Computer Security IS53012A/B

Ahmed al-Waili aalw001@gold.ac.uk
Mahmudul Fakrul mfakr001@gold.ac.uk
Taha Zahoni tzach011@gold.c.uk

Total Number of Hours Spent	85 hours
Hours Spent for Algorithm Design	17
Hours Spent for Programming	33
Hours Spent for Writing Report	12.5
Hours Spent for Testing	22.5
Note for the examiner (if any):	Please use command `java gui.menu` in the directory `/out/production/menu`

RSA (Ron Rivest, Adil Shamir and Leonard Adleman)

Section 1: Finding the Key components of RSA

RSA is a very simple yet powerful encryption method if the numbers that are used are large. Now, encryption is not made to be impossible but it is made to be slow to reverse, so the inventors of RSA though of an extremely clever **asymmetrical** encryption method that allows Key Pairs with computers, RSA is made of 2 keys the Public Key and the Private Key, however these Keys do have components in common as the RSA algorithm only uses 6 key components, *p, q, e, n, d* and *phi*. *P* and *Q* are random *large* primes numbers that can be chosen by the algorithm, we will get onto the reason why they are large later. Phi (Φ) is then calculated using *p* and *q*:

$$\Phi = (p-1) * (q-1)$$

Then, the algorithm needs the final value of *e* and n and to do another calculation is made for *n*:

$$n = p * q$$

Now, we are almost at the stage of actually encrypting the messages and text, however we have 1 more key components for the Key Pairs, and that is *e*. To find the value of *e* is bit more complicated than a simple calculation such as above, now the rules of the value of *e* states that is has to be larger than 1 and less than phi Φ, whilst also not having a common factor with phi Φ itself.

$$1 < e < \Phi, \text{ such that } gcd(e, \Phi) = 1$$

Our final component of the RSA algorithm is *d* to find the value of d is:

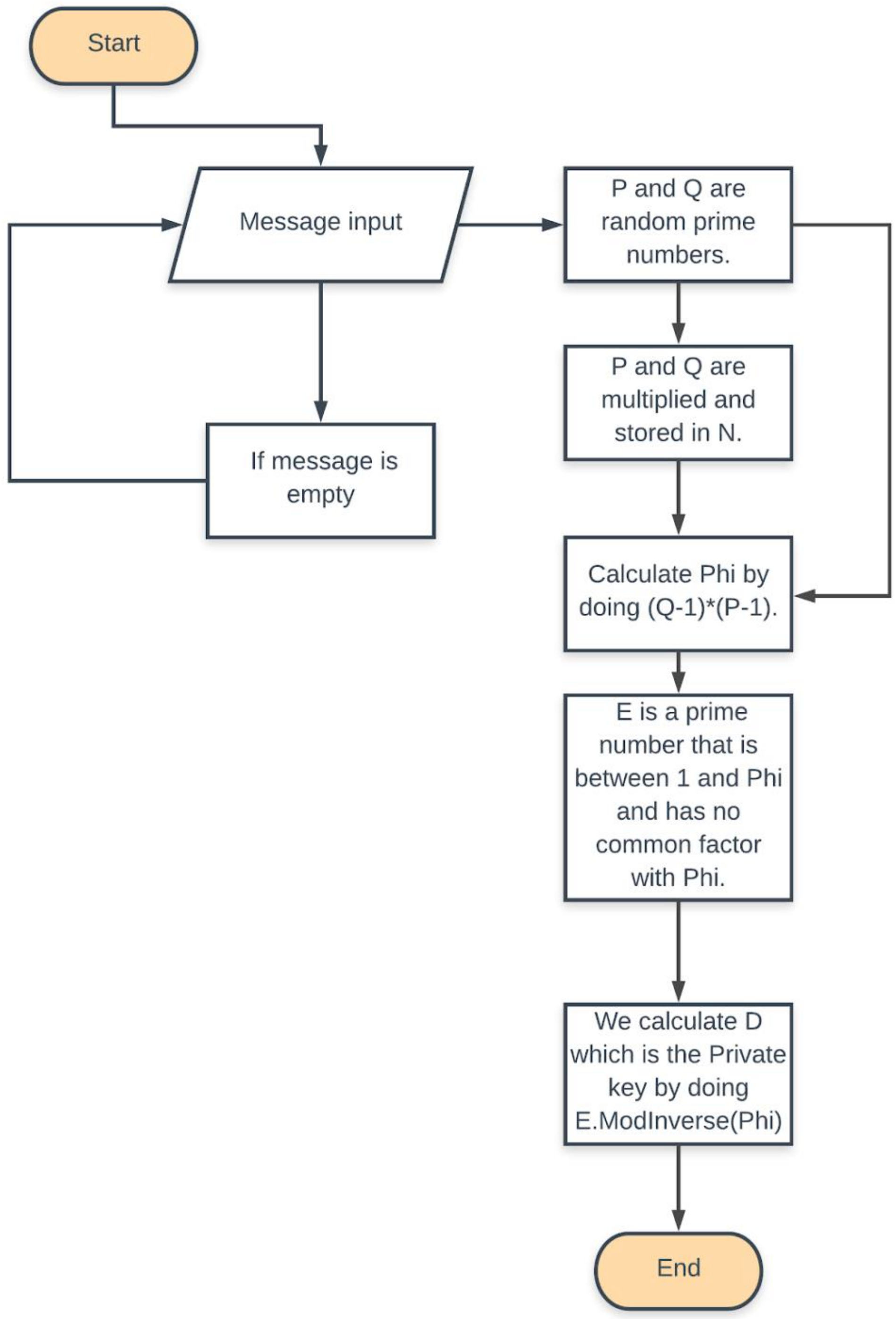
$$d = \text{modular inverse of } (e, \Phi) \text{ where,}$$
$$1 < d < \Phi \text{ such that } e*d = 1 \bmod \Phi$$

Section 1.2: Using the components

Now that we have all the key components of *p, q, e, n, d* and *phi*. We need to now generate the Public Key and Private key, as RSA is **asymmetrical** it therefore needs 2 Keys. The public key will be generated using the value of *e* and *n*, these two components will be public for everyone to see, therefore the Public Key of person A will be:

$$K_A = (e, n)$$

The private key for any one person will the *d* value of the RSA components, this is **private** and no one else apart from the one who created it can use or see it. Refer back to [Section 1] to see how value of *d* is calculated.


Start
Message input
If message is empty
P and Q are random prime numbers.
P and Q are multiplied and stored in N.
Calculate Phi by doing (Q-1)*(P-1).
E is a prime number that is between 1 and Phi and has no common factor with Phi.
We calculate D which is the Private key by doing E.ModInverse(Phi)
End

Section 2: Encryption and Decryption

The Public Key and Private Key has been calculated and stated for each and every use, we can now start to send messages to each other without an intruder getting the information.

Section 2.1: Encryption

Section 2.1.1: One-Way function explained

Encryption is done in such a way where the method to encrypt is hard to reverse, this is called a **One-Way Function,** to explain how the One-Way function works I will use paint as an example with symmetrical Public Key, although the real RSA uses advance mathematics, we will get onto that.

Let’s image Alice wants to send a message to Bob, the message she wants to send is a colour of ‘red’ and Bob wants to send a message back to Alice which is the colour of Blue. Now, for them to send the information to each other, they get the public colour which is ‘Yellow, so now Alice has the colour of Orange and Bob has the colour of Green, now they send this information to each other, the reason this is difficult because the intruder Charlie, will see the colours of Green and Orange, but he wouldn’t be able to reverse the function in such a way where the he gets the private colour of Alice and Bob, it will take him many trial and error attempts to do so. Once Alice and Bob receive each other’s colours, they mix their partners colours with their own private colour, which at the end entail them both having the same colour – which would be a light grey colour.

Section 2.1.2: RSA One-Way function

As mentioned in [Section 2.1.1] a One-Way function is used for the encryption between two parties, however it was mentioned with a symmetrical encryption and RSA works with asymmetrical encryption, to explain this I will explain in mathematical terms on how the encryption is done.

The way the advance One-Way function works is with the modular function in maths that can only be reversed if someone or intruder has access to the private key, but as the private key is not shared and kept on the user’s device this will be long and hard to calculated.

For the calculation function to work we need to declare that the message *m* will be converted into a integer of the message, such as our code shows it will be done with byte array in ASCII, however many different RSA do their own method using UTF-8 or Unicode. Now the message *m* needs to be converted into cipher text *c* which will therefore be sent to the second user Bob:

$$c = m^e \bmod n$$

This cipher text is now generated and sent over to the partner Bob, but by sight this cannot be understood and example will be shown below:
We can assume the example will use the character encoding of 0-25 of all characters of the English alphabet, 0 being a and 25 being z.

$$p = 3, q = 13, n = 3 * 13 = 39, phi = 24, e = 17, d = 17$$

$$\boldsymbol{K_a = (17, 39)}$$

$$m = \text{“}hi\text{”} = 89$$

$$c = 89^{17} \bmod 39$$

$$39 = 3\ 9 = \text{“}ci\text{”}$$

As you can see the message of “hi” using the RSA algorithm is converted into the message of “ci” which of course in the English language does not mean anything. To show how we get back to the original message please refer to [Section 2.2].

Section 2.2: Decryption

Taking into account the Encryption in [Section 2.1], we need a way of decrypting the message and being able to see the actual message itself. As stated above the rules of for the function is that it can only be reversed if *d* is known and as *d* is private to the user itself then only the receiver can get the information and see the message. So, Alice is sending a message to Bob, so Alice uses Bobs K_b Public Key so **only** Bob can view it as he of course knowns his own value of *d*. The calculations for the reverse method is:

Using the same notation as above, *m* being the message, *c* being the cipher text.

$$m = c^d \bmod n$$

Using the same example as above carrying on we can see that this is true:

$$p = 3, q = 13, n = 3 * 13 = 39, phi = 24, e = 17, d = 17$$

$$c = \text{“}ci\text{”} = 39$$

$$m = 39^{17} \bmod 39$$
$$m =$$

Bob has successfully got the message from Alice.

Section 3: Our Implementation

To make life a little easier I will be referring back to the sections above to show what code did what exactly.

The order of our stack of progress to complete the code goes as follows, bottom being the foundation of what is going to be implemented:

8	GUI
7	Convert messages String into Bytes
6	Decrypt using m = c^d mod n
5	Encrypt using c = m^e mod n
4	Calculate d using mod inverse
3	Check if e has a gcd with phi
2	Check if e follows the rules 1 < e < phi
1	p, q, e, n and phi calculations

Section 3.1: p, q, e, n and phi calculations

p, q and *e* are of course random prime numbers to the code is a simple one liner to get the values:

```
          Random r = new Random();//new random is mad e

    int bit_len = 16;
    //random prime from 0 to bit_len 65000 is calculated
    // for p, q, e
    p = BigInteger.probablePrime(bit_len, r);

    q = BigInteger.probablePrime(bit_len, r);

    //n is calculated by p * q
    n = p.multiply(q);

    BigInteger pMinus1 = p.subtract(BigInteger.ONE);
    BigInteger qMinus1 = q.subtract(BigInteger.ONE);
    //phi is (p-1)(q-1)
    phi = pMinus1.multiply(qMinus1);


    //e is random prime
    e = BigInteger.probablePrime(bit_len / 2, r);
```

As, you can all the values are calculated using the function of probable prime, this function returns a prime number that is of random *r* within the values of the bit length. The bit length *bit_len* indicates what the maximum integer of bit can be, such that if *bit_len* == 8 then the maximum random number that can be prime will be 251, as 8 bit is maximum up to 256, 2^8. We used a maximum of 2^{16} as the input text can only allow 2 characters within the program itself, so therefore this will allow any ASCII character on the average keyboard with just 2 characters.

Section 3.3 & 3.3: Check if *e* follows the rules 1 < e < phi and gcd of phi

The rules of *e* are stated in [Section 1], as *e* has to be a prime number between 1 and phi and also have no common factor with phi itself. To do, we first made a function that checks if *e* does or does not have any common factor with phi itself using the code below.

```
while (divisor() > 0 && e.compareTo(phi) < 0){//checks to see if e is diviable or not
   e.add(BigInteger.ONE);
}
public int divisor() {
   return phi.gcd(e).compareTo(BigInteger.ONE);
}//method checks if gcd is true
```

In the divisor() method you can see that we return the greatest common divisor comparted to 1, as the rule states *e*d mod phi* must equal to 1.

Now looking into the Boolean statement of the while loop you can see that we also compare to see if *e* comparted to *phi* to see if the value of *e* is less or larger than *phi*, while this is true we add one to the value of *e* to finally get the highest value that can fit within these boundaries.

Section 3.4: Calculating d using modular inverse

To calculate the values of *d* it has to follow the rules stated in [Section 1] that the value has to be modular inverse of e and phi where 1 < d < phi and such that e*d mod phi must equal 1. Now the while loop above works in junction with that as you can see the values must be checked to see if the it equals to one, in the divisor() method of the code. So, after this while loop is completed as it does not stop until the Boolean statement is false of course, we move on to:

```
d = e.modInverse(phi);//private key
```

where *d* is simply the modular inverse of the value of phi, using the Euclidian algorithm to find the modular inverse.

Section 3.5: Encrypting using c = m^e mod n

For the message to be calculated we must first convert the values of the message into byte[] so our method of finding the cipher text can be found, and to do we simply used the inbuilt function of Java[2] called .getBytes();

```
byte[] encrypted = rsa.encrypt(encryptText.getText().getBytes());//please reference RSA.java for comments here
```

This line of code is in the main method of guiRSA.java, the line *encryptText.getText()* takes the input from the input box that anyone writes in and then the *.getBytes()* method converts this into bytes assigning it to a variable *encrypted.*

The variable *encrypted* is then used as a parameter for the encrypt function that can be found in the RSA.java called *encrypt* hence the value of *rsa.encrypt.*

Line 3 below shows that the value of the message in bytes is then converted into a Big Integer to allow calculations to be made to it. By simply making the bytes to int

```
public byte[] encrypt(byte[] message) {//get cipher by encrypting message
    try {
        BigInteger temp = new BigInteger(message);
        c = temp.modPow(e, n);// encrypted using c = m^e mod n
        byte[] b = c.toByteArray();
        return b;
    }catch (NumberFormatException e){
        System.out.println("Please enter message");
        return null;
    }
}
```

As you can see in the method above, we used a Try and Catch method to see if the values are empty, then an error will be thrown. Further to that you can see evidence of the encryption calculation of:

$$c = m^e \bmod n$$

You can see this in line 4 of the code snippet above, as the temp value which is the integer value of message that has been converted into bytes is put to the power of *e* and then modded to the value of *n*.

Section 3.6: Decrypting using m = c^d mod n & convert message into String

For us to go into detail about how the decryption method works, we must first take step back and talk about the cipher text, the program first make the cipher text equal to what ever message is encrypted.

```
cipherText.setText(new String(encrypted));
```

Now, on screen the user is able to see what the cipher text and how the input text went from whatever was inputted into the cipher text, which is an assortment of ASCII characters.

Now, we can go into detail on how the decryption happens, as you can see by the code below it is very similar almost exactly the same as the encryption method which, is correct a very similar modular function is used for both.

```
public byte[] dencrypt(byte[] message) {//get message by decrypting the cipher
    try {
        BigInteger temp = new BigInteger(message);
        BigInteger m = temp.modPow(d, n);//decrypt  using the n = c^d mod n
        byte[] b = m.toByteArray();
        return b;
    }catch (NumberFormatException e){
        System.out.println("Please enter message");
        return null;
    }
}
```

All information that the encrypt method of course applies here, on how the values are converted into integer so calcualtions can be made to the messages. However, you may see a small difference in line 4 comparted to line 4 in [Section 3.5], this time the value of *d* is used instead of the value of e and of course the cipher text is what is used and not the message itself, as the message is the goal state, mentioned in [Section 2.1.2]

This method then return a byte array of *b*, but we are still not done as the byte array needs to be converted back into text just how we converted into String, which is done with the line below:

```
decryptText.setText(new String(decrypted));
```

new String() is a method built into java that can make any byte array in a String and can out put it.

We are done with the encryption but just to make sure our program works we applied validation to the values of the input and the values of the decryption text.

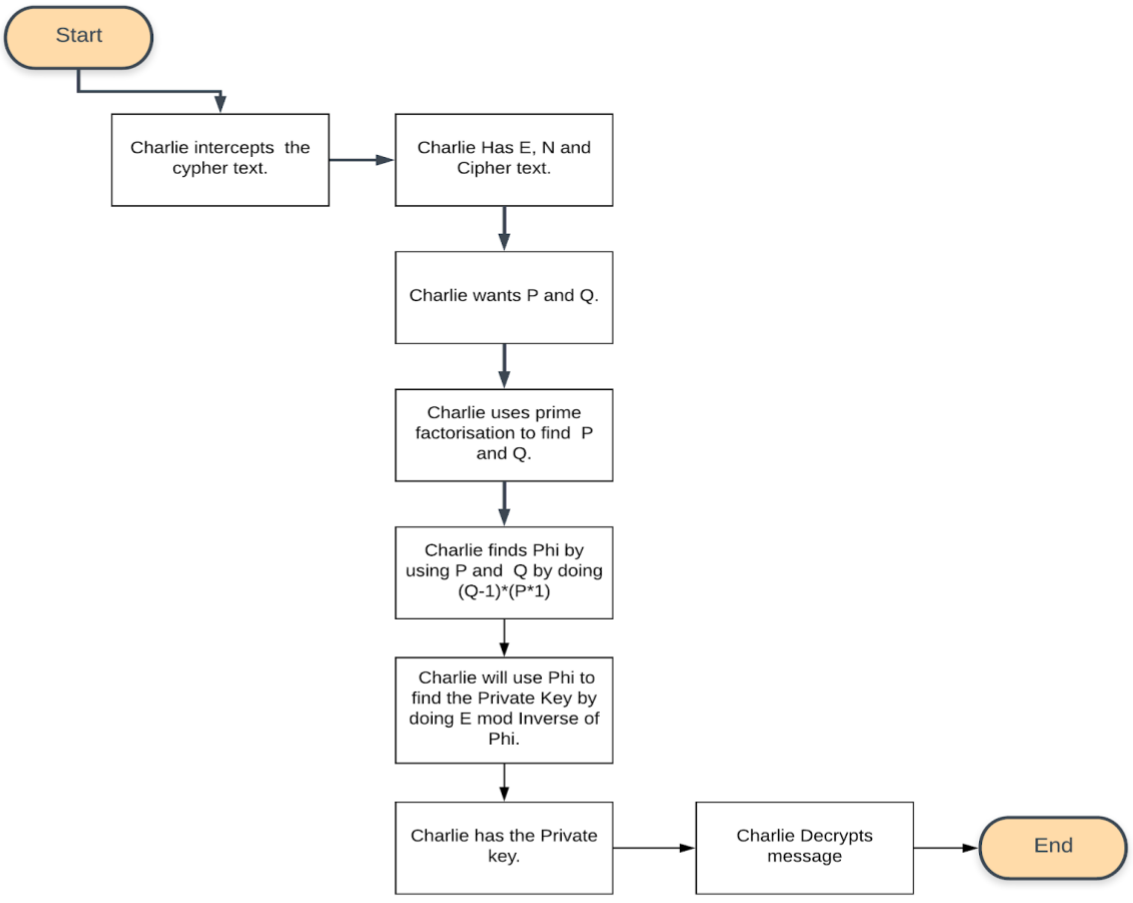
```
if(input.equals(new String(decrypted))){
    decryptText.setBackground(Color.GREEN);
}else{
    decryptText.setBackground(Color.RED);
}
```

This states, that if the Decrypted text is equal to what is inputted, then the RSA algorithm is successful and the text box will go green in celebration of your success, however will go red in the humility of your failure. There are certain scenarios which can affect the output and of course make the encryption fail.

Section 3.7: The Hacker Charlie's interception

RSA however, is not completely safe, if you have the computational power and the time on your hands then RSA can be easily deciphered and all messages can be found out, however with RSA and SHA256, we won't danger for many years to come, unless Quantum computers come before that.

There are people in the world that would like to intercept and listen to your messages and information sent over the internet without you know, and that's where Charlie comes in, now they're not very nice and have been listening to the conversation example in [Section 2.1.2] and because the numbers were so low Charlie was easily able to find the values of p and q with brute force.


Start
Charlie intercepts the cypher text.
Charlie Has E, N and Cipher text.
Charlie wants P and Q.
Charlie uses prime factorisation to find P and Q.
Charlie finds Phi by using P and Q by doing (Q-1)*(P*1)
Charlie will use Phi to find the Private Key by doing E mod Inverse of Phi.
Charlie has the Private key.
Charlie Decrypts message
End

As stated, the values of e and n are public to everyone and anyone can see them, so Charlie takes advantage of that, now if we look at the information in detail, what values does Charlie have?

Charlie has the values of:

$$n, e \text{ and } c$$

Now as mentioned to decrypt the value of c, the message $m = c^d \bmod n$. Charlie has 2 of the 3 values he needs to find out m, all he needs to do now is find out what d is which is private, but with enough time Charlie can brute force his way to find d.

Section 3.7.1: Charlie finding p and q

To find *d* Charlie first must find the value of phi, as d is mod inverse of (e, phi), and again for Charlie to find the value of phi, he must know what p and q is, and to do this is not easy with big numbers, but for demonstration purposes our code has simple numbers only up to the bit length of 16, 2^{16}. This information of Charlie is held in the file hacker.java

First what Charlie does is find every prime number up to the maximum length of the RSA encryption, for us it was 2^{16}, but in general it would be 2^{4096}.

```
  public void isPrime(){
      BigInteger n = new BigInteger("65536");
      for(int i = 0; i < 65536; i++){
//            System.out.println(n.toString());
          if(n.isProbablePrime(100)){
              series.add(n);
          }
          n = n.subtract(BigInteger.ONE);
      }
  }
```

Charlie then finds all the prime numbers from numbers 0 to 65536, and adds them into an array called series. He now has all the prime numbers up to 65536, and can multiply them each together one by one to get the value of *n*, and he can double check if this is correct as *n* is public for everyone to see.

```
  public BigInteger[] brute(BigInteger n){
      arr[0] = BigInteger.ZERO;
      arr[1] = BigInteger.ZERO;
      for(int i = 0; i < series.size(); i++){
          BigInteger p = series.get(i);

          for(int j = 1; j < series.size(); j++){
              BigInteger q = series.get(j);

              BigInteger guess = p.multiply(q);
              if(guess.equals(n)){
                  arr[0] = p;
                  arr[1] = q;
                  return arr;
              }
          }
      }
```

```
        return null;
    }
```

This two dimensional for loop, looks from the 2 length of 65536, and multiplies the numbers together until he finds that his guess for number 1 and guess for number 2 equal to the public value of *n*. He then adds this into a contained array in this scope and returns this array, so these two values can be used in any location.

Section 3.7.2: Charlie finding phi

Now that Charlie has the values of p and q which he estimated and then verified as only 2 prime numbers would times to the value of *n*. He now needs to find the value of phi, which is simple in theory as all he has to do is *p-1* times *q-1*.

```
public BigInteger findPHI(){
    BigInteger p = arr[0];
    BigInteger q = arr[1];

    BigInteger pMinus1 = p.subtract(BigInteger.ONE);
    BigInteger qMinus1 = q.subtract(BigInteger.ONE);

    BigInteger phi = pMinus1.multiply(qMinus1);

    return phi;
}
```

That is what this method does called findPHI(). It simply takes 1 away from *p* and takes 1 away from *q* and then multiplies it together to get the value of phi.

Section 3.7.3: Charlie finding *d*

Charlie has now all values he needs to calculate *d* which is equal to the mod inverse of (e, phi).

Charlie simple does:
d = modular inverse (e, phi)

Section 3.7.4: Charlie getting the message

Charlie now has all the vital information he needs to find the message m, all the values Charlie now has are:

n, e, c, p, q and phi

All is left is for Charlie to find the value of m which is the message, and to do so you simply follow the decryption function

$$m = c^d \ mod \ n$$

Charlie now has the message m, and he was able to intercept without the two parties knowing.

Section 4: Authentication using a Trusted Server s

We implemented a Server which is based on Needham-Schroeder Protocol.
Here, Alice and Bob don't know each other public keys. So they cannot trust messages sent to them, unless it is signed by the Server.

Below are the steps our program goes through:

1) Alice requests Bob's public key to the Server.

2) Sever then encrypts Bob's public key K_B using Alice's public key K_A, and that is how it signs it. The encryption makes sure that only Alice can decrypt the message to get K_B.

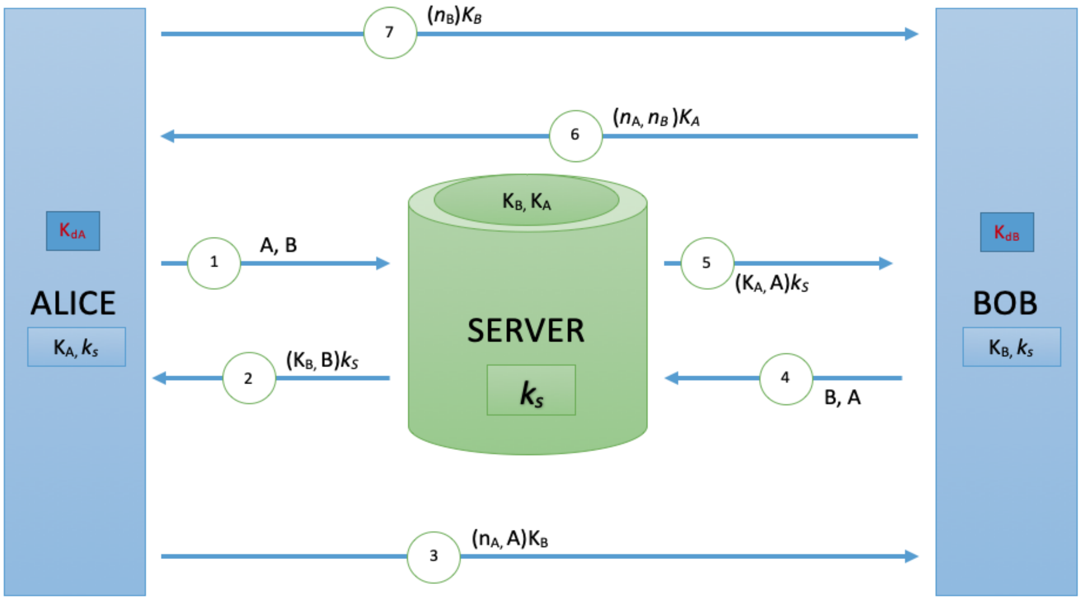
3) Alice then sends a nonce, n_A , encrypted with K_A.
The encryption makes sure that only Bob can read nonce $n_{A,}$ because he'd have to request Alice's public key to the server in order to decrypt it.

4) Bob then requests Alice's public key to the Server

5) Server sends K_A , encrypted with K_B . This way Bob reads Alice's nonce n_A ; to prove he is who Alice wants to communicate with.

6) Bob sends back Alice's nonce n_A and his nonce n_B Encrypted with his public key K_B.

7) Alice finally decrypts and reads both nonce, and sends back Bob's one encrypted with her public key to prove herself.



This way now the two parties can communicate with each other, having established a trusted communication.

Section 5: Problems with RSA and Needham-Schroeder

Section 5.1: Problems with RSA

Most common problem with the RSA algorithm is the fact that you can crack decrypted messages with the Public key. The private key is used to decrypt messages that are encrypted. There are certain requirements we need to meet in order for our RSA Algorithm to work in an appropriate manner. These requirements consist of E being a prime number that is between 1 and Phi, E must have no common factors with Phi, Q and P always being a prime number, E having a large prime number and must not hold a small value and N Being P * Q.

We have met all these requirements however, in our case Charlie still manages to brute force by doing prime factorisation. This is due to us using a 16-bit number which is 2^{16} giving us a maximum prime number from 0 to 65,536. If we used 4,096 bits we would have a more secure encryption giving Charlie a hard time trying to find P and Q meaning he would not be able to find the private key.

Section 5.1: Problems with Needham-Schroeder

Now that we got the RSA working as intended, we want to implement the Needham-Schroeder and use RSA to encrypt the messages between Alice and the Server. However, for us to implement the Needham-Schroeder we needed to meet requirements which includes; Alice and Bob, a server; a server signature. We also had to make sure our RSA encryption and decryption was working as intended. This is because when Alice requests Bobs Key the server decrypts it. This is the same with Alice's' key where the server decrypts it by using RSA.

The Needham-Schroeder has a few issues which is the replay attack.[1] If the attacker uses an older K(ab) value that has been compromised before. He can replay the message to Bob. Bob won't know if the key is fresh. By using the Kerberos Protocol which includes time stamps of Keys. The uses of sending a nonce across is more secure.

The problem we had with the Needham-Schroeder was to implement the Nonce. By adding in 2 nonces and had to concatenate them. However, they already were concatenated, we realised the mistake by having them as a number. The way we fixed this was having them concatenate as a string. A way of a nonce to work would be if person A sends person B a package. For example, when person A sends a package to person B. Person A would put a lock on the package. When person B would receive the package, person B then proceeds to put his lock on the package and sends it back to person A. Person A then proceeds to remove his lock and sends it back to person B.

Section 6: Bibliography

Java: https://www.oracle.com/uk/java/

BigInteger : https://www.tutorialspoint.com/java/math/biginteger_compareto.htm

RSA: https://en.wikipedia.org/wiki/RSA_(cryptosystem)

RSA Java: https://introcs.cs.princeton.edu/java/99crypto/RSA.java.html

RSA Java class: https://codedost.com/css/java-program-rsa-algorithm/

RSA Java: https://www.sanfoundry.com/java-program-implement-rsa-algorithm/

Section 7: Our code

Menu.java

```
package gui;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.math.BigInteger;

public class menu {
    private JButton RSAAlgorithmButton;
    private JButton serverButton;
    private JPanel backPanel;
    private JTextPane paraPanel;

    public menu() {
        RSAAlgorithmButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                (new rsaGUI()).show();
            }
        });
        serverButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                (new serverGUI()).show();
            }
        });
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("RSA by Ahmed Al-waili, Mahmudul Fakrul, Taha Zacholi");
        frame.setContentPane(new menu().backPanel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

RSA.java

```
package gui;

import java.io.IOException;
import java.math.*;
import java.util.Random;
import java.awt.FlowLayout;
import javax.swing.JFrame;

public class RSA extends JFrame{

    public BigInteger p, q, n, phi, e, d, c;
    int bit_len = 16;

    public RSA() {

        Random r = new Random();//new random is made
//        System.out.println("r " + r.toString());

        //random prime from 0 to bit_len 65000 is calculated
        // for p, q, e
        p = BigInteger.probablePrime(bit_len, r);
//        System.out.println("p " + p.toString());

        q = BigInteger.probablePrime(bit_len, r);
//        System.out.println("q " + q.toString());

        //n is calculated by p * q
        n = p.multiply(q);
//        System.out.println("n " + n.toString());



        BigInteger pMinus1 = p.subtract(BigInteger.ONE);
        BigInteger qMinus1 = q.subtract(BigInteger.ONE);

        //phi is (p-1)(q-1)
        phi = pMinus1.multiply(qMinus1);

        //e is random prime
        e = BigInteger.probablePrime(bit_len / 2, r);
//        System.out.println("e " + e.toString());

        //(e, n) make up the public Key

        //but then e is checked to see if it has a gcd with phi
        while (divisor() > 0 && e.compareTo(phi) < 0){//checks to see if e is diviable or not
            e.add(BigInteger.ONE);
        }
        d = e.modInverse(phi);//private key
//        System.out.println("d " + d.toString());
    }

    public int divisor() {
        return phi.gcd(e).compareTo(BigInteger.ONE);
    }//method checks if gcd is true

    public static String byteToString(byte[] message) {
        String output ="";
        for(byte i: message) {
            output += Byte.toString(i);
        }
        return output;
    }

    public byte[] encrypt(byte[] message) {//get cipher by encrypting message
        try {
            BigInteger temp = new BigInteger(message);
            c = temp.modPow(e, n);// encrypted using c = m^e mod n
            byte[] b = c.toByteArray();
            return b;
        }catch (NumberFormatException e){
            System.out.println("Please enter message");
            return null;
        }
    }



    public byte[] dencrypt(byte[] message) {//get message by decrypting the cipher
        try {
```

```java
            BigInteger temp = new BigInteger(message);
            BigInteger m = temp.modPow(d, n);//decrypt  using the n = c^d mod n
            byte[] b = m.toByteArray();
            return b;
        }catch (NumberFormatException e){
            System.out.println("Please enter message");
            return null;
        }
    }

    public BigInteger getE(){
        return e;
    }

    public BigInteger getN(){
        return n;
    }

}
```

rsaGUI.java

```
package gui;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class rsaGUI extends  NumberFormatException {
    private JPanel backPanel;
    private JTextField textInput;
    private JButton submitButton;
    private JLabel encryptLabel;
    private JLabel cipherLabel;
    private JLabel decryptLabel;
    private JScrollPane encryptPanel;
    private JTextArea decryptText;
    private JTextArea cipherText;
    private JTextArea encryptText;
    private JTextArea charlieDec;
    private JButton workOutDAndButton;
    private JButton brutePAndQButton;
    private JButton factoriseForNButton;
    private JLabel brutePnQ;
    private JLabel workOutD;
    private JLabel findN;
    private JButton checkIfCharlieIsButton;
    private JLabel pqCheck;
    private JLabel dphicorrect;
    private JLabel nCheck;
    private JLabel error;



    public rsaGUI() {
        RSA rsa = new RSA();//inistalise RSA method
        Persons alice = new Persons(rsa);//inistialise person with an rsa
        Persons bob = new Persons(rsa);//inistialise person with an rsa
        Hacker charlie = new Hacker();//inistialise person that is a hacker to read whats going on
        //all charliers brute force calculations before betton pressed
        charlie.isPrime();
        charlie.brute(rsa.n);
        charlie.findD(rsa.e);

        //Action listener for when enter is pressed
        textInput.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                String input = textInput.getText();//gets the input of the text field that user inputs
                if(input.equals("")){//check if empty to throw an error
                    error.setText("Please enter a message");
                }

                encryptText.setText(input);//the message is shown on screen with a .setText method

                //the string is then converted into bytes
                byte[] encrypted = rsa.encrypt(encryptText.getText().getBytes());//please reference RSA.java for
comments here
//                System.out.println(rsa.byteToString(encrypted) + "EN");

                try {//try and catch to see if the input is empty, if so then an error is thrown in console
                    cipherText.setText(new String(encrypted));//the bytes are then converted back into strings so it can
be displayed in gui
                }catch (NullPointerException cipherText){
                    System.out.println("Enter message");
                }
//                System.out.println(new String(encrypted));
//                System.out.println("heyo " + cipherText.getText());
//                byte[] testing = cipherText.getText().getBytes();
//                System.out.println(rsa.byteToString(testing) + "DE");

                //the string is then converted into bytes
//                byte[] decrypted = rsa.dencrypt(cipherText.getText().());//please reference RSA.java for comments
here
                try {//try and catch to see if the values are empty or null to throw an error
                    byte[] decrypted = rsa.dencrypt(encrypted);//please reference RSA.java for comments here
                    decryptText.setText(new String(decrypted));//the bytes are then converted back into strings so it can
be displayed in gui
//                System.out.println(new String(decrypted)
```

```java
                    if(input.equals(new String(decrypted))){
                        decryptText.setBackground(Color.GREEN);
                    }else{
                        decryptText.setBackground(Color.RED);
                    }
                } catch (NullPointerException decrpytText ){
                    System.out.println("Enter message");
                }
            }
        });
        //Action listener for when submit button is pressed
        submitButton.addActionListener(new ActionListener() {


            //this section shows the same as above but this is Action Listener is put on the button instead of the 'enter' key


            @Override
            public void actionPerformed(ActionEvent e) {
                String input = textInput.getText();
                if(input.equals("")){
                    error.setText("Please enter a message");
                }
                encryptText.setText(input);//the message is shown on screen with a .setText method

                //the string is then converted into bytes
                byte[] encrypted = rsa.encrypt(encryptText.getText().getBytes());//please reference RSA.java for comments here
//                System.out.println(rsa.byteToString(encrypted) + "EN");



                try {
                    cipherText.setText(new String(encrypted));//the bytes are then converted back into strings so it can be displayed in gui
                }catch (NullPointerException cipherText){
                    System.out.println("Enter message");
                }//                charlieDec.setText(new String(encrypted));//the bytes are then converted back into strings so it can be displayed in gui
```

```
//                System.out.println(new String(encrypted));



//                System.out.println("heyo " + cipherText.getText());

                byte[] testing = cipherText.getText().getBytes();
//                System.out.println(rsa.byteToString(testing) + "DE");

                //the string is then converted into bytes
//                byte[] decrypted = rsa.dencrypt(cipherText.getText().());//please reference RSA.java for comments
here
                try {
                    byte[] decrypted = rsa.dencrypt(encrypted);//please reference RSA.java for comments here
                    decryptText.setText(new String(decrypted));//the bytes are then converted back into strings so it can
be displayed in gui
//                System.out.println(new String(decrypted)
                    if(input.equals(new String(decrypted))){
                        decryptText.setBackground(Color.GREEN);
                    }else{
                        decryptText.setBackground(Color.RED);
                    }
                } catch (NullPointerException decrpytText ){
                    System.out.println("Enter message");
                }

            }
        });



        //for the code below please go to the hacker file to understand in full detail



        brutePAndQButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                //charlie brute force to check p and q, go to hacker file to understand more
                charlie.brute(rsa.n);
                brutePnQ.setText("p = " + charlie.arr[0] + " q = "+ charlie.arr[1]);
            }
        });
        workOutDAndButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                //charlie works out d by getting the values of p and q and phi
                workOutD.setText("d = " + String.valueOf(charlie.findD(rsa.e)) + " phi = " + charlie.findPHI());
            }
        });
        factoriseForNButton.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                findN.setText(String.valueOf(rsa.n));
            }
        });
        checkIfCharlieIsButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
//                JLabel pqCheck;
//                JLabel dphicorrect;
//                JLabel nCheck;

                //the code ran checks to see if p and q are accidently swapped although this cannot happen
                //this still does check just in case
                if(rsa.p.equals(charlie.arr[0])
                        || rsa.p.equals(charlie.arr[1])
                        || rsa.q.equals(charlie.arr[0])
                        || rsa.q.equals(charlie.arr[1])){
                    pqCheck.setText("CORRECT!");
                    pqCheck.setBackground(Color.GREEN);
                }

                if(charlie.findD(rsa.e).equals(rsa.d) && charlie.findPHI().equals(rsa.phi)){
                    dphicorrect.setText("CORRECT!");
                    dphicorrect.setBackground(Color.GREEN);
                }

                nCheck.setText(charlie.arr[0] + " * " + charlie.arr[1] + " = " +
String.valueOf(charlie.arr[0].multiply(charlie.arr[1])) );
```

```
            }
        });
    }

    public void show() {
        //gui coded to show on screen
        JFrame frame = new JFrame("RSA by Ahmed Al-waili, Mahmudul Fakrul, Taha Zacholi");
        frame.setContentPane(new rsaGUI().backPanel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Persons.java

```java
package gui;

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


public class Persons {


    //all variables are mad
    public RSA rsa;
    String nounce;
    String partnerNounce;
    //nounces are set for demonstration purposes
    //we used string as concatanation is easier to do
    String[] nounceArr = {"abcd","edfg","hijk","lmno","pqrs","tuvw","xyza","aabb","bbaa","baab",};
    int rnd = new Random().nextInt(9);

    Persons(RSA rsa){
        this.rsa = rsa;
        nounce = nounceArr[rnd];//get noounce when inialised
    }

    public String getPublicKey(){
        return "(" + rsa.getE().toString() + ", " + rsa.getN().toString() + ")";
    }

    public BigInteger getE(){
        return rsa.getE();
    }

    public BigInteger getN(){
        return rsa.getN();
    }

    public BigInteger[] getPartnerKey(Server s, Persons p){
        return s.getPublicKey(p);//get the public key of who the partner is from the server  s
    }
    //same encryption as from the RSA.java but different as e and n are parameters
    public byte[] serverEncrypt(Server s, byte[] message, BigInteger e, BigInteger n) {//get cipher
        BigInteger temp = new BigInteger(message);
        BigInteger c = temp.modPow(e, n);//public key
        byte[] b = c.toByteArray();

//        System.out.println(new String(b));

        return b;
    }

    //verification is made by the user and checks if the public key is usable
    public BigInteger verify(Server s, BigInteger e){//needs to be signed
        BigInteger temp = e.modPow(rsa.d,  rsa.n);
//        System.out.println("RSA n " + rsa.n);
        return temp;
    }

    public String viewNounce(){
        return nounce;
    }

    public byte[] getNounce(){
        return nounce.getBytes();
    }



    //send nounce to a person to and encrytps it
    public byte[] sendNounce(Persons to){
        System.out.println(nounce);
        byte[] b = to.rsa.encrypt(nounce.getBytes());

//        System.out.println("EN " + new String(b));

        to.setPartnerNounce(b);
        return nounce.getBytes();
    }
    //make the partenr connecting to a partner and give them their nounce
    public void setPartnerNounce(byte[] b){
        partnerNounce = new String(b);
    }

    //decrypt and able to read the nonce
    public byte[] decryptNounce(Persons from){
        byte[] b = partnerNounce.getBytes();
        return rsa.dencrypt(b);
    }
}
```

hacker.java

```java
package gui;

import java.math.BigInteger;
import java.util.ArrayList;

public class Hacker {

    ArrayList<BigInteger> series = new ArrayList<BigInteger>();
    BigInteger[] arr = new BigInteger[2];

    Hacker(){
    }

    public void isPrime(){
        BigInteger n = new BigInteger("65536");
        for(int i = 0; i < 65536; i++){
//            System.out.println(n.toString());
            if(n.isProbablePrime(100)){
                series.add(n);
            }
            n = n.subtract(BigInteger.ONE);
        }
    }

    public BigInteger[] brute(BigInteger n){
        arr[0] = BigInteger.ZERO;
        arr[1] = BigInteger.ZERO;
        for(int i = 0; i < series.size(); i++){
            BigInteger p = series.get(i);

            for(int j = 1; j < series.size(); j++){
                BigInteger q = series.get(j);

                BigInteger guess = p.multiply(q);
                if(guess.equals(n)){
                    arr[0] = p;
//                    System.out.println(arr[0]);
                    arr[1] = q;
//                    System.out.println(arr[1]);
                    return arr;
                }
            }
        }
        return null;
    }

    public BigInteger findPHI(){
        BigInteger p = arr[0];
        BigInteger q = arr[1];




        System.out.println(arr[0]);
        System.out.println(arr[1]);
       //System.out.println("phiiiii :" +p);



        BigInteger pMinus1 = p.subtract(BigInteger.ONE);
        BigInteger qMinus1 = q.subtract(BigInteger.ONE);

        BigInteger phi = pMinus1.multiply(qMinus1);


        //System.out.println("phi " + phi.toString());
        return phi;
    }

    public BigInteger findD(BigInteger e){
        return e.modInverse(findPHI());
    }




    public byte[] dencrypt(byte[] message, BigInteger d, BigInteger n) {
//        System.out.println("Before calc " + byteToString(message));



        BigInteger temp = new BigInteger(message);
        BigInteger m = temp.modPow(d, n);//decrypt

//        System.out.println("My message" + m.toString());

        byte[] b = m.toByteArray();

//        System.out.println("After calc " + byteToString(b));

        return b;
    }

    private static String byteToString(byte[] message) {
        String output ="";
        for(byte i: message) {
            output += Byte.toString(i);
        }
        return output;
    }
//8
}
```

server.java

```java
package gui;

import java.math.BigInteger;

public class Server {

    Persons a;
    Persons b;
    BigInteger[] arr = new BigInteger[2];
    Server(Persons a, Persons b){

        this.a = a;
        this.b = b;

    }




    //get public key of person pretty simple stuff
    public BigInteger[] getPublicKey(Persons p){
        arr[0] = p.getE();
//        System.out.println("get e " + p.getE().toString());

        arr[1] = p.getN();
//        System.out.println("get n " + p.getN().toString());

        return arr;
    }



    //sign the nonce that is sent by encrypting the partner public key
    public BigInteger[] sign(Persons to, Persons p){
        BigInteger[] newArr = getPublicKey(p);
        BigInteger[] fin = new BigInteger[2];

        fin[0] = signEn(to, newArr[0]);
        fin[1] = signEn(to, newArr[1]);

        return fin;//return encrypted signed
    }



    //sign the encryption and send it back to the person
    public BigInteger signEn(Persons to, BigInteger i){
        BigInteger c = i.modPow(to.rsa.e, to.rsa.n);
        return c;
    }

    //server signs the public key, by encrypting it with receiver public key
    //if server  >> alice, encrypt the bob(e, n) with alice's public key
    //then alice decrpyts with her own private key
    //now alice has bobs public key without anyone else knowing
    //alice then proceeds with Needham - Schorlder by sending a nonce
    //

    //alice request, so server sends


}
```

serverGUI.java

```
package gui;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.math.BigInteger;

public class serverGUI {
    private JTextPane cipher;
    private JPanel panel1;
    private JTextPane bobMessage;
    private JTextPane aliceMessage;
    private JTextField aliceInput;
    private JButton sendToBobButton;
    private JLabel publicKeyLabel;
    private JTextPane bobN;
    private JTextPane aliceN;
    private JTextPane cipherN;
    private JLabel recName;
    private JTextPane bobN2;
    private JTextPane cipherN2;
    private JTextPane aliceN2;

    RSA aliceRSA = new RSA();
    RSA bobRSA = new RSA();
    Persons alice = new Persons(aliceRSA);
    Persons bob = new Persons(bobRSA);
    Server ser = new Server(alice, bob);
    byte[] denm;
    byte[] decipher;



    public serverGUI() {
        sendToBobButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {

                messaging(aliceInput.getText(), alice, bob, ser);
                aliceMessage.setText(aliceInput.getText());
```

```java
                cipher.setText(new String(denm));

                bobMessage.setText(new String(decipher));

                publicKeyLabel.setText("(" + ser.getPublicKey(bob)[0] + ", " + ser.getPublicKey(bob)[1] + ")");
                recName.setText("Bob");


            }
        });
    }


    //this method below allows any message sent to any person that is instialised
    //hence why this is a method and can be called in the main function
    public void messaging(String message, Persons p1 , Persons p2, Server ser){


        //get the partner public key by asking the server
        BigInteger[] arr = p1.getPartnerKey(ser, p2);

        //server then signs the public key and sends it back to the user p1
        BigInteger[] bigArr = ser.sign(p1, p2);//server signs for alice

        //person 1 sends to person2 encrypted nounce with signed public key
        byte[] en = p1.sendNounce(p2);
        ///alice front
        aliceN.setText(p1.viewNounce());

        bobN.setText(new String(en));

        //decryption of nounce
        byte[] a = p2.decryptNounce(p1);
//          System.out.println("DE" + new String(a));
        cipherN.setText(new String(a));



        //person 2 sends back a nounce wiht person 1
        byte[] en2 = p2.sendNounce(p1);
        ///alice front
        aliceN2.setText(p2.viewNounce());

        bobN2.setText(new String(en2));

        //decrypts it
        byte[] a2 = p1.decryptNounce(p2);
//        System.out.println("DE" + new String(a));
        cipherN2.setText(new String(a2));

        BigInteger tempE = p1.verify(ser, bigArr[0]);

        //decrtpyes the nounces and also the messages
        denm = p1.serverEncrypt(ser, message.getBytes(), tempE, arr[1]);//encrypt

        //message is shown
        decipher = p2.rsa.dencrypt(denm);





    }

    public void show() {
        JFrame frame = new JFrame("RSA by Ahmed Al-waili, Mahmudul Fakrul, Taha Zacholi");
        frame.setContentPane(new serverGUI().panel1);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```