RSA Encryption in Mathematica

Chloe Wawrzyniak

Summer 2018

You should start today's class with a new document. Give your document a title. You'll want to copy the code for fast modular exponentiation from last time into this document - we'll need that function.

1 Example Cryptosystem

Use the fast modular exponentiation program we implemented last class to create two new functions: encrypt1 and decrypt1. These should encrypt and decrypt, respectively, messages according to the example from the morning's class:

Public Key: $n = 2449$, $b = 17$
Private Key: $p = 31$, $q = 79$, $a = 413$

Test your programs by encrypting a number between 1 and 2448. When you decrypt the result, you should get back to the original number you picked.

2 Create Your Own Cryptosystem

Use the function RandomPrime to pick two prime numbers between 1 and 1000. Create an RSA cryptosystem based on those values of p and q. In other words, create two new functions: encrypt2 and decrypt2, which encrypt and decrypt messages according to the public and private keys that you choose based on p and q.

Don't forget to check that these functions operate as expected.

3 RSA and Worded Messages

So far, we've only discussed how to send messages that are numbers. However, what if we want to send a message that's made up of words? Thankfully, we can do that! There are many ways to convert letters to numbers (and vice-versa), and then we can use the system we already have established for sending numeric messages. We're going to use Mathematica's built-in ToCharacterCode and FromCharacterCode functions:

1. Create a simple message by typing the following code:

```
message = ToCharacterCode["hello world"]
```

This should give you a list of numbers. These numbers are the character codes for each of the letters and the space in our above message.

2. Now we want to encrypt these codes by applying encrypt2 to each number. To do this, we use Mathematica's built-in Map function:

   ```
   encryptedmessage = Map[encrypt2, message]
   ```

   This will output a list of numbers, each of which is the encryption of the numbers from the original message.

3. Now, let's make sure our system is working by decrypting the above encrypted message:

   ```
   decryptedmessage = Map[decrypt2, encryptedmessage]
   ```

   This will result in a list of numbers, which should match the numbers in the first output, "message".

4. To convert these numbers back to words, we use the FromCharacterCode function:

   ```
   FromCharacterCode[decryptedmessage]
   ```

   If everything worked properly, this should read out "hello world" on the screen.

4 Signing Messages

Now that we know how to send messages which can't be intercepted, let's talk about how to sign messages so that the recipient can be sure of who sent it. The numbers behind the system are designed so that the encrypt and decrypt functions work as inverses of each other. With the public key, anyone can create the same encryption function. However, only you have access to the private key to make the decryption function. So, to sign a message, we swap the roles of the two functions. At the end of your message, include a code which corresponds to your name after you run it through your decryption function. Then, anyone who receives your message can run that through the encryption function that corresponds to your public key and reveal your name. Since you're the only one who could have coded the message using the private key, the recipient can therefore be sure that you're the one who sent the message.

Work with a partner to send messages back and forth with the crypto-system you created before. Practice signing your messages.