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:

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 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):

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:

Luckily, φ is a multiplicative function as well:

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):

What can we do with this knowledge? Lets first transform what we know into a more useful form. We know , and because 1k always is congruent to one, we can raise the whole thing to the k power and still be congruent to one:

Or, by the rules of exponents:

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

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

Wow, so now when we raise m to in modulus n, we end up with m again. Next we are going to set 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 memod(n), we will be able to decrypt that output message, o, using odmod(n).

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

A way to rewrite this is

We will make e equal to the first relative prime to that is greater than or equal to three. My code does this by starting with three and checking if the gcd of 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:

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:

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

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:

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