Shakeeb Hassan: 33529354: shakeebhassan123@hotmail.com
Mohamed Yassin: 33683874: yasin_warsame@hotmail.com

Mini Project Assessment

Assignment 1:

Total Number of Hours Spent	63.5
Hours Spent for Algorithm Design	8
Hours Spent for Writing Report	12
Hours Spent for Programming	23.5
Hours Spent for Testing	20
Note for examiner:	**go in directory /out/production/rsasim - and java menu should be executable in the terminal !**

When approaching this question, it was important to understand what the question wanted us to do. We took the approach of creating similar scenarios to the Alice, Charlie and Bob diagram. Similarly, implementing the three suggested steps

1. Implement a crypto random key generator, and the algorithm for mod Expo.
2. Implement the RSA encryption algorithm.
3. Implement the RSA decryption algorithm

To illustrate the algorithm we implemented, firstly we must say when creating this and answering this question, RSA Encryption is an incredibly powerful tool, bit length and keys created were intentionally with the thought that we wanted to create a safe system. Therefore, The length of 512 bits was chosen because it is considered to be secure enough for most practical purposes. RSA keys are typically chosen to be at least 2048 bits long today. However, it's important to note that the recommended key size can change over time as computing power increases[1], and it's always a good

idea to use the latest recommendations for secure key lengths. The security of RSA is based on the difficulty of factoring large numbers, which is an important open problem in mathematics. If an attacker can factor n, they can compute the private key and decrypt messages. For this reason, it is important to use large primes in RSA.

To show our understanding I will demonstrate the way the algorithm is used to encrypt and decrypt smaller bit lengths and thus messages. This saves time, despite being less safe and creating more risk of hackers (like Charlie) intercepting messages.

RSA works by generating a pair of keys: a public key used for encrypting messages, and a private key used for decrypting messages. The public key consists of two values: n, which is the product of two large prime numbers, and e, which is a number coprime with φ(n) (the Euler totient of n). The private key consists of one value: d, which is the modular multiplicative inverse of e (mod φ(n)). To encrypt a message, the message is raised to the power of e (mod n), resulting in the ciphertext. To decrypt the ciphertext, the ciphertext is raised to the power of d (mod n), resulting in the original message.[2]

```
private void generateKeys() {
    Random rand = new Random();

    // Generate two random prime numbers p and q
    p = BigInteger.probablePrime(512, rand);
    q = BigInteger.probablePrime(512, rand);

    // Calculate n = p*q and phi = (p-1)*(q-1)
    n = p.multiply(q);
    phi = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));

    // Find a random number e that is coprime with phi
    e = BigInteger.probablePrime(512, rand);
    while (!phi.gcd(e).equals(BigInteger.ONE) || e.compareTo(phi) >= 0) {
        e = BigInteger.probablePrime(512, rand);
    }

    // Calculate the modular multiplicative inverse of e mod phi to get d
    d = e.modInverse(phi); // d has private access
}
```

1. New instance of random class is called to create random numbers:

2. Two prime integers p and q are stored with 512 bits, using the probablePrime method of the BigInteger class:

3. Product of p and q are calculated to find n

4. N is calculated as phi=p-1*q-1 —> $\Phi = (p\text{-}1) * (q\text{-}1)$

5. `e = BigInteger.probablePrime(512, rand);` generates a random prime number e with 512 bits using the probablePrime method from the BigInteger class.

   The while loop that follows ensures that e is coprime with phi. Two numbers are coprime if they have no common factors other than 1. The loop condition `(!phi.gcd(e).equals(BigInteger.ONE) || e.compareTo(phi) >= 0)` checks whether e and phi are coprime, and also that e is less than phi. phi.gcd(e) calculates the greatest common divisor (GCD) of phi and e. If this is not equal to 1, then e and phi share a common factor other than 1, which means e is not coprime with phi.

   e. compareTo(phi) >= 0 checks that e is less than phi, since e needs to be smaller than phi to ensure that it is coprime. If either of these conditions is true, then the loop continues and generates a new random prime e using e = BigInteger.probablePrime(512, rand). Once a valid e value is found, the loop terminates and the program moves on to calculating the private key d.

6. Finding the private key:
   The private key, d, is calculated using the Euclidean algorithm. This algorithm finds the multiplicative inverse of e modulo φ(n), which is a number d such that

$$(e * d) \bmod \varphi(n) = 1$$

Encryption:

```
// Public method to encrypt a message using RSA
public BigInteger encrypt(String message) {
   BigInteger m = new BigInteger(message.getBytes());
   return m.modPow(e, n);
}
```

This function encrypt takes a user's string message as input and returns a BigInteger which is the encrypted version of the message.the input message is converted to bytes using the getBytes method. Next m we created a BigInteger as an array representation representing the message. A byte array is presented through a byte which is an 8-bit value that represents numbers from 0-255. This is interesting as through Java, a message is presented by an array of chars. This is where the getBytes method comes into play as it converts the message containing ASCII representation of the characters in a string. Then we call the modPow method on arguments m, n and e. This method performs modular exponentiation with public key value e modulus n. This message thus creates the encrypted message.

Decryption:

```
// Public method to decrypt a ciphertext using RSA
public String decrypt(BigInteger ciphertext) {
   BigInteger m = ciphertext.modPow(d, n);
   return new String(m.toByteArray());
}
```

Initially, we can see that to decrypt the ciphertext we will need to use the private key d. The formula for this is

$$m = c_d \bmod n$$

M represents the message, c represents the ciphertext modulus n.
This is then converted to a string using the m.toByteArray(), the encrypted message is returned as a string.

Since it was recommended to create a simple user system where we can display the scenario between Alice, Bob and Charlie, I decided to use the scanner class to allow users to input the

message themselves. Obviously this depends on the bit size as a larger bit size means users are able to input a longer message.

The system is fairly simple to understand and took relatively quick to create. The following:

```
// Check if Bob successfully decrypted the message
if (plaintext.equals(message)) {
   System.out.println("Bob successfully decrypted the message");
} else {
   System.out.println("Bob failed to decrypt the message.");
}
```

Make sure that the plaintext is equal to the message input from "Alice". For example, if the ciphertext contains an error or the bitlength is too small for the input message, the code fails and Bob fails to decrypt. An error is printed to the console illustrating this in action.

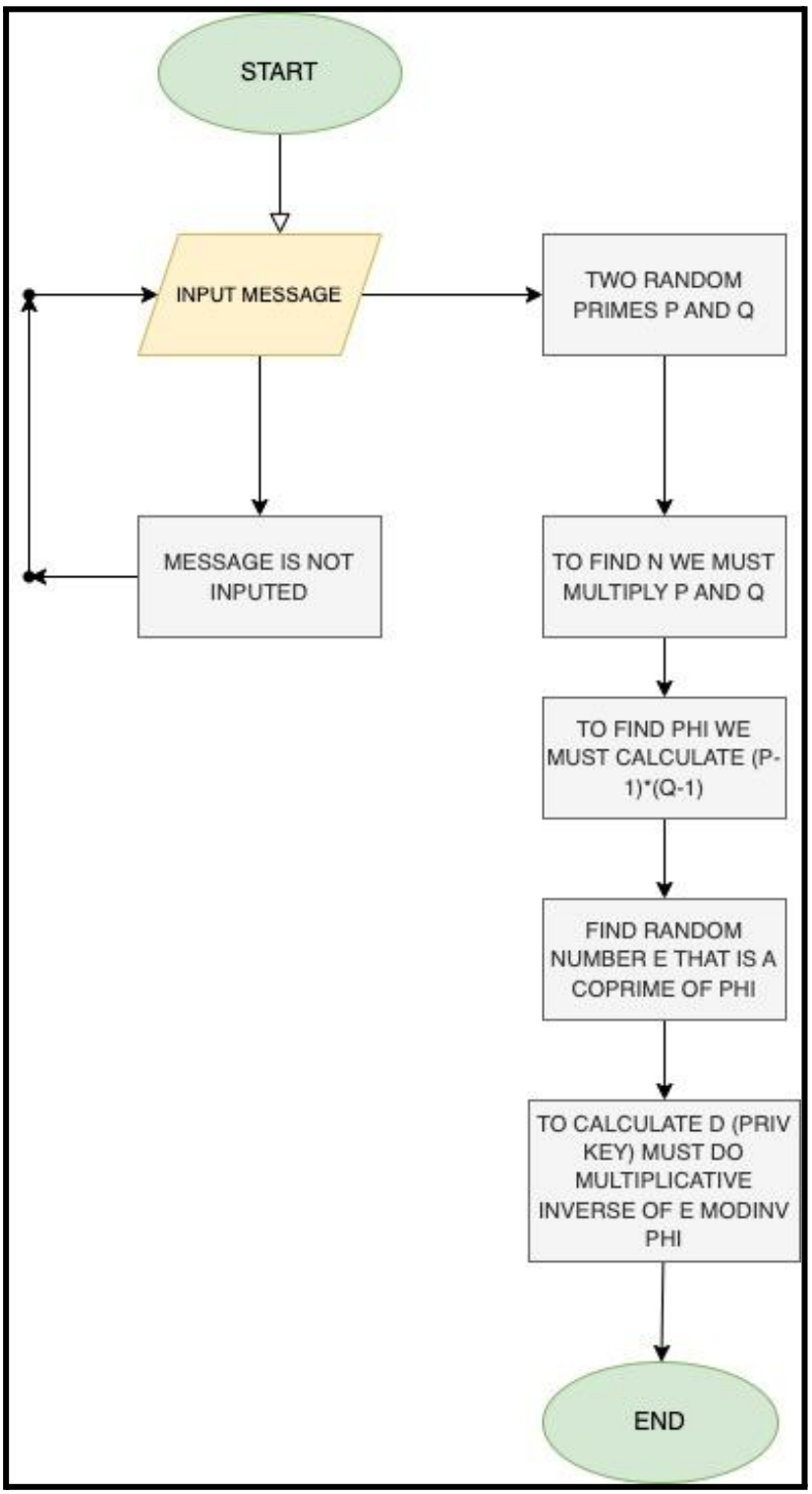
Example of RSA:

P = 3 and q = 13.
N = p * q / 3 * 13 = 39.
Phi = 24.
E = 17 and d = 17
Message m = "hi" = 89
C = 89^17 mod 39
39 = 3 9
= "ci"

M = 39^17 mod 39
M = 0, m = message. Bob has successfully decrypted the message. [3]

This assignment could have been improved in a few ways. For example we could have created a better system of the menu so that users could input their own values for p and q. This would have illustrated a much better understanding of the RSA algorithm. Moreover, creating a system that looks better and works better would also contribute to a better understanding of the algorithm, for us and for even other people. Using this project to help others teach the RSA system would be something interesting to perhaps improve in the future as we upload it to our portfolios or githubs.

Additionally, in terms of emulating the scenario of Alice, Bob and Charlie, perhaps another group member would have helped create a more improved program. Since this was just between two of us we had to take a lot more time creating the design, program and the report. This was definitely a limitation for us. Furthermore, showing how Charlie intercepted the message would have been an important feature to include, however, due to the nature of the RSA system, it would not have been safe if we created a small bit length key, effectively showing how Charlie could intercept through brute force method. This is not a secure and reliable way of decrypting messages.

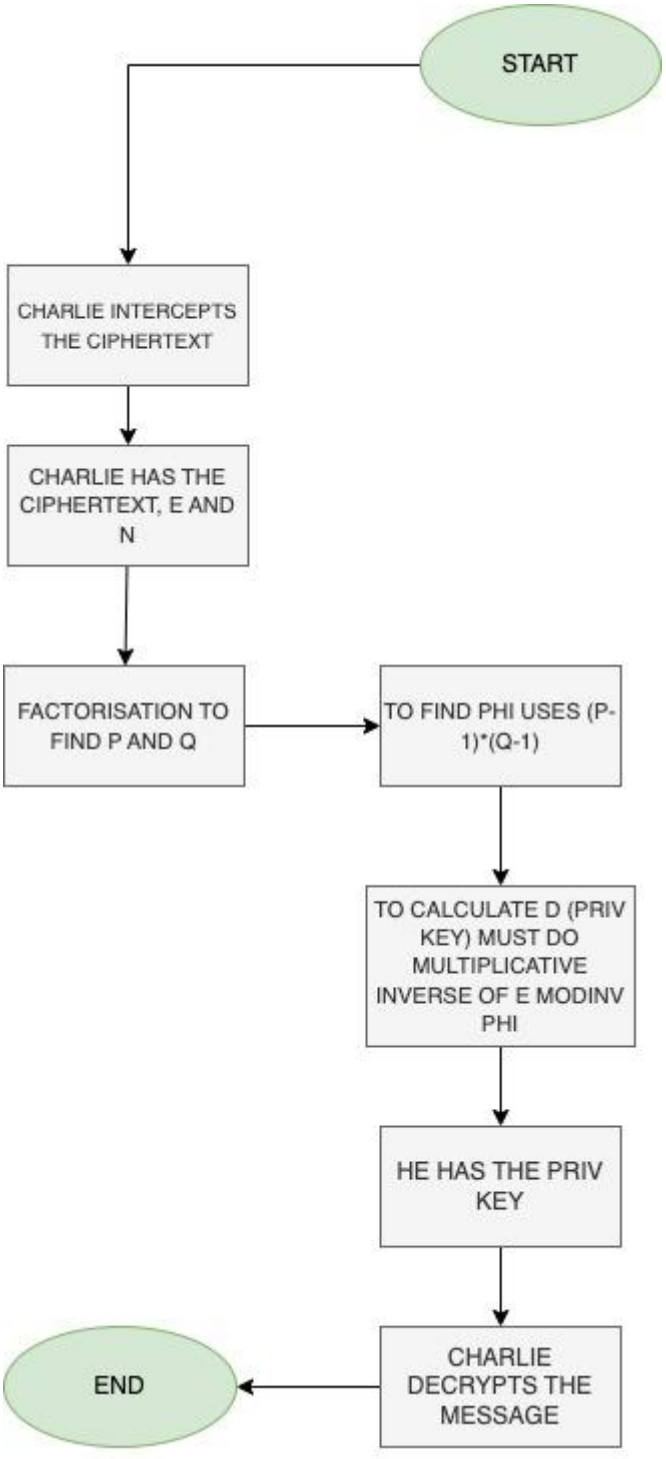


Bibliography:

[1] RSA Key Sizes: 2048 or 4096 bits? https://danielpocock.com/rsa-key-sizes-2048-or-4096-bits/

[2] https://courses.csail.mit.edu/6.006/fall10/handouts/recitation12-3.pdf

[3] https://www.ics.uci.edu/~irani/w17-6D/BoardNotes/13_CryptographyPost.pdf



```java
import java.math.BigInteger;
import java.util.Random;

//Shakeeb Hassan: 33529354: shakeebhassan123@hotmail.com
//Mohamed Yassin: 33683874: yasin_warsame@hotmail.com

public class RSAsim {
   private BigInteger p, q, n, phi, e, d;

   // Constructor method that generates RSA keys
   public RSAsim() {
       generateKeys();
   }

   // Private method to generate RSA keys
   private void generateKeys() {
       Random rand = new Random();

       // Generate two random prime numbers p and q
       p = BigInteger.probablePrime(512, rand);
       q = BigInteger.probablePrime(512, rand);

       // Calculate n = p*q and phi = (p-1)*(q-1)
       n = p.multiply(q);
       phi =
p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));

       // Find a random number e that is coprime with phi
       e = BigInteger.probablePrime(512, rand);
       while (!phi.gcd(e).equals(BigInteger.ONE) || e.compareTo(phi) >= 0) {
           e = BigInteger.probablePrime(512, rand);
       }

       // Calculate the modular multiplicative inverse of e mod phi to get d
       d = e.modInverse(phi); // d has private access
   }

   // Public method to get the RSA public key (e)
   public BigInteger getPublicKey() {
       return e;
   }
```

```
    // Public method to get the RSA private key (d)
    // Note: not secure to expose private key in this way, just for testing
purposes
    public BigInteger getPrivateKey() {
        return d;
    }

    // Public method to encrypt a message using RSA
    public BigInteger encrypt(String message) {
        BigInteger m = new BigInteger(message.getBytes());
        return m.modPow(e, n);
    }

    // Public method to decrypt a ciphertext using RSA
    public String decrypt(BigInteger ciphertext) {
        BigInteger m = ciphertext.modPow(d, n);
        return new String(m.toByteArray());
    }
}
```

```
import java.math.BigInteger;
import java.util.Scanner;
```

```
public class menu {

   public void run() {
       RSAsim rsaSimulation = new RSAsim();
       BigInteger publicKey = rsaSimulation.getPublicKey();
       BigInteger privateKey = rsaSimulation.getPrivateKey();
       Scanner scanner = new Scanner(System.in);

       System.out.println("Welcome to our RSA Algrotihm system!");
       System.out.println("---------------------------------");



       System.out.println("Your public key is: " + publicKey);
       System.out.println("---------------------------------");
       //generally would not show priv key but for testing purposes we
decdied to
       System.out.println("Your private key is: " + privateKey);
       System.out.println("---------------------------------");

       // Alice sends a message to Bob
       System.out.println("Alice, please enter your message:");
       String message = scanner.nextLine();
       System.out.println("---------------------------------");

       BigInteger ciphertext = rsaSimulation.encrypt(message);


       //        //error in the ciphertext causes code to crash and bob to
unsuccssefully decrypt the ciphertext
       //      ciphertext = ciphertext.flipBit(3); // flip the 3rd bit

       System.out.println("Alice's ciphertext: " + ciphertext);
       System.out.println("Sending message to Bob...");
       System.out.println("---------------------------------");

       // Charlie intercepts the message
       BigInteger interceptedCiphertext = ciphertext;
       System.out.println("Hacker Charlie intercepted the message!");
       System.out.println("Intercepted ciphertext: " +
interceptedCiphertext);
```

```
        System.out.println("----------------------------------");


        // Bob receives the message and decrypts it
        String plaintext = rsaSimulation.decrypt(interceptedCiphertext);
        System.out.println("Bob received the ciphertext: " +
interceptedCiphertext);
        System.out.println("Bob's plaintext: " + plaintext);
        System.out.println("----------------------------------");



        // Check if Bob successfully decrypted the message
        if (plaintext.equals(message)) {
            System.out.println("Bob successfully decrypted the message");
        } else {
            System.out.println("Bob failed to decrypt the message.");
        }


        // Charlie sees the plaintext
        System.out.println("----------------------------------");
        System.out.println("Hacker Charlie sees the plaintext: " + plaintext);
    }

    public static void main(String[] args) {
        menu menu = new menu();
        menu.run();
    }
}
```