CS337: Project 2

TA : Taehwan Choi
Release Date: October 6 (Monday)
Due Date: October 22(Wednesday), 11:59pm

1 Introduction

There are two types of encryption algorithms : symmetric and asymmetric. Symmetric encryption algorithm uses a same key for encryption and decryption. Thus, it is symmetric. It is used for a long time and bothers people how to share a key before they want to communicate with each other. On the other hand, asymmetric encryption algorithm uses different keys for encryption and decryption. And it is a remarkable progress in cryptography. The first asymmetric encryption algorithm was invented in 1978 by Ron Rivest, Adi Shamir and Leonard Adleman. Thus, it is named after their initials from each inventor, and becomes RSA.

The RSA algorithm depends on the complexity of prime factorization problem. What is prime factorization problem? Prime factorization is an act of splitting an integer into a set of smaller integers, which form the original integer when they are multiplied together. For example, the factors of 6 are 2 and 3 and the factoring problem is to find 2 and 3 given 6. It is known that 155-digit number RSA-155 was factored after seven months of extensive computations. Thus, it is not trivial to implement the RSA algorithm. However, for this project, we will use a small number to implement the RSA algorithm and have a chance to explore the RSA algorithm. You will not need to use a large number and you will be able to implement your project only with standard primitive types such as Java 'long' data type. There are some data types providing a large number such as Java BigInteger, but you don't have to use those existing libraries. You are only allowed to use Java primitive data types like 'long' data type. You will be penalized if you are caught to use those existing libraries. Remember that this project is designed to implement the RSA algorithm for educational purposes and this will be fun.

2 Deliverables

2.1 Files

You are expected to submit the following files:

1. RSAGenKey.java : Generates a private key
2. RSAEncrypt.java : Encrypts a file
3. RSADecrypt.java : Decrypts a file

4. names.txt : Your CS id, name and UTEID, section number including your partner's information one each line.
   butch, Butch Cassidy, bc0000, 55625
   sundance, Sundance Kid, sk0000, 55625

2.2 Turnin

You will turnin your files as follows:

```
$turnin --submit ctlight project2 RSAGenKey.java RSAEncrypt.java RSADecrypt.java
names.txt
```

1. Please do not turn in a compressed or tarred version of your files.
2. Please do not submit your files with a directory. Please strictly follow the turnin command as it is.
3. Please follow the file names. Your programs might not be graded if you don't follow the file names.
4. No late submission is allowed.
5. Please check your program in CS linux machines since it will be graded in CS linux machines. If your program is not working correctly in CS linux machines, your program will be panalized.

Please include the header for each file as follows:

```
/**
  File: RSAGenKey.java
  Description: Generates a private key of RSA
  Student Name: Butch Casidy, Sundance Kid
  Student UT EID: bc0000, sk0000
  Course Name: CS 337
  Section Number: 55625
*/
```

It is very important to follow the specifications since your project will be graded automatically. It is your responsibility to follow the specifications provided and you will be subject to a penalty if you do not meet the specifications.

3 Key Generation

Recall from the steps of the RSA scheme in section 3.3.2. In order to communicate with other party by RSA, you need to choose two large prime numbers p and q. If you multiply p and q, you will get n. As we discussed, it is hard to factor p and q from n. It is customary to choose e, public key, to be a small value less than n. If you pick e, d is determined uniquely. How do

we determine d? You learned the Extended Euclid Algorithm in class and you will implement the program, "RSAGenKey.java" to compute d, public key given p, q, and e.

For the purpose of this project we will restrict n to be between 2^{24} and 2^{30}. This restriction will allow you to do all the required computations for encryption and decryption using just the Java built-in *long* data type.

Submission

Your program, "RSAGenKey.java", takes input as p, q, and e as command line as follows:
`java RSAGenKey p q e`
The output of your program will be as follows:
`n`
e
d

The output must be **one in each line**. A sample input/ouput of the program is as follows:

```
$java RSA 6551 4733 8311
31005883
8311
11296191
```

4 File encryption

For this part of the project, you will write a Java program to encrypt an arbitrary file. Note that the RSA algorithm specifies how to encrypt a single number ($< n$). To encrypt a file, it is sufficient to break up the file into blocks so that each block can be treated as a number and encrypted by the RSA algorithm. For this project, you will use a block size of 3 bytes. To treat each block as a number, simply concatenate the bit representation of the 3 bytes to form a single number. Note that this number will be between 0 and 2^{24} and so, will be less than n (why?), allowing us to use the RSA encryption algorithm for a single number.

After encrypting the block, the encrypted number has to be written out to the output file. This number is between 0 and n, and therefore, potentially between 0 and 2^{30}. To write out this number, simply break the number into 4 bytes based on the bit representation of the number and write out the 4 bytes in order from *left to right*.

Let us now walk through the entire process with an example. Suppose the input file consists of the following bytes (in order) : 75, 34, 107, 23 .... To generate the first encrypted number, pick up the first block consisting of three bytes : 75, 34, 107. Form the number from this block by concatenating the bit representations of these bytes - 01001011, 00100010, 01101011. This number is 00000000 01001011 00100010 01101011. Now, encrypt this number by the RSA algorithm to get the number (say) 01011000 00110110 00001000 10001100. To write out this encrypted number to the file, break it up into 4 bytes, reading the bits from left to right - 01011000, 00110110, 00001000 and 10001100. Write out these 4 bytes in this order to the output file.

Remark: It is possible that the input file is not an exact multiple of 3 bytes. In such a case you may consider the last or the last two bytes to be null, i.e. as '00000000'.

Submission

The encryption algorithm should be runnable from a file named "RSAEncrypt.java" (that is, this file should have the *main* method). The program will be called with a plaintext file and a key file as the arguments. The key file will contain n,e and d, one each on a line. So for this part you should use (e, n) to encrypt the message.

The program will be invoked as :

```
$java RSAEncrypt file key.txt
```

where `file` is the name of the file to be encrypted.

The encrypted file that the program generates **should** be named file.enc where "file" is a plaintext.

5 File decryption

For this part of the project, you will write a program to decrypt the encrypted file that has been generated by the encryption procedure described earlier.

This procedure is the exact reverse of the encryption procedure. You will read from the encrypted file in blocks of 4 bytes, with each set of 4 bytes forming an encrypted number. The RSA decryption algorithm specifies how to decrypt an encrypted number. This will give you the number that was formed from the original plaintext file. Now, remember that this number was formed by concatenating 3 bytes from the plain text file. To get back those 3 bytes from this number, you will have to pick out the lower order 3 bytes from this number and write them out in order to the output file (the topmost byte of this number will always be 0, can you see why?).

Let us walk through this procedure with an example. Suppose the encrypted file consists of the following bytes : 88, 54, 8, 140, ... (these were the same bytes that were written out in the example for the encryption procedure). The encrypted number is formed by concatenating the bit representations of these numbers together - 01011000 00110110 00001000 10001100. Decrypting this number by the RSA algorithm gives the original number 00000000 01001011 00100010 01101011. This number is then split into the original 3 bytes by taking the lower order 3 bytes in the same order 01001011, 00100010, and 01101011.

Submission

The decryption algorithm should be runnable from a file named "RSADecrypt.java" (that is, this file should have the *main* method). The program will be called with the input encrypted file and the key file as arguments. The key file will be in the same format as described previously.

The program will be invoked as :

```
$java RSADecrypt file.enc key.txt
```

where file.enc is the name of the file to be decrypted.

The decrypted file that the program generates **should** be named "file.dec".

6 Notes

Note that your program should stick to the specifications laid out in this description *in all respects*, from the order in which bytes are written to the order of the command line arguments of the program. **Your encryption routines should work correctly when used with other correct**

decryption routines and vice versa. It is therefore necessary for you to follow every specification exactly.

The RSA encryption and decryption algorithms involve modular exponentiation. Straightforward exponentiation will require you to manage really huge numbers that will be outside the limits of the basic Java data type 'long'. **For this reason, it is necessary that you use the algorithm for fast modular exponentiation that is described in the class notes (Section 3.3.2)**.

It is not necessary for your programs to handle erroneous input formats. In cases like these, or say, when an attempt is made to decrypt a file with a key file different from that used to encrypt the file, it is okay for your program to crash. We will not test your code for such cases.

Enjoy the project!!