You must work either on your own or with one partner. You may discuss background issues and general solution strategies with others, but the project you submit must be the work of just you (and your partner). If you work with a partner, you and your partner must register as a group in CMS and submit your work as a group.

Objectives

In this project, you will learn to write Java programs in the “procedural” style—the way we have written MATLAB programs up to this point. (So this is not object-oriented programming yet!) Along the way we want to continue our exploration into security issues in computing by introducing RSA encryption, considered to be the strongest encryption algorithm today! Think about the credit card number that you enter onto a website when you make a purchase on the Internet... There are three separate questions in this project. You will start with a simple program with one class and one method only to work with arithmetic operators, methods in the Math class, type conversion, and printing. The second question is the mini-project on RSA encryption for which you will write some of the static methods in a class. The last part asks you to learn about the Java API (Application Programming Interface) by looking through parts of the on-line Java API documentation.

1 RSA Public Key Encryption

We learned about the simple substitution cipher in Project 3 and even worked on “cracking” it using frequency analysis. You have discovered that the character-by-character encryption method is susceptible to statistical attacks and therefore is not suitable for applications that require a high level of security, such as electronic commerce.

RSA encryption is widely considered to be the strongest encryption algorithm today. RSA is the initials of the inventor of the algorithm, published in 1978: Ronald Rivest, Adi Shamir, and Len Adleman. The RSA encryption algorithm is used almost everywhere on the internet where secure data transfer is required. The general idea is that it is easy to multiply numbers, even large ones, with a computer, but factoring numbers is hard when the numbers are very big. Suppose I multiply two numbers and tell you the product, say, 9123456780. Can you guess what those two numbers are? Sure, you can try all the possible combinations if you have a computer nearby and you can program, but this is going to take many trials. What if the number I give you is hundreds of digits long, not just a “wimpy” 10-digit number? It will take a computer many, many lifetimes of the universe to factor that number! A 7-bit number is at most 3-digit long, but in real world applications, the numbers chosen to generate the RSA keys are very large, typically 1024 bits or longer. Experts believe that keys generated by 4096-bit numbers cannot be factored in any foreseeable future, even taking into account the growth of computer speed. (Just in case you’re wondering... type int, or even long, in Java isn’t enough to store such large numbers. One would need an array to store an arbitrarily long integer, to be discussed later in the semester).

Here is how it goes. I find two huge prime numbers, \( p \) and \( q \), that are hundreds of digits long. \( p \) and \( q \) form the basis of my private key—I don’t tell anyone about those two numbers. The product \( pq \) is some even bigger number and is the basis of the public key that I tell you (or the world) about. You, or anyone who knows the public key, use the public key to encrypt your message that you will send to me, and I can decrypt the message using my private key. Anyone who has intercepted your encrypted message cannot crack the encryption even when they know what the public key is—there isn’t a way to factor huge numbers (in our lifetimes) to recover the private key for decryption. Below we present the formal algorithm. Don’t be scared by the mathematical terms—they’re just names! Focus on the individual steps. We are providing the methods for the more complicated calculations that are needed, so you will simply call those provided methods.

The algorithm

RSA key generation can be performed in a 5-step procedure:

1. Randomly choose two large prime numbers \( p \) and \( q \), such that \( p \neq q \).
2. Compute the product \( c = pq \). This is the modulus cipher.
3. Compute the totient of \( c \), \( \phi(c) \). In this case, \( \phi(c) = (p - 1)(q - 1) \).
4. Randomly choose an integer \( e \), such that \( 1 < e < \phi(c) \) and \( e \) is coprime to \( \phi(c) \).
   (Two integers are coprime if their greatest common divisor is 1.)

5. Find positive integer \( d \), such that \( de \equiv 1 \pmod{\phi(c)} \). \( d \) is called the multiplicative inverse of \( e \) modulo \( \phi(c) \).
   You can get \( d \) by finding some integer \( x \) such that \( d = (x \cdot \phi(c) + 1)/e \) is an integer.

Now \( e \) is the public key and \( d \) is the private key. Let integer \( m \) be the original message, or plaintext. (Recall that each character has a numeric equivalent, the ASCII value.) To encrypt \( m \), compute

\[
s = m^e \mod c
\]

where \( s \) is the encrypted message, or ciphertext. To decrypt \( s \), compute

\[
m = s^d \mod c
\]

which gives back the plaintext \( m \). Cool!

**An example**

Suppose the two primes generated are \( p = 67 \) and \( q = 89 \). Then the modulus cipher is \( c = pq = 5963 \). Pick \( e = 91 \) which satisfies the requirement that \( e \) is coprime to \((p-1)(q-1) = 5808\). Find \( d = 4723 \) so that \( de = 4723 \times 91 \equiv 1 \pmod{5808} \). Hence the public key is 91, private key is 4723.

Let the plaintext be \( m = 456 \). To encrypt, compute the ciphertext

\[
s = 456^{91} \mod 5963 = 3734
\]

To decrypt, compute the plaintext

\[
m = 3734^{4723} \mod 5963 = 456
\]

**Key generation, encryption, and decryption**

The file `RSA.java` contains a partially implemented RSA algorithm. However, some methods are incomplete. Your task is to finish the RSA implementation by completing methods `testPrime`, `gcd`, `genKeys`, and `powMod`. Basically, you will write the methods that generate the keys and support encryption and decryption. (We have provided the methods that actually start the encryption the decryption operations.) Even though the given file appears long, you will write only four of the methods and some of them are very short. Make sure that you *read the comment headers of these methods as well as the comments at the top of RSA.java*, so that you don’t miss any important instruction and hints. Modify only methods that are marked with comments “Implement this method”. You may write new methods if you wish to, but it is not a necessity for completing this assignment. In your code, you may (and may only) call methods that are either in RSA.java or in the standard Java library.

The first thing to do after downloading the code is to compile it. You will see that it compiles without error even though some of the methods are incomplete. This is because we put “stubs” in the methods—return statements that match the return type—but the value being returned is only a dummy initialized value. *Since we give you code that compile to begin with, we expect that your submission will at a minimum compile successfully, even if there are some logical errors.* Code that does not compile will incur a significant penalty. This means that (as usual) you should compile/test your code throughout program development, not just at the end after you finish all the methods!

There are some provided methods (e.g., `genPrime` and `genCoPrime`) that you might want to call in your code. So look at them to find out what they do and how to call them. Do not modify these provided methods. For the rest of the given code, you are encouraged to read and understand as much as you can, although you can complete this assignment by reading just the headers and specifications of the provided methods.

The `main` method is provided so that you can test your implementation. Do not modify `main`. To run the program in Dr. Java, compile it and type “java RSA” in the “Interactions” pane at the bottom of the screen. The usage will be printed in the same place where you typed your command. Read the usage so that you know how to run the program with various modes and parameters. (You may need to scroll a bit to see the whole message; alternatively, you can resize the pane by dragging its upper boundary.) Here are a few examples of what to type in the “Interactions” pane:
• To read the information on how to use RSA: java RSA

• To generate a set of keys: java RSA genkeys
  The keys will be shown in the Interactions pane. Suppose they are public key e=7739, private key d=3923, and modulus cipher c=8137.

• To encrypt the message m=456 with the above keys: java RSA enc 456 7739 8137
  The ciphertext s=726 will then be shown in the Interactions pane.

• to decrypt the ciphertext s=726 with the above keys: java RSA dec 726 3923 8137
  The decrypted plaintext m=456 will be shown.

Project 4 Challenge Question—optional

As discussed in the encryption portion of this project, our implementation of RSA encryption using type int forces us to use small primes, so it is possible to crack the private key! One can break the encryption by factoring the modulus cipher c into the original primes p and q and hence recovering the private key d.

Complete the implementation of method crack in RSA.java so that it correctly recovers the private key d from the given public key e and the modulus cipher c. In your code, you may (and may only) call methods that are either in RSA.java or in the standard Java library.

References
http://en.wikipedia.org/wiki/RSA
http://world.std.com/~franl/crypto/rsa-guts.html
http://mathcircle.berkeley.edu/BMC3/rsa/node4.html