CS2112—Spring 2012

Homework 2

Ciphers and Encryption

Due: February 20, 11:59PM

In this assignment you will be building multiple systems for the encryption and decryption of text. The first part will focus on simple ciphers and cipher cracking, while the second part requires you to implement the popular RSA public-key encryption algorithm, which is used everytime you visit an "https:" website. The goal is a command-line interface that can be used to generate, save, and use the ciphers you build. And you should implement the system in a way that uses inheritance to share code between different ciphers.

0 Changes

The homework has been changed since initial release in the following ways: to require implementing Vigenère ciphers, to simplify the required interface hierarchy, and to allow you to implement either of the two factoring algorithms rather than both.

Added command line specifications for the Vigenère cipher.

1 Instructions

1.1 Grading

Solutions will be graded on both correctness and style. A correct program compiles without errors or warnings, and behaves according the the requirements given here. A program with good style is clear, concise, and easy to read.

A few suggestions regarding good style may be helpful. You should use brief but mnemonic variables names and proper indentation. Your code should include comments as necessary to explain how it works, but without explaining things that are obvious.

1.2 Partners

You *must* work alone for this assignment. But remember that the course staff is happy to help with problems you run into. Use Piazza for questions, attend office hours, or set up meetings with any course staff member for help.

2 Familiarizing yourself with the code

2.1 Building the Documentation

Open the provided code in Eclipse. Follow the instructions at

http://www.cis.upenn.edu/~matuszek/General/Pages/eclipse-faq.html#run_javadoc

to build Javadoc for this project from within Eclipse. When the documentation finishes building, you should be able to view all the documentation from your web browser.

We have given you some documentation so that you can get started on the assignment. Feel free to expand upon this documentation in your final submission.

3 Classical Ciphers

3.1 Overview

A cipher is a way to protect a message by changing its letters or characters, so only the desired recipient(s) can read it. The original message is called the *plaintext*, and the transformed message is called the *ciphertext*. Monoalphabetic ciphers provide what is perhaps the most rudimentary encryption. These ciphers create a one-to-one correspondence between letters in the original message and letters in the encrypted message. For example, if we use a cipher where each letter is shifted to the right by one (A $\rightarrow$ B, B $\rightarrow$ C, ..., Z $\rightarrow$ A), we would encode the message CAT as DBU.

3.1.1 The Caesar Cipher

A Caesar cipher is a monoalphabetic cipher that functions by mapping an alphabet to a 'shifted' version of itself. The example in the previous section is a Caesar Cipher with shift parameter 1. A shift parameter of 0 results in the original alphabet. The shift parameter is not limited to values 1–26; it can be any integer.

3.1.2 Random Monoalphabetic Ciphers

The Caesar cipher is a particularly simple example of a *substitution cipher*, a cipher that replaces pieces of text by corresponding (and hopefully different) pieces of text. In a monoalphabetic substitution cipher, each letter (or character) is mapped to an arbitrary distinct letter in a one-to-one fashion. Knowing this substitution, the recipient can then invert it to recover the original plaintext. To make the cipher hard to break, we generate the substitution randomly.

Note that random number generators used in computing are usually not truly random, but rather *pseudorandom*. They are produced by an algorithm called a *random number generator*. For cryptography this is acceptable as long as the adversary has no practical way to distinguish between the pseudorandomness and true randomness. *Cryptographic* random number generators are random number generators with this property.

3.1.3 The Vigenère Cipher

Another way to strengthen the Caesar cipher is to use different substitutions on different letters. The Vigenère cipher[1] was once considered to be unbreakable. Rather than using a uniform shift for all letters, a repeating pattern of shifts is applied. Traditionally, the key is represented as a word, with 'A' representing a shift of 1, 'B' a shift of 2, and so on. Encrypting CATALOG with a key ABC would yield DCWBNRH, because the shifts ABCABCA are applied to the plaintext. The Vigenère cipher defeats simple frequency analysis especially if the key is long.

3.2 Implementation

Your task is to implement a Caesar cipher, a simple substitution cipher, and a Vigenère cipher. Your code will be accessed using the `CipherFactory.java` class. We have provided a Cipher interface in `Cipher.java`. In addition to implementing this interface, your ciphers should extend the abstract class in `AbstractCipher.java`. You should aim for elegant program design that *minimizes the repetition of common code* across similar classes and methods. Toward this end, you may wish to define additional abstract classes.

3.2.1 Letter encoding

It is important to have a consistent standard for the encoding of characters in both the encrypter and decrypter. You should convert all letters to their uppercase equivalents and maintain whitespace—spaces, tabs, and newlines. All other characters should be discarded. Using a Caesar cipher with parameter 1 under this schema, 'Ca, T1' would be encrypted as 'DB U'. You may assume that when decrypting you will only encounter uppercase letters and whitespace.

3.2.2 Saving the Cipher

Saving a substitution cipher to a file is fairly straightforward and relatively insecure. You should simply have the program save the encrypted alphabet, followed by a newline, to the stream. For example, saving a Caesar cipher with shift parameter 1 would create a file that looks like `BCDEFGHIJKLMNOPQRSTUVWXYZA\n`.

4 Cipher Cracking

4.1 Frequency Analysis

Monoalphabetic ciphers are easy to break using frequency analysis. One analyzes the frequency of letters in the target language and in the encoded message. This information can be used to reconstruct the cipher used to encrypt the message.

[1] Not actually invented by Vigenère

4.1.1 Implementation

You need to implement a scanner that will analyze the frequency of letters over multiple texts in the unencrypted language, and then use this analysis to crack messages encrypted with a monoalphabetic substitution cipher; this is done by mapping the two most common letters to each other, then the two second most common to each other, etc. You should do so by completing the methods provided in `FrequencyAnalyzer.java`. Like the encrypter, the FA should keep track of only uppercase English letters and handle other characters appropriately (convert or ignore).

5 RSA Encryption

RSA is probably the most widely used encryption schema in the world today. RSA is believed to be very secure, based on the widely believed assumption that it is difficult to factor large numbers. RSA is a *public-key cipher*: anyone can encrypt messages using the public key; however, knowledge of the private key is required in order to decrypt messages. Public-key cryptography makes the secure Internet possible. Before public-key cryptography, keys had to be carefully exchanged between people who wanted to communicate. Now RSA is routinely used to exchange keys without allowing anyone snooping on the channel to understand what has been communicated.

5.1 The Algorithm

5.1.1 Key Generation

1. Choose two random and prime numbers p and q. These must be kept secret.
2. Compute $n = p \cdot q$.
3. Compute $m = (p-1)(q-1)$, which is called the *totient* of n. It is the number of positive integers less than n that are relatively prime to it. Notice that computing m requires knowledge of p and q.
4. Choose an integer e such that $1 < e < m$ and $\gcd(e, m) = 1$.
5. Compute $d = e^{-1} \bmod m$. That is, d such that $1 = e \cdot d \mod m$. This ensures that raising any number to the $e \cdot d$ power will give the same number back.

The public key is the pair (n, e) and the private key is the pair (n, d).

5.1.2 Encryption

A message s is encrypted as message c via the following formula: $c = s^e \bmod n$.

5.1.3 Decryption

A message c is decrypted as message s via the following formula: $s = c^d \bmod n$.
This works because $(s^e)^d = s \mod n$

5.2 Implementation

5.2.1 Dealing With Large Numbers

RSA involves large numbers, so you should use the class `java.math.BigInteger` for all arithmetic. To generate large prime numbers, you should use the appropriate `BigInteger` constructor with `bitLength` $= 1024$ and `certainty` $= 20$. The numbers generated by this are only 'probably' prime, which is, in both theory and practice, a strong enough statement.

5.2.2 Message Format and Padding

Encryption
The most challenging part of implementing RSA is not the arithmetic (at least with the help of a class such as `BigInteger`). Rather, it is formatting the message so that it can be correctly encrypted. We will encode messages as blocks of 117 bytes, followed by 11 bytes of 'padding'. This means that if the encrypt function is given a String, it should first convert it to an array of bytes, a. We will process this array in x discrete blocks, where $x = \lceil a.length/117 \rceil$. Take each block of this array and copy it into a byte array of length 128. You should set the last byte of the new array to be the difference between 117 and the length of the block copied into it. This will indicate where the data ends during decryption. Any unused bytes in the array (those between the block and its length) should be set to 0.

Next we will turn each of the x byte arrays into an integer, apply the encryption algorithm, and store the result as the string representation of the integer. We will concatenate the results from encrypting each array and separate them by newlines.

Note: This system of dividing up the message into chunks in this manner is strongly discouraged in the RSA documentation. You should remember that you are implementing a slightly simplified version of RSA that maintains its key concepts, but sacrifices its strong security.

Decryption
Decryption is simply the inverse of this operation.

5.2.3 Saving the Keys

The keys, like the cipher schema, can be saved to files.

Public Key
When a public key is stored to a file, the file should contain the decimal representation of n, followed by a newline, followed by the decimal representation of e, and end with a newline.

Private Key
The storage of the private key is the same as the storage of the public key, except for the addition of a third line containing d. More precisely, when a private key is stored to a file, the file should contain the decimal representation of n, followed by a newline, followed by the decimal representation of e, followed by a newline, followed by the decimal representation of d, and end with a newline.

5.3 Cracking RSA

While very secure, RSA, especially given p and q with low bit-length, is crackable. All one has to do is get p and q by factoring n. The most obvious way to do this is to simply try dividing n by every number between 1 and$\sqrt{n}$. (We know that at least one of our factors is at most$\sqrt{n}$.) A faster approach, however, is uses to Pollard's Monte-Carlo factorization algorithm, which you can read about here. You should complete at least one of the factoring methods in `Factorer.java`. If you need help timing your algorithm, `System.nanoTime()` is a good place to start.

You should use your methods to factor various numbers (that are the products of two primes), report the largest number you were able to factor and the time it took in your `README.txt`, and estimate how long it would take to decrypt a message using a 1024-bit key in this manner. You do not have to add this functionality to the command line interface.

Useful Resources

- Class BigInteger
- Class String
- Class Byte

6 Command Line Invocation

The last piece of this program is creating a command line interface. This is different from the console interface created for Homework 1.

```
java -jar <YOUR_JAR> <CIPHER_TYPE> <CIPHER_FUNCTION> <OUTPUT_OPTIONS>
```

Cipher type

There are three diffferent cipher types that we have asked you to implement, and the flags for each of them are as follows. If a user attempts to execute two incompatible actions such as `java jar <your_jar> --random --savePu outfile.pu`, a reasonable warning should be printed to the console (print to `System.out`).

`--classical <cipher_file>`: A monoalphabetic substitution cipher should be loaded from the file specified.

`--caesar <shift_param>`: A Caesar cipher with the given shift parameter should be used for these operations.

`--random`: A random cipher should be generated and used by this program.

`--classicalfa [-t <examples> | -c <encrypted>]`: A classical (monoalphabetic cipher) should be constructed using frequency analysis with the files flagged `-t` listed in examples as the unencrypted language and files tagged `-c` as the encrypted language. You can have any number of `-t` and `-c` flags, and in any order.

`--rsa`: Creates a new RSA cipher

`--rsaPr <file>`: Creates an RSA encrypter/decrypter from the private key stored in the specified file

`--rsaPu <file>`: Creates an RSA (encrypter) from the public key stored in the specified file

`--vigenere <key>`: Creates a Vigenère cipher using the given keyword (given as a string)

`--vigenereL <cipher_file>`: Loads a Vigenère cipher from the given file

Obviously, you would like to do something with your cipher. At most one of the following options may also be specified by the user.

Cipher functions

`--em <message>`:encrypts the given message

`--ef <file>`:encrypts the provided file using the specified cipher scheme

`--dm <message>`:decrypts the given message

`--df <file>`:decrypts the provided file using the specified cipher scheme

Output Options

`--print`: prints the result of the cipher function (if any) to the console

`--out <file>`:prints the result of the cipher function (if any) to the specified file

`--save <file>`: saves the current cipher to the provided file (if the current cipher is RSA, this saves the private key

`--savePu <file>`: if the current cipher is RSA, this saves the public key to the given file

6.1 Examples

- Make a new Caesar cipher with shift parameter 15, apply it to the provided message, output the to 'encr.txt', and save the cipher to 'ca15' (a file).
  `java jar <your_jar> --caesar 15 --em 'ENCrypt Me!' --out encr.txt --save ca15`

- Load the cipher from 'ca15', decrypt the message in encr.txt, and print the result to the console
  ```
  java jar <your_jar> --classical ca15 --df encr.txt --print
  ```
- Create a frequency analyzer using 3 English texts and 1 encrypted text. Use the resulting cipher to decrypt the encrypted text, print the result, and save the cipher.
  ```
  java jar <your_jar> --classicalfa -t moby-dick.txt -c mystery.txt
    -t frankenstein.txt -t macbeth.txt --df mystery.txt --save clasCiph --print
  ```
- Create an RSA encrypter, encrypt the message given, save it to a file, and save the two keys to different files.
  ```
  java jar <your_jar> --rsa --em 'rsa is alright, i guess' --out encr.txt
    --save priv.pr --savePu pub.pu
  ```
- Load an RSA private key, decrypt a message, print it, and save it to a file.
  ```
  java jar <your_jar> --rsaPr --df encr.txt --out decr.txt --print
  ```

7 Extensions

For full credit, you are not required to do anything more than what is specified here. But for good karma you may add additional features. You might allow additional commands, more complex ciphers (there are plenty of systems that can be found pretty easily on the internet), cryptanalysis for Vigenère or other ciphers, a more clever way of storing ciphers in files, randomized padding (rather than consecutive 0s) in RSA, etc. Make sure to document anything you do that goes beyond what is requested, and be especially sure that any extensions you make do not break the required functionality of your program.

8 Submission

You should compress exactly these files into a `zip` file that you will then submit on CMS:

- *Source code*: Because this assignment is more open than the last, you should include all source code required to compile and run your project.
- `README.txt`: This file should contain your name, your NetID, all known issues you have with your submitted code, and the names of anyone you have discussed the homework with. It should also include the results of your factoring (how large a number you could factor and how long it took). Also, describe any extensions you implemented.

Do not include any files ending in `.class`. We expect you to stick to Java 6 features rather than use Java 7.

All `.java` files should compile and conform to the prototypes we gave you. We write our own classes that use your classes' public methods to test your code. *Even if you do not use a method we require, you should still implement it for our use.*