COMPSCI 4CR3 Assignment 3

Dev Mody

November 2024

Contents

1 Question 1

Let $a, m > 0$ be n-bit integers such that $gcd(a, m) = 1$

1.1 a)

How can we use Euler's Theorem to compute the inverse of a modulo m?

Euler's Theorem helps with Primality Testing in Public-Key Cryptography like the RSA Cryptosystem and is a generalization to Fermat's Little Theorem to any integer moduli. Euler's Theorem states

Euler's Theorem: Let a and m be integers with $\gcd(a, m) = 1$. Then $a^{\Phi(m)} \equiv 1 \pmod{m}$.

where **Euler's Phi Function** is defined as the number of integers in $\mathbb{Z}$ relatively prime to m ($\Phi(m) \in \mathbb{Z}_m$).

To compute the inverse of $a \pmod{m}$, we must also rely on the fact that the modular inverse of $a \pmod{m}$ is a number x such that $a \times x \equiv 1 \pmod{m}$. Using Euler's Theorem, we now know that: $a^{\Phi(m)} \equiv 1 \pmod{m}$. Now, $a^{\Phi(m)}$ can be expressed as $a \times a^{\Phi(m)-1}$. As a result, we know that $a^{\Phi(m)-1} \times a \equiv a^{\Phi(m)} \equiv 1 \pmod{m}$. Since we have found $a^{\Phi(m)-1} \times a \equiv 1 \pmod{m}$, the inverse of $a \pmod{m}$ is $x \equiv a^{\Phi(m)-1} \pmod{m}$.

1.2 b)

Show that we can also compute a^{-1} mod m using the Extended Euclidean Algorithm

The Extended Euclidean Algorithm (EEA) is good for finding the Great Common Divisor (GCD) of two integers r_0 and r_1 by computing the linear combination of the form $gcd(r_0, r_1) = s \times r_0 + t \times r_1$ for $s, t \in \mathbb{Z}$. The Extended Euclidean Algorithm is as follows:

Extended Euclidean Algorithm (EEA)

Input: positive integers r_0 and r_1 with $r_0 > r_1$

Output: $\gcd(r_0, r_1)$, as well as s and t such that $\gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$.

```
1. Initialization: $s_0 = 1$, $t_0 = 0$, $s_1 = 0$, $t_1 = 1$, $i = 1$
2. Algorithm:
   DO
      • $i = i + 1$
      • $r_i \equiv r_{i-2} \mod r_{i-1}$
      • $q_{i-1} = \frac{r_{i-2} - r_i}{r_{i-1}}$
      • $s_i = s_{i-2} - q_{i-1} \cdot s_{i-1}$
      • $t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$
   WHILE $r_i \neq 0$
3. Return $\gcd(r_0, r_1) = r_{i-1}, s = s_{i-1}, t = t_{i-1}$
```

To show that we can compute the modular inverse $a^{-1} \bmod m$ using EEA, we need to show that if a and m are relatively prime ($gcd(a, m) = 1$), EEA yields an $x \in \mathbb{Z}_m$ such that $a \times x \equiv 1 \pmod{m}$.

From above, we know that EEA finds the GCD of any two integers a and m but also finds $x, y \in \mathbb{Z}$ such that the Bezout Identity $a \times x + m \times y = gcd(a, m)$ holds.

For the modular inverse $a^{-1} \bmod m$ to exist, a and m must be coprime meaning that $a \times x + m \times y = gcd(a, m) = 1$.

Since $a \times x + m \times y = 1$, we can derive the statement $a \times x + m \times y \equiv 1 \pmod{m}$. Since $m \times y$ is divisible by m, we can disregard it altogether to derive $a \times x \equiv 1 \pmod{m}$. This shows that x is a solution to $a \times x \equiv 1 \pmod{m}$, meaning that $a^{-1} \equiv x \pmod{m}$.

1.3 c)

Implement your algorithms in a) and b) and run the algorithms on the following on the inputs from the document. What is a^{-1} mod m? Which algorithm is faster? What are the running times of the algorithms in milliseconds?

Given the two inputs a and m, I have determined that the modular inverse $a^{-1} \pmod{m}$ is

45584983244865491814527164519876268048460912115594852771815511636010737640
3525283213075123888664381192832794777034012202625978250778619258394374862217.

The Extended Euclidean Algorithm was MUCH faster than the Euler's Theorem Implementation. The run-time for Euler's Function was around `0.6549358367919922` milliseconds and the run-time using EEA was approximately `0.08392333984375` milliseconds. In this case, since our value of m was prime, we know that $\Phi(m)$, which is the number of values from 1 to $m-1$ that are coprime with m, is equal to $m-1$. This prevents the need of factorizing the large number m and makes that point of computation very convenient. However, the only point where Euler's Function can be costly in terms of computation is the shear amount of exponentiation required to compute the inverse. I implemented Square and Multiply, but it still managed to get this particular run-time.

2 Question 2

Recall that in class we discussed that, in an RSA public key (n, e), the public exponent e is often chosen to be a small value to speed up the encryption process. Now, consider the following RSA Public Key:

$$n = 508281196310201376192554864656699346831575429768465482788715190735760361687281737746563113895010157 \tag{1}$$

$$e = 7 \tag{2}$$

2.1 a)

Alice wants to encrypt the plaintext $x = 38745745356349$. Explain why, if Trudy knows that the plaintext x is small enough (as it is in the case), she can recover x.

In RSA Encryption, given a public key $k_{pub} = (n, e)$ and plaintext x, the sender of a message can encrypt the message into a ciphertext $y = e_{k_{pub}}(x) \equiv x^e \bmod n$. In this case, say that the plaintext x is significantly smaller than the number n. We know that since the $k_{pub} = (n, e)$ is public, Trudy has access to this information as well. If x is significantly smaller than n, then so is x^e. This means that when we compute $x^e \bmod n$, we're essentially recovering x^e. In other words, we know that if $x^e < n$, $x^e \bmod n = x^e$. As a result, we know that the ciphertext that Trudy has access to is $y = x^e$. Given this information and the public key information, Trudy can compute the e^{th} of y to recover the original plaintext x. In this case, since $e = 7$, the original plaintext can be recovered via $\sqrt[7]{y}$. Modern mathematical software can easily compute such roots, also given from the fact that the plaintext x is small.

2.2 b)

Find the smallest plaintext corresponding to ciphertext:

$$y = 4066488477440339689911514138508998613662966982287543919056006242924596335364286584668317646217011 \tag{3}$$

Given the strategy from part `a)`, since we know $k_{pub} = (n, e)$ where $e = 7$ and the encrypted ciphertext y, finding the smallest plaintext x that corresponds to y involves computing the 7th root of y as $x = \sqrt[7]{y}$. As a result, by performing this operation, we obtain the smallest plaintext $x = 63287374328731$

3 Question 3

Suppose there is a `key-gen` server that generates RSA public keys. Bob queires the server and receives a key pair $((n, e_1), d_1)$, where n is the modulus and e_1, d_1 are the public and private exponents, respectively. Alex also queries the server for a public key. By chance (albeit with low probability), he receives the key pair $((n, e_2), d_2)$, where the modulus n is the same as Bob's. Assume that $gcd(e_1, e_2) = 1$. Now, Alice wants to send a message to her friends, including both Alex and Bob. Explain how Trudy, having access to the public keys and the ciphertexts sent ot Bob and Alex, can recover Alice's original message.

Before explaining this process, let's understand what information is known for Trudy:

1. Trudy knows both of the public keys $k_1 = (n, e_1)$ and $k_2 = (n, e_2)$
2. Trudy has access to the two ciphertexts passed over: y_1 and y_2
3. e_1 and e_2 are coprime given that $gcd(e_1, e_2) = 1$

Since the same modulus n has been used for both Alex's and Bob's public keys, it is definitely possible to recover original plaintexts from the corresponding ciphertexts that are publically sent over. Here is a step-by-step explanation of how Trudy can recover these messages:

1. Alice encrypts the message using both public keys $k_1 = (n, e_1)$ and $k_2 = (n, e_2)$. Let the original plaintext be x, which results in the following ciphertexts to be generated: $y_1 = x^{e_1} \bmod n$ and $y_2 = x^{e_2} \bmod n$.
2. Now, Trudy intercepts y_1 and y_2 alongside $k_1 = (n, e_1)$ and $k_2 = (n, e_2)$. Since we know that e_1 and e_2 are coprime, there exists two integers a and b such that $a \times e_1 + b \times e_2 = gcd(e_1, e_2) = 1$. These values of a and b can be found using the Extended Euclidean Algorithm.
3. Now, given that Trudy has found such a and b, he can rewrite x in terms of y_1 and y_2 using a and b in the following way: $x = x^1 = x^{a \times e_1 + b \times e_2} = (x^{e_1})^a \times (x^{e_2})^b \bmod n$. Since $x^{e_1} \equiv y_1 \bmod n$ and $x^{e_2} \equiv y_2 \bmod n$, we get $x \equiv y_1^a \times y_2^b \bmod n$.
4. As a result, given that Trudy has access to e_1, e_2, a, b, y_1, and y_2, he can compute the original plaintext x as $y_1^a \times y_2^b \bmod n$. This works because the equation $a \times e_1 + b \times e_2 = 1$ is the opposite of encryption, isolating x modulo n.