CT255 Assignment 4

Diffie-Hellman Key Exchange

Problem 1 and 2 Source Code

```
import java.util.*;
import java.lang.Math;

public class CT255_DiffieHellman {

    public static void main(String[] args) {
        int primeNumber = randomPrimeNumber();
        int primitiveRoot = primitiveRoot(primeNumber);
        System.out.println("-----Diffie-Hellman Parameters-----");
        System.out.println("Random prime number generated: "+primeNumber);
        System.out.println("Random primitive root of "+primeNumber+" is
"+primitiveRoot);
        System.out.println();

        System.out.println("-----Emulating a Key Exchange Between Two Parties-----");

        //Each user needs a secret number (private key), argument X.
        //X < prime number
        Random randNum = new Random();
        //Random private number is generated for each user in the range (0-prime-1)
        int XA = randNum.nextInt(primeNumber);
        int XB = randNum.nextInt(primeNumber);
        //Using the private number, a public value is calculated and sent to the other
user
        //The genMethod returns a 64 bit variable to avoid overflow, this variable is
cast back to a 32-bit integer
        int YA = (int)genMethod(primitiveRoot, XA, primeNumber);
        int YB = (int)genMethod(primitiveRoot, XB, primeNumber);
        //Each user shares the public value, upon receiving the value, each user
calculates K by raising the
        //public value to their private value to the modulo of the shared prime number
        //Each user will calculate the same value for K.
        int K1 = (int)genMethod(YB, XA, primeNumber);
        int K2 = (int)genMethod(YA, XB, primeNumber);

        System.out.println("Alice private key: "+XA);
        System.out.println("Bob private key: "+XB);
        System.out.println();
        System.out.println("Alice's public key: " + YA);
        System.out.println("Bob's public key: " + YB);
        System.out.println();

        //Each user will calculate the same value for K.
        if(K1 == K2){
            System.out.println("Same keys calculated from both parties. "+K1+" = "+K2);
            System.out.println("Key calculated by Alice "+K1);
            System.out.println("Key calculated by Bob "+K2);
        }

        System.out.println();

        System.out.println("-----Man In The Middle Attack-----");
        //Alice's private key
        XA = randNum.nextInt(primeNumber);
        //Bob's private key
        XB = randNum.nextInt(primeNumber);
        //Mallory's private key
        int XC = randNum.nextInt(primeNumber);
```

```
        //Alice's public key
        YA = (int)genMethod(primitiveRoot, XA, primeNumber);
        //Bob's public key
        YB = (int)genMethod(primitiveRoot, XB, primeNumber);
        //Mallory's public key
        int YC = (int)genMethod(primitiveRoot, XC, primeNumber);

        //Mallory sends YC to both parties unknown to either Bob or Alice.
        //Alice sends YA to Bob, but is intercepted by Mallory and Alice receives YC.
        //Bob sends YB to Alice, but is intercepted by Mallory and Bob receives YC.
        System.out.println("Alice sends "+YA+" to Bob");
        System.out.println("Bob sends "+YB+" to Alice");
        System.out.println("Mallory intercepts both keys and sends "+YC+" to both
parties unbeknownst to them");
        System.out.println("Alice \"receives\" "+YC+" from Bob");
        System.out.println("Bob \"receives\" "+YC+" from Alice");

        //Alice calculates K using the data she received
        K1 = (int)genMethod(YC, XA, primeNumber);
        //Bob calculates K using the data he received
        K2 = (int)genMethod(YC, XB, primeNumber);
        //Mallory can read and fabricate messages between both sides
        int K3 = (int)genMethod(YA, XC, primeNumber);
        int K4 = (int)genMethod(YB, XC, primeNumber);

        System.out.println("Key calculated by Alice "+K1);
        System.out.println("Key calculated by Bob "+K2);
        System.out.println("Keys calculated by Mallory for Alice "+K3+" and for Bob
"+K4);
        System.out.println("Messages can read the messages exchanged by Alice and
Bob");
    }


    private static int randomPrimeNumber() {
        //Generating a random number greater than 10000 and less than 100000 (10000 < p
< 100000)
        Random randNum = new Random();
        int random_integer=0;
        boolean primeFound = false;
        while (!primeFound) {
            //If a prime number is not found, generate another number in the range and
check if its prime
            random_integer = randNum.nextInt(100000 - 10001) + 10001;

            //-----Checking for primality-----

            //If the candidate number passes these tests below, the number is a prime
and the "primefound" boolean won't
            //change and will exit the while loop and return the prime number to main
            primeFound = true;

            //If the candidate number is even it's not a prime number
            if (random_integer % 2 == 0) {
                primeFound = false;
            }
            else {
                //Checking odds and the "i" counter increments twice on each iteration
because even numbers have been take care of
                //It checks if each odd number between 3 and the square root to see if
it will divide evenly
                for (int i = 3; i <= Math.sqrt(random_integer); i += 2) {
                    //If any number divides evenly, candidate number is not a prime
                    if (random_integer % i == 0) {
                        primeFound = false;
                        break;
                    }
```

```
                }
            }
        }
        //Return prime number to main
        return random_integer;
    }

    //Finding the primitive root of a prime number using Euler's Totient Function
    //Relative prime numbers of the main prime number will need to be found
    //Source / Inspiration: https://www.geeksforgeeks.org/primitive-root-of-a-prime-number-n-modulo-n/
    private static int primitiveRoot(int prime) {
        //This set will hold all prime factors of the main prime number
        ArrayList<Integer> primeFactors = new ArrayList<>();
        //Since the input is a prime number, there are input-1 (n) relative prime numbers
        int n = prime - 1;
        int s = n; //Value stored for later use

        //The number of two's that divide evenly
        while (n % 2 == 0) {
            //If it divides evenly, store it in the array
            primeFactors.add(2);
            //Divide the value of Euler Totient function again for the next iteration
            n = n / 2;
        }
        //After all the 2's have been accounted for, n is odd now and look for other factors
        //I counter is increased twice on each iteration because we already took care of even factors
        for (int i = 3; i <= Math.sqrt(n); i = i + 2) {
            while (n % i == 0) {
                //Add factor if it divides evenly
                primeFactors.add(i);
                n = n / i;
            }
        }
        //Edge case if n is still greater than 2.
        if (n > 2) {
            primeFactors.add(n);
        }

        //Checking prime factors to see if any have a power of 1
        boolean PRFound = false;
        int count = 0;
        while (!PRFound) {
            boolean found = false;
            Random randCount = new Random();
            //Generate a random candidate primitive root to be checked
            count = randCount.nextInt(prime - 2) + 2; //Random number in the range (2 - (prime-1))

            //Loop through each of the elements of the list until a remainder of 1 is found
            for (Integer a : primeFactors) {
                if (genMethod(count, s / (a), prime) == 1) {
                    found = true;
                    break;
                }
            }

            //If there was no power with value 1, it is a primitive root and change the boolean variable to exit the loop
            if (!found) {
                PRFound = true;
            }
        }
```

```
        //Return the primitive root
        return count;
    }

    private static long genMethod(int base, int exponent, int mod) {
        //Method to calculate extremely large powers in 32bit arithmetic using
recursion and modular arithmetic
        //If exponent is 0, the result of calculation is 1.
        if (exponent==0) {
            return 1;
        }
        //If the exponent can be divided evenly, rerun the method but split into two
parts
        else if (exponent%2 == 0){
            long d = genMethod(base,exponent/2,mod);
            //Data overflow hazard
            //d multiplied to itself will yield results outside the range of values 32-
bit integers can represent
            //therefore giving an incorrect modulo result. The variable d and the
return of function are set to 64bit integers
            //However, the result is cast back to a 32-bit integer upon exiting the
method
            return (d*d)%mod;
        }
        else
            //If the exponent is odd, decrease exponent by 1 and run function again but
multiply result by the 1 exponent
            //That was taken away
            return ((base%mod)*genMethod(base,exponent-1,mod)) %mod;
    }
}
```