Public Encryption Implementation:

This implementation has led me down the rabbit hole of practical mathematics. I learned so many interesting things about modular operations, large primes, and how the world's data is kept secure!

Instructions on how to use the program:

1. Run FxApp

2. Press generate keys.

3. Type an integer in the top box you wish to encrypt. The message must be one long integer that is smaller than n in length. Your message MAY begin with zero or consist of all zeroes IF you have padding enabled. Your message can be very short if you have padding enabled. If you do not have padding enabled, a short message (less than 1/3 length n) may or may not be improperly encrypted depending on the value of the public key and if your message begins with zeroes, then the zeroes will be lost.

4. Press encrypt.

5. Get the encrypted message out of the box on the bottom

5. To decrypt an encrypted message encoded using the unique keys in the textboxes on the left, (such as the one you just received), type in the encrypted message into the top right box.

6. Press decrypt (make sure you don't change the status of the padding checkbox or the leftmost textboxes between encrypting and decrypting).

7. The message will be decrypted into the bottom rightmost box using the private key. Of course, in a real implementation, the encryptor and decryptor would be separate programs and a client would only receive the encryptor, public key, and n.

Note: you can absolutely use your own unique values of n, the public key, and the private key, just by typing them into the proper textboxes before encryption and decryption.

Language and Testing: I used java to code this, specifically the BigInteger class, which allows for math with numbers much bigger than the 2^32 limit than primitive int allows. I did a few tests in junit but have removed them from the final source code, as they mostly just made sure certain math functions worked the way they should. Most of my real testing was done through the GUI application itself, hence no jUnit tests will be submitted with this assignment.

How it works:

The difficulty of prime factorization has been a pain to mathematicians for thousands of years, but in public key cryptography, it is the most useful principle in the world.

In public key cryptography, the goal is to have a function that makes a message easy to encrypt, using all public information, but hard to decrypt without special information. Modular exponentiation and primes are the key mathematical concepts to achieving this special one-way function.

The first thing our code does is generate two very large primes. Well, hopefully primes, there is a tiny chance they are composite. To do this, the program searches for random large numbers and uses Fermat's test for Compositeness to guess if the number is prime or not.

The test for compositeness works as follows. A number is selected, ***a***, that is relatively prime to our "prime in question," ***n***. This selection is made by making sure the gcd of ***n*** and ***a*** is one using the Euclidean algorithm. If the gcd(n,a)>1, we can already throw n out as a candidate, since the fact that it is not coprime with a number less than it means that it is definitely not a prime number.

Next, the program checks to make sure the following condition is true:

$$a^{n-1} mod(n) \equiv 1$$

If this condition does not hold, n is definitely composite, and we throw out n and try again for another value.

If the condition does hold, however, n is very likely to be prime.

The BigInteger class has a function that performs this for us, and does even more primality tests besides the one listed above. The chances that the number it selects is actually composite, the method javadoc claims, is not higher than 2^{-100}.

Using this method, we obtain two prime values, p and q. The next thing we do is multiply those primes together to get n:

$$n = p * q$$

N will be vital in both the encrypting and decrypting process. The inability to factor N into its two relative primes without knowing them is the principle that the security of this process depends on.

Now, our program multiplies p-1 and q-1 to get ϕ(n):

$$\varphi(\mathrm{n}) = (\mathrm{p}-1)(\mathrm{q}-1)$$

What is ϕ(n) exactly? ϕ(n) is the amount integers that do not share a factor with n besides 1. Why does ϕ(n) equal (p-1)(q-1). Well applying the ϕ function to primes is easy! It is all integers besides the prime:

$$\varphi(\mathrm{prime}) = \mathrm{prime} - 1$$

Luckily, ϕ is a multiplicative function as well:

$$\varphi(\mathrm{AB}) = \varphi(\mathrm{A}) * \varphi(\mathrm{B})$$

Thus, our ϕ(n) is very simple to compute because we know the prime factorization of n. If we did not know the prime factorization, ϕ(n) would be nearly impossible to compute, which is the whole reason public encryption works.

Once we have ϕ(n), we can connect it to modular exponentiation using Euler's theorem, which states any number that does not share a common factor with n raised to the ϕ(n) power is congruent to 1 mod(n):

$$m^{\varphi(\mathrm{n})} \equiv 1\, mod(n)$$

What can we do with this knowledge? Lets first transform what we know into a more useful form. We know $m^{\varphi(\mathrm{n})} \equiv 1 mod(n)$, and because 1^k always is congruent to one, we can raise the whole thing to the k power and still be congruent to one:

$$(m^{\varphi(\mathrm{n})})k \equiv 1\, mod(n)$$

Or, by the rules of exponents:

$$m^{\mathrm{k}\varphi(\mathrm{n})} \equiv 1\, mod(n)$$

Ok, now let's multiply both sides by m:

$$m * m^{\mathrm{k}\varphi(\mathrm{n})} \equiv m\, mod(n)$$

We can once again, rewrite by the rules of exponents:

$$m^{\mathrm{k}\varphi(\mathrm{n})+1} \equiv m\, mod(n)$$

Wow, so now when we raise m to $k\varphi(n) + 1$ in modulus n, we end up with m again. Next we are going to set $k\varphi(n) + 1$ to two multiplied numbers e*d, our public and private key respectively. Each will act as the inverse of the other when doing exponents in modulus n. Thus if we encrypt a message, m, using m^emod(n), we will be able to decrypt that output message, o, using o^dmod(n).

How do we find e and d? Well, we know:

$$k\varphi(n) + 1 = e * d$$

A way to rewrite this is

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

We will make e equal to the first relative prime to $\varphi(n)$ that is greater than or equal to three. My code does this by starting with three and checking if the gcd of $\varphi(n)$ and 3 equals 1 via the Euclidean algorithm. If it does, then 3 is e! If not, the program increments 3 to 4 and tries again. It repeats this process until it gets a valid result. The public key usually ends up being 3,5,7,11,13, or 17. With e solved, we can now rewrite our equation above in a way that makes it easy to find d. specifically:

$$\varphi(n)x + ey = 1$$

How do we solve this equation? The extended Euclidean algorithm! The value for y that comes out after the extended Euclidean algorithm is used here is our private key, and can be used to undo raising a message to the power of e in modulus n.

My code implements the Euclidean algorithm in a shorthand way that avoids back substitution, essentially performing the substitution as we travel downward in the algorithm.

So, my code now had a value of n, d, and e.

All I do to encrypt an integer message, a, is raise it to the e mod(n) to get our decrypted message, b:

$$a^e mod(n) = b$$

To decrypt, I raise b to the d power mod(n):

$$b^d mod(n) = a$$

And that’s how it works!

Padding:

What to do if the number we are trying to encrypt, c, to the power of the public key is less than n:

$$c^e < n$$

We have a problem here. In this case, knowing e will allow us to undo the encryption to get c without knowing the private key. This is because we can just take the eth root of the “encrypted message” to get our original message.

When we raise c to the public key, we need it to pass n in size so it changes to something else modulus n. Once it passes n, then it becomes impossible to undo in this manner. How can we do this for short messages? Padding!

My padding algorithm works like so. If the message is too small to be encrypted, it tacks on a 1 at the end of the message. Then, it adds zeros until the message is a certain length. Then it encrypts that new message

To decrypt, we must do the opposite when our message reaches us. Our program decrypts the whole message. Then the program takes off zeroes from the message until it reaches a one, which is the stopper I placed earlier. Then it takes off the one at the end and we have our original message!

There is one more case which I pad, when the beginning numbers of the message is zero. To pad in this case, I add on a one to the beginning of the message before encryption. After decryption, I then remove the one at the beginning to get back the original message. This is to make sure that any and all zeros of a message are successfully pushed through.

So a message like 0001024 that is both two small and has zeroes at beginning is transformed into this before encryption:

1**0001024**10000000000000000000000…bunch more zeroes…000

Source code :

BigNumberUtils Class :

```
import java.math.*;
import java.util.*;
public class BigNumberUtils {

/*
 * This method gives the first number 3 or higher that has no common
divisors with the
 * parameter (including itself) besides one.
 */
	public BigInteger spitLowestRelativePrime(BigInteger big){
		boolean b=true;
		BigInteger i=new BigInteger("3");
		BigInteger one=new BigInteger("1");
		while(b){ //this while loop increments i until it finds a number
that works
			if(gcd(i,big).compareTo(one)==0){
				return i;
			}
			i=i.add(one);
		}
		return big;
	}


	/*
	 * This method generates a large number that is probably prime by
using
	 * the biginteger probableprime method. The chance that the number
found is not
	 * prime is tiny (2^-100).
	 */
	public BigInteger generateRandomBigPrime(BigInteger lowerBound, int
upperBound2toTheN){
		Random r= new Random();
		while(true){
			BigInteger
prime=BigInteger.probablePrime(upperBound2toTheN, r);
			if(prime.compareTo(lowerBound)>=0){
				return prime;
			}
		}
	}

/*
 * This method finds the gcd of two bigIntegers via the euclidean
algorithim.
 */
	public static BigInteger gcd(BigInteger p, BigInteger q) {
		BigInteger zero=new BigInteger("0");
		if (q.compareTo(zero)==0)  {
```

```
                  return p;
            }
            return gcd(q, p.remainder(q));
      }

      /*
       *
       * Through online research, I found a shorthand way to implement the
extended
       * euclidean algorithim that avoided back substitution, reducing the
amount
       * of code I had to write. This is NOT copied code,
       * however, the algorithim that I implemented I did not come up with
on my own
       *  and was described in detail by Anthony Vance
       * on youtube. I translated his line of reasoning
       * into code. https://youtu.be/Z8M2BTscoD4?t=10m6s
       */
      public BigInteger bigEuclideanExtendedShortHand(BigInteger phi,
BigInteger e,
                  BigInteger leftTop, BigInteger rightTop, BigInteger
leftBottom,
                  BigInteger rightBottom){
            BigInteger zero=new BigInteger("0");
            BigInteger one=new BigInteger("1");
            BigInteger leftTopDivideLeftBottom=leftTop.divide(leftBottom);
            BigInteger
leftBottomMultiplyLeftTopDivideLeftBottom=leftTopDivideLeftBottom.multiply(l
eftBottom);
            BigInteger
rightBottomMultiplyLeftTopDivideLeftBottom=leftTopDivideLeftBottom.multiply(
rightBottom);
            BigInteger
newLeftBottom=leftTop.subtract(leftBottomMultiplyLeftTopDivideLeftBottom);
            BigInteger
newRightBottom=rightTop.subtract(rightBottomMultiplyLeftTopDivideLeftBottom)
;


            if(newRightBottom.compareTo(zero)<=0){
                  while(newRightBottom.compareTo(zero)<=0){
                  newRightBottom=newRightBottom.add(phi);
                  }
            }

            if(newLeftBottom.compareTo(one)==0){
                  return newRightBottom;
            }
            return bigEuclideanExtendedShortHand(phi, e, leftBottom,
rightBottom, newLeftBottom, newRightBottom);
      }
```

```
}
```

PublicKeyEncryption Class:

```
import java.math.*;
public class PublicKeyEncryption {
	BigNumberUtils utils=new BigNumberUtils();
	BigInteger zero=new BigInteger("0");
	BigInteger one=new BigInteger("1");
	int digitsOfPrimes=103; //change this to change the number of digits
of the primes that are generated.

	double
base2PrimeDigitsAsDouble=(digitsOfPrimes*Math.log(10))/Math.log(2);/*
	This number translates the number of digits you want to have in your
prime
	into the minimum bitlength a number with such digits can be expressed,
which
	is the parameter the probablePrime method uses.
	*/

	int base2PrimeDigitsAsInt=(int)base2PrimeDigitsAsDouble; //casts the
double as int

	BigInteger lowerBoundForPrimes=zero;
	BigInteger p=utils.generateRandomBigPrime(lowerBoundForPrimes,
base2PrimeDigitsAsInt);
	BigInteger q=utils.generateRandomBigPrime(lowerBoundForPrimes,
base2PrimeDigitsAsInt);
	BigInteger n=p.multiply(q);
	BigInteger pMinus1=p.subtract(one);
	BigInteger qMinus1=q.subtract(one);
	BigInteger phi=pMinus1.multiply(qMinus1);
	BigInteger e=utils.spitLowestRelativePrime(phi);
	BigInteger d=utils.bigEuclideanExtendedShortHand(phi, e, phi, phi, e,
one);//d is found by using the euclidean algorithim on e and phi

	public BigInteger getN(){
		return n;
	}

	public BigInteger getPublicKey(){
		return e;
	}

	public BigInteger getPrivateKey(){
		return d;
	}

	/*
```

```
	 * This is kind of amazing. All the encryption really does is raise 
the message
	 * to the e power mod n. That's it. Yet, there is no way to undo this
	 * function without knowing d, which is obtained by knowing phi which 
is obtained by
	 * knowing the prime factorization of n.
	 */
	public BigInteger encrypt(BigInteger encryptThis, BigInteger 
publicKey, BigInteger primeMultiple){
		BigInteger result=encryptThis.modPow(publicKey, primeMultiple);
		return result;
	}

	/*
	 * Decrypting is equally amazing. All we do is raise the encrypted 
message to
	 * d power mod n, and it spits back are original message.
	 */
	public BigInteger decrypt(BigInteger decryptThis, BigInteger 
privateKey, BigInteger primeMultiple){
		BigInteger result=decryptThis.modPow(privateKey, primeMultiple);
		return result;
	}


/*
 * In the case that our message is too short (raising it to the eth power 
does not
 * cross n) or that the message begins with a zero, we must pad the number 
by
 * transforming it into a larger integer by first adding a one to the 
beginning and
 * end and then adding a bunch of zeroes until it is a size that is 
encryptable mod(n)
 */
	public BigInteger pad(String padThis){
		String padThisString=padThis.toString();
		padThisString="1"+padThisString+"1";
		boolean b=true;
		while(b){
			if(padThisString.length()<100){
				padThisString=padThisString+"0";
			}
			else{
				b=false;
			}
		}
		BigInteger padded=new BigInteger(padThisString);
		return padded;
	}


	/*
	 * Just undoes the padding to get back the proper original message
	 * after a message is decrypted. First it removes
	 * the zeroes, then the one at the front and back of the number.
	 */
```

```
      public String dePad(BigInteger dePadThis){
            String dePadThisString=dePadThis.toString();
            boolean b=true;
            while(b){
                  if(dePadThisString.charAt(dePadThisString.length()-
1)=='0'){
                        dePadThisString=dePadThisString.substring(0,
dePadThisString.length()-1);
                  }
                  else{
                        b=false;
                  }
            }

      dePadThisString=dePadThisString.substring(1,dePadThisString.length()-
1);
            return dePadThisString;
      }



}
```

FxApp Class

```
import javafx.scene.Scene;
import javafx.scene.layout.*;

import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.stage.Stage;

/*the app itself, what you should be running
*all the code here is just junk to load the fxml file which
*I setup using scene builder
 */

public class FxApp extends Application {
      private Stage primaryStage;
      private BorderPane mainLayout;

      @Override
      public void start(Stage primaryStage) throws IOException {
            this.primaryStage=primaryStage;
            this.primaryStage.setTitle("Public Key Encrypter");
            showMainView();
      }

      private void showMainView() throws IOException{
            FXMLLoader loader=new FXMLLoader();
            loader.setLocation(FxApp.class.getResource("FxView.fxml"));
            mainLayout=loader.load();
            Scene scene = new Scene(mainLayout);
            primaryStage.setScene(scene);
            primaryStage.show();
      }

      public static void main(String[] args) {
            launch(args);
      }
}
```

Controller for app

```
import java.math.*;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;

/*
 *
 * When using scenebuilder, it is necessary to make a controller class to
handle
 * the code that actually executes when different
 * things in your gui are activated. Each object here
 * is connected to some object in the fxml file, ad changing
 * it here changes it in the actually gui.
 */

public class GUIController {
      BigNumberUtils utils=new BigNumberUtils();
      PublicKeyEncryption keys;

      @FXML
      private Button generateButton;

      @FXML
      private Button encryptButton;

      @FXML
      private Button decryptButton;

      @FXML
      private TextArea NTextArea;

      @FXML
      private TextArea ETextArea;

      @FXML
      private TextArea DTextArea;

      @FXML
      private TextArea encryptThisMessage;
	@FXML
	private TextArea encryptedMessage;

	@FXML
	private TextArea decryptThisMessage;

	@FXML
	private TextArea decryptedMessage;

	@FXML
	private CheckBox padNumbers;

	@FXML
	private CheckBox breakApartLarge;


	/*
	 * WHen the generate keys button is pressed, it instantiates a new
	 * publicEncryption object which has unique n, public, and private
keys.
	 * It then fills the respected textboxes with those values.
	 */
	@FXML
	void generateButtonPressed(ActionEvent event){
		keys=new PublicKeyEncryption();
		BigInteger n=keys.getN();
		BigInteger e=keys.getPublicKey();
		BigInteger d=keys.getPrivateKey();
		String nAsString=n.toString();
		String eAsString=e.toString();
		String dAsString=d.toString();
		NTextArea.setText(nAsString);
		ETextArea.setText(eAsString);
		DTextArea.setText(dAsString);

	}

	/*
	 * WHen encrypt is pressed, it uses the keys in the left textboxes
	 * to encrypt the number entered by the user using c^e mod(n)
	 * If pad is selected, it pads numbers that are too short.
	 */
	@FXML
	void EncryptButtonPressed(ActionEvent event){
		String nAsString=NTextArea.getText();
		String eAsString=ETextArea.getText();
		String toEncryptString=encryptThisMessage.getText();
		BigInteger n=new BigInteger(nAsString);
		BigInteger e= new BigInteger(eAsString);
		BigInteger toEncrypt=new BigInteger(toEncryptString);
		if(padNumbers.isSelected()){

			toEncrypt=keys.pad(toEncryptString);

		}
		BigInteger encrypted=keys.encrypt(toEncrypt, e, n);
		String encryptedString=encrypted.toString();
```

```
            encryptedMessage.setText(encryptedString);
      }


      /*
       * WHen decrypt button is pressed it uses the private key on the left
       * to decrypt what the user has entered using c^d mod n. If depadding
is
       * selected, it will remove all extra stuff that the padding
       * algorithim has added to return the message to normal.
       */
      @FXML
      void decryptButtonPressed(ActionEvent event){
            String nAsString=NTextArea.getText();
            String dAsString=DTextArea.getText();
            String toDecryptString=decryptThisMessage.getText();
            BigInteger n=new BigInteger(nAsString);
            BigInteger d=new BigInteger(dAsString);
            BigInteger toDecrypt=new BigInteger(toDecryptString);
            BigInteger decrypted=keys.decrypt(toDecrypt, d, n);
            String decryptedString="";
            if(padNumbers.isSelected()){
                  decryptedString=keys.dePad(decrypted);
            }
            else{
                  decryptedString=decrypted.toString();
            }
            decryptedMessage.setText(decryptedString);
      }


}
```

Fxml Document to configure app gui

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="666.0" prefWidth="1000.0"
xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="GUIController">
   <top>
      <ToolBar prefHeight="40.0" prefWidth="200.0"
BorderPane.alignment="CENTER">
        <items>
          <Button fx:id="generateButton" mnemonicParsing="false"
onAction="#generateButtonPressed" text="Generate Keys" />
```

```
            <CheckBox fx:id="padNumbers" mnemonicParsing="false"
selected="true" text="Pad Numbers Too Small Or Numbers That Begin With Zero"
/>
            <CheckBox fx:id="breakApartLarge" mnemonicParsing="false"
opacity="0.0" selected="true" text="Break Apart Numbers Too Big" />
        </items>
      </ToolBar>
   </top>
   <left>
      <FlowPane prefHeight="200.0" prefWidth="200.0"
BorderPane.alignment="CENTER">
         <children>
            <Label text="N(Multiple of Two Huge Primes)">
               <FlowPane.margin>
                  <Insets top="10.0" />
               </FlowPane.margin>
            </Label>
            <ScrollPane prefHeight="188.0" prefWidth="200.0">
               <content>
                  <TextArea fx:id="NTextArea" prefHeight="177.0"
prefWidth="196.0" text="This number is the multiple of two large primes. It
is the modulus used in both encrypting and decrypting and is made public."
wrapText="true" />
               </content>
            </ScrollPane>
            <Label text="Public Key">
               <FlowPane.margin>
                  <Insets top="15.0" />
               </FlowPane.margin>
            </Label>
            <ScrollPane layoutX="10.0" layoutY="27.0" prefHeight="171.0"
prefWidth="200.0">
               <content>
                  <TextArea fx:id="ETextArea" prefHeight="177.0"
prefWidth="186.0" text="This number is the smallest relative prime to phi, a
number that equals (p-1)(q-1) where p and q are the primes that multiply to
make n." wrapText="true" />
               </content>
            </ScrollPane>
            <Label text="Private Key">
               <FlowPane.margin>
                  <Insets top="15.0" />
               </FlowPane.margin>
            </Label>
            <ScrollPane layoutX="10.0" layoutY="121.0" prefHeight="167.0"
prefWidth="200.0">
               <content>
                  <TextArea fx:id="DTextArea" prefHeight="177.0"
prefWidth="188.0" text="This number is the inverse of the public key in
modulus(n). It is found by performing the extended euclidean algorithim on
phi and the public key. It is used to undo the encryption that results in
raising a message to the e power in moduls n.  It is nearly impossible to
figure out without knowing the prime factorization of n.This stays private
and does not get sent out to anyone. Only the person recieving encrypted
data knows this value. to recieve the message." wrapText="true" />
               </content>
            </ScrollPane>
         </children>
         <BorderPane.margin>
            <Insets left="20.0" />
         </BorderPane.margin>
      </FlowPane>
   </left>
   <center>
      <GridPane>
         <columnConstraints>
            <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" />
         </columnConstraints>
         <rowConstraints>
            <RowConstraints maxHeight="391.0" minHeight="10.0"
prefHeight="35.0" vgrow="SOMETIMES" />
            <RowConstraints maxHeight="429.0" minHeight="10.0"
prefHeight="190.0" vgrow="SOMETIMES" />
            <RowConstraints maxHeight="285.0" minHeight="10.0"
prefHeight="110.0" vgrow="SOMETIMES" />
            <RowConstraints maxHeight="242.0" minHeight="10.0"
prefHeight="30.0" vgrow="SOMETIMES" />
            <RowConstraints maxHeight="258.0" minHeight="10.0"
prefHeight="258.0" vgrow="SOMETIMES" />
            <RowConstraints minHeight="10.0" vgrow="SOMETIMES" />
         </rowConstraints>
         <children>
            <Label alignment="CENTER" contentDisplay="CENTER"
prefHeight="21.0" prefWidth="595.0" text="Type Message You Want to Encrypt
Here" textAlignment="CENTER" wrapText="true" />
            <ScrollPane prefHeight="311.0" prefWidth="595.0"
GridPane.rowIndex="1">
               <content>
                  <TextArea fx:id="encryptThisMessage" prefHeight="183.0"
prefWidth="507.0" text="Enter an integer here that you want to encrypt. The
encryption algorithim will use the public key and n in the textboxs to the
left. If you know how to calculate these values for yourself, feel free to
do so and the program can encrypt using your values. This program can come
up with random very large keys by pressing the generate keys button. Leave
padding checked for both the encryting and decrypting process in order to
deal with numbers that are too small to encrypt otherwise. If you wish to
encrypt with your own values, it is recommended you uncheck these boxes."
wrapText="true">
                     <padding>
                        <Insets left="15.0" />
                     </padding>
                  </TextArea>
               </content>
               <GridPane.margin>
                  <Insets left="15.0" right="20.0" />
               </GridPane.margin>
            </ScrollPane>
            <Label alignment="CENTER" contentDisplay="CENTER"
prefHeight="17.0" prefWidth="596.0" text="Encrypted Message"
GridPane.rowIndex="3" />
            <Button fx:id="encryptButton" mnemonicParsing="false"
onAction="#EncryptButtonPressed" prefHeight="86.0" prefWidth="277.0"
text="Encrypt" GridPane.rowIndex="2">
               <GridPane.margin>
```

```
                  <Insets left="150.0" right="150.0" top="25.0" />
               </GridPane.margin>
            </Button>
            <ScrollPane prefHeight="124.0" prefWidth="200.0"
GridPane.rowIndex="4">
               <content>
                  <TextArea fx:id="encryptedMessage" prefHeight="428.0"
prefWidth="518.0" text="The integer that comes out here is an encrypted
message. Good luck trying to break it if you don't have the private key! "
wrapText="true" />
               </content>
               <GridPane.margin>
                  <Insets right="20.0" />
               </GridPane.margin>
            </ScrollPane>
         </children>
         <BorderPane.margin>
            <Insets left="15.0" />
         </BorderPane.margin>
      </GridPane>
   </center>
   <right>
      <FlowPane prefHeight="200.0" prefWidth="200.0"
BorderPane.alignment="CENTER">
         <children>
            <Label alignment="CENTER" contentDisplay="CENTER"
prefHeight="36.0" prefWidth="205.0" text="Paste Encrypted Message Here"
textAlignment="CENTER" wrapText="true">
               <FlowPane.margin>
                  <Insets right="15.0" top="15.0" />
               </FlowPane.margin>
            </Label>
            <ScrollPane prefHeight="173.0" prefWidth="200.0">
               <content>
                  <TextArea fx:id="decryptThisMessage" prefHeight="177.0"
prefWidth="187.0" text="Enter an encrypted message here and press decrypt to
undo it using private key and n from the textboxes to the left."
wrapText="true" />
               </content>
            </ScrollPane>
            <Button fx:id="decryptButton" mnemonicParsing="false"
onAction="#decryptButtonPressed" prefHeight="82.0" prefWidth="168.0"
text="Decrypt">
               <FlowPane.margin>
                  <Insets left="15.0" right="20.0" top="20.0" />
               </FlowPane.margin>
            </Button>
            <Label alignment="CENTER" contentDisplay="CENTER"
prefHeight="17.0" prefWidth="205.0" text="Decrypted Message">
               <FlowPane.margin>
                  <Insets bottom="10.0" top="8.0" />
               </FlowPane.margin>
            </Label>
            <ScrollPane prefHeight="252.0" prefWidth="200.0">
               <content>
```

```
                  <TextArea fx:id="decryptedMessage" prefHeight="260.0"
prefWidth="187.0" text="The message that prints out here will be the
decrypted message." wrapText="true" />
               </content>
            </ScrollPane>
         </children>
         <BorderPane.margin>
            <Insets right="15.0" />
         </BorderPane.margin></FlowPane>
   </right>
</BorderPane>
```