Advanced Security
Lab 8

Student Name: Jonathan Riordan
Student ID: C13432152

Part 1
The code:

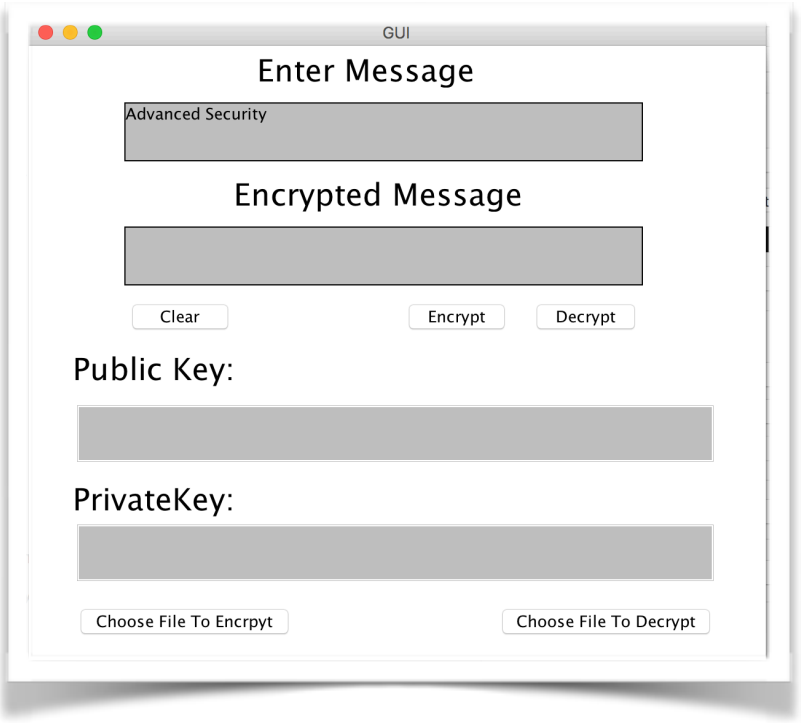
```
# Diffie-Helman

p = 23    # publicly known
g = 5    # publicy known

a = 6    # only Allice
b = 15    # only Bob knows this

def mod(num, g):
    A = (g ** num) % p
    return A


numberA = mod(a,g)
numberB = mod(b, g)


s = mod(a, numberB)
s2 = mod(b, numberA)

print "Publicly shared prime ", p
print "Publicly shared base", g

print "--------------------"
print "Alice sends over public channel: ", numberA
print "Bob sends over public channel: ", numberB
print "--------------------"

print "Alice shared secret ", s
print "Bob shared secret ", s2
```

The output of running my code:

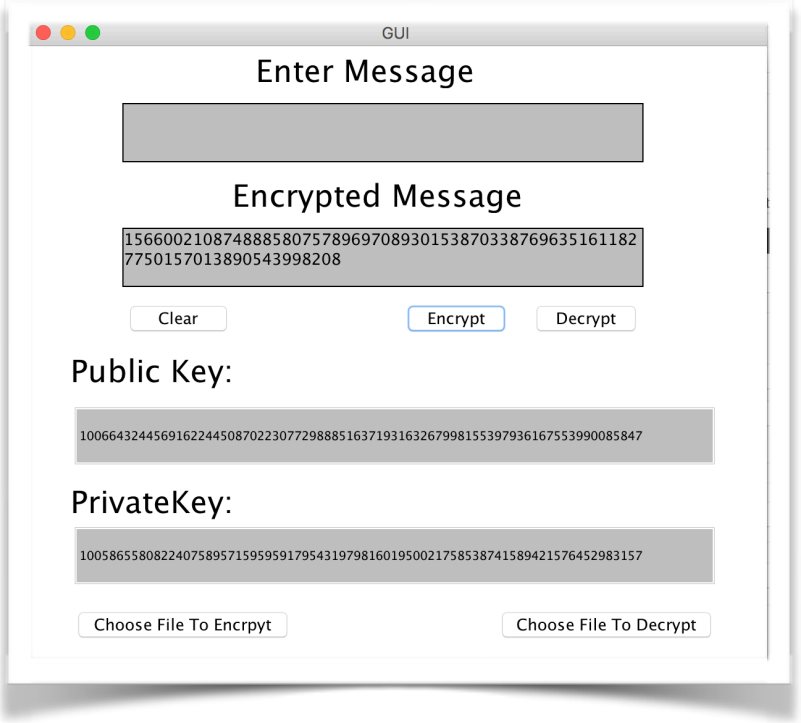
```
Publicly shared prime  23
Publicly shared base 5
--------------------
Alice sends over public channel:  8
Bob sends over public channel:  19
--------------------
Alice shared secret  2
Bob shared secret  2
```

Part 2.
I did the RSA algorithm in Java last year. The submission I sent is go the RSA running that uses a GUI. The user can type in a message and encrypt the message. The public and private keys will be displayed to the user. A user also has the ability to encrypt and decrypt a file. When a user encrypts a file, the keys get written to a text file.

The output of the code is in the screenshots below, I created a GUI for the user to enter in a message.



When the user clicks “encrypts”, the message is encrypted and the cipher text is displayed along with the keys. The user can click decrypt and the message will be decrypted. BigIntegers were used within the code.



The code in the PrimeGenerator file is:

```
/* Assignment for Security module
 * Jonathan Riordan
 * C13432152
 */
```

```
package ie.dit;

import java.math.BigInteger;
import java.security.SecureRandom;


public class PrimeGenerator {
	int divisor = 1;
	BigInteger p, q, n, k, d, z, e;
	private SecureRandom r = new SecureRandom();
	private static SecureRandom r2 = new SecureRandom();
	private static int bitSize = 128;
	BigInteger gcd;

	PrimeGenerator() {

		p = generatePrime(divisor);
		q = generatePrime(divisor);


		n = p.multiply(q);

		z = (p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE)));

		divisor = 2;
		e = generatePrime(divisor);

		gcd = GCD(z,e);

		while(gcd.compareTo(BigInteger.ONE) > 0 && e.compareTo(z) < 0) {
			e.add(BigInteger.ONE);
		}

	//d = modInverseFunction(e,z);
	d = e.modInverse(z);
	}

//	public static BigInteger modInverseFunction(BigInteger e, BigInteger z) {
//		int comp;
//		BigInteger x = new BigInteger("0");
//		BigInteger y = new BigInteger("1");
//		BigInteger p = new BigInteger("1");
//		BigInteger q = new BigInteger("0");
//		BigInteger quot, temp;
//
//		while(!z.equals(BigInteger.ZERO)) {
//			quot = e.divide(z);
//			temp = e;
//			z = temp.mod(z);
//
//			temp = x;
```

```
//                  x = p.subtract((quot.multiply(x)));
//
//
//                  y = q.subtract((quot.multiply(y)));
//
//          }
//
//          return z;
//      }

        // method to get gcd of two numbers.
        public static BigInteger GCD(BigInteger z, BigInteger e) {

                int compare;

                if(z.equals(BigInteger.ZERO)) {
                        return e;
                }

                BigInteger temp = z.mod(e);
                z = e;
                e = temp;

                while(!e.equals(BigInteger.ZERO)) {
                        temp = z.mod(e);
                        z = e;
                        e = temp;
                }

//
//              while(!e.equals(BigInteger.ZERO)) {
//                      compare = z.compareTo(e);
//                      if(compare == 1) {
//                              z = z.subtract(e);
//                      }
//                      else {
//                              e = e.subtract(z);
//                      }
//              }


                return z;
        }

        public static void main(String[] args) {
                System.out.println("GCD");

//              //BigInteger a = BigInteger.probablePrime(bitSize, r2);
                BigInteger a = new BigInteger("7");
//              //BigInteger b = BigInteger.probablePrime(bitSize, r2);
                BigInteger b = new BigInteger("20");
```

```
		//BigInteger gcd = GCD(a,b);
		//BigInteger md = modInverseFunction(a,b);
		System.out.println("A: " + a);
		System.out.println("B: " + b);
		//System.out.println("mod: " + md);
	}

	// Method to create a biginteger. A random number the size if the bit size is stored in
the
	// primeNum variable.This varaible is then passed to the isPrime method to check if it
is prime.
	private BigInteger generatePrime(int divisor) {
		boolean prime = true;
		BigInteger primeNum = null;

		while(prime == true) {
			primeNum = new BigInteger(bitSize / divisor, r);
			//System.out.println(primeNum.toString());

			if(numberPrime(primeNum)) {
				break;
			}

		}
		return primeNum;
	}



	// Checks to see if the number is prime with a certainity of 1.
	// if it is, return true
	// else it return false.
	private boolean numberPrime(BigInteger primeNum) {
		if(primeNum.isProbablePrime(1)) {
			return true;
		}
		else {
			return false;
		}
	}



	// the keys for the RSA for the program
	public BigInteger[] keys() {
		BigInteger[] keys = new BigInteger[3];
		// public key n
		keys[0] = n;

		// private key d
```

```
            keys[1] = d;

            // public key component
            keys[2] = e;

            return keys;
        }


}
```

The code in the RSA class is:

```
/* Assignment for Security module
 * Jonathan Riordan
 * C13432152
 */

package ie.dit;

import java.math.BigInteger;

public class RSA {

        public static BigInteger[] RSA_Keys = null;
        static byte[] encrypted;
        static BigInteger E;
        static BigInteger D;
        static BigInteger N;

        // Generate new keys.
        public RSA() {
                PrimeGenerator generator = new PrimeGenerator();
                RSA_Keys = new BigInteger[3];
                RSA_Keys = generator.keys();

                N = RSA_Keys[0];
                D = RSA_Keys[1];
                E = RSA_Keys[2];
        }


        // Method to create new keys and encrypt message and return a biginteger.
        public static BigInteger encyptFunction(BigInteger message)
        {
                PrimeGenerator generator = new PrimeGenerator();
                RSA_Keys = new BigInteger[3];

                RSA_Keys = generator.keys();

                N = RSA_Keys[0];
```

```
            D = RSA_Keys[1];
            E = RSA_Keys[2];


            return message.modPow(RSA_Keys[2], RSA_Keys[0]);
        }

        // Decrypt biginteger and return byte array
        public static byte[] decryptionFunction(BigInteger encrypted)
        {

            return encrypted.modPow(D, N).toByteArray();
        }

        // method to decrypt file, the encrypted message and keys are passed in and are
used to decrypt.
        public static byte[] decryptionFileFunction(BigInteger encrypted, BigInteger D,
BigInteger N)
        {

            return encrypted.modPow(D, N).toByteArray();
        }
}
```