Ceng471 HW1 Report

Ali Görkem Yalçın - 230201026

November 23, 2019

Introduction

I made use of the BigInteger class of Java in the homework.

Euclid's Extended Algorithm

In Euclid's Extended Algorithm, we can find the greatest common divisor of two numbers, using the GCD, we can find the multiplicative modular inverse and the relatively prime condition in an efficient way.

To calculate the GCD of the two integers, I used Euclid's Extended Algorithm. In short the algorithm calculates the GCD of two integers namely p and q, it also calculates the coefficients of the p and q:

$xp + yq = gcd(p, q)$

In this algorithm, the useful part is where the two integers are relatively prime. If they are then the x is the modular multiplicative inverse of p modulo q, y is the modular multiplicative inverse of q modulo p.
Idea is to start with the basic GCD algorithm and work our way towards our goal, until we can satisfy the linear equation, finding the GCD.
In our homework we weren't asked to the find the x and y but the GCD of the two integers. So I used the Extended Euclidian Algorithm to find the GCD of the two integers.

It checks if the q is zero, if so we return the other integer since gcd(x,0) = x.
If not we assign r as the p mod q to find the remainder of the two integers. Then now we have no need for p anymore, so we make the p equal to q and q equal to r to increase the depth of our solution.
Then we basically create the linear equation for the Extended Euclidian Algorithm to find the x and y values. But since were not asked to find the x and y values but the GCD of the two integers, I still used the Extended Euclidian Algorithm but altered it so it will return the GCD value.

To find the modular multiplicative inverse in the Euclid's way I checked if the GCD of two values is one. If not we can say that there is no inverse can be found.
If so, then we can say that x*p + modulus*y = 1
Then by taking the modulo modulus on both sides we get
x*p + modulus*y = 1 (mod modulus)
x*p = 1 (mod modulus)
Then this x is the multiplicative inverse of p.
I used the Euclid Algorithm to assume that they are coprime values.
Then in every loop where p>1
q = p mod modulus
t = modulus
then I updated the modulus as number mod modulus since we need to go deeper.
Then I updated the number value as the old modulus value to not loose the old modulus.
Then using the Euclidian Algorithm, starting from 0-1 I defined 2 integers(y,x) and in every loop I reassigned them values
y = x -(q*y)
x = t
I used x,y to keep track of the x and y values we use in the linear equation.
Then after the number equals to one, we get out of the loop and if the x value is negative, we make it positive by adding it the value of modulus, then we return the x value, the modulus multiplicative inverse of the number p.

To check relatively prime condition for two integers, I used the same GCD function
If the GCD of two integers are one, they are relatively prime, otherwise not.

Fermat's Little Theorem

In Fermat's Little Theorem, we can check if a number is prime(this is a possibility if we want to make the program run faster) or not.
Theorem says this:
if p is a prime number, then for any integer a, the number
$(a^p) = a(mod p)$

So in my implementation, I check for any random integer between 0 and the number I am checking if that random value to the power of the actual number modulus the random number:
$random^N = random(mod N)$
or not. We need to check N - 0 times to make sure the N is prime or not since if any random integer does not hold the equation, then the number is not prime. But since we are working with very big numbers I asked the user how many times do you want to check the number(in the parameter of the method), then

the method tests that many times, using random integers from 0 to N. If in any iteration says N is not prime, method exists the loop and returns false. If for every iteration, there is no abnormality, then when we exit the loop, method returns true.

Fast Modular Exponentiation

In fast modular exponentiation, I first checked if the modulus is 1, If so I returned 0.
If not I assigned the result as one and for 0 to the exponent(power) value I multiplied the result by the base number then immediately took the results mod modulus. By doing this, numbers did not get big resulting in a much faster calculation.

AES

In AES implementation, I made use of the javax.crypto library. Since AES works in 3 different key lengths, I chose 128 bits to encrypt the plaintext. If the user does not want to create her/his own SecretKey, he/she can use the generateKey method to create a key and then use that in both encryption and decryption of the plain/cipher text.

After finding the key, in encryption process, I convert the plainText to a char array, then get a AES Cipher object with key value as the secretKey user entered to the method I encrypt the plainText using the Base64.Encoder then return the cipherText.

For decrypting, method requires the cipherText and the secretKey. In decryption I use Base64.Decoder and AES Cipher in decryption mode to decrypt the cipherText and return the plainText.

Additional Notes

In the operations, users can change the parameters of the methods to calculate their own values.
I included a main method in the classes to ease the testing of the program.