EE 382V: Social Computing **Fall 2018**

Lecture 2: November 10

Lecturer: Vijay Garg *Scribe: Porter Perry*

2.1 Introduction

This lecture focuses on the underlining number theory, algorithms, and and mathematical utilities that form the basis for the RSA encryption and decryption system (crypto) for security and authentication.

2.2 Fermat's Little Theorem

First we explore *Fermat's Little Theorem* (FLT), which was stated in a letter dated in 1640 by the French mathematician Pierre de Fermat. This theorem is highly useful for simplifying the computation of exponents in modular arithmetic.

Theorem 2.1 (Fermat's Little Theorem) *For any prime number p, and any integer number a where $0 < a < p - 1$, then $a^{(p-1)} \equiv 1$ (mod p).*

Note: As explained during the lecture, the equivalence relational notation above is commonly used in academic mathematical writings. The modular equivalent relations are the same as applying the modulus operation on both sides of the equivalent equation. The above equivalence relation can be re-written as $a^{(p-1)} \pmod p = 1 \pmod p$.

2.2.1 FLT Examples

Ex 1. Let $p = 5$, and $a = 3$.

$$\begin{aligned} a^{(p-1)} \pmod p &= 3^{(5-1)} \pmod 5 \\ &= 3^4 \pmod 5 \\ &= 81 \pmod 5 \\ &= 1 \end{aligned}$$

Ex 2. Let $p = 7$, and $a = 4$.

$$\begin{aligned} a^{(p-1)} \pmod p &= 4^{(7-1)} \pmod 7 \\ &= 4^6 \pmod 7 \\ &= 4096 \pmod 7 \\ &= 1 \end{aligned}$$

2.3 Euler's Theorem

We now explore *Euler's Theorem*, written by the Swiss mathematician Leonhard Euler in 1736. This theorem is a generalization of Fermat's little theorem.

Theorem 2.2 (Euler's Theorem) *Let n be any number.*
Let a be relatively prime to n, where $gcd(a, n) = 1$ (the greatest common divisor between them is 1), then

$a^{\phi(n)} \equiv 1 (mod\ n)$

where $\phi(n)$ is Euler's totient function.

Definition 2.3 (Euler's totient function) $\phi(n) = |\{i_{1 \leq i \leq n} | gcd(i, n) = 1\}|$
Euler's totient function is the size of the set of positive integers, up to n, that are relatively prime to n. Said another way, it is the number of integers i in the range $1 \leq i \leq n$ for which the greatest common divisor $gcd(n, i)$ is equal to 1.

2.3.1 Euler's Totient Function Examples

Ex 1. Let $n = 5$.
$\phi(5) = |\{1, 2, 3, 4\}| = 4$

Ex 2. Let $n = 6$.
$\phi(6) = |\{1, 5\}| = 2$

Ex 3. Let $n = 7$.
$\phi(7) = |\{1, 2, 3, 4, 5, 6\}| = 6$

Ex 4. Let $n = 12$.
$\phi(12) = |\{1, 5, 7, 11\}| = 4$

Ex 5. Let $n = 13$.
$\phi(13) = |\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}| = 12$

2.3.2 Euler's Totient Function Properties

A couple interesting and useful properties of Euler's totient function that play a key role in the RSA cryptosystem are:

- Euler's totient function is a multiplicative, meaning that if two numbers are m and n are relatively prime, then $\phi(mn) = \phi(m)\phi(n)$.
- If a number p is prime, then $\phi(p) = (p - 1)$.

Claim 2.4 *Let p and q both be prime numbers, then $\phi(pq) = (p-1)(q-1)$*

2.3.3 Examples

Ex 1. Let $p = 2$ and $q = 3$.
$\phi(2 \times 3) = \phi(6) = (2-1)(3-1)$
$= (1)(2)$
$= 2$

This matches the result of $\phi(6)$ of Euler's Totient Function Ex. 2 in the previous section.

Ex 2. Let $p = 5$ and $q = 3$.
$\phi(5 \times 3) = \phi(15) = (5-1)(3-1)$
$= (4)(2)$
$= 8$
$= |\{1, 2, 4, 7, 8, 11, 13, 14\}| = 8$

2.4 Greatest Common Divisor

[From Wikipedia]
Euclid's algorithm is an efficient method for computing the greatest common divisor (GCD) of two numbers, the largest number that divides both of them without leaving a remainder.

The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if the larger number is replaced by its difference with the smaller number. For example, 21 is the GCD of 252 and 105 (as $252 = 21 \times 12$ and $105 = 21 \times 5$), and the same number 21 is also the GCD of 105 and $252105 = 147$. Since this replacement reduces the larger of the two numbers, repeating this process gives successively smaller pairs of numbers until the two numbers become equal. When that occurs, they are the GCD of the original two numbers. By reversing the steps, the GCD can be expressed as a sum of the two original numbers each multiplied by a positive or negative integer, e.g., $21 = 5 \times 105 + (2) \times 252$.

The *extended Euclidean algorithm* is an extension to the Euclidean algorithm, and computes, in addition to the greatest common divisor of integers a and b, also the coefficients of Bzout's identity, which are integers x and y such that

$ax + by = \gcd(a, b)$

The extended Euclidean algorithm is particularly useful when a and b are coprime. With that provision, x is the modular multiplicative inverse of a modulo b, and y is the modular multiplicative inverse of b modulo a. Similarly, the polynomial extended Euclidean algorithm allows one to compute the multiplicative inverse in algebraic field extensions and, in particular in finite fields of non prime order.

2.5 Large Prime Numbers

The generation of prime numbers is foundational to the RSA cryptosystem. It would be nice to have a fast algorithm to do this. This can be accomplished using by an algorithms that checks if a randomly generated number is prime or not.

In 2002, Agrawal, Kayal, and Saxena published the paper titled "PRIMES is in P" in which they presented a deterministic algorithm that determines if any given general number is prime or composite. While this algorithm is of huge academic importance, it is not typical used in practice in favor of other faster options.

The *Prime Number Theorem* describes limiting distribution of of prime numbers, which become less common as they become larger, and formally quantifies the rate at which they occur.

Theorem 2.5 (Prime Number Theorem) *Let n be any positive number.*
Let $\pi(n)$ be the prime counting function that gives the number of primes less than or equal to n. The function $n/\ln(n)$ is a good approximation for $\pi(n)$ in that the limit of the ratio of the two functions is equal to one as n increases without bound:

$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln(n)} = 1$

This means that for a large value of n, the probability that a randomly generated number not greater than n is prime is very close to $1/\ln(n)$.

The above probabilistic statement of the prime number theorem is used to determine approximately how many random numbers will need to be generated and tested before a prime number will likely be found. For example, to find a 64-digit random prime number, a total of $\ln 10^{64} \approx 148$ randomly generated numbers will be tested before a prime number is found. The quantity of numbers tested can be reduced by half if only odd random numbers are tested, which for this example would be $(\ln 10^{64})/2 \approx 74$.

2.6 Fast Computation of Modulus Exponents

Consider the example of computing $7^{1024} \bmod 5$. The direct method involves first calculating 7^{1024} by multiplying 7 by itself 1024 times, then taking the result modulo 5. A more efficient way to solve the above is to take advantage of the following modular multiplication rule:

$a^2 \bmod n = (a \times a) \bmod n = ((a \bmod n) \times (a \bmod n)) \bmod n$

We can use the method of repeated squaring to calculate the answer more efficiently:

$7 \bmod 5 = 2$
$7^2 \bmod 5 = (2 \times 2) \bmod 5 = 4$
$7^4 \bmod 5 = (4 \times 4) \bmod 5 = 16 \bmod 5 = 1$
$7^8 \bmod 5 = (1 \times 1) \bmod 5 = 1$
$7^{16} \bmod 5 = (1 \times 1) \bmod 5 = 1$
...
$7^{512} \bmod 5 = (1 \times 1) \bmod 5 = 1$
$7^{1024} \bmod 5 = (1 \times 1) \bmod 5 = 1$

The above example happened to have an exponent that was conveniently a power of 2. What if it was desired to solve 7^{1025}? Combining the repeated squaring result with the modular multiplication rule, we get the following:

$7^{1025} \bmod 5 = ((7^{1024} \times 7^1) \bmod 5 = (7^{1024} \bmod 5) \times (7^1 \bmod 5)) \bmod 5 = (1 \times 2) \bmod 5 = 2$

2.6.1 Repeated Squaring and Multiplication Algorithm

The exponential $a^e \pmod{n}$ can be solved using the following iterative algorithm:

```
Step 1:  Let e_k e_{k-1} ... e_1 e_0 be the binary representation of e.
Step 2:  Set the variable c to 1
Step 3:  Repeat steps 3a and 3b for i=k, k-1, ..., 1, 0
     Step 3a:  c := c^2 mod n
     Step 3b:  if (e_i) = 1, then c := (a * c) mod n
```

Alternatively, this can also be implemented as a recursive algorithm:

```
Function exp(a, e, n)
    if (e is 1) return a
    else
        if (e is even)
            return (exp(a, e/2, n)^2 mod n)
        else
            return (a * exp(a, e/2, n)^2 mod n)
```