**RSA Lab**

The following page lists the steps from the book for you to implement the RSA encryption. What is not stated is that you will read a file and write a file. Here are some things you might find useful:

**Program 1: Generating keys**

* Step 1 on the following page – You should have coded up Pseudoprime which you can use to get 2 1024 bit numbers – I suggest making those BigIntegers
* Step 2 – BigInteger allows you to multiply
* Step 3 = you just kind of pick a small odd integer for *e*, but make sure it doesn’t evenly divide (p-1)((q-1). Wikipedia suggests that “3” is too small and can break your encryption, but that 65537 is a commonly used value large enough, but still small enough.
* Step 3 is also worded a little strange in that you just pick *e*, but can calculate
* Step 4 – do this math. On page 950 of the book, it says you use Extended-Euclid. Be sure to compute and then use and don’t just use n. Also, you need to use Extended Euclid, not just Euclid as you need the x.
  + Restated, we want to compute
    - Note: That **isn’t** computed using modular exponentiation!
  + The x that Extended Euclid returns is d in this algorithm
    - so pass the Extended Euclid algorithm the and the and save the x as your d
  + One further note, a student stopped by and we tried to make exercise 31.7-1 work with some small numbers. The x came back as -93 instead of 187. We actually want x as a positive number. Because we are using modulo arithmetic, we can add the (280 in this little example) to get to the correct number. I don’t know if you will get back a negative x when you use Extended Euclid, but if you do, be sure to put it into the positives by adding the .
* Steps 5 and 6 – You should publish these keys somehow so you can use them later – Maybe save them to a file?

**Program 2: Using the public key**

* Let’s call this one, use the **public key**, as you can both encrypt and decrypt with the public key
* Now that you have done all that, you can write a separate program to use those keys. That procedure is in the text after step 6. Basically Modular Exponentiation if you can get the file into a BigInteger. I think I made you code that up, but if not it is on page 957 of the book.
* You will need your programs to pull the integers for encryption and decryption out of the files.
* Now we switch gears. The message is in a file, but not as integers. And we don’t want it as integers. We need to treat the file, not as we normally do, but just as a stream of bytes.
* The “Files” class has a static method called readAllBytes(file.toPath()), where the file is a File
* The “Files” class has a method called write that takes a Path and a byteArray
* BigInteger has a constructor that takes a byte array
* BigInteger has a method toByteArray that converts the number into a byteArray

**Program 3 use the private key:** Encrypting/Decrypting Files

* Let’s call this one - use the **private key**, as you can also both encrypt and decrypt with the private key
* Hopefully the only thing that you need to change between programs 2 and 3 is that instead of reading in e, you read in d

**Sanity Check:** Here is something that might help

If p=101, q=103, e=13, and your message = “A” You should be computing the following:

*During Key Generation*

* Instead of computing p and q, assign them as follows:
  + p=101, q=103
* Put the number 13 in the e.txt file. Your code should read this number from a file.
* N gets computed as 10403
* Phi(n) is computed as 10200
* D is computed as -3923,
  + but because it is negative, add Phi(n) which changes D to 6277
* I assume your code has now created the n.txt and d.txt files
  + Open those files with notepad, and be sure that those numbers are correct. Sometimes students write these as bytes, instead

During Encryption/Decryption

* Your code should read in the N, D, and E from text files
* Just for fun, print the N, D, and E and make sure they are the numbers above
  + Students have read these in as bytes, instead of numbers in the past
* When you read in the message, print the BigInteger you create
  + The message “A” (capital, and inside a text file) gets converted to the BigInteger 65
* The encrypted message using the **public key** should be 8692: That should be (65^13)%10403
  + Save that message to a file
  + When you open the file with notepad you should see some type of strangeness like !ô and not 65. I have had students write that int to the file, and not use the bytes
* The decrypted message using the **private key** should be back to 65. That should be (65^6277)%10403
* Whew

I am also providing the larger keys/files I used when I coded this up so that you can use my d, e and n to see if things match

