds multiples of n to cancel out the lower bits and then just discards the lower bits. This is the basis of the *REDC* algorithm.

Task. Implement functions for Montgomery *Reduction*, *Addition*, and *Multiplication* (using REDC). Further use these elementary algorithms to implement functions for *Inverse* and *Binary Exponentiation*.

Resources. You can refer to *Wikipedia* for more details on Montgomery Optimizations, and find check the reference implementation *here* for completing the task.

Montgomery's Ladder

Introduction. In the usual algorithm for *Binary Exponentiation*, in every iteration, there is a *multiplication* operation in addition to a *squaring* operation, depending on the bits of the exponent. The bits 0 and 1 result in different number of operations and hence differ in power consumption. If this is implemented in a cryptosystem, for example while evaluating $g^a \pmod n$ in the *Diffie-Hellman* protocol, and an intruder gains access to the power consumption of the host system, he can compare the power required at every iteration of the binary exponentiation, and will be able to determine the bits of a due to the nature of this algorithm.

Task 1. Using a time library, implement an algorithm for *Binary Exponentiation* which takes as input three integers g, a, n, evaluates $g^a \pmod n$ and outputs an array T, where T_i is the time taken in the i^{th} iteration. Now, design an algorithm (which is to be run on the same computer and compiler as the earlier algorithm) which takes as input g, n, T and evaluates the secret key a.

Resources. The attack described above is a *timing attack*. You can learn more about such attacks (*Side Channel Attacks*) from these *slides*.

To prevent such attacks, we use *Montgomery's Ladder*, which is an algorithm very similar to *Binary Exponentiation*, but performs equal number of operations (squaring and multiplication) on each iteration.

Task 2. Prove and implement the *Montgomery's Ladder* technique for modular exponentiation.

Resources. The pseudocode for *Montgomery's Ladder* can be found on *Wikipedia*.

Prime Number Generator for RSA

Introduction. The RSA cryptosystem requires us to generate two large random primes p, q. A simple way to do this is by selecting large random numbers and then performing primality tests. This is done until we arrive at a positive primality test. But a deterministic primality test for large numbers could be very costly, and hence we usually use a series of low and high level probabilistic primality tests which are much more efficient.

Just to give an idea of the accuracy of such a probabilistic method, let us consider the *Miller-Rabin Test*. This comprises of multiple rounds of test. In each test, we choose a random a, called the *witness*, and check the following congruences for the proposed odd number $p = 2^t q + 1$, where $2 \nmid q$.

$$a^q \equiv 1 \pmod p \qquad \text{and} \qquad a^{2^s q} \equiv -1 \pmod p, \text{ for any } 0 \leq 0 < t$$

If any of the above conditions hold, then p is a probable prime, and if none of these conditions hold, then p is definitely composite. The probability that this test gives a false positive is at most $\left(\frac{1}{4}\right)$ for each round. The probabilities are independent for every round. Hence, the probability of a false positive after 50 rounds would be at most $\left(\frac{1}{2^{100}}\right)$. This is enough for practical uses.

Task 1. There is also a deterministic version of the *Miller Rabin*. This involves fixing the choice of witnesses, and gives deterministic results upto some number. Your task is to implement this version for numbers upto 10^{18}, and also discuss the computational complexity of this algorithm.

Task 2. Implement the prime number generator for the RSA algorithm, which takes an integer n as input, and outputs an $n-$bit integer p with high probability of being a prime.

Resources. You can also find a list of deterministic witnesses at end of this *Numberphile video* on *Miller Rabin Test*. *This* is a reference implementation for designing a *Prime Number Generator*.