

Implementation of the RSA Cryptosystem

Bachelor of Science (Honors) in Computing

Patrick McDonnell| Cryptographic Foundations | D00006968

18/04/2018

# Methodology

The methodology used in this project was to utilize Big Integer and its in-built methods in Java. As well as using security. SecureRandom to provide cryptographically sound random numbers for use. In built functions used were modPow, modInverse, probablePrime, multiply and subtract to implement the RSA Cryptosystem.

The first step was to find two prime numbers for p and q. I used the probablePrime method of Big Integer to do this as well as the random number generated. The keySize in the formula is the bit size for the BigInteger number returned. This is decided upon above in the code.

BigInteger p = BigInteger.probablePrime(keySize, random);

Next is to use code to find q. then q and p are checked to see if they are equal if they are then another value for q is found and this goes until the two numbers are not equal.

Next, we need to work out the world in which we are working in or the modulus. This is found by using the formula p \* q.

modulus = p.multiply(q);

next we need to work out ΦN = (p-1)(q-1)

BigInteger phi = p.subtract(one).multiply(q.subtract(one));

Once ΦN is chosen then p and q are destroyed.

We then need to choose e or the public key and make sure that it is co-prime to ΦN. This is done by the code below:

while (!publicKey.gcd(phi).equals(one)); {

publicKey = BigInteger.probablePrime(keySize, random); }

N and e form the public key

Next, we generate d or the private key.

This is generated so that d.e [*三*](https://en.wikipedia.org/wiki/%E4%B8%89) 1 mod((p-1)(q-1)). The code for this si as follows:

privateKey = publicKey.modInverse(phi);

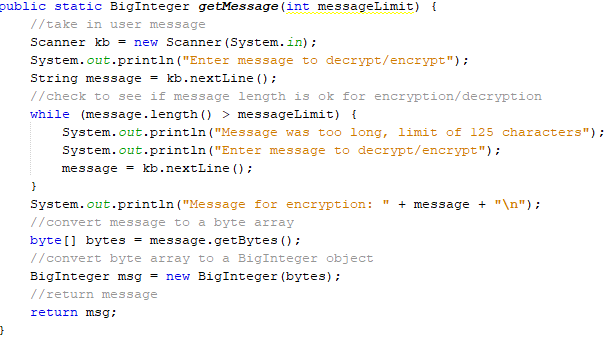
N and d form the private key.

The reason for destroying p and q is that d cannot be determined without them. As they are only known to the person generating the private keys then it is very secure. D or the private key is the only this kept secret N is known to the public as well as e or the public key.

So person A would send the public key which will be made up of e and N to person B, Person B then can use this information to encrypt a message to be sent back to person A, who keeps the private key secret. So only they will be able to work out what was sent by person B.

# Encrypt / Decrypt data

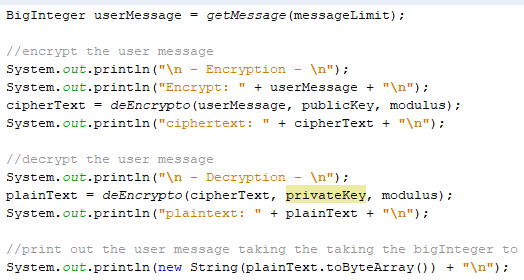
Here we are taking a message from a user and then encrypting and decrypting it using the RSA Cryptographic system.

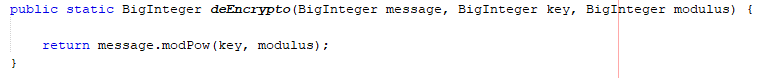


Above is the code that is used for taking in a message form the user for encryption. Because of the bitlength of BigInteger I choose I have limited the amount of characters that can be entered.

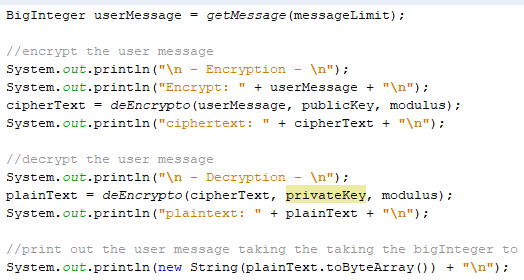
Once the message is taken in it is converted to a byte array.

Then that byte array is converted to a BigInteger object to be sent for encryption and decryption.

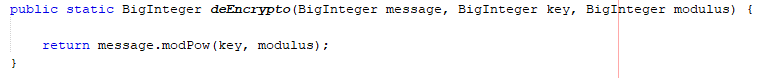


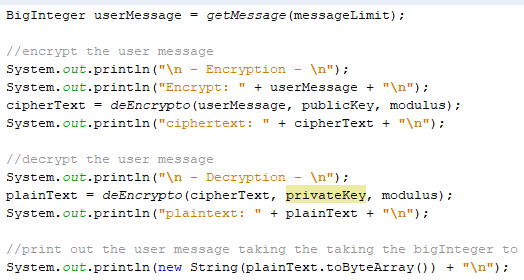
The above code is used for taking the BigInteger returned from the users inputted message - userMessage and then encrypting it. The userMessage is then sent along with e and N to a method for encryption, we use the deEncrypto method, which is located below

This method utilizes the modPow method of the BigInteger class. This returns a BigInteger value of the encrypted message. Giving you cipher text to be sent to a user.

Next up the user would then send the message to the recipient, we will assume this has happened and that we have just received the cipher text.

Once we have the cipher text we apply the decryption algorithm to it using d and N in order to decrypt it and get back the plain text of the message received. Below is the code used, in it we call the deEncrypto method to decrypt.



Lastly we print out the plaintext of the cipher text we received.

This is taking the plaintext BigInteger object converting it to a byte array and then converting that to a Sting object for us to understand.

# Limitations

One of the limitations encountered is the length of the text which I can encrypt. Because I choose the bitlength of the BigInteger it limits the amount with which I can encrypt. Currently it is set to 500, so with that 125 characters can be encrypted with no loss of integrity. When I increased that size to incorporate larger blocks of text the processing slowed way down. So it would be a trade off for processing and speed.

A possible way to over come this would be to break it up into smaller blocks and encrypt it in chunks. Which would then be send to the user. But then making sure all parts would be sent and in correct order would be another hurdle to overcome.

Another limitation is with chat characters it can process, the £ and € symbols will not work with the current model, this could be due to an American ascii bye array or another issue. But it is a very glaring limitation on the current system.

# Problems encountered

C:\Users\patri\OneDrive\Documents\ShareX\Screenshots\2018-04\netbeans64_2018-04-18_13-02-04.pngIssues that were encountered were how to work out ΦN and when trying to initially used it was not picking up 1, as the method subtract would no work with anything other than a BigInteger. This was solved by declaring a BigInteger variable one that held the “1” value for these calculations.

Another issue was how to taking in text based messages with symbols. Initially it was working with only numeric values and String characters were not working. Also negative numbers would not work nor would zero or one encrypt properly, once done they would just give back a cipher text of one or zero which would defeat the point of encrypting it.

This was solved by converting the String to a byte array and then using that to create a BigInteger value for it. It allowed us to encrypt and decrypt a much wider range of values.

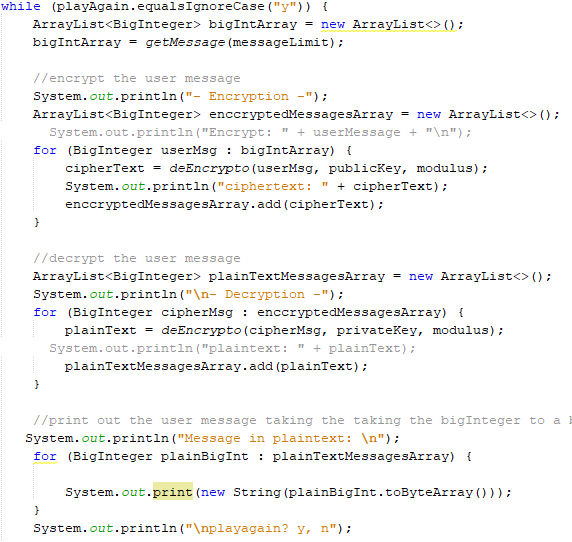
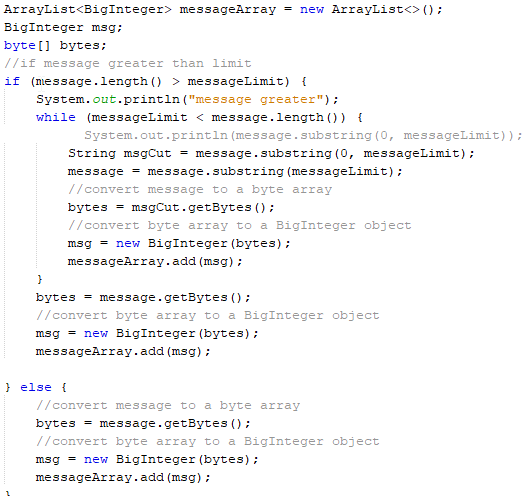
# Overcoming Limitations

A limitation that was prevalent was the inability to handle large messages. So without setting a limit on the encryption message length.

I overcame this issue buy breaking up the original message into smaller chunks of the original limitation size. Then converting them to a BigInteger value and storing them in an arrayList, effectively holding the message in order within the arrayList’s sequence.

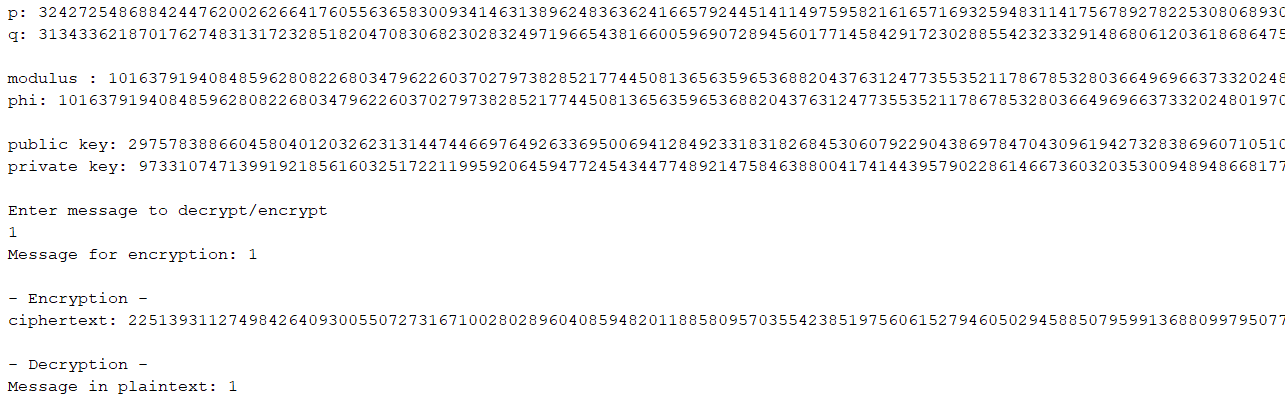
Next was the method of encryption and decryption, I iterated through the arrayList encrypting each element using public key data and then storing them in a new ArrayList<BigInteger> that hold the encrypted BigInteger data.

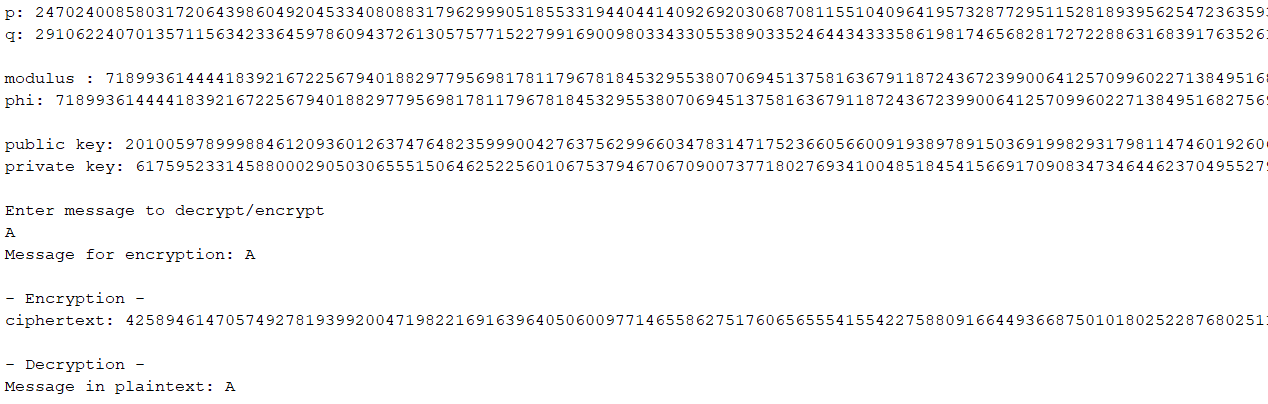
Then onto decryption, the program then iterated through the encrypted ArrayList and using the private key data unencrypted it and stored it in another arrayList that held this unencrypted BigInteger data.  
Then it was a case of printing out the unencrypted arrayList’s data for consumption by the user. Images below show the code used to enact this change to the original program listed above.

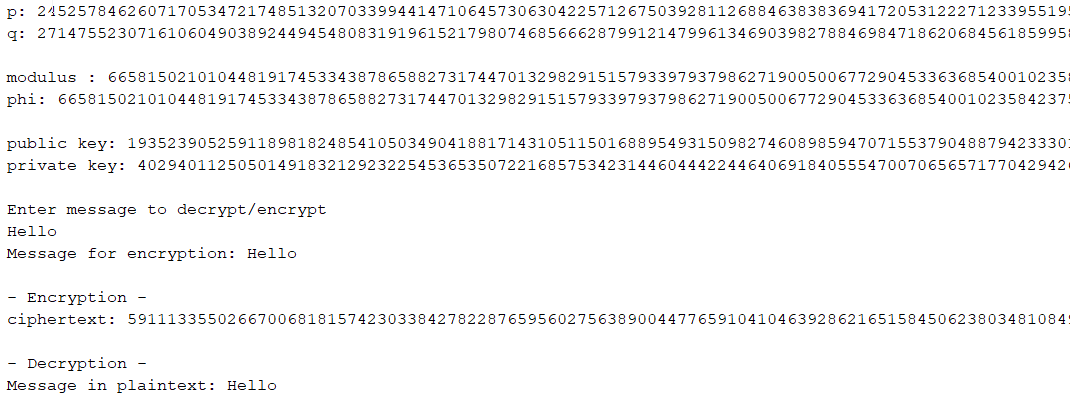


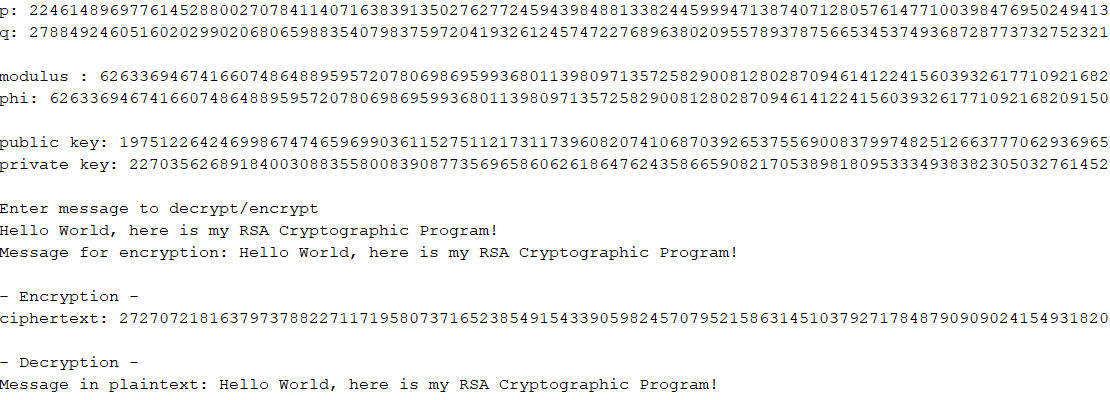
# Output examples

## Single digit / characters

Single character: 1

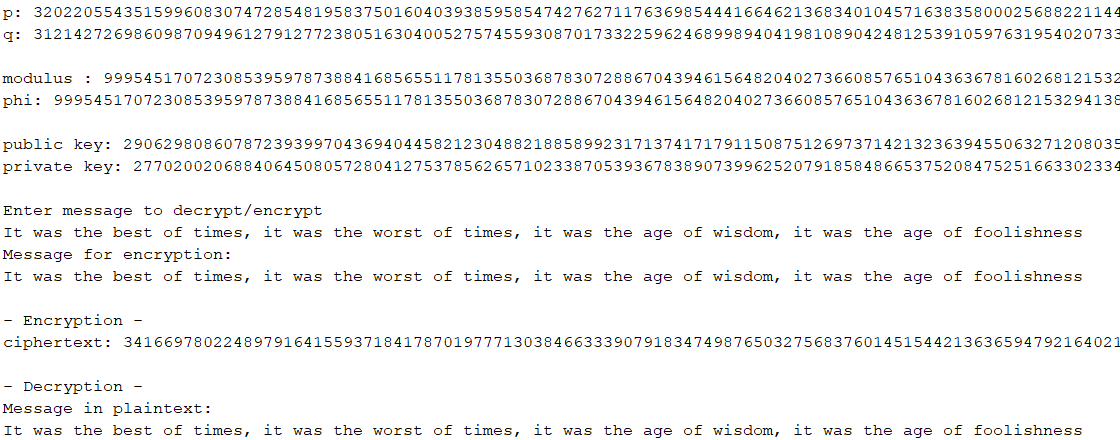
Single character: A

Single word: “Hello”

Short sentence: “Hello World, here is my RSA Cryptographic Program!”

Large piece of text: “It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness”

This was the initial output where there was a limit to the message size



New message where there is no limit “all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,all work and no play makes jack a dull boy,”

